UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
MESTRADO EM INFORMÁTICA

FERNANDO LÍBIO LEITE ALMEIDA

Escalonamento Dinâmico de Caminhos de Execução em Blocos de Instruções Dataflow

Vitória
2007

FERNANDO LÍBIO LEITE ALMEIDA

Escalonamento Dinâmico de Caminhos de Execução em Blocos de Instruções Dataflow

Dissertação apresentada ao Mestrado de Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Prof. Dr. Alberto Ferreira De Souza.

Vitória
2007

FERNANDO LÍBIO LEITE ALMEIDA

Escalonamento Dinâmico de Caminhos de Execução em Blocos de Instruções Dataflow

Dissertação apresentada ao Mestrado de Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Mestre em Informática.

Aprovada em 30 de Junho de 2007.

COMISSÃO EXAMINADORA

Prof. Dr. Aberto Ferreira de Souza
Universidade Federal do Espírito Santo
Orientador

Prof. Dr. Sérgio Antônio Andrade de Freitas
Universidade Federal do Espírito Santo

Prof. Dr. Edil Severiano Tavares Fernandes
Universidade Federal do Rio de Janeiro

Vitória
2007

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

Almeida, Fernando Líbio Leite, 1980-

A447e Escalonamento dinâmico de caminhos de execução em blocos de instruções Dataflow / Fernando Líbio Leite Almeida. – 2007.

60 f. : il.

Orientador: Alberto Ferreira de Souza.

Dissertação (mestrado) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Arquitetura de computador. 2. Processamento paralelo (Computadores). 3. Escalonamento dinâmico Dataflow. I. Souza, Alberto Ferreira de. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 004

Para Márcia e Maria Áurea.

AGRADECIMENTOS

Gostaria de agradecer à minha família, que sempre me apoiou em todos os momentos, e me possibilitou mais esta conquista. Em especial, à minha mulher Márcia e à minha mãe Maria Áurea, pela dedicação, atenção, incentivo e compreensão durante o desenvolvimento deste trabalho.

Agradeço também ao meu orientador Alberto, por todo apoio e companheirismo investidos neste trabalho.

Meus agradecimentos ao Colegiado do Mestrado em Informática, pela oportunidade de defender esta dissertação.

EPÍGRAFE

"A diferença entre o possível e o impossível está na vontade humana."

Louis Pasteur.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Neste trabalho de pesquisa são investigados novos mecanismos de detecção dinâmica (durante a execução do código) de oportunidades para o escalonamento de instruções para execução em paralelo não fortemente afetados pela latência da hierarquia de memória. Este trabalho de pesquisa foi motivado por resultados recentes obtidos com nossos simuladores da arquitetura *Dynamically Trace Scheduling VLIW* (DTSVLIW). Esta arquitetura explora a localidade dinâmica de execução do código para extrair paralelismo no nível de instrução (*Instruction Level-Parallelism* – ILP). Estudos experimentais mostram que a DTSVLIW sofre mais fortemente os efeitos da latência da hierarquia de memória que as arquiteturas Super Escalar e *Trace Cache*. Contudo, sem estes efeitos, a DTSVLIW teria um desempenho significativamente melhor que o das arquiteturas Super Escalar e *Trace Cache*, tanto em termos de sua capacidade de exploração do ILP, quanto da sua eficiência energética (consumo de energia por instrução).

O objetivo deste trabalho de pesquisa foi investigar mecanismos que permitam a tradução dinâmica, via *hardware*, de código escalar de arquiteturas do conjunto de instruções existentes para código EDGE (*Explicit Data Graph Execution*), para posterior execução em uma máquina EDGE. Para isso foi utilizada uma abordagem experimental, com uso de ambientes de simulação de arquiteturas do conjunto de instruções disponíveis publicamente e com código aberto, além da implementação de um novo simulador, baseado no simulador simplescalar.

ABSTRACT

In this research work we investigate new mechanisms of dynamic detention (during code execution) of opportunities for scheduling instructions for execution in parallel not strongly affected by the latency of the memory hierarchy. This work was motivated by recent results obtained with our simulators of the Dynamically Trace Scheduling VLIW (DTSVLIW) architecture. This architecture takes advantage of code execution locality to extract Instruction Level-Parallelism - ILP. In our experimental studies we observed that the latency of the memory hierarchy has a stronger impact on the DTSVLIW than on the Superscalar and Trace Cache architectures. However, without the memory hierarchy impact, the DTSVLIW would have a performance significantly better than that of the Superscalar or Trace Cache in terms of ILP or in terms of energy efficiency (consumption of energy per instruction).

The goal of this research work was to investigate mechanisms to allow the dynamic translation, via hardware, of scalar code of existent instruction set architectures (ISA) to Explicit Data Graph Execution (EDGE) code, for posterior execution in a EDGE machine also dynamically. For this we have used an experimental methodology, taking advantage of freely available simulators of existing ISA and a new simulator, based on the simplescalar, which we have implemented.

1 Introdução

O contínuo aumento do número máximo de transistores que se pode colocar em um único circuito integrado tem permitido a implementação de processadores com arquiteturas cada vez mais elaboradas e poderosas. A arquitetura da grande maioria dos processadores de alto desempenho de uso geral atuais é Super Escalar [Johnson91] ou Super Escalar com *Trace Cache* [Rotenberg96]. Em tais arquiteturas o alto desempenho advém, em grande parte, da exploração do paralelismo no nível de instrução (*Instruction-Level Parallelism* – ILP) existente nos programas [Wall93]. Para tirar proveito deste ILP, várias instruções são escalonadas para execução paralela nas diversas unidades funcionais do processador a cada ciclo de relógio.

A arquitetura teórica *Dynamically Trace Scheduling VLIW* (DTSVLIW) [DeSouza98, DeSouza00, Almeida03, Pedroni04] também escalona dinamicamente as instruções para execução paralela, mas não as executa paralelamente de imediato. Uma máquina, implementada segundo a arquitetura DTSVLIW, busca instruções escalares, uma a uma, da memória *cache* de instruções e as executa utilizando um processador *pipelined* simples – o Processador Primário da arquitetura (Figura 1.1). Adicionalmente, sua Unidade de Escalonamento realiza o escalonamento dinâmico do caminho produzido pela execução destas instruções escalares, montando, com isso, instruções longas (*Very Long Instruction Word* – VLIW [Fisher84]). Estas instruções VLIW são agrupadas em blocos e armazenadas em uma cache de instruções VLIW. Se um mesmo trecho de código necessitar ser executado novamente, as instruções deste trecho serão fornecidas pela cache VLIW e executadas por uma Máquina VLIW, parte da arquitetura DTSVLIW. A Figura 1.1 apresenta o diagrama de blocos da arquitetura DTSVLIW.

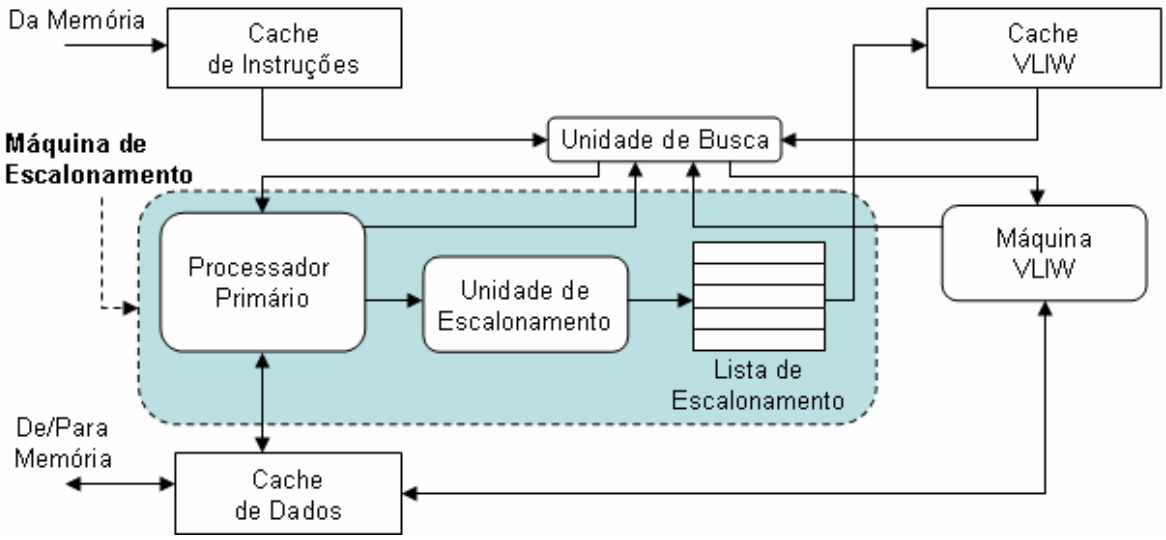


Figura 1.1: A Arquitetura DTSVLIW

Durante a maior parte do tempo de execução (mais que 95% dos ciclos no caso do SPEC2000 rodando em uma máquina DTSVLIW factível com a tecnologia atual [Freitas03]), a cache VLIW contém o código a ser executado e o escalonamento não precisa ser refeito. A simplicidade da unidade principal de execução de código da arquitetura DTSVLIW (a Máquina VLIW) e o fato do escalonamento não necessitar ser feito na maioria dos ciclos faz com que um processador implementado segundo esta arquitetura tenha um consumo de energia (e conseqüentemente dissipação de calor) significativamente menor que um processador equivalente implementado segundo a arquitetura Super Escalar [Pedroni04]. Na verdade, estudos demonstraram que a arquitetura DTSVLIW oferece também melhor desempenho (em termos de ILP) e é mais simples de implementar que as arquiteturas *DIF* [DeSouza00], Super Escalar [DeSouza00], *Trace Cache* [Freitas03] e *EPIC* [Santana03].

Contudo, resultados recentes mostraram que a arquitetura DTSVLIW possui uma capacidade mais limitada de inibir o efeito negativo no seu desempenho imposto pela latência do acesso à memória principal, quando comparada a algumas de suas contemporâneas [Almeida03]. A latência da hierarquia de memória afeta mais fortemente a DTSVLIW porque, no caso de uma falta (*miss*) na *cache* de dados de nível 1 (L1), sua máquina VLIW precisa parar até que a hierarquia de memória entregue o dado solicitado; enquanto que, no caso das arquiteturas Super Escalar, *Trace Cache* e *EPIC*, novas instruções podem ser trazidas e enviadas para execução neste caso.

Em busca de reduzir o impacto da latência da hierarquia de memória no desempenho DTSVLIW, foi desenvolvida uma versão desta arquitetura (DTSVLIW) com múltiplos contextos de execução implementados em *hardware* [Almeida04, Rounce06, Rounce07]. Uma máquina com múltiplos contextos implementados em *hardware*, ou *multithreaded* [Thekkath94, Tullsen95],

possui duas ou mais réplicas das estruturas internas (registradores, basicamente) responsáveis por armazenar o estado da máquina (contexto). Assim, a máquina pode passar da execução de um programa para a de outro muito rapidamente. Com esta capacidade, ao detectar uma falta nas *caches* que force um acesso à memória principal, a máquina pode trocar o programa em execução na esperança de encontrar outro programa em condições de executar instruções úteis. Resultados mostraram que o uso de múltiplos contextos de execução implementados em hardware pode ser uma alternativa para mitigar o efeito negativo da latência da hierarquia de memória sobre o desempenho DTSVLIW [Almeida04, Rounce06, Rounce07]. Contudo, as arquiteturas Super Escalar e *Trace Cache* também podem ser implementadas com múltiplos contextos e, deste modo, manter sua vantagem sobre a DTSVLIW em circunstâncias onde a latência da hierarquia de memória é muito grande.

1.1 Motivação

A capacidade limitada da arquitetura DTSVLIW de lidar com a latência variável da hierarquia de memória foi motivação para a realização este trabalho de pesquisa, onde buscou-se desenvolver uma nova arquitetura, capaz de lidar com a latência variável da hierarquia de memória, mas também de preservar outras características importantes da DTSVLIW, como o escalonamento dinâmico de caminhos por hardware e a compatibilidade de código para trás (*backward code compatibility*) [DeSouza98].

1.2 Objetivos

Para quase a totalidade das Arquiteturas do Conjunto de Instruções (*Instruction Set Architecture* – ISA) existentes atualmente, inclusive as VLIW, o código (programa) é uma lista de instruções e estas são especificadas na forma *operação(ões) operando 1, ..., operando n*, sendo que a(s) *operação(ões)* é(são) executada(s) sobre parte dos *operandos* e seu(s) resultado(s) é(são) eventualmente armazenado(s) em um (ou mais) deles. Cada instrução deve ser executada atomicamente (ou todos os efeitos resultantes da execução atômica devem ser preservados). Os operandos das intruções são tipicamente registradores da ISA. Além disso, a unidade funcional onde a instrução vai executar não é especificada, exceto no caso das arquiteturas VLIW, onde a posição de cada instrução na instrução longa determina em qual unidade funcional será feita sua execução.

Recentemente uma nova classe de ISAs, chamadas de *Explicit Data Graph Execution* (EDGE) [Burger04], foi proposta. Em uma ISA EDGE os programas são grafos e as instruções são os nós do grafo e podem ser executadas tão logo seus operandos de entrada estejam prontos. Mas eles não

precisam ser especificados explicitamente. Além disso, a unidade funcional onde a instrução será executada é especificada e também as unidades funcionais que receberão os resultados da operação. ISAs EDGE possuem, assim, uma característica principal: a comunicação direta entre instruções (que determina a seqüência de execução). Com esta comunicação direta as instruções podem executar na ordem do fluxo de dados (máquinas EDGE são, então, *dataflow* [Davis79, Veen86]), cada instrução iniciando assim que suas entradas estejam prontas. Isto permite a geração de código mais eficiente e a implementação de máquinas EDGE que consomem menos energia que máquinas atuais equivalentes em termos de hardware, uma vez que menos acessos precisam ser feitos ao banco de registradores (acessos aos bancos de registradores consomem uma quantidade significativa de energia [Pedroni04]).

O principal objetivo deste trabalho de pesquisa foi investigar novas arquiteturas de processador que permitissem a tradução dinâmica, via hardware, de código escalar de ISAs existentes para código *dataflow*, para posterior execução em uma máquina *dataflow* também dinamicamente. Esta arquitetura emprestaria características da DTSVLIW e de outras arquiteturas existentes com o propósito de mitigar o efeito negativo da latência da hierarquia de memória e alcançar desempenho superior ao DTSVLIW. Para alcançar o objetivo, usamos nossa experiência com a arquitetura DTSVLIW que, de modo equivalente ao que propusemos investigar, traduz código escalar para código VLIW e posteriormente executa este código em modo VLIW, dinamicamente.

1.3 Resultados Alcançados

Como resultado deste trabalho de pesquisa, foi desenvolvida e avaliada uma nova arquitetura de processador, que permite obter desempenho superior ao DTSVLIW na execução dos programas, usando escalonamento dinâmico por hardware como a DTSVLIW, e é mais tolerante à latência da hierarquia de memória do que a DTSVLIW.

1.4 Organização desta Dissertação

Após esta introdução, a nova arquitetura proposta, a arquitetura *Dynamically Trace Scheduling Dataflow* – ou DTSD, é apresentada no capítulo 2,. No capítulo 3 é detalhada a metodologia empregada na avaliação experimental da DTSD, enquanto que, no capítulo 4, os trabalhos correlatos são discutidos e uma análise crítica deste trabalho de pesquisa é feita. Por fim, no capítulo 5, são apresentadas as conclusões e propostas de trabalhos futuros.

