UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA POLITÉCNICA

DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO

Introdução de Predição de Desvios no Processador *FemtoJava* para Sistemas Embarcados

Autora: _____
Luciana Martins da Silva Ozório Santos

Orientador: _____
Prof. Felipe Maia Galvão França

Examinador: _____
Prof. José Paulo Brafman

Examinador: _____
Prof. Amarildo Teodoro da Costa

DEL
Janeiro de 2007

“Algo só é impossível até que alguém duvide e prove o contrário.”

(Albert Einstein)

A meus pais, fonte de apoio e incentivo na vida universitária.

Agradecimentos

Gostaria de agradecer a Deus pela graça de estar me formando em Engenharia, ao professor e querido orientador Felipe França por toda dedicação e apoio, aos colegas da Universidade Federal do Rio Grande do Sul por toda a assistência prestada, aos amigos Luiz Felipe de Souza e Silva, Rodrigo Fonseca Carneiro e Wandenberg Vieira Peixoto por tantos debates e trocas de idéias sem os quais a realização desse trabalho não teria sido possível e por todos os momentos em que compartilhamos o aprendizado durante os cinco anos de faculdade.

Também não poderia deixar de agradecer ao meu melhor amigo e namorado Marcelo Torres de Queiroz por todo incentivo, carinho, compreensão e apoio.

Resumo

O desenvolvimento de um processador exige que sejam observados fatores como desempenho, custo e consumo de energia. Uma das técnicas empregadas visando buscar ganhos para essas três características é a previsão de desvios. Este projeto propõe a implementação de um preditor de desvios clássico de 2 *bits* para o microprocessador *pipeline FemtoJava* para sistemas embarcados e um estudo sobre a futura implementação de um preditor de desvios utilizando a técnica de redes neurais sem peso, visando uma melhoria no sistema de predição de desvios do processador em questão.

Vale ressaltar que, por ser desenvolvido especificamente para sistemas embarcados, o projeto do processador e a introdução de novas características busca soluções que apresentem baixo consumo de área em *chip* e baixo consumo de energia uma vez que o sistema deve ser alimentado por uma bateria.

A Linguagem C foi utilizada para a implementação do preditor de desvios e todo o projeto foi desenvolvido visando sua possível implementação em *hardware* num futuro próximo.

A implementação do preditor mostrou-se válida já que o sistema de predição alcançou percentuais de acerto compatíveis com os preditores conhecidos no mercado e, apesar do ganho em relação ao tempo de execução dos programas não ter sido significativo, uma vez que o *pipeline* do processador apresenta apenas cinco estágios, a introdução da capacidade de prever desvios prepara o processador para a introdução da técnica de reuso dinâmico de traces em nível de arquitetura.

Palavras-chave: preditor de desvio, redes neurais sem peso, *pipeline*, *FemtoJava*, *hazards* de controle

Índice

Índice de Figuras

1 Introdução

Como se pode observar cotidianamente, devido à evolução tecnológica, cada vez mais sistemas computacionais são utilizados para facilitar tarefas do dia-a-dia.Eles são utilizados para diferentes propósitos, como entretenimento, comunicação, controle de eletrodomésticos e veículos, entre vários outros. Estes sistemas são chamados de sistemas computacionais embarcados, já que exemplos práticos de utilização destes são em fornos de microondas, videogames, impressoras, *mp3 players*, câmeras fotográficas digitais e telefones celulares. O uso de processadores e sistemas integrados em silício (SoC) dedicados a este propósito estão em franca expansão.

O desafio no desenvolvimento de sistemas embarcados, especialmente naqueles que são portáteis, é conseguir combinar características de forma a obter um sistema que apresente pouco tempo e baixo custo de produção, uma vez que esses aspectos têm enorme impacto na eletrônica de consumo; baixo consumo de energia, já que geralmente os sistemas dependem de uma bateria; e tempo de computação satisfatório.

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 estas descrições em um alto nível sejam automaticamente transformadas para o *hardware* real.

É nesse cenário de 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 se insere, facilitando a modelagem, a programação e a validação do sistema, já que é baseada em objetos.

Por ser baseada na orientação a objetos, facilita a modelagem do sistema através de alguma linguagem específica, como UML. Além de tudo, JAVA tem a vantagem de ser multiplataforma: todo o sistema pode ser executado e validado previamente em outra plataforma de desenvolvimento, e depois portado para a plataforma alvo.

A linguagem apresenta, também, outras vantagens para ser aplicada em sistemas embarcados: facilita e agiliza o desenvolvimento do produto, pois já é uma linguagem amplamente difundida e utilizada e oferece segurança e pequeno tamanho ocupado de memória

de instruções, já que JAVA foi originalmente desenvolvido para ser transmitido pela Internet. Por estas e outras razões a linguagem JAVA está se tornando cada vez mais popular em ambientes embarcados que, por sua vez, estão em franca expansão.

1.1 Motivação

Com o alto grau de empregabilidade da linguagem JAVA combinado à necessidade de desenvolver processadores cada vez mais eficientes que apresentem baixo consumo de potência e pouca área ocupada no *chip* surgiu a idéia de acrescentar ao já implementado processador *FemtoJava pipeline* [1] a capacidade de predição de desvios, aumentando um pouco sua eficiência uma vez que impedirá paradas no fluxo do *pipeline* devido às instruções de desvio condicional, e preparando o processador para a implementação da técnica de reuso dinâmico de traces em nível de arquitetura.

1.2 Trabalhos relacionados

A base para desenvolvimento deste trabalho é o projeto DeJavaVu (Identificação e Reuso de Computações Redundantes em Processadores JAVA).

Desde o início do projeto, vários trabalhos já foram desenvolvidos visando a implementação de um processador JAVA para sistemas embarcados com capacidade de reuso de computações redundantes, curto ciclo de produção, pouca área ocupada em *chip*, baixo consumo de potência e tempo de computação satisfatório.

Foram feitos estudos de microcontroladores JAVA dedicados [2] e computação redundante e reuso dinâmico de traces [3]. Em [1], o foco é o aumento da performance e redução do consumo de potência e, finalmente, em [4] é vista a memorização e reuso dinâmico de traços para uma arquitetura JAVA.

1.3 Proposta e objetivo do projeto

O projeto propõe a introdução da capacidade de predição de desvios no processador *FemtoJava pipeline* através da implementação de um clássico preditor de desvios de 2 bits. Toda a implementação do preditor é feita utilizando a Linguagem C e visa uma possível implementação em *hardware* num futuro próximo.

O projeto também propõe um estudo para a possível implementação de um preditor neural empregando a técnica de redes neurais sem peso objetivando um aproveitamento ainda

maior da eficiência do trabalho com resultados especulativos utilizando menor área em *chip* e apresentando custos mais baixos.

Por fim, o objetivo deste trabalho é aumentar a eficiência do processador, diminuindo o tempo e as dependências causadas pela execução de desvios condicionais numa arquitetura *pipeline*, além de preparar o *FemtoJava* para receber a capacidade de reuso dinâmico de traces em nível de arquitetura [3].

1.4 Estrutura do texto

A monografia está organizada de acordo com a estrutura que segue:

O Capítulo 2 explica o cenário em que se insere o desenvolvimento deste projeto, explicitando a arquitetura do processador, seu funcionamento e as principais ferramentas utilizadas em sua síntese e as técnicas para redução de custo de desvios.

A implementação do preditor de desvios de 2 *bits* proposta neste trabalho é explicada no Capítulo 3.

No Capítulo 4 são apresentados os resultados experimentais da introdução da técnica de predição de desvios no *FemtoJava*.

O Capítulo 5 propõe trabalhos futuros explicitando uma breve explicação sobre redes neurais sem peso e a idéia de implementação de um preditor neural e apresenta as conclusões deste projeto.

No Apêndice A é apresentado o trecho do código fonte referente à introdução de capacidade de predição de desvios no processador *FemtoJava* , o Apêndice B mostra a descrição da arquitetura utilizada e, finalmente, no Anexo I encontramos um exemplo de arquivo de memória de instruções sintetizado no *SASHIMI*.

2 Cenário

Este capítulo tem por objetivo ambientar o leitor no cenário para o desenvolvimento do preditor de desvios esclarecendo o funcionamento e evidenciando características da arquitetura *pipeline*, da técnica de previsão de desvios e do processador *FemtoJava*, incluindo o simulador *CACO-PS* [1] e o ambiente *SASHIMI* [2].

2.1 Arquitetura *Pipeline*

O desempenho de um processador pode ser aumentado significativamente se técnicas de exploração de paralelismo forem combinadas de forma apropriada em seu projeto. Um exemplo clássico desse tipo de técnica é a utilização do *pipeline, i.e.*, permitir o processamento simultâneo das distintas fases da execução de cada uma das instruções do processador, considerando seqüências de instruções dinâmicas.

A arquitetura *pipeline* permite a execução parcial de até *n* instruções simultaneamente, onde *n* é o número de estágios do *pipeline*, ou seja, quando um processador apresenta este tipo de arquitetura, as operações computacionais são decompostas em estágios processados em série por unidades de *hardware* separadas, tornando o processamento mais otimizado e rápido, no mesmo modo de uma linha de produção.

Em geral, instruções podem ser executadas em até cinco passos bem definidos, de forma que um processador com arquitetura pipeline com n=5 apresenta os estágios de *busca de instruções*, *decodificação*, *execução*, *acesso à memória* e *escrita de resultados* conforme mostrado na Figura 1.

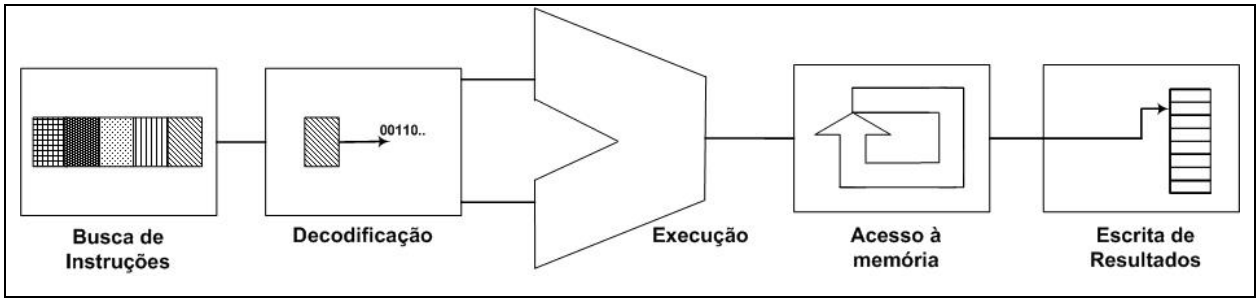

Figura 1: Cinco estágios de um *pipeline* comum.

Considerando o modelo de *pipeline* do processador *FemtoJava* que será utilizado neste trabalho, notamos uma pequena diferença de arquitetura em relação a um *pipeline* comum: apesar de também apresentar cinco estágios, o *pipeline* do *FemtoJava* não realiza acesso à memória uma vez que este processador implementa a pilha de operadores em um banco de registradores ao invés de utilizar a memória principal para este fim. Desta forma, o processador apresenta os estágios de *busca de instruções*, *decodificação*, *busca de operandos*, *execução* e *escrita de resultados* e sua arquitetura é mostrada na Figura 2.

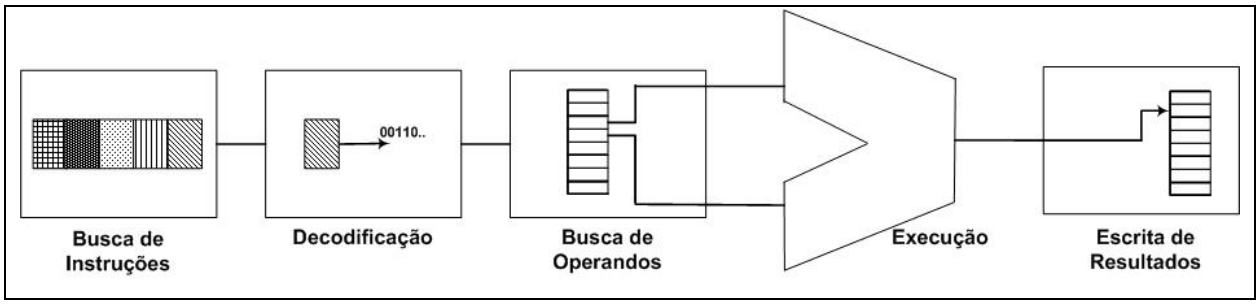

Figura 2: Os cinco estágios do pipeline FemtoJava.

De acordo com a Figura 2, podemos explicar o paralelismo em nível de instrução observando que, enquanto o resultado da *Instrução i* está sendo escrito, a *Instrução (i+1)* encontra-se em execução, operandos da *Instrução (i+2)* são buscados, a *Instrução (i+3)* é decodificada e a busca da *Instrução (i+4)* é realizada, tudo ao mesmo tempo conforme mostrado na Figura 3.

Ciclo de Clock	Busca de Instruções	Decodificação	Busca de Operandos	Execução	Escrita dos Resultados
1	Instrução i				
2	Instrução (i+1)	Instrução i			
3	Instrução (i+2)	Instrução (i+1)	Instrução i		
4	Instrução (i+3)	Instrução (i+2)	Instrução (i+1)	Instrução i	
5	Instrução (i+5)	Instrução (i+3)	Instrução (i+2)	Instrução (i+1)	Instrução i

Figura 3: Paralelismo em nível de instrução do *pipeline* do *FemtoJava.*

Quando um estágio finaliza sua tarefa, envia os resultados do seu trabalho para o próximo estágio e recebe os resultados do estágio anterior fazendo com que o processamento funcione em paralelo para diversas instruções, ou seja, não é preciso concluir todos os estágios do processamento de uma instrução para que se inicie o processamento da próxima.

O tempo para que os resultados de um estágio sejam passados para o próximo é determinado pelo estágio que leva maior tempo para concluir seu trabalho.

Apesar da maior velocidade na execução de instruções com o emprego da arquitetura *pipeline*, o fluxo de instruções sendo executadas simultaneamente pode causar situações de conflito ou dependências também conhecidas como *hazards*, *i. e.*, situações em que a instrução seguinte não pode ser executada no próximo ciclo de *clock*.

Existem três tipos de conflitos:

- Dependências estruturais

  São aquelas relacionadas ao conflito de acesso a recursos comuns. Ocorrem quando o *hardware* não pode suportar a combinação de instruções que o *pipeline* deseja executar no mesmo ciclo de *clock*. As possíveis soluções para esse tipo de dependência são: aumento da disponibilidade de recursos ou escolha de uma das instruções para esperar a conclusão da outra.

- Dependências de dados

  Ocorrem quando a execução de uma instrução depende do resultado de outra que ainda está em estágio anterior no *pipeline*. Para amenizar os atrasos devido a dependências de dados utiliza-se a técnica de adiantamento ou *bypass* que consiste em obter antecipadamente um determinado operando faltante a uma operação a partir de recursos internos de máquina.

- Dependências de controle

  Originárias da necessidade de tomar uma decisão com base nos resultados de uma instrução, enquanto outras estão sendo executadas. Conhecidas também como dependências procedurais, estão relacionadas a desvios condicionais.

Quando um processador de arquitetura *pipeline* tem que parar para poder executar uma instrução de desvio condicional, ele deve interromper a progressão das instruções pelo *pipeline*, desta forma, a instrução que deve ser executada se o desvio condicional não se realizar, sofre um atraso de pelo menos um ciclo de *clock* [5]. Essa parada do *pipeline* gerando atraso para a execução das instruções subseqüentes, que também é conhecido como inserção de bolhas no *pipeline*, é o que este projeto se propõe a amenizar uma vez que o uso de técnicas de previsão de desvios é a solução tradicionalmente empregada para o problema das dependências de controle.

