Algoritmos de Despacho de Instruções em Máquinas Super Escalares

Luiz Márcio Faria de Aquino Viana

Projeto de graduação submetido ao corpo docente do Departamento de Engenharia de Sistemas da Universidade do Estado do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Engenheiro de Sistemas e Computação.

Aprovada por:

Rio de Janeiro, RJ - Brasil
Julho de 1997

Conteúdo

Capítulo 1

Introdução

Os recentes avanços na confecção de integrados não foram suficientes para vencer a necessidade crescente de computadores mais velozes para processamento de *softwares* cada vez mais complexos. Desta forma os projetistas de computadores repensaram uma nova estratégia para o projeto de processadores.

Percebendo que o tradicional modelo de processamento seqüencial já está no limite, e que a única solução para aumentar de forma significativa o desempenho dos processadores é através do processamento paralelo das instruções vindas de um programa, fez-se surgir as primeiras máquinas super escalares.

No meio da computação encontramos processadores denominados escalares por executarem uma instrução por vez seqüencialmente. Por exemplo, os processadores da linha Intel 80x86. Há também os processadores vetoriais capazes de ativar em paralelo múltiplos elementos de processamento, em que todos executam a mesma instrução.

Os processadores super escalares se caracterizam por serem máquinas capazes de executar instruções diferentes simultaneamente utilizando apenas um único contador de programa para esta tarefa. As máquinas super escalares são uma combinação entre maquinas escalares e vetoriais capazes de executar diferentes operações simultaneamente.

Existem duas estratégias para o aproveitamento do paralelismo nas maquinas super escalares: através do escalonamento estático de instruções (realizado a nível de *software* com a otimização por parte do compilador do código executável do programa) ou através do escalonamento dinâmico (a nível de *hardware* realizado com o programa em execução). Atualmente estas técnicas andam praticamente juntas e os processadores que fazem o escalonamento dinâmico das instruções alcançam um desempenho significativamente maior se o código for devidamente otimizado.

1.1. MOTIVAÇÕES E OBJETIVO

O objetivo deste projeto é realizar um estudo dos algoritmos de despacho empregados em máquinas super escalares e analisar o efeito deles sobre o desempenho do processador.

Para analisar o efeito do algoritmo de despacho, foi desenvolvido um simulador de uma máquina super escalar baseado no modelo de arquitetura *load/store* **DLX** (vide [Patter85]), e denominado **Dlxwin**. O ambiente do simulador serviu de base para a implementação de quatro mecanismos de despacho, sendo três deles empregados em processadores da atualidade: Alpha AXP 21064, PowerPC 603 e Pentium. Também foram desenvolvidos dez programas de testes que avaliaram o desempenho de cada algoritmo.

Os resultados obtidos nos experimentos mostraram que a implementação adequada do algoritmo de despacho em uma arquitetura super escalar pode representar um ganho de até 75% no desempenho do processador.

No capítulo seguinte analisaremos alguns conceitos iniciais sobre paralelismo de baixo nível e máquinas super escalares, onde discutiremos o uso de múltiplas unidades funcionais independentes (chave do processamento paralelo), os problemas decorrentes das diversas formas de dependências (fator responsável pela limitação de desempenho em uma máquina paralela), também veremos alguns dos mecanismos básicos utilizados neste tipo de arquitetura e apresentaremos dois algoritmos pioneiros na tecnologia super escalar.

No capítulo 3 serão apresentados os principais recursos de exploração do paralelismo, utilizados em máquinas super escalares reais e que foram implementados no simulador **Dlxwin**.

Posteriormente no capítulo 4 será apresentado as características dos três processadores que tiveram seus algoritmos de despacho implementados, **Pentium**, **PowerPC 603** e **Alpha AXP 21064**, como exemplos de arquiteturas super escalares.

O capítulo 5 descreve os aspectos gerais da arquitetura **DLX** e do simulador desenvolvido **Dlxwin,** apresentando as diferenças estruturais entre as duas e os recursos implementados na ferramenta desenvolvida.

A analise dos resultados será apresentada no capítulo 6, onde mostraremos as características do ambiente de experimento produzido, e avaliaremos os resultados do processamento de cada programa de teste para os quatro algoritmos de despacho implementados.

O último capítulo traz as conclusões obtidas sobre os experimentos realizados e uma avaliação dos algoritmos de despacho implementados.

Capítulo 2

Conceitos Iniciais

2.1 MÁQUINAS COM MÚLTIPLAS UNIDADES FUNCIONAIS

Existem duas estratégias para projeto de um processador: (i) incluir em um único dispositivo funcional todas as funções do processador ou (ii) separar as funções realizadas pelo processador por vários dispositivos funcionais independentes.

A primeira estratégia era até então largamente utilizada. Fatores econômicos contribuíram para a utilização desta estratégia durante muito tempo.

Atualmente não se pode pensar em processadores de alto desempenho sem pensar em máquinas com múltiplos dispositivos funcionais independentes. Entre os fatores que levaram ao crescimento desta estratégia de projeto estão a queda no custo de produção de integrados e a oportunidade que esta estratégia oferece para aumentar o desempenho dos processadores.

O principal motivo que viabilizou o desenvolvimento de máquinas super escalares foi o conceito de separação e especialização, e é este conceito que faz a diferença entre o desempenho destas máquinas e o das máquinas escalares.

O conceito de separação e especialização traz grandes vantagens, pois unidades funcionais especializadas necessitam de uma seqüência de controle menos complexa e mais curta reduzindo desta forma o tempo de processamento da máquina.

Além das vantagens obtidas pela técnica de separação e especialização, as máquinas com múltiplas unidades funcionais podem fugir do tradicional estilo de processamento seqüencial e partir para um modo de processamento em que duas ou mais unidades funcionais possam executar instruções em paralelo.

Para obter as vantagens da execução em paralelo é importante que as instruções em execução simultânea sejam independentes uma das outras. O maior limitador no desempenho de processadores super escalares ou de qualquer outro que faça uso do processamento paralelo, é a dependência entre instruções conforme veremos a seguir.

2.2. LIMITAÇÕES AO PARALELISMO

A dependência provoca a quebra do paralelismo forçando o sequenciamento das instruções. Este sequenciamento prejudica o desempenho do processador e é extremamente indesejável, embora muitas vezes inevitável. Nas subseções seguintes examinaremos os três tipos de dependência que provocam a quebra do paralelismo e forçam o sequenciamento das instruções: dependência de dados, dependência de controle e dependência funcional.

2.2.1. DEPENDÊNCIA DE DADOS

A dependência de dados resulta do fluxo de dados durante a execução de um programa, isto é, de como os registradores são utilizados. Observe a seqüência de instruções na **figura 2.1**:

```
        :
1.   R1 : = R2 +  R3
2.   R1 : = R4
        :
```

Figura 2.1: *Trecho de código de um programa.*

A primeira instrução altera o conteúdo de **R1** com o valor da soma **R2+R3** e a segunda instrução altera **R1** com o valor de **R4**.

Se as duas instruções fossem despachadas em paralelo a execução da primeira poderia ser finalizada depois da segunda e assim o conteúdo final de **R1** seria **R2+R3** o que não reflete a semântica do código apresentado. Desta forma sempre que ocorrer uma dependência de dados a ordem de precedência das operações precisará ser respeitada para evitarmos a perda do contexto do programa.

As relações de dependência de dados podem se apresentar de três formas denominadas dependência verdadeira, anti-dependência ou dependência de saída.

DEPENDÊNCIA VERDADEIRA

Este tipo de dependência ocorre quando duas instruções em uma seqüência de código são tais que a primeira possui uma variável recebendo uma atribuição e a segunda possui esta variável participando da sua expressão. A primeira e a segunda instruções no exemplo da **figura 2.2** possuem dependência verdadeira provocada pelo registrador **R1**.

```
                :
1.      R1 : = R2 + R3
2.      R4 : = R1 * R5
3.      R3 : = R5 * R5
4.      R4 : = K
                :
```

Figura 2.2: *Trecho de código de um programa.*

ANTI-DEPENDÊNCIA

A **anti-dependência** ocorre quando duas instruções em uma mesma seqüência de código são tais que a primeira possui uma variável como dado de entrada em sua expressão e a segunda atribui valores de uma expressão a esta variável. No exemplo da **figura 2.2** o registrador **R3** utilizado na primeira e na terceira instruções cria uma relação de anti-dependência entre elas.

DEPENDÊNCIA DE SAÍDA

Esta forma de dependência de dados ocorre quando duas instruções em uma mesma seqüência de código atribuem valores a uma mesma variável. No exemplo da **figura 2.2** a segunda e quarta instruções possuem dependências de saída devido ao registrador **R4** utilizado como registrador destino por ambas.

Existem vários algoritmos implementados por *software* e *hardware* capazes de eliminar as dependências de dados dos tipos anti-dependência e dependência de saída. O mesmo não ocorre com a dependência verdadeira, sendo esta a razão de sua denominação. As dependências do tipo anti-dependência e dependência de saída são referenciadas também como dependências falsas.

2.2.2. DEPENDÊNCIA DE CONTROLE

A dependência de controle resulta do fluxo de controle do programa em execução, isto é, dos comandos de desvio existentes no programa.

Há três formas das dependências de controle se apresentarem: através de desvios incondicionais, através de desvios condicionais ou por interrupção de *hardware*.

A dependência de controle gerada por desvios incondicionais oferece poucos problemas, porque é possível conhecer o endereço alvo da instrução de desvio antes da execução, basta o algoritmo de busca ser capaz de reconhecer as instruções de desvio e modificar o endereço de busca para o endereço alvo conhecido.

No caso de desvios condicionais existe um agravante, pois até que a avaliação da condição seja concluída, não há como saber se o desvio ocorrerá ou não. Uma solução bastante simples é interromper a busca de novas instruções até que a ação tomada pelo desvio seja determinada. Esta solução apresenta queda de desempenho significativa. Experimentos realizados mostraram que cerca de 15 a 30% das instruções executadas por um processador correspondem à comandos de desvio (vide [Edil92]), portanto interromper a busca antecipada das instruções sempre que um comando de desvio condicional for encontrado não é uma boa estratégia. No capítulo 3 analisaremos alguns algoritmos que atenuam este efeito.

Outra forma de dependência de controle é gerada por interrupções de *hardware*. As interrupções de *hardware* são imprevisíveis, desta forma não dispomos de meios para antecipar a busca das instruções sucessoras ao comando de desvio. Além disso precisamos tomar bastante cuidado para não misturar instruções provenientes da rotina de tratamento da interrupção com as instruções do programa.

2.2.3. DEPENDÊNCIA FUNCIONAL

Como o número de unidades funcionais de um mesmo tipo, dentro de um processador é limitado, pode ocorrer de instruções independentes não serem despachadas paralelamente, por falta de dispositivos funcionais em quantidades suficientes.

Por exemplo, suponha que uma máquina super escalar com duas unidades funcionais de adição de inteiros esteja executando um código de programa onde duas instruções envolvendo adição de inteiros tenham sido previamente despachadas e estão em execução. Portanto temos as duas unidades funcionais de adição de inteiros em uso. Suponha que antes de uma das duas instruções ter sido concluída uma nova instrução de adição de inteiros é encontrada pelo escalonador. Neste momento o despacho de instruções é bloqueado, motivado pela falta de unidades funcionais para executar a instrução, até que uma das duas instruções em execução seja finalizada e libere uma das unidades de adição de inteiros.

Verificou-se experimentalmente que a relação entre o número de unidades funcionais duplicadas e o desempenho do processador não é linear. Em uma bateria de testes realizados observou-se que durante 91,78% do tempo de execução de um programa não mais do que quatro unidades de operações com inteiros foram utilizadas de um total de seis existentes (vide [Edil92]). O histograma apresentado na **figura 2.3** mostra os resultados obtidos no experimento.

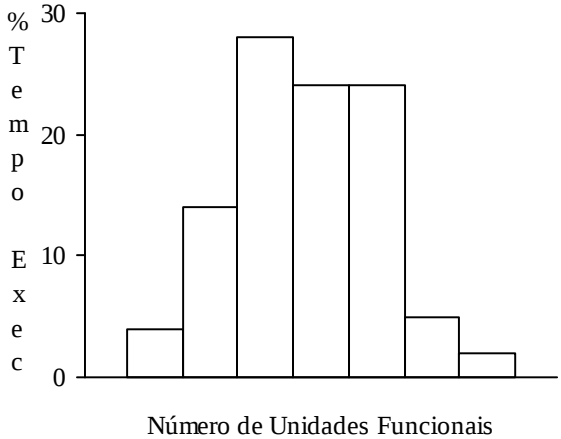


Figura 2.3: *Histograma de utilização concorrente da unidade de inteiros.*

Podemos observar que se esta máquina super escalar tivesse quatro unidades de inteiros ao invés de seis, teríamos um desempenho praticamente equivalente com um custo significativamente reduzido.

2.3. JANELA DE INSTRUÇÕES

A existência de máquinas com múltiplas unidades funcionais permite a execução de múltiplas instruções simultaneamente, de modo que o processador não precisa aguardar a conclusão de uma instrução para iniciar a execução da subsequente. Com isso, a unidade de controle precisa de um mecanismo que aumente a taxa de ocupação das unidades funcionais e o desempenho do processador.

Com a possibilidade de aumentar o número de instruções despachadas. A manutenção das instruções na memória principal para serem carregadas quando requeridas pela unidade de controle, pode resultar em uma perda significativa de tempo. Esta perda poderá ser minimizada se as instruções já estiverem no interior do processador. Algumas arquiteturas de computadores possuem um conjunto de registradores para armazenar as instruções. Viabilizando desta forma, o exame antecipado das instruções que serão despachadas. A esse conjunto de registradores que funcionam como *buffers* dá-se o nome de janela de instruções.

A utilização conjunta do recurso da janelas de instruções com técnicas de busca antecipada de instruções (*prefetching*) produz um significante aumento no desempenho dos processadores super escalares.

Nos processadores mais simples que empregam estas técnicas o mecanismo de janela de instruções se apresenta como uma fila onde a unidade de busca adiciona instruções de um lado e a unidade de despacho retira do outro. Devido ao problema com dependências de controle, sempre que um comando de desvio é despachado o conteúdo da janela de instruções deve ser descartado. Usando técnicas mais elaboradas, a unidade de busca antecipada de instruções pode ser capaz de identificar as instruções de desvio incondicional e carregar as instruções que serão requeridas após a transferência de controle. Veremos em seções subseqüentes mecanismos que minimizamos efeitos das dependências de controle.

Através do mecanismo de janelas em um processador super escalar, o escalonador pode fazer a analise dinâmica das instruções que serão despachadas e assim decidir se o despacho paralelo de duas ou mais instruções é possível ou não.

2.4. *PIPELINE*

A utilização de unidades funcionais independentes permite organizar o *hardware* do processador como se fosse uma linha de montagem. Por exemplo, suponha que a execução de uma instrução seja processada em três estágios, um estágio de busca e decodificação, outro para busca dos operandos e o estágio de execução.

Um processador com unidades funcionais independentes realizando cada tarefa, poderá iniciar o processamento de uma instrução e esperar que ela finalize o primeiro estágio e passe ao segundo para iniciar o processamento da instrução seguinte, sem esperar pela finalização completa da anterior. Deste modo, até três instruções podem estar em execução cada uma em um estágio diferente. A tabela apresentada na **figura 2.4** ilustra o aspecto deste *pipeline*.

Ciclo

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Busca e decodificação	**I1**	**I2**	**I3**			**I4**	**I5**	**I6**
Busca dos operandos		**I1**	**I2**	**I3**			**I4**	**I5**
execução			**I1**	**I2**	**I3**			**I4**

Figura 2.4: *Tabela de operação do Pipeline.*

Observe que cada instrução começa o processamento de um estágio assim que a instrução anterior passa ao estágio seguinte. A técnica do *pipeline* permite obter em condições ótimas a taxa de uma instrução por ciclo de máquina.

Na tabela observamos uma quebra no mecanismo de *pipeline* entre os ciclos 3 e 6, isto se deve à ocorrência de uma instrução de desvio. Instruções de desvio provocam dependências de controle, de modo que o processo de *pipeline* fica bloqueado até que o endereço da instrução sucessora seja conhecido. A esta quebra no mecanismo de *Pipeline* damos o nome de penalidade do desvio.

2.5. ESCALONAMENTO DE INSTRUÇÕES

Nas máquinas super escalares o algoritmo de escalonamento possui importância fundamental, visto que nas máquinas com esta arquitetura o despacho paralelo de instruções é a base de tudo. Existem duas formas de tratar o despacho paralelo de instruções em uma máquina super escalar. Através do escalonamento estático de instruções ou usando técnicas de escalonamento dinâmico.

2.5.1. ESCALONAMENTO ESTÁTICO DE INSTRUÇÕES

A técnica de escalonamento estático procura transferir para o nível do *software* a tarefa de organizar o código de um programa. Neste processo o compilador através de informações sobre a arquitetura do processador (número de unidades funcionais e etc.), procura gerar um código otimizado capaz de tirar proveito das facilidades de paralelismo da máquina. O escalonamento estático pode ser local ou global, conforme apresentaremos nas subseções seguintes.

2.5.1.1. ESCALONAMENTO ESTÁTICO LOCAL

Neste modo de escalonamento estático as instruções que serão analisadas devem pertencer a um mesmo bloco básico. Um bloco básico consiste em um conjunto de instruções limitados entre duas instruções que são destino de desvios ou por uma instrução destino de desvio e uma instrução de desvio. A **figura 2.5** mostra um trecho de programa e seus dois blocos básicos.

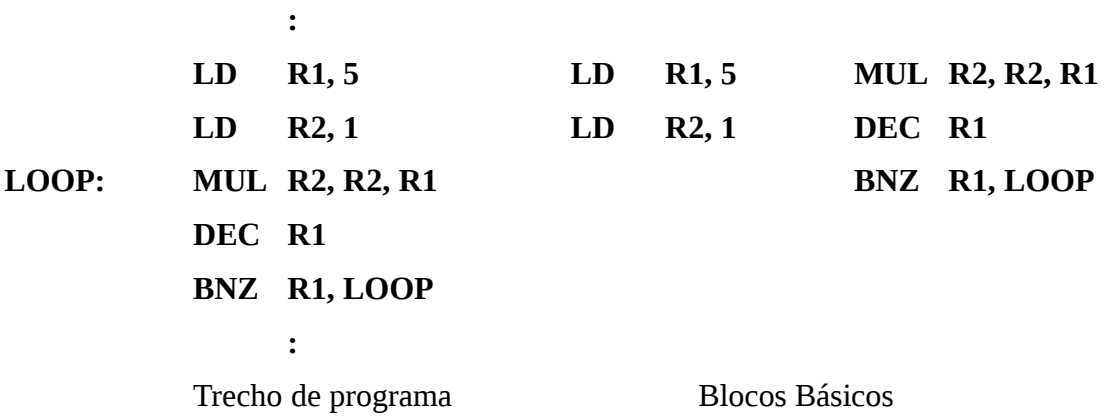


Figura 2.5: *Trecho de um programa com os blocos básicos.*

Após a divisão em blocos básicos cada um é analisado e otimizado independentemente procurando obter um tempo de execução ótimo, mas a obtenção de um tempo de execução ótimo para blocos básicos não significa necessariamente que o tempo total de processamento será reduzido. Isto é, o escalonamento localizado em blocos impede que seja realizada uma análise melhor do paralelismo existente, assim o código produzido pode não ser o mais eficiente. Deste modo desenvolveu-se um outro método de escalonamento estático denominado escalonamento estático global que mostraremos a seguir.

2.5.1.2. ESCALONAMENTO ESTÁTICO GLOBAL

Neste modo de escalonamento estático a reorganização do código envolve instruções pertencentes a blocos básicos diferentes.

O escalonamento estático global se desenvolve da seguinte forma: Inicialmente o programa é dividido em seus blocos básicos, então é construído um grafo de controle utilizando como nós os blocos básicos e arestas os caminhos possíveis ao fluxo de um programa. Em seguida são marcados os *traces*, isto é, os caminhos possíveis dentro do grafo desconsiderando as instruções de laço. A **figura 2.6** apresenta um exemplo de um grafo de controle.

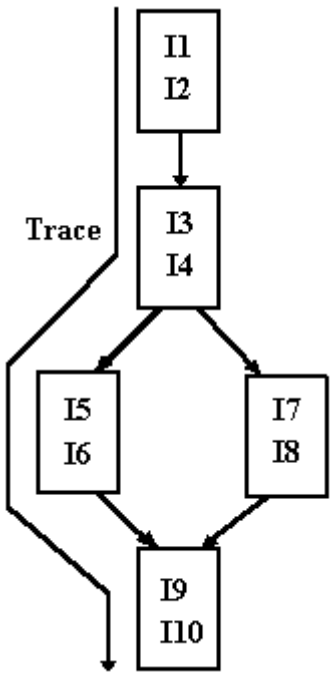


Figura 2.6: *Exemplo de grafo de controle.*

Definidos os *traces*, o algoritmo seleciona um deles e inicia o processo de escalonamento de suas instruções como se o mesmo fosse um bloco único. Ao considerarmos o *trace* como um bloco único poderemos tratá-lo de forma similar aos blocos básicos, isto é, poderemos escalonar as suas instruções de forma similar ao escalonamento estático local.

Para melhorar o desempenho do método procura-se utilizar o *trace* com maior chance de ser executado. Isto pode ser feito com o auxilio do programador, usando diretivas de compilação.

Observe que considerando um *trace* como um bloco contínuo o processo de escalonamento das instruções pode fazer com que instruções pertencentes a um bloco básico sejam reorganizadas em outro. Se tivéssemos certeza da execução exclusiva do *trace* considerado isto não levaria a conseqüências mais sérias, mas como na maioria dos casos a execução de um programa pode levar ao processamento de *traces* distintos, poderemos ter casos em que a equivalência semântica do código não será mantida. Por exemplo, na **figura 2.7** vemos dois *traces* possíveis, se o escalonamento iniciar pelo *trace 1* pode ocorrer da instrução **I5** ser reorganizada para despachar antes da instrução **I4** pertencente a outro bloco básico. De modo que se no momento da execução o programa for processado seguindo o caminho do *trace 2* o resultado final pode não ser o esperado, devido a equivalência semântica quebrada com a inserção do comando **I5**.

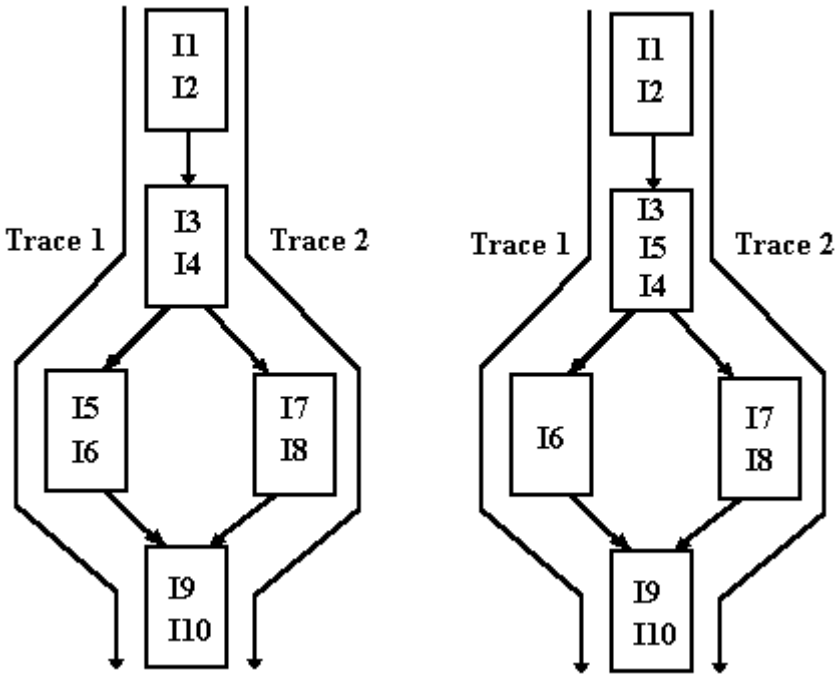


Figura 2.7: *Problema de equivalência semântica.*

Portanto, sempre que uma instrução for retirada de um bloco básico e inserida em outro, o compilador deverá tomar uma ação para garantir a equivalência semântica do programa. Isto é feito através da inserção de códigos de reparo. Um código de reparo que poderia ser utilizado no exemplo seria inserir uma instrução que desfaça a execução da instrução **I5** no inicio do bloco básico que contém **I7** e **I8**.

