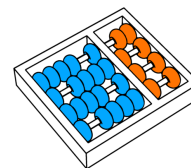


Thiago Borges Abdnur

**“ Construção e avaliação de uma solução eficiente
para comunicação entre processadores SPARCv8 ”**

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Thiago Borges Abdnur

“ Construção e avaliação de uma solução eficiente para comunicação entre processadores SPARCV8 ”

Orientador(a): **Prof. Dr. Rodolfo Jardim de Azevedo**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

ESTE EXEMPLAR CORRESPONDE À
VERSÃO FINAL DA DISSERTAÇÃO DEFEN-
DIDA POR THIAGO BORGES ABDNUR, SOB
ORIENTAÇÃO DE PROF. DR. RODOLFO
JARDIM DE AZEVEDO.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Abdnur, Thiago Borges, 1984-
Ab32c Construção e avaliação de uma solução eficiente para
comunicação entre processadores SPARCv8 / Thiago Borges
Abdnur. – Campinas, SP : [s.n.], 2012.

Orientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas,
Instituto de Computação.

1. Arquitetura de computadores. 2. Redes-em-chip. 3. Circuitos
integrados digitais. I. Azevedo, Rodolfo Jardim de, 1974-. II.
Universidade Estadual de Campinas. Instituto de Computação. III.
Título.

Informações para Biblioteca Digital

Título em inglês: Development and evaluation of an efficient solution for
SPARCv8 processors communication

Palavras-chave em inglês:

Computer architecture

Networks on a chip

Digital integrated circuits

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:

Rodolfo Jardim de Azevedo [Orientador]

Henrique Cota de Freitas

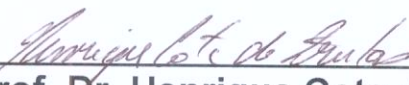
Guido Araújo

Data de defesa: 11-12-2012

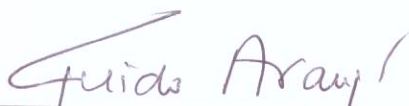
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

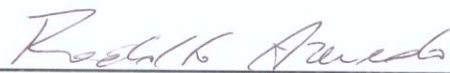
Dissertação Defendida e Aprovada em 11 de Dezembro de 2012,
pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Henrique Cota de Freitas
ICEI / PUC-Minas



Prof. Dr. Guido Costa Souza de Araújo
IC / UNICAMP



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Construção e avaliação de uma solução eficiente para comunicação entre processadores SPARCV8

Thiago Borges Abdnur¹

11 de dezembro de 2012

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Orientador)
- Prof. Dr. Henrique Cota de Freitas
- Prof. Dr. Guido Araújo
- Prof. Dr. Alexandro Baldassin (Suplente)
- Prof. Dr. Sandro Rigo (Suplente)

¹ Suporte financeiro das seguintes agências de fomento: CNPq e FAPESP (processo 2009/12853-2).

Resumo

Com a mudança da maior parte das arquiteturas convencionais para *multi-core*, a comunicação entre as diferentes unidades de processamento se torna um problema de destaque, principalmente no que tange à transferência de dados entre *cores*. Apesar do enorme impacto no desempenho, é limitado o número de trabalhos científicos que tratam sobre novas soluções para o problema, o foco mais comum é realizar a comunicação através da memória ou endereços específicos mapeados em memória. Nesta dissertação foi definido um modelo de comunicação que acrescenta três novas instruções ao conjunto de instruções do SPARCV8, permitindo que diferentes *cores* transportem dados entre si diretamente, sem a latência derivada do uso de uma memória compartilhada e de *locks*, como é o caso da atual implementação do LEON3. Avaliou-se esse modelo de comunicação através de diversos tipos de aplicações sintéticas como produtor-consumidor e *pipeline*. Para tornar o protótipo em FPGA mais realista, também foi construído um modelo de atraso para a memória principal do sistema, para que o desempenho relativo entre processador e memória fique mais próximo do real. Foi adicionado um suporte básico às novas instruções no compilador para seu uso em código C através de *asm-inline*. De forma geral, obteve-se ganhos de 3% à até 70 vezes, em termos de tempo de execução, em comparação ao uso de memória compartilhada e *locks*.

Abstract

As processors design shift towards multicore architectures, new challenges arise to increase the core to core communication efficiency. Despite the potential huge performance impact, the number of papers focusing on this problem is limited. In this project, we define a communication model, adding three new instruction to the SPARCV8 instruction set, to allow different cores to communicate directly, without the shared memory and lock latencies. We implemented the model inside the LEON3 VHDL and evaluated it using synthetic benchmarks like producer-consumer and pipeline. To make the FPGA prototype timings more realistic, we also implemented a new memory timer so that it keeps the processor-memory speed ratio closer to real values. We also created the basic compiler support for these new instructions through intrinsics, converted to inline assembly in C code. Our overall results improve the performance from 3% to up to 70 times faster.

Agradecimentos

Atribuir completamente o mérito de tudo que tenho e sou não seria suficiente para honrar aquilo que recebi da minha família durante toda a vida. Não me refiro apenas aos banais subsídios para garantir um conforto completo ou aos sacrifícios que, com toda certeza, muitos fizeram em silêncio. Não. Aqui discorro sobre coisas mais altas, coisas que estão em um outro nível. O Amor daqueles que são mais próximos é algo que agradeço todos os dias por poder apreciar. Mãe, pai, sou apenas um reflexo do trabalho excepcional que vocês fizeram. Eu amo vocês eternamente. Espero que possam sentir.

Tias e tios queridos! São todos pais e mães no meu coração. Inteligência, coração, humor, força e intensa união. Vocês me ensinaram o que é a mais pura moral e respeito. Há certas coisas que são nosso dever, e estão muito além de julgamentos ou questionamentos - e prometo honrar o que aprendi.

Seu Antônio Abrão e Dona Leontina, que agora juntos sorriem de um outro lugar, estarão sempre comigo. Um avô herói e acadêmico autodidata. Uma avó forte e apaixonada. Nunca houve nada além de carinho e lembranças maravilhosas. Estejam em paz.

Vovó Joana, a senhora divide comigo o que há de mais Sagrado, aquilo que está muito além desse mundo. Sabemos de uma mesma Verdade, sentimos e tememos ao mesmo Pai Criador. Agradeço por desde a mais tenra idade ter aprendido e compartilhado com a senhora o que é ter a verdadeira Fé. Essa é uma parte essencial de mim que é desperta nessa vida graças a sua influência e esse pacto é eterno.

O maior exemplo de hombridade e altruísmo que vivenciei partiu deixando uma saudade imensa e ainda muito a me ensinar. Vovô Asdrúbal, a lembrança de sua sabedoria está sempre em meu coração e me inspira a, algum dia, chegar ao menos próximo de onde o senhor chegou. Esse trabalho eu dedico integralmente à você.

O que seria de mim sem meus irmãos, filhos, filhas e, às vezes, esposas. Sou infinitamente rico e seguro pela certeza de ser amado e poder amar pessoas tão maravilhosas: Aninha, Auler, Bruno, Ferrugo, Fitas, Gabs, Jonas, Nicácio, Niga, Pedrin, Piga, Raôni, Ricardo, Rodrigão, eu seria completamente miserável sem a presença de vocês. Obrigado pelo insumo de vida! Sem vocês não há sorriso.

Ao meu mentor Rodolfo Azevedo, sempre presente e paciente diante a um aluno muitas vezes displicente, atribuo todas as excelentes ideias que serviram de base para a realização desse projeto. É realmente realizador ser tutorado em assuntos tão complexos por alguém tão inspirado e incontestavelmente competente. Não decepcioná-lo foi uma das principais motivações pelas quais cheguei até o fim. Você tem meu eterno respeito e gratidão.

Estar cercado por tanta competência e inspiração é uma sorte sem precedentes! Dr. Roberto Gallo e Henrique Kawakami, o trabalho só foi possível porque sempre estiveram dispostos a colaborar sem o menor questionamento, tanto com os vários equipamentos necessários quanto com o conhecimento e inspiração que transborda de vocês. Formei-me como profissional sob vossa tutela na Kryptus e tenho muito orgulho do que me tornei. Sendo amigos e dividindo a mesma paixão, sei que merecem conquistar o mundo e, independente do futuro, terei grande prazer em ajudá-los a fazê-lo.

Falo agora ao meu Irmão. Imploro que nunca me abandone nesse caminho tão incompreensível. Sua vontade e sabedoria são o que sempre me colocou novamente no eixo. Não sei para onde seremos guiados nessa exaustiva ilusão, mas se nos mantivermos na meta, simplesmente não há como haver distância. Ambos sabemos como a verdadeira distância é medida. Tente vir me buscar quando souber o que há do lado de Lá e seu eu me embriagar com o inimigo, abra novamente meus olhos. Respeitarei eternamente sua voz, pois a sua é a Dele. Meu Amigo, desejo-lhe a totalidade da felicidade do mundo, dos Mundos! Por todos os arrastes, sofrimentos, dúvidas e devaneios sou agradecido por tê-los realizado com um companheiro a quem hoje devo minha própria vida. Velho, você é a inspiração para cair fora dessa merda toda. E tamo junto mesmo que leve uma eternidade... easy-frag!

“Não há nada além Dele.”

Sumário

Resumo	vii
Abstract	viii
Agradecimentos	ix
1 Introdução	1
1.1 Contribuições deste trabalho	1
1.2 Organização	2
2 Trabalhos Relacionados	4
2.1 Arquiteturas utilizando barramentos	6
2.1.1 Caches individuais	6
2.1.2 Compartilhamento de <i>cache</i>	6
2.2 Arquiteturas híbridas (Cell)	8
2.3 Arquiteturas utilizando <i>Network-on-Chip</i>	9
2.3.1 TRIPS	10
2.3.2 SCC	11
2.3.3 Topologias Flattened Butterfly e Dragonfly	13
2.4 SPARCv8 e implementação (LEON3)	14
2.4.1 Arquitetura SPARCv8	14
2.4.2 LEON3	16
2.4.3 Infraestrutura de <i>software</i>	18
3 Implementação em <i>Hardware</i>	19
3.1 Infraestrutura geral do sistema	19
3.2 Roteador	21
3.3 Topologia e conexão entre roteadores	24
3.4 Interface com a CPU	25
3.5 Modificações no LEON3	26

3.6	Módulo de atraso da memória DDR2	29
4	Implementação em <i>Software</i> e Resultados	32
4.1	Suporte as novas instruções no compilador	32
4.2	Testes de desempenho da memória DDR2	34
4.3	Testes de desempenho da NoC	39
4.4	Testes de aplicação Produtor-Consumidor	41
4.5	Testes de aplicação em <i>Pipeline</i>	42
4.6	Testes de aplicação em <i>Ring</i>	50
5	Conclusão	51
5.1	Contribuições deste trabalho	51
5.2	Trabalhos futuros	52
	Referências Bibliográficas	53
A	Referência dos Arquivos de Projeto	57
B	Tabelas de Resultados dos Gráficos	58

Lista de Tabelas

2.1	Taxas de transferência no processador Cell.	9
2.2	Latências de acesso no SCC, onde n é o numero de <i>hops</i> até o MPB remoto ou até o controlador de memória DDR3 [21].	12
3.1	<i>Opcodes</i> das novas instruções de comunicação.	27
4.1	<i>Throughput</i> médio de leitura da DDR2 para variados ciclos de atraso. . . .	37
4.2	<i>Throughput</i> médio para cópia de vetor na DDR2 para variados ciclos de atraso após estabilização ($\geq 12\text{kB}$).	38
4.3	<i>Throughput</i> máximo da NoC para diferentes destinos e desempenho em médio em número de ciclos por <i>word</i> transportada (<i>clock</i> do sistema em 100 MHz)	41
A.1	Listagem e descrição dos arquivos de projeto.	57
B.1	Testes para aplicação em <i>pipeline</i> variando a carga. Vetor de transporte com 32 <i>words</i> . Sem atraso na memória.	58
B.2	Testes para aplicação em <i>pipeline</i> variando a carga. Vetor de transporte com 32 <i>words</i> . Atraso de 32 ciclos na memória.	59
B.3	Testes para aplicação em <i>pipeline</i> variando a carga. Vetor de transporte com 32 <i>words</i> . Atraso de 64 ciclos na memória.	59
B.4	Testes para aplicação em <i>pipeline</i> variando a carga. Vetor de transporte com 256 <i>words</i> . Sem atraso na memória.	60
B.5	Testes para aplicação em <i>pipeline</i> variando a carga. Vetor de transporte com 256 <i>words</i> . Atraso de 32 ciclos na memória.	60
B.6	Testes para aplicação em <i>pipeline</i> variando a carga. Vetor de transporte com 256 <i>words</i> . Atraso de 64 ciclos na memória.	61
B.7	Testes para aplicação em <i>pipeline</i> variando o tamanho do vetor de transporte com carga fixa de estágio de 250. Sem atraso na memória.	61
B.8	Testes para aplicação em <i>pipeline</i> variando o tamanho do vetor de transporte com carga fixa de estágio de 250. Atraso de 32 ciclos na memória. . .	62

B.9	Testes para aplicação em <i>pipeline</i> variando o tamanho do vetor de transporte com carga fixa de estágio de 250. Atraso de 64 ciclos na memória. . .	62
B.10	Testes da memória DDR2 sem atraso.	63
B.11	Testes da memória DDR2 com 16 ciclos de atraso.	64
B.12	Testes da memória DDR2 com 32 ciclos de atraso.	65
B.13	Testes da memória DDR2 com 48 ciclos de atraso.	66
B.14	Testes da memória DDR2 com 64 ciclos de atraso.	67
B.15	Testes de <i>throughput</i> da NoC sem atraso na memória.	68
B.16	Testes de <i>throughput</i> da NoC com 32 ciclos de atraso na memória.	69

Lista de Figuras

2.1	Estrutura básica de uma arquitetura SSM [19].	5
2.2	Estrutura básica de uma arquitetura DSM [19].	5
2.3	Múltiplas arquiteturas e configurações de caches	7
2.4	<i>Element interconnect bus</i> (EIB) do processador Cell [26].	8
2.5	Diagrama de blocos do chip TRIPS [17].	10
2.6	Arquitetura em alto nível de um <i>tile</i> do SCC [23].	12
2.7	Representação em alto-nível das conexões entre 64 <i>cores</i> em uma rede <i>Flat-tened Butterfly</i> [24].	13
2.8	Representação gráfica do janelamento de registradores [33].	14
2.9	Resumo dos formatos de instruções [33].	15
2.10	Exemplo de conexões ao barramento AMBA, com os diversos <i>IP cores</i> disponíveis. A imagem apresenta apenas um processador LEON3, mas até 16 processadores podem ser instanciados.	17
3.1	Representação da infraestrutura geral do sistema para a configuração 2x2.	20
3.2	Representação simplificada da arquitetura do roteador.	21
3.3	Representação do pacote de comunicação definido em <i>pack_t</i> para uma NoC 3x2.	21
3.4	Representação da máquina de estados de controle do roteador.	23
3.5	Arquitetura de alto nível completa do sistema utilizado no projeto, incluindo o módulo de atraso de memória (<i>Memory Delay</i>) e domínios de <i>clock</i>	30
4.1	Desempenho de escrita da memória DDR2 com ciclos de atraso variados.	36
4.2	Desempenho de leitura da memória DDR2 com ciclos de atraso variados.	37
4.3	Desempenho de leitura seguida de escrita da memória DDR2 com ciclos de atraso variados.	38
4.4	Desempenho da NoC para comunicação CPU0 \rightarrow CPU1 e CPU0 \rightarrow CPU3 com variação no atraso da memória DDR2.	40

4.5	Comparação entre tempos de execução utilizando a NoC e utilizando <i>pth-read.lock</i> , variando a carga dos estágios do <i>pipeline</i> e com vetor de transporte de tamanho fixo em 32 <i>words</i> (128 bytes). Nenhum atraso na memória. Ganho relativo da NoC, em porcentagem, no gráfico inferior (Tabela B.1).	46
4.6	Comparação entre tempos de execução utilizando a NoC e utilizando <i>pth-read.lock</i> , variando a carga dos estágios do <i>pipeline</i> e com vetor de transporte de tamanho fixo em 32 <i>words</i> (128 bytes). Ganho relativo da NoC, em porcentagem, nos gráficos inferiores.	47
4.7	Comparação entre tempos de execução utilizando a NoC e utilizando <i>pth-read.lock</i> , variando a carga dos estágios do <i>pipeline</i> e com vetor de transporte de tamanho fixo em 256 <i>words</i> (1kB). Ganho relativo da NoC, em porcentagem, nos gráficos inferiores.	48
4.8	Comparação entre tempos de execução utilizando a NoC e utilizando <i>pth-read.lock</i> , variando o tamanho do vetor de transporte e a carga dos estágios do <i>pipeline</i> fixa em 250. Ganho relativo da NoC, em porcentagem, nos gráficos inferiores.	49

Capítulo 1

Introdução

A presente solução para dar continuidade ao aumento da capacidade de processamento é o aumento do número de *cores* presentes no encapsulamento do processador, o que diminui a dificuldade no processo de fabricação, devido a replicação de unidades idênticas, mas introduz diversos desafios em relação a paralelização de *software* e comunicação eficiente entre as unidades de processamento.

Sendo a tecnologia CMP (*chip multiprocessor*) relativamente nova e promissora, ela é alvo de inúmeras pesquisas tanto na indústria como no meio acadêmico. Um dos vários problemas a serem pesquisados é a maneira como os *cores* se comunicam. Existem diversas soluções para diferentes arquiteturas e algumas delas serão apresentadas no Capítulo 2. Na maioria das arquiteturas *multi-core* comerciais, o nível mais baixo de interação se dá através do compartilhamento de memória *cache* L2 ou L3. Essa abordagem tem complexidade relativamente baixa mas implica em latência de acesso, dado que há uma hierarquia de memórias *cache* a considerar, inclusive níveis locais a cada *core*, até que informação seja encontrada na memória compartilhada. A própria latência de acesso a essa memória também é considerável se comparada às latências de conexões internas dos módulos que compõe os *cores* [3, 19, 36]. Algumas soluções mais robustas são implementadas na arquitetura Cell, onde há um barramento que conecta todos os processadores em anel [26], e na arquitetura TRIPS, que utiliza conceitos modernos de NoC (*Network-on-Chip*) para interconexão dos elementos de processamento [17, 30]. Esses três exemplos cobrem os três grandes modelos de conexão utilizados atualmente em arquiteturas *multi-core*.

1.1 Contribuições deste trabalho

Nas arquiteturas estudadas, o mecanismo de comunicação é feito através de periféricos mapeados em memória ou indiretamente em espaços de endereçamento específicos, introduzindo um *overhead* considerável para o controle e transmissão dos pacotes nas NoCs, o que

difículta essas abordagens para sistemas com poucos *cores*, como é o caso do LEON3 [30]. Baseando-se na comparação entre as soluções citadas anteriormente, este trabalho propõe um modelo de comunicação que se adapte à implementação GPL [13] em VHDL disponível do padrão SPARCv8 [33]. Este projeto visa ampliar o conjunto de instruções do padrão e modificar a implementação para que vários processadores possam ser instanciados e funcionem em sinergia, transportando dados entre si de maneira eficiente. Sendo assim, esse novo modelo foi desenvolvido de tal forma que instruções explícitas fossem disponibilizadas para enviar palavras de um processador a outro, minimizando o *overhead* com uma arquitetura de comunicação simplificada, o que é um diferencial desta implementação.

Além da implementação em *hardware*, tem-se como objetivo uma avaliação do mecanismo através de aplicações sintéticas, obtendo dados sobre o desempenho do modelo proposto, envolvendo comparações com o uso de vetores compartilhados na memória principal do sistema, que exigem algum mecanismo de controle de concorrência, no caso, *pthread_locks*.

Uma das principais premissas para o desenvolvimento dessa extensão é a de gerar código VHDL sintetizável para possibilitar que todos os testes sejam executados em *hardware* (FPGA), deixando clara a viabilidade do projeto.

O foco do trabalho reside na nova abordagem em se expor diretamente no conjunto de instruções, ou seja, à camada de *software*, o mecanismo de comunicação entre *cores* e não na arquitetura da NoC em si. Por isso optou-se por uma implementação simples da topologia de rede, com a menor latência de controle possível, focando-se na modificação do *pipeline* do processador para a adição das novas instruções e na avaliação de desempenho.

1.2 Organização

O trabalho foi estruturado da seguinte forma:

- No Capítulo 2 é apresentado um estudo de diversas arquiteturas e topologias para comunicação entre processadores, assim como as características principais do processador escolhido para a modificação, o LEON3.
- No Capítulo 3 é detalhado todo o esforço de desenvolvimento em *hardware* (VHDL) para implementar o mecanismo, incluindo os módulos de *hardware* adicionados e a alteração no *pipeline* do LEON3. Também são descritas as novas instruções incluídas no processador.
- No Capítulo 4 é apresentada a avaliação do sistema através de aplicações sintéticas, assim como todo o desenvolvimento em *software* necessário para dar suporte as novas instruções.

- No Capítulo 5 são apresentadas as conclusões e possibilidades de trabalhos futuros baseados nessa dissertação.

Capítulo 2

Trabalhos Relacionados

A melhora dos processos de fabricação de circuitos integrados nos últimos anos possibilitou a inserção de um número muito maior de componentes num mesmo encapsulamento, que antes eram integrados através de conexões em placas. O reuso de componentes também se tornou essencial para o rápido desenvolvimento de novos produtos, pois implicam em confiabilidade, já que foram testados e utilizados previamente, e baixo custo. Esses elementos agora podem ser conectados por arquiteturas de comunicação *on-chip*, sendo o conjunto denominado SoC (*System-on-Chip*), que devem ser flexíveis o suficiente para alcançar diversas restrições de projeto, tais como custo, desempenho, potência/consumo e confiabilidade [30]. Um dos usos de SoC em voga são as arquiteturas *multi-core* denominadas MPSoC (*Multiprocessor System-on-Chip*), onde se deseja que diversos núcleos idênticos trabalhem em conjunto de forma coerente, com o mínimo de alterações possível em suas estruturas.

Focamos o estudo nas arquiteturas paralelas MIMD (*Multiple Instruction streams, Multiple Sata streams*), onde cada processador carrega suas próprias instruções e opera em seus próprios dados. Essas máquinas exploram TLP (*Thread-Level Parallelism*), que é o caso do processador LEON3, utilizado na pesquisa como estudo de caso para avaliação. Podemos separar essas arquiteturas em duas grandes classes [19]:

Symmetric shared-memory (SSM): Arquiteturas que compartilham uma única memória centralizada, contento um ou mais níveis de *caches* privadas. Essa arquitetura escala bem para um número reduzido de processadores. Como o tempo de acesso à memória por qualquer processador é uniforme, esse estilo de arquitetura também é chamada UMA (*Uniform Memory Access*). A Figura 2.1 ilustra a descrição.

