

*Estudios*

*Arg III*

# Materia:

- ↳ Cache
- ↳ Memória Virtual } Hierarquia de Memória
- ↳ Pipeline de Instruções

$$1 \text{ kb} / 1 \text{ kbps} = ?$$

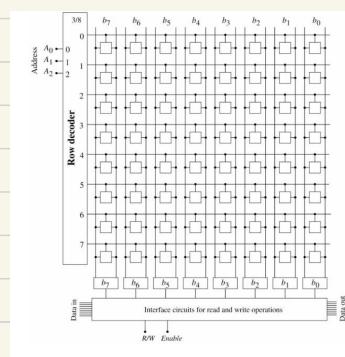
$$1 \times 1024 / 1 \times 1000 = 1,024 \text{s}$$

$$\begin{aligned} \text{Kilo} &= 2^{10} \\ \text{Mega} &= 2^{20} \\ \text{Giga} &= 2^{30} \\ \text{Tera} &= 2^{40} \end{aligned}$$

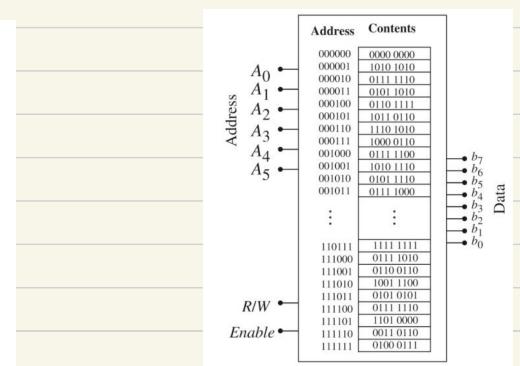
## Memória:

- > **SRAM (estática)**: Mantém os bits de dados armazenados somente enquanto a fonte de alimentação estiver conectada ao circuito.
- > **DRAM (dinâmica)**: São + simples e necessitam de menos área no chip, permitindo densidades de armazenamento maiores, mas mais lentas. São muito utilizados p/ memórias principais de PCs.

- > **Registrador**: É um elemento lógico utilizado p/ armazenar uma palavra de  $n$ -bits



Memória (arranjo de SRAM, matriz 8x8)



Memória (arranjo de SRAM, matriz 64x8)

★ 1 palavra de 8 bits  
= 1 byte

Tamanho da palavra	Número de valores	Abreviação para valor
8b	$2^8 = 256$	
10b	$2^{10} = 1024$	1kb
16b	$2^{16} = 65\,536$	64kb
20b	$2^{20} = 1\,048\,576$	1Mb
28b	$2^{28} = 268\,435\,456$	256Mb
30b	$2^{30} = 1\,073\,741\,820$	16b

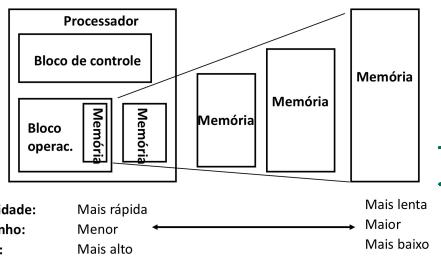
- Processador com uma palavra de 32b enxerga quantos endereços?
- $2^{32} = 2^2 \times 2^{30} = 4 \times G = 4\text{Giga}$  endereços.
- Se cada endereço armazena 1 byte, a memória é de 4GB.

# Hierarquia de Memória

→ o objetivo é oferecer uma ilusão de máximo tamanho de memória, com mínimo custo.

- maximizar velocidade

- cada nível possui uma cópia de parte da info armazenada no nível superior.



+ caches separados

L1 → (1º nível) cache integrada, de tamanho pequeno e tempo de acesso menor.

L2 → cache secundária, tamanho maior e tempo de acesso maior.

L3 → (fora) dentro do chip do processador, cache L2 dentro.

Registradores ↔ Memória  
gerenciada pelo compilador

Cache ↔ Memória Principal  
gerenciada pelo hardware

Memória Principal ↔ Disco  
gerenciada pelo hardware  
e SO ma memória virtual e  
pelo programador nos arquivos

A hierarquia de memória baseia-se no princípio da localidade:

→ Espacial: "Se um dado é referenciado, seus vizinhos logo tenderão a também ser". Move os blocos de palavras contíguas (juntas) p/ os níveis de hierarquia + próximos ao processador.

→ Temporal: "Um dado referenciado tende a ser referenciado novamente". Mantém itens de dados + recentemente acessados nos níveis da hierarquia + próximos ao processador.

