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

ANTONIO CARLOS SCHNEIDER BECK FILHO

Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, junho de 2004

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Beck Filho, Antonio Carlos Schneider

Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java / por Antonio Carlos Schneider Beck Filho. – Porto Alegre: PPGC da UFRGS, 2004.

130 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2004. Orientador: Carro, Luigi.

1. Microeletrônica. 2. Arquiteturas. 3. Java. 4. Consumo de Potência. 5. VLIW. I. Carro, Luigi. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Aos meus pais e à Sabrina.
Sempre ao meu lado.

AGRADECIMENTOS

Para a Sabrina. Apoio incondicional. Muitas vezes trocada pelo computador, (quase sempre) sem reclamar.

Aos pais e todo o resto da família. A distância não separa, une.

A todos os camagadas e colegas durante o mestrado. Que me apoiaram quando cheguei à desconhecida Porto Alegre. Em especial ao Émerson, Fernando, Gervini, Julius, Leomar, Márcio, Marcos.

Também aos professores André, Bampi, Flávio, Lisboa, Luba e Susin.

Finalmente, ao Luigi. Que foi louco suficiente em apostar em um cara de Santa Maria que ele não conhecia. Espero ter conseguido retribuir à ótima orientação.

Também, para os funcionários, colaboradores, Instituto de Informática, Universidade Federal do Rio Grande do Sul e CAPES. Ensino gratuito, com apoio e de qualidade.

Obrigado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
ASIP	Application Specific Instruction set Processor
CAD	Computer Aided Design
CI	Circuito integrado
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CMP	Chip MultiProcessing
DSP	Digital Signal Processing
E/S	Entrada/Saída
EDA	Eletronic Design Automation
GCC	GNU Compiler Collection
HDTV	High Definition Television
IMDCT	Inverse Modified Discrete Cosine Transform
IP	Intelectual Property
IPC	Instruções Por Ciclo
ISA	Instruction Set Architecture
J2ME	Java 2 Plataform, Micro Edition
JIT	Just-In-Time
JVM	Java Virtual Machine
MIDP	Mobile Information Device Profile
PDA	Personal Digital Assistent
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTOS	Real Time Operating System
SIMD	Single-Instruction, Multiple-Data
SMT	Simultaneous Multithreading
SOC	System on a Chip
TLB	Translation Lookaside Buffer
ULA	Unidade Lógica e Aritmética
UML	Unified Modelling Language
UFRGS	Universidade Federal do Rio Grande do Sul
VLIW	Very Long Instruction Word

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A contribuição deste trabalho foi orientada principalmente ao desenvolvimento de alternativas de *hardware* para a execução nativa de *bytecodes* Java em sistemas embarcados que naturalmente possuem restrições quanto à potência consumida, ao desempenho e à área ocupada. Primeiramente, o desenvolvimento do *Femtojava Low-Power* demonstra que a utilização de um *pipeline* e de um banco de registradores interno em arquiteturas de pilha resultam em uma redução significativa no consumo de potência. Após, a técnica de *folding*, que basicamente transforma várias operações de pilha em uma operação tipo RISC, é avaliada. A análise de uma segunda solução arquitetural, baseada em VLIW (*Very Long Instruction Word*), também traz resultados satisfatórios na redução do consumo de potência, sendo que a paralelização do código, feita por um analisador desenvolvido, é facilitada devido à utilização de uma arquitetura de pilha. O desempenho e a potência consumida de todas as arquiteturas propostas neste trabalho foram validadas utilizando-se o simulador CACO-PS, também desenvolvido no contexto desta dissertação. Os estudos de caso adotados para a validação das alternativas arquiteturais compreenderam algoritmos matemáticos, de ordenação, busca e processamento de sinais, bastante utilizados no domínio de sistemas embarcados. Resultados promissores principalmente em termos de energia consumida são alcançados, assim como na disponibilização de diferentes arquiteturas para a execução nativa de Java, principal proposta deste trabalho.

Palavras-chave: Microeletrônica, Java, potência, energia, folding, VLIW, sistemas embarcados, simulador

Using the VLIW Technique to Increase Performance and to Reduce Power Comsumption in Embedded Systems Based on Java

ABSTRACT

The main contribution of this work was the development of hardware alternatives for native execution of Java bytecodes for embedded systems that have power, performance and area constraints. Firstly, the development of the Femtojava Low-Power shows that the use of a pipeline and an internal register bank in stack architectures brings a significant reduction in the power consumption. After that, the *folding* technique, that basically changes a set of stack operations into a simple RISC one, is evaluated. Then, the analysis of a second architectural solution, based on VLIW (*Very Long Instruction Word*), demonstrates also good results concerning power consumption. Moreover, it is shown that the parallelization of the code is facilitated due to the specific stack architecture. The power consumption and performance of all architectures here proposed were evaluated using the CACO-PS simulator, which was also developed in this work. The case studies adopted for the validation of the architectures were mathematic, sort, search and DSP algorithms, widely used in the embedded system domain. Promising results mainly in energy consumption were achieved, as well as the disponibilization of different architectures for native execution of Java, the main objective of this work.

Keywords: Java, power, energy, folding, VLIW, embedded systems, simulator

1 INTRODUÇÃO

Como se pode observar cotidianamente, devido à evolução tecnológica, usam-se cada vez mais sistemas computacionais para facilitar tarefas do dia-a-dia na sociedade. Eles são utilizados para diferentes propósitos, como entretenimento, comunicação, controle de eletrodomésticos e veículos, entre vários outros. Classificados como sistemas computacionais embarcados, têm como exemplos práticos de utilização em: fornos de microondas, videogames, impressoras, *mp3 players*, câmeras fotográficas digitais e telefones celulares. Processadores e sistemas integrados em silício (SoC) dedicados a este propósito estão em franca expansão (TAKAHASHI , 2001).

Neste mesmo caminho, enquanto a tecnologia permite acrescentar mais e mais transistores dentro de um CI (Circuito Integrado), pesquisadores direcionam seus esforços para achar a melhor maneira para a utilização desta grande capacidade computacional. Conforme o SIA (*Semiconductor Industry Association*) *Roadmap* (SAI, 2003), a quantidade de transistores que poderá ser integrados em uma pastilha de silício chegará, em um futuro não muito distante, a um bilhão.

Em computadores pessoais, por exemplo, este potencial aplicado em várias técnicas para o aumento de performance, tanto para atender ao usuário comum com aplicações de escritório e entretenimento, quanto às grandes companhias, que utilizam este poder computacional para executar processos complexos, como previsão de tempo, simulações de partículas subatômicas ou de tráfego de veículos.

Todavia, em sistemas embarcados, o correto não é apenas ter o tempo de computação como principal métrica. Principalmente em sistemas embarcados portáteis, há uma grande preocupação com o consumo de energia, já que geralmente estes sistemas são dependentes de uma bateria, cuja duração não acompanha o aumento de desempenho dos processadores nem o aumento de complexidade dos algoritmos, como pode ser observado na Figura 1.1. Além disto, aspectos como tempo de projeto e custo de produção têm enorme impacto na eletrônica de consumo.

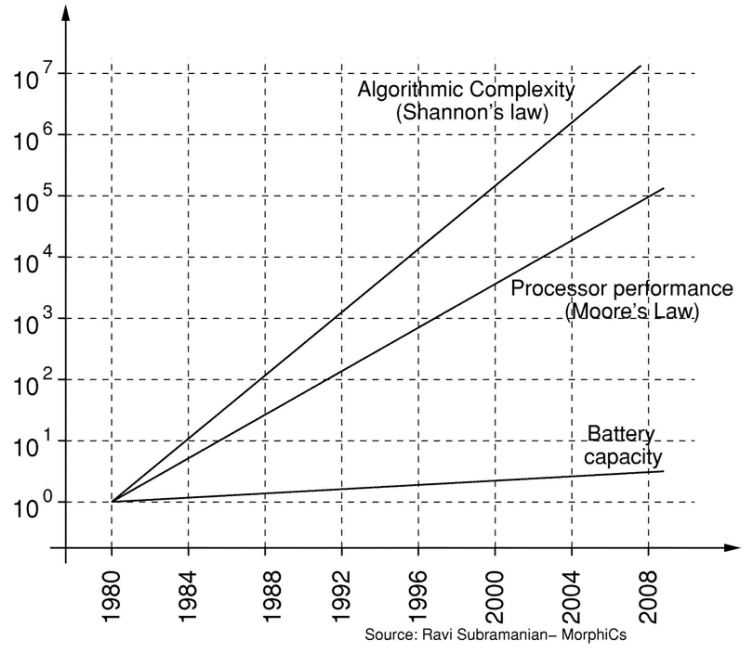

Figura 1.1 : Comparação entre a capacidade da bateria, performance dos processadores e complexidade dos algoritmos (RABAEY, 2000)

1.1 Sistemas Embarcados e SoC

Para aplicações embarcadas, a tendência é fazer uso da tecnologia não para alcançar a máxima performance possível, mas sim, a performance necessária para atender somente aos requisitos da aplicação, economizando-se em potência e em área. Somando-se a esse fato, observa-se que os sistemas embarcados estão agregando cada vez mais funções, oferecendo ao usuário diversos recursos antes inexistentes. Por exemplo, telefones celulares de última geração reúnem funções como acesso à Internet, visor colorido, reprodução de áudio e vídeo, conexão de dados via infravermelho, videogame, entre outros (NOKIA, 2004).

Em paralelo a isto, empresas estão cada vez mais preocupadas em reduzir o tempo do ciclo de projeto de seus produtos. Em contrapartida, elas precisam melhorar a qualidade destes: seja agregando novas funções, melhorando a interface com o usuário, ou ainda aumentando a vida útil da bateria. Esta meta fica quase inatingível quando considerada a complexidade de projeto que atualmente estes sistemas têm. Devido a isto, várias metodologias têm sido propostas, como em (EDWARDS et al., 1997) e (SANGIOVANNI-VINCENTELLI; MARTIN, 2001), assim como arquiteturas para sistemas embarcados (JACOME; VENCIANA, 2000).

Os SoC (*System on a Chip*) também têm sido um alvo constante de pesquisas, tanto em metodologias de projeto quanto arquiteturas (LYONNARD et al., 2001). O consumo de potência também é largamente pesquisado pelo fato de ser um dos principais problemas relacionados aos SoC em embarcados (PIGUET; RENAUNDUIM; OMMÈS, 2001). Assim, otimizações arquiteturais (PETROV; ORAILOGLU, 2001) e otimizações em memória (TANG; GUPTA; NICOLAU, 2002) foram propostas, visando redução do consumo.

Esta crescente complexidade exige que os sistemas sejam projetados em um alto nível de abstração. Em algum momento do projeto, esta especificação em alto nível deverá ser mapeada diretamente para o sistema final, que é um conjunto de *hardware* e *software*. Assim, é necessária a automatização do projeto, através do uso de ferramentas EDA (*Eletronic Design Automation*), para fazer com que esta descrição em um alto nível sejam automaticamente transformada em *hardware* e *software* reais.

Contudo, é necessário tomar decisões, mesmo em um alto nível de abstração, que terão impacto direto no desempenho, na potência e no custo do sistema final. No âmbito local (Grupo de Microeletrônica da UFRGS), diversos trabalhos têm sido realizados sobre sistemas embarcados e mais recentemente sobre SoC. O S3E2S (*Specification, Simulation and Synthesis of Embedded Electronic Systems*) (WAGNER et al., 1999) é um ambiente de especificação, simulação e síntese de sistemas embarcados baseado no paradigma orientado a objetos. O SASHIMI (*System AS Software and Hardware In MIcrocontrollers*) inclui uma metodologia e uma ferramenta para o projeto e geração de sistemas embarcados (ITO; CARRO; JACOBI, 2001). Além do mais, primitivas básicas de funcionamento (que são parametrizáveis conforme o requisito da aplicação) de um sistema operacional de tempo real para processadores Java estão sendo desenvolvidas (GERVINI et al. 2003).

1.2 Java em Sistemas Embarcados

É em todo este paradigma, da programação em um alto nível de abstração para reduzir o tempo do ciclo de projeto, do uso de ferramentas automatizadas, e de restrições de performance, área e potência, que a linguagem Java é inserida. A linguagem Java facilita a programação e a validação do sistema, já que é orientada a objetos. Assim, a modelagem do sistema através de alguma linguagem específica, como UML (RATIONAL, 2002), é facilitada. Além disto, Java tem a vantagem de ser multiplataforma: todo o sistema pode ser executado e validado previamente em uma plataforma de desenvolvimento, e depois facilmente portado para a uma ou mais plataformas alvo.

Somando-se a isso, a linguagem possui outras vantagens para ser aplicada em sistemas embarcados: facilita e agiliza o desenvolvimento do produto, pois já é uma linguagem amplamente difundida e utilizada; oferece segurança e pequeno tamanho ocupado de memória de instruções, já que Java foi originalmente desenvolvido para ser transmitido pela Internet e, analisando-se o número disponível de instruções e seus tipos, é de natureza CISC (*Complex Instruction Set Computer*). São por estas e outras razões que a linguagem Java está se tornando cada vez mais popular em ambientes embarcados, que por sua vez, estão em franca expansão (SCHLETT, 1998). É estimado que o número de dispositivos com Java embarcado como telefones celulares, PDA e *pagers* irá crescer de 176 milhões em 2001 para 721 milhões em 2005 (TAKAHASHI, 2001). Não obstante, é previsto que no mínimo 80% dos telefones celulares irão suportar Java em 2006 (LAWTON, 2002). Somando-se a isso, há previsões da industria que haverá dez vezes mais desenvolvedores para sistemas embarcados em relação a desenvolvedores de *software* de propósito geral no ano de 2010 (ATHERTON, 1998).

Contudo, a própria natureza da linguagem Java não é voltada para performance, e sim para a portabilidade. Então, otimizações tanto no código quanto na microarquitetura de processadores Java devem ser feitas levando em consideração também o quesito potência. Além disso, uma outra abordagem que poderia ser considerada é a comparação entre processadores para sistemas embarcados que executam diretamente o

código Java com processadores de propósito geral (como os processadores Motorola da série *DragonBall* ou Intel *XScale*, usados em PDA) que usam uma máquina virtual, como a J2ME (SUN, 2003), específica para o domínio de embarcados. Há ainda outras abordagens, como a de processadores Java trabalharem em conjunto com o processador nativo da máquina. Este co-processador Java adicional ficaria sempre desligado e, quando um programa Java fosse ser utilizado, ao invés de uma máquina virtual ser executada no processador nativo, o co-processador Java seria ligado, executando diretamente o código Java para o sistema (NAZOMI, 2003).

Conclui-se então que é necessário analisar cuidadosamente sistemas embarcados baseados em Java. Existem várias maneiras de um programa Java ser executado, como mostra a Figura 1.2. Uma delas é a execução nativa (diretamente em *hardware*) de *bytecodes* (a linguagem de montagem de Java). Como podem ser observadas nesta mesma figura, otimizações podem ser feitas no próprio código Java, antes de ser compilado, ou diretamente em seus *bytecodes*, para posterior execução (KAZI et al., 2000). A figura mostra que há diferentes possibilidades de execução de Java quando comparadas com a linguagem C, mostrando a grande flexibilidade dessa linguagem. Além disso, o consumo de energia destes processadores dedicados precisa ser considerado.

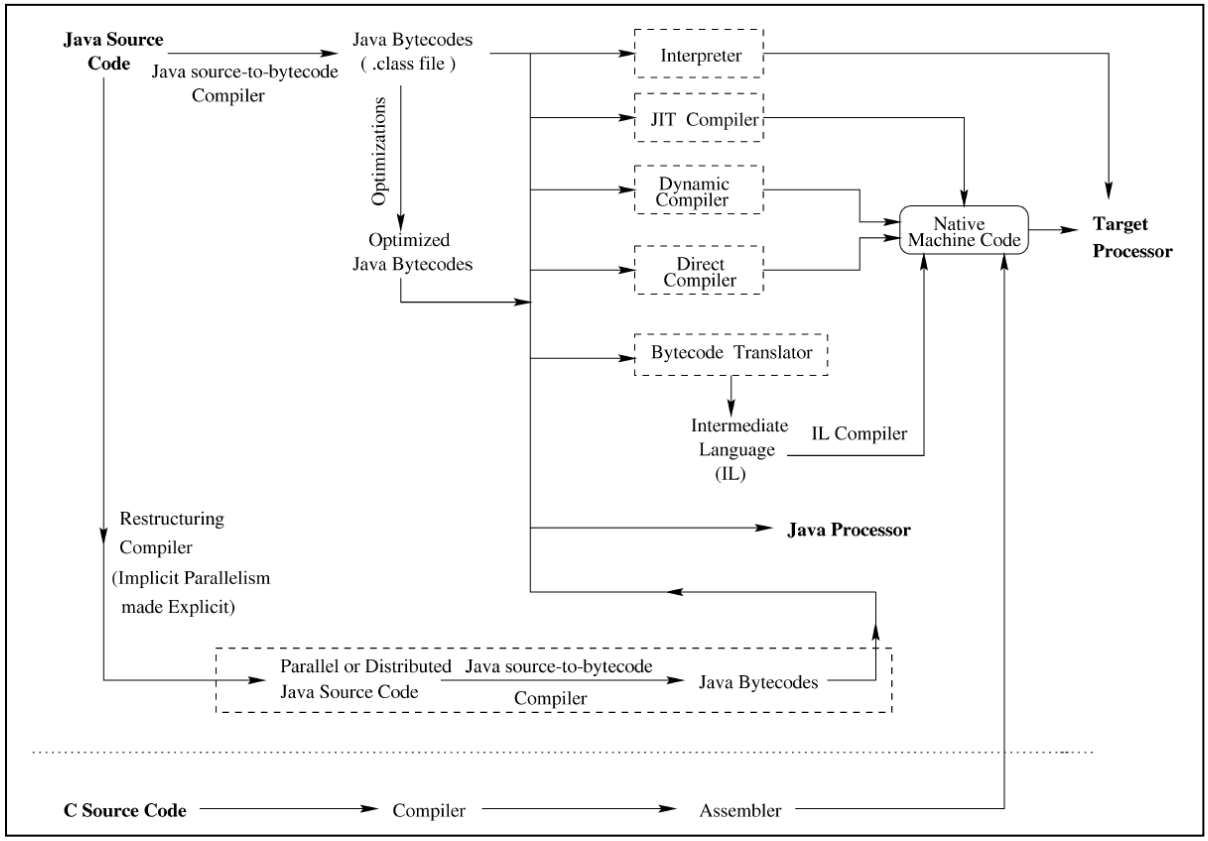

Figura 1.2 : Várias maneiras de se otimizar e executar *bytecodes* Java (KAZI et al., 2000)

1.3 Alternativas arquiteturais e otimizações no código para a execução nativa de Java

Os processadores Java hoje existentes geralmente limitam-se a executar o código nativo Java – ou *bytecodes* – de forma simples e sem se preocupar com a potência

dissipada. Além do mais, não aplicam técnicas para a redução no consumo de potência que já há muito tempo foram estudadas e validadas em processadores com outros modelos computacionais. Poucas alternativas foram estudadas, e o exemplo do processador MAJC da Sun Microsystems (TREMBLAY; CHAN; CHAUDHRY, 2000), que foi descontinuado, é uma delas. Partindo desta experiência, nota-se que apenas procurar o máximo de performance possível não é o suficiente. Também é preciso focar-se em outros critérios, como potência e área, oferecendo alternativas para que o usuário, através de ferramentas, possa escolher a melhor opção para aquela situação em específico. Isto só é possível através de duas vertentes: parametrização (onde recursos são oferecidos apenas quando necessário, economizando em potência e área consumidas) e o oferecimento de diferentes alternativas arquiteturais dos processadores.

As propostas *ad hoc* tradicionais utilizadas no desenvolvimento de sistemas embarcados não têm sido suficientes para tratar com o crescimento da complexidade das novas aplicações (restrições temporais, necessidades multimídia, etc.). O desenvolvimento de *software* embarcado não possui metodologias consolidadas devido ao fato de ser uma área de pesquisa em início de desenvolvimento (LEE, 2000), (SANGIOVANNI-VINCENTELLI; MARTIN, 2001) e (SHANDLE; MARTIN, 2002).

Somando-se a isto, nota-se que no ciclo normal de projeto de sistemas embarcados geralmente a escolha da plataforma se dá pelos motivos errados: a preferência por um processador baseia-se mais em sua disponibilidade ou na mão de obra disponível, por causa de restrições de tempo em conjunto com a falta de ferramentas para automatizar o processo; ao invés de escolher a melhor plataforma que se adapte aos requisitos da apliação.

Para lidar com estes fatores, estão em desenvolvimento no grupo algumas ferramentas, como uma que faz a estimativa de desempenho e potência de um determinado algoritmo em um alto nível de abstração. Isto é, analisando-se o *software* e a sua modelagem, pode-se prever o quanto irá ser gasto quando mapeado para um sistema real (MATTOS et al., 2004). Ainda, a geração de uma biblioteca de *software* está sendo criada, analisando os custos dos três principais eixos: potência, área e performance. Partindo de uma descrição do *software* em um alto nível de abstração, diferentes rotinas são escolhidas para compor o sistema final (REYNERI et al., 2001) (NANDI, 2001) (GIVARGIS; VAHID; HENKEL, 2001).

Assim, o principal objetivo deste trabalho foi aumentar o número de opções de *hardware* para sistemas embarcados baseados em Java, fazendo uso de várias técnicas arquiteturais conhecidas porém pouco exploradas em processadores de pilha, levando em consideração, além da área e performance, também o quesito potência. Unindo-se estas alternativas arquiteturais às ferramentas que estão sendo desenvolvidas pelo grupo, chega-se mais perto do objetivo final, o de criar um ambiente completo para facilitar a automatização na geração de sistemas embarcados a partir de um alto nível de abstração.

Teve-se como ponto de partida o primeiro processador Femtojava desenvolvido (ITO; CARRO; JACOBI, 2001), chamado neste trabalho de Femtojava Multiciclo para diferenciá-lo das novas versões. Durante o trabalho, novas arquiteturas para a execução nativa de *bytecodes* Java, com melhor desempenho e menor consumo de energia, foram projetadas. O objetivo final traçado, que é o desenvolvimento de uma versão VLIW (*Very Long Instruction Word*) (HENNESSY; PATTERSON, 2003) do Femtojava, foi composto de diversos passos intermediários. Para o desenvolvimento desta versão, foi necessário construir uma nova versão do Femtojava, dotada de um *pipeline*, visto que para a implementação de ambas as técnicas é necessário este recurso. Esta versão com

pipeline deu origem a versão Femtojava *Low-Power*, que além de possuir um *pipeline* de cinco estágios, implementa a pilha de operadores em um banco de registradores, ao invés de usar a memória principal para este fim.

Outra técnica específica para processadores de pilha, chamada *Folding* (O'CONNOR; TREMBLAY, 1997) (TON et al., 1997), também foi analisada e poderia ter dado origem a uma nova versão do processador Femtojava. Entretanto, através de simulações, percebeu-se que a implementação desta técnica não traria benefícios significativos em termos de performance e potência dissipada.

Todavia, a grande dificuldade de todo o ciclo de projeto deste trabalho seria a seguinte: como analisar as várias alternativas arquiteturais, levando o consumo de potência em consideração, sem ter que fazer o protótipo destas alternativas em *hardware*? Este problema faria com que ocorresse um grande atraso nas experimentações, tornando o objetivo final inviável considerando o tempo disponível para a sua concretização, ou ainda diminuindo bastante o conjunto de experimentações para chegar na melhor escolha possível. O motivo é que, para cada processador, além de especificado, ele teria que ser projetado e programado, em um nível de abstração relativamente baixo, em alguma linguagem de especificação de *hardware*, como VHDL, para posteriormente ser feito o seu protótipo em FPGA, e finalmente, ter o consumo de potência analisado.

Para resolver este problema, foi desenvolvido um simulador de potência, que permite que o sistema seja descrito em qualquer nível de abstração, fazendo com que o ciclo de projeto se tornasse mais rápido e, conseqüentemente, viável. O simulador construído chama-se CACO-PS (*Cycle-Accurate COnfigurable Power Simulator*), (BECK et al., 2003) que é um simulador de código compilado, que calcula a potência baseado na taxa de chaveamento dos componentes, além de ser um simulador ciclo-a-ciclo. Apesar de ser de código compilado, ele oferece a possibilidade da descrição estrutural de qualquer arquitetura (é de propósito geral).

A Figura 1.3 ilustra o ciclo de projeto deste trabalho. Partindo do Femtojava Multiciclo, já desenvolvido, foi projetado o Femtojava *Low-Power* (BECK; CARRO, 2003), dotado de um *pipeline*. A partir desta versão começou a ser desenvolvida uma nova versão VLIW (BECK; CARRO, 2004b). Intermediário a isto, foi analisada a possibilidade de implementação da técnica de *folding*. Todos os passos foram acompanhados através de simulação usando o simulador CACO-PS. Depois que os processadores foram especificados e validados, foram construídas suas versões finais em VHDL.

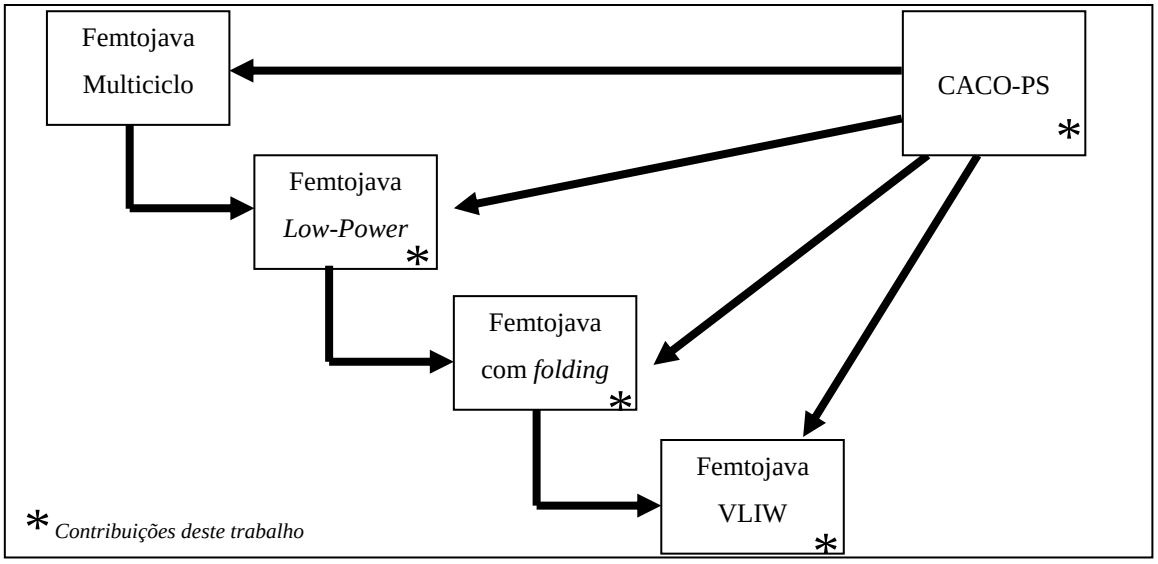

Figura 1.3 : Passos até chegar na última versão do Femtojava

Esta dissertação está organizada conforme a seguir:

O capítulo 2 mostra o estado da arte deste trabalho: estudam-se os processadores Java já existentes, seus funcionamentos, suas vantagens e desvantagens. Vários processadores Java já foram construídos. Alguns continuam no mercado e estão disponíveis comercialmente, outros são projetos acadêmicos e outros ainda já foram descontinuados. Entretanto, todos têm uma característica em comum: apesar de a maioria ser projetado para a utilização em sistemas embarcados, não possuem uma preocupação maior na aplicação de técnicas para a redução de potência dissipada nem na utilização da parametrização para a redução de área. Adicionalmente, processadores com arquitetura VLIW também são analisados.

O capítulo 3 apresenta o simulador CACO-PS. Primeiramente, demonstra-se o motivo pelo qual foi desenvolvido, depois, como ele foi projetado e construído, o seu funcionamento, como descrever uma arquitetura qualquer e fazer o cálculo de potência, e finalmente mostra-se um breve exemplo construído utilizando sua sintaxe e sua interface para execução. Além do mais, são apresentados todos os algoritmos programados em Java e utilizados para simulação em todas as arquiteturas desenvolvidas ao longo deste trabalho.

O capítulo 4 demonstra em detalhes o que é a técnica de *folding*. A princípio desenvolvida pela *Sun Microsystems* em seu processador picoJava-I, esta técnica transforma um conjunto de instruções baseadas no paradigma de máquinas de pilha em uma única instrução do tipo RISC (*Reduced Instruction Set Computer*). Vários estudos e otimizações desta técnica foram apresentados. Entretanto, mostra-se que através da análise de um conjunto de *benchmarks*, a técnica não traz benefícios reais em termos de performance. Além destes resultados, é mostrado o funcionamento de um *software* construído durante este trabalho para a análise dos *bytecodes* em busca das instruções candidatas a *folding*.

O capítulo 5 mostra o Femtojava *Low-Power*, detalhes sobre seus cinco estágios de *pipeline*, a implementação do banco de registradores para funcionar como pilha, os problemas encontrados na dependência de dados, além de demonstrar como o uso da técnica de *forwarding* que, tirando vantagem da arquitetura particular de pilha, pode

trazer enormes ganhos relativos à redução de potência, além do conhecido ganho em performance.

O capítulo 6 apresenta o desenvolvimento do Femtojava VLIW. Inicialmente o analisador de *bytecodes*, responsável pela busca de paralelismo nos programas Java, é estudado em detalhes: seu funcionamento, seus algoritmos, suas configurações. Finalmente, o processador VLIW em desenvolvimento e resultados parciais em termos de performance, energia e área são apresentados.

O último capítulo deste trabalho apresenta várias sugestões de trabalhos futuros: alguns rumos que podem ser tomados relativos ao simulador de potência CACO-PS, também alternativas arquiteturais que podem ser estudadas em um processador de pilha Java, com o uso de técnicas trazidas do mundo RISC para este outro paradigma. CMP (*Chip Multiprocessing*), onde vários processadores compatíveis em relação ao conjunto de instruções são integrados em um mesmo CI e se comunicam em alta velocidade (PALACHARLA; JOUPPU; SMITH, 1996), é um exemplo. O uso de técnicas antigas que nunca foram testadas em um processador de pilha, como *software pipelining* (JONES; ALLAN, 1990), que tem como intuito expor melhor paralelismo e consequentemente melhorar o desempenho da versão VLIW do processador, é outro exemplo. Finalmente, em conclusões, uma análise é feita do trabalho em geral, de suas contribuições e seus resultados, assim como dos pontos que devem ser aperfeiçoados.

2 PROCESSADORES JAVA E VLIW

2.1 A linguagem Java

A *Sun Microsystems* anunciou formalmente a linguagem Java em maio de 1995. Java é a primeira linguagem que tem integrada, sendo projetada assim a partir do zero, a capacidade para lidar com aplicações em rede. Java, por definição, é simples, orientada a objetos, distribuída, interpretada, robusta, segura, independente de plataforma e consequentemente portável, suporta múltiplas *threads* e é uma linguagem dinâmica. Além do mais, Java é uma linguagem fortemente tipada e que não utiliza, ao menos explicitamente, ponteiros, diminuindo a possibilidade da geração de código com erros. Java também oferece manutenção automática da memória, também chamada de *garbage collector*, fazendo com o que o programador não tenha que explicitamente desalocar da memória objetos que não estão sendo mais utilizados.

A grande vantagem de Java é ser simples e ao mesmo tempo poderosa. Java é uma linguagem de programação e também uma linguagem para *intranet* e internet, já que traz várias facilidades para ser executada em rede. Java segue a filosofia "*write once, run everywhere"*, o que quer dizer que, uma vez escrito um programa nesta linguagem, ele pode ser executado em qualquer plataforma sem nenhuma alteração. O código fonte Java é compilado para um código da máquina virtual Java. Este código é chamado de *bytecode.* Assim, este código sempre é executado sobre uma máquina virtual, que por sua vez faz a interface com a máquina nativa, onde está instalada. Como conseqüência, o programa Java pode ser colocado em uma página Web, e ser executado no cliente, que pode ser um PC. Este PC pode estar executando o sistema operacional Windows, ou ainda Linux. Mais, este programa Java pode ser executado em um *iMac*, da *Apple*, ou ainda em uma estação *Unix* da *Sun.*

Java possui diferentes plataformas de desenvolvimento, focadas em diferentes nichos de aplicação. A J2ME é feita para Java trabalhar com pequenos dispositivos, pois inclui um subconjunto de instruções Java e sua API (*Application Program Interface*) também possui um conjunto de instruções menor que das outras versões, J2SE e J2EE, mostradas na Figura 2.1. Além do mais, J2ME possui algumas características que são relevantes apenas para dispositivos embarcados. Por exemplo, o suporte gráfico e a capacidade de acesso a banco de dados no J2ME são menos sofisticados do que nas outras versões. O J2ME geralmente incorpora o CLDC (*Connected Limited Device Configuration*), que é implementado no topo das camadas do sistema operacional e serve como uma interface entre o sistema operacional e os programas baseados em Java. O CLDC geralmente usa a KVM, uma versão reduzida e menos funcional que a JVM, para sistemas embarcados. Finalmente, a MIDP (*Mobile Information Device Profile*)

fica acima da CLDC e oferece um conjunto de API que define, por exemplo, como telefones móveis irão fazer a interface com outras aplicações.

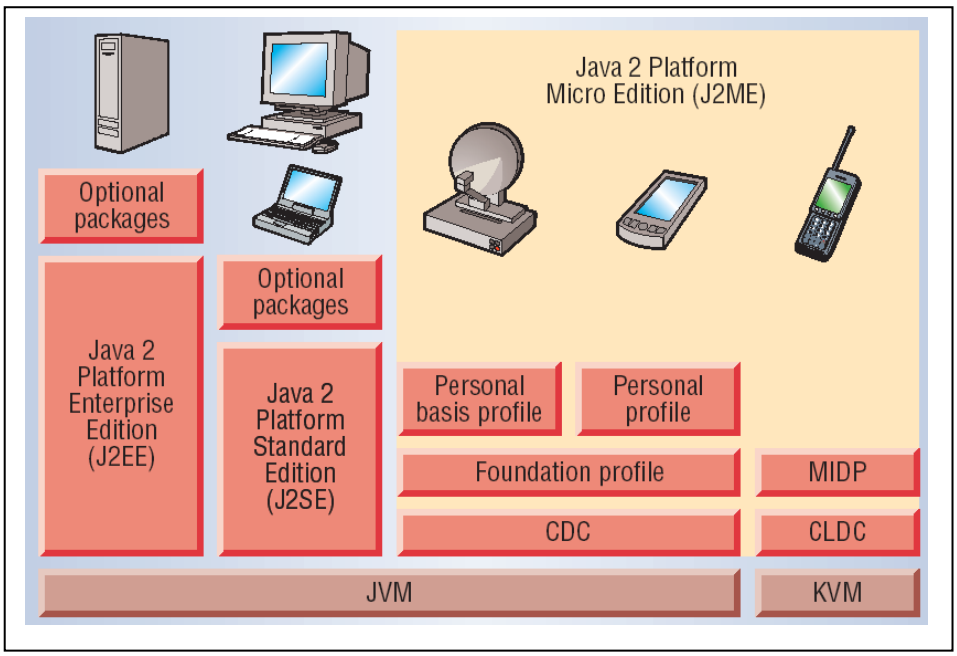

Figura 2.1 : As várias plataformas de desenvolvimento para Java (LAWTON, 2002)

Como já citado no capítulo 1 e mostrado na Figura 1.2, há várias maneiras de um programa Java ser executado. O uso da máquina virtual é apenas uma delas. O grande problema do uso da máquina virtual é que ela é uma camada adicional para a execução do programa Java na máquina de destino, tornando a execução mais lenta. Assim, outras alternativas foram implementadas para fugir da interpretação direta do código.

Uma delas é o uso de compilador JIT (*Just in Time*) (KRALL, 1998) (WILKINSON, 1998) (CRAMER et al., 1997). Geralmente o JIT funciona da seguinte maneira: da primeira vez que um programa é executado, o trecho de *bytecodes* que está sendo executado é passado para o código nativo da máquina. Da próxima vez que este trecho for utilizado, será executado o trecho já traduzido que estava guardado em memória, cujo desperdício é o grande problema desta técnica.

Também, compiladores de Java para o conjunto de instruções nativo da máquina já foram desenvolvidos (GNU, 2004). Entretanto, estes compiladores geralmente não suportam todos os recursos da linguagem. Além do mais, certa parte da filosofia de portabilidade de Java é perdida. Finalmente, uma outra alternativa é a execução direta de *bytecodes* em *hardware*. A execução de Java em *hardware* é particularmente interessante para sistemas embarcados, pois une a facilidade de desenvolvimento de Java mantendo simultaneamente a velocidade de execução e portabilidade que Java não oferece quando é executada em *software*.

De qualquer maneira, Java é baseado em um processador de pilha para executar seus *bytecodes*. Então, seu paradigma de execução é um pouco diferente de processadores convencionais (KOOPMAN, 1989), baseados em transferência de valores entre registradores. Por exemplo, para fazer a operação de multiplicação em um processador

RISC típico, suponha-se que haja uma instrução chamada mul. Ela é usada da seguinte forma:

```
mul r3, r5, r7
```

O registrador r3 recebe o resultado da multiplicação dos valores encontrados em r5 e r7. Em processadores de pilha, para fazer a mesma operação, são necessários os seguintes passos:

- Empilha operando 1
- Empilha operando 2
- Multiplica

A instrução multiplica busca estes dois operandos da pilha, faz a operação e põe o resultado novamente no topo da pilha.

Esta pilha na realidade é chamada de pilha de operandos (do inglês *operand stack*). É necessário ressaltar a diferença do nome, pois além da pilha de operandos, o processador Java também faz uso da pilha para desempenhar outras funções, entre elas, guardar os valores das variáveis automáticas, isto é, das variáveis locais do método, além de guardar dados de retorno para os métodos que foram invocados, como também ocorre em outros processadores que não implementam uma pilha de operandos.

Uma típica pilha que pode ser implementada em um processador Java é observada na Figura 2.2:

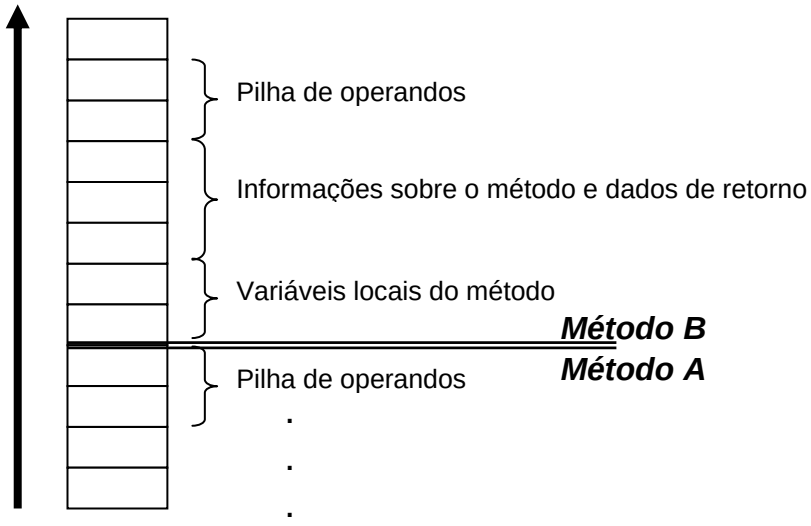

Figura 2.2 : Organização da pilha de um processador Java típico

A invocação e retorno de métodos é baseada na utilização de *frames*. Cada método possui um *frame* que pode ser intercalado com um outro *frame* de outro método. Isto é muito útil para a passagem de parâmetros: quando um método quer passar valores como parâmetro para outro método na invocação, o primeiro simplesmente empilha os valores, e faz a chamada de métodos. O próximo método é configurado para que os valores passados como parâmetro do método anterior sejam as primeiras variáveis locais daquele método. Isto economiza a necessidade de cópias de parâmetros entre métodos.

O conjunto de instruções de Java consiste em 201 *opcodes*, dos quais vários desempenham funções semelhantes. Pegando como exemplo a instrução iload, que

carrega no topo da pilha uma variável que está localizada no repositório de variáveis locais. O segundo byte desta instrução indica o índice de onde a variável desejada se encontra. Entretanto, também há mais quatro instruções que executam a mesma função: iload_0, iload_1, iload_2, iload_3, que carregam uma variável local de índice 0, 1, 2 e 3 do repositório, respectivamente, repetindo a função que poderia ser desempenhada pela instrução iload simples. Além do mais, há instruções load para tratar com diferentes formatos de número: *long*, *double*, *float* (por exemplo, fload), e para cada um desses, instruções com referência (como fload_0, fload_1, fload_2, fload_3). O benefício desta sobreposição de funções entre instruções traz como vantagem um código compilado resultante mais compacto. Todavia, a grande quantidade de instruções e variantes pode representar um grande obstáculo ao projetista no desenvolvimento de uma plataforma Java em *hardware*.

O conjunto de instruções pode, basicamente, ser dividido nos seguintes grupos:

- *Load/Store:* Contém todas as instruções de load e store, que transferem valores entre o repositório de variáveis locais do método e a pilha de operandos, em um total de 70 *opcodes*.
- Manipulação da pilha: Todas as instruções de manipulação da pilha, como dup (que duplica o valor do topo da pilha), push e pop. Há 9 *opcodes* neste grupo.
- Criação e manipulação de objetos: Contém instruções para criar e manipular instâncias das classes e vetores. Há 27 instruções neste grupo.
- Aritméticas e conversão de tipo: Existem 53 instruções neste grupo, entre elas, as de adição, subtração, multiplicação, divisão e manipulação de *bits*.
- Invocação e retorno de métodos: Há 10 instruções para a invocação e retorno de métodos.
- Miscelânea: Há ainda mais 4 *opcodes* para dar suporte a *multi-threading* e tratamento de exceções.

Nas seguintes seções, são mostradas diferentes técnicas de execução de Java em *hardware*. Algumas implementam processadores em si, outras são técnicas que, agregadas a um outro processador, facilitam ou tornam mais eficiente a execução do *bytecode* Java. Outra ainda faz o papel de co-processador para a *binary translation*.

2.2 picoJava-I

O processador picoJava-I (O'CONNOR; TREMBLAY, 1997) (MCGHAN; O´CONNOR, 1998) (HANGAL; O´CONNOR, 1999] foi o primeiro processador Java para sistemas embarcados a ser disponibilizado no mercado e foi produzido pela Sun Microsystems, em 1997. Sua microarquitetura pode ser observada resumidamente na Figura 2.3.

Este processador oferece uma relativa flexibilidade: as partes que estão em cinza na Figura 2.3 são configuráveis: o tamanho das *caches* de instrução e de dados é variável (pode ter de 0 a 16 Kbytes) e a unidade de ponto flutuante é opcional (quando ela não está presente, instruções de ponto flutuante são emuladas). O picoJava-I suporta todo o conjunto de instruções Java, entretanto, apenas estão implementadas diretamente em *hardware* aquelas que são as mais comuns em programas Java. A maioria destas instruções leva de um a três ciclos para serem executadas, como instruções de adição de inteiros ou leitura de um campo de objeto. Outras instruções mais complexas, como invocação de métodos, são imprescindíveis para manter a boa performance da execução de um programa Java. Assim, este tipo de instrução é executado através de microcódigo.

