

Tabela de conversão

Decimal Prefix		Binary Prefix	
Name (Symbol)	Value	Name (Symbol)	Value
Kilobyte (kB)	10^3 Byte	Kibibyte (KiB)	2^{10} Byte
Megabyte (MB)	10^6 Byte	Mebibyte (MiB)	2^{20} Byte
Gigabyte (GB)	10^9 Byte	Gibibyte (GiB)	2^{30} Byte
Terabyte (TB)	10^{12} Byte	Tebibyte (TiB)	2^{40} Byte
Petabyte (PB)	10^{15} Byte	Pebibyte (PiB)	2^{50} Byte
Exabyte (EB)	10^{18} Byte	Exbibyte (EiB)	2^{60} Byte
Zettabyte (ZB)	10^{21} Byte	Zebibyte (ZiB)	2^{70} Byte
Yottabyte (YB)	10^{24} Byte	Yobibyte (YiB)	2^{80} Byte

Arquitetura de processadores

Pontos Importantes:

- Nenhuma instrução opera com dados na memória, apenas nos registradores.
- Todas as instruções fazem uso da ULA.
- A instrução de `beq` utiliza a ULA para calcular o endereço de destino do desvio (PC + label) e também para calcular se o número é igual.
- O caminho de dados executa uma instrução em um único ciclo de clock.
 - Por isso, tem uma memória de dados e outra de instruções, para cada uma fazer a operação simultaneamente.
- **Hazard** é um problema encontrado em processadores de execução paralela. É quando um recurso necessário para a execução da próxima instrução ainda está sendo utilizado/calculado por outra execução. Existem 3 tipos: estruturas, de dados e de controle.
- No caso de um desvio, a label é armazenada em quantidade de palavras, mas o PC recebe em bytes. Por isso, para realizar o desvio o PC deve pegar o valor em bytes da label. Para transformar de palavras para bytes multiplica-se por 4.

Processador Monociclo

O que cada Control Signals faz

Control Signals	Localidade	Caso 1
RegDst	MUX	Se a instrução possuir 'rd', define o Write Register como 15-11, senão como 20-16
Branch		Realiza o desvio
MemWrite	Memória de Dados	Escreve na memória
MemRead	Memória de	Lê da memória

	Dados	
MemToReg	MUX	O dado a ser salvo no registrador é vem da memória
ALUSrc	MUX	Define que a entrada do segundo operando da ULA vem do controlador, senão pega do registrador
RegWrite	Banco de Registradores	Define que haverá uma escrita no Write Register

Classe de Instruções

- Acesso à memória: lw, sw
- Lógicas e aritméticas: add, sub, and, or, slt
- Devio condicional: beq, j

Ciclo Fetch -> Decode -> Execute

- Fetch: Obtém o endereço de memória armazenado no PC (Program Counter que Contém o endereço de memória do programa que está executando)
- Decode: Decodifica a instrução e obtém os dados necessários para a execução. Acesso aos registradores.
- Execute (Depende da classe de instrução):
 - Usa uma ULA para cálculos
 - Instruções aritméticas: resultado da operação
 - Instruções de acesso à memória: cálculo do endereço
 - Instruções de desvio: endereço do desvio
 - Faz acesso à memória
 - PC <- destino do desvio (caso seja fornecido) ou PC+4 (para ir para a próxima instrução pula-se 4 bytes ou 32 bits)

Visão geral da CPU

- Tem dois somadores, o que recebe o PC soma com 4 para carregar a próxima instrução e o outro para fazer o desvio.
- A CPU tem apenas uma memória, mas ela é representada em duas partes, Instruction Memory e Data Memory, sendo elas utilizadas para armazenar os dados dos registradores e para ler ou escrever dados, respectivamente.
- Os caminhos de dados não podem se juntar diretamente, por isso nessas junções utilizam multiplexadores (MUX), que são dispositivos que definem o resultado a partir de um sinal de controle e duas entradas.
- Como o ciclo da instrução acontece no caminho de dados ocorre:
 - O PC junto com o instruction memory realiza o `fetch`
 - A controladora realiza o `decode` da instrução e determina todos os sinais de controle para os multiplexadores
 - Execução