• Cache deve

- verificar se tem cópia da posição de memória correspondente
- se tem, encontrar a posição da cache onde está esta cópia
- se não tem, trazer o conteúdo da memória principal e escolher posição da cache onde a cópia será armazenada

processador gera endereço de memória e envia à cache

Cache:  
- pouco espaço de armazenamento  
- alto custo financeiro  
- baixo tempo de acesso.



## Funcionamento da Cache

acerto (hit)  $\rightarrow$  dados são encontrados em algum bloco do nível superior.

hit rate (taxa de acerto)  $\rightarrow$  percentual dos acessos à memória encontrados no nível superior.

hit time  $\rightarrow$  tempo p/ acessar o nível superior, que consiste em:

$$\text{Tempo de aceso} + \text{Tempo p/ determinar hit/miss.}$$

falta (miss)  $\rightarrow$  dados não foram encontrados no nível superior e devem ser buscados no inferior.

miss rate (taxa de falta)  $\rightarrow 1 - \text{hit rate}$

miss penalty (penalidade por falta)  $\rightarrow$  tempo p/ trocar o bloco no nível superior + tempo p/ entregar o bloco ao processador.

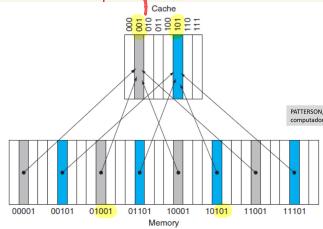
$$\text{Hit time} \ll \text{Miss Penalty}$$

## MAPEAMENTOS

**Mapeamento direto** - o local da memória cache onde vai escrever uma info cada posição (bloco) da memória principal é determinado pelo endereço

de cache guarda R extamente L palavras de memória principal à cache

- existe uma única posição em que aquele endereço pode ser copiado (Quantidade de blocos)



\* bit de tag  $\rightarrow$  ajuda a identificar com exclusividade qual é a palavra armazenada no cache. Função c/ o dado, ele armazena parte do endereço do bloco (bit de ordem + alta)  $\rightarrow$  chamados de tag.

$\hookrightarrow$  se não houver dados na posição: bit válido = 1 = presente, 0 = não presente

offset  $\rightarrow$  informações de byte (16 bytes/bloco  $\rightarrow$  4 bits de offset).

### Funcionamento:

- tenta achar uma instrução  $\rightarrow$  falta (temos que buscar a info na memória)

- buscamos essa info na memória p/ ser escrita na cache

- escrevemos o conteúdo no campo de dados, setar o bit de válido p/ 1 e copiamos p/tag os bits significativos do endereço.

**Exemplo:** 64 blocos  $\rightarrow$  6 bits de index

1b bytes/bloco  $\rightarrow$  4 bits de offset

Endereço 1200 é mapeado p/ qual bloco?

endereço de bloco  $\rightarrow \lfloor 1200/16 \rfloor = 75$

número do bloco  $\rightarrow 75 \text{ modulo } 64 = 11$



Cache completamente associativo - posso colocar a informação em qualquer lugar (é extremamente flexível). Não tem falhas por conflito, mas temos que comparar todos os possíveis p/ verificar a existência de uma informação  
 $\rightarrow$  de forma tag

Associatividade por conjunto - organiza o cache em grupos/conjuntos, mas temos a liberdade de escolher onde colocar a informação nesse conjunto.  
 $\rightarrow$  logo, temos

$\uparrow$  associatividade  $\downarrow$  miss rate  $\uparrow$  custo de tempo, acesso e complexidade

\* Encontrando um bloco:

$\hookrightarrow$  caches em hardware: reduzir comparações p/ reduzir custo.

$\hookrightarrow$  memória virtual: tabelas de lookup completas tornam a associatividade completa possível.

## Substituindo um bloco

Escolha de entrada que será substituída em caso de falha.

- 1) Least recently used (LRU): complexa e custosa para hardware com alta associatividade.
- 2) Aleatório: resultados próximos a LRU, mais fácil de implementar.

Memória virtual: aproximação de LRU com suporte de hardware.

• Quando ocorrer uma escrita, como manter a coerência com a memória principal?

- Write-through: a palavra é escrita tanto no bloco da cache, quanto no bloco da memória principal.

- Write-back: a palavra é escrita somente no bloco da cache. Quando este bloco for substituído, então a palavra será escrita na memória principal.

