Universidade de Brasília

Notas de Aula

Fundamentos de Sistemas Computacionais

Organização e Arquitetura de Computadores

Autor:
Felipe Rodopoulos

Professora:
Dr. Alba Melo

28 de setembro de 2017

Sumário

1 Revisão Inicial **1**
- 1.1 Arquitetura x Organização . . . 1
- 1.2 Execução de Programas . . . 1
  - 1.2.1 Desvios . . . 2
- 1.3 Pipelining . . . 2
  - 1.3.1 Hazards . . . 4
- 1.4 Barramento . . . 9
  - 1.4.1 Síncronos x Assíncronos . . . 10

2 Cache e RAM **11**
- 2.1 Cache . . . 11
  - 2.1.1 Colocações e Localização de Bloco em Cache . . . 12
  - 2.1.2 Substituição de Blocos em *Cache* . . . 14
  - 2.1.3 Leitura de Blocos em *Cache* . . . 14
  - 2.1.4 Escrita de Blocos em *Cache* . . . 15
  - 2.1.5 Medidas de Desempenho . . . 16
- 2.2 Melhoria de Desempenho de Cache . . . 16
  - 2.2.1 Reduzindo Cache Misses . . . 17
  - 2.2.2 Reduzindo Penalidade do Miss . . . 22
  - 2.2.3 Reduzindo Tempo de Hit . . . 24
- 2.3 Memória RAM . . . 24
  - 2.3.1 DRAM . . . 24
  - 2.3.2 Variações da DRAM . . . 26

3 RISC x CISC **27**
- 3.1 CISC . . . 27
- 3.2 RISC . . . 28
- 3.3 RISC x CISC . . . 32

4 Instruction Level Parallelism **35**

4.1 Depêndencias 36
4.1.1 Dependências de Dados 36
4.1.2 Dependências de Nomes 37
4.1.3 Dependências de Controle 38
4.2 Paralelismo a Nível de *Loop* 39
4.3 Semântica e Notações de Desvios 41
4.3.1 Previsões 41
4.4 Técnicas de Exploração 42
4.4.1 *Loop Unrolling* 42
4.4.2 Escalonamento Dinâmico 42
4.4.3 Scoreboarding 43
4.4.4 Tomasulo 45
4.4.5 Previsão Dinâmica de Desvios 46
4.4.6 Execução Especulativa 50
4.4.7 Arquiteturas Superescalares 51
4.4.8 Very Large Instruction Word - VLIW 53
4.4.9 Análise de Dependência pelo Compilador 54

5 Multiprocessadores Simétricos 57
5.1 Classificação de Arquiteturas 57
5.2 Multiprocessadores Simétricos 58
5.3 Coerência de Cache 61
5.4 Snoopy Caches 62
5.4.1 Principais Protocolos de Coerência 63
5.4.2 Protocolos Baseados em Diretórios 67

Capítulo 1

Revisão Inicial

1.1 Arquitetura x Organização

Arquitetura de computadores se refere a atributos do sistema que são visíveis ao programador (Assembly).

Exemplos: conjunto de instruçoes, endereçamento de memória, ...

Organização de computadores está relacionado às unidades operacionais do computador e suas interconexões (barramentos).

Exemplos: sinais de controle, tecnologia de memória e cache, ...

1.2 Execução de Programas

A execução de programas envolve a execução de instruções. No início do ciclo de instrução, o processador busca da memória a instrução cujo endereço está no **program counter (PC)**. A instrução é carregada no **registrador de instrução** (instruction register - IR) e o processador finalmente decodifica a instrução para sua futura execução.

A instrução decodificada pode ser de diferentes **tipos**:

- Transferência de dados entre processador e memória (LOAD e STORE);
- Transferência de dados entre periféricos e processador (*input* e *output*);

- Execução de operações lógicas e aritméticas;
- Alterações na sequência de execução das instruções (*branches*).

O **ciclo de instruções** envolve as seguintes etapas:

- **Instruction Address Calculation (IAC):** determinação do endereço da próxima instrução;
- **Instruction Fetch (IF):** carregamento da instrução no processador;
- **Instruction Decoding (ID):** determinação do tipo de instrução e seus operandos necessários;
- **Operand Fetch (OF):** busca dos operandos;
- **Data Operation (DO):** execução da operação;
- **Operand Store (OS):** escrita do resultado da operação.

1.2.1 Desvios

No caso de uma execução sequencial, o endereço da próxima instrução é obtido através de uma operação de soma no PC. Entretanto, as execuções de programas raramente são sequenciais, onde em alguns pontos do código caminhos alternativos são tomados. Blocos de decisão e *loops* são responsáveis por tais desvios, presentes em linguagens de alto nível. Nas instruções de máquina essas operações são implementadas por *branches* e *jumps*.

1.3 Pipelining

Pipelining é a forma de executar múltiplas instruções, de forma sobreposta, acelerando a execução de uma programa como um todo (e não executar instruções mais rapidamente).

As instruções são executadas de forma sobreposta, divida por estágios, que devem ser completados em um ciclo de *clock*. Qualquer combinação de estágios deve poder ocorrer ao mesmo tempo. Entretanto, tal fato acaba resultando em um desbalanceamento entre os estágios, dado que alguns estágios são mais lentos que outros, limitando o desempenho. Um exemplo, são estágios de execução (EX) com operações de ponto flutuante, que demoram mais que um ciclo.

Alguns estágios dependem da saída de estágios anteriores. Assim, *overheads* são postos por registradores de pipeline, chamados de **latches**. Os dados e controles necessários por estágios avançados são passados através dos *latches*, adiantando a passagem destes parâmetros.

Instruction Fetch

Busca a instrução da memória e carrega do *registrador de instrução*. A posição da instrução é determinada pelo `PC`. Ao fim do carregamento, o `PC` é incrementado em 4, determinando o endereço da próxima instrução (se não houver *branch*), porém este é inserido no `NPC`. Logo a sequência acaba sendo:

`IR ← Mem[PC]`

`NPC ← PC + 4`

Instruction Decode

Decodifica a instrução no IR e acessa os registradores indicados na mesma. O registradores fazem parte de um conjunto de registradores de uso do programador e são carregados para registradores temporários para uso na instrução (chamamos eles de `A` e `B`).

Os 16 bits menos significativos do `IR` são armazenados no registrador de imediato, o `Imm`. A decodificação e busca de operando é feita em parelelo, chamada de *decodificação de campo fixo*.

Execution

Executa as operações indicadas na instrução. Esta depende da instrução, podendo ser:

- **Referência a memória**, onde o efetivo é calculado com a ajuda do imediato: `ALUOutput ← A + Imm`;
- **ALU registrador-registrador**, que são operações aritméticas envolvendo registradores apenas: `ALUOutput ← A func B`;
- **ALU registrador-memória**, que são operações aritméticas envolvendo a memória e um registrador: `ALUOutput ← A op Imm`

- **Branches**, que são desvios condicionais no fluxo de execução do código:

  `ALUOutput` ← `NPC + Imm`

  `Cond` ← `(A op 0)`

Acesso à memória

Executa `loads`, `stores` e *branches*. Se a instrução for `LOAD`, o dado é lido da memória e colocado em um registrador próprio, o `LMD`.

`LMD` ← `Mem[ALUOutput]`

Se for um `STORE`, os dados contidos no registrador indicado são escritos em memória.

`Mem[ALUOutput]` ← `B`

Se for um *branch*, o `PC` ganha um valor diferente dependendo da tomada de decisão na condição. Logo:

```
if (cond)
PC ← ALUOutput
else
PC ← NPC
```

Write Back

Escreve os resultados das operações nos registradores e/ou memória. Logo envolve instruções dos tipos:

- **ALU registrador-registrador:** `Regs[IR16...20]` ← `ALUOutput`;
- **ALU registrador-imediato:** `Regs[IR11...15]` ← `ALUOutput`;
- **Load**: `Regs[IR11...15]` ← `LMD`.

1.3.1 Hazards

Situações que impedem que o próximo estágio da instrução seja executado no próximo ciclo de clock, sendo a maior fonte de atraso no pipeline, deixando-o ocioso.

Stall: nome do atraso resultante de um hazard

Bubble: atraso a nível de estágio

Hazard Estrutural

Para executar instruções de forma paralela as unidades funcionais devem estar sobrepostas (*pipelined*) e duplicadas. **Quando instruções não podem ser executadas em paralelo por falta de recurso** (*hardware*), temos um hazard estrutural. Logo, ocorrem por dois motivos:

1. As unidades funcionais não estão dispostas de maneira sobreposta;
2. Os recursos não estão replicados suficientemente, levando a conflitos de utilização.

As vezes vale a pena permitir um hazard estrutural pois o *stall* pode ser menos custoso que um *overhead* de paralelização ou inserção de mais *hardware*

Hazard de Dados

Ocorrem quando o *pipeline* para porque uma instrução precisa esperar o término de outra, o que chamamos de **dependência entre instruções**. Ocorre pois o *pipeline* pode mudar a ordem acesso aos operando. Temos três tipos de dependências:

- **RAW - Read After Write**: queremos ler um valor que ainda vamos escrever;
- **WAW - Write After Write**: queremos escrever um valor onde ainda precisamos escrever outro;
- **WAR - Write After Read**: queremos escrever um valor onde ainda precisamos ler outro;

Podemos tentar resolver com ***forwarding*** (ou *bypassing*): transferimos o dado necessário diretamente da unidade que o produz para a unidade que o necessita. Feitor por meio de ligação física direta.

Algumas instruções, como o `LOAD`, apenas disponibilizam o dado em um estágio muito avançado, atrasando o *pipeline* ao ponto de uma instrução seguinte. O **pipeline interlock** é o *hardware* que detecta tal hazard e insere *bubbles* o suficiente até que o *forwarding* possa ser realizado corretamente e a execução continue.

O compilador também pode auxiliar, identificando e evitando certos padrões, reeordenando o código de forma a evitar os *stalls*. Chamamos isso de **pipeline**

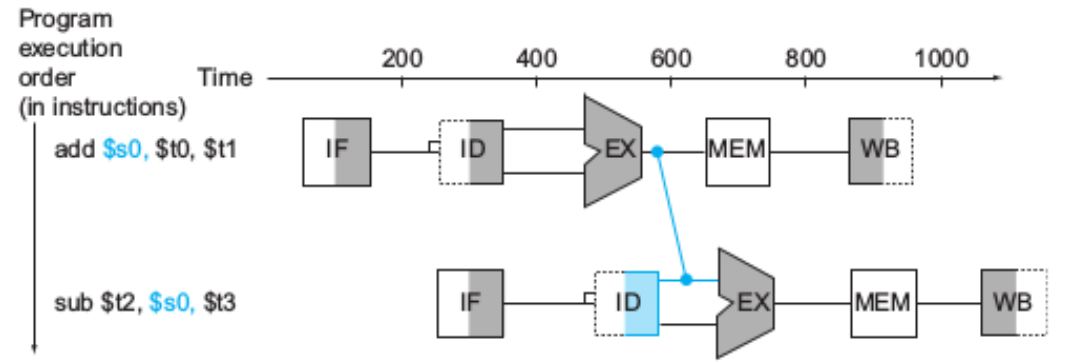


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

Figura 1.1: Esquema de Branch Forwarding

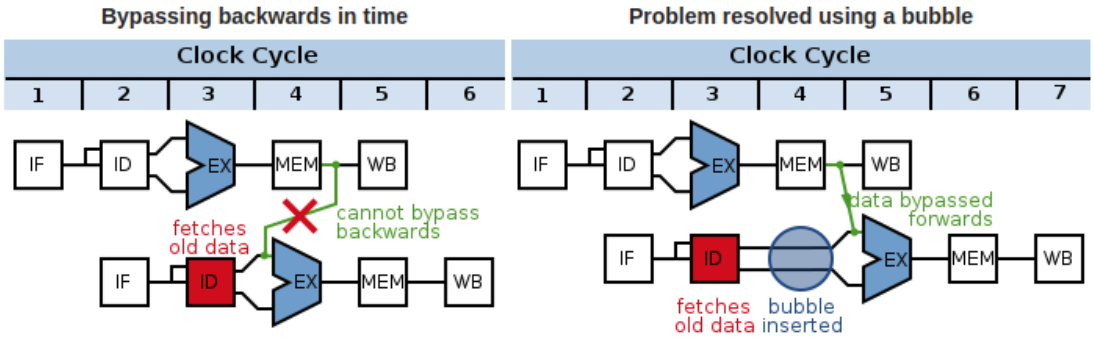


Figura 1.2: Inserção de *bubbles* para acontecer o *forwarding*

scheduling, onde o programa é separado em os blocos básicos e as instruções são escalonadas dentro deles.

Blocos Básicos: sequência de instruções onde não há desvios ou I/O. Todas as instruções são executadas se a primeira for.

Hazard de Controle

Define-se como **o atraso resultante da espera de saber se um desvio é tomado**. A instrução de desvio pode mudar o `PC` (*branch taken*) ou não (*not taken*) e, se tomado, este resultado só será conhecido no estágio MEM. Podemos reduzir combinando a previsão de tomada do desvio e cálculo prévio do novo do `PC`.

PREVISÃO NOT TAKEN
Todos os *branchs* são tratados como *not taken* e o *pipeline* é carregado com as instruções seguintes a ele. O estado da máquina não é alterado até o desvio ser conhecido e se a previsão falhar, as mudanças são desfeitas, o que chamamos de **back out**.

PREVISÃO TAKEN
Todos os *branchs* são tratados como *not taken* e o *pipeline* é carregado com as instruções a partir do endereço apontado por ele. Não é muito efetiva, já que em várias arquiteturas ainda não se conhece o alvo do desvio.

DELAYED BRANCH
Muito usado nos primeiros RISCs. O compilador tenta inserir um conjunto de instruções que sempre será executado após o *branch*, independente do seu resultado. Chamamos este conjunto de ***branch delay slot***, o qual estará sempre após a instrução de desvio e normalmente tem tamanho 1. Há três formas de se escalonar:

- **From before:** retira instruções independentes do desvio e posteriores a ele, inserindo-as no *slot*. Esta é a melhor opção e deve sempre ser tomada quando possível;
- **From target:** usado quando há maior probabilidade de desvio *taken*. Insere no *slots* as instruções presentes no endereço do desvio. Se a previsão falhar, haverá trabalho disperdiçado;
- **From fall through:** usando quando há maior probabilidade de desvio *not taken*. Insere no *slot* as instruções logo depois da instrução do desvio.

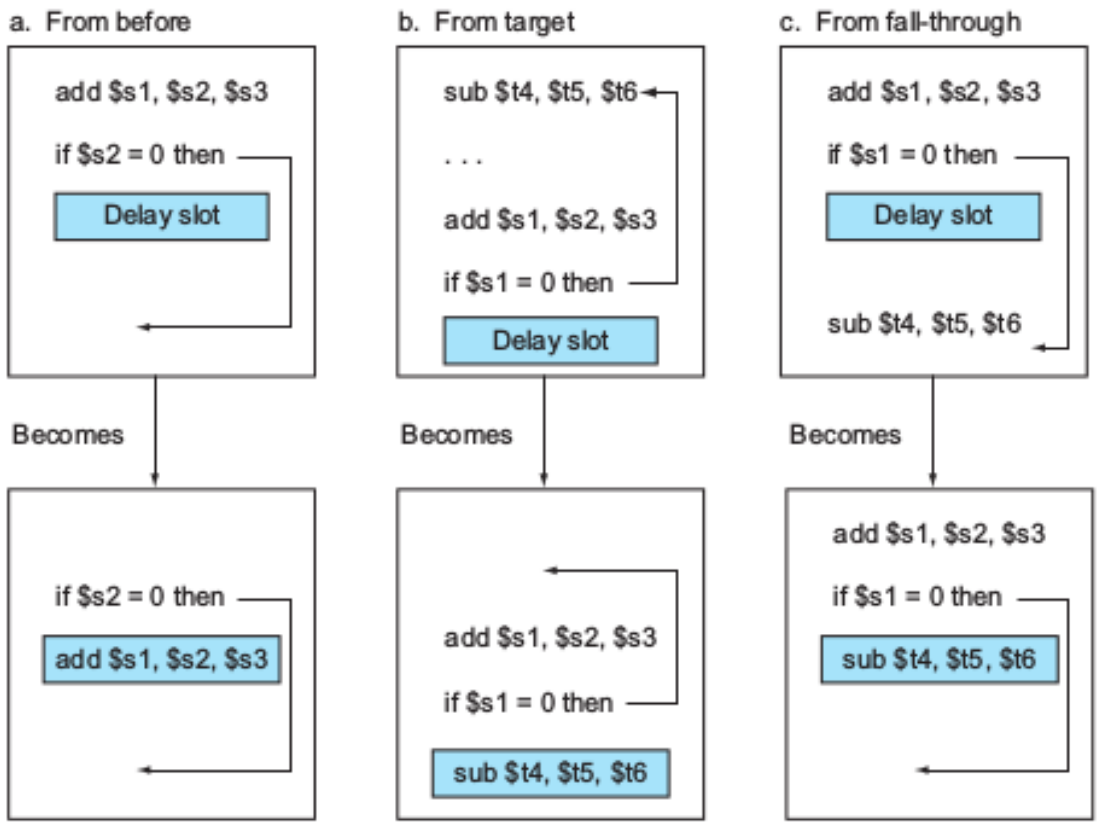


Figura 1.3: Escalonamentos do delayed branch

Se a previsão é correta, o *delay slot* é executado. Caso contrário, fazemos o **cancelamento do *branch***: inserimos um `NOP` no *delay slot* e executamos. Isso remove restrições extras a serem colocadas no *slot*. Normalmente, os *cancelling branches* são tomados quando há o *not taken*.

