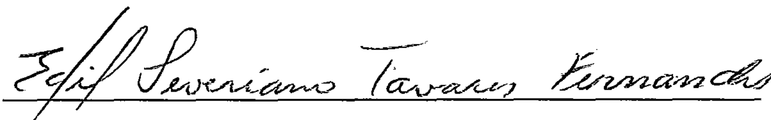
Arquiteturas Super Escalares: Efeito de Alguns Parâmetros sobre o Desempenho

Eliseu Monteiro Chaves Filho

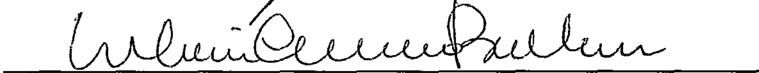
Tese submetida ao corpo docente da Coordenação dos Programas de Pós-graduação em Engenharia da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências em Engenharia de Sistemas e Computação.

Aprovada por:

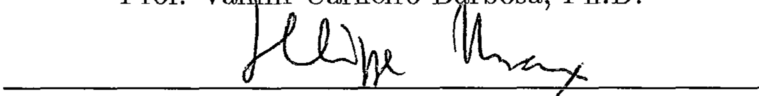


Prof. Edil S. T. Fernandes, Ph.D.
(Presidente)

Prof. Cláudio Luis de Amorim, Ph.D.

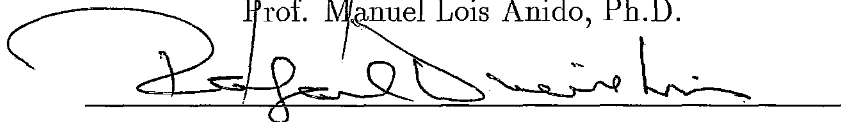


Prof. Valmir Carneiro Barbosa, Ph.D.



Prof. Philippe Navaux, Dr.Ing.

Prof. Manuel Lois Anido, Ph.D.



Prof. Rafael Dueire Lins, Ph.D.

Rio de Janeiro, RJ – Brasil
Dezembro de 1994

CHAVES FILHO, ELISEU MONTEIRO

Arquiteturas Super Escalares: Efeito de Alguns Parâmetros sobre o Desempenho.
[Rio de Janeiro] 1994.

xi, 150 p., 29,7cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 1994)

Tese – Universidade Federal do Rio de Janeiro, COPPE.

1 - Arquitetura de Computadores 2 - Processadores

3 - Paralelismo

4 - Arquiteturas Super Escalares

I. COPPE/UFRJ II. Título (série).

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

Arquiteturas Super Escalares: Efeito de Alguns Parâmetros sobre o Desempenho

Eliseu Monteiro Chaves Filho

Dezembro de 1994

Orientador: Edil S. T. Fernandes
Programa de Engenharia de Sistemas e Computação

Nas arquiteturas super escalares, múltiplas instruções são despachadas simultaneamente para execução em unidades funcionais independentes. Com isto, uma arquitetura super escalar é capaz de manter a execução de mais de uma instrução por ciclo, resultando em um ganho significativo em relação à arquiteturas *pipeline* convencionais que despacham apenas uma instrução por ciclo.

Quando comparamos as arquiteturas de processadores super escalares atualmente disponíveis, observamos várias diferenças, principalmente quanto à largura de despacho e quanto ao número e tipo das unidades funcionais. Estas diferenças naturalmente levantam a questão de como uma certa combinação de parâmetros da arquitetura afeta o desempenho. Atualmente, a escolha da configuração de uma arquitetura ainda é, em grande parte, determinada por fatores de implementação. No entanto, com os avanços esperados nas tecnologias de fabricação, haverá uma maior liberdade durante o projeto de novas arquiteturas. Com as restrições de implementação minimizadas, o balanceamento apropriado entre os componentes da arquitetura certamente será o aspecto central no projeto de uma arquitetura super escalar.

Com motivação nesta perspectiva, nesta tese investigamos o comportamento do desempenho de uma arquitetura super escalar em função de três parâmetros: a largura de despacho, o número de unidades funcionais e o grau de especialização destas unidades. Esta avaliação é feita no contexto de dois modelos de arquitetura, que diferem no modo como as dependências de controle são tratadas. Em um modelo, as dependências de controle são

resolvidas bloqueando-se o despacho de instruções. No outro modelo, instruções são executadas especulativamente na presença de dependências de controle. As dependências de controle podem representar uma séria limitação para o desempenho de uma arquitetura super escalar. Ao usar estes dois diferentes modelos, o nosso objetivo é verificar como estas dependências afetam o balanço dos componentes da arquitetura.

Para cada um destes modelos, medimos o desempenho para dois diferentes tipos de configuração de arquitetura. No primeiro tipo as unidades funcionais são homogêneas, ou seja, as unidades são idênticas e cada uma delas pode executar qualquer instrução. No segundo tipo de configuração as unidades funcionais são heterogêneas, isto é, cada unidade é especializada na execução de apenas um sub-conjunto de instruções. Dentro deste tipo, consideramos configurações com uma unidade de desvio que executa instruções de transferência de controle, unidades de memória que executam instruções de acesso à memória, e diferentes combinações de unidades para instruções aritméticas/lógicas sobre inteiros e de unidades de ponto flutuante. Em todas as configurações, também variamos a largura de despacho e o número de unidades funcionais.

Neste trabalho avaliamos o efeito combinado dos principais parâmetros de uma arquitetura super escalar, usando modelos que refletem a organização e características encontradas em arquiteturas reais. Desta forma, os resultados aqui apresentados servem como subsídio importante na orientação e avaliação de decisões no projeto de uma arquitetura super escalar.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

Superscalar Architectures: The Effect of Some Parameters on Performance

Eliseu Monteiro Chaves Filho

December, 1994

Thesis Supervisor: Edil S. T. Fernandes
Department of Systems and Computer Engineering

In superscalar architectures, multiple instructions are dispatched simultaneously for execution on independent functional units. With this ability, a superscalar architecture can sustain the execution of more than one instruction per clock cycle, what translates into a significant performance improvement over conventional, single-instruction dispatch pipelined architectures.

When comparing current superscalar architectures, we can identify several differences on their configurations, mainly concerning to the dispatch width and to the number and type of the functional units. In face of this diversity, a question which naturally arises is how the balance of the parameters of a superscalar architecture affects its performance. Today, the choice of an architectural configuration is still greatly determined by implementation factors, but expected improvements in integration technology will bring more flexiblity to the design of new architectures. The proper balancing of the components of the architecture will then represent the main tradeoff aspect in the design of a superscalar processor.

With this motivation, in this thesis we investigate the effects of three architectural parameters upon performance. These parameters are the dispatch width, the number of functional units and the specialization degree of the funcional units. This evaluation is carried within the context of two basic types of architectural models, that differ in the way control dependencies are handled. In one model, control dependencies are resolved by blocking instruction dispatch. In the other model, instructions are executed speculatively in the presence of control dependencies. Control dependencies represent a serious limitation to the performance of a superscalar architecture. Our goal in using these two models is to verify how they affect the balancing of components of a superscalar architecture.

For each of these models, we measure the performance of the architecture for two different types of architectural configurations. In the first type, functional units are homogeneous, in the sense they are all identical and execute any instruction. In the second type the functional units are heterogeneous, with each unit executing only a subset of instructions. Within this group, we consider configurations with a separate branch unit to execute control transfer instructions, with memory units that execute only memory access instructions, and configurations with different combinations of integer arithmetic/logical units and of floating-point units. For all configurations, we also change the dispatch width and the number of functional units.

In this work we evaluate the combined effect of the main parameters of a superscalar architecture, using models that reflect the organization and characteristics found in real architectures. The results here presented contribute with important, practical information on which to base the design decisions of new superscalar architectures.

À minha mãe

Agradecimentos

Ao professor Edil Fernandes, pela orientação competente.

Ao Luiz Fernando da Silva, pela ajuda na depuração dos simuladores das arquiteturas super escalares.

A Inês de Castro Dutra, por ter cedido o formato LaTeX e o utilitário para traçar gráficos, e pelas observações e sugestões quanto ao texto.

Aos colegas da COPPE que de alguma forma contribuiram para a elaboração desta tese.

Conteúdo

1 Introdução **1**

1.1 O Conceito de Arquiteturas Super Escalares 1
1.2 Exemplos de Arquiteturas Super Escalares 7
1.2.1 O Intel Pentium 7
1.2.2 O IBM RS/6000 e o IBM/Motorola PowerPC 10
1.2.3 O DEC Alpha 21064 14
1.2.4 Comparação 16
1.3 Motivação e Objetivos 18
1.4 Trabalhos Relacionados 19
1.5 Organização do Texto 21

2 Preliminares **23**

2.1 Limitações sobre o Paralelismo de Instrução 23
2.1.1 Dependências de Dados 25
2.1.2 Dependências de Controle 30
2.2 Tratamento das Dependências de Dados 33
2.2.1 Escalonamento Estático de Instruções 34
2.2.2 Escalonamento Dinâmico de Instruções 37
2.2.3 Escalonamento Estático vs. Dinâmico 41
2.3 Tratamento de Dependências de Controle 42
2.3.1 Previsão Estática de Desvios 44
2.3.2 Previsão Dinâmica de Desvios 45
2.3.3 Eficácia da Previsão de Desvios 48
2.4 Execução Especulativa 51
2.4.1 O Buffer de Reordenação 51
2.4.2 O Buffer de História 54
2.4.3 Registradores Futuros 55

3 Modelos de Arquitetura **57**

3.1 O Modelo Básico 57
3.2 O Modelo Bloqueante 61
3.3 O Modelo Especulativo 62
3.4 Validade dos Modelos 64

4 O Ambiente Experimental **66**
4.1 A Arquitetura Referência 66
4.2 O Simulador da Arquitetura SPARC 71
4.3 Os Simuladores das Arquiteturas Super Escalares 74
4.4 Os Programas de Teste 75

5 Avaliação do Modelo Bloqueante Homogêneo **78**
5.1 Configurações do Modelo Homogêneo 78
5.2 Resultados para os Progamas de Teste Inteiros 80
5.3 Resultados para os Programas de Teste de Ponto Flutuante 89
5.4 As Dependências de Controle no Modelo Bloqueante 92

6 Avaliação do Modelo Bloqueante Heterogêneo **95**
6.1 Configurações do Modelo Heterogêneo 95
6.2 O Modelo Bloqueante com Unidade de Desvio 96
6.3 O Modelo Bloqueante com Unidades de Memória 101
6.4 O Modelo Bloqueante com Diferentes ALUs 104
6.5 O Modelo Bloqueante com Unidades de Ponto Flutuante Heterogêneas 107

7 Avaliação do Modelo Especulativo **111**
7.1 Configurações do Modelo Especulativo 111
7.2 O Modelo Especulativo com Unidades Homogêneas 112
7.3 Modelo Especulativo com Unidades Heterogêneas 121

8 Conclusões **124**
8.1 O Trabalho Desenvolvido 124
8.2 O Modelo Bloqueante com Unidades Funcionais Homogêneas 125
8.3 A Especialização das Unidades Funcionais 126
8.4 As Dependências de Controle 127
8.5 Execução Especulativa de Instruções 128
8.6 Comentários Finais e Trabalhos Futuros 129

A Conjunto de Instruções SPARC **131**

B Descrição do Simulador SPARC **133**

C Descrição do Simulador da Arquitetura Super Escalar 139

D Bibliografia 143

Capítulo 1

Introdução

Para alcançar níveis mais elevados de desempenho, arquiteturas de processador têm incorporado facilidades que possibilitam a execução paralela de instruções. Um marcante passo nesta direção foi o uso da técnica *pipeline*, presente na maioria das arquiteturas lançadas nos últimos dez anos. Avanços na tecnologia de integração viabilizaram a implementação de arquiteturas *pipeline* com mais de uma unidade de execução de instruções, o que acrescentou uma outra forma de paralelismo à arquitetura. Mais recentemente, foram lançados vários processadores cujas arquiteturas possibilitam o acesso, decodificação e execução de múltiplas instruções simultaneamente. Estas arquiteturas receberam o nome de **arquiteturas super escalares**.

Este capítulo introduz o conceito de arquiteturas super escalares, mostrando porque este tipo de arquitetura possui, potencialmente, um desempenho superior ao de uma arquitetura *pipeline* convencional. Em seguida são descritas algumas arquiteturas super escalares hoje disponíveis comercialmente. Esta discussão inicial servirá de motivação para o trabalho de pesquisa apresentado nesta tese.

1.1 O Conceito de Arquiteturas Super Escalares

O desempenho de um processador pode ser medido através do tempo necessário para executar um programa: quanto menor for o tempo de execução, melhor será o desempenho. O tempo de execução de um programa depende de dois fatores: o número de instruções que são executadas, e o tempo médio de execução de cada instrução. O

tempo de execução de um programa é expresso em termos destes fatores da seguinte forma [Hennessy 90]:

$$tempo\ de\ execução = número\ de\ instruções \times tempo\ médio\ de\ execução\ de\ instrução$$

O tempo médio de execução de uma instrução depende do número médio de ciclos necessários para executar a instrução (*cpi*) e do tempo de ciclo do processador. Assim, a expressão para o tempo de execução de um programa fica:

$$tempo\ de\ execução = número\ de\ instruções \times cpi \times tempo\ de\ ciclo$$

Ganhos no desempenho podem ser obtidos reduzindo-se um ou mais fatores na expressão acima. Isto levou a diferentes abordagens para aumentar o desempenho. Até meados da década de 80, a ênfase estava na redução do número de instruções executadas. Até então, o tempo de acesso à memória principal era várias vezes maior que o tempo de ciclo do processador, e por isso o tempo total de execução de uma instrução era dominado pelo tempo necessário para acessar a instrução na memória. Assim, procurava-se reduzir o número de instruções executadas porque isto significava, na realidade, reduzir o número de acessos à memória. Isto foi conseguido com o aumento do nível de funcionalidade das instruções, de modo que uma única instrução passava a executar a mesma tarefa que antes era realizada por uma seqüência de várias instruções. Esta abordagem levou às arquiteturas conhecidas como CISC (*Complex Instruction Set Computers*) [Silbey 88].

Avanços na tecnologia de fabricação de memórias e a utilização de memórias *cache* reduziram o desbalanceamento entre os tempos de acesso da memória e de ciclo do processador. A ênfase voltou-se então para os outros dois fatores que determinam o desempenho. Para reduzir o tempo de ciclo, foi adotada uma abordagem justamente contrária à que prevalecia antes: diminuir o nível de funcionalidade das instruções. O objetivo, neste caso, era simplificar a implementação da arquitetura, de modo a eliminar os retardos inerentes a circuitos lógicos complexos que limitavam reduções no tempo de ciclo. Arquiteturas explorando esta idéia foram denominadas arquiteturas RISC (*Reduced Instruction Set Computers*) [Patterson 85] [Stallings 88].

Ao mesmo tempo, visando reduzir o número médio de ciclos por instrução, a técnica *pipeline* [Kooge 81] [Ramamoorthy 77] passou a ser amplamente usada. O desenvolvimento desta técnica é bem anterior à década de 80, mas somente a partir de

então tornaram-se disponíveis tecnologias que viabilizaram o seu emprego em microprocessadores [Childs 84]. Na técnica *pipeline*, os passos básicos que fazem parte da execução de uma instrução são realizados por estágios independentes, que operam em paralelo. A Figura 1.1(a) mostra a representação de um *pipeline* com quatro estágios.

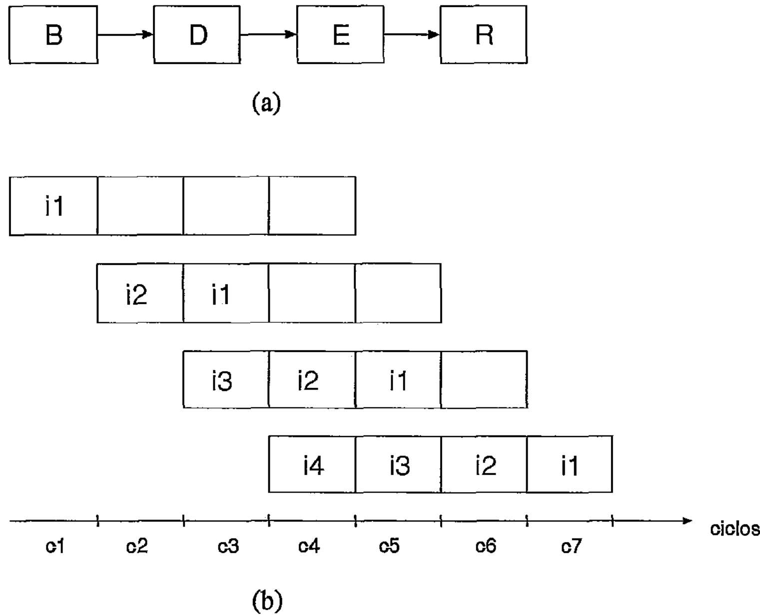


Figura 1.1: Representação e operação de um *pipeline* com quatro estágios.

Neste *pipeline*, a execução de uma instrução inicia-se no estágio B, o qual acessa o código da instrução na memória. O código da instrução é decodificado pelo estágio D. O estágio E executa a operação especificada pela instrução (p. ex., uma operação aritmética). A execução da instrução é completada pelo estágio R, que armazena o resultado produzido.

A Figura 1.1(b) mostra o fluxo de instruções através dos estágios de um *pipeline* típico. Esta figura pode ser interpretada como uma seqüência de "retratos" do *pipeline*, mostrando as instruções que estão ocupando os estágios do *pipeline* a cada ciclo. Como indica a figura, as instruções avançam para o próximo estágio a cada ciclo, com uma nova instrução sendo acessada pelo primeiro estágio e uma outra instrução sendo completada no último estágio. Desta forma, partes de várias instruções são executadas em paralelo, uma em cada estágio. Ao contrário, em uma arquitetura convencional, sem *pipeline*, as instruções são executadas seqüencialmente, ou seja, a

execução de uma nova instrução somente inicia quando a anterior estiver completada. Assim, não mais do que uma instrução está sendo executada a cada instante de tempo.

A partir da Figura 1.1(b) observa-se que, se for mantida a continuidade do fluxo de instruções, uma instrução é completada a cada ciclo, resultando em um fator cpi igual a um ciclo/instrução. Em uma arquitetura na qual instruções são executadas sequencialmente, o cpi está bem acima deste valor. Por exemplo, supondo que a execução de uma instrução envolve os quatro passos mostrados na Figura 1.1(a) e que cada passo consome um ciclo, o cpi seria de quatro ciclos/instrução. Na prática, o cpi obtido com a técnica *pipeline* apenas aproxima-se do valor ideal de um ciclo/instrução, porque existem situações onde o fluxo contínuo de instruções é interrompido (ver Capítulo 2). No entanto, arquiteturas *pipeline* reais apresentam um cpi entre 1,2 e 1,6 ciclos/instrução [Namjoo 88], o que é bem menor do que a média obtida com arquiteturas seqüenciais.

A expressão apresentada anteriormente relaciona o tempo de execução com o número médio de ciclos por instrução (cpi). Também é possível expressar o tempo de execução em termos do número médio de instruções que são completadas a cada ciclo. Denotando por ipc o número médio de instruções completadas por ciclo, e notando que cpi = 1/ipc, a expressão para o tempo de execução de um programa pode ser reescrita como:

$$tempo\ de\ execuc\tilde{a}o = \frac{n\acute{u}mero\ de\ instruc\tilde{o}es \times tempo\ de\ ciclo}{ipc}$$

Esta expressão indica que o desempenho aumenta com o ipc, o que pode ser visto intuitivamente: se um número maior de instruções é completado por ciclo, o tempo necessário para executar um mesmo número de instruções é menor. Em um *pipeline* como o que aparece na Figura 1.1, onde apenas uma instrução é completada a cada ciclo, o ipc máximo é uma instrução/ciclo. Seria possível obter um desempenho maior caso o ipc fosse elevado para além deste valor. Note que a limitação em uma instrução/ciclo deve-se ao fato de que cada estágio do *pipeline* opera sobre uma única instrução. Se cada estágio fosse capaz de operar sobre mais de uma instrução simultaneamente, mais de uma instrução poderia ser completada a cada ciclo.

A Figura 1.2 mostra o exemplo de um *pipeline* onde cada estágio processa duas instruções simultaneamente. Mantendo-se o fluxo contínuo de instruções, o ipc obtido será de duas instruções/ciclo. Esta figura mostra, em uma forma simplificada, como

instruções são executadas em um *pipeline* super escalar. Neste tipo de *pipeline*, a execução paralela de instruções também acontece dentro de cada estágio. O termo *pipeline escalar* será aqui usado para referir-se a um *pipeline* como o que aparece na Figura 1.1, onde apenas uma instrução é processada em cada estágio e completada a cada ciclo.

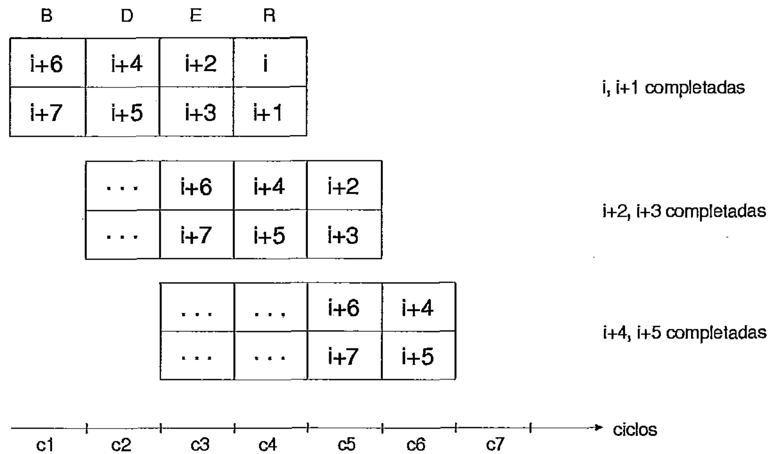


Figura 1.2: Executando mais de uma instrução por estágio do *pipeline*.

Arquiteturas super escalares são organizadas em torno de um *pipeline* como o mostrado na Figura 1.2, com estágios que operam sobre mais de uma instrução simultaneamente. A Figura 1.3 mostra a organização básica de uma arquitetura super escalar. A unidade de despacho corresponde aos estágios de busca e decodificação do *pipeline*. A cada ciclo, a unidade de despacho acessa e decodifica um certo número de instruções, e envia para as unidades funcionais instruções que foram decodificadas em ciclos anteriores. O envio de uma instrução para uma unidade funcional é chamado de **despacho de instrução**.

As unidades funcionais pertencem ao estágio de execução do *pipeline*. São estas unidades que realizam a operação especificada pela instrução (p. ex., uma adição ou um acesso à memória). Uma arquitetura super escalar é dotada de múltiplas unidades funcionais independentes, possibilitando a execução de operações em paralelo. Em algumas arquiteturas super escalares as unidades funcionais são homogêneas, no sentido que cada unidade pode executar qualquer operação. Em outras arquiteturas super escalares, as unidades funcionais são heterogêneas, ou seja, cada unidade executa apenas um subconjunto de operações (p. ex., uma unidade executa ape-

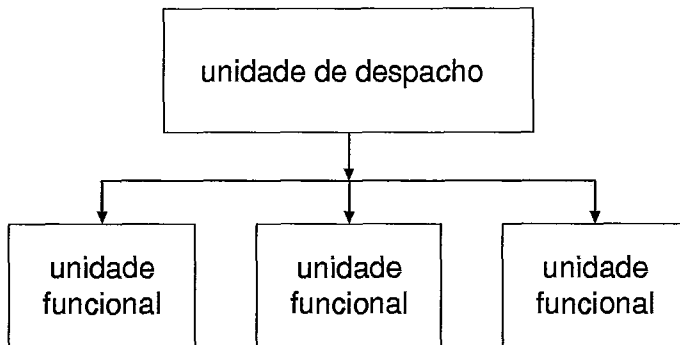


Figura 1.3: Arquitetura super escalar básica.

nas operações aritméticas e lógicas, enquanto uma outra executa apenas acessos à memória). As unidades funcionais podem ser seqüenciais ou *pipeline*. No primeiro caso, a unidade funcional inicia a execução de uma nova operação somente quando a operação anterior estiver completada. Ao contrário, unidades funcionais *pipeline* são implementadas sob a forma de estágios que operam em paralelo, podendo assim executar várias operações simultaneamente.

Em uma arquitetura super escalar, o despacho de uma instrução está sujeito a diversas condições. Por exemplo, o despacho é condicionado à disponibilidade dos recursos envolvidos na execução da instrução (p. ex., unidade funcional, registradores). O despacho também é condicionado às dependências entre instruções (dependências entre instruções seão discutidas no Capítulo 2). Independente destas restrições, o despacho de instruções pode ser **em-ordem** ou **fora-de-ordem**. No despacho em-ordem, as instruções são enviadas para as unidades funcionais exatamente na mesma ordem em que se encontram no código do programa. Note que isto não impede o despacho de múltiplas instruções em um mesmo ciclo. Por exemplo, suponha que as instruções i, $i+1$ e $i+2$ estejam nesta ordem no código. No despacho em-ordem, tais instruções podem ser despachadas simultaneamente, mas caso sejam despachadas em ciclos diferentes, i é despachada antes de $i+1$ e $i+2$. Ao contrário, no despacho fora-de-ordem, instruções podem ser despachadas em uma ordem diferente daquela em que se encontram no código.

É importante precisar o conceito de arquiteturas super escalares adotado neste trabalho. Alguns autores consideram arquiteturas super escalares como sendo "todos

os processadores que podem executar duas ou mais operações escalares em paralelo" [Lam 90]. Esta definição abrange uma gama extensa de tipos de arquiteturas, desde as VLIW [Fisher 83] até as *superpipelined* [Mirapuri 92] [Bashteen 91]. Em particular, estariam também incluídas nesta definição arquiteturas tais como SPARC, MIPS e Intel 80486, as quais possuem unidades funcionais independentes que operam em paralelo. No entanto, estas arquiteturas não são reconhecidas como sendo super escalares, pois apenas uma instrução é acessada, decodificada e enviada para execução a cada ciclo. Agerwala e Cocke definem uma arquitetura super escalar como sendo " uma máquina capaz de despachar múltiplas instruções por ciclo a partir de um fluxo único de instruções" [Agerwala 87]. Esta definição melhor caracteriza uma arquitetura super escalar (o uso do termo super escalar é, na realidade, creditado a estes dois autores). Neste trabalho, considera-se que uma arquitetura super escalar é aquela que apresenta as seguintes características: (1) um *pipeline* onde cada estágio processa mais de uma instrução escalar simultaneamente (o que implica a existência de múltiplas unidades funcionais no estágio de execução); (2) mais de uma instrução pode ser despachada para execução nas unidades funcionais a cada ciclo; (3) a execução de múltiplas operações em paralelo não requer o empacotamento prévio destas operações em uma única instrução.

1.2 Exemplos de Arquiteturas Super Escalares

Os mais recentes processadores destinados a aplicações de alto desempenho exibem arquiteturas super escalares. A título de exemplo, descrevemos a seguir as arquiteturas de quatro processadores super escalares comerciais: Intel Pentium, IBM RS/6000, IBM/Motorola/Apple PowerPC 603 e DEC Alpha 21064. É importante frisar que estas descrições não são completas, mas mostram como o conceito de arquitetura super escalar é, em seus principais aspectos, concretizado em implementações reais. O principal objetivo da apresentação destes exemplos é ressaltar as diferenças mais significativas entre processadores super escalares atualmente disponíveis. As diferenças que poderão ser observadas constituem um dos fatores motivantes deste trabalho.

1.2.1 O Intel Pentium

A Figura 1.4 mostra a organização do Intel Pentium [Alpert 93] [Crawford 93] [Saini 93]. O Pentium inclui três unidades funcionais, denominadas U-pipe, V-pipe e FPU.

As unidades U-pipe e V-pipe executam instruções aritméticas e lógicas sobre números inteiros, instruções de acesso à memória e instruções de desvio. A FPU executa instruções sobre números em ponto flutuante.

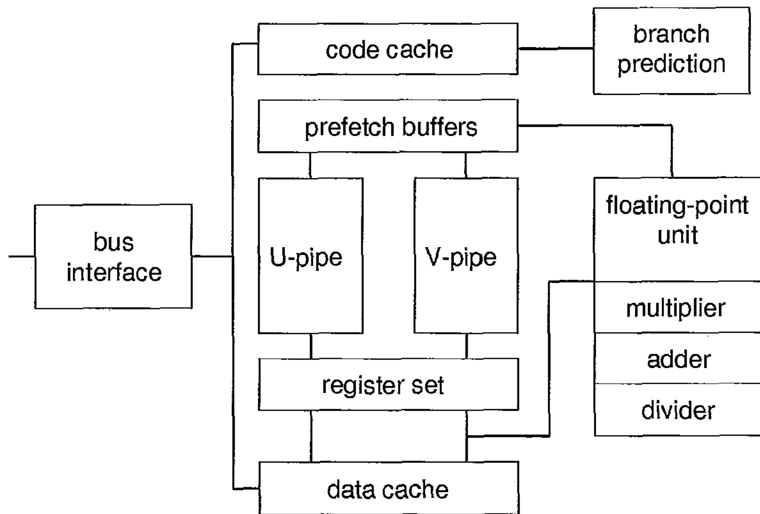


Figura 1.4: Organização da arquitetura do Intel Pentium.

A Figura 1.5 mostra a estrutura do *pipeline* super escalar do Pentium. A cada ciclo, o estágio PF acessa duas instruções, que são pré-decodificadas e despachadas pelo estágio D1. As unidades U-pipe e V-pipe possuem *pipelines* idênticos. O estágio D2 completa a decodificação da instrução. O estágio E acessa os operandos armazenados nos registradores e executa uma operação na ALU, um acesso à memória, ou uma instrução de desvio. O estágio WB armazena o resultado da operação. Na FPU, o estágio D2 completa a decodificação da instrução, e o estágio E acessa os operandos. Os estágios X1 e X2 executam a operação de ponto flutuante, e o estágio WF armazena o resultado. O estágio ER faz o tratamento de exceções que podem ocorrer durante a execução da instrução de ponto flutuante.

No Pentium, até duas instruções podem ser despachadas simultaneamente. O estágio D1 decide se as instruções podem ser enviadas para execução no mesmo ciclo. Para que duas instruções sejam despachadas no mesmo ciclo, três condições devem ser satisfeitas. A primeira condição é a de que as instruções devem ser "simples", isto é, aquelas cuja execução não é controlada pelo microprograma. Instruções deste tipo não provocam conflitos na utilização dos recursos, quando despachadas e executadas

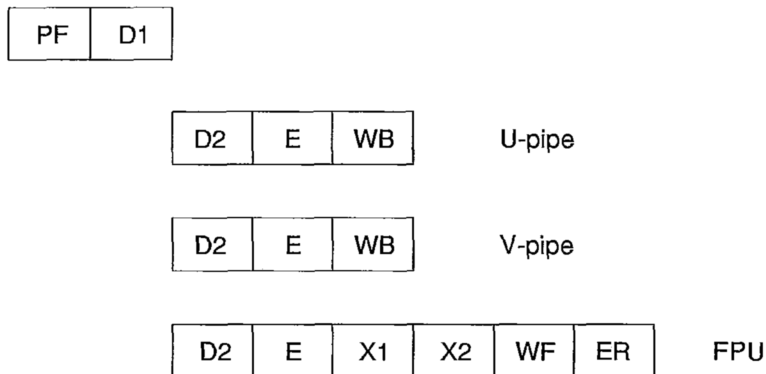


Figura 1.5: Estrutura do *pipeline* super escalar no Intel Pentium.

simultaneamente. Como segunda condição, o registrador destino de uma instrução não deve coincidir com os registradores fonte ou destino da outra instrução. Isto assegura que dependências de dados entre as instruções sejam satisfeitas (dependências entre instruções serão discutidas no Capítulo 2). A última condição é a de que a primeira instrução do par não pode ser uma instrução de desvio. Esta condição assegura que dependências de controle sejam respeitadas. Se uma destas condições não for satisfeita, a primeira instrução do par é despachada para a unidade U-pipe. No ciclo seguinte, a segunda instrução do par é associada com uma nova instrução, e o estágio D1 novamente verifica se ambas podem ser despachadas simultaneamente. Se este ainda não for o caso, a instrução do par anterior é despachada e o processo se repete.

Ao decodificar uma instrução de desvio, o estágio D1 faz uma previsão do resultado do desvio e instruções passam a ser acessadas a partir do destino previsto. O despacho destas instruções é bloqueado até que o desvio seja resolvido. Se a previsão estiver correta, estas instruções são despachadas, caso contrário, elas são descartadas e as instruções no destino correto serão acessadas. A previsão de desvios é feita dinamicamente, através de um *branch target buffer* (mecanismos de previsão de desvios serão discutidos no Capítulo 2).

1.2.2 O IBM RS/6000 e o IBM/Motorola PowerPC

O IBM RS/6000 e os processadores PowerPC do consórcio IBM/Motorola/Apple são baseados na arquitetura IBM POWER (*Performance Optimization With Enhanced RISC*) [Oehler 92] [Diefendorff 94]. A arquitetura do RS/6000 [Grohoski 90] [Oehler 90] é mostrada na Figura 1.6. O RS/6000 é organizado em torno de três unidades funcionais, denominadas ICU, FXU e FPU. A ICU realiza o acesso e despacho de instruções, e também executa as instruções de desvio (na sub-unidade denominada *branch processor*). A FXU executa instruções aritméticas e lógicas sobre inteiros e instruções de acesso à memória. A unidade FPU realiza as operações com números em ponto flutuante.

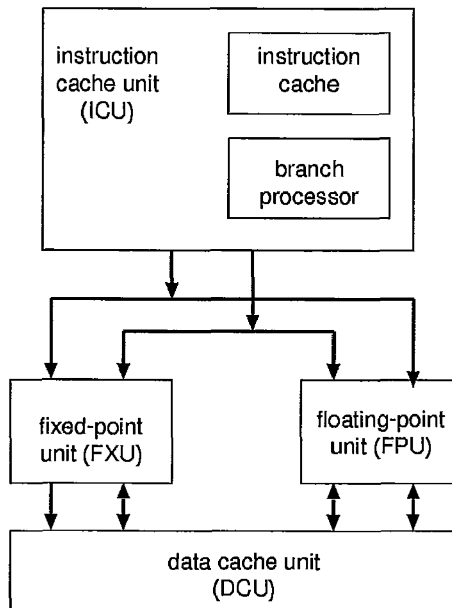


Figura 1.6: Organização do IBM RS/6000.

A Figura 1.7 mostra a estrutura do *pipeline* super escalar do RS/6000. A cada ciclo, o estágio IF acessa quatro instruções. O estágio Disp/BRE despacha até quatro instruções por ciclo: uma instrução para a FXU, uma instrução para a FPU e até duas instruções para a própria ICU. O par de instruções despachado para a ICU é formado por uma instrução de desvio e uma instrução que manipula os registradores de condição. O estágio Disp/BRE também executa as instruções de desvio. Quando possível, o desvio é executado no mesmo ciclo em que a instrução é despachada.

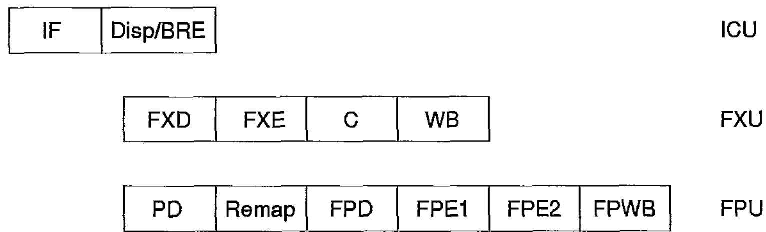


Figura 1.7: Estrutura do *pipeline* super escalar no RS/6000.

Na FXU, o estágio FXD decodifica as instruções e acessa os operandos armazenados no banco de registradores. O estágio FXE executa a operação na ALU. Nas instruções de acesso à memória, o acesso é executado pelo estágio C. O estágio WB armazena o resultado produzido pela instrução. Na FPU, o estágio PD pré-decodifica instruções e o estágio Remap faz uma renomeação de registradores para eliminar eventuais dependências entre instruções. O estágio FPD completa a decodificação da instrução e acessa os operandos no banco de registradores de ponto flutuante. A operação de ponto flutuante é executada nos estágios FPE1 e FPE2, e o resultado é armazenado pelo estágio FPWB.

No RS/6000, o despacho de instruções segue regras simples. Instruções para a FXU e para a FPU podem ser despachadas em qualquer combinação: uma instrução para a FXU e outra para a FPU, ambas para a FXU ou ainda ambas para a FPU. Qualquer que seja a combinação, as duas instruções são despachadas para a FXU e para a FPU, e uma unidade descarta instruções que sejam para a outra. A FXU e a FPU possuem *buffers* para armazenar as instruções despachadas que ainda não entraram em execução. O despacho é temporariamente suspenso se não existirem *buffers* disponíveis. Instruções com dependências são despachadas normalmente, e estas dependências são resolvidas nos estágios FXD e Remap.

A arquitetura do PowerPC 601 [Becker 93] [Diefendorff 94] é semelhante à arquitetura do RS/6000. O PowerPC 601 possui as mesmas três unidades funcionais encontradas no RS/6000: a FXU, que executa instruções sobre inteiros e instruções de acesso à memória; a FPU, que executa instruções sobre números em ponto flutuante; e a BPU, que executa instruções de desvio. No PowerPC 601, até três instruções podem ser despachadas por ciclo. O despacho para a FXU é feito apenas em-ordem, mas o despacho para a FPU e BPU pode acontecer fora-de-ordem. O despacho

para a FPU e BPU prossegue mesmo que o despacho para a FXU esteja bloqueado. O despacho é bloqueado quando é encontrada uma instrução com dependência ou quando ocorrer conflito na utilização de uma unidade funcional. Assim como no RS/6000, as instruções de desvio no PowerPC 601 podem ser executadas no mesmo ciclo do despacho. Um mecanismo de previsão estática é usado para prever o resultado de desvios condicionais que não podem ser executados imediatamente. As instruções no caminho previsto são despachadas, mas serão executadas apenas se a previsão do desvio estiver correta.

A arquitetura do PowerPC 603 [Burgess 94] apresenta várias diferenças em relação à arquitetura do RS/6000. A Figura 1.8 mostra a organização do PowerPC 603.

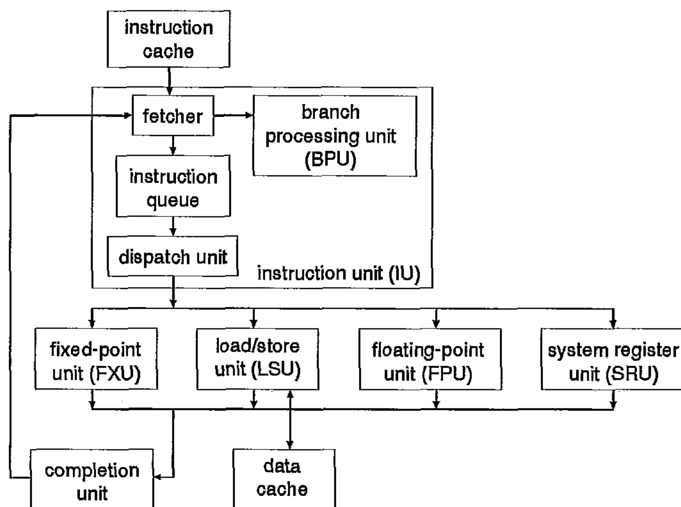


Figura 1.8: Organização do PowerPC 603.

O PowerPC 603 possui cinco unidades funcionais. Além da FXU, FPU e BPU, também encontradas no PowerPC 601, foram incluídas duas outras unidades: a LSU, que executa as instruções de acesso à memória (no PowerPC 601 e RS/6000, estas instruções são executadas pela FXU), e a SRU, que executa instruções sobre registradores especiais (p. ex., o registrador de condição). A unidade BPU, juntamente com os componentes que controlam a busca e o despacho de instruções, formam a unidade denominada IU (*Instruction Unit*).

A estrutura do *pipeline* super escalar no PowerPC 603 aparece na Figura 1.9. A cada ciclo, o estágio IF acessa duas instruções, colocando-as em uma fila. O estágio DSP

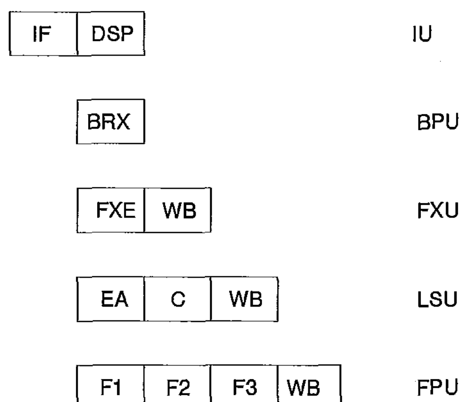


Figura 1.9: Estrutura do *pipeline* super escalar no PowerPC 603.

decodifica instruções, acessa operandos em registradores e despacha as instruções para as respectivas unidades funcionais. Até três instruções podem ser despachadas por ciclo. As duas primeiras instruções da fila são despachadas para as unidades FXU, FPU, LSU ou SRU. Instruções para estas unidades são despachadas em-ordem. Uma instrução de desvio é imediatamente despachada qualquer que seja a sua posição na fila, ou seja, instruções de desvio podem ser despachadas fora-de-ordem. O despacho é bloqueado quando uma unidade funcional estiver ocupada. Para evitar bloqueios no despacho provocados por dependências de dados, cada unidade funcional possui um *buffer* denominado estação de reserva. Uma instrução com dependência é despachada para a estação de reserva da unidade funcional, e ali permanece até que todos os seus operandos estejam disponíveis e possa ser executada. Enquanto isso, instruções subseqüentes podem ser despachadas e executadas. Este esquema é uma simplificação do algoritmo de Tomasulo, que será analisado em detalhes no Capítulo 2.

Instruções de desvio são executadas no estágio BRX, na BPU. Quando possível, estas instruções são executadas no mesmo ciclo em que são despachadas. Se uma instrução de desvio condicional não pode ser imediatamente executada, o estágio BRX faz uma previsão estática do resultado do desvio. As instruções no caminho previsto são executadas especulativamente (execução especulativa de instruções será abordada no Capítulo 2). No PowerPC 603 a execução especulativa ocorre através de uma instrução de desvio apenas, ou seja, a execução das instruções seguintes a um segundo desvio são executadas somente após a execução do desvio anterior. Operações inteiras na ALU são executadas no estágio FXE da FXU. Operações de ponto flutuante são

executadas ao longo dos estágios F1 a F3, na FPU. Acessos à memória são executados em dois estágios, na LSU: o estágio EA calcula o endereço efetivo da posição de memória, e o estágio C acessa a memória *cache* de dados.

Quando uma operação é completada no estágio de execução de uma unidade funcional, o resultado é armazenado em um registrador temporário, alocado pelo estágio DSP no despacho da instrução. O estágio WB da unidade funcional copia o resultado do registrador temporário no registrador destino, seguindo a ordem indicada pelo *buffer* de término (*completion buffer*). O *buffer* de término é uma fila, onde instruções são inseridas na ordem em que são despachadas. O registrador destino de uma instrução é atualizado somente quando a instrução encontra-se na primeira posição do *buffer*. Assim, os registradores são modificados em uma ordem consistente com a ordem de despacho das instruções. Este mecanismo foi incluído no PowerPC 603 para suportar interrupções precisas e execução especulativa de instruções. Este e outros mecanismos usados para execução especulativa serão descritos no Capítulo 2.

O PowerPC 604 [IBM 94] [Song 94] possui os mesmos tipos de unidades funcionais encontradas no PowerPC 603, exceto pela existência de uma unidade de inteiros específica para instruções com múltiplos ciclos de latência (multiplicação e divisão) e de duas unidades de inteiros para instruções com um único ciclo de latência. Todas unidades funcionais possuem duas estações de reserva. Até quatro instruções podem ser despachadas por ciclo. O PowerPC 604 também suporta execução especulativa de instruções, mas agora a especulação pode envolver até duas instruções de transferência de controle. O resultado de uma instrução de desvio é previsto usando-se previsão dinâmica, ao invés de previsão estática como acontece no PowerPC 601 e 603.

1.2.3 O DEC Alpha 21064

O DEC Alpha 21064 [McLellan 93] é a primeira implementação da arquitetura DEC Alpha AXP [Sites 92] [Sites 93]. Sua organização é mostrada na Figura 1.10. O Alpha 21064 possui três unidades funcionais: a unidade EBox executa instruções sobre inteiros e instruções de desvio; a unidade FBox executa instruções sobre números em ponto flutuante; a unidade ABox executa instruções de acesso à memória. A unidade IBox realiza a busca, decodificação e despacho de instruções.

