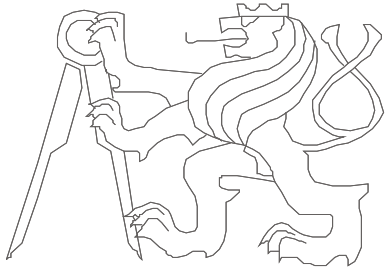
Architektura počítačů

Cache

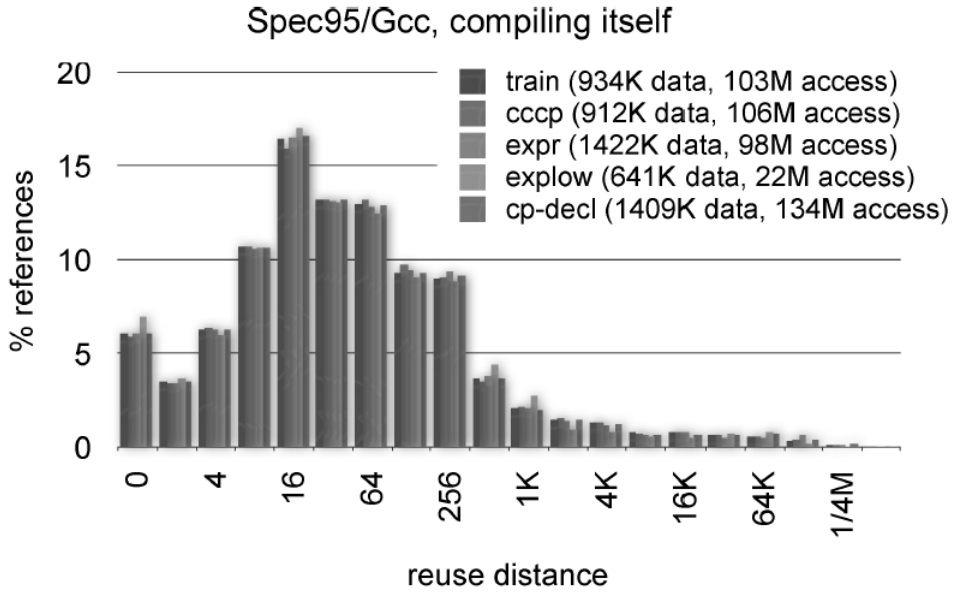
Snímky verze 5.2
– opravené překlepy a chyba na snímcích 77 a 78

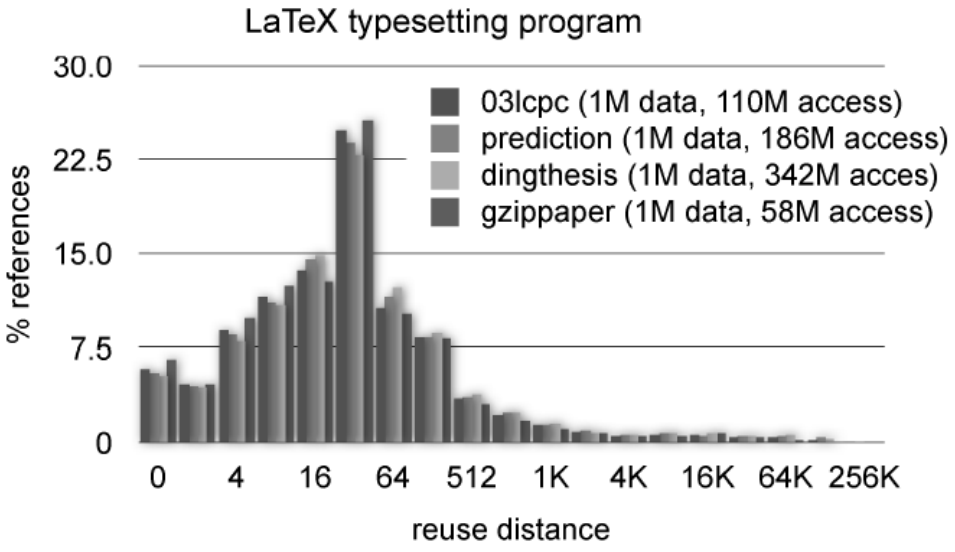
Richard Šusta, Pavel Píša



České vysoké učení technické, Fakulta elektrotechnická

Opakovaně použité proměnné ve vybraných programech





Grafy ukazují vzdálenost v bytech mezi naposledy použitou a opakovaně použitou proměnnou v programu

Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. **Program locality analysis using reuse distance.** *ACM Trans. Program. Lang. Syst. 31, 6, Article 20 (August 2009), 39 pg., available at:* https://dl.acm.org/doi/10.1145/1552309.1552310

Paměťová hierarchie – základní principy

- Programy/vlákna přistupují v daném okamžiku **jen k malé části** svého adresového prostoru.
- **Časová lokalita**
  - Položky, ke kterým se přistupovalo nedávno, budou zapotřebí brzy znovu.
  - Příklad: programová smyčka, proměnné instrukcí.
- **Prostorová lokalita**
  - Položky poblíž právě používaným budou nejspíš brzy také zapotřebí.
  - Příklad: lokální proměnné, sekvenční přístup ke kódu (paměť programu), datová pole (paměť dat).

ZNAČKA SAMOSTUDIA

Značka 3S signalizuje:

"Skrytý Snímek pro Samostudium",

který se na přednášce zpravidla nepromítal. Buď shrnuje výklad nebo ho rozšiřuje o další podrobnosti.

Co z uvedeného plyne?

- Je výhodné uspořádat paměťový prostor hierarchicky – paměťová hierarchie.
- Položky nedávno používané a blízké se mohou uložit do menší, ale rychlejší paměti.

*Urychlení paměti ?

Cache - skrytá paměť

Cache

Trapper's cache,
Mackenzie County,
Alberta
by Provincial Archives
of Alberta

Pronunciation:*kash, (cz: keš); **Etymology:**

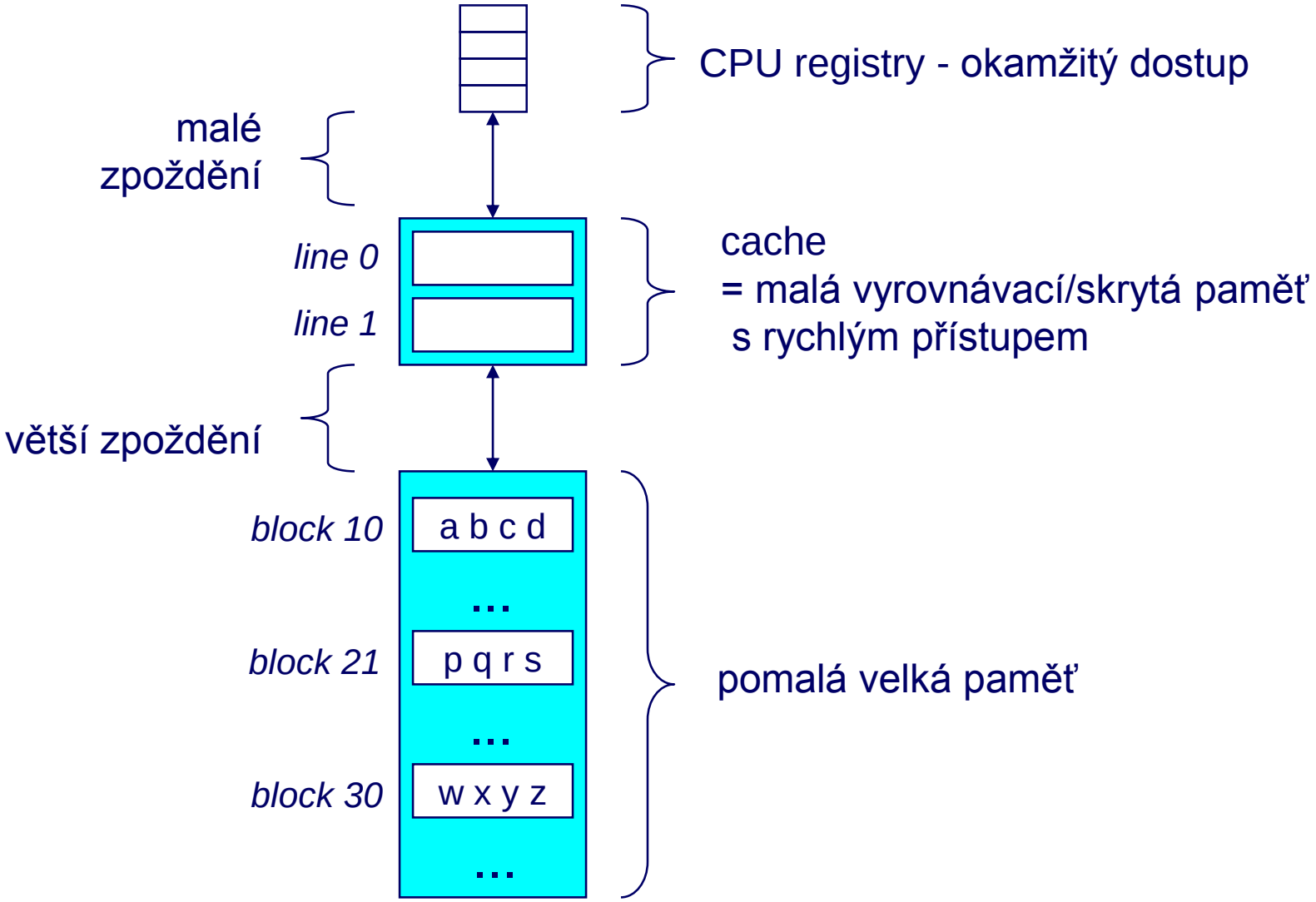
from French Canadian trappers' slang word derived from French *cacher*
= to hide, conceal = hiding place; especially used by settlers, explorers, or campers for concealing and preserving provisions or implements.

[*https://www.merriam-webster.com/dictionary/*]

Cache, výslovnost "keš"

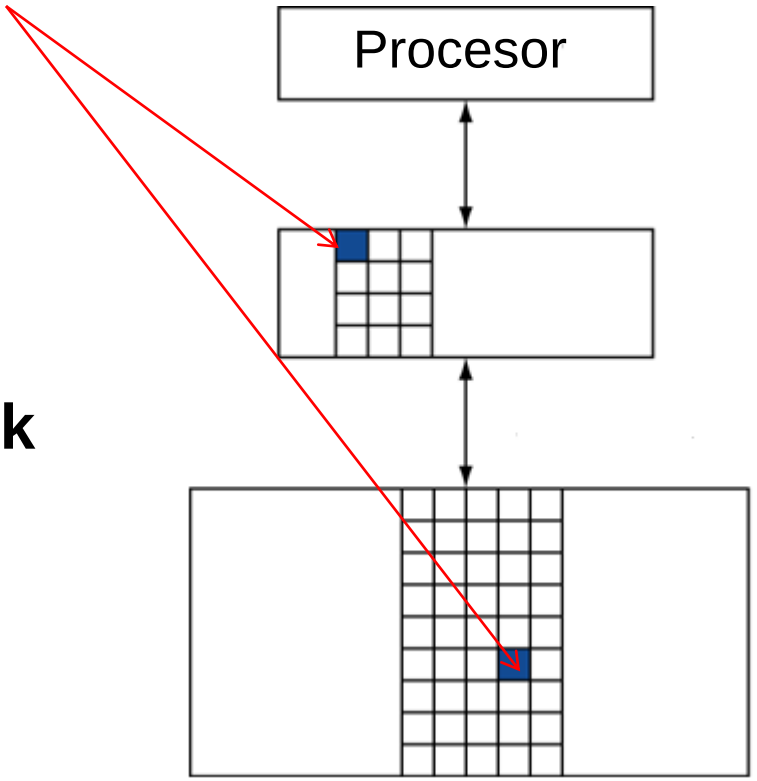
- CZ: **skrytá paměť** - vyrovnávací paměť s rychlým dostupem používaná ve výpočetní technice.
  - Řadí se mezi dva subsystémy s různou rychlostí, v nichž vyrovnává odlišné doby přístupu k informacím.
  - **Urychlí** přístup k opakovaně používaným datům, a to uchováváním jejich kopií, případně zápis dat jeho zpožděním.
  - V přednášce instalujeme **update češtiny,** který do ní přidá nové české slovo **„cache“** ☺
- **Eng: a computer memory** with very short access time used for storage of frequently or recently used instructions or data— called also *cache memory.*

Skrytá paměť - cache



Terminologie kolem skryté paměti

- **Cache hit** pojmenování situace, kdy se požadovaná hodnota načetla ze skryté paměti (cache).
- **Cache miss**, opak cache hit. Muselo číst z hlavní paměti.
- **Cache line** nebo **Cache block** – základní kopírovatelná jednotka mezi hierarchickými úrovněmi.
- V praxi se velikost | bloku (řádky) cache pohybuje od 8B do 1KB, typicky 64B.



Terminologie kolem skryté paměti II.

- **Hit Rate** - podíl počtu paměťových přístupů, které byly úspěšné (nalezla své údaje) při přístupu do cache.
- **Miss Rate** – podobně u neúspěšného přístupu.
- **Miss Penalty** – čas potřebný k načtení bloku (údajů) z paměti nižší hierarchické úrovně.
  MissPenalty – může být vypočtena rekurentně.
- **Average Memory Access Time (AMAT)**
  AMAT = HitTime + MissRate × MissPenalty

Příklad

- Mějme cache o velikosti 8-mi bloků. Kam se do ní umístí data z adresy 0xF0000014?
- Závisí na organizaci cache, který může být:
  - Plně asociativní, **Fully Associative Cache**
  - Přímo mapovaná, **Direct Mapped Cache**
  - S omezeným stupněm asociativity,
    Set-Associative Cache
    pro N=2 je 2-cestná cache,
    2-Way Set-Associative Cache

Obecná organizace cache

Cache je pole
S tříd (sets)

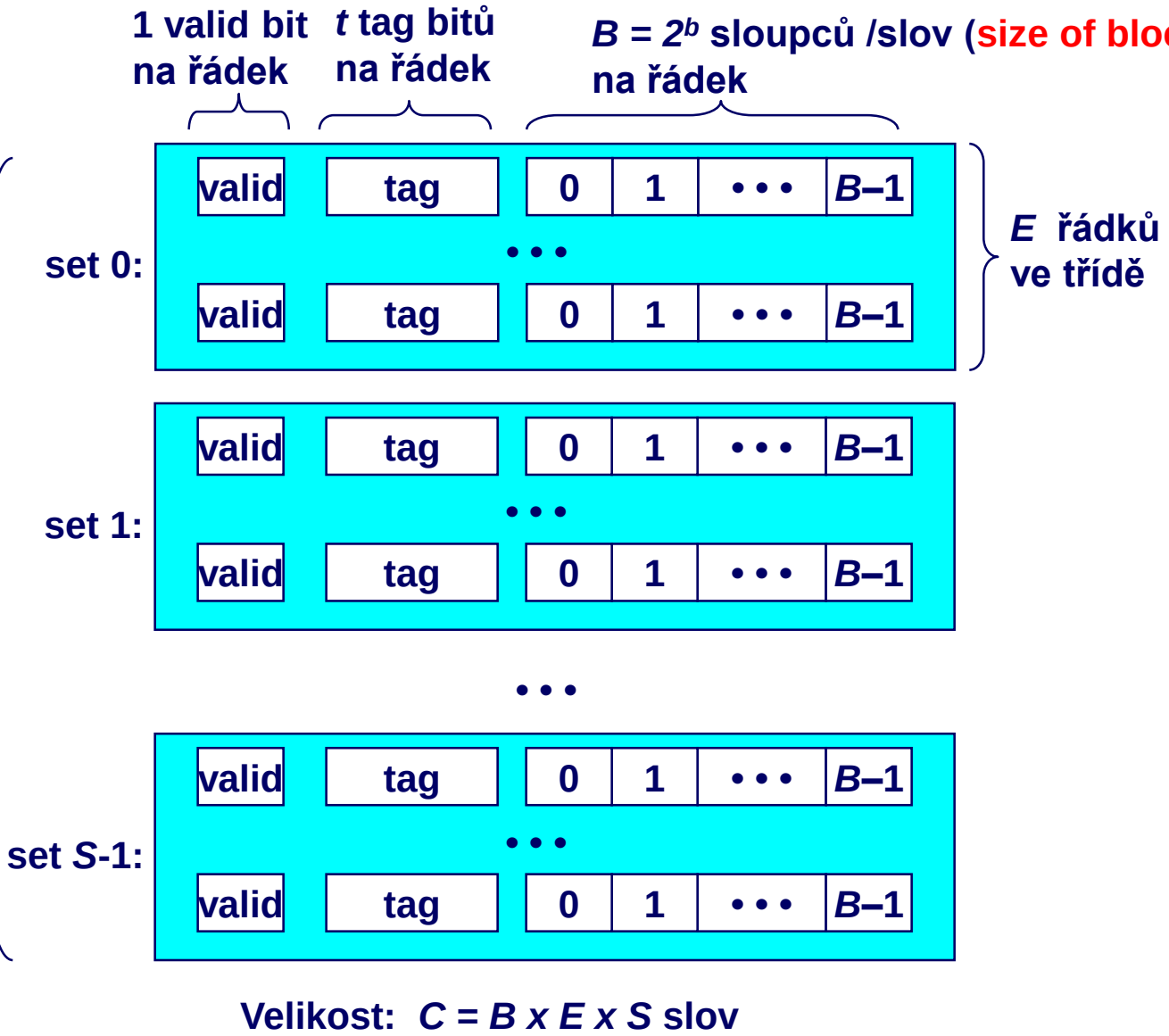
Třída obsahuje
E řádků (blocks)

Každý řádek
má B slov/sloupců

$S = 2^n$,
kde n>=0

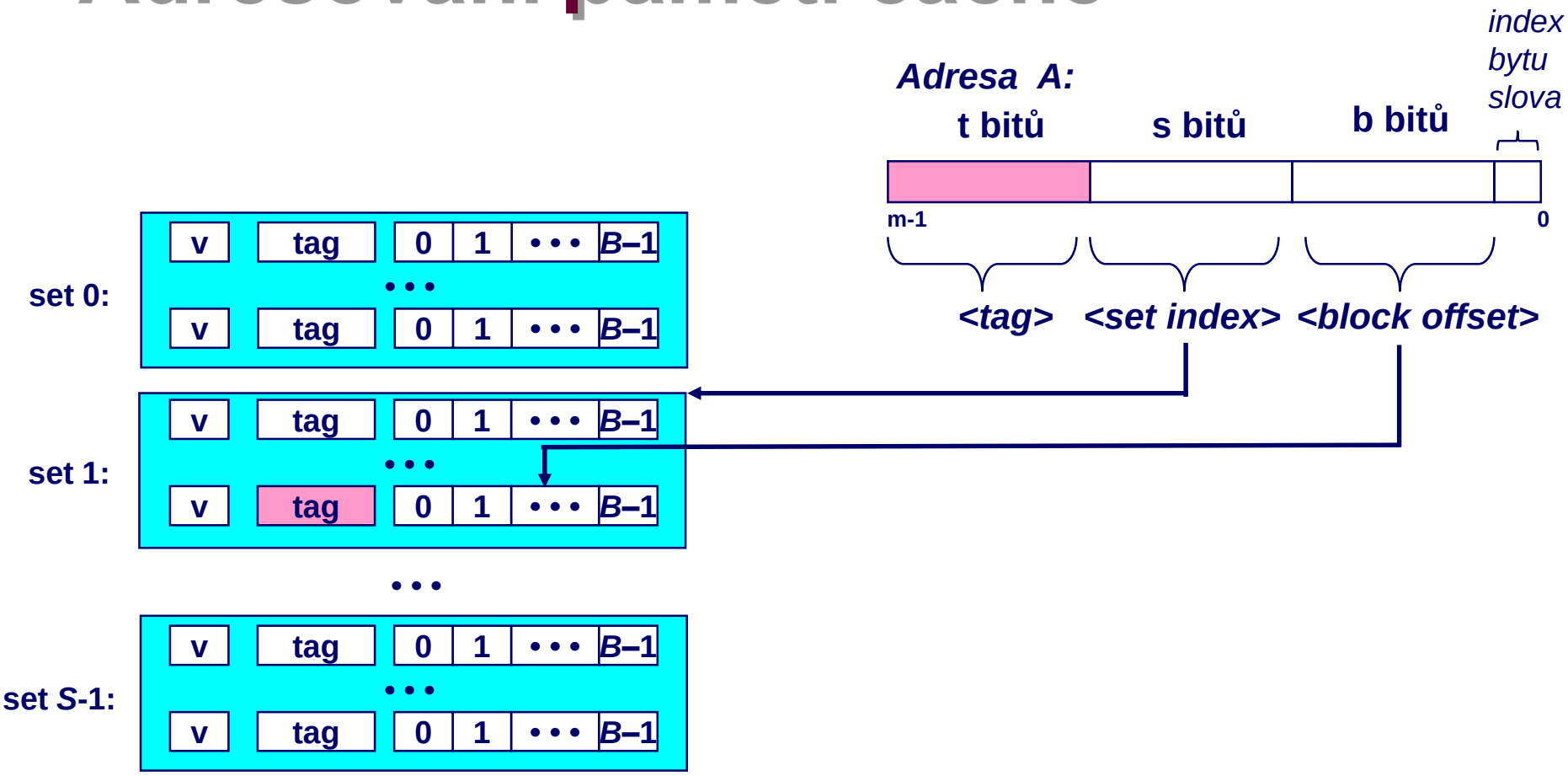
Set no. ≡ hash kód
Tag ≡ hash klíč

Slovo (sloupec)
má zpravidla
32-bitů či 64 bitů



Velikost: $C = B \times E \times S$ slov

Adresování paměti cache



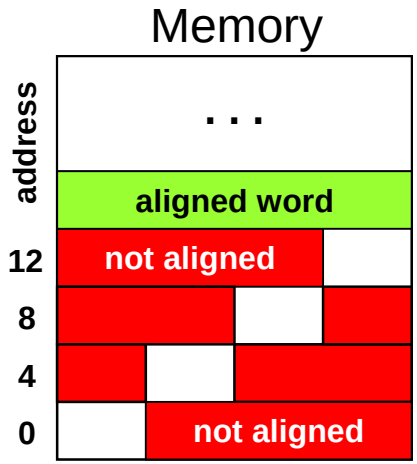
Poznámka

Paměť je organizovaná po bytech, ale běžně používané paměti se konfigurují hlavně na rychlý zápis/čtení celých slov. Jediné výjimky zpravidla tvoří přístupy na periférie, jejichž hardware může podporovat i nastavení zápisu/čtení konkrétního bytu slova (byte enable mask) .