As movimentações de instruções entre blocos básicos podem provocar problemas de equivalência semântica que precisam ser corrigidos com a adição de códigos de reparo, assim o algoritmo de escalonamento estático global se torna muito mais complexo e de maior dificuldade de implementação, alem de consumir um tempo de processamento muito maior.

Uma simplificação do processo pode ser obtida com a introdução do escalonamento dinâmico das instruções, conforme será visto a seguir.

2.5.2. ESCALONAMENTO DINÂMICO

O escalonamento dinâmico é realizado pelo *hardware* com o programa já em execução. Neste mecanismo de escalonamento a cada ciclo as instruções que serão despachadas são analisadas a procura de dependências entre elas e as instruções em execução naquele momento. Não havendo dependência entre as instruções, estas são então despachadas. Caso haja alguma dependência o despacho, normalmente, será bloqueado.

O bloqueio no despacho de instruções não é uma solução desejável, assim os vários algoritmos de escalonamento dinâmico existentes procuram resolver o problema das dependências a bloquear o despacho de instruções.

Analisaremos nas subseções seguintes dois algoritmos de despacho dinâmico que implementaram máquinas pioneiras da tecnologia super escalar, o algoritmo de **Thorton** (a técnica do s*coreboard*), implementado no **CDC-6600** da **Control Data Corporation** e o algoritmo de Tomasulo, implementado na unidade de ponto flutuante do processador **IBM 360/91** da **IBM**.

2.5.2.1. ALGORITMO DE THORTON

O algoritmo de **Thorton** (vide [Edil92] e [Eliseu94]) foi inicialmente implementado em 1964 pela **Control Data Corporation** no modelo **CDC-6600** e é ainda hoje utilizado em processadores de alto desempenho. O mecanismo desenvolvido por **James E. Thorton** foi o do *scoreboard* e tinha como objetivo manter as unidades funcionais do processador em constante operação. A organização básica de uma arquitetura com *scoreboard* funciona da seguinte forma.

Para cada unidade funcional o *scoreboard* mantém três *bits* de controle: o *bit* **B** indica se a unidade correspondente já está reservada a alguma instrução aguardando a liberação de recursos (registradores), o *bit* **X** informa que a unidade está processando uma instrução e o *bit* **R** indica as unidades que finalizaram a execução de uma instrução. Ainda para cada unidade funcional é mantida a indicação dos registradores usados como operandos fonte e destino. Além disso o mecanismo de *scoreboard* mantém um *bit* para cada registrador indicando aqueles que são operando destino de alguma instrução já despachada. A **figura 2.8** apresenta o esquema de controle empregado no mecanismo do *scoreboard*.

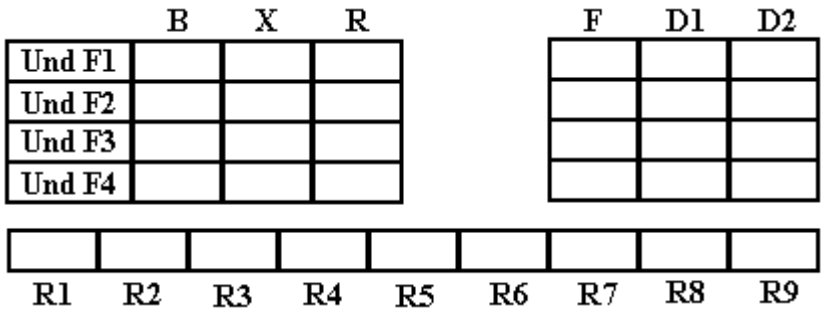


Figura 2.8: *O mecanismo do scoreboard.*

Com base nestas informações, o *scoreboard* decide se uma instrução pode ser despachada, se uma instrução já despachada pode entrar em execução ou se o resultado de uma instrução finalizada pode atualizar o registrador destino. O mecanismo opera da seguinte forma:

Inicialmente, ao receber uma instrução, esta é analisada para determinar, através dos *bits* de controle, se existe uma unidade funcional disponível para a sua execução e se o operador destino utilizado por ela é operador destino em alguma outra instrução já despachada, isto é, se não há uma dependência de saída entre a instrução analisada e uma outra já despachada. Se uma das duas hipóteses não for satisfeita o despacho da instrução é bloqueado. Caso contrario, a instrução é enviada à unidade funcional correspondente e o *bit* **B** de controle da referida unidade é ativado.

Depois o algoritmo verifica os registradores fonte das instruções despachadas que ainda não foram executadas, isto é, as que possuem o *bit* **X** inativo. Se nenhum dos registradores fonte é o destino de alguma instrução despachada anteriormente, o algoritmo libera o acesso aos operandos da instrução e esta entra em execução. Então o *bit* **X** passa para o estado ativo. Com este procedimento é assegurado que dependências verdadeiras serão respeitadas.

Então depois, o algoritmo analisa as instruções já concluídas, isto é, as que possuem o *bit* **R** ativo, se o registrador destino destas operações não estiver sendo usado como operando fonte por nenhuma outra instrução já despachada o algoritmo então libera a sua atualização. Caso contrário, o mecanismo impede a atualização do registrador por parte da unidade funcional. Deste modo é assegurado que anti-dependências serão respeitadas.

A grande vantagem do mecanismo de *scoreboard* é que não há necessidade de aguardar o término de uma instruções em execução para se despachar a seguinte.

2.5.2.2. ALGORITMO DE TOMASULO

Em 1967 a **IBM** lançou um novo modelo de processador, o **IBM 360/91,** voltado para aplicações científicas com intenso volume de processamento numérico. Este processador incorporou importantes conceitos arquiteturais. Principalmente na unidade de despacho de instruções de ponto flutuante onde o algoritmo de **Tomasulo** (vide [Toma65]) foi implementado pela primeira vez.

Além da inovação no despacho de instruções este processador implementou os seguintes novos recursos: *reservation stations*, *common data bus* e despacho associativo.

Reservation stations, também chamadas de unidades virtuais, são como *buffers* de espera para as unidades de execução. Durante o processamento de um programa as instruções despachadas são transferidas a elas enquanto aguardam a disponibilidade de recursos como os operandos e a própria unidade real.

O uso de unidades virtuais permite ao processador despachar um maior número de instruções por ciclo de máquina sem o custo adicional de ter unidades duplicadas.

Common data bus (**CDB**) é um barramento que conecta os componentes da unidade de ponto flutuante (registradores, unidades funcionais e *reservation stations*) e funciona como um caminho de comunicação por onde as unidades compartilham os resultados produzidos com o banco de registradores.

No algoritmo de **Tomasulo** cada unidade virtual possui campos identificando os recursos utilizados, chamados de *tags* associativos. Quando uma instrução é despachada para as *reservation station* correspondente os campos *tags* identificam a operação e após a sua execução o resultado e a identificação do *tag* se propaga pelo *common data bus* de modo que as unidades que aguardam o resultado deste *tag* possam iniciar a execução.

Além do campo *tag*, cada registrador possui um *bit* de controle (*busy bits*) que indica se o valor de um registrador está disponível para uso ou se está sendo avaliado. A **figura 2.9** apresenta o esquema de controle empregado no algoritmo de **Tomasulo**.

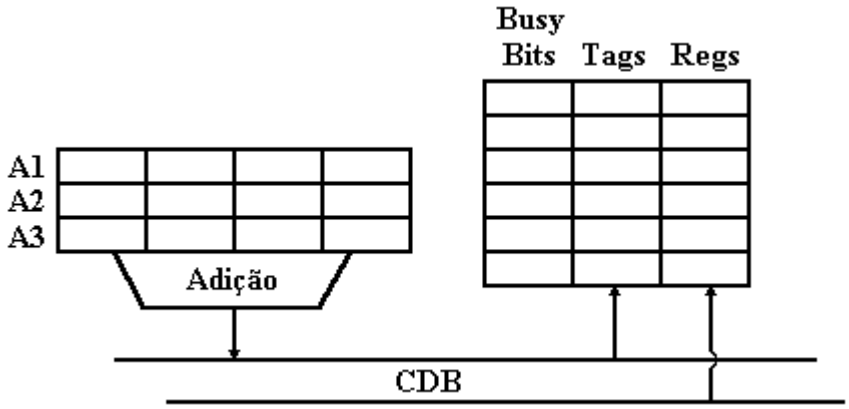


Figura 2.9: *Esquema de controle empregado no algoritmo de* ***Tomasulo***.

Para facilitar a compreensão do algoritmo de **Tomasulo** suponha um trecho de código, como o apresentado na **Figura 2.10**.

```
                :
1.      ADD   R0, R0, R1
2.      MUL   R2, R2, R0
                :
```

Figura 2.10: *Trecho de código de um programa.*

Ao despachar a primeira instrução à unidade de adição esta instrução é encaminhada a uma das *reservation stations* disponíveis, então o *tag* associado ao registrador **R0** recebe o valor do identificador da unidade funcional que modificará o seu conteúdo e o *busy bit* é ativado de modo a indicar que o valor deste registrador está sendo avaliado, conforme pode ser visto na **figura 2.11**. Assim quando o resultado da operação for obtido este será propagado pelo **CDB** junto com o *tag*, o registrador **R0** identificando o *tag* transmitido atualizará o seu conteúdo com o valor propagado pelo **CDB**.

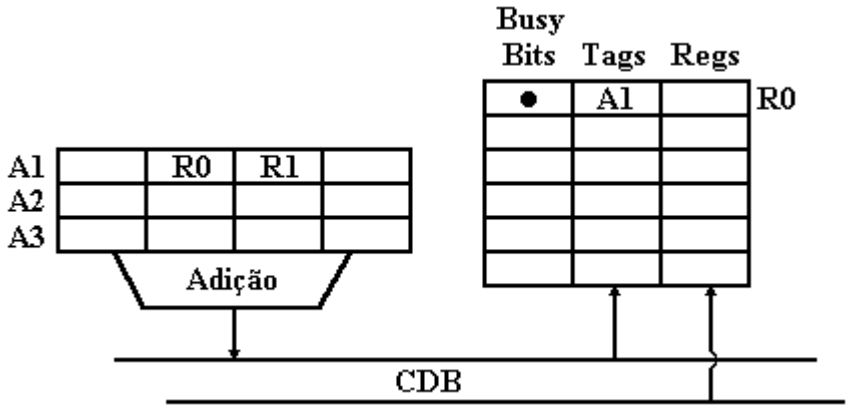


Figura 2.11: *Situação após o despacho da instrução* ***ADD R0, R0, R1***.

Se a operação de multiplicação for despachada antes da finalização da instrução de adição então o mecanismo transfere à *reservation station* da unidade de multiplicação o valor do registrador **R2** e o valor do *tag* associado ao registrador **R0** para o campo *tag* do operando correspondente na *reservation station* da unidade de multiplicação, como pode ser observado na **figura 2.12**. Isto porque o registrador **R0** está em uso (*busy bit* acionado) e o seu conteúdo depende do resultado de **A1**. Então o *tag* associado ao registrador **R2** recebe o identificador da unidade de multiplicação e o *busy bit* correspondente é ativado.

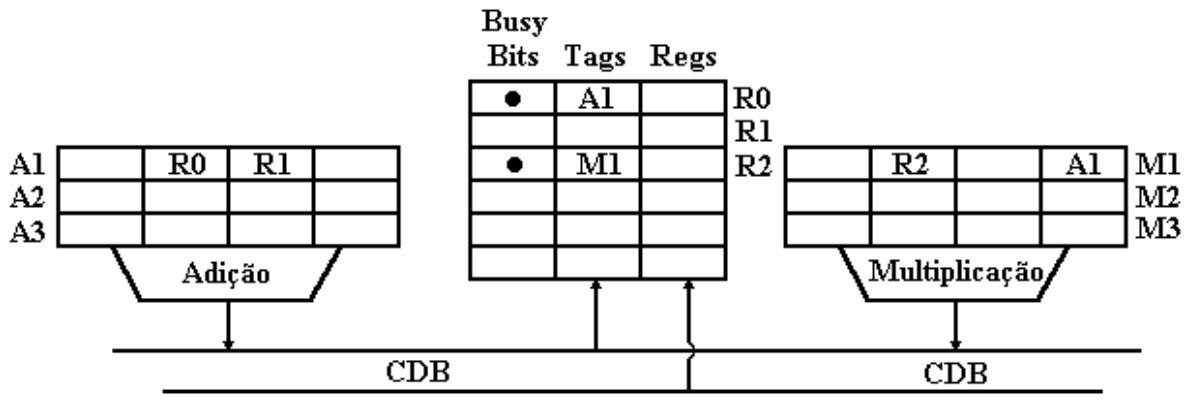


Figura 2.12: *Situação após o despacho da instrução* ***MUL R2, R2, R0***.

Desta forma o algoritmo de **Tomasulo** permitirá o despacho da instrução de multiplicação mas não a sua execução até que a instrução de adição tenha sido concluída e o seu resultado propagado através do **CDB, g**arantindo que na presença de dependências verdadeiras a ordem de precedência das instruções será mantida.

Existe apenas uma situação em que o algoritmo de **Tomasulo** implementado no **IBM 360/91** bloqueia o despacho de instruções. Quando não houver unidade virtual disponível para o despacho da instrução do topo da fila. Assim o algoritmo bloqueará o despacho desta e das instruções subsequentes até que uma das *reservation stations* da unidade requerida pela instrução seja liberada.

Capítulo 3

Recursos para Exploração do Paralelismo

3.1. RENOMEAÇÃO DE REGISTRADORES

No capítulo anterior analisamos os tipos de dependências de dados que podem ser encontradas durante a execução de um programa. As dependência verdadeira, anti-dependência e dependência de saída.

Conforme foi visto no capítulo anterior, as dependências dos tipos anti-dependência e dependência de saída podem ser contornadas e por isso são também chamadas de dependências falsas. Os processos que resolvem esses tipos de dependência, frequentemente derivam da técnica que será apresentada neste capítulo, chamada de renomeação de registradores. Já a dependência do tipo dependência verdadeira não pode ser evitada e por este motivo recebe este nome.

A técnica de renomeação de registradores consiste em disponibilizar temporariamente um outro registrador para substituir o registrador envolvido na dependência de dados entre as instruções anti-dependentes e com dependência de saída. Por exemplo, considere o trecho de código apresentado na **figura 3.**1.

```
        :
1.  ADD R0, R1, R2
        :
2.  MUL R1, R2, R3
        :
```

Figura 3.1: *Trecho antes da renomeação.*

As duas instruções possuem uma relação de anti-dependência entre si (o registrador **R1** é utilizado como fonte na primeira instrução e como destino na segunda). Pela técnica de renomeação de registradores devemos substituir o registrador envolvido (**R1**) por um outro registrador que esteja disponível (por exemplo **R5**) afim de eliminar a relação de dependência. Assim, observando o mesmo trecho de código após a renomeação dos registradores, **figura 3.2**, vemos que a relação de dependência entre as instruções desapareceram.

```
        :
1.  ADD R0, R1, R2
        :
2.  MUL R5, R2, R3
        :
```

Figura 3.2: *Trecho após a renomeação.*

É importante notar que se existir alguma instrução, após o trecho de código, que referencie **R0** como registrador fonte, será necessário usar o registrador **R5** em seu lugar.

Considerando agora o trecho de código apresentado na **figura 3.3**, onde duas instruções possuem dependência de saída entre si, isto é, um mesmo registrador é utilizado como destino pelas duas instruções.

```
        :
1.  ADD R0, R1, R2
        :
2.  MUL R0, R2, R3
        :
```

Figura 3.3: *Trecho antes da renomeação.*

Aplicando a técnica de renomeação de registradores, substituímos o registrador **R0** na segunda instrução por algum outro registrador que esteja disponível (por exemplo **R5**). Deste modo eliminamos a relação de dependência permitindo o despacho simultâneo das duas instruções, conforme pode ser visto na **figura 3.4**.

```
        :
1.  ADD R0, R1, R2
        :
2.  MUL R5, R2, R3
        :
```

Figura 3.4: *Trecho após a renomeação.*

Alguns processadores que utilizam técnicas de renomeação possuem dois conjuntos de registradores, um disponível aos programadores e outro disponível apenas ao mecanismo de renomeação. Nestes processadores durante a execução do código, as instruções a serem despachadas são analisadas e seus registradores renomeados de modo a eliminar as dependências falsas. Após a execução destas instruções os registradores originais são atualizados na seqüência original.

Outros processadores utilizam o recurso de registradores virtuais e reais. São considerados registradores virtuais os utilizados pelas instruções de máquina e reais os utilizados pelo processador no momento da execução das instruções. Para o uso deste recurso os processadores devem possuir uma tabela associando cada registrador virtual ao seu correspondente registrador real e uma outra tabela associando cada registrador real a um contador. Cada contador associado a um registrador real matem o registro da quantidade de instruções que utilizam o registrador e que se encontram em execução num determinado instante. Este registro é feito incrementando o contador toda vez que uma instrução despachada contiver o registrador e decrementando sempre que uma instrução que utiliza o registrador finalizar a execução. Assim, quando o contador possuir o valor zero o registrador real correspondente estará disponível e poderá ser reutilizado.

O algoritmo de Thorton apresentado no capítulo anterior não apresenta o mecanismo de renomeação de registradores e por isso sempre que instruções com dependência de saída são encontradas o despacho em paralelo é bloqueado, e quando instruções com dependência verdadeira ou anti-dependência são encontradas tais instruções são despachadas mas ficam aguardando no interior da CPU pela liberação dos registradores apropriados para a execução. Já o algoritmo de Tomasulo, também visto no capítulo anterior, apresenta uma forma de renomeação de registradores indireta com o uso dos *tags* associativos de modo que instruções com dependência de saída podem ser executadas em paralelo.

O simulador **Dlxwin** possui um recurso de unidades de reserva e *tags* associativos similar ao algoritmo de Tomasulo para escalonar instruções com dependência de dados e evitar a paralisação do mecanismo de despacho.

3.2 MÚLTIPLOS REGISTRADORES DE CONDIÇÃO

Com a possibilidade de executar múltiplas instruções simultaneamente a utilização de apenas um registrador de condição (*flag*) se tornou um agravante no desempenho de um processador super escalar. A presença de apenas um registrador de condição significa que a execução de instruções condicionais deverá ser seqüencial e ordenada conforme a precedência original existente no código. Isto é um problema que reduz significativamente o desempenho dos processadores super escalares já que em média 15 a 30% das instruções executadas por um processador correspondem à instruções de desvios (vide [Edil92]).

A solução para contornar este problema é a adoção de múltiplos registradores de condição (*flags*). A existência de múltiplos *flags* torna possível a execução de mais de uma instrução condicional simultaneamente, o que aumenta significativamente o desempenho do processador.

O controle da utilização do conjunto de registradores de condição pode ser efetuado pelo *software* ou pelo *hardware*. Na implementação feita pelo *software* os resultados das instruções condicionais são armazenadas em registradores de propósito geral possibilitando a execução simultânea de múltiplas instruções condicionais cada qual colocando o resultado de sua avaliação em um registrador de propósito geral diferente. Sendo a quantidade de instruções condicionais executadas em paralelo limitada apenas pelo número de registradores e unidades funcionais existentes no processador. Observe, por exemplo, o trecho de código da **figura 3.5**.

```
        :
1.      SEQ   R0, R1, R2
2.      BEQZ  R0, LOOP1
3.      SLE   R3, R4, R5
4.      BNEZ  R3, LOOP2
        :
```

Figura 3.5: *Trecho de código de um programa.*

Após a execução da primeira instrução o registrador **R0** conterá o resultado da comparação entre **R1** e **R2**. A segunda instrução, então, produzirá um salto se **R0** for igual a zero, isto é, se **R1** for diferente de **R2**. Analogamente, a terceira instrução colocará o resultado da comparação entre **R4** e **R5** no registrador **R3** e a quarta instrução produzirá um salto se **R3** for diferente de zero, ou seja, **R4** menor ou igual a **R5**.

Observe que as instruções do exemplo são independentes quanto ao compartilhamento dos dados, mas são dependentes quanto ao fluxo do programa. Sendo que a terceira e quarta instruções só serão executadas se o salto na segunda instrução não for efetuado, mas nesta situação o processador poderá tirar vantagem da execução paralela das duas instruções condicionais.

Podemos verificar que as vantagens obtidas nesta técnica dependem muito mais da otimização do código gerado pelo compilador de modo a tirar o maior proveito do processamento paralelo. Um compilador não otimizador poderia gerar um código equivalente mas com dependência de dados entre as duas instruções condicionais, o que sobrecarregaria demasiadamente as unidades funcionais responsáveis pela execução destas instruções e não beneficiaria a execução das duas instruções condicionais durante a maior parte do tempo.

É interessante ressaltar que para esta implementação o modo como os programas podem ser implementados são completamente diferentes do modo de programação atual. Podemos manter uma seqüência de instruções condicionais e depois uma seqüência de instruções de saltos referenciado pela correspondente instrução condicional, como pode ser observado na **figura 3.6**.

```
                :
1.      SEQ   R0, R1, R2
2.      SLE   R3, R4, R5
3.      BEQZ R0, LOOP1
4.      BNEZ R3, LOOP2
                :
```

Figura 3.6: *Trecho de um programa escrito para maquinas com múltiplos flags.*

A implementação do simulador **Dlxwin** faz uso do mecanismo de múltiplos registradores de condição para operações com número de ponto fixo, sendo o controle realizado por software. No **Dlxwin** qualquer um dos registradores de propósito geral existentes, com exceção do registrador **R0,** podem ser utilizados como registradores de condição.

3.3. TRATAMENTO DE DESVIOS

Operações de desvios representam de 15 a 30% das instruções existentes em um programa (vide[Edil92]) e os seus efeitos sobre o desempenho dos processadores super escalares é muito grande.

Desvios incondicionais não representam grandes problemas, pois é possível implementar na arquitetura do processador mecanismos para identificar a instrução de desvio e o endereço alvo durante a fase de decodificação. Já os desvios condicionais por dependerem da avaliação de uma expressão introduzem pelo menos um ciclo de espera.

Paralisar o despacho de instruções até que o endereço alvo de um desvio seja conhecido não é uma estratégia eficiente, então para reduzir o efeito das dependências de controle varias técnicas foram propostas: desvio com retardo (*delayed branch*), previsão de desvio (*branch prediction*) e busca antecipada as instruções dos dois fluxos.

