Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Mestrado em Sistemas e Computação

Projeto e Implementação de uma Plataforma MP-SoC usando SystemC

DISSERTAÇÃO DE MESTRADO

Rodrigo Soares de Lima Sá Rego

Natal/RN, Janeiro de 2006

Rodrigo Soares de Lima Sá Rego

Projeto e Implementação de uma Plataforma MP-SoC usando SystemC

Dissertação apresentada ao programa de Pós Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte, para obtenção do título de Mestre em Sistemas e Computação.

Professor Orientador: Dr. Ivan Saraiva Silva

Natal/RN, Janeiro de 2006

Banca Examinadora:

Prof. Dr. Ivan Saraiva Silva

Prof. Dr. Marius Strum

Prof. Dr. Eduardo Bráulio Wanderlei Netto

Prof. Dr. David Paul Déharbe

"*Se consegui enxergar além é porque estive no ombro de gigantes*"
Sir Isaac Newton

Agradecimentos

O homem é um animal social. Nenhuma realização é possível sem o apoio e a ajuda de muitas pessoas, que de forma direta ou indireta contribuem para sua concretização. É necessário, portanto, que saibamos demonstrar nossa gratidão a todos aqueles sem os quais as realizações seriam impossíveis.

Começo agradecendo à minha família, que sempre proveu apoio e reconhecimento, sempre me ajudando a me manter firme mesmo nos momentos mais difíceis. Com agradecimentos especiais para minha mãe, que mesmo apesar da distância se manteve sempre presente; para minha tia Miriam, que me acolheu de braços abertos em um momento tão delicado; e para a Débora, minha amada, que cumpriu muito bem o papel de musa inspiradora.

Em seguida, se faz necessário agradecer a todos os que participaram mais diretamente deste trabalho: primeiramente ao meu orientador e amigo desde 2000, Ivan, que sempre acreditou neste trabalho, mais até do que eu mesmo; ao Diego, que deu uma importante contribuição com o SPARC e resolveu diversos *bugs* de última hora; a Bruno e Daniel, que ajudaram no desenvolvimento do montador.

Agradeço também a todos os amigos que fiz ao longo destes 7 anos de UFRN. Alguns partiram após a graduação, outros trilharam o mesmo caminho que eu, a árdua e, às vezes ingrata, carreira acadêmica. O menino que entrou nesta Universidade em 1999 hoje se despede como um homem.

Por último, mas não menos importante, agradeço a Deus por ter me dado tantas oportunidades e colocado tantas pessoas maravilhosas em minha vida.

A todos, o meu "**Muito Obrigado!**",

Rodrigo Soares

Resumo

Este trabalho apresenta o conceito, desenvolvimento e implementação de uma plataforma MP-SoC, batizada **STORM** (MP-**S**oC Direc**T**ory-Based Platf**ORM**). A plataforma atualmente é composta pelos seguintes módulos: processador SPARC V8, processador GPOP, módulo de Cache, módulo de Memória, módulo de Diretório e dois diferentes modelos de *Network-on-Chip*, a NoCX4 e a Árvore Obesa. Todos os módulos foram implementados usando a "linguagem" SystemC, simulados e validados, tanto separadamente quanto em conjunto. A descrição dos módulos é apresentada em detalhes.

Para a programação da plataforma usando C foi implementado um montador SPARC, totalmente compatível com o código *assembly* gerado pelo compilador gcc. Para a programação concorrente foi implementada uma biblioteca de funções para gerenciamento de *mutexes*, com o devido suporte por parte do montador. São apresentadas 10 simulações do sistema, de complexidade crescente, para validação de todos os conceitos apresentados. As simulações incluem aplicações paralelas reais, como a multiplicação de matrizes, *Mergesort*, KMP, Estimação de Movimento e DCT 2D.

Palavras-chave: *System-on-Chip*, *Network-on-Chip*, Projeto Baseado em Plataforma, Coerência de Cache, Diretório, SPARC, Árvore Obesa, Processamento Paralelo.

Abstract

This work presents the concept, design and implementation of a MP-SoC platform, named **STORM** (MP-**S**oC Direc**T**ory-Based Platf**ORM**). Currently the platform is composed of the following modules: SPARC V8 processor, GPOP processor, Cache module, Memory module, Directory module and two different modles of Network-on-Chip, NoCX4 and Obese Tree. All modules were implemented using SystemC, simulated and validated, individually or in group. The modules' description is presented in details.

For programming the platform in C it was implemented a SPARC assembler, fully compatible with gcc's generated assembly code. For the parallel programming it was implemented a library for mutex managing, using the due assembler's support. A total of 10 simulations of increasing complexity are presented for the validation of the presented concepts. The simulations include real parallel applications, such as matrix multiplication, Mergesort, KMP, Motion Estimation and DCT 2D.

Keywords: System-on-Chip, Network-on-Chip, Platform-Based Design, Cache Coherence, Directory, SPARC, Obese Tree, Parallel Processing.

Índice

Lista de Figuras

Lista de Tabelas

1. Introdução

Os microprocessadores atuais possuem uma capacidade de processamento superior em muitas ordens de grandeza àqueles de uma década atrás. O desenvolvimento de transistores com nanômetros de largura e o aumento no número de estágios de pipeline para uma dezena ou mais foram fundamentais para que alcançássemos processadores a uma freqüência de alguns GigaHertz [TAN 99]. Apesar de sua velocidade de resposta ser bastante defasada em relação à dos processadores, o aumento no tamanho das memórias (Cache e RAM) dos sistemas contribui para o desempenho do mesmo, mantendo dados e instruções mais próximos do processador [TAN 99, HEN 03].

O avanço destas tecnologias tende a continuar, mas somente até certo ponto. A miniaturização dos transistores esbarra no tamanho mínimo da matéria, a molécula, seu limite teórico. Porém, é difícil imaginar um limite para a demanda por recursos computacionais. A demanda por capacidade de processamento das novas aplicações é a única que seguirá crescendo segundo a Lei de Moore, se não a uma taxa maior. O desafio para os projetistas é prover sistemas que atendam à demanda apesar dos impedimentos técnicos e teóricos.

Apesar de já existir desde os primórdios da computação, o processamento paralelo sempre ficou restrito a supercomputadores, pois sistemas com apenas 1 processador eram suficientes para um computador pessoal. Talvez pelo custo multiplicado pelo número de processadores do sistema, talvez pelas dificuldades que são somadas às de um sistema monoprocessado, talvez pela falta de necessidade de haver mais de um processador no sistema, o fato é que sistemas pessoais com mais de um processador são raros ainda nos dias de hoje, mas o mercado já começa a dar sinais de que esta não será uma situação duradoura.

Desde a década de 1990, a capacidade de integração de transistores em um único chip tornou-se tamanha que surgiu o conceito de sistemas computacionais inteiros em um chip, o *SoC* (System-on-Chip). Este conceito foi recebido com entusiasmo pelos pesquisadores da área, já que com esta integração a distância física entre os componentes deixa de ser um problema, pois todos os componentes agora estão no mesmo chip. O SoC permite também a procura por sistemas de interconexão entre módulos mais complexos e eficientes, como a *NoC* (Network-on-Chip). A NoC trata de efetivamente trazer uma rede de computadores inteira para dentro de um único microchip, com todos os seus conceitos, paradigmas e suas vantagens.

Por ser um conceito ainda novo, poucos estudos acadêmicos vêm sendo feitos no sentido de conceber um sistema de processamento paralelo integrado em um único chip, o que é, senão o mais provável, pelo menos um possível futuro para os sistemas computacionais. Este trabalho apresenta **STORM** (*MP-**S**oC Direc**T**ory-Based Platf**ORM***), uma plataforma computacional que

visa utilizar os conceitos recentes de SoC e NoC, assim como o conceito tradicional de Processamento Paralelo para construir um sistema funcionalmente completo.

Este texto está organizado da seguinte forma: A **Seção 2** apresenta uma revisão dos conceitos teóricos englobados no projeto da plataforma, situando o trabalho no atual estado da arte. Na **Seção 3** o trabalho desenvolvido é apresentado, são apresentados os componentes da plataforma desenvolvida, suas funções e decisões de projeto. A **Seção 4** mostra a metodologia adotada no trabalho e suas justificativas. A **Seção 5** apresenta os resultados obtidos nas simulações realizadas. O texto termina com as conclusões, na **Seção 6**.

2. Revisão Bibliográfica

Esta seção apresenta os principais conceitos que envolvem o projeto de uma plataforma MP-SoC. Por este ser um tema tão amplo e complexo, esta seção aborda diversos aspectos do projeto de sistemas digitais, dos mais tradicionais, como a teoria de processamento paralelo, até os mais recentes e temas de conferências da atualidade, como Sistemas-em-Chip. Estes conceitos são aqui apresentados de uma forma sucinta, sendo apenas uma revisão dos mesmos sob o ponto de vista do autor. O leitor que quiser se aprofundar em sua leitura é recomendado a seguir as referências feitas ao longo da seção.

2.1. Processamento Paralelo

Embora a freqüência de um microprocessador seja um aspecto importante no cálculo de seu desempenho, a execução puramente seqüencial de aplicações limita severamente o resultado final. Por esta razão, o paralelismo vem sendo explorado em diversos níveis nas máquinas mais recentes.

Em seu menor nível de granularidade, o paralelismo se apresenta em instruções de máquina. Em qualquer aplicação é possível identificar algumas instruções, ou trechos de código, que são independentes entre si, podendo, portanto, ser executados concorrentemente sem que isto prejudique a semântica da aplicação. Este nível de paralelismo já é bastante explorado, tendo em vista sua absoluta necessidade para que possa haver execução pipeline.

O paralelismo de instrução, porém, necessita de mecanismos de análise de dependência, assim como mecanismos para superar alguns tipos de dependência. Isso sem contar o seu limite de ganho de desempenho. Para obter ganhos maiores, faz-se também necessária a exploração de paralelismo de mais alto nível, como os de thread e de aplicação. O paralelismo de thread ocorre quando, em uma aplicação, existem diferentes fluxos de execução, ou threads, que podem ser executadas completamente em paralelo, podendo haver um ou mais pontos de sincronismo entre os fluxos. As linguagens de programação mais recentes, como Java, dão suporte para que o programador utilize threads em suas aplicações. Porém, a maioria das máquinas, por falta de recursos de exploração deste nível de paralelismo, executa as threads sequencialmente, desperdiçando assim uma oportunidade de ganho no desempenho. O paralelismo de aplicação é ainda mais simples: duas ou mais aplicações totalmente independentes são executadas em paralelo, não havendo entre elas nenhum ponto comum. As máquinas multitarefa têm muito a ganhar com este nível de paralelismo.

Por haver mais de uma forma de exploração do paralelismo, diversos modelos computacionais foram desenvolvidos, sendo cada um mais adequado para um caso específico. Neste universo de diferentes modelos, torna-se por vezes muito difícil o seu agrupamento. A classificação, que apesar de não ser perfeita e definitiva, é a mais utilizada e aceita é a de Flynn [FLY 72]. Ela se baseia em apenas duas características: as seqüências de instruções e de dados. Uma seqüência de instruções diz respeito ao registrador contador de programa, PC. Uma máquina que possua apenas um PC será classificada como *Single Instruction*, uma que possua dois ou mais registradores PC como *Multiple Instruction*. A seqüência de dados diz respeito às operações. Uma máquina que execute apenas uma operação por vez será classificada como *Single Data*, outra que execute duas ou mais como *Multiple Data*. A **Tabela 2.1** possui um resumo destes conceitos, e a **Figura 2.1** ilustra os diferentes modelos de arquiteturas paralelas.

Tabela 2.1. Taxonomia de Flynn

Nome	Instruções	Dados	Exemplo
SISD	1	1	Máquina de Von Neumann
SIMD	1	Mais de 1	Computador vetorial
MISD	Mais de 1	1	Computador sistólico
MIMD	Mais de 1	Mais de 1	Multiprocessador, multicomputador

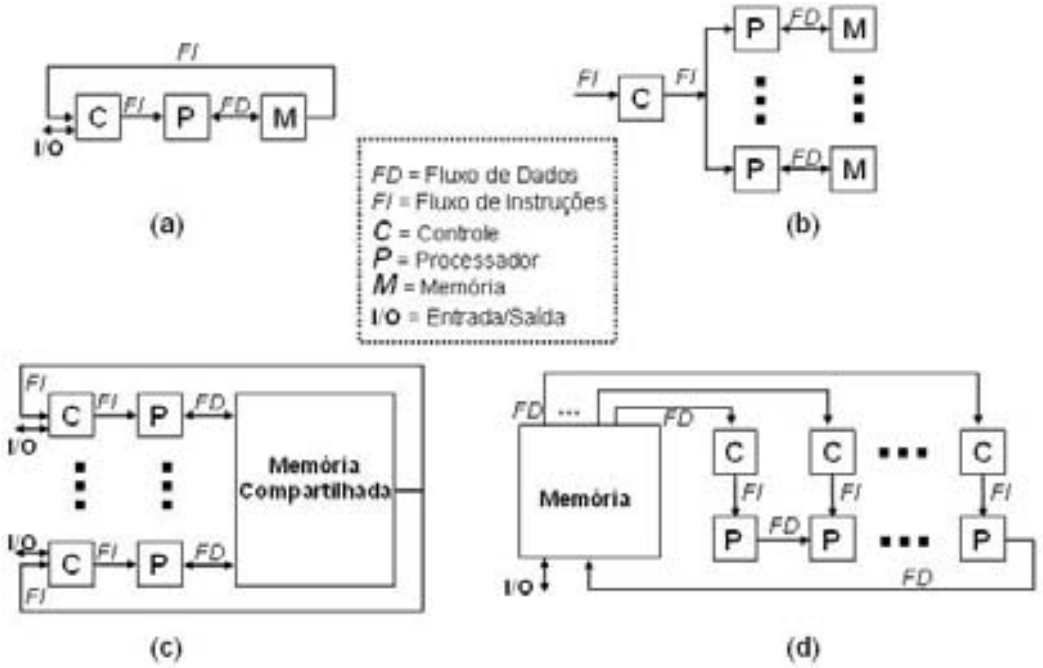


Figura 2.1. Arquiteturas paralelas segundo Flynn (a) SISD; (b) SIMD; (c) MIMD com Memória Compartilhada; (d) MISD

As máquinas *Single Instruction Single Data*, ou SISD, são aquelas que seguem a arquitetura de Von Neumann: uma memória, uma Unidade Lógica e Aritmética (ULA), uma Unidade Central de Processamento (CPU) e uma unidade de controle. Todos os sistemas monoprocessados podem ser incluídos nesta classificação, apesar de haver divergências sobre as arquiteturas que utilizam *pipeline*.

Os computadores verdadeiramente paralelos, que possuem múltiplos fluxos de instruções operando sobre múltiplos fluxos de dados, são classificados como MIMD (*Multiple Instruction Multiple Data*). Estes computadores vão desde simples sistemas com dois processadores ligados via um barramento até supercomputadores com centenas de processadores interligados por complexas redes de interconexão. Por ser um paradigma tão amplamente estudado e vasto, as máquinas MIMD são mais bem detalhadas na seção seguinte.

Nos primórdios da computação, todas as máquinas paralelas seguiam o paradigma MIMD [HWA 93]. Com o tempo, foram surgindo instruções específicas para o processamento de vetores, devido a uma demanda especial por esta classe de aplicações, utilizada em diversos campos da ciência. A especialização dos sistemas para processamento de vetores acabou por gerar uma nova classe de máquinas paralelas, a SIMD (*Single Instruction Multiple Data*). Um processador vetorial é equipado com múltiplos *pipelines* de vetores que podem ser usados concorrentemente. Existem duas famílias de processadores vetoriais: *memory-to-memory*, que suporta que o fluxo de operandos venha direto da memória para os *pipelines*, e daí de volta para a memória; e *register-to-register*, que usa registradores de vetor para fazer a interface entre a memória e os *pipelines*. O processamento vetorial e as máquinas MISD constituem um universo por si só, são mais discutidos em [TAN 99, HWA 93].

Por fim, existem também as máquinas MISD (*Multiple Instruction Single Data*). Nelas, um mesmo fluxo de dados passa por um vetor de processadores com diferentes fluxos de operações. Estas máquinas são conhecidas como máquinas sistólicas, e são mais utilizadas na execução de algoritmos específicos.

2.1.1. MIMD

Como se pode perceber, as máquinas paralelas SIMD e MISD se aplicam apenas a funções específicas, não tendo uso em outras classes de aplicações que fujam àquelas as quais melhor se adequam. Por isso, as máquinas MIMD prevalecem como as mais adequadas para o processamento paralelo em geral. Existem duas superclasses quando se trata de processamento paralelo MIMD, ou simplesmente processamento paralelo: os modelos com *memória compartilhada* e com memória distribuída, ilustrados na **Figura 2.2**. Estas superclasses definem

não só a arquitetura e escalabilidade do sistema, mas também a maneira como este é programado. Ambas são discutidas a seguir.

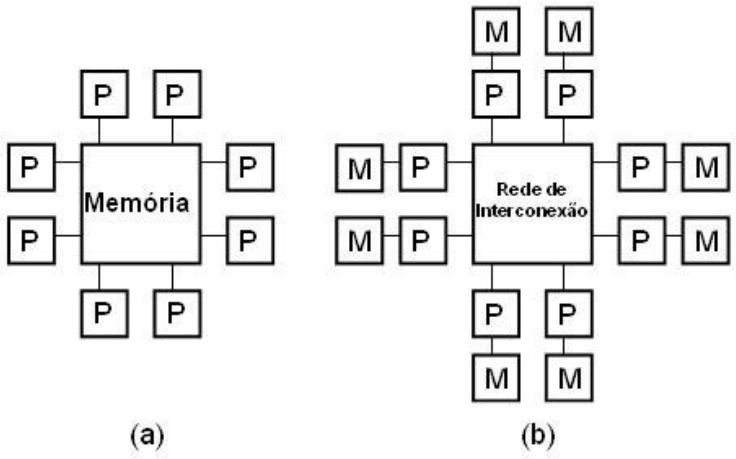


Figura 2.2. Os dois modelos MIMD (a) Máquina MIMD com memória compartilhada; (b) Máquina MIMD com memória distribuída

2.1.1.1. Memória Distribuída

Os sistemas com memória distribuída não possuem um único espaço de endereçamento. Neles, cada processador tem sua própria memória exclusiva e acessa apenas a ela, usando instruções LOAD e STORE. Outros processadores não têm acesso à sua memória. A ausência de uma memória compartilhada pro todos os processadores torna a comunicação entre processos nestes sistemas mais complexa, devendo ser feita com o uso de troca de mensagens, que circulam por uma rede de interconexão que liga todos os processadores. Estes sistemas são feitos com a integração de diversos computadores interligados por uma rede, daí seu nome de *multicomputador*.

Este modelo, baseado em troca de mensagens, torna a programação bem mais difícil. É responsabilidade do programador explicitar a troca de mensagens entre os processos, utilizando normalmente primitivas de SEND e RECEIVE. Para isto, é necessário que o programador saiba a distribuição dos dados entre as diferentes memórias. Uma distribuição malfeita pode levar o sistema a uma perda considerável de desempenho, o que não acontece nos sistemas com memória compartilhada. Porém, apesar deste problema, os sistemas com memória distribuída são amplamente utilizados, por causa de sua maior escalabilidade. Uma rede que suporte grande tráfego pode suportar milhares de processadores, já existindo sistemas comerciais desse porte [TAN 99].

2.1.1.2. Memória Compartilhada

As primeiras máquinas paralelas utilizavam o modelo de memória compartilhada. Este modelo é também conhecido por *multiprocessador*, por ser mais utilizado em sistemas dessa natureza. Nos sistemas de memória compartilhada, todos os processadores possuem acesso a todo o espaço de endereçamento da memória, podendo, assim, acessar qualquer dado com instruções comuns de LOAD e STORE. Os processos, onde quer que estejam sendo executados, se comunicam através de variáveis compartilhadas. Este modelo é reconhecidamente de mais fácil programação, pois ela é feita da maneira intuitiva como o programador entende o programa em linguagem de alto nível, não sendo necessário por parte deste qualquer conhecimento do hardware ou distribuição manual de processos.

Os sistemas multiprocessador se dividem de acordo com a maneira de implementação de sua memória compartilhada. As três classes são UMA (*Uniform Memory Access*), NUMA (*Non-Uniform Memory Access*) e COMA (*Cache-Only Memory Access*). Todos os sistemas de uma determinada classe possuem características específicas, tanto na implementação quanto na programação. As três classes de sistemas multiprocessador serão mais bem detalhadas adiante.

A maior desvantagem do modelo de memória compartilhada é sua reduzida escalabilidade. À medida que vão sendo incluídos mais processadores em um sistema, torna-se mais complexo o gerenciamento da memória compartilhada. Isto torna os grandes sistemas com memória compartilhada muito caros, e inviáveis de serem desenvolvidos. Este problema levou os projetistas a criarem um novo modelo de processamento paralelo: o *multicomputador*.

2.1.1.2. UMA

Nos sistemas multiprocessador que seguem o modelo *Uniform Memory Access* o acesso a qualquer módulo de memória é feito de forma uniforme entre todos os processadores; ou seja, o tempo de acesso à memória é constante. Isto pode ser devido à linearidade do sistema de interconexão, como um simples barramento, ou devido a uma nivelação explícita nos tempos de acesso dos diferentes módulos de memória. A uniformidade nos acessos à memória é um importante fator de simplificação da geração de código para o sistema, pois não há preocupação com a localização física dos dados e código. Esta uniformidade permite que o desempenho de determinada aplicação seja medido ou estimado com boa precisão.

Dentro do modelo UMA, destaca-se o SMP (*Symmetric MultiProcessor*). Um sistema é considerado simétrico quando é composto por processadores de capacidade de processamento equivalente, compartilham o mesmo conjunto de instruções e possuem acesso a todos os recursos do sistema, inclusive a funções de E/S. Nestes sistemas, os processadores são

considerados *simétricos*, ou seja, o SO não faz distinção entre eles, podendo alocar processos em qualquer um sem que isto afete o sistema em nenhum aspecto. Pode-se dizer que o SMP é a forma mais simples de processamento paralelo, não havendo nenhum tipo de preocupação com alocação de dados nos diferentes módulos de memória, ou tarefas entre diferentes processadores. Por esta simplicidade, os computadores pessoais multiprocessador utilizam-se deste modelo, como o Intel SMP [INT 97]. Porém, tal simplicidade tem um preço, e o maior problema dos sistemas SMP é a sua escalabilidade restrita.

A maioria dos sistemas SMP utiliza-se de um único barramento para interconexão, já que o uso de um sistema de barramentos iria acabar com a uniformidade de acesso à memória. Isto limita o número de processadores do sistema à capacidade do barramento. Outro problema ocorre com aqueles que utilizam redes de interconexão de tempo constante, como um *crossbar*. Nestes, a escalabilidade do sistema é limitada pela complexidade do *crossbar*, que é de ordem quadrática.

2.1.1.3. NUMA

Outro modelo de sistemas multiprocessador é o NUMA, no qual não há a obrigação de uniformidade do acesso à memória. Nos modelos NUMA existe o conceito de memória local e memória remota, ou pelo menos de memória próxima e memória distante, e o tempo de acesso a estes diferentes grupos pode variar enormemente. A não-exigência de tempo uniforme no acesso à memória elimina a grande limitação de escalabilidade dos sistemas UMA.

As máquinas NUMA compartilham algumas características por causa de sua não-uniformidade. Nelas existe um único espaço de endereçamento, visível a todos os processadores do sistema. O acesso às memórias remotas é mais lento do que o acesso às memórias locais, e este é sempre feito por intermédio de instruções LOAD e STORE. As máquinas NUMA normalmente possuem desempenho inferior às equivalentes UMA de mesmo *clock*, por isso a importância que SO e compilador adquirem na otimização de distribuição de dados e tarefas, e paralelização das aplicações.

Os sistemas NUMA mais simples possuem tempo de acesso à memória conhecido por causa da ausência de memórias *cache*. Estes sistemas são chamados NC-NUMA. Sua simplicidade de implementação e ausência de mecanismos de coerência de cache custam a estes sistemas o ganho de desempenho advindo do uso de memórias cache. Os sistemas NUMA mais complexos fazem uso de cache, e são conhecidos por CC-NUMA. Nestes, cada processador mantém uma memória cache, e a coerência entre elas é feita através de um mecanismo de diretório, pois seria impossível a implementação de protocolos *snoopy* [TAN 99].

2.1.1.4. COMA

Pelo que pôde ser visto, as máquinas UMA possuem bom desempenho, mas escalabilidade limitada, enquanto que as máquinas NUMA possuem grande escalabilidade, mas desempenho inferior por causa da distribuição dos dados. As máquinas CC-NUMA ainda possuem problemas em relação à implementação de mecanismo de consistência de cache e ao compartilhamento das páginas da cache. Uma alternativa radical para estes dois modelos é a eliminação da memória principal, transformando-a efetivamente em enormes caches. Esta alternativa é conhecida como COMA (*Cache-Only Memory Access*). Nos sistemas COMA uma página não possui "residência" fixa, elas migram pelo sistema de acordo com a demanda, o que leva a grandes problemas para rastreá-las. Também existe o problema de se eliminar a última cópia de uma página durante uma substituição. Ambos os problemas possuem soluções, que exigem mecanismos específicos para tratar esta nova classe de problemas. A total descentralização no acesso aos dados permite às máquinas COMA um grande potencial em ganho de desempenho, mas apesar disto poucas máquinas COMA foram construídas até o momento.

2.2. SoCs

A *International Technology Roadmap for Semiconductors* (ITRS) [ITR 04] prevê para o fim da década uma capacidade de integração de bilhões de transistores em um único microchip. Esta previsão, se concretizada, causará um enorme impacto no atual paradigma de projeto de sistemas. Para efeitos de comparação, o Pentium IV [INT 05] usa aproximadamente 55 milhões de transistores em sua implementação. Porém, com a atual tecnologia seria impossível desenvolver um processador que explore todo este potencial. O projeto de um único processador ocupando bilhões de transistores levaria uma eternidade, isto para não mencionar o tamanho da equipe necessária, impossibilidade de testar o chip, e os custos associados.

No fim da década de 1990 surgiu o conceito de *System-on-Chip* (SoC) [WAW 99, MAN 99], que entrevia a possibilidade de integrar um sistema computacional completo dentro de um chip. Esta nova linha permitiria que os atuais supercomputadores paralelos se tornassem um único chip, podendo ser utilizado em computadores pessoais e até mesmo sistemas embarcados. Com o avanço da tecnologia, os SoCs foram se tornando cada vez mais viáveis, e hoje em dia podem ser encontrados alguns modelos no mercado [INT 02, ART 03, HEL 02]. Os modelos atuais possuem capacidade para integração de no máximo algumas dezenas de módulos, incluindo-se aí processadores, módulos DSP, controladores de memória e E/S, ASICs, etc. A

tendência atual dos SoCs é o uso em sistemas embarcados, com um conjunto de aplicações específico.

Alguns pesquisadores, porém, vêem nos SoCs um enorme potencial para a exploração do processamento paralelo em larga escala. Os computadores teriam, finalmente, verdadeira concorrência na execução de tarefas, ao invés da simples emulação de paralelismo através da rápida troca de contexto dos atuais processadores. A execução concorrente de tarefas representaria um enorme salto no desempenho dos sistemas. Por outro lado, a sua implementação apresenta muitos desafios, a começar pelo sistema de comunicação.

Apesar de suas conhecidas limitações [TAN 99], os barramentos e sistemas de barramentos ainda predominam como subsistema de interconexão. A tecnologia de barramentos já é antiga, e possui inúmeros estudos e melhorias agregadas. O amplo uso em sistemas computacionais o torna sem dúvida uma escolha completamente confiável. Barramentos como o ARM AMBA [ARM 05] possuem um reconhecido desempenho mesmo em sistemas mais complexos. Porém, existem limitações de natureza técnica para a escalabilidade dos barramentos. Em teoria, um barramento com freqüência de relógio infinita poderia conectar um sem-número de módulos. Mas é natural que exista um limite para a freqüência operacional de qualquer sistema, e à medida que se conectam mais módulos a um barramento do que sua freqüência permite, o desempenho do sistema como um todo se degrada. Por outro lado, a cada módulo extra conectado a um barramento, a capacitância do mesmo é aumentada, o que não só exige mais energia como também contribui para a degradação do sistema. O uso de um sistema de barramentos contorna estas limitações até certo ponto, já que se cria aí o conceito de dados locais e remotos. Quanto mais dados remotos forem necessários, menor será o desempenho do sistema. Por estar razões, estudos mostram que a capacidade máxima de módulos conectados a um barramento é de no máximo algumas dúzias [ZEF 02].

Uma alternativa para o uso de barramentos veio das redes de computadores. A partir do momento que se tem um sistema computacional inteiro em um único chip, por que não interconectá-los com uma rede em um único chip? A partir desta premissa surgiu o conceito de *Network-on-Chip* (NoC) [BEN 02, JAN 03], que será visto com mais detalhes na **Seção 2.3**. Pode-se dizer que as NoCs têm como principal vantagem sobre os barramentos uma escalabilidade muito maior, permitindo a interconexão de centenas, ou até milhares de módulos. Apesar desta grande diferença de escalabilidade, o mercado ainda não abraçou a idéia das NoCs. Uma exceção é a *Aethereal*, NoC desenvolvida pela Philips [DIE 05].

Os MP-SoCs (*Multiprocessor SoC*) podem ser vistos, à princípio, como um sistema de processamento paralelo integrado em um chip. Existem controvérsias sobre esta definição, já que qualquer SoC possui mais de uma unidade de processamento, o que equivaleria a dizer que todos os SoCs são MP-SoCs. Porém, este autor crê que esta definição seja mais correta, tendo em vista a necessidade em se diferenciar os SoCs para propósito específico, normalmente

destinados a sistemas embarcados, dos SoCs de propósito geral, que são verdadeiros sistemas de processamento paralelo. Apesar desta diferenciação, toda a teoria aplicada a SoCs é também aplicável a MP-SoCs, incluindo-se aí o problema de interconexão. Os MP-SoCs, porém, possuem algumas questões adicionais, de estrita relação com o processamento paralelo, como sincronismo de processos, tratamento de interrupções, coerência de cache, suporte a sistema operacional, etc. Algumas destas questões se tornam mais complexas com o uso de NoCs como subsistema de interconexão de módulos, devido ao seu recente surgimento e falta de aplicação real, assim como a impossibilidade de broadcasting.

2.3. Projeto Baseado em Plataforma

A metodologia tradicional de projeto *full-custom*, onde a equipe projetista desenha o chip transistor a transistor, está caindo cada vez mais em desuso devido à crescente quantidade de transistores por chip, apesar de muitas vezes esta ser a forma de projeto que produz circuitos mais eficientes. Por causa do aumento na complexidade dos atuais sistemas, o tempo e custo para seu desenvolvimento tornam-se cada vez mais elevados. Equipes cada vez maiores precisam ser empregadas, e a cada novo sistema é necessário todo um trabalho de adaptação de interfaces dos módulos, alocação (*placement*) e roteamento dos mesmos dentro do chip, e teste do novo sistema. A partir da percepção de que este modelo de projeto se torna inviável para os novos sistemas, criou-se o conceito de Projeto Baseado em Plataforma.

Desde a década de 1980 existe uma preocupação com o tempo de projeto, quando surgiu o conceito de *Standard Cells* [BRO 89], uma abstração e padronização sobre os transistores para aceleração no projeto do circuito. Naquela época os transistores possuíam canais de 3 mícron [CRA 88], quase 30 vezes maiores que os atuais, e um chip complexo possuía alguns poucos milhares de transistores. Nos tempos atuais, esta preocupação ganhou uma dimensão muito maior: o custo de instalação de uma *foundry* que possua tecnologia de 0,10 mícron está estimado em 3,5 bilhões de dólares [DEA 01], e um conjunto de máscaras completo para esta tecnologia custa em torno de 1 milhão de dólares [SBN 99]. Excluindo-se aí o custo operacional, apenas para recuperar o investimento inicial do conjunto de máscaras vendendo chips a 10 dólares, é necessário que sejam vendidos 100.000 chips.

Outro grande problema enfrentado pelas empresas é o *time-to-market*, caracterizado pelo tempo decorrido desde o início do projeto de um chip até sua disponibilidade no mercado. Um *time-to-market* elevado representa por um lado maiores custos para o desenvolvimento do sistema, pois significa o pagamento de uma equipe por mais tempo, e por outro lado a demora no lançamento de um produto significa uma menor curva de faturamento. A curva de faturamento é uma função representada na **Figura 2.3** [MAD 04]. Na figura, pode-se ver que o

lucro gerado por um produto tem estrita relação com a velocidade de seu lançamento no mercado. À medida que o lançamento é adiado em relação ao surgimento inicial daquele tipo de produto no mercado, o lucro obtido com a venda do produto decai gradualmente. Com o acirramento da concorrência na área de micro-componentes, o time-to-market tem sido cada vez mais estreito.

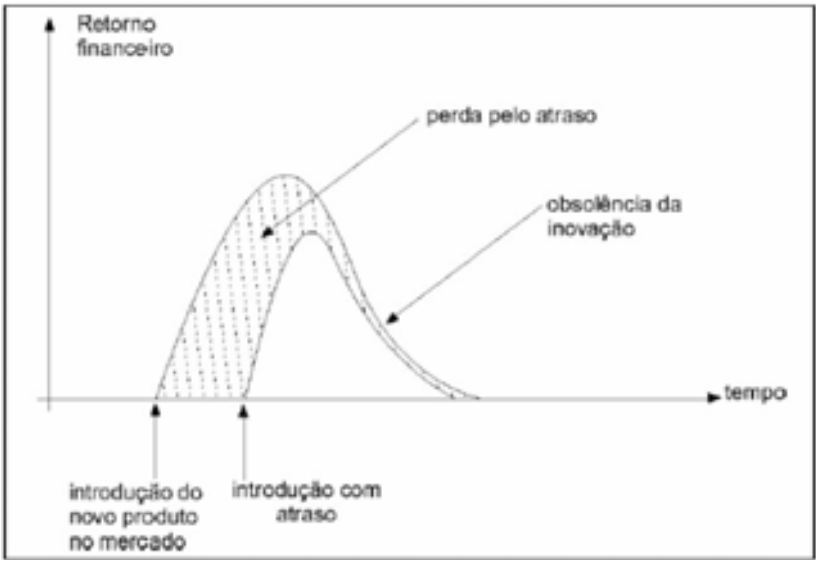


Figura 2.3. Retorno financeiro de projeto em função do tempo

Por estas razões, Projeto Baseado em Plataforma e Reuso têm sido assuntos cada vez mais discutidos, tanto pela comunidade acadêmica quanto pelas empresas fabricantes de chips. O conceito de plataforma não tem uma definição unânime, mas pode ser entendido como "uma abstração que cobre diversos possíveis refinamentos em baixo-nível" [SAN 01]. O conceito de plataforma está intimamente ligado ao reuso de componentes (e software). O objetivo do projeto de uma plataforma, ao invés de um simples circuito, é prover um ambiente de projeto mais rápido, utilizando-se para isto de um conjunto de componentes pré-definidos, que podem ser desde microprocessadores a ASICs, passando por módulos DSP e lógica reconfigurável. A plataforma também deve prover recursos de interconexão entre os componentes, usualmente barramentos, mas também podendo ser canais dedicados (o que requer mudanças no projeto a cada nova instância) ou NoCs. Assim sendo, o projeto de um sistema complexo utilizando-se uma plataforma consiste apenas em escolher os módulos necessários e interconectá-los, criando uma instância da plataforma. Esta metodologia pode diminuir o *time-to-market* de um produto de meses para duas semanas.

