

$$\text{Avaliação} \text{ Desempenho: } T_{exec} = \#CC \times T_{cc} \quad T_{cc} = 1/f \quad \#CC = CPI \times \#I / T_{exec} = \frac{\#I \times CPI}{f}$$

$$MIPS = \frac{\#I}{T_{exec} \times 10^6}$$

CPI divisões > CPI adições

CPI memória > CPI registros

CPI V.F.  $\geq$  CPI inteiros

$$\text{Hit rate} = \frac{\# \text{ Hits}}{\# \text{ Acessos}}$$

$$\text{Miss Rate} = 1 - \text{Hit Rate}$$

### Hierarquia de Memória:

Localidade espacial: Se um elemento de memória é acedido pelo CPU, então elementos com endereços na proximidade serão, com grande probabilidade, acedidos proximamente. (Analogia: localidade temporal). Um elemento da memória será acedido pelo CPU, com grande probabilidade de novo proximamente. Ex: As instruções e contadores dos ciclos.

Com todo isto é o híato proc-memória ou memory gap, dotou-se as máquinas com vários níveis de memória **PROCESSADOR  $\Rightarrow$  CACHE  $\Rightarrow$  MEMÓRIA CENTRAL  $\Rightarrow$  DISCO**

Falando da cache diz-se um hit quando o elemento está na cache e miss quando não está, tendo de ir a um nível inferior de memória. Com isto temos  $CPI_{\text{cache}} = CPI_{\text{CPU}} + CPI_{\text{MEM}}$ , que é os ciclos que demorou a buscar à memória central os dados para a cache. (memory stall cycle)  $CPI_{\text{MEM}} = (\text{missrate} \cdot t_{\text{latency}} + t_{\text{miss}}) \cdot \text{miss penalty}$  |  $T_{exec} = \#I \cdot (CPI_{\text{CPU}} + CPI_{\text{MEM}}) \cdot T_{cc}$

A memória (central) tem  $M$  bytes, logo o endereço tem  $m = \log_2(M)$  bits. A cache organiza-se em linhas de  $B$  bytes, logo  $b = \log_2(B)$  bits identificam bloco / byte. As linhas encontram-se em sets, logo  $s = \log_2(S)$  bits identifica um set. (Cada set contém  $E$  linhas). ( $S, E, B, m$ ) Capacidade da cache é  $S \cdot E \cdot B$  e o #Linhas =  $S \cdot E$ .

MAP. Direto ( $t=1$ )  $\rightarrow$  Cada endereço mapeia numa só linha, logo  $t$  colisões. O hit time por sua vez é pequeno, dado que apenas se verifica o set index. (Cada linha tem o valid depois da tag).

MAP. Associativo ( $s=0$ )  $\rightarrow$  Qualquer endereço pode mapear em qualquer linha ( $1, E, B, m$ ). Com isto, reduz-se as colisões mas o hit time aumenta para a procura de cada tag (que também aumenta).

MAP. n-way associativo  $\rightarrow$  os dois métodos, ou seja, sets com linhas, dessa forma terá menos colisões e menor tag / hit time porque terá liberdade de linhas, mas mantendo os endereços em ordem, com uma tag identificativa da linha no set.

Write-through  $\rightarrow$  escreve na cache e mem., logo leva tanto tempo como o nível mais lento. Para isso usa-se write-buffers para o processador e a escrita estarem assincronos. Perde muito tempo escrevendo na mem. é fácil. não aumenta miss penalty

WriteBack  $\rightarrow$  apenas escreve na mem quando o bloco necessita de ser substituído, logo - escrever + velocidade (cache > mem.). Aumenta a miss penalty. + complexo. LRU - substituir o que não é usada há menos tempo

Pipelining: Uma instrução é executada, fetch, decode, execute, write. Estas fases podem ser agrupadas ou reordenadas para permitir a execução das instruções em vários estágios intercalados.

Tempo de ciclo de relógio = logica combinatoria do estágio + Tregisto. Latência = nº estágios \* Tcc.

