UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RODRIGO BITTENCOURT MOTTA

Reduzindo o Consumo de Energia em MPSoCs Heterogêneos via Clock Gating

Dissertação submetida à avaliação, como requisito parcial para a obtenção de grau de Mestre em Ciência da Computação.

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, março de 2008.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Motta, Rodrigo Bittencourt

Reduzindo o Consumo de Energia em MPSoCs Heterogêneos via Clock Gating / Rodrigo Bittencourt Motta – Porto Alegre: Programa de Pós-Graduação em Computação, 2008.

106 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2008. Orientador: Flávio Rech Wagner.

1.Clock Gating. 2.MPSoCs Heterogêneos 3. Processadores Java 4. Sistemas Embarcados. I. Wagner, Flávio Rech. II Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic and Logic Unit
ASIP	Application Specific Instruction Set Processor
CAD	Computer Aided Design
CPI	Ciclos Por Instrução
DCF	Dynamic Core Freezing
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
HASHI	Heterogeneous Architectures Simulator through a High Level Interface
ILP	Instruction Level Parallelism
IMAR	Instruction Memory Address Register
ICS	Instruction Classes Stats
ISA	Instruction Set Architecture
JESS	Java Expert Shell System
JVM	Java Virtual Machine
MIC	Memory Interface Controller
MMR	Memory Management Routines
MP3	MPEG Audio Layer 3
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
PC	Program Counter
PMI	Processor Modeling Information
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SASHIMI	System AS Software and Hardware In MIcrocontrollers
SEEP	Sistemas Eletrônicos Baseados em Plataforma

SMP	Symmetric Multiprocessor
SP	Stack Pointer
VHDL	VHSIC hardware description language
VHSIC	Very-High-Speed Integrated Circuits
VLIW	Very Long Instruction Word

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Nesse trabalho é apresentada uma arquitetura que habilita a geração de MPSoCs (*Multiprocessors Systems-on-Chip*) heterogêneos escaláveis, baseados em barramento, suportando ainda o uso de diferentes organizações de memória. A comunicação entre as tarefas é especificada por meio de uma estrutura de memória compartilhada, que evita colisões e promove ganhos energéticos através do disparo dinâmico de *clock gating*. Também é introduzida a técnica DCF (*Dynamic Core Freezing*), que incrementa a eficiência energética do MPSoC tirando proveito dos ciclos ociosos dos processadores durante os acessos à memória. Mais, a combinação das organizações de memória propostas habilita a exploração de migração de tarefas na arquitetura proposta, por meio da troca de contexto das tarefas na memória compartilhada.

Além disso, é mostrado o simulador de alto-nível, baseado na arquitetura proposta, criado com o propósito de extrair os ganhos energéticos propiciados com o uso do *clock gating* e da técnica DCF. O simulador aceita como entrada arquivos de *trace* de execução de aplicações Java, com os quais ele gera um novo arquivo contendo o mapeamento das instruções encontradas nos arquivos de *trace* para diferentes classes de instrução. Dessa forma, podem ser modeladas diferentes arquiteturas de processadores, usando o arquivo com o mapeamento para simular o MPSoC. Mais, o simulador habilita ainda a exploração das diferentes organizações de memória da arquitetura proposta, de maneira que se pode estimar o seu impacto no número de instruções executadas, contenções no barramento, e consumo energético.

Experimentos baseados em uma aplicação sintética, executando em um MPSoC composto por diferentes versões de um processador Java mostram um grande aumento na eficiência energética com um custo mínimo em área. Além disso, também são apresentados experimentos baseados em aplicações do *benchmark* SPECjvm98, que mostram o impacto causado na eficiência energética quando o tipo de aplicação é alterado. Mais, os experimentos mostram drásticos ganhos energéticos obtidos com a aplicação da técnica DCF sobre as memórias do MPSoC.

Palavras-Chave: *Clock Gating*, MPSoCs Heterogêneos, Processadores Java, Sistemas Embarcados.

Reducing Energy Consumption in Heterogeneous MPSoCs through Clock Gating

ABSTRACT

In this work we present an architecture that enables the generation of bus-based, scalable heterogeneous Multiprocessor Systems-on-Chip (MPSoCs), supporting different memory organizations. Intertask communication is specified by means of a shared memory structure that assures collision avoidance and promotes energy savings through a dynamic clock gating triggering. We also introduce a Dynamic Core Freezing (DCF) technique, which boosts energy savings taking advantage of processor idle cycles during memory accesses. Moreover, the combination of the memory organizations enables the architecture to exploit easy task migration by means of the task context saving in the shared data memory.

Moreover, we show the high-level simulator, based on the proposed architecture, created in order to extract the energy savings enabled with the clock gating and the DCF techniques. The simulator accepts as input execution trace files of Java applications, from which it generates a new file that contains the mapping of the instructions found in the trace file for different instruction classes. This way, we can model different processor architectures, using the mapping file to simulate the MPSoC. Also, the simulator enables us to experiment with different memory organizations to estimate their impact on the executed instructions, bus contention, and energy consumption. As case study we have modeled different versions of a Java processor in order to experiment with different execution patterns over different memory organizations.

Experiments based on a synthetic application running on an MPSoC containing different versions of a Java processor show a large improvement in energy efficiency with a minimal area cost. Besides that, we also present experiments based on applications of the SPECjvm98 benchmark, which show the impact on the energy efficiency when we change the application type. Moreover, the experiments show a huge improvement in the energy efficiency when applying the DCF technique to the MPSoC memories.

Keywords: Clock Gating, Heterogeneous MPSoCs, Java Processors, Embedded Systems.

1 INTRODUÇÃO

O crescimento contínuo da demanda por dispositivos mais portáveis, e com funcionalidades mais complexas, apresenta desafios diretos ao desenvolvimento de processadores, que precisam acompanhar os crescentes requisitos de desempenho ao mesmo tempo em que reduzem seu consumo energético. Mais, estudos recentes esboçam um cenário no qual as tendências para o desenvolvimento de computadores móveis não são capazes de atender aos futuros requisitos de potência e desempenho (AUSTIN et al., 2004).

Paralelamente, cada geração de tecnologia de fabricação de circuitos integrados tem dobrado o número de transistores. Em computadores *desktop*, esse aumento no número de transistores tem conduzido, até recentemente, a uma exploração extensiva do ILP (*Instruction Level Paralellism*), através de técnicas como *superpipelining*, execução superescalar, escalonamento dinâmico, caches multinível, e execução especulativa (HENNESSY et al., 2003). Entretanto, como mostrado em (WILCOX et al., 1999), a exploração agressiva do ILP pode consumir até 55% do total de dissipação de potência de um processador RISC (*Reduced Instruction Set Computer*) tradicional, o que claramente não é compatível com aplicações embarcadas.

Por outro lado, uma possibilidade interessante é usar esse número extra de transistores para desenvolver um chip com múltiplos processadores (SCHAUMONT et al., 2005; MAGARSHACK et al., 2003; OLUKOTUN et al., 1996). Arquiteturas desse tipo têm sido usadas há algum tempo em servidores, mais recentemente em computadores *desktop*, e ultimamente estão aparecendo também no cenário embarcado.

As novas aplicações portadas para sistemas embarcados demandam complexos chips com múltiplos processadores para atender seus requisitos de processamento em tempo-real. Telefones celulares modernos, por exemplo, podem ter de 4 a 8 processadores embarcados, incluindo um ou dois processadores RISC para as interfaces com o usuário, processamento da pilha de protocolos e controle de outras funções; um DSP (*Digital Signal Processor*) para codificação e decodificação de voz e interface de rádio; um processador de áudio para tocar músicas; um processador de imagens para as funções da câmera digital integrada; e até mesmo um processador de vídeo para as novas funções de transmissão de vídeo. Adicionalmente, podem existir outros processadores para substituir outras funções tradicionalmente desenvolvidas como blocos de hardware dedicados (MARTIN, 2006).

Como aplicações embarcadas são inerentemente paralelas, elas tipicamente se beneficiam de ambientes de programação multitarefa. O suporte a múltiplas tarefas simplifica a codificação, permitindo uma solução modularizada e habilitando ainda o reuso de código. Esse suporte pode vir de duas fontes: de uma camada de software que multiplexa o hardware entre as tarefas concorrentes, ou de uma camada de hardware que

implementa essa função diretamente. Ambas as soluções podem ainda ser combinadas, com uma camada de software multiplexando uma camada de hardware com suporte direto à multitarefa.

Entretanto, esse novo paradigma impõe muitos desafios para os arquitetos de sistemas, projetistas de hardware e software, especialistas em verificação e integradores de sistemas.

1.1 Desafios Arquiteturais em MPSoCs

Muitas publicações recentes têm mostrado um aumento no interesse por arquiteturas MPSoC (*Multiprocessor System-on-Chip*) (AUSTIN et al., 2004; GEER, 2005) devido às suas inúmeras vantagens, tais como eficiência energética (SASANKA et al., 2004; KOGEL et al., 2004), aumento de performance sem acréscimo de complexidade ao sistema (WOLF, 2004; PAULIN et al., 2004), e até mesmo a exploração do espaço de projeto proporcionada pela migração de tarefas (BERTOZZI et al., 2006; BRIAO et al, 2007).

Entretanto, esses sistemas são difíceis de projetar, parte pelo fato de que eles implementam funções sofisticadas, e parte pelo fato de eles usarem uma vasta gama de tecnologias durante sua execução. Além disso, na implementação de MPSoCs, diferentes decisões de projeto vêm à tona, tais como o número e tipo de processadores (LEIBSON et al., 2005), as organizações e otimizações de memória (BOUCHEBABA et al., 2007; GUZ et al., 2007), o mecanismo de interconexão (SHEYNIN et al., 2006), o escalonamento das tarefas (RICHTER et al., 2003), entre outras.

O sistema de memória, em particular, é um fator chave para lidar com problemas de tempo-real e energia/potência. Como mostrado em (WOLF et al., 2003), o sistema de memória freqüentemente domina o consumo de potência de sistemas embarcados. Contudo, os sistemas de memória têm sido há muito tempo foco de otimizações arquiteturais para incrementar o desempenho médio à custa de grandes variações nos tempos de acesso. Muitas aplicações MPSoC têm comunicação intensiva com a memória, amplificando, dessa forma, o efeito dessas variações no tempo de processamento (WOLF, 2004).

Outra questão simples, mas importante, é a escolha do modelo de programação do sistema de memória, que pode usar memória compartilhada ou particionada. Em um MPSoC com uma estrutura de memória compartilhada, o número de processadores conduz a um aumento no número de contenções no mecanismo de interconexão (WINEGARDEN, 2000). Então, o sistema se torna menos previsível e o acesso à memória se torna não-uniforme. A contenção também piora a eficiência energética, porque, quando os processadores estão esperando para acessar a memória, eles estão também gastando potência devido à propagação do sinal de relógio. Como mostrado em (GOWAN et al., 1998), a potência gasta para propagar o sinal de relógio é o gargalo do consumo de potência de um processador e pode consumir até 35% de seu total.

Todavia, a potência do sinal de relógio pode ser reduzida usando uma técnica chamada *clock gating* (AGARWAL et al., 2007; MUELLER et al., 2004). Conectando o sinal de relógio e um sinal de controle a uma porta AND, o *clock gating* essencialmente desabilita o relógio de um componente como se o componente não estivesse sendo usado, evitando assim a dissipação de potência devida ao carregamento/descarregamento de circuitos não usados (BHUNIA et al., 2003).

Paralelamente, a utilização de estimadores de consumo de potência é um fator estratégico para o desenvolvimento de aplicações com restrições energéticas. Enquanto vários estimadores de consumo de potência têm sido desenvolvidos em um passado recente (GIVARGIS et al., 2002; GURUMURTHI et al., 2002; SIMUNIC et al., 2001; VIJAYKRISHNAN et al., 2003), há uma carência de estimadores flexíveis e genéricos o suficiente para analisar o consumo de potência em MPSoCs complexos com vários processadores comunicantes.

Ao mesmo tempo em que essa dissertação ataca o sistema de memória dos MPSoCs, propondo uma arquitetura baseada em memória compartilhada que usa *clock gating* para gerenciar os acessos e auxiliar na redução do consumo de potência, fornece também um simulador de MPSoCs com precisão de ciclo, que auxilia o desenvolvedor na exploração do espaço de projeto através da análise do impacto que as diferentes configurações de processadores e organizações de memória exercem sobre o consumo energético.

1.2 Contribuições dessa Dissertação

Esse trabalho apresenta uma arquitetura que habilita a geração de MPSoCs (*Multiprocessors Systems-on-Chip*) heterogêneos escaláveis, baseados em barramento, com uma estrutura de memória compartilhada. É proposto um mecanismo de disparo dinâmico de *clock gating* para gerenciar os múltiplos acessos à memória compartilhada, evitando as colisões e reduzindo ainda o consumo energético. Adicionalmente, é introduzida a técnica DCF (*Dynamic Core Freezing*), que permite aumentar a eficiência energética através da detecção dinâmica dos acessos à memória. Dessa forma, pode-se também usar o *clock gating* para tirar proveito dos ciclos ociosos dos processadores causados pelos atrasos nos acessos à memória. Mais, nesse trabalho são exploradas diferentes organizações de memória para permitir que a arquitetura proposta efetue migração de tarefas entre os processadores do MPSoC.

Para extrair o ganho energético do gerenciamento de memória propiciado pela arquitetura proposta, foi desenvolvido, adicionalmente, um simulador, chamado Hashi (*Heterogeneous Architectures Simulator through a High-Level Interface*). O simulador recebe como entrada arquivos de *trace* de execução de aplicações Java. Por meio desses arquivos é possível extrair a freqüência dinâmica das diferentes classes de instrução da aplicação e realizar uma simulação, com precisão de ciclo, de MPSoCs com diferentes processadores e diferentes organizações de memória. Para tanto, é necessário realizar, a priori, uma modelagem das diferentes classes de instrução executando em cada processador utilizado. O simulador permite a visualização detalhada do estado de cada um dos processadores durante o processo de simulação, bem como a análise do impacto da configuração utilizada sobre o número de contenções no barramento, desempenho, e eficiência energética.

Para extração de resultados foram modeladas diferentes versões de um processador Java de mesmo ISA (*Instruction Set Architecture*) no simulador Hashi, permitindo assim a realização de experimentos com diferentes configurações de MPSoC. Em um experimento baseado em uma aplicação sintética onde o número de acessos à memória compartilhada foi variado de 10% até 50% do número total de acessos à memória constatou-se um aumento médio de 25.9% no ganho energético. Mais, habilitando a técnica DCF foi atingido um incremento adicional de 16.8%. Em outro experimento, usando aplicações do pacote de *benchmarks* SPECjvm98 e as mesmas variações na taxa de acesso à memória compartilhada, foram constatados ganhos energéticos médios de

22.1%. Já com a técnica DCF ativa, foi possível aumentar os ganhos energéticos em 4.2%, em média.

Detalhando, as principais contribuições dessa dissertação são as seguintes:

- Desenvolvimento de uma infra-estrutura para gerenciamento de memória em MPSoCs heterogêneos, baseada em barramento, com suporte a diferentes organizações de memória;
- Aplicação de *clock gating* para evitar colisões no barramento, habilitando ainda redução do consumo energético;
- Proposição da técnica DCF para habilitar a redução do consumo de potência nos ciclos ociosos dos processadores durante os acessos à memória;
- Desenvolvimento de um simulador de MPSoCs heterogêneos baseado na arquitetura proposta, que suporta a modelagem de diferentes processadores e organizações de memória;
- Uso de processadores Java como caso de estudo;
- Estimação do custo em área da arquitetura proposta;
- Estimação da redução energética propiciada pela aplicação do *clock gating* e do DCF;
- Estimação do desempenho (em número de instruções executadas) e do número de contenções no barramento gerados com o uso de diferentes configurações de MPSoC e organizações de memória.

1.3 Organização da Dissertação

O resto dessa dissertação está organizado como segue. No Capítulo 2 é apresentada uma revisão bibliográfica sobre MPSoCs, onde são abordados assuntos como: modelos de comunicação e organizações de memória nesse tipo de sistema. O capítulo aborda também trabalhos que aplicam diferentes técnicas de redução energética usando MPSoCs no âmbito embarcado, e ainda trabalhos que se baseiam no uso do *clock gating* para reduzir o consumo de potência do chip. Adicionalmente são mostrados trabalhos em que essa técnica é aplicada em contextos diferentes do convencional.

No Capítulo 3 é esboçada a contextualização desse trabalho. Para tanto, o fluxo do projeto Sistemas Eletrônicos Embarcados baseados em Plataformas (SEEP) é detalhado, e suas etapas são explicitadas. Dessa forma, pode-se identificar onde a contribuição desse trabalho se situa dentro da metodologia do projeto. O capítulo apresenta ainda o Sashimi (*System AS Software and Hardware In MIcrocontrollers*), que inclui uma metodologia e uma ferramenta para o projeto e geração de sistemas embarcados e as diferentes versões do processador Java utilizadas no escopo desse trabalho.

O Capítulo 4 detalha a arquitetura proposta e as suas diferentes organizações de memória. É ilustrado também como a técnica *clock gating* é usada nesse contexto para gerenciar os acessos à memória compartilhada do MPSoC. Além disso, a técnica DCF é introduzida, detalhando como a eficiência energética é aprimorada aplicando *clock gating* nos ciclos ociosos dos processadores durante os acessos à memória. Por fim, o processo de configuração da arquitetura é explicado, e o experimento realizado para validar a arquitetura é exposto.

No Capítulo 5 é apresentado o simulador Hashi. O capítulo mostra o funcionamento do simulador, detalhando como é realizado o processo de análise dos arquivos de *trace* de execução, usado para fazer o mapeamento das instruções Java nas diferentes classes

de instrução usadas posteriormente na simulação do MPSoC. O processo de simulação e a estimação dos resultados são também delineados, abordando as principais rotinas funcionais do simulador. Apresenta-se ainda a interface do simulador, destacando as informações que podem ser extraídas.

Finalmente, o Capítulo 6 exibe as conclusões desse trabalho e algumas propostas de trabalhos futuros, tais como a implementação de memórias cache na arquitetura, a realização de experimentos com outros processadores, a troca da estrutura de barramento por uma estrutura baseada em NoC (*Network-on-Chip*), o particionamento das memórias compartilhadas de dados e instruções, e ainda a exploração da migração de tarefas entre os processadores do MPSoC.

2 TRABALHOS RELACIONADOS

A arquitetura proposta nesse trabalho envolve tópicos como MPSoC, sistemas embarcados, organizações de memória e *clock gating*. Dessa forma, nesse capítulo, esses conceitos são apresentados e os trabalhos relacionados são analisados focando no desempenho e na eficiência energética. Esse capítulo está organizado como segue. A Seção 2.1 descreve os modelos de comunicação comumente usados em projetos de MPSoC, apresentando o conjunto de vantagens e desvantagens de cada modelo. Na Seção 2.2 são mostradas algumas abordagens de organização de memória utilizadas em MPSoCs. A Seção 2.3 aborda a utilização de MPSoCs no âmbito de sistemas embarcados, mostrando técnicas que podem ser usadas para auxiliar no aumento de desempenho e redução de potência. Finalmente, a Seção 2.4 introduz o *clock gating*, apresentando ainda trabalhos relacionados onde a técnica é usada em contextos diferentes dos convencionais.

2.1 Modelos de Comunicação em MPSoCs

Os sistemas multiprocessadores atuais são muito parecidos com os sistemas de memória compartilhada de alguns anos atrás. Nesse tipo de sistema podem-se implementar diferentes modelos de comunicação. Cada um desses modelos apresenta seus próprios compromissos de desenvolvimento, mas também benefícios particulares. Através da mudança do nível de integração e como os recursos são compartilhados ou particionados, cada método apresenta um diferente conjunto de prós e contras.

Para propósitos de classificação, cada tipo de sistema será descrito com base no menor nível de integração entre os processadores. A Figura 2.1 exibe três modelos típicos de comunicação em sistemas multiprocessadores. Entretanto, deve-se notar que, apesar de existirem outros modelos de comunicação possíveis, a figura destaca apenas os modelos tradicionalmente utilizados no contexto de multiprocessadores. No contexto desse trabalho, o termo “I/O Interface” ou “I/O” estará se referindo à comunicação externa ao chip, incluindo memória principal, discos rígidos, rede, etc. Mais, deve-se observar que no modelo de cache compartilhada da Figura 2.1a, apenas os níveis mais altos de cache (L2 ou L3) são compartilhados, pois a cache L1 atualmente está tão integrada ao processador que é sempre privada. Adicionalmente, deve-se notar que a seta pontilhada exibida em cada modelo representa como é realizado o roteamento da comunicação entre os processadores.

A Figura 2.1a apresenta a abordagem utilizando memória cache compartilhada. Essa abordagem apresenta a menor latência de comunicação. Logo que o processador 1 escreve uma linha da cache, o processador 2 tem acesso sem ter que procurar por um bloco remoto. Isso mantém o tráfego de dados longe da interface de I/O, maximizando a banda para os outros dispositivos. Outra vantagem é que a cache compartilhada pode ser dinamicamente alocada entre os processadores. Supondo-se que o processador 1 está

utilizando toda a cache que lhe foi alocada, mas o processador 2 está utilizando apenas uma parcela da sua cache disponível, o controlador da cache pode, dinamicamente, fornecer para o processador 1 um pouco da cache que o processador 2 não está utilizando.

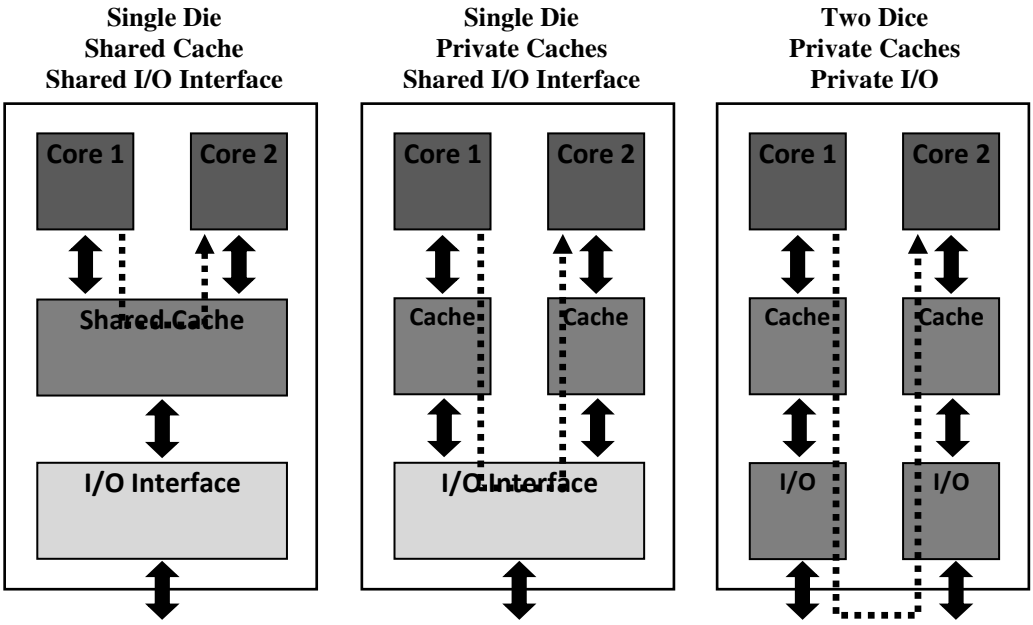


Figura 2.1: Diferentes Modelos de Comunicação em Sistemas Multiprocessadores

No entanto, enquanto as vantagens dessa abordagem vêm em forma de desempenho, as desvantagens vêm em forma de complexidade. Isso acontece porque o controlador da cache tem que gerenciar a política de compartilhamento para a cache e manipular ainda a alocação dinâmica. Essa última função requer que o controlador da cache monitore as necessidades de cada processador e decida como alocar recursos. Se essa alocação for feita de maneira incoerente, pode haver casos extremos onde um processador utilizando uma porção muito grande dos recursos causa problemas de desempenho. Mais, a banda alocada para a cache precisará ser maior, pois ela precisará servir dois processadores, ao invés de um. De maneira similar, deve haver uma escolha entre uma cache de múltiplas portas, que pode servir os dois processadores simultaneamente; e uma cache que simplesmente enfileira as requisições dos diferentes processadores. Outra desvantagem é que os processadores não podem ser separados para formar processadores de único núcleo (como no modelo de encapsulamento compartilhado, que será discutido mais adiante). Naturalmente, todas essas decisões se traduzem em um tempo maior para o desenvolvimento e a validação.

Esse modelo de comunicação foi adotado nas famílias de processadores IBM Power4 (WARNOCK, 2003) e Power5 (KALLA et al., 2004), e Sun UltraSPARC-IV (YONGHONG, 2005) e Niagara (GEPPERT, 2004).

A Figura 2.1b apresenta o modelo com I/O compartilhado. Essa abordagem tem a vantagem da simplicidade em relação ao modelo com cache compartilhada, mas isso é pago com menor desempenho e flexibilidade. Como no modelo anterior, a comunicação entre os núcleos é realizada sem a necessidade de sair do chip, e é roteada pela mesma lógica que manipula o barramento ou o tráfego de rede. Como as caches são separadas,

o controlador da cache é idêntico aos encontrados em processadores de único núcleo. Isso significa que são necessárias relativamente poucas modificações em um chip de único núcleo para se produzir um chip de múltiplos núcleos com interface de I/O compartilhada. Basicamente são necessários alguns aprimoramentos na interface de I/O, tais como a habilidade de trocar informações com múltiplas caches simultaneamente. Entretanto, como esses aprimoramentos são ortogonais aos processadores, podem ser implementados em fases posteriores do projeto.

As desvantagens desse modelo resultam da ausência de integração no nível da cache. Por isso, não se pode fazer balanceamento dinâmico de carga, o que conduz a desperdício de recursos. Naturalmente, o gargalo desse modelo é a interface de I/O, a qual deve ser robusta o suficiente para manipular simultaneamente os tráfegos interno e externo do chip.

Esse foi o modelo de comunicação adotado nas famílias de processadores Intel Itanium2 (NANRI et al., 2005), que incorpora um árbitro do barramento para gerenciar o tráfego entre os processadores e AMD Opteron (WOLFE, 2004).

A Figura 2.1c mostra o modelo com encapsulamento compartilhado. A grande vantagem desse modelo é a simplicidade de implementação, porque normalmente ele não requer modificações na lógica dos processadores. Conseqüentemente, esse modelo pode ser desenvolvido em um tempo bem menor (em relação aos outros dois) e pode ser implementado nas fases finais do projeto do chip. Esse modelo também habilita o uso de processadores com diferentes características arquiteturais.

A principal desvantagem dessa arquitetura é a latência de comunicação entre os núcleos, que é tão demorada quanto uma comunicação entre dois processadores em uma abordagem SMP (*Symmetric Multiprocessor*).

Essa abordagem foi utilizada pelas famílias de processadores Intel Pentium-D (PENG et al., 2007) e Xeon (CHANG et al., 2007).

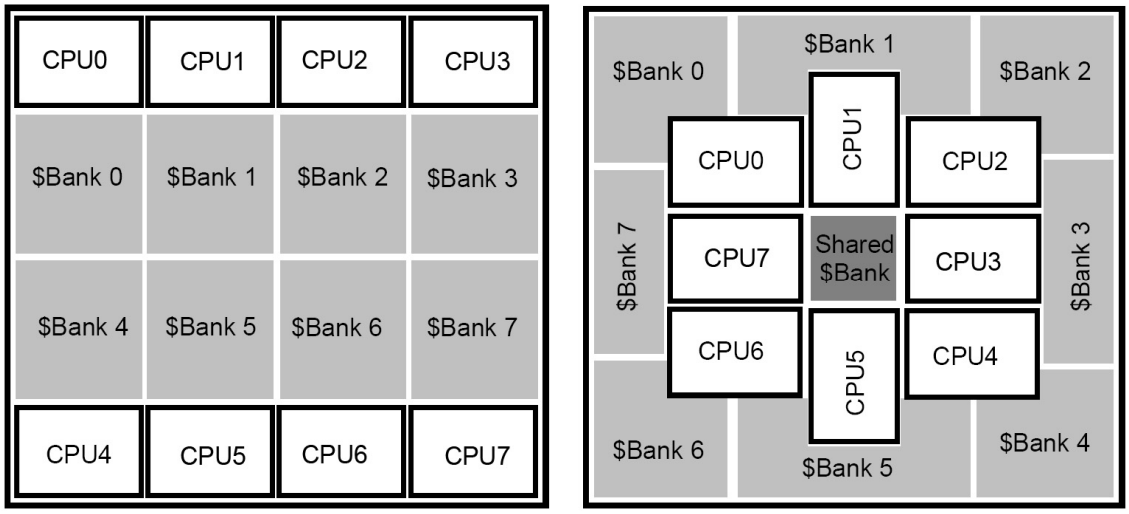
2.2 Organizações de Memória em MPSoCs

Sistemas com múltiplos processadores estão rapidamente se proliferando para todas as áreas da computação, graças a sua habilidade para proporcionar paralelismo, alcançando assim maior desempenho. Entretanto, esse novo paradigma introduz novos desafios arquiteturais. Em particular, os acessos aos dados das memórias são com freqüência o principal gargalo desses sistemas (WOLF, 2004), já que múltiplas *threads* competem pelos recursos limitados de memória do chip. Dessa forma, é natural que essa área de estudo seja alvo de inúmeras pesquisas.

Em (GUZ et al., 2007) é proposta uma organização de memória cache onde é feita uma distinção entre os dados compartilhados, que são acessados por múltiplos processadores, e os dados privados acessados individualmente por cada processador. O trabalho apresenta uma topologia que habilita o rápido acesso à memória compartilhada por todos os processadores, preservando ainda a proximidade à memória privada de cada processador.

Na Figura 2.2a é apresentada a topologia normalmente utilizada em sistemas de múltiplos processadores, onde a cache L2 é localizada no meio do chip. Em contrapartida, a Figura 2.2b apresenta a topologia proposta no trabalho, que é baseada em círculos concêntricos. O circulo mais interno contém as memórias caches L2 compartilhadas, que são cercadas por oito processadores simétricos, posicionados no

círculo central. No circulo mais externo estão posicionadas as caches L1 privadas, cercando todos os processadores. Os processadores e as caches L2 compartilhadas são interconectados através de uma NoC. Simulações demonstram um decréscimo na latência dos acessos à cache L2 de até 54% quando comparados aos projetos tradicionais de MPSoC, habilitando ganhos de desempenho de até 16.3%.



(a) CIM layout. (b) Nahalal layout.

Figura 2.2: Duas organizações de cache para um MPSoC de 8 processadores

Enquanto alguns projetos de MPSoC usam a cache L2 para maximizar a capacidade interna do chip e minimizar os acessos externos ao chip, outros a usam para replicar os dados e limitar o atraso causado pelas interconexões, minimizando assim o tempo de acesso da cache. Recentes projetos de caches híbridas procuram atingir um equilíbrio entre a latência e a capacidade replicando seletivamente alguns blocos da cache. Cooperative Caching (CHANG et al., 2006) e CMP-NuRapid (CHISHTI et al., 2005) têm caches L2 privadas e replicação restrita, de acordo com determinados critérios, enquanto que em (BECKMANN et al., 2006) é proposto um mecanismo de hardware que dinamicamente estima o custo (em número de faltas) e o benefício (em baixa latência de acerto) da replicação e ajusta o nível de replicação para minimizar o tempo de médio de acesso.

Já em (SOININEN et al., 2002) é proposta uma organização de memória configurável que permite que a quantidade de memória do sistema seja alterada de acordo com a necessidade da aplicação. Na arquitetura proposta existem vários blocos de memória que são conectados aos processadores através de uma estrutura de barramento. Para cada bloco de memória existe associado um barramento próprio com um árbitro. Os processadores têm conexões com todos os barramentos através de um *switch*. A idéia geral consiste em controlar esse *switch* através de um processador RISC mais uma lógica de controle. A arquitetura proposta pode ser configurada para implementar desde uma simples memória compartilhada até um sistema completamente paralelo, onde cada bloco de memória é dedicado a um processador. Enquanto que a grande vantagem dessa abordagem é o melhor aproveitamento das memórias, a desvantagem aparece quando há acessos simultâneos aos bancos de memória. Eles acabam causando uma latência na memória que depende do número de acessos realizados.

2.3 MPSoCs em Aplicações Embarcadas

Para entender o paradigma que surge no desenvolvimento de sistemas embarcados, é preciso primeiramente entender o que eles são. Um sistema embarcado pode ser definido como sendo qualquer dispositivo que tenha um computador programável, mas que não tenha como propósito ser um computador de propósitos gerais (WOLF, 2001). A demanda do mercado por esse tipo de dispositivo computacional ainda está em crescimento. Essa é uma indústria com um mercado mundial estimado em 45.9 bilhões de dólares em 2004. Mais, com uma expectativa anual de crescimento de 14% nos próximos anos, esse mercado irá atingir a marca de 88 bilhões de dólares em 2009 (KRISHNAN, 2008). A atual onipresença de tais sistemas é também refletida pelo uso de termos como computação ubíqua e computação pervasiva (GRIMHEDEN et al., 2005).

Sistemas de múltiplos processadores começaram a entrar no mercado durante os últimos anos e é esperado que eles estejam disponíveis em grande variedade nos próximos anos. Desse modo, o interesse nesse tipo de sistema também cresceu na comunidade embarcada.

Em (KUMAR et al., 2006) é mostrado o potencial do uso de MPSoCs heterogêneos para reduzir o consumo de potência em aplicações embarcadas. É assumido um chip único contendo um conjunto de processadores de mesmo ISA (*Instruction Set Architecture*), mas com diferentes níveis de desempenho e consumo de potência, representando diferentes escolhas no espaço de projeto. É considerada ainda a habilidade de alternar dinamicamente as tarefas entre os processadores, permitindo que uma arquitetura se adapte não apenas às diferenças entre as aplicações, mas também às diferenças entre as fases da mesma aplicação. Dessa forma, em tempo de execução a camada de software do sistema pode avaliar os requisitos da aplicação e escolher qual o processador que melhor se encaixa nesses requisitos, ao mesmo tempo em que minimiza o consumo de potência.

Para os experimentos realizados foram modelados quatro processadores: EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264) e EV8-, sendo que o último é uma versão hipotética do processador de EV8 (Alpha 21464). Assumiu-se os processadores implementados na tecnologia 0.10 mícron e que eles executam em uma freqüência de 2.1GHz. Os processadores têm ainda memórias cache L1 privadas e uma memória cache L2 de 3.5MB, compartilhada. A Tabela 2.1 sumariza as configurações dos processadores modelados, e a Tabela 2.2 apresenta os dados de potência e área dos mesmos.

Tabela 2.1: Configuração dos processadores

Processor	EV4	EV5	EV6	EV8-
Issue-width	2	4	6 (OOO)	8 (OOO)
I-Cache	8KB, DM	8KB, DM	64KB, 2-way	64KB, 4-way
D-Cache	8KB, DM	8KB, DM	64KB, 2-way	64KB, 4-way
Branch Pred.	2KB,1-bit	2K-gshare	hybrid 2-level	hybrid 2-level (2X EV6 size)
Number of MSHRs	2	4	8	16

Tabela 2.2: Estatísticas de potência e área de cada processador

Core	Peak-power (Watts)	Core-area (mm^2)	Typical-power (Watts)	Range (%)
EV4	4.97	2.87	3.73	92-107
EV5	9.83	5.06	6.88	89-109
EV6	17.80	24.5	10.68	86-113
EV8-	92.88	236	46.44	82-128

Assume-se também que os processadores que não estão sendo utilizados são completamente desligados, ao invés serem deixados ociosos. Dessa forma, os processadores não usados não consomem potência estática nem dinâmica. No entanto, isso introduz uma latência para ligar novamente o processador. No estudo, é estimado que esse tempo é equivalente a mil ciclos de relógio na freqüência de 2.1 GHz. Além disso, de acordo os experimentos realizados, o *overhead* do processo de alternação de processadores tem um impacto quase insignificante no desempenho.

A Figura 2.3(a) mostra o desempenho, medido em instruções executadas por segundo, de um representativo *benchmark*: *applu*. Na figura, é exibida uma curva diferente para cada processador, com cada ponto representando a execução de 1 milhão de instruções. Pode-se notar claramente que durante a execução da aplicação há fases distintas de desempenho, e que o desempenho relativo de cada processador também varia entre essas fases.