Uma grande vantagem das plataformas é o reuso não só de hardware, como de software. Um dos maiores atrativos na hora de se escolher um chip para um sistema é o conjunto de aplicações e ferramentas (compilador, sistema operacional, etc.) disponíveis para a arquitetura.

Um sistema no qual todas as aplicações devem ser reescritas a cada nova versão está fadado ao fracasso. Um exemplo real e amplamente conhecido é o da arquitetura x86, que pode ser considerada uma plataforma do ponto de vista do software. Mesmo o mais moderno membro da família, o Pentium IV HT, é capaz de executar aplicações escritas para o 486.

2.4. NoCs

O surgimentos dos SoCs criou a necessidade de um novo paradigma de interconexão, tendo em vista a impossibilidade de se usar barramentos em sistemas mais robustos. A solução adotada não foi uma idéia original, mas sim uma adaptação de algo que já existia: as redes de interconexão. Estas redes, por sua vez, já são uma adaptação das redes de computadores para atender aos propósitos específicos dos sistemas de processamento paralelo. As redes de interconexão são utilizadas para interconectar os módulos em um sistema paralelo. Elas existem tanto nos sistemas multiprocessador como multicomputador, possuindo características muito semelhantes nos dois sistemas. Todas elas são compostas por *roteadores*, elementos que passam os dados para outros roteadores, até que se chegue no nodo destino. As *Networks-on-Chip* (NoCs) nada mais são do que adaptações das redes de interconexão para integração em um único chip. Muitos estudos sobre as NoCs vêm sendo feitos nessa área nos últimos anos [BEN 02, KUM 02, JAN 03, BER 05], e as NoCs ganham cada vez mais importância no meio acadêmico.

O uso de NoC como subsistema de interconexão é bastante adequado no contexto atual. Sua grande escalabilidade permite o rápido dimensionamento do sistema, e em conjunto com uma plataforma possibilita o desenvolvimento de sistemas de portes bem variados em pouco tempo. Existem em vários modelos de NoCs, não havendo ainda um consenso sobre o melhor deles. Para permitir a rápida integração de diferentes modelos de NoCs em um sistema, a VSIA tem pesquisado protocolos de interface entre roteador e módulo para serem usados como futuro padrão. O VCI (*Virtual Component Interface*) é o resultado desta pesquisa [VSI 00]. Um outro protocolo, mais utilizado por ser mais completo, e que está se transformando em um padrão de fato é o OCP (*Open Core Protocol*) [OCP 01]. Apesar de um padrão ser necessário para maior agilidade na integração de diferentes módulos, somente o futuro dirá qual será este padrão.

Os diferentes modelos de NoCs, por mais distintos que sejam, possuem algumas características básicas comuns. Estas características advêm das redes de interconexão. São elas topologia, roteamento, chaveamento, controle de fluxo, arbitragem e memorização. Cada uma delas é detalhada a seguir.

2.4.1. Topologia

Esta talvez seja a mais marcante característica de uma NoC. A topologia de uma NoC não diz respeito diretamente aos roteadores, mas sim à maneira como eles estão interligados. As topologias existentes são agrupadas em dois grandes conjuntos: as redes diretas e as redes indiretas.

As *redes diretas* são aquelas nas quais cada roteador está ligado a um módulo do sistema por uma conexão ponto-a-ponto. Este conjunto de módulo-roteador pode ser visto como um nodo, e é passível de ser origem e destino de pacotes. Dentre as redes diretas pode-se destacar algumas topologias em especial, mostradas na **Figura 2.4**. Cada círculo representa um par módulo-roteador. Apesar de a topologia ideal ser a totalmente conectada, pois a distância entre quaisquer dois nodos é mínima, sua implementação torna-se inviável para sistemas de maior porte devido ao seu elevado grau de interconexões. Por causa deste problema foram propostas outras topologias mais escaláveis, como a Grelha 2D, bastante utilizada, e sua evolução, o Torus 2D, que interconecta as pontas da grelha. Existem alguns modelos mais complexos, como o Cubo 3D e 4D, também utilizados comercialmente em sistemas de processamento paralelo.

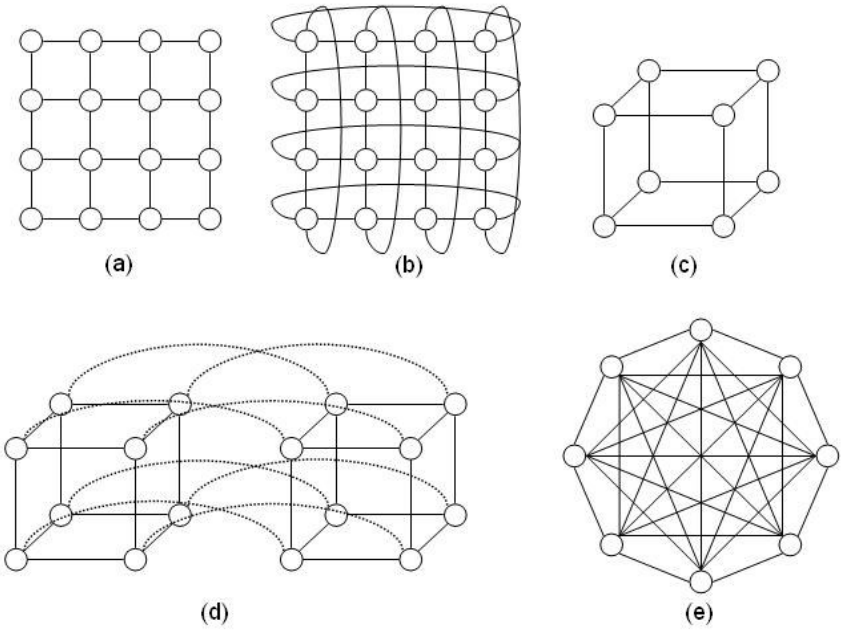


Figura 2.4. Topologias diretas: (a) Grelha 2D; (b) Torus 2D; (c) Cubo 3D; (d) Cubo 4D ou Hipercubo; (e) Totalmente Conectada

As *redes indiretas* se caracterizam pela não obrigatoriedade do par módulo-roteador. Nestas redes, alguns roteadores fazem este par, enquanto outros se conectam apenas a outros roteadores. As redes indiretas mais amplamente difundidas são o *crossbar* e as *multiestágio*,

mostradas na **Figura 2.5**. As redes do tipo *crossbar* são bastante antigas, sendo utilizadas há décadas por companhias telefônicas, mas mesmo sistemas mais modernos ainda a utilizam, como o Sun Enterprise 1000 [CHA 98]. Sua principal vantagem é permitir que todos os módulos se comuniquem ao mesmo tempo, não havendo obstrução dos canais. Porém, como o seu número de roteadores é da ordem do quadrado do número de módulos, sua escalabilidade é bastante limitada. As redes multiestágio, como a rede Ômega, são mais escaláveis, mas não é possível haver comunicação entre todos os módulos ao mesmo tempo.

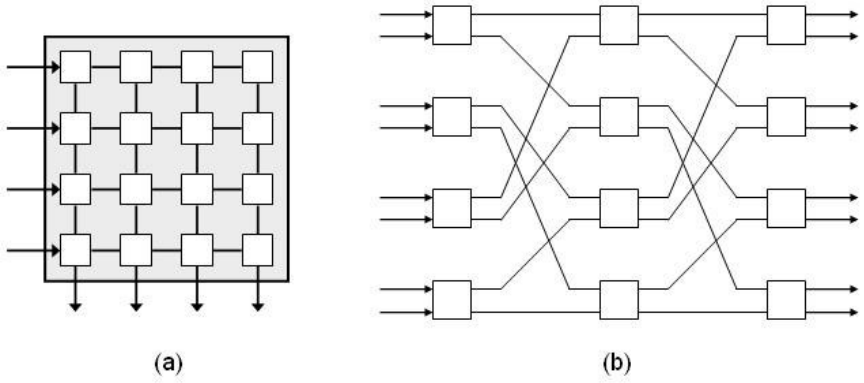


Figura 2.5. Topologias indiretas: (a) Crossbar; (b) rede multiestágio Ômega

2.4.2. Roteamento

Muitas das topologias vistas na seção anterior possuem mais de um caminho entre dois nós. O processo de escolha do caminho a ser percorrido pelos pacotes da rede caracteriza o seu roteamento. O algoritmo de roteamento deve evitar (ou tratar) a ocorrência de *deadlock* e *livelock*. Estes problemas podem comprometer o funcionamento do sistema até o ponto de sua total paralisação. O deadlock ocorre quando o algoritmo de roteamento cria uma dependência circular entre pacotes, ou seja, quando um pacote não pode ser transmitido antes de algum outro, e este segundo não pode ser transmitido antes do primeiro. Esta dependência pode ser mais complexa e difícil de se perceber, à medida que envolve mais pacotes. O *livelock* ocorre tipicamente em sistemas tolerantes a falhas, quando um pacote jamais chega a seu destino, circulando eternamente pela rede.

Os algoritmos de roteamento possuem duas classificações. A primeira delas determina aonde está implementado o algoritmo, se na fonte ou no roteador. Quando implementado na fonte (*roteamento fonte*), o módulo gerador do pacote determina o caminho a ser tomado, incluindo esta informação de alguma forma no pacote, o que claramente aumenta o seu

overhead. Quando implementado no roteador (*roteamento distribuído*), cada roteador determina se o pacote deve ser enviado para seu módulo local ou para um roteador vizinho, e qual deles.

Também se classifica os algoritmos de roteamento de acordo com sua adaptatividade às condições da rede no momento da transmissão. Aqueles que não se adaptam às condições correntes são chamados *determinísticos*. Nestes algoritmos de roteamento, uma vez determinada fonte e destino, o caminho entre elas é fixo e conhecido de antemão. Os algoritmos cujo funcionamento depende de condições dinâmicas da rede, e não é possível determinar o caminho que será percorrido por um pacote sem se conhecer as condições correntes, são chamados *adaptativos*.

Os algoritmos adaptativos podem ser utilizados para distribuir o tráfego de maneira mais homogênea pela rede, ou em sistemas tolerantes a falhas. Porém, estes algoritmos são mais complexos e exigem mais recursos de hardware. Um algoritmo simples e muito difundido em topologias dimensionais por ser livre de *deadlock* é o *roteamento dimensional* (também conhecido por *roteamento XY*). Neste roteamento, os pacotes percorrem uma dimensão da rede até a posição adequada, e depois a outra. Em uma topologia do tipo grelha, isto equivale a se deslocar pelo eixo x até a coordenada do destino, e então pelo eixo y até o destino.

2.4.3. Chaveamento

Enquanto o roteamento determina por onde os dados serão transmitidos, o chaveamento determina *como* isto acontece. O chaveamento é o mecanismo utilizado na transmissão de dados entre os roteadores de uma rede. Existem quatro técnicas mais comumente adotadas e difundidas. A saber: *chaveamento de circuito*, *store-and-forward*, *virtual cut-through* e *wormhole*.

O *chaveamento de circuito* é uma técnica baseada nas redes de telefonia. Ela consiste na alocação exclusiva de um caminho entre os módulos fonte e destino. Seu funcionamento se inicia com a injeção de um cabeçalho pela fonte. Este cabeçalho percorre a rede, de acordo com o roteamento utilizado, alocando os canais por onde passa. Uma vez que chegue ao destino, e já tenha sido formado um caminho na rede entre fonte e destino, começa a transmissão dos dados, sem nenhum tipo de contenção, pois o caminho está dedicado exclusivamente. A transmissão é finalizada com um terminador, que desfaz o caminho. Esta técnica é apenas adequada para sistemas onde haja pouca comunicação entre seus módulos, sendo estas longas e pouco freqüentes.

Para sistemas onde há comunicações mais freqüentes e curtas, o chaveamento *store-and-forward* (SAF) é mais adequado. Ele consiste na divisão da mensagem em pacotes de tamanho fixo, cada um contendo um cabeçalho com informações necessárias para seu roteamento. Os

pacotes vão sendo armazenados em cada roteador que passam, o qual decide por uma porta de saída e o transmite, até que este chegue ao destino. Isto exige *buffers* maiores do que no chaveamento de circuito, pois o pacote inteiro deve ser armazenado no roteador até que seja transmitido. Além disso, a inclusão de um cabeçalho a cada pacote aumenta o *overhead* de comunicação.

A técnica do *virtual-cut-through* (VCT) foi proposta em [KER 79] como uma otimização do chaveamento SAF. Ela funciona da mesma forma que o chaveamento *store-and-forward*, porém quando um roteador recebe um pacote e seu canal de saída está desocupado, o pacote é transmitido diretamente para a saída, desviando do buffer e assim diminuindo a latência de comunicação. Na pior das hipóteses, em uma rede sobrecarregada, o VCT funciona como o SAF, sendo todos os seus pacotes obrigados a serem armazenados no buffer.

O chaveamento *wormhole* foi proposto em [DAL 86], sendo este uma alternativa ao chaveamento VCT para a diminuição do tamanho dos *buffers* necessário a este último. Nesta técnica, o pacote é dividido em *flits* (***FL****ow control un****IT***), usualmente consistindo de uma a quatro palavras. O primeiro *flit* contém as informações de cabeçalho, sendo os outros transmitidos após este de maneira pipeline pelo mesmo canal. Sendo assim, os flits de outros pacotes não podem ser transmitidos até que o primeiro seja concluído, o que pode levar a um *deadlock*. O *deadlock* pode ser evitado com a criação de canais virtuais, que compartilham o mesmo canal físico.

2.4.4. Controle de Fluxo

O controle de fluxo gerencia a distribuição dos mesmos recursos (*buffers* e canais) entre diferentes pacotes. Como o número de pacotes em uma rede é, às vezes, maior do que a capacidade de vazão da mesma, faz-se necessário tomar decisões sobre o destino a ser dado a estes pacotes, como descarte, bloqueio no lugar onde estão, recebimento e armazenamento temporário, ou ainda desvio para outros caminhos. O mecanismo de controle de fluxo deve idealmente evitar o congestionamento dos canais e minimizar a latência das mensagens. São duas as técnicas mais utilizadas: a de *canais virtuais* e a *baseada em créditos*.

A técnica de controle de fluxo baseada em créditos é de mais simples implementação. Nela, o módulo receptor envia um sinal para o módulo transmissor informando o espaço disponível no buffer de entrada. Esta informação é vista como crédito. A transmissão de dados ocorre quando existem créditos disponíveis. Esta é uma técnica de controle de fluxo onde não há descarte de dados.

A técnica de canais virtuais é uma otimização para o uso de *buffers* de entrada baseados em FIFO. Neste tipo de memorização, o primeiro pacote recebido é o primeiro a ser transmitido.

Quando o pacote da cabeça da fila não pode ser transmitido porque o *buffer* de sua porta destino está saturado, ou por qualquer outra razão, o pacote fica contido até que seja transmitido. Caso hajam outros pacotes no buffer que tenham outras portas destino, eles estarão bloqueados até que o primeiro seja transmitido. Este problema é conhecido por bloqueio HOL (*Head of Line*). O uso de canais virtuais visa eliminar este problema. Com esta técnica, o *buffer* físico é divido logicamente em diversos canais. Estes canais funcionam efetivamente como pequenas filas, podendo transmitir seus pacotes independentemente da situação das outras filas.

2.4.5. Arbitragem

O mecanismo de arbitragem de um roteador define qual entrada poderá usar qual saída. Enquanto que o roteamento determina por qual porta de saída será transmitido um pacote, a arbitragem determina qual dos pacotes será escolhido para ser transmitido por uma porta de saída. O mecanismo de arbitragem deve definir, por meio de algum algoritmo, qual porta de entrada será servida a cada momento, resolvendo os conflitos no caso de múltiplas requisições simultâneas para uma mesma porta de saída. Uma má escolha no mecanismo de arbitragem pode fazer com que pacotes fiquem esperando indefinidamente por uma porta de saída, jamais sendo servidos. Este problema é conhecido por *starvation*.

A arbitragem pode ser feita tanto de maneira global para todas as portas de um roteador (*arbitragem centralizada*) ou individualmente, cada porta possuindo seu próprio árbitro (*arbitragem distribuída*).

Na *arbitragem centralizada* todas as requisições das portas de entrada são feitas para um único módulo, que concentra as funções de arbitragem e roteamento. Ele identifica a saída mais adequada para cada requisição e ele mesmo determina quais serão servidas, configurando o *crossbar*. Esta técnica de arbitragem é utilizada quando se deseja otimizar o uso do *crossbar* [TAM 93], mas diminui a capacidade de roteamento de pacotes ao longo do tempo.

Na *arbitragem distribuída* existe um módulo árbitro para cada porta de saída do roteador. O árbitro recebe as requisições das portas de entrada e, baseado em alguma política de escalonamento, define qual delas será servida. Esta política de escalonamento, usada em ambas as técnicas de arbitragem, pode se basear em diversos fatores, sendo as mais comuns as de prioridade estática, prioridade dinâmica, escalonamento por *deadline*, FCFS (*First-Come First-Served*), LRS (*Least Recenlty Served*) e *round-robin*.

2.4.6. Memorização

As redes chaveadas por pacote (SAF, VCT e *wormhole*) necessitam que seus roteadores armazenem os pacotes destinados às saídas bloqueadas, para evitar que eles sejam perdidos. O esquema de memorização utilizado tem grande impacto no desempenho da rede como um todo, e na área ocupada em chip por seus roteadores.

A memorização dos pacotes pode ser feita na entrada ou na saída do roteador, de forma centralizada ou distribuída. A forma mais amplamente utilizada é a memorização distribuída na entrada, onde existe um *buffer* para cada uma das portas de entrada. As diferentes metodologias de memorização dizem respeito à organização destes *buffers*.

A estratégia mais comum e simples é o uso de uma simples FIFO. Os pacotes vão sendo armazenados à medida que são recebidos em uma fila, e vão sendo transmitidos os pacotes na cabeça da fila. Esta metodologia possui a mais simples implementação e de menor custo, porém apresenta o problema do bloqueio HOL.

Para contornar este problema, pode-se dividir o *buffer* pelo número de entrada, obtendo assim N *buffers* para cada porta. Cada um destes *buffers* é conectado a apenas uma porta de saída, o que faz com que todos os pacotes em um *buffer* estejam destinados à mesma saída, o que evita o bloqueio HOL. Esta técnica é chamada SAFC (*Statically Allocated Fully Connected*). Porém, seu uso exige maior complexidade do *crossbar* e do controle de fluxo, e a divisão estática do espaço do *buffer* dificulta sua utilização máxima.

Duas estratégias foram criadas a fim de eliminar os problemas do SAFC, porém ambas possuem suas limitações. Na SAMQ (*Statically Allocated Multi-Queue*) as saídas dos *buffers* destinadas a uma mesma porta são multiplexadas, o que elimina apenas o problema da complexidade do *crossbar*. Na estratégia DAMQ (*Dinamically Allocated Multi-Queue*) a divisão do buffer de entrada é feita dinamicamente, de acordo com a necessidade do fluxo. Isto aumenta a utilização do espaço do buffer e reduz o custo do controle de fluxo. Porém, sua implementação física é mais complexa.

3. A Plataforma MP-SoC

3.1. Objetivos

STORM (*MP-**S**oC Direc**T**ory-Based Platf**ORM***) é uma plataforma desenvolvida com o intuito de estudar a viabilidade do projeto e uso de um sistema com características MP-SoC. Por ser uma plataforma, não possui uma arquitetura definida, mas sim um conjunto de módulos e especificações sobre como devem ser utilizados, sendo possível para o projetista criar instâncias de arquitetura com diversas características. Isto não só está de acordo com a atual tendência de projeto baseado em plataforma, como também permite a integração de diferentes módulos e uma maior facilidade na exploração de espaço de projeto.

O desenvolvimento de STORM possui objetivos bem específicos: produzir uma plataforma MP-SoC totalmente operacional e em sintonia com as modernas teorias de projeto de sistemas integrados, resolvendo questões de SoC, NoC e processamento paralelo, assim como a provisão de recursos de hardware para suporte na implementação/adaptação de um sistema operacional. Para tal, foi definido que a plataforma usaria uma NoC para interconexão. Esta é uma das grandes contribuições deste trabalho, tendo em vista que mesmo os trabalhos mais recentes de MP-SoC [BEN 05, TAE 05, LOG 05] se utilizam de barramentos, os quais, como já foi discutido, tendem a ceder lugar para as NoCs como subsistema de interconexão nos futuros SoCs.

O projeto da plataforma prevê a integração de dois diferentes microprocessadores: um desenvolvido no escopo específico do projeto e um comercial e bastante difundido, um módulo de cache, módulo de memória genérico, módulo controlador de cache, módulo de E/S, que provê recursos para todos os acessos extra-chip do MP-SoC, e dois diferentes modelos de NoC, para interconectar todos os módulos. Apesar de em hipótese alguma descartar ou impossibilitar a integração de diferentes módulos, o horizonte deste trabalho prevê apenas a implementação dos módulos supracitados. Um esquema detalhando os módulos integráveis a plataforma e suas relações pode ser visto na **Figura 3.1**.

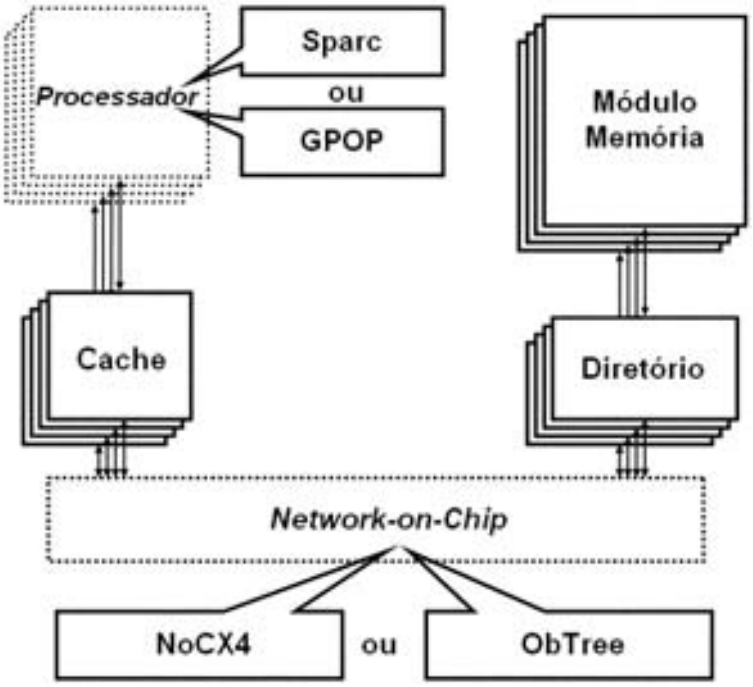


Figura 3.1. Módulos de STORM e sua integração

3.2. GPOP

O *Generic Parallel Oriented Processor*, ou GPOP, é o processador desenvolvido no âmbito deste trabalho. O objetivo de seu desenvolvimento é estudar e implementar as necessidades específicas de um ambiente de processamento paralelo sobre uma NoC, e de seu respectivo sistema operacional. O processador possui uma arquitetura simples, com dois estágios de pipeline. Seu conjunto de instruções segue o modelo RISC, e acessos à memória são feitos apenas via instruções de load/store. Esta decisão foi tomada para a simplificação se seu controle. Como contrapartida, o processador dispõe de um banco de 64 registradores, sendo 4 de propósito específico. Além disso, o seu conjunto de instruções inclui algumas instruções de suporte para o sistema operacional. Também foram implementados alguns mecanismos encontrados em sistemas reais, como interrupções, E/S e proteção de memória. Esta subseção detalha o funcionamento de cada um destes itens.

3.2.1. Unidade de Busca

A Unidade de Busca está diretamente ligada à Cache de Instruções e à Unidade de Execução. Ela mantém um apontador para a próxima instrução a ser buscada, armazenado no registrador PC (*Program Counter*). A cada instrução o conteúdo do registrador PC é

incrementado, mantendo o fluxo de programa. O registrador PC pode ser modificado pela Unidade de Execução no caso de um salto *taken*. Neste caso, a Unidade de Execução envia um sinal *Jump_Taken* e o novo conteúdo de PC para a Unidade de Busca.

O processo de busca de instrução na Cache de Instruções leva 3 ciclos. A Unidade de Busca posiciona um sinal de nova instrução para a Unidade de Execução e fica em espera até que esta última consuma a instrução. Assim que a Unidade de Execução recebe uma instrução, a Unidade de Busca já requisita para a Cache de Instruções a próxima instrução, para que as unidades trabalhem em paralelo.

3.2.2. Unidade de Execução

Esta unidade é responsável pelo real processamento das instruções. Ela está conectada à Cache de Dados e à Unidade de Busca. A Unidade de Execução assume as funções de todos os outros tradicionais estágios de pipeline: Decodificação, Busca de Operandos, Execução e Escrita de Resultados. Seu funcionamento se dá da seguinte forma: assim que recebe uma nova instrução da Unidade de Busca, a unidade verifica o seu opcode. Dependendo do tipo de instrução, a Unidade de Execução pode ir para diferentes estados. Caso seja uma operação lógico-aritmética, que exija o uso da ALU, os operandos são posicionados automaticamente pela seleção dos multiplexadores dos latches de entrada da ALU. No ciclo seguinte, a operação da ALU é executada, e seu resultado é armazenado no latch de saída. No último ciclo, o resultado é armazenado no banco de registradores. As instruções de GPOP possuem 3 endereços.

O acesso à memória é feito apenas através de instruções load e store. O processador dá suporte a 4 tipos de endereçamento: direto, indireto, imediato e indexado (pelo registrador INDEXREG). O dado recebido por um acesso à memória é sempre armazenado no registrador LOADREG. O dado a ser escrito na memória deve ser previamente armazenado no registrador STOREREG, pois o formato das instruções load e store não possui um campo para especificação de registradores. Para obter o melhor desempenho com o processador, faz-se necessário o uso otimizado dos registradores.

Para facilitar a implementação do Sistema Operacional e o processo de migração de processos para diferentes regiões de memória, todos os acessos à memória possuem seu endereço somado ao valor do registrador BASEREG. O registrador BASEREG é de acesso exclusivo do SO e seu valor deve sempre conter o endereço de memória inicial do atual processo em execução. Como todos os programas são compilados supondo seu início a partir da posição zero da memória, este mecanismo evita a necessidade do cálculo de todos os endereços de memória por parte do programa *loader*, o que agiliza a carga de processos. Outro aspecto

vantajoso desta implementação é a grande facilidade no processo de realocação de um processo para uma diferente região de memória. Ao invés de se recalcular todos os acessos à memória, basta para isso mudar o valor do registrador BASEREG.

3.2.3. Interrupção

A cada instrução executada, são testados os vetores de interrupção. Este teste pode ser desativado em condições especiais. As interrupções atualmente existentes são SVC (*Supervisor Call* – Chamada de Modo Supervisor) e de tempo, essenciais para o sistema operacional. GPOP mantém um conjunto de registradores especiais para o suporte de interrupções. O acesso a estes registradores é feito apenas via instruções privilegiadas, que só podem ser executadas pelo sistema operacional em modo supervisor. Os registradores contêm o endereço das rotinas de cada classe de interrupção.

Para possibilitar o escalonamento de processos, GPOP possui um mecanismo de *watchdog*. Este mecanismo funciona como uma contagem regressiva, gerando uma interrupção de tempo quando esta atinge zero. O mecanismo de *watchdog* é programável via instruções privilegiadas pelo software escalonador.

As interrupções de supervisor atualmente previstas pelo processador são: execução de instrução privilegiada, instrução inexistente e acesso indevido à memória. A ocorrência destas interrupções leva o processador a interromper o fluxo de programa, salvando o conteúdo do registrador PC e chamando a rotina de tratamento de interrupção de supervisor, armazenada em um registrador específico, previamente programado pelo sistema operacional durante o boot. A validade de um acesso à memória é verificada pelo mecanismo de proteção de memória.

3.2.4. Instruções Privilegiadas

As instruções privilegiadas são aquelas que têm uso apenas para as funções de sistema operacional, não sendo necessárias nas aplicações de usuário. Ao contrário, a execução destas instruções por aplicações de usuário pode levar o sistema a um funcionamento indevido, sendo necessário restringir a execução destas instruções às rotinas e funções do sistema operacional.

O conjunto de instruções de GPOP possui 5 instruções dessa natureza: TSTSET, TSTRST, WRTBACK, MOVESYS e CTDOWN. As instruções TSTSET (*test-and-set*) e TSTRST (*test-and-reset*) são bastante comuns para a garantia de exclusão mútua entre processos em um SO multitarefa, e em STORM ela é de importância fundamental para a gerência de processos paralelos. Por se tratar de um ambiente sem *broadcast*, sua execução envolve o suporte de quase todos os módulos da plataforma, assim como um tratamento especial por parte do SO. A

instrução WRTBACK (*write-back*) armazena todo o conteúdo *dirty* da Cache de Dados na memória. Esta instrução deve ser usada na desativação temporária de um GPOP do sistema, em um futuro suporte a controle de consumo de energia. A instrução MOVESYS escreve nos registradores de endereço das rotinas de interrupção, e a instrução CTDOWN ativa a contagem regressiva do *Watchdog* durante o processo de escalonamento de processo.

3.2.5. Proteção de Memória

O mecanismo de proteção à memória tem por objetivo evitar que um processo acesse uma posição de memória fora da região de memória alocada para o mesmo pelo SO. Não é difícil imaginar como a escrita em posições de memória reservadas para outros processos pode comprometer o funcionamento dos mesmos. Em um pior cenário, o acesso a posições de memória contendo código e dados do SO pode dar ao usuário o completo controle do sistema.

A implementação do mecanismo é simples e feita sem detrimento do desempenho do processador. O processador possui dois registradores específicos, UPPERREG e LOWERREG, que devem ser atualizados pelo SO a cada vez que um processo ganha o uso da CPU. Estes registradores contêm as posições de memória inicial e final reservadas para um processo. A cada acesso à memória feito, a posição acessada é comparada com o conteúdo dos registradores. Caso ela esteja dentro dos limites estabelecidos pelo SO, o acesso ocorre normalmente. Caso contrário, o acesso é interrompido antes que a requisição chegue à cache, e uma interrupção de supervisor é gerada.

3.3. Sparc

O SPARC (***S****calable* ***P****rocessor* ***ARC****hitecture*) [SUN 92] é uma arquitetura de microprocessador RISC desenvolvida pela Sun Microsystems. É um processador de porte variável, utilizado em sistemas que vão desde sistemas embarcados até grandes servidores. Ele foi escolhido para integração em STORM por ter arquitetura aberta e pela vasta gama de ferramentas disponíveis, dentre elas o compilador de C da GNU, o gcc [GCC 05], o que facilita a simulação de aplicações de maior porte. A integração de diferentes arquiteturas de processadores permite futuros estudos do uso de STORM como uma plataforma heterogênea.

3.3.1. Características

A arquitetura do SPARC V8 é um exemplo de RISC (*Reduced Instruction-Set Computer*) de 32 bits. Essa arquitetura é organizada logicamente em três unidades distintas: a IU (*Integer Unit*), a FPU (*Floating-Point Unit*) e o CP (*CoProcessor*), cada uma com os seus registradores, permitindo máxima concorrência na execução de instruções das três unidades. A **Figura 3.2** mostra essas três unidades.

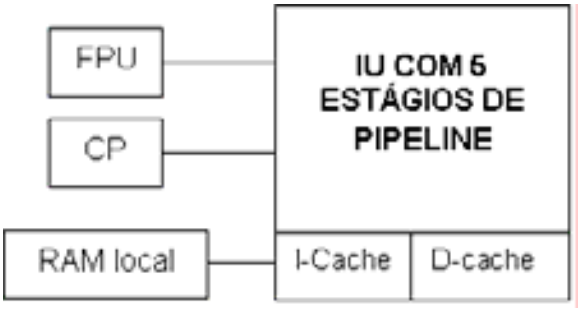


Figura 3.2. Visão geral do SPARC

O SPARC utiliza o modelo de execução registrador-para-registrador, ou seja, todos os acessos à memória são feitos via registradores e todas as instruções envolvem registradores. As instruções do SPARC V8 são executadas em um pipeline de instruções de cinco estágios – busca, decodificação, execução, memória e escrita.

A IU controla todas as operações do processador. Ela executa instruções lógicas e aritméticas com números inteiros; instruções de acesso à memória; atualiza o contador de programa PC (*Program Counter*) e ainda controla a execução da FPU e do co-processador. Para auxiliar na execução de todas essas tarefas, a IU conta com um total de 40-520 registradores, dependendo da implementação. A IU pode ser vista na **Figura 3.3**.

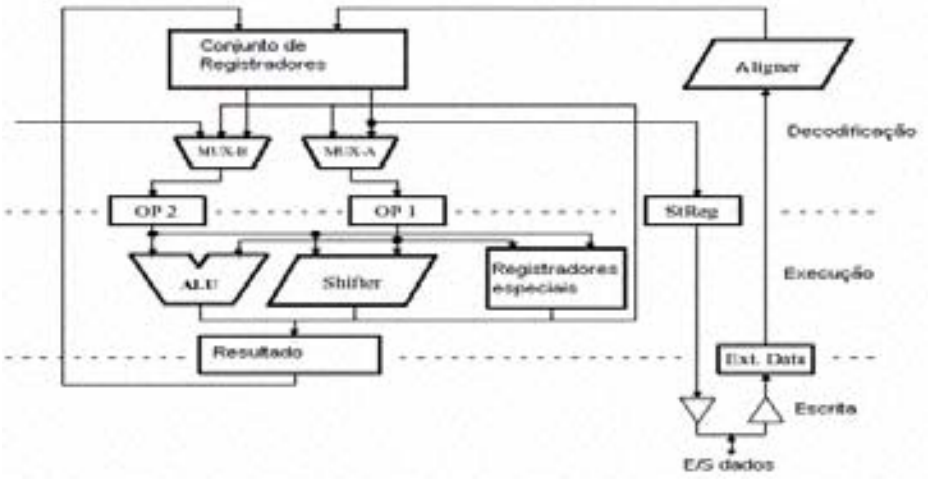


Figura 3.3. Integer Unit do SPARC

Os formatos e dados de ponto flutuante são implementados de acordo com o padrão ANSI/IEEE 754-1985. Entretanto, o SPARC não requer que todos os aspectos desse padrão sejam implementados em hardware, podendo ser emuladas por software as funcionalidades não presentes no software. Nos casos em que a FPU não está presente na implementação ou não esteja habilitada, o software deve emular uma instrução de interrupção de ponto flutuante – *fp_disabled*.

A arquitetura SPARC V8 suporta um único co-processador. Ele tem o seu próprio conjunto de registradores, que geralmente são de 32 bits. Para mover dados dos registradores entre o co-processador e a memória utilizam-se instruções de load/store. Se o co-processador não estiver presente na implementação ou não está habilitado, a interrupção *cp_disabled* deve ser gerada.

Os registradores da FPU e do co-processador são acessados, unicamente, através das instruções de load/store. Não existe nenhum caminho direto entre a IU e a FPU ou entre a IU e o CP, que permita este acesso.