Como os estágios mais lentos ditarão a Tcc, a maioria dos estágios fica inativo. CPI = 1 ideal mas é 01 porque há

Dependências de dados  $\Rightarrow$  stalling  $\rightarrow$  injetando bolhas / Realignment / Forwarding de dados, eliminando penalizações

Dependências de controle  $\Rightarrow$  previsão estática, dinâmica, branch delay slot, instruções constantes

Superalcançabilidade: Várias pipelinas TAKEN=6x TAB2.1

**STATIC ISSUE**

- compilador-

- VLIW = #I = # Pipelines

- exec. especulativa = + código para a especulação

- Loop unrolling + register renaming = várias cópias de um bloco

- Esto danará talvez - mais instruções + instruções independentes

CPI

**DYNAMIC ISSUE**

- processador

- só faz WB quando há confirmação (buffering)

- static in-order - em ordem do programa

- constante disponibilidade operandos e U. Funções

- dynamic out-of-order: as instruções são reordenadas para maximizar CPI, garantindo a execução do programa muito eficiente.

## Arq. Computadores

- Uma máquina com  $N$  bits admite  $2^N$  estados possíveis.
- Hierarquia de Memória, com as chamadas caches  $L_1, L_2, L_3$ .

## Medição do Desempenho

CPI - Cycles per instruction

Algumas instruções exibem diferentes CPI

CPI divisões > CPI adições

CPI acessos à memória > CPI acesso a registos

CPI vírgula flutuante > CPI operações internas

CPI é um valor médio.

Previsão do Tempo de Execução (Texec) :  $\# \text{ClockCycles} \times T_{cc}$

$$\begin{matrix} \uparrow & \uparrow \\ \text{nº médio} & \text{Período do relógio do CPU} \\ \text{de ciclos de} & \left(\frac{1}{f}\right) \\ \text{relógios} & \end{matrix}$$

Depende do CPI e do #I, logo

$$\#CC = CPI \times \#I$$

↓

$$\text{Texec} = \frac{\text{CPI} \times \#I}{f \leftarrow Hz}$$

Estas métricas não devem ser analisadas isoladamente, devendo sempre ser consideradas em conjunto, pois Texec é a única métricaável para avaliar o desempenho de um PC.

Alguns exemplos: Aumentar  $f$  (diminuir  $T_{cc}$ ) implica um aumento de CPI.

Se o  $T_{cc}$  diminui mas o tempo para ir à memória é igual, não será necessário mais CPI, logo Texec não diminui para metade.

Diminuir  $\#I \rightarrow$  Aumento CPI. Ao tornar as instruções mais complexas, será necessário mais CPI para a completar.

MIPS (Milhões de Instruções p/seg) - uma métrica enganadora

$$\frac{\#I}{\text{Texec} \times 10^6}$$

CPE - Ciclos por elemento

Wall Time → Tempo decorrido desde inicio até fim do programa.

Susceptível à carga do sistema ( $\epsilon/s$ , outros processos, etc.)

Tempo de CPU → Tempo dedicado a este processo

Menos sensível à carga do sistema

Os processadores têm lógica e contadores de eventos que são atualizados a cada ciclo

Para isso existe o PAPI para aceder a esses contadores.

### Hierarquia de Memória.

"A memória é incapaz de alimentar o processador com instruções e dados a uma taxa suficiente para o manter ~~ocioso~~ ocupado". Isto acontece porque o desempenho dos processadores aumentou exponencialmente ao longo dos anos, as memórias nem tanto. Isso criou um bottleneck enorme porque existe pouca largura de banda entre processador-memória. O memory-gap ou hiato de processador-memória é o grande obstáculo à melhoria dos sistemas de computação.

Existem a DRAM, não persistente → tempos de acesso elevados (mais capacidade) SRAM, muito persistente → " " " " reduzidos (menos capacidade)

### Localidade

"Os programas bem escritos tendem a aceder a dados que estão próximos (em termos de endereços de memória) de outros dados acedidos recentemente, bem como a referenciar repetidamente os mesmos dados"

Este princípio divide-se em: temporal e espacial

Localidade temporal - um elemento da memória acedido pelo CPU será, com grande prob., acedido de novo proximamente. Ex: as instruções dentro dos ciclos bem como as variáveis para contadores dos ciclos.

$for(i=0; i < N; i++)$  são bons exemplos de localidade temporal.

$\rightarrow a[i] = i;$

Localidade Espacial: se um elemento de memória é acedido pelo CPU, então elementos com endereços na proximidade serão, com grande probabilidade, acedidos proximamente. Ex: as instruções são acedidas em sequência, na maioria parte dos arrays.