O primeiro experimento, mostrado na Figura 2.3(b), baseia-se em uma função que procura minimizar a energia por instrução executada para efetuar a alternação de processador. Em cada intervalo, o processador com menor consumo energético é escolhido, desde que o seu desempenho se mantenha dentro de 10% do desempenho apresentado pelo processador EV8-. Pode-se observar que, apesar do desempenho do processador EV8- ser maior, em vários pontos da execução acontece uma alternação da tarefa para o processador EV6, devido ao seu menor consumo de potência. Há ainda esporádicas alternações para os processadores EV4 e EV5. Essa heurística proporciona em média uma redução energética de 38% com uma degradação média de 4% no desempenho.

O segundo experimento, mostrado na Figura 2.3(c), baseia-se em uma função que procura caracterizar a importância do consumo energético e do tempo de computação na mesma métrica. Dessa forma, o processador selecionado é o que maximiza o número de instruções executadas por segundo em relação ao consumo de potência. Além disso, o intervalo de desempenho para habilitar a alteração de processador deve estar na faixa de 50% do desempenho do processador EV8-. Dessa vez, a redução energética média alcançada foi de 73% com uma degradação média de 22% no desempenho.

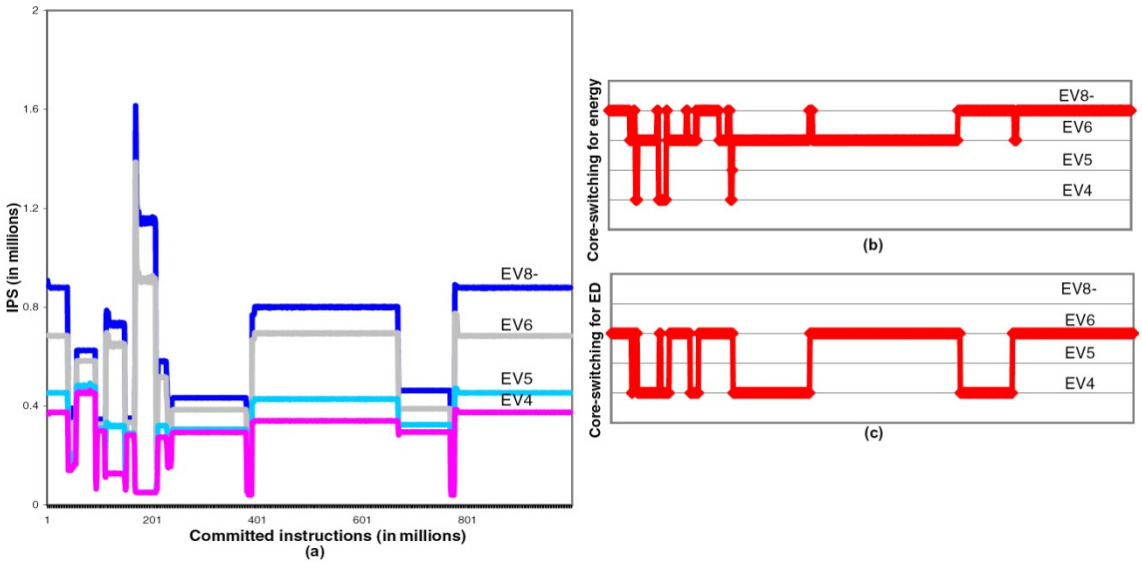


Figura 2.3: Experimentos realizados com a aplicação *applu*

Já em (IPEK et al., 2007) é apresentada uma arquitetura reconfigurável de multiprocessadores, que usa uma técnica chamada de *Core Fusion*, onde grupos fundamentalmente independentes de processadores podem, dinamicamente, se combinar para formar um processador mais poderoso, ou eles podem ser usados como elementos de processamento independentes, conforme exigido em tempo de execução pelas aplicações.

A arquitetura conceitual de um chip de 8 processadores de despacho duplo, e com a capacidade de aplicar *Core Fusion*, é exibida na Figura 2.4. A figura mostra uma configuração assimétrica, arbitrariamente escolhida, que compreende um processador de despacho óctuplo, outro de despacho quádruplo, e outros dois de despacho duplo. Um barramento conecta as caches L1 privadas de instruções e dados e efetua a coerência dos dados. A cache L2 e o controlador de memória residem no outro lado do barramento. Os processadores podem executar independentemente ou podem se fundir em grupos de dois ou quarto processadores para constituir processadores maiores.

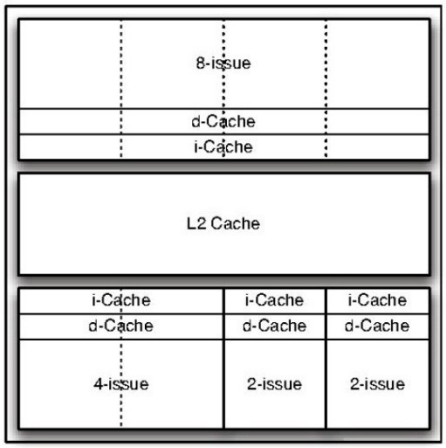


Figura 2.4: Arquitetura conceitual de um chip com 8 processadores com capacidade para aplicar a técnica *Core Fusion*

Uma abordagem interessante a respeito do suporte ao escalonamento em hardware em arquiteturas SMP (*Symmetric Multiprocessor*) é discutido em (NACUL et al., 2007). O trabalho propõe um sistema operacional de tempo-real em hardware que implementa o escalonamento das tarefas em uma arquitetura SMP com dois processadores homogêneos. A comunicação entre as tarefas é implementada por meio de um mecanismo de memória compartilhada, que usa uma infra-estrutura de conexão baseada em barramento. As colisões são evitadas por meio de operações *test-and-set*, que são efetuadas por cada um dos processadores. Entretanto, a arquitetura não detecta automaticamente essas colisões. Elas devem ser garantidas pelo programador no processo de codificação através de requisições de *lock* e *unlock*.

2.4 Estimadores de Consumo de Potência

A estimação do consumo de potência no desenvolvimento de aplicações com baixo consumo de potência é uma área de pesquisa relativamente madura em ambientes de único processador. Um trabalho, apresentado por (TIWARI et al., 1994), introduz o conceito de análise de potência em nível de instrução, um modelo que associa o consumo de potência a instruções ou pares de instruções. A construção do modelo (também chamado de caracterização) é realizada através da extração dos dados de simulações em baixo-nível do processador alvo, ou ainda através de medições no chip real. Após a construção do modelo, a potência consumida por um programa executando no processador pode ser estimada através do uso de um simulador do conjunto de instruções padrão, usado para extrair um *trace* de instruções e determinar a soma dos custos de cada instrução.

Uma solução melhor e mais precisa (porém, com um maior tempo de execução), pode ser obtida com modelos de potência microarquiteturais (BROOKS et al., 2000; VIJAYKRISHNAN et al, 2003), que assumem o conhecimento da microarquitetura interna do processador. Nessa abordagem, macro-modelos de potência são obtidos para as unidades principais (para simulações de baixo-nível), e suas contribuições são somadas durante a execução um programa alvo em um simulador microarquitetural. Abordagens microarquiteturais são necessárias para estimar com precisão a potência consumida por processadores mais avançados (processadores superescalares ou processadores com escalonamento dinâmico, por exemplo), onde múltiplas instruções podem ser despachadas em um determinado ciclo.

Mesmo que a potência consumida pelo processador seja significante, há também o consumo de potência no sistema de memória, barramentos de sistema, e periféricos. Na maioria dos sistemas reais, o total de potência consumida pelo processador em relação ao montante final não é dominante, especialmente quando comparado à hierarquia de memória. Então, estimadores em nível de sistema devem levar em conta todos os componentes do sistema. Muitas pesquisas (SIMUNIC et al., 2001; GURUMURTHI et al, 2002; HENKEL et al, 2002; GIVARGIS et al, 2002) propõem estimadores completos de sistema, que combinam conjuntos de simuladores com modelos de potência de processadores, memória, barramento, e periféricos.

Em princípio, a estimação do consumo de potência de um sistema baseado em processador sempre pode ser realizada através da execução de instrução por instrução em um estimador de potência. Todavia, em muitos casos, essa abordagem é muito lenta para otimizações de potência e exploração do espaço de projeto. Dessa forma, diversas técnicas foram propostas para a caracterização do consumo de potência em um nível mais grosso de granularidade que uma simples instrução. Técnicas de macro-

modelagem propõem a extração do consumo de potência para chamadas de sub-rotinas (LAJOLO et al., 2002), para chamadas do sistema operacional (TAN et al., 2002; DICK et al, 2003), e até mesmo para tarefas inteiras (ACQUAVIVA et al., 2003).

Todas as técnicas mencionadas foram aplicadas em sistemas de único processador. Há poucos trabalhos que consideram sistemas multiprocessadores. Um deles, proposto por (LOGHI et al., 2004), apresenta um simulador, chamado MPARM, que realiza simulações de ambientes multiprocessados com precisão de ciclo, gerando a análise de consumo de potência e desempenho do sistema. Entretanto, ele não oferece muita flexibilidade, pois todo o ambiente, incluindo o tipo de processador utilizado, é fixo.

2.5 Redução do Consumo Energético com Clock Gating

Embora a idéia do *clock gating* não seja nova, é bastante difícil determinar quando essa técnica deve ser aplicada. Alguns trabalhos deixam a cargo do desenvolvedor a tarefa de usar a técnica quando necessário (PALUMBO et al., 2002), enquanto que outros aplicam a técnica baseado na informação limitada sobre a máquina de estados da arquitetura (BENINI et al., 1994; RAGHAVAN et al., 1999). Há ainda técnicas que aplicam o *clock gating* automaticamente em *flip-flops* individuais (LANG et al., 1997). Entretanto os seus requisitos de área são grandes em relação à redução de potência alcançada.

Em (BHUNIA et al., 2003) a técnica é usada para propiciar a redução de potência do processador. Nesse trabalho, é usado um padrão determinístico para disparar o *clock gating*, baseado na observação de que é possível saber, a priori, a combinação correta dos componentes do *pipeline* do processador que precisam estar ativos em um determinado ciclo. Os experimentos realizados mostram uma redução média de 19.9% no consumo de potência de um processador superescalar de despacho óctuplo.

Já (STRIK et al., 2000) usa o *clock gating* em um contexto um pouco diferente do convencional. O trabalho apresentada o desenvolvimento de uma arquitetura com múltiplos processadores heterogêneos, que permite o processamento concorrente de múltiplos fluxos multimídia por meio de reconfigurações em tempo-real. Essas reconfigurações permitem que até 25 fluxos de vídeo possam ser intercambiados entre os múltiplos processadores do sistema.

O trabalho adota um modelo de processamento de sinais baseado em grafos Kahn, que são constituídos de tarefas interconectadas através de canais lógicos do tipo FIFO (*First-in, First-Out*). Cada processador presente no sistema pode executar até 4 diferentes tarefas. Ou seja, cada processador do sistema tem a ele associado 4 canais FIFO de entrada e 4 canais FIFO de saída, por onde transitam as instruções das múltiplas tarefas. A seleção do fluxo de instruções que um processador deve executar em um determinado momento é feita através do *clock gating*. Dessa forma, uma FIFO com o sinal de relógio ativo corresponde a uma tarefa ativa no processador. Similarmente, as FIFOS com sinal de relógio inativo representam as tarefas que estão bloqueadas. A seleção da tarefa ativa é feita por um escalonador, baseado no estado atual de cada FIFO.

Nessa dissertação é apresentada uma arquitetura que usa uma estrutura de memória compartilhada e que evita as colisões no barramento e promove redução energética por meio de um rápido mecanismo, baseado em lógica combinacional, que usa *clock gating*. Dessa forma, a arquitetura provê ainda independência de plataforma, escalabilidade e

heterogeneidade. Mais, a arquitetura disponibiliza ainda toda a infra-estrutura necessária para efetuar, de maneira simples, migração de tarefas entre os processadores.

3 CONTEXTUALIZAÇÃO DO TRABALHO

Essa dissertação é parte de um grande projeto, que foi desenvolvido para auxiliar na tarefa de projetar sistemas embarcados. Nesse capítulo são abordadas algumas ferramentas e conceitos originários desse projeto, já que eles servem de base para esse trabalho e contextualizam essa dissertação. Em particular, são apresentadas as versões do processador Java utilizadas nesse trabalho, as suas organizações de memória, e ainda a ferramenta que gera esse processador.

3.1 SEEP

O projeto SEEP (Sistemas Eletrônicos Embarcados Baseados em Plataformas) (LSE, 2003) propõe uma metodologia completa e integrada para o desenvolvimento de sistemas embarcados. O fluxo de desenvolvimento leva em consideração todos os aspectos variáveis que são importantes para o desenvolvimento desse tipo de sistema. Esse ciclo de desenvolvimento pode ser visualizado na Figura 3.1. Os detalhes de cada uma das fases desse diagrama podem ser encontrados em (LSE, 2003).

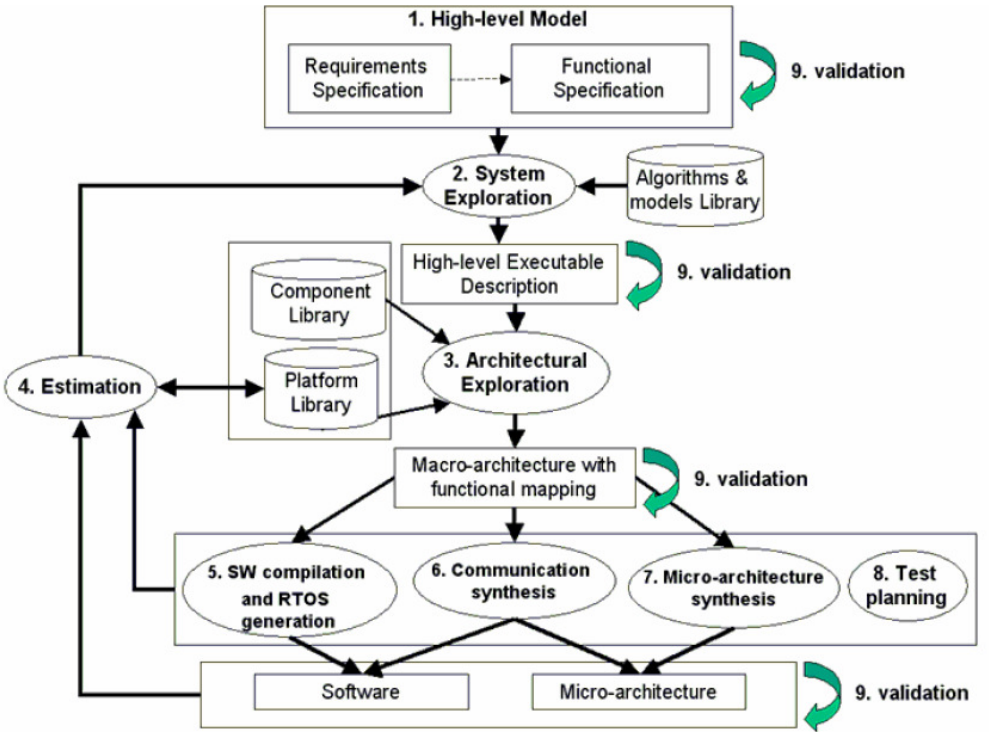


Figura 3.1: O fluxo de desenvolvimento para sistemas embarcados

Como mostrado, a especificação e modelagem do comportamento do sistema começam em alto nível de abstração. A partir do ponto inicial, as possibilidades para exploração do espaço de projeto tanto de software quanto de hardware abrem-se através do uso de bibliotecas. Essas bibliotecas contêm componentes IP de software e hardware já verificados, desempenhando, dessa forma, um aspecto fundamental para a escolha correta da implementação final.

Resumindo, o projeto SEEP sugere uma metodologia automática com a qual o desenvolvedor pode abstrair detalhes da implementação de hardware e software (até mesmo em nível de decidir o que deve se implementar em hardware e o que deve se implementar em software), e especificar um projeto com linguagens de modelagem de alto nível, tais como UML. A partir desse ponto, uma série de ferramentas e bibliotecas ajudam o desenvolvedor a percorrer todas as fases do projeto, como mostrado na Figura 3.1. Dentro desse contexto, o trabalho desenvolvido nessa dissertação deve fornecer mais uma arquitetura para servir como alternativa na exploração do espaço de projeto para a escolha da melhor solução para alcançar os requisitos de área, desempenho, e consumo de potência especificados.

3.2 FemtoJava

O FemtoJava é um processador de pilha desenvolvido para executar *bytecodes* Java nativamente. As características gerais do FemtoJava são: um conjunto reduzido de instruções, arquitetura *Havard*, pequeno tamanho, e facilidade de inserção e remoção de instruções. Há diferentes versões do processador FemtoJava disponíveis, e cada uma delas foi desenvolvida visando aplicações com restrições específicas. A versão Multiciclo (ITO et al., 2001) é ideal para aplicações que necessitam pouca memória e baixo poder de processamento, enquanto que a versão Low-Power (BECK et al., 2003a) pode propiciar ganhos de energia e melhorar o desempenho com algum custo em área. Finalmente, a versão VLIW (BECK et al., 2004) além de poder ser usada para aplicações que demandam alto poder de processamento, pode aumentar a eficiência energética em domínios onde a área do chip não é muito relevante. Todas as versões desse processador foram desenvolvidas especificamente para o mercado de sistemas embarcados.

3.2.1 FemtoJava Multiciclo

O processador Femtojava Multiciclo é o resultado de uma metodologia adotada para a geração semi-automática de um sistema embarcado a partir de uma descrição Java. Esse processador implementa um subconjunto de instruções Java: são apenas 68 no total. Neste subconjunto encontram-se instruções necessárias para operações básicas de pilha, manipulação de vetores, desvios condicionais e incondicionais, execução de métodos estáticos e acesso a campos de classes.

A Tabela 3.1 mostra o conjunto de instruções suportadas pelo Femtojava. Os *bytecodes* estendidos são necessários para executar instruções de E/S, programação de interrupções e também para colocar o processador em modo suspenso. O microcontrolador Femtojava só pode executar código de classes (isto é, não pode alocar objetos dinamicamente) porque seu conjunto de instruções apenas suporta *invokestatic*, *return* e *ireturn* como instruções para manipulação de métodos.

Tabela 3.1: Instruções suportadas pelo processador FemtoJava Multiciclo (ITO et al., 2001)

Instruction type	Mnemonics
Arithmetic & Logic	iadd, isub, imul, ineg, ishr, ishl, iushr, iand, ior, and ixor
Control Flow	goto, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, return, ireturn, and invokestatic
Stack	iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, and swap
Load/store	iload, iload_0, iload_1, iload_2, iload_3, istore, istore_0, istore_1, istore_2, and istore_3
Array	iaload, baload, caload, saload, iastore, bastore, castore, sastore, and arraylength
Extended	load_idx, store_idx, and sleep
Others	nop, iinc, getstatic, putstatic

A microarquitetura do FemtoJava Multiciclo pode ser visualizada na Figura 3.2. Além das características gerais citadas anteriormente, esse processador possui algumas peculiaridades interessantes, como o tamanho da máquina de controle ser diretamente proporcional ao número de instruções utilizadas. Isto só é possível porque o Femtojava Multiciclo é gerado a partir de um ambiente de CAD (*Computer Aided Design*), chamado Sashimi (ITO et al., 2001).

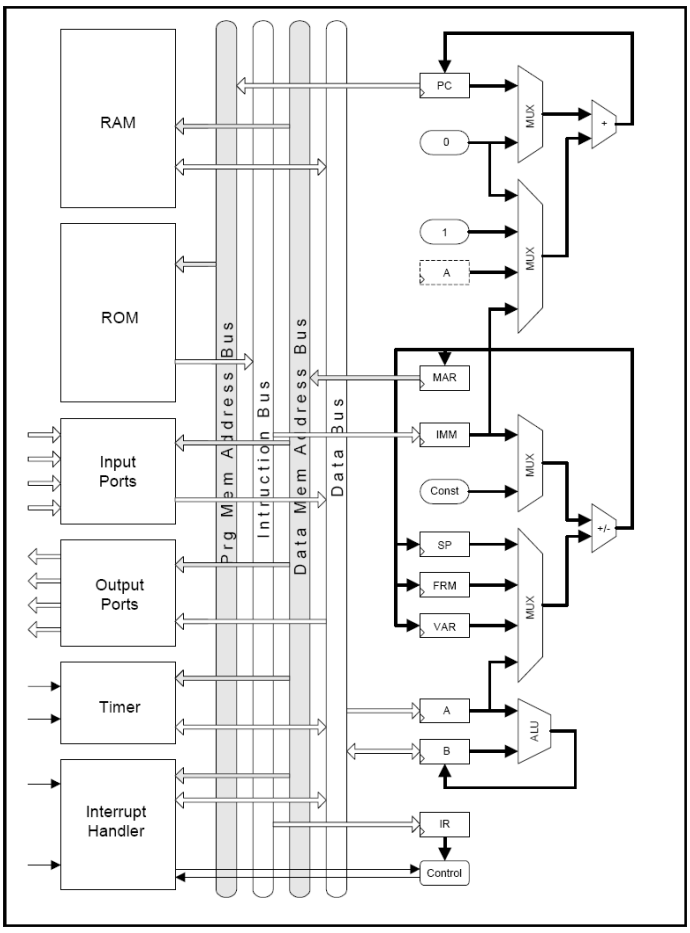


Figura 3.2: A microarquitetura do Femtojava Multiciclo (ITO et al., 2001)

3.2.2 FemtoJava Low-Power

O FemtoJava Low-Power é um processador com um pipeline de cinco estágios: busca de instruções, decodificação de instruções, busca de operandos, execução e escrita dos resultados, como pode ser observado na Figura 3.3. Essa versão do processador FemtoJava implementa o mesmo subconjunto de instruções que a versão Multiciclo, entretanto possui uma arquitetura que visa um baixo consumo de energia.

O *pipelining* é uma técnica que permite o aumento do paralelismo através da utilização de partes do caminho de dados do processador que usualmente estariam ociosas em um determinado ciclo de execução de uma instrução. O *pipelining* tem a vantagem de reduzir a taxa de Ciclos Por Instrução (CPI). Entretanto, essa técnica também causa problemas de dependências de dados e desvios. Os desvios tomados causam um aumento no CPI, pois enquanto o desvio está sendo avaliado em um determinado estágio do pipeline, o processamento das instruções nos estágios precedentes é perdido. No FemtoJava Low-Power, para economizar área, nenhum mecanismo especial é usado para tratar esse problema, mas, como a profundidade do pipeline é pequena (apenas 5 estágios), o impacto causado pelos desvios tomados é minimizado. Dessa forma, assume-se que os desvios não serão tomados, e paga-se uma penalidade de três ciclos quando os desvios forem tomados.

Para lidar com as dependências de dados, o FemtoJava Low-Power usa a técnica *forwarding* (HENNESSY et al., 2003). Se há uma instrução no estágio de execução que irá escrever seu resultado na pilha, e a instrução seguinte, no estágio de busca de operandos, precisa acessar a pilha para buscar esse resultado, uma dependência de dados verdadeira é caracterizada. Usando *forwarding*, o resultado é automaticamente passado do estágio de execução para o estágio de busca de operandos. Sem o *forwarding*, o CPI seria maior, pois seria necessário parar o pipeline, através da inserção de bolhas, até que a instrução mais antiga atingisse o estágio de escrita de resultados.

Grandes vantagens, com pequeno *overhead* em área, podem ser derivadas dessa técnica, especialmente em processadores de pilha (BECK, 2004a). Em instruções que manipulam a pilha, os operandos que sofrem *forwarding* para estágios anteriores do pipeline não serão mais usados. Então, não há necessidade de escrever esses resultados de volta na pilha. O resultado é a redução no consumo de potência por meio da diminuição do número de escritas na pilha.

Dois tipos de *forwarding* podem ocorrer: quando a instrução no estágio de execução consome um operando do topo da pilha (como *istore*, que salva o topo da pilha em algum lugar do pool de variáveis); ou quando a instrução consome dois operandos da pilha (como as operações lógicas e aritméticas: *iadd*, *isub*, *ior*). No primeiro caso, os operandos que sofrem *forwarding* vêm do estágio de execução para o estágio de busca de operandos. No segundo caso, o segundo operando vem do estágio de escrita de resultados.

A JVM (*Java Virtual Machine*) implementa cada método em um único *frame*, localizado na pilha. O *frame* é composto pela área de armazenamento de variáveis locais e pela pilha de operandos. A microarquitetura do FemtoJava Low-Power (Figura 3.3) é caracterizada pela presença de um banco de registradores que faz o papel da área de armazenamento de variáveis locais e da pilha de operandos. Então, o acesso externo à memória é reduzido, economizando o consumo de potência.

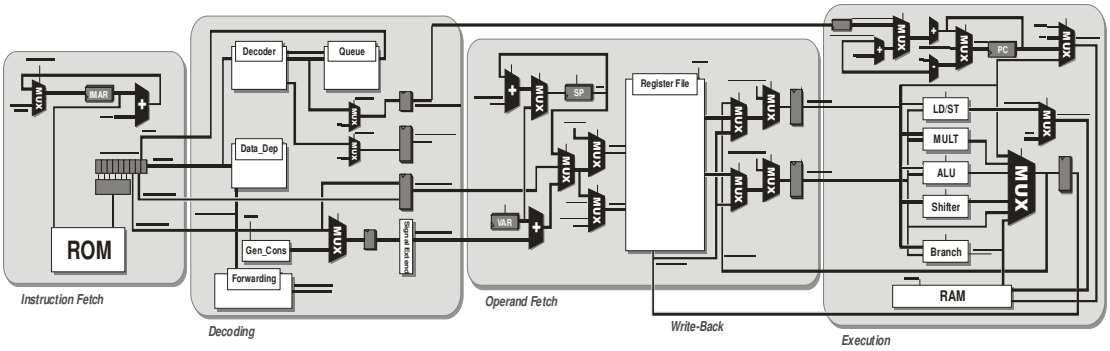


Figura 3.3: A microarquitetura do FemtoJava Low-Power

Estágio de Busca de Instruções

O primeiro estágio do pipeline do FemtoJava Low-Power é o estágio de busca de instruções (*Instruction Fetch* – IF). Esse estágio obtém as instruções requisitadas da memória através de uma palavra de 32 bits. O estágio IF é basicamente composto por uma fila de instruções de 9 registradores com 1 byte cada, um registrador para o endereçamento de memória da instrução (IMAR -*Instruction Memory Address Register*) e um somador para endereçar a próxima instrução seqüencial. Se um novo endereço não-seqüencial precisa ser carregado, um multiplexador carrega o valor armazenado pelo registrador contador de programa (PC – *Program Counter*) no IMAR.

O mecanismo de busca pode realizar uma pré-busca de *bytecodes*, minimizando assim o tempo em que o pipeline fica parado esperando por instruções. Então, o número de registradores que foi escolhido para manter o pipeline trabalhando o maior tempo possível com *bytecodes* Java pode ter dois operandos imediatos seguidos de uma instrução. Entretanto, se um endereço não-seqüencial tiver que ser carregado (se um desvio foi tomado, por exemplo), a fila de instruções terá que ser esvaziada. Nesse caso, devem-se esperar três ciclos para que o novo endereço possa ser carregado. Como um aumento no tamanho da fila de instruções pode causar grande impacto na taxa de CPI, usou-se para ela o menor tamanho possível no FemtoJava Low-Power.

Quando pelo menos quatro registradores na fila de instruções estão vazios, uma nova palavra de 32 bits é trazida da memória de instruções, apontada pelo IMAR. A primeira instrução da fila é enviada para o estágio de decodificação. O diagrama de blocos do estágio IF é mostrado na Figura 3.4.

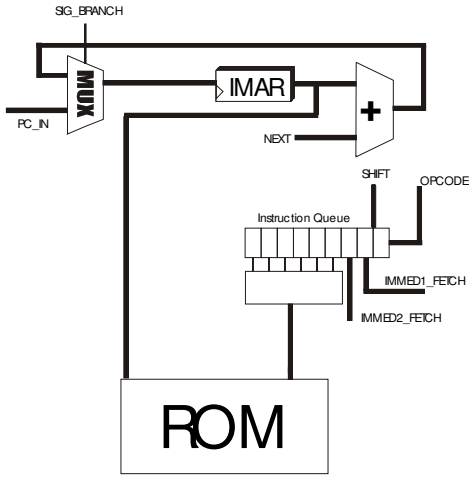


Figura 3.4: O estágio IF

Estágio de Decodificação de Instruções

O estágio de decodificação (*Instruction Decoding* – ID) tem quatro funções: gerar a palavra de controle para a instrução atual; informar para a fila de instruções o tamanho da instrução corrente, de forma a colocar a próxima instrução do fluxo de instruções no primeiro lugar da fila; iniciar a avaliação das dependências de dados e a subseqüente análise de *forwarding*. A informação do tamanho da fila de instruções é necessária porque o FemtoJava Low-Power emprega instruções de tamanho variável: elas podem ter um ou dois operandos imediatos, ou até mesmo nenhum. O diagrama de blocos do estágio ID é mostrado na Figura 3.5.

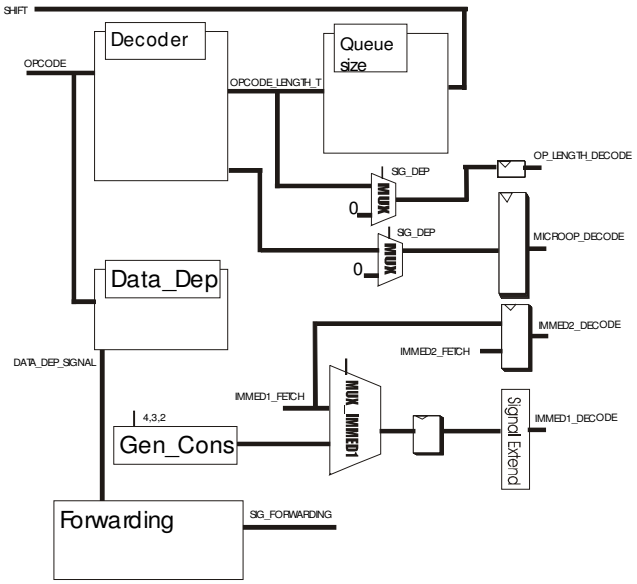


Figura 3.5: O estágio ID

Estágio de Busca de Operandos

O estágio de busca de operandos (*Operand Fetch* – OF) é onde os operandos são selecionados para o estágio de execução. Ele é basicamente composto por um banco de registradores de duas portas, que realiza duas leituras independentes ou uma leitura e uma escrita em um mesmo ciclo. Por meio de aplicações de *benchmark* ficou demonstrado que o uso de um banco de registradores com até 32 posições era o suficiente. Considerando que o processador é um ASIP, o tamanho do banco de registradores pode ser definido a priori em estágios iniciais do projeto através de simulação. A pilha e o *pool* de variáveis locais dos métodos estão disponíveis no banco de registradores. Dois registradores, o SP (*Stack Pointer*) e o VARS, que são implementados separadamente do banco de registradores, apontam para o topo da pilha e para o inicio da área de variáveis locais, respectivamente. Dependendo da instrução, um deles é usando como base para a busca de operandos. O diagrama de blocos do estágio OF é mostrado na Figura 3.6. Como pode ser observado, multiplexadores adicionais são necessários já que os operandos podem vir do banco de registradores, de *bytecodes* imediatos, ou de um *forwarding* do estágio WB (sinal result_exec) e/ou do estágio EX (exec_result_signal).

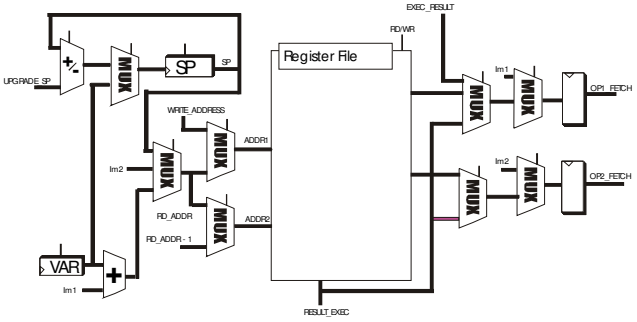


Figura 3.6: O estágio OF

Estágio de Execução

O estágio de execução (*EXecution* – EX) realiza todos os cálculos. Ele é composto de uma ALU (*Arithmetic and Logic Unit*) capaz de executar funções de soma/subtração e funções booleanas lógicas. Unidades funcionais de multiplicação, *load/store*, deslocamento, e desvios são também implementadas nesse estágio, mas em blocos diferentes. Isso é feito para proporcionar maior flexibilidade ao desenvolvedor durante a exploração do espaço de projeto nos estágios iniciais do projeto. Isso significa que o desenvolvedor pode selecionar as unidades funcionais desejadas de acordo com os requisitos da aplicação, aumentando dessa forma as possibilidades para otimização de área. O diagrama de blocos do estágio EX é mostrado na Figura 3.7.

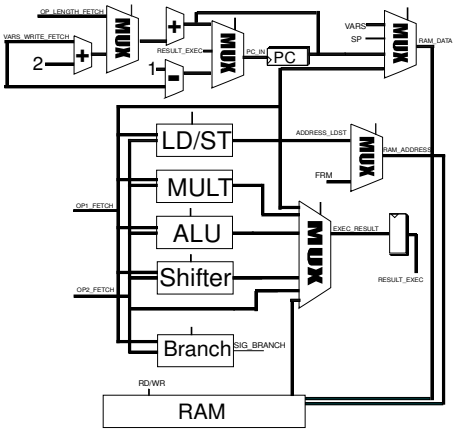


Figura 3.7: O Estágio EX

Estágio de Escrita de Resultados

O estágio de escrita de resultados (*Write Back* – WB) salva, se necessário, o resultado do estágio de execução de volta no banco de registradores, usando como base os registradores SP ou VARS. Há um banco de registradores unificado para a pilha e o *pool* de variáveis locais, porque isso facilita a chamada e retorno de métodos, aproveitando-se da especificação da JVM, onde cada método é localizado por um ponteiro de *frame* (FRM) na pilha. O diagrama de blocos do estágio WB é mostrado na Figura 3.8. O registrador VAR é repetido nesse estágio apenas para propósitos didáticos.

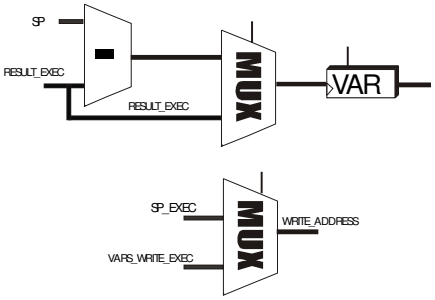


Figura 3.8: O estágio WB

3.2.3 A Organização de Memória do FemtoJava

Todas as versões do FemtoJava utilizam a mesma organização de memória. Ela é baseada na alocação de *frames* como manda a especificação da linguagem Java, e pode ser observada na Figura 3.9a (note que a pilha aumenta de cima para baixo, neste exemplo). O esquema implementado pelo Femtojava é compatível com a especificação Java e pode ser observado na Figura 3.9c. Comparado com outros processadores como picoJava (O'CONNOR et al., 1997) (Figura 3.9b), a alocação de *frames* no Femtojava é bem mais simples. Como o Femtojava não suporta *multithreading*, não é necessário guardar outros campos de informação no frame como o picoJava faz. Informações sobre monitores, vetores de métodos e da *constant pool* foram removidas.

