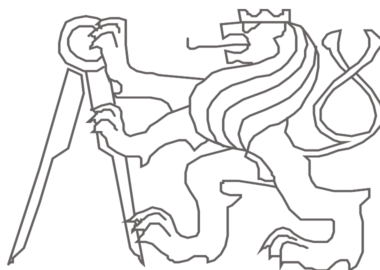
Paměť Cache

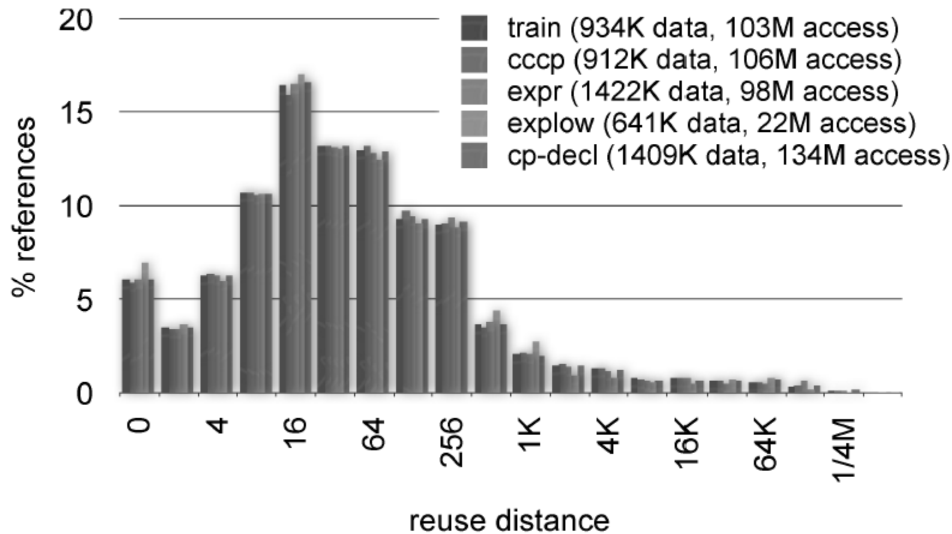
Snímky verze 4.0

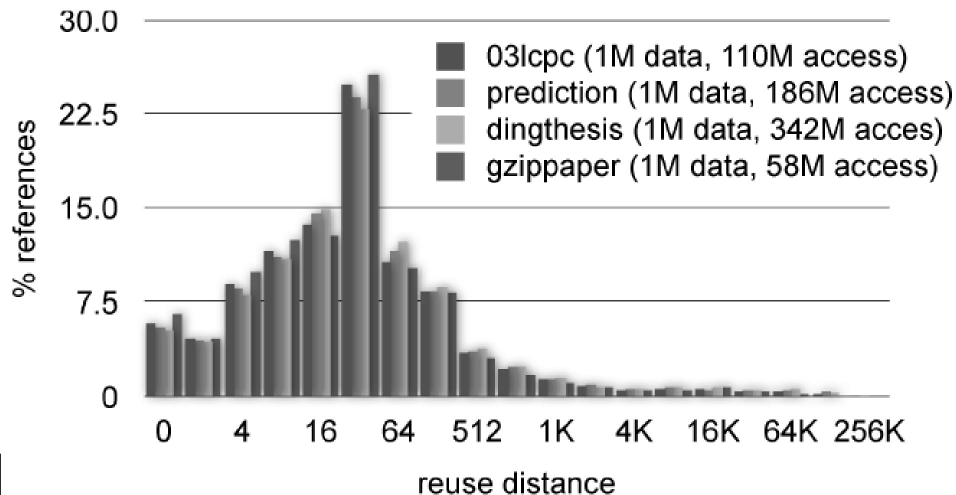
Richard Šusta, Pavel Píša



České vysoké učení technické, Fakulta elektrotechnická

Memory Reuse Distance in Some Programs





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 pages*

Paměťová hierarchie – základní principy

- Programy/procesy 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 brzy zapotřebí také.
  - Příklad: sekvenční přístup ke kódu (paměť programu), datová pole (paměť dat).

Značka samostudia

Značka bude 3S signalizovat:
"Skrytý Snímek pro Samostudium",
který se na přednášce zpravidla nepromítal.
Buď shrnuje výklad, nebo ho rozšiřuje.

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)

Cache

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

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

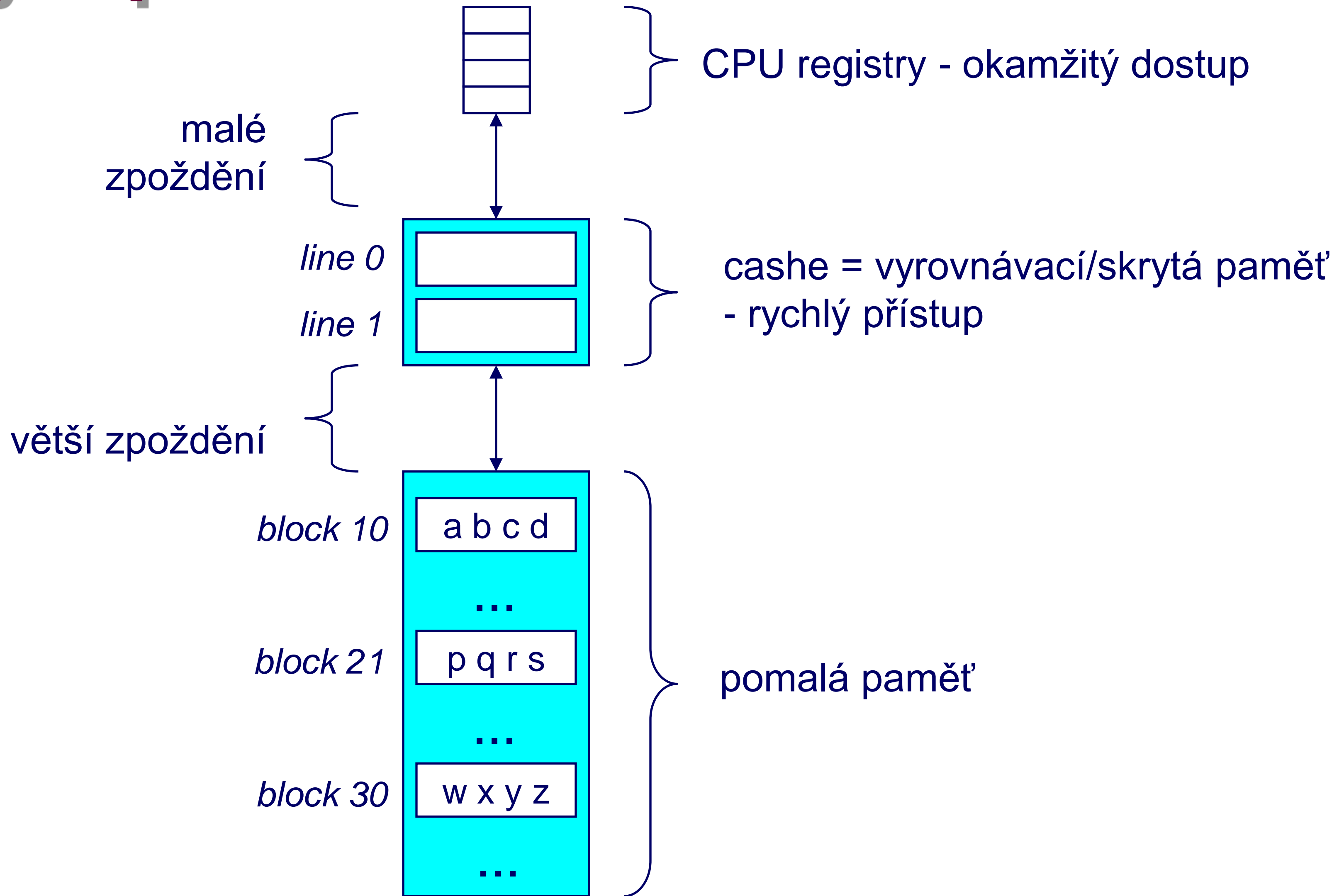
from French Canadian trappers' slang word derived from French *cacher* = to hide, conceal = hiding place; especially used by settlers, explorers, or campers for concealing and preserving provisions or implements.

