Arquitetura II

CI086 – Tópicos em Arquitetura de Computadores
CI702 – Arquitetura de Computadores

Roberto Hexsel
roberto@inf.ufpr.br
www.inf.ufpr.br/roberto/CI086.html

Por que estudar Arquitetura?

Por que estudar Arquitetura?

- ser um programador competente:
  usar caches e memória virtual de forma vantajosa;
- aprender algoritmos e técnicas para alta velocidade:
  tirar proveito da hierarquia de memória;
- usar bem recursos da máquina:
  idem com relação a E/S, segmentação;
- poder trabalhar com eletrônica embarcada:
  mercado novo e com poucos programadores competentes;
- entender artigos das revistas da ACM, IEEE;
- e, principalmente, ser feliz a vida é de vocês...

Bibliografia

texto:

Computer Architecture: A Quantitative Approach, H&P-QA
J L Hennessy e D A Patterson, 3a Ed, Morgan Kaufmann, 2003.
cuja tradução ruim é
Arquitetura de Computadores – uma abordagem quantitativa,
J L Hennessy e D A Patterson, 1a Ed, Campus, 2003,

textos auxiliares:

Computer Organization and Design, P&H-COD
D A Patterson e J L Hennessy, 3a Ed, Morgan Kaufmann, 2005
existe tradução da segunda edição

Readings in Computer Architecture, 🖗 na biblioteca
M D Hill, N P Jouppi, G S Sohi, Morgan Kaufmann, 2000.

Programa

- ⋆ Tendências tecnológicas (1) texto em `www.*://roberto/CI702.html`
- ⋆ avaliação de desempenho (1)
- ⋆ conjuntos de instruções - MIPS64 (2) TODOS DEVEM ESTAR COM O LIVRO!
- ⋆ revisão pipelines, processadores super-escalares (3)
- ⋆ paralelismo no nível de instrução, hw e sw (4)
- ⋆ hierarquia de memória, caches, mem virtual (4)
- ⋆ 03mai prova
- ⋆ multiprocessadores, mem logicamente compartilhada (4)
- ⋆ sistemas de E/S, discos (2)
- ⋆ multicomputadores e clusters, mem logicamente distribuída (2)
- ⋆ redes diretas (2)
- ⋆ 21jun prova e 04jul entrega do trabalho
- ⋆ 03jul final

Material adicional para o curso

`http://www.inf.ufpr.br/roberto/CI702.html`
`http://www.inf.ufpr.br/roberto/CI086.html`
`/public/soft/linux/simplescalar`
`~roberto/ci702`

lista de e-mail em `roberto-tac@inf.ufpr.br`

não guardem este endereço em máquinas com uindous

Organização de Computadores

Modelo de Von Newman .

- computador com programa armazenado (1945)
- memória é um vetor de bits
  - interpretação dos bits definida pelo arquiteto e programador/compilador
- parte da memória contém instruções
- parte da memória contém dados

Definição de Arquitetura de Computadores

1. arquitetura do conjunto de instruções (CdI):
   conjunto de instruções e registradores visíveis ao programador
   Instruction **S**et **A**rquitecture = ISA
2. organização:
   blocos como sistema de memória, barramentos, CPU
   mais de uma implementação de mesmo conjunto de instruções
   (AMD e Intel, 80{,1,2,3,4,5,6}86)
3. hardware:
   tecnologia de implementação,
   circuitos integrados (CMOS vs NMOS), pipelining vs multi-ciclo

Arquitetura engloba todos os três aspectos

Requisitos a serem atendidos pelo arquiteto I

Área de aplicação: *usos do computador*

desktop desempenho balanceado para diversas tarefas;
sistemas interativos com gráficos, vídeo e áudio;

servidor de computação científica alto desempenho com
operações de ponto flutuante e gráficos;

servidor comercial alto desempenho com bancos de dados e
sistemas transacionais;
confiabilidade, disponibilidade e escalabilidade;

embutido suporte a aplicações específicas;
baixo custo e pequeno consumo de energia

Requisitos a serem atendidos pelo arquiteto II

Compatibilidade de software: *software pré-existente*

nível de linguagem de programação maior flexibilidade;
novo compilador?

compatibilidade nos binários ISA completamente definido;
pouca flexibilidade mas sem investimentos em software novo.

Requisitos a serem atendidos pelo arquiteto III

Suporte a Sistema Operacional:

tamanho do espaço de endereçamento importantíssimo!!
pode limitar aplicabilidade;
$\longrightarrow$ EdE cresce $\geq \mathbf{1/2}$ bit aa

gerenciamento de memória necessário para SOs modernos;
paginado ou segmentado;

proteção usos diferentes por SO e usuários;
proteção à páginas ou a segmentos.

Requisitos a serem atendidos pelo arquiteto IV

Padrões: *requeridos pelo mercado*

ponto flutuante formatos e aritmética: IEEE754;

barramentos dispositivos de E/S: PCI, SCSI;

sistema operacional Unix, Windows, PalmOS;

redes suporte a tecnologias distintas (Ethernet, Infiniband);

linguagens de programação linguagem de alto nível afeta
projeto do conjunto de instruções.

Tecnologias Fundamentais

1. tecnologia de semicondutores: microprocessadores CMOS
2. tecnologia de semicondutores: memória dinâmica (DRAM)
3. tecnologia de armazenamento: discos magnéticos
4. tecnologia de interconexão: redes locais

Tendências da Tecnologia I

Tecnologias de implementação fundamentais
1– Circuitos Integrados para CPUs

núm de transistores	cresce ≈35% aa	4X em 4 anos
tamanho do CI	aprox 10 a 20% aa	lado do retângulo
efeito combinado	núm transistores/CI	cresce ≈55% aa

Ver fabricação de circuitos integrados em `www.intel.com`⟶ `museum` e em P&H-COD

Fotografia de um CI do Pentium IV

H&P QA Fig-1.7

Wafer de 8" com 564 MIPS R20K

Revisão

ARQUITETURA É:

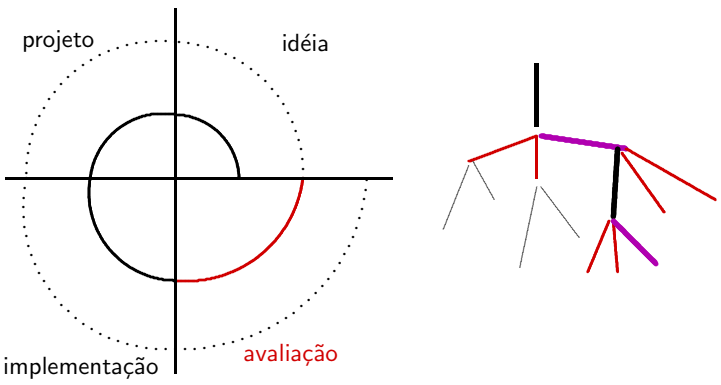
conjunto de instruções + organização + hardware

especif abstrata + *especif refinada* + *concreta*

TENDÊNCIAS DA TECNOLOGIA

	capacidade	velocidade
lógica	2x em 3 anos	2x em 3 anos
DRAM	4x em 3 anos	2x em 10 anos
disco	4x em 3 anos	2x em 10 anos
processador	—	2x em 1.5 anos

Processo de Projetar



Processo iterativo:

idéia $\Longrightarrow$ implementação $\Longrightarrow$ avaliação $\Longrightarrow$ re-projeto ...

avaliação geralmente implica em descartar idéias ruins

Ferramentas para Avaliação de Desempenho

- sistema: programas de teste, rastros (*traces*)
- hardware: custo, atrasos, área, potência
- simulação nos níveis: CdI, *register transfer*, portas, circuitos
- Teoria de Filas
- Regras de projeto
- princípios fundamentais
- Leis (de Moore, de Amdahl, et alli)

Avaliação de Desempenho

QUAL É O MELHOR AVIÃO?

avião	capacidade [p]	alcance [km]	veloc [km/h]	produção [p∗km/h]
B-737	101	1008	957	96657
B-747	470	6640	976	458720
Concorde	132	6400	2160	285120
DC-8	146	13952	870	127020

O que importa são DESEMPENHO e CUSTO

Métricas para Avaliação de Desempenho

aplicação	respostas por hora; operações por segundo
linguagem	produtividade do programador (?)
compilador	qualidade do código gerado
ISA	instruções por segundo (MIPS ou MFLOPS)
circ de dados	megaBytes por segundo
unid. funcionais	ciclos/operações por segundo
portas, fios	ciclos por segundo

Desempenho (i)

$$\text{desempenho}_{\boldsymbol{X}} \triangleq \frac{\mathbf{1}}{\text{tempo exec}_{\boldsymbol{X}}}$$

X é $\mathcal{M}$ vezes mais rápido que Y se
o tempo de execução de Y é $\mathcal{M}$ vezes mais longo que o de X

$$\frac{\text{desempenho}_{\boldsymbol{X}}}{\text{desempenho}_{\boldsymbol{Y}}} = \frac{\text{tempo exec}_{\boldsymbol{Y}}}{\text{tempo exec}_{\boldsymbol{X}}} = \mathcal{M}$$

Desempenho tem unidade de "coisas" por segundo;
maior desempenho é melhor!

Desempenho (ii)

$$\frac{\text{desempenho}_{\boldsymbol{X}}}{\text{desempenho}_{\boldsymbol{Y}}} = \frac{\text{tempo exec}_{\boldsymbol{Y}}}{\text{tempo exec}_{\boldsymbol{X}}} = \mathcal{M}$$

Exemplo:
Máquina A executa programa em 10 segundos;
máquina B executa mesmo programa em 15 segundos.
Quanto A é melhor que B ?

$$\frac{\text{desempenho}_{\boldsymbol{A}}}{\text{desempenho}_{\boldsymbol{B}}} = \frac{\text{tempo exec}_{\boldsymbol{B}}}{\text{tempo exec}_{\boldsymbol{A}}} = \frac{\mathbf{15}}{\mathbf{10}} \Rightarrow \mathbf{50\%}$$

Avaliação de Desempenho

- tempo de resposta, tempo decorrido [s/tarefa]
- tempo de CPU (usuário + sistema) [s]
- ciclo de relógio [ns] frequência do relógio [GHz]
- vazão/produção (*throughput*) [tarefas/s]

desempenho do sistema = tempo decorrido (sem carga)

desempenho de CPU = tempo de CPU dedicado ao usuário

desempenho de pico NUNCA é atingido na prática
desempenho de pico $\gg$ desempenho sustentado

Equação do Desempenho (i)

$$\begin{aligned} \text{tempo de CPU} &= \text{ciclos da CPU} \times \text{ciclo de relógio} \\ &= \frac{\text{ciclos da CPU}}{\text{freq de relógio}} \end{aligned}$$

$$\begin{aligned} \text{ciclos da CPU} &= \text{núm de instr} \times \text{núm de ciclos por instr} \\ \textit{CPI} &\triangleq \textit{ciclos por instrução} \end{aligned}$$

Equação do Desempenho (ii)

$$\begin{aligned} \text{tempo de CPU} &= \text{núm de instr} \times \text{CPI} \times \text{ciclo de relógio} \\ &= \frac{\text{núm de instr} \times \text{CPI}}{\text{freq de relógio}} \end{aligned}$$

esta é a equação mais importante do semestre!

Equação do Desempenho (iii)

Exemplo:
Programa executa na máquina A em 10s com relógio de 100MHz. Queremos máquina B que execute mesmo programa em 6s. Por causa da mudança no projeto da CPU, máquina B vai usar $\mathbf{1.2}$ vezes mais ciclos de relógio que A. Qual o relógio de B ?

$$\begin{aligned} \text{tempo de CPU}_{\boldsymbol{A}} &= \frac{\text{ciclos da CPU}_{\boldsymbol{A}}}{\text{freq de relógio}_{\boldsymbol{A}}} \\ \mathbf{10}\boldsymbol{s} &= \frac{\text{ciclos da CPU}_{\boldsymbol{A}}}{\mathbf{100} \times \mathbf{10^6}} \\ \Rightarrow \quad & \#\text{ciclos da CPU}_{\boldsymbol{A}} = \mathbf{1000} \times \mathbf{10^6} \end{aligned}$$

$$\text{tempo de CPU}_B = \frac{\mathbf{1.2} * \text{ciclos da CPU}_A}{\text{freq de relógio}_B}$$

$$\Rightarrow \text{freq de relógio}_B = \frac{\mathbf{1.2} * \mathbf{1000} \times \mathbf{10^6}}{\mathbf{6}s}$$

$$\Rightarrow \text{freq de relógio}_B = \mathbf{200} \times \mathbf{10^6}$$

$$\text{ganho de } \frac{\mathbf{10}}{\mathbf{6}} = \mathbf{0.67} \Rightarrow \text{freq de relógio } 100\% \text{ maior}$$

Exemplo: Programa executa na máquina A em 10s com relógio de 100MHz. Queremos máquina B que execute mesmo programa em 6s. Por causa da mudança no projeto da CPU, máquina B vai usar $\mathbf{1.2}$ vezes mais ciclos de relógio que A. Qual o relógio de B ?

Equação do Desempenho (iv)

$$\text{tempo de CPU} = \mathcal{N} \times \text{CPI} \times \mathcal{R}$$

núm instr x CPI x período do relógio

$$\text{CPI} = \sum_{j}^{n} \text{CPI}_j \times \text{F}_j$$

onde $\text{F}_j = \mathcal{I}_j / \mathcal{N}$

F_j é a freqüência da instrução $\mathcal{I}$

Equação do Desempenho (v)

$$\text{CPI} = \sum_{j=0}^{n} \text{CPI}_j \times \text{F}_j$$

instr	freq[%]	ciclos	CPI_j
ALU	40	1	.40
load	25	3	.75
store	10	3	.30
desvios	25	2	.50
		total	1.95

Fatores Determinantes

$$\text{tempo de CPU} = \text{núm de instr} \times \text{CPI} \times \text{ciclo de relógio}$$

tempo de CPU	desempenho do sistema
núm de instruções	compilador & processador
CPI	organização e arquitetura da CPU
freqüência de relógio	tecnologia de CIs & arquitetura

Administrativamente falando...

Todos já estão com o livro?

Mestrandos já estão com os artigos?

Ler:

How not to lie with statistics: the correct way to summarize benchmark results, Fleming & Wallace, Comm ACM 29(3), Mar 1986

Characterizing computer performance with a single number, J E Smith, Comm ACM 31(10), Oct 1988

Medidas de desempenho

MIPS = milhões de instruções por segundo

$$\text{MIPS} = \frac{\text{núm de instr}}{\text{tempo decorrido} \times \mathbf{10^6}}$$

$$= \frac{\text{freq de relógio}}{\text{CPI} \times \mathbf{10^6}}$$

Problemas

- independente do conjunto de instruções (RISC/CISC)
- varia para programas na mesma máquina (int×PF)
- pode variar na proporção inversa ao desempenho

Medidas de desempenho

MFLOPS = milhões de instr de ponto flutuante p/s

$$\text{MFLOPS} = \frac{\text{núm de instr de pto flutuante}}{\text{tempo decorrido} \times \mathbf{10^6}}$$

Problemas

- independente do conjunto de instruções (Cray/68882)
- varia para programas na mesma máquina (soma×div)
- média ponderada de custo de instruções:
  soma(a,b) $\propto$ 1 seno(x) $\propto$ 8

Medir desempenho com programas de teste

programas simples: quicksort, números primos
simples de implementar, fora da realidade

programas sintéticos: Dhrystone, Whetstone
simples de implementar, não são código usável

núcleos de programas: SPEC, Livermore Loops
fáceis de medir, não testam sistema de forma realista

programas de verdade: gcc, LaTeX, Spice
mistura deve refletir uso "normal" (browser?)

Medindo desempenho – SPEC

- primeiro conjunto em 1989
  - ⋆ 10 programas produzem um só número: SPECmarks
- segundo conjunto em 1992
  - ⋆ SPECint92 com 6 programas com inteiros
  - ⋆ SPECfp92 com 14 programas com ponto flutuante
- terceiro conjunto em 1995
  - ⋆ SPECint95 com 8 programas com inteiros
  - ⋆ SPECfp95 com 10 programas com ponto flutuante
  - ⋆ conjunto caduca em três anos
  - ⋆ versão base com mesmas flags de compilação (todos programs)
- quarto conjunto em 2000
  - ⋆ CINT2000 com 11 programas com inteiros (C e C++)
  - ⋆ CFP2000 com 14 programas com ponto flutuante (fortran{77,90}, C)

Comparação de resultados

	máq $\mathcal{A}$	máq $\mathcal{B}$
prog 1 [s]	1	10
prog 2 [s]	1000	100
total [s]	1001	110

com programa 1, $\mathcal{A}$ é 10 vezes mais rápido que $\mathcal{B}$
com programa 2, $\mathcal{B}$ é 10 vezes mais rápido que $\mathcal{A}$
erm...

$$\text{Média Aritmética} = \frac{1}{n}\sum_{i=1}^{n} \text{Tempo}_i$$

$\mathcal{B}$ é $1001/110 = 9.1$ vezes mais rápido que $\mathcal{A}$

Resumindo desempenho com um só Número

- média aritmética (ponderada) segue tempo de execução:
  $\sum T_i/n$ ou $\sum(p_i \cdot T_i)$
- média harmônica (ponderada) de taxas (MIPS/MFLOPS) segue tempo de execução:
  $n/\sum(1/R_i)$ ou $n/\sum(p_i \cdot R_i)$
- tempo de execução normalizado é útil para comparações de escalabilidade (máquina-A é X vezes mais rápida que modelo-base)
- para tempo de execução normalizado, deve usar média geométrica:
  $(\prod T_i/N_i)^{1/n}$

Lei de Amdahl (i)

Tempo de execução após melhoria =
(tempo de execução afetado / quanto melhorou)
+ tempo de execução não-afetado
esta também é importante

Exemplo:
programa executa em 100s, multiplicações consomem 80%
do tempo total. Quanto devo melhorar o circuito multiplicador
se quero tempo total em 20s ?

$$20 = 80/n + 20$$

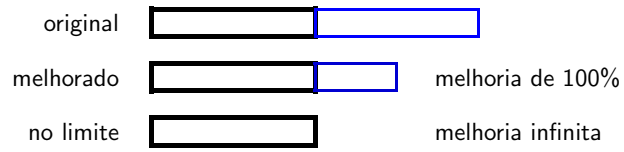
Idem se quero tempo total em 40s ?

$$40 = 80/n + 20$$

Lei de Amdahl (ii)

$$\text{Ganho}_{\text{total}} = \frac{\text{Tempo}_{\text{orig}}}{\text{Tempo}_{\text{melhor}}}$$

$$= \frac{1}{(1 - \text{Frac}_{\text{melhor}}) + (\text{Frac}_{\text{melhor}} \ / \ \text{Ganho}_{\text{melhor}})}$$



Lei de Amdahl (iii)

Exemplo:
programa executa em 100s, multiplicações consomem 80%
do tempo total. Melhorando o circuito multiplicador em 50%,
qual é o ganho total de velocidade ?

$$\text{Ganho}_{\text{total}} = \frac{1}{(1 - 0.80) + \frac{0.80}{1/0.50}}$$

$$= \frac{1}{0.20 + 0.80 * 0.50}$$

$$= \frac{1}{0.60} \approx 67\%$$

Resumo

Equação do Desempenho: **o que vale é o tempo de execução!**

$$\text{tempo de CPU} = \frac{\text{instruções}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instrução}} \times \frac{\text{tempo}}{\text{ciclo}}$$

Lei de Amdahl: **o que interessa é o desempenho global!**

$$\text{Ganho}_{\text{total}} = \frac{1}{(1 - \text{Frac}_{\text{melhor}}) + (\text{Frac}_{\text{melhor}} \ / \ \text{Ganho}_{\text{melhor}})}$$

revisão: Definição de Arquitetura

- Conjunto de Instruções: visão do programador, compilador
- Organização: visão do projetista do computador
- Implementação I: visão do projetista do processador
- Implementação II: visão do projetista dos circuitos integrados

revisão: Equação do Desempenho

$$\text{tempo de CPU} = \frac{\text{instruções}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instrução}} \times \frac{\text{tempo}}{\text{ciclo}}$$

$$|\text{código}| \times \text{CPI} \times |\text{ciclo}|$$

- tamanho do código – depende de algoritmo e compilador
- CPI – depende do Conj de Instruções e organização da CPU
- ciclo – depende da organização e implementação da CPU
- objetivo: **minimizar tempo e não termos isolados**

Arquitetura ao longo das décadas

DÉCADA	ARQUITETO ENVOLVE-SE COM
1960	projetar circuitos aritméticos eficientes
1970	projeto de conjunto de instruções [1]
1980	RISC vs CISC, super-escalaridade, compiladores [2]
1990	espaço de endereçamento dobra (32 $\longrightarrow$ 64 bits)
	otimização de desvios através de execução condicional
	suporte a multi-mídia; operações de ponto flutuante
2000	instruções largas (VLIW)
	maior suporte a especulação (execução condicional)
	emulação de 80x86 (Transmeta)

[1] Maior complexidade para aumentar produtividade dos programadores

[2] *If a compiler cannot generate it, desktop and server programs generally won't use it.* H&P QA pg 91

Conjunto de Instruções

Recorte:

- operações
- operandos
- endereçamento de operandos
- codificação
- implementação

Instruction Set Architecture ⟶ ISA
= parte visível ao programador e ao compilador ↔ interface

Conjunto de Instruções ≡ interface hw-sw

CDI especifica a **funcionalidade** do processador

- quais operações ele suporta
- quais mecanismos de armazenamento suporta e sua utilização
- como programador/compilador define programas a executar

Estudo de CDIs é a parte de arquitetura deste curso;
restante é micro-arquitetura

Conjunto de Instruções ≡ interface hw-sw

Interface bem projetada:

- sobrevive a muitas implementações (portabilidade, compatibilidade)
- é usada de muitas maneiras (generalidade)
- provê funcionalidade conveniente aos níveis acima
- permite implementação eficiente nos níveis abaixo

Receita para um "bom" Conjunto de Instruções

implementabilidade

permite implementações numa faixa de preço/desempenho

DEVE permitir implementações de alto desempenho

programabilidade

deve facilitar expressão de programas pelo programador/compilador

compatibilidade para o futuro e com o passado

implementabilidade e programabilidade através de gerações:

deve garantir que todo sw existente execute

x86: 8086,286,386,486,Pentium{,Pro,II,III,4}

Estas são as 3 "-idades" de um CdI

ISA - operações

tipo	exemplo
aritmética e lógica	add, sub, and, sll
transferência de dados	move, load
controle	branch, jump, call
sistema	syscall, trap
ponto flutuante	fadd, fmul, fdiv, sqrt
decimal	addd, convert
string	move, comp
multimidia 2D, 3D	MMX/SSE

ISA - operações

10 instruções do x86 mais populares em 5 programas SPECint95:

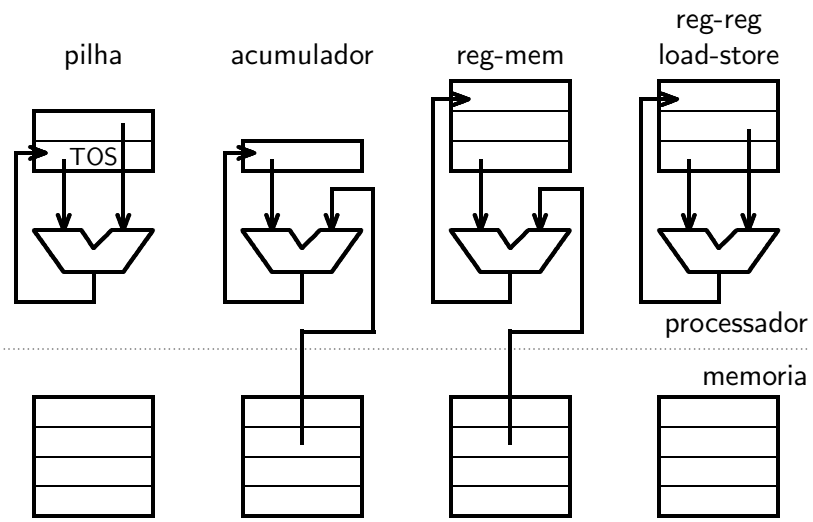
ord	instrução	% total	
1	load	22	
2	desv cond	20	**1/5**
3	compare	16	
4	store	12	ld+st $\longrightarrow$ **1/3**
5	add	8	
6	and	6	
7	sub	5	
8	move reg-reg	4	por que?
9,10	call+ret	1+1	
	total	96	

Amdahl legisla: estas **devem** ser otimizadas

ISA - operandos

- Localização dos operandos
  * no processador
  * em memória
- Quantos bits são necessários para identificar um operando?
  * no processador
  * em memória
- Quanto tempo é necessário para acessar um operando?
  * no processador
  * em memória

ISA - operandos no processador



ISA - operandos no processador

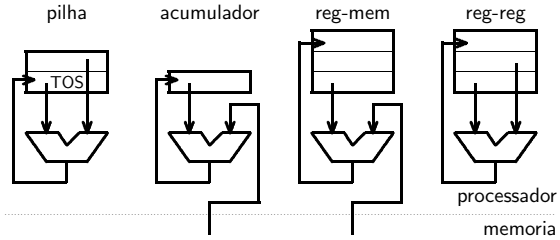
Operandos no processador: acesso rápido & identificação compacta

- Acumulador
  * código compacto (endereço implícito)
  * hardware simples
  * tráfego elevado de/para memória
- Pilha
  * código compacto (endereço do topo da pilha implícito)
- Registradores
  * registradores devem ser identificados
  * espaço de nomes adicional (variáveis em memória e registradores)
  * menos acessos à memória
  * maior velocidade porque regs. são mais rápidos que memória

ISA - operandos no processador

código para computar `C = A + B`

PILHA	ACUMUL	REG-MEM	REG-REG
push A	load A	load R1,A	load R1,A
push B	add B	add R3,R1,B	load R2,B
add	store C	store R3,C	add R3,R1,R2
pop C			store R3,C



ISA - operandos para operações de ULA

- Instruções da ULA combinam operandos
- número de operandos é implícito:
  - dois: `r1` ← `r1 OP r2`
    código denso
    destrutivo porque sobre-escreve destino
    assimétrico (A–B, A/B)
  - três: `r1` ← `r2 OP r3`
- operandos em registradores ou em memória
  - registrador + memória (IBM, x86) NÃO ORTOGONAL
  - reg + reg (RISCs, Cray) ortogonal com load/store
  - qualquer combinação (VAX) ortogonal,
    instr com tamanho variável
  - .

ISA - de volta a Lilliput

Ordem dos bytes nas palavras

- big endian: byte com endereço xxxx00_2 na posição mais significativa de uma palavra de 32 bits Motorola (powerPC), IBM (r6000), SPARC

end byte	MS			ms
0	0	1	2	3
4	4	5	6	7

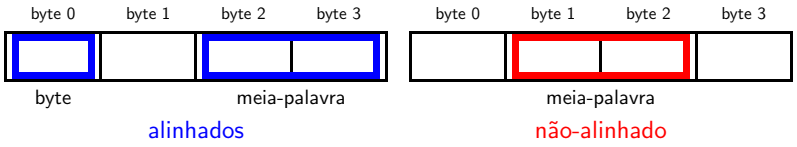
- little endian: byte com endereço xxxx00_2 na posição menos significativa de uma palavra de 32 bits DEC (alpha), Intel (x86)

end byte	MS			ms
0	3	2	1	0
4	7	6	5	4

- ambos – MIPS (e powerPC)

ISA - alinhamento de operandos

- Palavras tem "tamanho natural":
  byte, half-word, word, doubleword
  float, double, quadword
- Operando alinhado pelo tamanho natural se
  endereço MOD tamanho = 0
  - ⋆ palavra alinhada: ender mod 4 = 0 — 1024, 1028, 102c, 1030,...
    mod 4 porque palavra tem tamanho de 4 bytes
  - ⋆ palavra desalinhada ender mod 4 ≠ 0 — 1025 ou 1026 ou 1027



ISA - alinhamento de operandos

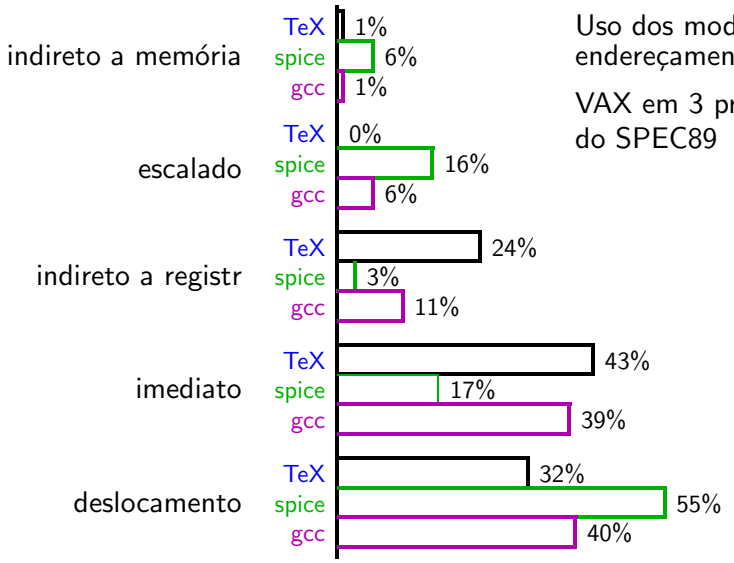
- sem restrições
  - ⋆ software mais simples
  - ⋆ hardware deve detectar e fazer ≥ 2 referências à memória
  - ⋆ lógica complexa; tempo extra em todas as instruções (pior caso)
  - ⋆ dificulta adiantamento de loads no pipeline
- com restrições
  - ⋆ software deve garantir alinhamento
  - ⋆ hardware verifica e provoca excessão se for o caso
  - ⋆ adiantamento de loads mais simples
- mais ou menos
  - ⋆ mais de uma instrução para acessos desalinhados (load-half)
  - ⋆ compilador ajuda no alinhamento
  - ⋆ hardware verifica e provoca excessão se for o caso

ISA - modos de endereçamento

MODO	ENDER EFETIVO	EXEMPLO
a registrador	R	add r4,r3
imediato	imed	add r4,#8
deslocamento	M[R+imed]	add r4,100(r1)
indireto a registrador	M[R]	add r4,(r1)
indexado	M[R_i+R_j]	add r3,(r1+r2)
absoluto	M[imed]	add r1,(1001)
indireto a memória	M[M[R]]	add r1,@(r3)
auto-incremento	M[R]; R += d	add r1,(r2)+
auto-decremento	R −= d; M[R]	add r1,–(r2)
escalado	M[R_i + R_j*d]	add r1,100(r2)[r3]

Apenas absoluto e indireto a registrador são fundamentais;
outros podem ser derivados destes

ISA - modos de endereçamento

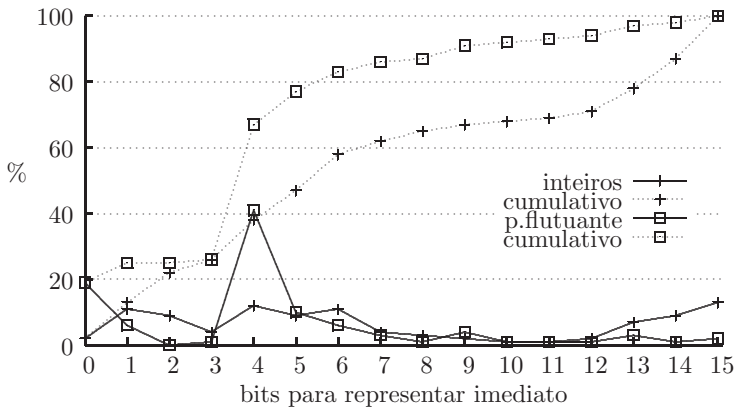


ISA - modos de endereçamento

- Distância dos deslocamentos (LD e ST)
  - 16 bits cobre 99% dos casos (dist $\leq \pm$ 32K bytes/pals)
    Alpha (SPEC CINT2000 e CFP2000)
- Uso de imediatos (tamanho das constantes)
  - 1/5 a 1/4 das instruções de ULA e movimentação de dados usam constantes
  - $\approx$ 90% das constantes são representadas em $\leq$16 bits
    instruções suportam constantes de 16bits
    (como representa maiores?)
  - 20 a 30% das constantes são negativas

ISA - modos de endereçamento

Uso de imediatos: Alpha SPEC CINT2000 e CFP2000
1/5-1/4 das instr de ULA e de movim de dados usam constantes



Administrativamente falando...

- Avaliação:
  - BCC
    - ⋆ 2 provas 35% + 35%
    - ⋆ trabalho 20%
    - ⋆ participação nas aulas de exercícios 10%
  - Mestrado
    - ⋆ 2 provas 30% + 30%
    - ⋆ trabalho 20% (entrega na semana das finais)
    - ⋆ participação nas aulas de exercícios 20%
- 5 aulas de exercícios
  - ⋆ listas distribuídas na aula

Administrativamente falando...

Todos devem ler o artigo sobre o Cray-1 para a próxima aula
The CRAY-1 Computer System, R M Russel, CACM 21(1), Jan 78

instruções vetoriais:

- `addv v1,v2,v3 # for (i=0;i<64;i++) v1[i]=v2[i]+v3[i];`
- encadeamento *(chaining)*:

```
# for (i=0;i<64;i++) R[i] = N[i]+ 3.1415*M[i];
chain:  ldv v1,endM
        ldv v6,endN
        muliv v1, 3.1415
        addv v5,v1,v6
        stv v5,endR
```

"saída" do ldv passa direto para muliv, para addv e para stv

ISA - controle de fluxo de execução

- Cada instrução é auto-contida (≠ `if-then-else, do-until`)
- Seqüência de execução de instruções é determinada pelo PC
  - execução seqüencial: PC++
  - instruções mudam conteúdo do PC explicitamente: PC ← ENDER
- Aspectos importantes:
  - desvios condicionais (toma desvio ou segue seqüencialmente)
  - determinação do endereço de destino
  - ligação para endereço de retorno de funções
  - salvamento de estado
  - interface com sistema operacional

ISA - controle de fluxo de execução

Tipos de instruções de controle:

- desvios condicionais (desvia ou prossegue) `branch`
- saltos incondicionais `jump`
- chamada de função `call`
  - geralmente incondicionais
  - necessita ligação para endereço de retorno
  - pode salvar estado (registradores)
- retorno de função `ret`
  - geralmente incondicionais
  - retorno indireto através do endereço de ligação
  - pode salvar estado (registradores)

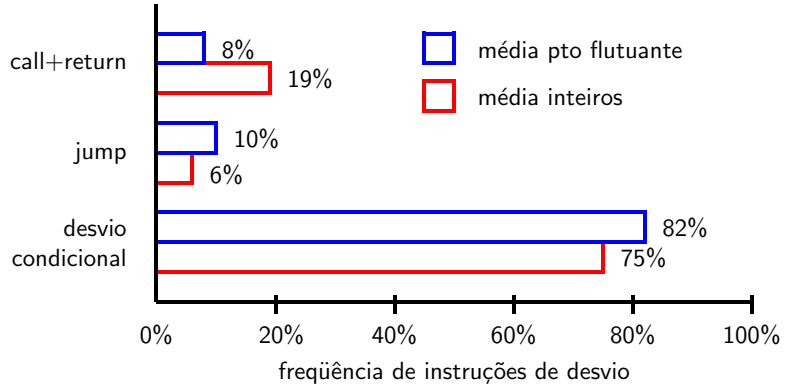
ISA - controle de fluxo de execução

Tipos de instruções de controle:

- chamadas de sistema `syscall, trap`
  - semelhante à chamada de função
  - endereço de destino definido no opcode
  - atravessa fronteira de proteção
- retorno de chamada de função `reti`
  - semelhante ao retorno de função
  - opcode diferente da chamada
  - atravessa fronteira de proteção

ISA - controle de fluxo de execução

Tipos de desvios: Alpha SPEC CINT2000 e CFP2000



ISA - controle de fluxo de execução

Tipos de mudança no fluxo:

- condicional ou incondicional
- salva conteúdo do PC
- o endereço de destino
  - endereço fixo (imediato ou PC+imediato)
  - definido em tempo de execução (registrador)

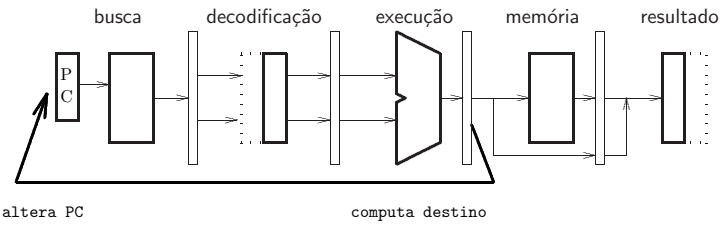
ISA - controle de fluxo de execução

- Mecanismos de decisão de desvios:
  - registrador de status (condition code)
    - `PC ⇐ (R1>R2 ? PC+desloc : PC)`
    - operação de ALU no caminho crítico da resolução do desvio
  - registrador de condição (CC = condition code register)
    - `R1 ⇐ (R2 - R3) mudaCC`
    - `PC ⇐ (CC ? PC+desloc : PC)`
    - necessita estado adicional (CC)
    - dificulta reordenação de código (pelo efeito colateral)
  - registrador de uso geral
    - `R1 ⇐ (R2 > R3)`
    - `PC ⇐ (R1>0 ? PC+desloc : PC)`
    - pouco estado adicional
    - usa registrador inteiro para poucos bits

ISA - controle de fluxo de execução

Mecanismos para especificar endereço de destino:

- endereço arbitrário como load/store
  - `PC ⇐ PC + (R1+desloc)`
  - versátil e ortogonal
  - mais demorado para decodificar
  - executa desvio e computa destino em estágios separados do pipeline (muda PC na busca e computa ender em exec)



ISA - controle de fluxo de execução

Mecanismos para especificar endereço de destino:

- relativo ao PC (deslocamento em imediato)
  `PC` $\Leftarrow$ `(PC+desloc)`
  código independente de posição facilita ligação
  destino pode ser computado no estágio de busca/decodificação
  imediato com 12 bits cobre 99% dos casos (MIPS)
  . (dist $\leq \pm$ 2K pals)
  imediato com 16 bits cobre 99,5% dos casos (ALPHA)
  . (dist $\leq \pm$ 32K pals)
  MAS endereço de destino não pode ser muito distante

ISA - controle de fluxo de execução

Mecanismos para especificar endereço de destino:

- indireto a registrador (endereço em registrador)
  `PC` $\Leftarrow$ `R1`
  código compacto
  destino a qualquer distância
  destino pode ser computado dinamicamente (retorno de função)
  instrução adicional para carregar registrador
  muda PC e "calcula" destino em estágios diferentes
- endereço implícito no opcode (vetor de syscalls)
  necessário para interface com sistema operacional

ISA - controle de fluxo de execução

Mecanismos para especificar endereço de ligação/retorno:

- necessário para retorno de função e chamada de sistema
- registrador implícito (R31 no MIPS)
  registrador deve ser salvo antes da próxima chamada
- registrador explícito
  pouquinho mais de flexibilidade
- pilha de chamada de funções SE pilha é parte da arquitetura
  complexidade adicional:
  instruções tipo `push` e `pop`
  hardware deve detectar over/under-flow na pilha

ISA - controle de fluxo de execução

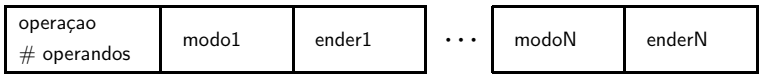
Salvar e recompor estado nos saltos/desvios

- que estado salvar
  - chamadas de função $\Longrightarrow$ registradores (CISCs)
  - chamadas de sistema $\Longrightarrow$ registradores, PC, flags, PSW
- ISA/hardware não necessita salvar registradores
  - função que chama salva regs em uso — caller save
  - função chamada salva regs que vai usar — callee save
- ISA/hardware pode salvar registradores
  - VAX call
  - pressupõe convenção de chamada/retorno pelo compilador
- processadores recentes não fazem salvamento

  mas SPARC tem janelas de registradores

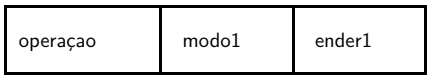
ISA - codificação das instruções

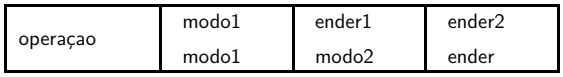
Tipos de codificação: comprimento variável, fixo e híbrido



Variavel: VAX, Intel x86

Fixo: Alpha, ARM, MIPS, PowerPC, SPARC





Hibrido: IBM 360/370, MIPS16, TI TMS5320c54x

ISA - resumo

Instruction Set Architecture = ISA
= parte visível ao programador e ao compilador

- operações — popularidade vs implementação
- operandos — popularidade vs implementação
- endereçamento de operandos — código compacto vs flexibilidade
- codificação — código compacto vs decodificação
  facilidade de implementação em pipeline

revisão: Desempenho

Equação do desempenho: **o que interessa é o tempo total!**

$$\text{tempo de CPU} = \frac{\text{instruções}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instrução}} \times \frac{\text{tempo}}{\text{ciclo}}$$

$$|\text{código}| \times \text{CPI} \times |\text{ciclo}|$$

Lei de Amdahl: **o que interessa é o desempenho global!**

$$\text{Ganho}_{\text{total}} = \frac{1}{(1 - \text{Frac}_{\text{melhor}}) + (\text{Frac}_{\text{melhor}} \;/\; \text{Ganho}_{\text{melhor}})}$$

revisão: Conjunto de Instruções

- Instruction Set Architecture = ISA
  = parte visível ao programador e ao compilador
- contrato entre hardware e software
- simplifica compilador se
  - ∗ ortogonal
  - ∗ regular
  - ∗ facilitar composições
- codificação simples e regular simplifica hardware
  - ⋆ operações — popularidade vs implementação
  - ⋆ operandos — popularidade vs implementação
  - ⋆ endereçamento de operandos — código compacto vs flexibilidade
  - ⋆ codificação — código compacto vs decodificação; facilidade de implementação em pipeline

Compiladores 0

Função do compilador:

- todos os programas corretos executam corretamente
- maioria dos programas compilados executa rapidamente
- compilação rápida
- suporte a depuração

bloco básico = trecho de código entre desvios

Compiladores 0

estágio	depende de	função
front end	linguagem	produz representação intermediária
otimizador de alto nível	linguagem (pouco)	transformação de loops integração de funções
otimizador global	máquina (pouco)	otimização local e global alocação de registradores
gerador de código	máquina (muito)	seleção de instruções otimização dependente de máquina

Compiladores 0 - estilos de transformação

- otimização no alto nível
  - efetuada na linguagem de alto nível
    para uso pelos outros estágios
- otimização local
  - otimização somente em blocos básicos
- otimização global
  - otimização que atravessa blocos básicos
  - transformação de loops
- alocação de registradores
  - associa registradores com operandos
- otimização dependente do processador
  - depende da arquitetura

Compiladores 0 - operações nos estágios

- tradução para representação intermediária
- expansão de funções — inlining
- otimização de loops — desenrolar; código invariante
- eliminação da variável de indução — cálculo de índices
- eliminação de sub-expressões comuns — cálculos repetidos
- otimizações de saltos
- propagação de constantes — variável que é constante
- alocação de registradores
- strength reduction — mult vs soma+desloca
- escalonamento do pipeline — reordenação; *branch delay slots*
- tradução para linguagem de máquina

Compiladores 0 - do que eles gostam?

- quem escreve um compilador deseja
  - regularidade — simplifica análise de casos
  - ortogonalidade — suporta todas as combinações
  - composabilidade — primitivas ao invés de soluções
  - as três permitem operações simples combinadas em operações complexas
- compiladores efetuam análise de casos gigantesca
  - opções demais dificultam escolhas
- conjuntos de instruções ortogonais quanto a
  - operações tipos de dados modos de endereçamento
  - completude
    uma, ou — *condições de desvio* $\longrightarrow$ *eq lt*
    todas as soluções, — *cond de desvio* $\longrightarrow$ *eq ne lt gt le ge*
    mas não só algumas escolhas idiossincráticas

Conjunto de Instruções do MIPS

Projeto de RISCs na década de 80 para obter
implementação de pipelines num único CI

Reduced Instruction Set Computers
reduced == simples, e não pequeno

- ênfase em
  - decodificação rápida
  - instruções com tamanho fixo
  - codificação regular
- compilador poderia escalonar instruções para execução
- código grande/esparso

MIPS64

- endereços alinhados de 32 bits
- modo de endereçamento é *deslocamento* — load/store
- tipos de dados simples: byte half word doubleword, single double
- registradores
  - 32 regs de uso geral, de 64 bits (R0 = 0) — regs
  - 32 regs de ponto flutuante de 64 bits — regsPF
  - registrador de status para ponto flutuante
  - sem registrador de status para inteiros — ($\nexists$CondCodeReg)
- três formatos de instrução com mesmo tamanho

MIPS - modos de endereçamento

MODO	ENDER EFETIVO	EXEMPLO
a registrador	R	add r4,r3,r2
imediato	imed	add r4,#8
deslocamento	M[R+imed]	add r4,100(r1)
indireto a registrador	[R]	jr r4
absoluto	ender	j ender

MIPS

- transferência de dados
  - load/store byte/half/word/doubleword
  - load/store PF single/double
  - move de-para regs e regsPF
- ULA
  - add/sub/mult/div
  - and/or/xor
  - sll/srl (lógicos), sra (aritmético) deslocamentos
  - loadHigh (usado para constantes de 32 bits)

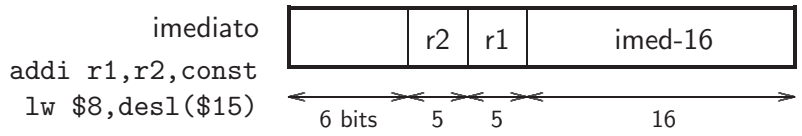
MIPS

- ponto flutuante
  - add/sub/mult/div single/double
  - conversões de-para inteiros
  - desvios (liga/desliga bits para desvios)
- controle
  - desvios condicionais: $=0$ $\neq 0$ bits_PF
  - jump/jr jump-register
  - jal jump-and-link-register
  - trap/rte return from exception

MIPS

Formatos das instruções

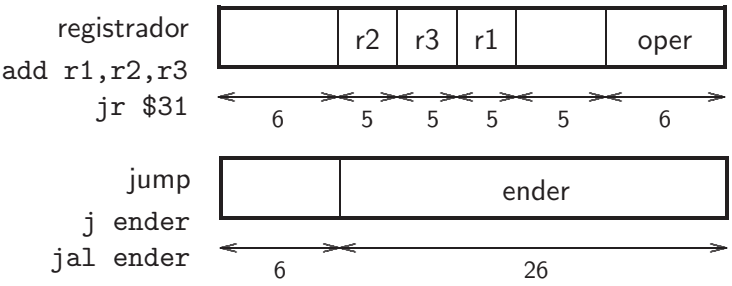
- formato-I
  - instruções de ALU com imediatos
  - load e store
  - desvios condicionais
  - jump-register



MIPS

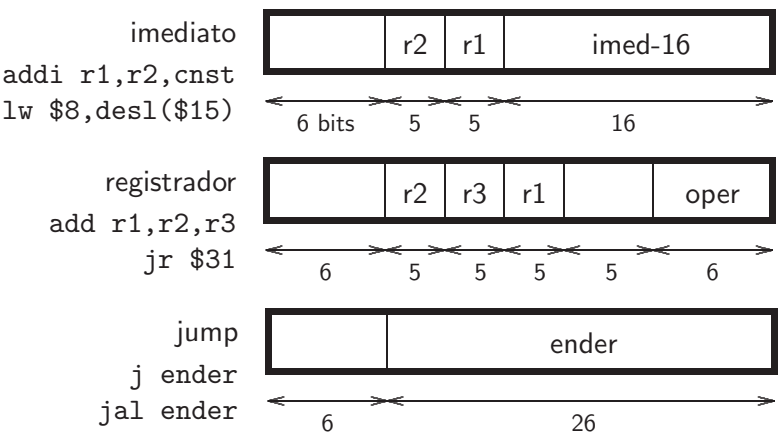
Formatos das instruções

- formato-R
  - instruções de ALU com três operandos
- formato-J
  - saltos incondicionais



MIPS

- instruções regulares facilitam decodificação ⟶ **hw rápido**
- instruções simples facilitam construção do compilador e geração de código ⟶ **sw rápido**



Administrativamente falando...

ou

filmes imperdíveis para computeiros:

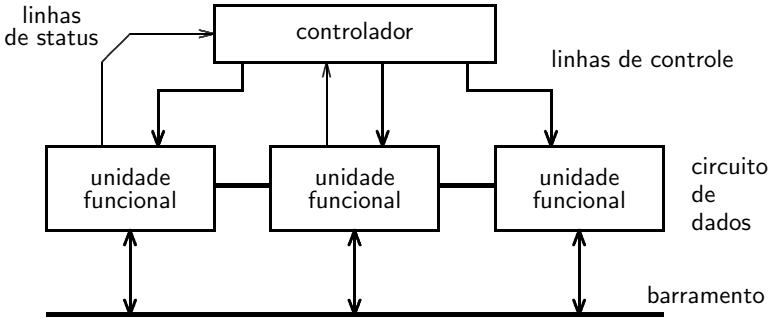
Enigma ⟶ ponto de vista dos europeus que quebraram o código
U571 ⟶ marinheiros/cowboys capturam máquina de codificação

assistam aos dois

Quem conseguir, leia *The enigma of inteligence* que é a biografia de Alan Turing, tido como o pai da IA além de muitas outras coisas

MIPS - microarquitetura

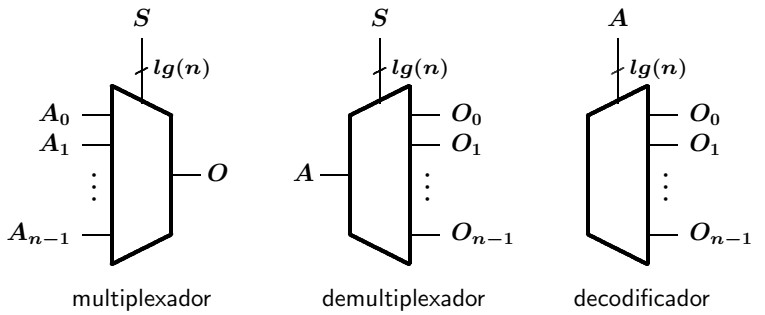
Microarquitetura = implementação do conjunto de instruções

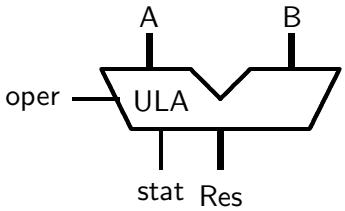


Estrutura ↔como componentes são interligados
Seqüenciamento ↔movimento de dados entre componentes

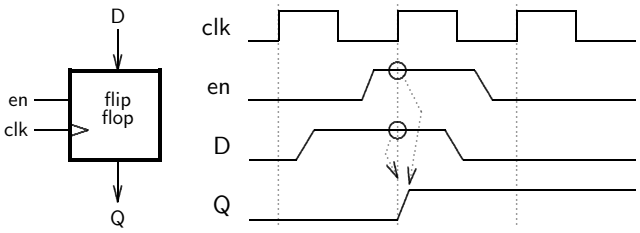
estrutura = *datapath*	estática
seqüenciador = controle	dinâmico

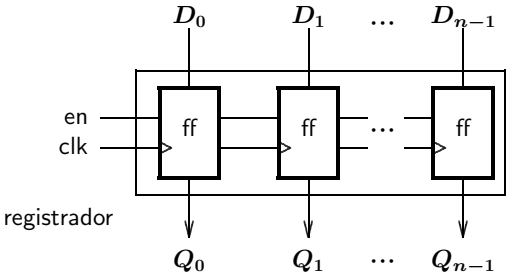
MIPS – circuitos combinacionais



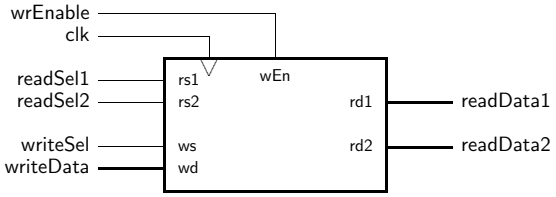


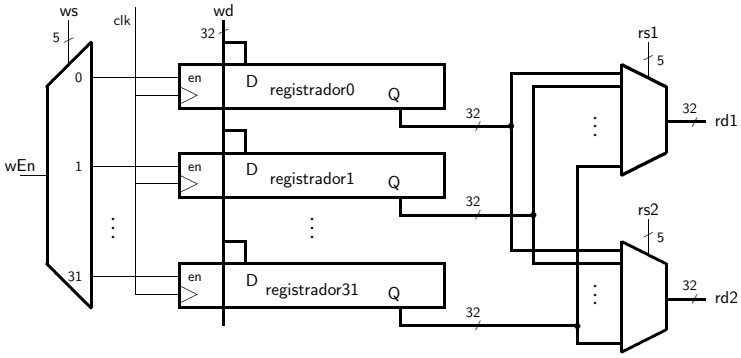
MIPS – circuitos seqüenciais



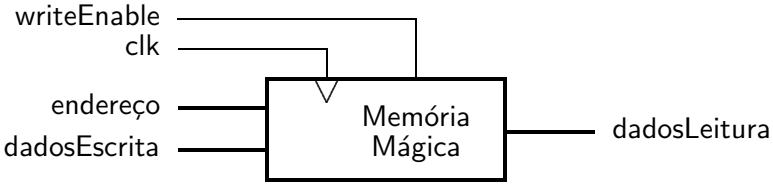


MIPS – bloco de registradores





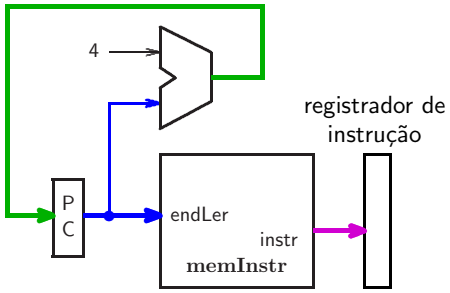
MIPS – memória idealizada



Modelo simplificado da memória

- leituras e escritas completam em 1 ciclo
  - ∗ leitura pode ocorrer a qualquer instante (combinacional)
  - ∗ escrita efetuada na borda ascendente do relógio
    escrita habilitada
    ⟶ dados e endereços estáveis na borda do relógio
- modelo mais realista será discutido adiante

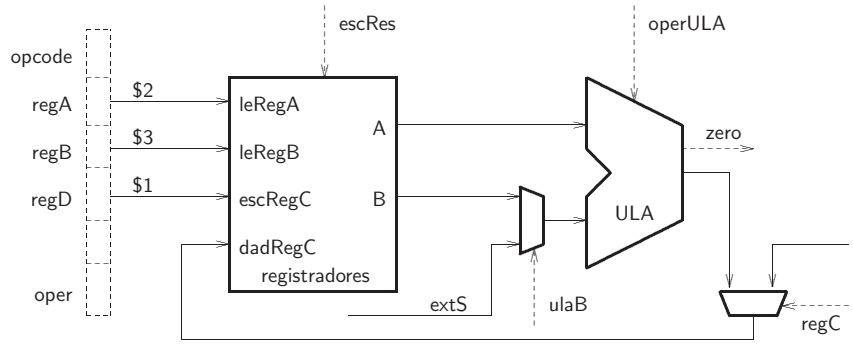
MIPS - busca de instruções



- conteúdo do PC indexa memória
- ao final do ciclo
  - nova instrução no Registrador de Instrução
  - PC é incrementado

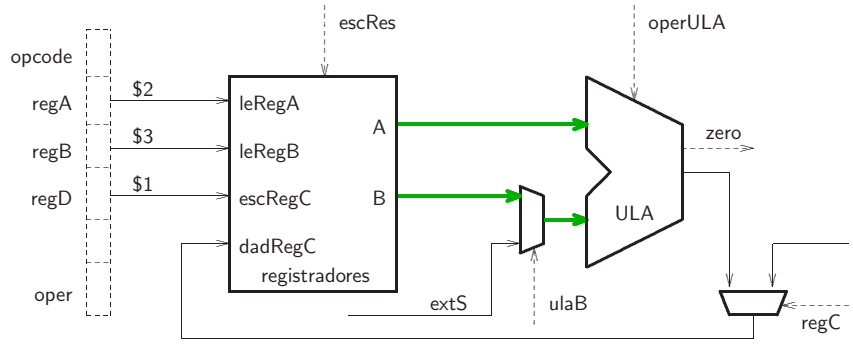
MIPS - instruções de ALU (i)

```
add $1, $2, $3 # $1 <- $2 + $3
```



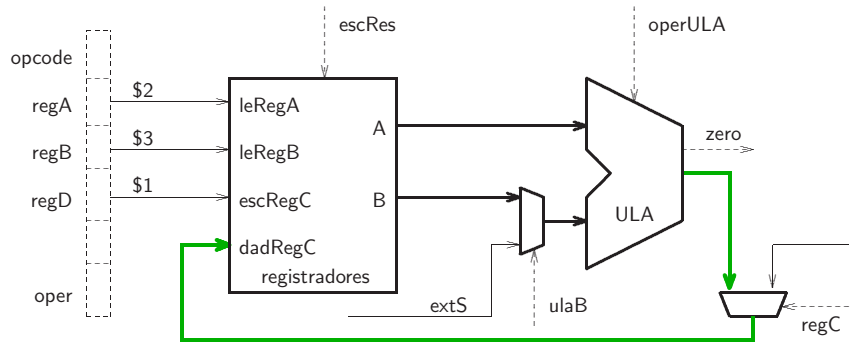
MIPS - instruções de ALU (ii)

```
add $1, $2, $3 # $1 <- $2 + $3
```



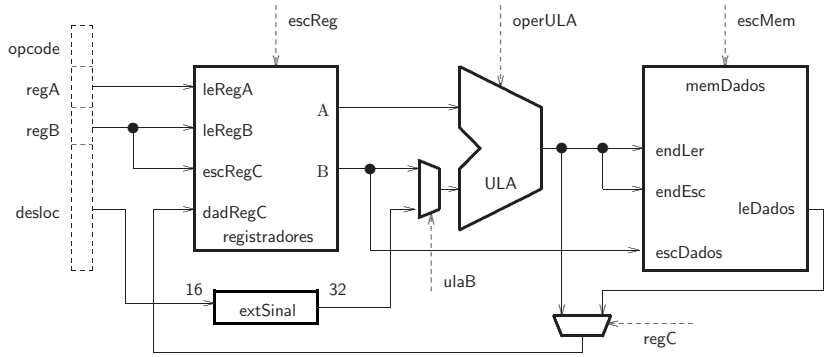
MIPS - instruções de ALU (iii)

```
add $1, $2, $3 # $1 <- $2 + $3
```



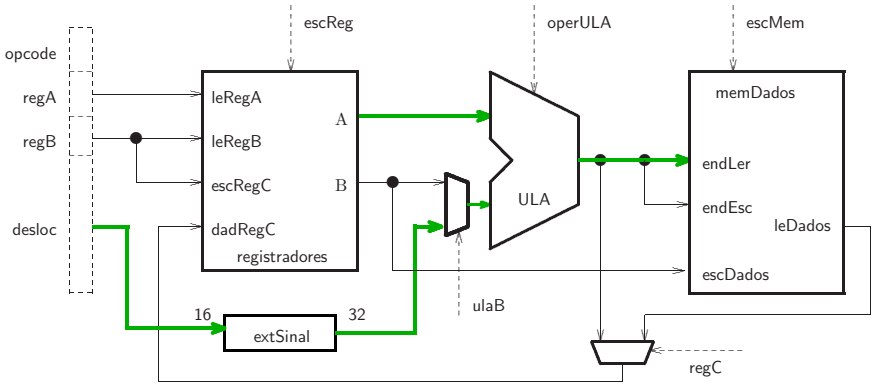
MIPS - acesso à memória (i)

```
lw $8, desloc($15) # $8 <- M[desloc + $15]
sw $8, desloc($15) # M[desloc + $15] <- $8
```



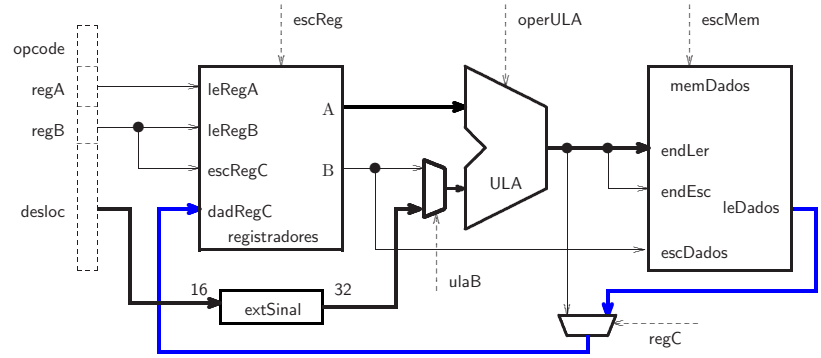
MIPS - acesso à memória (ii)

```
lw $8, desloc($15) # $8 <- M[desloc + $15]
```



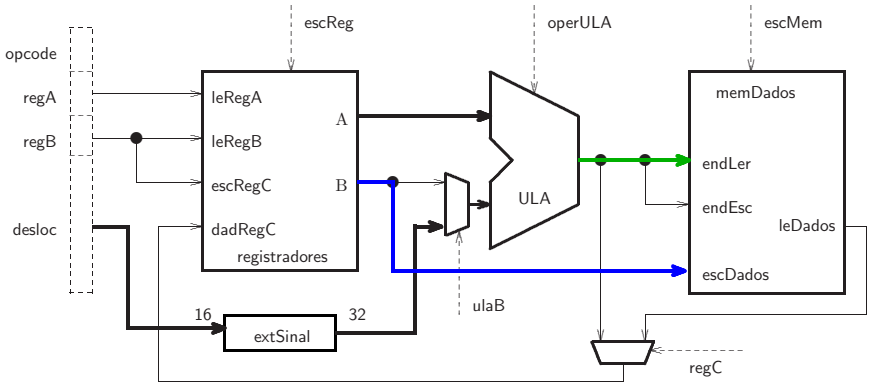
MIPS - acesso à memória (iii)

```
lw $8, desloc($15) # $8 <- M[desloc + $15]
```

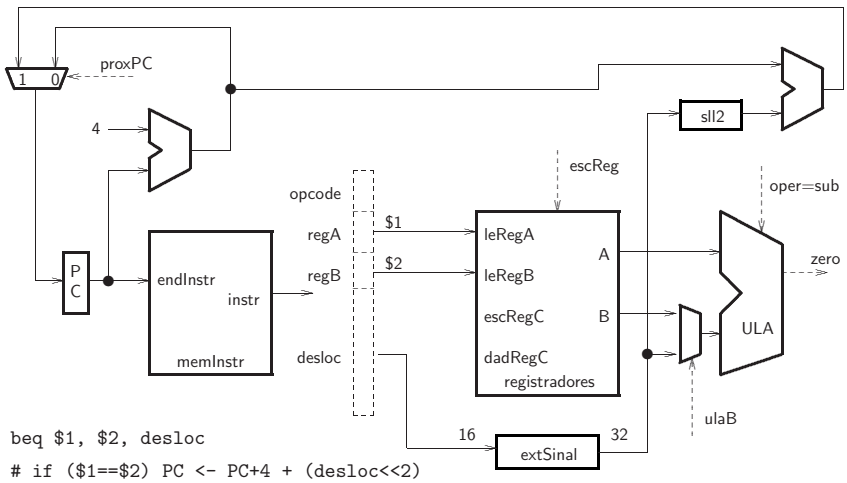


MIPS - acesso à memória (iv)

```
sw $8, desloc($15) # M[desloc + $15] <- $8
```

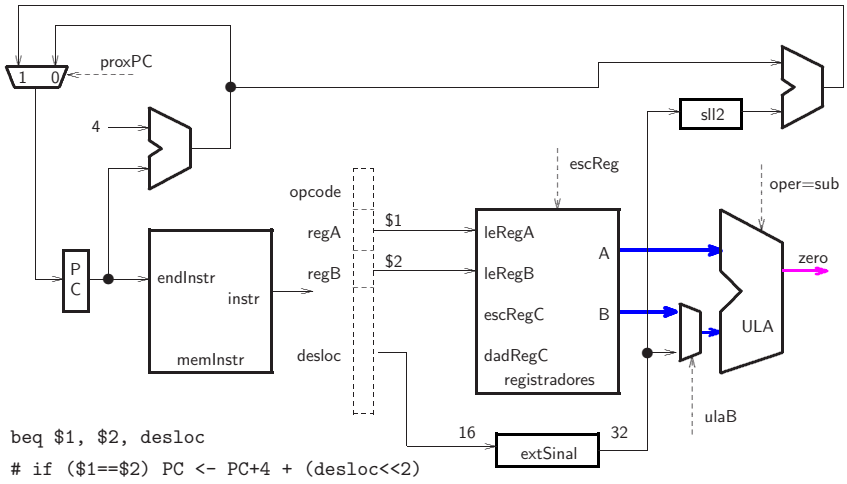


MIPS - desvios (i)



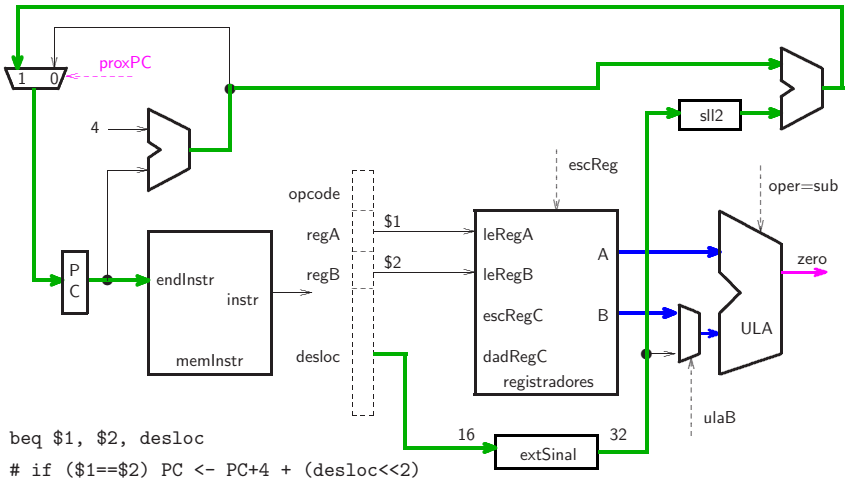

```
beq $1, $2, desloc
# if ($1==$2) PC <- PC+4 + (desloc<<2)
```

MIPS - desvios (ii)



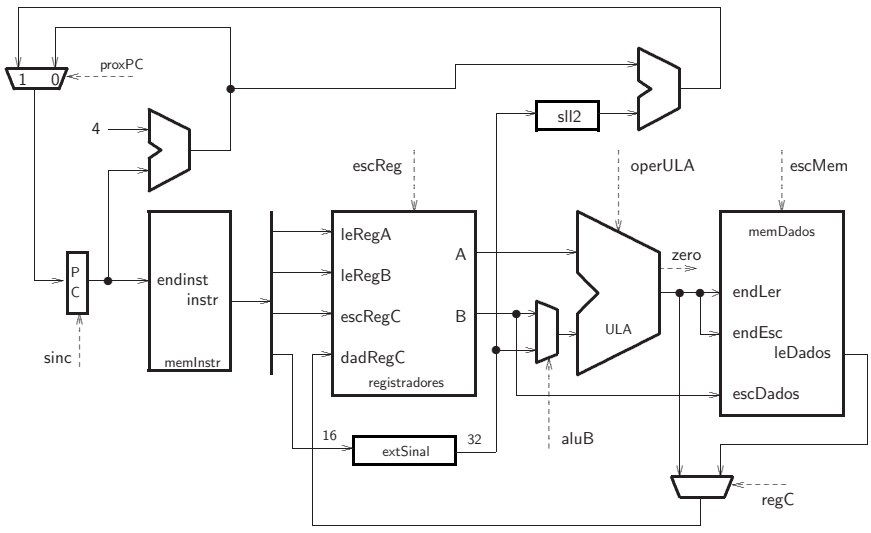

```
beq $1, $2, desloc
# if ($1==$2) PC <- PC+4 + (desloc<<2)
```


MIPS - desvios (iii)




```
beq $1, $2, desloc
# if ($1==$2) PC <- PC+4 + (desloc<<2)
```


MIPS - CPU completa



MIPS - exercícios

Desenhe os diagramas completos do processador para executar as instruções:

- jump
- jump-and-link
- jump-register
- addi (add imediato)

revisão: compiladores

- quem escreve um compilador deseja
  - regularidade
  - ortogonalidade
  - composabilidade
- conjuntos de instruções ortogonais quanto a
  - operações
  - tipos de dados
  - modos de endereçamento
  - completude

    uma, ou *condições de desvio* $\Longrightarrow$ *eq lt*

    todas as soluções, *condições de desvio* $\Longrightarrow$ *eq ne lt gt le ge*

    mas não algumas

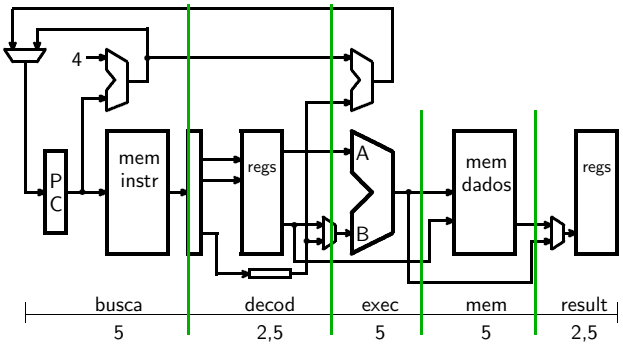
Modelo seqüencial

Conjunto de instruções de processadores "comuns" define um **modelo seqüencial de execução:**

cada instrução é completamente executada
e altera o estado do processador
antes do início da próxima instrução

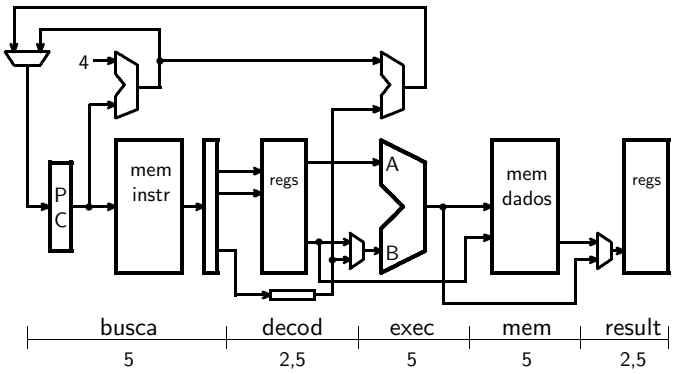
Este modelo facilita muito a vida do programador!