2 Escalonamento Dinâmico *Dataflow*

Arquiteturas de alto desempenho atuais se valem de muitas técnicas para obter maior desempenho, seja por *software*, como a de desenrolamento de *loops*, seja por *hardware*, como a de predição de desvios. Dentre elas, podemos destacar a exploração do paralelismo no nível de instrução (*Instruction-Level Paralelism* – ILP [Wall93]). A exploração do ILP pode ser feita de diferentes maneiras. Em máquinas Super Escalares [Johnson91] (Figura 2.1), as instruções são trazidas continuamente da cache de instruções e colocadas em uma janela de instruções onde, a cada ciclo de máquina, várias delas são analisadas (para identificar quais podem ser executadas em paralelo), selecionadas e enviadas para execução paralela. Um hardware de escalonamento dinâmico de instruções é usado para fazer a análise, seleção e envio das instruções para execução.

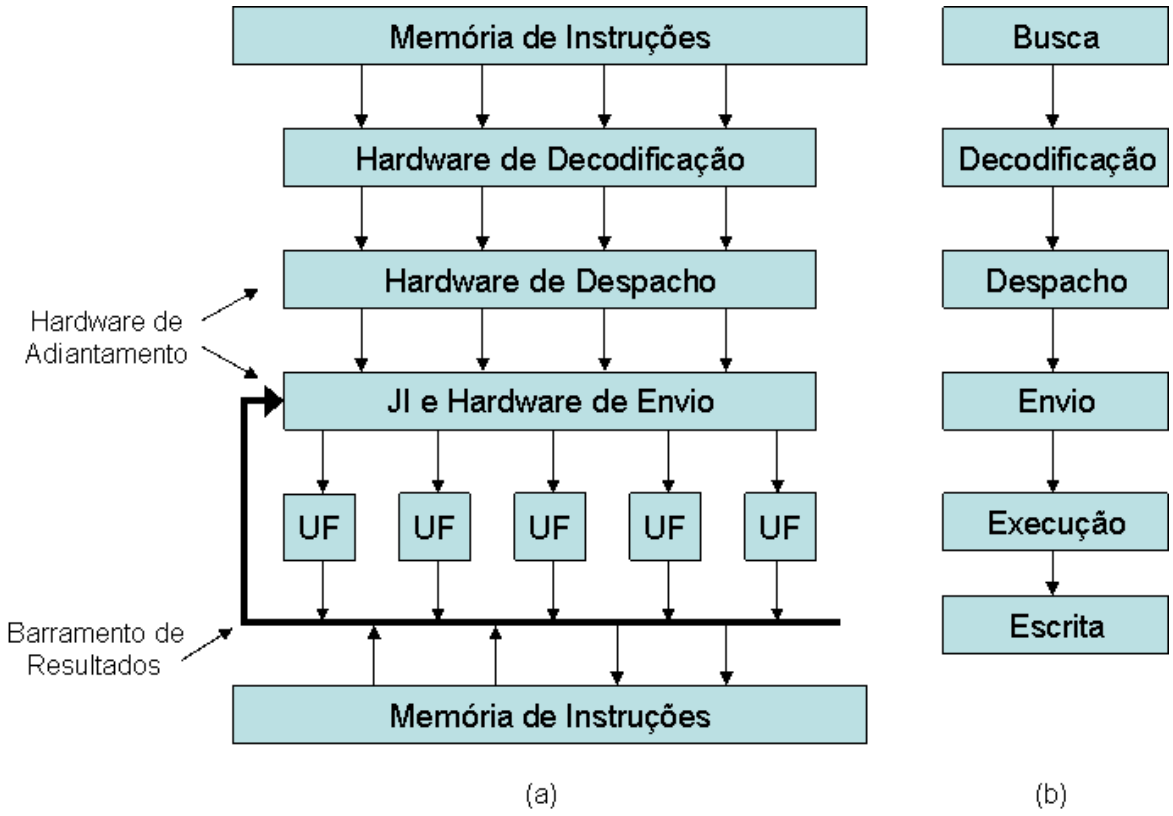
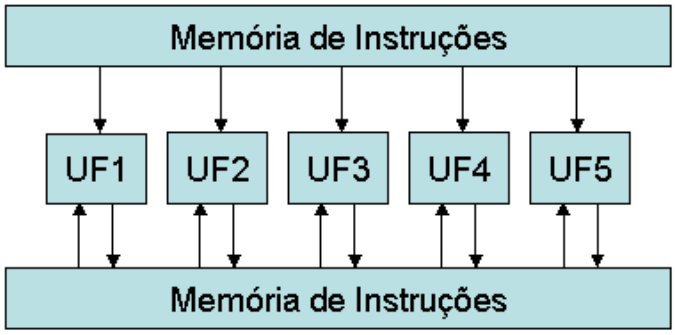


Figura 2.1: Máquina Superscalar. (a) Um caminho de dados Super Escalar simples (JI significa Janela de Instruções e UF significa Unidade Funcional). (b) Um *pipeline* de execução Super Escalar simples.

Em máquinas do tipo *Very Long Instruction Word* (VLIW [Fisher84], Figura 2.2), os blocos de instruções que podem ser executadas em paralelo (instruções longas) são pré-escalonados pelo compilador. Nestas máquinas, as instruções VLIW são continuamente trazidas da cache de instruções e enviadas para execução em um *hardware* paralelo simples, já que não são incorporadas facilidades de escalonamento dinâmico de instruções.



(a)

Ciclo de Clock	0	1	2	3	4	5	6	7
Unidade Funcional 1	I1			I12	I13	I16	I19	
Unidade Funcional 2	I2	I5	I8			I17	I20	I24
Unidade Funcional 3	I3		I9		I14		I21	I25
Unidade Funcional 4	I4	I6	I10				I22	
Unidade Funcional 5	I5	I7	I11		I15	I18	I23	I26

(b)

Figura 2.2: (a) Máquina VLIW hipotética. (b) O compilador é capaz de colocar cinco instruções em cada instrução longa. Se uma posição de instrução não pode ser ocupada, o compilador a deixa vazia.

Uma desvantagem das máquinas VLIW é a incompatibilidade de código entre gerações diferentes de uma mesma arquitetura (ausência de compatibilidade de código para trás), resultado do escalonamento estático de instruções VLIW feito pelo compilador para um hardware VLIW específico [Rau93b]. Outra desvantagem, também resultante do escalonamento estático, está associada às instruções de latência variável, como as de leitura e escrita na memória (por larga margem as instruções de latência variável de ocorrência mais comum durante a execução dos programas). O compilador precisa usar a latência mínima destas instruções durante o escalonamento e a máquina precisa interromper a execução paralela de instruções VLIW todas as vezes que uma destas instruções é encontrada e possui latência maior que a usada pelo compilador durante o escalonamento (como no caso de faltas na cache, por exemplo).

Um mecanismo eficiente de escalonamento dinâmico de instruções VLIW foi proposto por De Souza para a arquitetura *Dynamically Trace Scheduling VLIW* (DTSVLIW [DeSouza00]). Nesta

arquitetura, um hardware simples escalona dinamicamente as instruções escalares trazidas de uma cache de instruções escalares dentro de blocos de instruções VLIW, que são salvos em uma cache VLIW. No caso de um mesmo segmento de código ser revisitado durante a execução, estes blocos podem ser trazidos da cache VLIW e executados por uma máquina VLIW. Experimentos mostraram que a arquitetura DTSVLIW, quando parametrizada segundo a tecnologia existente atualmente, passa mais de 95% dos ciclos executando código VLIW [Freitas03].

O mecanismo de escalonamento dinâmico de código empregado pela arquitetura DTSVLIW permite implementar máquinas com compatibilidade de código para trás. Contudo, máquinas DTSVLIW também têm seu desempenho comprometido por instruções de latência variável que, quando encontradas durante a execução de um bloco, provocam a interrupção da execução VLIW até que a latência exigida por elas seja completada, no caso de ser diferente da latência mínima que, como no caso de máquinas VLIW, é usada por máquinas DTSVLIW durante o escalonamento [DeSouza00].

As máquinas Super Escalares têm compatibilidade de código para trás e não precisam interromper a execução sempre que uma instrução de latência variável apresenta uma latência maior que a mínima. Contudo, a complexidade do hardware necessário para o escalonamento Super Escalar de um grande número de instruções simultâneas é um fator limitante na implementação de máquinas Super Escalares de alto desempenho.

Máquinas que empreguem a arquitetura *Explicit Data Graph Execution* (EDGE [Burger04]) possuem vantagens típicas das arquiteturas VLIW e Super Escalar. Elas exploram o ILP com um hardware simples, já que as instruções são escalonadas para execução paralela pelo compilador EDGE e não pelo hardware. Além disso, elas toleram latência variável porque seu mecanismo de execução é *dataflow* (a execução das instruções é ativada pela disponibilidade dos operandos requeridos por elas). A seguir, discutimos mais pormenorizadamente a arquitetura EDGE.

2.1 A arquitetura EDGE

As instruções de máquinas RISC ou CISC atuais são caracterizadas por especificar um código de operação e os nomes dos operandos fonte e destino desta operação. Diferentemente, em arquiteturas EDGE uma instrução é caracterizada por especificar onde uma operação deve ser executada (em qual unidade funcional específica) e que outras instruções devem receber o resultado desta operação; além disso, uma instrução só é elegível para execução quando todos os seus operandos de entrada estão prontos. A execução EDGE ocorre de acordo com o fluxo de dados entre as instruções (execução *dataflow* [Davis79, Veen86]).

Assim como as máquinas VLIW, as máquinas EDGE não possuem compatibilidade de código para trás. Uma vez que um código foi compilado para uma dada máquina EDGE, este mesmo código só poderá ser executado nesta máquina ou versões suas posteriores (onde tenham sido tomados os cuidados necessários para isso), não sendo possível seu aproveitamento em máquinas anteriores que possuam menos unidades funcionais, já que as instruções EDGE especificam qual unidade funcional deve executá-las. Além disso, máquinas EDGE não executam código RISC ou CISC existente e vice-versa.

O principal objetivo deste trabalho é o desenvolvimento de arquiteturas capazes de transformar código escalar em código *dataflow* e de executar este código *dataflow* dinamicamente. Estas arquiteturas receberam o nome de *Dynamically Trace Scheduling Dataflow* – ou DTSD. A seguir é apresentada a proposta da arquitetura DTSD.

2.2 Uma Arquitetura DTSD

Um diagrama de blocos de uma arquitetura DTSD pode ser visto na Figura 2.3.

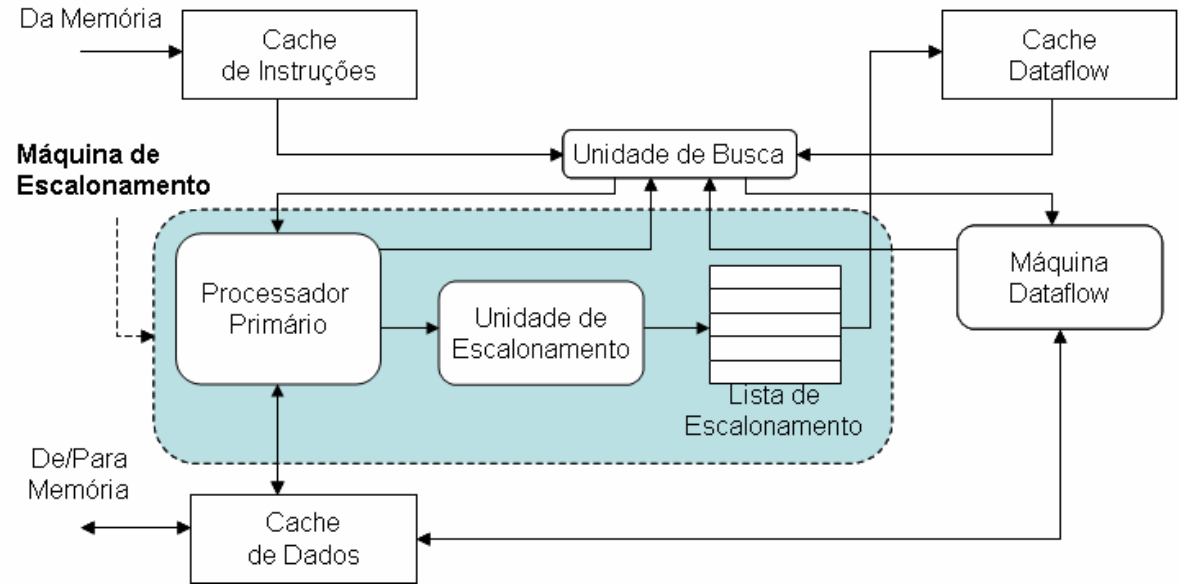


Figura 2.3: Diagrama de blocos de uma Arquitetura DTSD (*Dynamically Trace Scheduling Dataflow*).

Nesta arquitetura, instruções escalares são trazidas, uma a uma, da cache de instruções e executas por um processador *pipelined* simples – o Processador Primário da arquitetura (Figura 2.3). Adicionalmente, sua Unidade de Escalonamento realiza o escalonamento dinâmico *dataflow* do caminho produzido pela execução destas instruções escalares, montando, com isso, blocos de instruções *dataflow*; estes blocos são armazenados em uma cache de blocos de instruções *dataflow*. Se um mesmo trecho de código necessitar ser executado novamente, as instruções deste trecho

podem ser fornecidas pela cache *dataflow* e executadas pela Máquina *Dataflow* da arquitetura (Figura 2.3).

Para estudar a arquitetura DTSD foi proposta uma *dataflow* ISA, baseada na Alpha ISA [Compaq99], para ser interpretada pela Máquina *Dataflow*. A Alpha ISA foi escolhida como base por sua simplicidade, por ser bastante estudada na literatura e por nossa experiência anterior com ela, oriunda da pesquisa sobre arquiteturas DTSVLIW.

Na ISA *dataflow* proposta:

- As instruções são agrupadas em blocos e são numeradas de acordo com sua posição no bloco;
- Uma instrução especifica o código da operação que ela executa, seus operandos de entrada e os que recebem o resultado da operação, quantos *tokens* ela precisa receber para poder executar, e as instruções para as quais ela envia *tokens*;
- Instruções não lêem ou escrevem em registradores (exceto aquelas com este fim específico);
- Uma instrução está pronta para ser executada quando recebe todos os *tokens* de que precisa, ou se não depende de nenhum token.

Na ISA proposta os *tokens* recebidos informam a disponibilidade dos dados necessários para a execução de cada instrução.

O trecho de código escalar Alpha da Figura 2.4 pode ser transformando para o código *dataflow* mostrado na Figura 2.5.

```
ADD   R1, R2, R3  # R1 = R2 + R3
SUB   R4, R1, R5
ADD   R1, R4, R6
```

Figura 2.4: Simples trecho de código Alpha

```
1.RR OP1,R2:0:5          # lê o conteúdo de R2 (Read Register – RR) para
                           OP1, ou seja, OP1 = R2, nenhum token
                           necessário, envia um token para a instrução 5
2.RR OP2,R3:0:5
3.RR OP3,R5:0:6
4.RR OP4,R6:0:7
5.ADD OP1,OP1,OP2:2:6    # OP1=OP1+OP2, 2 tokens necessários, envia um
                           token para a instrução 6, recebe um token
6.SUB OP1,OP1,OP3:2:7,9  # OP1=OP1-OP3, 2 tokens necessários, envia
                           tokens para as instruções 7 e 9
7.ADD OP2,OP1,OP4:2:8    # OP2=OP1+OP4, 2 tokens necessários, envia um
                           token para a instrução 8
8.WR R1,OP2:1:           # Escreve o conteúdo de OP2 em R1 (Write
                           Register - WR), R1 = OP2, 1 token
                           necessário, a instrução não envia nenhum token
9.WR R4,OP1:1:
```

Figura 2.5: Código Alpha transformado em *dataflow*

No código *dataflow* da Figura 2.5, as instruções são numeradas de 1 a 9. Ao lado do número aparece o código da operação comandada por cada instrução e seus operandos de saída e entrada. Após o primeiro caractere ":" é indicado o número de *tokens* que a instrução precisa receber antes de poder executar e, após o segundo, a lista de instruções para as quais a instrução envia *tokens*.