Há ainda uma terceira possibilidade de execução de instruções: por emulação. Instruções ainda mais complexas, todavia não tão frequentemente utilizadas, como a instrução para a criação de um objeto, são emuladas. Quando o picoJava-I encontra uma destas instruções, ele ativa uma *trap* e o sistema operacional se encarrega de executar a instrução.

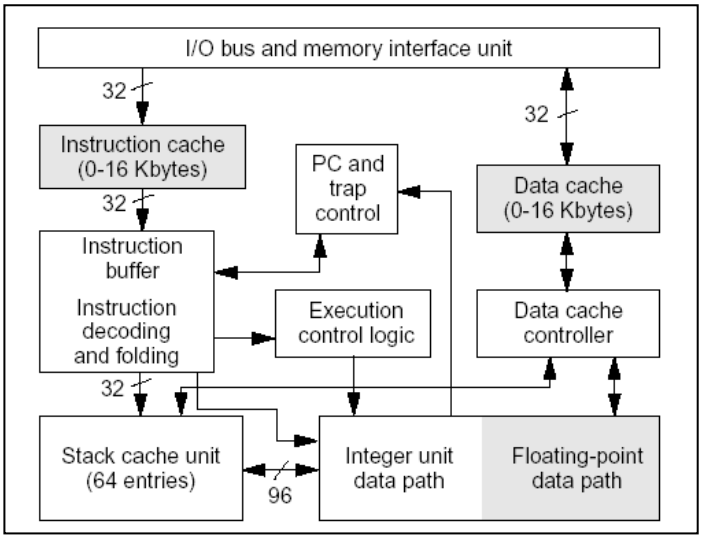

Figura 2.3 : Microarquitetura do processador picoJava-I (O'CONNOR; TREMBLAY, 1997)

Além das instruções Java convencionais, há um conjunto estendido delas, que permite aos programadores escreverem código no nível do sistema. Este conjunto estendido de instruções recai nos seguintes tipos: *load/store* em lugares arbitrários da memória, que podem ser usados, por exemplo, para a comunicação com dispositivos mapeados em memória; manutenção da *cache*, com uma instrução para apagar todo o conteúdo desta memória, com propósito de manter a integralidade dos dados; acesso aos registradores internos, que permite acessar o conteúdo dos registradores de estado do processador, útil para a troca de contexto; e miscelânea.

O *pipeline* do picoJava-I pode ser observado na Figura 2.4. O primeiro estágio, de busca de instruções, possui uma fila de instruções de 12 bytes, onde 4 bytes são buscados da memória *cache* de instruções por ciclo. O processador pode decodificar até 5 bytes que estão na frente da fila e mandá-los para o próximo estágio, onde são decodificados e poderão sofrer o processo de *folding*, que será explicado detalhadamente no capítulo 5. Depois disso, a instrução vai para o terceiro estágio onde será executada, e este processo poderá levar um ou mais ciclos. Também neste estágio, as instruções podem acessar a *cache* de dados, caso necessário. Finalmente, chega-se ao último estágio, onde o resultado é gravado no banco de registradores.

As primeiras 64 entradas da pilha são guardadas em um banco de registradores, organizados na forma de um *buffer* circular. Existem duas marcas: *low water mark* e *high water mark*, que indicam o início e o final da pilha no *buffer* circular. Quando o *high water mark* se aproxima muito do *low water mark*, significa que o topo da pilha está se aproximando de seu início no *buffer* circular, consequentemente correndo riscos de apagar dados válidos. Assim, quando isso acontece, uma parte do início da pilha é gravada na *cache* de dados e o *low water mark* é atualizado. A pilha agora tem mais espaço no *buffer* para crescer. Da mesma forma, quando as duas marcas estão muito

distantes, significa que o topo da pilha está muito longe de seu início no *buffer*. Então, dados da pilha abaixo deste início são carregados da *cache* de dados para o *buffer*, e o *low water mark*, atualizado. Todo este processo é chamado de *dribbling* (O'CONNOR; TREMBLAY, 1997).

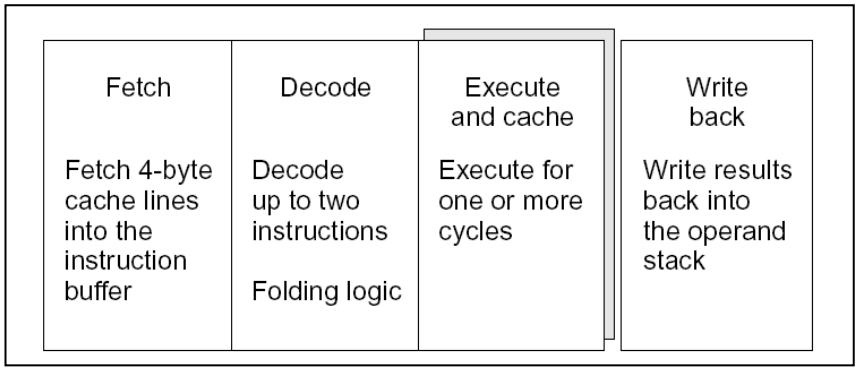

Figura 2.4 : Os quatro estágios do *pipeline* do Picojava-I (O'CONNOR; TREMBLAY, 1997)

Este processador não possui predição de desvio. Todos os desvios condicionais encontrados são considerados como falsos. Já que o *pipeline* deste processador é curto, com 4 estágios, uma penalidade de apenas 2 ciclos é paga caso um desvio condicional seja verdadeiro. O picoJava-I também tem suporte a monitores, que servem para manter a consistência dos dados, em casos onde objetos são compartilhados por várias *threads*. Além do mais, há suporte para *garbage collector* em *hardware* e *software*, que automaticamente detecta objetos não mais usados em memória e os apaga, de forma transparente ao usuário, como manda a especificação da linguagem Java.

A Figura 2.5 mostra o potencial da execução nativa em *hardware* de *bytecodes*, comparando o tempo de execução nativa de Java no picoJava-I, com a execução através de JIT e a execução interpretada em um Pentium e um 486, sendo que a freqüência de todos processadores está normalizada em 100Mhz. Os *benchmarks* foram o compilador Java e o programa *Raytracer*, que é uma aplicação gráfica composta basicamente de operações em ponto flutuante.

		Benchmark performance (s)	
Method	System	Javac	Raytracer
Native	picoJava-I	1.8	13.0
JIT	Pentium	9.3	64.5
	486	10.7	109.5
Interpreter	Pentium	20.4	174.3
	486	27.3	254.8

Figura 2.5 : Comparação no tempo de execução de dois programas Java (O'CONNOR; TREMBLAY, 1997)

2.3 picoJava-II

O processador picoJava-II (SUN, 1999) (SUN, 1999b) foi desenvolvido pela Sun Microsystems baseado em seu antecessor, explicado anteriormente. Ele herda diversas

características do picojava-I, mas possui um mecanismo de *folding* mais avançado e um *pipeline* com 6 estágios, mais longo, e que consequentemente permite períodos do ciclo de relógio mais curtos. Na realidade, a Sun Microsystems classifica o picoJava-II como um IP (*Intelectual Property*), deixando-o disponível para ser licenciado, fazendo parte de sua biblioteca. A sua descrição em RTL (*Register Transfer Level*) e documentação está aberta publicamente e várias empresas, como a IBM, Fujitsu e LG, já licenciaram esta tecnologia.

O *pipeline* pode ser observado na Figura 2.6. No primeiro estágio ocorre a busca de instruções na *cache* de instruções ou na memória principal. É no estágio de decodificação onde o microcódigo é preparado e onde o mecanismo de *folding* entra em funcionamento. No terceiro estágio há a busca de operandos, que serão usados no quarto estágio, de execução. O estágio 5 é adicional, e serve para a busca de algum dado na *cache* de dados. Um exemplo é a instrução getstatic, que busca um valor de uma variável estática da memória e a coloca no topo da pilha. Finalmente, o último estágio salva o resultado final da instrução.

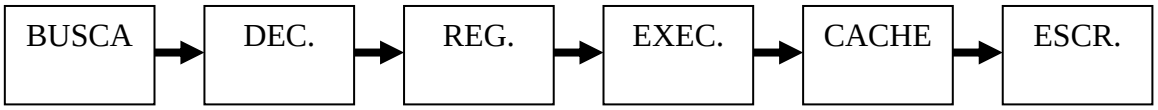

Figura 2.6 : Os seis estágios do *pipeline* do picojava-II

A microarquitetura do picoJava-II é resumidamente mostrada na Figura 2.7. A *cache* de instruções tem tamanho variável, da mesma forma que a primeira versão do picoJava, podendo apresentar 1, 2, 4, 8 ou 16 Kbytes, ou não estar presente. A *cache* de dados possui o mesmo tamanho variável da de instruções, e retorna valores vindos do gerenciador de *dribble*, técnica já explicada anteriormente. A unidade de inteiros (*Integer Unit Data Path*), executa todas as instruções que não são de ponto flutuante (as quais possuem uma unidade própria responsável, que é opcional). Ela também é responsável por executar as instruções estendidas, que é o mesmo conjunto do picoJava-I. A Unidade de Interface com Barramento (*I/O Bus and Memory Interface Unit*) fornece uma interface entre o núcleo, a memória e outros dispositivos de entrada e saída. Há também outras unidades que não estão sendo demostradas na figura, como a unidade de gerenciamento de Pilha, responsável pela manutenção da pilha, chamada de métodos, etc., e a Unidade de Desligamento, Relógio e Varredura, que integra o gerenciamento de energia, *reset* do sistema, teste e varredura.

2.4 aJile aJ-100

O aJile aJ-100 (AJILE 2001) é um micro controlador Java que executa diretamente *bytecodes* Java. Entretanto, ele possui suporte a primitivas de tempo real, implementadas como *bytecodes* estendidos, eliminando, conforme o fabricante, a necessidade da utilização de um RTOS. Além disso, também possui outras instruções que não fazem parte originalmente do conjunto da linguagem Java, específicas para sistemas embarcados. Isto, novamente conforme o fabricante, resulta em um atraso muito pequeno, menos de 1µs, na troca de contexto entre *threads*.

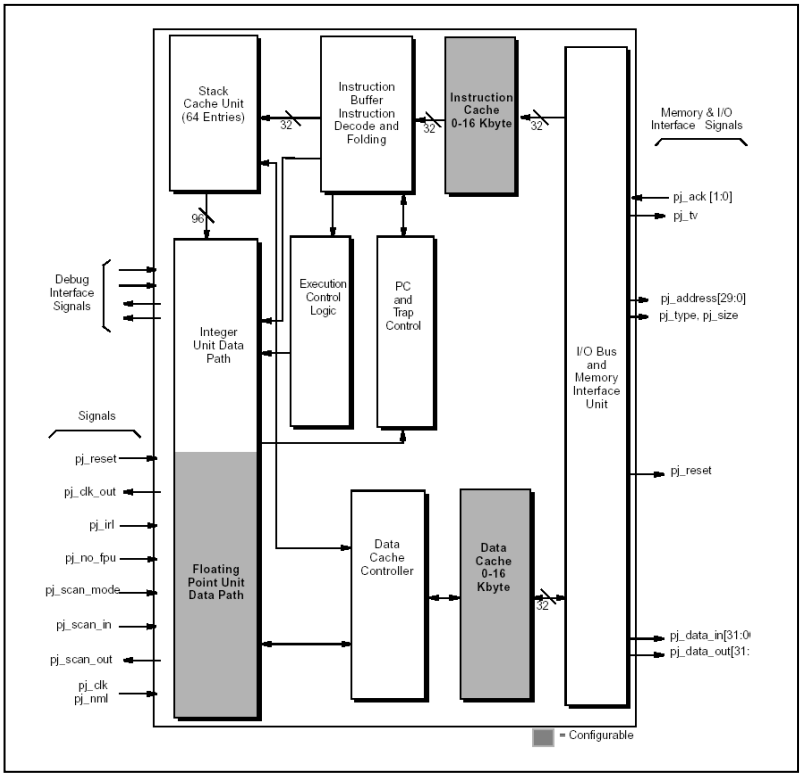

Figura 2.7 : A microarquitetura do picoJava-II (SUN, 1999a)

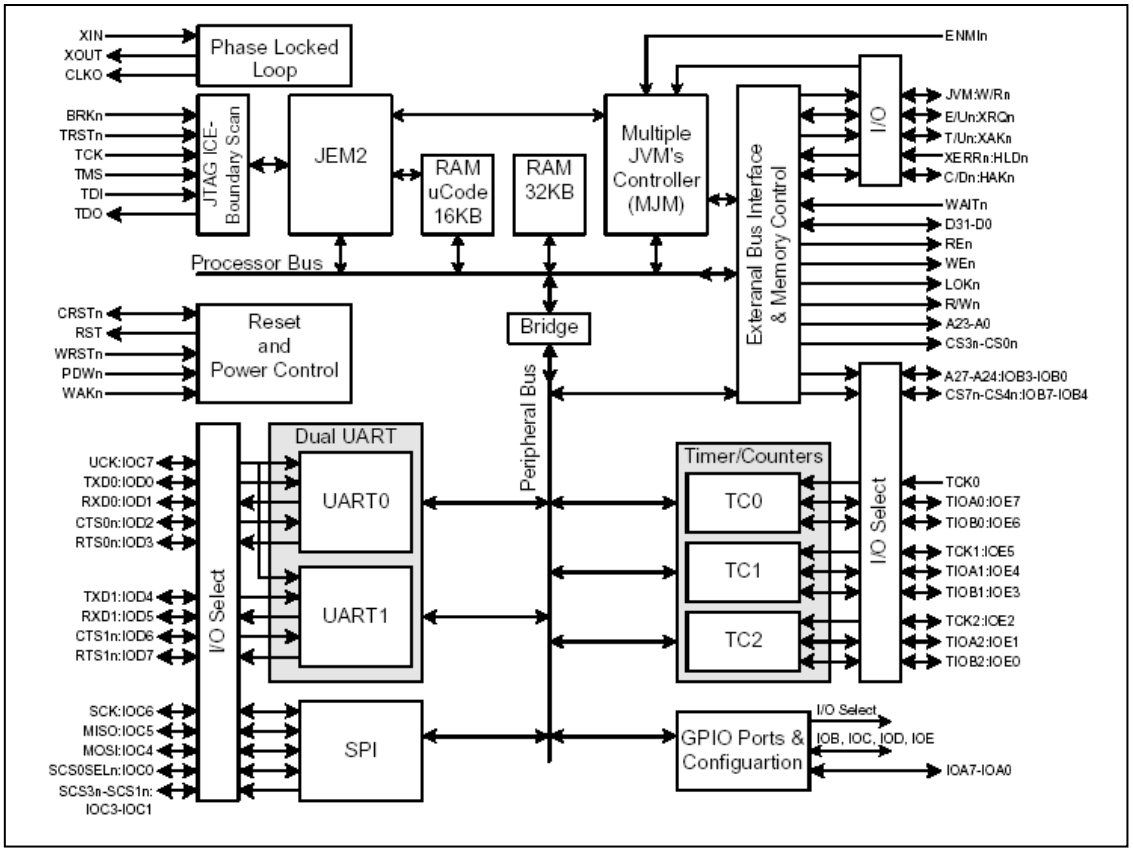

Figura 2.8 : Estrutura do SoC Ajile-aJ100 (AJILE 2001)

O aJile aJ-100 é na realidade um pequeno SoC, como pode ser observado na Figura 2.8, possuindo, além do processador (JEM2), um gerador de *clock*, controlador de barramento, memória e E/S, memória interna, controladores de porta serial e *timers*. A microarquitetura de seu núcleo pode ser observada na Figura 2.9.

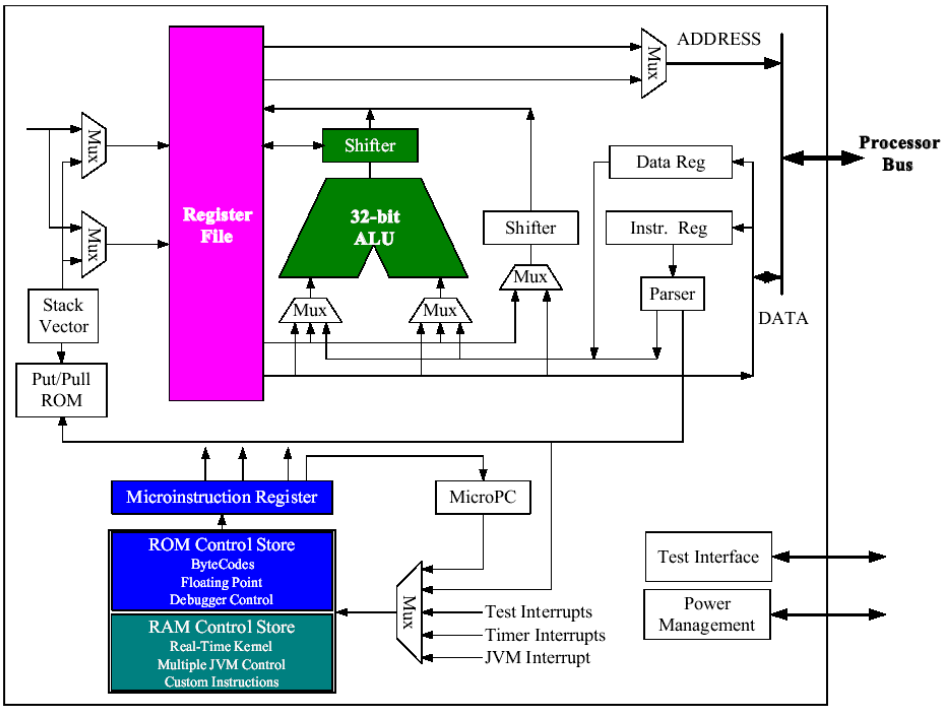

Figura 2.9 : Microarquitetura do núcleo do processador Ajile aJ-100 (AJILE 2001)

2.5 Jazelle

Jazelle (ARM, 2003), ao contrário dos anteriores, não é um processador, mas sim uma tecnologia desenvolvida pela ARM. Os processadores ARM historicamente suportam dois conjuntos de instruções: o próprio conjunto ARM, onde todas as instruções têm um comprimento de 32 *bits*, e o conjunto de instruções *Thumb*, que compacta as instruções mais usadas em um formato de 16 *bits*, oferecendo uma compactação do código na ordem de 35% a 40%. O conjunto de instruções suporta chamada de procedimentos entre o código ARM e *Thumb*. Assim, os programadores podem escolher em tempo de compilação quais partes da aplicação são compiladas para performance (ARM) ou tamanho do código (*Thumb*).

A tecnologia Jazelle estende este conceito adicionando um terceiro conjunto de instruções, o conjunto de instruções Java, no processador. Assim, para o programador, o ARM possui agora um novo modo, ou estado, no qual o processador se comporta como uma máquina Java: ele busca e decodifica *bytecodes* Java e mantém uma pilha de operandos. O processador pode chavear entre os estados Java e ARM/Thumb. Há um *bit* no registrador de estados do processador que indica qual modo ele se encontra, como mostra a Figura 2.10.

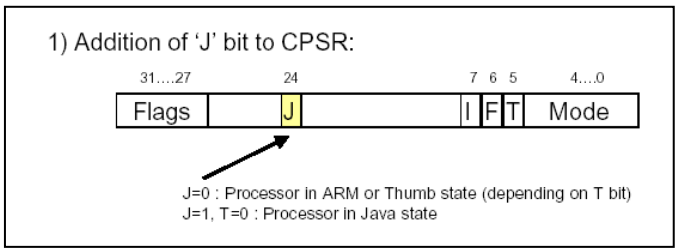

Figura 2.10 : Registrador de estados do ARM. O *bit* 24 indica se o processador está em modo Java ou não (ARM, 2003)

No estado Java, o processador reutiliza vários registradores ARM para funções específicas da máquina Java, como apontador do topo da pilha, os quatro primeiros elementos da pilha e apontador do repositório de variáveis locais do método. Com a reutilização do *hardware*, aproximadamente 12000 portas adicionais foram necessárias para implementar o estado Java no processador ARM.

Como o picoJava, a tecnologia Jazelle divide as instruções Java em classes: executadas diretamente, emuladas e não definidas. A maioria das instruções (140) são executadas diretamente em *hardware*, e o restante (94), são emuladas utilizando instruções ARM. As funções básicas de uma JVM, como o carregador de classes, *garbage collector*, verificador de classes, administrador de processos e memória não são executadas em *hardware*, como pode ser observado na Figura 2.11. Esta técnica parece trazer uma boa performance, como mostra um gráfico, mostrado na Figura 2.12, fornecido pelo fabricante. Entretanto, ele não possui parametrização alguma relacionada à área nem possui dispositivos para a redução da potência consumida no circuito. Além do mais, mesmo que o usuário queira apenas executar aplicações Java, terá que arcar com todo o custo adicional de *hardware*, e consequentemente também de potência dissipada, das instruções ARM.

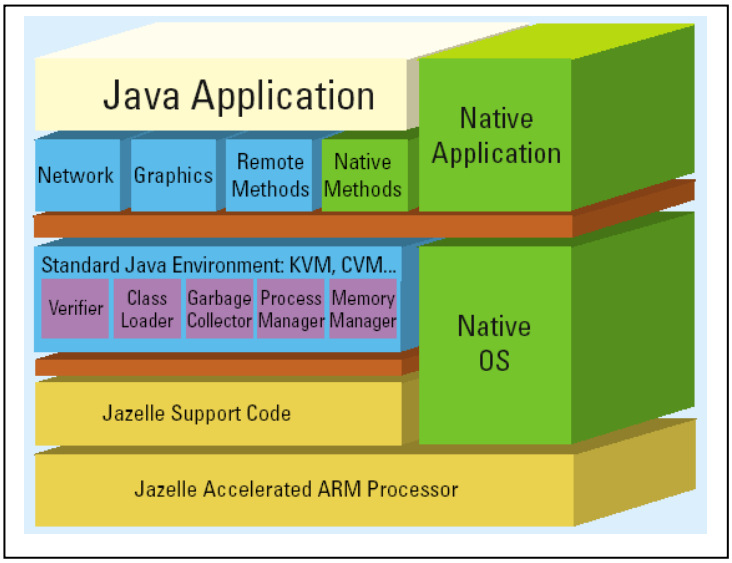

Figura 2.11 : Camadas do processador ARM com a tecnologia Jazelle e *runtime* Java (ARM, 2003)

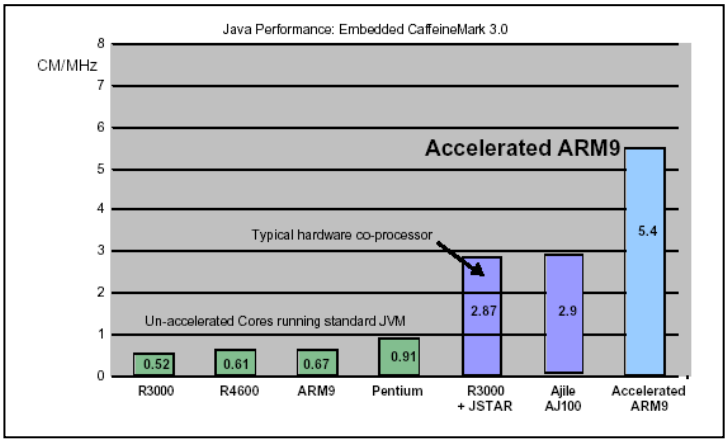

Figura 2.12 : Comparação de desempenho entre a ARM com a tecnologia Jazelle com outros processadores (ARM, 2003)

2.6 JStar

Este co-processador Java (NAZOMI, 2003) se difere dos outros porque, ao contrário dos anteriores, ao invés de executar diretamente *bytecodes*, ele faz o trabalho de *binary translation* para o processador alvo. Ele funciona da seguinte maneira: fica localizado entre a *cache* de instruções e o processador, ou ainda entre a memória e a *cache* de instruções, como pode ser observado na Figura 2.13. Quando ele não está sendo usado, permanece desligado. Entretanto, quando algum programa Java é executado, o processador é ligado, lê os *bytecodes* diretamente da memória e os transforma para a linguagem nativa do processador. Na realidade, o que ele faz é a interpretação dos *bytecodes*, fazendo em *hardware* o que a JVM faria em *software*.

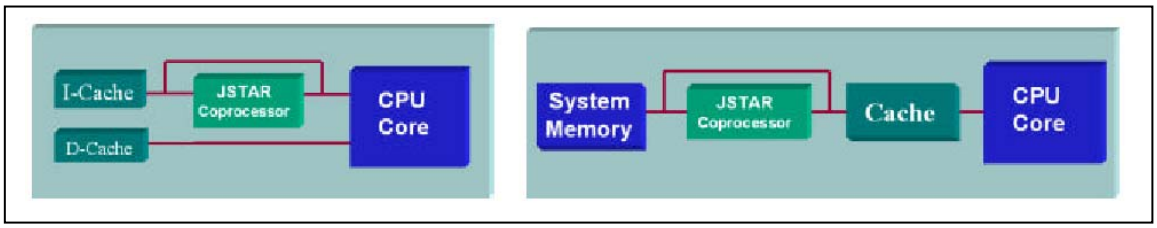

Figura 2.13 : Como o co-processador JStar pode funcionar integrado a outro processador (NAZOMI, 2003)

O processador ocupa aproximadamente 30000 portas para ser implementado e necessita de 45000 *bits* de memória, além de ser um núcleo sintetizável (*soft*), cuja implementação pode ser integrada a qualquer outro núcleo de processador no mesma pastilha, sendo compatível com processadores do tipo CISC e RISC com qualquer arquitetura de memória. O JStar está disponível para ser licenciado como um IP. Contudo, mesmo sendo um *soft* IP, pode trazer dificuldades para a adaptação com o processador, devido às mais diversificadas interfaces com memória existentes. Somando-se a isso, como o processador ARM, não oferece nenhuma parametrização relativa à área ou técnicas para diminuição da potência consumida.

2.7 Komodo

O projeto Komodo (BRINKSCHULTE et al., 1999) (BRINKSCHULTE et al., 1999b) implementa uma extensão do processador picoJava, preparando-o para suportar *multithreading*, com múltiplos bancos de registradores para pilhas, contadores de programas e fila de instruções. Além do mais, traz suporte a sistemas de tempo real.

Java não foi desenvolvido primeiramente para ser usado em sistemas de tempo real, já que o bom funcionamento de tais sistemas depende basicamente do melhor caso/pior caso para execução, além de previsibilidade e rapidez no tratamento de eventos. Há várias características do processador picoJava que promovem ou inibem a previsibilidade no tratamento de eventos em sistemas de tempo real.

Gargabe collection, por exemplo, é particularmente sensível em sistemas embarcados de tempo real, onde espaço e restrições de tempo são fatores cruciais. Em sistemas que não são de tempo real, geralmente o *garbage collector* trabalha até toda a desalocação dos objetos não usados da memória estar completa, sem ser interrompido. Para sistemas de tempo real, é preciso que ele seja interrompido em fatias de tempo bem definidas, e de forma que possa continuar posteriormente, para garantir a previsibilidade do sistema. Outro assunto interessante é a manutenção da *cache*, já que podem ocorrer falhas de *cache* e também ocasionar atrasos no sistema. Entretanto, processadores Java têm certa vantagem nisso, já que o tamanho de seu código de instruções, devido a sua natureza CISC, é geralmente menor do que em processadores RISC. Além disso, como o Komodo é um processador que suporta várias *threads*, os atrasos da *cache* na maioria das vezes pode ser ocultado: enquanto a *cache* é atualizada, outra *thread* é executada. Outro grande problema é o uso da técnica de *dribbling*, já explicada anteriormente, considerando-se o tempo gasto para a atualização do banco de registradores da pilha.

Além do mais, o Komodo trata as interrupções de um evento externo de forma diferente: se uma interrupção está ocorrendo, outro evento externo pode tomar o seu lugar apenas se for uma interrupção de uma prioridade mais alta ou não mascarável. Para evitar que eventos externos, por este motivo, demorem para serem atendidos, é usada a metodologia de tratar interrupções usando-se *threads*: quando acontece um evento externo, uma nova *thread* relacionada àquele evento é criada e executada, ao invés de ser consultado o vetor de interrupções e uma rotina referente à interrupção ser chamada. Isto só é possível já que o Komodo suporta *multithreading* em *hardware*, com suporte a prioridade de *threads*, também em *hardware.*

Como pode ser observado na Figura 2.14, o Komodo é uma extensão do picoJava que possui suporte a *multhreading* em *hardware*, permitindo uma rápida troca de contexto entre elas; uma unidade de sinal (*signal unit*), que basicamente cuida de eventos externos com prioridades diferentes; e graças ao administrador de prioridade (*priority manager*) há a possibilidade de verificação das *threads* mais importantes a serem executadas e quais estão suspensas ou ativas, características estas importantes para um sistema de tempo real.

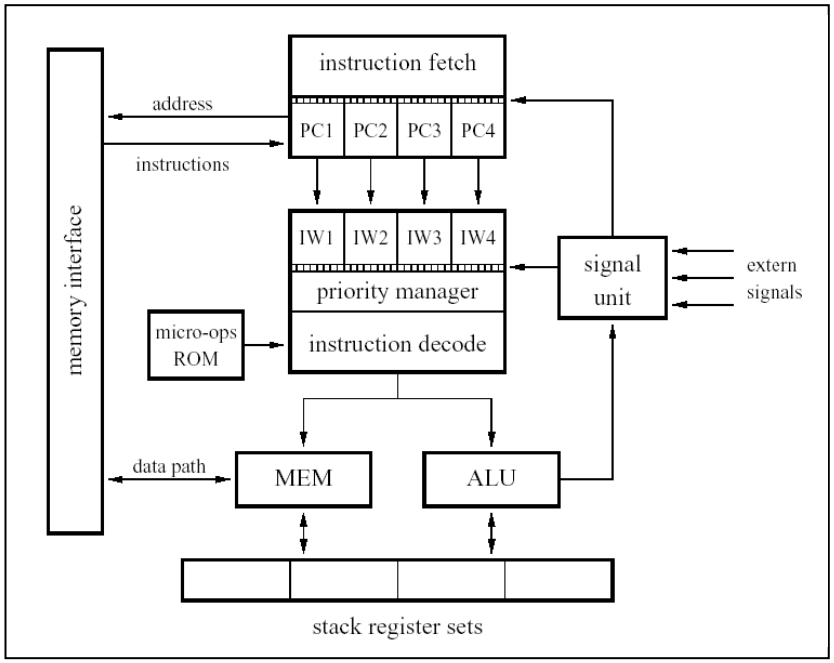

Figura 2.14 : A microarquitetura do processador Komodo, estendida do picoJava (BRINKSCHULTE, 1999)

2.8 MAJC

O processador MAJC (*Microprocessor Architecture for Java Computing*) (TREMBLAY; CHAN; CHAUDHRY, 2000) (KOWALCZYK et al., 2001) foi criado para lidar com tarefas complicadas e que exigem um alto poder de processamento, como *video on demand* com qualidade de HDTV, reconhecimento de voz, navegação virtual e em 3D. O MAJC explora paralelismo de várias formas: não apenas nas instruções, mas também nos dados, entre *threads* e entre processos. Além do mais, este processador foi desenvolvido de forma mais geral possível para facilitar a compatibilidade com outras ISA (*Instruction Set Architecture*) através de *binary translation* (como a tradução de *bytecodes* Java para o ISA do MAJC usando o *Sun's Hotspot Virtual Machine*).

O processador MAJC é um processador VLIW e pode ter vários níveis de parametrização. No nível mais baixo, consiste em fatias de instruções (ou *instruction slices*) que provêem o controle, *status* e os registradores da arquitetura, além dos demais recursos necessários para executar as instruções de um pacote VLIW. Assim, pode-se haver de uma a quatro fatias para executar um pacote VLIW, onde cada fatia é responsável por um fluxo de instruções que varia conforme o número de instruções que o pacote VLIW suporta. Por exemplo, se o pacote VLIW possui duas instruções, duas fatias são implementadas. Este conjunto de fatias de instruções forma uma *microthread*. Uma ou mais *microthreads*, adicionando-se registradores de controle e *status*, são combinados para formar uma unidade do processador. Finalmente, uma ou mais unidades formam um *cluster* de processadores. A arquitetura suporta uma rápida comunicação entre *microthreads* e unidades de processadores em um *cluster* através de interrupções velozes e canais virtuais.

Assim, a arquitetura suporta vários tipos de paralelismo:

- Paralelismo nos dados, através de SIMD (*Single-Instruction, Multiple-Data*).

- Paralelismo nas instruções, através de VLIW, onde seus pacotes podem conter de uma a quatro instruções
- Paralelismo nas *threads* e nos processos através de *multitrheading* vertical (ALVERON et al., 1990) e comunicação rápida entre processadores no mesmo chip (CMP).

As instruções de um pacote VLIW não podem ter dependência de controle nem de dados nem de recursos. Um compilador para o MAJC é responsável por se certificar que os pacotes VLIW respeitam estas regras. Além disso, ele também é responsável por verificar dependências entre pacotes e pela alocação de recursos, como uso das unidades funcionais. Em processadores VLIW convencionais, se não há paralelismo de instruções suficiente para completar o pacote VLIW, este pacote é completado com instruções de NOP (*no operation*). Isto aumenta a largura de banda necessária na comunicação de memória e da *cache*, trazendo outras conseqüências, como aumento de potência, além do incremento no tamanho do código. Assim, o pacote VLIW do MAJC possui um tamanho variável, que pode contar com uma a quatro instruções de 32 *bits* cada. Seu comprimento em número de instruções é indicado por um cabeçalho na primeira instrução (que por conseqüência suporta um conjunto de instruções menor do que os outros lugares do pacote), como mostra a Figura 2.15. Em conseqüência disto, o tamanho do executável final tem um tamanho muito parecido com daqueles de processadores RISC (TREMBLAY; CHAN; CHAUDHRY, 2000).

O ISA do MAJC é composto por instruções tipicamente RISC, alguma delas com suporte a SIMD, além de outras para melhorar a performance de aplicações típicas de serviço de banda larga (como várias instruções de ponto flutuante de precisão simples e dupla, bastante usadas em aplicações gráficas e de multimídia) assim como de programas Java. Por exemplo, dentro do ISA do MAJC há a instrução blkzero, que escreve zeros em uma determinada região da memória. É típico em máquinas virtuais Java inicializarem uma grande porção de memória com zeros. Também possui a instrução bndchl (*bound check*). Esta instrução verifica três condições necessárias de um vetor de acesso em Java e produz uma *trap* se alguma delas for verdadeira. Além destes exemplos, há outras instruções específicas para aumentar a performance de execução de programas Java.

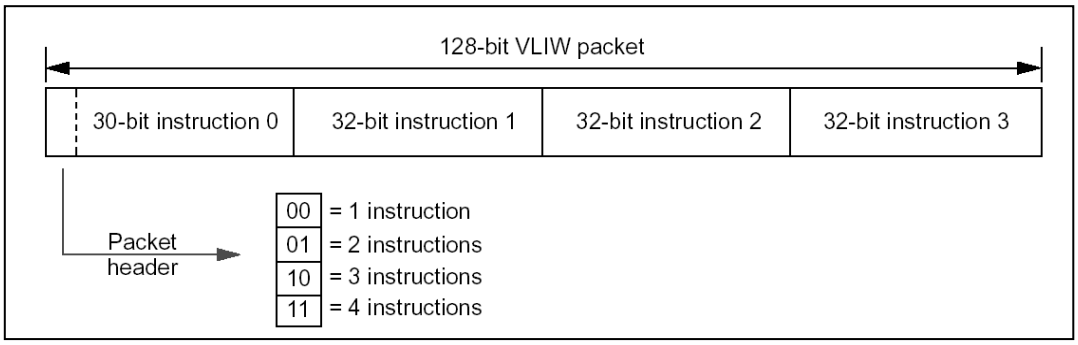

Figura 2.15 : O formato do pacote VLIW (TREMBLAY; CHAN; CHAUDHRY, 2000)

Além do mais, o formato das instruções usa quatro especificadores de registradores, característica esta bastante importante para aplicações multimídia, como *muladd* (multiplicação e adição na mesma operação), bitext (extração de *bits*), entre outras. Estas operações, que requerem três operandos de origem e um de destino, não podem ser implementadas em arquiteturas tradicionais. Também, o MAJC possui um conjunto

enorme (até 128 por fatia) de registradores de propósito geral, reduzindo assim o acesso à memória devido à falta de espaço que pode ocorrer no banco de registradores.

A Figura 2.16 mostra um exemplo de uma arquitetura MAJC. Este exemplo possui quatro unidades de processadores. Duas delas possuem quatro fatias de instrução. Assim, suas palavras VLIW podem ter até quatro instruções. Outras duas unidades possuem apenas duas fatias, limitando o tamanho máximo do pacote VLIW em duas instruções. Todas essas unidades formam um *cluster*.

O processador MAJC foi implementado em *hardware* em (KOWALCZYK et al., 2001b) (SUDHARSANAN, 2000) (SUDHARSANAN, 2000b). Ele é chamado de MAJC-5200, trabalha a 500Mhz e possui duas unidades de processadores. O núcleo de cada processador tem quatro fatias de instrução, e cada pacote VLIW pode conter até quatro instruções. Além do mais cada núcleo possui 224 registradores. Como poderia se esperar, a arquitetura precisou de um grande número de transistores, 13 milhões em tecnologia .22µm, para ser implementado. O diagrama de blocos do MAJC-5200 pode ser observado na Figura 2.17.

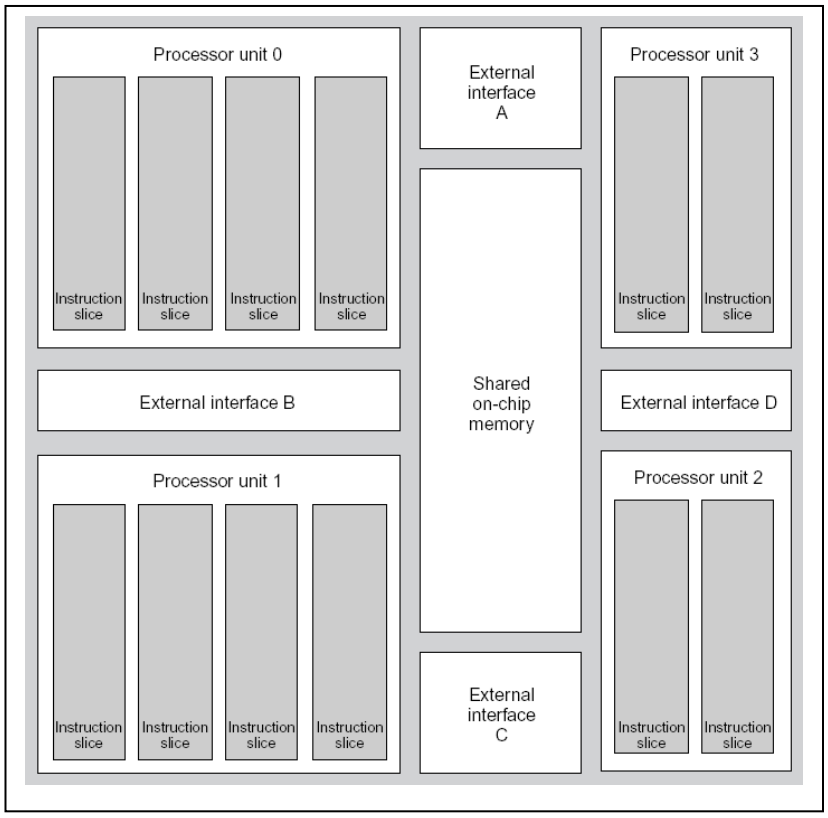

Figura 2.16 : Exemplo de uma possível implementação de uma arquitetura MAJC (TREMBLAY; CHAN; CHAUDHRY, 2000)

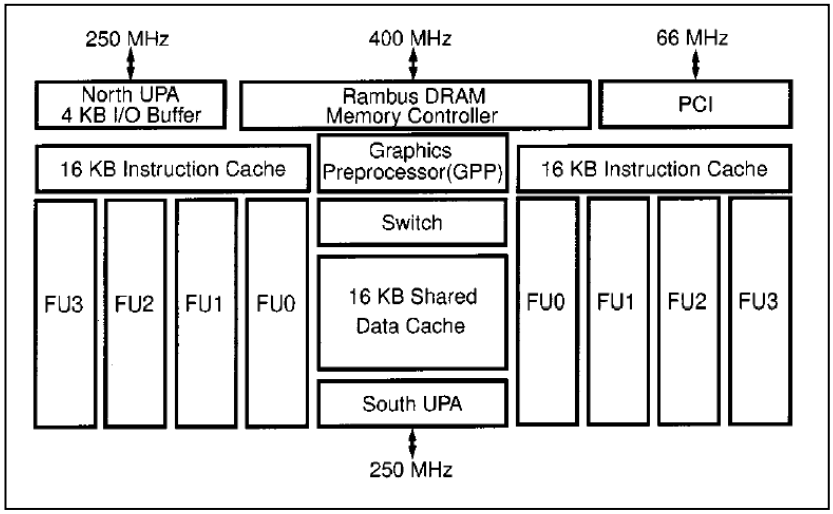

Figura 2.17 : Diagrama de blocos do MAJC-5200 (TREMBLAY; CHAN; CHAUDHRY, 2000)

2.9 Femtojava

O processador Femtojava (ITO; CARRO; JACOBI, 2001) foi criado com restrições de área e potência visando especificamente sistemas embarcados. Na realidade, o processador Femtojava é o resultado de uma metodologia adotada para a geração semi-automática de um sistema embarcado a partir de uma descrição Java. O Femtojava 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 2.1 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.

A organização de memória é baseada na alocação de *frames* como manda a especificação da linguagem Java e pode ser observada na Figura 2.18a (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 2.18c. Comparado com outros processadores como picoJava (Figura 2.18b), 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.

Tabela 2.1 : Instruções suportadas pelo processador Femtojava (ITO; CARRO; JACOBI, 2001)

Tipo de instrução	
Aritméticas e lógicas	iadd, isub, imul, ineg, ishr, ishl, iushr, iand, ior, and ixor
Controle de fluxo	goto, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt,

	if_icmpge, if_icmpgt, if_icmple, return, ireturn, invokestatic
Pilha	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, swap
Load/Store	iload, iload_0, iload_1, iload_2,iload_3, istore, istore_0, istore_1, istore_2, istore_3
Vetor	iaload, baload, caload, daload, iastore, bastore, castore, sastore, arraylength
Estendidas	Load_idx, store_idx, sleep
Outras	Nop, iinc, getstatic, putstatic

Além do mais, o Femtojava implementa o sistema de E/S mapeado em memória, como pode ser observado na Figura 2.19. Na mesma figura, observa-se também onde se encontra o vetor de interrupções.

Outras características do Femtojava são o conjunto reduzido de instruções, arquitetura *Harvard*, pequeno tamanho e facilidade de inserção e remoção de instruções. Sua microarquitetura pode ser observada na Figura 2.20. Somando-se a isso, possui algumas características interessantes, como o tamanho da máquina de controle ser diretamente proporcional ao número de instruções utilizadas. Isto só é possível porque o Femtojava é gerado a partir de um ambiente de CAD (*Computer Aided Design*), chamado Sashimi (ITO; CARRO; JACOBI, 2001).