Embora o pipeline de instruções de cinco estágios não seja uma característica obrigatória em um implementação de um processador SPARC V8, a maioria delas apresenta-o em cinco estágios: Busca (B), Decodificação (D), Execução (E), Memória (M) e Escrita (W). Durante a Busca, instruções são movidas da memória para o processador. No estágio de Decodificação, o processador lê os operandos do conjunto de registradores, decodifica instruções e detecta dependências entre instruções. Os operandos entram na ULA (*Unidade Lógica e Aritmética*) ou na unidade de *shift* no estágio de Execução. Instruções de load/store buscam operandos na memória no estágio de Memória. No estágio de Escrita, o processador escreve o resultado da ULA ou o dado da memória no conjunto de registradores.

Uma característica importante do SPARC é a janela de registradores. Ela garante maior agilidade no processo de trocas de contexto, e é particularmente eficiente em linguagens orientadas a objeto, como C++ ou Java. A qualquer momento, uma instrução pode acessar 32 diferentes registradores: sendo 8 deles globais e 24 registradores de janela. Uma janela é composta por 3 tipos de registradores:os de entrada, que possuem os valores passados como parâmetros para uma função; os locais, para armazenamento de variáveis locais; e os de saída, usados para passagem de parâmetros em uma possível chamada de função. Os registradores de entrada e de saída de duas diferentes janelas se sobrepõem, como visto na **Figura 3.4**.

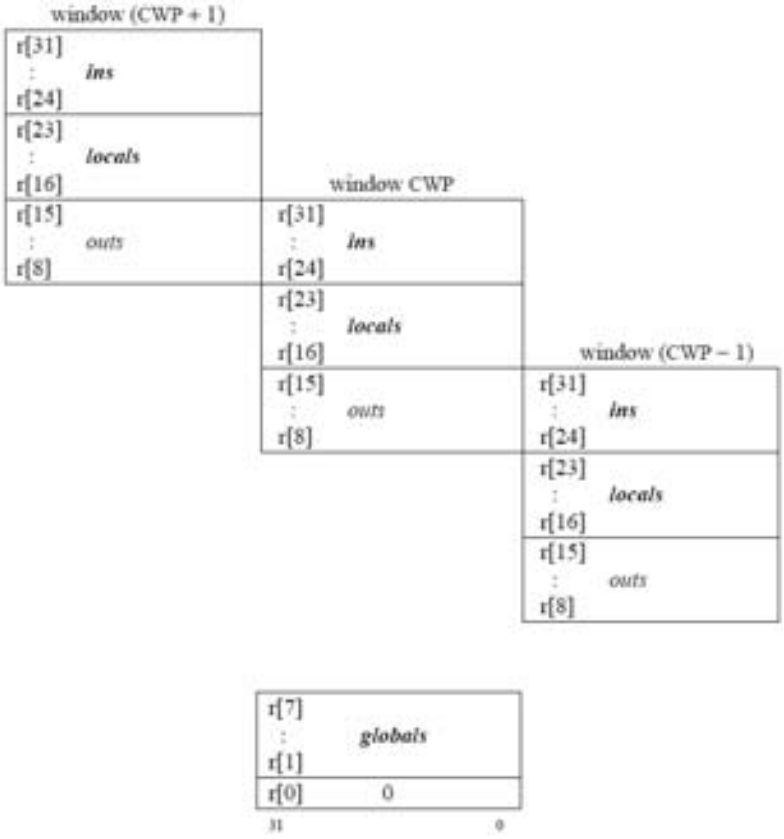


Figura 3.4. Janela de Registradores do SPARC

3.3.2. Implementação

Para a integração em STORM, uma arquitetura do SPARC V8 foi definida de acordo com os propósitos deste trabalho. Esta definição foi posteriormente implementada em SystemC *cycle-accurate*, assim como todo o resto da plataforma.

A arquitetura implementada possui uma unidade inteira, uma unidade de ponto flutuante, cache de instruções e dados. A unidade inteira possui cinco estágios de pipeline: busca, decodificação, execução, memória e escrita, que funcionam da maneira tradicional, descrita na seção anterior. O modelo possui trinta e duas janelas de registradores e 8 registradores globais, totalizando 520 registradores. O tratamento de interrupções também foi implementado.

Uma unidade de ponto-flutuante que executa de maneira concorrente com a IU está presente. Todos os sinais de controles e endereços de acesso à memória são gerados pela IU.

3.3.3 Modo Usuário e Modo Supervisor

A arquitetura SPARC suporta dois modos de operações: o modo supervisor, que é o modo de operação do sistema operacional e tem acesso a todo conjunto de instruções; e o modo usuário, que é o modo de funcionamento das aplicações do usuário. Qualquer tentativa de executar uma operação de modo supervisor em uma aplicação de modo usuário causa uma interrupção.

Os modos de operação delimitam não apenas um conjunto de instruções diferenciados, mas também espaços de endereçamentos de memória diferenciados.

3.3.4. Tratamento de Interrupções

O fluxo de execução do SPARC é baseado no modelo seqüencial, mas uma implementação da arquitetura SPARC com pipeline de instruções pode modificar o estado do processo em uma ordem diferente da definida pelo modelo seqüencial. Em decorrência disto, a ocorrência de uma interrupção pode deixar o hardware em um estado que não é consistente com nenhum valor especificado pelo contador de programa [SMI 88].

Quando uma interrupção ocorre, o estado do processo é salvo pelo hardware ou pelo software. As interrupções podem ser precisas ou imprecisas. Neste trabalho o contexto é salvo pelo hardware. Uma interrupção é dita precisa se obedece às seguintes propriedades:

1) Todas as instruções anteriores à instrução apontada pelo contador de programa foram completamente executadas e modificaram o estado do processo corretamente.
2) Todas as instruções após a instrução apontada pelo contador de programa não foram executadas e não modificaram o estado do processo.
3) Se uma interrupção é causada por uma exceção em uma instrução do programa, o contador de programa salvo aponta para a instrução interrompida. E esta instrução deve ter sido completada ou não iniciada.

Se o estado do processo salvo não obedece a alguma das condições acima a interrupção é dita imprecisa. O tratamento de interrupções imprecisas é dependente da implementação. Este trabalho adota o esquema de conhecido como *future file* [SMI 88].

Neste esquema *future file*, utilizam-se dois bancos de registradores separados: o primeiro reflete o estado da arquitetura seqüencial, e é referenciado como banco de registradores da arquitetura; o segundo é um banco de registradores atualizado sempre que alguma instrução termina, representando o futuro do banco de registradores da arquitetura. Este banco é chamado de banco de registradores de trabalho.

Se um erro ocorre em uma instrução, a mesma é reiniciada e o estado consistente é recuperado do banco de registradores da arquitetura.

3.3.5. Decisões de Projeto

A arquitetura SPARC é aberta e não define o caminho de dados do pipeline que implementa o conjunto de instruções. Isto implica em decisões de projeto que são visíveis ao nível de software. O principal efeito da implementação do pipeline é o número de ciclos necessário para execução das instruções. A Tabela 3.1 mostra estes valores para esta implementação.

Tabela 3.1. Número de ciclos gasto para execução de instruções

Instrução	Número de Ciclos
Load single, taken ticc, JMPL, RETT	*2*
Load double, store single, untaken Ticc	*3*
Store double, LDSTUB, SWAP	*4*
Todas as outras instruções	*1*

Além do número de ciclos gastos em cada instrução, existem outros pontos no SPARC deixados em aberto para os projetistas. A implementação proposta neste trabalho possui as seguintes características:

1) *Previsão de salto*: para evitar bolhas no pipeline devido à *hazards* de desvios, uma instrução é buscada a cada ciclo de *clock* para manter o pipeline cheio. No caso de instruções de desvios, considera-se que o desvio não foi tomado, assim a execução continua seqüencial. Se o desvio for tomado, as instruções que estão no pipeline são descartadas automaticamente.
2) *Mecanismos de bypass*: bolhas no pipeline também podem ser causadas devido à ocorrência de uma instrução que necessite de dados que estão prontos mas ainda não estão disponíveis nos registradores. Para evitar a ocorrência deste tipo de bolha a arquitetura conta com um mecanismo de *bypass* entre os estágios de decodificação, execução e memória.
3) *Redução do atraso dos desvios*: este método consiste em mover a avaliação e execução do desvio para o estágio de decodificação. Isto permite que o atraso na execução de desvios tomados seja de apenas um ciclo.

3.4. Cache

O uso de memória cache é feito desde que o acesso à memória principal tornou-se muito custoso por causa da diferença entre os períodos de relógio do processador e da memória. Para contornar este problema, criou-se uma pequena memória próxima do processador, com tempo de acesso muito menor, baseando-se no princípio da localidade [TAN 99]. O uso desta memória, chamada de memória cache, aumentou consideravelmente o desempenho do sistema, e hoje em dia sua presença é quase obrigatória nos sistemas computacionais.

Cada GPOP possui um módulo de memória cache L1 agregado. A cache é divida em duas memórias: Cache de Instruções e Cache de Dados, que apesar de serem efetivamente dois blocos possuem características semelhantes. Esta divisão na cache (*split*) é comum à maioria das arquiteturas pipeline atuais pois possibilita acessos simultâneos a instruções e dados. As memórias cache são completamente associativas. O seu algoritmo de substituição de páginas (LRU, LFU e FIFO), tamanho e tamanho de linha são configuráveis, podendo cada instância definir aquelas características que mais se adequam aos seus objetivos. Cada cache pode ter diferentes configurações.

O módulo de cache, além de armazenar informações para acesso mais rápido por parte do processador, também é responsável pela conexão entre o processador e seu roteador. Para tal, além de possuir os módulos de Cache de Instruções e Cache de Dados, a cache também possui um gerenciador de comunicação, CaCoMa (*Cache Communication Manager*). O CaCoMa não apenas se comunica com o roteador, como também armazena informações essenciais sobre o sistema. Um esquema completo contendo todos os módulos que compõem a cache é mostrado na **Figura 3.5**.

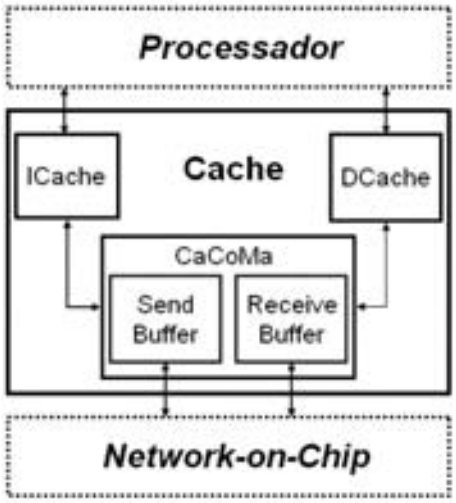


Figura 3.5. Cache de STORM e seus módulos internos

3.4.1. Cache de Instruções

A Cache de Instruções é, como seu nome indica, dedicada a armazenar as instruções executadas pelo processador mais recentemente. Sua implementação é uma simplificação a Cache de Dados, tendo em vista que a Cache de Instruções não permite operações de escrita, apenas de leitura. Por causa disso, seu controle de diretório é muito mais simples, pois ela nunca possui blocos *dirty*, apenas cópias *clean* dos blocos. Suas operações de diretório se resumem a *Read Hit* e *Read Miss* (detalhados na **Seção 3.5.1**). A Cache de Instruções é acessada exclusivamente pelo módulo de Busca do processador, permitindo que as instruções sejam acessadas em paralelo com os dados. Como já foi dito, os parâmetros da Cache de Instruções são configuráveis.

3.4.2. Cache de Dados

A Cache de Dados é bem mais complexa do que a Cache de Instruções, pois ela possui suporte a todas as operações do diretório. Ela é destinada ao armazenamento dos dados mais recentemente acessados pelo processador. Por armazenar dados, é necessário que ela suporte operações de escrita, além de leitura. Para isso, seus blocos, além de possuírem o tradicional bit de validade, para indicar se os dados contidos em determinada linha estão válidos, possuem também um bit de permissão de escrita, que indicam se uma operação de escrita pode ser feita sobre aquele bloco (ou seja, se o bloco está *dirty* no diretório). A implementação destes mecanismos de validade do bloco/permissão de escrita tornam o controle da Cache de Dados mais complexo. A Cache de Dados é acessada exclusivamente pelo módulo de Execução do processador.

3.4.3. CaCoMa

O CaCoMa (*Cache Communication Manager*) é um módulo especialmente desenvolvido para os propósitos da plataforma. Ele possui como função principal a gerência da comunicação das memórias cache (e consequentemente, do processador) com seu roteador, e com o resto da plataforma. O CaCoMa está ligado às caches de dados e instruções, e ao roteador da NoC.

O CaCoMa é o módulo que traduz o endereço lógico de uma posição de memória para seu endereço físico, e que recebe os pacotes da NoC e os trata. Por causa disso, todas as operações de acesso a memória do processador passam pelo CaCoMa, excetuando-se, naturalmente, aquelas que resultam em acerto na cache: *Read Hit* e *Write Hit*. A tradução entre os endereços

lógico e físico se dá pelo uso de uma tabela ATA (*Adress TAble*), montada durante o processo de inicialização da plataforma.

A ATA contém o endereço de NoC de todos os módulos de memória da plataforma. Além disso, possui também o último endereço lógico de cada memória. Com isto, é possível traduzir um endereço lógico contido em uma instrução de acesso a memória para um endereço físico, ou seja, o módulo de memória que possui aquele dado e em qual posição. Um exemplo de ATA pode ser visto na **Tabela 3.2**. Este exemplo supõe a instância da plataforma ilustrada na **Figura 3.6**. Nesta figura, os módulos de diretório e cache não são explicitados, ficando já implícita sua integração com os módulos de memória e GPOP, respectivamente.

Tabela 3.2. Exemplo de ATA

Endereço Virtual	Endereço Físico
0x000003FF	0,2
0x000007FF	1,2

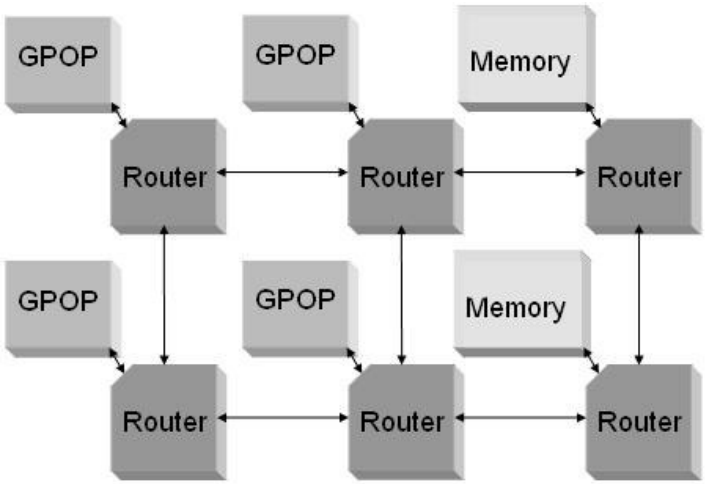


Figura 3.6. Instância 3x2 de STORM

A ATA mostrada supõe que os módulos de memória possuem 1k posições. Portanto, um *Read Miss* da posição lógica 0x00000790 será convertido em um pacote de requisição de bloco que contém a posição 0x00000390, a ser enviado para o módulo de memória encontrado na posição {1,2} da NoC. De forma análoga, um *Write Hit Without Permission* da posição lógica 0x00000350 será convertido em um pacote de requisição de escrita do bloco que contém a posição 0x00000350 que será enviado para o módulo de memória encontrado na posição {0,2} da NoC.

3.5. Diretório

O módulo de Diretório foi desenvolvido com o propósito específico de centralizar as informações sobre os blocos de um módulo de memória, mantendo assim a coerência de cache. A cada módulo de memória está conectado um módulo Diretório, que controla seu funcionamento e faz sua conexão com a NoC e, portanto, com o resto da plataforma. O módulo Diretório mantém informações sobre todos os blocos daquele módulo de memória, e assim é capaz de implementar coerência entre todas as caches do sistema. Porém, o algoritmo de diretório não funcionaria sem algumas modificações por parte das memórias cache, para dar suporte a suas operações.

Para manter informações sobre os blocos e implementar o método de diretório são necessárias duas tabelas: STA (*Status TAble*) e PTA (*Processor TAble*). A primeira contém informações sobre todos os blocos de uma memória, contendo uma linha para cada bloco. Cada linha possui um número *n+1* de colunas de 1 bit, sendo *n* o número de processadores do sistema. Por isso, o diretório é classificado como Dir_n *NB* (*Non-Broadcasting*). Esta sobrecarga (*overhead*) de memória é necessário para que um bloco de memória possa ser compartilhado por todas as caches do sistema. Porém, ele é atenuado pelo fato de se agrupar as posições de memória em blocos. Quanto maior o tamanho do bloco, menor será este overhead. Uma opção para diminuição do overhead é limitar o número de cópias por blocos.

O primeiro bit de cada linha da STA é o *Dirty Bit*, que indica de o bloco está *clean* ou *dirty*. Um exemplo de STA do módulo de memória {0,2} considerando a plataforma da **Figura 3.6** pode ser visto na **Tabela 3.3**. A tabela representa apenas um trecho da STA, tendo em vista que se faz necessária uma linha para cada bloco. No exemplo, pode-se ver que os blocos 2 e 6 estão *dirty*, e os demais *clean*. Cada bloco *dirty* só pode ter um dono atual, aquela cache que possui permissão de escrita sobre ele. Os blocos *clean* podem estar em apenas uma cache (bloco 1 e bloco 3), várias caches (bloco 0 e bloco 5), todas as caches (bloco 4) ou em nenhuma (bloco 7).

Tabela 3.3. Exemplo de STA

Dirty	P0	P1	P2	P3
0	1	1	1	0
0	0	1	0	0
1	0	0	1	0
0	0	0	1	0
0	1	1	1	1
0	0	1	0	1
1	1	0	0	0
0	0	0	0	0

A STA é necessária para que o Diretório saiba quando um bloco foi modificado e não atualizado (*dirty*) ou quando ele está atualizado na memória (*clean*), e qual cache possui uma cópia dele, para que possa enviar os pacotes de *write-back*, *write permission*, etc (ver seção seguinte).

Porém, a informação sobre os blocos é inútil sem os endereços de NoC dos processadores, já que seria impossível enviar os pacotes para os destinos corretos. Para isto, o Diretório mantém a PTA. A PTA, ao contrário da STA, é uma tabela estática, montada durante a inicialização do sistema (assim como a ATA dos CaCoMas). Ela possui apenas o endereço de NoC dos processadores, para que os campos P*X* da STA se refiram a um endereço conhecido. A PTA da plataforma da **Figura 3.5** é mostrado na **Tabela 3.4**.

Tabela 3.4. Exemplo de PTA

Processador	Endereço
P0	0,0
P1	0,1
P2	1,0
P3	1,1

3.5.1. Operações

Para o funcionamento do método de diretório para coerência de cache é necessária uma ação integrada entre Caches e Diretórios. As operações de diretório precisam do suporte por ambas as partes. Existem 5 operações básicas, algumas com subcasos, para a implementação do mecanismo de diretório. Uma representação gráfica de todos os casos pode ser vista na **Figura 3.7** e na **Figura 3.8**. Eles são:

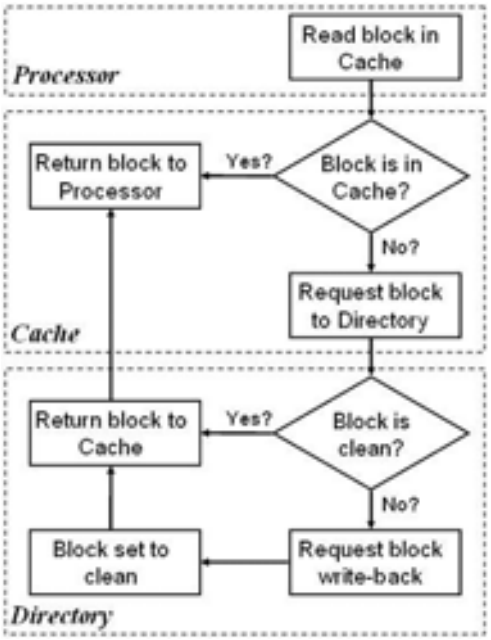


Figura 3.7. Diagrama de Leitura usando Diretório

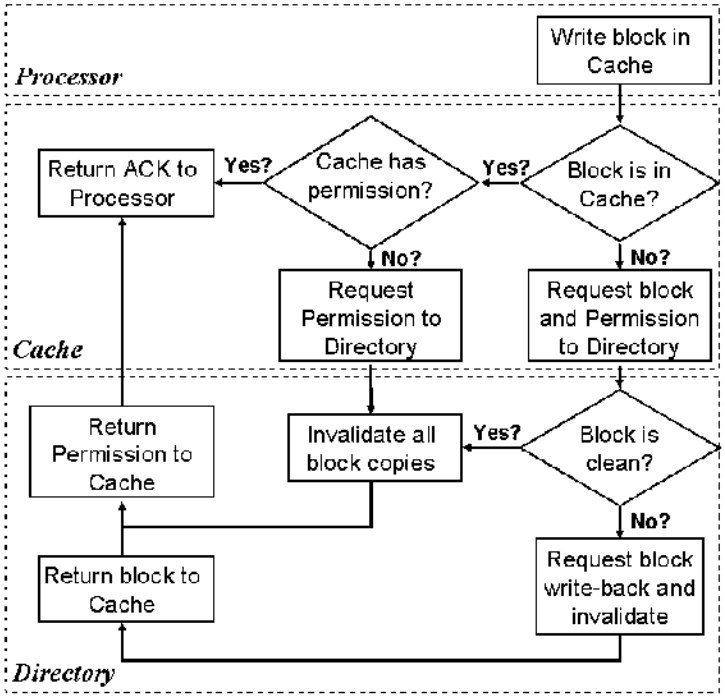


Figura 3.8. Diagrama de Escrita usando Diretório

Read Hit: A leitura de uma posição de memória. O bloco requisitado está na cache, a qual simplesmente retorna o dado para o processador.

Read Miss: O mesmo que acima, porém o bloco que contém o dado lido não está na cache. A Cache faz uma requisição do bloco para o respectivo diretório. Caso o bloco esteja *clean*, o

Diretório envia uma cópia do bloco para a Cache e atualiza o status do bloco Caso o bloco esteja *dirty*, o Diretório envia um pacote *write-back not invalidate* (write-back sem invalidação do bloco que está na Cache) para o atual dono do bloco. O dono então perde sua permissão de escrita no bloco, porém ainda o possui válido na Cache. O bloco é atualizado na memória e enviado para a Cache leitora.

Write Hit with Permission: Uma operação de escrita em uma posição de memória. Neste caso o bloco está na Cache, que já possui permissão de escrita naquele bloco (o que significa que o bloco está *dirty* e o dono é aquela Cache). A Cache então simplesmente escreve o novo dado.

Write Hit without Permission: Como acima, o bloco está na Cache, mas não existe permissão para escrita (o bloco está *clean*). A Cache então envia um pacote de *write-request* (requisição de escrita) para o Diretório respectivo, o qual invalida todas as outras cópias daquele bloco em outras caches. Quando o processo de invalidação termina, o Diretório garante a permissão de escrita naquele bloco para a Cache, e marca o bloco como *dirty*.

Write Miss: Novamente, o processador executa uma operação de escrita de um dado, porém o bloco que o contém não está na Cache. A Cache então envia um pacote de *write-read* (leitura com permissão de escrita) para o respectivo Diretório. Se o bloco estiver *clean*, todas as suas cópias são invalidadas (como no caso acima) e o bloco é enviado para a Cache já com permissão para escrita. Se o bloco estiver *dirty*, um pacote *Write-Back Invalidate* (write-back com invalidação do bloco) é enviado para a Cache dona do bloco, que envia o bloco atualizado para o Diretório e o invalida. O bloco então é atualizado na memória e enviado para a Cache requisitante, já com permissão de escrita. Em ambos os casos, o bloco é marcado como *dirty*.

3.5.2. Exemplos de Operações

Esta seção apresenta dois exemplos práticos do funcionamento do diretório. Eles supõem uma versão hipotética da plataforma com apenas dois processadores e uma memória (e consequentemente um diretório). Esta memória possui 8 blocos de dados, e as caches dos processadores possuem 4 linhas de 1 bloco cada.

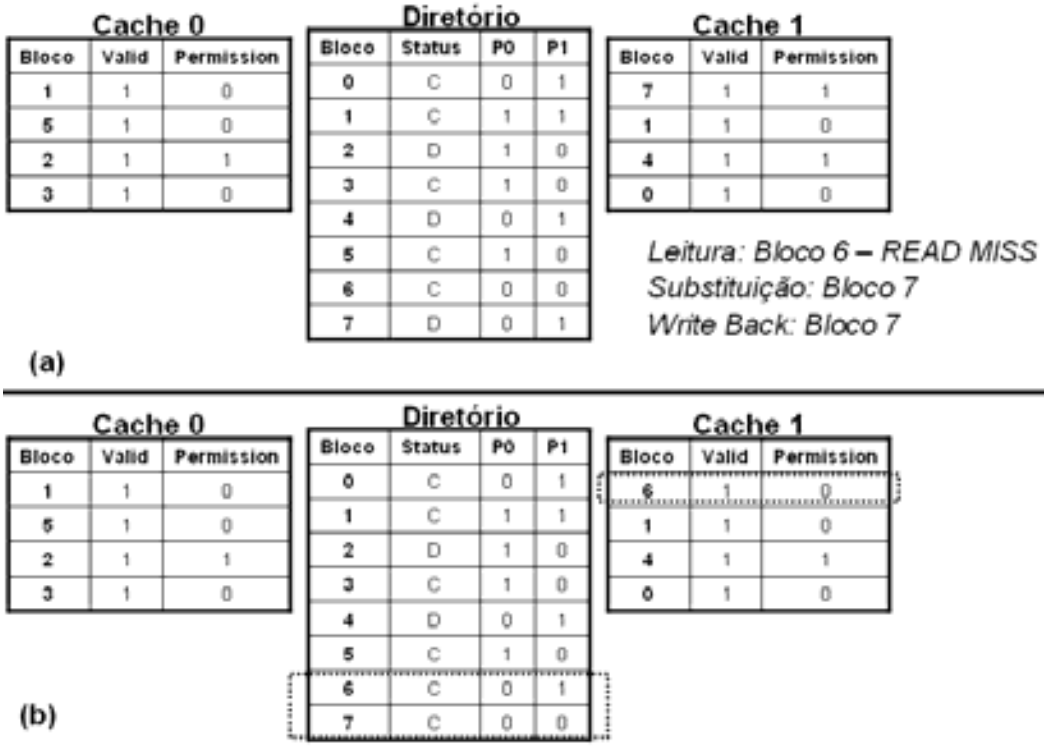


Figura 3.9. Exemplo de Read Miss

Após algum tempo de execução, o sistema se encontra no estado ilustrado na **Figura 3.9 (a)**. As memórias cache de ambos os processadores estão lotadas, com todas as linhas ocupadas. O processador P1 (Cache 1) executa uma instrução que requer a leitura de um dado no Bloco 6. Nota-se que o Bloco 6 não está na Cache 1, o que configura uma falha de leitura (*Read Miss*). Como todas as linhas estão ocupadas, o mecanismo de substituição de linhas da Cache 1 é acionado. Ele identifica que o Bloco 7 deve ser retirado para o recebimento do Bloco 6. Como o Bloco 7 está *Dirty* (modificado e ainda não atualizado na memória) é necessário que a Cache 1 faça uma atualização do valor atual do Bloco 7 para a memória antes de excluí-lo, caso contrário as modificações feitas naquele bloco seriam perdidas. O Bloco 7 é então escrito na memória e marcado como *Clean* (memória contém o valor atual do bloco) e sem cópias. A memória envia o Bloco 6 para a Cache (sem permissão de escrita, pois o acesso foi uma leitura).

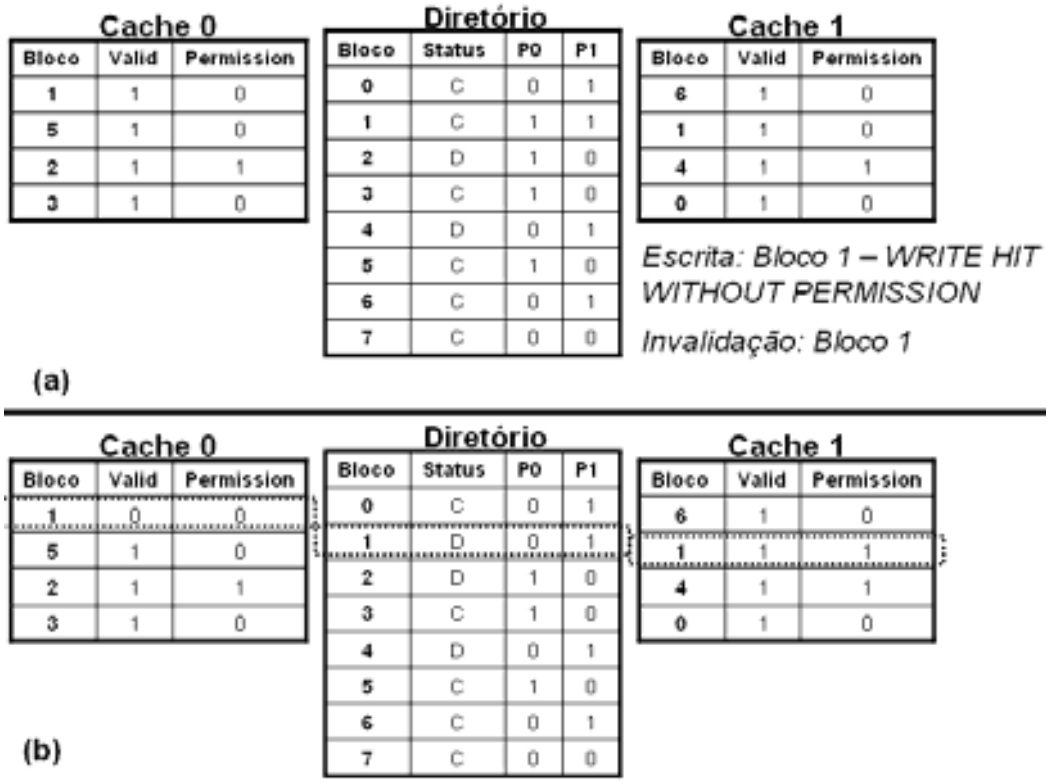


Figura 3.10. Exemplo de Write Hit Without Permission

Após a operação de leitura, o sistema se encontra no estado mostrado na **Figura 3.10 (a)**. Desta vez o processador P1 (Cache 1) realiza uma operação que requer escrita em um dado do Bloco 1. Este bloco já está válido em sua cache, que apenas envia para o Diretório um pedido de escrita. Porém, a Cache 0 também possui uma cópia deste mesmo bloco. Para evitar inconsistência no sistema de memória, o Diretório é obrigado a invalidar a cópia do Bloco 1 da Cache 0 antes de enviar a permissão de escrita para a Cache 1. Após receber um pacote de invalidação do Bloco 1, a Cache 0 marca esta linha como inválida. Só então o Diretório envia para a Cache 1 a permissão de escrita no Bloco 1, e o marca como *Dirty*.

3.5.3. Tempos Mínimos

Neste capítulo serão apresentados resultados de simulação da especificação executável de STORM obtida com sua implementação em SystemC [OSI 05]. Os detalhes desta implementação serão discutidos no **Capítulo 4**, e seus resultados de simulação apresentados no **Capítulo 5**. Os resultados apresentados neste capítulo abordam apenas os módulos individualmente, havendo pouca ou nenhuma interferência do resto do sistema nestes resultados. Portanto, eles serão apresentados juntamente da descrição do módulo para que o

leitor possa avaliar seu funcionamento pontualmente e consiga interpretar melhor os resultados obtidos com a simulação completamente integrada dos módulos, mostrados no **Capítulo 5**.

Esta seção apresenta os *tempos mínimos*, em ciclos, de todas as operações do Diretório. Estes tempos são medidos seguindo alguns princípios:

1) *Simulação em uma instância com 2 processadores e 1 memória*. Quando utilizada a NoCX4, os processadores são posicionados vizinhos à memória (um de cada lado). No caso da Árvore Obesa, todos os módulos são conectados ao mesmo roteador. Desta forma tem-se a menor influência possível nos resultados devido à distância topológica.
2) *Injeção especial de dados na plataforma* para que se tenha certeza de que os processadores e o diretório não realizam nenhuma outra operação que não a operação simulada. Isto garante que os resultados não serão distorcidos por causa de operações indesejadas durante a simulação.
3) *Resultados medidos a partir do recebimento da requisição pela Cache até a resposta da mesma* para que não haja influência do processador utilizado.

Os resultados obtidos nesta simulação são mostrados na **Tabela 3.5**. Eles representam o custo agregado ao uso do Diretório, essencial na manutenção da coerência de cache em um sistema desprovido de *broadcasting*.

Tabela 3.5. Tempos Mínimos para operações de Diretório (em ciclos)

Operação	NoCX4	Árvore Obesa
Read Hit	**1**	**1**
Read Miss Clean	**34**	**28**
Read Miss Dirty	**56**	**44**
Write Hit com Permissão	**1**	**1**
Write Hit sem Permissão	**25**	**19**
Write Miss Clean	**35**	**29**
Write Miss Dirty	**57**	**45**

3.5.4. Test-and-Set

Além de manter a coerência entre as memórias cache do sistema, uma plataforma de processamento paralelo como STORM deve prover suporte de SO para prevenir a ocorrência de condições de corrida (*race condition*) e gerenciar o sincronismo entre processos. A condição de corrida já é um problema em sistemas mono-processados multitarefa, se tornando uma questão

crítica em um ambiente realmente paralelo. Uma solução simples encontrada para este problema, utilizada em STORM, são instruções de *Test-and-Set*.

O processador GPOP possui duas instruções especiais, *Test-and-Set* e *Test-and-Reset*, as quais não poderiam ser implementadas sem adaptações na Cache e no Diretório. As duas são instruções privilegiadas, que precisam ser executadas em modo supervisor. Quando uma instrução *Test-and-Set* é executada, uma interrupção SVC ocorre e uma rotina especial de SO é executada para prosseguir com o processo. Uma vez em modo supervisor, um sinal é enviado do processador para o CaCoMa, que cria um pacote especial de *Test-and-Set* e o envia para o respectivo Diretório. O Diretório vê a posição de memória enviada no pacote e a testa. Caso ela possua o valor '0' armazenado, seu valor é modificado para '-1' e o Diretório responde com um pacote de OK para o CaCoMa que fez a operação. O SO então retorna o fluxo de programa para o processo que originou a chamada SVC, e este continua sua execução. Caso a posição de memória não esteja em '0', o Diretório envia um pacote de NOT OK de volta para o CaCoMa, e o SO o coloca em uma fila de processos em espera. Quando um processo sai da região crítica, ele deve executar a instrução *Test-and-Reset*, posicionando assim o valor da posição de memória em '0' e liberando o *flag*.

Este é uma solução que necessita que o programador especifique as regiões críticas em linguagem de alto nível, fazendo para isto uso de semáforos ou monitores, que serão transformados pelo compilador em chamadas ao sistema operacional via as instruções Test-*and-Set* e *Test-and-Reset*. Esta também é uma solução integrada entre arquitetura e SO, e não funcionará se o SO não possuir algum tipo de proteção no acesso às *flags* de semáforo por instruções de leitura e escrita comuns. Caso contrário, um código malicioso poderia escrever nas posições de memória que contém as *flags*, prejudicando completamente o funcionamento do sistema de *flags*.