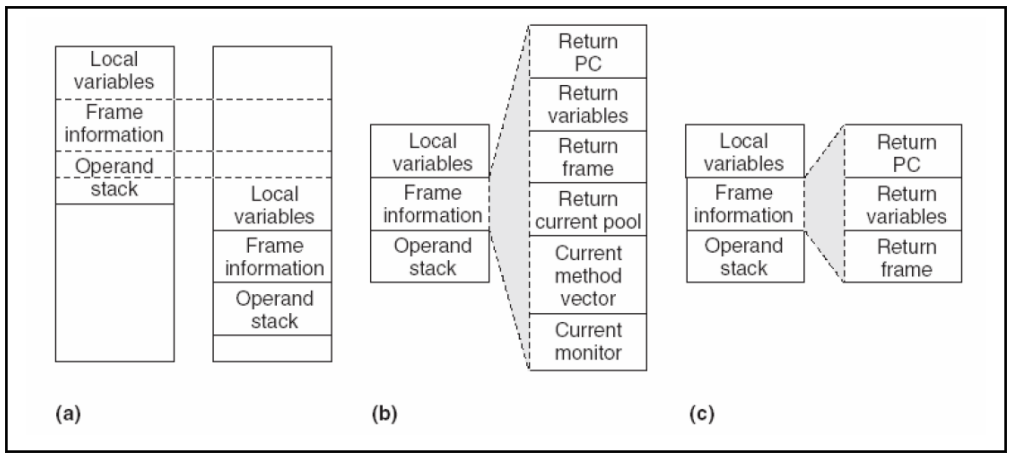


Figura 3.9: Funcionamento da memória no Femtojava Multiciclo (ITO et al., 2001)

A JVM ou um processador Java com recursos suficientes para um RTOS (*Real-Time Operating System*) podem organizar o programa e seus dados em memória utilizando técnicas já conhecidas de orientação a objetos. Entretanto, as características dinâmicas da linguagem Java nem sempre são suportadas. Por essa razão, no FemtoJava, foram definidos esquemas específicos para organizar as memórias de dados e instruções para obter uma simples implementação de hardware. Essa implementação provê capacidades para mapeamento de portas de I/O e mapeamento estático do código da aplicação.

A memória de dados é organizada como mostra a Figura 3.10a. Seu espaço inicial, da posição 00H até 10H, contém alguns registradores mapeados em memória. Esses registradores são reservados para interrupções e programação de *timer*, e ainda para operações de I/O. O restante da memória é para armazenamento de campos de classes (variáveis e constantes) e alocação de *frames* para os métodos Java.

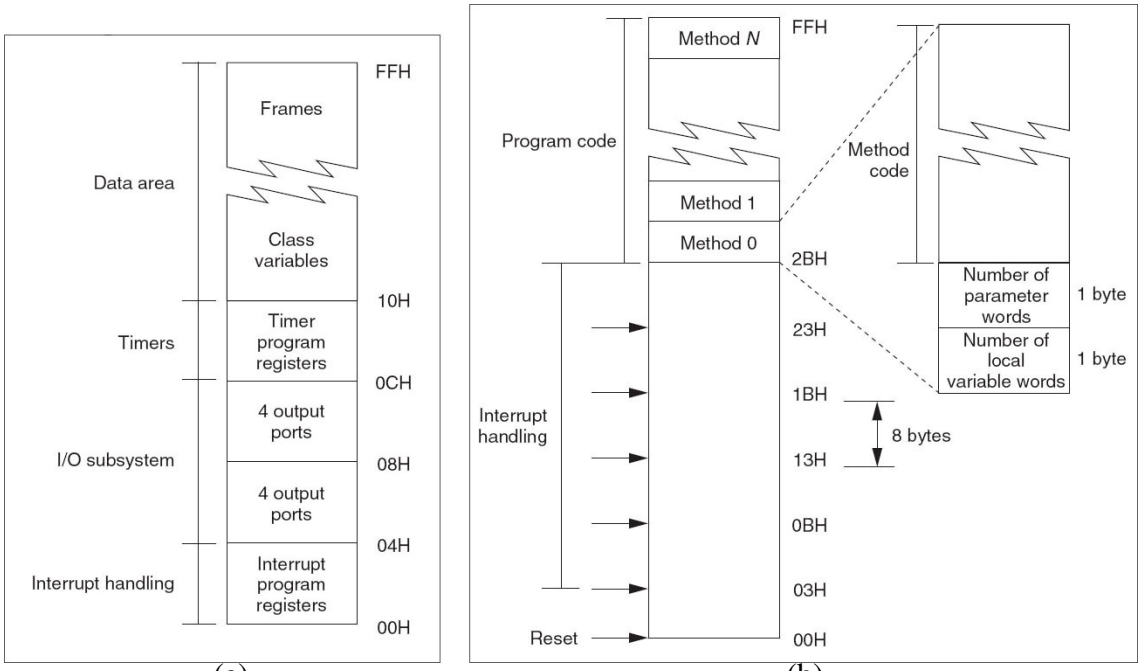


Figura 3.10: Organização da memória de dados e da memória de instruções do FemtoJava (ITO et al., 2001)

As técnicas usadas para mapear as informações de código e classes em memória de instruções devem ser cuidadosamente consideradas para produzir estruturas de hardware simples. Para o FemtoJava, o mapeamento da memória de instruções é ilustrado na Figura 3.10b. Na primeira parte da memória, é armazenado o código necessário para a chamada de métodos e manipulação de pedidos de interrupção. Para acelerar as chamadas de métodos, são armazenadas as informações no cabeçalho do método. Esses campos fazem com que o SP salte sobre a pilha e recupere corretamente os *frames* quando instruções *return* e *ireturn* executam.

3.3 Sashimi

A ferramenta Sashimi propõe a especificação de aplicações embarcadas através de uma linguagem simples, fornecendo implementação de hardware através de diferentes versões ASIP (*Application Specific Instruction set Processor*) do processador FemtoJava, que podem ser sintetizadas em um simples FPGA (*Field-Programmable Gate Array*). Usando o Sashimi, o projetista pode modelar, simular, e construir uma implementação de sistema diretamente em Java.

O ambiente do Sashimi provê bibliotecas que aumentam a precisão da simulação e permitem mapeamento direto das classes usadas pela simulação para o código final da implementação. Essas classes pré-definidas também cobrem todos os detalhes requeridos para gerar a interface do FemtoJava com o mundo externo (programação do mecanismo de interrupção, comunicação com tela de LCD, teclados, interface serial, etc.).

A Figura 3.11 ilustra o processo de projeto a partir do código Java até o chip do ASIP sintetizado. No ambiente do Sashimi, o usuário inicia com arquivos Java representando o código da aplicação. Nessa fase, o processo de desenvolvimento segue

o tradicional ciclo edição–compilação–execução de um computador *desktop* com o JDK (*Java Development Kit*) padrão. Nesse cenário, executar a aplicação é equivalente a simular a mesma em um hardware ainda não disponível, que nesse caso, é emulado pelo interpretador Java. Durante a simulação, o projetista pode usar classes pré-definidas para modelar o comportamento dos periféricos necessários. Mais tarde, no passo de síntese, o sistema irá substituir essas classes pelo código provendo a interface com componentes reais.

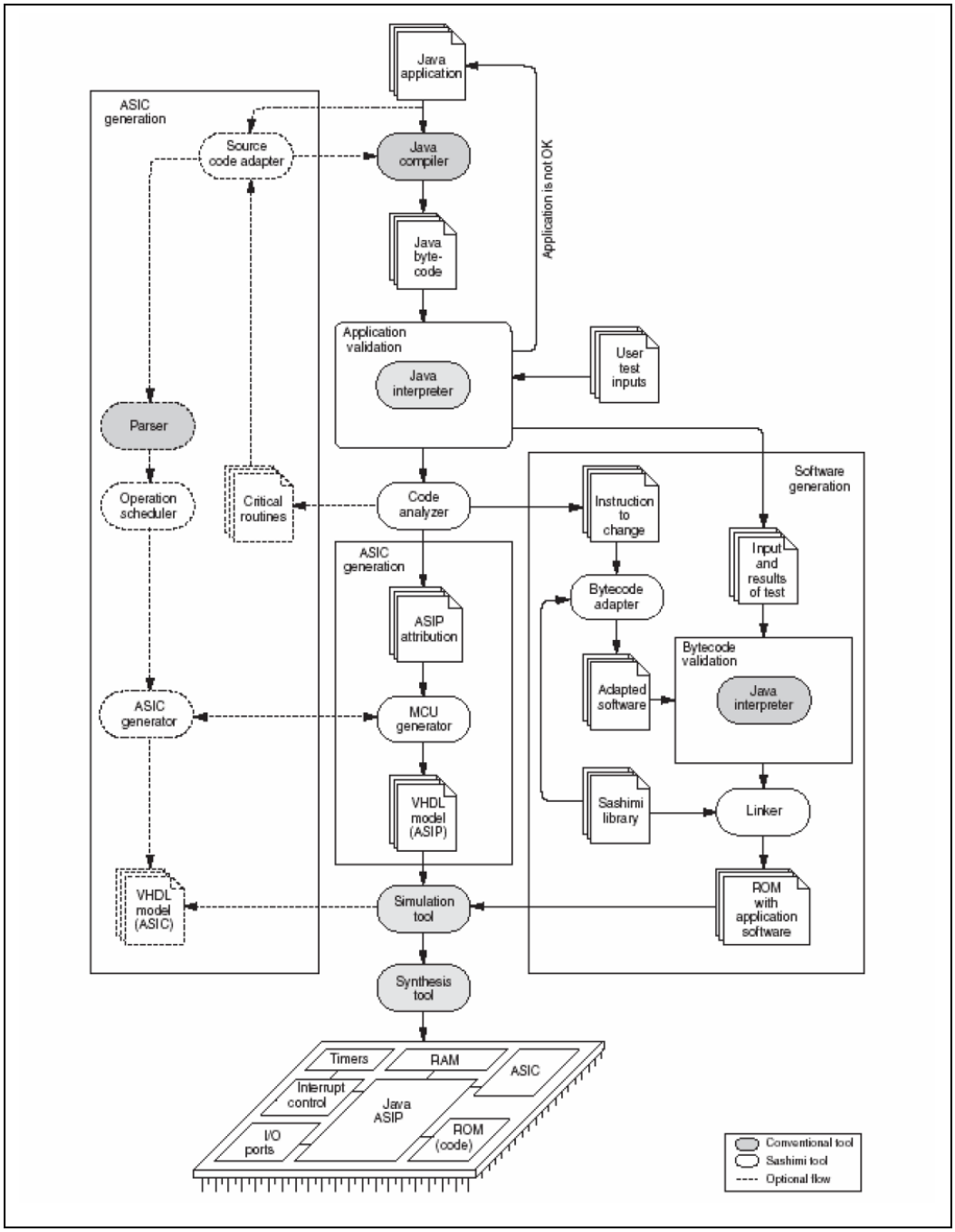


Figura 3.11: O fluxo de projeto automatizado do Sashimi para a geração ASIP do FemtoJava

Quando o projetista considera que a aplicação está corretamente modelada, os vetores de entrada fornecidos pelo usuário e os vetores de saída gerados pela simulação são salvos para serem usados mais tarde na fase de validação do *bytecode*.

A ferramenta de análise pegará os arquivos executáveis (arquivos class) para estimar dados de desempenho e área do hardware. Essa ferramenta pode estimar o desempenho usando as informações de chamadas de método, número de instruções por método, ciclos por instrução, e provável freqüência a ser atingida depois da síntese. Uma vez que o FemtoJava não suporta o conjunto completo de instruções da JVM, a fase de análise do código deve fornecer a informação para a geração do ASIP e adaptação do código.

A fase de adaptação da aplicação envolve algumas transformações nos arquivos class. Esse processo transforma instruções complexas em uma seqüência de instruções simples. As estruturas dos arquivos class mantêm o tamanho máximo da pilha para cada método da classe, e essa informação deve ser recalculada para permitir a correta adaptação do código. Depois desse passo, uma nova simulação pode ser feita, com as ferramentas disponíveis, para validar a transformação do *bytecode*. Esse processo usa os vetores de teste anteriormente armazenados e é realizado automaticamente.

O processo de substituição de *bytecodes* se baseia em critérios, tais como as instruções suportadas pelo processador alvo, os requisitos de área e desempenho, a taxa de uso de cada instrução, e a memória disponível para armazenar o código da aplicação. Nesse ponto do fluxo de projeto, um conjunto de arquivos class é adaptado para ser compatível com o conjunto de instruções específico que será implementado no ASIP do FemtoJava. Finalmente, as informações desnecessárias são removidas e a hierarquia do arquivo class é resolvida. O *linker* conecta e converte o código da aplicação e as bibliotecas em uma única imagem de memória de programa, memórias RAM (*Random Access Memory*) e ROM (*Read Only Memory*). Adicionalmente, os *bytecodes* inúteis para a geração do software embarcado (tais como as chamadas para o método *System.out.println*, usado para escrever uma linha de saída de dados durante a simulação) são descartadas, baseando-se na informação gerada durante as fases anteriores.

Quando as estimativas de desempenho não casam com os requisitos, o projetista pode escolher descartar algumas instruções, retornando para o a ferramenta de análise para identificar o código crítico.

4 ARQUITETURA PROPOSTA

O capítulo anterior detalhou o projeto no qual essa dissertação está inserida, as versões do processador Java utilizadas no contexto desse trabalho, bem como a sua organização de memória, e ainda a ferramenta usada para gerar uma versão ASIP sintetizável do processador. Nesse capítulo serão apresentadas a arquitetura para gerenciamento de memória em MPSoCs desenvolvida nesse trabalho, as organizações de memória suportadas, a técnica DCF, e ainda o processo de validação dos componentes propostos.

4.1 Visão Geral da Arquitetura

A idéia geral desse trabalho é prover um componente parametrizável e independente de arquitetura para efetuar o gerenciamento de memória em MPSoCs heterogêneos. Em sistemas desse tipo, a comunicação entre as tarefas que executam em cada processador é um item de fundamental importância. Há basicamente duas diferentes abordagens para habilitar esse tipo de comunicação. A primeira delas é baseada na troca de mensagens entre processadores e exige o uso de um protocolo para controlar as primitivas de envio e recebimento.

A segunda abordagem usa um mecanismo de memória compartilhada, que fornece a menor latência de comunicação entre os processadores. Logo que um processador escreve em uma célula de memória, todos os outros processadores têm acesso a ela sem precisar procurar por uma célula remota.

Esse trabalho, como mostrado na Figura 4.1, baseia-se na abordagem via memória compartilhada. Todos os processadores do MPSoC têm acesso, através de um barramento, à memória compartilhada de dados e à memória compartilhada de instruções. Quando se detecta uma requisição de leitura ou escrita em espaço de memória compartilhado, redireciona-se essa requisição ao componente *Shared-D Block* (para a memória de dados) ou *Shared-I Block* (para a memória de instruções). Cada um desses componentes é composto por um árbitro do barramento e a memória compartilhada correspondente. Cada processador possui ainda uma memória privada de dados e outra de instruções. Essas memórias podem ser identificadas dentro dos componentes *Private-D Block* e *Private-I Block*, que correspondem às memórias privadas de dados e instruções, respectivamente. Entretanto, esses componentes são apenas conceituais e são utilizados apenas para facilitar a didática na explicação da arquitetura (eles não adicionam nenhum *overhead* à arquitetura). O que existe na realidade são apenas as memórias privadas de instruções e dados, que são conectadas aos respectivos componentes *D-Memory Router* ou *I-Memory Router*, sendo que cada memória privada ilustrada na figura, dentro dos componentes *Private-D Block* e *Private-I Block*, corresponde ao processador de mesma numeração.

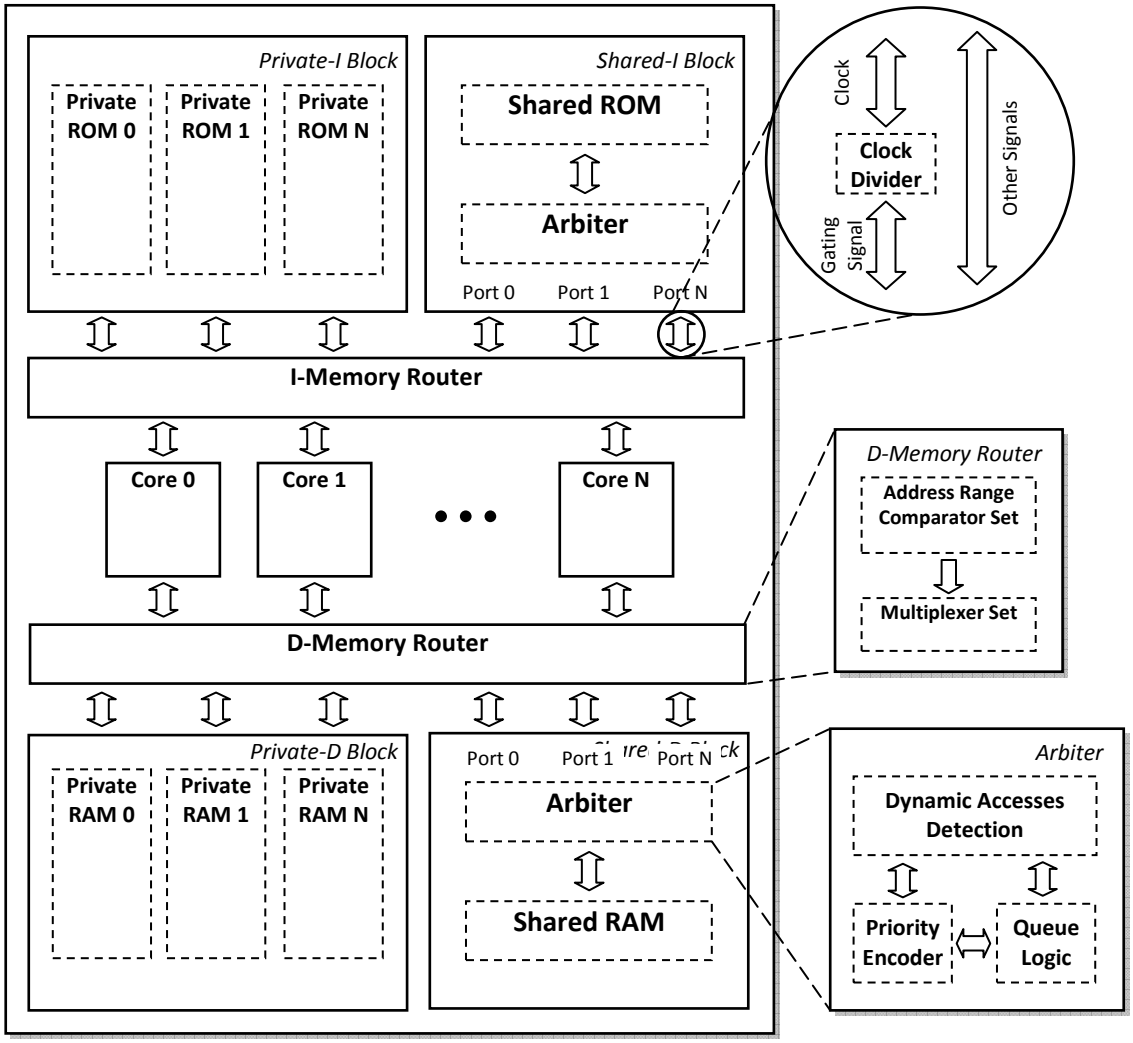


Figura 4.1: Arquitetura Proposta

Como nessa arquitetura há apenas um simples barramento, que é controlado pelo árbitro, e deve ser disputado pelos múltiplos processadores durante os acessos à memória compartilhada, surge um problema que deve ser tratado quando houver tentativas de acesso simultâneas. Para resolver esse problema foram analisadas quatro abordagens diferentes:

1. Nova instrução: Nessa abordagem cria-se uma nova instrução no ISA (*Instruction Set Architecture*) dos processadores, permitindo assim que o estado de seus elementos seqüenciais possa ser salvo, e possibilitando que, através do disparo dessa instrução por meio do componente *Arbiter*, o processador de menor prioridade espere o de maior prioridade acessar a memória compartilhada. Entretanto, além dessa abordagem ser dependente de plataforma, ela impõe um tempo extra de verificação em cada um dos processadores modificados.
2. Instruções NOP: Usando essa abordagem, o componente *Arbiter* dispara instruções NOP (*No Operation*) para o processador de menor prioridade até a liberação do barramento. Entretanto, o *overhead* dessa abordagem é alto, pois é necessário salvar o contexto da aplicação que estava executando no processador antes de dar lugar ao bloco de instruções NOP. Apesar dessa abordagem não exigir modificações arquiteturais nos processadores, o tempo de computação gasto pelo

processador que precisa esperar pelo acesso à memória, bem como a energia gasta pelo mesmo na execução das instruções NOP, são completamente desperdiçados.

3. Interrupção: Usando interrupção, o árbitro pode dinamicamente forçar o processador de menor prioridade a executar uma determinada rotina, enquanto espera pela liberação do barramento. Da mesma forma que a abordagem anterior, essa abordagem não exige modificações na arquitetura dos processadores. Todavia, além de ser necessário salvar o contexto da aplicação que estava executando, é preciso também que haja alguma rotina útil disponível para ser executada. Caso contrário, todo o tempo de computação do processador, bem como o gasto energético, serão perdidos.
4. Congelamento de estado: Essa abordagem soluciona os problemas das abordagens anteriores, e consiste em aplicar a técnica de *clock gating* no processador de menor prioridade até a liberação do barramento. Dessa forma, o estado de todos os elementos seqüenciais do processador de menor prioridade é mantido. Isso significa que, quando o relógio do processador for liberado, o processador irá retomar a execução da aplicação do ponto exato em que ela havia sido interrompida. Mais, além dessa técnica ser independente da arquitetura utilizada, ela não desperdiça energia do processador enquanto ele estiver aguardando para fazer o acesso à memória, já que não há atividade de chaveamento dos seus transistores enquanto o *clock gating* está sendo aplicado. Apesar de essa técnica ser comumente utilizada apenas para redução do consumo de potência de circuitos, ela foi adotada para evitar colisões no barramento na arquitetura proposta, como será explicado na seqüência.

4.2 Funcionamento da Arquitetura

O componente *Arbiter*, que pode ser observado com mais detalhes na Figura 4.2, possui a lógica combinacional para detectar variações nos barramentos de dados e endereços, identificando assim, os acessos à memória compartilhada. Esse componente é basicamente composto por um uma lógica de detecção de requisições de acesso à memória, um codificador de prioridade, e uma lógica para controlar a fila onde os processadores aguardam para realizar o acesso à memória. A lógica de detecção de requisições de acessos à memória é composta por um conjunto de comparadores e registradores que é replicado para cada um dos processadores do MPSoC e serve para efetuar comparações entre os sinais de conexão do processador com à memória do ciclo atual com os do anterior. Dessa forma, se podem identificar variações nesses sinais que ocorrem entre ciclos de execução adjacentes, o que caracteriza o acesso à memória. Mais, o uso dessa técnica dispensa qualquer modificação arquitetural nos processadores do MPSoC.

O codificador de prioridade é usado para selecionar o processador que efetuará o acesso à memória no ciclo corrente, sendo que essa seleção é feita com base na porta de conexão do processador com o componente *Shared-D Block*, a partir de uma lógica completamente combinacional, a qual garante que o processador realize o acesso de forma transparente.

Finalmente, a lógica que controla a fila alimenta o codificador de prioridade com os processadores que estão esperando para efetuar o acesso à memória, sendo também responsável por atualizar o estado da fila a cada ciclo. Além disso, essa lógica também é responsável pela geração dos sinais de *clock gating* para congelar o estado dos processadores que irão esperar para efetuar o acesso à memória no ciclo seguinte.

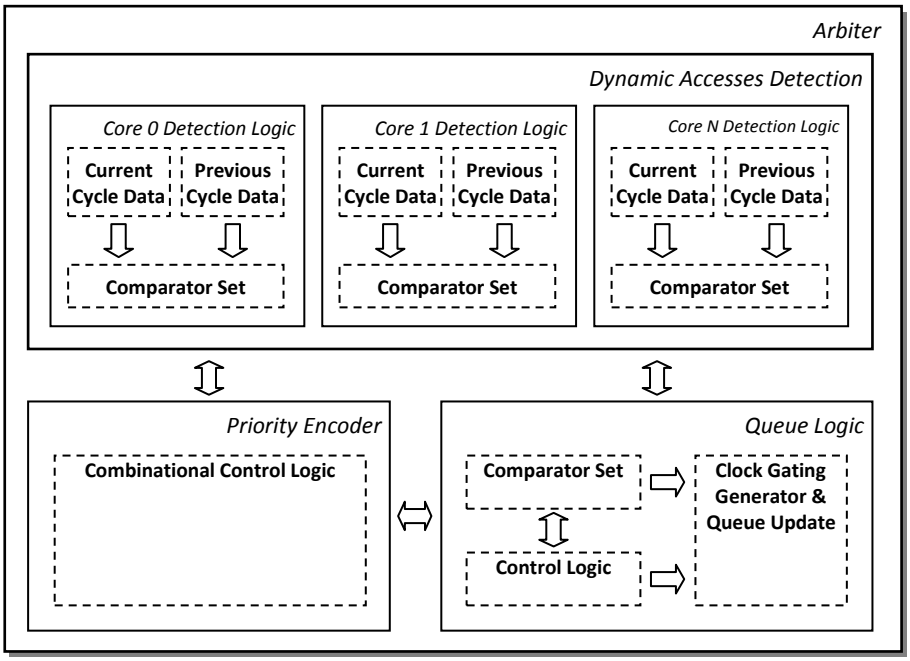


Figura 4.2: Detalhamento do Árbitro

Dessa forma, quando requisições de acessos simultâneas à memória são detectadas, o processador de mais alta prioridade realiza, transparentemente, a operação. Enquanto isso, os demais processadores têm seus elementos seqüenciais congelados através do *clock gating*. Os níveis de prioridade mais altos são alocados às portas de conexão mais baixas nos componentes *Shared-D Block* e *Shared-I Block*. Dessa forma, pode-se garantir maior previsibilidade e uniformidade para as portas de conexão mais baixas nos acessos à memória compartilhada. Os processadores congelados pelo *clock gating* aguardam na fila de espera até que o processador de mais alta prioridade conclua o acesso à memória. Quando isso acontece, o primeiro processador da fila é escalonado para realizar o acesso à memória, se não houver, no mesmo ciclo, requisições de acesso à memória de outros processadores com prioridade mais alta.

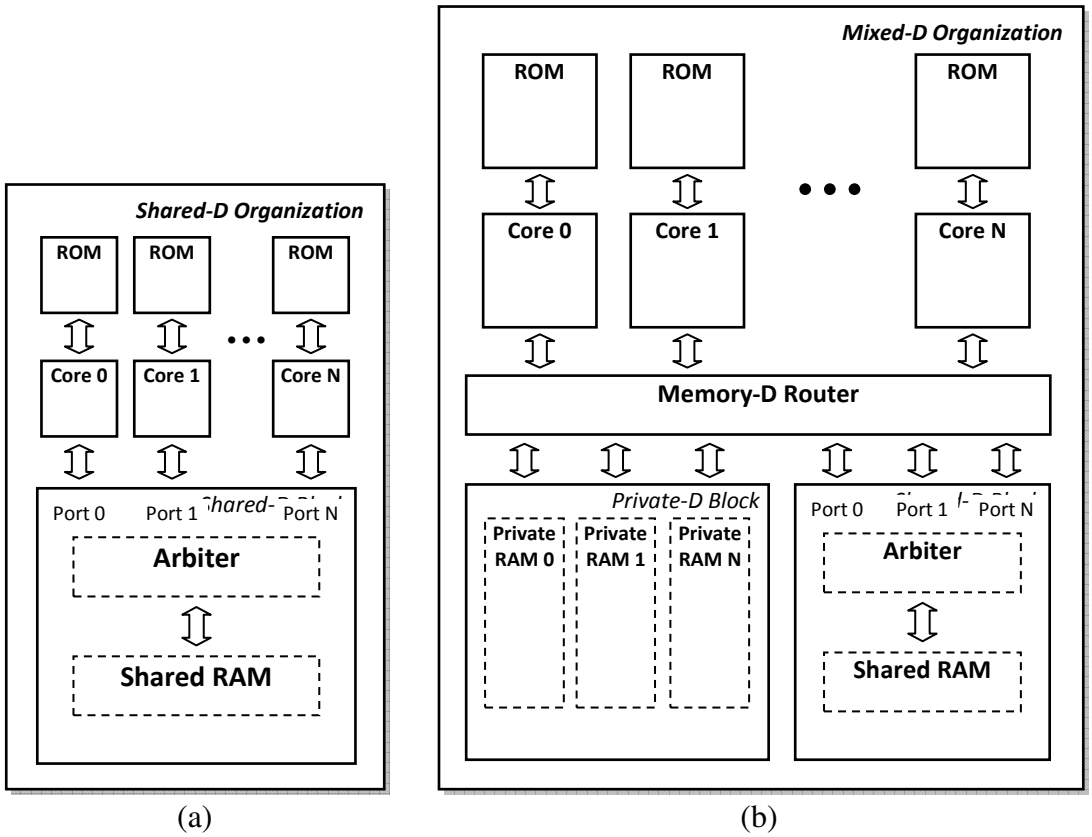
As colisões são evitadas automaticamente por meio do *clock gating*, que é disparado no mesmo momento em que os múltiplos acessos à memória são detectados. O sincronismo no disparo do *clock gating* é garantido com o uso de divisores de relógio, que são conectados na entrada do sinal de relógio de cada processador, permitindo, dessa forma, que os processadores executem em freqüências diferentes sem prejudicar a temporização global do MPSoC. Para isso, entretanto, o componente *Arbiter* deve executar na mesma freqüência do processador mais rápido, enquanto que a memória compartilhada deve executar na mesma freqüência do processador mais lento. Todos os processadores têm suas próprias memórias privadas de dados e instruções, sendo que a seleção da memória correta é realizada pelos componentes *D-Memory Router* ou *I-Memory Router* (para memória de dados e instruções, respectivamente), por meio de uma análise na faixa de endereçamento do acesso. Esses componentes são compostos basicamente por um conjunto de comparadores que geram os sinais de controle para um conjunto de multiplexadores, que por sua vez têm conexões com todas as memórias. Além disso, é importante ressaltar que as memórias privadas de instruções são do tipo *scratchpad* e devem ter seu conteúdo carregado estaticamente.

A arquitetura proposta oferece ainda a infra-estrutura necessária para habilitar migração de tarefas entre os processadores do MPSoC, considerando-se que o contexto das tarefas deve ser salvo em espaço compartilhado. Entretanto, mecanismos específicos para disparo e controle da migração de tarefas precisam ser introduzidos e eles estão fora do escopo desse trabalho.

4.2.1 Organizações de Memória

A arquitetura proposta dá suporte a quatro organizações de memória: duas para as memórias de dados e outras duas para as memórias de instruções. Além disso, as organizações de memória podem ser usadas isoladamente ou combinadas.

A primeira organização de memória de dados é a Shared-D, conforme é mostrado na Figura 4.3a. Essa organização inclui o componente *Shared-D Block*, de forma que somente a memória compartilhada é visível aos processadores. Desse modo, apesar de o *overhead* de memória ser extremamente pequeno, o desempenho dos processadores acaba comprometido, pois como todos os acessos são realizados sobre a mesma memória, o número de contenções do barramento cresce junto com o número de processadores do MPSoC.



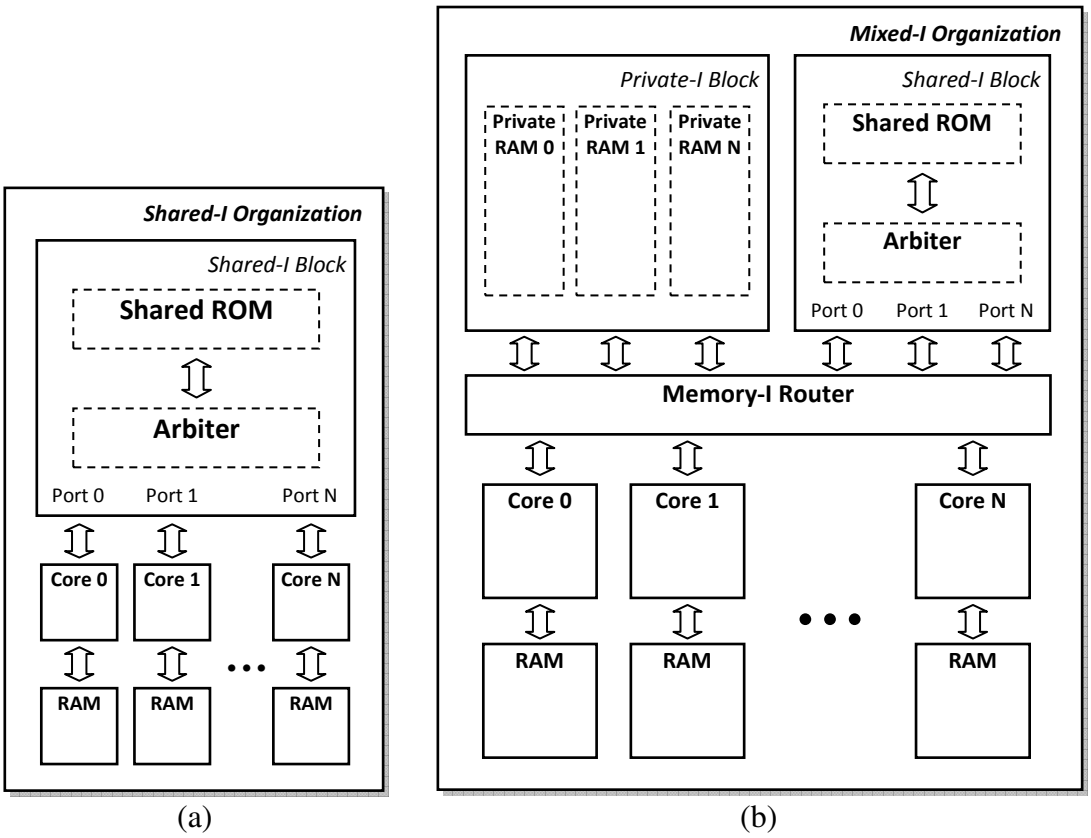
(a) (b)

Figura 4.3: Organizações de memória de dados

A segunda organização de memória de dados, mostrada na Figura 4.3b, é a Mixed-D. Essa organização inclui os componentes *Private-D Block*, *Shared-D Block*, e *D-Memory Router*. Cada acesso à memória é realizado ou na memória privada ou na memória compartilhada, por meio da faixa de endereçamento usada. Nessa organização, embora os processadores estejam conectados através do mesmo barramento, o

desempenho dos núcleos somente será reduzido quando houver acessos simultâneos à memória compartilhada.

De maneira similar, a primeira organização de memória de instruções é a Shared-I, ilustrada na Figura 4.4a, que inclui o componente *Shared-I Block*. Como esperado, todas as referências à memória são em espaço compartilhado. Finalmente, a segunda organização de memória de instruções é a Mixed-I, ilustrada na Figura 4.4b, que contém os componentes *Private-I Block*, *Shared-I Block*, e *I-Memory Router*. Na organização Mixed-I, os acessos à memória são multiplexados ou para a memória privada ou para a memória compartilhada.



(a) (b)

Figura 4.4: Organizações de memória de instruções

Embora tenham a mesma funcionalidade, o custo em área do componente *Shared-I Block* é menor que o do componente *Shared-D Block*, visto que as memórias de instruções não podem ser escritas. Dessa forma, o componente *Shared-I Block* não necessita de todas as conexões utilizadas pelo componente *Shared-B Block*. De maneira similar, o custo do componente *I-Memory Router* é menor que o do componente *D-Memory Router*.

4.2.2 Dynamic Core Freezing

A técnica *Dynamic Core Freezing* é usada para aumentar a eficiência energética do MPSoC. Seu funcionamento consiste na monitoração dinâmica dos pinos de conexão do processador com a memória, de forma a detectar qualquer requisição de acesso. Dessa forma, pode-se disparar dinamicamente o *clock gating*, aproveitando-se dos ciclos ociosos dos processadores durante os acessos à memória, eliminando assim o consumo

de potência dinâmica do processador nesses ciclos. O disparo do *clock gating* é realizado pelo componente *Arbiter* e ocorre no ciclo seguinte ao ciclo em que houve a detecção do acesso à memória. O DCF pode aumentar drasticamente a eficiência energética dependendo da relação de freqüência entre o processador selecionado e a memória e da taxa de comunicação exigida pela aplicação (aplicações altamente *data flow* tendem a ser fortemente beneficiadas com o uso dessa técnica). Pode-se ainda aplicar o DCF nos acessos às memórias privadas do MPSoC, aumentando ainda mais a eficiência energética.

Entretanto, como é perdido um ciclo para se realizar a detecção do acesso, o DCF não pode ser aplicado se o processador e a memória estiverem executando na mesma freqüência. Exemplificando: se o processador que está realizando o acesso está executando em uma freqüência quatro vezes maior que a da memória (ou seja, são necessários quatro ciclos para completar o acesso), o DCF pode anular o consumo energético do processador nos três ciclos seguintes ao ciclo em que o acesso foi detectado.

Além disso, é importante destacar que o uso do DCF não adiciona nenhum *overhead* à arquitetura proposta, pois ele utiliza a mesma lógica de controle necessária para implementar o mecanismo que impede as colisões do barramento no componente *Arbiter*.