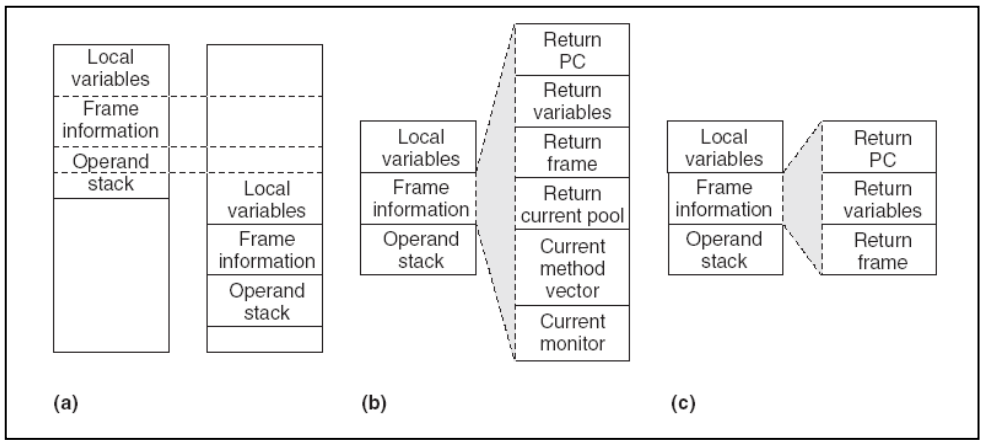

Figura 2.18 : Funcionamento da memória no processador Femtojava (ITO; CARRO; JACOBI, 2001)

Usando o Sashimi, o projetista pode modelar, simular e construir uma implementação do sistema diretamente em Java. O sistema disponibiliza bibliotecas para a simulação e permite um mapeamento direto das classes usadas por simulação para o código real da implementação final. Estas classes pré-definidas também cobrem todos os detalhes requeridos pela interface do micro controlador com o mundo externo (programação do mecanismo de interrupções, comunicação com o LCD, e comunicação com teclado). A partir do código gerado pelo compilador Java feito pelo projetista, uma versão ASIP (*Application Specific Instruction set Processor*) do Femtojava, onde sua unidade de controle possui apenas as instruções usadas por aquele programa em específico, é gerada. O fluxo completo do projeto pode ser observado na Figura 2.21.

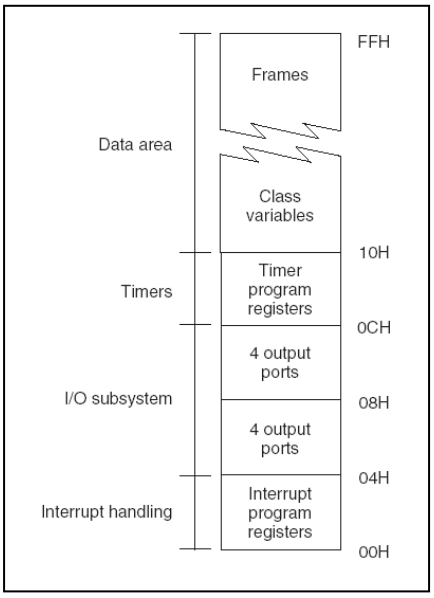

Figura 2.19 : E/S mapeada em memória e vetor de interrupções no processador Femtojava (ITO; CARRO; JACOBI, 2001)

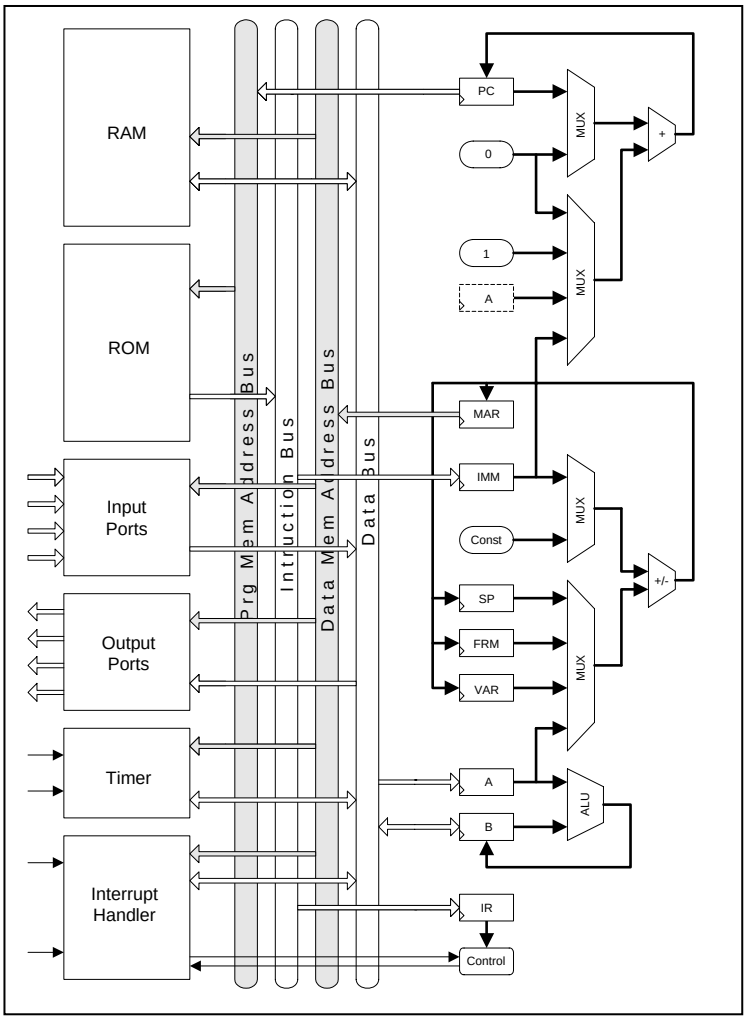

Figura 2.20 : A microarquitetura do processador Femtojava (ITO; CARRO; JACOBI, 2001)

As novas versões projetadas do Femtojava foram baseadas nesta versão apresentada, implementando, consequentemente, o mesmo conjunto de instruções. Além do mais, teve-se o cuidado de projetar estas novas versões de forma que elas sejam facilmente parametrizáveis em uma futura expansão da ferramenta Sashimi.

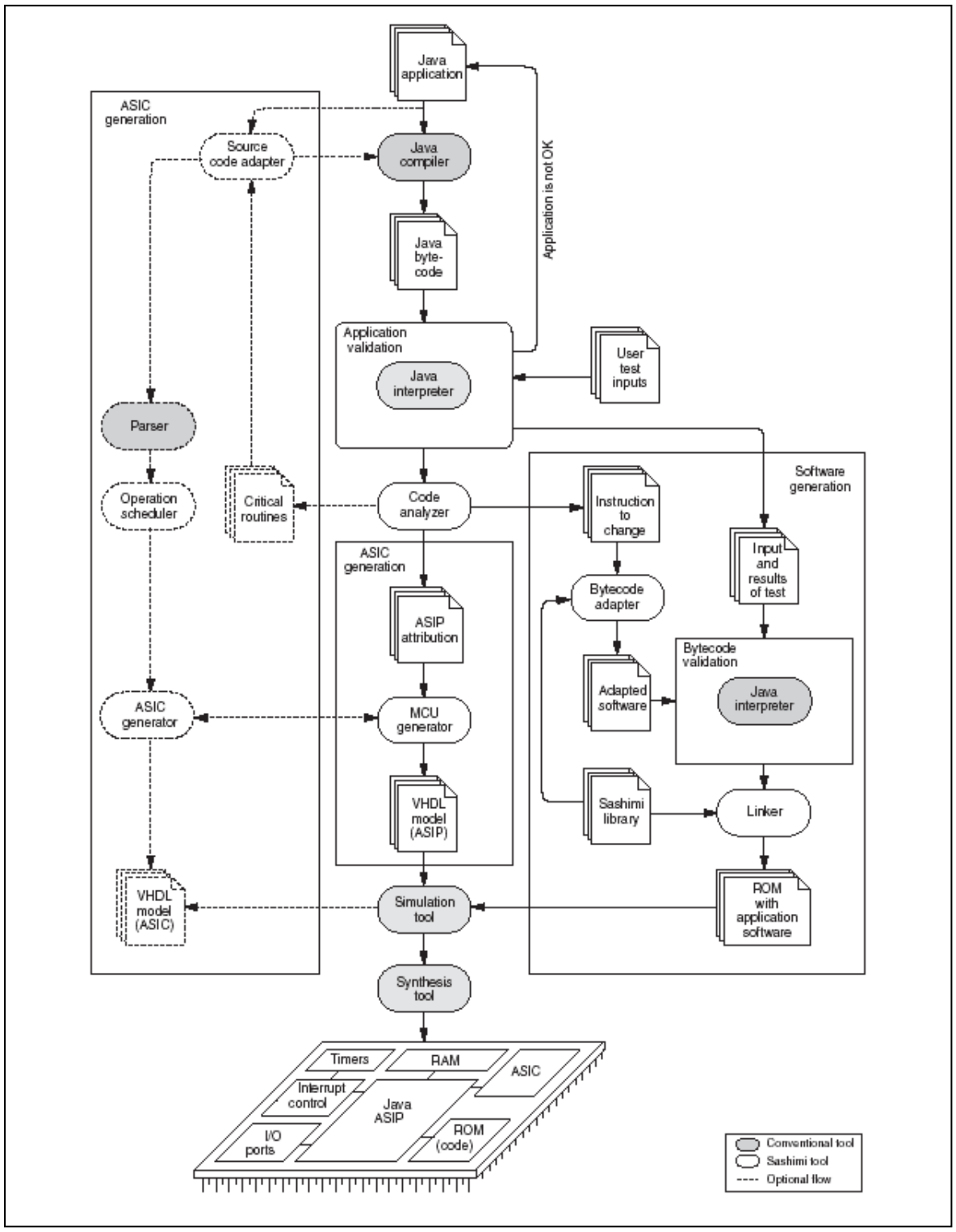

Figura 2.21 : O fluxo de projeto automatizado da ferramenta Sashimi para a geração ASIP do Femtojava

2.10 Outros processadores VLIW

Além do processador MAJC, já citado anteriormente, outros processadores VLIW já foram desenvolvidos. Em (NAKAMURA; SAKAI; AE, 1995), foi proposto um processador de pilha (com um conjunto de instruções próprio) baseado em VLIW para aplicações multimídia de tempo real, que pode manipular até 4 instruções em paralelo. Para cada instrução, há uma pilha de operações separada e um gerenciador próprio.

Além disso, outros processadores voltados especificamente para DSP foram desenvolvidos, como o processador Viper (GRAY et al., 1993), Fujitsu FR500 (SUGA; MATSUNAMI, 2000) e o Texas TMS320C6x (SESHAN, 1998) (que pode ser observado na Figura 2.22), todos baseados em instruções do tipo RISC.

Todavia, nenhum destes exemplos citados de processadores VLIW executam diretamente *bytecodes* Java. Como conseqüência, eles não exploram as vantagens e particularidades de Java na busca de paralelismo na aplicação, e de sua posterior execução. Então, torna-se bastante válida a exploração deste espaço de projeto, como será demonstrado no capítulo 6.

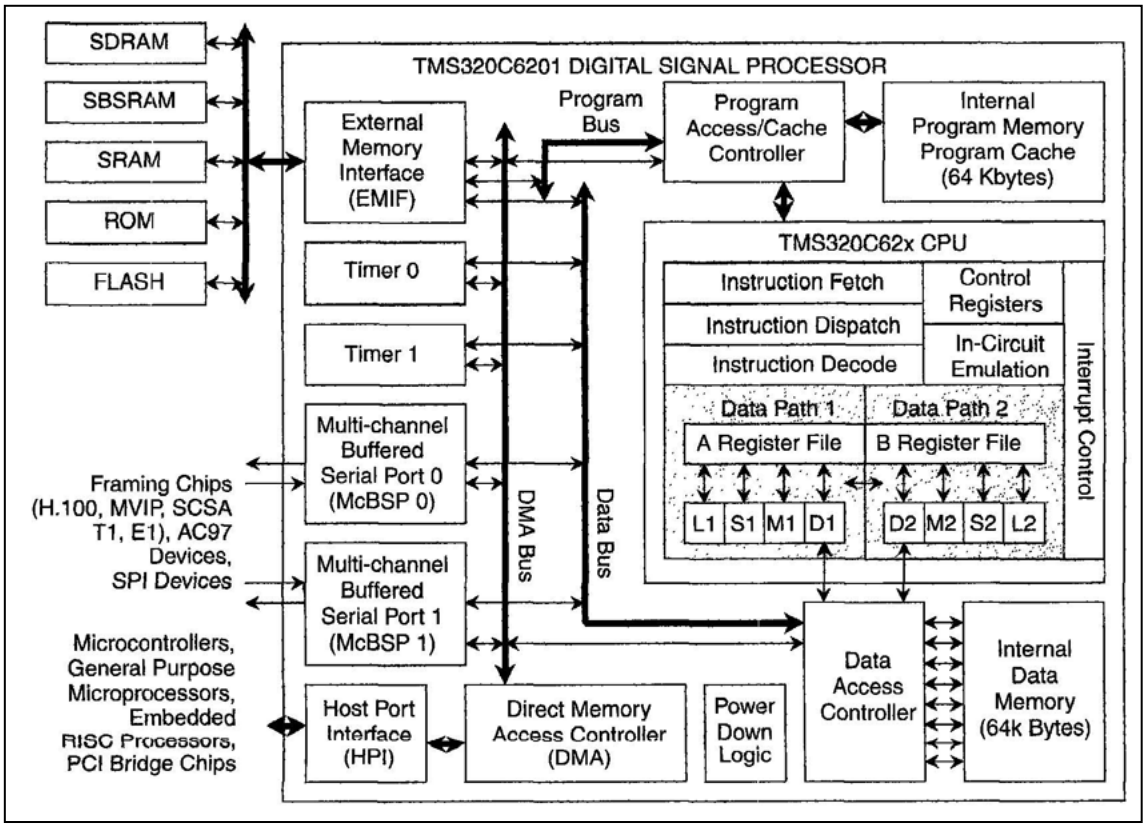

Figura 2.22 : Detalhes do processador Texas TMS320C6x (SESHAN, 1998)

3 CACO-PS

Em sistemas embarcados a avaliação da potência em fases iniciais é fundamental para a busca de soluções dentro do espaço de projeto. Nestes sistemas, na maioria das vezes são conhecidas *a priori* quais as aplicações que serão executadas. Além do mais, devido ao conjunto restrito de dados de entrada – que geralmente se resumem a simples comandos de controle do dispositivo – é possível analisar estas aplicações utilizando-se o pior caso da entrada de dados, isto é, aquele conjunto de dados que resultará em um maior número de instruções executadas da aplicação.

Para investigar várias soluções, e dentre destas escolher a mais satisfatória, uma ferramenta em um alto nível de abstração é necessária para ser utilizada no início do ciclo de projeto. O CACO-PS (*Cycle-Accurate COnfigurable Power Simulator*) (BECK et al., 2003) (BECK et al., 2003b) é um simulador que possui grande parte de seu código compilado, que calcula a potência baseado em ciclos de relógio (SIMUNIÉ; BENINI; DE MICHELI, 1999). Somando-se a isso, ele oferece a possibilidade da descrição estrutural de qualquer arquitetura (é de propósito geral). Além disso, é possível descrever a arquitetura em diferentes níveis de abstração, conforme o nível de detalhamento desejado pelo projetista. Trabalhos correlatos (TIWARI; MALIK; WOLFE, 1994) usam as instruções de uma aplicação como base para o cálculo de potência, entretanto não oferecem o nível de detalhamento desejado. Outras ferramentas de código compilado são geralmente restritas ao sistema que foi previamente programado, e simuladores que se baseiam em ciclo de relógio para o cálculo de potência exigem a descrição do sistema em RTL. A ferramenta CACO-PS não sofre de nenhuma destas restrições.

A primeira seção deste capítulo mostra uma breve revisão bibliográfica de vários simuladores de potência existentes atualmente, demonstrando as diferenças e virtudes do CACO-PS. A segunda seção mostra o funcionamento interno do simulador. As três seções seguintes mostram como descrever uma arquitetura, como construir uma biblioteca de componentes e o funcionamento do cálculo de potência. A sexta seção mostra os algoritmos programados em Java que foram utilizados para a validação do simulador e são utilizados durante todo este trabalho. Na seção 7 é mostrado um exemplo de uso do simulador, com a execução do conjunto de aplicações Java no Femtojava Multiciclo. Finalmente na última seção há um breve comentário sobre a versão de testes (injeção de falhas nas arquiteturas descritas) do simulador.

3.1 Trabalhos Anteriores

Vários estudos já foram feitos para se fazer a estimativa de potência logo ao início do ciclo de projeto, em um nível elevado de abstração. Em (TIWARI; MALIK;

WOLFE, 1994), a técnica para fazer a estimativa do gasto de potência é baseada na média da potência dissipada por cada instrução do processador, média esta obtida através de simulações. Outras técnicas (DALAL; RAVIKUMAR; 2001) (CHOI; CHATTERJEE, 2001) (CHEN et al., 2001), fazem também uma simulação baseada em nível das instruções, mas considerando modelos específicos de consumo de potência para componentes arquiteturais (unidades funcionais, conjunto de registradores, barramentos, memórias, unidades de controle) que são afetados por cada instrução, de acordo com a atividade de chaveamento (mudança de valor) nas suas entradas e também de suas propriedades físicas. Além disso, outras técnicas combinam tanto o método de estimativa baseado em nível das instruções quanto no método baseado em componentes (GIVARGIS; VAHID; HENKEL, 2001) (STITT et al., 2000).

Em (SIMUNIÉ; BENINI; DE MICHELI, 1999) é apresentado um simulador de código compilado que estima a potência dissipada em nível dos ciclos do processador. Esta ferramenta simula especificamente um sistema com processador ARM e o divide em componentes (processador, memória, cache L2, interconexões e outros componentes do sistema), onde cada componente possui um modelo de gasto de energia, que varia conforme os dados transitados ciclo por ciclo. Já em (LANDMAN; RABAEY, 1996) são apresentadas técnicas de análise de potência em RTL, ciclo a ciclo, para a unidade operativa, memória, unidade de controle e interconexões. Também em RTL, ciclo a ciclo, (LIU; SVENSSON, 1994) apresenta técnicas para o cálculo de consumo de potência, mas direcionados para SoC. Técnicas que fazem o cálculo de potência dissipada para partes específicas do processador também já foram estudadas: em (WILTON; JOUPPI, 1996) foi desenvolvida a ferramenta chamada CACTI, um modelo avançado para mensurar o consumo de potência dos acessos na *cache* e baseado em ciclos, que usa os dados de resistência e capacitância derivados da tecnologia utilizada e de dados de construção da *cache*.

Os simuladores que fazem a estimativa da potência dissipada que usam as instruções como base de cálculo têm como principal vantagem sobre aqueles que usam ciclos a maior velocidade de execução. Todavia, simuladores que usam instruções geralmente apresentam uma maior margem de erro na estimativa de potência, já que o nível de detalhamento é menor. O maior problema do uso desta técnica é a não consideração de fatores que podem levar a uma grande diferença final tanto em relação ao consumo de potência quanto no tempo de execução: paradas no *pipeline* devido à *hazards* de controle ou dependência de dados; falha de dados na *cache*, TLB ou tabelas de predição de desvio, entre outros.

Em ferramentas que utilizam código compilado, fica-se restrito à simulação em apenas uma arquitetura: aquela que foi modelada e previamente programada. Já simuladores que tomam por base os ciclos de relógio do sistema para fazer o cálculo da potência dissipada necessitam de uma complexa descrição da arquitetura, em um baixo nível de abstração, geralmente em RTL.

Neste trabalho, o simulador construído possui partes da descrição do sistema em código compilado, como será explicado adiante, e toma por base ciclos de relógio para fazer a estimativa de potência gasta em um sistema. O mesmo utiliza técnicas para a descrição estrutural e comportamental de uma arquitetura em uma sintaxe simples e fácil de ser utilizada, não ficando restrito a apenas uma arquitetura previamente programada: por isso é chamado de configurável. Além do mais, é possível definir componentes arquiteturais em diferentes níveis de abstração (desde transistores até processadores inteiros), livrando o projetista da tarefa de descrever em RTL todo o sistema como normalmente acontece em simuladores que manipulam ciclos de relógio.

Como se pode definir o nível de abstração destes componentes, o projetista, dependendo do detalhamento para análise desejado, pode utilizar diferentes níveis de abstração para diferentes partes do sistema. Além disso, o simulador pode modelar problemas como paradas no *pipeline* e falha de dados na *cache* já que toma por base ciclos do relógio para o cálculo.

Uma das técnicas que pode se beneficiar do uso deste simulador no início do ciclo de projeto, para futuramente se alcançar uma economia na potência dissipada, é o uso de bibliotecas de *software*. Certas rotinas diferentes, mas que desempenham a mesma função, devido ao conjunto de instruções utilizadas, quantidade de acesso à memória, eficiência e número de instruções, podem consumir um valor diferente de potência no circuito. Às vezes há uma troca: rotinas com mais instruções, mas que têm uma menor quantidade de acessos à memória, consomem menos potência. Troca-se área da memória de instruções por potência consumida. Um exemplo típico são os algoritmos de ordenação: *bubblesort*, *selectsort* e *quicksort*, que consomem diferentes quantidades de potência assim como ocupam tamanhos diferentes na memória. Pressupondo-se que uma aplicação é um conjunto de rotinas, pode-se escolher um determinado conjunto com o objetivo de diminuir a potência consumida por todo o sistema, seja trocando a potência por área, seja escolhendo algoritmos mais eficientes (REYNERI et al., 2001) (NANDI, 2001) (GIVARGIS; VAHID; HENKEL, 2001) (MATTOS et al., 2004).

3.2 O funcionamento do Simulador

O simulador CACO-PS foi escrito totalmente na linguagem C, tornando possível a compilação em diferentes sistemas operacionais. A escolha em se projetar o simulador desta forma foi feita por dois motivos: independência de plataforma através da recompilação do código e velocidade para simulação, já que a ferramenta sempre é executada diretamente no código nativo da máquina. A ferramenta já foi compilada e testada nos sistemas Red Hat Linux 7.0, Sun OS 5.7, utilizando-se o compilador *GCC*, e nos sistemas operacionais Windows 98 e Windows XP utilizando-se do compilador Borland C++ Builder 5.0 e *GCC*.

Basicamente, três arquivos são necessários para o funcionamento do CACO-PS:

- O que descreve, através de uma sintaxe própria e estruturada, a arquitetura desejada;
- O que descreve os componentes funcionais, que serão instanciados pela descrição da arquitetura;
- O arquivo que agrega uma função de cálculo de potência para cada componente.

Opcionalmente, podem-se indicar arquivos de imagem da memória de dados e memória de programa (ou instruções), no formato .MIF, que é o formato de memória utilizado pelos programas de VHDL da Altera (Altera 2004). Este arquivo basicamente é formado por um conjunto de linhas, onde em cada há um endereço de memória e depois seu valor. Um exemplo deste arquivo pode ser observado no Anexo II.

Além do mais, apesar de ser um simulador de propósito geral, algumas características específicas foram incluídas no simulador para trabalhar particularmente com o processador Femtojava. Uma delas é a opção de carregar um arquivo que contém todas as instruções do Femtojava (cada qual com o seu *opcode*, seus *microopcodes* para cada estado, e seu nome). A Figura 3.1 ilustra o funcionamento do simulador:

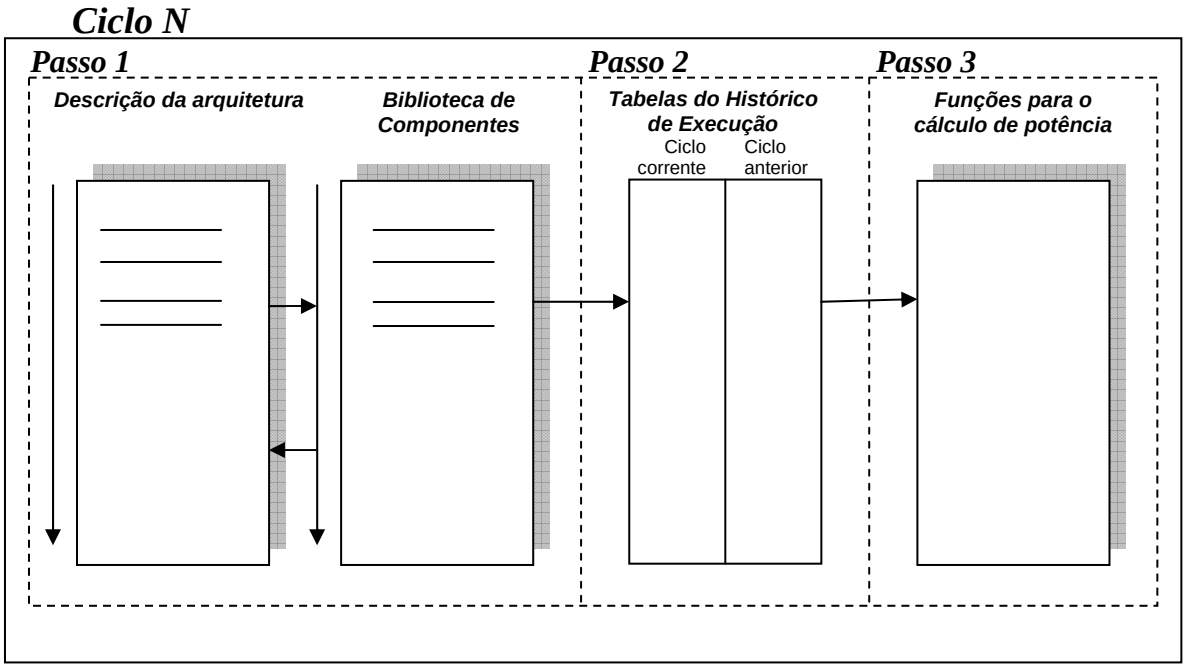

Figura 3.1 : Funcionamento do simulador

Em um determinado ciclo N, o arquivo de descrição da arquitetura é lido seqüencialmente. Cada linha é dividida em sinais de entrada, componente arquitetural e sinais de saída: o componente recebe os sinais de entrada, desempenha sua função e retorna o valor para um ou mais sinais de saída. Com um conjunto de linhas seguindo esta sintaxe é possível descrever qualquer arquitetura desejada. A terceira seção detalha esta sintaxe de descrição da arquitetura.

O passo 1, demonstrado na Figura 3.1, representa a leitura da descrição da arquitetura e a sua execução. A cada linha os valores de entrada são enviados para um componente. A função que este componente realiza está descrita na biblioteca de componentes. O resultado é gravado nos sinais de saída (que por sua vez serão usados como sinais de entrada para outros componentes) e a próxima linha é lida, até ser encontrado o final da descrição arquitetural.

Entretanto, se a simulação fosse implementada simplesmente desta forma, haveria uma dependência na ordem da declaração dos sinais que estão sendo usados na descrição da arquitetura. Se um sinal serve como entrada para algum componente, mas após isso ele serve de saída para algum outro componente (isto é, seu valor é atualizado), o primeiro componente receberia o valor errado deste sinal. Circuitos realimentados, como dois registradores que possuem suas saídas ligadas em uma ULA, com o resultado desta sendo gravado novamente no primeiro registrador (Figura 3.2), não seriam possíveis de serem implementados.

Para contornar este problema, o simulador usa a seguinte tática: Toda a descrição da arquitetura é executada pelo menos duas vezes. Depois de executada a primeira vez, todos os sinais terão algum valor. A simulação é executada novamente desde o principio utilizando os sinais encontrados na primeira execução. Caso houver algum valor discordante entre os sinais, executa-se desde o início a arquitetura novamente. Os valores de todos os sinais encontrados desta execução são comparados com os valores anteriores. Caso ainda houver algum sinal cujo valor seja diferente, repete-se a operação

até todos os sinais não tenham variado de uma execução para outra, isto é, até que todos os sinais do circuito tenham convergido. Só quando o circuito se estabiliza, o simulador passa para o próximo ciclo. Pode-se imaginar este efeito como o efeito de propagação de sinais durante um período de relógio. Sempre um sistema é projetado para que este período seja grande o suficiente para que os sinais se estabilizem (MARCON, 1992). Ressalta-se que o simulador possui um mecanismo de detecção para circuitos que ficam oscilando infinitamente.

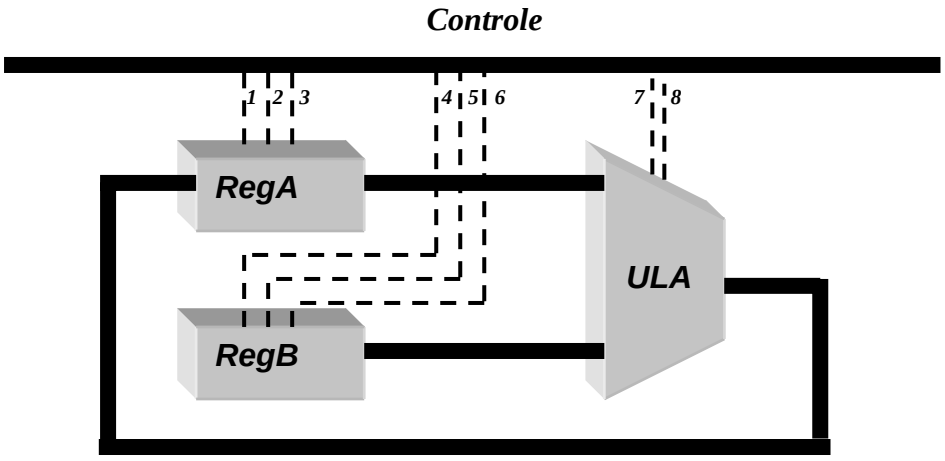

Figura 3.2 : Exemplo de uma pequena arquitetura a ser simulada

Quando os sinais estão estabilizados, eles são guardados em um histórico de execução. Este histórico contém, para cada componente instanciado, os valores dos sinais de entrada e de saída. Esta operação corresponde ao passo 2.

A partir deste histórico (que também guarda os valores dos sinais do ciclo anterior), a potência consumida naquele ciclo é calculada: para cada componente, há uma função de cálculo de potência correspondente (demonstrada com detalhes na seção 3.5 e no Anexo I). Esta função recebe como parâmetros os valores de entrada do ciclo atual e anterior do respectivo componente. Através destes valores, é possível saber a taxa de chaveamento dos *bits* de entrada para qualquer componente naquele ciclo. A partir desta taxa, a função calcula a quantidade de potência gasta por aquela instância do componente naquele ciclo. A potência gasta por cada componente instanciado é somada, resultando na potência total dissipada durante um ciclo. Somando-se este valor ao dos ciclos restantes, tem-se em mãos a energia total consumida pelo sistema na execução de determinada aplicação.

Os componentes podem ser, por exemplo, multiplexadores, portas lógicas, unidades aritméticas e lógicas, transistores ou até processadores inteiros. O nível de abstração dos componentes utilizados é definido pelo usuário. Quando um componente não se encontra na biblioteca, o usuário poderá adicioná-lo descrevendo-o na linguagem C. Sendo assim, a reutilização de componentes para a descrição de novas arquiteturas torna-se possível. O simulador oferece uma série de facilidades e macros para a construção destes componentes.

Nas próximas três seções os três arquivos essenciais para a execução e funcionamento do simulador são demonstrados com detalhes: a descrição da arquitetura, a biblioteca de componentes e o conjunto de funções de cálculo de potência para cada componente.

3.3 Descrevendo uma arquitetura no simulador

Nesta seção é demonstrado como descrever uma arquitetura para ser utilizada no simulador. A arquitetura de um sistema é descrita em um arquivo no formato texto, que possui um conjunto de linhas que seguem a seguinte sintaxe:

Entrada1, Entrada2, ... ,EntradaN -> componente id (X,Y,...,Z:controle) -> Saida1, ..., saídaN

Onde Entrada1, Entrada2, ... são os sinais (conjunto de *bits*) que servem de entrada para um componente. Como já citado anteriormente, os componentes podem desempenhar qualquer função em qualquer nível de abstração, desde um transistor até um processador inteiro. Para cada componente, é preciso dar um identificador id, já que um componente pode ser instanciado mais de uma vez pelo sistema, e este identificador serve para diferenciar uma instância de outra. Exemplificando, pode-se definir um componente registrador na biblioteca de componentes. Depois, pode-se usar este componente várias vezes. Para cada instância do componente registrador o projetista dá diferentes nomes, isto é, diferentes identificadores.

Cada componente é controlado por *bits* de um sinal de controle, caso for necessário. O resultado final, retornado por este componente, é propagado para as saídas. Estas saídas, por sua vez, podem ser utilizadas como entradas de outros componentes. Sendo assim, através de várias linhas como a exemplificada anteriormente, descreve-se um sistema completo.

Como exemplo, é apresentado o seguinte circuito: dois registradores cujas saídas servem de entrada para uma ULA, que pode somar, subtrair, fazer as operações lógicas AND e OR, conforme os seus sinais de controle. O resultado desta ULA novamente é guardado no primeiro registrador.

Como pode ser observado na Figura 3.2, o sinal de controle tem 8 *bits*. Os *bits* 0, 1 e 2 controlam o RegA. Os *bits* 3, 4 e 5 controlam o RegB. Os *bits* 6 e 7 controlam a ULA. Os *bits* que controlam os registradores servem para *enable, set* e *reset*. Os *bits* que controlam a ULA servem, nesta seqüência: 00 – para soma, 01 – para subtração, 10 – para a operação lógica AND, 11 – para a operação lógica OR. Salienta-se que a função de cada *bit* de controle foi definida quando os respectivos componentes (registrador e ULA) foram implementados na biblioteca de componentes (que será discutida na próxima seção). A Figura 3.3 mostra o exemplo demonstrado graficamente na Figura 3.2 na sintaxe utilizada pelo simulador.

```
regA_out,regB_out    -> ula exemplo (6,7:controle)    -> regA_in
regA_in              -> reg A (0,1,2:controle)        -> regA_out
regB_in              -> reg B (3,4,5:controle)        -> regB_out
```

Figura 3.3 : Na sintaxe do simulador, o exemplo apresentado na figura 3.2

Os componentes ula e reg se encontram definidos na biblioteca de componentes. Nota-se que há duas instâncias de reg. Elas são diferenciadas pelo identificador (A e B). Novamente ressalta-se que a ordem das linhas não altera de maneira alguma o resultado final.

Destaca-se que também é possível fazer o controle de um componente usando *bits* provenientes de mais de um sinal. Por exemplo, o registrador mostrado anteriormente terá o seu *bit* de *enable* controlado pelo *bit* 23 do sinal de controle, o *bit* de *set* sempre com o valor 1, e o *bit reset* controlado pelo *bit* 3 do sinal *poweron*:

```
sinalA                    -> reg exemplo3 (23,'1':controle;3:poweron)    -> sinalB
```

Todos os sinais foram declarados como inteiros na construção do simulador, e sendo assim, a largura total em *bits* dos sinais depende da arquitetura na qual o simulador está sendo executado. Nas arquiteturas Intel x86 e compatíveis atuais (a partir do processador 80386) o tamanho é de 32 *bits*. Caso uma arquitetura descrita necessitar de sinais maiores que 32 bits, mais de um sinal em paralelo precisa ser utilizado.

3.4 Construindo a biblioteca de componentes

3.4.1 Sintaxe e Definição

A biblioteca é composta por vários componentes que são declarados conforme a sintaxe mostrada na Figura 3.4:

```
1.   COMPONENTE("nome") {
2.   Comandos em C;
3.   ouput[n] =  resultados;
4.   }
```

Figura 3.4 : Esqueleto básico de um componente na biblioteca

Na Figura 3.4, o nome dado ao componente será utilizado quando for desejado instanciar este componente na descrição da arquitetura. Através de Comandos em C, na linha 2, o usuário pode definir a função desejada para aquele componente. Depois, este resultado é retornado (linha 3) e será propagado ao(s) sinal(is) de saída. Todavia, algumas facilidades precisam ser incorporadas no simulador para a descrição destes componentes. Quando o usuário declara o componente e a sua função, ele não sabe onde este componente será usado e com quais sinais de entrada e saída ele estará conectado. Sendo assim, há uma abstração dos sinais de entrada e saída para o usuário quando este estiver definindo o componente. Quando este componente é instanciado no arquivo de arquitetura, o simulador faz o trabalho de cola, unindo dinamicamente os valores das entradas da instância do componente aos sinais declarados como entradas para aquele componente. O mesmo acontece para os sinais de controle e o sinal de saída.

O simulador oferece vários sinais para a construção de componentes. São eles:

- O vetor input[] – input[0], input[1], ..., input[n], que representa os sinais de entrada.
- O vetor controle[] – controle[0], controle[1], ..., controle[n], que representa os *bits* do sinal de controle.
- O vetor output[] – output[0], output[1], ..., output[n], que representam o resultado retornado pelo componente (sinais de saída).

O número máximo dos vetores (representado como n), pode ser definido em no cabeçalho do simulador.

O componente que faz a operação lógica AND pode ser definido desta maneira, usando as variáveis oferecidas pelo simulador anteriormente:

```
COMPONENTE(“and”) {
output[0] =  input[0] & input[1];
}
```

Por exemplo, este componente pode ser instanciado, na descrição da arquitetura, como a seguir:

```
sinalA, sinalB                -> and exemplo4 ()                -> sinalC
```

Durante a execução, dinamicamente o simulador passa o valor do sinalA para o input[0], e do sinalB, para o input[1]. Após o componente efetuar sua função (no caso, um simples AND lógico), o resultado é armazenado em output[0], que dinamicamente é passado para o sinalC. É importante salientar que na descrição da arquitetura a ordem de declaração dos sinais *interessa*: o sinalA corresponde ao input[0] pois foi o *primeiro* a ser declarado.

Durante o decorrer de um ciclo, os sinais mudam seus valores até ficarem estáveis. Pensando nisso, o simulador também oferece recursos para quando o usuário quiser modelar um componente sensível à borda de relógio. A variável global clock_event indica, com o valor igual a 1, que houve uma transição de ciclo. Os vetores input_atual[] e controle_atual[] são os valores de entradas e de controle exatamente quando houve a transição de um ciclo para outro, como mostra a Figura 3.5. Sendo assim, torna-se possível modelar componentes sensíveis à borda de relógio, como registradores. Um exemplo deste uso pode ser encontrado no componente reg na Figura 3.6, que é um registrador de 16 *bits*.

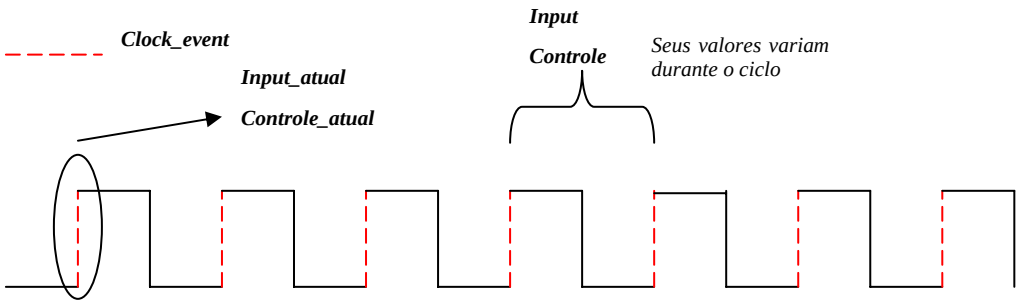

Figura 3.5 : Representação gráfica dos sinais oferecidos para a declaração de componente

```
1.  COMPONENTE("reg") {
2.  // enable = controle[0];
3.  // set = controle[1];
4.  // reset = controle[2];

5.  if (clock_event == 1) {
6.          if (controle_atual[0] == 1) {
7.                  ADICIONA_VALOR(input_atual[0]&MASK);
8.          }
9.  }

10. if (controle[1] == 0) ADICIONA_VALOR(0xffff);
11. if (controle[2] == 0) ADICIONA_VALOR(0x0000);
12. output[0] = (PEGA_VALOR & MASK);
13. }
```

Figura 3.6 : Exemplo de descrição de um componente na sintaxe do CACO-PS

O simulador ainda oferece duas macros não citadas anteriormente:

- ADICIONA_VALOR
- PEGA_VALOR

Estas macros são responsáveis por chamar funções que guardam um valor para cada instância do componente. Cada instância de um componente pode ter um valor qualquer guardado em uma estrutura interna do simulador, transparente ao projetista. Nota-se que, voltando ao exemplo dos registradores, se houver um registrador A e um registrador B, os valores guardados para cada um serão diferentes. Neste caso, a tabela interna é indexada pelo componente e pelo seu id.

No exemplo do registrador na Figura 3.6, foi definido que o primeiro *bit* de controle serviria para o *enable* do registrador, o segundo para *set* e o terceiro para *reset*. O primeiro IF (linha 5) verifica se houve uma transição de um ciclo para o outro. Caso positivo, se neste momento o sinal de *enable* estiver ligado, ele guarda o valor de input_atual[0], que é o valor que estava na entrada do registrador no exato momento da transição do ciclo (linhas 6 a 8).

Nota-se que os sinais de *reset* e *set* são assíncronos. Isto é, não são dependentes do sinal do relógio. A qualquer momento, quando um destes sinais for igual a zero (é utilizada a lógica inversa), os valores guardados serão 0xffff (todos *bits* em um, representado em hexadecimal) ou todos os *bits* em zero, para *set* e *reset*, respectivamente.

Finalmente, o resultado retornado para o sinal de saída será o valor gravado anteriormente para aquela instância do componente registrador.

Além do mais, há dois vetores, declarados como variáveis globais, chamados ROM e RAM. Estas variáveis podem ser usadas por qualquer componente, mas servem para facilitar a simulação da memória de dados e de instruções. Automaticamente os dados dos arquivos no formato .MIF passados como parâmetros são colocados nestes vetores.

Cada vez que o arquivo de componentes é modificado ou atualizado, ele precisa ser recompilado. É isto que torna possível um simulador que possui grande parte da descrição da arquitetura em código compilado ser genérico: ao invés de se programar um simulador diretamente para uma arquitetura em específico, como é feito geralmente, tem-se um conjunto de componentes que possuem a sua funcionalidade compilada em

código nativo da máquina. A descrição da arquitetura, por sua vez, é lida e guardada em memória. Na medida em que ela for executada, chama-se o código correspondente do componente desejado. Isto é, a descrição da arquitetura é interpretada, todavia, a funcionalidade dos componentes (que concentra a maior parte da computação) é compilada.

3.5 Definindo funções de potência

Considerando-se o inversor da Figura 3.7a, onde C_L é, para efeitos de simplificação, a capacitância de carga total do sistema, onde são contadas as capacitâncias de difusão interna das portas dos transistores, as capacitâncias de interconexão e a capacitância de *fan-out*. Quando houver uma transição de 0 para 1 na entrada, o capacitor C_L é carregado pelo transistor PMOS, e sua voltagem levanta de 0 para V_{DD} e uma certa quantidade de corrente é drenada da fonte de energia, como mostra a Figura 3.7b. Parte desta energia é dissipada no dispositivo PMOS, enquanto a restante é guardada no capacitância de carga (CL). Durante a transição de 1 para 0, esta capacitância é descarregada, e a energia que estava guardada é dissipada no transistor NMOS, como mostrado na Figura 3.7c. Esta é a fonte de dissipação da potência dinâmica. Assim, para cada componente, sabendo-se a taxa de chaveamento, isto é, a quantidade de transições de suas entradas, pode-se calcular a quantidade de potência dinâmica dissipada (RABAEY, 1996), dada pela fórmula:

$$\boldsymbol{P = C \,.\, f \,.\, Vdd^2} \qquad (1)$$

Nos experimentos realizados, a potência dinâmica foi calculada pela quantidade de capacitâncias de portas equivalentes que chavearam por ciclo para cada componente, tendo assim um valor proporcional ao total de potência dinâmica gasta pelo circuito.