As instruções de 1 a 4 são responsáveis pela leitura de registradores que serão operados pelas instruções 5, 6 e 7. Todas as quatro (instruções de 1 a 4) podem ser executadas em paralelo e copiam os registradores da ISA (R2, R3, R5 e R6) para registradores de renomeação (OP1 a OP4). A renomeção é realizada durante a transformação do código seqüencial em *dataflow* (escalonamento) e elimina as dependências de saída e anti-dependências.

A instrução 5 pode executar tão logo as instruções 1 e 2 terminem, enquanto que a 6 depende das instruções 3 e 5. A instrução 7 depende da 4 e da 6, a 8 da 7, e a 9 da 6. Note que no código Alpha são feitas 6 leituras e 3 escritas nos registradores da Alpha ISA, enquanto que no código *dataflow* são feitas apenas 4 leituras e 2 escritas. Contudo, supondo que as instruções Alpha e *dataflow* necessitem de apenas um ciclo para executar, o código *dataflow* requer mais ciclos que o Alpha – 5 ciclos contra 3 –, como mostra a seqüência de execução *dataflow* da Figura 2.6 (cada linha contém as instruções que podem ser executadas no ciclo).

Ciclo 1:	1.RR OP1, R2:0:5	2.RR OP2, R3:0:5	3.RR OP3, R5:0:6	4.RR OP4, R6:0:7
Ciclo 2:	5.ADD OP1, OP1, OP2:2:6			
Ciclo 3:	6.SUB OP1, OP1, OP3:2:7,9			
Ciclo 4:	7.ADD OP2, OP1, OP4:2:8	9.WR R4, OP1:1:		
Ciclo 5:	8.WR R1, OP2:1:			

Figura 2.6: Seqüência de execução *dataflow*

No entanto, é importante observar que, em uma máquina Alpha, a leitura e a escrita nos registradores toma tempo, que é apenas explicitado no código *dataflow*. Além disso, em blocos *dataflow* maiores as leituras e escritas em registradores da Alpha ISA são necessárias apenas no início e no final da execução do bloco, respectivamente, o que reduz o número de acessos ao banco de registradores. Os registradores de renomeção OP1, OP2, OP3 e OP4 podem ser armazenados em um banco de registradores menor e, eventualmente, com um número menor de portas de leitura e escrita, o que resultaria em um tempo de acesso e consumo de energia menores.

2.3 Instruções de Desvio Condicional e Execução Especulativa

A Figura 2.7 mostra o diagrama de um trecho de código de um programa hipotético que segue por um caminho (seqüência de instruções) A e, de acordo com o resultado de um teste T (uma instrução de desvio condicional), pode seguir pelo caminho B ou passar a executar a partir do início do caminho C.

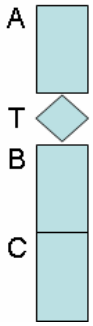


Figura 2.7: Um desvio condicional.

Em um processador seqüencial comum, a execução segue pelo caminho A até o desvio condicional T e, em seguida, por um dos caminhos B ou C. Uma forma de se obter maior desempenho do que a simples execução seqüencial é especular que um dos caminhos, B ou C, será executado. O caminho escolhido é então executado paralelamente de antemão, antes mesmo que o teste da condição do desvio tenha sido realizado (caso não haja dependências verdadeiras que impeçam esta execução). Caso a escolha feita tenha sido a correta, parte do caminho escolhido já foi executado e ganha-se o tempo que seria gasto por uma execução seqüencial. Caso contrário, o caminho correto é então executado, da mesma maneira que em uma execução seqüencial. Esta técnica é conhecida como execução especulativa.

2.3.1 Execução especulativa: Tratamento de um único desvio

Para exemplificar como o escalonamento de instruções de desvio condicional e execução especulativa são realizados na arquitetura DTSD, o trecho de código da Figura 2.8 é utilizado, onde o registrador mais a esquerda é o de escrita.

```
        ADD R3, R1, R2
Loop:   SUB R5, R3, R4
        ADD R4, R4, 1
        BNE R5, Loop
Exit:
```

Figura 2.8: Trecho de código Alpha com desvio condicional

Considerando que os registradores R1, R2 e R4 do trecho de código da Figura 2.8 possuam os valores 2, 3 e 4, respectivamente, obteremos o trecho de código *dataflow* da Figura 2.9, produzido dinamicamente a partir da tradução do caminho de execução definido pelos valores iniciais dos registradores R1, R2 e R4.

```
1. RR OP1, R1:0:3
2. RR OP2, R2:0:3
3. ADD OP3, OP1, OP2:2:4,6
4. WR OP3, R3:1:                  # Depende apenas do resultado do ADD
5. RR OP4, R4:0:6,8
6. SUB OP5, OP3, OP4:2:7,10
7. WR OP5, R5:2:                  # Depende também do desvio
8. ADD OP6, OP4, 1:1:9,11,13
9. WR OP6, R4:2:                  # Depende também do desvio
10. BNE OP5, LOOP:1:7,9           # Até o primeiro desvio condicional as instruções
                                  possuem ordem zero; toda vez que um desvio
                                  condicional é escalonado, as instruções seguintes
                                  passam a ter a ordem do desvio + 1; ao passo que
                                  instruções posteriores ao desvio são escalonadas,
                                  caso estas escrevam um registrador que já seria
                                  escrito antes deste desvio, este desvio passa a
                                  emitir um token para a instrução de escrita
                                  anterior para confirmá-la ou cancelá-la e tal
                                  escrita passa a depender também deste desvio;
11. SUB OP7, OP3, OP6:2:12,15
12. WR OP7, R5:1:
13. ADD OP8, OP6, 1:1:14
14. WR OP8, R4:1:
15. BNE OP7, LOOP:1:12,14
Exit:
```

Figura 2.9: Código Alpha com desvio condicional transformado em *dataflow*

Na ISA *dataflow* proposta, uma instrução de desvio condicional "possui memória" do que ela experimentou (se ela foi um desvio tomado ou não) durante a execução do caminho do qual ela fez parte (executado pelo Processador Primário da arquitetura DTSD, Figura 2.3). Caso a mesma condição se verifique novamente durante a execução *dataflow*, o desvio envia *tokens* que cancelam escritas anteriores que possam ser dispensadas, ou seja, escritas cujos registradores serão sobrescritos posteriormente. Mais do que simplesmente dispensáveis, tais escritas necessitam de fato ser canceladas, dado que a execução fora de ordem pode fazer com que uma escrita posterior venha a ficar pronta para execução antes de outra. Assim, se a ordem das escritas nos registradores não for preservada, a execução *dataflow* ficará incorreta.

Quando a condição observada pelo desvio durante o escalonamento *dataflow* não é verificada na execução *dataflow*, o desvio comanda uma mudança do fluxo de controle e seus tokens atuam como tokens usuais, validando a execução das instruções de escrita anteriores. Isto porque os resultados produzidos até o momento anterior ao desvio têm de ser escritos para garantir o correto estado da máquina. Quando o endereço alvo da mudança de fluxo de controle for o de um bloco *dataflow* já escalonado, a execução *dataflow* continua neste próximo bloco. Caso contrário, o Processador Primário da máquina DTSD assume o controle e escalona outro(s) bloco(s). Após algum tempo, a máquina DTSD passa a maioria dos ciclos executando em modo *dataflow*.

Como mencionado anteriormente, *tokens* enviados por instruções de desvio podem ser de dois tipos: (i) que habilitam a execução das instruções alvo dos *tokens*, e (ii) que anulam a execução das instruções alvo. Esta informação de tipo pode ser implementada por um bit, parte do *token*. Nos

tokens enviados por instruções que não são desvio condicional este bit sempre indica que a instrução alvo do *token* deve ser executada.

Desvios condicionais estabelecem, também, grupos de instruções sob sua influência. Um grupo de instruções é caracterizado pelo seu número de ordem. O número de ordem de cada instruções é armazenado junto com ela durante o escalonamento. Em cada bloco, o número de ordem inicia em zero, sendo incrementado a cada desvio condicional escalonado. O bit de tipo de *token* e a ordem das instruções são necessários para controlar a execução de escritas no banco de registradores (mais detalhes a seguir).

A DTSD atua como um interpretador da ISA original. Quando executando em modo *dataflow*, o estado da ISA original, representado pelo conteúdo dos registradores da ISA e pelo estado da memória, progride do mesmo modo que faria em uma execução escalar, sendo que a maioria de suas alterações ocorre apenas quando da transição de um bloco *dataflow* para outro ou para o Processador Primário. As informações transmitidas pelas instruções de desvio são usadas para garantir o correto estado do banco de registradores ao fim da execução de cada bloco.

O código *dataflow* apresentado anteriormente pode ser executado como mostrado na Figura 2.10 (as instruções 7 e 9 recebem *tokens* que as cancelam e não são executadas, dadas as condições iniciais apresentadas). Note que as instruções 11 e 13 são executadas especulativamente (antes de o desvio 10 ser resolvido). Isto é possível porque, apesar de elas estarem sendo executadas previamente, a confirmação de seu resultado só se dará com as escritas 12 e 14, que só são executadas após a verificação do desvio 15.

Ciclo 1	1.RR OP1, R1:0:3	2.RR OP2, R2:0:3	5.RR OP4, R4:0:6,8	
Ciclo 2	3.ADD OP3, OP1, OP2:2:4,6,11	8.ADD OP6, OP4, 1:1:9,11,13		
Ciclo 3	4.WR OP3, R3:1:	6.SUB OP5, OP3, OP4:2:7,10	11.SUB OP7, OP3, OP6:2:12	13.ADD OP8, OP6, 1:1:14
Ciclo 4	10.BNE OP5, LOOP:1:7,9			
Ciclo 5	15.BNE OP7, LOOP:1:12,14			
Ciclo 6	12.WR OP7, R5:1:	14.WR OP8, R4:1:		

Figura 2.10: Seqüência de execução *dataflow*. As instruções marcadas (de 11 a 15) possuem ordem 1 enquanto que as demais possuem ordem 0.

O código original (Figura 2.8) demoraria 7 ciclos em uma máquina escalar (3 por iteração), enquanto que o código *dataflow*, 6 ciclos (considerando leituras e escritas e 4 ciclos sem elas). Note que, se o loop fosse executado e escalonado por mais iterações, o benefício da execução *dataflow* seria ainda maior, uma vez que as instruções de leitura e escrita no banco de registradores da Alpha ISA só ocorrem uma vez e mais instruções poderiam ser executadas em paralelo. Uma análise cuidadosa do código *dataflow* escalonado mostraria que o número de ciclos de cada iteração do loop tende para 1 quando o número iterações escalonadas é elevado. É possível observar também, na Figura 2.10, que instruções do código escalar são executadas fora da ordem original e até mesmo

antes de desvios condicionais previstos no código original, isto é, algumas instruções do código original são executadas especulativamente.

2.3.2 Tratamento de mais de um desvio

Conforme explicado na subseção 2.3.1, desvios condicionais estabelecem um grupo de instruções sob sua influência, caracterizado pelo número de ordem, e podem enviar tokens que cancelam escritas anteriores a ele, para evitar possíveis escritas fora de ordem. Para que isto ocorra, os desvios condicionais precisam ser escalonados com tokens para as escritas anteriores a ele, cujo registrador a ser escrito, será novamente escrito por novas instruções. Como no momento do escalonamento de um desvio condicional não é possível determinar quais registradores serão sobrescritos até escalonarmos as instruções subseqüentes a ele, o meio encontrado para escalonar o desvio com todos os tokens necessários é guardar sua posição no bloco e atualizá-lo com os devidos tokens ao passo que novas instruções vão sendo escalonadas.

Como guardar as posições de todos os desvios condicionais e escritas de registradores para inserir nos desvios os devidos tokens tornaria a implementação em hardware complicada e custosa, decidimos que o escalonamento leva em consideração apenas o último desvio condicional encontrado e as escritas imediatamente anteriores a ele. Isto funciona bem quando olhamos para apenas um único desvio, mas pode ocasionar uma condição que resultaria em um problema. Para exemplificar tal situação diferenciada, olhemos para as três situações possíveis mostradas na Figura 2.11, onde as instruções aparecem de forma simplificada, com apenas um número de identificação, e, no caso das escritas de registradores, com o registrador de destino. As reticências representam possíveis instruções intermediárias que não afetam o exemplo e por isso não são exibidas.

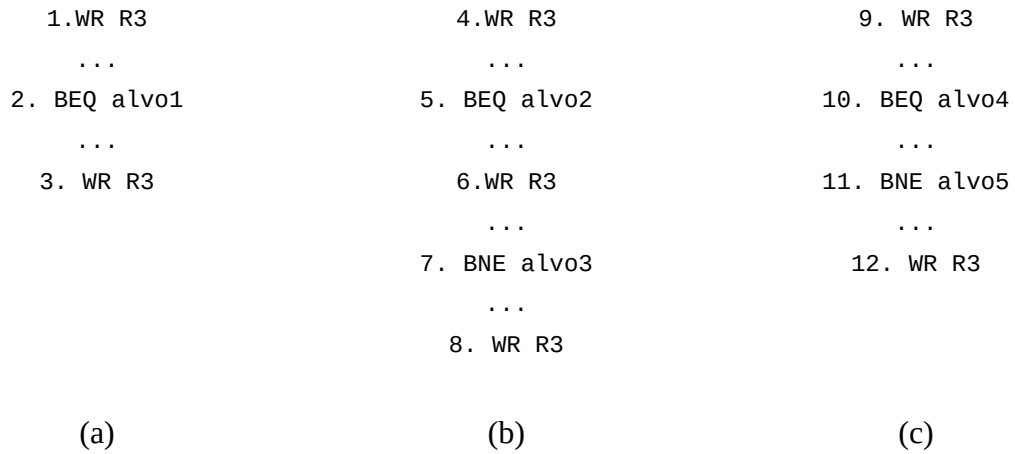


Figura 2.11: Três seqüências com desvios e escritas. (a) Seqüência simples em que temos uma escrita, um desvio condicional e nova escrita (b) Seqüência de escritas e desvios encadeados. (c) Seqüência de escrita, mais de um desvio e nova escrita.

A primeira seqüência, Figura 2.11(a), exibe a condição tratada na subseção anterior, onde há um desvio condicional entre duas escritas para um mesmo registrador. Neste caso, a solução para garantir a correta execução é colocar um token de cancelamento do desvio 2 para a escrita 1. Caso o desvio 2 observe a mesma condição de quando foi escalonado, este envia o token de cancelamento para a escrita 1, e só a escrita 3 acontece. Caso contrário, o desvio envia um token habilitando a escrita 1 e muda o fluxo de controle para alvo1.