Distributed shared-memory (DSM): Essas arquiteturas apresentam memórias fisicamente distribuídas, ilustradas na Figura 2.2. São bem mais robustas no que se

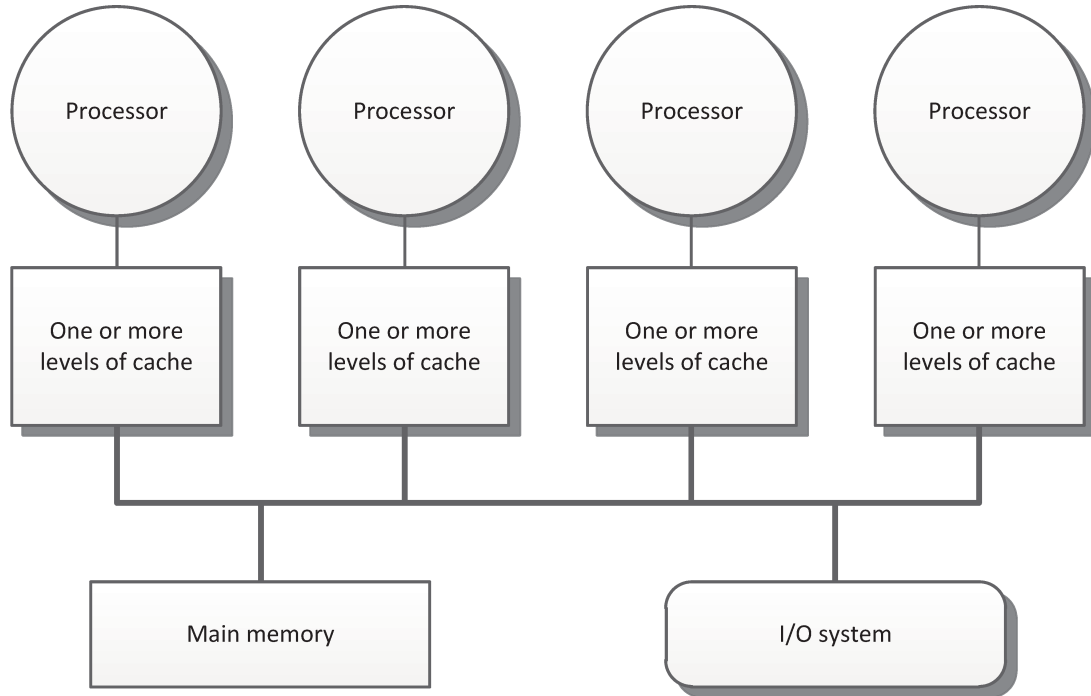


Figura 2.1: Estrutura básica de uma arquitetura SSM [19].

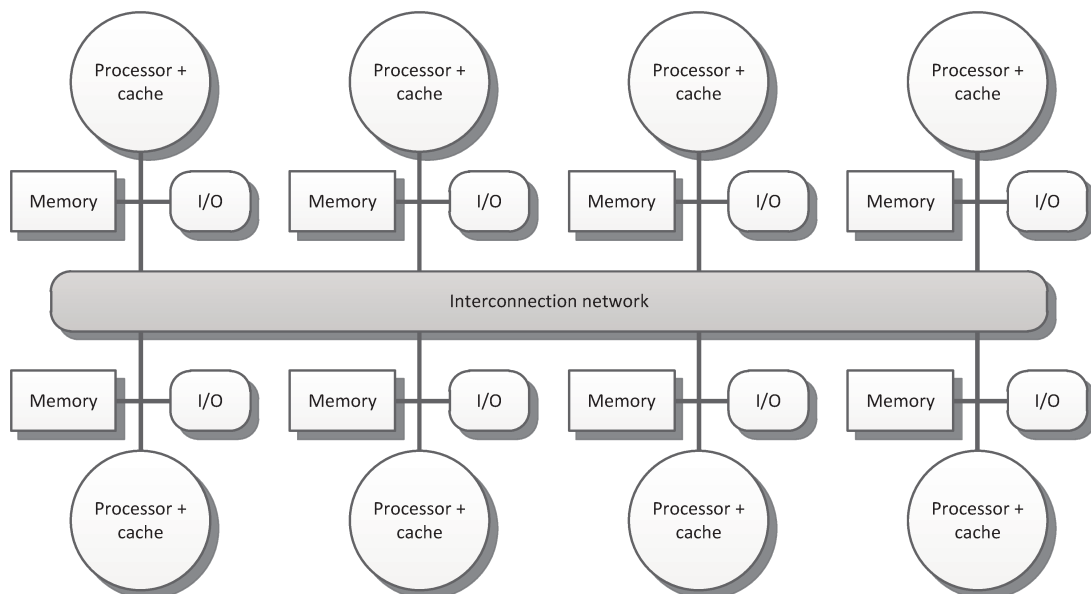


Figura 2.2: Estrutura básica de uma arquitetura DSM [19].

refere a quantidade de processadores, devido aos benefícios que uma rede de interconexão proporciona.

Arquiteturas SSM têm como principal característica a conexão dos vários núcleos de processamento e periféricos através de um barramento, onde o acesso pelos processadores à memória tem uma latência constante, modelo chamado UMA (*Uniform Memory Access*). Em arquiteturas DSM, essa latência pode variar pois as memórias podem ser diferentes e a distância entre o processador e a memória acessada tem grande influência na latência de acesso. Esse modelo é conhecido como NUMA (*Non-Uniform Memory Access*). Nesse caso, os elementos são conectados por algo mais sofisticado, como uma *Network-on-Chip*.

2.1 Arquiteturas utilizando barramentos

Com relação aos processadores, podemos dividir as arquiteturas baseadas em barramento em duas classes, onde na primeira se enquadram processadores com *caches* individuais, e na segunda processadores com hierarquias de *cache* compartilhadas.

2.1.1 Caches individuais

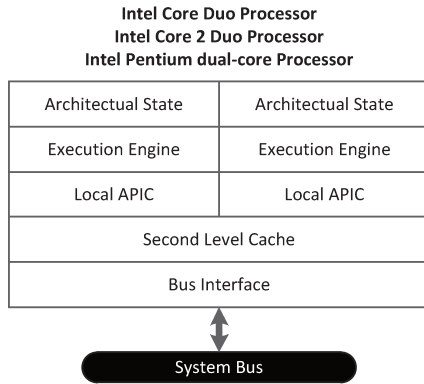
Nesse primeiro caso, cada um dos processadores está ligado diretamente ao barramento e possui uma *cache* para si, que não é compartilhada com outros *cores*. Sendo assim, todo tráfego de dados recai sobre o barramento, tanto o acesso a memória e periféricos quanto as transferências geradas pelo protocolo de coerência das *caches* em diferentes processadores. Como apenas um mestre pode utilizar o barramento de cada vez, ele se torna rapidamente inviável ao se aumentar o número de *cores*.

Como esse é o caso do processador LEON3 que implementa o barramento AMBA [2], utilizado no trabalho, a Seção 2.4 tratará exclusivamente de mais detalhes das consequências da utilização dessa arquitetura de conexão.

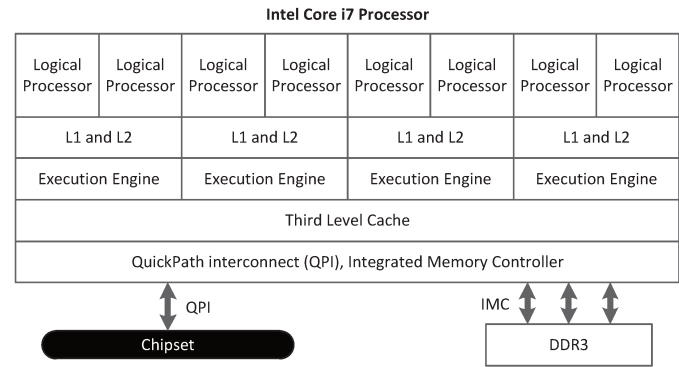
2.1.2 Compartilhamento de *cache*

A conexão por barramento foi utilizada para os primeiros processadores *multi-core* de propósito geral introduzidos no mercado. A IBM foi a primeira a lançar, em 2000, o processador POWER4 com dois *cores* [35]. Foi seguida pela AMD e Intel, que lançaram processadores de dois núcleos em 2005 e, em 2006, a Sun introduziu o T1 [19]. Estes processadores suportam *caching* tanto de dados privados como de dados compartilhados. Uma vantagem muito interessante dessa abordagem é a de que o transporte de dados entre os núcleos se dá por instruções comuns de **LOAD** e **STORE** em faixas de endereço

compartilhadas [1]. Como exemplo, temos os processadores Intel Core2 Duo e Core i7 (Figuras 2.3(a) e 2.3(b), respectivamente).



(a) Diagrama de blocos dos processadores Intel *dual core* [20]



(b) Diagrama de blocos dos processadores Intel *Core i7* [20]

Figura 2.3: Múltiplas arquiteturas e configurações de caches

Observa-se que há *caches* compartilhadas entre dois ou mais *cores*. Dado que existe uma hierarquia de memórias *cache* [4, 20], surgem dois aspectos importantes a serem tratados: coerência e consistência da memória *cache*. Por coerência, entende-se: quais valores podem ser retornados por uma leitura. A consistência determina quando um valor escrito será retornado devido a uma leitura [19]. Protocolos de coerência e consistência de *cache* são assuntos extensos e estão além do escopo do projeto. Estudos aprofundados podem ser encontrados em [1, 11, 19, 29, 34].

O compartilhamento de memória também tem consequências. O desempenho do sistema estará altamente limitado pela taxa de transferência das *caches* compartilhadas e da qualidade de implementação dos algoritmos de coerência de *cache* implementados em *hardware*.

A escalabilidade de *caches* compartilhadas é limitada, já que o aumento no número de processadores implica na criação de mais camadas na hierarquia de *cache*, maior complexidade no algoritmo para manter a coerência entre elas e, conseqüentemente, uma maior área de silício ocupada. Também começam a surgir gargalos nos níveis da hierarquia mais afastados dos processadores [4], onde a *cache* L2 pode, em algumas situações até piorar o desempenho [12].

É válido ressaltar que implementação de *caches* compartilhadas não é um mecanismo de comunicação propriamente dito, apenas evita acessos à memória principal enquanto os dados cabem nestas. Em aplicações cujo uso de memória é extensivo ou de forma muito esparsa, o gargalo de comunicação passa a ser a conexão com a memória principal do sistema.

Até a família *Core2*, a conexão dos *cores* com periféricos e com a memória era feita através do FSB (*Front Side Bus*). A primeira abordagem que surgiu na tentativa de melhorar a interconexão foi o *HyperTransport* da AMD, introduzido em 2001 [5]. Em 2008, uma nova implementação de infra-estrutura de conexão surgiu, o *QuickPath Interconnection*, que provê uma conexão direta entre processadores e memória. Baseado em conexões ponto-a-ponto, elimina-se a necessidade da abstração de mestre/escravo ou de árbitros de barramento, provendo agora transmissão de dados através de pacotes encapsulados em várias camadas, algo muito mais próximo de uma NoC (*Network-on-Chip*) que de um barramento [9]. Dentre as vantagens dessa nova abordagem, encontram-se a diminuição no número de fios necessários para a interconexão, um grande aumento na taxa de transferência e melhora no escalonamento e tratamento de concorrência para utilização de recursos [9].

Um exemplo mais elaborado de comunicação é implementado no processador Cell da IBM (Figura 2.4), que pode-se considerar como um híbrido barramento/NoC.

2.2 Arquiteturas híbridas (Cell)

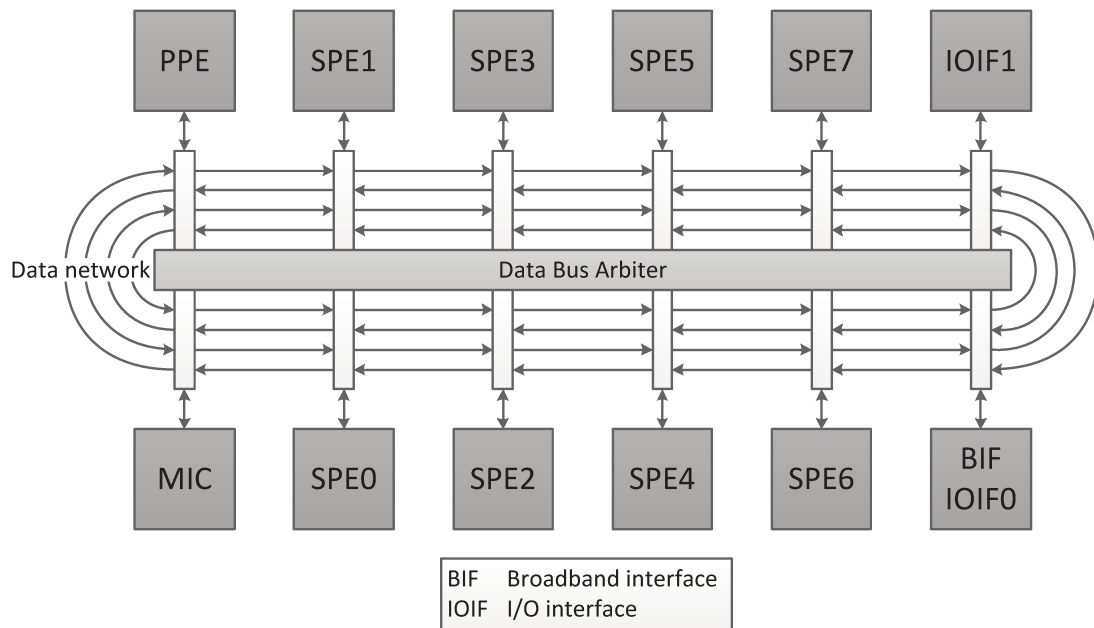


Figura 2.4: *Element interconnect bus* (EIB) do processador Cell [26].

Esse processador consiste em um *power processor element* (PPE) funcionando a 3.2 GHz, oito co-processadores especializados chamados *synergistic processor elements* (SPEs), um controlador de memória de alta velocidade e uma interface de barramento de banda

larga. O PPE e os SPEs se comunicam através de um barramento interno chamado *element interconnect bus* (EIB), funcionando a 1.6 GHz. Cada SPE contém uma memória de armazenamento local de 256-kbyte para instruções e dados e não têm *cache*. Os SPEs não têm acesso direto a memória principal, mas podem requisitar transferências através de DMA. Operações de DMA podem transferir dados entre as memórias locais aos SPEs e qualquer elemento conectado ao EIB. Sinais ou *mailboxes* podem ser usados para comunicação simples de baixa latência. Isso permite transferências diretas entre dois SPEs que podem trabalhar em *pipeline* ou no esquema produtor/consumidor sem burocracia [26], ou seja, sem ter a necessidade de um processador intermediário para executar as operações de comunicação. Apesar de não apresentar roteadores, a transferência direta de dados e a passagem de mensagens entre os SPEs é o que considera-se como característica híbrida.

A rede de comunicação consiste em quatro anéis de 16 bytes de largura cada, com dois indo para o sentido horário e dois no anti-horário, garantindo que qualquer transferência não atravessará mais do que a metade do anel [26]. Como as memórias locais dos SPEs são mapeadas no espaço de endereços, o PPE pode acessá-los através de simples **LOADs** e **STOREs**, apesar de serem bem menos eficientes que transferências por DMA.

Essa sofisticação leva a bons resultados em termos de taxas de transferência, alcançando até 204,8 Gbytes/s, conforme mostrado na Tabela 2.1.

Direção	Taxa de pico no EIB
Transferências internas ¹	204,8 Gbytes/s
MIC \leftrightarrow Memória principal	25.6 Gbytes/s
Controlador de E/S (entrada)	25 Gbytes/s
Controlador de E/S (saída)	35 Gbytes/s

Tabela 2.1: Taxas de transferência no processador Cell.

Essa variedade de mecanismos de comunicação possibilita a aplicação de diversos modelos de programação paralela, tais como *function-offload*, *device-extension*, *streaming-shared-memory-multiprocessor*, *asymetric-thread-runtime* [26].

2.3 Arquiteturas utilizando *Network-on-Chip*

A solução mais moderna para a interconexão entre elementos de um SoC se dá através da implementação de NoCs, que são altamente escaláveis e viabilizam a conexão de um

¹Entre quaisquer elementos ligados diretamente ao barramento (SPE \leftrightarrow SPE, PPE \leftrightarrow SPE, SPE \leftrightarrow controlador de memória, etc).

de rede (blocos N) contendo um roteador e tabelas de tradução de endereço do sistema, formando um aglomerado 4x10. Ainda se observa dois controladores de DMA, dois controladores de SDRAM (SDCs), um controlador de barramento externo (EBC) e o controlador chip-a-chip (C2C), que permite a conexão de vários chips para compor um sistema maior. Os blocos D e I representam *caches* L1 para dados e instruções, respectivamente. Canais virtuais são fechados entre dois elementos, numa abstração similar a *sockets*, para o envio de dados [17].

Há ainda uma segunda rede interconectando os *cores*, chamada OPN (*operand network*), que substitui os bancos de registradores tradicionais e cuida do transporte de operandos entre as unidades de execução. Mensagens de tamanho fixo em 138 bits são transportadas com uma latência de um ciclo por nó. Sua integração ao processador cria um canal direto de dados entre as ULAs (Unidades de Logica e Aritmética) de unidades de execução adjacentes [32].

Ambas as redes tem topologia *mesh* 2D, sendo a OCN 4x10 e a OPN 5x5.

É necessário deixar claro que não há um mecanismo explícito para o uso das redes. Utilizando um modelo de execução em blocos de instrução, gerados previamente pelo compilador, o transporte dos operandos gerados por uma dada unidade de execução, assim como os dados a serem escritos na memória, se dão de forma transparente pelas redes OPN e OCN, respectivamente [32].

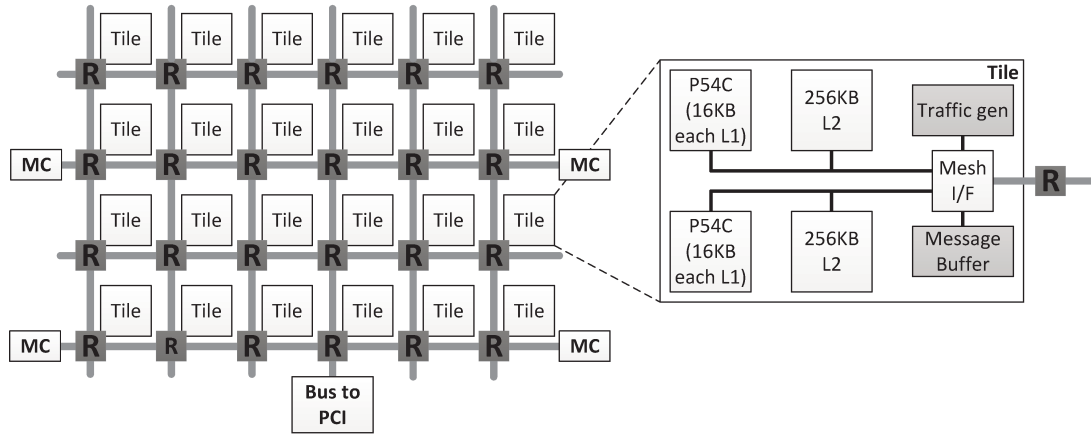
Análises de desempenho [17] mostram que a OCN garante baixas latências em requisições de dados e taxa de transferência de pico de 74 Gbytes/s, executando à 500 MHz.

2.3.2 SCC

A chamada Plataforma SCC (*Single-chip Cloud Computer*) apresenta 24 *tiles*, cada um com 2 *cores* P54C (microarquitetura Intel P5 32 bits) onde cada *core* conta com 16kB de *cache* L1 para dados e mais 16kB para instruções. Externamente à cada *core* também há uma *cache* L2 de 256kB [23]. A Figura 2.6 ilustra esses detalhes.

Esses processadores são conectados através de uma rede *mesh* 2D, onde cada *tile* contendo dois *cores* Pentium segunda geração (P54C) se conecta a um roteador, que faz o processo de empacotamento dos dados transferidos. Cada *tile* conta com um *message passing buffer* (MPB) de 16kB, um espaço de memória não-cacheável de 384kB compartilhado entre todos os processadores, onde dados podem ser enviados diretamente entre os *tiles* com simples escritas aos espaços de endereços correspondentes, como descrito em [22]. Não há protocolo de coerência de *cache* entre cores em diferentes *tiles*, é necessário que o próprio programador garanta a coerência, apesar de haver facilidades na API RCCE [23].

A arquitetura disponibiliza uma API chamada RCCE (pronunciada “*rocky*”) que provê

Figura 2.6: Arquitetura em alto nível de um *tile* do SCC [23].

uma série de funções para a passagem de mensagens, sincronização, gerenciamento de memória e gerenciamento de energia. Apesar de expor explicitamente o mecanismo de comunicação, há um *overhead* considerável na execução destas funções devido ao desvio de fluxo, interpretação de argumentos e construção de pacotes feitos pelo próprio *core* [22].

Tipo de acesso	Latência aproximada para leitura de uma linha de <i>cache</i>
<i>Cache</i> L2	18 ciclos
MPB local com <i>bypass</i>	15 ciclos
MPB local sem <i>bypass</i>	53 ciclos
MPB remoto	$45 + 4 \cdot n \cdot 2$ ciclos
Memória externa (DDR3)	$45 + 4 \cdot n \cdot 2 + 30$ ciclos + 16 ciclos (DDR3)

Tabela 2.2: Latências de acesso no SCC, onde n é o numero de *hops* até o MPB remoto ou até o controlador de memória DDR3 [21].

As latências de acesso são consideráveis em termos de ciclos, chegando a mais de 91 ciclos para acesso a memória externa, como mostrado na Tabela 2.2 [21]. O protocolo de roteamento utilizado é o XY, onde os pacotes transitam horizontalmente (em X) nos roteadores, até chegar a coluna do destino, e então trafegam verticalmente (em Y) até o *tile* de destino, o que garante a ordem de entrega e fixa os nós pelos quais o pacote trafega até chegar ao destino [21]. Essa simplificação reduz consideravelmente a complexidade dos roteadores e dos sinais necessários para controle de tráfego, caracterizando um roteamento determinístico e garantindo a ausência de *deadlocks*. Uma desvantagem dessa abordagem é o fato de que, como os caminhos de pacotes são fixos, pode haver um gargalo no tráfego

caso um *core* já esteja usando uma parte do caminho necessário para outro.

2.3.3 Topologias Flattened Butterfly e Dragonfly

Considerada uma das topologias de interconexão mais eficientes em termos de custo e resultado, a *Flattened Butterfly* é escalável para um número grande de núcleos (64 à 128 em [24]). Como mostrado na Figura 2.7, há um roteador para 4 *cores*, onde cada um dos roteadores está conectado diretamente a todos os outros roteadores que estão na mesma linha e coluna que ele. Sendo assim, a distância máxima entre quaisquer um deles será dois nós. Isso é implementado através de *bypass channels*, *muxes* controlados pelo algoritmo de roteamento que permitem que o sinal passe diretamente através roteador até o próximo [24]. Assim como na arquitetura TRIPS, essa interconexão também provê a criação de canais virtuais entre os nós [24].

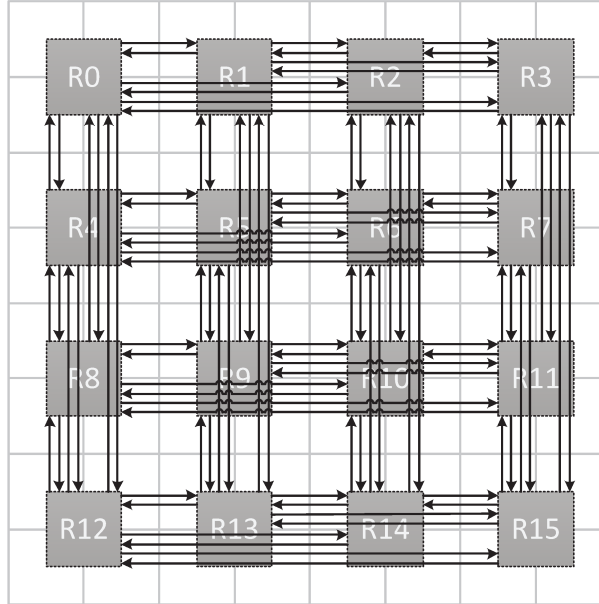


Figura 2.7: Representação em alto-nível das conexões entre 64 *cores* em uma rede *Flattened Butterfly* [24].

Resultados apresentados em [24] mostram uma melhora de até 50% no *throughput* em relação às conexões em *mesh* comuns.

Uma versão dessa topologia escalável para uma quantidade ainda maior de nós é apresentada em [25] com nome de *Dragonfly*.

Estudos mais gerais sobre *NoCs*, conceitos, comparações e tendências, foram feitos em [14], onde um grande número de trabalhos científicos sobre topologias de interconexões são citados, servindo de base de referências.

correspondentes à janela do contexto atual, sendo 8 locais, 8 denominados *in* e 8 denominados *out*. O uso da instrução **SAVE** provoca a mudança de janela, o que faz com que os registradores de *out* da janela anterior se tornem os de *in* da atual. Registradores locais não são propagados entre janelas, mas podem ser modificados sem comprometer os valores da janela anterior. Isso é representado na Figura 2.8. O número de janelas é variável e definido na implementação. A instrução **RESTORE** restaura os valores da janela anterior. Caso não haja janelas suficientes durante a execução de um **SAVE**, uma interrupção (*trap*) é gerada para que o conteúdo de todas as janelas seja salvo na pilha [33].