Implementação mono-ciclo



- ⋆ **busca** – busca instrução que será executada
- ⋆ **decodificação** – decodifica instrução e acessa conteúdo dos registradores
- ⋆ **execução** – executa operação de ULA ou cálculo de endereço
- ⋆ **memória** – acesso à memória de dados
- ⋆ **resultado** – armazena resultado no bloco de registradores

Implementação multi-ciclo



Instrução completa em **alguns** ciclos de relógio: **CPI>1**

$$\begin{aligned} T_{\text{ciclo}} &\geq \max(T_{\text{busca}}, T_{\text{decod}}, T_{\text{ex}}, T_{\text{mem}}, T_{\text{res}}) \\ &\geq 5 \text{ unidades de tempo} \end{aligned}$$

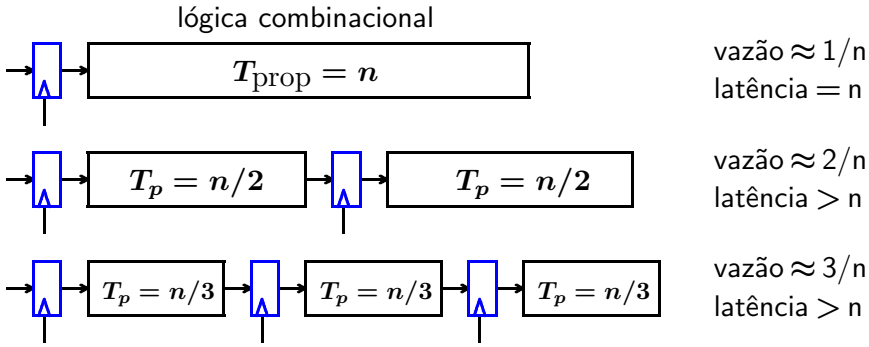
Implementação multi-ciclo

instr ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13
ld r1,200(r8)	B	D	E	M	R								
add r2,r1,r6						B	D	E	R				
st r2,100(r9)										B	D	E	M

$\text{CPI} = \sum_{j=0}^{n} \text{CPI}_j \times \text{F}_j$

instr	freq[%]	ciclos
ALU	40	4
load	25	5
store	10	4
desvios	25	4
	CPI	4.25

Segmentação



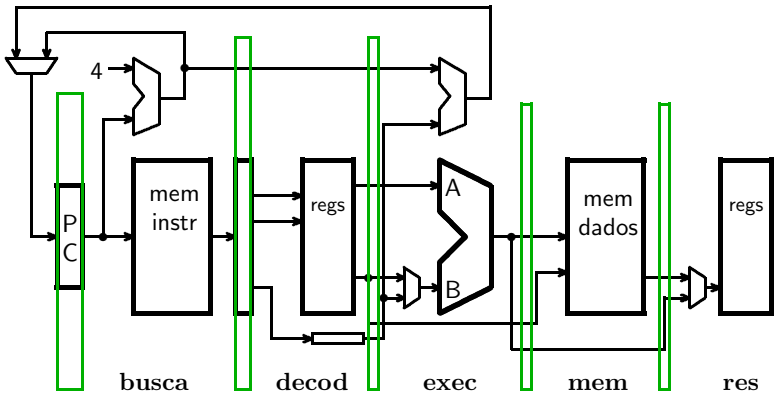
vazão $\approx 1/n$
latência $= n$

vazão $\approx 2/n$
latência $> n$

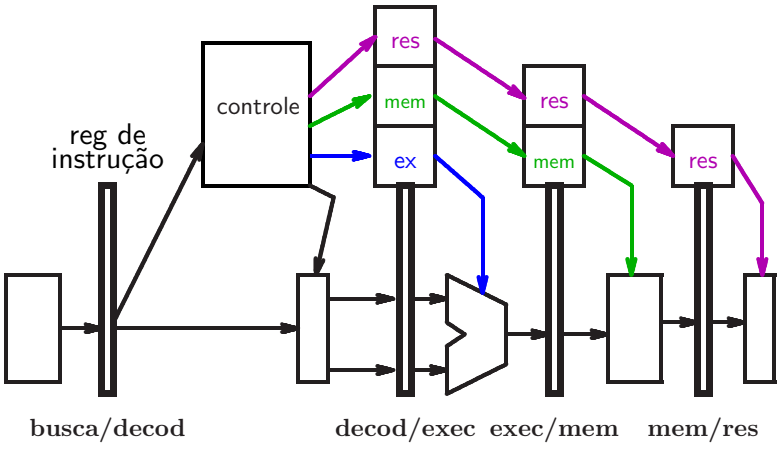
vazão $\approx 3/n$
latência $> n$

- **Segmentação:** adicionar registradores entre unidades funcionais
- **vazão** é proporcional ao número de estágios
- **latência** aumenta por causa dos registradores

Implementação com segmentos



Implementação com segmentos - controle



Segmentação

- Circuito de dados
  - componentes diversos com tempos de propagação distintos
  - instruções usam mais componentes que os disponíveis
    $\Longrightarrow$ **riscos estruturais** structural hazards
- dependências de dados entre instruções nos diversos estágios
  - $\exists$ dependências entre instruções nos estágios
    $\Longrightarrow$ **dependências de dados** data dependencies
- o escalonamento de instruções pode ser afetado por instruções nos outros estágios
  - $\exists$ relacionamento entre instruções nos estágios
    $\Longrightarrow$ **dependências de controle** control dependencies

Desempenho da segmentação

$$\text{ganho} = \frac{\text{CPI sem pipeline}}{\text{CPI com pipeline}} \times \frac{\text{ciclo sem pipeline}}{\text{ciclo com pipeline}}$$

instr ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13
ld r1,200(r8)	B	D	E	M	R								
add r3,r4,r6						B	D	E	R				
st r2,100(r9)										B	D	E	M

ld r1,200(r8)	B	D	E	M	R								
add r3,r4,r6		B	D	E	B	R							
st r2,100(r9)			B	D	E	M	R						

Desempenho da segmentação (cont.)

$$\text{tempo de CPU} = \text{núm instr} \times \text{CPI} \times \text{período do relógio}$$

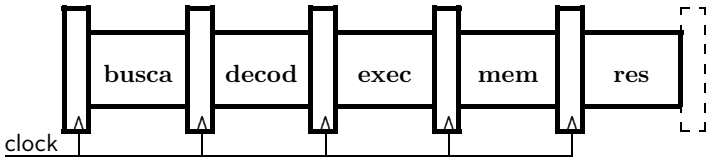
$$\text{ganho} = \frac{\text{tempo méd/instr sem pipeline}}{\text{tempo méd/instr com pipeline}}$$

$$= \frac{\text{CPI sem pipeline}}{\text{CPI com pipeline}} \times \frac{\text{ciclo sem pipeline}}{\text{ciclo com pipeline}}$$

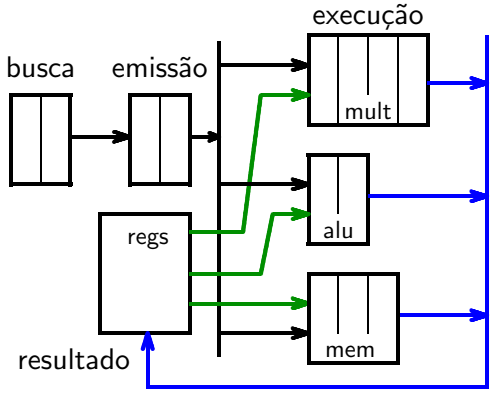
$$= \frac{1}{1 + \text{ciclos perdidos/instr}} \times \text{profundidade}$$

Tipos de processadores segmentados

linear

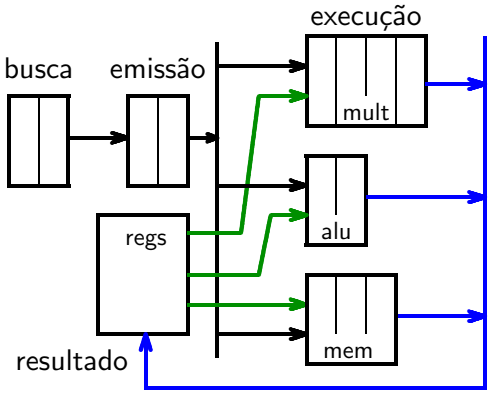


não-linear



Pipeline não-linear

Instruções completam fora-de-ordem por causa das ≠s latências das unidades funcionais
$\mathcal{T}_{\text{mult}}$=4
$\mathcal{T}_{\text{alu}}$=2
$\mathcal{T}_{\text{mem}}$=4



Barramento de resultado deve ser reservado quando instrução é emitida para execução (reg de deslocamento)
⟶ se barramento estará ocupado, segura execução

Riscos com segmentação

Riscos são condições que levam a comportamento incorreto **se** medidas apropriadas não forem tomadas

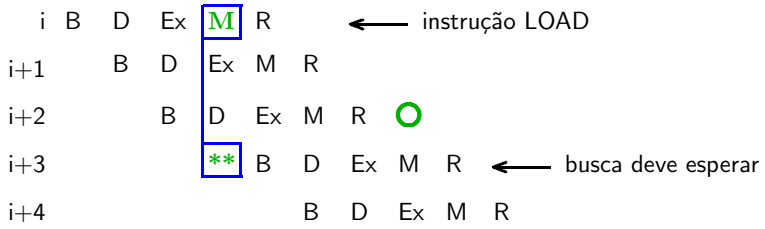
- Riscos estruturais — structural hazards
  - quando duas instruções **diferentes** necessitam do **mesmo** recurso no **mesmo** ciclo
- riscos com dados — data hazards
  - quando duas instruções **diferentes** usam **mesmo** local de armazenamento;
  - resolução do risco deve garantir aparência de que instruções executaram na ordem seqüencial correta
- riscos de controle — control hazards
  - quando uma instrução determina **quais** instruções serão executadas a seguir (desvios, saltos, funções)

Riscos com segmentação

- Riscos
  - potenciais violações de dependências no programa
  - implementação deve garantir que dependências não são violadas
- Resolução de riscos
  - estática: compilador/programador garante corretude
  - dinâmica: hardware verifica conflitos em tempo de execução
- Inter-travamentos no pipeline interlocks
  - mecanismo para resolução dinâmica de riscos
  - deve detectar e resolver dependências em tempo de execução

Riscos estruturais

- Duas instruções **diferentes** necessitam do **mesmo** recurso no **mesmo** ciclo
  - exemplo: processador em que `loads` e `stores` usam mesma porta de memória que busca de instruções



  - nenhuma instrução completa no oitavo ciclo (bolha)
  - Arquitetura Harvard *versus* Arquitetura Princeton

Resolução de riscos estruturais

Riscos estruturais podem ser reduzidos se:

- cada instrução usa recurso somente uma vez,
- sempre no mesmo estágio, e
- durante um ciclo.
- RISCs são projetados para satisfazer estes requisitos:
  - ▷ ∄ riscos estruturais além dos acessos à memória porque conj de instruções é simples, **E se** pipeline é linear
  - ▷ por causa do pipeline, memória deve ser capaz de atender ≥ 1 requisição/ciclo 1 instr/ciclo + 1/4 ld-st/ciclo

Riscos com dados

DATA HAZARDS:

Quando duas instruções **diferentes** usam
mesmo local de armazenamento

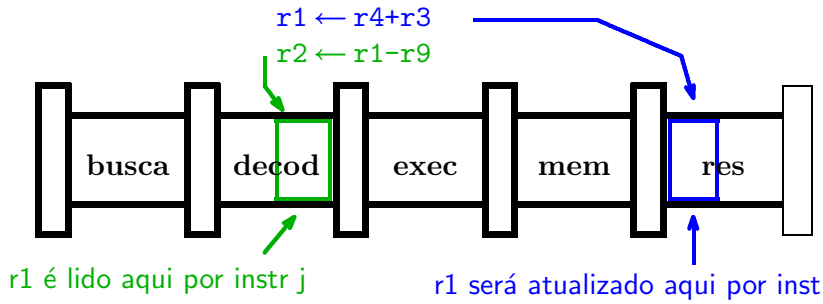
Deve parecer que instruções executam na ordem seqüencial correta

```
i:  r1 ← r4 + r5
j:  r2 ← r1 - r9     r1 foi produzido por i
k:  r1 ← r6 ⊕ r3     valor de r1 em i é sobre-escrito
    resultado de i;j;k é o mesmo que i;k;j ?
```

Convenção: **nome do risco** é a ordem do programa que
deve ser preservada pela implementação (segment, superescalar)

Riscos com dados - RAW

- Read-After-Write (RAW)
  instr **j** tenta ler operando r1 ANTES que instr **i** escreva resultado
  ```
  i:  r1 ← r4 + r3
  j:  r2 ← r1 - r9
  ```
- Risco decorre de uma dependência de dados, causada pela comunicação entre as duas instruções add e sub através de r1



Riscos com dados - WAR

- Write-After-Read (WAR)
  instr **j** escreve operando em r2 ANTES que instr **i** leia-o
  ```
  i:  r4 ← r2 + r3
  j:  r2 ← r1 - r9
  ```
- Risco decorre de uma anti-dependência
  dependência artificial causada pelo re-uso do nome r2
- anti-dependências não ocorrem num pipeline linear de 5 estágios:
  - ▷ leituras ocorrem sempre no estágio 2 (decod)
  - ▷ escritas ocorrem sempre no estágio 5 (result)
  - ▷ MAS anti-dependências ocorrem em pipelines não-lineares
    p.ex CRAY e super-escalares

Riscos com dados - WAW

- Write-After-Write (WAW)
  instr **j** escreve operando em r3 ANTES que instr **i** escreva-o

```
i:  r3 ⟵ r4 + r5
j:  r3 ⟵ r8 - r9
```

- Risco decorre de uma dependência de saída
  dependência artificial causada pelo re-uso do nome r3
- dependências de saída não ocorrem num pipeline linear de 5 estágios
  - ▷ escritas em regs ocorrem somente no estágio 5 (result)
  - ▷ MAS dependências de saída ocorrem em pipelines não-lineares
- Read-After-Read (RAR) não é risco

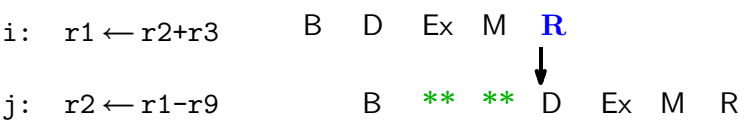
Dependências e riscos

dependências são características do programa
o programador escreveu o código numa certa ordem

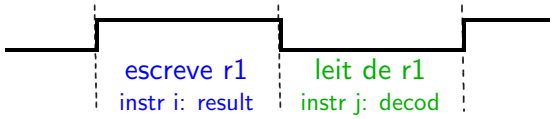
riscos decorrem de características da implementação
o projetista do processador escolheu truques "perigosos"

Solução simples para RAW (parcial)

- Circuito detecta risco, e então insere bolha:
  atrasa instrução **j** até ocorrer escrita do resultado de **i**
  - ∗ solução simples – segura instrução dependente na **busca** *stall*
  - ∗ desempenho ruim por causa dos ciclos desperdiçados

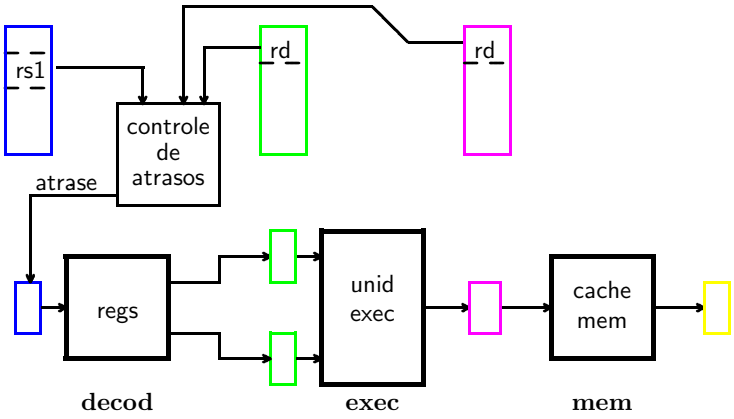


- solução pressupõe que, em cada ciclo,
  registradores são atualizados e então são lidos



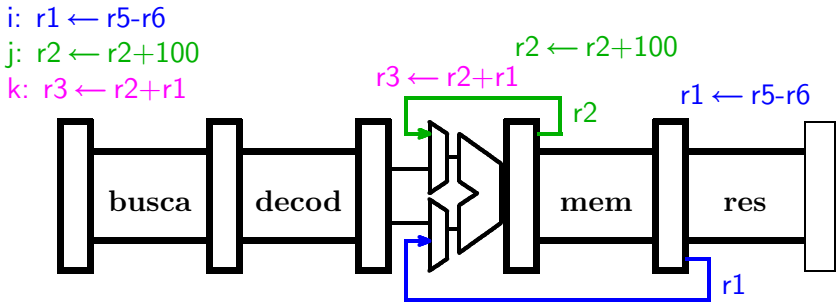
Controle de atrasos *(stalls)*

- Compara com estágios posteriores
  - ★ `if (rs1(decod)==rd(exec) || rs1(decod)==rd(mem)) { bolha }`
  - ★ mesmo para `rs2`
  - ★ nem todos conflitos são riscos: st não escreve reg, addi não lê reg...



Adiantamento

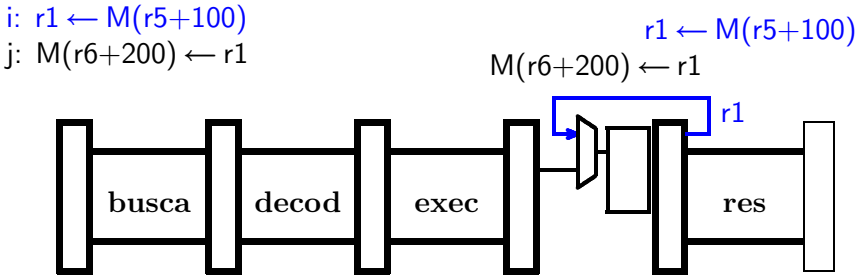
Ao invés de atrasar, adianta resultado para entradas da ULA:
usa controle de atrasos para decidir se deve adiantar, e
usa multiplexadores para escolher fonte do resultado.



Adiantamento = forwarding, bypassing, short-circuiting

Adiantamento

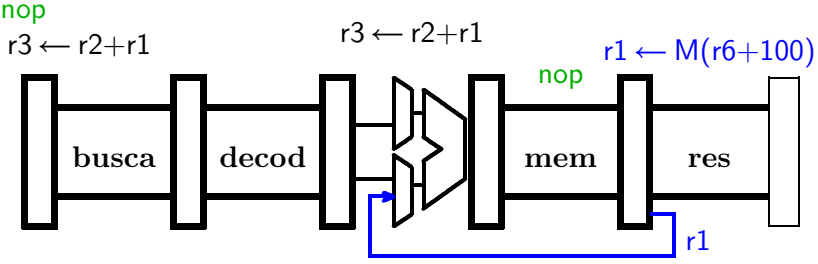
Adiantamento para o estágio de memória
`load` seguido de `store`



Adiantamento

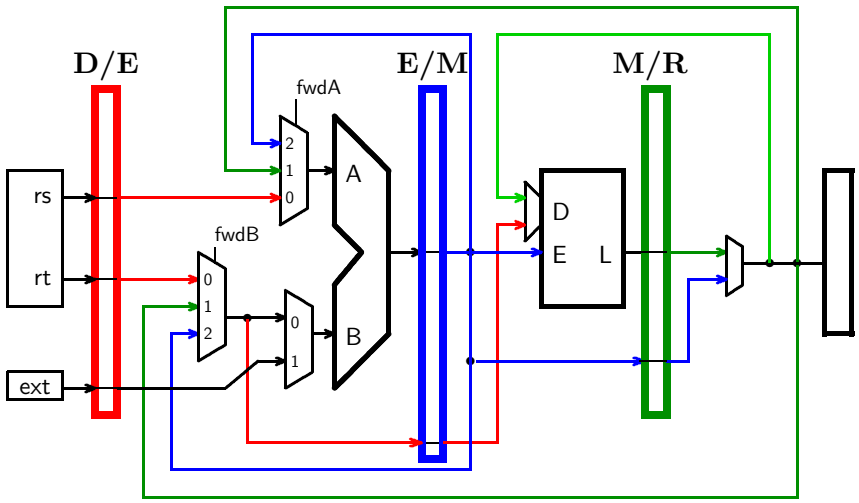
Adiantamento para o estágio de memória: `load` seguido por `add`

i: r1 ← M(r6+100)
j: nop
k: r3 ← r2+r1



- Risco deve ser detectado por hardware e bolha inserida (nop)
  - ⋆ desempenho cai por causa da bolha
- Compilador deve tentar preencher bolha com instrução "boa"
  - ⋆ load delay slot usado no CdI MIPS-I e eliminado em MIPS-II por que?

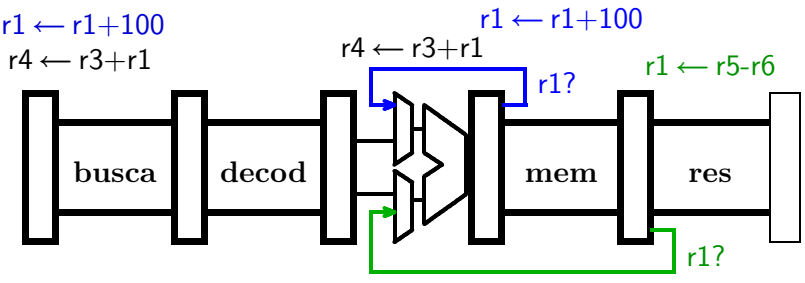
Adiantamento - circuito completo



Pergunta:

Quem fornece r1 para instrução **k**?

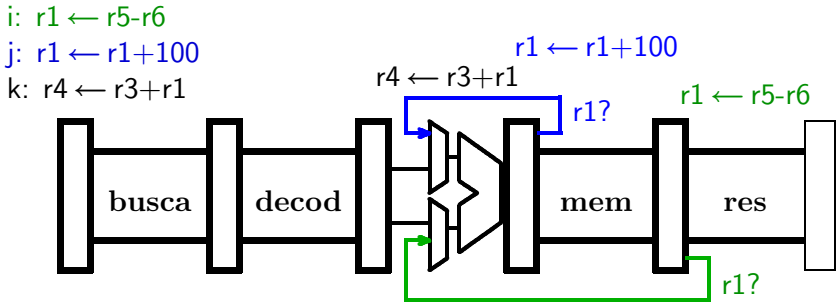
i: r1 ← r5-r6
j: r1 ← r1+100
k: r4 ← r3+r1



(a) instr **i**
(b) instr **j**
(c) instr **k**
(d) qualquer dentre **j,k**

Resposta:

Quem fornece r1 para instrução **k**?



(a) instr **i**

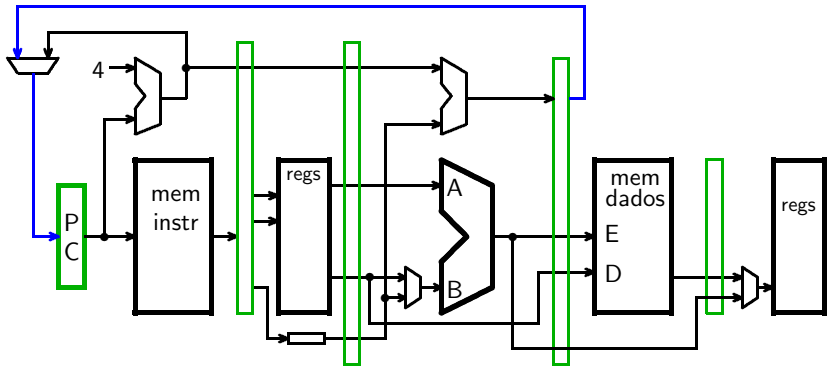
(b) instr j é a que mantém a ordem seqüencial

(c) instr **k**

(d) qualquer dentre **j,k**

Riscos de Controle

Quando uma instrução determina **quais** instruções serão executadas a seguir (desvios, saltos, funções)



Desvio é efetuado em **mem**, após comparação na ULA

$\Longrightarrow$ as instruções no caminho não-tomado devem ser anuladas

Riscos de Controle

		1	2	3	4	5	6	7	8	9	
i:	beq r1,r0,8	B	D	E	M	R					
i+1:	sub r2,r8,r9		B	D	E	*	*			anulada	
i+2:	lw rx,C(ry)			B	D	*	*	*		anulada	
i+3:	sw ri,D(rj)				B	*	*	*	*	anulada	
i+8:	add r3,r4,r5					B	D	E	M	R	
i+9:	add r6,r7,r8						B	D	E	B	R

Se CPI=1, 30% das instruções são desvios, penalidade de 3 ciclos

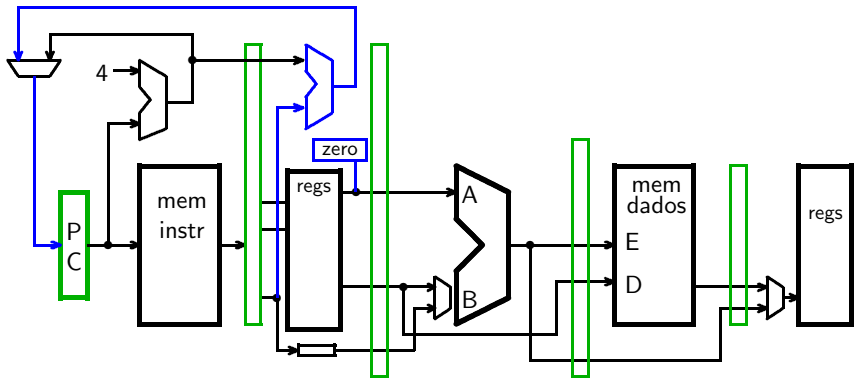
$\Longrightarrow$ CPI_{novo}=1.9

- Desempenho pode ser melhorado:
  - ∗ descobre direção antes: compara com zero em **decod**
  - ∗ computa endereço de destino: adiciona somador em **decod**

  $\longrightarrow$ penalidade de UM ciclo ao invés de três

Riscos de Controle

Desvio é efetuado em **decod**, após comparação com ZERO



Se CPI=1, 30% das instruções são desvios, penalidade de 1 ciclo
$\Longrightarrow$ CPI_{novo}=1.3

Tratamento de Riscos de Controle

- Tratamento de riscos de controle é MUITO importante
  pelo impacto no desempenho
  - de 1/3 a 1/6 de todas instruções são desvios
  - penalidade de um, dois ou três ciclos
  - pipelines com mais estágios $\Longrightarrow$ desempenho pior
- Atrasa instruções até decidir se desvia
- Previsão de desvios $\longrightarrow$ prever direção do desvio
  - reduz/elimina penalidade se previsão correta
  - pode aumentar penalidade quando erra
  - Técnicas:
    previsão estática – segue sempre mesmo caminho
    previsão dinâmica – depende do comportamento do programa

Previsão de desvios

- **Previsão Estática**
  - ▷ prevê sempre não-tomado — MIPS, SPEC95, ≈ 40%
  - ▷ prevê sempre tomado — MIPS, SPEC95, ≈ 60%
  - ▷ prevê que para-trás é tomado
  - ▷ previsão associada a opcodes
  - ▷ desvios atrasados — delayed branches
- **Previsão dinâmica**
  - ▷ será visto mais tarde

Desvios Atrasados

- Sempre executa a próxima instrução — OK com pipelines simples

```
i:    PC ⟵ PC+8 if r1 == 0
i+1:  r2 ⟵ r8 - r9          sempre é executada
...
i+8:  r3 ⟵ r4 + r5
```

- É necessário que branch delay slot seja preenchido
  - ▷ preenche com instrução antes do desvio (`i-1`)
  - ▷ preenche com instrução de destino
    - ∗ SE é seguro executar instrução de destino
    - ∗ ajuda somente no caso do desvio tomado
  - ▷ preenche com instrução após o desvio (`i+2`)
    - ∗ SE é seguro executar instrução após o desvio
    - ∗ ajuda somente no caso do desvio não-tomado

Desvios Atrasados, Loads Atrasados

Arquiteturas que expõem riscos do pipeline ao software,
como nos casos de *delayed load*s e *branch delay slot*s
resultam em programas com número significativo de nops
inseridos pelo compilador.

Estas soluções misturam arquitetura (ISA)
com implementação (pipeline linear)

ISSO É UMA MÁ IDÉIA©

resumo – Riscos de Controle

- Riscos de controle tem efeito devastador no desempenho
- Mecanismos simples de previsão de desvios:
  - ⋆ segura fluxo até decidir direção do desvio
    (penalidade 1-2 ciclos)
  - ⋆ prediz que desvio é sempre não-tomado
    (penalidade 1 ciclo, ≈40% das vezes)
  - ⋆ prediz que desvio é sempre tomado
    (penalidade 1 ciclo, ≈60% das vezes)
- desvio atrasado "esconde" penalidade quando desvio não-tomado
  - ⋆ ocupar o espaço pode ser difícil
  - ⋆ mistura arquitetura com implementação

revisão – Riscos

- **modelo seqüencial de execução:**
  cada instrução é completamente executada
  e altera o estado do processador
  antes do início da próxima instrução
- **Riscos** são potenciais violações de dependências no programa
- implementação deve garantir que dependências não são violadas
- Resolução de riscos:
  * estática: compilador/programador garante corretude
  * dinâmica: hardware verifica conflitos em tempo de execução
- Inter-travamento no pipeline — interlocks
  * mecanismo para resolução dinâmica de riscos
  * deve detectar e resolver dependências em tempo de execução

mais revisão – Riscos

- Risco estrutural — structural hazards
  ▷ duas instruções **diferentes** necessitam do
  mesmo recurso no **mesmo** ciclo
- Risco de dados — data hazards
  ▷ duas instruções **diferentes** usam **mesmo** local de armazenamento
  aparência de execução na ordem seqüencial correta
- Risco de controle — control (flow) hazards
  ▷ uma instrução determina **quais** instruções serão executadas a seguir (desvios, saltos, funções)
  pior impacto no desempenho

Dependências de Controle

- Desempenho dos processadores é limitado pelos desvios
- Maioria das dependências de dados podem ser eliminadas pelo compilador + adiantamento;
- Dependências de controle não podem ser eliminadas por causa do comportamento dinâmico dos programas...
- Regra dos 85/60: medições indicam que aproximadamente
  * 85% dos desvios "para trás" são tomados
  * 60% dos desvios "para frente" são tomados
- **solução:** adivinhar destino do desvio para antecipar a busca da próxima instrução
  porque fazer alguma coisa é melhor que não fazer nada

Riscos de controle e previsão de desvios

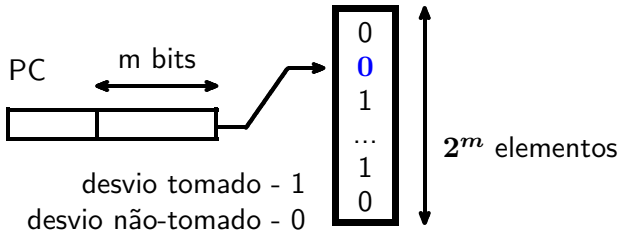
Previsão Dinâmica de Desvios

- Passado recente pode auxiliar a prever futuro próximo:
  usar a história dinâmica dos desvios para prever sua direção
  * se desvio será tomado ou não-tomado
  * qual o endereço de destino
- porque é dinâmico, previsor pode se ajustar a
  * fases distintas da execução do programa
  * comportamento de desvios específicos

Previsão Dinâmica de Desvios

Tabela com $\mathbf{2^m}$ bits no estágio de busca

- acessa tabela com **m** bits do PC
  * tabela contém **1** se desvio foi tomado na última execução
  * tabela contém **0** se desvio não foi tomado na última execução
- prevê que nesta execução tomará mesmo caminho que na anterior
- atualiza o bit quando errar a previsão



Previsão Dinâmica de Desvios

Exemplo: loop executa três voltas e então prossegue:

Na previsão com um bit, ocorrem duas previsões erradas
a cada mudança c.r.a história recente

resultado	t	t	t	n	t	t	t	n	t	t	t	n	t
previsão	t	t	t	T	N	t	t	T	N	t	t	T	N
acerto?				E	E			E	E			E	E

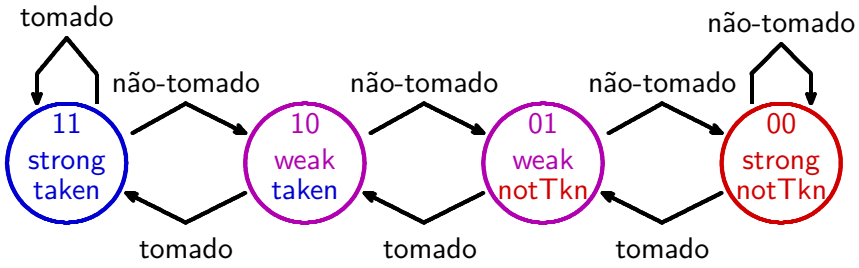
Com este método são duas as previsões erradas
a cada vez que seqüência muda...

Previsores melhores: contadores de 2 bits

Contador de dois bits implementa histerese e melhora previsão:

resultado	t	t	t	n	t	t	t	n	t	t	t	n	t
estado	wt	st	st	st	wt	st	st	st	wt	st	st	st	wt
previsão	t	t	t	t	t	t	t	t	t	t	t	t	t
acerto?				E				E				E	

Previsões erradas caem para a metade.



Previsores melhores: correlação de previsões

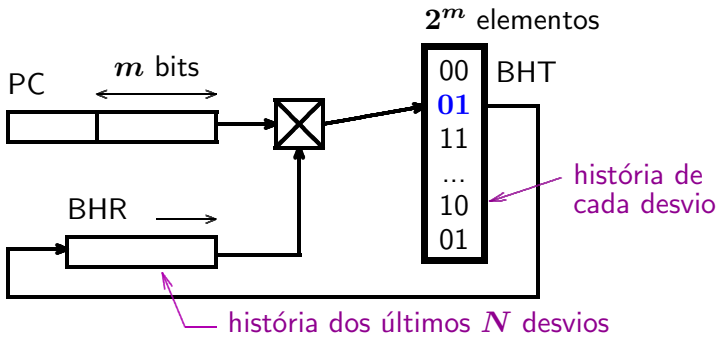
Desvios em instruções distintas podem ser relacionados:

```
if (a < 0) a = 0;
if (b > 0) b = 0;
if (a != b) { ...}
```

Se as duas primeiras condições são verdadeiras, a terceira será falsa; portanto a idéia é:

- manter história de todos os desvios recentes
  $\Longrightarrow$ história aproxima caminho seguido pelo programa
- Branch History Register (BHR) é um registrador de deslocamento que mantém resultado dos últimos **N** desvios
- PC e BHR são combinados para acessar tabela de previsão
- taxas de acerto ficam entre 80% e 90%

Previsores melhores: correlação de previsões



Apontador para tabela de previsão é uma combinação dos conteúdos do PC e do *Branch History Register* soma, sobreposição, ou concatenação

Gshare: um BHR de m-bits, ou-exclusivo do BHR com PC

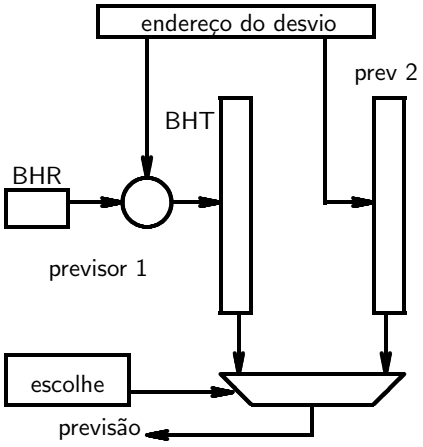
Previsores ainda melhores: mais de um previsor

Desvios tem comportamentos distintos
⟶ usar previsores diferentes em desvios diferentes.

escolha:
se previsor escolhido errou
e outro acertou
inverte escolha

previsores híbridos,
competitivos (tournament)

Gshare + contador de 2 bits



Previsão de endereço de destino

Tabela de endereços de destino (Branch Target Buffer [BTB])
permite prever endereço de destino
se acerta previsão, evita inserção de bolha no pipeline

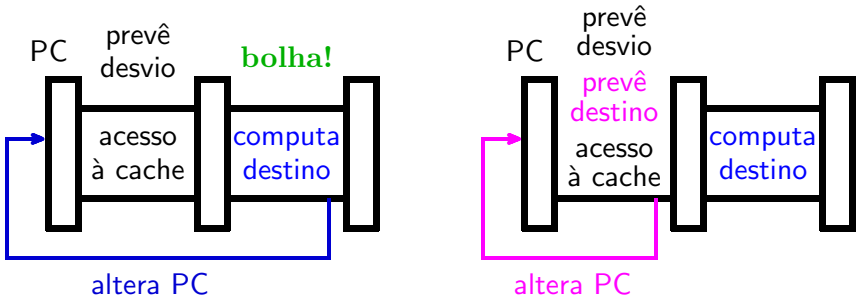


Tabela de endereços de destino

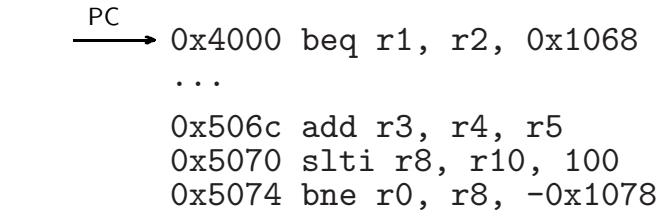
A tabela de endereços de destino é uma cache especial:

- índice e etiqueta são comparados com endereço da instrução de desvio
- conteúdo do BTB é o endereço de destino (cache de destinos)
- bits de previsão mantidos junto com endereço de destino
- geralmente associado à cache de instruções

Branch Target Cache:

* armazena instrução de destino junto com endereço
* permite eliminar o desvio do fluxo de instruções
branch folding

Tabela de endereços de destino



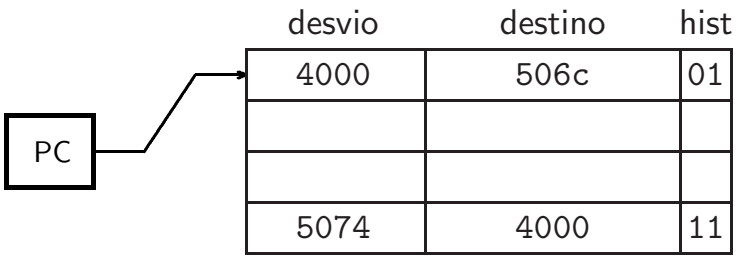


Tabela de endereços de destino

Administração da tabela:

- armazena desvios não-tomados?
- o que fazer quando inicia execução (tabela vazia)?
- o que fazer com conflitos na tabela (tabela cheia)?
- o que fazer quando errar a previsão?
- quanto custa (CPI) errar uma previsão?

Previsão de chamada e retorno de funções

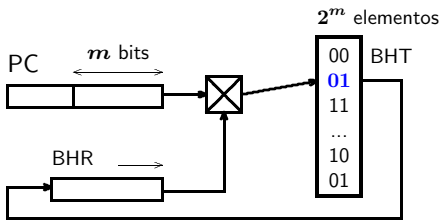
- Destino de saltos indiretos para retorno de função (`jr`) é muito difícil de prever
  - uma função pode ter vários endereços de retorno estáticos e/ou dinâmicos
- **Solução:** implementar uma **pilha de retorno de funções** no estágio de busca do processador
  - salva endereço de retorno na pilha — instrução `jal`
  - desempilha ao retornar e prevê destino — instrução `jr r31`
- O quê fazer com saltos indiretos que ∉ à funções? — instrução `jr`

Resumo de Previsão de Desvios

- Previsão é útil em pipelines simples
- é imprescindível em processadores super-escalares
- processadores modernos usam
  - previsores sofisticados
  - tabela de endereço de destino
  - pilha de retorno de funções

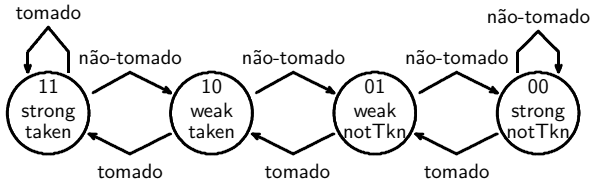
Pergunta:

```
1 for (i=0; i<100; i++)
2    if (i%2 == 0)
3       ...
4    else
5       ...
```



Quantos erros de previsão na linha **2** para previsor BHR+BHT?
BHT iniciaizado com 00 (não-tomado)

(a) 49
(b) 50
(c) 51
(d) n.d.a.



Resposta:

Estados: 11-St 10-wt 01-wn 00-Sn

BHR com 8 bits, insere resultado na posição menos-significativa

BHT com 256 posições, indexada com 8 bits do PC

PC = 0x200 (para simplificar), operação do PC com BHR é XOR

Estruturas inicializadas com 0 em todas as posições

```
1 for(i=0; i<100; i++)
2    if (i%2 == 0)
3       ...
4    else
5       ...
```

quantos erros de previsão na linha 2?
OU
quantas voltas do laço para treinar o previsor?

Resposta: (cont)

```
i dsv  BHR       pos BHT  nova  novoBHR    acerto?
0  nt  0000.0000  0 00 -> 00 0000.0000 0   sim
1  t   0000.0000  0 00 -> 01 0000.0001 1   não
2  nt  0000.0001  1 00 -> 00 0000.0010 2   sim
3  t   0000.0010  2 00 -> 01 0000.0101 5   não
4  nt  0000.0101  5 00 -> 00 0000.1010 a   sim
5  t   0000.1010  a 00 -> 01 0001.0101 15  não
6  nt  0001.0101 15 00 -> 00 0010.1010 2a  sim
7  t   0010.1010 2a 00 -> 01 0101.0101 55  não
8  nt  0101.0101 55 00 -> 00 1010.1010 aa  sim
9  t   1010.1010 aa 00 -> 01 0101.0101 55  não - 5 erros
10 nt  0101.0101 55 00 -> 00 1010.1010 aa  sim
11 t   1010.1010 aa 01 -> 10 0101.0101 55  sim <--acerto
```


Interrompemos a nossa programação...

Interrupções e Excessões

- Interrupções
  - ∗ geralmente com causa externa ao processador (assíncrona)
  - ∗ não são relacionadas a uma instrução específica
  - ∗ Exemplos: interrupção de dispositivo; falta de energia
- Excessões
  - ∗ relacionadas à execução de uma instrução específica (síncrona)
  - ∗ Exemplos: overflow, falta de página
- *Traps*
  - ∗ relacionadas à execução de uma instrução específica
  - ∗ rotina (hw+sw) de tratamento de excessões

Eventos Síncronos vs Assíncronos

- Síncrono evento relacionado com a seqüência de instruções ocorre durante a execução de uma instrução
  - ⋆ deve parar instrução que está executando correntemente
  - ⋆ falta de página durante instrução `load` ou `store`
  - ⋆ excessões aritméticas
  - ⋆ chamadas de sistema (instruções `trap` ou `syscall`)
- Assíncrono evento sem relação com seqüência de instruções causada por um evento externo
  - ⋆ desnecessário interromper instruções que estão executando
  - ⋆ interrupções por dispositivos de E/S
  - ⋆ interrupções por eventos catastróficos (erro na memória)
- Semi-síncrono
  - ⋆ evento externo que pode interromper seqüência de instruções para garantir serviços (alta disponibilidade)

Interrupções e Excessões – arquitetura

- Registrador de Interrupção
  - ∗ vetor de bits indicando quais interrupções/excessões ocorreram
- Registrador de máscara
  - ∗ vetor de bits indica quais inters/excessões estão desabilitadas
  - ∗ escrever no registrador de máscara é instrução privilegiada
  - ∗ alguns bits podem ser atualizados em modo usuário (overflow)
  - ∗ algumas interrupções/excessões não são mascaráveis

Interrupções/Excessões Precisas

Uma interrupção ou excessão é considerada **precisa** se existe uma única instrução (ou ponto de interrupção) tal que todas as instruções anteriores àquela tenham alterado o estado, e nenhuma instrução após (e incluindo) aquela tenham modificado o estado.

Isso significa que a execução pode ser re-iniciada a partir do ponto de interrupção e resultados corretos serão produzidos.

Ver artigo de Smith & Pleszkun para soluções.
IEEE-TC 37:5 1988

Interrupções Precisas

- **Interrupções Precisas** facilitam MUITO o trabalho do SO na continuação do programa, ou na depuração
- Excessão/trap
  - ▷ todas instruções antes da causadora completam
  - ▷ nenhuma das instruções após a causadora completam
  - ▷ instrução causadora **ou completou ou não iniciou**
  - ▷ PC aponta instrução causadora
- Interrupção
  - ▷ Mesmo que excessão, mas $\not\exists$ instrução causadora
  - ▷ Deve parecer que ocorreu **entre** duas instruções
  - ▷ PC aponta para instrução que seria executada

Tratamento de interrupções/traps

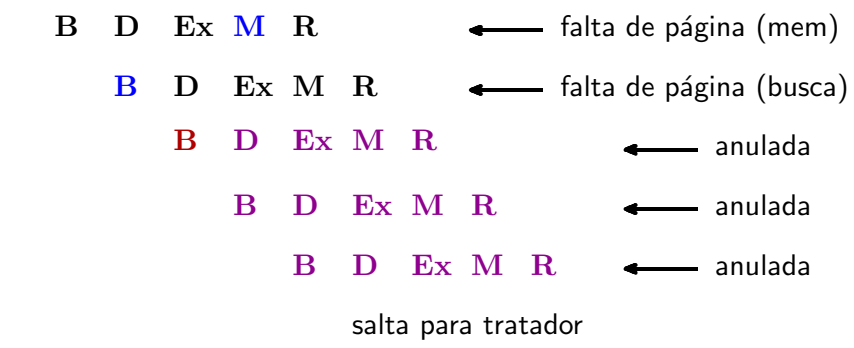
Quando ocorre interrupção/trap:

- efetua salto para rotina do SO
  - ∗ vetor de tratadores, em geral amarrado no cj de instruções
- computa endereço de retorno
- salva informação de estado essencial
  - ∗ PC
  - ∗ CCs (condition codes)
  - ∗ PSW (processor status word)
- troca modo de execução: usuário $\Longrightarrow$ sistema

Implementação de interrupções/traps

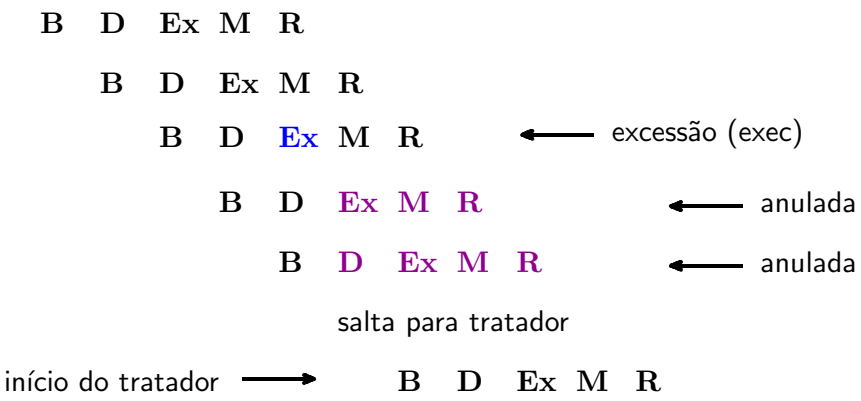
- Precisão é importante
- Interrupções e excessões simultâneas
  - ▷ interrupções de E/S
  - ▷ **busca** – falta de página, referência desalinhada, protection fault
  - ▷ **decod** – instrução ilegal ou privilegiada
  - ▷ **exec** – excessões de aritmética (overflow, divisão por zero)
  - ▷ **mem** – falta de página, referência desalinhada, protection fault
  - ▷ **res** – nenhuma

Falta de Página



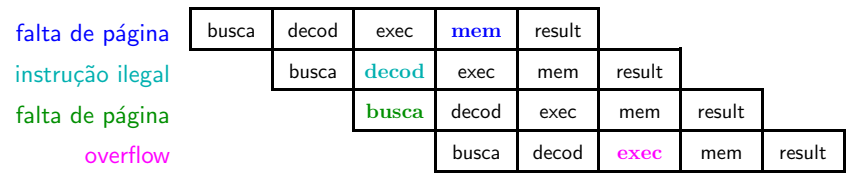
- Instrução anterior completou **res**//**mem**
- anula todas instruções subseqüentes
- impede que instrução causadora altere estado (se for `load`)

Excessão de Aritmética



- Instruções anteriores podem completar
- anula todas instruções subseqüentes

Excessões Múltiplas



- Pipeline pode resolver
  - ∗ estado c.r.a excessões passa pelos estágios junto com instrução
  - ∗ mantém valor do PC junto a cada instrução
  - ∗ não trata excessão até instrução chegar a **result**
- Interrupção é detectada na **busca** e instrução NOP é inserida
- Quando instrução chega a **result**
  - ∗ salva PC⟶ EPC; endereço do vetor de interrupções ⟶ PC
  - ∗ transforma instruções nos outros estágios em NOPs

Complicações no Conjunto de Instruções

- Estado espalhado em vários recursos (CCs)
- Atualização antes de **mem**/**res** (auto-incremento)
- Instruções que completam fora de ordem
  - outros mecanismos serão estudados adiante
- Interrupções ocorrem a qualquer momento
- O caso freqüente não é o único/mais importante
  - Desempenho não é o parâmetro mais crítico
  - Os casos raros *devem* funcionar corretamente

Resumo – previsão de desvios e excessões

- Previsão de desvios é imprescindível
  - contador de 2 bits com histerese para prever cada desvio
  - tabela(s) com $\mathbf{2}^{m}$ previsores + registrador de deslocamento
  - registrador de história dos desvios *branch history register*
  - tabela com endereços de destino *branch target buffer*
  - pilha de retorno de funções
- Interrupções e excessões complicam muito projeto de pipelines
  - precisão é importante mas implementação é cara (hardware)
  - drenar o pipeline custa caro...
  - detecta evento cedo e trata-o tarde em **result**

Revisão – previsão de desvios e excessões

- Previsão de desvios é imprescindível pelo efeito no desempenho
  - contador de 2 bits com histerese para prever cada desvio
  - tabela(s) com $\mathbf{2}^{m}$ previsores + registrador de deslocamento
  - registrador de história dos desvios *branch history register*
  - tabela com endereços de destino *branch target buffer*
  - pilha de retorno de funções
- Interrupções e excessões complicam muito projeto de pipelines
  - precisão é importante mas implementação é cara (hardware)
  - drenar os segmentos custa caro...
  - detecta evento cedo e trata-o tarde em **result**

Superpipelining

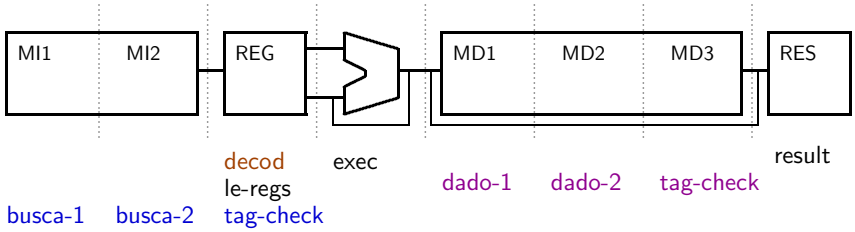
tempo de CPU = núm instr x CPI x período do relógio

Se aumentar velocidade do relógio (e reduzir período),
pode haver ganho de desempenho
⟶ implementar estágios com menor latência e em maior número
⟶ reduz período mas aumenta CPI

Com um pouco de sorte (uh?) relógio compensa CPI

Exemplos:
MIPS 4000 usa dois estágios na busca e três em MEM/cache
Pentium III tem pipeline com 10 estágios e
Pentium IV tem pipeline com 20 estágios,
com dois ciclos só para transmitir bits através da CPU

Superpipelining – MIPS R4000



busca dura **2** ciclos – verifica acerto na L1-I enquanto decodifica
desvios tomados causam três bolhas

mem dura **3** ciclos por causa da verificação de acerto
⟶ não pode escrever valor até estar certo do acerto na L1-D
só pode usar valor de `load` depois de dois ciclos
usa valor do `load` *antes de verificar etiqueta?*

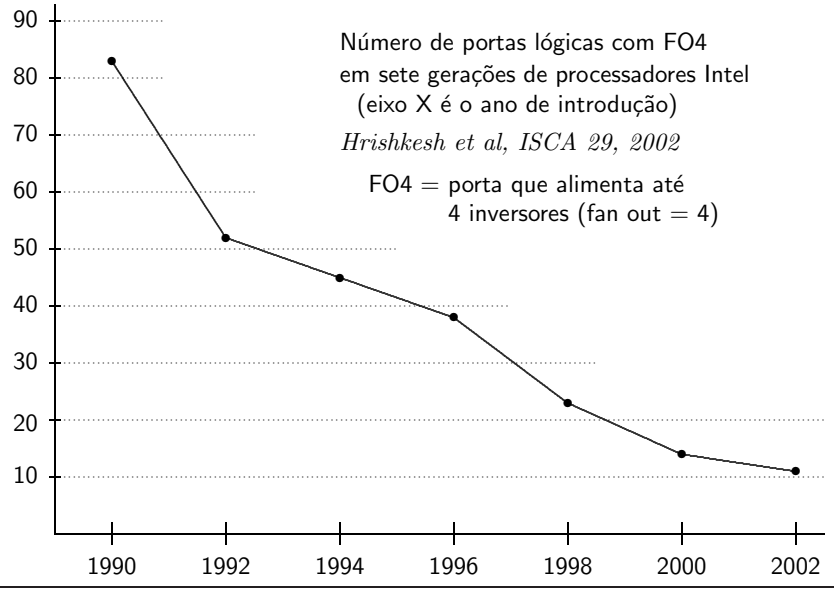
Superpipelining

Ganhos:

freqüência do relógio mais alta	moda até há pouco
permite ligar CPU rápida à memória lenta	interf c/ mem segmentada

Perdas:
CPI mais alto
penalidade maior nos desvios
penalidade maior nas faltas nas caches (mais ciclos)
penalidade maior no uso do valor de `load`
penalidade maior nas excessões
maior complexidade:
circuitos de adiantamento, bloqueios...