O mecanismo de *Test-and-Set* é implementado no SPARC usando-se as instruções LDSTUB (*Atomic Load-Store*) e SWAP (Troca conteúdos de registrador e memória atomicamente). Quando executada, a instrução LDSTUB lê uma posição da memória e, em caso esta seja '0', modifica seu valor para '-1'. O suporte de Cache e Diretório funcionam de maneira idêntica ao TST do GPOP. De maneira análoga, a instrução SWAP é utilizada para escrever novamente '0' na posição de memória após a saída da região crítica. Uma rotina de espera pelo acesso à região crítica deve ser escrita em software.

3.6. Memória

Existe um consenso no ambiente acadêmico, no qual o modelo de memória distribuída é de programação mais difícil, porém proporciona enorme escalabilidade. Por outro lado, os modelos

de memória compartilhada são de mais fácil programação, mas sua escalabilidade é severamente limitada. Esta linha de raciocínio vem de muitas décadas, na época onde os sistemas multiprocessador, de memória compartilhada, utilizavam necessariamente barramentos. O advento dos SoCs e das redes-em-chip (NoC) pode mudar este paradigma e agregar as vantagens dos dois sistemas.

STORM faz uso de memória compartilhada, onde diversos blocos de memória espalhados pela NoC formam um único espaço de endereçamento, visível por todos os processadores. O modelo de memória compartilhada é o mais comum em arquiteturas multiprocessador devido a sua maior facilidade de programação. Neste modelo não existem memórias locais ou remotas, assim todos os processadores possuem livre acesso a todo o espaço de endereçamento (porém os processos específicos possuem regiões de memórias restritas, definidas pelo SO). Por isso, o acesso à memória pode ser feito com instruções comuns de Load e Store, sem haver necessidade de maiores preocupações com a localização dos dados por parte do programador. Apesar do uso de memória compartilhada, sua escalabilidade não é comprometida, pois o uso de diretório para manutenção da coerência de cache elimina o gargalo dos protocolos snoop: o broadcast.

Os blocos de memória podem ser de tamanho variado, e podem ser alocados em qualquer posição na NoC (exceto na posição {0,0}, reservada para o processador de boot – ver a **Seção 3.8**). As memórias são dividas em blocos, a unidade de transferência de dados na plataforma. O tamanho dos blocos é o mesmo para a memória e para as caches, e pode variar em diferentes instâncias da plataforma. Cada módulo de memória possui necessariamente um módulo Diretório agregado.

3.7. Interconexão

O sistema de interconexão é um dos principais aspectos no desenvolvimento dos sistemas-em-chip. Por causa da limitada escalabilidade dos barramentos e a tendência de popularização das redes-em-chip em um futuro próximo, optou-se pelo uso destas últimas para interconexão dos módulos de STORM. Dois diferentes modelos de NoC foram desenvolvidos no escopo deste trabalho: a NoCX4, uma grelha, e a Árvore Obesa, de uma topologia indireta própria, proposta por este autor. A integração de dois modelos de NoC de topologias distintas corrobora a facilidade de exploração de espaço de projeto na plataforma.

3.7.1. NoCX4

A NoCX4 é uma Network-on-Chip desenvolvida mais recentemente no âmbito deste trabalho [SOA 03-1]. Seu propósito inicial era a análise de sua integração na arquitetura

reconfigurável X4CP32 [SOA 03-2, SOA 03-3]. Porém, ela é bastante genérica para a integração em um MP-SoC, e o know-how já adquirido com trabalhos anteriores a tornou a melhor opção como subsistema de interconexão em STORM.

Ela consiste em uma grelha de roteadores, cada um ligado a um módulo da plataforma (topologia direta). Cada roteador possui endereço único na NoC, baseado na sua posição na grelha, o que faz com que cada endereço corresponda a um módulo. Seus pacotes, compostos por palavras de 32-bits, consistem em uma palavra de cabeçalho e as outras de carga efetiva, sendo que o campo de carga pode ter tamanho variável, de zero (pacotes compostos somente pelo cabeçalho) até o tamanho do bloco. O formato do pacote é mostrado na **Figura 3.11**.

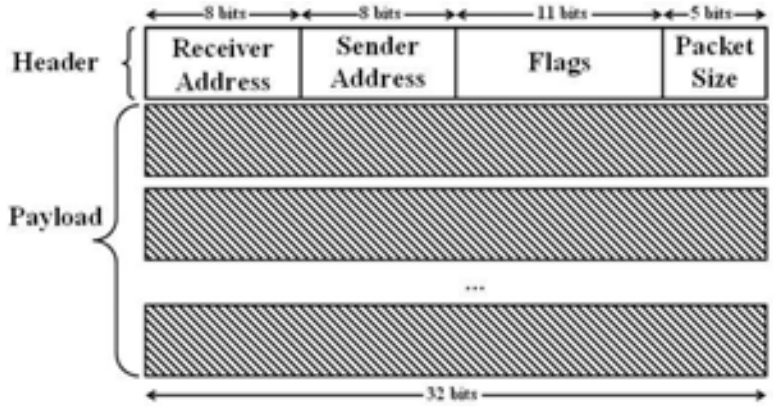


Figura 3.11. Formato de Pacote da NoCX4

Os roteadores possuem 5 portas: norte, sul, leste, oeste e módulo. As quatro primeiras o conectam a outros roteadores da NoC, enquanto que esta última faz a conexão com o módulo a ele agregado. Todas as portas consistem em dois canais *half-duplex* de 32-bits, o que significa que uma capacidade de transmissão de uma palavra de um pacote por ciclo em cada direção.

A NoCX4 usa roteamento determinístico XY. Apesar de não possuir mecanismos para se adaptar ao fluxo na rede, a exclusão do problema de deadlock e a sua simplicidade de implementação tornam este tipo de roteamento atraente. Os pacotes são transmitidos por chaveamento store-and-forward. Este tipo de chaveamento é mais adequado do que o chaveamento por circuito, pois o tamanho reduzido dos pacotes transmitidos em STORM não justifica a completa alocação de um canal para sua transmissão. Apesar de não possuir o mesmo desempenho do chaveamento wormhole, sua maior simplicidade e ausência de deadlock foram decisivas para a sua escolha. O controle de fluxo em NoCX4 é feito com base em créditos, no qual não existe descarte de pacotes.

3.7.1.1. Roteador

O módulo roteador de NoCX4 é simples, composto por apenas alguns módulos internos. Sua arquitetura pode ser vista na **Figura 3.12.**

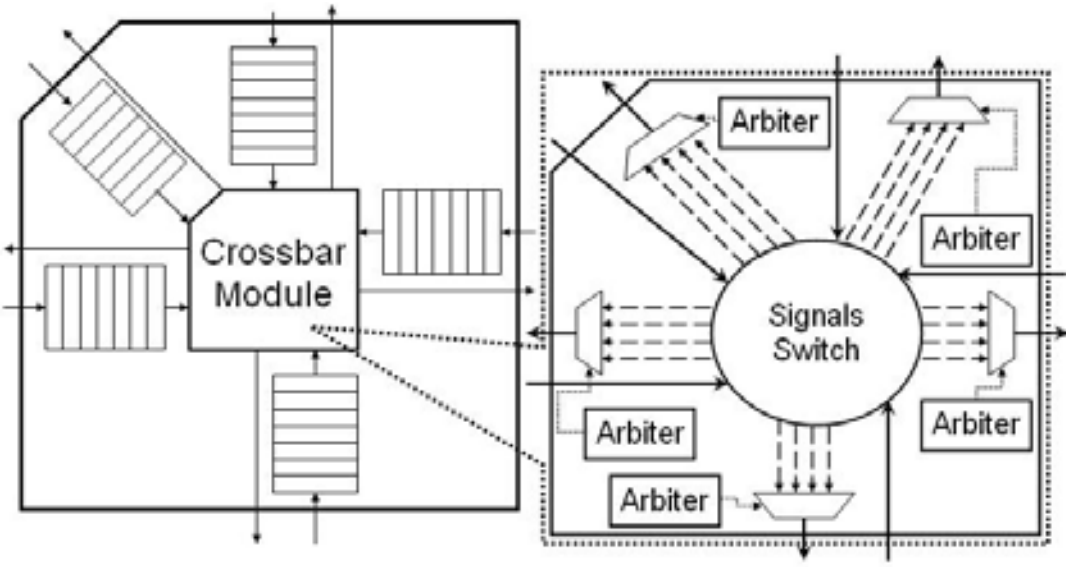


Figura 3.12. Roteador da NoCX4

Cada roteador possui 5 portas de comunicação, sendo que 4 delas o conectam a outros registradores, e a quinta o conecta a um módulo da plataforma. Cada porta é composta por dois canais simples, um de entrada e um de saída. As entradas são bufferizadas, ou seja, existe um buffer para armazenar os dados recebidos até que sejam transmitidos. Atualmente, dois tipos de buffer podem ser utilizados: FIFO (*First-In First-Out*) e SAFC (*Statically Allocated, Fully Connected*). O algoritmo de chaveamento é implementado no buffer, que faz uma requisição de envio para o árbitro apropriado.

As saídas dos *buffers* são ligadas a um módulo *Crossbar*, que funciona também como árbitro e controlador de fluxo. No interior do módulo *Crossbar* existem cinco multiplexadores, sendo cada um o controlador de uma das portas de saída do roteador. Cada multiplexador recebe como entrada as saídas de todos os *buffers* de entrada, exceto aquele correspondente à sua porta (pois um pacote não pode ser roteado de volta; o pacote não pode entrar e sair pela mesma porta). Cada um dos multiplexadores é controlado por um árbitro, que funciona também como controlador de fluxo. O árbitro recebe informação dos créditos (quantidade de posições livres no buffer de entrada) de seu roteador vizinho.

Cada vez que um buffer faz uma requisição de envio para um árbitro, este último avalia o tamanho do pacote a ser transmitido e o espaço no buffer de entrada vizinho. Caso a transmissão

do pacote seja possível, o árbitro declara o buffer como “elegível” para o envio. Um dentre os buffers “elegíveis” é escolhido para realizar o envio, baseando-se numa política *round-robin* sem prioridade. O buffer eleito começa então sua transmissão sem interrupções, pois aquela porta de saída fica alocada para ele. Quando o pacote inteiro é transmitido, o buffer verifica a existência de outros pacotes para serem enviados. Caso não haja nenhuma requisição ou nenhuma das requisições seja elegível, não há transmissão de dados naquele ciclo. Este mecanismo distribuído de arbitragem permite que todas as portas de comunicação sejam utilizadas ao mesmo tempo.

3.7.1.2. Desempenho

O uso de FIFO, porém, causa retenção desnecessária de dados porque precisa que o dado no topo da fila seja transmitido antes de qualquer outro. No caso de uma porta de saída estar congestionada, um pacote destinado àquela porta certamente bloqueará o fluxo de dados, mesmo que depois deste pacote hajam outros que serão roteados para portas descongestionadas. Este problema é conhecido como *bloqueio HOL* (*Head of Line*). Por causa deste problema, a NoCX4 permite a integração de bufferização SAFC. Neste tipo de bufferização, cada entrada passa a possuir um buffer para cada porta de saída. Estes *buffers* são exclusivos, e neles não ocorre roteamento, pois ele só está ligado a uma saída específica. O roteamento é feito pelo roteador anterior, pois o pacote já deve ser escrito no buffer de entrada correto. A bufferização SAFC possui alguns inconvenientes, como o aumento da complexidade do controle de fluxo, agora multiplicado pelo número de entradas, e do núcleo de chaveamento. Porém, seu desempenho pode compensar estas eventuais desvantagens.

Para verificar o desempenho do modelo NoCX4, como um todo, e de seus dois tipos de buferização, foi feita uma implementação da NoCX4 em SystemC *cycle-accurate* e realizado um conjunto de simulações. Uma implementação em SystemC foi preferida ao invés de uma em HDL por causa da facilidade em simular milhões de ciclos sem perda de precisão de ciclos, algo necessário para medir o ganho do uso da buferização SAFC.

As simulações foram feitas em uma matriz 3x3 de elementos geradores e consumidores de pacotes, representando diferentes módulos integrados à NoC. Foi definido um padrão para a comunicação entre estes elementos, conforme mostrado na **Figura 3.13**. Dentro de cada quadrado, que representa um elemento de comunicação, são mostrados, respectivamente, o número de elementos para os quais ele envia pacotes, o número de elementos que enviam pacotes para ele, e o número de rotas de comunicação seu roteador participa, sem contar aquelas que têm como destino o próprio elemento. Entre parênteses estão as percentagens de pacotes enviados, pacotes recebidos e pacotes roteados sobre o total do sistema. Para melhor

entendimento: o elemento {0,0} (superior esquerdo) da matriz gera pacotes destinados a apenas um outro elemento, precisamente 10% de todos os pacotes gerados em todo o sistema, não recebe pacotes e seu roteador não faz parte de nenhuma rota de comunicação entre outros elementos. Estes parâmetros foram criados com o objetivo de ter um elemento gerando grande quantidade de pacotes (o elemento {0,1}), um elemento recebendo grande quantidade de pacotes (o elemento {1,0}) e um gargalo em um roteador (o roteador {1,1}). Com este perfil de simulação, tem-se efetivamente 50% de todos os pacotes passando pelo roteador {1,1}, sendo 35% deles destinados a outros roteadores e mais 15% destinados ao seu elemento de comunicação – um grave gargalo.

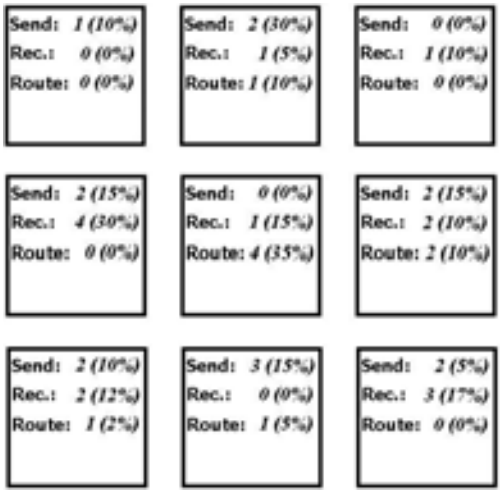


Figura 3.13. Parâmetros de Comunicação da Simulação

Com estes parâmetros de simulação, um gerador de pacotes foi utilizado para fornecer os arquivos de entrada. O gerador foi utilizado para gerar cargas de 5%, 10%, 15%, 20% e 25% de comunicação sobre 10.000 instruções. As mesmas cargas foram utilizadas para gerar pacotes de tamanho 2, 4, e 8, totalizando 15 arquivos de entrada. A carga utilizada representa 18kBytes por 5% de comunicação por 1 palavra de pacote. No caso mais extremo, 25% de comunicação com pacotes de tamanho 8, isto representa a transmissão de 720kBytes.

Os mesmos arquivos de entrada foram utilizados na simulação de duas versões da NoCX4, uma com buferização FIFO, utilizando um buffer de 32 posições por porta de entrada e a outra com buferização SAFC, utilizando quatro buffers de 8 posições por porta de entrada, e de um sistema de barramentos, apresentado em [SOA 04]. Resultados de trabalhos anteriores [SOA 03-1] demonstraram que o tamanho dos buffers tem pouca influência no desempenho do sistema. Os resultados de *throughput* (vazão de pacotes pela rede, em bytes/ciclo), considerando-se apenas o *payload* dos pacotes, podem ser vistos na **Figura 3.14**; a duração das simulações, em ciclos, é mostrada na **Figura 3.15**; e a na **Figura 3.16** está a latência média.

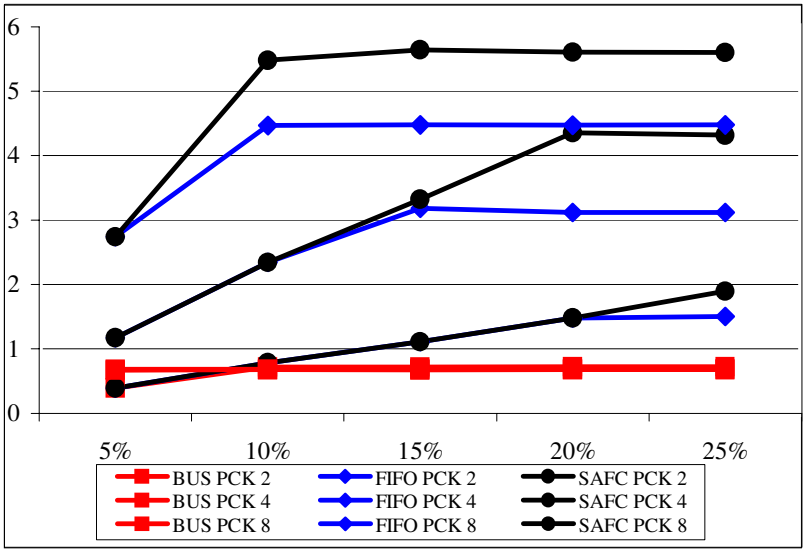


Figura 3.14. Throughput (bytes/ciclo)

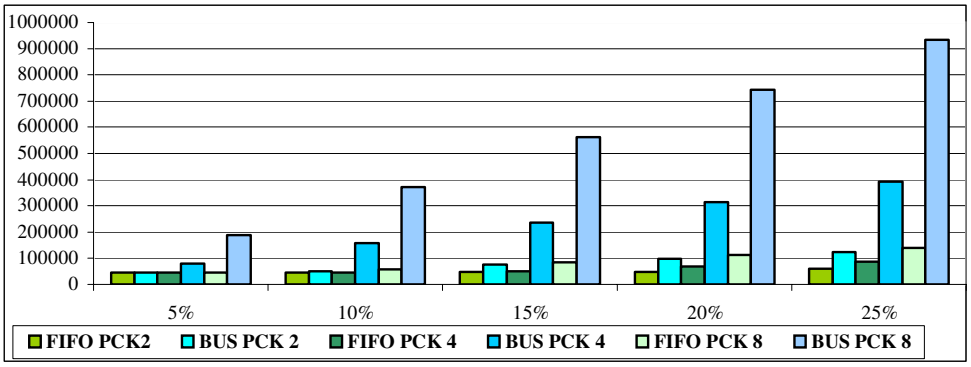


Figura 3.15. Duração da simulação (ciclos)

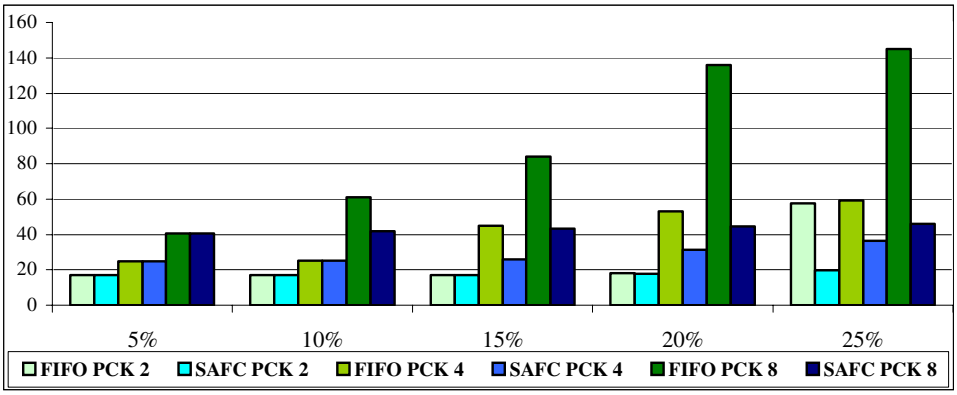


Figura 3.16. Latência Média (ciclos)

Pela análise dos resultados pode-se ver a incontestável superioridade da NoC sobre o sistema de barramentos. A comunicação pipeline em paralelo provida pela NoC é suficiente para dar vazão aos pacotes gerados, até determinado limite. Como pode ser comprovado pelo throughput, o *throughput* do sistema de barramentos chega ao seu limite mesmo com as menores cargas. Enquanto isso, a NoC mantém um crescimento no throughput até um limite, que varia de acordo com a buferização utilizada: menor para FIFO, maior para SAFC. Este fato é claramente representado para o caso SAFC PCK 4 em 20% de comunicação. Seu throughput se assemelha àquele do caso FIFO PCK 8, que possui pacotes com o dobro do tamanho. Há um crescimento linear neste caso, chegando ao limite apenas em 25% de carga de comunicação. Os resultados de latência média apenas corroboram este fato, sendo que para todos os casos a latência da NoC usando SAFC se mantém praticamente inalterada, enquanto que com o uso de FIFO há um crescimento bastante acentuado.

3.7.2. Árvore Obesa

A maioria das pesquisas envolvendo NoC baseiam-se em topologias dimensionais, como Grelha, Cubo, Hipercubo e suas variações. Estas topologias são regulares, o que torna o roteamento mais simples, e seus roteadores podem ser replicados pelo sistema. Porém, as topologias dimensionais possuem desvantagens como grande diâmetro [HWA 87], o que acarreta em alta latência na transmissão de pacotes, limitando assim a taxa de transferência. Por serem topologias diretas, cada Elemento de Processamento (*Processing Element* - PE) do sistema possui um roteador correspondente. O alto número de roteadores no sistema não apenas ocupa uma área maior no chip, mas também pode vir a ser um fator limitante para a escalabilidade do mesmo.

A *Fat Tree* (Árvore Gorda) surgiu como uma solução para os problemas encontrados nas topologias diretas. Esta topologia consiste em uma árvore que possui diversos nós raiz. Cada roteador possui um roteador pai correspondente. Ao contrário das topologias dimensionais, que são cíclicas, as *fat-trees* são livres de *deadlock* [CHO 01]. Em [LEI 85] encontra-se uma prova formal de que a *fat-tree* é a topologia com melhor relação custo/benefício para VLSI. As redes *fat-tree* necessitam de um número de roteadores menor do que linear, mas a replicação de roteadores pais e interconexões redundantes, necessárias para alcançar maior *throughput*, podem ser muito custosas.

Para tentar superar os problemas encontrados nas topologias *fat-tree* tradicionais, a topologia *Obese Tree* (Árvore Obesa) foi desenvolvida no decorrer deste trabalho. Ela é uma adaptação da topologia *fat-tree*, porém cada roteador possui apenas um pai e está conectado a três outros roteadores "irmãos". Estas adaptações baseiam-se em três premissas: que os PEs e

processos que se comunicam são alocados o mais próximo possível, ; que uma comunicação de baixa latência significa um melhor desempenho do sistema como um todo; e que a NoC deve ter o menor impacto possível na escalabilidade do sistema, deixando mais área do chip para os elementos de processamento. A ObTree provê a menor distância topológica dentre as topologias avaliadas, e a menor área estimada.

3.7.2.1. Topologia ObTree

A Árvore Obesa possui topologia indireta e consiste de dois diferentes tipos de roteador. Como em qualquer árvore, os roteadores são alocados em níveis. O nível mais baixo é composto por Roteadores Folha (*Leaf Router*) e os demais níveis são compostos por Roteadores Intermediários (*Intermediary Router*). Os Roteadores Folha são conectados aos elementos de processamento enquanto que os Roteadores Intermediários são conectados a outros roteadores. A topologia ObTree é mostrada na **Figura 3.17**.

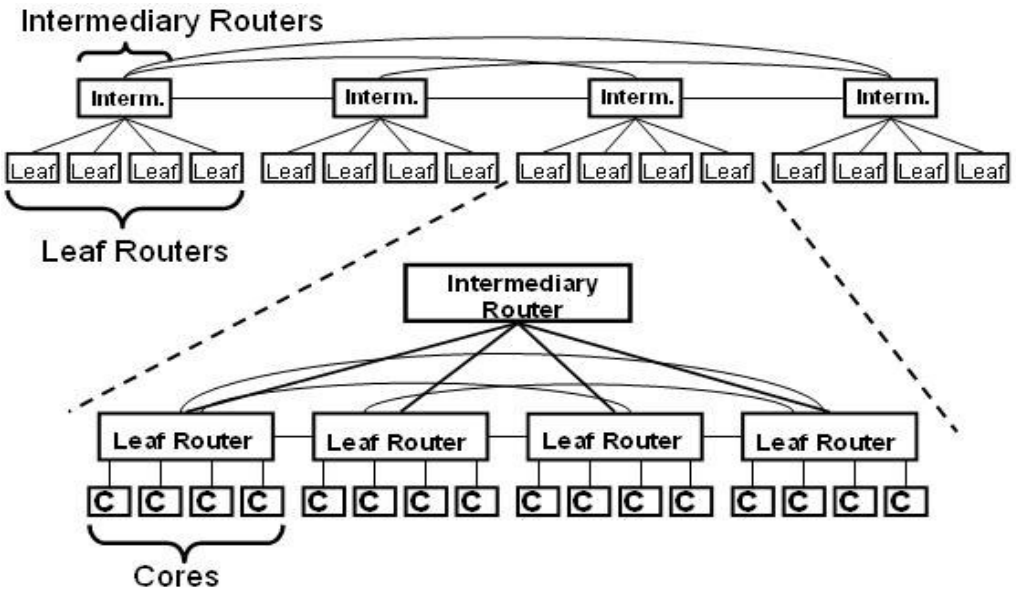


Figura 3.17. Topologia Árvore Obesa

Ao contrário de outras topologias em árvore, na Árvore Obesa todos os roteadores filhos de um mesmo pai são completamente interconectados. Com isto o tráfego entre nós de nível mais baixo fica restrito aos roteadores filhos, sem a criação de *bottleneck* (gargalo) no roteador pai. Com a eliminação deste gargalo diminui a necessidade de um pai por roteador, a base da *fat tree*, e a criação de conexões redundantes entre roteadores. Desta forma o número de roteadores no sistema é mínimo. O uso de conexões laterais entre os roteadores irmãos diminui a distância topológica média entre dois PEs, já que o pacote não necessita ser transmitido para o roteador

pai e de volta para o roteador filho: a transferência é feita diretamente entre os roteadores irmãos. Tanto a baixa latência nas transferências quanto o menor número de roteadores contribuem para o desempenho e a escalabilidade da Árvore Obesa.

3.7.2.2. Arquitetura do Roteador

Na Árvore Obesa os PEs são conectados aos Roteadores Folha e a comunicação em todos os outros níveis é feita por Roteadores Intermediários. A diferença entre eles é somente que nos Roteadores Folha as portas para o nível mais baixo (portas conectadas aos PEs) são totalmente conectadas, para que um pacote transferido entre dois PEs sob o mesmo Roteador Folha só passam por um roteador. Nos Roteadores Intermediários esta conexão não é necessária. Os pacotes nunca são transferidos entre dois roteadores irmãos através do roteador pai, eles usam suas conexões laterais. Esta diferença está ilustrada na **Figura 3.18**.

Os roteadores são conectados entre si por dois canais unidirecionais. Eles usam bufferização na entrada por FIFOs, chaveamento VCT (*Virtual Cut-Through*), arbitragem distribuída FCFS (*First-Come, First-Served*), controle de fluxo baseado em crédito e roteamento de menor caminho. Este roteamento é o mesmo utilizado nas *fat trees* tradicionais [LEI 92], exceto pela utilização das conexões laterais.

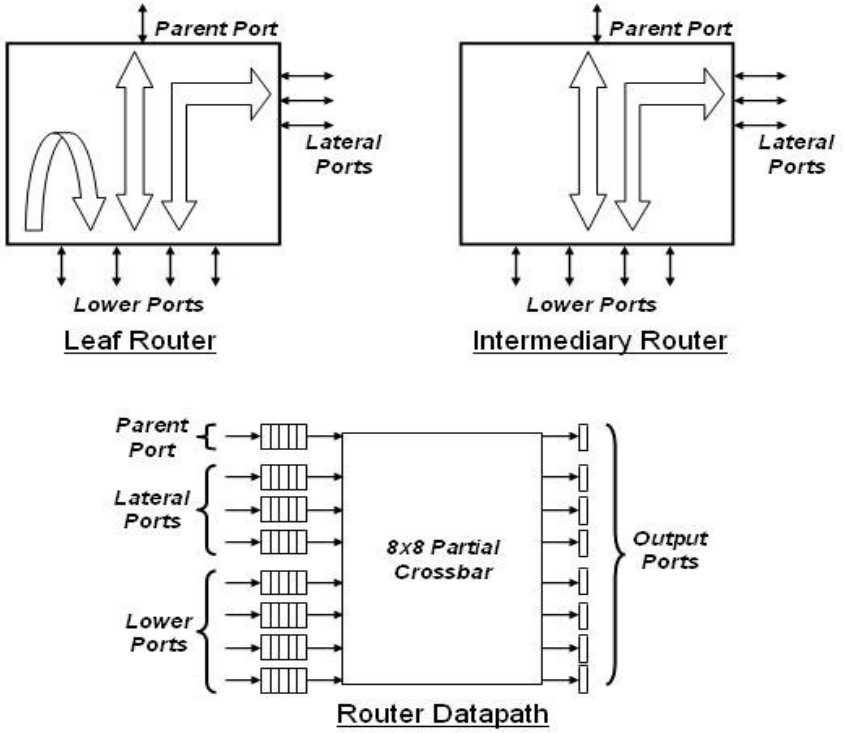


Figura 3.18. Roteador Folha e Roteador Intermediário

3.7.2.3. Comparação Teórica

A topologia ObTree provê comunicação de baixa latência por possuir uma quantidade mínima de roteadores no sistema. Uma comparação entre a Árvore Obesa e outras topologias pode ser vista na **Tabela 3.6**. Os resultados da tabela consideram um sistema de 256 PEs.

Tabela 3.6. Comparação entre Topologias

	Grelha	Cubo	Hiper Cubo	Árvore Gorda	Árvore Obesa
Diâmetro	31	18	13	7	6
# de Roteadores	256	256	256	256	84
Distância Top. Média	11,66	7,52	6,01	6,38	5,37
Portas por Roteador	5	7	9	8	8
Estimativa de Área Ocupada	1.280	1.792	2.304	2.048	672

Na tabela, o campo "Diâmetro" representa a maior distância entre dois PEs. O campo "# de Roteadores" mostra quantos roteadores são necessários para a integração dos 256 PEs, e o campo "Portas por Roteador" contém o número de portas de cada roteador. Este último pode ser visto como um parâmetro da complexidade dos roteadores de cada topologia, já que ele indica o número de *buffers*, árbitros, mecanismos de controle de fluxo, etc. por roteador, e a complexidade de seu *crossbar*. Quanto maior a complexidade dos roteadores, maior a área ocupada em chip, e possivelmente uma menor freqüência de operação. Normalmente os roteadores de topologias regulares (Grelha, Cubo e Hipercubo) empregam *crossbar* completos, não permitindo apenas a conexão de uma porta consigo mesmo. O *crossbar* das Árvores Gordas costuma ser parcialmente completo, não interconectando as quatro portas superiores. O *crossbar* da Árvore Obesa possui baixo grau de interconexão, como foi mostrado na seção anterior. Apesar de não haverem resultados de ocupação de área em uma implementação VLSI da Árvore Obesa, ela pode ser grosseiramente estimada pela multiplicação do número total de roteadores pelo número de portas de cada roteador, como no campo "Estimativa de Área Ocupada" da tabela. Pela comparação dos resultados nota-se claramente o menor impacto da Árvore Obesa no número de transistores usados no sistema.

O campo "Distância Topológica Média" é uma média de todas as distâncias topológicas na rede, representado por quantos roteadores, em média, um pacote deve passar em uma comunicação, usando para isso o menor caminho.

A superioridade da Árvore Obesa sobre as outras topologias é clara na tabela. Ela possui o menor diâmetro e a menor distância entre dois PEs, com a menor quantidade de roteadores, ocupando a menor área em chip. Isto produz uma comunicação de baixa latência, algo crítico em alguns SoCs, especialmente nos MP-SoCs, onde os processadores constantemente acessam dados remotos, e em sistemas de tempo-real, que precisam de respostas rápidas.

3.8. Boot do Sistema

O boot de STORM possui dois níveis: o de hardware e o de software. O nível de hardware precisa reconhecer o tamanho da rede, os componentes a ela conectados e suas características, e distribuir estas informações para onde for necessário. O nível de software inicializa o SO, carregando seu código binário de um dispositivo externo e montando tabelas com informações sobre os processos, canais de entrada/saída, disposição física dos componentes, etc. O nível de software necessita da conclusão do nível de hardware para começar a operar.

3.8.1. Nível de Hardware

O boot em nível de hardware tem por objetivo reconhecer o sistema, montar as tabelas ATA e PTA com estas informações e distribuí-las entre os seus componentes. Por isso, este nível é dividido efetivamente em duas fases: *reconhecimento do sistema* e *distribuição de tabelas*. A inicialização do dispositivo de E/S não está incluída pois o sistema de E/S da plataforma ainda não está devidamente especificado.

Como a princípio as características do sistema são desconhecidas, definiu-se que a posição 0,0 da rede (superior esquerda) deve ser ocupada por um processador, que está ligado a uma cache diferenciada. O seu CaCoMa possui todas as tabelas, e uma rotina em hardware para implementar o boot físico. É este o componente ativo durante todo o processo.

O CaCoMa diferenciado, ou CaCoMaB, inicia a rotina de boot coletando informações de todos os componentes na rede. Como a NoC não possui mecanismo de broadcasting, isto é feito através de um *loop*. A cada iteração no *loop*, O CaCoMaB envia um pacote para um componente e espera sua resposta. Este pacote possui uma instrução de identificação na forma de um bit ativado em seu cabeçalho. Se houver um componente para receber o pacote, este responderá com o seu tipo (Processador ou Memória) e informações extras, como o tamanho da memória e o tipo do processador. Os pacotes são enviados individualmente, percorrendo as linhas. Enquanto identifica a primeira linha, a cada pacote enviado o CaCoMaB ativa um contador. Quando este contador atinge zero sem que um pacote de resposta chegue, isto significa que a linha acaba ali. A fórmula utilizada para calcular a contagem leva em

consideração o tempo fixo de envio e resposta, assim como a distância entre as roteadores. Por não haverem outros pacotes trafegando na NoC, o tempo de resposta é determinístico, dado pela **Equação 3.1**

Equação 3.1 Tempo de resposta

$$Tr = Di + Dr * i$$

Tr: Tempo de resposta
Di: *Delay* inicial mínimo em todas as transmissões
Dr: *Delay* extra a cada roteador
i: Distância topologia entre o CaCoMaB e o destinatário do pacote

Como a rede deve ser uma matriz regular, por causa do roteamento dimensional, então o tamanho de todas as linhas é o mesmo, não precisando ser medido novamente. O mesmo procedimento e raciocínio valem para o tamanho da coluna, assim como a fórmula para calcular a contagem regressiva. Para que o pacote sem destinatário não fique eternamente armazenado, o roteador da NoC o descarta. À medida que vai recebendo as respostas, o CaCoMaB monta as tabelas ATA com o endereço dos módulos de memória e faixa de endereçamento de cada um e as tabelas PTA com o endereço dos processadores.