3.3.1. DESVIO COM RETARDO

Uma das formas de atenuar o efeito da dependência de controle é utilizar o recurso de desvio com retardo (*delayed branch*). Esta estratégia procura minimizar o custo impondo a execução de algumas instruções posteriores ao desvio. O número de instruções executadas depende diretamente do tempo de espera até que se conheça o fluxo a ser tomado. Por exemplo, considere o trecho de código apresentado na **figura 3.7**.

```
                :
1.              ADD     R1, R0, 10
2.      LOOP1:  ADD     R2, R2, 1
3.              SUB     R1, R1, 1
4.              BNEQZ   R1, LOOP1
5.              MULTR2, R2, 2
                :
```

Figura 3.7: *Trecho de código de um programa.*

Se o retardo imposto pela instrução (4) for de um ciclo então a instrução seguinte (5) será sempre executada para cada interação do *looping* e isto levará a um resultado diferente do esperado pelo programador. Uma solução é rescrever o código acrescentando uma instrução **NOP** (*no-operation*) após a instrução (4), como pode ser visto na **figura 3.8**.

```
                :
1.              ADD     R1, R0, 10
2.      LOOP1:  ADD     R2, R2, 1
3.              SUB     R1, R1, 1
4.              BNEQZ   R1, LOOP1
5.              NOP
6.              MULTR2, R2, 2
                :
```

Figura 3.8: *Exemplo de trecho de um programa com instrução de **NOP** após um desvio.*

Esta solução apesar de corrigir um problema acrescenta outro ao levar para o nível de *software* o problema dos ciclos de espera. Existem técnicas mais efetivas de manipulação do código (técnicas de otimização) que podem ser aplicadas. (1) Uma delas consiste em deslocar o desvio **n** instruções antes (onde **n** é o retardo do desvio), mas apenas quando as instruções precedentes não afetam a avaliação da expressão do desvio esta técnica poderá ser empregada. (2) Outra é inserir após o desvio uma cópia do trecho do programa, formado pelos n comandos que iniciam na instrução destino do desvio. Esta técnica tem a desvantagem de apenas poder ser utilizada se o endereço alvo for conhecido e os dois fluxos de execução forem independentes. (3) Um terceiro processo consiste em admitir que o desvio não será tomado e a execução das instruções seguintes ao desvio sempre será executada. Este método traz desvantagens semelhantes a técnica (2).

Outra forma de atenuar o custo dos desvios é procurar prever o seu resultado e prosseguir com o despacho das instruções de um dos fluxos possíveis. Se eventualmente o resultado previsto não for correto as instruções acessadas incorretamente serão descartadas. A eficiência deste método está intimamente ligado a taxa de acerto das previsões.

3.3.2. MECANISMOS DE PREVISÃO DE DESVIOS

Nesta seção analisaremos as técnicas de previsões de desvios existentes. Existem duas estratégias para implementação de um mecanismos de previsão de desvios: (i) através de previsões estáticas ou (ii) utilizando mecanismos de previsões dinâmicas. Analisaremos na subseção seguinte as previsões estáticas.

3.3.2.1. PREVISÕES ESTÁTICAS

As técnicas de previsões estáticas de desvios são implementadas assumindo que um comando de desvio será sempre tomado, ou não. Existem três estratégias empregadas em previsões fixadas de desvios:

i) o desvio será considerado sempre tomado;
ii) o desvio será considerado sempre não-tomado;
iii) a previsão depende do código da instrução.

DESVIO SERÁ SEMPRE TOMADO

Esta estratégia se baseia no fato de que a maioria das instruções de desvio modificam o fluxo de execução.

Desta forma, assim que uma instrução de desvio for decodificada a unidade de busca desvia o controle e começa a pegar instruções a partir do endereço alvo do desvio. Se após a execução do desvio ficar determinado que a previsão foi correta, o processamento continua com as instruções seguintes já na fila, reduzindo a zero o custo do desvio, mas se ficar determinado que a previsão foi incorreta a janela de instruções é descartada e o mecanismo de busca continua a partir da próxima instrução. Experimentos mostraram que a taxa de acerto das previsões variam de 57,4 a 99,6% dependendo da característica do programa em execução (vide [Edil92]).

DESVIO SERÁ SEMPRE NÃO-TOMADO

A estratégia de não desviar o fluxo de controle imediatamente quando se encontra uma instrução de desvio é a implementada pela maioria das arquiteturas. Esta técnica mantém o fluxo natural de busca das instruções sendo por isso de fácil implementação. Experimentos realizados mostram que a taxa de acertos das previsões variam de 22,2 a 46,0% (vide [Edil92]).

A PREVISÃO DEPENDE DA INSTRUÇÃO

Algumas arquiteturas ao invés de fixarem uma das opções de fluxo (desvio tomado ou não-tomado) para todas as instruções de desvio, elas escolhem a opção mais apropriada em função do código da instrução. Experimentos realizados por **J.E.Smith** apresentaram uma taxa média de acertos de 86,7% utilizando esta técnica (vide [Edil92]).

3.3.2.2. PREVISÕES DINÂMICAS

Os mecanismos de previsão dinâmica baseiam a escolha do fluxo de controle em função de interações anteriores da mesma instrução de desvio. A idéia deste mecanismo é que toda instrução de desvio possui uma tendência que pode ajudar nas previsões das futuras interações do comando. Dois processos de previsão de desvio são bastante difundidos:

i) *Branch History Table* (BHT);
ii) *Branch Target Buffer* (BTB).

TABELA DE HISTÓRIA DOS DESVIOS

A Tabela de História dos Desvios (BHT) é uma tabela que armazena em cada entrada um ou mais *bits* para registrar a história dos desvios. O acesso e gerenciamento das entradas se assemelha ao empregado nas memórias cache, podendo ser associativo, de mapeamento direto ou por conjunto.

Após o estágio de busca a instrução é decodificada e se for identificada como um desvio os *bits* menos significativo do endereço é utilizado para acessar a tabela. Existindo uma entrada na tabela associada o mecanismo utiliza os *bits* de previsão para determinar se o desvio será ou não tomado. A **figura 3.9** exemplifica o mecanismo.

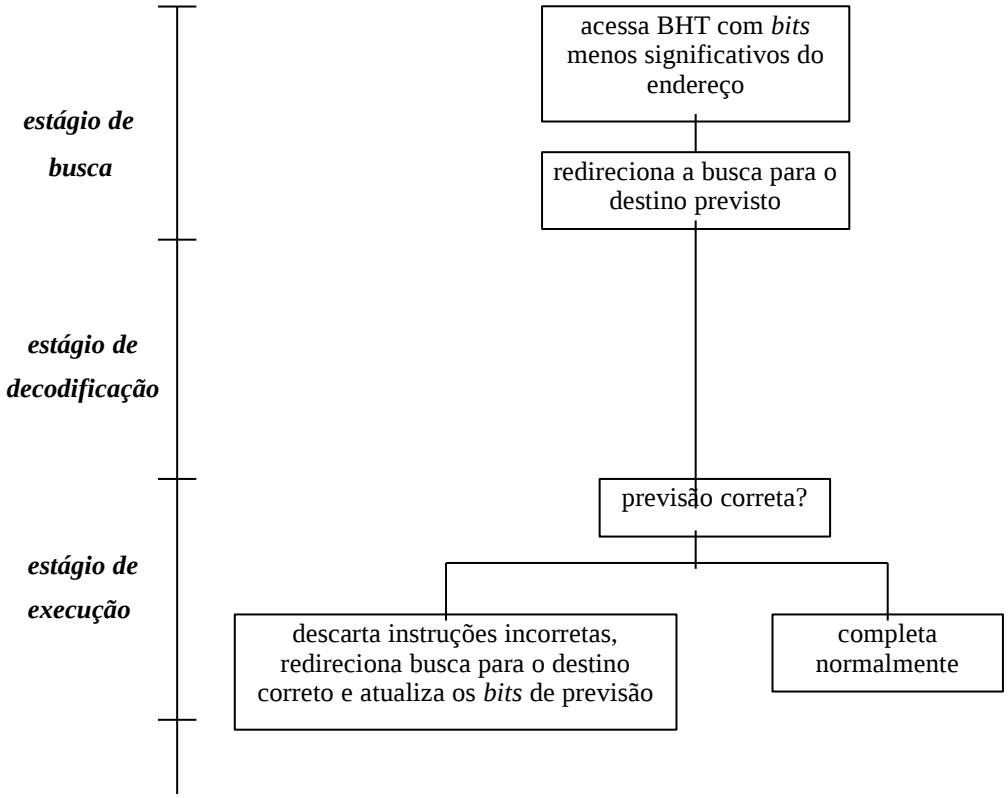


Figura 3.9: *Diagrama de funcionamento do mecanismo de previsão de desvios com BHT.*

A quantidade de *bits* de previsão utilizados em cada entrada melhora o desempenho do mecanismo, pois registra um número maior de ocorrências de cada desvio.

O caso mais simples utiliza apenas um *bit* de previsão, este *bit* armazena a informação do resultado do ultimo desvio realizado. Se o resultado da ultima execução do desvio foi desvio não-tomado, então a previsão será de não desviar o fluxo de controle, mas se o resultado foi de desvio tomado, a previsão será desviar o fluxo de controle. O diagrama da **figura 3.10** apresenta as transições de estados para mecanismos de previsão de desvios utilizando um *bit* para a previsão.

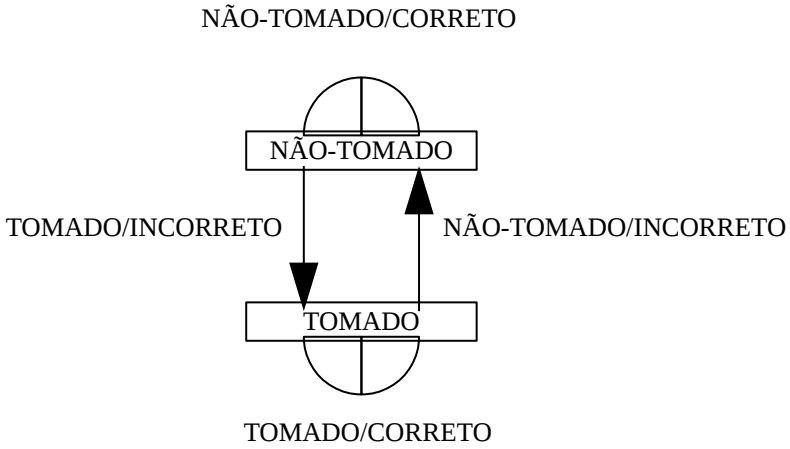


Figura 3.10: *Diagrama de transição de estados para um bit de previsão.*

O uso de um de um único *bit* de previsão tem suas desvantagens. Quando o código executado possui dois laços encadeados a taxa de acerto das previsões diminui muito. Por exemplo suponha o trecho de código apresentado na **figura 3.11**.

```
:
for(i = 0; i < N1; i++)
        for(j = 0; j < N2; j++)
:
```

Figura 3.11: *Trecho de código de um programa.*

Após a última interação do laço mais interno o *bit* de previsão se torna não-tomado, mas o laço interno será reinicializado na interação seguinte do laço mais externo e a primeira previsão do desvio falhará. Nesta situação teremos duas falhas para cada interação do laço externo e dependendo da relação de amplitude entre eles a taxa de acertos pode se tornar muito ruim. Uma forma de contornar este problema é utilizar dois *bits* para previsão dos desvios. O diagrama de estados apresentado na **figura 3.12** ilustra o mecanismo de previsão de desvios utilizando dois *bits* de previsão.

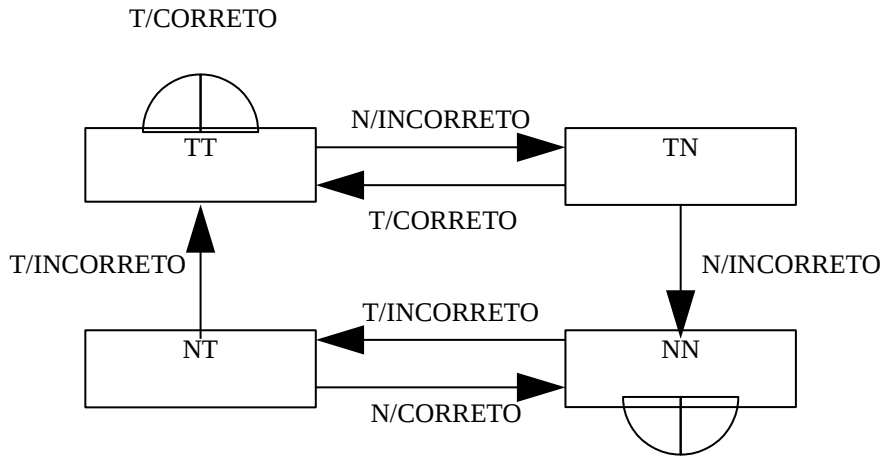


Figura 3.12: *Diagrama de estados para dois bits de previsão.*

A previsão é sempre função do *bit* mais antigo deste modo garantimos uma taxa de acerto maior em situações de laços alinhados. Experimentos mostram que a taxa de acerto utilizando dois *bits* de previsão alcançam em média 85%.

O uso da BHT como mecanismo de previsão de desvios possui uma deficiência para instruções de desvio que não possuem o endereço alvejado no próprio código. Nessa situação o mecanismo de busca terá que parar de acessar as instruções seguintes até que se conheça o endereço alvo. Existem muitos comandos de desvio com estas características tais como retorno de subrotinas e chamadas ao sistema operacional. Para contornar o problema podemos utilizar uma outra forma de tabela denominada *Branch Target Buffer* (BTB).

TABELA DE DESTINO DOS DESVIOS

A Tabela de Destino dos Desvios (*Branch Target Buffer*) é um aperfeiçoamento do mecanismo da BHT descrito na seção anterior. Nesta técnica cada entrada armazena o endereço alvejado pelo desvio, desta forma as instruções de desvio que não carregam o endereço destino no próprio código são previstas como qualquer outro tipo de desvio e o *hardware* implementado no estágio de busca se torna muito mais simples, pois não precisa fazer a decodificação da instrução nem calcular o endereço efetivo do destino do desvio.

O funcionamento é semelhante ao da BHT. No caso da instrução não possuir uma entrada na BTB o endereço destino do desvio é inserido junto com o endereço do desvio. Em toda previsão incorreta o valor do endereço de destino do desvio contido na tabela é atualizado com o destino correto.

No simulador **Dlxwin** foi implementado um mecanismo de previsão de desvios utilizando um *Branch Target Buffer* com dois *bits* de previsão e um processo de execução especulativa dos desvios utilizando um *Reorder Buffer* e Registradores Futuros, técnicas que serão apresentadas na seção seguinte.

3.4. EXECUÇÃO ESPECULATIVA

Os mecanismos de previsão de desvios abordados nas seções anteriores resolvem apenas uma parte dos problemas. As técnicas de previsão possibilitam que a busca antecipada das instruções seguintes ao desvio continue por um dos caminhos do fluxo de controle, de modo que a unidade de controle ao requisitar por estas instruções as encontre disponíveis na fila, mas com os processos de previsão de desvios não é possível executar as instruções. Para solucionar esta deficiência vários métodos de execução especulativa foram desenvolvidos. Veremos nas subseções seguintes três métodos bastante difundidos:

i) *Buffer* de Reordenação (*Reorder Buffer*);
ii) *Buffer* de História (*History Buffer*);
iii) Registradores Futuros (*Future File*);

3.4.1. *BUFFER* DE REORDENAÇÃO

O *Reorder Buffer* se apresenta como uma fila onde cada entrada armazena informações sobre a unidade funcional que executa a instrução, o registrador destino, o seu novo valor e um *bit* indicando se a entrada está pronta para atualização.

O mecanismo é bastante simples funcionando da seguinte forma: a cada instrução despachada uma nova entrada é inserida na tabela com o *bit* de instrução pronta para atualização limpo e o identificador do registrador destino da instrução. Quando a execução da instrução é finalizada a entrada correspondente na fila é atualizada e o *bit* instrução pronta ativado. Assim, quando a instrução atinge o topo da fila e tem o *bit* de pronta ativado o registrador associado é então atualizado.

Este processo permite a execução especulativa das instruções posteriores ao desvio, mantendo a ordem de atualização dos registradores. Por exemplo, quando uma instrução de desvio é despachada para a unidade de execução correspondente, é inserida uma nova entrada no *buffer* de reordenação. Toda instrução posterior ao desvio terá uma entrada na fila posicionada após a entrada associada ao desvio e portanto só atualizarão seus registradores destinos quando estas chegarem ao topo da fila, isto é, após o resultado do desvio ser

conhecido. Então se uma previsão for incorreta basta eliminar todas as entradas do *buffer* de reordenação posteriores ao desvio que será mantida a integridade dos dados nos registradores.

3.4.2. *BUFFER* DE HISTÓRIA

O *History Buffer* é um processo semelhante ao *buffer* de reordenação, mas com uma vantagem. Enquanto o *buffer* de reordenação disponibiliza os resultados das instruções apenas quando estas alcançam o topo da fila, o *buffer* de história disponibiliza imediatamente, liberando desta forma as instruções que são dependentes.

A idéia é armazenar o valor original do registrador destino junto com a identificação da unidade de execução e do registrador destino, na fila do *buffer* no momento do despacho da instrução. Assim que o resultado da instrução for conhecido seu registrador destino é atualizado possibilitando que instruções dependentes possam ser despachadas mais rapidamente.

Mesmo que um comando de desvio seja despachado o mecanismo continua a execução das instruções seguintes e se após a execução do desvio a previsão for identificada como incorreta o *buffer* é percorrido a partir da ultima entrada até a entrada associada ao desvio, atualizando os valores dos registradores destino com o valor original desta forma o estado original dos registradores é recuperado.

O processo de restauração do estado inicial do processador pode ser significativamente demorado, dependendo do numero de entradas no *buffer*. Isto porque a varredura para restauração das condições iniciais dos registradores é feita seqüencialmente. O procedimento que veremos a seguir corrige este problema.

3.4.3. REGISTRADORES FUTUROS

A técnica de registradores futuros (*Future File*) é estruturalmente dependente do mecanismo do *Buffer* de reordenação apresentado na suseção anterior. Nesta técnica cada registrador é duplicado em dois conjuntos, conjunto futuro e conjunto real.

Toda instrução despachada recebe uma entrada associada no *buffer* de reordenação. Quando a execução da instrução é concluída o resultado é armazenado no registrador futuro correspondente e fica disponível às outras instruções. Assim que as entradas atingem o topo da fila os registradores reais são atualizados a partir dos registradores futuros.

Quando uma instrução de desvio é lançada, as instruções sucessivas no caminho do fluxo selecionado são despachadas e executadas. Se após a execução do desvio for determinado que a previsão foi incorreta, o conteúdo do conjunto de registradores reais é copiado para o conjunto futuro, as entradas no *buffer* seguintes ao desvio descartadas e o caminho do fluxo de busca as novas instruções redirecionado.

Este método apresenta grande vantagem sobre o *buffer* de história, porque as cópias entre os registradores podem ser feitas em paralelo representando um custo muito menor.

As técnicas de execução expeculativa utilizando *buffer* de reordenação e registradores futuros, e a técnica de previsão de desvios com o mecanismo de *Branch Target Buffer*, são alguns dos recursos implementados no simulador **Dlxwin**.

Capítulo 4

Exemplos de Arquiteturas Super escalares

As arquiteturas que apresentaremos a seguir são exemplos de arquiteturas super escalares reais, e tiveram seus algoritmos de despacho implementados no simulador **Dlxwin**.

4.1. O PROCESSADOR PENTIUM

Este processador possui três unidades funcionais (vide [Eliseu94]). A unidade FPU responsável pela execução de instruções de ponto flutuante e as unidades U-pipe e V-pipe responsáveis pela execução de instruções aritméticas e lógicas com números inteiros, acesso à memória e instruções de desvio.

As unidades U-pipe e V-pipe são idênticas, isto é, possuem o mesmo *pipeline*. Deste modo o Pentium se comporta como se fosse constituído por dois processadores paralelos. Devido a esta implementação o algoritmo de despacho deste processador se tornou muito simples. O esquema do *pipeline* super escalar do processador Pentium apresentado na **figura 4.1** mostra esta simplificação.

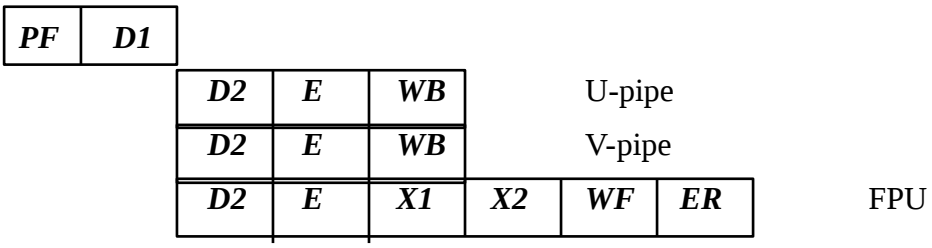


Figura 4.1: *Pipeline super escalar do Pentium.*

O algoritmo de despacho do Pentium

A cada ciclo o estágio de *prefetch* **PF** acessa duas instruções. O estágio seguinte **D1** decodifica e despacha as instruções. Identificando uma instrução de desvio é feito uma previsão do resultado dinamicamente através do mecanismo de *Branch Target Buffer* e as instruções seguintes passam a ser acessadas a partir do endereço destino previsto e o despacho destas instruções fica bloqueado até que o desvio seja resolvido.

É o algoritmo de despacho que analisa as instruções e decide pelo despacho simultâneo ou não segundo três condições: (1) As instruções devem ser simples, isto é, não podem ser instruções de múltiplos ciclos de latência, como funções de manipulação de *strings*, que provocariam conflitos na utilização dos recursos. (2) O registrador destino de uma instrução não deve coincidir com o registrador fonte e destino da outra, isto é, as instruções não podem ter dependência de dados do tipo verdadeira e dependência de saída. (3) A primeira instrução do par analisado não pode ser uma instrução de desvio. Se uma das três condições ocorrer apenas a primeira das duas instruções será despachada e no próximo ciclo um novo exame das instruções será feito.

4.2. O POWER PC 603

Muito diferente do Pentium o PowerPC 603 foi desenvolvido visando uma estrutura mais horizontal com múltiplas unidades funcionais (vide [Eliseu94]). Este processador representa um marco na tecnologia de microprocessadores. O modelo estudado PowerPC 603 possui cinco unidades funcionais divididas da seguinte forma: uma unidade de ponto fixo **FXU**, uma unidade de acesso à memória **LSU**, uma unidade de ponto flutuante **FPU** e uma unidade de processamento de desvios **BPU.** Cada unidade funcional possui uma Estação de Reserva (*reservation station*) que armazenam as instruções que aguardam a liberação de recursos e não podem ser despachadas de imediato por possuírem alguma forma de dependência.

A maior novidade está na unidade de desvio **BPU** que é inteiramente independente da unidade de ponto fixo podendo realizar até cálculos de endereçamento. A associação com a unidade de busca de instruções **IU** fez com que as previsões dos desvios fossem realizadas ainda no ciclo de despacho reduzindo a zero o custo nas previsões corretas.

A integridade dos dados para previsões de desvio incorretas é garantida pelo mecanismo de renomeação de registradores que utiliza 12 registradores futuros para inteiros e 12 para ponto flutuante. A atualização dos registradores é feita por uma unidade responsável por finalizar as instruções utilizando um *Reorder Buffer*. O esquema apresentado na **figura 4.2** ilustra a estrutura *pipeline* do Power PC 603.

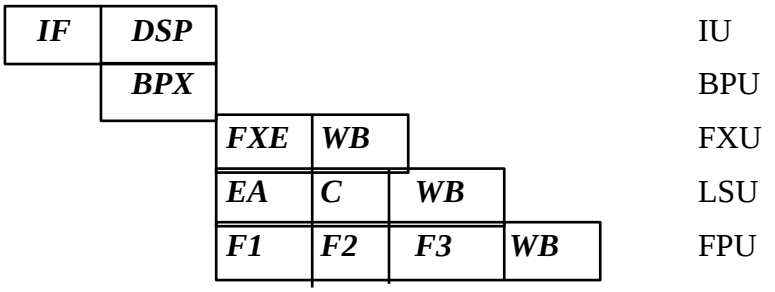


Figura 4.2: *Pipeline super escalar do Power PC 603.*

Algoritmo de despacho do Power PC 603

A cada ciclo o estágio de *prefetch* **IF** acessa duas instruções e as coloca em uma fila. O estágio **DSP** decodifica e acessa os operandos em registradores e posteriormente despacha a instrução a unidade funcional correspondente, até três instruções podem ser despachadas por ciclo. Neste momento instruções de desvios são detectadas e seus destinos previstos modificando o endereço de busca das próximas instruções. Se uma instrução condicional não puder ser processada, então é realizado uma previsão estática do desvio.