A Figura 1.11 mostra a estrutura do *pipeline* super escalar no Alpha 21064. Os quatro primeiros estágios fazem parte da unidade IBox. O estágio IF acessa duas instruções a

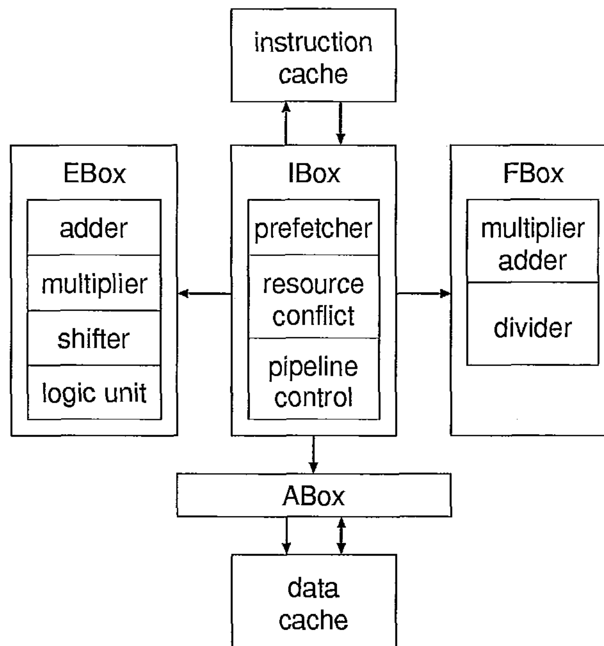


Figura 1.10: Organização do DEC Alpha 21064.

cada ciclo. O estágio SW direciona as duas instruções para os *buffers* de decodificação corretos e faz a previsão do resultado de instruções de desvio. O estágio D decodifica as instruções, em preparação para o despacho. O estágio I verifica dependências entre instruções, acessa operandos em registradores e despacha as instruções para as unidades funcionais apropriadas.

Os estágios A1 e A2 da EBox executam a operação inteira na ALU, e o estágio IWR armazena o resultado no registrador destino. O resultado de uma instrução de desvio é determinado no estágio A1. Na FBox, a operação de ponto flutuante é executada ao longo dos estágios F1 a F5, e o resultado é armazenado pelo estágio FWR. Na ABox, o endereço efetivo da posição de memória é calculado no estágio EA. Em uma instrução *load*, a memória *cache* é acessada no estágio ACC, e o dado armazenado no registrador pelo estágio MW. Em uma instrução *store*, o dado é armazenado na *cache* pelo estágio MW.

No Alpha 21064, até duas instruções podem ser despachadas a cada ciclo. Instruções são despachadas em-ordem. Conflitos na utilização de recursos impedem o despacho de algumas combinações de instruções em um mesmo ciclo. O estágio SW verifica as

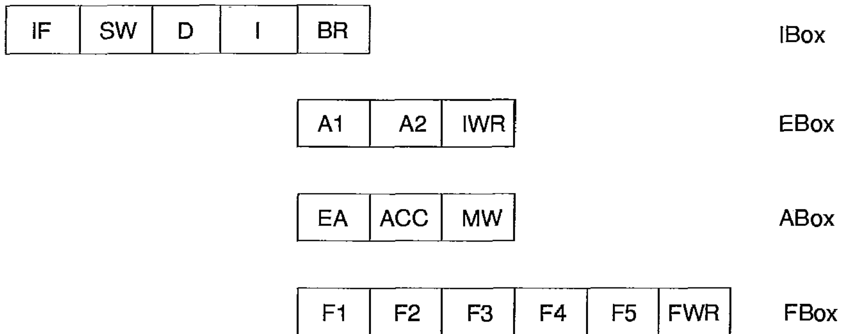


Figura 1.11: Estrutura do *pipeline* super escalar no DEC Alpha 21064.

restrições de despacho. Quando não for possível despachar duas instruções no mesmo ciclo, o estágio SW serializa as instruções, enviando-as em ciclos diferentes para o estágio seguinte. Dependências de dados entre instruções são verificadas no estágio I, e existindo alguma dependência, o despacho de instruções é serializado ou então bloqueado. O despacho de instruções também é bloqueado quando uma instrução de desvio chega ao estágio A1 e não pode ser executada imediatamente.

1.2.4 Comparação

A Tabela 1.1 resume as principais características dos processadores super escalares aqui descritos. Esta tabela também inclui as características de três outros processadores: Motorola M88110 [Diefendorff 92], HP PA7100 [Asprey 93] e Sun SuperSPARC [Blanck 92]. As descrições destes processadores deixam de ser aqui incluídas para não tornar esta seção por demais extensa. Na tabela são destacados: número de instruções que podem ser despachadas por ciclo, as condições que limitam o despacho de instruções, o número e o tipo de unidades funcionais, e a existência de facilidades especiais, tais como previsão de desvios e execução especulativa.

Na maioria dos processadores, apenas duas instruções são despachadas por ciclo. O RS/6000 e o PowerPC 604 possuem a maior capacidade de despacho, com até quatro instruções despachadas por ciclo. Existe uma diferença no balanço entre o número de instruções despachadas e o número de unidades funcionais. No Pentium, a relação é de duas instruções despachadas para duas unidades, enquanto no M88110 a

Processador	Unidades	Despacho	Limitações no Despacho	Previsão	Execução Especulativa	Observações
Pentium	2 integer 1 floating-point	2	dependências dados dependências controle recursos	dinâmica	não	
Alpha 21064	1 integer 1 floating-point 1 branch 1 load/store	2	dependências dados dependências controle recursos	estática dinâmica	não não	
RS/6000	1 integer 1 floating-point 1 branch	4	recursos	estática	não	resolução de dependências após despacho
PowerPC 601	1 integer 1 floating-point 1 branch	3	dependências dados dependências controle recursos	estática	não	
PowerPC 603	1 integer 1 floating-point 1 branch 1 load/store 1 system	3	recursos	estática dinâmica	sim	estações de reserva especulação de um desvio
PowerPC 604	3 integer 1 floating-point 1 branch 1 load/store 1 system	4	recursos	estática dinâmica	sim	estações de reserva especulação de dois desvios
M88110	2 integer 1 bit field 1 fp add/sub 1 multiply 1 divide 1 load/store 1 branch 2 graphics	2	recursos	estática	sim	estações de reserva *scoreboarding* (Capitulo 2) especulação de um desvio
PA7100	1 integer 1 floating-point	2	dependências dados dependências controle recursos	estática	não	
Super SPARC	1 integer 1 floating-point	3	dependências dados dependências controle recursos	estática	não	

Tabela 1.1: Comparação de alguns processadores super escalares.

relação é de duas instruções para dez unidades funcionais. Entre estes dois extremos, existem diferentes combinações destes dois parâmetros. Em alguns processadores as dependências são resolvidas no momento do despacho, através de regras que impedem o despacho de instruções dependentes no mesmo ciclo. Isto acontece, por exemplo, no Pentium, no Alpha 21064 e no PowerPC 601. Em outros processadores, como o PowerPC 603 e PowerPC 604, as dependências impõem restrições mínimas sobre o despacho.

Alguns processadores incorporam unidades que executam diversos tipos de instruções. Este é o caso do Pentium, PA7100 e SuperSPARC, nos quais uma mesma unidade executa instruções aritméticas e lógicas sobre inteiros, de transferência de controle e de acesso à memória. Em outros processadores, as unidades funcionais apresentam um maior nível de especialização. No RS/6000 e no PowerPC 601, instruções de transferência de controle são executadas por uma unidade específica. No Alpha 21064, PowerPC 603 e PowerPC 604, as instruções de acesso à memória também são executadas por uma unidade à parte. Uma divisão de funcionalidade ainda mais fina é encontrada no M88110, que dedica unidades separadas para execução de algumas instruções aritméticas e lógicas que em outros processadores são executadas por uma mesma unidade. O M88110 também possui unidades separadas para executar diferentes operações de ponto-flutuante. Alguns processadores possuem unidades replicadas, normalmente unidades que executam operações sobre inteiros: o Pentium possui duas unidades de inteiros, enquanto o PowerPC 604 inclui três unidades de inteiros.

Todos os processadores acima relacionados usam mecanismos de previsão de desvios, mas apenas em alguns a previsão é dinâmica. Outros processadores, basicamente os que colocam restrições mínimas no despacho, são capazes de executar instruções especulativamente.

1.3 Motivação e Objetivos

Como pode ser visto a partir da seção anterior, existem várias diferenças entre os processadores super escalares atualmente disponíveis. Em face desta diversidade, uma questão que naturalmente aparece é a de como o desempenho é afetado por estas diferentes combinações de parâmetros da arquitetura. Atualmente, a configuração de uma arquitetura ainda é, em grande parte, determinada por fatores de implementação tais como densidade de integração, dissipação de potência e custo. No entanto, com os

avanços esperados nas tecnologias de fabricação, haverá uma maior liberdade durante o projeto de novas arquiteturas. Com as restrições de implementação minimizadas, o balanceamento apropriado entre os componentes da arquitetura certamente será o aspecto central no projeto de uma arquitetura super escalar.

Com motivação nesta perspectiva, este trabalho tem como objetivo investigar a relação entre a configuração de uma arquitetura super escalar e o seu desempenho. Mais precisamente, pretende-se investigar como os principais parâmetros de uma arquitetura super escalar atuam, em conjunto, sobre o desempenho. Estes parâmetros seriam: o número de instruções despachadas, o número de unidades funcionais, o nível de especialização das unidades funcionais e as restrições no despacho de instruções. Desta forma, o presente estudo pretende contribuir com informações e conhecimentos que sirvam de subsídios importantes na orientação e avaliação de decisões no projeto de uma arquitetura super escalar.

A metodologia usada neste trabalho é essencialmente experimental. Foram elaborados dois modelos de arquitetura que diferem no modo como o despacho de instruções é afetado por dependências de controle. No primeiro, o despacho de instruções é interrompido pelas instruções de desvio. No segundo, é acrescentado o suporte necessário para execução especulativa de instruções, permitindo que o despacho e execução de instruções continue na presença de dependências de controle. Com esta diferença, pretende-se comparar arquiteturas que resolvem dependências no despacho com aquelas onde o despacho é limitado somente pela disponibilidade de recursos. Dentro de cada um destes modelos, foram consideradas configurações de arquitetura com diferentes combinações de capacidades de despacho, número de unidades funcionais e níveis de especialização das unidades funcionais. Com estas variações pretende-se, conforme mencionado acima, verificar como estas características atuam sobre o desempenho de uma arquitetura super escalar. Para alcançar estes objetivos, foram implementados simuladores completamente parametrizados, que reproduzem o comportamento dos dois modelos de arquitetura em suas diferentes configurações.

1.4 Trabalhos Relacionados

Arquiteturas com unidades funcionais distintas datam da década de 60, mas esta característica era encontrada apenas em sistemas de grande porte. Os exemplos clássicos são o CDC 6600 [Thornton 64], [Thornton 70], e o IBM 360 modelos 85, 91

e 195 [Amdahl 64], [Bell 71]. Pleszkun e Sohi mostram que um ganho significativo no desempenho pode ser obtido com o uso de múltiplas unidades funcionais [Pleszkun 88]. Estes autores utilizaram a arquitetura do Cray-1 como modelo básico, modificando-a com a replicação da unidade funcional que executa as operações escalares. Com múltiplas unidades funcionais, foi obtido um ganho no desempenho de até 39% em relação à arquitetura original.

O primeiro trabalho sobre despacho de múltiplas instruções deve-se a Tjaden e Flynn [Tjaden 70] [Tjaden 73]. Estes autores abordam o problema de como encontrar instruções independentes que podem ser despachadas simultaneamente. A solução proposta emprega uma fila (denominada *pre-decode stack*), onde novas instruções são inseridas a cada ciclo. Cada instrução é comparada com as instruções precedentes que se encontram na fila, e aquelas que não apresentam dependências são enviadas para execução. No entanto, o esquema de Tjaden e Flynn exige um circuito lógico que realize $O(n^2)$ comparações, para uma fila com n instruções. Um outro mecanismo de despacho múltiplo de instruções, denominado *dispatch stack*, foi proposto por Acosta [Acosta 86]. O *dispatch stack* também usa uma fila, mas para cada instrução da fila estão associadas informações sobre suas dependências. Estas informações indicam as instruções que podem ser despachadas, e são atualizadas na medida que instruções são executadas. A inclusão de informações explícitas sobre dependências evita a comparação entre instruções. Um outro mecanismo, proposto por Dwyer e Torng [Dwyer 92], consiste em uma adaptação do *dispatch stack* para suportar execução especulativa.

Existem vários trabalhos propondo modelos de arquiteturas super escalares que incorporam diferentes facilidades. Hwu e Patt [Hwu 86], [Patt 85a], [Patt 85b] descrevem um modelo denominado HPS, (*High Performance Substrate*), que se baseia na construção dinâmica de um grafo de dependências para determinar as instruções independentes que podem ser despachadas para execução. Na realidade, este mecanismo é baseado no algoritmo de Tomasulo, descrito no Capítulo 2. Sohi e Vajapeyam [Sohi 87], [Sohi 90] descrevem um modelo de arquitetura que integra, em um único mecanismo, a resolução de dependências entre instruções e o suporte para interrupções precisas.

Os trabalhos acima citados em geral avaliam parâmetros no contexto da proposta de uma nova técnica ou mecanismo. O presente trabalho se diferencia por mostrar o efeito combinado dos principais parâmetros de uma arquitetura super escalar, com o uso de modelos que incluem características encontradas em arquiteturas super esca-

lares reais.

Buleo [Buleo93] avalia a eficiência do algoritmo de Tomasulo e analisa o efeito de alguns parâmetros tais como o número de CDB's e o número de unidades virtuais. Ao contrário, no presente trabalho o algoritmo de Tomasulo não é alvo de avaliação, sendo usado apenas no papel de um mecanismo eficiente para a resolução de dependências de dados. A maioria dos parâmetros aqui considerados não estão diretamente relacionados com o algoritmo de Tomasulo, mas sim com as características mais gerais da arquitetura onde este mecanismo encontra-se inserido.

Hsing [Hsing93] avalia o efeito das interrupções precisas e da execução especulativa no desempenho de arquiteturas super escalares. O presente trabalho não considera o efeito de interrupções, embora estas introduzam dependências de controle. Por outro lado, execução especulativa foi considerada porque ela altera sensivelmente o nível de utilização das unidades funcionais. Mas, novamente, a execução especulativa não é aqui o alvo principal de estudo, mas sim como esta facilidade altera o balanceamento entre os componentes de uma arquitetura super escalar.

Além dos trabalhos acima mencionados, existem vários outros que abordam diversos aspectos relacionados com arquiteturas super escalares, tais como: grau de paralelismo de instrução encontrado em aplicações, escalonamento estático e dinâmico de instruções, previsão de desvios, execução especulativa e interrupções precisas. Para evitar repetições, referências e comentários sobre tais trabalhos serão feitos à medida que estes tópicos específicos forem abordados ao longo dos próximos capítulos.

1.5 Organização do Texto

Este texto está dividido em três partes. A primeira parte, introdutória, inclui o presente capítulo e o Capítulo 2. No Capítulo 2 é discutido o problema das dependências de dados e de controle. Tal discussão é essencial porque são as dependências que determinam o paralelismo de instrução a ser explorado por uma arquitetura super escalar. Além de definir estas duas formas de dependências, serão apresentadas técnicas de escalonamento estático e dinâmico de instruções, que tratam dependências de dados, e mecanismos de previsão dinâmica de desvios e de execução especulativa de instruções, que atacam as dependências de controle.

A segunda parte do texto aborda aspectos práticos da tese. No Capítulo 3 são a-

presentados os dois modelos de arquitetura considerados neste trabalho. O Capítulo 4 trata do ambiente experimental, onde são descritos os simuladores correspondentes aos modelos de arquitetura. São também apresentados os programas de teste usados na avaliação dos modelos.

Na terceira parte são apresentados os resultados dos experimentos. Os Capítulos 5 e 6 avaliam o desempenho do modelo bloqueante. O Capítulo 5 apresenta resultados para configurações deste modelo com unidades funcionais homogêneas, isto é, as unidades funcionais executam qualquer tipo de instrução. No Capítulo 6 consideram-se as configurações com unidades funcionais heterogêneas, ou seja, cada unidade é especializada e executa apenas um certo tipo de instrução. O modelo especulativo é avaliado no Capítulo 7, em configurações com unidades funcionais homogêneas e heterogêneas. O Capítulo 8 apresenta as principais conclusões do trabalho.

Capítulo 2

Preliminares

Arquiteturas super escalares possibilitam a execução de múltiplas instruções por ciclo. No entanto, existem relações entre as instruções de um programa que limitam a execução paralela de instruções. Estas relações são denominadas **dependências**. A existência de dependências entre instruções torna necessária a introdução de mecanismos na arquitetura e/ou de técnicas de otimização no compilador, para minimizar as limitações impostas sobre o paralelismo.

Este capítulo discute o problema das dependências, como elas afetam a execução paralela de instruções, e como elas são tratadas a nível de *hardware* e de *software*. Esta visão preliminar é necessária não somente pelo papel central do tratamento de dependências dentro do contexto de arquiteturas super escalares, mas também porque vários dos mecanismos aqui descritos serão incluídos nos modelos de arquitetura usados neste trabalho.

2.1 Limitações sobre o Paralelismo de Instrução

O desempenho obtido com uma arquitetura super escalar depende fundamentalmente da sua capacidade de executar instruções em paralelo. O **paralelismo de instrução** de um programa refere-se ao volume de instruções do programa que podem ser executadas em paralelo. O paralelismo de instrução é uma característica fortemente ligada às peculiaridades do código, e assim varia entre programas e entre classes de aplicações. Vários estudos já foram realizados para medir o grau de paralelismo de

Trabalho	Programas	Paralelismo	Modelo
Butler 91	computação com inteiros e com ponto-flutuante	5,8 - 17	número ilimitado de unidades funcionais e escalonamento global
Jouppi 89	*livermore loops* e utilitários UNIX	1,6 - 3,2	número limitado de unidades funcionais e escalonamento local
Nicolau 84	programas científicos	2,2 - 28	número ilimitado de unidades funcionais e escalonamento global
Wall 91	*livermore loops*, *linpack* e utilitários UNIX	5 - 7	número ilimitado de unidades funcionais e escalonamento global
Smith 89	computação com inteiros	2,5 - 4,1	número limitado de unidades funcionais e escalonamento global

Tabela 2.1: Paralelismo de instrução obtido em diversos estudos.

instrução em diferentes programas e classes de aplicações. Como exemplo, podemos citar [Butler 91], [Jouppi 89], [Nicolau 84], [Wall 91], [Smith 89]. A Tabela 2.1 resume os resultados obtidos nestes trabalhos. Esta tabela indica os tipos de programas considerados em cada estudo, e o número médio de instruções que podem ser executadas em paralelo encontrado para tais programas.

Alguns trabalhos reportam um alto grau de paralelismo de instrução (Butler e Nicolau), enquanto outros (Wall, Smith e Jouppi) indicam um nível de paralelismo comparativamente menor. No entanto, mesmo em vista daqueles que apontam um menor paralelismo, estes resultados sugerem que o desempenho potencial de uma arquitetura super escalar é consideravelmente maior que o de uma arquitetura que não possui a capacidade de executar mais de uma instrução por ciclo.

No entanto, o ponto mais importante por detrás destes resultados não é esta constatação, mas uma pergunta: o que limita o paralelismo de instrução? Quais são

as peculiaridades do código de um programa, mencionadas acima, que impedem um paralelismo de instrução irrestrito? O paralelismo de instrução - e por conseguinte o desempenho de uma arquitetura super escalar - é limitado pelas dependências entre instruções. Existem duas formas de dependências entre instruções, denominadas **dependências de dados** e **dependências de controle**, que são discutidas a seguir.

2.1.1 Dependências de Dados

Como a própria denominação indica, as dependências de dados estão relacionadas com o fluxo de dados entre as instruções do programa [Hennessy 90] [Johnson 91]. Para mostrar os diversos tipos de dependências de dados, considere o trecho de código na Figura 2.1. O mnemônico `op Ra,Rb,Rc` representa a instrução `Ra op Rb` $\rightarrow$ `Rc`.

```
ADD R1,R3,R1
SUB R1,R4,R2
OR  R5,R4,R3
AND R5,R6,R2
```

Figura 2.1: Código com dependências de dados.

Um primeiro tipo de dependência de dados aparece quando uma instrução usa o resultado produzido por uma outra instrução. Por exemplo, na Figura 2.1, a instrução `SUB` depende do resultado da instrução `ADD`. Esta forma de dependência de dados é chamada **dependência verdadeira**, ou dependência RAW (*Read-After-Write*). Como esta última denominação indica, a instrução dependente deve ler os seus operandos somente após a instrução precedente ter escrito o seu resultado. Caso esta ordem não seja respeitada, a instrução dependente usa um valor antigo que existia antes da modificação pela precedente, levando a uma execução incorreta do código.

Durante a execução de um programa, um registrador recebe diferentes valores de uma variável. Em geral, o valor de uma variável não pode ser modificado antes que seja usado, criando assim uma dependência entre as instruções que usam e atualizam a mesma variável. Esta forma de dependência de dados é chamada **anti-dependência**, ou dependência WAR (*Write-After-Read*). Na Figura 2.1, a instrução dependente `OR`

modifica o conteúdo de um registrador referenciado como operando pela instrução `ADD` precedente. Assim, a instrução `OR` deve escrever o seu resultado somente após a instrução `ADD` ter lido os seus operandos. Se esta ordem não for respeitada, a instrução precedente usará indevidamente o novo valor produzido pela instrução dependente, resultando em uma execução incorreta do código.

Um terceiro tipo de dependência de dados ocorre quando um registrador é reusado para armazenar um novo valor de uma variável. O novo valor da variável deve ser escrito no registrador somente após a escrita (e eventual uso) do último valor. Caso contrário, o valor obtido quando a variável é referenciada é um valor antigo. Isto cria uma dependência entre instruções que atualizam um mesmo registrador, e que é denominada **dependência de saída** ou dependência WAW (*Write-After-Write*). Na Figura 2.1, esta forma de dependência aparece entre as instruções `SUB` e `AND`, que alteram o mesmo registrador. A instrução dependente `AND` deve escrever o seu resultado somente após a instrução precedente `SUB` tê-lo feito. Se esta ordem não for respeitada, o valor deixado no registrador compartilhado será aquele produzido pela instrução precedente, o que não está de acordo com a semântica do código.

Nos três casos acima, a execução da instrução dependente está subordinada à execução de uma instrução precedente. Estas instruções devem ser executadas seqüencialmente, na ordem em que aparecem no código. É por este motivo que as dependências de dados limitam o paralelismo de instrução, pois elas impedem que as instruções interdependentes sejam executadas em paralelo. Em se tratando de uma arquitetura super escalar, tal restrição reduz o número médio de instruções executadas em um mesmo ciclo, prejudicando assim o desempenho.

Um exemplo prático dá uma melhor idéia das limitações impostas pelas dependências de dados e seus efeitos sobre o desempenho de uma arquitetura super escalar. A Figura 2.2 mostra um comando de alto nível e o código *assembly* correspondente, gerado sem nenhum tipo de otimização. Nesta figura, são usadas instruções da arquitetura SPARC (ver Capítulo 4 e Apêndice A). A Figura 2.3 mostra a execução destas instruções em uma arquitetura super escalar hipotética, que possui um *pipeline* com quatro estágios: busca, decodificação, execução e escrita de resultado (na figura estão representados apenas os três últimos estágios). Cada estágio do *pipeline* processa até quatro instruções simultaneamente, e a cada ciclo até quatro instruções podem ser despachadas para execução. Instruções são despachadas na ordem em que se encontram no código. Operandos em registradores são acessados no estágio de decodificação, no momento do despacho. Todas as instruções possuem latências de

um ciclo.

```
*bp = (*bp & rm[offset]) | ((cd << offset) & MASK)

cd em R3
offset em R4
bp em R5

 1:  SLL R3,R4,R2
 2:  SETHI high(mask),R1
 3:  OR low(mask),R1,R1
 4:  AND R1,R2,R2
 5:  SETHI high(rm),R1
 6:  OR low(rm),R1,R1
 7:  ADD R1,R4,R1
 8:  LD [R1],R6
 9:  LD [R5],R7
10:  AND R6,R7,R1
11:  OR R1,R2,R1
12:  ST R1,[R5]
```

Figura 2.2: Comando de alto nível e o código *assembly* correspondente.

Por causa das dependências de dados, no máximo duas instruções são executadas em paralelo. Isto representa um desperdício de recursos, pois embora existam quatro unidades funcionais, apenas duas são utilizadas simultaneamente. São necessários 19 ciclos para executar 12 instruções, resultando em um ipc médio de 0,63 instrução/ciclo. Este exemplo mostra como as dependências de dados podem limitar drasticamente o paralelismo de instrução, reduzindo o ipc de uma arquitetura super escalar a um valor típico de uma arquitetura *pipeline* escalar.

Para se ter uma idéia dos efeitos das dependências de dados sobre a execução de um programa real, considere o gráfico mostrado na Figura 2.4. Este gráfico mostra o ganho de desempenho fornecido por uma arquitetura super escalar cobrindo a execução de dez milhões de instruções de quatro programas do conjunto SPEC (Capítulo

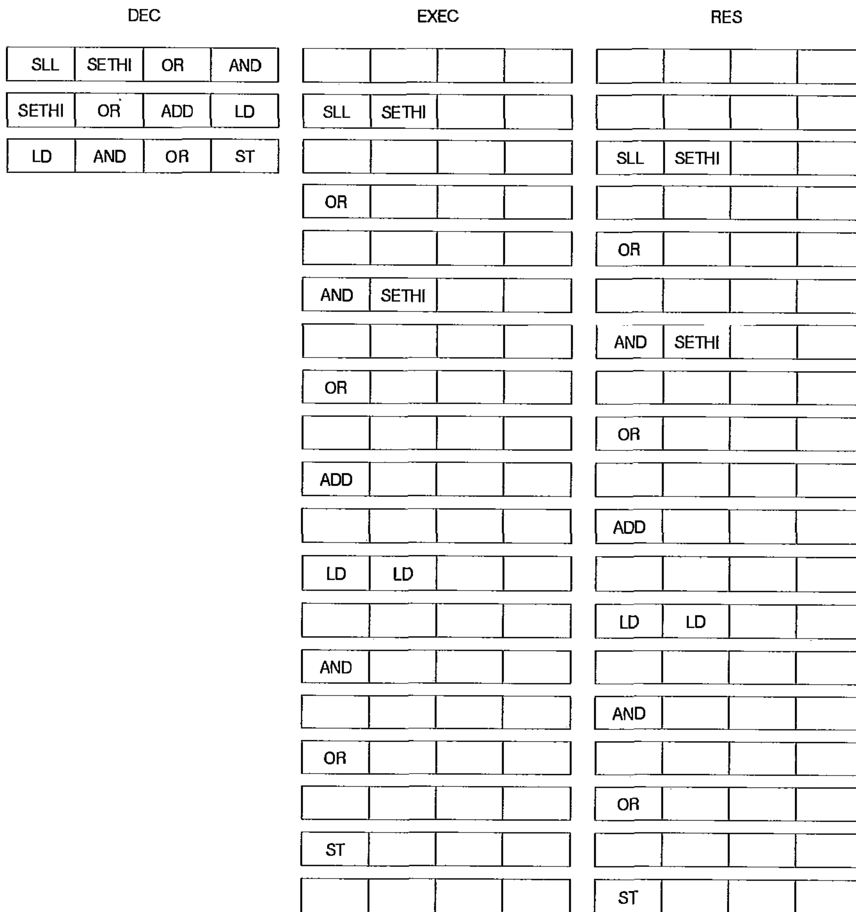


Figura 2.3: Execução de código com dependências de dados em um *pipeline* super escalar.

4). O ganho é apresentado em função do número de unidades funcionais. As curvas foram obtidas com o simulador da arquitetura super escalar descrito no Capítulo 3. Nesta arquitetura, até oito instruções podem ser acessadas, decodificadas e despachadas a cada ciclo. Não existem conflitos pelo uso de recursos, e as memórias *cache* de instruções e de dados possuem taxa de *hit* de 100%. As instruções de desvio são executadas de forma transparente (ver seção seguinte). Nestas condições, apenas as dependências de dados limitam a execução paralela de instruções.

Para alguns programas, as dependências de dados fazem com que o ganho real esteja bem abaixo de um ganho linear (este é o caso dos programas *espresso* e *eqntott*). Em outros programas, as limitações decorrentes das dependências de dados são menos

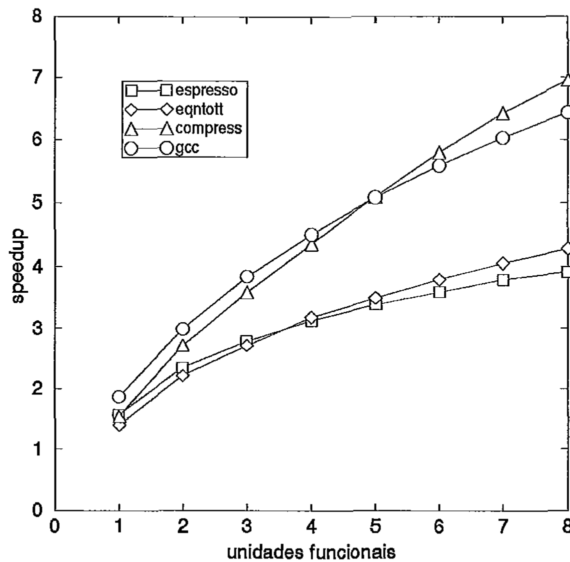


Figura 2.4: Efeito das dependências de dados sobre o desempenho de uma arquitetura super escalar.

severas, e o desempenho torna-se menor que o ideal apenas em configurações com um número grande de unidades funcionais (caso dos programas *compress* e *gcc*). No entanto, na média o ganho chega a ser 33% menor que o ideal, o que indica a importância de minimizar as limitações impostas pelas dependências de dados. Algumas técnicas e mecanismos usados com esta finalidade serão discutidos mais à frente.

Finalizando a discussão sobre dependências de dados, é importante fazer duas observações. Em todos os exemplos acima, dependências de dados ocorrem devido ao uso de um registrador comum. Dependências de dados também podem ocorrer pelo uso de uma mesma posição de memória. Na realidade, dependências através de posições de memória apresentam um tratamento mais difícil do que as dependências através de registradores [Johnson 91]. Isto acontece porque, em alguns casos, o endereço acessado é conhecido apenas em tempo de execução. Desta forma não podem ser aplicadas técnicas de tratamento de dependências em tempo de compilação, obrigando às vezes a adoção de soluções conservativas as quais impedem que instruções de acesso à memória sejam executadas em paralelo ou fora-de-ordem. Os modelos de

arquitetura usados neste trabalho incluem um mecanismo que detecta dinamicamente as dependências entre instruções de acesso à memória (Capítulo 3).

Uma segunda observação é quanto à natureza dos diferentes tipos de dependências de dados. Dependências verdadeiras estão associadas ao fluxo de dados através das operações de um programa, enquanto anti-dependências e dependências de saída aparecem simplesmente porque uma mesma locação de dados (em registrador ou na memória) é reusada para armazenar valores diferentes de uma mesma variável ou variáveis diferentes. Dependências verdadeiras devem ser tratadas mesmo em arquiteturas *pipeline* escalares, onde uma única instrução se encontra no estágio de execução a cada momento [Hennessy 83], [Hennessy 90]. Nestas arquiteturas, anti-dependências e dependências de saída são automaticamente respeitadas, porque as instruções passam pelo estágio de execução na ordem em que se encontram no código do programa. No entanto, em arquiteturas super escalares as instruções podem ser despachadas e executadas em uma ordem diferente daquela em que se encontram no código do programa, e assim torna-se necessário introduzir mecanismos explícitos para o tratamento destes tipos de dependências.

2.1.2 Dependências de Controle

Instruções de transferência de controle (ou simplesmente, instruções de desvio) determinam o fluxo de instruções executadas pelo processador. As próximas instruções a serem executadas após uma instrução de desvio tornam-se conhecidas apenas quando a instrução de desvio é executada. Esta relação entre as instruções que se seguem no fluxo e uma instrução de desvio precedente é denominada **dependência de controle**.

A Figura 2.5 compara o efeito das dependências de controle em uma arquitetura escalar e em uma arquitetura super escalar. A Figura 2.5(a) mostra o estado dos quatro estágios de um *pipeline* escalar em cada ciclo, enquanto a Figura 2.5(b) corresponde a um *pipeline* super escalar onde duas instruções podem ser processadas em cada estágio. Nesta figura, *b* é uma instrução de desvio que transfere o controle para a instrução *d*. As partes hachuradas representam estágios vazios.

No ciclo em que a instrução de desvio é decodificada, o endereço da instrução destino ainda não é conhecido. Se o *pipeline* continuar operando normalmente, as instruções seqüenciais serão acessadas e executadas. No entanto, se o desvio foi tomado, estas instruções não deveriam ser executadas. Para eliminar este risco, a solução mais

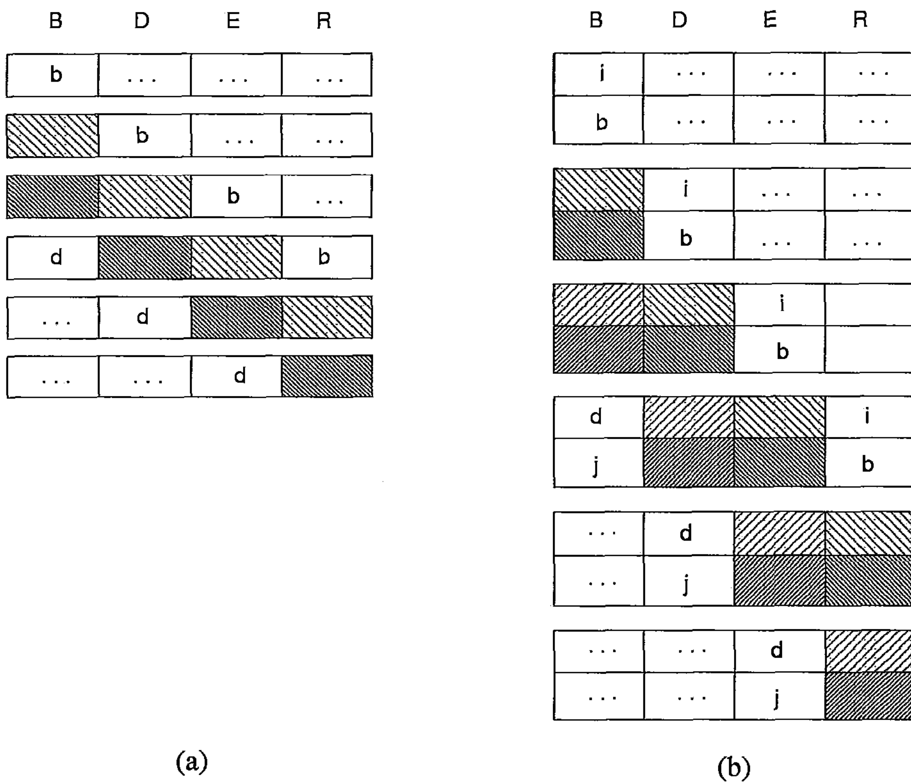


Figura 2.5: Efeito das dependências de controle.

simples consiste em suspender o acesso de novas instruções até que a instrução de desvio seja executada, conforme mostra a Figura 2.5. Nesta figura, a instrução de desvio é executada no ciclo seguinte à sua decodificação, e a instrução no destino de desvio é acessada no próximo ciclo.

O **retardo de desvio** é definido como o intervalo de tempo entre a decodificação da instrução de desvio e a decodificação da instrução destino, enquanto a **penalidade de desvio** é o número de instruções que deixam de ser executadas durante o retardo de desvio [Lilja 88]. No exemplo da Figura 2.5, o retardo de desvio é igual a dois ciclos tanto no *pipeline* escalar como no super escalar. No entanto, no *pipeline* escalar a penalidade de desvio é duas instruções, enquanto no *pipeline* super escalar a penalidade é quatro instruções. Este exemplo mostra que a capacidade da arquitetura super escalar de processar várias instruções simultaneamente acaba por multiplicar os efeitos negativos das dependências de controle. Isto acontece porque uma eventual paralização do *pipeline* impede que um maior número de instruções sejam executadas.

Infelizmente, este efeito é tanto mais severo quanto maior for o grau de paralelismo da arquitetura.

Na seção anterior, foi apresentado um gráfico (Figura 2.4) que mostra o ganho obtido com uma arquitetura super escalar na presença de dependências de dados. Aquele gráfico foi obtido através da simulação de uma arquitetura na qual o resultado de cada desvio é previamente conhecido. Quando uma instrução de desvio é decodificada, o estágio de busca é imediatamente direcionado para acessar as instruções no destino do desvio. Desta forma, as dependências de controle são transparentes, e não introduzem nenhuma penalidade. É interessante agora retirar esta idealização, e verificar qual o efeito das dependências de controle sobre o desempenho real de uma arquitetura super escalar. O gráfico à direita na Figura 2.6 mostra o desempenho quando são acrescentadas as dependências de controle. Este gráfico foi obtido com a simulação de uma arquitetura super escalar na qual as dependências de controle são resolvidas bloqueando-se o despacho de novas instruções quando uma instrução de desvio é encontrada. O despacho é bloqueado até que a instrução de desvio seja executada. A busca e decodificação de instruções continuam durante os ciclos em que o despacho permanece bloqueado. Novamente, foram usados programas do conjunto SPEC, e os valores mostrados no gráfico cobrem a execução de dez milhões de instruções. Para efeito de comparação, à esquerda na figura é repetido o gráfico para o caso onde existem apenas dependências de dados.

Com a presença apenas das dependências de dados, o ganho no desempenho continua a aumentar com o número de unidades funcionais, embora com uma taxa menor que a ideal. No entanto, quando as dependências de controle são introduzidas, a partir de um certo ponto o desempenho permanece estável mesmo quando são acrescentadas unidades funcionais. Estes gráficos mostram que dependências de controle limitam o paralelismo de instrução de uma forma ainda mais severa do que as dependências de dados, e reduzem sensivelmente o desempenho de uma arquitetura super escalar.

Nesta seção foram apresentados os possíveis tipos de dependências entre instruções, e os seus efeitos sobre o paralelismo de instrução e o desempenho de arquiteturas super escalares. Nas seções a seguir são descritas algumas técnicas e mecanismos para o tratamento de dependências de dados e de controle.

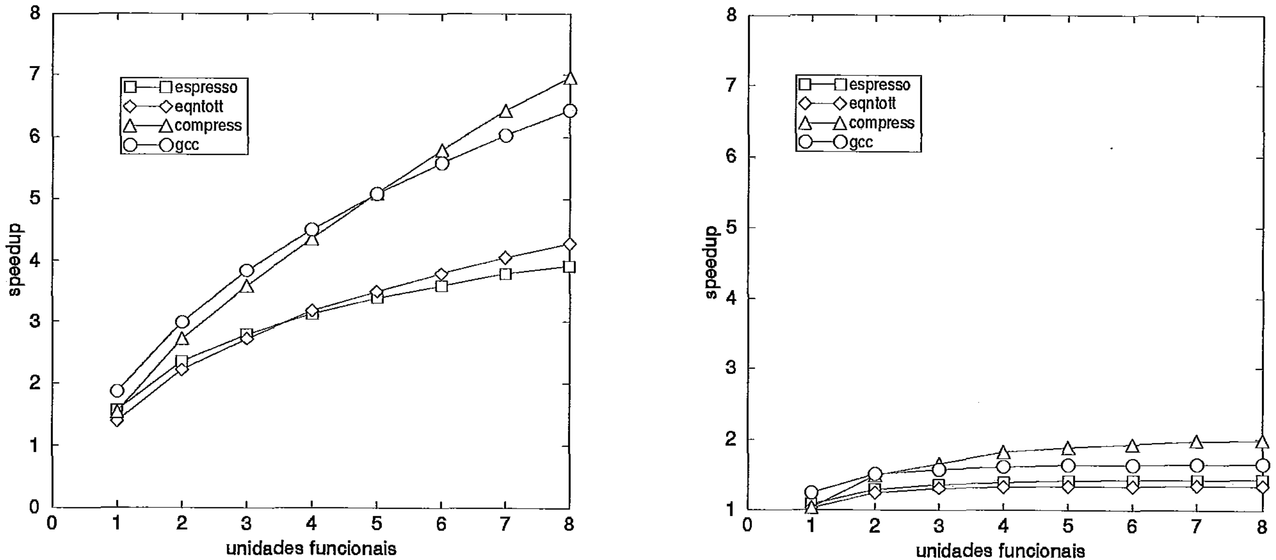


Figura 2.6: Efeito das dependências de controle sobre o desempenho de uma arquitetura super escalar.

2.2 Tratamento das Dependências de Dados

O problema das dependências de dados é tratado com técnicas de escalonamento de instruções [Lam 90]. O escalonamento de instruções consiste em alterar a ordem original das instruções, de modo a possibilitar que um maior número de instruções seja executado em paralelo. Esta reordenação[1] deve ser feita preservando a correção semântica do código e ainda levando em consideração eventuais limitações de recursos na arquitetura.

Como exemplo, a Figura 2.7 mostra o trecho de código apresentado na Figura 2.2 após ser reordenado. A Figura 2.8 mostra a execução do código escalonado no mesmo *pipeline* hipotético considerado na Figura 2.3. Como indica a Figura 2.8, o escalonamento age agrupando instruções independentes que podem ser despachadas e executadas simultaneamente. Com isto aumenta o número de instruções completadas a cada ciclo,

[1] Os termos escalonamento e reordenação serão aqui usados indistintamente.

resultando em uma melhoria do desempenho da arquitetura super escalar. Enquanto a execução do código não-escalonado consome 19 ciclos, para o código escalonado são necessários 17 ciclos. Crawford [Crawford 93] cita que, para o Intel Pentium, a diferença de desempenho entre códigos não-escalonado e escalonado chega a 30%.

```
5:  SETHI high(rm),R1
9:  LD [R5],R7
6:  OR low(rm),R1,R1
7:  ADD R1,R4,R1
8:  LD [R1],R6
2:  SETHI high(mask),R1
1:  SLL R3,R4,R2
3:  OR low(mask),R1,R1
4:  AND R1,R2,R2
10:  AND R6,R7,R1
11:  OR R1,R2,R1
12:  ST R1,[R5]
```

Figura 2.7: Exemplo de código escalonado.

Existem dois momentos onde o escalonamento de instruções pode ser realizado [Chang 91]. No escalonamento estático, a reordenação é feita durante a compilação do programa. No escalonamento dinâmico a reordenação acontece em tempo de execução, sendo realizada por mecanismos incluídos na arquitetura com esta finalidade.

2.2.1 Escalonamento Estático de Instruções

No escalonamento estático, o compilador possui um módulo escalonador, que recebe como entrada a seqüência de instruções fornecida pelo módulo gerador de código e produz como saída o código reordenado [Warren 90]. O escalonamento estático pode ser local ou global. No primeiro caso, a reordenação de instruções acontece dentro dos blocos básicos do programa, enquanto no segundo caso o escopo de reordenação envolve vários blocos básicos.

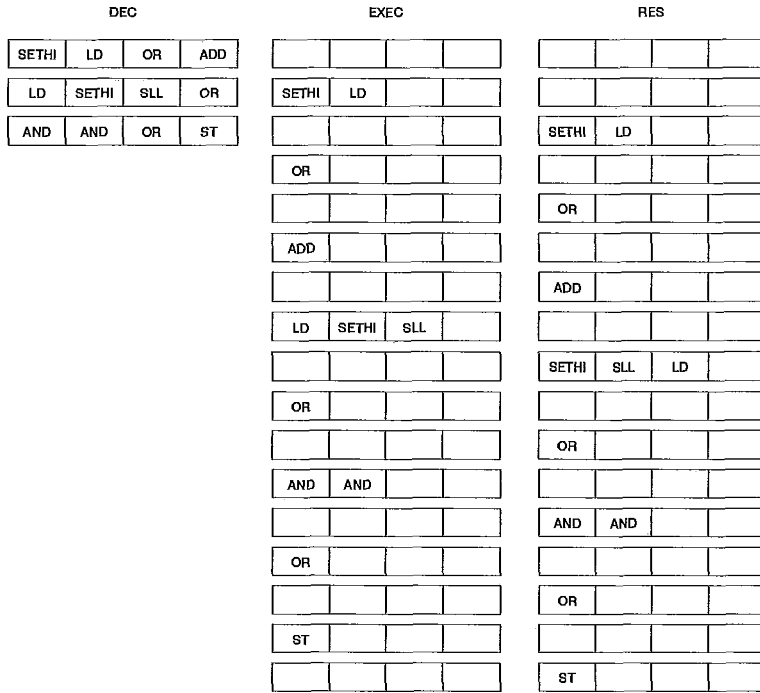


Figura 2.8: Execução do código escalonado.

O problema do escalonamento estático local consiste em obter uma reordenação das instruções que minimize o tempo de execução do bloco básico, sujeito às limitações impostas pelas dependências de dados entre as instruções e pelos recursos disponíveis na arquitetura. Dependências entre as instruções do bloco básico podem ser expressas através de um **grafo de precedência** [Ferrante 87] [Smotherman 91]. Neste grafo, cada nó corresponde a uma instrução do bloco básico e os arcos direcionados indicam as relações de dependência entre as instruções. O grafo de precedência indica uma ordem parcial entre as instruções, que deve ser obedecida durante o escalonamento para que as dependências entre instruções sejam respeitadas.