4.2.3 Configuração dos Componentes

Todos os componentes da arquitetura proposta foram descritos em VHDL e são independentes de plataforma. O número de linhas de código necessário para implementar todos os componentes da arquitetura, mostrados na Figura 4.1, é de aproximadamente 3.000 (não computando, naturalmente, o número de linhas de código dos processadores), sendo que a descrição foi realizada no estilo estrutural. Além disso, os processadores conectados à arquitetura não necessitam de qualquer adaptação arquitetural, pois o processo de detecção das requisições de acesso à memória é realizado com base na análise dos sinais externos de conexão entre o processador e a memória.

Mais, todos os parâmetros de configuração da arquitetura proposta, tais como o número de processadores, a largura dos barramentos de dados e endereços, faixas de endereçamento pertencente às memórias de dados e instruções privadas e compartilhadas, e ainda as constantes de sincronismo dos divisores de relógio podem ser configurados em único um arquivo de configuração em VHDL. Dessa forma, é realizada a instanciação automática de todos os componentes necessários para implementar o MPSoC desejado. Esse processo utilizado para a configuração permite que a inserção ou substituição de processadores do MPSoC ocorra de uma maneira bastante simples e sem necessidade de re-escrita de código.

4.3 Validação da Arquitetura

Para realizar a validação da arquitetura descrita na seção anterior utilizou-se o problema do produtor–consumidor (TANENBAUM, 2003), onde são implementados dois processos que compartilham um *buffer* de dados de tamanho fixo. Enquanto que o processo produtor insere informações no *buffer*, o processo consumidor remove informações do mesmo. Para modelar esse problema na arquitetura proposta, foram criadas duas aplicações Java, uma contendo o código do processo produtor, e outra contendo o código do processo consumidor. A partir dessas aplicações foram gerados

dois ASIPs do processador FemtoJava (um na versão Multiciclo e o outro na versão Low-Power), com o auxílio da ferramenta Sashimi (descrita na Seção 3.3). No ASIP Multiciclo usou-se a aplicação com o processo consumidor, e no ASIP Low-Power usou-se a aplicação com o processo consumidor. Além disso, utilizou-se a organização de memória Shared-D, conforme mostrado na Figura 4.5, sendo que a memória compartilhada foi populada com o *buffer* de dados. Mais, para esse experimento, o codificador de prioridade do componente *Arbiter* foi modificado para escolher randomicamente o processador com a preferência no acesso à memória, evitando assim uma situação de *starvation* no processo alocado ao processador conectado à porta mais baixa do componente *Shared-D Block*. Nas simulações realizadas se considerou os processadores e as memórias executando em uma freqüência de 12 MHz.

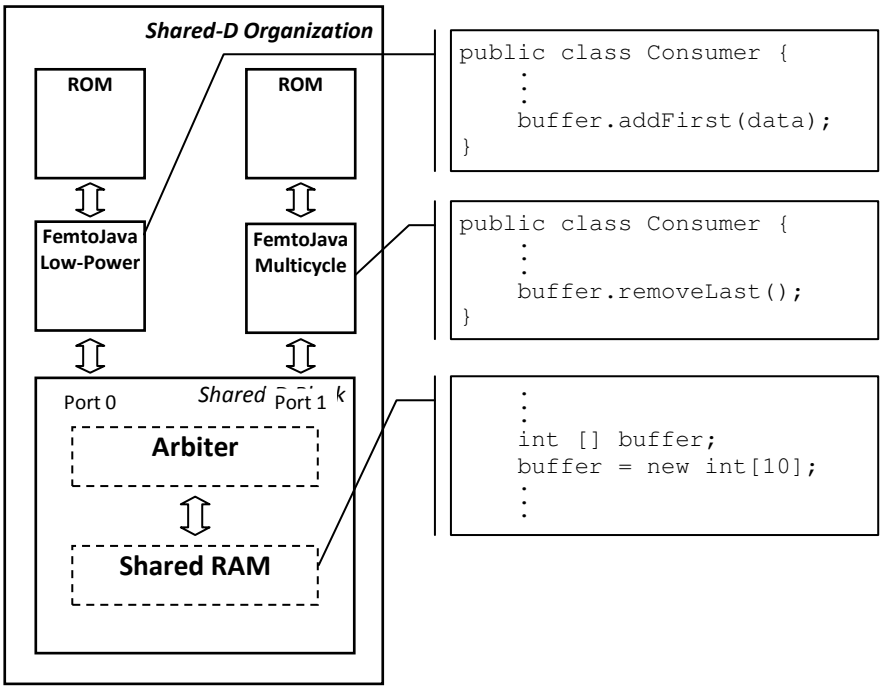


Figura 4.5: A validação da arquitetura proposta

A Figura 4.6 apresenta uma forma de onda obtida da simulação do experimento descrito anteriormente. A simulação foi realizada através do simulador Active HDL (ALDEC, 2008). Na figura, o primeiro e segundo sinais representam os sinais de relógio dos processadores FemtoJava Low-Power e FemtoJava Multiciclo, respectivamente. As setas posicionadas sobre a forma de onda do sinal de relógio do processador multiciclo indicam os diferentes momentos onde houve requisições simultâneas de acesso à memória compartilhada que resultaram na aplicação do *clock gating*. Os demais sinais correspondem às conexões do componente *Shared-D Block*. É interessante observar os sinais *core_select* e *core_gating*. O primeiro corresponde ao sinal de seleção do processador que ganha o acesso ao barramento, sendo que ele é responsável pelo controle de todos os multiplexadores do componente *Shared-D Block*. Já o segundo é o sinal responsável pela geração do *clock gating* para cada processador do MPSoC, e é conectado externamente a uma porta AND, juntamente com o sinal de relógio principal. A saída dessa porta AND, por sua vez, é conectada ao divisor de relógio do processador correspondente. Deve-se notar ainda que o sinal de relógio da memória, denominado *clock_ram*, é mais rápido que o sinal de relógio dos processadores. Isso acontece porque

o processador FemtoJava foi validado usando um dispositivo FPGA, e a memória desse FPGA executava no dobro da freqüência de relógio do processador.

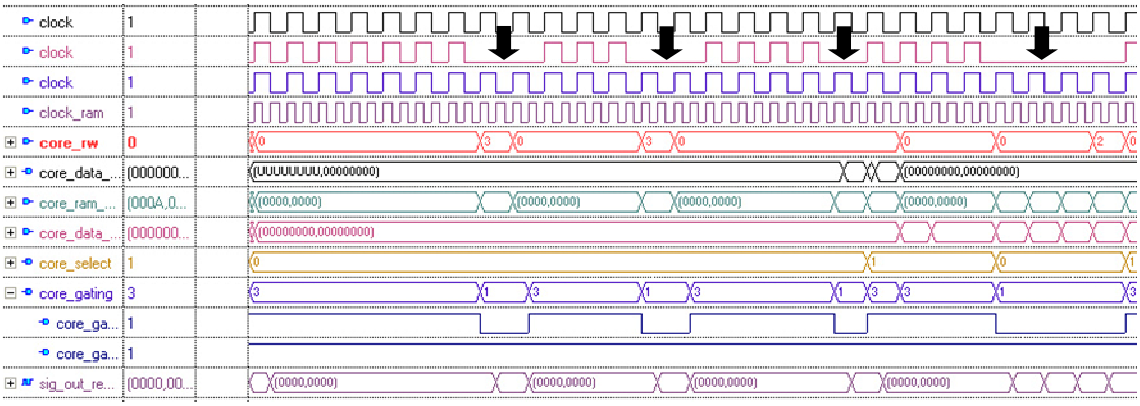


Figura 4.6: Formas de onda da simulação

5 O SIMULADOR HASHI

O capítulo anterior detalhou a arquitetura proposta nessa dissertação. Foram abordados o gerenciamento de memória realizado através do *clock gating*, as diferentes organizações de memória disponibilizadas, o uso da técnica DCF para aumento da eficiência energética do MPSoC através do disparo dinâmico do *clock gating*, as questões de configuração dos componentes da arquitetura, e ainda o experimento para validação arquitetural. Nesse capítulo será apresentado o simulador de MPSoCs Heterogêneos que implementa uma versão em alto nível da arquitetura proposta. Ele foi usado na extração dos resultados dessa dissertação, que serão apresentados no próximo capítulo. A Seção 5.1 detalha o simulador, explicitando as suas fases de execução e mostrando alguns dos trechos de código relevantes para o processo de simulação. Na Seção 5.2 é apresentada a interface do simulador, detalhando quais informações, relativas à execução do MPSoC, podem ser extraídas.

5.1 Detalhamento do Simulador

O Hashi é um simulador de MPSoCs heterogêneos que implementa uma versão em alto nível da arquitetura descrita no capítulo anterior. O simulador permite a criação de MPSoCs com um número variável de processadores executando em freqüências distintas, bem como a simulação com diferentes organizações de memória. É possível ainda estimar a eficiência energética do *clock gating* no gerenciamento da memória compartilhada, e também da técnica DCF sobre os acessos às memórias privadas e memória compartilhada. A simulação é realizada a partir da análise de arquivos de *trace* de execução de aplicações Java, que devem ser informados como entrada para o simulador. O simulador realiza então um mapeamento das instruções encontradas nos arquivos de *trace* para diferentes classes de instruções. Esse mapeamento é utilizado posteriormente para geração de instruções aos processadores do MPSoC, durante a fase de simulação. Entretanto, o simulador exige a modelagem de informações a respeito da execução de cada classe de instrução em cada um dos processadores utilizados, bem como a especificação de detalhes arquiteturais dos processadores simulados. Além disso, é importante destacar que o simulador permite a simulação de MPSoCs com as diferentes organizações de memória de dados, descritas no capítulo anterior, mas considera que as memórias de instruções são sempre privadas.

O simulador foi desenvolvido inteiramente em linguagem C++. A sua interface permite uma visualização detalhada das informações a respeito do estado corrente de cada processador, bem como informações gerais sobre a execução do MPSoC. O Hashi foi compilado e avaliado em uma máquina com o sistema operacional Windows XP usando a ferramenta Borland C++ Builder 6.0. Ele tem aproximadamente 900 linhas de código, sendo que o tempo de simulação, usando 256 processadores, e considerando

ainda um computador com um processador Athlon XP de 3.8, e com 1 GB de memória RAM, é menor que 5 minutos.

A Figura 5.1 mostra o fluxo de execução do simulador. Como se pode observar, o fluxo é composto por duas fases. Na primeira delas é realizada a extração do comportamento da aplicação, baseado nos arquivos de *trace* de execução informados. Na segunda fase é realizada a simulação do MPSoC e a geração dos resultados. Essas fases serão detalhadas nas próximas subseções.

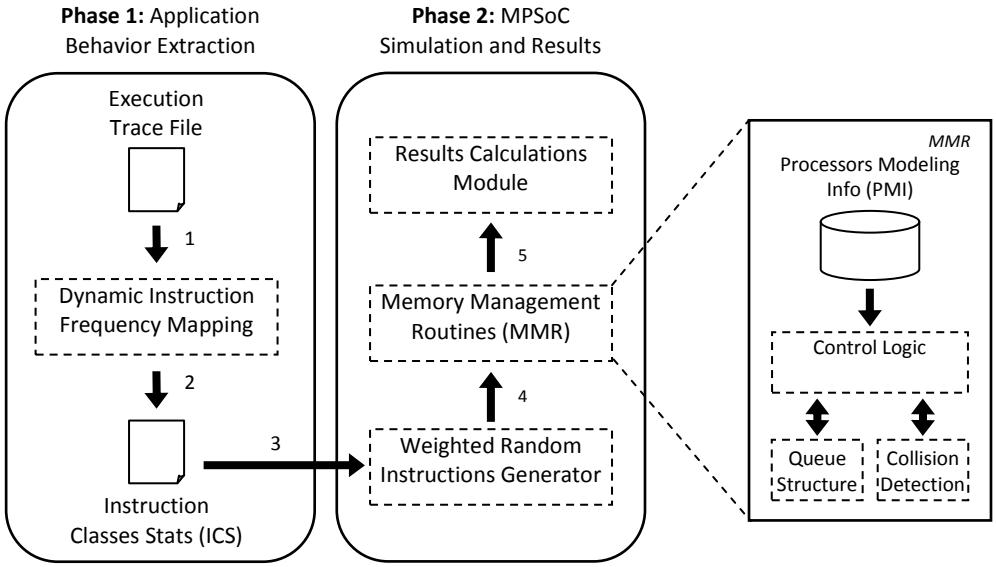


Figura 5.1: Fluxo de execução do simulador Hashi

5.1.1 Fase 1: Extração do Comportamento da Aplicação

Nessa fase deve-se fornecer ao simulador Hashi um arquivo de *trace* de execução de uma aplicação JAVA. A partir desse ponto, o simulador realizará uma análise do arquivo de *trace*, de forma a mapear cada uma de suas instruções para uma das seguintes classes de instruções: *leitura de variáveis locais*, *escrita de variáveis locais*, *leituras em memória*, *escritas em memória*, *computação*, *desvios*, *chamadas/retornos*, *empilhamento de constantes*, *operações variadas na pilha*, e *criação de objetos*. Todas as outras instruções são mapeadas para a classe *demais instruções*. Como se pode observar na Figura 5.1, o passo final dessa fase consiste na geração de um arquivo ICS (*Instruction Classes Stats*), o qual armazena o resultado do mapeamento realizado no passo imediatamente anterior.

É possível ainda fornecer múltiplos ou nenhum arquivo de *trace* de execução ao simulador Hashi. Se múltiplos arquivos forem informados, o simulador criará um arquivo ICS contendo a freqüência dinâmica média de cada classe de instrução para grupo de aplicações. Isso é bastante útil para se extrair um padrão de execução de um grupo de aplicações com características *data flow* ou *control flow*, por exemplo. Se, por outro lado, nenhum arquivo for informado, o utilizador deve informar manualmente a freqüência desejada de cada classe de instrução no arquivo ICS.

5.1.2 Fase 2: Simulação do MPSoC

Nessa fase, o simulador Hashi inicia uma simulação do MPSoC com precisão de ciclo. O primeiro passo dessa simulação consiste na geração de instruções para cada

processador do MPSoC, como pode-se observar no trecho de código simplificado do simulador mostrado na Figura 5.2. No entanto, antes de gerar de uma instrução para um processador, é necessário confirmar se ele não se encontra na fila, aguardando para efetuar um acesso à memória compartilhada, e ainda se a última instrução já terminou de executar. Essas verificações são realizadas nas linhas 01 e 03, respectivamente. Se as duas verificações forem confirmadas, a instrução pode ser gerada para o processador (linha 04). Essa geração é realizada através de uma rotina, que seleciona uma instrução randomicamente. Essa geração randômica é baseada em pesos, que correspondem às informações contidas no arquivo ICS, gerado no passo anterior. Após a geração é necessário armazenar o número de ciclos que a instrução corrente levará para executar no processador, bem como incrementar o número de instruções do MPSoC (linhas 05 e 06, respectivamente). O número de ciclos da instrução corrente é obtido do arquivo PMI (*Processor Modeling Information)*. Esse arquivo deve ser modelado, a priori, pelo utilizador, e precisa ser populado com os dados de consumo de potência, número de ciclos, e acessos à memória de cada instrução Java executando em cada processador utilizado. O arquivo PMI fica acessível internamente a todos os blocos mostrados na Figura 5.1.

A cada ciclo também deve ser somado o consumo de potência da instrução ao gasto energético do processador, bem como ao gasto energético geral do MPSoC (linhas 09 e 10, respectivamente). Ambas as informações são também obtidas do arquivo PMI.

```
01 if (!core(i).queue) {
02
03   if (!core(i).iCycles) {
04     iGeneration(core(i).id);
05     core(i).iCycles=pmi.coreVersion(instruction,cycles);
06     mpsoc.iExecuted++;
07   }
08
09   core.energy+=pmi.coreVersion(instruction,power);
10   mpsoc.energy+=pmi.coreVersion(instruction,power);
11   core(i).iCycles--;
11 }
```

Figura 5.2: Geração das instruções para os processadores do MPSoC

Em seguida, se a instrução gerada precisar acessar a memória de compartilhada, ela precisa passar pelo módulo MMR (*Memory Management Routines*), que é uma representação em alto nível do componente *Shared-D Block* (detalhado no capítulo anterior). Além disso, no módulo MMR são coletadas informações a respeito das contenções no barramento, bem como o estado atual da fila de acessos à memória compartilhada. A Figura 5.3 exibe um trecho de código simplificado que mostra algumas das ações executadas pelo módulo MMR. A linha 01 verifica se a instrução atual realiza acesso à memória. Isso acontece por meio de uma rotina que se baseia no arquivo PMI para determinar se uma determinada instrução realiza acessos à memória e também o número de ciclos necessários em cada acesso. No próximo bloco de código, da linha 03 a 09, é verificado se a memória já não está sendo acessada por algum outro processador no ciclo corrente. Se já estiver, e não for o mesmo processador que já estava efetuando o acesso no ciclo anterior, há uma contenção no barramento, devendo-se então incrementar o número de contenções no barramento, e alocar o processador corrente para a fila de espera. Deve-se ainda computar o ganho energético do uso do

clock gating, se ele estiver ativo. Caso contrário, deve-se computar o custo de propagação do sinal de relógio no processador.

Por outro lado, se a memória não estiver sendo acessada no ciclo corrente (linhas 14 a 24), deve-se verificar se o processador estava na fila. Em caso positivo, deve-se remover o processador da fila. Deve-se ainda computar o ganho energético do uso do DCF, se ele estiver ativado e não for o primeiro ciclo de acesso à memória do processador corrente. Caso contrário, deve-se computar o gasto energético. Deve-se também indicar que a memória estará em uso no próximo ciclo.

```
01 if (memoryAcess(core(i).instruction)) {
02
03   if (memoryLocked && (core(i).id != current)) {
04     mpsoc.contentions++;
05     queueIn(core(i).id);
06
07     if (cg)
08       core(i).eSavings+=pmi.coreVersion(instruction,power);
09     else
10       core(i).energy+=pmi.coreVersion(instruction,cPPower);
11
12     core(i).idle++;
13     mpsoc.idle++;
14   } else {
15
16     if (onQueue(core(i).id)
17       queueOut(core(i).id)
18
19     if (dcf && (core(i).iAcesses > 1))
20       core(i).eSavings+=pmi.coreVersion(instruction,power);
21     else
22       core(i).energy+=pmi.coreVersion(instruction,Power);
23
24     memoryLock();
25 }
```

Figura 5.3: Tratamento das instruções de acesso à memória

Por fim, o simulador realiza todos os cálculos de acordo com as informações coletadas pelo módulo MMR para obter os ganhos energéticos relativos à aplicação do *clock gating* e do DCF. Mais, as seguintes informações são atualizadas para cada um dos processadores: ganho energético, número de acessos nas memórias privadas e na memória compartilhada, instruções executadas, número de ciclos ociosos, número de ciclos para terminar a execução da instrução corrente, posição na fila de espera (se o processador estiver congelado), estado de cada estágio do pipeline (se o processador tiver pipeline). Todos os valores utilizados nos cálculos são baseados nas informações contidas no arquivo PMI.

Além disso, o simulador oferece duas opções de execução. Na primeira delas, o utilizador deve especificar o manualmente o número de ciclos que devem ser simulados. Na segunda opção pode-se realizar uma simulação ciclo a ciclo, acompanhando o progresso da execução a cada passo.

5.1.3 A Interface do Simulador

A Figura 5.4 mostra a interface do simulador Hashi. O painel *Trace File Details (Instructions)* exibe informações detalhadas sobre o arquivo *trace* de execução informado como entrada ao simulador. Para cada instrução encontrada são mostrados o número de ocorrências e a porcentagem relativa. No painel *Instruction Mapping Details* é exibido o resultado do processo de mapeamento das instruções para as diferentes classes de instrução. Para cada classe de instrução é mostrado o número de ocorrências, bem como sua porcentagem relativa. O painel *System Panel* exibe informações detalhadas sobre cada um dos processadores do MPSoC. Nesse painel se podem observar as seguintes informações para cada um dos processadores: identificador, instrução em execução, número de acessos à memória, se está atualmente na fila de espera aguardando para efetuar acessos à memória, instruções executadas, energia gasta, energia salva, número de ciclos ociosos na instrução corrente, número total de ciclos ociosos, número total de ciclos da instrução corrente, número restante de ciclos da instrução corrente, versão, e a taxa relativa de freqüência. Finalmente, o painel *Execution Details* exibe as informações gerais a respeito da execução do MPSoC. Podem-se acompanhar os números de ciclos executados, de contenções no barramento, de instruções executadas, e de acessos à memória compartilhada. Além disso, são mostrados nesse painel o consumo energético total do MPSoC, bem como o ganho energético.

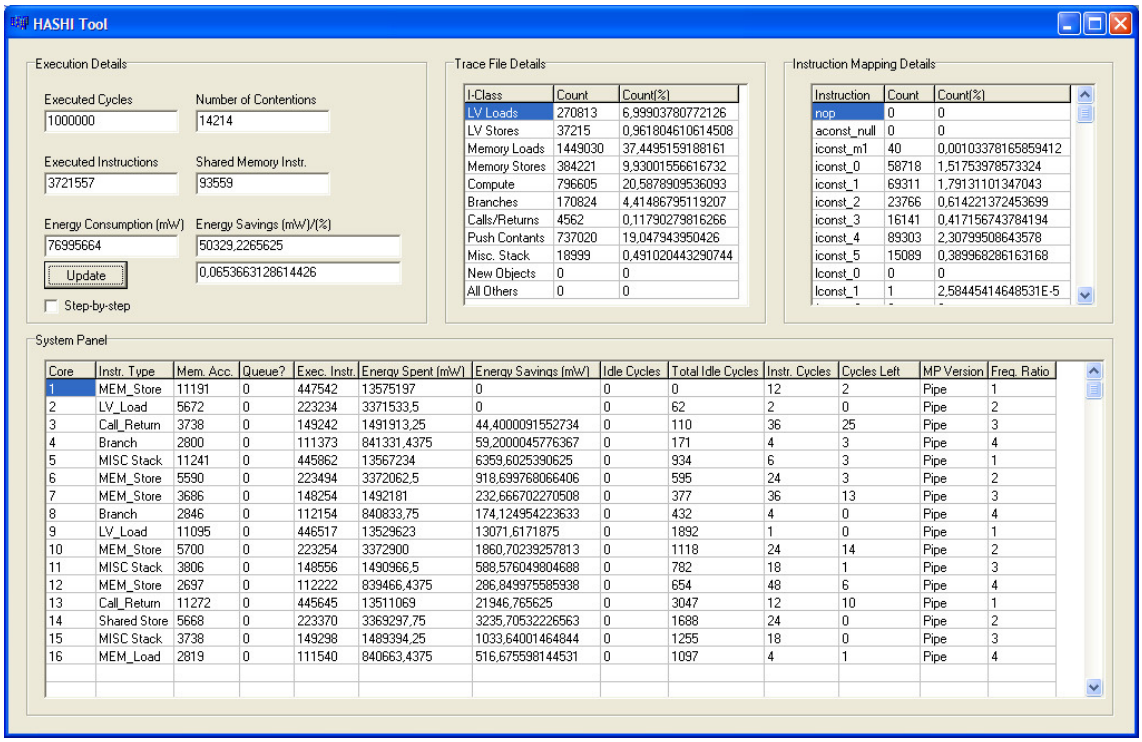


Figura 5.4: Interface do simulador Hashi

Além disso, é possível, através da interface do simulador, observar o estado atual da fila de espera de acesso à memória compartilhada, bem como o estado atual do pipeline de cada um dos processadores simulados.

6 RESULTADOS

Enquanto que os capítulos anteriores detalharam a arquitetura proposta e o simulador de MPSoCs, utilizados no escopo dessa dissertação, esse capítulo apresentará experimentos utilizando diferentes organizações de memória de dados baseados em diferentes aplicações de *benchmark*. A Seção 6.1 apresenta o ambiente de simulação utilizado nos experimentos, discutindo as diferentes ferramentas utilizadas no processo. Na Seção 6.2 é mostrado o custo em área da arquitetura proposta, usando as organizações de memória Shared-D e Mixed-D, em relação com as diferentes configurações de MPSoC usadas nos experimentos. Na Seção 6.3 são apresentados resultados de contenções no barramento, desempenho (em número de instruções executadas), e redução energética em um MPSoC com a organização Shared-D executando uma aplicação sintética. A Seção 6.4 realiza os mesmos experimentos, mas leva em consideração a organização Mixed-D, e ainda apresenta os resultados relacionados ao DCF. Finalmente, a Seção 6.5 apresenta os resultados de redução energética usando aplicações do pacote de *benchmark* SPECjvm98. Adicionalmente, deve-se destacar que experimentos com diferentes organizações de memória de instruções estão fora do escopo dessa dissertação e ficam como trabalhos futuros.

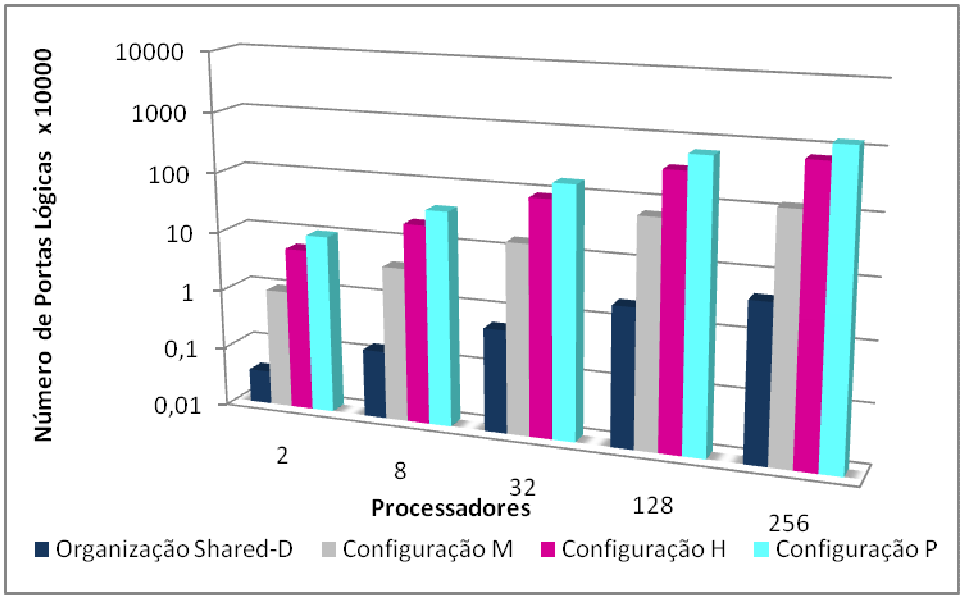
6.1 O Ambiente de Simulação

Os resultados dos experimentos realizados foram estimados usando o simulador Hashi para obter os valores de contenção no barramento, número de instruções executadas, e eficiência energética. Todos os resultados foram gerados a partir de simulações com 10 milhões de ciclos. A ferramenta Mentor Leonardo Spectrum (MENTOR, 2008) foi utilizada para estimação da área ocupada no chip, que foi computada em número de portas lógicas, após a síntese das descrições VHDL da arquitetura proposta, bem como das versões Multiciclo e Low-Power do processador FemtoJava, que serão referenciadas daqui para a frente simplesmente como multiciclo e pipeline. O consumo de potência dos diferentes tipos de instrução executando em ambos os processadores foi extraído através da ferramenta Synopsys Power Compiler (SYNOPSYS, 2008). Todos os resultados são baseados na tecnologia TSMC 0.18 (TSMC, 2008).

Além disso, foram consideradas três diferentes configurações de MPSoC nos experimentos realizados. A primeira configuração (P) assume que o MPSoC é composto unicamente de processadores com pipeline. A segunda configuração (M) assume que todos os processadores do MPSoC são multiciclo. Finalmente, a terceira organização (H) é um MPSoC heterogêneo com distribuição simétrica de processadores: 50% dos processadores são multiciclo e 50% possuem pipeline.

6.2 O Custo em Área

A Figura 6.1 mostra um comparativo da área total, contabilizada em número de portas lógicas, entre as diferentes configurações usando a organização de memória Shared-D e o custo em área do componente *shared-d block* (representado na figura pelo item Organização Shared-D), que representa toda a lógica necessária para implementar o gerenciamento da memória compartilhada do MPSoC, excluindo-se a área dos processadores. Além disso, deve-se ressaltar que o custo em área das memórias foi desconsiderado nos resultados. Como pode ser observado, cada aumento no número de processadores causa um aumento linear na área ocupada por cada configuração, que é proporcional ao aumento de área que ocorre na organização Shared-D. Isso ocorre porque cada aumento no número de processadores do MPSoC gera um aumento constante na lógica do componente *shared-d block*. Mais, o custo em área da organização Shared-D corresponde a apenas 0.35%, 3.6%, e 0.65% da área total ocupada pelas configurações P, M, e H, respectivamente. Conforme já foi dito, essas



taxas se mantêm constantes independentemente do número de processadores utilizado.

Figura 6.1: Área das diferentes configurações com a organização Shared-D

A Figura 6.2 mostra um comparativo da área total, contabilizada em número de portas lógicas, entre as diferentes configurações usando a organização de memória Mixed-D e o custo em área dos componentes *shared-d block* e *memory-d router* (representados na figura pelo item Organização Mixed-D), que representa toda a lógica necessária para implementar o gerenciamento das memórias compartilhada e privada do MPSoC, excluindo-se a área dos processadores. Da mesma forma que na figura anterior, o custo em área das memórias não foi considerado nos resultados. Nota-se, observando a figura, que nesse caso também ocorre um aumento proporcional na área ocupada por todos os itens comparados. Ou seja, o taxa de incremento da lógica dos componentes *shared-d block* e *memory-d router* ocorre na mesma proporção que a taxa de incremento da lógica das configurações P, M e H quando mais processadores são inseridos ao MPSoC. O custo em área da organização Mixed-D é de 1%, 9.5%, e 1.8% em relação às

configurações P, M, e H, respectivamente. Essas taxas se mantêm constantes independentemente do número de processadores usado.

A Figura 6.3a mostra a ocupação relativa em área de cada uma das configurações, usando a organização Mixed-D, sendo que os blocos P, M e H representam as configurações P, M e H, respectivamente. A Figura 6.3b mostra a relação de área entre a configuração M, que é a configuração de menor área, e a organização Mixed-D, representada pelo bloco MD, que corresponde à área dos componentes *shared-d block* e *memory-d router*. Finalmente, a Figura 6.3c apresenta o comparativo de área entre a organização Mixed-D e a organização Shared-D, representada pelo bloco SD, que corresponde à área do componente *shared-d block*.

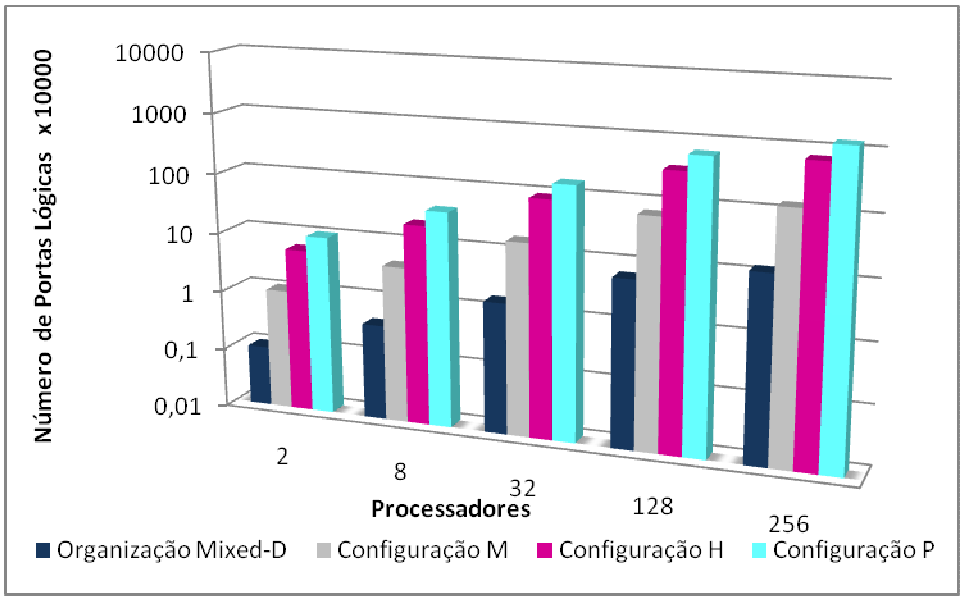
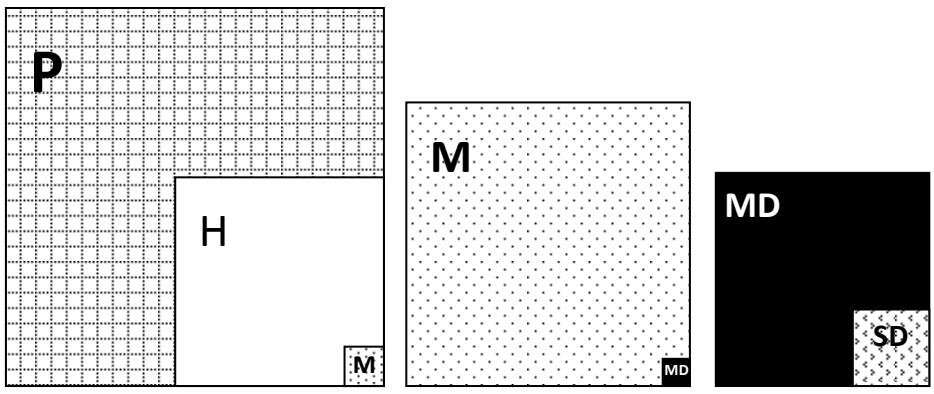


Figura 6.2: Área das diferentes configurações com a organização Mixed-D



(a) (b) (c)

Figura 6.3: Área relativa entre as diferentes configurações e a arquitetura proposta

6.3 Ganhos com uma Aplicação Sintética e a Organização Shared-D

Para os experimentos dessa seção considerou-se o simulador Hashi previamente configurado conforme a Tabela 6.1. Ou seja, ao invés de alimentar o simulador com um *trace* de execução a fim de extrair o comportamento da aplicação, utilizaram-se os

dados da primeira e da segunda coluna, que foram obtidos de (O'CONNOR et al., 1997) e especificam a freqüência dinâmica média de diferentes classes de instruções em dois típicos programas Java: Javac e Raytracer, do *benchmark* SPECjvm98 (SPECjvm98, 2008). As três colunas seguintes indicam o número de ciclos, o consumo de potência e o número de acessos à memória de cada classe de instrução executando no processador

Tabela 6.1: Freqüência Dinâmica de Diferentes Classes de Instruções Versus Estatísticas de Execução nas Diferentes Versões do Processador Java

		FemtoJava Pipeline			**FemtoJava Multi-cycle**		
Instruction Class	Freq. (%)	Cycles	Power (mW)	Memory	Cycles	Power (mW)	Memory
LV loads	34,5	1	3,2	0	4	0,45	5
LV stores	7,0	1	3,7	0	6	0,38	2
Memory loads	20,2	1	3,5	1	7	0,27	2
Memory stores	4,0	12	3,3	1	13	0,27	2
Compute	9,2	1	2,7	0	3	0,64	1
Branches	7,9	1	3,2	0	5	0,6	2
Calls/returns	7,3	12	4,0	0	14	0,4	6
Push constant	6,8	1	2,6	0	4	0,4	1
Stack operations	2,1	6	3,0	0	6	0,5	1
New objects	0,4	150	3,0	0	150	0,4	6
All Others	0,6	1	3,0	0	3	0,4	1

com pipeline. Finalmente, as três últimas colunas mostram as mesmas informações relativas ao processador multiciclo.

É importante ressaltar que todas as medidas de eficiência energética dos próximos experimentos são referentes aos ciclos ociosos dos processadores, quando o *clock gating* é disparado. A eficiência energética mede a porcentagem de energia que foi salva com o uso do *clock gating* comparado a uma abordagem sem *clock gating*. Além disso, deve-se notar que o processador multiciclo tem a pilha de operandos implementada na memória principal. Desse modo, nesse processador, cada referência à pilha de operandos é computada como um acesso à memória. Já no processador com pipeline, a pilha de operandos é implementada dentro do processador, no banco de registradores.