Logic Design Basics

- Informação sempre codificada em binário
 - Voltagem baixa = 0
 - Voltagem alta = 1
- Elementos Combinacionais
 - Operam com os dados
 - A saída é uma função da entrada
- São aqueles que operam sobre duas entradas e a partir delas define uma saída. Realiza o processamento dos dados.
- AND $\rightarrow Y = A \& B$
- ULA $\rightarrow Y = F(A,B)$, sendo F um sinal de controle que distingue qual operação a ULA deve executar
- MUX $\rightarrow Y = S ? 11 : 10$
- Somador $\rightarrow Y = A + B$
- Elementos de Estado (Sequenciais) eles não operam sobre a informação, apenas armazenam.
 - Registrador: armazena dados num circuito
 - Usa o sinal de clock para determinar quando atualizar o valor armazenado.
 - Edge-triggered: ele apenas atualiza quando o ciclo de clock termina (quando muda de 0 pra 1) e envia de volta na saída o dado atualizado.
 - Registrador com controle de escrita:
 - Apenas atualiza ao final do clock quando o controle de escrita for 1.

Metodologia de Clock

- A lógica combinacional transforma os dados durante um ciclo de clock.
 - A duração do clock é definida pelo tempo que ele demora para executar a instrução mais demorada.
-

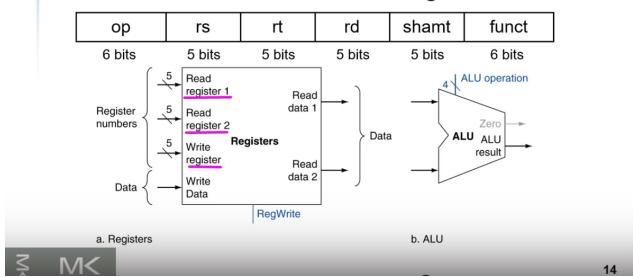
Contruindo um Caminho de Dados

- Elementos que processam os dados e endereços numa CPU:
 - Registradores, ULAs, MUXs, memórias, etc.
- Fetch: Obter a instrução. Enquanto o banco de memória está trabalhando, o PC é incrementado em 4 para obter o endereço da próxima instrução e a saída do fetch é definida a partir dos 6 bits do opcode.

Instruções do tipo R

Instruções do tipo R

- Lê dois registradores operando
- Faz a operação lógica/aritmética
- Escreve o resultado num registrador

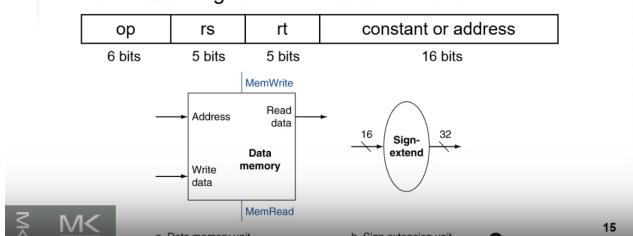


- O banco de registradores é uma unidade autônoma que, dado o número do registrador, recupera os dados naquele registrador
- A entrada do banco de registradores recebe a saída do fetch. Essa entrada consiste no número dos registradores de 1 e 2 e a saída devolve o valor nos registradores.
- Do banco de registradores, a informação segue para a ULA

Instruções do tipo I

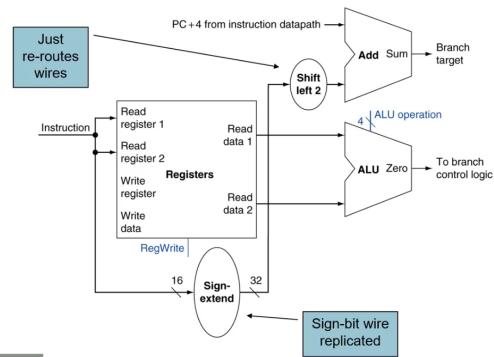
Instruções load/store

- Lê os registradores operando
- Calcula o endereço de memória usando o offset de 16 bits
 - Usa a ULA e também um extensor de sinal do offset
- Load: lê da memória e escreve no registrador
- Store: lê do registrador e escreve na memória



- Lê os registradores operando.
- Calcula o endereço de memória usando o offset de 16 bits.
 - Usa a ULA e também um extensor de sinal do offset.
- Load: Lê a memória e escreve no registrador.
- Store: Lê do registrador e escreve na memória.
- MemWrite e MemRead não podem ser iguais.