A vantagem principal desse modelo se dá na passagem de argumentos entre funções, não sendo necessário a escrita desses na pilha e, conseqüentemente, evita-se acessos à memória externa, que são muito mais lentos [33].

Com instruções de tamanho fixo de 32 bits, a arquitetura implementa 3 formatos de instrução:

Formato 1: Instrução **CALL** (desvio de fluxo absoluto).

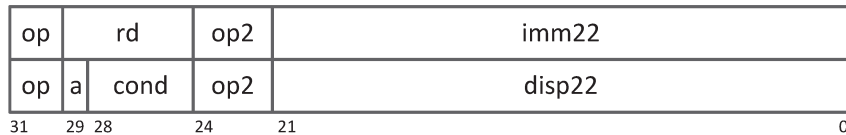
Formato 2: Instrução **SETHI** e *Branches* (desvios de fluxo relativos ao PC (*Program Counter*)).

Formato 3: Instruções de acesso à memória, lógicas, aritméticas e outras.

Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



Format 3 (*op* = 2 or 3): Remaining instructions



Figura 2.9: Resumo dos formatos de instruções [33].

A distribuição dos bits em cada formato é apresentado na Figura 2.9. Detalhes de cada campo de bits podem ser encontrados em [33] e serão apresentados à medida que forem necessários para a compreensão do texto [33].

Traps definidos na arquitetura permitem desvios de fluxo de execução para funções de 4 instruções definidas numa tabela onde cada entrada é chamada de *trap handler*. No contexto deste trabalho, utiliza-se os termos *traps* e interrupções como sinônimos. São suportadas interrupções de *hardware*, geradas por periféricos externos ao processador ou para sinalizar certas condições de execução excepcionais do próprio processador (divisão por zero, falha de acesso à memória, etc.), e interrupções de *software*, geradas pela instrução TA. Entradas na tabela de interrupção têm diferentes prioridades, o que permite controlar a ordem de execução quando mais de uma interrupção é gerada simultaneamente [33].

2.4.2 LEON3

Desenvolvido inicialmente pela Agência Espacial Europeia (ESA) e atualmente pela empresa Aeroflex Gaisler [15], o processador LEON3 é uma compatível com a arquitetura SPARCV8, desenvolvida para aplicações embarcadas de alto desempenho com baixa complexidade e baixo consumo. O código descrevendo todo o sistema, escrito em VHDL, está disponível sob a licença GPL [13] para uso em pesquisa científica ou uso não comercial.

O LEON3 apresenta um *pipeline* em 7 estágios (arquitetura Harvard [19]), *caches* separadas para instruções e dados, multiplicador e divisor e *hardware*, suporte a depuração e extensão para múltiplos processadores [16].

Os 7 estágios do *pipeline* implementados são:

Fetch: Leitura da próxima instrução apontada pelo PC.

Decode: Decodificação da instrução.

Register Access: Acesso ao conteúdo dos registradores apontados pelo decodificador.

Execute: Execução da instrução (ULA, multiplicador, divisor)

Memory: Acesso a conteúdo de memória.

Exception: Resolução e checagem de interrupções e exceções.

Write-back: Escrita nos registradores de destino.

Para evitar bolhas no *pipeline* de execução, é implementado *register-forwarding* entre os estágios [19].

Disponibilizando *caches* altamente configuráveis, é possível adaptar o processador a diversos usos. Apenas uma *cache* L1 é conectada a cada processador, existindo dentro dela espaços separados para dados e instruções, e não existe *cache* L2. É possível instanciar *caches direct-mapped* ou *multi-set* [19] com variações tanto no seu tamanho total

como na largura das linhas. Uma de três políticas de substituição podem ser selecionadas: LRU (*Least Recently Used*), LRR (*Least Recently Replaced*) ou (pseudo-) aleatória. O comportamento da *cache* de dados é sempre *write-through*, o que implica em acessos constantes ao barramento e, conseqüentemente, à memória externa. Também é possível ativar o *snooping* na *cache* de dados, que opera observando acessos de escrita a endereços, cujo conteúdo já está armazenado, por outros mestres e invalidando as linhas correspondentes [16,19].

A *cache* é conectada como mestre no barramento AMBA AHB 2.0 [2,30], que é um barramento multiplexado de alto desempenho que suporta transações, simples ou em rajadas (*bursts*), alinhadas em 8, 16 e 32 bits. Suportando um máximo de 16 mestres e 16 escravos, o árbitro AHB escalona os diferentes mestres que requisitam acesso através da política de *round-robin*.

Periféricos mais lentos, como UART e *timers* são ligados ao barramento APB, que por sua vez é conectado ao AHB através de um *bridge* AHB-APB mas não é utilizado nesse projeto.

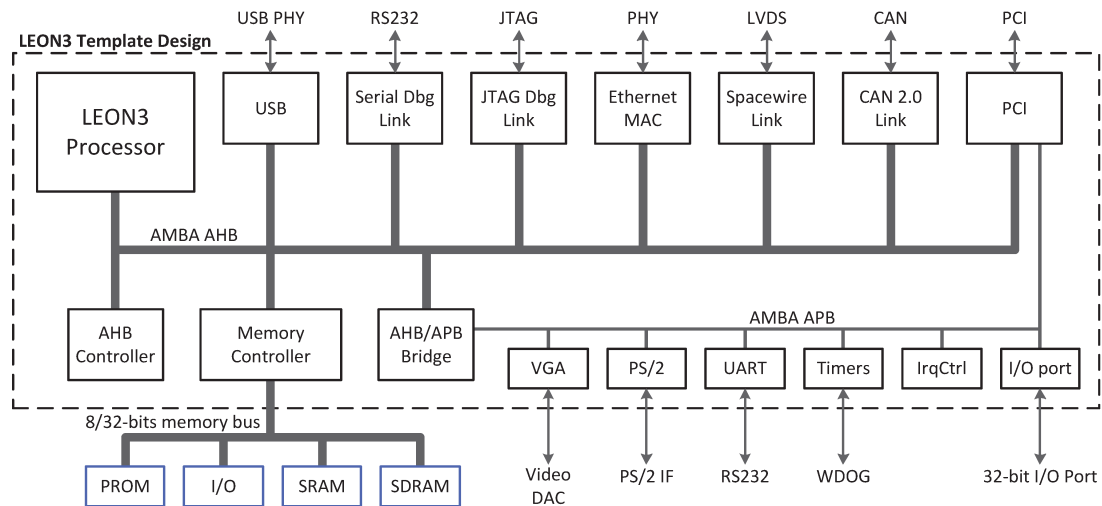


Figura 2.10: Exemplo de conexões ao barramento AMBA, com os diversos *IP cores* disponíveis. A imagem apresenta apenas um processador LEON3, mas até 16 processadores podem ser instanciados.

Dentre os diversos controladores disponibilizados na biblioteca de IPs da Gaisler, que acompanha o LEON3, são utilizados o controlador de memória DDR2, para utilização como memória principal do sistema, controlador da porta *ethernet*, para comunicação com o sistema operacional, e o DSU (*Debug Support Unit*), que é conectado como escravo no barramento AHB e também conectado diretamente aos processadores, permitindo inspeção e controle completo sobre os mesmos, listando valores de registradores, *cache*, carregamento de programas na memória, inclusão de *break-points* para interrupção da

execução entre outros [16]. A Figura 2.10 mostra um exemplo de sistema em alto-nível que pode ser instanciado utilizando a biblioteca e suas conexões com o barramento AMBA.

2.4.3 Infraestrutura de *software*

O suporte a compilação para o LEON3 conta com duas versões do compilador GCC. O primeiro chamado BCC utilizado para aplicações *bare-metal*, onde não há sistema operacional e apenas algumas das funções da biblioteca padrão (NewLib) são suportadas. Não há suporte, por exemplo, a sistema de arquivos e o suporte a *threads* é muito limitado [15].

A segunda versão, gerada através do Buildroot, permite a compilação do sistema operacional Linux aplicando uma série de alterações no *kernel* para suporte aos periféricos específicos do LEON3 [27].

A utilização do sistema operacional facilita a execução dos testes e armazenamento de resultados e essa foi a opção adotada no projeto.

Capítulo 3

Implementação em *Hardware*

Este capítulo apresenta o modelo de comunicação implementado no LEON3 para avaliar a possibilidade de expor um mecanismo de comunicação diretamente para a camada de aplicação através de novas instruções.

É importante ressaltar que a solução apresentada, de forma geral, é independente do processador utilizado, com exceção das modificações no próprio VHDL do processador LEON3, apresentadas na Seção 3.5. Sendo assim, toda a infraestrutura de comunicação pode ser reutilizada e testada em outras arquiteturas com baixo esforço de implementação.

3.1 Infraestrutura geral do sistema

As principais premissas utilizadas, na construção da solução, são:

Transmissão direta de dados entre *cores*: Deve-se ter a possibilidade de enviar dados diretamente de um processador a outro, sem a utilização do barramento para tal. Sendo assim, uma NoC atende o propósito melhor que um barramento, já que é mais escalável.

Código Sintetizável: A implementação deve ser sintetizável para possibilitar testes em *hardware* (FPGA) [28].

Minimizar Latência: Também desejamos minimizar a latência gerada por cabeçalhos de pacotes e algoritmos de roteamento, pois o objetivo é avaliar a exposição das instruções e não as capacidades da NoC em si. Das estruturas estudadas, a mais simples é uma rede com topologia *mesh-2D* utilizando o algoritmo de roteamento XY [18, 37], detalhados nas Seções 3.3 e 3.2, respectivamente. Isso não impede que outras redes possam ser adequadas ao sistema com relativa facilidade.

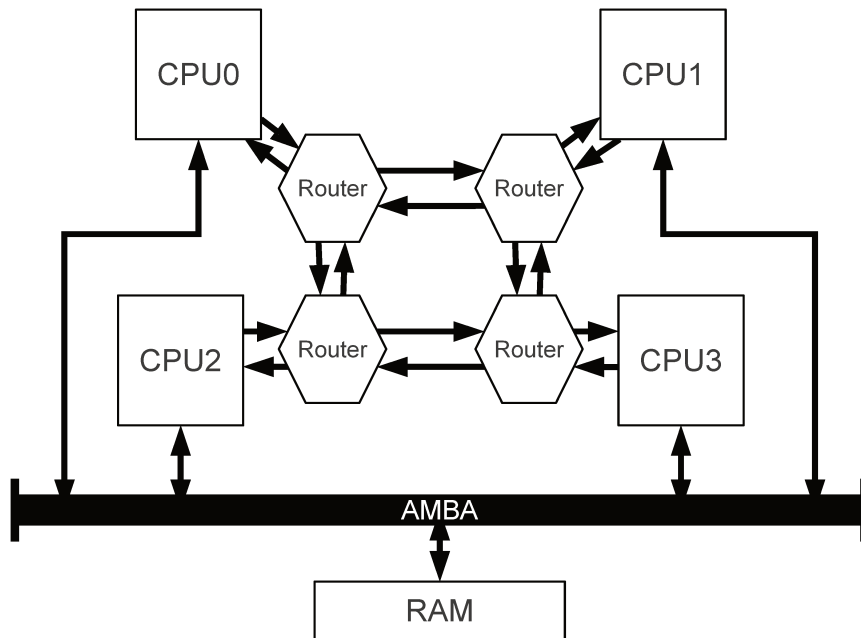


Figura 3.1: Representação da infraestrutura geral do sistema para a configuração 2x2.

Para tal, conecta-se cada processador a um roteador, detalhado na Seção 3.4, que, por sua vez, pode ser conectado à até quatro outros, nas direções norte, sul, leste e oeste. A Figura 3.1 mostra a arquitetura geral do sistema após a adição dos roteadores e suas conexões.

Para a utilização do mecanismo, determinou-se que três instruções devem ser adicionadas ao processador, uma para envio de uma palavra, à qual deu-se o nome de **SEND**, outra para recebimento de uma palavra, nomeada **RECV** e uma terceira para controle do roteador, chamada **NCTRL**. Os detalhes serão discutidos na Seção 3.5.

Esses são elementos suficientes para avaliar as possíveis melhoras propostas no projeto no que diz respeito a tempo de acesso a dados compartilhados. Visa-se ganho em relação ao uso *locks* e, conseqüentemente, à latência de acesso ao barramento e à memória externa.

Dividimos o desenvolvimento em quatro partes: roteador (Seção 3.2), conexão entre roteadores (Seção 3.3), interface com a CPU (Seção 3.4) e modificações no *pipeline* do LEON3 para adição de instruções (Seção 3.5). As seções a seguir tratam de cada uma dessas partes com detalhes.

3.2 Roteador

Cada um dos roteadores instanciados possui um módulo de interface com a CPU (detalhada na Seção 3.4) e até quatro portas de comunicação com outros roteadores, nas quatro direções: norte, sul, leste e oeste (referenciadas por N, S, E, W, respectivamente).

Esses canais consistem, basicamente, em duas FIFOs (*First-In First-Out*), uma para entrada de pacotes e outra para saída. A sua profundidade pode ser controlada através de uma constante denominada `ROUTER_FIFO_SIZE` definida no pacote principal *router_pkg.vhd* (consultar Apêndice A). Por padrão esse valor é 8, o que significa que cada FIFO comporta até 8 pacotes. Uma visão geral da arquitetura do roteador é apresentada na Figura 3.2.

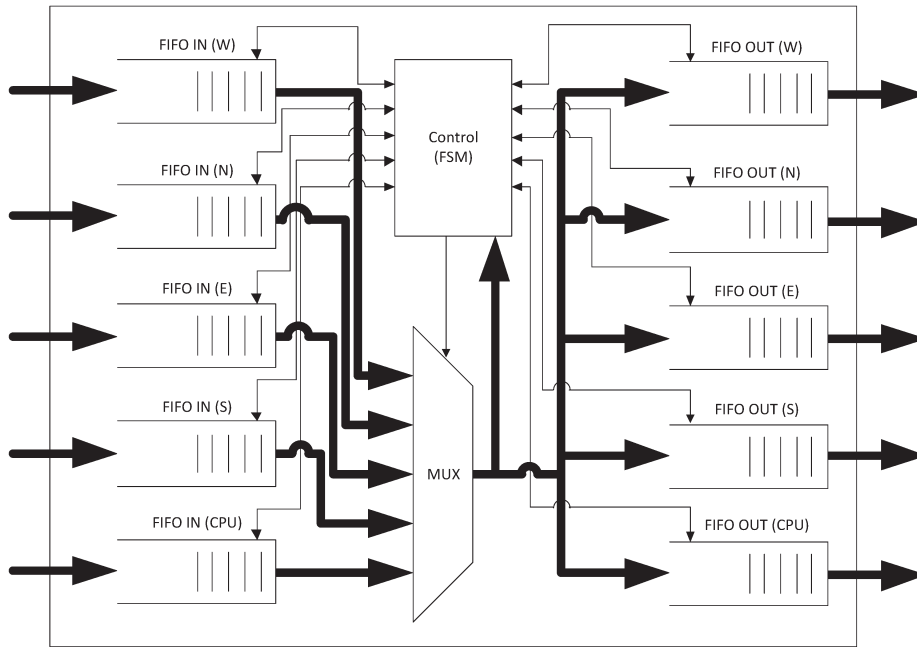


Figura 3.2: Representação simplificada da arquitetura do roteador.

NoC Package format (*pack_t*):

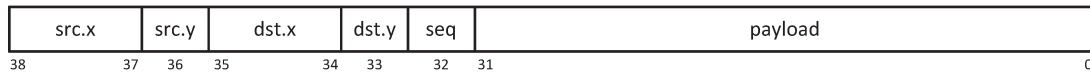


Figura 3.3: Representação do pacote de comunicação definido em *pack_t* para uma NoC 3x2.

Cada pacote, ilustrado na Figura 3.3 e definido no tipo *pack_t*, é composto pela coordenada de origem, coordenada de destino, bit de sequência e *payload*, um vetor de 32 bits,

por padrão, que pode ser modificado através da constante `PAYLOAD_SIZE`. O comprimento das coordenadas varia de acordo com as dimensões da topologia selecionada. Isso garante a minimização na quantidade de fios conectando um roteador a outro. O total de bits do pacote pode ser calculado através da equação:

$$2\lceil\log_2(\text{NOC_DIM_X})\rceil + 2\lceil\log_2(\text{NOC_DIM_Y})\rceil + 1 + \text{PAYLOAD_SIZE} \quad (3.1)$$

onde:

- `NOC_DIM_X` e `NOC_DIM_Y` referem-se as dimensões X e Y da topologia selecionada, respectivamente.
- `PAYLOAD_SIZE` refere-se a quantidade de bits de dados transportados por pacote, no caso, uma palavra de 32 bits.

Sendo assim, supondo uma NoC 3x2, tem-se pacotes de $2\lceil\log_2(3)\rceil + 2\lceil\log_2(2)\rceil + 1 + 32 = 39$ bits. Detalhes sobre o bit de sequência ficarão mais claros a seguir, na descrição da máquina de estados de controle do roteador.

Cada uma das portas N, S, E e W é composta por um conjunto paralelo de sinais de entrada e outro de saída, ambos do mesmo tipo *link_t*, representados pelas setas pretas entre os roteadores na Figura 3.1. Esse tipo consiste no pacote descrito anteriormente com a adição de mais dois sinais de controle: *ready*, para indicar que há um pacote disponível na origem e *get*, indicando que o destino está pronto para receber o pacote e este já pode ser descartado na origem no próximo ciclo. Sendo assim, o cálculo da largura total para a conexão entre dois roteadores é:

$$2(2 + 2\lceil\log_2(\text{NOC_DIM_X})\rceil + 2\lceil\log_2(\text{NOC_DIM_Y})\rceil + 1 + \text{PAYLOAD_SIZE}) \quad (3.2)$$

No exemplo anterior, tem-se uma conexão com $2(2 + 39) = 82$ bits de largura.

No momento da instanciação do roteador, é necessário indicar, através dos *generics* `LINK_W`, `LINK_N`, `LINK_E` e `LINK_S`, quais portas estarão disponíveis para conexão. Isso permite a otimização da instância gerada, eliminando-se FIFOs e ignorando-se estados referentes a portas não utilizadas.

A máquina de estados de controle do roteador é composta por 6 estados: `CK_CPU`, `CK_W`, `CK_N`, `CK_E`, `CK_S` e `CK_CTRL`. Ela é responsável pelo transporte de pacotes entre as FIFOs internas e, conseqüentemente, pelo algoritmo de roteamento.

Como dito anteriormente, o algoritmo de roteamento adotado foi o XY, que funciona da seguinte maneira: se a coordenada X de destino de um dado pacote é menor que a coordenada X do roteador, envia-se o pacote pela porta W, se é maior, pela porta E. Caso seja igual, verifica-se a coordenada Y, que se, por sua vez, for menor que a coordenada

Y do roteador, provoca o envio do pacote pela porta N, se maior, pela porta S, sendo igual, o pacote encontrou seu destino e é entregue à interface da CPU. A vantagem dessa abordagem é que, sendo um algoritmo determinístico, a ordem de entrega dos pacotes é sempre garantida, dado que os pacotes provenientes do processador A para o processador B sempre seguem o mesmo caminho pelos roteadores.

Esse roteamento é feito em cada um dos estados, que checka a porta correspondente por novos pacotes. No estado **CK_CPU**, verifica-se se há pacotes a serem enviados na interface da CPU. Pacotes de *loopback*, cujo destino é o próprio roteador, são válidos e são reinseridos na interface. Caso contrário, as coordenadas são avaliadas e o pacote é retirado da interface e inserido na FIFO de saída da direção correta, como descrito no algoritmo. Em seguida transita-se para o próximo estado cuja porta correspondente está habilitada.

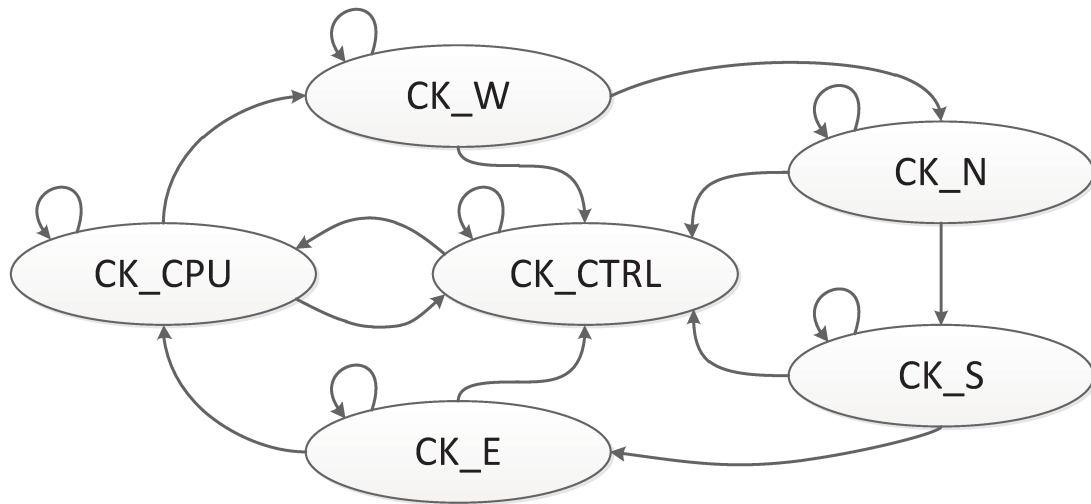


Figura 3.4: Representação da máquina de estados de controle do roteador.

Nos estados **CK_W**, **CK_N**, **CK_E** e **CK_S**, primeiramente checka-se se há pacote disponível na entrada da porta correspondente através do sinal de entrada *ready*, se houver, o pacote é inserido na FIFO de entrada e sinaliza-se sua recepção através do sinal de saída *get*. Nesse mesmo estado, verifica-se se há pacote disponível na FIFO de entrada, se houver, ele é inserido na FIFO de saída da direção correta, seguindo o algoritmo XY. Sempre que a FIFO de saída de uma porta não estiver vazia, o sinal de saída *ready* correspondente estará habilitado, indicando ao roteador conectado nela que há um pacote disponível. Em seguida transita-se para o próximo estado cuja porta correspondente esteja habilitada. Após a checagem da última porta habilitada, retorna-se ao estado **CK_CPU**. A Figura 3.4 mostra todas as transições de estado possíveis.

O bit de sequência contido no pacote permite que seja enviada uma sequência contínua

de pacotes, garantindo que todos chegarão ao destino sequencialmente evitando a intercalação com pacotes provenientes de outras origens. A implementação desse recurso se dá da seguinte forma: dado que um pacote sequencial é disponibilizado em uma das portas, a máquina de estados não transita para o próximo estado enquanto ela continuar a receber pacotes sequenciais daquela porta. O primeiro pacote não sequencial recebido indica o fim da sequência e a transição para o próximo estado ocorre normalmente. Esse mecanismo deve ser utilizado com cuidado, pois caso os pacotes não sejam consumidos constantemente pelo destino ou sequências muito grandes sejam geradas, as FIFOs no caminho do pacote sequencial podem encher e pacotes provenientes de outros roteadores não serão transmitidos, pois o pacote sequencial nunca é consumido e a máquina de estados nunca verifica as outras filas por novos pacotes (*starvation / deadlock*).

Em todos os estados, verifica-se se há comando de controle proveniente da CPU. Caso haja, transita-se imediatamente para o estado `CK_CTRL`. Há três comandos implementados para o roteador. Um que habilita seu funcionamento, outro para desabilitar e um terceiro para limpar todas as FIFOs de entrada e saída, inclusive as contidas na interface da CPU. Novos comandos podem ser facilmente adicionados caso necessário.