Já a Figura 2.11(b), exibe uma seqüência do tipo escrita, desvio, nova escrita, novo desvio, e mais uma escrita, sendo que as três escritas são para o mesmo destino, o registrador R3. Aplicando a mesma lógica para este caso, seria colocado um token no desvio 5 para a escrita 4, e outro no desvio 7 para a escrita 6. Note que não seria colocado um token do desvio 7 para a escrita 4, pois cada desvio só considera as escritas imediatamente anteriores a ele. Caso o desvio 5 não seguisse o mesmo rumo de quando foi escalonado, este enviaria um token confirmando a escrita 4 e comandaria a execução para alvo2. Caso contrário, o desvio 5 enviaria um token que cancelaria a escrita 4. Neste caso, o desvio 7 seria então avaliado e a sua execução em relação à escrita 6 seria análoga à execução do desvio 5 em relação à escrita 4. Contudo, a escrita 8 pode não possuir nenhuma dependência com relação as instruções anteriores e ser executada antes da escrita 4 ou da escrita 6, gerando um erro. Para evitar isso, em ambos os casos (a) e (b) da Figura 2.11, mesmo que as escritas posteriores a um desvio fiquem prontas para executar, estas não são executadas até que o desvio seja decidido, pois possuem ordem maior que o desvio – a informação de ordem é usada para impedir sua execução até que os desvios associados sejam executados. Assim, a execução correta do bloco é garantida, pois não há como inverter a ordem de escritas para um mesmo registrador.

Outra situação a ser considerada é a mostrada na Figura 2.11(c), onde é exemplificada uma seqüência com uma escrita, um desvio, outro desvio e uma nova escrita para o mesmo registrador. Seguindo a mesma lógica para este caso, não colocaríamos token nem no desvio 10, pois não aconteceu uma nova escrita após o mesmo, e nem no desvio 11, pois antes dele não havia uma escrita para o registrador da escrita 12. Neste caso, a escrita 9 não dependerá de nenhum dos desvios, e executará independentemente deles. Já a escrita 12 somente será executada caso o desvio 11 siga o mesmo caminho de quando foi escalonado e somente após esta definição. O problema encontrado neste caso é que a independência da escrita 9 de tokens de cancelamento vindos dos desvios a deixa à mercê da seqüência de instruções que vêm antes dela e produz o resultado que ela deverá escrever no registrador R3. Se tal seqüência levar tempo suficiente para que os dois desvios sejam resolvidos e para que o resultado a ser gravado pela escrita 12 fique pronto, isto provocará a escrita 12 antes da 9, o que seria um erro de execução. Para impedir que tal situação aconteça, durante o escalonamento é mantida uma lista de escritas; guardamos também a ordem (definida

pelos desvios) da última escrita em cada registrador (zero no caso da escrita 9) e sua posição no bloco sendo escalonado. Assim, ao escalonarmos as escrita 12, que possui ordem 2, dado que passaram-se dois desvios condicionais, um token para a escrita 12 é colocado na escrita 9, garantindo que, se os desvios entre elas seguirem o mesmo caminho, tal token garanta que a escrita 12 não aconteça antes da 9. Então, a regra para colocação de um destes tokens é a de que a escrita anterior no mesmo registrador que consta na lista de escrita esteja a dois ou mais desvios para trás da escrita sendo escalonada, o que é calculado através da ordem dos desvios armazenada nas escritas.

Qualquer outra disposição de escritas e desvios pode ser relacionada como uma combinação de uma das aqui exemplificadas, e assim, serão tratadas corretamente.

2.4 Instruções de leitura e escrita na memória e *memory desambiguation*

O trecho de código Alpha da Figura 2.12 soma o valor no registrador R1 a todos os elementos do vetor A, armazenado na memória a partir do endereço em R2, e cujo tamanho está em R3.

```
Loop:   ADD   R5, R2, R4   # calcula o endereço de A(índice)
        LD    R6, 0(R5)    # lê a posição de memória A(índice) para R6
        ADD   R6, R6, R1   # R6 = A(índice) + R1
        ST    R6, 0(R5)    # A(índice) = A(índice) + R1
        ADD   R4, R4, 1    # índice = índice + 1
        SUB   R6, R3, R4   # R6 = tamanho – índice
        BNE   R6, Loop     # se R6 for diferente zero (tamanho diferente
                           de índice), volta para Loop
```

Figura 2.12: Trecho de código Alpha com acesso a memória

Três iterações do código Alpha da Figura 2.12 (que ocorreriam com os valores iniciais R4 = 0 e R3 = 3, por exemplo) podem ser traduzidas para o bloco *dataflow* da Figura 2.13.

```
1.RR OP1,R2:0:5,15,25
2.RR OP2,R4:0:5,9
3.RR OP3,R1:0:7,17,27
4.RR OP4,R3:0:10,20,30

5.ADD OP5,OP1,OP2:2:6,8,12
6.LD OP6,0,OP5:1:7
7.ADD OP6,OP6,OP3:2:8
8.ST OP6,0,OP5:2:
9.ADD OP2,OP2,1:1:10,13,15,19
10.SUB OP6,OP4,OP2:2:11,14
11.BEQ OP6,EXIT:1:12,13,14

12.WR R5,OP5:2:
13.WR R4,OP2:2:
14.WR R6,OP6:2:

15.ADD OP5,OP1,OP2:2:16,18,22
16.LD OP6,0,OP5:1:17
17.ADD OP6,OP6,OP3:2:18
18.ST OP6,0,OP5:2:
19.ADD OP2,OP2,1:1:20,23,25,29
20.SUB OP6,OP4,OP2:2:21,24
21.BEQ OP6,EXIT:1:22,23,24

22.WR R5,OP5:2:
23.WR R4,OP4:2:
24.WR R6,OP6:2:

25.ADD OP5,OP1,OP2:2:26,28,32
26.LD OP6,0,OP5:1:27
27.ADD OP6,OP6,OP3:2:28
28.ST OP6,0,OP5:2:
29.ADD OP2,OP2,1:1:30,33
30.SUB OP6,OP4,OP2:2:31,34
31.BEQ OP6,LOOP:1:32,33,34

32.WR R5,OP5:2:
33.WR R4,OP4:2:
34.WR R6,OP6:2:
```

Figura 2.13: Tradução do código Alpha com acesso a memória para *dataflow*

Da mesma maneira que no exemplo da sub-seção anterior, o código acima pode ser escalonado para executar como no trecho da a seguir:

Ciclo 1:	1.RR OP1,R2:0:5,15,25	2.RR OP2,R4:0:5,9	3.RR OP3,R1:0:7,17,27	4.RR OP4,R3:0:10
Ciclo 2:	5.ADD OP5,OP1,OP2:2:6,8,12	9.ADD OP2,OP2,1:1:10,13,15,19		
Ciclo 3:	6.LD OP6,0,OP5:1:7	10.SUB OP6,OP4,OP2:2:11,14	15.ADD OP5,OP1,OP2:2:16,18,22	19.ADD OP2,OP2,1:1:20,23,25,29
Ciclo 4:	7.ADD OP6,OP6,OP3:2:8	11.BEQ OP6,EXIT:1:12,13,14	16.LD OP6,0,OP5:1:17	20.SUB OP6,OP4,OP2:21,24
	25.ADD OP5,OP1,OP2:2:26,28,32	29.ADD OP2,OP2,1:1:30,33		
Ciclo 5:	8.ST OP6,0,OP5:2:	17.ADD OP6,OP6,OP3:1:18	21.BEQ OP6,EXIT:1:22,23,24	26.LD OP6,0,OP5:1:27
	30.SUB OP6,OP4,OP2:2:31,34			
Ciclo 6:	18.ST OP6,0,OP5:2:	27.ADD OP6,OP6,OP3:28	31.BEQ OP6,LOOP:1:32,33,34	
Ciclo 7:	28.ST OP6,0,OP5:2:	32.WR R5,OP5:2:	33.WR R4,OP4:2:	34.WR R6,OP6:2:

Figura 2.14: Seqüência de execução em modo *dataflow*

Note que, neste caso, não há imposição de limitações de quantidades de unidades funcionais para execução em paralelo das instruções, apenas as limitações impostas pelas dependências de dados. Assim, supondo que cada instrução execute em 1 ciclo, seriam necessários 7 ciclos para executar este trecho, dadas as condições iniciais mencionadas. O código original levaria 21 ciclos para ser executado em uma máquina escalar.

Exceções geradas por *loads* e *stores* só são tratadas após a escrita nos registradores ou posição de memória associados – o tratamento de exceções é discutido na Seção 2.7. Um buffer de leitura e escrita na memória pode ser utilizado para detectar acessos à uma mesma posição de memória fora de ordem (leitura-escrita ou escrita-escrita) – estes casos também são tratados como exceções. Para isso, do mesmo modo que na arquitetura DTSVLIW [DeSouza00] a ordem dos acessos à memória é memorizada no escalonamento e condições de *memory aliazing* são detectadas e tratadas durante a execução.

2.5 Unidades de Execução

A Figura 2.15 mostra o diagrama de blocos da Máquina de *Dataflow* da arquitetura DTSD proposta (à direita – ver Figura 2.3, pág. 8) e, no detalhe (à esquerda), um bloco de unidades funcionais (BUF). A Máquina *Dataflow* da arquitetura DTSD busca instruções de uma cache de blocos *dataflow* e as executa em blocos de unidades funcionais (BUF, três no exemplo da figura). O banco de registradores da ISA (REGS) e uma fila de leitura e escrita na memória (FLE), que apoia o acesso à hierarquia de memória (MEM), completam a Máquina *Dataflow*.

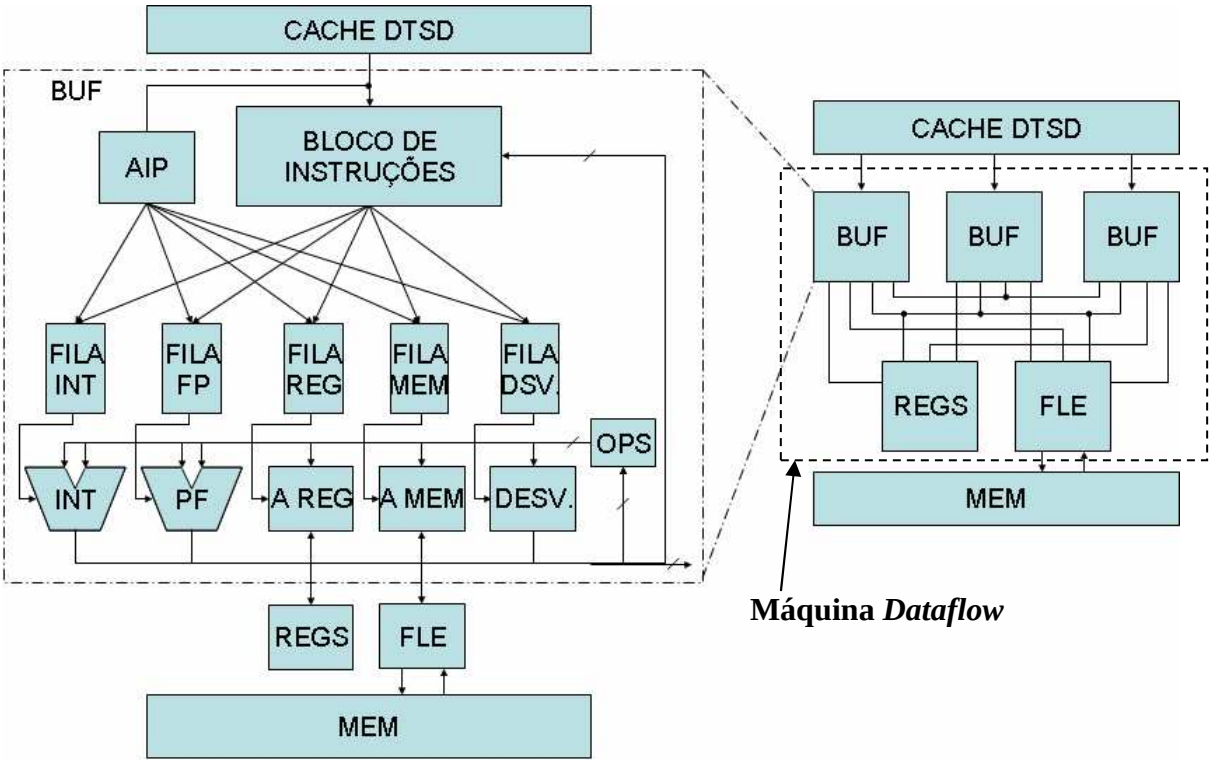


Figura 2.15 Máquina *Dataflow* da arquitetura DTSD

A Máquina *Dataflow* terá tantos BUFs quantos forem convenientes para otimizar o desempenho (número de instruções executadas por ciclo) face às restrições freqüência de relógio e consumo de energia. Cada um deles, a cada ciclo, pode receber parte do bloco (uma coluna) a ser

executado via mecanismo de busca de instruções *dataflow*. A cada ciclo, ao passo em que as instruções são lidas para cada BUF, o hardware de Adiantamento de Instruções Prontas (AIP) detecta quais instruções já vêm prontas para a execução, ou seja, não dependem de nenhum *token* para executar (como, por exemplo, as de leitura de registradores – RR), e as coloca diretamente na fila de instruções prontas correspondente (FILA INT, FILA FP, etc.).

O bloco a ser executado não é trazido todo de uma vez em um único ciclo da cache DTSD, mas sim uma linha de cache por vez. Isto é feito do mesmo modo que na arquitetura DTSVLIW [DeSouza00]. Cada linha do cache pode conter uma fração do número de instruções do bloco. Instruções que dependem de *tokens* para executar levam consigo o número de *tokens*, ou seja, de operandos de que necessitam para executar. Como o bloco não é trazido todo de uma vez da cache, em alguns casos é possível que um *token* seja enviado para uma instrução que ainda não foi lida da cache. Dessa forma, para garantir a execução correta, a estrutura onde ficará o bloco lido da cache contempla um contador de *tokens* recebidos para cada instrução do bloco, que inclusive pode conter valores negativos. Assim, se, por exemplo, uma instrução depende de um único *token* para executar e este *token* chegar antes dela ser lida da cache, seu contador conterá o valor -1. Ao ser lida, o número de *tokens* necessário é somado ao seu respectivo contador e, neste caso, o contador da instrução passará ao valor zero, o que indicará que esta instrução estará pronta para executar e será então colocada na fila da unidade funcional a que corresponde.

Instruções prontas vão sendo enviadas para as respectivas unidades funcionais na ordem em que chegam em sua fila, conforme a disponibilidade destas; elas lêem dos registradores de renomeação locais a cada BUF (não mostrados na Figura 2.15) antes de serem enviadas para execução. Ao passo que são executadas, seus resultados e *tokens* são enviados conforme seus destinos determinam. Os *tokens*, ao chegar ao Bloco de Instruções de um BUF, poderão tornar uma ou mais instruções prontas para serem executadas, desde que estas estejam aguardando apenas o *token* em questão. Neste caso, as instruções que ficarem prontas pela chegada deste *token* são despachadas automaticamente para a fila da unidade funcional em que executam e este ciclo se repete até que o bloco termine de ser executado.

O término de um bloco se dará quando não houver nenhuma instrução em execução nas unidades funcionais. Esta condição é suficiente dado que, quando um bloco começa a ser executado, a busca da primeira linha de instruções do bloco trará ao menos uma instrução de leitura de registrador. Como esta já vem pronta para execução, ela é enviada diretamente para a unidade de leitura de registradores. Quando esta instrução termina de executar, o envio de seu(s) *token*(s) habilitará ao menos mais uma instrução para execução que, quando executada, habilitará outra e

assim sucessivamente, até que todas as instruções tenham sido executadas ou um desvio siga uma direção diferente da observada durante o escalonamento.