2.2 Técnicas de Redução do Custo de Desvios

As instruções de desvio condicional representam em média 20% [6] do total de instruções de um programa. Dessa forma, quando o objetivo é explorar o paralelismo do código, ou seja, executar múltiplas instruções no mesmo ciclo de *clock*, as dependências de controle tornam-se um fator limitante, uma vez que cada instrução de desvio condicional provocará uma parada na execução do programa.

Para evitar pausas no *pipeline* entra em cena a técnica de execução especulativa que consiste em antecipar a computação de instruções críticas executando antecipadamente operações que “podem” ser necessárias mais adiante no programa e, no caso de posterior

descoberta de que essas operações não deveriam ter sido executadas, os resultados das mesmas são ignorados.

Existem diversas técnicas para especular o resultado de um desvio e evitar a dependência de controle reduzindo a degradação no desempenho, ou seja, técnicas para definir prematuramente o resultado de um desvio e permitir a execução do programa sem pausas.

Utilizando esse resultado especulativo, as instruções pertencentes ao fluxo com maior probabilidade de execução podem ser buscadas, decodificadas e executadas antecipadamente. Porém é necessário que o processador tenha mecanismos para desfazer eventuais computações provenientes de caminhos previstos erroneamente, acrescentando complexidade no *hardware* resultante.

Considerando como critério o nível de implementação, identificamos dois tipos de técnicas de redução do custo de desvios: técnicas implementadas em *software* e técnicas implementadas em *hardware*.

2.2.1 Técnicas Implementadas em *Software*

Empregadas durante a compilação do programa de aplicação onde parte do problema é transferida para o nível de *software*. O *hardware* que hospeda a aplicação retarda a execução do comando de desvio, processando em paralelo uma ou mais instruções subseqüentes e, compete ao *software* de suporte a tarefa de reorganizar as instruções do programa, indicando assim quais as instruções que serão executadas em paralelo com o comando de desvio. Nas subseções subseqüentes há uma resumida explicação sobre o funcionamento de algumas técnicas implementadas em *software*.

2.2.1.1 Delayed Branch

Consiste na reorganização das instruções do programa preservando a equivalência semântica dos programas e minimizar os retardos impostos pela ramificação.

O compilador tenta movimentar as instruções do programa de aplicação, alocando-as após o comando de desvio condicional. Para fazer estas movimentações, o compilador deve levar em consideração as relações de dependência entre as instruções, mantendo assim a semântica do programa.

2.2.1.2 Branch Folding

Algumas arquiteturas, no seu conjunto de instruções, incluem instruções específicas para configurar as condições de desvio (*flags* de condição) e trocar o fluxo de controle. Condições de desvio são tipicamente realizadas como o resultado de uma comparação específica entre dois registradores, ou como o resultado de uma operação aritmética. Neste caso, é possível que a instrução que resolva a condição em que o desvio esta baseado tenha completado sua execução e o resultado do desvio seja conhecido quando o desvio é encontrado. Este "desvio resolvido" pode ser removido do fluxo de instruções e substituído pelo seu sucessor lógico. Com a técnica de *branch folding* é alcançado o efeito de um desvio de ciclo zero. Esta técnica foi implementada no processador CRISP da AT&T e é utilizada no PowerPC 601 e no IBM RISC System 6000.

2.2.1.3 In-line

As técnicas de predição de desvios apresentam uma reduzida taxa de acertos no caso de tratamento de instruções de retorno de funções, já que um procedimento pode ser chamado de diferentes pontos do programa e, portanto, a técnica de predição precisaria armazenar longos padrões de ativações/retornos para aumentar a taxa de acerto.

As instruções geradas pelo compilador para a troca de parâmetros e transferência do controle entre o procedimento e o ambiente que o ativou, representa uma parcela não desprezível do tempo de processamento do programa. Este fato motivou o desenvolvimento de técnicas de otimização de código como a *in-line*, que consiste em substituir as *procedures* dos programas pelo código objeto correspondente nos locais onde estas são ativadas. O tempo de execução do programa com *procedures* codificadas é menor, pois as instruções de chamada/retorno se tornam redundantes e são retiradas eliminando, assim, as instruções para passagem de parâmetro. A única desvantagem é que o código objeto tem seu tamanho aumentado.

2.2.1.4 Desenrolamento de *loops*

Esta técnica reduz o custo das instruções de desvios condicionais existentes no comando *for.* O mapeamento, quando possível, em registradores da variável de controle e do número de iterações elimina instruções de *load/store* produzindo uma significativa redução no tempo de processamento da estrutura *for.*

2.2.2 Técnicas implementadas em *Hardware*

Implementadas pela unidade de controle do processador, essas técnicas atuam em tempo de execução do programa, diferentemente das técnicas implementadas por *software*.

Existem dois tipos de técnicas implementadas por *hardware*: as técnicas estáticas, onde a previsão ocorre baseado em definições feitas em tempo do projeto de um novo processador, e as técnicas dinâmicas, que como o próprio nome diz, realizam dinamicamente as predições de desvio baseando-se nas informações coletadas em tempo de execução.

2.2.2.1 Previsão estática

Na previsão estática o resultado especulativo de um desvio é sempre o mesmo e pode ser executado em nível de *software*, durante a compilação, ou em nível de *hardware*, em tempo de execução. As previsões estáticas podem ser implementadas segundo três variações: o desvio sempre ocorrerá, nunca ocorrerá e o código da operação determina a previsão.

A primeira técnica explora o fato de que a maioria dos desvios condicionais provoca uma transferência no fluxo de controle, assim, a unidade de busca de instruções busca a instrução contida no endereço alvo ao invés do trecho de código adjacente ao comando de desvio. Esta técnica foi utilizada pelo IBM 360/91.

A segunda técnica de previsão é implementada na maioria dos processadores que fazem busca antecipada de instruções (*look-ahead processors*). Assumindo que a transferência de controle nunca ocorrerá, a unidade de instruções continua buscando as instruções adjacentes ao comando de desvio. Esta técnica foi implementada em vários processadores, como por exemplo, o i960CA, MC68020 e o VAX 11/780.

Finalmente, a terceira técnica, ao invés de fixar uma única direção, leva em conta o código de operação do comando de ramificação para decidir se o fluxo de controle será transferido para o endereço alvo ou não, já que alguns códigos de operação de desvios têm uma tendência maior para um dos fluxos. Essa técnica utiliza resultados de estudos sobre o comportamento dos diversos tipos de comandos de desvio, levando em consideração a probabilidade de o controle ser transferido.

Com o objetivo de fixar a direção que será utilizada, o projetista monitora o comportamento dos comandos de desvio durante a simulação de programas representativos e determina o número de vezes que cada tipo de comando provocou uma transferência de controle. Com isso, utiliza uma freqüência dinâmica de cada tipo de desvio e a predição é fixada e armazenada, por exemplo, em uma memória ROM no interior do processador, a qual é consultada em tempo de execução para decidir se o desvio ocorrerá ou não.

2.2.2.2 Previsão dinâmica

Em alguns processadores, a unidade de controle realiza dinamicamente a predição de desvios. Em geral, essas técnicas são mais eficientes do que as estáticas, pois armazenam informações das instruções de desvio coletadas em tempo de execução e quando o desvio for novamente executado, o mecanismo de predição verifica o que ocorreu no passado mais recente e, baseado nessa informação, especula o resultado que será produzido pela instrução de desvio.

As informações indicando o que ocorreu recentemente quando da execução de alguns comandos de desvio, ficam armazenadas numa pequena tabela localizada no interior do processador. Essa tabela é denominada Tabela de História dos desvios(*Branch History Table*). Por exemplo, o processador pode incluir uma pequena tabela para armazenar informações relacionadas com as mais recentes execuções dos comandos de desvio. Os campos de cada entrada podem conter ou o endereço do desvio e o endereço da instrução sucessora ou o endereço do desvio e a instrução sucessora. No primeiro caso, tão logo uma instrução tenha sido buscada, o mecanismo de previsão de desvios, pode iniciar a busca da instrução sucessora. O endereço da instrução é usado como chave para acesso à tabela. Se a instrução estiver armazenada no campo de endereço de desvio isto significa que o endereço no campo endereço da sucessora será utilizado para buscar a próxima instrução. Neste caso, a técnica só prevê que um desvio será tomado se o endereço dele for encontrado na tabela, caso contrário, o mecanismo opta por seguir o fluxo seqüencial.

No segundo caso, ao invés de termos o endereço da instrução sucessora, temos a própria instrução. Com isso, além de manter as vantagens do esquema anterior, elimina-se a busca antecipada, já que a instrução sucessora já se encontra na tabela.

Essa tabela é manipulada da mesma forma que no primeiro caso, dispensando, porém, a busca antecipada. Por esta razão, além de ser utilizada pelo mecanismo de previsão, ela desempenha o papel de uma janela alternativa de instruções, isto é, uma segunda janela contendo comandos provenientes do fluxo de controle que será executado se o desvio ocorrer.

Para diminuir o tempo de acesso a essa tabela, pode ser utilizada uma memória associativa na sua implementação. Assim, o conteúdo do PC pode ser comparado simultaneamente com todos endereços contidos na tabela. Se um dos endereços for o mesmo, a correspondente instrução sucessora pode ser obtida imediatamente.

Devido ao custo, estas tabelas possuem um número limitado de entradas, o que faz com que nem todos comandos de desvio de um programa possam ser armazenados nela ao mesmo tempo. Para fazer a substituição e o gerenciamento dessas tabelas podemos usar os mesmos algoritmos de substituição utilizados pelos Sistemas Operacionais na implementação de memória virtual.

2.2.2.2.1 Predição determinada pela história do desvio

Essa técnica verifica o que ocorreu com as *k* mais recentes execuções de um desvio e realiza uma predição do resultado que será produzido pela corrente execução do desvio. Os *k* mais recentes resultados de cada desvio ficam armazenados numa Tabela da História dos Desvios (*BHT - Branch History Table*) que é atualizada após a conclusão da instrução de desvio.

Fisicamente, as entradas contendo a história dos desvios podem ser armazenadas num conjunto de registradores ou então numa memória cache no interior do processador.

Quando uma instrução é acessada, ela é pré-decodificada para que seja determinado se aquela é uma instrução de desvio. Se for o caso, os bits menos significativos do endereço do desvio são utilizados para indexar a *BHT*. O bit na entrada selecionada fornece a previsão do desvio: por exemplo, 1 (um) indica que o desvio será previsto como tomado, enquanto 0 (zero) indicaria desvio não-tomado. A instrução a ser acessada no próximo ciclo é determinada de acordo com esta indicação.

No estágio de execução, o estado do *bit* na *BHT* é comparado com o resultado do desvio, para verificar se a previsão foi correta. Caso seja, a execução prossegue normalmente e, caso contrário, as instruções buscadas antecipadamente são descartadas e a busca é redirecionada para o destino correto.

Um esquema bastante simples de predição consiste em utilizar o resultado da última execução da instrução de desvio. Nesse caso, um *bit* seria suficiente para armazenar o resultado anterior da instrução de desvio. Se a predição indicar que o desvio deve ser tomado e se o estágio de execução indicar o contrário, a tabela *BHT* é atualizada, as instruções nos estágios precedentes são descartadas e o estágio de busca inicia a transferência de instruções pertencentes ao fluxo apropriado. Se a instrução de desvio estiver sendo executada pela primeira vez, utiliza-se uma das duas técnicas estáticas apresentadas previamente e em seguida, inclui-se o desvio na *BHT*. Um esquema de predição utilizando 1 *bit* de história pode ser visto na Figura 4.

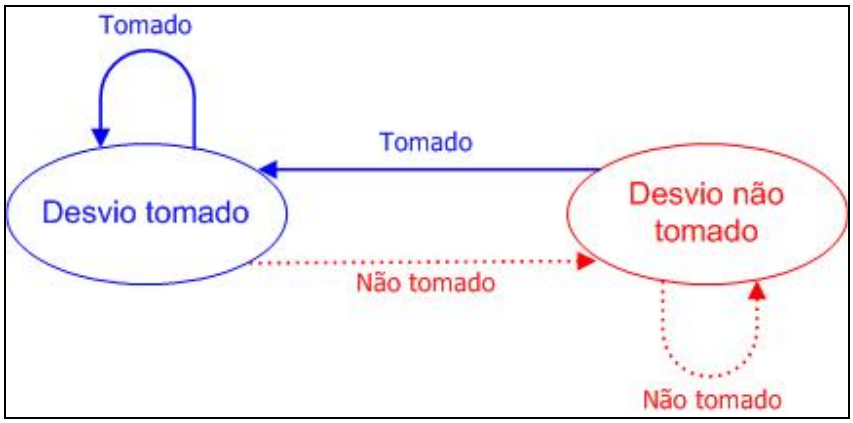

Figura 4: Autômato para implementação de preditor de 1 *bit*.

O número de *bits* de história (previsão) é um fator de extrema relevância na escolha do algoritmo de previsão. Acima foi mostrado um autômato para previsão com 1 *bit* de história. O maior problema em se usar esta técnica é quando se faz necessário prever o destino de desvios de controle de laços, e o laço é executado mais de uma vez (*loops* aninhados). Para *n* iterações de um *loop*, as primeiras *n-1* iterações são tomadas e o desvio ao final do *loop* é previsto corretamente. Entretanto, ao final da última iteração o desvio é não-tomado e a previsão é incorreta, uma vez que, durante a última execução o desvio foi tomado.

A próxima vez que o *loop* é executado, o desvio de controle é tomado e mais uma vez o algoritmo erra a previsão, pois da última vez o desvio fora não-tomado. Usando-se 1 *bit* de

história é necessária uma previsão errada para que o algoritmo passe a prever a situação inversa. Se a previsão inicial é de desvio tomado e o resultado é não-tomado, o algoritmo passa a prever não-tomado na próxima execução.

Outro esquema que poderia ser utilizado é o mecanismo com 2 *bits* de história, assim é possível registrar o resultado das duas últimas execuções, e a próxima previsão é modificada apenas se as duas últimas previsões foram incorretas. Nos estados onde os dois *bits* coincidem, a previsão segue o resultado indicado por ambos. Nos estados onde os dois *bits* diferem, a previsão segue a indicação do *bit* que registra o estado mais antigo, conforme mostrado na Figura 5. Estudos realizados mostram que, com 2 *bits* de previsão, é possível alcançar uma taxa média de acerto individual de 90%[7].