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

Cache, výslovnost "keš"

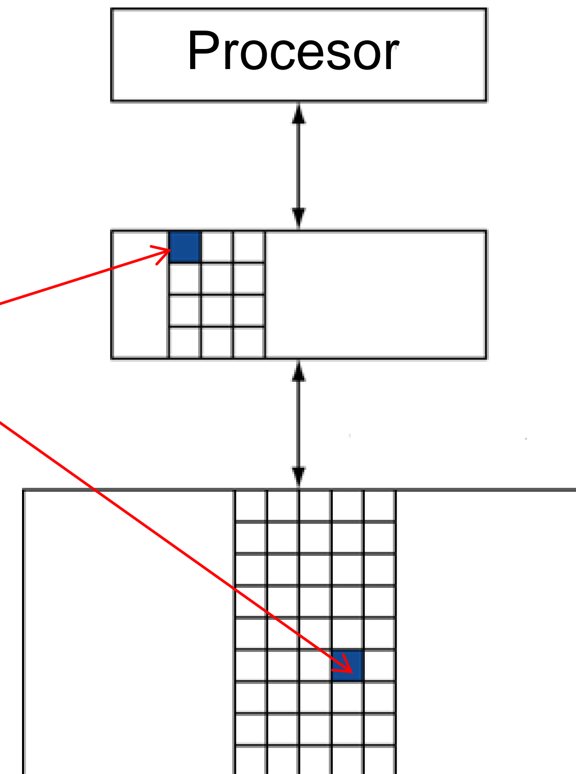
- **a computer memory** with very short access time used for storage of frequently or recently used instructions or data— called also *cache memory.*
- CZ: **skrytá paměť** - vyrovnávací paměť používaná ve výpočetní technice.
- Řadí se mezi dva subsystémy s různou rychlostí, kde vyrovnává odlišné doby přístupu k informacím.
- **Urychlí** přístup k opakovaně používaným datům, a to uchováváním jejich kopií, případně zápis dat jeho zpožděním.

Skrytá paměť - cashe



Terminologie kolem skryté paměti

- **Cache hit** pojmenování situace, kdy požadovaná hodnota ve skryté paměti (cache) **je**.
- **Cache miss**, opak. **Není** tam.
- **Cache line** nebo **Cache block** – základní kopírovatelná jednotka mezi hierarchickými úrovněmi.
- V praxi se velikost Cache Line pohybuje od 8B do 1KB, typicky 64B.



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,
  - Přímo mapované, nebo
  - S omezeným stupněm asociativity N=2 (2-cestná, 2-way cache).

Terminologie kolem skryté paměti II.

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

  $\mathbf{AMAT} = \mathrm{HitTime} + \mathrm{MissRate} \times \mathrm{MissPenalty}$

Obecná organizace Cashe

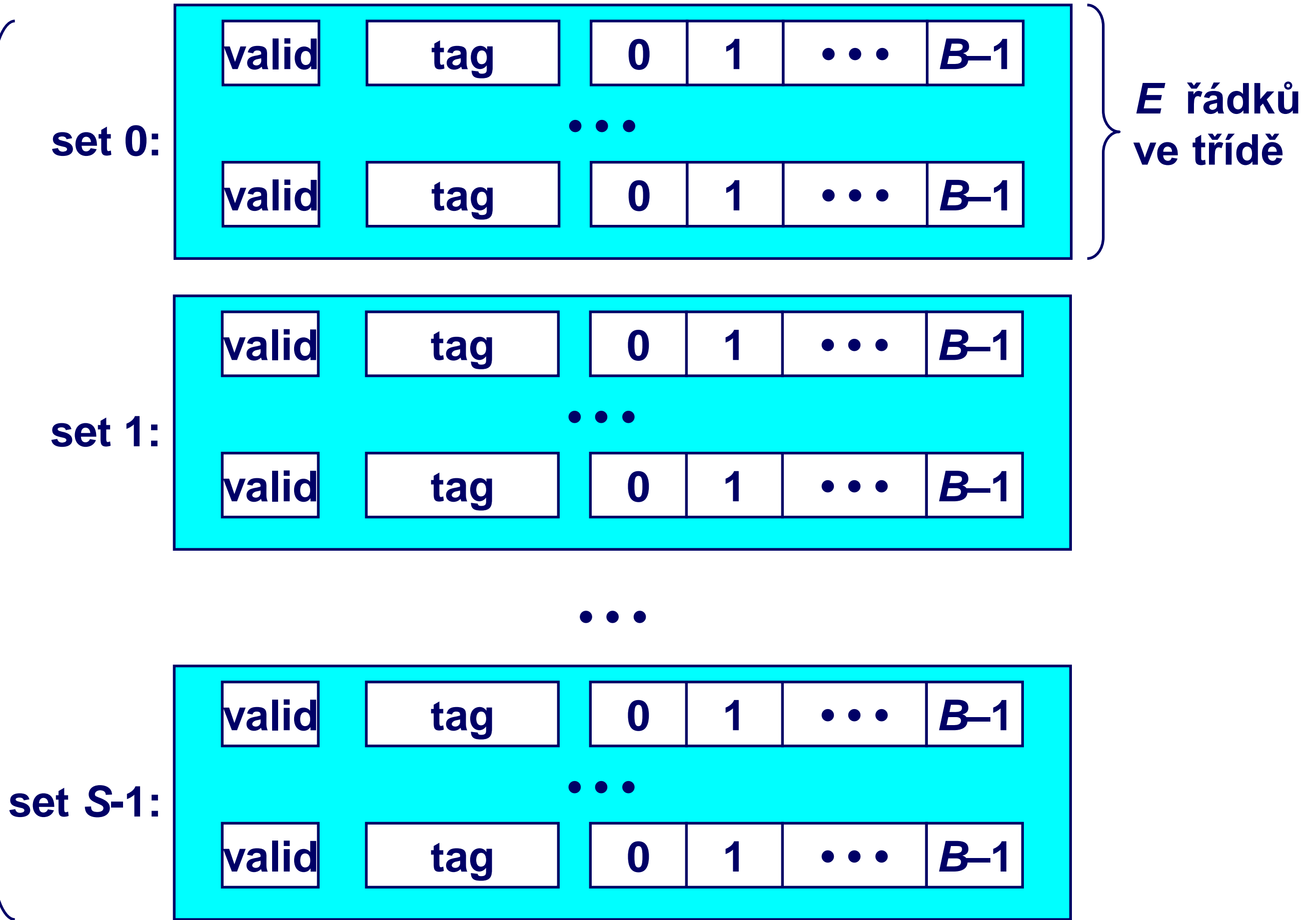
Cache je pole
S tříd (sets)

Třída obsahuje
E řádků (blocks)