Superpipelining



Desempenho de Pipelines

$$\text{ganho} = \frac{\text{CPI sem pipeline}}{\text{CPI com pipeline}} \times \frac{\text{ciclo sem pipeline}}{\text{ciclo com pipeline}}$$

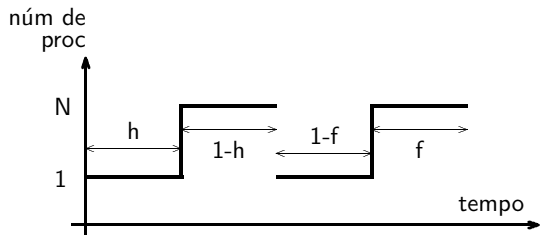
Desempenho ótimo — ignorando latches

ganho = número de estágios

Causa das Perdas:

- dependências estruturais — escalonamento de instruções resolve
- dependências de dados — escalonamento e adiantamento resolvem
- dependências de controle — previsão resolve $\approx 85\%$ dos casos
- Lei de Amdahl: ganho é limitado pelo pior componente

Lei de Amdahl

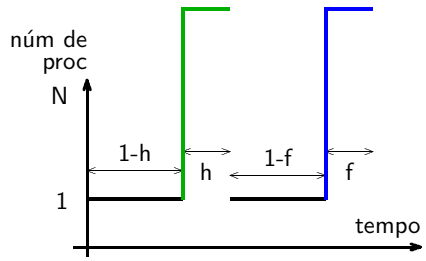


Ganho $= \frac{1}{(1-f)+f/g}$

- h fração de tempo em código serial
- $1-h$ fração de tempo que pode ser paralelizada (N processadores)
- f fração que pode ser vetorizada
- g ganho durante f

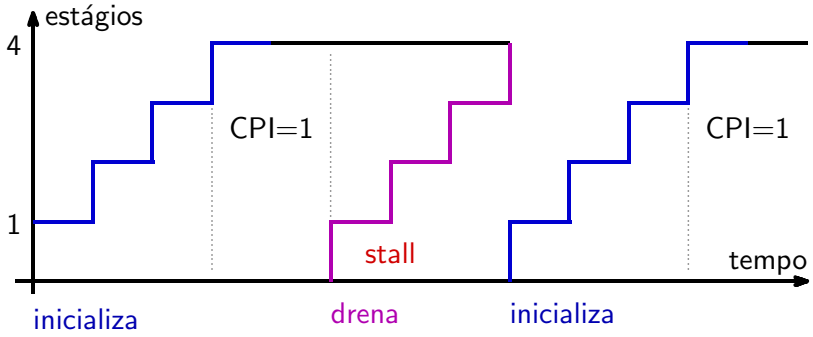
Lei de Amdahl

$$\lim_{g \to \infty} \frac{1}{(1-f) + f/g} = \frac{1}{(1-f)}$$

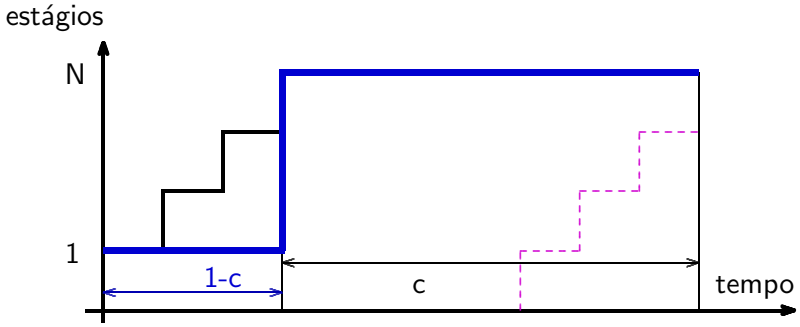


Desempenho é limitado pela parte que
não pode ser paralelizada (h) ou vetorizada (f)

Desempenho de Pipelines I



Desempenho de Pipelines II

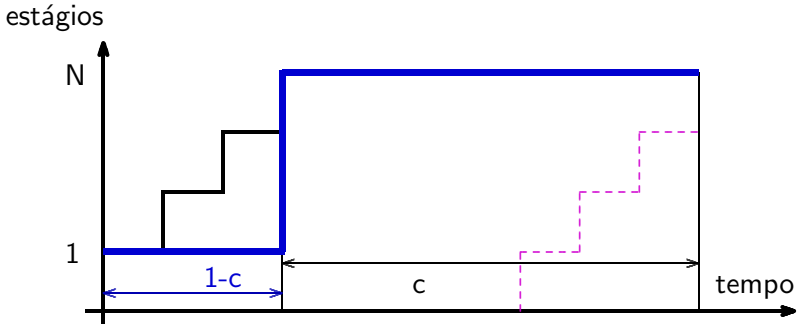


- c fração de tempo com pipeline cheia
- $1 - c$ fração de tempo em que ocorrem bolhas
- quando c é um pouquinho abaixo de 100%,
  a queda no desempenho é enorme!
  $\longrightarrow$ a fração $1 - c$ deve ser minimizada

Desempenho de Pipelines III

Se executar mais de uma instrução por ciclo,
a fração $1 - c$ pode ser reduzida pelo fator de escalaridade S

$$\text{Ganho} = \frac{1}{\frac{(1-c)}{S} + \frac{c}{g}}$$



Pipeline para obter CPI ≈ 1

- Gargalo de Flynn
  - emissão de uma instrução por ciclo limita CPI=IPC=1
  - riscos + overhead $\longrightarrow$ CPI≥ 1 IPC≤ 1
  - ganhos cada vez menores com superpipelining
- solução: emitir mais de uma instrução por ciclo

	1	2	3	4	5	6	7	8	9
inst0	B	D	E	M	R				
inst1	B	D	E	M	R				
inst2		B	D	E	M	R			
inst3		B	D	E	M	R			

  - paralelismo no nível de instrução (PNI)
    instruction-level paralelism (ILP) Fischer81
  - primeiro super-escalar: IBM América $\longrightarrow$ RS/6000 $\longrightarrow$ POWER1

Processadores Escalares e Vetoriais

- Processador **escalar** opera em uma palavra de memória: **escalar**
  - processadores escalares são todos aqueles vistos até agora
- processador **vetorial** opera em um grupo de palavras: **vetor**
  - The CRAY-1 Computer System, R M Russel, CACM 21:1, jan78
  - certas instruções usam vetores como operandos
  - registradores contém vetores (64 elementos/registrador)
  - acessos à memória para carregar ou escrever vetores inteiros
  - unidades funcionais são pipelines para elementos de vetores
  - ver Apêndice G em `www.mkp.com/CA3`

Paralelismo no nível de instrução

Qual a diferença entre dependência e risco?

Paralelismo no nível de instrução (PNI)

PNI é uma propriedade do software (e não do hardware)

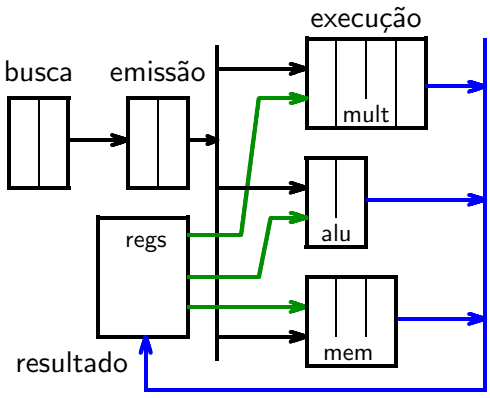
quanto paralelismo existe entre as instruções de um programa?
depende do software hw pode ou não explorá-lo

Inúmeras maneiras de explorar PNI:

- pipelining (Apêndice A)
- superescalar (Cap 3)
- execução fora de ordem (escalonamento dinâmico – Cap 3)
- escalonamento pelo compilador (escalonamento estático – Cap 4)

Pipelines não-lineares

Quatro fases:
busca ;
decod+emissão ;
execução ;
resultado

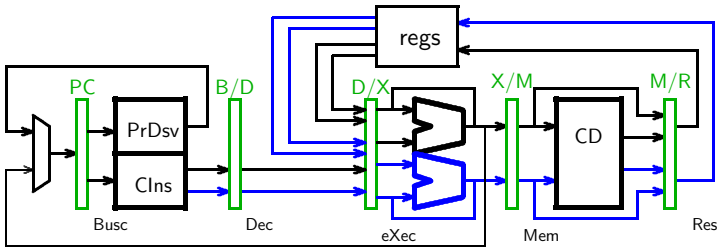


Estágio(s) de emissão envia(m) instruções para unidade funcional
latência da execução depende da unidade funcional
$\mathcal{T}_{\text{mult}} > \mathcal{T}_{\text{mem}} > \mathcal{T}_{\text{alu}}$

Superescalar: implementação base

- superescalar com escalonamento estático, em-ordem
  - ▷ executa programas seqüenciais sem modificação
  - ▷ descobre sozinho o que pode ser executado em paralelo
  - ▷ exemplos: MIPS 8000, Sun UltraSPARC, Alpha 21264

Pipeline de 5 estágios com emissão dupla



- O que é necessário para
  - * buscar duas instruções por ciclo?
  - * decodificar duas instruções por ciclo?
  - * executar duas instruções de ALU no mesmo ciclo?
  - * acessar a cache de dados duas vezes no mesmo ciclo?
  - * escrever dois registradores no mesmo ciclo?
- e se forem 4 ou 8 instruções num ciclo?

Busca com largura N

- O que é necessário na busca de N instruções em um ciclo?
- se as instruções são seqüenciais
  - * e no mesmo bloco (**k**) da cache ⟶ nada
  - * e em blocos diferentes ⟶ cache intercalada + rede de combinação

	0	1	2	3	4	5	6	7			
bloco k	x	y	z	w					⟶	xyzw	
bloco i			x	y							rede combina
bloco j					z	w			⟶	xyzw	os 2 blocos

- se as instruções não são seqüenciais
  - * dois acessos em série: acesso1 ⟶ prevê_destino ⟶ acesso2
- desvios no meio de um bloco: fácil se desvios não-tomados (NT)
  - * acesso serial + previsão em paralelo
  - * se previsão é tomado (T) ⟶ descarta instrs após desvio

Decodificação com largura N

- O que é necessário para decodificar N instruções em um ciclo?
- decodificar as instruções
  - fácil se instruções de tamanho fixo (múltiplos decodificadores ||s)
  - difícil, porém possível, se tamanho variável ⟶ x86
- ler operandos dos registradores
  - 2N portas de leitura no bloco de registradores
  - na verdade, menos que 2N porque muitos valores são adiantados
- como fica a lógica de controle dos atrasos (*stalls*)?

Decodificação com largura N

- lógica de controle de atrasos num pipeline simples:
  - `rs1(D)==rd(D/X) || rs1(D)==rd(X/M) || rs1(D)==rd(M/R)`
  - mesmo para `rs2`
  - com adiant completo: `rs1(D)==rd(D/X) && opc(D/X)==load`
- dobrando a largura de emissão, quadruplica lógica de atrasos
  - ⋆ não são só 2 instruções em Decod, mas 2 instr em todos estágios
  - ⋆ `rs1(D1)==rd(D/X1) && opc(D/X1)==load`
  - ⋆ `rs1(D1)==rd(D/X2) && opc(D/X2)==load`
  - ⋆ repetir para `rs1(D2), rs2(D1), rs2(D2)`
  - ⋆ testar dependência da segunda instr na primeira: `rs1(D2)==rd(D1)`
- num pipeline de largura N, circuito de atrasos cresce com N^2 mesmo vale para adiantamento, só que pior...
- **|lógica de controle de atrasos| $\propto N^2$**

Execução com largura N

- O que é necessário para executar N instruções em um ciclo?
- múltiplas unidades funcionais. N de cada tipo?
  - ⋆ N ULAs? Pode ser, ULAs são pequenas
  - ⋆ N divisores de ponto flutuante? Não, circuito é enorme e divPF é infreqüente
- tipicamente, usa combinação proporcional ao uso
  - ⋆ RS/6000: 1 ULA/endereços/desvios + 1 PF
  - ⋆ Pentium: 1 ULA complexa + 1 ULA simples
  - ⋆ PentiumII: 1 ULA/PF + 1 ULA + 1 load + 1 store + 1 desvios
  - ⋆ Alpha 21164: 1 ULA/PF/desvios + 2 ULA + 1 load/store
- **circuito de adiantamento $\propto N^2$**
  - lógica de controle é pequena porque variáveis tem 5 bits (regs)
  - circuitos de dados é gigantesco (32 ou 64 bits por caminho)
  - layout da fiação é infernal, MUXes são enormes e lentos
  - menos horrível se agrupar unidades funcionais em clusters

Interface de memória com largura N

- O que é necessário para acessar memória 2 vezes no mesmo ciclo?
- cache de dados com múltiplos bancos (detalhes mais tarde)
  - ▷ necessita lógica de detecção de conflitos (2 refs ao mesmo banco)
  - ▷ necessita lógica de detecção de riscos RAW
- aproximadamente 20% das instruções são `loads` e 15% `stores`
  - ▷ para largura N, são necessárias
    0,2N portas de leitura e
    0,15N portas de escrita na memória

Gravação de resultados com largura N

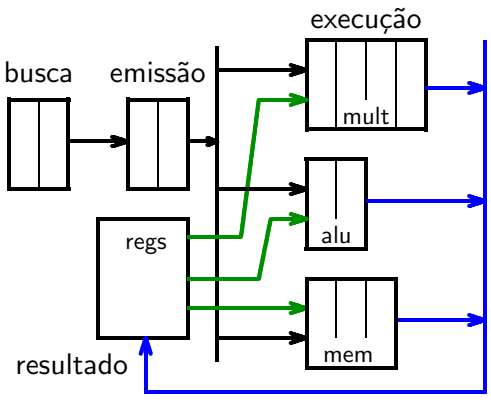
- O que é necessário para escrever 2 registradores no mesmo ciclo?
- apenas mais uma porta de escrita no bloco de registradores
  - ▷ tudo o que deveria já foi feito nos estágios anteriores...
- MAS o tratamento de excessões é ainda mais complicado
  deve-se usar buffer de re-ordenação à lá Smith&Plezkun

Modelo de pipeline

Quatro fases:
busca ;
decod+emissão ;
execução ;
resultado

cada fase pode ser
um pipeline com
mais de um estágio

nos interessa a
emissão (*issue*)



Ordem de execução dos programas

- **modelo seqüencial de execução:**
  cada instrução é completamente executada
  e altera o estado do processador
  antes do início da próxima instrução
  ⟶ modelo habitual de raciocinar sobre execução de programas
- modelos com paralelismo:
  paralelismo explícito ⟶ multiprocessadores, threads, vetores
  paralelismo implícito ⟶ hardware esconde execução paralela

Ordem

- **Emissão de instruções para execução**
  * emissão em ordem — o que vimos até agora
  * emissão fora-de-ordem
    processador examina várias instruções para decidir quais podem seguir para execução
- **ordem de execução**
  * execução em ordem — o que vimos até agora
  * execução fora-de-ordem
    se operandos e unidade funcional disponíveis então executa
- **ordem de escrever resultado** (*writeback*)
  * resultados gravados em ordem — o que vimos até agora
  * resultados gravados fora-de-ordem
    grava resultado quando não viola nenhuma dependência

Ordem de emisão

- Emissão em ordem — o que vimos até agora
  emissão pára se ∃ qualquer tipo de dependência ou risco
  . risco estrutural, de dados, ou de controle
- emissão fora-de-ordem
  processador examina várias instruções para decidir quais podem seguir para execução *lookahead*
  * se instrução bloqueia por causa de conflito por recurso
    processador segue buscando até encontrar instrução independente
  * devem ∃r recursos para executar instruções independentes
  * ordem seqüencial de execução deve ser obedecida
    mesmo se instruções executam e completam fora-de-ordem

Ordem de execução

- Execução em ordem — o que vimos até agora
  - conflitos por recursos seguram execução de instruções + novas
  - idem para dependências de dados ou de controle
- execução fora-de-ordem — explora paralelismo entre instruções
  - se operandos e unidade funcional disponíveis então executa
  - dependências de dados devem ser respeitadas
  - como descobre e resolve riscos RAW, WAR, WAW?
    - algoritmo do placar — scoreboarding, Thornton CDC6600 1961
    - algoritmo de Tomasulo — IBM 360/91 1967
    - Register Update Unit — Sohi IEEE-TC 39:3 1990

Ordem de gravar resultado

- Resultados gravados em ordem — o que vimos até agora
  - instrução lenta segura outras que não dependem do seu resultado
  - facilita recuperação de excessões e implementação do SO
    mas desempenho é ruim
- resultados gravados fora-de-ordem
  - grava resultado quando não viola nenhuma dependência
  - o que acontece se instrução mais-nova grava resultado antes que excessão seja detectada em instrução mais-velha?
    . — Smith & Plezkun IEEE-TC 37:5 1988

Emissão em-ordem + resultado em-ordem

pipeline
- decodifica 2/ciclo
- executa 2/ciclo
- escreve 2/ciclo

i1 executa em 2 ciclos
i3 e i4 competem por UF
i5 depende de result de i4
i5 e i6 competem por UF

ciclo	decodif		execução			resultado	
1	i1	i2					
2	i3	i4	i1	i2			
3	i3	i4	i1				
4		i4			i3	i1	i2
5	i5	i6			i4		
6		i6		i5		i3	i4
7				i6			
8						i5	i6

Emissão em-ordem + resultado fora-de-ordem

pipeline
- decodifica 2/ciclo
- executa 3/ciclo
- escreve 2/ciclo

i1 executa em 2 ciclos
i3 e i4 competem por UF
i5 depende de result de i4
i5 e i6 competem por UF

ciclo	decodif		execução			resultado	
1	i1	i2					
2	i3	i4	i1	i2			
3		i4	i1		i3	i2	
4	i5	i6			i4	i1	i3
5		i6		i5		i4	
6				i6		i5	
7						i6	

Emissão fora-de-ordem + resultado fora-de-ordem

pipeline
- decodifica 2/ciclo
- executa 3/ciclo
- escreve 2/ciclo

i1 executa em 2 ciclos
i3 e i4 competem por UF
i5 depende de result de i4
i5 e i6 competem por UF

ciclo	decodif		janela	execução			resultado	
1	i1	i2						
2	i3	i4	i1,i2	i1	i2			
3	i5	i6	i3,i4	i1		i3	i2	
4			i4,i5,i6		i6	i4	i1	i3
5			i5		i5		i4	i6
6							i5	

Processador Super-escalar

Processador possui dois ou mais Circuitos de Dados paralelos

Estágio de decodificação escolhe tuplas de instruções
e as despacha de acordo com disponibilidade de unidades funcionais

Inter-travamento entre estágios e unidades funcionais
resolve dependências de dados e de controle

Algoritmo do Placar ou Algoritmo de Tomasulo: detalhes adiante
resolvem "problema dos gladiadores"
$\Longrightarrow$ que gladiador luta contra qual, e quando

Unidade de reordenamento enfilera resultados de acordo com
dependências entre resultados e operandos

Processador Superescalar

Dependências de controle são resolvidas com
execução especulativa

Instruções nos dois lados do desvio são
executadas especulativamente;
⟶ quando decide, anula efeito das instruções do caminho errado

Registradores fantasma mantém valores da execução especulativa;
quando resolve desvio copia de regs fantasma para regs visíveis

Resumo

- Superpipelining
  - aumenta freqüência do relógio & reduz lógica em cada estágio
  - ganho: desempenho $\propto$ freqüência do relógio
  - perda: CPI aumenta por causa das penalidades (desvios, depend)
- Processadores superescalares
  - recursos replicados para usar paralelismo no nível de instrução
  - algoritmo para garantir dependências de dados e controle
    algoritmo do placar ou de Tomasulo
  - PNI é característica do SW HW pode ou não suportar PNI>1
- Ordem modelo seqüencial deve ser respeitado pelo HW
  - emissão em-ordem VS fora-de-ordem
  - execução em-ordem VS fora de ordem
  - gravar resultados em-ordem VS fora-de-ordem

Revisão

- Superpipelining
  - aumenta freqüência do relógio & reduz lógica em cada estágio
  - ganho: desempenho $\propto$ freqüência do relógio
  - perda: CPI aumenta por causa das penalidades (desvios, depend)
- Processadores superescalares
  - recursos replicados para usar paralelismo no nível de instrução
  - algoritmo para garantir dependências de dados e controle
    algoritmo do placar ou de Tomasulo
  - PNI é característica do SW HW pode ou não suportar PNI>1
- Ordem modelo seqüencial deve ser respeitado pelo HW
  - emissão em-ordem VS fora-de-ordem
  - execução em-ordem VS fora de ordem
  - gravar resultados em-ordem VS fora-de-ordem

Trecho de código comum

```
for (i=0; i<TAM; i++)        // Multiplica elmtos de
      A[i] = B[i] * C[i];    // B e C, armazena em A

      addi r1,r0,TAM         # tamanho dos vetores
      or r2,r0,r0            # indice nos vetores
      la r4, endB            # ender vetor B
      la r5, endC            # ender vetor C
      la r6, endA            # ender vetor A
loop: lw r3, 0(r4+r2)        # carrega B[i]
      lw r7, 0(r5+r2)        # carrega C[i]
      mul r7,r7,r3           # r7 ←r7*r3
      addi r1,r1,-1          # decr contador
      sw r7, 0(r6+r2)        # guarda A[i]
      addi r2,r2,8           # acerta indice
      bne r1,r0,loop         # terminou?
```


Ordem de execução

```
      ...
      lw r3, 0(r4+r2)      # carrega B[i]
      lw r7, 0(r5+r2)      # carrega C[i]
      mul r7,r7,r3         # r7 ←r7*r3
      addi r1,r1,-1        # decr contador
      ...
```

- **Problema:**
  - dois LWs emitidos para unidade de memória
  - MUL é bloqueado por causa do risco RAW em r3 e r7
  - ADDI pára porque MUL parou
    ADDI pára por causa da execução em-ordem

Ordem de execução – soluções

- Escalonamento estático (sw)
  - compilador re-ordena as instruções
  - compilador pode usar algoritmos poderosos
  - hardware pode ser simples
  - escalonamento não se adapta à execução: faltas nas caches
- Escalonamento dinâmico (hw)
  - hardware re-ordena instruções
  - hw trata eventos desconhecidos em tempo de compilação
  - software é mais portável
  - hardware é (muito) mais complexo
  - é usado em muitos processadores super-escalares

Riscos em registradores

- Emissão fora-de-ordem não pode violar riscos RAW
  que são as únicas dependências verdadeiras
- riscos WAR e WAW podem inibir reordenação desnecessariamente
  dependências falsas:

```
lw r2, 8(r4)
mul r2,r2,r5
add r5,r6,r9
```

se r5 é modificado fora-de-ordem, MUL pode ler operando incorreto

Riscos em registradores

- Compiladores podem evitar alguns riscos WAR e WAW
  mas não todos eles

```
      add r2, r8, r4
      bne r1,r0, else
         add r2, r9, r4
else: add r6, r2, r5
```

  * não há risco WAW se desviar
    há risco WAW se seguir (em r2)
  * instrução pode ter risco consigo mesma
    no escalonamento dinâmico de loops

Re-nomeação de registradores em harwdare

- Considere registradores do CdI como nomes para valores
  não como locais de armazenagem
- hardware pode re-nomear registradores para evitar
  riscos WAR e WAW

```
de:
  lw r2, 8(r4)
  mul r2,r2,r5
  add r5,r6,r9
para:
  lw t1, 8(r4)
  mul t2,t1,r5
  add t3,r6,r9
```

Compilador deve garantir que usos subsequentes de r2 e r5
usem t1 e t3 ⟶ mesmo através de desvios

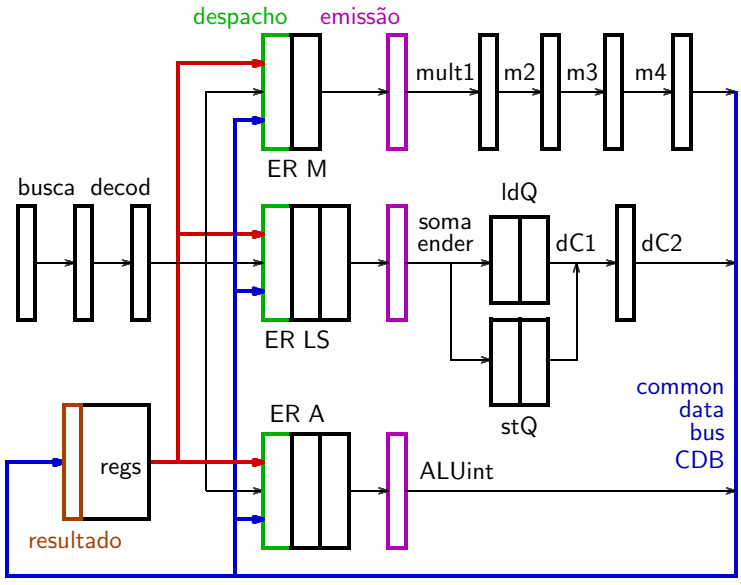
Algoritmo de Tomasulo

- usado no IBM 360/91, em 1967
  - ∗ modelo rápido para aplicações científicas
  - ∗ unidades de PF segmentadas
    somador e multiplicador (divisão efetuada no multiplicador)
- escalonamento dinâmico nas unidades de ponto flutuante

Algoritmo de Tomasulo Generalizado

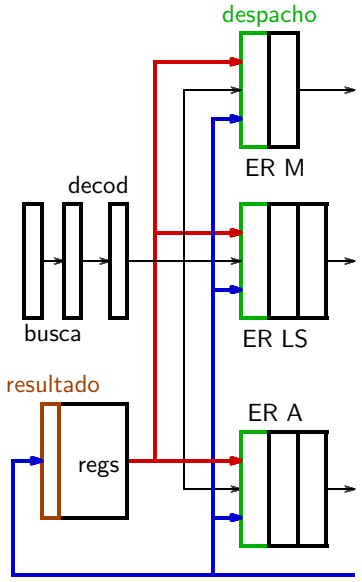
- Melhora paralelismo em processador segmentado
- aplica-se algoritmo em todas as unidades segmentadas
  ⟶ trata instruções de memória como as demais
- Usa etiquetas (tags) para identificar valores
  valores podem ser nomeados por tags e nomes de registradores
- estações de reserva (ER) implementam controle distribuído
  - ∗ uma ER por unidade funcional
  - ∗ tag identifica resultado da instrução na ER
- Common Data Bus (CDB) difunde todos resultados
  etiqueta da unidade funcional é transmitida junto com valor

Algoritmo de Tomasulo Generalizado



Algoritmo de Tomasulo – despacho

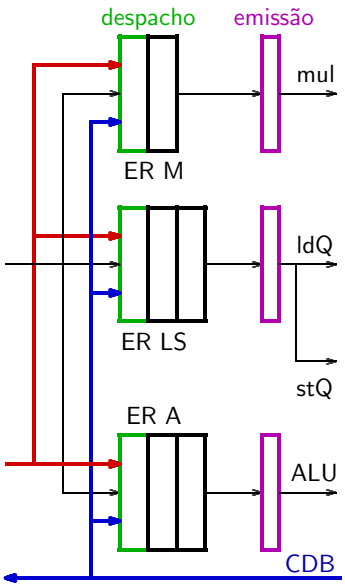
Despacho:
recebe próx instr de busca/decod
há estação de reserva livre?
não – bloqueia emissão
sim – despacha para emissão:
copia regs prontos para ER
copia tags dos regs
não-prontos para ER



Algoritmo de Tomasulo – emissão

Emissão:
se operandos estão prontos
emite operação para Unid Func
senão, espera operando ser
transmitido no CDB
(tag, valor)

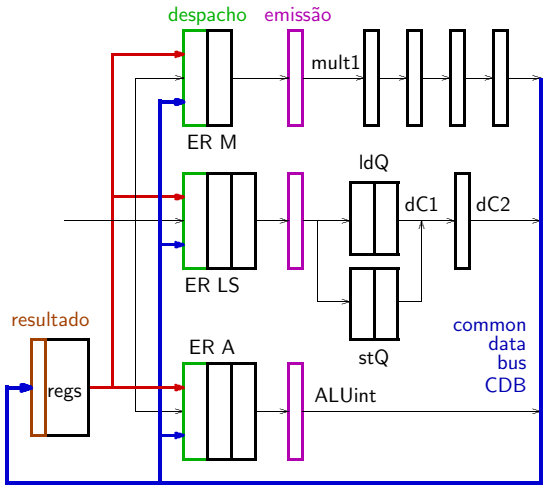
DESVIOS:
nenhuma instrução executa
até que desvios anteriores
sejam resolvidos
⟶ excessões precisas



Algoritmo de Tomasulo – finalização

Finalização:
se CDB está livre,
então transmite
resultado no CDB
senão, bloqueia
enquanto espera
para transmitir

ADIANTAMENTO:
CDB adianta valor
para ins dependentes
1 ciclo entre res⟶ uso



todas estações de reserva com tags iguais à tag no CDB (resultado) recebem valor

Algoritmo de Tomasulo – EdR

- Estações de Reserva
  - controlam a execução das instruções
  - detecção de riscos distribuída
  - tag de 4 bits identifica cada elemento das estações de reserva
- todos os receptores de valor usam tag para identificar dado que passa pelo CDB
- no despacho
  - ocorre mapeamento entre valores (resultados) e tags
  - isso feito, "nomes" de registradores são desnecessários

Algoritmo de Tomasulo – Estações de Reserva

- Premissa: tag=0 $\longrightarrow$ dado/valor disponível
- campos das estações de reserva

Op	opcode (operação)
Qj, Qk	etiquetas das fontes
	Qj | **Qk**=0 $\longrightarrow$ **Vj** | **Vk** contém valor pronto
Vj, Vk	valores dos operandos (fontes)
B	*(busy)* em uso
A	campo de endereço nas EdR da unidade de memória

- campos do bloco de registradores

Qi	etiqueta da unidade funcional que produzirá valor
	Qi=0 $\longrightarrow$ registrador contém valor pronto
Vi	valor/conteúdo do registrador

Loads e Stores

- Loads e Stores tratados como demais instruções
  - usam unidade de memória
  - endereços passam através de uma load queue
- **stores:** cálculo do ender efetivo é separado do acesso à memória
  - cálculo do ender efetivo despachado para ER
  - acesso à memória despachado para fila de escrita
  - unidade de memória envia endereços para fila de escrita
  - valor é associado ao endereço pela tag na fila de escrita
- resolução de riscos de memória
  - emissão de loads e stores ocorre em-ordem
  - compara ender dos loads com ender anteriores na fila de escrita
    bloqueia se há endereços repetidos RAW?
    loads podem ultrapassar stores se endereços diferem

Loads e Stores

store queue: stQ

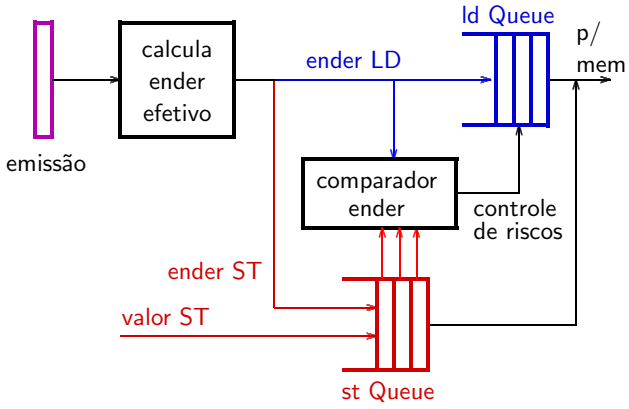
A endereço

Qi etiqueta de dados

Vi valor do operando

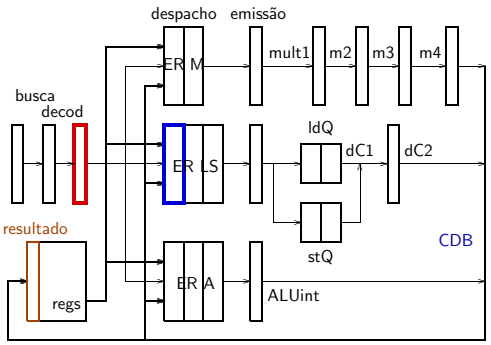
load queue: ldQ

A endereço



Exemplo

```
      addi r1,r0,TAM      # tamanho dos vetores
      or r2,r0,r0         # indice nos vetores
      la r4,endB          # ender vetor B 1000
      la r5,endC          # ender vetor C 2000
      la r6,endA          # ender vetor A 3000
loop: lw r3, 0(r4+r2)     # carrega B[i]
      lw r7, 0(r5+r2)     # carrega C[i]
      mul r7,r7,r3        # r7 ←r7*r3
      addi r1,r1,-1       # decr contador
      sw r7, 0(r6+r2)     # guarda A[i]
      addi r2,r2,8        # acerta indice
      bne r1,r0,loop      # terminou?
```

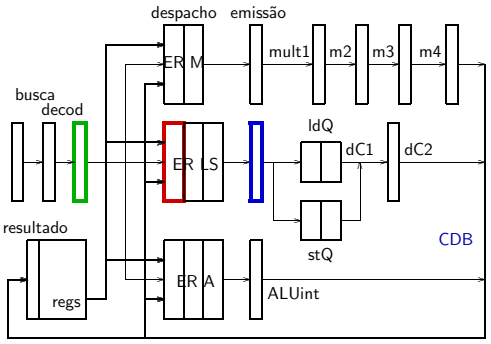

```
ciclo 1

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=?  V=?
```

reg	Q	V
0		
1		100
2		0
3	6	
4		1000
5		2000
6		3000
7		

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1						
A2	2						
A3	3						
M1	4						
M2	5						
LS1	6	1	ld	1000,0		0	0
LS2	7						
LS3	8						



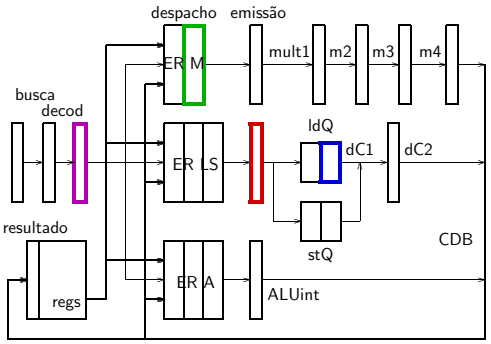

```
ciclo 2

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=?  V=?
```

reg	Q	V
0		
1		100
2		0
3	6	
4		1000
5		2000
6		3000
7	7	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1						
A2	2						
A3	3						
M1	4						
M2	5						
LS1	6	1	ld	1000,0		0	0
LS2	7	1	ld	2000,0		0	0
LS3	8						



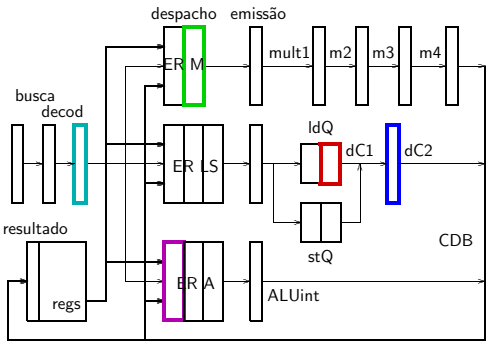

```
ciclo 3

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=?  V=?
```

reg	Q	V
0		
1		100
2		0
3	6	
4		1000
5		2000
6		3000
7	4	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1						
A2	2						
A3	3						
M1	4	1	mul			6	7
M2	5						
LS1	6	1	ld	1000+0		0	0
LS2	7	1	ld	2000,0		0	0
LS3	8						



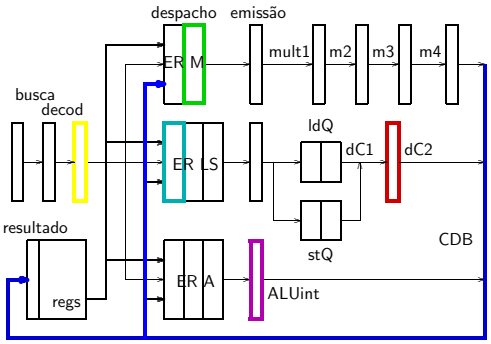

```
ciclo 4

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=?  V=?
```

reg	Q	V
0		
1	1	100
2		0
3	6	
4		1000
5		2000
6		3000
7	4	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	100	1	0	0
A2	2						
A3	3						
M1	4	1	mul			6	7
M2	5						
LS1	6	1	ld	1000+0		0	0
LS2	7	1	ld	2000+0		0	0
LS3	8						



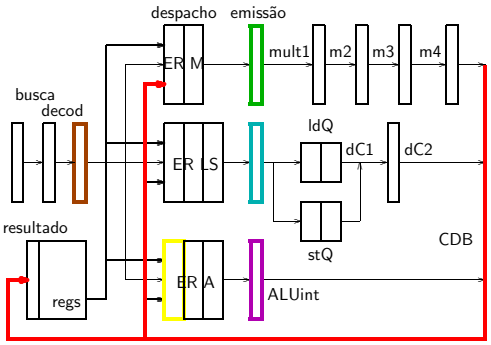

```
ciclo 5

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=4 V=?
```

reg	Q	V
0		
1	1	100
2		0
3	0	13
4		1000
5		2000
6		3000
7	4	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	100	1	0	0
A2	2						
A3	3						
M1	4	1	mul	13		0	7
M2	5						
LS1	6	0					
LS2	7	1	ld	2000+0		0	0
LS3	8	1	st	3000,0		0	4



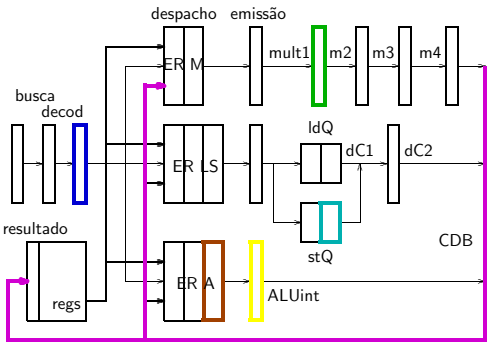

```
ciclo 6

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=?
```

reg	Q	V
0		
1	1	100
2	2	0
3	0	13
4		1000
5		2000
6		3000
7	4	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	100	1	0	0
A2	2	1	add	0	8	0	0
A3	3						
M1	4	1	mul	13	11	0	0
M2	5						
LS1	6	0					
LS2	7	0					
LS3	8	1	st	3000,0		0	4



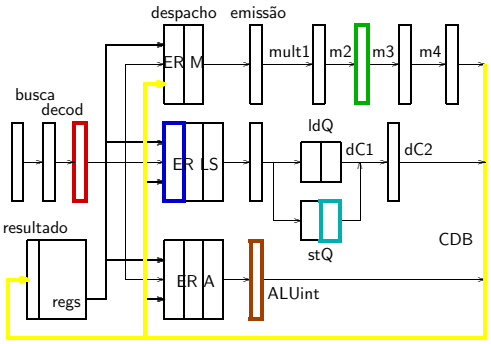

```
ciclo 7

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=?
```

reg	Q	V
0		
1	0	99
2	2	0
3	0	13
4		1000
5		2000
6		3000
7	4	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	0					
A2	2	1	add	0	8	0	0
A3	3	1	bne	99	0	0	0
M1	4	1	mul	13	11	0	0
M2	5						
LS1	6	0					
LS2	7	0					
LS3	8	1	st	3000+0		0	4



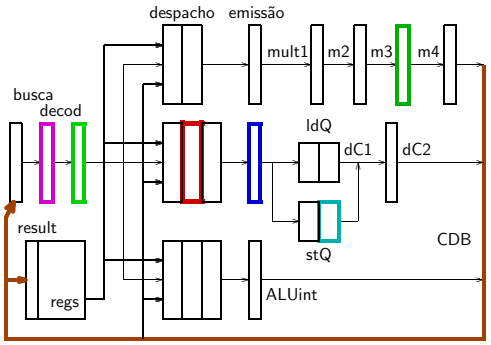

```
ciclo 8

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=?
```

reg	Q	V
0		
1	0	99
2	0	8
3	6	13
4		1000
5		2000
6		3000
7	4	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	0					
A2	2	0					
A3	3	1	bne	99	0	0	0
M1	4	1	mul	13	11	0	0
M2	5						
LS1	6	1	ld	1000,8		0	0
LS2	7	0					
LS3	8	1	st	3000+0		0	4



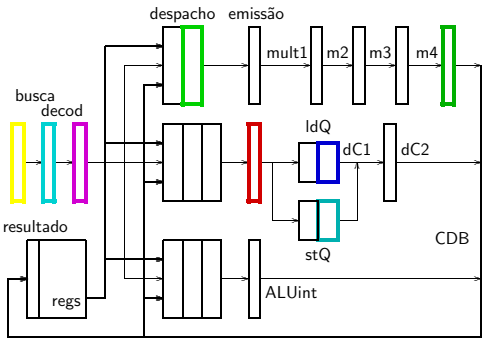

```
ciclo 9

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=?
```

reg	Q	V
0		
1	0	99
2	0	8
3	6	13
4		1000
5		2000
6		3000
7	7	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	0					
A2	2	0					
A3	3	0					
M1	4	1	mul	13	11	0	0
M2	5						
LS1	6	1	ld	1000,8		0	0
LS2	7	1	ld	2000,8		0	0
LS3	8	1	st	3000+0		0	4



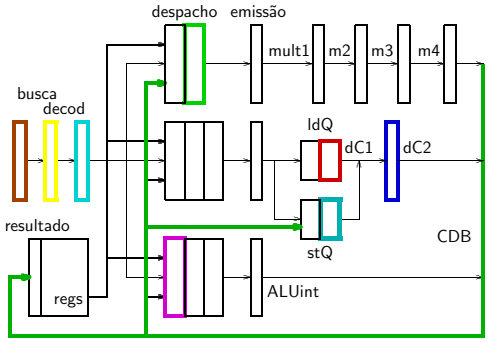

```
ciclo 10

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3 2x
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=?
```

reg	Q	V
0		
1	0	99
2	0	8
3	6	13
4		1000
5		2000
6		3000
7	5	

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	0					
A2	2	0					
A3	3	0					
M1	4	1	mul	13	11	0	0
M2	5	1	mul			7	6
LS1	6	1	ld	1000+8		0	0
LS2	7	1	ld	2000,8		0	0
LS3	8	1	st	3000+0		0	4



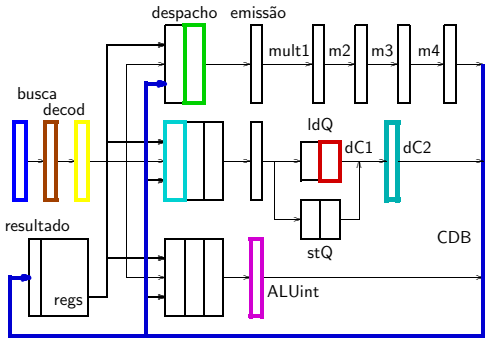

```
ciclo 11

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=143
```

reg	Q	V
0		
1	1	99
2	0	8
3	6	13
4		1000
5		2000
6		3000
7	5	143

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	99	1	0	0
A2	2	0					
A3	3	0					
M1	4	0					
M2	5	1	mul			7	6
LS1	6	1	ld	1000+8		0	0
LS2	7	1	ld	2000+8		0	0
LS3	8	1	st	3000+0	143	0	0



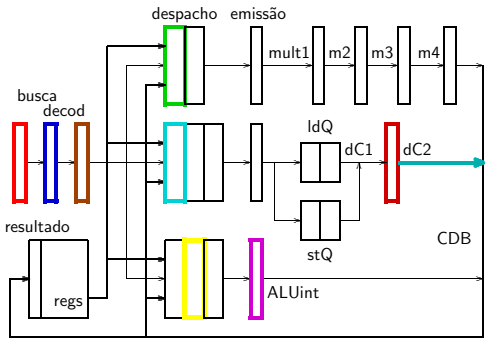

```
ciclo 12

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=3000 Q=4 V=143
```

reg	Q	V
0		
1	1	99
2	0	8
3	0	12
4		1000
5		2000
6		3000
7	5	143

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	99	1	0	0
A2	2	0					
A3	3	0					
M1	4	0					
M2	5	1	mul	12		0	7
LS1	6	1	st	3000,8		0	5
LS2	7	1	ld	2000+8		0	0
LS3	8	1	st	3000+0	143	0	0



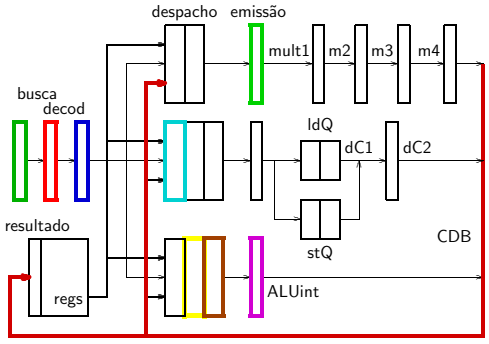

```
ciclo 13

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=?  V=?
```

reg	Q	V
0		
1	1	99
2	2	8
3	0	12
4		1000
5		2000
6		3000
7	5	143

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	99	1	0	0
A2	2	1	add	8	8	0	0
A3	3	0					
M1	4	0					
M2	5	1	mul	12		0	7
LS1	6	1	st	3000,8		0	5
LS2	7	1	ld	2000+8		0	0
LS3	8	0					




```
ciclo 14

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=?  Q=?  V=?
```

reg	Q	V
0		
1	1	99
2	2	8
3	0	12
4		1000
5		2000
6		3000
7	5	143

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1	1	sub	99	1	0	0
A2	2	1	add	8	8	0	0
A3	3	1	bne		0	1	0
M1	4	0					
M2	5	1	mul	12	10	0	0
LS1	6	1	st	3000,8		0	5
LS2	7	0					
LS3	8	0					

Algoritmo de Tomasulo – resumo

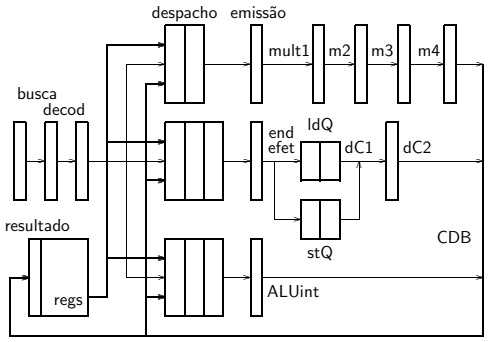
- Estações de Reserva
  - emissão fora-de-ordem quando operandos disponíveis
- Re-nomeação de registradores
  - elimina riscos WAW e WAR importante com poucos regs
  - permite desenrolar laços dinamicamente
  - lógica é relativamente complexa
- Common Data Bus
  - transmite resultados para múltiplos destinos
  - gargalo central pode ser duplicado, com mais hw
- excessões precisas – não implementadas
  - poderia usar buffer de re-ordenação para atualizar registradores

Algoritmo de Tomasulo – tratamento de riscos

PROBLEMA	SOLUÇÃO
risco estr: Estação de Reserva	bloqueia no despacho
risco estr: unidade funcional	bloqueia na ER
RAW	resolve com as etiquetas
WAR	copia oper para ER no despacho
WAW	renomeação de registradores

Exercícios

1. Re-escreva o código do laço para minimizar as bolhas.
2. Desenrole o loop para minimizar as bolhas. Qual o ganho com relação ao exercício 1?
3. Considerando a latência da multiplicação (4 ciclos), quantas vezes o loop deve ser desenrolado para minimizar as bolhas?
4. Complete a seqüência de diagramas após o ciclo 9, em que o destino do desvio é decidido (veja diagrama do ciclo 10). Simule mais uma volta do loop.




```
ciclo

loop: r3 ←mem(r4+r2)
      r7 ←mem(r5+r2)
      r7 ←r7 * r3
      r1 ←r1 - 1
      mem(r6+r2) ←r7
      r2 ←r2 + 8
      (r1!=0)?PC←loop

stQ: A=     Q=   V=
```

reg	Q	V
0		
1		
2		
3		
4		1000
5		2000
6		3000
7		

RS	tag	bsy	Op	Vj	Vk	Qj	Qk
A1	1						
A2	2						
A3	3						
M1	4						
M2	5						
LS1	6						
LS2	7						
LS3	8						

revisão – Algoritmo de Tomasulo

- Estações de Reserva
  - ∗ emissão fora-de-ordem quando operandos disponíveis
  - ∗ emissão continua adiante dos desvios/jumps
- Re-nomeação de registradores elimina riscos WAW e WAR
  - ∗ permite desenrolar laços dinamicamente
    múltiplas iterações usam lugares ≠s para valores/regs
- Common Data Bus transmite resultados para múltiplos destinos
- Lógica de detecção de riscos é distribuída pelas ER e CDB
  - ∗ se múltiplas instruções esperam por um mesmo resultado, com outro já disponível, então valor no CDB libera todas p/ execução
- escalonamento é dinâmico porque grafo de dependências é construído durante execução

Superescalar + Fora-de-Ordem + Especulação

- Superescalar + Fora-de-Ordem + Especulação
  conceitos que funcionam bem (melhor) quando combinados
- CPI ≥ 1
  consegue com superescalar
- superescalar sofre mais riscos? por que?
  resolve com escalonamento dinâmico (exec/reslt fora-de-ordem)
- dependências RAW causam problemas?
  resolve com janela de instruções grande
- desvios impedem que janela fique cheia?
  resolve com especulação

Excessões precisas e especulação

- **Especulação** – adivinha futuro e executa tentativamente
- importante na previsão de desvios:
  prevê destino e inicia execução naquele caminho
- se especulação foi errada,
  deve voltar e re-iniciar execução de onde previu erradamente
  mesmo comportamento que em excessões precisas
- mesma técnica para excessão precisa e especulação:
  completar em-ordem *in-order commit*

Especulação e excessões precisas

- Excessões tem semântica seqüencial von Neumann
  * todas instruções antes da excessão completam
  * todas instrs após excessão devem parecer nunca ter iniciado
  * mesmas condições que para um desvio previsto erradamente
- qual a dificuldade com excessões/interrupções precisas?
  * completar fora-de-ordem $\rightsquigarrow$ deve desfazer escritas anteriores
  * em-ordem $\rightsquigarrow$ nenhuma escrita posterior à instr de desvio,
    é confirmada até que desvio complete
  * fora-de-ordem pode ocorrer
- fora-de-ordem:
  excessões precisas e recuperação de especulação errada
  são o mesmo problema $\longrightarrow$ mesma solução

Algoritmo de Tomasulo e excessões precisas

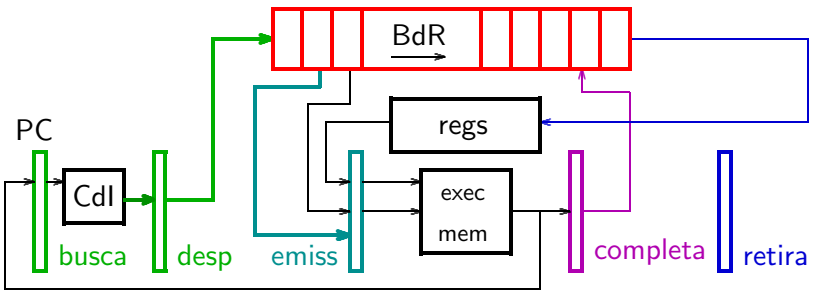
- Algoritmo de Tomasulo faz:
  * emissão em-ordem
  * execução fora-de-ordem
  * resultados fora-de-ordem
- necessário resolver ordenação dos resultados para que excessões sejam precisas
  * "ponto de quebra" definido:
    instruções mais velhas completaram (escreverem resultado)
    instruções mais novas não alteram estado
- habilidade de abortar e re-iniciar cada instrução
  ⟶ estado preciso

Qual o problema com estado preciso?

- problema no estágio de escrita/resultado *writeback*
  mistura o que deveria estar separado:
  * difunde valores para Estações de Reserva, e
    adianta valores para instruções ⟶ pode ser fora-de-ordem
  * escreve valores nos registradores ⟶ melhor se é em-ordem

- **P:** qual a solução para todo problema de funcionalidade?
  R: adicionar um nível de indireção
  * como é feito para execução fora-de-ordem
    separa DECOD em
    (1) DESPACHO em-ordem e (2) EMISSÃO fora-de-ordem
  * usa um buffer de instruções para separar os dois estágios
    placar ou estações de reserva

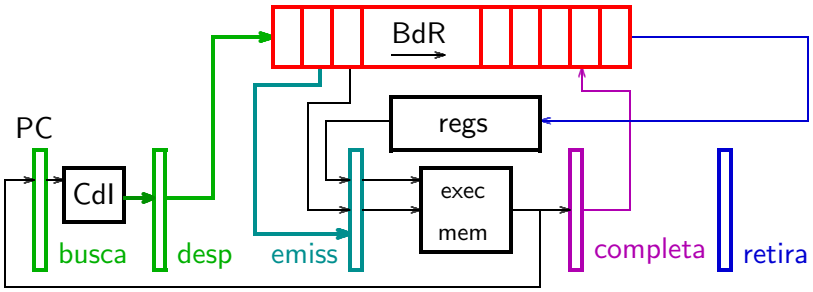
Buffer de Re-ordenação (BdR)



buffer de instruções ≡ Buffer de Reordenação

- mantém resultados prontos a caminho dos regs e da cache de dados
  pode ser combinado com as ERs (como na figura) ou separado
- divide estágio de RESULTADO em COMPLETA e RETIRA

Completa e Retira



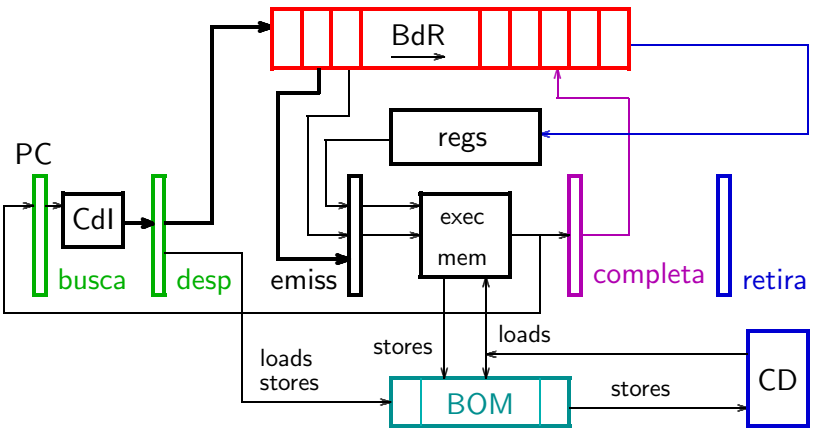
- **completa (CM)** *commit*
  - ⋆ valores prontos são gravados no BdR fora-de-ordem
  - ⋆ estágio fora-de-ordem ⟶ riscos resultam em **espera** (wait)
- **retira (RT)** *retire, graduate*
  - ⋆ BdR escreve no bloco de registradores em-ordem
  - ⋆ estágio em-ordem ⟶ riscos resultam em **bloqueios** (stalls)

Buffer de Ordenação de Memória

BdR escreve nos regs em-ordem – como fica escrever em memória?

- mesmo que antes ⟶ escreve na cache no estágio de memória
  - ∗ **ruim!!!** memória imprecisa é pior que regs imprecisos
- Buffer de ordenação de memória (BOM)
  - ∗ fila de escrita, *store buffer*, *load/store queue*
  - ∗ STOREs em COMP (mas não retirados) escrevem no BOM
  - ∗ em RET, STORE na cabeça do BOM é escrito em memória
  - ∗ LOADs procuram no BOM e na cache em paralelo
    adiantamento do BOM se referencia mesmo endereço

BdR + BOM



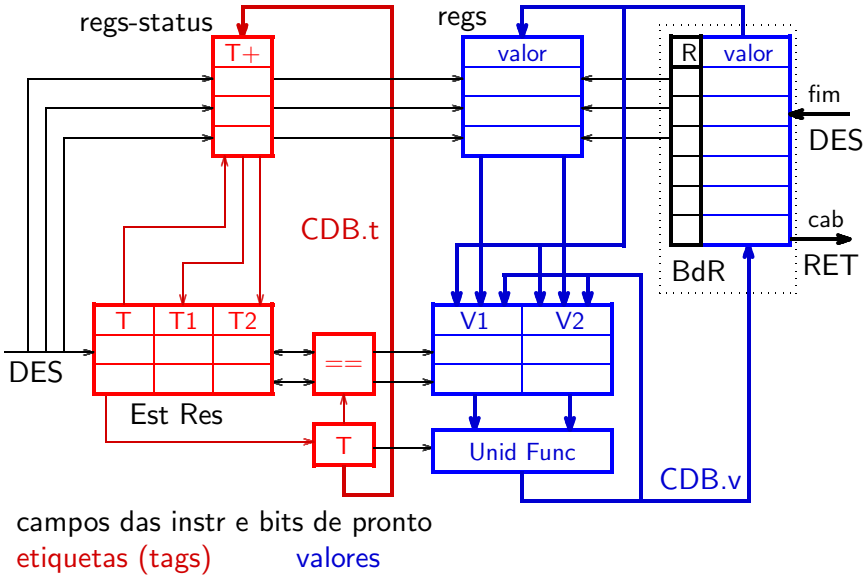
Esta é uma versão simplificada, porém realista, do PentiumPro (P6)

Tomasulo + BdR

- adicionar BdR ao algoritmo de Tomasulo
  - combinação de BdR e ER é chamada de *Register Update Unit* (RUU) ou "método de Sohi"
  - BdR e ER separadas são chamados de *estilo P6*
- Exemplo: P6-simples
  - BdR e ER separados
  - organização: 1 ALU, 1 load, 1 store, 2 PF (latência de 3 ciclos)

```
loop: ldf f0,X(r1)          for (i=0; i<r2; i++)
      mulf f4,f0,f2            Z[i] = X[i] * f2
      stf f4,Z(r1)
      add r1,r1,#8
      bne r1,r2,loop        (ignorar desvio)
```


Organização no Estilo P6



campos das instr e bits de pronto
etiquetas (tags) valores

Estruturas de dados

- Estações como em Tomasulo simples
- Buffer de Reordenação (BdR)
  - **cabeça, fim** – mantém ordem seqüencial
  - **R** – registrador destino da instr
  - **V** – valor produzido pela instrução
- etiqueta
  - #BdR (elmto do BdR) (Tomasulo usava #ER)
- status dos registradores
  - **T+** – bit **+** indica "pronto no BdR"
  - tag $= 0 \longrightarrow$ resultado pronto em registrador (preciso)
  - tag $\neq 0 \longrightarrow$ resultado não está pronto
  - tag $=$ N+ $\longrightarrow$ resultado pronto no BdR #N

Estruturas de dados

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)							f0	
	2	mulf f4,f0,f2							f2	
	3	stf f4,Z(r1)							f4	
	4	add r1,r1,8							r1	
	5	ldf f0,X(r1)							CDB	
	6	mulf f4,f0,f2							T	V
	7	stf f4,Z(r1)								

Est Resv

#	UF	bsy	T	Vj	Vk	Qj	Qk
1	ALU						
2	load						
3	store						
4	FP1						
5	FP2						

...