**Memória virtual:** somente write-back é viável dado o tempo de escrita em disco.

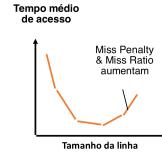
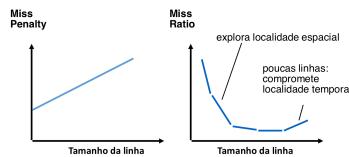
## Acesso à Cache:

- Quando a cache estiver sem espaço, qual bloco será substituído?
  - Mapeamento Direto: o bloco que estiver no slot
  - Associativa por conjunto e Completamente Associativa: usar uma política de substituição
    - LRU: substituir o bloco menos recentemente utilizado
      - Item (endereço de linha) vai para a frente da lista
    - LFU: substituir o bloco menos frequentemente utilizado
      - Contador incrementado quando bloco é acessado
    - FIFO: substituir o primeiro bloco que entrou na cache
    - Aleatório: escolher um bloco qualquer

**Linha de dados** = menor unidade de dados que pode ser transferida da memória principal para a cache.

↑ tamanho da linha ↑ localidade espacial ↑ miss penalty (demora + prazo preencher a linha) ↑ miss ratio (se: (tam. linha) ↑ tam cache)

$$\text{tempo médio de acesso} = \text{Hit Time} \times (1 - \text{Miss Ratio}) + \text{Miss Penalty} \times \text{Miss Ratio}$$



## MEMÓRIA VIRTUAL

- técnica que nos permite ver a memória principal como uma cache de grande capacidade de armazenamento.
- é apenas mais um nível na hierarquia de memória, que traz automaticamente p/ a Mem. Primária os blocos de info. necessários.
- o usuário tem a impressão de trabalhar c/ uma memória única, do tamanho da secundária mas c/ velocidade da primária.

Tempo médio de acesso Tma é dado por

$$Tma = Tm + (1 - h) Ts$$

onde  $Tm$  = tempo de acesso à MP  
 $Ts$  = tempo de acesso ao disco  
 $h$  = hit ratio

p.ex. se  $Tm = 20 \text{ ns}$ ,  $Ts = 20 \text{ ms}$ ,  $h = 0.9999$   
então  $Tma = 2,02 \mu\text{s}$  (100 x maior do que  $Tm$ )

**★ Miss penalty é muito maior. Se a info não está na memória, está no disco!**

#### • Logo:

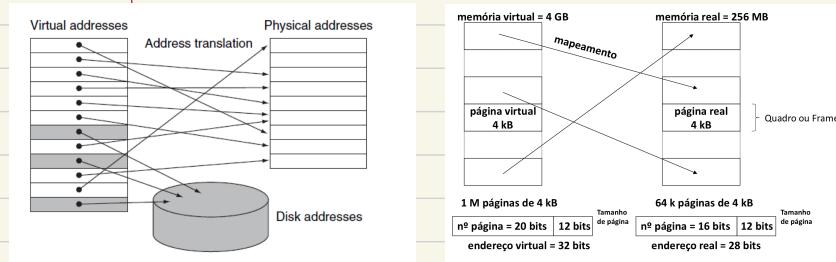
- miss ratio precisa ser bem menor do que em cache
- alta penalidade do miss => necessário buscar blocos maiores em disco
- princípio de **localidade** opera sobre blocos maiores de dados ou instruções e leva a hit ratios bem mais elevados
- **Mapeamento totalmente associativo das páginas**
- misses são tratados por software (há tempo disponível)
- técnica de escrita **write-through** não é uma opção. Usa-se **write-back**.

- > **endereço virtual**: gerado pelo programa (deve endereçar todo espaço em disco).
- > **endereço real**: endereço na memória principal (menor número de bits).

→ o SO usa a MMU

**★ A MMU (Memory Management Unit) gerencia a hierarquia da memória e faz o mapeamento do endereço virtual p/ o real.**

**Paginacão** → é uma forma de gerenciamento do uso de memória e mechanismos de tradução de endereços virtuais em reais  
 ↗ função de mapeamento



**★ Espaços de memória real e virtual divididos em blocos chamados páginas.**

- > mapeamento completamente associativo (+ eficiente e ajuda a diminuir a alta penalidade dos page faults).
- > page fault → ocorre quando a página virtual não está na memória principal.
- > page tables guardam a correspondência entre páginas virtuais e reais e permitem a tradução de endereços.
- > cada processo tem sua própria tabela de páginas.