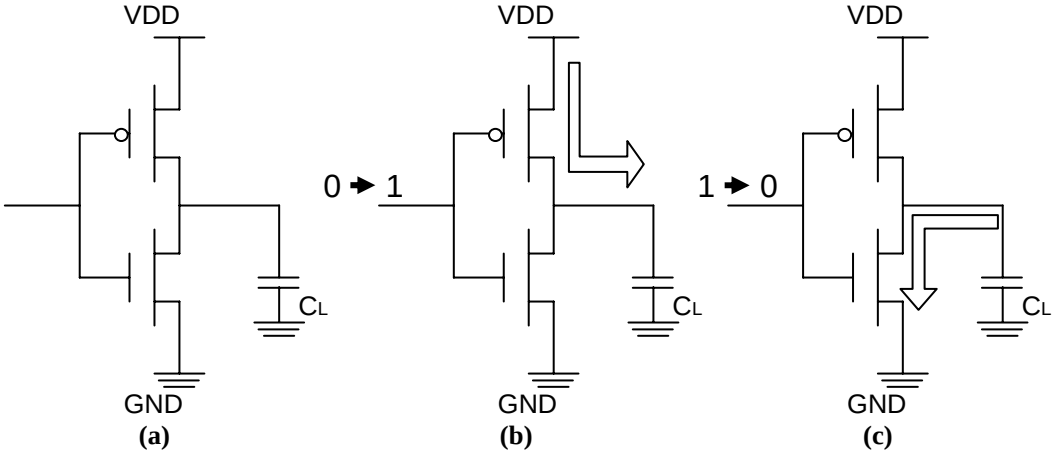

Figura 3.7 : Potência dinâmica dissipada em um inversor CMOS

Entretanto, apesar de oferecer funcionalidades adicionais para o cálculo de potência dinâmica, pode-se computar outros tipos de potência – como a estática – para cada componente, fornecendo valores fixos para cada uma delas, que será computado para aquele componente a cada ciclo. Entretato, neste trabalho, apenas a potência dinâmica foi considerada, visto que atualmente esta ainda é responsável pela maioria do consumo total de potência do sistema (KIM et al., 2003).

Como já citado anteriormente, para cada componente há uma função (também descrita em linguagem C) de cálculo de potência correspondente. Este arquivo de descrição de potência funciona de forma bem semelhante ao arquivo de descrição de componentes.

As funções para o cálculo de potência de cada componente recebem como parâmetros os dados de entrada do ciclo corrente e do ciclo anterior. Através destes dados, é possível saber quantos e quais *bits* na entrada de determinado componente foram chaveados, isto é, mudaram seu valor do ciclo anterior em relação ao ciclo corrente. A partir deste dado, a função retorna um valor de potência correspondente àquele componente e a suas entradas. Para o projetista, funções especiais e macros para fazer o cálculo da potência consumida devido ao chaveamento das entradas de cada componente são oferecidas, dentre outras facilidades. Detalhes de como implementar uma função de cálculo de potência para um componente podem ser encontrados no Anexo I.

3.6 Os programas Java para *benchmark*

Cinco diferentes tipos de algoritmos foram implementados e simulados nas arquiteturas do Femtojava descritas neste trabalho. Cálculo do seno, bastante utilizado em bibliotecas aritméticas; ordenação e busca, usados em agendas; IMDCT (*Inverse Modified Discrete Cosine Transformation*), uma importante parte do algoritmo de descompressão de MP3; e uma biblioteca que emula somas de ponto flutuante, já que o processador Femtojava não possui esta unidade a fim de economizar área.

Dois diferentes algoritmos de busca são utilizados. O primeiro faz uma pesquisa seqüencial em um vetor ordenado. O segundo faz uma pesquisa binária neste mesmo vetor. Os algoritmos de ordenação arranjam um conjunto de 10 números os colocando em ordem crescente. Quatro diferentes tipos de ordenação são feitos: usando as técnicas *bubblesort*, *selectsort*, *quicksort* e *insertsort*. Já a emulação de soma de pontos flutuantes faz 20 somas de dois números em ponto flutuante e coloca os resultados em um vetor na memória. Finalmente, o algoritmo de cálculo de seno utiliza o método CORDIC (*Coordinate Rotation Digital Computer*) (VOLDER, 1959) para calcular o resultado. Três diferentes versões de laços desenrolados (*loop unrolled*) também foram feitas para a versão IMDCT.

3.7 Exemplo de implementação – Femtojava Multiciclo

A arquitetura do processador foi descrita utilizando-se a sintaxe já apresentada, e todos os componentes foram construídos e incluídos na biblioteca. As funções para cálculo de potência também foram desenvolvidas. O arquivo de descrição da arquitetura totalizou 75 linhas. 65 componentes foram criados. A descrição do Femtojava Multiciclo em CACO-PS pode ser observada no Anexo III.

Os componentes implementados possuem um nível médio de abstração: não estão implementados usando transistores, por exemplo, mas estão implementados em um nível suficiente para o projetista acompanhar os principais sinais no *datapath*. Os principais componentes são: multiplexadores, a ALU (que incorpora as funções básicas aritiméticas e lógicas, um deslocador e um multiplicador), registradores e a parte responsável pela decodificação do Femtojava. Para o cálculo de potência de unidades que estão em um nível de abstração um pouco maiores e que nem sempre são

simétricas, como o decodificador de instruções, foi utilizado como base para os gastos em termos de chaveamento um esquema em portas lógicas gerado pelo programa Leonardo Spectrum (MENTOR, 2004), dado o VHDL correspondente daquele componente.

A aplicação utilizada para a validação da descrição da arquitetura demonstrada consiste em dois processos sendo executados simultaneamente sobre o mesmo processador. Estes processos são gerenciados por um escalonador específico para esta arquitetura. Os dois processos que são escalonados são algoritmos clássicos de ordenação: um processo executa o algoritmo *bubblesort*, e outro, o algoritmo *selectsort*. Cada processo ordena um vetor de dez elementos. A cada período de 750 ciclos, o processo escalonador assume o controle do processador e faz o chaveamento de contexto entre os outros dois processos.

Esta aplicação foi assim definida visando chegar o mais perto possível de uma aplicação real, que geralmente possui vários processos sendo executados simultaneamente. Desta forma, o conjunto de três processos sendo executados sobre a arquitetura torna a validação dos resultados mais consistentes, visto que a aplicação é relativamente complexa. Além disso, também foram simuladas as aplicações de *bubblesort*, *quicksort* e *selectsort* separadamente, apenas para base de comparação.

Os resultados são os mostrados na Tabela 3.1, obtidos em um computador PC AMD Athlon 2.2+ com 512MB de memória RAM, com o sistema operacional Fedora Linux Yarrow:

Tabela 3.1 : Execução de alguns algoritmos no processador Femtojava Multiciclo descrito na sintaxe do CACO-PS

Aplicação	***nº de Ciclos***	***Potência***				***Tamanho da memória de programa***	***Tempo de execução***
		RAM	**ROM**	**Núcleo**	**Total**		
Escalonador + BubbleSort + SelectSort	16625	58489000	1227510	30105716	89822226	278	5,53s
Bubblesort	6774	20930000	423890	12821962	34175851	88	2,32s
SelectSort	5335	20838000	358800	10115480	31312280	94	1,79s
QuickSort	3940	18446000	266340	7393279	26105619	129	1,31s

Na primeira coluna é listada a aplicação simulada na arquitetura descrita, e na segunda o número de ciclos consumidos por esta aplicação. Nas colunas seguintes é mostrada a potência dissipada no núcleo, RAM, ROM e total, em número de chaveamento de portas. Apesar de, por padrão, serem mostradas apenas a ROM, RAM e núcleo separadamente, facilmente o projetista pode estender esta visualização para partes específicas do circuito, conforme desejado. Depois, na penúltima coluna, o tamanho da memória de programa de cada aplicação, em *bytes*, é fornecido. Finalmente, é mostrado o tempo de execução que a aplicação demorou para ser simulada.

Os resultados da Tabela 3.1 mostram a utilidade do simulador na exploração de um amplo espaço de projeto no desenvolvimento de um SOC completo. O exemplo ilustra que diferentes algoritmos de ordenamento têm diferentes impactos de desempenho, potência e memória ocupada, e também demonstra o efeito da inclusão do escalonador nestes fatores.

O simulador oferece uma interface no formato texto, que pode ser facilmente manipulada pelo usuário. Possui opções como geração de *trace* do circuito executado,

execução utilizando *breakpoints*, execução passo a passo, visualização de cada sinal em separado, entre várias outras opções.

3.8 CACO-PS – versão para testes

O simulador foi estendido para suportar injeção de falhas do tipo *stuck-at* nas arquiteturas descritas usando sua sintaxe. O simulador, além de inserir as falhas, pode detectar, em locais especificados pelo usuário, se esta falha foi propagada ou não.

O sistema funciona da seguinte maneira: primeiramente todo o sistema é simulado, e ao final de sua execução, os valores corretos de todos os sinais e da memória (caso o usuário tenha especificado o uso de memória para um processador, por exemplo) são guardados. A partir daí, uma simulação inteira é feita novamente, mas desta vez, com o primeiro *bit* do primeiro sinal declarado sempre em zero (*stuck-at-0*). Este procedimento é feito para todos os *bits* de todos os sinais. Quando o último sinal do último *bit* é alcançado, repete-se todo o procedimento agora com *bits* em *stuck-at-1*.

Ao final de toda a simulação, o processador mostra em uma tabela se a falha de cada *bit* propagou ou não para a memória, ou para algum sinal em específico, escolhido pelo usuário. Por exemplo, no caso do Femtojava, o projetista poderia saber a porcentagem de falhas do tipo *stuck-at-0* e *stuck-at-1* que propagaram para o sig_alu_out, que é a saída da ULA principal do processador. Isto é muito útil particularmente para procurar melhores alternativas para o teste funcional, onde é usado o próprio conjunto de instruções do processador para alcançar uma boa cobertura de testes (HENTSCHKE et al., 2004).

Tanto o simulador quanto os exemplos citados anteriormente podem ser encontrados no anexo IV (CD-ROM), e também em:

http://www.inf.ufrgs.br/~caco/caco-ps

4 FEMTOJAVA *LOW-POWER*

Dentre todos os processadores citados no capítulo 2, em nenhum, com exceção do Femtojava Multiciclo, quando da construção de suas microarquiteturas, teve-se a preocupação de se aplicar técnicas para a redução do consumo de potência, assim como otimizações nos *bytecodes* Java com o mesmo propósito. Além disso, estas microarquiteturas não são totalmente parametrizáveis, muitas vezes possuindo recursos que a aplicação não exige, mostrando uma preocupação secundária com a área do sistema e a potência consumida.

Contudo, o processador Femtojava Multiciclo pode não ter um desempenho satisfatório para certas aplicações de sistemas embarcados, como descompactação de áudio e vídeo. Assim, para suprir este problema de performance, um processador com um *Pipeline* de 5 estágios, chamado de Femtojava *Low-Power* (BECK; CARRO, 2003) (GOMES; BECK; CARRO, 2004), foi desenvolvido. Além da melhora na velocidade de execução de programas Java, mostra-se que, através de otimizações de sua microarquitetura, como o uso da técnica de *forwarding* (HENNESSY; PATTERSON, 2003), os acessos para escrita no banco de registradores onde está localizada a pilha podem ser reduzidos em média de 70% em aplicações típicas de sistemas embarcados, pelo fato do processador ser baseado em uma máquina de pilha.

4.1 Microarquitetura do Femtojava *Low-Power*

Como já citado, a versão Femtojava *Low-Power* possui um *pipeline* de cinco estágios: busca de instruções, decodificação, busca de operandos, execução e escrita de resultados, como pode ser observado na Figura 4.1. Uma das principais características deste processador é a implementação da pilha de operandos (*operand stack*) e do depósito de variáveis locais do método em um banco de registradores, ao invés de usar a memória principal para estes propósitos, como é feito no Femtojava Multiciclo. Nas seguintes subseções, serão demonstrados os cinco estágios em detalhes.

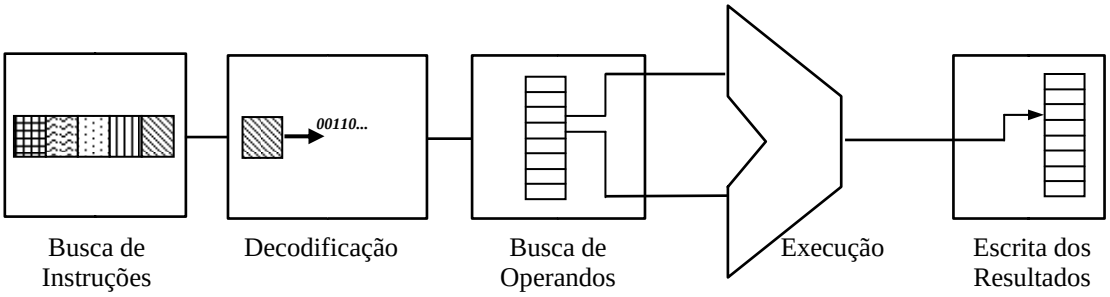

Figura 4.1 : Os cinco estágios do *pipeline* do Femtojava *Low-Power*

4.1.1 Estágio 1 – Busca de instruções

O primeiro estágio do *pipeline* é o de busca de instruções, representado na Figura 4.2. Este estágio faz a requisição de instruções da memória de programa através de uma palavra de 32 *bits*. Ele é basicamente composto por uma fila de instruções de 9 registradores de 1 byte de comprimento cada, um registrador para indicar a seqüência de instruções a ser buscada na memória (IMAR) e um somador para endereçar o próximo conjunto de instruções a ser buscado na seqüência. Se uma instrução em um endereço não seqüencial precisa ser carregada (no caso de um desvio ou invocação de método, por exemplo), um multiplexador carrega o novo valor do contador de programa (PC) no registrador IMAR.

O mecanismo pode fazer a busca antecipada de *bytecodes*, evitando que o *pipeline* fique parado na busca de instruções. O conjunto de 9 registradores na fila de instruções é o número mínimo para que o *pipeline* siga executando sequencialmente as instruções sem que haja necessidade de inserção de bolhas em seu interior. As instruções suportadas pelo Femtojava têm um tamanho variável, podendo ter nenhum, um ou dois operandos imediatos. Se um endereço não seqüencial for tomado (como em um desvio), é necessário limpar a fila de instruções. Nesta implementação, são necessários 3 ciclos de relógio para que, depois do desvio tomado, a nova instrução comece a ser executada. Se o tamanho da fila de instruções fosse maior, mais ciclos de relógio seriam necessários para enchê-la e, consequentemente, o IPC (Instruções Por Ciclo) do processador seria prejudicado.

Quando no mínimo 4 registradores da fila estão vazios, uma nova palavra de 32 *bits* vem da memória, apontada pelo IMAR. A primeira instrução da fila é mandada para o segundo estágio, de decodificação.

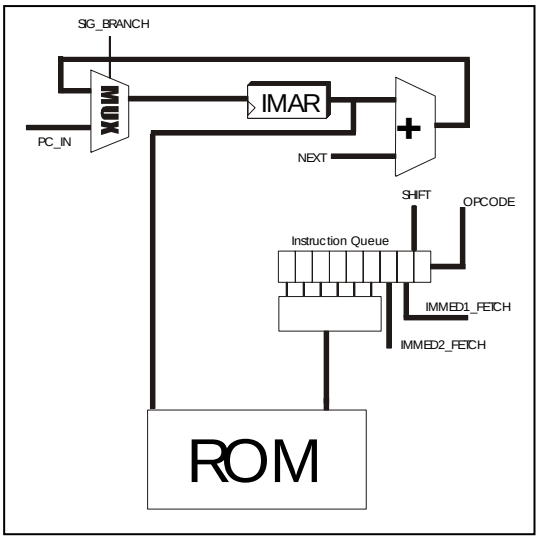

Figura 4.2 : Estrutura simplificada do estágio de busca de instruções do Femtojava *Low-Power*

4.1.2 Estágio 2 – Decodificação

Este estágio tem basicamente quatro funções: gerar a palavra de controle para a instrução corrente; informar o tamanho desta instrução para a fila de instruções, de modo que a próxima instrução do fluxo fique exatamente no primeiro lugar da fila, já que, como explicado anteriormente, o tamanho das instruções é variável; analisar a

dependência de dados das instruções; fazer a análise de *forwarding*, passando estas informações para os estágios seguintes. O diagrama de blocos deste estágio pode ser observado na Figura 4.3.

4.1.3 Estágio 3 – Busca de Operandos

É neste estágio onde os operandos são escolhidos e buscados para a execução da instrução. Ele é basicamente composto por um banco de registradores de duas portas de leitura. Neste banco podem ser feitas duas leituras independentes ou uma escrita a cada ciclo de relógio. O conjunto de *benchmarks* utilizado mostrou que a implementação de um tamanho de 32 registradores (ou 32 locais para a pilha) foi suficiente. Entretanto, considerando que o processador é do tipo ASIP, configurado para uma determinada aplicação, este tamanho pode ser definido *a priori* em estágios anteriores do ciclo de projeto.

A pilha de operandos e o repositório de variáveis locais do método estão localizados neste banco de registradores. A pilha e o repositório de variáveis são deixados juntos no mesmo banco de registradores para facilitar a chamada e retorno de métodos, tirando vantagem da especificação da JVM, onde cada método é localizado por um apontador de quadro (*frame*) na pilha, como já explicado anteriormente. Os dados de retorno do método, que também poderiam estar localizados neste banco, foram implementados na memória principal, visto que, comparando com a pilha de operandos e repositório de variáveis locais, eles são pouco usados, economizando-se assim em área do núcleo do processador.

Outros dois registradores, chamados de SP e VARS (implementados separadamente do banco de registradores), apontam para o topo da pilha de operandos e início do repositório de variáveis, respectivamente. Dependendo da instrução, um deles é usado como base para a busca de operandos. A Figura 4.4 ilustra este estágio. Como pode ser observado, multiplexadores adicionais são necessários, já que os operandos podem vir do banco de registradores, dos *bytecodes* (como dados imediatos), do estágio de execução ou do quinto estágio, sendo estes dois últimos através do mecanismo de *forwarding* que será explicado a seguir.

4.1.4 Estágio 4 – Execução

É neste estágio onde as instruções desempenham o seu papel. É composto de uma ULA capaz de executar as tarefas de adição e subtração, além das funções lógicas básicas, como operações de AND, OR e XOR. Um multiplicador, uma unidade de *load/store*, um deslocador (*shifter*) e uma unidade de desvio também se encontram neste estágio, mas em diferentes blocos, para facilitar a futura parametrização do processador. Desta maneira, o projetista é capaz de escolher quais unidades funcionais têm que ser incorporadas de acordo com as necessidades da aplicação. O estágio de execução é mostrado na Figura 4.5.

Nota-se que não há predição de desvio, com o objetivo de economizar área e pela facilidade de implementação. Todos os desvios são considerados previamente como não verdadeiros. Caso o desvio seja verdadeiro, não há uma perda muito grande - uma penalidade de apenas três ciclos é paga - já que o *pipeline* tem poucos estágios. Todavia, futuramente será implementado uma unidade de predição de desvio ou ao menos será oferecido ao projetista a possibilidade de escolha padrão (considerar um desvio sempre verdadeiro ou não verdadeiro) na geração da versão ASIP do processador, usando a análise prévia do código do programa.

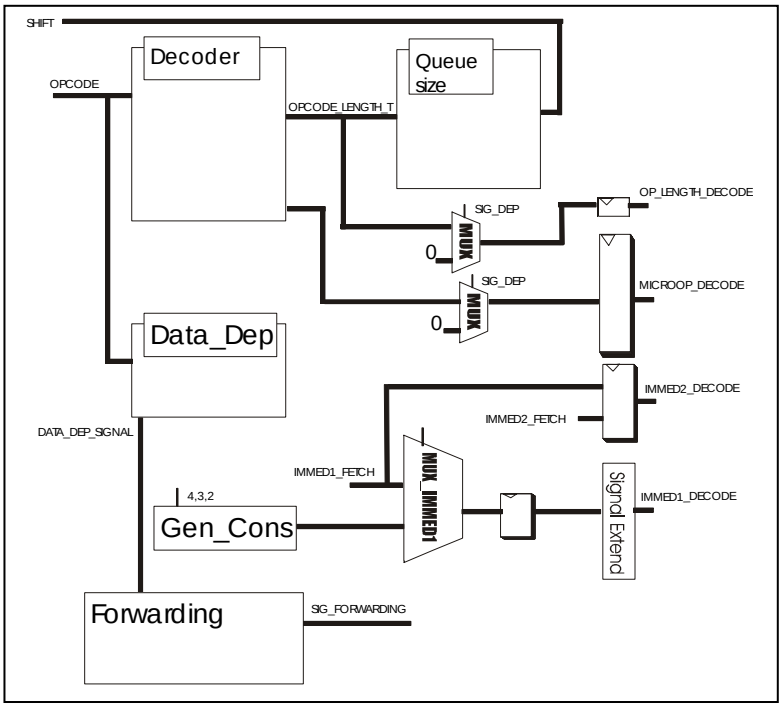

Figura 4.3 : O estágio de decodificação do Femtojava *Low-Power*

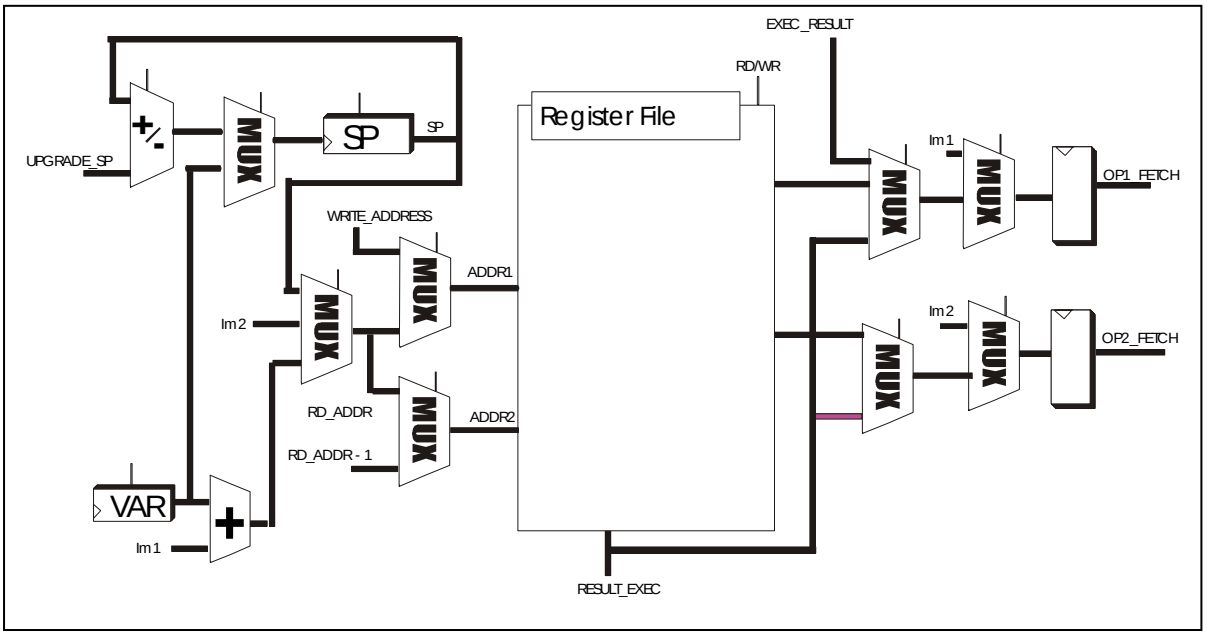

Figura 4.4 : O terceiro estágio do Femtojava *Low-Power*: Busca de operandos

4.1.5 Estágio 5 – Gravação dos resultados

O quinto estágio é responsável por salvar, se necessário, o resultado do estágio de execução no banco de registradores, usando SP ou VARS como base. Como o banco de registradores não pode ser lido e escrito simultaneamente para assegurar a conscitência dos dados, quando uma instrução no quinto estágio grava o seu resultado e uma outra instrução no terceiro estágio precisa buscar algum operando, uma bolha é

inserida no *pipeline*. A Figura 4.6 mostra a escolha do endereço e do valor a ser gravado no banco de registradores.

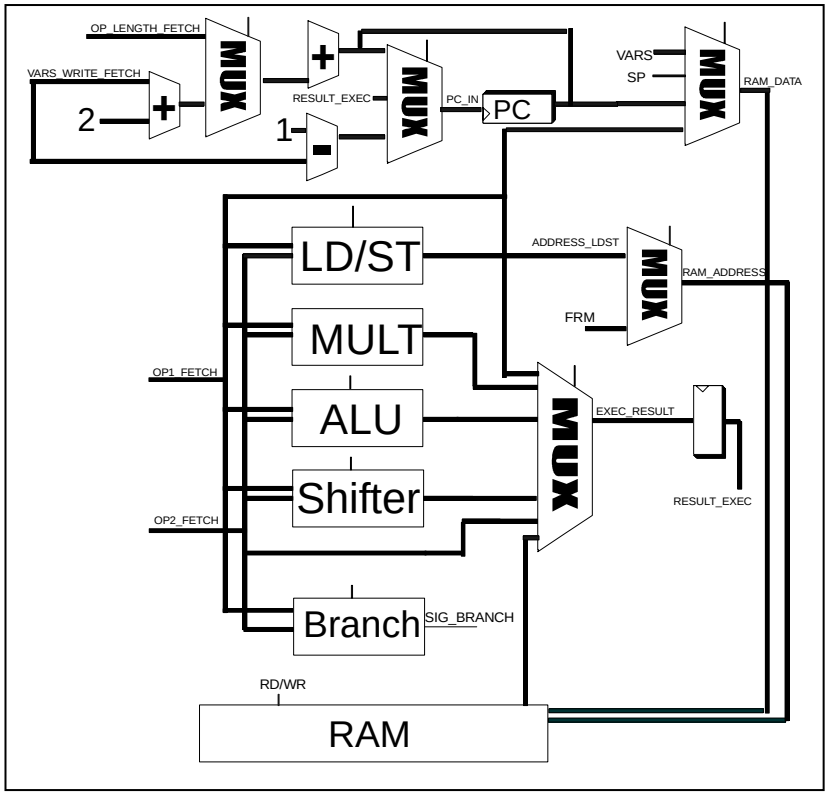

Figura 4.5 : O estágio de execução no Femtojava *Low-Power*

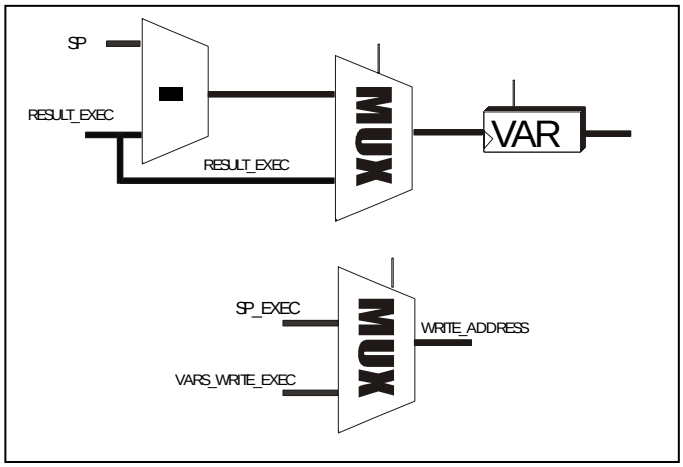

Figura 4.6 : A escolha do endereço e do valor para escrita no banco de registradores no último estágio

4.2 Análise de dependência de dados e *forwarding*

4.2.1 Forwarding em processadores RISC

Em processadores com vários estágios de *pipeline*, é muito comum acontecerem dependência de dados ou instruções. Estas dependências fazem com que instruções não possam ser executadas em paralelo (em um processador superescalar, por exemplo), ou ainda que seja necessário haver um intervalo maior na execução (em números de ciclos) entre duas instruções. Algumas dependências são consideradas falsas e podem ser resolvidas através de renomeação de registradores e reordenação de instruções, entre outras técnicas. Todavia, a dependência verdadeira do tipo RAW (*Read After Write*) não pode ser contornada de maneira alguma. Esta dependência ocorre quando uma instrução precisa usar um dado resultante da instrução que foi executada logo antes. Um exemplo típico é esta seqüência de instruções do tipo RISC:

```
ADD R1, R2, R3
SUB R4, R5, R6
MUL R7, R1, R4
```

Nota-se que há dependências do tipo RAW entre as instruções MUL e ADD e entre MUL e SUB, já que a instrução MUL vai utilizar os valores de R1 e R4, resultados das operações de ADD e SUB, respectivamente. Considerando um *pipeline* típico de um processador RISC, com cinco estágios: busca de instruções e decodificação; busca de operandos; execução; busca/escrita em memória; escrita dos resultados, estas três instruções seriam compostas neste *pipeline* como mostrado na Figura 4.7.

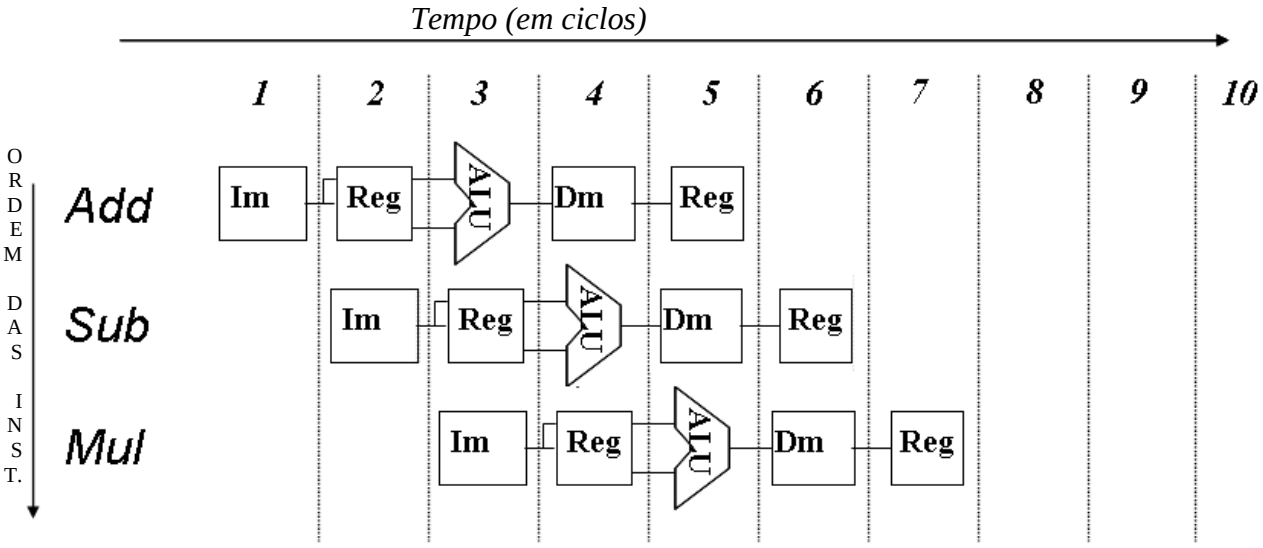

Figura 4.7 : Seqüência de instruções em um processador com *pipeline* típico RISC (HENNESSY; PATTERSON, 2003)

Entretanto, se as instruções fossem colocadas exatamente desta maneira no *pipeline*, um problema ocorreria. Quando a instrução MUL fosse fazer a sua busca de operandos, no ciclo 4, os valores que ela utilizaria, dos registradores R1 e R4, ainda não teriam sido escritos, já que o valor de R1 começaria a ser escrito no ciclo 5, por ADD, e ficaria disponível para leitura no ciclo 6, e o registrador R4 começaria a ser escrito no ciclo 6 por SUB e ficaria disponível no ciclo 7. Como conseqüência, a instrução MUL iria utilizar os valores antigos dos registradores, e conseqüentemente, inconsistentes.

Uma possível solução é apresentada na Figura 4.8. Bolhas são inseridas no *pipeline*, garantindo que os resultados de ADD e SUB, escritos em R1 e R4, estejam prontos quando a instrução MUL ler estes valores no segundo estágio (neste caso, no ciclo 7). Nota-se, entretanto, que uma penalidade de 3 ciclos foi paga para manter a consistência dos dados.

Outra solução é fazer uso da técnica de *forwarding*. Ressalta-se que, quando os resultados de ADD e SUB são utilizados por MUL, no estágio de execução, eles já estão computados (isto é, já passaram cada um de seus estágios de execução). Então, faz-se um repasse destes resultados que estão em estágios posteriores, mas que ainda não foram gravados no banco de registradores, para o terceiro estágio onde se encontra a instrução MUL, que já poderá utilizar estes resultados em sua execução. Esta técnica é chamada de *forwarding* e é ilustrada na Figura 4.9. Salienta-se que não há nenhuma penalidade em número de ciclos quando utilizada esta técnica, apenas uma leve redução na freqüência de operação devido à multiplexação extra necessária.

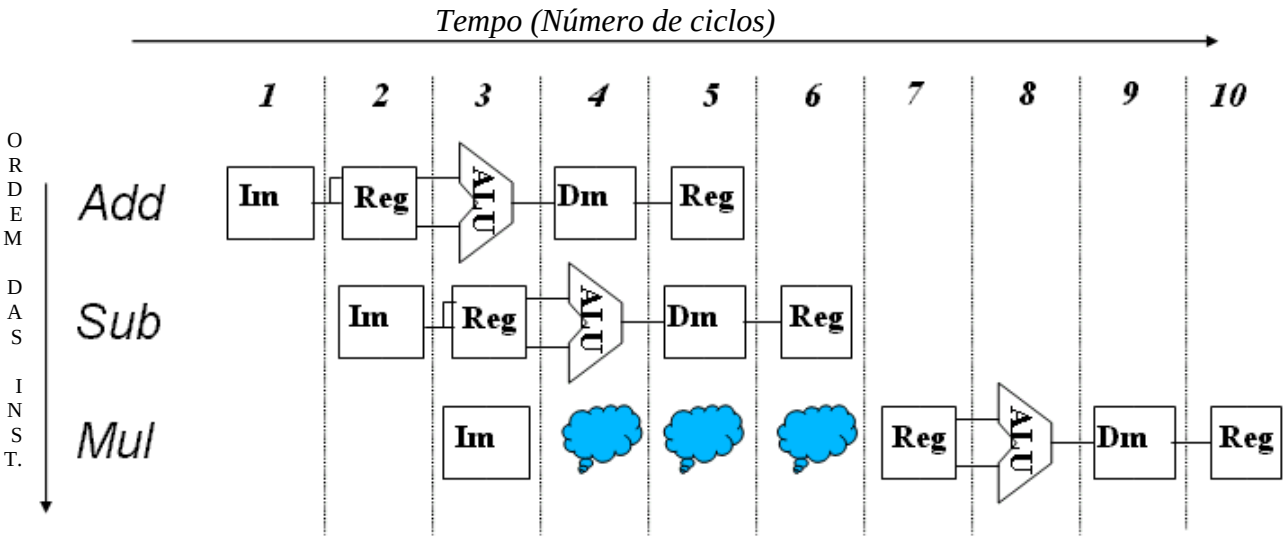

Figura 4.8 : Uma solução para evitar a inconsistência de dados em uma seqüência de instruções em um processador RISC

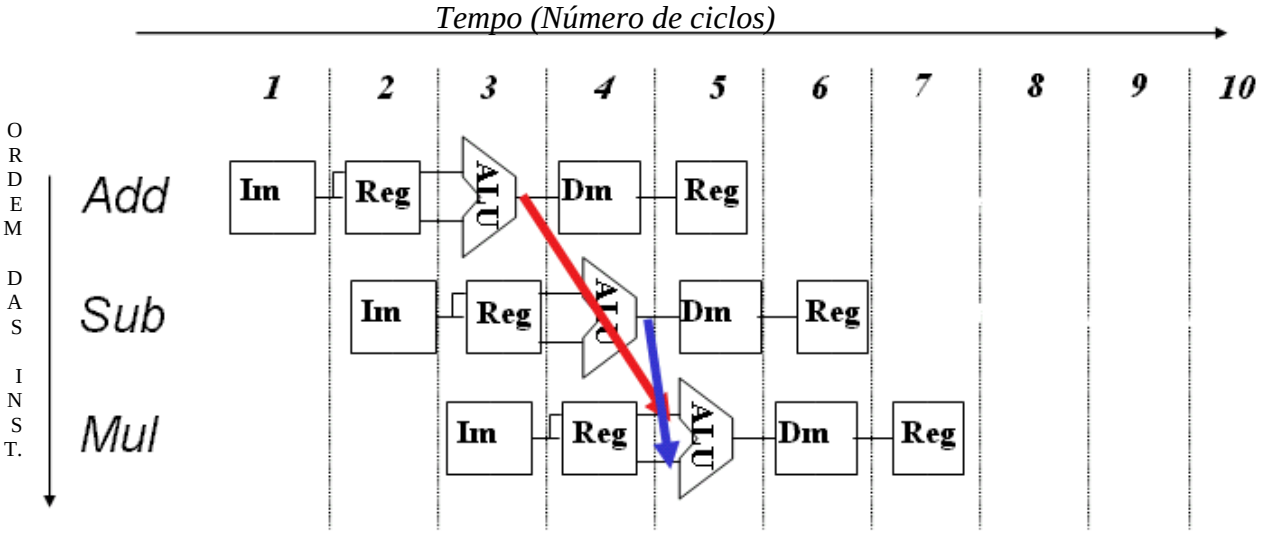

Figura 4.9 : O uso de *forwarding* para evitar o problema de consistência de dados

4.2.2 Forwarding no Femtojava Low-Power

O processador Femtojava *Low-Power* também faz uso da técnica de *forwarding*. Se há uma instrução no estágio de execução que irá escrever o seu resultado no banco de registradores (pilha de operandos ou variável local) no próximo ciclo, e a próxima instrução do fluxo precisa acessar o banco de registradores para buscar algum operando que faça parte do resultado da instrução anterior, uma dependência verdadeira (RAW) é caracterizada, como já explicado anteriormente.

Dois tipos de *forwarding* podem ocorrer: quando uma instrução no estágio de busca de operandos (terceiro estágio) necessita de um operando do topo da pilha (como a instrução istore, que grava o topo da pilha em algum lugar do depósito de variáveis locais); ou quando alguma instrução neste mesmo estágio consome dois operandos da pilha (como em operações aritméticas: iadd, isub, ior). No primeiro caso, o operando repassado vem do estágio de execução (quarto estágio). No segundo caso, o segundo operando necessário vem do ultimo estágio, de escrita dos resultados. A Figura 4.10 ilustra as duas alternativas de *forwarding*.

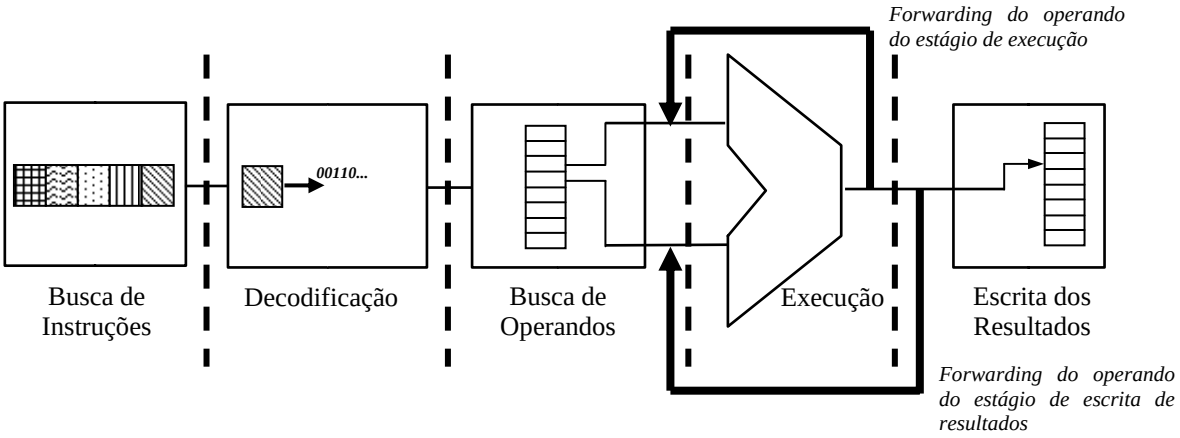

Figura 4.10 : *Forwarding* no processador Femtojava *Low-Power*

Especificamente em processadores de pilha, o uso desta técnica traz uma vantagem quando comparada com processadores RISC: ao contrário destes, em instruções que manipulam a pilha a maioria dos operandos repassados através de *forwarding* para estágios anteriores do *pipeline* não são usados mais no futuro. Como conseqüência, não há necessidade de gravar o resultado destas instruções no banco de registradores. Isto resulta em uma grande redução no consumo de potência, porque o número de escritas na pilha é reduzido. Na última seção deste capítulo, a de resultados, mostra-se um ganho de em média 8 vezes dentro do conjunto de *benchmarks* utilizado em termos de redução de potência.

4.2.3 Dependência de dados no Femtojava

A análise de dependência de dados serve para evitar conflitos de acesso ao banco de registradores (como uma leitura e escrita simultâneas em diferentes estágios do *pipeline*) e para aplicar a técnica de *forwarding*, garantindo a consistência dos dados.

Para isso, as instruções foram classificadas em grupos, conforme o tipo de acesso na pilha de operandos e no repositório de variáveis locais:

- *Instruções que escrevem na pilha de operandos:* iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, pop, pop2, bipush, sipush, getstatic, ldc, ldc_w
- *Instruções que lêem e escrevem um operando na pilha:* ineg
- *Instruções que lêem e escrevem um operando na pilha - O dado com forwarding é gravado na pilha mesmo assim:* dup
- *Instruções que lêem uma variável local e escrevem na pilha de operandos:* iload, iload_0, iload_1, iload_2, iload_3
- *Instruções que escrevem uma variável local e lêem na pilha de operandos:* istore_0, istore_1, istore_2, istore_3, istore
- *Instruções que lêem e escrevem uma variável local:* iinc
- *Instruções que lêem na pilha dois operandos:* icmpeq, icmpne, icmplt, icmpge, icmpgt, icmple, store_idx, putstatic, ifeq, ifne, iflt, ifge, ifgt, ifle
- *Instruções que lêem na pilha dois operandos e escrevem um:* iadd, isub, iand, ior, ixor, ishl, ishr, iushr, imul, iaload, baload, caload, saload
- *Instruções que lêem três operandos na pilha:* iastore
- *Instruções controladas pela máquina de estados:* goto, ireturn, return, invokestatic

A partir desta classificação, o controle de dependência de dados e *forwarding* segue as seguintes regras:

- Se houver uma instrução no 4º estágio que irá escrever um dado na pilha de operandos e há uma leitura deste dado por uma instrução no 3º, é feito o *forwarding* do operando do 4º para o 3º estágio. Exemplo: bipush no 4º estágio, istore no 3º estágio.
- Se houver uma instrução no 5º estágio que escreve um dado na pilha de operandos e há uma leitura deste dado por uma instrução no 3º, é feito o *forwarding* do operando do 5º para o 3º estágio. Exemplo: bipush no 5º estágio, instrução qualquer no 4º, istore no 3º.
- Se houver uma instrução no 4º estágio que irá escrever na pilha de operandos e outra que também escreve um dado no 5º, e uma leitura de dois operandos por uma instrução que está no 3º estágio, é feito o *forwarding* do operando do 4º para o 3º, sendo este o primeiro operando usado pela instrução no 3º, e o *forwarding* do segundo operando do 5º estágio para o 3º. Exemplo: bipush no 5º estágio, bipush no 4º, iadd no 3º.
- Se houver uma instrução no 4º estágio que irá escrever no repositório de variáveis locais e outra que lê no repositório e está no 3º, primeiro é feita uma verificação: se é a mesma variável local, estão o *forwarding* é feito, caso contrário, nenhuma ação é tomada. Exemplo: istore_2 no 4º estágio e iload_2 no 3º - há *forwarding*. istore_3 no 4º, iload_1 no 3º, não há *forwarding*.
- Se houver uma instrução no 5º estágio que escreve no repositório de variáveis locais e outra que lê no repositório no 3º, primeiro também é feita uma verificação: se é a mesma variável local, estão o *forwarding* é feito, caso contrário, uma bolha é inserida no *pipeline*, já que não é permitido fazer uma escrita e uma leitura simultânea no 5º e no 3º estágios no banco de registradores. Exemplo: istore_1 no 5º estágio, instrução qualquer no 4º, iload_1 no 3º - há *forwarding*. istore_1 no 5º, instrução qualquer no 4º, iload_2 no 3º - não há *forwarding* e uma bolha é inserida.