Esta técnica possui limitações. Em tempo de compilação, é difícil de prever se vai haver o desvio e o conjunto de instruções podem conter outras depêndencias e *branches* que impedem a previsão.

1.4 Barramento

Definição: meio de transmissão compartilhado que conecta dispositivos. Sua velocidade máxima é limitada pelo seu tamanho e o número de dispositivos conectados a ele.

Barramentos do tipo CPU-Memória são pequenos e de alta velocidade, enquanto os de I/O são maiores, acomodando diversos tipos de dispositivos, seguindo um determinado padrão.

Um barramento é composto por:

- **Linhas de dados:** cada linha transporta 1 bit por vez;
- **Linhas de endereço:** onde o endereço da palavra acessada é colocado
- **Linhas de controle:** controlam o uso das linhas
  - Memory write & memory read
  - Transfer ACK
  - Bus request, que pede o controle do barramento
  - Bus grant, que obtém o controle do barramento
  - Clock e reset

A **leitura de dados da memória** é feita com o envio de um endereço e sinais de controle indicativo leitura. A memória coloca o dado no barramento e retira - ou *deasserts* - o sinal de espera (*wait*).

A **escrita dos dados em memória** é feita com a CPU enviando o endereço e do dado ao barramento. A memória retira estes sinais e atualiza o dado. Geralmente, a CPU não espera a confirmação.

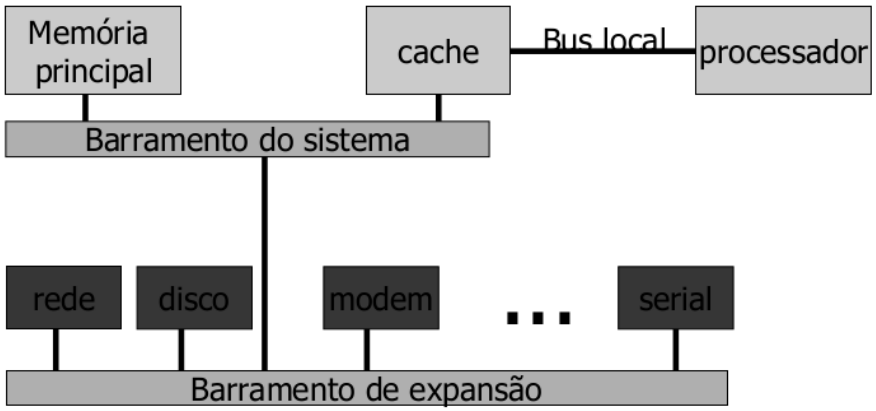


Figura 1.4: Arquitetura com diferentes barramentos

Os **bus masters** são dispositivos que podem iniciar transações no barramento, como a CPU. Sistemas com vários masters necessitam de um esquema de arbitragem para resolução de conflitos, geralmente usando prioridade fixa ou randômica.

Split transaction buses são buses *pipelined* ou *packet-switched*. As transações no barramento são divididas em etapas, de forma a não bloquear o mesmo durante toda a transação. A unidade envolvida participa da arbitragem. As transações possuem *tags*, para serem identificadas.

Exemplo: transação de *read* pode ser dividida em *read-request* e *memory-reply*.

1.4.1 Síncronos x Assíncronos

Nos **barramentos síncronos** uma das linhas de controle é um *clock*. Os protocols para endereço e dados são fixos e baseados neste *clock*. Normalmente são os barramentos CPU-Memória, uma vez que são rápidos e baratos. Devidos as distorções no *clock* (*clock skew*), eles não podem ser longos.

Já **barramentos assíncronos** usam protocolos de *handshaking* entre o emissor e receptor do dado, provando um *overhead* de sincronização a cada transação. Por isso são mais lentos, mas permitem que mais tipos de dispositivos sejam utilizados, sendo ótimos para gerênciar *buses* de I/O.

Capítulo 2

Cache e RAM

Definição **Princípio de Localidade:** programas tendem a reusar dados e instruções que foram usadas recentemente. Podemos dividir em:

- **Localidade temporal:** itens acessados recentemente têm grande probabilidade de serem acessados novamente;
- **Localidade espacial:** itens cujos endereços estão próximos tendem a ser acessados em tempos próximos.

Logo temos a possibilidade de prever instruções que serão utilizadas em um futuro próximo com base em acessos recentes. Isso justifica o uso da **hierarquia de memória**.

2.1 Cache

Definição Memória rápida e com pouca capacidade de armazenamento. Interposta entre a CPU e a memória principal. A medida de dados é feita em **WORDS** (palavras), a qual normalmente trocada entre CPU e cache. Definimos:

Cache hit: o dado procurado está em cache

Cache miss: o dado procurado NÃO está em cache

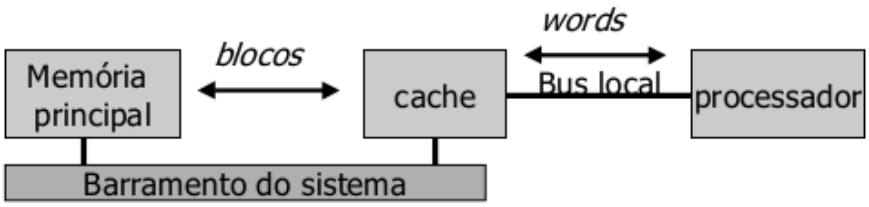


Figura 2.1: Detalhes de ligação entre cache, processador e memória

2.1.1 Colocações e Localização de Bloco em Cache

A cache é bem menor que a memória, logo precisamos de um modo de inserir blocos da memória na cache e indexa-los de forma eficaz. O formato de um endereço do ponto de vista da *cache* segue a Figura 2.2. Dentro de um endereço de tamanho A, temos:

Figura 2.2: Estrutura do endereço do dado, do ponto de vista da **cache**

- **Índice -** I: a posição do bloco dentro da cache. No caso do set associative, é o índice do conjunto em que o bloco se encontra.

$$I = \lg \frac{C}{b} \tag{2.1}$$

- **Deslocamento - D**: é índice da palavra em que o dado se encontra, em relação ao bloco. Deve ser o número de bits para representar o número de palavras em um bloco;

$$D = \lg w \tag{2.2}$$

- **Tag - T**: verificador usado para indicar se há um miss ou um hit na hora do acesso. É sempre o que sobra no endereço depois do índice e deslocamento.

$$T = A - (I + D) \tag{2.3}$$

A seguir, iremos definir como esse endereço se estrutura ao longo dos três tipos de colocação do bloco em cache. A Figura 2.3 dá um ótimo exemplo de como um mesmo dado é colocado e disponibilizado ao longo dos três métodos. Antes, defina que a cache tem tamanho C, com blocos de tamanho $B = w.b$, onde w é o

tamanho da palavra e b o número de bytes por palavra. Para sabermos o número total de blocos N_B dentro da cache, usaremos a Equação 2.4.

$$N_B = \frac{C}{b} \tag{2.4}$$

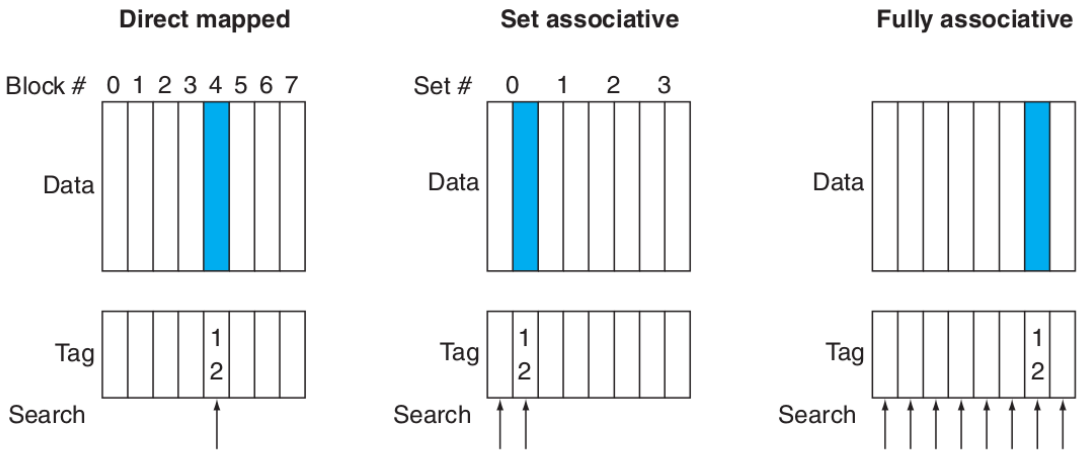


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Figura 2.3: Diferentes abordagens para colocação de cache. Desenho retirado do Patterson

Mapeamento Direto

O bloco só pode ser posto em um lugar na *cache*, ou seja, o bloco da memória ocupa uma única linha da *cache*. Para achar a linha fazemos:

$$i = B_M \bmod N_B \tag{2.5}$$

onde i é o número da linha e B_M o número do bloco da memória principal.

Set Associative

Aqui o bloco pode ser colocado em um conjunto de blocos. Ao ser acessado novamente, o controlador tem que procurar o bloco dentro deste conjunto. Dentro do conjunto, o bloco pode ser colocado em qualquer posição (varia por algoritmo).

Quando temos n blocos em cada conjunto, a colocação em *cache* é chamada mn-way *set associative*. Observe que 1-way set associative equivale ao mapeamento direto. Dado isso, o índice diminui, pois há um número menor de índices na cache e o cálculo do seu tamanho no endereço se dá por:

$$I = \lg \frac{C}{n.b} \tag{2.6}$$

Fully Associative
Aqui o bloco pode ser posto em qualquer lugar da *cache*, ou seja, o conjunto de blocos é a cache inteira. Logo o bloco pode ser colocado em qualquer lugar da cache, o que leva a diminuição de misses, pois os blocos não tem uma posição fixa. Entretanto, há um overhead: para obter o bloco, devemos procurá-lo pela cache inteira.

Como não há índice de blocos e conjunto, **o tamanho do índice no endereço é 0** e só temos o deslocamento e a tag.

2.1.2 Substituição de Blocos em *Cache*

Quando há um *cache miss*, o controlador da cache deve selecionar um bloco para ser substituído. Em *caches* de mapeamento direto, não há escolha, pois somente um bloco pode ser substituído.

Já em esquemas associativos há duas maneiras:

- **Random:** visando uniformidade, os blocos são selecionados aleatoriamente;
- **LRU:** substitui blocos menos recentemente utilizados. Esta tem técnica tem custo alto e geralmente são usadas aproximações.

2.1.3 Leitura de Blocos em *Cache*

A leitura e comparação de *tag* pode ser feita simultaneamente. Por isso, a leitura já é feita quando o endereço está disponível.

Se temos um *read hit*, o dado é repassado a CPU imediatamente. Se temos um *miss*, a leitura prévia é ignorada e o bloco é carregado da memória principal. O diagrama da Figura 2.4 mostra as transições.

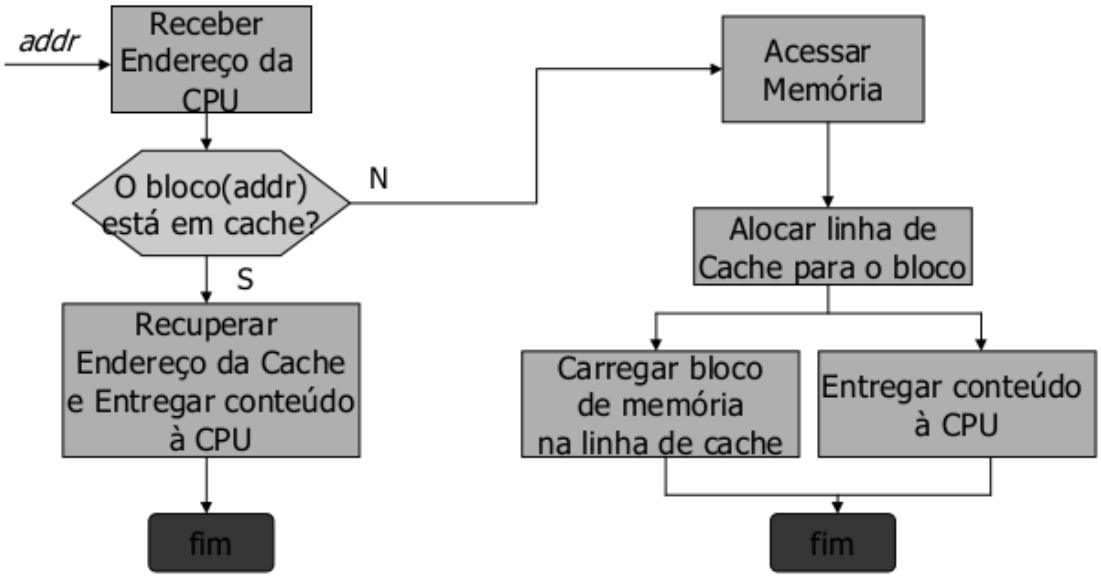


Figura 2.4: Fluxograma de decisão de leitura da cache

2.1.4 Escrita de Blocos em *Cache*

Bloco só podem ser alterados/escritos após a confirmação da *tag*, o que torna a operação mais lenta. Além disso, os processadores informam o número de bytes exatos a serem escritos, limitando a operação a essas posições. Na leitura, o acesso à dados a mais não trás problemas.

Temos duas políticas de escrita:

- **Write-through:** dado escrito tanto no bloco em *cache* como na memória. São políticas mais simples e os *read misses* nunca ocasionam escritas para a memória, já que essa política garantem sempre os dados válidos na memória a cada escrita;
- **Write-back:** dado é escrito somente em *cache*. Apenas quando há substituição que o bloco modificado é escrito em memória. Acabam por ser mais rápidas, pois ocorrem só na velocidade de *cache* e usam menos largura de banda já que envia nada a memória.

Definição **Dirty bits:** indicam se o bloco foi modificação enquanto estava em *cache* (*dirty*) ou não (*clean*), sendo que estes últimos não são escritos em memória na substituição.

Definição **Write Stalls:** ocorrem quando a CPU tem que esperar que o dado seja escrito em memória. Para reduzir este tempo, são usada **write-buffers**, onde

a CPU escreve o dado e continua suas tarefas. O dado então é enviado de maneira assíncrona para a memória.

Em um **write miss**, o dado antigo não é mais necessário e logo temos duas abordagens para tal:

- **Write allocate:** o bloco é carregado em *cache* no momento do *write miss* e logo depois o *write hit* ocorre;
- **No-write allocate:** o bloco é modificado em memória e não é carregado em *cache*.

Políticas *write-back* tendem a usar o primeiro e *write-through* o segundo.

2.1.5 Medidas de Desempenho

Calculamos o tempo média de acesso à memória:

$$A = T + M * P \tag{2.7}$$

onde A pode ser medido em frações de segundo ou ciclos de clock e:

- **Hit time -** T**:** tempo gasto em um *hit* em cache;
- **Miss rate -** M**:** taxa de misses na cache;
- **Miss penalty -** P**:** penalidade associada a cada miss.

2.2 Melhoria de Desempenho de Cache

As caches foram introduzidas para reduzir o gap entre os tempos de aceso à memória principal e a velocidade do processador. O ideal de um projeto de cache seria obter hits rápidos e poucos misses. Logo, temos que melhorar o desempenho dessas caches.

Em se tratando de **tipos de cache misses**, tempos três tipos, onde eles são relacionados no gráfico da Figura 2.5:

- **Compulsórios:** no primeiro acesso ao bloco, o mesmo não estará na cache e logo deverá ser trazido para a memória. Também chamados de *cold misses* ou *reference misses*;

- **Capacidade:** ocorrem quando um bloco escolhido para substituição é acessado novamente;

- **Conflito:** ocorrem em caches com mapeamento direto ou associativas, se muitos dos blocos referenciados são mapeados para o mesmo conjunto. Também chamados de *collision misses* ou *interference misses*.