Každý řádek
má B slov/sloupců

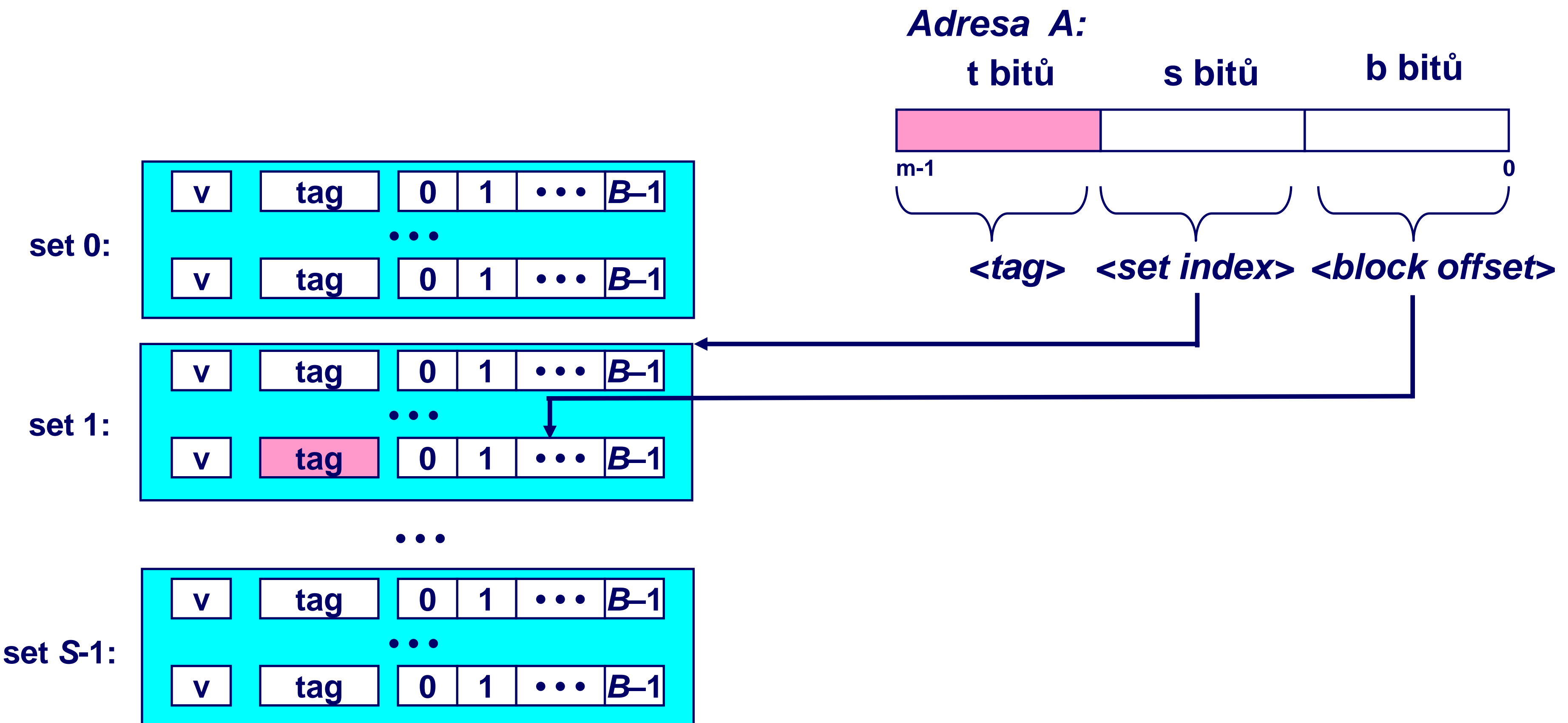
Set # ≡ hash kód

Tag ≡ hash klíč

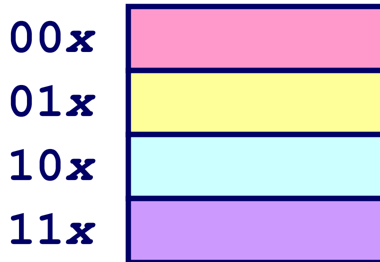


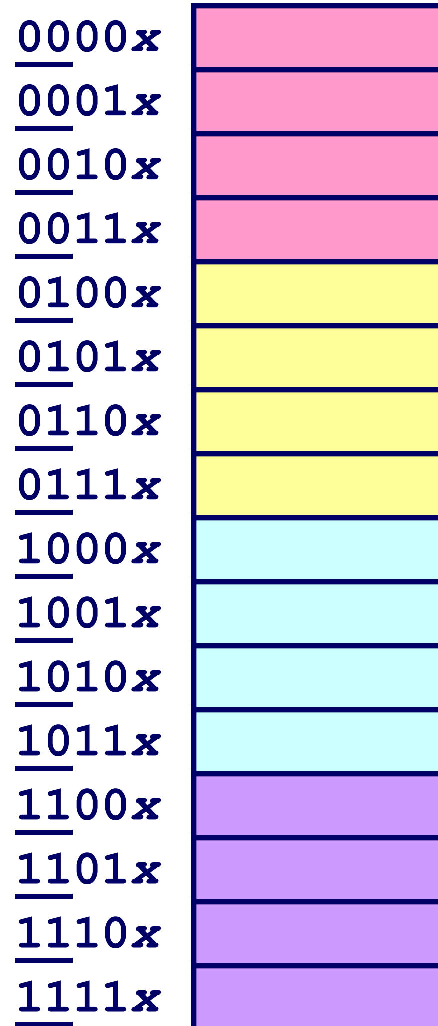
Velikost: $C = B \times E \times S$ slov

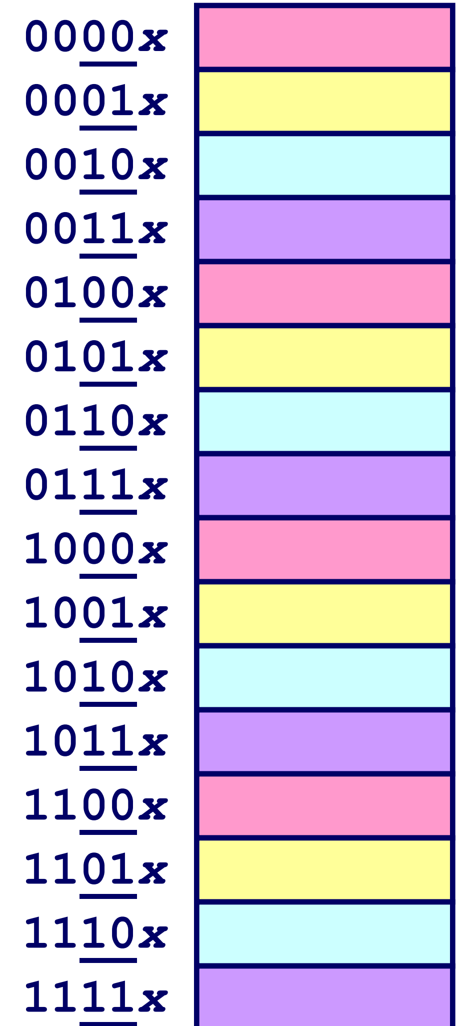
Adresování Cashe



Proč se používají prostřední bity?

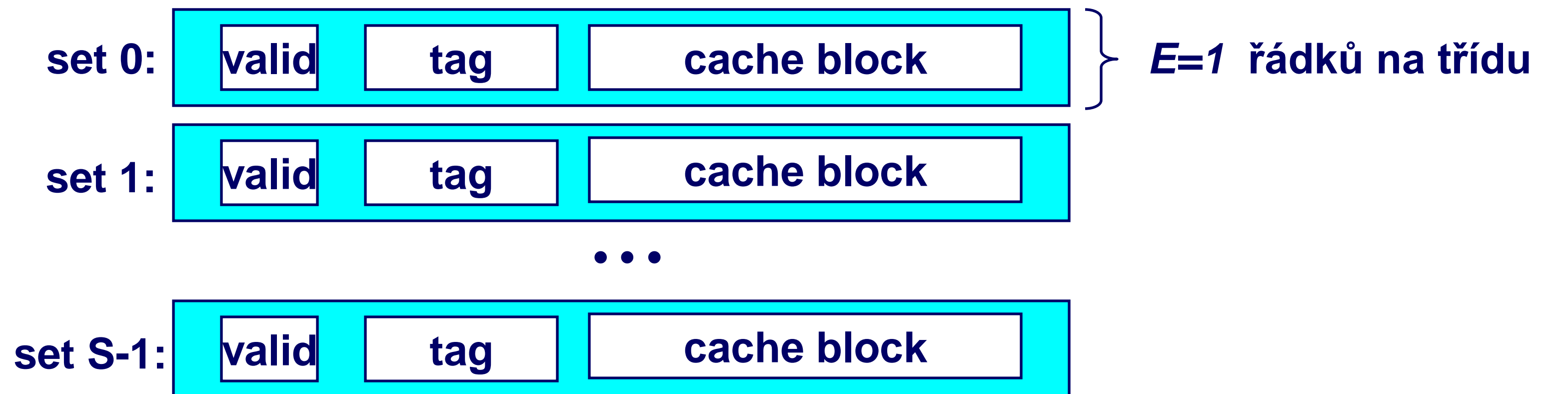






Direct-Mapped Cache

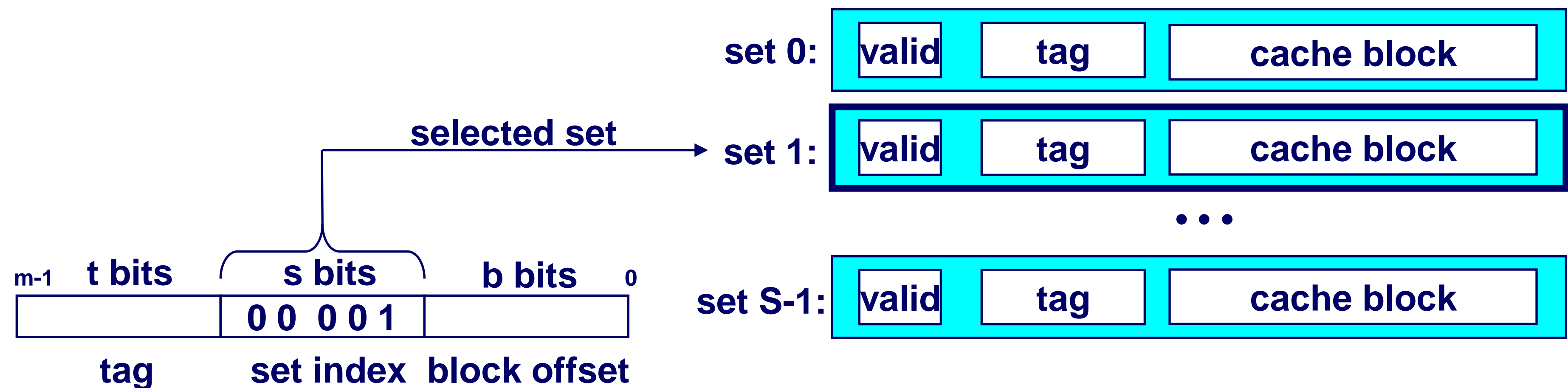
1 řádek (block) na 1 třídu (set)