Há ainda instruções especiais, como de desvios e chamada e retorno de métodos, que são controladas pela máquina de estados e por serem relativamente complexas, não sofrem nenhum controle de dependência de dados ou *forwarding.* Este controle já está embutido em suas microoperações. Também há instruções como dup, que mesmo tendo seus operandos repassados, necessitam que o seu resultado seja gravado no banco de registradores de qualquer forma. Destaca-se, todavia, que o número de instruções deste tipo é muito pequeno comparadas àquelas em que o *forwarding* é feito e a escrita no banco de registradores não é necessária.

Com estas regras relativamente simples, o controle de dependência de dados e *forwarding* fica extremamente simples e barato: apenas alguns *flip-flops* foram utilizados para guardar as informações de dependência de cada instrução em cada estágio e multiplexadores para fazer o repasse de operandos.

4.3 Resultados de implementação e performance

A Tabela 4.1 mostra a área e freqüência de operação na síntese em VHDL dos núcleos das versões *Low-Power* com *forwarding* e Multiciclo. É importante salientar que o banco de registradores, usado como pilha de operandos e repositório de variáveis locais na versão *Low-Power*, tem tamanho para 32 operandos, o máximo requerido por todas as aplicações. Os resultados foram obtidos através do Quartus II versão 3.0 da Altera (ALTERA, 2004), usando a família APEX 20KE.

Tabela 4.1 : Área ocupada na implementação VHDL das versões Multiciclo e *Low-Power*

Processador	*Multiciclo*	*Low-Power*
Área	1604 LCs	3749 LCs
Freqüência de Operação	8,89 Mhz	34,13 Mhz

Na versão Multiciclo, apesar da possibilidade de geração de uma versão ASIP para cada algoritmo, a área é computada levando em conta a máquina de controle suportando todo o conjunto de instruções, assim como na versão *Low-Power*. Na versão *Low-Power*, o banco de registradores foi implementando em células lógicas, apesar de que a memória interna do FPGA pudesse ser usada por este propósito. Na Tabela 4.2, mostra-se a diferença em freqüência e área quando o banco de registradores é implementado em células lógicas e quando é implementado usando a memória interna (GOMES; BECK; CARRO; 2004). Nesta mesma tabela, também é mostrada a diferença na implementação do banco de registradores em VHDL. Na primeira forma, já apresentada na Tabela 4.1, o banco de registradores é implementado de uma forma totalmente estrutural. Em contrapartida, também foi implementado o banco de registradores de forma comportamental, deixando para o compilador a tarefa de otimizar o banco na síntese.

O aumento de freqüência do processador Low-Power comparado ao Multiciclo ocorre porque o processamento de cada instrução está dividido em estágios menores e mais equilibrados. Considerando uma instrução de adição como exemplo. Na execução no Multiciclo, a busca de operandos, a soma e preparação de escrita ocorre em um ciclo de relógio. Já na arquitetura *Low-Power*, estas operações são feitas em estágios de *pipeline* – e consequentemente ciclos de relógio – diferentes.

Tabela 4.2 : Diferença de área e freqüência na forma de implementar o banco de registradores do Femtojava *Low-Power*

Maneira de implementação do banco de registradores	*Em células lógicas, de maneira estrutural*	*Em células lógicas, de maneira comportamental*	*Na memória do FPGA*
Área	3749 LCs	3106 LC	1945 LC
Freqüência de Operação	34,13 Mhz	30.63 MHZ	30.74 MHZ

Como pode ser observado, há uma redução de quase metade da área em células lógicas e uma perda de quase 7% na freqüência de operação quando usada a memória interna do FPGA (em relação à forma de implementação estrutural do banco de registradores), mostrando que a busca de operandos é um dos caminhos críticos de todo sistema, em relação ao tempo de operação. Além do mais, quando o banco de registradores é implementado de uma forma comportamental, há uma economia de área de 643 células lógicas (-17%), entretanto a freqüência do sistema cai em 10.5%, também comparando com a forma de implementação estrutural.

A Tabela 4.3 mostra o desempenho de cada processador, em número de ciclos, de cada arquitetura. Os resultados foram obtidos usando o simulador CACO-PS.

Tabela 4.3 : Desempenho em número de ciclos dos diferentes algoritmos nas arquiteturas

ALGORITMO	***NÚMERO DE CICLOS***		
	Multiciclo	***Low-Power***	***Low-Power com Forwarding***
Seno	2447	1237	755
Bubblesort	6774	3234	2468
Selectsort	5335	2703	1930
Insertsort	4093	2071	1571
Busca binária	1162	602	403
Busca seqüencial 1	8497	3803	2765
Busca seqüencial 2	7586	2779	1997
IMDCT	140300	61841	40306
Somas ponto flutuante	30747	18735	14531

Operando na mesma freqüência, o núcleo do Femtojava *Low-Power*, implementando a técnica de *forwarding*, é a arquitetura que possui o maior consumo de potência por ciclo, já que esta é a arquitetura mais complexa, com mais componentes combinacionais e registradores. Este comportamento pode ser observado na Figura 4.11.

Apesar das versões com *pipeline* consumirem mais potência por ciclo que a versão multiciclo, a energia total gasta pelo sistema para executar cada algoritmo é menor quando há *pipeline*. São duas as razões para isso: o maior *throughput* alcançado por estas versões e a diminuição de acessos à memória principal.

A Figura 4.12 mostra a energia consumida por causa dos acessos à RAM. Como o Femtojava Multiciclo usa a memória principal para a pilha de operandos e repositório de variáveis locais, há uma grande diferença entre esta arquitetura e as que possuem *pipeline*. Elas, por sua vez, apenas fazem acesso na memória principal em chamadas e

retornos de métodos ou em instruções específicas, como *getstatic* e *putstatic*. Esta figura mostra a vantagem de fazer esta implementação em um banco de registradores ao invés de usar a memória principal para este propósito.

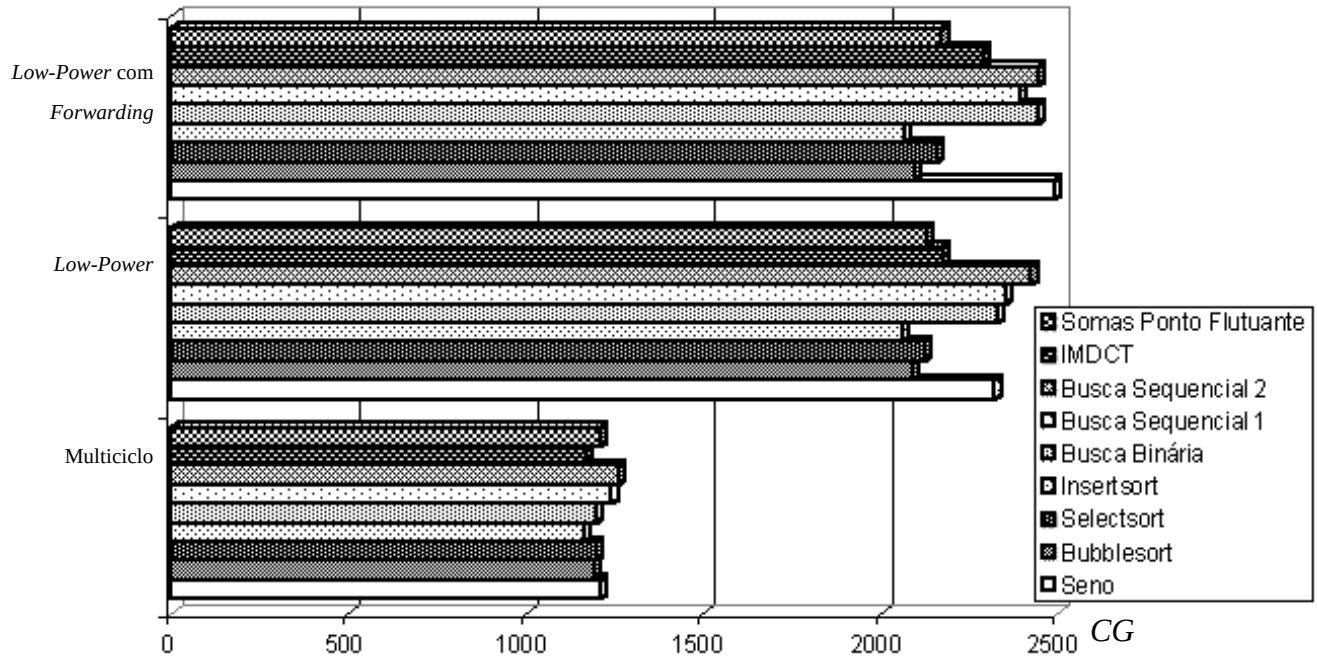

Figura 4.11 : Energia consumida por ciclo nas versões Multiciclo e *Low-Power*, com e sem *forwarding*

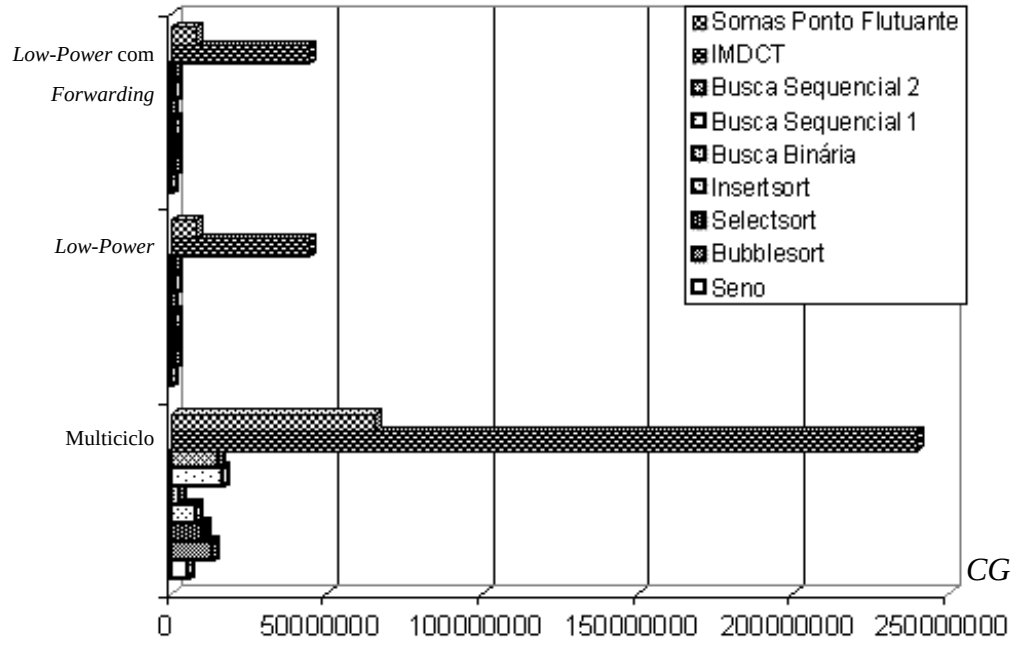

Figura 4.12 : Energia gasta devido aos acessos à memória principal, em cada algoritmo nas diferentes arquiteturas

A Figura 4.13 mostra a energia consumida pelo núcleo das arquiteturas apresentadas até agora. A diferença de consumo total de energia entre a versão multiciclo e as versões com *pipeline* é devido à melhor média de IPC. Visto que o multiciclo necessita de 3 a 14 ciclos para executar uma instrução, o número de ciclos para ser executado um algoritmo é muito maior do que nas outras versões. Além do mais, proporcionalmente, os elementos das versões com *pipeline* são melhor aproveitados ao decorrer do tempo. Também, como pode ser observado, a potência consumida no núcleo é reduzida da versão com *pipeline* sem *forwarding* para a versão que implementa a técnica. Esta diferença tem duas razões: A melhor utilização das unidades funcionais por causa da diminuição de bolhas dentro do *pipeline*; e a diminuição de escritas no banco de registradores, já que a média de potência consumida pelos registradores por ciclo foi reduzida em 70%, devido ao uso da técnica de *forwarding*.

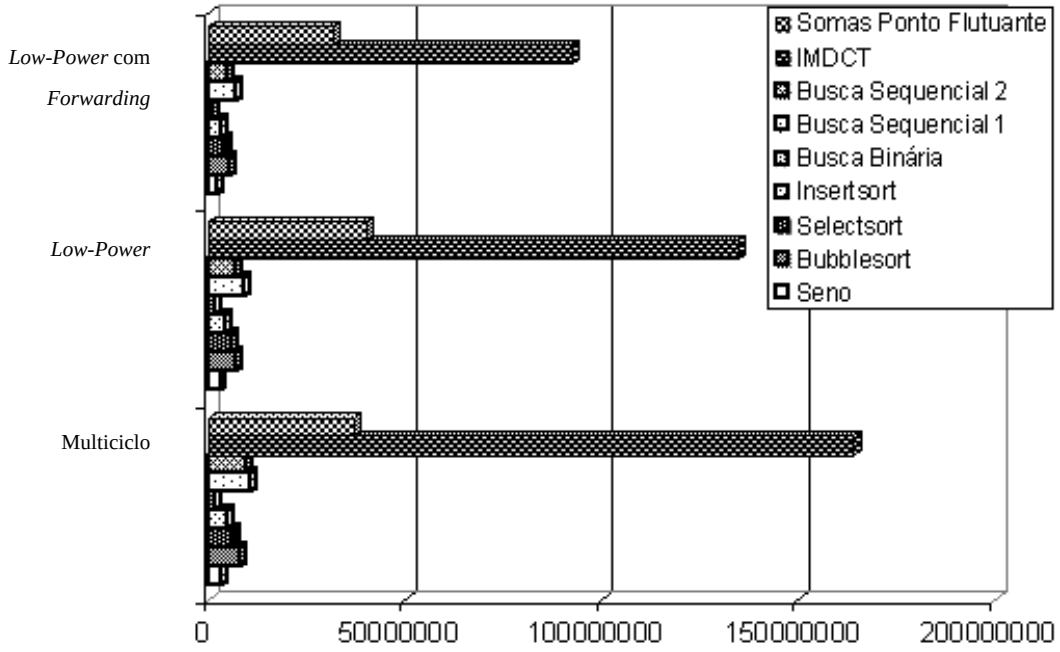

Figura 4.13 : Energia gasta no núcleo nas arquiteturas por cada algoritmo

Como pode ser observado na Figura 4.14, a energia total (considerando ROM, RAM e núcleo) é drasticamente reduzida em alguns algoritmos. Para os algoritmos que não apresentam uma redução tão drástica, há duas razões. A primeira decorre dos programas que fazem uma grande quantidade de chamadas de métodos. Como conseqüência, a memória principal também é bastante acessada, já que os valores de retorno e informações sobre o método não ficam guardados no banco de registradores no núcleo do processador. A outra razão é por causa dos programas que fazem uso intensivo do repositório de variáveis locais do método. Nota-se que mesmo que a técnica de *forwarding* seja aplicada também no repositório de variáveis, estes valores, ao contrário do que acontece *forwarding* com a pilha de operandos, precisam ser gravados de qualquer maneira, pois não há garantia que eles não sejam usados no futuro.

Em aplicações de sistemas embarcados, onde muitas delas possuem exigências de tempo real, um *throughput* específico precisa ser garantido para a aplicação. Assumindo que esse *throughput* é alcançado pela versão multiciclo, a freqüência de operação das

versões com *pipeline* pode ser reduzida com o objetivo de economizar potência, já que estas arquiteturas podem executar mais instruções por ciclo, como já demonstrado na Tabela 4.3.

Além do mais, assumindo que a potência dinâmica é a dominante na potência consumida pelo sistema, e todas as portas do microprocessador formam uma capacitância de chaveamento coletiva C com uma freqüência de chaveamento em comum F, a seguinte fórmula é obtida (RABAEY, 1996):

$$P = C \cdot f \cdot Vdd^2 \qquad (1)$$

Como pode ser observado em (POUWELSE; LANGENDOWN; SIPS, 2001), a tensão de operação do processador Transmeta TM5400 (TRANSMETA, 2004), desenvolvido para sistemas embarcados, diminui em um fator de 4,6% quando a freqüência de operação é diminuída em 10%.

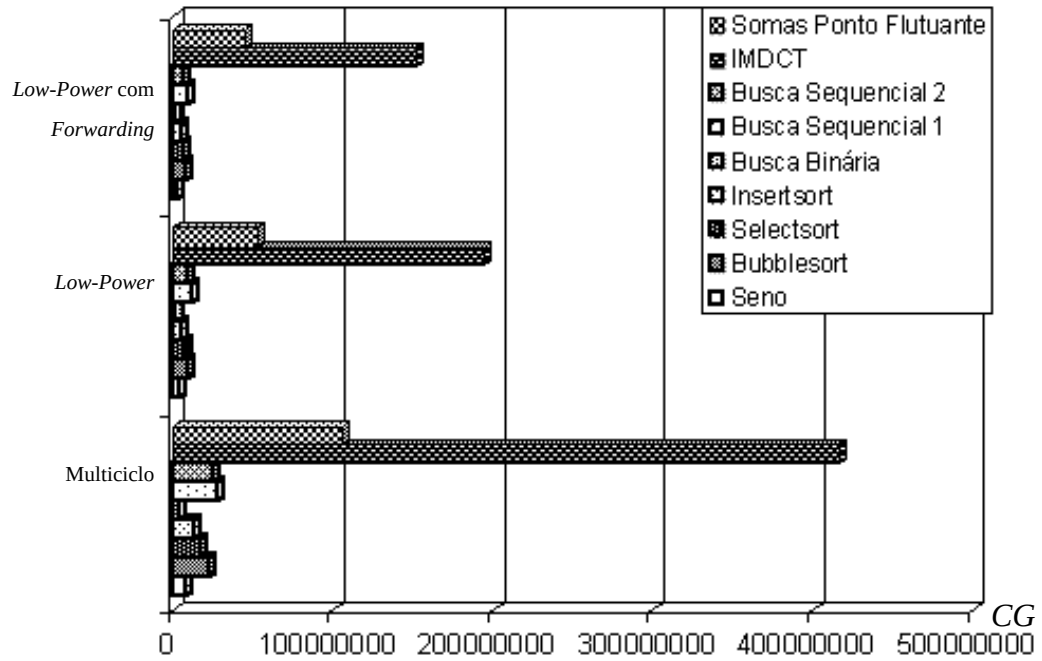

Figura 4.14 : Energia total gasta por cada algoritmo em cada arquitetura do Femtojava

A Figura 4.15 mostra a economia relativa no consumo de energia quando a freqüência da versão com *pipeline* (sem *forwarding*) é reduzida para alcançar exatamente o mesmo *throughput* da versão multiciclo, e quando a tensão de operação é reduzida graças à redução de freqüência, usando como base a equação em (1).

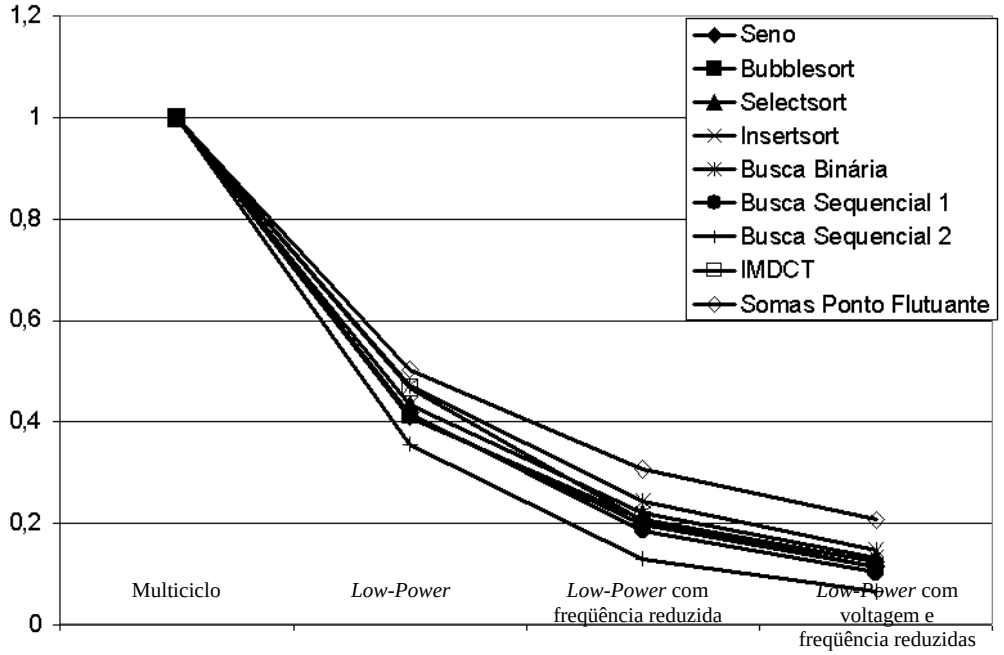

Figura 4.15 : Energia consumida pelo núcleo do Femtojava *Low-Power* sem *forwarding* com a freqüência e tensão de operação reduzidas

Aplicando a técnica de *forwarding*, o consumo de energia é reduzido ainda mais, como pode ser observado na Figura 4.16, quando é mostrada a redução relativa no consumo de energia quando há a comparação entre as duas versões, que tiveram suas freqüências e tensões de operação reduzidas para alcançar o mesmo *throughput* da versão multiciclo.

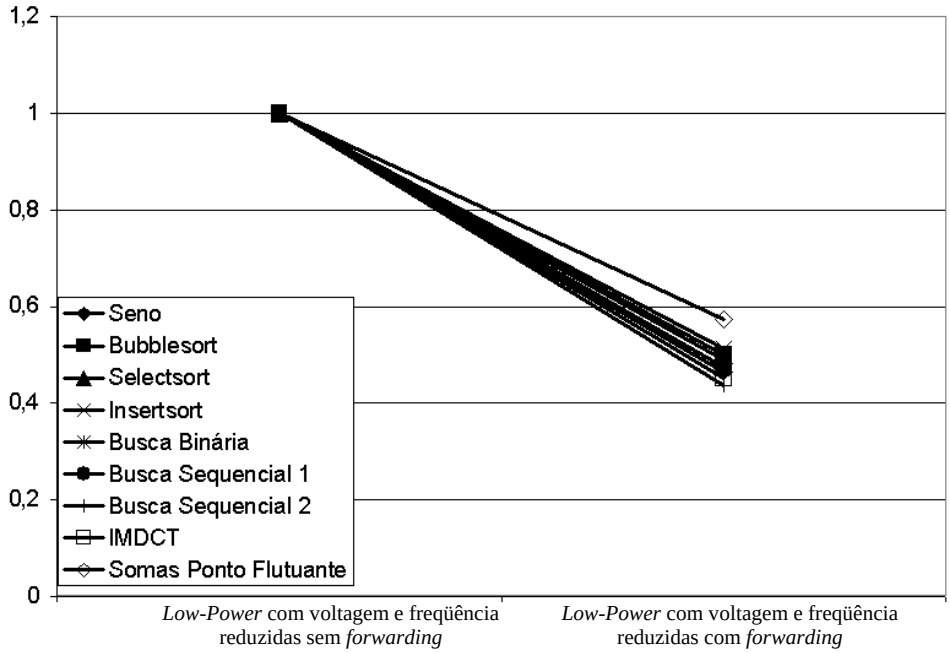

Figura 4.16 : Comparação entre a energia das arquiteturas *Low-Power* não usando e usando a técnica de *forwarding*, ambas com a freqüência e voltagem reduzidas

Finalmente, mostra-se a vantagem do incremento de área do processador para suportar o *pipeline, forwarding* e a pilha implementados no banco de registradores, trocando esta área por uma enorme economia na potência consumida. A Figura 4.17 mostra a comparação relativa entre o incremento de área e a média de energia economizada por todas as aplicações testadas.

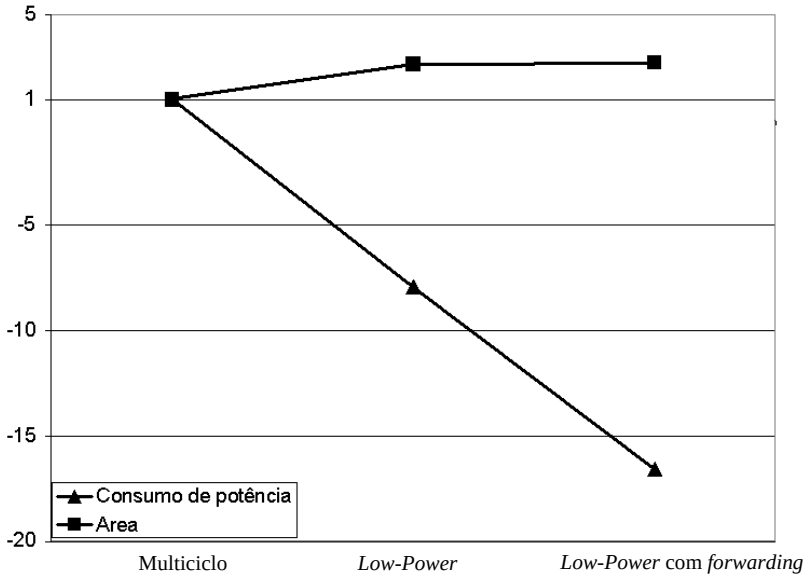

Figura 4.17 : Custo-benefício entre o aumento de área e o consumo de energia na arquitetura *Low-Power*

Como pode ser observado na Figura 4.17, enquanto a área aumentou em um fator de aproximadamente 2,7, uma economia em um fator de 16 vezes foi alcançada em termos de energia, na média de todos os algoritmos usados do *benchmark* adotado.

Todos os resultados, em termos de potência, foram obtidos no simulador CACO-PS. Entretanto, independente do simulador utilizado ou de alguma margem de erro existente devido à simulação comparado ao sistema real, o Femtojava *Low-Power* traz indubitavelmente vantagens no consumo de potência e energia devido a dois principais fatores: a média de IPC aumenta significantemente e a escrita nos registradores é reduzida de forma drástica devido o uso da técnica de *forwarding*.

5 A TÉCNICA DE FOLDING

A técnica de *folding* de instruções foi proposta pela primeira vez no processador picoJava-I (O'CONNOR; TREMBLAY, 1997), com o intuito de providenciar uma solução para o problema de acesso ineficiente de operandos em máquinas de pilha. Na maioria das implementações destes processadores, as operações precisam de vários passos para serem completadas, conseqüentemente afetando o *throughput* das instruções executadas. Estas implementações geralmente acessam os operandos da pilha, fazem uma operação sobre eles e colocam o resultado novamente nesta pilha. Como já explicado anteriormente, processadores baseados em pilha têm uma lógica computacional um pouco diferente de computadores convencionais, como RISC. Por exemplo, para somar dois números, são necessários três passos:

- Empilha-se o primeiro operando
- Empilha-se o segundo operando
- Retira-se estes dois operandos da pilha, faz-se a soma, e coloca-se o resultado no topo da pilha

Geralmente há mais um passo adicional, que é gravar o resultado do topo da pilha no repositório de variáveis locais do método.

Além do mais, máquinas de pilha convencionais limitam os seus acessos dos operandos na parte superior da pilha. Como resultado, quando os operandos não estão disponíveis no topo da pilha, mas são necessários para uma operação, eles precisam ser copiados do repositório de variáveis locais do método para o topo da pilha. Eliminando esse passo extra, seria possível melhorar a performance de uma arquitetura de pilha típica.

Um exemplo bem simples de operação de *folding* pode ser observado na Figura 5.1. Na Figura 5.1a, considerando que um operando já se encontra no topo da pilha (T), é necessário primeiramente colocar o segundo operando, que não está disponível na pilha, mas sim no depósito de variáveis locais do método, também no topo da pilha (L0). Depois que T e L0 estiverem no topo da pilha eles são somados. Após, o que se encontra no topo da pilha é o resultado desta operação. Na Figura 5.1b, é mostrada a operação com *folding*. Novamente, parte-se do principio que o primeiro operando da pilha, T, já se encontra no topo. Entretanto, não é necessário o passo intermediário, que se resume em uma instrução a mais para colocar o L0 no topo da pilha, para depois efetuar sua soma com T. Já que L0 já se encontra na *cache* da pilha, mas em um local diferente (no repositório de variáveis locais), retira-se diretamente o valor de lá e a soma com T é feita, colocando o resultado no topo da pilha. Então, com a operação de *folding*, foi economizada uma instrução: a de carregar uma variável local (L0) no topo da pilha.

Isso só foi possível porque é permitido acessar qualquer endereço na pilha, não apenas os endereços localizados ao topo.

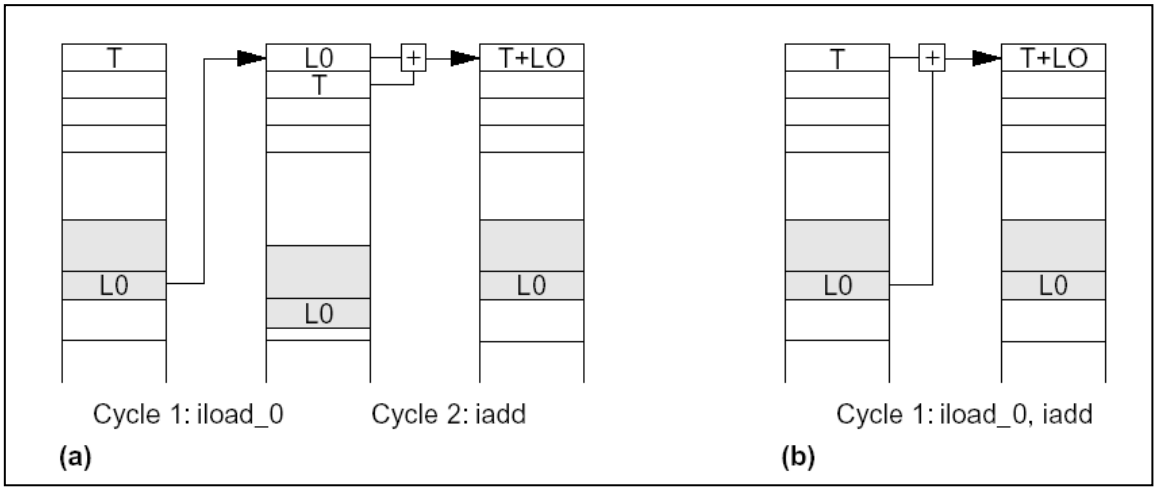

Figura 5.1 : Exemplo da operação de *folding* (O'CONNOR; TREMBLAY, 1997)

Esse é um típico exemplo de *folding* de duas instruções. Entretanto, há várias outras seqüências de instruções no qual *folding* pode funcionar. O picoJava-I, entretanto, não implementa um grande conjunto de instruções a serem detectadas, visto que a detecção é feita dinamicamente, isto é, em tempo de execução. O *hardware* adicional necessário, baseado em uma máquina de estados, pode se tornar bastante caro e até proibitivo caso existam muitas possibilidades de seqüências de instruções a serem detectadas.

5.1 Técnicas de detecção dinâmica de folding

Nesta seção são demonstradas técnicas para a análise de seqüências de instruções possíveis para a implementação do *folding*. No exemplo anterior, apenas duas instruções foram consideradas para ser feito o *folding*. Entretanto, o *folding* pode ser feito em seqüências enormes de instruções. Contudo, quanto maior o número de instruções na seqüência, mais *hardware* é necessário. E este *hardware* não é gasto apenas com a máquina de estados para a detecção da seqüência. Também é gasto, por exemplo, com mais elementos combinacionais, como multiplexadores, para buscar diferentes operandos e colocá-los em diferentes locais das unidades funcionais, ou ainda com mais portas de leitura na memória.

Como já mencionado, a técnica de *folding* foi primeiramente introduzida na arquitetura do picoJava-I. Todavia, a técnica de *folding* do picoJava-I detecta apenas padrões onde instruções de *load* são imediatamente seguidas por instruções que usem o dado empilhado por ela. O sucessor, picoJava-II, melhorou o método de detecção, com seis diferentes tipos de grupos de *folding* de instruções, usando sete diferentes padrões baseados nestes seis grupos.

Em (TOM et al., 1997) é proposto o método POC de detecção de instruções nas quais podem ser feitas *folding*. O referido trabalho divide as instruções Java em três classes:

- *Produtores:* Todas as instruções que colocam dados na pilha de operandos, que podem ser provenientes do repositório de variáveis locais do método ou ainda serem constantes ou dados imediatos.
- *Consumidores:* As instruções que tiram dados da pilha de operandos e colocam estes dados no repositório de variáveis locais do método.
- *Operadores:* As instruções que tiram dados da pilha de operandos, executam algum tipo de operação, e então colocam novamente outro dado – o resultado desta operação – nesta pilha.

Depois de uma análise em uma série de programas Java, desde aplicações gráficas simples, jogos comuns, *benchmarks* sintéticos e o compilador Java da Sun Microsystems, notou-se que o *folding* pode ser feito quando algum produtor coloca (ou produz) dados na pilha de operandos e um ou mais consumidores tiram este dado (ou consomem) da pilha. Ou ainda quando algumas instruções produtor colocam dados na pilha e uma instrução operador processa este dado e coloca o resultado na pilha, que por sua vez é consumido por uma instrução consumidor. A Tabela 5.1 mostra a classificação adotada para cada conjunto de instruções e suas subcategorias dentro de cada conjunto. Nesta tabela, nota-se que um operador pode ser instruções do tipo E (de *Execution*, ou execução), B (de *Branch* ou desivo) e C (de *Complex* ou complexas). Produtores são do tipo L (de *Load*) e consumidores, S (de *Store*). A razão para dividir em subconjuntos (ou tipos) as instruções operador é que, por exemplo, pode ser feito *folding* com instruções tipo E com um consumidor (tipo S), entretanto, *folding* não ocorre com um tipo B seguido de um consumidor, tipo S, já que instruções tipo B são instruções de desvio, que obviamente não podem ser unidas com instruções seguintes.

Tabela 5.1 : Classificação das instruções Java seguindo o modelo POC

Classificação	***Tipos***	***Descrição***
Produtor	L	Instruções de *load* do repositório de variáveis locais do método / Instruções que empilham constantes
Consumidor	S	Instruções de *store* para o repositório das variáveis locais do método
Operador	E	Instruções de execução
	B	Instruções de controle e de desvio
	C	Instruções complexas ou microprogramadas

Baseado nesta classificação, procurou-se os padrões mais utilizados de instruções para folding. Já que há milhares deles, a utilização de todos tornaria inviável a implementação em hardware no decodificador para a detecção e substituição das instruções. Com estas seqüências de instruções mais comuns achadas, notou-se que os padrões mais freqüentes são seqüências de 2 até 4 instruções. Então foram propostas as estratégias 2-foldable, 3-foldable e 4-foldable, que detectam e substituem seqüências de 2, 3 e 4 instruções, respectivamente, para folding. Ainda conforme (TON et al., 1997), os padrões mais freqüentes para estes três tipos de seqüência são listados nas Tabelas 5.2, 5.3 e 5.4.

Tabela 5.2 : Seqüência de instruções mais comum para *2-foldable*

1ª instrução	*2ª instrução*
L	S
L	E
L	B
L	C
E	S

Tabela 5.3 : Seqüência de instruções para *3-foldable*

1ª instrução	*2ª instrução*	*3ª instrução*
L	L	E
L	L	B
L	L	C
L	E	S

Tabela 5.4 : Seqüência de instruções para *4-foldable*

1ª instrução	*2ª instrução*	*3ª instrução*	*4ª instrução*
L	L	E	S

Estas seqüências são procuradas em ordem decrescente, isto é: primeiro procura-se seqüências *4-foldable*. Caso esta seqüência não for achada, tenta-se, reaproveitado as instruções, detectar *3-foldable* e finalmente *2-foldable*.

Um modelo avançado de POC foi proposto em (KIM; CHANG, 2000), aumentando o conjunto de seqüências detectáveis. Como pode ser analisado na Figura 5.2, o modelo antigo só tentava detectar o que é classificado como a seqüência normal (*Normal Sequence*). Ao contrário, neste novo modelo são analisadas seqüências não contíguas de instruções.

A primeira seqüência é classificada como tipo I, observada na Figura 5.2 (*type I*). A primeira instrução iload_2 pode ser unida para *folding* com as últimas três instruções, formando o grupo de instruções B. Cada grupo de instruções pode ser substituído por uma única instrução de *folding*. Importante ressaltar que este tipo de seqüência de instrução para *folding* só pode acontecer devido às características particulares de uma arquitetura de pilha. Neste caso, houve um carregamento da variável local com índice 2 para o topo da pilha. Depois mais quatro instruções se passaram (grupo A) e fizeram uso da pilha. Após isso, a antiga variável ficou novamente ao topo da pilha, podendo ser usada pelas três últimas instruções do grupo B.

Todavia, a análise de dependência de dados também precisa ser feita. É o caso da seqüência classificada de tipo II (*type II*), também observada na Figura 5.2. Ao contrário do tipo I, onde o grupo de instruções A poderia ficar antes do B e vice-versa, neste tipo, o grupo B, quando transformado em uma instrução de *folding*, precisa ficar antes do grupo A. A razão disto é que o grupo B primeiramente lê a variável local 3, e o conjunto

A escreve nela a seguir, caracterizando uma dependência que precisa ser respeitada para manter a consistência dos dados.

O tipo III (*type III*, na Figura 5.2) mostra a impossibilidade de formar apenas dois grupos de instruções para *folding* devido à dependência de dados. O conjunto de instruções teve que ser divido em três, porque a primeira instrução lê a variável local 3, e o grupo A escreve nela. Só que o grupo A também lê a variável de índice 8, enquanto que as próximas instruções, que formam o grupo D, escrevem nela. É por este motivo, de dependência de dados, que não se pode fazer uma separação na seqüência de instruções semelhante aos tipos II e III. Finalmente há o último tipo (*type IV*, observado na mesma figura), onde as instruções estão aninhadas de forma eqüidistante formando os grupos B, C, D, junto com as três instruções centrais que formam o grupo A. Dentre os *benchmarks* utilizados no referido trabalho, os tipos II e III raramente aparecem.

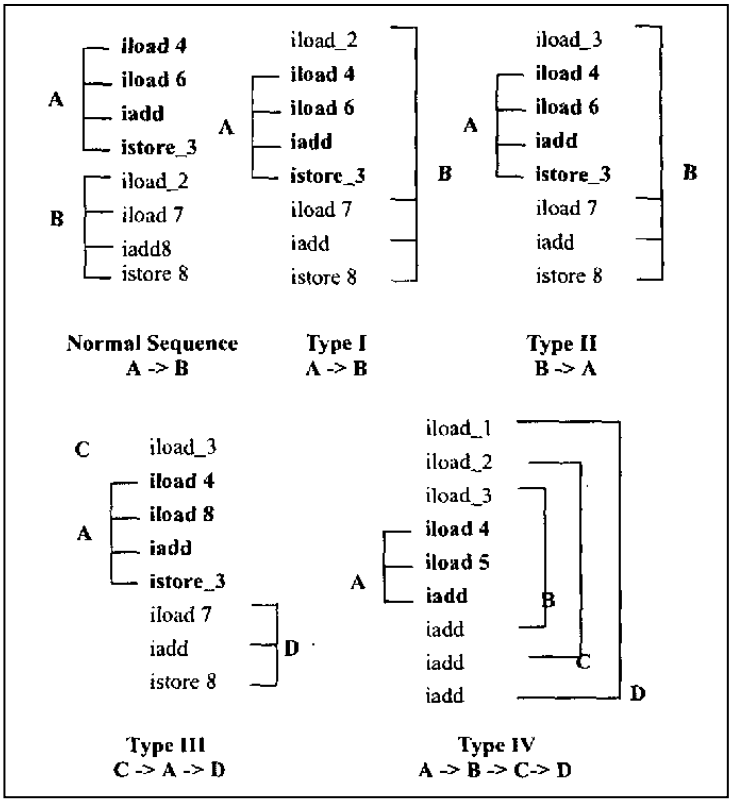

Figura 5.2 : Outras formas de detectar seqüências de instrução para *folding*

5.2 *Folding* no Femtojava *Low-Power*

Como explicado anteriormente, a detecção e substituição de instruções para *folding* seriam feitas de forma estática, previamente, em estágios iniciais do ciclo de projeto. Assim, o processador Femtojava, da visão de seu decodificador, consideraria uma instrução de *folding* apenas como mais uma instrução adicional convencional, sem fazer distinção de seu conjunto original de instruções.

Considerando a seqüência de instruções descritas na Figura 5.3, elas seriam executadas no *pipeline* do Femtojava *Low-Power* como ilustrado na Figura 5.4.