3.3 Topologia e conexão entre roteadores

Foi criado um módulo separado responsável pela conexão entre os roteadores chamado *noc_connection*, localizado no arquivo *noc_connection.vhd*. Nele pode-se adicionar novas configurações de conexão através da modificação de algumas tabelas.

Cada roteador possui um identificador inteiro que está entre 0 e `NOC.TOPOLOGY-1`, onde `NOC.TOPOLOGY` representa o total de roteadores e processadores instanciados. É previsto que o identificador de um dado roteador corresponda ao identificador do processador ao qual ele está ligado.

Dado que deseja-se alterar a topologia da rede, a primeira alteração deve ser feita na constante `DIR_EN`, que representa quais portas estão habilitadas em cada roteador. Essa tabela será utilizada no laço de geração de roteadores, removendo portas não utilizadas.

Em seguida adiciona-se o gerador de conexões para a nova topologia. Ela mapeará portas de entrada em portas de saída. Como exemplo, em uma topologia 2x1, temos a porta de saída W do roteador 1 ligada a porta de entrada E do roteador 0, e vice-versa. Todas as outras estarão desabilitadas.

O trecho de código abaixo ilustra as modificações e adições necessárias para adicionar a topologia 2x2, no arquivo *noc_connection.vhd*.

```

1  — from file "noc_lib/lsc/routernoc/noc_connection.vhd"
2  ...
3  constant DIR_EN : dir_en_t := (
4      ...

```

```

5      NOC_2x2 => (
6          W => (0, 1, 0, 1, others => 0),
7          N => (0, 0, 1, 1, others => 0),
8          E => (1, 0, 1, 0, others => 0),
9          S => (1, 1, 0, 0, others => 0)
10     ),
11     ...
12     others => dir_name_null
13 );
14
15 — NOC_2x2 topology generation
16 top2x2_gen: if NOC_TOPOLOGY = NOC_2x2 generate
17     lksi(0) <= (
18         W => link_null ,
19         N => link_null ,
20         E => lkso(1).W,
21         S => lkso(2).N
22     );
23     lksi(1) <= (
24         W => lkso(0).E,
25         N => link_null ,
26         E => link_null ,
27         S => lkso(3).N
28     );
29     lksi(2) <= (
30         W => link_null ,
31         N => lkso(0).S,
32         E => lkso(3).W,
33         S => link_null
34     );
35     lksi(3) <= (
36         W => lkso(2).E,
37         N => lkso(1).S,
38         E => link_null ,
39         S => link_null
40     );
41     lksi(4 to NOC_MAX_TOPOLOGY) <= (others => links_null);
42 end generate;
43 ...

```

Essa modularização permite a criação de novas topologias mais facilmente, localizando todo o processo em apenas um arquivo.

3.4 Interface com a CPU

Assim como as portas de ligação à outros roteadores, a interface com a CPU, chamada **CPUInterface**, também é composta por duas FIFOs, para envio e recebimento de pacotes. Ela apresenta duas funcionalidades extras que justificam sua separação do roteador: tradução de identificador em coordenadas e eliminação de pacotes inválidos.

O processador, ao escrever na entrada da interface, indica o identificador do processador de destino, um inteiro entre 0 e `NOC_TOPOLOGY - 1`, além da palavra a ser transferida.

A interface traduz esse valor para coordenadas da NoC, consultando uma tabela através da função `to_coord_t`, definida no pacote principal `router_pkg.vhd`. Ela também adiciona as coordenadas de origem.

Caso o identificador de destino seja inválido, como por exemplo um valor maior que `NOC_TOPOLOGY-1`, o pacote é descartado imediatamente. Isso permite algumas otimizações no código do roteador, já que pode-se supor sempre pacotes válidos vindos da CPU.

Há mais dois sinais de entrada vindos da CPU que permitem a execução dos comandos de controle, um para sua habilitação (`ctrlen`) e outro conjunto com o comando em si (`ctrl`), definidos no tipo `fromcpu_int_t`, que contém todos os sinais vindos da CPU. Esses sinais referentes ao controle são repassados diretamente ao roteador, que trata a requisição como descrito na Seção 3.2.

São disponibilizados para a CPU sinais de saída com a próxima palavra da fila e dois contadores que indicam quantas palavras estão disponíveis tanto na entrada como na saída da interface. Esses sinais são necessários para o funcionamento correto das instruções no *pipeline* do processador e são justificados na Seção 3.5. Todos os sinais de saída para CPU estão definidos no tipo `tocpu_int_t`.

3.5 Modificações no LEON3

O suporte ao mecanismo de comunicação implementado no processador se dá através da adição de 3 novas instruções: **SEND**, **RECV** e **NCTRL**.

O formato 3 de instrução SPARC (vide Figura 2.9) foi selecionado por atender todos os requisitos das novas instruções, descritos a seguir. O formato comporta um registrador de destino e até dois de argumento, onde um deles pode ser substituído por um imediato de 13 bits.

A instrução **SEND** requer dois argumentos de entrada, um com o identificador do processador de destino e outro com a palavra de 32 bits a ser transmitida. Utiliza-se o bit mais significativo do registrador de processador de destino como bit de sequência caso se queira uma transmissão contínua de pacotes, como detalhado na Seção 3.2, ou seja, enquanto esse bit for 1, os pacotes serão transmitidos sequencialmente, onde o último pacote da sequência é indicado por esse mesmo bit igual a 0.

A instrução **RECV** necessita de um registrador para receber o próximo valor disponível na fila de pacotes. É previsto um segundo registrador de entrada para a instrução que indica o identificador do processador de origem do qual deseja-se receber o pacote, mas esse recurso foi deixado para trabalhos futuros. Logo, esse valor é ignorado e considerado reservado na implementação atual.

A instrução **NCTRL** recebe como argumento um registrador com a função de controle a ser executada pelo roteador. Foram implementadas 3 funções, habilitação e desabilitação

do roteador e limpeza das FIFOs. Deseja-se que seja possível a adição de novas funcionalidades, inclusive com possíveis valores de retorno, como por exemplo estatísticas do roteador ou origem do último pacote recebido.

Como não há um RTC (*Real Time Clock*) em *hardware*, o sistema operacional utiliza de interrupções geradas por *timers* ligados ao barramento para manter a contagem de tempo, sendo assim, funções como *gettimeofday* têm precisão máxima de milissegundos e o mascaramento de interrupções, utilizado pelas novas instruções como mostrado mais adiante, podem ter sua medição comprometida. Adicionou-se então uma funcionalidade à instrução **NCTRL** para reiniciar ou retornar um contador de microssegundos de 32 bits, provendo maior precisão nas medições e evitando erros nestas devido ao mascaramento de interrupções.

A Tabela 3.1 mostra os *opcodes* disponíveis na arquitetura SPARCV8, segundo [33], e selecionados para as novas instruções.

Mnemônico	op	rd	op3	rs1	opf	rs2
SEND	10	00000	001001 (0x09)	dest_id_reg	000000000	data_reg
RECV	10	data_rcv_reg	011001 (0x19)	src_id_reg	000000000	00000
NCTRL	10	data_nctrl_reg	101100 (0x2C)	00000	000000000	nctrl_cmd_reg

Tabela 3.1: *Opcodes* das novas instruções de comunicação.

Como descrito na Seção 2.4.2, o *pipeline* do processador LEON3 é dividido em 7 estágios: *fetch*, *decode*, *register access*, *execute*, *memory*, *exception* e *write-back*. Esta seção apresenta a funcionalidade e modificações necessárias em cada estágio na unidade de inteiros da CPU (*ui3.vhd*).

No estágio de *fetch*, nenhuma alteração foi necessária. Ele é responsável por requisitar à *cache* a instrução indicada pelo PC (*Program Counter*). A alteração desse registrador é gerada aqui. Seu valor é incrementado no fluxo contínuo de execução, mas caso alguma instrução de desvio de fluxo tenha sido decodificada, seu valor é modificado de acordo. Pode-se também evitar, no estágio de *decode*, que seu valor seja modificado através do sinal **de_hold_pc**, recurso necessário na implementação das novas instruções.

Já com o conteúdo da instrução disponível, no estágio de *decode* interpreta-se cada campo da instrução preparando, por exemplo, os endereços a serem requisitados no banco de registradores de acordo com os campos **rs1** e **rs2**. Determina-se aqui a origem dos operandos de entrada para a ULA (Unidade Lógica Aritmética). A alteração nesse estágio está na geração do sinal **de_hold_pc**. No caso da instrução **SEND**, tem-se que verificar nesse momento se há espaço disponível na fila de saída e quantas instruções **SEND** válidas foram decodificadas anteriormente. Por instrução válida entende-se instruções que necessariamente irão escrever na fila no estágio *execute*. Por isso temos que garantir que há espaço

ainda no estágio de *decode*. Acessando os valores em registradores de estágios seguintes, consegue-se essa informação. A fila de saída ainda deve suportar pelo menos mais uma palavra caso não haja instruções de **SEND** válidas nos estágios de *register access* e *execute*, pelo menos duas caso haja um **SEND** válido em um dos dois estágios, ou três, caso haja um **SEND** válido nos dois estágios. Trata-se a instrução **RECV** de forma similar, mas considera-se quantos valores estão disponíveis para serem recebidos na fila de entrada ao invés de quanto espaço disponível existente na fila de saída.

Caso os requisitos anteriores sejam atendidos, permite-se a passagem da instrução de **SEND** ou **RECV** válida para o próximo estágio, caso contrário, invalida-se a instrução e força-se o PC a não incrementar no próximo ciclo através do sinal *de_hold_pc*. Isso faz com que a instrução entre novamente no *pipeline* no próximo ciclo. Por isso, sempre que uma instrução é invalidada, inicia-se uma contagem até um valor máximo, determinado na constante *NOC_TIMEOUT_VAL* que, quando atingido, permite que o processador prossiga a execução, emitindo uma instrução com uma marca de *timeout*, ou seja, as instruções de **SEND** e **RECV** são bloqueantes até que aconteça um *timeout*. Tomou-se esse cuidado para permitir que sequência contínuas de instruções **SEND** e **RECV** possam ser emitidas pois, dessa forma, consegue-se extrair o máximo de capacidade da NoC. A instrução **NCTRL** não exige nenhum tratamento especial nesse estágio.

O estágio *register access* seleciona a operação a ser executada pela ULA assim como uma segunda fase de seleção dos operandos, permitindo que valores em registradores de outros estágios sejam utilizados (*register forwarding*). Nesse estágio também seleciona-se a operação de comunicação realizada no estágio seguinte, *execute*, independente da instrução ser válida ou não.

No estágio *execute*, já de posse dos valores contidos nos registradores indicados na instrução, executa-se a operação selecionada, escrevendo o valor a ser enviado na interface da NoC, no caso do **SEND**, ou lendo-se o próximo valor disponível, no caso do **RECV**, o que ocorre apenas se a instrução for válida. Caso seja marcada como *timeout*, ignora-se o **SEND** ou retorna-se um valor padrão para o **RECV**. Não há mecanismo implementado para a identificação de um *timeout* além do retorno de um valor padrão, mas esse recurso poderia ser adicionado como uma das funcionalidades da instrução **NCTRL**. A instrução **NCTRL** também é executada nesse estágio, escrevendo-se a palavra de controle e habilitando-se o sinal *ctrlen*. O valor do contador de microssegundos também é lido ou reiniciado nesse estágio. Nesse momento, o resultado de um **RECV** ainda não foi escrito no banco de registradores e está contido em registradores intermediários do *pipeline*. A escrita só acontecerá no estágio de *write-back*.

Além de cuidar do acesso à memória externa através da *cache* de dados, é no estágio de *memory* que se detecta interrupções geradas interna e externamente. Uma vez detectada uma interrupção, o fluxo de execução mudará para o endereço de tratamento já no próximo

ciclo, retornando após o tratamento, a última instrução indicada como não concluída no *pipeline* no momento da interrupção. Ao contrário das outras instruções SPARC, não pode-se permitir que as instruções de comunicação válidas sejam executadas mais de uma vez, pois valores já foram retirados ou colocados nas filas anteriormente, o que levaria a duplicação de dados enviados ou perda de dados recebidos, já que o registrador de destino ainda não foi escrito com o valor recebido. Para evitar esses problemas, desabilita-se o tratamento de quaisquer interrupções enquanto houverem instruções **SEND** ou **RECV** válidas no *pipeline*. As interrupções não são perdidas, apenas tratadas posteriormente. Permitir o tratamento de interrupções durante a execução dessas instruções não é um problema trivial, pois implica ou na reestruturação da arquitetura de interrupções do *pipeline*, o que não é recomendável, ou na criação de um módulo, contendo *buffers* temporários, que garanta a aplicação correta das operações na NoC (*commits* das operações) fazendo uma análise do está ocorrendo dentro do *pipeline*, o que adicionaria alguma latência no processo de comunicação.

O estágio de *exception* calcula uma série de sinais para preparar o *pipeline* para o atendimento de uma interrupção caso uma tenha sido detectada no ciclo anterior. Dentre eles está, por exemplo, o endereço de desvio para o tratamento da interrupção.

Finalmente, no estágio de *write-back*, escreve-se no registrador de destino o resultado da operação, no caso, a palavra recebida pela instrução **RECV** através da NoC.

Todas as alterações foram feitas de forma que uma aplicação que não contenha as novas instruções execute exatamente da mesma maneira e no mesmo tempo que o processador original, sem adição de novas latências. A GRLib provê uma bateria de testes de sanidade para o LEON3, que foram executados e completados com sucesso.

3.6 Módulo de atraso da memória DDR2

Devido a limitações da FPGA utilizada nos testes (Altera - Stratix III [7]), o *clock* máximo para o sistema é de 100 MHz, garantindo o funcionamento estável nessa FPGA. Nessa frequência, todos os tempos de *setup*, *hold* e propagação são respeitados em todos os caminhos críticos evitando qualquer meta-instabilidade no circuito.

A placa do *kit* de desenvolvimento utilizado (*Stratix III FPGA Development Kit* [8]) possui um conector DIMM para memória DDR2, que é utilizada como memória principal do sistema. A memória conectada tem 1 GB de capacidade e a PHY de controle da DDR2 presente na FPGA permite que ela funcione numa frequência de, no mínimo, 200 MHz. Apesar de teoricamente permitir frequências de operação mais baixas, de até 125 MHz, os testes executados mostraram que o sistema se torna muito instável em frequências menores que 200 MHz. Supõe-se que seja alguma limitação no controlador provido com o sistema do LEON3 no que se refere a calibragem dos atrasos das linhas de dados. A

correção desse problema foi considerada além do escopo do projeto.

Dados esses valores de *clocks*, percebe-se uma anomalia no sistema, onde a frequência de operação da memória é o dobro da frequência do sistema e, mais especificamente, do processador. Como deseja-se explorar, na solução proposta, as vantagens da nova técnica em relação ao custo de acesso à memória principal feito pela *cache* de dados quando suas linhas são inválidas, caso típico no uso de memória compartilhada entre diferentes processadores, é preciso inserir um módulo que gere um atraso relativo à leitura de dados da memória principal.

Aumentar esse custo torna o sistema mais próximo de um sistema comum, onde o acesso a dados válidos na *cache* é consideravelmente mais rápido que a leitura da memória principal ligada ao barramento.

Com esse intuito, foi desenvolvido um módulo que é instanciado entre o controlador da memória DDR2 e o barramento AHB, como mostrado na Figura 3.5 como *Memory Delay*. Esse módulo um registrador de 32 bits onde pode-se habilitar e desabilitar a inserção de atraso e o número de ciclos de atraso inseridos.

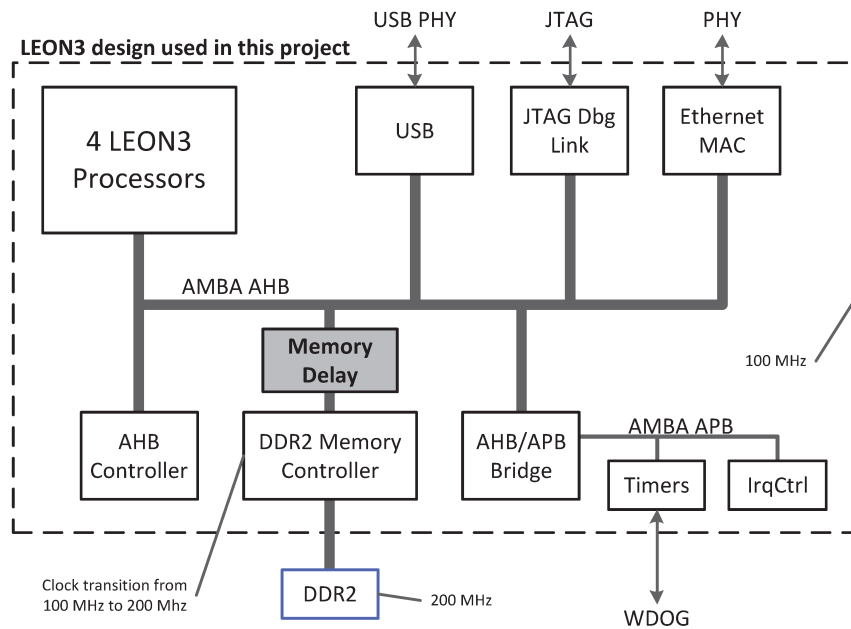


Figura 3.5: Arquitetura de alto nível completa do sistema utilizado no projeto, incluindo o módulo de atraso de memória (*Memory Delay*) e domínios de *clock*.

Seu funcionamento consiste basicamente em armazenar as requisições AHB para a memória em uma FIFO ao invés de passá-las diretamente ao controlador. Após a contagem do número de ciclos configurados, caso o atraso esteja habilitado, as requisições são enviadas a memória. No caso de uma requisição de escrita, o barramento pode ser

imediatamente liberado após o armazenamento da requisição na FIFO, pois não há nenhuma dependência de dados vindos do controlador para que a requisição seja concluída. Sendo assim, não há nenhum ônus no tempo de escrita de dados. Já no caso de uma leitura, o barramento é bloqueado pelo número de ciclos determinado pelo registrador de configuração até que a operação de leitura de fato seja feita. Assim que a memória retorna os dados requisitados, o barramento é liberado com a resposta da requisição de leitura.

No caso de um *burst* de leitura, sempre executado pela *cache* para o preenchimento completo de uma linha, é preciso garantir que o atraso inserido seja por *burst* e não para cada *word* requisitada, ou seja, se foram configurados 32 ciclos de atraso, cada *burst* de leitura terá no mínimo 32 ciclos de atraso. Implementou-se esse mecanismo iniciando-se a leitura da memória assim que a requisição é feita, gerando um *burst* de leitura para o controlador DDR2 e armazenando as respostas em outra FIFO. Assim que são completados os ciclos de atraso mínimos necessários, as respostas começam a ser enviadas no barramento sequencialmente, retirando-se os valores dessa segunda FIFO que funciona como um *buffer*.

Como mostrado na Seção 4.2, o modelo de atraso proposto provou-se eficaz e coerente com as expectativas previstas nessa seção. Através dele pode-se evidenciar a relação entre o custo de acesso à memória principal e o ganho relativo ao uso da NoC através das novas instruções, como indicado principalmente nos experimentos da Seção 4.5.

Tratou-se nesse capítulo de todo o esforço de desenvolvimento em *hardware* necessário para a implementação do novo modelo de comunicação proposto. No próximo capítulo será apresentada a implementação em *software* necessária para o uso do mecanismo e sua avaliação, incluindo resultados experimentais.

É válido enfatizar que para todos os testes e cálculos apresentados no próximo capítulo, o *clock* do sistema é de **100 MHz** e o da memória DDR2 principal é de **200 MHz**.

Capítulo 4

Implementação em *Software* e Resultados

Este capítulo apresenta o esforço de desenvolvimento em toda a parte de *software*, contendo a adição de suporte aos novos mecanismos providos pela NoC ao compilador e os programas sintéticos desenvolvidos para testar o desempenho do novo mecanismo de transporte de dados em comparação ao uso de *pthread_locks* em diversos cenários dentro do sistema operacional GNU/Linux.

4.1 Suporte as novas instruções no compilador

Para a geração do *cross-compiler* utilizado no projeto foi usada a ferramenta *Buidroot* [27], que além disso facilita a aplicação de *patches* necessários no compilador e a geração do sistema operacional embarcado com todas as alterações necessárias para suportar o LEON3 e seus periféricos. Esse compilador é para uso exclusivo em aplicações que executam no Linux, incluindo o próprio sistema operacional. Para aplicações *bare-metal*, alterou-se e utilizou-se em estágios iniciais do projeto o compilador BCC, fornecido pela própria Gaisler [15].

O objetivo da alteração do compilador é disponibilizar as instruções **SEND**, **RECV** e **NCTRL** para uso em código C através de *asm-inline* [31]. Para tanto, a alteração deve ser feita apenas no montador contido no *binutils*.

Há dois arquivos que definem o conjunto de instruções do SPARCV8:

sparc-opc.c: arquivo de definição de formato de instruções, mnemônicos, *opcodes* e número e posição de registradores de argumentos. Aqui foram adicionadas as instruções com todas as variações permitidas para argumentos, inclusive suporte a imediatos como identificador de destino.

sparc-dis.c: definição de como o *disassembler* deve interpretar e comparar os *opcodes*, decidindo corretamente como mostrar e gerar os imediatos e registradores das instruções.

Após a modificação, gerou-se novamente as ferramentas do *binutils* permitindo a utilização das três instruções através das seguintes macros definidas no arquivo *noc_api.h*, utilizadas em todos os testes:

```

1 // from file "sw/pipeline/noc_api.h"
2 ...
3 #define NOC_SEND_SEQ      0x80000000
4
5 #define NCTRL_CMD_NULL    0x00000000
6 #define NCTRL_ROUTER_NULL 0x00000000
7 #define NCTRL_ROUTER_CLR  0x00000001
8 #define NCTRL_ROUTER_EN   0x00000002
9 #define NCTRL_ROUTER_DIS  0x00000003
10 #define NCTRL_CNT_RST     0x00000004
11 #define NCTRL_CNT_GET     0x00000005
12
13 #define ASMNCTRL(none, val, dest) \
14     __asm__ __volatile__ (        \
15         "nctrl %1, %2, %0;\n"      \
16         : "=r" (dest)              \
17         : "r" (none), "r" (val)    \
18     );
19
20 #define ASMSEND(id, val) \
21     __asm__ __volatile__ ( \
22         "send %0, %1;\n"    \
23         : /* no output */   \
24         : "r" (id), "r" (val) \
25     );
26
27 #define ASMRECV(id, dest) \
28     __asm__ __volatile__ ( \
29         "recv %1, %0;\n"    \
30         : "=r" (dest)      \
31         : "r" (id)         \
32     );
33
34 #define NOC_CLEAR(a) { \
35     ASMNCTRL(NCTRL_CMD_NULL, NCTRL_ROUTER_DIS, a); \
36     ASMNCTRL(NCTRL_CMD_NULL, NCTRL_ROUTER_CLR, a); \
37     ASMNCTRL(NCTRL_CMD_NULL, NCTRL_ROUTER_EN, a); \
38 }
39
40 #define ASMNOP() \
41     __asm__ __volatile__ ( \
42         "nop;\n"          \
43     );
44
45 #define ASMLD(addr, dest) \
46     __asm__ __volatile__ ( \
47         "ld [%1], %0;\n"    \
48         : "=r" (dest)      \

```

```

49      : "r" (addr)          \
50    );
51
52 #define ASM_ST(val, addr)    \
53   __asm__ __volatile__ (     \
54     "st %0, [%1];\n"         \
55     : /* no output */        \
56     : "r" (val), "r" (addr) \
57   );
58
59 #define NOC_CNT_RST(a) {      \
60   ASM_NCTRL(NCTRL_CMD_NULL, NCTRL_CNT_RST, a); \
61 }
62
63 #define NOC_CNT_GET(a) {      \
64   ASM_NCTRL(NCTRL_CMD_NULL, NCTRL_CNT_GET, a); \
65 }
66
67 ...