Uma vez que toda a rede for reconhecida e as tabelas estiverem preenchidas, o CaCoMaB então começa a enviar as tabelas para os módulos apropriados, iniciando assim a segunda fase do boot físico. Da mesma forma que na fase de identificação do sistema, a fase de distribuição de tabelas consiste em um *loop*. A cada iteração no *loop*, o CaCoMaB calcula um endereço de NoC (percorrendo as linhas), identifica o componente ali localizado e envia a tabela apropriada para aquele componente (ATA para processadores, PTA para memórias). A PTA é enviada de forma única, por só conter endereços de NoC. Cada endereço de NoC é composto por exatamente 1 byte, sendo 4 bits para indicar a linha e 4 para indicar a coluna, enquanto que os pacotes da NoC possuem largura de 32 bits. Para otimizar o tempo de transmissão das tabelas, cada palavra do pacote contém 4 endereços da tabela. Em um sistema com dezenas de processadores pode ser necessário mais de um pacote para transmitir toda a tabela. Em todo o caso, no primeiro pacote são enviadas informações como o número de linhas, número de colunas, número de processadores e número de módulos de memória do sistema.

A transmissão da ATA se dá de forma semelhante à PTA, porém em duas partes. Na primeira são enviados os endereços e informações do sistema, de maneira análoga à ATA. Na segunda parte, são enviados os endereços finais de cada módulo de memória. Como STORM possui suporte a um espaço de endereçamento de 32 bits, este campo consome uma palavra por módulo de memória. Somente quando todo este processo é feito, o CaCoMaB envia um sinal para seu processador sair do estado de *reset* e começar o *boot* software do sistema. Cada CaCoMa envia este mesmo sinal para seu processador respectivo ao receber a ATA.

3.8.2. Nível de Software

Nesse momento a plataforma de hardware está devidamente configurada, com o espaço de endereçamento da memória e a quantidade de processadores definidos, bem como os dispositivos de E/S reconhecidos. A plataforma está pronta para ser utilizada, mas antes que isso seja possível é necessário inicializar o Sistema Operacional. O Sistema Operacional, como qualquer software, é constituído de um conjunto de arquivos que precisam ser carregados em memória para serem executados. Esse processo de carga é realizado logo que termina o *boot* de hardware, uma vez que nesse estado já se tem definido o local a partir do qual a carga dos arquivos deve ser realizada.

O procedimento de carga fica armazenado em uma memória não-volátil na plataforma e carrega os arquivos a partir de um ponto determinado de um dispositivo de armazenamento. A ordem e a quantidade de arquivos a serem carregados dependem do Sistema Operacional, mas basicamente deve ser feita uma carga dos *drivers* dos dispositivos de E/S, bem como a preparação do sistema para que possam ser realizados escalonamentos de processos e tratamento de interrupções, tanto de processos como de E/S. Essa preparação consiste em alocar espaço de memória para tabelas de processos, de interrupção e de E/S. A tabela de processos auxilia o escalonador no revezamento pela utilização dos processadores, guardando o contexto dos processos em estado de espera ou pronto. A tabela de interrupções guarda o contexto de processos que sofreram algum tipo de interrupção até que essa interrupção seja devidamente tratada. A tabela de E/S guarda o contexto de processos que solicitaram uma operação de E/S e que não podem prosseguir até que essa operação seja completada. Uma vez carregados os *drivers* dos dispositivos e criadas as tabelas o escalonador pode começar a ser executado e o sistema poderá ser efetivamente utilizado.

3.8.3. Resultados de Simulação

O boot de hardware foi simulado em diversas instâncias da plataforma, sendo todas elas matrizes quadradas. Como a proporção de módulos de memória e processadores pode influenciar o tempo de boot, devido às diferentes tabelas enviadas para cada um destes, definiu-se que o último módulo de cada linha seria uma memória, e todos os outros, processadores. Uma ilustração das instâncias pode ser vista na **Figura 3.19**.

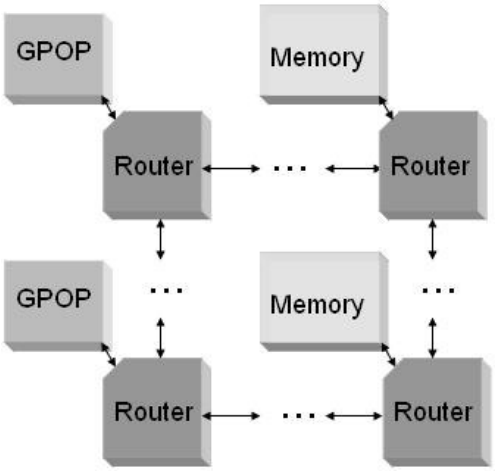


Figura 3.19. Instâncias de Boot

As instâncias começam de uma matriz 2x2 de componentes, até 16x16, o máximo suportado pelo sistema. O tempo começa a ser medido de um *reset* no sistema e termina quando o último módulo de memória recebe a sua tabela PTA. A **Figura 3.20** mostra o número de ciclos necessários para cada simulação.

Pode-se ver, no gráfico, um crescimento acentuado a cada simulação. Isto se dá porque em uma matriz quadrada o número de módulos não aumenta linearmente. Outra razão para que isto aconteça é que a distância entre o CaCoMaB e os módulos conectados à outra extremidade da rede cresce a cada simulação, portanto os módulos inseridos são cada vez mais numerosos e mais distantes do CaCoMaB.

A metodologia utilizada neste trabalho não provê ainda informações sobre o período de relógio. Apesar disto, mesmo supondo uma freqüência teórica de 50 MHz para o sistema, o tempo de inicialização física para uma versão com 256 módulos seria de 1,33 ms.

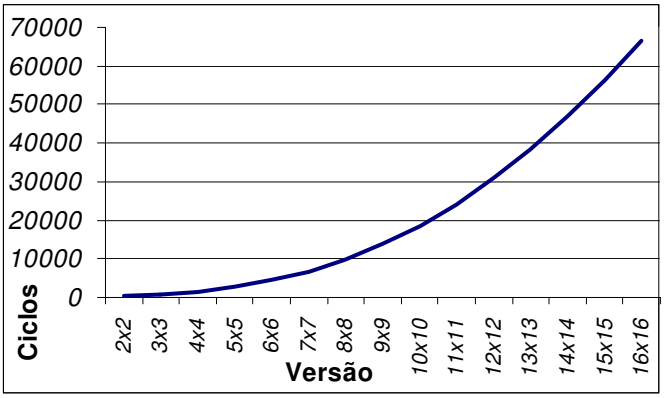


Figura 3.20. Resultados de Boot

3.9. E/S

A Entrada/Saída é um aspecto delicado em sistemas de processamento paralelo. Pelo fato de diversos processos estarem sendo executados ao mesmo tempo, há uma grande chance de que mais de um processo queira acessar o mesmo dispositivo de E/S simultaneamente. A gerência destes acessos exige suporte não só de hardware como do sistema operacional.

Nos MP-SoCs o aspecto de E/S é ainda mais delicado. Os sistemas multicomputador normalmente possuem dispositivos de E/S exclusivos para cada processador, o que elimina o problema do acesso concorrente. Os sistemas multiprocessador são normalmente de tamanho reduzido, o que limita a concorrência de acessos, e os maiores podem possuir mais de um mecanismo de E/S. Porém, os MP-SoCs estão restritos a um único chip, que possuem severa limitação nos pinos que fazem E/S. Isso gera uma concorrência por um recurso, que aumenta de acordo com a magnitude do sistema. MP-SoCs com milhares de processadores vão precisar de alternativas especiais para gerenciar o gargalo pelos pinos do chip, sob pena de um severo entrave ao seu desempenho.

O problema da E/S não pôde ser abordado no decorrer deste trabalho por falta de tempo hábil e das prioridades estabelecidas para o projeto. Pode ser considerado um trabalho futuro imediato do grupo de pesquisa.

4. Metodologia

4.1. SystemC

O projeto de STORM incluiu a criação, especificação, implementação, simulação e validação de uma plataforma MP-SoC totalmente funcional sobre uma NoC, provendo todos os recursos necessários para execução de tarefas e um sistema operacional. Para a implementação da plataforma, o uso de uma linguagem puramente de descrição de hardware (HDL) foi descartado por diversas razões. Em primeiro lugar, o baixo nível exigido para a implementação em tal descrição é desnecessário para os propósitos do projeto, que não visa o detalhamento da arquitetura, mas sim sua funcionalidade. Em segundo lugar, o tempo de simulação cresce consideravelmente com o detalhamento arquitetural da implementação. E, por fim, a dificuldade em se explorar espaço de projeto com uma implementação RTL. Por estas razões, a "linguagem" SystemC [OSI 05], que permite uma descrição funcional do sistema, foi escolhida por melhor se adequar às necessidades do projeto.

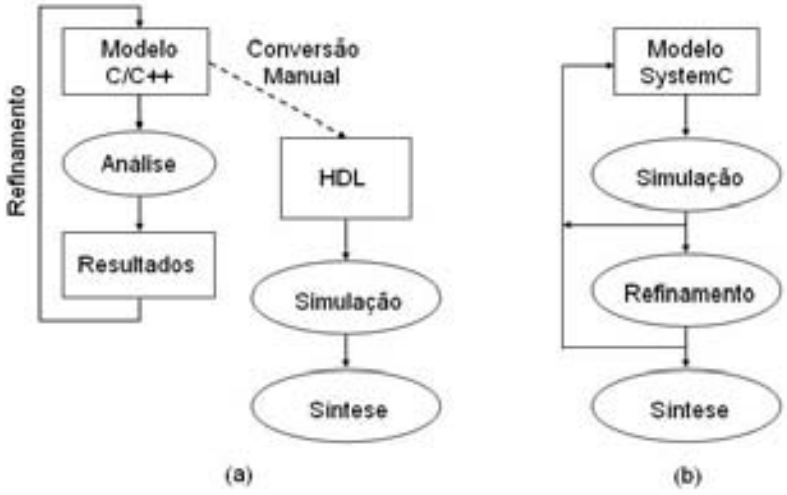


Figura 4.1. Duas metodologias de projeto (a) Metodologia tradicional; (b) Metodologia SystemC

SystemC não é propriamente uma linguagem, mas sim uma biblioteca de *templates* e funções para C++. Esta biblioteca tem como objetivo possibilitar ao programador a construção de blocos (módulos) de hardware, e simulá-los obedecendo a um relógio (*clock*). O uso de SystemC possibilita a implementação em diversos níveis de abstração, começando de uma aplicação escrita puramente em C++, sendo refinada até o nível RTL. Esta possibilidade é uma

quebra no paradigma tradicional de desenvolvimento de hardware, onde faz-se necessário realizar um salto da implementação em linguagem de programação para a descrição RTL, como mostrado na **Figura 4.1**.

O resultado da implementação em SystemC é uma *especificação executável*. Existem várias benefícios em seu uso, como evitar inconsistências, erros e interpretação ambígua da especificação. Com a especificação executável é possível validar a funcionalidade do sistema e verificar seu desempenho antes de sua implementação física. A verificação também se torna bem mais simples, pois os mesmo casos de testes utilizados desde a implementação em linguagem de alto nível até os níveis de abstração mais baixos.

A maior desvantagem do uso de SystemC era a falta de resultados de síntese, como área ocupada, freqüência máxima de relógio e consumo de energia. Porém, já existem ferramentas que fazem a síntese de uma especificação SystemC para silício [FOR 05]. Apesar de não estar dentro do escopo deste projeto, a obtenção destes resultados pode ser feita com a mesma implementação utilizada para simulação, não mais sendo imprescindível uma implementação em HDL para tal.

4.2. Implementação

4.2.1. STORM

Todos os componentes previstos no projeto de STORM e descritos no **Capítulo 3** foram implementados usando a ferramenta SystemC. Como esta ferramenta é na verdade um conjunto de bibliotecas de *templates* em C++ e não uma linguagem específica, é possível ter a implementação de um mesmo módulo em diversos níveis de abstração e de precisão. Para este projeto foi definido que a implementação deveria produzir resultados com precisão de ciclo (*cycle-accurate*), por serem mais expressivos e melhor aceitos entre a comunidade acadêmica. Isto não significa, porém, que todas as portas e sinais da microarquitetura dos módulos foram utilizados na implementação. Muito pelo contrário, a descrição dos componentes é sempre feita de maneira mais alto-nível possível, desde que isto não comprometa o nível de precisão desejado. O elevado tempo de simulação que uma descrição mais precisa acarreta pode tornar proibitivos os objetivos desta implementação, que é justamente o de simulação real de aplicações complexas na plataforma.

A implementação consiste em um arquivo principal (*main*) que instancia todos os componentes da plataforma e os interconecta. Cada componente é um módulo do tipo SC_MODULE descrito em dois arquivos: um com extensão ‘.h’ e outro ‘.cpp’, este último contendo a funcionalidade do módulo. Todos os módulos são reentrantes, podendo ser

instanciados diversas vezes em uma mesma plataforma. Os detalhes da implementação (arquivos, portas, funções, etc.) são mostrados no **Anexo III**.

Por não haver um SO para gerenciar a memória, as decisões relativas à alocação foram implementadas diretamente no código da plataforma e do montador. A divisão da memória e suas regiões estão ilustradas na **Figura 4.2**. Todos os valores estão expressados em termos de palavras de 32 bits.

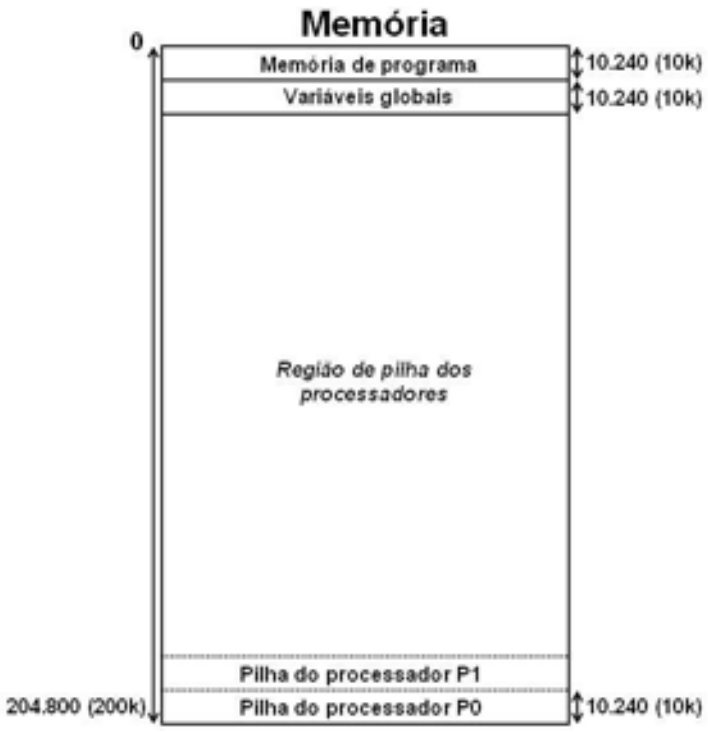


Figura 4.2. Regiões de Memória

É importante notar que todos os valores utilizados para delimitação das regiões de memória na figura são meros parâmetros, que podem ser facilmente modificados a qualquer momento. Estes são os valores que foram utilizados nas simulações apresentadas no **Capítulo 5**.

4.2.2. Montador

Além da implementação em SystemC de STORM, para este trabalho também foi implementado um montador (*assembler*) SPARC. Ele foi projetado com o intuito de fazer a conexão entre o compilador gcc e a plataforma. Esta conexão permitiria a implementação de aplicações em C ao invés de *assembly*, um requisito essencial para a simulação de aplicações de maior porte.

O montador SPARC reconhece o código *assembly* gerado pelo gcc (utilizando-se a opção -S do compilador). Sua saída é um arquivo binário que pode ser imediatamente utilizado como entrada para a plataforma. Por não haver nenhum tipo de SO para gerenciar a execução de STORM, o montador implementado possui algumas particularidades que auxiliam neste processo.

A primeira delas é um mecanismo para contornar a falta de um *loader*. O arquivo *assembly* gerado pelo compilador nem sempre começa pela função *main()*. Pelo contrário, todas as funções incluídas pela diretiva *#include* aparecem em primeiro lugar. Para que o programa comece sua execução pela função *main()* o montador inclui duas instruções virtuais no início do programa binário. São elas:

```
b    main   ! Salta para o label main, sempre associado à função main() do código C
nop         ! NOP obrigatório após uma instrução de salto
```

Assim, não importa a localização da função *main* no código *assembly*, a execução do programa sempre começará por ela, sem a necessidade de um programa *loader*.

Outro importante recurso implementado, especialmente para aplicações paralelas, é a alocação de variáveis globais. Por não haver um gerenciamento da memória sobre os dados de usuário, definiu-se uma região de memória para alocação das variáveis em tempo de compilação. Porém, ao invés de serem alocadas em posições consecutivas, as variáveis são alocadas em blocos de memória consecutivos. Como a unidade de transferência de dados entre memória e cache é o bloco, a alocação de mais de uma variável global em um mesmo bloco poderia causar problemas, já que elas são o único mecanismo de comunicação entre processos em um ambiente de memória compartilhada. Um processo que acesse determinada variável global, possivelmente dentro de uma região crítica, receberia também outras variáveis naquele mesmo bloco, que possivelmente fazem parte de outra região crítica e necessitam de um mecanismo de exclusão mútua para o acesso. Isto arruinaria o mecanismo de *mutexes* utilizado, que não poderia garantir o acesso exclusivo de um único processador a uma variável global.

O último mecanismo especial implementado no montador diz respeito diretamente ao *mutex*. Por serem na verdade simples variáveis globais, a alocação deles deveria ser junto de outras variáveis globais. Porém, os *mutexes* devem ser acessados exclusivamente pelas instruções atômicas *LDSTUB* e *SWAP*, nunca por instruções comuns de *load* e *store*. Para garantir uma proteção destas variáveis especiais contra o acesso indevido, criou-se uma região especial para alocação dos *mutexes*. Como são, porém, simples variáveis globais perante o compilador, a maneira encontrada de diferenciação dos *mutexes* de outras variáveis globais foi pelo seu nome. Toda a variável global que possuir "*mutex*" em seu nome é identificada pelo montador e alocada em uma região exclusiva de memória. Como o acesso aos *mutexes* é sempre

feito por instruções atômicas, nunca havendo transferência de blocos entre memória e cache, a alocação deles pode ser feita em posições contíguas.

4.2.3. Biblioteca Mutex.h

A linguagem C não possui suporte nativo à programação paralela. O uso de *threads* é possível em sistemas reais por causa da biblioteca POSIX Thread (*pthread*), mas para tal é necessário o suporte de um sistema operacional. Como não existe este tipo de suporte em STORM, foi necessária a criação de uma biblioteca própria com funções para suporte à programação concorrente, a *mutex.h*. Esta biblioteca possui 3 funções simples e uma variável global. Seu código-fonte pode ser visto no **Anexo II.5**.

As duas primeiras funções implementam um mecanismo simples de requisição de liberação de *mutex*, com as funções *down()* e *up()*, respectivamente. Ambas as funções recebem um ponteiro de inteiro, o ponteiro da variável que será utilizada como *mutex*, e retornam um valor inteiro, o valor do *mutex*.

A função *down()* (aquisição do *mutex*) consiste em um *loop* de espera. O valor de retorno é inicializado com "-1" (*mutex* ocupado) e é feito um *load* atômico da variável *mutex* passada por parâmetro utilizando a instrução *LDSTUB* do SPARC. Isto só é possível graças a um recurso do gcc, que permite a utilização de instruções específicas de um processador dentro do código C. Este recurso, representado pelo comando *asm*, necessita que o usuário explicite a instrução a ser utilizada e seus parâmetros. No caso do *LDSTUB*, foi definido que existem dois parâmetros, sendo o segundo a saída da instrução (na linha '**asm ("ldstub %1,%[output]"**'). O primeiro parâmetro é do tipo posição de memória, e ela deve ser dada pelo ponteiro recebido como parâmetro da função (na linha '**: "m" (*s)**'). O segundo parâmetro, a saída da instrução, é do tipo registrador, e deve ser guardado na variável *retorno* (na linha '**: [output] "=r" (retorno)**'). A função permanece em *loop* até que o valor da variável retorno, dado pela instrução *LDSTUB*, seja igual a "0" (*mutex* livre). Quando isto acontece, o Diretório se encarrega de posicionar o valor daquele *mutex* como "-1" (*mutex* ocupado) para garantir a atomicidade. Ao final a função retorna o valor da variável *retorno*.

A função *up()* (liberação do *mutex*) é implementada de forma semelhante, porém sem a necessidade de *loop*. Para a liberação do *mutex* foi utilizada a instrução *SWAP* (troca conteúdo de registrador com conteúdo de posição de memória atomicamente). Como o valor de mutex livre é "0", o valor de *retorno*, a variável que fará a troca, é inicializado como "0". Então a instrução *SWAP* é chamada, utilizando-se o mesmo recurso da função *down()*, com apenas uma diferença. Como o valor a ser escrito na memória com a troca está na variável retorno, e ela está em um registrador, é necessário que o conteúdo daquele registrador seja mantido intacto até o

final da execução de *SWAP*. O gcc permite esta manobra, apenas mudando-se o "**=r**", que indica que a saída daquela instrução deve ser escrita na variável indicada, para "**+r**", que indica a mesma coisa, mas impõe a restrição de que o valor do registrador de retorno deve ser preservado até o fim da execução da instrução.

Tendo sido implementado um mecanismo eficiente de exclusão mútua, é necessário que haja um mecanismo para divisão de tarefas (alocação de processos) embutida no código C da aplicação. Este mecanismo se dá de uma forma bastante simples, simplesmente atribuindo um identificador único para cada processador e distribuindo a aplicação de acordo com este identificador (*ID*). A aquisição de um *ID* único se dá da seguinte maneira: a biblioteca possui uma variável específica *current_id*, que começa em zero (para o bom funcionamento deste mecanismo é necessário que esta variável inicie com o valor "0" antes do início da execução dos processadores). Para seu acesso ideal foi implementada a função *get_id()*, que retorna o seu valor atual e subsequentemente o incrementa em 1. O acesso ideal deste mecanismo deve ser feito da seguinte forma:

```
down(&mutex);                  // Adquire permissão para acessar a variável global/
unsigned int id = get_id();    // Recebe o ID único e o copia para uma variável local
up(&mutex);                    // Libera o mutex
```

Após todos os processadores terem executados este trecho de código, idealmente posicionado logo no início do programa, todos terão um valor único em suas variáveis locais *id*. A partir daí, pode-se fazer a paralelização do código baseando-se neste *id*.

Esta metodologia é posta à prova e validado nas simulações de aplicações concorrentes apresentadas no capítulo seguinte.

5. Resultados Experimentais

Após a apresentação dos módulos que compõem STORM e de sua utilização, feitas no **Capítulo 3** e **Capítulo 4**, respectivamente, este capítulo apresenta as simulações realizadas para validação da plataforma e da metodologia. Por ter caráter de validação, e não de avaliação de desempenho, as simulações são apresentadas em ordem crescente de complexidade. As aplicações simuladas representam as fases pelas quais passou o projeto até chegar à fase atual, dando assim uma clara idéia de como foi o processo de implementação do SPARC, implementação do montador, refinamento e integração de ambos, integração com o código *assembly* gerado pelo gcc, e por fim a implementação de mecanismos de alto nível para programação concorrente.

Um total de 10 aplicações são apresentadas neste capítulo, algumas possuindo mais de uma simulação. Elas estão divididas em pequenos relatórios que seguem um mesmo *template*. Este *template* é apresentado em detalhes a seguir.

O propósito destas simulações é tão somente de validação da plataforma e da metodologia utilizada em sua programação. As aplicações variam em complexidade e utilização, indo desde um simples cálculo do fatorial de um número (**Simulação 1**) até a implementação paralela da Transformada Discreta do Cosseno (DCT) 2D (**Simulação 10**). Todas, porém, estão dentro da proposta deste capítulo e se prestam à validação de algum aspecto da plataforma.

A lista de aplicações começa com a mais simples, a implementação em *assembly* do cálculo do fatorial, utilizada para validação das questões mais rudimentares do projeto, como busca de instruções, integração do SPARC com a plataforma, execução de saltos e gramática do montador. Estes aspectos são novamente verificados na aplicação seguinte, a identificação do maior valor em um vetor. Nesta aplicação, porém, são adicionados aspectos como acesso à memória e vetores. Em seguida, a mesma aplicação é implementada em C para a verificação do caminho do código C até a execução na plataforma e o tratamento de construções particulares utilizadas pelo gcc. Logo após é mostrada a implementação do algoritmo de ordenação *Quicksort*, um código simples, mas que exige grande suporte por parte do montador e da plataforma, e exigiu uma grande quantidade de ciclos para sua execução. Temos então o teste da biblioteca desenvolvida em C para prover suporte à programação concorrente, que faz uso de duas instruções específicas do *assembly* SPARC. Então a primeira aplicação paralela é simulada, a multiplicação de matrizes. Por sua implementação ser propositalmente de fácil paralelização, ela foi utilizada para verificação do ganho de desempenho com o paralelismo, e para comparação entre os dois modelos de NoC de STORM. A seguir tem-se uma seqüência de 3 aplicações de classes diversas, uma de ordenação (*Mergesort*) uma de busca de padrão (*KMP*) e a terceira uma aplicação utilizada em multimídia (Estimação de Movimento). Todas possuem

implementação paralela. Sua execução concorrente serve para validação do mecanismo de *mutex* implementado na biblioteca de suporte à programação paralela, *mutex.h* em situações diversas. Por fim, a última aplicação simulada é a Transformada Discreta do Cosseno (DCT) 2D. Sua paralelização foi feita de uma maneira diferente das outras aplicações paralelas, pois ao invés de se ter somente processos que executam em paralelo e precisam do auxílio de *mutexes* para proteção das variáveis compartilhadas, a DCT necessita que os processos sejam executados em determinada ordem, formando um *pipeline* de execução. Mais uma vez, a biblioteca *mutex.h* provê suporte suficiente para esta manobra. Todas as aplicações foram compiladas sem o uso de nenhum tipo de otimização.

Após a apresentação das simulações este texto chega a seu término com as conclusões, discutidas no **Capítulo 6**.

Simulação XX: NOME

Programa: fonte.c

Código-Fonte: Anexo onde pode ser encontrado o código fonte.

Descrição: Breve descrição da aplicação e do algoritmo. Descrição de seu funcionamento e detalhes importantes de sua implementação.

Propósito: O propósito da simulação daquela aplicação em particular. Explicação dos objetivos e resultados esperados, assim como sua importância para a validação da plataforma.

Paralelização: Este campo está presente quando a aplicação envolver o uso de mais de um processador. Nele será descrita a paralelização lógica e a função de cada processador.

Arquitetura: Descreve a instância de STORM utilizada na simulação, seguido de uma figura esquemática. Na figura está indicada a distribuição lógica dos identificadores (*ID*) entre os processadores. Este identificador é determinante para a função do processador na aplicação.

Tabela de Resultados: Tabela que engloba todos os resultados obtidos na simulação da aplicação. Seus campos estão descritos na próxima página.

Tabela 5.1. Resultados obtidos nas simulações

Nome da Aplicação		
Ciclos gastos	Número total de ciclos decorridos na simulação.	
Processadores		**Total/Média**
Instruções executadas	Quantidade de instruções executadas por cada processador.	
Ciclos de execução	Número de ciclos até o término da execução de cada processador.	
Instruções por ciclo	Número de instruções executadas dividido pelos ciclos de execução.	
Saltos	Quantidade de saltos executados.	
Taken (%)	Quantidade de saltos tomados.	
Cache	**C0**	**Total/Média**
Acessos [Write/Read]	Número de acessos à cache para escrita/leitura.	
Hit %	Taxa de *Hit* da cache para operações de escrita/leitura	
Ciclos de espera	Somatório dos ciclos gastos em *Write Miss*, *Write Request*, *Read Miss*, *Test-and-Set* e *Miss* da cache de instruções.	
Diretório	**D0**	**Total/Média**
Requisições recebidas	Total de requisições recebidas pelo Diretório.	
Espera total	Somatório da espera individual de todas as requisições.	
Espera média/requisição	Campo "Espera total" dividido pelo campo "Requisições recebidas".	
Espera máxima	Maior espera individual de uma única requisição.	
NoC	Modelo de NoC utilizado na simulação.	
Bytes (pacotes) transmitidos	Quantidade total de bytes transmitidos pela rede, e entre parênteses o total de pacotes transmitidos.	
Throughput bytes/ciclo	Quantidade de bytes transmitidos dividida pela quantidade total de ciclos da simulação.	

Comentários: Comentários dos resultados obtidos, destacando os mais importantes para a simulação específica. Correlação com a metodologia de desenvolvimento e validação da plataforma.

Simulação 1: Fatorial ASM

Programa: fatorial.s

Código-Fonte: Anexo II.1.

Descrição: Programa escrito em *assembly* que calcula o fatorial de um número, passado no registrador %g1. O resultado final é gravado no próprio registrador %g1. O programa é bastante simples: o registrador %g2 recebe o valor de %g1, e é decrementado enquanto %g1 recebe o resultado da multiplicação de %g1 por %g2. Quando %g2 atinge o valor 0 o programa termina. Para esta simulação foi calculado o fatorial do número 10.

Propósito: Este programa foi o primeiro a ser simulado na plataforma. Serviu para o teste de instruções mais básicas do SPARC, incluindo-se aí a comparação e salto, essenciais para a construções do tipo *IF-THEN-ELSE* e *loops* em linguagem de alto nível. Também serviu para testar a integração inicial entre montador e plataforma.

Arquitetura: Esta simulação utilizou 1 SPARC e 1 módulo de Memória interligados por 2 roteadores da NoCX4, como visto na **Figura 5.1**.

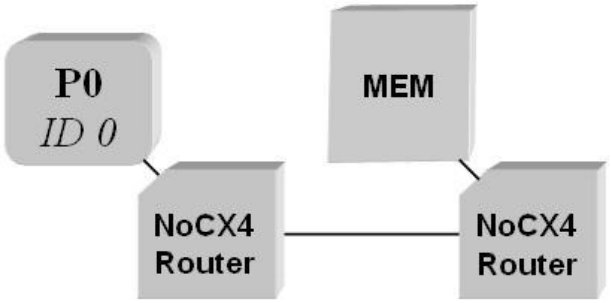


Figura 5.1. Ambiente de simulação do Fatorial ASM

Tabela 5.2. Resultados do Fatorial ASM

Fatorial ASM		
Ciclos gastos	285	
Processadores	**P0**	**Total/Média**
Instruções executadas	43	43
Ciclos de execução	285	285
Instruções por ciclo	0,15	0,15
Saltos	10	10
Taken (%)	9 (90%)	90 (90%)
Cache	**C0**	**Total/Média**
Acessos [Write/Read]	0/0	0/0
Hit %	-/-	-/-
Ciclos de espera	68	68
Diretório	**D0**	**Total/Média**
Requisições recebidas	2	2
Espera total	4	4
Espera média/requisição	2	2
Espera máxima	2	2
NoC	**NoCX4**	
Bytes (pacotes) transmitidos	96 (4)	
Throughput bytes/ciclo	0,336	

Comentários: Este programa não possui resultados expressivos, serviu apenas para validação e refinamento das ferramentas existentes. Após a realização de uma simulação bem sucedida deste código pôde-se passar para um programa mais complexo, com acesso à memória.

Simulação 2: Maior ASM

Programa: maior.s

Código-Fonte: Anexo II.2.

Descrição: Percorre um vetor de inteiros e identifica seu maior elemento. O registrador %g1 possui o endereço-base do vetor; o registrador %g2 serve de apontador de deslocamento do vetor, para os acessos indexados; o registrador %g3 recebe o valor da posição do vetor apontada por %g2; e finalmente o registrador %g4 contém o valor do maior elemento do vetor. Toda vez que %g3 recebe um valor maior do que o armazenado em %g4, o valor de %g3 é movido para %g4.

Propósito: O programa foi simulado para o teste de outras construções básicas, como o acesso à memória e acesso indexado (de um vetor). Após a validação deste programa todos os outros passaram a ser escritos na linguagem C.

Arquitetura: Esta simulação utilizou 1 SPARC e 1 módulo de Memória interligados por 2 roteadores da NoCX4, como visto na **Figura 5.2**.

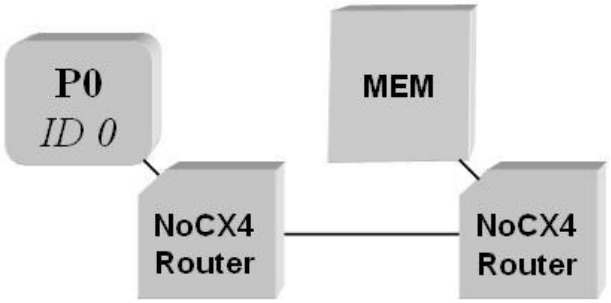


Figura 5.2. Ambiente de simulação do Maior ASM

Tabela 5.3. Resultados do Maior ASM

Maior ASM		
Ciclos gastos	671	
Processadores	**P0**	**Total/Média**
Instruções executadas	94	94
Ciclos de execução	671	671
Instruções por ciclo	0,14	0,14
Saltos	21	21
Taken (%)	17 (80,95%)	17 (80,95%)
Cache	**C0**	**Total/Média**
Acessos [Write/Read]	0/10	0/10
Hit %	-/80%	-/80%
Ciclos de espera	203	203
Diretório	**D0**	**Total/Média**
Requisições recebidas	5	5
Espera total	10	10
Espera média/requisição	2	2
Espera máxima	2	2
NoC	**NoCX4**	
Bytes (pacotes) transmitidos	240 (10)	
Throughput bytes/ciclo	0,357	

Comentários: Este foi o programa mais complexo implementado diretamente em *assembly*. Após o seu funcionamento completo já havia suporte suficiente por parte do montador e da plataforma para o teste da metodologia completa: gcc => Montador => Plataforma.

Simulação 3: Maior

Programa: maior.c

Código-Fonte: Anexo II.3.

Descrição: A exemplo do programa escrito em *assembly*, percorre um vetor de inteiros (variável *vetor*) de 10 posições, identifica e retorna o seu maior elemento (variável *maior*).

Propósito: Após a simulação do mesmo programa, porém escrito em *assembly*, o sistema já deveria estar em condições de passar por um teste completo. Este foi o primeiro programa escrito em C, compilado com o gcc, montado e carregado na plataforma. Seu principal objetivo, além de testar a metodologia completa, é o de avaliar as construções *assembly* geradas pelo compilador gcc a partir de um código C, e compará-lo ao mesmo código escrito em *assembly*.

Arquitetura: Esta simulação utilizou 1 SPARC e 1 módulo de Memória interligados por 2 roteadores da NoCX4, como visto na **Figura 5.3**.

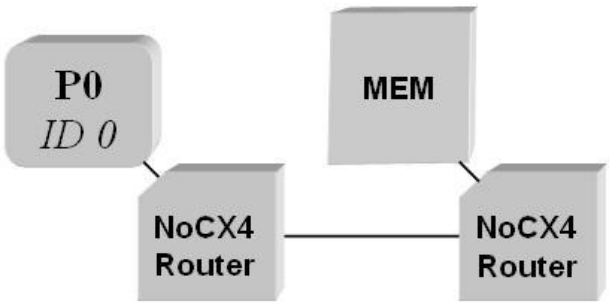


Figura 5.3. Ambiente de simulação do Maior

Tabela 5.4. Resultados do Maior

Maior		
Ciclos gastos	1510	
Processadores	**P0**	**Total/Média**
Instruções executadas	222	222
Ciclos de execução	1510	1510
Instruções por ciclo	0,15	0,15
Saltos	34	34
Taken (%)	21 (61,76%)	21 (61,76%)
Cache	**C0**	**Total/Média**
Acessos [Write/Read]	15/61	15/61
Hit %	93,3%/96,7%%	93,3%/96,7%%
Ciclos de espera	239	239
Diretório	**D0**	**Total/Média**
Requisições recebidas	7	7
Espera total	14	14
Espera média/requisição	2	2
Espera máxima	2	2
NoC	**NoCX4**	
Bytes (pacotes) transmitidos	336 (14)	
Throughput bytes/ciclo	0,222	

Comentários: Este talvez tenha sido o mais importante *benchmark* simulado, pois corroborou a metodologia que havia sido proposta para este trabalho. Além desta validação, a simulação se mostrou bastante rica para um maior entendimento da geração de código *assembly* pelo gcc, o que acarretou em diversas adaptações por parte do Montador. Também é interessante avaliar as diferenças entre duas versões de um mesmo programa, uma escrita diretamente em *assembly* e a outra em C, e comparar os resultados obtidos em ambas as simulações. O código escrito *assembly* é claramente mais eficiente do que o gerado automaticamente, executando em menos da metade do tempo.