Algoritmos de escalonamento local já foram extensivamente estudados dentro do contexto de compactação de microcódigo [Davidson 81] [Landskov 80]. Existe uma semelhança entre esta aplicação e o escalonamento de código para arquiteturas super escalares. Nos dois casos, o objetivo do escalonamento é obter uma reordenação que maximize o número de operações que são executadas em paralelo. Por este motivo, algoritmos de compactação de microcódigo podem ser igualmente usados no escalonamento de instruções para arquiteturas super escalares. O *list scheduling* é um

exemplo de algoritmo de escalonamento local [Adam 74].

Prova-se que o problema de escalonamento local é NP-completo [Landskov 80]. Isto significa que algoritmos ótimos, que produzem um código escalonado com o menor tempo de execução possível, apresentam um tempo de computação exponencial com o número de instruções. Algoritmos do tipo *branch-and-bound*, que testam exaustivamente todas as possíveis combinações de escalonamento, garantem que o código escalonado possui o menor tempo de execução. No entanto, o tempo de geração de código torna inviável o uso prático destes algoritmos. O *list scheduling*, mencionado acima, é um algoritmo não-ideal que apresenta o melhor compromisso entre qualidade e tempo de geração de código [Fisher 81]. Na maioria dos casos, este algoritmo produz resultados bem próximos do ótimo ou até mesmo o resultado ótimo.

Como mencionado, no escalonamento estático global a reordenação envolve instruções pertencentes a diferentes blocos básicos. Exemplos de algoritmos de escalonamento global são: *trace scheduling* [Fischer 81], *loop unrolling* [Weiss 87], *software pipelining* [Lam 88], *percolation scheduling* [Forster 72]. Ebcioglu [Ebcioglu 91] e Golumbic [Golumbic 90] discutem o escalonamento global de instruções no contexto de arquiteturas super escalares específicas.

A principal vantagem do escalonamento local é a existência de algoritmos que são relativamente simples e produzem um código quase-ótimo, ou até mesmo o código ótimo em algumas situações. No entanto, obter um tempo de execução ótimo para cada bloco básico individualmente não significa necessariamente que o tempo de execução do programa como um todo é minimizado. A limitação do escopo de escalonamento impede que sejam consideradas várias possibilidades de reordenação de código, que resultariam na execução de um maior número de instruções em paralelo e reduziriam ainda mais o tempo de execução do programa. Devido a esta limitação, técnicas de escalonamento local geralmente são mais ineficientes quando comparadas com técnicas de escalonamento global.

A principal desvantagem no escalonamento global está no fato que a reordenação de instruções deve considerar não somente as limitações impostas pelas dependências de dados, mas também restrições provenientes das dependências de controle. A movimentação de instruções através das fronteiras de blocos básicos, garantindo ao mesmo tempo a correção semântica do código, faz com que os algoritmos de escalonamento global sejam mais complexos, apresentando uma maior dificuldade de implementação e possuindo um maior tempo de geração de código.

2.2.2 Escalonamento Dinâmico de Instruções

Como mencionado, a reordenação de instruções também pode ser feita dinamicamente pelo *hardware*, durante a execução do programa. O escalonamento dinâmico tem o mesmo objetivo do escalonamento estático, qual seja, o de encontrar instruções independentes que possam ser executadas em paralelo.

Mecanismos de escalonamento dinâmico se baseiam em princípios de funcionamento semelhantes. A cada ciclo, são verificadas as dependências entre novas instruções e destas com as que já se encontram em execução. Instruções independentes são executadas imediatamente ou quando os recursos necessários estiverem disponíveis. A execução de instruções dependentes é adiada até que as dependências estejam satisfeitas. Os mecanismos de escalonamento dinâmico diferem basicamente no momento em que a verificação das dependências é realizada. Alguns mecanismos determinam as dependências no despacho, e não enviam para as unidades funcionais as instruções com dependências. Em outros mecanismos, as instruções são despachadas mesmo que existam dependências, mas estas instruções entram de fato em execução somente quando as dependências estiverem satisfeitas.

Como exemplos de mecanismos de escalonamento dinâmico, podemos citar os propostos por Tjaden [Tjaden 70], Acosta [Acosta 86] e Dwyer [Dwyer 92], e ainda mecanismos de fato implementados em sistemas comerciais, tais como o *scoreboarding* e o algoritmo de Tomasulo. Novamente, a descrição de todos estes seria por demais extensa, fugindo ao escopo desta seção. Assim, será aqui examinado apenas o algoritmo de Tomasulo, que será usado nos modelos de arquitetura super escalar avaliados neste trabalho.

O algoritmo de Tomasulo [Tomasulo 67] foi originalmente usado na unidade de ponto-flutuante do IBM 360/91 [Anderson 67]. O processador PowerPC 604 usa uma versão simplificada deste mecanismo. A Figura 2.9 mostra a organização básica de uma arquitetura que implementa o algoritmo de Tomasulo. Nesta figura, supõe-se uma arquitetura com apenas duas unidades funcionais.

A cada unidade funcional estão associadas uma ou mais unidades virtuais (também chamadas estações de reserva). Cada unidade virtual armazena uma instrução já despachada, que se encontra em execução ou que está aguardando algum operando ou a disponibilidade da unidade funcional para entrar em execução. Em cada unidade virtual, o campo OPC armazena o código de operação da instrução. Os campos OP1

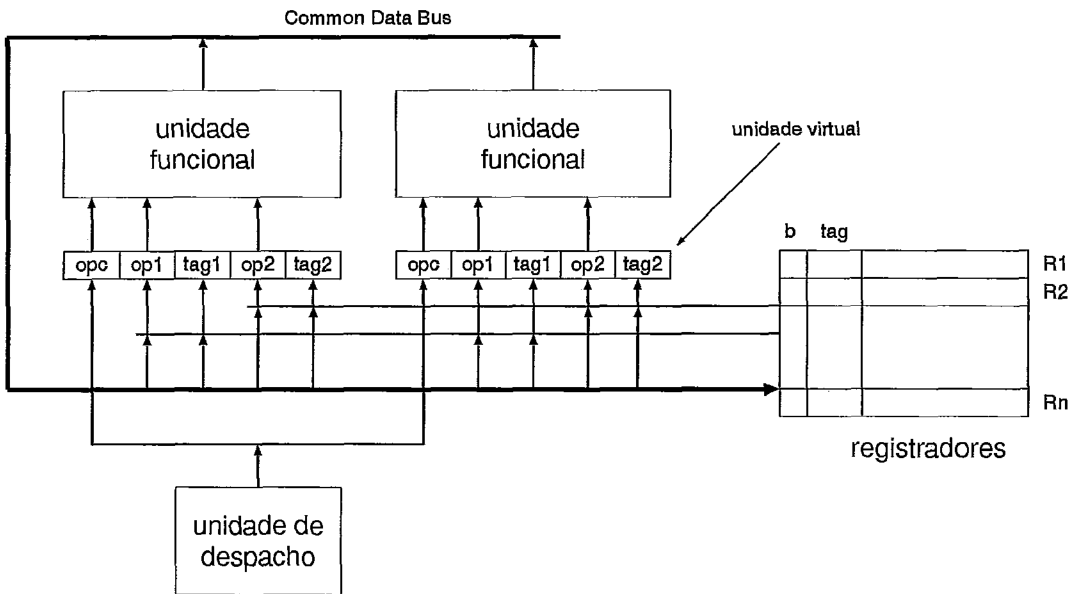


Figura 2.9: Organização de uma arquitetura baseada no algoritmo de Tomasulo.

e OP2 armazenam os operandos da instrução (na figura, supõe-se que as instruções possuem dois operandos). Associados a estes campos existem os campos TAG1 e TAG2, respectivamente, cuja função será explicada mais à frente. Para cada registrador existe um campo TAG e um bit de alocação, denominado B. O *Common Data Bus*, ou CDB, é a via através da qual são transferidos os resultados produzidos pelas unidades funcionais para as unidades virtuais e para o conjunto de registradores.

O mecanismo opera da seguinte forma. No despacho de uma instrução, é alocada uma unidade virtual associada à unidade funcional que executa a instrução. Caso não haja uma unidade virtual disponível, o despacho é bloqueado. O bit de alocação de cada registrador fonte especificado na instrução é testado. Se o bit de alocação não estiver ativo, o registrador correspondente possui um dado válido, que é então copiado para um dos campos de operando da unidade virtual alocada. Se o bit de alocação estiver ativo, isto significa que o registrador foi alocado como destino por uma instrução despachada anteriormente, mas que ainda não terminou. Neste caso, o campo TAG associado ao registrador contém a identificação da unidade virtual que armazena tal instrução. Esta identificação é copiada para o campo TAG apropriado da unidade virtual. Com isto, fica indicado na unidade virtual que o operando associado não se encontra disponível e também a unidade virtual com a instrução que

produzirá este operando. Completando o procedimento de despacho, o bit de alocação do registrador destino especificado na instrução é ativado, e o campo TAG correspondente é atualizado com a identificação da unidade virtual para onde a instrução foi despachada.

Uma instrução pode entrar em execução quando todos os seus operandos estiverem disponíveis na unidade virtual, não importando a sua ordem de chegada em relação às demais instruções despachadas para a mesma unidade funcional. Quando uma instrução é executada, o resultado produzido e a identificação da unidade virtual onde ela se encontra são comunicados através do CDB para todas as unidades virtuais e para o conjunto de registradores. As unidades virtuais comparam a identificação da unidade virtual presente no CDB com os valores em seus campos TAG. No caso de coincidência, isto significa que a instrução armazenada na unidade virtual usa o resultado presente no CDB. A unidade virtual armazena então o resultado no campo de operando associado ao *tag* coincidente, e muda este *tag* para indicar a disponibilidade do operando. A identificação da unidade virtual no CDB também é comparado com os *tags* associados aos registradores. O resultado é armazenado no registrador com um *tag* coincidente, e o seu bit de alocação é desativado para indicar a disponibilidade do resultado. Por último, a unidade virtual que tem sua instrução executada e o resultado comunicado pelo CDB é liberada.

Em uma implementação real, a comparação da identificação da unidade virtual presente no CDB com os *tags* nas unidades virtuais e no conjunto de registradores é feita usando associatividade, de modo que a identificação no CDB seja comparada simultaneamente com todos os *tags*. A comparação por associatividade é usada no lugar de uma comparação seqüencial para não aumentar o ciclo do processador.

É importante verificar como o algoritmo de Tomasulo resolve dependências de dados. Dependências verdadeiras são respeitadas porque uma instrução em uma unidade virtual não é executada enquanto algum de seus operandos não estiver disponível. A instrução torna-se apta para execução somente quando o operando é produzido e armazenado na unidade virtual. Considere agora a seqüência de instruções ... i_j ... i_k ..., onde existe uma anti-dependência entre i_j e i_k. Quando i_j é despachada, os valores em seus registradores fonte são copiados para a unidade virtual. Com isto a anti-dependência é respeitada porque, mesmo que i_k seja executada antes de i_j e altere algum destes registradores, uma cópia do valor original do operando já estará armazenado na unidade virtual. Considere agora que existe uma dependência de saída entre i_j e i_k. Quando i_j é despachada, o *tag* do seu registrador destino recebe

o número da unidade virtual para onde a instrução foi enviada. Quando i_k é depois despachada, este mesmo *tag* recebe o número de uma outra unidade virtual. Se i_j é executada depois de i_k, o resultado não é armazenado no registrador destino porque o *tag* atribuído por i_j foi reescrito com o *tag* atribuído por i_k. Desta forma, somente o resultado de i_k é armazenado no registrador, como exigido pela dependência de saída.

Weiss [Weiss 84] comparara a eficiência do *scoreboarding* e do algoritmo de Tomasulo usando como referência o esquema de escalonamento de instruções do Cray-1. No Cray-1, as dependências são verificadas no momento do despacho, sendo resolvidas com o bloqueio do despacho. Foram usados simuladores da arquitetura do Cray-1, com o mecanismo de escalonamento original sendo substituído pelo *scoreboarding* e pelo algoritmo de Tomasulo. Com o *scoreboarding*, foi obtido um ganho médio de 1,28 em relação modo de escalonamento original do Cray-1, enquanto o ganho médio com o algoritmo de Tomasulo foi 1,58. Weiss aponta como principal fator para o melhor desempenho do algoritmo de Tomasulo a sua melhor eficiência em resolver anti-dependências e dependências de saída.

Na discussão sobre escalonamento estático, foi visto que a reordenação de instruções pode acontecer apenas dentro das fronteiras dos blocos básicos, ou pode abranger diversos blocos básicos. O escalonamento dinâmico também pode ser feito a nível local ou global.

Segundo a definição de bloco básico vista anteriormente, o término de um bloco é marcado por uma instrução de desvio, que transfere o controle de execução para o início de um outro bloco. Considere agora um mecanismo de despacho que ao encontrar uma instrução de desvio bloqueia o despacho de novas instruções até que o desvio seja executado. Neste modo de funcionamento o escalonamento é local, porque o despacho não prossegue normalmente através da fronteira de um bloco básico. O desvio ao final do bloco básico interrompe o fornecimento de instruções, e o escalonamento continua apenas sobre as instruções do bloco básico corrente. As instruções do próximo bloco básico começam a ser escalonadas quando o despacho é reiniciado, somente após a execução do desvio no final do bloco anterior.

Ao contrário, considere agora que o despacho não é bloqueado quando uma instrução de desvio é encontrada, mas apenas quando não existem recursos (p. ex, unidades funcionais) disponíveis. Neste caso o escalonamento é global, porque o despacho continua normalmente quando a fronteira de um bloco básico é alcançada. O conjunto das instruções sendo escalonadas é formado tanto por instruções do bloco corrente como

por instruções pertencentes ao próximo bloco (e eventualmente dos blocos seguintes, dependendo da disponibilidade de recursos).

O escalonamento dinâmico global introduz um problema. Note que agora o despacho de instruções continua mesmo quando a instrução de desvio não foi ainda executada. Considere um desvio condicional: através de qual dos dois possíveis ramos do desvio o despacho deve prosseguir, dado que a condição de desvio ainda não foi avaliada e o destino do desvio ainda não é conhecido? É possível fazer uma previsão do resultado do desvio, mas as instruções despachadas no ramo previsto não podem alterar o estado da arquitetura até que a instrução de desvio seja avaliada e possa ser determinado se a previsão foi correta ou não.

Isto mostra que o escalonamento dinâmico global exige a inclusão na arquitetura de dois mecanismos de suporte. Primeiro, um mecanismo eficiente de previsão dinâmica de desvios, que possua uma boa taxa de acerto na previsão do resultado de um desvio condicional. Segundo, um mecanismo de execução especulativa de instruções, que permita a recuperação de um estado anterior da arquitetura caso seja determinado que algumas instruções foram executadas indevidamente. Previsão dinâmica de desvios e execução especulativa de instruções serão descritos na discussão sobre tratamento de dependências de controle.

2.2.3 Escalonamento Estático vs. Dinâmico

O escalonamento estático apresenta como principal vantagem o fato de não exigir suporte de *hardware*. A conseqüente simplificação na implementação da arquitetura pode resultar em ganhos no desempenho, através de uma redução no tamanho do ciclo do processador. As desvantagens apontadas no escalonamento estático são o aumento no tempo de geração de código e, em algumas técnicas, um aumento no tamanho do código (o caso, por exemplo, do *trace scheduling*, que requer a adição de código de reparo).

A principal vantagem no escalonamento dinâmico é sua capacidade de realizar algumas otimizações que não podem ser feitas em tempo de compilação [Chang 91], como por exemplo:

- execução fora-de-ordem de instruções *load* (*load bypassing*): instruções *load* carregam valores que serão usados por outras instruções. Assim, estas instruções normalmente

se encontram no caminho crítico de um bloco básico, e o seu escalonamento é prioritário. No entanto, é comum que o escalonamento de um *load* altere a sua ordem em relação a uma instrução *store*. Quando isto acontece, o escalonamento estático nem sempre é possível, porque os endereços das locações de memória acessadas podem não ser conhecidos no momento da compilação, e portanto não há como saber se uma mudança de ordem ofende alguma relação de precedência. Em tempo de execução os endereços das locações de memória são conhecidos, e o escalonador dinâmico pode determinar se as instruções de acesso referenciam a mesma posição de memória (esta verificação é chamada *memory disambiguation*). Se a instrução *load* for independente, o escalonador dinâmico pode despachá-la para execução antes da instrução *store*;

- tolerância a falhas na memória *cache*: quando uma instrução *load* ou *store* resulta em uma falha na memória *cache*, o escalador dinâmico pode despachar fora-de-ordem outras instruções no caminho crítico, otimização esta que não pode ser realizada em tempo de compilação. Desta forma, latências de falhas na *cache* podem se tornar transparentes, o que diminui a sensibilidade do desempenho em relação à memória *cache*;

O uso de técnicas de escalonamento estático ou dinâmico não é excludente. É possível um compilador que faça um escalonamento estático, enquanto um mecanismo embutido na arquitetura explora oportunidades para um escalonamento complementar que não podem ser capturadas em tempo de compilação. Este trabalho concentrou-se apenas no escalonamento dinâmico de instruções porque compiladores com otimização de código para arquiteturas super escalares são se encontravam localmente disponíveis.

2.3 Tratamento de Dependências de Controle

As dependências de controle e seus efeitos sobre o desempenho de uma arquitetura super escalar foram discutidas em 2.1.2. Em uma situação extrema, as dependências de controle interrompem a busca de novas instruções até que a instrução de desvio seja executada. Isto reduz a taxa de busca de instruções (número de ins-truções acessadas por ciclo), afetando o balanceamento da arquitetura. O potencial de uma arquitetura super escalar é efetivamente explorado somente quando o forne-cimento de instruções é suficiente para manter as unidades funcionais ocupadas. Se a taxa de busca for menor que a taxa de execução potencial, o desempenho fica, simplesmente, limitado pelo acesso às instruções. Para minimizar as conseqüências das dependências

de controle, é necessário que a busca de novas instruções não seja interrompida quando uma instrução de desvio é encontrada.

O mecanismo de desvios atrasados (*delayed branches*) [McFarling 86] [Lilja 88] é comumente empregado em arquiteturas *pipelined* escalares para reduzir o efeito das dependências de controle. Nesta técnica, as n instruções após uma instrução de desvio são sempre acessadas e executadas, qualquer que seja o eventual resultado do desvio. Estas instruções ocupam as **posições de atraso** (*delayed slots*). As instruções nas posições de atraso mascaram o retardo de desvio, pois a busca destas instruções prossegue enquanto o resultado do desvio não é determinado. O compilador efetua um escalonamento de instruções, preenchendo as posições de atraso com instruções que possam ser executadas independentemente do resultado do desvio. Quando uma tal instrução não pode ser encontrada, instruções `NOP` são usadas para preencher as posições de atraso. Em [Hennessy 90] são discutidas as possíveis estratégias para o preenchimento das posições de atraso. É importante observar que esta solução é uma forma de tratar as dependências de controle através de um escalonamento estático de instruções.

O mecanismo de desvios atrasados é conceitualmente simples, e de fácil implementação. Infelizmente, este mecanismo não apresenta uma boa escalabilidade, limitando a sua eficácia em arquiteturas super escalares [Sites 93]. Note que em uma arquitetura escalar, onde o *pipeline* acessa apenas uma instrução a cada ciclo, a instrução de desvio e as instruções nas posições de atraso são acessadas em ciclos diferentes. Ao contrário, em um *pipeline* super escalar múltiplas instruções são acessadas em um mesmo ciclo. As instruções nas posições de atraso, que antes eram acessadas em ciclos distintos, são agora acessadas juntamente com a instrução de desvio, e assim deixam de mascarar o retardo de desvio. Torna-se então necessário aumentar o número de posições de atraso, provendo até d x l posições adicionais, onde d é o retardo de desvio e l é o número de instruções acessadas. Este aumento no número de posições de atraso dificulta o preenchimento com instruções úteis, aumentando o uso de instruções `NOP` que consomem ciclos e nào contribuem para a execução do programa. Além disso, aumenta a possibilidade de interferência entre o escalonamento realizado para o preenchimento das posições de atraso e o efetuado para contornar as dependências de dados. Por estes motivos, desvios atrasados não são empregados em arquiteturas super escalares.

Uma outra técnica de tratamento de dependências de controle é denominada **previsão de desvios**. Observe que, para reduzir os efeitos das dependências de controle, é ne-

cessário permitir que a busca de instruções continue na presença de desvios. Para um desvio condicional, o fluxo de controle prossegue através da instrução seqüencial quando o desvio é não-tomado, ou a partir de alguma outra instrução quando o desvio é tomado. A interrupção da busca acontece porque não é possível saber, *a priori*, através de qual destes possíveis ramos o fluxo de controle prosseguirá. No entanto, ao invés de simplesmente suspender a busca, pode ser mais vantajoso prever o resultado do desvio e continuar acessando instruções através do caminho previsto [Smith 81] [Lee 84]. Se eventualmente a previsão estiver errada, as instruções acessadas indevidamente devem ser descartadas e a busca deve ser redirecionada para o destino correto. Neste caso, o desvio continua a introduzir um custo. No entanto, se a previsão for correta, a busca pode prosseguir normalmente e o custo do desvio terá sido efetivamente anulado. A eficácia deste método depende, basicamente, da freqüência de acerto das previsões.

A previsão do resultado de um desvio pode ser feita estática ou dinamicamente. Estas duas formas de previsão são discutidas a seguir.

2.3.1 Previsão Estática de Desvios

Na previsão estática, o resultado previsto para um certo desvio é sempre o mesmo, em qualquer execução deste desvio. Este tipo de previsão pode ser realizado a nível de *software*, durante a compilação do programa, ou em tempo de execução, pelo *hardware*. Na previsão estática em *software*, o compilador insere no código da instrução de desvio uma indicação sobre o provável resultado do desvio. Quando o desvio é decodificado, esta informação é verificada e a busca prossegue através do ramo indicado. Para prever o resultado do desvio, o compilador pode se basear no contexto em que se encontra o desvio: por exemplo, desvios que fecham *loops* possuem uma grande de chance de serem tomados. O compilador pode também usar o histórico de resultados anteriores do desvio, anotado a partir de execuções do programa para conjuntos representativos de dados de entrada.

Na previsão estática em *hardware*, a previsão é feita a partir da interpretação do código da instrução. A diferença em relação à previsão em *software* é que, neste caso, não existe nenhuma informação explícita no código usada na previsão. Por exemplo, nas instruções de desvio relativas ao contador de programa, a previsão pode se basear no sinal do deslocamento: desvios com deslocamento negativo, que transferem o controle para uma instrução anterior (provavelmente a instrução no início de um

loop), são previstos como tomados, enquanto desvios com deslocamento positivo são considerados como não-tomados. A arquitetura DEC Alpha AXP e os processadores PowerPC 601 e PowerPC 603 [Burgess 94] incluem uma previsão estática por *hardware* baseada no sentido do desvio (indicado pelo sinal do deslocamento). A arquitetura Alpha inclui ainda outras formas de previsão estática [Sites 93].

2.3.2 Previsão Dinâmica de Desvios

A previsão dinâmica de desvios ocorre em tempo de execução, sendo realizada apenas a nível de *hardware*. Ao contrário do caso estático, onde a previsão de um desvio é fixa em todas as suas execuções, agora a previsão é feita com base no histórico de execuções anteriores do desvio, podendo ser modificada nas execuções futuras. Como visto no Capítulo 1, mecanismos de previsão dinâmica de desvios são hoje encontrados em diversos processadores super escalares. Lee & Smith [Lee 84] descrevem e comparam alguns mecanismos de previsão dinâmica de desvios. Uma outra descrição também pode ser encontrada em [Hennessy 90]. Cragon [Cragon 92] usa modelos matemáticos para avaliar o desempenho de diferentes mecanismos de previsão dinâmica.

Um mecanismo bem simples de previsão dinâmica é aquele que usa uma tabela de história de desvios, ou BHT (*Branch History Table*). Esta tabela é implementada sob a forma de uma pequena memória (normalmente integrada com o processador em um mesmo dispositivo), onde cada entrada (locação) armazena um ou mais bits usados para registrar a história dos desvios. O diagrama na Figura 2.10 resume a operação do mecanismo de previsão com BHT, salientando os eventos que ocorrem em cada estágio de um *pipeline* típico.

Quando uma instrução é acessada, ela é pré-decodificada para que seja determinado se aquela é uma instrução de desvio. Se este for o caso, os bits menos significativos do endereço da instrução são usados para indexar a BHT. O bit na entrada selecionada fornece a previsão do desvio: por exemplo, um bit 0 indicaria que o desvio deve ser previsto como não-tomado, enquanto um bit 1 indicaria previsão de desvio 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. Se este for o caso, a execução prossegue normalmente, caso contrário as instruções acessadas indevidamente são descartadas e a busca é redirecionada para o destino correto.

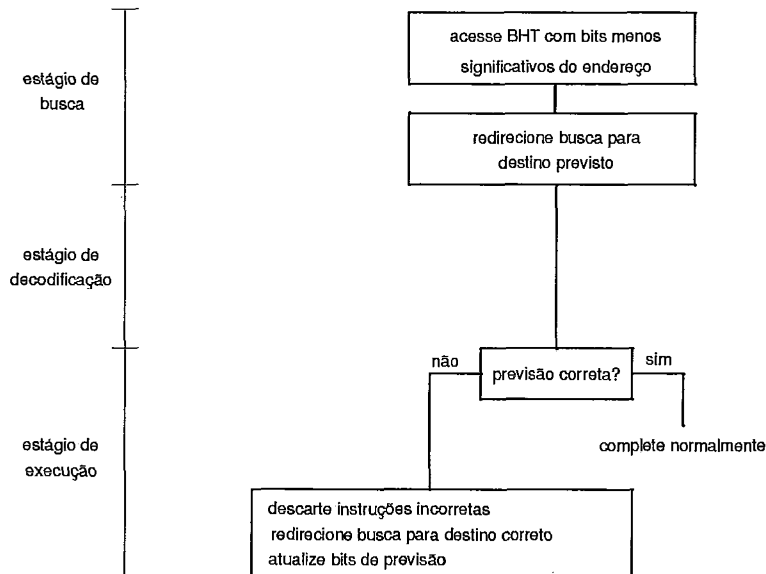


Figura 2.10: Funcionamento da previsão dinâmica de desvios com BHT.

No caso mais simples, cada entrada na BHT contém um único bit, o que permite o registro de apenas dois estados: um dos estados, denominado N, indica que o desvio foi não-tomado em sua última execução e que a previsão será não-tomado na execução corrente; o outro estado, denominado T, indica que o desvio foi tomado na execução anterior e que será previsto como também tomado na execução corrente. A Figura 2.11 mostra o diagrama de transição de estados que indica como é determinado se a previsão foi correta e como o bit de previsão é alterado.

Neste diagrama, o rótulo em cada arco indica o resultado da execução atual do desvio (N para não-tomado e T para tomado) e o resultado da previsão (correta, incorreta). Cada arco leva ao novo estado do bit de previsão. Por exemplo, se o estado atual do bit é N e o resultado do desvio é T, isto significa que a previsão foi incorreta e o bit de previsão é comutado para o estado T.

O principal problema na previsão dinâmica com BHT é a ocorrência de colisões. Considere duas instruções de desvio em endereços diferentes, mas cujos bits menos significativos coincidem. Estes dois desvios selecionam a mesma entrada na BHT, e assim a informação de previsão de um desvio é usada na previsão do outro desvio, reduzindo a taxa de acerto na previsão de ambos os desvios.

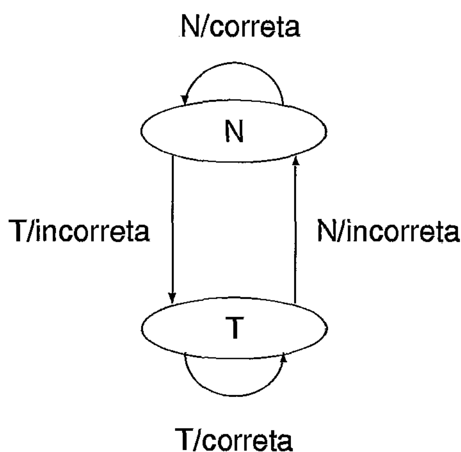


Figura 2.11: Diagrama de transição de estados para um único bit de previsão.

Um problema semelhante ocorre em memórias *cache* com mapeamento direto [Smith 82]. No caso de memórias *cache*, usa-se acesso associativo puro ou por conjunto para reduzir a ocorrência de colisões. De maneira semelhante, a BHT pode ser modificada de forma que cada entrada da BHT possua o endereço de uma instrução, juntamente com os bits de previsão. Na busca da instrução, o endereço completo da instrução acessada é comparado associativamente com os endereços armazenados na BHT. Caso ocorra um *hit*, são usados os bits de previsão na entrada onde está armazenado o endereço coincidente. Supondo que o código não é auto-modificável, existe uma correspondência fixa entre endereço e instrução, e assim a informação de previsão selecionada estará sempre associada a uma mesma instrução de desvio. Se no acesso à BHT ocorrer um *miss*, é feita uma previsão estática do desvio para determinar qual a instrução acessada no próximo ciclo. Se após a decodificação for verificado que a instrução que ocasionou o *miss* é um desvio, o seu endereço e o resultado da execução serão inseridos na BHT. O diagrama na Figura 2.12 mostra as operações que ocorrem em cada estágio de um *pipeline* na previsão com BHT associativa.

Um aperfeiçoamento deste mecanismo consiste em armazenar em cada entrada o endereço da instrução destino obtido na última execução de um desvio. Com isto, o *hardware* necessário para determinar o endereço destino de um desvio é incluído apenas no estágio de execução do *pipeline*. Este dispositivo modificado é chamado tabela de destinos de desvios, ou BTB (*Branch Target Buffer*). Cada entrada no BTB

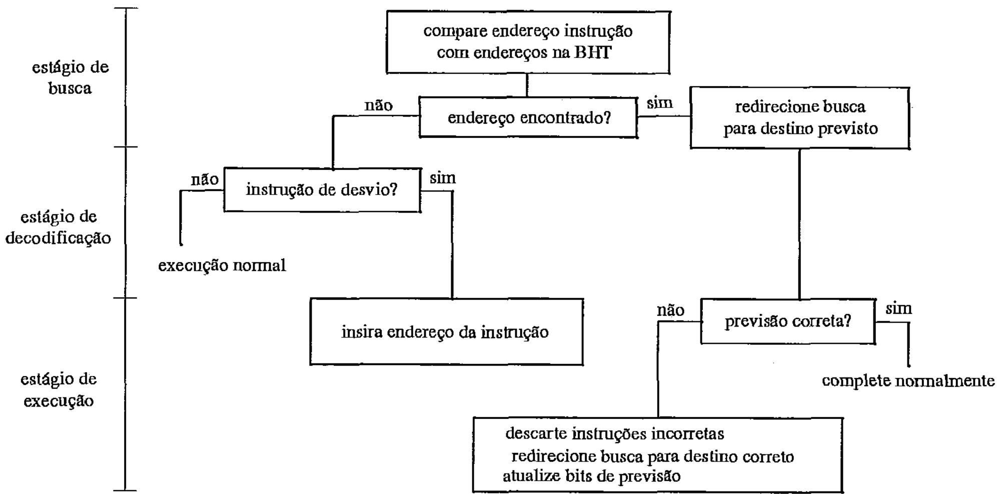


Figura 2.12: Operação de uma BHT associativa.

possui o endereço de uma instrução de desvio, o endereço de destino (previsto) do desvio, e os bits de previsão. A operação é semelhante a 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. A Figura 2.13 mostra um diagrama para a previsão com BTB.

2.3.3 Eficácia da Previsão de Desvios

Como frisado anteriormente, para que a técnica de previsão de desvios seja realmente eficaz, é necessário que a maioria das previsões sejam corretas. Em uma BHT associativa ou em uma BTB, a taxa acertos depende de dois fatores: primeiro, da freqüência com que as informações de previsão são encontradas na tabela, e segundo da freqüência de acertos na previsão de cada desvio. Ou seja:

$$\text{taxa de acerto} = \text{taxa de } \textit{hit} \text{ na tabela de previsão} \times \text{taxa de acerto individual}$$

O primeiro fator depende da configuração da tabela de previsão, p. ex., do número de entradas. A taxa de acerto individual depende em parte do número de bits de

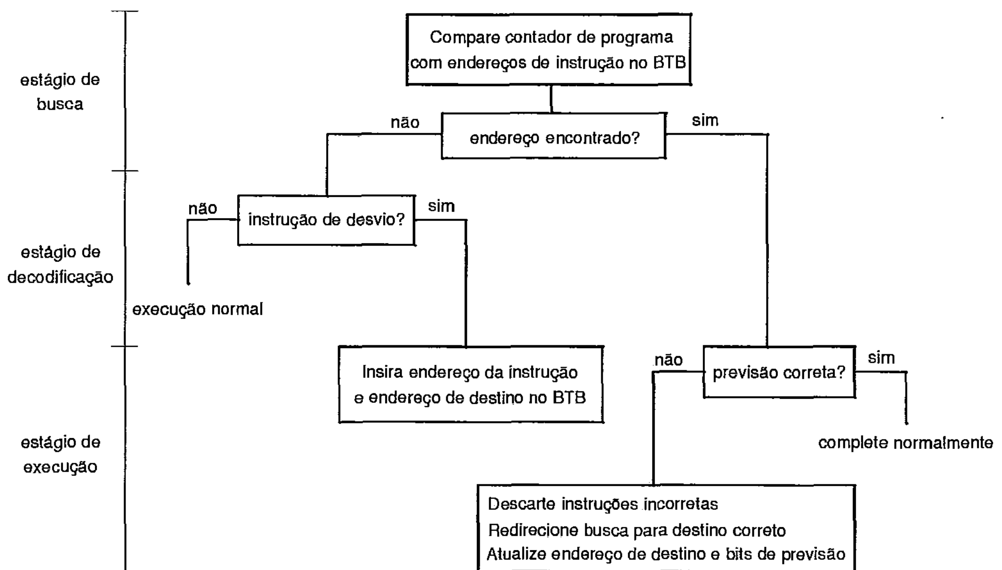


Figura 2.13: Operação de uma BTB.

previsão. Com um maior número de bits é possível registrar a história dos desvios a partir de um passado mais distante, tornando a previsão mais imune a erros. Para tornar mais claro este ponto, considere um desvio que na maioria das vezes transfere o controle para um mesmo ramo, mas que eventualmente faz uma transferência de controle para o ramo alternativo. Este comportamento é comum em *loops* aninhados como o que aparece na Figura 2.14.

```
for (i = 0; i < 10; i++) {
  for (j = 0; j < 10; j++) {
    ...
  }
}
```

Figura 2.14: Aninhamento de *loops* que provoca erros de previsão de desvios.

Com um único bit, a previsão do desvio que fecha o *loop* interno é incorreta a cada última iteração (não-tomado, mas previsto como tomado) e a cada primeira iteração

(tomado, mas previsto como não-tomado). A taxa de acerto é de 80%, embora em 90% das execuções o desvio tenha o mesmo comportamento. Em um mecanismo de previsão com dois bits de previsão, é possível registrar o resultado das duas últimas execuções, e a próxima previsão é modificada apenas se as duas últimas previsões foram incorretas. No exemplo acima, a previsão com dois bits será incorreta apenas na última iteração do *loop* interno, obtendo-se a melhor taxa de acerto possível.

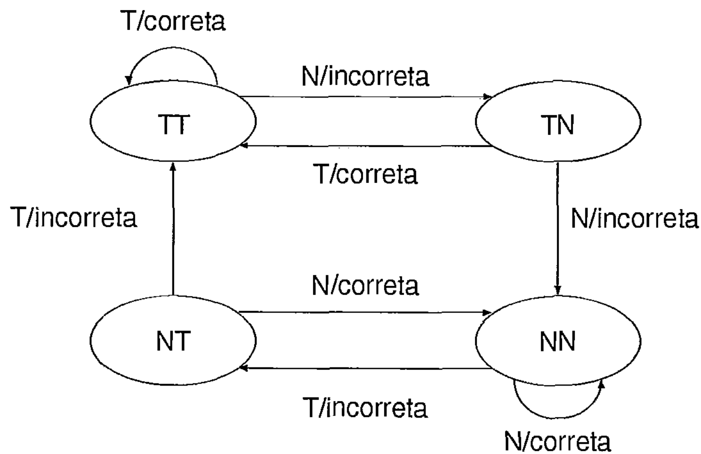


Figura 2.15: Diagrama de transição de estados para dois bit de previsão.

A Figura 2.15 mostra como é feita a previsão com dois bits. 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 resultado mais antigo. Informalmente, pode-se dizer que é necessário uma confirmação da indicação fornecida pelo bit mais recente, com dois resultados consecutivos diferentes do previsto, para que a previsão seja alterada.

Estudos como os realizados por Lee & Smith [Lee 84] mostram que, com dois bits de previsão, é possível alcançar uma taxa média de acerto individual de 90%. Com uma taxa média de *hit* de 95% na tabela de previsão, isto significa uma taxa média de acerto de 85%. Na prática, são reportadas taxas médias de acerto de 75% para o BHT no Intel Pentium [Alpert 93].

2.4 Execução Especulativa

É essencial observar que a previsão de desvios possibilita apenas a continuidade da busca de instruções na presença de dependências de controle. Estes mecanismos não atacam a outra parte do problema, qual seja, o fato das dependências de controle tornarem a execução dependente do resultado de um desvio condicional. Uma instrução de desvio condicional deve ser antes executada para que seja determinado se as instruções acessadas antecipadamente podem, de fato, ser executadas.

Na abordagem mais simples, dependências de controle são satisfeitas bloqueando-se o despacho de novas instruções até que o desvio seja executado. Em uma solução menos restritiva, as instruções acessadas após o desvio são despachadas e executadas tentativamente, não alterando o estado da arquitetura. Se após a execução do desvio for verificado que tais instruções pertencem ao ramo correto, é permitido que os seus resultados modifiquem o estado da arquitetura. Caso contrário, os resultados são descartados e a execução é reiniciada com as instruções no ramo correto. Esta solução é denominada execução especulativa de instruções. É importante observar que a execução especulativa encontra-se associada a um mecanismo de previsão de desvios, pois é necessário que a busca de instruções continue durante a execução do desvio. Além disso, o mecanismo de previsão deve ser eficiente, de modo que na maioria dos casos a execução especulativa de instruções seja de fato válida.

A idéia de execução especulativa requer um mecanismo para manter um estado temporário criado pelas instruções executadas tentativamente, ou que possibilite a recuperação de um estado que existia anteriormente à execução destas instruções. Aqui serão descritos mecanismos que foram originalmente concebidos para suportar interrupções precisas em arquiteturas com execução de instruções fora-de-ordem [Smith 88] [Wang 93]. Estes mecanismos podem ser igualmente usados para suportar execução especulativa de instruções.

2.4.1 O Buffer de Reordenação

O *buffer* de reordenação (*reorder buffer*) é um mecanismo usado para manter o estado temporário criado pelas instruções executadas especulativamente. A Figura 2.16(a) mostra a organização do *buffer* (de reordenação), enquanto a Figura 2.16(b) mostra a estrutura de uma arquitetura com este mecanismo.

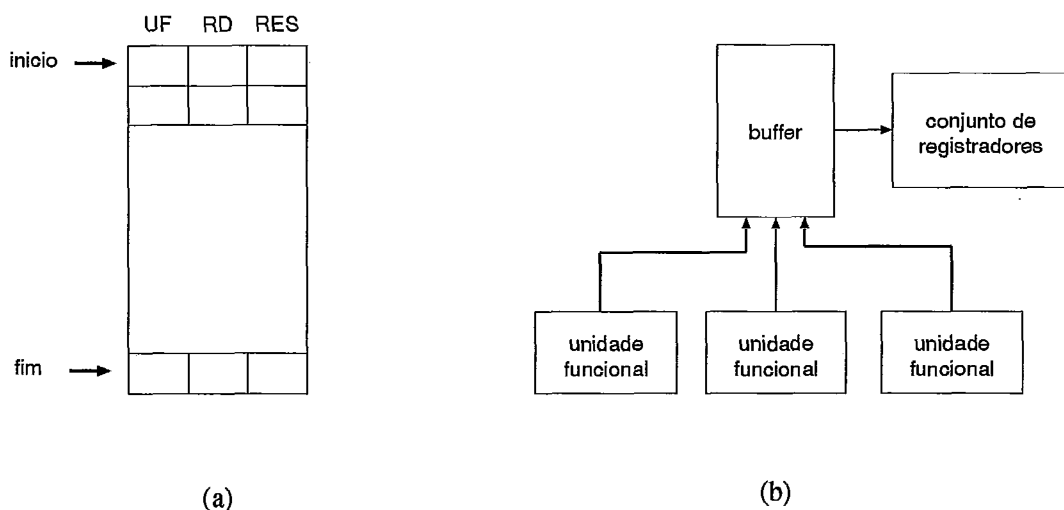


Figura 2.16: O *buffer* de reordenação.

Em cada posição do *buffer* são armazenadas três informações relativas a uma certa instrução: a identificação da unidade funcional para a qual a instrução foi despachada (campo UF), a identificação do registrador destino da instrução (RD) e o resultado produzido pela instrução (RES). O termo "entrada" será aqui usado para referir-se às informações associadas a uma instrução que estão armazenadas em uma posição do *buffer*. O *buffer* de reordenação é gerenciado como uma fila. Uma nova entrada é inserida após a última entrada no *buffer*, enquanto uma entrada é retirada apenas quando ela se encontra na primeira posição do *buffer*. As entradas avançam para a posição seguinte do *buffer* à medida que entradas são retiradas.

Este mecanismo opera da seguinte forma. No despacho de uma instrução, a identificação da unidade funcional que executará a instrução e o número do registrador destino são inseridos no *buffer*. Após a execução da instrução, o resultado é armazenado na entrada correspondente no *buffer*. O acesso em fila vale apenas para a inserção e retirada de entradas. O resultado pode ser armazenado tão logo a instrução seja completada, qualquer que seja a posição da entrada dentro do *buffer*.

Quando uma entrada atinge o início do *buffer* e a instrução correspondente já terminou, o resultado ali armazenado é copiado para o registrador destino indicado, e a entrada é retirada do *buffer*. Neste momento, já foram completadas todas as instruções que foram despachadas antes da instrução correspondente à entrada no início

do *buffer*. Isto acontece porque entradas são inseridas e retiradas na mesma ordem do despacho, e também porque uma entrada é retirada somente quando a instrução correspondente é executada. Assim o *buffer* de reordenação, como o próprio nome indica, age reordenando a seqüência de modificação dos registradores: embora instruções sejam executadas fora-de-ordem, os resultados modificam os registradores de acordo com a ordem de despacho.

Vamos agora examinar como o *buffer* de reordenação é usado para permitir execução especulativa. Quando uma instrução de desvio é despachada, uma entrada é inserida no *buffer* e o despacho de instruções pertencentes ao caminho previsto continua normalmente. Se a execução do desvio indicar que a previsão estava correta, a operação prossegue normalmente, como descrito acima. No entanto, se a previsão estiver incorreta, as entradas do *buffer* seguintes à do desvio são eliminadas. Isto equivale a descartar os resultados das instruções despachadas após o desvio que já foram completadas. As identificações das unidades funcionais nas entradas do *buffer* são usadas para localizar e descartar instruções ainda não completadas. Até este momento, nenhuma das ins-truções descartadas modificou os registradores. Os registradores serão modificados apenas pelas instruções despachadas antes do desvio, que correspondem às entradas no *buffer* antes daquela correspondente ao desvio.

Note que sob o ponto de vista do programa em execução, a memória principal também faz parte do estado da arquitetura. Assim, também é necessário controlar a modificação da memória por instruções *store* que seguem uma instrução de desvio. Quando um *store* é despachado, uma entrada é alocada no *buffer* de reordenação conforme descrito acima. Quando esta instrução é executada, o resultado é armazenado na entrada no *buffer* de reordenação, e em uma fila de escrita de memória é armazenado o endereço da locação de memória. Quando a entrada da instrução *store* atinge o início do *buffer* de reordenação, o resultado ali armazenado é associado ao endereço correspodente na fila de escrita de memória e o acesso de escrita é executado.

Este mecanismo apresenta um problema. Considere duas instruções i_j e i_k, despachadas nesta ordem, onde i_k depende de i_j. A instrução i_k entra em execução somente quando a entrada de i_j alcançar o início do *buffer* e o registrador destino é atualizado. No entanto, isto pode acontecer vários ciclos após a execução de i_j: basta que entre i_j e i_k seja despachada uma instrução com latência de vários ciclos. Neste caso, a execução de i_k é adiada indevidamente, pois o resultado por ela usado já foi produzido há vários ciclos. Assim, o *buffer* de reordenação pode aumentar o tempo de resolução das dependências, o que desencoraja o seu uso apesar da simplicidade de operação e

de implementação. Este problema é corrigido no mecanismo descrito a seguir.