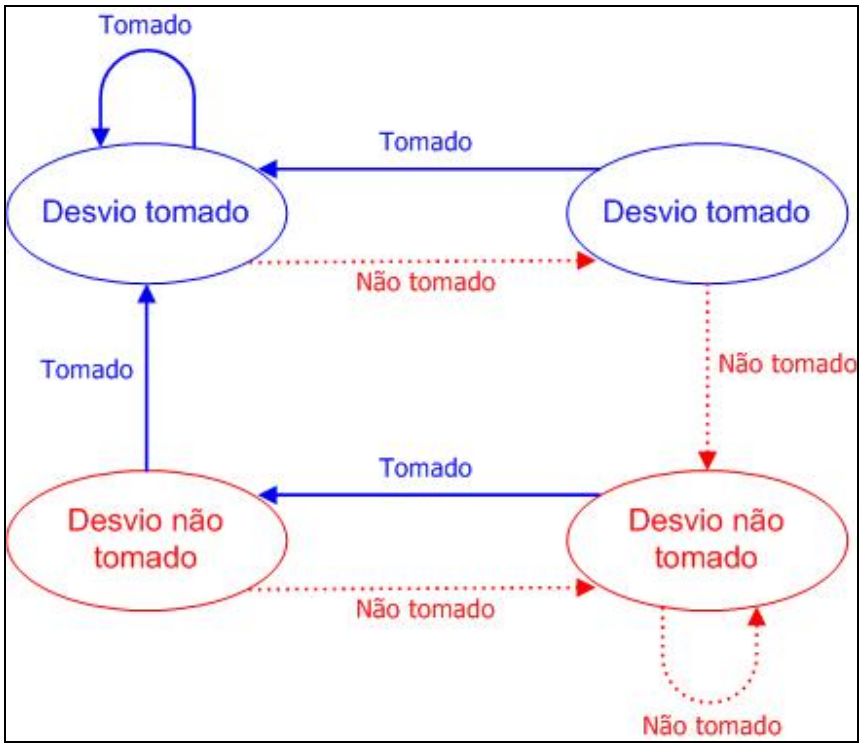

Figura 5: Organização do autômato para preditor de 2 *bits*.

O preditor de 2 *bits* de história será utilizado no presente projeto para acrescentar a capacidade de predição de desvios ao processador *FemtoJava*.

2.2.2.2.2 Previsão usando contador saturado

Nos experimentos descritos por James Smith, a substituição dos *bits* de história por contadores (representados em complemento a dois) aumentou as percentagens de acerto da técnica de predição. Se o contador não for negativo, a técnica prediz que o desvio será tomado. Caso contrário, a previsão indicará que a instrução adjacente é a sucessora. Após a execução do comando de desvio, o contador será incrementado ou decrementado se o desvio for respectivamente, tomado ou não. A operação incrementar/decrementar é desabilitada quando um dos valores extremos do contador for atingido.

Essa técnica produziu previsões mais precisas do que as técnicas baseadas nos *bits* de história descritas anteriormente. Nesses experimentos, o autor usou uma tabela *hash* somente com 16 entradas e por esse motivo, diferentes desvios podem ser mapeados na mesma entrada.

2.2.2.2.3 Previsão via tabela com alvos dos desvios (Branch Target Buffer)

Uma técnica alternativa para previsão é a que emprega uma tabela contendo os alvos das instruções de desvios. Denominada *BTB* (*Branch Target Buffer)*, essa tabela é uma evolução da tabela que contém a história dos desvios e seu modelo pode ser visto na Figura 6.

Como anteriormente, a tabela *BTB* inclui campos para identificar a instrução de desvio e para armazenar a história das recentes execuções do comando de desvio (ou o contador saturado). Adicionalmente, a *BTB* inclui um campo contendo informações sobre a instrução sucessora do desvio: geralmente o campo armazena o endereço efetivo da sucessora; em outras implementações, a instrução sucessora também.

A *BTB* torna o processador mais eficiente do que aqueles que usam simplesmente uma *BHT* por causa do potencial oferecido pelas informações sobre a sucessora do desvio. Em paralelo com a busca da próxima instrução, podemos detectar antecipadamente se ela é um desvio e se esse for o caso, qual o endereço da instrução que deve suceder o comando de

desvio que está sendo buscado. Através dessa antecipação, o estágio de busca pode ser prontamente redirecionado para o fluxo com maior probabilidade de execução, reduzindo, desse modo, a incidência de instruções introduzidas indevidamente nos estágios iniciais do *pipeline*.

Identificação do desvio	Bits de história	Informação sobre instrução sucessora
0xA2	10	0x64
0x9A	01	0xAB
0xA6	11	0x0F

Figura 6: Modelo de organização da *Branch Target Buffer*.

A *BTB* funciona da seguinte maneira: o estágio de busca compara o endereço da instrução que está buscando com os endereços que estão na *BTB*. Se o endereço está na *BTB* então uma previsão é feita em função dos *bits* de história correspondentes. Se a previsão diz que o desvio será tomado, então o endereço no campo de destino será usado para acessar a próxima instrução. Quando o desvio é resolvido, no estágio de execução, a *BTB* pode ser corrigida com a informação correta sobre o que aconteceu com o desvio, caso a previsão feita anteriormente tenha sido incorreta.

O funcionamento é semelhante ao do mecanismo com *BHT* associativa, com apenas algumas diferenças. No caso de *miss*, o endereço destino também é inserido na entrada juntamente com o endereço do desvio. Quando acontece uma previsão incorreta, o endereço destino é atualizado para refletir o destino correto do desvio.

2.3 Processador FemtoJava

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 68 instruções JAVA 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 1** mostra o conjunto de instruções suportadas pelo *FemtoJava*.

Tabela 1: Instruções suportadas pelo *FemtoJava Low-Power*.

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_ml**, **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**

O processador 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. Outras características do *FemtoJava,* além do conjunto reduzido de instruções, são arquitetura *Harvard*, pequeno tamanho e facilidade de inserção e remoção de instruções, além disso, por ter sido criado especificamente para sistemas embarcados, apresenta restrições de área e potência.

A organização de memória é baseada na alocação de *frames* como manda a especificação da linguagem JAVA e pode ser observada na Figura 7(a). Comparado com outros processadores a alocação de *frames* no *FemtoJava* é bem mais simples. O esquema implementado pelo *FemtoJava* é compatível com a especificação JAVA e pode ser observado na Figura 7(b).

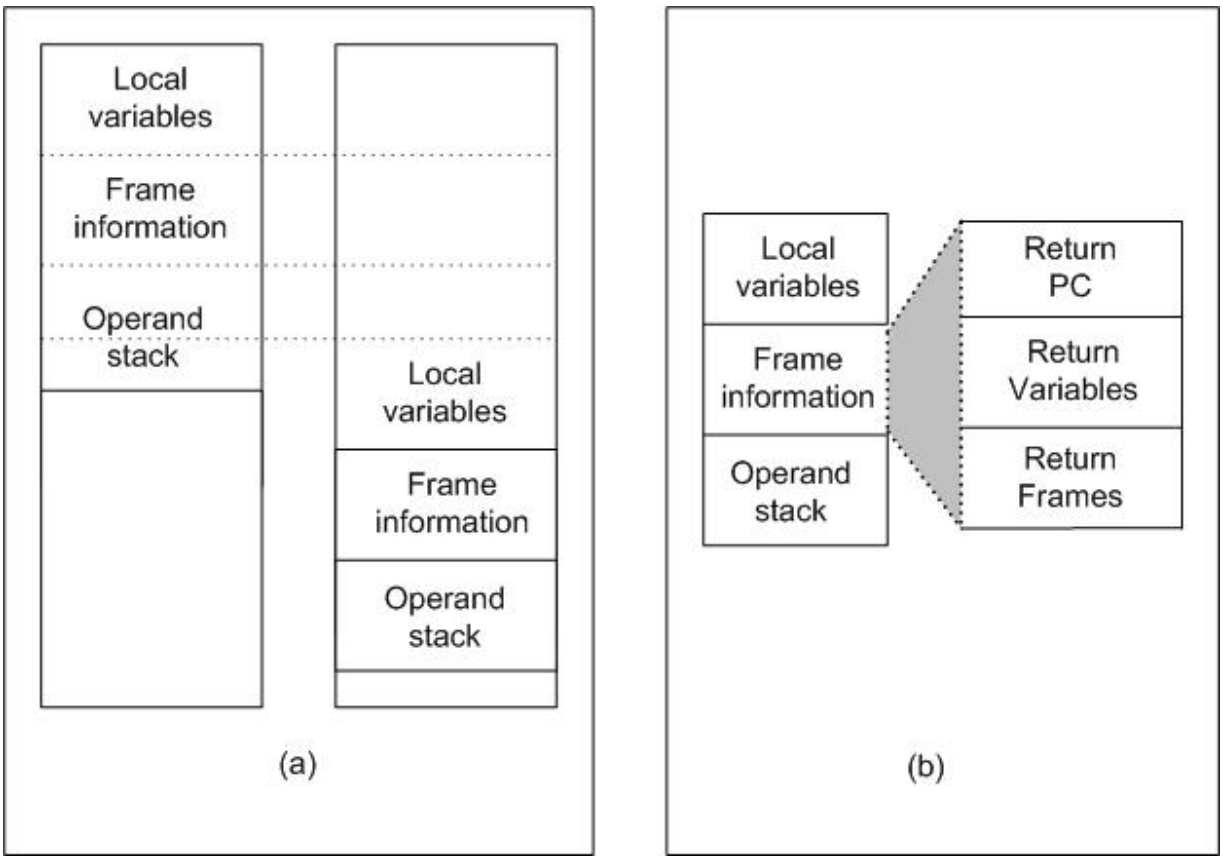

Figura 7: Funcionamento da memória do processador *FemtoJava*.

Por ser gerado a partir de um ambiente de CAD (*Computer Aided Design*), chamado *SASHIMI*, o *FemtoJava* apresenta tamanho da máquina de controle diretamente proporcional ao número de instruções utilizadas.

Conforme já citado, a versão *FemtoJava Low-Power* utilizada para a implementação desse projeto, possui um *pipeline* de cinco estágios: *busca de instruções*, *decodificação*, *busca de operandos*, *execução* e *escrita de resultados*.

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.

A seguir são apresentados alguns detalhes de cada um dos cinco estágios.

⇒ **Estágio 1 – Busca de instruções**

O primeiro estágio do *pipeline* é o de busca de instruções, que faz a requisição de instruções da memória de programa através de uma palavra de 32 *bits* e está representado na Figura 8.

Este estágio 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 buscados 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.

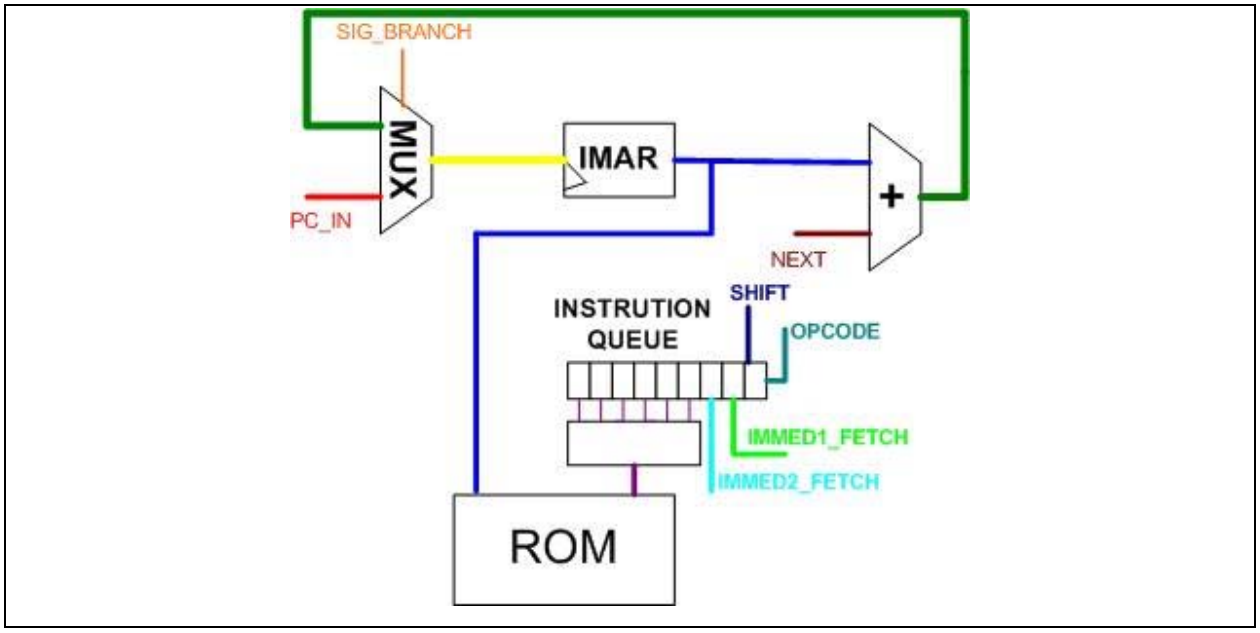

Figura 8: Estágio de busca de instruções.

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 seqüencialmente 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. Na implementação sem o preditor de desvios, são necessários três ciclos de *clock* 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, conseqüentemente, o IPC (Instruções Por Ciclo) do processador seria prejudicado.

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

⇒ **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 9.

⇒ **Estágio 3 – Busca de Operandos**

É neste estágio que os operandos são escolhidos e buscados para a execução da instrução. O estágio 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.

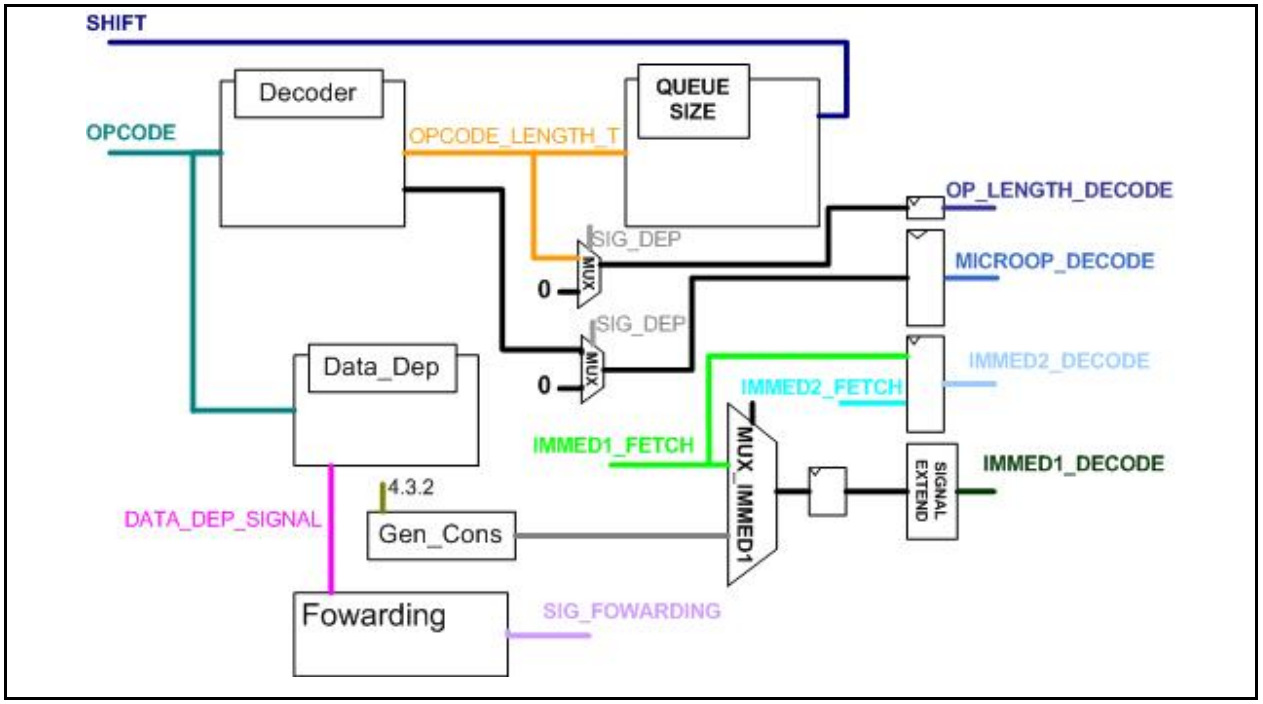

Figura 9: Estágio de decodificação.