Caso seja um array multidimensional int a[3][4] em row-wise (por linhas)

a	0	1	2	3
0	0	4	8	12
1	16	20	24	28
2	32	36	40	44

Caso seja:

→ for(j=0; j < 4; j++) (com esta disposição)

→ for(i=0; i < 3; i++) (não se aproveita)

a[i][j]++; (a vantagem de estarem)

em endereços perto porque  
se salta sempre de linha

e elas estão em row-wise

(com esta disposição)

já se aproveita o row-wise  
porque se está a ser  
acessados endereços na  
proximidade.

a	0	1	2	3
0	0	4	8	12
1	16	20	24	28
2	32	36	40	44

Caso seja:

→ for(i=0; i < 3; i++)

→ for(j=0; j < 4; j++)

a[i][j]++;



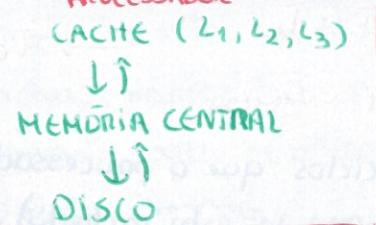
j tem de andar

mais rápido que i

(colunas 1º que as linhas)

Com tudo isto é como o bloco processador-memória é necessário ter  
programas bem definidos para exibir uma boa localidade, concentrando os acessos  
a endereços próximos e também acentuando os mesmos endereços repetidamente  
especial temporal

Desta forma, dotou-se as máquinas com vários níveis de memória, há  
mais rápidos (e maiores e menor capacidade) quanto mais perto do processador.  
Cada nível contém uma cópia do código e dados <sup>PROCESSADOR</sup> mais usados em cada instante,  
explorando assim a localidade



Falando da cache, temos algumas terminologias:

Linha → a cache está dividida em linhas. Cada linha tem o seu endereço e tem a  
capacidade de um bloco.

CACHE

Bloco → Quantidade de info. que é transferida de cada vez da memória central para a cache. É igual à capacidade da linha.

000  
001  
010  
011  
100  
101  
110  
111

000
001
010
011
100
101
110
111

Bloco

Hit → Diz-se hit quando o elemento de memória do CPU  
encontra na cache, sendo necessário ir a um nível inferior.

Miss → Diz-se miss quando o elemento de memória do CPU  
não se encontra na cache, sendo necessário ir a um nível inferior.  
A hierarquia de memória

hit rate → Percentagem de hits ocorridos

$$\text{Hit Rate} = \frac{\# \text{hits}}{\# \text{acessos}}$$

miss rate → Percentagem de miss

$$\text{Miss Rate} = 1 - \text{Hit Rate}$$

hit time → Tempo necessário para determinar se o elemento a que o CPU está a aceder se encontra ou não na cache.

Miss Penalty - Penalização incorrida para aceder a um dos blocos superiores da hierarquia, ocorre um miss

com isto quando ocorre Localidade Temporal

A 1ª vez o endereço é acedido, é carregado do nível inferior para a cache - cold miss

O próximo acesso encontra os dados na cache - hit

(exclui se tenha havido colisão, resultando numa miss)

Localidade Espacial: Quando um endereço é carregado para a cache, é carregado um bloco de endereços consecutivos.

O acesso seguinte a um endereço na vizinhança resulta num hit

$$T_{\text{exec}} = \#I \cdot CPI \cdot T_{\text{cc}} \rightarrow T_{\text{exec}} = \#I \cdot (CPI_{\text{CPU}} + CPI_{\text{MEM}}) \cdot T_{\text{cc}}$$

$$CPI = CPI_{\text{CPU}} + CPI_{\text{MEM}}$$

CPI<sub>CPU</sub> - nº de ciclos que o processador necessita para executar aquela instrução (o hit time já está incluído)

CPI<sub>MEM</sub> → nº de ciclos que o processador para, em média, à espera que os dados da memória central, porque não os encontrou na cache. São chamados memory stall cycles ou wait states

Como os dados e instruções comportam-se de forma diferente temos:

$$CPI_{\text{MEM}}: (\text{miss rate}_i + \gamma_{\text{Mem}} \cdot \text{miss rate}_D) * \text{miss penalty}$$

Logo temos

$$T_{\text{exec}} = \#I \cdot [CPI_{\text{CPU}} + (\text{mA}_i + \gamma_{\text{Mem}} \cdot \text{mA}_D) * mp] * T_{\text{cc}}$$

$\downarrow$   
acesso  
latência

↑  
misspenalty  
term de SCR  
em CC.