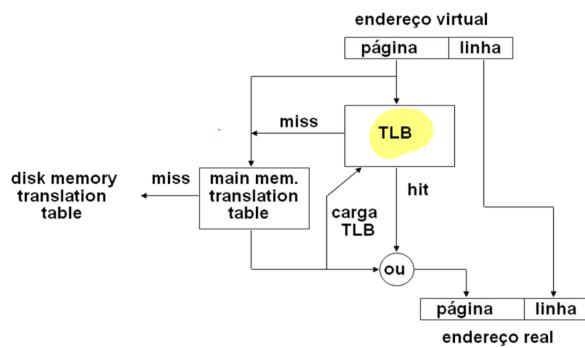
DMTT  
MMTT

# SUBSTITUIÇÃO DE PÁGINAS

**DMTT:** Disk Memory Translation Table → põe os mapeamentos de end. virtuais e reais do disco.

**MMTT:** Main Memory Translation Table → põe todos os mapeamentos entre end. virtuais e físicos da memória principal.

**TLB:** translation lookaside buffer → transforma um endereço virtual em um físico. pr a MMU (Memory Management Unit)

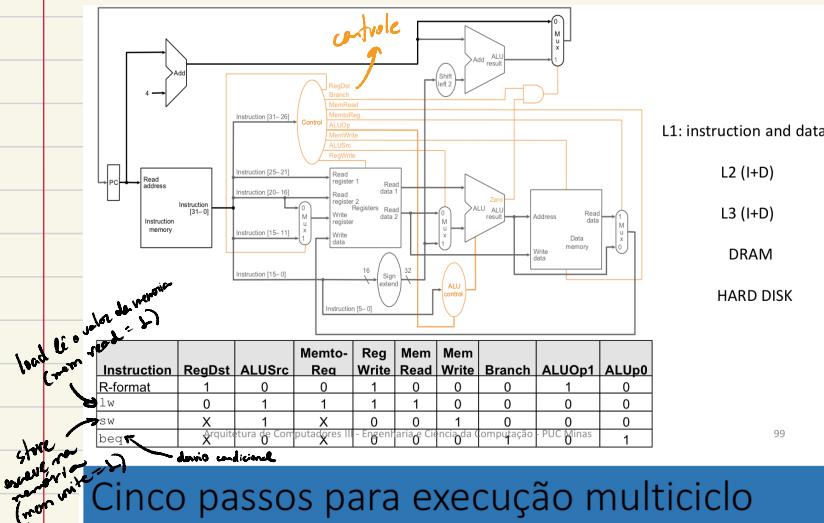


# PIPLINES

## O Processador: Caminhos de Dados e Controle

objetivo: aumento de desempenho  
(divisão de uma tarefa em N etapas)

arquitetura do processador MIPS



## Cinco passos para execução multiciclo

- Busca de instrução
  - Pode ser descrito de forma sucinta usando a "Linguagem Transferência-Registrador" - RTL "Register-Transfer Language"
  - $IR = \text{Memory}[PC]$
  - $PC = PC + 4$
- Decodifica instrução e Busca Registrador
  - Leia os registradores  $rs$  e  $rt$  para o caso de precisarmos deles
  - Compute o endereço de desvio no caso da instrução ser um desvio
  - RTL:
    - $A = \text{Reg}[IR[25-21]]$ ;
    - $B = \text{Reg}[IR[20-16]]$ ;
    - $ALUout = PC + (\text{sign-extend}[IR[15-0]] << 2)$ ;
  - Nós não ativamos linhas de controle baseados em tipo de instrução.
- Execução, Cálculo de Endereço de Memória, ou Conclusão de Desvio
  - A ULA está desempenhando uma das 3 funções, baseada no tipo de instrução
  - Referência à memória:  $ALUout = A + \text{sign-extend}[IR[15-0]]$ ;
  - Tipo-R:  $ALUOp1 = A \oplus B$ ;
  - Desvio: if ( $A == B$ )  $PC = ALUout$ ;
- Acesso à Memória ou Conclusão de instruções tipo-R
  - Carrega ou armazena na memória:  $MDR = \text{Memory}[ALUout]$ ; ou  $\text{Memory}[ALUout] = B$ ;
  - Finaliza instruções Tipo-R:  $\text{Reg}[IR[15-11]] = ALUout$ ;
- Passo de "Write-back"
  - $\text{Reg}[IR[20-16]] = MDR$ ;

Instruções levam de 3 a 5 ciclos.

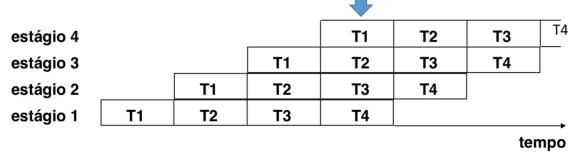
multiciclo  $\Rightarrow$  o add que antes levava 5ns c/ 1 ciclo, leva 3ns c/ 3 ciclos.