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 10 ilustra este estágio.

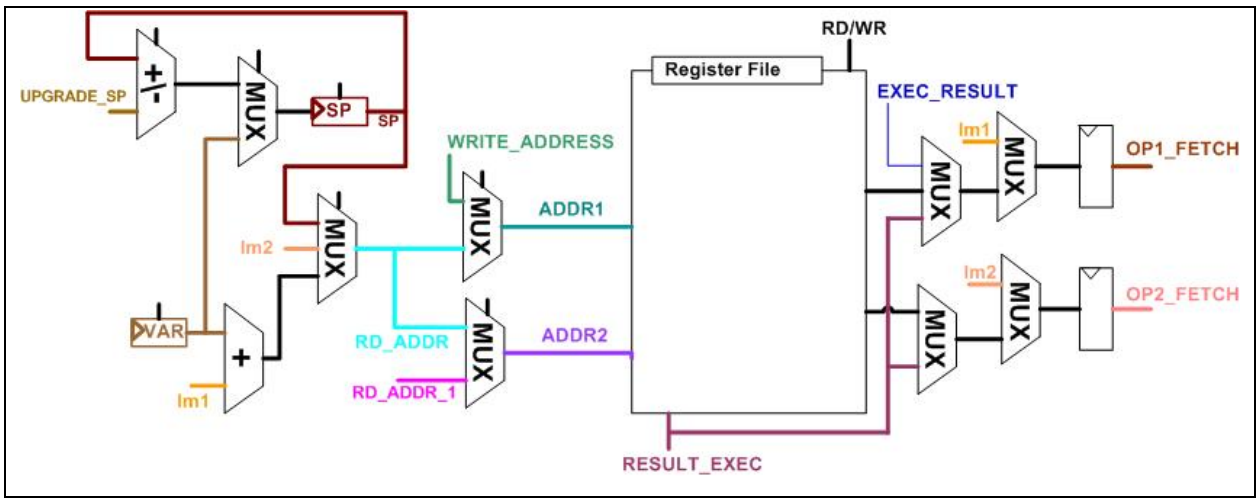

Figura 10: Estágio de busca de operandos.

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.*

⇒ 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 11.

⇒ **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 da maneira que foi implementado em FPGA, 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 12 mostra a escolha do endereço e do valor a ser gravado no banco de registradores.

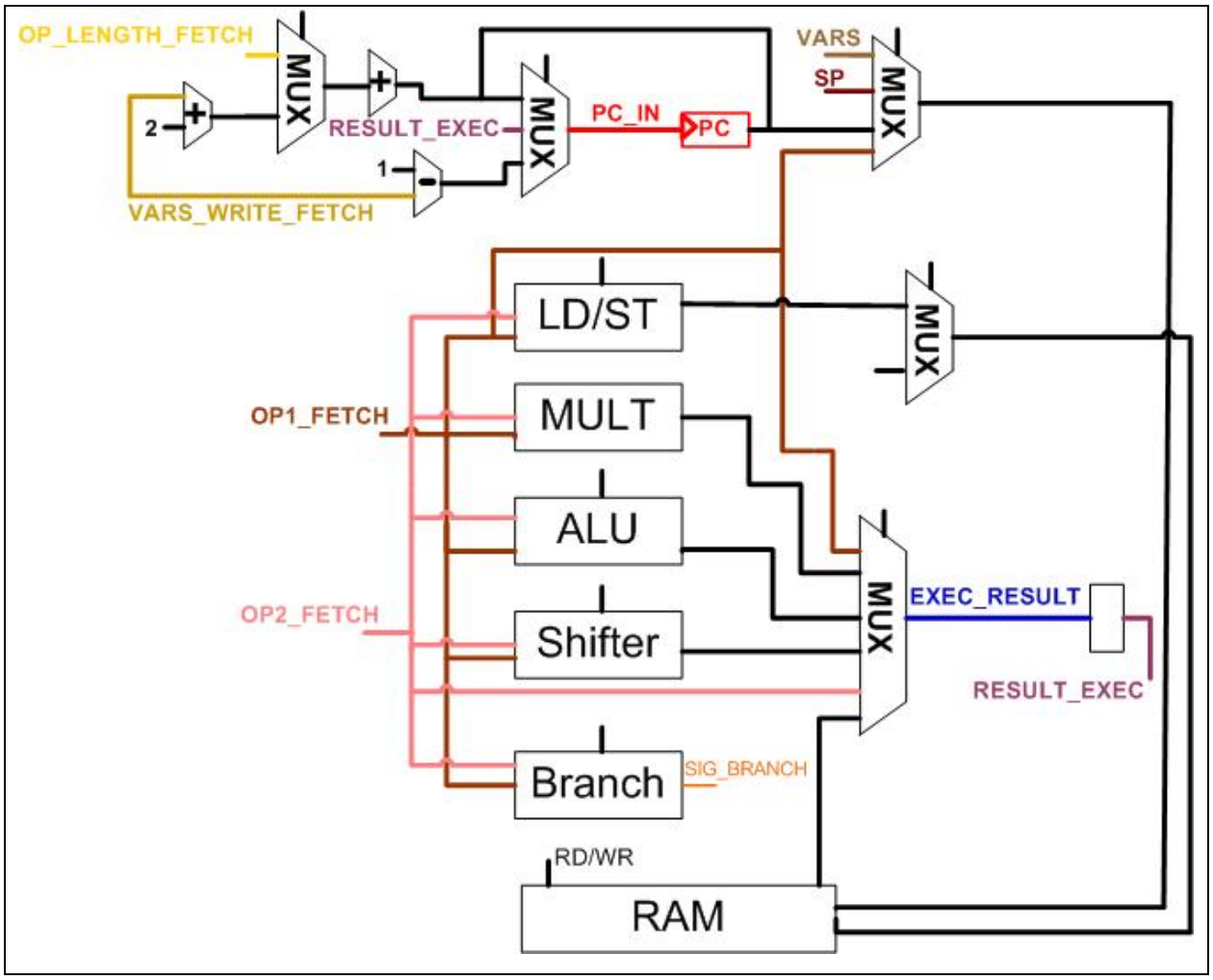

Figura 11: Estágio de execução.

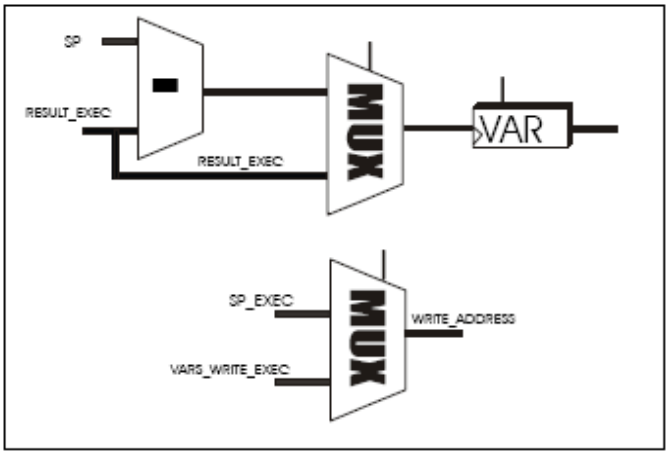

Figura 12: Estágio de escrita de resultados [1].

2.3.1 CACO-PS

O CACO-PS (***C**ycle-**A**ccurate **CO**nfigurable **P**ower **S**imulator*) é um simulador de código compilado, que calcula a potência baseado em ciclos de relógio. Apesar de ser um simulador de código compilado, ele oferece a possibilidade da descrição estrutural de qualquer arquitetura. Além disso, é possível descrever a arquitetura em diferentes níveis de abstração, conforme o nível de detalhamento desejado pelo projetista.

O simulador CACO-PS foi escrito totalmente na linguagem C, tornando possível a compilação em diferentes sistemas operacionais, garantindo 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.

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

• Aquele que descreve, através de uma sintaxe própria e estruturada, a arquitetura desejada;

• Aquele 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 , este arquivo basicamente é formado por um conjunto de linhas, onde em cada há um endereço de memória e depois seu valor. A Figura 13 ilustra o funcionamento do simulador.

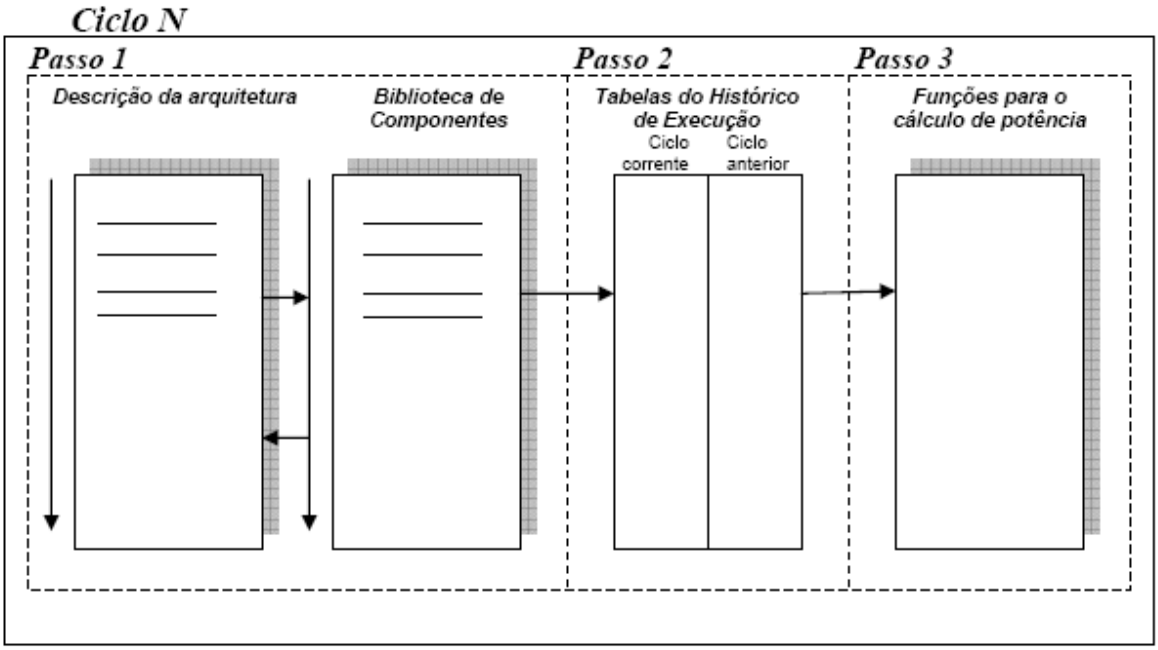

Figura 13: Funcionamento do simulador CACO-PS[1].

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.

O passo 1, demonstrado na Figura 13, 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 14), 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 haja 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 haja algum sinal cujo valor seja diferente, repete-se a operação até que 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.

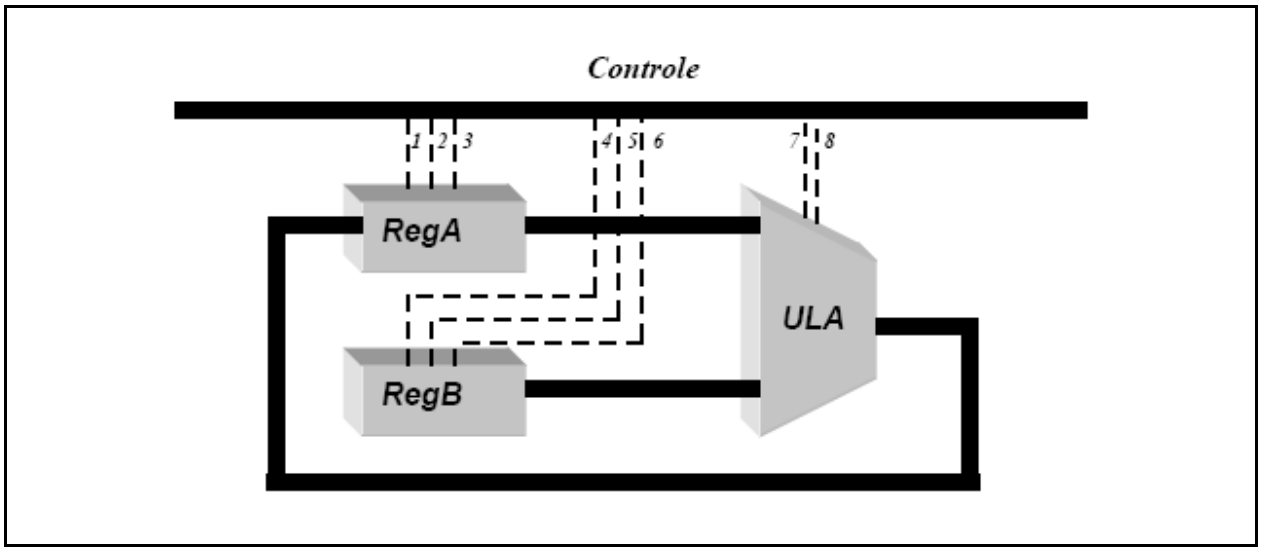

Figura 14: Pequeno circuito a ser simulado[1].

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.

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, além disso, o simulador oferece uma série de facilidades e macros para a construção destes componentes.

2.3.2 SASHIMI

Trata-se de um ambiente destinado à síntese de sistemas microcontrolados especificados em linguagem JAVA. O ambiente *SASHIMI* (***S****ystem* ***A****s* ***S****oftware and* ***H****ardware* ***I****n* ***MI****crocontrolers*) utiliza as vantagens da tecnologia JAVA e fornece ao projetista um método simples, rápido e eficiente para obter soluções baseadas em *hardware* (HW) e *software* (SW) para microcontroladores. O conjunto de ferramentas disponível no *SASHIMI* também foi desenvolvido inteiramente em JAVA, tornando-o altamente portável para diversas plataformas.

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 15.

O *hardware* resultante do sistema *SASHIMI* é composto essencialmente por um microcontrolador *FemtoJava* dedicado a aplicação modelada cuja operação é compatível com a operação da Máquina Virtual JAVA. As informações extraídas na etapa de análise de código permitem determinar um conjunto de instruções, quantidade de memória de programa e de dados, tamanho da palavra de dados e demais componentes adequados aos requisitos da aplicação alvo, podendo o projetista interferir nos resultados desse processo.

O modelo do microcontrolador resultante é descrito em linguagem VHDL e sintetizável por ferramentas como MaxplusII da *Altera Corporation*. A principal adaptação da arquitetura do microcontrolador consiste em modificar o mecanismo de decodificação de forma a reconhecer apenas o subconjunto de instruções contidas na aplicação. Em função dos diferentes formatos e da complexidade de algumas instruções JAVA, o *hardware* de decodificação se torna proporcionalmente complexo, de forma que um menor número de instruções suportadas permite uma economia significativa de recursos (células lógicas em FPGA), possibilitando ainda a integração de outros componentes no mesmo chip.

A máquina de controle sofre modificações no processo de adaptação mediante a geração apenas das palavras de controle correspondentes às instruções suportadas.