Nas próximas três figuras, os resultados obtidos consideram a organização de memória Shared-D, para três diferentes configurações de MPSoC (P, M, e H), explicadas na seção anterior.

A Figura 6.4 mostra o número de contenções no barramento em função do número de processadores, baseadas no número de colisões evitadas durante os acessos à memória compartilhada. Na configuração P, a contenção é substancialmente menor que a das outras duas configurações. Isso acontece porque o número de acessos à memória do processador multiciclo é consideravelmente maior que o do processador com pipeline. Isso pode ser facilmente notado na Tabela 6.1, ao se observar o número de acessos à memória de cada classe de instrução em cada um dos processadores. Como a pilha de operandos do processador multiciclo está na memória, cada acesso à pilha representa um acesso também à memória. Entretanto, quando o número de processadores aumenta, os resultados de contenção no barramento começam a ficar

parecidos para todas as configurações. Isso acontece com 16 processadores, quando o barramento da configuração M satura,. A partir desse ponto, o número de contenções da configuração M é rapidamente alcançado pelas demais configurações.

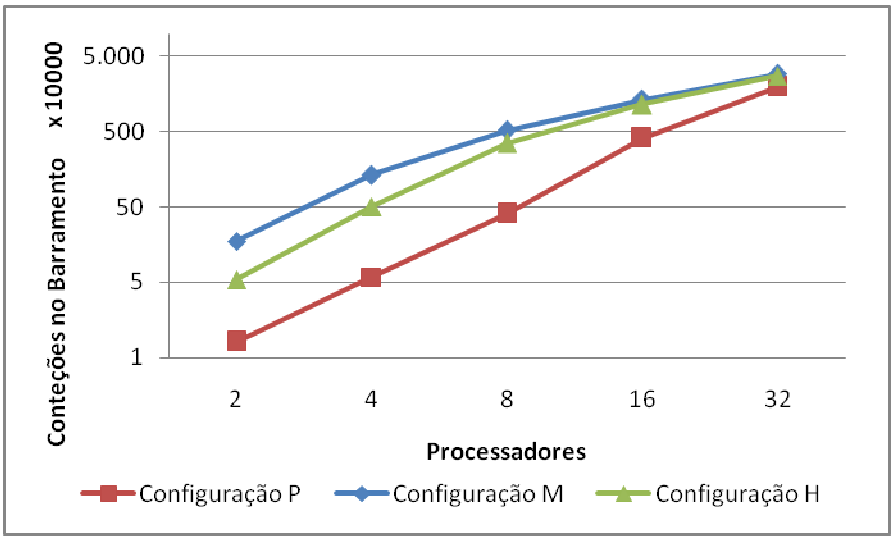


Figura 6.4: Contenções no barramento em função do número de processadores

A Figura 6.5 mostra o número total de instruções executadas por todos os processadores. A configuração P apresenta um número elevado de instruções executadas em relação às demais configurações. Isso ocorre porque quase todas as instruções são executadas em apenas um ciclo no processador com pipeline. As configurações M e H são penalizadas porque o processador multiciclo necessita de vários ciclos para executar cada instrução. Todavia, quando ocorre a saturação do barramento (com 4, 8, e 16 processadores, para as configurações M, H, e P, respectivamente), a taxa de aumento do número de instruções executadas de cada configuração permanece constante.

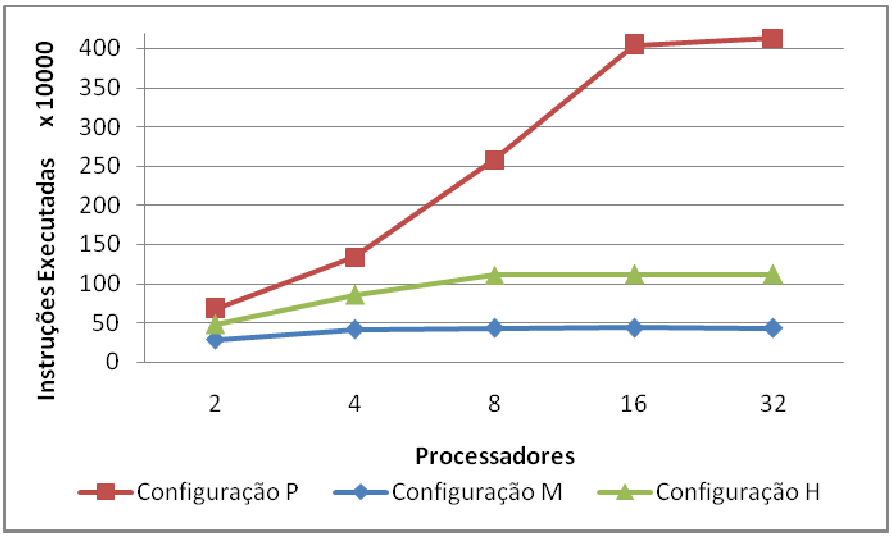


Figura 6.5: Número total de instruções executadas em função do número de processadores

Na Figura 6.6 é apresentado o ganho energético em função do número de processadores. Naturalmente, esse ganho é proporcional às contenções no barramento, uma vez que, quando os processadores estão aguardando na fila para realizar o acesso à memória, eles estão também congelados através do *clock gating* para economizar potência. Na configuração P, o ganho energético é menor que o das demais configurações, pois o número de acessos à memória do processador com pipeline em cada instrução é menor que o do processador multiciclo. Considerando-se um MPSoC com dois processadores, enquanto que na configuração P o ganho energético é de 0.4%, nas configurações M e H ele é de 9% e 5%, respectivamente. Mais uma vez, isso ocorre porque o número de acessos à memória do processador multiciclo é extremamente elevado mesmo com um número substancialmente baixo de processadores. Apesar disso, a configuração P começa a exibir um ganho energético considerável (de 5%) a partir de 8 processadores. Todavia, nesse mesmo ponto as configurações M e H apresentam um ganho de 65% e 44%, respectivamente. O aumento elevado no ganho energético da configuração P, que se justifica devido ao elevado número de contenções no barramento, faz com que o ganho energético da configuração P se aproxime rapidamente dos resultados das configurações M e H, que, por sua vez, começam a reduzir a taxa de ganho a partir de certo ponto, devido à saturação do barramento. Deve-se notar, que os elevados ganhos energéticos apresentados nesse experimento ocorrem devido à existência de uma única memória de dados, que é compartilhada. Dessa forma, quando há um número elevado de processadores o ganho energético se deve à saturação do barramento que, conseqüentemente, faz com que muitos processadores estejam congelados pelo *clock gating* aguardando para efetuar o acesso.

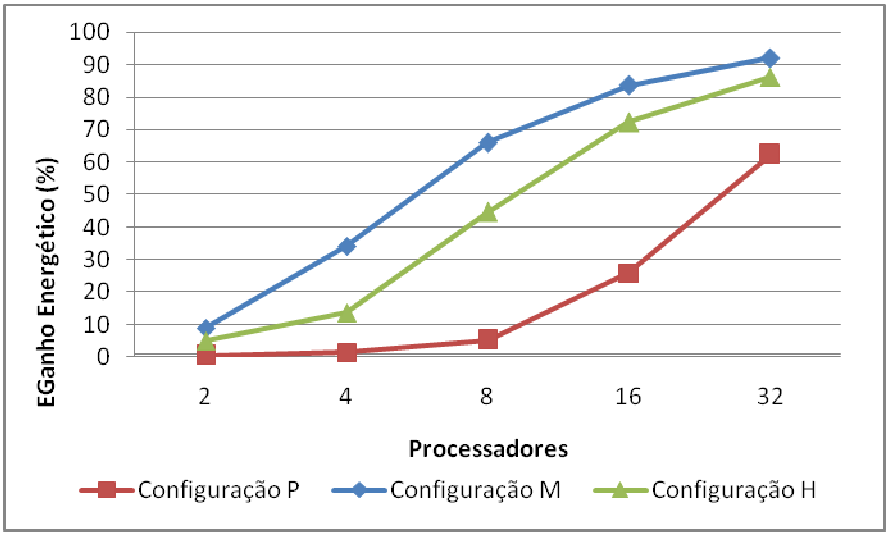


Figura 6.6: Ganho energético em função do número de processadores

6.4 Ganhos com uma Aplicação Sintética e a Organização Mixed-D

Da mesma forma que na seção anterior, nos próximos três experimentos considerou-se o simulador Hashi previamente configurado conforme a Tabela 6.1, mas usando dessa vez a organização de memória Mixed-D. Além disso, considerou-se uma aplicação *multithread* onde o número de acessos à memória compartilhada varia de 10% até 50% do número total de acessos à memória de cada processador. Entretanto, para facilitar a visualização dos resultados, e sabendo-se que todos os resultados da faixa de valores citada anteriormente obedecem ao mesmo comportamento, seus valores

intermediários não são mostrados nas próximas figuras. A faixa de valores, que representa a quantidade de acesso à memória compartilhada, foi extraída de (GUZ et al., 2007), e é baseada em experimentos com diferentes classes de aplicações *multithread* dos *benchmarks* SPEComp (ASLOT et al., 2001) e Splash 2 (WOO et al., 1995).

A Figura 6.7 mostra o número de contenções no barramento em função do número de processadores. Considerando que 10% dos acessos à memória são relativos à memória compartilhada, o comportamento das três configurações é semelhante, e elas apresentam um número quase insignificante de contenções até 32 processadores. Essa baixa discrepância nos resultados de contenção das diferentes organizações ocorre porque a pilha de operandos do processador multiciclo está agora na memória privada, o que causa uma drástica redução no número de acessos à memória compartilhada desse processador. Por outro lado, a partir de 64 processadores, a contenção no barramento começa a apresentar valores elevados, pois apesar da baixa porcentagem individual de acessos à memória compartilhada, o número de processadores é suficientemente grande para causar freqüentemente acessos simultâneos a essa memória. Quando os acessos à memória compartilhada são aumentados para 50% do total de acessos à memória, a contenção no barramento mostra valores significativos a partir de 8 processadores. Com 256 processadores, entretanto, o número de contenções é bastante semelhante para todas as configurações, independente da porcentagem de acessos à memória compartilhada, devido à saturação do barramento.

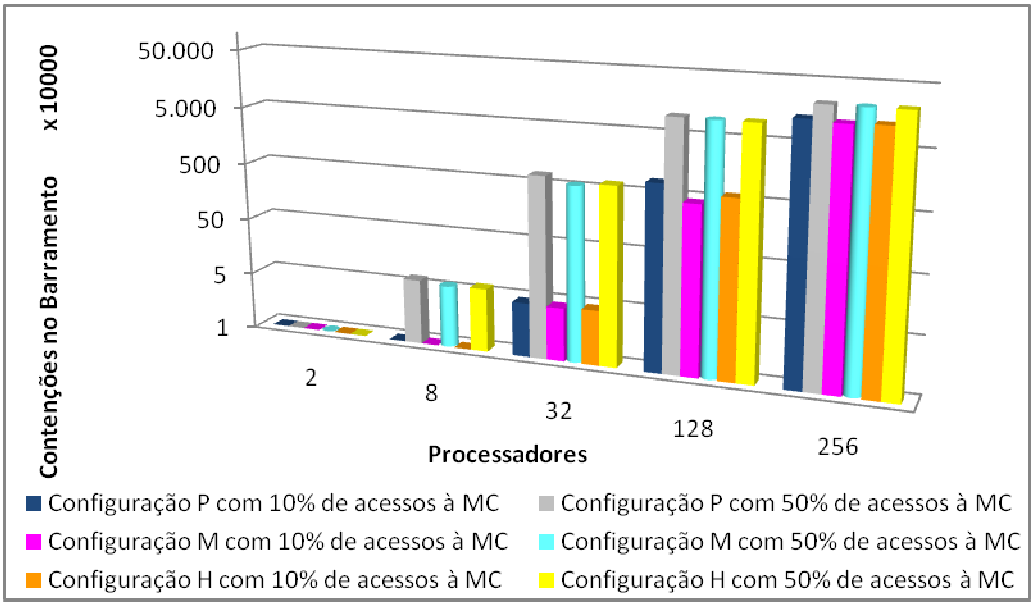


Figura 6.7: Contenções no barramento em função do número de processadores

A Figura 6.8 mostra o número total de instruções executadas em todos os processadores em função do número de processadores. Nas configurações com 10% dos acessos à memória relativos à memória compartilhada, a saturação do barramento aparece quando o número de processadores é superior a 128. É possível observar que a partir desse ponto, o incremento na taxa de instruções executadas permanece constante. Já nas configurações com 50% dos acessos à memória relativos à memória compartilhada, a saturação aparece antes (a partir de 32 processadores), pois o número de contenções é superior. Mais uma vez, o número de instruções executadas pela configuração P é maior que nas demais. Pois, enquanto que o processador com pipeline

executa a maioria das instruções em apenas um ciclo, o processador multiciclo gasta vários ciclos na execução de cada instrução.

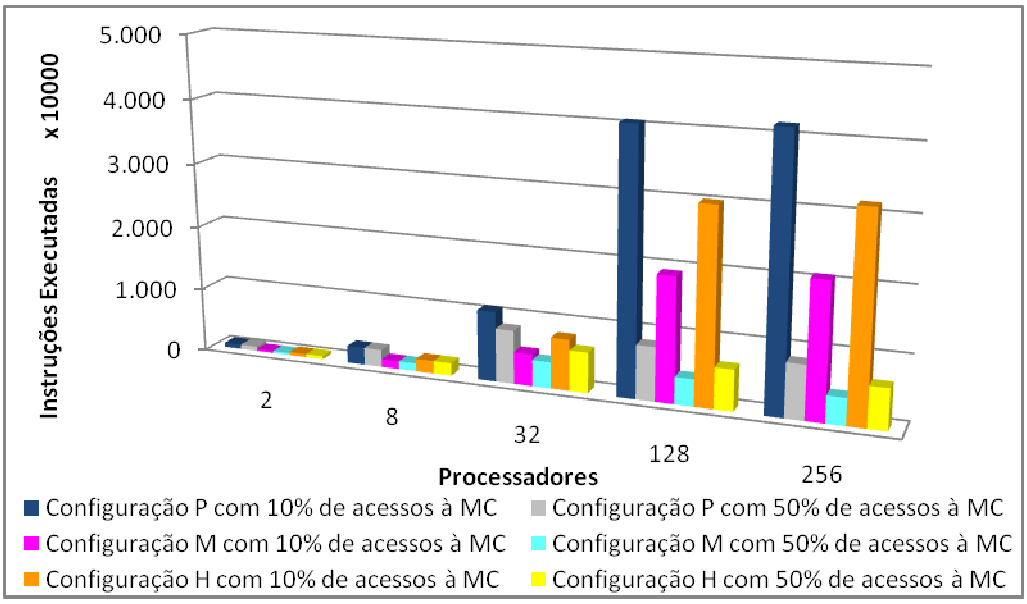


Figura 6.8: Número total de instruções executadas em função do número de processadores

A eficiência energética em função do número de processadores é mostrada na Figura 6.9. Como esperado, a eficiência energética das configurações com 10% de acessos à memória compartilhada somente se torna atrativa quando o número de processadores é alto, a partir de 128, quando a eficiência energética atinge 8.3%, 2.4%, e 5.4% nas configurações P, M, e H, respectivamente. Isso acontece porque há poucas oportunidades para usar *clock gating* e reduzir o consumo de potência. Já com 256 processadores, a eficiência energética sofre um aumento expressivo (52.26%, 38.18%, e 49.73% nas configurações P, M, e H, respectivamente), pois a baixa porcentagem de acessos à memória compartilhada é compensada pelo alto número de processadores. Quando os acessos à memória compartilhada são aumentados para 50% do total de acessos à memória, a eficiência energética se acentua a partir de 32 processadores, quando os ganhos atingem 24%, 13%, e 20.5% nas configurações P, M, e H, respectivamente. Nota-se, além disso, que conforme o número de processadores aumenta, a eficiência energética vai ficando cada vez mais próxima nas configurações com a mesma porcentagem de acessos à memória compartilhada. Isso acontece, novamente, devido à saturação do barramento.

Na Figura 6.10 é comparada a eficiência energética da configuração P, com 10% e 50% dos acessos à memória relativos à memória compartilhada (casos A e D, respectivamente), com a mesma configuração usando a técnica DCF. Aplicou-se o DCF nos acessos à memória compartilhada, com taxas de acesso de 10% (caso B) e 50% (caso E). Aplicou-se também o DCF nos acessos às memórias privada e compartilhada, com taxas de acesso de 90% e 10% (caso C), e 50% e 50%, respectivamente (caso F). Além disso, para poder utilizar o DCF, aumentou-se a freqüência dos processadores para alcançar uma relação de 2:1 entre os processadores e as memórias.

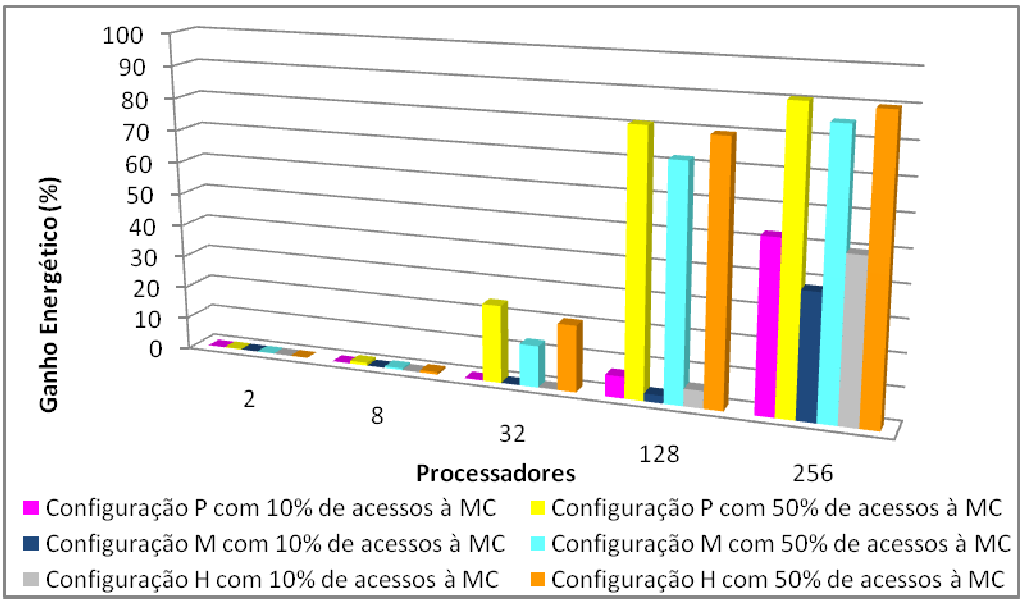


Figura 6.9: Ganho energético em função do número de processadores

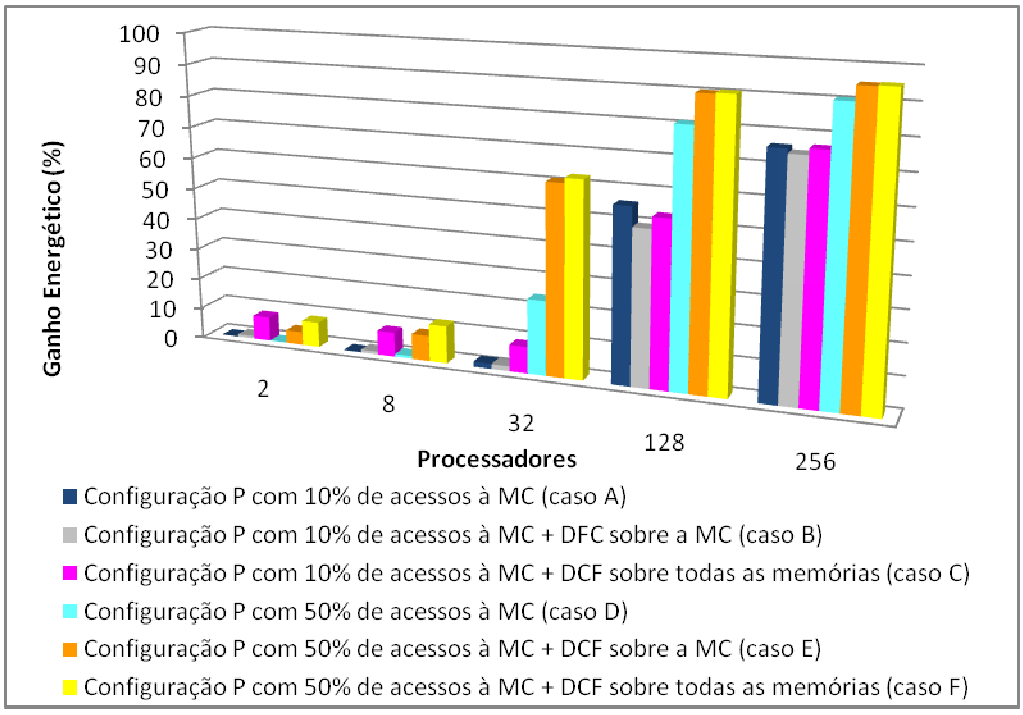


Figura 6.10: Ganho energético da configuração P usando diferentes taxas de acessos à memória compartilhada com e sem DCF

Usando a técnica DCF, quase todas as diferentes configurações apresentam um salto inicial na eficiência energética: 3.84%, 7.8%, e 7.7% para os casos E, C, e F, respectivamente. Isso acontece porque o *clock gating* não é somente aplicado para evitar as colisões do barramento, mas também para reduzir o consumo de potência dos processadores nos ciclos ociosos durante os acessos à memória. O salto inicial só não aparece no caso B, que tem uma taxa de eficiência energética de apenas 0.8%. Isso ocorre devido à baixa porcentagem de acessos à memória compartilhada, bem como ao

fato de o DCF não estar sendo aplicado aos acessos da memória privada. Além disso, como se pode notar, a eficiência do DCF vai aumentando conforme são adicionados mais processadores ao MPSoC, até o ponto onde o barramento satura. A eficiência energética média com o uso do DCF para os casos B, C, E, e D, é de 13%, 17%, 19%, e 18.3%, respectivamente, quando comparada à configuração correspondente sem DCF.

6.5 Ganhos com Aplicações do Conjunto de Benchmarks SPECjvm98

Nos próximos experimentos, foram utilizados processadores executando em diferentes freqüências e usando a organização Mixed-D. Para cada uma das configurações considerou-se os processadores executando a 50 MHz, 100 MHz, 150 MHz, e 200 MHz. Em todas as configurações foi considerado que 25% dos processadores executam em cada uma das diferentes freqüências. Além disso, os processadores foram intercalados seqüencialmente, de acordo com a sua freqüência, nas portas do componente *shared-d block*. Considerando-se, por exemplo, a configuração P: a primeira porta do componente *shared-d block* foi alocada para um processador com 200 MHz, a segunda porta foi alocada para um processador com 150 MHz, a terceira porta foi alocada para um processador com 100 MHz, a quarta porta foi alocada para um processador com 50 MHz, e assim sucessivamente. Mais, o simulador Hashi foi carregado com diferentes aplicações de *benchmark* (três delas do conjunto de *benchmarks* SPECjvm98), destacadas a seguir:

- Javac é um compilador Java, da Sun Microsystems JDK 1.0.2. Ele é um programa orientado a objetos, com aproximadamente 25.000 linhas de código em 170 classes diferentes. O tamanho aproximado de seu arquivo de *bytecodes* é de 422 Kbytes.
- JESS (*Java Expert Shell System*) é baseado no sistema especialista NASA CLIPS. Em termos simples, um sistema desse tipo aplica continuamente um conjunto de sentenças *if–then*, chamadas de regras, para um determinado conjunto de dados, chamado de lista de fatos. O seu *workload* representa um conjunto de quebra-cabeças comumente usado com o NASA CLIPS. Para aumentar o tempo de execução o *benchmark* iterativamente confirma um novo conjunto de fatos representando o mesmo quebra-cabeça, mas com literais diferentes. Os fatos antigos não são retratados. Dessa maneira, o motor de inferência precisa pesquisar através de conjuntos de regras progressivamente maiores conforme a execução progride.
- Compress é uma aplicação que realiza a descompressão de arquivos de áudio em conformidade com a especificação de áudio ISO MPEG Layer–3. O seu *workload* consiste de aproximadamente 4MB de dados de áudio.
- MP3 é um decodificador de arquivos de áudio, comumente usado em aplicações embarcadas. Ele foi construído sem uso de ponto flutuante e alocação dinâmica de objetos.

A Tabela 6.2 detalha a freqüência dinâmica de cada classe de instrução, gerada pelo simulador Hashi, para cada uma das aplicações de *benchmark* utilizadas. Pode-se notar que, enquanto as aplicações MP3 e Compress apresentam um grande volume de acessos à memória, devido ao seu comportamento altamente *data flow*, as demais aplicações possuem um considerável volume de desvios, destacando seu comportamento *control flow*.

Tabela 6.2: Resultado da análise do simulador HASHI nas aplicações de benchmark

	Dynamic Frequency on Java Programs (%)			
Instruction Class	**MP3**	**Javac**	**Compress**	**JESS**
LV Loads	7,0	33,8	32,4	35,3
LV Stores	1,0	3,4	8,8	4,3
Memory Loads	37,5	15,8	19,4	17,2
Memory Stores	10,0	3,7	4,8	2,8
Compute	20,5	8,7	11,2	4,0
Branches	4,4	9,3	6,4	10,7
Calls/Returns	0,1	10,5	3,6	14,7
Push Constant	19,0	9,9	7,2	5,8
Misc. Stack Operations	0,5	3,3	5,8	2,3
New Objects	0,0	0,3	0,0	0,5
All Others	0,0	1,3	0,0	2,4

A Figura 6.11 mostra a economia energética em função do número de processadores usando a aplicação MP3 como entrada do simulador Hashi, e considerando que 10% dos acessos à memória são relativos ao espaço compartilhado. As configurações com o DCF desativado apresentam ganhos significativos somente a partir de 64 processadores, quando o número de contenções começa a aumentar. As configurações com DCF ativo apresentam um grande salto inicial na economia energética: 20.26%, 5,29%, e 10.32% nas configurações P, M, e H, respectivamente. Esse incremento inicial alcançado por meio do DCF permanece constante até 64 processadores, quando a economia energética do clock gating sobre os processadores na fila de espera começa a aumentar. Além disso, pode-se observar que a configuração P obteve maior redução de energia que as demais configurações. Pois, enquanto o processador multiciclo gasta vários ciclos em cada classe de instrução, o processador com pipeline gasta apenas um ciclo na maioria delas.

Já a Figura 6.12 mostra a redução de energia em função do número de processadores usando a aplicação MP3 como entrada para o simulador Hashi, e considerando que 50% dos acessos à memória são relativos à memória compartilhada. Dessa vez, as configurações usando com o DCF desativado apresentaram uma redução de energia significativa a partir de 8 processadores: 4.45%, 1.7%, e 3,4% nas configurações P, M, e H, respectivamente. Isso ocorre porque o aumento de acessos à memória compartilhada causa também um aumento no número de contenções no barramento. A partir de 64 processadores, a taxa de aumento na redução de energia começa a diminuir devido à saturação do barramento. As configurações com DCF mostram um incremento inicial substancialmente maior que nas configurações com DCF da Figura 10, de 24.93%, 7.25%, e 16.46%, para as configurações P, M, e H, respectivamente. Entretanto, esse aumento se justifica pela aplicação do *clock gating* para evitar as colisões do barramento e não propriamente ao DCF.

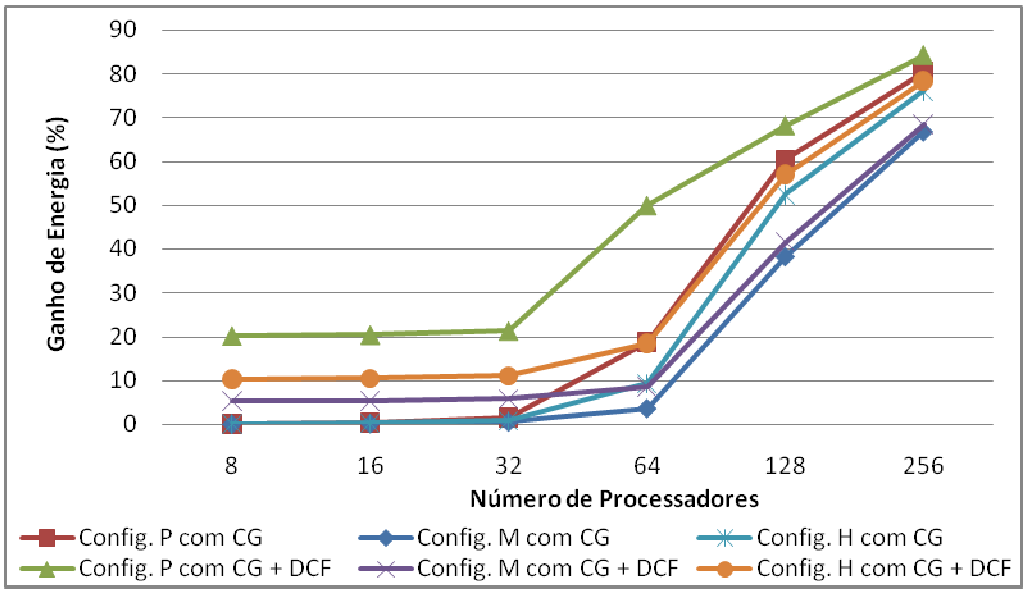


Figura 6.11: Ganho de energia em função do número de processadores usando a aplicação MP3 com 10% dos acessos à memória relativos à memória privada

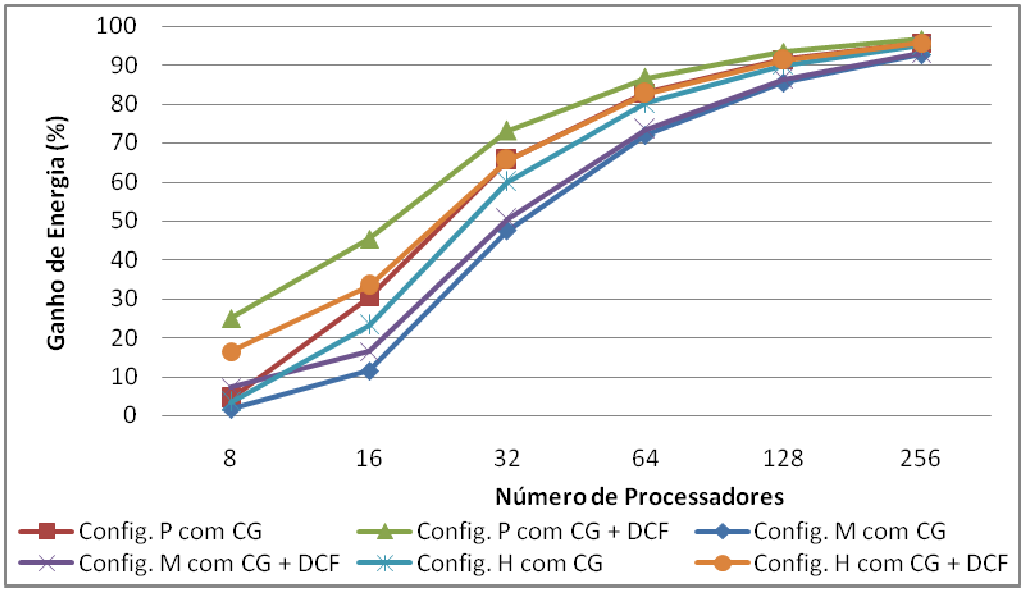


Figura 6.12: Ganho de energia em função do número de processadores usando a aplicação MP3 com 50% dos acessos à memória relativos à memória privada

A Figura 6.13 mostra o ganho de energia em função do número de processadores usando a aplicação Compress como entrada do simulador Hashi, e considerando que 10% dos acessos à memória são relativos ao espaço compartilhado. Apesar de essa aplicação ser altamente *data flow*, bem como a aplicação MP3, as configurações sem DCF só passam a apresentar ganhos substanciais de energia a partir de 128 processadores: 19.78%, 4.20%, e 11.04%, para as configurações P, M, e H, respectivamente. Nota-se que, com 128 processadores, a relação de ganho de energia entre a configuração P e as demais é relativamente grande. Isso ocorre porque além de os processadores do MPSoC estarem executando em freqüências diferentes, a maioria

das instruções leva vários ciclos para ser executada no processador multiciclo. Isso, somado ao baixo percentual de acesso à memória compartilhada utilizado nesse experimento, faz com que haja poucas oportunidades para aplicação do *clock gating* nas configurações M e H. Entretanto, com 512 processadores a relação de ganho entre as diferentes configurações é bastante reduzida, sendo que o ganho de energia é de 60.25%, 40.52%, e 53.52% nas configurações P, M, e H, respectivamente. Isso se deve ao grande número de processadores, que acaba se sobrepondo à baixa freqüência de acessos à memória compartilhada do processador multiciclo. As configurações com DCF ativo forçam um salto inicial no ganho energético, que se mantém até 64 processadores, quando a freqüência de acessos à memória compartilhada cresce substancialmente, permitindo melhor aproveitamento do *clock gating* nos acessos simultâneos.

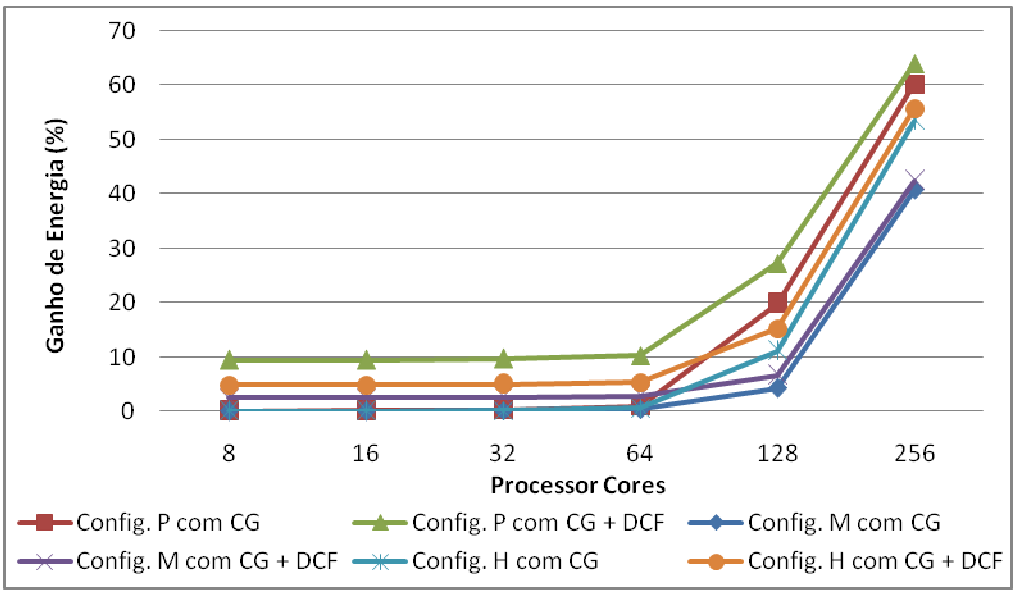


Figura 6.13: Ganho de energia em função do número de processadores usando a aplicação Compress com 10% dos acessos à memória relativos à memória privada

Já a Figura 6.14 mostra a redução de energia em função do número de processadores usando a aplicação Compress como entrada para o simulador Hashi, e considerando que 50% dos acessos à memória são relativos à memória compartilhada. Nas configurações sem DCF, o ganho de energia começa a se mostrar interessante a partir de 16 processadores: 4.22%, 1.7%, e 2.83% nas configurações P, M, e H, respectivamente. Entretanto, na faixa entre 32 e 64 processadores é que se pode observar a maior taxa de aumento no ganho de energia em todas as configurações, pois o número de processadores somado à freqüência de acessos à memória compartilhada atinge o ponto ótimo, sem saturar o barramento. A partir de 64 processadores, entretanto, nota-se que a taxa de incremento no ganho de energia começa a reduzir, devido à saturação do barramento. Já nas configurações com DCF habilitado, é interessante notar que o incremento inicial no ganho de energia é praticamente o mesmo que se observa no experimento anterior. A explicação é simples: com apenas 8 processadores o *clock gating* ainda não oferece benefícios significativos no gerenciamento do acesso à memória compartilhada. Dessa forma, quase todo o ganho de energia é relativo ao uso do DCF. Entretanto, a partir de 16 processadores o *clock gating* passa a ser mais efetivo, pois a freqüência de acessos à memória compartilhada sofre um incremento substancial.