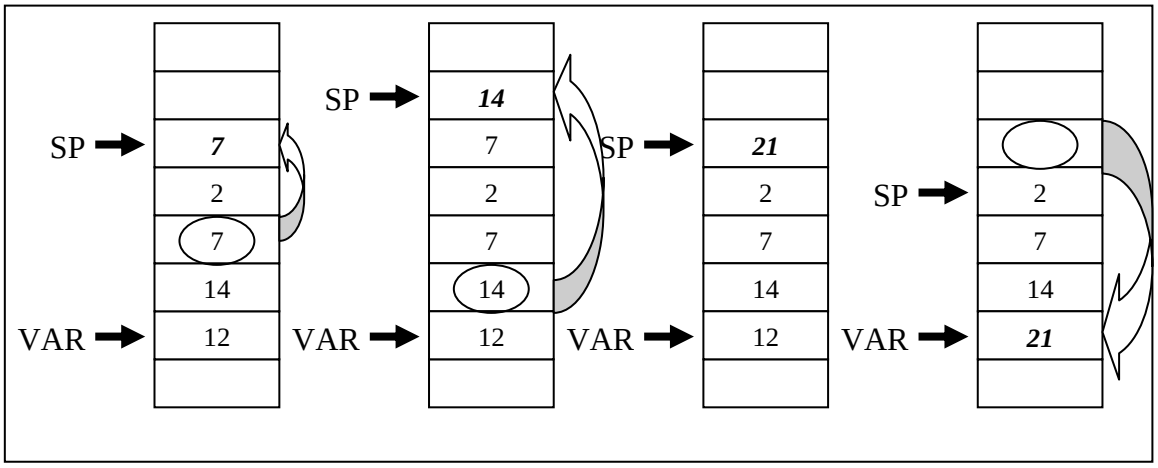

Figura 5.3 : Típica seqüência de instruções para ser feito *folding*

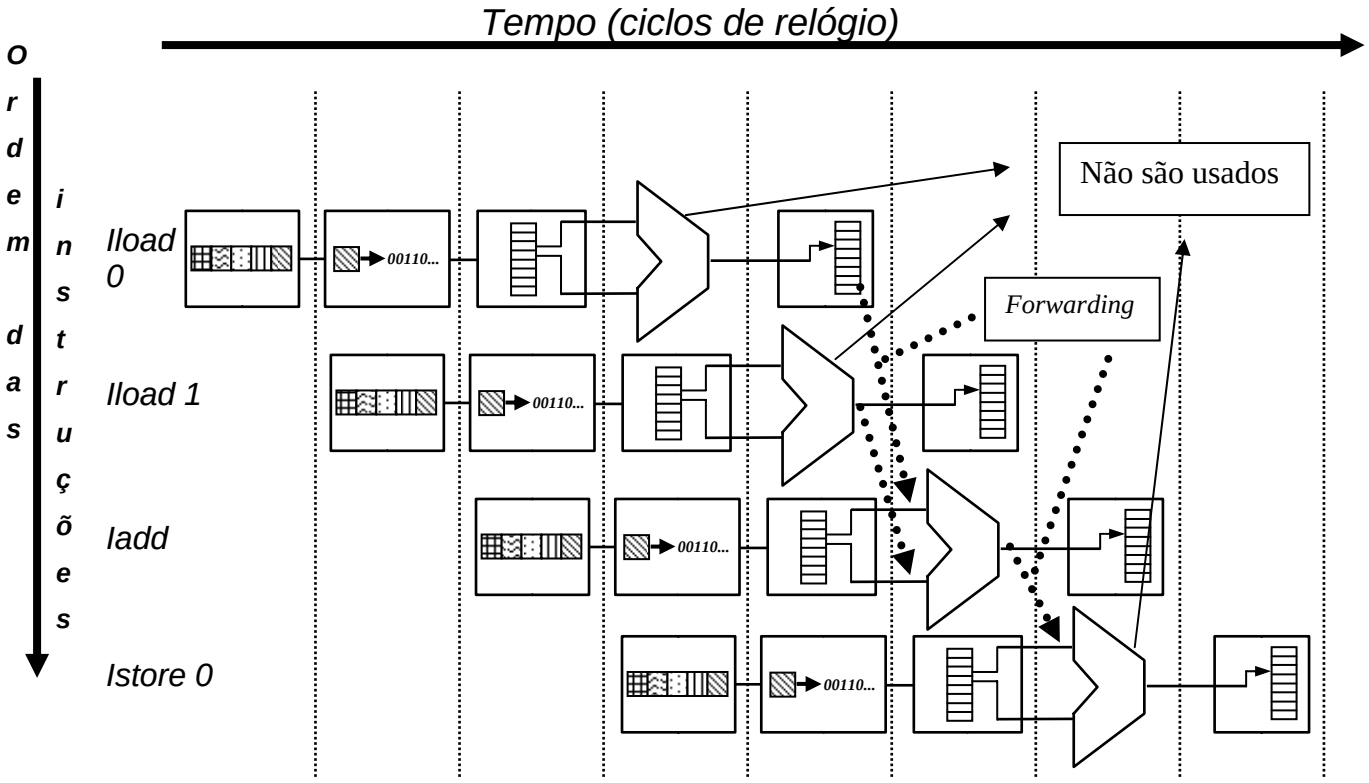

Figura 5.4 : Uma seqüência de instruções sendo executadas no *pipeline* do Femtojava *Low-Power*

As instruções Iload efetivamente utilizam o primeiro estágio, para a sua busca, o segundo estágio, para sua decodificação, o terceiro estágio, para a busca de operandos no repositório de variáveis locais e o quinto estágio, para a escrita do operador buscado no topo da pilha, no banco de registradores. Como não há nada para ser operado, a instrução passa em branco pelo quarto estágio. Particularmente neste caso, a escrita nos registradores nem chega a acontecer, pois os dois operandos são repassados (ocorre o

forwarding) para a instrução ladd. Esta, por sua vez, usa todos os últimos três estágios, o terceiro para a busca dos dois operandos anteriores, o quarto para a soma deles, e o quinto para gravar o resultado (neste caso, novamente há *forwarding* para a próxima instrução). A instrução lstore_0 não usa o quarto estágio, pois não há nenhuma operação a ser feita. Ela usa o terceiro para buscar o resultado do ladd e o quinto para gravá-lo no banco de registradores, no repositório de variáveis locais.

Fazendo uma análise mais profunda, nota-se que o quarto estágio, nestas quatro instruções, é só usado uma vez, isto é, há apenas uma unidade funcional envolvida em toda esta operação. Além do mais, os dois lload utilizam o quinto estágio apenas para escrever seus resultados (ou para *forward*) que por sua vez já vão em seguida ser utilizados na instrução iadd. Já aí surge uma primeira alternativa para *folding*. Une-se os dois lload à instrução iadd.

Esta instrução seria semelhante a:

```
iload_add 2 1
```

Ela teria dois dados imediatos (2 e 1), que na mesma instrução, seriam buscados no banco de registradores (repositório de variáveis locais) no terceiro estágio, somados no quarto e guardados no topo da pilha no quinto. Entretanto, depois este resultado será gravado em um local do repositório de variáveis locais, pela instrução *Istore_0*. Assim, poder-se-ia gravar este resultado diretamente no repositório de variáveis locais ao invés de no topo da pilha. Considerando-se isso e estendendo este *folding* de instruções, a instrução ficaria da seguinte maneira:

```
iload_add_istore 0 2 1
```

Onde a soma dos valores de índice 2 e 1 no repositório de variáveis locais seriam gravadas no local de índice 0 no mesmo repositório. Note que esta instrução tornou-se muito semelhante a uma instrução típica RISC que usa três registradores:

```
add r0, r1, r2
```

A Figura 5.5 ilustra o *folding* da seqüência de instruções anterior e a Figura 5.6 como ele ficaria no *pipeline* do Femtojava *Low-Power*. Note que no *pipeline*, não há nenhuma sobreposição de funções e há um melhor aproveitamento de todos os estágios. *Hardware* adicional para a nova palavra de controle, assim como registradores e multiplexadores, usados para manipular o terceiro imediato que seria o índice de escrita, seriam necessários. Além do mais, a fila de instruções teria que ser maior, para lidar com instruções de 3 operandos imediatos.

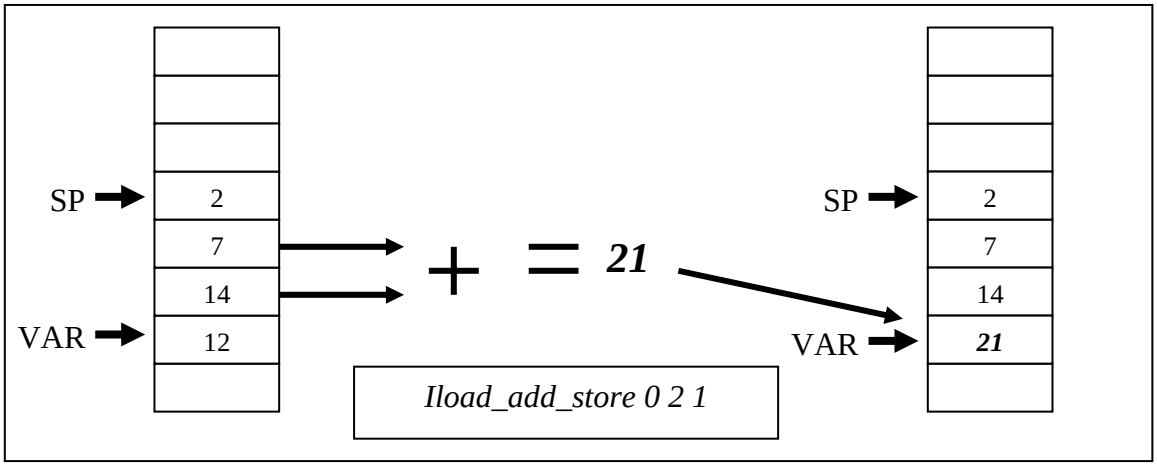

Figura 5.5 : Seqüência de instruções anterior unidas com *folding*

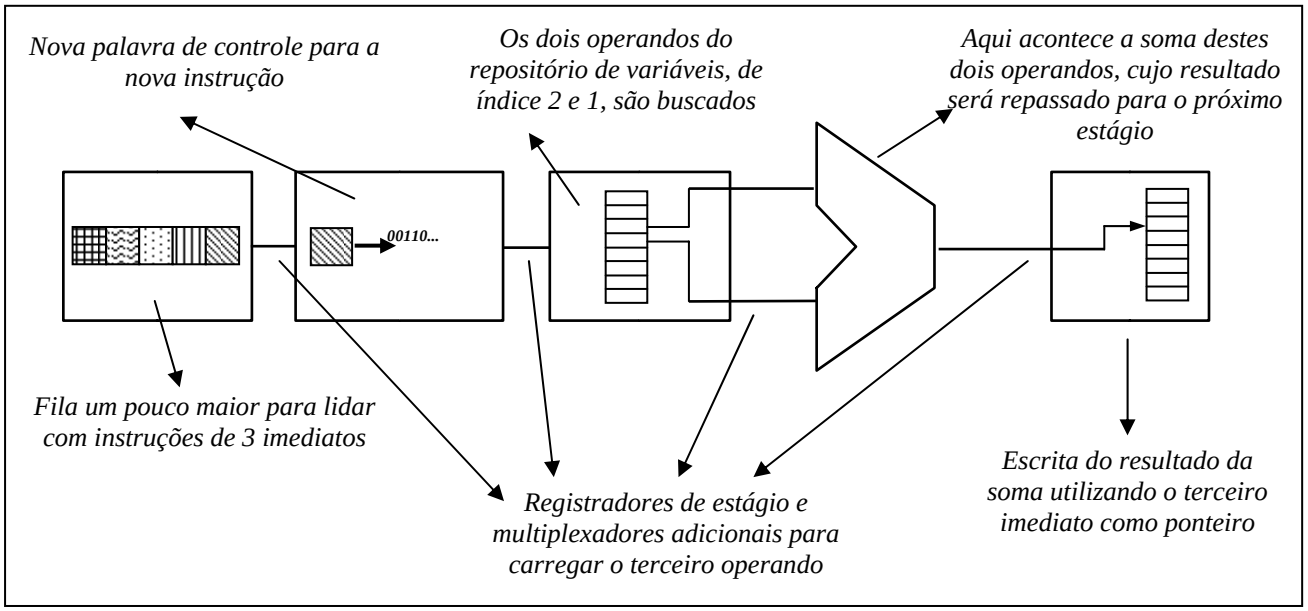

Figura 5.6 : Instrução de *folding* da seqüência de instruções anterior no *pipeline* do Femtojava *Low-Power*

5.3 Analisador estático para *folding*

Até agora, em todos os trabalhos estudados, foram propostos métodos de detecção de *folding* dinâmicos, isto é, em tempo de execução do programa Java. Isto traz algumas conseqüências como aumento de área e de potência consumida. Além do mais, o conjunto de seqüências de instruções a ser detectado fica limitado, pois é diretamente proporcional à área final do sistema. Em sistemas embarcados, onde a maioria dos programas executados é gravada em algum tipo de ROM, isto é, não são modificados depois de produzidos, a detecção dinâmica de seqüências de instruções para *folding* é repetir o mesmo trabalho para o mesmo conteúdo várias vezes. Assim, a análise e substituição de código para *folding* de forma estática economizariam em área e potência. Somando-se a isso, a versão ASIP do Femtojava *Low-Power* só suportaria as

instruções de *folding* que aquela aplicação necessita, utilizando uma versão estendida do programa *Sashimi* para isto.

Assim, provando a possibilidade da implementação de *folding* no Femtojava *Low-Power* como apresentado na seção anterior, foi desenvolvido um analisador estático de *folding* para o Femtojava, onde o conjunto de *benchmarks* deste trabalho, apresentado no capítulo 3, foi analisado.

O analisador recebe como entrada o arquivo de programa no formato .MIF (ALTERA, 2004) e faz a análise dos *bytecodes* de maneira semelhante ao modelo POC. Além do mais, pode substituir a seqüência de instruções de *folding* por uma nova instrução, caso desejado pelo usuário. Existe um arquivo de configuração para a detecção de seqüências, formados por grupos de seqüências de *folding* que ele pode detectar. Um exemplo de um grupo está na Figura 5.7.

```
BEGIN fold 4
iconst_m1,iconst_0,iconst_1,iconst_2,iconst_3,iconst_4,bipush,sipush,iload,iload_0,iload_1,iload_2,iload_3,getstatic
iconst_m1,iconst_0,iconst_1,iconst_2,iconst_3,iconst_4,bipush,sipush,iload,iload_0,iload_1,iload_2,iload_3,getstatic
iadd,isub,imul,ishl,ishr,iushr,iand,ior,ixor
istore,istore_0,istore_1,istore_2,istore_3
END
```

Figura 5.7 : Exemplo de um grupo de *folding* no arquivo de configuração do analisador

A análise começa lendo este arquivo e gravando sua estrutura em memória. A partir disto, para cada instrução do programa Java é verificado se há uma seqüência de instruções para *folding*. A procura por esta seqüência sempre parte daquela que possui um maior número de instruções para o menor. Em outras palavras, tenta-se procurar sempre para determinada instrução aquela seqüência que possui mais instruções para *folding*. Se não for achada, passa para uma seqüência menor e assim sucessivamente.

A seqüência de instruções para a geração de *folding* é dada por grupos. Existem vários grupos como o do exemplo, cujo nome é “4”. Para cada instrução lida, é verificado na primeira linha de cada grupo se há esta instrução. Caso houver, verifica-se se a próxima instrução existe na segunda linha daquele grupo, e assim sucessivamente, até acabarem as linhas daquele grupo. Por exemplo, se a instrução `iconst_1` for lida do arquivo de memória, procura-se na primeira linha de cada grupo se existe esta instrução. Caso esta instrução foi achada no grupo “4”, exatamente o que está sendo usado como exemplo, então a instrução seguinte na memória, por exemplo, `bipush`, é procurada na segunda linha do grupo “4”. E assim acontece para as instruções `iadd` e `istore_2`, terceira e quarta instruções.

Importante destacar que as mesmas instruções podem estar em diferentes grupos. Neste grupo do exemplo, procura-se por uma seqüência de quatro instruções. Entretanto, a instrução `iconst_1` poderia estar em outros grupos que procuram por seqüências de 3 ou 2 instruções, por exemplo.

A partir daí, caso for achada uma seqüência de instruções que esteja em algum grupo, o analisador pode substituí-la por um novo *opcode*. Esta lista de novos *opcodes* encontra-se em um outro arquivo, onde cada seqüência a ser substituída possui estas linhas:

```
+ iload bipush isub istore
T 4
= 0xd1
```

Se for achada esta seqüência de instruções do exemplo, do grupo “4”, ela será substituída pelo *opcode* 0xd1. Salienta-se que quando há, por exemplo, um iload, não há distinção entre este iload ou suas formas reduzidas: iload_0, iload_1, iload_2. Para efeitos de substituição, eles são tratados da mesma forma. Esta instrução terá três dados imediatos. Um índice do iload, o dado imediato que bipush carrega, e o índice de istore.

Depois de substituídas todas as seqüências por um novo código, o analisador ainda faz a relocação de todos os desvios e invocações de método. Então, entrega a saída pronta para já ser usada na suposta arquitetura cuja máquina de controle suporta *folding*, além de apresentar a porcentagem de instruções que foram substituídas, sendo possível assim saber quanto o código diminuiu.

5.4 Análise do custo-benefício para a implementação de *folding*

O analisador apresentado na seção anterior foi estendido para fazer a análise do *trace* dinâmico de um programa Java, gerado pelo simulador CACO-PS. Desta forma, é possível saber exatamente, em número de ciclos, o ganho de performance do conjunto de *benchmarks* adotado neste trabalho. A Figura 5.8 mostra um exemplo da saída do programa, quando colocado um *trace* dinâmico para análise.

Como pode ser observado, primeiramente é mostrado o número total de instruções executadas. Após, todas as seqüências de instruções que são candidatas a *folding*. Somando-se a isso, o programa também faz a análise do uso de getstatic na seqüência de instruções para *folding*. São duas as razões para isso: a primeira é que a instrução getstatic usa a unidade de *load/store*, no quarto estágio, e sendo assim, esta instrução não poderia entrar em um grupo de *folding* com uma instrução que também utilize o quarto estágio, como a de soma, por exemplo. Como conseqüência, a instrução getstatic ficaria praticamente fora de qualquer seqüência de *folding*. Este primeiro problema teria uma solução, que seria modificar a estrutura do *pipeline* do Femtojava *Low-Power* para colocar a unidade de *load/store* em algum estágio anterior. O segundo problema provém do uso de dois getstatic em um mesmo conjunto de *folding*: a memória principal precisaria ter duas portas de leitura, aumentando a área do processador.

```
Soma de pontos flutuantes:

Total de instrucoes executadas: 5314

iload bipush iushr istore = 25
iload iload isub istore = 6
iload bipush ishl istore = 3
iload iload if_icmpne  = 27
iload iload iand  = 25
getstatic iload iadd  = 25
getstatic iload iaload  = 24
sipush bipush if_icmplt  = 1
iload bipush iaload  = 6
iload iload isub  = 6
iload iload ishl  = 6
iload iload iushr  = 3
iload ifgt   = 1
getstatic ishl   = 50
getstatic ifne   = 21
iload ixor   = 25
bipush ixor   = 27
iload ifne   = 55
getstatic iand   = 50
bipush iadd   = 1
bipush isub   = 7
bipush ishl   = 9
iload ior   = 16
bipush ifne   = 6
iload iushr   = 3
bipush istore   = 46
iload istore   = 41
iand istore   = 50
ior istore   = 5
iadd istore   = 6

sem getstatic: 548 / 10,31%
1 getstatic: 767 / 14,4%
2 getstatics: nao tem
```

Figura 5.8 : Resultado do programa analisador de *folding* para um exemplo

Então, a análise foi classificada de três formas diferentes: *sem getstatic*: a forma mais realista e que poderia ser implementada sem grandes alterações no processador Femtojava *Low-Power*; *1 getstatic*: nesta implementação, a estrutura do processador teria que ser mudada, a fim de tirar a busca de dados na memória do quarto estágio, como já explicado; *2 getstatic*: a implementação mais cara de todas: além da mudança da arquitetura, duas unidades de *load/store* com uma memória de duas portas seriam necessárias. A última é a mais otimista de todas, considerando 2 instruções getstatic no grupo de *folding*, procedimento este não adotado por trabalhos anteriores.

A Tabela 5.5 apresenta um resumo do conjunto de *benchmarks* deste trabalho. A primeira coluna apresenta cada programa executado, a segunda o número de instruções necessárias para sua execução. As colunas seguintes apresentam, conforme as restrições da instrução getstatic explicadas anteriormente, o número de instruções das quais foram feitas *folding*. As colunas com o sinal % apresentam o quanto o programa, proporcionalmente ao número de instruções executadas, foi mais rápido.

Como pode ser observado, na melhor das hipóteses, permitindo duas instruções getstatic em uma mesma seqüência de *folding*, há uma redução de apenas 14,77% em média no número de instruções a serem executadas de todos os algoritmos testados. Como não houve ganhos substanciais, a implementação desta técnica no processador Femtojava *Low-Power* foi abandonada, apesar de algumas instruções terem sido implementadas e simuladas como forma de teste.

Tabela 5.5 : Resultado da análise do conjunto de instruções para *folding* no conjunto de *becnhmark* utilizado neste trabalho

Programa	***Instruções executadas***	***Sem getstatic***	***%***	***1 getstatic***	***%***	***2 getstatic***	***%***
BubbleSort	1123	73	6,50	140	12,47	140	12,47
SelectSort	952	174	18,28	183	19,22	183	19,22
QuickSort	694	154	22,19	154	22,19	154	22,19
InsertSort	708	97	13,70	113	15,96	138	19,49
SinCordic	442	76	17,19	88	19,91	88	19,91
Float Soma	5314	548	10,31	767	14,43	767	14,43
IMDCT	25478	1964	7,71	2000	7,85	3260	12,80
Busca Binária	42	4	9,52	4	9,52	4	9,52
Busca Seqüencial	375	66	17,60	66	17,60	66	17,60
Seqüencial Ordenada	165	2	1,21	5	3,03	5	3,03
Hashing	17	2	11,76	2	11,76	2	11,76
Média			***12,36***		***14,00***		***14,77***

6 FEMTOJAVA VLIW

O processamento paralelo de instruções em um processador vem sendo bastante utilizado para aumentar o desempenho dos programas executados. Atualmente existem no mercado vários processadores classificados como superescalares. Estes processadores fazem uma análise dinâmica das instruções que são independentes entre si para que possam ser executadas ao mesmo tempo. Consequentemente, há um custo adicional, tanto em área quanto em potência consumida, da lógica – relativamente complexa – para fazer esta análise.

Seguindo a mesma filosofia de execução de instruções em paralelo, há uma segunda opção: utilizar-se da técnica de VLIW (*Very Long Instruction Word*). A diferença básica entre esta técnica e a superescalar é que a segunda faz a análise (ou agendamento) em tempo de execução (Figura 6.1a), enquanto a primeira faz de forma estática, durante a compilação (Figura 6.1b). A origem do nome vem exatamente desta análise estática, já que o compilador une instruções que possam ser executadas simultaneamente em um “pacote”, ou instruções bastante longas, que geralmente são de 64 ou 128 bits.

Cada instrução neste pacote tem seu caminho, ou fluxo, previamente definido. Assim, é possível configurar restrições que podem refletir na economia de área e potência. Pode-se definir que, como exemplo, determinado fluxo não possui um multiplicador. Assim, instruções que ocupam o local daquele fluxo no pacote não podem ser instruções de multiplicação. Da mesma maneira, pode-se compartilhar recursos entre fluxos. Poderia-se compartilhar um deslocador entre todos os fluxos, considerando um processador VLIW de 4 instruções por pacote. Então, só seria permitido haver uma instrução de deslocamento em todo o pacote ou, em outras palavras, execuções de deslocamento não poderiam ser executadas em paralelo.

Apesar de nem sempre conseguir obter o mesmo paralelismo que processadores superescalares conseguem (por exemplo, há referências de memória que só são resolvidas durante a execução do programa e consequentemente é impossível paralelizar instruções que as usam durante a compilação), a VLIW mostra-se bastante eficiente e satisfatória. Particularmente no ramo dos sistemas embarcados, onde muitas vezes os programas executados são sempre os mesmos, gravados em algum tipo de ROM, a técnica VLIW apresenta uma grande vantagem: os custos de área e potências associados à análise dinâmica praticamente inexistem.

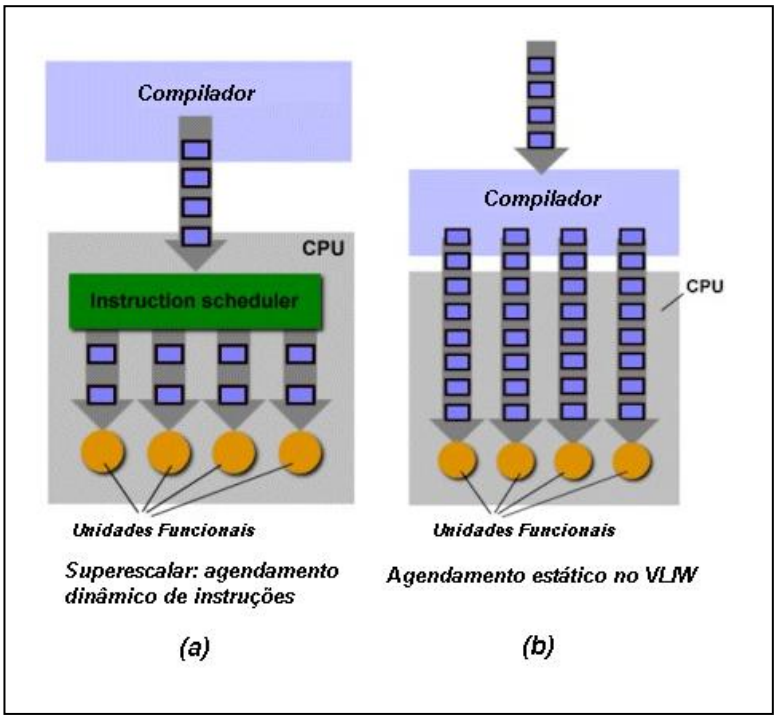

Figura 6.1 : Como funciona o agendamento em processadores Superescalares e VLIW

A versão *Low-Power* do Femtojava, apresentada no capítulo 4, consegue executar programas com um melhor desempenho e com menor energia consumida do que a versão original, a Multiciclo. Mesmo assim, algoritmos mais complexos e com restrições de tempo mais rígidas podem necessitar de um desempenho ainda maior na execução nativa de código Java. Com o intuito de lidar com estes programas, uma versão VLIW do processador Femtojava (BECK; CARRO, 2004) (BECK; CARRO, 2004b), projetada a partir da versão *Low-Power*, começou a ser desenvolvida. Além do mais, características específicas para a procura de paralelismo para processadores de pilha também foram estudadas.

Neste capítulo é mostrada a construção do analisador de *bytecodes* que transforma o programa Java em um conjunto de instruções VLIW. O analisador é parametrizável, podendo gerar palavras com tamanho variável de instruções. O analisador é o ponto crítico no desenvolvimento da versão VLIW, já que a sua eficiência é imprescindível para o bom funcionamento do *hardware* desta versão. Após, um estudo sobre a implementação do *hardware* do VLIW é apresentado, mostrando os resultados para diferentes versões através de simulação.

6.1 Análise e geração do código VLIW

Os passos desempenhados pelo programa analisador são demonstrados nesta seção, desde a análise dos *bytecodes* em busca de paralelismo de instruções até a geração do código.

6.1.1 Busca e identificação de blocos básicos e métodos

Após ter lido o arquivo original de programa, no formato MIF (como mostrado no anexo II), e colocado as instruções em uma estrutura em memória, o primeiro passo do analisador é achar os métodos e blocos básicos. Conforme (AHO; SETHI; ULLMAN, 1995), um bloco básico é "uma seqüência de enunciados consecutivos, na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final". Blocos básicos são facilmente identificados dentro de laços, por exemplo. Uma vez identificado o bloco básico, há a certeza de que aquele trecho do código será executado de forma seqüencial.

Todo o programa é colocado em uma estrutura de memória, como já mencionado. Sendo assim, é possível atribuir valores e marcas (*flags*) para cada instrução. Há uma marca que indica o limite de um bloco básico. Então, a procura por um bloco básico é feita de forma bem simples: o programa lê as instruções sequencialmente procurando por uma instrução de desvio (condicional ou incondicional) ou de chamada ou retorno de método. Além do mais, dentre os valores guardados na estrutura para cada instrução, estão seus dados imediatos e endereços de chamadas de métodos.

Há ainda uma outra marca associada a cada instrução, e serve para indicar se o valor corrente é uma instrução ou na verdade um parâmetro de um método. Cada método encontrado no conjunto de *bytecodes* gerados pela ferramenta Sashimi possui sempre dois parâmetros, antes de suas instruções em si: o número de variáveis locais ocupada por este método e o número de parâmetros passados pelo método anterior. Estes dados facilitam o controle do processador na chamada e retorno de métodos. Assim, a marca serve para indicar futuramente ao analisador para não considerar tais parâmetros como instruções. Por exemplo, um parâmetro de valor `0x02` poderia ser considerado erroneamente como a instrução `iconst_02` na hora da análise.

6.1.2 Análise de endereços para relocação

O segundo passo é analisar todo o código novamente a procura de desvios ou chamada de métodos. A cada desvio ou método encontrado, para o seu endereço de destino, um valor único é dado. Cada instrução possui em sua estrutura dois valores chamados *origem* e *alvo*. A instrução que faz o desvio ganha um número identificador em sua variável *origem*. A instrução onde resulta o desvio ganha o mesmo valor, entretanto no campo *alvo*. Isto é necessário porque, após o *bytecode* paralelizado estiver pronto, é muito provável que as instruções mudem de lugar, e sendo assim, a relocação dos endereços de desvio e chamadas de método precisa ser efetuada a fim de garantir o funcionamento correto do programa. Identificados a origem e destino previamente e relacionando-os às instruções, basta depois, para cada instrução que tiver um número único de origem, buscar por seu alvo correspondente e fazer o ajuste do endereço.

6.1.3 Análise do código em busca de paralelismo

Após todos os dados do programa terem sido analisados e colocados em uma estrutura de memória, como as marcas dos blocos básicos e métodos, dados de origem e alvo de desvios para futura relocação, o código é analisado novamente para ser paralelizado. A busca de paralelismo é feita dentro dos blocos básicos, utilizando as marcas previamente configuradas para a detecção. Contudo, a busca de paralelismo em processadores baseados em pilha é diferente de processadores RISC.

Considerando seguinte seqüência de instruções do tipo RISC:

```
add r1, r5, r3
mul r2, r3, r5
```

O resultado da soma dos valores encontrados em r5 e r3 é colocado em r1. Da mesma forma, o resultado da multiplicação entre os valores de r3 e r5 é gravado em r2. As duas instruções poderiam ser colocadas em paralelo, já que a instrução add não utiliza o resultado de mul e vice-versa.

Todavia, em processadores de pilha, esta busca é feita de forma diferente. Ao contrário de uma instrução do tipo RISC, como demonstrada anteriormente, onde a soma é indicada por uma instrução e três registradores, dois de origem e um de destino, a soma em um processador Java, de pilha, funciona da seguinte maneira:

```
bipush op1
bipush op2
iadd
istore
```

Onde primeiro são empilhados os valores op1 e op2 na pilha de operandos. Estes operandos são somados pela instrução iadd, e depois podem ser gravados no depósito de variáveis locais do método, na memória principal, ou ainda pode ser deixado na própria pilha para uma futura computação.

A busca por paralelismo em um programa Java é feita diretamente nos *bytecodes*. O algoritmo funciona da seguinte maneira: todas as instruções que dependem do resultado da anterior são agrupadas em um conjunto chamado neste trabalho de bloco de operações. Todo o programa Java é dividido em grupos como este. Estes grupos podem ser paralelizados respeitando o número de unidades funcionais existentes. Por exemplo, se há apenas um multiplicador, duas instruções que usam esta unidade funcional não podem ficar em paralelo. Um exemplo deste procedimento é apresentado na Figura 6.2.

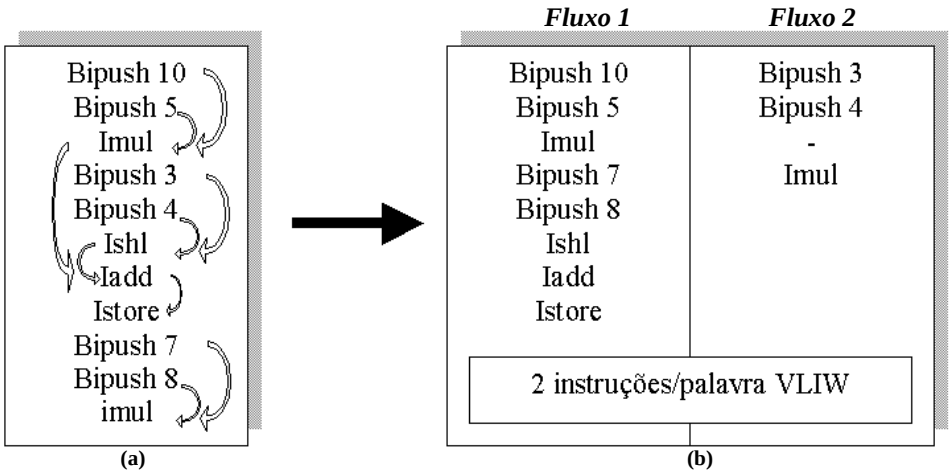

Figura 6.2 : O processo de formação da palavra VLIW

No exemplo na Figura 6.2a, a primeira instrução imul (multiplicação) irá consumir os operandos empilhados previamente pelos dois bipush (bipush 10 e bipush 5). Depois disso, a instrução ishl (deslocamento para esquerda) irá consumir outros dois operandos empilhados pelos próximos bipush (bipush 3 e bipush 4). Após, a instrução iadd (soma)

irá consumir os resultados de imul e ishl. Finalmente, a instrução istore irá salvar o resultado da instrução iadd no depósito de variáveis locais. Como há uma relação entre todas estas instruções, isto é, umas usam os resultados das operações das anteriores, elas formam um bloco de operação. Outro bloco de operação é formado pelas duas últimas instruções bipush (bipush 7 e bipush 8) e imul, já que estas instruções apenas dependem entre si e não utilizam nenhum resultado das instruções do bloco anterior. Em outras palavras, as pilhas de operandos destes dois blocos são independentes. Conseqüentemente, eles podem ser paralelizados, como pode ser observado na Figura 6.2b. É importante perceber que as duas instruções imul não podem ficar no mesmo pacote VLIW, já que está sendo considerado neste exemplo que o processador VLIW possui apenas um multiplicador.

Para detectar os blocos de operação no programa, as instruções são classificadas em grupos. A divisão dos grupos é baseada nas instruções que consomem operandos e quantos são consumidos, daquelas que produzem operandos e daquelas que consomem e produzem. Então, cada vez que uma instrução é encontrada, é verificado o seu grupo. Sabendo-se quantos operandos aquela instrução consome ou produz, é possível montar cada bloco de operação.

Além do mais, cada instrução é classificada dentro de uma unidade funcional. A instrução imul, por exemplo, é classificada na unidade funcional multiplicador. As instruções swap, iadd, isub, ineg, ishl, ishr, iushr, iand, ior, ixor, iinc são classificadas na unidade funcional ALU. Classificando as instruções desta maneira, é possível restringir as instruções paralelizadas geradas no pacote. No exemplo acima, a instrução imul não pode ser colocada em paralelo com a outra instrução idêntica pois havia apenas um multiplicador. Em todos os exemplos considerados neste trabalho sempre pressupõe-se que há apenas um multiplicador, mas que há para cada fluxo restante todas as outras unidades funcionais disponíveis. Esta restrição visa, além da economia em área, já que o multiplicador é a maior de todas as unidades funcionais, validar o analisador quando se impõe restrições no número de unidades funcionais.

6.1.4 Alinhamento de instruções que acessam a memória

Como observado em (TIWARI; MALIK; WOLFE, 1994) (SIMUNIC; MICHELI; BENINI, 1999)(CHEN et al., 2002), uma das maiores causas de consumo de energia são os acessos à memória. Assim, outra otimização nos *bytecodes* Java foi feita preocupando-se com este problema. Depois da procura por paralelismo, outra busca é feita: instruções que lêem a memória principal (getstatic) são alinhadas na mesma palavra VLIW. Isto só ocorre se elas buscam o valor no mesmo endereço e entre estas duas instruções de busca na memória o valor a ser buscado não muda, isto é, não há escrita naquele endereço de memória. Conseqüentemente, com as instruções alinhadas, apenas um getstatic é executado, passando diretamente o seu resultado para as outras instruções getstatic que estão na mesma palavra, economizando assim potência decorrente dos outros getstatic que não precisaram ser executados. A freqüência de ocorrência desta situação depende muito do programa executado. Em programas que possuem muitas variáveis estáticas, por exemplo, a propensão de acontencimento destes alinhamentos são maiores. A Figura 6.3 ilustra o procedimento.

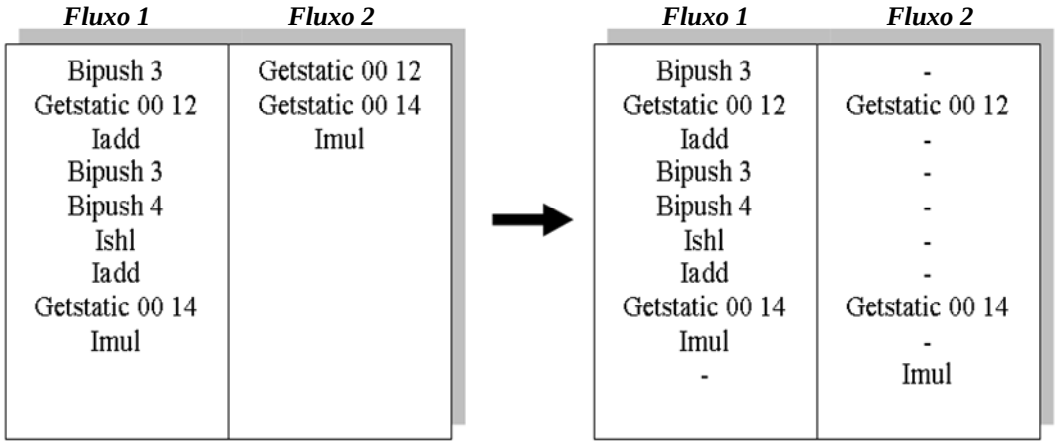

Figura 6.3 : Processo de alinhamento dos acessos à memória

O analisador possui um arquivo de configuração para auxiliar na parametrização da geração. A partir deste arquivo, é possível definir o tamanho do pacote VLIW em número de instruções, se deve haver ou não alinhamento de memória e em quantos *bytecodes*, quando achada uma instrução getstatic, o analisador vai fazer a procura para achar uma outra instrução idêntica que faz a busca no mesmo endereço. Na Figura 6.3, se essa busca estivesse configurada para, por exemplo, 5 bytecodes, a instrução getstatic 00 14 no segundo fluxo não seria alinhada com a instrução do primeiro, pois o analisador iria procurar uma instrução igual apenas até a instrução iadd. O valor padrão adotado foi de 15 *bytecodes*, valor que se demonstrou satisfatório para o conjunto de exemplos utilizado.

É neste arquivo de configuração que é possível classificar as instruções dentro de unidades funcionais, como já explicado. Além do mais, é possível informar quais unidades funcionais e em qual número elas serão encontradas no processador VLIW. Pode-se, por exemplo, configurar o gerador para considerar haver apenas uma ALU. Como conseqüência, todas as operações aritméticas e lógicas não poderão se encontrar no mesmo pacote VLIW. Esta opção é bastante interessante para analisar o impacto no desempenho do paralelismo quando área é economizada em função da limitação na quantidade de unidades funcionais.

6.2 O processador VLIW

O processador que está sendo desenvolvido é basicamente uma extensão do Femtojava *Low-Power*, com suas unidades funcionais e decodificadores de instrução replicados (BECK; CARRO, 2004b). Os decodificadores adicionais não suportam as instruções de chamada e retorno de métodos, já que as instruções responsáveis por estas funções estão sempre no fluxo principal. O depósito de variáveis locais encontra-se apenas no banco de registradores do primeiro fluxo. Quando instruções de outros fluxos precisam ler ou escrever algum valor de uma variável local do método, elas precisam buscar deste local. Cada fluxo de instrução tem seu próprio banco de registradores para a pilha de operandos, que por sua vez possuem menos registradores que o banco principal, já que a pilha de operandos destes fluxos secundários não cresce tanto quanto a primeira.

O tamanho do banco de registradores dos fluxos secundários é de oito registradores, já que a pilha de operandos não cresce mais do que isso nestes fluxos, considerando o conjunto de *benchmarks* utilizado, nem é necessário guardar nenhuma variável local. A palavra VLIW tem um tamanho variável, evitando acessos desnecessários à memória. Um cabeçalho na primeira instrução da palavra informa para o controlador de busca de instruções quantas instruções a palavra corrente possui.

Uma das grandes vantagens do processador de pilha é a maneira intrínseca na linguagem de como os blocos de operandos se comunicam entre si. Em processadores VLIW convencionais, geralmente é necessário um sistema de comunicação entre as unidades funcionais para o repasse de valores entre as mesmas. Entre as soluções encontradas estão o uso de barramentos ou de *crossbars*. Entretanto, tanto uma solução quanto a outra tornam mais complexo todo o controle do processador para manter a consistência dos dados. Há ainda a opção de usar um banco de registradores compartilhado. Contudo, instruções adicionais para a sincronização dos fluxos são necessárias para manter a conscientência dos dados.

Em Java, sempre quando um bloco de operação chega ao final, se necessário, seu resultado é gravado no banco de variáveis locais, que pode ser compartilhado por todos os fluxos no futuro. A característica básica do bloco de operação é exatamente esta: ele põe operandos na pilha e os consome da mesma maneira (o SP original é igual ao SP final), gravando o seu resultado no banco de variáveis locais, caso preciso.

Se um fluxo de execução necessitar de algum resultado de um bloco de operação que se encontra em um outro fluxo, basta acessar o banco de registradores do fluxo principal em busca das variáveis locais, que possui como incremento apenas uma porta de leitura a mais destinada para este fim (os acessos às variáveis locais não podem ser feitos de maneira simultânea). Nenhum outro controle é necessário, barramento nem *crossbars* complexos, já que a linguagem Java suporta instrisicamente esta comunicação.

A Figura 6.4a mostra uma seqüência de código onde três blocos de operação estão destacados. Conforme observado na Figura 6.4b, esta seqüência é paralelizada para um processador VLIW com duas instruções por pacote. Nota-se que depois que as instruções foram paralelizadas, o terceiro bloco de operação que está no primeiro fluxo necessita de um resultado do segundo bloco de operação, que por sua vez, encontra-se no segundo fluxo. Como o resultado foi escrito no banco de registradores, basta o terceiro bloco, através da instrução `iload_1` (pois o bloco que está no segundo fluxo gravou o seu resultado com a instrução `istore_1`), fazer esta busca. Nenhuma alteração no compilador ou no analisador são necessárias para a comunicação entre os fluxos.

É importante ressaltar que estas características facilitam o desenvolvivento de um programa analisador de paralelismo dos *bytecodes* Java. Não é necessário nenhum cuidado especial para tratar com a comunicação entre os fluxos, e para construir os blocos de operação é apenas necessária a informação, para cada instrução, de quantos operandos são consumidos ou produzidos na pilha.

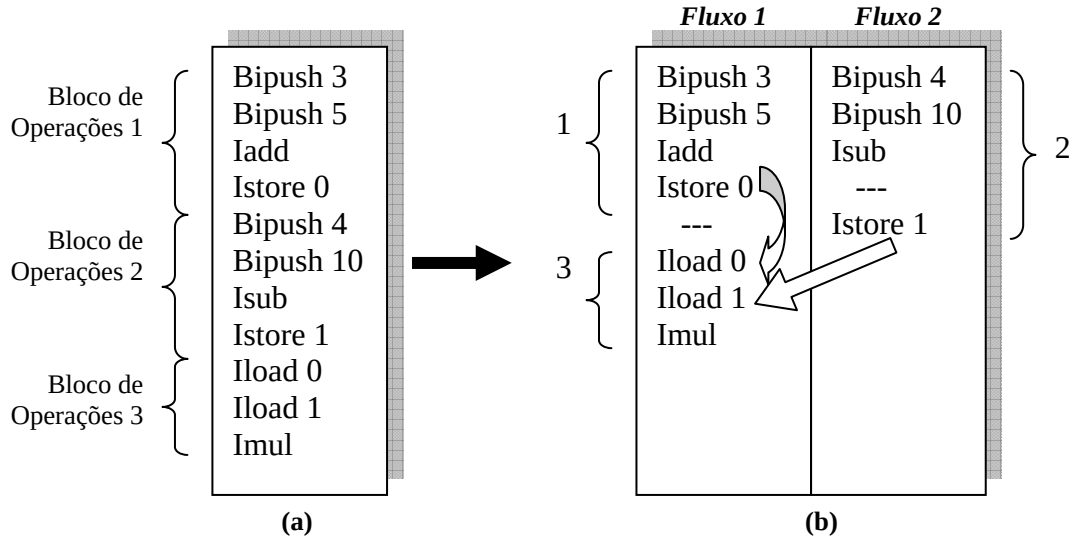

Figura 6.4 : Seqüência de códigos paralelizada: Comunicação intrínseca entre os fluxos

6.3 Resultados Parciais

Para chegar a resultados parciais, o analisador VLIW já produzido foi estendido para agir sobre *traces* dinâmicos gerados pelo CACO-PS de programas executados no processador Femtojava. Em posse desse *trace*, é feita uma paralelização das instruções. Como a paralelização é feita no trace dinâmico, observa-se exatamente o comportamento, em termos de performance, de como se fosse executado em um *hardware* VLIW.