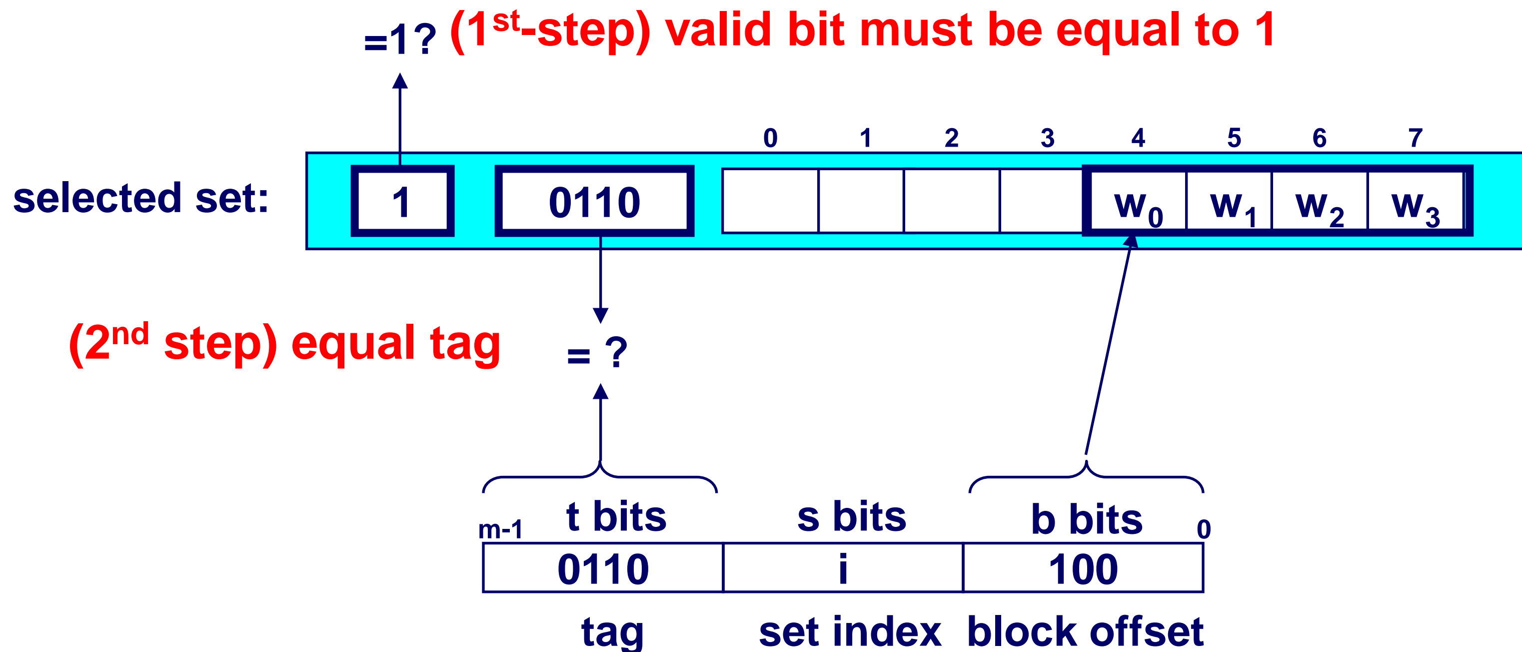
Accessing Direct-Mapped Caches

Set selection

- index of set determines row, because 1 set has only 1 row!



Accessing Direct-Mapped Caches



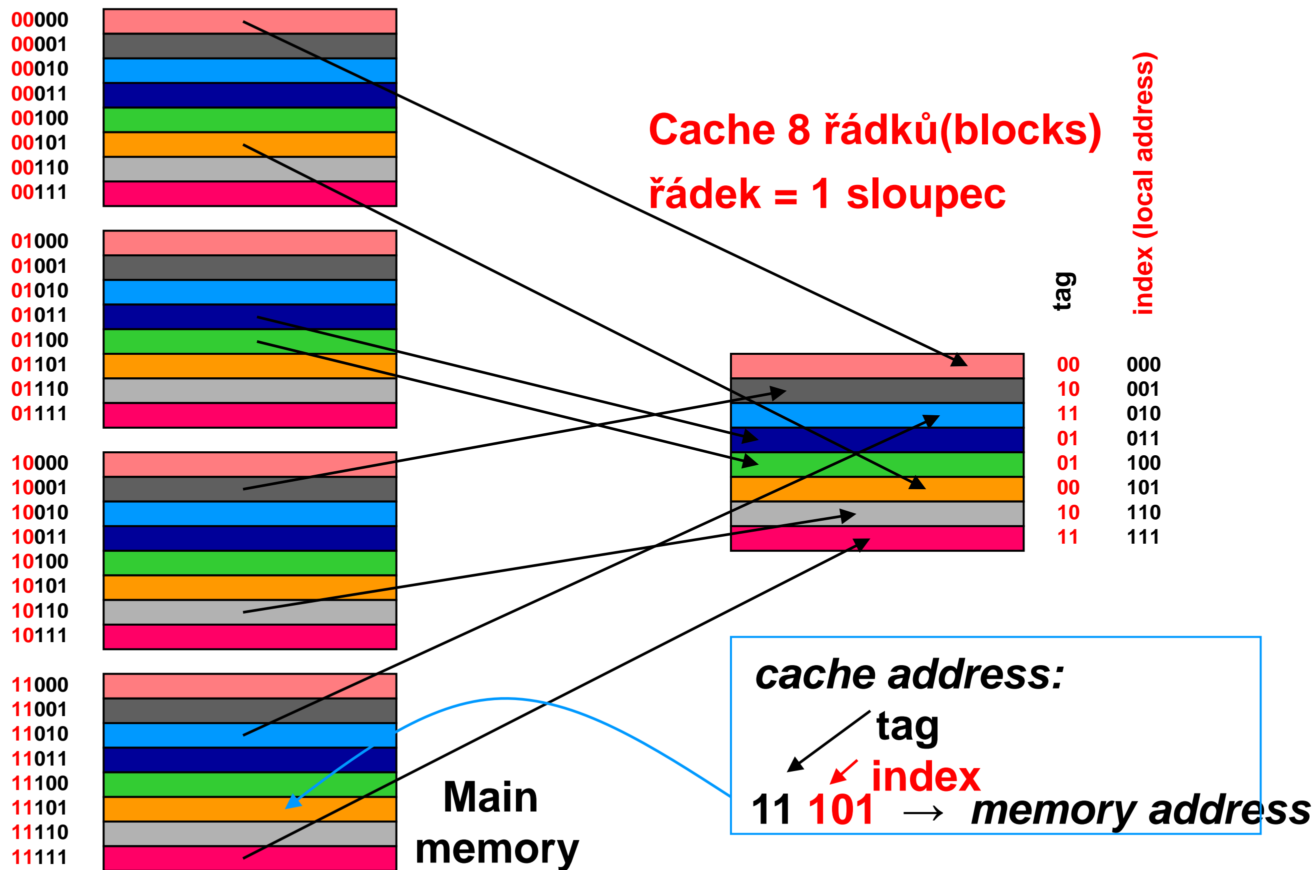
(3rd step) If (1) and (2) are satisfied,

then cache *hit* and select starting byte by block offset

else cache miss

Example Direct-Mapped Cache 8bit Processor

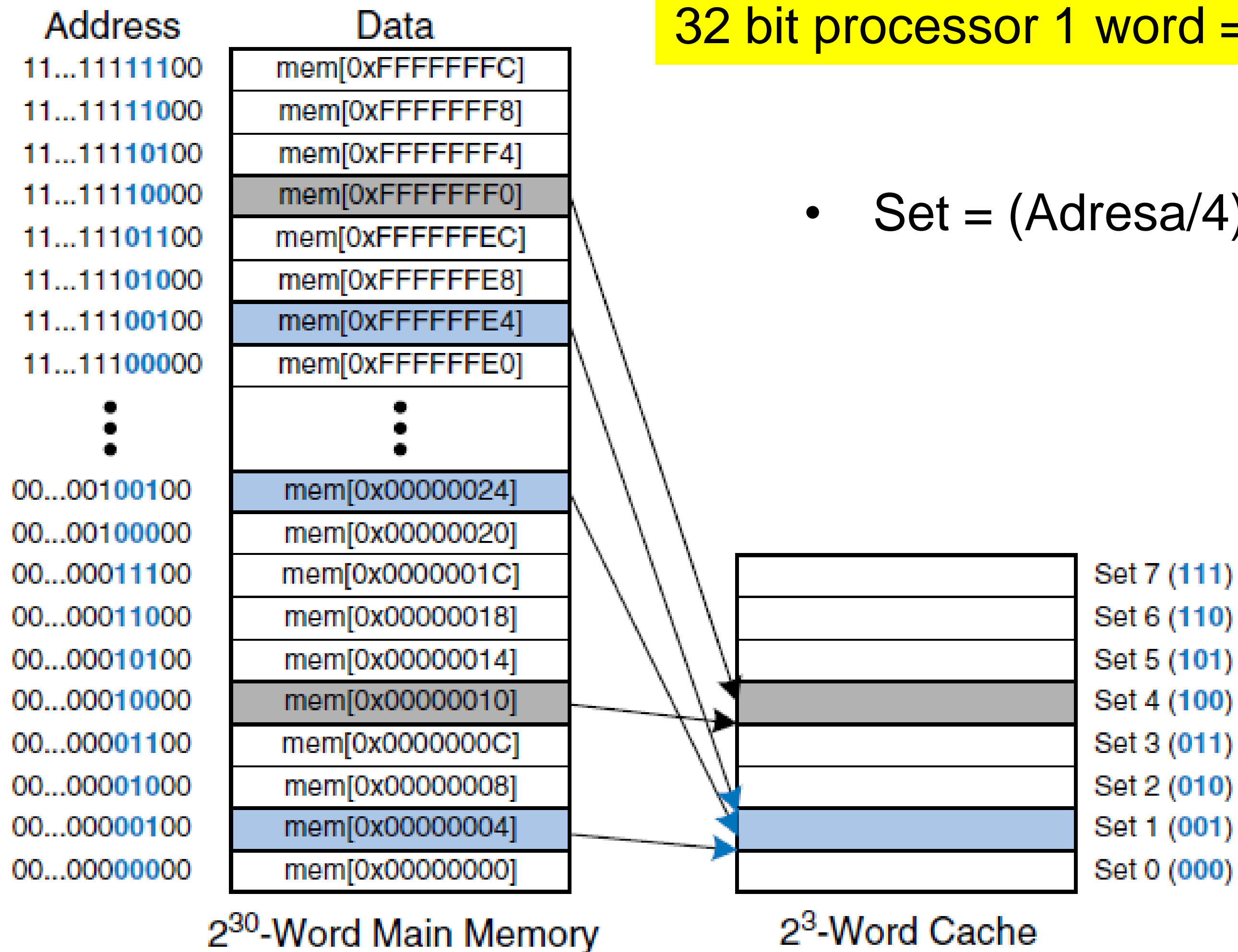
8bit processor with memory 32 bytes - 1 word = 1 byte