Além disso, percebe-se que a relação de ganho de energia entre as configurações com DCF e sem DCF permanece praticamente inalterada na faixa entre 16 e 128 processadores. Depois desse ponto, devido à saturação do barramento, os resultados tendem a se aproximar cada vez mais.

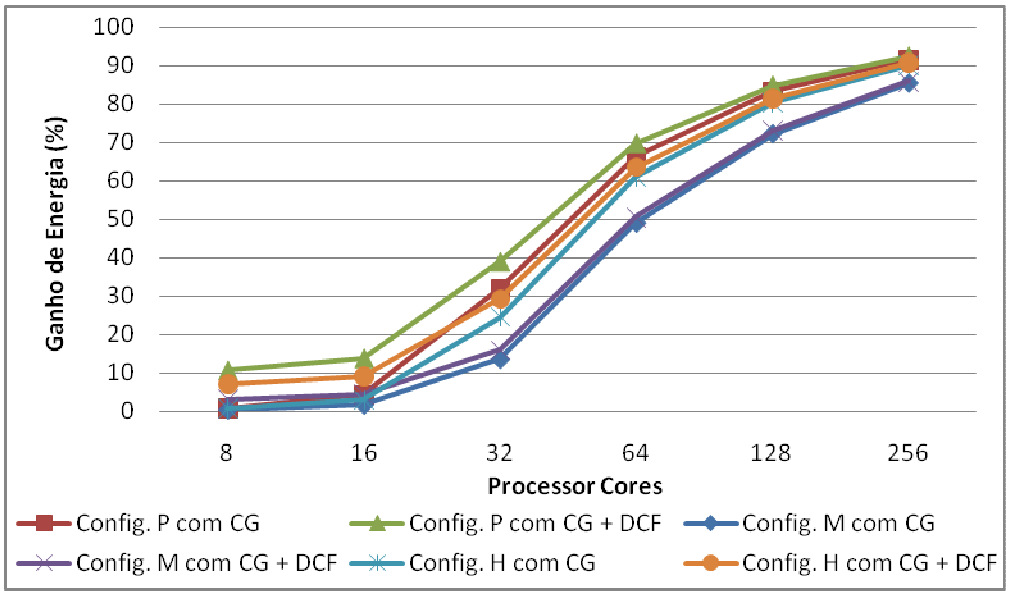


Figura 6.14: Ganho de energia em função do número de processadores usando a aplicação Compress com 50% dos acessos à memória relativos à memória privada

A Figura 6.15 mostra o ganho energético em função do número de processadores usando a aplicação JESS como entrada do simulador Hashi, e considerando que 10% dos acessos à memória são relativos ao espaço compartilhado. Diferentemente das outras duas aplicações, mostradas nos experimentos anteriores, que eram altamente *data flow*, a aplicação Jess é altamente *control flow*. Dessa forma, pode-se notar que para esse tipo de aplicação, o ganho energético acaba prejudicado. Sem o DCF, até 128 processadores o ganho energético é irrelevante. Esse ganho só se mostra acentuado quando são utilizados 256 processadores, quando se podem observar taxas de 14.53%, 15.73%, e 16.36% para as configurações P, M, e H, respectivamente. Além disso, é interessante notar que o ganho energético da configuração P é menor que o das demais configurações, ao contrário do que foi mostrado nos experimentos anteriores. Isso acontece porque o uso de uma aplicação com uma baixa quantidade de acessos à memória, somado ao uso de um baixo percentual de acessos ao espaço compartilhado, acaba mascarando o fato de o processador multiciclo levar mais tempo na execução de cada instrução. Nas configurações com DCF habilitado observa-se um significativo salto inicial no ganho energético: 4.38%, 1.6%, e 2.36% para as configurações P, M, e H, respectivamente. Observa-se que a configuração P com DCF ativo sofre um incremento maior que as demais configurações. Isso se justifica pelo fato de que o DCF é aplicado tanto às memórias privadas como à memória compartilhada, e como as instruções são executadas mais rapidamente no processador com pipeline, há um número maior de acessos às memórias nesse processador. Entretanto, na medida em que o número de processadores vai aumentando, a relação de ganho energético entre as configurações começa a reduzir, devido à saturação do barramento.

A Figura 6.16 mostra a redução de energia em função do número de processadores usando a aplicação JESS como entrada para o simulador Hashi, e considerando que 50% dos acessos à memória são relativos à memória compartilhada. Ganhos energéticos consideráveis para as configurações sem DCF somente são observados a partir de 32 processadores. Nesse ponto, o ganho para as configurações P, M, e H é de 2.9%, 2.72%, e 2.76%, respectivamente. A proporção de ganho energético das configurações permanece quase constante até 256 processadores, quando o ganho atinge 82.47%, 78.03%, e 81.67% para as configurações P, M, e H, respectivamente.

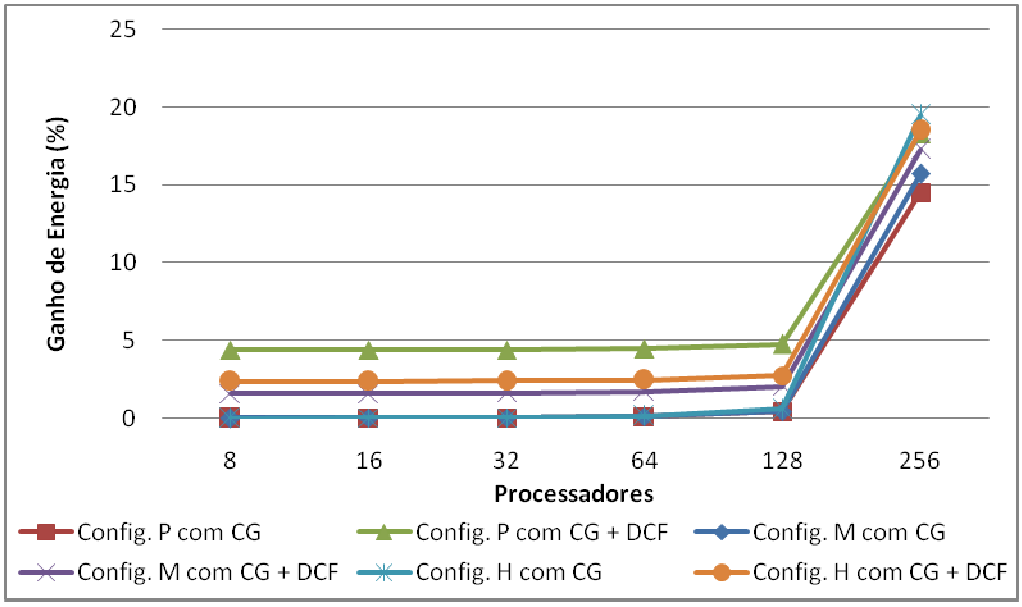


Figura 6.15: Ganho de energia em função do número de processadores usando a aplicação JESS com 10% dos acessos à memória relativos à memória privada

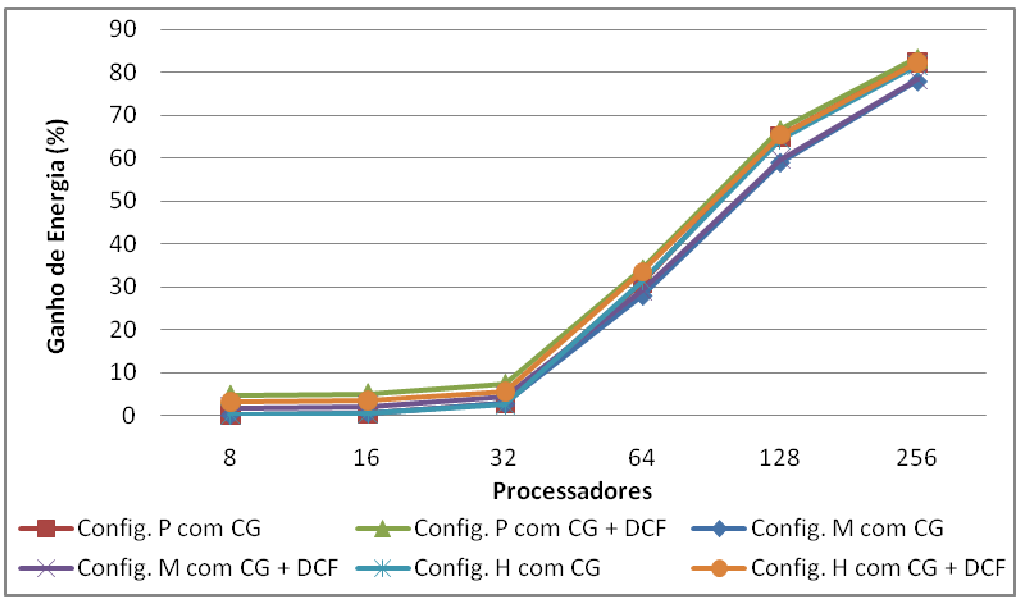


Figura 6.16: Ganho de energia em função do número de processadores usando a aplicação JESS com 50% dos acessos à memória relativos à memória privada

Por outro lado, as configurações com DCF habilitado apresentam um ganho energético inicial levemente maior que o observado no experimento anterior: 4.7%, 1.7%, e 3.2% para as configurações P, M e H, respectivamente. Além disso, nesse caso, já se pode observar um pequeno incremento no ganho energético quando o número de processadores é aumentado de 8 para 16. O pico na taxa de incremento no ganho energético acontece entre 32 e 128 processadores. A partir disso, há uma redução nessa taxa devido à saturação do barramento. É importante observar que o substancial aumento na taxa de acessos è memória compartilhada proporcionou um aumento dramático no ganho energético (a partir de 32 processadores), que no experimento anterior só passou a se intensificar a partir de 128 processadores.

A Figura 6.17 mostra o ganho energético em função do número de processadores usando a aplicação Javac como entrada para o simulador Hashi, e considerando que 10% dos acessos à memória são relativos ao espaço compartilhado. Essa aplicação, da mesma forma que anterior, apresenta um comportamento altamente *control flow*. Observa-se que as configurações sem DCF somente têm algum ganho energético a partir de 128 processadores, quando as taxas atingem 0.84%, 0.68%, e 0.73% para as configurações P, M, e H, respectivamente. Com 256 processadores o ganho energético sofre um enorme incremento, apresentando taxas de 28.4%, 21.67%, e 26.71% para as configurações P, M, e H, respectivamente. Por outro lado, as configurações com DCF mostram um incremento inicial no ganho energético com taxas de 5.17%, 1.7%, e 2.72% para as configurações P, M, e H, respectivamente. Esse ganho permanece constante até sofrer um leve aumento na faixa entre 64 e 128 processadores. A partir desse ponto, devido ao gerenciamento dos acessos à memória compartilhada com o *clock gating*, o ganho energético mostra um aumento intensificado até 256 processadores, quando as taxas de ganho são de 31.89%, 23,13%, e 28.97% para as configurações P, M, e H, respectivamente.

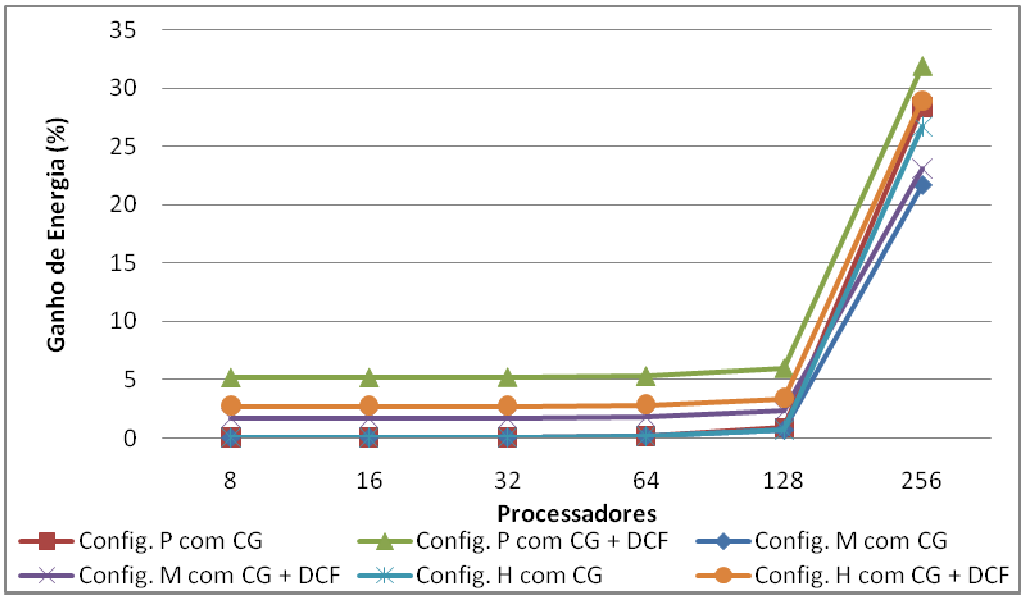


Figura 6.17: Ganho de energia em função do número de processadores usando a aplicação Javac com 10% dos acessos à memória relativos à memória privada

A Figura 6.18 mostra o ganho energético em função do número de processadores usando a aplicação Javac como entrada para o simulador Hashi, e considerando que

50% dos acessos à memória são relativos ao espaço compartilhado. Nas configurações sem DCF, observa-se um leve aumento no ganho energético a partir de 16 processadores. Entretanto, ganhos significativos aparecem a partir de 32 processadores, quando as taxas equivalem a 4.96%, 3.67%, e 4.18% para as configurações, P, M, e H, respectivamente. Desse ponto até 128 processadores, o ganho energético sofre um amento agressivo. A partir disso, a saturação do barramento faz com que a taxa de ganho energético seja bastante prejudicada. Já as configurações com o DCF ativo sofrem um incremento inicial no ganho energético que corresponde a 5.59%, 1.85%, e 3.7% para as configurações P, M, e H, respectivamente. A partir disso, nota-se um leve incremento no ganho energético até 32 processadores. De 32 até 128 processadores, esse incremento se intensifica da mesma forma que na configuração sem DCF. A partir disso, até 256 processadores, a taxa de ganho energético não cresce mais na mesma proporção. Isso novamente é causado pelas contenções no barramento.

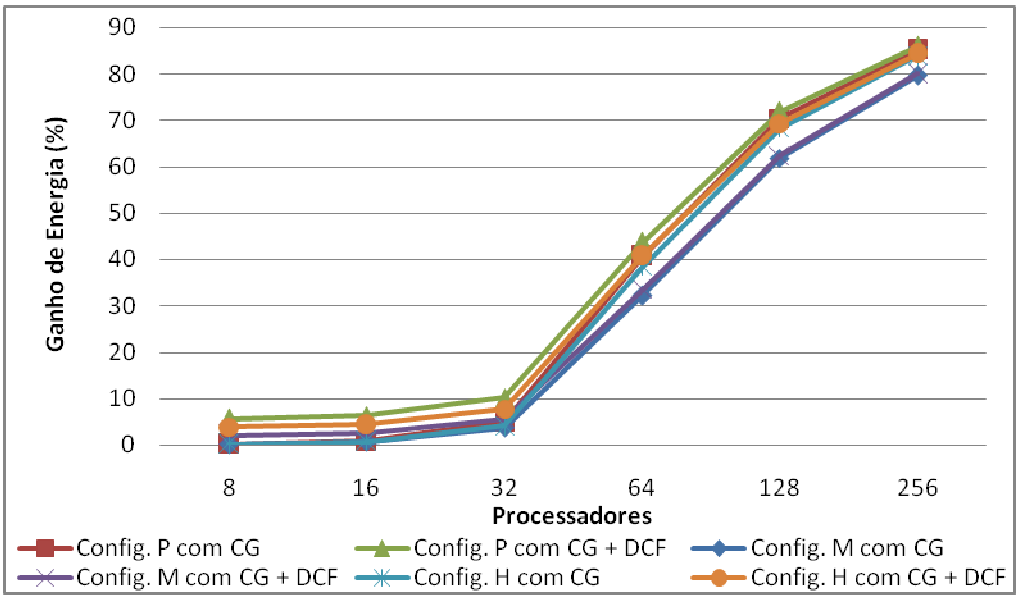


Figura 6.18: Ganho de energia em função do número de processadores usando a aplicação Javac com 50% dos acessos à memória relativos à memória privada

6.6 Análise dos Resultados

Os experimentos apresentados nessa seção demonstraram que a aplicação executada exerce grande influência sobre os ganhos energéticos que podem ser extraídos com uso da arquitetura proposta nessa dissertação. Aplicações com comportamentos altamente *data flow* têm um grande volume de acessos à memória, permitindo, dessa forma, uma grande eficiência no gerenciamento de memória através do *clock gating* e da técnica DCF. Entretanto, em aplicações desse tipo, a saturação do barramento tende a aparecer rapidamente quando o número de processadores do MPSoC é elevado. Por outro lado, o uso de aplicações com comportamento *control flow*, faz com que o aproveitamento do *clock gating* e da técnica DCF na redução energética apareçam quando há um número relativamente grande de processadores no MPSoC, já que o volume de acessos à memória é substancialmente pequeno. Pôde-se observar, ainda, que nos experimentos baseados na aplicação sintética o ganho obtido com o uso do DCF foi bem maior em relação aos experimentos baseados no grupo de *benchmarks* SPECjvm98. Isso se justifica pelo fato de que o número de acessos às memórias foi bastante penalizado no

segundo caso, onde alguns dos *benchmarks* apresentam um comportamento predominantemente *control-flow*, tendo um baixo número de acessos às memórias.

Adicionalmente, como pôde ser observado ao longo desse capítulo, os tipos de processador e organização de memória utilizados no MPSoC também podem causar um impacto sensível no número de instruções executadas, nas contenções do barramento, e, conseqüentemente, na redução energética propiciada pelo uso da arquitetura proposta. Em casos extremos, considerando, por exemplo, que há um número pequeno de processadores somado a um baixo percentual de acessos à memória, o custo em área para implementar a arquitetura proposta é insignificante (na maioria dos casos) em relação à área ocupada pelos processadores do MPSoC.

7 CONCLUSÕES E TRABALHOS FUTUROS

Nessa dissertação foi apresentada uma arquitetura para geração de MPSoCs heterogêneos baseados em comunicação via memória compartilhada que habilita a utilização de diferentes organizações de memória. Foi ilustrado o funcionamento do mecanismo para disparo dinâmico de *clock gating*, usando para evitar colisões e reduzir o consumo energético do MPSoC. Introduziu-se ainda a técnica DCF, que permite aumentar a eficiência energética através da aplicação do *clock gating* nos ciclos ociosos dos processadores durante os acessos à memória.

Adicionalmente, foi apresentado o simulador Hashi, que permite simulações em alto-nível, e com precisão de ciclo, da arquitetura proposta. Foi mostrado como o simulador consegue extrair o comportamento de aplicações Java a partir da análise de arquivos de *trace* de execução. Também foi detalhado o processo de simulação, que necessita de uma modelagem arquitetural prévia dos processadores utilizados no MPSoC, bem como a modelagem das diferentes classes de instrução executando em cada um dos processadores.

Essa dissertação mostrou que a arquitetura proposta tem potencial para ser uma alternativa eficiente no domínio de sistemas embarcados. Os resultados apontaram que a arquitetura proposta pode aumentar drasticamente a eficiência energética com um custo mínimo em área. Foram apresentados experimentos baseados em uma aplicação sintética usando diferentes versões de um processador Java, que indicaram uma redução energética média de 25.9%. Adicionalmente, com o DCF ativo pôde-se aumentar a eficiência energética em 16.8% em média. Outros experimentos usando aplicações do pacote de *benchmarks* SPECjvm98 indicaram uma redução energética médios de 22.1%. Com o DCF ativo pôde-se aumentar em média 4.2% a eficiência energética.

Finalmente, esse trabalho também abre várias perspectivas de trabalhos futuros, que serão descritos na próxima seção.

7.1 Trabalhos Futuros

7.1.1 Implementação de Memórias Cache

Memórias cache tiram proveito do princípio de localidade das informações em sistemas computacionais, permitindo que o desempenho geral do processador seja incrementado através da redução dos acessos às memórias mais lentas. Além disso, com essa implementação seria possível utilizar um número maior de organizações de memória, fornecendo assim um leque maior de alternativas para exploração do espaço de projeto.

7.1.2 Experimentos com Outros Processadores

Como a arquitetura proposta é independente de plataforma, um experimento interessante seria a utilização de outros processadores no processo de simulação. Dessa forma, poderia se avaliar se o grau de impacto em eficiência energética, colisões, e desempenho observado nos processadores Java, utilizados nos experimentos dessa dissertação, se aplicam em mesmo nível às demais arquiteturas.

7.1.3 Particionamento da Memória Compartilhada

O uso de uma única memória compartilhada tende a reduzir o desempenho do MPSoC quando o número de processadores é elevado, devido ao alto número de contenções. Essa situação pode ser agravada ainda se houver uma alta taxa de comunicação entre as tarefas em execução. Uma solução para esse problema é particionar a memória compartilhada, alocando as partições resultantes para diferentes grupos de processadores. Essa solução também gera novos problemas a serem solucionados. É necessário determinar, por exemplo, qual o número de processadores que seria alocado para cada partição de memória, e se o endereçamento das várias partições de memória é global ou individual. Em termos arquiteturais, se o endereçamento das memórias for individual, a modificação necessária é relativamente simples. Entretanto seria necessário o desenvolvimento de um mecanismo para permitir a troca de informações entre os diversos grupos de processadores. Por outro lado, se o endereçamento for global, seria necessária uma modificação no processo de geração das memórias.

7.1.4 Modificação na Estrutura de Interconexão

A utilização da estrutura de interconexão por um barramento simples cria uma barreira no nível de escalabilidade que pode ser atingido com a arquitetura proposta, devido ao elevado número de contenções geradas. Já o uso de um mecanismo de interconexão baseado em NoCs pode reduzir de forma dramática esse número de contenções, exigindo um certo custo em área. Essa modificação poderia englobar ainda a sugestão citada na subseção anterior. Dessa forma, o componente *shared-d block* poderia ter conexões com várias memórias compartilhadas por meio da NoC, como mostrado na Figura 7.1, que, apenas para exemplificar, se baseia na organização de memória Shared-D. Como pode ser observado, adicionalmente, nesse exemplo os processadores estão todos conectados ao componentes *Arbiter* e não na NoC.

Para implementar esse mecanismo seria necessária a criação de uma estrutura para realizar o empacotamento/desempacotamento dos dados que serão enviados através da NoC (componente *Pack Up/Down Mechanism*, da Figura 7.1). O componente *Arbiter* também necessitaria de pequenas modificações. Primeiramente, o componente estaria agora habilitado a enviar múltiplas requisições de acesso através do mecanismo de interconexão, e essas múltiplas requisições seriam limitadas pelo tamanho dos *buffers* da NoC. Então, a primeira modificação exigiria uma pequena modificação no componente *Arbiter*, para permitir que ele redirecionasse várias requisições simultaneamente. A segunda modificação vem em decorrência da variabilidade do tempo que pode levar para os dados trafegarem pela NoC. Dessa maneira, o DCF teria que ser usado obrigatoriamente para congelar o estado dos processadores que aguardam os dados da memória, sendo que o tempo de congelamento deveria ser controlado por um novo e simples mecanismo (dentro do componente *Arbiter*), que simplesmente detectaria quando um dado requisitado por algum processador do MPSoC já está

disponível. Esse papel é desempenhado pelo componente *Dynamic Accesses Detection & Requisition Controller*, que pode ser visualizado na Figura 7.1.

A utilização de uma NoC nesse contexto habilita ainda a comunicação por meio de troca de mensagens entre os processadores, que não é provida pela arquitetura atual. Entretanto, seria necessária a realização de experimentos para verificar se o tempo para o envio das informações, que é arbitrário em uma NoC, não causaria atrasos significativos durante os acessos à memória.

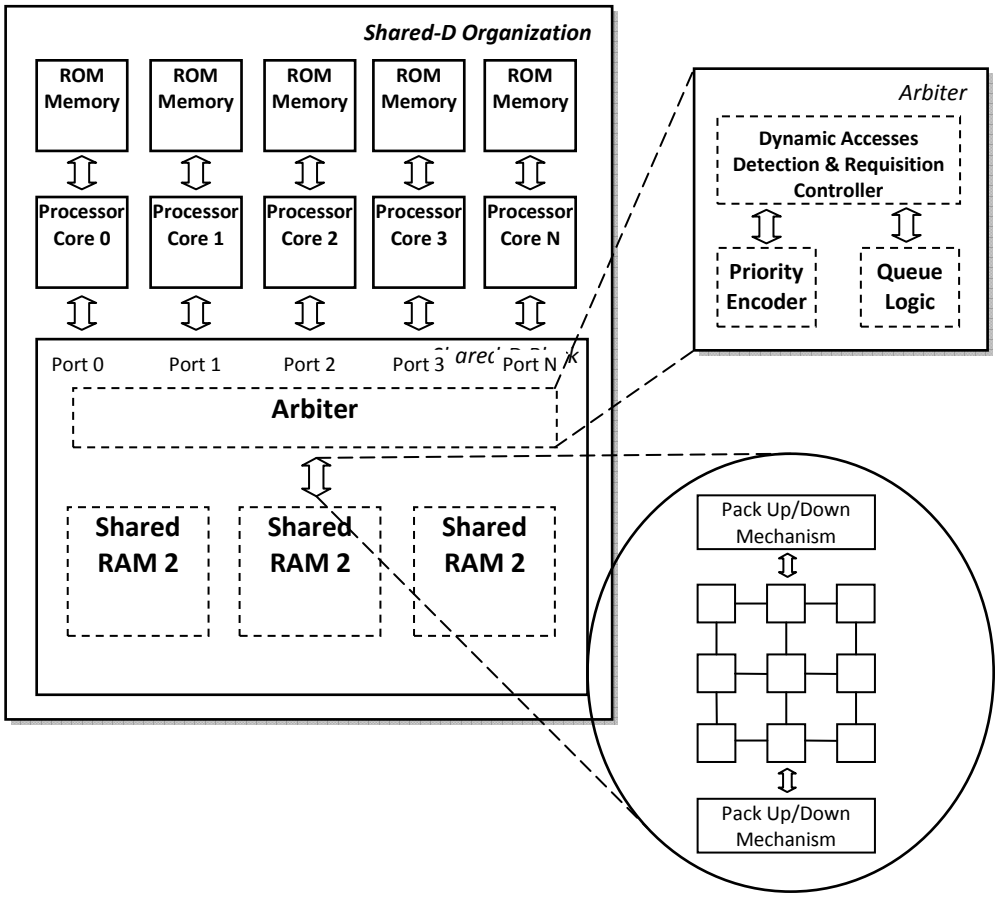


Figura 7.1: Organização Shared-D com uma NoC e múltiplas memórias compartilhadas

7.1.5 Experimentos com Migração de Tarefas

A infra-estrutura apresentada nessa dissertação oferece os recursos necessários para implementar a migração de tarefas nos processadores do MPSoC. O uso do processador FemtoJava para esse propósito torna o processo bastante simples, pois o contexto da tarefa é salvo na pilha do processador. Como no FemtoJava Multiciclo a pilha é mapeada em memória, que é compartilhada na arquitetura proposta, basta fazer uma simples atualização no registrador PC de cada processador envolvido no processo para migrar uma tarefa. Já no FemtoJava Low-Power a pilha é mapeada no banco de registradores, dentro do processador. O processo de migração de tarefas nesse processador exige o disparo de uma interrupção para que o contexto de uma tarefa em execução seja salvo na memória, além da atualização do registrador PC no processador que receberá a tarefa. Em ambos os casos, é necessário o desenvolvimento de um mecanismo para efetuar o disparo da interrupção e atualizar o registrador PC dos processadores envolvidos na migração. Analisando em um nível um pouco mais alto, é

necessário ainda um mecanismo que decida quando uma migração deve ser realizada e quais os processadores que devem fazer parte do processo. Entretanto, para a realização de experimentos preliminares esse último mecanismo pode ser negligenciado, e o processo de decisão sobre a migração pode ser realizado automaticamente de acordo com um conjunto de critérios específicos. Adicionalmente, seria interessante analisar o *overhead* do processo de migração, que tende a ser insignificante no processador FemtoJava Multiciclo, e bastante pequeno no FemtoJava Low-Power.

REFERÊNCIAS

ACQUAVIVA, A. et al. A Simulation Model for Streaming Applications over a Power Manageable Wireless Link. In: INTERNATIONAL WORKSHOP ON MODELING ANALYSIS AND SIMULATION OF WIRELESS AND MOBILE SYSTEMS, MSWiM, 2004, Venice. **Proceedings…** New York: ACM Press, 2004. p. 39-46.

ALDEC Homepage. Disponível em: <http://www.aldec.com/products/active-hdl>. Acesso em: 12 abr. 2008.

AGARWAL, N.; DIMOPOULOS, N. High Level Fixed Point VLSI Design with Automated Clock Gating. In: IEEE PACIFIC RIM CONFERENCE ON COMMUNICATIONS, COMPUTERS AND SIGNAL PROCESSING, PacRim, 2007. **Proceedings**… Washington: IEEE Computer Society, 2007. p. 359-362.

ASLOT , V. et al. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In: INTERNATIONAL WORKSHOP ON OPENMP APPLICATIONS AND TOOLS: OPENMP SHARED MEMORY PARALLEL PROGRAMMING, 2001. **Proceedings…** London: Springer-Verlag, 2001. p. 1-10.

AUSTIN, T. et al. Mobile Supercomputers. **Computer**, New York, v. 37, n. 5, p. 81-83, May. 2004.

BECK FILHO, A.C.S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION,VLSI-SoC, Darmstadt, 2003. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 239-245.

BECK FILHO, A.C.S. **Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java**. 2004. Dissertação(Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BECKMANN, B.M.; MARTY, M.R.; WOOD, D. A. ASR: Adaptive Selective Replication for CMP Caches. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 39., Orlando, 2006. **Proceedings...** Washington: IEEE Computer Society, 2003. p..443-454.

BENINI, L.; SIEGEL, P.; MICHELI, D.G. Saving power by synthesizing gated clocks for sequential circuits. **IEEE Design & Test of Computers**, California, v. 11, n. 4, p. 32-40, Dec. 1994.

BERTOZZI, S. et al. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, Munich, 2006. **Proceedings…** Belgium: European Design and Automation Association, 2006. p. 1-6.

BHUNIA, S.et al. Deterministic clock gating for microprocessor power reduction. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, HPCA, Anaheim, 2003. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 113-122.

BOUCHEBABA, Y. et al. MPSoC memory optimization for digital camera applications. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN ARCHITECTURES, METHODS AND TOOLS, DSD, 2007, Lübeck. **Proceedings…** Washington: IEEE Computer Society, 2007. p. 424-427.

BRIAO, E.W. et al. Impact of task migration in NoC-based MPSoCs for soft real-time applications. In: IFIP INTERNATIONAL VERY LARGE SCALE INTEGRATION, VLSI-SoC, Atlanta, 2007. **Proceedings…** Washington: IEEE Computer Society, 2007. p. 296-299.

BROOKS, D. et al. Power-Aware Micro-Architecture: Design and Modeling Challenges for Next-Generation Microprocessors. **IEEE Micro**, Los Alamitos, v. 20, n. 6, p. 24-44, Nov.-Dec. 2000.

CHANG, J. et al. The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series. **IEEE Journal of Solid-State Circuits**, New York, v. 42, n. 4, p. 846-852, April 2007.

CHANG , J.; SOHI, G.S. Cooperative Caching for Chip Multiprocessors. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 2006, Boston. **Proceedings…** Washington: IEEE Computer Society, 2007. p. 264-276.

CHISHTI, Z.; POWELL, M.D.; VIJAYKUMAR, T.N. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, Madison , 2005. **Proceedings…** Washington: IEEE Computer Society, 2005. p. 357-368.

DICK, R. et al. Analysis of Power Dissipation in Embedded Systems using Real-Time Operating Systems. **IEEE Transactions on CAD**, New York, v. 22, n. 5, p. 615-627, May 2003.

GEER, D. Chip makers turn to multicore processors. **Computer**, New York, v. 38, n. 5, p. 11-13, May 2005.

GEPPERT, L. Sun's big splash [Niagara microprocessor chip]. **IEEE Spectrum**, New York, v. 42, n. 1, p. 56-60, Jan. 2005.

GIVARGIS, T.; VAHID, F.; HENKEL, J. Instruction-Based System-Level Power Evaluation of System-on-a-Chip Peripheral Cores. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, ISSS, 2002, Madrid. **Proceedings…** Washington: IEEE Computer Society, 2002. p. 163-169.

GOWAN, M.; BIRO, L.; JACKSON, D. Power considerations in the design of the Alpha 21264 microprocessor. In: DESIGN AUTOMATION CONFERENCE, DAC, 1998, San Francisco. **Proceedings…** New York: ACM Press, 1998. p. 726-731.

GRIMHEDEN, M.;TÖRNGREN, M. What is embedded systems and how should it be taught?---results from a didactic analysis. **ACM Transactions on Embedded Computing Systems**, New York, v.4, n.4, p. 633-651, 2005.

GURUMURTHI, S. et al. Using Complete Machine Simulation for Software Power Estimation: the SoftWatt Approach. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, ISCA, Cambridge, 2002. **Proceedings…** Washington: IEEE Computer Society, 2002. p. 124-133.

GUZ, Z. et al. Nahalal: Memory Organization for Chip Multiprocessors. **IEEE Computer Architecture Letters**, [S.l.], v. 6, n. 1, 2007.

HENKEL, J.; LI, Y. Avalanche: an Environment for Design Space Exploration and Optimization of Low-Power Embedded Systems. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, New York, v.10, n.4, p. 454-468, 2002.

HENNESSY, J.L. et al. **Computer architecture:** a quantitative approach. 3rd ed. San Francisco: Morgan Kaufmann Publishers, 2003.

IPEK, E. et al. A Reconfigurable Chip Multiprocessor Architecture to Accommodate Software Diversity. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 2007, Florida. **Proceedings…** Washington: IEEE Computer Society, 2007. p. 1-6.

ITO, S.; CARRO, L.; JACOBI, R. Making Java Work for Microcontroller Applications. **IEEE Design & Test Archive**, California, v. 18, n. 5, p. 100-110, 2001.

KALLA, R.; SINHAROY, B.; TENDLER, J.M. IBM Power5 chip: a dual-core multithreaded processor. **IEEE Micro**, Los Alamitos, v. 24, n. 2, p. 40-47, Mar.-Apr. 2004.

KOGEL, T.; MEYR, H. Heterogeneous MP-SoC – the solution to energy-efficient signal processing. In: DESIGN AUTOMATION CONFERENCE, DAC, 2004, Anaheim. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 686-691.

KRISHNAN, R. **Future of Embedded Systems Technology**. Disponível em: <http://www.electronics.ca/reports/embedded/systems_technology.html>. Acesso em: 10 jan. 2008.

KUMAR, R. et al. Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures. **IEEE Computer Architecture Letters**, [S.l.], v.1, n.1, p. 5-8, Jan. 2006.

LAJOLO, M. et al. Cosimulation-Based Power Estimation for System-on-Chip Design. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, New York, v. 10, n. 3, p. 253-266, June 2002.

LANG, T.; MUSOLL, E.; CORTADELLA, J. Individual flip-flops with gated clocks for low power datapaths. **IEEE Transactions on Circuits and Systems II**, New York, v. 44, n. 6, p. 507–516, June 1997.

LEIBSON, S.; KIM, J. Configurable processors: a new era in chip design. **Computer**, New York, v. 38, n. 7, p. 51-59, July 2005.

LOGHI, M. et al. Analyzing On-chip communication in a MPSoC Environment. In: DESIGN, AUTOMATION, AND TEST IN EUROPE, DATE, 2004, Munich. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 752–757.

MAGARSHACK, P.; PAULIN, P. System-on-chip beyond the nanometer wall. In: DESIGN AUTOMATION CONFERENCE, DAC, 2003, Anaheim. **Proceedings…** New York: ACM Press, 2003. p. 419-424.

MARTIN, G. Overview of the MPSoC design challenge. In: DESIGN AUTOMATION CONFERENCE, DAC, 2006, San Francisco. **Proceedings…** New York: ACM Press, 2006. p. 274-279.

MENTOR Homepage. Disponível em: <http://www.mentor.com/Synthesis>. Acesso em: 10 jan. 2008.

MUELLER, M. et al. The impact of clock gating schemes on the power dissipation of synthesizable register files. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2004, Vancouver. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 609-612.