Se um desvio segue uma direção diferente da observada durante o escalonamento ele, ao invés de cancelar as escritas a registrados anteriores a ele, irá habilitá-las e estas serão executadas. Além disto, quando um desvio segue uma direção diferente da observada durante o escalonamento sua ordem é guardada durante a execução deste bloco e utilizada para que, no momento de trazer as instruções para as unidades funcionais, caso estas não possuam ordem menor que a do desvio, as mesmas sejam canceladas.

2.6 Escalonamento de Instruções

O escalonamento de instruções da máquina DTSD proposta é realizado a partir do algoritmo de inserção de instruções na lista de escalonamento apresentado a seguir de maneira simplificada.

```
Para cada operando, Faça
   Se o operando não consta na Lista de Resultados
          Coloca uma instrução de Leitura de Registrador para este operando no Bloco
          A instrução de leitura recebe a Ordem Atual
          Coloca o operando na Lista de Resultados como produzido por esta Leitura de
          Registrador
          Incrementa o ponteiro de Próxima Instrução no bloco
   Fim se
Fim faça
Se há uma instrução de escrita na Lista de Escritas para o resultado produzido e esta possui
Ordem igual a Ordem Atual
   Coloca a instrução sobre tal escrita
Senão
   Coloca a instrução no bloco
   Incrementa o ponteiro de Próxima Instrução no bloco
Fim se
Instrução no bloco recebe a Ordem Atual
Para cada operando, faça
   Coloca um token na Instrução de Leitura deste operando para a Instrução
Fim faça
Se a Instrução produz um resultado
   Se a Instrução substituiu uma escrita
          Remove o token da Instrução que o enviava para tal escrita
   Senão
          Se há uma escrita na Lista de Escritas para o resultado da Instrução com Ordem menor
que a Ordem Atual
                Coloca um token para esta escrita no Último Desvio Condicional
          Fim se
   Fim se
   Coloca o resultado da Instrução na Lista de Resultados
   Coloca uma instrução de Escrita de Registrador para o resultado no Bloco
   Atualiza a Lista de Escritas com esta Escrita
   Incrementa o ponteiro de Próxima Instrução no bloco
Fim se
Se a instrução é um desvio condicional
   Incrementa a Ordem Atual
   Esta instrução passa a ser o Último Desvio Condicional
Fim se
```

Figura 2.16: Algoritmo simplificado de escalonamento de instruções

Aplicando o algoritmo apresentado ao trecho de código da Figura 2.17, teremos como resultado o escalonamento de um bloco ilustrado nas figuras de Figura 2.18 a Figura 2.19.

```
          ADD R3, R1, R2
Loop:     ADD R5, R3, R4
          SUB R4, R4, 1
          BNE R5, Loop
```

Figura 2.17: Código de exemplo para o escalonamento

Nas figuras a seguir (Figura 2.18 a Figura 2.19), são exibidos o bloco sendo escalonado, a Lista de Escritas, a Lista de Resultados e o trecho de código escalonado como exemplo. Ambas as listas, de escritas e de resultados, são mostradas como listas simples de uma forma meramente ilustrativa,

pois estas seriam de fato implementadas como pequenas memórias indexadas a partir dos números dos registradores. Para fins didáticos, listamos nos exemplos apenas até o registrador 5. Além disto, o cruzamento das setas simboliza o registrador que armazena a posição no bloco de instruções onde a próxima instrução deverá ser colocada. No trecho de código à direita, o retângulo tracejado indica a última instrução escalonada. Cada instrução está representada por sua operação, número de *tokens* que ela necessita e entre colchetes a lista de posições para onde ela deve enviar *tokens*.

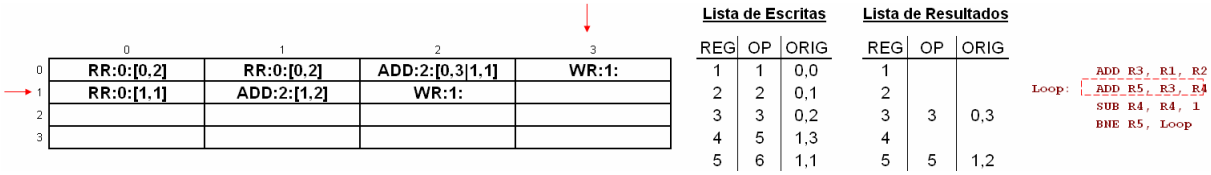


Figura 2.18: Início de escalonamento de um bloco a partir do trecho de código à direita. Aqui o bloco possui largura 4 e profundidade 4. Também são mostradas a Lista de Escritas e a Lista de Resultados. À esquerda é mostrado o bloco após a inserção das primeiras duas instruções. O cruzamento das setas aponta para a posição onde a próxima instrução escalonada deve ser colocada.

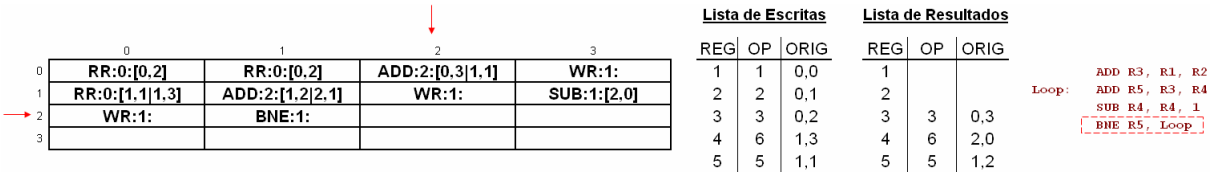


Figura 2.19: Estado do Bloco e das listas após o escalonamento do BNE.

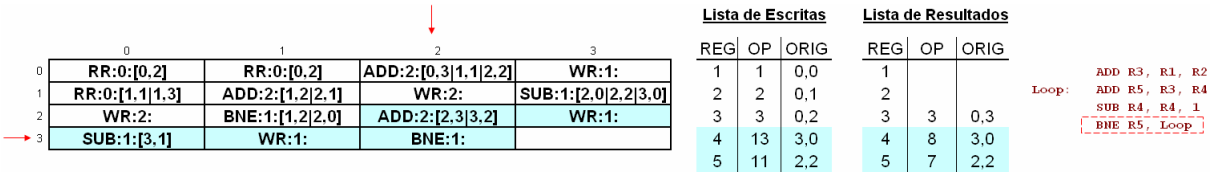


Figura 2.20: Estado do Bloco e das listas após o escalonamento do BNE pela segunda vez. As instruções e as posições marcadas nas listas possuem Ordem 1.

Após o escalonamento do segundo BNE, a próxima instrução a ser escalonada seria novamente o ADD do *Loop*. Porém, como não há espaço para ela, dado que além da operação de ADD seria necessária uma operação de escrita de registradores para seu resultado (WR), este bloco é então finalizado e salvo na cache de blocos DTSD com o endereço do primeiro ADD do trecho de código – o endereçamento e a escrita do bloco são feitos do mesmo modo que na arquitetura DTSVLIW [DeSouza00]. Um novo bloco é então iniciado, onde é inserida a instrução de ADD do *Loop*, que dará o seu endereço para este novo bloco.

2.7 Tratamento de Exceções

Exceções podem ser geradas como resultado da execução de algumas instruções, como *loads/stores* (faltas de página, violações de acesso) ou divisão (divisão por zero). Além disso, em qualquer ponto da execução de um programa podem ocorrer interrupções. Para tratar exceções e interrupções, na arquitetura DTSD as instruções não sinalizam exceções até que uma operação de escrita de seu resultado seja executada para um registrador da ISA ou memória – um bit de exceção é adicionado a cada registrador de renomeação e viabiliza a sinalização de exceções somente nestes casos. Instruções executadas especulativamente que observam condições de exceção ligam este bit e outras instruções que vierem a ler deste registrador de renomeação o propagam. Uma exceção é sinalizada quando um bit de exceção ligado é propagado para um registrador da ISA ou para uma operação de escrita na memória. O mecanismo de tratamento de exceções opera como a seguir.

Na DTSD, assim como a DTSVLIW, o mecanismo de tratamento de exceções conhecido como *Checkpointing* [Hwu87] é utilizado. Um ponto de restauração é determinado ao iniciar a execução de cada bloco, quando todos os registradores da ISA são salvos em registradores específicos para isto. Instruções de escrita na memória executadas fazem com que o conteúdo da memória que elas sobrescrevem sejam salvos na lista de restauração de memória. Esta lista contém o endereço, o conteúdo e o tipo de dado sobrescrito. Caso a máquina DTSD detecte uma exceção durante a execução de um bloco, a unidade de escalonamento entra em modo de recuperação. Neste modo, os registradores da arquitetura recebem os valores salvos no início da execução do bloco e cada entrada da lista de restauração de memória é escrita de volta na memória cache de dados. Caso a exceção tenha sido provocada por uma ambigüidade no acesso à memória, o bloco em questão é invalidado na cache de blocos. Em seguida, a execução volta ao normal.

Em caso de exceção por ambigüidade de memória, a execução continua em modo seqüencial e o novo bloco é escalonado agora de forma a previnir novas exceções deste tipo: a dependência de dados provoca naturalmente a inclusão de *tokens* que evitarão a exceção observada no escalonamento anterior. Para outros tipos de exceções, a execução continua em modo de exceção até que a exceção se repita, ponto a partir do qual o sistema operacional trata a exceção. No modo de exceção, apenas o processador primário executa.

3 Metodologia

Neste trabalho, foi estudada uma arquitetura de escalonamento dinâmico *dataflow*, que recebeu o nome de DTSD. Esta foi implementada em um simulador capaz de executar código Alpha, para avaliação de seu desempenho. O simulador é paramétrico e modela as características da DTSD.

Para colocar os resultados obtidos no contexto dos processadores existentes atualmente, os parâmetros de nosso simulador DTSD foram ajustados de modo a tornar seu núcleo de execução comparável ao do processador Alpha 21264 [Compaq99]. Este processador foi escolhido porque ele segue a arquitetura Super Escalar [Johnson91], usada pela maioria dos processadores de alto desempenho disponíveis atualmente, e por existir um simulador do mesmo, já validado com uma máquina real, disponível publicamente [Desikan01].

Para ampliar o escopo de análise, foram também realizados experimentos com o simulador DTSVLIW, cujos parâmetros também ajustamos de modo a tornar seu núcleo de execução comparável ao do processador Alpha 21264. A arquitetura DTSVLIW também foi simulada porque o modelo de escalonamento dinâmico da DTSD é baseado em seu funcionamento.

Foi possível utilizar um núcleo de execução comum aos simuladores DTSD, DTSVLIW e Super Escalar porque todos são baseados no *simplescalar* [Austin97]. Assim como os simuladores Alpha e DTSVLIW, o simulador DTSD é *execution-driven* – ele simula fielmente a arquitetura DTSD descrita. Todos os simuladores usam a máquina hospedeira para chamadas ao sistema operacional (SO).

As arquiteturas Super Escalar e DTSVLIW não são descritas pormenorizadamente aqui, uma vez que descrições detalhadas das mesmas podem ser encontradas em bons livros texto [Compaq99, Johnson91] ou na literatura [DeSouza00] da área de arquitetura de computadores, respectivamente.

Na avaliação experimental, após quantificar o desempenho da arquitetura DTSD, este foi comparado ao das arquiteturas, DTSVLIW e Super Escalar, para avaliar comparativamente o desempenho da arquitetura DTSD.

3.1 Simuladores Utilizados nos Experimentos

Exceto quando especificado de outra forma, as configurações utilizadas nos experimentos são como as indicadas nas tabelas de 1 a 3. As máquinas DTSVLIW utilizadas usam o mecanismo de

compactação de blocos descrito em [DeSouza01]. Máquinas DTSD não necessitam de compactação de bloco, pois não escalonam instruções *nop* como a DTSVLIW. As *caches DTSD e VLIW* utilizadas são bastante pequenas e tem 48KB (Tabelas 1 e 2). Nós escolhemos este tamanho para que, juntamente com a *Instruction Cache* (16KB), elas tenham tamanho igual ao da cache de instruções do processador Alpha 21264.

Com os programas de teste empregados (Seção 3.3), o simulador DTSVLIW executa, em média, 123.465,6 instruções por segundo em uma máquina com processador Centrino 1.6GHz com 1Gbyte de RAM, enquanto que o simulador DTSD executa 59.526,47 instruções por segundo na mesma máquina e o Super Escalar, 115.976,9.

Tabela 1: Configuração DTSD

Processador Primário	• pipeline de quatro estágios (fetch, decode, execute e write back) • sem hardware de predição de desvios • desvios tomados geram uma bolha de 2 ciclos no pipeline • *Instruction Cache* de 16KB, *2-way set associative*, latência 1
DTSD Cache	48KB, *2-way set associative*, latência 1
Unidades Funcionais	4 de leituras de registradores, 4 de escritas, 4 inteiras, 2 de ponto flutuante, 2 de acesso à memória e 1 de desvio
Tamanho da Lista de Escalonamento	2 vezes o número de Instruções Longas (LI) do bloco

Tabela 2: Configuração DTSVLIW

Processador Primário	• pipeline de quatro estágios (fetch, decode, execute e write back) • sem hardware de predição de desvios • desvios tomados geram uma bolha de 2 ciclos no pipeline • *Instruction Cache* de 16KB, *2-way set associative*, latência 1
VLIW Cache	48KB, *2-way set associative*, latência 1
Unidades Funcionais	4 inteiras e 2 de ponto flutuante
Tamanho da *Scheduling List*	2 vezes o número de LIs do bloco

Tabela 3: Configuração Alpha21264

Pipeline	• 7 estágios – *Fetch*, *Slot*, *Map*, *Issue*, *Regread*, *Execute*, *Write-back* e *Retire*. • *Fetch*, *Slot* e *Map* – 4 instruções por ciclo • *Issue*, *Regread* e *Write-back* – 6 instruções por ciclo • *Retire* – 11 instruções por ciclo
Unidades Funcionais	4 inteiras e 2 de ponto flutuante
Tamanho das *Issue queues*	Instruções Inteiras 20, Ponto Flutuante 15
Preditor de Desvios	*Tournament branch predictor* com uma combinação de três preditores: *two level local preditor* (1024 10-bit *local history*), *path-based global predictor* (12-bit *history register* que aponta para uma tabela de 4K contadores saturados de 2 bits) e um *choice preditor* que escolhe a predição de um dos dois anteriores (4K contadores saturados de 2 bits)

3.2 Parâmetros de Simulação

O único parâmetro de simulação avaliado nos experimentos foi o tamanho do bloco das arquiteturas DTSD e DTSVLIW.

3.3 Programas de Teste

Os programas de teste escolhidos para realização dos experimentos são programas típicos de máquinas de uso geral. São eles:

- Binaria: algoritmo de busca binária [Horowitz78];
- Bolha: algoritmo de ordenação, método da bolha [Knuth73];
- Integral: algoritmo de integração numérica segundo o método do trapézio [Conte65];
- Livermor: algoritmo para determinação do menor componente de um vetor de dimensão *n* [McMahon83];
- Lu: algoritmo de decomposição para solução de equações lineares baseado na Eliminação Gaussiana [Forsythe67];
- Quick: algoritmo de ordenação [Knuth73];

A Tabela 4 apresenta o número de instruções necessárias para a completa execução de cada um dos programas de teste nos simuladores.