2.4.2 O Buffer de História

A desvantagem do *buffer* de reordenação decorre do fato que os registradores (e a memória) não são atualizados quando uma instrução é completada. Se a instrução modificasse imediatamente o registrador destino, o início da execução de outras instruções seria determinado apenas pelas dependências. Uma forma de permitir que isto aconteça consiste em salvar o valor original do registrador destino, tornando possível a restauração deste valor caso uma instrução tenha que ser anulada. O *buffer* de história (*history buffer*) é um mecanismo que permite esta recuperação de estado. Como mostra a Figura 2.17(a), o *buffer* de história possui uma estrutura similar ao *buffer* de reordenação. A única diferença é a adição do campo ORG, usado para armazenar o valor original do registrador destino da instrução.

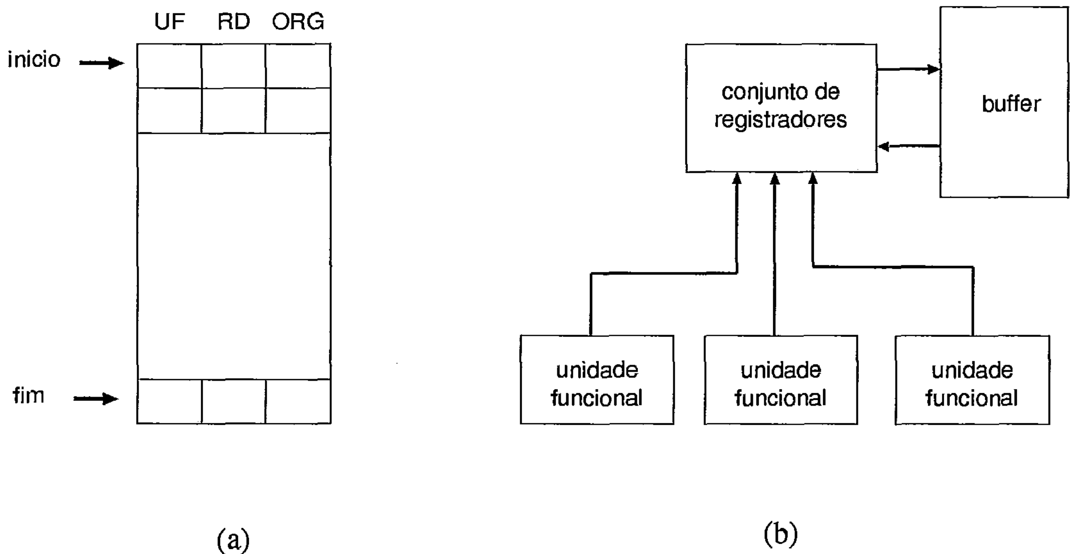


Figura 2.17: O *buffer* de história.

No despacho de cada instrução, uma entrada é inserida no *buffer* de história, contendo o valor do registrador destino. Tão logo a instrução é completada, o resultado produzido é armazenado no registrador destino, independente da posição da entrada correspondente no *buffer* de história. Entradas que alcançam o início do *buffer* cujas

instruções já foram executadas são retiradas do *buffer*.

Enquanto o *buffer* de reordenação era usado para manter um estado temporário criado por instruções executadas especulativamente, o *buffer* de história é usado para recuperar o estado anterior à execução destas instruções. Quando uma instrução de desvio é completada e a previsão do seu resultado estiver correta, a execução prossegue normalmente. No entanto, quando a previsão estiver errada, é necessário recuperar o estado anterior. Para tanto, o *buffer* de história é percorrido a partir do seu final até a posição onde se encontra a instrução de desvio, e os valores originais armazenados nas entradas são copiados para os registradores, desfazendo assim as modificações efetuadas pelas instruções despachadas após o desvio. Note que o percurso na direção da instrução de desvio faz com que os valores originais sejam restaurados em uma ordem inversa com a qual foram alterados. No final, o estado dos registradores é o mesmo que existia imediatamente após o despacho do desvio, mesmo que duas ou mais instruções tenham modificado um mesmo registrador.

A principal desvantagem do *buffer* de história está justamente no modo como a recuperação de estado é feita. O percurso através do *buffer* para a restauração dos registradores é um processo seqüencial que pode consumir vários ciclos. Isto pode introduzir um custo adicional que, apesar de se manifestar apenas nos casos de previsão incorreta, pode apresentar um efeito significativo sobre o desempenho. Esta desvantagem do *buffer* de história é resolvido no mecanismo descrito a seguir.

2.4.3 Registradores Futuros

Neste mecanismo é mantido o *buffer* de reordenação, mas os registradores são duplicados em dois conjuntos idênticos, denominados conjunto futuro (*future file*) e conjunto real (*architectural file*). A estrutura de uma arquitetura com conjunto futuro é mostrada na Figura 2.18.

A operação acontece da seguinte forma. Quando uma instrução é despachada, uma entrada é inserida no *buffer* de reordenação como descrito anteriormente. Quando a instrução é completada, o resultado é armazenado nesta entrada e no registrador destino dentro do conjunto futuro. As instruções acessam operandos a partir do conjunto futuro, de forma que a atualização imediata deste conjunto assegura que a execução de uma instrução não seja atrasada desnecessariamente. O registrador destino no conjunto real é atualizado com o resultado armazenado no *buffer* de reordenação so-

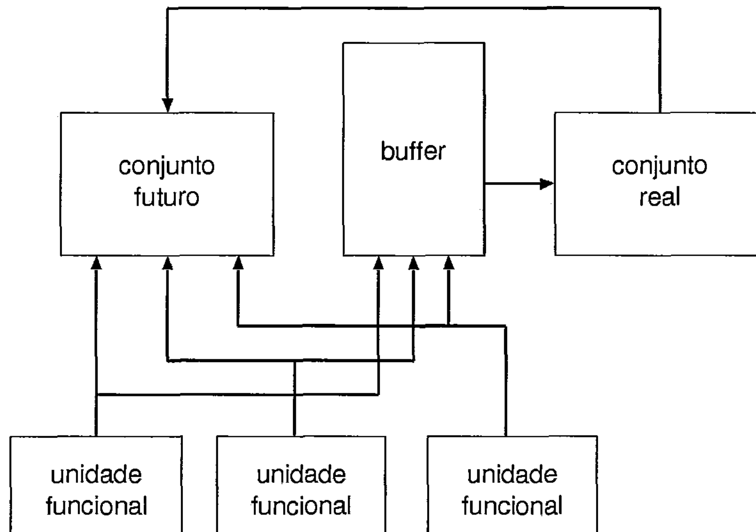


Figura 2.18: Arquitetura com conjunto futuro de registradores.

mente quando a entrada atingir o início do *buffer* e a instrução correspondente estiver concluída.

Vamos agora verificar como se comporta o mecanismo na presença de instruções de desvio. Quando a instrução de desvio alcançar o início do *buffer*, é verificado se a previsão do seu resultado estiver correta e, se este é o caso, a operação prossegue normalmente como descrito acima. Se a previsão estiver incorreta, o conteúdo do conjunto real é copiado para o conjunto futuro. Note que nenhuma das instruções despachadas após o desvio modificou o conjunto real, e assim estes registradores não foram alterados indevidamente. A cópia para o conjunto futuro recupera o estado anterior ao despacho da instrução de desvio. A vantagem sobre o *buffer* de história está no fato que as cópias entre registradores dos dois conjuntos podem ser feitas em paralelo, consumindo um número de ciclos bem menor que na varredura do *buffer* de história.

Capítulo 3

Modelos de Arquitetura

Este capítulo apresenta os modelos de arquiteturas super escalares que foram utilizados nos nossos experimentos: o **modelo bloqueante** e o **modelo especulativo.** Estes dois modelos diferem no modo como as dependências de controle são tratadas. Como visto no capítulo anterior, dependências de controle podem representar uma séria limitação no desempenho de uma arquitetura super escalar. Utilizando estes dois modelos, podemos verificar como o tratamento das dependências de controle afetam o balanço dos componentes de uma arquitetura super escalar.

Nossos modelos de arquitetura incorporam alguns dos mecanismos apresentados no capítulo anterior, e aqui será descrito apenas o funcionamento integrado destes componentes. Inicialmente é descrito o **modelo básico**, a partir do qual são derivados os modelos bloqueante e especulativo. As características particulares destes dois modelos serão descritas após a apresentação do modelo básico.

3.1 O Modelo Básico

As estruturas e mecanismos presentes no modelo básico também são encontrados nos modelos bloqueante e especulativo, e operam da mesma forma como será agora descrito. Assim, sempre que o termo "modelo básico" for aqui usado, entenda-se que o mesmo vale para os outros dois modelos de arquitetura. A organização do modelo básico é mostrada na Figura 3.1.

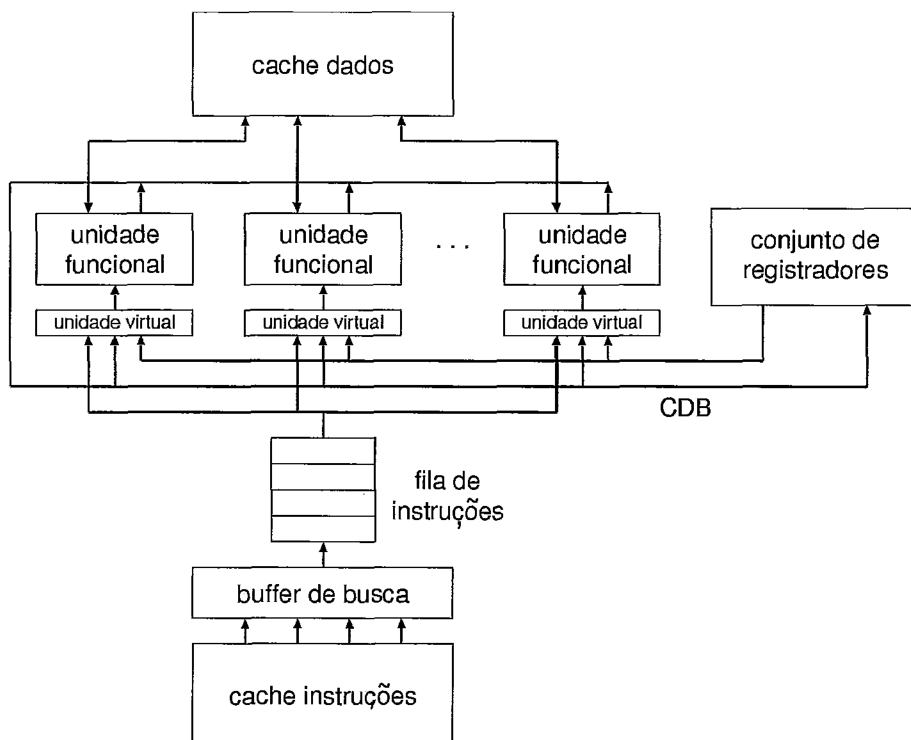


Figura 3.1: Organização do modelo básico.

Nesta arquitetura, várias instruções são acessadas simultaneamente em uma memória *cache*. O número máximo de instruções acessadas a cada ciclo é chamado **largura de busca**. Sob o ponto de vista de sua organização lógica, a memória *cache* de instruções é do tipo associativa por conjunto. Para viabilizar o acesso de múltiplas instruções a cada ciclo, supõe-se que a memória *cache* é fisicamente organizada em bancos independentes, com um número de bancos igual à largura de busca. O tamanho de uma linha física (em palavras) é igual ao número de bancos. Instruções em uma mesma linha lógica são armazenadas intercaladamente através dos bancos, ou seja, a instrução i dentro da linha lógica está armazenada no banco $i \ mod \ b$, onde b é a largura de busca. O tamanho da linha lógica pode ser maior que o da linha física. Neste caso, a linha lógica ocupa mais de uma linha física. Múltiplas instruções podem ser acessadas em um único ciclo, mesmo que algumas destas instruções estejam em linhas físicas diferentes. Esta organização física da memória *cache* de instruções é similar à da memória *cache* de instruções do IBM RS/6000 [Grohoski 90].

As instruções acessadas na memória *cache* são armazenadas temporariamente no **buffer de busca**. Se ocorrer falha na memória *cache*, as instruções que estiverem na *cache* são imediatamente armazenadas no *buffer* de busca, e as instruções faltando são carregadas na *cache* e depois transferidas para o *buffer* de busca. Novos acessos à memória *cache* são bloqueados durante o atendimento de uma falha. Esta restrição garante que as instruções serão inseridas em-ordem na fila de instruções (o motivo desta restrição será apresentado mais à frente, na descrição do modelo especulativo).

Instruções são retiradas do *buffer* de busca e inseridas na **fila de instruções**. Durante esta transferência, as instruções são pré-decodificadas, produzindo um formato de instrução que facilita a implementação do despacho. Por exemplo, tal formato indica o tipo de unidade funcional requerido pela instrução, se um desvio é incondicional ou condicional, ou ainda se uma instrução de acesso à memória é um *load* ou *store*.

Para resolver dependências de dados entre instruções, o modelo básico incorpora um mecanismo de escalonamento dinâmico de instruções baseado no algoritmo de Tomasulo. Como visto no Capítulo 2, o algoritmo de Tomasulo é capaz de resolver dependências de dados sem interromper o despacho de instruções. Assim, com a inclusão do algoritmo de Tomasulo, é possível minimizar o efeito das dependências de dados sobre o despacho de instruções, permitindo-nos focalizar apenas os efeitos das dependências de controle. Como visto no Capítulo 1, o algoritmo de Tomasulo é usado em algumas arquiteturas, como por exemplo o PowerPC 604. Assim, o uso deste mecanismo não foge à nossa premissa de avaliar arquiteturas super escalares com base em modelos que refletem características encontradas em arquiteturas reais.

Instruções são retiradas da fila e despachadas em-ordem para as unidades virtuais. O termo **largura de despacho** será aqui usado para denotar o número máximo de instruções que podem ser despachadas por ciclo. O algoritmo de despacho usado no modelo básico procura otimizar a utilização dos recursos disponíveis. O algoritmo de despacho mais simples consistiria em encontrar a primeira unidade virtual livre, dentre aquelas associadas às unidades funcionais que executam a instrução a ser despachada. No entanto, esta política não garante a melhor utilização das unidades funcionais. Por exemplo, pode ser encontrada uma unidade virtual associada a uma unidade funcional que esteja ocupada, enquanto existe uma unidade funcional ociosa também com unidades virtuais livres. Para tornar mais eficiente o uso dos recursos, foi adotado o algoritmo de despacho que aparece na Figura 3.2.

Em um primeiro nível, o algoritmo procura uma unidade funcional ociosa. Se for

```
procure unidade funcional ociosa com unidade virtual livre;
se (unidade funcional encontrada)
        despache instrução para a unidade funcional;
senão
        enquanto (unidade funcional não encontrada) {
                se (unidade funcional apontada pelo registrador de despacho possui unidade virtual livre)
                        despache instrução para unidade funcional;
                senão
                        se (todas unidades funcionais já foram verificadas)
                                termine;
                incremente registrador de despacho módulo número de unidades funcionais;
        }
```

Figura 3.2: Algoritmo de despacho de instruções no modelo básico.

encontrada uma unidade funcional ociosa com unidade virtual livre, a instrução é despachada para esta unidade virtual. Com isto, o algoritmo procura manter todas as unidades funcionais ocupadas. O segundo nível do algoritmo opera quando não é encontrada uma unidade funcional ociosa. Um registrador especial, denominado **registrador de despacho** indica a próxima unidade funcional para onde deve ser despachada uma instrução. Este registrador é usado para distribuir circularmente as instruções entre as unidades funcionais. Com este procedimento, o algoritmo procura balancear a carga de execução de instruções entre as unidades funcionais.

O modelo básico inclui uma memória *cache* de dados, do tipo associativa por conjunto. A princípio, assume-se que a memória *cache* de dados possui uma porta dedicada para cada unidade funcional, permitindo múltiplos acessos simultâneos. Esta facilidade será restringida mais adiante, quando forem acrescentadas ao modelo unidades funcionais específicas para executar instruções de acesso à memória. Para permitir acessos fora-de-ordem à memória, foi incluído no modelo básico um mecanismo para detectar dependências entre acessos. Este mecanismo segue duas regras: (1) uma instrução *load* é executada fora-de-ordem somente quando não há uma instrução *store* anterior com endereço efetivo desconhecido ou coincidente; (2) uma instrução *store* é executada fora-de-ordem somente quando não existe uma instrução *load* ou *store* anterior com endereço efetivo desconhecido ou coincidente. A primeira restrição re-

solve dependências verdadeiras entre os acessos, enquanto a segunda assegura que as anti-dependências e dependências de saída serão respeitadas.

No modelo básico, instruções são executadas através de um *pipeline* com quatro estágios. O primeiro estágio acessa instruções na memória *cache*, colocando-as no *buffer* de busca. O segundo estágio retira instruções do *buffer* de busca, pré-decodifica e as insere na fila de instruções. Este estágio também despacha instruções que se encontram na fila. O terceiro estágio procura, nas unidades virtuais, instruções prontas para serem executadas, e inicia a execução destas instruções nas unidades funcionais. O último estágio propaga o resultado de instruções completadas, através do CDB, para as unidades virtuais e para o conjunto de registradores. Alguns detalhes adicionais do funcionamento do *pipeline*, específicos dos modelos bloqueante e especulativo, são descritos nas seções a seguir.

3.2 O Modelo Bloqueante

Sob o ponto de vista estrutural, o modelo bloqueante é completamente idêntico ao modelo básico, mostrado na Figura 3.1. Neste modelo, o despacho de novas instruções é bloqueado quando uma instrução de desvio é despachada. O despacho é liberado somente quando o resultado do desvio é determinado e o contador de programa é alterado com o endereço da próxima instrução. Com o uso deste modelo, o objetivo é avaliar o desempenho de arquiteturas onde a resolução de dependências de controle é feita no despacho. Este método de resolução de dependências de controle é adotado em arquiteturas tais como Intel Pentium, HP PA7100 e DEC Alpha 21064.

Durante o intervalo de tempo em que o despacho permanece bloqueado, novas instruções continuam a ser acessadas e inseridas na fila de instruções. Ao ser encontrada uma instrução de desvio, é feita uma previsão estática de que o desvio não será tomado, e o acesso continua com as instruções adjacentes. Quando o desvio for executado e a previsão foi incorreta, todas as instruções no *buffer* de busca e na fila de instruções são descartadas antes que o despacho seja liberado.

3.3 O Modelo Especulativo

No modelo especulativo, o despacho não é bloqueado por instruções de desvio. Ao contrário, o despacho prossegue normalmente, e as instruções subseqüentes são executadas especulativamente. Este método de tratamento de dependências de controle é encontrado nas arquiteturas Motorola M88110, e IBM/Motorola/Apple PowerPC 603 e 604. O termo **profundidade de especulação** será aqui usado para indicar o número máximo de instruções de desvio através do qual a execução especulativa pode prosseguir. Um *buffer* de reordenação com conjunto de registradores futuros são usados para suportar a execução especulativa. A Figura 3.3 mostra a organização do modelo especulativo.

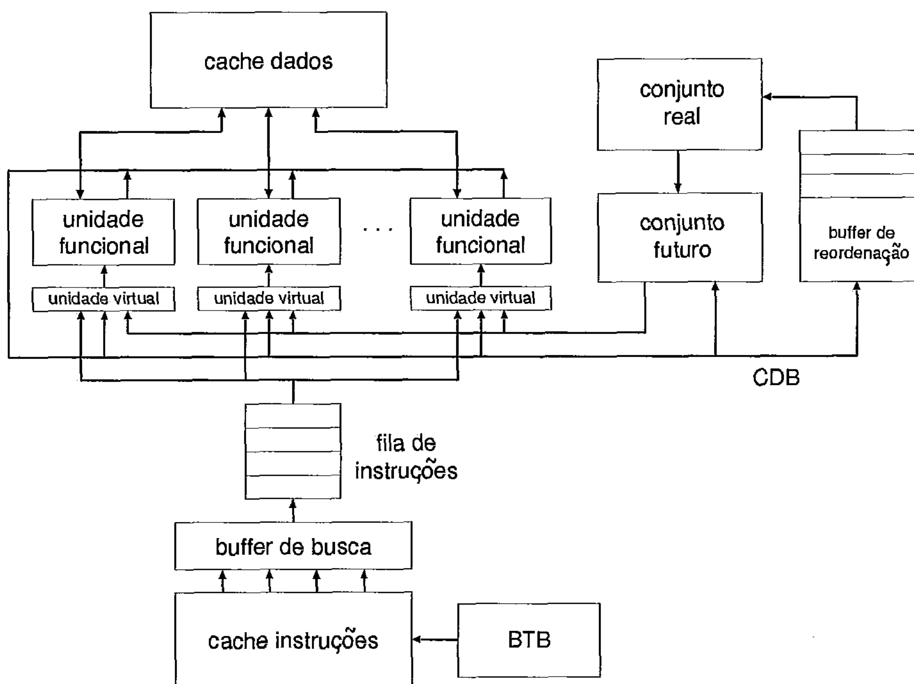


Figura 3.3: Organização do modelo especulativo.

No despacho, um *tag* identificando a unidade virtual alocada para a instrução e um outro *tag* indicando o registrador destino da instrução são inseridos em uma entrada no final do *buffer* de reordenação. Note que tais informações são inseridas no *buffer* de reordenação na mesma ordem que as instruções correspondentes se encontram no código do programa. Isto acontece porque, como mencionado na descrição do modelo

básico, instruções são inseridas em-ordem na fila de instruções. Assim, o despacho a partir da fila de instruções faz com que as informações também sejam inseridas em-ordem no *buffer* de reordenação. Este procedimento assegura que a atualização do conjunto de registradores reais, a partir dos valores no *buffer* de ordenação, esteja de acordo com a ordem das instruções no código do programa, o que é necessário para a recuperação do estado correto do conjunto de registradores futuros.

Quando uma instrução é completada, o resultado é transferido (via CDB) para o *buffer* de reordenação e para o conjunto futuro. Para determinar a entrada do *buffer* de reordenação onde o resultado deve ser escrito, o *tag* de unidade virtual presente no CDB é comparado com os *tags* armazenados no *buffer*. Em uma implementação real, esta comparação pode ser feita associativamente, como acontece no conjunto de registradores. O resultado é posteriormente transferido do *buffer* de reordenação para o conjunto real quando a entrada atingir o início do *buffer* de reordenação. A transferência de resultados do *buffer* de reordenação para o conjunto real é feita pelo quarto estágio do *pipeline*, juntamente com a propagação de resultados através do CDB.

Se durante a execução de um de desvio for verificado que o seu resultado foi previsto incorretamente, o despacho é bloqueado até que a entrada correspondente alcance o início do *buffer* de reordenação. Note que, tão logo seja executada uma instrução de desvio prevista incorretamente, a busca de instruções é redirecionada para o caminho correto, prosseguindo normalmente. Assim, se o despacho não fosse bloqueado, instruções no ramo correto do desvio poderiam usar valores deixados no conjunto futuro pelas instruções executadas especulativamente. Neste modelo, esta é a única situação onde o despacho é bloqueado por uma instrução de desvio.

Quando a entrada correspondente à instrução de desvio atingir o início do *buffer* de reordenação, as instruções nas unidades virtuais são descartadas. Para tanto, são usados os *tags* das unidade virtuais armazenadas nas entradas seguintes à do desvio, no *buffer* de reordenação. Em seguida, estas entradas são também descartadas. Neste momento, os resultados das instruções despachadas antes do desvio já estarão armazenados no conjunto real, e o conteúdo do conjunto real é então copiado para o conjunto futuro. Após a cópia para o conjunto futuro, todos os bits de alocação dos registradores deste conjunto são desativados. Como todas as instruções despachadas antes do desvio já foram completadas, estes bits foram ativados por instruções despachadas após o desvio. Apenas os valores dos registradores são copiados do conjunto real para o conjunto futuro, pois os registradores no conjunto real não possuem bits

de alocação ou *tags* associados.

No modelo especulativo, a previsão de desvios é dinâmica, sendo usado um *branch target buffer* (BTB) (ver Capítulo 2). Quando uma instrução de desvio é acessada, o BTB é consultado e o estágio de busca do *pipeline* é redirecionado para o destino previsto. Se o desvio não for encontrado no BTB, é feita uma previsão estática de desvio não-tomado, e as instruções adjacentes são acessadas. Quando a instrução de desvio for executada e a previsão estiver correta, a informação no BTB é conservada. Caso contrário, o BTB é atualizado e a busca de novas instruções é redirecionada para o endereço correto. São usados dois bits de previsão, que são modificados conforme descrito no Capítulo 2.

3.4 Validade dos Modelos

Uma das preocupações neste trabalho foi a validade dos modelos de arquitetura aqui adotados, no sentido de incluírem técnicas e mecanismos de fato encontrados em arquiteturas super escalares reais.

Os modelos aqui descritos caracterizam-se principalmente pelos mecanismos usados no tratamento de dependências de dados e de controle. A maioria das arquiteturas super escalares atuais empregam alguma forma de escalonamento dinâmico de instruções, aliado ao escalonamento estático realizado pelo compilador, para resolver dependências de dados. Estes mecanismos diferem basicamente no seu grau de sofisticação: alguns realizam uma simples inversão de instruções, enquanto outros usam estações de reserva e renomeação de registradores. Os modelos aqui considerados guardam uma maior semelhança com estas últimas arquiteturas que usam formas mais sofisticadas de resolução de dependências de dados. Acreditamos que o uso destes mecanismos sejam a tendência em futuros processadores super escalares de médio e alto desempenho.

Quanto ao tratamento das dependências de controle, os dois modelos aqui usados foram elaborados exatamente para retratar as diferenças existentes entre arquiteturas super escalares reais, no que se refere a este aspecto. Estas diferenças resumem-se basicamente no bloqueio ou não do despacho de instruções para resolver dependências de controle. O modelo bloqueante e o modelo especulativo correspondem a estas duas diferentes abordagens.

Em conclusão, os dois modelos descritos neste capítulo não incluem nenhuma facilidade específica, que não seja encontrada sob alguma forma em arquiteturas super escalares reais. Isto confere uma validade aos resultados que serão aqui apresentados, no sentido de que podem ser usados para se avaliar o desempenho potencial de arquiteturas super escalares correntes.

Capítulo 4

O Ambiente Experimental

No capítulo anterior apresentamos os dois modelos de arquiteturas super escalares considerados neste trabalho. Este capítulo descreve o ambiente experimental desenvolvido para a avaliação destas arquiteturas. Inicialmente, é descrita a arquitetura usada como referência na avaliação do desempenho dos modelos bloqueante e especulativo. Depois, são descritos os simuladores da arquitetura referência e das arquiteturas super escalares.

4.1 A Arquitetura Referência

Neste trabalho, o ganho de desempenho obtido com as arquiteturas super escalares é medido usando-se a arquitetura Sun SPARC [Sun87] [Garner 88] como referência. A arquitetura SPARC é um exemplo de arquitetura RISC de alto desempenho, e o seu sucesso é atestado pelo amplo uso da linha de estações de trabalho SPARCstation nas mais diferentes aplicações. Assim, uma comparação de desempenho usando esta arquitetura como referência é extremamente significativa.

A denominação SPARC (*Scalable Processor ARChitecture*) não refere-se a um processador em particular, mas à definição de uma arquitetura. Algumas características da arquitetura, como por exemplo o número de registradores e a organização do *pipeline*, são deixadas em aberto e dependem de cada particular implementação. A Figura 4.1 mostra a organização da arquitetura SPARC.

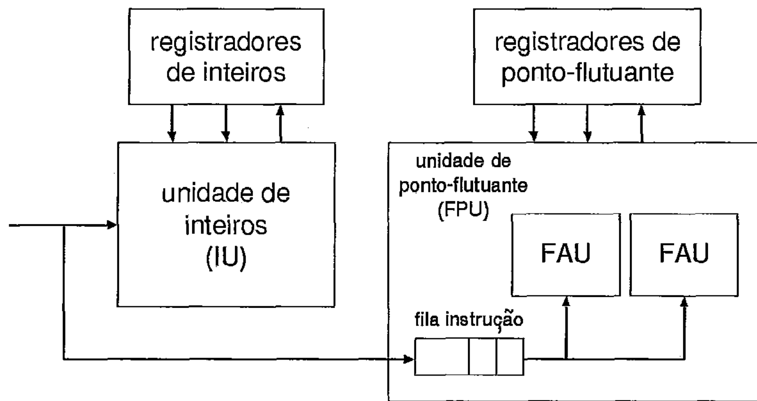


Figura 4.1: Organização da arquitetura SPARC.

A arquitetura SPARC possui duas unidades funcionais, a unidade de inteiros (IU) e a unidade de ponto flutuante (FPU). A unidade de inteiros executa instruções aritméticas e lógicas sobre inteiros de 32 bits, instruções de acesso à memória e instruções de desvio. A FPU opera sobre dados em ponto flutuante, de acordo com o padrão ANSI/IEEE 754. A FPU pode incluir uma ou mais unidades aritméticas de ponto flutuante (*Floating-point Arithmetic Units*, ou FAU). O número de FAU's depende da implementação.

As unidades IU e a FPU são independentes, e operam em paralelo. A cada ciclo a IU acessa apenas uma instrução, que pode ser destinada para a própria IU ou para a FPU. A FPU monitora os acessos a instruções, reconhecendo as instruções de ponto flutuante. Ao encontrar uma instrução de ponto flutuante, a FPU a coloca em uma fila para posterior execução, enquanto a IU ignora aquela instrução. Uma instrução na fila da FPU é executada somente se existir uma FAU disponível e quando os registradores especificados na instrução não estiverem alocados por outra instrução em execução. Se uma destas condições não for satisfeita, a FPU suspende temporariamente a execução de novas instruções, até que os recursos necessários estejam disponíveis. Se a fila da FPU estiver completamente preenchida, a IU suspende a busca de instruções até que a fila tenha espaço para receber novas instruções. O tamanho da fila da FPU é igual ao número de FAU's existentes na unidade de ponto flutuante.

A FPU possui seu próprio conjunto de registradores, com 32 registradores de 32 bits. O conjunto de registradores associado a IU é logicamente organizado em bancos

de registradores de 32 bits, denominados janelas de registradores (*register windows*). Este mecanismo foi criado para reduzir o custo das operações de chamada e retorno de rotinas, e detalhes de sua operação podem ser encontrados em [Tamir 83] e [Katevenis 85].

No Apêndice A apresentamos o conjunto de instruções da arquitetura SPARC. Por ser uma arquitetura RISC, são oferecidos apenas os tipos básicos de instruções. A arquitetura SPARC emprega a técnica de desvios atrasados (discutida no Capítulo 2) para reduzir o custo das instruções de desvio. Desvios são executados com atraso de uma instrução. As instruções de desvio condicional são dotadas de uma facilidade especial, chamada anulação de instruções (*instruction nullify*). No código destas instruções existe um bit, denominado *annul*, que indica se a instrução na posição de atraso deve ser executada ou não. Se *annul* estiver desativado, a instrução na posição de atraso será executada. Se *annul* estiver ativado, a instrução na posição de atraso é executada apenas se o desvio for tomado. Se o desvio não for tomado, a instrução na posição de atraso não será executada. A anulação de instruções tem como objetivo facilitar o escalonamento de instruções. Uma das estratégias para preencher a posição de atraso consiste em usar a instrução no destino do desvio [Hennessy 90]. No entanto, esta estratégia é aplicável apenas se for semanticamente correto executar a instrução na posição de atraso - que, no código não escalonado, é a instrução no destino do desvio - caso o desvio não seja tomado. O mecanismo de anulação de instruções torna esta restrição mais flexível. Se a instrução na posição de atraso não deve ser executada quando o desvio não for tomado, o compilador gera a instrução de desvio com o bit *annul* ativado. Assim, a instrução na posição de atraso é anulada quando o desvio não for tomado, mantendo a correção do programa.

Compiladores para a arquitetura SPARC são extremamente eficientes em alocar instruções úteis para as posições de atraso. Muchnick [Muchnick 88] descrevem um algoritmo de escalonamento de instruções para a arquitetura SPARC que preenche até 95% das posições de atraso com instruções úteis, sendo que cerca da metade das posições são preenchidas sem o uso da facilidade de anulação de instruções. Isto torna o código extremamente eficiente, pois o número de ciclos desperdiçados nas posições de atraso é muito pequeno. O compilador C usado neste trabalho para gerar o código dos programas de teste utiliza esta técnica de escalonamento de instruções. Isto torna ainda mais significativa a avaliação do desempenho das arquiteturas super escalares, já que esta avaliação será feita usando-se um código cujas dependências de controle possuem um efeito pequeno sobre o desempenho da arquitetura referência.

O processador Fujitsu MB86900 [Namjoo 88] é um exemplo de implementação da arquitetura SPARC. O simulador da arquitetura referência, descrito na próxima seção, reflete as características deste processador. O MB86900 possui um *pipeline* de quatro estágios: busca, decodificação, execução e escrita de resultado. O MB86900 usa a técnica *forwarding* [Hennessy 90] para tratar as dependências (verdadeiras) de dados entre instruções. Considere duas instruções consecutivas i e $i+1$, onde $i+1$ depende do resultado de i. Se $i+1$ chegar ao estágio de decodificação no mesmo ciclo em que i ainda estiver no estágio de execução, o operando de $i+1$ a ser lido naquele momento ainda não estará pronto no registrador. A princípio, a instrução $i+1$ deveria esperar um ciclo adicional no estágio de decodificação, até que o resultado da instrução i fosse armazenado no registrador destino, pelo estágio de escrita. No entanto, isto resultaria em uma paralização no *pipeline*, e a conseqüente queda no desempenho. A técnica *forwarding*, usada para resolver este problema, opera da seguinte forma. Quando for detectada uma dependência entre instruções nos estágios de decodificação e execução, o resultado produzido no final do ciclo corrente é realimentado diretamente para a entrada da ALU. Assim, no ciclo seguinte, a instrução que se encontrava no estágio de decodificação pode avançar para o estágio de execução e usar o resultado, ao mesmo tempo em que ele é armazenado no registrador. Com este esquema, dependências de dados não introduzem nenhuma penalidade no desempenho do *pipeline*.

Como especificado na definição da arquitetura SPARC, o MB86900 implementa desvios com atraso de uma instrução. Ao encontrar uma instrução de desvio condicional, o estágio de decodificação calcula o endereço destino, avalia o código de condição e redireciona o estágio de busca conforme o resultado do desvio. No ciclo em que isto acontece, o estágio de busca completa o acesso da instrução na posição de atraso. Ao final do ciclo seguinte, o estágio de busca completa o acesso da instrução no destino do desvio. Desta forma desvios condicionais, quer sejam tomados ou não, são executados em um único ciclo. Ao contrário dos desvios condicionais, instruções de desvio incondicional são executadas em dois ciclos. Isto acontece porque nestas instruções o endereço destino não é calculado a partir de um dado imediato (como nas instruções de desvio condicional), mas sim a partir de um valor em registrador. Assim, o estágio de decodificação necessita de um ciclo adicional para ler este valor, antes de calcular o endereço destino.

O MB86900 não inclui memórias *cache* separadas para instruções e dados, e o único barramento de dados externo é usado tanto para o acesso a instruções como a dados. Por isso, as instruções de acesso à memória também são executadas em dois ciclos. No ciclo em que uma instrução de acesso à memória chega ao estágio de execução, o

Tipo de Instrução	Latência
instruções aritméticas e lógicas	1 ciclo
desvio condicional tomado	1 ciclo
desvio condicional não-tomado	1 ciclo
desvio incondicional	2 ciclos
load/store halfword, word	2 ciclos
load/store doubleword	3 ciclos

Tabela 4.1: Latências de instruções no MB86900.

acesso não pode ser efetuado porque os barramentos externos estão sendo utilizados pelo estágio de busca de instrução. O acesso acontece apenas no ciclo seguinte, quando os barramentos são alocados para a instrução de acesso à memória. A Tabela 4.1 resume as latências de instruções no MB86900.

O MB86900 inclui apenas a unidade de inteiros SPARC. Nas estações SPARCstation 2, a unidade de ponto flutuante é implementada com as unidades aritméticas Weitek WTL 1164 (ALU) e WTL 1165 (multiplicador). Estas unidades realizam adição e multiplicação em precisão simples em 10 ciclos, adição em precisão dupla em 13 ciclos e multiplicação em precisão dupla em 15 ciclos.

Como mencionado, o processador MB86900 não inclui memórias *cache*. No entanto, neste trabalho foi considerada a existência de memórias *cache* separadas para instruções e dados, ambas do tipo associativa por conjunto, com 32 bytes por linha, 2 linhas por conjunto e 128 conjuntos, totalizando 8 Kbytes cada. Tal configuração é comum em vários processadores atuais. A latência de *hit* na memória *cache* de instruções é de 1 ciclo. A latência de *hit* na memória *cache* de dados depende do tipo de acesso, e segue as latências na Tabela 4.1. Foi considerado também a operação do MB86900 em um sistema com memória *cache* secundária operando em *pipeline*, onde o primeiro acesso a uma palavra de 32 bits consome 4 ciclos e acessos subseqüentes a palavras consecutivas consomem 1 ciclo por palavra. Para um barramento de dados de 32 bits entre as memórias *cache* primária e secundária, os valores acima resultam em uma latência de 11 ciclos para preencher uma linha da memória *cache* primária. Todas as latências acima mencionadas são levadas em consideração pelo simulador

da arquitetura SPARC, descrito a seguir.

4.2 O Simulador da Arquitetura SPARC

O simulador da arquitetura referência - daqui em diante chamado "simulador SPARC" - segue as características do MB86900 descritas na seção anterior. Ele reproduz a operação do *pipeline* do MB86900, incluindo o *forwarding* e a execução de desvios com atraso. As instruções são executadas com as latências especificadas anteriormente. No simulador também foram incluídas memórias *cache* separadas para instruções e para dados.

O simulador SPARC recebe como entrada um arquivo executável no formato UNIX *a.out*, interpreta o cabeçalho do arquivo, carrega o código na memória simulada e inicia a execução do programa a partir do ponto de entrada especificado no cabeçalho. Existem duas limitações no simulador SPARC. Primeiro, ele não suporta programas que realizam *binding* dinâmico. Isto representa uma desvantagem apenas quanto ao tamanho do código executável do programa, já que o *binding* deve ser feito em tempo de compilação. Segundo, por motivos de simplicidade, o simulador não suporta programas concorrentes. Alguns programas de teste usados neste trabalho, que criam um processo filho para realizar tarefas de pré-processamento, foram modificados para executarem como um único processo.

O simulador SPARC está voltado para o sistema operacional UNIX. Ao encontrar uma instrução `TRAP` no fluxo de execução do programa, ele interpreta o código da chamada, acessa os parâmetros na memória simulada, e realiza a chamada ao sistema operacional. Caso haja algum valor de retorno, o simulador transfere esse valor para a memória simulada. Desta forma, sob ponto de vista do programa em execução, a chamada ao sistema operacional é de fato executada.

O simulador SPARC também recebe como entrada o número de instruções que devem ser simuladas, e gera arquivos de *trace* que serão usados pelos simuladores das arquiteturas super escalares. São gerados dois arquivos de *trace*:

- *trace* de desvios: contém informações sobre as instruções de desvio executadas pelo programa. Para cada instrução de desvio executada, o arquivo possui um registro com o endereço da instrução de desvio, o endereço destino e uma indicação se o desvio foi tomado ou não;

- *trace* de dados: contém informações sobre as instruções de acesso à memória que foram executadas pelo programa de teste. Para cada instrução de acesso, existe um registro no arquivo com o endereço da instrução e o endereço da posição de memória acessada.

Em geral, simuladores produzem um único arquivo de *trace* contendo todas as instruções executadas. A alternativa de gerar os dois arquivos descritos acima possui uma vantagem sobre o método convencional. Em uma simulação orientada por *trace* (*trace-driven simulation*), é suficiente conhecer o fluxo de controle durante a execução real do programa. O padrão de referências a dados na memória pode também ser necessário para reproduzir o comportamento de uma memória *cache* de dados. O fluxo de controle é indicado pela seqüência de instruções de desvio executadas pelo programa, que pode ser registrada por um arquivo *trace* de desvios como o descrito acima. Observe que as instruções entre dois desvios adjacentes são executadas seqüencialmente. Estas instruções podem ser acessadas diretamente a partir do arquivo executável, sendo desnecessário incluí-las no *trace*. Isto reduz substancialmente o tamanho do arquivo de *trace*. Por exemplo, considere um *loop* com 10 instruções, incluindo a instrução de desvio ao seu final. Com instruções de 4 bytes, seriam necessários 4000 bytes para registrar 100 iterações do *loop*, no método convencional. Na solução aqui adotada, cada registro no *trace* de desvio ocupa 12 bytes, e assim são necessários apenas 1200 bytes para registrar as mesmas 100 iterações do *loop*, já que apenas as informações sobre a instrução de desvio que fecha o *loop* são gravadas. Este método possibilitou simulações com um número de instruções bem maior do que seria possível com o método convencional.

Além de gerar os arquivos de *trace*, o simulador SPARC gera um arquivo de estatística, contendo o número de ciclos consumidos na execução do programa, o número médio de ciclos por instrução, as taxas de *hit* nas memórias *cache* e o ponto de entrada do programa. Estas informações serão utilizadas pelos simuladores das arquiteturas super escalares.

A correta operação do simulador SPARC é essencial, já que os *traces* e medidas de tempo por ele fornecidos serão usados como base de comparação na avaliação do desempenho das arquiteturas super escalares. Sob o ponto de vista funcional, o simulador é correto na medida que um programa por ele executado se comporta exatamente da mesma forma que o programa executado em uma SPARCstation 2, produzindo resultados iguais. Sob o ponto de vista temporal, verifica-se que o número médio de ciclos por instrução (cpi) contabilizado pelo simulador SPARC é idêntico

ao cpi característico do MB86900. Namjoo [Namjoo 88] cita que o fator cpi para o MB86900 situa-se entre 1.5 e 1.6 ciclos/instrução para programas com um distribuição típica de instruções. Este valor foi comparado com o cpi obtido pelo simulador SPARC na execução de alguns programas de teste usados neste trabalho. A Tabela 4.2 mostra os resultados obtidos.

Programa	cpi no simulador SPARC
espresso	1.72 cpi
eqntott	1.45 cpi
compress	1.69 cpi
gcc	1.80 cpi

Tabela 4.2: Fatores cpi do simulador SPARC.

É importante perceber que o fator cpi depende em parte das características do programa. Por exemplo, as dependências de instruções aritméticas e lógicas em relação à instruções *load*, que determinam a freqüência de travamento do *pipeline*, podem aumentar o cpi. Isto é o que ocorre com o programa *espresso*, onde 20% das instruções são *loads* (ver Capítulo 6). Um outro fator que aumenta o cpi é a característica de localidade do programa, que determina a taxa de *hit* das memórias *cache*. O programa *gcc* apresenta o cpi mais alto porque a taxa de *hit* das memórias *cache* para este programa é apenas 94%, enquanto que para os outros programas é de 99%. Finalmente, a eficiência do escalonador de instruções do compilador também repercute sobre o cpi. Por estes motivos, as variações dos fatores cpi fornecidos pelo simulador em relação aos valores citados em [Namjoo 88] são esperados e, na faixa em que situam-se, são plenamente aceitáveis como representativos de um processador *pipeline* real.

Uma descrição da estrutura e do funcionamento interno do simulador SPARC é apresentada no Apêndice B.

4.3 Os Simuladores das Arquiteturas Super Escalares

Além do simulador SPARC, foram desenvolvidos simuladores para os dois modelos de arquiteturas super escalares discutidas no Capítulo 3. Os simuladores recebem como entrada o arquivo objeto de um programa e os arquivos de *trace* e de estatística gerados pelo simulador SPARC. Em particular, o arquivo de estatística contém o número de ciclos contabilizado pelo simulador SPARC na execução do programa, valor este usado pelo simulador da arquitetura super escalar para calcular o ganho de desempenho. Os simuladores também recebem um arquivo de parâmetros, contendo as seguintes informações de configuração da arquitetura super escalar:

- a configuração das memórias *cache* de instruções e de dados: o tamanho da linha, o número de linhas por conjunto e o número de conjuntos;
- a largura de busca;
- o tamanho da fila de instruções;
- a largura de despacho;
- os tipos de unidades funcionais, e o número de unidades funcionais de cada tipo;
- o número de unidades virtuais associadas a cada unidade funcional;
- o número de CDBs;
- os tempos de latências das instruções;
- para o simulador especulativo: a configuração do BTB, a profundidade de especulação e o tamanho do *buffer* de reordenação.

Além do ganho de desempenho, os simuladores das arquiteturas super escalares produzem outras estatísticas que permitem uma análise detalhada do balanceamento da arquitetura. Estas informações são:

- as taxas de *hit* nas memórias *cache* de instruções e de dados;
- a taxa de bloqueio na busca de instruções, provocado pela falta de espaço na fila de instruções;

- a taxa de bloqueio de despacho, provocado pela falta de instruções na fila;
- a taxa de bloqueio de despacho pela falta de unidades virtuais livres;
- a taxa de bloqueio de despacho devido a instruções de desvio, no modelo bloqueante;
- a taxa de bloqueio de despacho, provocado pela falta de espaço no *buffer* de reordenação e pela profundidade máxima de especulação excedida, no modelo especulativo;
- a distribuição do número de instruções despachadas por ciclo;
- a taxa de utilização das unidades funcionais;
- a taxa de unidades funcionais ociosas, em virtude das unidades virtuais estarem vazias e aguardando por operandos;
- o número médio de unidades virtuais ocupadas por unidade funcional;
- a distribuição do número de instruções completadas por ciclo;
- o número médio de instruções completadas por ciclo (ipc).