No estágio seguinte as instruções são decodificadas e despachadas as unidades funcionais apropriadas. O despacho é feito em ordem, com exceção de instruções de desvios que podem ser despachadas fora de ordem. Instruções com dependência de dados são despachadas as *reservation stations* e lá ficam a espera até que todos os seus operandos estejam disponíveis. Deste modo instruções subsequentes podem ser despachadas e executadas. O despacho de uma instrução é bloqueado quando a unidade funcional correspondente estiver ocupada e nenhuma das suas unidedes de reserva disponíveis.

Após a execução das instruções os resultados das operações são armazenados em um conjunto de registradores futuros e atualizados seguindo um modelo de *buffer* de reordenação que garante o integridade dos dados em situações de previsões incorretas.

4.3. ALPHA AXP 21064

A estrutura básica do Alpha AXP 21064 (vide [Eliseu94], [Alpha93] e [Smith94]) é composta por 32 registradores de inteiros de 64 *bits*, 32 registradores de ponto flutuante de 60 *bits*.

Este processador possui quatro unidades funcionais independentes a **ABOX** responsável por instruções de acesso à memória, **EBOX** responsável pela execução de instruções com inteiros, **FBOX** executa instruções de ponto flutuante e **IBOX** que cuida da busca, decodificação e despacho das instruções. O *pipeline* é composto por três caminhos paralelos para processamento de ponto fixo **EBOX**, ponto flutuante **FBOX** e acesso à memória **ABOX**. Sendo que os caminhos de ponto fixo e acesso à memória possuem o mesmo número de estágios.

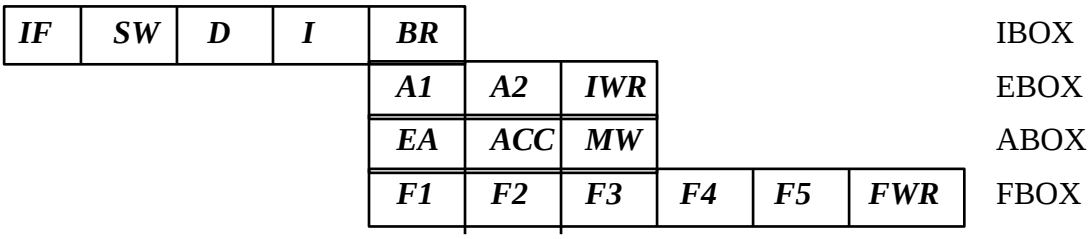


Figura 4.3: *Pipeline super escalar Alpha 21064.*

Conforme podemos observar na **figura 4.3**, o *pipeline* deste processador é extremamente vertical e se propaga por diversos estágios, 10 para instruções de ponto flutuante e 7 para instruções de acesso à memória e processamento de inteiros.

Algoritmo de despacho do Alpha AXP 21064

O estágio **IF** acessa duas instruções a cada ciclo. O estágio **SW** controla as restrições de despacho e se não for possível despachar duas instruções em um mesmo ciclo o despacho é sequenciado.

As dependência de dados são analisadas no estágio **I,** e existindo alguma dependência, o despacho é serializado ou bloqueado. O escalonamento de instruções será bloqueado se uma instrução de desvio alcançar o estágio **A1** e não puder ser executa imediatamente.

Capítulo 5

Aspectos gerais do simulador

5.1. A ARQUITETURA DLX

Nesta seção descreveremos um modelo simples de arquitetura *load/store* chamada DLX, apresentada inicialmente por **Patterson** (vide [Patter85]). Este modelo foi escolhido para representar a arquitetura básica do simulador **Dlxwin** por ser simples a sua implementação de fácil extensão a uma arquitetura super escalar.

A arquitetura DLX como a maioria das arquiteturas *load/store* enfatiza um conjunto de instruções simples, um *pipeline* eficiente, a decodificação fácil das instruções e o código objeto eficiente.

5.1.1. CARACTERÍSTICAS DA ARQUITETURA DLX

Nossa arquitetura base consiste de uma máquina com endereçamento de 32 *bits* e memória endereçada por *byte* no formato *big endian*. Todos os acessos à memória são feitos por instruções de *load/store* e trabalham diretamente sobre os registradores.

Existem dois bancos de registradores: um com trinta e dois registradores de ponto fixo de 32 *bits* denominados GPR (registradores de propósito geral), trinta e dois registradores de ponto flutuante de 32 *bits* chamados de FPR (*floating-point registers*), que podem ser utilizados como registradores de dupla precisão (64 *bits*), quando utilizados aos pares, e seis registradores de controle de 32 *bits* SR (*status registers*) responsáveis pelos mecanismos de controle do processador. Todas as instruções são alinhadas em 32 *bits*.

5.1.2. TIPOS DE OPERAÇÕES

No modelo DLX existem quatro classes de operações: instruções de acesso à memória, instruções aritméticas e lógicas, desvios condicionais e saltos, e instruções de ponto flutuante.

OPERAÇÕES DE ACESSO À MEMÓRIA

Qualquer registrador de propósito geral ou ponto flutuante pode ser carregado ou armazenado diretamente, exceto o registrador R[0] que possui o valor fixo zero. Todo o acesso à memória é endereçado por registrador e possui 16 *bits* de *offset*. As instruções de acesso à memória estão relacionadas na tabela da **figura 5.1**.

LB, LBU, SB	Carrega com sinal, sem sinal e armazena *byte*
LH, LHU, SH	Carrega com sinal, sem sinal e armazena *hword*
LW, SW	Carrega e armazena *word*
LF, LD, SF, SD	Carrega e armazena *float* e *double*
MOVI2S, MOVS2I	Move de/para GPR de/para SR
MOVF, MOVD	Copia valores entre registradores FPR
MOVFP2I, MOVI2FP	Move de/para FPR de/para GPR

Figura 5.1: *Instruções de transferencia de dados da arquitetura DLX.*

OPERAÇÕES ARITMÉTICAS E LÓGICAS

Todas as instruções aritméticas e lógicas com inteiros podem ser realizadas entre registradores ou entre registrador e valor imediato com sinal de 16 *bits*. A tabela da **figura 5.2** apresenta o conjunto de instruções aritméticas e lógicas da arquitetura DLX.

ADD, ADDI, ADDU, ADDUI	Adição, adição com imediato; com e sem sinal
SUB, SUBI, SUBU, SUBUI	Subtração, subtração com imediato; com e sem sinal
MULT, MULTI, MULTU, MULTUI	Multiplicação, multiplicação com imediato; com e sem sinal
DIV, DIVI, DIVU, DIVUI	Divisão, Divisão com imediato; com e sem sinal
AND, ANDI	E lógico, E lógico com imediato
OR, ORI	OU lógico, OU lógico com imediato
XOR, XORI	OU Exclusivo lógico, OU Exclusivo lógico com imediato
LHI	Carrega imediato na parte mais significativa do registrador
SLL, SLLI, SRL, SRLI	Shift Left e Shift Right com e sem imediato
SRA, SRAI	Arithmetic Shift Right com e sem imediato
SLT, SLTI, SGT, SGTI, SLE, SLEI, SGE, SGEI, SEQ, SEQI, SNE, SNEI	Operadores relacionais: <, >, <=, >=, =, !=

Figura 5.2: *Instruções aritméticas e lógicas da arquitetura DLX.*

OPERAÇÕES DE DESVIOS CONDICIONAIS E SALTOS

Os desvios condicionais testam o valor do registrador fonte quanto a ser igual ou diferente de zero. Endereços de desvios são indicados como deslocamento de 16 *bits* adicionado ao *Program Counter*. A tabela da **figura 5.3** apresenta as instruções de desvios da arquitetura DLX.

BEQZ, BNEZ	Desvio condicional; GPR igual/diferente de zero
BFPT, BFPF	Desvio condicional; FP *status register* igual/diferente de zero
J, JR	Salto; deslocamento de 26 bits
JAL, JALR	Salto a subrotina; R31<- PC+4 e PC <- offset
TRAP	Chamada rotina do Sistema Operacional
RFE	Retorno de uma exceção

Figura 5.3: *Instruções de saltos e desvios da arquitetura DLX.*

OPERAÇÕES DE PONTO FLUTUANTE

As instruções de ponto flutuante manipulam os registradores FPR com simples e dupla precisão. Ao utilizar dupla precisão os registradores são manipulados aos pares. Todas as operações relacionais afetam o *status register* FP que recebe apenas dois valores: zero (falso) e um (verdadeiro). A tabela de instruções de ponto flutuante pode ser observada na **figura 5.4**.

ADDD, ADDF	Adição registradores *double/float*
SUBD, SUBF	Subtração registradores *double/float*
MULTD, MULTF	Multiplicação registradores *double/float*
DIVD, DIVF	Divisão registradores *double/float*
CVTF2D, CVTF2I	Conversão de dados *float* para *double/integer*
CVTI2D, CVTI2F	Conversão de dados *integer* para *double/float*
CVTD2I, CVTD2F	Conversão de dados *double* para integer/*float*
LTD, LTF, GTD, GTF, LED, LEF, GED, GEF, EQD, EQF, NED, NEF	Operações relacionais: <, >, <=, >=, =, !=

Figura 5.4: *Instruções de ponto flutuante da arquitetura DLX.*

5.1.3. FORMATO DAS INSTRUÇÕES

Todas as instruções são de 32 *bits* com 6 *bits* identificando o código da operação. Existem três formatos de instruções: o formato I-Type apresentado na **figura 5.5a**, é utilizado em todas as instruções que usam valor imediato, R-Type, visto na **figura 5.5b**, usado por instruções que utilizam operandos registradores e J-Type para operações de salto com deslocamento de 26 *bits*, apresentado na **figura 5.5c**.

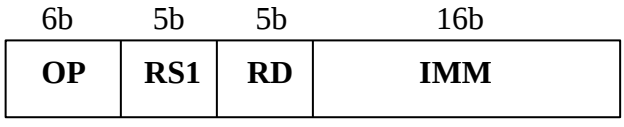


Figura 5.5a: *Formato das instruções do tipo I-Type.*

Figura 5.5b: *Formato das instruções do tipo R-Type.*

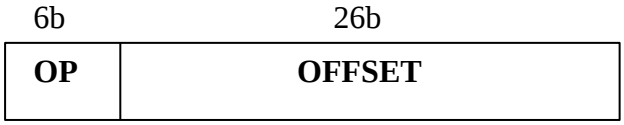


Figura 5.5c: *Formato das instruções do tipo J-Type.*

5.1.4. ESTÁGIOS DE EXECUÇÃO E *PIPELINE*

Toda instrução do DLX é executada em cinco estágios: *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execution and Effective Address Calculation* (EX), *Memory Access* (MEM) e *Write Back* (WB). As instruções usam o mesmo caminho de dados inclusive instruções de ponto flutuante.

A estrutura do DLX possui quatro unidades funcionais de execução: *integer unit, FP/integer multiply, FP adder, FP/integer divide*. A **figura 5.6** apresenta o *pipeline* empregado na arquitetura DLX.

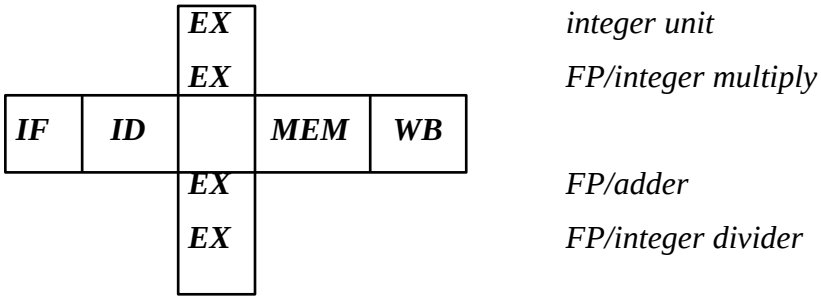


Figura 5.6: *Pipeline super escalar do DLX.*

5.2. O MODELO IMPLEMENTADO

Conforme pode ser visto, o modelo DLX está muito bem definido nos moldes de um processador escalar. Na verdade o modelo DLX, como é definido, representa uma arquitetura escalar utilizando microprograma, o que não satisfaz o modelo necessário a implementação, portanto definiremos a seguir as características do simulador implementado e apresentaremos suas diferenças básicas com a arquitetura DLX.

5.2.1. CARACTERÍSTICAS DO SIMULADOR DLXWIN

O simulador implementado possui endereçamento de 32 *bits* com memória acessada por *byte* no formato *litle endian*, diferente da arquitetura DLX que usa o formato *big endian*. Esta modificação foi motivada por facilidades de implementação do simulador em um computador utilizando processadores da família Intel 80x86.

É uma arquitetura *load/store* com trinta e dois registradores de propósito geral de 32 *bits* e 32 registradores de ponto flutuante de 64 *bits* que podem ser utilizados como registradores de 32 *bits*. A arquitetura DLX utiliza registradores de ponto flutuante de 32 *bits* que agrupado em pares formam 16 registradores de 64 *bits*. Esta modificação foi motivada por facilidades de implementação.

As instruções são de 32 *bits* alinhadas e de formato similar as implementadas no processador DLX original apresentado na seção anterior. Na intenção de construir um modelo próximo da realidade, foram incluídos vários recursos existentes em máquinas super escalares no simulador: busca antecipada de instruções, fila de instruções, previsão antecipada de desvios, *Branch Target Buffer*, execução especulativa de desvios, escalonamento dinâmico de instruções, registradores futuros, unidades virtuais, múltiplos conjuntos de registradores de condição e *buffer* de reordenação.

5.2.2. ESTÁGIOS DE EXECUÇÃO E PIPELINE

As instruções do simulador são executadas em seis estágios: *Instruction Fetch* (IF), *Instruction Decode* (ID), *Dispatch* (DISP), *Execution* (EX), *Memory Access* (MEM) e *Write Back* (WB).

Todas as instruções possuem o mesmo ciclo de latência e usam caminhos de dados equivalentes, inclusive as instruções de ponto flutuante que nas arquiteturas estudadas apresentavam um caminho de dados próprio e tempo de latência maior.

Diferente da arquitetura DLX, foram implementadas quatro unidades funcionais. Uma para processamento aritmético e lógico (ALU), uma unidade para processamento de desvios (BPU), uma unidade de ponto flutuante (FPU) e uma unidade de operação de transferência de dados (LSU). Todas as unidades funcionais possuem estações de reserva.

O *pipeline* super escalar do simulador **Dlxwin** é composto por seis estágios, conforme pode ser observado na **figura 5.7**, e não por cinco como ocorre na arquitetura DLX. Esta modificação foi realizada para permitir a implementação de um mecanismo de execução expeculativa de desvios.

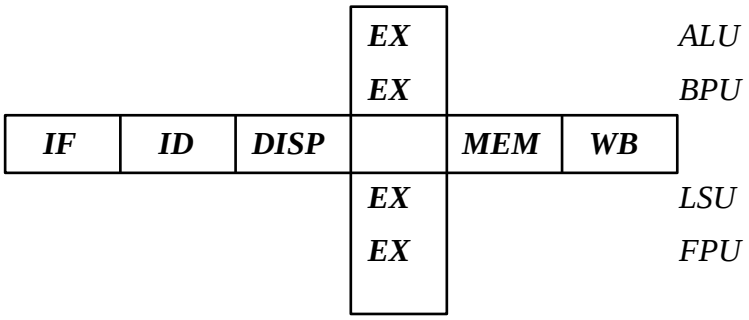


Figura 5.7: *Pipeline super escalar do simulador* ***Dlxwin***

5.2.3. RECURSOS IMPLEMENTADOS

Os recursos implementados foram baseados no estudo de várias arquiteturas existentes no mercado. O simulador foi desenvolvido procurando ampliar ao máximo a quantidade de mecanismos implementados de modo a permitir o uso dos algoritmos em análise sem necessidade de modificações.

Todos os parâmetros definidos no simulador como por exemplo o número de entradas na fila de instruções e o número de unidades virtuais por unidade de execução, foram atribuídos com base nos estudos de vários modelos reais.

Busca antecipada de instruções

O mecanismo de busca antecipada de instruções lê até quatro instruções por ciclo e insere em uma fila de instruções. Este mecanismo é bloqueado quando a fila de instruções estiver cheia.

A previsão de desvio é feita durante a fase de busca das instruções, utilizando o mecanismo do *Branch Target Buffer*, e para instruções de desvio que não se encontram na BTB é realizada uma previsão estática considerando o desvio como não-tomado.

Fila de instruções

A fila de instruções possui oito entradas que armazenam o endereço e o código das instruções acessadas pelo mecanismo de busca.

Previsão antecipada de desvios

A previsão antecipada de desvios é realizada através da técnica do *Branch Target Buffer* (BTB) utilizando uma tabela com 16 *slots* de 8 entradas e dois *bits* de previsão. O algoritmo implementado no BTB é uma adaptação do algoritmo LRU. Quando a instrução de desvio não possui uma entrada na tabela é realizada a previsão estática assumindo o desvio como não-tomado.

Execução especulativa de desvios

Até duas instruções de desvio podem ser executadas especulativamente. A integridade dos dados quanto a previsão incorreta de desvios é garantida pelos mecanismos de *buffer* de reordenação e registradores futuros, sendo o controle realizado na fila de despacho com a adição de *bits* identificadores do nível de execução especulativa.

Escalonamento dinâmico de instruções

Os algoritmos implementados no simulador **Dlxwin** (Pentium, PowerPC 603 e Alpha AXP 21064), se caracterizam pelo escalonamento dinâmico das instruções.

Registradores Futuros

Cada registrador de propósito geral (GPR), ponto flutuante (FPR) e de *status* (SR) possui um registrador futuro associado que garante as unidades de reserva que aguardam dados a disponibilidade dos mesmos com pelo menos um ciclo de antecedência, além de tornarem possível a execução especulativa de desvios.

Unidades Virtuais

As unidades virtuais armazenam as instruções que estão a espera de recursos. O modelo foi implementado com três unidades virtuais para cada unidade real.

Múltiplos conjuntos de registradores de condição

Toda instrução condicional que opera sobre números inteiros pode armazenar os resultados em qualquer um dos 32 registradores de propósito geral.

Reorder Buffer

O *buffer* de reordenação implementado possui oito entradas que garantem a atualização apropriada dos registradores de propósito geral (GPR), dos registradores de ponto flutuante (FPR) e dos registradores de *status* (SR).

5.3. O SIMULADOR DLXWIN

O simulador desenvolvido foi escrito em C++, em uma plataforma Pentium executando o sistema operacional Microsoft Windows. As classes foram projetadas de modo a representar os diversos mecanismos de um computador super escalar, conforme pode ser visto na **figura 5.8**.

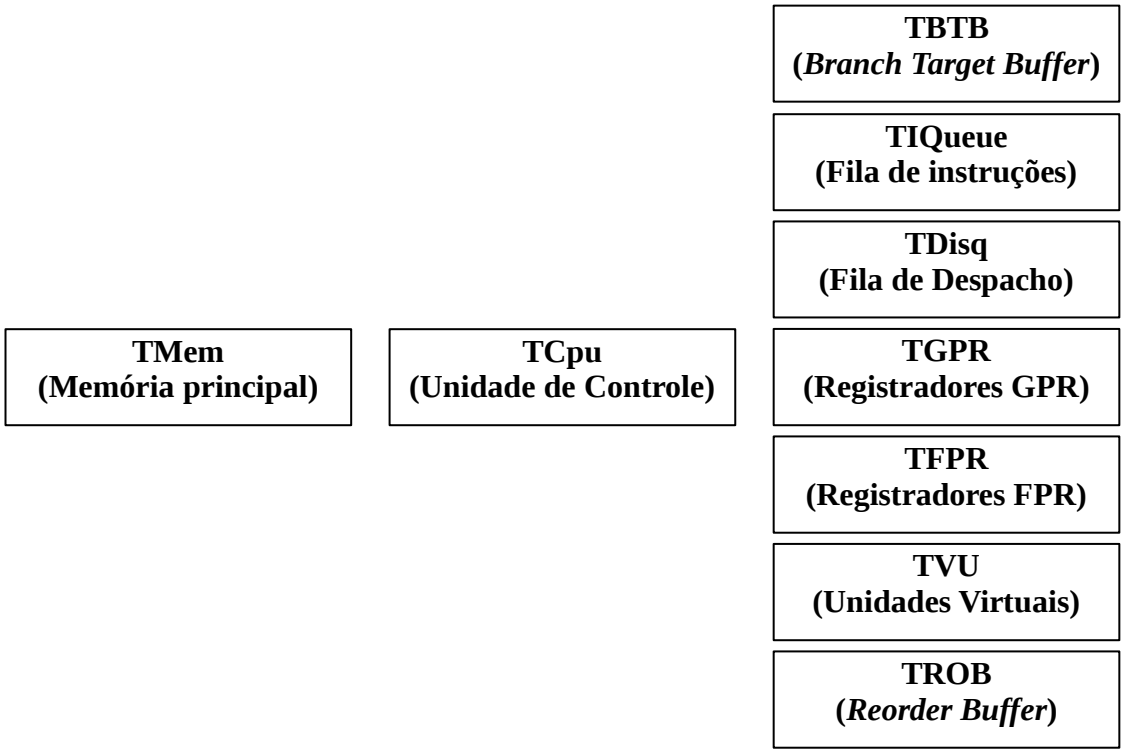


Figura 5.8: *Apresentação das classes implementadas no simulador.*

A classe **TMem** é responsável por simular a memória principal. **TCpu** é a classe que representa a unidade de processamento central, sendo responsável pela integração dos outros componentes e pelo fluxo de dados do processador. **TBTB** é a classe que simula a tabela de destinos de desvios (*Branch Target Buffer*). As classes **TIQueue** e **TDisq** representam respectivamente as filas de instrução e despacho. **TGPR** e **TFPR** são respectivamente os bancos de registradores de propósito geral e ponto flutuante. As unidades virtuais são caracterizadas pela classe **TVU**. **TROB** representa o *buffer* de reordenação para a máquina **Dlxwin**.

5.3.1. FLUXO DE DADOS DO SIMULADOR DLXWIN

O fluxo dos dados e o controle do processamento é realizado pela classe **TCpu** e pode ser observado no diagrama da **figura 5.9**.