NACUL, A.C.; REGAZZONI, F.; LAIOLO, M. Hardware Scheduling Support in SMP Architectures. In: DESIGN, AUTOMATION, AND TEST IN EUROPE, DATE, 2007, Nice. **Proceedings…** San Jose: EDA Consortium, 2007. p. 642-647.

NANRI, T.; WATANABE, Y.; SATO, H. Performance comparison of vector-calculations between Itanium2 and other processors. In: INNOVATIVE ARCHITECTURE ON FUTURE GENERATION HIGH-PERFORMANCE PROCESSORS AND SYSTEMS, IWIA, 2005, Oahu. **Proceedings…** Washington: IEEE Computer Society, 2005. p. 141-146.

O'CONNOR, J.M.; TREMBLAY, M. picoJava-I: the Java virtual machine in hardware. **IEEE Micro**, [S.l.], v. 17, n. 2, p. 45-53, 1997.

OLUKOTUN K. et al. The case for a single-chip multiprocessor. **ACM SIGPLAN Notices**, New York, v. 31 , n. 9, p. 2-11, 1996.

PAULIN, P.G. et al. Application of a multi-processor SoC platform to high-speed packet forwarding. In: DESIGN, AUTOMATION, AND TEST IN EUROPE, DATE, 2004, Munich. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 58-63.

PENG, L. et al. Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study. In: INTERNATIONAL PERFORMANCE, COMPUTING, AND COMMUNICATIONS CONFERENCE, IPCCC, 2007, Louisiana. **Proceedings…** Washington: IEEE Computer Society, 2007. p. 55-64.

PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, Cambridge, 1996. **Proceedings…** New York: ACM Press, 1996. p. 2-11.

RAGHAVAN, N.; AKELLA, V.; BAKSHI, S. Automatic insertion of gated clocks at register transfer level. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, VLSID, 1999, Goa. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 48–54.

RICHTER, K.; RACU, R.; ERNST, R. Scheduling analysis integration for heterogeneous multiprocessor SoC. In: INTERNATIONAL REAL-TIME SYSTEMS SYMPOSIUM, RTSS, 2003, Cancun. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 236-245.

SASANKA, R. et al. The energy efficiency of CMP vs. SMT for multimedia workloads. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 2004, France. **Proceedings…** New york: ACM Press, 2004. p. 196-206.

SCHAUMONT, P. et al. Cooperative multithreading on embedded multiprocessor architectures enables energy-scalable design. In: DESIGN AUTOMATION CONFERENCE, DAC, 2005, Anaheim. **Proceedings…** New york: ACM Press, 2005. p. 27-30.

SHEYNIN, Y.; SUVOROVA, E.; SHUTENKO, F. Complexity and low power issues for on-chip interconnections in MPSoC system level design. In: COMPUTER SOCIETY ANNUAL SYMPOSIUM ON EMERGING VLSI TECHNOLOGIES AND ARCHITECTURES, ISVLSI, 2006, Karlsruhe. **Proceedings…** Washington: IEEE Computer Society, 2006. p. 283.

SIMUNIC, T.; BENINI, L.; DE MICHELI, G. Energy-Efficient Design of Battery Powered Embedded Systems. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 2001, San Diego. **Proceedings…** New York: ACM Press, 2001. p. 212-217.

SOININEN, J.-P.; PELKONEN, A.; ROIVAINEN, J. Configurable memory organisation for communication applications. In: EUROMICRO SYMPOSIUM ON DIGITAL SYSTEMS DESIGN, DSD, 2002, Dortmund. **Proceedings…** Washington: IEEE Computer Society, 2002. p. 86-93.

SPECjvm98 Homepage. Disponível em: <http://www.spec.org/osg/jvm98/>. Acesso em: 10 jan. 2008.

STRIK, M. et al. Heterogeneous multi-processor for the management of real-time video and graphics streams. **IEEE Journal of Solid-State Circuits**, Piscataway, v. 35, n. 11, p. 244-245, 2000.

SYNOPSYS Homepage. Disponível em: <http://synopsys.com/products/power/power_ds.html>. Acesso em: 10 jan. 2008.

TAN, T.; RAGHUNATHAN, A.; JHA, N. Embedded Operating System Energy Analysis and Macro-Modeling. In: CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, ICCD, 2002, Freiburg. **Proceedings…** Washington: IEEE Computer Society, 2002. p. 515-222.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2. ed. São Paulo, Brasil: Prentice Hall, 2003.

TIWARI , V.; MALIK, S.; WOLFE, A. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. **IEEE Transactions on Very Large Scale Integration (VLSI) Sustems**, New York, v.2, n. 1, p. 437-445, Dec. 1994..

TSMC Homepage. Disponível em: <http://www.tsmc.com.tw>. Acesso em: 10 jan. 2008.

VIJAYKRISHNAN, N. et al. Evaluating Integrated Hardware-Software Optimizations using a Unified Energy Estimation Framework. **IEEE Transactions on Computers**, Los Alamitos, v. 52, n. 1, p. 59-76, 2003.

YONGHONG, S.; KALOGEROPULOS, S.; TIRUMALAI, P. Design and implementation of a compiler framework for helper threading on multi-core processors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, Saint Louis, 2005. **Proceedings…** Washington: IEEE Computer Society, 2005. p.99-109.

WARNOCK, J.D. Circuit design issues for the POWER4 chip. In: INTERNATIONAL SYMPOSIUM ON VLSI TECHNOLOGY, SYSTEMS, AND APPLICATIONS, 2003, Florida. **Proceedings...** Washington: IEEE Computer Society, 2003. p.125-128.

WILCOX, K.; MANNE, S. Alpha processors: A history of power issues and a look to the future. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO-33, 1999, Haifa. **Proceedings…** Washington: IEEE Computer Society, 1999.

WOLF, W. H. **Computers as components:** principles of embedded computing system design. San Francisco, CA: Morgan Kaufmann Publishers, 2001.

WINEGARDEN, S. Bus Architecture of a System on a Chip with User Configurable System Logic. **IEEE Journal of Solid State Circuits**, New York, v. 35, n. 3, p. 425-433, 2000.

WOLF, W.; KANDEMIR, M. Memory system optimization of embedded software. **Proceedings of the IEEE**, New York, v. 91, n. 1, p. 165-182, 2003.

WOLF, W. The future of multiprocessor systems-on-chips. In: DESIGN AUTOMATION CONFERENCE, DAC, 2004, San Diego . **Proceedings…** New York: ACM Press, 2004. p. 681-685.

WOLFE, M. Supercompilers, the AMD Opteron, and your cell phone. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDS, 2004. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 98-104.

WOO, S.C. et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 1995, Ligure. **Proceedings…** New York: ACM Press , 1995. p. 24–37.

APÊNDICE A. TABELAS DE CONTENÇÃO, DESEMPENHO E ENERGIA EXECUTANDO APLICAÇÃO SINTÉTICA

Tabela A.1: Organização Shared-D com config. P

Configuration P + Shared-D Organization

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	8.143	678.038	163.785	6.790.610	28.237	0.41
4	58.355	1.337.302	323.787	13.436.015	202.304	1,48
8	406.264	2.579.825	624.243	25.882.574	1.408.550	5,16
16	4.085.138	4.042.427	977.368	40.603.660	14.154.565	25.85
32	19.795.275	4.132.567	1.000.000	41.579.748	68.586.768	62.26
64	51.881.460	4.129.967	1.000.000	41.326.120	179.968.832	81,33
128	115.892.156	4.128.509	1.000.000	41.290.000	401.127.616	90.67
256	243.843.429	4.131.655	1.000.000	41.422.092	846.273.600	95,33
512	499.814.102	4.136.247	1.000.000	41.533.656	1.728.081.664	97,65

Tabela A.2: Organização Shared-D com config. M

Configuration M + Shared-D Organization

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	174.345	285.705	656.825	721.553	71.368	9,00
4	1.329.767	415.296	955.320	1.055.748	546.984	34,13
8	5.212.074	434.028	999.985	1.102.621	2.138.259	65.98
16	13.194.467	434.894	1.000.000	1.109.468	5.634.733	83,55
32	29.192.572	434.553	1.000.000	1.110.583	12.737.318	91,98
64	61.190.266	435.090	1.000.000	1.111.324	25.385.568	95.81
128	125.202.684	435.474	1.000.000	1.106.479	51.667.164	97.90
256	253.201.959	435.341	1.000.000	1.106.826	111.213.432	99,01
512	509.205.128	435.289	1.000.000	1.105.192	216.068.368	99.49

Tabela A.3: Organização Shared-D com config. H

Configuration H + Shared-D Organization

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	54.525	477.293	437.024	3.615.413	189.058	4.97
4	507.317	858.441	771.812	6.578.310	1.047.200	13.73
8	3.493.307	1.110.829	995.106	8.496.607	6.839.656	44,60
16	11.476.188	1.115.918	1.000.000	8.515.813	22.205.806	72,28
32	27.492.066	1.116.773	1.000.000	8.500.369	53.259.612	86.24
64	59.484.673	1.110.118	1.000.000	8.509.618	116.130.680	93.17
128	123.489.252	1.110.430	1.000.000	8.504.274	242.091.952	96.61
256	251.489.101	1.109.299	1.000.000	8.501.912	484.970.688	98.28
512	507.477.460	1.111.908	1.000.000	8.528.836	988.798.976	99,14

Tabela A.4: Organização Mixed-D com config. P e 10% de acessos à MC

Configuration P + Mixed-D Organization and 10% of memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	68	680.123	16.384	6.814.821	236	0,00
4	466	1.360.710	32.868	13.629.061	1.615	0,01
8	1.979	2.709.741	64.921	27.246.068	6.862	0.03
16	9.223	5.434.525	131.209	54.494.000	32.214	0.06
32	44.356	10.857.934	260.039	108.904.784	153.754	0.14
64	277.202	21.653.410	519.779	217.209.472	960.988	0.44
128	8.377.039	40.639.964	974.788	407.700.000	29.030.188	6,65
256	133.404.846	41.623.649	1.000.000	417.814.304	462.854.656	52,56
512	389.153.876	41.749.708	1.000.000	418.723.648	1.345.764.096	76.27

Tabela A.5: Organização Mixed-D com config. P e 50% de acessos à MC

Configuration P + Mixed-D Organization and 50% of memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	1.885	682.551	82.168	6.815.923	6.530	0,10
4	11.837	1.346.526	161.453	13.581.510	41.018	0.30
8	68.304	2.702.963	324.237	27.042.484	236.776	0.87
16	477.152	5.271.514	631.685	52.895.744	1.653.971	3.03
32	7.574.818	8.277.268	993.467	83.235.448	26.238.648	23.97
64	39.376.195	8.344.364	1.000.000	83.904.720	136.541.504	61.94
128	103.489.762	8.322.734	1.000.000	83.533.872	357.798.656	81,07
256	231.487.120	8.327.303	1.000.000	83.534.728	801.003.584	90.56
512	487.512.558	8.330.842	1.000.000	83.478.616	1.688.465.664	95.29

Tabela A.6: Organização Mixed-D com config. M e 10% de acessos à MC

Configuration M + Mixed-D Organization and 50% of memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	1.746	310.718	74.254	790.433	471	0.06
4	11.349	619.485	148.640	1.578.433	3.064	0,19
8	60.464	1.234.337	296.495	3.142.151	16.326	0,52
16	381.085	2.427.347	581.450	6.180.040	102.853	1.64
32	5.692.967	4.081.511	981.279	10.405.808	1.538.058	12,88
64	37.237.979	4.158.746	1.000.000	10.586.840	10.017.534	48,62
128	101.164.627	4.165.293	1.000.000	10.614.965	27.203.040	71.93
256	229.069.179	4.178.397	1.000.000	10.654.539	61.587.960	85,25
512	485.148.204	4.168.864	1.000.000	10.623.182	130.430.456	92,47

Tabela A.7: Organização Mixed-D com config. M e 50% de acessos à MC

Configuration M + Mixed-D Organization and 10% of memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	61	310.453	14.714	791.929	17	0,00
4	406	619.192	29.936	1.581.856	110	0,01
8	2.104	1.241.529	60.102	3.164.109	569	0.02
16	8.950	2.480.545	118.291	6.326.752	2.417	0.04
32	42.933	4.958.339	238.300	12.642.179	11.592	0.09
64	245.905	9.899.135	474.502	25.222.752	66.397	0.26
128	4.344.326	19.196.345	922.108	48.920.708	1.171.852	2,34
256	121.842.029	20.818.040	1.000.000	53.073.500	32.771.442	38,18
512	377.732.652	20.832.088	1.000.000	53.117.824	101.562.048	65.66

Tabela A.8: Organização Mixed-D com config. H e 10% de acessos à MC

Configuration H + Mixed-D Organization and 10% of memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	72	495.076	15.449	3.804.695	20	0,00
4	438	990.086	31.081	7.606.281	633	0,01
8	2.000	1.986.770	62.694	15.216.324	3.665	0.02
16	9.449	3.953.218	124.428	30.401.376	18.167	0.06
32	45.784	7.899.353	248.297	60.771.616	91.724	0.15
64	276.261	15.778.637	498.932	121.168.472	571.094	0.47
128	6.145.313	30.061.269	950.717	230.330.464	12.993.556	5,34
256	127.642.097	31.583.901	1.000.000	241.675.232	239.055.728	49,73
512	383.551.819	31.630.897	1.000.000	241.803.600	717.711.552	74.80

Tabela A.9: Organização Mixed-D com config. H e 50% de acessos à MC

Configuration H + Mixed-D Organization and 50% of memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
2	1.660	492.294	77.750	3.799.834	449	0,01
4	11.927	991.821	156.206	7.596.506	17.301	0.23
8	65.392	1.965.432	310.507	15.090.850	115.574	0.76
16	429.386	3.850.339	605.457	29.606.808	813.923	2.68
32	6.622.319	6.272.381	988.373	48.189.116	12.447.588	20.53
64	38.349.220	6.353.277	1.000.000	48.755.868	71.790.632	59,55
128	102.344.533	6.332.223	1.000.000	48.715.888	191.328.768	79,71
256	230.336.733	6.345.098	1.000.000	48.743.532	430.599.264	89.83
512	486.385.192	6.342.371	1.000.000	48.646.840	906.475.008	94.91

Tabela A.10: Organização Mixed-D e DCF ativo sobre a MC e MP com config. P e 10% de acessos à MC

DCF over SM and PM
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
1	0	317.745	7.643	3.144.257	266.543	7.81
2	300	637.468	15.254	6.295.622	536.586	7.85
4	1.726	1.267.185	30.578	12.581.086	1.071.441	7,85
8	9.280	2.534.384	60.952	25.158.592	2.157.633	7.90
16	43.944	5.061.715	121.458	50.219.000	4.400.542	8.06
32	257.566	10.105.855	243.253	99.935.864	9.371.252	8,57
64	3.721.730	19.155.744	459.283	189.774.736	28.974.020	13,25
128	62.447.546	20.825.864	500.000	206.337.984	234.256.544	53.17
256	190.505.259	20.828.162	500.000	206.154.688	678.872.064	76,71
512	446.440.111	20.841.370	500.000	206.395.376	1.559.647.744	88.31

Tabela A.11: Organização Mixed-D e DCF ativo sobre a MC e MP com config. P e 50% de acessos à MC

DCF over SM and PM
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
1	0	317.610	37.799	3.152.525	265.635	7.77
2	7.961	632.135	75.377	6.272.508	556.926	8,15
4	55.212	1.252.388	150.469	12.418.102	1.243.305	9.10
8	364.657	2.429.673	291.949	24.029.912	3.305.555	12.09
16	3.421.537	4.002.510	478.959	39.604.700	15.210.155	27.75
32	18.891.456	4.169.389	5.000.000	41.278.888	69.063.904	62.59
64	50.864.283	4.173.116	5.000.000	41.349.672	179.311.984	81.26
128	114.847.493	4.169.597	5.000.000	41.396.236	400.769.408	90.64
256	242.898.284	4.166.744	5.000.000	41.240.616	845.617.920	95.35
512	498.864.861	4.169.188	5.000.000	41.348.956	1.733.328.384	97.67

Tabela A.12: Organização Mixed-D e DCF ativo sobre a MC com config. P e 10% de acessos à MC

DCF over SM
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
1	0	320.165	7.750	3.391.964	26.861	0,79
2	282	639.717	15.187	6.774.261	53.610	0.79
4	1.782	1.269.881	30.493	13.544.087	111.902	0,82
8	9.222	2.541.910	60.857	27.078.610	242.961	0,89
16	44.912	5.074.758	121.947	54.048.524	578.479	1.06
32	256.092	10.093.609	242.063	107.544.664	1.726.916	1.58
64	3.729.188	19.139.653	459.015	204.173.584	14.521.169	6,64
128	62.494.472	20.811.923	5.000.000	221.909.136	217.556.576	49.50
256	190.381.684	20.837.983	5.000.000	222.269.552	660.274.944	74,81
512	446.252.698	20.892.896	5.000.000	222.724.944	1.544.966.400	87,40

Tabela A.13: Organização Mixed-D e DCF ativo sobre a MC com config. P e 50% de acessos à MC

DCF over SM
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
1	0	317.300	37.822	3.281.539	131.121	3.84
2	7.885	632.563	75.888	6.538.507	290.406	4.25
4	56.328	1.257.323	150.761	12.945.690	717.838	5.25
8	359.840	2.425.872	290.512	25.081.452	2.254.449	8.25
16	3.417.933	4.001.416	478.560	41.305.992	13.502.350	24,64
32	18.907.219	4.160.313	5.000.000	42.963.812	67.084.408	60,96
64	50.916.319	4.153.730	5.000.000	42.942.972	178.148.016	80.58
128	178.148.016	4.163.674	5.000.000	42.991.024	400.178.944	90.30
256	242.897.694	4.166.762	5.000.000	42.999.080	841.449.664	95,14
512	498.872.585	4.165.577	5.000.000	43.075.420	1.730.569.472	97.57

APÊNDICE B. TABELAS DE CONTENÇÃO, DESEMPENHO E ENERGIA EXECUTANDO APLICAÇÕES DO BENCHMARK SPECJVM98

Tabela B.1: JESS sobre config. P e 10% de acessos à MC

JESS + CLOCK GATING
Configuration P + Mixed-D Organization and 10% Shared memory accesses

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	82	548.772	11.026	20.233.450	21	0,00
8	587	1.100.085	21.965	40.475.692	1.782	0,00
16	2.969	2.193.407	44.037	80.891.736	10.397	0,01
32	14.512	4.401.115	88.227	161.704.912	58.252	0,04
64	68.867	8.808.190	176.264	323.060.352	320.199	0,10
128	472.677	17.538.528	351.275	643.108.800	2.765.592	0,43
256	33.572.333	30.261.088	605.886	1.099.096.960	186.841.040	14,53

Tabela B.2: JESS sobre config. P e 50% de acessos à MC

JESS + CLOCK GATING
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.072	553.547	55.328	20.607.704	857	0,00
8	17.009	1.097.434	109.742	41.002.952	53.860	0,13
16	103.629	2.180.229	218.210	81.364.128	444.907	0,54
32	837.242	4.268.808	427.212	157.849.008	4.667.255	2,87
64	19.190.843	6.082.607	607.825	222.437.296	98.886.024	30,77
128	83.234.529	6.063.130	607.597	222.246.976	414.226.336	65,08
256	211.114.227	6.088.303	608.709	222.829.680	1.048.460.160	82,47

Tabela B.3: JESS sobre config. M e 10% de acessos à MC

JESS + CLOCK GATING
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	102	291.793	5.843	2.297.806	6	0,00
8	807	583.942	11.540	4.595.472	266	0,01
16	3.850	1.166.171	23.373	9.180.933	1.528	0,02
32	17.610	2.333.477	46.601	18.359.348	7.395	0,04
64	87.104	4.659.074	92.586	36.667.696	40.860	0,11
128	623.445	9.275.159	185.523	72.928.664	324.454	0,44
256	51.413.036	14.740.350	295.330	114.744.856	21.365.352	15,70

Tabela B.4: JESS sobre config. M e 50% de acessos à MC

JESS + CLOCK GATING
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.287	289.683	28.966	2.311.238	146	0,01
8	19.494	581.928	58.110	4.610.850	6.242	0,14
16	121.992	1.154.773	115.823	9.137.308	50.529	0,55
32	1.036.064	2.239.666	223.233	17.620.352	491.860	2,72
64	22.480.959	2.980.602	298.091	23.185.072	8.997.244	27,96
128	86.485.913	2.983.422	298.346	23.189.192	33.474.032	59,08
256	214.465.206	2.983.548	298.535	23.209.952	82.433.384	78,03

Tabela B.5: JESS sobre config. H e 10% de acessos à MC

JESS + CLOCK GATING
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	127	383.089	7.528	6.228.150	73	0,00
8	695	769.922	15.436	12.464.333	616	0,00
16	3.269	1.533.704	30.862	24.898.652	3.098	0,01
32	15.707	3.089.214	61.738	49.814.760	15.620	0,03
64	77.812	6.145.813	123.178	99.506.344	87.328	0,09
128	549.550	12.261.385	244.293	198.276.960	741.378	0,37
256	44.296.031	20.203.469	403.878	325.916.864	63.735.548	16,36

Tabela B.6: JESS sobre config. H e 50% de acessos à MC

JESS + CLOCK GATING
Configuration H + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.529	383.511	38.415	6.271.068	1.365	0,02
8	18.352	767.095	76.842	12.533.033	16.725	0,13
16	115.796	1.528.311	152.696	24.901.306	128.645	0,51
32	968.656	2.975.265	297.346	48.474.068	1.323.646	2,66
64	21.473.006	4.049.465	405.684	65.796.596	30.028.088	31,34
128	85.415.654	4.051.827	405.704	65.886.568	118.398.600	64,25
256	213.493.841	4.056.635	405.646	65.828.360	293.310.688	81,67

Tabela B.7: JESS sobre config. P com DCF e 10% de acessos à MC

JESS + CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	61	549.250	10.962	19.272.788	880.861	4,37
8	581	1.097.401	22.325	38.555.824	1.767.246	4,38
16	2.833	2.198.658	44.078	77.130.520	3.530.762	4,38
32	13.799	4.393.994	87.970	154.163.760	7.106.599	4,41
64	70.756	8.785.754	175.788	308.132.480	14.382.715	4,46
128	467.713	17.537.299	351.606	614.526.080	30.828.440	4,78
256	33.671.220	30.228.375	605.469	1.051.049.792	235.402.560	18,30

Tabela B.8: JESS sobre config. P com DCF e 50% de acessos à MC

JESS + CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	1.888	549.888	54.566	19.282.910	951.809	4,70
8	16.567	1.097.029	109.526	38.500.004	1.950.353	4,82
16	102.495	2.179.421	217.825	76.651.288	4.209.489	5,21
32	833.082	4.260.441	426.670	149.607.504	11.957.080	7,40
64	19.271.295	6.065.007	608.397	211.336.528	109.621.808	34,15
128	83.215.974	6.065.904	607.681	211.617.664	422.688.288	66,64
256	211.210.475	6.077.495	607.880	211.580.976	1.060.382.848	83,37

Tabela B.9: JESS sobre config. M com DCF e 10% de acessos à MC

JESS + CLOCK GATING & DCF active
Configuration M + Mixed-D Organization (10% of the memory accesses related to SM)

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	107	291.540	5.860	2.256.941	36.527	1,59
8	719	585.162	11.713	4.516.777	73.011	1,59
16	3.446	1.169.095	23.299	9.027.263	147.005	1,60
32	18.172	2.332.577	46.636	18.049.672	297.818	1,62
64	90.841	4.657.868	93.714	36.050.376	623.021	1,70
128	641.247	9.274.647	185.714	71.720.280	1.485.306	2,03
256	52.285.241	14.670.450	295.159	112.480.992	23.469.754	17,26

Tabela B.10: JESS sobre config. M com DCF e 50% de acessos à MC

JESS + CLOCK GATING & DCF active
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.518	290.794	29.142	2.258.726	39.346	1,71
8	19.773	582.662	58.230	4.503.519	85.232	1,86
16	124.609	1.154.486	115.741	8.938.130	208.165	2,28
32	1.037.661	2.240.269	223.956	17.285.268	793.877	4,39
64	22.518.257	2.981.850	298.197	22.811.880	9.373.118	29,12
128	86.505.406	2.978.712	297.591	22.801.420	33.865.640	59,76
256	214.468.699	2.978.482	298.191	22.810.308	82.830.280	78,41

Tabela B.11: JESS sobre config. H com DCF e 10% de acessos à MC

JESS + CLOCK GATING & DCF active
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	154	384.036	7.718	6.074.387	146.606	2,36
8	727	769.004	15.340	12.160.564	294.502	2,36
16	3.284	1.541.742	30.520	24.328.294	592.019	2,38
32	15.590	3.072.698	60.956	48.619.420	1.193.290	2,40
64	80.613	6.149.604	123.045	97.136.640	2.440.594	2,45
128	550.873	12.262.849	245.030	193.591.920	5.440.554	2,73
256	44.672.841	20.164.155	403.434	317.837.088	72.137.272	18,50

Tabela B.12: JESS sobre config. H com DCF e 50% de acessos à MC

JESS + CLOCK GATING & DCF active
Configuration H + Mixed-D Organization 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.558	385.421	38.661	6.086.776	200.980	3,20
8	19.074	764.374	76.325	12.115.198	412.727	3,29
16	115.314	1.527.193	152.409	24.135.654	914.631	3,65
32	975.856	2.974.737	297.720	47.004.888	2.860.645	5,74
64	21.518.162	4.053.942	405.580	63.804.236	32.253.706	33,58
128	85.519.260	4.047.972	404.809	63.760.456	120.333.304	65,37
256	213.263.645	4.066.194	406.032	64.135.320	296.857.792	82,23

Tabela B.13: MP3 sobre config. P e 10% de acessos à MC

MP3 + CLOCK GATING
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	1.312	976.161	45.574	18.560.164	401	0,00
8	10.573	1.948.215	91.692	37.046.712	30.683	0,08
16	63.680	3.885.084	182.721	73.702.864	257.099	0,35
32	445.457	7.662.462	359.854	144.892.048	2.378.101	1,61
64	10.254.028	12.900.294	604.915	240.502.208	55.997.264	18,89
128	74.121.723	12.929.561	608.730	241.056.768	369.444.320	60,51
256	201.995.219	12.970.589	608.724	241.503.536	999.045.120	80,53

Tabela B.14: MP3 sobre config. P e 50% de acessos à MC

MP3 + CLOCK GATING
Configuration P + Mixed-D Organization (50% of the memory accesses related to SM)

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	44.034	968.339	230.343	19.939.440	26.934	0,13
8	484.484	1.831.977	436.402	37.179.416	1.732.105	4,45
16	5.191.936	2.655.788	631.844	53.414.128	23.356.074	30,42
32	21.151.646	2.670.485	635.054	53.658.236	102.292.400	65,59
64	53.160.153	2.663.393	633.906	53.589.180	258.213.600	82,81
128	117.164.935	2.662.034	634.019	53.594.132	570.512.192	91,41
256	245.144.070	2.667.893	634.731	53.712.124	1.201.054.720	95,72

Tabela B.15: MP3 sobre config. M e 10% de acessos à MC

MP3 + CLOCK GATING
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	785	352.897	16.509	2.001.545	50	0,00
8	6.008	704.760	33.223	3.998.070	2.000	0,05
16	33.807	33.807	66.334	7.976.259	13.522	0,17
32	198.256	2.799.318	131.109	15.837.612	91.434	0,57
64	2.278.643	5.402.150	253.960	30.328.242	1.153.721	3,66
128	55.304.953	6.312.502	296.510	35.180.044	21.891.186	38,36
256	183.454.956	6.298.410	296.380	35.089.020	70.935.240	66,90

Tabela B.16: MP3 sobre config. M e 50% de acessos à MC

MP3 + CLOCK GATING
Configuration M + Mixed-D Organization (50% of the memory accesses related to SM)

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	21.048	352.234	83.516	2.040.666	1.395	0,07
8	205.825	686.990	163.136	3.954.818	68.566	1,70
16	2.275.174	1.201.439	286.157	6.829.272	883.546	11,46
32	17.357.854	1.286.369	306.781	7.315.853	6.617.836	47,50
64	49.337.288	1.287.364	306.659	7.319.179	18.854.176	72,04
128	113.333.377	1.288.026	306.490	7.327.801	43.331.872	85,54
256	241.328.133	1.288.614	306.356	7.329.720	92.300.208	92,64

Tabela B.17: MP3 sobre config. H e 10% de acessos à MC

MP3 SEM CLOCK GATING
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	1.181	575.622	27.207	5.598.422	603	0,01
8	8.776	1.150.425	54.349	11.186.773	8.268	0,07
16	48.484	2.299.471	108.372	22.318.554	55.247	0,25
32	302.982	4.564.254	214.737	44.262.500	410.339	0,92
64	5.291.686	8.388.262	393.370	80.785.504	8.356.077	9,37
128	66.535.713	8.749.364	411.153	84.118.688	93.145.224	52,55
256	194.591.509	8.740.619	410.979	84.057.784	269.408.960	76,22

Tabela B.18: MP3 sobre config. H e 50% de acessos à MC

MP3 + CLOCK GATING
Configuration H + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	35.200	571.658	136.257	5.771.395	19.820	0,34
8	339.511	1.101.666	262.566	11.077.741	389.504	3,40
16	3.851.593	1.737.516	414.115	17.397.886	5.235.218	23,13
32	19.639.966	1.768.020	421.244	17.704.212	26.619.310	60,06
64	51.648.396	1.771.322	421.145	17.703.690	71.506.792	80,16
128	115.628.352	1.774.445	421.685	17.719.122	158.941.424	89,97
256	243.644.754	1.773.210	421.306	17.695.806	336.594.400	95,01

Tabela B.19: MP3 com DCF sobre config. P e 10% de acessos à MC

MP3 + CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	1.224	974.877	45.615	14.616.087	3.688.667	20,15
8	10.672	1.949.314	91.752	29.195.384	7.418.829	20,26
16	64.611	3.883.206	182.182	58.245.712	14.988.632	20,47
32	445.097	7.665.122	360.795	115.258.208	31.320.658	21,37
64	10.433.836	12.861.794	604.748	192.301.008	192.301.008	50,00
128	74.064.518	12.943.078	608.220	193.483.744	416.798.304	68,30
256	202.078.150	12.953.420	609.035	193.526.912	1.044.740.032	84,37

Tabela B.20: MP3 com DCF sobre config. P e 50% de acessos à MC

MP3 + CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	45.689	968.336	231.024	14.549.491	4.023.856	21,66
8	487.511	1.835.548	437.214	27.923.704	9.275.492	24,93
16	5.174.272	2.659.212	631.578	41.265.572	34.198.672	45,32
32	21.160.214	2.665.145	634.753	41.373.276	113.091.696	73,22
64	53.153.412	2.669.838	635.077	41.416.616	269.236.960	86,67
128	117.159.051	2.663.796	634.389	41.396.772	583.216.064	93,37
256	245.183.263	2.664.024	634.813	41.313.796	1.211.913.216	96,70

Tabela B.21: MP3 com DCF sobre config. M e 10% de acessos à MC

MP3 + CLOCK GATING & DCF active
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	745	352.791	16.619	1.885.731	104.445	5,25
8	6.132	6.132	33.068	3.770.342	210.541	5,29
16	34.339	1.407.162	65.943	7.523.929	430.382	5,41
32	430.382	2.801.399	132.174	14.970.690	919.661	5,79
64	2.236.172	5.406.863	253.864	28.738.808	2.714.411	8,63
128	55.501.413	6.295.856	296.262	33.243.002	23.788.102	41,71
256	183.303.951	6.314.960	296.564	33.347.878	72.687.712	68,55

Tabela B.22: MP3 com DCF sobre config. M e 50% de acessos à MC

MP3 + CLOCK GATING & DCF active
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	21.238	351.475	83.724	1.882.503	114.282	5,72
8	205.231	686.277	163.382	3.674.937	287.415	7,25
16	2.263.579	1.202.226	285.968	6.414.428	1.260.630	16,43
32	17.317.542	1.289.479	306.922	6.896.469	7.012.399	50,42
64	49.269.911	1.293.967	307.016	6.925.693	19.234.430	73,53
128	113.295.729	1.292.145	306.758	6.917.828	43.726.804	86,34
256	241.326.130	1.288.982	305.703	6.899.471	92.710.088	93,07

Tabela B.23: MP3 com DCF sobre config. H e 10% de acessos à MC

MP3 SEM CLOCK GATING & DCF active
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	1.191	1.191	27.302	5.027.895	575.462	10,27
8	8.487	1.152.315	54.435	10.045.992	1.156.559	10,32
16	48.862	2.299.190	108.484	20.044.014	2.349.339	10,49
32	300.266	4.567.275	214.627	39.770.824	4.954.036	11,08
64	5.289.216	8.386.829	393.521	72.713.280	16.597.544	18,58
128	66.477.967	8.753.099	411.465	75.804.368	101.333.160	57,21
256	194.431.315	8.765.176	411.227	75.836.120	277.143.136	78,52

Tabela B.24: MP3 com DCF sobre config. H e 50% de acessos à MC

MP3 + CLOCK GATING & DCF active
Configuration H + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	35.107	572.445	135.884	4.995.222	800.969	13,82
8	342.093	1.101.599	262.322	9.613.774	1.894.495	16,46
16	3.856.811	1.736.324	413.743	15.153.025	7.590.758	33,38
32	19.620.790	19.620.790	421.612	15.454.397	29.436.650	65,57
64	51.634.509	1.770.964	421.710	15.439.708	73.462.040	82,63
128	115.629.658	1.774.472	421.600	15.452.453	161.250.896	91,26
256	243.621.777	1.771.807	421.084	15.453.747	339.342.912	95,64

Tabela B.25: Javac sobre config. P e 10% de acessos à MC

JAVAC COM CLOCK GATING
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy'	Saved Energy	Saved Energy (%)
4	94	655.229	13.064	19.877.970	29	0,00
8	822	1.311.445	26.180	39.765.416	2.106	0,01
16	4.360	2.622.440	52.611	79.579.680	14.817	0,02
32	20.472	5.247.762	104.778	159.006.112	80.730	0,05
64	108.025	10.477.282	209.535	317.376.800	527.853	0,17
128	861.293	20.836.387	415.265	629.324.352	5.336.162	0,84
256	69.251.853	30.259.410	606.559	905.859.136	359.406.624	28,41

Tabela B.26: Javac sobre config. P e 50% de acessos à MC

JAVAC COM CLOCK GATING
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.667	652.773	64.656	20.292.274	1.030	0,01
8	22.303	1.302.061	127.456	40.408.732	71.466	0,18
16	149.239	2.594.792	253.821	79.903.480	669.655	0,83
32	1.421.475	4.974.361	488.376	151.778.496	7.928.268	4,96
64	25.512.334	6.229.464	610.008	188.258.688	128.916.736	40,65
128	89.533.113	6.230.752	609.340	188.160.592	446.321.568	70,34
256	217.591.662	6.222.807	610.398	187.840.880	1.073.648.512	85,11

Tabela B.27: Javac sobre config. M e 10% de acessos à MC

JAVAC COM CLOCK GATING
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	106	325.667	6.621	2.316.179	6	0,00
8	962	649.618	12.955	4.630.256	312	0,01
16	4.777	1.296.786	26.133	9.259.112	1.849	0,02
32	22.863	2.593.360	52.199	18.502.718	9.783	0,05
64	116.342	5.179.494	103.602	36.936.536	55.222	0,15
128	940.029	10.282.122	205.956	73.223.888	500.319	0,68
256	71.286.164	14.785.264	295.215	104.335.336	28.868.560	21,67

Tabela B.28: Javac sobre config. M e 50% de acessos à MC

JAVAC COM CLOCK GATING
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.884	324.652	31.868	2.333.421	186	0,01
8	24.223	646.782	63.553	4.644.524	8.155	0,18
16	151.754	1.283.351	125.770	9.190.117	63.513	0,69
32	1.412.498	2.459.033	241.085	17.487.920	666.925	3,67
64	25.841.071	3.045.168	298.799	21.512.270	10.226.540	32,22
128	89.977.401	3.036.443	298.094	21.434.452	34.761.132	61,86
256	217.871.935	3.044.207	298.304	21.492.952	83.692.776	79,57