Uma descrição da estrutura e funcionamento interno dos simuladores das arquiteturas super escalares pode ser encontrada no Apêndice C.

4.4 Os Programas de Teste

A escolha adequada de programas de teste é essencial na avalição do desempenho de uma arquitetura, quer esta avaliação seja realizada através de medidas em uma máquina real ou através de simulações. O conjunto de programs de teste deve refletir as tarefas computacionais normalmente realizadas pelo processador em um ambiente real. Hennessy e Patterson [Hennessy 90] identificam quatro classes de programas de teste:

- programas reais: utilitários, compiladores, formatadores de texto, programas de CAD, etc;

- *kernels*: programas que são constituídos por pequenos trechos de código encontrados freqüentemente em aplicações reais. Exemplos são o *Livermore Loops* e o *Linpack*;

- programas de teste sintéticos (*synthetic benchmarks*): formados por trechos de código elaborados com a finalidade de capturar as operações mais freqüentemente encontrados em programas. Exemplos são o *Dhrystone* e o *Whetstone*;

- *toy benchmarks*: como a própria denominação indica, tais programas de teste não reproduzem as características de programas reais, e que não possuem utilidade alguma a não ser como exercícios elementares de programação. Exemplos: *Sieve of Erastosthenes*, *Fibonnacci*, *Quicksort*, *BCDBin*.

Os *kernels* são úteis para avaliar isoladamente o desempenho de uma certa facilidade de um processador. Por exemplo, o *Linpack* é adequado para medir o desempenho de unidades de ponto flutuante. No entanto, devido ao padrão de execução, extremamente contido, o uso destes programas na avaliação de outros aspectos da arquitetura pode levar a resultados irreais. Por exemplo, *kernels* intensivos em *loops*, que apresentam uma alta localidade no padrão de acessos às instruções, geralmente superestimam o desempenho das memórias *cache* de instruções e de mecanismos de previsão dinâmica de desvios. Os programas de teste sintéticos apresentam as mesmas falhas que os *kernels*, e adicionalmente padecem do defeito de não serem retirados de programas reais, mas criados artificialmente. Em termos de avaliação do paralelismo a nível de instruções, o uso destes programas de teste pode ser altamente tendencioso. Os vícios de localidade, e a existência de *loops* com baixos níveis de dependências ou que mapeiam em uma determinada configuração de arquitetura, podem elevar artificialmente a média de instruções completadas por ciclo.

O uso dos *kernels*, dos programas sintéticos e dos *toy benchmarks* é muito conveniente. Estes programas normalmente não realizam chamadas ao sistema operacional. Assim torna-se fácil, por exemplo, construir o simulador de uma arquitetura que seja capaz de executar apenas estes tipos de programas. No entanto, para realizar testes que reproduzam com alguma precisão o comportamento de programas reais é necessário pagar o preço da complexidade, utilizando-se programas reais.

Neste trabalho, foram usados programas de teste que fazem parte do SPEC (*System Performance Evaluation Cooperative*), uma entidade formada por alguns fabricantes de processadores e sistemas, que tem como objetivo padronizar um conjunto de programas de teste que viabilize uma avalição realística de desempenho. Existem dois conjuntos de programas SPEC. O SPECint [SPEC92a] é formado por progra-

mas de teste inteiros e inclui utilitários, um compilador, e programas de CAD. O SPECfp [SPEC92b] é constituído por programas de ponto flutuante usados nas áreas de engenharia e física. Destes dois conjuntos, usamos os seguintes programas:

- *espresso*: um minimizador de expressões booleanas;
- *eqntott*: um gerador de mapa de PLAs a partir de expressões booleanas;
- *compress*: utilitário UNIX para compressão de arquivos que usa o algoritmo Lempel-Ziv;
- *gcc*: o compilador GNU C;
- *doduc*: um programa para a simulação do comportamento dinâmico de um componente de um reator nuclear, usando a técnica de Monte Carlo;
- *tomcatv*: programa para geração de malhas;
- *fpppp*: programa de química quântica, usado para calcular a derivada total do choque de dois elétrons.
- *nasa7*: conjunto de sete rotinas de manipulação de matrizes;

Estes programas foram usados sem modificações, com exceção do *eqntott* e *gcc*. Estes dois programas criam um processo que executa o pré-processador *cpp* para fazer um tratamento prévio dos arquivos de entrada. A saída pré-processada é transferida para o programa principal através de um *pipe*. Como o simulador SPARC não suporta programas concorrentes, as chamadas `fork`, `exec`, e `pipe` nestes programas foram desativadas, e suas entradas redirecionadas para ler arquivos em disco. Estes arquivos de entrada foram pré-processados à parte.

Os programas de teste inteiros foram compilados com o compilador UNIX C (cc) para o sistema operacional SunOS 4.1.1 com a opção de compilação *-fast*, que procura gerar código com o melhor tempo de execução (no caso, na arquitetura referência). Os programas de ponto flutuante foram traduzidos pelo compilador ANSI f77 para o mesmo sistema operacional, e com a mesma opção de otimização.

Para estes programas de teste, foram gerados *traces* cobrindo a execução de 10 milhões de instruções. A escolha deste tamanho foi determinada principalmente por motivos de tempo de simulação. Em uma SPARCstation 2, os simuladores das arquiteturas super escalares consomem 90 minutos de execução, ou cerca da metade deste tempo em uma SPARCStation 10, para estações sem outra carga de trabalho. Tendo em vista que se fazia necessário realizar mais de 1000 simulações, decidiu-se limitar o tamanho dos *traces* ao valor acima.

Capítulo 5

Avaliação do Modelo Bloqueante Homogêneo

Neste capítulo apresentamos e discutimos os resultados experimentais obtidos com o modelo bloqueante homogêneo. Como descrito no capítulo anterior, neste modelo o despacho de instruções é bloqueado quando uma instrução de desvio é encontrada, sendo retomado apenas quando o desvio é executado. A segunda característica deste modelo é que todas as unidades funcionais são idênticas, cada uma delas podendo executar qualquer tipo de instrução.

O objetivo aqui é mostrar como se comporta o desempenho deste particular modelo de arquitetura super escalar em função de três parâmetros: a largura de despacho, o número de unidades funcionais e o número de unidades virtuais. Além do ganho de desempenho, são apresentadas outras medidas importantes, como por exemplo o número médio de instruções despachadas por ciclo, o número médio de instruções completadas por ciclo e a utilização dos recursos.

5.1 Configurações do Modelo Homogêneo

Para caracterizar o desempenho do modelo bloqueante homogêneo, foram realizadas simulações usando diversas configurações de arquitetura. Cada uma destas configurações corrresponde a uma arquitetura com uma certa largura de despacho, e com um certo número de unidades funcionais e de unidades virtuais. A Tabela 5.1 mostra

os valores considerados para estes três parâmetros. Os resultados aqui apresentados foram obtidos a partir de configurações formadas pelas possíveis combinações dos valores mostrados na tabela, totalizando 128 configurações diferentes de arquitetura.

Parâmetro	Valores Usados
Largura de Despacho	2, 4, 6, 8 instruções
Número de Unidades Funcionais	1 a 8 unidades
Número de Unidades Virtuais	1 a 4 unidades

Tabela 5.1: Valores de parâmetros nas configurações avaliadas.

Os valores na tabela são representativos de processadores super escalares correntes. Como visto no Capítulo 1, o Intel Pentium e o DEC Alpha 21064 despacham duas instruções por ciclo, enquanto o IBM RS/6000 e o PowerPC 604 podem despachar até quatro instruções. Avanços na tecnologia certamente possibilitarão arquiteturas com larguras de despacho maiores, e assim também foram consideradas configurações onde são despachadas seis e oito instruções por ciclo. O Intel Pentium e o HP PA7100 incorporam duas unidades funcionais homogêneas. Outros processadores, como o DEC Alpha 21064 e o PowerPC 604 possuem quatro unidades funcionais, enquanto o Motorola M88110 inclui dez unidades funcionais. Embora nestes três últimos processadores as unidades funcionais sejam especializadas, ocupando individualmente uma área de integração menor que a de uma unidade funcional que executa todas as instruções disponíveis, é razoável supor implementações futuras com um número equivalente de unidades funcionais homogêneas. Assim, aqui foram consideradas configurações com até oito unidades funcionais homogêneas. O número de unidades virtuais foi estipulado com base em resultados preliminares aos aqui apresentados, os quais mostraram que não há vantagem, no modelo bloqueante, em incluir um número elevado de unidades virtuais. Por este motivo, foram consideradas até quatro unidades virtuais associadas a cada unidade funcional.

Em todas as configurações de arquitetura, considerou-se uma memória *cache* de instruções com oito bancos físicos, possibilitando uma largura de busca de oito instruções por ciclo (esta é a largura de busca encontrada na arquitetura PowerPC). Sob o ponto de vista de organização lógica, a memória *cache* de instruções possui 64 bytes por li-

nha, com 4 linhas por conjunto e 128 conjuntos, totalizando 32 Kbytes. A memória *cache* de dados possui a mesma organização lógica. Processadores super escalares reais apresentam uma grande largura de banda para a memória principal, graças a um amplo barramento de dados externo. Por exemplo, o DEC Alpha 21064 possui um barramento de dados de 128 bits (16 bytes) [McLellan 93]. Para todas as configurações de arquitetura, supõe-se um barramento de dados com esta largura, e ainda a existência de um sub-sistema de memória com *cache* secundária onde o primeiro acesso consome quatro ciclos e acessos subseqüentes a endereços consecutivos consomem um ciclo cada. Com estes valores, são necessários sete ciclos para carregar uma linha na memória *cache* primária. Também em todas as configurações, foi usada uma fila de instruções com 32 entradas. Os tamanhos da largura de busca e da fila de instruções foram escolhidos para minimizar a ocorrência de bloqueios no despacho por falta de instruções na fila. Com isto, procurou-se enfatizar o papel dos parâmetros que mais diretamente determinam o paralelismo da arquitetura - a largura de despacho e o número de unidades funcionais e virtuais.

5.2 Resultados para os Progamas de Teste Inteiros

Os resultados nesta seção foram obtidos com os programas de teste inteiros apenas, sendo que os resultados para os programas de ponto flutuante serão apresentados na próxima seção. Para manter no texto um volume de informações aceitável, os gráficos e tabelas neste capítulo representam a média dos valores individuais, obtidos com cada programa de teste.

Os gráficos na Figura 5.1 fornecem uma visão geral do desempenho do modelo bloqueante homogêneo. Os gráficos correspondem a larguras de despacho diferentes e, em cada gráfico, as curvas correspondem a configurações com diferentes números de unidades virtuais *por unidade funcional.* Como mostram os gráficos, foi obtido um aumento no desempenho de até 64% em relação à arquitetura referência. Note que as configurações com apenas uma unidade funcional e uma unidade virtual constituem uma exceção, apresentando uma degradação no desempenho em relação à arquitetura referência. Nestas configurações, a unidade funcional permanece ociosa no ciclo seguinte ao término da execução de uma instrução. Isto acontece porque a única unidade virtual existente fica ocupada até que seja propagado o resultado da instrução completada, impedindo o despacho de uma nova instrução naquele ciclo. Desta forma, a utilização da unidade funcional é de no máximo 50%. O descarte de instruções aces-

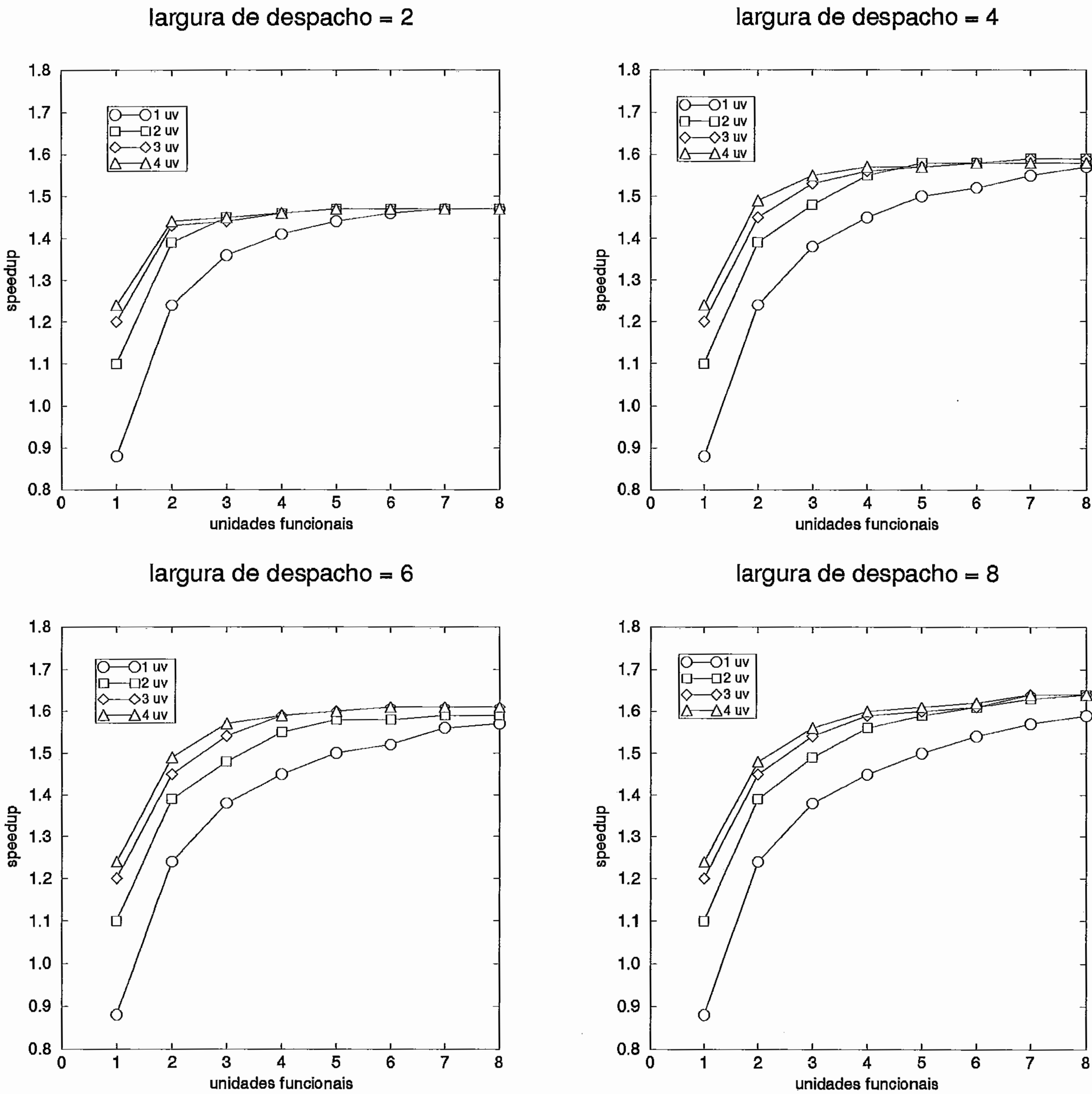


Figura 5.1: Ganho de desempenho com o modelo bloqueante homogêneo.

sadas erroneamente após um desvio também contribui para reduzir a utilização média da unidade funcional. A combinação destes fatores resulta em uma degradação do desempenho da arquitetura super escalar em relação a arquitetura referência.

Os gráficos na Figura 5.2 mostram o comportamento do desempenho em relação a largura de despacho. Cada gráfico corresponde a um certo número de unidades virtuais por unidade funcional, e em cada gráfico as curvas correspondem a configurações com um número de unidades funcionais diferente. A partir dos gráficos nesta figura e aqueles na Figura 5.1, é possível verificar o balanço que existe entre a largura de despacho e o número de unidades funcionais. Observa-se nos gráficos da Figura 5.2 que larguras de despacho maiores que duas instruções produzem algum ganho de desempenho somente quando existem pelo menos três unidades funcionais. Não há vantagem em aumentar a largura de despacho em configurações com um número menor de unidades funcionais (mesmo acrescentando-se várias unidades virtuais às unidades funcionais) porque neste caso o desempenho torna-se limitado basicamente pelo baixo grau de paralelismo espacial da arquitetura, e não pelo volume de instruções enviadas para execução. Por outro lado, pelos gráficos da Figura 5.1, vê-se que em configurações com largura de despacho de duas instruções, não há ganho quando são incluídas mais que três unidades funcionais. Com larguras de despacho de quatro e seis instruções, o desempenho estabiliza-se a partir de quatro unidades funcionais. Com uma largura de despacho de oito instruções, o desempenho de fato estabiliza-se com sete unidades funcionais mas, a partir de quatro unidades funcionais, o acréscimo de novas unidades resulta em um ganho muito pequeno.

De uma maneira geral, observa-se que o desempenho não é muito sensível a aumentos na largura de despacho, mesmo em configurações com um grande número de recursos. Por exemplo, em configurações com oito unidades funcionais e quatro unidades virtuais, verifica-se um ganho de apenas 14% quando a largura de despacho passa de duas para oito instruções. Isto acontece porque a largura de despacho disponível não é completamente utilizada. Este fato fica evidente no gráfico da Figura 5.3, que mostra a distribuição do número de instruções despachadas por ciclo. Para evitar limitações no despacho devido à disponibilidade de recursos, este gráfico foi obtido usando-se configurações com oito unidades funcionais e quatro unidades virtuais por unidade funcional. No gráfico estão incluídos apenas valores acima de 1%.

Como mostra o gráfico, larguras de despacho de duas e quatro instruções são completamente utilizadas em 38% e 16% dos ciclos, respectivamente. Uma largura de despacho de seis instruções é completamente utilizada em somente 8% dos ciclos,

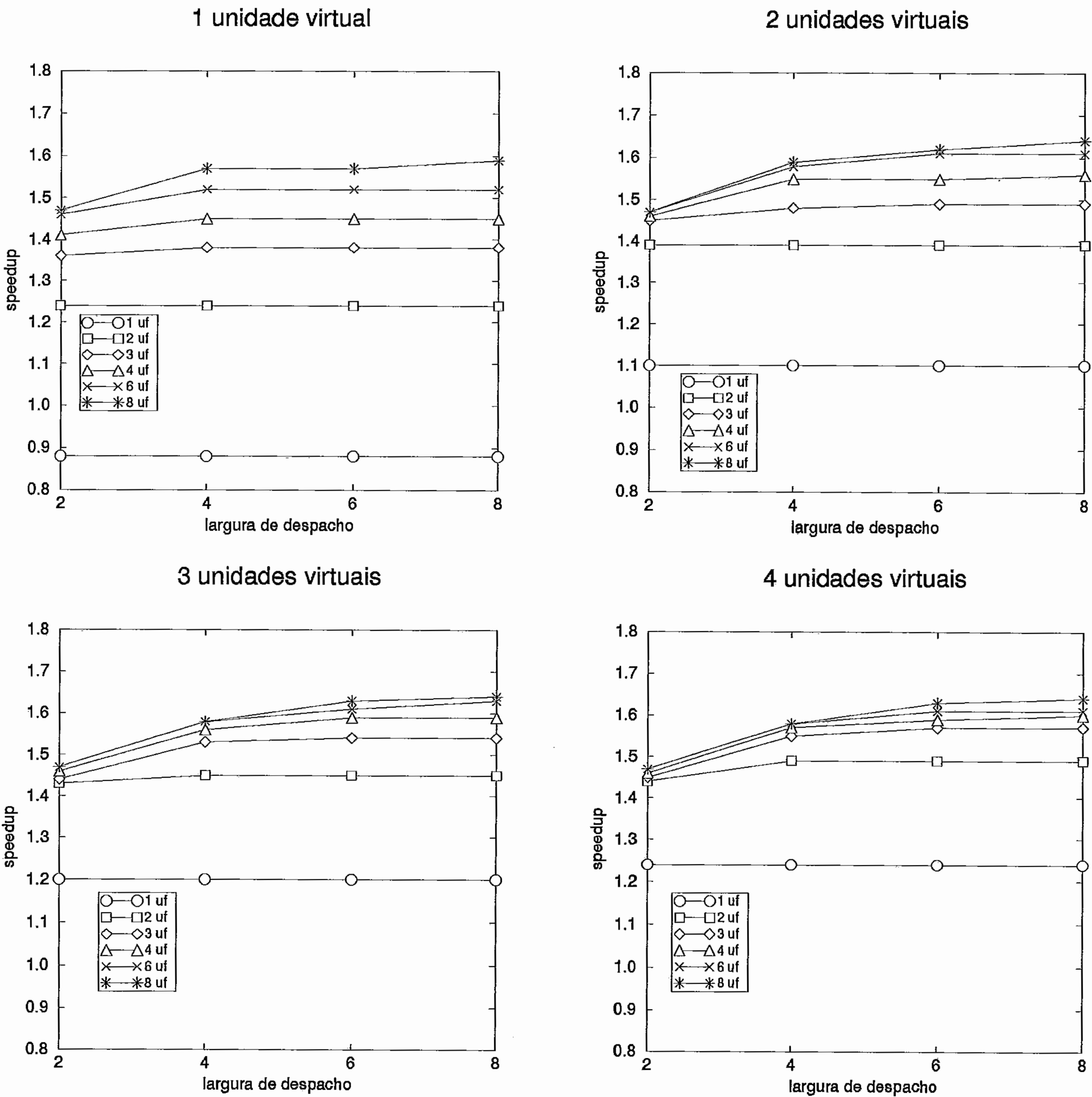


Figura 5.2: Efeito da largura de despacho.

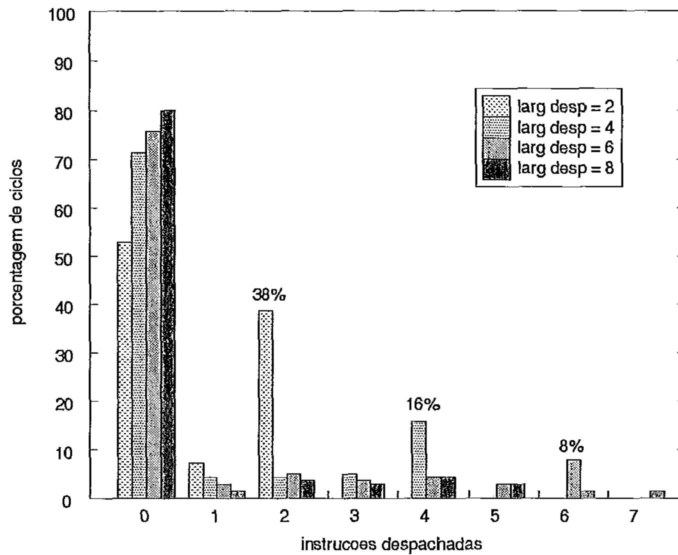


Figura 5.3: Distribuição do número de instruções despachadas por ciclo.

enquanto a largura de despacho de oito instruções é completamente aproveitada em menos de 1% dos ciclos. Estes números explicam porque o ganho no desempenho é mais significativo quando a largura de despacho aumenta de duas para quatro instruções (veja Figura 5.1), pois neste caso a contribuição para o volume de instruções despachadas é comparativamente maior. O gráfico na Figura 5.3 também indica que, quanto maior a largura de despacho, menor a sua taxa de aproveitamento. Isto acontece porque na medida que a largura despacho aumenta, é maior a probabilidade de surgir uma instrução de desvio que bloqueie o envio das instruções seguintes naquele mesmo ciclo. A baixa utilização da largura de despacho é apenas um dos efeitos das dependências de controle no modelo bloqueante. A influência das dependências de controle neste modelo será discutida mais à frente.

Uma maneira de avaliar o efeito do número de unidades funcionais é através da taxa de instruções completadas por ciclo (ipc). O gráfico na Figura 5.4 mostra a distribuição do número de instruções completadas por ciclo. Para evitar uma limitação no ipc pelo volume de instruções despachadas e pela disponibilidade de recursos, este gráfico foi obtido usando-se configurações com largura de despacho de oito instruções e com quatro unidades virtuais por unidade funcional. No gráfico estão incluídos apenas valores acima de 1%.

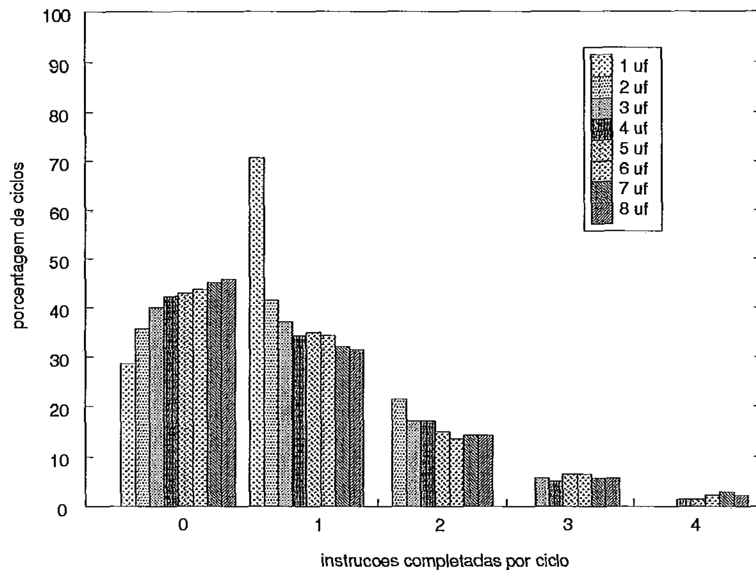


Figura 5.4: Distribuição do número de instruções completadas por ciclo.

O primeiro ponto que se observa neste gráfico é a alta proporção de ciclos onde nenhuma instrução é completada, atingindo cerca de 44% em configurações com oito unidades funcionais. Isto apenas reflete a baixa utilização da largura de despacho disponível. Como visto na Figura 5.3, para uma largura de despacho de oito instruções, em cerca de 80% dos ciclos nenhuma instrução é despachada, limitando a utilização das unidades funcionais e por conseguinte o ipc. Assim, a baixa proporção de instruções completadas por ciclo é um outro efeito das dependências de controle no modelo bloqueante. Observe que a proporção de ciclos onde nenhuma instrução é completada aumenta com o número de unidades funcionais. Isto acontece porque o pequeno volume de instruções despachadas passa a ser distribuído entre um número maior de unidades funcionais, o que diminui ainda mais a ocupação de cada unidade funcional. Isto é confirmado pela Tabela 5.2, que mostra a utilização média das unidades funcionais. Esta tabela foi obtida usando-se configurações com largura de despacho de oito instruções, e com quatro unidades virtuais por unidade funcional.

Os resultados sobre o ipc e a utilização das unidades funcionais indicam que não é vantajoso incluir mais que quatro unidades funcionais no modelo bloqueante. Primeiro, a proporção de ciclos em que mais de quatro instruções são completadas simultaneamente é insignificante (menor que 1%). Segundo, a utilização de cada unidade

Número de Unidades Funcionais	Utilização Média
1	71 %
2	42 %
3	30 %
4	23 %
5	18 %
6	15 %
7	13 %
8	12 %

Tabela 5.2: Utilização das unidades funcionais.

funcional fica abaixo dos 20% quando mais do que quatro unidades funcionais são incluídas, não compensando o custo de unidades adicionais. A princípio, estes resultados parecem desanimadores quanto ao benefício de se acrescentar unidades funcionais à uma arquitetura super escalar. No entanto, note que as unidades funcionais ficam ociosas porque o despacho é bloqueado após cada instrução de desvio. Caso contrário, novas instruções despachadas manteriam as unidades funcionais ocupadas, e a resposta ao aumento do número de unidades seria mais satisfatória. A alternativa de não bloquear o despacho após instruções de desvio é investigada no Capítulo 7.

Por último, resta considerar o efeito do número de unidades virtuais. Os gráficos na Figura 5.5 mostram como o desempenho é afetado pelo número de unidades virtuais associadas a cada unidade funcional. Nesta figura, cada gráfico corresponde a uma diferente largura de despacho, e em cada gráfico as curvas correspondem a diferentes números de unidades funcionais.

O efeito do número de unidades virtuais é mais perceptível em configurações com poucas unidades funcionais. Por exemplo, em configurações com apenas uma unidade funcional, verifica-se um ganho de até 30% quando o número de unidades virtuais passa de uma para quatro unidades. Ao contrário, para configurações com quatro unidades funcionais, observa-se um ganho de no máximo 10% para uma mesma variação no número de unidades virtuais. Com poucas unidades funcionais, o papel

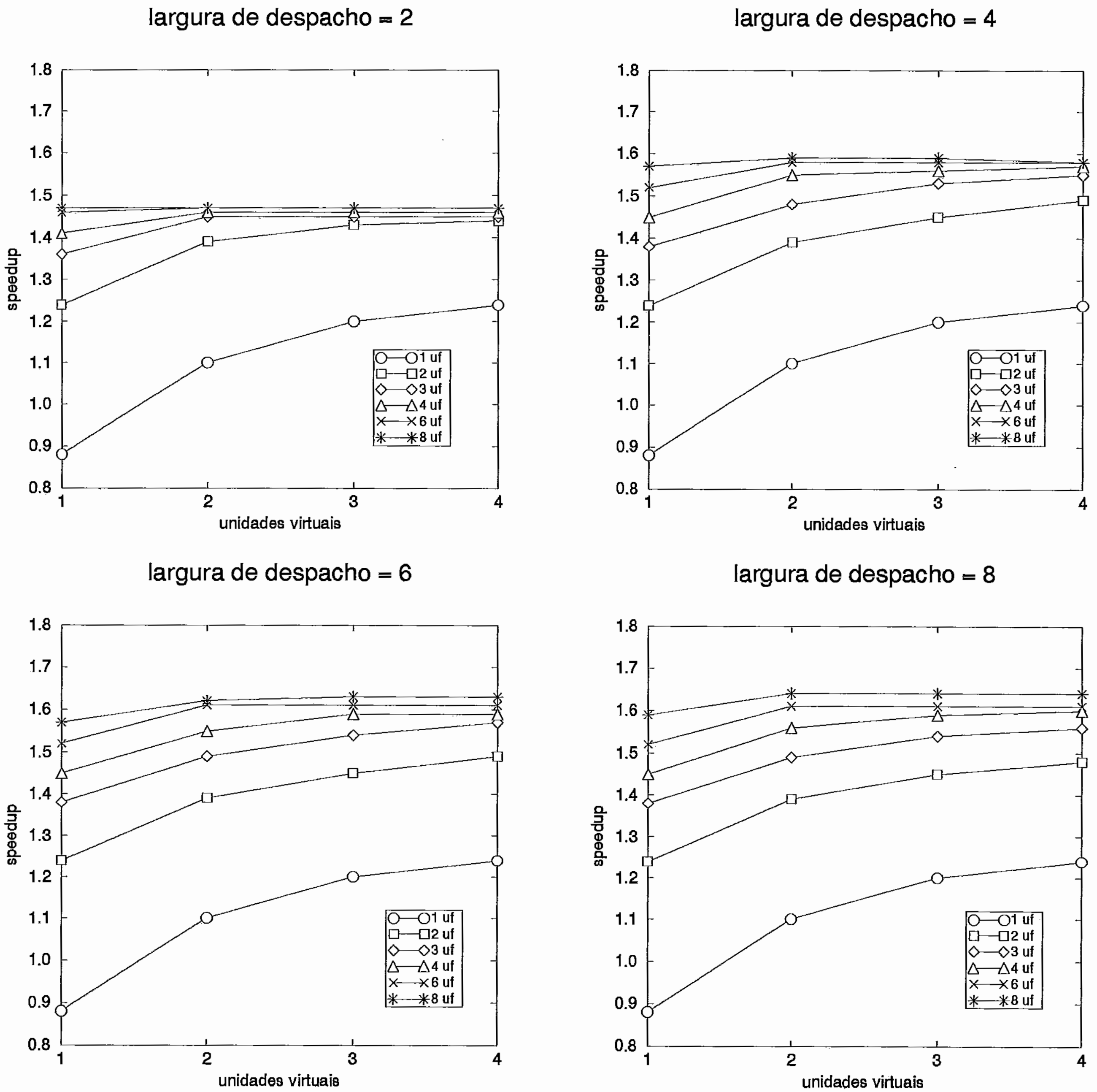


Figura 5.5: Efeito do número de unidades virtuais.

de armazenamento temporário das unidades virtuais é essencial para manter uma continuidade no despacho das instruções, e por isso o desempenho torna-se bastante sensível ao número de unidades virtuais. Este efeito das unidades virtuais pode ser visualizado da seguinte forma. Em configurações com poucas unidades funcionais, torna-se necessário várias unidades virtuais associadas a cada uma delas para armazenar verticalmente as instruções despachadas. Com um número maior de unidades funcionais, poucas unidades virtuais são suficientes para armazenar horizontalmente o mesmo número de instruções. A influência das unidades virtuais também depende, é claro, da largura de despacho, pois quanto maior o volume de instruções despachadas, mais importante se torna o efeito de armazenamento vertical. O gráfico da Figura 5.6 mostra a utilização média das unidades virtuais, em configurações com diferentes números de unidades funcionais.

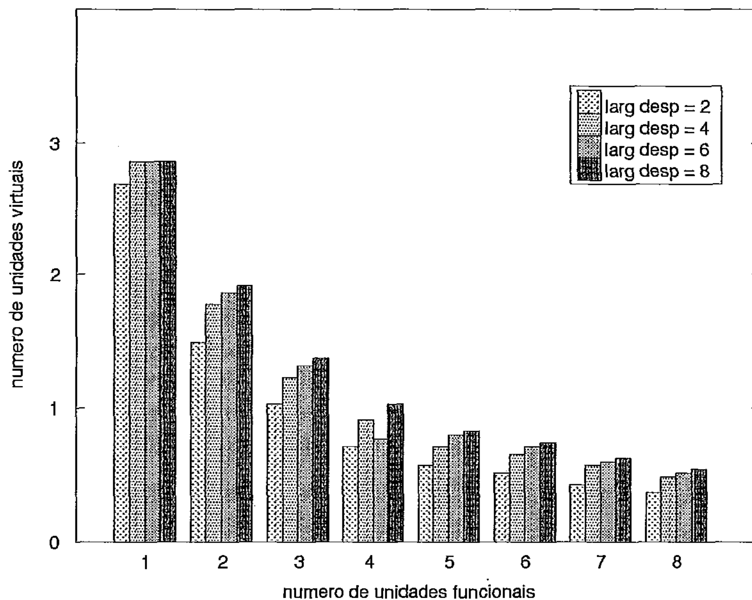


Figura 5.6: Utilização das unidades virtuais.

De uma maneira geral, este gráfico reflete as observações acima. O número de unidades virtuais usadas aumenta com a largura de despacho, principalmente para configurações com poucas unidades funcionais, devido ao armazenamento vertical. Com várias unidades funcionais o uso médio de unidades virtuais é menor, devido ao armazenamento horizontal. No pior caso, com uma largura de despacho de oito instruções e uma única unidade funcional, não foram usadas mais que três unidades virtuais.

Devido a baixa utilização da largura de despacho, o número de unidades virtuais efetivamente usadas é bem menor que o número máximo de instruções que podem ser despachadas.

5.3 Resultados para os Programas de Teste de Ponto Flutuante

Nesta seção é avaliado o desempenho do modelo bloqueante homogêneo para os programas de teste de ponto flutuante. Com esta separação, procurou-se deixar visíveis eventuais diferenças no desempenho em relação ao tipo de aplicação. Nesta avaliação, foram inicialmente consideradas configurações com um número fixo de oito unidades funcionais para instruções inteiras, cada uma destas com quatro unidades virtuais. Com isto procurou-se evitar limitações quanto à execução das instruções inteiras. Foram avaliadas configurações com até oito unidades funcionais de ponto flutuante, cada uma com até quatro unidades virtuais.

Os gráficos na Figura 5.7 mostram o ganho de desempenho em função do número de unidades de ponto flutuante. Cada gráfico corresponde a uma certa largura de despacho, e cada curva em um gráfico corresponde a um certo número de unidades virtuais por unidade funcional.

Como mostram os gráficos, o desempenho estabiliza-se a partir de quatro unidades funcionais de ponto flutuante, e o efeito das unidades virtuais é significativo apenas quando passa-se de uma para duas unidades virtuais (por unidade funcional). A largura de despacho possui algum efeito apenas quando passa de duas para quatro instruções despachadas por ciclo, mas este efeito é menos significativo que nos programas inteiros. O ponto que mais se destaca no gráfico da Figura 5.7 é o desempenho sensivelmente maior que o obtido com os programas inteiros. Esta diferença pode ser explicada a partir da análise de alguns trechos de código *assembly* dos programas de ponto flutuante aqui usados. Observa-se que o volume de instruções inteiras independentes é maior que nos próprios programas de teste inteiros. Um exemplo típico está nas instruções de acesso à memória. São freqüentes seqüências de várias instruções *load* que carregam dados de precisão dupla em registradores (os programas de teste de ponto flutuante aqui usados realizam apenas aritmética de precisão dupla). Em geral, as instruções utilizam modos de endereçamento relativo à base ou indexado, e carregam valores em registradores diferentes. Assim, estas instruções são independentes

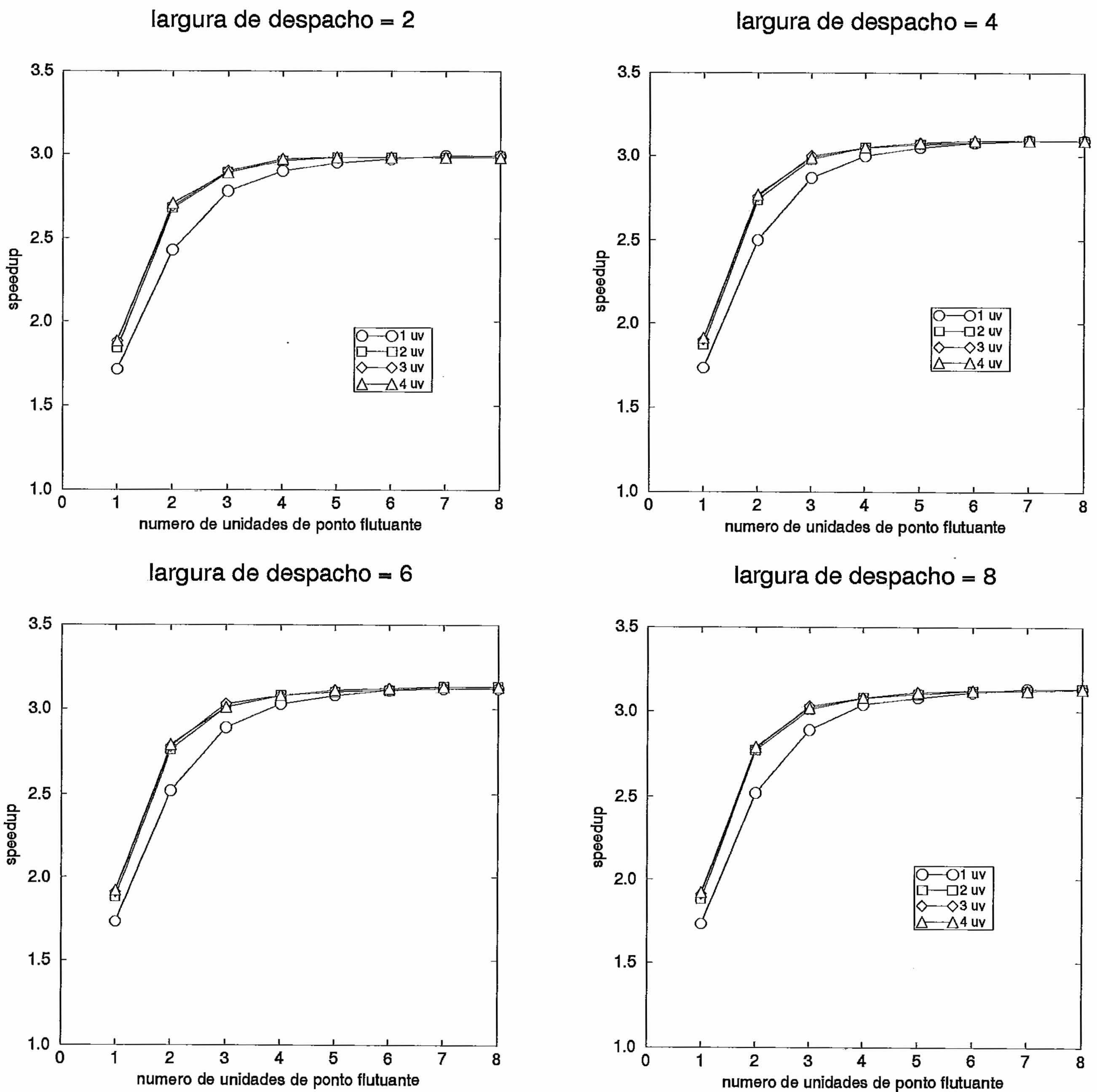


Figura 5.7: Desempenho do modelo bloqueante homogêneo em relação ao número de unidades de ponto flutuante.

entre si, e podem ser executadas em paralelo. O mesmo acontece com as instruções *store* que armazenam os resultados das operações aritméticas de ponto flutuante. A existência destas seqüências de instruções confere aos programas de ponto flutuante um maior nível de paralelismo de instrução.

No entanto, o principal motivo para o ganho maior está no fato que a proporção de instruções de desvio nos programas de ponto flutuante é bem menor que nos programas inteiros. Isto fica evidente na Tabela 5.4, que mostra a porcentagem de instruções de desvio dentro dos 10 milhões de instruções executadas. Em conseqüência do menor número de desvios, a freqüência de bloqueio do despacho também é menor, o que aumenta a utilização das unidades funcionais e o número de instruções completadas por ciclo. Além disso, com uma freqüência menor de desvios, os blocos básicos tornam-se maiores, aumentando a eficiência do mecanismo de escalonamento dinâmico de instruções.

Tendo em vista esta observação, é interessante observar o comportamento do desempenho dos programas de ponto flutuante em relação ao número de unidades funcionais para instruções inteiras. Devido à menor freqüência de bloqueio do despacho, a disponibilidade de recursos para a execução das instruções inteiras deve ter um efeito significativo sobre o desempenho. Para verificar esta hipótese, foram avaliadas configurações com o número de unidades funcionais inteiras variando entre uma e oito unidades, e com até quatro unidades virtuais em cada uma destas unidades funcionais. Nestas configurações foi incluído um número fixo de quatro unidades funcionais de ponto flutuante, cada uma delas com duas unidades virtuais. A largura de despacho foi fixada em oito instruções. A Figura 5.8 mostra o desempenho obtido em função do número de unidades funcionais inteiras. Neste gráfico, cada curva corresponde a um certo número de unidades virtuais por unidade funcional inteira.

Como esperado, o número de unidades funcionais inteiras possui um efeito mais sensível sobre o desempenho. Enquanto para os programas inteiros o desempenho estabiliza-se já com quatro unidades funcionais inteiras, agora isto acontece somente quando são incluídas seis unidades funcionais. Para os programas de ponto flutuante, isto ocorre porque as limitações impostas pelas dependências de controle são menos severas. As curvas da Figura 5.8, quando comparadas com as curvas de desempenho para os programas inteiros, constituem um exemplo real que reforça o ponto que as dependências de controle podem ser mais significativas do que as dependências de dados no sentido de limitar o desempenho de uma arquitetura super escalar. O aspecto das dependências de controle é focalizado na seção a seguir e mais à frente, nos

Programa	Número de Instruções de Desvio
Inteiros	
espresso	20%
eqntott	22%
compress	11%
gcc	20%
Ponto Flutuante	
doduc	5%
tomcatv	4%
nasa	3%
fpppp	1%

Tabela 5.3: Porcentagem de instruções de desvio nos programas de teste.

Capítulos 6 e 7.

5.4 As Dependências de Controle no Modelo Bloqueante

No modelo bloqueante, as dependências de controle possuem efeitos particularmente severos. Como visto ao longo deste capítulo, estes efeitos manifestam-se em vários aspectos da arquitetura. O primeiro deles é na utilização da largura de despacho. O número de instruções que são de fato despachadas a cada ciclo é pequeno, pois o aparecimento de uma instrução de desvio impede o despacho das instruções que completariam a largura disponível. O segundo efeito é na utilização das unidades funcionais. Como o volume de instruções entregue às unidades funcionais é pequeno, as unidades funcionais permancecem a maior parte do tempo ociosas, e por isso o número de instruções completadas por ciclo torna-se bastante limitado.

