Arquitetura de Memória

Anotações do material suplementar (apresentações PPT) ao Livro do Hennessy e Patterson e do material do Prof. Celso Alberto Saibel Santos (DI/CT).

O que é Arquitetura de Computadores?

Arquitetura de Computadores

=

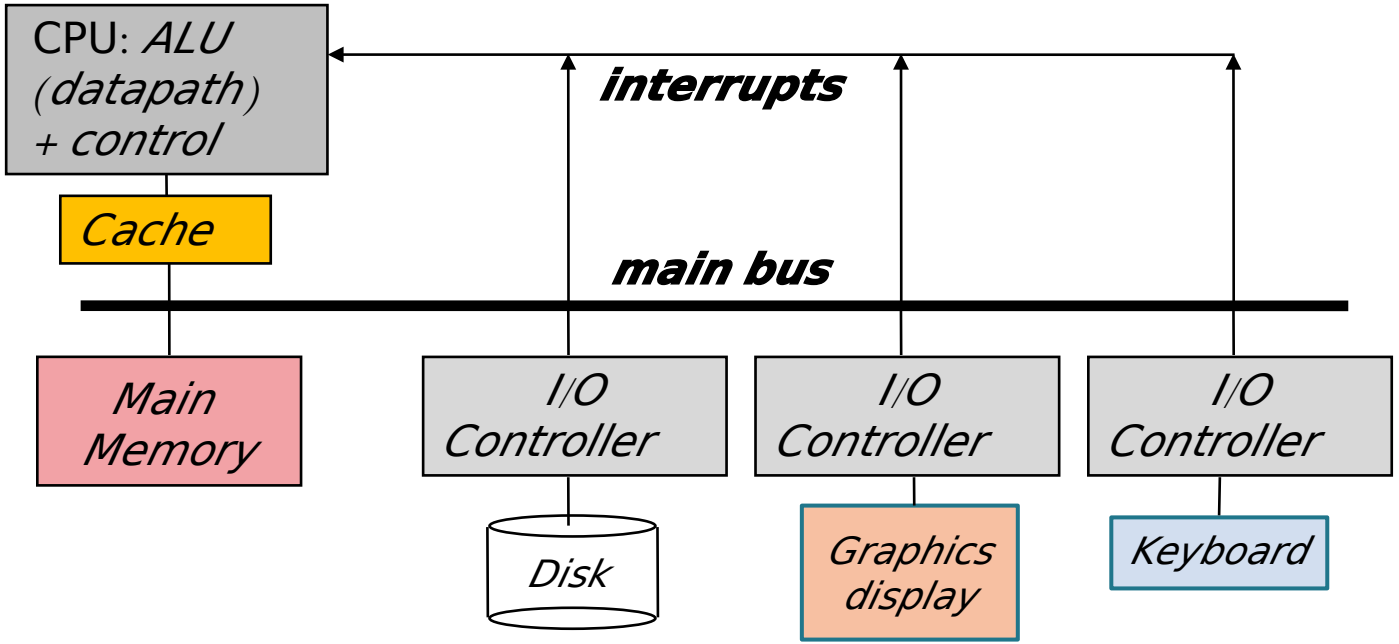
Conjunto de Instruções

(*Instruction Set Architecture*)

+

Organização da Máquina

Componentes básicos (Von Neumann)



Reflexões sobre Arquiteturas de Memórias

1. A “velocidade” de memória aumenta muito menos rapidamente que a dos processadores (*Lei de Moore*);
2. Máquinas mais rápidas: sistema de memória que não pode desperdiçar os ganhos do processador;

Solução: ***Hierarquia de memória* e *caches***

3. Usuário deve “acreditar” que tem um espaço de memória enorme apesar da quantidade de memória ser limitada;
4. O SO precisa suprir o mesmo espaço de memória para vários usuários sem que haja conflito;

Solução: ***Memória Virtual***

Princípio da localidade

- **Localidade Temporal**: Uma palavra de memória que acaba de ser referenciada, tende a ser novamente referenciada
- **Localidade Espacial**: Itens cujos endereços estão próximos daqueles que acabam de ser referenciados tendem a ser referenciados brevemente

Como fazer isso?

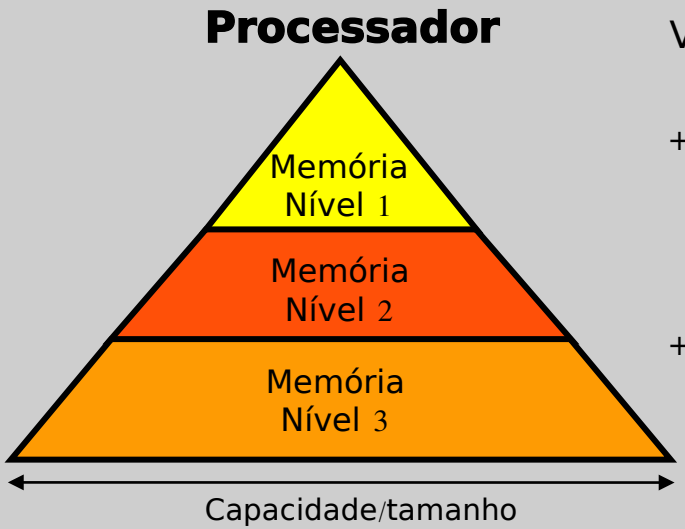
Usar mecanismos que permitam armazenar temporariamente o material com que se trabalha mais frequentemente.

Observar a estrutura dos programas – muitas vezes, eles acessam uma pequena porção de memória em curto espaço de tempo (*ex: variáveis* – temporal; *laços* – temporal e espacial; *percorrer arrays* – espacial)

Hierarquia de Memória

Dividir o espaço de memória em níveis com tamanhos, velocidades e custos diferentes por nível;

Princípio da localidade: mais rápida, mais próxima à CPU.

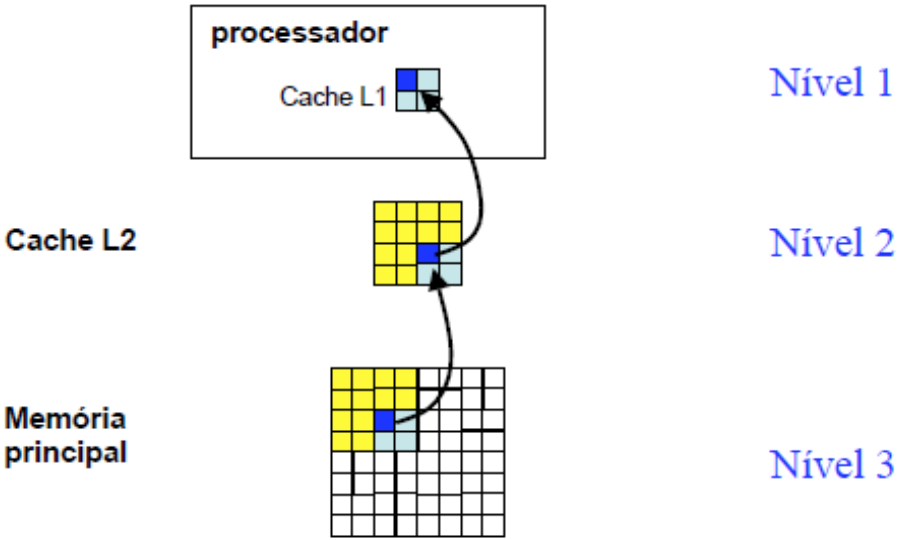


Velocidade	Tam	Custo:	/bit	Exemplo	
+ rápida	menor	+	alto	M	SRAM
				G	DRAM
+ lenta	maior	+	baixo	T	Disk

Funcionamento da hierarquia (1)

Unidades de mapeamento entre níveis: blocos;

Se um bloco está no nível i, estará também no nível i+1 da hierarquia (cópias);



Funcionamento da hierarquia (2)

Gerenciamento é feito entre **apenas entre níveis adjacentes**:

– Se a informação solicitada pela CPU está no nível “mais alto” da hierarquia: “acerto” (*hit*), se não está: “falha” (*miss*);
– O nível inferior é acessado para que o bloco desejado seja recuperado.

Desempenho: *taxa de acertos* dos acessos, i.e. fração dos acessos à memória encontrados no nível “mais alto” (hit).

taxa de falhas = 1 – *taxa de acertos*

Funcionamento da hierarquia (3)

Desempenho ~ tempo de acesso à memória

1. **Tempo de acerto** (t_a): *tempo para acessar o nível superior da hierarquia + tempo necessário para se determinar se o acesso é ou não um acerto*
2. **Penalidade por falta** (t_f): *tempo para substituir um dos blocos do nível superior pelo bloco do nível inferior com a informação desejada + tempo para enviar o dado ao processador*

$t_a << t_f$: tempo de acesso ao nível inferior é muito maior

Hierarquia: influência direta na gerência de memória feita pelo SO, no código gerado pelo compilador e até mesmo no tipo de aplicação a ser executada na máquina.

Conceitos básicos sobre Memória Cache

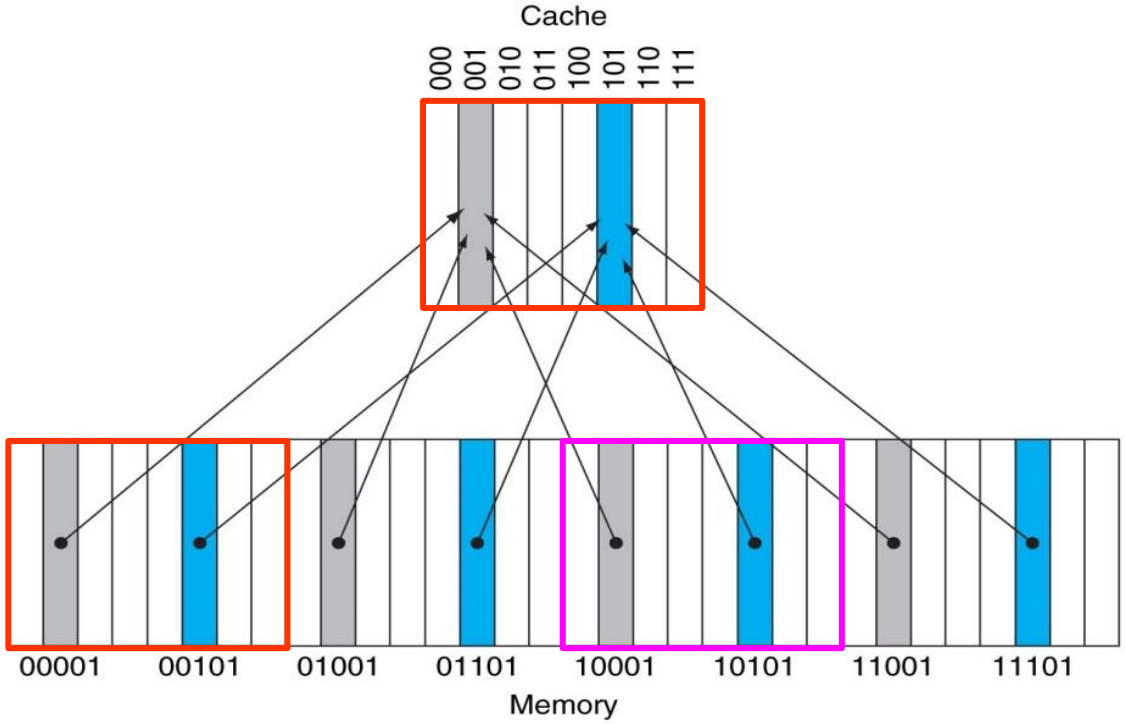
De maneira mais abrangente: cache é qualquer memória gerenciada que tira proveito das propriedades de localidade

- Como fazer para guardar os dados e instruções recentemente usados numa cache?
- Como saber se um item de dado está ou não na *cache*?
- Se estiver, como achá-lo?

*M*apeamento direto: cada endereço de memória é armazenado em uma posição específica da cache:

- Diferentes posições de memória **têm a mesma posição** na cache;
- Usa um mecanismo **modulo *N*** (*N* é potência de 2);
- Precisa de uma ***tag*** indicando o **endereço** atualmente armazenado;
- Precisa de um **bit de validade** indicando se o dado está na cache;

Cache com mapeamento direto de 8 palavras/entradas



Endereço X mapeado para "X mod 8". São usados $\log 2(8) = 3$ bits como cache *index*.

Assim, os endereços 00001_2, 01001_2, 10001_2 e 11001_2 serão todos mapeados para a entrada 001_2 da cache, enquanto que os endereços 00101_2, 01101_2, 10101_2 e 11101_2 serão mapeados para a entrada 101_2 da cache.

Exemplos de acesso à cache

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.6b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (5.6c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (5.6d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (5.6e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (5.6f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

(1) Estado inicial da cache, após inicialização da máquina

índice	val	tag	informação
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

(2) A referência ao endereço $10110\ (22)$ gera uma falta

índice	val	tag	informação
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

(2.1) Tratamento da falta: buscar no nível inferior da hierarquia de memória o bloco de endereço 10110 e copiar para a cache

índice	val	tag	informação
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(3) A referência ao endereço 11010 (26) gera uma falta

índice	val	tag	informação
000	0		
010	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(3.1) Tratamento da falta: buscar no nível inferior da hierarquia de memória o bloco de endereço 11010 e copiar para a cache

índice	val	tag	informação
000	0		
010	1	11	Mem[11010]
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(4) A referência ao endereço 10110 (22) gera um acerto. Dado lido da cache na linha 110

índice	val	tag	informação
000	0		
010	1	11	Mem[11010]
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(5) A referência ao endereço 11010 (26) gera um acerto. Dado lido da cache na linha 010

índice	val	tag	informação
000	0		
010	1	11	Mem[11010]
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(6) A referência ao endereço 10000 (16) gera uma falha.

índice	val	tag	informação
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(6.1) Tratamento da falta: buscar no nível inferior da hierarquia de memória o bloco de endereço 10000 e copiar para a cache

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(7) A referência ao endereço 00011 (3) gera uma falta

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(7.1) Tratamento da falta: buscar no nível inferior da hierarquia de memória o bloco de endereço 00011 e copiar para a cache

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(8) A referência ao endereço $10000\ (16)$ gera um acerto. O dado é lido na cache na linha 000

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(9) A referência ao endereço 10010 (18) gera uma falta porque a “tag” armazenada na posição da cache destinada ao endereço 18 (linha 010) é 11 e não 10.

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

hierarquia de memória o bloco de endereço 10010 e copiar para a cache na linha 010 (apaga o conteúdo anterior)

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	10	Mem[10010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

(10) A referência ao endereço $10000\ (16)$ gera um acerto. O dado é lido na cache na linha 000

índice	val	tag	informação
000	1	10	Mem[10000]
001	0		
010	1	10	Mem[10010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Algumas observações importantes

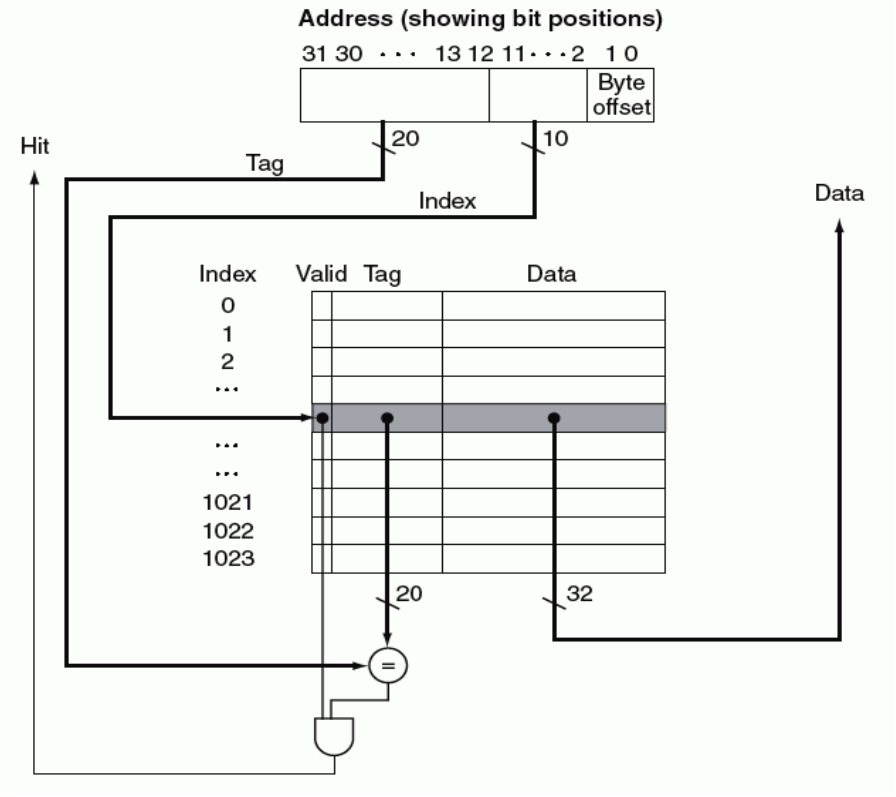
Normalmente, *caches* separadas para **instruções** e **dados**

- O uso de apenas uma cache (dados + instruções) seria mais eficiente em termos de espaço, porém, ao separar, o dobro de vazão pode compensar...

No *start-up*, a cache está vazia (ou toda inválida) e isso gera uma taxa de falhas muito elevada!

Mapeamento direto resolve a **localidade temporal**, mas não a **espacial**!

Exemplo de cache real simples com mapeamento direto



Apenas localidade temporal...

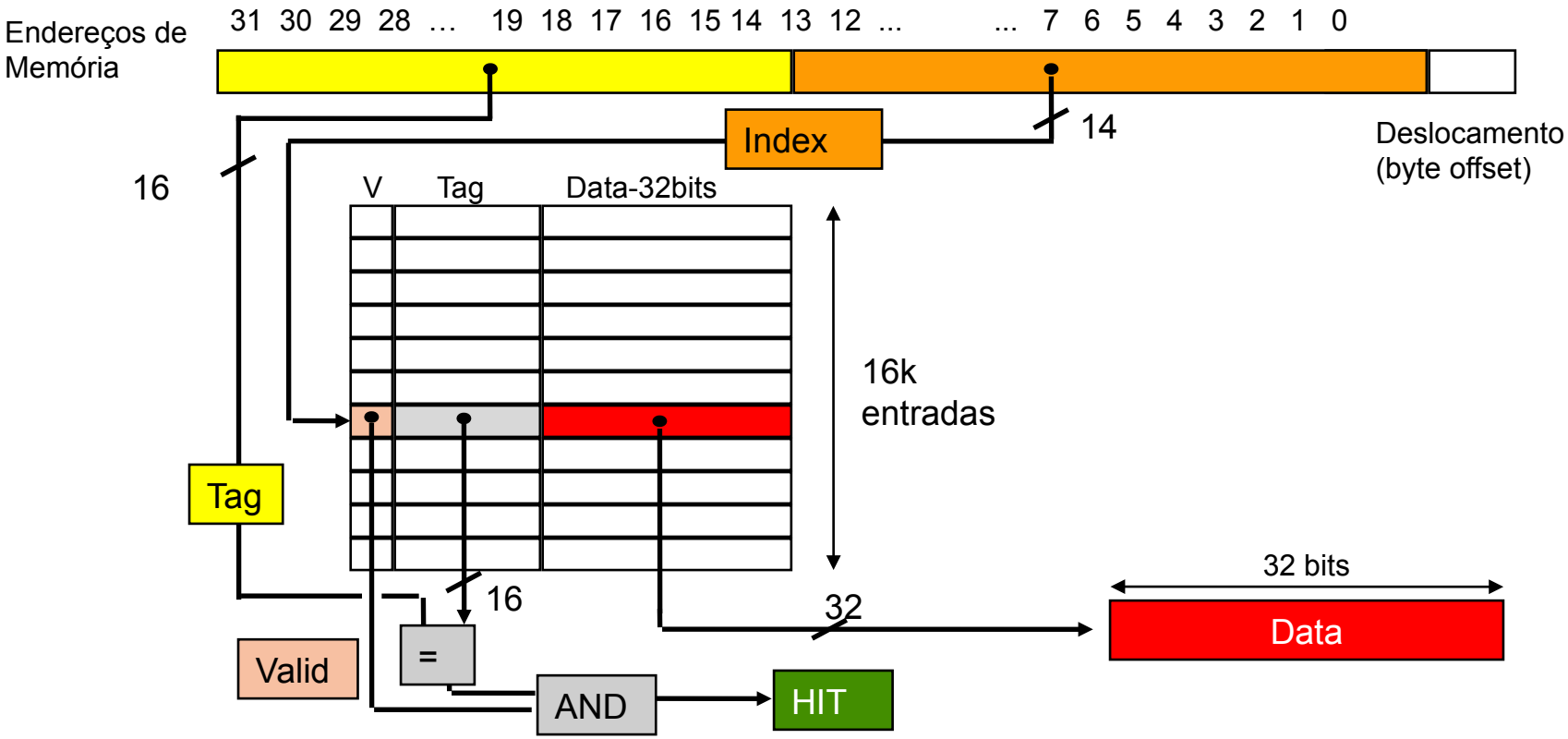
A cache anterior explora apenas a localidade temporal

Para lidar com a localidade espacial, blocos de mais de uma palavra são transferidos para a *cache*

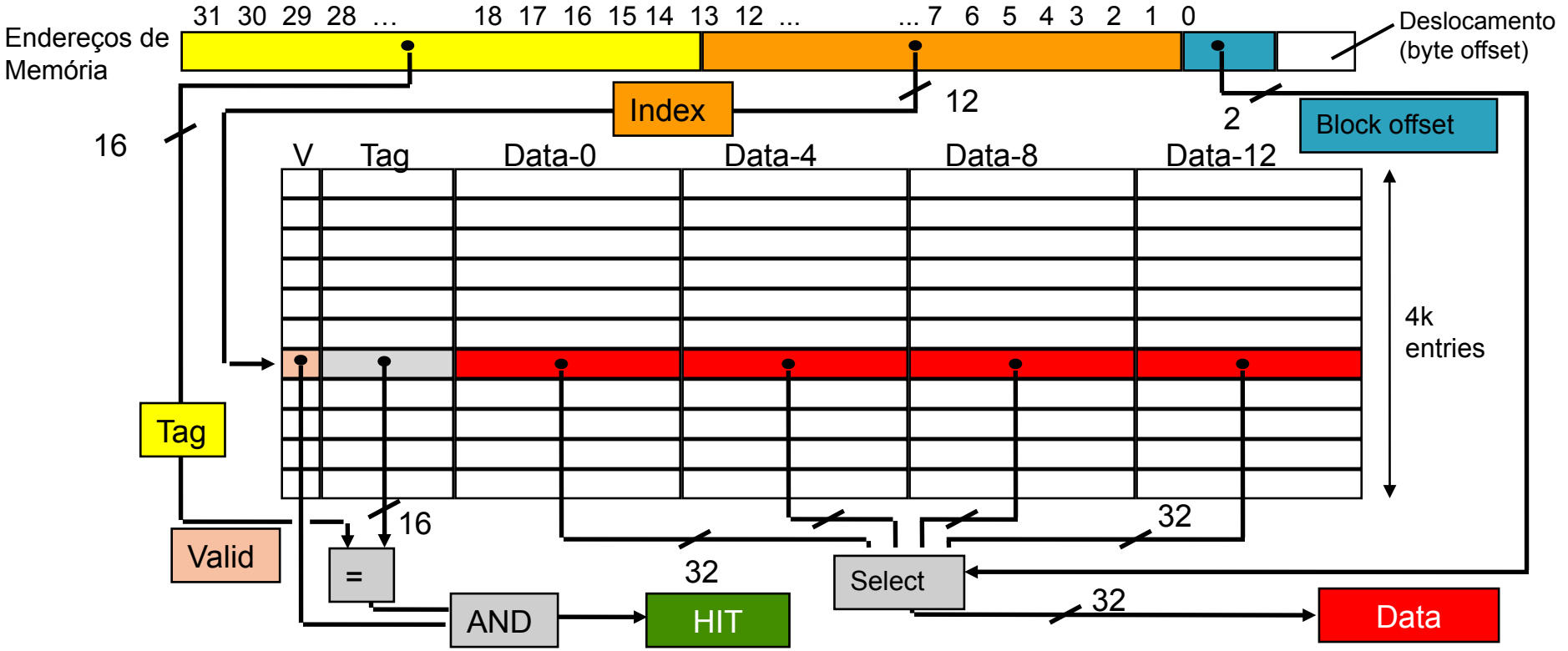
Nos exemplos a seguir, as caches têm a mesma capacidade (64KB), mas a primeira tem 16K entradas para blocos de 4 bytes e a segunda, 4K entradas para blocos de 16 bytes:

- Os acessos em sequência a endereços próximos devem gerar **acertos** ao invés de **faltas**
- Isso pode provocar um aumento significativo na taxa de **acertos**, mesmo com pequenos tamanhos de blocos

Exemplo de cache com 64KB (DECStation 3100)



Cache com 64KB (DECStation 3100)



Tamanho do Bloco de Cache

Tamanho maior, melhora a localidade espacial dos dados, mas:

- Bloco maior = maior penalidade de falhas (*miss penalties*) = mais tempo para encher um bloco
- Bloco muito grande em relação à cache = aumento na taxa de falhas = menos blocos e mais trocas sem que muitas palavras sejam usadas

$$T_{Médio\ de\ Acesso} = t_{acerto} \times (1 - taxa_{falhas}) + penalidade\_falha \times taxa_{falhas}$$