Segmentos do P6

Estrutura do processador: BUS, DES, EMI, EXE, COM, RET
Busca, Despacho, Emissão, Exec, Completa, Retira

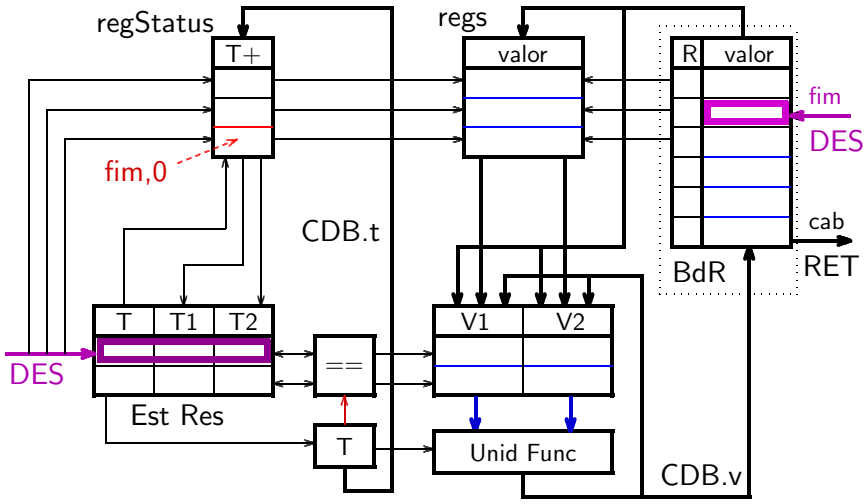
- DES (despacho)
  (ER, BdR, BOM, cheios) ? (bloqueia) :
  { aloca elementos em ER, BdR, BOM,
  atribui #BdR à etiqueta da ER
  atribui #BdR ao status do reg destino, bit + desligado
  copia regs/valores prontos para ER
  }
- EXE (execução)
  libera elemento da ER
  (em Tomasulo fazia na finalização,
  pode fazer em EXE porque #ER não é mais etiqueta)

Segmentos do P6

Estrutura do processador: BUS, DES, EMI, EXE, COM, RET

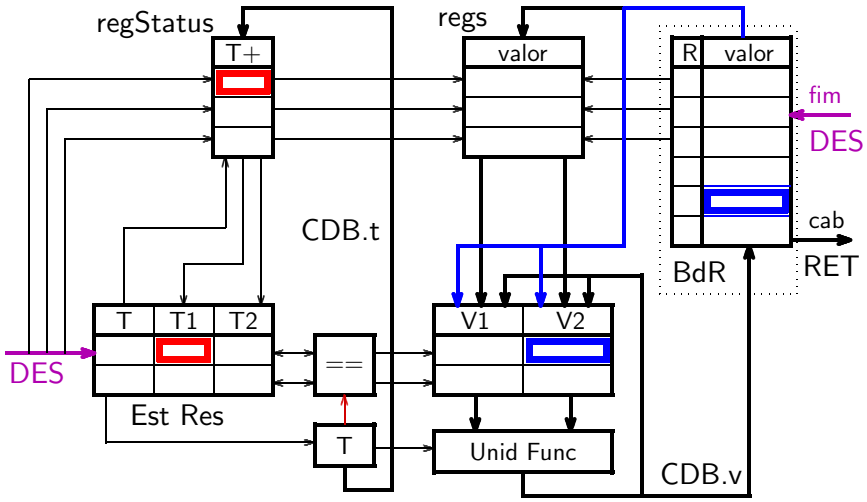
- COM (completa)
  (CDB ocupado) ? (espera) : *wait*
  { escreve valor no elmto do BdR indicado pela etiqueta na ER,
  marca elmto do BdR como pronto,
  liga bit + (pronto-no-BdR) no status do reg destino
  }
- RET (retira)
  (cabeça do BdR não está pronto) ? (bloqueia) : *stall*
  { escreve valor da cabeça do BdR no bloco de regs,
  se STORE escreve cabeça do BOM na cache,
  trata quaisquer excessões,
  libera elementos do BdR e BOM
  }

P6 DESpacha (i)



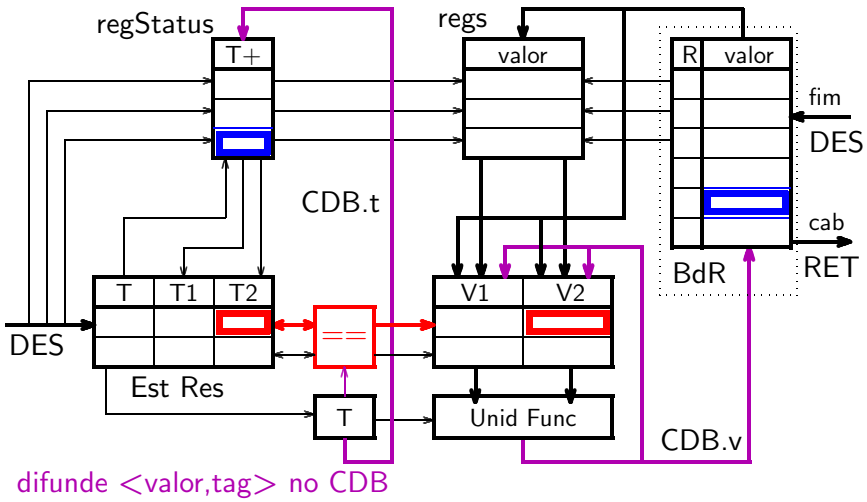
aloca elmtos no BdR e ER (etiqueta dest da ER = #BdR)
regStatus aponta para #BdR, bit pronto-no-BdR=0 (+)

P6 DESpacha (ii)



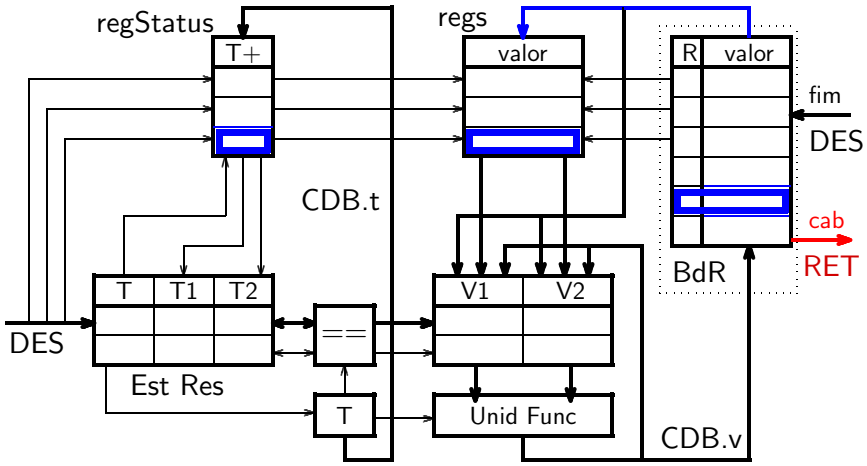
verifica tags de operandos do regStatus (tag==0: copia dos regs)
tag !=0: copia tag p/ ER tag ==n+: copia tag do BdR[n]

P6 COMpleta



difunde <valor,tag> no CDB
escreve result no BdR, liga bit + em regStatus
'casa' tags, grava CDB.v nas ER das instr dependentes

P6 RETira



bloqueia até que instr na cabeça do BdR tenha completado

escreve valor da cabeça do BdR nos regs (na cache se STORE),

limpa elmto de regStatus libera elmto do BdR

Exemplo P6 – ciclo 1

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
h,t	1	ldf f0,X(r1)	f0		&X[0]				f0	B-1
	2	mulf f4,f0,f2							f2	
	3	stf f4,Z(r1)							f4	
	4	add r1,r1,8							r1	
	5	ldf f0,X(r1)							CDB	
	6	mulf f4,f0,f2							T	V
	7	stf f4,Z(r1)								

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU							
2	load	1	ldf	B-1		[r1]		
3	store							
4	FP1							
5	FP2							

aloca BdR[1]

aloca EstRes[2]

ajusta regStat[f0]

Exemplo P6 – ciclo 2

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
h	1	ldf f0,X(r1)	f0		&X[0]	c2			f0	B-1
t	2	mulf f4,f0,f2	f4						f2	
	3	stf f4,Z(r1)							f4	B-2
	4	add r1,r1,8							r1	
	5	ldf f0,X(r1)							CDB	
	6	mulf f4,f0,f2							T	V
	7	stf f4,Z(r1)								

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU							
2	load	1	ldf	B-1		[r1]		
3	store							
4	FP1	1	mulf	B-2		R[f2]	B-1	
5	FP2							

aloca BdR[2]

aloca EstRes[4]

ajusta regStat[f4]

Exemplo P6 – ciclo 3

hd	BdReord + BOrdMem							
tl	#	instr	R	V	ender	EM	EX	CM
h	1	ldf f0,X(r1)	f0		&X[0]	c2	c3	
	2	mulf f4,f0,f2	f4					
t	3	stf f4,Z(r1)			&Z[0]			
	4	add r1,r1,8						
	5	ldf f0,X(r1)						
	6	mulf f4,f0,f2						
	7	stf f4,Z(r1)						

rStat	
reg	T+
f0	B-1
f2	
f4	B-2
r1	

CDB	
T	V

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU							
2	load	0	—	—		*ldf em EXEC*		
3	store	1	stf	B-3		R[r1]	B-2	
4	FP1	1	mulf	B-2		R[f2]	B-1	
5	FP2							

aloca BdR[3]

aloca ER[3]

libera EstRes[2]

Exemplo P6 – ciclo 4

hd	BdReord + BOrdMem							
tl	#	instr	R	V	ender	EM	EX	CM
h	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4
	2	mulf f4,f0,f2	f4			c4		
	3	stf f4,Z(r1)			&Z[0]			
t	4	add r1,r1,8	r1					
	5	ldf f0,X(r1)						
	6	mulf f4,f0,f2						
	7	stf f4,Z(r1)						

rStat	
reg	T+
f0	B-1+
f2	
f4	B-2
r1	B-4

CDB	
T	V
B-1	[f0]

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	1	add	B-4	R[r1]	8		
2	load	0						
3	store	1	stf	B-3		R[r1]	B-2	
4	FP1	1	mulf	B-2	CDB.V	R[f2]	*b-1*	
5	FP2							

aloca BdR[4]
aloca ER[1]

ldf completa,
[f0] passa no CdB,
marca f0 pronto no BdR: B-1+

Exemplo P6 – ciclo 5

hd	BdReord + BOrdMem							
tl	#	instr	R	V	ender	EM	EX	CM
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4
h	2	mulf f4,f0,f2	f4			c4	c5,1	
	3	stf f4,Z(r1)			&Z[0]			
	4	add r1,r1,8	r1			c5		
t	5	ldf f0,X(r1)	f0					
	6	mulf f4,f0,f2						
	7	stf f4,Z(r1)						

rStat	
reg	T+
f0	B-5
f2	
f4	B-2
r1	B-4

CDB	
T	V

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	1	add	B-4	R[r1]	8		
2	load	1	ldf	b-5				B-4
3	store	1	stf	B-3		R[r1]	B-2	
4	FP1	0	—	—		*mulf em EXEC*		
5	FP2							

aloca BdR[5]
aloca ER[2]

libera ER[4]

ldf retira,
grava [f0] em regs

Exemplo P6 – ciclo 6

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5
h	2	mulf f4,f0,f2	f4			c4	c5,2		f2	
	3	stf f4,Z(r1)			&Z[0]				f4	B-6
	4	add r1,r1,8	r1			c5	c6		r1	B-4
	5	ldf f0,X(r1)	f0						CDB	
t	6	mulf f4,f0,f2	f4						T	V
	7	stf f4,Z(r1)								

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0	—	—		*add em EXEC*		
2	load	1	ldf	B-5				B-4
3	store	1	stf	B-3		R[r1]	B-2	
4	FP1	0						
5	FP2	1	mulf	B-6		R[f2]	B-5	

aloca BdR[6]

aloca ER[5]

libera ER[1]

Exemplo P6 – ciclo 7

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5
h	2	mulf f4,f0,f2	f4			c4	c5,3		f2	
	3	stf f4,Z(r1)			&Z[0]				f4	B-6
	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0		&X[1]	c7			CDB	
t	6	mulf f4,f0,f2	f4						T	V
	7	stf f4,Z(r1)		∄ ER livre					B-4	[r1]

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	1	ldf	B-5		CDB.V		*b-4*
3	store	1	stf	B-3		R[r1]	B-2	
4	FP1	0						
5	FP2	1	mulf	B-6		R[f2]	B-5	

add completa
[r1] passa no CDB
marca r1 pronto no BdR: B-4+

bloqueia DESPacho porque ∄ ER livre para stf

Exemplo P6 – ciclo 8

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5
h	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
	3	stf f4,Z(r1)			&Z[0]	c8			f4	B-6
	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0		&X[1]	c7	c8		CDB	
t	6	mulf f4,f0,f2	f4						T	V
	7	stf f4,Z(r1)		∄ ER livre					B-2	[f4]

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0	—	—		*ldf em EXEC*		
3	store	1	stf	B-3	CDB.V	R[r1]	*b-2*	
4	FP1	0						
5	FP2	1	mulf	B-6		R[f2]	B-5	

mulf completa
[f4] passa no CDB

bloqueia DESP: ∄ ER livre para stf

bloqueia add em RETira: retira em-ordem

Exemplo P6 – ciclo 9

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5+
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
h	3	stf f4,Z(r1)			&Z[0]	c8	c9		f4	B-6
	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0	[f0]	&X[1]	c7	c8	c9	CDB	
	6	mulf f4,f0,f2	f4			c9			T	V
t	7	stf f4,Z(r1)			&Z[1]				B-5	[f0]

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	1	stf	B-7		B-4.v	B-6	
4	FP1	0						
5	FP2	1	mulf	B-6	CDB.V	R[f2]	*b-5*	

ldf completa
[f0] passa no CDB

libera BdR[3]
. (stf em EX)
e aloca BdR[7]

bloqueia add:
RETira em-ordem

Exemplo P6 – ciclo 10

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5+
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
h	3	stf f4,Z(r1)			&Z[0]	c8	c9	c10	f4	B-6
	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0	[f0]	&X[1]	c7	c8	c9	CDB	
	6	mulf f4,f0,f2	f4			c9	c10,1		T	V
t	7	stf f4,Z(r1)			&Z[1]					

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	1	stf	B-7		B-4.v	B-6	
4	FP1	0						
5	FP2	0	—	—		*mulf em EXEC*		

libera ER[5]

bloqueia add
bloqueia ldf
RETira em-ordem

Exemplo P6 – ciclo 11

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5+
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
	3	stf f4,Z(r1)			&Z[0]	c8	c9	c10	f4	B-6
h	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0	[f0]	&X[1]	c7	c8	c9	CDB	
	6	mulf f4,f0,f2	f4			c9	c10,2		T	V
t	7	stf f4,Z(r1)			&Z[1]					

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	1	stf	B-7		B-4.v	B-6	
4	FP1	0						
5	FP2	0						

retira stf
e libera BdR[3]

bloqueia add
bloqueia ldf
RETira em-ordem

Exemplo P6 – ciclo 12

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5+
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
	3	stf f4,Z(r1)			&Z[0]	c8	c9	c10	f4	B-6
h	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0	[f0]	&X[1]	c7	c8	c9	CDB	
	6	mulf f4,f0,f2	f4			c9	c10,3		T	V
t	7	stf f4,Z(r1)			&Z[1]					

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	1	stf	B-7		B-4.v	B-6	
4	FP1	0						
5	FP2	0						

retira add
EM-ORDEM
(bloq desde c7)

e libera BdR[4]

bloqueia ldf
RETira em-ordem

Estado preciso com BdR

- **Razão de ser do BdR é manter estado preciso**
- **como?**

1. espera até que condição precisa chegue ao estágio onde RETira
2. limpa conteúdo do BdR, ER, e regStatus *enche de zeros*
3. re-inicia

- funciona porque zeros são o estado correto para re-iniciar
  - **0** em BdR/ER ⟶ elementos estão vazios
  - tag=0 na tabela de status dos regs ⟶ registrador atualizado
- e porque escritas nos registradores e na cache ocorrem em RETira
- exemplo: falta de página em STORE

Exemplo de Estado Preciso no P6 – ciclo 9

hd	BdReord + BOrdMem								rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	B-5+
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
h	3	stf f4,Z(r1)			&Z[0]	c8	c9		f4	B-6
	4	add r1,r1,8	r1	[r1]		c5	c6	c7	r1	B-4+
	5	ldf f0,X(r1)	f0	[f0]	&X[1]	c7	c8	c9	CDB	
	6	mulf f4,f0,f2	f4			c9			T	V
t	7	stf f4,Z(r1)			&Z[1]				B-5	[f0]

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	1	stf	B-7		B-4.v	B-6	
4	FP1	0						
5	FP2	1	mulf	B-6	CDB.V	R[f2]	*b-5*	

FALTA de PÁGINA
em stf em c9

Exemplo de Estado Preciso no P6 – ciclo 10

hd		BdReord + BOrdMem							rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
h,t	3	stf f4,Z(r1)			&Z[0]	c8	c9	c10	f4	
	4	add r1,r1,8							r1	
	5	ldf f0,X(r1)							CDB	
	6	mulf f4,f0,f2							T	V
	7	stf f4,Z(r1)								

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	0						
4	FP1	0						
5	FP2	0						

instrução com excessão (stf) chega à cabeça do BdR
LIMPA TUDO!

Exemplo de Estado Preciso no P6 – ciclo 11

hd		BdReord + BOrdMem							rStat	
tl	#	instr	R	V	ender	EM	EX	CM	reg	T+
	1	ldf f0,X(r1)	f0	[f0]	&X[0]	c2	c3	c4	f0	
	2	mulf f4,f0,f2	f4	[f4]		c4	c5	c8	f2	
h,t	3	stf f4,Z(r1)			&Z[0]				f4	
	4	add r1,r1,8							r1	
	5	ldf f0,X(r1)							CDB	
	6	mulf f4,f0,f2							T	V
	7	stf f4,Z(r1)								

Est Res

#	UF	bsy	oper	T	Vj	Vk	Qj	Qk
1	ALU	0						
2	load	0						
3	store	1	stf	B-3	R[f4]	R[r1]		
4	FP1	0						
5	FP2	0						

re-inicia de onde parou

Desempenho do Estilo P6

Qual é o custo de excessões precisas?

- em geral, mesmo desempenho que Algoritmo de Tomasulo
  - ∗ pouco melhor porque libera ER antes (EXEC)
    e portanto incorre em menos riscos estruturais
- se BdR não for suficientemente grande, desempenho ruim
  - ∗ BdR causa riscos estruturais
- regras de projeto para BdR N=largura
  - ▷ |BdR| $\geq$ (N × núm de estágios entre DESP e RET)
  - ▷ |BdR| $\geq$ (N × T acerto na L2)
  - ▷ qual o porquê destas regras?

Problemas com P6

- Organização foi popular (início 90s)
  - ∗ relativamente fácil de implementar corretamente
  - ∗ para recuperar estado, zera tudo
  - ∗ exemplos: PentiumPro, AMD K6, PowerPC
- Problemas com implementação de alto desempenho
  - ▷ excessiva movimentação de valores:
    regs/BdR⟶ ER⟶ BdR⟶ regs
  - ▷ muitos barramentos e muxes (valores de/para muitos lugares)
  - ▷ ER misturam valores com etiquetas (controle)
    caminhos ficam longos e isso baixa freqüência do relógio

Solução: implementação do MIPS R10K

- Separa controle (BdR/ER) dos dados (regs/Unid Funcionais)
  - ▷ um bloco de regs grande mantém todos os dados ⟶ sem cópias
  - ▷ BdR e ER usados somente para controle e etiquetas ⟶ pequenos
  - ▷ bloco de registradores próximo das UFs, todo o resto fica ao lado

Detalhes em K C Yeager, *The MIPS R10000 Superscalar Microprocessor*, IEEE Micro 16:2, Abr1996.

resumo – Especulação e Excessões Precisas

- Excessões tem semântica seqüencial von Neumann
  - ∗ todas instruções antes da excessão completam
  - ∗ todas instrs após excessão devem parecer nunca ter iniciado
  - ∗ mesmas condições que para um desvio previsto erradamente
- qual a dificuldade com excessões/interrupções precisas?
  - ∗ completar fora-de-ordem ⇝ deve desfazer escritas anteriores
- implementação: Tomasulo com Buffer de Reordenação Estilo P6
  - ▷ instruções completam mas só escrevem resultado nos registradores
    se ∄ excessão
  - ▷ se há excessão (ou previsão de desvio errada)
    zera ER, BdR, e tabela de status dos regs, e
    re-inicia de onde parou

revisão – Especulação e Excessões Precisas

- Excessões tem semântica seqüencial von Neumann
  - todas instruções antes da excessão completam
  - todas instrs após excessão devem parecer nunca ter iniciado
  - mesmas condições que para um desvio previsto erradamente
- qual a dificuldade com excessões/interrupções precisas?
  - completar fora-de-ordem $\rightsquigarrow$ desfazer escritas anteriores
- implementação I: Tomasulo + Buffer de Reordenação Estilo P6
  - ▷ instruções completam mas só escrevem resultado nos registradores
    se $\nexists$ excessão
  - ▷ se ($\exists$ excessão || previsão_desvio_errada)
    zera ER, BdR, e tabela statusRegs; re-inicia de onde parou
- implementação II: MIPS R10K
  separa controle (BdR/ER) dos dados (regs/Unid Funcionais)

Trilha do desempenho

- segmentação simples: emissão em-ordem, largura=1
- primeira extensão: super-escalar, emissão múltipla, largura>1
- segunda extensão: escalonamento de instruções para > PNI
  - $\mathbf{1}^{\underline{\mathbf{a}}}$ opção: escalonamento dinâmico
  - $\mathbf{2}^{\underline{\mathbf{a}}}$ opção: escalonamento estático

Very Long Instruction Word – VLIW

Problemas com implementação super-escalar

▷ largura de busca + previsão de desvios
▷ adiantamento $\propto \boldsymbol{N^2}$
▷ avaliação de dependências $\propto \boldsymbol{N^2}$

Alternativa: **Very Long instruction Word – VLIW**
$\longrightarrow$ pipeline com emissão simples de grupo com $\boldsymbol{N}$ instruções (VLIW)

▷ compilador garante independência das instrns numa VLIW
▷ processador não testa dependências "dentro" de uma VLIW
▷ VLIW atravessa segmentos como pacote indivisível
▷ alocação $\pm$ fixa: $\mathbf{1}^{\underline{\mathbf{a}}}$ é ULA, $\mathbf{2}^{\underline{\mathbf{a}}}$ é LOAD, etc

ULA	ld	beq	ULA

História

- iniciou com microcódigo "horizontal"
- projetos na academia
  - ▷ ELI-512 [Fischer 85]
  - ▷ IMPACT [Hwu 91]
- produtos comerciais
  - ▷ Multiflow [Colwell, Fischer 85] $\rightsquigarrow$ falhou
  - ▷ Cydrome [Rau 85] $\rightsquigarrow$ falhou
  - ▷ EPIC, IA-64, Itanium [Colwell, Fischer, Rau 97] $\rightsquigarrow$??
  - ▷ Transmeta [Ditzel 99] $\rightsquigarrow$?? traduz x86$\longrightarrow$ VLIW
  - ▷ vários controladores embutidos (TI, Motorola) $\rightsquigarrow$ sucesso

VLIW per se

- VLIW puro: nenhum teste de dependência em hw
  nem mesmo entre grupos de VLIWs
- compilador responsável por escalonar todos os segmentos
  - ▷ inclusive os ciclos com bloqueios/stalls
  - ▷ é possível se conhece exatamente a estrutura da CPU e latências
- $1^{\underline{o}}$ problema: estrutura e latências diferem entre implementações
  - ▷ deve re-compilar para novas implementações
  - ▷ TransMeta faz re-compilação em tempo de execução
- $2^{\underline{o}}$ problema: latências **não** são fixas numa implementação
  - ▷ não usar caches? uh?
  - ▷ escalonar prevendo uma falta na cache?
    para que a cache então?

VLIW viável

Explicitly Parallel Instruction Computing (EPIC) ISA-64, Itanium

- menos rígido que VLIW talvez nem é VLIW...
- palavras de instruções com tamanho variável
  - ▷ implementadas como "amontoados" com bits de dependência
  - ▷ torna código compatível com máquinas de larguras $\neq$
- pressupõe que hw implementa lógica de bloqueio inter-amontoados
  - ▷ torna código compatível com $\neq$s núm_segmentos, latências
  - ▷ permite bloqueios por faltas na cache, e execução fora-de-ordem
- explora toda informação sobre paralelismo que compilador descobre
- compatível com $\neq$s implementações da mesma arquitetura

Escalonamento e Paralelismo no Nível de Instrução

Não faz sentido implementar um pipeline com largura N se,
na média,
muito menos que N instruções independentes executam num ciclo

desempenho é importante

mas utilização também é (sustentada/pico)

Exemplo: SAXPY

Single Precision A∗X+Y
rotina de álgebra linear para resolver sistemas de equações
parte dos "Livermore Loops"

```
for (i=0; i<N ; i++)
   Z[i] = A*X[i] + Y[i];

   ldf f0, X(r1)          // loop:
   mulf f4,f0,f2          // A em f2
   ldf f6, Y(r1)
   addf f8,f6,f4
   stf f8,Z(r1)
   add r1,r1,8            // i em r1
   ble r1,r2,loop         // N*4 em r2
```


Desempenho e Utilização – SAXPY escalar

Processador segmentado, escalar
MULF 5 ciclos, ADDF 2 ciclos, ambos segmentados
adiantamento completo, desvios previstos como tomados

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`ldf f0, X(r1)`	B	D	X	M	R											
`mulf f4,f0,f2`		B	D	d*	E	E	E	E	E	R						
`ldf f6, Y(r1)`			B	p*	D	X	M	R								
`addf f8,f6,f4`					B	D	d*	d*	d*	E	E	R				
`stf f8,Z(r1)`						B	p*	p*	p*	D	X	M	R			
`add r1,r1,8`										B	D	X	M	R		
`ble r1,r2,loop`											B	D	X	M	R	
.																

d*=dep dados; p*=dep segm, bolha

uma iteração com 7 instruções — latência de 15 ciclos

desempenho: 7 instr/15 ciclos $\longrightarrow$ IPC $= 0{,}47$

utilização: 0,47 IPCreal / 1,0 IPCideal $\longrightarrow 47\%$

Desempenho e Utilização – SAXPY superescalar

Processador segmentado, superescalar, largura N=2
MULF 5 ciclos, ADDF 2 ciclos, ambos segmentados

```
ldf f0, X(r1)
mulf f4,f0,f2
ldf f6, Y(r1)
addf f8,f6,f4
stf f8,Z(r1)
add r1,r1,4
ble r1,r2,loop
.
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B	D	X	M	R										
B	D	d*	d*	E	E	E	E	E	R					
	B	p*	p*	D	X	M	R							
	B	p*	p*	D	d*	d*	d*	d*	E	E	R			
		B	p*	D	p*	p*	p*	p*	X	d*	M	R		
				B	p*	p*	p*	p*	D	p*	X	M	R	
				B	p*	p*	p*	p*	D	p*	d*	X	M	R

d*=dep dados; p*=dep segm, bolha

latência ainda em 15 ciclos — por que???
utilização: 0,47 IPCreal / 2,0 IPCideal ⟶ 27%
mais riscos ⟶ mais bloqueios — por que?
cada bloqueio é mais caro — por que?

Escalonamento e Emissão

Escalonamento de instruções
⟶ escolhe ordem de execução das instr

- importante para melhorar utilização e desempenho
- relacionado à emissão de instr⟶ quando é que instr executam
  - ▷ VLIW puro: escalonamento estático com emissão estática
  - ▷ superescalar em-ordem, EPIC:
    escalonamento estático com emissão dinâmica (ou quase)
- escalar em-ordem depende de bom escalonamento pelo compilador

Escalonamento de Instruções

- **idéia:** instruções independentes entre operações lentas e uso RAW
  - ∗ evita bloqueio enquanto espera resolução do risco RAW
  - ∗ ≈ mesmo que escalonamento dinâmico (Tomasulo)
- para isso, necessita de instruções independentes
- escopo de escalonamento: região de código sob exame
  - ∗ quanto maior melhor – maior # instr independentes para escolher
  - ∗ com escopo definido, escalonar é simples
  - ∗ o problema é expandir o escopo ⇝ cruzar desvios?
- escalonamento pelo compilador
  - ▷ desenrolar loops — laços
  - ▷ *software pipelining* — laços
  - ▷ escalonamento de traçados *trace scheduling* — resto

Escalonamento: compilador ou hardware

compilador

- ▷ maior escopo (todo o programa), pode procurar mais longe
- ▷ possibilita hw simples com relógio rápido
- ▷ acurácia da previsão de desvios é ruim
- ▷ ∄ informação sobre latências (faltas na cache)
- ▷ recuperação de especulação errada é cara

hardware

- ▷ previsão de desvios com melhor acurácia
- ▷ informação dinâmica sobre latências (cache) e dependências
- ▷ fácil de especular e recuperar em caso de erro
- ▷ |buffer_de_instruções| finito limita escopo para escalonamento
- ▷ hw mais complicado (mais energia), relógio lento

Escalonamento por SW – desenrolar o SAXPY

Deseja-se separar operações dependentes

- ∗ escopo de uma iteração é muito pequeno
- ∗ cadeia mais longa de operações tem 9 ciclos
  - ▷ resultado do ldf – 1 ciclo
  - ▷ adianta para mulf – 5 ciclos
  - ▷ adianta para adf – 2 ciclos
  - ▷ adianta para stf – 1 ciclo
  - ▷ não dá para esconder latência de 9 ciclos em 7 instruções, mas dá se forem 2×9 ciclos em 14 instruções
- ∗ **desenrolar laço** = escalonar usando duas iterações

Desenrolar fase 1: agrupar iterações

Combinar 2 iterações (no geral N)

fundir controle do laço: incremento do índice e desvio

ajustar usos implícitos das variáveis de indução (r1 neste caso)

```
ldf f0, X(r1)        ldf f0, X(r1)
mulf f4,f0,f2        mulf f4,f0,f2
ldf f6, Y(r1)        ldf f6, Y(r1)
addf f8,f6,f4        addf f8,f6,f4
stf f8,Z(r1)         stf f8,Z(r1)
add r1,r1,8          --- --,--,--
ble r1,r2,loop       --- --,--,----
ldf f0, X(r1)        ldf f0, X+8(r1)
mulf f4,f0,f2        mulf f4,f0,f2
ldf f6, Y(r1)        ldf f6, Y+8(r1)
addf f8,f6,f4        addf f8,f6,f4
stf f8,Z(r1)         stf f8,Z+8(r1)
add r1,r1,8          add r1,r1,8+8
ble r1,r2,loop       ble r1,r2,loop
```

Desenrolar fase 2: escalonar segmentos

Escalonar para reduzir bloqueios por riscos RAW

.

```
ldf f0, X(r1)
mulf f4,f0,f2
ldf f6, Y(r1)
addf f8,f6,f4
stf f8,Z(r1)
ldf f0, X+8(r1)
mulf f4,f0,f2
ldf f6, Y+8(r1)
addf f8,f6,f4
stf f8,Z+8(r1)
add r1,r1,16
ble r1,r2,loop
```

```
ldf f0, X(r1)
ldf f0, X+8(r1)
mulf f4,f0,f2
mulf f4,f0,f2
ldf f6, Y(r1)
ldf f6, Y+8(r1)
addf f8,f6,f4
addf f8,f6,f4
stf f8,Z(r1)
stf f8,Z+8(r1)
add r1,r1,16
ble r1,r2,loop
```


Desenrolar fase 3: re-nomear registradores

Escalonamento causa riscos WAR:

é necessário re-nomear registradores

.

```
ldf f0, X(r1)
mulf f4,f0,f2
ldf f6, Y(r1)
addf f8,f6,f4
stf f8,Z(r1)
ldf f0, X+8(r1)
mulf f4,f0,f2
ldf f6, Y+8(r1)
addf f8,f6,f4
stf f8,Z+8(r1)
add r1,r1,16
ble r1,r2,loop
```

```
ldf f0, X(r1)
ldf f10, X+8(r1)
mulf f4,f0,f2
mulf f14,f10,f2
ldf f6, Y(r1)
ldf f16, Y+8(r1)
addf f8,f6,f4
addf f18,f16,f14
stf f8,Z(r1)
stf f18,Z+8(r1)
add r1,r1,16
ble r1,r2,loop
```


Desempenho do SAXPY desenrolado

```
                    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
ldf f0, X(r1)       B D X M R
ldf f10, X+8(r1)      B D X M R
mulf f4,f0,f2           B D E E E E E R
mulf f14,f10,f2           B D E E E E E R
ldf f6, Y(r1)               B D X M R
ldf f16, Y+8(r1)              B D X M s* s* R
addf f8,f6,f4                   B D d* E E s* R     dep em f4
addf f18,f16,f14                  B p* D E p* E R
stf f8,Z(r1)                        B D X M R
stf f18,Z+8(r1)                       B D X M R
add r1,r1,16                            B D X M R
ble r1,r2,loop                            B D X M R
.
```

d*=dep dados; p*=dep segm; s*=risco estrut regs

duas iterações com 12 instruções — latência de 17 ciclos

desempenho: 12 instr/17 ciclos ⟶ IPC = 0,71

utilização: 0,71 IPCreal / 1,0 IPCideal ⟶ 71% ($\gg$ 47%)

Nem tudo são flores...

Desenrolar loops é bom mas

- código é maior ($\approx N\times$)
- escalonamento pode ser ruim nas fronteiras dos laços
- não é possível tratar dependências entre iterações

```
for (i=0; i<N; i++)
    X[i] = A*X[i-1]; /* iteração depende da anterior */
```

laço desenrolado mantém uma cadeia de dependências que não pode ser escalonada

- o que fazer se (número_de_iterações MOD desenroladas $\neq 0$) ??

Software pipelining

Software pipelining = Desenrolamento simbólico de laços

Se as iterações do laço são independentes, então pode extrair mais paralelismo se usar instruções de iterações distintas

- **idéia:**
  - ▷ inicia com laço original
  - ▷ escreve novo laço com instruções de iterações distintas
  - ▷ adicione prólogo e epílogo para acertar início/final
- **ganhos:**
  - ▷ maximiza distância entre resultado e uso
  - ▷ código menor
  - ▷ enche e esvazia segmentos uma vez por volta, ao invés de uma vez em cada iteração desenrolada

Analogia com pipeline

- segmentação em hw
  cada ciclo contém:
  estágio 3 da instr 1, est 2 da instr i+1, est 1 da instr i+2
- segmentação em sw
  ciclo $\rightsquigarrow$ iteração "física" do laço
  instrução $\rightsquigarrow$ iteração "lógica" original do laço
  estágio $\rightsquigarrow$ instrução
- uma iteração "física" contém
  instruções de múltiplas iterações originais
  instr 3 da iteração 1, instr2 da iter i+1, instr 1 da iter i+2

Exemplo 1

iteração física: stf da iter original i ; ldf,mulf da iter original i+1
prólogo: inicializa pipeline ⟶ ldf,mulf da iteração 0
epílogo: completa trabalho ⟶ stf da última iteração

```
ldf f0,X(r1)                ldf f0,X-8(r1)   ← prólogo
mulf f4,f0,f2               mulf f4,f0,f2
stf f4,X(r1)          loop: stf f4,X(r1)     ← iteração
add r1,r1,8                 ldf f0,X(r1)     ← física
ble r1,r2,loop              mulf f4,f0,f2    ←
ldf f0,X(r1)                add r1,r1,8      ←
mulf f4,f0,f2               ble r1,r2,loop   ←
stf f4,X(r1)                stf f4,X+8(r1)   ← epílogo
add r1,r1,8
ble r1,r2,loop
```


Diagrama de tempo

Mesmo diagrama com nova terminologia:
ciclos ⇝ iteração física (horizontal)
instruções ⇝ iteração lógica (vertical)
estágios ⇝ instruções (LM=ldf,mulf S=stf)

	fís1	fís2	fís3	fís4	fís5	
lóg1	LM	S				← prólogo
lóg2		LM	S			
lóg3			LM	S		
lóg4				LM	S	← epílogo

iteração física 2 tem **stf** da iter lógica 1, e ldf,mulf da iter lógica 2

Note que numa iteração física os grupos de instruções estão na ordem invertida ⟶ OK porque grupos não são relacionados (exec //)

Exemplo 2

Altera estrutura para tolerar mais latência (3 iterações)

```
ldf f0, X(r1)               ldf f0, X-8(r1)
mulf f4,f0,f2               mulf f4,f0,f2    ← prólogo
stf f4, X(r1)               ldf f0, X-4(r1)
add r1,r1,8           loop: stf f4,X-8(r1)   ← iteração
ble r1,r2,loop              mulf f4,f0,f2    ← física
ldf f0, X(r1)               ldf f0, X(r1)    ←
mulf f4,f0,f2               add r1,r1,4      ←
stf f4, X(r1)               ble r1,r2,loop   ←
add r1,r1,8                 stf f4, X-4(r1)
ble r1,r2,loop              mulf f4,f0,f2    ← epílogo
ldf f0, X(r1)               stf f4, X(r1)
mulf f4,f0,f2
stf f4, X(r1)               L  M  S
add r1,r1,8                    L  M  S
ble r1,r2,loop                    L  M  S
                                     L  M  S
```

Software pipelining

- ▷ não aumenta (muito) o tamanho do código
- ▷ pode variar grau de paralelismo para tolerar maiores latências
  software super-pipelining
  uma iteração física com iterações lógicas i, i+2, i+4
- ▷ difícil de tratar condicionais dentro de laços
- ▷ alocação de registradores pode ser complicada

Suporte de hardware para escalonamento

Instruções condicionais ("predicadas")

Dois níveis de predicação

- predicação completa: cada instrução carrega um predicado IA-64
- move condicional: registrador é copiado se condição é válida
  conditional move = CMOVE

```
cmoveqz r1,r2,r3      # if (r3 == 0) r1←r2;
```

if-conversion converte fluxo de controle em fluxo de dados
⟶ elimina desvios

pode causar montes de replicação de código Alpha, IA-32

Suporte de hardware para escalonamento

Instruções condicionais ("predicadas")

Exemplo: emissão de 2 instruções/ciclo

PRIMEIRA INSTR	SEGUNDA INSTR
`lw r1,16(r2)`	`add r3,r4,r5`
	`add r6,r3,r7`
`beqz r10,label`	
`lw r8,0(r10)`	
`lw r9,0(r8)`	

desperdiça oportunidade de emissão porque
3º LW depende do resultado do **2º** LW.

Suporte de hardware para escalonamento

Instruções condicionais ("predicadas")

Usar versão condicional da instrução load: LWC = load condicional
load é efetivado se terceiro operando é zero

PRIMEIRA INSTR	SEGUNDA INSTR
`lw r1,16(r2)`	`add r3,r4,r5`
`lwc r8,0(r10),r10`	`add r6,r3,r7`
`beqz r10,label`	
`lw r9,0(r8)`	

Se a seqüência após o desvio fosse curta,
todo o bloco básico poderia ser convertido
para execução condicional e o desvio eliminado

Suporte de hardware para especulação em memória

Compilador só pode mover LOADs adiante de STOREs
se há certeza absoluta de que o movimento é correto

Arquitetura pode incluir instruções CHK e LDA
que verificam conflitos de endereçamento
CHK = check address LDA = load advanced

- ▷ instrução CHK é inserida no local original do LOAD
  e LDA é movido para antes do(s) STORE(s)
- ▷ quando load avançado LDA (especulativo) é executado,
  endereço efetivo é armazenado (no BdR?)
- ▷ se um STORE subseqüente referencia aquele endereço
  antes da instrução CHK então a especulação falhou
- ▷ se somente LOAD foi especulado, então basta repetir
  referência à memória quando instrução CHK é executada

Suporte de hardware para especulação em memória

Compilador não consegue garantir que **X+r1 ≠ Y+r2**

```
ldf f2,X(r1)        ldf f2,X(r1)
mulf f4,f2,f0       ldfa f6,Y(r2)    ← ld adiantado
stf f4,X(r1)        mulf f4,f2,f0
ldf f6,Y(r2)        stf f4,X(r1)
mulf f8,f6,f0       chk Y(r2)        ← verifica
stf f8,Y(r2)        mulf f8,f6,f0
                    stf f8,Y(r2)
```

Se quando executa primeiro STORE **X+r1 = Y+r2**,
então repete `ldf f6,Y(r2)` antes do segundo `mulf`

Mecanismo é chamado *memory conflict buffer* e usado no IA-64

Comportamento com excessões

Compilador não pode violar comportamento c.r.a excessões

Exemplo 1: instruções condicionais não podem causar excessões
se a condição é falsa, e o efeito da instr é anulado
anula inclusive/especialmente efeitos excepcionais

Exemplo 2: busca antecipada não pode causar excessões

Suporte de hardware para especulação em memória

O que acontece se o LOAD adiantado causa uma falta de página?
Nada, se usar LDE = load especulativo

```
ldf f2,X(r1)        ldf f2,X(r1)
mulf f4,f2,f0       ldfe f6,Y(r2)    ← ld especulativo
stf f4,X(r1)        mulf f4,f2,f0
bfnez f4,label      stf f4,X(r1)
ldf f6,Y(r2)        mulfe f8,f6,f0   ← mul especulativo
mulf f8,f6,f0       bfnez f4,label
stf f8,Y(r2)        stf f8,Y(r2)
```

- ▷ Bit de interrupção/excessão é associado ao `f6`
- ▷ bit é propagado para `f8` ⟶ `mulfe` é especulativo
- ▷ excessão é tratada quando `f8` é usado pelo stf
- ▷ mecanismo chamado de *poison bit* ou *deferred interrupt* no IA-64

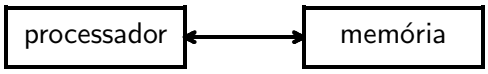
resumo – Escalonamento Estático

- desenrolar laços
  - ▷ reduz a freqüência de desvios
  - ▷ aumenta tamanho do código
- software pipelining
  - ▷ não há dependências no corpo do laço
  - ▷ não reduz a freqüência de desvios
  - ▷ necessita blocos com prólogo e epílogo
- escalonamento de traçados
  - ▷ pode usar para código sem laços
  - ▷ não é simples
- suporte pela arquitetura (conjunto de instruções)
  - ▷ loads especulativos, loads avançados
  - ▷ predicação, move-condicional

Sistema de Memória

- **Sistemas de memória**
  - ⋆ hierarquia de memória
  - ⋆ vazão e latência
  - ⋆ DRAM e SRAM
- Memória cache
  - ⋆ organização
  - ⋆ leitura
  - ⋆ escrita
  - ⋆ otimizações
- Memória Virtual
  - ⋆ endereçamento
  - ⋆ paginação
  - ⋆ TLB
  - ⋆ excessões

Sistemas de Memória



Atualmente, o desempenho de computadores é limitado pela **latência** da memória e pela **vazão** até a memória

latência é o tempo de um único acesso [s]
tempo de acesso à memória $\gg$ ciclo do processador

vazão é o número de acessos por unidade de tempo [refs/s]
se uma fração m das instruções acessam a memória
$\longrightarrow$ ocorrem $1 + m$ referências/instrução
$\longrightarrow$ CPI=1 se e só se ocorrerem $1 + m$ referências/ciclo

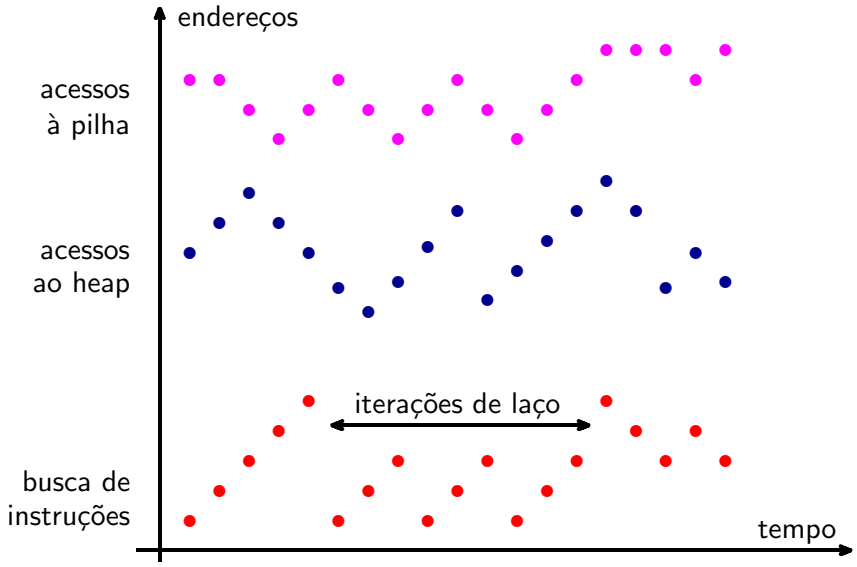
Vazão e Latência

- **Latência** da memória é o intervalo desde a requisição pelo processador até a disponibilidade para o processador [s]
- **Vazão** é a taxa de transferência de/para a memória [x/s]
  bandwidth = largura de banda
- Vazão e latência são intimamente relacionadas:
  Se **R** é o número de requisições que a memória pode atender simultaneamente, então

$$V = R/L \qquad [x/s = x \,/\, s]$$

Porque raramente $V = 1/L$?

Padrões de Referência



Princípios da Localidade I

Referências à memória apresentam duas propriedades:

Localidade Temporal: se um local é referenciado, possivelmente, este será referenciado de novo em futuro próximo

Localidade Espacial: se um local é referenciado, possivelmente, locais próximos serão referenciados em futuro próximo

Princípios da Localidade II