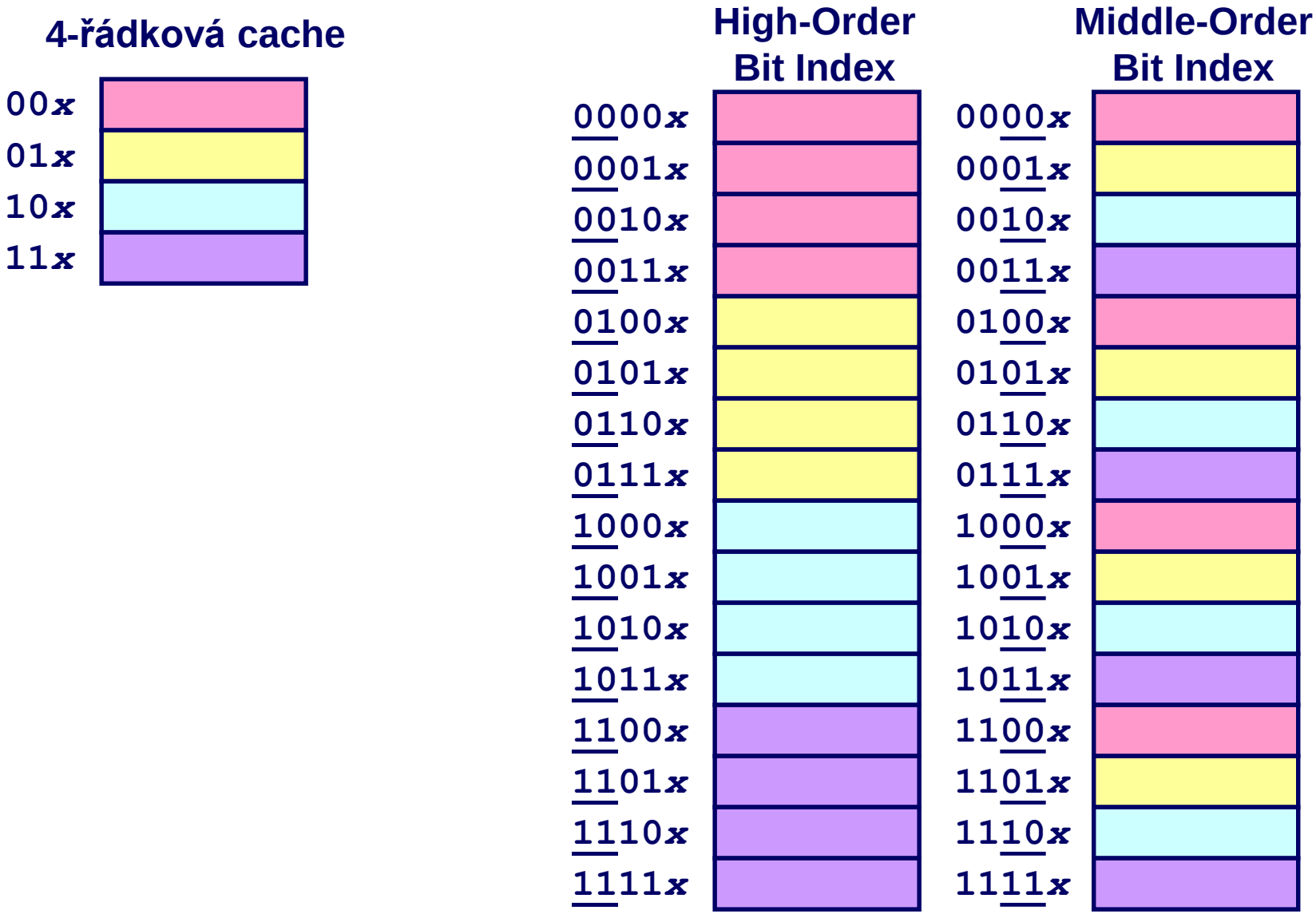
Máme-li 32-bitový procesor, pak jeho slovo má délku 4 byty. Adresy, které vydává procesor při čtení a zápisu do paměti, končí 00. U 64-bitového procesoru pak 000.

Pozn. Některé 64-bitové procesory se sice umí přepnout na 32-bitový mód, ale pořád vnitřně ukládají 64-bitová slova do cache.

Srovnej s „memory alignment“ na snímcích 31 a 32 minulé přednášky.

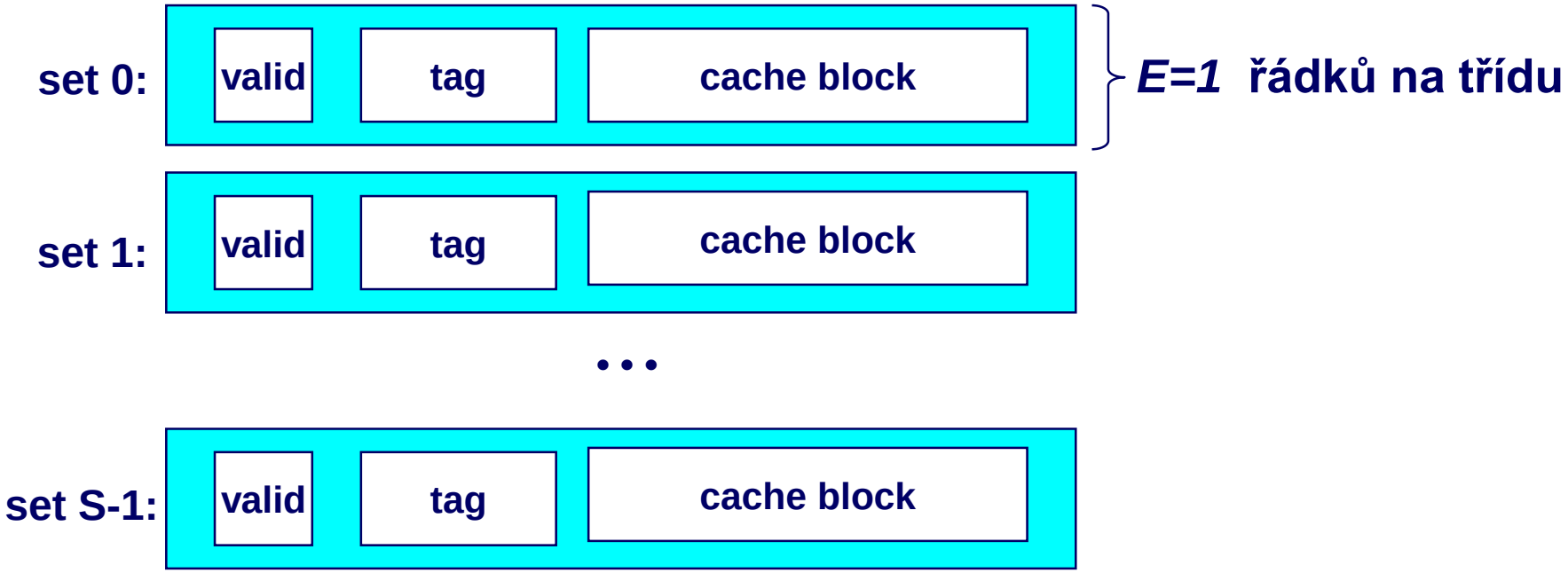


Proč se používají prostřední bity?



Direct-Mapped Cache
cz: Přímo mapovaná cache

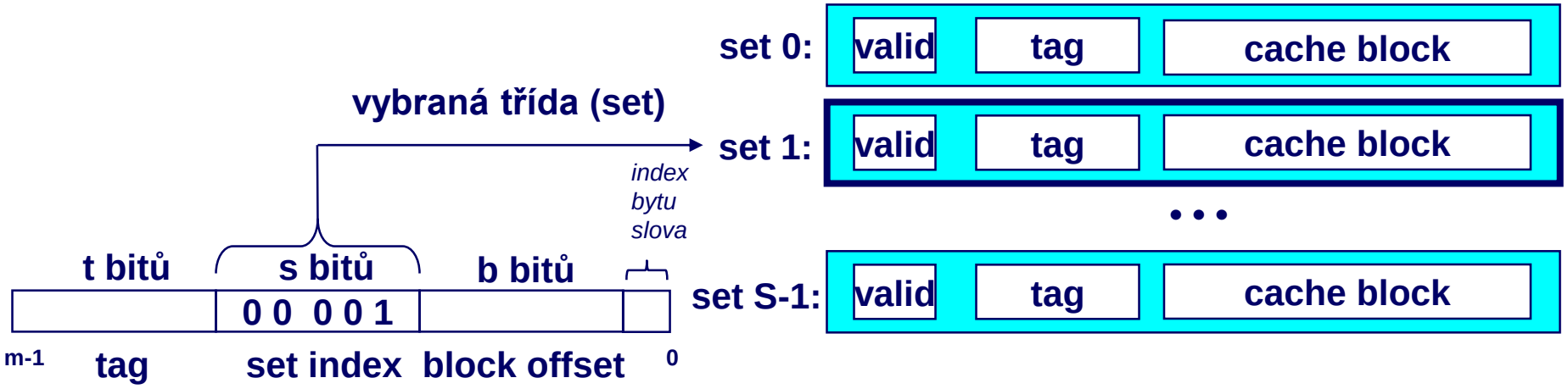
1 řádek (block) na 1 třídu (set)



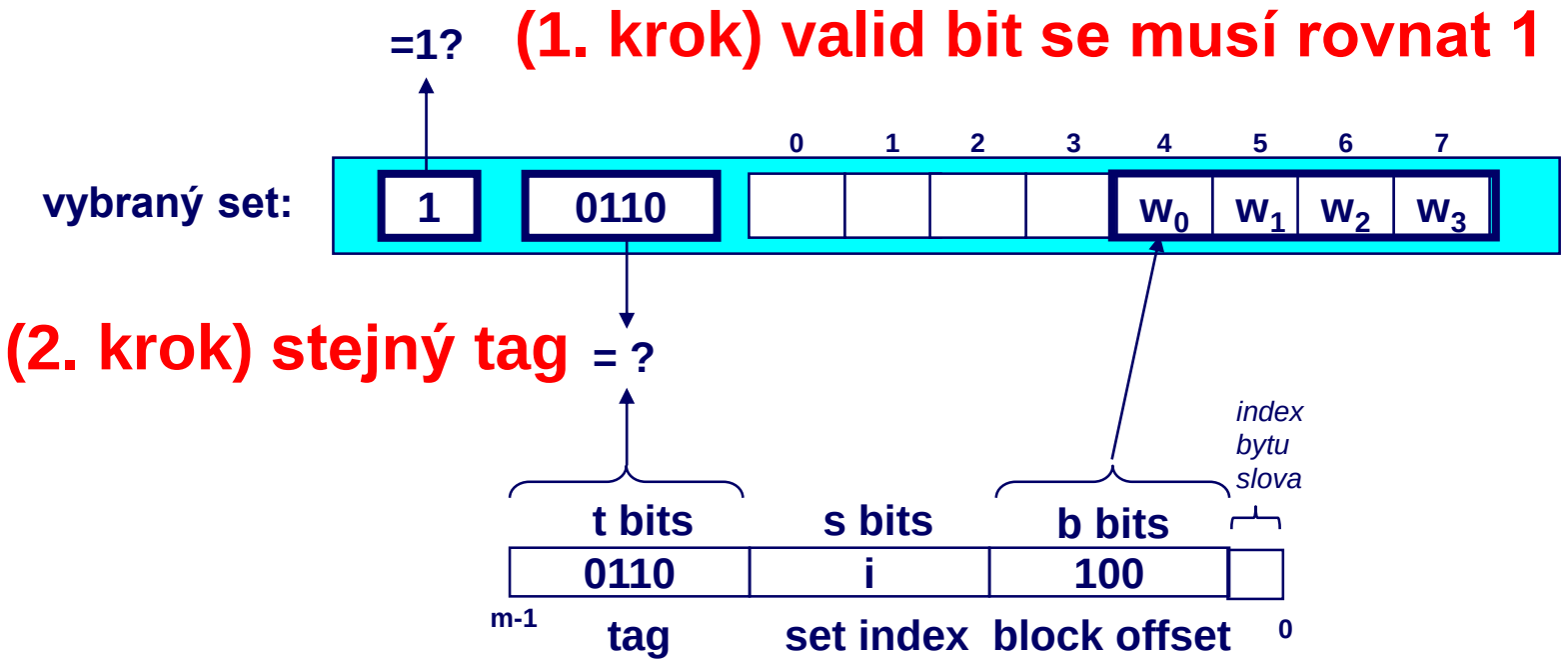
Accessing Direct-Mapped Caches

Výběr setu

- index přímo určuje řádku, 1 set má pouze 1 řádku!



Accessing Direct-Mapped Caches



(3. krok) Jsou-li splněné podmínky (1) a (2),

potom nastane „cache *hit*“

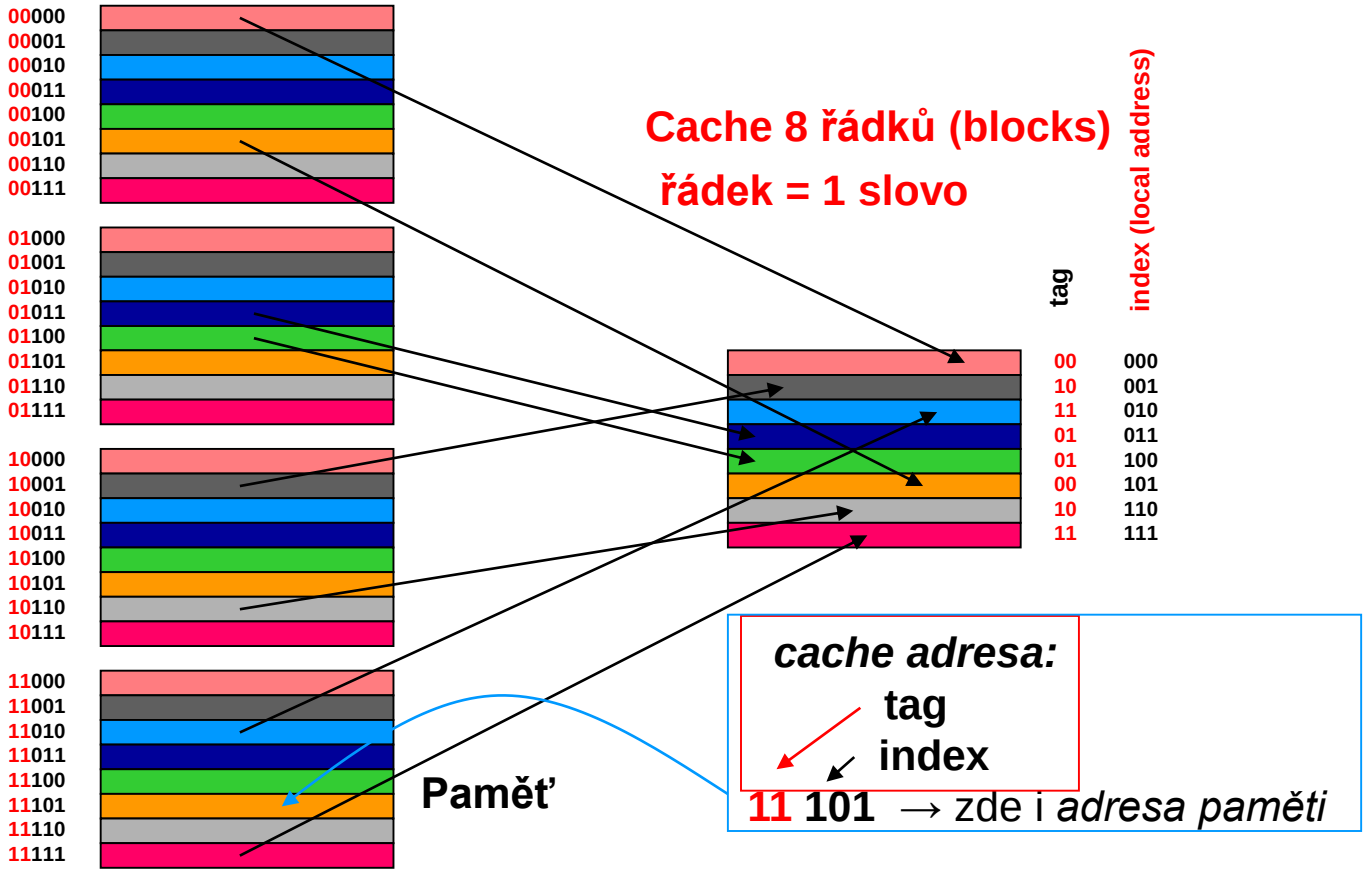
a vybere se slovo offset-em bloku

jinak došlo na „cache miss“

a musí se načíst celá řádka z hlavní paměti

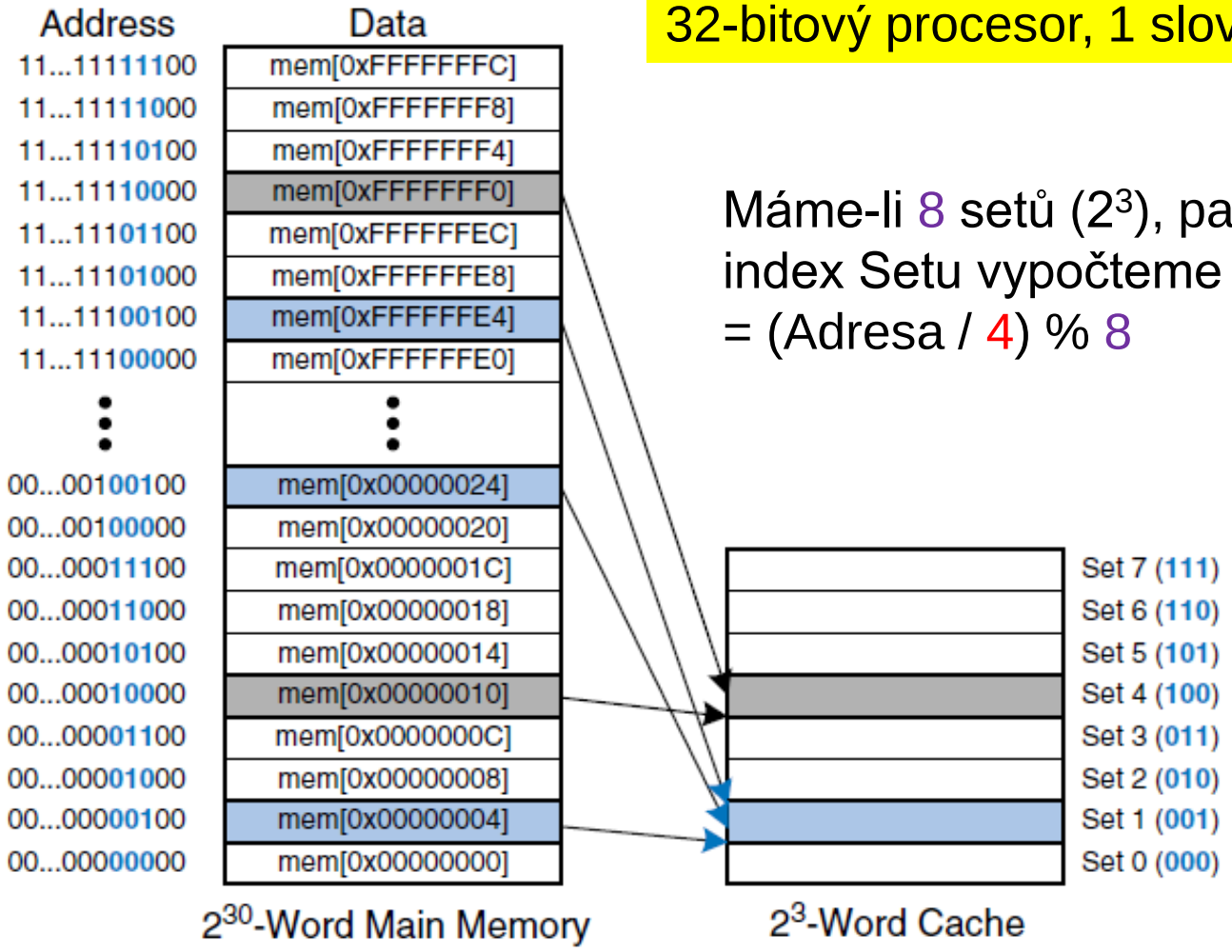
Příklad Direct-Mapped Cache 8-bitový procesor

8-bitový processor (1 slovo = 1 byte) adresující paměť o délce 32 bytů



Příklad: Direct Mapped Cache

32-bitový procesor, 1 slovo = 4 byty

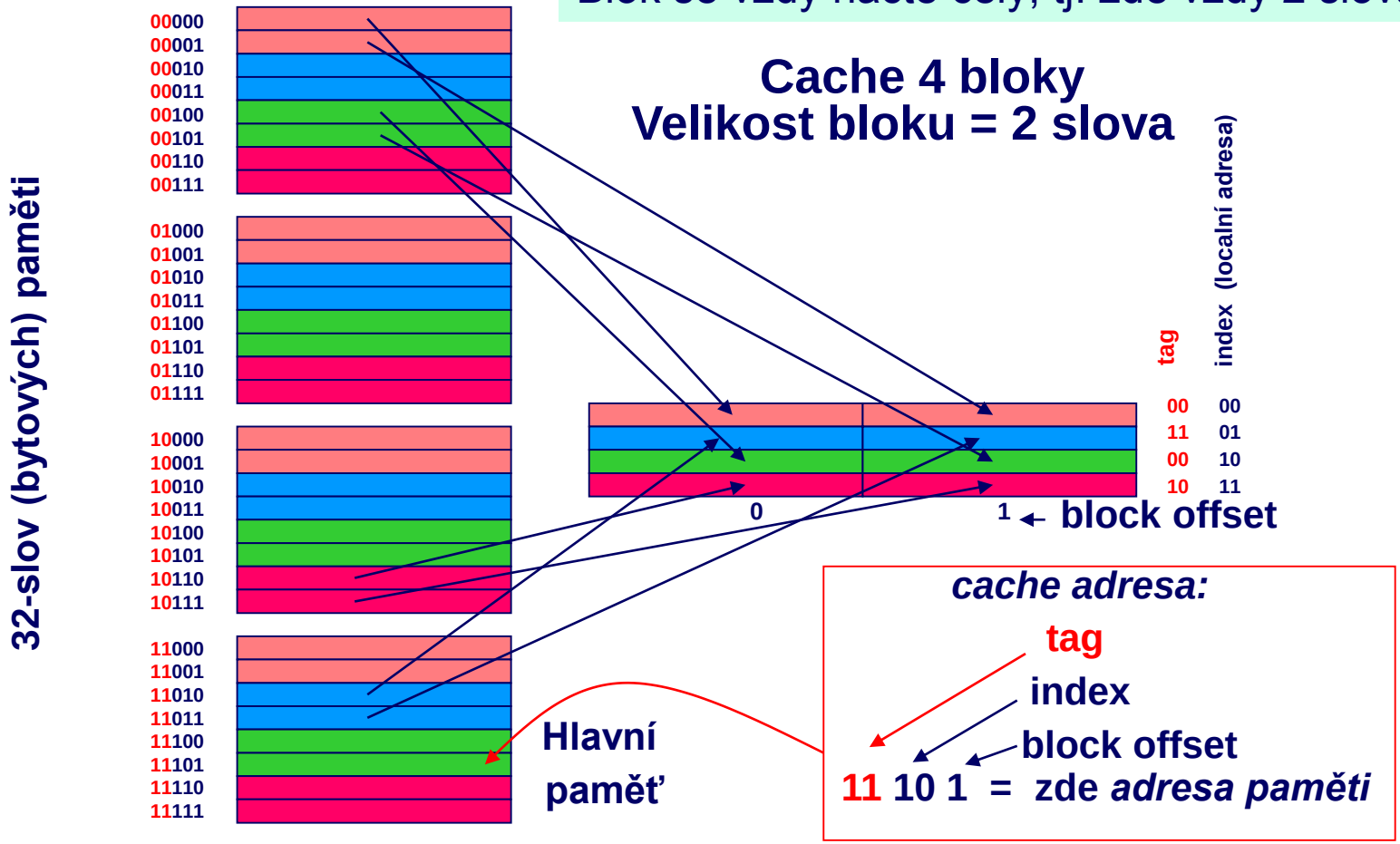


Máme-li 8 setů (2^3), pak index Setu vypočteme takto = (Adresa / 4) % 8

Direct-Mapped Cache – velikost bloku 2 slova

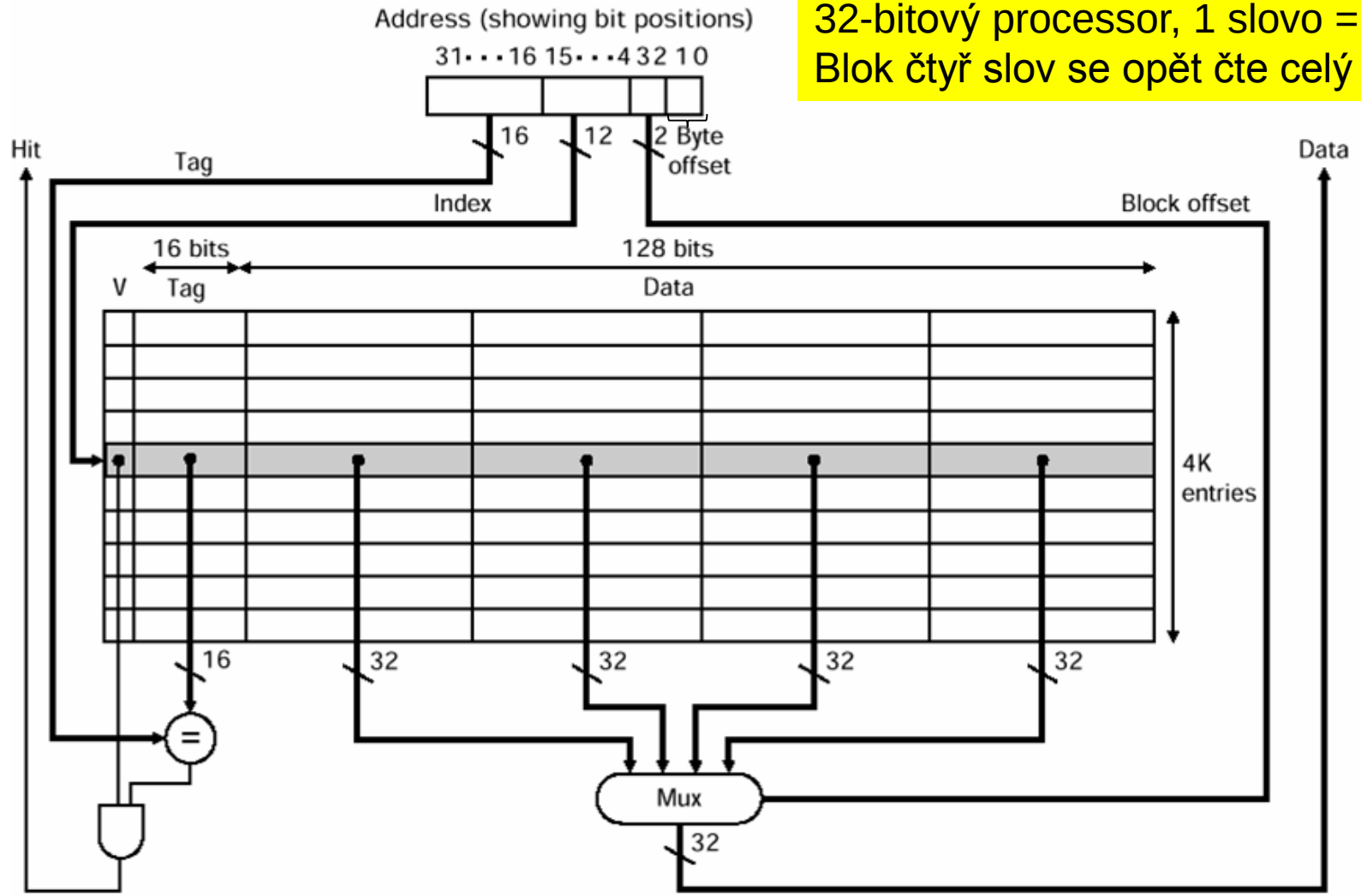
8-bitový processor (1 slovo = 1 byte) adresující paměť o délce 32-bytů

Blok se vždy načte celý, tj. zde vždy 2 slova!