Example: Direct Mapped Cache

32 bit processor 1 word = 4 bytes

- Set = (Adresa/4) mod 8



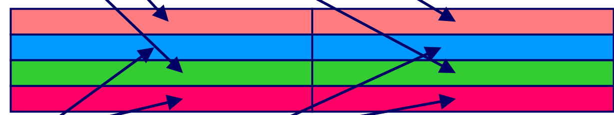
Direct-Mapped Cache - Block size 2 words

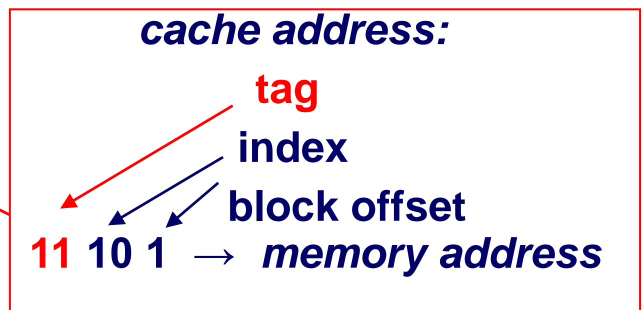
8bit processor with memory 32 bytes - 1 word = 1 byte

32-word word-addressable memory

Main memory

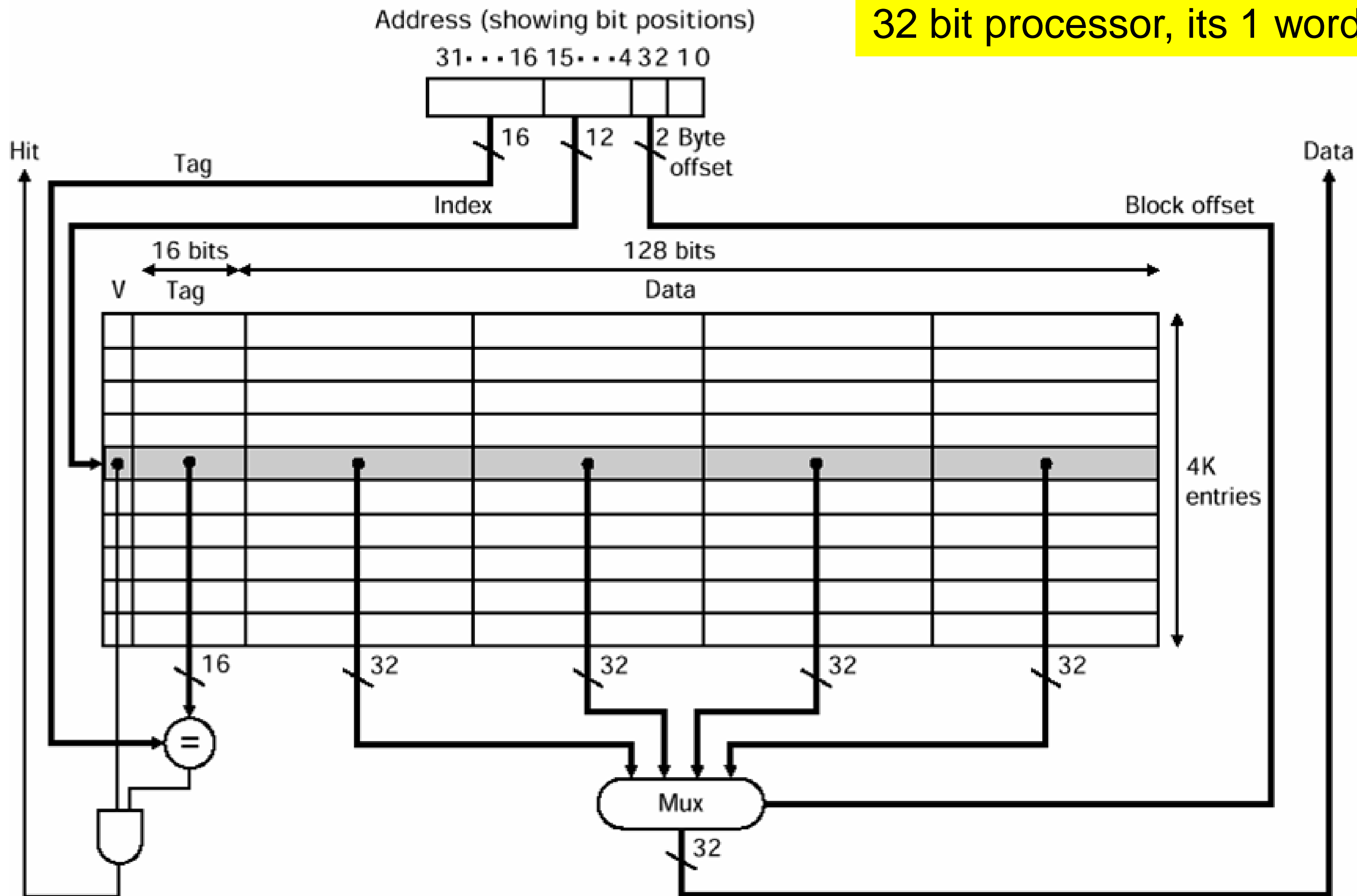
Cache 4 blocks
Block size = 2 word





Direct mapped cache implementation

32 bit processor, its 1 word = 4 bytes



* Example: Direct Mapped Cache

The following slides were created by

PSOE Dan Garcia

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

and they were only slightly adapted.

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 values in main memory

32 bit processor, its 1 word = 4 bytes

Memory

Address (hex)	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

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)

```
000000000000000000 0000000001 0100
000000000000000000 0000000001 1100
000000000000000000 0000000011 0100
000000000000000010 0000000001 0100
```

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)

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
•••			•••			
1022	0					
1023	0					

1. Read 0x00000014

000000000000000000 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So we read block 1 (0000000001)

000000000000000000 **0000000001** 0100

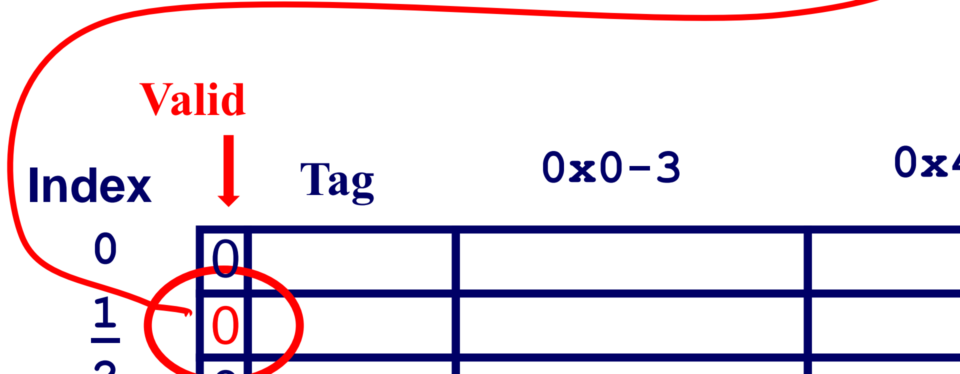
Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

No valid data !

0000000000000000000 0000000001 0100

Tag field — Index field — Offset



Valid

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

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

0000000000000000000 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read from cache at offset, return word b

000000000000000000 0000000001 0100

Tag field	Index field	Offset
000000000000000000	0000000001	0100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

2. Read 0x0000001C = 0…00 0..001 1100

000000000000000000	0000000001	1100
Tag field	Index field	Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
•••			•••			
1022	0					
1023	0					

Index is Valid !