## Organização da cache - Hierarquia de Memória

A memória (central) tem  $M$  bytes.

O endereço tem  $m = \log_2(N)$  bits

A cache organiza-se em linhas de  $B$  bytes.

$b = \log_2(B)$  bits identificam um bloco / byte.

As linhas encontram-se agrupadas em  $S$  sets

$s = \log_2(S)$  identificam um set.

Cada set contém  $E$  linhas

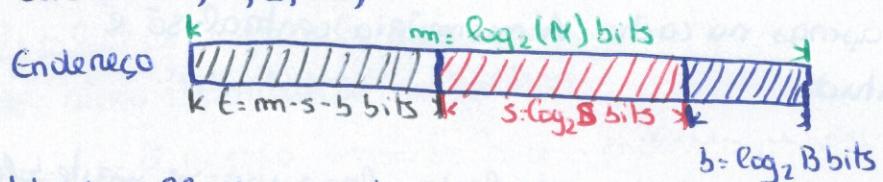
A organização da cache é caracterizada por:

$(S, E, B, m)$

No total de linhas é  $S * E$

Capacidade da cache =  $S * E * B$

Cache  $(S, E, B, m)$



block offset -  $b$  bits identificam o byte dentro do bloco

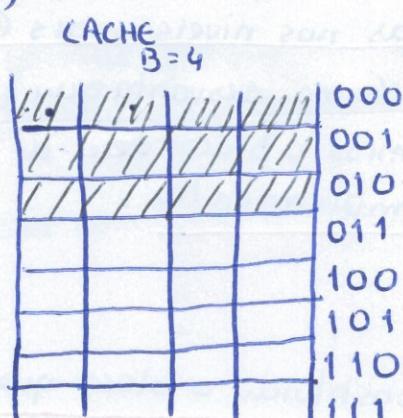
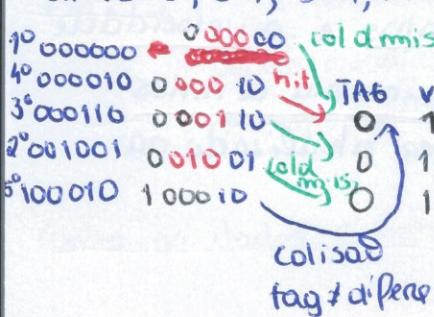
set index  $\rightarrow s$  bits identificam o set dentro da cache

tag  $\rightarrow t$  bits para a tag

## Mapeamento Direto

Organização da cache com  $E=1$ . Cada set só tem 1 linha

Ex  $(S=8, E=1, B=4, m=6)$

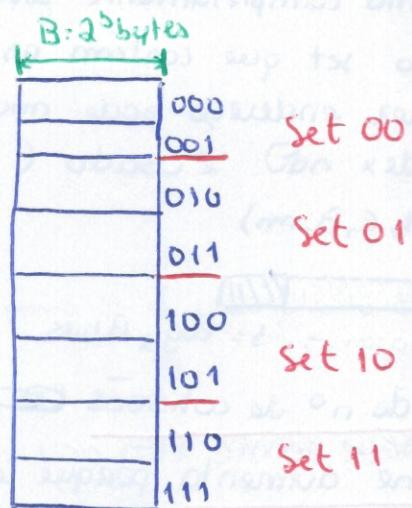


- cada endereço de memória mapeia numa só linha

- nº elevado de colisões

- o set index, e a linha correspondente é só determinado pelos bits,  $\log_2$  hit time é curto.

Cada linha de cache tem um bit extra (valid) que indica se os dados dessa linha são válidos



amento completamente associativo

único set que contém as linhas todas ( $S=1$ )

qualquer endereço pode mapear em qualquer linha

set index não é usado ( $S=1 \Rightarrow S=0$ )

Cache ( $T, E, B, m$ )



$$\text{tag} = m_{S-B} \quad b = \log_2 B \text{ bits}$$



Reduzido nº de colisões (~~quando o tag é igual e o bloco é diferente~~)

hit time aumenta porque é necessário comparar todas as tags da cache.

Para combater este dois métodos, surgiu o mapeamento n-way set associative.