Implementace Direct Mapped Cache

32-bitový processor, 1 slovo = 4 byty
Blok čtyř slov se opět čte celý



Příklad: Direct Mapped Cache
Přímo mapovaná cache

Následující názorný příklad vytvořil

PSOE Dan Garcia

`www.cs.berkeley.edu/~ddgarcia`

Jeho snímky se jen nepatrně upravily
a byly ponechané v angličtině.

Accessing data in a direct mapped cache

Ex.: 16KB of data, direct-mapped, 4 word blocks

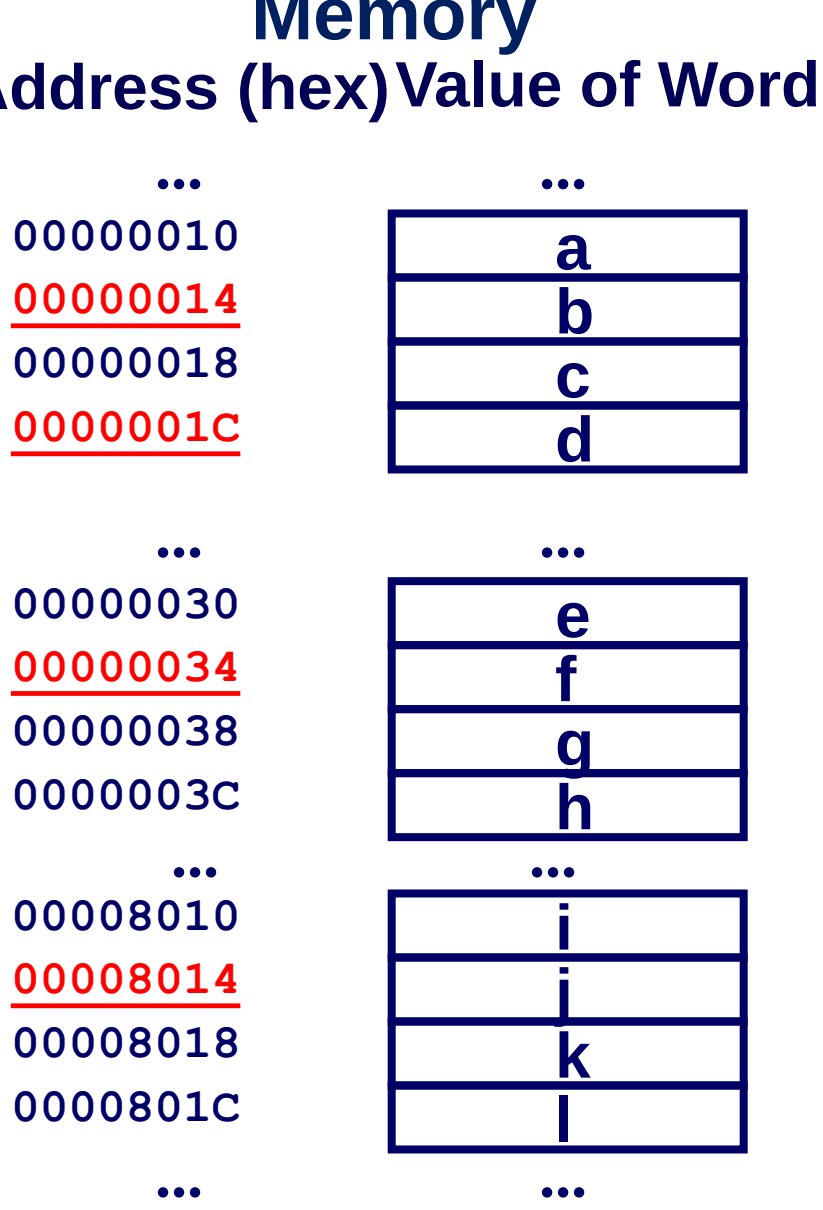
Read 4 addresses

1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x00008014

Memory values on right:

- The letter **a**, **b**..., and **l** - only indicate some 32-bit values in main memory

32 bit processor, its 1 word has 4 bytes



Accessing data in a direct mapped cache

4 Addresses:

- `0x00000014, 0x0000001C, 0x00000034, 0x00008014`

4 Addresses divided into
Tag | Index | Byte Offset fields

(they are separated below by spaces for convenience)

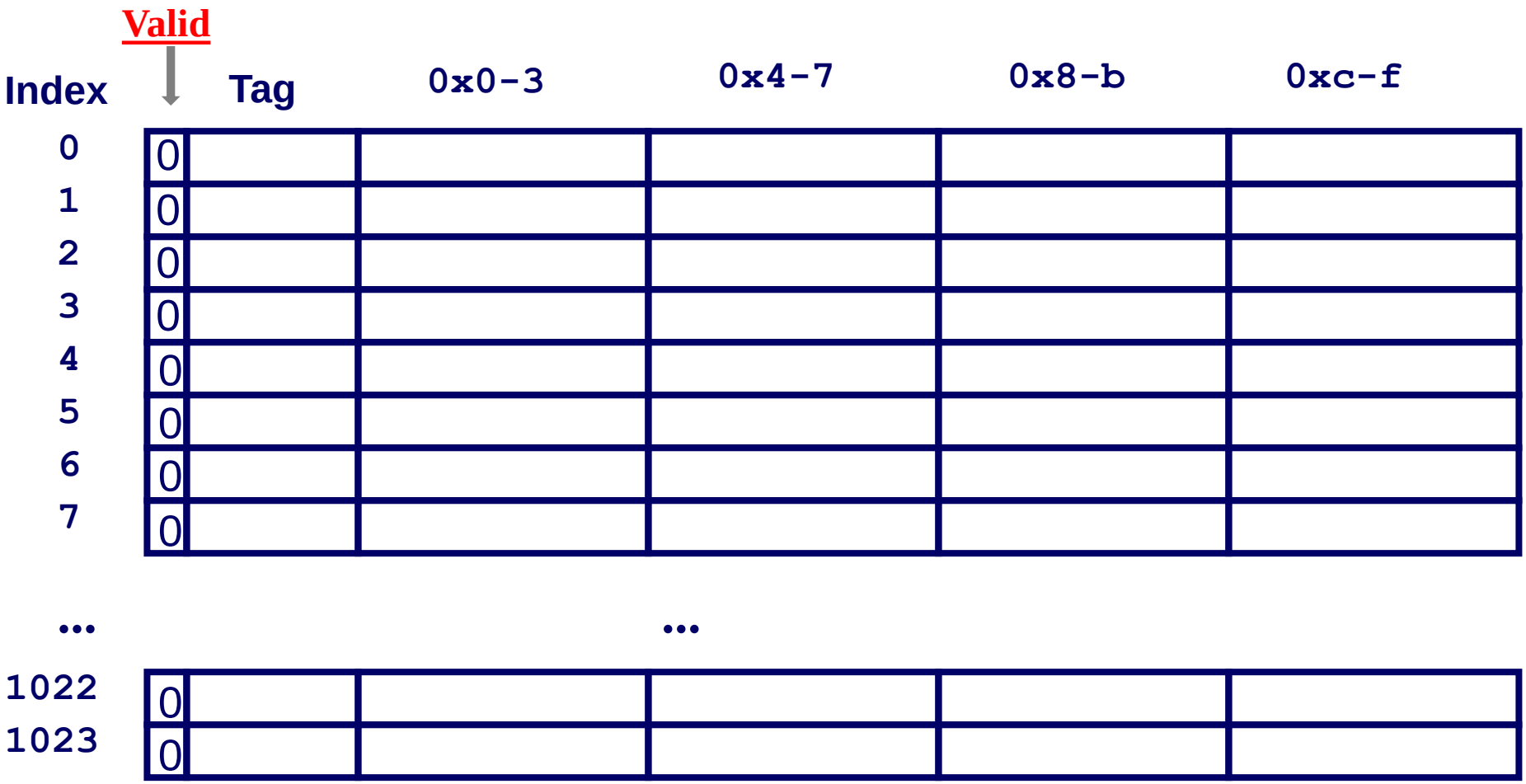
```
00000000000000000 0000000001 01 00
00000000000000000 0000000001 11 00
00000000000000000 0000000011 01 00
00000000000000010 0000000001 01 00
```

Tag **Index** **Offset**

16 KB Direct Mapped Cache, 16B blocks

Valid bit: determines whether anything is stored in that row

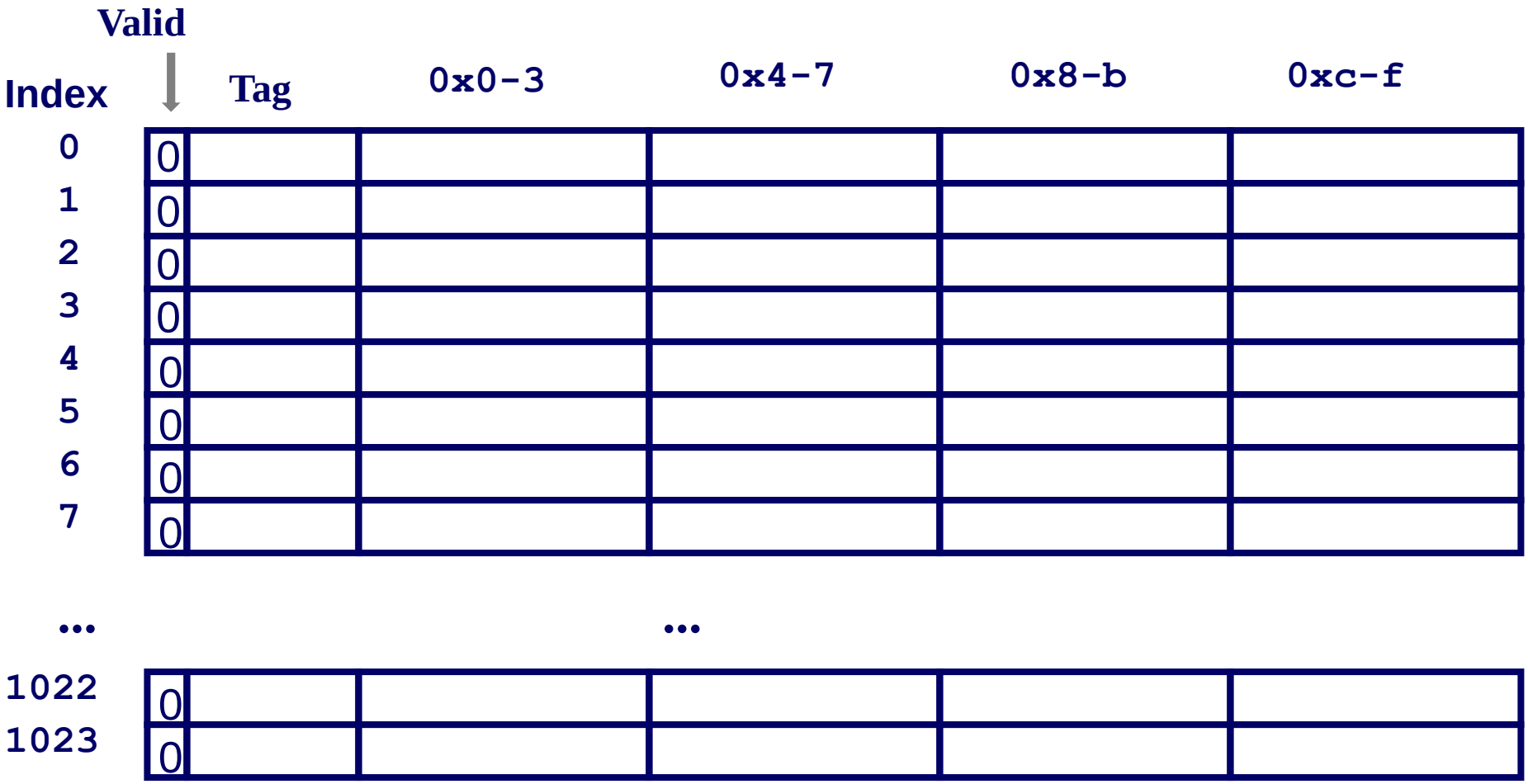
When computer is initially turned on after power-up, all entries are invalid.



1. Read 0x00000014

`00000000000000000 0000000001 01 00`

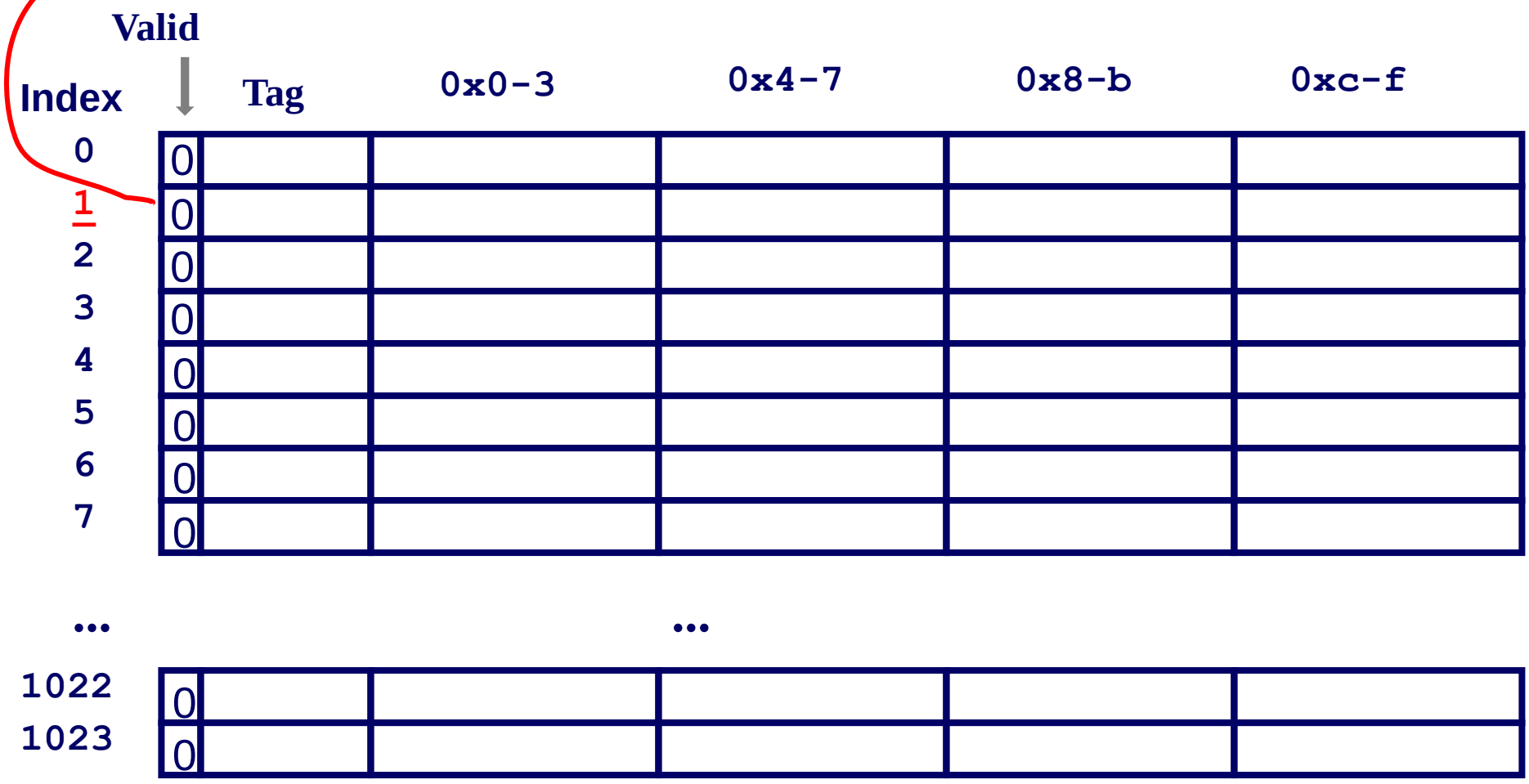
Tag field | Index field | Offset



So we read block 1 (0000000001)

0000000000000000 0000000001 01 00

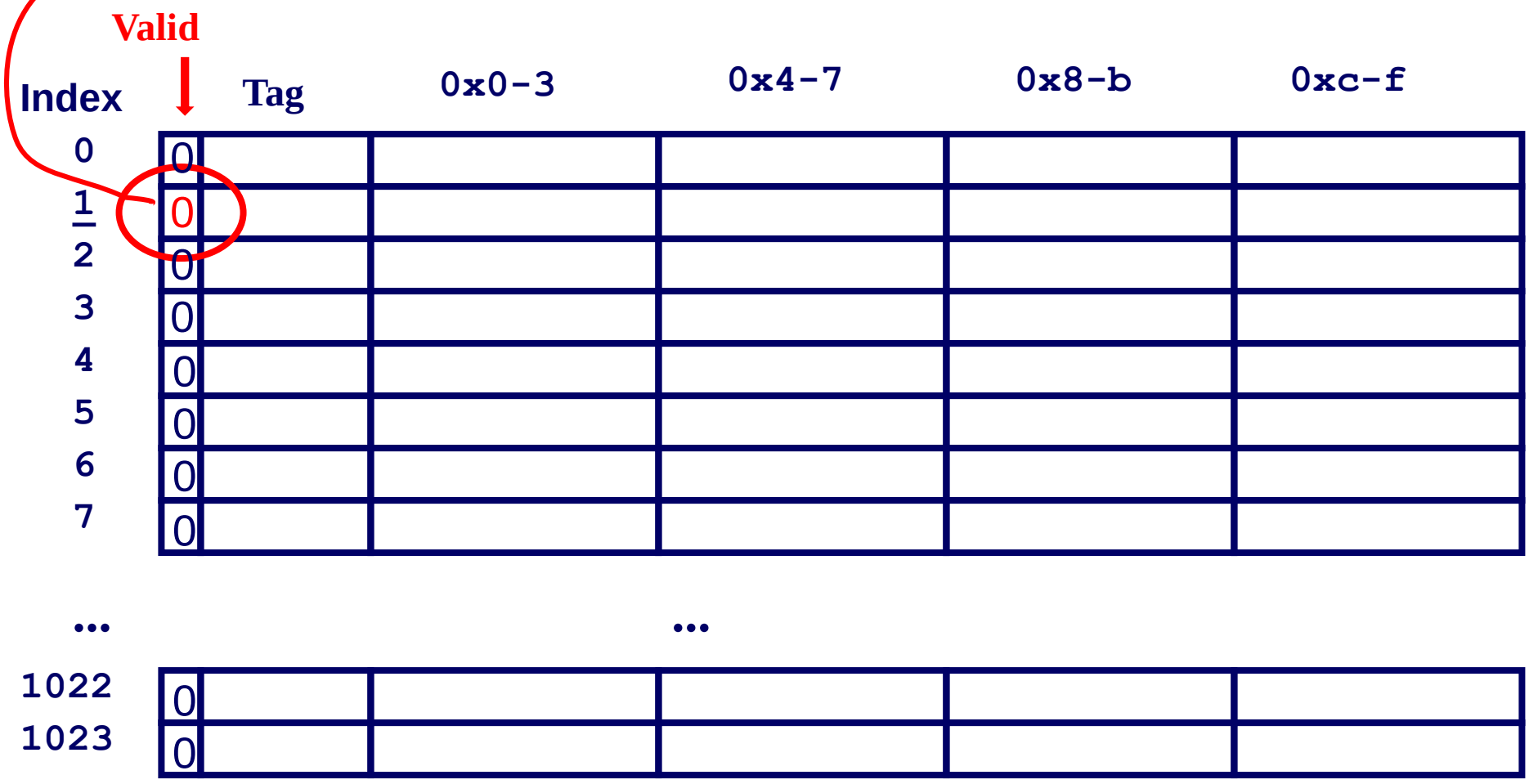
Tag field | Index field | Offset



No valid data !