Tabela B.29: Javac sobre config. H e 10% de acessos à MC

JAVAC COM CLOCK GATING
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	136	444.577	8.825	6.172.104	71	0,00
8	933	886.252	17.855	12.335.955	784	0,01
16	4.433	1.776.067	35.368	24.659.480	4.048	0,02
32	21.396	3.544.919	70.975	49.285.376	21.797	0,04
64	111.755	7.090.488	141.847	98.446.720	131.912	0,13
128	922.411	14.076.847	282.252	195.603.696	1.332.487	0,68
256	71.933.327	20.246.680	405.297	280.001.152	102.042.744	26,71

Tabela B.30: Javac sobre config. H e 50% de acessos à MC

JAVAC COM CLOCK GATING
Configuration H + Mixed-D Organization (50% of the memory accesses related to SM)

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	3.328	441.291	43.493	6.220.287	1.788	0,03
8	24.450	887.904	87.073	12.414.507	22.449	0,18
16	151.547	1.755.937	171.996	24.637.846	173.033	0,70
32	1.467.004	3.370.523	329.117	47.161.592	2.058.028	4,18
64	26.114.179	4.167.810	407.456	58.071.868	36.276.004	38,45
128	90.339.407	4.149.909	406.971	57.709.172	124.466.664	68,32
256	218.099.530	4.163.597	407.424	58.045.956	301.681.216	83,86

Tabela B.31: Javac com DCF sobre config. P e 10% de acessos à MC

JAVAC COM CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	109	655.735	13.066	18.795.158	1.024.592	5,17
8	816	1.311.499	26.545	37.576.560	2.054.957	5,19
16	4.273	2.625.509	52.553	75.165.808	4.108.748	5,18
32	20.574	5.244.804	104.868	150.295.056	8.274.377	5,22
64	107.645	10.489.104	210.031	300.228.768	16.868.298	5,32
128	877.793	20.846.153	417.042	596.184.448	37.914.908	5,98
256	68.665.612	30.339.351	606.696	862.504.128	403.849.056	31,89

Tabela B.32: Javac com DCF sobre config. P e 50% de acessos à MC

JAVAC COM CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.769	658.054	64.901	18.794.398	1.112.166	5,59
8	22.384	1.306.403	127.234	37.534.492	2.258.934	5,68
16	151.261	2.598.136	255.279	74.515.736	5.041.768	6,34
32	1.408.648	4.974.291	487.094	142.712.800	16.185.787	10,19
64	25.493.599	6.222.857	6.222.857	177.593.728	138.066.240	43,74
128	89.567.865	6.230.147	609.930	177.517.232	455.599.424	71,96
256	217.453.162	6.238.884	611.129	177.975.072	1.086.786.176	85,93

Tabela B.33: Javac com DCF sobre config. M e 10% de acessos à MC

JAVAC COM CLOCK GATING & DCF active
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	106	325.502	6.477	2.272.658	39.419	1,70
8	1.009	648.681	13.010	4.543.903	79.216	1,71
16	4.810	1.295.928	25.955	9.087.644	159.396	1,72
32	22.030	2.592.739	51.929	18.158.772	325.861	1,76
64	118.014	5.179.590	104.103	36.272.284	683.461	1,85
128	914.592	10.285.933	205.269	71.962.856	1.736.438	2,36
256	71.720.529	14.770.099	295.347	102.323.560	30.791.174	23,13

Tabela B.34: Javac com DCF sobre config. M e 50% de acessos à MC

JAVAC COM CLOCK GATING &DCF
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	2.984	323.703	31.607	2.272.059	42.764	1,85
8	23.674	646.967	63.361	4.532.951	92.962	2,01
16	151.928	1.282.135	125.647	8.981.582	231.758	2,52
32	1.414.601	2.461.704	240.637	17.149.052	985.704	5,44
64	25.834.233	3.045.673	298.534	21.103.416	10.626.968	33,49
128	89.814.968	3.048.775	298.235	21.118.110	35.103.080	62,44
256	217.965.934	3.036.072	298.002	21.046.654	84.119.424	79,99

Tabela B.35: Javac com DCF sobre config. H e 10% de acessos à MC

JAVAC COM CLOCK GATING & DCF active
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	111	440.016	8.697	5.992.293	167.284	2,72
8	846	886.189	17.422	11.994.718	337.527	2,74
16	4.540	1.771.814	35.673	23.979.530	678.298	2,75
32	21.122	3.548.721	71.138	47.949.464	1.375.112	2,79
64	111.466	7.077.672	141.361	95.746.256	2.828.303	2,87
128	929.033	14.087.878	281.891	190.262.256	6.692.260	3,40
256	72.597.620	20.201.322	404.957	271.740.160	110.821.496	28,97

Tabela B.36: Javac com DCF sobre config. H e 50% de acessos à MC

JAVAC COM CLOCK GATING & DCF active
Configuration H + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	3.272	441.841	43.495	5.991.881	230.320	3,70
8	24.315	882.552	86.491	11.957.136	475.338	3,82
16	155.193	1.755.517	171.610	23.747.268	1.079.148	4,35
32	1.477.476	3.372.432	330.325	45.509.968	3.807.848	7,72
64	26.065.522	4.161.788	407.316	56.064.116	38.601.904	40,78
128	90.109.350	4.167.111	407.159	56.077.680	126.660.504	69,31
256	218.167.145	4.158.083	406.977	55.951.532	302.344.416	84,38

Tabela B.37: Compress sobre config. P e 10% de acessos à MC

COMPRESS COM CLOCK GATING
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	317	931.626	23.261	19.278.156	97	0,00
8	2.668	1.864.977	46.818	38.532.788	6.892	0,02
16	14.638	3.728.151	93.306	76.984.376	53.683	0,07
32	74.165	7.422.870	185.771	153.524.080	324.693	0,21
64	537.264	14.749.430	369.074	303.840.416	3.086.602	1,01
128	22.352.905	24.254.029	606.993	493.667.136	121.725.064	19,78
256	150.196.047	24.297.120	606.470	494.300.640	749.340.736	60,25

Tabela B.38: Compress sobre config. P e 50% de acessos à MC

COMPRESS COM CLOCK GATING
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	8.880	928.685	113.466	19.966.856	4.051	0,02
8	82.387	1.842.627	224.297	39.336.180	279.740	0,71
16	687.432	3.549.093	433.338	74.918.304	3.300.164	4,22
32	10.060.475	5.052.463	616.537	105.741.744	49.802.836	32,02
64	42.033.340	5.058.578	616.595	105.746.464	206.502.448	66,13
128	106.020.802	5.060.989	617.694	105.883.736	524.652.352	83,21
256	234.042.373	5.063.923	616.542	105.853.312	1.151.792.512	91,58

Tabela B.39: Compress sobre config. M e 10% de acessos à MC

COMPRESS COM CLOCK GATING
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	223	369.240	9.253	2.258.983	14	0,00
8	1.731	738.507	18.594	4.513.401	524	0,01
16	9.276	1.476.754	36.963	9.024.198	3.596	0,04
32	50.645	2.950.946	73.943	18.025.650	22.282	0,12
64	291.844	5.875.747	147.381	35.859.424	142.323	0,40
128	5.802.775	11.177.725	278.960	67.684.448	2.965.332	4,20
256	125.801.048	11.859.672	296.330	71.526.920	49.343.972	40,82

Tabela B.40: Compress sobre config. M e 50% de acessos à MC

COMPRESS COM CLOCK GATING
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	5.765	369.634	45.100	2.281.362	378	0,02
8	51.188	733.317	89.869	4.522.487	17.007	0,37
16	360.596	1.438.229	175.506	8.823.596	152.200	1,70
32	5.477.518	2.414.318	294.299	14.633.625	2.295.423	13,56
64	36.988.522	2.461.143	299.767	14.922.276	14.339.718	49,00
128	100.982.330	2.460.547	299.972	14.900.901	38.833.636	72,27
256	228.893.377	2.467.727	300.028	14.946.737	87.769.688	85,45

Tabela B.41: Compress sobre config. H e 10% de acessos à MC

COMPRESS COM CLOCK GATING
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	342	572.444	14.179	5.978.954	174	0,00
8	2.315	1.144.701	28.575	11.953.179	2.119	0,02
16	12.252	2.286.411	56.986	23.888.798	12.462	0,05
32	61.057	4.566.781	114.274	47.696.156	71.119	0,15
64	388.256	9.086.757	227.151	94.894.176	542.502	0,57
128	13.397.814	16.219.637	405.476	168.341.216	20.890.702	11,04
256	139.988.319	16.412.282	409.120	170.248.720	195.996.480	53,52

Tabela B.42: Compress sobre config. H e 50% de acessos à MC

COMPRESS COM CLOCK GATING
Configuration H + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	8.224	571.279	69.320	6.070.813	4.481	0,07
8	68.017	1.132.066	138.018	12.049.280	68.659	0,57
16	517.310	2.204.836	269.286	23.391.128	681.574	2,83
32	8.094.977	3.380.384	412.228	35.602.008	11.496.916	24,41
64	40.076.015	3.384.842	413.301	35.668.724	55.586.600	60,91
128	104.119.028	3.376.364	412.781	35.568.652	143.702.240	80,16
256	232.091.171	3.382.681	413.000	35.620.252	320.500.608	90,00

Tabela B.43: Compress com DCF sobre config. P e 10% de acessos à MC

COMPRESS COM CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	340	933.880	23.343	17.328.440	1.811.225	9,46
8	2.680	1.859.591	46.443	34.671.444	3.620.845	9,46
16	14.067	3.724.236	92.975	69.291.432	7.264.103	9,49
32	75.362	7.432.317	186.272	138.299.744	14.787.181	9,66
64	529.350	14.751.019	370.007	274.645.952	31.561.966	10,31
128	22.349.473	24.265.527	606.595	447.826.496	167.694.416	27,24
256	150.274.936	24.271.444	606.315	448.070.016	796.400.768	64,00

Tabela B.44: Compress com DCF sobre config. P e 50% de acessos à MC

COMPRESS COM CLOCK GATING & DCF active
Configuration P + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	8.795	928.452	113.014	17.321.934	1.957.293	10,15
8	81.720	1.843.731	224.570	34.399.072	4.138.635	10,74
16	680.864	3.549.675	432.212	66.377.160	10.686.533	13,87
32	10.099.921	5.044.942	616.172	94.527.840	60.473.316	39,01
64	42.085.515	5.049.764	616.563	94.551.216	218.442.256	69,79
128	106.035.655	5.065.057	617.135	94.819.064	532.517.248	84,89
256	234.054.433	5.058.234	617.469	94.659.544	1.162.538.624	92,47

Tabela B.45: Compress com DCF sobre config. M e 10% de acessos à MC

COMPRESS COM CLOCK GATING & DCF active
Configuration M + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	227	369.342	9.455	2.197.730	56.062	2,49
8	1.751	738.691	18.354	4.392.973	112.681	2,50
16	9.498	1.475.532	36.792	8.783.660	227.133	2,52
32	48.458	2.951.287	73.283	17.549.790	467.178	2,59
64	300.253	5.876.096	147.575	34.930.060	1.036.393	2,88
128	5.864.262	11.170.165	280.157	65.950.456	4.678.404	6,62
256	126.358.618	11.811.946	295.479	69.485.352	51.294.208	42,47

Tabela B.46: Compress com DCF sobre config. M e 50% de acessos à MC

COMPRESS COM CLOCK GATING & DCF active
Configuration M + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	5.504	5.504	45.119	2.195.165	61.212	2,71
8	49.773	734.109	89.248	4.371.248	135.790	3,01
16	358.602	1.438.427	174.975	8.546.780	387.893	4,34
32	5.504.728	2.412.674	294.742	14.230.351	2.683.475	15,87
64	37.056.888	2.454.802	300.027	14.476.950	14.758.750	50,48
128	101.018.663	2.457.698	300.153	14.490.319	39.233.404	73,03
256	228.941.844	2.464.445	300.253	14.524.266	88.180.376	85,86

Tabela B.47: Compress com DCF sobre config. H e 10% de acessos à MC

COMPRESS COM CLOCK GATING &DCF
Configuration H + Mixed-D Organization and 10% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	303	571.241	14.248	5.696.197	285.733	4,78
8	2.250	1.144.652	28.605	11.370.743	575.678	4,82
16	11.891	2.286.996	57.287	22.736.774	1.160.666	4,86
32	60.258	4.566.132	114.107	45.426.368	2.359.985	4,94
64	396.048	9.083.399	227.938	90.376.656	5.108.527	5,35
128	13.470.408	16.220.274	406.387	160.493.616	28.873.496	15,25
256	140.371.746	16.363.897	408.907	161.734.448	203.818.912	55,76

Tabela B.48: Compress com DCF sobre config. H e 50% de acessos à MC

COMPRESS COM CLOCK GATING &DCF
Configuration H + Mixed-D Organization and 50% of the memory accesses related to SM

Cores	Contentions	Instructions	SM Instrucions	Spent Energy	Saved Energy	Saved Energy (%)
4	8.366	570.334	69.573	5.683.894	393.571	6,48
8	68.414	1.134.762	138.776	11.272.739	849.169	7,01
16	513.708	2.203.194	269.097	21.947.856	2.173.487	9,01
32	8.096.734	3.377.547	411.603	33.457.254	13.784.763	29,18
64	40.074.747	3.382.333	413.093	33.499.930	58.173.104	63,46
128	104.000.707	3.390.874	413.072	33.592.912	145.231.088	81,21
256	232.066.660	3.384.132	412.959	33.498.152	322.945.376	90,60

APÊNDICE C, DIAGRAMAS ARQUITETURAIS DOS COMPONENTES PROPOSTOS

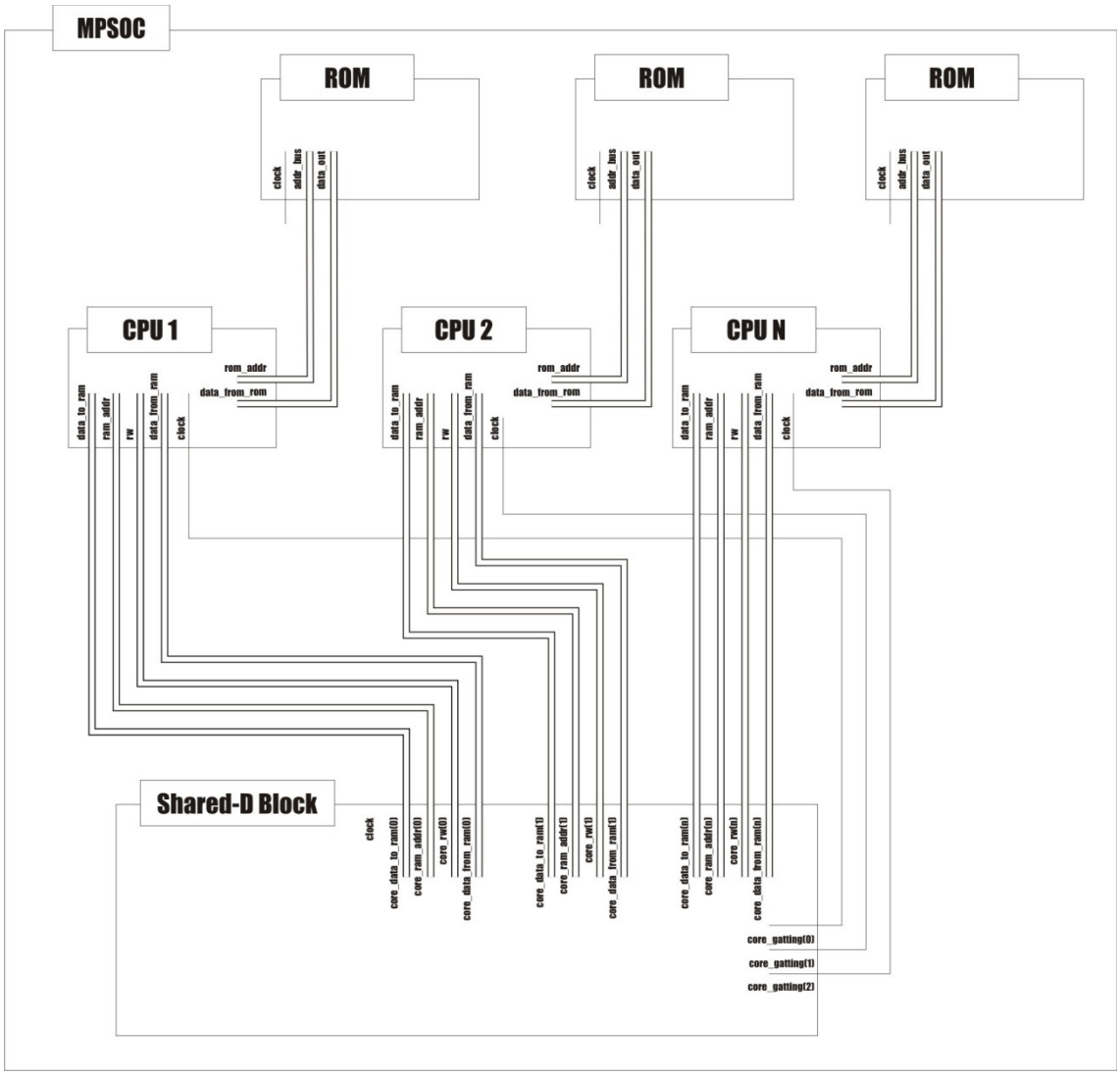


Figura C.4: Diagrama arquitetural de um MPSoC com a organização Shared-D

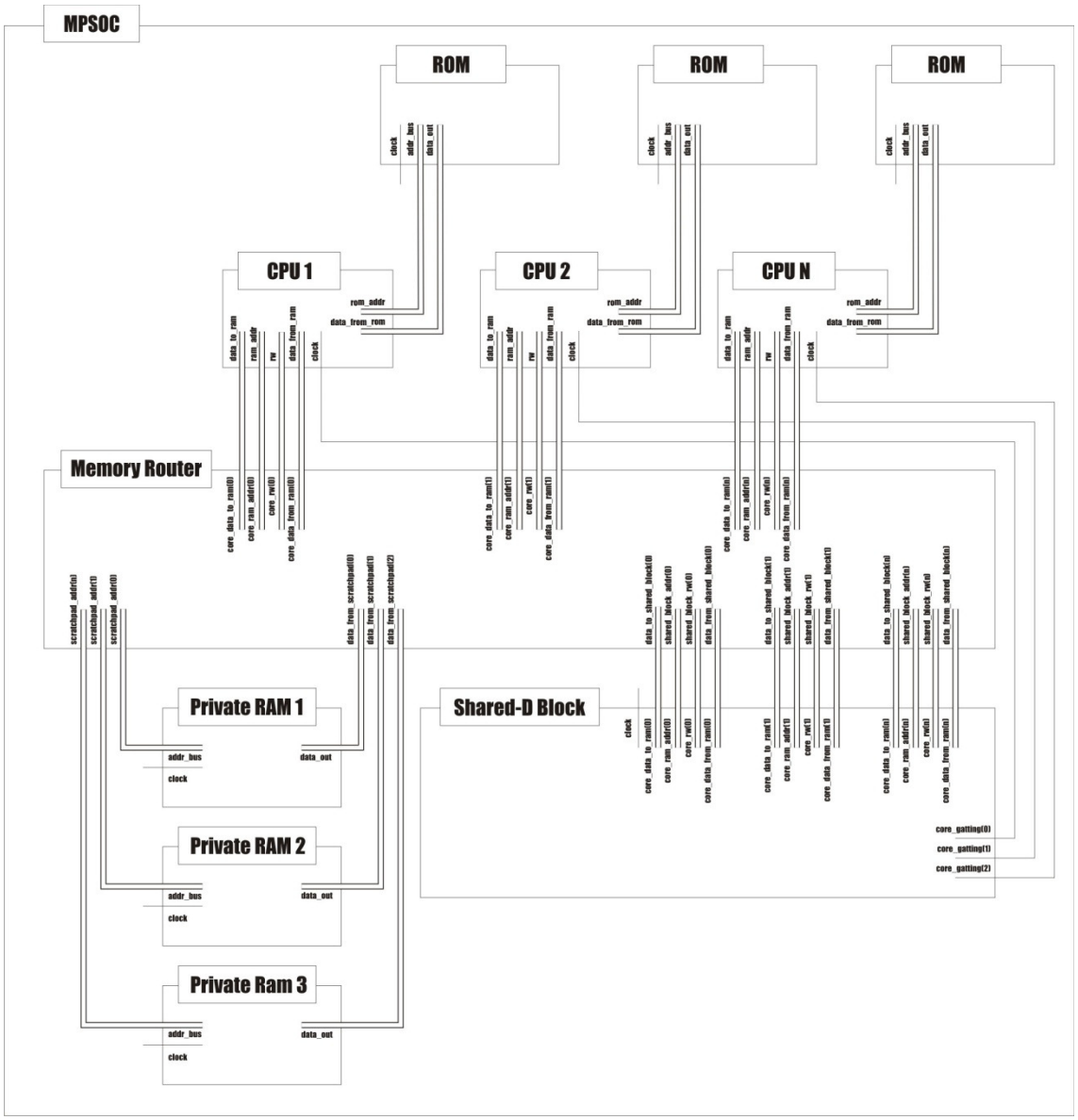


Figura C.4: Diagrama arquitetural de um MPSoC com a organização Mixed-D

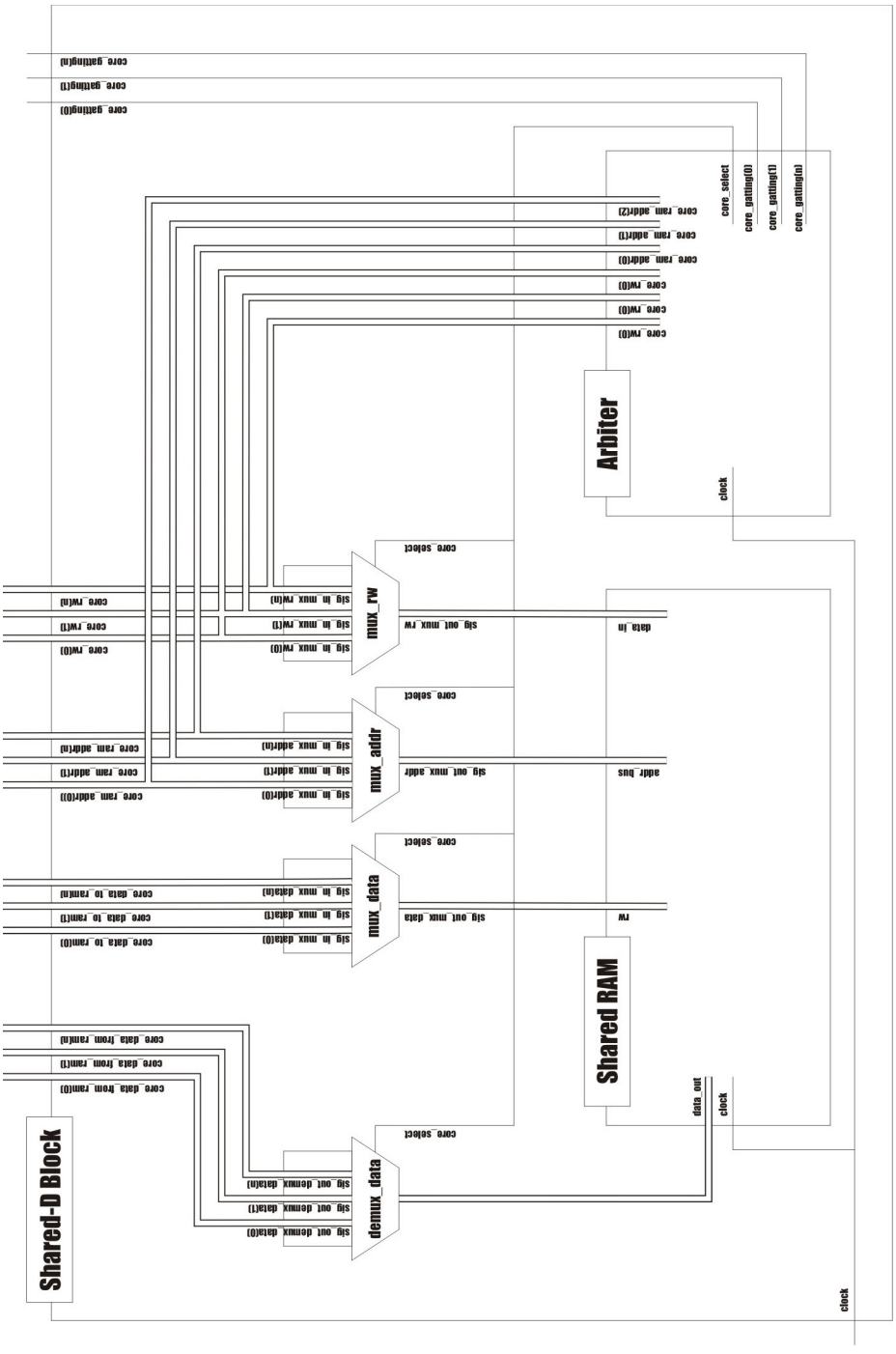


Figura C.1: Diagrama arquitetural do componente Shared-D Block

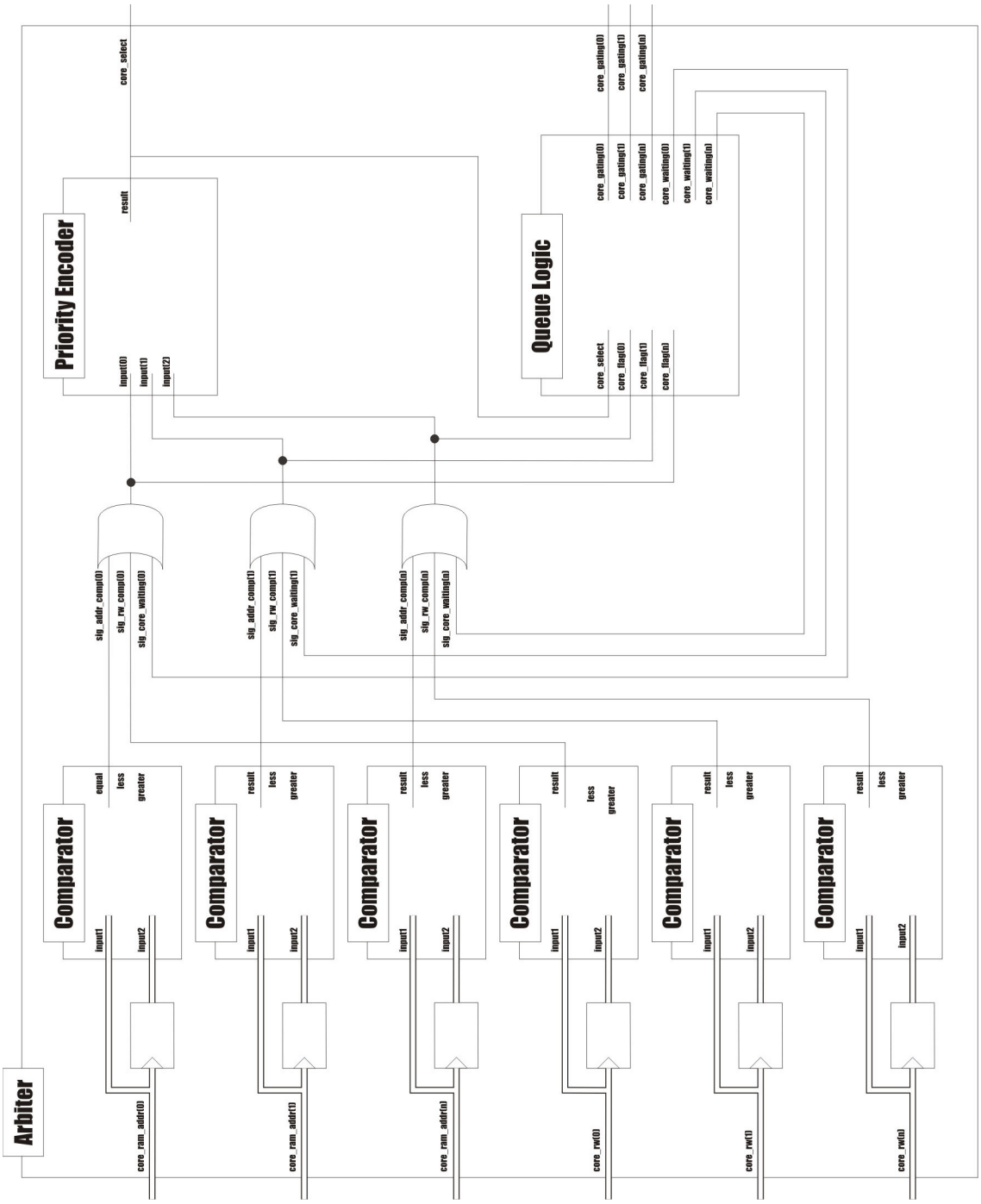


Figura C.2: Diagrama arquitetural do componente Arbiter

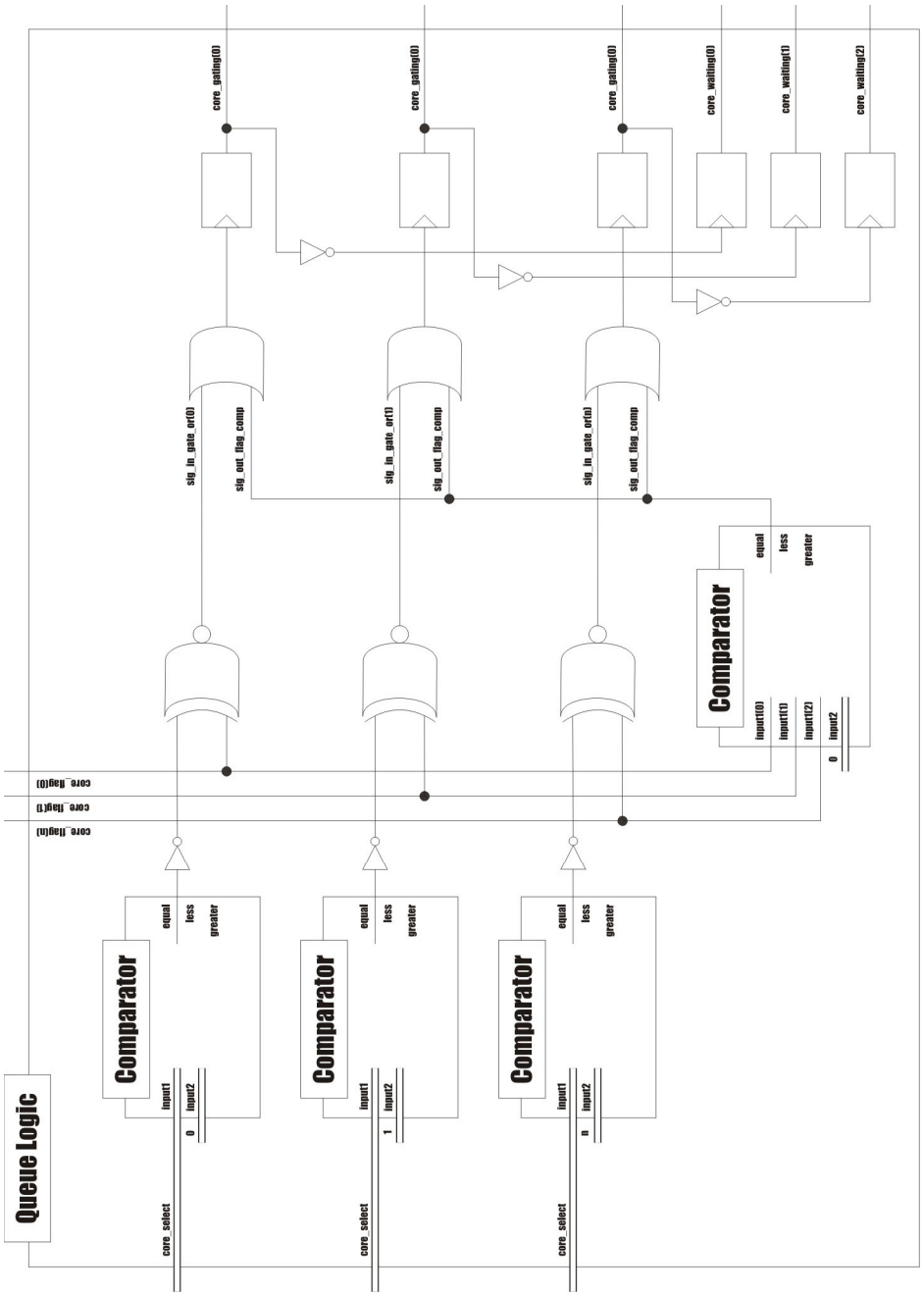


Figura C.3: Diagrama arquitetural do componente Queue Logic

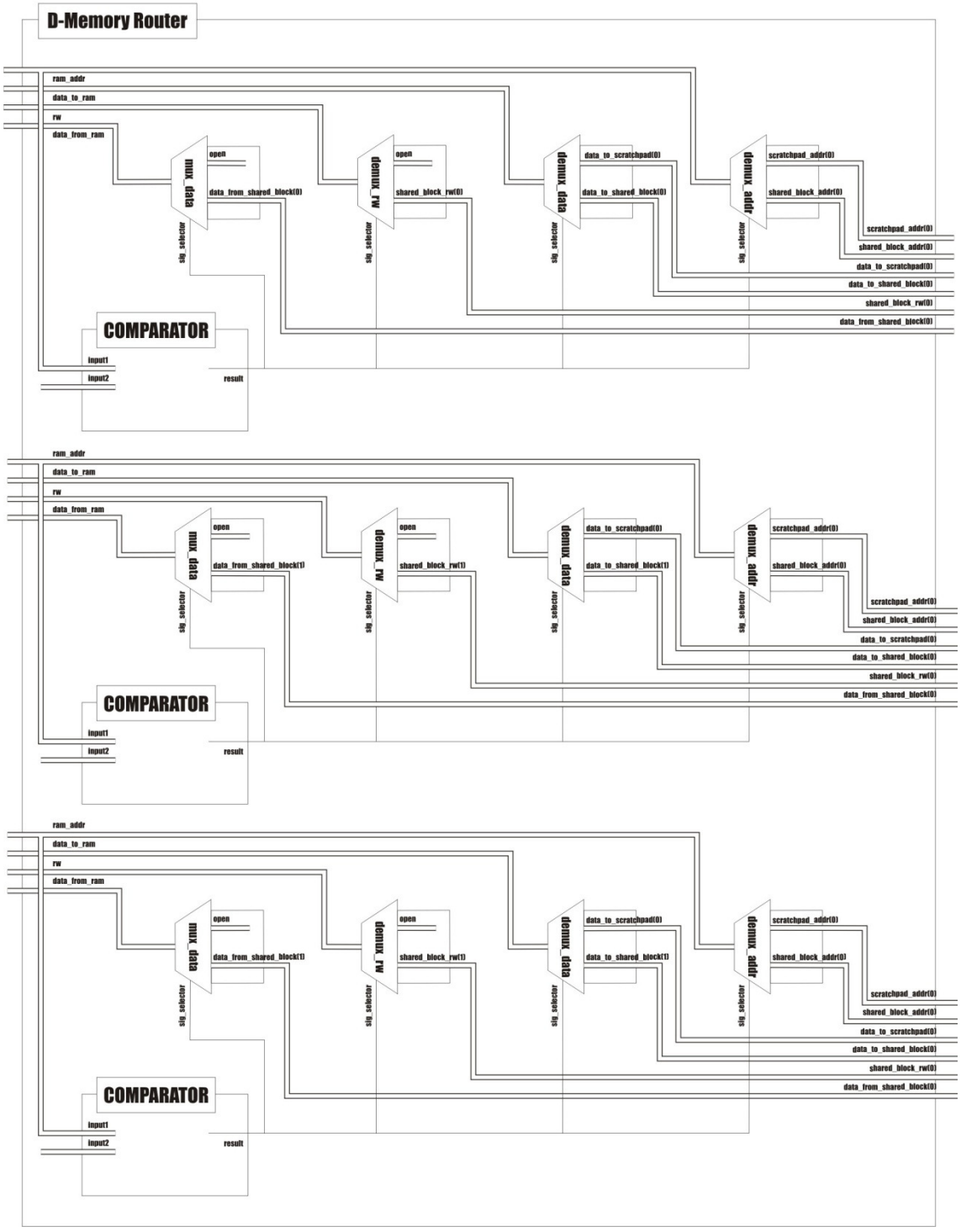


Figura C.4: Diagrama arquitetural do componente D-Memory Router