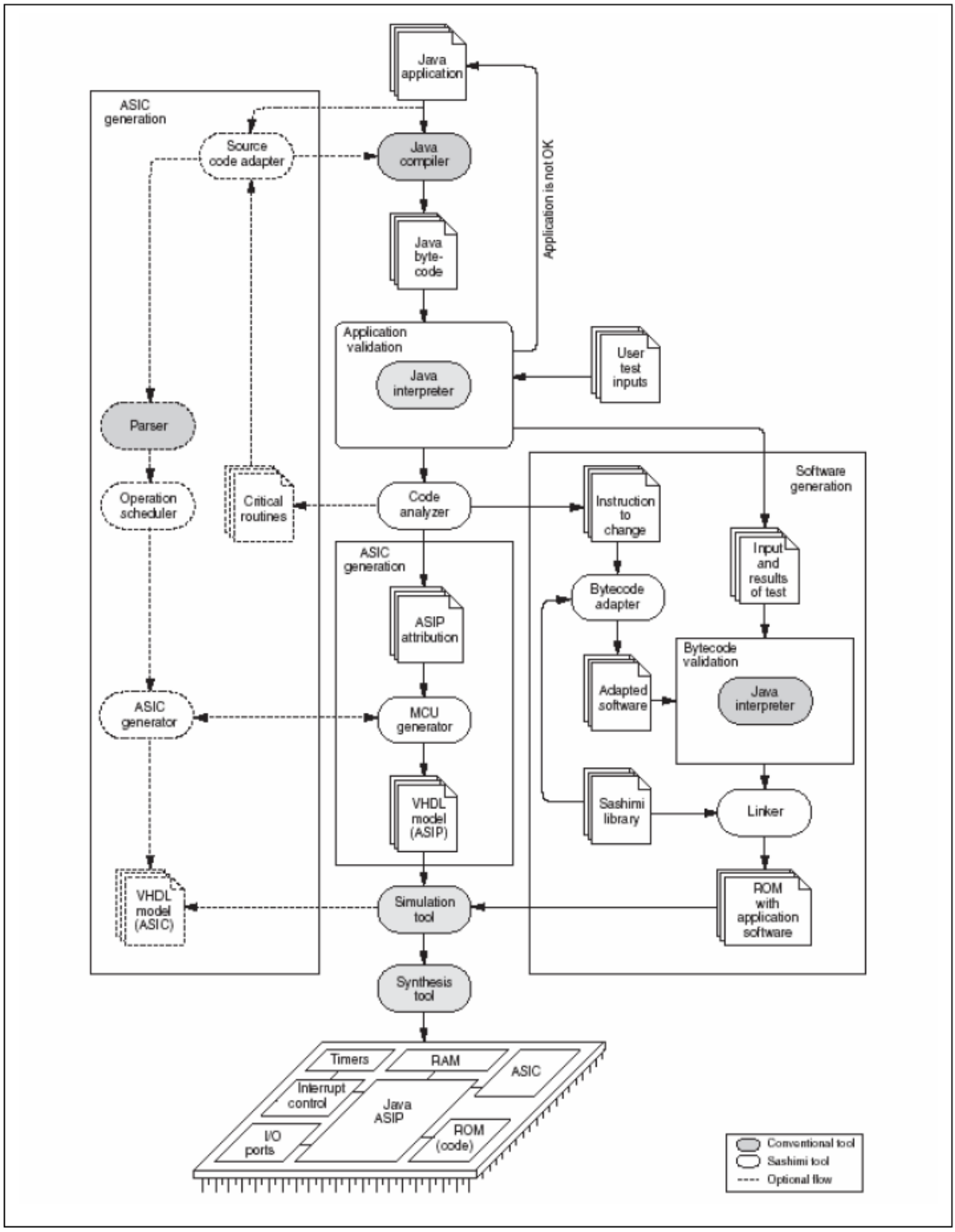

Figura 15: Fluxo de projeto automatizado da ferramenta *SASHIMI* para geração *ASIP* do *FemtoJava* [1].

3 Implementação do Preditor Clássico de 2 bits

O primeiro obstáculo a ser superado é conseguir identificar dentro do conjunto de instruções que se encontram na fila, quais delas são de desvio.A tarefa seria bastante simples se não fosse o fato de que as instruções suportadas pelo processador *FemtoJava* apresentam tamanho variável podendo ter nenhum, um ou dois operandos imediatos, de forma que para conseguir identificar corretamente as instruções de desvio, o primeiro passo é determinar exatamente onde começa e termina cada instrução da fila.

O estágio de busca de instruções, primeiro estágio do *pipeline* do processador *FemtoJava*, é composto por uma fila 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, conforme explicado anteriormente.

A busca por instruções de desvio é realizada da seguinte forma: dentro de um *loop* for, lê-se o conteúdo da fila de registradores e informa-se o tamanho da instrução encontrada, com essa informação calcula-se o próximo registrador da fila a ser lido e, no caso da instrução apresentar tamanho igual a 3, testa-se a possibilidade da mesma estar dentro do conjunto de instruções de desvio.

Depois de identificada a instrução como sendo de desvio, o próximo obstáculo é descobrir uma maneira de identificar o PC da instrução de desvio.

Para calcular o PC, foram criadas três variáveis auxiliares:

k: variável que indica a posição da fila onde será colocada a próxima seqüência de instruções.

imar_temp: variável que armazena o valor do IMAR

atualiza_imar: *flag* que indica quando ***imar_temp*** deve ser atualizado, ou seja, quando deve guardar o valor apontado pelo IMAR na transição do *clock*.

O procedimento do cálculo do PC se desenrola da seguinte forma:

Quando o *fetch* está ligado, ou seja, quando a gravação de valores vai ser realizada, se ***atualiza_imar*** estiver configurado para 1, ***imar_temp*** armazena o valor do IMAR no próximo evento de *clock* e configura ***atualiza_imar*** para 0. No caso do *fetch* estar ligado, mas

atualiza_imar estar configurado para 0, a única ação realizada é trocar a configuração de ***atualiza_imar*** para 1.

Dentro do *loop* de leitura da fila de registradores, o valor do PC é determinado com a seguinte expressão:

$$PC = imar\_temp - (k\text{-}i);$$

onde i é a variável de controle do *loop* e o valor de *k* é dado pelo sétimo, quinto e sexto *bits* de controle do componente ***shift_reg_dec4.***

Após identificar e calcular o PC relativo à instrução de desvio, é passado para o componente ***table*** um *flag* que indica que um desvio condicional foi encontrado, o PC deste desvio e o *offset* que deve ser somado ao PC no caso do desvio ser tomado.

O componente ***table*** é responsável por pesquisar em uma tabela de 2048 entradas se existe o PC do desvio. A opção por uma tabela de 2048 entradas foi feita baseada em estudos de implementações de preditor de 2 *bits*. Essa tabela guarda o PC do desvio, o PC da próxima instrução no caso do desvio ser tomado e os dois *bits* de história do desvio.

Feita a busca, no caso em que o PC do desvio se encontra na tabela, o componente ***table*** deve passar para o componente ***branch_predictor*** e também para o componente ***shift_reg_dec_4*** o *flag* ***branch_taken***, sinalizando se o desvio identificado deve ou não ser tomado, de acordo com os 2 *bits* da tabela e com o autômato de preditor de 2 *bits*.

A partir desta sinalização, o componente ***shift_reg_dec_4*** copia o IMAR, esvazia a fila de instruções para atualizá-la com as novas instruções a serem buscadas e configura o **flag_imar_copiado** para 1. De forma que o componente ***branch_predictor*** passa para um *multiplexador 4x1* o PC relativo à próxima instrução no caso do desvio ser tomado e configura o sinal ***sig_branch_predictor*** para 1.

Esse *mux 4x1* é responsável por determinar qual endereço deve ser carregado no IMAR. Seus sinais de controle são ***sig_branch_predictor*** e ***sig_branch*** sendo estes, respectivamente, o *bit* mais e menos significativo.

O ***sig_branch*** é gerado somente no quarto estágio, o de execução, quando se pode conferir se a previsão foi ou não correta. Ele será mais bem explicado em breve quando falarmos da correção em caso de erro de previsão.

As quatro entradas do *mux* são os endereços no caso da execução ser seqüencial, no caso de haver desvio tomado e no caso de necessidade de correção de previsão, de forma que, variando os *bits* de controle conseguimos determinar qual dos endereços deve ser carregado no IMAR.

No caso em que é feita a busca e o PC da instrução de desvio não é encontrado na tabela, duas situações podem ocorrer:

- Se a tabela ainda apresentar entradas disponíveis, o PC do desvio é guardado na próxima posição disponível da tabela e o desvio é configurado como não tomado.
- Se a tabela está cheia, então é escolhida aleatoriamente uma das entradas para ser reescrita com o PC da instrução de desvio em questão e esse desvio também é configurado inicialmente como não tomado.

No quarto estágio, a previsão é comparada com o que realmente aconteceu na execução através dos sinais ***sig_branch_predictor*** e ***sig_branch***. Caso esses dois sinais sejam diferentes, significa que foi feita uma previsão errada e o primeiro passo é atualizar os *bits* de história deste desvio na tabela.

Identificando o erro de previsão, temos que verificar se ocorreu o caso em que o preditor classifica o desvio como tomado e na verdade ele é não tomado. Em caso afirmativo, é necessário limpar o *pipeline* e corrigir o fluxo do programa. No caso em que o preditor classifica o desvio como não tomado e na verdade ele é tomado, não há necessidade de *flush* do *pipeline* pois o próprio processador fará esse trabalho.

4 Resultados Experimentais

Para testar a eficiência do preditor de desvios implementado, foram utilizados arquivos de memória de instruções de algoritmos de ordenação de memória. Os arquivos do tipo.*MIF* foram sintetizados a partir da ferramenta *SASHIMI* explicada no segundo capítulo desta dissertação.

Foram sintetizados algoritmos de *bubble sort*,*quick sort, insert sort* e *select sort* e, após executar cada um deles,obtivemos índices de acertos de aproximadamente 77,77%, 83,33%, 80% e 80%, respectivamente. Calculando a média aritmética do índice de acertos do preditor, para estes quatro algoritmos, obtivemos o valor de 80,28% de acertos e o cálculo da média harmônica nos dá um valor aproximado de 80,22%. O índice de acertos em cada programa e a média harmônica são mostrados no **Gráfico 1**.

De acordo com a literatura [6], este índice é considerado satisfatório por estar dentro da faixa de acertos dos preditores de desvios utilizados em processadores atualmente.

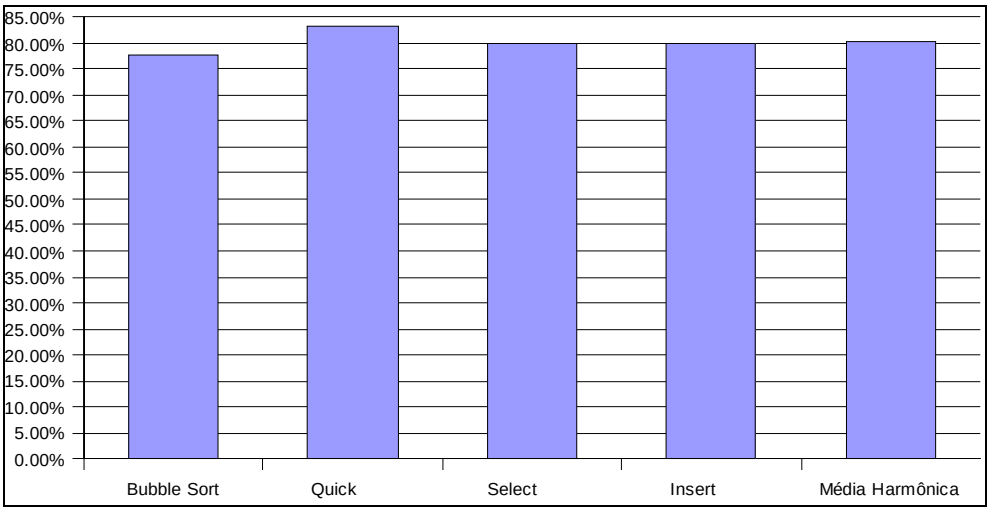

Gráfico 1: Índice de acertos por programa e média harmônica.

O ganho em relação ao número de ciclos para cada programa testado foi em torno de 1%, conforme mostrado na **Tabela 2**. O ganho em relação ao número de ciclos deve ser interpretado considerando que, na versão da arquitetura que apresenta preditor de desvios não ocorre inserção de bolhas quando um desvio condicional é encontrado, entretanto, no caso de necessidade de correção (*flush* do *pipeline*) pode haver perda de até dois ciclos de *clock*. Desta

forma, o baixo ganho referente ao tempo de execução pode estar relacionado ao custo da correção de erros de previsão.

Tabela 2: Resultado em relação ao número de ciclos.

Algoritmo	**Número de ciclos**	
	Arquitetura sem preditor de desvios	Arquitetura com preditor de desvios de 2 *bits*
Bubble Sort	2468	2442
Quick Sort	2304	2279
Insert Sort	1571	1554
Select Sort	1930	1909

Vale ressaltar que estes resultados foram obtidos sem variedade de tipos de programas já que todos os algoritmos utilizados tratam de ordenação de memória. Além disso, os programas utilizados apresentam reduzida entrada de dados (cada memória a ser ordenada apresentava somente 10 entradas) o que prejudica um pouco a avaliação de eficiência do componente proposto. Isso ocorreu devido ao fato do trabalho ter sido realizado somente com arquivos de memória de dados já sintetizados no *SASHIMI* uma vez que não houve completo acesso a arquivos fonte de programas que pudessem ser sintetizados para outros experimentos.

Outra observação importante é o fato do tamanho da tabela ter sido mantido constante. Aparentemente, não existem grandes variações de aproveitamento no que se refere a indice de acertos com o aumento do tamanho da tabela, mas também não se pode garantir que o tamanho escolhido é de fato o ideal.

Os erros de previsão podem ocorrer por vários fatores dentre estes, destacamos algumas situações:

- Quando o PC referente a um desvio é incluído na tabela, escolhemos configurar este desvio como não tomado inicialmente, uma vez que não conhecemos o histórico dele, essa atribuição inicial pode ser equivocada.
- Quando a tabela está completa, escolhemos aleatoriamente um desvio a ser retirado. Se, por acaso, o desvio retirado ocorrer com alta freqüência, mais uma vez, quando ele for novamente incluído na tabela, seus bits de historia inicialmente o

classificarão como desvio não tomado o que novamente pode prejudicar o índice de acertos do preditor.

- Também é válido considerar que mesmo com os bits de história, o comportamento do desvio pode variar fazendo com que alguns erros ocorram, afinal, é justamente quando ocorre um erro que o desvio muda de estado no autômato do preditor de desvios de 2 *bits* explicado no Capítulo 2.

5 Conclusão e Trabalhos Futuros

Este capítulo trata de sugestões a serem implementadas para aumentar ainda mais o rendimento do processador *FemtoJava* e apresenta a conclusão deste projeto.

5.1 Proposta de Implementação de um Preditor Neural

As subseções seguintes fazem uma breve apresentação do funcionamento da técnica de redes neurais sem peso e propõem a implementação de um preditor neural seguindo um modelo que utiliza esta técnica.

5.1.1 Técnica de redes neurais sem peso

Uma das vantagens das Redes Neurais Sem-Peso (RNSP) está na pequena distância semântica entre seus modelos e sistemas digitais convencionais. Além disso, seu mecanismo de aprendizado não se baseia na alteração de valores de pesos explícitos, mas sim na alteração do conteúdo de memórias distribuídas. Para isso, são utilizadas memórias RAM convencionais que podem ser escritas e lidas e operam de forma similar a neurônios de MaCulloc-Pitts.

5.1.2 Modelo *WISARD*

WISARD (**Wi**lkie, **S**tonham and **A**leksander's **R**ecognition **D**evice) [7] é uma máquina para aprendizado e reconhecimento de imagens que utiliza princípios neurais e foi o primeiro neurocomputador comercial a ser construído.