Tabela 4: Número de Instruções dos Programas de Teste

Programa de teste	Número de Instruções Executadas
binaria	1780
bolha	51894
integral	322168
livermor	11128
lu	141205
quick	53101

Estes programas foram escritos em linguagem C e encontram-se listados no Apêndice A. Vale destacar que estes programas de teste são simples e não correspondem a programas típicos executados cotidianamente em máquinas atuais. Nós optamos por utilizá-los neste trabalho preliminar de avaliação da arquitetura DTSD porque nosso simulador DTSD ainda não se encontra maduro o suficiente para executar programas mais complexos.

3.4 Métricas

Neste trabalho o número médio de instruções executadas por ciclo (*instructions per cycle* – IPC) é utilizado como medida de desempenho. Dessa forma, é possível abstrair a velocidade de clock empregada em possíveis processadores implementados com as arquiteturas aqui estudadas, dado que, para esta avaliação, seria necessário ao menos o estudo de uma implementação física dos mesmos, o que vai além do escopo deste trabalho.

Na DTSVLIW, instruções adicionais são executadas (instruções de cópia e "nops" – *no operation*) devido ao processo de escalonamento e a *aliasing exceptions*. Usar simplesmente o número de instruções executadas na DTSVLIW inflaria esta medida de desempenho. Para evitar isso, o simulador DTSVLIW incorpora um modo especial de simulação em que, ao mesmo tempo em que um programa é executado na máquina DTSVLIW, ele também é executado em uma máquina escalar. A simulação inicia na máquina DTSVLIW e, a cada instrução de desvio, a máquina escalar é avançada até encontrar o mesmo desvio. Como a ordem das instruções de desvio é preservada pela arquitetura DTSVLIW, uma comparação entre o estado das duas ISAs permite checar se a execução está correta (o teste é importante apenas para depuração do simulador, na verdade). Ao fim de uma simulação, a máquina escalar indica o número de instruções executadas, enquanto que a máquina DTSVLIW indica o número de ciclos necessários para a execução. A razão entre estes dois números é o IPC apresentado nos experimentos.

Já as arquiteturas Alpha e DTSD não executam "nops", apesar de a arquitetura DTSD necessitar executar instruções de cópia nos casos em que uma instrução precisa enviar mais *tokens* do que o parâmetro máximo permitir. Por essa razão, no caso da DTSD, o mesmo mecanismo de medida do IPC (descrito acima) é utilizado.

Os resultados são sumarizados utilizando-se a média harmônica, por ser a mais apropriada quando taxas como o IPC são avaliadas [Jacob95].

4 Experimentos

Neste trabalho foi estudada uma arquitetura DTSD com apenas um Bloco de Unidades Funcionais (BUF). Deixamos o estudo completo com mais de uma BUF para trabalhos futuros devido à complexidade adicional que seria imposta ao simulador DTSD.

Foram examinadas diferentes configurações de tamanho de bloco para as arquiteturas DTSD e DTSVLW, e o desempenho destas arquiteturas com estas configurações foi confrontadas com o do processador Alpha21264.

4.1 Tamanho de Bloco na DTSD e DTSVLIW

Os programas de teste foram executados pelos simuladores das arquiteturas DTSD e DTSVLIW utilizando blocos com 8, 16, 32, 64 e 96 instruções. Nas figuras 4.1 (DTSD) e 4.2 (DTSVLIW), a seguir, exibimos o desempenho destas arquiteturas em cada programa de teste com as diferentes configurações de bloco utilizadas.

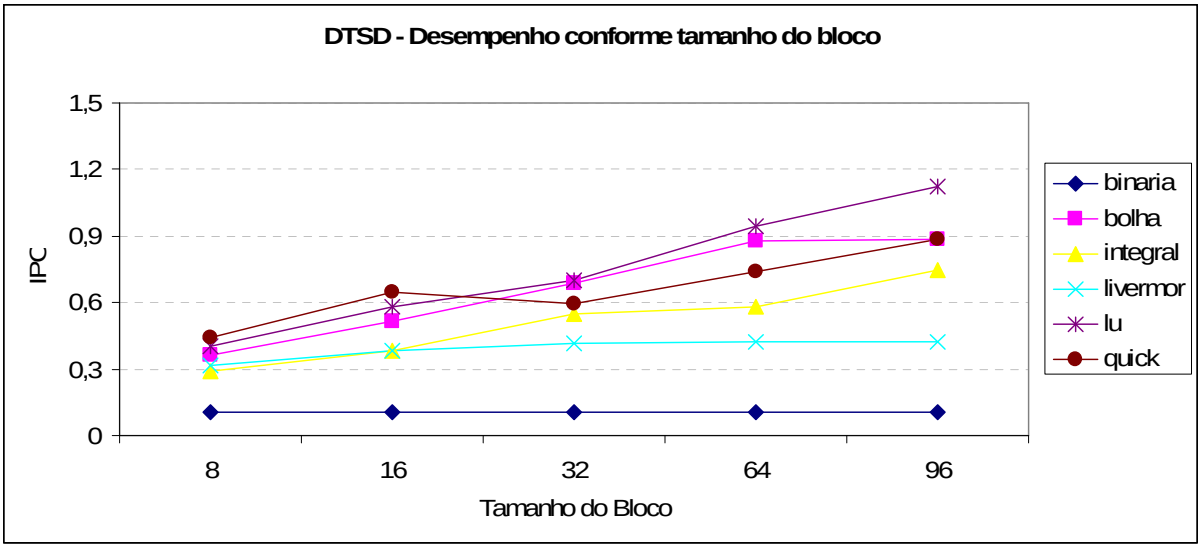


Figura 4.1: Desempenho (em Instruções por Ciclo – IPC) da Arquitetura DTSD para diferentes tamanhos de Bloco

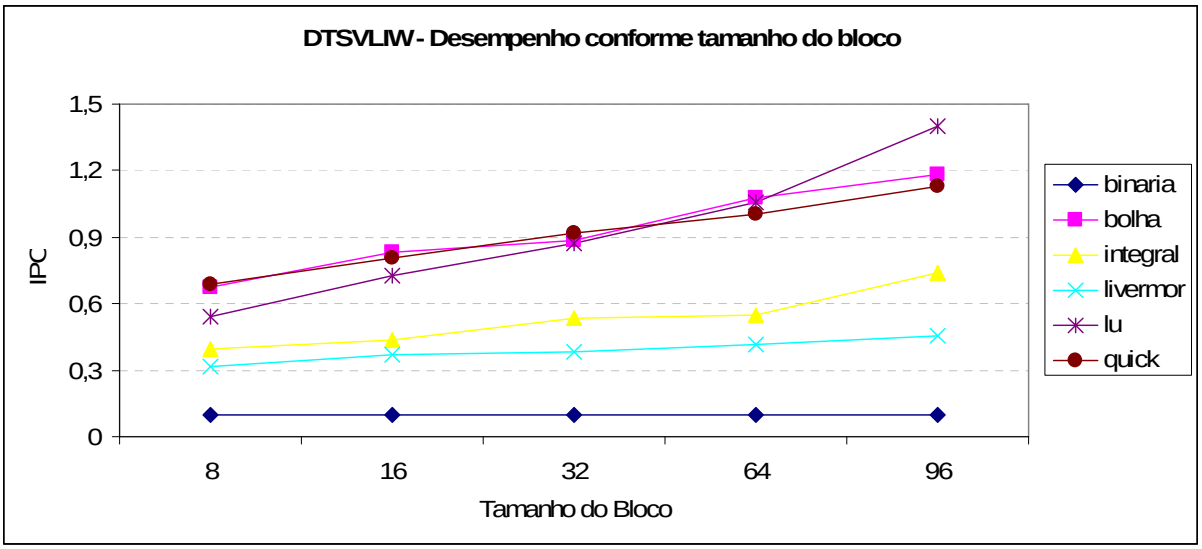


Figura 4.2: Desempenho (em Instruções por Ciclo – IPC) da arquitetura DTSVLIW para diferentes tamanho de bloco

É possível verificar nos gráficos que o desempenho DTSD na execução dos programas de teste não aumenta tanto quanto o desempenho DTSVLIW com o aumento do tamanho do bloco. Além disso, somente três dos programas (*binaria*, *integral* e *quick*) se beneficiaram do aumento do tamanho de bloco de 64 para 96 instruções, no caso da DTSD.

4.2 Desempenho comparativo das arquiteturas

A Figura 4.3 a seguir apresenta o desempenho comparativo das três arquiteturas, quando executando os programas de teste já mencionados. Note que, nos gráficos, o desempenho do processador Alpha21264 é representado por uma linha constante, dado que, para este, não há variação de tamanho de bloco por não possuir esta característica arquitetural.

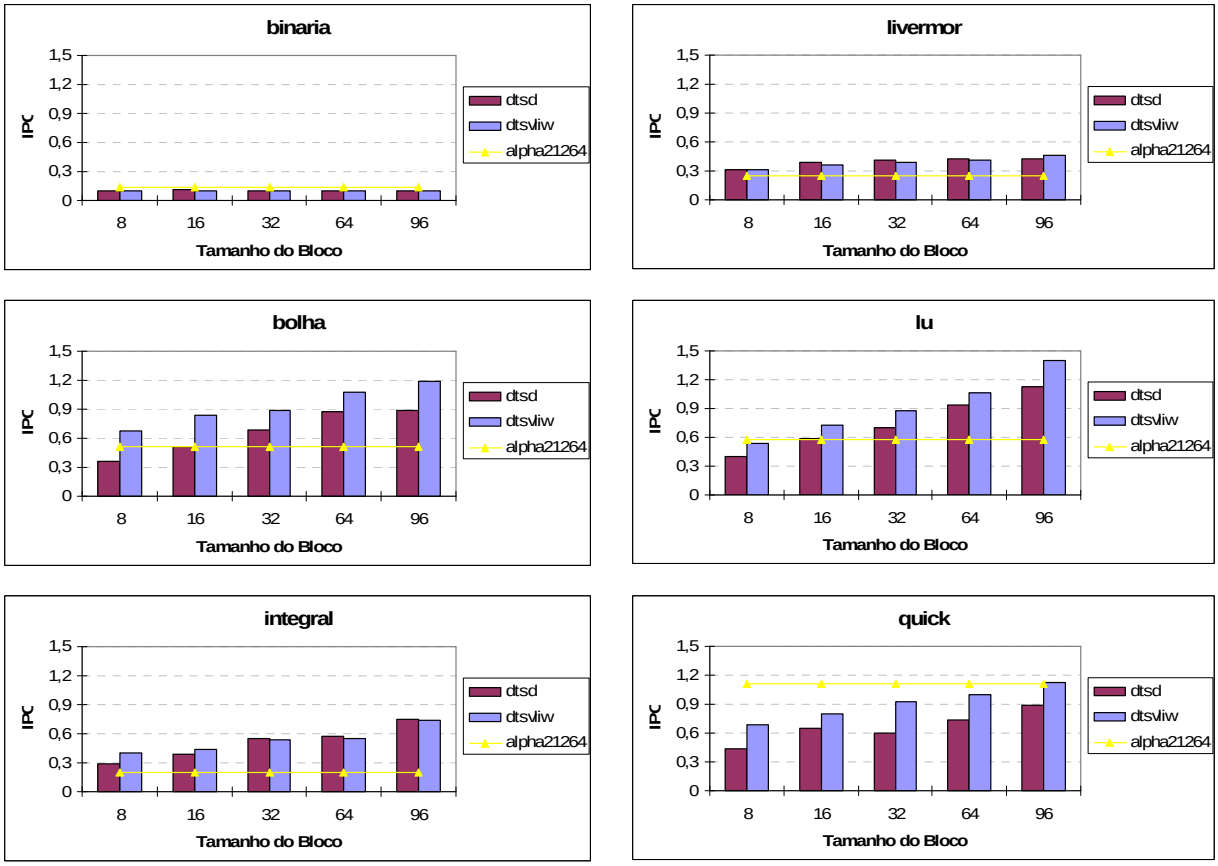


Figura 4.3: Desempenho comparativo entre as arquiteturas (Instruções por Ciclo – IPC)

Para o programa *binaria*, a DTSD obteve desempenho ligeiramente menor do que a Alpha e ligeiramente maior do que a DTSVLIW para todos os tamanhos de bloco. Também podemos observar que, neste caso, as variações de tamanho de bloco examinadas não impactaram no desempenho da DTSD e DTSVLIW.

No caso do programa de teste *bolha*, observamos que o desempenho das arquiteturas DTSD e DTSVLIW aumenta conforme aumentamos o tamanho do bloco. Para blocos com 16, 32, 64 ou 96 instruções, o desempenho das arquiteturas DTSD e DTSVLIW (exceto pela DTSD com bloco de 16 instruções) supera o desempenho da Alpha21264, sendo que, em todos estes casos, a DTSVLIW apresentou desempenho superior ao da DTSD.

O resultado dos experimentos com os programas *integral* e *livermor* mostra que o desempenho do Alpha foi inferior ao das arquiteturas DTSD e DTSVLIW para todos os tamanhos de bloco experimentados. No caso de *integral*, o desempenho da DTSD supera a da DTSVLIW para blocos com 32, 64 e 96 instruções, e no caso do *livermor*, para blocos com 8, 16, 32 e 64 instruções.

Nos experimentos com o programa *lu*, o desempenho do Alpha foi superado pelo das arquiteturas DTSD e DTSVLIW com todas as configurações de tamanho de bloco avaliadas, exceto para blocos com 8 instruções. A DTSVLIW superou a DTSD em todos os casos.

Já com o programa de teste *quick*, o único caso em que o processador Alpha21264 foi superado, e não por muito, foi aquele com a arquitetura DTSVLIW configurada com blocos de até 96 instruções. Nos experimentos com este programa de teste o desempenho DTSD ficou sempre abaixo do DTSVLIW.

4.3 Percentual dinâmico de instruções de cópia

Além do desempenho em termos de IPC, outro parâmetro importante da DTSD que foi analisado é a quantidade de instruções do programa escalonadas para execução versos a quantidade de instruções de cópia de registradores (RR e WR) geradas por bloco. Em nossos experimentos, nosso simulador contou o número de blocos de instruções gerados e o número de instruções de cópia incluídas nestes blocos. Utilizamos estes dados para construir os gráficos da Figura 4.4.