Após a integração de todas as ferramentas, aplicações muito mais complexas puderam ser simuladas.

Simulação 4: Quicksort

Programa: quicksort.c

Código-Fonte: Anexo II.4.

Descrição: Este programa implementa o famoso algoritmo de ordenação *Quicksort* recursivo. O *Quicksort* é um eficiente algoritmo de ordenação do tipo dividir para conquistar (*divide and conquer*). Em seu caso médio possui complexidade O(n log n), e O(n^2) no pior caso. O programa principal *main* inicializa o vetor de 64 posições a ser ordenado de forma decrescente e chama a função de *quicksort*, que recursivamente divide e ordena o vetor em ordem crescente.

Propósito: Este programa foi simulado não só por representar um avanço significativo na complexidade das aplicações testadas, mas também para testar duas construções em C que não puderam ser testadas com o programa anterior: a chamada de função e implementação de função recursiva, que faz uso intenso da pilha.

Arquitetura: Esta simulação utilizou 1 SPARC e 1 módulo de Memória interligados por 1 roteador da Árvore Obesa, como visto na **Figura 5.4**.

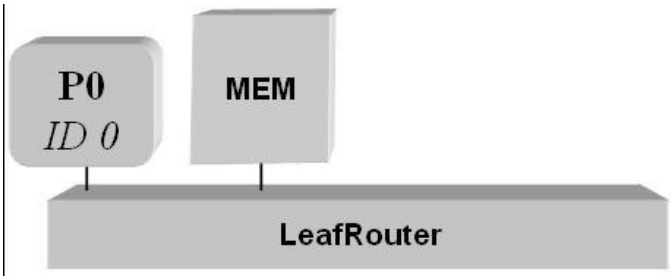


Figura 5.4. Ambiente de simulação do *Quicksort*

Tabela 5.5. Resultados do *Quicksort*

Quicksort		
Ciclos gastos	55.403	
Processadores	**P0**	**Total/Média**
Instruções executadas	9.024	9.024
Ciclos de execução	55.403	55.403
Instruções por ciclo	0,16	0,16
Saltos	901	901
Taken (%)	502 (55,71%)	502 (55,71%)
Cache	**C0**	**Total/Média**
Acessos [Write/Read]	866/3346	866/3346
Hit %	97,2%/99,9%	97,2%/99,9%
Ciclos de espera	1.564	1.564
Diretório	**D0**	**Total/Média**
Requisições recebidas	55	55
Espera total	110	110
Espera média/requisição	2	2
Espera máxima	2	2
NoC	**Árvore Obesa**	
Bytes (pacotes) transmitidos	2.520 (110)	
Throughput bytes/ciclo	0,045	

Comentários: O *Quicksort* é um algoritmo extremamente eficiente. Seu desempenho poderá ser comparado com a simulação do algoritmo *Mergesort* paralelo, mostrada mais a frente, na **Simulação 7**.

O próximo passo após o funcionamento correto das construções básicas de C e da simulação de uma aplicação complexa como o *Quicksort* é o estabelecimento de uma forma de programação concorrente em C.

Simulação 5: Mutex

Programa: mutex.c

Código-Fonte: Anexo II.5.

Descrição: Este programa implementa o recurso de programação paralela conhecido como *mutex*. O programa é bastante simples: na função principal (*main*) os processadores tentam acessar a função *get_id()* que está implementada na biblioteca *mutex.h*. Para tal, é necessário que consigam acesso ao *mutex1* através da função *down()*. A variável global *application_id*, também declarada na biblioteca, começa com o valor 0. A cada acesso à função *get_id()* ela é incrementada. Ao final de uma simulação bem sucedida deste programa cada processador deve possuir um valor diferente para sua variável local *id*.

Propósito: A simulação deste programa tem o propósito claro de testar o funcionamento da biblioteca *mutex.h*, implementada em C usando algumas instruções específicas do SPARC para acesso atômico à memória. Com o uso desta biblioteca torna-se possível a definição de regiões críticas no código e conseqüente acesso exclusivo a ela. A função *get_id()* deve prover um valor identificador lógico diferente para cada processador, sendo assim possível paralelizar o código escrito em C sem necessidade de recursos extras ou edição do código *assembly*.

Arquitetura: Esta simulação utilizou 3 SPARC e 1 módulo de Memória interligados por 1 roteador da Árvore Obesa, como visto na **Figura 5.5**.

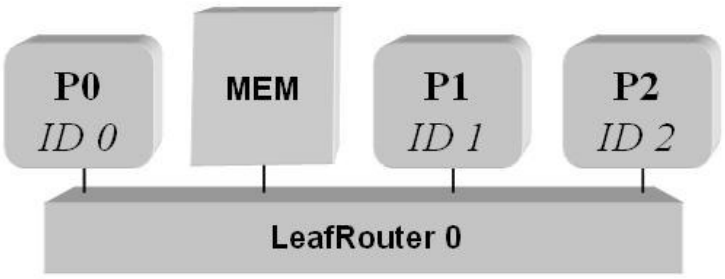


Figura 5.5. Ambiente de simulação do *Mutex*

Tabela 5.6. Resultados do *Mutex*

Mutex				
Ciclos gastos		1.568		
Processadores	**P0**	**P1**	**P2**	**Total/Média**
Instruções executadas	57	102	156	315
Ciclos de execução	742	1.122	1568	1.568
Instruções por ciclo	0,07	0,09	0,10	0,20
Saltos	2	12	24	38
Taken (%)	2 (100%)	7 (58,33%)	13 (54,16%)	22 (57,89%)
Cache	**C0**	**C1**	**C2**	**Total/Média**
Acessos [Write/Read]	8/8	13/18	19/30	40/56
Hit %	62,5%/87,5%	77%/94,5%	84,4%/97,6%	77,5%/94,7%
Ciclos de espera	457	589	717	1.763
Diretório	**D0**			**Total/Média**
Requisições recebidas	61			61
Espera total	258			258
Espera média/requisição	4,22			4,22
Espera máxima	24			24
NoC	**Árvore Obesa**			
Bytes (pacotes) transmitidos	2.196 (126)			
Throughput bytes/ciclo	1.400			

Comentários: Esta simulação consiste apenas em um processo de aquisição do identificador único. Foram simulados 3 processadores e uma memória em um único roteador de Árvore Obesa para se ter uma idéia do tempo decorrido neste processo de aquisição, que se torna obrigatório nas simulações concorrentes, a seguir. Pode-se ver pelos resultados que há uma longa espera por parte dos processadores, e por outro lado uma utilização intensa da NoC. Ao final, o tempo necessário para este procedimento padrão não é alto o suficiente para ter influência significativa no desempenho das aplicações, o que pode ser confirmado em comparando-se o tempo desta simulação e das simulações conseguintes.

O funcionamento desta biblioteca é o último passo antes da simulação de aplicações paralelas. Com as simples funções nela implementadas é possível paralelizar o código pelo *id* de cada processador, em nível lógico, sem necessidade de se conhecer a arquitetura. É possível também garantir a exclusão mútua no acesso a variáveis globais, que são a única forma de

comunicação entre processos em um ambiente com memória compartilhada. As próximas aplicações simuladas apresentam o poder desta biblioteca.

Simulação 6: Multiplicação de Matrizes

Programa: mul_mat.c

Código-Fonte: Anexo II.6.

Descrição: Programa que implementa uma operação bastante utilizada nas aplicações matriciais: a Multiplicação de Matrizes. Duas matrizes de tamanho 8x8, *M_A* e *M_B*, são multiplicadas e seu resultado é guardado na matriz de resultado *M_C*. O algoritmo utilizado é o mais conhecido, de complexidade $O(n^3)$, que percorre as linhas de *M_A* e as colunas de *M_B*, multiplicando e acumulando o resultado, e por fim salvando-o na matriz *M_C*.

Propósito: A multiplicação de matrizes é uma aplicação simples e que requer intenso esforço computacional devido à sua complexidade cúbica. Ademais, sua paralelização é bastante simples se forem consideradas algumas limitações. Por estas razões, foi escolhida para ser a primeira aplicação real paralela a ser simulada.

Paralelização: É feita automaticamente, considerando apenas a multiplicação de matrizes quadradas, cujo tamanho seja um múltiplo do número de processadores. Assim, cada processador calcula uma ou mais linhas da matriz *M_C*. A o número de linhas que cada processador deve calcular é dado pela constante *STEP*. Por causa de sua paralelização automática, a multiplicação de matrizes foi simulada com diferentes quantidades de processadores.

Arquitetura: Esta aplicação possui 5 simulações, sendo 4 delas usando a NoCX4 (com 1, 2, 4 e 8 processadores) e uma, com 8 processadores, que utiliza a Árvore Obesa para comparação. As arquiteturas utilizadas estão representadas nas figuras que seguem.

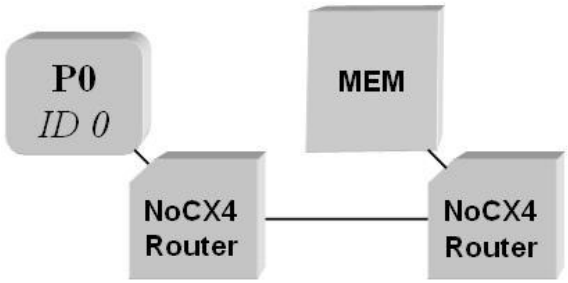


Figura 5.6. Ambiente de simulação da *Multiplicação de Matrizes* com 1 processador

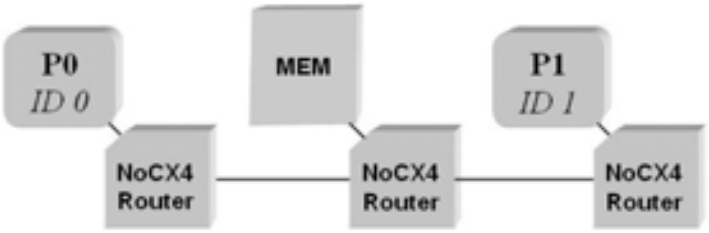


Figura 5.7. Ambiente de simulação da *Multiplicação de Matrizes* com 2 processadores

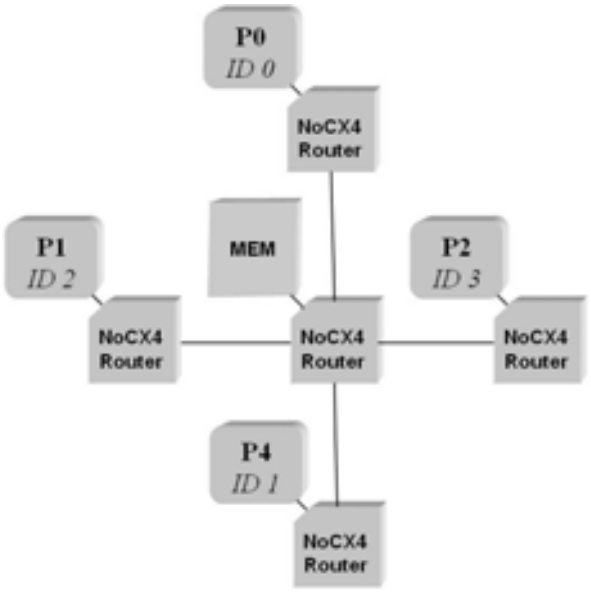


Figura 5.8. Ambiente de simulação da *Multiplicação de Matrizes* com 4 processadores

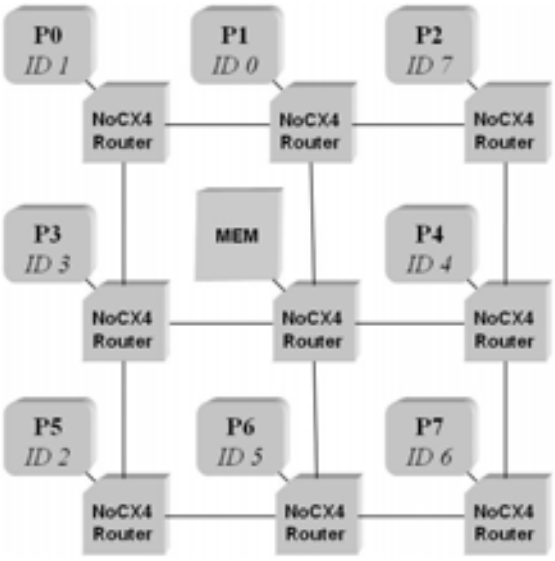


Figura 5.9. Ambiente de simulação da *Multiplicação de Matrizes* com 8 processadores

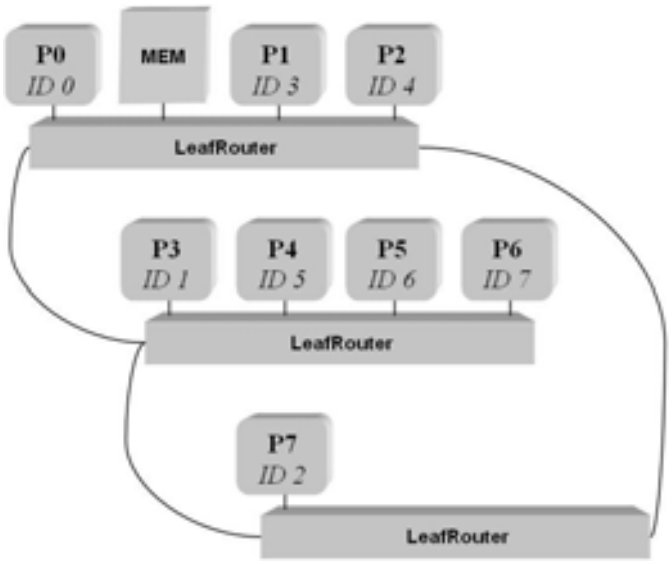


Figura 5.10. Ambiente de simulação da *Multiplicação de Matrizes* com 8 processadores usando Árvore Obesa

Tabela 5.7. Resultados da Multiplicação de Matrizes com 1 processador

Multiplicação de Matrizes (1 Processador)		
Ciclos gastos	114.000	
Processadores	**P0**	**Total/Média**
Instruções executadas	18.762	18.762
Ciclos de execução	114.000	114.000
Instruções por ciclo	0,16	0,16
Saltos	1307	1307
Taken (%)	723 (55,31%)	723 (55,31%)
Cache	**C0**	**Total/Média**
Acessos [Write/Read]	1.689/5.419	1.689/5.419
Hit %	99,3%/99,7%	99,3%/99,7%
Ciclos de espera	4.918	4.918
Diretório	**D0**	**Total/Média**
Requisições recebidas	178	178
Espera total	356	356
Espera média/requisição	2	2
Espera máxima	2	2
NoC	**NoCX4**	
Bytes (pacotes) transmitidos	4.356 (356)	
Throughput bytes/ciclo	0,038	

Tabela 5.8. Resultados da Multiplicação de Matrizes com 2 processadores

Multiplicação de Matrizes (2 Processadores)			
Ciclos gastos		58.268	
Processadores	**P0**	**P1**	**Total/Média**
Instruções executadas	9.414	9.469	18.883
Ciclos de execução	57.810	58.268	58.268
Instruções por ciclo	0,16	0,16	0,32
Saltos	655	667	1322
Taken (%)	363 (55,41%)	369 (55,32%)	732 (55,37%)
Cache	**C0**	**C1**	**Total/Média**
Acessos [Write/Read]	849/2.715	855/2.727	1.704/5.442
Hit %	99,2%/99,5%	99,2%/99,5%	99,2%/99,5%
Ciclos de espera	3.117	3.256	6.373
Diretório	**D0**		**Total/Média**
Requisições recebidas	219		219
Espera total	514		514
Espera média/requisição	2,34		2,34
Espera máxima	35		35
NoC		**NoCX4**	
Bytes (pacotes) transmitidos		6.068 (440)	
Throughput bytes/ciclo		0,104	

Tabela 5.9. Resultados da Multiplicação de Matrizes com 4 processadores

Multiplicação de Matrizes (4 Processadores)					
Ciclos gastos			31.223		
Processadores	**P0**	**P1**	**P2**	**P3**	**Total/Média**
Instruções executadas	4.741	4.849	4.903	4.795	19.228
Ciclos de execução	29.729	30.795	31.223	30.302	31.223
Instruções por ciclo	0,16	0,16	0,16	0,16	0,61
Saltos	329	353	365	341	1.388
Taken (%)	183 (55,6%)	195 (55,2%)	201 (55,0%)	189 (55,4)	768 (55,3%)
Cache	**C0**	**C1**	**C2**	**C3**	**Total/Média**
Acessos [Write/Read]	429/1.363	441/1.387	447/1399	435/1.375	1.752/5.524
Hit %	98,8%/ 99,0%	98,9%/ 99,0%	98,9%/ 99,1%	98,8%/ 99,1%	98,9%/99,1%
Ciclos de espera	2.239	2.667	2.775	2.493	10.174
Diretório	**D0**				**Total/Média**
Requisições recebidas	319				319
Espera total	1.265				1.265
Espera média/requisição	3,96				3,96
Espera máxima	35				35
NoC			**NoCX4**		
Bytes (pacotes) transmitidos			9.780 (644)		
Throughput bytes/ciclo			0,313		

Tabela 5.10. Resultados da Multiplicação de Matrizes com 8 processadores

Multiplicação de Matrizes (8 Processadores)									
Ciclos gastos					20.939				
Processador	**P0**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**Total/ Média**
Instruções executadas	2.526	2.463	2.869	2.707	2.770	2.635	2.797	2.761	21.528
Ciclos de execução	17.555	16.593	20.939	18.886	19.511	18.650	19.749	19.980	20.939
Instruções por ciclo	0,14	0,15	0,14	0,14	0,14	0,14	0,14	0,14	1,028
Saltos	194	180	270	234	248	218	254	246	1.844
Taken (%)	107 55,1%	100 55,6%	145 53,7%	127 54,3%	134 54,0%	119 54,6%	137 54,0%	133 54,0%	1.002 54,3%
Cache	**C0**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**Total/ Média**
Acessos [Write/Read]	233/ 715	226/ 701	271/ 791	253/ 755	260/ 769	245/ 739	263/ 775	259/ 767	2.010/ 6.012
Hit %	98,3%/ 98,3%	98,2%/ 98,3%	98,5%/ 98,6%	98,4%/ 98,4%	98,5%/ 98,4%	98,4%/ 98,4%	98,5%/ 98,5%	98,5%/ 98,4%	98,3%/ 98,4%
Ciclos de espera	3.063	2.474	4.423	3.319	3.577	3.532	3.659	4.086	28.133

Diretório	**D0**	**Total/Média**
Requisições recebidas	681	681
Espera total	6.230	6.230
Espera média/requisição	9,14	9,14
Espera máxima	79	79

NoC	**NoCX4**
Bytes (pacotes) transmitidos	19.796 (1.376)
Throughput bytes/ciclo	0,94

Tabela 5.11. Resultados da Multiplicação de Matrizes com 8 processadores usando Árvore Obesa

Multiplicação de Matrizes (8 Processadores - ObTree)									
Ciclos gastos					20.027				
Processador	**P0**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**Total/ Média**
Instruções executadas	2.508	2.562	2.761	2.491	2.716	2.770	2.815	2.554	21.177
Ciclos de execução	16.659	18.017	18.896	16.910	19.134	19.602	20.027	17.571	20.027
Instruções por ciclo	0,15	0,15	0,15	0,15	0,14	0,14	0,14	0,15	1,057
Saltos	190	222	246	186	236	248	258	200	1.786
Taken (%)	105 55,3%	121 54,5%	133 54,0%	103 55,4%	128 54,2%	134 54,0%	139 53,9%	110 55%	973 54,4%
Cache	**C0**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**Total/ Média**
Acessos [Write/Read]	231/ 711	247/ 743	259/ 767	229/ 707	254/ 757	260/ 769	265/ 779	236/ 721	1.981/ 5.954
Hit %	98,2%/ 98,3%	98,4%/ 98,4%	98,5%/ 98,4%	98,2%/ 98,3%	98,4%/ 98,4%	98,5%/ 98,4%	98,5%/ 98,6%	98,3%/ 98,3%	98,3%/ 98,4%
Ciclos de espera	2.277	2.772	2.998	2.639	3.515	3.664	3.802	2.930	24.597

Diretório	**D0**	**Total/Média**
Requisições recebidas	652	652
Espera total	6.882	6.882
Espera média/requisição	10,55	10,55
Espera máxima	79	79

NoC	**NoCX4**
Bytes (pacotes) transmitidos	19.332 (1.318)
Throughput bytes/ciclo	0,96

Comentários: Esta simulação é a mais extensa apresentada neste trabalho. Pode-se perceber que, à medida que aumenta o número de processadores, maior é o tempo gasto pelo diretório para processar as requisições, que vai se tornando cada vez mais o gargalo do sistema. Apesar disto, o índice de IPC (Instruções por Ciclo) é sempre crescente, passando de 1 quando

se utiliza 8 processadores. Apesar de parecer um índice baixo, deve-se lembrar de que a arquitetura SPARC implementada não é superescalar, e nem conta com mecanismos comuns nestas arquiteturas, como um mecanismo de previsão de salto ou memória de traço (*trace memory*), que aumentam sensivelmente a quantidade de instruções executadas por ciclo. Nota-se também daí que o ganho do paralelismo torna-se cada vez menor como aumento de processadores devido ao tempo maior gasto em operações de *miss* na cache e de acesso atômico à memória. Apesar de nesta simulação não ter sido atingido o ponto crítico onde o aumento no número de processadores em paralelo tem efeito degradante sobre o desempenho do sistema, pode-se inferir que ele não está muito distante.

Por ser a única simulação que envolve a quantia de 8 processadores, resolveu-se usá-la para uma comparação entre os dois modelos de NoC utilizados em STORM: a NoCX4 e a Árvore Obesa. Logo de início é possível observar um ganho de 4,5% no desempenho total do sistema. Este ganho poderia ter sido maior se não fosse pelo gargalo formado no Diretório, pois levando-se apenas em consideração os ciclos de espera em operações de *miss* da cache e acesso atômico à memória, operações estas que dependem diretamente da NoC utilizada pois resultam em transmissão de pacotes pela mesma, existe um ganho de 14,5% quando utilizada a Árvore Obesa. Porém, é notório que o tempo gasto por requisição pelo Diretório aumenta em 15,4%, mais de 1 ciclo em média, o que impede que o ganho obtido por uma transmissão mais veloz de pacotes seja integralmente absorvido pela aplicação. Uma otimização no Diretório utilizado ou ainda o uso de outros módulos de memória para aliviar o gargalo do Diretório certamente aumentaria ainda mais o ganho com o uso da Árvore Obesa.

Apesar de a latência de transmissão dos pacotes não ter sido medida, ela é claramente percebida se comparados os ciclos de espera das Caches individualmente. Os últimos processadores a terminar de executar, o P2 no caso da NoCX4 e o P6 no caso da ObTree, possuem entre si uma diferença de 16,3% na quantidade de ciclos gastos ociosamente em espera. Por passarem mais tempo em execução, os efeitos de uma menor latência na comunicação tornam-se mais perceptível entre eles. O ganho geral de *throughput* é de 2,1%.

Simulação 7: Mergesort

Programa: mergesort.c

Código-Fonte: Anexo II.7.

Descrição: Esta é a simulação do algoritmo de ordenação *Mergesort*. Ele é similar ao *Quicksort* pois também utiliza a estratégia de dividir para conquistar (*divide and conquer*). Porém, ao contrário do *Quicksort*, ele possui complexidade O(*n* log *n*) mesmo no pior caso. Outra diferença é que o *Mergesort* ocupa mais memória. Esta implementação em particular utiliza uma função de mescla (função *merge()*) mais eficiente, tanto em termos de desempenho quanto de memória ocupada, necessitando de um vetor extra com metade do tamanho do vetor original.

Propósito: O *Mergesort* é um algoritmo mais facilmente paralelizável do que o *Quicksort*, portanto foi escolhido para ser o algoritmo de ordenação paralelo simulado. Ademais, pode-se comparar os resultados de ambos os algoritmos.

Paralelização: O *Mergesort* foi implementado para rodar em 2 processadores. Ambos os processadores começam inicializando as posições do vetor paralelamente. Após esta fase, cada processador chama a função *mergesort()* passando como parâmetros a sua metade do vetor e um vetor local, necessário pelo algoritmo para fazer operações intermediárias. Após ambos os processadores terminarem a ordenação de suas metades do vetor, o processador *P0* faz uma mescla das duas metades do vetor para ordená-lo por completo, através da função *merge()*. O controle do término é feito via a variável *mutex2*.

Arquitetura: Esta simulação utilizou 2 SPARC e 1 módulo de Memória interligados por 1 roteador da Árvore Obesa, como visto na **Figura 5.11**.

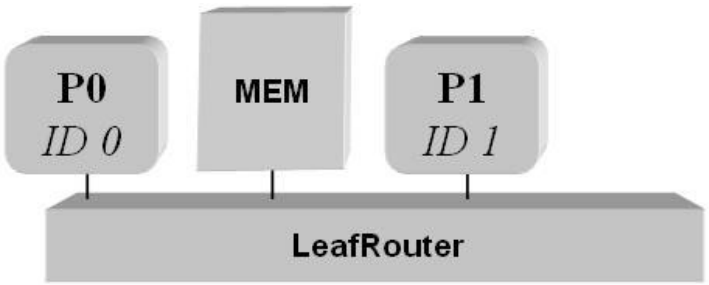


Figura 5.11. Ambiente de simulação do *Mergesort*

Tabela 5.12. Resultados do *Mergesort*

Mergesort			
Ciclos gastos		80.752	
Processadores	**P0**	**P1**	**Total/Média**
Instruções executadas	13.895	10.860	24.755
Ciclos de execução	80.752	63.132	80.752
Instruções por ciclo	0,17	0,17	0,30
Saltos	1.181	909	2090
Taken (%)	626 (53%)	488 (53,68%)	1.114 (53,3%)
Cache	**C0**	**C1**	**Total/Média**
Acessos [Write/Read]	1.576/3.675	1.274/2.886	2.850/6.561
Hit %	98,5%/99,8%	98,4%/99,7%	98,4%/99,8%
Ciclos de espera	2.280	1.880	4.160
Diretório	**D0**		**Total/Média**
Requisições recebidas	147		147
Espera total	382		382
Espera média/requisição	2,59		2,59
Espera máxima	14		14
NoC		**Árvore Obesa**	
Bytes (pacotes) transmitidos		6.196 (304)	
Throughput bytes/ciclo		0,076	

Comentários: Esta simulação produz um resultado interessante: apesar de ser a implementação paralela de um algoritmo de ordenação, ela possui desempenho inferior à implementação escalar do *Quicksort*, mesmo tendo igual complexidade. Isto se deve a diferenças nos algoritmos, onde o pior caso para um não necessariamente é o pior caso para outro.

Pode-se notar pelos resultados que durante a maior parte do tempo os processadores ficam ocupados rodando uma mesma porção do código e acessando uma mesma região de memória diversas vezes, tanto que o tempo de espera é muito baixo, assim como o *throughput* da NoC. O IPC por processador também está dentre os mais altos dentre as simulações aqui apresentadas.

Simulação 8: KMP

Programa: kmp.c

Código-Fonte: Anexo II.8.

Descrição: O algoritmo de busca conhecido por KMP (*Knuth-Morris-Pratt*) foi inicialmente proposto pelos autores homônimos em [KNU 77]. O seu objetivo é identificar um determinado padrão *P* em uma seqüência *S*, verificando apenas uma vez cada termo da seqüência. Para isto, o algoritmo prevê a composição de uma tabela de resultados parciais, o vetor *KMPNext[]* no programa, que possui informações sobre o próximo termo a ser verificado no caso de um *miss*. Após a composição desta tabela, o programa percorre em *loop* toda a extensão da seqüência até encontrar o primeiro *match* perfeito.

Propósito: Assim como algoritmos de ordenação são fundamentais em diversas classes de aplicações que envolvem a manipulação e armazenamento de dados, os algoritmos de busca estão presentes em aplicações que vão desde editores de texto até análise de código genético. O algoritmo KMP é um algoritmo de busca de simples compreensão e de fácil implementação, tendo sido por estas razões escolhido para simulação.

Paralelização: Apesar de a maneira mais simples de paralelização ser a simples quebra da seqüência de busca em partes iguais, para este caso preferiu-se uma abordagem diferente: são feitas buscas por dois diferentes padrões em uma mesma seqüência, em paralelo. Cada processador possui seu padrão *P* e executa o algoritmo independentemente do outro processador.

Arquitetura: Esta simulação utilizou 2 SPARC e 1 módulo de Memória interligados por 1 roteador da Árvore Obesa, como visto na **Figura 5.12**

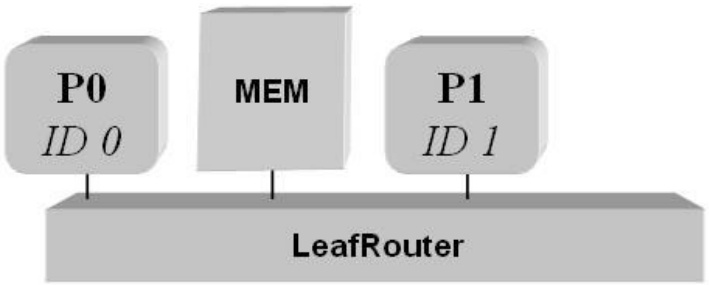


Figura 5.12 Ambiente de simulação do *KMP*

Tabela 5.13. Resultados do KMP

KMP			
Ciclos gastos		266.264	
Processadores	**P0**	**P1**	**Total/Média**
Instruções executadas	44.011	27.435	71.446
Ciclos de execução	266.264	166.601	266.264
Instruções por ciclo	0,17	0,16	0,268
Saltos	5.758	3.596	9.354
Taken (%)	2.843 (49,4%)	1.803 (50,1%)	4.646 (49,6%)
Cache	**C0**	**C1**	**Total/Média**
Acessos [Write/Read]	2.871/15.816	1.829/9.853	4.700/25.669
Hit %	99,7%/99,2%	99,5%/99,2%	99,6%/99,2%
Ciclos de espera	4.735	3.656	8.391
Diretório	**D0**		**Total/Média**
Requisições recebidas	295		295
Espera total	694		694
Espera média/requisição	2,35		2,35
Espera máxima	29		29
NoC		**Árvore Obesa**	
Bytes (pacotes) transmitidos		13.700 (593)	
Throughput bytes/ciclo		0,051	

Comentários: Uma rápida análise da tabela de resultados permite chegar à conclusão de que o processador P1 encontrou o seu padrão muito antes do processador P0. Isto é perfeitamente aceitável, já que os dois estão efetuando buscas independentes. O padrão buscado por P1 começa na posição 552, enquanto o buscado por P0 só será encontrado na posição 921 do vetor. Por se tratar de um algoritmo de intenso esforço computacional pode-se notar que o tempo de espera é bastante baixo em relação ao tempo total de execução, assim como a utilização da NoC e do Diretório.

Simulação 9: Estimação de Movimento

Programa: estimacao.c

Código-Fonte: Anexo II.9.

Descrição: A Compensação de Movimento é uma etapa da codificação MPEG. Seu princípio se baseia na redundância encontrada entre os quadros (*frames*) consecutivos. Ela visa compactar a informação contida nos *frames* eliminando esta redundância. O processo se dá pela identificação do deslocamento de um bloco entre dois *frames*. A este processo se dá o nome de *Estimação de Movimento*. Existem vários algoritmos para identificar este deslocamento. O algoritmo aqui utilizado produz o resultado ótimo, pois realiza uma busca completa (algoritmo *full search*). O algoritmo consiste em varrer todo o espaço de busca até identificar um bloco, e a partir daí gerar um vetor de movimento (*motion vector*). O programa implementado para esta simulação realiza a principal parte deste algoritmo, que é a identificação do bloco.

Propósito: Uma área cada vez mais importante em aplicações computacionais, a multimídia teve duas aplicações importantes simuladas neste trabalho: a *Estimação de Movimento* e a Transformada Discreta do Cosseno (*DCT*). A Estimação de Movimento é um processo particularmente importante na codificação MPEG, pois exige esforço computacional intenso e produz excelentes ganhos de compressão.

Paralelização: Apesar de possuir semelhanças com o algoritmo de busca simulado anteriormente, o KMP, a Estimação de Movimento foi simulada com outra proposta: a de ter dois processadores buscando uma mesma informação paralelamente. O importante desta simulação é o sincronismo entre os processos, e a definição de um mecanismo de parada assim que a primeira ocorrência do bloco fosse encontrada. Para este mecanismo foi necessária a utilização do *Mutex*. No início da execução, o processador de *ID* 0 inicializa a variável de posição do bloco com -1. Quando um dos processadores encontra o bloco esta variável recebe as coordenadas do início do bloco no espaço de busca. A cada nova iteração no laço o processador verifica se a variável que contém a posição do bloco foi modificada. Caso tenha sido, ele sai do laço e termina sua execução. Caso contrário, ele executa a nova iteração.

Arquitetura: Esta simulação utilizou 2 SPARC e 1 módulo de Memória interligados por 1 roteador da Árvore Obesa, como visto na **Figura 5.13**

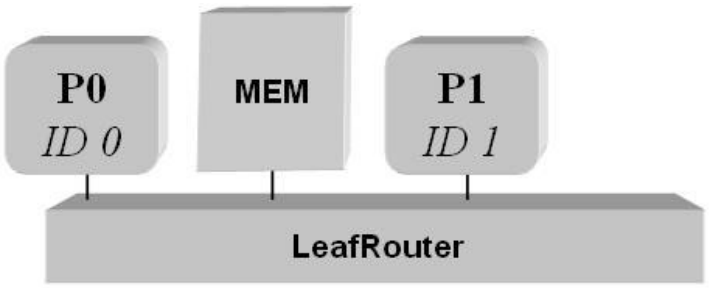


Figura 5.13 Ambiente de simulação da Estimação de Movimento

Tabela 5.14. Resultados da Estimação de Movimento

Estimação de Movimento			
Ciclos gastos		10.004	
Processadores	**P0**	**P1**	**Total/Média**
Instruções executadas	1.438	1.357	2.795
Ciclos de execução	10.004	9.628	10.004
Instruções por ciclo	0,14	0,14	0,279
Saltos	147	139	286
Taken (%)	81 (55,1%)	86 (50,1%)	167 (58,39%)
Cache	**C0**	**C1**	**Total/Média**
Acessos [Write/Read]	162/349	112/368	274/717
Hit %	95,7%/98,3%	94,7%/97,3%	95,3%/97,9%
Ciclos de espera	1.878	1.907	3.785
Diretório	**D0**		**Total/Média**
Requisições recebidas	145		145
Espera total	399		399
Espera média/requisição	2,75		2,75
Espera máxima	29		29
NoC		**Árvore Obesa**	
Bytes (pacotes) transmitidos		5.284 (295)	
Throughput bytes/ciclo		0,528	

Comentários: Avaliando o tempo de execução dos processadores pode-se ver que o mecanismo de parada funcionou como esperado. Menos de 400 ciclos após o término da execução do processador P1, que encontrou uma ocorrência do bloco, o processador P0 encerra sua execução mesmo sem ter encontrado o bloco. Por esta aplicação envolver intenso acesso a uma matriz houve um elevado número de acessos à memória. Somando-se isso à necessidade de efetuar uma operação de acesso atômico à memória a cada iteração, teve-se uma alta taxa de utilização da NoC.