A máquina *WISARD* é capaz de efetuar uma classificação de uma imagem em preto e branco a cada 1/25 segundos sobre imagens de 512x512 pixels.

O modelo é composto basicamente por dois elementos: *uplas* e *discriminadores.* Um discriminador é composto por um conjunto de uplas e *WISARD* é um conjunto de discriminadores. A seguir é apresentado o funcionamento desses elementos.

- Uplas

  Cada upla é uma pequena memória geralmente com endereços de 2, 4 ou 8 *bits* e um *bit* de dado por endereço. Isso significa que podemos endereçar 2^2, 2^4 ou 2^8 posições de 1 *bit* por upla. Para cada treinamento realizado, a upla recebe como

entrada um subconjunto dos pixels da imagem (esse subconjunto é fixo para cada upla).

A upla lê esse conjunto de pixels e o mapeia para um endereço de sua memória. Como a upla está sendo treinada, nesta posição da memória é escrito '1' representando que aquela posição de pixels foi "vista". Assim, o conteúdo de cada posição da memória da upla representa o conhecimento adquirido a partir de um conjunto de pixels das imagens presentes no treinamento. O esquema do processo de treinamento de uma upla está representado na Figura 16.

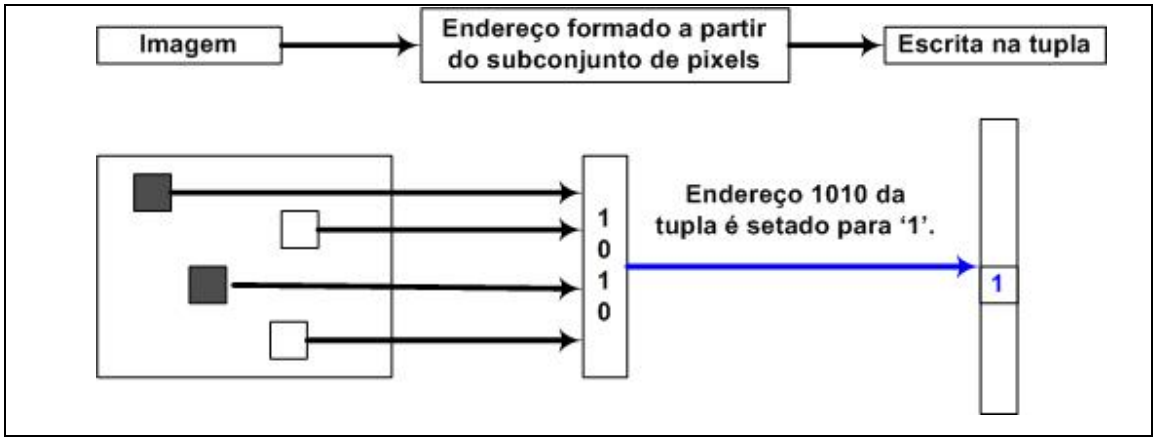

Figura 16: Treinamento de uma upla.

Durante a classificação, o mesmo mapeamento é realizado e a saída da upla é o conteúdo do endereço acessado. A upla, portanto, responde se aquela configuração de pixels é reconhecida ou não, *i.e.*, esteve ou não presente no treinamento.

- Discriminadores

Como cada upla enxerga apenas um pequeno pedaço da imagem, são necessárias várias uplas para cobrir uma imagem por completo. A esse conjunto de uplas utilizados para cobrir uma imagem por inteiro denominamos discriminador.

Um discriminador é responsável por sumarizar as respostas das suas uplas, sendo sua resposta o somatório das respostas das uplas.

WISARD é um modelo de multi-discriminadores, pois utiliza um discriminador para cada classe presente no treinamento. A resposta do modelo é baseada na classe cujo discriminador

teve a maior saída. Isso porque o discriminador que apresenta a maior saída é aquele cujas uplas “reconheceram” mais subconjuntos de imagens parecidas com a que está sendo classificada. A resposta de discriminadores pode ser vista na Figura 17.

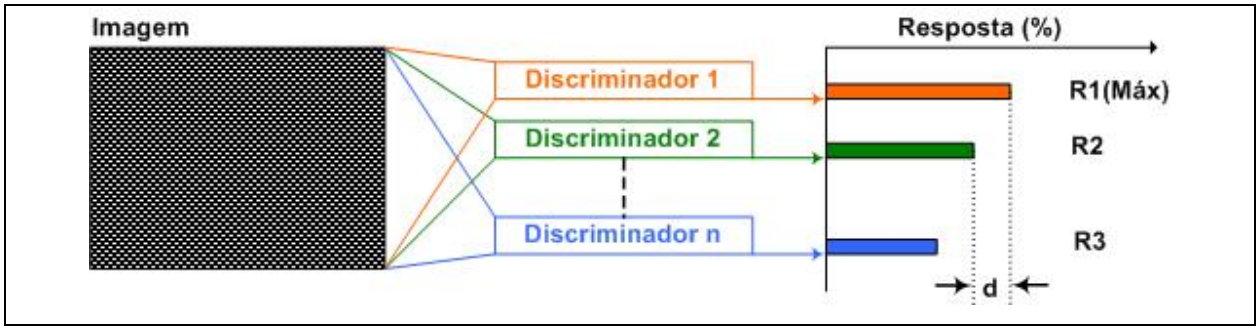

Figura 17: Resposta dos discriminadores onde *d* é a confiança da resposta.

Neste modelo, a parte de interpretação do resultado é muito importante, pois a resposta do *WISARD* não é ‘sim’ ou ‘não’. A resposta informa que a imagem se parece mais com uma determinada classe do que com outras ou ainda que a imagem pareça muito com duas ou mais classes conhecidas e dificilmente seja de uma outra classe em particular.

Se só forem classificadas imagens que estavam presentes no treinamento a resposta chega a 100% [9], pois uma vez que a informação está gravada na memória, o modelo nunca “esquece” com o passar do tempo e treinamentos posteriores não afetam a memória atual.

5.1.3 Preditor de desvios baseado no Modelo *WISARD*

Com base no modelo cujo funcionamento é explicado na subseção anterior, surgiu a idéia de implementação de um preditor neural que diminuiria bastante o custo de *hardware*.

A idéia inicial é utilizar uma lista circular de quatro posições no lugar da tabela de 2048 entradas utilizada no preditor de 2 *bits* implementado. Cada posição da lista guarda 8 *bits*: 7 se referem ao PC da instrução de desvio e 1 a previsão, ou seja, determina se o desvio deve ser previsto como tomado ou não tomado.

O modelo de preditor neural apresenta dois discriminadores, uma para a classe de desvios tomados e outro para a de não tomados.

O treinamento é realizado até que a confiança da resposta atinja um certo limite ainda a ser determinado e a resposta, assim como no *WISARD*, é dada em função do discriminador

cujas uplas “reconhecerem” melhor a entrada a ser classificada. O modelo desta idéia é apresentado na Figura 18.

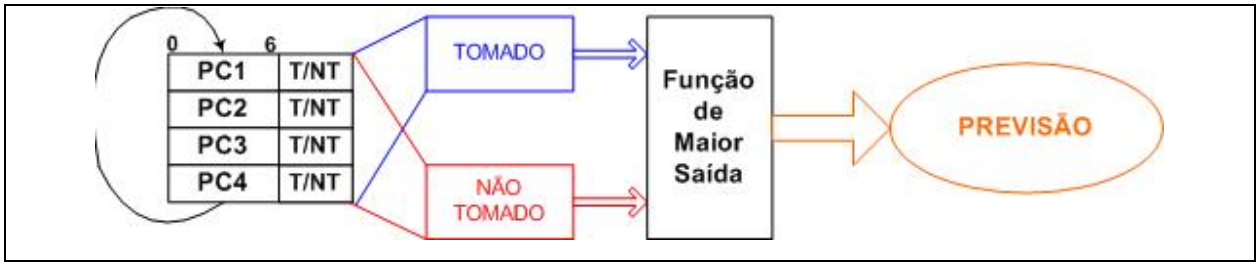

Figura 18: Modelo inicial para o preditor neural.

Como trabalho futuro fica a sugestão de implementação deste preditor utilizando 4, 8 e 16 *bits*/upla e determinando qual dessas implementações é a mais eficiente.

5.2 Conclusão

O modelo de preditor clássico de 2 *bits* foi implementado com sucesso utilizando a Linguagem C e mostrou-se eficiente, apresentando funcionamento com índice de acertos dentro da média esperada.

Os experimentos de teste de funcionamento do preditor foram realizados utilizando apenas algoritmos de ordenação de memória o que acaba por restringir um pouco a avaliação do componente proposto uma vez que não houve variação no tipo de programa utilizado.

A tabela necessária para a realização da previsão também foi mantida em tamanho fixo apresentando 2048 entradas, não havendo, desta forma, comprovação de que este é o tamanho ideal de tabela para o preditor de 2 *bits* implementado no processador *FemtoJava.*

Comprovou-se também a viabilidade da implementação deste preditor em *hardware*, apesar de, neste caso, haver um aumento do custo de desenvolvimento do processador uma vez que a implementação de entidades de armazenamento como *Branch History Table* e *Branch Buffer Table* introduzem no projeto um aumento de gastos no que se refere a utilização de *hardware*.

A idéia do preditor neural baseada no modelo *WISARD* aparentemente é implementável em *software* e as expectativas é que seu funcionamento alcance índices de acerto tão bons quanto aqueles alcançados pelo preditor de 2 *bits* implementado, utilizando muito menos *hardware* e conseqüentemente diminuindo os custos do projeto e a área ocupada em *chip*.

Referência Bibliográfica

[1] BECK, ANTONIO CARLOS SCHNEIDER (2004), Uso da técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em sistemas embarcados baseados em JAVA, Porto Alegre, RS

[2] ITO, SERGIO AKIRA (2000), Projeto de aplicações específicas com microcontroladores JAVA dedicados, Porto Alegre, RS

[3] COSTA, AMARILDO TEODORO DA (2001), Explorando dinamicamente o reuso de trace em nível de arquitetura de processador, Rio de Janeiro, RJ

[4] SILVA, BRUNO RODRIGUES (2006), Memorização e reuso dinâmico de traços em uma arquitetura de processador JAVA, Rio de Janeiro, RJ

[5] HENNESSY, PATTERSON. Organização e Projeto de Computadores – A interface *hardware*/*software*. 2ª Edição. Editora LTC.

[6] HENNESSY, PATTERSON. Arquitetura de Computadores – Uma abordagem quantitativa. 3ª Edição. Editora Campus. 2003

[7] DAL PIZZOL, GUILHERME (1999). Reduzindo o custo de desvios – Mecanismo de predição de desvios.

[8] ALEKSANDER, MORTON. An introduction to Neural Computing. Chapman and Hall. 1990

[9] SOARES, CÍNTIA M., *et alli*. (1998). Uma implementação em *software* do classificador *WISARD*