00000000000000000000 0000000001 01 00

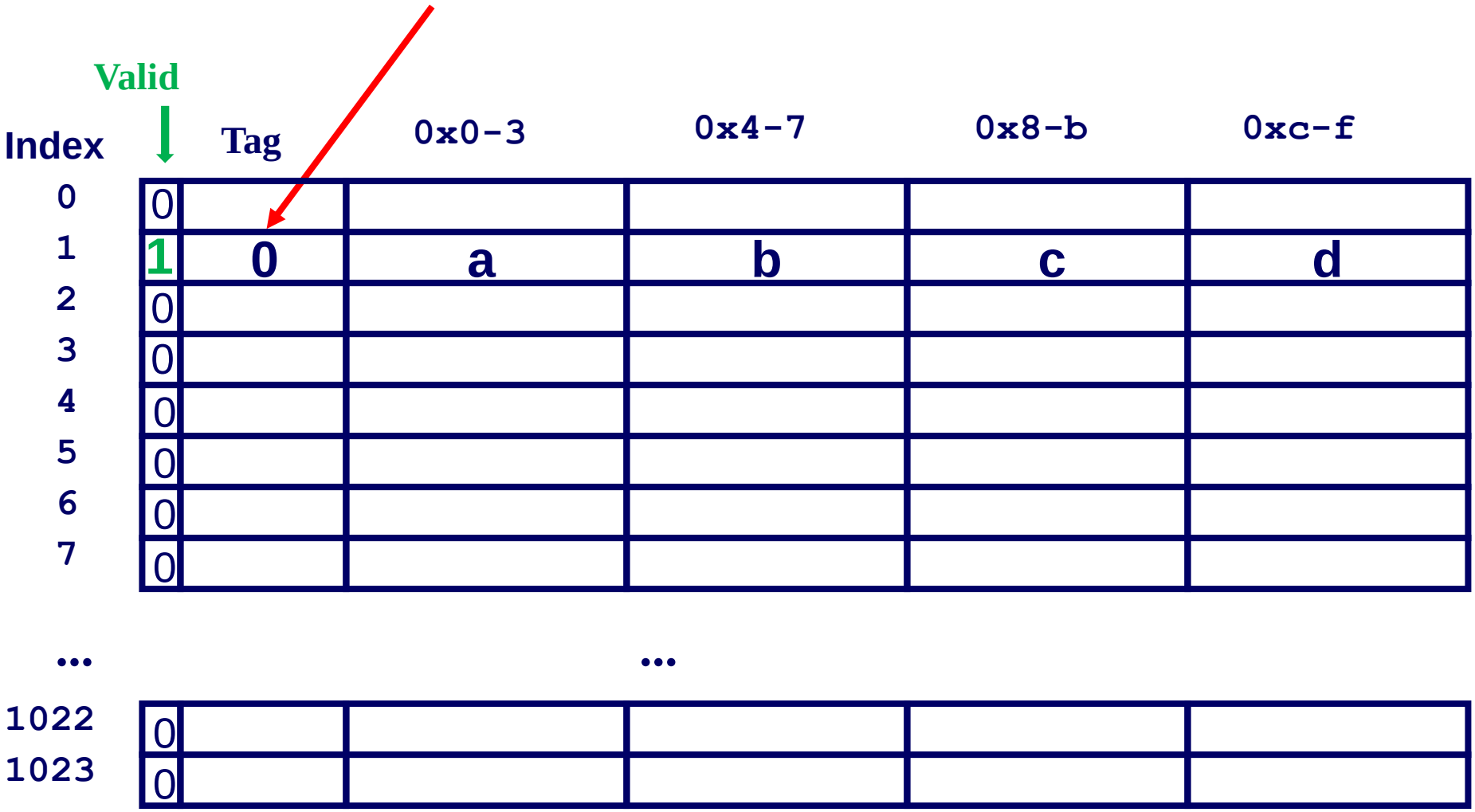
Tag field | Index field | Offset



So load that data into cache, setting tag, and valid

`00000000000000000000 0000000001 01 00`

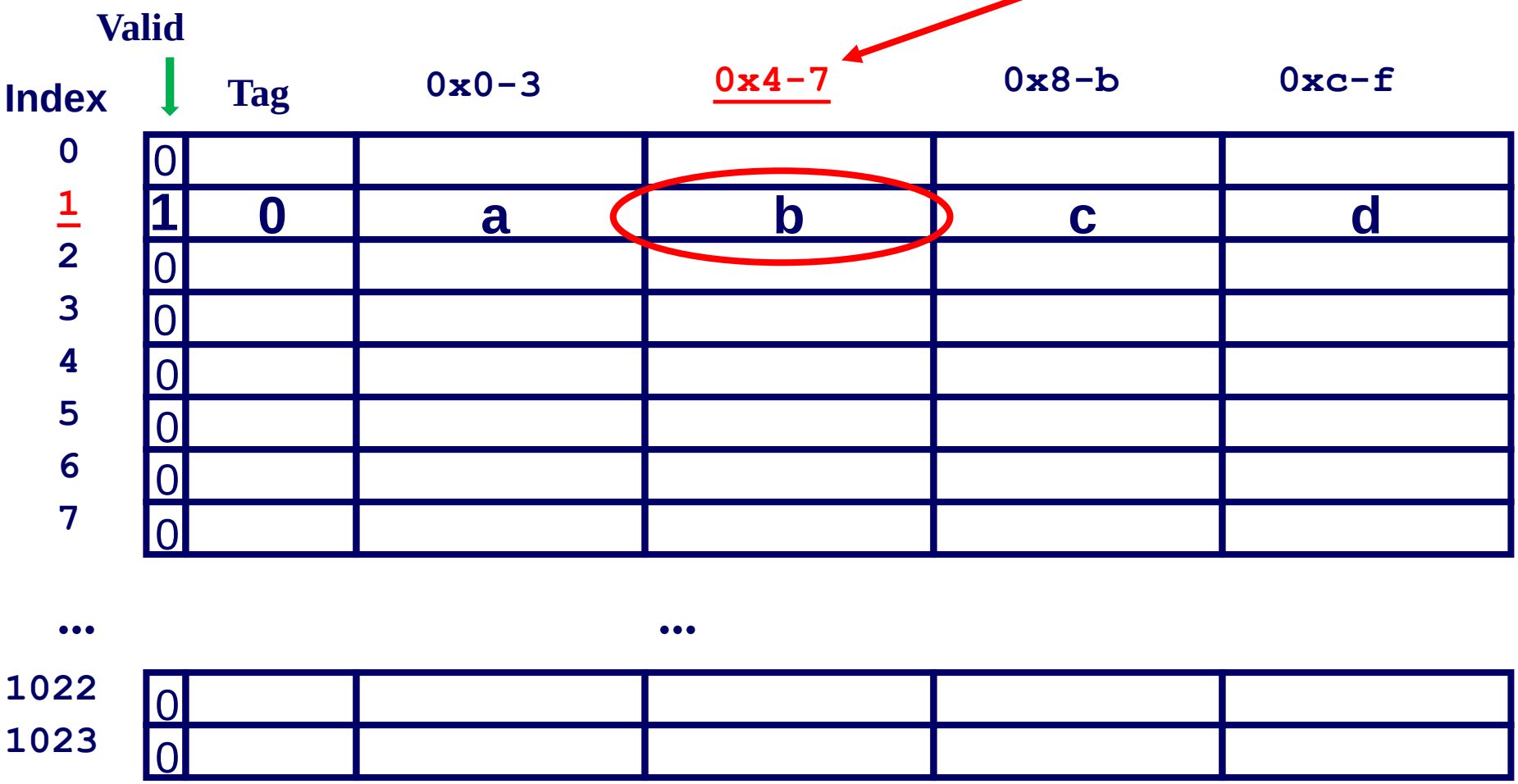
Tag field | Index field | Offset



Read from cache at offset, return word b

0000000000000000 0000000001 01 00

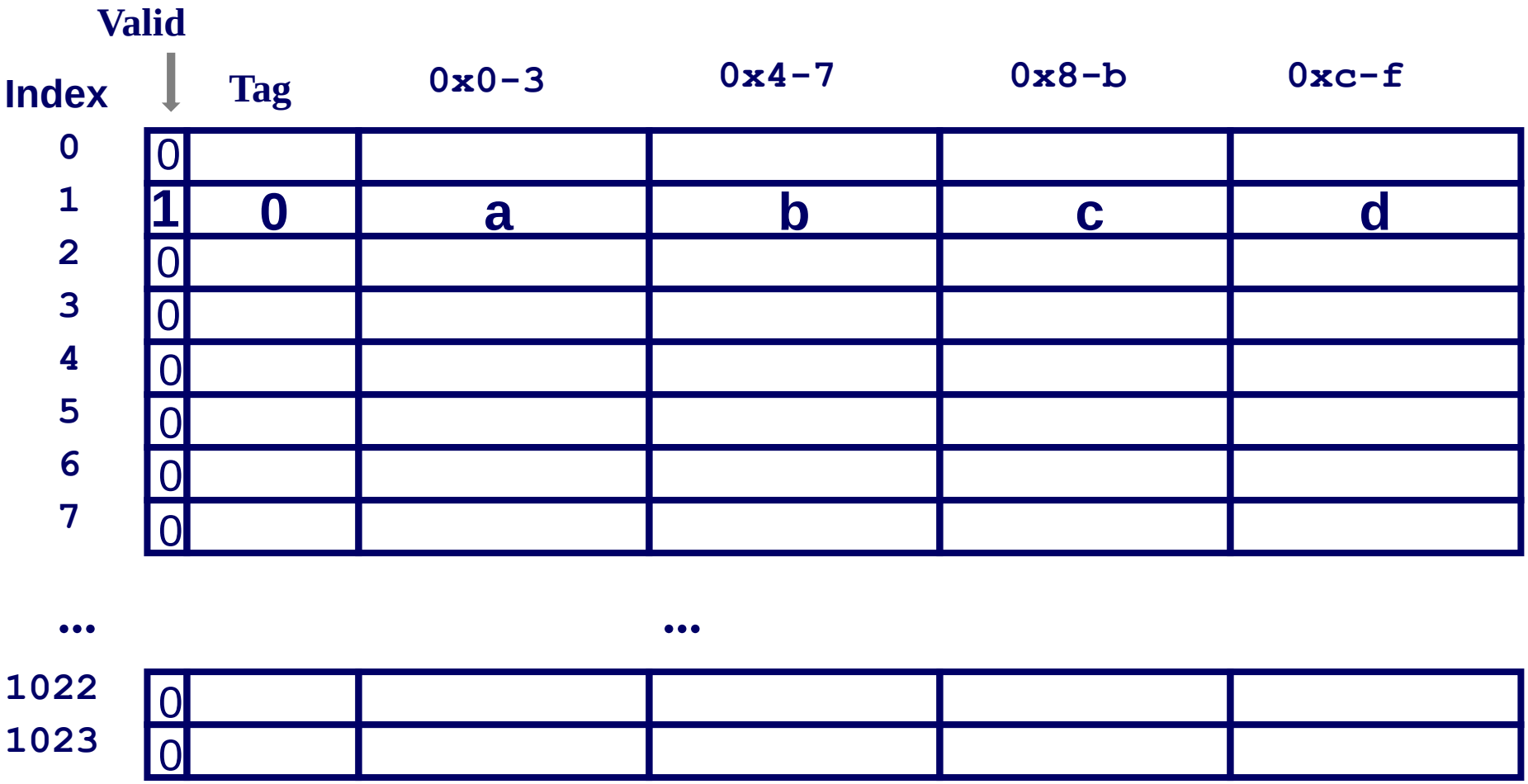
Tag field | Index field | Offset



2. Read 0x0000001C = 0…00 0..001 1100

```
00000000000000000000 0000000001 11 00
```

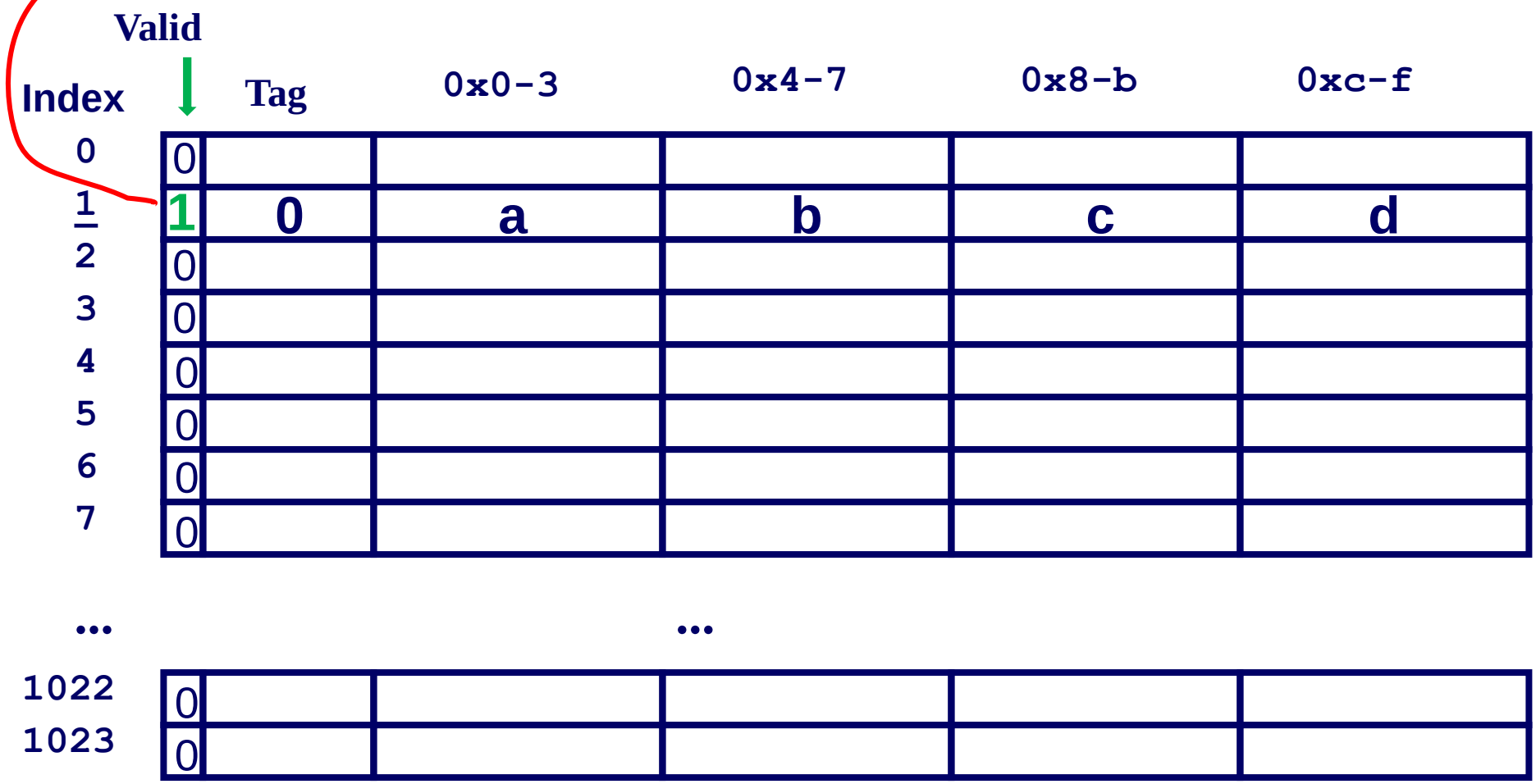
Tag field | Index field | Offset



Index is Valid !