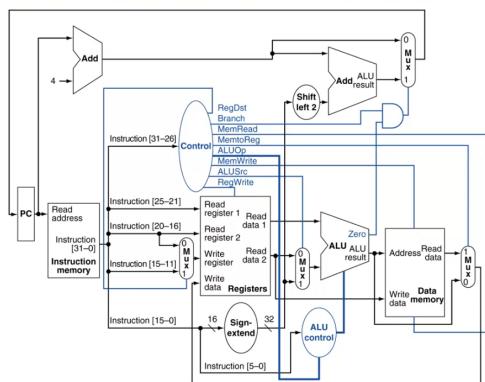
Instruções de desvio



- Utiliza duas ULAs, uma faz a soma para a soma para identificar o endereço de memória da próxima label. Já a outra realiza a comparação para ver se o resultado da subtração é igual a zero, se for igual a zero ele passa (BEQ).

Caminho de dados com controle

Caminho de dados com controle



Controladora da ULA

- A ULA é utilizada para:
 - Load/Store: $F = \text{add}$
 - Desvio (beq): $F = \text{subtract}$
 - Instruções tipo R: F depende do campo funct

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

- ALUOp é gerado pela controladora principal gera a partir do opcode.

Controle da ULA

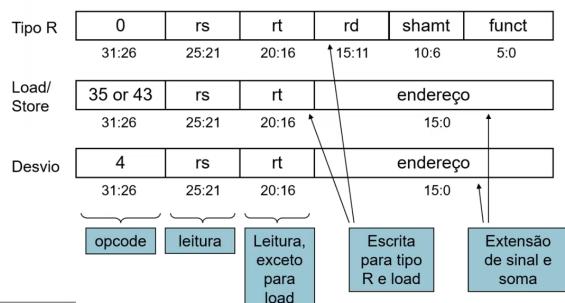
- ALUOp = opcode
 - Uma combinação deste com funct gera o código de controle da ULA

opcode	ALUOp	Operação	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
	11	AND	100100	AND	0000
		OR	100101	OR	0001
	10	set-on-less-than	101010	set-on-less-than	0111

- Está marcado como XXXXXX pois as instruções não possuem campo funct.

A unidade de controle principal

- Os sinal de controle derivam da instrução



Exemplos de Caminho de dados

R-type Instructions

Control Signals	Valor
RegDst	1
Branch	0
MemWrite	0
MemRead	0
MemToReg	0
ALUSrc	0
RegWrite	1

Load Word

Control Signals	Valor
RegDst	0
Branch	0
MemWrite	0
MemRead	1

MemToReg	1
ALUSrc	0
RegWrite	1

Store Word

Control Signals	Valor
RegDst	0
Branch	0
MemWrite	1
MemRead	0
MemToReg	0
ALUSrc	0
RegWrite	0

BEQ

Control Signals	Valor
RegDst	0
Branch	1
MemWrite	0
MemRead	0
MemToReg	0
ALUSrc	0
RegWrite	0

Jump

- A instrução `j` (jump) não utiliza os control signals, ao invés disso ela atualiza o valor de PC.
- Ela passa o endereço de desvio nos bits 25-0 da instrução, desloca-se esse valor 2 casas a esquerda e soma ele com o PC.

Control Signals	Valor
RegDst	0
Branch	0
MemWrite	0
MemRead	0
MemToReg	0
ALUSrc	0
RegWrite	0

Problemas do desempenho do processador monociclo

- A instrução mais demorada determina o período de clock.
 - Caminho crítico: instrução `load` <- A mais demorada.

- o Memória de instrução -> banco de registradores -> ULA -> memória de dados -> banco de registradores

MIPS Pipeline

Cinco estágio:

1. IF - Instruction Fetch: Recuperação da instrução da memória.
2. ID - Instruction Decode: Decodificação da instrução & leitura dos registradores.
3. EX - Execution: Execução da operação ou cálculo do endereço.
4. MEM: Acesso à memória de dados.
5. WB - Write-Back: Escrita do resultado no registrador.