## Pipeline Escalar

→ você permite que + de 1 instrução seja executada na arquitetura, mas elas estão em estágios (e blocos) diferentes.

↳ 1 instrução por vez em cada estágio (por isso não tem o paralelismo de fato). → Continua terminando 1 instrução por vez, sequencialmente.

- Diagrama espaço - tempo



- bloco operacional e bloco de controle independentes pr cada estágio.
- necessidade de buffers entre os estágios.

s tarefas

N estágios

primeira tarefa: N ciclos de relógio

s - 1 tarefas seguintes: s - 1 ciclos de relógio

Tempo total com pipeline = N + (s - 1)

Tempo total sem pipeline = s N

de ter o pipeline  
não teria feito.

$$\text{Speed-up teórico} = \frac{s N}{N + (s - 1)}$$

\* o compilador define a direção + provável

↳ inicio de laço (ou devia pr frente) → + improvável

↳ final de laço (ou devia pr trás) → + provável

\* predição dinâmica é melhor (vai se ele já executou essa instrução no passado e outra só desvia ou não) → se derrete, vc consegue avançar na memória c/ a esperança deles serem cache

→ outra na tabela look-up

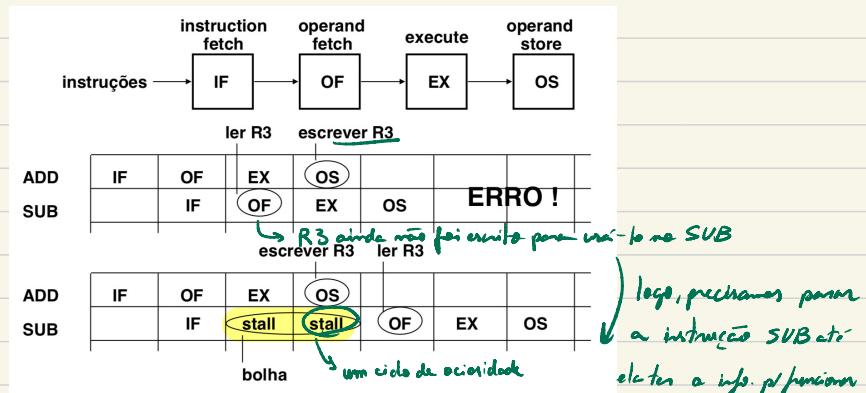


## Dependências

→ problema: instruções consecutivas podem fazer aceno aos mesmos operandos (a execução da instrução seguinte pode depender de um operando calculado pela instrução anterior).

## > Vendoclinia:

1. ADD R3, R2, R1;  $R3 = R2 + R1$  } a instrução 2 depende do valor de R3  
 2. SUB R4, R3, 1;  $R4 = R3 - 1$  } calculado pela instrução 1.  
 (imediato) "read-after-write hazard"



## > Antidependência

1. ADD R3, R2, R1 ;  $R3 = R2 + R1$
  2. SUB R2, R4, L ;  $R2 = 4 - 1$

\* Só é um problema no pipeline superescalares.

> De Saïda:

1. ADD R<sub>3</sub>, R<sub>2</sub>, R<sub>1</sub>  
 2. SUB R<sub>2</sub>, R<sub>3</sub>, L  
 3. ADD R<sub>3</sub>, R<sub>2</sub>, R<sub>5</sub>

} também não dá problema no superregister  
 Le 3 saídos em memo operando em R<sub>3</sub>  
 } é preciso que o resultado em R<sub>3</sub> esteja

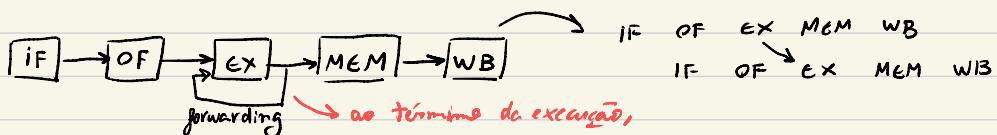
"write-after-write hazard"

## Forwarding (adiantamento de dados)

ADD R3, R2, R0 }  
 SUB R4, R3, 8 }

IF OF EX MEM WB  
 IF ← ← OF  
 EX } problema que causa bolha pq dividimos o tempo de leitura e escrita

Caminho interno dentro do pipeline entre a saída da ALU e a entrada da ALU evita stall (bolha) do pipeline



forwarding

ao término da execução,

eu tento a leitura de dado

que foi calculada adiantando p/

a entrada da execução → (intento adiantando p/ quem

está vindo, p/ não barrar o OF)

IF → instruction fetch (busca da instrução)

OF / ID → instruction decode / operand fetch (decodifica a instrução p/ identificar o tipo de operação e os operandos envolvidos).