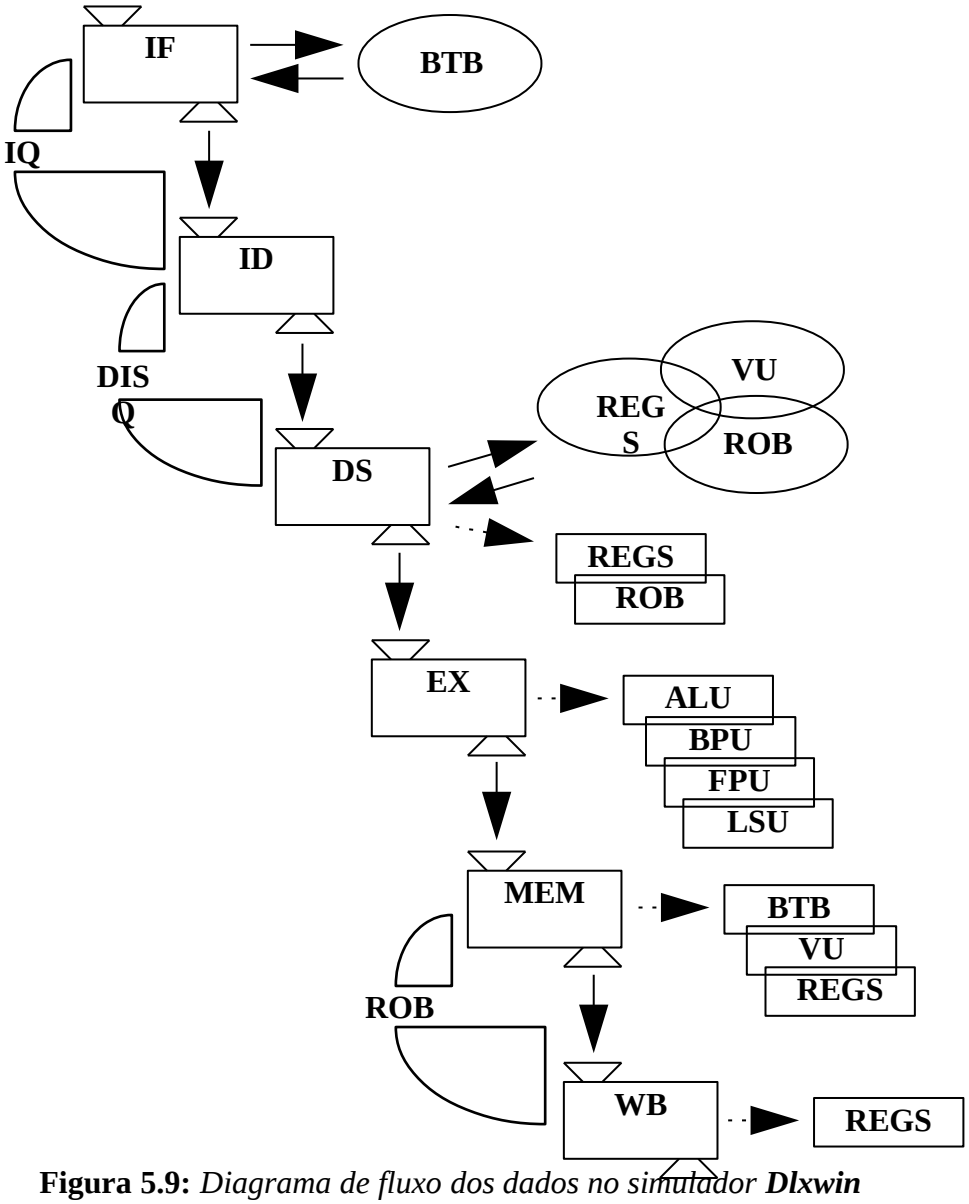


Figura 5.9: *Diagrama de fluxo dos dados no simulador* ***Dlxwin***

A cada ciclo o estágio de busca de instruções (**IF**) acessa até quatro instruções que são colocadas na fila de instruções (**IQ**). Para cada instrução de desvio acessada é realizado uma consulta a tabela de destinos de desvios (**BTB**), não importando se a instrução é de desvio condicional ou incondicional. Se a instrução não for encontrada na tabela é realizado uma previsão estática considerando que o desvio não ocorrerá. O mecanismo de busca é interrompido quando a fila de instruções fica sem entradas vazias ou quando existirem duas instruções de desvio em processamento, e uma terceira for encontrada.

O estágio seguinte (**ID**) retira até duas instruções da fila (**IQ**) para decodificação a cada ciclo, e as coloca na fila de despacho (**DISQ**). O processo de decodificação é interrompido quando não existem instruções para serem retiradas da fila de instruções ou quando a fila de despacho estiver cheia.

No estágio de despacho (**DS**), podemos ter um dos quatro algoritmos implementados em atividade: o algoritmo Escalar, do Pentium, do PowerPC 603 ou do Alpha AXP 21064. Estes algoritmos decidem, com base no estado dos registradores (**REGS**), unidades virtuais (**VU**) e *buffer* de reordenação (**ROB**), quantas instruções poderão ser despachadas no ciclo. Toda instrução despachada é colocada em uma unidade virtual correspondente, então uma nova entrada no *buffer* de reordenação é inserida e o estado do registrador destino atualizado. Este processo é interrompido quando a fila de despacho estiver vazia, quando o *buffer* de reordenação estiver preenchido, quando não houverem unidades virtuais disponíveis ou quando o algoritmo implementado solicitar o bloqueio.

O estágio de execução (**EX**) analisa as unidades virtuais e retira para processamento as instruções que estiverem prontas. Neste estágio as instruções são executadas mas seus resultados ainda não são obtidos. As instruções de acesso à memória se preparam para a leitura ou gravação colocando o endereço alvo no registrador de acesso à memória (**MAR**).

As instruções são finalizadas e os acessos a memória concluídos no estágio (**MEM**). Este estágio atualiza a tabela de destinos de desvios, as unidades virtuais que esperam liberação de operandos, os registradores futuros e o *buffer* de reordenação.

O último estágio (**WB**) é responsável por retirar as instruções concluídas do *buffer* de reordenação e atualizar os registradores reais.

5.3.2. APRESENTAÇÃO DO SIMULADOR DLXWIN

O simulador foi desenvolvido de modo a permitir uma visão de todo o processador. A opção pelo uso de janelas permitiu a visualização detalhada de cada mecanismo implementado. O processamento dos programas pode ser passo-a-passo, para um determinado número de ciclos, ou do inicio ao fim do programa. Isto permite uma análise detalhada dos resultados. No modo passo-a-passo podemos observar o fluxo de dados do simulador para cada algoritmo de despacho selecionado. A troca entre os algoritmos de despacho pode ser feita sem a necessidade de sair do simulador.

Os programas usados no experimento podem ser carregados de dentro do ambiente e executados várias vezes, o que permite uma maior facilidade na análise de cada experimento.

Foram criadas ao todo treze janelas que permitem uma visão total do simulador durante a operação. As janelas criadas correspondem à: *Program*, *Memory*, *Regs*, *GPR*, *FPR*, *Instruction Queue*, *Dispatch Queue*, *Virtual Units*, *Exec Units*, *Reorder Buffer*, *Branch Target Buffer*, *Exec Stream* e *Dispatch Stream*.

Program

A janela programa permite uma visão do programa carregado na memória.

Memory

Esta janela reproduz o conteúdo da memória principal.

Regs

Permite a visualização dos registradores de controle e dos parâmetros medidos.

GPR

Através da janela GPR é possível visualizar o banco de registradores de propósito geral, observar os registradores que estão ocupados a espera da conclusão de uma instrução e ver as modificações efetuadas nos registradores reais e futuros.

FPR

Nesta janela pode ser visto o banco de registradores de ponto flutuante com os registradores reais e futuros. Também é possível observar os registradores que estão marcados como ocupados a espera de resultados.

Instruction Queue

A janela fila de instruções permite a observação do mecanismo de busca.

Dispatch Queue

A visualização da fila de despacho permite analisar o mecanismo de despacho e a execução especulativa de desvios.

Virtual Units

Esta janela permite a observação das instruções nas unidades virtuais, podendo ser vista quais instruções estão prontas para execução e quais estão a espera de operandos.

Exec Units

A janela de execução mostra uma visão das instruções que passaram pelo estágio de execução mas ainda não foram concluídas, isto é, instruções que estão no estágio (**MEM**). A observação desta janela e das janelas *Regs, GPR* e *FPR*, permite a compreensão da execução do programa.

Reorder Buffer

Nesta janela, pode ser visto o *buffer* de reordenação onde observamos a ordem de atualização dos registradores e o fluxo de execução do programa. Também podemos observar as instruções que estão prontas para serem concluídas.

Branch Target Buffer

Esta janela permite uma visão do funcionamento da tabela de destinos de desvios.

Exec Stream

A janela *Exec Stream* é uma das principais janelas para depuração e compreensão dos programas de teste. Nesta janela podemos visualizar o fluxo de execução das instruções com a identificação da instrução e do ciclo no qual ela foi executada. Esta janela armazena os últimos quatrocentos ciclos de execução de um programa, permitindo uma análise bastante extensa. Nesta janela também pode ser observado a ocorrência de execuções em paralelo.

Dispatch Stream

Nesta janela podemos visualizar os últimos quatrocentos ciclos do fluxo de despacho de um programa e desta forma analisar o funcionamento dos algoritmos de despacho implementados.

5.3.3. LIMITAÇÕES DO SIMULADOR DLXWIN

No simulador não existe um mecanismo de sequenciamento dos acessos a memória, normalmente implementado por meio de uma fila, assim todas as instruções de acesso à memória são executadas em ordem, e fica de responsabilidade do programador evitar que instruções de escrita na memória sejam inseridas imediatamente após instruções de desvios. Então, para simplificar o trabalho do programador o número de execuções especulativas foi limitado em duas, sendo que nos programas que não fazem acesso à memória este número pode ser ampliado, o que modifica muito o desempenho dos algoritmos.

5.4. AMBIENTE DE COMPILAÇÃO E DEPURAÇÃO

Para desenvolvimento dos programas de teste foi construído um *assembler* capaz de ler um arquivo fonte *ascii* e produzir dois arquivos de saída, um arquivo binário usado pelo simulador e um arquivo de listagem com a representação hexadecimal e dos mnemônicos das instruções. O arquivo de listagem foi fundamental para encontrar erros de compilação nos programas.

Os programas de teste criados foram depurados no simulador usando o algoritmo Escalar e posteriormente os outros algoritmos. A depuração realizada com o algoritmo Escalar tinha como objetivo evitar que os problemas de controle de acesso à memória prejudicassem o desenvolvimento do programa de teste.

No próximo capítulo analisaremos os programas usados no experimento e os resultados obtidos.

Capítulo 6

Análise dos Resultados

Nos capítulos anteriores foram apresentados os diversos aspectos da arquitetura *super escalar.* Neste capítulo será visto como foram implementados os experimentos e será feita uma análise sobre os resultados obtidos.

6.1. AMBIENTE DE EXPERIMENTO

O simulador **Dlxwin** possui vários recursos utilizados nas arquiteturas super escalares atuais. Entre os recursos implementados podemos citar: janela de instruções, previsão antecipada de desvios, *buffer* de reordenação, registradores futuros, execução especulativa de desvios e unidades virtuais.

A implementação de uma arquitetura sofisticada como ambiente base do simulador forneceu a flexibilidade desejada para a implementação dos algoritmos de despacho analisados atendendo a todas as solicitações impostas por eles. Esta flexibilidade favoreceu também a imparcialidade dos experimentos, visto que apenas o mecanismo de despacho de instruções foi modificado para a implementação de cada algoritmo.

Apesar do alto grau de sofisticação do simulador **Dlxwin**, existe uma limitação de projeto motivada pelo fato deste não possuir um mecanismo de sequenciamento de escrita na memória. Assim após uma instrução de desvio condicional não poderá existir uma instrução de escrita na memória. Este problema pode ser contornado introduzindo instruções de *no-operation* entre o desvio condicional e a instrução de escrita na memória.

6.2. PROGRAMAS DE TESTES

Foram escritos para avaliação dos algoritmos de despacho dez pequenos programas de teste. Sete são de características distintas para medição do desempenho real dos algoritmos de despacho e três são versões otimizadas de um mesmo programa para avaliar o ganho de desempenho dos algoritmos com a otimização do código.

Os programas de teste foram escritos visando a obtenção de códigos próximos aos que seriam produzidos por compiladores de máquinas reais. Assim diversas características encontradas nos códigos gerados por compiladores convencionais foram reproduzidas nos programas de teste.

A tabela da **figura 6.1** apresenta a relação dos programas de testes implementados.

PROGRAMA	DESCRIÇÃO
Array	Cria um *array* de 100 elementos do tipo *byte* aleatórios;
Copymem	Copia uma seqüência de 32 elementos do tipo *word* entre posições de memória;
Copymem2	Primeiro modelo otimizado do programa Copymem;
Copymem3	Segundo modelo otimizado do programa Copymem;
Forwhil2	Executa dois *loops* aninhados ;
Forwhile	Executa um *loop* simples;
Polar	Transforma um *array* de vetores em coordenadas polares em um *array* de vetores em coordenadas cartesianas;
Rand	Retorna um número aleatório (obs: este programa não executa desvios);
Sincos	Calcula o seno e o coseno de um *array* de 7 elementos;
Sort	Ordena um *array* de 10 elementos aleatórios;

Figura 6.1: *Relação dos programas de teste implementados.*

6.3. MÉTODOS DE MEDIÇÃO

Foram três os critérios utilizados na medição do desempenho dos algoritmos: (i) o tempo de execução do programa medido em ciclos, (ii) a razão entre o tempo de processamento do programa sem qualquer forma de paralelismo (tempo seqüencial) e o tempo de processamento quando aplicado o algoritmo de despacho selecionado (tempo paralelo). Esta razão possui o nome de *speedup* e representa o quanto o processamento paralelo de um programa é mais rápido do que o processamento escalar.

speedup = tempo_sequencial / tempo_paralelo

Outro critério de medição empregado foi (iii) a taxa de ocupação das unidades funcionais. Neste critério foi medido a proporção do tempo de processamento em que as unidades de execução ficam ocupadas. Esta taxa representa o grau de utilização das unidades funcionais produzido pelo algoritmo selecionado.

6.4. ANÁLISE DOS RESULTADOS

Os experimentos foram efetuados executando cada programa com todos os algoritmo de despacho. Para cada algoritmo selecionado, o ambiente do simulador era reiniciado e o programa em questão recarregado de modo a criar estados iniciais idênticos para cada maquina em teste. A seguir serão apresentados os programas usados no experimento e os resultados obtidos.

Array

O programa **Array** constrói um vetor de 100 elementos aleatoriamente. Neste programa as instruções de desvio representam 8% das instruções executadas pelo processador. O baixo número de instruções de desvio pode ser observado na listagem do programa apresentada na **figura 6.2**.

Devido ao baixo número de instruções de desvios executadas os valores medidos para a taxa de ocupação das unidades de execução e *speedup* apresentaram valores acima da média dos experimentos para todos os algoritmos, conforme pode ser visto na tabela da **figura 6.3** e nos gráficos da **figura 6.4**.

```
00000000  401E1000          ADDI  R30,R0,1000 ;; @INICIO: R30 <- @STACK
00000004  FC1EB240          ADD   R22,R0,R30
00000008  FD084246          XOR   R8,R8,R8
0000000C  E8000010          J     10
00000010  EC000028          JAL   28          ;; @1@10:
00000014  41080001          ADDI  R8,R8,1
00000018  191C00BF          SB    BF[R8],R28  ;;    @C0-1[R8] <- R28
0000001C  6D170064          SLTI  R23,R8,64   ;; @1@20:
00000020  02E0FFF0          BEQZ  R23,FFF0    ;; se R8 < 100 goto @1@10
00000024  F8000000          HALT
                            :
00000038  47DE0008          SUBI  R30,R30,8   ;; @RAND:
0000003C  2FDF0000          SW    0[R30],R31
00000040  2FD60004          SW    4[R30],R22
00000044  47DE0014          SUBI  R30,R30,14
00000048  2FC80000          SW    0[R30],R8
0000004C  3FC80004          SD    4[R30],F8
00000050  3FC9000C          SD    C[R30],F9
00000054  47DE0004          SUBI  R30,R30,4
00000058  FC1EB240          ADD   R22,R0,R30
0000005C  300800BC          LF    F8,BC[R0]   ;; F8 <- @SEED
00000060  400803E5          ADDI  R8,R0,3E5   ;; R8 <- 997
00000064  2EC80000          SW    0[R22],R8
00000068  32C90000          LF    F9,0[R22]
0000006C  FD204C0C          CVTI2F F9,F9
00000070  FD094405          MULTF F8,F8,F9
00000074  3AC80000          SF    0[R22],F8
00000078  2AC80000          LW    R8,0[R22]
0000007C  510800FF          ANDI  R8,R8,0FF
00000080  2C0800BC          SW    BC[R0],R8   ;; @SEED <- R8
00000084  FC08E240          ADD   R28,R0,R8
00000088  43DE0004          ADDI  R30,R30,4
0000008C  37C9000C          LD    F9,C[R30]
00000090  37C80004          LD    F8,4[R30]
00000094  2BC80000          LW    R8,0[R30]
00000098  43DE0014          ADDI  R30,R30,14
0000009C  2BD60004          LW    R22,4[R30]
000000A0  2BDF0000          LW    R31,0[R30]
000000A4  43DE0008          ADDI  R30,R30,8
000000A8  0BE00000          JR    R31
                            :
000000BC  3F07464B          DF    0.528416312342    ;; @SEED:
000000C0  00000000          DW    0                 ;; @ARRAY:
```

Figura 6.2: *Listagem do código do programa* ***Array.***

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	4738	4,315389	71,93
Pentium	3029	6,750991	89,34
PowerPC 603	2532	8,076650	91,19
Alpha 21064	2636	7,757875	95,26

Figura 6.3: *Tabela de resultados do programa* ***Array****.*

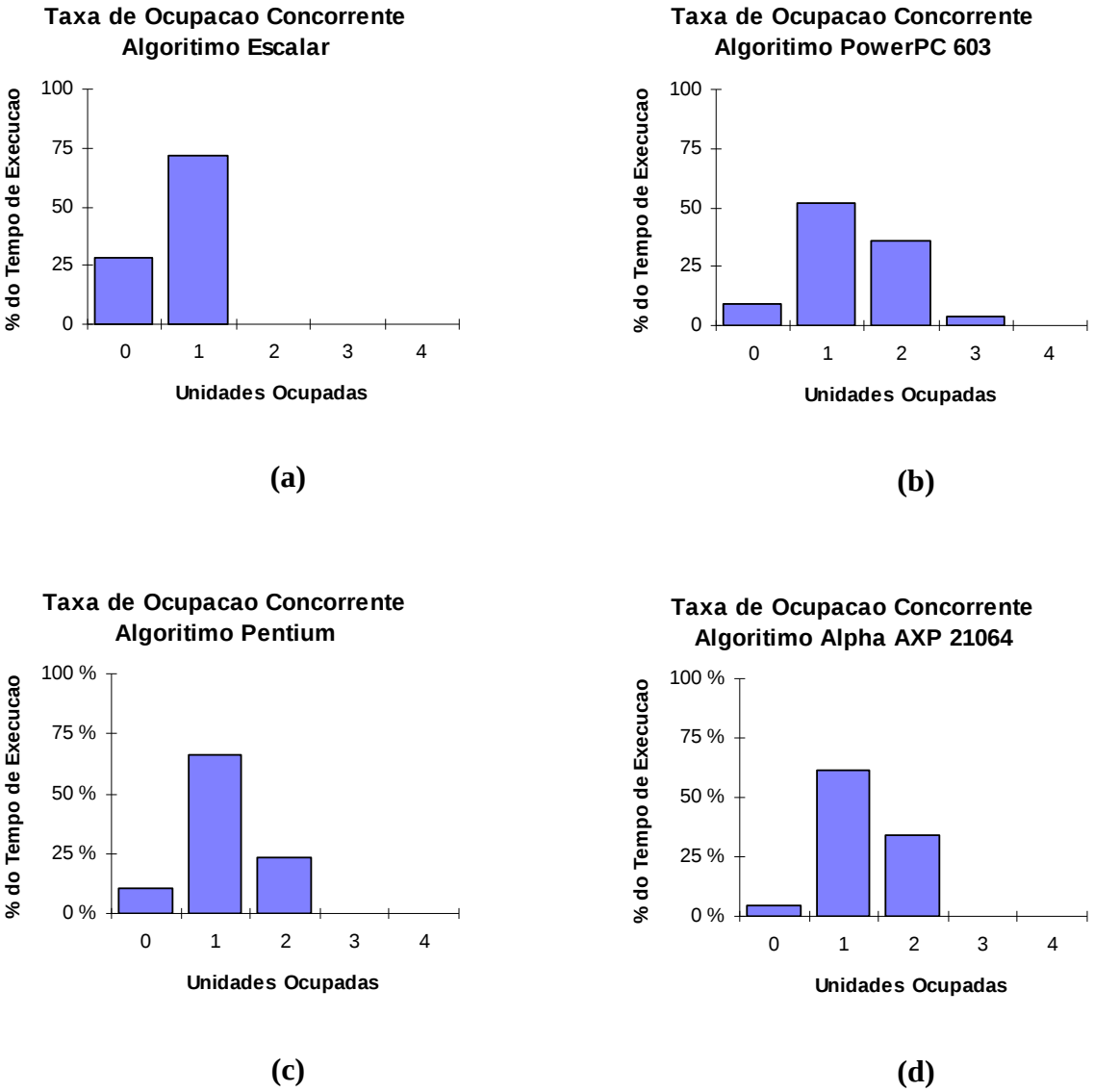


Figura 6.4: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Array****.*

Copymem

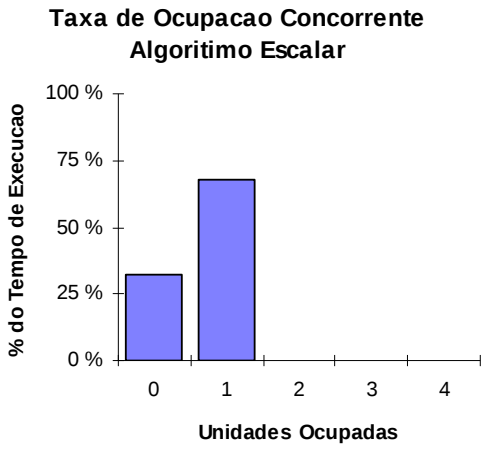
Este programa copia um bloco de 32 palavras de 32 *bits* de uma posição da memória para outra. Por ser um programa baseado em desvios, onde 25% das instruções executadas representam instruções deste tipo (ver listagem na **figura 6.5**), apresentou uma taxa de ocupação das unidades de execução inferior a encontrada no programa **Array**, conforme pode ser observado na tabela da **figura 6.6** e nos gráficos da **figura 6.7**.

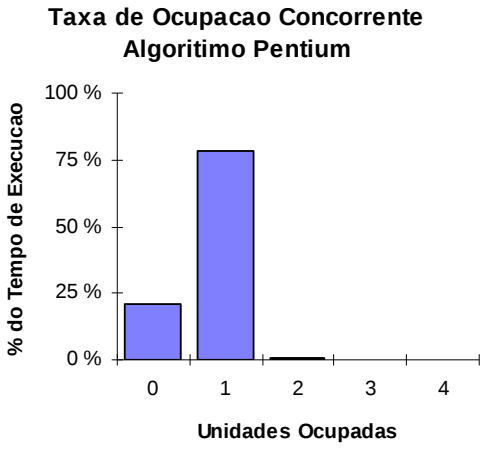
```
00000000  40010000   ADDI  R1,R0,0          ;; @INICIO:
00000004  40020020   ADDI  R2,R0,20
00000008  28230000   LW    R3,0[R1]         ;; @1@10:
0000000C  2C230040   SW    40[R1],R3
00000010  40210004   ADDI  R1,R1,4
00000014  44420001   SUBI  R2,R2,1
00000018  0440FFF0   BNEZ  R2,FFF0          ;; se R2 != 0 goto @1@10
0000001C  F8000000   HALT
```

Figura 6.5: *Listagem do código do programa* ***Copymem.***

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	241	4,075000	68,05
Pentium	205	4,794118	79,02
PowerPC 603	100	9,878788	87,00
Alpha 21064	118	8,358974	88,98

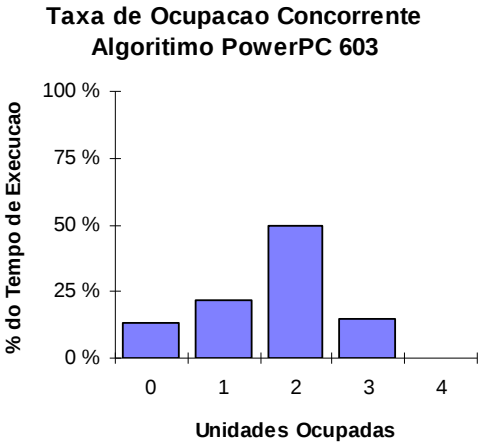
Figura 6.6: *Tabela de resultados do programa* ***Copymem****.*

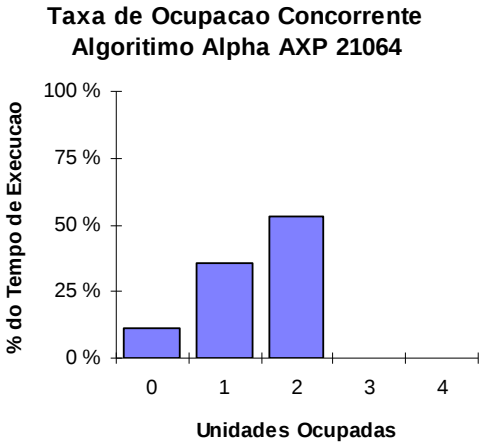




(a)

(b)





(c) **(d)**

Figura 6.7: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Copymem****.*

Copymem2

O programa **Copymem2**, como pode ser visto na **figura 6.8**, é uma variação do programa **Copymem** apresentado no item anterior. Este programa e o programa **Copymem3** foram escritos com o intuito de mostrar o ganho de desempenho que pode ser obtido com pequenas modificações no código. Como pode ser observado, comparando o código dos dois programas, a única alteração realizada foi o movimento da instrução de decremento do contador para uma outra posição afim de evitar a dependência de dados com o comando de desvio condicional.