É interessante comparar os efeitos das dependências de dados com os resultantes das dependências de controle. Em uma arquitetura baseada no algoritmo de Toma-

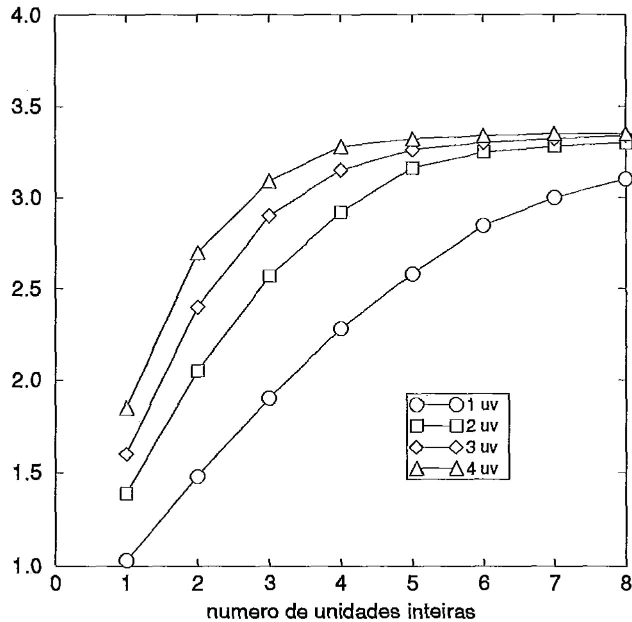


Figura 5.8: Desempenho em relação ao número de unidades funcionais inteiras, para os programas de teste de ponto flutuante.

sulo, uma unidade funcional pode permanecer ociosa por dois motivos. Primeiro, a unidade funcional está ociosa porque todas as instruções em suas unidades virtuais estão esperando por operandos. Segundo, a unidade funcional está ociosa porque não existem instruções em suas unidades virtuais. A primeira condição é decorrente das dependências de dados, enquanto a segunda condição ocorre quando o volume de instruções despachadas não é suficiente para manter as unidades funcionais ocupadas. Por sua vez, isto pode acontecer porque a janela de instruções encontra-se freqüentemente vazia, ou porque o despacho é constantemente bloqueado devido às dependências de controle.

A Tabela 5.4 mostra, para os programas de teste inteiros, a proporção de ciclos em que as unidades funcionais permanecem ociosas por falta de instruções ou por instruções esperando operandos. A tabela também mostra a proporção de ciclos em que o despacho é bloqueado por janela de instruções vazia e por instruções de desvio. Estes valores foram obtidos a partir de configurações com largura de despacho de oito

instruções, e com oito unidades funcionais e quatro unidades virtuais.

Programa	Ociosa por Operando	Ociosa por Instrução	Bloqueio por Fila Vazia	Bloqueio por Desvios	*Speedup*
espresso	30 %	59 %	1.0 %	81 %	1.43
eqntott	28 %	60 %	1.2 %	79 %	1.35
compress	41 %	43 %	0.5 %	79 %	1.98
gcc	30 %	60 %	1.0 %	81 %	1.65

Tabela 5.4: Taxas de ociosidade e de bloqueio.

Com exceção de um único programa, a taxa de ociosidade devido à falta de instruções é quase o dobro da taxa de ociosidade devido a instruções esperando operandos. A exceção é o programa *compress*, justamente o que apresenta o maior ganho. Para todos os programas, a freqüência de bloqueio do despacho devido à janela de instruções vazia é praticamente insignificante frente à taxa de bloqueio por instruções de desvio. Estes números indicam que a baixa utilização das unidades funcionais deve-se principalmente à falta de instruções nas unidades virtuais, e que isto decorre sobretudo do bloqueio do despacho provocado pelas dependências de controle.

No modelo bloqueante, o tratamento das dependências de controle é decisivo para o desempenho. No próximo capítulo, é apresentado o desempenho do modelo bloqueante com unidades funcionais heterogêneas. Como será visto, uma das preocupações no modelo heterogêneo será exatamente minimizar as limitações impostas pelas dependências de controle, de forma a possibilitar níveis mais altos de desempenho.

Capítulo 6

Avaliação do Modelo Bloqueante Heterogêneo

No capítulo anterior apresentamos os resultados experimentais obtidos com o modelo bloqueante com unidades funcionais homogêneas. Como visto no Capítulo 1, em algumas arquiteturas reais as unidades funcionais são heterogêneas, ou seja, cada unidade funcional executa apenas um sub-conjunto de instruções. Neste capítulo é avaliado o desempenho de um modelo de arquitetura super escalar com unidades funcionais especializadas. O objetivo aqui é verificar a influência de cada tipo de unidade funcional, bem como determinar a combinação de diferentes tipos de unidades funcionais que maximiza o desempenho.

No modelo com unidades funcionais heterogêneas, o despacho de instruções também é bloqueado quando uma instrução de desvio é encontrada. No entanto, a introdução de unidades funcionais especializadas será aproveitada para reduzir o efeito negativo das dependências de controle sobre o desempenho.

6.1 Configurações do Modelo Heterogêneo

Como mencionado no Capítulo 4, para os modelos de arquiteturas super escalar considerados neste trabalho foi adotado o conjunto de instruções da arquitetura SPARC. A arquitetura SPARC fornece quatro tipos de instruções: instruções aritméticas e lógicas sobre inteiros, instruções de transferência de controle, instruções de acesso à memória,

e instruções de ponto flutuante (ver Apêndice A). Assim, é natural considerar-se unidades funcionais heterogêneas que executam estas classes de instruções. Foram então considerados os seguintes grupos de diferentes configurações de unidades funcionais:

- no primeiro grupo, denominado Tipo D, a arquitetura possui dois tipos de unidades funcionais: uma unidade de desvio, específica para a execução de instruções de desvio, e unidades funcionais que executam os demais tipos de instruções. Nestas configurações, foi incluída apenas uma unidade de desvio, já que no máximo uma instrução de desvio está em execução a cada momento. Foi incluída apenas uma unidade virtual associada à unidade de desvio, para garantir que as instruções de desvio sejam executadas em ordem;

- no segundo grupo, denominado Tipo M, a arquitetura inclui três tipos de unidades funcionais: a unidade de desvio, unidades de memória que executam as instruções de acesso à memória, e unidades funcionais idênticas que executam as instruções aritméticas e lógicas;

- no terceiro grupo, denominado Tipo AL, são também incluídas as unidades de desvio e de memória. No entanto, as unidades que executam as instruções aritméticas e lógicas são diferentes, cada uma delas executando um sub-conjunto destas instruções. Como discutido adiante, a combinação destas unidades será orientada pelo perfil da utilização dinâmica das instruções.

- os três grupos de configurações acima são usados para os programas de teste inteiros. Para os programas de teste de ponto flutuante, é considerado um quarto grupo, denominado Tipo F, que inclui os tipos de unidades funcionais encontradas nas configurações Tipo M, e mais três tipos de unidades funcionais para instruções de ponto flutuante. Estas são: unidades para soma/subtração, unidades para multiplicação/divisão, e unidades para conversão entre valores inteiros e em ponto-flutuante e para conversão de valores em ponto flutuante com precisões diferentes.

6.2 O Modelo Bloqueante com Unidade de Desvio

Conforme mencionado no Capítulo 5, as dependências de controle representam a principal limitação para o desempenho do modelo bloqueante. A maneira mais eficaz para eliminar este problema consiste, obviamente, em não bloquear o despacho ao ser encontrada uma instrução de desvio. Esta será a solução adotada no modelo

especulativo, avaliado no próximo capítulo. No entanto, isto requer a introdução de mecanismos que, na prática, aumentam a complexidade de implementação da arquitetura. Assim, é interessante investigar alternativas mais simples, que não exijam um suporte agressivo de *hardware*. A inclusão de unidades funcionais heterogênas dá a oportunidade para introduzir-se modificações que reduzem o impacto das dependências de controle no modelo bloqueante.

Em um modelo de arquitetura super escalar como o aqui considerado, é possível identificar três componentes no custo das instruções de desvio. O primeiro deles é a **latência de condição**. A execução de uma instrução de desvio condicional envolve o teste dos bits de condição, e assim pode ser necessário esperar pelo término de alguma instrução aritmética ou lógica anterior que atualiza o estado dos bits de condição antes que o desvio possa ser executado. Em desvios incondicionais, esta latência não existe. O segundo componente é a **latência de início**. A partir do momento em que o código de condição é atualizado, a instrução de desvio não entra imediatamente em execução. Nos modelos aqui considerados, uma instrução armazenada em uma unidade virtual entra em execução apenas no ciclo seguinte em que seus operandos ficaram disponíveis, e assim a latência de início das instruções de desvio é de pelo menos um ciclo. A latência de início pode ser maior, dependendo se a unidade funcional que excuta o desvio estiver ocupada. O terceiro componente é a **latência de término**. Uma instrução de desvio consome um ciclo para ser executada por uma unidade funcional. No entanto, um ciclo adicional é necessário para que o resultado (i.e., o endereço da próxima instrução a ser executada) seja armazenado no registrador destino (o contador de programa). Assim, uma instrução de desvio possui uma latência de término de dois ciclos.

A eliminação ou redução destas latências diminui o intervalo de tempo durante o qual o despacho de instruções permanece bloqueado. Com isto instruções passam a ser despachadas em ciclos onde isto não acontecia, aumentando a utilização das unidades funcionais e por conseguinte o número de instruções completadas por ciclo.

Uma alternativa simples para reduzir o custo das dependências de controle foi adotada nas arquiteturas RS/6000 e PowerPC. Nestas arquiteturas, uma instrução de desvio pode ser despachada e executada em um único ciclo. Desvios incondicionais são sempre executados no mesmo ciclo em que são despachados. Desvios condicionais também podem ser executados no ciclo do despacho, se os bits de condição já estiverem atualizados naquele momento. Este esquema é chamado *zero-cycle branch* porque, se executado no ciclo do despacho, a instrução de desvio não consome ciclos adicionais

na sua execução.

Para avaliar o ganho obtido com esta solução, introduzimos uma unidade de desvio que realiza o despacho de instruções e também executa as instruções de desvio. Esta unidade despacha instruções, exceto desvios, para as outras unidades funcionais, e despacha desvios para ela mesma. Desvios incondicionais, e os desvios condicionais que encontram os bits de condição prontos, são executados pela unidade de desvio no mesmo ciclo em que ocorreu o despacho. Para possibilitar o despacho e a execução de desvios em um mesmo ciclo, considera-se que o contador de programa faz parte da unidade de desvios. Desta forma, é eliminada a comunicação do endereço destino através do CDB, e o contador de programa pode ser atualizado no mesmo ciclo em que o endereço destino é determinado. Com a introdução da unidade de desvio operando desta maneira, as latências de início e de término das instruções de desvio são eliminadas.

As Figuras 6.1 e 6.2 mostram o desempenho obtido com as configurações que incluem a unidade de desvio. Para fins de comparação, os gráficos à esquerda mostram o desempenho fornecido pelo modelo bloqueante homogêneo, apresentados no capítulo anterior. As curvas apresentam o mesmo comportamento observado no modelo homogêneo, mas os níveis de desempenho atingidos são diferentes. Agora, o menor ganho em relação à arquitetura referência é de 10%, e não mais ocorre a degradação de desempenho nas configurações com uma unidade funcional e uma unidade virtual. O maior ganho chega a 97%, versus o ganho máximo de 64% no modelo homogêneo. Como mencionado, este aumento no desempenho é uma conseqüência do menor tempo durante o qual o despacho de instruções permanece bloqueado. Nas configurações com unidade de desvio, obtém-se uma uma redução de 4% a 12% na freqüência de bloqueio do despacho. Mesmo uma redução relativamente pequena na taxa de bloqueio possui um efeito considerável sobre o desempenho, pois tal redução representa a oportunidade de despachar várias instruções em ciclos onde nenhuma instrução era despachada.

Como visto no capítulo anterior, no modelo bloqueante as dependências de controle limitam severamente a utilização da largura de despacho. O gráfico à direita na Figura 6.3 mostra a distribuição do número de instruções despachadas por ciclo no modelo com unidade de desvio. Este gráfico foi obtido usando-se configurações com oito unidades funcionais e quatro unidades virtuais. À esquerda nesta figura é repetido o gráfico com a distribuição de instruções despachadas no modelo bloqueante homogêneo.

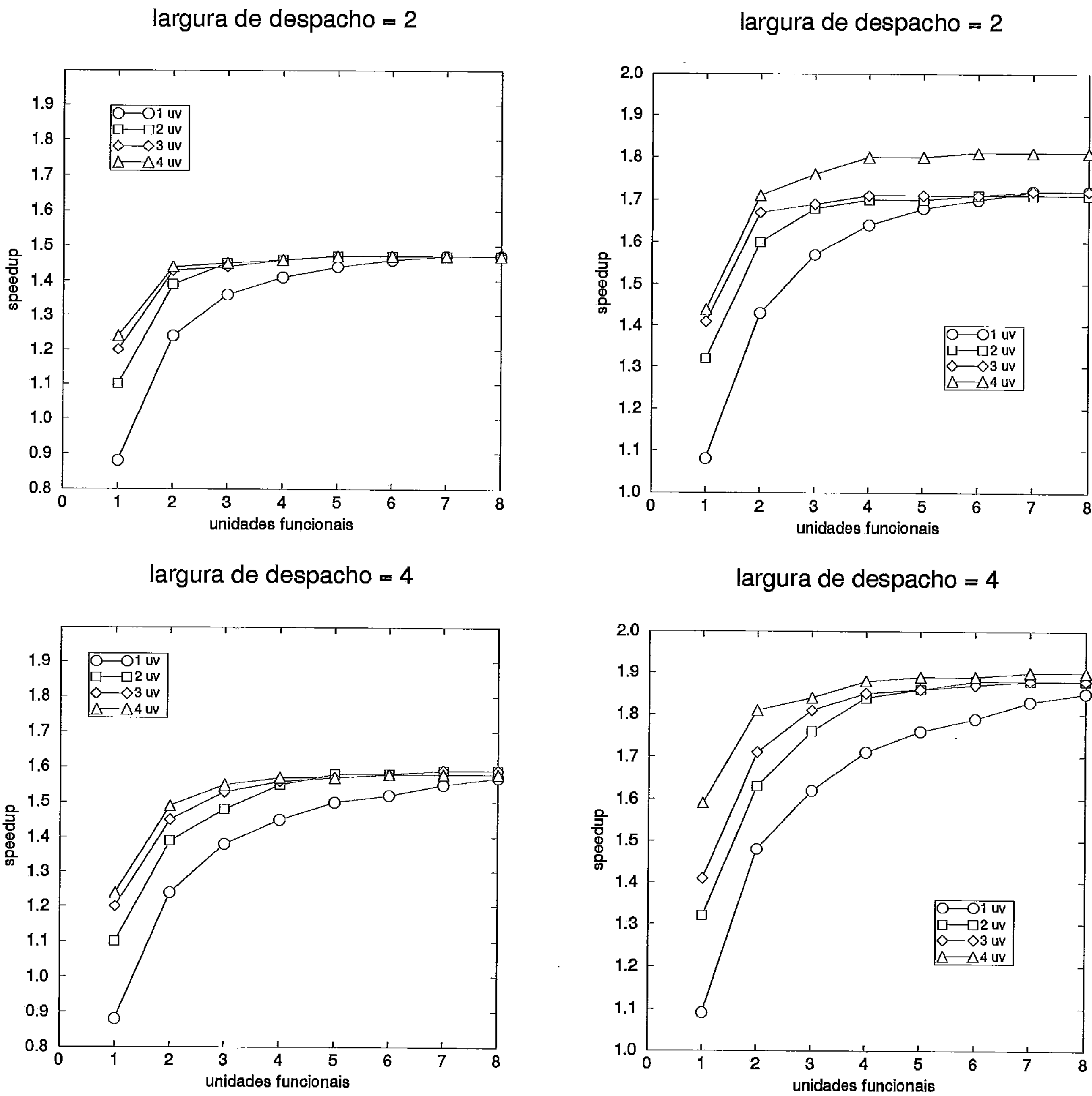


Figura 6.1: Desempenho do modelo bloqueante com unidade de desvio, comparado com o modelo bloqueante homogêneo.

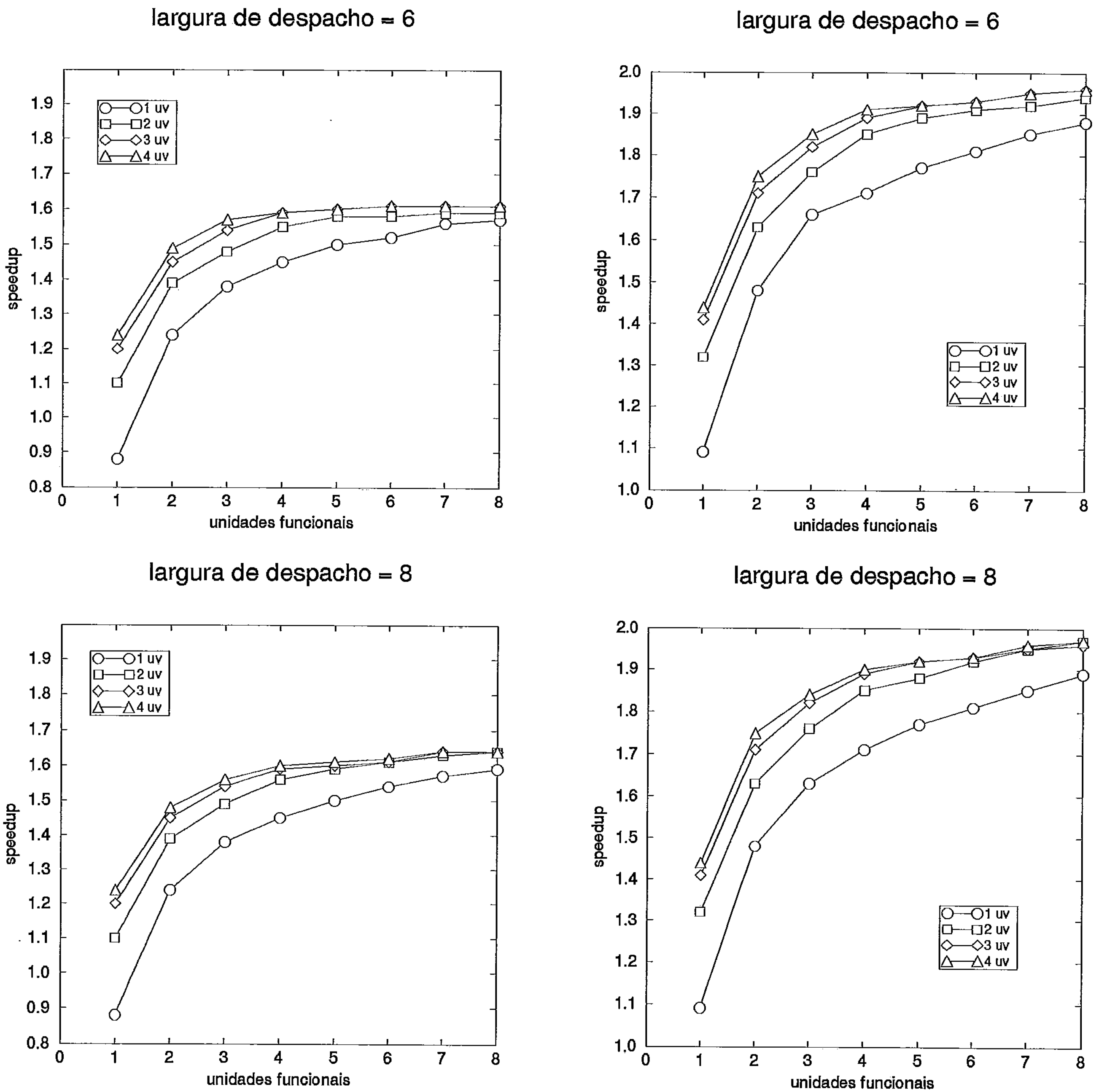


Figura 6.2: Desempenho do modelo bloqueante com unidade de desvio, comparado com o modelo bloqueante homogêneo (continuação).

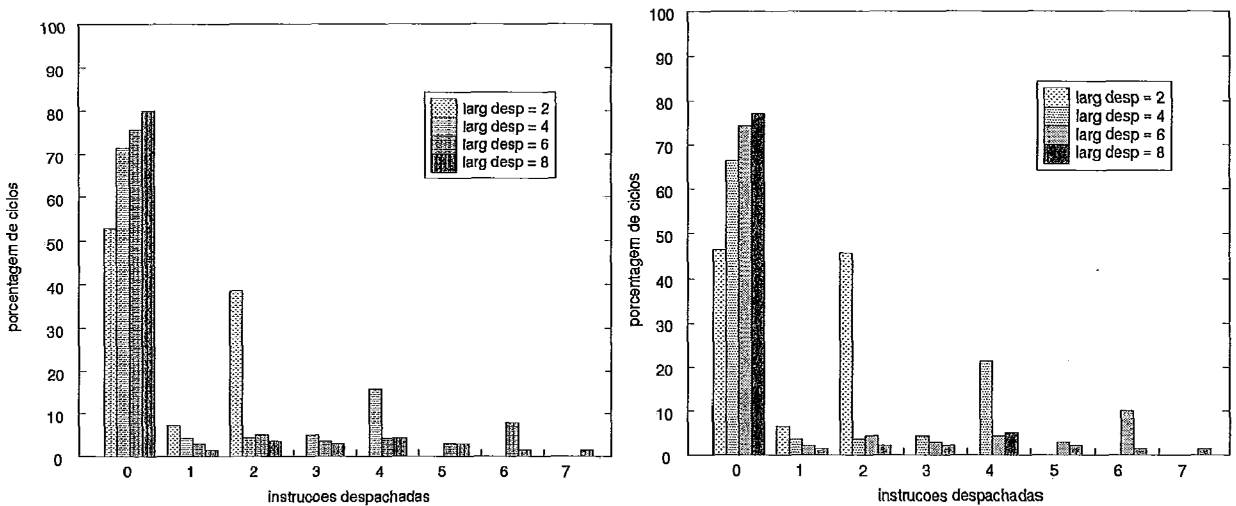


Figura 6.3: Distribuição de instruções despachadas para os modelos homogêneo e heterogêneo com unidade de desvio.

Verifica-se um aumento na porcentagem de ciclos onde a largura disponível é completamente utilizada. Para larguras de despacho de duas e quatro instruções, a utilização completa aumenta de 10% e 5%, respectivamente, em relação ao modelo sem unidade de desvio. Este aumento deve-se principalmente aos desvios incondicionais que agora, quando capturados em uma posição intermediária da largura de despacho, não impedem que as instruções restantes sejam despachadas. No entanto, ainda é alta a porcentagem de ciclos nos quais nenhuma instrução é despachada, refletindo as instruções de desvio condicionais que não encontram os bits de condição atualizados no momento do despacho. Nestes casos o despacho continua a ser bloqueado, resultando em ciclos subseqüentes onde nenhuma instrução é despachada. Como poderá ser observado mais adiante, a proporção de desvios incondicionais é bem menor que a de desvios condicionais (veja Figura 6.5), e por este motivo não há uma redução sensível na proporção de ciclos sem despacho.

6.3 O Modelo Bloqueante com Unidades de Memória

Vamos agora considerar as configurações do Tipo M, que além da unidade de desvio, incluem unidades separadas para a execução de instruções de acesso à memória.

Com esta configuração, podemos verificar qual a contribuição para o desempenho ao permitirmos múltiplos acessos simultâneos à memória. Inicialmente, foram usadas configurações com um número fixo de unidades funcionais que executam instruções aritméticas e lógicas. Em todas estas configurações, foram incluídas oito unidades funcionais deste tipo, cada uma delas com quatro unidades virtuais. A largura de despacho foi fixada em oito instruções por ciclo. Com isto, procurou-se isolar o efeito do número de unidades de memória, minimizando limitações quanto ao volume de instruções despachadas e quanto à capacidade de processamento de instruções aritméticas e lógicas.

O gráfico na Figura 6.4 mostra o desempenho em função do número de unidades de memória. Neste gráfico, cada curva corresponde a um certo número de unidades virtuais para cada unidade de memória. No gráfico também foi incluído o patamar de desempenho obtido com uma configuração heterogênea Tipo D, com oito unidades funcionais.

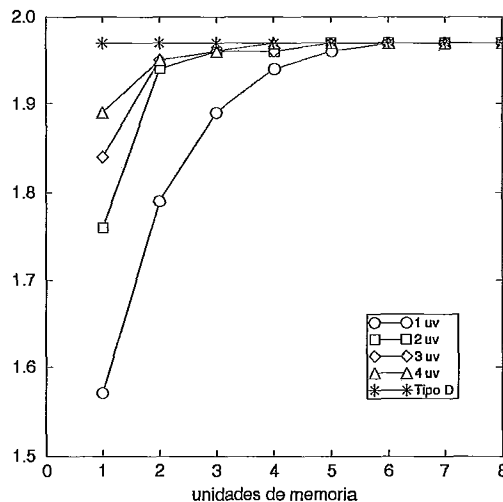


Figura 6.4: Desempenho para as configurações com unidades de memória.

Com apenas uma unidade de memória, observa-se uma redução de até 20% no desempenho em relação a uma configuração com oito unidades homogêneas. Um nível de desempenho equivalente ao da configuração do Tipo D é alcançado quando são incluídas duas unidades de memória com quatro unidades virtuais cada, e o mesmo

nível de desempenho é recuperado quando existem quatro unidades de memória com quatro unidades virtuais. De uma maneira geral, estes resultados mostram que o desempenho é bastante sensível quanto ao número de unidades de memória. Isto é uma decorrência da alta proporção de instruções de acesso à memória dentre as instruções executadas. O gráfico na Figura 6.5 mostra a distribuição dos tipos de instruções executadas pelos programas de teste inteiros. Como mostra a figura, existe uma alta proporção de instruções de acesso à memória, superada apenas pelas instruções ADD e SETHI. Em particular, o total de instruções *load* é quase o dobro das instruções *store*. As instruções *load* criam dependências para as instruções aritméticas e lógicas que utilizam os valores carregados. Além disso, estas instruções são normalmente independentes entre si, e podem ser executadas simultaneamente. Assim, na medida que aumenta o número de unidades de memória, possibilitando a execução de *loads* em paralelo, o tempo de espera das instruções dependentes diminui, o que tem uma conseqüência direta sobre o desempenho.

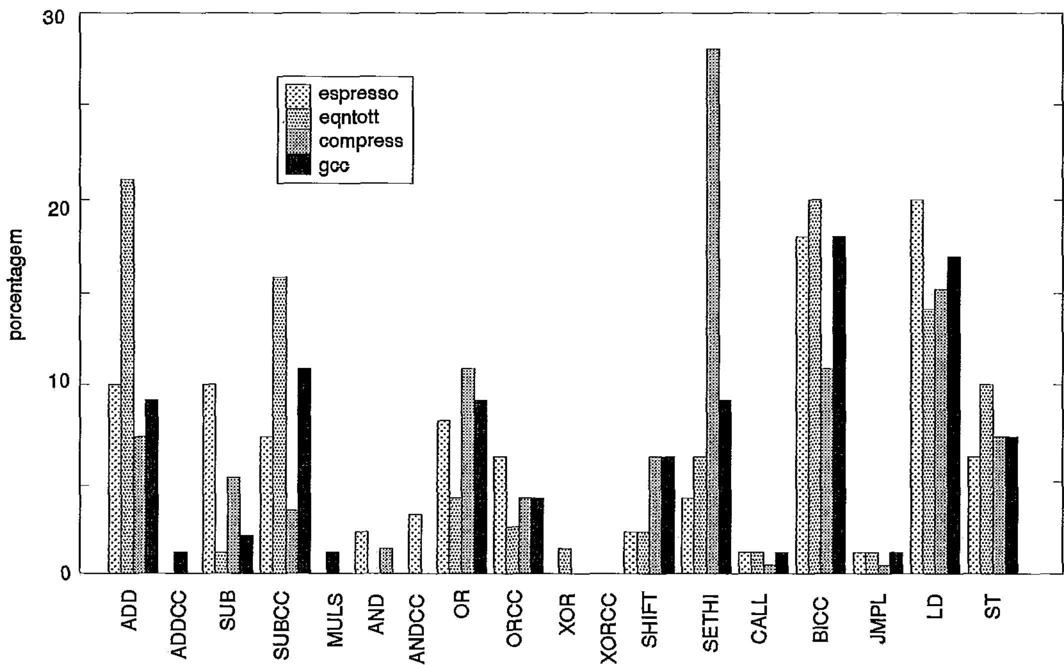


Figura 6.5: Distribuição do tipo de instruções executadas.

Vamos agora verificar o comportamento das configurações com unidades de memória em relação ao número de unidades que executam as instruções aritméticas e lógicas. Isto é mostrado no gráfico à direita, na Figura 6.6. Este gráfico foi obtido usando-

se configurações com quatro unidades de memória, cada uma com quatro unidades virtuais, e com largura de despacho de oito instruções. O gráfico à esquerda na Figura 6.6 mostra o ganho de desempenho para as configurações Tipo D, nas quais instruções aritméticas/lógicas e instruções de acesso à memória são executadas pelas mesmas unidades funcionais.

As configurações com unidades de memória separadas apresentam níveis de desempenho superiores aos obtidos com a configuração heterogênea Tipo D, para até cinco unidades funcionais. Para um número maior de unidades funcionais, os dois tipos de configurações apresentam desempenho semelhante. A diferença observada é conseqüência da execução das instruções de acesso à memória em unidades específicas. Com isto, a latência destas instruções resulta em um impacto bem menor sobre as instruções aritméticas e lógicas. Não mais existe o tempo de espera de uma instrução aritmética/lógica por uma unidade funcional que permanece ocupada com uma instrução de acesso à memória com *cache miss*. Este efeito é mais significativo nas configurações com poucas unidades funcionais, pois nestes casos aumentam as chances de despachar instruções aritméticas/lógicas para uma unidade funcional que está executando um acesso à memória.

6.4 O Modelo Bloqueante com Diferentes ALUs

Consideramos agora as configurações Tipo AL, onde a execução de instruções aritméticas e lógicas é distribuída através de unidades funcionais (ALUs) diferentes. O objetivo agora é verificar se existe algum ganho em uma arquitetura com ALUs especializadas, em contraste com arquiteturas cujas ALUs podem executar qualquer instrução aritmética/lógica. As instruções de desvio e as instruções de acesso à memória continuam sendo executadas por unidades funcionais específicas. Os resultados foram obtidos usando-se configurações com quatro unidades de memória, cada uma com quatro unidades virtuais. Como visto na seção anterior, com este número de unidades de memória não há redução no desempenho devido à capacidade de execução de instruções de acesso à memória. Em todas as configurações, a largura de despacho é de oito instruções. Foram consideradas três arranjos diferentes de tipos de ALUs:

Arranjo 1: cada ALU executa todas as instruções aritméticas e lógicas que aparecem na distribuição de instruções, na Figura 6.5;

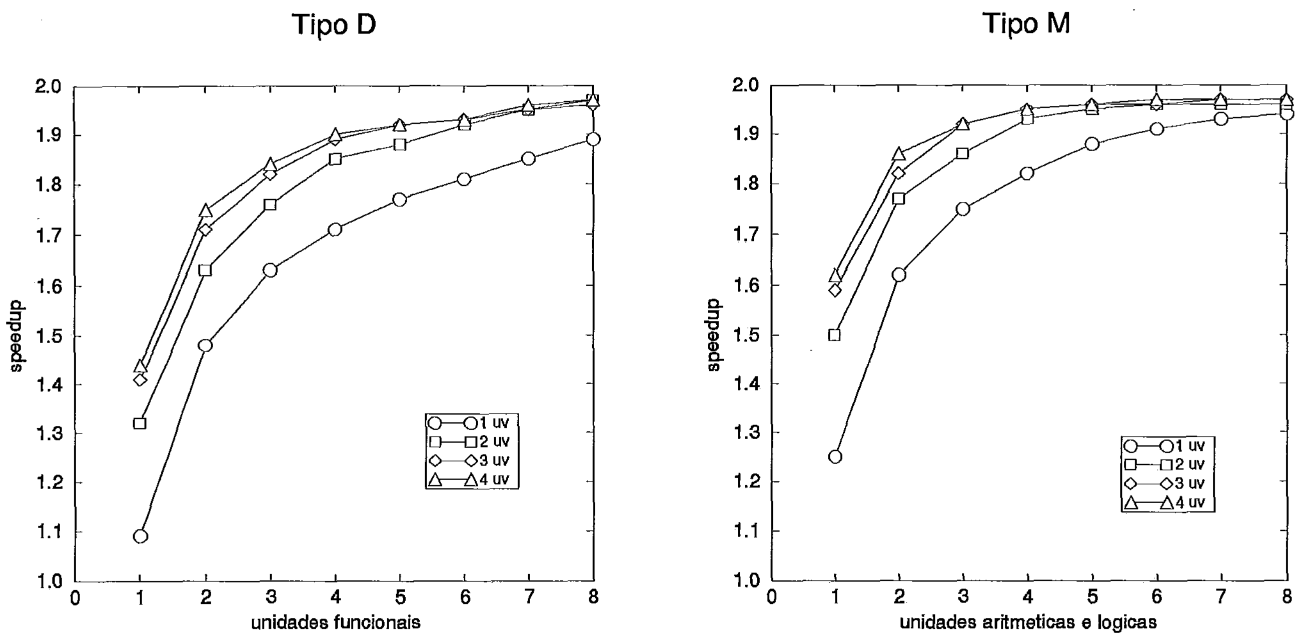


Figura 6.6: Desempenho das configurações Tipo D e Tipo M, em função do número de unidades que executam instruções aritméticas e lógicas.

Arranjo 2: para cada uma das treze diferentes instruções aritméticas/lógias que aparecem na Figura 6.5 existe uma ALU correspondente, que executa apenas aquela instrução;

Arranjo 3: o tipo das ALUs é determinado com base na freqüência de execução das instruções, conforme explicado à frente.

Os gráficos na Figura 6.7 mostram os níveis de desempenho obtidos com os Arranjos 1 e 2. Para o Arranjo 1, o número de unidades funcionais varia entre uma e oito unidades. Para o Arranjo 2, foram replicadas apenas as unidades funcionais que executam as instruções mais freqüentes, indicadas pela distribução mostrada na Figura 6.5 (`ADD`, `SUB`, `SUBCC`, `OR`, `ORCC`, `SHIFT` e `SETHI`). O número destas unidades funcionais varia entre uma e quatro unidades, e o número de unidades replicadas é o mesmo para os diferentes tipos de unidades.

O Arranjo 2 apresenta um desempenho superior apenas quando no Arranjo 1 existe uma única ALU. Esta vantagem desaparece rapidamente à medida que são acrescen-

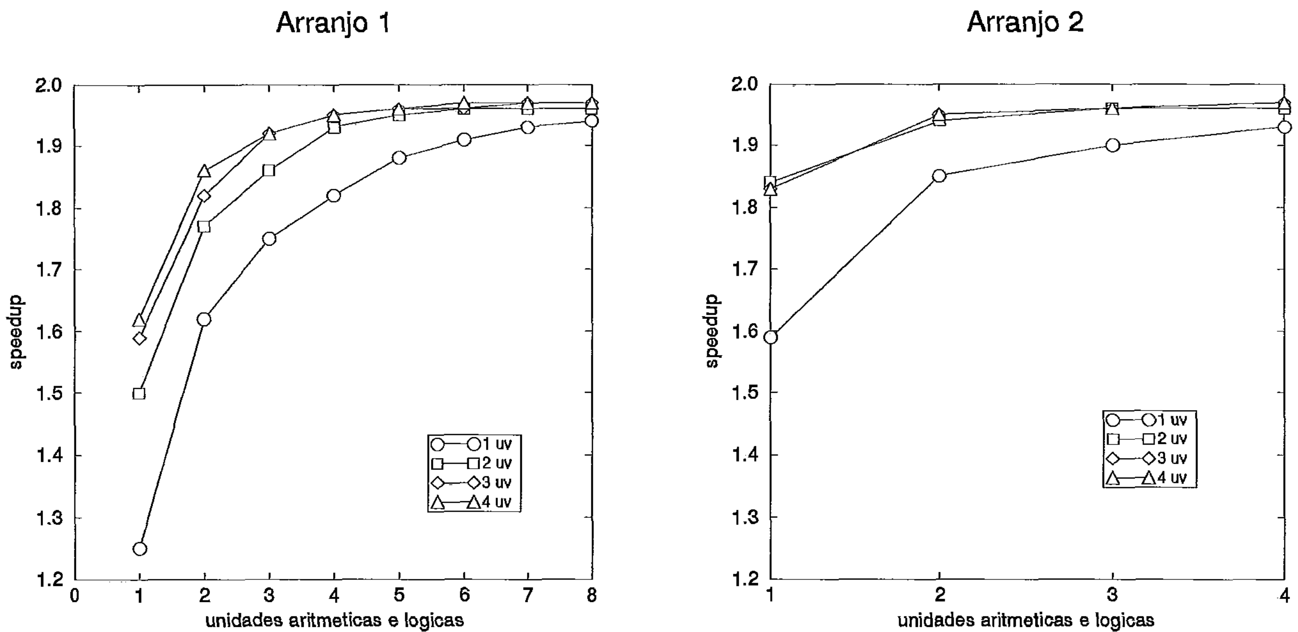


Figura 6.7: Comparação dos Arranjos 1 e 2 de ALUs.

tadas ALUs no Arranjo 1: uma configuração no Arranjo 1 com 4 ALUs e um total de 16 unidades virtuais fornece o mesmo desempenho de uma configuração no Arranjo 2 com 20 ALUs (ALUs duplicadas para as instruções mais usadas) e um total de 40 unidades virtuais. Estes resultados indicam que não há vantagem em projetar uma arquitetura com ALUs extremamente especializadas, pois é possível obter-se um desempenho equivalente com um número bem menor de ALUs complexas. A implementação de configurações no Arranjo 2 consumiria uma área de integração maior que a necessária para configurações com desempenho equivalente no Arranjo 1.

Os Arranjos 1 e 2 representam os dois extremos no nível de especialização das unidades aritméticas/lógicas. A razão para a proximidade no desempenho entre estes dois extremos deve-se ao baixo uso de algumas instruções. No Arranjo 2, as ALUs que executam tais instruções permanecem a maior parte do tempo ociosas, e a presença destas unidades em separado não contribui para aumentar o desempenho. A execução de instruções concentra-se em um sub-conjunto formado por poucas unidades, fazendo com que um pequeno número de ALUs complexas forneça o mesmo nível de paralelismo obtido com as ALUs especializadas.

Se por um lado estes resultados mostram que não há sentido em prover ALUs específicas para as instruções com pouco uso, por outro lado também pode-se argumentar que não há necessidade em capacitar todas as unidades a executar estas instruções, como acontece no Arranjo 1. Em uma implementação real, tanto a inclusão de unidades pouco utilizadas como a existência de um *hardware* replicado em cada unidade que também será pouco usado representa um desperdício em termos de área de integração. Isto sugere que um nível de especialização entre aqueles dos Arranjos 1 e 2 acima pode representar um melhor compromisso entre desempenho e custo de implementação. Assim, foi investigado um terceiro arranjo de unidades aritméticas e lógicas, onde são replicadas apenas as unidades para as instruções mais freqüentemente executadas.

No Arranjo 3 existem dois tipos de unidades. O primeiro tipo executa as instruções mais usadas: `ADD`, `SUB`, `SUBCC`, `OR`, `ORCC`, `SHIFT` e `SETHI`. O outro tipo de unidade executa as instruções menos usadas: `MULS`, `AND`, `ANDCC`, `XOR`, `XORCC`. Em configurações com este arranjo, foi replicada apenas o primeiro tipo de unidade, enquanto foi mantida apenas uma unidade que executa as instruções menos usadas. Em termos de implementação, isto representa um melhor uso da área de integração porque é replicado apenas o *hardware* que apresenta uma alta taxa de utilização. A Figura 6.8 mostra o desempenho obtido com o Arranjo 3. Estas configurações, incluem de uma a oito ALUs para as instruções mais usadas. Para efeito de comparação, na Figura 6.8 é repetido o gráfico para o Arranjo 1.

Como mostra a figura, apenas com a replicação das unidades funcionais para as instruções mais usadas obtém-se um desempenho equivalente ao do Arranjo 1. Este resultado mostra que é possível encontrar uma melhor distribuição de funcionalidade para as unidades aritméticas e lógicas que forneça o mesmo desempenho e apresente um menor custo de implementação, comparado ao uso de unidades mais complexas que executam todas as instruções aritméticas e lógicas.

6.5 O Modelo Bloqueante com Unidades de Ponto Flutuante Heterogêneas

No capítulo anterior foi verificado que existe uma vantagem significativa para o desempenho ao acrescentar-se várias unidades de ponto flutuante (ver Figura 5.7). No entanto, a replicação de unidades de ponto flutuante homogêneas é muito limitada

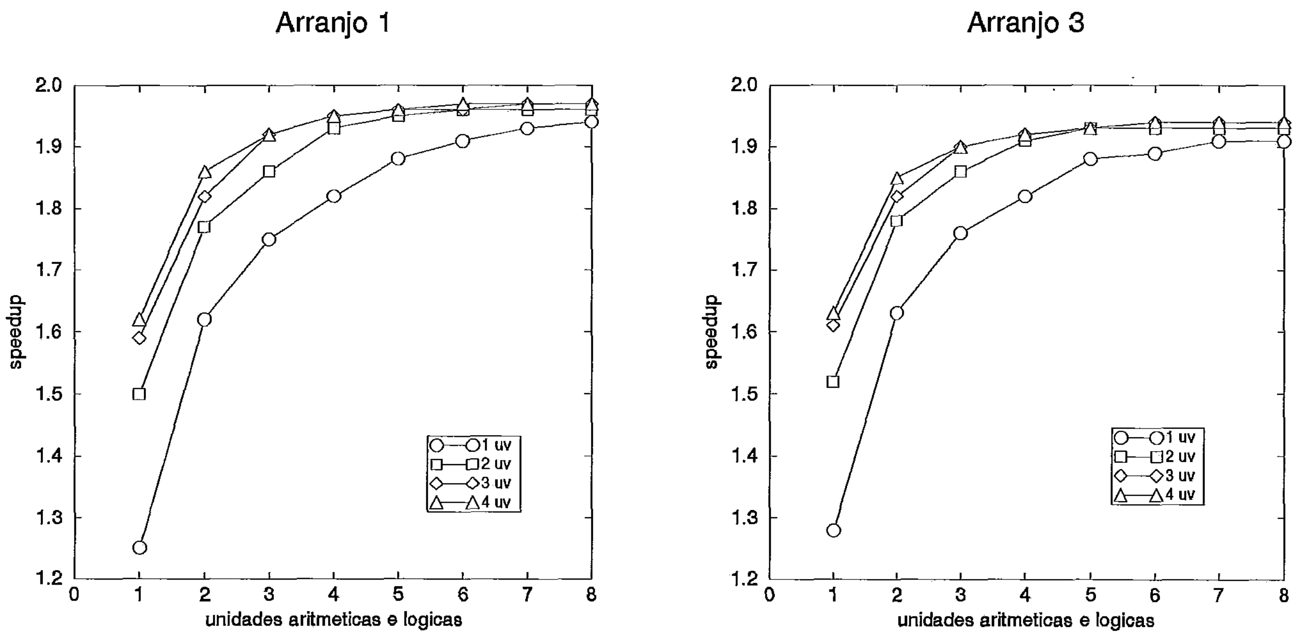


Figura 6.8: Comparação dos Arranjos 1 e 3.

pela área de integração exigida por cada uma destas unidades. Uma alternativa consiste em distribuir a execução de instruções de ponto flutuante através de unidades funcionais mais simples, e replicar apenas aquelas que são mais utilizadas. Com isto pode-se melhorar o desempenho com um custo de implementação menor, já que não é necessário replicar unidades complexas que executam todas as instruções de ponto flutuante.

Como mencionado anteriormente, nas configurações Tipo F foram considerados três unidades de ponto flutuante: a unidade de adição executa as instruções de adição e subtração; a unidade de multiplicação executa as instruções de multiplicação, divisão e raiz quadrada; a unidade de conversão realiza as operações de conversão entre tipos e precisões diferentes. Supõe-se que tais unidades comunicam-se apenas através de um conjunto de registradores compartilhado. Foram aqui consideradas configurações do Tipo F com o número de unidades de adição e de multiplicação variando entre uma e quatro unidades (para cada tipo) e com apenas uma unidade de conversão (devido à baixa freqüência de uso das instruções correspondentes). Todas configurações incluem oito unidades funcionais que executam instruções aritméticas/lógicas e de acesso à

memória, cada uma com quatro unidades virtuais. A largura de despacho foi fixada em oito instruções.

O gráfico na Figura 6.9 mostra o desempenho obtido com as unidades de ponto flutuante heterogêneas. As curvas em cada gráfico correspondem a um certo número de unidades de multiplicação. Cada gráfico corresponde a um número diferente de unidades virtuais por unidade de adição e multiplicação.

Para avaliar o efeito das unidades de ponto flutuante heterogêneas, é necessário comparar os gráficos na Figura 6.9 com o gráfico na Figura 5.7, o qual mostra o desempenho obtido com unidades de ponto flutuante complexas. Tal comparação indica que existe vantagem em adotar unidades de ponto flutuante heterogêneas somente em relação à configurações com apenas uma unidade de ponto flutuante complexa. Neste caso, o uso de unidades de adição e de multiplicação separadas resulta em um desempenho maior, devido à capacidade de executar diferentes tipos de instruções de ponto flutuante em paralelo. No entanto, esta vantagem desaparece quando consideramos configurações com mais de uma unidade complexa. Para manter o mesmo nível de desempenho, é necessário replicar as unidades de ponto flutuante heterogêneas por um número de vezes igual ao número de unidades complexas. Por exemplo, como mostram os grficos nas figuras acima mencionadas, para duas unidades complexas o mesmo desempenho é alcançado somente se forem incluídas duas unidades de adição e duas de multiplicação, o que provavelmente exigirá uma área de integração pelo menos equivalente. Note que este comportamento é diferente do observado com diferentes ALUs para instruções aritméticas/lógicas inteiras. Naquele caso, foi possível separar um *hardware* que atendia instruções pouco utilizadas, e cuja replicação não era necessária para manter-se o mesmo nível de desempenho. Com isto, é possível obter-se uma economia em área de integração. No entanto, para as aplicações de ponto flutuante, os resultados indicam que a funcionalidade contida em uma unidade complexa é efetivamente usada, e que a separação desta funcionalidade em unidades mais simples não leva a uma economia na implementação da arquitetura.