EX → execute (a operação é executada).

MEM → memory access (caso a instrução envolva memória (load ou store), utiliza ela).

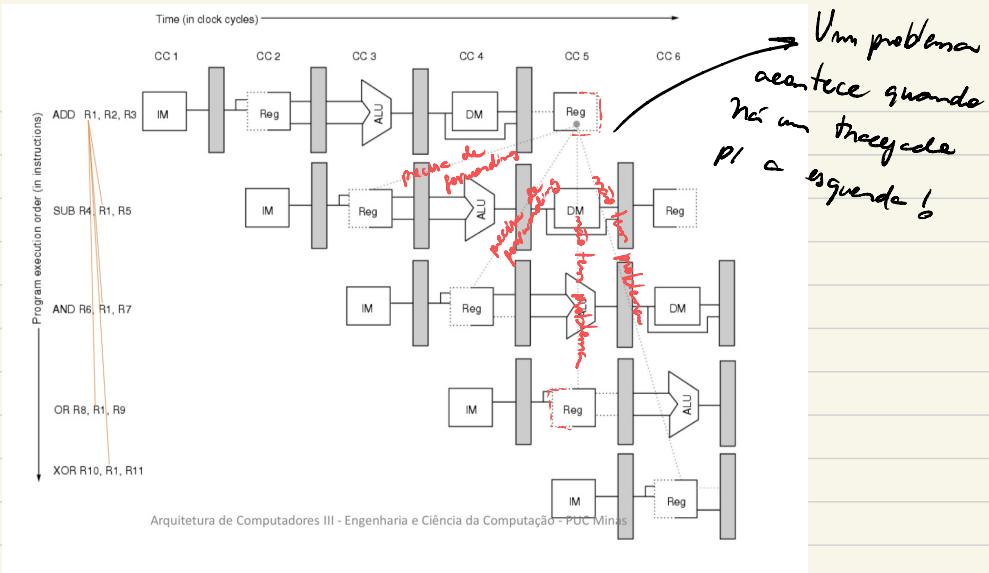
WB → write back (o resultado da operação é escrito de volta no banco de registradores, tornando-o disponível p/ futuras instruções).

(adiantamento de dados)

\* Com o Forwarding, o dado é enviado diretamente da saída da ALU ou da memória p/ os estágios que necessitam dele, sem esperar pela conclusão do ciclo completo.

\* O fluxo faz uma comparação de endereços de saída de um barramento de instruções p/ os requisitados de próx. barramento e verifica se precisa ou não de adiantamento de dados.

\* Forwarding nem sempre é possível!



\* Unidade de Detecção de Hazard  $\Rightarrow$  ocorre um stall, deixando que uma instrução que não encerre a má prossiga.  
 ↳ isso acontece somente se eu tiver um load (pq eu só leio na memória c/ o load).

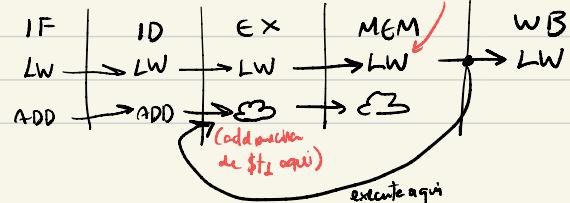
### Forwarding Necessário:

- ↳ dependências RAW (Read After Write)  $\Rightarrow$  (add não disponível no EX/MEM ou MEM/WB e pode ser encaminhado diretamente p/ a entrada da ALU da instrução SUB, evitando uma bolha).
- ↳ operações consecutivas c/ uso imediato de registradores

### Forwarding Não é suficiente:

- ↳ hazards de carga-uso (load-use)  $\Rightarrow$  uma bolha é obrigatória, mesmo c/ forwarding
- ↳ dependências WAR e WAW
- ↳ faltas de predição de derriva (interrompem o fluxo do pipeline, gerando bolhas até que o fluxo se resolva).

LW \$t<sub>2</sub>, 0(\$t<sub>2</sub>)  
 ADD \$t<sub>3</sub>, \$t<sub>1</sub>, \$t<sub>4</sub>



## Descartando Instruções

→ IF Flush → a instrução não é executada e gera uma bolha (executa um 0).