```
for (i=0; i<100; i++)
   for (j=0; j<200; j++)
      res[i] += vetor[i] * matriz[i][j];
```

localidade temporal: código, variáveis locais `i` e `j`

localidade espacial: elementos de `res[]`, `vetor[]`, `matriz[]` e código (blocos básicos)

Princípios da Localidade III

Caches podem tirar proveito dos dois tipos de localidade:

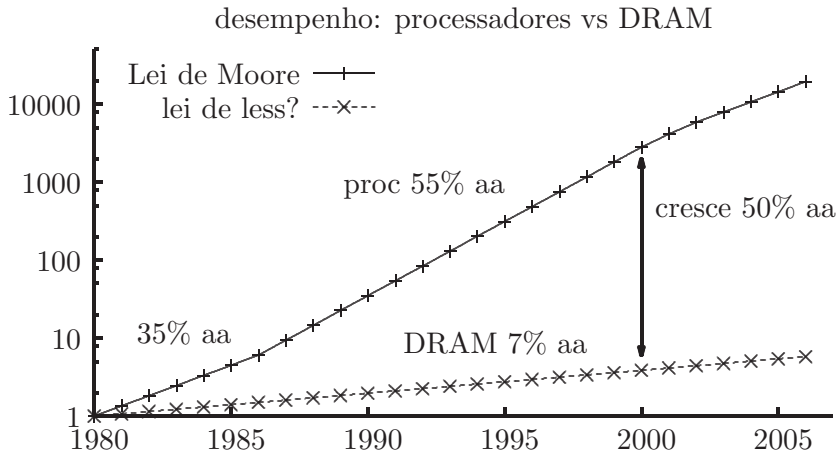
da localidade temporal ao lembrar do
conteúdo de posições acessadas recentemente

⟶ localidade temporal ajuda escolher o que expurgar da cache

da localidade espacial ao buscar
blocos em áreas acessadas recentemente

⟶ localidade espacial ajuda escolher o que carregar na cache

Hierarquias de Memória



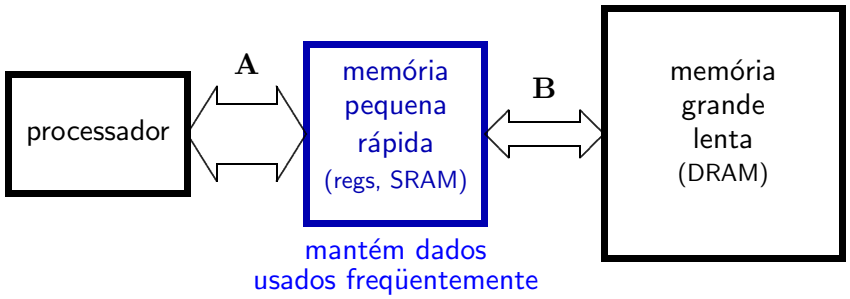
Processador que emite quatro instruções por ciclo poderia executar 800-1000 instruções durante uma falta na cache!!!

Hierarquias de Memória

Solução para aumento na latência relativa:
esconder a latência
interpondo memória pequena mas com alta velocidade
entre processador e memória DRAM

Memória cache permite esconder latência da memória porque
padrões de referência são muito previsíveis ⟶ localidade

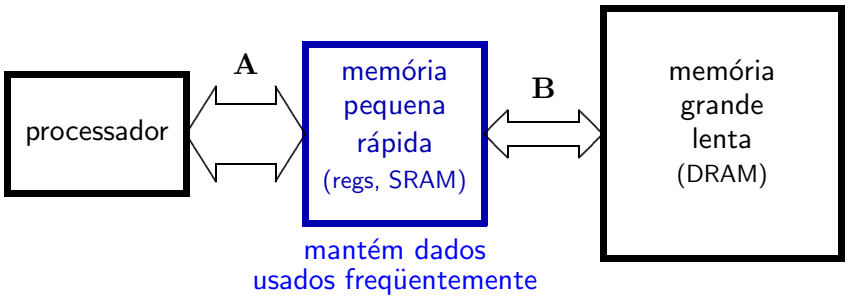
Hierarquia de Memória



tamanho:	registradores $\ll$ SRAM $\ll$ DRAM	por que?
latência:	registradores $\ll$ SRAM $\ll$ DRAM	por que?
vazão:	interna ao CI $\gg$ externa ao CI	por que?

Memória rápida será efetiva se e só se tráfego em B $\ll$ A

Hierarquia de Memória



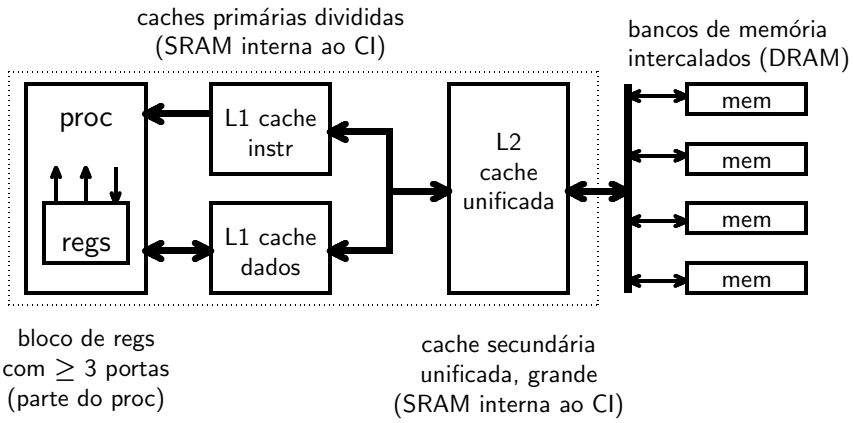
No acesso ao conteúdo:

acerto se encontra	(dado $\in$ cache) $\longrightarrow$ latência baixa	*hit*
falta se não encontra	(dado $\notin$ cache) $\longrightarrow$ latência elevada	*miss*

Gerência da Hierarquia de Memória

- Gerência por software (registradores)
  - ▷ é parte do estado da CPU visível compilador/programador
  - ▷ SW tem controle completo sobre alocação de armazenadores
    mas HW pode fazer coisas às escondidas,
    como re-nomeação de registradores
- gerência por hardware (caches)
  - ▷ não é parte do estado visível
  - ▷ HW decide automaticamente o que manter em memória rápida
    mas SW pode dar pistas,
    como fazer busca antecipada ou marcar *don't cache*

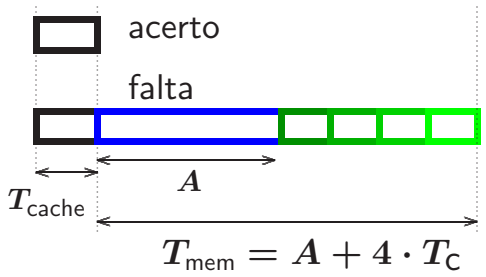
Sistemas de Memória Típicos



"Lei de less": Processador super-escalar pode executar
$\approx 10^3$ instruções durante falta na cache

Tempo médio de acesso à memória

$$T_{\text{med}} = T_{\text{cache}} * (\text{acertos} + \text{faltas} * T_{\text{mem}})$$



A = tempo de acesso à memória

T_C = tempo para copiar da memória para a cache (4 palavras por bloco)

Custo de acerto: $\approx$ ciclo do processador
penalidade por falta: $\gg$ ciclo do processador

Memória – vazão

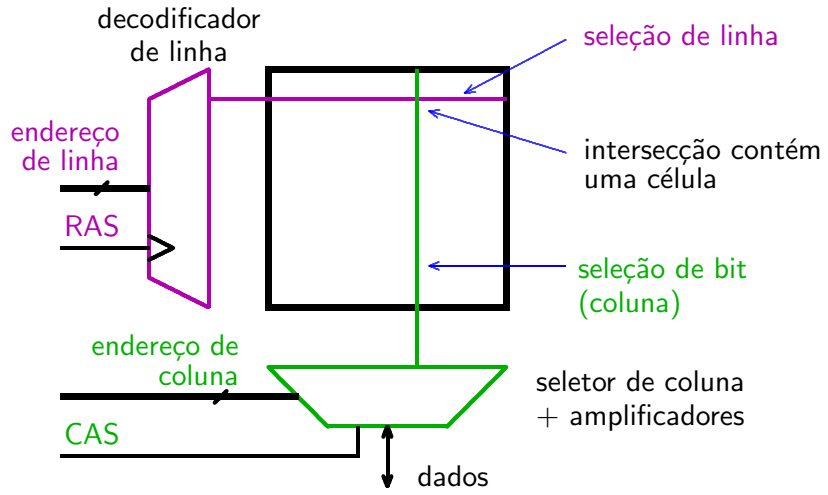
- **Vazão:** número de acessos por unidade de tempo [acessos/s]
  - ▷ taxa de transferência [bytes/segundo]
  - ▷ afeta tempo de carga dos blocos da cache e operações de E/S
  - ▷ pode ser aumentada com **$$$**
  - ▷ vazão de pico $\leq$ largura/unidade_de_tempo
    vazão de pico é aquela impossível de ser sustentada
  - ▷ vazão efetiva $\approx$ largura/(unidade_de_tempo + overhead)
  - ▷ se fração M das instruções referencia memória
    $\rightarrow 1 + M$ referências à memória/instrução
    $\rightarrow$ CPI=1 só se executa $\geq 1 + M$ referências/ciclo

Memória – latência

- **Latência:** tempo para um único acesso [s]
  - ▷ afeta **todas** as operações na memória
    ⟶ afeta penalidade nas faltas em cache
  - ▷ não pode ser diminuída só com **$$$**
    mas pode ser escondida
  - ▷ no limite: velocidade da luz
    $\simeq \mathbf{0.7}c$ no silício/alumínio $\approx$ 1 palmo/ns

c = velocidade da luz (30cm/ns no vácuo)

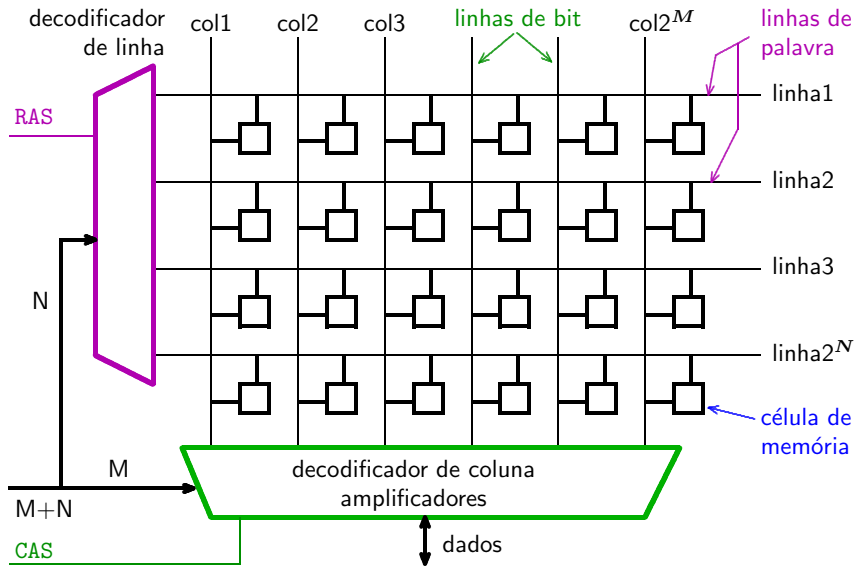
Organização da Memória Principal – introdução



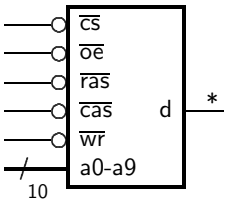
Organização da Memória Principal – introdução

- Memória principal usa DRAM *Dynamic Random Access Memory*
  - ▷ **latência:** afeta penalidade nas faltas em cache
    tempo de acesso: tempo entre requisição e dados prontos
    tempo de ciclo: tempo entre duas requisições
  - ▷ organizada como matriz (quadrado/retângulo)
    endereços multiplexados e tranferidos em 2 partes
    linhas, controlado por ⟶ *RAS = row address strobe*
    colunas, controlado por ⟶ *CAS = column address strobe*
    porque memória é organizada numa matriz 2D
- Cache usa SRAM *Static Random Access Memory*
  - ▷ sem refresh (6 transistores/célula ao invés de 1)
    tamanho de células SRAM/DRAM $\approx$ 4-8
    custo e tempo de ciclo DRAM/SRAM $\approx$ 8-16

Organização de CI de DRAM

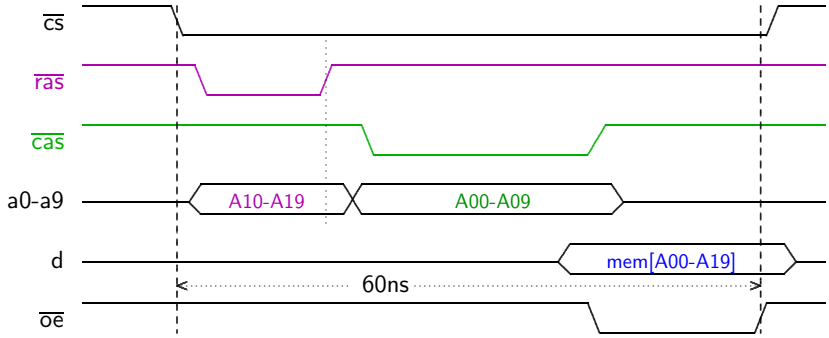


Sistemas de Memória



cs = chip select ras = row address strobe
oe = output enable cas = column addrs strobe
wr = write
Um acesso à memória custa aproximadamente 60ns:

20ns RAS + 20ns CAS + 20ns OE



Memória Dinâmica

- DRAM é otimizada para densidade e não para velocidade
  - ▷ célula com um transistor que se comporta como capacitor
  - ▷ bits armazenados como carga elétrica no capacitor
  - ▷ capacitor se descarrega na leitura ⟶ leitura destrutiva
  - ▷ carga se dispersa com o tempo (‘vaza’)
    refresca memória acessando cada linha da matriz para ‘leitura’
  - ▷ deve ser ‘refrescada’ periodicamente — 5-8ms, 1% do tempo
- ciclo dura aprox dobro do tempo de acesso
  - ▷ é necessário pré-carregar as linhas de dados antes de acessar

Célula de Memória Dinâmica

Escrita:

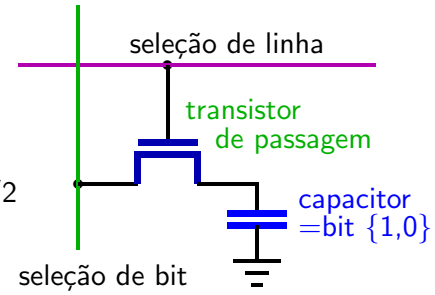
1. ativa seleção de bit (bit line)
2. seleciona linha

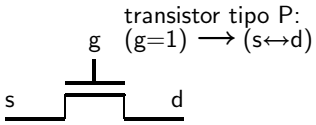
Leitura:

1. carrega linha de bit até Vdd/2
2. seleciona linha
3. linha de bit e capacitor dividem carga
4. amplifica diferença de voltagem amplificador sente diferença de $\mathbf{10^6}$ elétrons
5. escreve e reforça valor

Refresh:

1. lê conteúdo de cada célula





Célula de Memória Dinâmica – escrita/leitura

Escrita:

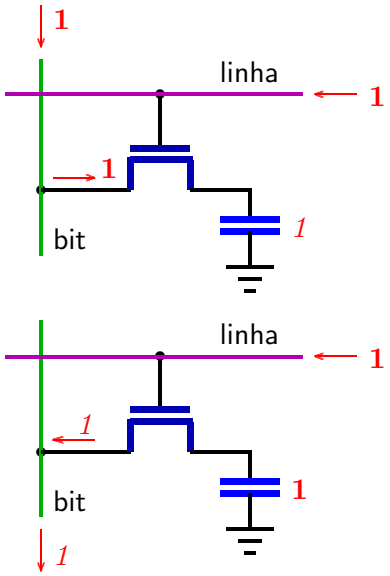
1. força valor na linha de bit
2. seleciona linha
3. capacitor mantém valor por 60ms, então refresca

Leitura:

1. carrega linha de bit até Vdd/2
2. seleciona linha
3. linha de bit e capacitor dividem carga
4. amplificador detecta valor (1/0)
5. re-escreve valor

Refresh:

igual a leitura



DRAM – histórico

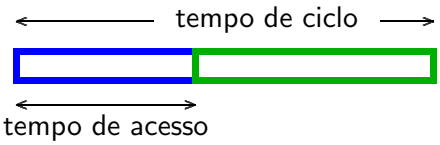
ano	capacidade [Mbits]	acesso [ns] mesma col	ciclo [ns] nova col
1980	0,064	150	250
1983	0,256	120	220
1986	1	100	190
1989	4	80	160
1992	16	60	120
1996	64	50	100
2000	256	40	90
2002	512	5	50
2004	1024	3	45
	60% aa	7% aa	

capacidade cresce 60% aa (melhor desde 96)

tempo de acesso reduz 7% aa

Desempenho da Memória Principal

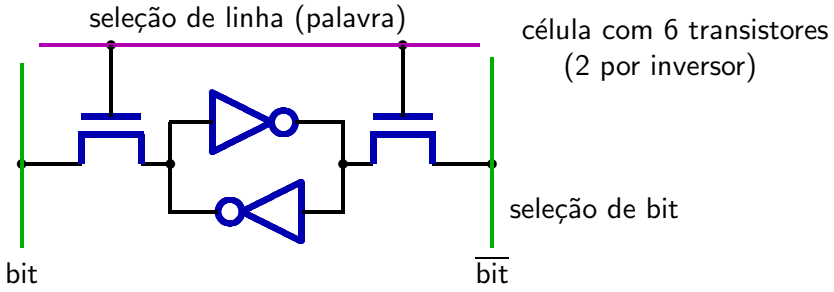
- Tempo de Ciclo $\gg$ tempo de acesso ($\approx$ 2:1) (leit/escr)
- Tempo de Ciclo (leit/escr)
  $\longrightarrow$ com que freqüência pode iniciar um acesso?
- Tempo de Acesso (leit/escr)
  $\longrightarrow$ quanto tempo demora para acessar o conteúdo, após iniciar acesso?
- Tempo de ciclo limita vazão da memória



Memória Estática – SRAM

- Organização retangular (linha/coluna)
  - ▷ linhas de endereços e dados não são multiplexadas
- optimizada para **1º** velocidade, e **2º** densidade
  - ▷ 4-6 transistores por célula
  - ▷ bits armazenados em flip-flops/latches
  - ▷ estática $\longrightarrow$ sem refresh
  - ▷ consumo de energia é maior do que DRAM
- tempo de acesso = duração do ciclo
- densidade/desempenho
  - ▷ 1/4 da densidade de DRAM
  - ▷ 4-8x velocidade
- em 1998, capacidade de 1Mbit, tempo de acesso de 7 a 20ns

Célula de Memória Estática (Static RAM)



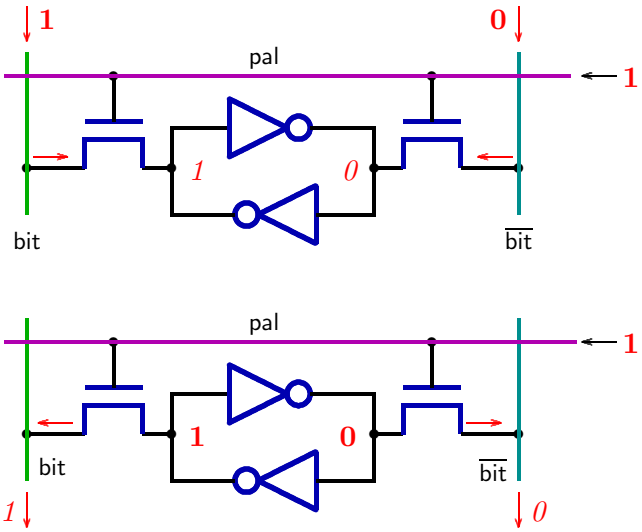
Escrita:

1. ativa linhas de bit
   bit=1, $\overline{\text{bit}}$=0 (FF $\leftarrow$ 1)
2. seleciona linha/palavra

Leitura:

1. carrega linhas de bit até Vdd/2
2. seleciona linha/palavra
3. célula puxa uma das linhas para **0**
4. amplificador detecta diferença entre bit e $\overline{\text{bit}}$

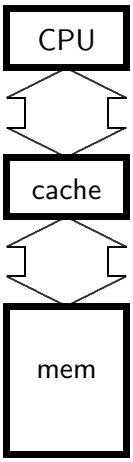
Célula de Memória Estática – escrita/leitura



Escrita:
força novo valor nas linhas de bit e ativa linha de palavra

Leitura:
ativa linha de palavra e célula força bit armazenado

Memória Principal Simples (simplória?)



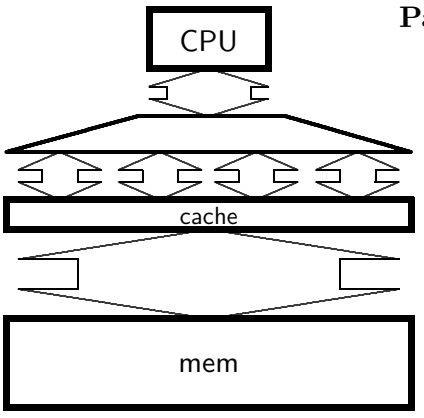
Parâmetros:
4 ciclos para emitir endereço
56 ciclos para acessar cada palavra
4 ciclos para transmitir palavra para cache/CPU

Penalidade para carregar bloco de 4 palavras:
(4 + 56 + 4) x 4 palavras = 256 ciclos

Simples:
CPU, cache, barramento e memória mesma largura (32 bits)

Barato mas desempenho ruim!

Memória Principal Larga



Parâmetros:
4 ciclos para emitir endereço
56 ciclos para acessar 4 palavras
4 ciclos para transf p cache/CPU

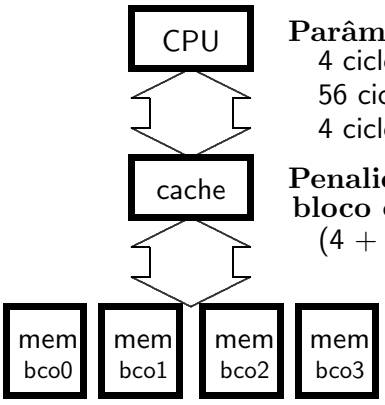
Penalidade para carregar bloco de 4 palavras:
(4 + 56 + 4) = 64 ciclos

Largo:
CPU/mux 1 palavra
mux/cache, barramento, memória N palavras
(Alpha com 64 e 256 bits)

Desempenho bom mas caro!

L1 com largura 1 e L2 com largura 4, remove o MUX do caminho entre L1 e CPU

Memória Principal Intercalada



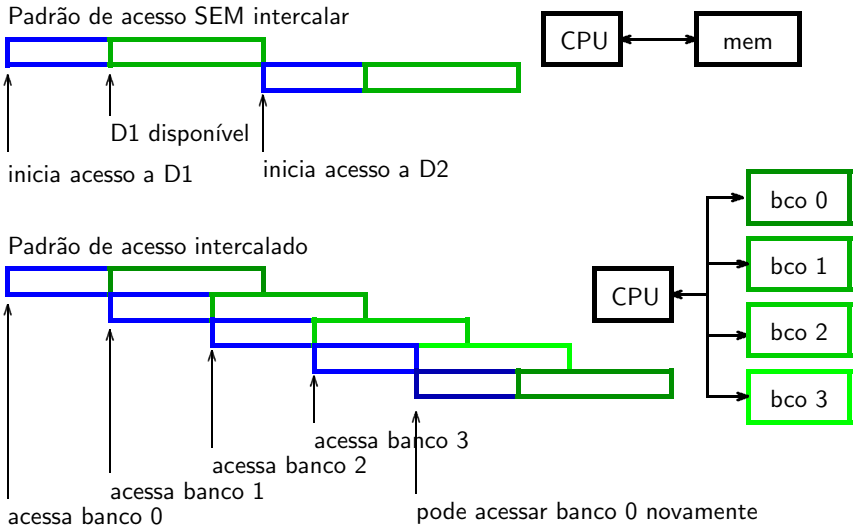
Parâmetros:
4 ciclos para emitir endereço
56 ciclos para acessar cada palavra
4 ciclos para transm pal para cache/CPU

Penalidade para carregar bloco de 4 palavras:
$(4 + 56 + 4x4) = 76$ ciclos

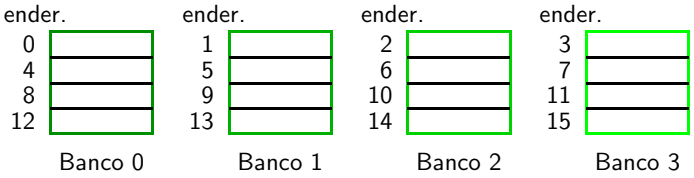
Intercalada:
CPU, cache, barramento de 1 pal
Memória dividida em N módulos

Boa relação de custo/desempenho

Intercalar para aumentar a vazão



Intercalar para aumentar a vazão



- Palavras vizinhas alocadas em bancos distintos
  - ▷ alocação definida pelos bits menos-significativos do endereço
- Bancos podem ser acessados concorrentemente
  - ▷ se número de bancos $\gg$ tamanho do bloco da cache
  - ▷ acessos a endereços $\neq$s não-contíguos podem ser emitidos para bancos distintos: 1000,1004,1008,100c || 1014,1018,101c,1020
    pressupõe que $\exists \geq 8$ bancos
- número de bancos $\geq$ # ciclos para acessar palavra num banco

Memória Intercalada

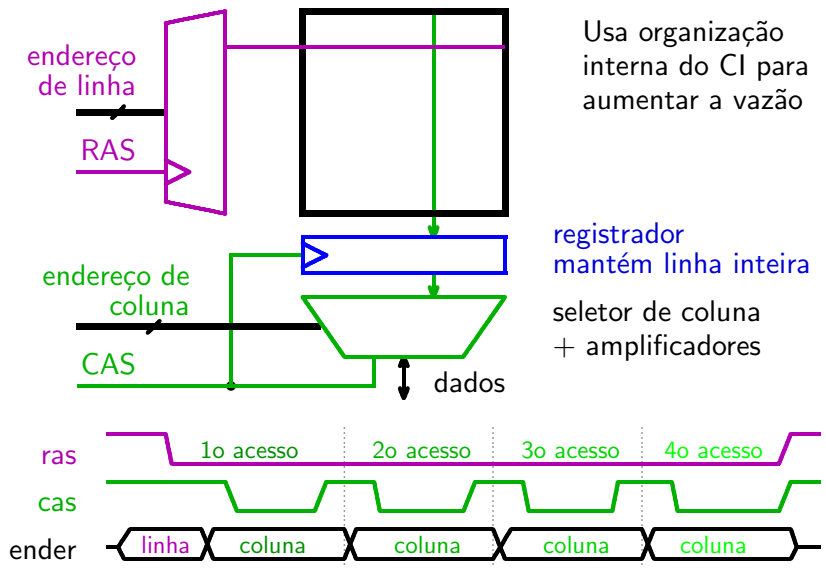
- Intercalação simples
  - ▷ necessita circuito de controle em cada banco
  - ▷ emite endereço para todos os bancos (*broadcast*)
  - ▷ efetua acesso em paralelo
  - ▷ multiplexa o barramento para transferir
  - ▷ bom para sistemas com cache porque unidade de transfer é bloco
  - ▷ bom com escrita preguiçosa (escreve todo o bloco sujo)
- Intercalação complexa (super-bancos)
  - ▷ acesso concorrente a palavras não-contíguas
  - ▷ acessos concomitantes

    ⟶ super-bancos endereçados individualmente

    ⟶ interface de memória com pipeline

<table>
<tr><td rowspan="2">número do super-banco</td><td colspan="2">deslocamento no super-banco</td></tr>
<tr><td>núm do banco</td><td>deslocamento no banco</td></tr>
</table>

Fast Page Mode



Fast Page Mode

- Fast Page Mode — inventado para adaptadores de vídeo
  - ▷ registrador mantém toda uma linha da matriz
  - ▷ mudar coluna para acessar outros bits na mesma linha
- Extended Data Out – acrescenta latch entre mux e saída

  ⟶ permite mudar CAS mais cedo
- Synchronous DRAM – acrescenta sinal de relógio para acelerar sinalização entre processador e controlador de memória
- Double Data Rate RAM – como SDRAM, mas transmite dados nas duas bordas do relógio

Pergunta:

Considere um sistema de memória com tempo de acesso (mesma linha) de 14 ciclos. Quantos bancos são necessários para se obter o melhor desempenho ao menor custo?

a) 4 bancos
b) 8 bancos
c) 16 bancos
d) 32 bancos

Resposta:

Considere um sistema de memória com tempo de acesso (mesma linha) de 14 ciclos. Quantos bancos são necessários para se obter o melhor desempenho ao menor custo?

a) 4 bancos
b) 8 bancos
c) 16 bancos # bancos $\geq$ # ciclos, # bancos $= \mathbf{2}^{B}$
d) 32 bancos

Pergunta:

Considere um sistema com memória intercalada em 16 bancos de 32 bits. O bloco da cache é de quatro palavras.

Quantas referências concorrentes podem ser sustentadas?

Resposta:

Considere um sistema com memória intercalada em 16 bancos de 32 bits. O bloco da cache é de quatro palavras.

Quantas referências concorrentes podem ser sustentadas?

Depende das combinações de tipos de acesso:

⟶ a quatro blocos de cache

(faltas de leitura, cache não-bloqueante, sem conflitos nos bancos)

⟶ escritas a 16 palavras consecutivas

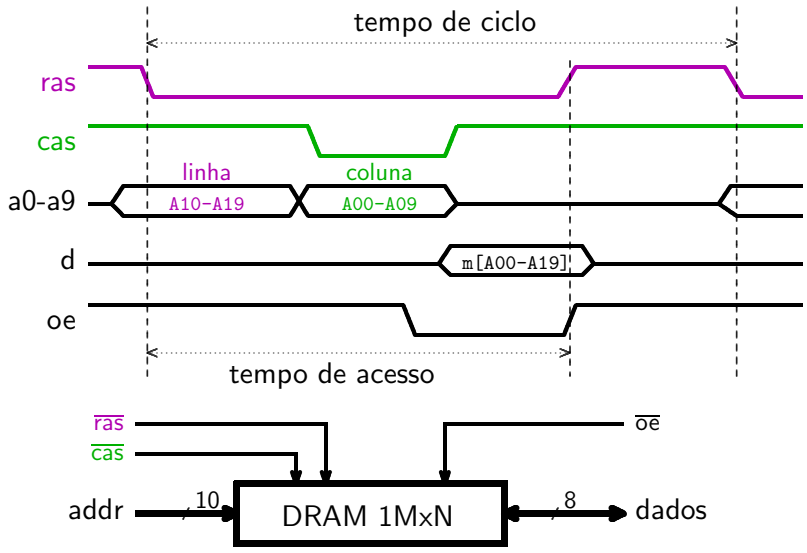
(se barramento suportar)

⟶ leitura de dois blocos, e escritas $n \leq 8$ palavras

(se cache não-bloqueante e barramento suportarem)

etc...

Temporização de Leitura



Parâmetros de Temporização de Leitura

$\mathcal{T}_{RAC}$ intervalo entre descida de RAS e dados válidos

- ⋆ quotado como velocidade da DRAM **(tempo de acesso)**
- ⋆ valor típico para DRAM de 4Mb é $\mathcal{T}_{RAC}$=60ns

$\mathcal{T}_{RC}$ intervalo mínimo entre o início de um acesso a uma linha e o próximo **(tempo de ciclo)**

- ⋆ $\mathcal{T}_{RC}$=110ns para DRAM de 4Mb com $\mathcal{T}_{RAC}$=60ns

$\mathcal{T}_{CAC}$ intervalo mínimo entre descida do CAS e dados válidos

- ⋆ $\mathcal{T}_{CAC}$=15ns para DRAM de 4Mb com $\mathcal{T}_{RAC}$=60ns

$\mathcal{T}_{PC}$ intervalo mínimo entre o início de um acesso a uma coluna e o próximo acesso

- ⋆ $\mathcal{T}_{PC}$=35ns para DRAM de 4Mb com $\mathcal{T}_{RAC}$=60ns

Memória Intercalada – vazão entre L1 e L2

- Processador com relógio de 1GHz, 2 instruções por ciclo
  - ⋆ 0.02 = taxa de faltas de instruções na L1i 1.25 refs/instrução
  - ⋆ 0.10 = taxa de faltas de dados na L1d

qual a demanda na L2?

$1 * 0.02 + 0.25 * 0.10 = 0.045$ faltas/instr

2 instr/ns $* 0.045$ faltas/instr $= 0.09$ faltas/ns

- a cada falta carrega 4 pals da L2 na L1d+L1i:
  0.09 bloco/ns $* 4$ pals/bloco $= 0.36$ pals/ns $\longrightarrow 2.78$ ns/pal
- 0.09 bloco/ns $* 4$ pals/bloco $* 4$ byte/pal $\longrightarrow 694.4$ Mbyte/s

bancos	*SRAM 32bits,* $\mathcal{T}_{ac}$*=12ns*	OK?
4	16b / (12+4) ns = 1.0 Gbyte/s	$>$ 0.694 Gb/s
8	32b / (12+8) ns = 1.6 Gbyte/s	$\gg$ 0.694 Gb/s
16	64b / (12+16) ns = 2.28 Gbyte/s	$\gg$ 0.694 Gb/s

Memória Intercalada – latência na L2

- Processador com relógio de 1GHz, 2 instruções por ciclo
  - ⋆ 0.02 = taxa de faltas de instruções na L1i 1.25 refs/instrução
  - ⋆ 0.10 = taxa de faltas de dados na L1d

qual a demanda na L2?

$1 * 0.02 + 0.25 * 0.10 = 0.045$ faltas/instr $* 2 = 0.09$ faltas/ns

- a cada falta carrega 4 pals da L2 na L1d+L1i:
  0.09 bloco/ns $* 4$ pals/bloco $= 0.36$ pals/ns $\longrightarrow 2.78$ ns/pal
- *quantos bancos para suprir demanda/vazão?*

bancos	*SRAM 32bits,* $\mathcal{T}_{ac}$*=12ns*	OK?
4	3.0 ns/pal	$\gg \mathbf{1}$ ns	$\times$
8	1.5 ns/pal	$> \mathbf{1}$ ns	$\times$
16	0.75 ns/pal	$< \mathbf{1}$ ns	$\surd$

$\times$ *vazão OK mas taxa xfer* $>$ *ciclo*

Memória Intercalada – vazão entre L2 e RAM

- Processador com relógio de 1GHz, 2 instruções por ciclo
  - ⋆ 0.02 = taxa faltas instruções L1i 0.10 = taxa faltas dados L1d
  - ⋆ 0.075 = taxa de faltas de dados na L2 1.25 refs/instrução

qual a demanda na RAM?

$(1 * 0.02 + 0.25 * 0.10) * 0.075 = 0.003375$ faltasL2/instr

2 instr/ns $* 0.003375$ faltas/instr $= 0.00675$ faltas/ns

- a cada falta carrega 8 pals da RAM para a L2:
  6.75 bloco/μs $* 32$ bytes/bloco $= 216$ bytes/μs $\longrightarrow 216$ Mb/s

bancos	$\mathcal{T}_{RAC}$*=30ns,* $\mathcal{T}_{RC}$*=70ns*	OK?
4	16b / (70+16) ns = 186 Mb/s	$<$ 216 Mb/s
8	32b / (70+32) ns = 313 Mb/s	$>$ 216 Mb/s
16	64b / (70+64) ns = 477 Mb/s	$\gg$ 216 Mb/s

4 ns por ciclo do barramento (250MHz), DRAM 32bits

Balanceamento de Vazão entre CPU e Memória

- Requisitos de vazão pelo processador
  - ⋆ processador super-escalar com ciclo de 1ns
  - ⋆ sem cache de dados
  - ⋆ 1 acesso à memória (64 bits) por ciclo do processador
- Vazão suportada pela memória
  - ⋆ DRAM com ciclo de 90ns
  - ⋆ CIs com 256Mbits
    ⟶ 128 bancos para prover 1 palavra por ciclo

    256Mb x 1 ⟶ 64x1 x 128 = 8192 chips ⟶ 256 Gbytes
    64Mb x 4 ⟶ 16x4 x 128 = 2048 chips ⟶ 64 Gbytes
- Estes números não são razoáveis...

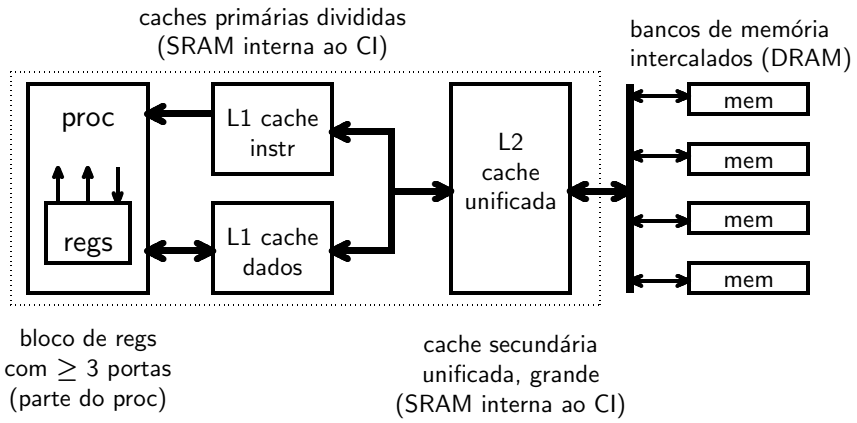
Balanceamento ainda

- Adicionar uma cache de dados
  - ⋆ taxa de faltas de 5%
  - ⋆ blocos com 4 palavras de 64 bits
  - ⋆ escrita preguiçosa
  - ⋆ 25% das faltas são em blocos sujos
    ⟶ demanda de aprox 1 pal a cada 4 ciclos
    demanda diminui por fator de 4
- Capacidade mínima ainda é impraticável:
  - ⋆ 256Mb x 1 ⟶ 64x1 x 128 / 4 = 2048 chips ⟶ 64 Gbytes
    64Mb x 4 ⟶ 16x4 x 128 / 4 = 512 chips ⟶ 16 Gbytes
- Solução: adicionar mais um nível de cache *mais caro que 16G?*
- Outra Solução: usar CIs mais largos (x8, x16, x32)
- Outra Solução: usar CIs mais avançados
  fast page mode, EDO, Sdram, DRdram...

Cache

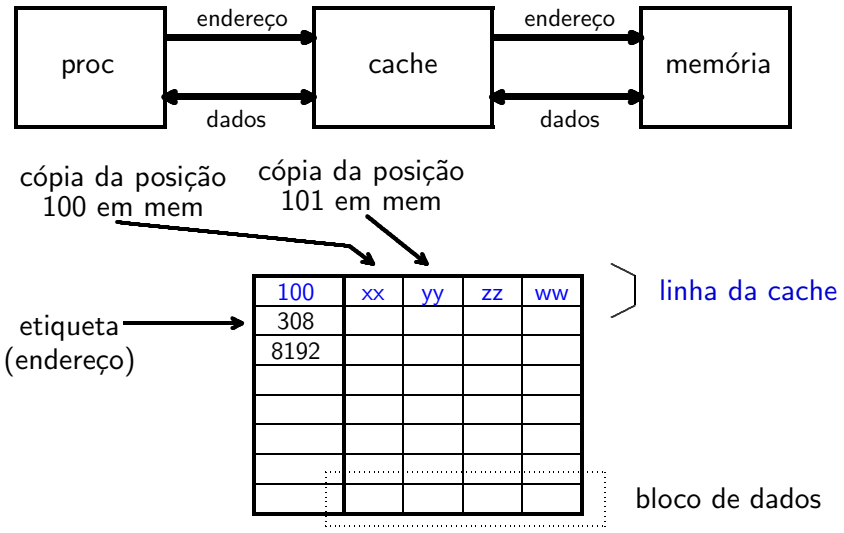
- Sistemas de memória
- **Memória cache**
  - ∗ **organização**
  - ∗ **leitura**
  - ∗ **escrita**
  - ∗ **otimizações**
  - ∗ **busca antecipada**
- Memória Virtual
  - ⋆ endereçamento
  - ⋆ paginação
  - ⋆ TLB
  - ⋆ excessões

Sistemas de Memória Típicos



"Lei de Less": Processador super-escalar pode executar
$\approx 10^3$ instruções durante falta na cache

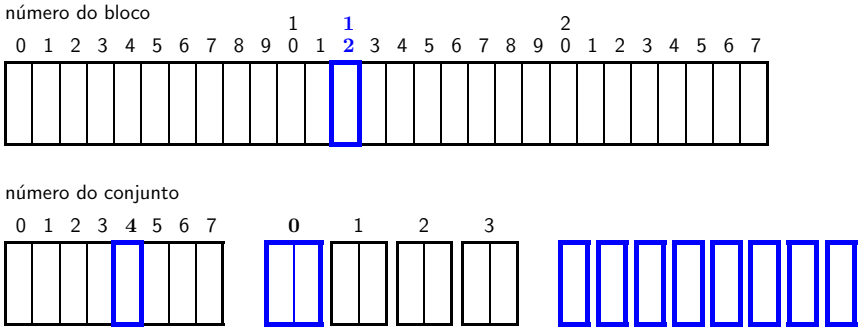
Cache por dentro



Algoritmo para leitura

cache recebe endereço;
se objeto existe na cache (acerto hit)
    entrega para processador;
senão (falta miss)
    lê bloco da memória
        espera o tempo de acesso à memória
    entrega ao processador
    e atualiza cache

Alocação de blocos



mapeamento direto

bloco 12 mapeia somente no bloco 4
12 % 8 = 4

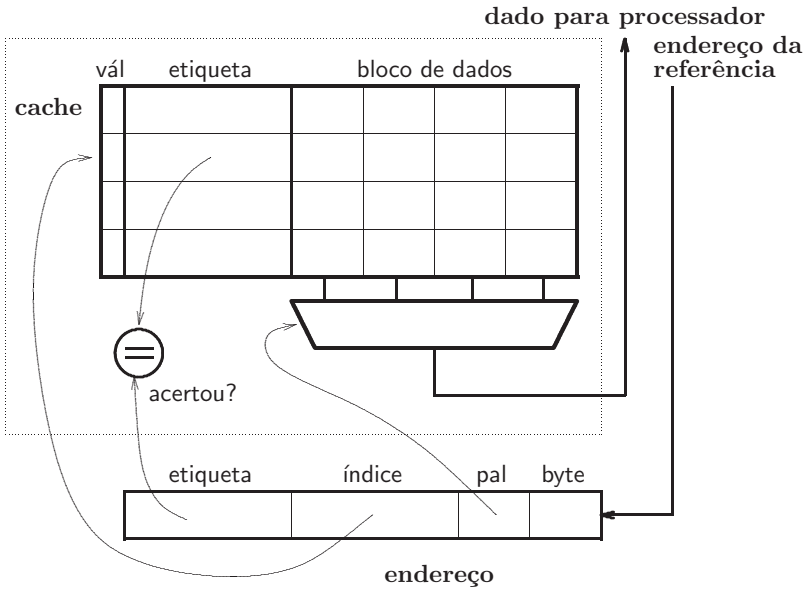
associativa por conjuntos (binária)

bloco 12 mapeia no conjunto 0
12 % 4 = 0

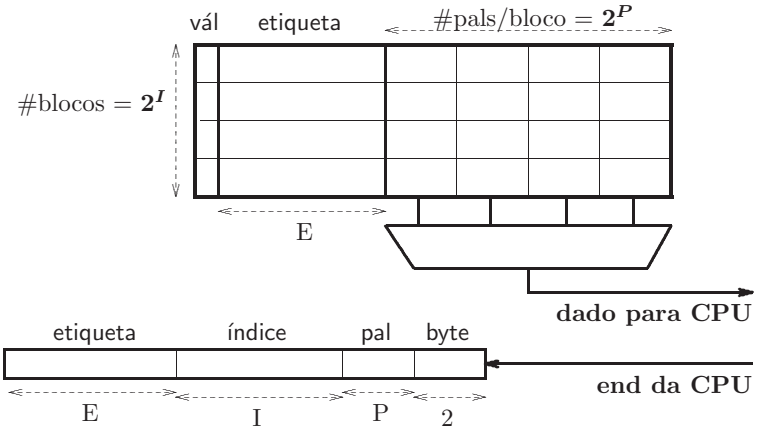
associativa (total)

bloco 12 mapeia em qualquer conjunto

Mapeamento Direto



Organização da cache

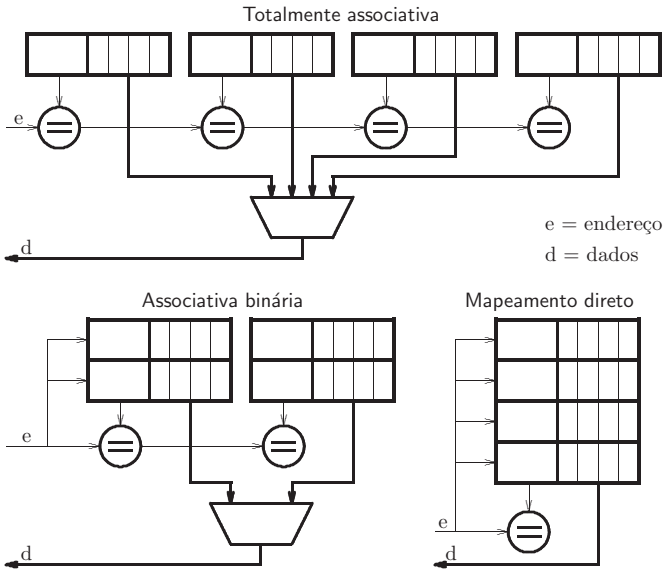


tamanho da cache = num_blocos * tam_bloco * tam_palavra

Quanto mais blocos $\Longrightarrow$ menor a taxa de faltas

Blocos com 4-16 palavras tiram vantagem de localidade espacial

Associatividade I



Associatividade II

M[12] = 99; M[13] = 200; M[14] = 33

Associativa - 4 conjuntos

0		99		200		33		

aloc: 12%1 = ?

Associativa binária - 2 conjuntos

0		99		33
1		200		

aloc: 12%2 = 0

aloc: 13%2 = 1

Mapeam direto - 1 conjunto

0	99	aloc: 12%4 = 0
1	200	aloc: 13%4 = 1
2	33	aloc: 14%4 = 2
3		aloc: N%4 = 3

Escolha de vítima para reposição: com mapeamento direto ∄ escolha...

1) escolhe o bloco usado no passado mais distante (LRU)

2) escolhe a esmo

Taxas de Faltas e de Acertos

Se encontra na cache — **acerto**

Se não encontra na cache — **falta**

$$\text{taxa de acertos} \triangleq \frac{\text{número de acertos}}{\text{número de referências}}$$

$$\text{taxa de faltas} \triangleq 1 - \text{taxa de acertos}$$

Pergunta:

Qual a taxa de faltas se |conj_dados| $\gg$ capacidade_da_cache?

(a) baixa
(b) média
(c) alta
(d) faltas independem dos tamanhos relativos

Resposta:

Qual a taxa de faltas se |conj_dados| $\gg$ capacidade_da_cache?

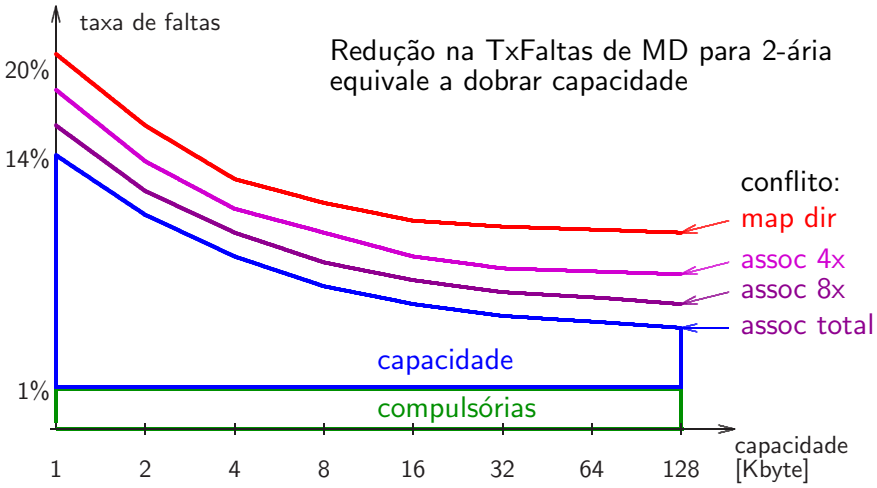
(a) baixa
(b) média
(c) alta
(d) faltas independem dos tamanhos relativos

Qual a taxa de faltas se |conj_dados| $\ll$ capacidade_da_cache?

Causas de faltas

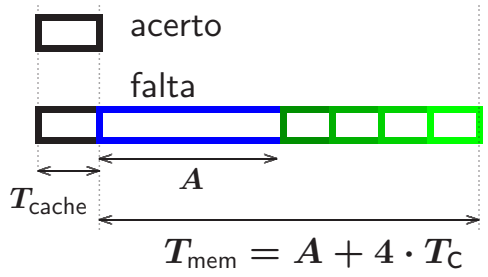
- faltas compulsórias – linhas são tocadas pela primeira vez.
  - ▷ Solução: blocos com muitas palavras para fazer busca antecipada implícita;
- faltas por capacidade – conj. de trabalho $\gg$ tamanho da cache.
  - ▷ Solução: aumentar tamanho da cache;
- faltas por conflito – mapeamento de endereços $\neq$s em mesmo bloco da cache.
  - ▷ Solução: aumentar associatividade.
  - ▷ ocorrem mesmo quando a cache não está cheia;

Causas de faltas



Tempo médio de acesso à memória I

- 1 ciclo por acerto na cache (T_{cache})
- penalidade: 2 a 100 ciclos para acessar memória (T_{mem})
- tMed $= T_{\text{cache}} +$ (faltas $\cdot\, T_{\text{mem}}$)



A = tempo de acesso à memória

T_C = tempo para copiar da memória para a cache (4 palavras por bloco)

Custo de acerto: $\approx$ ciclo do processador
penalidade por falta: $\gg$ ciclo do processador

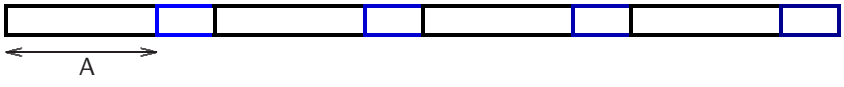
Tempo médio de acesso à memória II

Tempo de acesso não pode ser (muito) reduzido

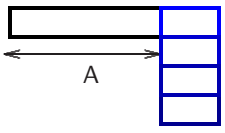
tempo de transferência depende de projeto do barramento:

* barramento estreito: uma palavra por ciclo ($\downarrow$);
* barramento/memória largos: mais de uma palavra por ciclo; ($\uparrow$);
* memória entrelaçada: acesso paralelo e transferência serializada;

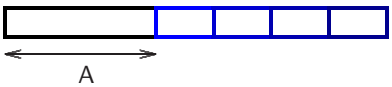
barramento estreito 4(A+1)



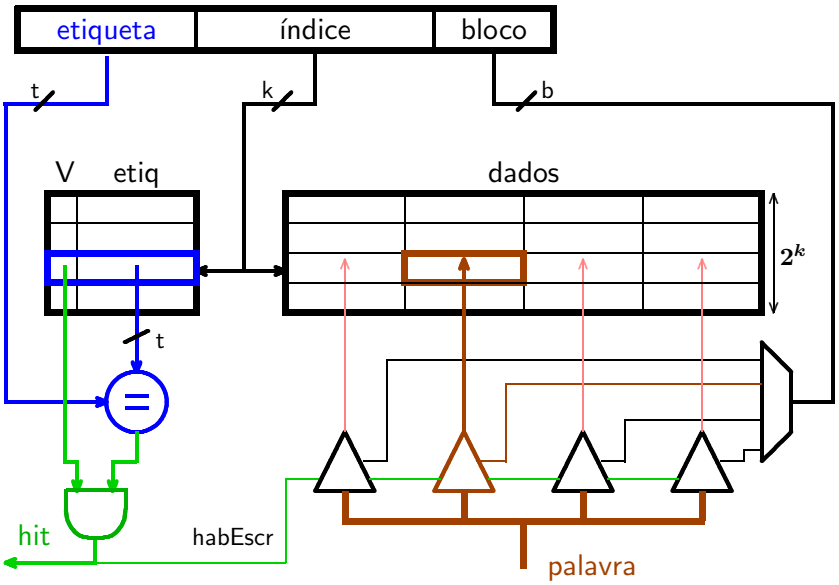
barramento largo (A+1)



memória entrelaçada (A+4)

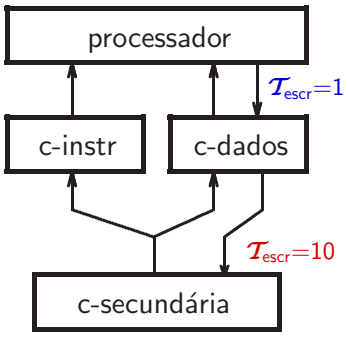


Implementação da Escrita



Políticas de escrita

- Escrita forçada (*write-through*)
  - ▷ propaga escrita até memória
  - ▷ escritas completam na velocidade da memória
- Escrita preguiçosa (*write-back*)
  - ▷ acumula escritas na cache ⟶ bit `sujo`
  - ▷ falta em leitura custa propagação até memória



Políticas de escrita

- Escrita forçada *write-through*
  - ⋆ cada escrita é propagada até a memória
  - ⋆ escritas ocorrem na velocidade da memória
- Escrita preguiçosa *write-back*
  - ⋆ acumula escritas na cache
  - ⋆ a cada escrita, o bit `sujo` é ligado
  - ⋆ bloco permanece sujo na cache até que seja substituído
  - ⋆ na substituição, se bloco vítima está sujo então o bloco inteiro é enviado para atualizar memória
  - ⋆ falta na leitura pode causar escrita do bloco sujo
- Fila de Escrita *write-buffer*
  - ⋆ imprescindível com escrita forçada
  - ⋆ com escrita forçada, fila tem uma palavra de largura
  - ⋆ com escrita preguiçosa, fila tem um bloco de largura

O que fazer quando ocorre uma falta na escrita?

- Aloca espaço na escrita *write-allocate*
  - ⋆ espaço é alocado na cache para bloco faltante, e então é atualizado
  - ⋆ se uma palavra no bloco foi atualizada, outras também o serão...
  - ⋆ se cache com escrita preguiçosa, falta provoca até duas transações:
    1) se bloco está sujo, expurga-o
    2) carrega bloco faltante
- Não aloca espaço na escrita *no-write-allocate*
  - ⋆ não é alocado espaço na cache para bloco faltante
  - ⋆ se não ocorreu falta de leitura, bloco pode não ser necessário...
  - ⋆ bloco faltante é atualizado diretamente na memória
  - ⋆ fila de escrita é imprescindível
- Combinações comuns
  - ⋆ escrita forçada & não-alocação de espaço
  - ⋆ escrita preguiçosa & alocação de espaço na escrita *fetch-on-write*

Fila de escrita I

Referências de escrita bloqueiam processador até que acesso à nível mais baixo da hierarquia complete

10% das referências são escritas;
escritas na cache secundária custam $\approx$ 10 ciclos
$\longrightarrow \mathcal{T}_{\text{mem}} = \mathbf{0.9 * 1 + 0.1 * 10 = 1.9}$ ciclos

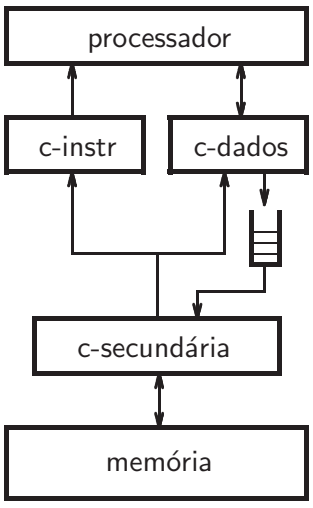
solução: fila de escritas
para desacoplar velocidade do processador da velocidade da memória

Fila de escrita II

Fila reduz tempo médio da referência de escrita

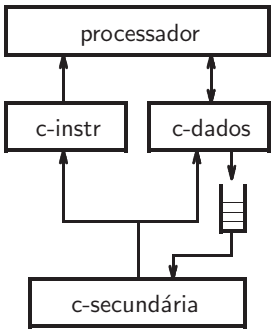
cada elemento da fila contém um par <endereço, valor>

controlador de cache efetua atualização da memória



Fila de escrita III

- Processador executa `sw $5,24($8)`
  - ▷ na fase MEM, processador insere na fila o par <(24+$8), $5>
  - ▷ se há espaço na fila, escrita completa em um ciclo
  - ▷ senão, processador bloqueia até abrir espaço na fila, enquanto escrita anterior é propagada até L2
- Fila tem capacidade para 2-16 registros
  - ▷ fila fica cheia na entrada de funções com muitos parâmetros...

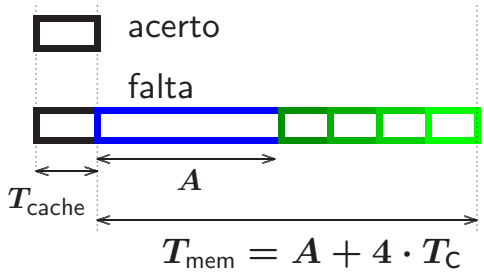


revisão – Revisão de Caches

- Localidade Temporal
  - ⋆ objeto será referenciado novamente no futuro próximo: pilha, código
- LocalidadeEspacial
  - ⋆ objetos vizinhos serão referenciados no futuro próximo: vetores, instruções
- Taxa de Acertos = núm_acertos / núm_referências
  - ⋆ Taxa de Faltas = 1 - Taxa_de_acertos
- 3 Cs: faltas compulsórias, por conflitos, por capacidade
  - ⋆ compulsórias: são compulsórias...
  - ⋆ conflitos: endereços mapeiam no mesmo bloco ↑associatividade
  - ⋆ capacidade: |conj de dados| > |cache|; ↑tamanho
- Políticas de escrita
  - ⋆ escrita forçada, escrita preguiçosa, fila de escrita
  - ⋆ nas faltas: aloca espaço, não-aloca espaço

Tempo médio de acesso à memória

$$T_{\text{med}} = T_{\text{cache}} + (\text{faltas} \cdot T_{\text{mem}})$$



A = tempo de acesso à memória

T_C = tempo para copiar da memória para a cache (4 palavras por bloco)

Custo de acerto: ≈ ciclo do processador
penalidade por falta: ≫ ciclo do processador

Desempenho de caches

$$\text{tempo de CPU} = \#\text{Instr} \times (\text{CPI}_{\text{exec}} + \text{CPI}_{\text{mem}}) \times \text{ciclo}$$

$$\text{CPI}_{\text{mem}} = \frac{\text{numRefs}}{\text{instr}} \times (T_{\text{acerto}} + \text{txFaltas} \cdot \text{penalidade})$$

Tempo médio de acesso à memória

$$T_{\text{mem}} = T_{\text{acerto}} + \text{txFaltas} \cdot \text{penalidade}$$

Melhoria no desempenho de caches

Tempo médio de acesso a memória =
tempo de acerto +
taxa de faltas x penalidade pela falta

Para melhorar o desempenho é necessário:

- **reduzir a taxa de faltas**
- reduzir a penalidade
- reduzir o tempo de acerto

Estratégia de projeto mais simples é projetar a cache primária maior possível, sem elongar o ciclo de relógio, e sem estágios adicionais no pipeline fica mais complicado com emissão fora-de-ordem

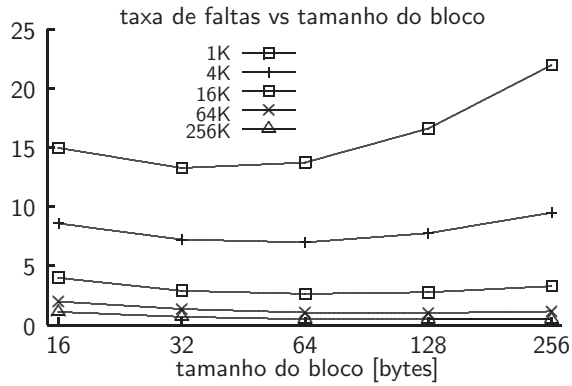
Reduzir a taxa de faltas

- **Faltas compulsórias:** primeira referência a um bloco (a frio)
  - ▷ faltas que ocorreriam numa cache infinita
- **Faltas por capacidade:** cache não comporta conjunto de trabalho
  - ▷ faltas que ocorreriam mesmo com políticas perfeitas de alocação e reposição
- **Faltas por conflito:** colisões no mapeamento de blocos na cache
  - ▷ faltas que não ocorreriam se a cache fosse totalmente associativa e reposição fosse com LRU
  - ▷ podem ocorrer com cache não-cheia

Reduzir a taxa de faltas – parâmetros de projeto

- Maior capacidade
  - ▷ reduz faltas por capacidade e por conflito
  - ▷ tempo de acerto pode aumentar
- Tamanho do bloco
  - ▷ localidade espacial reduz faltas compulsórias
  - ▷ menor número de blocos pode aumentar conflitos
  - ▷ bloco maior pode aumentar penalidade (tempo de preenchimento)
- Associatividade
  - ▷ reduz faltas por conflito (até associatividade 4-8)
  - ▷ pode aumentar tempo de acesso

1. Reduzir faltas – Tamanho do Bloco



- localidade espacial reduz faltas compulsórias
- para mesmo tamanho, menor número de blocos aumenta conflitos
- bloco maior aumenta tempo de preenchimento >penalidade

2. Reduzir faltas – Associatividade

Regra 2:1

taxa de faltas de uma cache com mapeamento direto de tamanho N

$=$

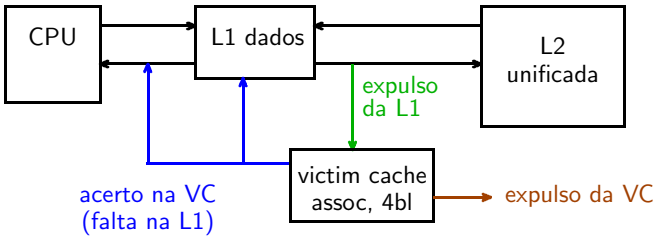
taxa de faltas de cache com associatividade binária de tamanho $N/2$

Achtung: desempenho é medido pelo tempo de execução!!
maior associatividade pode aumentar tempo de acesso

Em [Hill88] sugere-se que relação entre tempo de acerto de
assoc-binária / mapeamentoDireto é
1,02 para caches internas e
1,10 para caches externas

dados estarão disponíveis somente após comparação da etiqueta

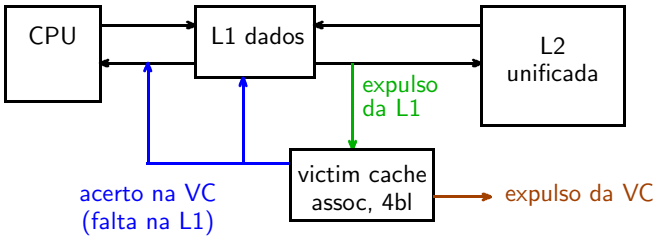
3. Reduzir faltas – Cache de Vítimas



Cache de Vítimas é uma cache associativa pequena (4-8 blocos), ligada à cache com mapeamento direto para conter linhas recentemente expurgadas

[Jouppi90]: CdV com 4 elementos remove de 20% a 95% dos conflitos numa cache de 4Kbytes com mapeamento direto; Usado no Alpha e máquinas da HP

3.1 Reduzir faltas – Cache de Vítimas



- Procura na L1;
- se falta, procura da cache de vítimas;
- se encontrou, troca linha por aquela que vai ser expulsa da L1;
- se falta na CdV, vítima da L1 ⟶ CdV;
- Efeito combinado: tempo de acesso do mapeamento direto com redução nas faltas por conflito.

4. Reduzir faltas – Busca antecipada por hardware

- Busca antecipada de instruções
  - ▷ Alpha 21064 busca dois blocos numa falta
  - ▷ bloco extra é colocado num *stream buffer*
  - ▷ numa falta, procura no *stream buffer*
- Busca antecipada de dados
  - ▷ 1 stream buffer para dados resolve 25% das faltas numa cache de 4KB; 4 stream buffers resolvem 43% [Jouppi90]
  - ▷ para aplicações científicas, 8 streams resolveram 50-70% das faltas com duas caches de 64KB, assoc-quaternária [Palacharla94]
- Busca antecipada necessita de sistema de memória com vazão sobrando e que possa ser utilisada sem penalidade

5. Reduzir faltas – Busca antecipada por software

- Busca antecipada de dados
  - ⋆ carregar dados em registrador (HP PA-RISC load)
  - ⋆ carregar dados para a cache (MIPS-4, PowerPC)
  - ⋆ instruções de busca antecipada não podem causar faltas de página
    ⟶ é uma forma de execução especulativa
- Executar instruções de busca antecipada custa tempo
  - ⋆ custo da busca antecipada é menor que os ganhos pelos acertos?
  - ⋆ processadores super-escalares gulosos facilitam despacho destas instruções
- Busca antecipada causa **poluição** na cache
  - ⋆ carregar blocos em *stream buffers* e não na cache
  - ⋆ numa falta, carrega stream buffers com blocos após o faltante
  - ⋆ numa falta, procura na cache e nos SBs
  - ⋆ 1 SB é bom para instruções $\geq$ 1 SBs para dados ($\approx$ 4..8)

6. Reduzir faltas – Otimização pelo compilador

- Instruções
  - ▷ reordenar funções na memória para reduzir faltas por conflito
  - ▷ verificar perfil de execução para descobrir conflitos,
    e então re-ligar aplicativos
- Dados
  - ▷ **agrupar vetores:** acessos a vetor de elementos compostos tem melhor localidade espacial que 2 vetores
  - ▷ **troca de loops:** mudar aninhamento dos loops para acessar dados na ordem de armazenamento em memória
  - ▷ **fusão de loops:** combinar 2 loops independentes com comportamento similar e sobreposição de variáveis
  - ▷ **blocagem:** melhorar a localidade temporal ao acessar "blocos" de dados repetidamente ao invés de percorrer linhas/colunas inteiras

Exemplo: agrupamento de vetores

```
/* Antes: 2 vetores */
int val[TAM];
int key[TAM];

/* Depois: 1 vetor de estruturas */
struct agrup {
   int val;
   int key;
}
struct agrup doisVetores[TAM];
```

Redução de conflitos entre `val` e `key`:
melhora localidade espacial

Exemplo: troca de loops

```
1  for (k=0; k < 100; k++)           /* Antes */
2     for (j=0; j < 100; j++)
3        for (i=0; i < 5000; i++)
4           x[i][j] = 2 * x[i][j];  /* |passo|=100 */

5  for (k=0; k < 100; k++)           /* Depois */
6     for (i=0; i < 5000; i++)
7        for (j=0; j < 100; j++)
8           x[i][j] = 2 * x[i][j];  /* |passo|=1 */
```

Acessos seqüenciais ao invés de passadas de 100 palavras

⟶ melhora localidade espacial

passada ≡ *stride*

Exemplo: fusão de loops

```
/* Antes: duas faltas por acesso em a e em c */
1  for (i=0; i < N; i++)
2     for (j=0; j < N; j++)
3        a[i][j] = 1/b[i][j] * c[i][j];
4  for (i=0; i < N; i++)
5     for (j=0; j < N; j++)
6        d[i][j] = a[i][j] + c[i][j];

/* Depois: uma falta por acesso em a e em c */
7  for (i=0; i < N; i++)
8     for (j=0; j < N; j++) {
9        a[i][j] = 1/b[i][j] * c[i][j];
a        d[i][j] = a[i][j] + c[i][j];
b  }        /* re-uso e melhor localidade espacial */
```

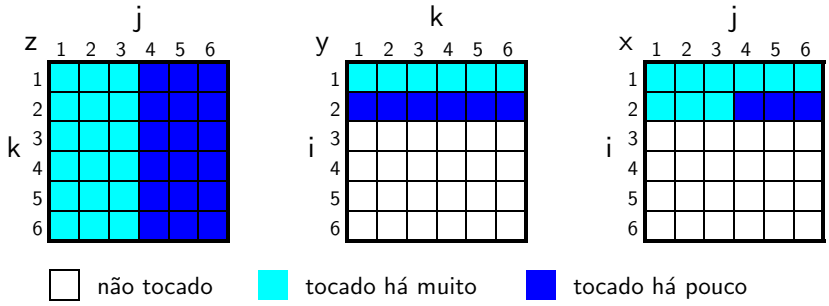

Exemplo: blocagem

```
1  for (i=0; i < N; i++)              /* Antes */
2     for (j=0; j < N; j++) {
3        for (r=0.0, k=0; k < N; k++)
4           r = r + y[i][k] * z[k][j];
5        x[i][j] = r;
6  }
```

- Dois loops internos:
  - ⋆ lê todos os NxN elementos de `z[]`
  - ⋆ lê N elementos de uma linha de `y[]` repetidamente
  - ⋆ escreve N elementos de 1 coluna de `x[]`
- Faltas por capacidade são função de N e tamanho da cache
  - ⋆ núm faltas = $2N^3 + N^2$, se não ocorrerem conflitos...
- Idéia: computar sub-matriz que cabe na cache

Exemplo: blocagem

```
1  for (i=0; i < N; i++)                    /* Antes */
2     for (j=0; j < N; j++) {
3        for (r=0.0, k=0; k < N; k++)
4           r = r + y[i][k] * z[k][j];
5        x[i][j] = r;
6  }
```



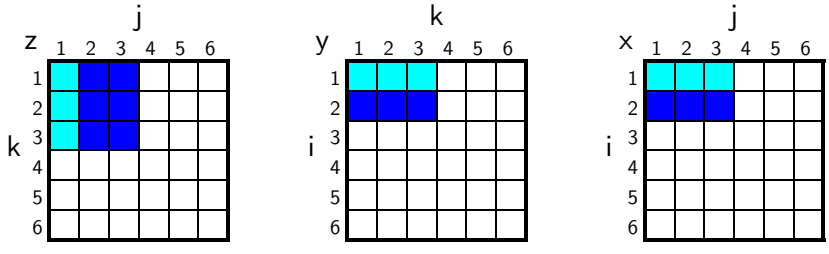
Exemplo: blocagem

```
1  for (jj=0; jj < N; jj = jj+B)            /* Depois */
2  for (kk=0; kk < N; kk = kk+B)
3  for (i=0; i < N; i++)
4     for (j = jj; j < min(jj+B,N); j++) {
5        for (r = 0.0, k = kk; k < min(kk+B,N); k++)
6           r = r + y[i][k] * z[k][j];
7        x[i][j] = x[i][j] + r;              /* x[i][j] parcial */
8     }
```

- **B** é chamado de *fator de blocagem*
- Faltas por capacidade caem de $2N^3 + N^2$ para $2N^3/B + N^2$
- Faltas por conflito talvez diminuem

Exemplo: blocagem