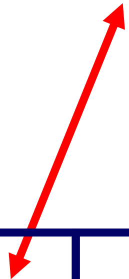
0000000000000000000	0000000001	1100
Tag field	Index field	Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Index valid, Tag Matches

00000000000000000000 0000000001 1100

Tag field | Index field | Offset



Valid

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Index Valid, Tag Matches, return d

Tag field	Index field	Offset
0000000000000000000	0000000001	1100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

3. Read 0x00000034 = ..011 0100

000000000000000000 0000000011 0100

Tag field | Index field | Offset

Valid

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So read block 3

0000000000000000000 0000000011 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

No valid data

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Load that cache block, return word f

000000000000000000 0000000011 0100

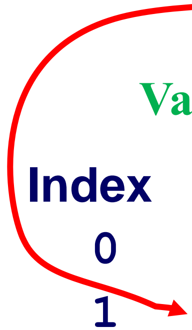
Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

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

0000000000000000010 0000000001 0100

Tag field Index field Offset



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

00000000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

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

000000000000000000010 0000000001 0100

Tag field is different | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

0000000000000000000010 0000000001 0100

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

00000000000000000010 0000000001 0100

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					

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

* Read address 0x00000030 !

  000000000000000000 0000000011 0000

* Read address 0x0000001c !

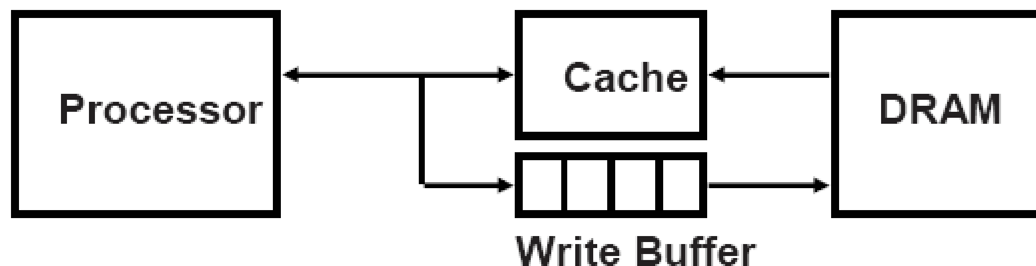
  000000000000000000 0000000001 1100

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 cache, kdy hrozí ztráta dat.



Realističtější formát řádky cache

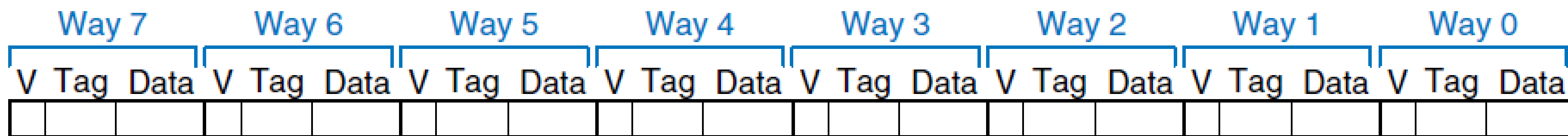
- **Tag** je index odpovídajícího bloku v operační paměti (v podstatě se jedná o hodnotu ukazatele/adresy dělenou délkou bloku).
- **Data** pole obsahující vlastní hodnoty na příslušných adresách či příslušné adrese.
- **Validity bit** – bit platnosti. Indikuje, zda je obsah pole Data vůbec platný.

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.

Plně asociativní cache

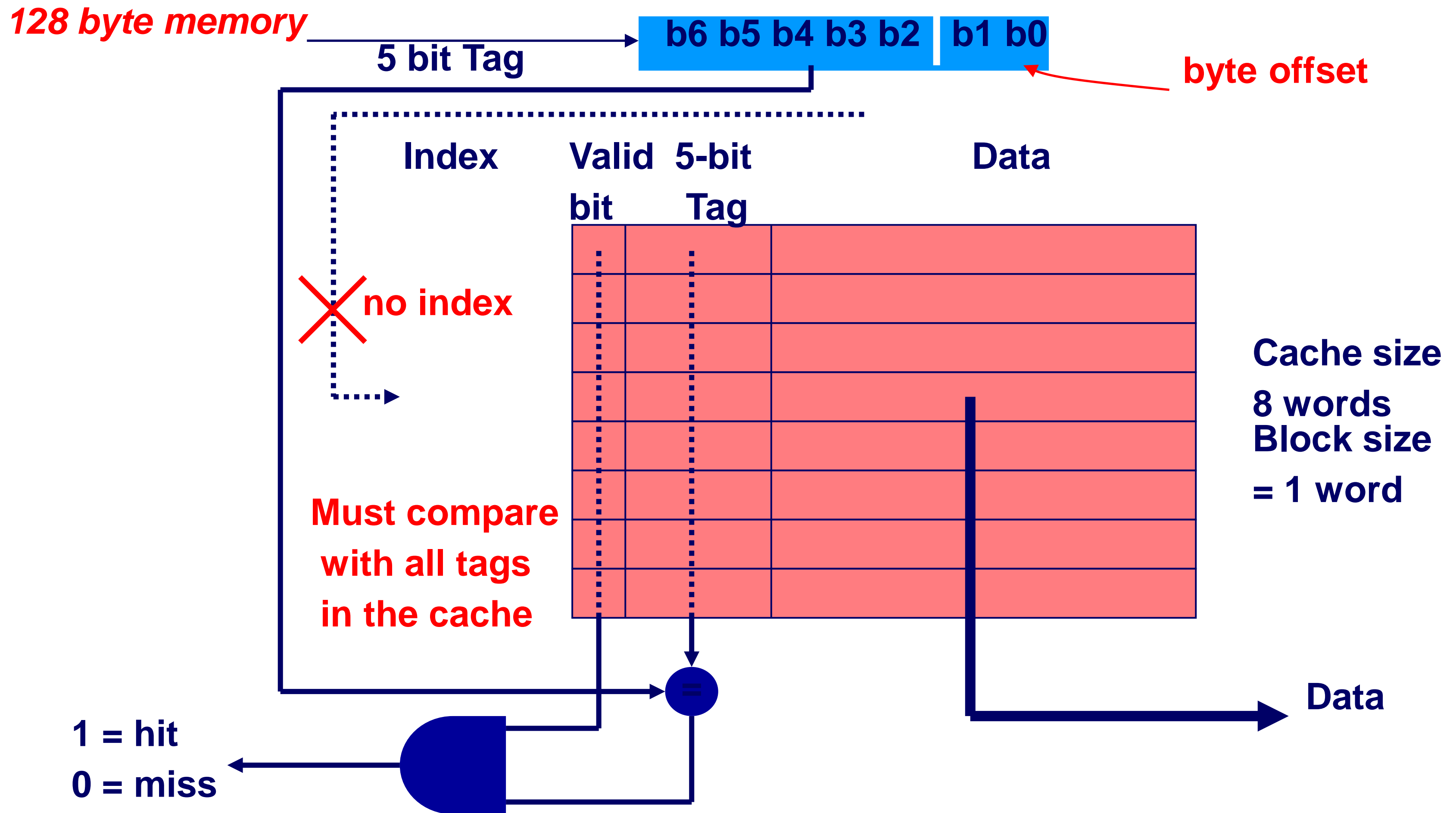
- Plně asociativní cache obsahuje jenom jeden set, stupeň asociativity je roven počtu bloků (N=B). Adresa paměti se může mapovat kamkoliv.
- … má pro danou kapacitu má nejméně konfliktů, ale potřebuje nejvíce HW prostředků (komparátory) – roste plocha čipu
- …je vhodná pro relativně malé cache.



A fully associative cache has only S=1 set !