### Casos que exigem descarte de instruções

#### 1. Hazard de Controle (Desvios)

- Quando há um desvio condicional ou incondicional, o pipeline pode começar a buscar e decodificar instruções erradas antes de saber se o desvio será tomado ou não.
- Exemplo:

```
text
BEQ $t1, $t2, label    # Desvio condicional
ADD $t3, $t4, $t5      # Instrução seguinte (pode ser inválida)
```

- Se o desvio for tomado, as instruções já carregadas no pipeline são inválidas e precisam ser descartadas (*flush*), gerando bolhas nos estágios afetados 1 5 8.

## Fórmulas:

$$\text{Hit ratio} = \frac{\text{n. de hits}}{\text{hits + misses}}$$

(taxa de acerto)

$$\text{Miss ratio} = \frac{\text{n. de misses}}{\text{hits + misses}}$$

(taxa de erro)

$$\text{AMAT} = \text{Hit Time} + \text{Miss ratio} \cdot \text{Miss Penalty}$$

Average Memory Access Time

$$\text{Tempo efetivo} \quad T_{ce} = hT_c + (L - h) T_{m\alpha}$$

de acesso à memória cache

Considerando misses

Tempo médio de acesso

$$T_{m\alpha} = T_m + (L - h) \cdot T_s$$

tempo de acesso à MP → tempo de acesso à disco

hit ratio

\* P/ descobrir a qntd de bits da tag:

1º KB totais  $\rightarrow$  1024 (p/ transformar em Bytes)

2º  $\frac{\text{Bytes totais}}{\text{tamanho do bloco (em Bytes)}}$  = qntd de blocos = qntd de tags

3º  $\log_2(\text{qntd de tags}) = \text{bits da tag}$

\* P/ descobrir a qntd de bit da tag de coj. associativa por copart:

1º KB totais  $\rightarrow$  1024 = Bytes

2º -  $\frac{\text{Bytes totais}}{\text{tamanho de bloco (em bytes)}} \cdot \frac{\text{qntd de vias}}{\text{qntd de copartes}} = \text{qntd de tags}$

3º -  $\log_2(\text{qntd de tags}) = (2^{\text{bits}} = \text{qntd de tags})$

\* P/ recalcular CPI:

- miss na busca de instruções  $\rightarrow$  afeta todos os instruções
- miss na busca de dados  $\rightarrow$  afeta load/store

► Penalidade =  $\% \text{ dos instruções afetadas} \cdot \text{taxa de miss} \cdot \text{qntd de ciclos de penalidade}$   
(ex: 0,4 (20%))

► CPI novo = CPI + Penalidade anterior

\* Número de ciclos p/ executar as instruções em um pipeline de 5 estágios sem bolhas:

$\text{nº de ciclos} = \text{nº de instruções} + (\text{qntd de estágios} - 1)$

$\hookrightarrow (5-1)$

Imo p/ cada instrução q<sup>t</sup> tada por miss

# Como posso verificar se nenhuma bolha foi gerada

Perplexity

Para verificar se nenhuma bolha (stall) foi gerada em um pipeline, siga estas etapas:

## 1. Identificar Dependências entre Instruções

Analise o fluxo de instruções para detectar três tipos de dependências:

- **RAW (Read After Write):** Uma instrução precisa ler um registrador que ainda não foi escrito por uma instrução anterior.
- **WAR (Write After Read):** Uma instrução escreve em um registrador que outra instrução ainda precisa ler.
- **WAW (Write After Write):** Duas instruções escrevem no mesmo registrador em ordem incorreta.

### Exemplo da Questão 11:

```
asm
1. ADD R1, R2, R3 ; Escreve em R1
2. LW R5, 0(R2) ; Lê R2, escreve em R5
3. SW R5, 0(R4) ; Lê R5 e R4
4. SUB R4, R1, R5 ; Lê R1 e R5, escreve em R4
```

- **RAW entre ADD (1) e SUB (4):** SUB precisa de R1, que é escrito por ADD.
- **RAW entre LW (2) e SW (3):** SW precisa de R5, que é escrito por LW.
- **RAW entre LW (2) e SUB (4):** SUB precisa de R5, que é escrito por LW.

## 2. Verificar a Presença de Forwarding

O adiantamento de dados (forwarding) permite que resultados sejam passados diretamente entre estágios do pipeline, evitando bolhas.

Se o pipeline possui forwarding:

- Dependências RAW podem ser resolvidas sem stalls.
- Dependências WAR/WAW são irrelevantes em pipelines clássicos (RISC).