```
1  for (jj=0; jj < N; jj = jj+B)            /* Depois */
2  for (kk=0; kk < N; kk = kk+B)
3  for (i=0; i < N; i++)
4     for (j = jj; j < min(jj+B,N); j++) {
5        for (r = 0.0, k = kk; k < min(kk+B,N); k++)
6           r = r + y[i][k] * z[k][j];
7        x[i][j] = x[i][j] + r; }            /* x[i][j] parcial */
```



Melhoria no desempenho de caches

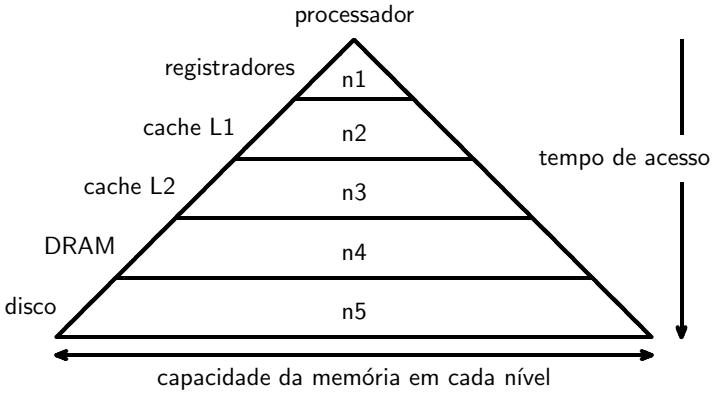
Tempo médio de acesso a memória =
tempo de acerto +
taxa de faltas x penalidade pela falta

Para melhorar o desempenho é necessário:

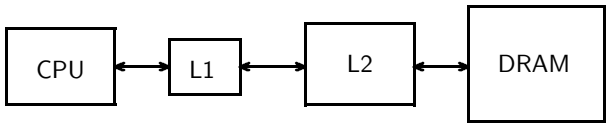
- reduzir a taxa de faltas
- **reduzir a penalidade nas faltas**
- reduzir o tempo de acerto

A. Reduzir a penalidade – hierarquia de caches

Memória não pode ser simultaneamente grande e rápida;
⟶ hierarquia com caches maiores mais longe do processador



A.a Reduzir a penalidade – hierarquia de caches

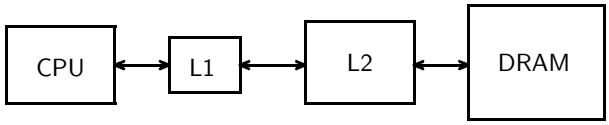


Taxa de faltas local = faltas na cache / acessos à cache
Taxa de faltas global = faltas na cache / referências pelo processador

$$\begin{aligned} T_{mem} &= tA_{L1} + F_{L1} \times P_{L1} \\ &= tA_{L1} + F_{L1} \times [tA_{L2} + F_{L2} \times P_{DRAM}] \end{aligned}$$

onde tA é o tempo de acerto,
F é taxa de faltas, e
P é a penalidade por falta

A.b Reduzir a penalidade – hierarquia de caches



Propriedade de Inclusão:
Conteúdo das caches maiores *inclui* o conteúdo das menores
⟶ cache menor contém cópias dos dados na cache maior

- ▷ qual a relação entre associatividade, tamanho e inclusão?
- ▷ qual a relação entre bits de tag/índice e assoc, tamanho e inclusão?
- ▷ [Baer,Wang88] *On the Inclusion Properties for Multi-Level Cache Hierarchies*

Caches *exclusivas* trocam linhas numa falta
⟶ L1 ⟵ novo; L2 ⟵ velho

B. Red penalidade – prioridade para faltas de leitura

- Escrita forçada com fila de escrita causa risco RAW
  - ▷ processador tenta ler dado que está na fila de escrita
  - ▷ solução ruim: espera até que fila esvazie e então continua LOAD
    ⟶ aumenta penalidade nas faltas de leitura
  - ▷ solução melhor: verifica se ∃m dependências;
    se não, LOAD prossegue sem bloquear
- Escrita preguiçosa: falta na leitura substitui bloco sujo
  - ▷ solução ruim: escreve bloco sujo e então busca bloco faltante
  - ▷ solução melhor:
    1. escreve bloco sujo num buffer/fila,
    2. lê bloco faltante e entrega ao processador,
    3. então escreve bloco sujo
  - ▷ + rápido porque CPU prossegue assim que ler bloco faltante

C. Reduzir a penalidade – minimizar tempo de carga

- Tempo de preenchimento do bloco é longo
- ...mas não precisa esperar até que bloco seja preenchido
- **Early restart** – assim que palavra requisitada chegar da memória, entrega ao processador, que continua a executar
- **Critical word first** – busca palavra requisitada primeiro e a entrega ao processador assim que chegar da memória
  - ▷ processador continua enquanto bloco está sendo preenchido
  - ▷ técnica é útil com blocos grandes
  - ▷ processador requisita **p2**:

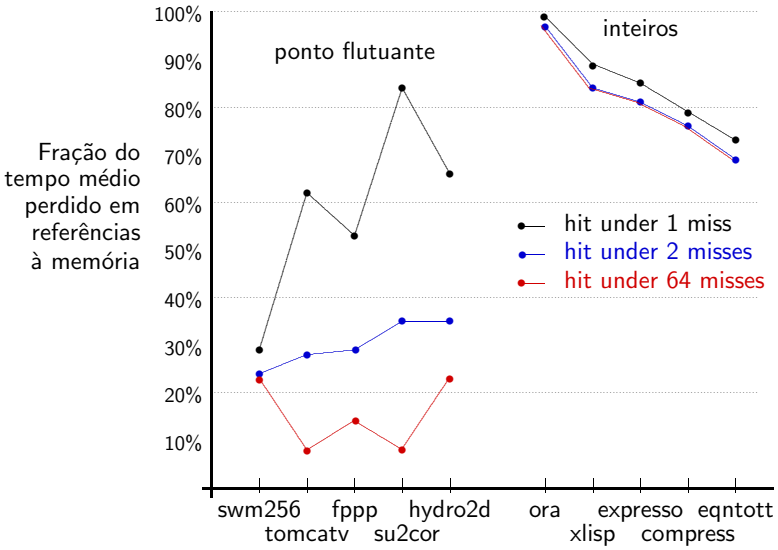
p0	p1	**p2**	p3

  - ▷ **p2** é entregue, e bloco é preenchido com **p2** ⟶ p3 ⟶ p0 ⟶ p1
  - ▷ localidade espacial indica que próximo acesso será
    na palavra seguinte, que pode não ter chegado ainda

D. Reduzir a penalidade – cache não-bloqueante

- Cache não-bloqueante entrega dados enquanto trata de uma falta
  - ▷ processador com execução fora-de-ordem
  - ▷ memória com múltiplos bancos ⟶ cache & DRAM
  - ▷ [Kroft81] *Lock-up Free Instruction Fetch/Prefetch Cache Organization*
- **acerto sob falta** (*hit under miss*) reduz penalidade efetiva porque cache fornece dados, ao invés de bloquear processador na falta
- **acerto sob múltiplas faltas** (*miss under miss*) reduz penalidade ainda mais ao sobrepor tratamento de múltiplas faltas
  - ▷ complexidade do controlador da cache é MUITO maior por causa dos acessos concomitantes à memória
  - ▷ necessita de memória com múltiplos bancos
  - ▷ PentiumPro permite até 4 faltas pendentes

D.1 Reduzir a penalidade – cache não-bloqueante



Melhoria no desempenho de caches

Tempo médio de acesso a memória =
tempo de acerto +
taxa de faltas x penalidade pela falta

Para melhorar o desempenho é necessário:

- reduzir a taxa de faltas
- reduzir a penalidade nas faltas
- **reduzir o tempo de acerto**

I. Reduzir o tempo de acerto – caches simples e pequenas

Tempo de acerto curto se:

- ▷ cache pequena ⟶ reduz tempo de acesso
- ▷ cache simples ⟶ mapeamento direto

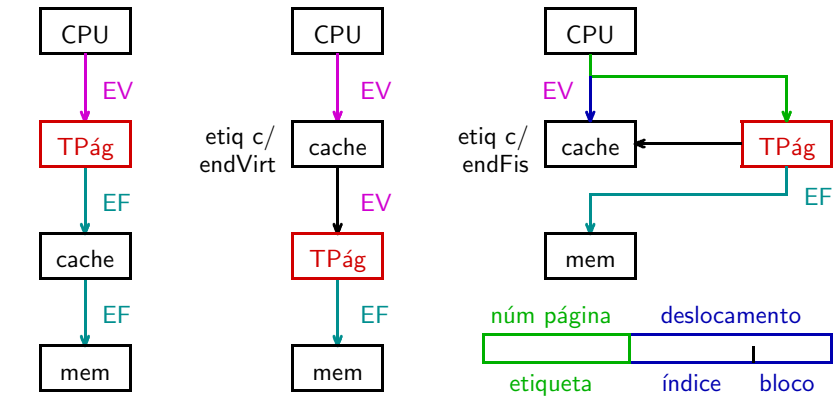
Alpha 21164 tem 8KB para dados e 8KB para instruções, 96KB cache secundária, unificada, associatividade ternária, e relógio muito rápido!

II. Reduzir o tempo de acerto – evitar tradução de endereços

- Cache endereçada com: **endereço virtual** vs **endereço físico**
- a cada troca de contexto deve expurgar a cache
  - ⋆ senão ocorrem falsos-acertos
  - ⋆ processo que entra no processador tem montes de faltas compulsórias
  - ⋆ conseqüência da troca de contexto pode durar 10^5 ciclos
  - ⋆ solução: usar **identificador de processo** na etiqueta na cache
- como trata de sinônimos?
  - ⋆ dois endereços virtuais que mapeiam no mesmo endereço físico
  - ⋆ solução: garantir que indexação da cache ocorre com bits que coincidem no EV e EF

EVirtual	núm página virtual	deslocamento
ender na cache	|—**índice**—|	
EFísico	núm página física	deslocamento

II.i Reduzir T de acerto – evitar tradução de ender



organização convencional

cache ender virtual
traduz somente nas faltas

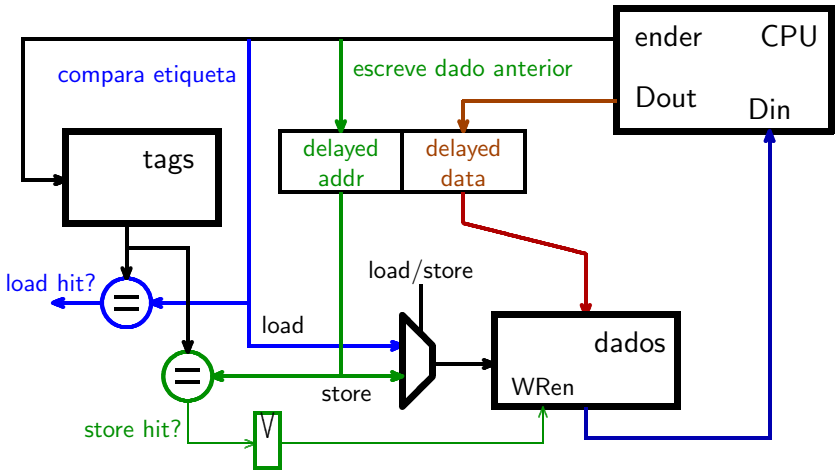
sobrepõe indexação com tradução EV-EF
índice não pode mudar durante tradução
|cache| = |página|

III.i Reduzir o tempo de acerto – escritas em pipeline

Escrita consome dois ciclos

- um para comparar etiqueta e outro para escrever na cache
- armazena dados e endereço em buffer
- escreve dados na cache durante comparação de etiqueta na próxima escrita
- risco RAW entre leitura e escrita atrasada ⟶ adiantamento

III.ii Reduzir tempo de acerto – escritas em pipeline



Resumo – melhorar desempenho de caches

T_méd_acesso_mem = T_acerto + tx_faltas x penalidade/falta

Para melhorar o desempenho é necessário:

★ **reduzir a taxa de faltas**
- ▷ parâmetros de projeto: núm,tam blocos, associatividade
- ▷ cache de vítimas, stream buffers, busca antecipada
- ▷ otimização do código: agrupamento, troca de índices, fusão, blocagem

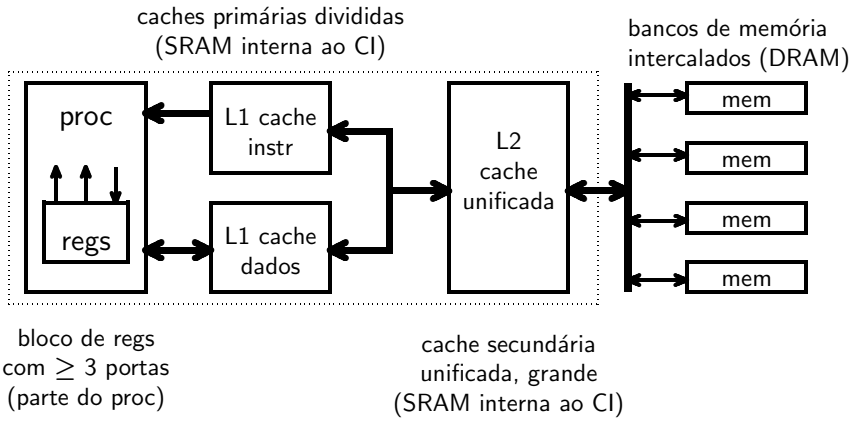
★ **reduzir a penalidade nas faltas**
- ▷ hierarquia de caches
- ▷ priorizar faltas na leitura (critical word first, early restart)
- ▷ cache não-bloqueante

★ **reduzir o tempo de acerto**
- ▷ caches simples e pequenas
- ▷ não traduzir endereço virtual para físico
- ▷ escritas em pipeline

Cache

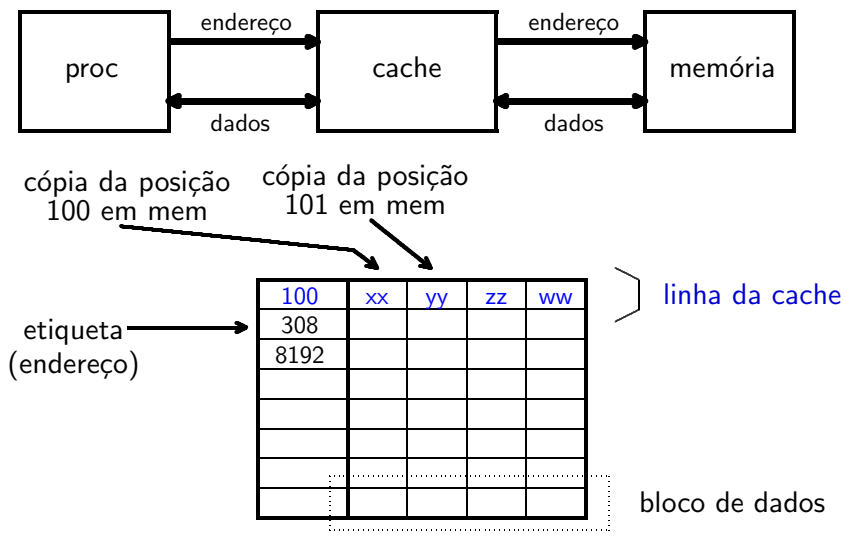
- Sistemas de memória
- **Memória cache**
  - ⋆ **organização**
  - ⋆ **leitura**
  - ⋆ **escrita**
  - ⋆ **otimizações**
  - ⋆ **busca antecipada**
- Busca antecipada
- Memória Virtual
  - ⋆ endereçamento
  - ⋆ paginação
  - ⋆ TLB
  - ⋆ excessões

Sistemas de Memória Típicos



"Lei de Less": Processador super-escalar pode executar
$\approx 10^3$ instruções durante falta na cache

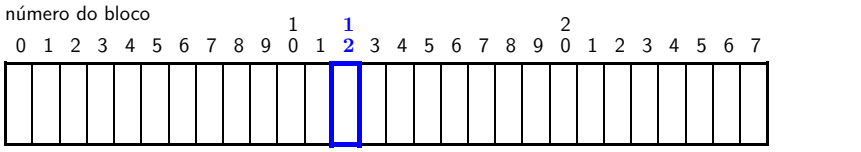
Cache por dentro

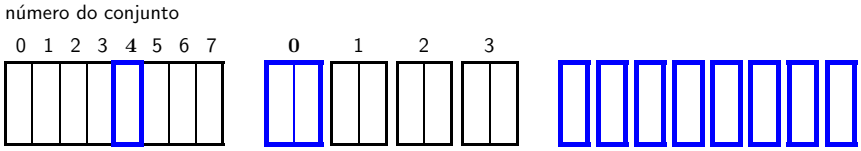


Algoritmo para leitura

cache recebe endereço;
se objeto na cache (acerto hit)
    entrega para processador;
senão (falta miss)
    lê bloco da memória
        espera
    entrega ao processador
    e atualiza cache

Alocação de blocos





mapeamento direto

bloco 12 mapeia somente no bloco 4
12 % 8 = 4

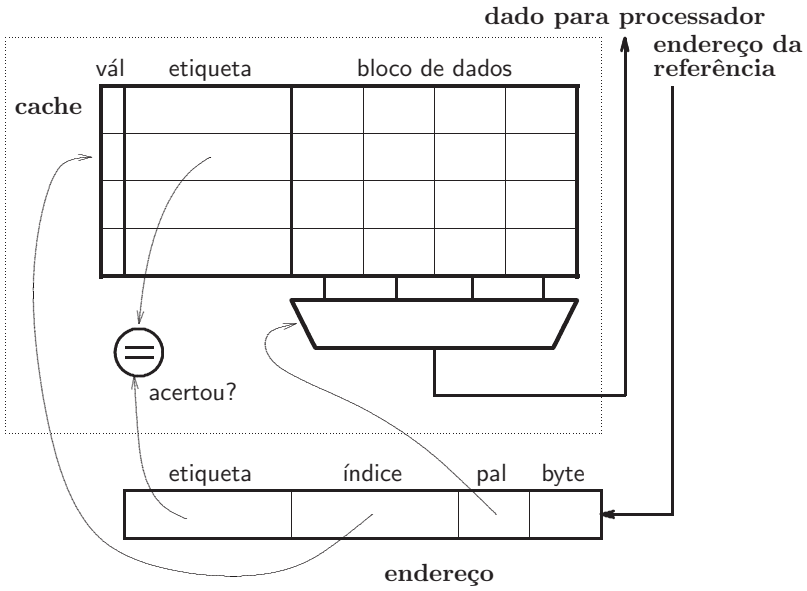
associativa por conjuntos (binária)

bloco 12 mapeia no conjunto 0
12 % 4 = 0

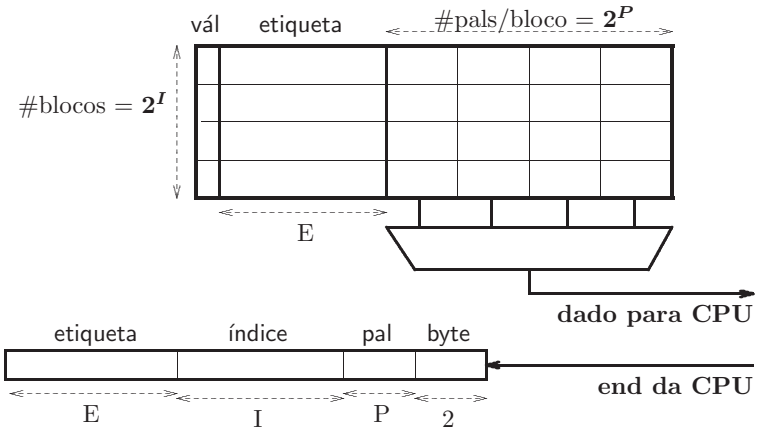
associativa (total)

bloco 12 mapeia em qualquer conjunto

Mapeamento Direto



Organização da cache III



tamanho da cache = num_blocos * tam_bloco * tam_palavra

Quanto mais blocos $\Longrightarrow$ menor a taxa de faltas

Blocos com 4-16 palavras tiram vantagem de localidade espacial

Associatividade II

M[12] = 99; M[13] = 200; M[14] = 33

Associativa - 4 conjuntos

0		99		200		33		

aloc: 12%1 = ?

Associativa binária - 2 conjuntos

0		99		33
1		200		

aloc: 12%2 = 0

aloc: 13%2 = 1

Mapeam direto - 1 conjunto

0	99	aloc: 12%4 = 0
1	200	aloc: 13%4 = 1
2	33	aloc: 14%4 = 2
3		aloc: N%4 = 3

Escolha de vítima para reposição: com mapeamento direto $\not\exists$ escolha...

1) escolhe o bloco usado no passado mais distante (LRU)

2) escolhe a esmo

Taxas de Faltas e de Acertos

Se encontra na cache — **acerto**

Se não encontra na cache — **falta**

$$\text{taxa de acertos} \triangleq \frac{\text{número de acertos}}{\text{número de referências}}$$

$$\text{taxa de faltas} \triangleq 1 - \text{taxa de acertos}$$

Pergunta:

Qual a taxa de faltas se |conj_dados| $\gg$ capacidade_da_cache?

(a) baixa
(b) média
(c) alta
(d) faltas independem dos tamanhos relativos

Resposta:

Qual a taxa de faltas se |conj_dados| $\gg$ capacidade_da_cache?

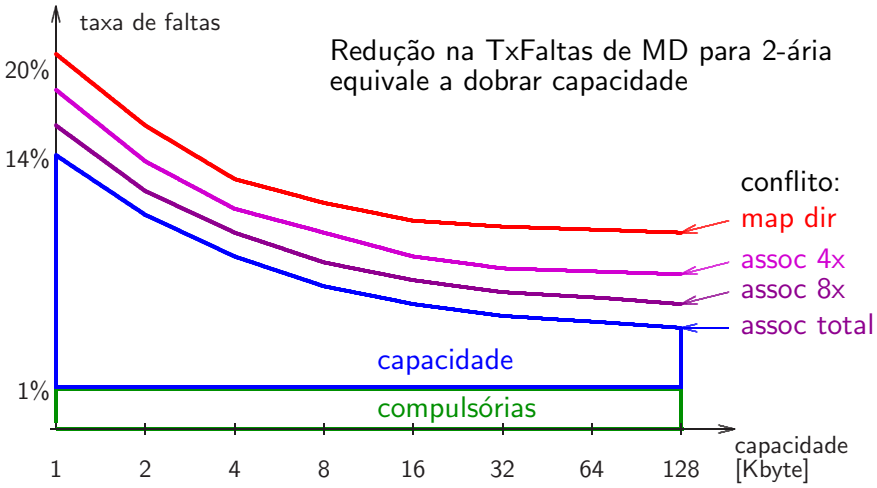
(a) baixa
(b) média
(c) alta
(d) faltas independem dos tamanhos relativos

Qual a taxa de faltas se |conj_dados| $\ll$ capacidade_da_cache?

Causas de faltas

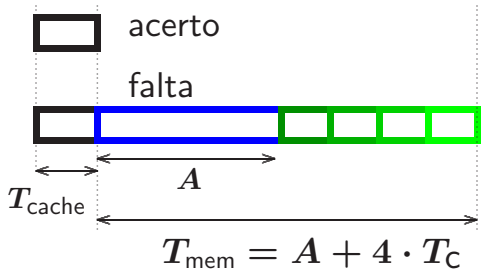
- faltas compulsórias – linhas são tocadas pela primeira vez.
  - ⋆ Solução: blocos com muitas palavras para fazer busca antecipada implícita;
- faltas por capacidade – conj. de trabalho $\gg$ tamanho da cache.
  - ⋆ Solução: aumentar tamanho da cache;
- faltas por conflito – mapeamento de endereços no mesmo bloco da cache.
  - ⋆ Solução: aumentar associatividade.
  - ⋆ Faltas por conflito ocorrem mesmo quando a cache não está cheia; estas se devem aos conflitos de mapeamento nos blocos da cache

Causas de faltas



Tempo médio de acesso à memória I

- 1 ciclo por acerto na cache (tCache)
- penalidade: 2..100 ciclos para acessar memória (tMem)
- tMed = tCache ∗ (acertos + faltas ∗ tMem)



A = tempo de acesso à memória

T_C = tempo para copiar da memória para a cache (4 palavras por bloco)

Custo de acerto: ≈ ciclo do processador
penalidade por falta: ≫ ciclo do processador

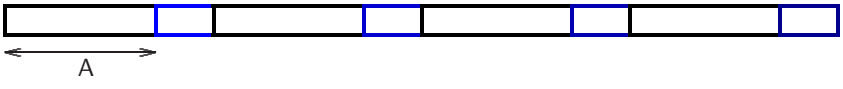
Tempo médio de acesso à memória II

Tempo de acesso não pode ser (muito) reduzido

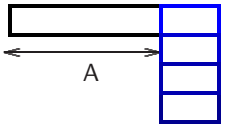
tempo de transferência depende de projeto do barramento:

⋆ barramento estreito: uma palavra por ciclo ($↓);
⋆ barramento/memória largos: mais de uma palavra por ciclo; ($↑);
⋆ memória entrelaçada: acesso paralelo e transferência serializada;

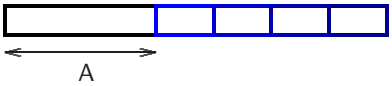
barramento estreito 4(A+1)



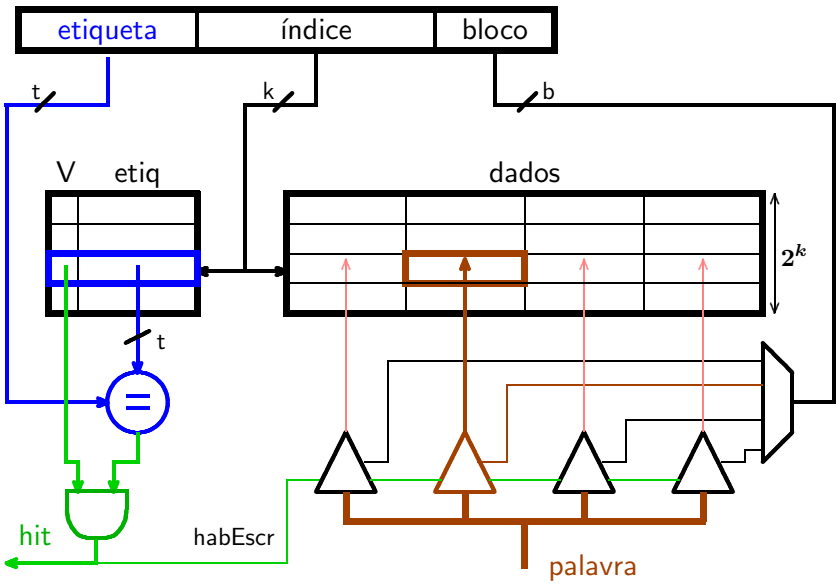
barramento largo (A+1)



memória entrelaçada (A+4)

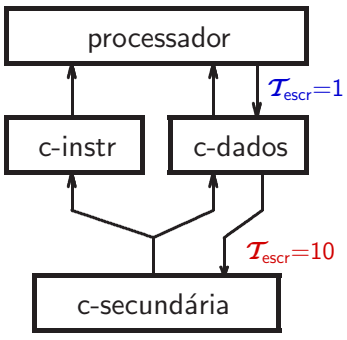


Implementação da Escrita



Políticas de escrita

- Escrita forçada (*write-through*)
  - ⋆ propaga escrita até memória
  - ⋆ escritas com velocidade da memória
- Escrita preguiçosa (*write-back*)
  - ⋆ acumula escritas na cache ⟶ bit `sujo`
  - ⋆ falta em leitura custa propagação até memória



Políticas de escrita

- Escrita forçada *write-through*
  - ⋆ cada escrita é propagada até a memória
  - ⋆ escritas ocorrem na velocidade da memória
- Escrita preguiçosa *write-back*
  - ⋆ acumula escritas na cache
  - ⋆ a cada escrita, o bit `sujo` é ligado
  - ⋆ bloco permanece sujo na cache até que seja substituído
  - ⋆ na substituição, se bloco vítima está sujo então bloco inteiro enviado para atualizar memória
  - ⋆ falta na leitura pode causar escrita do bloco sujo
- Fila de Escrita *write-buffer*
  - ⋆ imprescindível com escrita forçada
  - ⋆ com escrita forçada, fila tem uma palavra de largura
  - ⋆ com escrita preguiçosa, fila tem um bloco de largura

O que fazer quando ocorre uma falta na escrita?

- Aloca espaço na escrita *write-allocate*
  - ⋆ espaço é alocado na cache para bloco faltante, e então é atualizado
  - ⋆ se uma palavra no bloco foi atualizada, outras também o serão...
  - ⋆ se cache com escrita preguiçosa, falta provoca até duas transações:
    1) se bloco está sujo, expurga-o
    2) carrega bloco faltante
- Não aloca espaço na escrita *no-write-allocate*
  - ⋆ não é alocado espaço na cache para bloco faltante
  - ⋆ se não ocorreu falta de leitura, bloco pode não ser necessário...
  - ⋆ bloco faltante é atualizado diretamente na memória
  - ⋆ fila de escrita é imprescindível
- Combinações comuns
  - ⋆ escrita forçada & não-alocação de espaço
  - ⋆ escrita preguiçosa & alocação de espaço na escrita *fetch-on-write*

Fila de escrita I

Referências de escrita bloqueiam processador até que acesso à nível mais baixo da hierarquia complete

10% das referências são escritas;
escritas na cache secundária custam $\approx$ 10 ciclos
$\Longrightarrow T_{\text{mem}} = 0.9 * 1 + 0.1 * 10 = 1.9$ ciclos

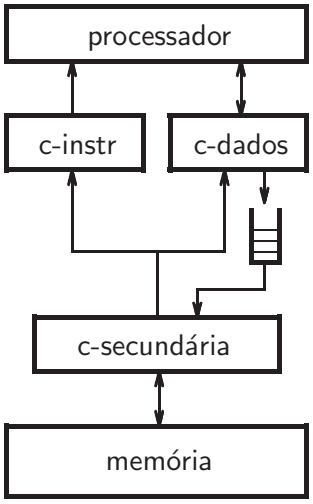
solução: fila de escritas
para desacoplar velocidade do processador da velocidade da memória

Fila de escrita II

Fila reduz tempo médio da referência de escrita

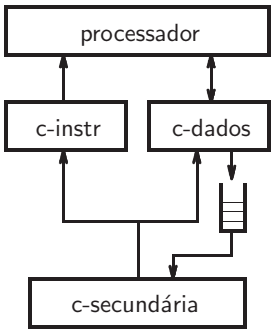
cada elemento da fila contém um par <endereço, valor>

controlador de cache efetua atualização da memória



Fila de escrita III

- Processador executa `sw $5,24($8)`
  - ⋆ na fase MEM, processador insere na fila par <(24+$8), $5>
  - ⋆ se há espaço na fila, escrita completa em um ciclo
  - ⋆ senão, processador bloqueia até abrir espaço na fila, enquanto escr anterior é propagada até L2
- Fila tem capacidade para 2-16 registros
  - ⋆ fila enche na entrada de funções com muitos parâmetros...

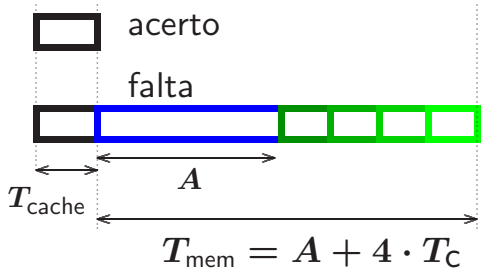


revisão – Revisão de Caches

- Localidade Temporal
  - ⋆ objeto será referenciado novamente no futuro próximo: pilha, código
- LocalidadeEspacial
  - ⋆ objetos vizinhos serão referenciados no futuro próximo: vetores, instruções
- Taxa de Acertos = núm_acertos / núm_referências
  - ⋆ Taxa de Faltas = 1 - Taxa_de_acertos
- 3 Cs: faltas compulsórias, por conflitos, por capacidade
  - ⋆ compulsórias: são compulsórias...
  - ⋆ conflitos: endereços mapeiam no mesmo bloco ↑associatividade
  - ⋆ capacidade: |conj de dados| > |cache|; ↑tamanho
- Políticas de escrita
  - ⋆ escrita forçada, escrita preguiçosa, fila de escrita
  - ⋆ nas faltas: aloca espaço, não-aloca espaço

Tempo médio de acesso à memória

$$T_{\text{med}} = T_{\text{cache}} * (\text{acertos} + \text{faltas} * T_{\text{mem}})$$



A = tempo de acesso à memória

T_C = tempo para copiar da memória para a cache (4 palavras por bloco)

Custo de acerto: $\approx$ ciclo do processador
penalidade por falta: $\gg$ ciclo do processador

Desempenho de caches

$$\text{tempo de CPU} = \#\text{Instr} \times (\text{CPI}_{\text{exec}} + \text{CPI}_{\text{mem}}) \times \text{ciclo}$$

$$\text{CPI}_{\text{mem}} = \frac{\text{numRefs}}{\text{instr}} \times (T_{\text{acerto}} + \text{txFaltas} \cdot \text{penalidade})$$

Tempo médio de acesso à memória

$$T_{\text{mem}} = T_{\text{acerto}} + \text{txFaltas} \cdot \text{penalidade}$$

Melhoria no desempenho de caches

Tempo médio de acesso a memória =
tempo de acerto +
taxa de faltas x penalidade pela falta

Para melhorar o desempenho é necessário:

- **reduzir a taxa de faltas**
- reduzir a penalidade
- reduzir o tempo de acerto

Estratégia de projeto mais simples é projetar a cache primária maior possível, sem elongar o ciclo de relógio, e sem estágios adicionais no pipeline fica mais complicado com emissão fora-de-ordem

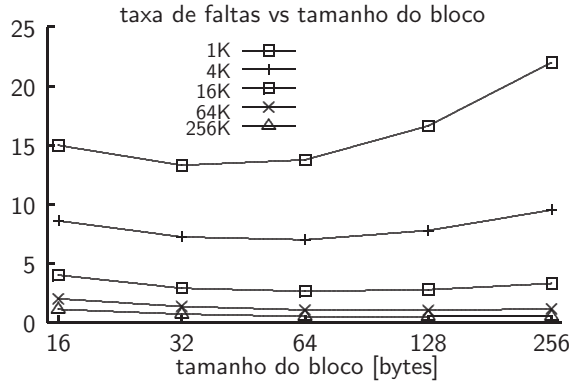
Reduzir a taxa de faltas

- **Faltas compulsórias:** primeira referência a um bloco (partida a frio)
  - ▷ faltas que ocorreriam numa cache infinita
- **Faltas por capacidade:** cache não comporta conjunto de trabalho
  - ▷ faltas que ocorreriam mesmo com políticas perfeitas de alocação e reposição
- **Faltas por conflito:** colisões no mapeamento de blocos na cache
  - ▷ faltas que não ocorreriam se a cache fosse totalmente associativa e reposição fosse com LRU
  - ▷ podem ocorrer com cache não-cheia

Reduzir a taxa de faltas – parâmetros de projeto

- Maior capacidade
  - ▷ reduz faltas por capacidade e por conflito
  - ▷ tempo de acerto pode aumentar
- Tamanho do bloco
  - ▷ localidade espacial reduz faltas compulsórias
  - ▷ menor número de blocos pode aumentar conflitos
  - ▷ bloco maior pode aumentar penalidade (tempo de preenchimento)
- Associatividade
  - ▷ reduz faltas por conflito (até associatividade 4-8)
  - ▷ pode aumentar tempo de acesso

1. Reduzir faltas – Tamanho do Bloco



- localidade espacial reduz faltas compulsórias
- para mesmo tamanho, menor número de blocos aumenta conflitos
- bloco maior aumenta tempo de preenchimento >penalidade

2. Reduzir faltas – Associatividade

Regra 2:1

taxa de faltas de uma cache com mapeamento direto de tamanho N

=

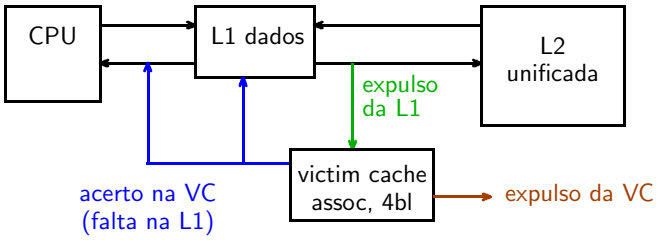
taxa de faltas de cache com associatividade binária de tamanho $N/2$

Achtung: desempenho é medido pelo tempo de execução!!
maior associatividade pode aumentar tempo de acesso

Em [Hill88] sugere-se que relação entre tempo de acerto de assoc-binária / mapeamentoDireto é
1,02 para caches internas e
1,10 para caches externas

lembre que dados estão disponíveis só após comparação da etiqueta

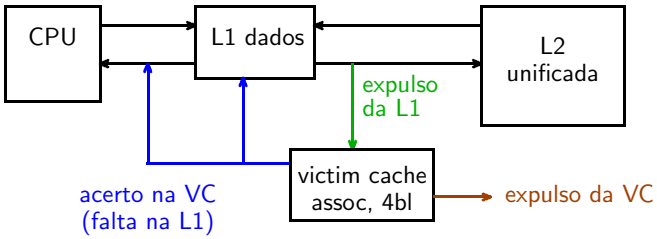
3. Reduzir faltas – Cache de Vítimas



Cache de Vítimas é uma cache pequena (4-8 blocos), associativa, ligada à cache com mapeamento direto para conter linhas recentemente expurgadas

[Jouppi90]: CdV com 4 elementos remove de 20% a 95% dos conflitos numa cache de 4Kbytes com mapeamento direto;
Usado no Alpha e máquinas da HP

3.1 Reduzir faltas – Cache de Vítimas



- Procura na L1;
- se falta, procura da cache de vítimas;
- se encontrou, troca linha por aquela que vai ser expulsa da L1;
- se falta na CdV, vítima da L1 $\longrightarrow$ CdV;
- Efeito combinado: tempo de acesso do mapeamento direto com redução nas faltas por conflito.

4. Reduzir faltas – Busca antecipada por hardware

- Busca antecipada de instruções
  - ⋆ Alpha 21064 busca dois blocos numa falta
  - ⋆ bloco extra é colocado num *stream buffer*
  - ⋆ numa falta, procura no *stream buffer*
- Busca antecipada de dados
  - ⋆ 1 *stream buffer* para dados resolve 25% das faltas numa cache de 4KB; 4 stream buffers resolvem 43% [Jouppi90]
  - ⋆ para aplicações científicas, 8 streams resolveram 50-70% das faltas com duas caches de 64KB, assoc-quaternária [Palacharla94]
- Busca antecipada necessita de sistema de memória com vazão sobrando e que possa ser usada sem penalidade

5. Reduzir faltas – Busca antecipada por software

- Busca antecipada de dados
  - ⋆ carregar dados em registrador (HP PA-RISC load)
  - ⋆ carregar dados para a cache (MIPS-4, PowerPC)
  - ⋆ instruções de busca antecipada não podem causar faltas de página → forma de execução especulativa
- Executar instruções de busca antecipada custa tempo
  - ⋆ custo da busca antecipada é menor que os ganhos pelos acertos?
  - ⋆ processadores super-escalares gulosos facilitam despacho destas instruções
- Busca antecipada causa **poluição** na cache
  - ⋆ carregar blocos em *stream buffers* e não na cache
  - ⋆ numa falta, carrega stream buffers com blocos após o faltante
  - ⋆ numa falta, procura na cache e nos SBs
  - ⋆ $\mathbf{1}$ SB é bom para instruções $\geq \mathbf{1}$ SBs para dados ($\approx \mathbf{4..8}$)

6. Reduzir faltas – Otimização pelo compilador

- Instruções
  - ⋆ reordenar funções na memória para reduzir faltas por conflito
  - ⋆ verificar perfil de execução para descobrir conflitos, e então re-ligar aplicativos
- Dados
  - ⋆ **agrupar vetores:** vetor de elmtos compostos tem melhor localidade espacial que 2 vetores
  - ⋆ **troca de loops:** mudar aninhamento dos loops para acessar dados na ordem de armazenamento em memória
  - ⋆ **fusão de loops:** combinar 2 loops independentes com comportamento similar e sobreposição de variáveis
  - ⋆ **blocagem:** melhorar a localidade temporal ao acessar "blocos" de dados repetidamente ao invés de percorrer linhas/colunas inteiras

Exemplo: agrupamento de vetores

```
/* Antes: 2 vetores */
int val[TAM];
int key[TAM];

/* Depois: 1 vetor de estruturas */
struct agrup {
   int val;
   int key;
}
struct agrup doisVetores[TAM];
```

Redução de conflitos entre `val` e `key`:
melhora localidade espacial

Exemplo: troca de loops

```
1  for (k=0; k < 100; k++)          /* Antes */
2     for (j=0; j < 100; j++)
3        for (i=0; i < 5000; i++)
4           x[i][j] = 2 * x[i][j];  /* passo=100 */

5  for (k=0; k < 100; k++)          /* Depois */
6     for (i=0; i < 5000; i++)
7        for (j=0; j < 100; j++)
8           x[i][j] = 2 * x[i][j];  /* passo=1 */
```

Acessos seqüenciais ao invés de passadas de 100 palavras
$\Longrightarrow$ melhora localidade espacial passada == *stride*

Exemplo: fusão de loops

```
/* Antes: duas faltas por acesso em a e em c */
1  for (i=0; i < N; i++)
2     for (j=0; j < N; j++)
3        a[i][j] = 1/b[i][j] * c[i][j];
4  for (i=0; i < N; i++)
5     for (j=0; j < N; j++)
6        d[i][j] = a[i][j] + c[i][j];

/* Depois: uma falta por acesso em a e em c */
7  for (i=0; i < N; i++)
8     for (j=0; j < N; j++) {
9        a[i][j] = 1/b[i][j] * c[i][j];
A        d[i][j] = a[i][j] + c[i][j];
B  }        /* re-uso e melhor localidade espacial */
```

Exemplo: blocagem

```
1  for (i=0; i < N; i++)                /* Antes */
2     for (j=0; j < N; j++) {
3        for (r=0, k=0; k < N; k++)
4           r = r + y[i][k] * z[k][j];
5        x[i][j] = r;
6  }
```

- Dois loops internos:
  - ⋆ lê todos os NxN elementos de `z[]`
  - ⋆ lê N elementos de uma linha de `y[]` repetidamente
  - ⋆ escreve N elementos de 1 coluna de `x[]`
- Faltas por capacidade são função de N e tamanho da cache
  - ⋆ núm faltas $= 2N^3 + N^2$, se não ocorrerem conflitos...
- Idéia: computar sub-matriz que cabe na cache

Exemplo: blocagem

```
1  for (i=0; i < N; i++)                /* Antes */
2     for (j=0; j < N; j++) {
3        for (r=0, k=0; k < N; k++)
4           r = r + y[i][k] * z[k][j];
5        x[i][j] = r;
6  }
```

H&P QA Fig-5.21

Exemplo: blocagem

```
1  for (jj=0; jj < N; jj = jj+B)       /* Depois */
2  for (kk=0; kk < N; kk = kk+B)
3  for (i=0; i < N; i++)
4     for (j = jj; j < min(jj+B-1 , N); j++) {
5        for (r = 0, k = kk; k < min(kk+B-1, N); k++)
6           r = r + y[i][k] * z[k][j];
7        x[i][j] = r;
8  }
```

- **B** é chamado de *fator de blocagem*
- Faltas por capacidade caem de $2N^3 + N^2$ para $2N^3/B + N^2$
- Faltas por conflito talvez diminuem

Exemplo: blocagem

```
1  for (jj=0; jj < N; jj = jj+B)       /* Depois */
2  for (kk=0; kk < N; kk = kk+B)
3  for (i=0; i < N; i++)
4     for (j = jj; j < min(jj+B-1 , N); j++) {
5        for (r = 0, k = kk; k < min(kk+B-1, N); k++)
6           r = r + y[i][k] * z[k][j];
7        x[i][j] = r;
8  }
```

H&P QA Fig-5.22

Melhoria no desempenho de caches

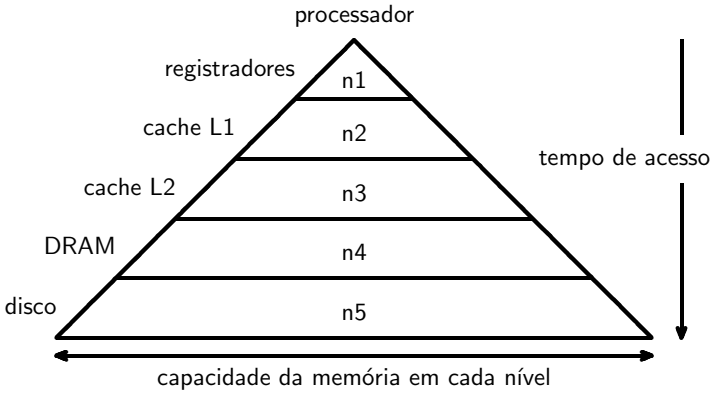
Tempo médio de acesso a memória =
tempo de acerto +
taxa de faltas x penalidade pela falta

Para melhorar o desempenho é necessário:

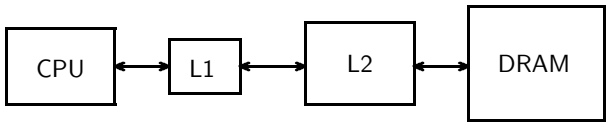
- reduzir a taxa de faltas
- **reduzir a penalidade nas faltas**
- reduzir o tempo de acerto

A. Reduzir a penalidade – hierarquia de caches

Memória não pode ser simultaneamente grande e rápida;
$\Longrightarrow$ hierarquia com caches maiores mais longe do processador



A.a Reduzir a penalidade – hierarquia de caches

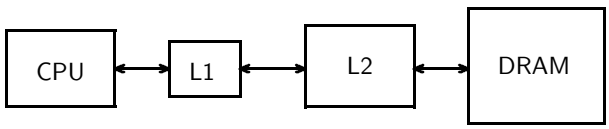


Taxa de faltas local = faltas na cache / acessos à cache
Taxa de faltas global = faltas na cache / referências pelo processador

$$\begin{aligned} T_{mem} &= tA_{L1} + F_{L1} \times P_{L1} \\ &= tA_{L1} + F_{L1} \times [tA_{L2} + F_{L2} \times P_{DRAM}] \end{aligned}$$

onde tA é o tempo de acerto,
F é taxa de faltas, e
P é a penalidade por falta

A.b Reduzir a penalidade – hierarquia de caches



Propriedade de Inclusão:
Conteúdo das caches maiores *inclui* o conteúdo das menores
$\Longrightarrow$ cache menor contém cópias dos dados na cache maior

- ⋆ qual a relação entre associatividade, tamanho e inclusão?
- ⋆ qual a relação entre bits de tag/índice e assoc, tamanho e inclusão?
- ⋆ [Baer,Wang88] *On the Inclusion Properties for Multi-Level Cache Hierarchies*

Caches *exclusivas* trocam linhas numa falta
$\Longrightarrow$ L1 $\longleftarrow$ novo; L2 $\longleftarrow$ velho

A.c Reduzir a penalidade – hierarquia de caches

Hierarquia no Itanium-2

- L1: 2 x 16KB, 4 conjuntos, linhas com 64 bytes, quatro portas (2-load + 2-store), latência de 1 ciclo
- L2: 256KB, 4 conjuntos, linhas com 128 bytes, quatro portas (2-load + 2-store), latência de 5 ciclos
- L3: 3MB, 12 conjuntos, linhas com 128 bytes, uma porta de 32 bytes, latência de 12 ciclos

B. Reduzir a penalidade – prioridade para faltas de leitura

- Escrita forçada com fila de escrita causa risco RAW
  - ⋆ processador tenta ler dado que está na fila de escrita
  - ⋆ solução ruim: espera até que fila esvazie e então prossegue com LOAD
    $\Longrightarrow$ aumenta penalidade nas faltas de leitura
  - ⋆ solução melhor: verifica se $\exists$ dependências;
    se não, LOAD prossegue sem bloquear
- Escrita preguiçosa: falta na leitura substitui bloco sujo
  - ⋆ solução ruim: escreve bloco sujo e então busca bloco faltante
  - ⋆ solução melhor:
    1. escreve bloco sujo num buffer/fila,
    2. lê bloco faltante e entrega ao processador,
    3. então escreve bloco sujo
  - ⋆ menos perda de tempo porque CPU prossegue assim que ler bloco faltante

C. Reduzir a penalidade – minimizar tempo de carga

- Tempo de preenchimento do bloco é longo
- ...mas não precisa esperar até que bloco seja preenchido
- **Early restart** – assim que palavra requisitada chegar da memória, entrega ao processador, que continua a executar
- **Critical word first** – busca palavra requisitada primeiro e a entrega ao processador assim que chegar da memória
  - ⋆ processador continua enquanto bloco está sendo preenchido
  - ⋆ técnica é útil com blocos grandes
  - ⋆ processador requisita **p2**:

p0	p1	**p2**	p3

  - ⋆ **p2** é entregue, e bloco é preenchido com p2 $\longrightarrow$ p3 $\longrightarrow$ p0 $\longrightarrow$ p1
  - ⋆ localidade espacial indica que próximo acesso será na palavra seguinte, que pode não ter chegado ainda

D. Reduzir a penalidade – cache não-bloqueante

- Cache não-bloqueante entrega dados enquanto trata de uma falta
  - ⋆ processador com execução fora-de-ordem
  - ⋆ memória com múltiplos bancos $\Longrightarrow$ cache & DRAM
  - ⋆ [Kroft81] *Lock-up Free Instruction Fetch/Prefetch Cache Organization*
- **acerto sob falta** (*hit under miss*) reduz penalidade efetiva fornecendo dados, ao invés de bloquear processador
- **acerto sob múltiplas faltas** (*miss under miss*) reduz penalidade ainda mais ao sobrepor tratamento de múltiplas faltas
  - ⋆ complexidade do controlador da cache é MUITO maior por causa dos acessos concomitantes à memória
  - ⋆ necessita de memória com múltiplos bancos
  - ⋆ PentiumPro permite até 4 faltas pendentes

Melhoria no desempenho de caches

Tempo médio de acesso a memória =
tempo de acerto +
taxa de faltas x penalidade pela falta

Para melhorar o desempenho é necessário:

- reduzir a taxa de faltas
- reduzir a penalidade nas faltas
- **reduzir o tempo de acerto**

I. Reduzir o tempo de acerto – caches simples e pequenas

Tempo de acerto curto se:

- cache pequena $\Longrightarrow$ reduz tempo de acesso
- cache simples $\Longrightarrow$ mapeamento direto

Alpha 21164 tem 8KB para dados e 8KB para instruções,
96KB cache secundária, unificada, associatividade ternária,
e $\Longrightarrow$ relógio muito rápido!

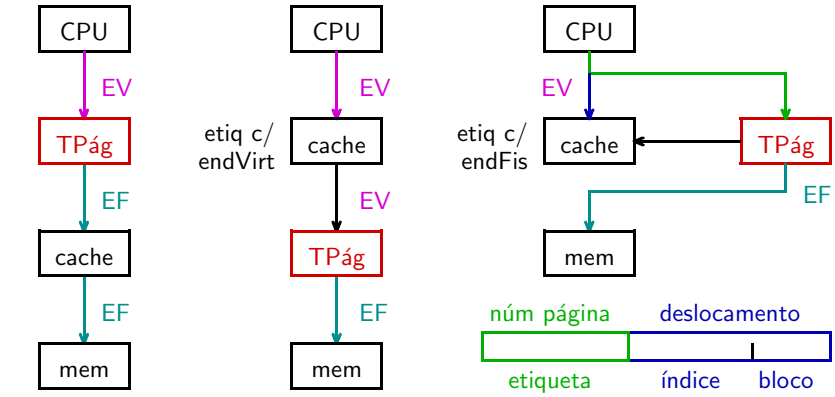
II. Reduzir o tempo de acerto – evitar tradução de endereços

Cache endereçada com: endereço virtual vs endereço físico

- a cada troca de contexto deve expurgar a cache
  - ⋆ senão ocorrem falsos-acertos
  - ⋆ processo que entra no processador tem montes de faltas compulsórias
  - ⋆ conseqüência da troca de contexto pode durar 10^5 ciclos
  - ⋆ solução: usar **identificador de processo** na etiqueta na cache
- como trata de sinônimos?
  - ⋆ dois endereços virtuais que mapeiam no mesmo endereço físico
  - ⋆ solução: garantir que indexação da cache ocorre com bits que coincidem no EV e EF

EVirt	núm página virtual	deslocamento
ender. cache	|—**índice**—|	
EFís	núm página física	deslocamento

II.i Reduzir T de acerto – evitar tradução de ender's



organização convencional

cache ender virtual
traduz somente nas faltas

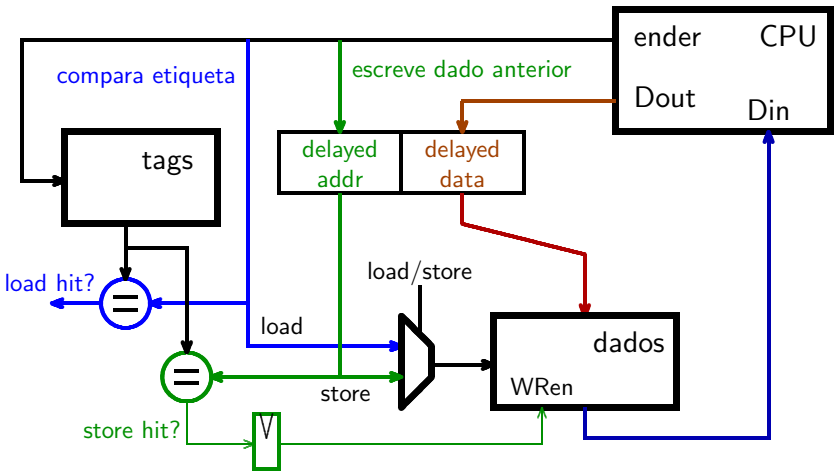
sobrepõe indexação com tradução EV-EF
índice não pode mudar durante tradução
|cache| = |página|

III.i Reduzir o tempo de acerto – escritas em pipeline

Escrita consome dois ciclos

- um para comparar etiqueta e outro para escrever na cache
- armazena dados e endereço em buffer
- escreve dados na cache durante comparação de etiqueta na próxima escrita
- risco RAW entre leitura e escrita atrasada $\Longrightarrow$ adiantamento

III.ii Reduzir tempo de acerto – escritas em pipeline



Resumo – melhorar desempenho de caches

T_méd_acesso_mem = T_acerto + tx_faltas x penalidade/falta

Para melhorar o desempenho é necessário:

- ⋆ **reduzir a taxa de faltas**
  - ▷ parâmetros de projeto: núm,tam blocos, associatividade
  - ▷ cache de vítimas, stream buffers, busca antecipada
  - ▷ otimização do código: agrupamento, troca de índices, fusão, blocagem
- ⋆ **reduzir a penalidade nas faltas**
  - ▷ hierarquia de caches
  - ▷ priorizar faltas na leitura (critical word first, early restart)
  - ▷ cache não-bloqueante
- ⋆ **reduzir o tempo de acerto**
  - ▷ caches simples e pequenas
  - ▷ não traduzir endereço virtual para físico
  - ▷ escritas em pipeline

Revisão – melhorar desempenho de caches

T_méd_acesso_mem = T_acerto + tx_faltas x penalidade/falta

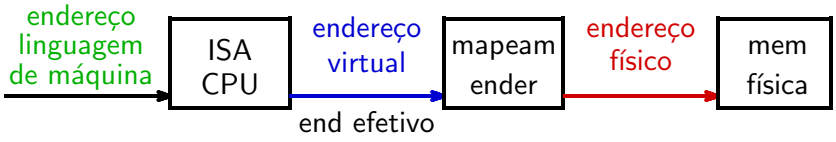
Para melhorar o desempenho é necessário:

- ⋆ **reduzir a taxa de faltas**
  - ▷ parâmetros de projeto: núm,tam blocos, associatividade
  - ▷ cache de vítimas, stream buffers, busca antecipada
  - ▷ otimização do código: agrupamento, troca de índices, fusão, blocagem
- ⋆ **reduzir a penalidade nas faltas**
  - ▷ hierarquia de caches
  - ▷ priorizar faltas na leitura (critical word first, early restart)
  - ▷ cache não-bloqueante
- ⋆ **reduzir o tempo de acerto**
  - ▷ caches simples e pequenas
  - ▷ não traduzir endereço virtual para físico
  - ▷ escritas em pipeline

Sistema de Memória

- Sistemas de memória
- Memória cache
  - ⋆ organização
  - ⋆ leitura
  - ⋆ escrita
  - ⋆ otimizações
  - ⋆ busca antecipada
- **Memória Virtual**
  - ▷ endereçamento e proteção
  - ▷ paginação
  - ▷ TLB
  - ▷ excessões

Tipos de *loci* em memória



- Endereço de linguagem de máquina
  como especificado pelo montador
- Endereço virtual ⟶ endereço efetivo
  conj de instruções especifica tradução dos endereços
  gerados pelo montador para endereços usados pelo ligador
- Endereço físico
  sistema operacional especifica mapeamento de endereço virtual
  para o número de uma posição na memória física

Espaço de Endereçamento – três modelos

- Primeiro Modelo (até ≈1960)
  - ∗ Programador supõe que espaço de endereçamento é contínuo
    espaço se extende de 0x0000 0000 a 0xffff ffff
  - ∗ programa deve ser carregado sempre no endereço físico inicial
  - ∗ memória pode conter somente um programa em execução
- Segundo Modelo (até 1972 - IBM 370, idéia de 1962)
  - ∗ EdE contínuo, > que memória física ⟶ overlays
  - ∗ Registrador de tradução para fazer relocação do programa
    ⟶ programa pode ser carregado em qualquer endereço físico
  - ∗ mais de um programa carregado em memória para execução
- Terceiro Modelo (após 1972 - IBM 370)
  - ∗ EdE > que memória física, tradução de ender transparente
  - ∗ muitos programas carregados na memória

Endereços Absolutos

No início da década de 50, computador da moda era o EDSAC, e endereço virtual = endereço na memória física

Só um programa executa na máquina,
com acesso irrestrito a todos seus recursos (RAM+E/S)

Endereços no programa dependem de
onde o programa está carregado na memória

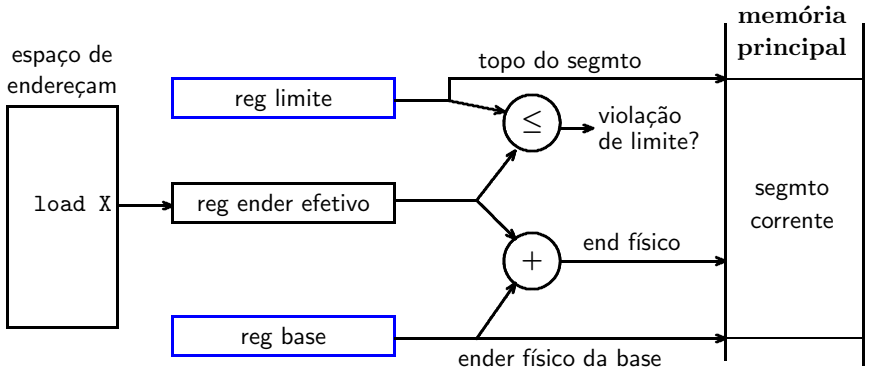
É fácil escrever sub-rotinas com código independente de posição?
como?

Tradução Dinâmica de Endereços

- Motivação:
  operações de E/S nas máquinas antigas eram lentas e processador era envolvido em todas as transferências (polling)

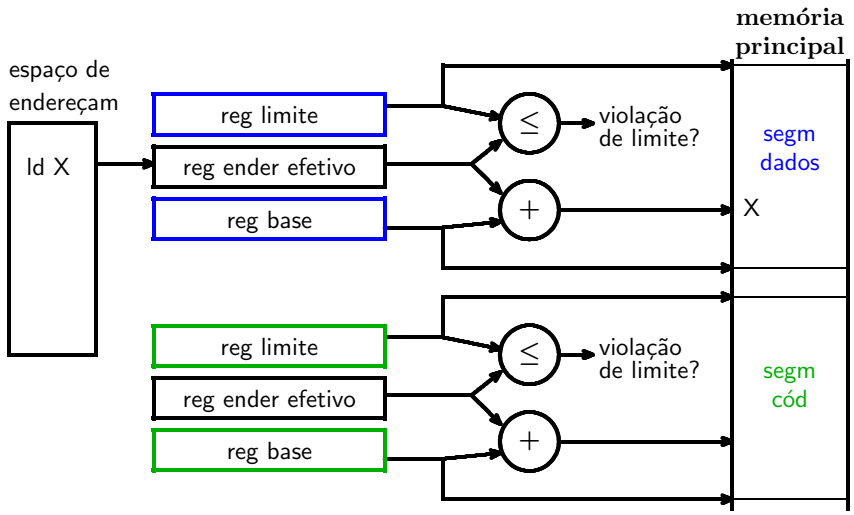
  Produtividade maior se dois ou mais programas executassem concorrentemente ⟶ **multiprogramação**
- Programas independentes de posição:
  facilitam a programação e gerenciamento da memória
  ⟶ é necessário um `registrador_base`
- Proteção:
  programas independentes não deveriam afetar-se acidentalmente
  ⟶ é necessário um `registrador_limite`

Tradução com registradores base e limite



Registradores base e limite são visíveis somente quando
processador executa em modo supervisor

Áreas Separadas para Programa e Dados



∃m vantagens nesta separação?

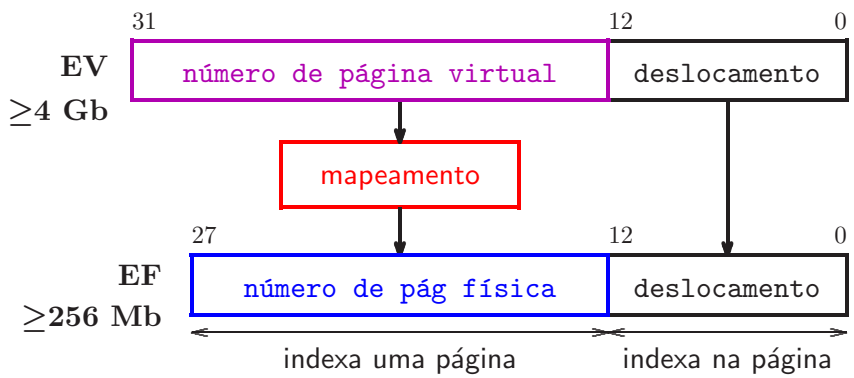
Fragmentação da Memória

- Programas com segmentos de tamanho fixo devem ser acomodados na memória que está disponível
- à medida que programas/usuários entram e saem do sistema, memória fica com buracos onde estavam os programas
  ⟶ fragmentação
- De quando em quando, programas devem ser re-locados/movidos para abrir espaço
  ⟶ compactação

SO da Microsoft™ obriga usuários a fazer compactação de disco, um problema que foi resolvido (não pela primeira vez) no Unics, em 1979. *That IS innovation*™.

Paginação

Espaço de endereçamento dividido em páginas de 4-8 Kbytes
Tabela de Páginas mapeia endereços virtuais em endereços físicos



Quais as vantagens das páginas de tamanho fixo
e da indireção através da tabela de páginas?

Paginação (cont)

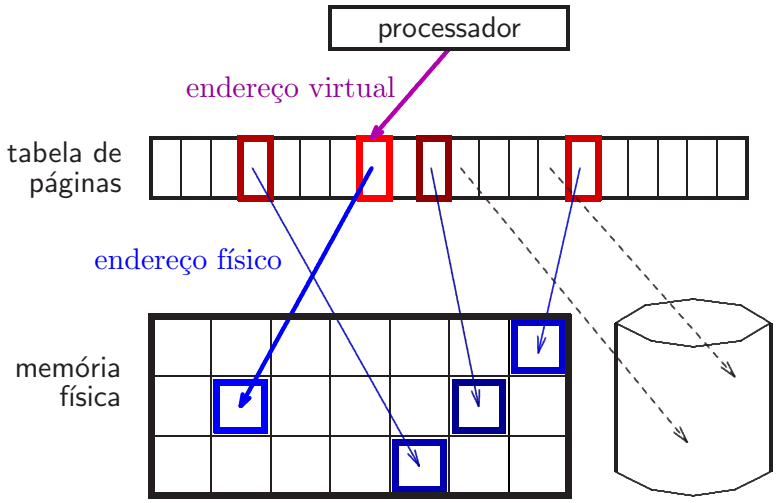


Tabela de Páginas

- Qual o tamanho da tabela de páginas?
- Quantas tabelas de páginas são necessárias?
- O espaço necessário para as tabelas de páginas é proporcional ao espaço de endereçamento, ao núm de usuários ...
  - ▷ espaço necessário é grande!
  - ▷ grande demais para manter em registradores (muitas páginas)
- Alocar TP em registradores especiais, só para o usuário corrente
  - ▷ pode não ser factível para TPs grandes
- Alocar TP na memória principal
  - ▷ necessita uma referência para buscar endereço base da página e...
  - ▷ outra para acessar o dado/instrução
    ⟶ dobra o número de referências à memória

Paginação sob demanda

- Paginação reduz fragmentação externa (buracos entre programas)
- Problema:
  acomodar programa com muitos dados em memória pequena?
  - ▷ usar overlays – programador deve controlar quais trechos de código/dados estão residentes em RAM eeeeca!
  - ▷ programação com overlays é complicada!
- **Paginação sob demanda:**
  "uma página da memória secundária é trazida para a memória primária sempre que for (implicitamente) requisitada pelo processador" Atlas 1962
  - ▷ memória primária atua como cache para memória secundária

DRAM como Cache de Disco

Se página virtual não está em memória física,
⟶ deve ser copiada do disco
⟶ alguma página deve ser ejetada para abrir espaço
localidade recomenda:
vítima é a que foi usada no passado mais distante (LRU)

Exemplo: 11, 10, 12, 9, 11, 7, 11, 13 LRU c.r.a 13? 11?

Escrita é preguiçosa:

- páginas semente de leitura (código) são substituídas
- páginas com atualizações são marcadas `sujas` e, antes de substituídas, disco deve ser atualizado

Cache vs Paginação sob Demanda

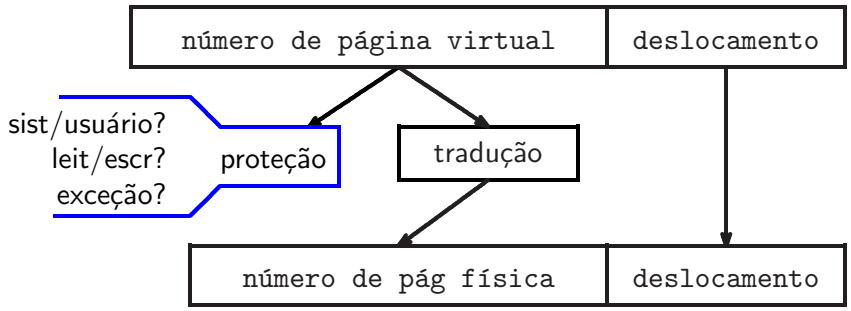
cache	paginação
linha na cache	quadro (moldura)
bloco na cache 32-64 bytes	página 4-8K bytes
faltas na cache 1-20%	faltas de página <0.001%
acerto na cache 1 ciclo	acerto na TP 100 ciclos
falta na cache 100 ciclos	falta de página 10^6 ciclos
falta tratada em HW	falta tratada espec em SW

Sistemas modernos de Memória Virtual

- **Proteção e privacidade**
  - ▷ vários usuários,
  - ▷ cada um com seu espaço de endereçamento privativo,
  - ▷ e um ou mais espaços de endereçamento compartilhados

  tabela de páginas ≈ espaço de endereçamento mapeamento
- **paginação sob demanda**
  - ▷ capacidade de executar programas maiores que a memória física
- **melhor utilização dos recursos**
  - ▷ uso compartilhado da memória
  - ▷ menor fragmentação externa
- **custo:** tradução de endereço a cada referência à memória

Tradução de Endereços e Proteção



- cada referência a dado ou instrução necessita de tradução e verificação de proteção

  proteção obtida através da indireção
- um bom sistema de memória virtual deve ser rápido ($\leq$ 1 ciclo) e usar espaço eficientemente

Tabela de Páginas

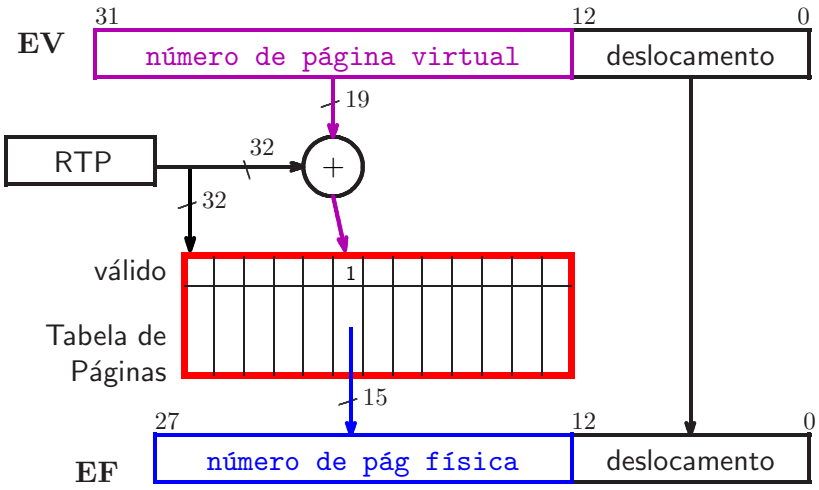


Tabela de Páginas Linear

Registrador de Tabela de Páginas (RTP) aponta para início da Tabela de Páginas do processo

A cada troca de contexto o SO atualiza o RTP para que este aponte a tabela de páginas do processo de usuário.

Cada elemento da tabela de páginas contém:

nPF número da página física residente em memória

nPD número da página em disco (se página foi movida para disco)

nil página não-existente

stat bits de status e proteção/uso used, dirty, {RO,RW,EX}

Tamanho da TP linear

Com endereços de 32 bits, páginas de 4 Kbytes, 4 bytes/elemento:
$\longrightarrow 2^{20}$ elementos = 4 Mbytes por processo
$\longrightarrow$ 4 Gbytes de swap para conter espaço de endereço completo

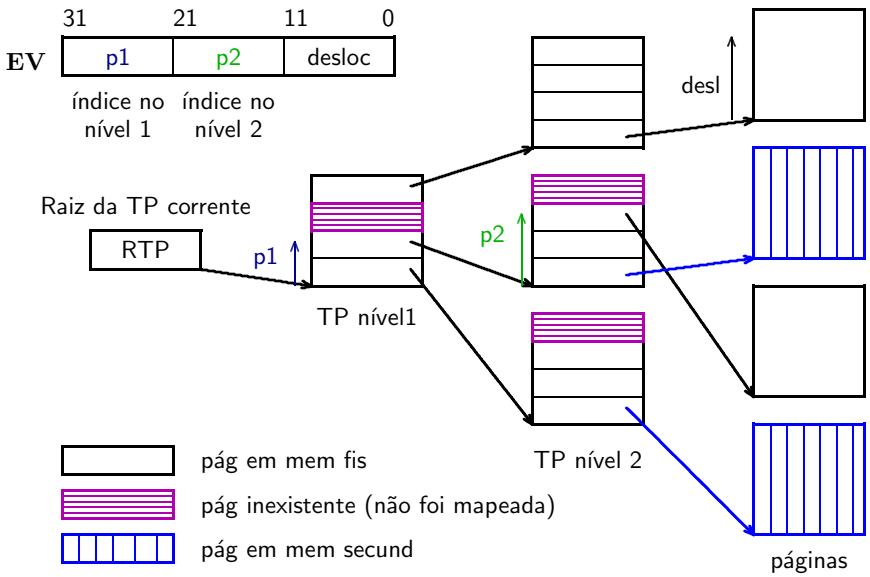
Páginas maiores!

mais fragmentação interna — última página meio vazia
penalidade por falta maior — mais tempo para ler do disco

Processadores de 64 bits
$\longrightarrow$ mesmo páginas de 1 Mbyte implicam em tabelas com
2^{44} elementos de 8 bytes (≈ 35 TBytes)

Há salvação?
Como?

Tabela de Páginas Hierárquica



Cache de Mapeamentos I

A cada referência, processador consulta TP
para descobrir endereço físico do objeto
$\longrightarrow$ faz uma referência à Tabela de Página para obter endereço
e então faz referência ao objeto...
$\longrightarrow$ para cada referência, DOIS acessos à memória...

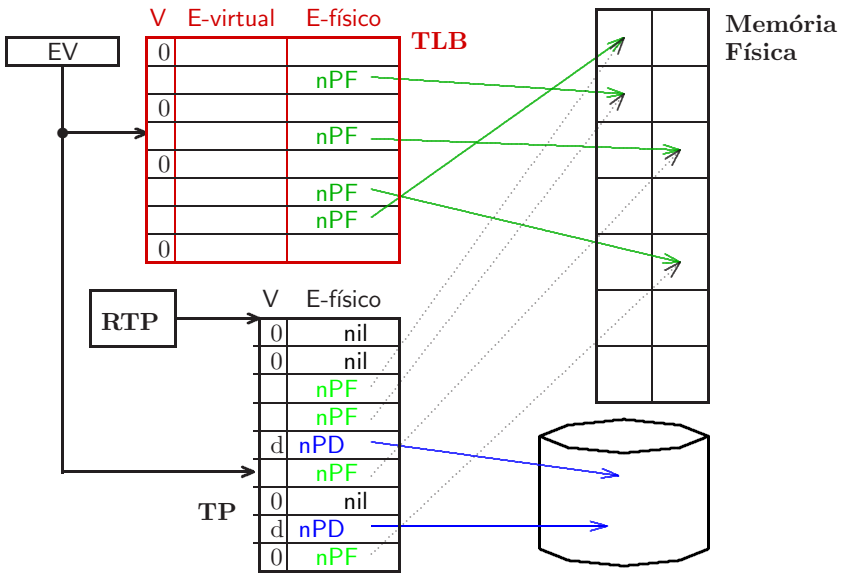
Solução:
manter mapeamentos numa Cache de Mapeamentos
ou Translation Lookaside Buffer (TLB)
ou Translation Buffer (TB, no Vax)

Translation Lookaside Buffer

Processador procura mapeamento na TLB;
se encontra, completa referência;
senão, busca mapeamento da Tabela de Páginas (em mem física)
e guarda na TLB para uso futuro

Se encontra na TLB → tradução em $\leq$ **1** ciclo — TLB hit
se falta na TLB → procura na TP (30-100 ciclos) — TLB miss

Cache de Mapeamentos II



Parâmetros de projeto de TLBs

- Tamanho da TLB: 32 a 128 blocos
- tamanho de bloco: 1..2 mapeamentos por bloco
- associatividade alta
  - ▷ cada elemento mapeia uma página
    menor localidade espacial entre as páginas
    → maior número de conflitos
  - ▷ TLBs grandes (256-512 elmtos) tem associatividade 4-8
  - ▷ TLBs menores com associatividade total
  - ▷ substituição de blocos com FIFO ou aleatória
    implementar LRU na TLB é caro — por que?

Pergunta:

Qual a cobertura da TLB?

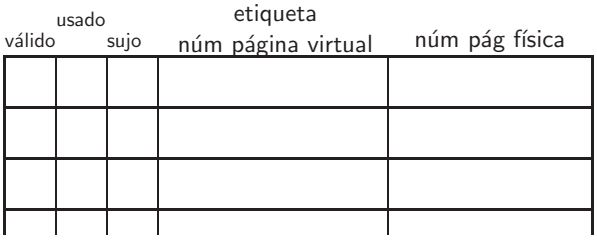
a) |TLB| × |pag|
b) |TLB| × |Tab_pag|
c) |pag| × |Tab_pag|
d) nenhuma das acima

Resposta:

Qual a cobertura da TLB? ou,
qual o maior espaço de endereçamento mapeado pela TLB?

a) |TLB| × |pag|
b) |TLB| × |Tab_pag|
c) |pag| × |Tab_pag|
d) nenhuma das acima

Campos da Cache de Mapeamentos (TLB)



`válido == 1` se mapeamento é válido
`usado == 1` se página foi referenciada recentemente (LRU)
`sujo == 1` se ocorreu uma ou mais escritas na página

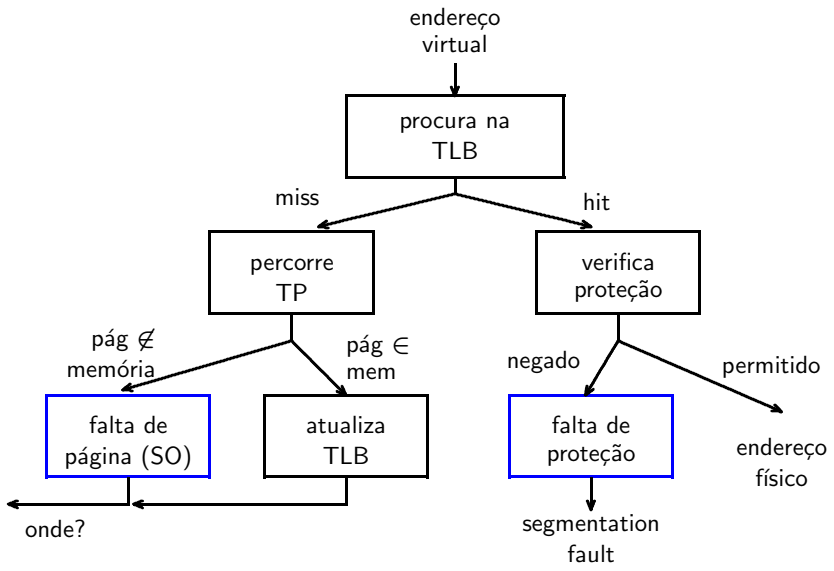
Se página virtual não está em memória,
NÃO pode haver um mapeamento da página na TLB

Tratamento de Faltas na TLB

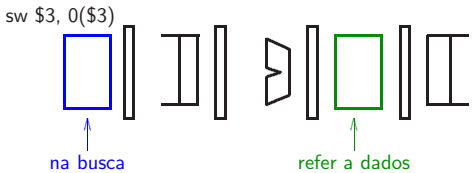
- **Em software** MIPS, Alpha
  - ▷ falta na TLB causa exceção, SO caminha pela TP hierárquica e re-carrega elemento da TLB
  - ▷ instrução que causou a falta na TLB é re-iniciada (busca)
  - ▷ caminhada ocorre em modo privilegiado (end fís), sem tradução
- **Em hardware** SparcV8, x86, PowerPC
  - ▷ unidade de gerenciamento de memória (MMU) caminha pela TP e re-carrega elemento da TLB
  - ▷ instrução que causou a falta na TLB é re-iniciada (busca)
  - ▷ se página (dados/código, ou TPs $2^{\underline{o}}$ nível) não está em memória, MMU sinaliza uma falta de página na instrução que causou falta na TLB

Por que a instrução causadora **deve** ser buscada novamente?

Tradução de Endereços



TLB & Cache – excessões



Faltas & Excessões

Causas: endereço ilegal — fora do espaço de endereçamento
ou desalinhado — end%4≠0

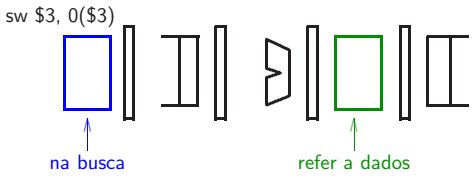
ou falta na TLB

ou falta na tabela de páginas — *page fault*

TLB & Cache

Falta na TLB pode ocorrer

- na busca de instrução
  (i) processador esvazia pipeline; (ii) re-carrega TLB; e
  (iii) busca instrução novamente custo: drenar pipeline + carga da TLB
- referência a dados (`ld` e `st`)
  (i) instrs à frente completam; instr não pode alterar estado: `ld r1,0(r1)`
  (ii) processador re-carrega TLB; e (iii) busca instrução novamente
  custo: drenar pipeline + carga da TLB + estágios até MEM



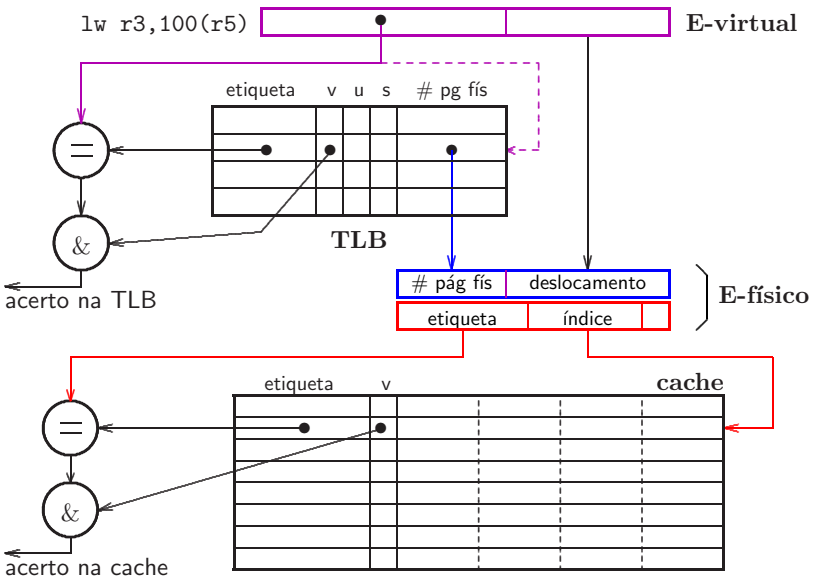
Reposição de Páginas

Se página virtual não está em memória
SO assume controle e requisita cópia ao controlador de disco
SO escolhe vítima para ser substituída pela nova página;
⟶ se vítima estiver suja, SO deve atualizar disco

Localidade implica em vítima ser
página usada no passado mais distante (LRU)

Como implementar LRU para 4 conjuntos com 3 bits?
Pode ser pseudo-LRU...

TLB & Cache



Resumo – memória física e virtual

- Aumento da vazão
  - ▷ intercalação de blocos
  - ▷ barramentos largos + intercalação + split transactions
- Redução da Latência
  - ▷ fast page mode
  - ▷ re-escrever Leis de Maxwell?
- Memória Virtual
  - ▷ Proteção *vs* utilização dos recursos
    proteção obtida através da indireção
  - ▷ |TP| grande ⟶ tabela de página hierárquica (multi-nível)
  - ▷ TLB para reduzir custo da tradução
  - ▷ relação entre Mem Virtual e tratamento de excessões
    falta de página é evento freqüente ⟶ tratamento rápido

Processamento Paralelo & Multiprocessadores

- **Motivação**
- **Tipos de máquinas paralelas**
- Coerência entre caches
- Sincronismo
- Ordenação de operações em memória

Motivação

Problema:

algumas computações são muito grandes para uniprocessadores

Por **grande** entende-se muito tempo ou muita memória

Solução:

a) projetar processadores mais rápidos

⟶ de 1985-2005 o crescimento de desempenho foi enorme

⟶ processadores modernos já são MUITO complexos

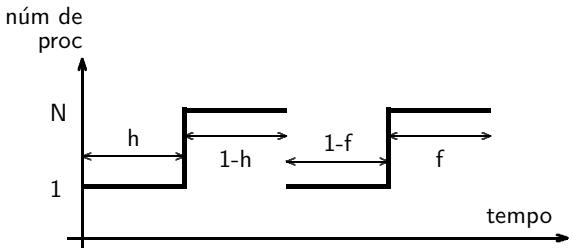
b) usar vários processadores trabalhando em paralelo

⇝ progresso no desenvolvimento de software tem sido lento

⟶ progresso bom sw de servidores e algumas aplics embutidas

⟶ base de sw para desktops é enorme...

Complicador: Lei de Amdahl



Ganho de 80 com 100 processadores só se $\mathbf{0,25\%}$ do trabalho é serial

fazer Exercício 6.1

Complicador: Custos da cooperação

- Cooperação entre processadores envolve:
  - ▷ programar aplicações para execução paralela — difícil
  - ▷ computação ⟶ processadores trabalham — bom
  - ▷ comunicação ⟶ processadores esperam — ruim
- Custo da comunicação depende de:
  - ▷ sw/hw para empacotar mensagens
  - ▷ rede que transporta mensagens
- Evento de comunicação custa vários micro-segundos
  - ▷ processador @ 1GHz ⟶ 1μs=1000 ciclos
  - ▷ deve-se explorar localidade, busca antecipada, e o que mais houver...

Aplicações

- Processamento paralelo
  - ⋆ paralelismo real num único programa
  - ⋆ compartilhamento de dados pode ser freqüente/intenso
- Processamento de Transações
  - ⋆ paralelismo entre transações independentes
  - ⋆ ênfase em produtividade *(throughput)*
- Sistemas Operacionais
  - ⋆ programa paralelo grande que executa por muito tempo
  - ⋆ SO paralelo é geralmente otimizado "a mão"
  - ⋆ compartilhamento de dados in-freqüente
    ⟶ estruturas de dados protegidas com granularidades variadas
- estas formas são distintas da exploração de paralelismo entre as instruções de uma única linha de execução
  single-thread vs *multiple-threads*

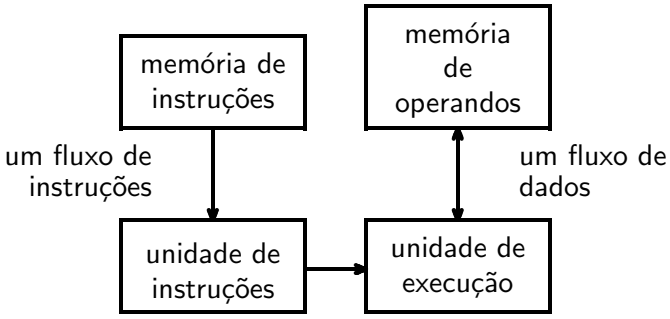
Taxonomia de Flynn

Taxonomia de Flynn, publicada 1966, e baseada no
número de fluxos de instruções e
número de fluxos de dados

- SISD – Single Instruction, Single Data
  - ▷ uniprocessador
- SIMD – Single Instruction, Multiple Data
  - ▷ processadores vetoriais/sistólicos
- MISD – Multiple Instruction, Single Data
  - ▷ poucos exemplos práticos (processadores sistólicos ?)
- MIMD – Multiple Instruction, Multiple Data
  - ▷ em uso comum; muito flexível

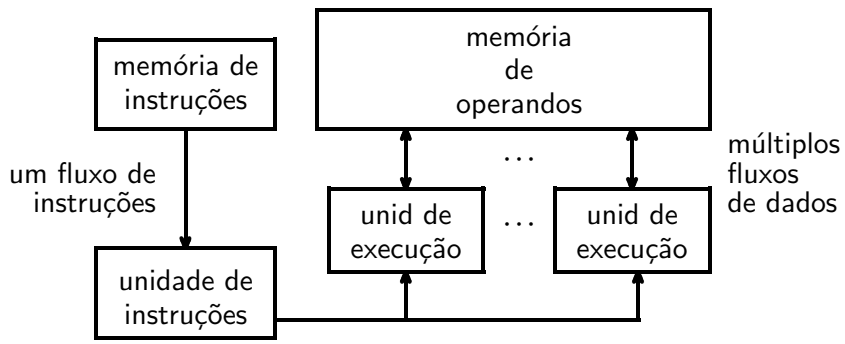
Taxonomia de Flynn – SISD

- SISD – Single Instruction stream, Single Data stream
  - ▷ armazenagem de instruções e operandos pode ser única
    Harvard vs Princeton
  - ▷ ...o bom e velho uniprocessador



Taxonomia de Flynn – SIMD

- SIMD – Single Instruction stream, Multiple Data streams
  - ▷ armazenagem de instruções e operandos usualmente separada
  - ▷ modelo de programação é *data parallel*
  - ▷ apropriado para problemas numéricos resolvidos com laços
  - ▷ paralelização automática pode ser possível —geralmente é



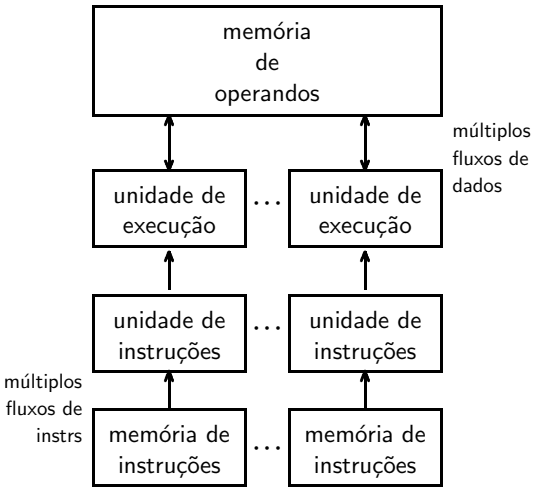
Taxonomia de Flynn – MIMD

- MIMD – Multiple Instruction streams, Multiple Data streams

mais flexível que SIMD,
"melhor" para uso geral

exemplo:
uma aplicação paralela
ou multi-programação

paralelização automática
é difícil (em geral)

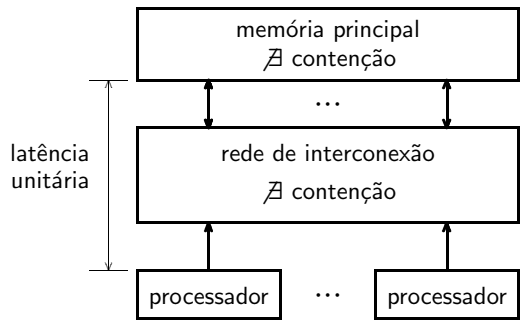


Eixos de Projeto

- Modelos de programação
  - ▷ sequencial (SISD)
  - ▷ data parallel (SIMD)
  - ▷ memória compartilhada (MIMD) *shared memory*
  - ▷ troca de mensagens (MIMD) *message passing*
- Interface binária da aplicação
  camada de sw que mapeia modelo de programação no hardware
  - ▷ memória compartilhada ⟶ comunicação com `Load's Store's`
  - ▷ troca de mensagens ⟶ comunicação com `send() recv()`
- Hardware
  - ▷ memória compartilhada ⟵ comunicação "grudada" na memória
  - ▷ troca de mensagens ⟵ comunicação "grudada" no sist de E/S

Modelo PRAM

- **Parallel RAM** = perfeição (≈ irreal)
  - ▷ memória completamente compartilhada
  - ▷ latência unitária
  - ▷ largura de banda infinita ⟶ não há contenção
  - ▷ localização dos dados irrelevante



Perfeição não é realizável

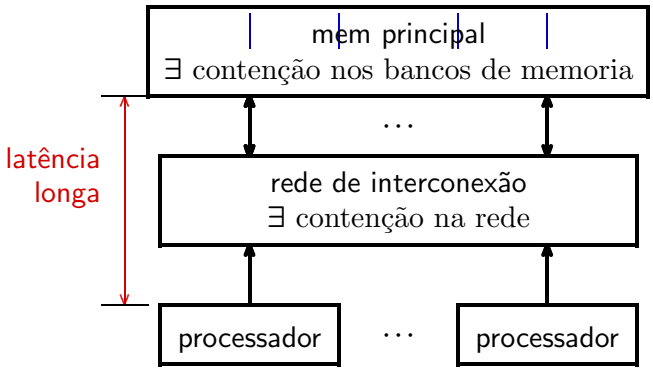
- Latência cresce se tamanho do sistema cresce
  ▷ veloc da luz é fixa ⟶ latência ∝ distância
- Vazão é limitada pela organização da memória
  e da rede de interconexão
- estas levam a uma divisão entre arquiteturas MIMD com
  Uniform Memory Access –**UMA**

  e

  Non-Uniform Memory Access –**NUMA**

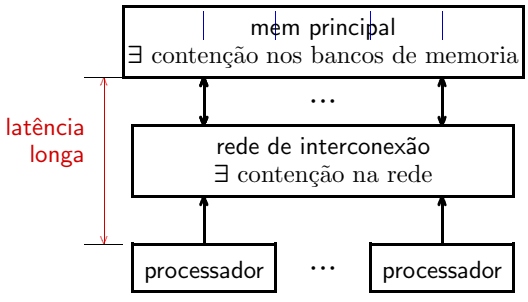
UMA – Uniform Memory Access

- latência no acesso à memória é a mesma para todos processadores
  mas pode ser longa
- latências crescem com tamanho do sistema
  aumentar escala é difícil



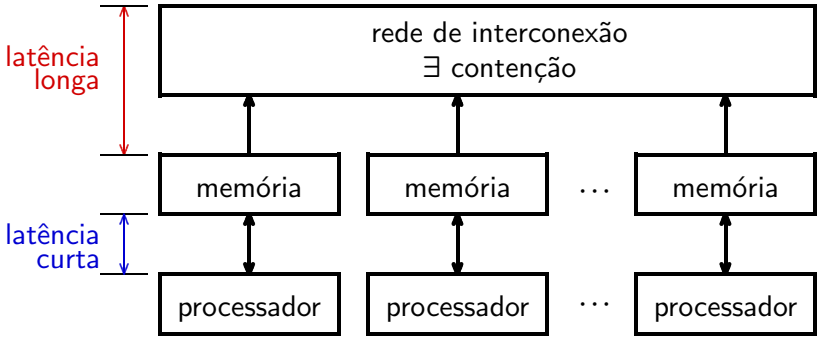
UMA – Uniform Memory Access

- Localização dos dados é irrelevante (não importa)
- contenção limita vazão ⟶ na rede e na memória
- geralmente usam caches
- usado em pequenos multiprocessadores = MPs simétricos
  Symmetric MultiProcessors ou SMPs



NUMA – Non-Uniform Memory Access

- Latência pequena no acesso à memória local
- latência grande no acesso à memória remota $P \rightarrow M_{loc} \rightarrow R \rightarrow M_{rem}$
- vazão para memória local pode ser mais alta que para remota
- contenção na rede e no acesso à memória



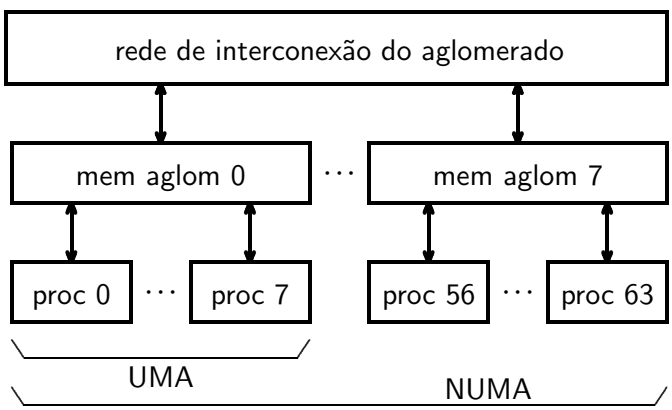
Multiprocessadores NUMA
Non-Uniform Memory Access

- Memória logicamente compartilhada mas fisicamente distribuída
  - ▷ um espaço de endereçamento lógico
  - ▷ pode ser tratado como memória compartilhada
- desempenho depende fortemente da localização dos dados

- Multicomputadores
  - ▷ cada processador tem espaço de endereçamento privativo
  - ▷ comunicação através de troca de mensagens (explícita)

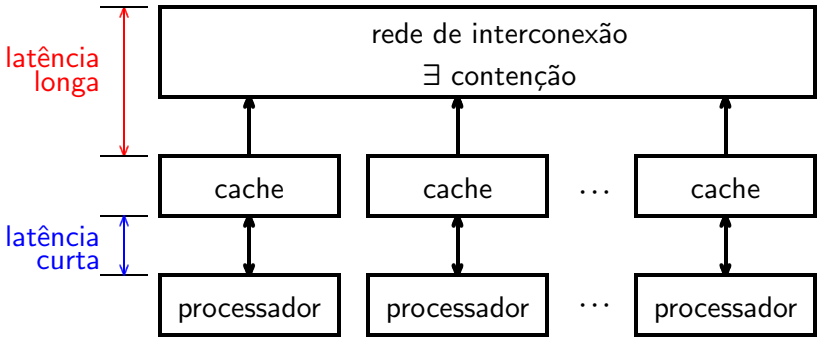
Aglomerados *(clusters)*

- Nós UMA pequenos num sistema NUMA grande
- híbrido? aglomerado de aglomerados?



COMA – Cache Only Memory Architecture

- Dados nas caches **migram** para onde são necessários
  localização dos dados importa muito pouco
- COMA é uma forma de NUMA
- área de pesquisa quente (90-94), só uma implementação (KSR-1)



Pergunta

Com 100 processadores interligados por uma rede perfeita e ideal,
qual fração do código pode ser executada serialmente,
se desejo ganho de {50,75,100}, com relação a 1 processador?

Resposta I

Com 100 processadores interligados por uma rede perfeita e ideal,
qual fração do código pode ser executada serialmente,
se desejo ganho de {50,75,100}, com relação a 1 processador?

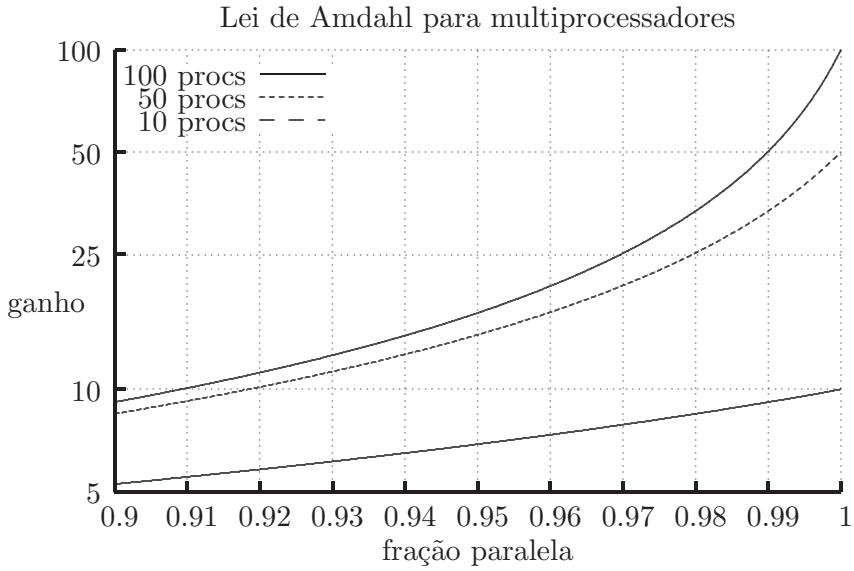
$$\text{Ganho}_{\text{total}} = \frac{1}{(1 - \text{Frac}_{\text{melhor}}) + (\text{Frac}_{\text{melhor}} \;/\; \text{Ganho}_{\text{melhor}})}$$

$50 = 1/(1 - 0.99) + (0.99/100)$ $\quad 1\%$

$75 = 1/(1 - 0.9965) + (0.9965/100)$ $\quad 0,35\%$

$100 = 1/(1 - 0.999999...) + (0.999999.../100)$ $\quad \approx 0\%$

Resposta II



O que é a fração serial?

- Inicialização
  - ▷ de estruturas de dados
  - ▷ criação de processos nos processadores
  - ▷ distribuição de trabalho entre os processadores
- Sincronização
  - ▷ contenção no acesso às regiões críticas — semáforos
  - ▷ mudanças de fases do programa — loop mais externo
- Término
  - ▷ coleta dos resultados parciais
  - ▷ E/S para disco

Comunicação Inter-processos

Na maioria das aplicações paralelas interessantes,
processos paralelos (tarefas) devem comunicar-se

Método de comunicação leva à divisão entre:
memória logicamente distribuída ⟶ troca de mensagens
memória logicamente compartilhada ⟶ memória compartilhada

Tipos de Máquinas Paralelas

Duas religiões em computação paralela:

- memória logicamente distribuída ⟶ **multicomputadores**
- memória logicamente compartilhada ⟶ **multiprocessadores**
- comunicação **ex**plícita vs comunicação **im**plícita

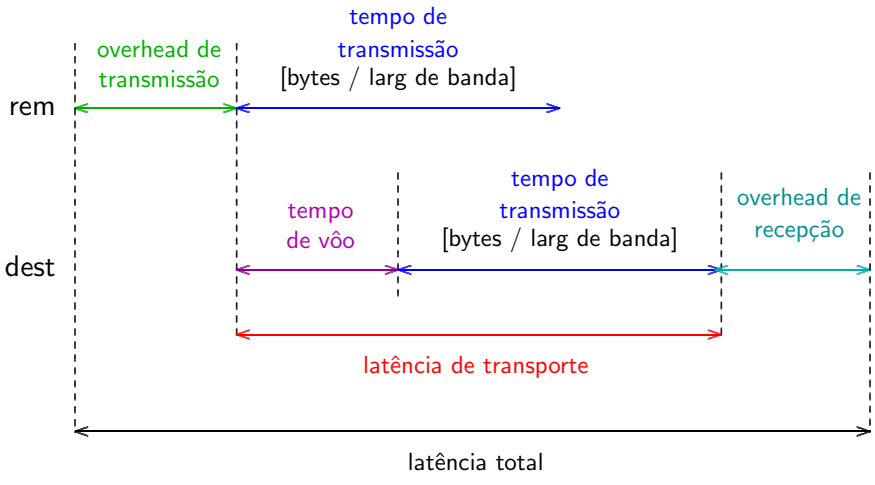
Nos dois casos, há uma **rede** no caminho crítico

na verdade, multiprocessadores são multicomputadores travestidos:
sempre há uma rede de interconexão;
⇝ eventos de comunicação baseados em mensagens...

Métricas de Desempenho na Comunicação
largura de banda/vazão

- A vazão pode ser limitada por
  - ▷ processador, memória, rede de interconexão
- a largura de banda pode ser medida em dois pontos:
  - ▷ na rede ⟶ **bisection bandwidth**:
    no pior diâmetro (c.r.a vazão global), corta os fios e mede vazão
  - ▷ em cada nó da rede ⟶ **na interface com a rede**
- mecanismo de comunicação afeta a largura de banda
  ⟶ **ocupação** de recursos utilização
  - ▷ tamanho das mensagens afeta vazão
    (acessos à memória + overhead)
  - ▷ taxa de comunicação
    (# interrupções ⟶ intervalo entre msgns)

Métricas de Desempenho na Comunicação
latência



Métricas de Desempenho na Comunicação
latência

- latência = Tx_ovhd
  + tempo_de_vôo + tempo_de_transmissão
  + Rx_ovhd
  - ▷ tempo_de_vôo $\propto$ velocidade da luz ($\approx$ 0,65c), distância
  - ▷ tempo_de_transmissão $\propto$ 1/relógio da rede
  - ▷ Tx_ovhd e Rx_ovhd dependem do mecanismo de comunicação e de sua implementação em hw/sw
- latência afeta desempenho e facilidade de programação
  - ▷ desempenho: recursos ocupados, ou processador esperando
  - ▷ programação: alocação de dados para aumentar localidade
- há forte relação entre *overhead* e ocupação...

Métricas de Desempenho na Comunicação
granularidade

- esconder latência pela superposição de comunicação e computação
  - ▷ superposição geralmente dificulta a programação
  - ▷ granularidade $\equiv$ computação / comunicação

As três métricas (vazão, latência, granularidade)
são influenciadas pela aplicação:
tamanho dos ítens (msgns) afeta vazão e latência
padrões de comunicação podem causar congestionamento

Mem compartilhada *vs* Mem distribuída

- **Memória compartilhada**
  - ▷ hw mais complicado (caches são coerentes)
  - ▷ conceitualmente similar a uniprocessadores e MPs centralizados
  - ▷ 'facilidade' de programar com modelo de memória compartilhada
  - ▷ fácil de programar quando padrões de comunicação são complexos
    ou mudam dinamicamente durante execução
  - ▷ baixa latência quando transfere ítens pequenos `loads` e `stores`
  - ▷ compartilhamento controlado por hw permite migração automática dos dados
  - ▷ MP com memória compartilhada emula troca de mensagens com facilidade

Mem compartilhada *vs* Mem distribuída

- **Memória distribuída**
  - ▷ hardware mais simples ($\not\exists$ coerência entre caches)
  - ▷ padrões de comunicação **devem** ser explicitados com precisão
  - ▷ programador deve estruturar programa para minimizar custos da comunicação
  - ▷ sincronização associada aos eventos de comunicação
  - ▷ emular memória compartilhada é difícil e/ou ineficiente
    SO envolvido em proteção,
    nomeação e
    envio/recepção de msgns

Multicomputadores

- **Arquitetura** – memória distribuída mais detalhes em breve
  - ▷ cada nó de processamento é computador completo (cpu+mem)
  - ▷ cada computador tem seu espaço de endereçamento privativo
  - ▷ cada nó tem identificador único, processos em cada nó tbém
  - ▷ comunicação entre nós somente através de mensagens
- Efeitos na latência
  - ▷ rede é pipeline para mensagens SE implementação permite
  - ▷ latência = tempo_por_nó $*$ núm_de_nós
    $+$
    comprimento_da_msgm / bytes_por_un_de_tempo
  - ▷ com msgns longas, primeiro termo torna-se insignificante
    $\longrightarrow$ comunicação infreqüente e com msgns longas
    $\longrightarrow$ granularidade relativamente grande (comput/comun $\gg 1$)

Multiprocessadores

- **Arquitetura** – memória logicamente compartilhada
  - ▷ cada nó possui parte da memória global da máquina
  - ▷ todos os nós enxergam toda a memória
  - ▷ memória local ao nó + memória remota ao nó NUMA
- Efeitos na latência
  - ▷ Referências a objetos compartilhados não distinguem localização, se local ou remota diferença no desempenho
  - ▷ mesmo modelo de programação de uniprocessador
- Exemplos:
  - ▷ Servidores da Sun, Silicon Graphics, IBM
  - ▷ motherboards com mais de uma CPU
  - ▷ Kendall Square Research KSR-1 (COMA) faliu em 95

Resumo

- Vazão através da rede (bisecção), e na interface processador-rede
- latência = Tx_ovhd
  + tempo_de_vôo + tempo_de_transmissão
  + Rx_ovhd
- granularidade ≡ computação / comunicação
- **multiprocessadores**: vários processadores interligados
  espaço de endereçamento compartilhado
  comunicação implícita, granularidade pequena/média
- **multicomputadores**: máquinas independentes cooperam
  espaços de endereçamento disjuntos
  comunicação explícita, granularidade grande

Processamento Paralelo & Multiprocessadores

- Motivação
- Tipos de máquinas paralelas
- **Coerência entre caches**
- Sincronismo
- Ordenação de operações em memória

Revisão – Eixos de Projeto

- Modelos de programação
  - ▷ sequencial (SISD)
  - ▷ data parallel (SIMD)
  - ▷ memória compartilhada (MIMD) *shared memory*
  - ▷ troca de mensagens (MIMD) *message passing*
- Interface binária da aplicação
  camada de sw que mapeia modelo de Programação no hardware
  - ▷ memória compartilhada ⟶ comunicação com `load, store`
  - ▷ troca de mensagens ⟶ comunicação com `send(), recv()`
- Hardware
  - ▷ memória compartilhada ⟵ comunicação via sist de memória
  - ▷ troca de mensagens ⟵ comunicação via sist de E/S

Revisão

- Vazão da rede (bisecção), e na interface processador-rede
- latência = Tx_ovhd
  + tempo_de_vôo + tempo_de_transmissão
  + Rx_ovhd
- granularidade ≡ computação / comunicação
- multiprocessadores: vários processadores interligados
  espaço de endereçamento compartilhado
  comunicação implícita, granularidade pequena/média
- multicomputadores: máquinas independentes cooperam
  espaços de endereçamento disjuntos
  comunicação explícita, granularidade grande

Execução atômica