`00000000000000000000 0000000001 11 00`

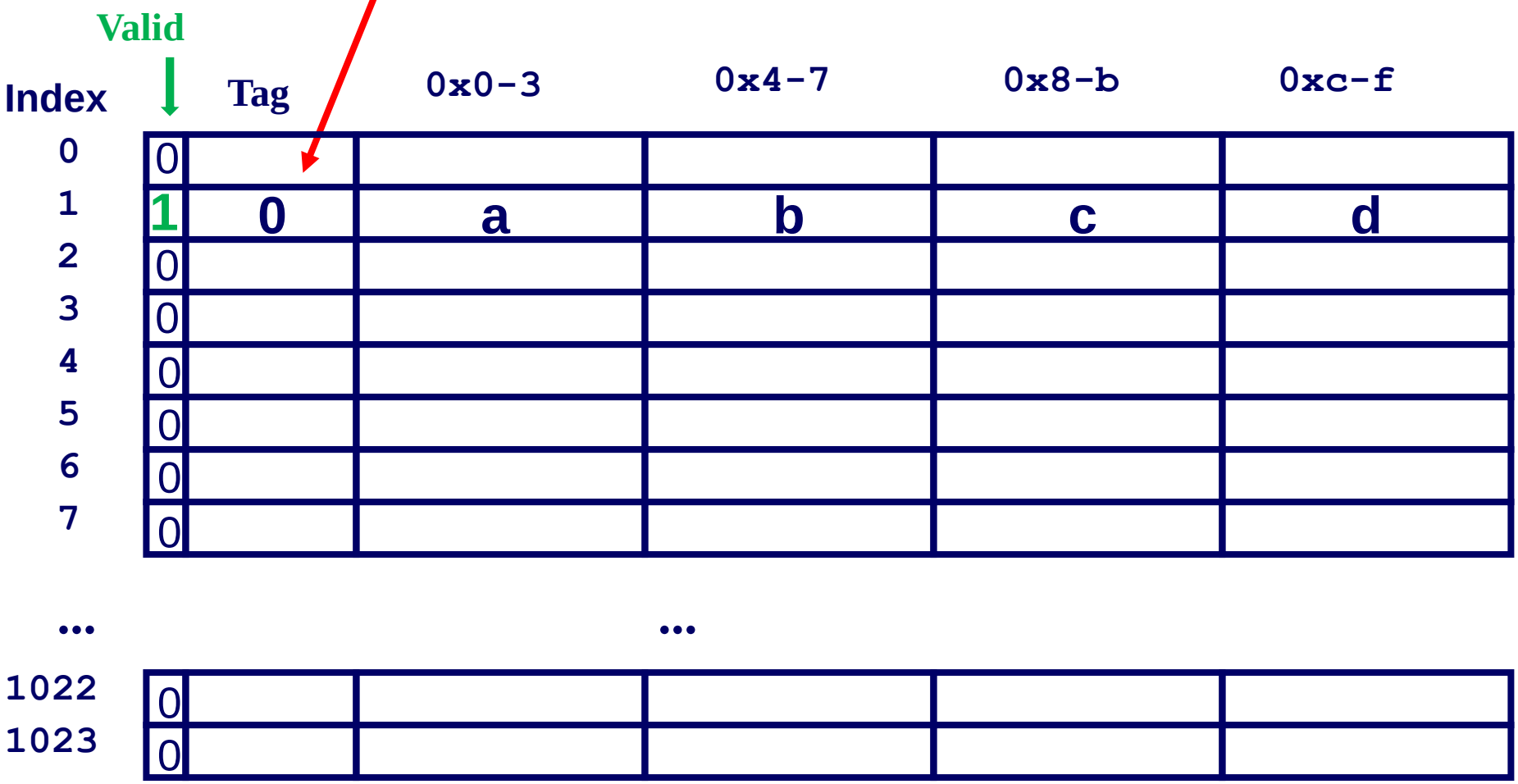
Tag field | Index field | Offset



Index valid, Tag Matches

00000000000000000000 0000000001 11 00

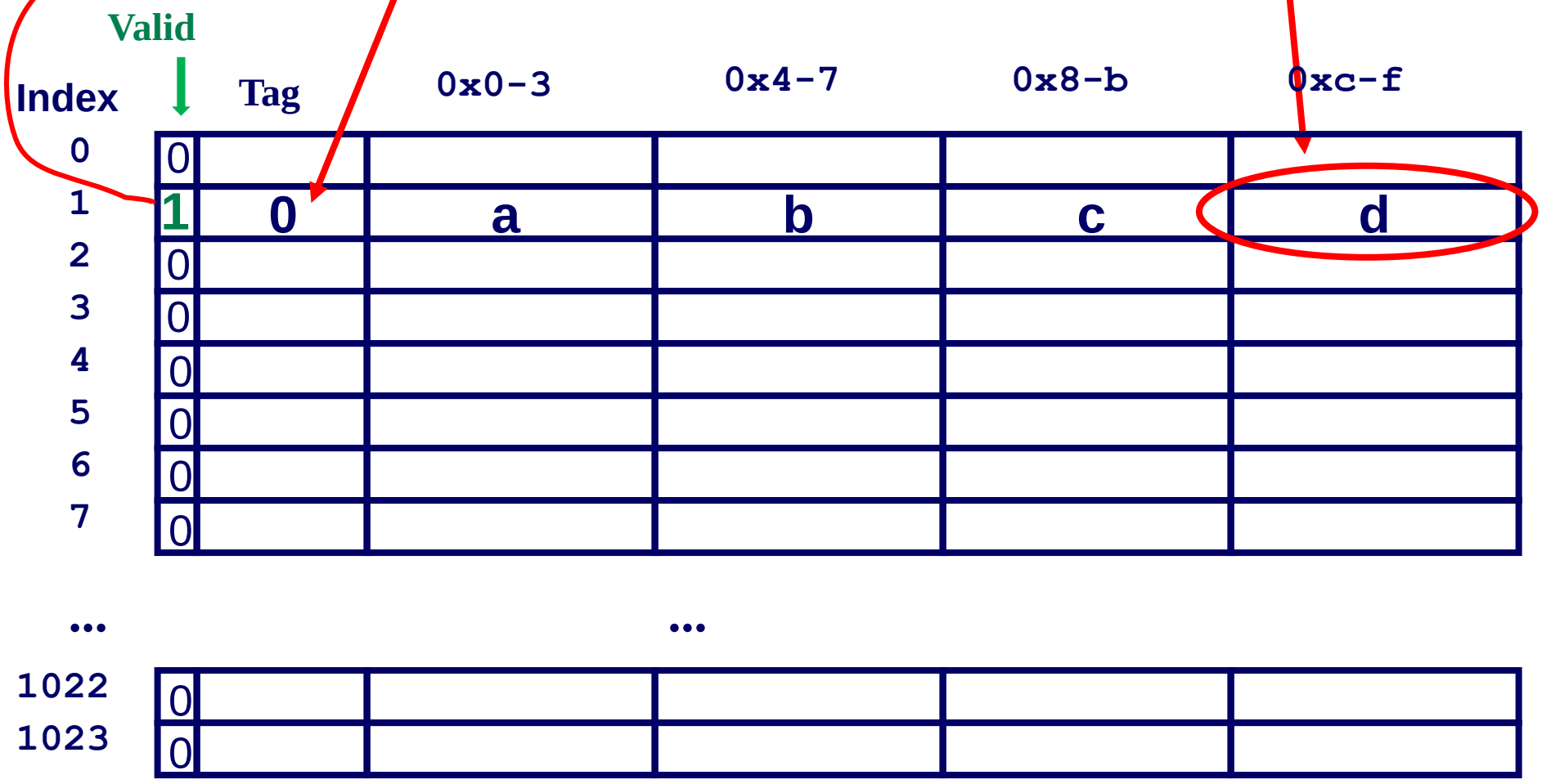
Tag field | Index field | Offset



Index Valid, Tag Matches, return d

00000000000000000000 0000000001 11 00

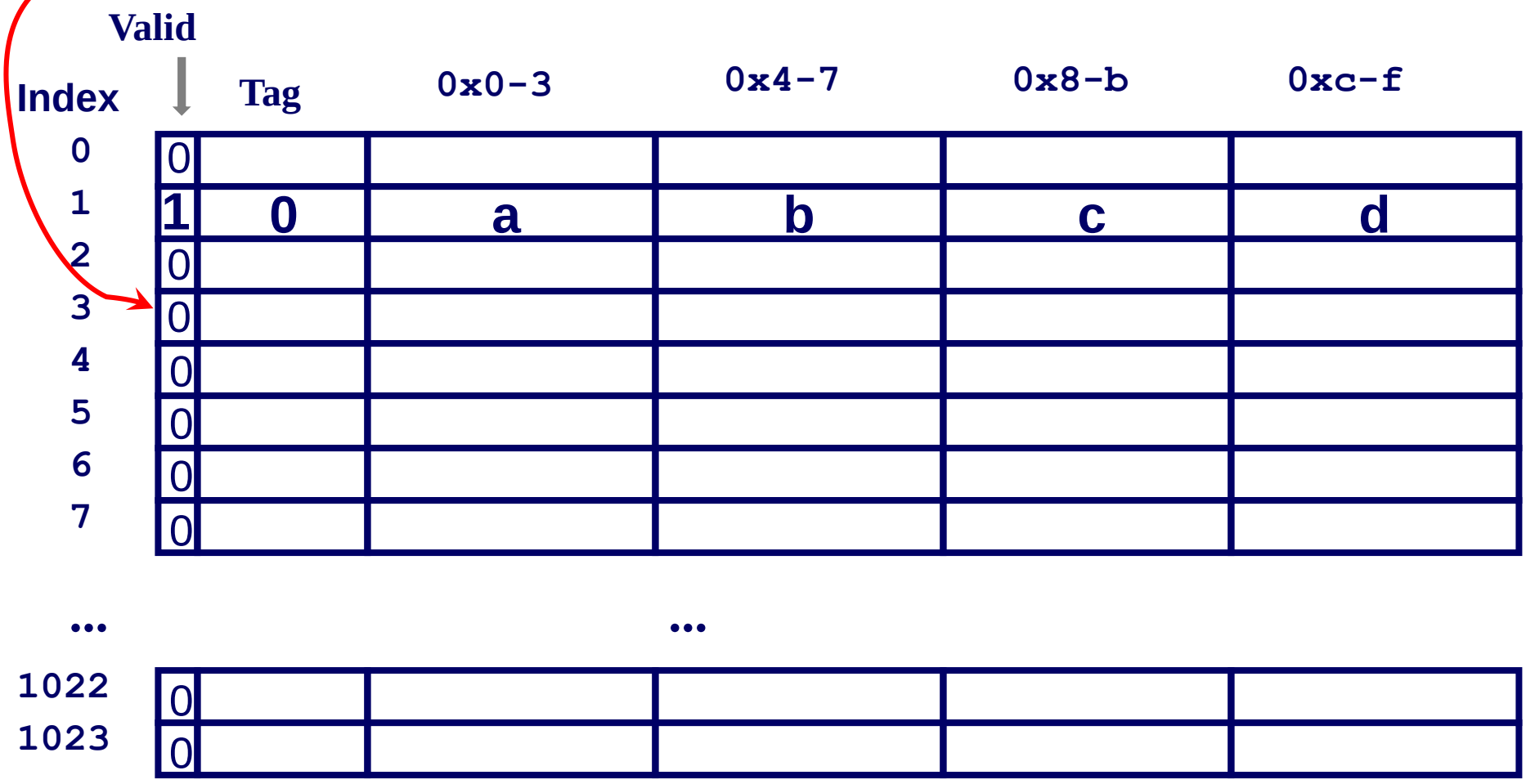
Tag field | Index field | Offset



3. Read 0x00000034 = ..011 0100

00000000000000000000 0000000011 01 00

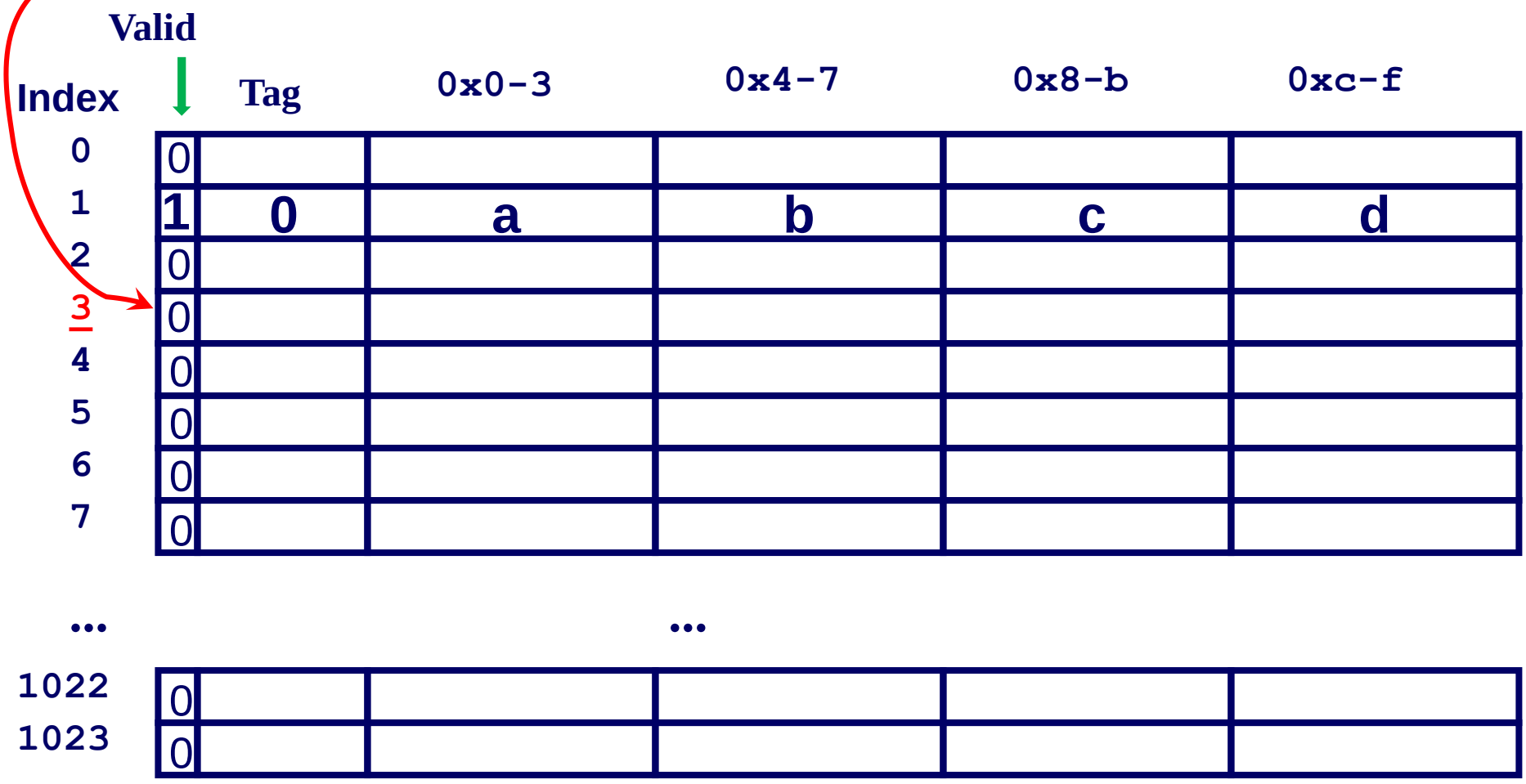
Tag field | Index field | Offset



So read block 3

0000000000000000 0000000011 01 00

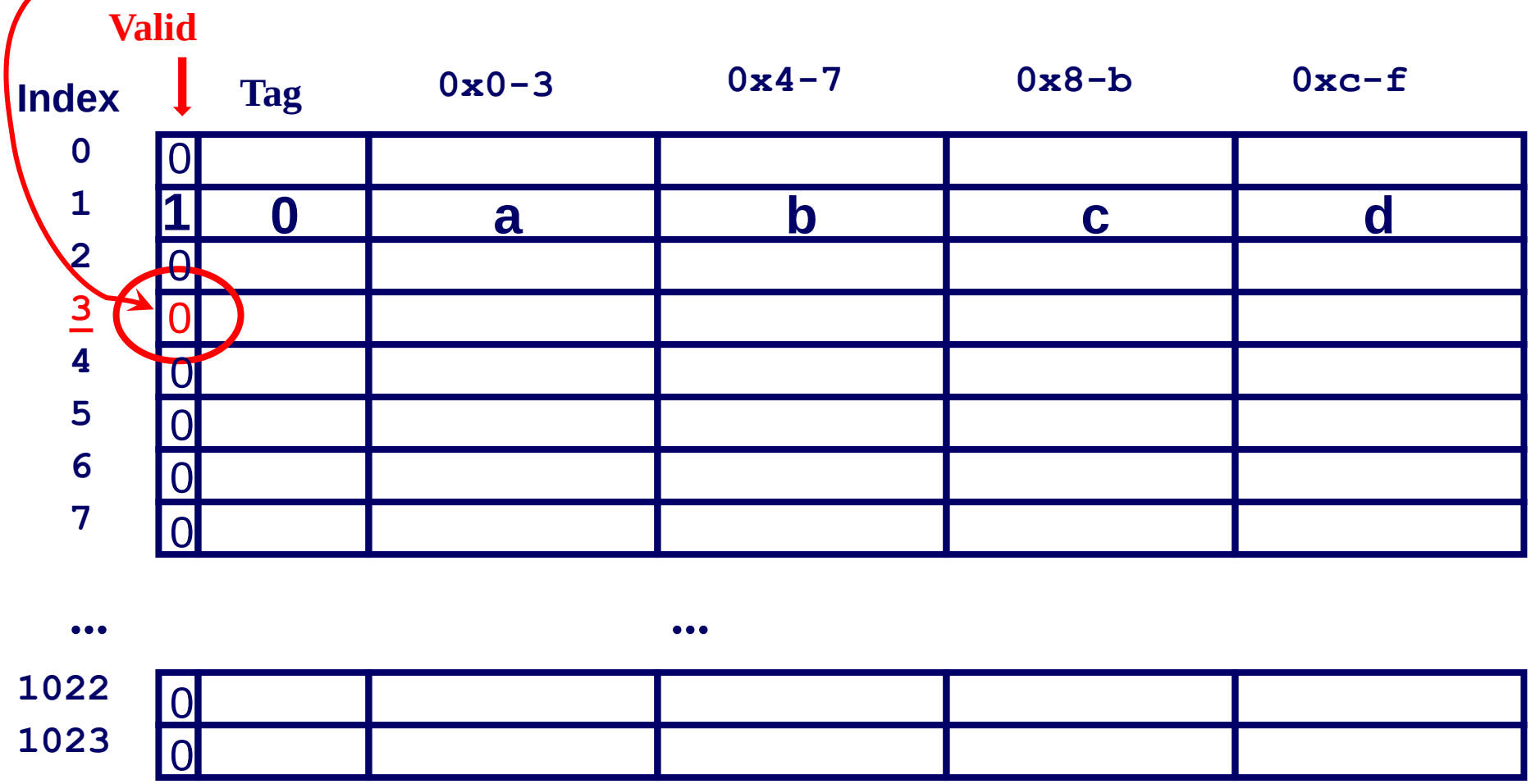
Tag field | Index field | Offset



No valid data

00000000000000000000 0000000011 01 00

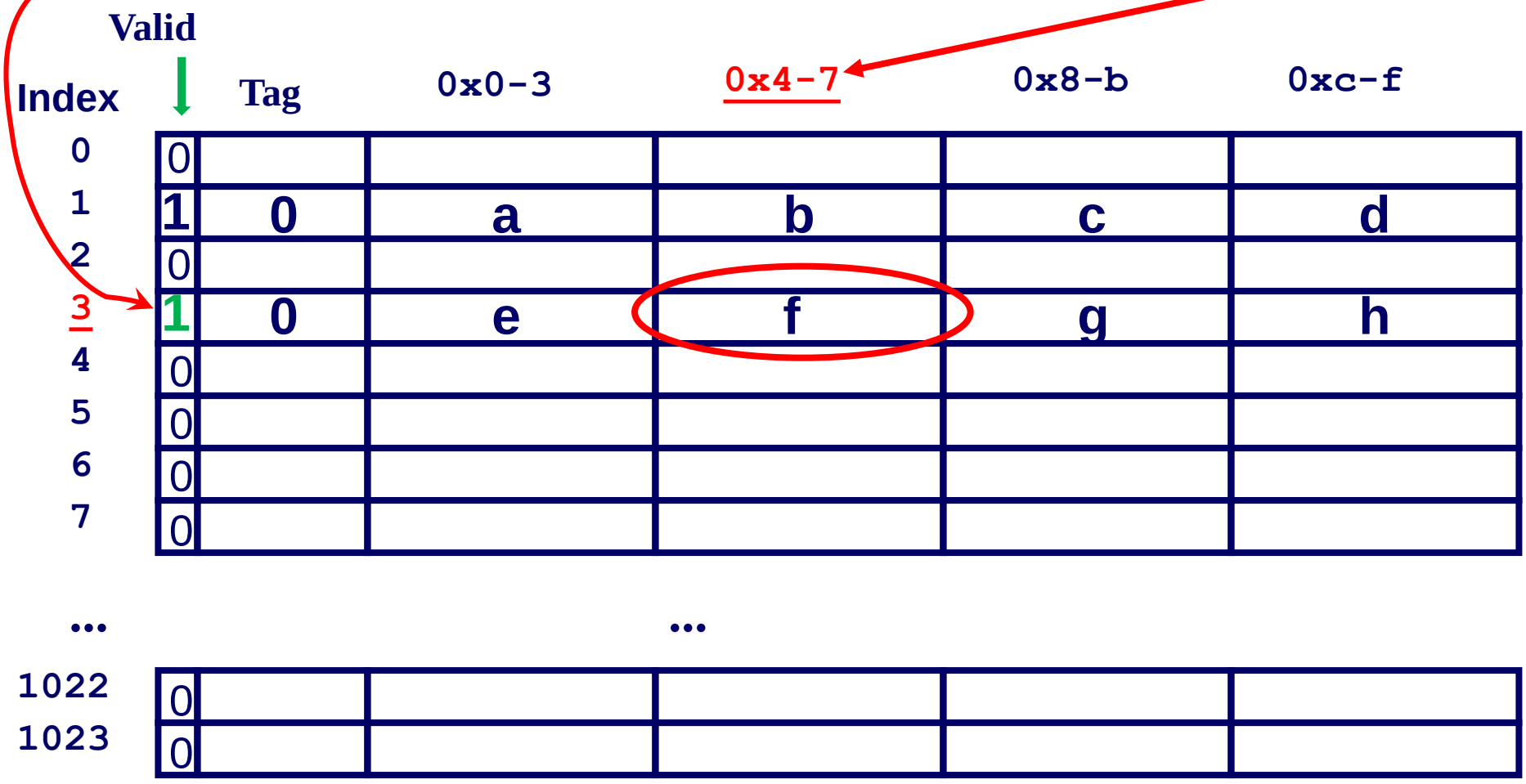
Tag field | Index field | Offset



Load that cache block, return word f

0000000000000000 0000000011 01 00

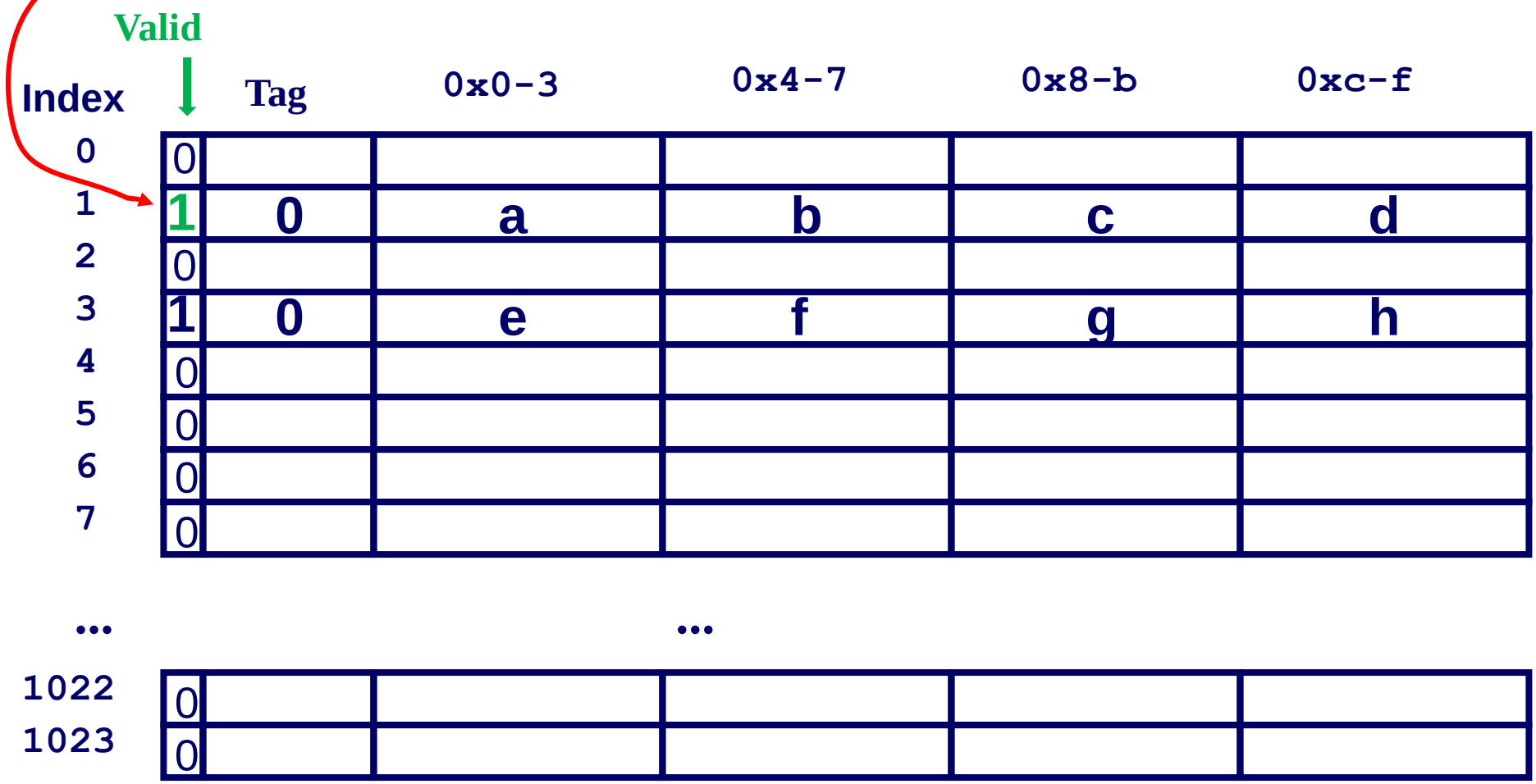
Tag field | Index field | Offset



4. Read 0x00008014 = ...1000 0000 0001 0100

0000000000000010 0000000001 01 00

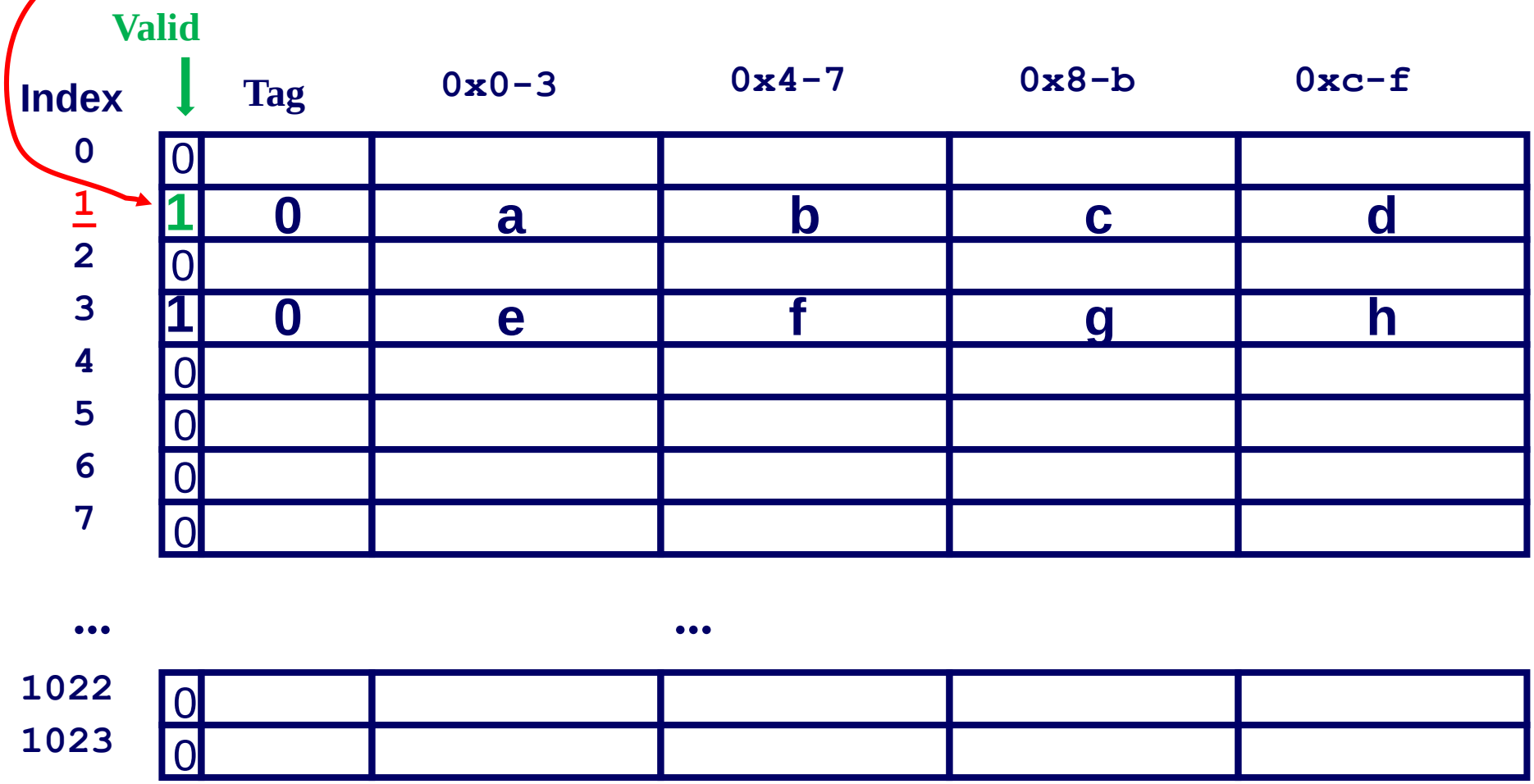
Tag field | Index field | Offset



So read Cache Block 1, Data is Valid

0000000000000010 0000000001 01 00

Tag field | Index field | Offset



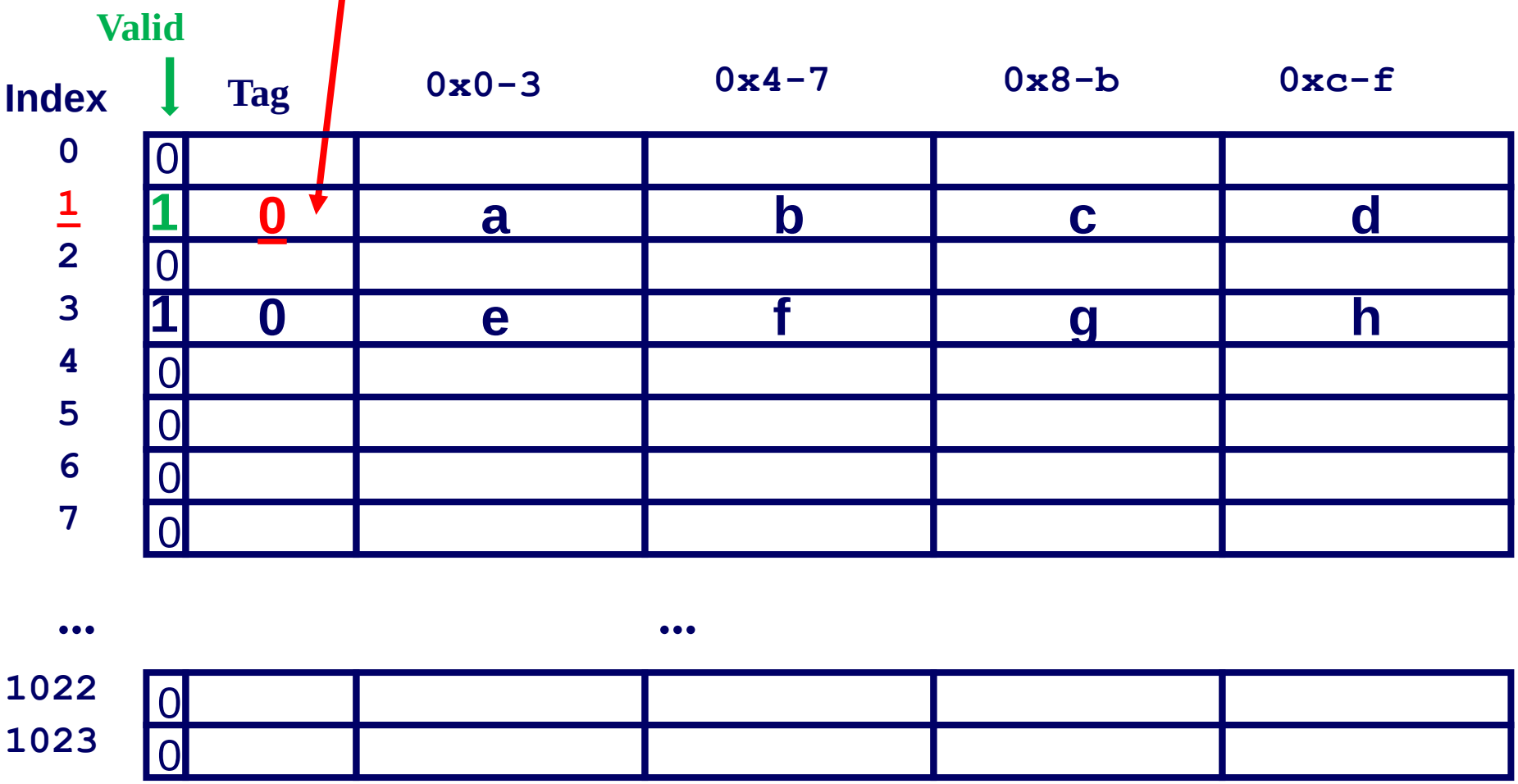
Cache Block 1 Tag does not match (0 != 2)

0000000000000010 0000000001 01 00

Tag field is different

Index field

Offset



Miss, so replace block 1 with new data & tag

`0000000000000010 0000000001 01 00`

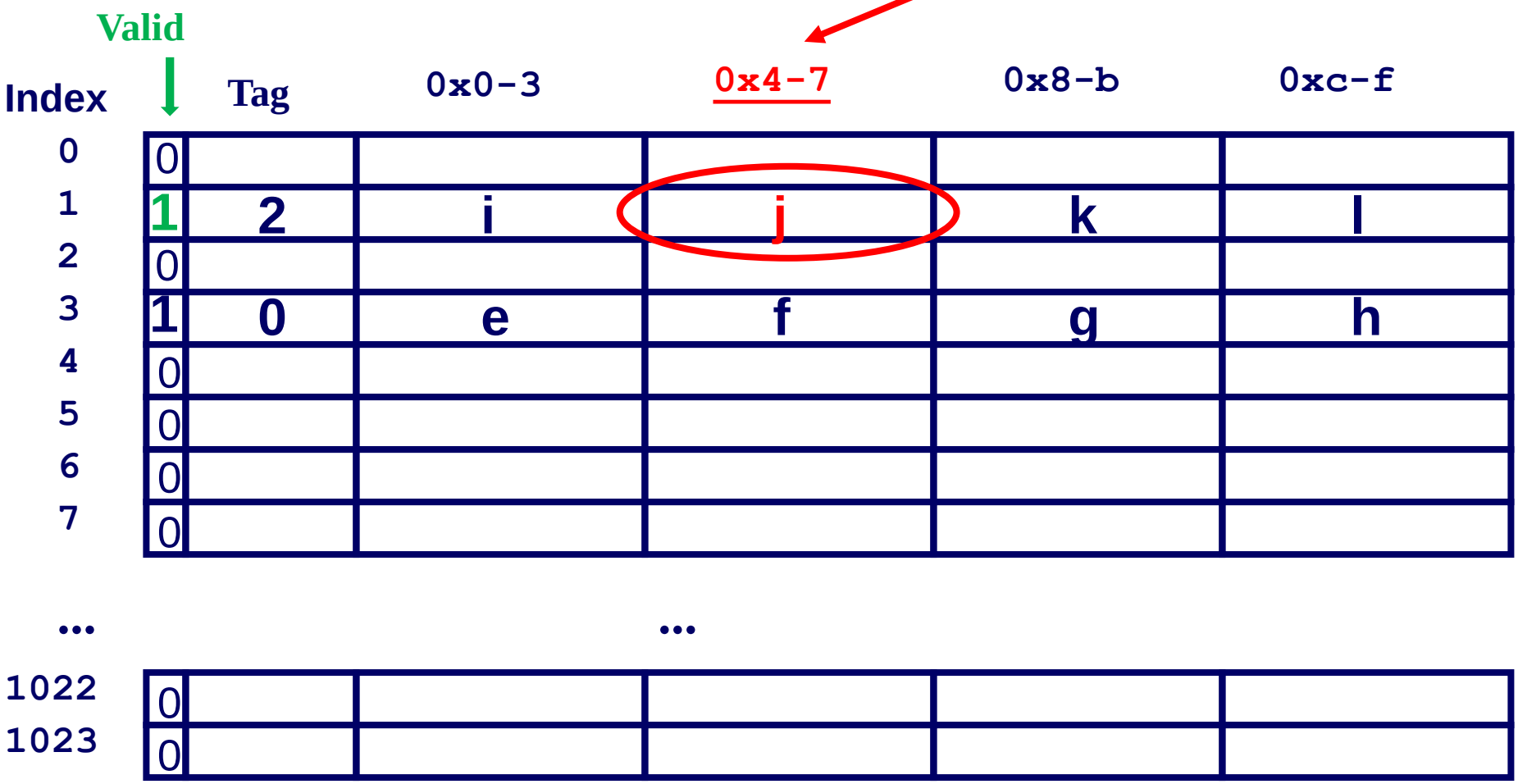
Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

And return word j

0000000000000010 0000000001 01 00

Tag field | Index field | Offset



The end of Garcia's example. Continue by yourself!

* Read address 0x00000030 !

  00000000000000000000 0000000011 0000

* Read address 0x0000001c !

  00000000000000000000 0000000001 1100

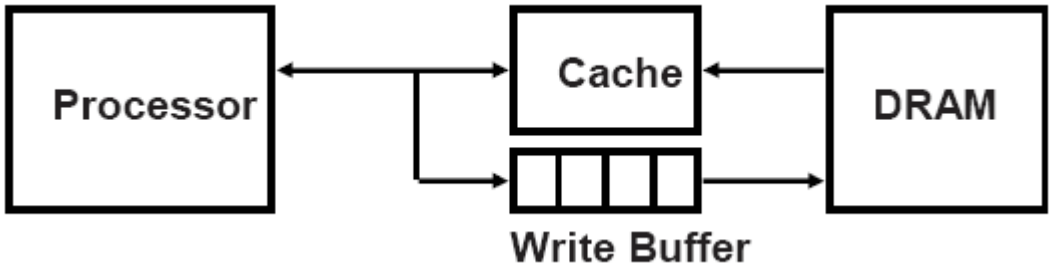
Cache

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		

Řešení situace: **Zápis dat** procesorem do paměti

Na cestě je i cache!

- **Konzistence dat** – samozřejmý požadavek na shodu obsahu stejných adres na různých médiích.
- **Write through** – současně se zápisem do cache se data zapíší do zápisové fronty a pak asynchronně do paměti.
- **Write back** – data se do cache zapíší s poznámkou **Dirty** (D bit řádky). Ke skutečnému zápisu dat do hlavní paměti dojde až v okamžiku případného rušení příslušného řádku v cache, kdy hrozí ztráta dat.



Realističtější formát řádky cache

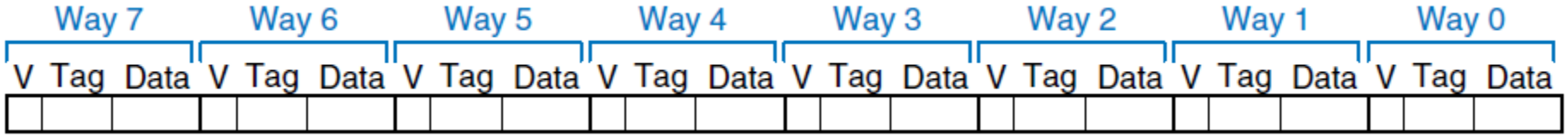
- **Tag** je index odpovídajícího bloku v operační paměti (v podstatě se jedná o hodnotu ukazatele/adresy celočíselně vydělenou délkou celého bloku v bytech).
- **Data** pole obsahující vlastní hodnoty na příslušných adresách či příslušné adrese.
- **Validity bit** – bit platnosti. Indikuje, zda je obsah pole Data vůbec platný.

V	Další bity, např. D	Tag	Data

Dirty bit – pomocná informace v cache, která rozšiřuje pole v obsahu paměti. Indikuje, že řádek obsahuje **jinou hodnotu** než hlavní paměť, takže bude nutné zapsat změnu před nahráním nového obsahu do řádku.

Fully Associative Cache - Plně asociativní cache

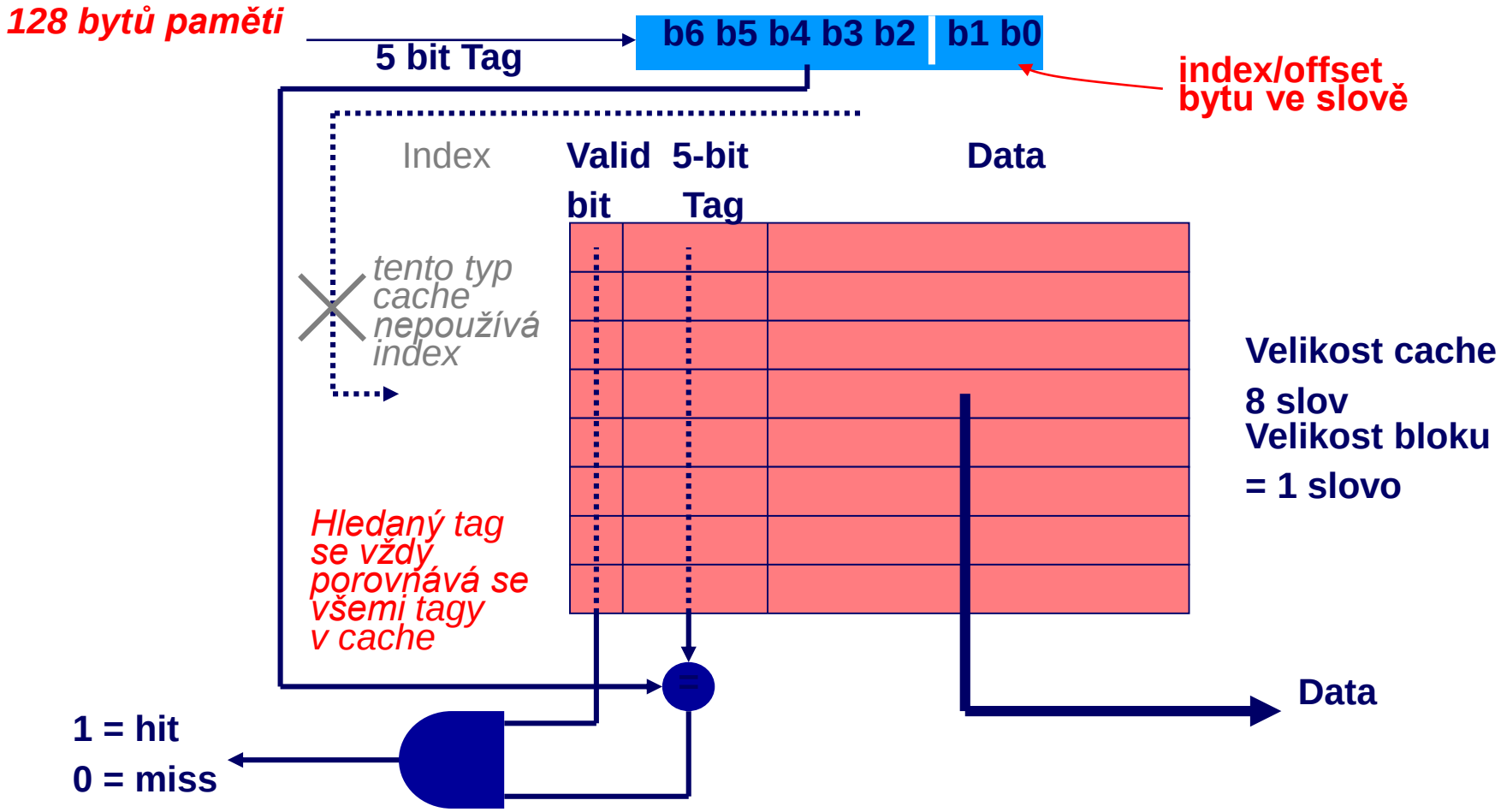
- Plně asociativní cache obsahuje jenom jediný set a její stupeň asociativity je roven počtu bloků (N=B). Adresa paměti se může mapovat kamkoliv.
- … má pro danou kapacitu má nejméně konfliktů, ale potřebuje nejvíce HW prostředků (komparátory) – roste plocha čipu
- …je vhodná pro relativně malé cache.



Plně asociativní cache má S=1, tedy jediný set !

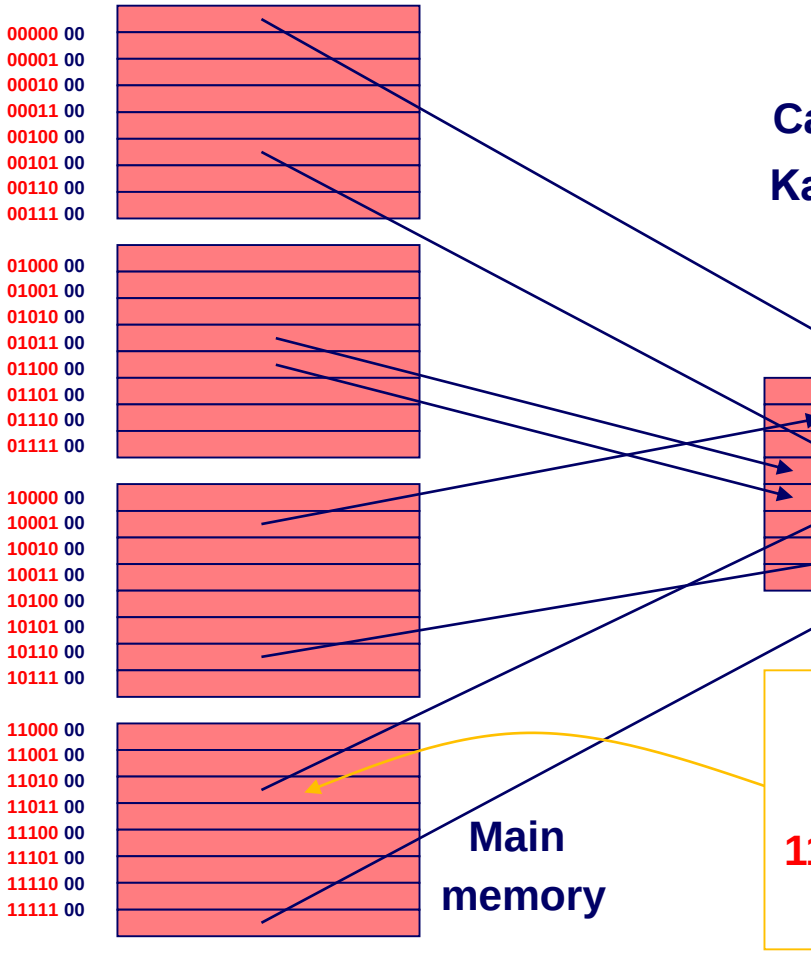
Nalezení slovo v plně asociativní cache

Paměť o délce 128 bytů a čtená/zapisovaná jako 4-bytová slova



Fully-Associative Cache - Plně asociativní cache

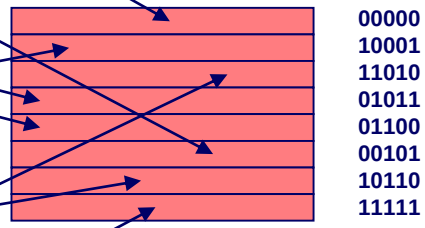
Paměť o délce 128 bytů a čtená/zapisovaná jako 4-bytová slova



Cache má 8 bloků

Každý blok obsahuje 1 čtyřbytové slovo

tagy v cache



Pořadí uložených tagů může být jakékoli dle přístupů z programu a správy cache

cache:

tag

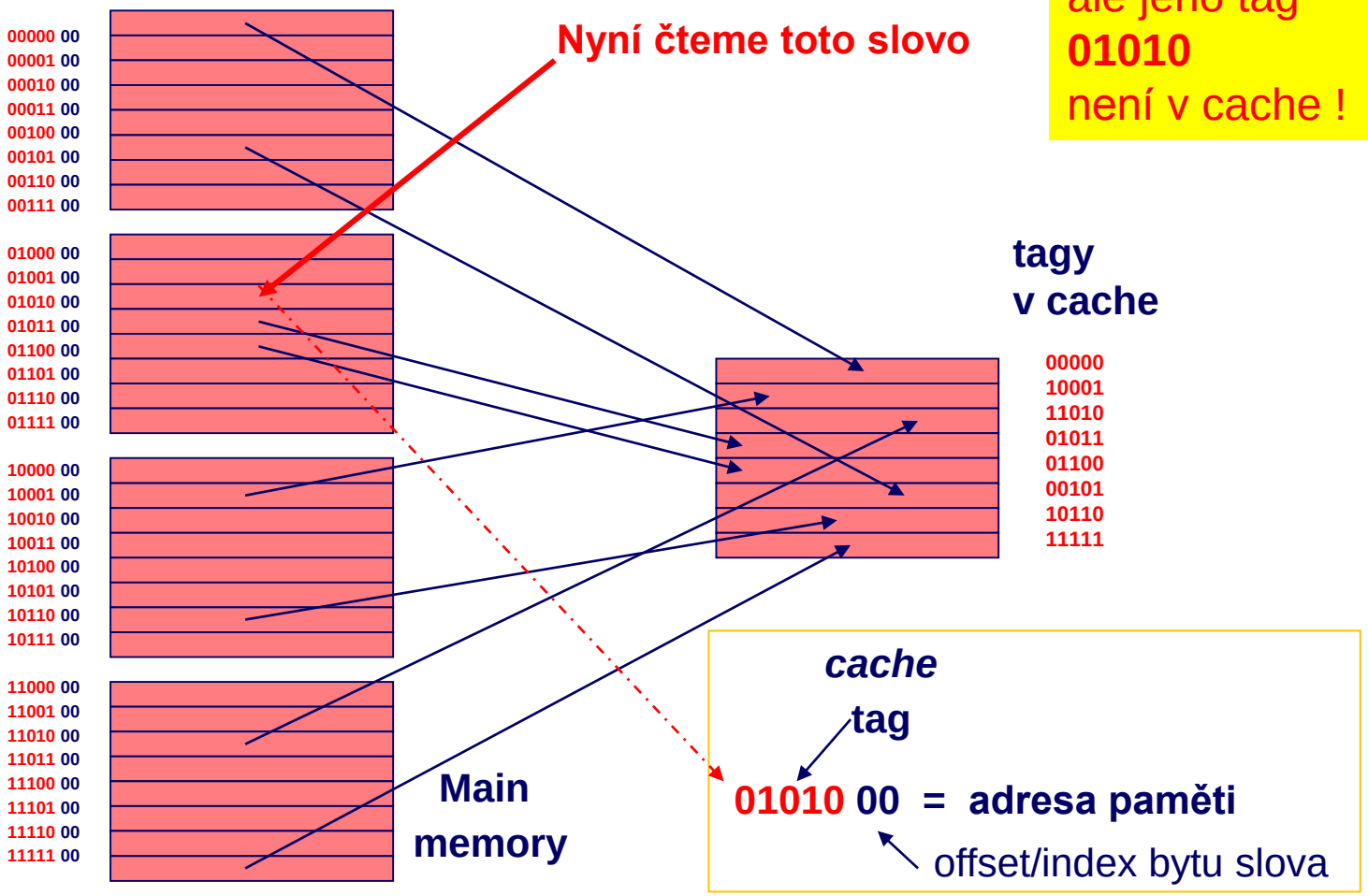
11011 00 = adresa paměti

offset/index bytu slova

Fully-Associative Cache

Paměť o délce 128 bytů a čtená/zapisovaná jako 4-bytová slova

Cache má 8 bloků (řádků) a každý blok obsahuje 1 čtyřbytové slovo



Řešení situace **Cache Miss**, data v cache nejsou

Data se nejprve musí z hlavní paměti přečíst.
Když je ale asociativní cache plná, co v ní nahradíme?
Přímo mapovaná cache měla jasnou volbu, ale zde není.

Strategie uvolňování bloků/řádek cache

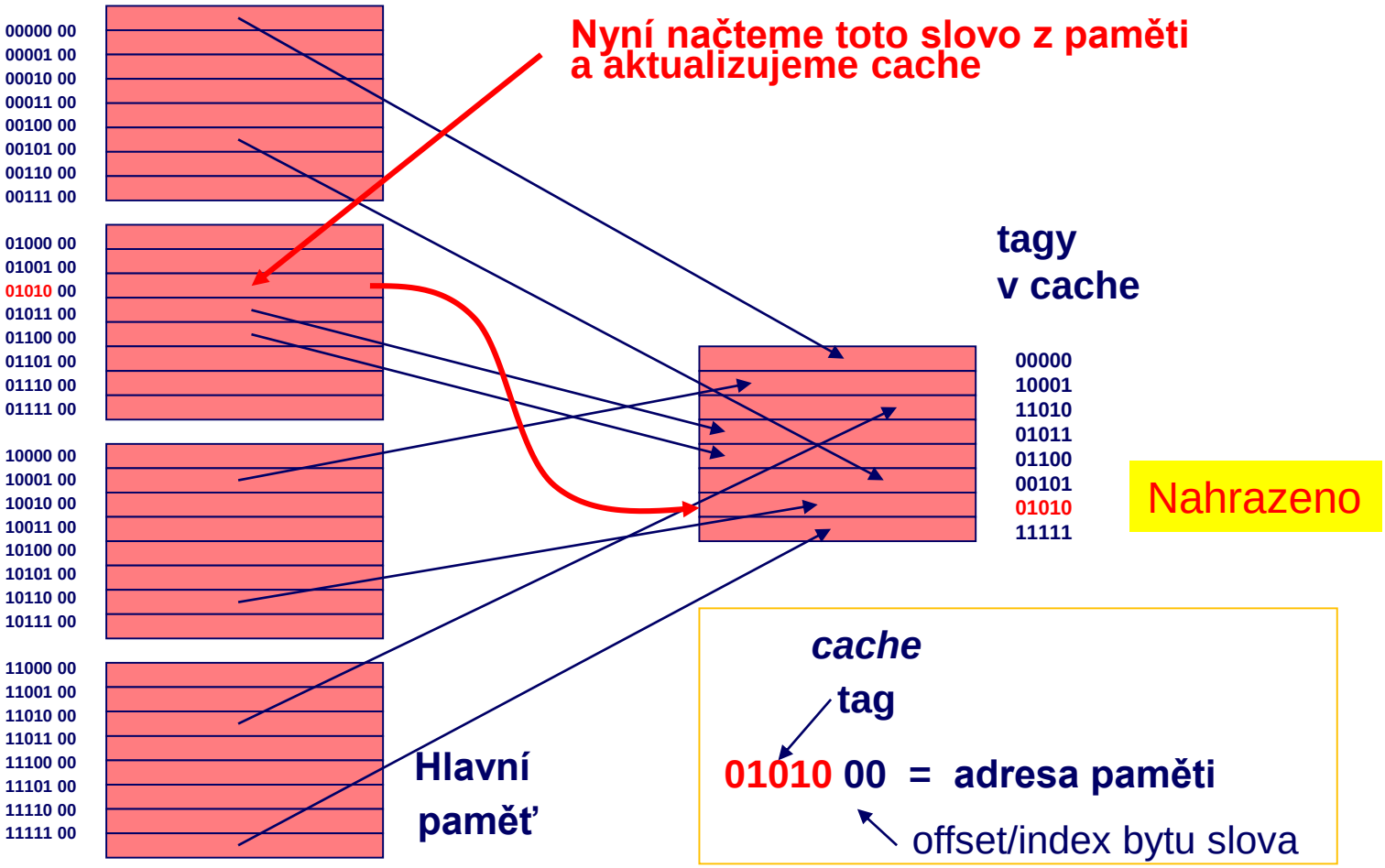
- **Náhodná** (Random) – vybere se libovolný blok. Snadné, ale hloupé.
- **LRU** (Least Recently Used) musíme znát informace o posledním použití tohoto bloku (náročnější).
- **LFU** (Least Frequently Used), u každého bloku se pamatujeme informace o tom, jak často byl požadován.
- **ARC** (Adaptive Replacement Cache), v níž se vhodným způsobem kombinuje strategie LRU a LFU.

- **Write-back**. Zároveň musíme obsah uvolňovaných řádek cache do hlavní paměti zapsat (D bity označené řádky). Zajištěno automaticky.