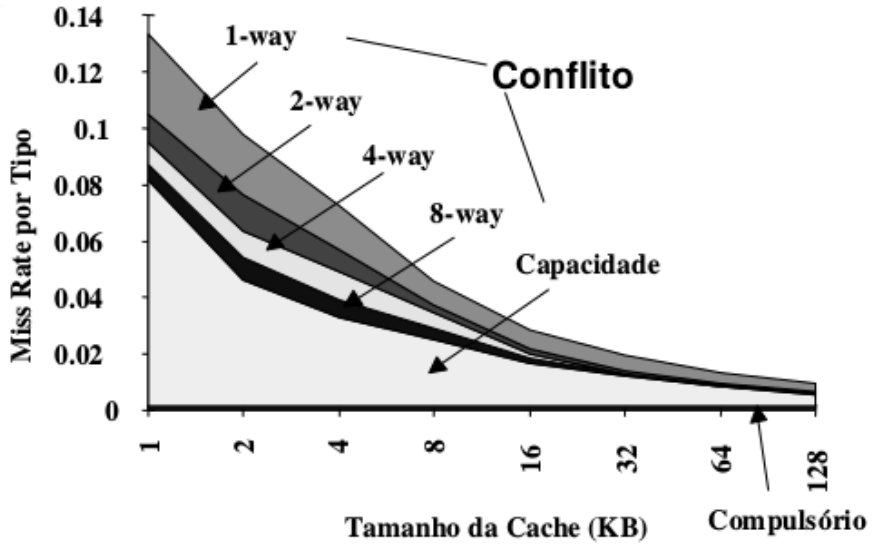


Figura 2.5: Taxas de *misses* por tipo de acesso em um SPEC92, por Patterson

2.2.1 Reduzindo Cache Misses

Aumentar o Tamanho do Bloco

Quanto maior o tamanho do bloco, menor o número de misses compulsórios, baseando-se no princípio de localidade temporal. Porém, blocos grandes fazem aumentar o número de misses de conflito e capacidade e aumentam a penalidade do miss.

A escolha do tamanho do bloco leva em conta a latência (tempo de acesso) à memória e a largura de banda à mesma. Em geral, se a largura de banda e a latência são grandes, blocos grandes são escolhidos.

Tamanho do Bloco	Tamanho da Cache				
	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Figura 2.6: Proporção de *miss rate* ao balancear tamanho de bloco e tamanho de cache

Aumentar a Associatividade

A justificativa desta técnica é que com maior associatividade, há mais blocos que potencialmente podem ser escolhidos para substituição, possibilitando o uso de algoritmos mais elaborados.

Para utilizar essa técnica, deve-se observar duas regras gerais, obtidas empiricamente:

- Uma cache 8-way set associative reduz a taxa de misses tanto quanto uma cache totalmente associativa.
- Uma cache de mapeamento direto de tamanho N apresenta aproximadamente o mesmo miss rate que uma cache 2-way associativa de tamanho $\frac{N}{2}$.

Logo, devemos ter em mente que, ao aumentar a associatividade, o projeto da cache fica mais complexo e, portanto, o tempo de hit aumenta. Logo, tem de ser pesar as duas grandezas, pois é capaz do tempo médio aumentar.

Caches Vítimas

A cache vítima é uma pequena cache totalmente associativa que é adicionada entre a cache e o nível inferior de memória imediatamente inferior. A cache vítima contém somente blocos que foram substituídos da cache por causa de um miss.

Na ocorrência de um misss, verifica-se se o bloco encontra-se na cache vítima. Se sim, o bloco vítima substitui um bloco da cache. Alguns estuso mostram que

caches vítimas de poucos bloco (4 em geral) conseguem reduzir o número de misses de conflito de caches pequenas de mapeamento direto.

Caches Pseudo-Associativas

Também chamadas de associativas por coluna, essas caches tentam obter a taxa de misses da cache associativa e o tempo de hit da cache de mapeamento direto.

No caso de hit, o acesso a esta cache funciona como o de um mapeamento direto. Em caso de miss, o pseudo-set é obtido pela inversão dos bits mais significativos do índice e a entrada da cache obtida desta maneira é verificada.

Elas possuem um tempo de hit rápido e um tempo de hit lento. Como tal tempo é variável, isso pode complicar o projeto do pipeline. Logo, ela é mais adequada em caches de segundo nível.

Pré-carga por Hardware de Instrução e Dados

Intuitivamente, a taxa de misses pode se reduzida se trouxermos os dados e instruções para cache antes dos mesmos terem sido referenciados. É uma ténica comum trazer dois blocos a cada miss: o bloco referenciado e o próximo (localidade espacial). O bloco referenciado é colocado na cache o próximo em um *stream buffer*.

Pré-carga com Auxílio do Compilador

Aqui, o compilador insere instruções de pré-carga no código do programa. Em geral, garantimos que o dado ou instrução pré-carregados não causarão excessões de memória virtual (reais ou de proteção). Por esta razão, as pré-cargas mais efetivas são as que não causam alteração no programa (*non-binding prefetching*).

A pré-carga só faz sentido se o processador puder continuar a operação enquanto o dado a ser pré-carregado não chega. Note que as caches devem ser capazes de tratar diversas requisições sem se bloquear (*lockup-free caches*).

A execução de instruções de pré-carga adiciona um overhead à execução, logo é necessário certificar que ele não irá aumentar o tempo geral.

Otimizações do Compilador

Técnicas em código, sem usar hardware adicional. Podem ser usadas técnicas de profiling para rearrumar o código, reduzindo misses das instruções.

Merging de Arrays
Ao referenciar *arrays* com o mesmo índice no mesmo *loop*, acabamos por gerar misses de conflito, já que as posições dos arrays podem estar em blocos diferentes.

```
int val[TAM];
int chave[TAM];
...
for (i=0; i<MAX; i++)
  a = a + val[i] + chave[i];
```

(a) **Código original:** dois *arrays* logicamente conectados, sendo percorridos. Aqui a cada acesso distindo à `chave[i]` e `valor[i]`, podemos ter troca de bloco no cache

```
struct merge {
  int val;
  int chave;
};
struct merge array_merge[TAM];
...
for (i=0; i<MAX; i++)
  a = a + array_merge[i].val + array_merge[i].chave;
```

(b) **Código arrumado:** Agora os dados estão sob uma mesma *struct*, logo estão adjacentes na memória. Ganho na localidade espacial quando o bloco for carregado.

Figura 2.7: Exemplo de reestruturação de código para *array merge*

Troca de Loops
Loops aninhados que não acessam dados de maneira sequencial podem acabar por acessar bloco diferentes a cada alternância entre as posições. Acesso à matrizes é um bom exemplo. O compilador pode alterar o código de maneira que o acesso seja sequencial.

```
for (j=0; j<100; j++)
  for (i=0; i<5000; i++)
    a[i][j] = a[i][j] * 2;
```

(a) **Código original:** exemplo de acesso não sequencial, onde se itera por linha na matriz

```
for (i=0; i<5000; i++)
  for (j=0; j<100; j++)
    a[i][j] = a[i][j] * 2;
```

(b) **Código arrumado:** agora iteramos por coluna, aproveitando a localidade espacial

Figura 2.8: Exemplo de reestruturação de código para troca de *loops*

Junção de Loops
O acesso a matrizes com os mesmo loops porém fazendo cálculos diferentes. Logo,

podemos tirar proveito da localidade temporal, fazendo que os dados sejam acessados diversas vezes enquanto então em cache.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
     a[i][j] = b[i][j] + c[i][j];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
     d[i][j] = b[i][j] * c[i][j];
```

(a) **Código original:** dois loops iterando sobre a mesma matriz, mas fazendo operações diferentes

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  {
     a[i][j] = b[i][j] + c[i][j];
     d[i][j] = b[i][j] * c[i][j];
  }
```

(b) **Código arrumado:** junção dos dois loops originais em um só. Tiramos proveito da localidade temporal para os dois loops, dado que os dados já estarão em cache

Figura 2.9: Exemplo de reestruturação de código para junção de *loops*

Blocagem

As vezes, quando diversas matrizes são acessas sendo umas por linhas e outras por colunas, podemos realizar uma otimização mais complexa. Operamos em submatrizes (ou blocos), ao invś de se operar sobre a matriz inteira.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  {
     r = 0;
     for (k=0; k<N; k++)
       r = r + b[i][k] * c[k][j];
     a[i][j] = r;
  }
```

(a) **Código original:** `a` e `b` percorrem por linha, enquanto `c` percorre por coluna. Percorremos a matriz inteira sequencialmente.

```
for(jj=0; jj<N; jj=jj+B)
  for(kk=0; kk<N; kk=kk+B)
    for (i=0; i<N; i++)
      for (j=jj; j<(min(jj+B-1,N); j++)
      {
         r = 0;
         for (k=kk; k<(min(kk+B-1,N); k++)
            r = r + b[i][k] * c[k][j];
         a[i][j] = r;
      }
```

(b) **Código arrumado:** criação de submatrizes e percorrimento dessas

Figura 2.10: Exemplo de reestruturação de código com blocagem

2.2.2 Reduzindo Penalidade do Miss

Priorizar Misses de Leitura

Para acelerar a conclusão da execução de instruções, geralmente *write buffers* são adicionados ao hardware. Logo, quando o store termina, geralmente o dado se encontra neste buffer e não no bloco de cache associado.

Esta decisão pode fazer com que dependendo da ordem de acesso aos dados, valores antigos sejam lidos.

Exemplo:

```
1.  LOAD 512, r3
2.  LOAD r1, 1024
3.  LOAD r2, 512
```

Caso os endereços 512 e 2014 estejam mapeados no mesmo índice de cache e a escrita de (1) demorar a se completar, a instrução 3 pode ler o valor antigo da memória, havendo um hazard RAW de dados.

A maneira mais simples de evitar que este problema aconteça, consiste em fazer com que o read miss espere até que o *write buffer* esteja vazio. Mas isso aumenta a penalidade do read miss.

Para priorizar misses de leitura, o hardware de muitos processadores verificam o contéudo do *write buffer* e, caso não haja conflito, deixa o read miss continuar.

Utilizar Sub-blocos

Consiste em dividir cada bloco da cache em sub-blocos e adicionar um bit de validade a cada sub-bloco. O tag continua associado ao bloco.

Em um read miss, somente um sub-bloco é lido da memória e assim temos a redução da penalidade (menos dados).

Early Restart e Palavra Crítica Primeiro

Estas técnias não esperam que o bloco inteiro seja colocado em cache, deixando a CPU continuar logo que a palavra desejada esiver carregada.

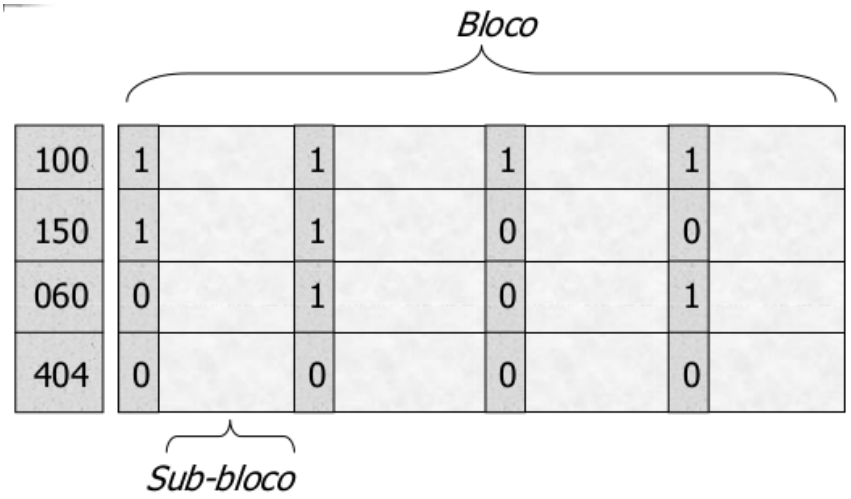


Figura 2.11: Esquema de representação de sub-blocos

- **Early restart:** carrega as palavras na ordem sequencial e, logo que a palavra desejada tiver sido carregada, a CPU continuaa a execução.
- **Palavra crítica primeiro:** solicita a palavra que causou o miss primeiro e permite que a CPU continue a execução logo que a mesma chegar

Lockup-free Caches

Também chamada de *non-blocking cache*, ela é capaz de fornecer dados que estãao na cache mesmo durante um cache miss (*hit under miss*). As mais complexas permitem o tratamento simultâneo de vários misses (*miss under miss*), se a memória for capaz de tratar diversos misses.

O projeto dessas caches é bem complexo, já que lida com diversos acessos incompletos.

Caches de Segundo Nível

Como a diferença de desempenho entre RAM e CPU é grande, a inclusão de uma cache mais lentar e de maior capacidade entre a cache tradicional e a memória é interessante. Para serem efetivas, as caches de segundo nível (L2) devem ser bem maiores que a de primeiro nível (L1).

Geralmente, a propriedade de inclusão multinível é observada porém, caso se opte

por tamanhos de blocos distintos, o projeto da hierarquia de memória fica complexo.

2.2.3 Reduzindo Tempo de Hit

Esta técnica é interessante uma vez que limita a taxa de clock do processador.

Caches Simples e Pequenas

Virtual Caches

Geralmente, a CPU lida com endereços virtuais que devem ser convertidos para endereços físicos e as caches tradicionais utilizam endereços físicos. As caches virtuais utilizam endereços virtuais, não fazendo a tradução de endereços em um cache hit.

No entanto, a cada troca de contexto, a cache deve ser esvaziada (*flushed*), pois cada processo possui seu próprio espaço de endereçamento virtual.

Pipelining de Escritas

O pipelining de escritas ocorre entre escritas distintas. Para que o dado seja escrito, primeiramente deve ser feta uma comparação com a tag e depois a escrita é feita no bloco correto. A comparação com a tag é eita no estágio 1 e a escrita no estágio 2.

2.3 Memória RAM

Definição memória semicondutora volátil, onde há a necessidade de um fornecimento constante de energia para manter os valores.

2.3.1 DRAM

Composta por células que armazenam dados, onde um transistor é utilizado para armazenar um bit. É organizada como uma matriz retangular, endereçada por linhas e colunas.

O endereço é dividido em duas metades: *row access strobe* (RAS) e *column access strobe* (CAS). O protocolo de acesso à essa memória é assíncrono. A Figura 2.12 mostra o esquema.

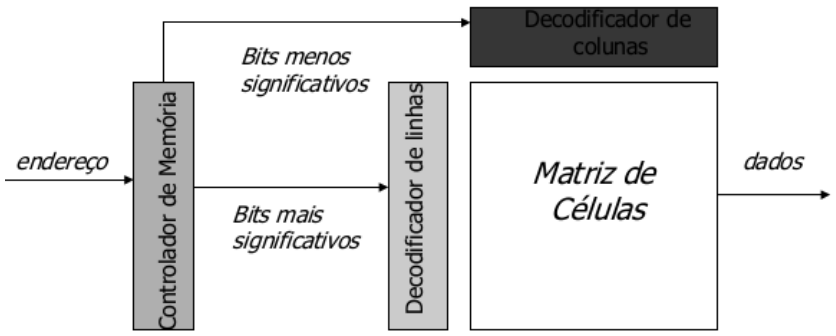


Figura 2.12: Esquema de acesso à memória DRAM

As leituras podem danificar o dado contido na célula. Por isso, as DRAMs necessitam de refreshes periódicos para mantê-los. Todos os bits de uma linha podem ser atualizados pela leitura desta linha. Quando há o *refresh*, a memória fica indisponível, um *overhead* que explica o porquê ela é usada mais em memóriais principais.

Para melhorar o desempenho dessas memórias, foca-se em aumentar sua largura de banda. Temos três frentes.

Aumento de Tamanho da Palavra
Aumentamos a largura de banda do barramento e logo, transferimos mais dados. Como o acesso da CPU ainda é de uma palavra, é necessário que haja um multiplexador entre a CPU e a *cache*.

Exemplo: o Alpha AXP usa barramentos de 256 bits para transferências entre memória RAM e *cache* L2.

Memória Entrelaçada
Os chips de memória podem ser organizados em bancos, permitindo que diversas palavras sejam lidas e escritas simultaneamente. Um único controlador de memória é utilizado para isso.

Uma boa função de mapeamento dos endereços para os bancos é crucial para o bom desempenho desta estratégia. A função de módulo normalmente é utilizada,o que favorece acessos sequenciais.

Bancos Independentes de Memória
Aqui existem múltiplos controladores de memória, que permitem que os bancos

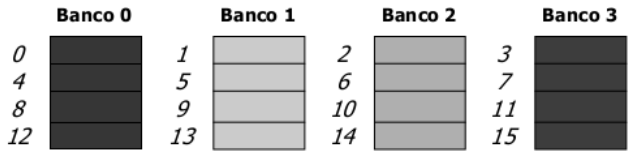


Figura 2.13: Memória entrelaçada 4-way

operem de maneira independente. Cada banco necessita de linhas de endereço separadas e, frequentemente, de barramentos separados.

2.3.2 Variações da DRAM

SDRAM
Utiliza um protocol síncrono para troca de dados com o processador, utilizando um relógio externo. O processador informa o endereço e o dado à SDRAM, a qual responde após um número fixo de ciclos de clock. Durante este período, o processador está liberado para outras tarefas.

Rambus DRAM
Utiliza um barramento de alta velocidade para trocas entre CPU e memória. Este barramento possui 18 linhas de dados, onde 16 são para o endereço e 2 para paridade. Possui sinais de clock para fazer transações síncronas no barramento.

Static DRAM
Utiliza de 4 a 6 transistores por bit, onde os valores binários são armazenados utilizando configurações de flip-flops. o SRAM não necessita de refreshes periódicos, mas ainda precisa de um fornecimento constante de energia.

SRAMs são mais rápidas, mais caras e menos densas que as DRAMs e são muito utilizadas em memórias *cache*.

Capítulo 3

RISC x CISC

Definição **Gap Semântico:** diferença entre as operações oferecidas por uma linguagem e as operações oferecidas pela arquitetura.

A cada vez que as linguagens de programação iam se modernizando, o gap semântico ia aumentando. Ele deu origem a diversas ineficiências na execução dos programas, tamanho excessivo de código binário e complexidade do compilador.

Afim de reduzir esses efeitos, foram projetadas arquiteturas contendo grande número de intruções, diversos modos de endereçamento e instruções complexas oriundas das linguagens de alto nível, surgindo assim as máquinas com conjuntos complexos de intruções, as CISC.

3.1 CISC

Com instruções mais complexas, o objetivo era simplificar o compilador e melhorar o desempenho. Como instruções de alto nível já são traduzidas para linguagem de máquina, por haver uma instrução correspondente, a tradução é mais simples. Porém, um conjunto grande e complexo de instruções traz problemas:

- O controle do mecanismos do pipeline é complicado;
- O compilador deve agora, ao analisar o código, saber quais instruções de alto nível podem ser diretamente mapeadas para as instruções complexas de máquina, o que nem sempre é fácil;
- Muitos compiladores acabam por deixar de usar instruções complexas em suas traduções

- O tamanho em bits das instruções em CISC acaba sendo maior que as de RISC;

Por isso, o código obtidos de arquitetura CISC são geralmente grande, ainda que visem serem pequenos, rápidos e menores que RISC. Os projetistas advogam que, com instruções maiores e mais complexas, o tempo de execução é reduzido. Porém a unidade de controle de máquinas CISC deve ser mais complexa, ocupando mais espaço, acabando por **aumentar o tempo de execução de uma única instrução**.

3.2 RISC

Em outra corrente, a eliminação do gap semântico é proposta com arquiteturas mais simples, as RISC. Aqui as instruções são mais simples, permitindo que uma instrução acabe em um ciclo de clock. É dada ênfase em operações entre registradores.

Essas arquiteturas são justificadas com base em estudos dos anos 80 com linguagens de alto nível, identificando que as operações mais abundantes são atribuições (passagem de dados) e desvios. Observou-se também que as variáveis acessadas eram em geral simples e locais, referenciado em média 0.5 operandos em memória e 1.4 registradores por instrução. Por fim, identificou-se que as procedures tinham chamadas menores que 6 parâmetros, usando menos que 6 variáveis locais. Logo não eram necessárias muitas palavras para ativar procedures.

Com base nisso, era mais interessante prover uma arquitetura que atacasse esses padrões de código mais utilizados, o que combinava mais com RISC, do que com arquiteturas que abarcassem aspectos mais gerais, o que se encaixava em CISC.

Como RISC veio como uma contra-proposta a CISC, diversas definições acabaram sendo atribuídas a RISC. Diz-se que "qualquer computador depois de 1985"é RISC. Porém, existem alguns pontos em comum.

Grande Conjunto de Registradores

Um número grande de registradores de propósito geral, com uso de compilador para otimizar seu uso. Como o número de atribuições observadas nos estudos eram grandes, os uso de vários registradores otimizava a execução.

Era necessária definir uma estratégia para manter os operandos mais utilizados em registradores e reduzir os acessos à memória. Daí, temos duas estratégias:

- **Por software:** análise do compilador para tentar alocar registradores às variáveis mais utilizadas durante um certo período;
- **Por hardware:** grande conjunto de registradores, permitindo que mais variáveis sejam postas na CPU.

Deste último, nasceu a **janela de registradores**. O conjunto grande de registradores reduz o acesso à memória, mas o projetista deve organizar o acesso de maneira eficiente.

Como a maioria das referências é a variáveis locais, a abordagem mais comum é armazenar essas variáveis em registradores, separando alguns para as variáveis globais.

Porém, variáveis locais são atreladas ao contexto de sua procedure. Na chamada de uma procedure-filha, as variáveis tem de ser salvas em memória para poderem ser utilizadas posteriormente. Ao terminar, teríamos que carregar as variáveis da procedure-pai novamente juntamente com os valores de retorno da filha.

Para resolver isso, as janelas de registradores, que são vários pequenos conjuntos destes, são utilizadas para cada procedure. Uma chamada a uma procedure-filha faz com que uma janela de registradores diferente seja utilizada ao invés de salvar tudo e memória e trabalhar com um conjunto novo. Com isso, procedures adjacentes podem possuir janelas sobrepostas, para facilitar a passagem de parâmetros. Em dado momento, somente uma janela de registradores é visível e endereçavel.

A janela é dividida em 3 regiões de tamanho fixo:

- Registradores de Parâmetros
- Registradores Locais
- Registradores Temporários

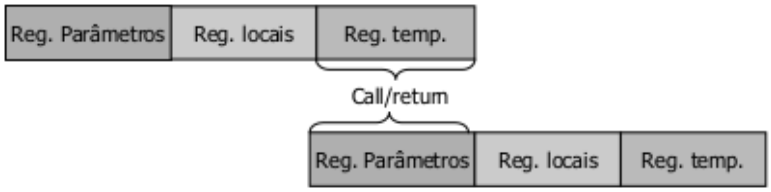


Figura 3.1: Esquema de uma janela de registradores. Perceba a sobreposição dos temporários e de parâmetros para permitir passagem de valores entre procedures

Como só existe um conjunto finito de janelas, somente os dados das procedures mais recentes são mantidos em registradores. Logo os dados de procedures antigas

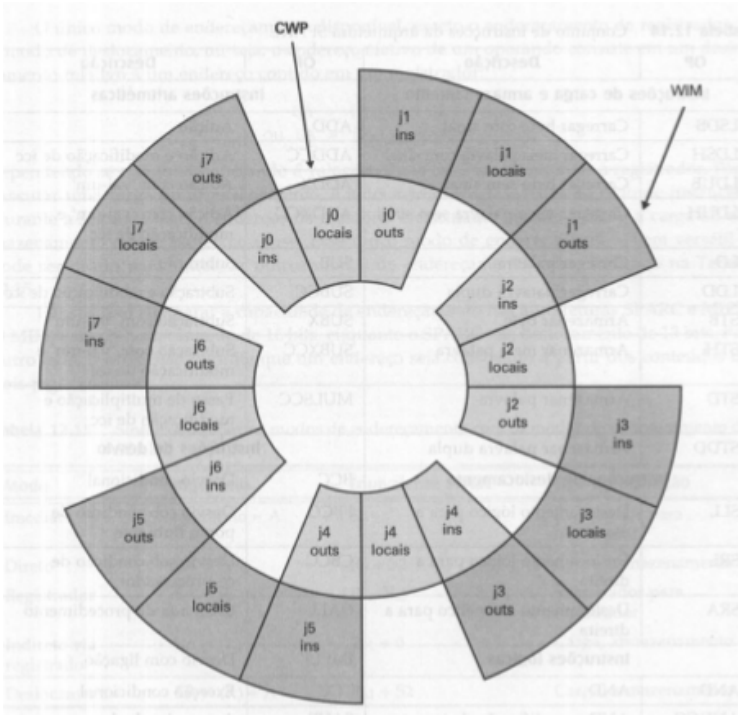


Figura 3.2: Oito janelas de registradores formando uma pilha circular. Presente em arquiteturas SPARC

ficam em memória e a organização real do *register file* é uma fila circular de janelas sobrepostas, como mostrado na 3.2.

Variáveis Globais

Se tratando de variáveis globais, inicialmente eram mantidas em memória, sugere-se mantê-las em um conjunto de **registradores globais** no processador. Estes registradores possuem número fixo e são acessíveis por todas as procedures.

Exemplo: podemos dispor os registradores da seguinte forma

- Registradores 0-7 contém variáveis globais
- Registradores 8-31: referem-se à janela corrente

Otimizações pelo Compilador

Como em programa escrito em linguagem de alto nível não faz referência a registradores, essa tarefa fica a cargo do compilador, que deve concentrar os acessos nos registradores, reduzindo `LOAD`s e `STORE`s.

Em geral, o compilador elege variáveis candidatas para residir em um registrador, chamadas ***quantity***, atribuindo-a à um registrador virtual. A idéia é mapear di-

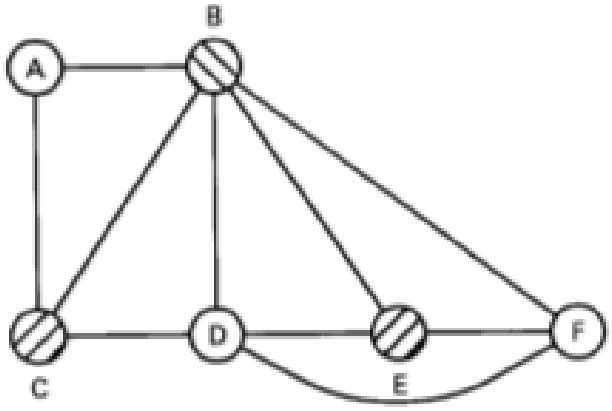


Figura 3.3: Exemplo de coloração de grafo. Os conjuntos $\{A, D\}$ e $\{B, C, E\}$ são mapeados em registradores reais. F é mapeado em memória.

versos registradores virtuais em um único registrador real, de modo que os virtuais não se sobreponham no tempo. Caso não haja número suficiente de registradores reais em um momento, o compilador recorre à memória.

Para aplicar esta técnica, são aplicados métodos de coloração de grafos. Primeiro o programa é analisado e um grafo de interferências de registradores é construído. O registradores virtuais são os nodos e as interdepêndencias são as arestas. Ao tentar colorir um grafo com n cores, onde n é o número registradores reais, mapeamos os nodos (virtuais) de mesma cor em único registrador real.

Conjunto Pequeno de Instruções

O conjunto de instruções deve ser projetado de maneira que a grande maioria das **instruções sejam executada em um ciclo de máquina**, acelerando a execução de programas. A maioria das **instruções deve ser entre registradores, com operações de `LOAD` e `STORE` simples**, o que simplifica as unidades de controle.

A maioria das arquiteturas RISC oferece modos mais simples de endereçamento, geralmente menos que 5. Modos menos comuns de endereçamento, como o indireto e o indexado, podem ser sintetizados via software. Os mais comuns são:

- Endereçamento de registradores
- ENdereçamento relativo ao PC
- Deslocamento

Não são usados modos de endereçamento que combinam load/store com operações aritméticas e somente um operando é endereçado a cada instrução. Logo, o formato de instruções são simples, o que permite:

- O tamanho da instrução é fixo e alinhado por *word*, com geralmente 4 bytes;
- A localização do código da operação é conhecida e de tamanho fixo. Ainda, a decodificação e buscas de operandos pode ser feita simultaneamente.

O compilador é uma parte fundamental para a obtenção de um bom código. Além disso, o desenvolvimento de chips RISC são mais rápidos, já que o número de instruções é simples e reduzido.

Otimizações no pipeline de instruções

Como arquiteturas RISC possuem instruções simples e regulares, acaba por ser mais fácil otimizar o *hardware* e, consequentemente, no pipeline de instruções.

3.3 RISC x CISC

Observa-se que os projetos de arquitetura CISC vem empregando algumas características de RISC e vice-e-versa.

Como exemplos, as atuais arquiteturas do PowerPC, que é classificada como RISC, empregam implementações de CISC. O Pentium II, apesar de ser classificado como CISC, incorpora diversas características de RISC.

Process.	# tam inst	Tam inst (bytes)	# modo ender.	Ender. Indir.	Load/st + aritm	Max oper	End nao alinhado
AMD2900	1	4	1	Não	Não	1	não
MIPSR2000	1	4	1	Não	Não	1	não
SPARC	1	4	2	Não	Não	1	não
HP/PA	1	4	10	Não	Não	1	não
RS/6000	1	4	4	Não	Não	1	sim
IBM 3090	4	8	2	Não	Sim	2	sim
80486	12	12	15	Não	Sim	2	Sim
NSC32016	21	21	23	Sim	Sim	2	Sim
MC68040	11	22	44	Sim	Sim	2	Sim
VAX	56	56	22	Sim	Sim	6	Sim

Figura 3.4: Tabela comparativa de RISC (entradas escuras) X CISC (claras)

Capítulo 4

Instruction Level Parallelism

Definição o grau no qual instruções de um mesmo programa podem ser avaliadas em paralelo, ou seja, **a medida de sobreposição potencial entre as instruções**.

Note que: O pipelining é uma das técnicas que explora ILP e é geralmente a base para as demais otimizações propostas.

Um programa pode ser definido como a composição de dois tipos de instruçoes:

- **Atribuições**, da forma $A = B + C$;
- **Branches**, resultando em desvios no fluxo de execução das instruções (condicionais e laços).

O código executado entre dois *branches*, i.e. um bloco básico ou *branch path*, é sequencial. Logo, ele é o candidato óbvio para a exploração de paralelismo. No entanto, estudos mostram que tipicamente, o tamanho deste bloco é pequeno, indo de 3 a 0 instruções. Além disso, depêndencias existentes dentro deste bloco acabam por limitar o paralelismo.

Dessa forma, uma técnica efetiva de exploração da ILP deve ser capaz de explorar o paralelismo entre blocos distintos. Os candidatos mais fáceis para tal são as interações de um *loop* - o *loop level parallelism*.

Exemplo: aqui, cada iteração do *loop* pode ser executada simultaneamente, apesar de haver a dependência de dados no interior de cada iteração.

Tarefas do Compilador

Para manter um pipeline cheio, o paralelismo entre as instruções deve ser explorado de maneira a obter sequências de instruções independentes que possam ser sobrepostas no pipeline. Duas sequências dependentes devem ser separadas por uma distâncias que seja igual à latência do pipeline para a primeira instrução, em ciclos de clock.

A habilidade do compilador de separar instruções está limitada por dois fatores:

- A quantidade de ILP presente no programa
- As latências das unidades funcionais do pipeline

Quando e Como Explorar o ILP

Existem diversas técnicas para explorar ILP, onde a maioria necessita que se saiba quando e como a ordem das instruções de um programa pode ser alterada sem que sejam produzidos resultados incorretos.

Deve-se então saber a dependência entre as instruções para se determinar uma maneira de reordená-las sem que as dependências sejam violadas.

4.1 Depêndencias

O grau de dependência entre as instruções determina quanto paralelismo existe em um programa e como o mesmo pode ser explorado. **Duas instruções serão paralelas quando podem ser executadas simultaneamente em um pipeline, sem a necesidade de atrasos.** Se duas instruções não são paralelas, então elas **são dependentes** e logo não podem ser reordenadas.

Dependências são propriedades de programas e a presença delas indica o potencial para atrasos no pipeline. Porém, qual atraso e seu impacto é uma propriedade do pipeline.

4.1.1 Dependências de Dados

Definição quando uma instrução depende do resultado de uma instrução específica ou do resultado encadeado de instruções, então temos uma dependência de dados. Também chamada de dependência real.

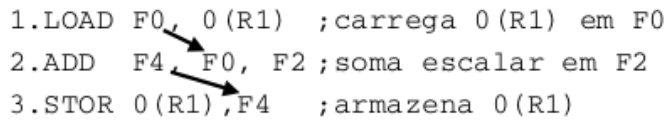


Figura 4.1: Aqui, temos uma dependência encadeada: a instrução 3 depende da 2, que depende da 1

A existência de dependência de dados implica na existência de uma cadeia de *hazards* RAW entre as instruções. Caso as instruções dependentes de dados sejam escalonadas simultaneamente, o pipeline será atrasado.

Uma dependência de dados possui as seguintes características:

- Indica a possibilidade de um atraso no pipeline;
- Determina em que ordem as operações devem ser executadas;
- Determina um limite superior na quantidade de paralelismo que pode ser explorada

Pelo fato delas limitarem a ILP, elas devem ser tratadas. Temos duas maneiras:

- **Manter a dependência**, evitando o atraso, o que pode ser feito pelo pipeline ou pelo compilador;
- **Rearrumar o o código** de maneira a eliminar a dependência, feito apenas pelo compilador

Um valor de dado pode fluir entre instruções através de registradores ou posições de memória. **Por um registrador, a detecção de dependência é mais simples, pois os nomes são fixos**. As dependências que ocorrem através de posições de memória possuem um tratamento mais difícil.

Exemplo: 100(R2) e 20(R4) podem endereçar a mesma posição de memória, mas é dificil o compilador saber disso. Logo, ele toma o caminho mais conservador que é manter a dependência.

4.1.2 Dependências de Nomes

Definição quando não há valor transferido entre instruções, mas ambas usam o mesmo nome (de registrador ou posição de memória) e pelos menos uma delas altera o valor deste nome, temos uma dependência de nome. Temos dois tipos:

- **Anti-Dependência:** ocorre quando uma instrução lê o valor de um nome e uma outra instrução quer escrever neste mesmo nome. Ou seja, quando **temos um WAR**. Exemplo:

```
ADD R1, R1, R3
LOAD R3, 0(R4)
```

- **Dependência de Saída:** ocorre quando duas instruções querem escrever no mesmo nome. Ou seja, quando **temos um WAW**. Exemplo:

```
LOAD R1, 0(R2)
ADD R1, R1, 10
```

Como não há valor sendo transmitido entre instruções, as dependências de nomes são mais fáceis de serem eliminadas se o nome utilizado for alterado, de maneira a remover o conflito. Chamamos isso **register renaming**. Esta renomeação pode ser feita tanto por *hardware* como por compilador.

4.1.3 Dependências de Controle

Definição quando o ordenamento de uma instrução está atrelado a um *branch*, temos uma dependência de controle.

Toda a instrução, exceto as pertencentes ao primeiro bloco básico, estão atreladas a algum conjunto de *branches* e logo é dependente de controle destes. No caso geral, as dependências de controle devem ser preservadas.

Exemplo: vemos abaixo que `s1` é dependente de `p1`. Já `s2` é dependente de `p2`, mas não de `p1`

```
if (p1) {
s1
}
if(p2){
s2
}
```

Essas dependências geram duas restrições:

- Uma instrução que é dependente de controle de um branch não pode ser movida para antes deste branch;
- Uma instrução que não é dependente de controle de um branch não pode ser movida para depois do branch, pois caso fosse, sua execução seria então

controlada por ele.

Em um pipeline básico, as dependências de controle são preservadas por dois motivos:

- As instruções são executadas em ordem;
- Uma instrução que é dependente de um branch não é executada até que a direção do branch seja conhecida, devido aos hazards de controle.

As propriedades a seguir são preservadas pela dependência de controle afim de garantir a correção do programa.

Comportamento em Face a Exceções

O reordenamento da execução de das instruções não deve permitir o surgimento de exceções no programa. Por isso, uma postura conservadora é adotada e o reordenamento não é feito, mesmo que não haja dependência de dados.

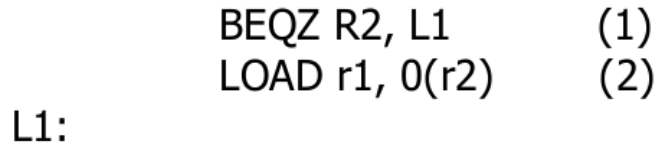


Figura 4.2: Se ignoramos a dependência de controle entre 1 e 2, se a instrução 2 é executada antes do branch, podemos levar a um estado de erro, o que não ocorreria se o branch fosse tomado

Fluxo de Dados

Definição o fluxo real de dados entre as instruções que os produzem e as que os consomem.

O branches fazem com que o fluxo de dados seja dinâmico, permitindo que o dado utilizado em determinada instrução possa vir de vários pontos. Logo, o reordenamento não pode afetar este fluxo.

4.2 Paralelismo a Nível de *Loop*

A maneira mais simples de se detectar paralelismo é no interior de *loops*. A análise de paralelismo neste nível consiste em se determinar se acesso a dados em iterações

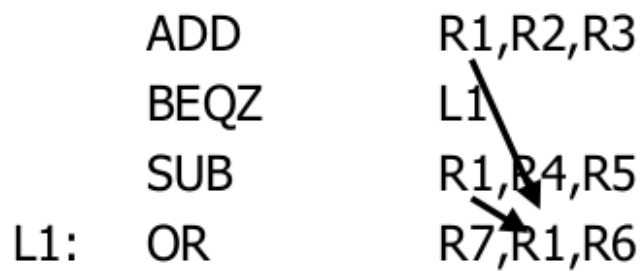


Figura 4.3: O valor de R1 em 4 vem de 1 se o desvio for tomado e de 3 se não for

```
for (i=0; i<1000; i++)
  x[i]=x[i]+s;
```

(a) Aqui a dependência ocorre entre os dois usos de `x[i]`, sendo interna a cada iteração. Entre *loops* não há dependencia e eles são paralelos

```
for (i=0; i<1000; i++)
  x[i+1]=x[i]+s; /*s1*/
```

(b) Aqui, `x[i+1]` e `x[i]` dependem, criando dependência entre os laços, obrigando-os a serem executados em ordem. Chamamos esse tipo de **loop-carried dependencie**

```
for (i=0; i<1000; i++){
  a[i]=a[i]+b[i]; /*s1*/
  b[i+1]=c[i]+d[i]; /*s2*/
}
```

(c) Aqui a dependência é entre `s1` e `s2`, pelo fato de `b[i]` depender de `b[i+1]`. Mas note que `s2` não depende de `s1`

Figura 4.4: Avaliações de paralelismo a nível de *loop*

posteriores são dependentes de valores de iteração anteriores, i.e., se i depende de $i-1$. Geralmente, esta análise é feita no código fonte, pois a tradução pra a linguagem de máquina cria dependências no registrador de incremento de índice. As Figuras 4.4a, 4.4b e 4.4c refletem bem isso.

Um loop é paralelo se puder ser escrito de maneira que não haja dependência circular entre as dependências *loop-carried*. Note no exemplo da Figura 4.4b que a dependência é circular, pois `s1` depende dela própria.

No exemplo da Figura 4.4c, não há dependência circular, pois `s2` não depende de `s1` (mas `s1` depende de `s2`). Logo, podemos reescrevê-lo afim de se tornar paralelo. A Figura 4.5 mostra o conserto.

```
a[0]=a[0]+b[0];
for (i=0; i<999; i++){
  b[i+1]=c[i]+d[i]; /*s2*/
  a[i+1]=a[i+1]+b[i+1]; /*s1´*/
}
b[1000]=c[999]+d[999];
```

Figura 4.5: Agora o loop pode ser paralelo. Semelhante ao exemplo na Figura 4.4a

4.3 Semântica e Notações de Desvios

Em desvios condicionais, se a condição for verdadeira, o controle é transferido para a instrução destino do branch - a *branch target address* - e neste caso é dito **tomado (*taken - T*)**. Se a condição é falsa, a execução continua com a instrução imediatamente posterior ao *branch* e o desvio é dito **não tomado (*not taken - NT*)**.

O desvio pode ser *backwards*, indo para trás ou *forward*, indo para frente.

4.3.1 Previsões

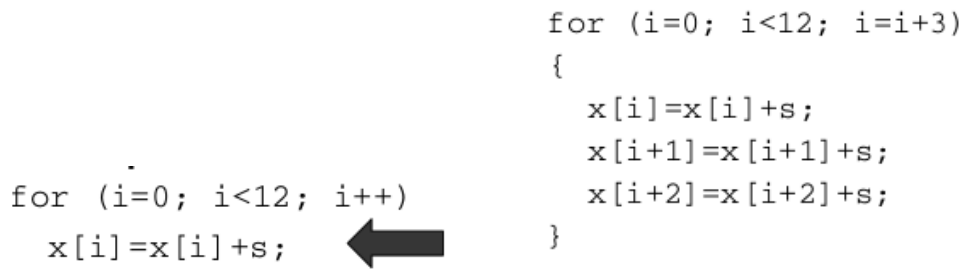
Os sinais dos desvios são geralmente preditos por esquemas de previsão de desvio e medimos sua acurácia: *branch prevision accuracy*, ou BPA. O caminho predito é o *predicted path* e o caminho cuja previsão diz que não será tomado é o *not predicted path*.

Definição **Janela de execução do processador:** parte do código que está em processo de execução na CPU

Um desvio é geralmente previsto depois que entra na janela de execução. Após a previsão, o desvio fica **pendente**. Quando a condição do desvio é efetivamente avaliada, o branch é dito **resolvido**.

Definição **Código estático:** código na ordem de escrita pelo programador ou compilador.

Definição **Código dinâmico:** diferentes instâncias de instruções estáticas, que surgem durante a execução, ordenadas pelo tempo no qual foram executadas.



(a) Loop paralelo, com bloco básico pequeno

(b) Bloco básico estendido e número de iterações menor

Figura 4.6: Exemplo de *loop unrolling*

4.4 Técnicas de Exploração

4.4.1 *Loop Unrolling*

Definição aumento do bloco básico entre as iterações de um *loop*.

É uma das técnicas mais simples para explorar ILP e implica em um rearrumação do código, normalmente feita pelo compilador. Aplicamos tal técnica em loops independentes, ou seja, loops paralelos. As Figuras 4.6a e 4.6b explicam bem.

Apesar do *loop unrolling* aumentar o tamanho do código do programa, esta transformação é bastante utilizada pois permite o uso mais eficiente do pipeline.

4.4.2 Escalonamento Dinâmico

Permite que o *hardware* reordene a execução das instruções de maneira a reduzir os atrasos no pipeline. Como a reordenação é feita em tempo de execução, o escalonamento é dinâmico. Ele pode ser combinado com o escalonamento estático.

Como **vantagens**, podemos tratar situações onde as dependências não são conhecidas em tempo de compilação e permite que o código que foi compilada para um tipo de pipeline rode eficientemente em outro tipo. Como **desvantagens**, temos o aumento da complexidade de *hardware*.

O pipeline básico faz a busca de instruções na ordem do programa. Neste caso, se uma instrução atrasa o pipeline - ou seja, geral um *stall* - nenhuma outra instrução pode prosseguir.

```
d=1; /*s1*/
a=d; /*s2*/
e=3; /*s3*/
f=5; /*s4*/
```

Figura 4.7: A instrução `s2` necessita de `s1`. Porém, `s3` e `s4` são atrasadas, mesmo não dependendo de `s1` e `s2`. Podemos eliminar isto se executarmos fora de ordem

A execução fora de ordem permite que as instruções possam começar sua execução no momento em que seus operando estão disponíveis. Ela também implica que o término das instruções também será fora de ordem, o que pode gerar erros, complicando o tratamento de exceções.

A execução fora de ordem divide o estágio ID em duas partes:

- **Issue:** decodifica as instruções e verifica hazards estruturais
- **Read Operands:** espera até que não haja hazards de dados e lê operandos.

A execução de uma instrução se inicia no instante t e termina no instante v, onde $v > t$. Em um pipeline com escalonamento dinâmico, todas as instruções entram no estágio *issue* na ordem do programa. Entretanto, no estágio *read operands*, elas podem ser atrasadas ou fazerem um *bypass* das outras instruções - o que é a própria execução fora de ordem.

4.4.3 Scoreboarding

Técnica que permite que as instruções sejam executadas fora de ordem, quando há recursos suficientes e independência de dados. O *scoreboarding* é o elemento principal desta técnica, pois ele controla a execução das instruções e faz a detecção de *hazards*.

Como várias instruções podem estar no estágio EX simultaneamente, o *scoreboarding* **só é efetivo quando há múltiplas unidades funcionais**.

Toda a instrução que está ou já passou no estágio *issue*, mas não foi terminada, possui um entrada no *scoreboarding*. Ele mantem todas as informaçoes necessárias para detectar as dependências de dados.

O *scoreboarding* determina o instante no qual a instrução pode ler os operando e iniciar a execução. Caso não seja possível, ele monitora toda mudança no *hardware* para determinar quando uma instrução pode ser executada. Por isso, ele controla também quando o resultado pode ser escrito no estágio WB.

Suas etapas são:

- **Issue:** se a unidade funcional destino estiver livre e nenhuma instrução ativa tiver o mesmo registrador destino, o scoreboard faz um issue da instrução. Essas garantias previnem *hazards* WAW.
- **Read Operands:** o scoreboard verifica se os operandos estão disponíveis, ou seja, se nenhuma instrução anterior vai escrever neles. Caso estejam disponíveis, o scoreboard instrui a unidade funcional para a leitura dos mesmos e o início da execução;
- **Execution:** ao receber os operandos, a unidade funcional inicia a execução. Quando o resultado final estiver pronto, o scoreboard é notificado;
- **Write Result:** o scoreboard recebe a notificação e verifica a existência de hazards. Uma instrução i é impedida de escrever seu resultado quando existe uma instrução j, precedente a i, que não leu os operandos e um dos operandos de j é o resultado de i. Ou seja, uma instrução é impedida de escrever seu resultado quando este é o operando de uma outra instrução anterior, que ainda não leu os operandos.

A estrutura de dados do *scoreboarding* possui 3 partes:

- **Estado da Instrução:** indica em qual estágio cada instrução está
- **Estado da Unidade Funcional:** possui 9 campos para cada unidade funcional. Eles são: busy, operação, registrador destino (Fi), registradores fonte (Fj,Fk), unidades funcionais que produzem os registradores fonte (Qj,Qk) e flags indicando se os registradores fonte estão prontos (Rj,Rk);
- **Estado do Registrador Resultado:** indica a unidade funcional que escreverá cada registrador

O *scoreboarding* apresenta algumas **deficiências**:

- Necessitava de um grande número de barramentos
- A escolha das instruções se resumia a um mesmo bloco básico, o que não trás grande paralelismo.

4.4.4 Tomasulo

Combina o *scoreboarding* com renomeação de registradores. Possui unidades chamadas **estações reservas**. Elas servem para armazenar os operando das operações que estão esperando o *issue*, buscando-os logo que os mesmo estiverem disponíveis.

Quando várias escritas são feitas para o mesmo registrador, somente o último é utilizado. Note que os valores anteriores ao último não importam para instruções após a escrita do último. Quando operações são iniciadas, os registradores que contem operando pendentes são renomeados para os nomes das estações reservas, uma vez que elas contém os valores anteriores desse registrador. **Isso previne *hazards* WAR e WAW**. Como pode haver mais estações reserva do que registradores reais, esta técnica **pode eliminar mais hazards do que o compilador poderia fazer**.

Cada unidade funcional possui estações reservas próprias, que controlam quando uma instrução pode ser executada naquela unidade, o que garante um controle descentralizado. Os valores dos operandos são obtidos pelas unidades funcionais através de barramentos compartilhados. Isso permite que todas as unidades possam obter o operando simultaneamente. *Loads* e *stores* são tratados como unidades funcionais.

As estações reservas acabam por guardar:

- As instruções esperando por execução;
- Os operandos destas instruções já calculados ou então a fonte destes;
- Informações de controle da execução da instrução.

Temos os seguintes estágios nesta arquitetura:

- **Issue:** retira uma instrução da fila de operações de ponto flutuante. Se há estação reserva livre, inicia a instrução e envia os operandos disponíveis para a estação reserva. Loads e stores podem prosseguir se houver um buffer disponível. Faz também a renomeação dos registradores
- **Execute:** se algum operando não estiver disponível, monitora o CDB. Quando todos os operandos estiverem disponíveis, recupera-os da estação reserva e executa a operação, se não houver hazards RAW;
- **Write Result:** quando o resultado estiver disponível, escreve-o no CDB e também em registradores ou estações reserva que estiverem esperando.

A estrutura de dados do Tomasulo possui 6 campos:

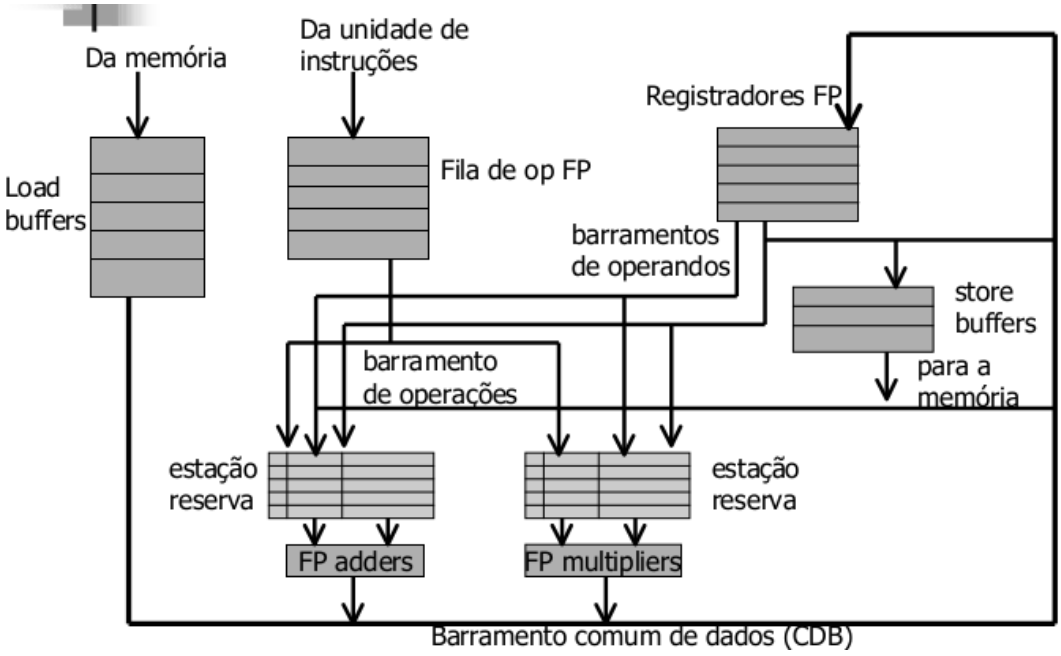


Figura 4.8: Arquitetura do Tomasulo

- *Op*: operação a ser executada
- Q_i, Q_j: estações reserva que produzirão o operando fonte, onde o valor 0 indica que o operando já está disponível ou não é utilizado;
- V_i, V_j: valor dos operandos i e j;
- **Busy/Free**

Em termo de **vantagens** o Tomasulo provê um hardware distribuido para detectar *hazards* RAW e elimina os *hazards* WAW e WAR, através da renomeação de registradores nas estações reservas.

Como **desvantagens**, temos um necessidade de hardware complexo, o barramento comum dos dados (CDB) pode se tornar um gargalo e não trata o problema dos desvios.

4.4.5 Previsão Dinâmica de Desvios

A frequência dos *branches* torna necessária a utilização de técnicas para reduzir os atrasos potenciais causados por estes desvios, sendo as dependências de controle um grande limitador para a ILP. Nos esquemas de previsão estática de desvios,

o comportamento em tempo de execução de cada desvio não é levado em conta. Toda decisão recai sobre o compilador.

A previsão dinâmica de desvios usa o hardware para analisar comportamentos passados do branch, prevendo seu próximo resultado. A **efetividade de um esquema de previsão de branches** depende de sua acurácia e do custo de uma previsão correta e de uma previsão incorreta.

Normalmente, os previsores se baseiam em uma tabela de histórico de desvios, chamada *branch history table*, ou **BHT**. Ela é uma pequena memória, indexada pela porção menos significativa do endereço da instrução de branch, que contem um bit indicando se o desvio foi recentemente tomado ou não.

Previsão One-bit

O esquema mais simples de previsão, o qual utiliza a BHT. Prevê que o branch sempre vai ser executado da maneira que foi executado na iteração anterior. Logo, o diagrama de estados segue o mostrado na Figura 4.9. Se o *branch* é tomado, a busca instruções continua a partir do endereço do *target* do desvio.

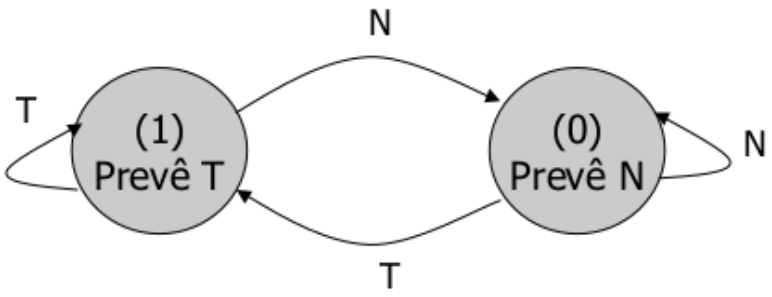
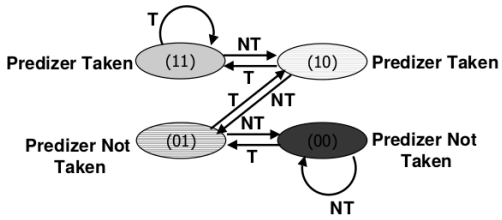


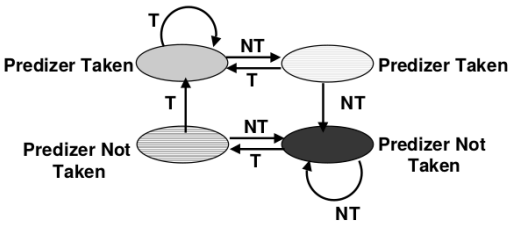
Figura 4.9: Autômato da Previsão One-Bit

A previsão é uma dica sobre a direção do desvio e a carga de instruções começa a ser prevista. Se a dica estiver errada, o bit de previsão é invertido. Idealmente, deve haver um autômato por *branch* estático. Possui acurácia de 77-79% e é usado no DEC Alpha 21064, com 2K autômata.

Este esquema tem problemas com loops, onde geralmente ocorrem 2 previsões incorretas.



(a) Two-Bit Saturada



(b) Two-Bit Patterson

Previsão Two-bit

Foi proposto para sanar a deficiênia do esquema de 1 bit, onde agora para haver a alteração de estado, a previsão deve ser incorreta por duas vezes, no mínimo. É possível haver esquemas n-bits, onde o de 2-bits faz parte. Um contador é incrementado quando o desvio for tomado e decrementado quando não for. Quando este for maior que $\frac{(2^n-1)}{2}$, a previsão do desvio é T. Caso contrário, é NT.

A acurária aqui é de de 78-89% (SPECInt89) e é utilizado no NexGen 586 (2K automata) e no Pentium (256 automata).

Previsores Correlatos (ou de dois níveis)

Muitas vezes, o comportamento de um branch está relacionado com o comportamento de outro branch. Por isso, os previsores correlatos se baseiam nos últimos branches executados, mesmo que estes não sejam o branch que está sendo previsto. A previsão é feita em dois níveis, com duas estruturas correspondentes:

- ***branch history register* - (BHR):** guarda o história de execução dos desvios. Cada vez que uma instância dinâmica de um branch for resolvida, é feita uma operação de shift a esquerda com seu sinal no BHR. É usado como índice na BPT;

- ***branch pattern table* - (BPT):** armazena um autômato de 2-bits para cada padrão possível no BHR.

Esta abordagem possui acurácia de 94.2% para um BHR de 8 bits e uma BPT de 16 entradas. É usado no NexGen Nx686 e no Pentium Pro (P6).

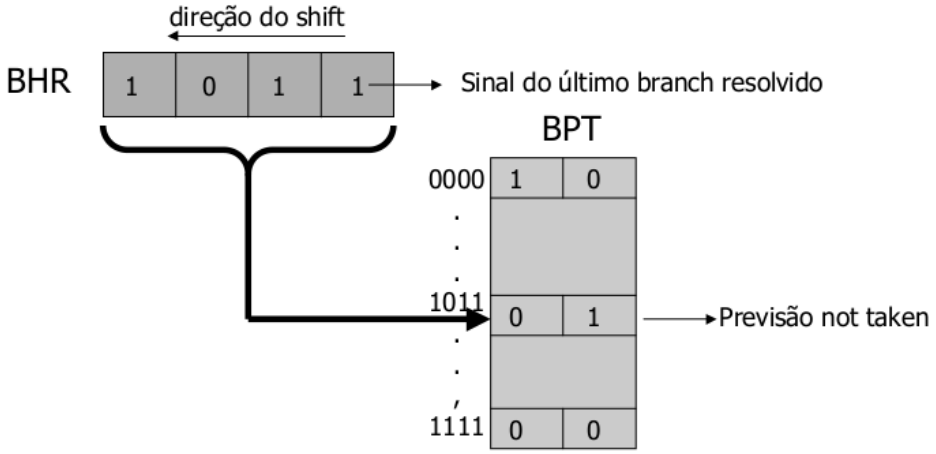


Figura 4.11: Indexação da BHR e BPT na Previsão Correlata

Branch Target Buffer

Além da previsão da direção, é necessário o endereço da instrução prevista para a execução. Por isso, o *branch target buffer* opera como **uma cache que guard o endereço previsto da próxima instrução a ser executada após o branch**. Se o PC de uma instrução a ser executada é igual a um PC no BTB, o PC previsto é utilizado como o próximo PC.

Só é necessário guardar os *branches* previstos como *taken*, já que os *branches* não tomados se comportam como as instruções sequenciais.

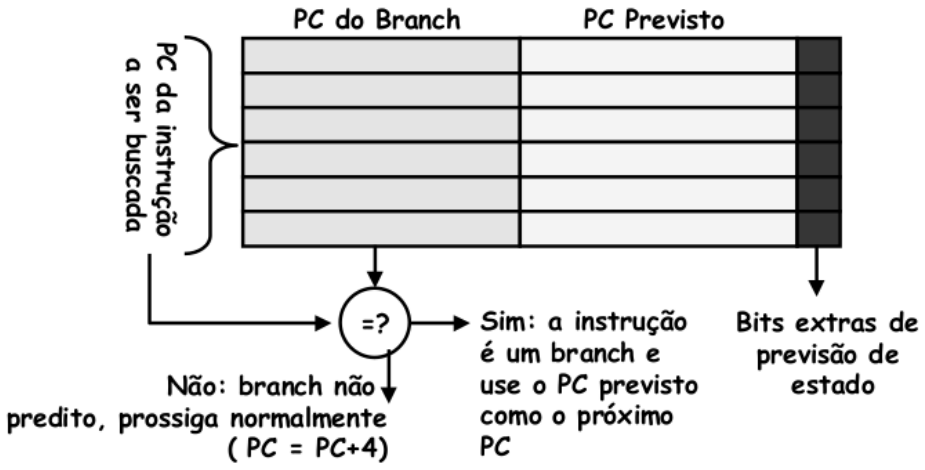


Figura 4.12: Esquema da previsão dinâmica de buffer, pelo Patterson

4.4.6 Execução Especulativa

A ideia aqui é executar condicionalmente código que está depois de um branch, antes que o branch seja resolvido, o que involve **a execução do código antes que se tenha certeza de que o mesmo deve ser executado**. É usada em conjunto com a previsão de desvios.

Se a previsão estiver correta, o potencial do ILP é maior do que um caminho do branch. Por isso, o potencial de paralelismo aqui é limitado por duas previsões incorretas (ou *mispredictions*) de direções do branch. A execução especulativa pode ser feita por hardware ou software e temos três tipos, vistos a seguir.