A cache é dividida em  $S$  sets de  $E$  linhas. Desta forma reduz as colisões, e o nº de bits de tag e o hit time.

Quando o CPU troca uma palavra ou byte na cache?

1. Write-through - a info. é escrita na cache e na memória central.

2. Write-back - a info. é escrita apenas na cache. A memória central só é atualizada quando o bloco for substituído

O Write-through leva tanto tempo quanto o nível mais lento, logo usa-se write-buffer.

Dessa forma o processador continua a executar e a escrita nos níveis mais lentos é assíncrona a isso. (o write-buffer pode encher). Várias escritas sucessivas no mesmo bloco implica perder muito tempo escrevendo para os níveis mais lentos. Mais simples de implementar e não aumenta a miss penalty.

O Write-back apenas escreve na memória quando o bloco necessita de ser substituído, logo existe menos escritas nos níveis mais lentos e a velocidade de escrita é a velocidade da cache. Como desvantagem, aumenta a miss penalty, porque quando existe um miss o bloco tem de ser atualizado na memória e é mais complexo de implementar.

Políticas de Substituição

• RANDOM

• LRU (Least Recently Used) - é substituído o bloco que não é usado há mais tempo

## Aprendimento de Instruções

Uma instrução é executada: Leitura (fetch), decodificação, leitura dos operadores, execução (ALU) e escrita dos resultados.

Estas fases podem ser agrupadas ou reordenadas para permitir a execução das instruções em vários estágios encadeados ← Pipeline

Dividir o processo em estágios independentes.

Objetos movem-se através dos estágios

Em cada instante, múltiplos objetos são processados simultaneamente

Lógica Combinatória + Reg  $\stackrel{100\text{ps}}{\longrightarrow}$ ,  $\stackrel{20\text{ps}}{\longrightarrow}$ , ciclo  $\geq 120\text{ps}$ , Latência = nº estágios \* ciclo = 360ps

Estágio-

$$f_{\text{req}} \leq \frac{1}{120\text{ps}}$$

$$f_{\text{req}} \leq 8.336\text{Hz}$$

OP<sub>1</sub> A

OP<sub>2</sub> B

OP<sub>3</sub> C

→

A B C

A B C

A B C

Três instruções em simultâneo.

Período do relógio é limitado pelo estágio mais lento.

Logo, os outros estágios ficam inativos durante parte do tempo

Não se pode estar a ter estágios infinitos ~~para~~ porque existem custos de registo.

O pipeline permite reduzir o período do relógio, mas o CPI com pipeline costuma ser superior a 1, devido às dependências entre instruções

Dependências de controlo - jxx (jump (if not) zero/equal)

Pode acontecer stalling.

Os registos são escritos apenas no estágio de WRITEBACK

Se uma instrução tenta ler um registo antes da escrita estar terminada é necessário resolver a dependência RAW (Read after Write), injetando "bolhas" (NOPs) no estágio de execução para adiar até ao ciclo imediatamente a seguir à escrita.

Para isso temos:

Dependências de controlo → Provemos que o salto é sempre tomado, corrigindo quando a instrução de salto terminar o estágio de execução. Caso esteja errada, colocarmos "bolhas/nops" nos erros que não há problema dado que as instruções ainda não fizeram WRITEBACK

Dependências de dados fazemos data forwarding, que consiste em passar o valor necessário para o estágio de DECODE (caso ele esteja no EU ou W).

### - Resumo: Pipeline

Executa n instruções simultaneamente em estágios diferentes

• Permite aumentar a frequência do relógio

• Dependências de dados:

→ stalling: injeção de bolhas (NOPs)

→ realimentação: elimina as penalizações, fazendo forwarding.

• Dependências de controle:

→ Saltos condicionais implicam previsão do salto

→ Previsão errada implica stalling / "bolhas" do pipeline.

## ~~Scalability~~ - vários pipelines

### STATIC ISSUE

- compilador -  $NIW = #I = #P_{\text{pipelines}}$
- exec. especulativa + código
- Loop Unrolling + register renaming
- =  
Várias cópias dum ciclo

### DYNAMIC ISSUE

- processador
- só faz WB quando há confirmação (buffering resultados)
- em tempo de execução decide:
  - Static in-order = em ordem (de acordo com a disponibilidade)
  - Dynamic OUT-OF-ORDER = com coordenação (máxima CPI)

+ instruções independentes

< CPI