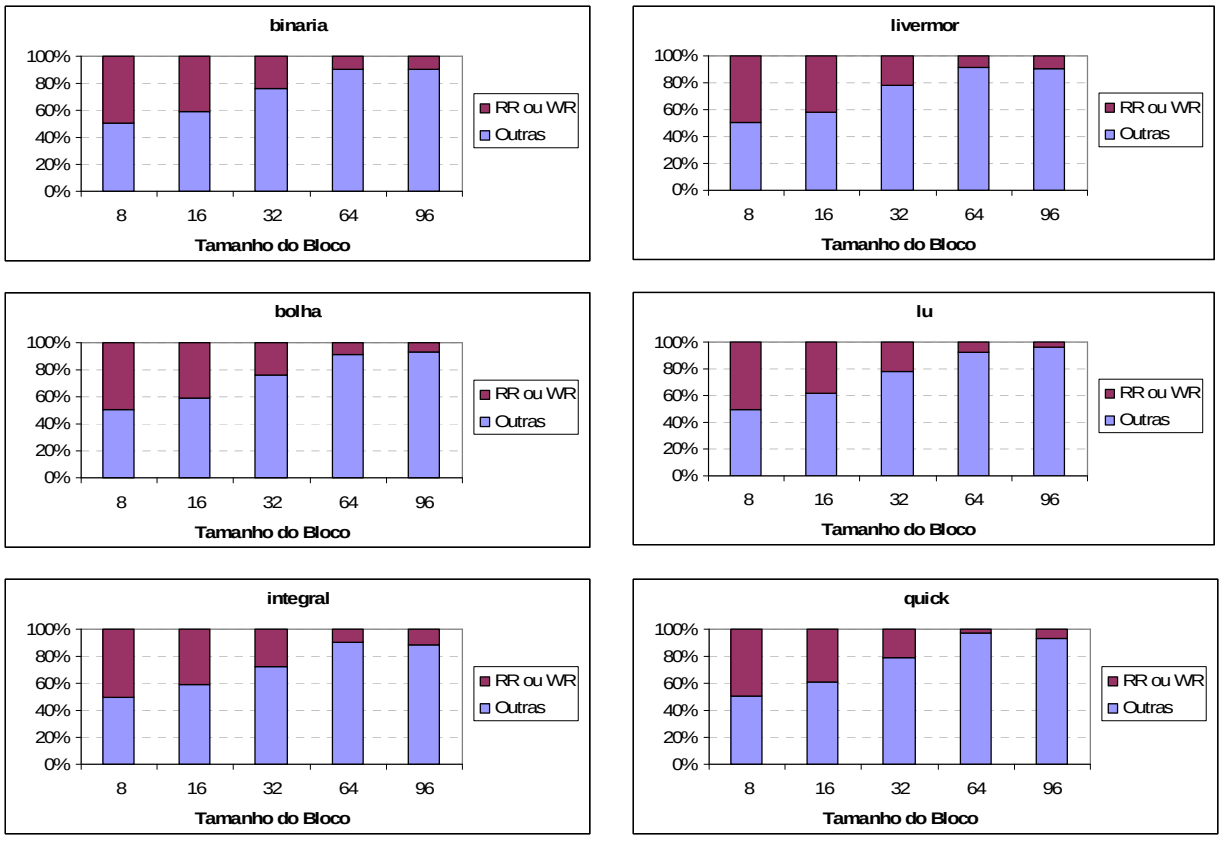


Figura 4.4: Distribuição das instruções nos blocos DTSD escalonados

Como os gráficos da Figura 4.4 mostram, quanto maior o bloco, menor o número percentual de instruções de cópia necessárias, sendo que, a partir de blocos com 64 instruções, o percentual de instruções de cópia nivela em torno de 10%. Para blocos muito pequenos, com 8 instruções, as instruções de cópia ocupam até 50% do bloco, o que impacta diretamente no desempenho. É importante observar que todos os acessos ao banco de registradores da ISA são feitos com

instruções de cópia. Assim, em uma máquina DTSD com blocos de 64 instruções ou mais, o número de acessos ao banco de registradores da ISA deve cair para cerca de 10% do número de acessos observado em uma máquina Super Escalar ou DTSVLIW. Essa redução certamente impacta positivamente no consumo de energia DTSD [Pedroni04] e mostra, também, a viabilidade, do ponto de vista do acesso ao banco de registradores da ISA, de máquinas DTSD com vários blocos de unidades funcionais (BUF).

5 Discussão

Os resultados experimentais obtidos com a execução dos programas de teste na arquitetura DTSD implementada com as características propostas demonstram que a arquitetura pode superar a DTSVLIW e a Alpha21264 em alguns dos casos estudados, contudo pode também apresentar desempenho significativamente inferior em muitos outros. Acreditamos, no entanto, que a DTSD pode superar as arquiteturas DTSVLIW e Super Escalar na maioria, senão na totalidade dos casos. Isso porque os estudos sobre a implementação desta arquitetura ainda estão na sua infância. Várias descobertas sobre a melhor forma de se implementar uma arquitetura que escalona dinamicamente por hardware foram feitas nos já quase 10 anos de estudos sobre a DTSVLIW, mas estas descobertas ainda não foram incorporadas à DTSD. Para citar um exemplo, foram observados ganhos de aproximadamente 20% no desempenho DTSVLIW (em termos de IPC) apenas evitando fazer a busca de blocos enquanto o bloco sendo escalonado não contenha certo número mínimo de instruções [DeSouza00] (foi empregado, para a DTSVLIW, um mínimo de 3/4 da capacidade do bloco nos experimentos relatados aqui). Isso evita que sejam salvos na Cache VLIW blocos com poucas instruções, logo com pouco paralelismo. A facilidade de evitar a busca de blocos enquanto o bloco sendo escalonado não contenha certo número mínimo de instruções ainda não foi implementada no nosso simulador DTSD. Muitas outras descobertas como esta podem ser incorporadas ao nosso simulador DTSD e, acreditamos, propiciarão substanciais ganhos de desempenho.

5.1 Trabalhos Correlatos

Máquinas Dataflow experimentais já existem há quase 30 anos [Davis97], mas ainda não há consenso sobre se o modelo de execução governado pelo fluxo de dados, no nível da arquitetura do conjunto de instruções, é uma maneira viável de se explorar paralelismo ou não. Vale destacar que máquinas *Dataflow* não podem ser utilizadas para executar código seqüencial diretamente. Por essa razão, elas não oferecem compatibilidade de código com versões anteriores, tal como a DTSVLIW e a DTSD oferecem.

Programas para ISAs *dataflow* indicam explicitamente as dependências entre as instruções [Veen86, Rau93a]. Isto é tipicamente implementado incluindo-se em cada instrução uma lista de

instruções sucessoras. Uma instrução é sucessora de outra se utiliza como operando de entrada o resultado produzido por esta outra instrução. Tão logo os operandos de entrada de uma instrução dentro da máquina *dataflow* estejam prontos, seu hardware pode executá-la e atualizar as tabelas que irão disparar a busca ou execução de novas instruções. Estudos anteriores sobre projetos de máquinas *dataflow* demonstraram ineficiências deste tipo de arquitetura [Veen86] – comparada às suas concorrentes controladas pelo fluxo de controle (*control-flow*), a abordagem do modelo *dataflow* de exploração do paralelismo no nível de instrução incorre em maiores custos de implementação dos circuitos de controle. Os custos envolvidos na detecção de instruções prontas para execução, e na utilização de seus resultados para habilitar outras instruções, geralmente resultavam em um baixo desempenho quando executando aplicações com baixo nível de paralelismo, tais como programas seqüenciais [Lee_B94].

Em uma máquina *dataflow*, instruções estão prontas para execução quando seus operandos de entrada estão disponíveis. Por isso, a máquina tem de manter várias (de centenas a milhares) instruções ainda não prontas em tabelas dentro do processador para montar uma janela de instruções grande o suficiente que possibilite encontrar paralelismo no nível de instrução em quantidade significativa. Detectar instruções prontas simultaneamente nesta grande janela de instruções é uma tarefa muito custosa de se implementar em *hardware*. Outra operação de elevado custo de implementação em *hardware* é o mapeamento dos resultados produzido por instruções executadas para suas instruções sucessoras.

É interessante ressaltar que os núcleos de máquinas super escalares funcionam de modo *dataflow*: os resultados produzidos pelas unidades funcionais são propagados de volta para a janela de instruções ou estações de reserva e disparam a execução de outras instruções. Os custos de gerenciamento desse modo de execução restrito ao núcleo são equivalentes aos custos de uma máquina *dataflow* genérica.

ISAs *Explicit Data Graph Execution* (EDGE) [Burger04] vem sendo estudadas há alguns anos. Em uma ISA EDGE os programas são grafos e as instruções são os nós do grafo e podem ser executadas tão logo seus operandos de entrada estejam prontos. Mas eles não são especificados explicitamente. Na DTSD os operandos são explicitados e tokens são usados para ativar a execução de instruções. Isso evita que operandos inteiros (tipicamente 64 bits) tenham que ser passados diretamente entre instruções, diminuindo o número de barramentos internos da máquina, o que tem impacto positivo no consumo de energia e simplifica a implementação.

Na arquitetura EDGE proposta por Burger [Burger04] o escalonamento é estático e feito pelo compilador. Experimentos com a DTSVLIW mostraram que o escalonamento dinâmico DTSVLIW supera o escalonamento estático EPIC [Santana03], e acreditamos que o escalonamento dinâmico

DTSD também deve superar o escalonamento estático EDGE. O escalonamento estático também trás restrições à compatibilidade de código para trás, problema que a arquitetura DTSD não possui.

5.2 Análise Crítica deste Trabalho de Pesquisa

A arquitetura implementada e testada, ainda não possui todas as características propostas. Uma característica importante é a de poder ser implementada com mais de um Bloco de Unidades Funcionais (BUF). Contudo, a implementação de mais de um BUF leva a questões mais complexas não avaliadas neste trabalho, como a da viabilidade de implementação de BUFs grandes versus o custo de implementação em hardware do sistema de envio de *tokens* entre BUFs e seu impacto no desempenho.

Uma questão não tão complexa, que é uma característica da DTSVLIW e não foi implementada na DTSD, é a condição de só tentar fazer a busca de um bloco pronto a partir do momento em que o bloco sendo escalonado no momento tenha um certo tamanho (como mencionado acima).

Neste trabalho, gostaríamos de ter utilizado também os programas de avaliação do conjunto SPEC (www.specbench.org), mais amplamente utilizados na comunidade científica para a avaliação de arquiteturas. Contudo, problemas na implementação do simulador DTSD não permitiram executar os programas do SPEC2000 até o final de cada programa (seqüências incomuns de instruções ainda não são tratadas corretamente pelo nosso simulador). Ainda assim, tais problemas não invalidam os resultados obtidos com os programas de teste utilizados, pois como nosso simulador é *execution-driven* e executa em paralelo o simplescalar, que é um simulador já validado [Austin97], é garantido que, a cada bloco executado, o estado da máquina DTSD seja comparado ao da máquina seqüencial de testes para validação.

Outro aspecto importante são as instruções de cópias de registradores escalonadas. A idéia de utilizá-las foi motivada pelo alto custo e complexidade de acesso ao banco de registradores global por todos os BUFs de uma arquitetura DTSD implementada com mais de um deles, que era o nosso objetivo quando iniciado este trabalho. Contudo, como já tivemos resultados em que o desempenho DTSD se mostrou competitivo, estes nos levam a imaginar se não obteríamos resultados ainda melhores caso não utilizássemos as instruções de cópia. Neste caso, as instruções do programa ainda não renomeadas leriam e escreveriam seus resultados diretamente do banco de registradores. Um bloco sem as instruções de cópia de registradores seria inteiramente composto por instruções do programa, o que proveria um maior aproveitamento de paralelismo, principalmente para blocos pequenos, onde vimos que até 50% deles (blocos com 8 instruções) pode ser ocupados por instruções de cópia de registradores. Muitos dispositivos como celulares e computadores de mão

utilizam microprocessadores e máquinas DTSD com blocos pequenos podem ser uma alternativa para estes dispositivos.

6 Conclusão

Neste capítulo são apresentados um sumário do trabalho, as conclusões, e propostas de trabalhos futuros.

6.1 Sumário

Neste trabalho foi apresentado um estudo de uma arquitetura que escalona dinamicamente, por hardware, caminhos de execução em blocos de código *dataflow*. Para o estudo realizado, limitamos a nossa análise a um caso particular onde a arquitetura proposta possuía apenas um bloco de unidades funcionais. Com esta característica, experimentamos diferentes tamanhos de bloco de instruções e observamos o desempenho da arquitetura proposta quando executando programas de teste simples. Para avaliar seu desempenho, este foi comparado às arquiteturas DTSVLIW e Super Escalar empregando parâmetros arquiteturais similares.

6.2 Conclusões

Os resultados obtidos da execução dos programas de teste com a arquitetura DTSD implementada com as características propostas demonstraram que a arquitetura pode superar a DTSVLIW e a Super Escalar em alguns casos, mas pode também apresentar desempenho inferior em muitos outros. Isso se deve principalmente à não incorporação de descobertas feitas, quando dos estudos sobre a arquitetura DTSVLIW, sobre como implementar uma máquina que escalona código dinamicamente por hardware. Também foi possível constatar que blocos muito pequenos desperdiçam possíveis oportunidades de paralelismo devido ao grande número de instruções de cópia de registradores (resultado do método de tradução para código *dataflow*) salvas nos blocos escalonados em relação ao número de instruções escalonadas.

6.3 Trabalhos Futuros

Como proposta de trabalhos futuros, será importante avaliar a implementação de uma máquina DTSD com mais de um bloco de unidades funcionais (BUF) e avaliar as questões pertinentes relativas a desempenho, viabilidade e custo/benefício de sua construção. Outro aspecto importante a ser avaliado futuramente é como a DTSD se comporta quando impomos um número mínimo de

instruções em cada bloco. Esta característica pode ser implementada fazendo com que a DTSD só procure por um bloco já pronto quando o que estiver sendo escalonado no momento tiver um certo número de instruções.

Para uma avaliação mais detalhada da arquitetura, é necessário desenvolver o simulador DTSD ainda mais, de modo que este possa executar os programas do pacote SPEC 2000 por completo. Seguindo por um outro caminho, será avaliada uma DTSD para blocos pequenos onde não seja necessário o escalonamento de instruções de leitura e escrita de registradores, na tentativa de se obter bom desempenho visando microprocessadores de pequeno porte.

Por fim, é importante examinar todos os aspectos relevantes para o aumento de desempenho DTSD associados a máquinas que escalonam dinamicamente por *hardware*.

7 Bibliografia

[Almeida03] F. L. Almeida, A. F. De Souza, “O Efeito da Latência no Desempenho da Arquitetura DTSVLIW”, IV Workshop em Sistemas Computacionais de Alto Desempenho, São Paulo – SP, p. 64-71, 2003.

[Almeida04] F. L. Almeida, A. F. De Souza, “Uma Arquitetura DTSVLIW com Múltiplos Contextos de Execução”, V Workshop em Sistemas Computacionais de Alto Desempenho – WSCAD'2004, Foz do Iguaçu – RS, 2004.

[Austin97] T. Austin, D. Burger, “The SimpleScalar Tool Set”, Technical Report TR-1342, Computer Science Department, University of Wisconsin – Madison, June 1997.

[Burger04] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, “Scaling to the End of Silicon with EDGE Architectures”, IEEE Computer, Vol. 37, No. 7, p. 44-55, July 2004.

[Compaq99] Compaq Computer Corporation, “Alpha 21264 Microprocessor Hardware Reference Manual”, Compaq Computer Corporation, 1999.

[Conte65] S. D. Conte, “Elementary Numerical Analysis”, McGraw-Hill Book Co., New York, NY, USA, 1965.

[Davis79] A. L. Davis, “A Data Flow Evaluation System Based on the Concept of Recursive Locality”, Proceedings of the National Computing Conference, AFIPS Press, Reston, Va., p. 1079-1086, 1979.

[Desikan01] R. Desikan, D. Burger, S. W. Keckler, “Measuring Experimental Error in Microprocessor Simulation”, Proceedings of the 28th Annual International Symposium on Computer Architecture, p. 226-277, 2001.

[DeSouza98] A. F. de Souza, P. Rounce, “Dynamically Trace Scheduled VLIW Architectures”, Lecture Notes in Computer Science, Vol. 1401, p. 993-995, 1998.

[DeSouza00] A. F. De Souza, P. Rounce, “Dynamically Scheduling VLIW Instructions”, Journal of Parallel and Distributed Computing, Vol. 60, No. 12, p. 1480-1511, December 2000.

[DeSouza01] A. F. de Souza, P. Rounce, “Improving the DTSVLIW Performance via Block Compaction”, Proceedings of the 13th Symp. on Computer Architecture and High Performance Computing – SBAC-PAD’2001, p. 98-105, 2001.

[Fisher84] J. Fisher, “The VLIW Machine: A Multiprocessor for Compiling Scientific Code”, IEEE Computer, p. 45-53, July 1984.