Desempenho do pipeline

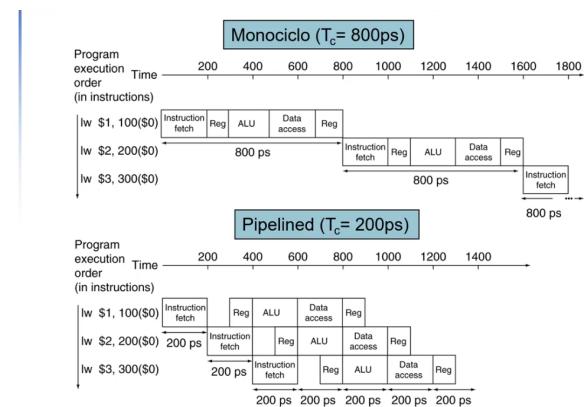
Suponha que o tempo dos estágios seja:

- 100ps para leitura ou escrita de registradores;
- 200ps para outros estágios.

Comparando um caminho de dados pipeline com monociclo:

Instr	IF	ID	EX	MEM	WB	Total
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
R-format	200ps	100ps	200ps		100ps	600ps
beq	200ps	100ps	200ps			500ps

Devido ao clock de um processador monociclo ser definido pelo tempo de execução da instrução mais demorada.



Melhoria do Monociclo para o Pipeline

- Vazão: Maior quantidade de resultados no mesmo tempo. O que não quer dizer que os dados são produzidos mais rápidos, pois a latência (tempo para cada instrução) não melhora, podendo ser até pior.

Hazard

São situações que surgem devido ao paralelismo que impedem de começar a próxima instrução no próximo ciclo. Existem 3 hazards:

- Hazard Estrutural: um recurso necessário está indisponível.
- Hazard de Dados: precisa aguardar uma instrução anterior terminar para usar sua saída.
- Hazard de Controle: decidir uma ação de controle depende de uma instrução anterior (beq, por exemplo).

Hazards estruturais

É quando há algum conflito ao utilizar um recurso.

Boa pergunta de prova

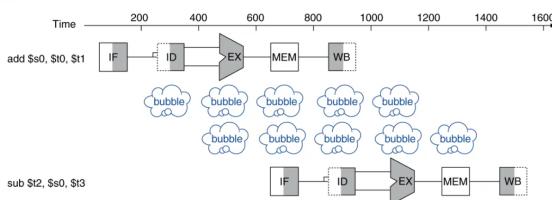
As instruções load e store acessam a memória de dados, supondo que um processador MIPS tenha apenas uma unidade de memória, não seria possível recuperar uma instrução em determinado ciclo, causando um stall (bubble). Por isso, o caminho de dados possui diferentes unidades de memória para instruções de dados.

Hazards de Dados

Acontece quando uma instrução depende do resultado de outra instrução anterior

Hazards de Dados

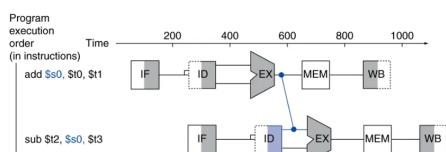
- Uma instrução depende do resultado de outra instrução anterior
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



Forwarding

Forwarding (Bypassing)

- Já usa um resultado assim que for calculado
 - Não espera ser armazenado num registrador
 - Requer conexões extras no caminho de dados.

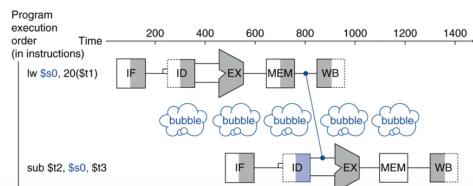


Nem sempre é possível evitar stalls usando forwarding. Quando isso ocorre, utiliza-se o **Load-Use**.

Load-Use

Hazard de dados Load-Use

- Nem sempre é possível evitar *stalls* usando *forwarding*
 - Se o valor necessário ainda não foi calculado



Além desses dois, o que pode ser feito para evitar stalls é reordenar instruções. Até mesmo o compilador pode reordenar as instruções automaticamente.