```
/* P1 */          /* P2 */
c = a = 0;        d = b = 0;
b = b + 1;        a = a + 1;
c = a + b;        d = c + b;
print c;          print d;
```

/ Quais os valores de "print c" e "print d"? */*

Comandos 'atômicos' em C
não são
executados atomicamente
pelo processador

```
a = a + 1; ≡  lw r1,0(r2)
              addi r1,r1,1
              sw r1,0(r2)
```


Sincronização e Atomicidade

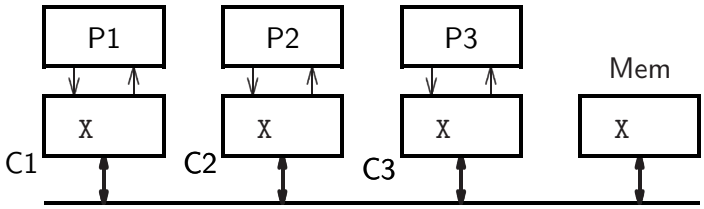
- Operações atômicas devem ser serializadas pelos mecanismos de escrita em memória
- Problemas decorrentes de disputas:
  - ▷ latência sem contenção
  - ▷ serialização se há contenção
    ⟶ dificulta escalar para sistemas maiores

Barramento é um meio de comunicação compartilhado
comunicação por **difusão** *broadcast*

⇝ comunicação é serializada através do barramento

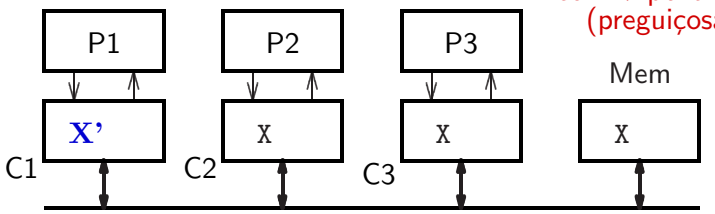
Coerência de caches

Inicialmente, três cópias idênticas de X em C1, C2 e C3



Processador P1 atualiza cópia para X'

X em C2 e C3 caducam com ∀ política de escrita (preguiçosa/forçada)



Coerência de caches

P1, P2 e P3 carregam X em suas caches;
P1 atualiza sua versão;
C2 e C3 ficam com sua cópia desatualizada.
⟶ isso ocorre **mesmo** que as caches usem escrita forçada (ou escrita preguiçosa)

Informalmente, **coerência entre caches** é um método para garantir que os acessos à memória sejam coerentes, apesar das caches.

Causas de In-coerência

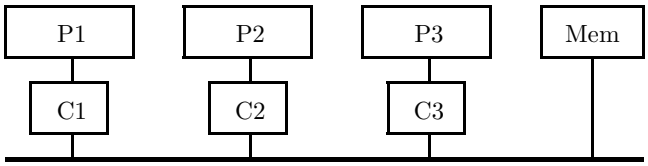
- Compartilhamento de dados para escrita
  é a causa considerada mais comumente
- migração de processos
  processo que sai deixa dados para trás
- entrada/saída
  geralmente corrigida com invalidação de blocos nas caches

Não é necessária uma noção de tempo_absoluto:
o que é necessário é a coerência 'aparente',
ao invés de coerência absoluta
⟶ a incoerência temporária entre memória
e cache com escrita preguiçosa pode ser aceitável

Soluções para Problemas de Coerência

- Evitar caches privativas — mover caches para sistema de memória
- Dados compartilhados não podem ficar na cache
  - ▷ possivelmente, a solução mais simples por software
- Invalidar blocos em momentos estratégicos
  - ▷ por exemplo, após seções críticas
  - ▷ fazer invalidações seletivas pode ser difícil... — em hw e/ou sw
- Usar uma tabela global
  - ▷ tabela mantém localização dos dados compartilhados
  - ▷ já foi tentado, mas não cresce em escala
- Usar caches que espionam o barramento — em uso comum
- Usar diretórios distribuídos junto com a memória
  - ▷ necessário em sistemas grandes (>64 processadores)

Métodos de Coerência para Barramentos



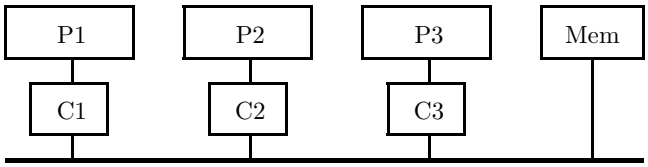
- As caches usam escrita preguiçosa — (tipicamente)
  - ▷ porque estas minimizam tráfego no barramento
- Os blocos nas caches podem estar num destes estados — (típicos)
  - ▷ INVÁLIDO: conteúdo do bloco não pode ser usado
  - ▷ VÁLIDO: não está sujo, compartilhado (há ≥ 1 cópia)
  - ▷ SUJO: única cópia suja
  - ▷ RESERVADO: não-sujo, única cópia (aumenta eficiência)

Métodos de Coerência para Barramentos

- **Espionagem** *(snooping)*
  - ▷ todas as caches observam **todo** o tráfego no barramento
  - ▷ etiquetas com duas portas: CPU e barramento
  - ▷ ações de um controlador de cache são visíveis pelos demais
    difusão no barramento
- O que ocorre nas escritas?

invalidar cópias nas outras caches: **protocolos de invalidação**

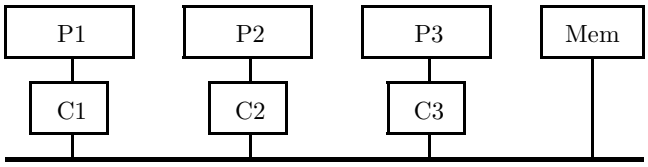
atualizar cópias nas outras caches: **protocolos de atualização**



Protocolo de Invalidação

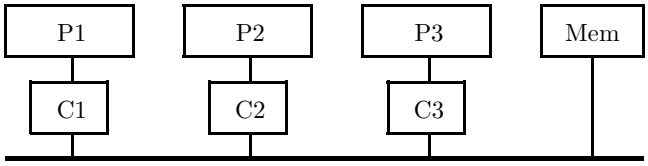
atividade no processador	atividade no barramento	cache C1	cache C2	memória ender X
				0
P1 lê X	falta em X	0		0
P2 lê X	falta em X	0	0	0
P1 faz X=1	invalidação para X	1	-	0
P2 lê X	falta em X *	1	1	1

* P1 responde à falta em C2 e atualiza memória



Protocolo de Atualização

atividade no processador	atividade no barramento	cache C1	cache C2	memória ender X
				0
P1 lê X	falta em X	0		0
P2 lê X	falta em X	0	0	0
P1 faz X=1	broadcast de X=1	1	1	1
P2 lê X	acerto em X	1	1	1



Há mais tráfego no barramento, se P2 não usar X novamente.

Protocolo de Invalidação Simplificado

- Caches usam escrita preguiçosa
- Estados
  - ▷ INVÁLIDO: conteúdo do bloco não pode ser usado
  - ▷ COMPART-ilhado: não está sujo, compartilhado (há ≥ 1 cópia)
  - ▷ EXCLUSIVO: única cópia suja
- a cada transação no barramento, o controlador de cache:
  - ▷ verifica se bloco está na cache
  - ▷ se estiver na cache, efetua mudança de estado cfe ME (adiante)

Protocolo de Invalidação Simplificado (cont.)

H-P QA
fig 6.11

Protocolo de Invalidação Simplificado (cont.)

H-P QA
fig 6.11

Desempenho de Protocolos para Barramentos

- Tipos de faltas nas caches:
  - ▷ capacidade (mais significativa)
  - ▷ conflitos
  - ▷ compulsórias
  - ▷ **coerência** ($4^{\underline{o}}$ C)
- Faltas por coerência
  são faltas adicionais causadas pelo protocolo de coerência
- Falso compartilhamento
  - ▷ a unidade de coerência é um **bloco** de cache
  - ▷ pode ocorrer que um bloco inteiro seja compartilhado, mas palavras individuais não são compartilhadas

Escalabilidade

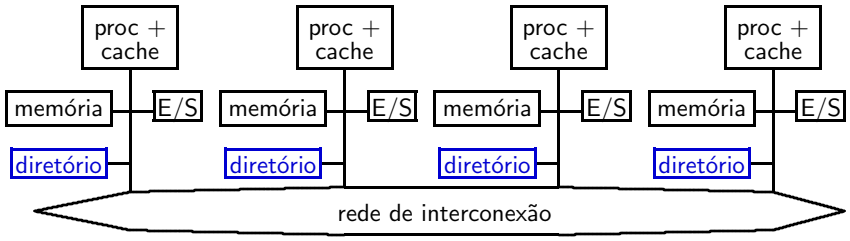
Multiprocessadores construídos com barramentos escalam bem até 10-20 processadores

barramento é o gargalo !!!

por que não usar mais barramentos? ⟶ dividir tráfego entre barramento de endereços & coerência, e barramento de dados

mesmo assim, limite é lá pelos 20 processadores...

Coerência de Caches com Diretórios

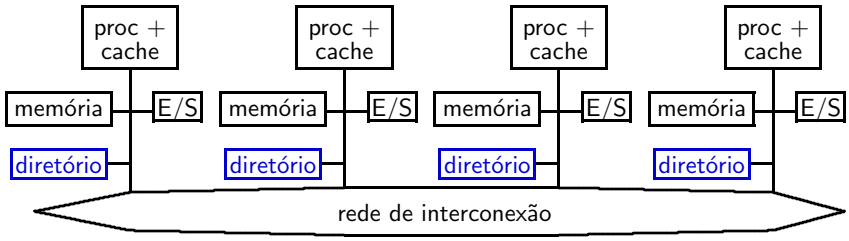


Para construir MPs grandes e escaláveis:

Informação sobre compartilhamento é mantida em diretórios distribuídos fisicamente pela memória

A informação de estado é mantida nas **caches** e nos **diretórios em memória**

Coerência de Caches com Diretórios



Não há um barramento para serializar acessos de escrita

Transações do protocolo de coerência usam mensagens e recibos (ACKs)

Coerência de Caches com Diretórios

O espaço de endereçamento é alocado estaticamente entre os processadores

Endereço especifica número do nó e endereço no nó | nó | endereço |

nó de residência nó em cuja memória o endereço foi alocado, nó contém diretório que controla estado do bloco

nó local nó que mantém cópia de bloco em sua cache
nó é local c.r.a processador que está referenciando o bloco

nó remoto nó que também mantém cópia do bloco cuja cópia deve ser invalidada

Coerência de Caches com Diretórios

Informação sobre compartilhamento mantida em diretório na memória:

- estados de blocos em memória:
  - ▷ **shared**: ≥ 1 processador/es tem cópia/s; memória atualizada
  - ▷ **uncached**: não há cópias em caches
  - ▷ **exclusive**: 1 processador tem cópia; memória está desatualizada
- elementos do diretório apontam para procs. com cópias
  - ▷ p.ex. com vetor de bits — um bit por processador
- tipos de apontadores:
  - ▷ mapa completo – N processadores, N apontadores (bits)
  - ▷ limitado – número fixo de apontadores (4-8)
  - ▷ lista – lista encadeada de blocos de caches

Coerência de Caches com Diretórios

- bloco no estado **uncached** ⟶ memória atualizada
  - ▷ falta na leitura: envia dado a quem pede;
    marca bloco como **shared**; apontador=dono
  - ▷ falta na escrita: envia dado a quem pede;
    marca bloco como **exclusive**; apontador=dono
- bloco no estado **shared** ⟶ memória atualizada
  - ▷ falta na leitura: envia dado da memória a quem pede;
    apontador=dono,novo
  - ▷ falta na escrita: envia dado da memória a quem pede;
    invalida todas as outras cópias;
    marca bloco como **exclusive**; apontador=dono

Coerência de Caches com Diretórios

- bloco no estado **exclusive** ⟶ memória des-atualizada
  - ▷ falta na leitura: pede dado a quem o possui;
    atualiza memória;
    marca bloco como **shared**; apontador=dono,novo
  - ▷ Write-Back: marca bloco como **uncached**;
    apontador=NIL
  - ▷ falta na escrita: pede dado a quem o possui;
    marca bloco como **exclusive**; apontador=dono

Coerência de Caches com Diretórios – ME cache

H-P QA
fig 6.29

Coerência de Caches com Diretórios – ME diretório

H-P QA
fig 6.30

Coerência de Caches com Diretórios – exemplo

Considere três processadores, P1, P2, P3, e a seqüência de eventos

1. bloco X não está em nenhuma cache
2. os três processadores efetuam leitura de X
3. P3 atualiza X (fica bloqueado)
4. acerto em C3, mas P3 não é dono do bloco
5. C3 requisita exclusividade (para poder escrever)
6. memória envia invalidações para C1 e C2
7. memória recebe respostas (acks), marca linha como exclusiva
   informa a C3 que agora é a dona do bloco
8. C3 atualiza bloco, marca-o como sujo
   P3 prossegue

Resumo

- Coerência de caches: aparência de tempo absoluto
  mantém ordem aparente das operações na memória
- Protocolos com espionagem em meio compartilhado (difusão)
  - ▷ invalidação – invalida cópias
  - ▷ atualização – atualiza cópias nas outras caches
- Protocolos com comunicação ponto-a-ponto
  - ▷ invalidação
  - ▷ atualização (é mais barato atualizar com msgns)
  - ▷ diretório distribuído mantém localização das cópias
    - ∗ mapa completo
    - ∗ mapa parcial
    - ∗ lista encadeada

Processamento Paralelo & Multiprocessadores

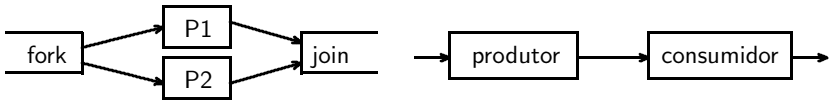
- Motivação
- Tipos de máquinas paralelas
- Coerência entre caches
- **Sincronismo**
- **Ordenação de operações em memória**

Revisão

- Coerência de caches: aparência de tempo absoluto
  mantém ordem aparente das operações na memória
- Protocolos com espionagem em meio compartilhado (difusão)
  - ▷ invalidação – invalida cópias
  - ▷ atualização – atualiza cópias nas outras caches
- Protocolos com comunicação ponto-a-ponto (redes)
  - ▷ invalidação
  - ▷ atualização (é mais barato atualizar com msgns)
  - ▷ diretório distribuído mantém localização das cópias
    - ∗ mapa completo
    - ∗ mapa parcial
    - ∗ lista encadeada

Sincronismo

- Sincronismo é necessário sempre que há processos concorrentes
  num sistema mesmo em uni-processador
- **Fork**s e **Join**s: os processos de um programa paralelo
  podem ter que esperar até que certos eventos tenham ocorrido
- **Produtor-Consumidor:** o processo consumidor deve esperar
  até que o processo produtor tenha produzido dados
- **Uso exclusivo de recurso:** o sistema operacional deve garantir
  que somente um processo de cada vez use um recurso



Execução atômica

```
/* P1 */          /* P2 */
c = a = 0;        d = b = 0;
b = b + 1;        a = a + 1;
c = a + b;        d = c + b;
print c;          print d;
```

/∗ Quais os valores de "print c" e "print d"? ∗/

Comandos 'atômicos' em C
não são
executados atomicamente
pelo processador

```
a = a + 1;  ≡  lw r1,0(r2)
               addi r1,r1,1
               sw r1,0(r2)
```

Comunicação e Sincronização

Sincronização **pode** ser programada com `loads` e `stores` normais:

```
/* proc 1 */                    /* proc 2 */
       flag1=0                          flag2=0
       ...                              ...
       flag1=1                          flag2=1
LAB1:  if(flag2 == 1)           LAB2:  if(flag1 == 1)
       goto LAB1                        goto LAB2
       /* seção crítica */              /* seção crítica */
       flag1=0                          flag2=0
```

MAS é difícil de programar e depurar ⟶ este exemplo pode bloquear

Semáforos (locks)

Um **semáforo** é um T.A.D. composto de um inteiro não-negativo, e das operações: *E W Dijkstra, 1965*

P(s): if **s** $>$ *0 decrementa* **s** *de 1; senão espera*

V(s): incrementa **s** *de 1 e acorda um dos processos a esperar*

P e V **devem** ser executados atomicamente, sem interrupções, ou acessos a **s** por outros processos

```
Processo( )
  P(s)
    região crítica
  V(s)
```

Valor inicial de **s** determina número máximo de processos na região crítica.

Exemplos de Primitivas de Sincronização (hw)

Test_and_set e Reset

```
TEST_AND_SET(lock)
    { tmp←mem[lock]; mem[lock]←1; return tmp; }

RESET(lock) { mem[lock]←0; }
```

Programa fica em loop até que `lock=0` é retornado *spin-lock*

```
while ( TEST_AND_SET(lock) == 1) { };
   /* região crítica */
RESET(lock);
```

Exemplos de Primitivas de Sincronização (hw)

Fetch_and_Add

```
FETCH_and_ADD(x,a)
    { tmp←mem[x]; mem[x]←tmp+a; return tmp; }
```

Fetch_and_Φ

```
FETCH_and_Φ(x,a)
    { tmp←mem[x]; mem[x]←Φ(tmp,a); return tmp; }
```

Swap

```
SWAP(x,a) { tmp←mem[x]; mem[x]←a; a←tmp; }
```


Sincronização em modo usuário

Sincronização com **spin locks**:
⟶ baixa latência para obter trava
⟶ região crítica pode ficar travada por pouco tempo

```
        li   r2, 1
lockit: swap r2, 0(r1)     # troca atômica
        bnez r2, lockit    # se falhou, repete
        # região crítica
        ...
unlock: st   r0, 0(r1)     # libera
```

Em sistema com caches:
variável com a trava mantida na cache
`swap` ≡ escrita ⟶ **montes de tráfego de coerência**

Sincronização em modo usuário

Em sistema com caches:
variável com a trava mantida na cache
`swap` ≡ escrita ⟶ montes de tráfego de coerência

Alternativa: espera em loop até liberar, então tenta obter trava

```
tryit:  ld   r4, 0(r1)    # em cache como SHARED
                          # qdo outro proc escreve:
                          # SHARED ⟶ INVAL
        bnez r4, tryit    # espera até ficar livre
        li   r2, 1        # livre! talvez ainda está?
lockit: swap r2, 0(r1)    # troca atômica
        bnez r2, tryit    # se falhou, repete
```

Exemplos de Primitivas de Sincronização – LL+SC

Load_linked + Store_conditional não-bloqueante

- par de instruções usa registrador especial para manter `flag`, endereço e resultado do `store_conditional`
- `load_linked`
  lê valor e escreve endereço em reg de endereçamento global
- `store_conditional`
  escreve valor **se** reg de endereço global não foi alterado;
  retorna indicação de sucesso (1) ou falha (0)
- opera sob a premissa de que máquina tem barramento

Exemplos de Primitivas de Sincronização – LL+SC

★ `load_linked` lê valor e escreve endereço em reg de endereçamento global

★ `store_conditional` escreve valor **se** reg de endereço global não foi alterado; retorna indicação de sucesso (1) ou falha (0)

```
Load_linked(R,m) {              /* R=reg, m=variável */
   <flag,end> ← <1,m>;
   R ← mem[m]; }

Store_conditional(m,R) {
   if ( <flag,end> == <1,m> )
         cancela reserva de m por outros procs;
         mem[m] ← R; R ← 1;
   else R ← 0; }
```


Exemplos de Primitivas de Sincronização – LL+SC

★ `load_linked` lê valor e escreve endereço em reg de endereçamento global

★ `store_conditional` escreve valor **se** reg de endereço global não foi alterado; retorna indicação de sucesso (1) ou falha (0)

Exemplo: troca atômica de `r4` com `mem[r1]` *atomic swap*

```
try: mov  r3, r4       # carrega valor para troca
     ll   r2, 0(r1)    # load linked: r2←mem[r1]
     sc   r3, 0(r1)    # store conditional:
                       # OK: mem[r1]←r3; r3←1
                       # NO: r3←0
     beqz r3, try      # se SC falhou, repete
     mov  r4, r2       # escreve valor trocado
```

Desempenho de locks

Instruções com read-modify-write atômico bloqueantes

test_and_set, fetch_and_Φ, swap

vs

Instruções com read-modify-write atômico não-bloqueantes

load_linked+store_conditional

vs

Protocolos baseados em load's e store's normais

Desempenho depende de vários fatores inter-relacionados:

latência sem contenção

grau de contenção

serialização se há contenção *impede aumentar escala*

caches

execução de load's e store's fora de ordem

Atomicidade, Consistência e Ordenação de Eventos

- Atomicidade
  - ⋆ informalmente: acessos a uma variável acontecem um de cada vez, e afetam todas as cópias imediatamente
- Consistência Seqüencial
  - ⋆ informalmente: extensão intuitiva do modelo de memória dos uniprocessadores
- Consistência Relaxada
  - ⋆ informalmente: somente operações de sincronismo são consistentes
    $\longrightarrow$ melhor desempenho
- Ordenação de Eventos = característica da implementação
  - ▷ ordenação forte – consistência seqüencial
  - ▷ ordenação fraca – consistência relaxada

coerência de caches $\longrightarrow$ mesmo endereço

consistência de caches $\longrightarrow$ endereços distintos

Ordenação de Eventos na Memória

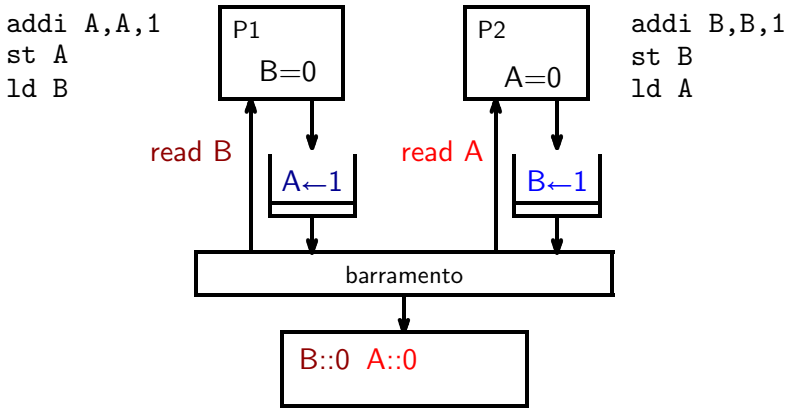
- Atualizações podem ser re-ordenadas pelo sistema de memória

Exemplo:

```
P1:  A=0;                    P2:  B=0;
     ...                          ...
     A=1;                         B=1;
L1:  if (B==0) ...           L2:  if (A==0) ...
```

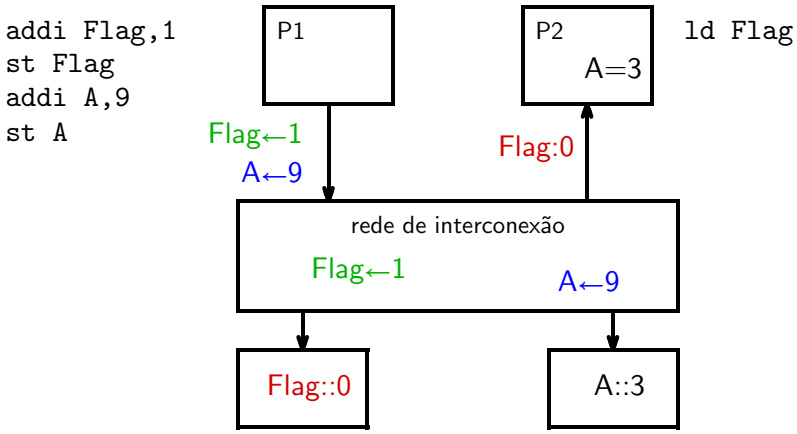
- intuitivamente, é impossível que ambos A e B sejam zero
  MAS isso pode ocorrer se as atualizações
  forem re-ordenadas pelo sistema de memória
- num Multi-Proc, as regras que ordenam a atualização em memória
  devem ser cuidadosamente definidas e atendidas pela implementação

Causas de Inconsistência – Filas de Escrita



leituras podem ultrapassar valores que estão esperando vez na fila
porque endereços são distintos

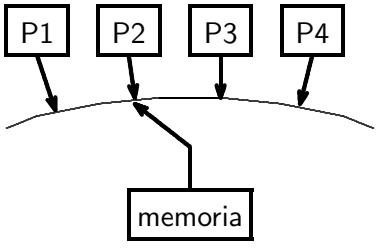
Causas de Inconsistência – Atrasos na Rede



tempo de viagem é $\propto$ à distância entre fonte e destino

Consistência Seqüencial

Um sistema é seqüencialmente consistente se o resultado de uma execução é o mesmo caso as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador individual aparecem na ordem especificada pelo seu programa. *Leslie Lamport, 1979*



Consistência Seqüencial

Um sistema é seqüencialmente consistente se o resultado de uma execução é o mesmo caso as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador individual aparecem na ordem especificada pelo seu programa. *Leslie Lamport, 1979*

P1: α, β, γ
P2: $\mathcal{P}, \mathcal{Q}, \mathcal{R}$
P3: $\clubsuit, \heartsuit, \spadesuit, \diamondsuit$

ordem global: $\clubsuit, \alpha, \beta, \mathcal{P}, \heartsuit, \mathcal{Q}, \gamma, \mathcal{R}, \spadesuit, \diamondsuit$

Consistência Seqüencial =
interleaving arbitrário que **preserva a ordem**
de referências à memória de programas seqüenciais

Atomicidade

- **Definição:**
  um acesso pelo processador $\mathcal{P}$ à variável X é **atômico** se nenhum outro processador pode acessar qualquer cópia de X enquanto o acesso pelo processador $\mathcal{P}$ estiver ocorrendo.
- atomicidade no hardware pode ser relaxada
  - ⋆ o hardware pode *parecer* que tem comportamento atômico sem que todas as cópias sejam atualizadas simultaneamente
- Atomicidade é importante para sincronismo baseado em operações read-modify-write
- Problemas com ordenação podem ocorrer mesmo com atomicidade
  - ⋆ atomicidade é uma propriedade de acessos à variáveis individuais, e não garante ordenação de atualizações de variáveis distintas.
  - ⋆ operações atômicas são serializadas pelos mecanismos de escrita em memória

Ordenação de Execução (de Programas)

- Considere diferentes seqüências de execução de instruções

```
inicialmente A=B=C=0
P1                P2                P3
a:  A←1           c:  B←1           e:  C←1
b:  print BC      d:  print AC      f:  print AB
```

- Só podemos observar indiretamente a seqüência real de eventos, com base no que é impresso

BC, AC, AB representa os valores impressos

- ▷ se processo é executado em ordem, um conjunto de seqüências é possível: acbdef [10, 10, 11], ecdfab [11, 01, 01], etc
- ▷ se instruções podem executar fora de ordem, outras seqüências são possíveis: adbcfe [00, 10, 11] e indesejáveis?

Ordenação de Execução (de Programas)

- Cada *interleaving* sugere que um conjunto específico de valores será impresso **se os acessos à memória são atômicos**
  - ▷ se acessos à memória não são atômicos, diferentes processadores podem perceber seqüências distintas:

    `[01, 10, 01]` `P1:` e⟶c
    `P2:` a⟶e
    `P3:` c⟶a

- atomicidade + operações de memória ordenadas ⟶ **ordenação forte**
- num sistema que é **seqüencialmente consistente** os eventos são **fortemente ordenados**

Coerência entre caches (revisitada)

- Informalmente, coerência entre caches é um método para fazer com que escritas em qualquer local na memória sejam percebidos na mesma ordem pelos outros processadores
  - ▷ coerência se aplica a referências a um único endereço, ao invés de ordenação de referências a endereços distintos (=consistência)
  - ▷ caches podem ser encaradas como filas de escrita (enormes) com atrasos potencialmente muito longos (por vezes infinitos)
  - ▷ coerência entre caches é um mecanismo que dá a aparência de atomicidade

Consistência Relaxada

- **Consistência seqüencial** pode diminuir desempenho porque a consistência é forte e as restrições de ordenamento tbém
- **Consistência relaxada:** pode relaxar ordem de execução **ou** atomicidade da escrita
  - ▷ relaxa ordem de escritas para leituras
  - ▷ relaxa ordem de escritas para escritas
  - ▷ relaxa ordens de leituras para leituras, e leituras para escritas
  - ▷ lê escritas pelos outros adiantadamente
  - ▷ lê escritas próprias antecipadamente
- relaxamento é geralmente determinado pela implementação em hardware vários modelos já foram propostos
- ∃m mecanismos que permitem ignorar relaxamento

Consistência Relaxada

- Consistência seqüencial pode diminuir desempenho
  porque a consistência é forte e as restrições de ordenamento tbém
- **Consistência relaxada:** pode relaxar
  ordem de execução **ou** atomicidade da escrita
  - ⋆ relaxa ordem de escrita para leitura
  - ⋆ relaxa ordem de escrita para escrita
  - ⋆ relaxa ordens de leitura para leitura, e leitura para escrita
  - ⋆ lê escrita pelos outros adiantadamente
  - ⋆ lê escritas próprias antecipadamente
- relaxamento é geralmente determinado pela implementação em hw
- ∃ mecanismos que permitem ignorar relaxamento
  - ▷ cercas *(fence operations)*
  - ▷ semântica das instruções de sincronização explícitas

Cercas em memória

Cercas são instruções para *fences*
forçar serialização dos acessos à memória

Processadores com modelos de ordenação fracos nos quais
`load`'s e `store`'s para endereços distintos podem ser re-ordenados
devem prover instruções para forçar a serialização
dos acessos à memória

Cercas são operações custosas
mas paga-se o custo da serialização somente quando necessário

Ordenação Fraca

- **Ordenação fraca:**
  não se presume nada sobre
  ordenação de operações **entre** pontos de sincronização explícita
- regras informais:
  - ▷ instruções de sincronização notificam hardware
    (controladores de caches)
  - ▷ ações para manter consistência ocorrem somente nas instruções de sincronização

Programas propriamente sincronizados

```
processo 1                    processo 2
   ...                        ...
   acquire(mutex);            acquire(mutex);
      <regiao critica>           <regiao critica>
   release(mutex);            release(mutex);
```

Variáveis de sincronização (mutexes) são disjuntas das variáveis com dados
acessos à variáveis compartilhadas para escrita são protegidos por regiões críticas
⟶ não há corridas exceto pelos semáforos/locks

Em geral, não é possível provar que um programa é livre de corridas no acesso aos dados.

Programas propriamente sincronizados