[Forsythe67] G. Forsythe, C. B. Moler, “Computer Solution of Linear Algebraic Systems”, Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[Freitas03] C. D. D. Freitas, A. F. De Souza, “Single Instruction Fetch Does Not Inhibit Instruction-Level Parallelism”, In: Workshop on Exploring the Trace Space for Dynamic Optimization Techniques - in conjunction with the International Conference on Supercomputing, SC2003, San Francisco - California, p. 13 – 21, 2003.

[Horowitz78] E. Horowitz, S. Sahni, “Fundamentals of Computer Algorithms”, Computer Science Press Inc., Maryland, USA, 1978.

[Hwu87] W. W. Hwu, Y. N. Patt, “Checkpoint Repair for Out-of-order Execution Machines,” Proceedings of the 14th Annual International Symposium on Computer Architecture, pp. 18-26, 1987.

[Jacob95] B. Jacob, T. Mudge, “Notes on Calculating Computer Performance”, Technical Report CSE-TR-231-95, Department of Electrical Engineering and Computer Science, University of Michigan, USA, March 1995.

[Johnson91] M. Johnson, “Superscalar Microprocessor Design”, Prentice-Hall, 1991.

[Knuth73] D. Knuth, “Art of Computer Programming”, Addison=Wesley Publishing Co., USA, 1973.

[Komati03] K. S. Komati, A. F. De Souza, “Using Weightless Neural Networks for Vergence Control in an Artificial Vision System”, Journal Of Applied Bionics And Biomechanics, Auckland, New Zealand, v. 1, n. 1, p. 21-32, 2003.

[Lee_B94] B. Lee, A. R. Hurson, “Dataflow Architectures and Multithreading”, IEEE Computer, p. 27-39, 1994.

[McMahon83] F. H. McMahon, “Fortran Kernels: MFLOPS”, Lawrence Livermore National Laboratory, 1983.

[Pedroni04] F. T. Pedroni, F. L. L. Almeida, A. F. De Souza, “Implementação de uma Versão Power Aware do Simulador DTSVLIW”, aceito para publicação nos anais do V Workshop em Sistemas Computacionais de Alto Desempenho - WSCAD'2004, Foz do Iguaçu – RS, 2004.

[Rau93a] B. R. Rau, J. A. Fisher, “Instruction-Level Parallelism: History, Overview and Perspective”, The Journal of Supercomputing, Vol. 7, p. 9-50, 1993.

[Rau93b] B. R. Rau, “Dynamically Scheduled VLIW Processors”, Proceedings of the 26th Annual International Symposium on Microarchitetcture, p. 80-92, 1993.

[Rotenberg96] E. Rotenberg, S. Bennett, J. E. Smith, “Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching”, Proceedings of the 29th Annual International Symposium on Microarchitecture, p. 24-34, 1996.

[Rounce06] P. Rounce, A. F. De Souza, “The DTSVLIW: A Multi-Threaded Trace-Based VLIW Architecture”, Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing, p. 63-70, 2006.

[Rounce07] P. Rounce, A. F. De Souza, “Dynamic Instruction Scheduling in a Trace-Based Multi-Threaded Architecture”, a ser publicado no International Journal of Parallel Programming, 2007.

[Santana03] S. C. Santana A. F. De Souza, “A Comparative Analysis Between EPIC Static Instruction Scheduling and DTSVLIW Dynamic Instruction Scheduling”, In: Workshop on Exploring the Trace Space for Dynamic Optimization Techniques - in conjunction with the International Conference on Supercomputing, SC2003, San Francisco - California, p. 59 – 67, 2003.

[Thekkath94] R. Thekkath, S. J. Eggers, “The Effectiveness of Multiple Hardware Contexts”, In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, p. 328-337. ACM Press, October 1994.

[Tullsen95] D. M. Tullsen, S. J. Eggers, H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism”, Proceedings of the 22nd Annual International Symposium on Computer Architecture, p. 392-403, June 22-24, 1995.

[Veen86] A. H. Veen, “Dataflow Machine Architecture”, ACM Computing Surveys, Vol. 18, No. 4, pp. 335-396, December 1986.

[Wall93] D. W. Wall, “Limits of Instruction-Level Parallelism”, Digital Western Research Laboratory – WRL Research Report 93/6, November 1993.

Apêndice A

Código em C do programa binaria:

```
#define N    600
#define true      1
#define false     0
int    m,k,x[N+1] = {-1
 ,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
 , 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40
 , 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60
 , 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80
 , 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,100
 ,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120
 ,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140
 ,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160
 ,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180
 ,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200
 ,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220
 ,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240
 ,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260
 ,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280
 ,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300
 ,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320
 ,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340
 ,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360
 ,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380
 ,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400
 ,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420
 ,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440
 ,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460
 ,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480
 ,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500
 ,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520
 ,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540
 ,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560
 ,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580
 ,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600};

int dado,inicio,meio,fim;

main()
{
  dado = 99;
  inicio = 1;
  fim = N;
  meio = (1 + N)/2;
  while ((dado != x[meio]) && (inicio != fim))
  {
    if (dado > x[meio])
       inicio = meio + 1;
    else
       fim = meio - 1;
    meio = (inicio + fim) / 2;
  }
  if (dado == x[meio])
    printf ("meio = %d\n", meio);
  else
    printf ("meio = %d\n", -1);
}
```

Código em C do programa bolha:

```
#define N    600
#define true      1
#define false     0
int       x[N] = {
 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
 , 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40
 , 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60
 , 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80
 , 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,100
 ,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120
 ,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140
 ,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160
 ,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180
 ,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200
 ,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220
 ,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240
 ,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260
 ,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280
 ,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300
 ,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320
 ,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340
 ,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360
 ,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380
 ,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400
 ,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420
 ,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440
 ,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460
 ,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480
 ,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500
 ,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520
 ,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540
 ,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560
 ,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580
 ,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600};

int                 fim,ind,aux;
int    troquei;

main()
{
  fim = (N/10) - 1;
  do
  {
    troquei = false;
    for(ind = 0; ind <= (fim - 1); ind++)
             if (x[ind] < x[ind + 1])
    {
      aux    = x[ind];
      x[ind] = x[ind + 1];
      x[ind + 1] = aux;
      troquei = true;
    }
    fim = fim - 1;
  }
       while(troquei);
       printf ("x[9] = %d\n", x[9]);
}
```

Código em C do programa integral:

```
#define     LIM   10
#define INCR      0.001

float Y(t)
float t;
{
      return(t*t);
}

float t,tant,Area,BMaior,BMenor;

float
main()
{
      tant = 0;
      Area = 0.0;
      for(t = INCR; t <= LIM; t += INCR)
      {
            BMaior =  Y(t);
            BMenor =  Y(tant);
            Area   += (BMaior + BMenor) * INCR / 2;
            tant   =  t;
      }
      printf ("Area = %f\n", Area);
}
```

Código em C do programa livermor:

```
#define N    600
int    m,k,x[N] = {
 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
 , 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40
 , 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60
 , 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80
 , 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,100
 ,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120
 ,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140
 ,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160
 ,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180
 ,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200
 ,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220
 ,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240
 ,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260
 ,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280
 ,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300
 ,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320
 ,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340
 ,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360
 ,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380
 ,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400
 ,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420
 ,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440
 ,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460
 ,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480
 ,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500
 ,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520
 ,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540
 ,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560
 ,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580
 ,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600};

main()
{
      m = 0;
      for (k = 1; k < N; k++)
            if (x[k] > x[m])
                  m = k;
      printf ("m = %d\n", m);
}
```

Código em C do programa lu:

```
#define                N        24

int        i,j,k,imax;
float  A[24][24] = {
{4.0,1.0,3.0,3.0,2.0,6.0,9.0,3.0,5.0,8.0,5.0,3.0,3.0,4.0,7.0,4.0,4.0,2.0,7.0,4.0,7.0,3.0,1.0,3.0},
{4.0,5.0,4.0,3.0,2.0,1.5,2.0,9.0,2.3,6.7,4.0,6.0,8.0,2.0,8.0,9.0,7.0,5.0,5.0,6.0,2.0,5.0,3.3,9.0},
{5.0,3.0,2.0,6.0,7.0,8.0,4.0,5.6,8.2,7.7,9.0,5.0,4.0,3.0,1.0,7.0,5.0,1.5,2.0,1.0,9.0,7.0,2.8,8.0},
{5.0,3.0,2.0,6.0,9.0,8.0,4.0,5.6,8.2,2.0,5.0,2.0,3.0,4.0,7.0,4.0,4.0,2.0,7.0,4.0,7.0,3.0,1.0,3.0},
{7.0,5.0,1.0,2.0,3.0,1.5,2.0,4.0,2.3,6.7,4.0,6.0,2.0,3.0,8.0,9.0,2.0,5.0,5.0,6.0,2.0,5.0,3.3,9.0},

{4.0,5.0,9.0,3.0,2.0,1.5,2.0,9.0,2.3,6.0,6.0,1.0,4.0,5.0,4.0,3.0,2.0,1.5,2.0,9.0,2.3,6.7,4.0,6.0},
{5.0,3.0,2.0,6.0,9.0,8.0,4.0,5.6,8.2,2.0,5.0,2.0,4.0,5.0,4.0,3.0,2.0,1.5,2.0,9.0,2.3,6.7,4.0,6.0},
{3.0,2.1,8.0,3.0,3.5,2.6,9.1,5.6,5.2,2.0,5.0,6.0,4.0,5.1,9.0,3.0,2.0,1.5,2.0,9.0,2.3,6.0,6.0,1.0},
{5.0,2.5,5.0,1.0,7.0,3.2,4.0,5.6,8.2,7.7,9.0,5.0,6.0,7.7,2.0,5.0,7.0,5.6,4.0,8.6,5.2,2.0,7.0,6.0},
{9.0,5.5,4.0,6.0,7.0,1.1,4.0,2.7,4.5,5.5,2.0,4.3,3.0,7.0,5.0,9.0,2.0,6.5,7.0,4.0,9.3,3.0,2.0,4.0},
{4.0,1.0,3.0,1.9,7.3,9.4,9.0,3.0,5.0,8.0,5.0,3.0,3.6,9.0,8.0,3.0,2.0,6.6,9.0,5.6,5.2,2.0,7.0,6.0},
{2.0,3.0,1.0,7.0,5.0,6.5,8.0,9.0,2.3,2.7,4.0,6.0,8.0,2.0,8.0,4.0,7.0,5.0,5.0,6.0,2.0,5.0,3.3,9.0},
{5.0,1.0,2.0,6.0,7.0,9.0,4.0,5.6,8.2,7.7,9.0,5.0,4.0,3.0,1.0,2.0,5.0,1.5,2.0,1.0,9.0,7.0,2.8,8.0},
{5.0,3.0,2.0,6.0,9.0,3.0,4.0,5.6,8.2,2.0,5.0,2.0,3.0,4.0,7.0,1.0,4.0,2.0,7.0,4.0,7.0,3.0,1.0,3.0},
{7.0,5.0,1.0,2.0,3.0,6.5,2.0,4.0,2.3,6.7,4.0,6.0,2.0,3.0,8.0,2.0,2.0,5.0,5.0,6.0,2.0,5.0,3.3,9.0},

{4.0,7.0,9.0,3.0,2.0,8.5,2.0,9.0,2.3,6.0,6.0,1.0,4.0,5.0,4.0,7.0,2.0,1.5,2.0,9.0,2.3,6.7,4.0,6.0},
{5.0,3.0,2.0,6.0,9.0,3.0,4.0,5.6,8.2,2.0,5.0,2.0,4.0,5.0,4.0,2.0,2.0,1.5,2.0,9.0,2.3,6.7,4.0,6.0},
{3.0,2.1,8.0,3.0,3.5,1.6,9.1,5.6,5.2,2.0,5.0,6.0,4.0,5.1,9.0,9.0,2.0,1.5,2.0,9.0,2.3,6.0,6.0,1.0},
{5.0,2.5,5.0,1.0,7.0,9.2,4.0,5.6,8.2,7.7,9.0,5.0,6.0,7.7,2.0,3.0,7.0,5.6,4.0,8.6,5.2,2.0,7.0,6.0},
{9.0,5.5,4.0,6.0,7.0,2.1,4.0,2.7,4.5,5.5,2.0,4.3,3.0,7.0,5.0,4.0,2.0,6.5,7.0,4.0,9.3,3.0,2.0,4.0},
{4.0,1.0,3.0,1.9,7.3,5.4,9.0,3.0,5.0,8.0,5.0,3.0,3.6,9.0,8.0,3.0,2.0,6.6,9.0,5.6,5.2,2.0,7.0,6.0},
{1.0,5.5,4.0,4.0,7.0,1.1,5.0,2.7,4.5,8.5,2.0,4.3,3.0,7.0,5.0,9.0,2.0,6.5,7.0,2.0,9.3,3.0,2.0,4.0},
{3.0,3.0,2.0,7.0,9.0,3.0,1.0,5.6,8.2,7.0,5.0,2.0,3.0,4.0,7.0,1.0,4.0,2.0,7.0,1.0,7.0,3.0,1.0,3.0},
{2.4,3.0,2.0,3.7,7.0,2.0,9.0,5.6,8.2,1.7,9.0,3.2,5.0,1.7,2.0,2.0,7.0,8.0,4.0,9.6,8.2,7.7,9.0,5.0}},
        aux,pivot;


float
main()
{
        for (i = 0; i <= (N-2); i++)
        {
                pivot = 0;
                imax  = i;
                for (j = i; j <= (N-1); j++)
                {
                        aux  = (A[j][i] < 0) ? -A[j][i] : A[j][i];
                        if (aux > pivot)
                        {
                                pivot = aux;
                                imax  = j;
                        }
                }
                if (imax != i)
                        for (j = i; j <= (N-1); j++)
                        {
                                aux = A[i][j];
                                A[i][j] = A[imax][j];
                                A[imax][j] = aux;
                        }
                for (j = i+1; j <= (N-1); j++)
                        A[j][i] /= A[i][i];
                for (k = i+1; k <= (N-1); k++)
                        for (j = i+1; j <= (N-1); j++)
                                A[k][j] -= (A[k][i]*A[i][j]);
        }
       printf ("A[5][10] = %f\n", A[5][10]);
}
```

Código em C do programa quick:

```
#define N     600
#define true      1
#define false     0
int    m,k,x[N+1] = {-1
 ,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580
 ,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560
 ,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540
 ,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520
 ,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500
 ,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480
 ,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460
 ,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440
 ,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420
 ,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400
 ,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380
 ,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360
 ,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340
 ,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320
 ,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300
 ,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280
 ,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260
 ,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240
 ,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220
 ,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200
 ,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180
 ,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160
 ,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140
 ,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120
 , 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80
 , 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,100
 , 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60
 , 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40
 ,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
 ,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600};

int                 fim,aux;
unsigned char       troquei;

void
quick(int inicio, int fim)
{
  int antes,depois,meio,aux;

  antes = inicio;
  depois = fim;
  meio = x[(inicio + fim)/2];
  do
  {
    while(x[antes] < meio)
      antes++;
    while(meio < x[depois])
                depois--;
    if (antes <= depois)
    {
      aux = x[antes];
                x[antes++] = x[depois];
                x[depois--] = aux;
    }
  } while(depois >= antes);
  if (inicio < depois)
    quick(inicio,depois);
  if (antes < fim)
    quick(antes,fim);
}

main()
{
  quick(1,N);
       printf ("x[N] = %d\n", x[N]);
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo