OTIMIZAÇÕES DEPENDENTES DO HARDWARE

TeSP de Aplicações Móveis
André Martins Pereira

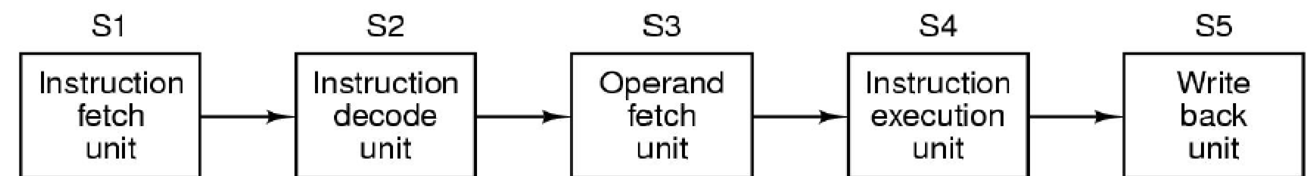
INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

OTIMIZAR DESEMPENHO

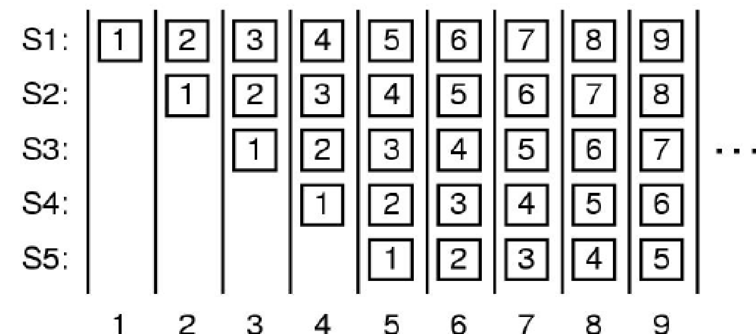
- Como melhorar hardware para otimizar desempenho no *hardware*?
  - Paralelismo:
    - Ao nível do processo (multicore)
    - Ao nível da instrução num *core* (*Instruction Level Parallelism*)
      - Na execução do código (pipeline, superescalaridade)
      - Apenas nos dados (processamento vetorial)
    - Na transferência de informação de/para memória
      - Paralelismo desfasado (*interleaving*)
      - Paralelismo "real" (> largura barramento, mais canais, …)
    - Hierarquia de memória – *Caching*!

PIPELINE

- Emparelhar instruções para que sejam executadas "ao mesmo tempo"
  - Processador tem várias fases de execução
  - Se 2 instruções ***i*** e ***j*** não partilham dados, podem ser executadas em conjunto
  - ***i*** estará sempre na fase de execução seguinte à fase de ***i***
- Objetivo: CPI=1
- Problemas:
  - Dependência de dados
  - Acessos a memória
  - Saltos condicionais; soluções existente:
    - Executar sempre instrução "a seguir"
    - Histórico de saltos
    - Executar 2 percursos alternativos até a tomada de decisão



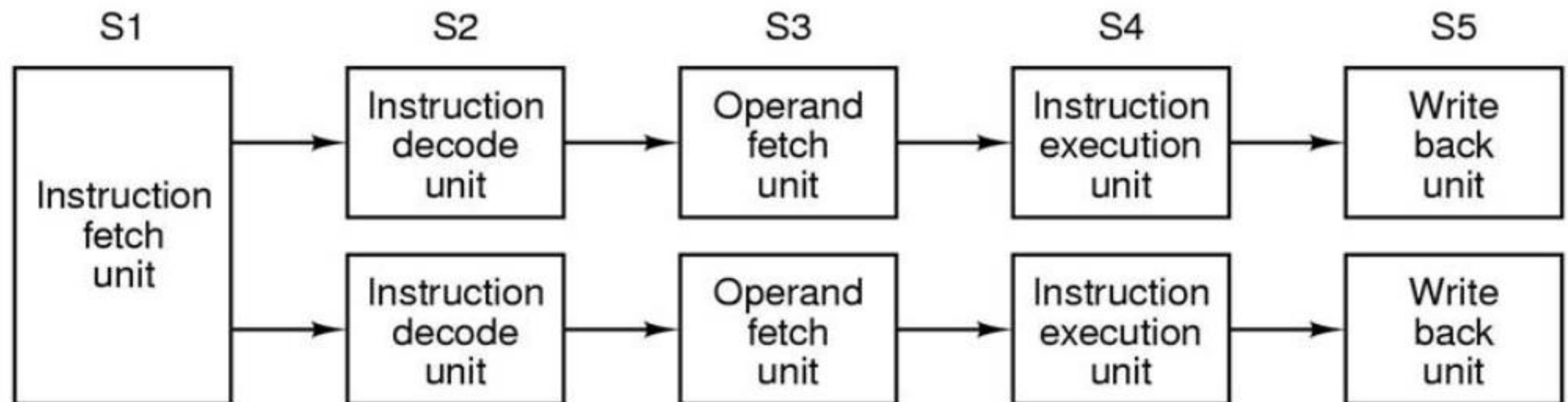
(a)



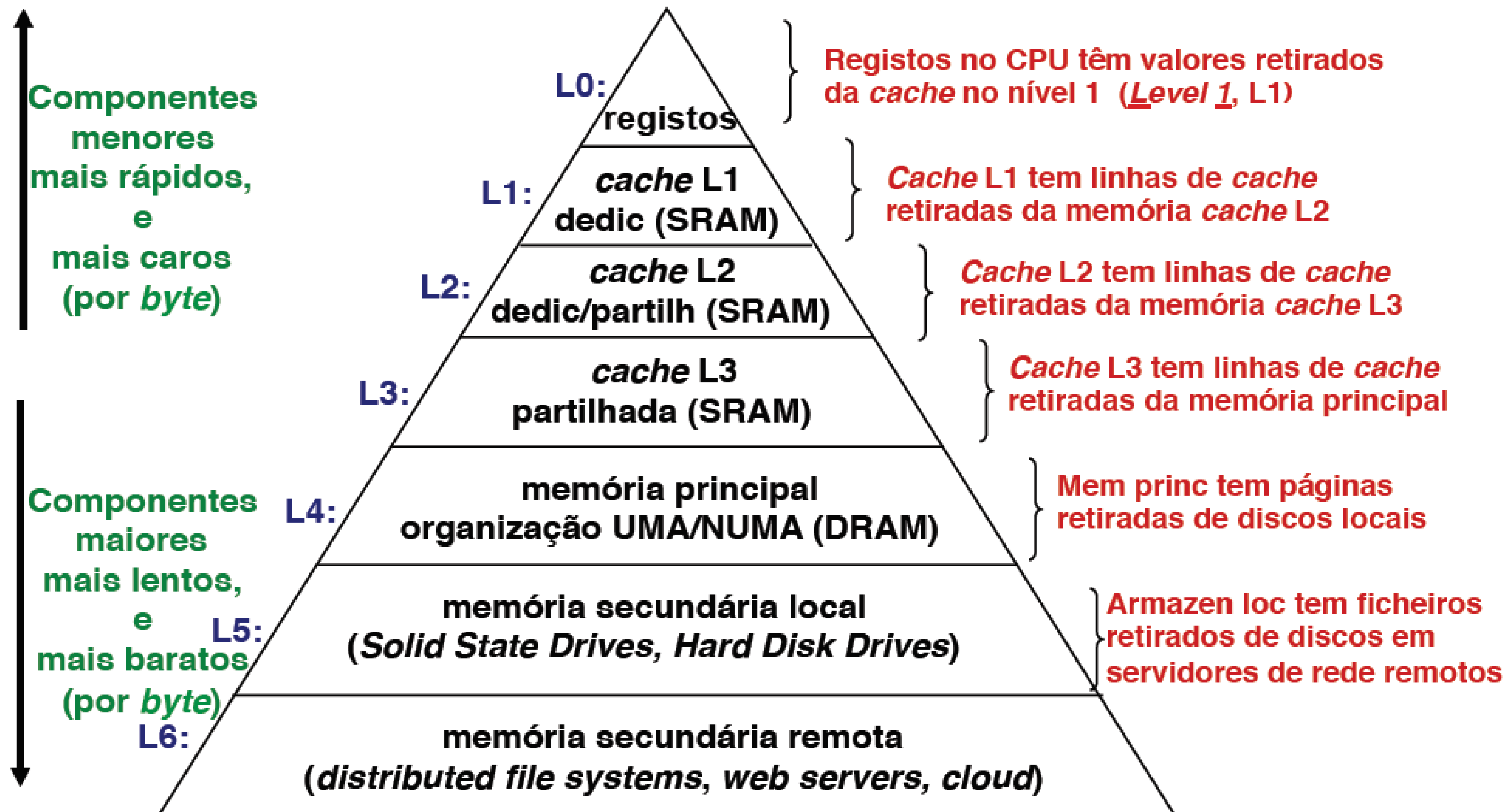
(b)

SUPERESCALARIDADE

- Quando um processador tem 2 ou mais **ALU**'s
  - Exemplo: uma **ALU** para cada tipo de dados numérico (float, int, double, ...)
- Paralelismo mais eficiente do que com *Pipeline*
- E se for necessário executar mais do que 2 instruções sobre *floats* seguidas, ainda que sobre registos diferentes?
  - Solução: Combinar pipeline com superescalaridade!



HIERARQUIA DE MEMÓRIA

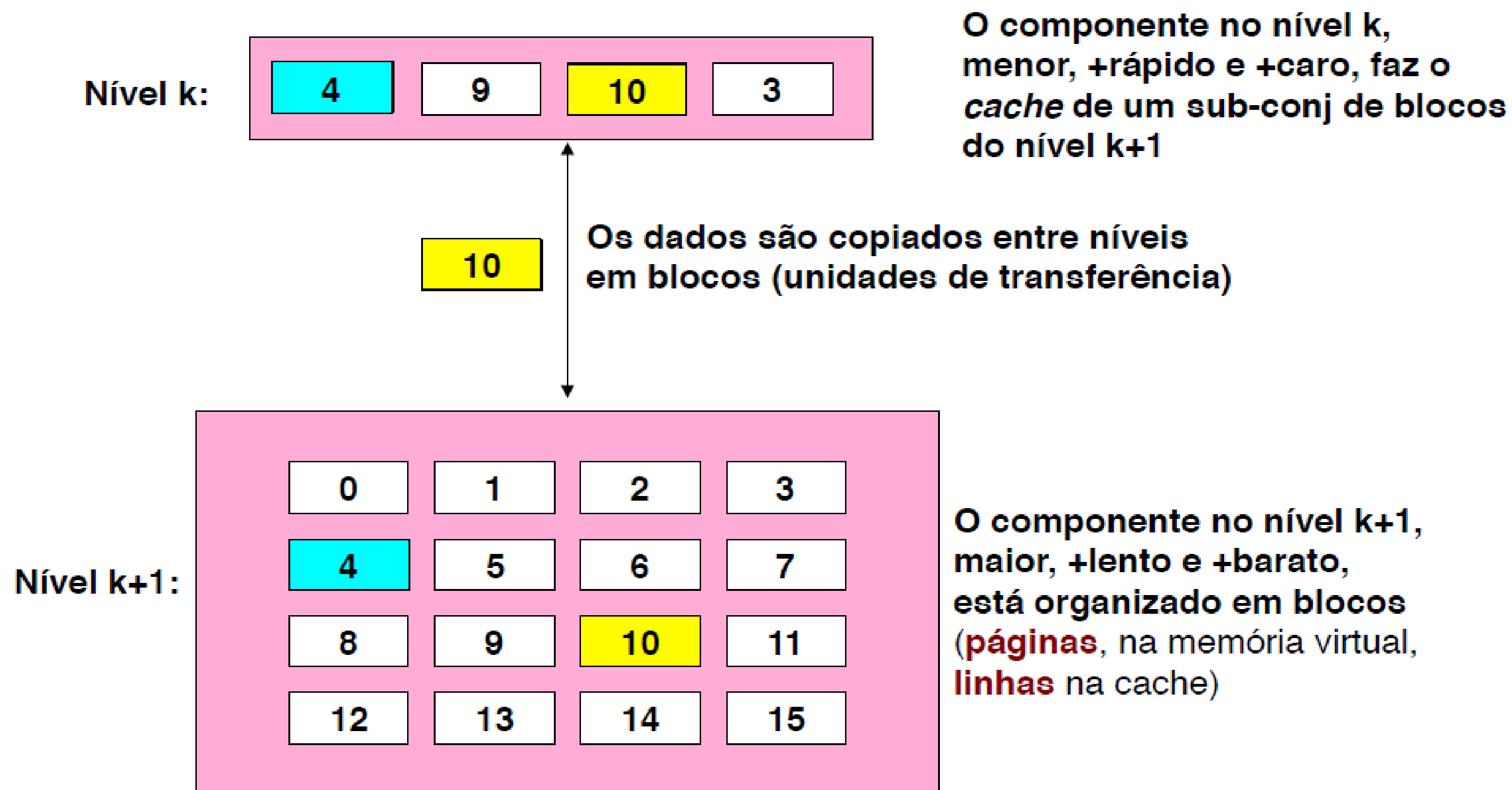


HIERARQUIA DE MEMÓRIA (2)

- Princípio da localidade:
  - Programas tendem a reusar dados e instruções próximas daquelas que foram recentemente usadas
  - Quando um dado/instrução é movido para cache, há 2 tipos de localidade a considerar
    - Localidade Espacial: itens em localizações próximas tendem a ser referenciados em tempos próximos
    - Localidade Temporal: itens recentemente referenciados serão provavelmente referenciados no futuro próximo
- Dados
  - os elementos do *array* são referenciados em instruções sucessivas: Espacial
  - a variável sum é acedida em cada iteração: Temporal
- Instruções!
  - as instruções são acedidas sequencialmente: Espacial
  - o ciclo é repetidamente acedido: Temporal

HIERARQUIA DE MEMÓRIA (3)

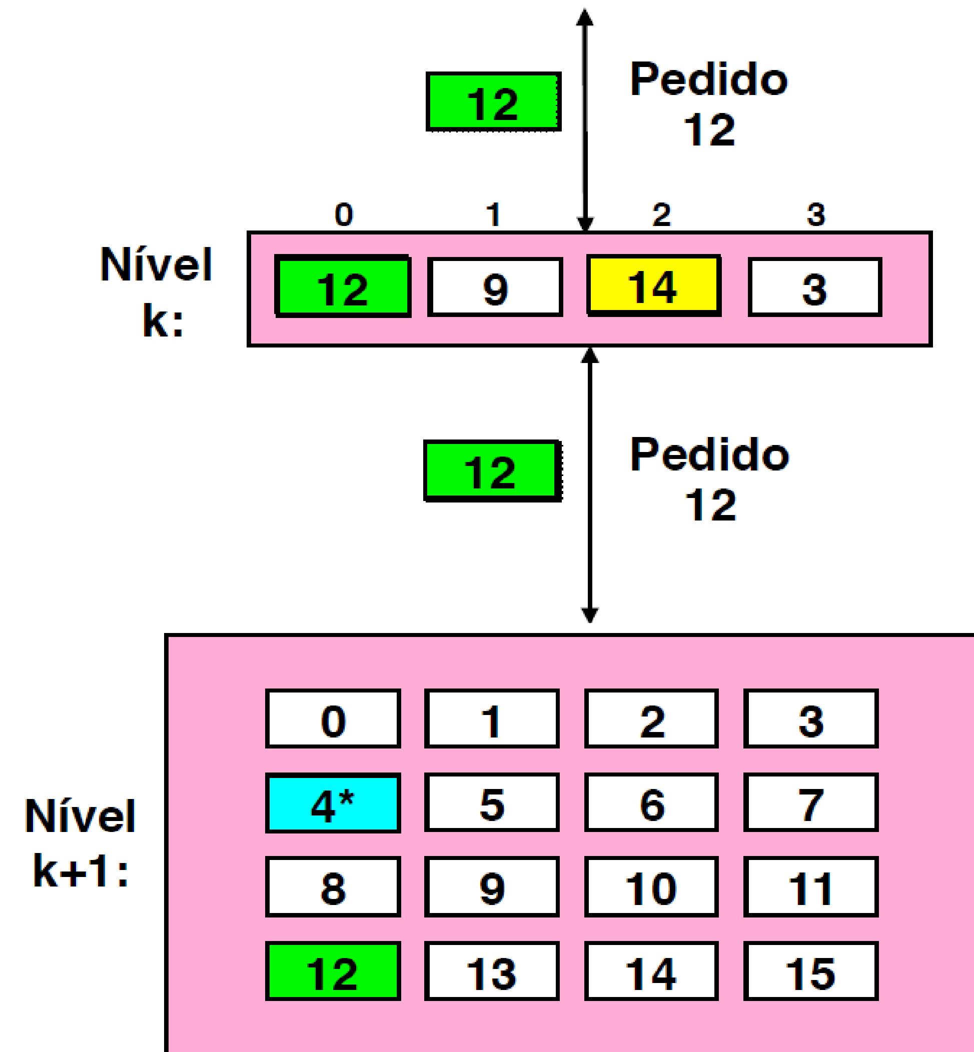
- Este princípio afeta a política de *caching*:
  - O que se vai colocar em *cache*? Quando se vai colocar? Quando se vai remover?



INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

HIERARQUIA DE MEMÓRIA (4)

- Um programa pede pelo objeto **d**, que está
- armazenado num bloco **b**
  - *Cache hit*: o programa encontra **b** na cache no nível k
    - Ex.: bloco 14
  - *Cache miss*: **b** não está no nível k, logo a *cache* de nível k deve procura-lo no nível k+1
    - Ex.: bloco 12
  - Se a *cache* nível k está cheia, então um dos blocos deve ser substituído. Qual?
    - *Replacement policy*: que bloco de ser retirado? (ex.: LRU)
    - *Placement policy*: onde colocar o novo bloco? (`b mod 4`)



HIERARQUIA DE MEMÓRIA (5)

- Miss Rate
  - percentagem de referências à memória que não tiveram
  - sucesso na cache (#misses / #acessos)
  - valores típicos:
    - 3-10% para L1
    - pode ser menor para L2 (< 1%), dependendo do tamanho, etc.
- Hit Time
  - tempo para a cache entregar os dados ao processador
  - (inclui o tempo para verificar se a linha está na cache)
  - valores típicos :
    - 1-2 ciclos de clock para L1
    - 3-10 ciclos de clock para L2
- Miss Penalty
  - tempo extra necessário para ir buscar uma linha após miss
  - tipicamente 50-100 ciclos para aceder à memória principal

HIERARQUIA DE MEMÓRIA (6)

- Regras na codificação de programas
  - Referenciar repetidamente uma variável é positivo! (localidade temporal)
  - Referenciar elementos consecutivos de um array é positivo! (localidade espacial)
  - Exemplos:
    - *cache* fria/vazia, palavras de 4-bytes, blocos (linhas) de *cache* com 4-palavras

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

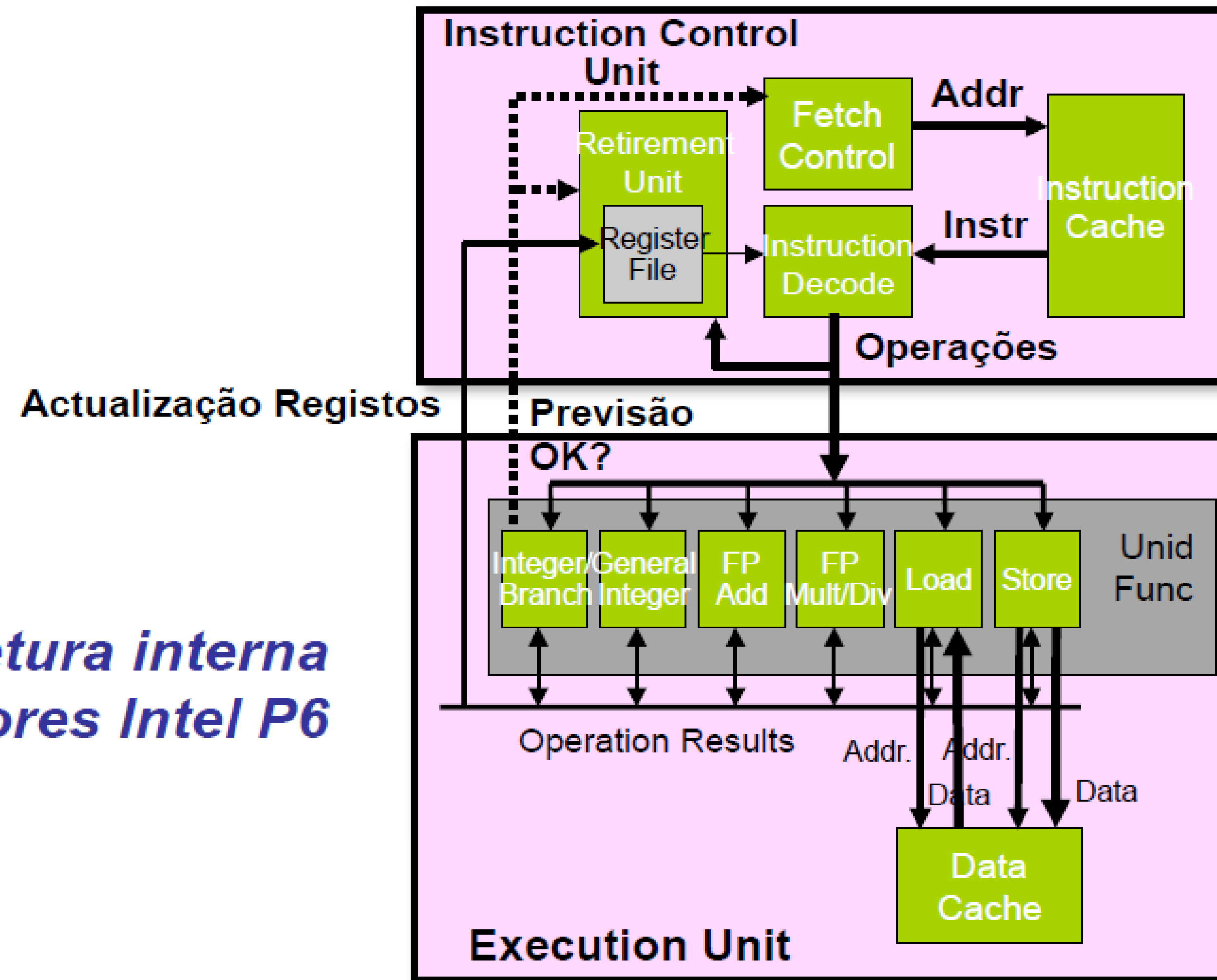
```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

***Miss rate* = 1/4 = 25%**

***Miss rate* = até 100%**

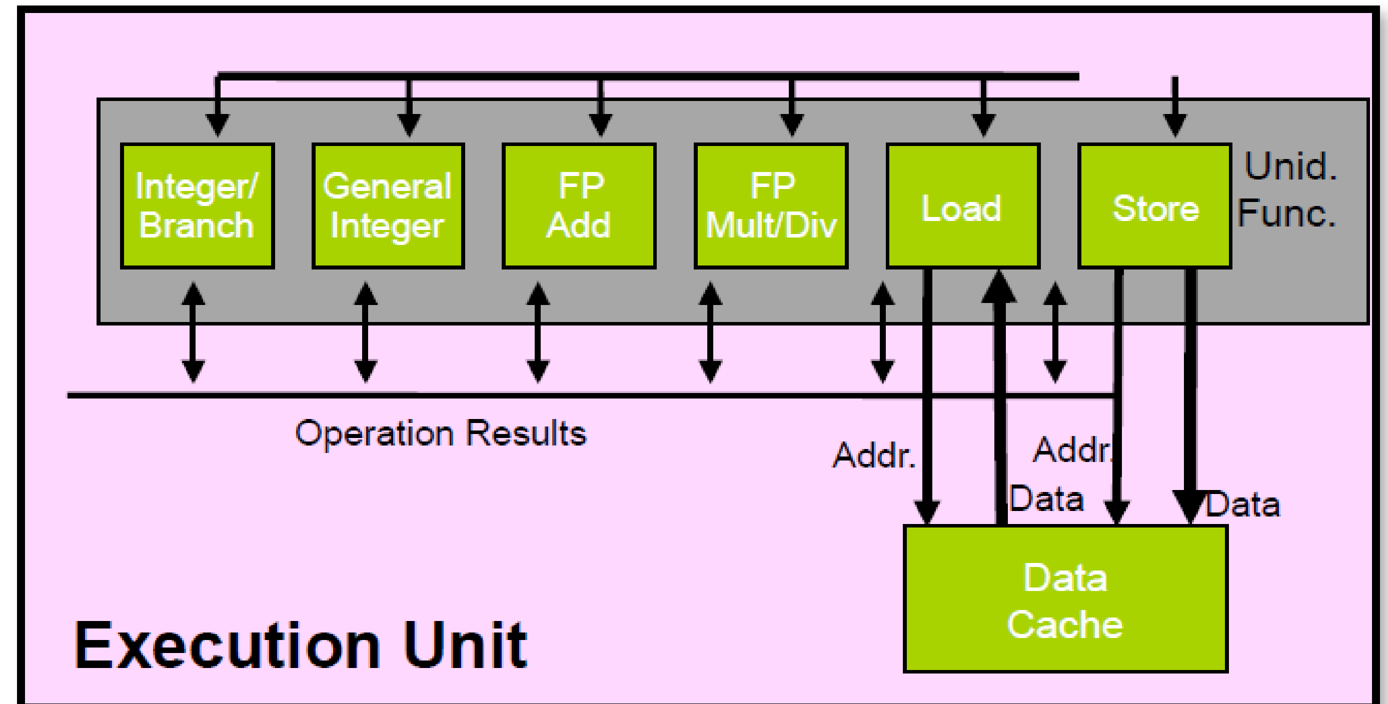
ARQUITETURA DE NOVOS PROCESSADORES



A arquitetura interna dos processadores Intel P6

EXECUÇÃO COM PARALELISMO

- Execução paralela de várias instruções
  - 2 integer
  - 1 FP Add
  - 1 FP Multiply ou Divide
  - 1 load
  - 1 store



EXECUÇÃO COM PARALELISMO (2)

- Algumas instruções requerem mais do que 1 ciclo

Instrução	Latência	Ciclos/Emissão
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Add	3	1
Double/Single FP Multiply	5	2
Double/Single FP Divide	38	38

EXECUÇÃO COM PARALELISMO (3)

```
.L24:                            # Loop:
  imull  (%eax,%edx,4),%ecx      # t *= data[i]
  incl   %edx                    # i++
  cmpl   %esi,%edx               # i:length
  jl     .L24                    # if < goto Loop
```

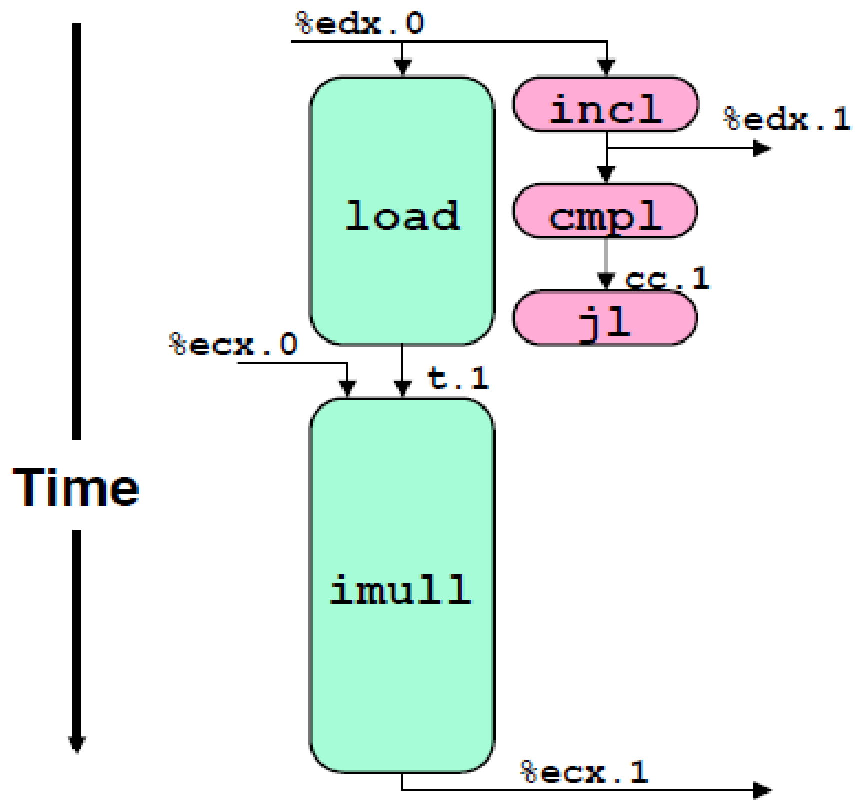
```
.L24:
  imull  (%eax,%edx,4),%ecx

  incl   %edx
  cmpl   %esi,%edx
  jl     .L24
```

```
load (%eax,%edx.0,4)   ➔ t.1
imull t.1, %ecx.0      ➔ %ecx.1
incl %edx.0            ➔ %edx.1
cmpl %esi, %edx.1      ➔ cc.1
jl    -taken cc.1
```

INSTITUTO POLITECNICO DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

EXECUÇÃO COM PARALELISMO (4)

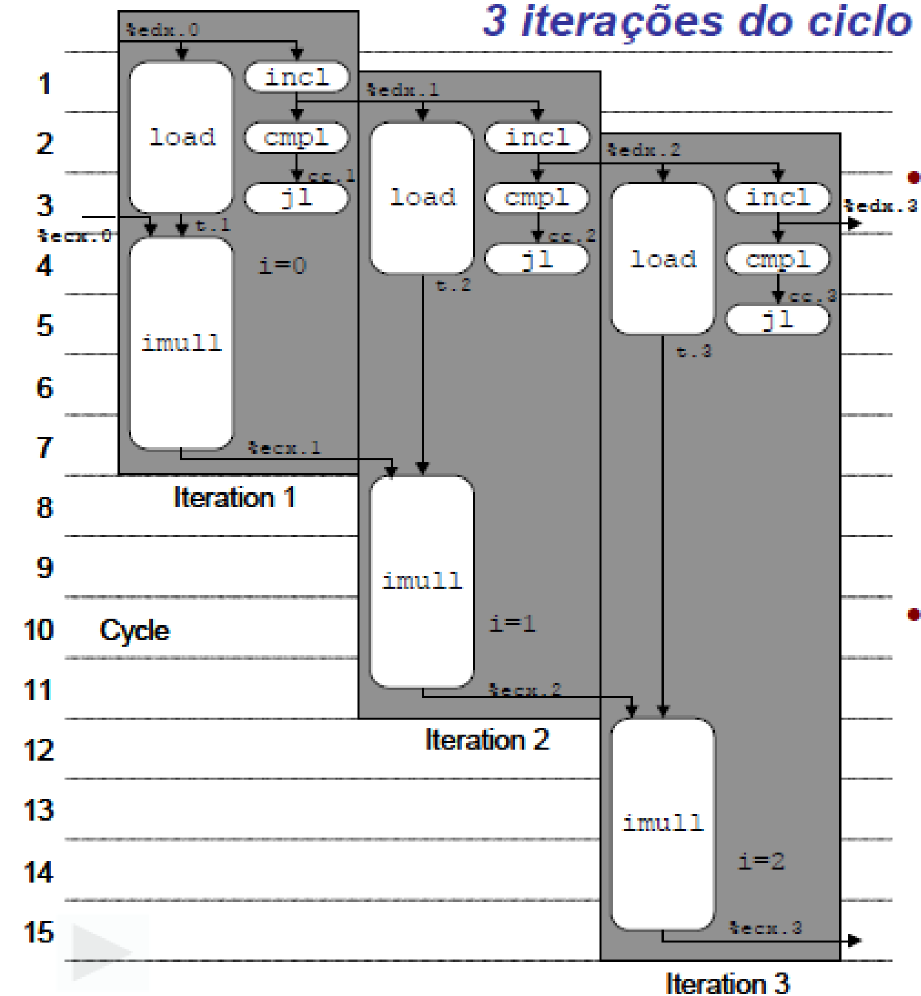


Instrução	Latência	Ciclos/Emissão
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Add	3	1
Double/Single FP Multiply	5	2
Double/Single FP Divide	38	38

- Porque é que o *load* pode executar **exatamente** ao mesmo tempo do *incl*?
- Mas… esta ordem de execução não vai contra o código escrito em *assembly*?
- Haveria outra hipótese para esclaradidade e *pipeline*?

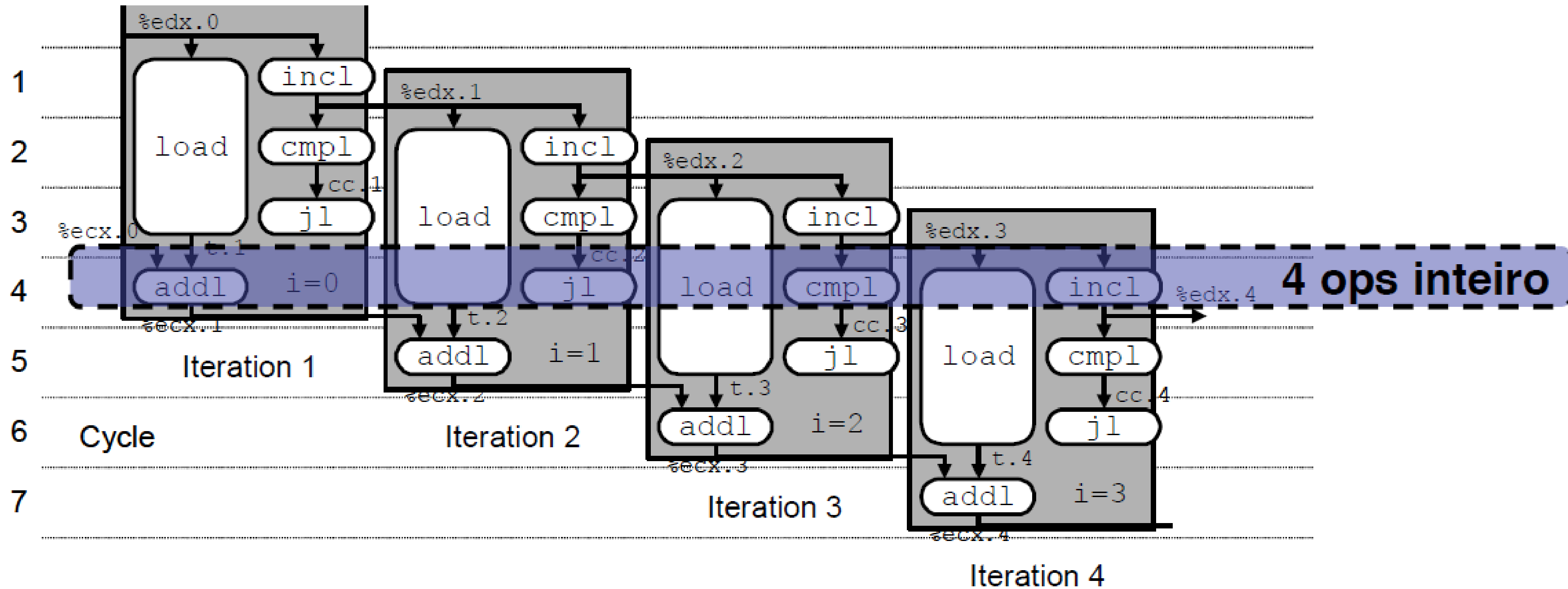
```
load (%eax,%edx.0,4)   ➔ t.1
imull t.1, %ecx.0      ➔ %ecx.1
incl %edx.0            ➔ %edx.1
cmpl %esi, %edx.1      ➔ cc.1
jl    -taken cc.1
```

EXECUÇÃO COM PARALELISMO (5)



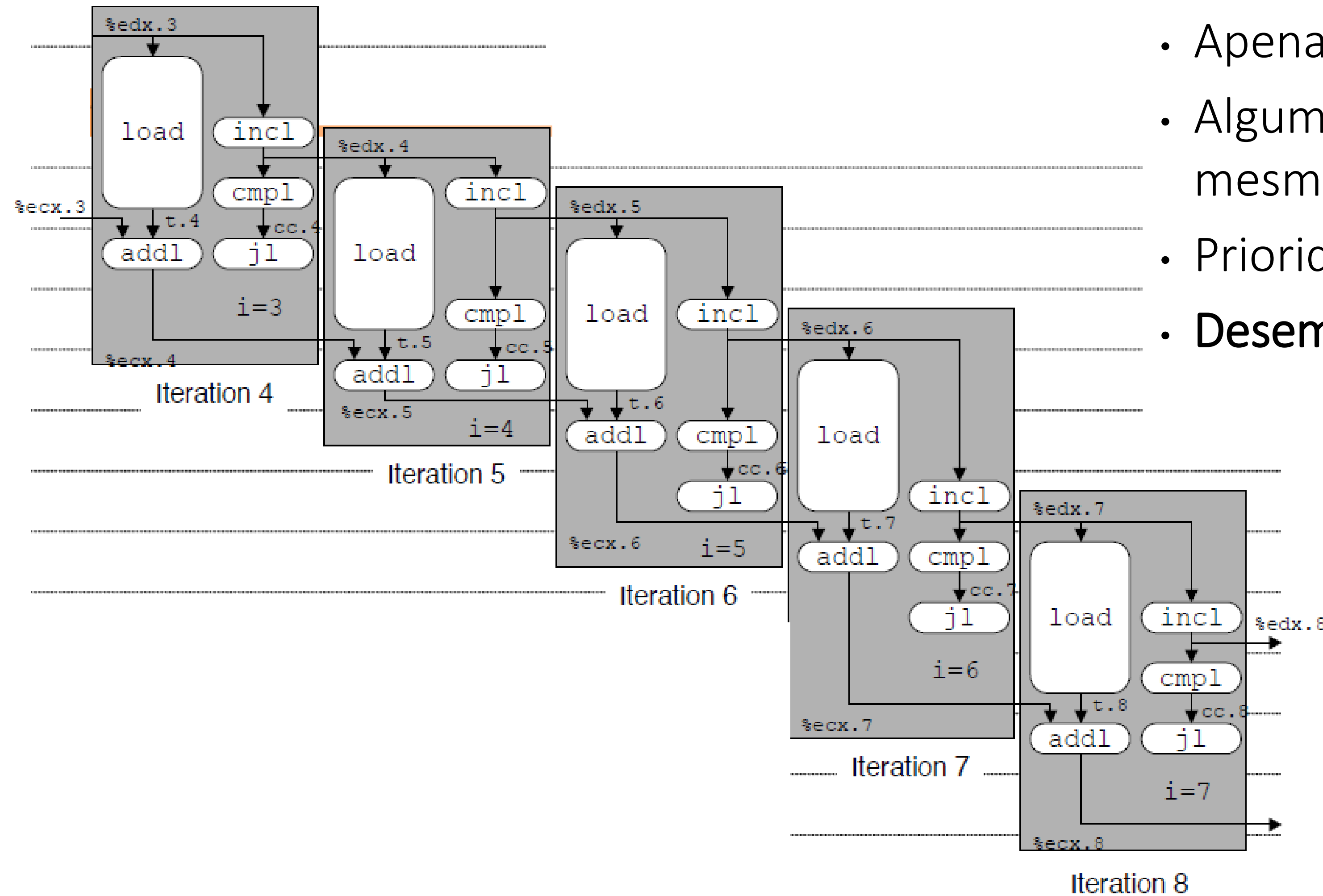
- Análise com recursos <u>**ilimitados**</u>
  - Este *pipelining* implica a existência de **mais do que 2 unidades para operações sobre inteiros**
- **Desempenho**
  - Factor limitativo:
    - latência da multipl. de inteiros
  - CPE: 4.0

EXECUÇÃO COM PARALELISMO (6)



- Com recursos ilimitados:
  - pode começar uma nova iteração em cada ciclo de clock
  - valor teórico de CPE: 1.0
  - requer a execução de 4 operações c/ inteiros em paralelo

EXECUÇÃO COM PARALELISMO (7)



- Apenas 2 unid funcionais de inteiros
- Algumas operações têm de ser atrasadas, mesmo existindo operandos
- Prioridade: ordem de exec do programa
- **Desempenho**: 2 CPE

OTIMIZAÇÕES DEPENDENTES DO HARDWARE