Single Path

Neste caso, quando a CPU encontra um branch, o sinal do branch é predito e a execução prossegue pelo caminho predito. Enquanto o branch não for resolvido, todas as escritas a registradores e memória e todas as operações de I/O devem ser feitas condicionalmente

Estas operações somente serão finalizadas (*committed*) quando todos os branches especulados tiverem sido previstos corretamente. Se houver uma previsão incorreta, as operações não serão finalizas (*squashed*).

O custo em hardware do *single path* é pequeno e cresce linearmente com o número de branches pendentes. Por isso, é o tipo de execução especulativa mais utilizado nos microprocessadores atuais.

Eager Execution

Aqui, a execução prossegue pelos dois caminhos do branch, ou seja, não há previsão e sim a execução das duas situações possíveis. Quando um branch é resolvido, todo estado dos caminhos não tomado é descartado.

Sob recursos ilimitados, essa execução oferece o melhor desempenho. Mas com tamanha taxa de descarte e desperdício, é um algorítmo não realizável.

Disjoint Eager Execution

É uma combinação das duas técnicas anteriores, onde os recursos são atribuídos aos branches que tem mais probabilidade de serem executados, considerando todos

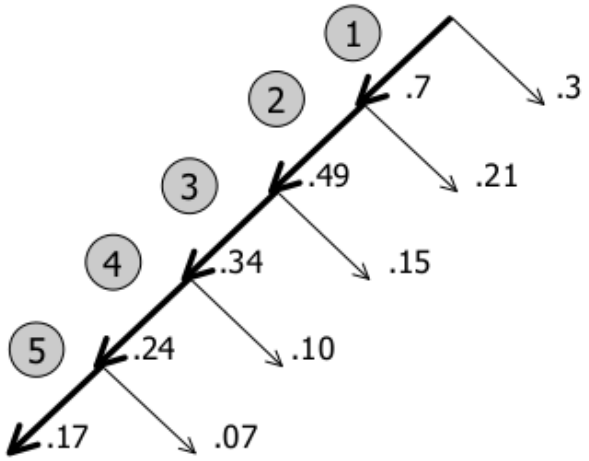


Figura 4.13: Exemplo de execução no Simple Path

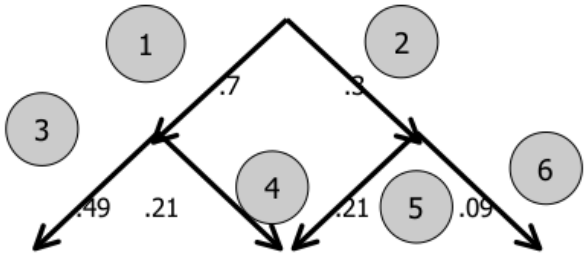


Figura 4.14: Exemplo de execução no Eager Execution

os branches pendentes. Ou seja, é considerada a probabilidade cumulativa de execução.

Todos os branches são preditos e alguns deles também são executados agressivamente, i. e. os dois caminhos são executados. O custo deste esquema aumenta como uma fração da raiz quadrada do tamanho do caminho principal.

4.4.7 Arquiteturas Superescalares

Definição Arquiteturas que permitem que diversas instruções sejam iniciadas em um único ciclo de clock e executadas de maneira independente.

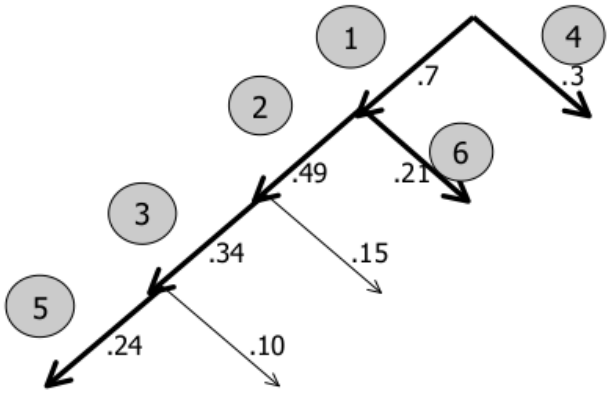


Figura 4.15: Exemplo de execução no Disjoint Eager Execution

Para que isso seja possível, deve-se ser capaz de buscar mais de uma instrução por ciclo de clock e executar diversas instruções simultaneamente. Entretanto, isso trás uma série de limitações:

- **Dependências**, uma vez que instruções executadas paralelamente podem depender uma da outra;
- **Conflitos estruturais**, onde instruções irão utilizar as mesmas estruturas;
- **Tempos diferentes de execução das instruções**, podendo causar descompassos entre instruções dependentes;
- **Hardware bastante complexo**, dado que a complexidade do hardware depende do número de instruções a serem executadas simultaneamente.

O conceito do projeto está muito associado a estruturas RISC, porém pode ser perfeitamente aplicado a estruturas CISC. O Pentium 4 é um exemplo clássico de CISC com arquitetura superescalar. Nele o processador busca as instruções em memória na ordem do programa estático. Porém cada instrução é traduzida em uma ou mais operações RISC de tamanho fixo, chamadas micro-operações. O processador as executa em um pipeline escalar e faz o *commit* de cada uma delas, restaurando o fluxo original do programa. A Figura 4.18 a arquitetura geral do Pentium 4.

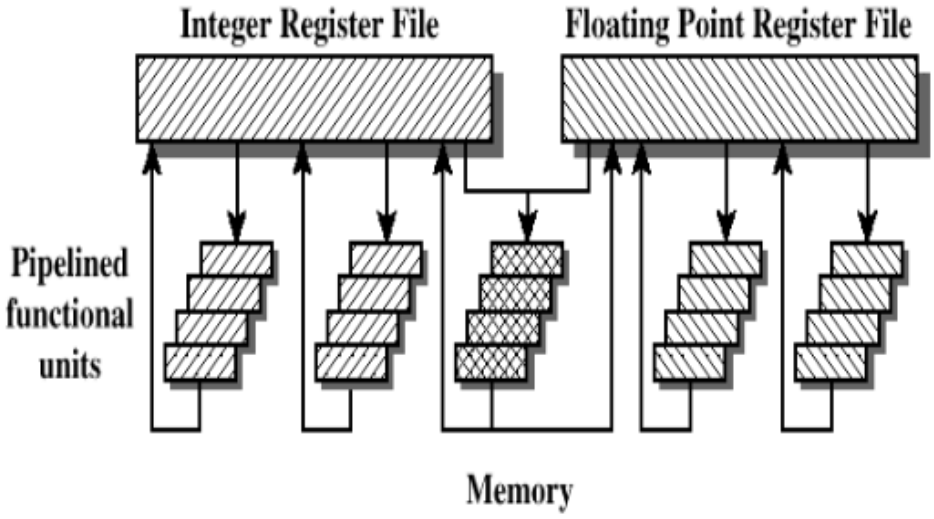


Figura 4.16: Organização geral de uma Arquitetura Superescalar

	1	2	3	4	5	6	7	8	9	10
Instrução 1	IF	ID	EX	Mem	**WB**					
Instrução 2	IF	ID	EX	Mem	**WB**					
Instrução 3		IF	ID	EX	Mem	**WB**				
Instrução 4		IF	ID	EX	Mem	**WB**				
Instrução 5			IF	ID	EX	Mem	**WB**			
Instrução 6			IF	ID	EX	Mem	**WB**			

Figura 4.17: Simulação de execução de instruções em Arquitetura Superescalar

4.4.8 Very Large Instruction Word - VLIW

Definição arquitetura que combina diversas operações em uma única instrução muito longa. A tarefa de se determinar quais instruções que serão emitidas simultaneamente, entretanto, recai sobre o compilador.

Neste caso, também teremos diversas unidades funcionais independentes. Uma instrução VLIW poderia incluir 2 operações inteira, 2 de ponto flutuante, 2 referências à memoria e um branch, por exemplo. Dessa forma, se cada unidade funcional tem 16 bits, teríamos um palavra de 112 bits. Note que **deve haver paralelismo suficiente na sequencia de código para manter todas as unidades ocupadas**.

Como a emissão de instruções e escalonamento é feito pelo compilador, essa ar-

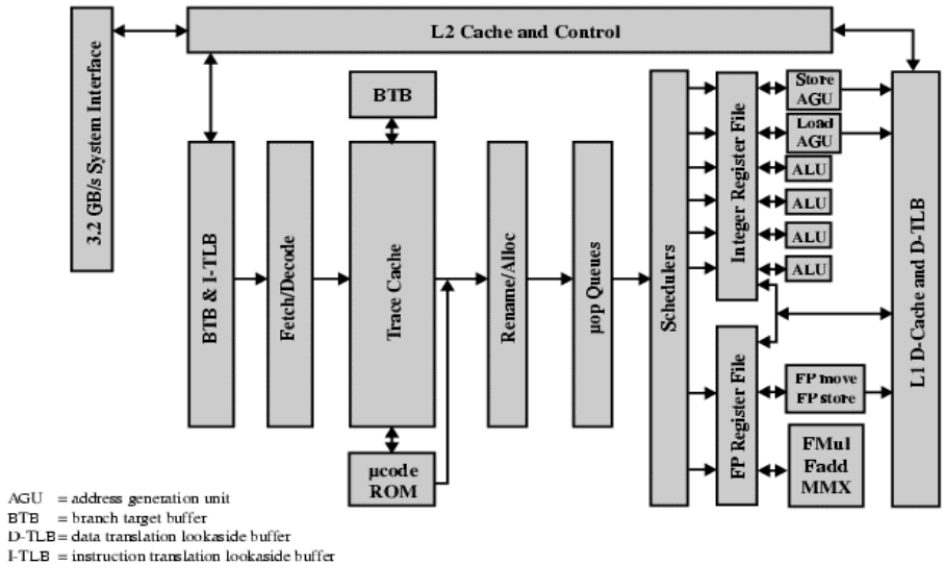


Figura 4.18: Arquitetura geral do processador do Pentium 4

quitetura **requer pouco ou nenhum *hardware* adicional**. Entretanto, essa abordagem trás alguns problemas:

- **Demandam grande número de portas** sendo uma para cada unidade funcional. Isso aumenta o uso do material - silício - e degrada a velocidade do clock.
- Como há múltiplos acessos simultâneos à memória, **a complexida da memória também aumenta**;
- **O tamanho do código aumenta bastante**, apesar de ser possível a implementação de grande quantidade de *loop unrolling*;
- Quando as instruções não são totalmente cheias, **temos espaço disperdiçado na VLIW**;
- Como as operações são emitidas de maneira síncrona, **um atraso em qualquer unidade funcional, representa um atraso no processador como um todo**

4.4.9 Análise de Dependência pelo Compilador

A análise de dependências é fundamental na exploração do paralelismo, uma vez que a detecção destas em um programa é importante para:

- Determinar um bom escalonamento de código
- **Determinar os loops que contém paralelismo**, uma vez que essa propriedade existe quando o *loop* não for *loop-carried*. Note que, entre tanto, *loop-carried* não circulares podem ser eliminadas. As verdadeiras limitantes de paralelismo são as dependências *loop-carried* circulares.
- Eliminar dependências de nomes

Vemos que a determinação da existência de dependências, no caso geral, é um problema NP-Completo. Porém, existem testes exatos para situações restritas que podem ser aplicados, sendo um **teste considerado exato quando ele determina precisamente que a dependência existe**.

Além de detectar dependências, o compilador deve ser capaz de classificá-las, permitindo que as dependências de nome sejam resolvidas, já que estão identificadas.

Entretanto, em algumas situações, a análise não é efetiva, principalmente uando existem arrays e ponteiros. Temos os casos:

- Objetos referenciados via ponteiros;
- Indexação indireta de arrays, através de outro array;
- Dependência que existe somente em relação a alguns valores de input.

Capítulo 5

Multiprocessadores Simétricos

5.1 Classificação de Arquiteturas

A divisão das arquiteturas de Flynn possui quatro categorias, segundo o numero de instruções executadas e dados acessados em um instante. Esta divisão foi o primeiro passo na classificação de arquiteturas paralelas e distribuídas. É considerada imprecisa. A Figura 5.3 dá a visão geral de todas.

Single Instruction, Single Data (SISD)

Processadores pipeline e monoprocessadores se encaixam aqui. Possuem os seguintes componentes:

- UC: unidade de controle
- UP: unidade de processamento, por exemplo a ULA
- MEM: memória

Single Instruction, Multiple Data (SIMD)

A extensão do SISD com multiplicação dos *hardwares* de unidade de processamento e memória (processadores vetoriais). Uma única instrução é buscada e é posta em todos os processadores para execução em posições de memória diferentes. Esta arquitetura é muito usada em GPUs.

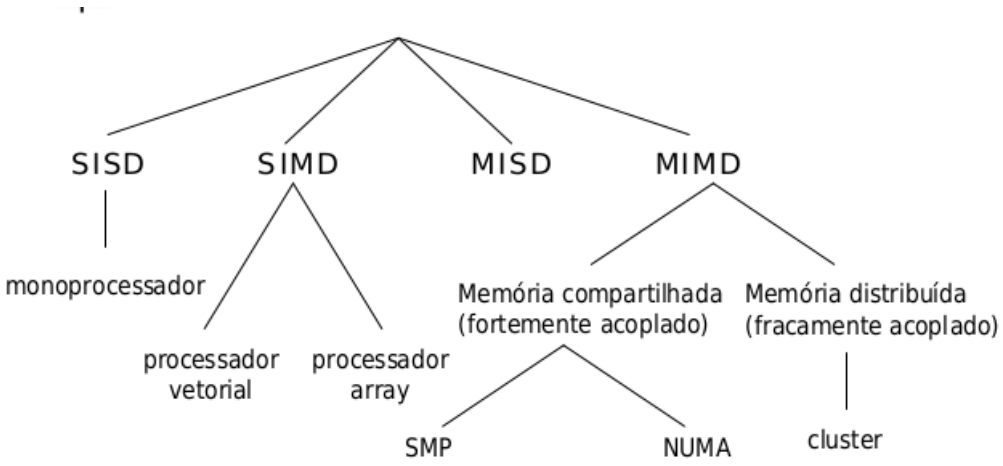


Figura 5.1: Taxonomia das Arquiteturas, por Stallings

Multiple Instruction, Single Data

Busca de várias instruções para execução sob um mesmo dado em memória. Dessa forma a replicação está nas unidades de controle e processamento. Esta arquitetura não possui exemplo de caso real, por não fazer sentido.

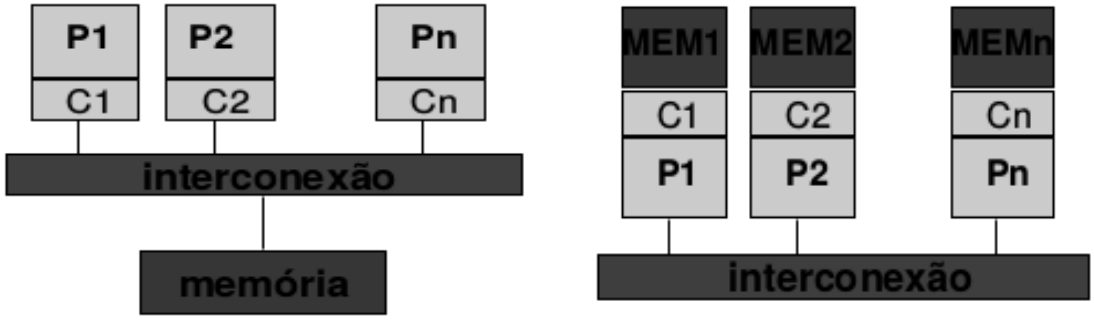
Multiple Instruction, Multiple Data (MIMD)

É uma arquitetura equivalente a multiprocessadores (ou multicores). Além disso, redes de computadores também se encaixam neste exemplo, dado que as unidades representadas são abstratas. Possuem duas arquiteturas:

- Fortemente acopladas: Todos os processadores tem acesso direto à memória, sendo ela **compartilhada**. Exemplo: multiprocessadores;
- Fracamente acopladas: cada processadore tem acesso direto apenas a sua memória. Para acesso à outras memórias, **deve** pedir para o processador detentor da memória alvo. Exemplos: multicomputadores.

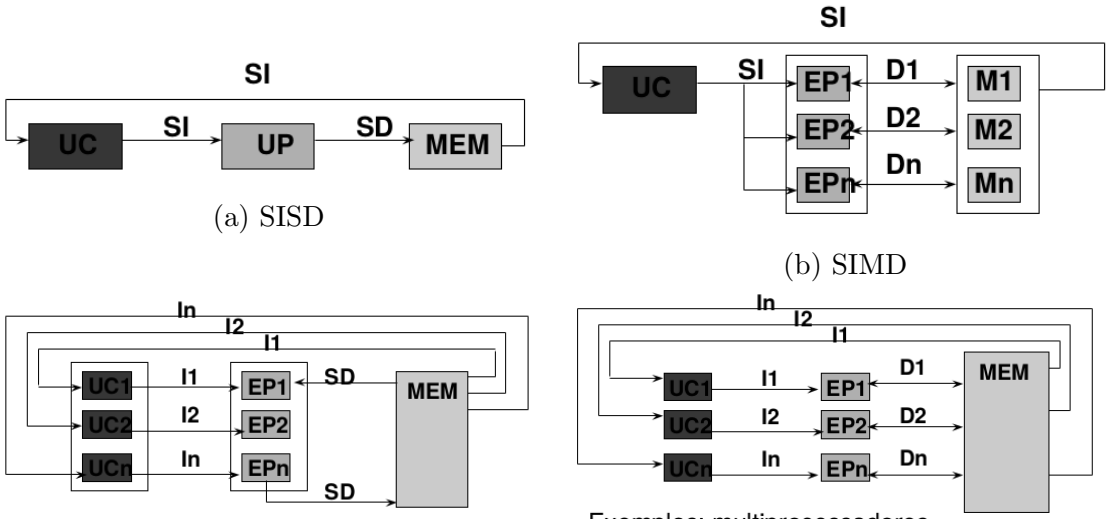
5.2 Multiprocessadores Simétricos

Inicialmente conhecidos como Uniform Memory Access Time (UMA). Possuem dois ou mais processadores com capacidade equivalentes, compartilhando memórias



(a) Fortemente Acoplada (b) Fracamente Acoplada

Figura 5.2: Tipos de Arquiteturas MIMD



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figura 5.3: Tipos de arquiteturas de processadores, segundo Flynn

e sistema de I/O, interconectados por um barramento ou crossbar, sendo essa arquitetura **transparente ao usuário**.

Todos os processadores podem desempenhar as mesmas funções e o sistema é controlado por um sistema operacional integrado. É uma implementação relativamente simples, uma vez que apenas aumentamos o número de processados, mas mantendo mantendo a memória e barramento únicos.

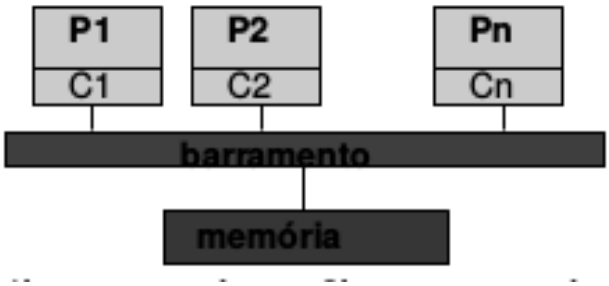


Figura 5.4: Estrutura de um multiprocessador simétrico. A memória aqui não é necessariamente única

Definição **Crossbar:** vários barramentos de maneira que você consiga uma conexão direta com todos os barramentos, evitando conflito de acesso. Ele é quadrátrico em relação ao número de elementos (n^2 conexões quando se tem n elementos). Isso permite a garantia de acesso em tempo uniforme. É um hardware mais caro.

Quando interconectados por barramentos, a arquitetura não é escalável, devido a problemas de contenção no acesso à memória. A cada processador adicionado, existe mais conflito no barramento. Com o crossbar, o problema é o custo do mesmo.

Assim, a introdução cache entre o processador e barramento aliviou um pouco o problema do acesso constante ao barramento, pois o acesso ficou local, gerenciado por algoritmos eficientes de coerência das caches.

Entretanto, mesmo com a introdução da cache, o número de processadores que podem ser conectados ao barramento não chega a 30, sendo geralmente 8. Para tentar resolver o problema de contenção em UMAs, foram propostas arquiteturas NUMA.

5.3 Coerência de Cache

As arquiteturas paralelas fortemente acopladas são afetadas por 3 grandes problemas.

- **Contenção da memória:** ocorre pois cada módulo de memória só pode tratar uma requisição por vez. Logo as requisições para um mesmo módulo são enfileiradas.
- **Contenção de comunicação:** o tempo de latência para redes de interconexão é dado pelo tempo que uma requisição leva para atravessar a rede de interconexão. Isso é diferente de largura de banda, que é a quantidade de informação que pode ser transmitida por uma rede em um determinada tempo.
- Tempo de latência

As caches em multiprocessadores podem ser:

- **Privadas:** cada processador tem sua cache;
- **Compartilhada:** só existe uma cache, compartilhada entre vários processadores

Definição **Coerência de cache:** garantia que todas as caches possuem sempre o mesmo valor para todas as cópias dos dados, **quando observadas**.

Logo, queremos garantir que todas as cópias em cache retornem sempre o último valor escrito, ou seja, uma visão coerente e uniforme da memória para todos os processadores, independente do fato de existirem várias caches.

Um **protocolo de coerência de cache** consistem em:

- Um conjunto de estados possíveis das caches locais;
- Um conjunto de estados possíveis da memória
- Transições de estados causadas por mensagem ou eventos locais.

Definimos então duas políticas parada garantir a coerência.

Política 1: **Write-invalidate**
Todas as cópias de um mesmo bloco são invalidadas antes que uma escrita/atualização seja feita em um processador.

Em caso de leitura, se a cópia existir localmente, a operação é feita também localmente. Caso não exista:

- Se existir uma cópia em leitura, fazemos uma outra cópia do bloco
- Se existir uma cópia em escrita:
  - Fazemos uma cópia do bloco e invalidamos a cópia em escrita; ou
  - Fazemos uma cópia do bloco e tornamos a cópia em escrita um *read-only*

Premissa: quando é observada a escrita de um bloco compartilhado, **invalide** seu bloco.

Exemplo: processador P_1 quer escrever na sua cache C_1 no bloco b. Ele avisa aos demais processadores sobre a escrita. Agora, cada processador P_i analisa se ele possui uma cópia de b e caso positivo, ele invalida a cópia. Ao fim, P_1 escreve o dado em sua cache. Se P_i desejar obter o bloco invalidado, terá que buscar da memória.

Política 2: **Write-update**
As escritas são feitas de maneira atômica em todas cópias dos blocos. Ou seja, assim que uma escrita de um bloco em cache é percebida, a cache solicita o bloco da memória.

Premissa: quando é observada a escrita de um bloco compartilhado, **atualize** seu bloco.

Exemplo: processador P_1 quer escrever na sua cache C_1 no bloco b. Ele avisa aos demais processadores sobre a escrita. Agora, cada processador P_i analisa se ele possui uma cópia de b e caso positivo, ele busca a cópia atualizada e a atualiza. Depois, P_1 escreve o dado em sua cache e libera o barramento. No fim, todas as caches estão com os dados atualizados.

5.4 Snoopy Caches

Em arquiteturas conectadas por um barramentos, broadcasts são feitos a cada operação de coerência e assim, as caches sempre estão "escutando"a rede para receber os comandos de coerência. Ao receber um comando, checam se possuem o bloco referênciado e, caso afirmativo, aplicam a operação de coerência. **Uma cache que implementa esse comportamento é uma snoopy cache**.

Diferentemente das caches de monoprocessadores, as snoopy caches possuem as seguintes características:

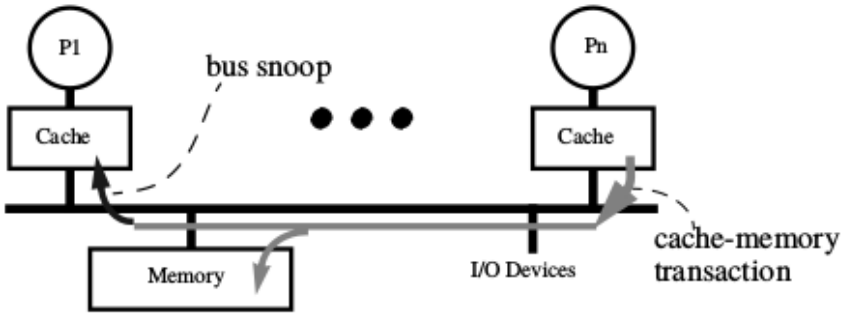


Figura 5.5: Funcionamento de uma snoopy cache

- O controlador da cache é uma máquina de estados finitos que implementa o protocolo de coerência de acordo com o diagrama de transição de estados;
- O difretório da cache precisa acrescentar um conjunto de bits para representar os estados;
- O controlador do barramento precisa monitorar toda a operação para descobrir se uma ação é ou não necessária.

5.4.1 Principais Protocolos de Coerência

Existem dezenas de protocolos para snoopy caches, dado que eles foram os primeiros protocolos a serem propostos, mas veremos os quatro principais.

Write Once

O primeiro protocol *write-invalidate* proposto. É um protocol híbrido, que usa a política *write-through* para a primeira escrita e a política *write-back* para as escritas subsequentes.

Transições de Leitura:

- **Read-Hit:** são feitos sempre na cópia local;
- **Read-Miss:** se não há cópia Dirty, a memória está coerente e provê uma cópia para a cache. Se existir uma cópia Dirty, a cache que possui a cópia Dirty envia uma cópia para a cache solicitante, a memória é atualizada e ambas as cópias mudam seu estado para Valid.

Transições de Escrita:

- **Write-hit:** se a cópia estiver DIRTY ou RESERVED, o write pode ser feito e o estado da cópia muda para dirty. Se o estado for VALID, é gerado um comando de invalidação `write_inv` para todas as caches, a memória é atualizada e a cópia é colocada como RESERVED;
- **Write-miss:** se não há cópia DIRTY, é enviado um comando `read_inv` para todas as caches, a cópia é atualizada localmente e o seu estado é DIRTY. Se há cópia DIRTY, a cópia é fornecida pela cache correspondente, que invalida a sua cópia após o fornecimento. A cópia é alterada localmente e seu estado é DIRTY.

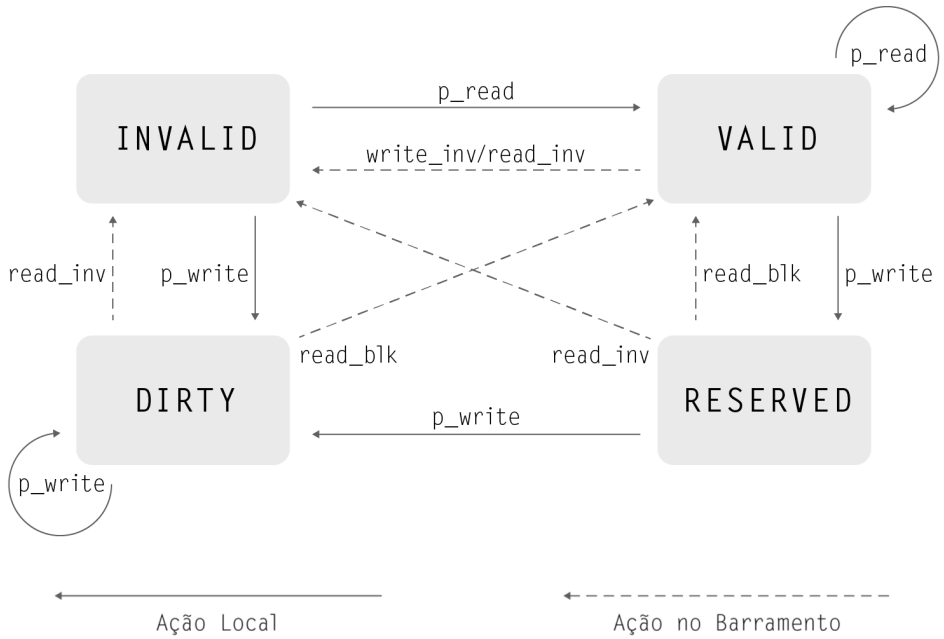


Figura 5.6: Diagrama de estados do *Write Once*

Firefly

É um tipo de protocolo *write-update* que foi implementado em estações de trabalho multiprocessada Firefly. A política utilizada entre as caches e a memória é a de *write-through*. Necessita de uma linha especial de barramento para fazer as atualizações, chamado de *shared line*.

TRANSIÇÕES DE LEITURA:

- **Read-Hit:** são feitos sempre na cópia local;
- **Read-Miss:** se existem cópias SHARED, as caches fornecem o bloco. Se a cópia está DIRTY, a cache fornece o bloco e atualiza a memória. O estado de ambas as cópias muda para SHARED. Se não há cópias, a memória fornece uma cópia, que é colocada no estado VALID-EXCLUSIVE.

TRANSIÇÕES DE ESCRITA:

- **Write-hit:** se o bloco estiver DIRTY ou VALID-EXCLUSIVE, a escrita é local e o novo estado é DIRTY. Se a cópia estiver em SHARED, todas as cópias e a memória são atualizadas. Se não existe mais compartilhamento, o estado muda para VALID-EXCLUSIVE;
- **Write-miss:** se a cópia for fornecida pela memória, ela é atualizada localmente e o seu estado fica DIRTY. Senão, todas as outras cópias são atualizadas e o novo estado é SHARED.

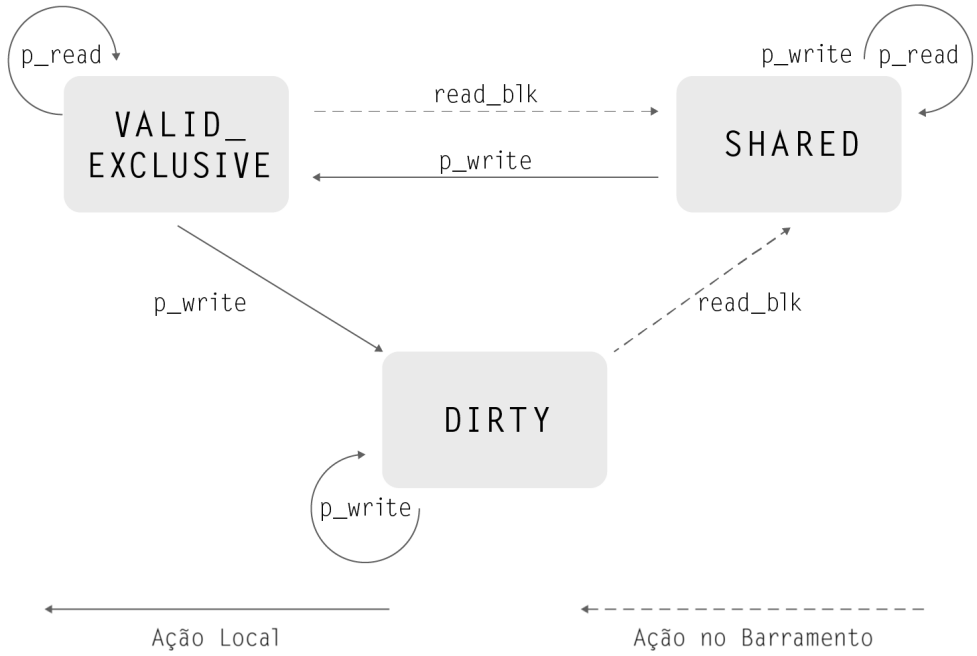


Figura 5.7: Diagrama de estados do Firefly

MSI

ESTADOS

- **Modified** ou Dirty: somente um processador possui uma cópia válida do bloco em sua cache. A cópia da memória principal está incoerente.
- **Shared**: o bloco está presente em diversas caches e está coerente com a memória.
- **Invalid**

Este protocol segue a política de cache **write-back** e é do tipo **write-invalidate**.
Note que:

- Ao longo do protocol apenas um processador pode estar no estado MODIFIED;
- O estado MODIFIED só pode coexistir com estados INVALID.

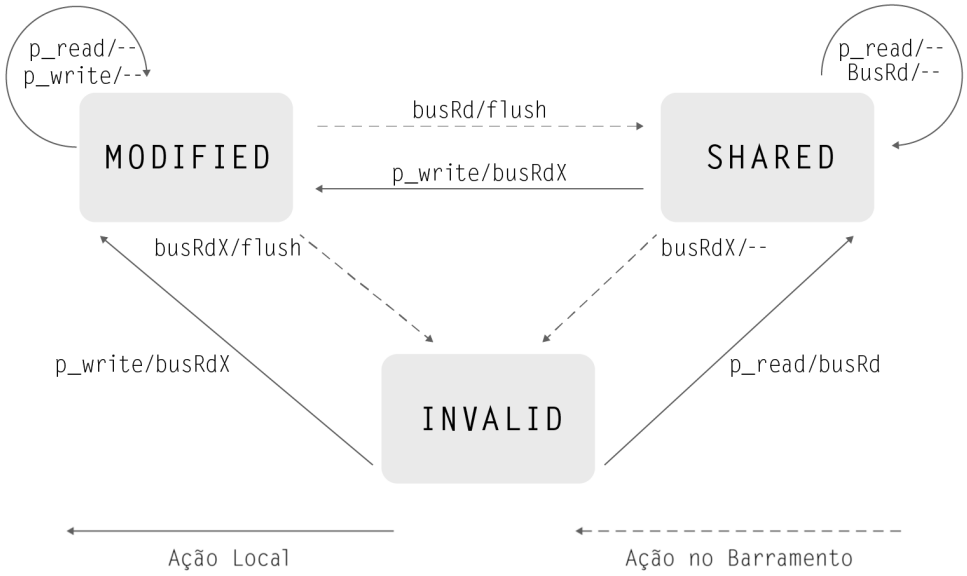


Figura 5.8: Diagrama de estados do MSI

MESI

Foram propostos para explicitar o estado onde um dado foi lido com a intenção de ser escrito. A inclusão deste terceiro estado diminui o número de transações no barramento para este caso.

Este protocol segue a política de cache **write-back** e é do tipo **write-invalidate**.

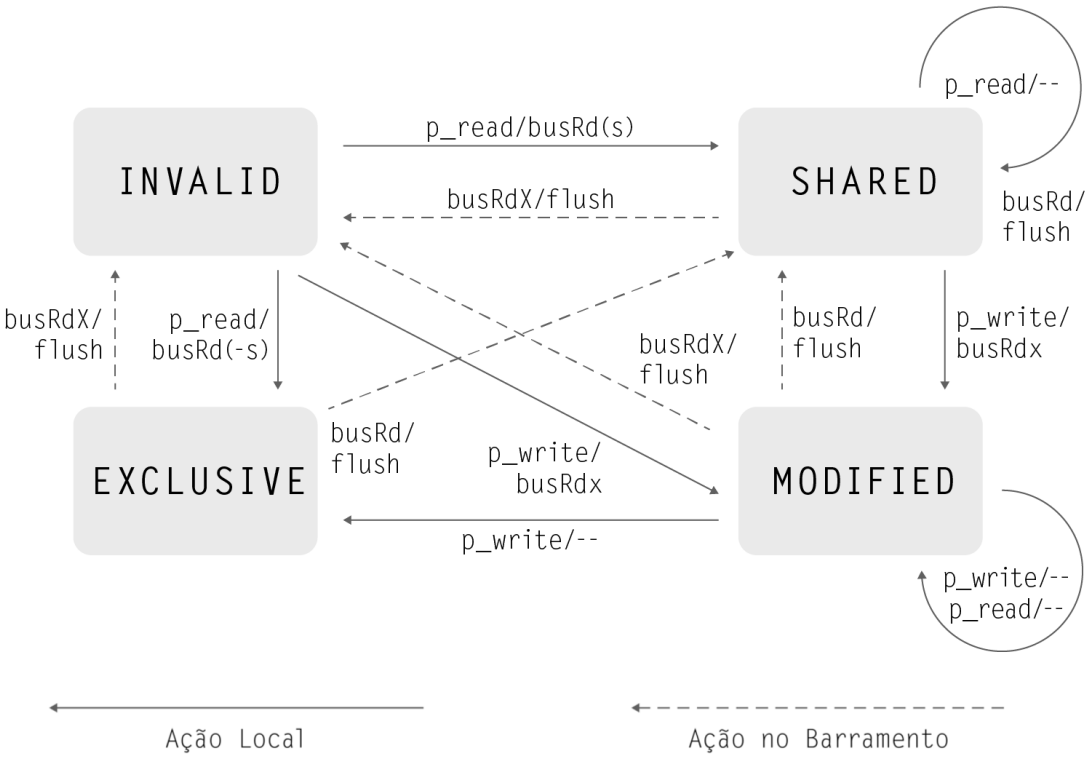


Figura 5.9: Diagrama de estados do MESI

Observações
Protocolos *write-invalidate* apresentam problemas quando existe um escritor e vários leitores. Para esta situação, podemos reduzir o número de *read misses* se, no primeiro read miss que ocorrer, uma cópia for enviada a todas as caches que possuem o bloco no estado Invalid.

Protocolos *write-update* tem o problema de que algumas atualizações podem ser aplicadas em cópias que não mais serão lidas. Por isso, alguns protocols adicionam estados que representam o número de vezes que um bloco foi alterado sem que nenhuma leitura fosse gerada. Se este número atingir um certo limite, as cópias são invalidadas.

5.4.2 Protocolos Baseados em Diretórios

Em algumas arquiteturas, não dispomos de barramento, logo um *broadcast* tem um custo muito alto. Ao invés de usarmos o snoopy cache, fazemos um multicast

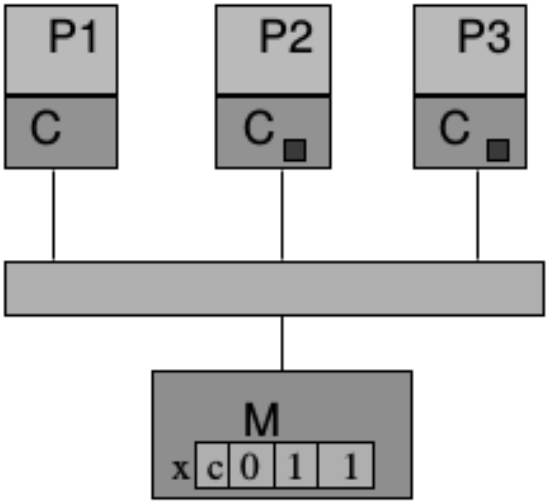


Figura 5.10: Representação de Diretórios Full-map do estado das caches

somente para as caches que com certeza terão o bloco.

Aqui, a memória possui uma lista dos blocos em cache: para cada bloco nas caches, temos a localização de todos os nodos que possuem uma cópia. Esta lista pode ser centralizada ou distribuída e é chamada de **diretório**.

Em cada entrada do diretório, ou seja, para cada bloco, temos todas as caches que possuem o bloco e um *dirty bit*, indicando a permissão de escrita.

Full-map Directories

Aqui, todas as caches do sistema podem abrigar simultaneamente qualquer bloco de dados. Cada entrada de diretório possui, no mínimo, N bits, onde N representa o número de processadores no sistema. Este bit representa o estado do bloco na cache do processador: se está presente ou ausente. Como exemplo, podemos checar a Figura 5.10

Escrita: quando um processador quer escrever, sua cache verifica se o seu bloco local tem permissão de escrita. Caso não, a cache faz *write* para a memória. As demais caches que possuem o bloco recebem o sinal do módulo da memória para invalidar o dado e enviam um `ACK` para a memória, afim de confirmar. Por fim, o módulo atualiza o diretório e envia a permissão escrita a C3.

Perceba que há um *overhead*: o tamanho de cada entrada de diretório é proporcional ao número de processadores do sistema e isso **não é escalável**. Por isso,

vários esquemas alternativos surgiram.

Limited Directories

Propostos com o intuito de **reduzir o tamanho dos diretórios**. Aqui o número de cópias em cache simultâneas para um bloco particular é limitado.

DEFINIÇÃO **Dir**$_i X$, onde i é o número de cópias simultâneas e X pode ser NB (*no broadcast*) ou B (*broadcast*).

São similares aos full-map, exceto no caso onde existem mais de i solicitações de leitura simultâneas. Neste caso, podemos tomar duas atitudes:

- Dir$_i$NB: uma das cópias é invalidada para dar lugar à nova cópia;
- Dir$_i$NB: caso haja j cópias em leitura e $j > i$, é enviado um *broadcast* na rede para cada operação de coerência.

Escrita: quando um processador P_i quer escrever, sua cache C_i verifica se o bloco local em questão está inválido. Caso positivo, solicita uma cópia da memória, a qual pode detectar que já existem cópias em outras caches distintas. A memória escolhe cópia na cache C_k uma para invalidar (se torna *evicted*) e envia o sinal. A cache C_k recebe a invalidação, seta o bloco como inválido e envia uma confirmação `ACK` para a memória. Por fim, a memória atualiza o diretório, colocando a C_i no lugar de C_k na posição do bloco solicitado e envia a permissão de escrita para C_i.

Diretórios Encadeados

É um tipo de diretório **escalável, porém não limita o número de cópias**. A localização das cópias é obtida através de uma cadeia de ponteiros, como mostrado na Figura 5.11.

Esquemas Baseados em Software

Para decidir se o dado é "seguro", temos que marcar todos os dados compartilhados que foram lidos e escritos em algum momento. Marcamos eles como *non-cachable*.

Por fim, precisamos decidir em que fases de um programa uma variável *read-write* pode ser classificada como segura. Ao final da fase, é inserida uma primitiva de invalidação.

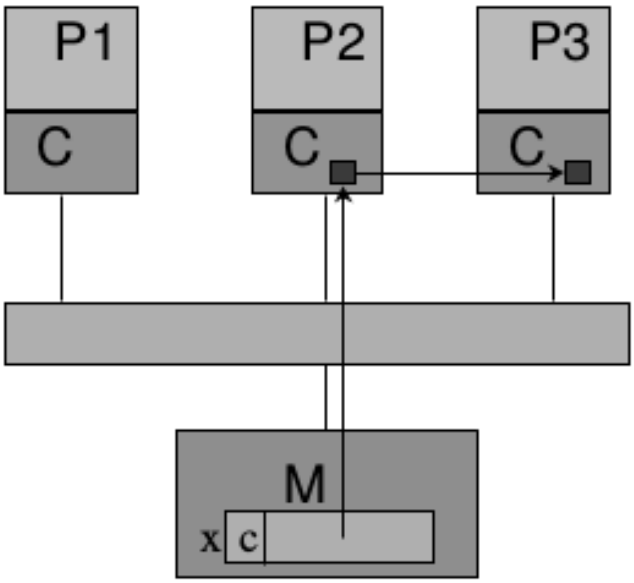


Figura 5.11: Representação de Diretórios Encadeados

Referências Bibliográficas