## 2. Hazards de Dados

Causados por dependências entre instruções que não podem ser resolvidas imediatamente.

### Tipos de Hazards de Dados:

Tipo	Descrição	Exemplo	Solução
RAW (Read After Write)	Uma instrução tenta ler um registrador antes que ele seja escrito por uma instrução anterior.	ADD R1, R2, R3 → SUB R4, R1, R5	Forwarding ou inserção de bolhas.
WAR (Write After Read)	Uma instrução tenta escrever em um registrador antes que outra instrução o leia.	Raro em pipelines clássicos.	Reordenação de instruções (out-of-order).
WAW (Write After Write)	Duas instruções tentam escrever no mesmo registrador fora de ordem.	ADD R1, R2, R3 → MUL R1, R4, R5	Reordenação de instruções.

## 3. Hazards de Controle

Causados por desvios (branches/jumps) cujo destino só é conhecido tarde no pipeline.

### Exemplo:

Uma instrução `beq` resolve o desvio no estágio EX (3º estágio). Se o desvio for tomado, as duas instruções já buscadas (IF e ID) são invalidadas, gerando 2 bolhas:

```
text
beq $t0, $t1, Label # Desvio resolvido no EX
add $t2, $t3, $t4 # Instrução inválida (flush)
sub $t5, $t6, $t7 # Instrução inválida (flush)
```

### Solução:

- Predição de desvios (estática ou dinâmica).
- Execução especulativa (executar instruções supondo que o desvio não será tomado).

## Quando as Bolhas São Inevitáveis?

Cenário	Ciclos de Bolha	Motivo
Load-Use Hazard (sem forwarding)	1-2	Dependência RAW entre <code>lw</code> e instrução que usa o registrador.
Desvio não previsto	2-3	Pipeline precisa descartar instruções já buscadas.
Acesso a memória lenta	Variável	Espera por dados da memória principal.

## Exemplo Prático de Bolhas no Pipeline

Considere o seguinte código sem *forwarding*:

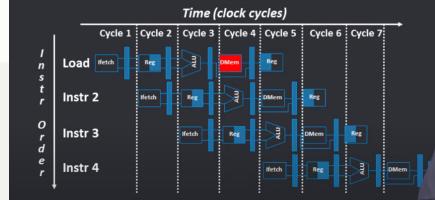
text

```
1. lw $t1, 0($t0)    # IF | ID | EX | MEM | WB
2. add $t2, $t1, $t3  #     IF | ID | STALL | STALL | EX | MEM | WB
```

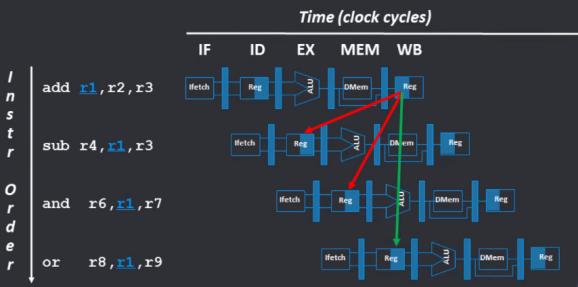
- A instrução 2 precisa de `$t1`, que só estará disponível após o WB da instrução 1.
- Bolhas necessárias: 2 ciclos de stall entre ID e EX da instrução 2.

## Structural Hazards

- Suppose 1 memory w/ 1 port

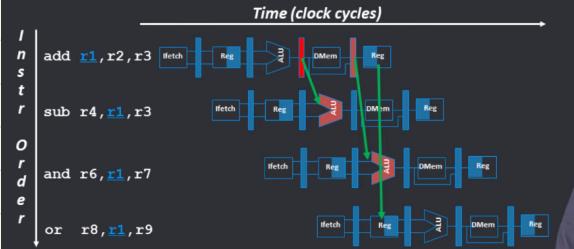


## Data Hazards

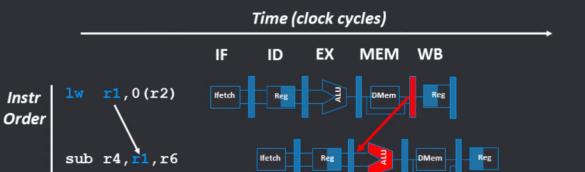


## Forwarding / Bypassing

- Don't wait until result written back to RF but forward it to next stage immediately



## Data Hazard Even with Forwarding



- Load-use data hazard

## Pipeline Interlock for Load-Use Hazard