Pode ser observado comparando os valores das tabelas apresentadas nas **figuras 6.6** e **6.9** que a modificação melhorou o desempenho de todos os algoritmos analisados, sendo que os algoritmos Escalar e Pentium obtiveram os maiores ganhos com a otimização do código. Os resultados mostram que as arquiteturas que possuem um algoritmo de escalonamento menos restritivo sofrem menos influencia com a otimização do código de um programa do que as arquiteturas que utilizam apenas escalonamento estático, e que o grau de influência é proporcional as restrições impostas pelo algoritmo de despacho.

A comparação do gráfico apresentado na **figura 6.10c** com o da **figura 6.7b** mostra que o algoritmo do Pentium aumentou sua taxa de ocupação concorrente com duas unidades de execução ativas de aproximadamente 0 para 75% do tempo de execução do programa. Este resultado confere com o ganho de desempenho obtido com a otimização do código nos experimentos com o algoritmo do Pentium, um algoritmo que oferece muitas restrições ao despacho paralelo das instruções.

O algoritmo do Alpha AXP 21064 apresentou pouca variação nos resultados apresentados nos gráficos da **figura 6.10d** e **figura 6.7d**, o que justifica a pequena variação de desempenho encontrada na comparado dos resultado de **Copymem2** e **Copymem**.

Já o PowerPC 603 compensou a perda na taxa de ocupação concorrente de três unidades de execução pelo aumento desta taxa na ocupação de duas unidades, conforme pode ser visto nas **figuras 6.7c** e **6.10b**, produzindo valores de desempenho semelhantes em **Copymem2** e **Copymem.**

```
00000000  40010000   ADDI  R1,R0,0          ;; @INICIO:
00000004  40020020   ADDI  R2,R0,20
00000008  28230000   LW    R3,0[R1]         ;; @1@10:
0000000C  44420001   SUBI  R2,R2,1
00000010  2C230040   SW    40[R1],R3
00000014  40210004   ADDI  R1,R1,4
00000018  0440FFF0   BNEZ  R2,FFF0          ;; se R2 != 0 goto @1@10
0000001C  F8000000   HALT
```

Figura 6.8: *Listagem do código do programa* ***Copymem2***.

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	177	5,556818	92,66
Pentium	141	6,985714	69,50
PowerPC 603	97	10,187500	86,60
Alpha 21064	112	8,810811	88,39

Figura 6.9: *Tabela de resultados do programa* ***Copymem2****.*

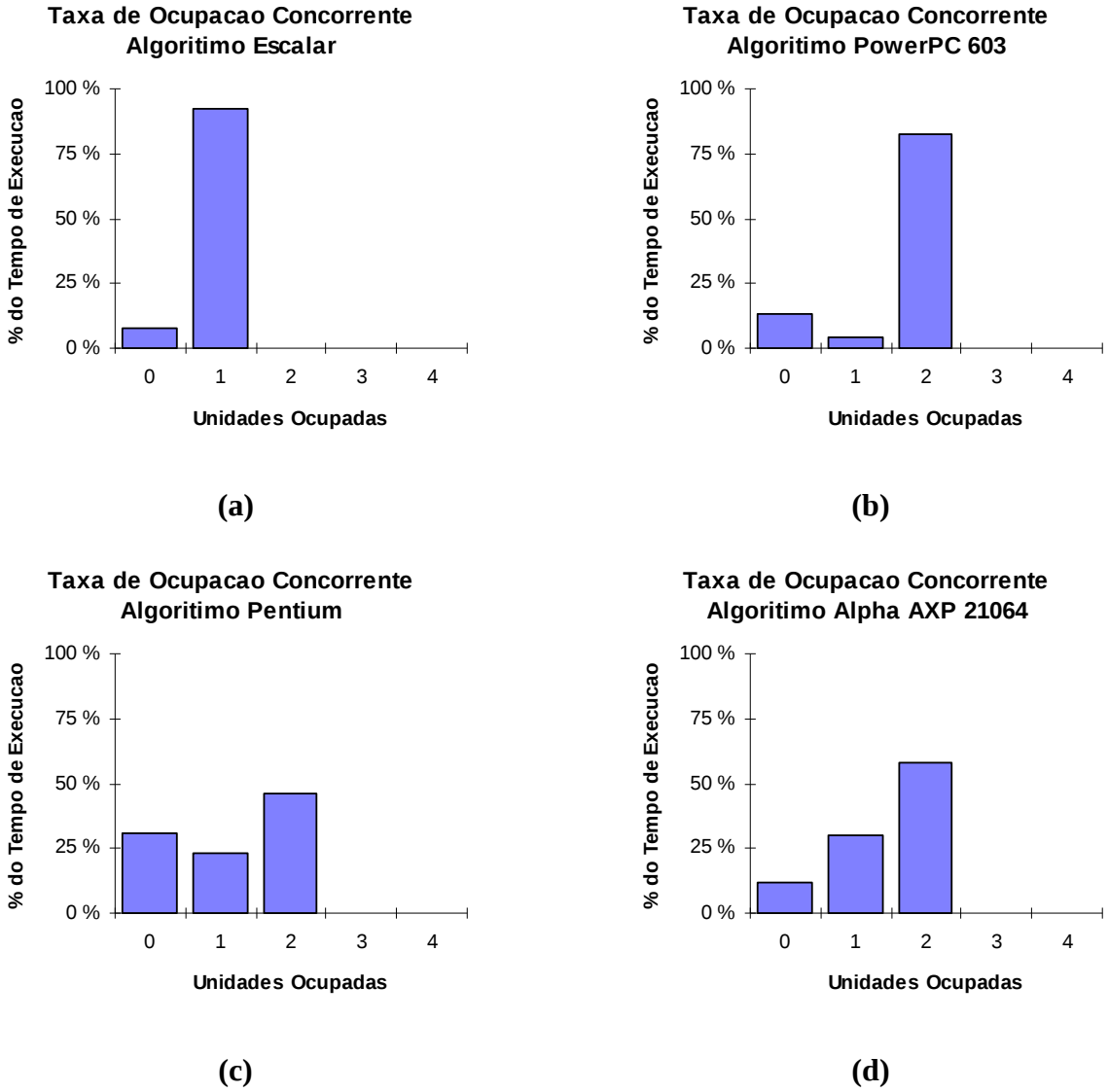


Figura 6.10: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Copymem2****.*

Copymem3

O programa **Copymem3**, apresentado na **figura 6.11**, é mais uma variação do programa **Copymem**. Nele as instruções de acesso a memória que usam o registrador **R1** como fonte foram afastadas do comando de incremento deste registrador, afim de eliminar a dependência de dados.

Esta modificação não apresentou o resultado desejado, conforme pode ser observado na comparação dos valores tabelados nas **figura 6.9** e **6.12**. Os algoritmos do Pentium, PowerPC e Alpha apresentaram resultados equivalentes aos obtidos no experimento do programa **Copymem2**. Isto porque nenhum deles possui restrições quanto a dependências do tipo anti-dependência.

```
00000000  40010000   ADDI  R1,R0,0          ;; @INICIO:
00000004  40020020   ADDI  R2,R0,20
00000008  28230000   LW    R3,0[R1]         ;; @1@10:
0000000C  2C230040   SW    40[R1],R3
00000010  44420001   SUBI    R2,R2,1
00000014  40210004   ADDI  R1,R1,4
00000018  0440FFF0   BNEZ  R2,FFF0          ;; se R2 != 0 goto @1@10
0000001C  F8000000   HALT
```

Figura 6.11: *Listagem do código do programa **Copymem3**.*

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	209	4,701923	78,47
Pentium	141	6,985714	69,50
PowerPC 603	97	10,187500	86,60
Alpha 21064	112	8,810811	88,39

Figura 6.12: *Tabela de resultados do programa **Copymem3**.*

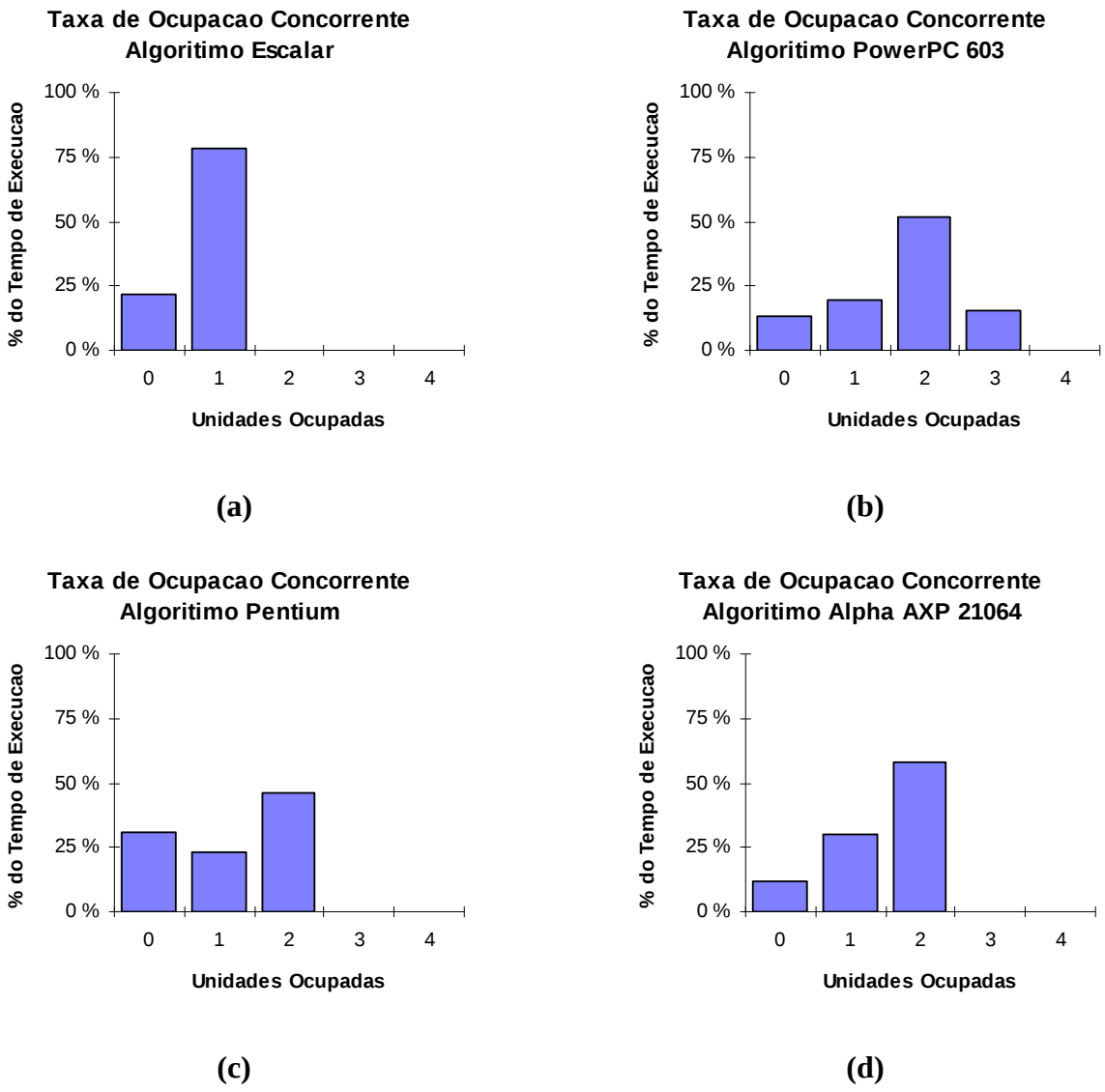


Figura 6.13: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Copymem3****.*

Forwhil2

A Execução deste programa possui uma relação na qual 50% das instruções executadas são de desvios. Além da elevada taxa de comandos de desvios existe uma forte relação de dependência entre as instruções, conforme pode ser visto na **figura 6.13**.

Os resultados obtidos nos testes deste programa foram em média um dos mais baixos, conforme pode ser visto na tabela da **figura 6.14**. Observando os resultados vemos que o algoritmo do PowerPC 603 teve desempenho muito abaixo dos outros. Este desempenho inferior se deve ao fato do simulador **Dlxwin** ter sido preparado para executar no máximo dois comandos de desvios especulativamente e paralisar o mecanismo de busca assim que um terceiro desvio for encontrado. Esta paralisação provocou um espaço de três ciclos entre as instruções durante o processamento, o que em conjunto com as previsões incorretas de desvio, ocorridas ao final de cada interação do *loop* mais interno, proporcionaram uma ociosidade de 75% nas unidades funcionais como pode ser observado no gráfico da **figura 6.15c**.

```
00000000  40010005   ADDI    R1,R0,5         ;; @INICIO:
00000004  40020010   ADDI    R2,R0,10        ;; @1@10:
00000008  44420001   SUBI    R2,R2,1         ;; @1@20:
0000000C  0440FFFC   BNEZ    R2,FFFC         ;; se R2 != 0 goto @1@20
00000010  44210001   SUBI    R1,R1,1
00000014  0420FFF0   BNEZ    R1,FFF0         ;; se R1 != 0 goto @1@10
00000018  F8000000   HALT
```

Figura 6.13: *Listagem do código do programa* ***Forwhil2.***

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	302	3,528239	58,94
Pentium	287	3,713287	61,67
PowerPC 603	333	3,198795	30,63
Alpha 21064	210	5,081340	83,81

Figura 6.14: *Tabela de resultados do programa* ***Forwhil2****.*

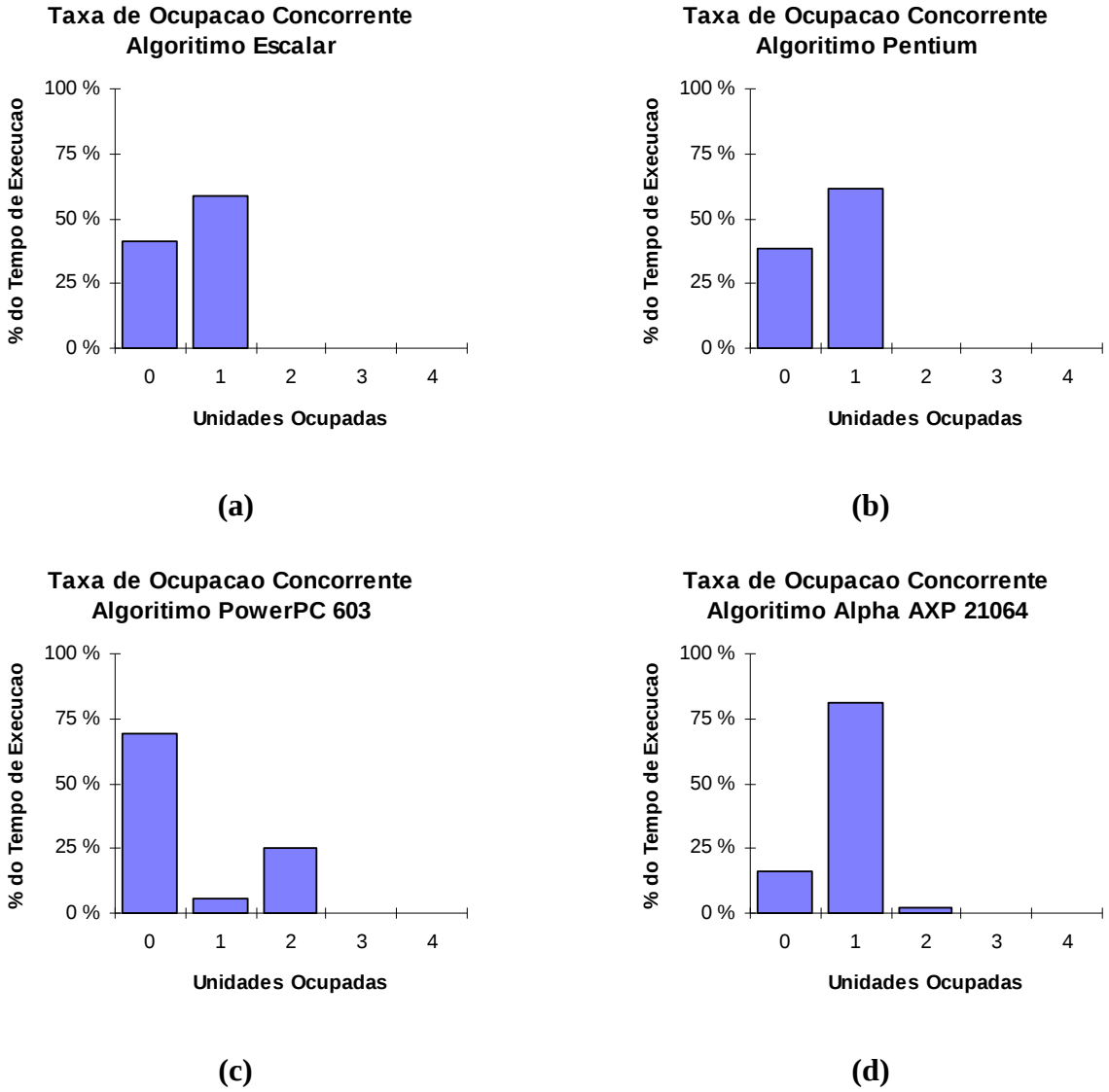


Figura 6.15: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Forwhil2****.*

Forwhile

Analogamente ao programa **Forwhil2** 50% das instruções executadas neste programa são de desvios, conforme pode ser observado na listagem do programa apresentada na **figura 6.16**. Os resultados obtidos nos dois experimentos, ver tabelas apresentadas nas **figuras 6.14** e **6.17**, foram similares para quase todos os algoritmos. A única diferença foi no desempenho do algoritmo do PowerPC que desta vez apresentou um resultado melhor que o dos outros algoritmos. Esta mudança foi motivada pelo fato de não haver um *loop* mais externo, o que proporcionou uma taxa de acertos na previsão de desvios alta.

```
00000000  40080010   ADDI    R8,R0,10        ;; @INICIO:
00000004  45080001   SUBI    R8,R8,1         ;; @1@10:
00000008  0500FFFC   BNEZ    R8,FFFC         ;; se R8 != 0 goto @1@10
0000000C  F8000000   HALT
```

Figura 6.16: *Listagem do código do programa* ***Forwhile.***

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	65	3,187500	53,85
Pentium	61	3,400000	55,74
PowerPC 603	47	4,434783	44,68
Alpha 21064	48	4,340426	72,92

Figura 6.17: *Tabela de resultados do programa* ***Forwhile****.*

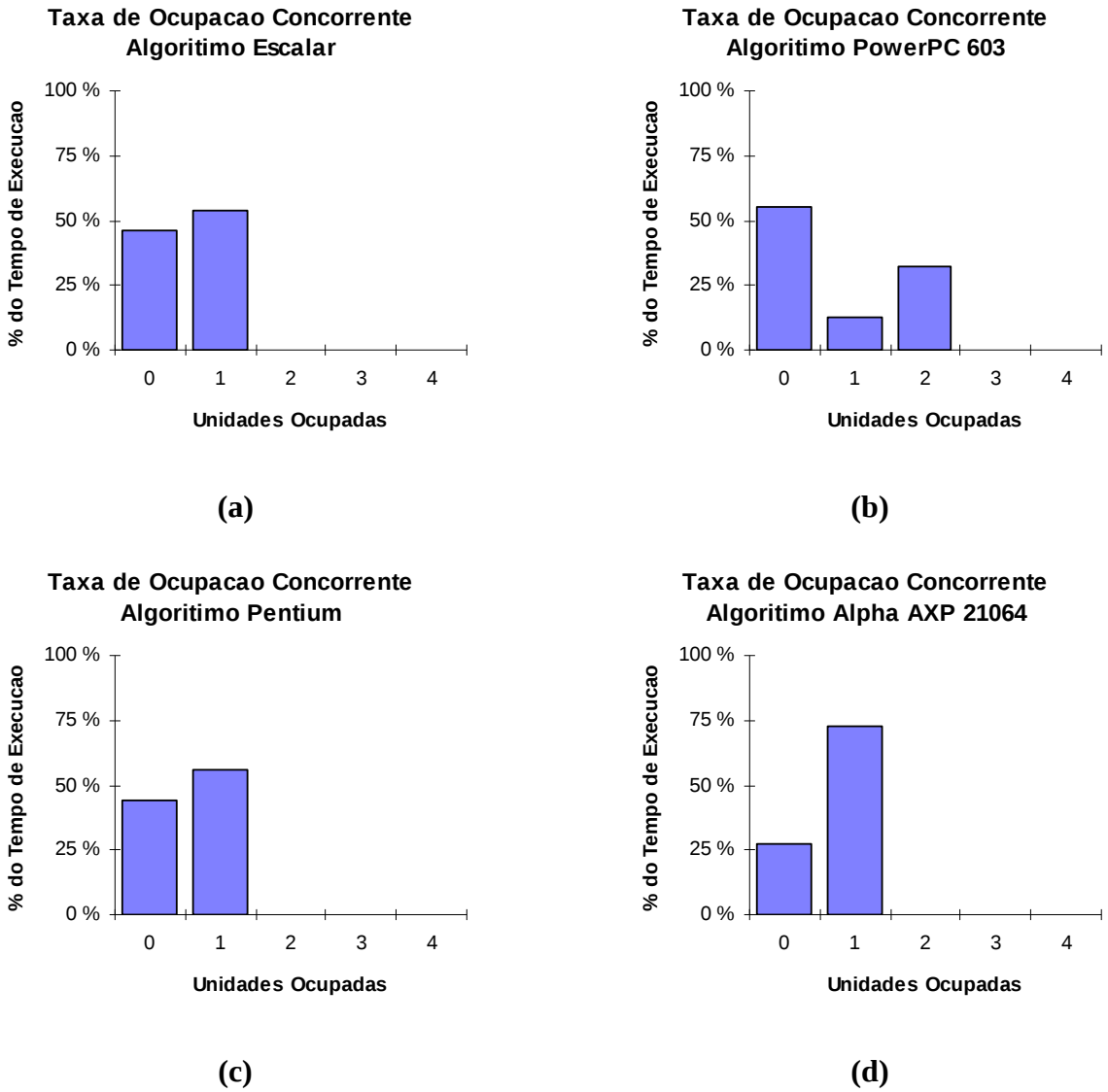


Figura 6.18: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Forwhile****.*

Polar

Dos programas de teste desenvolvidos, **Polar** é o mais completo. Este programa cria aleatoriamente uma tabela com dez vetores na forma polar e em seguida calcula a projeção destes vetores sobre os eixos cartesianos. A listagem completa do programa polar está apresentada nas páginas seguintes (**figura 6.19**).