A medição de energia foi baseada na técnica de instruções (TIWARI; MALIK; WOLFE, 1994). Com posse do trace paralelizado, varre-se cada fluxo da palavra VLIW. Para cada instrução lida, um valor de energia, dividido em núcleo, memória RAM e ROM, é dado. Esses valores foram obtidos da seguinte maneira: para cada instrução é executado um programa com várias delas com diferentes operandos na versão *Low-Power* do Femtojava. Depois, é feita uma média de energia gasta por cada uma e estes valores são guardados em uma tabela, que será usada mais tarde pelo analisador. Fazendo-se a simulação convencional para o CACO-PS e comparando com a simulação por instruções, a taxa de erro foi menor que 2% no processador Femtojava *Low-Power*. Para o fluxo principal VLIW foi considerado uma energia igual à do Low-Power. Para os secundários, foi considerado menor tamanho do banco de registradores assim como os decodificadores que não suportam desvios nem retornos de métodos.

A área foi estimada a partir dos componentes já prontos em VHDL. Como já é sabido exatamente quais componentes são necessários para cada fluxo, a estimativa em células lógicas pôde ser feita. Além do mais, já está em desenvolvimento uma descrição do VLIW de 2 instruções por pacote no CACO-PS. Esta versão ainda não suporta chamadas e retornos de métodos.

A Tabela 6.1 mostra a estimativa de área ocupada em células lógicas em diferentes versões do processador Femtojava VLIW. A estimativa é feita para diferentes números de instruções por pacote VLIW, de duas instruções até oito. Ainda, é feita a estimativa quando um multiplicador é compartilhado por todos os fluxos ou quando há um multiplicador para cada fluxo. Finalmente, há a contagem considerando os bancos de registradores implementados em memória ou em células lógicas. Nota-se que assim,

para cada versão do processador VLIW relacionada ao número de instruções por palava, há quatro possibilidades de implementação. Salienta-se, observando a Tabela 6.1, que a versão que ocupa mais área é a versão VLIW com 8 instruções por pacote, com todos os bancos de registradores implementados em células lógicas e sem multiplicador compartilhado.

Tabela 6.1 : Estimativa de área para diferentes versões do Femtojava VLIW (células lógicas)

Número de Instruções por Pacote VLIW	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Sem multiplicador compartilhado e banco de registradores em células lógicas	6110	8308	10505	12703	14901	17099	19297
Com multiplicador compartilhado e banco de registradores em células lógicas	5693	7474	9255	11036	12817	14597	16378
Sem multiplicador compartilhado e banco de registradores em memória	4626	6701	8775	10850	12924	14999	17074
Com multiplicador compartilhado e banco de registradores em memória	4209	5867	7524	9182	10840	12498	14155

A Figura 6.5 mostra, de forma gráfica, uma comparação entre as versões Multiciclo, *Low-Power* e algumas versões VLIW. Ressalta-se que estas versões VLIW apresentadas são as que ocupam mais área, com registradores implementados em células lógicas e sem multiplicador compartilhado. Entretanto, para todos os experimentos feitos, o multiplicador é compartilhado. Isto traz como benefício uma boa economia de área. Contudo, a performance é um pouco degradada, pois não é possível que instruções de multiplicação sejam executadas em paralelo.

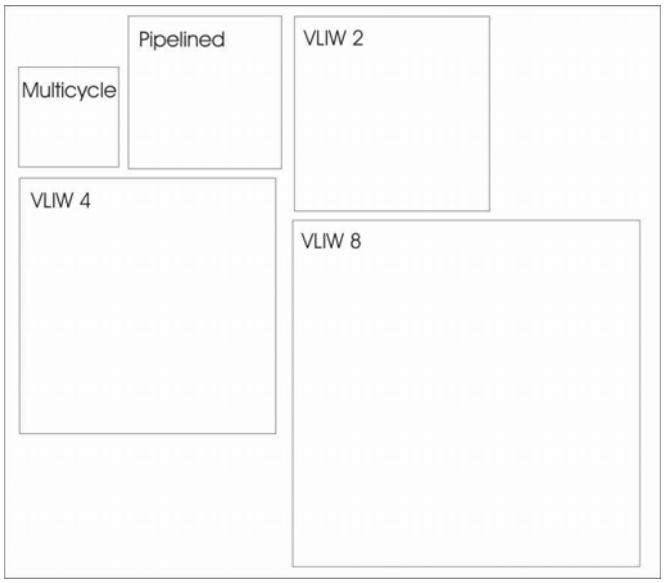

Figura 6.5 : Comparação gráfica da área das diferentes versões do processador Femtojava

A Tabela 6.2 mostra a performance de cada algoritmo do conjunto de *benchmarks* escolhido em cada arquitetura, em número de ciclos. Pode-se observar que em alguns algoritmos executados na versão VLIW, o resultado em número de ciclos é o mesmo. Isto acontece porque mesmo tendo mais instruções por palavra disponível para a paralalelização, o limite de paralelismo já foi alcançado com menos instruções. Os números mostrados do Multiciclo e *Low-Power* foram obtidos simulando-se as respectivas descrições no simulador CACO-PS. Os dados do processador VLIW foram obtidos usando o analisador VLIW estendido, como já explicado anteriormente.

Tabela 6.2 : Performance das diferentes arquiteturas, em número de ciclos

Algoritmo	*Número de Ciclos*				
	Multiciclo	*Low-Power*	*VLIW*		
			2	*4*	*8*
Seno	2447	755	599	592	583
Ord./Bubble	6950	2424	2104	1967	1967
Ord./Select	5335	1930	1707	1670	1670
Ord./Insert	5111	1934	1601	1331	1331
Busca Binária	1162	403	368	365	365
Busca Sequencial	7586	1997	1775	1775	1775
IMDCT	140300	40306	33050	32994	32994
IMDCT u1	97354	31500	19325	12313	9944
IMDCT u2	92882	30369	18689	11737	9432
IMDCT u3	51345	18858	12789	8929	7741
Soma ponto flutuante	30747	14531	12474	12313	12313

Como podem ser observados na Tabela 6.2, melhores resultados são alcançados quando executadas versões com laços desenrolados da IMDCT (IMDCT u1, IMDCT u2 e IMDCT u3). A razão é a diminuição de desvios condicionais. Como conseqüência, reduz-se o número de ciclos pagos por laços que são verdadeiros, três, no caso do Femtojava, para encher novamente a fila de instruções. Neste processador, sempre é considerado que a condição de um salto não é verdadeira, por questões de facilidade de implementação. Para a versão VLIW, além disso, a grande vantagem é que o uso desta técnica expõe mais o paralelismo dos *bytecodes* para o analisador, já que o tamanho dos blocos básicos aumenta significativamente. A principal desvantagem das versões desenroladas é o aumento no tamanho do programa, e conseqüentemente, da memória de instruções.

Operando na mesma freqüência, as arquiteturas VLIW são as que possuem um maior consumo de potência por ciclo no núcleo, já que elas são as mais complexas, com mais unidades funcionais e registradores. Este comportamento pode ser observado na Figura 6.6 (onde VLIW 2 significa 2 instruções por palavra e assim por diante).

A potência consumida por ciclo nas arquiteturas VLIW é muito maior que nas arquiteturas Multiciclo e *Low-Power*, principalmente nas versões com mais instruções por palavra, onde os fluxos, mesmo quando não usados, gastam potência com os componentes seqüenciais. Porém, quando mensurada a energia total do núcleo das arquiteturas a diferença diminui consideravelmente, principalmente por causa do aumento da média de IPC executada. Desta forma, os componentes do processador são melhores aproveitados ao decorrer do tempo. Ganhos de energia podem ser observados principalmente nas versões de 2 e 3 instruções por palavra nos algoritmos desenrolados da IMDCT.

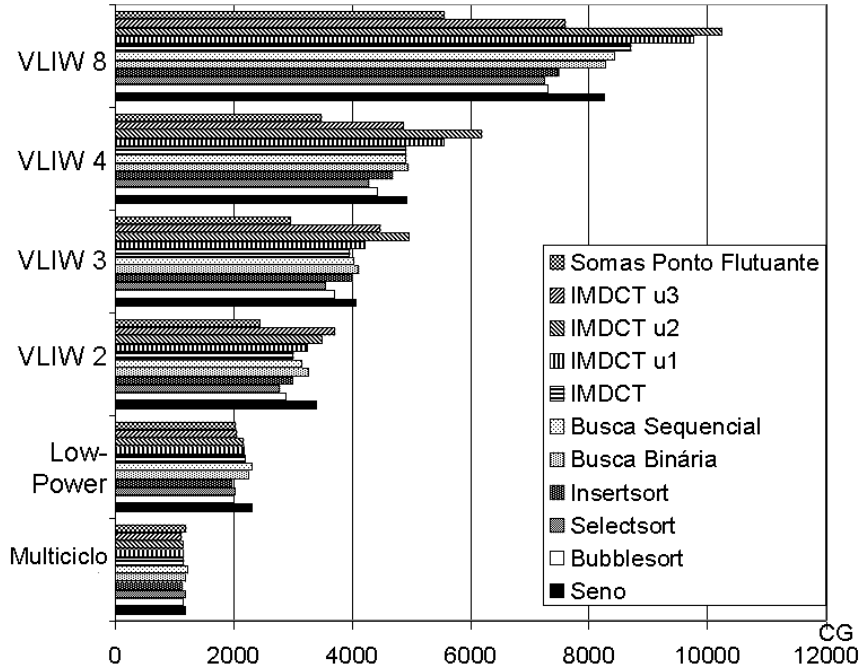

Figura 6.6 : Potência consumida por ciclo no núcleo em cada algoritmo nas diferentes arquiteturas

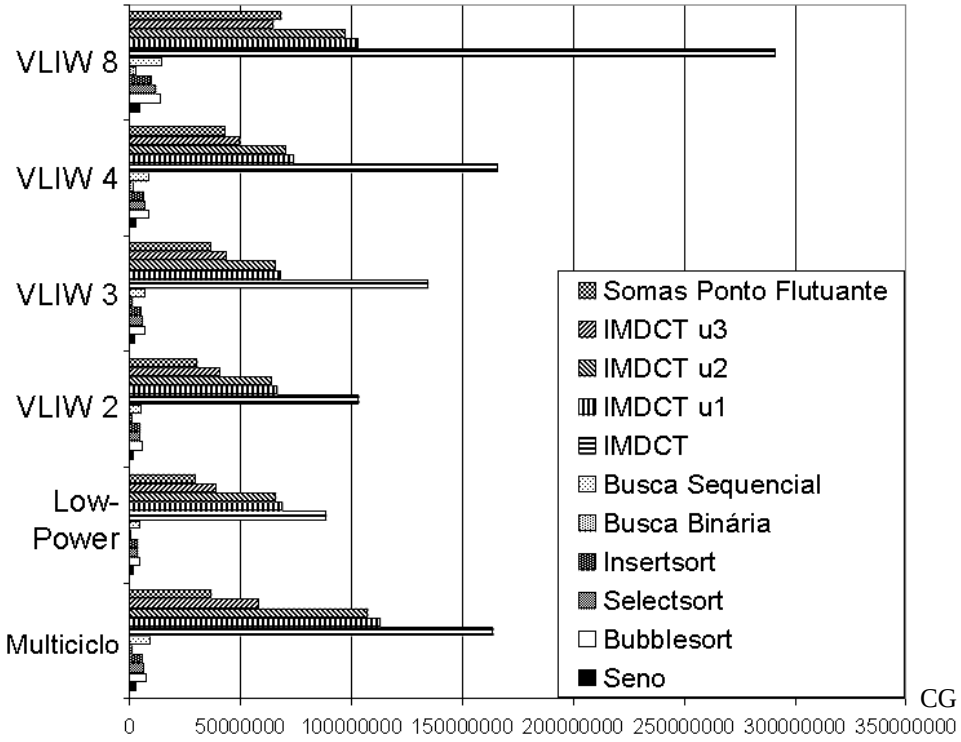

Figura 6.7 : Total de energia consumida no núcleo por arquitetura

A versão Multiciclo usa a memória principal como pilha de operandos e depósito de variáveis locais do método. Há uma grande diferença em termos de consumo de energia entre esta arquitetura e a *Low-Power*, com *pipeline*. Esta, por sua vez, somente acessa a memória em chamadas e retorno de métodos ou em instruções específicas, como getstatic e putstatic. A Figura 4.12 no capítulo 4 mostra a vantagem de implementar a pilha de operandos e o depósito de variáveis em um banco de registrador no núcleo ao invés de usar a memória principal.

Na Figura 6.8, mostra-se a vantagem quando usando a técnica VLIW com e sem alinhamento de getstatic, como já foi explicado na seção 6.1.4. Nesta figura é demonstrada a energia total gasta com acessos à memória quando é feito o alinhamento e quando não é. Nota-se uma expressiva redução em alguns algoritmos, principalmente no filtro IMDCT. É importante notar que há uma pequena perda de paralelismo quando utilizada esta técnica, já que algumas instruções são deslocadas para os getstatic ficarem alinhados na mesma palavra. Esta perda fica em média, entre os algoritmos analisados, de 1,5%.

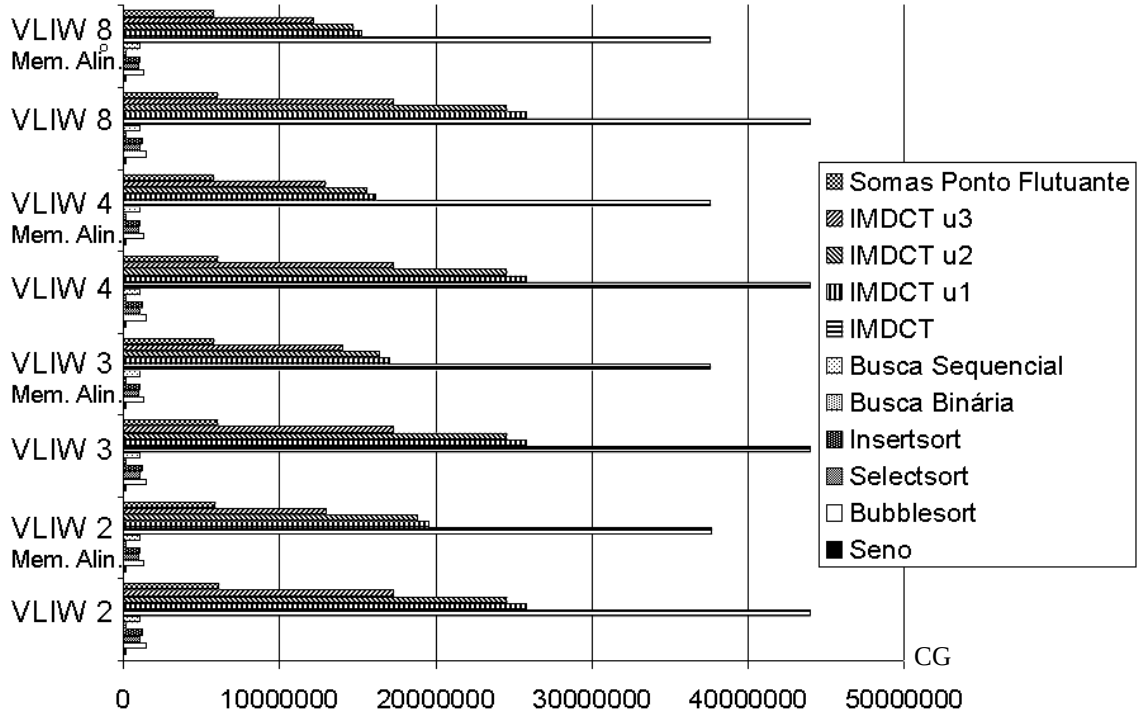

Figura 6.8 : Energia total gasta em buscas na memória para cada algoritmo e arquitetura quando a instrução getstatic é alinhada e quando não é

Finalmente, mostra-se na Figura 6.9 o consumo total de energia (considerando núcleo, RAM e ROM) entre todas as arquiteturas utilizadas. A diferença de consumo entre a versão Multiciclo e *Low-Power* tem duas razões principais: o uso do banco de registradores no núcleo (quando a versão Multiciclo tem a pilha na memória principal) e a diminuição de escritas na pilha devido ao uso da técnica de *forwarding*. Na versão VLIW, as vantagens são na utilização de bancos de registradores menores da pilha de operandos usados pelos fluxos secundários, ao invés de usar sempre o banco principal, de maior tamanho, e ainda no reaproveitamento nos acessos à memória. Contudo, este ganho apenas é visível quando os fluxos secundários são bastante utilizados, exatamente o que acontece nas versões desenroladas da IMDCT. Caso contrário, os fluxos pouco utilizados vão servir como um gasto adicional de energia, já que há vários registradores que, mesmo não sendo utilizados, consomem potência por ciclo, considerando que o relógio continua atuando em todos eles.

Já na Figura 6.10, a versão Multiciclo é modificada e tem a sua pilha também implementada em um banco de registradores, assim como o Femtojava *Low-Power*. Desta maneira, pode-se observar de maneira mais clara a diferença de energia total consumida devido às diferenças arquiteturais de cada processador.

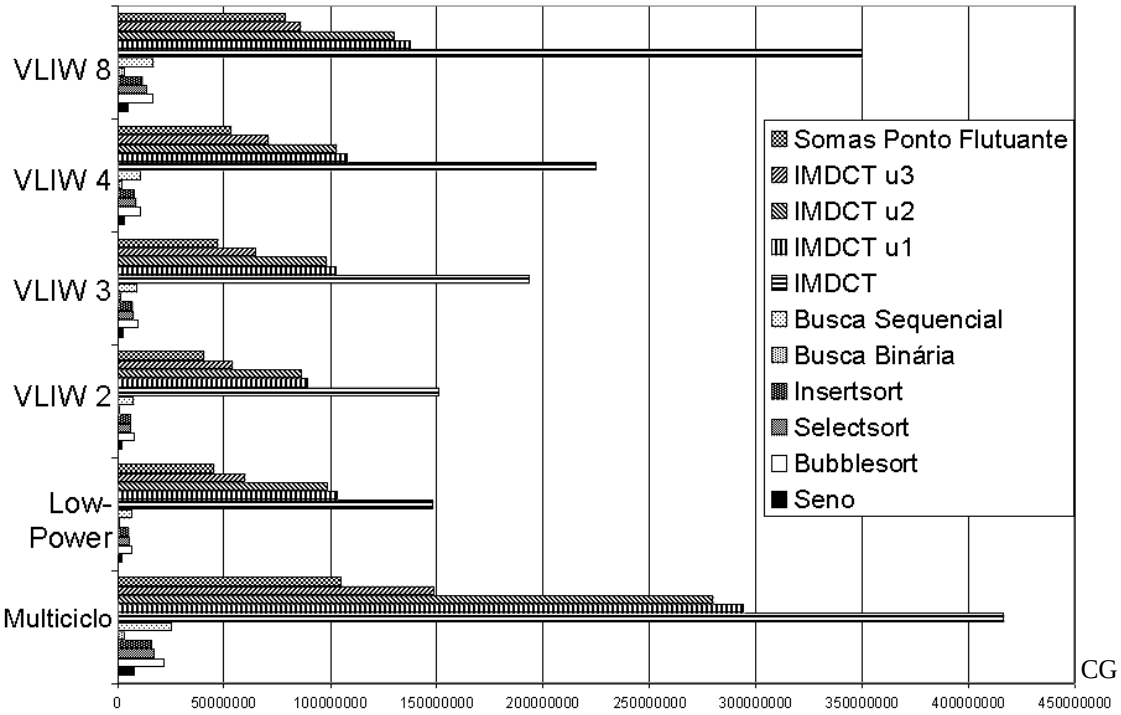

Figura 6.9 : Energia total consumida pelos algoritmos nas diferentes versões

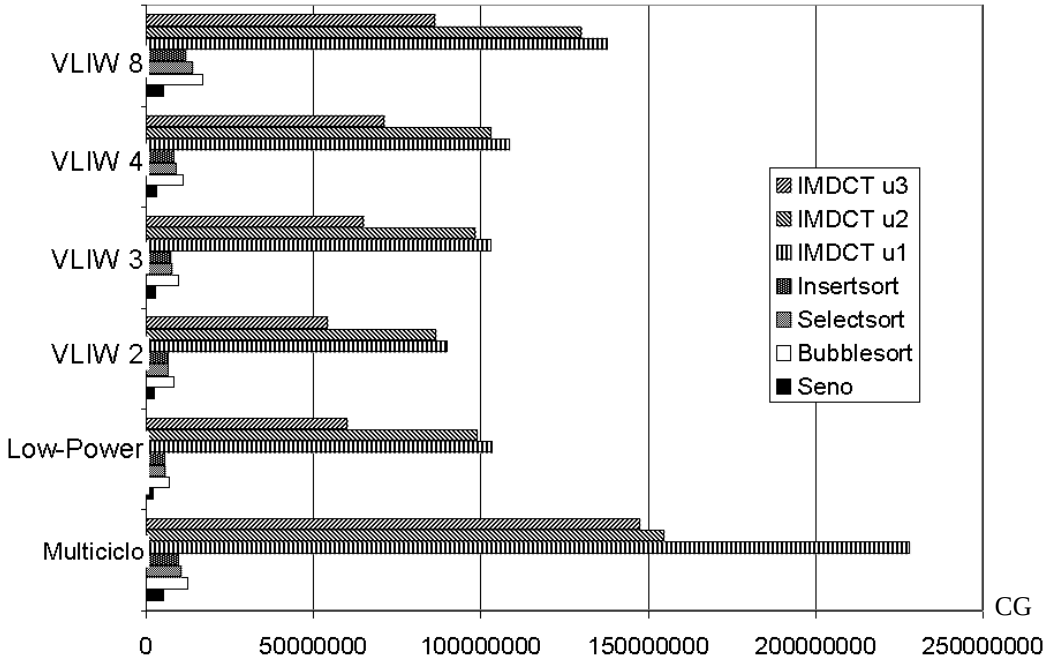

Figura 6.10 : Energia total consumida pelos algoritmos nas diferentes versões, considerando agora a versão Multiciclo que tem a pilha implementada em um banco de registradores

A Figura 6.11 mostra o resultado da utilização da mesma técnica da seção 4, quando a freqüência de operação e voltagem foram reduzidas para alcançar a mesma performance do processador multiciclo, baseados nos dados do processador Transmeta TM5400 (TRANSMETA, 2004).

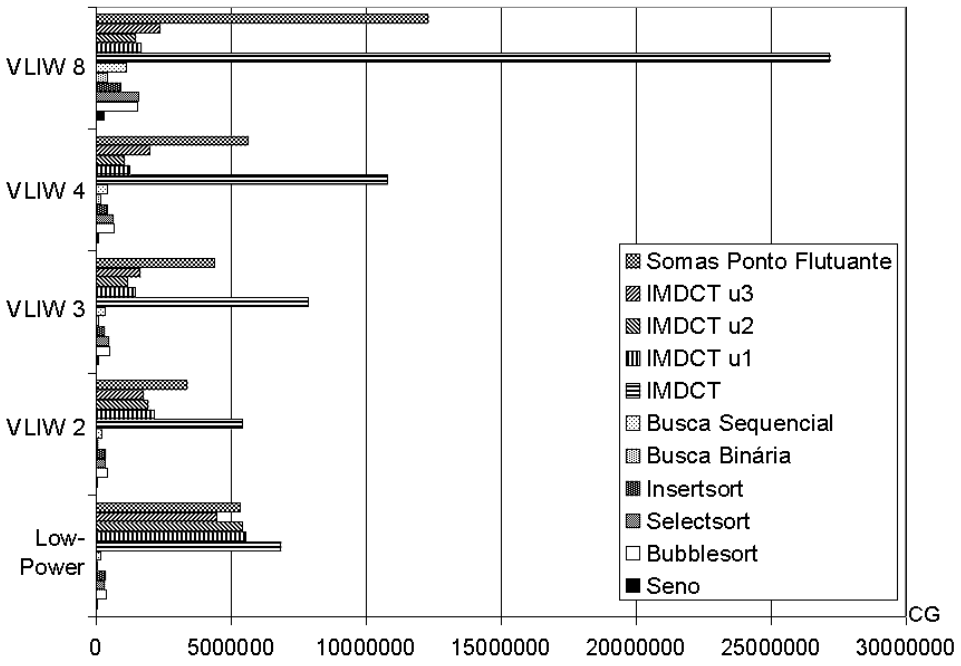

Figura 6.11 : Energia total consumida pelas versões *Low-Power* e VLIW considerando que todas possuem o mesmo desempenho

Salienta-se que para os algoritmos que tem um alto grau de paralelismo, o VLIW torna-se uma arquitetura de baixa energia, gastando ainda menos que a versão *Low-Power*. Como pode ser observado na Figura 6.11, na IMDCT u2 na versão de 4 instruções por palavra, a energia gasta é significativamente menor que a versão *Low-Power*. Quando há bastante paralelismo, os fluxos secundários são massivamente utilizados e os registradores destes fluxos não ficam consumindo energia em vão. Entretanto, há um limite. O algoritmo de soma de pontos flutuantes, por exemplo, é um intermediário. A sua execução em VLIW compensa até 4 instruções por palavra. Com mais instruções do que isso, como não há paralelismo suficiente para ser exposto, há a desvantagem de que todos os componentes seqüenciais dos fluxos adicionais estarem gastando energia. Os algoritmos de busca e de ordenação sempre saem em desvantagem em relação à energia nas versões VLIW, já que não há quase nenhum paralelismo a ser explorado neles.

No processador Femtojava, os acessos à memória são responsáveis por aproximadamente 20% da energia total gasta pelo sistema. Todavia, dependendo da estrutura, tamanho e nível da memória, este gasto pode ser de até 50% do total ou mais (MONTANARO et al., 1996)(INOUE; ISHIHARA; MURAKAMI, 1999). Na Figura 6.12 é mostrada a diferença quando o alinhamento de acesso à memória é e não é usado considerando diferentes proporções de energia (eixo horizontal) gasta pelos acessos à memória. Os dados foram retirados da média do consumo de energia das três versões com laços desenrolados da IMDCT. Como pode ser observado, quando os acessos à memória são responsáveis por 50% do consumo de energia total do sistema, utilizando o alinhamento de acessos à memória pode-se economizar aproximandamente 15% da energia total. Então, comprova-se que esta técnica traz uma boa economia de energia e não custa praticamente nada em termos de desempenho e *hardware*.

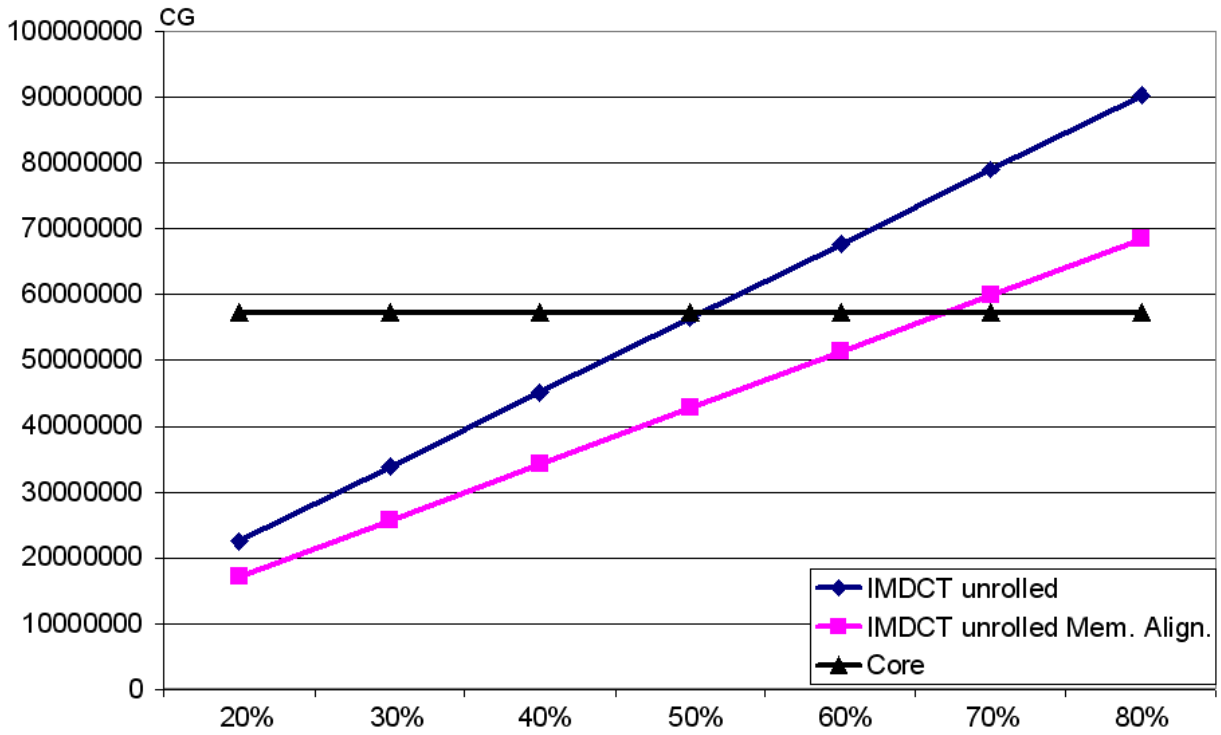

Figura 6.12 : Vantagens do alinhamento de memória no consumo total de energia no sistema

Em nenhum dos exemplos estudados foi achado um algoritmo que gaste menos energia na versão de 8 instruções por palavra. Entretanto, não foram usadas técnicas mais sofisticadas, como *software pipelining* (JONES; ALLAN, 1990) e *superblock* (LEE; TIRUMALAI; NGAI, 1993), para expor ainda mais o paralelismo das aplicações. Para programas que possuem muitos laços, *software pipelining* pode mostrar ganhos de performance de 50% até 300%, e *superblock* de 10% até 25% (LEE; TIRUMALAI; NGAI, 1993). Mesmo não sendo utilizadas estas técnicas, ganhos de até três vezes em termos de consumo de energia foram alcançados na versão VLIW 4 quando comparada com a versão *Low-Power* nas versões de laços desenrolados da IMDCT, demonstrando o potencial de um processador Java VLIW.

É claro que, como pôde ser observado, há um acréscimo de área considerável no sistema. Entretanto, este acréscimo parece ser bastante justificado pelos ganhos em energia e performance em alguns algoritmos, além deste ser comum em máquinas que exploram paralelismo nas aplicações.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho mostrou a exploração do espaço de projeto em sistemas embarcados baseados em Java. Através do simulador construído, o projetista se torna apto para explorar as diversas possibilidades arquiteturais do sistema em níveis mais altos de abstração. A expansão do processador Femtojava para a versão *Low-Power* e para as versões VLIW prova que é possível, através de diferentes técnicas, a exploração de características particulares de processadores de pilha para reduzir a potência e energia do sistema, uma das preocupações principais no projeto de embarcados atualmente.

O capítulo 2 mostrou a potencialidade de pesquisa em processadores que executam nativamente Java, e que a preocupação com a potência dissipada com o sistema mostra-se sempre secundária nestes. O capítulo 3 apresentou o simulador CACO-PS, que foi utilizado durante todo o restante do trabalho para a validação do sistema e cálculo de potência. Utilizando o simulador, a versão *Low-Power* do Femtojava foi desenvolvida e validada, e posteriormente programada e simulada em VHDL. O processador apresentou bons resultados relativos à potência e energia consumidas pelo sistema, verificado no capítulo 4.

Já resultados relativos ao folding de instruções para o conjunto empregado de *benchmarks* mostraram-se insatisfatórios. Assim, a implementação desta técnica foi abandonada, como demonstrado no capítulo 5. Todos os esforços foram concentrados no desenvolvimento de um analisador VLIW e do seu processador. Como a versão *Low-Power*, o processador VLIW também apresentou resultados satisfatórios na redução de energia geral do sistema para alguns algoritmos, mesmo sem a utilização de técnicas para a maior exposição do paralelismo nos programas.

Todavia, há vários pontos onde este trabalho pode ser expandido. Expansões no simulador, análises mais profundas nas versões *Low-Power* e VLIW, e ainda a exploração de outras alternativas arquiteturais vindas do mundo RISC são possíveis. A próxima seção explora com mais detalhes alguns destes possíveis trabalhos futuros.

7.1 Trabalhos Futuros

7.1.1 Expansões no CACO-PS

O simulador CACO-PS tem várias expansões e otimizações a serem feitas. Algumas delas são bastante importantes para facilitar a modelagem de componentes, como a implementação do suporte a múltiplos níveis na descrição de um sistema, já que atualmente a descrição de um sistema qualquer tem que ser feita em um formato plano. Por exemplo, se o processador Femtojava foi descrito na sintaxe do simulador, e é

desejado colocar quatro Femtojava juntos para estudar a comunicação entre eles, é preciso fazer a digitação de toda a descrição do processador quatro vezes, ao invés de usar instâncias desta descrição.

Outra expansão interessante é a implementação de uma interface gráfica para ajudar na descrição de sistemas e torná-la mais legível. O usuário seria capaz de, usando a interface gráfica, montar sistemas completos usando os componentes descritos na biblioteca. Esta interface também ajudaria na depuração, pois seria possível para o usuário enxergar os valores de todos os sinais de entrada e saída de cada componente a cada ciclo de relógio durante a execução do sistema de forma gráfica.

Além do mais, há algumas restrições na sintaxe que precisam ser resolvidas, como a não possibilidade de selecionar *bits* separados de um sinal qualquer para servir de entrada para um componente. Para fazer isso, atualmente, é necessário declarar um componente intermediário, que filtrará os *bits* desejados e os colocará em um sinal de saída, este servindo de entrada para o componente desejado inicialmente.

Além disso, em algum lugar ainda não definido na descrição da arquitetura haverá a possibilidade do usuário declarar o tamanho do sinal. Esta declaração tem dois propósitos: o uso de máscaras automáticas para o sinal desejado na declaração dos componentes; e para a geração de falhas no módulo de testes do simulador. Para o segundo item, geração de falhas, é necessário saber previamente o tamanho de cada sinal. Como o simulador injeta falhas em todos os *bits* dos sinais do sistema e detecta a propagação das falhas em sinais escolhidos pelos usuários, a inclusão de falhas em *bits* de alguns sinais que nunca são utilizados nunca será detectada, e conseqüentemente, haverá um erro de medição no resultado final de detecção de falhas.

Atualmente, a busca de componentes é feita de forma seqüencial, comparando o nome do componente que está sendo usado na arquitetura no momento com o nome de cada componente encontrado na biblioteca. Não há maneiras de usar técnicas de busca de componentes na biblioteca: isto implicaria no usuário ter que ordenar manualmente os componentes em ordem alfabética, por exemplo. Isso acontece porque os componentes são buscados através de comparação de *strings* dentro de uma função. O objetivo é fazer com que cada componente seja uma função. Como cada função na realidade é um *ponteiro*, quando lida a arquitetura, guarda-se agora não mais o nome do componente, mas sim o seu índice de um vetor de ponteiros das funções que representam cada componente. Assim, a busca será direta, acelerando a execução do programa. Já existe atualmente uma versão utilizando esta técnica sendo testada.

Em paralelo a isto, o CACO-PS será transformado em uma biblioteca para ser usado com a linguagem de descrição *SystemC* (SYSTEMC, 2004). Esta nova biblioteca permitirá que o usuário calcule a taxa de chaveamento de cada componente e, conseqüentemente, a potência total do sistema, nos moldes de como o simulador CACO-PS funciona atualmente. Não seria necessário fazer alteração alguma nas descrições de um sistema já existente ou ainda alterações mínimas, como alguma chamada de método ou declaração de sinais. A vantagem da migração do CACO-PS para *SystemC* é que esta linguagem de descrição é amplamente utilizada e conhecida, com várias arquiteturas e sistemas já descritos. Além do mais, toda a parte de sintaxe de descrição de sistema em que a linguagem do CACO-PS é carente, *SystemC* suporta. A única desvantagem do *SystemC*, em termos de declaração e construção de sistemas, é que é uma linguagem compilada: toda vez que uma arquitetura é descrita é necessária a recompilação, ao contrário do uso da interpretação na descrição da arquitetura no CACO-PS.

Também já está disponível e em fase de testes um programa que faz a transição da linguagem VHDL para a linguagem nativa do simulador. Este programa, todavia, funciona com um VHDL estrutural, pós-sintese, gerado pelo programa Leonardo Spectrum da Mentor (MENTOR, 2004).

7.1.2 Femtojava Low-Power

No processador Femtojava *Low-Power* é necessário fazer mais estudos em sua arquitetura para procurar os caminhos críticos do sistema com o intuito de aumentar a sua freqüência. Mesmo dando um resultado relativamente bom comparado ao multiciclo, como pôde ser observado no capítulo 4, há otimizações arquiteturais que ainda podem ser feitas. Um bom alvo de estudos é o multiplicador, que é o caminho crítico do quarto estágio, de execução. Uma solução é dividir o multiplicador em estágios: mesmo que uma instrução de multiplicação leve mais ciclos para ser executada, as outras serão beneficiadas pelo aumento de freqüência do processador.

Uma unidade de ponto flutuante já foi desenvolvida e está sendo testada. Como a unidade de ponto flutuante é bem maior e mais lenta que o multiplicador, estudos como dividi-la em estágios também precisam ser feitos. Nenhum *bit* adicional na palavra de controle precisa ser colocado, ao contrário do Femtojava Multiciclo, pois a implementação atual já prevê a inclusão de novas unidades funcionais no estágio de execução.

Finalmente, uma extensão do *Sashimi* pode ser desenvolvida, que servirá para a geração do *hardware* ASIP da versão Femtojava *Low-Power*, nos mesmos moldes de como é feito com a versão Multiciclo atualmente. Basicamente esta versão *Low-Power* pode ser parametrizável na sua parte de controle, decodificação e dependência de dados, assim como em suas unidades funcionais no estágio de execução. Por exemplo, se a instrução de multiplicação não é utilizada pelo programa Java final, o multiplicador não precisa estar presente no Femtojava *Low-Power* que irá executar este programa Java.

7.1.3 Femtojava VLIW

Além de pesquisas que ainda podem ser feitas no *hardware* deste processador, certamente a área que pode trazer melhores resultados é na análise do código à procura de paralelismo. A construção de grafos de dependência mais complexos, com a utilização de algoritmos mais sofisticados para a análise do paralelismo com o intuito de melhor distribuição de blocos de operandos a serem paralelizados, certamente é um bom ponto de estudo. Além do mais, a análise de paralelismo entre blocos básicos pode ser feita: quando é sabido que um bloco básico é executado logo a partir de outro, estes dois podem ser paralelizados. Outra alternativa ainda é usar os *slots* não utilizados na palavra VLIW para fazer execução especulativa: quando há um laço logo em frente, e não se sabe previamente que caminho será tomado, executa-se especulativamente ambos os caminhos e, depois de sabida a condição do laço, descarta-se aquela execução especulativa que foi errada e mantém-se o estado da certa.

Técnicas para expor mais o paralelismo, como *software pipelining*, onde a idéia básica é a reestruturação dos núcleos dos laços em busca de mais paralelismo (JONES; ALLAN; 1990), também devem ser estudadas. O paralelismo adicional é conseguido através da execução de iterações posteriores dos núcleos dos laços antes que as anteriores tenham acabado. Desta forma, uma série de dependências de dados pode ser resolvida. Diversos algoritmos para diferentes reestruturações a fim de deixar o menor número de dependências possíveis foram propostos (JONES; ALLAN; 1990). O uso da

técnica de *software pipelining* pode ser muito útil para processadores que exploram o paralelismo ao extremo, como processadores do tipo superescalar, VLIW e SMT.

Salienta-se que ainda não foi estudado o uso de *software pipelining* especificamente em arquiteturas de pilha. Em conseqüência disso, uma análise dos algoritmos existentes, que levam em consideração microoperações de máquinas RISC para efetuar a busca do paralelismo, devem ser analisados com o objetivo de verificar a possibilidade de serem aplicados em processadores de pilha.

Para esta versão, também há a possibilidade de expansão da ferramenta Sashimi. Este processador, entretanto, seria bem mais parametrizável que o *Low-Power*. Além das unidades funcionais de cada fluxo serem parametrizáveis, há também um outro eixo a ser explorado: quais unidades funcionais devem ser compartilhadas entre os diversos fluxos, analisando o custo-benefício entre o incremento de área e a performance perdida na paralelização quando as unidades são compartilhadas. Além de tudo isso, há também a parametrização no suporte de quantas instruções por palavra VLIW o processador pode suportar. Para determinadas aplicações, o suporte de duas instruções por palavra pode ser o suficiente e até o máximo a ser atingido. Então, o uso de palavras maiores seria um desperdício. Esta análise poderia ser feita de forma automática por uma ferramenta ou ainda pelo projetista usando simulação.

7.1.4 Outras técnicas a serem exploradas

Dando continuidade a exploração do espaço de projeto de *hardware*, várias técnicas que já foram implementadas em processadores do tipo RISC podem ser aplicadas em processadores Java. Algumas delas são:

- **SMT**

Proposto primeiramente em (YAMAMOTO et al., 1994) e (HIRATA, 1992), foi batizado com este nome em (TULLSEN; EGGERS; LEVY, 1996). Esta técnica resumidamente baseia-se em encontrar, em tempo de execução, paralelismo de instruções entre as *threads*, despachando instruções de diferentes *threads* para diferentes unidades funcionais no mesmo ciclo. Para isso, é necessário *hardware* adicional para o controle e para guardar o estado de cada *thread* (TULLSEN; EGGERS; LEVY, 1996). A grande desvantagem da utilização desta técnica é exatamente a área necessária para o controle e para salvar os estados (BURNS; GAUDIOT, 2002). SMT é uma evolução de processadores superescalares, que fazem análise do paralelismo de apenas uma *thread* para a alocação de instruções em unidades funcionais, e de processadores *multithreaded*, que fazem a análise do paralelismo de instruções de diferentes *threads*, mas em ciclos alternados. Segundo (EGGERS et al., 1997), SMT é a melhor técnica para combater o desperdício horizontal e vertical das unidades funcionais.

A implementação do SMT em processadores Java é bastante interessante, já que Java suporta intrinsecamente *threads* em sua natureza, tendo primitivas para sincronização entre *threads* na própria linguagem. Um grande problema de arquiteturas SMT é a área utilizada para o seu suporte, em especial a área ocupada pelos registradores de propósito geral (BURNS; GAUDIOT, 2002). Entretanto, o impacto em máquinas de pilha pode ser diferente, visto que o número de registradores nestas máquinas está diretamente ligado ao tamanho da pilha de operandos, ou em quanto a pilha pode crescer em cada *thread*.

- **Binary Translation**

Esta técnica se resume em analisar em tempo de execução o código binário da máquina para execução e substituí-lo por outro (ALTMAN; KAELI; SHEFFER, 2000). Esta técnica geralmente é utilizada para executar códigos que não são provenientes da máquina de destino: os códigos binários são convertidos em tempo de execução para o código nativo da máquina. Por exemplo, o processador Transmeta (TRANSMETA, 2003) faz *binary translation* do código X86. O código binário X86 é recebido e, em tempo de execução, é transformado na linguagem nativa da máquina..