Hazards de Controle

O desvio determina o fluxo de controle, recuperar a próxima instrução depende do resultado desse desvio. Não é sempre possível recuperar a instrução correta com pipeline, pois ele ainda estará na fase ID do desvio.

No pipeline MIP, a comparação da branch deveria ser feita antes do pipeline. Para isso, deveria adicionar um hardware para fazer isso na etapa ID.

Predição de Desvio

Pipelines mais longos não podem determinar prontamente o resultado do desvio antecipadamente
-> As penalidades de stall tornam-se inaceitáveis

Prediz a saída do desvio -> O Stall acontece apenas quando erra

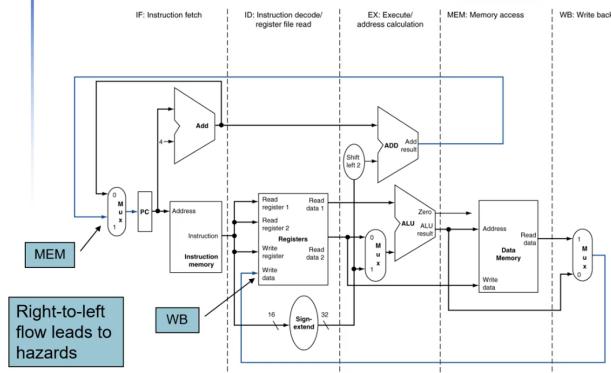
No pipeline MIPS, é possível predizer que o desvio não é tomado, recupera a instrução depois do desvio.

Tipos de predições de desvio:

- Predição de desvio estática
 - Baseada no comportamento típico. Prediz que o desvio "para trás" é tomado. Prediz que o fluxo normal não é tomado.
 - Exemplo: laços e condicionais.
- Predição de desvio dinâmica
 - O hardware mede o comportamento atual dos desvios e grava o histórico recente de cada desvio.
 - Assume que o comportamento futuro seguirá a tendência, quando errar faz um stall enquanto recupera a nova instrução e atualiza o histórico.

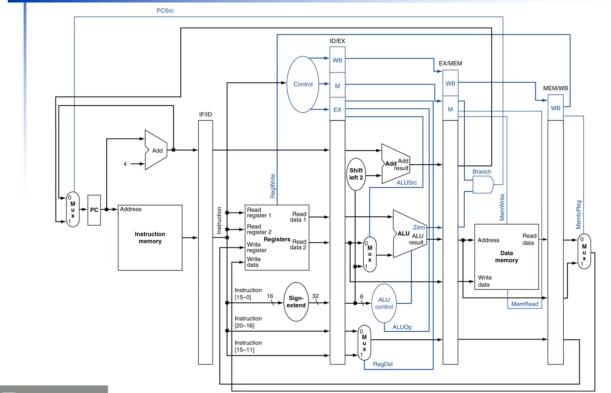
Caminho de dados em um MIPS com pipeline

MIPS Pipelined Datapath



Controle no Pipeline

Controle no Pipeline



Hierarquia de memória

Principais Conceitos

- Localidade Temporal:
 - Dados acessados recentemente tendem a ser acessados novamente em breve.
Exemplo: Instruções num laço.
- Localidade Espacial:
 - Dados próximos aos acessados recentemente tendem a ser acessados em breve.
Exemplo: Vetores.
- Memória Principal:
 - Processador endereça diretamente.
 - Memória Cache e Memória RAM.
- Memória Secundária:
 - Memórias não voláteis.

Transferência de dados entre os níveis

A transferência é realizada em blocos. Um bloco é uma unidade de cópia e pode ser múltiplas palavras.

Acerto

Se o dado está presente no nível superior e o dado consegue ser acessado é instituído um **acerto**. A razão entre acertos e total de acessos é entendida como a **taxa de acerto**.

Falha

Se o dado não estiver presente no nível superior, institui-se uma **falha**. Quando ocorre a falha o dado é copiado do nível inferior para o superior. A **taxa de falha** é a razão entre falhas e total de acessos.

A penalidade para uma falha é o tempo gasto para copiar o dado.