Fully-Associative Cache

Paměť o délce 128 bytů a čtená/zapisovaná jako 4-bytová slova

Cache má 8 bloků a každý blok obsahuje 1 čtyřbytové slovo

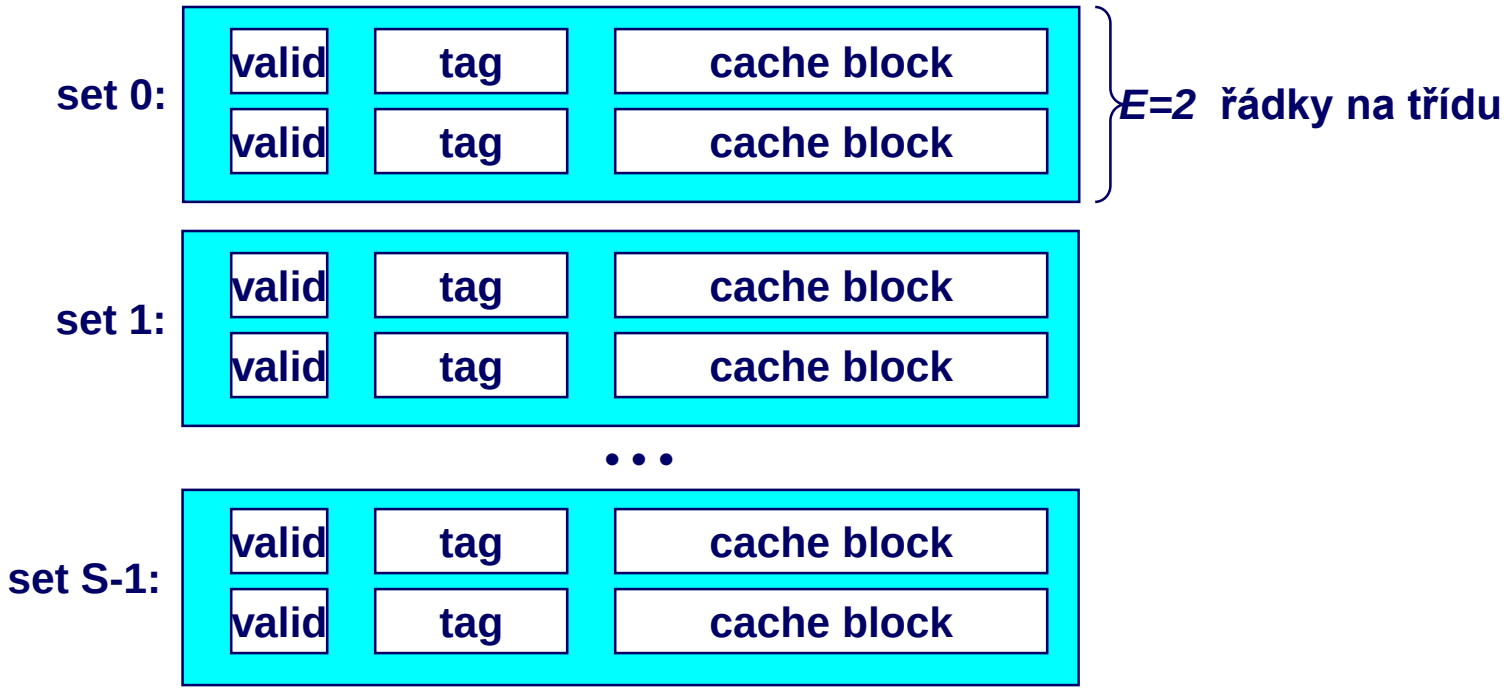


Diskuze k plně asociativní cache

- Šířka pole Tag odpovídá počtu bitů adresy minus délka pole indexu bytu v jednom slově, ta je 0 u 8-bitového procesoru, 1 u 16-bitového, 2 u 32-bitového, 3 u 64-bitového. Má-li blok více slov, odečtou se ještě bity offsetu do bloku.
- Každý blok (řádek) cache má samostatný komparátor shodnosti bitů dvou bitových řetězců o šířce Tagu.
- Počet řádků cache určuje její kapacitu.
- Cache musí mít strategii uvolňování obsahu (migrace dat mezi hierarchickými úrovněmi) v případě vyčerpání její kapacity.
- Lze však implementovat jednodušší varianty.

Set-Associative Cache
s omezeným stupněm asociativity

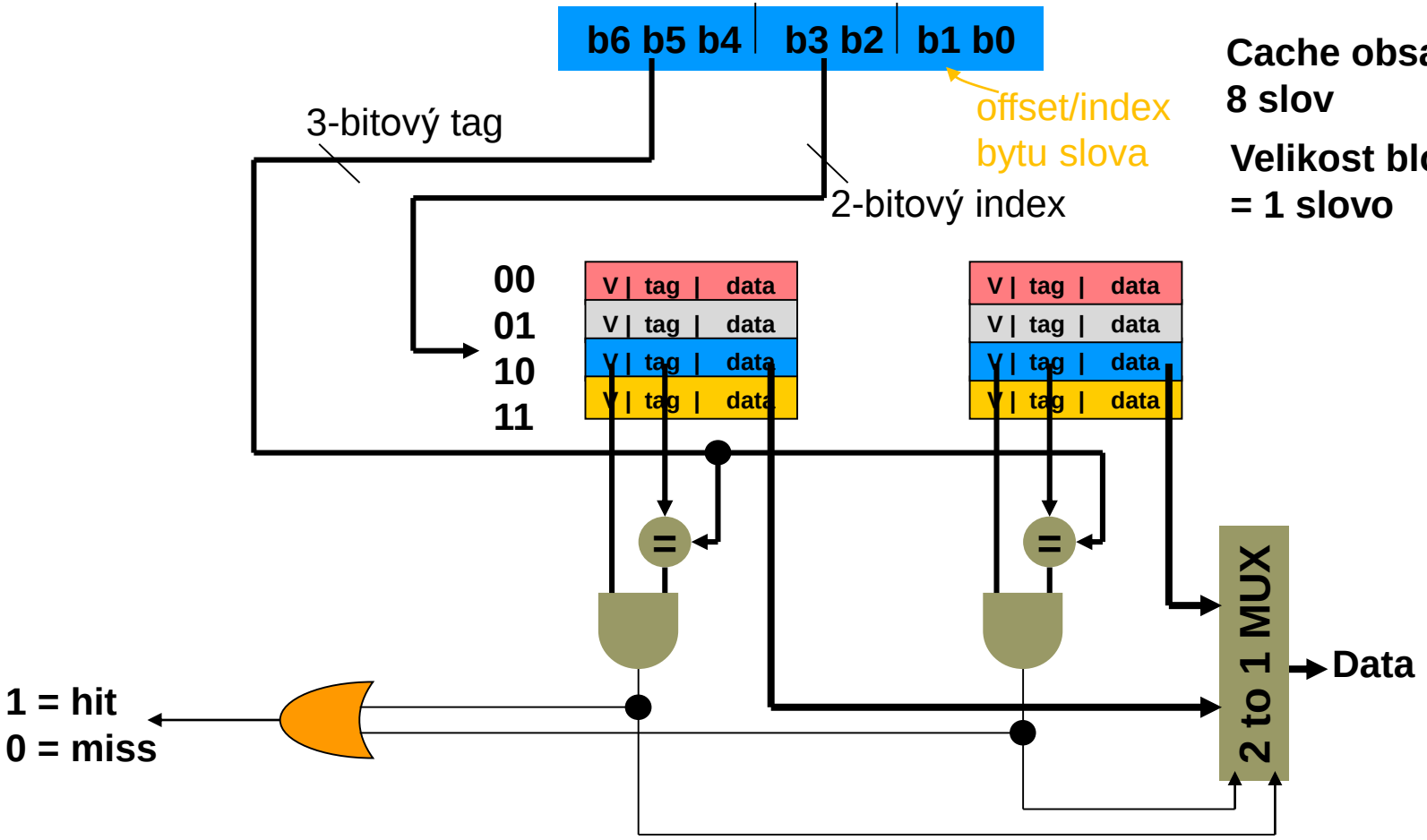
Více řádků (bloků) v jedné třídě (set)



Poznámka: Plně asociativní cache je speciálním případem B-cestně asociativní cache s jedním setem.

Two-Way Set-Associative Cache

Paměť o délce 128 bytů a čtená/zapisovaná jako 4-bytová slova



Konstrukce: Dvě přímo mapované cache se spojí asociativním výběrem dat

SP s omezeným stupněm asociativity N=2

Co přináší zvětšení stupně asociativity?

Capacity – C

Number of sets – S

Block size – b

Number of blocks – B

Degree of associativity – N

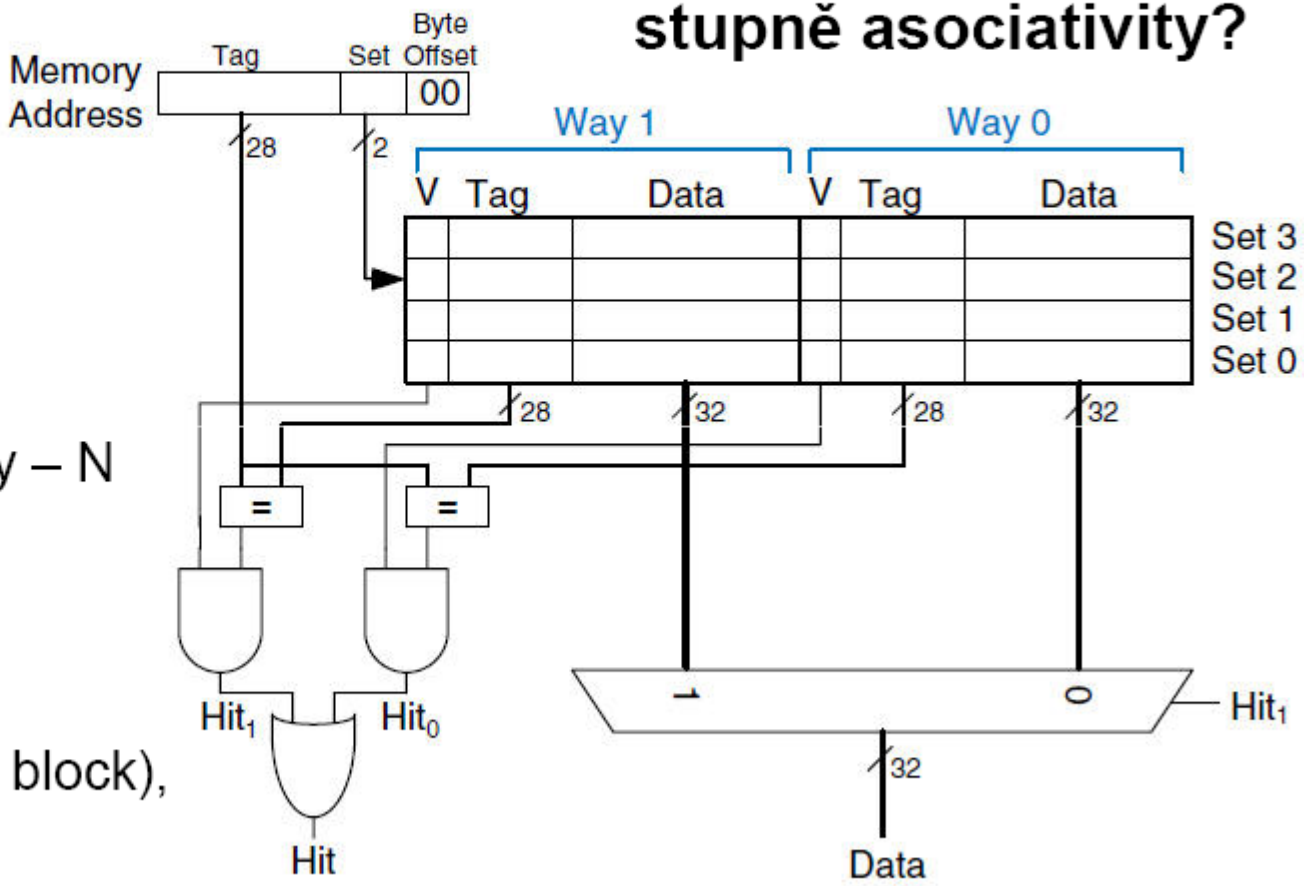
C = 8 (8 words),

S = 4,

b = 1 (one word in the block),

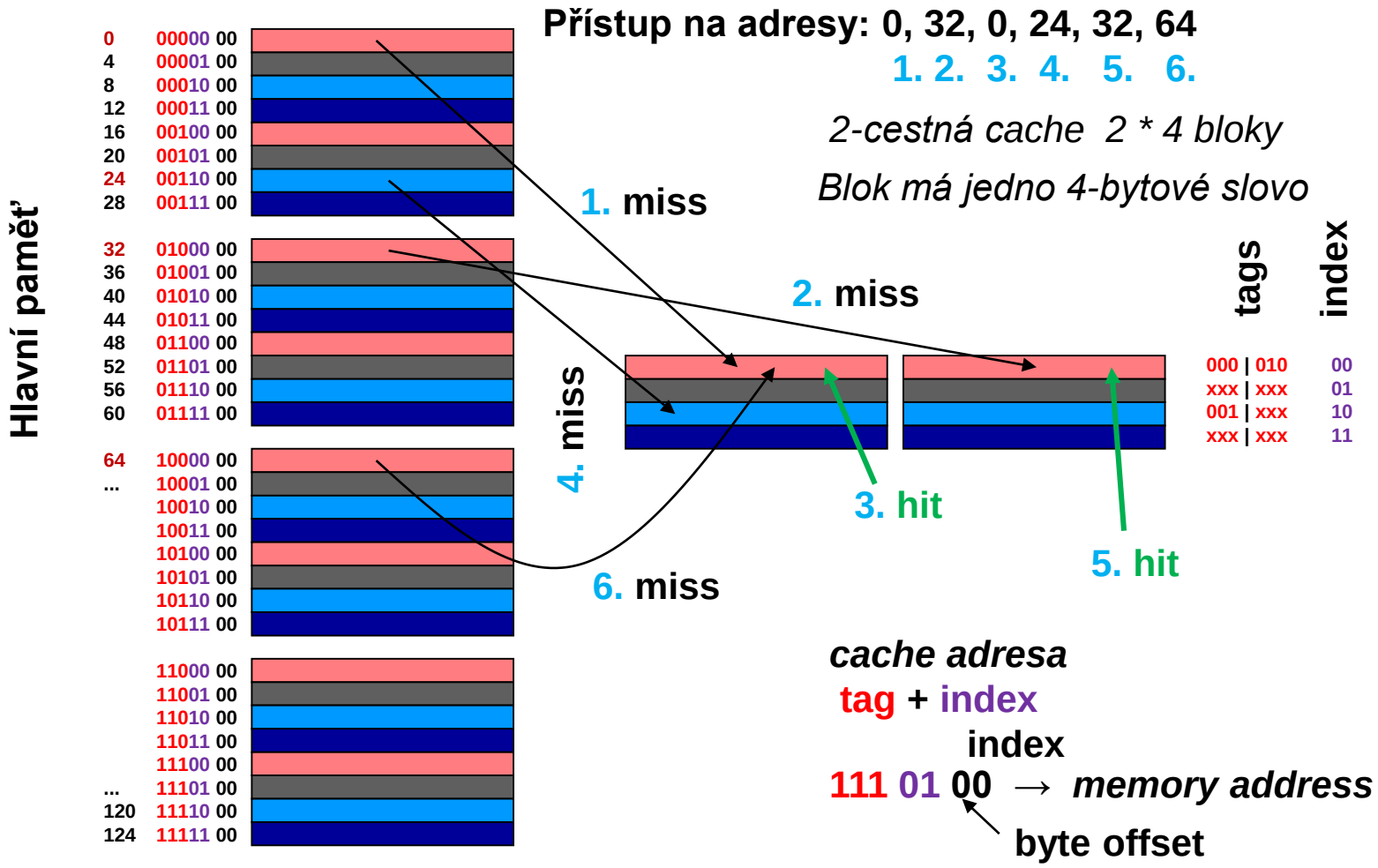
B = 8

N = 2



Miss Rate: Two-Way Set-Associative Cache

Paměť o délce 128 bytů a čtená/zapisovaná jako 4-bytová slova



LRU příklad ve 2 cestné-cache

Tag 0, 2, 0, 1, 4

0: miss, načteme set 0 (loc 0)

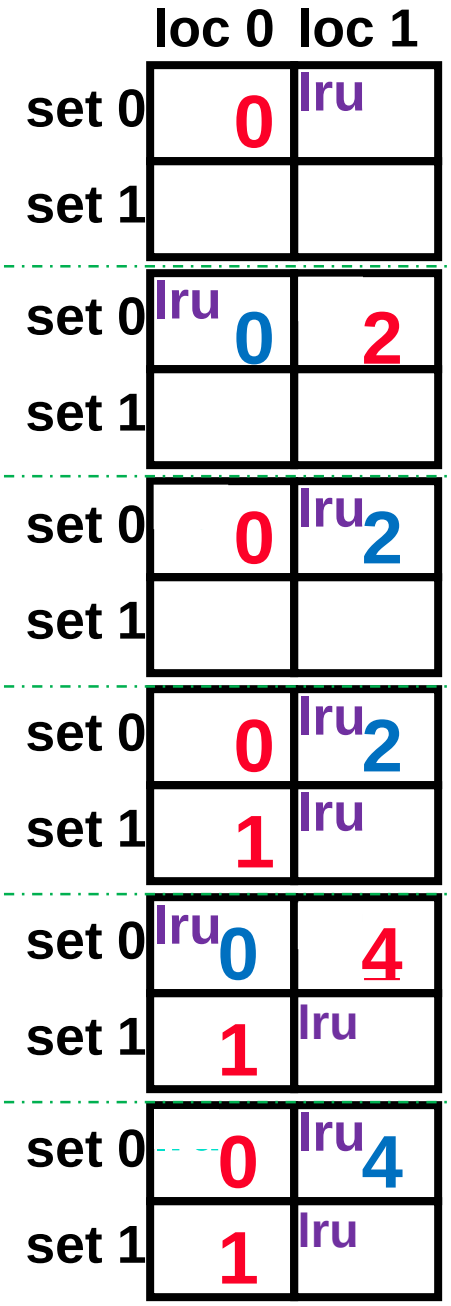
2: miss, načteme set 0 (loc 1)

0: hit

1: miss, načteme set 1 (loc 0)

4: miss, načteme set 0 (loc 1),
kde nahradíme předchozí tag 2 novým 4

0: hit

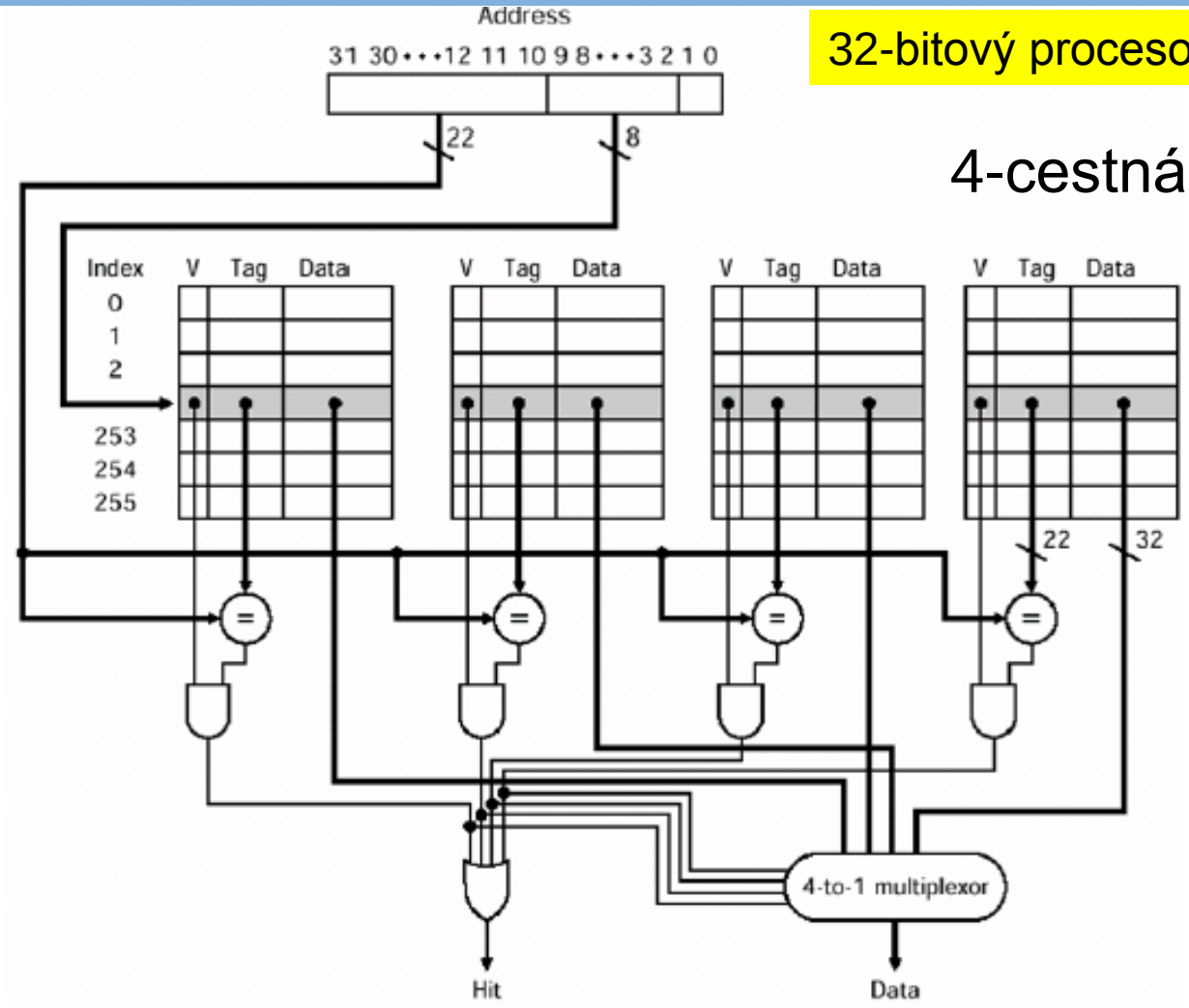


[Source: Garcia, UCB]

Diskuze: LRU implementace

- 2-cestná asociativní cache umožňuje snadné sledování LRU pomocí jednoho bitu.
- LRU ale vyžaduje složitější hardware a více času u 4 a vícecestných cache. Složitost LRU implementace drasticky roste s faktoriálem počtu cest, což zvyšuje nejen velikost paměti, ale i prodlužuje čas na její aktualizaci!
- 4-cestná cache má už 24 stavů and LRU vyžaduje přidat 5 bitů na každý set. 8-cestná cache potřebuje 40320 states a 16 bitů na set.
- LRU nahrazovací strategie se kvůli tomu aproximují jednoduššími mechanismy PLRU (pseudo-LRU) jako „**binary tree pseudo-LRU“**. Algoritmy PLRU již leží mimo rámec APO. Zájemci najdou podrobnosti například na: https://en.wikipedia.org/wiki/Pseudo-LRU , nebo v článku K. Kędzierski, M. Moreto, F. J. Cazorla and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, 2010, pp. 1-12.