Simulação 10: DCT 2D

Programa: dct.c

Código-Fonte: Anexo II.10.

Descrição: A Transformada Discreta do Cosseno (DCT) 2D é uma ferramenta matemática com diversas aplicações na eletrônica [AGO 01]. Sua aplicação mais conhecida e utilizada talvez seja como parte principal do algoritmo de compressão de imagem JPEG. Por causa de sua alta complexidade computacional, muitos autores propuseram versões simplificadas da DCT. A versão escolhida para simulação é uma otimização da utilizada em [KOV 95], baseada no princípio da separabilidade. A otimização implementada tão-somente elimina as operações desnecessárias. Como a DCT envolve constantes em ponto flutuante e a unidade de ponto flutuante do SPARC ainda não foi implementada, elas foram multiplicadas por 256 e o resultado das multiplicações envolvendo as constantes foi dividido por 256 após sua computação, conforme feito em [AZE 03].

Propósito: A Transformada Discreta do Cosseno (DCT) é uma das operações mais importantes na área de Processamento Digital de Sinais (DSP) porque converte a informação do domínio do tempo ou do espaço para o domínio da freqüência, o que resulta em sua compactação. Sua utilização é particularmente importante nas aplicações de compressão de imagem e vídeo, que possuem enorme volume de dados. Além disso, esta implementação da DCT visa testar um mecanismo que garanta o sincronismo entre os processos para que ocorra um *pipeline* de processamento.

Paralelização: Como o algoritmo utilizado faz uso do princípio da separabilidade da DCT 2D, dividindo-a em duas operações de DCT 1D, para esta implementação são usados dois processadores que formam um *pipeline*. O processador de *ID* 0 busca o primeiro bloco a ser processado e chama a função *dct()*. Assim que a primeira DCT termina seu resultado é salvo em um *buffer* intermediário de forma transposta e o processador de *ID* 1 começa a operar a segunda DCT sobre o bloco. Concomitantemente, o processador de *ID* 0 busca o segundo bloco e o opera.

Para que este *pipeline* funcione adequadamente foi utilizada uma variável de controle *id1_ok*. Sempre que termina a DCT de seu bloco o processador de *ID* 0 testa esta variável. Caso ela esteja em 0 o processador copia o resultado de sua matriz local para a matriz global *buffer* e muda o valor da variável *id1_ok* para 1. Por sua vez, o processador de *ID* 1 só copia o conteúdo

do *buffer* intermediário para sua matriz local se o valor da variável *id1_ok* estiver em 1, o que significa que o bloco que está no *buffer* é válido. Após a cópia do bloco do *buffer* intermediário para sua matriz local o valor da variável torna a ser 0, o que indica para o processador de *ID* 0 que o bloco que está no buffer intermediário já foi consumido pelo outro processador. Para esta simulação foram utilizados 4 blocos de 8x8 pixels.

Arquitetura: Esta simulação utilizou 2 SPARC e 1 módulo de Memória interligados por 1 roteador da Árvore Obesa, como visto na **Figura 5.14**.

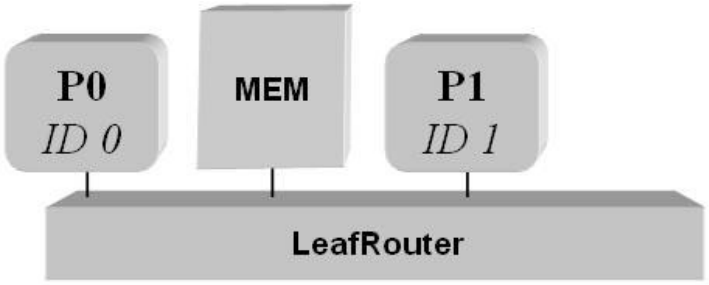


Figura 5.14. Ambiente de simulação da DCT

Tabela 5.15. Resultados da DCT 2D

DCT 2D			
Ciclos gastos		186.063	
Processadores	**P0**	**P1**	**Total/Média**
Instruções executadas	24.132	29.712	53.844
Ciclos de execução	147.483	186.063	186.063
Instruções por ciclo	0,16	0,16	0,289
Saltos	1.334	2.003	3.337
Taken (%)	710 (53,2%)	1.136 (56,7%)	1.846 (55,31%)
Cache	**C0**	**C1**	**Total/Média**
Acessos [Write/Read]	2.349/8.141	3.050/9.178	5.399/17.319
Hit %	98,5%/99,5%	98,1%/99,6%	98,3%/99,5%
Ciclos de espera	4.884	12.016	16.900
Diretório	**D0**		**Total/Média**
Requisições recebidas	689		689
Espera total	1.518		1.518
Espera média/requisição	2,20		2,20
Espera máxima	18		18
NoC		**Árvore Obesa**	
Bytes (pacotes) transmitidos		21.716 (1.452)	
Throughput bytes/ciclo		0,116	

Comentários: A análise dos resultados mostra que o mecanismo de sincronismo teve o efeito desejado: o processador P1 passa 12.000 ciclos em espera, a maior parte deles esperando pelo término da primeira DCT do processador P1. Isto é devido à latência inicial do *pipeline* formado nesta implementação. A diferença entre os ciclos de execução dos processadores pode ser vista como um indicativo do valor desta latência – 38.580 ciclos – e também do tempo gasto no cálculo de cada DCT 1D. Outra forma de estimar este valor é dividindo-se o tempo total te execução de P0 pelo número de blocos – quatro – o que resulta em 36.870 ciclos.

6. Conclusões

A nova tendência de projeto, o Sistema-em-Chip, aumenta consideravelmente a complexidade e o custo de desenvolvimento sistema. Os aspectos envolvidos em tais projetos são dos mais diversos, ainda existindo por isso muitas questões para estudo neste campo.

Este projeto teve por objetivo a proposta de uma plataforma para desenvolvimento de sistemas MP-SoC. Esta plataforma provê dois modelos de *Network-on-Chip*, sendo um deles proposto no âmbito do trabalho, dois processadores, sendo um proposto no âmbito do trabalho e o outro um processador comercial com suporte à linguagem C, mecanismo de coerência de cache e uma biblioteca para programação concorrente em C. Toda a plataforma foi implementada em SystemC e diversas simulações foram feitas para a validação de todos os módulos e conceitos propostos, tanto individualmente quanto com a integração completa do sistema.

Sua característica mais importante do ponto de vista acadêmico, porém, é a variedade de experimentos que podem ser feitos com um trabalho tão abrangente. Por envolver diversos conceitos, é possível estudar os mais variados aspectos de sua implementação. Uma implementação funcional da plataforma também permite o desenvolvimento de ferramentas que facilitam seu uso, como sistema operacional e bibliotecas específicas de programação. Permite também estudos mais profundos em sistemas de interconexão, coerência de cache, sincronismo entre processos, escalonamento de processos, Entrada/Saída, consumo de energia, ocupação de área, otimização dos módulos, dentre outros. Estas são contribuições que este trabalho deixará para o grupo de pesquisa.

De uma maneira mais imediata, a conclusão deste trabalho é mais um passo no sentido da criação de sistemas MP-SoC reais de grande porte. Apesar de já existirem no mercado, os SoCs atuais são de pequeno porte, integrando poucos blocos. Com a previsão de sistemas em chip de grande porte até o fim da década, trabalhos como este são de grande valia para o estudo das diversas questões que envolvem a integração de centenas de blocos em um único chip, e para apontar soluções para os atuais problemas.

Alguns trabalhos futuros mais imediatos podem ser mencionados neste momento. O primeiro, que não pôde ser concluído no âmbito deste trabalho por falta de tempo hábil, é a definição do módulo de Entrada/Saída. Sua implementação pode envolver a integração com bibliotecas padrão do C, para que haja real Entrada/Saída de dados durante a execução de um código, como por exemplo a leitura de um arquivo ou impressão de mensagem na tela.

Outro trabalho futuro, que depende da integração com o primeiro, é a definição de uma HAL (*Hardware Abstraction Layer*) para abstração das questões físicas do hardware por parte do programador/desenvolvedor.

Aproveitando-se a HAL é possível construir alguns mecanismos de Sistema Operacional para gerenciamento da plataforma, mais notadamente para o gerenciamento de memória, gerenciamento de processos, suporte a interrupções, suporte a E/S, *boot* do sistema, dentre outros.

Do ponto de vista arquitetural, outros trabalhos são possíveis. A integração de outros processadores e gerenciamento de uma plataforma heterogênea, com binários executáveis de diferentes arquiteturas na memória; a adaptação da plataforma para um modelo de Memória Distribuída, com o devido suporte para comunicação e sincronismo entre processos através da implementação de diretivas *Send*/*Receive*; estudo de diferentes mecanismos de coerência de cache, como modelos de coerência fraca de cache e/ou COMA (***C****ache-****O****nly* ***M****emory* ***A****ccess*); integração de módulo reconfigurável e definição para seu acesso, gerenciamento uso e comunicação; estudo sobre um possível funcionamento assíncrono dos módulos e da NoC; estudo sobre o consumo de energia da plataforma e proposta de mecanismos (físicos e lógicos) para sua minimização.

Alguns trabalhos e experimentos que o autor desejou realizar, mas não teve condições incluem: integração de diferentes modelos de NoC e subseqüente avaliação do desempenho; melhoria no módulo de Diretório, para criação de uma fila de requisições de Test-and-Set ao invés de uma espera ocupada; otimizações em geral, especialmente no que tange à diminuição do *overhead* no acesso à memória, possivelmente com a criação de múltiplas filas de acesso à memória no Diretório e divisão do módulo de memória em mini-módulos internos; implementação de prioridade de pacotes na NoC; implementação de mecanismo de previsão de salto no SPARC e busca antecipada na memória; dentre outros.

[WAW 99] R. Wawrzyniak, "Systems-on-a-chip: A brave new world" Semico Research Corporation Report SC101-1-99, Sep. 1999.

[VSI 00] VSI Alliance "Virtual Component Interface Standard – OCB 2 1.0" *VSI Alliance*, 2000.

[AGO 01] Agostini, L.; Silva, I. S.; Bampi, S.; "Pipelined Fast 2-D DCT Architecture for JPEG Image Compression"; *XIV Symposium on Integrated Circuits and System Design, SBCCI 2001*, Pirenópolis, Brazil. pp. 226-231.

[CHO 01] Cho, S.L.; Yang, M.K.; Lee, J.; "Analytical modeling of a fat-tree network with buffered switches**",** *Communications, Computers and signal Processing*, *2001*, pp:184 – 187

[DEA 01] S.P. Dean; "ASICs and Foundries" *Integrated Systems Design*, Mar 2001.

[OCP 01] OCP International Partnership; "Open Core Protocol Specification" Release 1.0; 2001.

[SAN 01] Sangiovanni-Vincentelli, A.; Martin G.; "Platform-based design and software design methodology for embedded systems"; *Design & Test of Computers, IEEE* Volume 18, Issue 6, pp. 23 – 33, Nov.-Dec. 2001.

[BEN 02] L. Benini and G. De Micheli, "Networks on Chips: a New SoC Paradigm"*, IEEE Computer*, , pp. 70-78, Jan. 2002.

[HEL 02] Justin Helmig, "Developing core software technologies for TI's OMAPTM platform," Texas Instruments, 2002. Available at http://www.ti.com

[INT 02] Intel, "Product Brief: Intel IXP2850 Network Processor," 2002, Disponível em http://www.intel.com

[KUM 02] Kumar, S. et al; "A network on chip architecture and design methodology" *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp.105 – 112, 2002.

[ZEF 02] Zeferino, C.A.; et al; "A study on communication issues for systems-on-chip" *15th Symposium on Integrated Circuits and Systems Design.* pp.121 – 126, 2002.

[AZE 03] Azevedo, A.; Soares, R.; Silva, I.S.: "Implementação da DCT 2D EM Arquiteturas Reconfiguráveis Utilizando o X4CP32" *Proceedings of Iberchip 03*, Havana, Cuba, 2003.

[ART 03] Alain Artieri, Viviana D'Alto, Richard Chesson, Mark Hopkins, and Marco C. Rossi, "NomadikTM Open Multimedia Platform for Next-generation Mobile Devices," STMicroelectronics Technical Article TA305, 2003, Disponível em http://www.st.com

[HEN 03] John L. Hennessy, David A. Patterson; "Arquitetura de Computadores – Uma Abordagem Quantitativa"; Terceira edição, *Editora Campus*, 2003.

[JAN 03] Jantsch, A. and Tenhunen, H. (Editors); "Networks on Chip"; *Kluwer Academic Publishers*, 2003.

[SOA 03-1] Soares, R., Silva, I.S., Azevedo, A., "When Reconfigurable Architecture Meets Network-on-Chip", *17th Symposium on Integrated Circuits and Systems Design*, pp. 216 – 221, 2003.

[SOA 03-2] Soares, R., Pereira, A., Silva, I. S. "X4CP32: A New Hybrid ParallelReconfigurable General-Purpose Processor". *Proceedings of the 15th Symposium on Computer Architecture and High Performance - SBAC-PAD*, 2003.

[SOA 03-3] Soares, R.; Azevedo, A.; Silva, I.S.; "X4CP32: a Coarse Grain General Purpose Reconfigurable Microprocessor", *Parallel and Distributed Processing Symposium, 2003*. Proceedings. International, April 22-26, 2003 pp. 171 -178. Nice, France.

[ITR 04] International Technology Roadmap for Semiconductors. Disponível em public.itrs.net

[MAD 04] Madeira, S. "Um Ambiente Baseado em Componentes para Desenvolvimento de Softwares de Sistemas Embutidos", *Dissertação de Mestrado, UFSC*, 2004.

[SOA 04] Soares, R., Pereira, A., Silva, I. S. "A Case-Study of Communication in a Reconfigurable Architecture: The X4CP32's Communication Buffer", *Proceedings of the X Workshop Iberchip*, 2004. Cartagena de Indias, Colombia.

[ARM 05] ARM "AMBA Bus" http://www.arm.com/products/solutions/AMBA_Spec.html

[AMD 05] Amde, M.; et al; "Asynchronous on-chip networks" *Computers and Digital Techniques, IEE Proceedings- Volume 152, Issue 2,* pp:273 – 283. Mar 2005.

[BEN 05] Benini, L.; et al. "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC." *The Journal of VLSI Signal Processing*, 41(2): 169 – 182, September 2005.

[BER 05] Bertozzi, D.; et al; "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip**"** *Parallel and Distributed Systems, IEEE Transactions on Volume 16, Issue 2,* pp:113 – 129. Feb 2005.

[DIE 05] Dielissen, J. et al. "Concepts and Implementation of the Philips Network-on-Chip". Disponível em www.design-reuse.com

[FOR 05] Cynthesizer: http://www.forteds.com/products/cynthesizer.asp

[GCC 05] GNU Compiler Collection: www.gcc.gnu.org

[INT 05] Intel's Pentium 4: http://www.intel.com/products/processor/pentium4/index.htm

[LOG 05] Loghi, M.; Poncino, M.; "Exploring energy/performance tradeoffs in shared memory MPSoCs: snoop-based cache coherence vs. software solutions" *Design, Automation and Test in Europe, Proceedings* pp:508 - 513 Vol. 1. 2005.

[OSI 05] Open SystemC Initiative: www.systemc.org

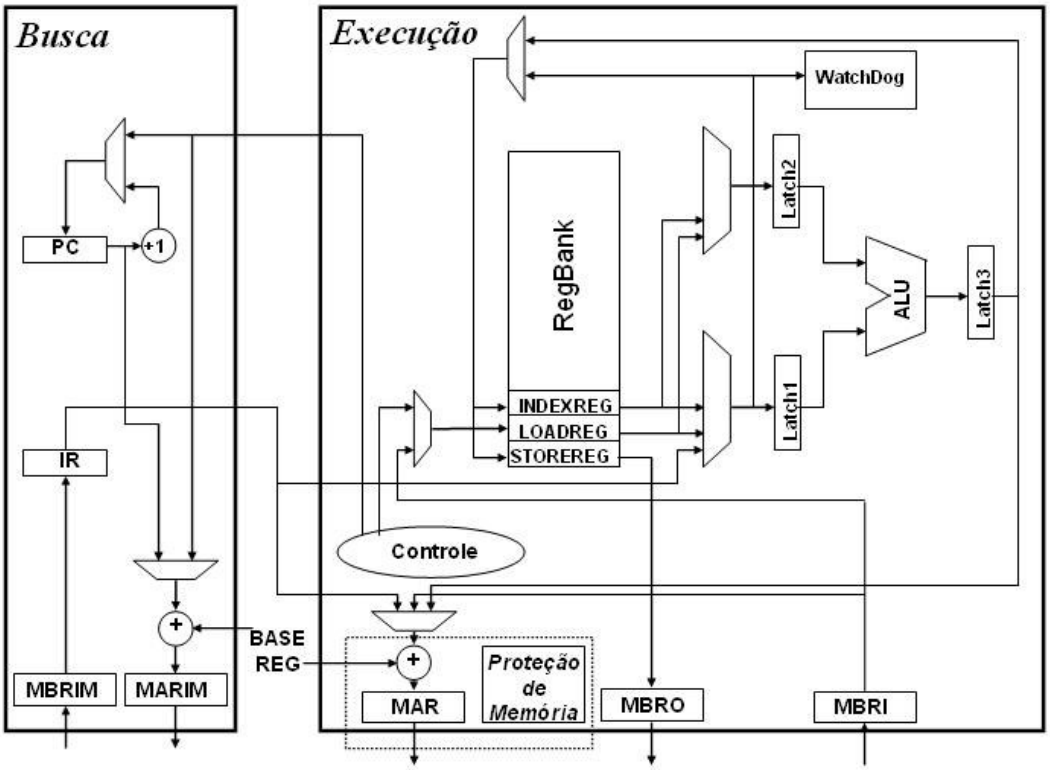
[TAE 05] Taeweon Suh; Daehyun Kim; Lee, H.-H.S.; "Cache coherence support for non-shared bus architecture on heterogeneous MPSoCs" *Proceedings. 42nd Design Automation Conference*, 2005.

ANEXOS

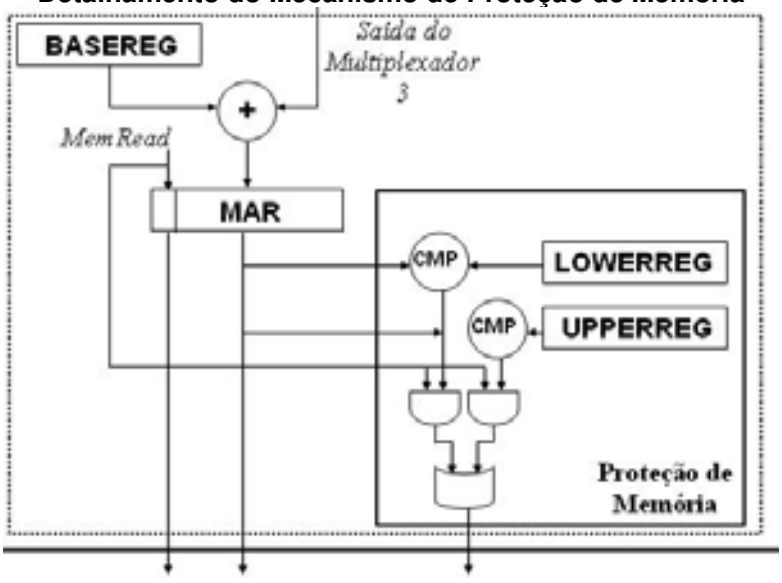
Anexo I: Especificações do Processador GPOP

I.1. Arquitetura

Caminho de Dados



Detalhamento do Mecanismo de Proteção de Memória



I.2. Conjunto de Instruções

OPCODE	FORMATO	SIGNIFICADO
0x00	NOP	Nenhuma operação
0x01	DLOAD M	Load direto
0x02	ILOAD M	Load indireto
0x03	CLOAD C	Load constante
0x04	XLOAD M	Load indexado
0x05	DSTORE M	Store direto
0x06	ISTORE M	Store indireto
0x07	CSTORE C	Store constante
0x08	XSTORE M	Store indexado
0x09	LOADIO M	Load de E/S
0x0A	STORIO M	Store de E/S
0x0F	RADD R, R, R	Soma inteira
0x10	RADDC R, R, R	Soma inteira com carry
0x11	RSUB R, R, R	Subtração inteira
0x12	RSUBC R, R, R	Subtração inteira com carry
0x13	RMUL R, R, R	Multiplicação inteira
0x14	RMULS R, R, R	Multiplicação inteira com sinal
0x15	RDIV R, R, R	Divisão inteira
0x16	RDIVS R, R, R	Divisão inteira com sinal
0x17	INCR R	Incremento
0x18	DECR R	Decremento
0x19	ADJ R	Ajusto BCD-Decimal
0x1A	CMP R, R	Comparação de dois registradores
0x1B	CMPSG R, R, R	Comparação de dois registradores e move o maior
0x1C	CMPEL R, R, R	Comparação de dois registradores e move o menor
0x21	RAND R, R, R	'E' lógico
0x22	ROR R, R, R	'OU' lógico
0x23	RXOR R, R, R	'OU EXCLUSIVO' lógico
0x24	RNOT R, R	'NÃO' lógico
0x29	JUMP M	Salto incondicional
0x2A	JZER M	Salta se zero
0x2B	JNZER M	Salta se diferente de zero
0x2C	JNEG M	Salta de negativo
0x2D	JGREAT M	Salta se maior
0x2E	JLOWER M	Salta de menor
0x2F	JEQUAL M	Salta de igual
0x30	JEGREAT M	Salta se igual ou maior
0x31	JELOWER M	Salta se igual ou menor
0x32	JCARRY M	Salta se existir *carry*
0x33	JOVFLOW M	Salta se existir *overflow*
0x34	CALL M	Chamada de subrotina
0x35	RETN	Retorno de subrotina
0x36	IRET	Retorno de interrupção
0x3B	MOVE R, R	Move conteúdo de registrador
0x3C	MOVEZ R, R	Move conteúdo de registrador se zero
0x3D	MOVENZ R, R	Move conteúdo de registrador se diferente de zero
0x3E	MOVEC R, R	Move conteúdo de registrador se houver *carry*
0x3F	MOVOV R, R	Move conteúdo de registrador se houver *overflow*
0x44	SHARR R	*Shift* aritmético para a direita
0x45	SHLOR R	*Shift* lógico para a direita

0x46	SHLEFT R	*Shift* para a esquerda
0x47	ROTR R	Rotação para a direita
0x48	ROTL R	Rotação para a esquerda
0x4D	SETF F	Seta flag
0x4E	RSTF F	Reseta flag
0x4F	INVF F	Inverte flag
0x54	PUSH M	Empilha
0x55	POP M	Desempilha
0x56	PUSHREG R	Empilha de registrador
0x57	POPREG R	Desempilha para registrador
0x58	PUSHCONT	Empilha contexto
0x59	POPCONT	Desempilha contexto
0x5A	INCRSP	Incrementa registrador SP
0x5B	DECRSP	Decrementa registrador SP
0x5C	RSTSP	Reseta registrador SP
0x61	TSTSET M	Test-and-Set
0x62	TSTRST M	Test-and-Reset
0x63	WRTBACK	*Write-back* da cache de dados
0x64	MOVESYS R, Rs	Transfere dados para registradores de sistema
0x65	CTDOWN R	Define valor de contagem para o *watchdog*
0xFF	HALT	Paralisa o processador

Anexo II: Código-Fonte das Aplicações Simuladas

II.1. Fatorial ASM

Descrição: Calcula o fatorial de um número.

Código C

/ Não possui – código escrito em assembly */*

Código Assembly

```
main:                           ! Inicializa os registradores
    mov     10, %g1             ! %g1 := 10
    sub     %g1, 1, %g2         ! %g2 := %g1 - 1

fat:                            ! Laço que calcula o fatorial
    smul    %g1, %g2, %g1       ! %g1 := %g1 * %g2
    subcc   %g2, 1, %g2         ! %g2 := %g2 - 1
    bne     fat                 ! Salta para fat se %g2 = 0
    nop
```

II.2. Maior ASM

Descrição: Identifica o maior elemento em um vetor de inteiros com 10 posições.

Código C

/ Não possui – código escrito em assembly */*

Código Assembly

```
main:                       ! Inicializa os registradores
        mov 0,%g4           ! maior := 0
        sethi %hi(vetor),%g1
        or %g1,%lo(vetor),%g1 ! %g1 := vetor
        mov 0,%g2           ! ponteiro := 0

loop:                       ! Laço que percorre o vetor
        ld [%g1+%g2],%g3    ! temp := vetor[ponteiro]
        cmp %g3,%g4
        ble eh_menor        ! Salta para eh_menor se (maior > temp)
        nop
        mov %g3,%g4         ! maior := temp

eh_menor:
        add %g2,4,%g2       ! ponteiro := ponteiro + 1
        cmp %g2,40
        bl loop             ! Salta para loop se ainda não chegou ao fim do vetor
        nop
        ret
        restore

vetor:                      ! Vetor a ser percorrido
        .long 6
        .long 10
        .long 3
        .long 4
        .long 10
        .long 1
        .long 6
        .long 15
        .long 9
        .long 8
```

II.3. Maior

Descrição: Identifica o maior elemento em um vetor de inteiros com 10 posições.

Código C

```
static int vetor [10] = {16, 2, 3, 4, 17, 6, 7, 8, 9, 3};       /* Vetor a ser percorrido */

int main () {
        int i, maior=0;
        for (i=0; i<10; i++) {                  /* Laço que percorre o vetor */
                if (vetor[i] > maior)
                        maior = vetor[i];       /* Atualiza valor de maior */
        }
        return maior;
}
```

II.4. Quicksort

Descrição: Ordena os elementos de um vetor em ordem crescente em O(n log n) passos no caso médio e O(n^2) no pior caso.

Código C

```
#define SIZE 64          /* Tamanho do vetor */

int vetor[SIZE];         /* Vetor a ser ordenado */

void quicksort (int *a, int lo, int hi)      /* Função de Quicksort */
{
        int i=lo, j=hi, h;          /* lo e hi são os limites inferior e superior do vetor */
        int x=a[(lo+hi)/2];         /* Escolhe o pivô */

        /* particionamento */
        do {
                while (a[i]<x) i++;
                while (a[j]>x) j--;

                if (i<=j) {
                        h=a[i]; a[i]=a[j]; a[j]=h;
                        i++; j--;
                }
        } while (i<=j);

        /* recursão */
        if (lo<j) quicksort(a, lo, j);
        if (i<hi) quicksort(a, i, hi);
}

int main()
{
        int i;
        for (i=0; i<SIZE; i++) vetor[i] = SIZE-i;          /* Inicializa o vetor */

        quicksort(vetor, 0, SIZE);

        return 0;
}
```

II.5. Mutex

Descrição: Implementa o conceito de *mutex* na linguagem C. Implementa uma função que provê identificador único para cada processador.

Código C

```
/****************************** mutex.c ******************************/

#include "mutex.h"

int mutex1;                             /* Variável usada para exclusão mútua */

int main() {
        down(&mutex1);                  /* Requisição de acesso à região crítica */
        unsigned int id = get_id();     /* Busca identificador único */
        up(&mutex1);                    /* Libera a região crítica */

        return id;                      /* Retorna seu identificador único */
}



/****************************** mutex.h ******************************/

unsigned int current_id;            /* Valor do próximo identificador */

int down(int *s) {                  /* Função de requisição de acesso à região crítica */
        int retorno=-1;             /* Inicializa variável como ocupada */
        do {
                asm ("ldstub    %1,%[output]"       // Implementa o Test-and-Set: retorna
                        : [output] "=r" (retorno)   // valor 0 se o mutex estiver livre e -1
                        : "m" (*s));                // caso o contrário
        } while(retorno);           /* Permanece no laço enquanto o mutex estiver ocupado */

        return retorno;             /* Retorna após conseguir acesso ao mutex */
}

int up(int *s) {                    /* Função que libera o acesso à região crítica */
        int retorno=0;              /* Posiciona o valor a ser escrito na memória como "0" */
        asm ("swap      %1,%[output]"       // Troca atomicamente o valor de retorno com
                : [output] "+r" (retorno)   // o valor que está na memória, posicionando
                : "m" (*s));                // o valor "0"e liberando o mutex
        return retorno;             /* Saída da função */
}

unsigned int get_id() {             /* Função que retorna valor identificador */
        return (current_id)++;      // Retorna o identificador atual e incrementa para a próxima
}                                   // chamada desta função
```

II.6. Multiplicação de Matriz

Descrição: Multiplica duas matrizes paralelamente fazendo uso da biblioteca mutex.h.

Código C

```
#include "mutex.h"

#define SIZE 8                  /* Tamanho das matrizes */
#define PROCESSORS 8            /* Número de processadores do sistema */
#define STEP (SIZE/PROCESSORS)  /* Constante utilizada para paralelização automática */

int mutex1, mutex2;             /* Mutexes */

int M_A[SIZE][SIZE] = {         /* Primeiro operando */
        1,2,3,4,5,6,7,8,
        9,10,11,12,13,14,15,16,
        17,18,19,20,21,22,23,24,
        25,26,27,28,29,30,31,32,
        33,34,35,36,37,38,39,40,
        41,42,43,44,45,46,47,48,
        49,50,51,52,53,54,55,56,
        57,58,59,60,61,62,63,64
};

int M_B[SIZE][SIZE] = {         /* Segundo operando */
        1,2,3,4,5,6,7,8,
        9,10,11,12,13,14,15,16,
        17,18,19,20,21,22,23,24,
        25,26,27,28,29,30,31,32,
        33,34,35,36,37,38,39,40,
        41,42,43,44,45,46,47,48,
        9,50,51,52,53,54,55,56,
        57,58,59,60,61,62,63,64
};

int M_C[SIZE][SIZE];            /* Matriz de resultado */

int main() {
        down(&mutex1);                  /* Aquisição de identificador único */
        unsigned int id=get_id();
        up(&mutex1);

        int i, j, k;                    /* Variáveis controladoras de laço */

        for (i=id*STEP; i<(id+1)*STEP; i++)
                for (j=0; j<SIZE; j++) {
                        int soma=0;
                        for (k=0; k<SIZE; k++)          /* Multiplica e acumula */
                                soma += M_A[i][k]*M_B[k][j];

                        down(&mutex2);          /* Requisição de acesso à região crítica */
                        M_C[i][j] = soma;       /* Escrita do resultado na matriz M_C */
                        up(&mutex2);            /* Saída da região crítica */
```

```
                }
        return id;
}
```

II.7. Mergesort

Descrição: Ordena os elementos de um vetor em ordem crescente em O(n log n) passos no caso médio e no pior caso.

Código C

```
#include "mutex.h"

#define SIZE 64          /* Tamanho do vetor */

int array_A[SIZE];       /* Vetor a ser ordenado */

/* Mescla e ordena duas metades de um vetor */
void merge(int lo, int m, int hi, int array_B[]) {
        int i, j, k;

        i=0; j=lo;


        /* Copia primeira metade do vetor array_A para o vetor array_B
        while (j<=m)
                array_B[i++] = array_A[j++];

        i=0; k=lo;

        /* Copia de volta na ordem do maior elemento */
        while (k<j && j<=hi) {
                if (array_B[i] <= array_A[j])
                        array_A[k++] = array_B[i++];
                else
                        array_A[k++] = array_A[j++];
        }

        /* Copia de volta os elementos restantes da primeira metade */
        while (k<j)
                array_A[k++] = array_B[i++];
}


void mergesort(int lo, int hi, int array_B[]) {
        if (lo<hi) {       /* Divisão do vetor e recursão */
                int m=(lo+hi)/2;
                mergesort(lo, m, array_B);
                mergesort(m+1, hi, array_B);
                merge(lo, m, hi, array_B);
        }
}
```

```
int mutex1, mutex2;      /* Mutexes */

int main()
{
        down(&mutex1);                  /* Aquisição de identificador único */
        unsigned int id=get_id();
        up(&mutex1);

        int i, array_B[SIZE/2];

        switch (id) {
        case 0: {
                for (i=0; i<SIZE/2; i++)                // Inicializa primeira metade do vetor
                        array_A[i] = SIZE-i;            // em ordem decrescente

                mergesort(0, SIZE/2-1, array_B);        /* Ordena primeira metade do vetor */

                down(&mutex2);                          // Espera término do outro processador
                merge(0, SIZE/2-1, SIZE-1, array_B);    // para iniciar processo de mescla final
                up(&mutex2);
                break;
        }

        case 1: {
                down(&mutex2);
                for (i=0; i<SIZE/2; i++)                // Inicializa segunda metade do vetor
                        array_A[i+SIZE/2] = SIZE/2-i;   // em ordem  decrescente

                mergesort(SIZE/2, SIZE-1, array_B);     /* Ordena segunda metade do vetor */

                up(&mutex2);            // Libera o mutex para que o outro processador inicie a
                break;                  // mescla final
        }
        default: break;
        }

        return id;
}
```

II.8. Knuth-Morris-Pratt

Descrição: Busca um padrão *P* de tamanho *n* em uma seqüência *S* de tamanho *l* com complexidade O(*n* + *l*).