Como o processador acessa um dado

O processador acessa um dado por meio de blocos através de endereços armazenados na memória principal.

Mapeamento

Surge da necessidade de saber se o dado está presente no cache e onde ele se encontra na memória cache.

Para isso, mapeia-se os dados entre a memória RAM e a cache.

Mapeamento Direto

Mapeamento que divide a memória RAM em pedaços do mesmo tamanho da memória cache.

Dado o endereço da memória principal, é possível achar o endereço na memória cache.

```
cache = bloco % qtdBlocosCache  
sendo,  
cache = Endereço na Memória Cache  
bloco = Endereço de Início do Bloco da Memória Principal  
qtdBlocosCache = Quantidade de Blocos que cabem na cache
```

Um endereço do bloco possui um correspondente na memória cache, mas um bloco na memória cache possui vários correspondentes na memória principal.

- Total de blocos na memória principal sempre vai ser uma potência de 2^n . Pois pode-se endereça-la em bits.
- Endereço na cache: n bits menos significativos do endereço do bloco.
- Sendo que n é baseado na quantidade de blocos da cache. Ou seja, se a cache tem 8 blocos, $n=3$, pois $8 = 2^3$.

Tags e Bit de Validade

Tag

- Como saber qual o endereço principal de um bloco na memória cache?

- Armazenar o endereço juntamente com o dado, como os bits menos significativos são o endereço do cache, utiliza-se apenas os bits mais significativos necessários e armazena na `tag`.

Bit de Validade

- Como saber se o dado presente num bloco da cache tem um correspondente na memória principal?
 - Declara-se um `bit de validade` que vale 1 se estiver presente e 0 caso contrário. Inicialmente vale 0.

Verificando os Dados

Quando acessar o dado da memória é verificado primeiro o `bit de validade`, caso ele não seja 1 significa que ele é um dado inválido, resultando em uma falha e na busca na memória principal e pelo o dado correto.

Quando o `bit de validade` for 1, o dado na memória é um dado consistente. Então é realizado a concatenação da `tag` com o endereço do cache e então é validado se ele coincidi com o endereço localizado na memória principal.

Memória Virtual

É uma simulação que realiza o uso da memória principal como "cache" da memória secundária. É gerenciada pela CPU e pelo sistema operacional.

Objetivos

- Swapping: utilizar mais memória que existe fisicamente por meio de uma área de troca, ou seja, por meio de um `swap`.
- Proteção: compartilhamento eficiente e seguro de memória. Cada programa possui um endereço virtual privado contendo código e dados, protegendo-os de outros programas acessarem.
- Relocação: endereço virtual \Leftrightarrow endereço físico. A CPU e o SO realizam uma tradução do endereço virtual para o físico. Cada bloco da memória virtual seria uma página, caso o dado necessário não esteja na página é constituído uma `falha de página`, sendo necessário trazer uma outra página ou outros dados à página.

Tradução de Endereços

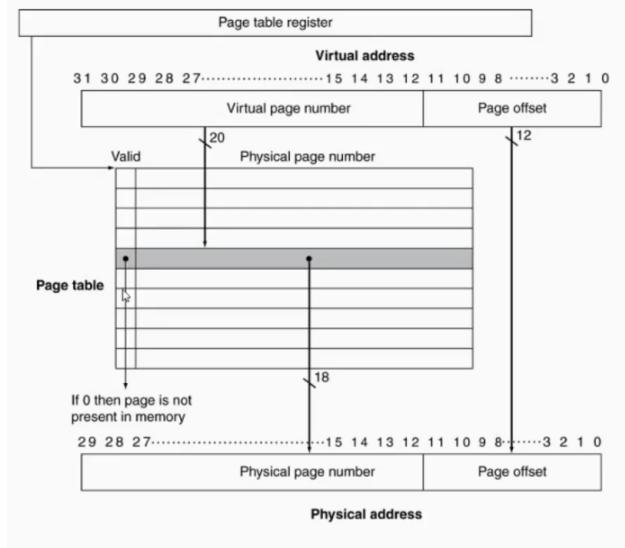
É uma tradução direta, então não é necessário utilizar a `tag`.