```

A linha 3 define a máscara para gerar uma transmissão sequencial que deve ser combinada com o identificador do processador de destino através da operação **OR** enquanto a transmissão for sequencial. As linhas 5 a 11 definem todas as operações suportadas pela instrução **NCTRL** para apagamento das FIFOs, habilitação e desabilitação dos roteadores e reinício e leitura do contador de microssegundos. As macros na sequência são utilizadas para as instruções em si e serão exemplificadas nas próximas seções. Algumas outras macros também foram definidas para facilitar a legibilidade do código e serão referenciadas mais adiante no texto.

Esses já são todos os recursos necessários para utilizar todas as funcionalidades adicionadas em um código C. Alguns cuidados foram tomados em relação a utilização da medição de tempo provida pela instrução **NCTRL**, pois deve-se garantir que a leitura do contador seja feita sempre no mesmo processador para evitar a comparação entre valores originados de processadores distintos. Para isso, criam-se *threads* exclusivamente para a captura do tempo inicial e final de cada execução, forçando que estas sempre executem no processador 0.

4.2 Testes de desempenho da memória DDR2

Como descrito no modelo de atraso de memória na Seção 3.6, é possível ajustar o número de ciclos de atraso antes que a memória retorne valores lidos, o que simula uma memória mais lenta e aumenta o custo de tempo para acesso a valores não presentes na *cache*.

É necessário medir o comportamento da inserção desses ciclos de atraso para avaliar o quanto o acesso a memória será afetado, provando a eficácia do modelo, necessário para remover a anomalia de proporção entre as frequências de operação da memória e

do processador, deixando mais claro o ganho provido pelo mecanismo de comunicação proposto nesse projeto.

Os valores referentes a todos os gráficos apresentados nessa seção estão contidos nas Tabelas B.10, B.11, B.12, B.13 e B.14 no apêndice B. Os erros das medidas não são mostrados nos gráficos para facilitar a legibilidade, mas estão presentes nas tabelas.

Parte do código utilizado na medição é apresentado a seguir:

```

1  // from file "sw/memtest/memtest.c"
2  ...
3  memset(vec0, 0x0, 128 * 1024);
4  memset(vec1, 0x0, 128 * 1024);
5
6  NOC.CNT_RST(e0);
7
8  //////////////////////////////////////
9  // Write time
10 //////////////////////////////////////
11 NOC.CNT_GET(e0);
12
13 for (i = 0; i < ((size * 1024) / 4); ++i) {
14     g = i;
15     a = (uint32_t)(vec0 + i);
16     ASM_ST(g, a);
17 }
18
19 NOC.CNT_GET(ef);
20
21 wtime = ef - e0;
22
23 //////////////////////////////////////
24 // Read time
25 //////////////////////////////////////
26 NOC.CNT_GET(e0);
27
28 for (i = 0; i < ((size * 1024) / 4); ++i) {
29     a = (uint32_t)(vec0 + i);
30     ASM_LD(a, g);
31 }
32
33 NOC.CNT_GET(ef);
34
35 rtime = ef - e0;
36
37 //////////////////////////////////////
38 // RW time
39 //////////////////////////////////////
40 NOC.CNT_GET(e0);
41
42 for (i = 0; i < ((size * 1024) / 4); ++i) {
43     a = (uint32_t)(vec0 + i);
44     ASM_LD(a, g);
45     a = (uint32_t)(vec1 + i);
46     ASM_ST(g, a);
47 }
48
49 NOC.CNT_GET(ef);

```

```

50
51     rwtime = ef - e0;
52     ...

```

As linhas 2 e 3 são necessárias para garantir que o *kernel* aloque as páginas necessárias para os vetores antes da medição de tempo, evitando latência adicional durante as medições.

Inicialmente, nas linhas de 10 a 20, mede-se o tempo para executar uma escrita de tamanho, em kilobytes, determinado pela variável `size`. Executa-se esse trecho de código para diferentes valores dessa variável e ciclos de atraso incrementais, resultando no gráfico da Figura 4.1.

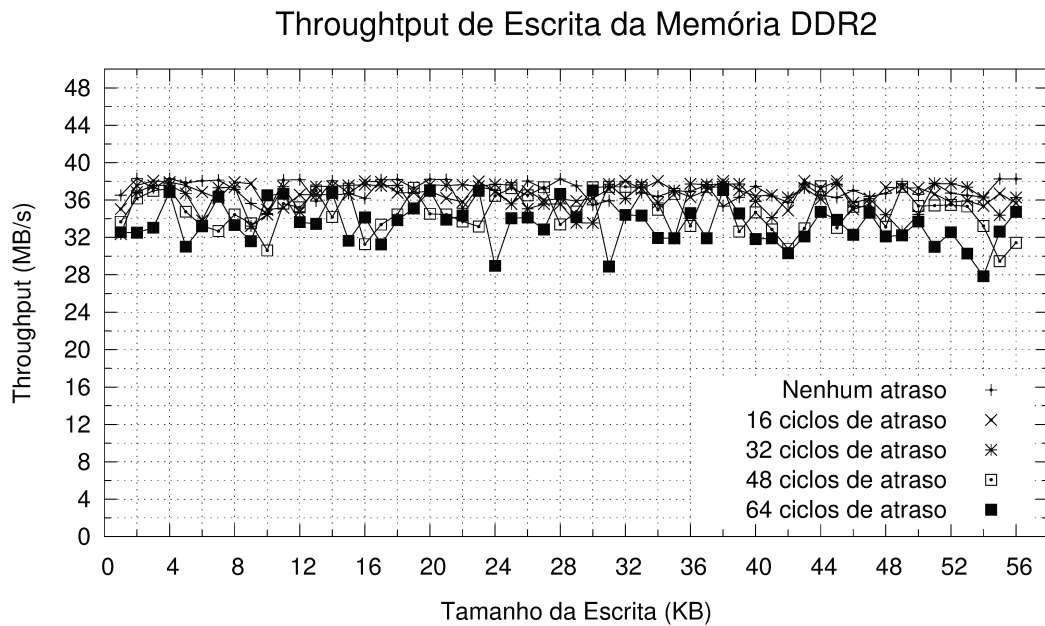


Figura 4.1: Desempenho de escrita da memória DDR2 com ciclos de atraso variados.

Percebe-se que, como previsto, a alteração do número de ciclos de leitura da memória afeta muito pouco o tempo de escrita, pois temos no sistema uma *cache write-through* e um *buffer* de escrita na entrada da memória. A pequena diferença no desempenho se dá devido ao acesso à memória principal para leitura da seção de código, feito pela *cache* de instruções, e ao escalonamento dos processos, o que afeta inevitavelmente a medição. Temos uma média de *throughput* de escrita na DDR2 de 35.6 MB/s.

Nas linhas de 25 a 34 mede-se o tempo de leitura de forma similar à escrita apresentada anteriormente. O resultado é apresentado na Figura 4.2.

Como ainda não há valores válidos na *cache*, pois as linhas foram invalidadas pela escrita anterior, temos taxas relativamente constantes independente do tamanho do vetor

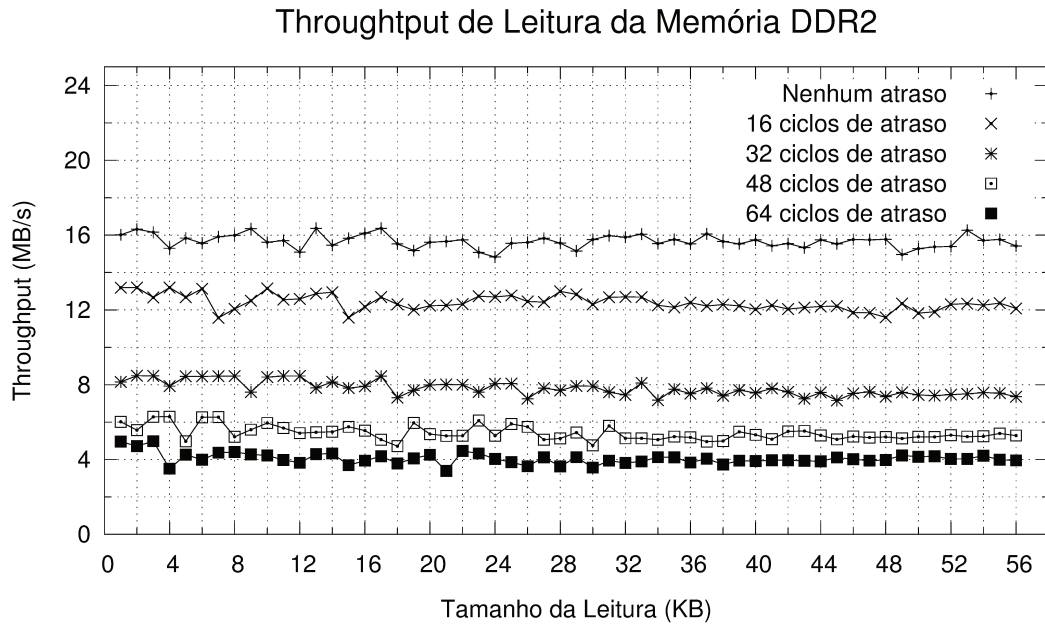


Figura 4.2: Desempenho de leitura da memória DDR2 com ciclos de atraso variados.

lido. Observa-se claramente a consequência da adição de ciclos de atraso para a leitura, onde a cada 32 ciclos de latência adicionados, o *throughput* de leitura da memória cai aproximadamente pela metade. A Tabela 4.1 mostra a média do desempenho da leitura para cada uma das curvas apresentadas.

Atraso (ciclos)	<i>Throughput</i> médio de leitura (MB/s)
0	15,7
16	12,4
32	7,8
48	5,4
64	4,0

Tabela 4.1: *Throughput* médio de leitura da DDR2 para variados ciclos de atraso.

Num terceiro passo, referente às linhas 39 a 50, executa-se a cópia de um vetor *vec0* para o vetor *vec1*. Como o vetor *vec0* foi lido no passo anterior, seus valores estão presentes e são válidos na *cache*, o que gera o pico de *throughput* para vetores de até 8kB de comprimento, tamanho exato da *cache* de dados configurada no sistema, como observado na Figura 4.3.

Como o gargalo está sempre na leitura, as taxas medidas na parte constante do gráfico são proporcionais mas menores que as taxas de leitura, pois adicionou-se o passo de escrita.

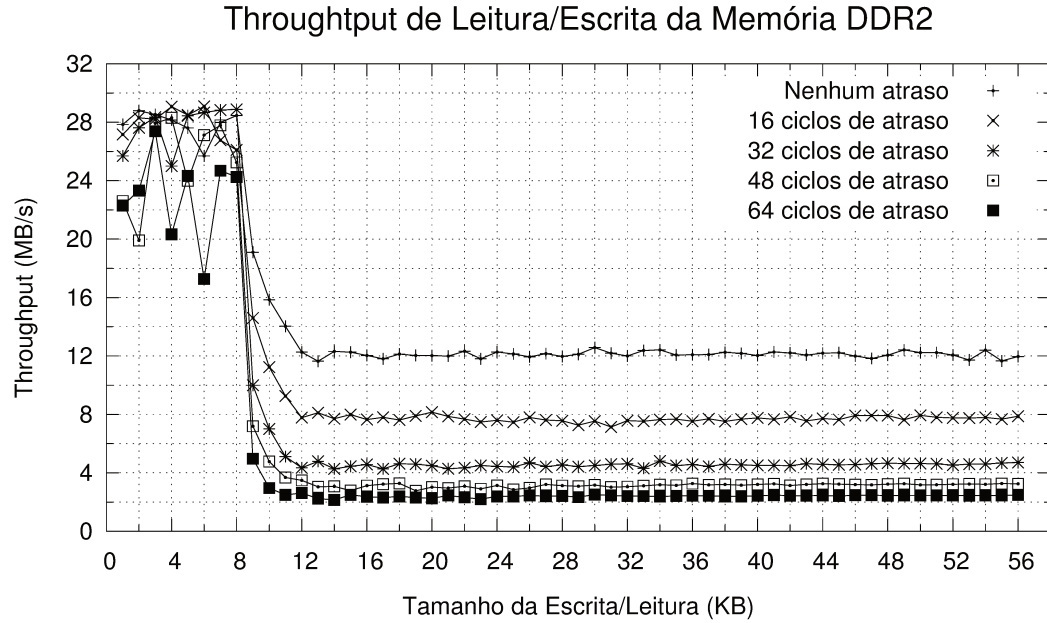


Figura 4.3: Desempenho de leitura seguida de escrita da memória DDR2 com ciclos de atraso variados.

A Tabela 4.2 mostra a media desses valores após a estabilização.

Atraso (ciclos)	<i>Throughput</i> da cópia (MB/s)
0	12,1
16	7,7
32	4,5
48	3,1
64	2,4

Tabela 4.2: *Throughput* médio para cópia de vetor na DDR2 para variados ciclos de atraso após estabilização ($\geq 12\text{kB}$).

Avaliando todos os dados apresentados, foi possível confirmar experimentalmente que chegou-se a um modelo razoável para inserir atraso na resposta da leitura de memória, aumentando o custo de acesso pela *cache* à memória principal e tornando o sistema utilizado no projeto mais próximo de um sistema comum. Utilizou-se para os próximos experimentos os atrasos de 32 e 64 ciclos, o que torna a leitura da memória duas ou quatro vezes mais lenta, respectivamente.

4.3 Testes de desempenho da NoC

Deseja-se extrair, nos testes dessa seção, o maior *throughput* possível para o transporte de dados na NoC. Para isso, segue a parte essencial do código desenvolvido com o objetivo de extrair esses valores.

```

1 // from file "sw/noctp/noctp.c"
2 ...
3 void * producer(void *arg) {
4     register int i = 0, val = 0, cpu = 0, loop = 0;
5     thread_params *tparams;
6
7     tparams = (thread_params *) arg;
8     ...
9     cpu = tparams->cpu;           // Destination CPU
10    loop = tparams->loop_size;     // Number of words to send
11
12    for (i = 0; i < loop; ++i) {
13        val = 0xAAAAAAAA;
14        ASM_SEND(cpu, val);
15    }
16 }
17
18 void * consumer(void *arg) {
19     register int i = 0, val = 0, loop = 0, cpu = 0;
20     thread_params *tparams;
21
22     tparams = (thread_params *) arg;
23     ...
24     loop = tparams->loop_size;    // Number of words to receive
25     cpu = tparams->cpu;           // Source CPU
26
27     for (i = 0; i < loop; ++i) {
28         ASM_RECV(cpu, val);
29     }
30 }
31
32 int main(int argc, char **argv) {
33     ...
34     // Reset us counter
35     THREAD_TIMER_RST();
36     // Get current us time
37     THREAD_TIMER_GET(e0);
38
39     // create producer thread in CPU0
40     pthread_create(&thread[0], &thread_attr[0], producer, (void *)&tparams[0]);
41     // create consumer thread in CPU1 / CPU3
42     pthread_create(&thread[1], &thread_attr[1], consumer, (void *)&tparams[1]);
43
44     for (i = 0; i < 2; i++) {
45         pthread_join(thread[i], NULL);
46     }
47     // Get final us time
48     THREAD_TIMER_GET(ef);
49     ...
50 }

```

Executa-se uma *thread* produtora na CPU0, enviando sempre o mesmo valor para uma segunda *thread* consumidora que executa nas CPUs 1, vizinha à 0, ou 3, à 1 *hop* de distância da CPU0. É válido lembrar que a disposição e conexão entre as CPUs é mostrada na Figura 3.1.

Decidiu-se por enviar sempre o mesmo valor para evitar a latência adicional de execução de instruções intermediárias entre os envios. Ativou-se também a otimização de *loop-unrolling* para que repetidas instruções de **SEND** e **RECV** fossem geradas pelo compilador sequencialmente, o que maximiza a utilização das FIFOs da CPU e dos roteadores. Dessa forma, tem-se uma noção da máxima potencialidade de transporte de dados provida pelo mecanismo proposto.

Como mostrado na função **main** acima, a medida de tempo é tomada antes da criação das *threads* produtora e consumidora e após o término de sua execução.

O gráfico na Figura 4.4 mostra resultados para a comunicação com uma CPU diretamente adjacente à CPU0 (CPU1) e para a mais distante (CPU3), que exige que os dados passem por um roteador intermediário (ligado à CPU1) antes de chegar ao destino. Os valores utilizados estão contidos nas Tabelas B.15 e B.16, no Apêndice B. Os erros das medidas não são mostrados no gráfico para facilitar a legibilidade, mas estão presentes nas tabelas.

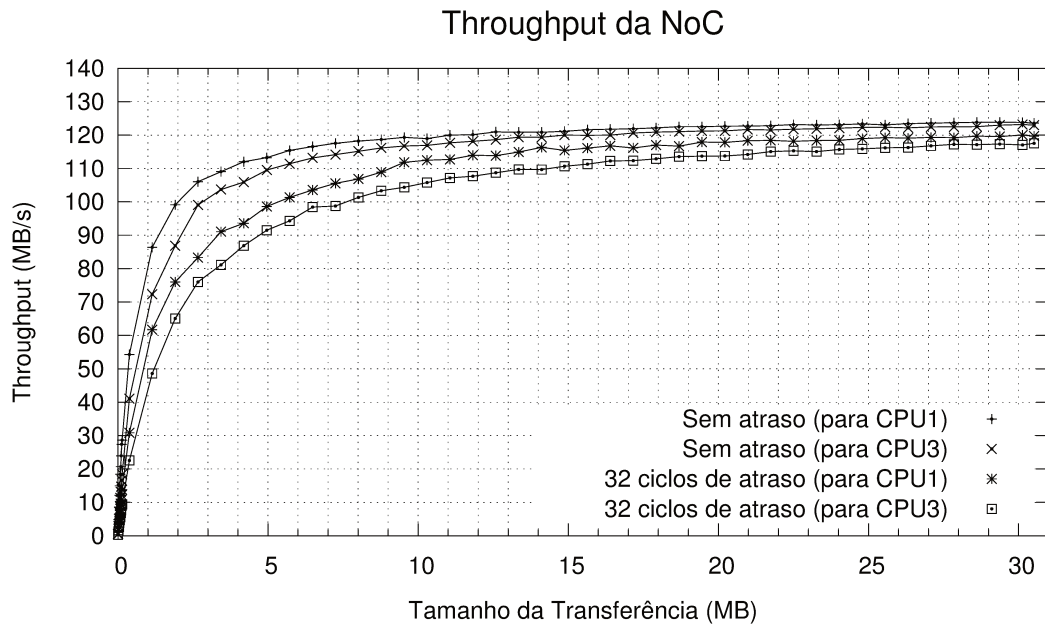


Figura 4.4: Desempenho da NoC para comunicação CPU0 → CPU1 e CPU0 → CPU3 com variação no atraso da memória DDR2.

Para cada execução, varia-se o parâmetro `loop_size`, que define o número de *words* a serem enviadas e recebidas, passado para as *threads* e relativo ao eixo X no gráfico. Para cada curva diferente varia-se a CPU que receberá os dados (parâmetro `cpu`) e o atraso da memória DDR2.

Observa-se que o desempenho claramente aumenta a medida que a quantidade de dados transportado cresce. Isso se dá devido a amortização do tempo gasto para acesso da *cache* de instruções à memória e do tempo de escalonamento de *threads* do sistema operacional. Como esses tempos crescem com a inserção de atraso na memória, nota-se uma diminuição no desempenho que, apesar de muito pequena para vetores maiores, é ainda assim perceptível.

A Tabela 4.3 apresenta os valores máximos de *throughput* medidos para a NoC a partir do ponto onde estes são constantes, juntamente com um cálculo médio de quantos ciclos são gastos no transporte de uma *word*.

Destino	Atraso (ciclos)	<i>Throughput</i> máximo (MB/s)	ciclos/word
CPU1	0	125,7	3,0
CPU3	0	125,7	3,0
CPU1	32	123,8	3,1
CPU3	32	123,6	3,1

Tabela 4.3: *Throughput* máximo da NoC para diferentes destinos e desempenho em médio em número de ciclos por *word* transportada (*clock* do sistema em 100 MHz)

Os valores medidos são bem razoáveis considerando-se a estrutura em *hardware* da NoC onde os dados são movidos através de FIFOs, dando ao transporte uma característica de *pipeline*, o que levaria idealmente a algo próximo de uma *word* por ciclo. Somando-se a latência de roteamento ao fato de que parte das instruções de `SEND` e `RECV` não são executadas diretamente uma após a outra nos processadores chega-se a um valor próximo a 3 ciclos por *word*.

4.4 Testes de aplicação Produtor-Consumidor

Foi desenvolvida uma aplicação onde a CPU0 deve gerar dados a serem consumidos pelas outras 3 CPUs. Ela funciona da seguinte forma: a *thread* na CPU0 executa uma função que gera palavras pseudo-aleatórias que devem ser enviadas às *threads* executando nas CPUs 1, 2 e 3, que recebem esses dados e os armazenam em memória. Um problema de produtor-consumidor 1:3.

Como a rotina do produtor na CPU0 faz a geração dos dados através de uma função matemática, não há necessidade de acesso à memória desde que tome-se o cuidado de

que todas as variáveis utilizadas na função estejam em registradores. A cada iteração da *thread* na CPU0, é enviada uma quantidade parametrizável de *words* aleatórias para uma das CPUs. No caso da versão que utiliza *locks*, isso é feito através de um vetor compartilhado em memória (vetor de transporte) e de variáveis de contenção da biblioteca *pthread*. Utilizando as novas instruções de *SEND* e *RECV*, isso se torna desnecessário, evitando qualquer acesso à memória para essa versão, o que leva a ganhos consideráveis. Em ambos os casos, nos consumidores, esses dados são escritos em um vetor ao serem recebidos.

Para os casos testados, em termos de tempo de execução, a aplicação que faz uso das novas instruções foi de 10 à 70 vezes mais rápida do que a que faz uso de *locks*, dependendo do comprimento do vetor de transporte utilizado. Quanto menor esse vetor, maior é o ganho. Isso se dá devido a duas diferenças essenciais entre as aplicações: a primeira é que a versão que utiliza memória compartilhada deve obrigatoriamente escrever os dados a serem transportadas nesta, e conseqüentemente, somar um custo de tempo para acessar a memória externa. A segunda é o custo de escalonamento das *threads* quando o vetor de transporte não está acessível ou por ainda não ter sido consumido ou por estar na posse da *thread* concorrente.

Nesse caso o ganho foi significativo o suficiente mesmo sem a inserção de atraso na memória, que apenas o tornaria ainda maior.

4.5 Testes de aplicação em *Pipeline*

Com base na maneira com que a transmissão de dados é implementada na NoC, espera-se que os melhores resultados sejam alcançados em aplicações do tipo *pipeline*, onde uma *thread* produz dados para outra que os processa e repassa para uma terceira e assim por diante, onde cada uma dessas *threads* é um estágio do *pipeline*.