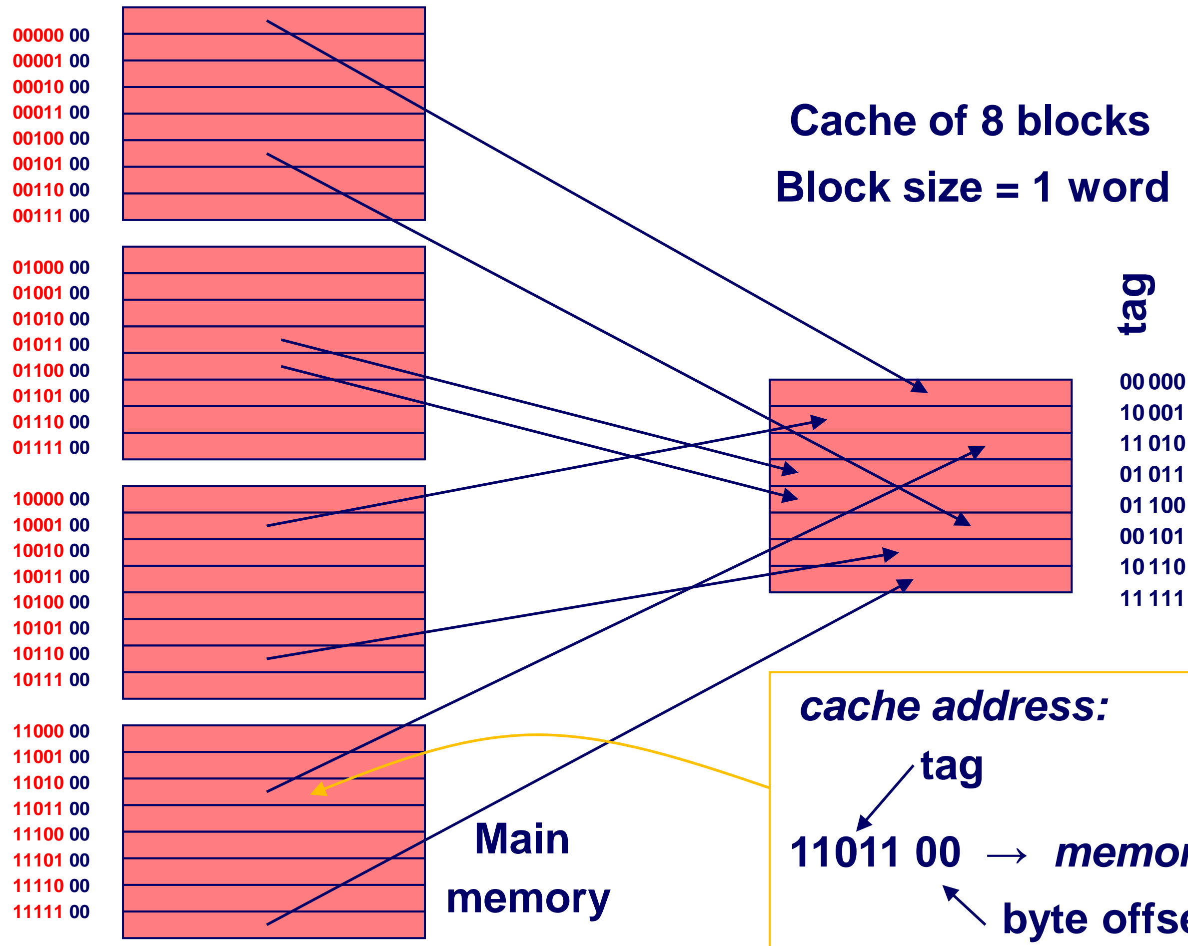
Finding a Word in Associative Cache

128 byte memory addressable as 4 byte words



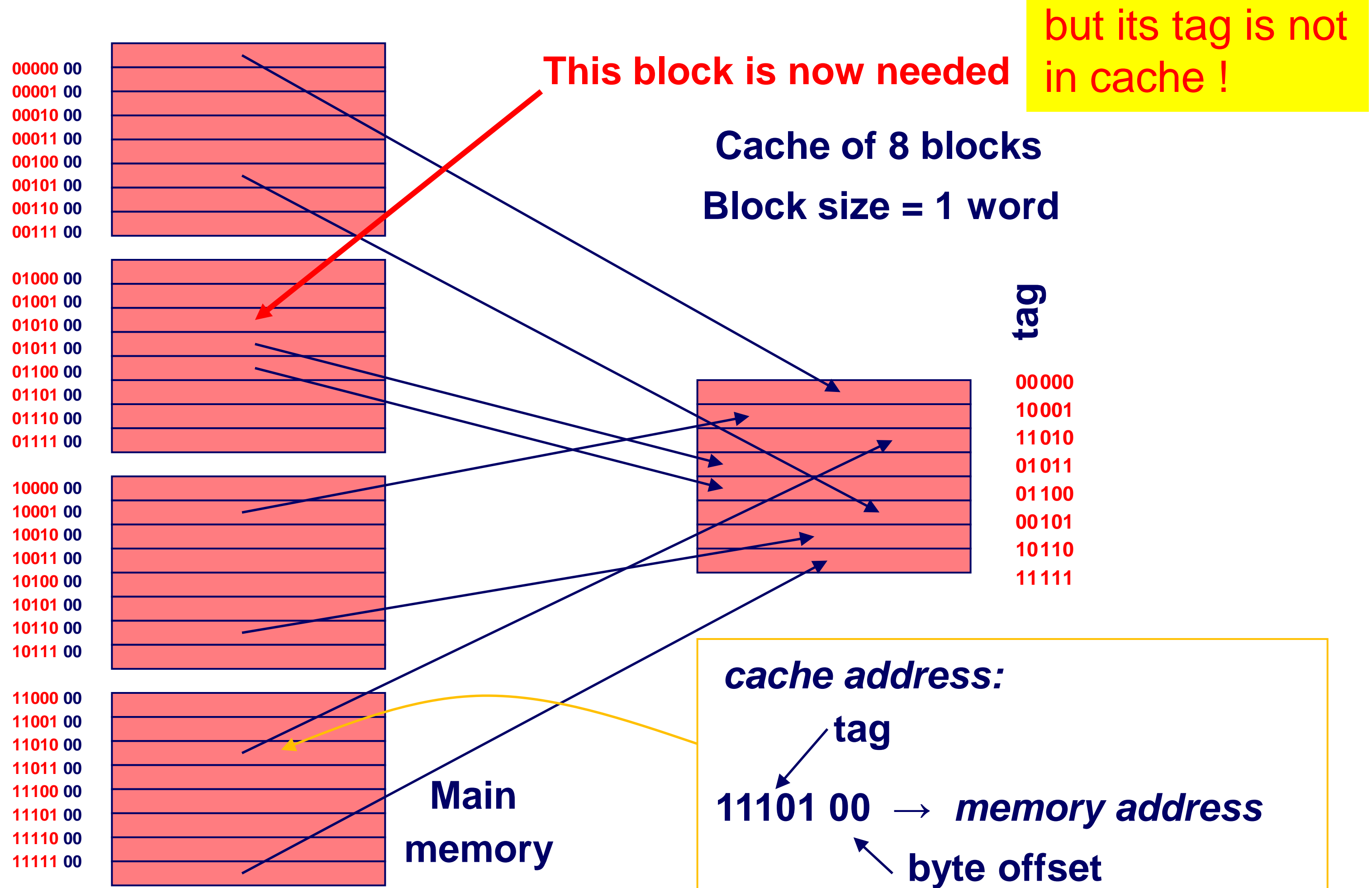
Fully-Associative Cache

128 byte memory addressable as 4 byte words



Fully-Associative Cache

128 byte memory addressable as 4 byte words



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.

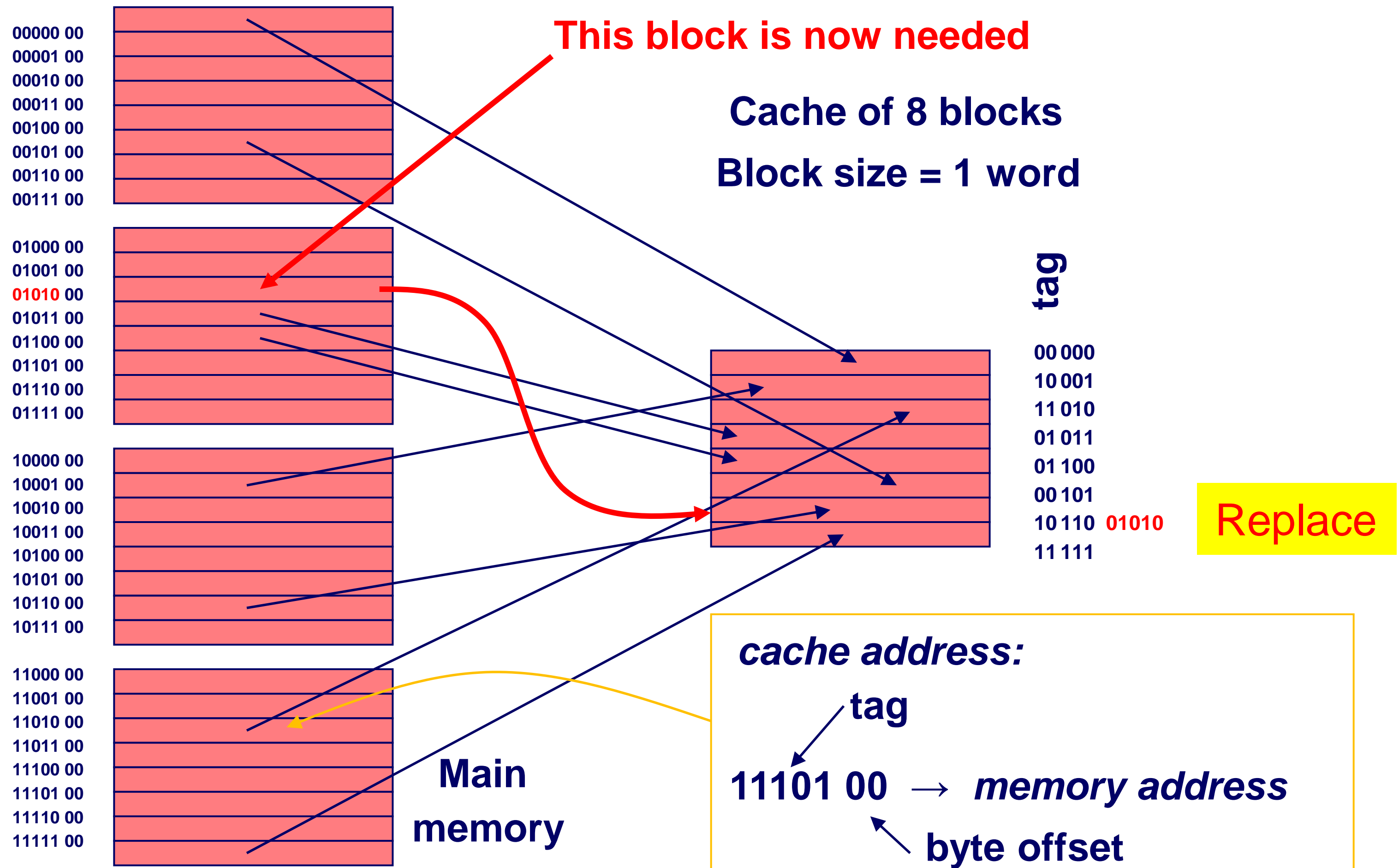
Strategie uvolňování bloků/řádek cache

- **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 nížse 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