APÊNDICE A

Este apêndice apresenta o trecho do arquivo ***cacops_modulos.c*** referente à implementação da capacidade de predição de desvios no processador *FemtoJava*.

```
COMPONENTE("shift_reg_dec4") {
#ifdef TESTE_FEMTOJAVA_PIPELINE
#else
static int regs[9] = {0,0,0,0,0,0,0,0,0}, imar_temp=0, atualiza_imar = 0,k=0,pc=0, branch_id=0, b_offset=0, branches=0; // 9 registradores de 8 bits
#endif
int shift = 0,i,j,para,w,y,l,achei = 0;
if (clock_event == 1) {
if (controle_atual[2] == 1) shift = 4;
if (controle_atual[3] == 1) shift += 2;
if (controle_atual[4] == 1) shift += 1;
branch_id=0;
output[6]=0;
for(i=0;i<9 && para == 0;i++)
{
pc = imar_temp -(k-i);
// Informa o tamanho de cada instrucao
if ( (regs[i] >= 0xe0 && regs[i] <= 0xef) || regs[i] == 0x15 || regs[i] == 0x10 || regs[i] == 0x36 || regs[i] == 0x12 || regs[i] == 0x19)
{
i++;
}
 else if (regs[i] == 0x11 || regs[i] == 0xB2 || regs[i] == 0xB3 || regs[i] == 0xA7 || regs[i] == 0x84 ||(regs[i] >= 0xd0 && regs[i] <= 0xdf) ||regs[i] == 0x13 ||(regs[i] == 0xA6) || (regs[i]>= 0x99 && regs[i]<= 0xA4) )
{
switch (regs[i]){
case 0x9F: // icmpeq
case 0xA6: // acmpne
case 0xA0: // icmpne
case 0xA1: // icmplt
```

```
case 0xA2: // icmpge
case 0xA3: // icmpgt
case 0xA4: // icmple
case 0x99: // ifeq
case 0x9A: // ifne
case 0x9B: // iflt
case 0x9C: // ifge
case 0x9D: // ifgt
case 0x9E: // ifle
branch_id=1;
branches= branches+1;
b_offset=regs[i+1];
output[7]=regs[i+2];
para = 1;
break;
}
}
if (para == 0)
i=i+2;
}
if (input[2]==1)
{
for (w=i+1;w<9;w++)
regs[w]=0x00;
}
output[6]=1;
}
if (input[3]==1)
{
for(l=0;l<9 && achei == 0;l++)
{
if ( (regs[l] >= 0xe0 && regs[l] <= 0xef) || regs[l] == 0x15 || regs[l] ==
0x10 || regs[l] == 0x36 || regs[l] == 0x12 || regs[l] == 0x19)
{
l++;
else if (regs[l] == 0x11 || regs[l] == 0xB2 || regs[l] == 0xB3 || regs[l] ==
0xA7 || regs[l] == 0x84 ||(regs[l] >= 0xd0 && regs[l] <= 0xdf) ||regs[l] ==
0x13 ||(regs[l] == 0xA6) || (regs[l]>= 0x99 && regs[l]<= 0xA4) )
```

```
{
switch (regs[l]){
case 0x9F: // icmpeq
case 0xA6: // acmpne
case 0xA0: // icmpne
case 0xA1: // icmplt
case 0xA2: // icmpge
case 0xA3: // icmpgt
case 0xA4: // icmple
case 0x99: // ifeq
case 0x9A: // ifne
case 0x9B: // iflt
case 0x9C: // ifge
case 0x9D: // ifgt
case 0x9E: // ifle
achei = 1;
for (y=l+1;y<9;y++)
{
regs[y]=0x00;
}
break;
}
}
if (achei == 0)
l=l+2;
}
}
}
for (j=0;j<shift;j++)
{
for (i=8;i>0;i--)
{
regs[i] = regs[i-1];
}
regs[0] = 0;
}
if (controle_atual[5])// fetch ligado, grava valor vindo
{
```

```
if (atualiza_imar == 1)
{
imar_temp = input_atual[1];
atualiza_imar = 0;
}
atualiza_imar = 1;
k=0;
if(controle_atual[8]== 1) k = 4;
if (controle_atual[6] == 1) k += 2;
if (controle_atual[7]== 1) k+= 1;
k=k+3;
if (controle_atual[8] == 0) {
if (controle_atual[6] == 0  && controle_atual[7] == 0) // escreve reg[0]
regs[0] = (input_atual[0] >> 24) & 0xff;
if (controle_atual[6] == 0) {
if (controle_atual[7] == 0 ) regs[1] = (input_atual[0] >> 16) & 0xff;
else regs[1] = (input_atual[0] >> 24) & 0xff;
}
if  (controle_atual[6]  ==  0  &&  controle_atual[7]  ==  0)  regs[2]  =
(input_atual[0] >> 8 ) & 0xff;
else  if  (controle_atual[6]  ==  0  &&  controle_atual[7]  ==  1)  regs[2]  =
(input_atual[0] >> 16 ) & 0xff;
else  if  (controle_atual[6]  ==  1  &&  controle_atual[7]  ==  0)  regs[2]  =
(input_atual[0] >> 24 ) & 0xff;
if  (controle_atual[6]  ==  0  &&  controle_atual[7]  ==  0)  regs[3]  =
(input_atual[0] ) & 0xff;
else  if  (controle_atual[6]  ==  0  &&  controle_atual[7]  ==  1)  regs[3]  =
(input_atual[0] >> 8 ) & 0xff;
else  if  (controle_atual[6]  ==  1  &&  controle_atual[7]  ==  0)  regs[3]  =
(input_atual[0] >> 16) & 0xff;
else  if  (controle_atual[6]  ==  1  &&  controle_atual[7]  ==  1)  regs[3]  =
(input_atual[0] >> 24) & 0xff;
if  (controle_atual[6]  ==  0  &&  controle_atual[7]  ==  1)  regs[4]  =
(input_atual[0]) & 0xff;
else  if  (controle_atual[6]  ==  1  &&  controle_atual[7]  ==  0)  regs[4]  =
(input_atual[0] >> 8) & 0xff;
else  if  (controle_atual[6]  ==  1  &&  controle_atual[7]  ==  1)  regs[4]  =
(input_atual[0] >> 16) & 0xff;
```

```
}
else regs[4] = (input_atual[0] >> 24) & 0xff;
if (controle_atual[8] == 0){
if (controle_atual[6] == 1 && controle_atual[7] == 0) regs[5]                =
(input_atual[0]) & 0xff;
else  if  (controle_atual[6]  ==  1  &&  controle_atual[7]  ==  1)  regs[5]  =
(input_atual[0]  >> 8) & 0xff;
}
else regs[5] = (input_atual[0] >> 16) & 0xff;
if (controle_atual[6] == 1 && controle_atual[7] == 1 && controle_atual[8] == 0
) regs[6] = (input_atual[0]) & 0xff;
else if (controle_atual[8] == 1 )  regs[6] = (input_atual[0] >> 8) & 0xff;
if (controle_atual[8])  regs[7] = (input_atual[0]) & 0xff;
}
if (controle_atual[0] == 0) for(i=0;i<9;i++) regs[i] = 0xff;
if (controle_atual[1] == 0) for(i=0;i<9;i++) regs[i] = 0x00;
#ifdef IMPRIME
if (test_mode == 0) {
for (i=0;i<9;i++) printf("%.2x ",regs[i] & 0xff); printf("\n");
}
#endif
output[0] = regs[8] & 0xff;
output[1] = regs[7] & 0xff;
output[2] = regs[6] & 0xff;
output[3]=pc;
output[4]=branch_id;
output[5]=b_offset;
output[6]= 1;
}

COMPONENTE("table"){
static struct T_Tabela{
unsigned char PC;
unsigned char next_PC;
unsigned char bits;
}tabela[MAX];
static int  indice=0, indice_branch=0;
int indice_escolhido=0, i=0;
```

```
int stime;
long ltime;
if (clock_event == 1)
{
if (input[0]==1)
{
do
{
if (tabela[i].PC == input[1])
{
if (tabela[i].bits == 0x02 || tabela[i].bits == 0x03)
{
output[0]=tabela[i].next_PC;
output[1]=1;
output[2]=i;
}
}
}     while ((tabela[i].PC!= input[1])&&(i<indice));
 if ((i==indice)&&(tabela[i].PC!= input[1]))
{
if (indice<MAX)
{
indice++;
tabela[indice].PC=input[1];   //escrever na tabela
tabela[indice].next_PC=input[1]+ input[2];
tabela[indice].bits=0x01; // not taken
output[2]=indice;
output[1]=0;
}
else //escolher um desvio para ser substituido pelo desvio atual
{
ltime = time(NULL);
stime = (unsigned) ltime/2;
srand (stime);
indice_escolhido= rand()%MAX;
tabela[indice_escolhido].PC=input[1];     //escrever na tabela
tabela[indice_escolhido].next_PC=input[1]+ input[2];
tabela[indice_escolhido].bits=0x01;
```

```
output[2]= indice_escolhido;
output[1]=0;
}
}
}
if(input[5]==1)
{
tabela[input[3]].bits=input[4];
}
}
}

COMPONENTE("branch_predictor") {
if ((clock_event == 1)&&(input[1]==1)&&(input[2]==1))
{
output[0]=input[0];
output[1]=1;
}
}

COMPONENTE ("flag") {
if (clock_event == 1)
{
if (input[0]== 1)
output[0]= input[0];
}
}

COMPONENTE ("branch_predictor_inspector"){
static int mistakes=0;
if (clock_event == 1)
{
if (input[0] == 1)
{
if (input[1]!= input[2])
{
mistakes = mistakes+1;
output[0]= input[3];
```

```
output[2]=1;
switch (input[4]){
case 0x00:
output[1]=0x01;
break;
case 0x01:
output[1]=0x03;
break;
case 0x02:
output[1]=0x00;
break;
case 0x03:
output[1]=0x02;
break;
}

if ((input[1]==0)&&(input[2]==1))
{//correção do pipeline
output[3]=1;
output[4]=1;
output[5]=1;
}
}
}
}
}
```