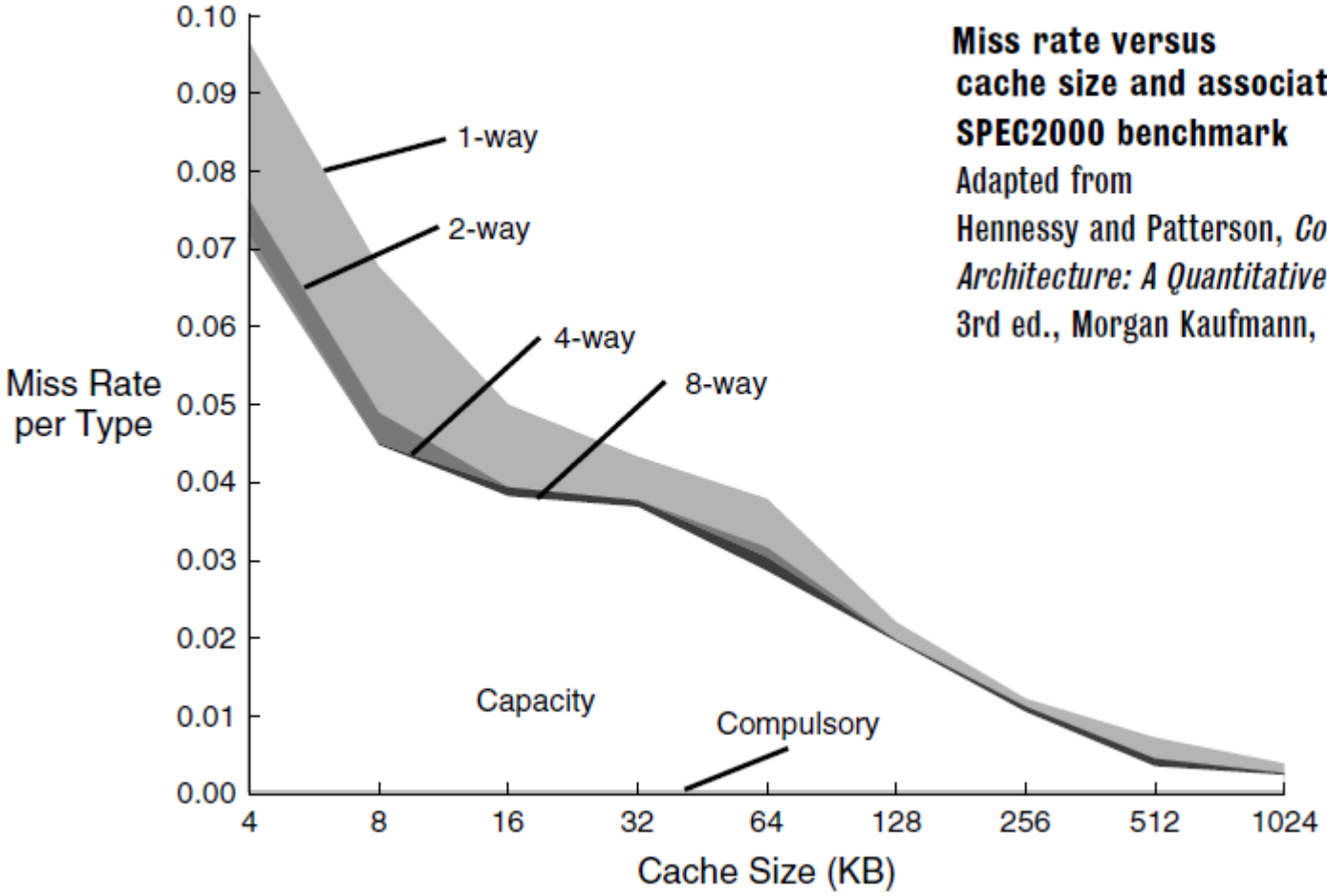
Cache s omezeným stupněm asociativity N=4



32-bitový procesor - 1 slovo = 4 byty

4-cestná, 4-way cache

Porovnání



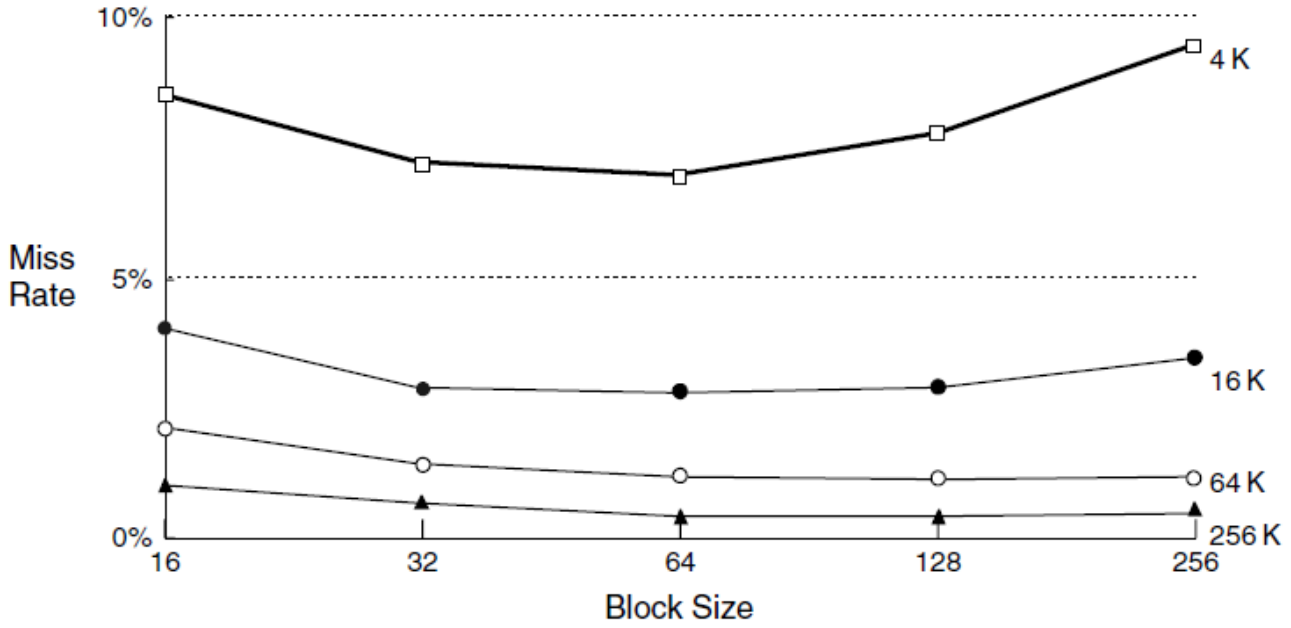
Miss rate versus cache size and associativity on SPEC2000 benchmark
Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Pamatujte: 1. miss rate není vlastností cache!

2. miss rate není vlastností programu!

Co přináší prostorová lokalita?

Miss rate můžeme redukovat zvýšením velikosti bloku – co znamená využití principu prostorové lokality. Na druhou stranu, zvětšování velikosti bloku při pevné velikosti cache znamená rovněž snižování počtu setů – to se projeví nárůstem konfliktů (nárůstem miss rate)…



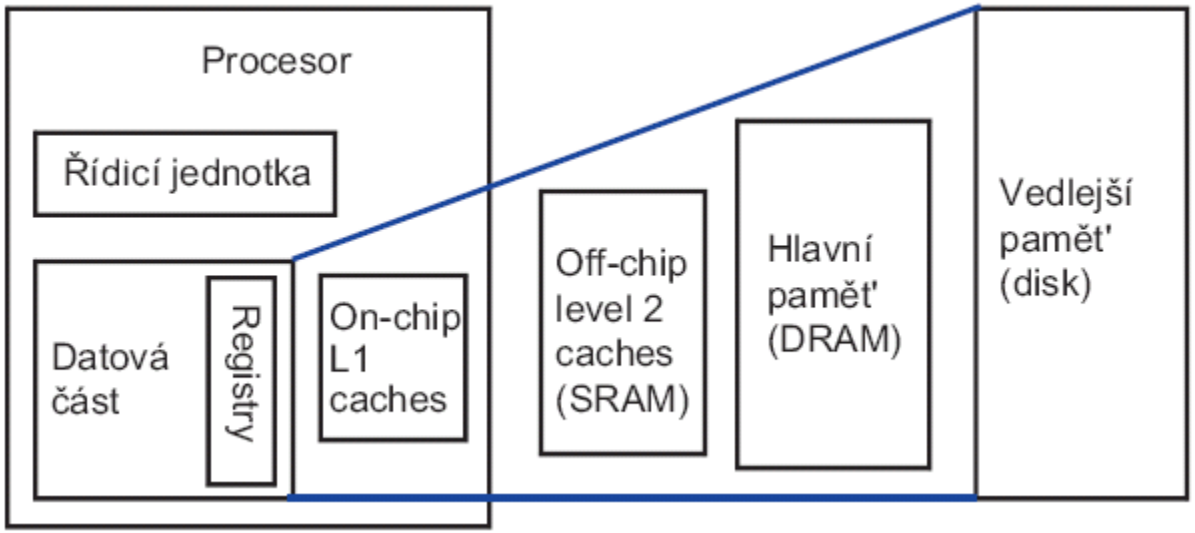
Miss rate versus block size and cache size on SPEC92 benchmark Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Trend - **Víceúrovňové SP**

- Primární SP je bezprostředně připojena k procesoru
  - Rychlá, malá. Nejdůležitější: minimální Hit Time, bývá 2 až 8-cestná
- L2 SP ošetřuje výpadky primární SP
  - Větší, pomalejší, ale stále rychlejší než hlavní paměť. Nejdůležitější: low Miss Rate
- Hlavní paměť ošetřuje výpadky L2
- Současné nejvýkonnější systémy mají i L3

	Typicky pro L1	**Typicky pro L2**
Počet bloků	250-2000	15 000-250 000
KB	16-64	2 000-3 000
Velikost bloku v B	16-64	64-128
Miss penalty (v hod)	10-25	100-1 000
Miss rates	2-5%	0,1-2%

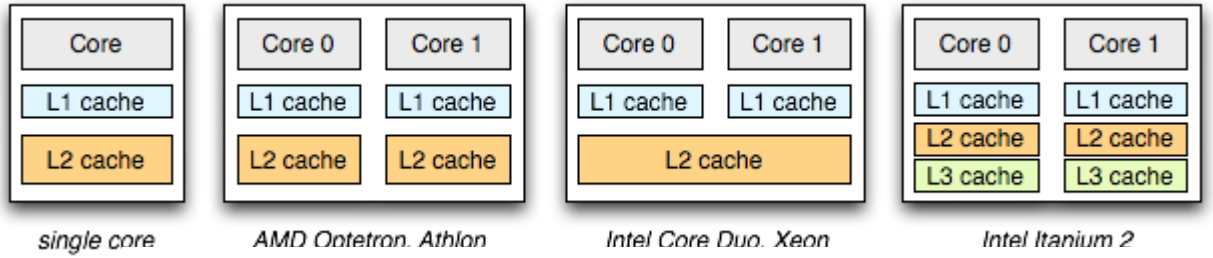
Paměťová hierarchie



Rychlost	(2008)	0,5-2,5 ns	50 – 70 ns	5 – 20 ms
	cena/GB	$2k-5k	$20 – 75	$0,2 - 2

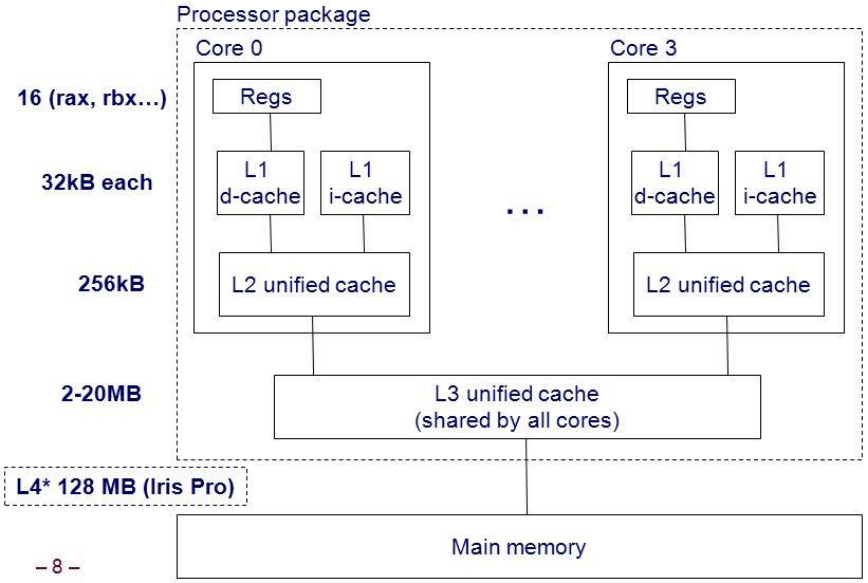
- Jedna a tatáž informace se může objevit na více místech hierarchické paměti!

Organizace u některých procesorů Intel



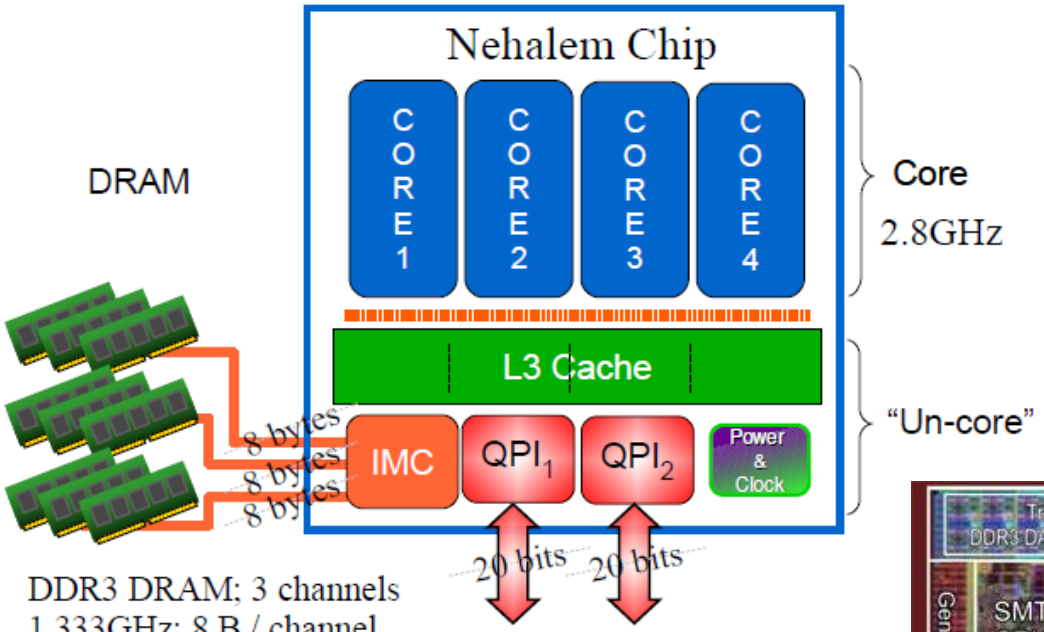
single core *AMD Opteron, Athlon* *Intel Core Duo, Xeon* *Intel Itanium 2*

Intel Core i7 cache hierarchy (2014)

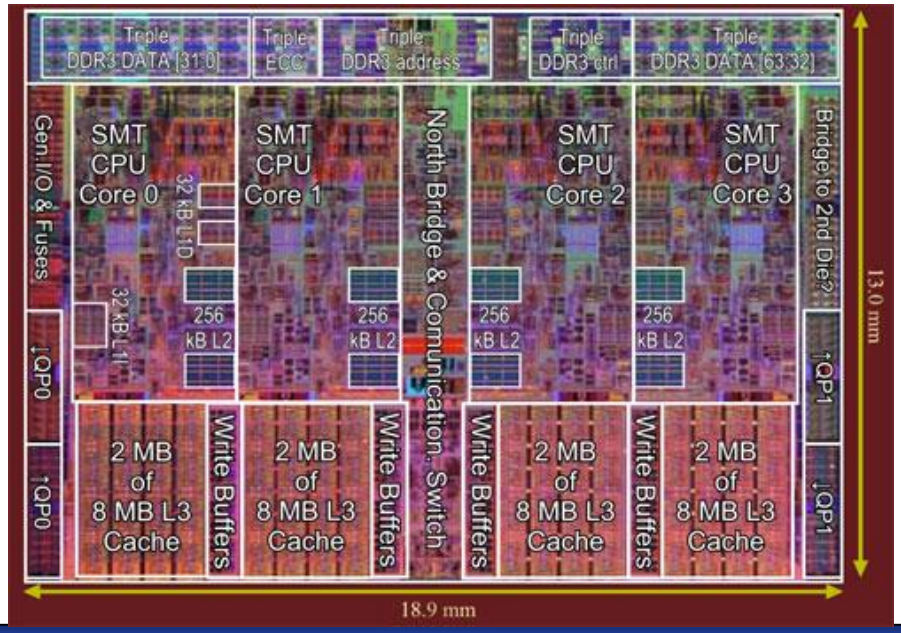


Zdroj: Intel

Příklad procesoru
Harvardská architektura - Intel Nehalem (2008)



- IMC: integrovaný řadič třech DDR3 paměťových bank (memory controller)
- QPI: Quick-Path Interconnect ports
- Podívejte se na velkosti jednotlivých cache v hardwaru!!!



Jak a kde pak ale hledanou informaci najdeme?

- Podle adresy a případně dalších informací (např. o platnosti).
- Hledat začneme v paměti nejvyšší hierarchické úrovně (nejblíže procesoru).
- Požadavky:
  - Paměťová konzistence.
- Prostředky:
  - Virtualizace adresy,
  - Mechanizmy uvolňování místa a migrace informace mezi paměťovými úrovněmi.
  - Hit, miss.

Pochopili jste tuto přednášku?

- Pokud ano, tak již si uvědomujete, že využití 2 principů (principy časové a prostorové lokality) může vést k významnému urychlení Vašeho programu, a to efektivním využitím cache…!!!
- Existují HW a SW (kompilátorem) techniky, které na základě těchto principů optimalizují práci z cache. HW techniky z pohledu programátora ovlivnit nemůžete. U kompilátoru můžete nastavit stupeň optimalizace…
  (Rozbor HW technik je mimo rozsah APO, patří do A4M36PAP.)
- Nicméně, i sebelepší kompilátor pouze kompiluje co napsal programátor. Výběr algoritmů, uložení datových struktur v paměti a manipulace s nimi – to vše je určeno programátorem. Proto stále je v rukou programátora „nejvíc“ práce a od něj do značné míry závisí jak bude program „rychlý“.

Pochopili jste tuto přednášku?

- Instrukční cache – pokročilé
  - Vhodným uspořádáním kódu, příp. přeuspořádáním funkcí v paměti
  - Profilace
- Datová cache – snadné
  - Vhodným uspořádáním dat – data, která plánujeme používat sekvenčně, řadit sekvenčně v paměti, apod.
  - Sloučení polí nebo souvisejících datových struktur
  - Práce po blocích dat – co nejdřív používat již použité
  - iterace ve vnořených cyklech – viz úvodní příklad – s cílem procházet paměť sekvenčně a ne po skocích
  - sloučení dvou smyček do jedné – Loop fusion
  - atd.

Plýtvání pamětí

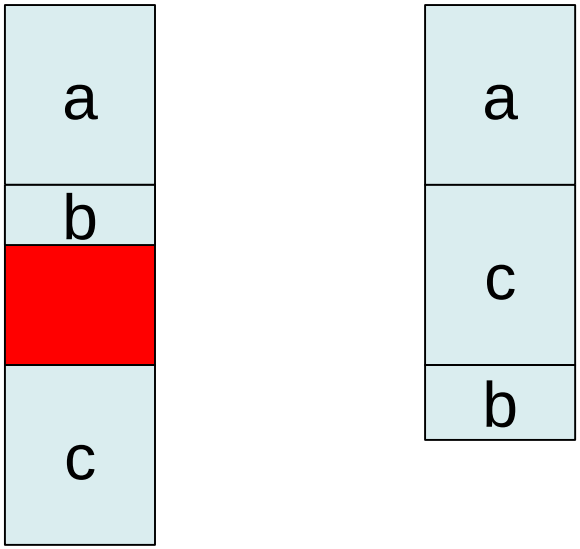
- Spatřujete rozdíl v těchto deklaracích?

- **/* Před optimalizací */**

```
int a=0;
char b='a';
int c=1;
```

- **/* Po optimalizaci */**

```
int a=0;
int c=1;
char b='a';
```



Pochopili jste tuto přednášku?

- Prostorová lokalita – konflikty v cache:

```
/* Před optimalizací */
int values[SIZE];
int keys[SIZE];
int scores[SIZE];

/* Po optimalizaci */
struct item{
 int value;
 int key;
 int score;
};
struct item records[SIZE];
```

Předpokládejme 2-cestně asociativní cache…

```
for(i=0; i<SIZE; i++)
  for(j=0; j<SIZE; j++)
    …
```

Data, k nimž přistupujete krátce za sebou, uložte vedle sebe (seskupte je).

Pochopili jste tuto přednášku?

- Časová lokalita:

```
/* Před optimalizací */
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      d[i][j] = a[i][j] - c[i][j];
```

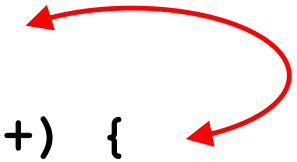
```
/* Po optimalizaci */
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      { a[i][j] = b[i][j] * c[i][j];
      d[i][j] = a[i][j] - c[i][j];}
```

Nejedná se jenom o úsporu instrukcí, ale také efektivněji používáme cache…

Pochopili jste tuto přednášku?

- Dalším příkladem je násobení matic

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++) {
    tmp = 0;
    for (k=0; k < N; k++)
       tmp += y[i][k]*z[k][j];
    x[i][j] = tmp;
  }
```



Pomůže nám nějak když prohodíme tyto dva řádky? Bude program ekvivalentní?

(Viz příklad z předchozí přednášky…)

Pochopili jste tuto přednášku?

- Násobení matic se lépe provede blokovým násobením

  Idea: Výpočet se rozdělí na submatice BxB, které se vejdou do cache.. => eliminace „capacity misses“

```
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i++)
      for (j = jj; j < min(jj+B-1,N); j++) {
      tmp = 0;
        for (k = kk; k < min(kk+B-1,N); k++)
           tmp += y[i][k]*z[k][j];
        x[i][j] = x[i][j] + tmp;
      }
```

Ke čtení: http://suif.stanford.edu/papers/lam-asplos91.pdf

Pochopili jste tuto přednášku?

- Hledání prvočísel - Eratostenovo sito:

```
/*Před optimalizací*/
boolean array[max];
for(i=2;i<max;i++) {
  array = 1;
}
for(i=2;i<max;i++)
  if(array[i])
    for(j=2*i;j<max;j+=i)
      array[j] = 0;
    /* zápis 0 - řádek se načte do cache a nastaví se v něm
       dirty bit pro uložení do paměti */
```

Pochopili jste tuto přednášku?

- Hledání prvočísel - Eratostenovo sito:

```
/*Po optimalizaci*/
boolean array[max];
for(i=2;i<max;i++) {
  array = 1;
}
for(i=2;i<max;i++)
  if(array[i])
    for(j=2*i;j<max;j+=i)
      if(array[j]!=0)
        array[j] = 0; /* dirty bit jen někdy */
```

- Redukujte neužitečné zápisy (redukce zápisů do paměti – (dirty cache lines musejí být vždy zapsány do paměti před novým naplněním)

Pochopili jste tuto přednášku?

Někdy se hodí přemýšlet také o zarovnání dat v paměti (aligning), protože nezarovnaný prvek může zabrat více slov v cache. V C lze zkontrolovat, zda kompilátor zarovnává double čísla dle slov v cache, a pokud ne:

- *přidělíme, kolik potřebujeme + jeden prvek navíc*
- *operací AND zarovnáme adresu počátku dat například takto:*

```
const int SIZE_OF_ARRAY = 64;   const int SIZED = sizeof(double); // 8
// trik pro debugger - definice pole ve struktuře nám umožní vidět celé pole
struct MyArr { double Item[SIZE_OF_ARRAY];  } *myArr;
double *p = (double*) malloc( SIZED * (SIZE_OF_ARRAY+1) );
// nastavíme spodní bity na 0   ( -SIZED = 0b1111 ... 1111 1000 )
myArr = (MyArr*) ( (long) ( (unsigned char *)p + SIZED-1 ) & -SIZED );
int i=0; double d=1.0/2; double * pd = myArr->Item;  // příklad užití pole
do { *pd++ = (d *= 2); } while (++i < SIZE_OF_ARRAY);
// pozn1. do...while cyklus se vykoná rychleji než while...do či for, viz další přednášky
// pozn2. něco podobného později využijeme i k zarovnání na stránky virtuální paměti
```