A produção de dados para o estágio seguinte exige que sejam implementadas filas em vetores compartilhados na memória com algum controle de concorrência associado, tornando o acesso a esses trechos de memória mutuamente exclusivos, no caso *pthread_locks*. Para esse caso, propõe-se a seguinte aplicação:

```

1 // from file "sw/pipeline/stage_lock.c"
2 ...
3 extern uint32_t data[4][512];
4 ...
5 inline int int_rand(void)
6 {
7     next = next * 1103515245 + 12345;
8     return((unsigned)next);
9 }
10
11 void inc_vec(int loops, uint32_t *vec, int size) {
12     int i = 0, j = 0;
```

```

13     for (i = 0; i < loops; ++i) {
14         for (j = 0; j < size; ++j) {
15             vec[j] += 1;
16         }
17     }
18 }
19
20 void * stage(void *arg) {
21     uint32_t ldata[512];
22     ...
23     cpu = sched_getcpu();
24     tparams = (thread_params *) arg;
25     ...
26     if (tparams->stage == 0) {
27         // Initial producer stage
28         for (i = 0; i < tparams->loop_size; ++i) {
29             // fill vector
30             for (j = 0; j < tparams->vec_size; ++j) {
31                 ldata[j] = int_rand();
32             }
33             // simulate processing delay
34             inc_vec(tparams->load, ldata, tparams->vec_size);
35             // send data to next stage
36             k = tparams->prod_to;
37             pthread_mutex_lock(&data_mutex[k]);
38             ...
39             memcpy(data[k], ldata, tparams->vec_size * sizeof(uint32_t));
40             ...
41             pthread_mutex_unlock(&data_mutex[k]);
42         }
43     } else if (tparams->stage == -1) {
44         // Last consumer stage
45         for (i = 0; i < tparams->loop_size; ++i) {
46             // get data from last stage
47             k = cpu;
48             pthread_mutex_lock(&data_mutex[k]);
49             ...
50             memcpy(ldata, data[k], tparams->vec_size * sizeof(uint32_t));
51             ...
52             pthread_mutex_unlock(&data_mutex[k]);
53             // simulate processing delay
54             inc_vec(tparams->load, ldata, tparams->vec_size);
55         }
56     } else {
57         // other stages
58         for (i = 0; i < tparams->loop_size; ++i) {
59             // get data from previous stage
60             k = cpu;
61             pthread_mutex_lock(&data_mutex[k]);
62             ...
63             memcpy(ldata, data[k], tparams->vec_size * sizeof(uint32_t));
64             ...
65             pthread_mutex_unlock(&data_mutex[k]);
66             // simulate processing delay
67             inc_vec(tparams->load, ldata, tparams->vec_size);
68             // send data to next stage
69             k = tparams->prod_to;
70             pthread_mutex_lock(&data_mutex[k]);

```

```

71         ...
72         memcpy(data[k], ldata, tparams->vec_size * sizeof(uint32_t));
73         ...
74         pthread_mutex_unlock(&data_mutex[k]);
75     }
76 }
77 ...
78 }

```

Existem 3 vetores compartilhados para o transporte de dados, um para o envio de dados da CPU0 para CPU1, outro para CPU1 e CPU2 e mais um para CPU2 e CPU3, contidos no vetores globais `data[1]`, `data[2]` e `data[3]`, respectivamente. Para cada um deles, há um *mutex* associado no vetor `data_mutex[]`.

Preenche-se um vetor de tamanho `vec_size` (esse parâmetro determina o tamanho do vetor de transporte) com dados pseudo-aleatórios no primeiro estágio através da função `int_rand()`. Em todos os estágios simula-se o processamento incrementando cada posição do vetor recebido ou gerado com a `inc_vec()`, que recebe o parâmetro `load` determinando quantas vezes esse incremento será executado e, conseqüentemente, modificando o tempo que um determinado estágio leva para processar os dados recebidos. Deve-se adquirir posse do *mutex* antes de ler ou escrever dados nos vetores compartilhados. Todo o processamento é feito em um vetor local à função para minimizar a concorrência. Foi omitido o controle feito para garantir que o vetor tenha sido consumido antes de ser preenchido novamente, através do uso de variáveis `pthread_cond_t` e das funções `pthread_cond_signal()` e `pthread_cond_wait()`, mas este está presente na versão utilizada para os testes.

Implementou-se exatamente o mesmo experimento utilizando as novas instruções da NoC. A parte principal do código é apresentada à seguir:

```

1 // from file "sw/pipeline/stage.c"
2 ...
3 extern uint32_t data[4][512];
4 ...
5 void * stage(void *arg) {
6     ...
7     cpu = sched_getcpu();
8     tparams = (thread_params *) arg;
9     ...
10    if (tparams->stage == 0) {
11        // Initial producer stage
12        for (i = 0; i < tparams->loop_size; ++i) {
13            // fill vector
14            for (j = 0; j < tparams->vec_size; ++j) {
15                data[cpu][j] = int_rand();
16            }
17            // simulate processing delay
18            inc_vec(tparms->load, data[cpu], tparams->vec_size);
19            // send data to next stage
20            k = tparams->prod_to;
21            for (j = 0; j < tparams->vec_size; ++j) {
22                ASMSend(k, data[cpu][j]);
23            }

```



```

24     }
25   } else if (tparams->stage == -1) {
26     // Last consumer stage
27     for (i = 0; i < tparams->loop_size; ++i) {
28       // get data from last stage
29       k = tparams->cons_from;
30       for (j = 0; j < tparams->vec_size; ++j) {
31         ASMLRECV(k, data[cpu][j]);
32       }
33       // simulate processing delay
34       inc_vec(tparams->load, data[cpu], tparams->vec_size);
35     }
36   } else {
37     // other stages
38     for (i = 0; i < tparams->loop_size; ++i) {
39       // get data from last stage
40       k = tparams->cons_from;
41       for (j = 0; j < tparams->vec_size; ++j) {
42         ASMLRECV(k, data[cpu][j]);
43       }
44       // simulate processing delay
45       inc_vec(tparams->load, data[cpu], tparams->vec_size);
46       // send data to next stage
47       k = tparams->prod_to;
48       for (j = 0; j < tparams->vec_size; ++j) {
49         ASMLSEND(k, data[cpu][j]);
50       }
51     }
52   }
53   ...
54 }

```

Como não é mais necessário o uso de memória compartilhada e, consequentemente, dos vetores de transporte, utiliza-se menos memória, no caso, por volta de 42% a menos para a aplicação utilizando a NoC.

Os dois códigos são propositalmente muito similares, com exceção do uso de *locks*, para deixar clara a comparação entre as duas abordagens.

Para ambos os casos, no momento da criação das *threads* de cada estágio, força-se que a execução do estágio 0 seja sempre na CPU0, igualando o parâmetro **stage** a 0. O mesmo vale para as outras três CPUs, onde a CPU3 será o último estágio, totalizando 4 estágios no *pipeline*. Os tempos são tomados antes da criação e depois do término das *threads* e foram normalizados em relação ao maior deles para cada conjunto de medidas apresentada nos gráficos a seguir. A quantidade de dados produzida também é sempre constante em todos os casos e determinada pela multiplicação dos parâmetros **loop_size** e **vec_size**.

Inicialmente mantemos o valor do parâmetro **vec_size** fixo em 32, o que significa que o vetor de transporte tem 32 *words* (128 bytes). Variando a carga, parâmetro **load**, temos o gráfico na Figura 4.5.

Nesse caso, observa-se que, quanto menor a carga, maior o ganho ao se utilizar as

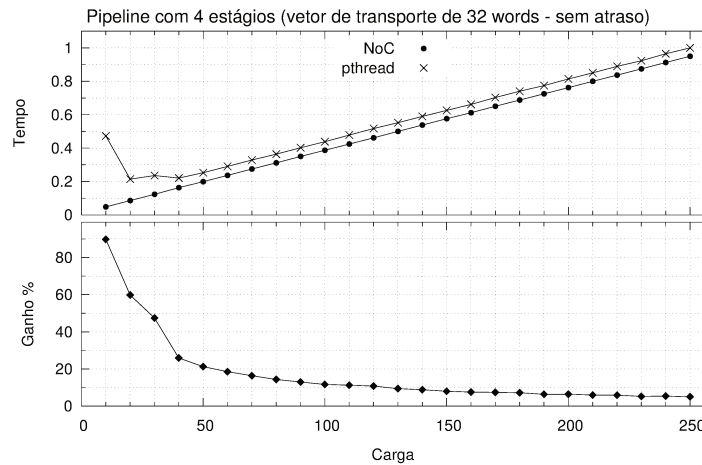


Figura 4.5: Comparação entre tempos de execução utilizando a NoC e utilizando *pthread-read_lock*, variando a carga dos estágios do *pipeline* e com vetor de transporte de tamanho fixo em 32 *words* (128 bytes). Nenhum atraso na memória. Ganho relativo da NoC, em porcentagem, no gráfico inferior (Tabela B.1).

novas instruções. Isso se dá devido ao baixo aproveitamento do *slack* de tempo cedido pelo sistema operacional pelas *threads* utilizando *locks*, pois sua tendência de escalonar, ao encontrar vetores não disponíveis, é muito maior e a troca de contexto é muito cara. Isso não acontece no caso em que se usa as instruções **SEND** e **RECV**, pois estas bloqueiam a *thread* numa espera ocupada até que a mesma utilize todo o seu tempo, minimizando a troca de contexto. O ganho chega a ser de 90% para uma carga muito pequena.

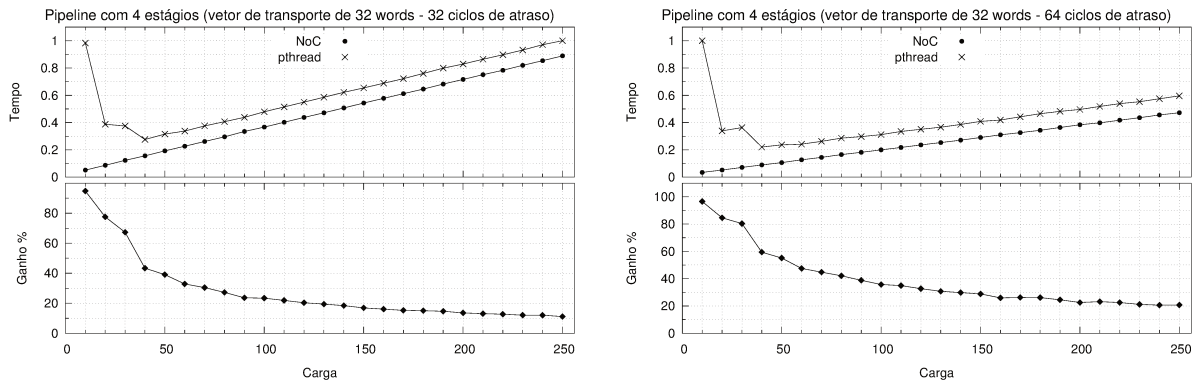
A medida que aumenta-se a carga em cada estágio, o ganho diminui dado que a ocupação dos estágios aumenta e a troca de contexto diminui na aplicação utilizando *locks*. O ganho do novo mecanismo aparece no final da curva, onde o ganho fica constante e então fica evidente o custo de acesso à memória externa. Nesse caso temos um ganho constante de aproximadamente 5%.

Para deixar mais claro o ganho na constante, repetiu-se o mesmo experimento adicionando-se 32 e 64 ciclos de atraso na memória, como mostrado nos gráficos da Figura 4.6.

Para a Figura 4.6(a), onde foram inseridos 32 ciclos de atraso, percebe-se um aumento do ganho no tempo de execução de 6% na parte constante em relação ao caso anterior, totalizando 11%. Para o atraso de 64 ciclos na Figura 4.6(b), o ganho total na parte constante é de aproximadamente 21%.

Repete-se o experimento para *vec_size* fixo em 256 (1kB) e tem-se os resultados mostrados nos gráficos da Figura 4.7.

Para a memória sem atraso adicional, na Figura 4.7(a), o ganho na constante está por volta de 3,0%. Aumentando o atraso para 32 ciclos (Figura 4.7(b)), o ganho é de 7,6%.



(a) 32 ciclos de atraso na memória (Tabela B.2).

(b) 64 ciclos de atraso na memória (Tabela B.3).

Figura 4.6: Comparação entre tempos de execução utilizando a NoC e utilizando *pthread_lock*, variando a carga dos estágios do *pipeline* e com vetor de transporte de tamanho fixo em 32 words (128 bytes). Ganho relativo da NoC, em porcentagem, nos gráficos inferiores.

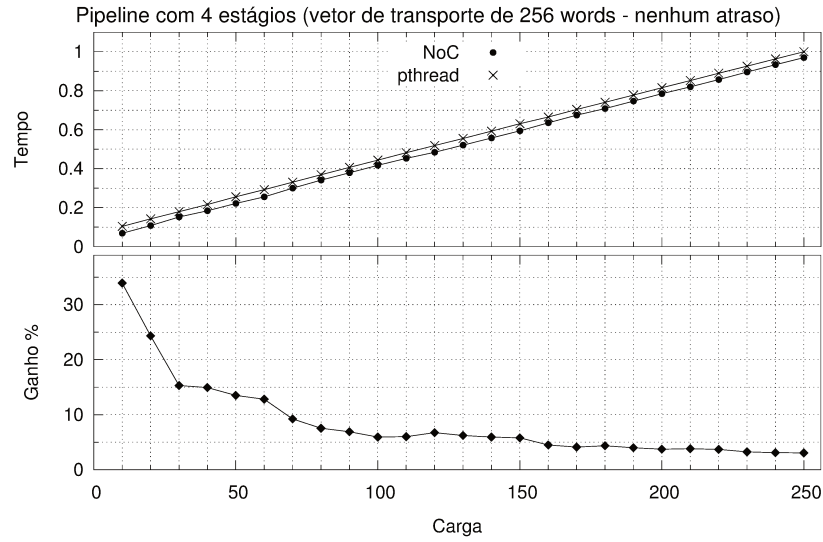
Para 64 ciclos (Figura 4.7(c)), o ganho é de 14,8%. Como os vetores de transporte são maiores que no primeiro caso, o tempo para processá-los também é maior, o que traz uma vantagem ao se utilizar *locks* pela diminuição nas trocas de contexto. Por isso o ganho relativo é diminuído, pois a alteração no tamanho do vetor de transporte, no caso do uso da NoC, apenas aumenta a latência entre o transporte de vetores consecutivos. Mesmo assim, a diferença nos ciclos de atraso ainda é expressiva.

Deseja-se também explorar a influência do tamanho do vetor de transporte (parâmetro *vec.size*) para uma carga constante. Escolheu-se o ponto de pior caso de ganho mostrado nos gráficos anteriores onde a carga tem valor de 250. Os resultados são apresentados nos gráficos da Figura 4.8.

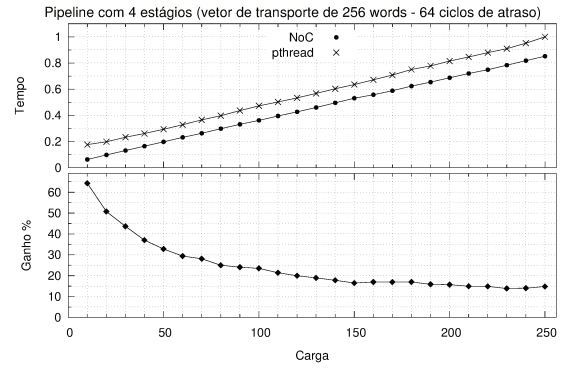
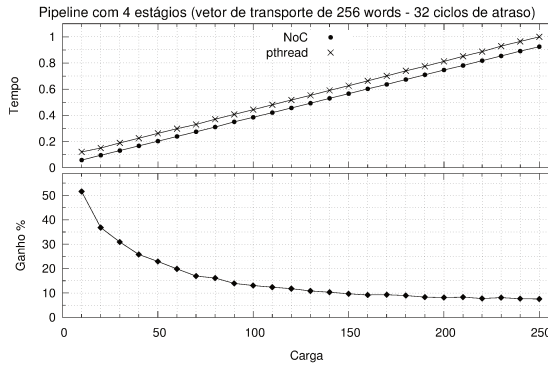
Como a carga é constante e a quantidade total de dados transportados entre os estágios também é, o tempo de execução da aplicação que usa a NoC é independente da quantidade de dados transportados a cada iteração e é constante em todos os casos. Para vetores pequenos, desvantagens aparecem ao se utilizar *locks* pelos mesmos motivos apresentados nos primeiros gráficos, onde o custo de escalonamento é alto.

Como esperado, os ganhos quando os tempos são constantes são muito próximos dos apresentados nos gráficos da Figura 4.7, sendo de 3,0% para 4.7(a), 7,8% para 4.7(b) e 13,3% para 4.7(c).

Os testes dessa seção mostram o quão dependente o mecanismo é do desempenho da memória quando comparado ao uso de *locks*. Desta forma, infere-se que o potencial ganho com o uso desse novo modelo, em sistemas onde o custo de acesso à memória principal é ainda maior, é grande.

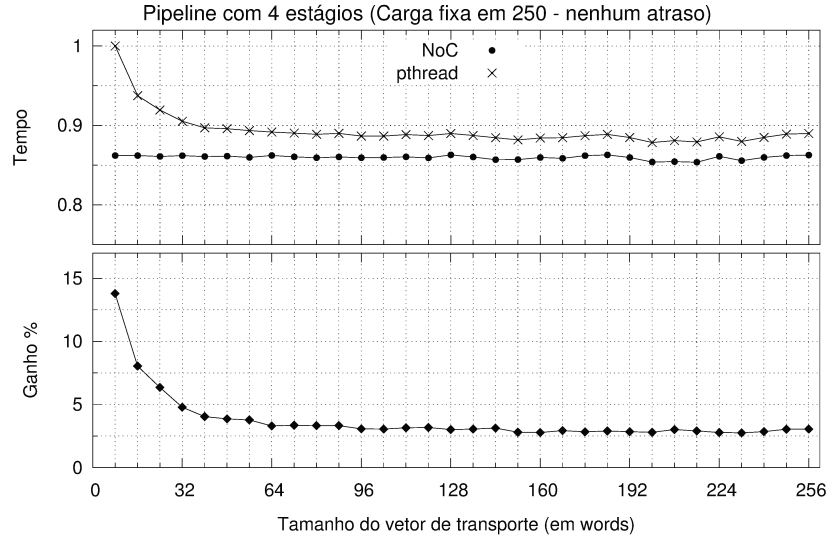


(a) Nenhum atraso na memória (Tabela B.4).

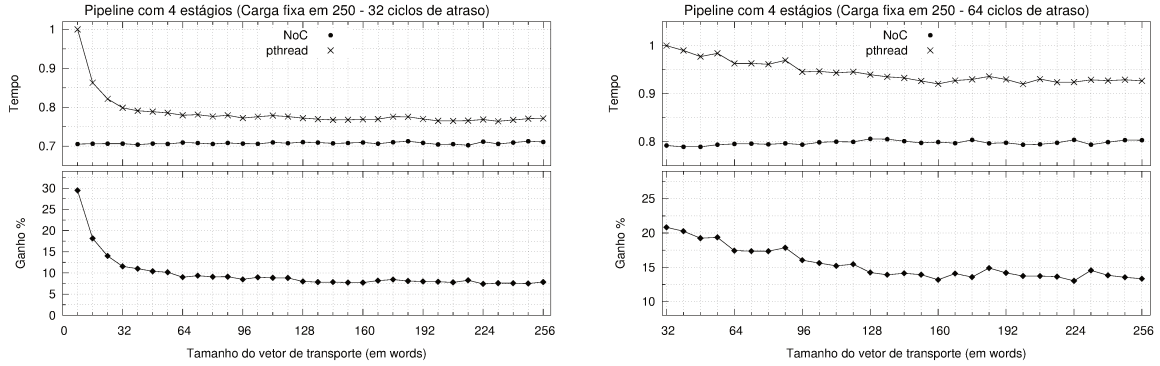


(b) 32 ciclos de atraso na memória (Tabela B.5). (c) 64 ciclos de atraso na memória (Tabela B.6).

Figura 4.7: Comparação entre tempos de execução utilizando a NoC e utilizando *pthread_lock*, variando a carga dos estágios do *pipeline* e com vetor de transporte de tamanho fixo em 256 *words* (1kB). Ganho relativo da NoC, em porcentagem, nos gráficos inferiores.



(a) Nenhum atraso na memória (Tabela B.7).



(b) 32 ciclos de atraso na memória (Tabela B.8). (c) 64 ciclos de atraso na memória (Tabela B.9).

Figura 4.8: Comparação entre tempos de execução utilizando a NoC e utilizando *pthread_lock*, variando o tamanho do vetor de transporte e a carga dos estágios do *pipeline* fixa em 250. Ganho relativo da NoC, em porcentagem, nos gráficos inferiores.

4.6 Testes de aplicação em *Ring*

Essa aplicação é muito similar à aplicação em *pipeline* apresentada na Seção 4.5, com a diferença que o último estágio realimenta o primeiro com os dados que produz ao invés de descartá-los. Da mesma forma que a aplicação em *pipeline*, existe uma carga associada a cada *thread* e um tamanho parametrizável de vetor de transporte, mas aqui utiliza-se um vetor a mais para envio de dados da CPU3 para a CPU0.

Todos os resultados para essa aplicação foram muito similares aos obtidos na seção anterior com ganhos variando de 0,7% a 1,2% dos apresentados anteriormente.

Apresentados todos os testes executados, percebe-se que o potencial de ganho com a utilização desse novo mecanismo é grande. O texto segue no próximo capítulo apresentando as conclusões finais e possibilidades de trabalhos futuros baseados nesse projeto.

Capítulo 5

Conclusão

Tendo como principais objetivos a implementação e avaliação de um novo modelo de comunicação entre processadores através de instruções específicas, avaliação desse modelo, suporte no compilador e resolução da anomalia de frequência da memória, esse trabalho foi bem sucedido, chegando a um protótipo sintetizável do novo modelo de comunicação integrado ao processador LEON3. A arquitetura da NoC implementada minimiza o *overhead* adicional de uma interconexão e apresenta resultados muito promissores mesmo para uma pequena quantidade de processadores.

Devido a limitações no tamanho da FPGA e na anomalia relativa a diferença de *clocks* entre o processador e a memória, um grande esforço foi despendido na adaptação a essas limitações, o que afetou o tempo dedicado à avaliação do sistema, optando-se por aplicações sintéticas que intuitivamente apresentariam ganhos notáveis. Isso foi mostrado nas análises do Capítulo 4, onde sempre consegue-se algum ganho ao se utilizar o novo modelo proposto. Nos casos onde ciclos de atraso são inseridos na memória, isso fica ainda mais evidente, chegando-se a ganhos de até 96% em relação ao uso de memória compartilhada. Ganhos de pior caso variam entre 3% e 21% nos testes realizados. Para o caso produtor-consumidor, o ganho foi de até 70 vezes.

Isso indica que uma nova abordagem de exposição de instruções para a comunicação entre processadores tem um grande potencial a ser explorado e esse trabalho cumpre com o objetivo de gerar um protótipo e uma avaliação inicial dos resultados. Sendo toda a implementação aberta, tanto de *hardware* como de *software*, há uma vasta possibilidade para a exploração de suas potencialidades com modificações e melhorias de forma geral.

5.1 Contribuições deste trabalho

São destacadas aqui as principais contribuições desse trabalho:

- Protótipo sintetizável e altamente configurável de um novo modelo de comunicação entre processadores.
- Versão modularizada e modificável de uma NoC para viabilizar o mecanismo, permitindo fácil modificação para uso de outras redes.
- Estudo detalhado do *pipeline* do processador.
- Código aberto e disponível de toda a implementação para referência em projetos futuros.

Além do artigo escrito e apresentado oralmente no ERAD-SP de 2012, pretende-se resumir os resultados aqui apresentados num segundo artigo, no formato padrão de 6 páginas, para alguma conferência na área, como o WSCAD. Serão adicionados neste os resultados da modificação do decodificador de MP3 dist10 [10] paralelizado e utilizando os recursos aqui propostos. Essa implementação será em breve concluída e servirá como exemplo prático de aplicação do modelo.

5.2 Trabalhos futuros

Além da implementação de maneiras mais simples e automáticas para se usar os novos recursos (com um suporte mais elaborado de compilação), destacam-se as seguintes possíveis continuações desse trabalho:

- Elaboração de novos testes e adaptação de *benchmarks* paralelos comerciais ao novo modelo.
- Adaptação do sistema a FPGAs maiores, abrindo possibilidade para a avaliação de ganho com um número maior de CPUs.
- Adaptação do modelo para outras CPUs onde a anomalia de memória não ocorra, o que reafirmaria os resultados aqui previstos.
- Modificações na topologia e algoritmo de roteamento utilizados.
- Análise de impacto no consumo de potência da adição dos recursos de comunicação.