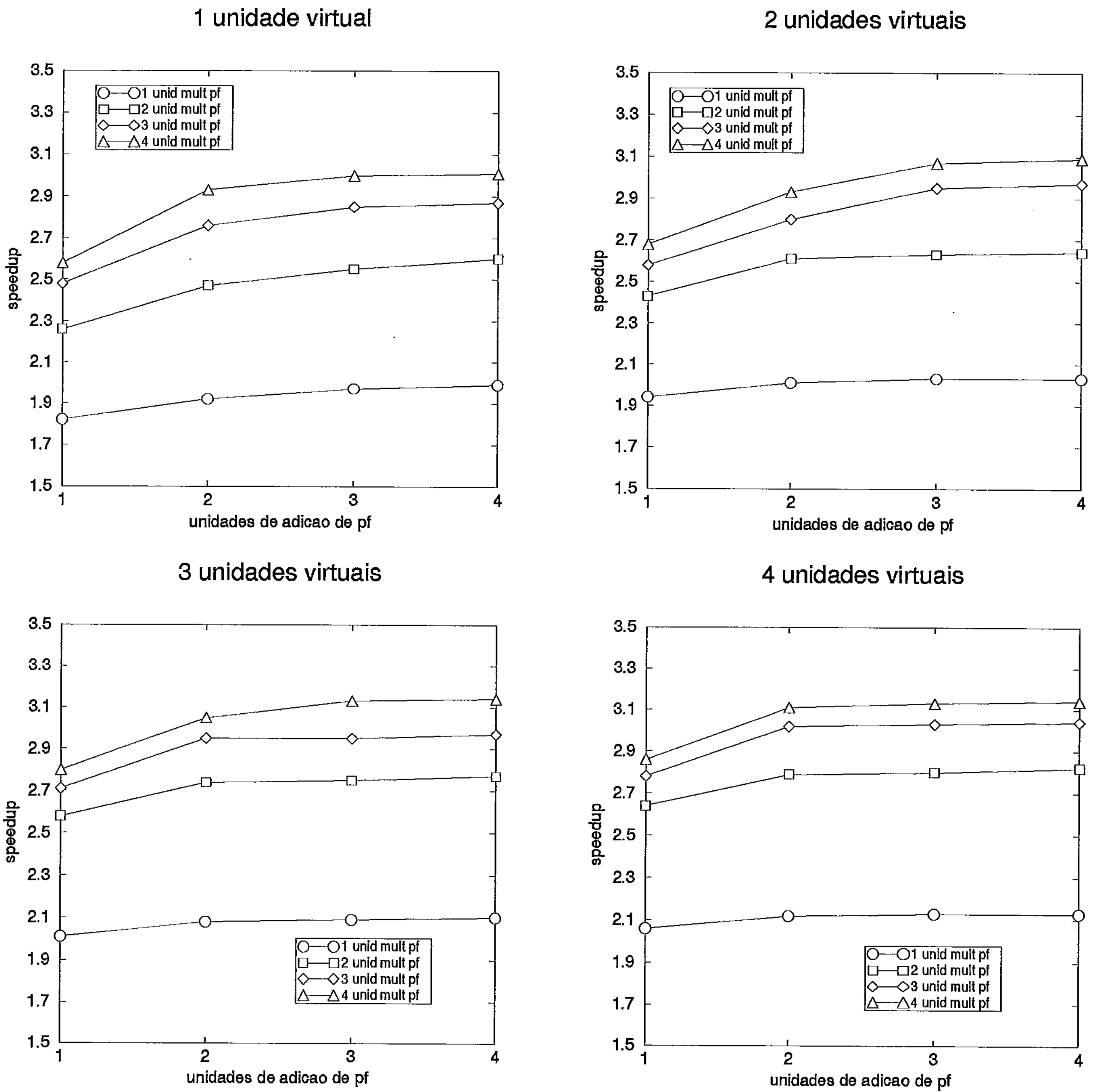


Figura 6.9: Desempenho do modelo bloqueante com unidades de ponto flutuante heterogêneas.

Capítulo 7

Avaliação do Modelo Especulativo

Nos capítulos anteriores foram avaliados modelos de arquiteturas onde as dependências de controle são tratadas bloqueando-se o despacho de novas instruções até que a instrução de desvio seja executada. Como visto, esta simples alternativa resulta em uma severa limitação para o paralelismo da arquitetura, principalmente quanto à utilização da largura de despacho, das unidades funcionais e à taxa de instruções completadas por ciclo. Com a introdução de unidades funcionais heterogêneas foram realizadas modificações para reduzir as latências na execução das instruções de desvio, de forma a diminuir o número de ciclos durante os quais o despacho permanece bloqueado. Embora estas modificações tenham de fato surtido algum efeito para elevar o desempenho, constatou-se que o bloqueio do despacho ainda representava um forte impedimento para a completa utilização do potencial paralelismo da arquitetura.

Neste capítulo é investigado um modelo de arquitetura que possui a capacidade de executar instruções especulativamente. Com a possibilidade de anular os efeitos de instruções executadas indevidamente, não é mais necessário bloquear o despacho após cada instrução de desvio. O objetivo aqui é mostrar como esta alternativa afeta o balanço da arquitetura e o seu desempenho final.

7.1 Configurações do Modelo Especulativo

A estrutura e o funcionamento do modelo de arquitetura super escalar com execução especulativa foi descrito no Capítulo 4. As configurações de arquitetura aqui conside-

radas são as mesmas usadas com o modelo bloqueante. Em todas elas, as memórias *cache* de instruções e de dados possuem linhas de 64 bytes, com 4 linhas por conjunto e 128 conjuntos, totalizando 32 Kbytes cada. A memória *cache* de instruções é organizada com oito bancos físicos, permitindo uma largura de busca de oito instruções por ciclo. A fila de instruções possui 32 entradas. São consideradas as mesmas larguras de despacho, ou seja, duas, quatro, seis e oito instruções.

Existem três novos parâmetros específicos do modelo especulativo, quais sejam, o tamanho do BTB, a profundidade de especulação e o tamanho do *buffer* de reordenação. Todas as configurações aqui usadas possuem um BTB com 256 entradas. Lee & Smith [Lee 84] mostram que a taxa de acerto na BTB não aumenta significativamente para tamanhos maiores de BTB. A profundidade de especulação foi fixada em quatro instruções de desvio. Novamente, este valor foi escolhido com base no estudo acima referido o qual mostra que, para vários programas pertencentes a diferentes classes de aplicações, existe uma probabilidade acumulada de 100% de ocorrerem pelo menos quatro instruções de desvio em uma seqüência de 10 instruções. Finalmente, o tamanho do *buffer* de reordenação foi fixado em 32 entradas. Este valor também se baseia no trabalho mencionado acima.

Este capítulo inicialmente avalia configurações com unidades funcionais homogêneas, e depois configurações heterogêneas com unidades de memória e unidades aritméticas e lógicas especializadas. Existe uma diferença entre as configurações homogêneas no modelo especulativo e aquelas consideradas no Capítulo 5. No modelo especulativo sempre existe uma unidade de desvio. Esta unidade é necessária porque as instruções de desvio devem ser executadas na ordem em que são despachadas. Se todas as unidades funcionais no modelo especulativo pudessem executar instruções de desvio, a execução ocorreria fora da ordem de despacho. Tal problema não existe no modelo bloqueante, porque o bloqueio do despacho naturalmente garante que as instruções de desvio sejam executadas em ordem.

7.2 O Modelo Especulativo com Unidades Homogênea

As Figuras 7.1 e 7.2 mostram os gráficos de desempenho para o modelo especulativo homogêneo. Para comparação, à esquerda são repetidos os gráficos para o modelo bloqueante. O primeiro ponto a observar é o aumento substancial nos níveis de desempenho. Com o modelo bloqueante, o maior *speedup* em relação à arquitetura

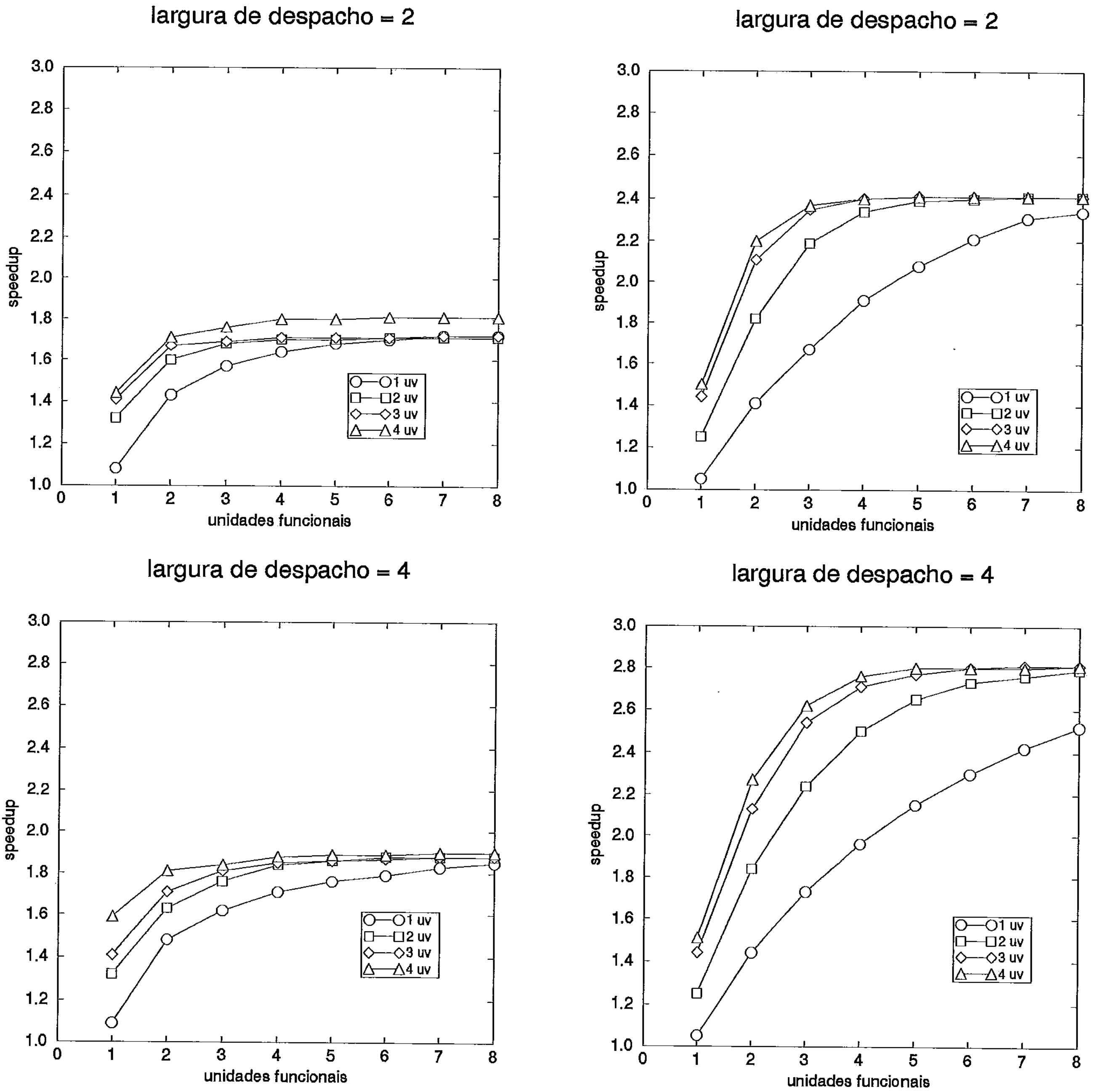


Figura 7.1: *Speedup* para o modelo especulativo homogêneo.

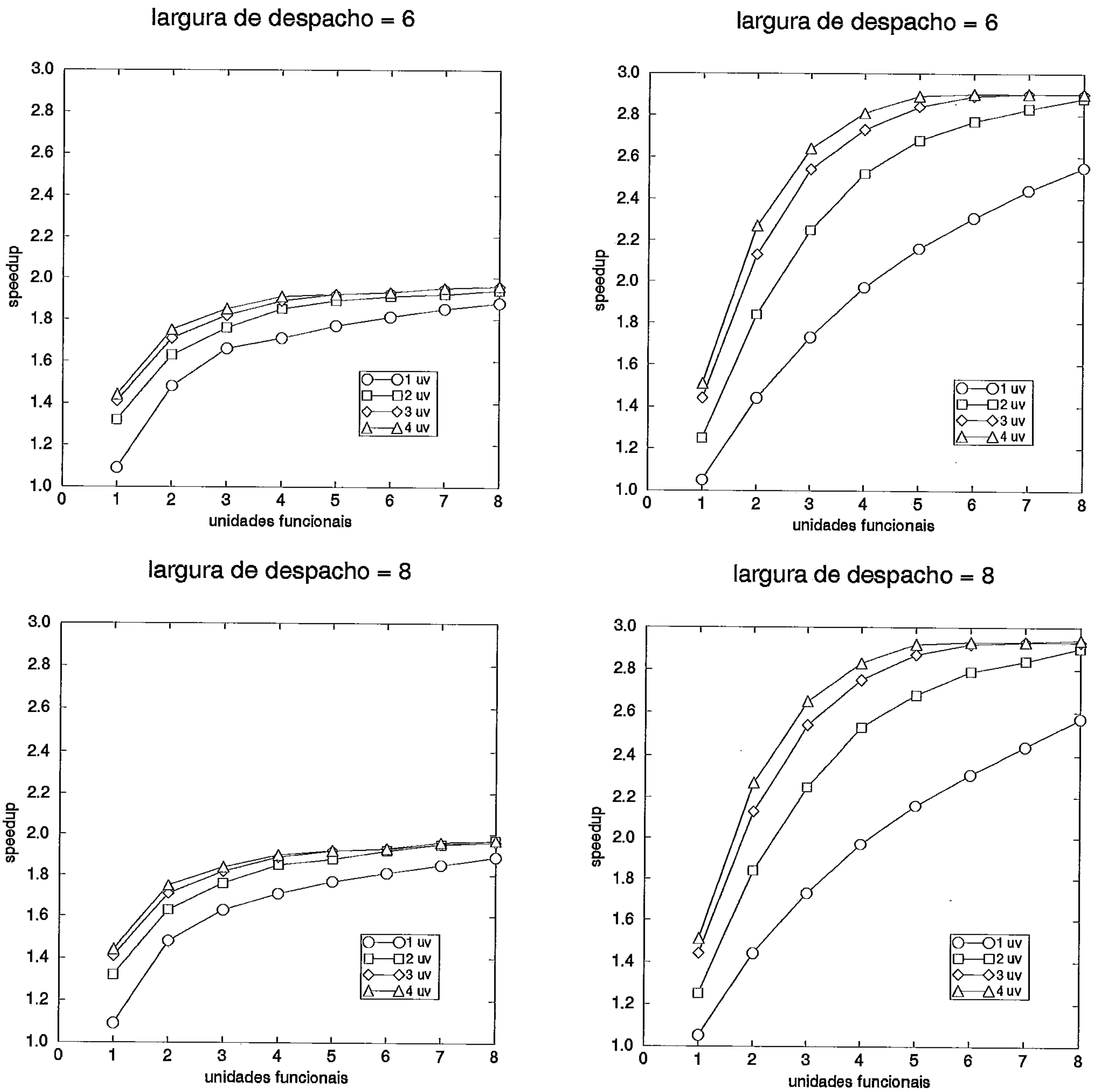


Figura 7.2: *Speedup* para o modelo especulativo homogêneo (continuação).

referência foi 1,97. Este valor já é ultrapassado na configuração especulativa mais limitada (com largura de despacho de duas instruções), onde o *speedup* chega a 2,4. O máximo desempenho obtido com o modelo especulativo com unidades homogêneas é três vezes maior que o fornecido pela arquitetura referência. Estes gráficos também mostram diferenças quanto ao efeito do número de unidades funcionais. Enquanto no modelo bloqueante não existe vantagem em acrescentar mais que três unidades funcionais, no modelo especulativo são necessárias de quatro a seis unidades funcionais para estabilizar o desempenho.

Os efeitos da largura de despacho e do número de unidades virtuais sobre o desempenho, podem ser apreciados na Figura 7.3. Nesta figura, cada gráfico corresponde a um certo número de unidades virtuais por unidade funcional, e cada curva corresponde a um certo número de unidades funcionais. Da mesma forma como no modelo bloqueante, no modelo especulativo o desempenho não se altera com a largura de despacho nas configurações com menos que três unidades funcionais. Como explicado anteriormente, nestes casos o desempenho é limitado pela capacidade de executar instruções em paralelo. O aumento mais notável no desempenho ocorre quando a largura de despacho passa de duas para quatro instruções. No modelo bloqueante o ganho no desempenho também concentra-se para a mesma variação na largura de despacho, mas naquele caso o aumento é comparativamente modesto: enquanto no modelo bloqueante o ganho era de 8%, no modelo especulativo o ganho é de 16%. Além disso, para larguras de despacho maiores que quatro instruções o aumento no desempenho também é mais acentuado no modelo especulativo. Quanto ao efeito das unidades virtuais, observa-se um aumento no desempenho quando são incluídas duas unidades virtuais por unidade funcional, bem maior que o observado no modelo bloqueante. Mas, como acontece no modelo bloqueante, o ganho de desempenho é comparativamente menor quando são incluídas três unidades virtuais, e praticamente não existe vantagem em acrescentar um número maior de unidades virtuais.

A maior sensibilidade do modelo especulativo em relação aos recursos disponíveis deve-se basicamente ao melhor aproveitamento da largura de despacho no modelo especulativo. Isto fica evidente a partir do gráfico na Figura 7.4, que mostra a distribuição do número de instruções despachadas por ciclo. Este gráfico foi obtido para configurações com oito unidades funcionais e quatro unidades virtuais por unidade funcional. Para efeito de comparação, na figura é repetido o gráfico com a distribuição de despachos para o modelo bloqueante.

Observa-se agora que a proporção de ciclos onde a largura de despacho é comple-

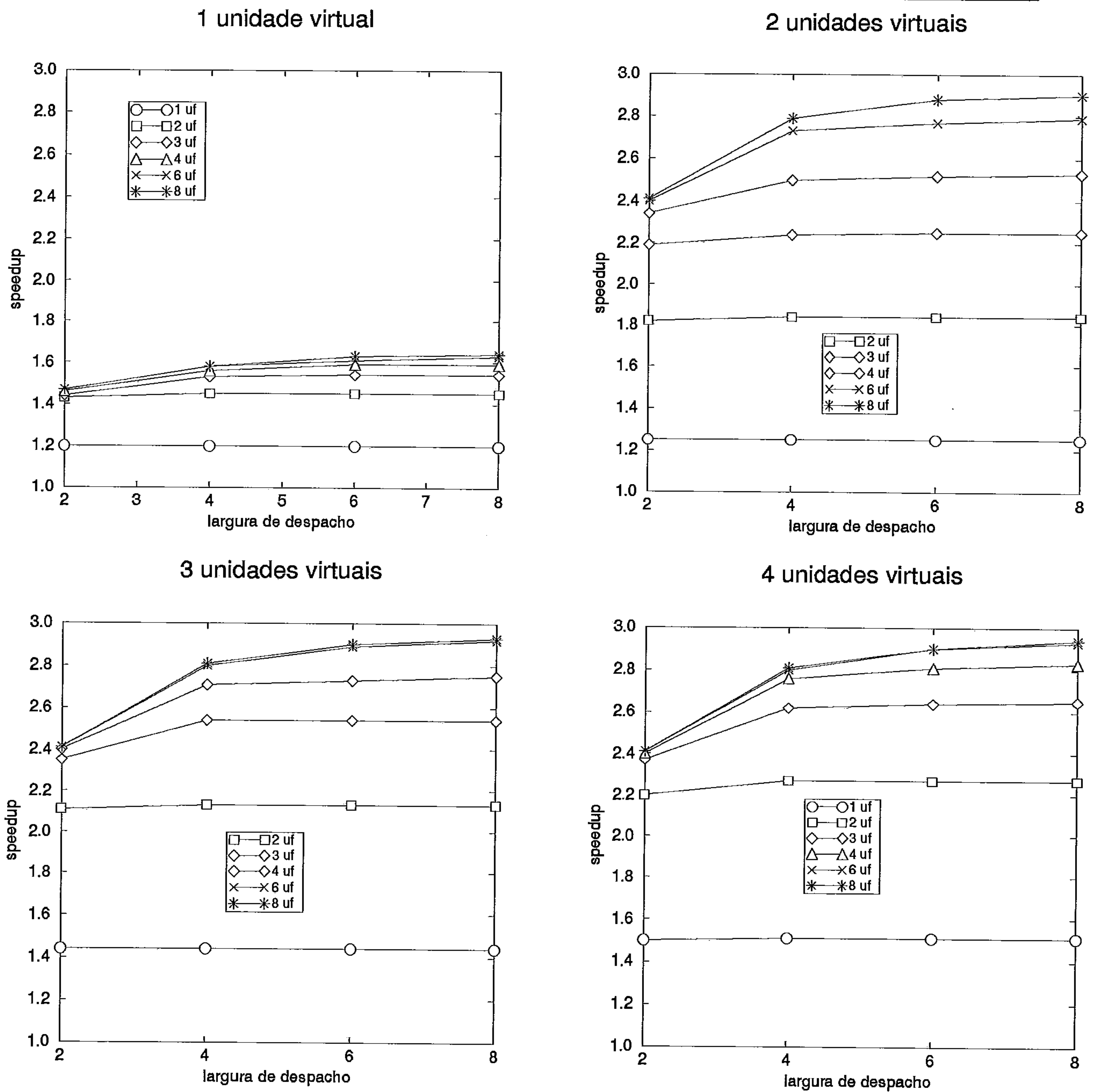


Figura 7.3: Efeito da largura de despacho sobre o desempenho.

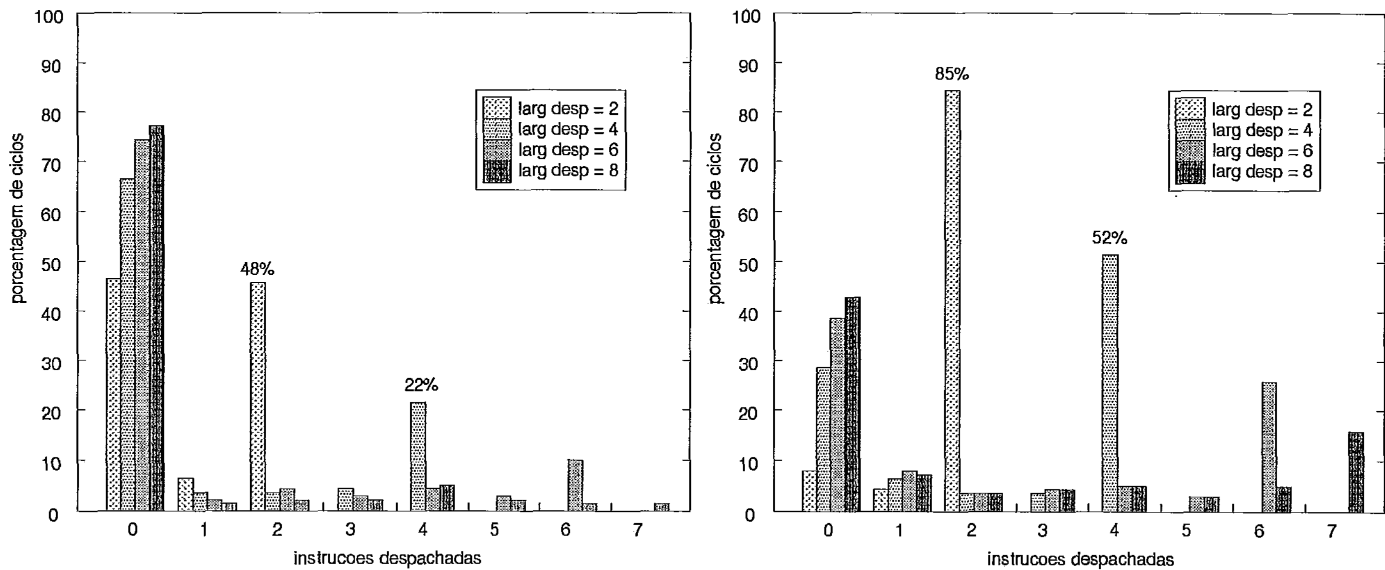


Figura 7.4: Distribuição de instruções despachadas no modelo especulativo, comparado com a distribuição no modelo bloqueante.

tamente utilizada é bem maior que no modelo bloqueante. Por exemplo, enquanto no modelo bloqueante as larguras de despacho de 2 e 4 instruções são totalmente aproveitadas em 48% e 22% dos ciclos, respectivamente, no modelo especulativo esta taxa sobe para 85% e 52% dos ciclos. Além destes melhores valores, a proporção de ciclos onde nenhuma instrução é despachada cai de 80%, no modelo bloqueante, para no máximo 50%, no modelo especulativo.

No modelo especulativo, o despacho não é bloqueado a cada desvio, o que resulta no aumento das taxas de aproveitamento. No entanto, existem casos onde o despacho ainda é bloqueado, o que impede um completo aproveitamento da largura de despacho, como a princípio podia se esperar. No modelo especulativo, o despacho é bloqueado em cinco circunstâncias diferentes: (1) quando a fila de instruções estiver vazia; (2) quando a profundidade de especulação máxima for ultrapassada, (3) na execução de um desvio cujo resultado foi previsto incorretamente; (4) por falta de recursos (no caso, unidades virtuais) e (5) quando o *buffer* de reordenação estiver cheio. A Tabela 7.1 mostra a proporção de ciclos em que cada um destes eventos acontece, para as possíveis larguras de despacho. Esta tabela foi obtida para configurações com oito unidades funcionais e com quatro unidades virtuais por unidade funcional. A tabela

Largura de Despacho	Fila Vazia	Profundidade Especulação	Unidades Virtuais	*Buffer* Reordenação	Previsão Incorreta
	Despacho Nulo				
2	5.38%	0.80%	0%	0.31%	5.66%
4	6.20%	7.62%	0%	3.26%	6.64%
6	6.34%	9.26%	0%	6.07%	6.63%
8	6.39%	9.80%	0%	7.71%	6.73%
	Despacho Parcial				
2	3.71%	0.32%	0%	0.63%	0%
4	4.13%	3.39%	0%	4.07%	0%
6	4.45%	4.74%	0%	8.03%	0%
8	4.60%	5.89%	0%	9.35%	0%

Tabela 7.1: Freqüência de eventos que bloqueiam o despacho.

mostra dois conjuntos de valores. No primeiro (despacho nulo), são contabilizados apenas os ciclos nos quais nenhuma instrução foi despachada. No segundo conjunto (despacho parcial), a contagem é feita apenas para os ciclos em que a largura de despacho não foi completamente utilizada.

Para uma largura de despacho de duas instruções, a fila de instruções vazia e a previsão incorreta de desvios foram os principais fatores que provocaram ciclos sem despacho. No entanto, para larguras de despacho maiores, a bloqueio por profundidade de especulação excedida torna-se dominante. Isto acontece porque, com uma largura de despacho maior, aumentam as chances de despachar um número maior de instruções de desvio no mesmo ciclo, esgotando assim rapidamente a profundidade de especulação e impedindo o despacho nos próximos ciclos. Para larguras de despacho de duas e quatro instruções, a fila de instruções vazia é o evento que mais limita a completa utilização da largura de despacho. No entanto, para larguras de despacho maiores, os bloqueios por *buffer* de reordenação cheio tornam-se dominantes.

A taxa de instruções completadas por ciclo é uma outra medida que bem revela o impacto da execução especulativa sobre a utilização dos recursos. A Figura 7.5 mostra o gráfico com a distribuição de instruções completadas por ciclo. Este gráfico

foi obtido a partir de configurações com largura de despacho de oito instruções e com quatro unidades virtuais por unidade funcional.

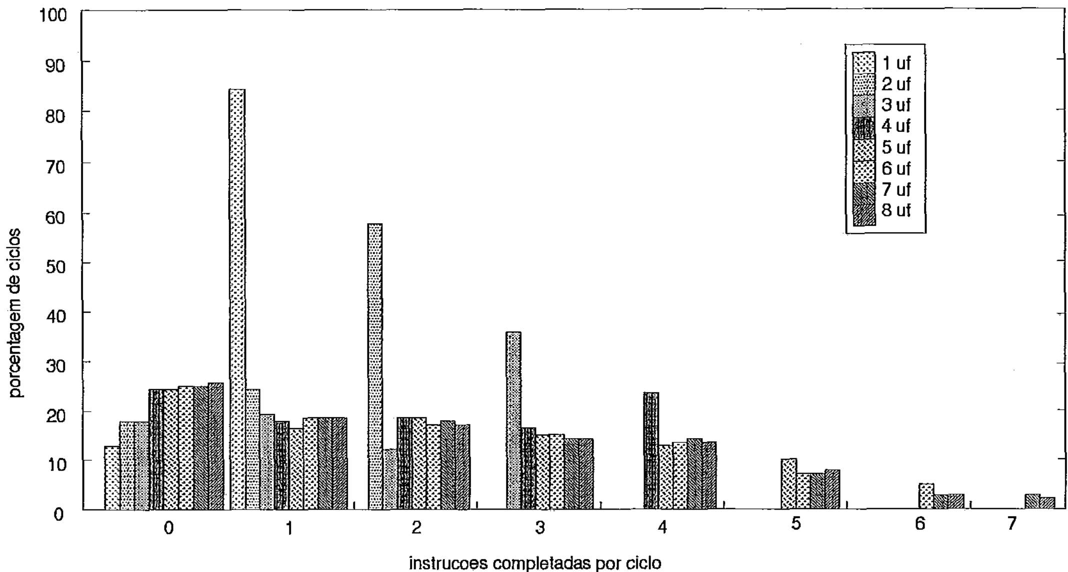


Figura 7.5: Distribuição de instruções completadas no modelo especulativo.

No modelo bloqueante, a proporção de ciclos onde nenhuma instrução é completada situa-se acima dos 40% em configurações com três ou mais unidades funcionais (veja Figura 5.4). Além disso, não mais do que quatro instruções são completadas simultaneamente, e isto ocorre em pouco mais de 1% dos ciclos. Ao contrário, no modelo especulativo, a proporção de ciclos em que nenhuma instrução é completada é de somente 24%. Este valor não aumenta com o número de unidades funcionais de forma tão acentuada como no modelo bloqueante, mantendo-se o mesmo para configurações com quatro ou mais unidades funcionais. Agora, o aumento do número de unidades funcionais não afeta sensivelmente a utilização de cada unidade, já que o volume de instruções disponíveis para execução não é limitado pelos bloqueios do despacho. No modelo especulativo, sete instruções são completadas na mesma proporção em que quatro instruções são completadas no modelo bloqueante.

A Figura 7.5 indica que ocorre uma melhor utilização das unidades funcionais no modelo especulativo. Em configurações com duas unidades funcionais, ambas permanecem ocupadas em 60% dos ciclos. Em configurações com quatro unidades funcionais, tais unidades permanecem ocupadas em quase 30% dos ciclos. No modelo

bloqueante, as taxas de ocupação das unidades funcionais em configurações com duas e quatro unidades funcionais caem para 20% e 1% dos ciclos, respectivamente. Estes valores são importantes porque eles mostram que no modelo especulativo o custo de adicionar unidades funcionais é explorado de maneira bem mais efetiva.

Os resultados apresentados até agora para o modelo especulativo referem-se aos programas inteiros. O gráfico na Figura 7.6 mostra o desempenho do modelo especulativo para os programas de ponto flutuante. As curvas foram obtidas a partir de configurações com oito unidades de inteiros. Cada unidade de inteiros possui uma unidade virtual, enquanto cada unidade de ponto flutuante possui até quatro unidades virtuais. A largura de despacho foi fixada em oito instruções.

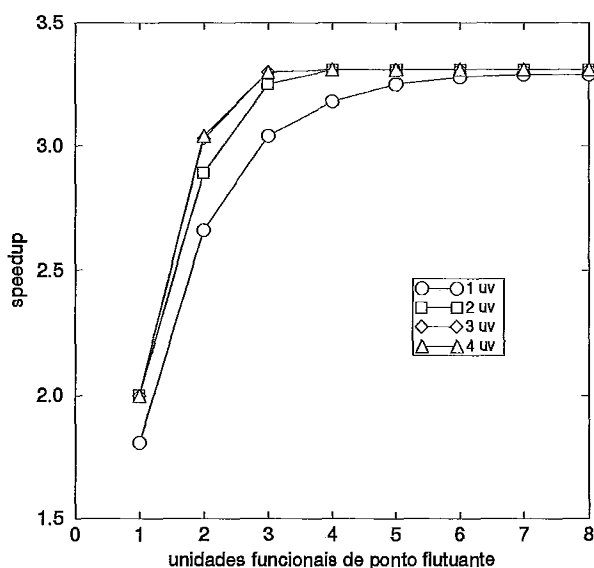


Figura 7.6: Desempenho do modelo especulativo para os programas de ponto flutuante.

Observa-se uma semelhança entre este gráfico e o correspondente para o modelo bloqueante (ver Figura 5.7). Os níveis de desempenho no modelo especulativo são apenas um pouco maiores que no modelo bloqueante. Isto se justifica pelo menor volume de dependências de controle nos programas de ponto flutuante, conforme demonstrado na Tabela 5.3. Com isto, a interrupção do despacho no modelo bloqueante torna-se

menos freqüente, elevando a utilização da largura de despacho e das unidades funcionais para níveis idênticos ao do modelo especulativo. Em ambos os modelos, o desempenho estabiliza-se com quatro unidades de ponto flutuante.

7.3 Modelo Especulativo com Unidades Heterogêneas

Até o momento foram examinadas configurações do modelo especulativo nas quais as unidades funcionais executam todas as instruções (exceto as de desvio que, como mencionado são executadas por uma unidade à parte). Podemos agora avaliar o modelo especulativo com unidades funcionais heterogêneas, repetindo para este modelo as experiências realizadas no Capítulo 6 com o modelo bloqueante.

Inicialmente, a execução das instruções de acesso à memória é transferida para unidades de memória. A Figura 7.6 mostra o comportamento do desempenho em relação ao número de unidades de memória. Este gráfico foi obtido a partir de configurações com oito unidades funcionais que executam as demais instruções. Cada uma destas unidades funcionais possui quatro unidades virtuais, e a largura de despacho é de oito instruções.

Como ocorreu com o modelo bloqueante, há uma redução no desempenho quando existe apenas uma unidade de memória, mas agora a queda no desempenho é comparativamente maior, chegando a 32% (no modelo bloqueante, a redução foi de 20%). Na realidade, esta maior sensibilidade ao número de unidades de memória era esperada, porque o escalonamento através de blocos básicos efetuado no modelo especulativo contribui para aumentar o número de instruções de acesso à memória que podem ser executadas simultaneamente. Assim, uma limitação na capacidade de executar tais instruções em paralelo possui um maior impacto sobre o desempenho. Um nível de desempenho equivalente ao obtido com unidades homogêneas é recuperado quando são incluídas quatro unidades de memória, cada uma com pelo menos duas unidades virtuais.

Em seguida avaliamos o desempenho de configurações com diferentes tipos de ALUs. Novamente, foram usados três possíveis arranjos. No primeiro arranjo, cada ALU executa todas as instruções aritméticas/lógicas, no segundo arranjo existe uma ALU específica para cada uma destas instruções, e no terceiro arranjo os tipos de ALUs são determinados pela distribução de instruções executadas. Os resultados estão na

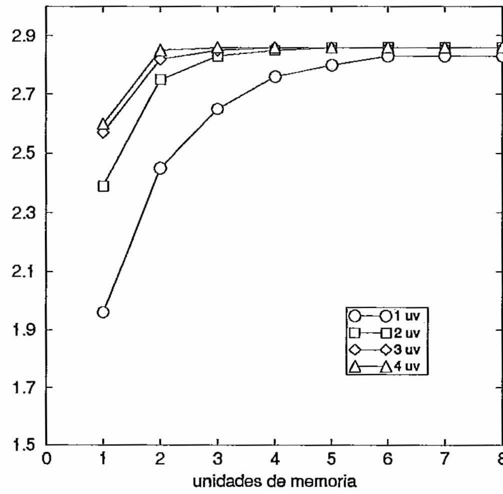


Figura 7.7: Desempenho do modelo especulativo em relação ao número de unidades de memória.

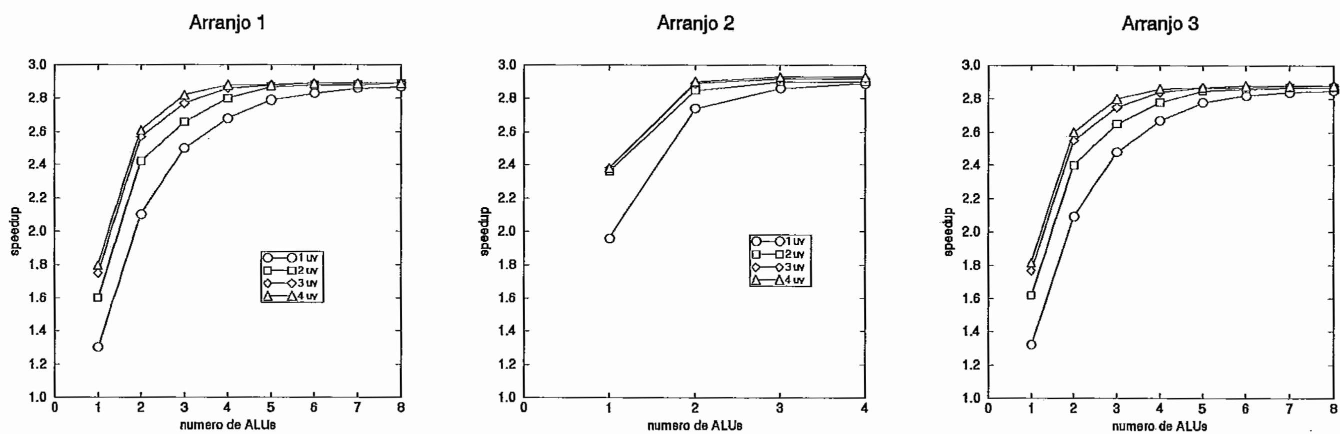


Figura 7.8: Desempenho do modelo especulativo com diferentes combinações de ALUs.

Figura 7.8. Os gráficos nesta figura mostram um comportamento semelhante ao observado com o modelo bloqueante. A opção por ALUs extremamente especializadas é vantajoso, porque obtém-se os mesmos níveis de desempenho alcançados com ALUs complexas, mas às custas de um número muito maior de unidades. É mais compensador um compromisso intermediário, separando-se as unidades de acordo com o perfil de utilização das instruções. Uma mudança presente nos gráficos acima está no maior número de de ALUs necessárias para atingir-se o desempenho máximo. Como já discutido, isto é uma conseqüência do maior volume de instruções despachadas por ciclo.

Capítulo 8

Conclusões

O tema central nesta tese é o balanceamento entre os principais componentes de uma arquitetura super escalar, e a contribuição destes componentes sobre o desempenho. Como salientado no Capítulo 1, este estudo foi motivado pelas diferenças arquiteturais encontradas nos atuais processadores super escalares. O conhecimento de como o desempenho de uma arquitetura se comporta em relação à combinação dos seus componentes é essencial para direcionar o projeto de arquiteturas que exploram o paralelismo a nível de instrução.

8.1 O Trabalho Desenvolvido

Neste trabalho, foi avaliado o desempenho em função da largura de despacho, do número de unidades funcionais e virtuais, e do tipo das unidades funcionais. Os efeitos destes parâmetros foram investigados no contexto de dois modelos de arquitetura que diferem quanto ao tratamento das dependências de controle. No modelo bloqueante, as dependências de controle interrompem o despacho normal de instruções. No modelo especulativo, o despacho e a execução de instruções prossegue tentativamente na presença de dependências de controle. Os parâmetros e modelos de arquitetura aqui considerados refletem os principais pontos de diferença entre arquiteturas super escalares correntes.

8.2 O Modelo Bloqueante com Unidades Funcionais Homogêneas

Inicialmente, avaliamos o modelo bloqueante com unidades funcionais do mesmo tipo, capazes de executar qualquer instrução (unidades funcionais homogêneas). Para os programas de teste inteiros, não existe vantagem significativa em adotar uma largura de despacho maior que quatro instruções. Na realidade, uma largura de despacho de quatro instruções pouco acrescenta: o desempenho aumenta de apenas 8% quando a largura de despacho passou de duas para quatro instruções. Isto acontece porque o excessivo bloqueio do despacho provocado pela dependências de controle impede a plena utilização da largura de despacho disponível. Este efeito é particularmente notável para larguras de despacho acima de quatro instruções, onde em até 80% dos ciclos nenhuma instrução é despachada devido aos bloqueios. Tal limitação impede que o aumento significativo da largura de despacho contribua para um melhor desempenho. Estes resultados justificam a escolha da largura de despacho em alguns processadores super escalares reais com modelo bloqueante.

A inclusão de um grande número de unidades funcionais também não resulta em ganhos significativos. Por exemplo, para larguras de despacho de duas instruções, não mais do que duas unidades funcionais homogêneas são necessárias para estabilizar o desempenho. Para larguras de despacho maiores, o desempenho estabilizou-se com quatro unidades funcionais. Os freqüentes bloqueios no despacho, que limitam a quantidade de instruções disponíveis para execução, fazem com que um número tão pequeno de unidades funcionais seja suficiente para que o desempenho máximo seja alcançado.

As unidades virtuais desempenham um papel importante, ao diminuir as exigências quanto ao número de unidades funcionais. Novamente, devido ao baixo volume de instruções despachadas, apenas duas unidades virtuais por unidade funcional são suficientes para armazenar temporariamente as instruções despachadas, evitando bloqueios por escassez de recursos. Com apenas uma unidade virtual, a utilização da unidade funcional fica bastante limitada. Estes resultados indicam que a disponibilidade de recursos encontrada em processadores reais com despacho bloqueante é adequada para as necessidades de programas inteiros.

O desempenho obtido com o modelo bloqueante homogêneo para os programas de ponto flutuante foi bem superior ao alcançado com os programas inteiros. No en-

tanto, é importante ressaltar que isto deve-se mais às características dos programas de ponto flutuante - menor freqüência de desvios, e blocos básicos maiores - do que a vantagens advindas do modelo de execução. O desempenho aumentou consideravelmente quando foram incluídas até quatro unidades de ponto flutuante, cada uma delas com duas unidades virtuais. Devido ao número de desvios comparativamente menor, os programas de ponto flutuante apresentam um desempenho bem mais sensível ao número de unidades funcionais inteiras do que os próprios programas inteiros. Por exemplo, o desempenho aumentou em 40% quando o número de unidades inteiras passou de duas para seis unidades inteiras, enquanto que para os programas inteiros a variação foi de apenas 14%. Estes resultados sugerem que, sob o ponto de vista das aplicações de ponto flutuante, os processadores super escalares atuais apresentam um nível de paralelismo limitado. Um melhor balanço para atender os dois tipos de aplicações seria obtido com uma largura de despacho de quatro instruções e com quatro unidades inteiras, ao invés de uma largura de despacho de somente duas instruções, com duas unidades funcionais inteiras.

8.3 A Especialização das Unidades Funcionais

Além de avaliar o efeito da largura de despacho e do número de unidades funcionais e de unidades virtuais, também verificamos como a combinação de diferentes tipos de unidades funcionais afeta o desempenho. Foram utilizadas configurações com unidade de desvio, com unidades de memória, com diferentes tipos de unidades para instruções aritméticas e lógicas sobre inteiros (ALUs) e com diferentes tipos de unidades de ponto flutuante.

Os resultados mostraram que há uma queda no desempenho quando se restringe o número de unidades que podem executar instruções de acesso à memória. Para configurações com apenas uma unidade de memória, obtém-se um desempenho 25% menor que o obtido nas configurações com unidades homogêneas. Para alcançar um desempenho equivalente ao destas configurações, foi necessário incluir pelo menos quatro unidades de memória. Isto sugere que no projeto de uma arquitetura super escalar deve ser dada uma importância especial em permitir múltiplos acessos simultâneos à memória. Arquiteturas super escalares correntes com unidades funcionais heterogêneas possuem apenas uma unidade de memória, e permitem apenas um acesso de cada vez à memória.