Em caso de falha de página, ela deve ser recuperada do disco e substituir a memória virtual. Essa substituição é realizada pelo SO e consome milhões de ciclos de clock.

O objetivo é minimizar a taxa de falhas de página, para isso utiliza-se o `mapeamento associativo`, que, ao contrário do mapeamento direto, não necessariamente irá sempre ocupar o mesmo espaço da memória física. Além do mapeamento associativo, o SO utiliza algoritmos de substituição inteligentes.

Tabela de Páginas

É um vetor de páginas indexado pelo endereço virtual da página. De forma que atua como uma estrutura de dados para intermediar a tradução da memória virtual para a memória física.



Substituição e escrita

Least-Recently Used (LRU)

Substitui-se a página usada há mais tempo.

Seta o bit de referência como 1 quando o dado é acessado e é zerado periodicamente pelo SO.

Escrita

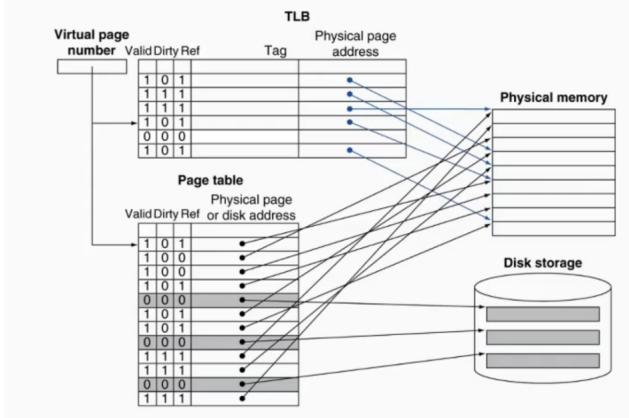
A escrita de dados é feita pelo processador na memória cache. Isso pode causar inconsistências com dados nos outros níveis (RAM, disco). Há duas estratégias para resolver esse problema:

- **Write-Through:** Os dados escritos no cache são imediatamente escritos na RAM. O problema disso é que o processador permanece ocioso enquanto a escrita não for concluída. Nesse caso usa-se um armazenamento intermediário chamado **buffer de escrita**. Com isso, o processador entende que a escrita terminou ao escrever no buffer e o sistema de memória de forma independente, sincroniza os dados do buffer com a RAM.
- **Write-Back:** os dados são escrritos apenas no cache e são levados para a RAM apenas quando é necessário buscar algum dado nela, ou seja, em casos de falha.

A escrita no disco consome milhões de ciclos, por isso o write-through é impraticável, usa-se o write-back. Para o write-back ser realizado é necessária a inclusão de um terceiro bit na tabela de páginas, o **dirty-bit** sinaliza o dado que está inconsistente com a memória física. Dessa forma, quando acessar a memória física é realizada a escrita e só depois é realizada a substituição da página.

TLB (Translation Look-aside Buffer)

Assim como a memória cache está para a memória principal, a TLB está para a Tabela de Páginas. A TLB é um cache para a memória de páginas. Ele é localizado no cache do processador, onde tem um espaço destinado às entradas da tabela de páginas.



O TLB é organizado da mesma forma que o cache.

Quando um endereço virtual é solicitado o processador procura primeiro no TLB. Se esse endereço não estiver no TLB, é feito um acesso à tabela de página para se obter o endereço e incluir-lo no TLB.

O mapeamento entre a tabela de páginas e o TLB é feito da mesma maneira que o mapeamento entre a memória principal e o cache

Falhas de acesso no TLB

Quando há uma falha no TLG, há dois possíveis motivos:

- O dado procurado está na memória principal.
 - Carrega a entrada da tabela de páginas no TLB. ou
- O dado procurado não está na memória principal.
 - Falha de página: o SO carrega a página e atualiza a tabela de páginas.

Interação entre TLB e cache

É feita a tradução do endereço virtual para o endereço real do TLB e então é enviado ao cache.

Não é possível enviar diretamente o endereço virtual na cache pois, por cada programa possuir sua própria memória virtual, podem existir dois endereços iguais com dados diferentes. Enquanto na real, esses dados estão ocupando endereços diferentes.