Referências Bibliográficas

- [1] *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998.
- [2] Amba specification (rev. 2). May 1999.
- [3] Marco A. Z. Alves and Philippe O. A. Navaux. Investigation through set associativity on shared l2 cache memory in a multi-core chip architecture. *VI Workshop de Processamento Paralelo e Distribuído, Porto Alegre.*, 2008.
- [4] Marco Antonio Zanata Alves. Avaliação do compartilhamento das memórias cache no desempenho de arquiteturas multi-core. Master's thesis, Universidade Federal do Rio Grande do Sul - UFRGS, 2009.
- [5] Don Anderson. *HyperTransport Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [6] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. volume 37, pages 44–55. July 2004.
- [7] Altera Corporation. Stratix III Documentation. <http://www.altera.com/literature/lit-stx3.jsp>, 2012.
- [8] Altera Corporation. Stratix III FPGA Development Kit. <http://www.altera.com/products/devkits/altera/kit-siii-host.html>, 2012.
- [9] Intel Corporation. An introduction to the intel quickpath interconnect. 2009.
- [10] dist10. dist10 MP3 Decoder by ISO MPEG Audio Subgroup Software Simulation Group, 1996.
- [11] K. Farkas, Z. Vranesic, and M. Stumm. Cache consistency in hierarchical-ring-based multiprocessors. In *Supercomputing '92. Proceedings*, pages 348–357, Nov 1992.

- [12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafae, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. recognized as Best Paper by the program committee.
- [13] Free Software Foundation. Gnu general public license v.2, June 1991. <http://www.gnu.org/licenses/gpl-2.0.txt>.
- [14] Henrique C. Freitas and Philippe Navaux. Networks-on-chip: Conceitos, arquiteturas e tendências. In *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2009.
- [15] Jiri Gaisler. The Leon3 Processor. <http://www.gaisler.com>, 2008.
- [16] Gaisler Research. *GRLIB IP Core User's Manual*, 1.0.20 edition, 2009.
- [17] P. Gratz, Changkyu Kim, R. McDonald, S.W. Keckler, and D. Burger. Implementation and evaluation of on-chip network architectures. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 477–484, Oct. 2006.
- [18] Pan Hao, Hong Qi, Du Jiaqin, and Pan Pan. Comparison of 2d mesh routing algorithm in noc. In *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pages 791 –795, oct. 2011.
- [19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, September 2006.
- [20] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, 2009.
- [21] Intel Corporation. *The SCC Programmer's Guide*, revision 0.70 edition, August 2010.
- [22] Intel Labs. *RCCE: a Small Library for Many-Core Communication*, 0.72 edition, May 2010.
- [23] Intel Labs. *The SCC Platform Overview*, revision 0.75 edition, September 2010.
- [24] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182, Washington, DC, USA, 2007. IEEE Computer Society.

- [25] John Kim, William Dally, Steve Scott, and Dennis Abts. Cost-efficient dragonfly topology for large-scale systems. *IEEE Micro*, 29(1):33–40, 2009.
- [26] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10–23, May-June 2006.
- [27] Peter Korsgaard. Buildroot. <http://buildroot.uclibc.org/>, 2012.
- [28] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.
- [29] M.M. Michael and A.K. Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 142–151, Jan 1999.
- [30] Sudeep Parischa and Nikil Dutt. *On-chip Communication Architectures: System on Chip interconnect*. Addison Wesley, 1st edition, 2008.
- [31] Sandeep S. GCC-Inline-Assembly-HOWTO. <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>, 2003.
- [32] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M.S. Govindan, P. Gratz, D. Gulati, H. Hanson, Changkyu Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S.W. Keckler, and D. Burger. Distributed microarchitectural protocols in the trips prototype processor. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 480–491, dec. 2006.
- [33] SPARC. *SPARC V8 Architecture Manual*. SPARC International Inc., revision sav080si9308 edition, 1992.
- [34] K. Strauss, Xiaowei Shen, and J. Torrellas. Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 327–338, 0-0 2006.
- [35] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, jan. 2002.
- [36] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition)*. Addison Wesley, 3rd edition, May 2004.

- [37] Wang Zhang, Ligang Hou, Jinhui Wang, Shuqin Geng, and Wuchen Wu. Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip. In *Intelligent Systems, 2009. GCIS '09. WRI Global Congress on*, volume 3, pages 329 –333, may 2009.

Apêndice A

Referência dos Arquivos de Projeto

Listagem dos principais arquivos de implementação com um resumo de seu conteúdo.

Diretório / Arquivo	Descrição
bcc-patch/	Arquivos de referência para modificação do <i>binutils</i> (4.1).
grrlib-gpl-1.1.0-b4113/	Arquivos originais GRRlib da Gaisler.
noc.lib/	Módulos de <i>hardware</i> específicos do projeto.
noc.lib/lsc/debugnoc/	Relativo a <i>debug</i> do processador.
noc.lib/lsc/leon3noc/	Arquivos do processador LEON3 modificado.
noc.lib/lsc/leon3noc/iu3.vhd	Descrição do <i>pipeline</i> do processador (<i>Integer Unit</i>).
noc.lib/lsc/memdelay/	Módulo de inserção de atraso na memória.
noc.lib/lsc/mem.lib/	Descrição das FIFOs e memórias internas.
noc.lib/lsc/routernoc/	Roteador, conexões e interfaces.
noc.lib/lsc/routernoc/cpuinterface.vhd	FIFOs de interface com a CPU.
noc.lib/lsc/routernoc/fifoio.vhd	FIFOs de entrada e saída das portas do roteador.
noc.lib/lsc/routernoc/noc.connection.vhd	Definições de topologia e conexão da NoC.
noc.lib/lsc/routernoc/router.vhd	Descrição completa do roteador.
noc.lib/lsc/routernoc/router_pkg.vhd	Pacote contendo componentes e configurações da NoC.
noc.lib/lsc/sparcnoc/	Mnemônicos e opcodes das novas instruções.
noc.top/	<i>Top-level design</i> do sistema utilizado no projeto.
noc.top/config.fpga.sdc	Configurações e geração seletiva de módulos de <i>hardware</i> para FPGA.
noc.top/leon3mp.2x2.timer_delay.sof	Imagem para programação da FPGA utilizada nos testes.
noc.top/leon3mp.sdc	<i>Design Constraints</i> .
noc.top/leon3mp.vhd	Arquivo <i>top-level</i> .
noc.top/Makefile	<i>Makefile</i> para compilação (<i>make quartus</i> executa a compilação para FPGA Stratix III).
noc.top/sw/	Aplicações sintéticas para testes.
noc.top/sw/memtest/	Testes do modelo de atras da memória (4.2).
noc.top/sw/noctp/	Testes de <i>throughput</i> da NoC (4.3).
noc.top/sw/pc/	Testes do tipo Produtor-Consumidor (4.4).
noc.top/sw/pipeline/	Testes do tipo <i>Pipeline</i> (4.5).
noc.top/sw/plot/	<i>Scripts</i> para geração de gráficos.
noc.top/sw/ring/	Testes do tipo <i>Ring</i> (4.6).

Tabela A.1: Listagem e descrição dos arquivos de projeto.

Apêndice B

Tabelas de Resultados dos Gráficos

O gráfico em 4.5 é gerado a partir da tabela B.1.

Load	NoC	NoC Error	pthread	pthread Error	Gain (%)
10	0.05	0.00	0.47	0.01	89.81
20	0.09	0.00	0.21	0.01	59.81
30	0.12	0.00	0.24	0.01	47.46
40	0.16	0.00	0.22	0.00	25.92
50	0.20	0.00	0.25	0.00	21.23
60	0.24	0.00	0.29	0.00	18.56
70	0.28	0.00	0.33	0.00	16.35
80	0.31	0.00	0.36	0.00	14.35
90	0.35	0.00	0.40	0.00	12.97
100	0.39	0.00	0.44	0.00	11.65
110	0.42	0.00	0.48	0.00	11.25
120	0.46	0.00	0.52	0.00	10.79
130	0.50	0.00	0.55	0.00	9.40
140	0.54	0.00	0.59	0.00	8.82
150	0.58	0.00	0.63	0.00	8.03
160	0.61	0.00	0.66	0.00	7.53
170	0.65	0.00	0.70	0.00	7.42
180	0.69	0.00	0.74	0.00	7.18
190	0.72	0.00	0.77	0.00	6.38
200	0.76	0.00	0.81	0.00	6.37
210	0.80	0.00	0.85	0.00	5.95
220	0.84	0.00	0.89	0.00	5.87
230	0.87	0.00	0.92	0.00	5.27
240	0.91	0.00	0.96	0.00	5.39
250	0.95	0.00	1	0.00	5.03

Tabela B.1: Testes para aplicação em *pipeline* variando a carga. Vetor de transporte com 32 *words*. Sem atraso na memória.

O gráfico em 4.6(a) é gerado a partir da tabela B.2.

Load	NoC	NoC Error	pthread	pthread Error	Gain (%)
10	0.05	0.00	0.98	0.04	94.74
20	0.09	0.00	0.39	0.02	77.53
30	0.12	0.00	0.38	0.02	67.23
40	0.16	0.00	0.28	0.00	43.33
50	0.19	0.00	0.32	0.00	39.07
60	0.23	0.00	0.34	0.00	32.85
70	0.26	0.00	0.38	0.00	30.45
80	0.30	0.00	0.41	0.00	27.23
90	0.33	0.00	0.44	0.00	23.62
100	0.37	0.00	0.48	0.00	23.41
110	0.40	0.00	0.51	0.00	21.88
120	0.44	0.00	0.55	0.00	20.35
130	0.47	0.00	0.59	0.00	19.43
140	0.51	0.00	0.62	0.00	18.43
150	0.54	0.00	0.65	0.00	16.92
160	0.58	0.00	0.69	0.00	16.10
170	0.61	0.00	0.72	0.00	15.25
180	0.65	0.00	0.76	0.00	15.04
190	0.68	0.00	0.80	0.00	14.65
200	0.72	0.00	0.83	0.00	13.58
210	0.75	0.00	0.86	0.00	13.05
220	0.78	0.00	0.90	0.00	12.71
230	0.82	0.00	0.93	0.00	12.04
240	0.85	0.00	0.97	0.00	12.03
250	0.89	0.00	1	0.00	11.13

Tabela B.2: Testes para aplicação em *pipeline* variando a carga. Vetor de transporte com 32 *words*. Atraso de 32 ciclos na memória.

O gráfico em 4.6(b) é gerado a partir da tabela B.3.

Load	NoC	NoC Error	pthread	pthread Error	Gain (%)
10	0.03	0.00	1	0.03	96.55
20	0.05	0.00	0.34	0.02	84.58
30	0.07	0.00	0.36	0.05	80.37
40	0.09	0.00	0.22	0.01	59.40
50	0.11	0.00	0.24	0.01	55.10
60	0.13	0.00	0.24	0.01	47.49
70	0.14	0.00	0.26	0.00	44.77
80	0.17	0.00	0.29	0.00	42.09
90	0.18	0.00	0.30	0.01	38.79
100	0.20	0.00	0.31	0.00	35.74
110	0.22	0.00	0.33	0.00	34.93
120	0.24	0.00	0.35	0.00	32.66
130	0.25	0.00	0.37	0.00	30.77
140	0.27	0.00	0.39	0.00	29.78
150	0.29	0.00	0.41	0.00	28.83
160	0.31	0.00	0.42	0.00	25.95
170	0.33	0.00	0.44	0.00	26.22
180	0.34	0.00	0.46	0.00	26.12
190	0.36	0.00	0.48	0.00	24.51
200	0.38	0.00	0.50	0.00	22.54
210	0.40	0.00	0.52	0.00	23.16
220	0.42	0.00	0.54	0.00	22.52
230	0.44	0.00	0.55	0.00	21.17
240	0.46	0.00	0.57	0.00	20.60
250	0.47	0.00	0.60	0.00	20.75

Tabela B.3: Testes para aplicação em *pipeline* variando a carga. Vetor de transporte com 32 *words*. Atraso de 64 ciclos na memória.

O gráfico em 4.7(a) é gerado a partir da tabela B.4.

Load	NoC	NoC Error	pthread	pthread Error	Gain (%)
10	0.07	0.00	0.10	0.00	33.92
20	0.11	0.00	0.14	0.00	24.35
30	0.15	0.00	0.18	0.00	15.30
40	0.18	0.00	0.22	0.00	14.93
50	0.22	0.00	0.26	0.00	13.51
60	0.26	0.00	0.29	0.00	12.80
70	0.30	0.00	0.33	0.00	9.23
80	0.34	0.00	0.37	0.00	7.53
90	0.38	0.00	0.41	0.00	6.89
100	0.42	0.00	0.44	0.00	5.94
110	0.45	0.00	0.48	0.00	6.00
120	0.48	0.00	0.52	0.00	6.74
130	0.52	0.00	0.56	0.00	6.21
140	0.56	0.00	0.59	0.00	5.94
150	0.59	0.00	0.63	0.00	5.79
160	0.64	0.00	0.67	0.00	4.48
170	0.67	0.00	0.70	0.00	4.12
180	0.71	0.00	0.74	0.00	4.35
190	0.75	0.00	0.78	0.00	3.99
200	0.78	0.00	0.82	0.00	3.72
210	0.82	0.00	0.85	0.00	3.81
220	0.86	0.00	0.89	0.00	3.69
230	0.90	0.00	0.93	0.00	3.25
240	0.93	0.00	0.96	0.00	3.10
250	0.97	0.00	1	0.00	3.04

Tabela B.4: Testes para aplicação em *pipeline* variando a carga. Vetor de transporte com 256 *words*. Sem atraso na memória.

O gráfico em 4.7(b) é gerado a partir da tabela B.5.

Load	NoC	NoC Error	pthread	pthread Error	Gain (%)
10	0.06	0.00	0.12	0.00	51.58
20	0.09	0.00	0.15	0.00	36.82
30	0.13	0.00	0.19	0.00	30.92
40	0.17	0.00	0.23	0.00	25.79
50	0.20	0.00	0.26	0.00	22.93
60	0.24	0.00	0.30	0.00	19.90
70	0.27	0.00	0.33	0.00	16.96
80	0.31	0.00	0.37	0.00	16.13
90	0.35	0.00	0.41	0.00	13.94
100	0.39	0.00	0.44	0.00	13.07
110	0.42	0.00	0.48	0.00	12.39
120	0.46	0.00	0.52	0.00	11.79
130	0.49	0.00	0.55	0.00	10.87
140	0.53	0.00	0.59	0.00	10.34
150	0.57	0.00	0.63	0.00	9.71
160	0.60	0.00	0.66	0.00	9.23
170	0.64	0.00	0.70	0.00	9.29
180	0.67	0.00	0.74	0.00	8.97
190	0.71	0.00	0.77	0.00	8.32
200	0.75	0.00	0.81	0.00	8.14
210	0.78	0.00	0.85	0.00	8.31
220	0.82	0.00	0.89	0.00	7.77
230	0.85	0.00	0.93	0.00	8.12
240	0.89	0.00	0.96	0.00	7.68
250	0.92	0.00	1	0.00	7.56

Tabela B.5: Testes para aplicação em *pipeline* variando a carga. Vetor de transporte com 256 *words*. Atraso de 32 ciclos na memória.

O gráfico em 4.7(c) é gerado a partir da tabela B.6.

Load	NoC	NoC Error	pthread	pthread Error	Gain (%)
10	0.06	0.00	0.18	0.01	64.31
20	0.10	0.00	0.20	0.00	50.73
30	0.13	0.00	0.23	0.00	43.66
40	0.16	0.00	0.26	0.00	37.04
50	0.20	0.00	0.29	0.00	32.78
60	0.23	0.00	0.33	0.00	29.40
70	0.26	0.00	0.37	0.00	28.12
80	0.30	0.00	0.40	0.00	25.00
90	0.33	0.00	0.44	0.00	24.09
100	0.36	0.00	0.47	0.00	23.52
110	0.39	0.00	0.50	0.01	21.39
120	0.43	0.00	0.53	0.00	19.98
130	0.46	0.00	0.57	0.00	18.96
140	0.50	0.00	0.60	0.00	17.81
150	0.53	0.00	0.64	0.00	16.49
160	0.56	0.00	0.67	0.00	16.99
170	0.59	0.00	0.71	0.00	16.94
180	0.62	0.00	0.75	0.00	16.99
190	0.65	0.00	0.78	0.01	15.90
200	0.69	0.00	0.81	0.00	15.69
210	0.72	0.00	0.85	0.01	14.94
220	0.75	0.00	0.88	0.00	14.89
230	0.78	0.00	0.91	0.00	13.88
240	0.82	0.00	0.95	0.00	14.02
250	0.85	0.00	1	0.00	14.81

Tabela B.6: Testes para aplicação em *pipeline* variando a carga. Vetor de transporte com 256 *words*. Atraso de 64 ciclos na memória.

O gráfico em 4.8(a) é gerado a partir da tabela B.7.

Vector Size (words)	NoC	NoC Error	pthread	pthread Error	Gain (%)
8	0.86	0.00	1	0.00	13.80
16	0.86	0.00	0.94	0.00	8.04
24	0.86	0.00	0.92	0.00	6.36
32	0.86	0.00	0.91	0.00	4.79
40	0.86	0.00	0.90	0.00	4.04
48	0.86	0.00	0.90	0.00	3.86
56	0.86	0.00	0.89	0.00	3.78
64	0.86	0.00	0.89	0.00	3.30
72	0.86	0.00	0.89	0.00	3.35
80	0.86	0.00	0.89	0.00	3.33
88	0.86	0.00	0.89	0.00	3.33
96	0.86	0.00	0.89	0.00	3.06
104	0.86	0.00	0.89	0.00	3.05
112	0.86	0.00	0.89	0.00	3.15
120	0.86	0.00	0.89	0.00	3.18
128	0.86	0.00	0.89	0.00	3.01
136	0.86	0.00	0.89	0.00	3.05
144	0.86	0.00	0.88	0.00	3.13
152	0.86	0.00	0.88	0.00	2.80
160	0.86	0.00	0.88	0.00	2.78
168	0.86	0.00	0.88	0.00	2.92
176	0.86	0.00	0.89	0.00	2.84
184	0.86	0.00	0.89	0.00	2.90
192	0.86	0.00	0.88	0.00	2.84
200	0.85	0.00	0.88	0.00	2.80
208	0.85	0.00	0.88	0.00	3.01
216	0.85	0.00	0.88	0.00	2.90
224	0.86	0.00	0.89	0.00	2.78
232	0.86	0.00	0.88	0.00	2.75
240	0.86	0.00	0.88	0.00	2.85
248	0.86	0.00	0.89	0.00	3.04
256	0.86	0.00	0.89	0.00	3.06

Tabela B.7: Testes para aplicação em *pipeline* variando o tamanho do vetor de transporte com carga fixa de estágio de 250. Sem atraso na memória.

O gráfico em 4.8(b) é gerado a partir da tabela B.8.

Vector Size (words)	NoC	NoC Error	pthread	pthread Error	Gain (%)
8	0.71	0.00	1	0.00	29.47
16	0.71	0.00	0.86	0.00	18.18
24	0.71	0.00	0.82	0.00	14.01
32	0.71	0.00	0.80	0.00	11.55
40	0.70	0.00	0.79	0.00	11.01
48	0.71	0.00	0.79	0.00	10.40
56	0.71	0.00	0.79	0.00	10.16
64	0.71	0.00	0.78	0.00	9.00
72	0.71	0.00	0.78	0.00	9.35
80	0.71	0.00	0.78	0.00	9.09
88	0.71	0.00	0.78	0.00	9.12
96	0.71	0.00	0.77	0.00	8.50
104	0.71	0.00	0.78	0.00	8.98
112	0.71	0.00	0.78	0.00	8.86
120	0.71	0.00	0.78	0.00	8.82
128	0.71	0.00	0.77	0.00	8.01
136	0.71	0.00	0.77	0.00	7.84
144	0.71	0.00	0.77	0.00	7.84
152	0.71	0.00	0.77	0.00	7.75
160	0.71	0.00	0.77	0.00	7.73
168	0.71	0.00	0.77	0.00	8.18
176	0.71	0.00	0.78	0.00	8.45
184	0.71	0.00	0.78	0.00	8.09
192	0.71	0.00	0.77	0.00	7.98
200	0.70	0.00	0.76	0.00	7.93
208	0.71	0.00	0.76	0.00	7.78
216	0.70	0.00	0.77	0.00	8.26
224	0.71	0.00	0.77	0.00	7.42
232	0.71	0.00	0.76	0.00	7.61
240	0.71	0.00	0.77	0.00	7.59
248	0.71	0.00	0.77	0.00	7.52
256	0.71	0.00	0.77	0.00	7.84

Tabela B.8: Testes para aplicação em *pipeline* variando o tamanho do vetor de transporte com carga fixa de estágio de 250. Atraso de 32 ciclos na memória.

O gráfico em 4.8(c) é gerado a partir da tabela B.9.

Vector Size (words)	NoC	NoC Error	pthread	pthread Error	Gain (%)
32	0.79	0.00	1	0.00	20.82
40	0.79	0.00	0.99	0.00	20.27
48	0.79	0.00	0.98	0.01	19.25
56	0.79	0.00	0.98	0.00	19.36
64	0.80	0.00	0.96	0.00	17.43
72	0.80	0.00	0.96	0.00	17.36
80	0.79	0.00	0.96	0.01	17.35
88	0.80	0.00	0.97	0.01	17.85
96	0.79	0.00	0.95	0.01	16.04
104	0.80	0.00	0.95	0.01	15.61
112	0.80	0.00	0.94	0.00	15.22
120	0.80	0.00	0.95	0.00	15.45
128	0.81	0.00	0.94	0.00	14.25
136	0.80	0.00	0.93	0.00	13.91
144	0.80	0.00	0.93	0.01	14.13
152	0.80	0.00	0.93	0.00	13.93
160	0.80	0.00	0.92	0.00	13.18
168	0.80	0.00	0.93	0.00	14.09
176	0.80	0.00	0.93	0.00	13.57
184	0.80	0.00	0.94	0.00	14.90
192	0.80	0.00	0.93	0.01	14.19
200	0.79	0.00	0.92	0.01	13.73
208	0.79	0.00	0.93	0.00	13.73
216	0.80	0.00	0.92	0.00	13.64
224	0.80	0.00	0.92	0.01	13.03
232	0.79	0.00	0.93	0.01	14.55
240	0.80	0.00	0.93	0.00	13.83
248	0.80	0.00	0.93	0.01	13.54
256	0.80	0.00	0.93	0.00	13.34

Tabela B.9: Testes para aplicação em *pipeline* variando o tamanho do vetor de transporte com carga fixa de estágio de 250. Atraso de 64 ciclos na memória.

As tabelas B.10, B.11, B.12, B.13 e B.14 são referentes aos gráficos apresentados na seção 4.2.