APÊNDICE B

Este apêndice apresenta o arquivo de descrição de arquitetura do *FemtoJava pipeline* com preditor de desvios.

```
// VERSAO 32 BITS
// Pipeline versao 12:
// Tira os sinais de controle que não são usados nos estágios futuros,
economizando alguns bits de registradores de estágio e ficando mais elegante
// Com a fila de instrucoes arruamada

////////////////////////////////////////////////////////////////////////
//
// ESTAGIO 1 - BUSCA DE INSTRUCOES
////////////////////////////////////////////////////////////////////////
//

sig_fetch,sig_branch->or_2p 1 ()-> sig_enable_imar
sig_imar_in->reg imar (0,'1','1':sig_enable_imar)-> sig_imar_out
nada->      gera4 1 ()-> value_4
sig_imar_out,value_4->  adder add_imar () -> sig_imar_in_t
sig_imar_in_t,pc_in,pc_branch,new_imar -> mux4_1 up_imar
(0:sig_branch_predictor;0:sig_branch)-> sig_imar_in
sig_imar_out-> conv_addr 1 ()-> rom_address, rom_resol_1a, rom_resol_0a
rom_resol_1a-> dffa a ('1','1','1':nada) -> rom_resol_1b
rom_resol_1b-> dffa b ('1','1':nada;0:sig_not_branch) -> rom_resol_1c
rom_resol_0a-> dffa d ('1','1','1':nada) -> rom_resol_0b
rom_resol_0b-> dffa e ('1','1':nada;0:sig_not_branch) -> rom_resol_0c
sig_branch-> dffa branch1 ('1','1','1':nada) -> sig_branch_1
sig_branch_1-> dffa branch2 ('1','1','1':nada) -> sig_branch_2
rom_resol_1b,rom_resol_0b ->and_2p or_resol ()-> shift_resol_and
shift_resol_and, sig_branch_2 -> and_2p 1 ()-> shift_resol
shift_resol-> dffa resol1 ('1','1','1':nada) -> shift_resol_1
rom_address->rom32_v12 1 () -> instr_bus
branch_id, pc_b, branchoffset, indice_to_modify, new_state, error  -> table 1
() -> new_pc, branch_taken, indice_branch, branch_bits
```

```
new_pc, branch_taken, flag_imar_copiado  -> branch_predictor 1 () ->
pc_branch, sig_branch_predictor

// REGISTRADOR DE ESTAGIO
instr_bus, sig_imar_out, branch_taken, flush -> shift_reg_dec4 Reg_Dec
('1',0:sig_not_branch;2,1,0:opcode_length;2,1,0:queue_mux_out;0:shift_resol) -
> opcode,immed1_fetch,immed2_fetch,pc_b,branch_id,branchoffset,
flag_imar_copiado, new_imar

// Notifica qdo está no estágio
branch_id->flag 1 ()-> estagio1

////////////////////////////////////////////////////////////////////////////
//
// ESTAGIO 2 - DECODIFICACAO
////////////////////////////////////////////////////////////////////////////
//

// Primeira parte - Avisa para a BUSCA DE INSTRUCOES quando buscar instrucoes
e quanto deve andar a sua fila
opcode_length_t->or_no_2_bit2 1 (4,3:queue_fsm_out)-> opcode_length_t2
opcode_length_t2,nada-> mux2_1 data_dep1 (0:sig_dep)-> opcode_length
queue_mux_out->sh_reg_3 queue_count
(0,'1':count_enable;0:sig_not_branch;0:sig_fetch) -> queue_count_out_0

// No caso de um salto em que apenas um opcode eh aproveitavel eh preciso
ajustar os primeiros passos
queue_count_out_0, value_4-> mux2_1 mux_pos_branch (0:shift_resol_1)->
queue_count_out
queue_count_out,opcode_length->adder add_queue_count ()-> adder_queue_out

// vai ser reset ou set, coisa parecida. Servira para indicar o estado inicial
nada->queue_fsm2 queue_fsm
(0:sig_branch;0:rom_resol_1a;0:rom_resol_0a;0:rom_resol_1b;0:rom_resol_0b)->
queue_fsm_out,op_length_stop,queue_fsm_bit
adder_queue_out,queue_fsm_out->mux2_1 mux_queue (4:queue_fsm_out)->
queue_mux_out
queue_mux_out->get_2 1 ()-> queue_count_bit
```

```
queue_fsm_bit->   not_1b 1 ()-> count_enable
queue_fsm_bit,queue_count_bit->or_2p 1 ()-> sig_fetch

// Segunda parte - Decodifica o OPCODE e coloca os bits em um registrador
// mudado para _32 bits, colocado LDC e LDC_W - Dep. de dados, microop e
tamanho da instrução
opcode,sig_pc_out->decoder2 1 (0:sig_branch)-> decoder2_out_t,opcode_length_t
decoder2_out_t,nada->mux2_1 data_dep2 (0:sig_dep)-> decoder2_out

// O funcionamento do controle e geracao de constantes neste mux eh o mesmo do
iconst_mux do femtojava anterior
nada->gen_cons 1 (4,3,2:decoder2_out)-> mux_immed1_in1
nada ->iconsctrl 1 (4,3,2:decoder2_out)-> control_iconst
mux_immed1_in1,immed1_fetch-> mux2_1 mux_immed1 (0:control_iconst)->
mux_immed1_out

// Verificacao de dependecia de dados e ja serve para o forwarding
opcode->data_dep_ver_v9 0 (0:sig_branch)->
data_dep_st_w,data_dep_var_w,dd_rsp,dd_rvar,dd_rsp_2op,dd_wsp_force2
sig_dep_neg,data_dep_st_w->and_2p 0 ()-> dd_wsp
sig_dep_neg,data_dep_var_w->and_2p 0 ()-> dd_wvar

// Deteccao de NOP
data_dep_st_w,data_dep_var_w,dd_rsp,dd_rvar -> or_4p 1 ()-> nop_signal
nop_signal->dffa ff_nop ('1','1','1':nada)      -> nop_signal2
nop_signal2->dffa ff_nop2 ('1','1','1':nada)-> nop_signal3

// Leitura na pilha
dd_rsp->dffa dd_rsp1 ('1','1','1':nada)-> dd_rsp2
dd_rsp2->dffa dd_rsp2 ('1','1','1':nada)-> dd_rsp3
dd_rsp2,dd_rsp3->or_2p 1 ()-> dd_rsp2_3
dd_rsp2_3->not_1b 1 ()-> dd_rsp2_3_not

// Escrita na pilha
dd_wsp->dffa dd_sp1 ('1','1','1':nada)-> dd_wsp2_f
sig_forward_sp_not,dd_wsp2_f->and_2p 1 ()-> dd_wsp2
dd_wsp2->dffa dd_sp2 ('1','1','1':nada)-> dd_wsp3_f_a
sig_forward_1B_not,dd_wsp3_f_a      ->and_2p 1 ()-> dd_wsp3_f_b
```

```
sig_forward_2op_not,dd_wsp3_f_b     ->and_2p 1 ()-> dd_wsp3

// Escrita em VARS
dd_wvar     ->dffa dd_var1 ('1','1','1':nada)-> dd_wvar2
dd_wvar2->dffa dd_var2 ('1','1','1':nada)-> dd_wvar3

// Mesmo com forwarding, eh obrigatoria a escrita na pilha na instrucao
anterior (exemplo: DUP)
// Por enquanto soh funciona para o PROXIMO operando
dd_wsp_force2->   dffa dd_sp_force ('1','1','1':nada)-> dd_wsp_force3
dd_wsp_force2->   not_1b 1 ()-> dd_wsp_force2_not

// Dependencia devido a escrita e dois ciclos depois leitura (3o estagio vs 5o
estagio)
dd_rsp,dd_rvar->or_2p 1 ()-> dd_rspvar
dd_wsp3,dd_wvar3,dd_wsp_force3      ->or_3p 1 ()-> dd_wspvar
dd_rspvar,dd_wspvar->and_2p 1 ()-> sig_dep3
sig_dep3->not_1b 1 ()-> sig_dep3_not
sig_dep3-> pass 1 ()-> sig_dep

// Forwarding!!!
// Quando ha escrita na pilha, depois uma leitura, forward
dd_wsp2_f,dd_rsp->and_2p 1 ()-> sig_forward_t1_sp

// Quando ha escrita em VARS, depois uma leitura, e os VARS sao os mesmos,
forward
mux_immed1_out,immed1_decode->compare_8b 1 ()-> same_var_temp

// Serve para comparar quando ha escrita em VARS e leitura na PILHA, quando
VARS + INDICE escrita = SP + indice leitura
mux_immed1_out,immed1_decode,sp_out,vars_out    -> compare_sp_vars 1 () ->
same_var_sp
same_var_sp,same_var_temp-> and_2p 1 ()-> same_var
same_var->not_1b 1 ()-> not_same_var
dd_wvar2,dd_rvar->and_2p 1 ()-> sig_forward_t1_vars_t
sig_forward_t1_vars_t,not_same_var ->and_2p 1 ()-> sig_forward_t1_vars
sig_forward_t1_vars->not_1b 1 ()-> sig_forward_vars_not
```

```
// Se houver dep de dados 3, cancela forward
sig_forward_t1_sp,sig_dep3_not->and_2p 1 ()-> sig_forward_t2_sp
sig_forward_t1_vars,sig_dep3_not ->and_2p 1 ()-> sig_forward_t2_vars
sig_forward_t2_sp->not_1b 1 ()-> sig_forward_sp_not
sig_forward_t2_sp,sig_forward_t2_vars -> or_2p 1 ()-> sig_forward

// Se alguma instrucao anterior a que esta recebendo os dados do forward ler
da pilha, alem de escrever
// o segundo operando vem da pilha
// Se ha um forward, e a instrucao atual le dois operandos da pilha:
dd_rsp_2op,sig_forward_t1_sp  ->    and_2p 1 ()
      -> sig_forward_2op_t

// Se a instrucao anterior ler algum operando da pilha, nao pode dar o
forwarding.
// O segundo operando da instrucao atual vira da pilha
dd_rsp2->not_1b 1 ()-> dd_rsp2_not
dd_rsp2_not,sig_forward_2op_t->and_2p 1 ()-> sig_forward_2op_t2

// Finalmente, se passar por tudo isso MAS a instrucao anterior anterior nao
tiver sido executada (foi executada antes)
// Devido a alguma dependencia de dados, pega-se o segundo operando da pilha
sig_forward_2op_t2,sig_dep_c_not ->and_2p 1 ()-> sig_forward_2op_t3
sig_forward_2op_t3->not_1b 1 ()-> sig_forward_2op_not

// Ou ainda devido a um NOP ao inves de uma dependencia de dados!
sig_forward_2op_t3,nop_signal3->and_2p 1 ()-> sig_forward_2op_t4

// Se houver dep de dados 3, cancela forward 2
sig_forward_2op_t4,sig_dep3_not->and_2p 1 ()-> sig_forward_2op
// Se houve uma dependencia de dados no ciclo anterior,
// O primeiro operando vem de result_exec (5o estagio), devido ao bubble
colocado no meio
// Se forward == 1 e sig_dep == 1, no proximo ciclo, forward_1B = 1
sig_dep->dffa ff_dep ('1','1','1':nada)-> sig_dep_b
sig_dep_b->dffa ff_dep_b ('1','1','1':nada)-> sig_dep_c
sig_dep_c->not_1b 1 ()-> sig_dep_c_not
sig_forward_t1_sp->dffa ff_forw ('1','1','1':nada)-> sig_forward_b
```

```
sig_dep_b,sig_forward_b->and_2p 1 ()-> sig_forward_1B
sig_forward_1B->not_1b 1 ()-> sig_forward_1B_not

// Fim de forward!

// REGISTRADORES DE ESTAGIO

decoder2_out->mascara1 1 ()-> microop_decode_temp
microop_decode_temp->reg32 ID1_25bits ('1','1',0:sig_not_branch)->
microop_decode_control
mux_immed1_out->reg8 ID2 ('1','1',0:sig_not_branch)-> immed1_decode
immed1_fetch,immed2_fetch->concatena 1 ()-> immed1_2
immed1_2->reg ID3 ('1','1',0:sig_not_branch)-> immed2_decode
sig_dep->not_1b 1 ()-> sig_dep_neg
opcode_length,nada->mux2_1 opcode_l (0:op_length_stop)-> opcode_length_v
opcode_length_v->reg ID4_2bits ('1','1',0:sig_not_branch)-> op_length_decode

// Forwarding
sig_forward->dffa forward_1 ('1','1','1':nada)-> forward1_decode
sig_forward_1B->dffa forward_1B ('1','1','1':nada)-> forward1B_decode
sig_forward_2op->dffa forward_2op ('1','1','1':nada)-> forward2op_decode
// Notifica qdo está no estágio
estagio1->flag 2 ()-> estagio2

////////////////////////////////////////////////////////////////////////////
//
// ESTAGIO 3 - BUSCA DE OPERANDOS
////////////////////////////////////////////////////////////////////////////
//
microop_decode_control->mascara2 1 ()-> microop_op_fetch_temp
microop_op_fetch_temp-> reg32 OF1_17bits ('1','1',0:sig_not_branch)->
microop_op_fetch_control
op_length_decode->reg OF5_2bits ('1','1',0:sig_not_branch)-> op_length_fetch
immed1_decode->sig_ext_32 1 ()-> immed1_decode_se

//forwarding
reg_bank_d1_f,exec_result,result_exec-> mux4_1 forwd1
(0:forward1B_decode;0:forward1_decode) -> reg_bank_data1
```

```
reg_bank_d2_f,result_exec->mux2_1 forwd2 (0:forward2op_decode)->
reg_bank_data2
immed1_decode_se,reg_bank_data1->mux2_1 mux_op1 (0:microop_decode_control)->
mux_op1_out
mux_op1_out->     reg32 OF2 ('1','1',0:sig_not_branch)      -> op1_opfetch
immed2_decode,reg_bank_data2->      mux2_1 mux_op2
(13:microop_decode_control)-> mux_op2_out
mux_op2_out->     reg32 OF3 ('1','1',0:sig_not_branch)      -> op2_opfetch
vars_out,immed1_decode_se->adder var ()-> vars_out_t
vars_out_t,immed2_decode->mux2_1 var_nada (14:microop_decode_control)->
vars_out_rel
sp_out,vars_out_rel-> mux2_1 mux_sv (1:microop_decode_control)-> read_address
nada->      gera1 1 ()-> value_1
read_address,value_1->  subtr 1 ()-> reg_bank_addr2
vars_out_rel->reg OF4 ('1','1',0:sig_not_branch)-> vars_write_fetch

////////////// Atualizacao do SP
microop_decode_control->get_9 1 ()-> sp_control_t

// Evita que quando houver um branch o SP seja atualizado erroneamente pela 3a
instrucao depois da instrucao de branch
sp_control_t,sig_not_branch->and_2p 1 ()-> sp_control
write_address_sp->reg SP (0,'1','1':sp_control)-> sp_out

//// ----> invokestatic
sv_value,result_exec->mux2_1 mux_w_sp (21:microop_decode_control)-> upgrade_sp
sp_out,upgrade_sp->addsub ad2 (11:microop_decode_control)   -> sp_relative
sp_relative,vars_out->mux2_1 w_sp (23:microop_decode_control)->
write_address_sp
nada->gera_sv_value 1 (12:microop_decode_control)     -> sv_value_t
sv_value_t->and_b1_b2 0 (9:microop_decode_control)-> sv_value
write_address_sp->reg OF6 ('1','1','1':nada)-> write_SP_fetch

//Forwarding - FFs de estagio
sig_forward_t2_sp,dd_wsp_force2_not ->and_2p 1 ()     -> forward2_fetch_t
forward2_fetch_t->dffa forward_b2 ('1','1','1':nada)-> forward2_fetch

// Avisa qdo chegou ao estágio
```

```
estagio2->flag 3 ()-> estagio3

////////////////////////////////////////////////////////////////////////////
//
// ESTAGIO 4 - EXECUCAO
////////////////////////////////////////////////////////////////////////////
//
microop_op_fetch_control      ->mascara3 1 ()-> microop_exec_temp
microop_exec_temp->reg8 EX1_4bits ('1','1','1':nada)  -> microop_exec_control
vars_write_fetch->reg EX3 ('1','1','1':nada)-> vars_write_exec
write_SP_fetch->  reg EX4 ('1','1','1':nada)-> write_SP_exec

// Unidades funcionais
op1_opfetch,op2_opfetch->arit_log_unit_32 1 (5,4,3:microop_op_fetch_control)->
r_1
op1_opfetch,op2_opfetch->multiplier_32 1 ()-> r_2
op2_opfetch->get_bits 1 (3:microop_op_fetch_control)-> r_3
op1_opfetch,op2_opfetch->branch_32 1 (8,5,4,3:microop_op_fetch_control)-> r_4
op2_opfetch,op1_opfetch->shifter_32 1 (4,3:microop_op_fetch_control)-> r_5
op2_opfetch,op1_opfetch->divisor_32 1 (3:microop_op_fetch_control)-> r_6
op1_opfetch,op2_opfetch->ld_st 1 (5,4,3:microop_op_fetch_control)->
address_ldst
microop_op_fetch_control      ->and_0_1_2_3 1 ()-> rw_ram

//// ----> invokestatic
address_ldst,sig_frm_out-> mux2_1 invoxe_mux (9:microop_op_fetch_control)->
ram_address
op1_opfetch,sig_pc_out,sp_out,vars_out-> mux4_1 st_data
(11,10:microop_op_fetch_control)-> ram_data
ram_data,ram_address-> ram32 1 (0:rw_ram)-> r_7

// Selecao do resultado
op1_opfetch,r_1,r_2,r_3,nada,r_5,r_6,r_7 -> mux8_1 exec_res
(2,1,0:microop_op_fetch_control) -> exec_result
exec_result->reg32 EX2 ('1','1','1':nada)-> result_exec

// Atualização do registrador PC
```

```
r_4,microop_op_fetch_control, sig_branch_t      -> and_branch_v10 1 ()->
sig_branch
sig_branch->not_1b not_branch ()-> sig_not_branch
nada->      gera2 1 ()-> value_2
vars_write_fetch,value_2->adder pc_adder2 ()-> jmp_end_branch
sig_branch->and_not return_verify (17:microop_op_fetch_control)-> sig_pc_ctrl
op_length_fetch,jmp_end_branch->mux2_1 new_end_pc (0:sig_pc_ctrl)-> upgrade_pc
sig_pc_out,upgrade_pc->adder pc_adder ()-> jmp_end

//// ----> invokestatic
vars_write_fetch,value_1->subtr 3 ()-> invoke_end
jmp_end,invoke_end,exec_result,nada->mux4_1 jmp_mux
(16,12:microop_op_fetch_control)-> pc_in
pc_in->reg PC ('1','1','1':nada)-> sig_pc_out

// Forwarding
forward2_fetch,sig_forward_2op,sig_forward_1B -> or_3p 1 ()-> forward2_exec_t
forward2_exec_t->dffa forward_c2 ('1','1','1':nada)-> forward2_exec

//////////////////
// Hardware Adicional para invokestatic, return e ireturn
//////////////////
sig_frm_in->reg FRM (9:microop_op_fetch_control;0,'1':set)-> sig_frm_out
sig_frm_out,value_1->addsub ad2 (13:microop_op_fetch_control)-> sig_frm_in
value_1->pass set ()-> set

// Avisa qdo chegou ao estágio
estagio3->flag 4 ()-> estagio4

// Checa a previsao, calcula a porcentagem de erro e atualiza a tabela de
historia de desvios
estagio4, sig_branch, branch_taken, indice_branch, branch_bits->
branch_predictor_inspector ()> indice_to_modify, new_state, error,
sig_branch_t, flush, sig_branch_predictor
```

```
////////////////////////////////////////////////////////////////////////////
//
// ESTAGIO 5 - WRITE BACK
////////////////////////////////////////////////////////////////////////////
//

//// ----> invokestatic
sp_out,result_exec->subtr 2 ()-> write_vars_t
write_vars_t,result_exec->    mux2_1 w_vars (3:microop_exec_control)->
write_vars
write_vars->reg VARS (2,'1','1':microop_exec_control)-> vars_out
write_SP_exec,vars_write_exec ->    mux2_1 1 (1:microop_exec_control)->
write_address

//////////////////
// Banco de registradores para pilha e variáveis locais
//////////////////

// forwarding
microop_exec_control->  get_0 1 ()-> sig_reg_bank_t
forward2_exec->   not_1b 1 ()-> forward2_exec_n
sig_reg_bank_t,forward2_exec_n->    and_2p 1 ()-> sig_reg_bank
read_address,write_address->mux2_1 rw_addr (0:sig_reg_bank)-> reg_bank_addr1
reg_bank_addr1,reg_bank_addr2,result_exec->register_bank_v9_32 1
(0:sig_reg_bank)-> reg_bank_d1_f,reg_bank_d2_f
```

ANEXO I

Este anexo apresenta um exemplo de arquivo de memória de instruções (***bubble.mif***) sintetizado no *SASHIMI* para a arquitetura utilizada no projeto.

```
-- Memory Initialization File - generated by SASHIMI 0.8 Lite

WIDTH = 8;
DEPTH = 256;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
    0    : 002b00b8; -- SASHIMI.int0Method.()V.0  --
--                          -- invokestatic
    1    : 0000b100; -- nop                       -- nop
-- return                   --
    2    : 00000000; -- SASHIMI.tf0Method.()V.0   -- nop
-- nop                      -- nop
    3    : 0000b100; -- nop                       -- nop
-- return                   --
    4    : 00000000; -- SASHIMI.int1Method.()V.0  -- nop
-- nop                      -- nop
    5    : 0000b100; -- nop                       -- nop
-- return                   --
    6    : 00000000; -- SASHIMI.tf1Method.()V.0   -- nop
-- nop                      -- nop
    7    : 0000b100; -- nop                       -- nop
-- return                   --
    8    : 00000000; -- SASHIMI.spiMethod.()V.0   -- nop
-- nop                      -- nop
    9    : 0000b100; -- nop                       -- nop
-- return                   --
    a    : 00000000; -- Sort.initSystem.()V.0     -- nop
-- nop                      -- nop
    b    : 1300b200; --                           --
-- getstatic                --
    c    : 573500b8; -- pop                       --
--                          -- invokestatic
    d    : 030101b1; -- iconst_0                  --
-- Sort.bubbleSort.(I)I.2   -- return
    e    : b21400b3; -- getstatic                 --
--                          -- putstatic
    f    : 041a1400; -- iconst_1                  -- iload_0
--                          --
    10   : 5b00a264; --                           --
-- if_icmpge                -- isub
    11   : b364041a; -- putstatic                 -- isub
-- iconst_1                 -- iload_0
    12   : 00b21500; --                           -- getstatic
--                          --
```

```
    13   : 1400b215; --                          --
-- getstatic                 --
    14   : b24100a4; -- getstatic                --
--                           -- if_icmple
    15   : 00b21600; --                          -- getstatic
--                           --
    16   : 00b22e15; --                          -- getstatic
-- iaload                    --
    17   : 1500b216; --                          --
-- getstatic                 --
    18   : a22e6404; -- if_icmpge                -- iaload
-- isub                      -- iconst_1
    19   : 00b22300; --                          -- getstatic
--                           --
    1a   : 1500b216; --                          --
-- getstatic                 --
    1b   : 00b23c2e; --                          -- getstatic
-- istore_1                  -- iaload
    1c   : 1500b216; --                          --
-- getstatic                 --
    1d   : b21600b2; -- getstatic                --
--                           -- getstatic
    1e   : 64041500; -- isub                     -- iconst_1
--                           --
    1f   : 00b24f2e; --                          -- getstatic
-- iastore                   -- iaload
    20   : 1500b216; --                          --
-- getstatic                 --
    21   : 4f1b6404; -- iastore                  -- iload_1
-- isub                      -- iconst_1
    22   : 041500b2; -- iconst_1                 --
--                           -- getstatic
    23   : 1500b364; --                          --
-- putstatic                 -- isub
    24   : b2b8ffa7; -- getstatic                --
--                           -- goto
    25   : 60041400; -- iadd                     -- iconst_1
--                           --
    26   : a71400b3; -- goto                     --
--                           -- putstatic
    27   : b3039eff; -- putstatic                -- iconst_0
--                           --
    28   : 00b21400; --                          -- getstatic
--                           --
    29   : 00a21a14; --                          -- if_icmpge
-- iload_0                   --
    2a   : 1600b221; --                          --
-- getstatic                 --
    2b   : 2e1400b2; -- iaload                   --
--                           -- getstatic
    2c   : b22300b2; -- getstatic                --
--                           -- getstatic
    2d   : 9f2e1400; -- if_icmpeq                -- iaload
--                           --
    2e   : b3030500; -- putstatic                -- iconst_0
--                           --
```

```
    2f   : 00b23000; --                          -- getstatic
--                          --
    30   : b3600414; -- putstatic                -- iadd
-- iconst_1                 --
    31   : ffa71400; --                          -- goto
--                          --
    32   : 3000b2da; --                          --
-- getstatic                --
    33   : 0500a004; --                          --
-- if_icmpne                -- iconst_1
    34   : 3100b304; --                          --
-- putstatic                -- iconst_1
    35   : 0000ac03; -- no code                  -- no code
-- ireturn                  -- iconst_0
    36   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    37   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    38   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    39   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    3a   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    3b   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    3c   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    3d   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    3e   : 00000000; -- no code                  -- no code
-- no code                  -- no code
    3f   : 00000000; -- no code                  -- no code
-- no code                  -- no code
END;
```