Os resultados mostraram uma certa igualdade entre os algoritmos do PowerPC 603 e Alpha AXP 21064, conforme pode ser visto nos resultados apresentados na **figura 6.20**. Isto foi motivado pela baixa taxa de utilização concorrente de mais de duas unidades de execução, devido à dependência de dados e de controle existentes no programa, conforme pode ser visto nos gráficos da **figura 6.21**.

```
00000000  401E1000        ADDI  R30,R0,1000 ;;    R30 <- @STACK
00000004  FC1EB240        ADD   R22,R0,R30
00000008  30010374        LF    F1,374[R0]  ;;    F1 <- @MAXSZ
0000000C  30020378        LF    F2,378[R0]  ;;    F2 <- @MAXAN
00000010  FD084246        XOR   R8,R8,R8
00000014  E8000020        J     20          ;;    goto @1@20
00000018  EC000074        JAL   74          ;; @1@10: goto @RAND
0000001C  FC3C4405        MULTF F8,F1,F28
00000020  EC00006C        JAL   6C          ;;      goto @RAND
00000024  FC5C4C05        MULTF F9,F2,F28
00000028  41080008        ADDI  R8,R8,8
0000002C  3908025C        SF    25C[R8],F8  ;;    @VECT.R[R8] <- F8
00000030  39090260        SF    260[R8],F9  ;;    @VECT.A[R8] <- F8
00000034  6D170080        SLTI  R23,R8,80         ;; @1@20:
00000038  02E0FFE0        BEQZ  R23,FFE0    ;; se R8<NITENS goto @1@10
0000003C  FD084246        XOR   R8,R8,R8
00000040  E800002C        J     2C          ;;    goto @1@40
00000044  31080264        LF    F8,264[R8]  ;; @1@30:F8 <- @VECT.R[R8]
00000048  31090268        LF    F9,268[R8]  ;;    F9 <- @VECT.A[R8]
0000004C  FD20C012        MOVF  F24,F9
00000050  EC0000DC        JAL   DC          ;;    goto @COS
00000054  FD1C5405        MULTF F10,F8,F28  ;;    X <- R * COS(A)
00000058  EC0000A4        JAL   A4                ;;    goto @SIN
0000005C  FD1C5C05        MULTF F11,F8,F28  ;;    Y <- R * SIN(A)
00000060  41080008        ADDI  R8,R8,8
00000064  390A02E0        SF    2E0[R8],F10 ;;    @VECT.X[R8] <- F10
00000068  390B02E4        SF    2E4[R8],F11 ;;    @VECT.Y[R8] <- F11
0000006C  6D170080        SLTI  R23,R8,80   ;; @1@40:
00000070  02E0FFD4        BEQZ  R23,FFD4    ;; se R8<NITENS goto @1@30
00000074  F8000000        HALT
                          :
0000008C  47DE0014        SUBI  R30,R30,14        ;;@RAND
00000090  2FDF0000        SW    0[R30],R31
00000094  2FD60004        SW    4[R30],R22
00000098  3BC80008        SF    8[R30],F8
0000009C  3BC9000C        SF    C[R30],F9
000000A0  3BCA0010        SF    10[R30],F10
```

```
000000A4  FC1EB240        ADD   R22,R0,R30
000000A8  30080228        LF    F8,228[R0]        ;;    F8 <- @SEED
000000AC  3009022C        LF    F9,22C[R0]        ;;    F9 <- @C997
000000B0  FD094405        MULTF F8,F8,F9
000000B4  FD005012        MOVF  F10,F8
000000B8  FD405409        CVTF2I      F10,F10
000000BC  FD40540C        CVTI2F      F10,F10
000000C0  FD0AE403        SUBF  F28,F8,F10
000000C4  381C0228        SF    228[R0],F28       ;;    @SEED <- F28
000000C8  2BDF0000        LW    R31,0[R30]
000000CC  2BD60004        LW    R22,4[R30]
000000D0  33C80008        LF    F8,8[R30]
000000D4  33C9000C        LF    F9,C[R30]
000000D8  33CA0010        LF    F10,10[R30]
000000DC  43DE0014        ADDI  R30,R30,14
000000E0  0BE00000        JR    R31
                          :
000000FC  47DE0008        SUBI  R30,R30,8         ;; @SIN:
00000100  2FDF0000        SW    0[R30],R31
00000104  2FD60004        SW    4[R30],R22
00000108  47DE0004        SUBI  R30,R30,4
0000010C  3BC80000        SF    0[R30],F8
00000110  30080248        LF    F8,248[R0]        ;;    n = 1 (C1)
00000114  E8000048        J     48                ;;    goto @2@10
                          :
0000012C  47DE0008        SUBI  R30,R30,8         ;; @COS:
00000130  2FDF0000        SW    0[R30],R31
00000134  2FD60004        SW    4[R30],R22
00000138  47DE0004        SUBI  R30,R30,4
0000013C  3BC80000        SF    0[R30],F8
00000140  30080244        LF    F8,244[R0]        ;;    n = 0 (C0)
00000144  E8000018        J     18                ;;    goto @2@10
                          :
0000015C  47DE0020        SUBI  R30,R30,20        ;; @2@10:
00000160  3BC90000        SF    0[R30],F9
00000164  3BCA0004        SF    4[R30],F10
00000168  3BCB0008        SF    8[R30],F11
0000016C  3BCC000C        SF    C[R30],F12
00000170  3BCD0010        SF    10[R30],F13
00000174  3BCE0014        SF    14[R30],F14
00000178  3BCF0018        SF    18[R30],F15
0000017C  3BD0001C        SF    1C[R30],F16
00000180  FC1EB240        ADD   R22,R0,R30
00000184  300D0248        LF    F13,248[R0]       ;;          (C1)
00000188  300E0244        LF    F14,244[R0]       ;;          (C0)
0000018C  3010024C        LF    F16,24C[R0]       ;;          (C)
00000190  FDC04812        MOVF  F9,F14            ;;          r = 0
00000194  FDA05012        MOVF  F10,F13           ;;          sign = 1
00000198  FD005812        MOVF  F11,F8            ;; @2@20:   a = n
0000019C  FDA06012        MOVF  F12,F13           ;;          b = 1
000001A0  E8000010        J     10                ;;    goto @2@40
000001A4  FD986405        MULTF F12,F12,F24       ;; @2@30:   b *= ang
000001A8  FD8B6407        DIVF  F12,F12,F11       ;;          b /= a
000001AC  FD6D5C03        SUBF  F11,F11,F13       ;;          a -= 1
000001B0  FD607415        SGTF  F11,F14                 ;; @2@40:
000001B4  E3FFFFF0        BFPT  FFFFFFF0    ;; se a > 0 goto @2@30
000001B8  FD4C7C05        MULTF F15,F10,F12       ;;    tmp = sign * b
000001BC  FD2F4C01        ADDF  F9,F9,F15         ;;    r += b
000001C0  FDCA5403        SUBF  F10,F14,F10       ;;    sign = -sign
000001C4  FD0D4401        ADDF  F8,F8,F13         ;;
000001C8  FD0D4401        ADDF  F8,F8,F13         ;;    n += 2
000001CC  FD808415        SGTF  F12,F16
```

```
000001D0  E3FFFFC8          BFPT  FFFFFFC8          ;; se b>0 goto @2@20
000001D4  FD20E012          MOVF  F28,F9
000001D8  33C90000          LF    F9,0[R30]
000001DC  33CA0004          LF    F10,4[R30]
000001E0  33CB0008          LF    F11,8[R30]
000001E4  33CC000C          LF    F12,C[R30]
000001E8  33CD0010          LF    F13,10[R30]
000001EC  33CE0014          LF    F14,14[R30]
000001F0  33CF0018          LF    F15,18[R30]
000001F4  33D00020          LF    F16,20[R30]
000001F8  43DE0020          ADDI  R30,R30,20
000001FC  33C80000          LF    F8,0[R30]
00000200  43DE0004          ADDI  R30,R30,4
00000204  2BDF0000          LW    R31,0[R30]
00000208  2BD60004          LW    R22,4[R30]
0000020C  43DE0008          ADDI  R30,R30,8
00000210  0BE00000          JR    R31
                    :
00000228  3F07464B          DF    0.528416312342    ;; @SEED:
0000022C  44794000          DF    997.0             ;; @C977:
                    :
00000244  00000000          DF    0                 ;; @C0:
00000248  3F800000          DF    1                 ;; @C1:
0000024C  38D1B717          DF    0.0001            ;; @C:
                    :
00000264  00000000          DF    0                 ;; @VECT.R:
00000268  00000000          DF    0                 ;; @VECT.A:
                    :
000002E8  00000000          DF    0                 ;; @VECT.X:
000002EC  00000000          DF    0                 ;; @VECT.Y:
                    :
00000374  42C80000          DF    100.0             ;; @MAXSZ:
00000378  40C90FDB          DF    6.283185307       ;; @MAXAN:
```

Figura 6.19: *Listagem do código do programa* ***Polar****.*

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	24189	3,839425	63,99
Pentium	19009	4,885732	79,20
PowerPC 603	14821	6,267611	82,67
Alpha 21064	14888	6,239404	82,80

Figura 6.20: *Tabela de resultados do programa* ***Polar****.*

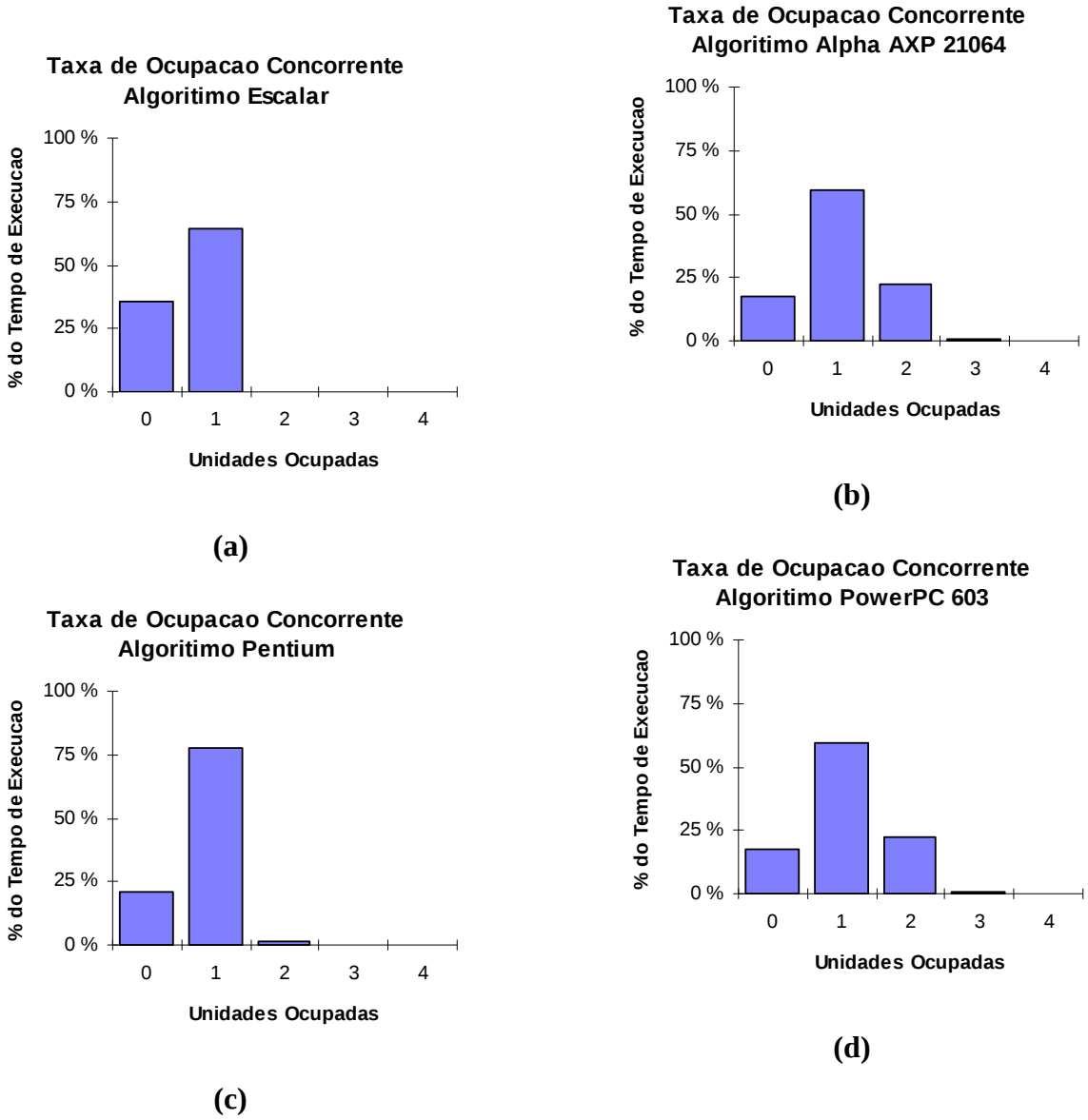


Figura 6.21: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Polar****.*

Rand

O programa **Rand** lê o conteúdo de uma posição da memória e processa um valor aleatório utilizando como semente o conteúdo lido. Este programa não possui instruções de desvios. O programa **Rand** está listado na **figura 6.22**.

Os resultados apresentados mostram uma equivalência de desempenho entre os algoritmos do Pentium, PowerPC 603 e Alpha AXP 21064, conforme pode ser observado na tabela da **figura 6.23** e nos gráficos da **figura 6.24**.

A implementação do algoritmo do Pentium obteve na maioria dos testes resultados inferiores aos obtidos pelos algoritmos do PowerPC e Alpha. Estes resultados foram motivados por uma forte restrição existente no algoritmo de despacho do Pentium quanto ao despacho de desvios. O fato deste programa não possuir desvios permitiu ao algoritmo do Pentium uma igualdade de resultados com os outros.

```
00000000  401E1000          ADDI  R30,R0,1000 ;; @INICIO: R30 <- @STACK
00000004  FC1EB240          ADD   R22,R0,R30
00000008  30080054          LF    F8,54[R0]   ;; F8 <- @SEED
0000000C  400803E5          ADDI  R8,R0,3E5   ;; R8 <- RNDGEN
00000010  47DE0004          SUBI  R30,R30,4
00000014  2FC80000          SW    0[R30],R8
00000018  33C90000          LF    F9,0[R30]
0000001C  43DE0004          ADDI  R30,R30,4
00000020  FD204C0C          CVTI2F      F9,F9
00000024  FD094405          MULTF F8,F8,F9
00000028  3808008C          SF    8C[R0],F8
0000002C  2808008C          LW    R8,8C[R0]
00000030  510800FF          ANDI  R8,R8,0FF
00000034  2C080054          SW    54[R0],R8   ;; @SEED <- R8
00000038  FC08E240          ADD   R28,R0,R8
0000003C  F8000000          HALT
                    :
00000054  3F07464B          DF    0.528416312342    ;; @SEED:
00000058  00000000          DW    0                 ;; @ARRAY:
```

Figura 6.22: ***Listagem do código do programa Rand.***

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
Escalar	27	3,692308	62,96
Pentium	17	6,000000	70,59
PowerPC 603	17	6,375000	70,59
Alpha 21064	17	6,375000	70,59

Figura 6.23: *Tabela de resultados do programa* ***Rand****.*

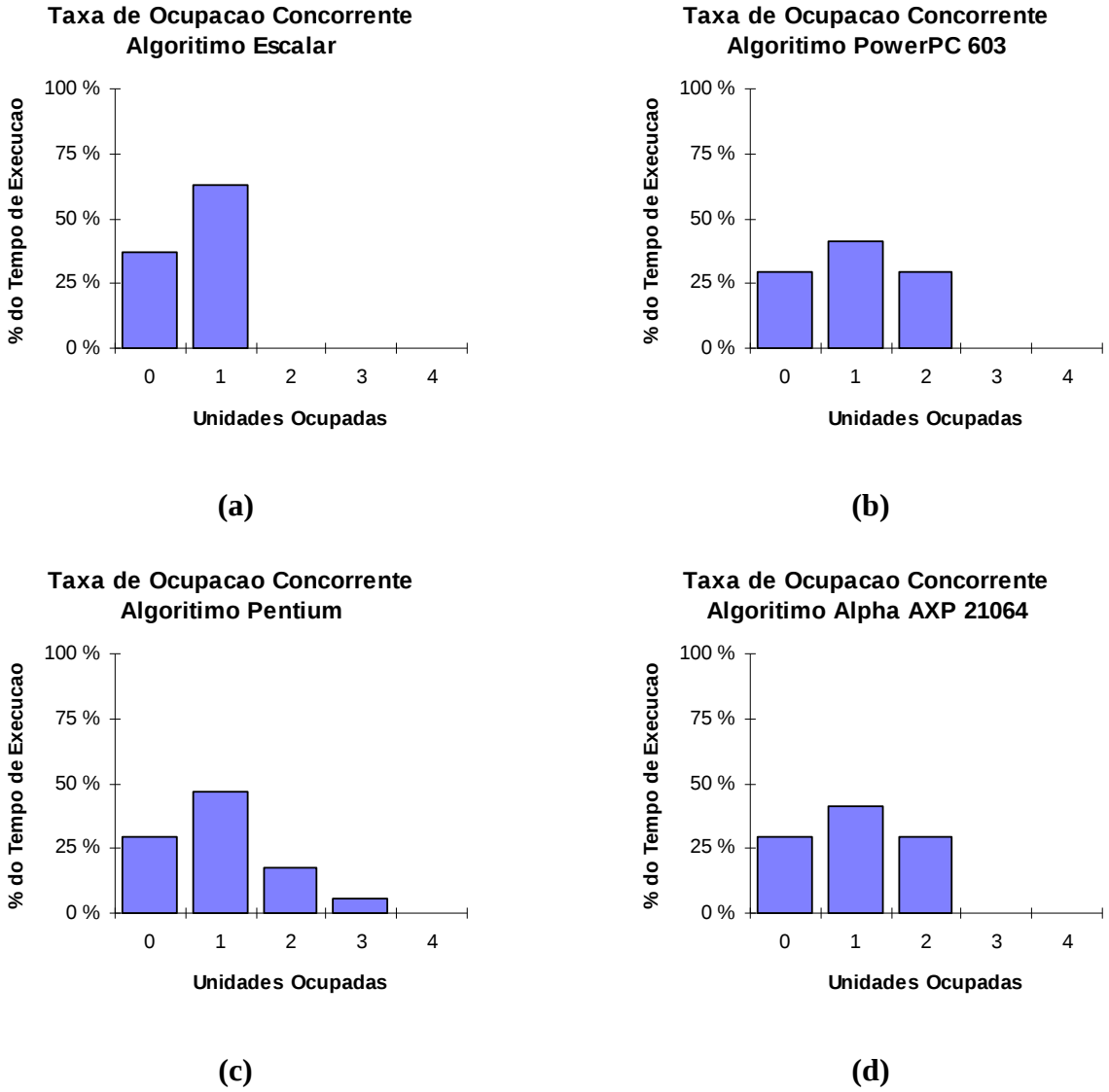


Figura 6.24: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Rand****.*

Sincos

O programa **Sincos** calcula o seno e o cosseno de sete ângulos expressos em radianos. Conforme pode ser observado na tabela da **figura 6.26** e nos gráficos das **figuras 6.27b** e **6.27d**, novamente encontramos uma equivalência de desempenho entre o PowerPC 603 e o Alpha AXP 21064. Este resultado se assemelha ao apresentado no experimento com o programa **Polar**. Sendo motivado pela baixa taxa de utilização concorrente de mais de duas unidades de execução, devido as dependências de dados e de controle existentes no programa. Conforme pode ser visto na listagem do programa apresentada nas páginas seguintes (**figura 6.25**).

```
00000000  401E1000          ADDI  R30,R0,1000       ;;    R30 <- @STACK
00000004  FC1EB240          ADD   R22,R0,R30
00000008  FD084246          XOR   R8,R8,R8
0000000C  E8000024          J     24                ;;    goto @1@20
00000010  31180184          LF    F24,184[R8]       ;; @1@10:
00000014  EC000030          JAL   30                ;;    goto @SIN
00000018  FC000240          ADD   R0,R0,R0          ;;          'NOP'
0000001C  391C01B4          SF    1B4[R8],F28
00000020  EC000058          JAL   58                ;;    goto @COS
00000024  FC000240          ADD   R0,R0,R0          ;;          'NOP'
00000028  391C01E4          SF    1E4[R8],F28
0000002C  41080004          ADDI  R8,R8,4
00000030  6D17001C          SLTI  R23,R8,1C         ;; @1@20:
00000034  02E0FFDC          BEQZ  R23,FFDC    ;; se R8<NITENS goto @1@10
00000038  F8000000          HALT
                    :
00000044  47DE0008          SUBI  R30,R30,8         ;; @SIN:
00000048  2FDF0000          SW    0[R30],R31
0000004C  2FD60004          SW    4[R30],R22
00000050  47DE0004          SUBI  R30,R30,4
00000054  3BC80000          SF    0[R30],F8
00000058  3008016C          LF    F8,16C[R0]        ;;    n = 1 (C1)
0000005C  E8000048          J     48                ;;    goto @2@10
                    :
00000074  47DE0008          SUBI  R30,R30,8         ;; @COS:
00000078  2FDF0000          SW    0[R30],R31
0000007C  2FD60004          SW    4[R30],R22
00000080  47DE0004          SUBI  R30,R30,4
00000084  3BC80000          SF    0[R30],F8
00000088  30080168          LF    F8,168[R0]        ;;    n = 0 (C0)
0000008C  E8000018          J     18                ;;    goto @2@10
                    :
000000A4  47DE0020          SUBI  R30,R30,20        ;; @2@10:
000000A8  3BC90000          SF    0[R30],F9
000000AC  3BCA0004          SF    4[R30],F10
000000B0  3BCB0008          SF    8[R30],F11
000000B4  3BCC000C          SF    C[R30],F12
000000B8  3BCD0010          SF    10[R30],F13
000000BC  3BCE0014          SF    14[R30],F14
000000C0  3BCF0018          SF    18[R30],F15
000000C4  3BD0001C          SF    1C[R30],F16
```

```
000000C8  FC1EB240          ADD   R22,R0,R30
000000CC  300D016C          LF    F13,16C[R0]       ;;          (C1)
000000D0  300E0168          LF    F14,168[R0]       ;;          (C0)
000000D4  30100170          LF    F16,170[R0]       ;;          (C)
000000D8  FDC04812          MOVF  F9,F14            ;;          r = 0
000000DC  FDA05012          MOVF  F10,F13           ;;          sign = 1
000000E0  FD005812          MOVF  F11,F8            ;; @2@20:   a = n
000000E4  FDA06012          MOVF  F12,F13           ;;          b = 1
000000E8  E8000010          J     10                ;;    goto @2@40
000000EC  FD986405          MULTF F12,F12,F24       ;; @2@30:   b *= ang
000000F0  FD8B6407          DIVF  F12,F12,F11       ;;          b /= a
000000F4  FD6D5C03          SUBF  F11,F11,F13       ;;          a -= 1
000000F8  FD607415          SGTF  F11,F14           ;; @2@40:
000000FC  E3FFFFF0          BFPT  FFFFFFF0    ;; se a > 0 goto @2@30
00000100  FD4C7C05          MULTF F15,F10,F12       ;;    tmp = sign * b
00000104  FD2F4C01          ADDF  F9,F9,F15         ;;    r += b
00000108  FDCA5403          SUBF  F10,F14,F10       ;;    sign = -sign
0000010C  FD0D4401          ADDF  F8,F8,F13         ;;
00000110  FD0D4401          ADDF  F8,F8,F13         ;;    n += 2
00000114  FD808415          SGTF  F12,F16
00000118  E3FFFFC8          BFPT  FFFFFFC8          ;; se b>0 goto @2@20
0000011C  FD20E012          MOVF  F28,F9
00000120  33C90000          LF    F9,0[R30]
00000124  33CA0004          LF    F10,4[R30]
00000128  33CB0008          LF    F11,8[R30]
0000012C  33CC000C          LF    F12,C[R30]
00000130  33CD0010          LF    F13,10[R30]
00000134  33CE0014          LF    F14,14[R30]
00000138  33CF0018          LF    F15,18[R30]
0000013C  33D00020          LF    F16,20[R30]
00000140  43DE0020          ADDI  R30,R30,20
00000144  33C80000          LF    F8,0[R30]
00000148  43DE0004          ADDI  R30,R30,4
0000014C  2BDF0000          LW    R31,0[R30]
00000150  2BD60004          LW    R22,4[R30]
00000154  43DE0008          ADDI  R30,R30,8
00000158  0BE00000          JR    R31
                  :
00000168  00000000          DF    0                 ;; @C0:
0000016C  3F800000          DF    1                 ;; @C1:
00000170  38D1B717          DF    0.0001            ;; @C:
                  :
00000184  00000000          DF    0                 ;; @VANG:   ( 0d)
00000188  3E860A92          DF    0.261799388       ;;          (15d)
0000018C  3F060A92          DF    0.523598776       ;;          (30d)
00000190  3F490FDB          DF    0.785398163       ;;          (45d)
00000194  3F860A92          DF    1.047197551       ;;          (60d)
00000198  3FA78D36          DF    1.308996939       ;;          (75d)
0000019C  3FC90FDB          DF    1.570796327       ;;          (90d)
                  :
000001B4  00000000          DF    0                 ;; @VSIN:
                  :
000001E4  00000000          DF    0                 ;; @VCOS:
```