O principal uso de *binary translation* já foi citado anteriormente. Todavia, esta técnica também pode ser usado para outros fins. Um deles é fazer uma máquina reconfigurável em relação ao seu paralelismo. O código é analisado em tempo de execução em busca de paralelismo. Conforme esta busca é efetuada, e o paralelismo no código é encontrado, uma nova instrução, do tipo VLIW, é gerada para ser executada. O impacto do *binary translation* em um processador de pilha, devidamente preocupado com o consumo de potência, é que, conforme o paralelismo encontrado em tempo de execução, algumas partes, como unidades funcionais, podem ser desligadas economizando energia. O principal contraponto do uso desta técnica é o *hardware* adicional necessário para fazer esta análise dinâmica. Além do mais, esta análise com *binary translation* também poderia transformar seqüências de instruções a serem executadas em um *array* reconfigurável em tempo de execução, aumentando ainda mais a performance do processador.

- **CMP**

Chip Multiprocessing (PALACHARLA; JOUPPU; SMITH, 1996) tem como principio ter múltiplos processadores dentro de uma mesma pastilha de silício, que podem suportar o mesmo conjunto de instruções, mesmo que possuam arquiteturas diferentes. Estas trabalham em paralelo (por exemplo, cada um executando uma *thread* de um processo) se comunicando em alta velocidade, já que estão dentro da mesma pastilha. Cada processador tem sua própria memória *cache*, TLB, etc. O grande problema do uso desta técnica é que o paralelismo fica restrito ao ILP de cada *thread*.

Expandindo e mostrando uma das utilidades desta última idéia, em (KUMAR, 2003) é usado um conjunto de processadores com o mesmo ISA, cada um com diferentes níveis de performance, tamanho e área. Conforme a necessidade da aplicação que está sendo executada no momento, um dos processadores é escolhido. Os outros processadores permanecem desligados. A vantagem desta técnica é que, no momento em que a aplicação não exige o máximo poder computacional, um processador simples pode executá-la, poupando grande quantidade de energia que seria gasta caso um processador mais poderoso estivesse a executando.

Estudos iniciais em (MATTOS; BECK; CARRO, 2004b) mostram o enorme espaço de projeto existente, considerando área, performance e dissipação de potência, quando vários algoritmos que desempenham a mesma função, entretanto implementados de forma diferente, são executados em diferentes processadores da mesma ISA.

O uso de *Chip Multiprocessing* em Java pode ser feito da seguinte forma: cada processador dentro do chip executa uma *thread* Java. Dependendo da aplicação, *threads* que precisam de um maior poder computacional podem ser distribuídas para processadores Java mais robustos, enquanto as outras, menos exigentes, podem ser distribuídas para processadores mais simples. Além do mais, diferentes processadores

podem executar diferentes *threads* levando em consideração diferentes requisitos, como execução de tempo real e potência.

A idéia de (KUMAR, 2003), sobre o uso de diferentes implementações da mesma ISA, pode ser expandida para a técnica citada anteriormente. Quando algumas *threads* pararam de ser executadas ou estão bloqueadas, alguns processadores podem ser desligados, deixando apenas os processadores com *threads* ativas ligados.

7.1.5 Geração totalmente automática de sistemas embarcados baseados em Java

Tendo todas estas alternativas arquiteturais parametrizáveis, uma nova ferramenta pode ser planejada para trabalhar em conjunto com o CACO-PS, para, a partir de um conjunto de algoritmos a serem mapeados para *hardware*, que foram obtidos da melhor maneira a partir de ferramentas anteriores em desenvolvimento, escolher a melhor alternativa para a geração do sistema final, em *hardware*, conforme os três eixos principais: área, performance e potência.

Com esta nova ferramenta, junto com as ferramentas SASHIMI, CACO-PS e as outras que estão sendo desenvolvidas pelo grupo, poder-se-á obter um conjunto eficiente para a geração de sistemas embarcados. Este tipo de resultado trará grande facilidade aos desenvolvedores, já que, em um curto espaço de tempo, estes poderão ao mesmo tempo usufruir da difundida linguagem Java e ter a preocupação explícita com a potência consumida, área e performance, dependo da aplicação desejada.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores**: Princípios, Técnicas e Ferramentes. Rio de Janeiro: LTC, 1995.

ALTERA Homepage. Disponível em: <http://www.altera.com>. Acesso em: 27 maio 2004.

ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Welcome to the Opportunities of Binary Translation. **Computer**, Los Alamitos, CA, v.33, n.3, p. 40-45, Mar. 2000.

ATHERTON, R.W. Moving Java to the Factory. **IEEE Spectrum**, New York, v. 35, n. 12, p. 18-23, Dec. 1998.

ALVERSON, R. W. et al. The Tera Computer System. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 4., 1990, Amsterdam. **Proceedings...** New York: ACM Press, 1990. p. 1 – 6.

AJILE SYSTEMS INC. **aJ-100 Reference Manual**. Disponível em: <http://www.ajile.com/downloads/aJ-100ReferenceManual.pdf>. Acesso em: 27 maio 2004.

ARM Jazelle Technology. Disponível em <http://www.arm.com/products/solutions/Jazelle.html>. Acesso em: 27 maio 2004.

BECK FILHO, A. C. S.; MATTOS, J. C.; WAGNER, F. R.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 349 – 354.

BECK FILHO, A. C. S.; ROSA JUNIOR, L. S.; WAGNER, F. R.; CARRO, L. A General Purpose Compiled-Code Power Simulator. In: SOUTH SYMPOSIUM ON MICROELETRONICS**,** SIM, 18., 2003. **Proceedings…** Novo Hamburgo: Feevale, 2003. p. 133 – 136.

BECK FILHO, A. C. S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 12., 2003, Darmstadt. **Proceedings…** Darmstadt: Technische Universitat Darmstadt, 2003. p. 239 – 244.

BECK FILHO, A. C. S.; CARRO, L. Um Processador Java VLIW com Baixo Consumo de Potência para Sistemas Embarcados. In: WORKSHOP IBERCHIP, 10., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 114.

BECK FILHO, A. C. S.; CARRO, L. A VLIW Low Power Java Processor for Embedded Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 17., 2004, Porto de Galinhas. **Proceedings…** New York: ACM Press, 2004. p. 157 – 162.

BRINKSCHULTE, U. et al. The Komodo Project: Thread-Based Event Handling Supported by a Multithreaded Java Microcontroller. In EUROMICRO CONFERENCE, 25., 1999, Milano. **Proceedings…** Los Alamitos: IEEE Computer Society, 1999. v. 2, p. 2122-2127

BRINKSCHULTE, U. et al. A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling. In PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 1999. Newport Beach. **Proceedings…** Washington: IEEE Computer Society, 1999, p. 34-39.

BURNS, J.; GAUDIOT, J.-L. SMT Layout Overhead and Scalability. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.13, n.2, p. 142-155, Feb. 2002.

CHEN, R.; IRWIN, M. J.; BAJWA, R. Architecture-Level Power Estimation and Design Experiments. **ACM Transactions on Design Automation of Electronic Systems**, New York, v.6, n.1, p. 50-66, 2001.

CHEN, G. et al. Tuning garbage collection for reducing memory system energy in an embedded java environment. **ACM Transactions on Embedded Computing Systems**, New York, v.1, n.1, p. 27-55, Nov. 2002.

CHOI, K.; CHATTERJEE, A. Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems. In INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS**,** 14., 2001, Montréal. **Proceedings…** New York: ACM Press, 2001. p. 147 – 152.

CRAMER, T. et al. Compiling Java Just in Time. **IEEE Micro**, Los Alamitos, v.17, n.2, p. 36-43, 1997.

DALAL, V.; RAVIKUMAR, C. P. Software Power Optimizations in an Embedded System. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 14., 2001, Bangalore. **Proceedings…** Washington: IEEE Computer Society, 2001. p. 254-259.

EDWARDS, S. et al. Design of Embedded Systems: Formal Models, Validation, and Synthesis. **Proceedings of the IEEE**, New York, v.85, n.3, p. 366-390, Mar. 1997.

EGGERS, S. et al. Simultaneous Multithreading: A Platform for Next-generation Processors. **IEEE Micro**, Los Alamitos, v. 17, n. 5, p. 12-18, 1997.

GERVINI, A. I.; CORREA, E. F.; CARRO, L.; WAGNER, F. R. Avaliação de Desempenho, Área e Potência de Mecanismos de Comunicação em Sistemas Embarcados. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, SEMISH, 30., 2003, Campinas. **Trabalhos Apresentados**. Campinas: SBC, 2003. p. 211 – 223.

GIVARGIS, T.; VAHID, F.; HENKEL, J. System-Level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, ICCAD, 2001, San Jose. **Proceedings…** Piscataway: IEEE Press, 2001. p. 25 – 30.

THE GNU Compiler for the Java Programming Language. Disponível em: <http://gcc.gnu.org/java/>. Acesso em: 27 maio 2004.

GOMES, V. F.; BECK, A. C. S.; CARRO, L. A VHDL Implementation of a Low Power Pipelined Java Processor for Embedded Applications In: WORKSHOP IBERCHIP, 10., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 102.

GRAY, J. et al. VIPER: A 25MHz, 100 MIPS Peak VLIW Microprocessor. In: IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1993, San Diego. **Proceedings…** Piscataway: IEEE Press, 1993. p. 4.1.1 – 4.1.5.

HANGAL, S.; O'CONNOR, J. M. Performance Analysis and Validation of the picoJava Processor. **IEEE Micro**, Los Alamitos, v.9, n.3, p. 66-72, 1999.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: A Quantitative Approach. 3rd ed. Amsterdam: Morgan Kaufmann, 2003.

HENTSCHKE, R.; BECK, A. C. S.; MATTOS, J. C. B.; CARRO, L.; LUBASZEWSKI, M.; REIS, R. Using Genetic Algorithms to Accelerate Automatic Software Generation for Microprocessor Functional Testing. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 5., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 98.

HIRATA, H. et al. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992, Queensland. **Proceedings…** New York: ACM Press, 1992. p. 136-145.

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

INOUE, K.; ISHIHARA, T.; MURAKAMI, K. Way-predicting set-associative cache for high performance and low energy consumption. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 19., 1999, San Diego. **Proceedings...** New York: ACM Press, 1999. p. 273 – 275.

JACOME, M.; VECIANA, G. Design Challenges for New Application-Specific Processors. **IEEE Design & Test**, California, v.17, n.2, p. 40-50, Apr./June 2000.

JONES, R. B.; ALLAN, V. H. Software Pipelining: a Comparison and Improvement. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 24., 1990, Albuquerque. **Proceedings…** New York: ACM Press, 1990. p. 46–56.

KAZI, I. H. et al. Techniques for obtaining high performance in Java programs. **ACM Comput. Surveys**, New York, v.32, p. 213–240, 2000.

KIM, A.; CHANG, M. Advanced POC model-based Java instruction folding mechanism. In: EUROMICRO CONFERENCE, 26., 2000, Maastricht. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000. v.1, p. 332-338.

KIM, N. S. et al. Leakage Current: Moore's Law Meets Static Power. **Computer**, Los Alamitos, v.36, n.12, p. 68-75, Dec. 2003.

KOOPMAN JUNIOR, P. K. **Stack Computers, the new wave**. Chichester: Ellis Horwood, 1989.

KOWALCZYK A. First-Generation MAJC dual microprocessor. In: ISSCC, 2001. **Dig. Tech. Papers**. New York: IEEE, 2001. p. 236–237.

KOWALCZYK, A. et al. The first MAJC microprocessor: A dual CPU Systems-on-chip, **IEEE JSSC**, New York, v.36, p. 1609-1616, Nov. 2001.

KRALL, A. Efficient JavaVM Just-in-Time Compilation. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, 1998, Paris. **Proceedings…** Washington: IEEE Computer Society, 1998. p. 205 – 212.

KUMAR, R. et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 36., 2003. **Proceedings…** New York: ACM Press, 2003. p. 81 – 92.

LANDMAN, P.; RABAEY, J. Activity-Sensistive Architectural Power Analysis. **IEEE Transactions on CAD of Integrated Circuits**, New York, v.15, n.6, p. 571-587, 1996.

LAWTON, G. Moving Java into Mobile Phones. **Computer**, v.35, n.6, p. 17-20, 2002.

LEE, M.; TIRUMALAI, P.; NGAI, T. Software Pipelining and Superblock Scheduling: Compilation Techniques for VLIW Machines. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 26., 1993, Hawaii. **Proceedings…** Los Alamitos: IEEE Computer Society, 1993. p. 202 – 213.

LEE, E. What's Ahead for Embedded Software? **Computer**, New York, v. 33, n. 9, p.18-26, Sept. 2000.

LEONARDO Spectrum Homepage. Disponível em: <http://www.mentor.com/sysnthesis> . Acesso em: 3 abr. 2004.

LIU, D.; SVENSSON, C. Power Consumption Estimation in CMOS VLSI Chips. **IEEE Journal of Solid-State Circuits**, New York, v.29, n.6, p. 663-670, 1994.

LYONNARD, D. et al. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System on Chip. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings…** New York: ACM Press, 2001. p. 518 – 523.

MARCON, C. A. M. **Hardware Description in C**: hdc : v 2.0. Porto Alegre: CPGCC da UFRGS, 1992. 26 f. (RP-203).

MATTOS, J. C. B.; BRISOLARA, L.; HENTSCHKE, R.; CARRO, L.; WAGNER, F. R. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2004, Toulouse. **Proceedings…** Boston: Kluwer Academic Publishers, 2004. p.237 – 246.

MATTOS, J. C. B.; BECK, A. C. S.; CARRO, L. Design Space Exploration with Automatic Selection of SW and HW for Embedded Applications. In: INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING AND SIMULATION, SAMOS, 4., 2004, Samos. **Proceedings…** Berlim: Springer, 2004. p. 303 – 312.

MCGHAN, H.; O'CONNOR, M. PicoJava: A Direct Execution Engine for Java Bytecode. **Computer**, Los Alamitos, v. 31, n.10, p. 22 – 30, 1998.

MONTANARO, J. et. al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. **IEEE Journal of Solid-State Circuits**, New York, v.31, n.11, p. 1703-1714, Nov. 1996.

NAKAMURA, K.; SAKAI, K.; AE, T. Real-Time Multimedia Data Processing using VLIW Hardware Stack Processor. In: IEEE WORKSHOP ON PARALLEL AND DISTRIBUTED REAL-TIME SYSTEMS, WPDRTS, 1995, Geneva. **Proceedings…** Washington: IEEE Computer Society, 1995. p. 84 – 89.

NANDI, A.; MARCULESCU, R. System-Level Power/Performance Analysis for Embedded Systems Design. In: DESIGN AUTOMATION CONFERENCE, DAC, 38. 2001, Las Vegas. **Proceedings…** New York: ACM, 2001. p 599 – 604.

NAZOMI COMMUNICATIONS. Nazomi Offers JVM Porting Options Program (JPOP) To Expand JSTAR99 Coprocessors For Java99 Technology and Java Card99 Applications. Disponível em: <http://www.nazomi.com/popUpPR.asp?art=12>. Acesso em: 19 maio 2002.

NOKIA N-Gage Home Page. Disponível em: <http://www.n-gage.com>. Acesso em: 14 maio 2002.

O'CONNOR, J. M.; TREMBLAY, M. picoJava-I: The Java virtual machine in hardware. **IEEE Micro**, Los Alamitos, v.17, n.2, p. 45-53, 1997 ,

PALACHARLA, S.; JOUPPI, N.; SMITH, J. E. **Quantifying the complexity of superscalar processors.** [S.l.]: University of Wisconsin-Madison, CS Department, 1996. (Technical Report 1328).

PETROV, P.; ORAILOGLU, A. Speeding Up Control-Dominated Applications through Microarchitectural Customizations in Embedded Processors. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings…** New York: ACM Press, 2001. p. 512 – 517.

PIGUET, C.; RENAUDIUM, M.; OMMÈS, T. Low Power Systems on Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2001, Munich. **Proceedings…** Piscataway: IEEE Press, 2001. p. 488.

POUWELSE, J.; LANGENDOWN, K.; SIPS, H. Dynamic Voltage Scaling on a Low-Power Microprocessor. In: INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, 7., 2001, Italy. **Proceedings…** New York: ACM Press, 2001. p. 251 – 259.

RABAEY, J. M. **Digital Integrated Circuits:** A Design Perspective. Upper Saddle River: Prentice Hall, 1996

RABAEY, J. M. Silicon Platforms for the Next Generation Wireless Systems – What Role does Reconfigurable Hardware Play? In: FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 10., 2000, Austria. **Proceedings…** Berlim: Springer-Verlag, 2000. p. 277 – 285.

RATIONAL SOFTWARE CORPORATION. **Unified Modeling Language. Notation Guide**. **Version 1.0**. Santa Clara, 1997. Disponível em: <http://www.rational.com>. Acesso em: 21 jan. 2002.

REYNERI, L. M.; CUCINOTTA, F.; SERRA, A.; LAVAGNO, L. A Hardware/Software Co-design Flow and IP Library Based on Simulink. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings…** New York: ACM Press, 2001. p. 593 – 598.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Plataform-Based Design and Software Design Metodology for Embedded Systems. **IEEE Design & Test**, California, v.18, n.6, p. 23 – 33, Nov./Dec. 2001.

SCHLETT, M. Trends in Embedded-Microprocessor Design. **Computer**, Los Alamitos, v.31, n. 8, p. 44–49, 1998.

SESHAN, N. High VelociTI Processing, **IEEE Signal Processing Magazine**, New York, v. 15, n.2, p. 86-101, Mar. 1998.

SHANDLE, J.; MARTIN, G. **Making Embedded Software reusable for SoCs**. Disponível em: <http://www.eedesign.com/story/OEG20020301S0104>. Acesso em: 26 fev. 2002.

SEMICONDUCTOR Industry Association Home Page. Disponível em: <http://public.itrs.net/Files/2003ITRS/Home2003.htm>. Acesso em: 23 mar. 2002.

SIMUNIÉ, T. ; BENINI, L. ; DE MICHELI, G. Cycle-Accurate Simulation of Energy Consumption in Embedded Systems. In: DESIGN AUTOMATION CONFERENCE**,**

DAC, 36., 1999, New Orleans. **Proceedings…** New York: ACM Press, 1999. p. 867 – 872.

SIMUNIC, T.; MICHELI, G.; BENINI, L. Energy-Efficient Design of Battery-Powered Embedded Systems. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED**,** 1999, San Diego. **Proceedings…** New York: ACM Press, 1999. p. 212 – 217.

STITT, G. et al. A First-Step Towards an Architecture Tuning Methodology. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, CASES, 2000, San Jose. **Proceedings…** New York: ACM Press, 2000. p. 187 – 192.

SUDHARSANAN, S. MAJC-5200: A High Performance Microprocessor for Multimedia Computing. In: WORKSHOPS ON PARALLEL AND DISTRIBUTED PROCESSING, IPDPS, 15., 2000. **Proceedings...** London: Springer-Verlag, 2000. p. 163 – 170.

SUDHARSANAN, S. et al. A. Image and Video Processing Using MAJC 5200. In: INTERNATIONAL CONFERENCE ON IMAGE PROCESSING, ICIP, 2000, Vancouver. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000. p. 122 – 125.

SUGA, A.; MATSUNAMI, K. Introducing the FR500 embedded microprocessor. **IEEE Micro**, Los Alamitos, p. 21 – 27, July/Aug. 2000.

SUN MICROSYSTEMS INC. **picoJava-II Programmer's Reference Manual**. [S.l.], 1999.

SUN MICROSYSTEMS INC. **picoJava-II Microarchitecture Guide**. [S.l.], 1999.

SYSTEMC Homepage. Disponível em: <http://www.systemc.org>. Acesso em: 21 mar. 2002.

TAKAHASHI, D. **Java Chips Make a Comeback**. [S.l.]: Red Herring, 2001.

TANG, W.; GUPTA, R.; NICOLAU, A. Power Saving in Embedded Processors through Decode Filter Cache. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2002, Paris. **Proceedings…** Washington: IEEE Computer Society, 2002. p. 443 – 448.

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) Systems**, New York, v.2, n.4, p. 437-445, 1994.

TON, L. et al. Instruction Folding in Java Processor. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, ICPADS, 1997. **Proceedings…** [S.l.: s.n.], 1997. p. 138-143.

TRANSMETA CORPORATION. **Tm5400 processor specifications.** Disponível em: <http://www.transmeta.com>. Acesso em: 3 fev. 2002.

TREMBLAY, M. et al. The MAJC architecture: A synthesis of parallelism and scalability. **IEEE Micro**, Los Alamitos, v.20, n. 6, p. 12–25, Nov./Dec. 2000.

TULLSEN, D.; EGGERS, S.; LEVY, H. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 22., 1996, S. Margherita Ligure. **Proceedings…** New York: ACM Press, 1996. p. 392 – 403.

VOLDER, J. The CORDIC trigonometric computing technique. **IRE Transactions Electronic Computers,** [S.l.], v.EC-8, n.3, p. 330-334, Sept. 1959.

WAGNER, F.; OYAMADA, M.; CARRO, L.; KREUTZ, M. Object-Oriented Modeling and Co-Simulation of Embedded Systems. In: IFIP TC10/WG10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 10., 1999, Lisboa. **VLSI**: Systems on a chip. Boston: Kluwer, 2000. p. 497 – 508.

WILKINSON, T. **Kaffe**: A JIT and interpreting virtual machine to run Java code. 1998. Disponível em: <http://www.transvirtual.com>. Acesso em: 10 maio 2002.

WILTON, S.; JOUPPI, N. CACTI: An Enhanced Cache Access and Cycle Time Model. **IEEE Journal of Solid-State Circuits**, New York, v.31, n. 5, p. 677-688, 1996.

YAMAMOTO, W. et al. Performance Estimation of Multistreamed, Superscalar Processors. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 27., 1994, Hawaii. **Proceedings…** Los Alamitos: IEEE Computer Society, 1994. p. 195 – 204.

ANEXO A DESCRIÇÃO DO CÁLCULO DE POTÊNCIA NO CACO-PS

As funções de cálculo de potência são encontradas no arquivo *cacops_potencia.c*. Para as funções de cálculo de potência, as seguintes funções são oferecidas pelo simulador:

- transicao_input(x , y) indica se houve uma transição (ou chaveamento do valor do ciclo anterior em relação ao ciclo atual) do bit y do sinal de entrada x.

Usando um multiplexador de 4 entradas como exemplo:

entrada1, entrada2, entrada3, entrada4-> mux4_1 exemplo5 (4,5:controle) -> sinal_saida

Se dentro da função de cálculo de potência, o seguinte comando for dado:

transicao_input (2,4)

A função retornará o valor 1 caso o bit número 4 (que é o quinto bit, visto que a contagem começa do zero) do sinal entrada3 teve seu valor chaveado de 0 do ciclo anterior, para 1 do ciclo corrente, ou ainda de 1 para 0. Se não houve chaveamento o valor de retorno será zero.

- transicao_controle (x) indica se houve uma transição no sinal de controle X.

Usando como exemplo o mesmo multiplexador anterior, dentro da função de cálculo de potência, o projetista chama a função:

transicao_controle(1)

Esta retornará o valor 1 caso o segundo bit de controle declarado na descrição tiver o seu valor chaveado do ciclo anterior para o corrente. No exemplo, este bit seria o bit 5 do sinal controle.

- transicao (x , y , z) recebe como parâmetros dois valores inteiros x e y e indica (retornando o valor 1) se entre eles o bit de número Z é diferente ou igual. Pode ser usada para auxiliar taxas de chaveamento internas do componente na função de cálculo de potência.

Como exemplo, na Figura 0.1, é mostrado como é feito o cálculo de potência dinâmica de um componente registrador. No arquivo *cacops_potencia.c*, encontra-se declarado um COMPONENTE_POTENCIA, de mesmo nome do COMPONENTE declarado na biblioteca de componentes. Neste exemplo, o nome do componente previamente declarado é um registrador, então a sua função de cálculo de potência é declarada da seguinte forma:

```
1.  CALCULO_POTENCIA(“reg”) {
2.      int result=0, i;
3.      for (i=0;i<16;i++)
4.          result += cg[1];
5.      if ( transicao_controle(0) ) result += cg[2];
6.      if (controle_anterior[0] == 1) {
7.          for (i=0;i<16;i++)
8.              if (transicao_input(0,i)) result += cg[0];
9.      }
10.     return result;
11. }
```

Figura 0.1 : Exemplo de uma função de cálculo de potência no CACO-PS

Primeiramente, na linha 2, é declarada uma variável temporária que auxiliará no cálculo da potência.

Nas linhas 3 e 4 há um laço, que é repetido 16 vezes, e coloca o valor do vetor cg[1] na variável result. Note que esta variável será retornada (linha 10) com o valor de consumo total de potência daquele componente. O laço é repetido 16 vezes porque o componente reg é um registrador de 16 bits. Então, em cg[1] encontra-se um valor fixo de potência que sempre será gasto por cada bit do registrador a cada ciclo, independente se alguma entrada variou ou não.

A linha 5 verifica se houve uma transição no primeiro bit de controle do registrador. Caso houve esta transição, o valor de cg[2] é somado ao resultado final (variável result).

O if na linha 6 verifica se o bit de controle 0 (correspondente ao *enable* do registrador) tinha seu valor igual a 1 quando passou de um ciclo de relógio para outro. Se isto ocorreu, significa que os dados do registrador serão atualizados com os dados de entrada. Então, para cada bit do registrador, é verificado se houve uma transição ou não (linhas 7 e 8). Para cada bit do input[0] (primeira e única entrada do registrador) que mudou de valor, o valor de cg[0] é adicionado ao resultado final.

Finalmente, o resultado final (a variável result) é retornada. E este valor retornado é que será contado ciclo por ciclo para o cálculo total de potência. Todavia, uma coisa ainda não foi esclarecida: Onde se encontram os valores do vetor cg[], que é usado para incrementar o resultado; e por que não usar valores diretamente ao invés de usar o vetor cg[] ?

Começando pela resposta da segunda pergunta: é feito para facilitar o trabalho do projetista. Considere que a tecnologia de fabricação do registrador mudou. Os valores de potência mudaram, todavia a função de cálculo de potência continua a mesma. Então, o projetista precisa apenas mudar os valores do vetor cg[] e não editar diretamente a função previamente programada. Os valores de cg[] são declarados, para cada função de cada componente de potência, no inicio do arquivo *cacops_potencia.c*. Para cada função de cálculo de potência, existem as seguintes linhas:

```
1.  // START reg
2.  //
3.  // CG 0 = 4
4.  // CG 1 = 5
5.  // CG 2 = 16
6.  //
7.  // END
```

Cada função de cálculo de potência tem que ter agregada um conjunto de linhas como essas:

- START nome, onde nome é o mesmo nome do componente e da função de cálculo de potência. Neste exemplo, reg.
- CG indice = valor, onde é colocado o valor para determinado índice do vetor CG[] que será usado para a função de cálculo de potência. No exemplo, CG[0] tem o valor 4. Então (observando a função de cálculo de potência do registrador), para cada bit da entrada do registrador que foi chaveado, com o *enable* ligado, o valor de 4 (neste caso, número de capacitâncias de porta) será somado ao resultado.
- END, que indica que a declaração de valores do vetor CG[] para aquele componente chegou ao fim. A partir daí, pode-se colocar um START com outro componente.

É importante salientar duas coisas: estas linhas são comentadas na sintaxe da linguagem C e interpretadas pelo simulador. Assim, apenas é necessária a recompilação do arquivo quando as funções do cálculo de potência foram mudadas, não os valores dos vetores CG[].

Como pôde ser observado, a metodologia usada para a simulação foi usar o número de chaveamento de portas (CG) para cada componente, pois através deste número pode-se concluir a potência dinâmica consumida pelo circuito. Nos exemplos que foram demostrados nesto trabalho, os dados de CG para componente foram retirados a partir da descrição em nível das portas lógicas de cada componente.

O usuário, quando programa a função de cálculo de potência para a ROM e RAM, pode, antes de retornar o resultado, somá-lo às variáveis globais total_ROM e total_RAM. Fazendo isso, quando a potência calculada é mostrada para o usuário, os valores estarão distinguidos, para a memória RAM, ROM e o resto do sistema (note que o resto do sistema é o total da potência consumida menos os valores da RAM e ROM).

ANEXO B EXEMPLO DE UM ARQUIVO NO FORMATO .MIF (BUBBLESORT)

```
-- MAX+plus II - Memory Initialization File - generated by SASHIMI CAD Tool

WIDTH = 8;
DEPTH = 256;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
    0   :   b8; -- invokestatic
    1   :   00; --
    2   :   2b; --
    3   :   00; -- SASHIMI.int0Method.()V.0
    4   :   00; --
    5   :   b1; -- return
    6   :   00; -- nop
    7   :   00; -- nop
    8   :   00; -- nop
    9   :   00; -- nop
    a   :   00; -- nop
    b   :   00; -- SASHIMI.tf0Method.()V.0
    c   :   00; --
    d   :   b1; -- return
    e   :   00; -- nop
    f   :   00; -- nop
    10   :   00; -- nop
    11   :   00; -- nop
    12   :   00; -- nop
    13   :   00; -- SASHIMI.int1Method.()V.0
    14   :   00; --
    15   :   b1; -- return
    16   :   00; -- nop
    17   :   00; -- nop
    18   :   00; -- nop
    19   :   00; -- nop
    1a   :   00; -- nop
    1b   :   00; -- SASHIMI.tf1Method.()V.0
    1c   :   00; --
    1d   :   b1; -- return
    1e   :   00; -- nop
    1f   :   00; -- nop
    20   :   00; -- nop
    21   :   00; -- nop
    22   :   00; -- nop
    23   :   00; -- SASHIMI.spiMethod.()V.0
    24   :   00; --
    25   :   b1; -- return
    26   :   00; -- nop
    27   :   00; -- nop
    28   :   00; -- nop
    29   :   00; -- nop
    2a   :   00; -- nop
    2b   :   00; -- Sort.initSystem.()V.0
    2c   :   00; --
    2d   :   b2; -- getstatic
    2e   :   00; --
    2f   :   10; --
```

```
30   :   b8; -- invokestatic
31   :   00; --
32   :   3a; --
33   :   b2; -- getstatic
34   :   00; --
35   :   11; --
36   :   10; -- bipush
37   :   08; --
38   :   f2; -- store_idx
39   :   b1; -- return
3a   :   03; -- Sort.bubbleSort.(I)V.4
3b   :   01; --
3c   :   03; -- iconst_0
3d   :   3c; -- istore_1
3e   :   1b; -- iload_1
3f   :   1a; -- iload_0
40   :   04; -- iconst_1
41   :   64; -- isub
42   :   a2; -- if_icmpge
43   :   00; --
44   :   3f; --
45   :   1a; -- iload_0
46   :   04; -- iconst_1
47   :   64; -- isub
48   :   3d; -- istore_2
49   :   1c; -- iload_2
4a   :   1b; -- iload_1
4b   :   a4; -- if_icmple
4c   :   00; --
4d   :   30; --
4e   :   b2; -- getstatic
4f   :   00; --
50   :   12; --
51   :   1c; -- iload_2
52   :   2e; -- iaload
53   :   b2; -- getstatic
54   :   00; --
55   :   12; --
56   :   1c; -- iload_2
57   :   04; -- iconst_1
58   :   64; -- isub
59   :   2e; -- iaload
5a   :   a2; -- if_icmpge
5b   :   00; --
5c   :   1b; --
5d   :   b2; -- getstatic
5e   :   00; --
5f   :   12; --
60   :   1c; -- iload_2
61   :   2e; -- iaload
62   :   3e; -- istore_3
63   :   b2; -- getstatic
64   :   00; --
65   :   12; --
66   :   1c; -- iload_2
67   :   b2; -- getstatic
68   :   00; --
69   :   12; --
6a   :   1c; -- iload_2
6b   :   04; -- iconst_1
6c   :   64; -- isub
6d   :   2e; -- iaload
6e   :   4f; -- iastore
6f   :   b2; -- getstatic
70   :   00; --
71   :   12; --
72   :   1c; -- iload_2
73   :   04; -- iconst_1
74   :   64; -- isub
75   :   1d; -- iload_3
76   :   4f; -- iastore
77   :   84; -- iinc
78   :   02; --
79   :   ff; --
7a   :   a7; -- goto
```

```
7b   :   ff; --
7c   :   cd; --
7d   :   84; -- iinc
7e   :   01; --
7f   :   01; --
80   :   a7; -- goto
81   :   ff; --
82   :   bc; --
83   :   b1; -- return
84   :   00; -- no code
85   :   00; -- no code
```

ANEXO C DESCRIÇÃO DO FEMTOJAVA MULTICICLO EM CACO-PS

```
////////////////////////////////////////////////////////////////////////
// SYS - Engloba todo o Femtojava
////////////////////////////////////////////////////////////////////////

// CORE - Engloba DATAPATH e CONTROL
////////////////////////////////////////////////////////////////////////

// CONTROL - Parte de controle/Decodificador de instrucoes
////////////////////////////////////////////////////////////////////////

// Portas
// instr
// int - Pino de interrupcao

// Saidas
// ctrl_word

instr -> reg IR (30,'1','1':ctrl_word) -> sig_opcode

sig_opcode -> fsm Control () -> ctrl_word

// FIM do CONTROL

// DATAPATH
////////////////////////////////////////////////////////////////////////

// Entradas
// clock - implicito
// reset -> stuck-at 1
// data_in = data_in
// ctrl_word = sig_ctrl_word
// immediate (8 bits) = IMM_in

///// PC_INIT_MUX

// Entradas
// 1a
sig_pc_alu_out -> reg PC (0,'1','1':ctrl_word) -> sig_pc_out

// 2a
nada -> gen_0 1 () -> no_value_0
no_value_0 -> gen_cons pc_init_generator (2,1,0:no_value_0) -> sig_pc_zero_value

// Saida
sig_pc_out, sig_pc_zero_value -> mux2_1 pc_init_mux (13:ctrl_word) ->
sig_pc_init_mux_out

////////////////////////////////////////////////////////////////////////
///// PC_INCR_MUX

// Entradas
// 1a
nada -> gen_1 1 () -> no_value_1
no_value_1 -> gen_cons pc_incr_generator (2,1,0:no_value_1) -> sig_pc_incr

// 2a - Saida do registrador IMM
```

```
// 3a
nada -> gen_m2 1 () -> sig_pc_m2_value

// 4a - Saida do registrador A

// Saida
sig_pc_incr,sig_immediate_out,sig_pc_m2_value,sig_a_out -> mux4_1 pc_incr_mux
(14:ctrl_word;0:sig_flag_mux_out) -> sig_pc_incr_mux_out

////////////////////////////////////////////////////////////////////////
///// Somador da saida dos dois mux anteriores - A saida entra no registrador PC

sig_pc_init_mux_out,sig_pc_incr_mux_out -> adder pc_alu () -> sig_pc_alu_out


///// ICONST_MUX
// Entradas
// 1a
nada -> gen_cons const_generator (17,16,15:ctrl_word) -> iconst_x
// 2a - Saida do registrador IMM

// Saida
nada -> iconsctrl 1 (17,16,15:ctrl_word) -> control_iconst
iconst_x, sig_immediate_out -> mux2_1 iconst_mux (0:control_iconst) -> sig_iconst_out

////////////////////////////////////////////////////////////////////////
///// ADDR_MUX

// Entradas
// 1a - Saida do registrador FRM
sig_addr_alu_out -> reg FRM (4:ctrl_word;0,'1':set) -> sig_frm_out
// 2a - Saida do registrador SP
sig_addr_alu_out -> reg SP (1:ctrl_word;0,'1':set) -> sig_sp_out
// 3a - Saida do registrador VARS
sig_addr_alu_out -> reg VARS (3:ctrl_word;0,'1':set) -> sig_vars_out
// 4a - Saida do registrador A

// Saida
sig_frm_out,sig_sp_out,sig_vars_out,sig_a_out -> mux4_1 addr_mux (12,11:ctrl_word) ->
sig_addr_mux_out

////////////////////////////////////////////////////////////////////////
///// Somador da saida dos dois mux anteriores - A saida entra em varios registradores
sig_addr_mux_out, sig_iconst_out -> addsub addr_alu (21:ctrl_word) -> sig_addr_alu_out

////////////////////////////////////////////////////////////////////////
///// Registrador MAR e IMM
sig_addr_alu_out -> reg MAR (2:ctrl_word;0,'1':set) -> sig_mar_out

ctrl_word      -> get_bit_26 immed ()                      -> ctrl_word_26
ctrl_word      -> get_bit_27 immed ()                      -> ctrl_word_27

ctrl_word_26,ctrl_word_27     -> or_2p immed ()            -> sig_immedh_enable_ctrl
sig_immedh_enable_ctrl        -> pass immed ()             -> sig_immedl_enable_ctrl

immediate             -> reg8 IMM_l (0,'1','1':sig_immedl_enable_ctrl)    -> IMM_l_out

IMM_l_out             -> imm_log_h 1 (27,26:ctrl_word)                    -> IMM_h_in
IMM_h_in              -> reg8 IMM_h (0,'1','1':sig_immedh_enable_ctrl)    -> IMM_h_out

IMM_l_out,IMM_h_out   -> imm_cat 1 ()                              -> sig_immediate_out

////////////////////////////////////////////////////////////////////////
///// A_IN_MUX e B_IN_MUX
// MUX A
// Entradas
// 1a - Saida do registrador B
// 2a - Saida do registrador PC
// 3a - data_in - Entrada externa do datapath
// 4a - Saida do addsub dos mux iconst e addr

// Saida
sig_b_out,sig_pc_out,data_in,sig_addr_alu_out -> mux4_1 a_in_mux (8,7:ctrl_word) ->
sig_a_in
```

```
// MUX B
// Entradas
// 1a - Saida do registrador A
// 2a - Resultado da ALU de A e B
// 3a - data_in - Entrada externa do datapath
// 4a - Saida do mux iconst

// Saida
sig_a_out,sig_alu_result,data_in,sig_iconst_out -> mux4_1 b_in_mux (10,9:ctrl_word) ->
sig_b_in

////////////////////////////////////////////////////////////////////////
///// Saida dos MUX de A e B nos registradores A e B

sig_a_in -> reg A (5,'1','1':ctrl_word) -> sig_a_out
sig_b_in -> reg B (6,'1','1':ctrl_word) -> sig_b_out

////////////////////////////////////////////////////////////////////////
///// ALU A e B
sig_b_out, sig_a_out -> alu alu0 (25,24,23,22:ctrl_word) -> sig_alu_result

////////////////////////////////////////////////////////////////////////
///// FLAG MUX
// Entradas
// 1a - valor 0
// 2a -> 7a - if_zero,if_not_zero,if_lt_zero,if_ge_zero,if_gt_zero,if_le_zero
sig_alu_result -> get_bit 1 ('0','0','0':nada) -> if1
sig_alu_result -> get_bit 2 ('0','0','1':nada) -> if2
sig_alu_result -> get_bit 3 ('0','1','0':nada) -> if3
sig_alu_result -> get_bit 4 ('0','1','1':nada) -> if4
sig_alu_result -> get_bit 5 ('1','0','0':nada) -> if5
sig_alu_result -> get_bit 6 ('1','0','1':nada) -> if6

// 8a - valor 1
// Saida
no_value_0,if1,if2,if3,if4,if5,if6,no_value_1 -> mux8_1 flag_mux (20,19,18:ctrl_word) ->
sig_flag_mux_out

// Out - Datapath
sig_mar_out    ->     pass d1 ()     ->     data_addr
sig_pc_out     ->     pass d2 ()     ->     instr_addr
sig_b_out      ->     pass d3 ()     ->     data_out_dat

// FIM do DATAPATH
////////////////////////////////////////////////////////////////////////

// Entradas
// data_in
// instr
// int

// Saidas
// inta        =      controle(29)
// data_out    =      data_out = sig_b_out (datapath) - guardado em um REG
// ram_addr    =      data_addr = sig_mar_out (datapath)
// rom_addr    =      instr_addr = sig_pc_out (datapath)
// mem_rw      =      controle(28) - guardado em um FF

// Interface Datapath - Core
// Out
instr_addr     ->     pass dc1 ()    ->     rom_addr
data_addr      ->     pass dc2 ()    ->     ram_addr
data_out_dat   ->     pass dc3 ()    ->     sig_data_out
// In
instr          ->     pass dc4 ()    ->     immediate
// data_in -> data_in

// Interface Control - Core
// Out
// ctrl_word   ->     pass cc1 ()    ->     sig_ctrl_word
// In
// instr -> pass -> instr
// int -> pass -> int
```

```
sig_data_out   -> reg data_ram ('1','1','1':no_value_1)    ->     data_out
ctrl_word      -> get_bit_28 1 ()                          ->     controle_28
controle_28    -> dffa mem_rw_sig ('1','1','1':no_value_1) ->     mem_rw

// Interface Core - SYS - OUT
rom_addr       ->     pass cs1 ()    ->     sig_rom_addr
ram_addr       ->     pass cs2 ()    ->     sig_ram_addr
data_out       ->     pass cs3 ()    ->     sig_data_to_ram
mem_rw         ->     pass cs4 ()    ->     sig_mem_rw

// FIM DO CORE
////////////////////////////////////////////////////////////////////////

// Comecam aqui os outros componentes menores do FJ_SYS...

////////////////////////////////////////////////////////////////////////
// RAM - Memoria RAM....
////////////////////////////////////////////////////////////////////////
// Entradas
sig_dbus_out          -> pass ram1 ()                     -> data_in_ram
sig_ram_addr          -> pass ram2 ()                     -> addrbus
sig_mem_rw            -> pass ram3 ()                     -> rw_ram

data_in_ram,addrbus   -> ram 1 (0:rw_ram)          -> data_out_ram

// Saidas
data_out_ram          -> pass ram4 ()                     -> sig_data_from_ram

////////////////////////////////////////////////////////////////////////
// ROM - Memoria ROM....
////////////////////////////////////////////////////////////////////////

sig_rom_addr          ->     rom 1 ()              ->     databus
databus               ->     pass rom1 ()          ->     sig_data_from_rom

////////////////////////////////////////////////////////////////////////
// IBUS_ARB - Arbitro do "barramento" de Instrucoes
////////////////////////////////////////////////////////////////////////
sig_data_from_rom            ->     pass ibus1 ()         ->     rom_bus
rom_bus                      ->     pass abc ()    ->     ibus_out
ibus_out              ->     pass ibus2 ()         ->     sig_data_from_ibus_arbr

////////////////////////////////////////////////////////////////////////
// DBUS_ARB - Arbitro do "barramento" de dados
////////////////////////////////////////////////////////////////////////
// Entradas
sig_data_from_ram     -> pass dbus2 ()      ->     ram_bus
sig_data_to_ram              -> pass dbus1 ()      ->     core_bus
sig_ram_addr          -> pass dbus4 ()      ->     addr_bus
sig_mem_rw            -> pass dbus5 ()      ->     rw

// Saidas
dbus_out              -> pass dbus24 ()     ->     sig_dbus_out

// Operacoes

addr_bus,nada,nada,nada,nada,nada,nada,nada,ram_bus -> dbus_arb1 1 () -> sig_dbus_out_db
core_bus,sig_dbus_out_db      ->     dbus_arb2 1 (0:rw)    ->     dbus_out

////////////////////////////////////////////////////////////////////////
// Interface com o CORE
dbus_out              -> pass dbusXXX ()    ->     data_in
sig_data_from_ibus_arbr       -> pass ibus3 ()      ->     instr

// FIM DO SYS
////////////////////////////////////////////////////////////////////////

no_value_1 -> pass set () -> set
```