É importante observar que o uso de múltiplas unidades de memória exige a possibilidade de realizar vários acessos simultâneos à memória *cache*. Esta não é uma exigência irrealizável. Tal facilidade já é encontrada no Intel Pentium, no qual a memória *cache* é organizada em bancos *interleaved*, permitindo dois acessos simultâneos a bancos diferentes. Os resultados aqui apresentados sugerem apenas que tal facilidade deve ser estendida, de forma a permitir quatro acessos simultâneos à *cache*.

Verificou-se que não existe vantagem em incluir ALUs extremamente especializadas. A solução adequada consiste em orientar a escolha dos tipos de ALUs de acordo com o perfil de utilização dinâmica das instruções aritméticas e lógicas sobre inteiros. Replicando-se apenas ALUs para as instruções executadas mais freqüentemente, é possível obter um desempenho equivalente ao de uma arquitetura com unidades que executam qualquer instrução aritmética/lógica, com uma economia em área de integração. Por outro lado, a especialização das unidades de ponto flutuante não contribui para reduzir o custo de *hardware*. Existe vantagem apenas ao usarmos uma unidade de adição e outra de multiplicação separadas, ao invés de uma única unidade de ponto flutuante. No entanto, os resultados mostraram que para alcançar o desempenho obtido com n unidades de ponto flutuante homogêneas, é necessário que tenhamos o mesmo número de unidades de adição e de multiplicação.

8.4 As Dependências de Controle

Um dos aspectos mais importantes que verificamos durante este trabalho refere-se ao efeito das dependências de controle sobre o desempenho de uma arquitetura super escalar. O algoritmo de Tomasulo é extremamente eficiente no tratamento das dependências de dados. No entanto, mesmo com o uso deste mecanismo, o desempenho do modelo bloqueante pode ser considerado, no máximo, como satisfatório. Isto acontece porque as dependências de controle limitam sobremaneira a utilização da largura de despacho e das unidades funcionais. De fato, o desempenho do modelo bloqueante aumentou com a inclusão de unidades funcionais, mas às custas de uma sub-utilização cada vez maior destas unidades. Assim, no modelo bloqueante, o simples aumento da largura de despacho e/ou do número de unidades funcionais como meio de incrementar o desempenho conduz, na realidade, a um mau compromisso entre o custo de implementação e o uso eficiente destes recursos. Antes, deve-se aumentar a utilização de um menor número de recursos. Por este motivo, neste trabalho foram investigadas formas para reduzir o impacto das dependências de controle.

Neste sentido, a primeira alternativa consistiu em modificar o modelo bloqueante para torná-lo mais eficiente quanto à execução das instruções de desvio. A introdução de uma unidade funcional específica que executa desvios, reduziu as latências de execução destas instruções. Esta simples modificação resultou em um aumento significativo no desempenho. O máximo desempenho com o modelo bloqueante homogêneo, obtido com largura de despacho de oito instruções e oito unidades funcionais, foi alcançado pelo modelo com unidade de desvio em configurações com largura de despacho de apenas duas instruções e com somente duas unidades funcionais. Este resultado mostra que o uso de mecanismos para execução eficiente de desvios, como por exemplo o mecanismo de *zero-cycle branches* empregado em alguns processadores, representa uma vantagem significativa para o desempenho de uma arquitetura super escalar.

8.5 Execução Especulativa de Instruções

No entanto, uma análise mais detalhada do modelo bloqueante com unidade de desvio mostra um nível de ociosidade de recursos ainda alto. Por exemplo, a proporção de ciclos onde nenhuma instrução é despachada permanece praticamente a mesma no modelo bloqueante. Os resultados obtidos com a introdução da unidade de desvio indicam que o desempenho é bastante sensível às dependências de controle, mas também que uma redução significativa dos efeitos destas dependências exige soluções mais agressivas. Com o modelo especulativo, procurou-se observar até que ponto os efeitos das dependências de controle podem ser reduzidos através de um suporte intensivo de *hardware*, e como isto afeta o balanço entre os componentes da arquitetura.

De fato, com a execução especulativa de instruções, os níveis de utilização dos recursos aumentam substancialmente. A proporção de ciclos sem despacho caiu de 80% para cerca de 40% dos ciclos. Em conseqüência, o desempenho ficou bem mais sensível à largura de despacho. No modelo bloqueante com unidade de desvio, o desempenho aumenta em apenas 11% quando a largura de despacho passa de duas para oito instruções. No modelo especulativo, um aumento de 16% já é obtido com uma largura de despacho de quatro instruções. No entanto, assim como no modelo bloqueante, não se obtém ganhos significativos com larguras de despacho maiores que quatro instruções.

A utilização das unidades funcionais aumenta com o melhor uso da largura de despacho. Como resultado, é suficiente um pequeno número de unidades funcionais para

atingir-se os mesmos níveis de desempenho fornecidos pelo modelo bloqueante. Por exemplo, com apenas duas unidades funcionais e uma largura de despacho de duas instruções, obtém-se um desempenho superior a qualquer configuração do modelo bloqueante. Por outro lado, devido ao maior volume de instruções disponíveis para execução, torna-se necessário aumentar o número de unidades funcionais para que o desempenho máximo seja atingido. Agora, são necessárias até seis unidades funcionais homogêneas para estabilizar o desempenho. Além disso, enquanto no modelo bloqueante não havia diferença significativa em incluir três unidades virtuais por unidade funcional, no modelo especulativo o aumento de desempenho justifica a inclusão de uma terceira unidade virtual. O comportamento do modelo especulativo em relação ao uso de unidades funcionais heterogêneas é semelhante ao do modelo bloqueante. O desempenho apresenta uma queda maior quando é usada apenas uma unidade de memória, sendo necessárias quatro unidades de memória para se atingir o máximo desempenho. Novamente, não existe vantagem em usar-se ALUs especializadas para cada tipo de instrução aritmética/lógica. A replicação apenas das ALUs para as instruções mais usadas representa um melhor compromisso.

8.6 Comentários Finais e Trabalhos Futuros

As informações contidas neste trabalho fornecem subsídios para o projeto de uma arquitetura super escalar, na medida que eles mostram como se comporta o desempenho em função dos recursos incluídos na arquitetura. Os resultados sugerem que existem dois casos distintos, onde vale a pena ou não incluir um volume grande de recursos. A diferença entre estes dois casos está em como são tratadas as dependências de controle. Quando as dependências de controle são tratadas de um modo mais restritivo, não existe vantagem em dotar a arquitetura com uma ampla largura de despacho ou um grande número de unidades funcionais, pois o incremento de desempenho é obtido juntamente com uma sub-utilização dos recursos. Dependendo do nível de desempenho que se deseje obter, um melhor compromisso para uma arquitetura super escalar seria não concentrar o uso da área de integração disponível para as unidades de funcionais, mas sim dedicar uma parte maior para a implementação de técnicas que possibilitam uma melhor utilização das unidades.

É importante ressaltar que este trabalho focaliza apenas parte dos parâmteros que determinam o desempenho de uma arquitetura super escalar. Em particular, não consideramos os efeitos de parâmetros relacionados com o acesso de instruções, como

por exemplo, a configuração da memória *cache* de instruções, a largura de busca e o tamanho da fila de instruções. Este aspecto será examinado em um futuro trabalho. Também pretendemos em um próximo trabalho examinar isoladamente a eficiência dos diferentes mecanismos de execução especulativa apresentados no Capítulo 2. Finalmente, pretendemos ainda analisar o efeito do escalonamento estático de instruções sobre o desempenho do modelo bloqueante.

Apêndice A

Conjunto de Instruções SPARC

LDSB	Load signed byte	SAVE	Save caller's window
LDSH	Load signed halfword	RESTORE	Restore caller's window
LDUB	Load unsigned byte	Bicc	Branch on icc
LDUH	Load unsigned halfword	FBfcc	Branch on fcc
LD	Load word	CBccc	Branch on ccc
LDD	Load doubleword	CALL	Call
LDF	Load floating-point	JMPL	Jump and link
LDDF	Load double floating-point	RETT	Return from trap
LDFSR	Load floating-point state register	Ticc	Trap on icc
LDC	Load coprocessor	FiTOs	Convert integer to single
LDDC	Load double coprocessor	FiTOd	Convert integer to double
LDCSR	Load coprocessor state register	FiTOx	Convert integer to extended
STB	Store byte	FsTOi	Convert single to integer
STH	Store halfword	FdTOi	Convert double to integer
ST	Store word	FxTOi	Convert extended to integer
STD	Store doubleword	FsTOd	Convert single to double
STF	Store floating-point	FsTOx	Convert single to extended
STDF	Store double floating-point	FdTOs	Convert double to single
STFSR	Store floating-point state register	FdTOx	Convert double to extended
STDFQ	Store double floating-point queue	FxTOs	Convert extended to single
STC	Store coprocessor	FxTOd	Convert extended to double
STDC	Store double coprocessor	FMOV	Move between fp registers
STCSR	Store coprocessor state register	FNEG	Negate
STDCQ	Store double coprocessor queue	FABS	Absolute value
LDSTUB	Atomic load-store	FSQRT	Fp square root
SWAP	Swap register with memory	FADD	Fp add
ADD (ADDcc)	Add (modify icc)	FSUB	Fp sub
ADDX (ADDXcc)	Add with carry (modify icc)	FMUL	Fp multiply
TADDcc	Tagged add modify icc	FDIV	Fp divide
SUB (SUBcc)	Subtract (modify icc)	FCMP	Fp compare
SUBX (SUBXcc)	Subtract with carry (modify icc)		
TSUBcc	Tagged subtract modify icc		
MULScc	Multiply step modify icc		
AND (ANDcc)	And (modify icc)		
ANDN (ANDNcc)	And Not (modify icc)		
OR (ORcc)	Or (modify icc)		
ORN (ORNcc)	Or Not (modify icc)		
XOR (XORcc)	Exclusive Or (modify icc)		
XNOR (XNORcc)	Exclusive Nor (modify icc)		
SLL	Shift left logical		
SRL	Shift right logical		
SRA	Shift right arithmetic		
SETHI	Set highest 22 bits		

Apêndice B

Descrição do Simulador SPARC

O simulador SPARC foi escrito na linguagem C++, em uma plataforma Sun SPARCstation 2 com sistema operacional SunOS 4.1.1 (Unix 4.3 BSD). As rotinas nos módulos que formam o simulador reproduzem as operações executadas pelos estágios do *pipeline* e por outros componentes do processador MB86900 (ver Capítulo 3). A estrutura do simulador é mostrada na Figura B.1.

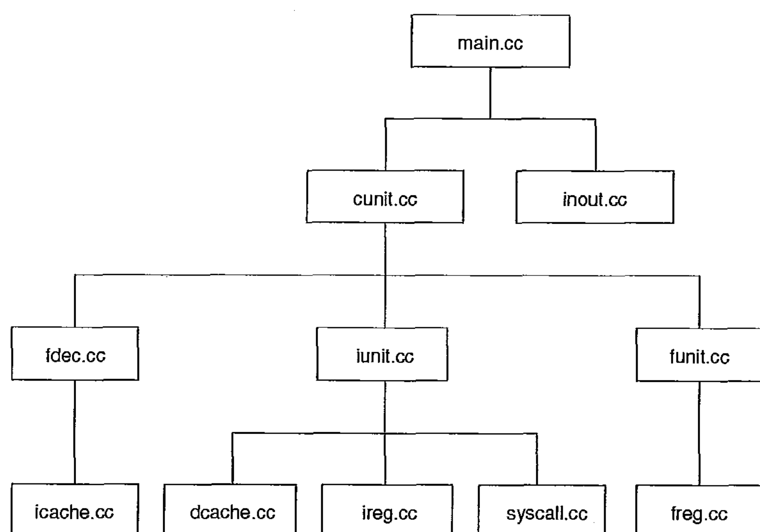


Figura B.1: Estrutura do simulador SPARC.

As funções dos módulos que aparecem na figura são as seguintes:

- `main.cc`: inicialização do simulador;
- `cunit.cc`: simula a unidade de controle do processador;
- `fdec.cc`: simula os estágios de busca e decodificação do *pipeline*;
- `iunit.cc`, `funit.cc`: correspondem ao estágio de execução do *pipeline*, e simulam a execução de instruções inteiras e de ponto flutuante, respectivamente;
- `ireg.cc` e `freg.cc`: simulam as operações de acesso aos conjuntos de registradores de inteiros e de ponto flutuante, respectivamente;
- `icache.cc` e `dcache.cc`: simulam, respectivamente, as memórias *cache* de instruções e de dados;
- `syscall.cc`: implementa as chamadas de sistema operacional para o programa sendo executado pelo simulador;
- `inout.cc`: contém as rotinas de acesso aos arquivos de *trace* e de *interface* com o usuário.

A Figura B.2 mostra o fluxo de controle entre as principais rotinas do simulador. Na figura estão indicados os módulos onde se encontram estas rotinas.

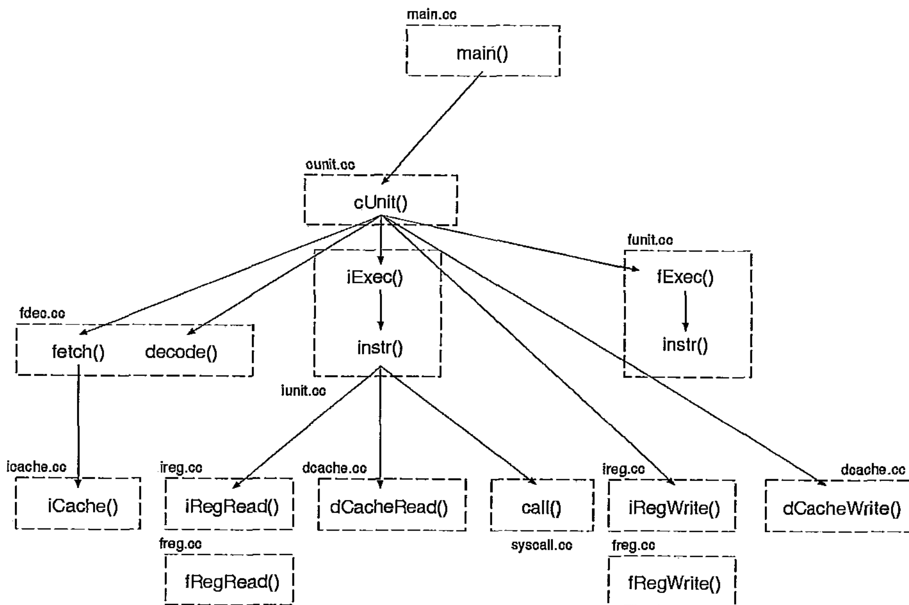


Figura B.2: Fluxo de controle no simulador SPARC.

A rotina `main()` inicia lendo os parâmetros de entrada, quais sejam, o nome do arquivo

executável e o número de instruções a serem simuladas. Em seguida, `main()` carrega o código executável em um vetor que simula a memória principal. A carga é feita de acordo com o endereço de entrada e o tamanho do segmento de texto, indicados no cabeçalho do arquivo executável. Após a carga do código executável, `main()` passa o controle para `cUnit()`, indicando o endereço de entrada e o número de instruções a serem simuladas. Em cada iteração, `cUnit()` chama rotinas que realizam as tarefas de cada estágio do *pipeline*. A execução destas rotinas é repetida até que tenha sido simulado o número indicado de instruções. As rotinas se comunicam através de estruturas de dados compartilhadas, que representam os *buffers* de transferência entre os estágios do *pipeline*. Em cada iteração, `cUnit()` executa os seguintes passos:

(1) `cUnit()` chama a rotina `fetch()` para acessar a memória *cache* de instruções. Por sua vez, esta rotina chama `iCache()`, que verifica se a instrução encontra-se na *cache* e carrega linhas quando necessário. A rotina `fetch()` coloca o código da instrução no *buffer* de entrada para a rotina do estágio de decodificação;

(2) `cUnit()` chama a rotina `decode()` para decodificar a instrução. Esta rotina extrai os campos do código da instrução e os coloca no *buffer* de entrada para a rotina do estágio de execução;

(3) `cUnit()` chama a rotina `iExec()` ou `fExec()`, conforme a instrução seja inteira ou de ponto flutuante, respectivamente. `iExec()` ou `fExec()` copia os campos da instrução decodificada para um *buffer* privativo e em seguida chama a rotina que executa a instrução. Para cada instrução oferecida pela arquitetura SPARC existe uma rotina específica em `iunit.cc` ou `funit.cc` que simula a execução da instrução (na Figura B.2, esta rotina é indicada por `instr()`). Para maior eficiência, a chamada da rotina é feita através de um vetor de ponteiros de funções usando como índice o código de operação da instrução;

(4) a rotina que simula a execução da instrução lê os operandos em registradores através das rotinas `iRegRead()` ou `fRegRead()`. Em instruções de acesso à memória são chamadas as rotinas `dCacheRead()` ou `dCacheWrite()`, que simulam o acesso memória *cache* de dados. Um caso especial é o da instrução `TRAP`, usada pelo programa em execução para chamadas ao sistema operacional. A rotina que simula esta instrução passa o número de chamada para uma outra rotina (no módulo `syscall.cc`), que de fato realiza a chamada ao sistema operacional. Seguindo a convenção adotada no compilador C Unix, esta rotina obtém nos registradores simulados os parâmetros da chamada. Resultados retornados pela chamada de sistema são armazenados nos

registradores simulados apropriados, conforme esperado pelo programa em execução.

(5) após a execução da instrução, `cUnit()` chama `iRegWrite()` ou `fRegWrite()` para escrever o resultado no registrador destino. Este passo corresponde ao estágio de escrita de resultado do *pipeline*.

Cada iteração de `cUnit()` corresponde a um ciclo do processador. O contador de ciclos é incrementado ao final de cada iteração, após a execução dos passos enumerados acima. Com isto é reproduzida a execução paralela dos estágios do *pipeline* dentro de cada ciclo. Note que cada rotina opera sobre uma instrução diferente. Por exemplo, em uma mesma iteração `fetch()` acessa a instrução i, `decode()` decodifica a instrução $i-1$ acessada por `fetch()` na iteração anterior, `iExec()` (ou `fExec()`) executa a instrução $i-2$ acessada por `fetch()` duas iterações atrás e decodificada por `decode()` na última iteração, e assim por diante.

Um dos principais aspectos no simulador SPARC é a reprodução das travas (*interlocks*) de *pipeline*. Devido as dependências entre instruções ou a falhas na memória *cache*, é necessário paralizar a execução de certos estágios do *pipeline*. A simulação precisa destas condições é essencial para garantir a correção temporal do simulador. O quadro na Figura B.3 mostra as condições de travamento do *pipeline* reproduzidas pelo simulador. Esta figura mostra uma matriz de atuação de travas, onde cada coluna corresponde ao estágio que ativa (T) ou desativa (D) uma trava, e cada linha corresponde ao estágio afetado pela trava. Na interseção da linha-coluna está o nome da trava.

O estágio de busca é travado pelo estágio de execução (via `fetchLock`) quando a instrução neste estágio é um desvio condicional e a instrução seqüencial à posição de atraso e/ou a instrução no destino do desvio ainda não foram acessadas. Este travamento ocorre devido a um *miss* na memória *cache* de instruções. O estágio de busca permanece travado até que tenha decorrido o tempo de latência de *miss*, após o que a trava é desativada também pelo estágio de execução. O estágio de decodificação é automaticamente travado porque a instrução neste estágio não pode prosseguir para o estágio de execução. O estágio de decodificação é travado por ele mesmo (via `decLock`) quando a instrução decodificada é uma instrução de desvio e a instrução na posição de desvio ainda não foi acessada, devido a um *cache miss*. O estágio de decodificação desativa a trava após ter decorrido o tempo de latência de *miss*. O estágio de execução é travado por ele mesmo (via `execLock`) em uma condição semelhante ao travamento do estágio de busca, ou seja, quando a instrução

	Fetch	Decode	Exec	WrBack
Fetch			fetchLock (T) fetchLock (D)	
Decode		decLock (T) decLock (D)		
Exec			execLock (T) execLock (D)	
WrBack				wrBackLock (T) wrBackLock (D)

Figura B.3: Condições de travamento do *pipeline*.

em execução é um desvio e as instruções nos dois possíveis destinos ainda não foram acessadas devido a um *cache miss*. Este estágio retira a trava após a latência de *miss*. Os estágios anteriores são automaticamente travados porque as instruções não podem seguir em frente. O estágio de escrita de resultado é travado por ele mesmo (via `wrBackLock`) quando a instrução neste estágio é de acesso à memória, com *miss* na memória *cache* de dados. A trava é desativada por este estágio após o tempo de latência de *miss*. Novamente, os estágios anteriores são automaticamente travados porque as instruções não podem seguir em frente. Note que o *pipeline* não é travado devido a dependências de dados entre instruções nos estágios de decodificação e execução, já que o MB86900 incorpora um mecanismo de *forwarding* (ver Capítulo 4). No entanto, a trava do estágio de escrita garante a dependência de uma instrução no estágio de decodificação em relação a uma instrução *load* que resulte em *cache miss*.

O simulador também reproduz a sincronização entre a unidade de inteiros e a unidade de ponto flutuante. Esta sincronização é necessária porque a transferência de dados entre a memória e os registradores de ponto flutuante é executada pela unidade de inteiros. A cada registrador de ponto flutuante está associado um bit de alocação, que é manipulado da seguinte forma:

(1) a rotina que implementa a instrução *load* verifica se o bit de alocação está ativado. Se este é o caso, uma instrução de ponto flutuante em execução usa o registrador como destino. O estágio de execução é travado (via `execLock`) até que a latência

da instrução de ponto flutuante expire. Quando isto acontece, o bit de alocação e a trava do estgio de execução são desativados. Ao contrário, se o bit de alocação encontrava-se desativado, a rotina que implementa a instrução *load* ativa o bit. A desativação do bit é feita pela rotina `fRegWrite()`, quando o dado é armazenado no registrador pelo estgio de escrita de resultado;

(2) a rotina que implementa a instrução *store* verifica se o bit de alocação está ativado. Se este é o caso, existe uma instrução de ponto flutuante em execução que usa o registrador como destino, e o estágio de execução é travado até que a latência da instrução expire. Se o bit não não está ativado, a instrução *store* é completada normalmente;

(3) a rotina que implementa uma instrução de ponto flutuante verifica se o bit de alocação de cada registrador-fonte está ativado. Se este é o caso, uma instrução *load* em execução tem aquele registrador como destino. O estágio de execução é travado até que a latência da instrução *load* expire e o bit seja desativado. No início da execução da instrução de ponto flutuante, o bit de alocação do registrador-destino é ativado.

Quando uma instrução de transferência de controle ou de acesso à memória é encontrada, as rotinas (em `iunit.cc`) que simulam a execução destas instruções escrevem as informações sobre a execução nos arquivos de *trace* apropriados. Na realidade, os *traces* de instruções são gerados em segmentos, cada segmento contido em um arquivo. Quando um segmento de *trace* é completado, o arquivo é fechado e compactado, e um novo arquivo é criado para o segmento seguinte. No uso dos *traces*, a cada momento apenas um destes arquivos (para cada *trace*) encontra-se carregado na memória. Esta divisão em segmentos permite que o *trace* seja compactado à medida que é gerado, o que reduz as exigências quanto ao espaço de aramazenamento em disco.

Apêndice C

Descrição do Simulador da Arquitetura Super Escalar

Os simuladores das arquiteuras super escalares foram escritos na linguagem C, em uma plataforma Sun SPARCstation 2 com sistema operacional SunOS 4.1.1.

Os módulos que formam o simulador desempenham as seguintes funções:

`main.c`: inicialização do simulador e controle de execução das rotinas que simulam o funcionamento do *pipeline*; `busca.c`: simula o estágio de busca do *pipeline*; `decod.c`, `desp.c`: simulam o estágio que decodifica e despacha instruções; `exec.c`: simula o estágio de execução do *pipeline*; `prop.c`: simula o estágio de propagação de resultados via CDB.

A rotina `main()` inicia lendo o arquivo de parâmetros. Alguns dos valores de parametros neste arquivo indicam a configuração da arquitetura e, de acordo com estes valores, `main()` aloca as estruturas de dados que representam os componentes da arquitetura. Como as estruturas são alocadas dinamicamente, é possível simular diferentes configurações de arquitetura sem a recompilação do simulador. A rotina `main()` lê ainda um segundo arquivo de parâmetros, contendo as latências de cada instrução. Por último, o programa executável é carregado na memória simulada e os arquivos de *trace* são abertos. Note que nos simuladores das arquiteturas super escalares as instruções não são de fato executadas. Os arquivos de *trace* serão usados para obter-se informações a respeito da execução real das instruções de desvio e das instruções de acesso memória.

Após esta inicialização, `main()` chama a rotina `pipeline()`, que controla a simulação dos estágios do pipeline. A cada iteração, `pipeline()` chama diversas outras rotinas para realizar a tarefa de cada estágio. Cada iteração corresponde a um ciclo, e as iterações se repetem até que seja simulado o número de instruções indicado no arquivo de configuração. A seqüência de passos em cada iteração de `pipeline()` é a seguinte:

(1) a rotina `busca()` é chamada para acessar instruções na memória *cache*. A rotina verifica se o *buffer* de busca está livre e, se for o caso, armazena neste *buffer* as instruções que estão na memória *cache*. O número de instruções a serem acessadas é igual ao tamanho da largura de busca, indicado no arquivo de configuração. Se uma ou mais instruções não se encontram na memória *cache*, as posições correspondentes no *buffer* de busca são reservadas, de forma a assegurar o acesso em ordem de instruções (ver Capítulo 3). Este passo corresponde à operação do estágio de busca.

(2) a rotina `decod()` é chamada para decodificar as instruções. Esta rotina retira as instruções no *buffer* de busca e, a partir do código de instrução, gera diversas informações necessárias nas próximas etapas da simulação. Algumas destas informações são: endereço da instrução, tipo da unidade funcional que executa a instrução, se a instrução é de desvio condicional ou incondicional, ou se a instrução é de acesso memória. Estas informações são armazenadas em uma entrada da fila de instruções.

Um tratamento especial é dado na decodificação das instruções de desvio. Como mencionado no Capítulo 3, instruções de desvio na arquitetura SPARC são dotadas de uma facilidade que permite a anulação da instrução na posição de atraso, conforme o resultado do desvio. Quando um desvio condicional é encontrado, é verificado se o bit *annul* está ativado e, se este é o caso, o *trace* de desvio é consultado para se determinar o resultado da execução do desvio. Se o *trace* indica que o desvio foi tomado, a instrução na posição de atraso é decodificada normalmente. Caso contrário, a instrução na posição de atraso é eliminada do *buffer* de busca. Na realidade, o tratamento é um pouco mais complexo, porque a instrução na posição de atraso também pode ser uma instrução de desvio. Neste caso, a definição da arquitetura SPARC prevê várias possibilidades para a instrução a ser executada após o desvio. A discussão detalhada de como este caso é tratado não cabe aqui. A rotina de decodificação cobre todas as possibilidades válidas segundo a definição da arquitetura SPARC, de modo que apenas as instruções que foram de fato executadas pelo simulador SPARC são inseridas na fila de instruções.

(3) a rotina `despacho()` é chamada para realizar o despacho de instruções. Esta rotina

aplica o algoritmo descrito no Capítulo 3 para encontrar a unidade funcional para onde a instrução será despachada. Se a unidade funcional é encontrada, várias informações na entrada da fila são copiadas para a unidade virtual. Para os registradores fonte com bits de alocação ativados, os *tags* também são copiados para a unidade virtual. O bit de alocação do registrador destino é ativado, e o número da unidade virtual alocada é copiada para o campo de *tag* correspondente. Um nó com a identificação da unidade funcional e da unidade virtual que recebeu a instrução é inserido em uma lista de instruções despachadas. Estes dois últimos passos correspondem à operação do estágio de decodificação. Seguindo a definição do modelo bloqueante, quando uma instrução de desvio é encontrada neste passo, o despacho de novas instruções é bloqueado até que a instrução de desvio seja executada e chegue ao estágio de propagação.

(4) em seguida é chamada a rotina `exec()` que procura instruções despachadas com todas as dependências resolvidas e que podem entrar em execução. Para tanto, a rotina percorre a lista de instruções despachadas, e usa a identificação de unidade funcional e unidade virtual em cada nó para verificar se a unidade virtual contém uma instrução pronta e se a unidade funcional correspondente está livre. Se estas duas condições são satisfeitas, a instrução é marcada como em execução. Um tratamento especial é dado às instruções de acesso à memória. Quando uma tal instrução é encontrada, é aplicado o algoritmo de remoção de ambigüidades para determinar se o acesso pode ser executado naquele momento (ver Capítulo 3). Se houver uma dependencia ou condição de ambigüidade, a execução da instrução é postergada. Caso contrário, é verificado se a locação de memória referenciada encontra-se na memória cache. Esta verificação determina a latência (de *hit* ou *miss*) atribuída à instrução. Para estas operações, é consultado o *trace* de acessos memória para se obter o endereço referenciado na execução real da instrução.

(5) a rotina `decLat()` é chamada para decrementar a latência de todas as instruções que estão em execução. Esta rotina procura na lista de instruções despachadas aquelas que se encontram em execução, e decrementa a latência da instrução. Instruções cuja latência torna-se nula após o decremento são marcadas como executadas. Estes dois últimos passos correspondem ao estágio de execução do *pipeline*.

(6) a rotina `propaga()` é chamada para propagar os resultados das instruções completadas. Esta rotina procura na lista de instruções despachadas aquelas que estão marcadas como executadas. Para cada instrução encontrada nesta condição a rotina procura, usando também a lista de instruções despachadas, aquelas que dependem da

instrução executada. Usando a identificação de unidade virtual no nó da lista, o *tag* na unidade virtual é liberado. O bit de alocação do registrador destino da instrução é desativado. Finalmente, o nó da instrução propagada é retirado da lista de instruções despachadas. Este passo corresponde ao estágio de propagação do *pipeline*.

Um tratamento especial é dado às instruções de desvio. Em uma implementação real, oestágio de propagação armazena o endereço destino no contador de programa, possivelmente redirecionando o estágio de busca para o destino correto. Para reproduzir este funcionamento, quando uma instrução de desvio é propagada, o *trace* de desvios é consultado para determinar se o desvio foi tomado ou não-tomado na execução real. Se o desvio foi tomado, todas as instruções no *buffer* de busca e na fila de instruções são descartadas, já que no modelo adotado usa-se uma previsão estática de desvio não- tomado. O endereço destino obtido no arquivo de *trace* é usado para redirecionar a busca de novas instruções. O despacho de instruções, que foi bloqueado anteriormente, é liberado.

Apêndice D

Bibliografia

[Acosta 86] Acosta, R. D., J. Kjelstrup, H. C. Torng, *An Instruction Issuing Approach to Enhancing Performance in Multiple Funcional Unit Processors*, IEEE Transactions on Computers (35)9, September 1986, pp. 815-828.

[Adam 74] Adam, T. L., K. M. Chandy, J. R. Dickson, *A Comparison of List Schedules for Parallel Processing Systems*, Communication of the ACM (17)12, December 1974, pp. 685-690.

[Agerwala 87] Agerwala, T., J. Cocke, *High Performance Reduced Instruction Set Processors*, Technical Report RC 12434, IBM Thomas J. Watson Research Center, January 1987.

[Aho 88] Aho, A. V., R. Sethi, J. D. Ulman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1988.

[Alpert 93] Alpert, D., D. Avnon, *Architecture of the Pentium Microprocessor*, IEEE Micro (13)3, June 1995, pp.11-21.

[Amdahl 64] Amdahl, G. M., G. A. Blaauw, F. P. Brooks Jr., *Architecture of the IBM System/360*, IBM Journal of Research and Development (8)2, April 1964, pp. 87-101.

[Anderson 67] Anderson, S. F. *et al.*, *The IBM System/360 Model 91: Floating-Point Execution Unit*, IBM Journal of Research and Development (11)1, January 1967, pp. 34-53.

[Asprey 93] Asprey, T. *et al.*, *Performance Features of the PA7100 Microprocessor*, IEEE Micro (13)3, June 1993, pp. 11-21.

[Bashteen 91] Bashteen, A., I. Lui, J. Mullan, *A Superpipeline Approach to the MIPS Architecture*, Proceedings of the COMPCON, 1991, pp.8-12.

[Becker 93] Becker, M. *et al.*, *The PowerPC 601 Microprocessor*, IEEE Micro (13)5, October 1993, pp. 54-68.

[Bell 71] Bell, C. G., A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, NY, 1971.

[Blanck 92] Blanck, G., S. Krueger, *The SuperSPARC Microprocessor*, Proceedings of the COMPCON, 1992, pp.136-141.

[Burgess 94] Burgess, B. *et al.*, *The PowerPC 603 Microprocessor*, Communications of the ACM (37)6, June 1994, pp. 34-42.

[Buleo93] Buleo, Fernando M., *Efeito do Escalonamento Dinâmico no Desempenho de Processadores Super Escalares*, Tese de M.Sc., Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, 1993.

[Butler 91] Butler, M., Y. N. Patt, *Single Instruction Paralelism is Greater than Two*, Proceedings of the 18th International Symposium on Computer Architecture, 1991, pp. 276-286.

[Chang 91] Chang, P. P. *et al.*, *Comparing Static and Dynamic Code Scheduling for Multiple Instruction Issue Processors*, Proceedings of the 24th Symposium on Microarchitecture and Microprogramming, 1991, pp. 25-33.

[Childs 84] Childs, R. E. *et al.*, *A Processor Family for Personal Computers*, Proceedings of the IEEE (72)3, March 1984, pp. 342-351.

[Crawford 93] Crawford, J., D. Alpert, B. Fu, *An Overview of Intel's Pentium Processor*, The Distinguished Lecture Series VI, University Video Communications, 1993.

[Davidson 81] Davidson, S., *et al.*, *Some Experiments on Local Microcode Compaction for Horizontal Machines*, IEEE Transactions on Computers (30)7, July 1981, pp. 460-477.

[Diefendorff 92] Diefendorff, K., M. Allen, *Organization of the Motorola M88110 Superscalar RISC Microprocessor*, IEEE Micro (12)2, April 1992, pp. 40-63.

[Diefendorff 94] Diefendorff, K., *History of the PowerPC Architecture*, Communications of the ACM (37)6, June 1994, pp. 28-33.

[Dwyer 92] Dwyer, H., H. C. Torng, *An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts*, Proceedings of the 25th Annual International Symposium on Microarchitecture, 1992, pp. 272-281.

[Ebcioglu 91] Ebcioglu, K., R. Groves, *Some Global Compiler Optimizations and Architectural Features for Improving Performance of Superscalars*, Proceedings of the 24th Annual International Symposium on Microarchitecture, 1991, pp. 1-13.

[Fernandes 92] Fernandes, E. S. T., F. M. B. Barbosa, *Effects of Building Blocks on the Performance of Super-Scalar Architectures*, Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992, pp. 36-45.

[Ferrante 87] Ferrante, J., K. Ottenstein, J. Warren, *The Program Dependence Graph and its Use in Optimization*, ACM Transactions on Programming Languages and Systems (9)4, September 1987, pp. 319-349.

[Fisher 81] Fisher, J. A., *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Transactions on Computers (30)7, July 1981, pp. 478-490.

[Fisher 83] Fisher, J. A., *Very Long Instruction Word Architectures and the ELI-512*, Proceedings of the 10th Annual International Symposium on Computer Architecture, 1983, pp. 140-150.

[Forster 72] Forster, C. C., E. M. Riseman, *Percolation of Code to Enhance Parallel Dispatching and Execution*, IEEE Transactions on Computers (21)12, December 1972, pp. 1411-1415.

[Garner 88] Garner, R. B. *et al.*, *The Scalable Processor Architecture (SPARC)*, Proceedings of the COMPCON, 1988, pp. 278-283.

[Golumbic 90] Golumbic, M. C., V. Rainish, *Instruction Scheduling Beyond Basic Blocks*, IBM Journal of Research and Development (34)1, January 1990, pp. 93-97.

[Grohoski 90] Grohoski, G. F., *Machine Organization of the IBM RISC System/6000*

Processor, IBM Journal of Research and Development (34)1, January 1990, pp. 37-58.

[Gross 82] Gross, T. R., J. L. Hennessy, *Optimizing Delayed Branches*, Proceedings of the 15th Annual International Symposium on Microarchitecture, 1982, pp. 114-120.

[Hennessy 83] Hennessy, J., T. Gross, *Postpass Code Optimization of Pipeline Constraints*, ACM Transactions on Programming Languages and Systems (5)3, July 1983, pp. 422-448.

[Hennessy 90] Hennessy, J. L., D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffman Publishers Inc., Palo Alto, CA, 1990.

[Hinton 89] Hinton, G., *80960 - Next Generation*, Proceedings of the COMPCON, 1989, pp. 13-17.

[Hsing93] Hsing, Tse H., *Efeito da Predição de Desvios e da Interrupção Precisa no Desempenho de Processadores Super Escalares*,

[Hwu 86] Hwu, W., Y. N. Patt, *HPSm: A High Performance Restricted Data Flow Architecture Having Minimal Functionality*, Proceeding of the 13th Annual Symposium on Computer Architecture, 1986, pp. 297-307.

[IBM 94] PowerPC 604 RISC Microprocessor Technical Summary, IBM Order Number MPR604TSU-01, April 1994.

[Johnson 91] Johnson, M., *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Jouppi 89] Jouppi, N. P., D. W. Wall, *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines*, Proceeedings of the Third Conference on Architectural Support for Programming Languages and Operating Systems, 1989, pp. 272-282.

[Katevenis 85] Katevenis, M. G. H., *Reduced Instruction Set Computer Architectures for VLSI*, ACM Doctoral Dissertation Award, The MIT Press, Cambridge, MA, 1985.

[Keller 75] Keller, R. M., *Look-Ahead Processors*, ACM Computing Surveys (7)4, December 1975, pp. 177-195.

[Kogge 81] Kogge, P. M., *The Architecture of Pipelined Computers*, McGraw-Hill Co.,

New York, N.Y., 1981.

[Kohn 89] Kohn, L., N. Margulis, *Introducing the Intel i860 64-Bit Microprocessor*, IEEE Micro (9)4, August 1989, pp. 15-30.

[Lam 88] Lam, M. S., *Software Pipelining: An Effective Scheduling for VLIW Machines*, Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1988, pp. 318-328.

[Lam 90] Lam, M.S., *Instruction Scheduling for Superscalar Architectures*, Annual Review of Computer Science, Vol. 4, 1990, pp. 173-201.

[Landskov 80] Landskov, D. *et al.*, *Local Microcode Compaction Techniques*, ACM Computing Surveys (12)3, September 1980, pp. 261-294.

[Lee 84] Lee, J. K. F., A. J. Smith, *Branch Prediction Strategies and Branch Target Buffer Design*, IEEE Computer (17)1, January 1984, pp. 6-22.

[Lilja 88] Lilja, D. J., *Reducing the Branch Penalty in Pipelined Processors*, IEEE Computer (21)7, July 1988, pp. 47-55.

[McFarling 86] McFarling, S., J. Hennessy, *Reducing the Cost of Branches*, Proceedings of the 13th Annual Symposium on Computer Achitecture, 1986, pp. 396-304.

[McLellan 93] McLellan, E., *The Alpha AXP Architecture and 21064 Processor*, IEEE Micro (13)3, June 1993, pp. 36-47.

[Mirapuri 92] Mirapuri, S., M. Woodacre, N. Vasseghi, *The MIPS R4000 Processor*, IEEE Micro (12)2, April 1992, pp. 10-22.

[Muchnik 88] Muchnick, S. S., *Optimizing Compilers for SPARC*, Sun Technology, Summer 1988, pp. 64-77.

[Namjoo 88] Namjoo, M. *et al.*, *CMOS Gate Array Implementation of the SPARC Architecture*, Proceedings of the COMPCON, 1988, pp. 10-13.

[Nicolau 84] Nicolau, A., J. A. Fisher, *Measuring the Parallelism Available for Very Long Instruction Word Architectures*, IEEE Transactions on Computers (33)11, November 1984, pp. 968-976.

[Oehler 90] Oehler, R. R., R. D. Groves, *IBM RISC System/6000 Processor Archi-*

tecture, IBM Journal of Research and Development (34)1, January 1990, pp. 23-36.

[Oehler 92] Oehler, R., M. W. Blasgen, *Evolution of the PowerPC Architecture*, The Distinguished Lecture Series V, University Video Communications, 1992.

[Patt 85a] Patt, Y. N., W. Hwu, M. Shebanow, *HPS: A New Microarchitecture: Rationale and Introduction*, Proceedings of the 18th Annual Workshop on Microprogramming, 1985, pp. 103-108.

[Patt 85b] Patt, Y. N. *et al.*, *Critical Issues Regarding HPS, A High Performance Microarchitecture*, Proceedings of the 18th Annual Workshop on Microprogramming, 1985, pp. 109-116.

[Patterson 85] Patterson, D. A., *Reduced Instruction Set Computers*, Communications of the ACM (28)1, January 1985, pp. 8-21.

[Pleszkun 88] Pleszkun, A. R., G. S. Sohi, *The Performance Potential of Multiple Functional Unit Processors*, Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp. 37-44.

[Ramamoorthy 77] Ramamoorthy, C. V., H. F. Li, *Pipeline Architecture*, ACM Computing Surveys (9)1, March 1977, pp. 61-102.

[Saini 93] Saini, A., *An Overview of the Intel Pentium Processor*, Proceedings of the COMPCON, 1993, pp. 60-71.

[Silbey 88] Silbey, A. A., V. M. Milutinovic, *Advanced Microprocessors and High-Level Language Processor Architectures*, in *Computer Architecture: Concepts and Systems*, V. M. Milutinovic Ed., Elsevier Science Publishing Co., New York, NY, 1988.

[Sites 92] Sites, R. L., *Alpha Architecture*, The Distinguished Lecture Series IV, University Video Communications, 1992.

[Sites 93] Sites, R. L., *Alpha AXP Architecture*, Communications of the ACM (36)2, February 1993, pp. 33-44.

[Smith 88] Smith, J. E., A. R. Pleszkun, *Implementing Precise Interrupts in Pipelined Processors*, IEEE Transactions on Computers (37)5, May 1988, pp. 562-573.

[Smith 89] Smith, M. D., M. Johnson, M. A. Horowitz, *Limits on Multiple Instruction*

Issue, Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, 1989, pp. 290-302.

[Smotherman 91] Smotherman, M. *et al.*, *Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling*, Proceedings of the 18th International Symposium on Computer Architecture, 1991, pp. 93-102.

[Sohi 87] Sohi, G. S., S. Vajapeyam, *Instruction Issue Logic for High Performance Interruptable Pipelined Processors*, Proceedings of the 14th Annual Symposium on Computer Architecture, 1987, pp. 27-34.

[Sohi 90] Sohi, G. S., *Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers*, IEEE Transactions on Computers (39)3, March 1990, pp. 349-359.

[Song 94] Song, S. P., M. Denman, *The PowerPC 604 RISC Microprocessor*, Internal Technical Information, Somerset Design Center, March 1994.

[SPEC 92a] SPEC Steering Committee, *SPEC INT92 Release V1.1 Technical Manual*, 1992.

[SPEC 92b] SPEC Steering Committee, *SPEC FP92 Release V1.1 Technical Manual*, 1992.

[Stallings 88] Stallings, W., *Reduced Instruction Set Architecture*, Proceedings of the IEEE (76)1, January 1988, pp. 38-55.

[Sun 87] Sun Microsystems, Inc. *The SPARC Architecture Manual*, Version 7, Mountain View, CA, 1987.

[Tamir 83] Tamir, Y., C. H. Squin, *Strategies for Managing the Register File in RISC*, IEEE Transactions on Computer Systems (32)11, November 1983, pp. 977-988.

[Thornton 64] Thornton, J. E., *Parallel Operation in Control Data 6600*, Proceedings of the AFIPS Conference 26, 1964, pp. 33-40.

[Thornton 70] Thornton, J. E., *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, IL, 1970.

[Tjaden 70] Tjaden, G. S., M. J. Flynn, *Detection and Parallel Execution of Inde-*

pendent Instructions, IEEE Transactions on Computers (19)10, October 1970, pp. 889-895.

[Tjaden 73] Tjaden, G. S., M. J. Flynn, *Representation of Concurrency with Ordering Matrices*, IEEE Transactions on Computers (22)8, August 1973, pp. 752-761.

[Tomasulo 67] Tomasulo, R. M., *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal of Research and Development (11)1, January 1967, pp. 25-33.

[Ullah 93] Ullah, N., M. Holle, *The MC88110 Implementation of Precise Exceptions in a Superscalar Architecture*, ACM Computer Architecture News (21)1, March 1993, pp. 15-25.

[Wall 91] Wall, D. W., *Limits of Instruction-Level Parallelism*, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 176-188.

[Wang 93] Wang, C.-J., F. Emnett, *Implementing Precise Interruptions in Pipelined RISC Processors*, IEEE Micro (13)4, August 1993, pp. 36-43.

[Warren 90] Warren Jr., H. S., *Instruction Scheduling for the IBM RISC System/6000 processor*, IBM Journal of Research and Development (34)1, January 1990, pp. 85-92.

[Weiss 87] Weiss, S., J. E. Smith, *A Study of Scalar Compilation Techniques for Pipelined Supercomputers*, Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, 1987, pp. 105-109.