Size (KB)	Read	Read Error	Write	Write Error	RW	RW Error
1	16.02	0.29	36.51	1.05	27.85	0.41
2	16.32	0.01	38.30	0	28.79	0.04
3	16.15	0.26	37.26	0.78	28.54	0.58
4	15.30	0.75	38.30	0	28.15	0.87
5	15.84	0.50	37.85	0	27.61	1.17
6	15.56	0.54	38.05	0	25.69	1.78
7	15.91	0.44	38.13	0.02	27.98	1.13
8	15.99	0.38	37.15	0.96	28.46	0.90
9	16.34	0.01	35.63	1.67	19.10	0.79
10	15.62	0.50	34.54	1.98	15.86	0.39
11	15.71	0.45	38.15	0.02	14.04	0.07
12	15.10	0.59	38.17	0	12.26	0.28
13	16.36	0.00	35.89	1.54	11.64	0.41
14	15.46	0.49	38.05	0.14	12.32	0.24
15	15.83	0.35	36.71	1.21	12.26	0.29
16	16.10	0.28	36.18	1.39	12.04	0.36
17	16.37	0.00	38.16	0	11.81	0.35
18	15.53	0.46	38.19	0.01	12.13	0.31
19	15.17	0.47	37.28	0.95	12.03	0.30
20	15.62	0.41	38.22	0	12.03	0.31
21	15.66	0.39	38.19	0	12.00	0.29
22	15.75	0.34	35.12	1.74	12.32	0.22
23	15.08	0.45	37.43	0.81	11.81	0.32
24	14.83	0.48	37.05	1.03	12.27	0.24
25	15.56	0.37	35.60	1.31	12.14	0.26
26	15.62	0.35	38.03	0.20	11.95	0.27
27	15.84	0.30	37.21	0.84	12.16	0.26
28	15.57	0.37	38.24	0	11.98	0.26
29	15.15	0.43	37.51	0.71	12.12	0.23
30	15.76	0.33	35.50	1.54	12.57	0.13
31	15.97	0.28	35.93	1.21	12.19	0.24
32	15.88	0.27	34.05	1.48	12.01	0.25
33	16.05	0.22	36.79	0.99	12.38	0.18
34	15.55	0.33	36.37	1.07	12.42	0.16
35	15.76	0.28	36.99	0.87	12.07	0.22
36	15.53	0.34	36.91	0.92	12.09	0.24
37	16.07	0.21	37.62	0.60	12.10	0.22
38	15.66	0.28	35.34	1.41	12.26	0.21
39	15.53	0.30	36.31	1.07	12.18	0.19
40	15.74	0.27	37.47	0.76	12.03	0.19
41	15.43	0.31	36.44	0.98	12.27	0.17
42	15.55	0.37	35.75	1.16	12.21	0.17
43	15.33	0.29	37.72	0.55	12.07	0.20
44	15.74	0.25	36.51	0.95	12.19	0.19
45	15.53	0.30	36.27	1.01	12.22	0.20
46	15.77	0.24	37.03	0.85	11.99	0.18
47	15.75	0.24	36.37	1.04	11.83	0.19
48	15.78	0.24	36.71	0.85	12.06	0.17
49	14.96	0.27	37.74	0.50	12.42	0.13
50	15.28	0.32	36.60	0.90	12.23	0.17
51	15.37	0.26	37.78	0.49	12.24	0.15
52	15.40	0.29	36.74	0.83	12.07	0.17
53	16.26	0.11	36.50	0.97	11.72	0.16
54	15.72	0.23	36.33	0.88	12.40	0.13
55	15.77	0.21	38.26	0.01	11.66	0.18
56	15.42	0.25	38.26	0.00	11.96	0.16

Tabela B.10: Testes da memória DDR2 sem atraso.

Size (KB)	Read	Read Error	Write	Write Error	RW	RW Error
1	13.19	0.01	35.03	1.14	27.17	0.04
2	13.20	0	37.56	0	28.31	0
3	12.67	0.49	38.05	0	28.21	0.66
4	13.20	0	37.92	0	29.08	0.02
5	12.68	0.44	37.56	0	28.49	0.45
6	13.13	0.00	36.86	0.79	29.09	0.02
7	11.58	0.72	36.19	1.54	26.78	1.12
8	12.06	0.60	37.92	0	26.11	1.48
9	12.49	0.45	37.74	0.01	14.60	0.66
10	13.15	0.00	34.97	1.99	11.24	0.29
11	12.55	0.42	35.19	1.89	9.26	0.19
12	12.59	0.40	36.59	1.34	7.79	0.25
13	12.88	0.27	36.58	1.32	8.11	0.14
14	12.94	0.14	36.58	1.35	7.71	0.24
15	11.58	0.55	36.65	1.30	7.99	0.17
16	12.17	0.46	38.02	0	7.65	0.25
17	12.68	0.32	37.93	0.01	7.80	0.21
18	12.30	0.39	36.77	1.19	7.62	0.23
19	12.00	0.46	36.81	1.20	7.89	0.16
20	12.22	0.44	36.94	1.06	8.16	0.10
21	12.24	0.42	36.20	1.26	7.86	0.18
22	12.32	0.39	35.81	1.47	7.68	0.21
23	12.74	0.28	38.00	0	7.48	0.21
24	12.69	0.28	36.90	1.05	7.60	0.19
25	12.77	0.26	35.57	1.68	7.47	0.19
26	12.45	0.32	36.96	1.05	7.80	0.16
27	12.42	0.34	35.97	1.36	7.62	0.18
28	12.99	0.17	36.14	1.32	7.57	0.18
29	12.83	0.23	35.88	1.46	7.26	0.17
30	12.30	0.34	36.18	1.27	7.54	0.16
31	12.67	0.26	37.17	0.86	7.14	0.15
32	12.69	0.26	38.05	0.00	7.57	0.16
33	12.69	0.26	37.15	0.87	7.54	0.15
34	12.25	0.32	38.03	0.00	7.64	0.15
35	12.14	0.32	37.01	0.86	7.68	0.15
36	12.38	0.31	36.32	1.21	7.54	0.15
37	12.21	0.35	37.05	0.82	7.70	0.17
38	12.29	0.31	38.03	0.00	7.52	0.13
39	12.21	0.30	37.06	1.01	7.68	0.14
40	12.03	0.33	36.42	1.13	7.76	0.13
41	12.24	0.29	36.56	1.02	7.67	0.13
42	12.04	0.32	34.88	1.45	7.83	0.12
43	12.13	0.29	38.06	0.00	7.55	0.15
44	12.18	0.28	37.25	0.77	7.72	0.12
45	12.20	0.27	38.04	0.00	7.65	0.12
46	11.86	0.27	35.00	1.40	7.92	0.10
47	11.85	0.28	35.58	1.25	7.93	0.10
48	11.60	0.27	37.37	0.70	7.93	0.11
49	12.34	0.26	37.30	0.76	7.64	0.11
50	11.83	0.25	37.31	0.75	7.94	0.10
51	11.90	0.26	36.65	0.97	7.81	0.10
52	12.30	0.24	35.78	1.28	7.76	0.10
53	12.34	0.23	35.93	1.16	7.75	0.10
54	12.26	0.23	35.37	1.24	7.79	0.10
55	12.37	0.23	36.70	0.95	7.68	0.07
56	12.08	0.22	35.67	1.32	7.87	0.08

Tabela B.11: Testes da memória DDR2 com 16 ciclos de atraso.

Size (KB)	Read	Read Error	Write	Write Error	RW	RW Error
1	8.15	0.34	32.36	2.52	25.70	0
2	8.48	0.01	36.85	0	27.63	0.06
3	8.47	0.00	37.56	0	28.36	0.04
4	7.92	0.57	37.56	0	25.00	2.58
5	8.45	0	36.71	0	28.45	0.03
6	8.45	0.00	33.81	3.27	28.67	0.02
7	8.46	0.00	37.31	0.03	28.83	0.01
8	8.46	0.00	37.38	0	28.88	0.02
9	7.60	0.57	33.21	3.02	10.00	0.67
10	8.41	0.05	34.46	2.93	7.01	0.32
11	8.46	0.00	37.43	0	5.11	0.34
12	8.46	0.00	34.76	2.80	4.35	0.23
13	7.82	0.42	37.43	0.01	4.79	0.13
14	8.15	0.31	37.47	0.01	4.27	0.21
15	7.82	0.43	37.56	0	4.45	0.19
16	7.91	0.37	37.65	0	4.57	0.18
17	8.46	0.00	37.51	0.01	4.27	0.18
18	7.31	0.49	37.56	0	4.62	0.15
19	7.71	0.38	35.07	2.57	4.59	0.17
20	7.98	0.32	37.63	0	4.49	0.18
21	8.01	0.30	37.56	0	4.29	0.19
22	8.00	0.31	37.63	0	4.35	0.16
23	7.61	0.35	37.46	0.20	4.50	0.14
24	8.05	0.27	37.68	0	4.44	0.13
25	8.06	0.27	37.61	0.01	4.41	0.14
26	7.26	0.33	35.05	2.59	4.69	0.12
27	7.82	0.33	35.57	2.09	4.43	0.15
28	7.70	0.31	35.59	2.10	4.54	0.13
29	7.93	0.37	33.58	2.71	4.43	0.09
30	7.92	0.28	33.56	2.73	4.50	0.11
31	7.60	0.29	37.70	0	4.59	0.11
32	7.45	0.28	36.17	1.55	4.61	0.10
33	8.09	0.25	37.65	0	4.32	0.09
34	7.17	0.22	35.70	1.98	4.81	0.07
35	7.76	0.29	31.89	2.96	4.50	0.05
36	7.52	0.26	37.73	0.00	4.58	0.10
37	7.82	0.27	37.67	0.01	4.44	0.06
38	7.40	0.23	37.70	0.01	4.60	0.05
39	7.71	0.25	37.73	0.01	4.54	0.07
40	7.56	0.25	35.86	1.87	4.51	0.06
41	7.80	0.22	34.13	2.38	4.52	0.07
42	7.62	0.23	36.28	1.04	4.49	0.06
43	7.25	0.24	37.30	0.42	4.62	0.06
44	7.59	0.24	36.11	1.64	4.58	0.04
45	7.15	0.15	37.70	0.01	4.56	0.06
46	7.54	0.20	35.71	1.61	4.56	0.05
47	7.62	0.23	36.16	1.57	4.62	0.04
48	7.37	0.19	34.43	2.23	4.67	0.03
49	7.60	0.27	32.64	2.58	4.64	0.02
50	7.46	0.23	34.46	2.17	4.64	0.03
51	7.41	0.18	37.74	0.00	4.62	0.05
52	7.47	0.17	37.75	0.01	4.53	0.07
53	7.51	0.16	37.37	0.35	4.60	0.06
54	7.58	0.20	36.23	1.51	4.61	0.05
55	7.56	0.22	34.38	2.27	4.67	0.03
56	7.36	0.14	36.27	1.49	4.71	0.02

Tabela B.12: Testes da memória DDR2 com 32 ciclos de atraso.

Size (KB)	Read	Read Error	Write	Write Error	RW	RW Error
1	6.01	0.25	33.67	0	22.61	2.43
2	5.57	0.53	36.17	0	19.91	3.71
3	6.29	0.00	36.99	0.06	27.93	0.03
4	6.29	0	37.20	0	28.31	0
5	4.98	0.65	34.75	1.42	23.99	2.19
6	6.26	0	33.25	3.37	27.13	1.24
7	6.27	0.00	32.68	3.29	27.79	0.71
8	5.21	0.54	34.44	1.73	25.25	1.88
9	5.60	0.44	33.51	3.27	7.19	0.64
10	5.95	0.32	30.60	4.26	4.78	0.36
11	5.69	0.39	35.61	1.44	3.69	0.22
12	5.41	0.44	35.23	1.98	3.49	0.01
13	5.46	0.41	37.01	0	3.05	0.17
14	5.48	0.40	34.19	2.96	3.08	0.17
15	5.75	0.35	37.21	0.01	2.78	0.16
16	5.55	0.37	31.27	4.01	3.13	0.15
17	5.06	0.40	33.37	2.95	3.23	0.14
18	4.72	0.40	34.46	2.77	3.28	0.12
19	5.96	0.23	37.31	0.01	2.77	0.08
20	5.36	0.38	34.53	2.81	3.03	0.13
21	5.26	0.42	34.45	2.77	2.96	0.10
22	5.27	0.34	33.72	2.79	3.09	0.11
23	6.08	0.19	33.17	2.99	2.91	0.08
24	5.27	0.33	36.46	0.92	3.13	0.11
25	5.90	0.25	37.27	0	2.86	0.03
26	5.75	0.27	36.52	0.81	2.97	0.08
27	5.06	0.26	37.37	0.01	3.20	0.08
28	5.13	0.25	33.39	2.52	3.13	0.05
29	5.44	0.28	34.76	2.55	3.08	0.09
30	4.75	0.17	37.36	0.01	3.16	0.03
31	5.79	0.24	37.41	0.01	3.03	0.06
32	5.14	0.25	37.43	0	3.05	0.06
33	5.13	0.26	37.34	0	3.11	0.06
34	5.07	0.20	34.96	2.41	3.18	0.04
35	5.22	0.23	36.67	0.64	3.14	0.05
36	5.18	0.24	33.23	2.84	3.27	0.01
37	4.96	0.15	37.37	0.01	3.17	0.05
38	4.99	0.14	37.40	0.00	3.21	0.05
39	5.48	0.22	32.64	3.20	3.16	0.07
40	5.32	0.21	34.69	2.31	3.21	0.05
41	5.09	0.21	32.85	3.02	3.25	0.03
42	5.51	0.21	30.80	3.44	3.13	0.07
43	5.52	0.21	32.98	2.99	3.21	0.03
44	5.29	0.17	37.46	0	3.26	0.03
45	5.08	0.23	33.02	2.92	3.23	0.03
46	5.23	0.11	35.28	2.15	3.20	0.06
47	5.17	0.13	35.31	2.15	3.17	0.06
48	5.20	0.07	33.15	2.88	3.23	0.03
49	5.12	0.06	37.41	0	3.26	0.03
50	5.22	0.05	35.36	2.07	3.18	0.04
51	5.21	0.06	35.42	2.05	3.19	0.04
52	5.31	0.07	35.48	2.00	3.22	0.05
53	5.22	0.05	35.32	2.10	3.24	0.02
54	5.24	0.10	33.24	2.81	3.22	0.02
55	5.39	0.08	29.46	3.28	3.27	0.02
56	5.28	0.12	31.44	3.08	3.25	0.03

Tabela B.13: Testes da memória DDR2 com 48 ciclos de atraso.

Size (KB)	Read	Read Error	Write	Write Error	RW	RW Error
1	4.96	0.00	32.55	0	22.30	1.72
2	4.72	0.25	32.53	2.99	23.33	2.23
3	4.98	0	33.04	3.13	27.38	0
4	3.51	0.60	36.85	0	20.33	2.84
5	4.26	0.46	31.02	3.11	24.33	2.30
6	3.99	0.49	33.18	2.77	17.27	3.69
7	4.36	0.40	36.36	0	24.68	2.38
8	4.40	0.38	33.33	3.34	24.26	2.78
9	4.28	0.36	31.60	3.25	4.97	0.72
10	4.21	0.38	36.49	0.02	2.96	0.31
11	3.97	0.40	36.66	0	2.50	0.23
12	3.83	0.38	33.63	3.21	2.62	0.06
13	4.28	0.34	33.46	3.12	2.23	0.14
14	4.32	0.33	36.75	0	2.15	0.12
15	3.69	0.34	31.64	3.54	2.49	0.10
16	3.93	0.34	34.14	2.80	2.37	0.12
17	4.17	0.32	31.25	3.74	2.31	0.10
18	3.78	0.32	33.84	2.99	2.39	0.05
19	4.06	0.30	35.10	1.86	2.31	0.07
20	4.25	0.28	36.99	0	2.25	0.10
21	3.39	0.18	33.89	2.97	2.45	0.04
22	4.46	0.26	34.33	2.50	2.31	0.05
23	4.33	0.26	37.00	0.01	2.20	0.07
24	4.03	0.25	28.96	3.99	2.39	0.05
25	3.86	0.24	34.05	2.87	2.39	0.04
26	3.64	0.15	34.12	2.84	2.45	0.02
27	4.12	0.23	32.85	3.00	2.40	0.05
28	3.63	0.15	36.66	0.43	2.42	0.05
29	4.12	0.23	34.19	2.78	2.34	0.07
30	3.57	0.06	37.03	0.00	2.51	0.01
31	3.94	0.23	28.87	4.17	2.46	0.03
32	3.81	0.13	34.41	2.70	2.41	0.03
33	3.89	0.19	34.33	2.67	2.41	0.04
34	4.13	0.17	31.93	3.41	2.38	0.04
35	4.12	0.15	31.90	3.45	2.42	0.03
36	3.84	0.06	34.59	2.53	2.43	0.03
37	4.04	0.12	31.91	3.41	2.44	0.02
38	3.73	0.07	37.07	0.00	2.39	0.03
39	3.93	0.11	34.55	2.56	2.39	0.03
40	3.91	0.10	31.82	3.55	2.43	0.02
41	3.96	0.15	31.90	3.38	2.48	0.03
42	3.96	0.09	30.32	3.52	2.44	0.02
43	3.93	0.13	32.10	3.35	2.43	0.04
44	3.91	0.07	34.74	2.42	2.45	0.02
45	4.11	0.07	33.88	2.51	2.41	0.03
46	4.02	0.09	32.28	3.22	2.47	0.03
47	3.94	0.09	34.66	2.43	2.47	0.03
48	3.98	0.11	32.11	2.99	2.45	0.01
49	4.22	0.08	32.22	3.18	2.46	0.01
50	4.15	0.12	33.71	2.48	2.46	0.01
51	4.17	0.10	31.00	3.23	2.45	0.03
52	4.03	0.11	32.55	3.08	2.45	0.02
53	4.03	0.11	30.28	3.48	2.45	0.02
54	4.21	0.09	27.84	3.79	2.46	0.02
55	3.99	0.12	32.61	3.03	2.46	0.03
56	3.96	0.05	34.73	2.29	2.48	0.01

Tabela B.14: Testes da memória DDR2 com 64 ciclos de atraso.

As tabelas B.15 e B.16 são referentes ao gráfico apresentado na figura 4.4.

Size (MB)	To CPU1			To CPU3		
	MB/s	MB/s Error	cycles/word	MB/s	MB/s Error	cycles/word
0.01	0.98	0.04	390.15	0.58	0.01	661.21
0.02	4.84	0.16	78.84	2.85	0.08	133.99
0.03	8.57	0.31	44.52	5.11	0.08	74.72
0.05	12.07	0.41	31.61	7.10	0.11	53.72
0.06	14.92	0.40	25.56	8.93	0.33	42.71
0.08	18.36	0.58	20.78	11.32	0.17	33.69
0.10	20.69	1.12	18.44	13.14	0.17	29.03
0.11	23.96	0.62	15.92	14.70	0.16	25.95
0.13	27.39	0.67	13.93	16.62	0.25	22.96
0.14	28.66	0.76	13.31	18.40	0.28	20.73
0.38	54.28	0.30	7.03	41.02	0.82	9.30
1.14	86.39	1.32	4.42	72.37	0.52	5.27
1.91	99.12	0.25	3.85	86.82	0.98	4.39
2.67	106.03	0.36	3.60	99.06	0.45	3.85
3.43	109.02	0.54	3.50	103.70	0.41	3.68
4.20	112.00	0.12	3.41	105.92	0.67	3.60
4.96	113.26	0.92	3.37	109.55	0.36	3.48
5.72	115.41	0.15	3.31	111.41	0.22	3.42
6.48	116.53	0.39	3.27	113.15	0.23	3.37
7.25	117.53	0.13	3.25	114.16	0.27	3.34
8.01	118.22	0.06	3.23	115.15	0.23	3.31
8.77	118.68	0.27	3.21	116.12	0.23	3.29
9.54	119.29	0.20	3.20	116.71	0.26	3.27
10.30	118.95	0.48	3.21	116.91	0.29	3.26
11.06	120.01	0.30	3.18	117.67	0.16	3.24
11.83	120.11	0.37	3.18	118.14	0.16	3.23
12.59	120.92	0.11	3.15	118.64	0.13	3.22
13.35	120.86	0.34	3.16	119.34	0.18	3.20
14.11	120.84	0.27	3.16	119.36	0.22	3.20
14.88	121.13	0.16	3.15	119.99	0.22	3.18
15.64	121.63	0.26	3.14	119.93	0.35	3.18
16.40	121.74	0.23	3.13	119.99	0.33	3.18
17.17	121.88	0.16	3.13	120.61	0.25	3.16
17.93	122.17	0.13	3.12	120.90	0.24	3.16
18.69	122.51	0.12	3.11	121.03	0.15	3.15
19.45	122.53	0.27	3.11	121.21	0.19	3.15
20.22	122.68	0.10	3.11	121.26	0.18	3.15
20.98	122.72	0.33	3.11	121.67	0.21	3.14
21.74	122.78	0.27	3.11	121.53	0.13	3.14
22.51	123.14	0.16	3.10	121.76	0.15	3.13
23.27	123.02	0.24	3.10	121.89	0.19	3.13
24.03	123.11	0.16	3.10	122.06	0.13	3.13
24.80	123.33	0.16	3.09	122.30	0.18	3.12
25.56	123.08	0.24	3.10	122.35	0.27	3.12
26.32	123.41	0.13	3.09	122.16	0.16	3.12
27.08	123.56	0.17	3.09	122.44	0.18	3.12
27.85	123.69	0.12	3.08	122.47	0.20	3.11
28.61	123.84	0.04	3.08	122.53	0.24	3.11
29.37	123.82	0.12	3.08	123.00	0.05	3.10
30.14	123.81	0.14	3.08	123.15	0.10	3.10
30.52	123.56	0.06	3.09	122.96	0.07	3.10
38.33	124.30	0.12	3.07	123.73	0.12	3.08

Tabela B.15: Testes de *throughput* da NoC sem atraso na memória.

Size (MB)	To CPU1			To CPU3		
	MB/s	MB/s Error	cycles/word	MB/s	MB/s Error	cycles/word
0.01	0.42	0.02	897.76	0.28	0.00	1358.57
0.02	2.07	0.13	184.04	1.37	0.02	277.85
0.03	3.61	0.06	105.63	2.58	0.09	147.65
0.05	5.05	0.06	75.51	3.89	0.16	98.08
0.06	6.59	0.14	57.85	4.69	0.10	81.40
0.08	8.10	0.17	47.10	5.52	0.11	69.13
0.10	9.63	0.13	39.61	6.85	0.31	55.69
0.11	10.67	0.17	35.75	7.59	0.08	50.28
0.13	12.16	0.12	31.37	8.96	0.12	42.58
0.14	13.79	0.64	27.67	9.44	0.13	40.43
0.38	30.82	0.35	12.38	22.53	0.29	16.93
1.14	61.71	1.49	6.18	48.59	0.41	7.85
1.91	76.02	0.76	5.02	65.06	0.08	5.86
2.67	83.31	1.99	4.58	76.02	1.37	5.02
3.43	91.04	0.74	4.19	81.15	0.47	4.70
4.20	93.58	2.48	4.08	86.89	0.33	4.39
4.96	98.61	1.03	3.87	91.51	0.52	4.17
5.72	101.32	1.08	3.76	94.31	0.47	4.04
6.48	103.53	0.46	3.68	98.46	0.41	3.87
7.25	105.54	0.73	3.61	98.76	0.71	3.86
8.01	106.84	0.33	3.57	101.32	0.47	3.77
8.77	108.90	1.08	3.50	103.31	0.50	3.69
9.54	111.86	0.82	3.41	104.38	0.58	3.65
10.30	112.47	0.67	3.39	105.79	0.34	3.61
11.06	112.71	0.60	3.38	107.16	0.25	3.56
11.83	113.94	0.73	3.35	107.65	0.36	3.54
12.59	113.81	0.19	3.35	108.69	0.24	3.51
13.35	114.91	0.71	3.32	109.70	0.16	3.48
14.11	116.34	0.40	3.28	109.65	0.43	3.48
14.88	115.40	0.62	3.31	110.64	0.45	3.45
15.64	116.07	0.37	3.29	111.32	0.19	3.43
16.40	116.76	0.35	3.27	112.22	0.37	3.40
17.17	116.14	0.44	3.28	112.34	0.29	3.40
17.93	117.01	0.27	3.26	112.89	0.69	3.38
18.69	116.69	0.60	3.27	113.54	0.24	3.36
19.45	117.89	0.20	3.24	113.68	0.28	3.36
20.22	117.81	0.50	3.24	113.75	0.38	3.35
20.98	118.29	0.33	3.22	114.19	0.38	3.34
21.74	118.40	0.37	3.22	115.04	0.23	3.32
22.51	118.13	0.23	3.23	115.26	0.19	3.31
23.27	118.38	0.47	3.22	115.05	0.32	3.32
24.03	118.47	0.48	3.22	115.70	0.27	3.30
24.80	118.97	0.37	3.21	115.85	0.25	3.29
25.56	119.23	0.32	3.20	116.19	0.19	3.28
26.32	119.10	0.15	3.20	116.28	0.32	3.28
27.08	119.31	0.26	3.20	116.76	0.49	3.27
27.85	119.24	0.50	3.20	117.25	0.36	3.25
28.61	119.65	0.30	3.19	117.14	0.36	3.26
29.37	119.56	0.35	3.19	117.30	0.39	3.25
30.14	119.99	0.20	3.18	117.04	0.34	3.26
30.52	119.65	0.25	3.19	117.54	0.28	3.25
38.33	120.39	0.42	3.17	119.04	0.23	3.20

Tabela B.16: Testes de *throughput* da NoC com 32 ciclos de atraso na memória.