```
processo 1                    processo 2
   ...                        ...
   acquire(mutex);            acquire(mutex);
   fence;                     fence;
      <regiao critica>           <regiao critica>
   fence;                     fence;
   release(mutex);            release(mutex);
```

Modelos de ordenação relaxados permitem a re-ordenação de instruções pelo compilador, ou processador, desde que a re-ordenação não atravesse uma cerca

Além disso, o processador não pode executar especulativamente, ou fazer busca antecipada, atravessando cercas.

Resumo

- Primitivas de sincronização em barramento (que serializa acessos) `test_and_set`, `fetch_and_`Φ
- Primitivas de sincronização distribuídas (sem serializar acessos) `load-linked` e `store-conditional`
- coerência de caches ⟶ mesmo endereço
- consistência de memória ⟶ endereços distintos
- Consistência seqüencial: a ordem interna dos acessos de cada processo é respeitada
- Consistência fraca: ordem de acessos não é respeitada
- Cercas garantem pontos de sincronização ordenados

Entrada e Saída

- **Desempenho**
- **Tipos e Características de Dispositivos**
- **Arquitetura do Sistema de E/S**
  - ▷ **Barramentos**
  - ▷ **Processamento de E/S**
- Discos
  - ▷ Sistemas de Discos de Alto Desempenho

Desempenho

I/O certainly has been lagging in the last decade
Seymour Cray, 1976

Also, I/O needs a lot of work
David Kuck, 1988

If, ..., performance of CPUs improves as 55% per year and I/O did not improve, every task would become I/O bound
H&P, 2001

Quem se importa com desempenho de E/S?

- Desempenho de CPU cresce 55% ao ano
- Desempenho de sist de E/S é limitado por partes *mecânicas* (discos)
  melhora <10% aa em operações/segundo ou MBytes/segundo
- Lei de Amdahl:
  ganho de desempenho é limitado pela parte mais lenta
  - ▷ 10x CPU e 10% E/S ⟶ desempenho cresce 5x perde 50%
  - ▷ 100x CPU e 10% E/S ⟶ desempenho cresce 10x perde 90%
- Gargalo em E/S
  - ▷ fração cada vez menor do tempo na CPU
  - ▷ utilidade/valor cada vez menor de CPUs mais rápidas

Quem se importa com desempenho de E/S?

Se desempenho de E/S é limitado por partes mecânicas,

e desempenho da eletrônica segue a Lei de Moore,

então é melhor aumentar complexidade da eletrônica para melhorar desempenho global dos sistemas:

- reduzir número de fios + aumentar veloc da comunicação USB
- aumentar complexidade do controlador de disco caches+DMA
- melhorar a saída do adaptador gráfico mesmo monitor

Desempenho de E/S

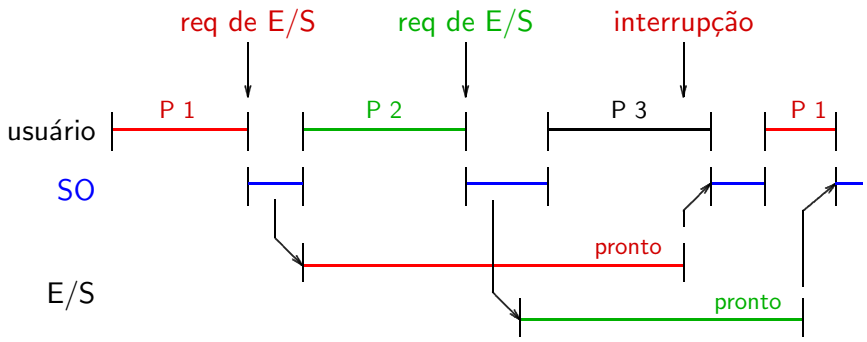
There is an old network saying:
Bandwidth problems can be cured with money.
Latency problemas are harder because the speed of light is fixed,
you can't bribe God David Clark

- Produção *(throughput)*
  - ▷ largura de banda ou operações por segundo
- Latência
  - ▷ tempo de resposta – *a produtividade dos humanos cresce super-linearmente se o tempo de resposta diminui*
  - ▷ tempo decorrido – *depende de concorrência*

latência de operações escondida se concorre com computação

Concorrência entre E/S e Computação

E/S concorre com computação de maneiras complexas



Desempenho

$$\mathcal{T}_{\text{tarefa}} = \mathcal{T}_{\text{cpu}} + \mathcal{T}_{\text{ES}} - \mathcal{T}_{\text{concorr}}$$

Exemplo 1: E/S completamente escondido: $\mathbf{10 = 10 + 4 - 4}$

Exemplo 2: acelera CPU 2x, qual é o novo desempenho?

a) $\mathcal{T}_{\text{tarefa}} = \mathbf{5 + 4 - 4 = 5}$ melhor ← esconde

b) $\mathcal{T}_{\text{tarefa}} = \mathbf{5 + 4 - 0 = 9}$ pior ← expõe

c) $\mathcal{T}_{\text{tarefa}} = \mathbf{5 + 4 - 2 = 7}$ médio ← esconde um pouco

Exemplos de Aplicações

- Supercomputadores
  - taxa de transferência é importante
    ⟶ muitos MBytes/seg para arquivos grandes
- Processamento de Transações
  - taxa de serviço é importante
  - acessos aleatórios aos dados
    ⟶ taxa de acessos a disco por segundo
- Sistemas de Arquivos para Multi-programação (ex. Unix)
  - tratamento de arquivos pequenos é importante
  - acessos seqüenciais
  - muitas criações e remoções

Características dos Dispositivos

- **Comportamento**
  - entrada (lê uma vez só)
  - saída (escreve uma vez só)
  - armazenagem (lê muitas vezes, também escreve)
- **contra-parte**
  - humano
  - computador
- **taxa de transferência**
  - taxa de pico
  - taxa sustentada (sustentável em condições "normais")

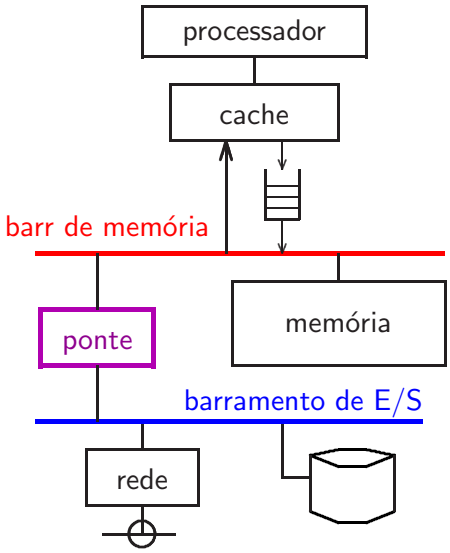
Tipos de Dispositivos

dispositivo	comportamento	parceiro	taxa [bytes/seg]
teclado	entrada	humano	10
mouse	entrada	humano	20
scanner	entrada	humano	400 K
monitor gráf	saída	humano	60 M
rede local	E/S	computador	0.5-10 M
fita	armazenagem	computador	2 M
disco	armazenagem	computador	2-10 M

Arquitetura do Sistema de E/S

Hierarquia de vias:
largura de banda é menor
a medida em que desce
na hierarquia
barramentos distintos
em cada nível

Processamento de E/S:
controlado por programa
ADM
processadores de E/S



Características dos Barramentos

Opção	alto desempenho	baixo custo
linhas de dados, endereços $\neq$s	sim	não
largura do barram de dados	largo	estreito
tamanho das transferências	rajadas	palavra
número de mestres	mais de um	um
barramento em pipeline	sim	não
operação	síncrona	assíncrona

Características dos Barramentos

- Barramento ≡ meio de comunicação compartilhado ⟶ *broadcast*
- Barramento de E/S: dispositivos com
  grande faixa de valores de vazão e latência
  - ▷ **Mestre** é capaz de iniciar transferência
  - ▷ **Escravo** responde aos comandos do mestre
- Num barramento em pipeline, há dois tipos de transações:
  comandos/requisições e respostas
  transação = requisição ⟶ *processamento* ⟶ resposta

Tipos de Barramentos – CPU-mem

- Barramento que interliga CPU e memória
  - ∗ desempenho muito elevado:
    800MByte/s a 64Gbyte/s
    largura de 64 a 256 bits,
    relógio de 100 a 400 MHz
    pode usar as duas bordas do relógio *(double data rate)*
  - ∗ projeto altamente otimizado
  - ∗ número de dispositivos é limitado pelo projeto # pequeno!
    menor comprimento ⟶ maior velocidade
  - ∗ tendência: maior velocidade e menor largura
- Barramento entre CPU e cache
  - ∗ desempenho mais alto ainda
- Barramento de E/S

Tipos de Barramentos – E/S

- Barramento que liga CPU e memória
- Barramento entre CPU e cache
- Barramento de E/S
  - ∗ compatibilidade com algum padrão é importante (PCI, SCSI)
  - ∗ projetado para tolerar número variado de dispositivos
  - ∗ projetado para tolerar diversidade em vazão e latência
  - ∗ maior comprimento ⟶ menor velocidade
  - ∗ recentemente:
    usa mais eletrônica para simplificar (menos fios)
    e/ou
    maior desempenho (protocolos mais complexos)

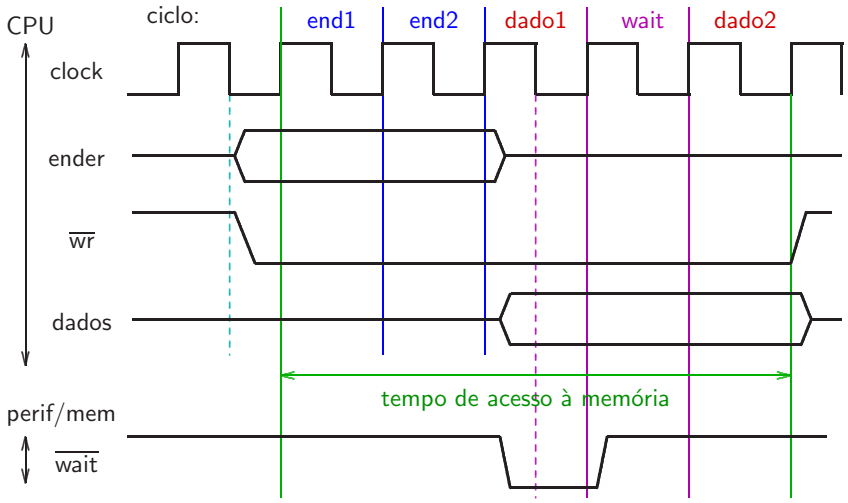
Método de Comutação do Barramento

- Barramento com comutação de circuito *circuit switched*
  - similar a um telefonema
  - posse do barramento é mantido até que transação completa
  - protocolo mais simples
  - latência dos dispositivos afeta utilização do barramento
- Barramento com comutação de pacotes *packet switched*
  - similar ao envio de uma carta
  - barramento é liberado após envio do comando
    outro mestre pode utilizar barramento
    até que resposta seja emitida
  - protocolo mais complexo mas permite segmentação
  - circuito de controle mais complexo,
    mas utilização do barramento é maior

Barramento Síncrono *vs* Assíncrono

- **Barramento Síncrono**
  - maior velocidade de operação ⟶ maior vazão
  - projeto elétrico complexo e restrições fortes na temporização
  - maior desempenho porque não tem realimentação
    sinais percorrem barramento em somente um sentido:
    mestre ⟶ escravo
- Barramento Assíncrono

Barramento Síncrono

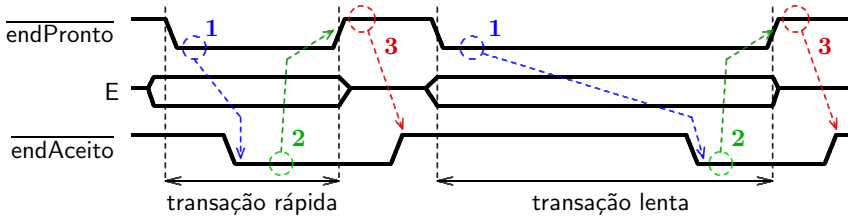


Barramento Síncrono *vs* Assíncrono

- Barramento Síncrono
- **Barramento Assíncrono**
  - menor velocidade de operação ⟶ menor vazão
  - projeto elétrico/temporal mais simples
  - pior desempenho porque tem realimentação
    sinais percorrem todo o barramento em dois sentidos:
    mestre(comando)⟶ escravo ; escravo(resposta)⟶ mestre
  - a cada evento, sinais devem ser sincronizados aos relógios internos dos dispositivos
    ⟶ relógios ≠s em dispositivos distintos (freqüência e fase)

Barramento Assíncrono

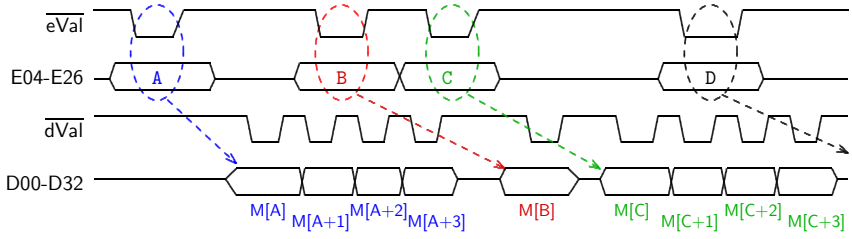
Transação de Endereçamento



Seqüência de eventos

1: mestre inicia transação	enderPronto	↘
escravo responde com	enderAceito	↘
2: mestre responde com	enderPronto	↗
3: escravo completa transação com	enderAceito	↗

Split Transaction Bus



- Barramento é liberado após envio do comando
  outro mestre pode usar barram até que resposta seja emitida
- protocolo: *pended packet switched*
- maior utilização do barramento
- pacotes com identificador de transação

Pergunta

Num sistema de E/S, a CPU é o mestre ou o escravo no barramento?

a) mestre
b) escravo
c) mestre e escravo
d) mestre ou escravo
e) nenhuma das acima

Resposta

Num sistema de E/S, a CPU é o mestre ou o escravo no barramento?

a) mestre
b) escravo
c) mestre e escravo
d) mestre ou escravo, dependendo se transação é leitura ou escrita
e) nenhuma das acima

Pergunta

Qual o tipo de barramento que permite maior nível de sobreposição
de comunicação (através do barramento)
com computação (pela CPU e/ou pelos dispositivos)?

a) barramento síncrono
b) barramento assíncrono
c) *split transaction* síncrono
d) *split transaction* assíncrono
e) nenhum dos acima

Resposta

Qual o tipo de barramento que permite maior nível de sobreposição
de comunicação (através do barramento)
com computação (pela CPU e/ou pelos dispositivos)?

a) barramento síncrono
b) barramento assíncrono
c) *split transaction* síncrono
d) *split transaction* assíncrono
e) nenhum dos acima

As transações divididas permitem maior superposição
entre as transações dos diversos pares mestre-escravo...

Processamento de E/S (i)

- **E/S controlada por programa**
  - CPU gerencia explicitamente todas as transferências
    loop: `load` ⟶ `store`
  - pode causar enorme desperdício de tempo de CPU
  - não é muito usado — há casos especiais interessantes
    - ▷ aplicações embutidas de baixo custo
    - ▷ SO verifica todos dispositivos lentos de uma vez
- Acesso Direto à Memória
- Processadores de E/S dedicados

Processamento de E/S (ii)

- E/S controlada por programa
- **Acesso Direto à Memória** — *Direct Memory Access = DMA*
  - Processador inicializa controlador de DMA com
    `endereço inicial`,
    `tamanho`,
    `tipo da transferência`
  - controlador de DMA dispara transferência de um bloco
    e gera interrupção após completar transferência
  - bom para volumes grandes e periféricos gulosos
- Processadores de E/S dedicados

Processamento de E/S (iii)

- E/S controlada por programa
- Acesso Direto à Memória
- **Processadores de E/S dedicados**
  - controladores capazes de executar 'programas' de E/S
    podem ser quase tão sofisticados quanto a CPU
  - podem ser de uso geral ou dedicado
  - e.g. canais de E/S da IBM

Comunicação com Processadores de E/S

Início/controle das operações

- **Periféricos mapeados como memória**
  - acesso em faixa de endereços de E/S para disparar operação
  - dispositivos no mesmo espaço de endereçamento que a memória
  - faixa de endereços de E/S são protegidos pelo SO
  - endereços não podem ser carregados nas caches (em geral)
  - dispositivo modelado por estrutura de dados
    $\longrightarrow$ programação simples
- Periféricos acessados com instruções especiais

Comunicação com Processadores de E/S

Início/controle das operações

- Periféricos mapeados como memória
- **Periféricos acessados com instruções especiais**
  - instruções especiais iniciam operações de E/S `in, out`
  - instruções são previlegiadas
    executam somente em modo supervisor
  - difícil de modelar dispositivos em linguagem de alto nível

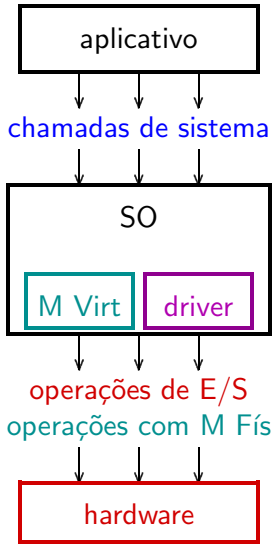
Comunicação com Processadores de E/S

Notificação de término das operações

- **E/S por programa** *polling*
  - ▷ processador fica em loop até que bit de status mude
  - ▷ verificações periódicas
  - ▷ latência elevada
- **Interrupção**
  - ▷ quando operação completa, interrompe processador
  - ▷ troca de contexto para tratar evento
  - ▷ baixa latência

Gerenciamento de E/S

- SO gerencia recursos de E/S
  - ⋆ serializa requisições para recursos compartilhados
  - ⋆ aloca espaço em disco, etc
- aplicativos invocam serviços do SO
  chamadas de sistema
- SO converte requisições em
  invocações de drivers
  - ⋆ driver contém programa específico para dispositivo ⟶ comandos, registradores, etc
- driver comanda dispositivo
  - ⋆ operações específicas para dispositivo ou barramento



Proteção de E/S

- Instruções de E/S são previlegiadas
  - ▷ somente o SO pode executar instruções de E/S
- Dispositivos mapeados em memória são protegidos
  contra acessos em modo usuário,
  e seus endereços/valores não são armazenados em cache
  - ▷ somente o SO pode acessar áreas de E/S mapeadas em memória
- Código do SO verifica cada chamada de sistema por usuários
  - ▷ se operação é permitida, então é executada

resumo – Sistemas de E/S

- $\mathcal{T}_{\text{tarefa}} = \mathcal{T}_{\text{cpu}} + \mathcal{T}_{\text{ES}} - \mathcal{T}_{\text{concorr}}$
- Hierarquia de barramentos — desempenho
  des(CPU-cache) $\gg$ des(cache-mem) $\gg$ des(E/S)
- Tipos de barramento:
  - ⋆ comutação de circuitos × comutação de pacotes
  - ⋆ síncrono × assíncrono × split transaction
- Processamento de E/S
  - ⋆ por programa
  - ⋆ acesso direto a memória
  - ⋆ canais de E/S
- Mapeamento do espaço de endereçamento
  - ∗ como memória × como E/S
- Notificação de término: interrupção × *polling*

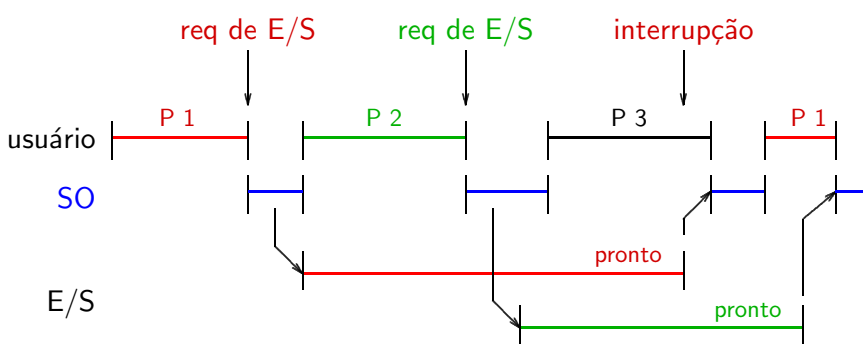
Entrada e Saída

- Desempenho
- Tipos e Características de Dispositivos
- Arquitetura do Sistema de E/S
- **Discos**
  - ▷ Mecanismo, componentes de gravação e de posicionamento
  - ▷ Controlador
  - ▷ **RAID** – Sistemas de Discos de Alto Desempenho

fonte:
[RW94] *An introduction to Disk Drive Modeling*,
Ruemmler & Wilkes, IEEE Computer 27(3):17-28, Mar 1994

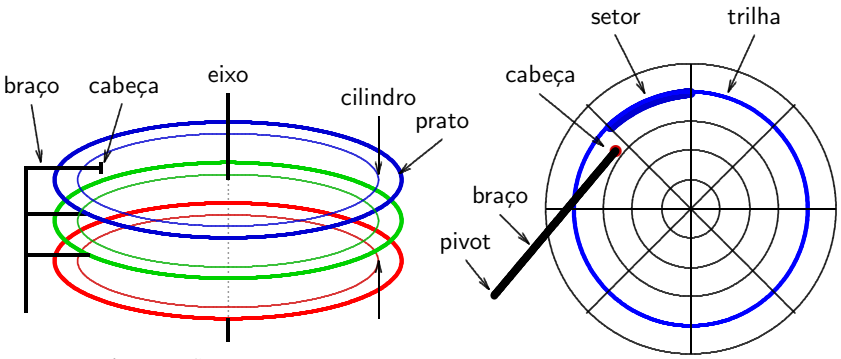
revisão – Concorrência entre E/S e Computação

E/S concorre com computação de maneiras complexas



$$\mathcal{T}_{\text{tarefa}} = \mathcal{T}_{\text{cpu}} + \mathcal{T}_{\text{E/S}} - \mathcal{T}_{\text{concorr}}$$

Discos Magnéticos



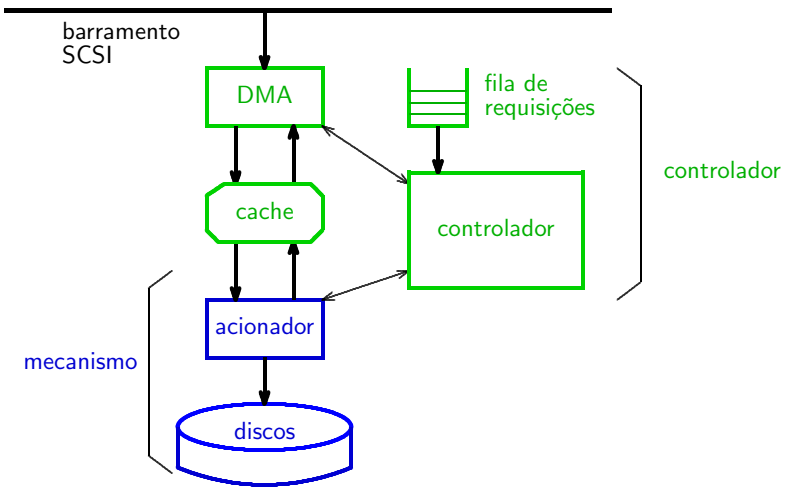
Ruemmler & Wilkes, Computer Mar94

Discos Magnéticos – Parâmetros Típicos

Característica	mín	máx
diâmetro [polegadas]	1,0	* 3,5
capacidade formatada [GB]	1	>100
discos/pratos	1	20
trilhas por superfície	6.000	25.000
setores por trilha	100	600
bytes por setor	512	4.096
velocidade [rpm]	3.600	15.000
cache [MB]	0,125	≥4
taxa transferência [MB/s]	2,5-5	27-40

* tamanho típico (+popular) em 2000

Discos – organização



Ruemmler & Wilkes, Computer Mar94

Mecanismo – componentes de gravação

- Diâmetro: 1.0, 1.3, 2.5, 3.5 ,8 polegadas
- densidade linear de gravação: 100 Kbpi (bits/inch)
- densidade de trilhas: 20 Ktpi
- efeito combinado: densidade por área cresce 60% aa
- velocidade 3.600, 7.200, 10.000, 15.000 cresce 12% aa
- uma cabeça ativa por vez, taxa de leitura $\geq$ 100 Mbps
- conteúdo de um setor
  - ▷ número do setor;
  - ▷ espaço;
  - ▷ informação do setor com Cód Det Corr Erros;
  - ▷ espaço;
  - ▷ ...

Mecanismo – componentes de posicionamento (i)

- Densidade é tão alta que noção de cilindro é quase irrelevante
- busca consiste de
  - ▷ aceleração até atingir $1/2\ V_{\text{máx}}$
  - ▷ $V_{\text{máx}}$ em distâncias longas
  - ▷ desaceleração até trilha desejada
  - ▷ estabilização da cabeça sobre a trilha (1-3 ms)
  - ▷ seeks longos: $T \propto \text{distância}\ \ (V_{\text{máx}})$
  - ▷ seeks médios: $T \propto \sqrt{\text{distância}}$
- recalibrar posição a cada 15-30 min, durante 500-800 ms
- acompanhamento de trilhas

Mecanismo – componentes de posicionamento (ii)

- Densidade é tão alta que noção de cilindro é quase irrelevante
- busca: aceleração ; curso ; desaceleração ; estabilização
- recalibrar posição a cada 15-30 min, durante 500-800 ms
- acompanhamento de trilhas
  - ▷ troca de cabeças $\longrightarrow$ reposicionar braço ($\approx$ 0.5-1.5 ms)
  - ▷ troca de trilha $\longrightarrow$ estabilizar braço ($\approx$ 1-3 ms)
  - ▷ controlador tenta leitura otimista assim que chega na trilha

    pode fazer escrita otimista?

    ($\approx$ 0.75 ms até estabilizar para escrever)

Mecanismo – leiaute dos dados

- Disco visto pelo SO como vetor linear de blocos (256-1024 bytes)
- controlador mapeia vetor nos setores físicos 1D $\rightsquigarrow$ 3D
  V[i] $\rightsquigarrow$ d[setor, trilha, superfície]
- #bits cresce $\approx$ linearmente com comprimento da trilha
  zoneamento: número de setores depende do raio
  $\exists$ 3-50 zonas com mesmo número de setores / zona
- deslocamento de setores nas trilhas:
  setor**0** de cada trilha deslocado para esconder
  tempo de reposicionamento *track skewing*
- trilhas/setores sobressalentes: referências a setores danificados
  são re-mapeadas para setores/trilhas de reserva
  - ⋆ na formatação – pula endereço da trilha com defeito *slip sparing*
  - ⋆ em uso – re-mapeia endereço do setor/trilha para sobressalentes

Controlador (i)

- Funções do controlador SCSI
  - ▷ mediar acessos ao mecanismo
  - ▷ executar sistema de acompanhamento de trilhas
  - ▷ transferir dados entre disco e cliente
  - ▷ gerenciar buffers/cache
- operação do controlador custa 0.3-1 ms (caindo lentamente)
  eletrônica segue Lei de Moore mas
  funcionalidade mais complexa com o tempo
- interface com barramento
- buffer usado como cache

Controlador (ii)

- Funções do controlador SCSI
- operação do controlador custa 0.3-1 ms (caindo lentamente)
  eletrônica segue Lei de Moore mas funcionalidade cada vez mais complexa
- interface com barramento
  - ▷ transferências em modo síncrono, na vel máxima do barramento
  - ▷ pode operar com split transactions (latências enormes)
  - ▷ $\exists$ buffer entre mecanismo e barramento por causa das diferenças de velocidade
- buffer usado como cache

Controlador – cache (leitura)

Políticas da cache: *read-ahead* ≈ busca antecipada

- usa × ignora dados em acerto parcial?
- ignora cache em requisições grandes? segmentação dos acessos
- mantém × descarta bloco após transferência?
- *on-arrival read-ahead*: assim que chegar na trilha, lê trilha toda
- *read-ahead* agressivo: atravessa trilhas e/ou cilindros
- *read-ahead* conservador: pára no final de trilha/cilindro
- cache associativa: particionar cache para ≠s seqüências entrelaçadas
- como cache do disco interfere com *buffer cache* do Unix?

Controlador – cache (escrita)

Cache pode corromper sistema de arquivos se faltar energia

- controlador avisa que completou operação após escrever na cache
- isso deve ser evitado na escrita de metadados
- se cache tem bateria, problema desaparece o que fazer no boot?
- ganhos adicionais
  - ▷ dados na cache sobre-escritos antes de gravados no disco
  - ▷ controlador pode otimizar escrita de vários blocos
  - ▷ mesmos problemas que fila de escrita – RAW, WAW
- como cache do disco interfere com *buffer cache* do Unix?

Cache com fila de comandos: controlador pode otimizar operações porque conhece geometria do disco

Operação de Discos

- Comandos
  - ▷ Latência/atraso no controlador + tempo na fila (OS)
  - ▷ 0,5ms se não encontra na cache, 0,1ms se encontra na cache
- Seek/busca movimentação do braço entre trilhas/cilindros
  - ▷ move a cabeça até a trilha desejada
  - ▷ tempo depende da posição inicial da cabeça
  - ▷ valores típicos médios entre 5-12ms
- Latência rotacional
  - ▷ espera até que setor desejado passe sob a cabeça (≈1/2 volta)
  - ▷ na média, 0.5/rpm ⟶ 0,5/(7200rpm/60spm)=4,2ms
- Transferência de dados
  - ▷ taxa de transferência entre 2 e 40 MByte/s

Desempenho

- Tempo médio de acesso
  = tempo médio de movimentação do braço (seek)
  + latência rotacional média
  + tempo de transferência
  + tempo do controlador
- Exemplo: 7200 rpm, 10MByte/s
  tempo médio de movimentação do braço: 10ms
  tempo do controlador: 0,5ms
  tempo para ler bloco de 4Kbytes (uma página)

  10ms + 0,5/(7200rpm/60spm) + 4KB/10MB/s + 0,5ms

  10ms + 4,2ms + 0,4ms + 0,5ms = 14,65ms

Desempenho (cont)

Localidade:
discos exibem localidade
⟶ em "acessos locais"
tempo de busca (seek) cai em 1/3

Cache:
buffer em memória (*Unix buffer cache*)
e na unidade de disco
⟶ latência cai para hit+transferência

distância # trilhas	fração
0	24%
15	23%
30	8%
45	4%
60	3%
75	3%
90	1%
105	3%
120	3%
135	2%
150-165	(3+2)%
180-195	(3+3)%

unix time-sharing

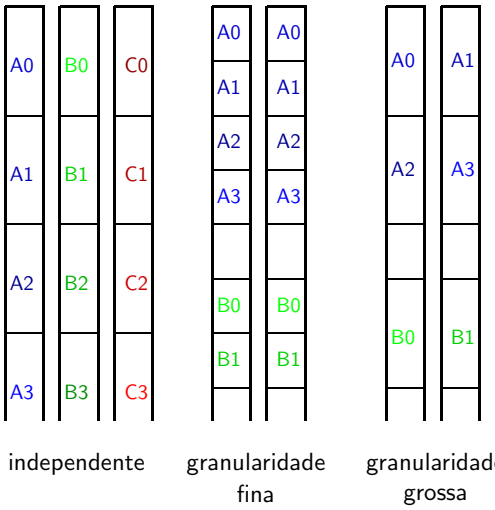
Tendências

Capacidade	↗ 100% aa	(2x / 1.0 anos)
Taxa de Transferência	↗ 40% aa	(2x / 2.0 anos)
Rotação+posicionamento	↘ 8% aa	(0.5x / 10 anos)
MB/$	↗ 100% aa	(2x / 1.5 anos)

Matrizes de Discos

Conjunto de discos individuais:
cada disco com seu braço/cabeça

Distribuição dos dados:
endereçamento
    independente
listras de
    blocos pequenos
listras de
    blocos grandes



Matrizes de Discos (i)

- Endereçamento independente
  - ▷ software/usuário distribui os dados
  - ▷ balanceamento de carga entre discos pode ser problemático
    ⟶ é difícil de sincronizar acessos em paralelo
- Listras de blocos pequenos *fine-grain striping*
- Listras de blocos grandes *coarse-grain striping*
- Granularidade escolhida em função da aplicação e tipo de carga

Matrizes de Discos (ii)

- Endereçamento independente
- Listras de blocos pequenos *fine-grain striping*
  - ▷ |bloco| = um bit, um byte, ou um setor
  - ▷ #discos∗|bloco| define menor quantidade de dados acessível
  - ▷ balanceamento de carga perfeito: **1** requisição atendida por vez
  - ▷ taxa efetiva de transferência ≈ N vezes melhor que um disco só
  - ▷ tempo de acesso pode aumentar,
    a não ser que discos sejam sincronizados
- Listras de blocos grandes *coarse-grain striping*
  - ⋆ paralelismo na transferência de grandes volumes de dados
  - ⋆ concorrência para transferências pequenas
  - ⋆ balanceamento de carga pela aleatoriedade
- Granularidade escolhida em função da aplicação e tipo de carga

Matrizes de Discos (iii)

- Endereçamento independente
- Listras de blocos pequenos *fine-grain striping*
  - ⋆ |bloco| = um bit, um byte, ou um setor
  - ⋆ #discos∗|bloco| define menor quantidade de dados acessível
  - ⋆ balanceamento de carga perfeito: **1** requisição atendida por vez
  - ⋆ taxa efetiva de transferência ≈ N vezes melhor que um disco só
  - ⋆ tempo de acesso pode aumentar, a não ser que discos sejam sincronizados
- Listras de blocos grandes *coarse-grain striping*
  - ▷ paralelismo na transferência de grandes volumes de dados
  - ▷ concorrência para transferências pequenas
  - ▷ balanceamento de carga pela aleatoriedade
- Granularidade escolhida em função da aplicação e tipo de carga

Mecanismos de Redundância

- Falhas em discos são parcela grande de falhas de hardware
  ⟶ striping aumenta o número de arquivos perdidos por falha
- Replicação dos dados
  espelhamento dos discos
  ⟶ permite leituras em paralelo
  ⟶ escritas devem ser sincronizadas
- Proteção com paridade
  ⟶ usar disco para manter a paridade

Redundant Arrays of Inexpensive Disks – RAIDs

- Conjuntos de discos pequenos e baratos resultam em alto desempenho e alta confiabilidade
  D = número de discos de dados no conjunto
  V = número de discos de verificação no conjunto
- Nível 1: discos espelhados (D=1, V=1)
  - ▷ desperdício é elevado
- Nível 2: código de detecção de erros (D=10, V=4)
  - ▷ mesmo tipo de código de detecção de erros usado com DRAMs
  - ▷ todos os bits do conjunto são lidos
  - ▷ agrega bits atualizados com bits que permanecem; recomputa a paridade
  - ▷ re-escreve todo o conjunto, incluindo a verificação

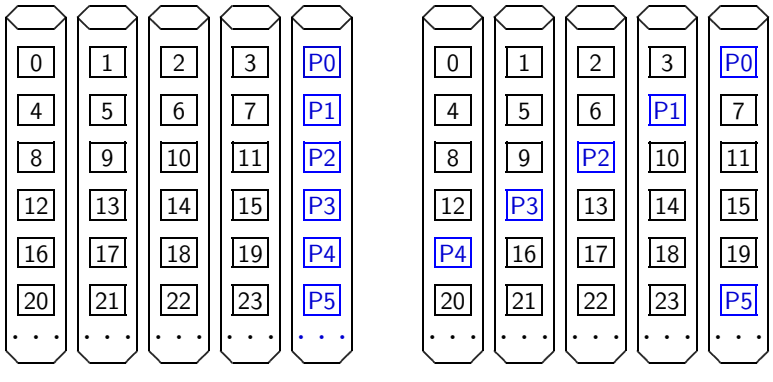
Redundant Arrays of Inexpensive Disks – RAIDs

- Nível 3: paridade com bits entrelaçados (D=4, V=1)
  - ▷ disco com falha é identificado facilmente pelo controlador
  - ▷ não é necessário código especial para descobrir disco em falha
- Nível 4: paridade com blocos entrelaçados
  - ▷ usado com blocos grandes
  - ▷ similar ao RAID 3, mas pode efetuar mais de um acesso com poucos dados a cada vez
  - ▷ escrita deve atualizar disco com dados e paridade

Redundant Arrays of Inexpensive Disks – RAIDs

- Nível 5: paridade distribuída com blocos entrelaçados
  - ▷ paridade é distribuída pelos discos no conjunto
  - ▷ atualizações distintas de paridade vão para discos distintos
- Nível 6: matriz bi-dimensional
  - ▷ matriz de dados é bi-dimensional, com paridade nas linhas e nas colunas
  - ▷ permite recuperação de duas falhas

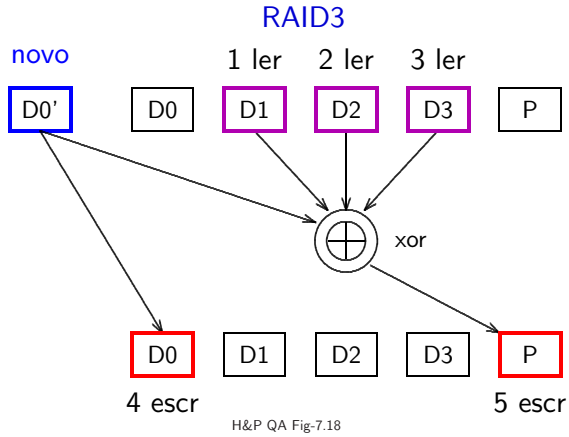
RAID 4/5 – Blocos de Paridade



H&P QA Fig-7.19

RAID – Atualização "Pequena"

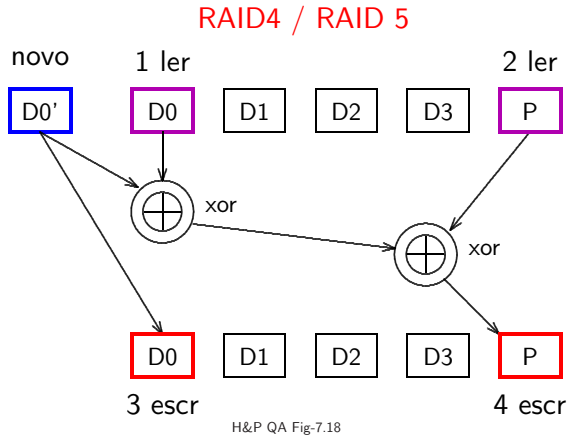
Qual é o número de operações de leitura/escrita nos discos individuais para efetuar escrita de poucos dados (= atualização pequena)?



H&P QA Fig-7.18

RAID – Atualização "Pequena"

Qual é o número de operações de leitura/escrita nos discos individuais para efetuar escrita de poucos dados (= atualização pequena)?



H&P QA Fig-7.18

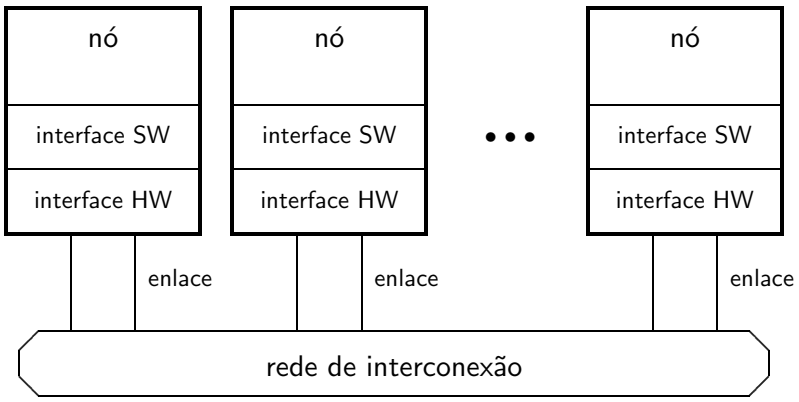
resumo – Discos

- Tempo médio de acesso
  = tempo médio de movimentação do braço (seek)
  + latência rotacional média
  + tempo de transferência
  + tempo do controlador
- Cache no controlador para tirar proveito de localidade
  - ⋆ falta de energia durante escrita de metadados corrompe sist de arquivos
  - ⋆ mesmos problemas que fila de escrita (riscos RAW a WAW)
- RAID – usar discos baratos para
  aumentar desempenho – striping
  e
  melhorar confiabilidade – paridade

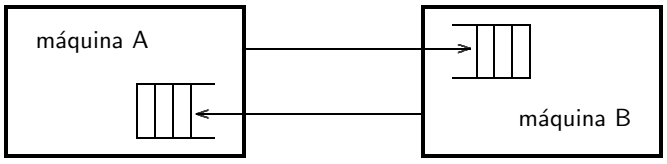
Redes e Clusters

- Protocolos
- Escopo, Tipos de Redes, Meios Físicos
- Espaço de projeto
  - ▷ topologias
  - ▷ comutação
  - ▷ controle
  - ▷ sincronicidade
- Roteamento
- Controle de Tráfego

Modelo de rede



Modelo de comunicação



- Quem transmite?
- quando transmite?
- como interpreta mensagem?
- como garante que mensagem chegou ao destino?
- como garante que mensagem está correta?
- **protocolo** determina as respostas

Protocolos

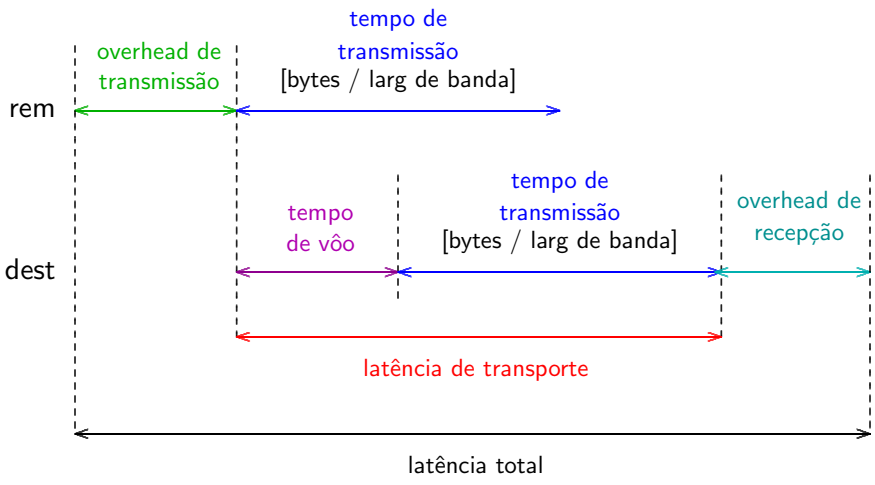
Protocolo deve contemplar:

- endereçamento ⟶ máquinas + processos
- proteção entre processos na mesma máquina
- tipos de mensagem: *request, reply, acknowledgement*
- entrega confiável
  - ▷ conteúdo alterado? checksum
  - ▷ mensagem perdida? retransmissão + temporizadores + buffers
- minúcias:
  - ▷ ordem dos bytes
  - ▷ duplicação de mensagens
  - ▷ ordenação de mensagens
  - ▷ controle de fluxo (≠s velocidades de/entre fonte e destino)

Anatomia de uma mensagem

ender fonte	*nome de máquina*
ender destino	
processo fonte	*nome de processo ou porta*
processo destino	
tipo de msgm	{*req*, *rsp*, *ack*, *crtlFluxo*}
núm seqüência	*seqüenciamento, duplicações*
controle de fluxo	*buffer overflow/underflow*
checksum	*detecção de erros*
carga	O QUE REALMENTE INTERESSA

Parâmetros de Desempenho



Equação do desempenho

latência total

= overhead_transmissão — proc 100% ocupado com envio
+ tempo_de_vôo — veloc da luz + atrasos hw
+ |mensagem| / largura_de_banda — [bits / bits/s]=[s]
+ overhead_recepção — proc 100% ocupado recebendo

- largura de banda medida no fio (inclui cabeçalhos) [bit/s]
- tempo de vôo é aprox 20cm/ns + atrasos nos repetidores [s]
- ovhead TX inclui tempo nos componentes de SW e HW [s]
- ovhead RX geralmente maior que de TX (+interrupção) [s]
- tamanho da mensagem inclui cabeçalhos [bit]

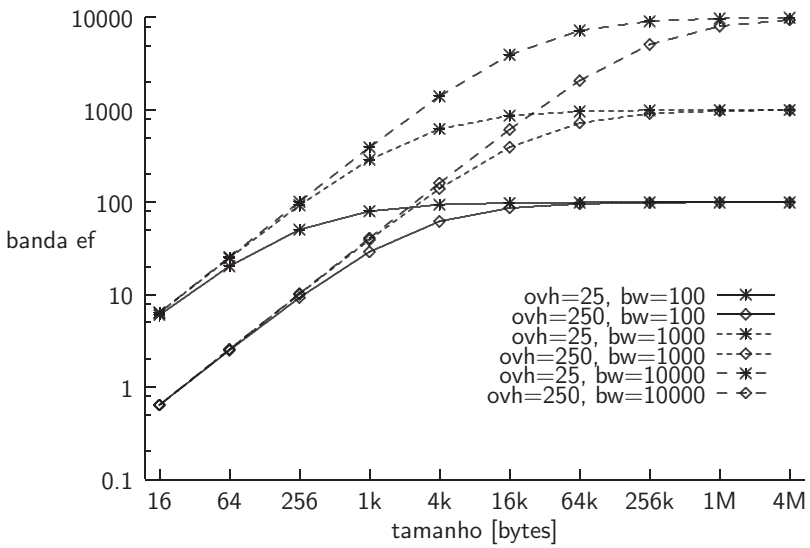
Largura de banda efetiva

- em redes de pequeno diâmetro, tempo de vôo pode ser ignorado
- em rede de longa distância, overheads de TX e de RX podem ser ignorados
- **latência total** pode ser simplificada

  latência total ≈ overheads + |mensagem| / largura_de_banda
- **largura de banda efetiva** entregue pela rede é

  larg de banda efetiva = |mensagem| / latência_total

Largura de banda efetiva



moral da estória: overhead DEVE ser pequeno

Escopo e Tipos de Redes

escopo	tipo	
interna ao CI	crossbar, completamente conectada	
motherboard	barramento	
System Area Network	barramento, anel, malha, cubo	SAN
Local Area Network	barramento, anel	LAN
Wide Area Network	malha esparsa, ponto-a-ponto	WAN

Meios Físicos

- *Flat cable*
- par trançado
- cabo coaxial
- fibra ótica
  - ▷ mono-modo
  - ▷ multi-modo
  - ▷ multi-côr
- ondas de rádio

Espaço de Projeto

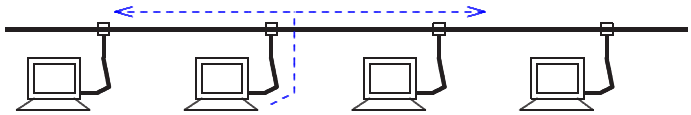
- **Critérios de desempenho**
  - ▷ latência – tempo de viagem de uma mensagem
  - ▷ largura de banda – quanto tráfego a rede pode tratar
  - ▷ conectividade – # de vizinhos imediatos
  - ▷ custo – rede representa qual fração do custo de HW
  - ▷ confiabilidade – caminhos redundantes
  - ▷ funcionalidade – combinação de msgns, tolerância a falhas

Espaço de Projeto (cont)

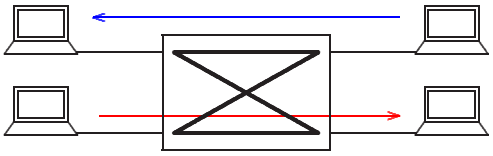
- **Parâmetros de projeto**
  - ▷ topologia – estática × dinâmica com sub-classes
  - ▷ modo de operação – síncrono × assíncrono
  - ▷ estratégia de comutação – circuitos, re-despacho, cut-through
  - ▷ estratégia de controle – centralizado × distribuído
- produto cartesiano destes 4 parâmetros define espaço de projeto

Meio Compartilhado *vs* Comutado

Meio compartilhado (Ethernet em cabo coaxial)



Meio comutado (Ethernet comutada)



Topologias – Redes com Conexão Estática

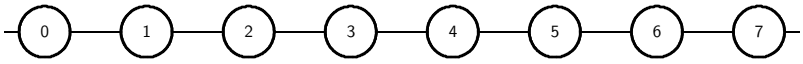
- Redes construídas com conexões diretas entre os nós da rede
- padrão de conexões fixado na construção da rede
- grau de um nó d: número de enlaces ligados ao nó
- diâmetro da rede D: caminho mais curto maximal entre $\forall$ 2 nós
- largura de um canal w: número de bits no canal/enlace
- diâmetro da bisecção: menor diâmetro da rede com 2 metades =s
  channel bisection width b: mínimo número de canais na bisecção
  wire bisection width $B = bw$: reflete densidade da fiação

Topologias (i) – Redes com Conexão Estática

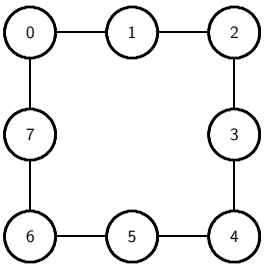
- Linear – N nós conectados por $N-1$ enlaces
  não é um barramento, enlaces podem ser bi-direcionais
  d=2 D=N-1 (elevado para N grande)
- anel – N nós conectados por N enlaces
  enlaces uni-direcionais
  d=2 D=N
  enlaces bi-direcionais
  d=2 D=$\lfloor N/2 \rfloor$
- *barrel shifter* – anel com enlaces adicionais para vizinhos a uma distância que é potência de 2: $i \rightleftharpoons j$ se $|j-i| = 2^r$
  N=2^n d=$2n$-1 D=$n/2$
- completamente conectado – cada nó é ligado a todos outros nós
  d=N-1 D=1

Topologias (ii) – Redes com Conexão Estática

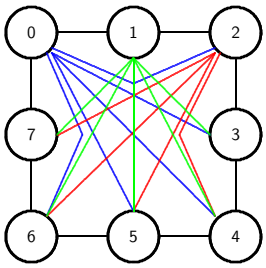
arranjo linear d=2 D=7



anel unidir d=2 D=8
anel bidir d=2 D=4



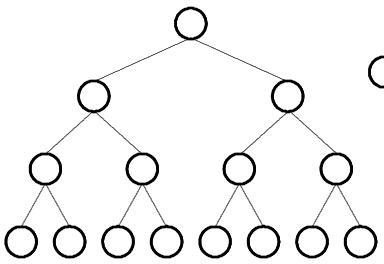
(quase) completamente conectado
d=7 D=1



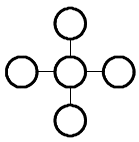
Topologias (iii) – Redes com Conexão Estática

- Árvore binária – N=2^k-1 nós em k níveis $d \leq 3$ D=$2(k-1)$
- estrela – árvore de dois níveis d=$\{1, N-1\}$ D=2
- árvore gorda – diâmetro dos canais aumenta próximo da raiz

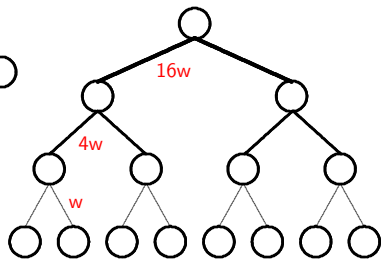
árvore binária
d≤3 D=2(k-1)



estrela
d=1,N-1 D=2



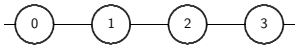
árvore gorda
d≤3 D=2(k-1)



Topologias (iv) – Redes com Conexão Estática

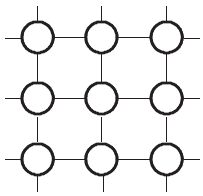
- Arranjo Linear é uma malha $k = 1$ (1D) com $N{=}n$ nós

  $d{=}2$ $D{=}N{-}1$



- Malha – rede k-dimensional com $N{=}n^k$ nós

  $d{=}2k$ (nós internos) $D{=}k(n{-}1)$

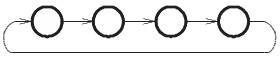


- toróide – malha $n \times n$ com conexões nas extremidades

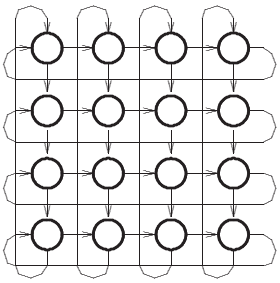
  $d{=}4$ $D{=}2\lfloor n/2 \rfloor$

Topologias (v) – Redes com Conexão Estática

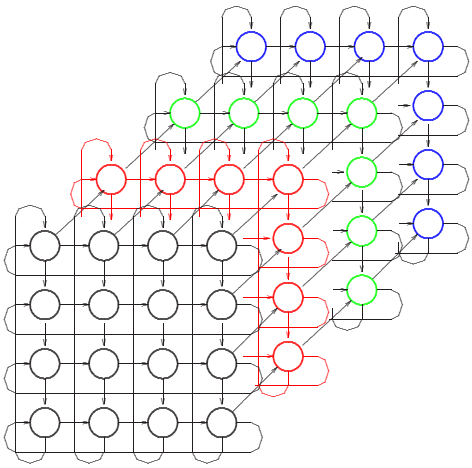
Anel (1D, k=4 n=1)



Malha (2D, k=4 n=2)



Cubo (toróide 3D, k=4 n=3)

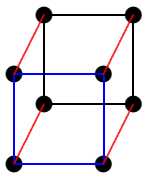


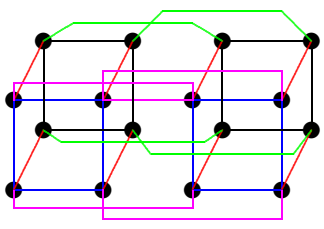
Topologias (vi) – Redes com Conexão Estática

- Cubos: n-cubo k-ário – interliga N nós,
  via enlaces com k nós/enlace, em cubo com dimensão n

$$N = k^n \,, \quad k = \sqrt[n]{N} \,, \quad n = \log_k N$$

dimensão 3
3 enlaces/nó





dimensão 4
4 enlaces/nó

quando $k = 2$ rede é hipercubo

Conexão Estática – Custo *vs* Desempenho

rede	d grau	D diâm	#enl	bissec	#nós
arranj linear	2	N-1	N-1	1	N
anel (1D)	2	$\lfloor N/2 \rfloor$	N	2	N
totmte conect	N-1	1	N(N-1)/2	$(N/2)^2$	N
árvore binária	3	2(h-1)	N-1	1	h=$\lceil \lg_2 N \rceil$
malha 2D	4	2(r-1)	2N-2r	r	r=$\sqrt{N}$
toro 2D	4	$2\lfloor r/2 \rfloor$	2N	2r	r=$\sqrt{N}$
hipercubo	n	n	nN/2	N/2	n=$\lg_2$N
n-cubo k-ário	2n	$n\lfloor k/2 \rfloor$	nN	$2k^{n-1}$	N=k^n

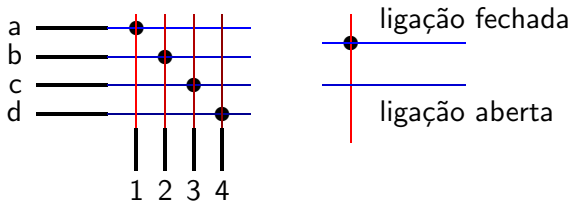
Conexão Estática – Custo *vs* Desempenho

Rede com 64 nós

topologia	bissecção	portas	enlaces
barramento	1	1	1
anel (1D)	2	2	128
toróide (2D)	16	4	192
hipercubo	32	6	256
totalmte conect	1024	255	2080

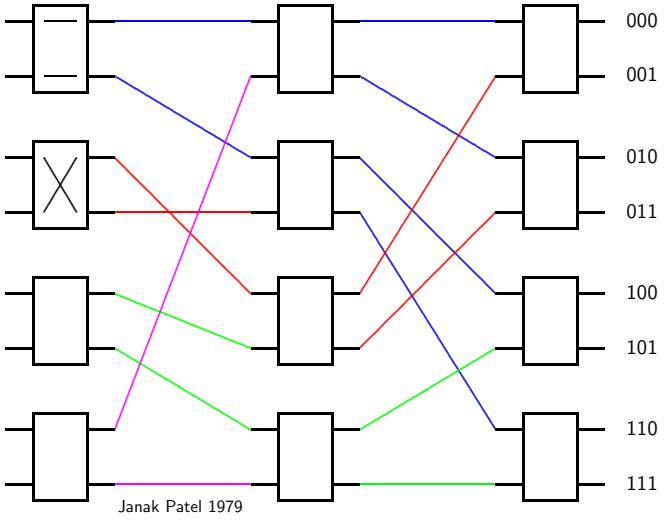
Topologias – Redes com Conexão Dinâmica

- barramento
  - ▷ conexão entre mestre e escravo definida a cada transação
- crossbar
  - ▷ é a rede mais cara ⟶ O(n^2) comutadores
  - ▷ é a rede mais eficaz $i, j \in \{0..n\}, i \neq j \Rightarrow P_i \rightleftharpoons P_j$



- redes multi-estágio (ômega, delta, ...)
  - ▷ custo menor que crossbar ⟶ O(n log n) comutadores

Topologias – R Conexão Dinâmica - rede Delta



Topologias (i) – Redes com Conexão Dinâmica

característica	barramento	multiestágio	crossbar
latência mínima	constante	$O(\log_k n)$	constante
banda / proc	O(w/n)..O(w)	O(w)..O(nw)	O(w)..O(nw)
complex cabeam	O(w)	$O(nw \log_k n)$	$O(n^2 w)$
complex comut	O(n)	$O(n \log_k n)$	$O(n^2)$
conectividade	$1 \longrightarrow 1$	†	‡

barramento: n proc, largura w

multi-est: nxn MIN, kxk comutadores de largura w

crossbar: nxn largura w

† algumas permutações e broadcast se rede livre

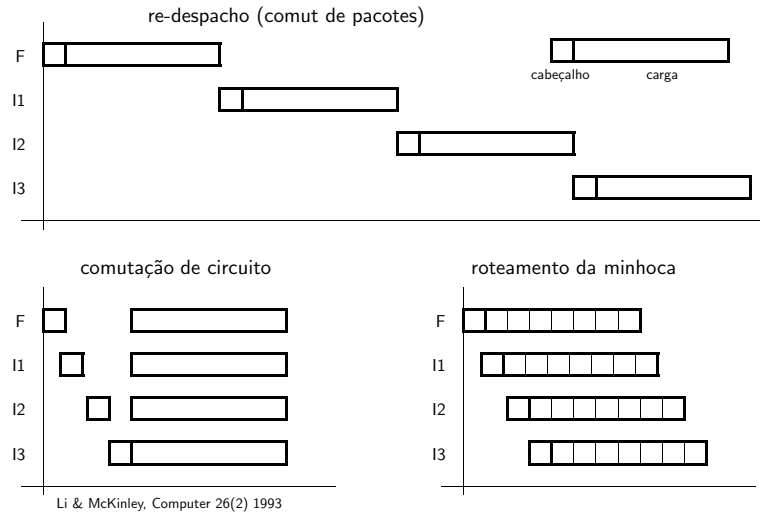
‡ todas permutações, uma de cada vez

Comutação

- **Comutação**
  - ▷ re-despacho (store-and-forward)
    $L_{\mathrm{rd}} \propto$ núm comutadores · |msgm|
  - ▷ cut-through – se cabeçalho bloqueado, comutador engole msgm
    $L_{\mathrm{ct}} \propto f$(núm comutadores) + |msgm|/banda · tempoTX
  - ▷ wormhole – se cabeçalho bloqueado, msgm bloqueia caminho
    $L_{\mathrm{wh}} \cong L_{\mathrm{ct}}$ se rede sem tráfego

Li & McKinley, *A survey of wormhole routing techniques in direct networks*, Computer 26(2),1993

Comutação



Li & McKinley, Computer 26(2) 1993

Roteamento

- **Roteamento**
  - ▷ source-based – emissor define rota até destino
    rota é parte da msgm
  - ▷ circuito virtual – negocia-se caminho antes de enviar
    nome do circuito é parte da msgm
  - ▷ destination-based – roteadores no caminho escolhem rota
    endereço destino é parte da msgm
    - ∗ determinístico
    - ∗ adaptativo
    - ∗ aleatório

Estratégia de Controle

- **Controle Centralizado**

topologia	P(bloqueios)	# comutadores	distância
crossbar	$\propto 1/(N-2)$	N^2	$\propto 1$
ômega	>crossbar	$N/2 \log N$	$\propto \log N$
árvore bin	?	$N-1$	$2(h-1)$
barramento	$\propto N$	1	1 ?

árvore – altura $h = \lceil \log N \rceil$

- **Controle Distribuído**
  - ▷ anel, malha, toróide
  - ▷ hipercubo
  - ▷ fully-connected

resumo – Redes

- Protocolo – endereços e rotas, confiabilidade, proteção
- **latência total**
  = overhead_transmissão proc 100% ocupado com envio
  + tempo_de_vôo veloc da luz + atrasos hw
  + |mensagem| / largura_de_banda [bits / bits/s]=[s]
  + overhead_recepção proc 100% ocupado recebendo
- **Parâmetros de projeto**
  * topologia – ligação estática × dinâmica
  * modo de operação – síncrono × assíncrono
  * estratégia de comutação – circuitos, re-despacho, cut-through
  * estratégia de controle – centralizado × distribuído
- topol: ligações estáticas cubo k-ário n-dim, completam conectado
- topol: ligações dinâmicas barramento, multi-estágio, crossbar

revisão – Redes

- Protocolo – endereços e rotas, confiabilidade, proteção
- **latência total**
  = overhead_transmissão proc 100% ocupado com envio
  + tempo_de_vôo veloc da luz + atrasos hw
  + |mensagem| / largura_de_banda [bits / bits/s]=[s]
  + overhead_recepção proc 100% ocupado recebendo
- **Parâmetros de projeto**
  * topologia – ligação estática × dinâmica
  * modo de operação – síncrono × assíncrono
  * estratégia de comutação – circuitos, re-despacho, cut-through
  * estratégia de controle – centralizado × distribuído
- topol: ligações estáticas cubo k-ário n-dim, completam conectado
- topol: ligações dinâmicas barramento, multi-estágio, crossbar

Ficou faltando...

- Onde é a interface com hierarquia de memória?
  CPU⟶ multiprocessador × rede⟶ multicomputador
- há um processador adicional dedicado à tratar comunicação?
  * como particiona as tarefas?
  * qual o custo da comunicação entre CPU_proc e CPU_com?
- o que fazer em HW e o que fazer em SW?
  * roteamento
  * checksum
  * ...
- proteção provida por SO × usuário

Aglomerados de PCs

- *Cluster* construído com PCs + rede local + GNU/Linux
- em (algumas) aplicações com granularidade grande:
  * preço/desempenho entre 3..50× o de sistemas dedicados (MPPs)
  * desempenho entre 1/3..1/2 do de sistemas dedicados (MPPs)
- componentes de um nó = componentes de um PC
  disco pode ser problemático xterms no DInf sem disco!!!
- já usamos labs do DInf como clusters
- referência: *Cluster Computing White Paper*

HW de rede

- problemas com padrão de comunicação complexos só atingem bom desempenho em MPPs
- problemas trivialmente paralelos não chegam a usar toda a banda disponível (Ethernet)
- maioria dos problemas entre estes dois extremos

Exemplos

Beowulf 2^+ Ethernet's, sw roteia tráfego entre as 2^+ interfaces físicas transparentemente à aplicação
banda 'suficiente', latência $\approx 100\mu$s

Myrinet rede comutada ($\longrightarrow$ multi-estágio)
banda 0,5–2Gbps, latência $\approx$ 2,6–3,2μs (MPI)

Infiniband barramento serial unidirecional, rede comutada, RDMA
banda 2×2Gbps (2,4,8), pode agregar 4,12 canais (2–96Gbps)
latência 160ns/comutador, ponta-a-ponta $\approx 6\mu$s

VIA (extinto?) comunicação entre espaços de endereçamento de usuário, com pouca intervenção do SO
banda $\geq$ 1Gbps, latência $\approx 10\mu$s, Zero-copy

Software

- ferramentas de programação
  - ∗ linguagens – C, C++, fortran, ada ...
  - ∗ bibliotecas – PVM (1989), MPI transformam PCs em clusters
  - ∗ depuradores – depuração de código e de desempenho
- gerência de recursos
  - ∗ instalação, administração
  - ∗ escalonamento de hw e sw
  - ∗ armazenamento
  - ∗ diagnóstico, monitoramento, disponibilidade
- funções do SO
  - ∗ multiplexar processos nos recursos físicos *resource mgmt*
  - ∗ prover abstrações úteis *beautification*

Ambientes e modelos de programação

Ambientes/modelos implementados em bibliotecas e middleware

- processos seqüenciais comunicantes (troca de mensagens)
  Calculus of Communicating Systems, Robin Milner
  Comunicating Sequential Processes, C A R Hoare
- Message Passing Interface ou MPI padrão de fato
- Parallel Virtual Machine ou PVM já foi o padrão
- data-parallel (SIMD) difícil de implementar c/ eficiência
- *message-driven* fast_messages + splitC
- memória compartilhada distribuída Shrimp
- e muitos outros...

Características de SOs para aglomerados

- Facilidade de gerenciamento
  single system image – interface comum esconde diversidade
- estabilidade
  - ∗ robusteza contra processos errados/maliciosos
  - ∗ recuperação de falhas por reconfiguração dinâmica
  - ∗ usabilidade sob carga
- desempenho, especialmente das partes críticas:
  escalonamento, gerenciamento de memória, arquivos e E/S, rede
- extensibilidade dos nós, SO aberto
- escalabilidade intra-nós: depende dos nós individuais
  . inter-nós: é limitada pela rede
- heterogeneidade mesmo SO em ≠s processadores
  . mesmas APIs em ≠s computadores

Funcionalidades: SO x middleware

- SOs são complexos ⟶ alterações afetam partes indesejadas
- funções no *middleware* podem ser ex-portadas para outros SOs
  exemplo: memória compartilhada distribuída (DSM) em mw

Single System Image

- Extensões do SO que permitem compartilhamento transparente de recursos
- 2 variantes:
  * administração do sistema (mw?)
  * escalonamento de tarefas + uso transparente de recursos remotos
- SSI implementado no SO implica em compartilhamento de estado entre TODOS os nós de um cluster
  ⟶ manter informação atualizada é caríssimo!!

Distributed Shared Memory

- Permite compartilhamento de memória (RAM) entre todos os nós
- paradigma de programação paralela para usar mem compartilhada entre processos de uma aplicação
- geralmente implementado em bibliotecas porque o uso normal é programar com consistência seqüencial forte e isso resulta em desempenho ruim
- programação com consistência relaxada é aceitável em aplicações paralelas mas complexo demais para uso normal

Comunicação no nível de usuário

- Aglomerados de alto desempenho NÃO usam Ethernet mas Myrinet, Infiniband, ou ...
- para uso ser vantajoso, adaptador de rede deve ser acessado com pouco *overhead*
- todas (?) as ações de comunicação devem ocorrer em espaço/modo de usuário, tipicamente com DMA em modo usuário
- problema: multiplexar tráfego de vários processos na mesma interface física de forma segura
- gerenciamento de filas de TX e RX pelo usuário
  SO estabelece conexão e pega/paga interrupções circuito virtual?
  * menos trocas de contexto e menos cópias
  * Remote DMA (RDMA): acesso via DMA à memória de outros nós do aglomerado ⟶ baixa latência+overhead

E/S paralela

- Problemática na carga e salvamento de arquivos de dados
- usualmente, alguns nós concentram E/S ⟶ tornam-se gargalos
- é saudável ter algum espaço local a cada nó