Código C

```
#include "mutex.h"

#define PSIZE 16
#define SSIZE 1024
#define PROCESSORS 2

void preKmp(int *x, int m, int kmpNext[]) {      /* Monta a tabela KMP */
        int i, j;
        i = 0;
        j = kmpNext[0] = -1;
        while (i < m) {
                while (j > -1 && x[i] != x[j])
                        j = kmpNext[j];
                        i++;
                        j++;
                        if (x[i] == x[j])
                                kmpNext[i] = kmpNext[j];
                        else
                                kmpNext[i] = j;
        }
}

int KMP(int *x, int m, int *y, int n) {
        int i, j, kmpNext[PSIZE];

        preKmp(x, m, kmpNext);          /* Pré-processamento – configuração da tabela KMP */

        i = j = 0;
        while (j < n) {                         /* Laço de busca */
                while (i > -1 && x[i] != y[j])
                        i = kmpNext[i];
                        i++;
                        j++;
                        if (i >= m)
                                return (j-i);       /* Retorna a posição inicial do padrão */
        }
}

int array_P[2][PSIZE] = {
        7,9,6,7,1,7,9,8,0,2,3,2,8,8,5,5,
        3,3,2,1,5,8,4,0,0,9,2,6,8,9,2,0
};

int array_S[SSIZE] = {
        9,4,7,1,6,4,2,9,6,5,1,6,0,2,6,2,5,3,5,8,5,4,5,6,4,0,7,0,5,8,8,4,
        4,7,5,0,3,8,0,1,3,3,8,3,5,6,7,0,9,5,0,4,1,7,0,7,8,8,7,3,6,8,9,0,
```

```
        7,6,3,1,4,5,2,9,8,2,5,3,8,4,6,8,9,6,4,3,6,5,0,6,5,8,1,3,8,0,5,5,
        6,8,6,3,3,1,4,3,3,9,7,2,6,5,0,7,3,6,0,9,1,1,5,8,1,6,1,9,8,7,4,5,
        7,3,0,3,4,4,6,7,4,5,1,2,0,1,9,4,0,0,3,1,1,1,2,4,9,3,3,8,2,9,5,0,
        4,5,3,8,9,9,8,5,5,9,7,7,3,9,1,3,9,7,6,2,8,8,8,9,4,1,7,6,2,4,6,7,
        9,1,7,1,1,7,8,8,9,8,5,4,7,9,9,8,6,5,2,6,6,0,5,2,3,5,8,7,9,7,6,1,
        8,6,4,1,3,4,9,4,2,7,8,1,6,7,1,2,5,5,0,1,5,5,5,0,2,5,0,4,2,6,7,3,
        2,3,6,8,7,6,2,2,5,1,5,1,8,7,5,5,2,7,6,0,4,3,2,7,9,2,3,3,9,0,6,3,
        3,3,1,2,1,6,4,8,7,0,1,7,7,8,3,1,5,1,3,9,5,6,8,6,0,1,9,9,3,8,5,6,
        3,8,1,4,4,5,4,3,7,5,3,4,3,6,8,8,9,1,9,6,9,8,4,0,9,4,1,5,2,6,3,5,
        7,4,1,1,0,5,7,7,0,0,4,5,8,4,3,7,7,2,6,7,2,0,9,4,6,0,9,0,9,2,7,6,
        7,8,9,9,5,6,6,7,6,2,2,6,6,7,4,4,2,2,3,4,4,2,8,1,4,7,1,3,2,1,1,9,
        9,3,8,7,9,6,6,8,1,1,4,7,8,0,3,2,4,8,7,9,0,7,2,7,7,5,2,9,6,4,0,8,
        7,0,5,8,6,3,8,9,4,3,9,5,5,2,7,0,3,4,1,5,4,3,2,1,8,5,2,7,1,2,5,0,
        4,2,0,0,5,9,0,2,4,9,7,9,3,4,9,8,1,0,4,5,5,6,8,6,3,2,3,4,6,0,6,0,
        2,7,2,9,6,4,3,0,5,0,9,1,5,1,9,8,3,3,5,9,2,5,7,5,7,0,2,5,0,8,5,4,
        7,7,5,3,4,9,3,9,9,5,0,6,8,2,4,1,5,1,2,9,6,9,7,3,1,9,8,1,9,5,7,7,
        5,3,2,9,4,6,0,5,3,3,2,1,5,8,4,0,0,9,2,6,8,9,2,0,8,2,3,9,8,3,8,3,
        8,1,4,2,9,4,7,2,7,1,5,4,0,1,7,2,0,9,8,9,8,2,1,8,5,6,7,3,9,8,8,7,
        9,4,9,8,8,9,2,8,0,7,2,0,0,9,4,1,8,3,2,8,5,5,6,2,1,6,5,1,4,5,8,3,
        9,0,3,0,9,5,8,1,4,2,4,4,4,8,5,4,1,9,3,9,4,9,1,6,7,9,9,1,6,9,6,6,
        9,9,8,0,6,8,2,0,0,6,5,6,4,2,1,8,4,4,7,8,5,0,6,3,1,5,6,8,5,5,6,6,
        6,6,7,3,4,9,5,6,7,0,3,1,5,4,9,9,8,8,9,5,1,8,8,2,3,7,2,0,2,8,7,0,
        4,4,3,0,5,9,7,2,1,0,5,8,6,7,9,6,5,1,1,6,9,2,1,2,9,3,5,3,4,2,3,0,
        8,9,1,3,8,8,7,1,0,4,2,6,1,1,4,9,2,5,7,3,9,8,6,8,2,1,3,6,5,7,6,3,
        6,9,8,6,7,7,9,9,1,1,7,5,5,1,4,9,9,1,3,8,0,1,9,4,4,2,0,9,9,8,2,7,
        0,2,5,9,9,5,9,2,8,6,7,5,0,1,5,1,5,8,9,5,1,8,9,5,1,1,4,2,1,8,2,1,
        2,7,1,3,4,2,5,3,0,5,8,0,8,5,1,3,5,3,8,6,3,9,1,4,2,7,9,6,7,1,7,9,
        8,0,2,3,2,8,8,5,5,8,5,3,4,9,7,1,4,7,0,7,9,1,4,1,1,3,7,8,4,7,8,4,
        7,0,7,2,0,7,7,5,6,4,1,2,5,8,3,1,7,3,9,6,7,3,8,0,6,7,8,2,4,8,6,4,
        1,6,8,1,3,7,9,1,3,0,3,1,0,9,2,7,4,1,4,3,6,4,3,4,3,4,6,0,2,5,6,3
};

int mutex1,mutex2;

int pos[PROCESSORS];

int main() {
        down(&mutex1);
        unsigned int id=get_id();
        up(&mutex1);

        int var = KMP(array_P[id], PSIZE, array_S, SSIZE);

        down(&mutex2);
        pos[id] = var;
        up(&mutex2);

        return id;
}
```

II.9. Estimação de Movimento

Descrição: Implementação paralela do algoritmo de busca completa (*full search*) de um bloco em um espaço de busca para geração de vetores de movimento (*motion vector*).

Código C

```
#include "mutex.h"

#define BLOCK 4         /* Tamanho do bloco */
#define SIZE 8          /* Tamanho do espaço de busca */

int mutex1, mutex2;
int pos[2];             /* Posição do bloco no espaço de busca */

int search_block[BLOCK][BLOCK] = {      /* Bloco a ser encontrado */
        19, 14, 6, 14,
        6, 16, 6, 15,
        8, 10, 18, 4,
        11, 20, 32, 31
};

int image[SIZE][SIZE] = {               /* Imagem (Espaço de Busca) */
        31, 26, 28, 32, 36, 33, 37, 37,
        21, 23, 16, 17, 16, 9, 9, 17,
        23, 16, 12, 2, 9, 25, 32, 27,
        12, 19, 27, 32, 16, 8, 5, 12,
        2, 17, 21, 34, 19, 14, 6, 14,
        0, 4, 0, 2, 6, 16, 6, 15,
        2, 6, 0, 1, 8, 10, 18, 4,
        0, 9, 12, 26, 11, 20, 32, 31
};

/* Função de busca completa */
matching(int lower_x, int upper_x, int lower_y, int upper_y) {
        int i, j, k, l;
        int found;

        for (i=lower_x; i<upper_x; i++)         // Laço que percorre o espaço de busca
                for (j=lower_y; j<upper_y; j++) {       // do processador
                        down(&mutex2);
                        if (pos[0] != -1) {             /* Testa se o bloco já foi encontrado */
                                up(&mutex2);            /* Em caso positivo pára a busca */
                                return;
                        }
                        up(&mutex2);

                        found = 1;
                        for (k=i; k<i+BLOCK; k++) {             // Procura o bloco enquanto
                                for (l=j; l<j+BLOCK; l++) {     // ocorre match
                                        if (search_block[k-i][l-j] != image[k][l]) {
                                                found = 0;      /* Bloco não encontrado */
                                                break;
```

```
                                        }
                                }
                                if (!found) break;
                        }
                        if (found) {            /* Salva a posição do bloco caso tenha encontrado */
                                down(&mutex2);
                                pos[0] = i;
                                pos[1] = j;
                                up(&mutex2);
                                return;         /* Pára a busca */
                        }
                }
}


int main() {
        down(&mutex1);
        unsigned int id = get_id();
        if (id == 0) {                  /* O processador de ID 0 inicializa a variável pos*/
                pos[0] = -1;
                pos[1] = -1;
        }
        up(&mutex1);

        int lower_x, upper_x, lower_y, upper_y;

        switch (id) {   /* Define a divisão do espaço de busca para cada processador */
        case 0: {
                lower_x = 0;
                upper_x = SIZE/2;
                lower_y = 0;
                upper_y = SIZE;
                break;
        }
        case 1: {
                lower_x = SIZE/2;
                upper_x = SIZE;
                lower_y = 0;
                upper_y = SIZE;
                break;
        }
        default: break;
        }

        matching(lower_x, upper_x, lower_y, upper_y);       /* Chama a função de busca */
        return id;
}
```

II.10. Transformada Discreta do Cosseno (DCT) 2D

Descrição: Implementação paralela da DCT 2D, operação básica em aplicações multimídia.

Código C

```
#include "mutex.h"

#define BLOCK 8  /* Tamanho do bloco */
#define SIZE 16     /* Tamanho da imagem a ser processada */

int mutex1, mutex2;
int id1_ok;            /* Implementa o handshake entre processos */

int input[SIZE][SIZE] = {   /* Imagem a ser processada */
     209, 204, 196, 191, 183, 188, 175, 164, 162, 153, 165, 163, 149, 153, 143, 147,
     197, 199, 185, 174, 175, 164, 171, 159, 145, 151, 165, 163, 177, 177, 177, 177,
     208, 216, 227, 215, 213, 227, 220, 232, 222, 211, 205, 197, 207, 208, 220, 234,
     195, 209, 196, 200, 187, 197, 206, 195, 195, 202, 206, 205, 196, 184, 199, 186,
     208, 218, 209, 204, 212, 212, 203, 217, 222, 224, 214, 213, 200, 186, 183, 173,
     192, 192, 203, 190, 185, 189, 182, 193, 188, 201, 195, 197, 182, 175, 180, 194,
     179, 191, 184, 196, 192, 193, 203, 189, 176, 161, 162, 152, 139, 154, 150, 138,
     178, 169, 160, 170, 181, 181, 187, 177, 174, 158, 167, 166, 160, 158, 155, 152,
     173, 180, 173, 180, 172, 160, 169, 164, 153, 148, 149, 141, 135, 148, 144, 153,
     162, 166, 170, 154, 142, 136, 127, 129, 123, 107, 92, 96, 94, 92, 78, 87,
     151, 146, 147, 144, 143, 137, 145, 149, 154, 147, 160, 175, 166, 159, 168, 164,
     164, 161, 157, 143, 150, 153, 157, 142, 145, 150, 156, 142, 129, 137, 148, 141,
     151, 147, 153, 156, 163, 162, 153, 149, 157, 145, 141, 156, 154, 143, 138, 133,
     153, 161, 158, 151, 146, 131, 125, 140, 146, 130, 115, 125, 133, 145, 132, 144,
     146, 155, 154, 138, 130, 136, 132, 117, 128, 136, 120, 113, 126, 122, 126, 125,
     134, 120, 128, 127, 114, 101, 99, 109, 96, 95, 99, 111, 107, 114, 122, 127,
};

int buffer[BLOCK][BLOCK];       /* Buffer intermediário entre P0 e P1 */
int output[SIZE][SIZE];         /* Saída: imagem processada */


/* Constantes utilizadas na DCT multiplicadas por 256 */
const int m1=181;  /* m1 = cos(4*PI/16) */
const int m2=98;   /* m2 = cos(6*PI/16) */
const int m3=139;  /* m3 = cos(2*PI/16) - cos(6*PI/16) */
const int m4=334;  /* m4 = cos(2*PI/16) + cos(6*PI/16) */

/* Funcão que implementa a DCT 1D */
void DCT (int in[BLOCK][BLOCK], int out[BLOCK][BLOCK]) {
     int b[BLOCK], c[BLOCK], d[BLOCK+1], e[BLOCK+1], f[BLOCK];
     int k, i;

     for (k=0; k<BLOCK; k++) {     /* Executa os 6 passos do algoritmo para cada linha */
               /* Passo 1 */
               b[0] = in[k][0] + in[k][7];
               b[1] = in[k][1] + in[k][6];
               b[2] = in[k][3] - in[k][4];
               b[3] = in[k][1] - in[k][6];
```

```
            b[4] = in[k][2] + in[k][5];
            b[5] = in[k][3] + in[k][4];
            b[6] = in[k][2] - in[k][5];
            b[7] = in[k][0] - in[k][7];

            /* Passo 2 */
            c[0] = b[0] + b[5];
            c[1] = b[1] - b[4];
            c[2] = b[2] + b[6];
            c[3] = b[1] + b[4];
            c[4] = b[0] - b[5];
            c[5] = b[3] + b[7];
            c[6] = b[3] + b[6];

            /* Passo 3 */
            out[k][0] = (c[0] + c[3]);
            out[k][4] = (c[0] - c[3]);
            d[3] = c[1] + c[4];
            d[4] = c[2] - c[5];

            /* Passo 4 */
            e[2] = m3 * c[2];
            e[3] = m1 * c[6];
            e[4] = m4 * c[5];
            e[6] = m1 * d[3];
            e[7] = m2 * d[4];

            /* Passo 5 */
            out[k][2] = (c[4]*256 + e[6])/256;
            out[k][6] = (c[4]*256 - e[6])/256;
            f[4] = e[3] + b[7]*256;
            f[5] = b[7]*256 - e[3];
            f[6] = e[2] + e[7];
            f[7] = e[4] + e[7];

            /* Passo 6 */
            out[k][1] = (f[4] + f[7])/256;
            out[k][3] = (f[5] - f[6])/256;
            out[k][5] = (f[5] + f[6])/256;
            out[k][7] = (f[4] - f[7])/256;
        }
}

int main () {
    down(&mutex1);
    unsigned int id = get_id();
    up(&mutex1);            /* Adquire o identificador único */

    int i, j, k, l;
    int temp_input[BLOCK][BLOCK], temp_output[BLOCK][BLOCK];

    switch (id) {
    case 0: {
        id1_ok = 0;
        for (i=0; i<SIZE; i+=BLOCK)        /* Percorre todos os blocos da imagem */
```

```
            for (j=0; j<SIZE; j+=BLOCK) {

                /* Copia bloco da entrada para matriz local */
                for (k=0; k<BLOCK; k++)
                    for (l=0; l<BLOCK; l++)
                        temp_input[k] [l] = input[i+k][j+l];

                /* Executa função DCT no bloco */
                DCT(temp_input, temp_output);

                int ok = 0;
                while (!ok) {     /* Espera pelo processador de ID 1 */
                    down(&mutex2);
                    if (!id1_ok) {
                        /* Copia e transpõe matriz local para buffer intermediário */
                        for (k=0; k<BLOCK; k++)
                            for (l=0; l<BLOCK; l++)
                                buffer[k][l] = temp_output[l][k];

                        id1_ok = 1;
                        ok = 1;
                    }
                up(&mutex2);
                }
            }
        break;
}

case 1: {
    for (i=0; i<SIZE; i+=BLOCK)        /* Percorre todos os blocos da imagem */
            for (j=0; j<SIZE; j+=BLOCK) {

                int ok = 0;
                while (!ok) {
                    down(&mutex2);
                    if (id1_ok) {     /* Espera pelo processador de ID 0 */
                        /* Copia buffer intermediário para matriz local */
                        for (k=0; k<BLOCK; k++)
                            for (l=0; l<BLOCK; l++)
                                temp_input[k][l] = buffer[k][l];

                        id1_ok = 0;
                        ok = 1;
                    }
                up(&mutex2);
            }

            /* Executa segunda função DCT no bloco */
            DCT(temp_input, temp_output);

            /* Salva resultado da segunda DCT na saída */
            for (k=0; k<BLOCK; k++)
                for (l=0; l<BLOCK; l++)
                    output[i+k][j+l] = temp_output[k][l];
            }
```

```
                break;
            }
            default: break;
        }
        return id;
    }
```

Anexo III: Detalhamento da implementação em SystemC

A implementação SystemC de STORM possui os seguintes arquivos:

- *main.cpp:* Contém a função *main()*. Realiza a simulação propriamente dita.
- *specs.h:* Contém as constantes utilizadas em todos os arquivos, exceto os do SPARC.
- *sparc.h/ sparc.cpp:* Implementação do processador SPARC. Conecta todos os 5 estágios.
- *sparc_fetch.h/ sparc_fetch.cpp:* Implementação do estágio de busca do SPARC.
- *sparc_decode.h/ sparc_decode.cpp:* Implementação do estágio de decodificação do SPARC.
- *sparc_execute.h/ sparc_execute.cpp:* Implementação do estágio de execução do SPARC.
- *sparc_memory.h/ sparc_memory.cpp:* Implementação do estágio de memória do SPARC.
- *sparc_write.h/ sparc_write.cpp:* Implementação do estágio de escrita do SPARC.
- *tipos.h/ tipos.cpp:* Implementação das classes utilizadas no SPARC.
- *funcoes.cpp:* Implementação das instruções do SPARC, utilizado pelo estágio de execução.
- *definicao.h/ definicao.cpp:* Definição dos *OpCodes* do SPARC.
- *Cache.h/ Cache.cpp:* Implementação da Cache integrada ao processador.
- *DCache.h/ DCache.cpp:* Implementação da Cache de Dados.
- *ICache.h/ ICache.cpp:* Implementação da Cache de Instruções.
- *CaCoMa.h/ CaCoMa.cpp:* Implementação do *CaCoMa* (***Ca****che* ***Co****mmunication* ***Ma****nager*).
- *block_structure.h:* Implementação da estrutura de blocos, usada na Cache.
- *Memory.h/ Memory.cpp:* Implementação do módulo de memória.
- *Directory.h/ Directory.cpp:* Implementação do módulo de Diretório.
- *Sending_Buffer.h/ Sending_Buffer.cpp:* Implementação do *Sending Buffer* (Buffer de Envio), utilizado no *CaCoMa* e no Diretório.
- *Receiving_Buffer.h/ Receiving_Buffer.cpp:* Implementação do *Receiving Buffer* (Buffer de Recebimento), utilizado no *CaCoMa* e no Diretório.
- *NoC.h/ NoC.cpp:* Implementação em baixo nível do roteador da NoCX4.
- *Crossbar.h/ Crossbar.cpp:* Implementação do mecanismo de *crossbar* do roteador da NoCX4.
- *Mux_NoC.h/ Mux_NoC.cpp:* Implementação do multiplexador utilizado nas portas do roteador da NoCX4.
- *NoC_Arbiter.h/ NoC_Arbiter.cpp:* Implementação do módulo árbitro utilizado nas portas do roteador da NoCX4.

- *Fifo.h/ Fifo.cpp:* Implementação da FIFO, responsável pela bufferização de pacotes no roteador da NoCX4.
- *MeshRouter.h/ MeshRouter.cpp:* Implementação em alto nível do roteador da NoCX4.
- *LeafRouter.h/ LeafRouter.cpp:* Implementação em alto nível do *Leaf Router* (Roteador Folha) da Árvore Obesa.
- *IntermediaryRouter.h/ IntermediaryRouter.cpp:* Implementação do *Intermediary Router* (Roteador Intermediário) da Árvore Obesa.
- *Buffer.h:* Implementação da FIFO, responsável pela bufferização de pacotes nos roteadores em alto nível.
- *Ports.h:* Implementação das portas utilizadas nos roteadores em alto nível. Realiza arbitragem e controle de fluxo.

III.1. main.cpp

Funcionalidade: Lê os arquivos *system.dat* (que contém os parâmetros da instância a ser usada na simulação) e o arquivo *.bin*, gerado pelo Montador, que contém o binário do programa a ser simulado. A partir das informações lidas, instancia os módulos a serem usados para aquela simulação, conecta seus sinais. Carrega o programa binário na memória. Por fim, realiza a simulação. Sua duração pode ser definida no arquivo *system.dat*, ou esta pode ocorrer enquanto os processadores não terminarem a execução de seu programa.

III.2. specs.h

Funcionalidade: Este arquivo deve ser incluído por todos os módulos de STORM. Ele possui a definição dos valores de todas as constantes utilizadas na implementação SystemC, excetuando-se aqueles do SPARC. Estas definições estão divididas em categorias, como “Geral”, “ObTree” e “Cache”. A mudança nestes valores permite uma rápida exploração de espaço de projeto em alguns aspectos.

III.3. sparc.h/ sparc.cpp

Funcionalidade: Implementação do módulo SPARC. Este módulo engloba todos os estágios de pipeline, interligando-os por meio de sinais. Provê também a interface de comunicação do

SPARC com a Cache (portas). Este arquivo, assim como todos relacionados ao SPARC, é de autoria de José Diego Saraiva da Silva.

III.4. sparc_fetch.h/ sparc_fetch.cpp

Funcionalidade: Implementação do estágio de busca do SPARC. Seu funcionamento consiste basicamente em um laço no qual é feito um acesso à Cache de Instruções para a busca da próxima instrução a ser executada. A posição acessada é dada pelo registrador PC (*Program Counter*). Os registradores PC e NPC (*Next Program Counter*) são incrementados em 4 a menos que ocorra um salto. O valor recebido da Cache de Instruções (a instrução a ser executada) é passada para o estágio *Decode*, assim que o mesmo estiver pronto.

III.5. sparc_decode.h/ sparc_decode.cpp

Funcionalidade: Implementação do estágio de decodificação do SPARC. Este estágio recebe a instrução do estágio *Fetch* e identifica seu conteúdo. Caso a instrução seja de salto (*branch*) ou chamada de subrotina (*call*), ela é imediatamente executada. Se o seu resultado for modificar o fluxo de programa, seu resultado é enviado para o *Fetch*. Caso a instrução deva ser enviada para o *Execute*, seus operandos são identificados. Se algum dos registradores usados na instrução estiver em uso (dependência de dados) o *Decode* paralisa sua execução até que o mesmo seja destravado (o que ocorre no estágio *Write*). O *Decode* também atrasa sua execução caso o estado Execute não esteja pronto (em *Stall* devido ao atraso do estágio *Memory*). Ao final, a instrução e seus operandos são passados para o *Execute*, e o *Decode* recebe uma nova instrução do *Fetch*.

III.6. sparc_execute.h/ sparc_execute.cpp

Funcionalidade: Implementação do estágio de execução do SPARC. Este é um estágio complexo, pois é aonde a maioria das instruções são realmente executadas. O *Execute* recebe a instrução a ser executada e seus parâmetros do *Decode*. Todas as instruções são implementadas na forma de funções, chamadas pelo Execute. Elas estão no arquivo *funcoes.cpp*. Em algumas instruções (como as de acesso à memória), os parâmetros são calculados e passados para o *Memory*. Caso a instrução envolva a escrita em registradores, este é marcado como "em edição", e seu novo valor é salvo no banco de registradores de rascunho, para os acessos

subseqüentes àquele registrador. O valor do registrador só é atualizado no banco de registradores oficial no *Write*. As *flags* resultantes da operação são enviadas para o *Decode*, para execução de possíveis instruções de salto subseqüentes. O *Execute* também pode ter sua execução paralisada pelo *Memory*. Quando isto ocorre, o sinal de *Stall* é enviado para o *Decode*, para que ele também paralise sua execução.

III.7. sparc_memory.h/ sparc_memory.cpp

Funcionalidade: Implementação do estágio de memória do SPARC. Todos os acessos à memória, e consequentemente a comunicação com a Cache de Dados, são realizados neste estágio. O *Memory* recebe a instrução a ser executada do *Execute*. Caso a instrução seja de acesso à memória, ela é executada. O endereço de acesso, já calculado pelo *Execute*, é posicionado para a *DCache*, assim como o dado a ser escrito, caso haja um. As instruções *LDSTUB* e *SWAP* (utilizadas na implementação de *mutex*) também são executadas neste estágio. O *Memory* espera a confirmação da operação por parte da *DCache* antes de prosseguir sua execução. Como o tempo de acesso à memória é bastante variável, esta espera pode ser curta ou muito longa. Enquanto o *Memory* não recebe a confirmação, um sinal de *Stall* é enviado para o *Execute*, pois o *Memory* ainda não pode receber uma nova instrução. Este sinal é repassado para os outros estágios, até que o processador fica completamente travado à espera da resposta da *DCache*. Quando a resposta é recebida, o sinal de *Stall* é desativado e o processador continua sua execução. O resultado da execução do *Memory* é enviado para o *Write*.

III.8. sparc_write.h/ sparc_write.cpp

Funcionalidade: Implementação do estágio de escrita do SPARC. Este é o único estágio que escreve no banco de registradores. Ao final da execução de uma instrução, seu resultado (se houver algum) é salvo pelo *Write*. A escrita definitiva no banco de registradores pode marcá-lo como "livre de edição" (caso tenha sido modificado no *Execute*), ou como "destravado" (caso tenha sido modificado pelo *Memory*). Após a passagem pelo *Write* a instrução termina sua execução.

III.9. tipos.h/ tipos.cpp

Funcionalidade: Implementação de classes utilizadas no SPARC. Dentre elas estão interface de estágios, banco de registradores, banco de registradores de busca, banco de registradores de rascunho, mecanismo de travamento de registradores e coleta de resultados.

III.10. funcoes.cpp

Funcionalidade: Implementação das funções de execução de instruções do SPARC, utilizadas no *Execute*. Todo o funcionamento das instruções lógicas e aritméticas, cálculo de endereço nas instruções de acesso à memória e geração de *flags* da ULA, dentre outros, são implementados neste arquivo.

III.11. definicao.h/ definicao.cpp

Funcionalidade: Definição de todos os OpCodes (*Operation Codes*) das instruções do SPARC, divididos por formato de instrução.

III.12. Cache.h/ Cache.cpp

Funcionalidade: Implementação do módulo de Cache integrado ao processador. Inclui três módulos internos: A Cache de Instruções (ICache), Cache de Dados (DCache) e o CaCoMa, interligando-os por meio de sinais. Provê interface para comunicação com o processador e com a NoC.

III.13. DCache.h/ DCache.cpp

Funcionalidade: Implementação da Cache de Dados. Possui características configuráveis pelo usuário, como tamanho, associatividade e política de substituição de blocos. A *DCache* possui interface de comunicação com o processador e com o *CaCoMa*, para o qual pode fazer requisições de blocos no caso de um *miss*.

III.14. ICache.h/ ICache.cpp

Funcionalidade: Implementação da Cache de Instruções. Assim com a *DCache*, possui características configuráveis pelo usuário, que podem ser as mesmas da *DCache* ou não. Estas características são constantes no arquivo *specs.h*. Ao contrário da *DCache*, a *ICache* não permite operações de escrita, somente de leitura. Portanto, sua implementação é bem mais simples. A *ICache* também possui interface para comunicação com o processador e com o *CaCoMa*.

III.15. CaCoMa.h/ CaCoMa.cpp

Funcionalidade: Implementação do *CaCoMa* (***Ca****che* ***Co****mmuncation* ***Ma****nager*). O *CaCoMa* possui interface de comunicação com o processador (para *Test-and-Set*), com a *DCache*, a *ICache* e a *NoC*. O *CaCoMa* recebe requisições destes módulos, podendo também receber requisições do Diretório para *Write-Back* ou invalidação de algum bloco, tratando-os com a seguinte prioridade: invalidação de bloco, *Write-Back* de bloco, leitura da *DCache*, leitura da *ICache*, escrita da *DCache*, substituição de bloco da *DCache*, substituição de bloco da *ICache*, *Test-and-Set*.

Cada uma destas requisições resulta na montagem e envio de um pacote, de tamanho que varia de acordo com a requisição específica, para o Diretório, exceto nos casos de invalidação e *Write-Back* de bloco, o que resulta em uma comunicação com a *DCache*. Nos casos em que há geração e envio de pacotes, normalmente existe algum tipo de resposta por parte do Diretório (exceto no caso de substituição de bloco). Após o recebimento do pacote advindo do Diretório, o *CaCoMa* volta para seu estado *IDLE* e espera uma nova requisição. Para poder realizar a comunicação com a NoC, o CaCoMa conta com um módulo *Sending Buffer* e um *Receiving Buffer*.

III.16. block_structure.h

Funcionalidade: Implementação da estrutura de blocos da Cache. A memória contida nas caches (*ICache* e *DCache*) são compostas por um vetor de objetos desta classe. A classe contém um vetor de *sc_int<WORD>* do tamanho do bloco, que pode ser mudado no arquivo *specs.h* (constante *BLOCK_SIZE*). Além disso, contém também a *tag* do bloco, um bit de válido, um bit de permissão e métodos para facilitar a utilização da classe e depuração, como teste de endereço e impressão do bloco.

III.17. Memory.h/ Memory.cpp

Funcionalidade: Implementação do módulo de memória. A Memória possui apenas interface de comunicação com o Diretório, pois se comunica exclusivamente ele. Possui um vetor de inteiros onde são armazenados os dados, e é capaz de realizar operação de leitura (de posição e de bloco) e escrita.

III.18. Directory.h/ Directory.cpp

Funcionalidade: Implementação do módulo de Diretório. O Diretório possui interface de comunicação com a Memória e com a NoC, sendo que esta última depende dos módulos *Sending Buffer* e *Receiving Buffer*, a exemplo do *CaCoMa*. O Diretório recebe diversos tipos de requisições, podendo vir de alguns ou de todos os processadores do sistema. Ele possui um vetor de inteiros PTA, com informação sobre o endereço físico dos processadores do sistema, para poder montar devidamente os pacotes de resposta às requisições. Possui também uma matriz do tipo *bool*, a STA, que contém informações sobre todos os blocos de sua memória.

O Diretório, em seu estado *IDLE*, espera a chegada de alguma requisição. Estas requisições podem ser dos seguintes tipos: substituição de bloco com *Write-Back* (um bloco que estava *Dirty*); substituição de bloco sem Write-Back (um bloco que estava *Clean*); leitura (*Read Miss*); escrita (*Read Miss* ou *Read Hit Without Permission*) e *Test-and-Set*. Após a devida invalidação de cópias (no caso de requisição de escrita) ou geração de uma operação de *Write-Back* (no caso de leitura ou escrita de um bloco *Dirty*), o Diretório atende à requisição, enviando um pacote para o processador requisitante e volta ao estado de *IDLE* para esperar pela próxima requisição.

III.19. Sending_Buffer.h/ Sending_Buffer.cpp

Funcionalidade: Implementação do *Sending Buffer* (Buffer de Envio), utilizado no CaCoMa e no Diretório para realizar o envio de dados pela NoC. O *Sending Buffer* recebe pacotes montados pelo módulo acoplado e os envia pela NoC, seguindo o seu protocolo de comunicação. O *Sending Buffer* utiliza um buffer circular para armazenamento de dados, controlado pelos apontadores *queue_head* e *queue_tail*.

III.20. Receiving_Buffer.h/ Receiving_Buffer.cpp

Funcionalidade: Implementação do *Receiving Buffer* (Buffer de Recebimento), utilizado no CaCoMa e no Diretório para receber dados pela NoC. O *Receiving Buffer* recebe pacotes da NoC e avisa ao seu módulo controlador quando possui algum pacote válido (pela porta *Has_Data*). Assim como o *Sending Buffer*, o *Receiving Buffer* também utiliza um buffer circular para armazenamento de dados, controlado pelos apontadores *queue_head* e *queue_tail*. Os pacotes podem ser acessados por operações de *pop* dos dados que estão no topo da fila, ou pela busca de um pacote pelo seu módulo de origem.

III.21. NoC.h/ NoC.cpp

Funcionalidade: Implementação em baixo nível do roteador da NoCX4. O roteador possui os seguintes módulos internos: *crossbar*, multiplexador, árbitro e FIFO. Exceto pelo *crossbar*, todos os outros módulos são replicados para cada porta do roteador. Por se tratar de uma topologia em grelha, o roteador possui 5 portas de comunicação: Norte, Sul, Leste, Oeste e Módulo.

III.22. Crossbar.h/ Crossbar.cpp

Funcionalidade: Implementação do mecanismo de *crossbar* do roteador da NoCX4. O C*rossbar* interliga todas as portas entre si, exceto uma porta a si mesma, pois a NoCX4 não permite o roteamento de um pacote para a porta de origem. O multiplexador (*Mux_NoC*) e árbitro (*NoC_Arbiter*) estão contidos no módulo *Crossbar*.

III.23. Mux_NoC.h/ Mux_NoC.cpp

Funcionalidade: Implementação do multiplexador utilizado nas portas do roteador da NoCX4. O muliplexador possui 4 portas de entrada e uma de saída, estando a porta de saída ligada à porta de saída do roteador, e suas quatro portas de entrada às portas de saída das FIFOs de entrada do roteador. O multiplexador recebe do seu árbitro o sinal que indica qual porta deve ser selecionada para saída.

III.24. NoC_Arbiter.h/ NoC_Arbiter.cpp

Funcionalidade: Implementação do módulo árbitro utilizado nas portas do roteador da NoCX4. O árbitro recebe requisições das FIFOs que tiverem seu pacote roteado para a sua porta. Uma das FIFOs é escolhida, de acordo com um simples *round-robin*, no qual a prioridade de comunicação é atribuída de forma cíclica. Se o pacote a ser transmitido não couber na FIFO do roteador vizinho, uma outra FIFO é escolhida. Após o término da transmissão de um pacote, o árbitro novamente verifica as requisições e escolhe uma FIFO para transmissão.

III.25. Fifo.h/ Fifo.cpp

Funcionalidade: Implementação da FIFO, responsável pela bufferização de pacotes no roteador da NoCX4. Além disso, a FIFO também realiza o roteamento de pacotes. Atualmente o único roteamento implementado é o dimensional. A FIFO recebe pacotes de um roteador vizinho, calcula a porta pela qual ele deve ser enviado e requisita uma transmissão de pacote para o árbitro. A FIFO sempre indica o número de posições disponíveis para o árbitro do seu roteador vizinho, pela porta *AvailableOut*, implementando assim o controle de fluxo baseado em crédito.

III.26. MeshRouter.h/ MeshRouter.cpp

Funcionalidade: Implementação em alto nível do roteador da NoCX4. Todos os mecanismos de roteamento, controle de fluxo, bufferização e arbitragem são feitas em software, implementados em funções, porém tentando emular um dispêndio de ciclos para estas operações semelhante ao do roteador em baixo nível. Esta implementação utiliza os arquivos *Buffer.h* e *Ports.h*, que, apesar de não serem módulos de SystemC, implementam funções de hardware. O fato de estes não serem módulos de SystemC evita que eles sejam alocados no *kernel* do SystemC, o que acarreta em um ganho no tempo de simulação, porém há uma perda de precisão na quantidade de ciclos.

III.27. LeafRouter.h/ LeafRouter.cpp

Funcionalidade: Implementação em alto nível do *Leaf Router* (Roteador Folha) da Árvore Obesa. Esta implementação é feita de forma bem semelhante à do *MeshRouter*.

III.28. IntermediaryRouter.h/ IntermediaryRouter.cpp

Funcionalidade: Implementação do *Intermediary Router* (Roteador Intermediário) da Árvore Obesa. Também de forma muito semelhante ao *MeshRouter*. A diferença entre o *LeafRouter* e o *IntermediaryRouter* é que este último não permite a comunicação entre as portas inferiores. Além disso, o *IntermediaryRouter* possui informação sobre o seu nível na árvore, para que possa realizar o roteamento.

III.29. Buffer.h

Funcionalidade: Implementação da FIFO, responsável pela bufferização de pacotes nos roteadores em alto nível. Por não ser um módulo em SystemC, e sim apenas uma classe, sua comunicação com o roteador se dá através de funções.

III.30 Ports.h

Funcionalidade: Implementação das portas utilizadas nos roteadores em alto nível. Realiza arbitragem e controle de fluxo. Da mesma forma que o *Buffer.h*, o *Ports.h* não constitui um módulo de SystemC, e sim uma classe, com métodos e atributos. A comunicação com o roteador se dá pela chamada destes métodos