Figura 6.25: *Listagem do código do programa* ***Sincos****.*

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
ESCALAR	4131	3,765617	62,77
PENTIUM	3241	4,805556	76,40
POWERPC	2711	5,738745	75,69
ALPHA	2727	5,705062	76,13

Figura 6.26: *Tabela de resultados do programa* ***Sincos****.*

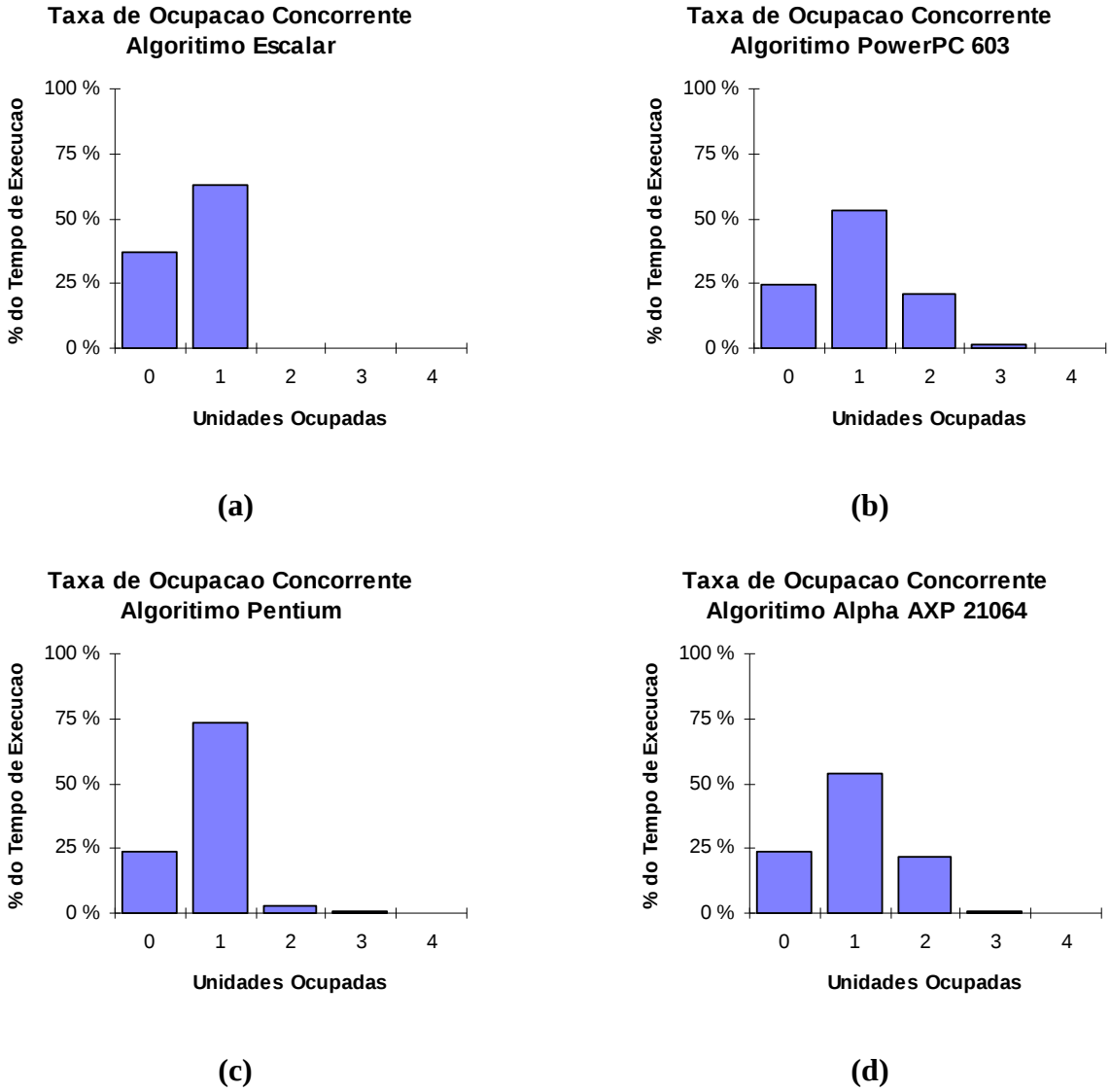


Figura 6.27: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Sincos****.*

Sort

O programa **Sort** gera um vetor de dez elementos aleatoriamente e posteriormente faz a ordenação do vetor. Este programa possui dois *loops* aninhados, conforme pode ser visto na listagem do programa apresentada na **figura 6.28**. Esta característica semelhante existente entre o programa **Sort** e o programa **Forwhil2** justificam os resultados baixos obtidos, conforme pode ser visto na **figura 6.29**.

Neste programa o algoritmo de despacho do Alpha AXP 21064 teve desempenho superior aos dos outros algoritmos, pois conseguiu manter a taxa de ociosidade das unidades funcionais a valores mais baixos, em torno de 20%, enquanto os outros algoritmos obtiveram valores acima de 25% conforme pode ser visto nos gráficos da **figura 6.30**.

```
00000000  401E1000        ADDI  R30,R0,1000 ;; @INICIO: R30 <- @STACK
00000004  FC1EB240        ADD   R22,R0,R30
00000008  FD084246        XOR   R8,R8,R8
0000000C  E8000010        J     10                ;; goto @1@20
00000010  EC000068        JAL   68                ;; @1@10:
00000014  41080001        ADDI  R8,R8,1
00000018  191C00FF        SB    FF[R8],R28  ;;    @ARRAY-1[R8] <- R28
0000001C  6D17000A        SLTI  R23,R8,A          ;; @1@20:
00000020  02E0FFF0        BEQZ  R23,FFF0    ;; se R8 < 10 goto @1@10
00000024  400A0009        ADDI  R10,R0,9
00000028  FD084246        XOR   R8,R8,R8          ;; @1@30:
0000002C  FD294A46        XOR   R9,R9,R9
00000030  E8000028        J     28                ;; goto @1@60
00000034  150B0100        LBU   R11,100[R8]       ;; @1@40:
00000038  150C0101        LBU   R12,101[R8]
0000003C  FD6CBA4C        SLE   R23,R11,R12
00000040  02E00014        BEQZ  R23,14      ;; se R11<=R12 goto @1@50
00000044  FC000240        ADD   R0,R0,R0          ;;          'NOP'
00000048  190C0100        SB    100[R8],R12
0000004C  190B0101        SB    101[R8],R11
00000050  40090001        ADDI  R9,R0,1
00000054  41080001        ADDI  R8,R8,1           ;; @1@50:
00000058  FD0ABA4A        SLT   R23,R8,R10        ;; @1@60:
0000005C  02E0FFD8        BEQZ  R23,FFD8    ;; se R8 < R10 goto @1@40
00000060  454A0001        SUBI  R10,R10,1
00000064  0520FFC4        BNEZ  R9,FFC4     ;; se R9 != 0 goto @1@30
00000068  F8000000        HALT
                          :
00000078  47DE0008        SUBI  R30,R30,8         ;; @RAND:
0000007C  2FDF0000        SW    0[R30],R31
00000080  2FD60004        SW    4[R30],R22
00000084  47DE0014        SUBI  R30,R30,14
00000088  2FC80000        SW    0[R30],R8
0000008C  3FC80004        SD    4[R30],F8
00000090  3FC9000C        SD    C[R30],F9
00000094  47DE0004        SUBI  R30,R30,4
00000098  FC1EB240        ADD   R22,R0,R30
```

```
0000009C  300800FC        LF    F8,FC[R0]         ;; F8 <- @SEED
000000A0  400803E5        ADDI  R8,R0,3E5         ;; R8 <- 997
000000A4  2EC80000        SW    0[R22],R8
000000A8  32C90000        LF    F9,0[R22]
000000AC  FD204C0C        CVTI2F      F9,F9
000000B0  FD094405        MULTF F8,F8,F9
000000B4  3AC80000        SF    0[R22],F8
000000B8  2AC80000        LW    R8,0[R22]
000000BC  510800FF        ANDI  R8,R8,0FF
000000C0  2C0800FC        SW    FC[R0],R8         ;; @SEED <- R8
000000C4  FC08E240        ADD   R28,R0,R8
000000C8  43DE0004        ADDI  R30,R30,4
000000CC  37C9000C        LD    F9,C[R30]
000000D0  37C80004        LD    F8,4[R30]
000000D4  2BC80000        LW    R8,0[R30]
000000D8  43DE0014        ADDI  R30,R30,14
000000DC  2BD60004        LW    R22,4[R30]
000000E0  2BDF0000        LW    R31,0[R30]
000000E4  43DE0008        ADDI  R30,R30,8
000000E8  0BE00000        JR    R31
                          :
000000FC  3F07464B        DF    0.528416312342    ;; @SEED:
00000100  00000000        DW    0                 ;; @ARRAY:
```

Figura 6.28: *Listagem do código do programa* ***Sort****.*

ALGORITMO	CICLOS	SPEEDUP	TAXA DE OCUPAÇÃO
ESCALAR	1308	3,791890	63,23
PENTIUM	944	5,255567	72,78
POWERPC	849	5,900943	67,84
ALPHA	758	6,705416	81,27

Figura 6.29: *Tabela de resultados do programa* ***Sort****.*

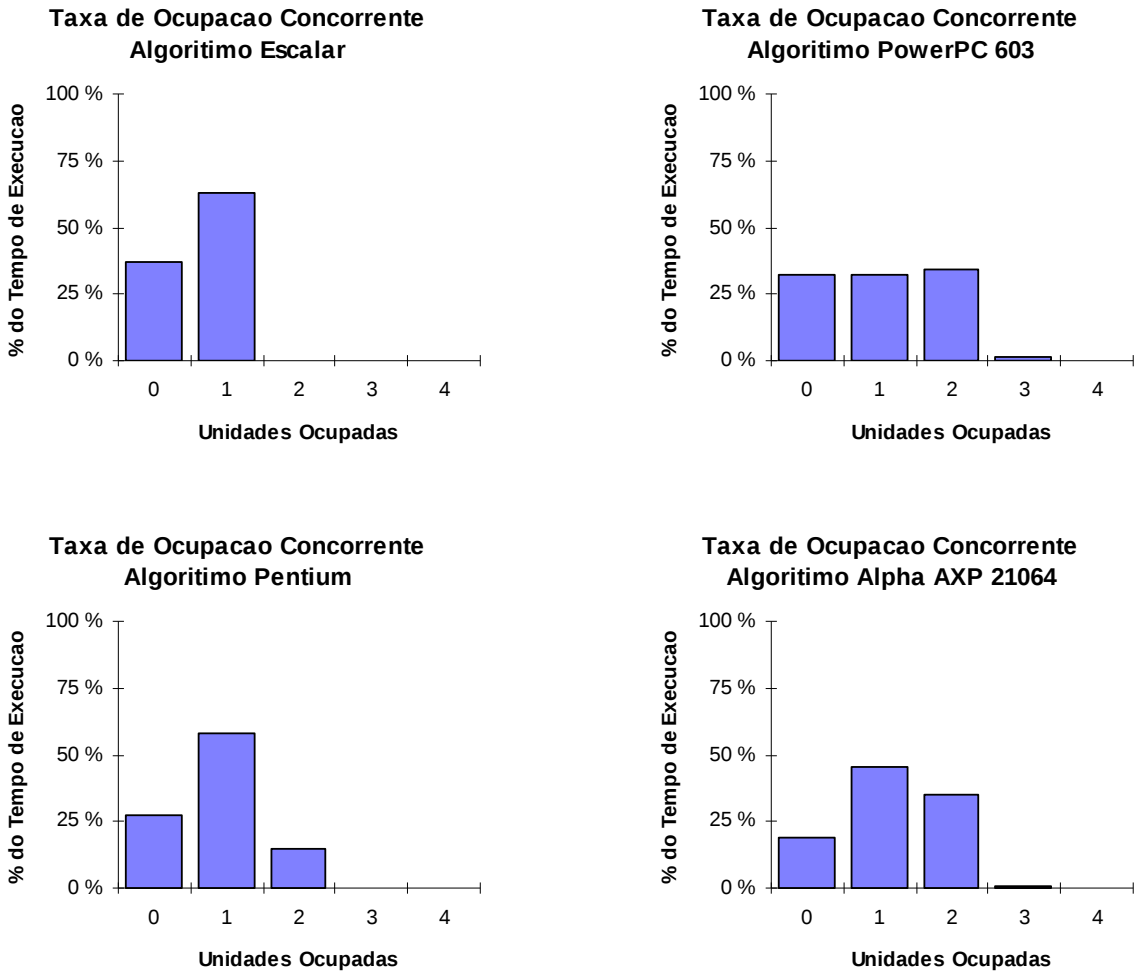


Figura 6.30: *Gráficos representativos da taxa de ocupação concorrente das unidades de execução obtidos nos experimentos com o programa* ***Sort****.*

Nas tabelas a seguir estão relacionados os valores calculados do *speedup* e da taxa de ocupação das unidades de execução produzidas nos experimentos. Em destaque estão os algoritmos que tiveram melhor desempenho médio entre os algoritmos analisados.

Speedup

	Escalar	***Pentium***	***PowerPC 603***	***Alpha 21064***
Array	4,315389	6,750991	8,076650	7,757875
Copymem	4,075000	4,794118	9,878788	8,358974
Copymem2	5,556818	6,985714	10,187500	8,810811
Copymem3	4,701923	6,985714	10,187500	8,810811
Forwhil2	3,528239	3,713287	3,198795	5,081340
Forwhile	3,187500	3,400000	4,434783	4,340426
Polar	3,839425	4,885732	6,267611	6,239404
Rand	3,692308	6,000000	6,375000	6,375000
Sincos	3,765617	4,805556	5,738745	5,705062
Sort	3,791890	5,255567	5,900943	6,705416
Media	4,045411	5,357668	7,024632	6,818512

Figura 6.31: *Tabela de resultados do critério speedup obtidos nos experimentos.*

Taxa de Ocupação

	Escalar	***Pentium***	***PowerPC 603***	***Alpha 21064***
Array	71,93	89,34	91,19	95,26
Copymem	68,05	79,02	87,00	88,98
Copymem2	92,66	69,50	86,60	88,39
Copymem3	78,47	69,50	86,60	88,39
Forwhil2	58,94	61,67	30,63	83,81
Forwhile	53,85	55,74	44,68	72,92
Polar	63,99	79,20	82,67	82,80
Rand	62,96	70,59	70,59	70,59
Sincos	62,77	76,40	75,69	76,13
Sort	63,26	72,78	67,84	81,27
Media	67,69	72,37	72,35	82,85

Figura 6.32: *Tabela de resultados do critério taxa de ocupação obtidos nos experimentos.*

Conforme pode ser observado na tabela apresentada na **figura 6.31**, o desempenho do algoritmo do PowerPC medido em *speedup* foi superior ao dos outros em quase todos os testes, sendo os únicos resultados desfavoráveis a este algoritmo obtidos nas execuções dos programas **Forwhil2** e **Sort**. Estes resultados desfavoráveis foram motivados pelo fato dos programas **Forwhil2** e **Sort** serem compostos por dois *loops* aninhados e o simulador **Dlxwin** ter sido preparado para executar no máximo dois comandos de desvios especulativamente e paralisar o mecanismo de busca assim que um terceiro desvio for encontrado. Esta paralisação provocou um espaço de três ciclos entre as instruções durante o processamento, o que em conjunto com as previsões incorretas de desvio, ocorridas ao final de cada interação do *loop* mais interno, proporcionaram uma baixa taxa de ocupação das unidades funcionais pelo algoritmo do PowerPC como pode ser observado nos resultados destes experimentos apresentados na tabela da **figura 6.32**.

O algoritmo do processador Alpha obteve superioridade na análise da taxa de ocupação das unidades de execução, conforme pode ser visto na tabela da **figura 6.32**, mas seu desempenho quase sempre foi inferior ao desempenho do PowerPC (*speedup* e tempo de processamento, ver tabela da **figura 6.31**). Este resultado pode ser compreendido com a análise do algoritmo de despacho dos dois processadores. Com base nesta análise vemos que o PowerPC têm a propriedade de despachar até três instruções por ciclo, o que aumenta sensivelmente a probabilidade de execução paralela. Já o algoritmo do Alpha, até duas instruções podem ser despachadas a cada ciclo.

A implementação do algoritmo do Pentium foi por muitas vezes de desempenho inferior aos dos processadores PowerPC e Alpha, mas nos exemplos com pouca quantidade de *loops*, como **Array**, **Polar**, **Rand** e **Sincos**, o algoritmo do Pentium conseguiu bons resultados como pode ser observado nas **figuras 6.31** e **6.32**. Estes resultados foram motivados pelo fato do algoritmo do Pentium possuir uma restrição no despacho de desvios. Deste modo o Pentium embora algumas vezes tenha obtido desempenho próximo ao dos outros algoritmos, só conseguiu se igualar a eles quando o programa em analise não possuía desvios (programa **Rand**).

Para avaliar o ganho de performance com o tratamento apropriado do código de um programa foram feitas três versões do programa **Copymem.** Cada versão possui uma mudança no posicionamento do decremento do contador de modo a evitar dependências de dados e proporcionar um aumento na taxa de ocupação concorrente. Conforme pode ser observado no quadro da **figura 6.33**, a diferença de desempenho observada nas otimizações realizadas ficou entre 36,36 e 45,71% para os algoritmos Escalar e Pentium respectivamente e em 3.12 e 5,41% respectivamente para os algoritmos do PowerPC e Alpha. Os valores mais baixos observados no ganho de desempenho entre melhor e pior resultados obtido pelos algoritmos do PowerPC e Alpha, são justificadas pelo fato destes algoritmos terem muito menos restrições de despacho no mecanismo de escalonamento, do que os algoritmos Escalar e Pentium, e por isso os ganhos de desempenho com a otimização do código são menores.

	Copymem	***Copymem2***	***Copymem3***	***Ganho****
Escalar	4,075000	5,556818	4,701923	36,36%
Pentium	4,794118	6,985714	6,985714	45,71%
PowerPC 603	9,878788	10,187500	10,187500	3,12%
Alpha 21064	8,358974	8,810811	8,810811	5,41%

*Representa o ganho de desempenho entre o melhor e pior resultado.

Figura 6.33: *Tabela de resultados obtidos nos experimentos de otimização do código.*

Capítulo 7

Conclusão

O tema central deste projeto é o estudo dos algoritmos de despacho e a análise do efeito deles no desempenho das maquinas super escalares. Este estudo foi motivado pelo fato de ser o algoritmo de despacho um elemento decisivo no sucesso de uma arquitetura super escalar.

Para comprovar a importância do mecanismo de escalonamento de instruções, foi implementado o algoritmo de despacho de três processadores existentes no mercado Pentium, Alpha AXP 21064 e PowerPC 603. Todos sobre uma mesma arquitetura base, do simulador **Dlxwin**, de modo a proporcionar ao mecanismo de despacho um papel diferenciador nos resultados de desempenho obtidos. Também foi implementado um algoritmo escalar que utiliza os recursos de paralelismo próprios do simulador mas com despacho seqüencial, o que proporcionou uma comparação entre o ganho de desempenho obtido com a implementação de um algoritmo super escalar sobre um algoritmo escalar utilizando os recursos de paralelismo embutidos no processador (*pipeline*, *busca antecipada de instruções* e *previsão de desvios*).

Os resultados obtidos nos testes comprovaram que em certas circunstâncias um algoritmo de despacho adequado pode dar a um processador super escalar a possibilidade de executar um programa até 5 vezes mais rápido, em média, do que um processador escalar sem recursos de paralelismo implementado no *hardware* e até 1.75 vezes mais rápido, em média, do que com a implementação de um algoritmo de despacho escalar em uma máquina utilizando recursos de paralelismo embutidos no *hardware* (*pipeline, busca antecipada de instruções e previsão de desvios*).

Avaliação dos algoritmos implementados

PowerPC 603

O processador PowerPC foi desenvolvido objetivando a redução na penalidade do desvio, com um algoritmo que permite o despacho simultâneo de até três instruções, mecanismos de previsão de desvios, execução especulativa de desvios e unidades de reserva, este processador possui muita semelhança com o processador implementado e seu algoritmo de despacho obteve desempenho superior na maioria dos testes.

Nos experimentos realizados o algoritmo de despacho do PowerPC obteve a melhor média de *speedup* 7,02.

Alpha AXP 21064

O algoritmo implementado pelo processador Alpha conseguiu bom desempenho, apesar das fortes restrições impostas por seu algoritmo quanto ao lançamento de instruções dependente o que limita o despacho paralelo de instruções. No Alpha até duas instruções podem ser despachadas por ciclo.

O processador Alpha AXP 21064 obteve a segunda melhor média na medição do *speedup* 6,82 e a melhor média na taxa de ocupação das unidades de execução 82,85%.

Pentium

O algoritmo do Pentium foi desenvolvido para solucionar o problema de prover instruções a dois processadores idênticos sem execução especulativa e unidades de reserva, portanto o algoritmo implementado era bastante restritivo e dificilmente superaria o desempenho dos outros processadores. Em programas com pouca ocorrência de desvios seu desempenho foi comparável ao do Alpha AXP 21064.

Dos três algoritmos de despacho super escalar analisados o Pentium obteve o pior resultado no critério *speedup* com 5,36 de média e o segundo melhor resultado na taxa de ocupação 72,37% de média.

Avaliação do trabalho

As informações apresentadas neste trabalho fornecem subsídios para a comprovação da importância do algoritmo de despacho no desempenho de máquinas super escalares. Os algoritmos analisados são a base dos processadores atuais de mercado e grande parte do diferencial deles está incluído no mecanismo de despacho, e com base nos resultados obtidos podemos comprovar que a implementação de um bom algoritmo de despacho, adequado a característica do computador, pode representam um ganho que varia de 30 a 75% no desempenho do processador.

A implementação dos três algoritmos de despacho empregados por processadores atuais no simulador **Dlxwin** e os experimentos realizados, constataram que o PowerPC 603 é o processador de melhor desempenho entre os três analisados. Seu bom desempenho nos experimentos foi motivado pela pouca restrição encontrada no despacho das dependência de dados e controle oferecida pelo algoritmo. Na prática esta pouca restrição apresentada pelo algoritmo produz um custo extra causado pelos componentes adicionais que precisam ser incorporados ao processador para controlar e resolver dependências.

Analisando custo de projeto, um algoritmo mais restritivo como o implementado pelo Pentium necessita de menos unidades adicionais para controlar e resolver dependências, desta forma seu custo é muito menor.

Entre os recursos implementados nas máquinas super escalares, a possibilidade de execução especulativa é a que conduz ao custo de projeto mais elevado, pela quantidade de componentes que precisam ser adicionados ao processador para possibilitar este recurso. Dos três processadores que tiveram o algoritmo de despacho implementado no simulador **Dlxwin**, o PowerPC 603 é na prática o único com possibilidade de executar desvios especulativamente, mas seu desempenho em vários experimentos não foi muito melhor do que o obtido pelo Alpha AXP 21064 que na prática não dispõem desta facilidade. Assim o Alpha AXP 21064 é dos três processadores o de melhor relação custo beneficio.