128 byte memory addressable as 4 byte words

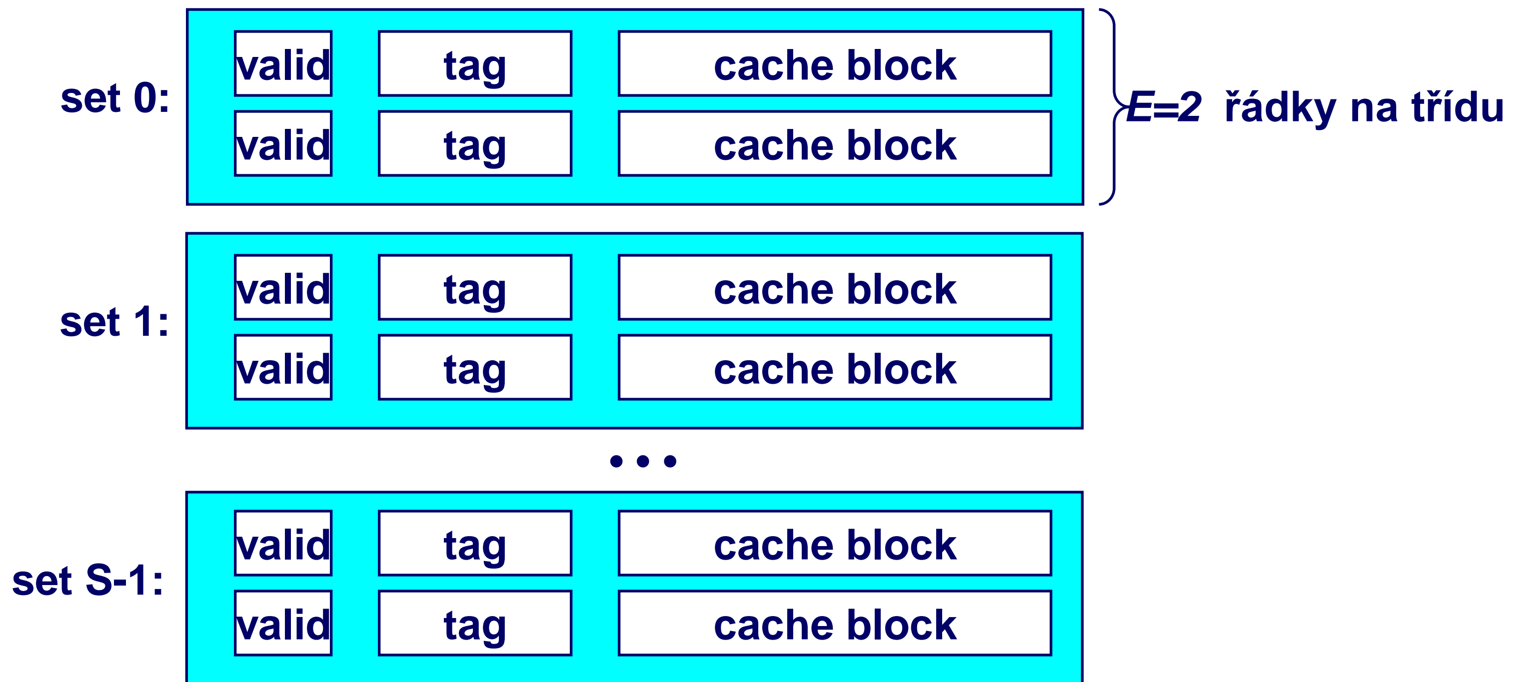


Diskuze k plně asociativní cache

- <u>Šířka pole Tag odpovídá šířce adresy</u>.
- Každý řádek cache obsahuje tolik jednobitových komparátorů, kolik je šířka adresy.
- 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

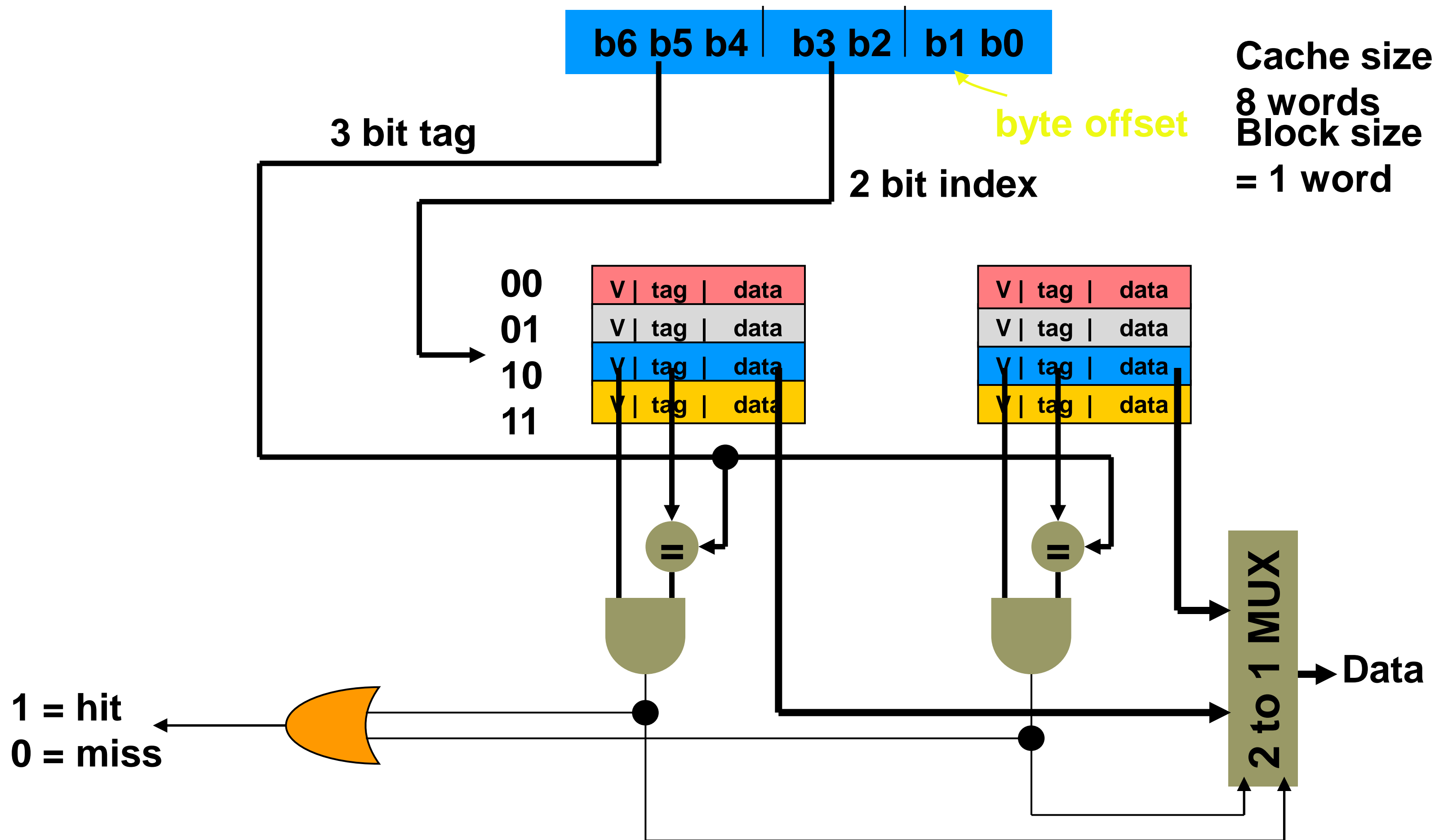
Více řádků (blocks) v třídě (set)



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

Two-Way Set-Associative Cache

128 byte memory addressable as 4 byte words



Construction: 2 direct mapped cashes are combined with associative selection

SP s omezeným stupněm asociativity N=2

Co přináší zvětšení stupně asociativity?

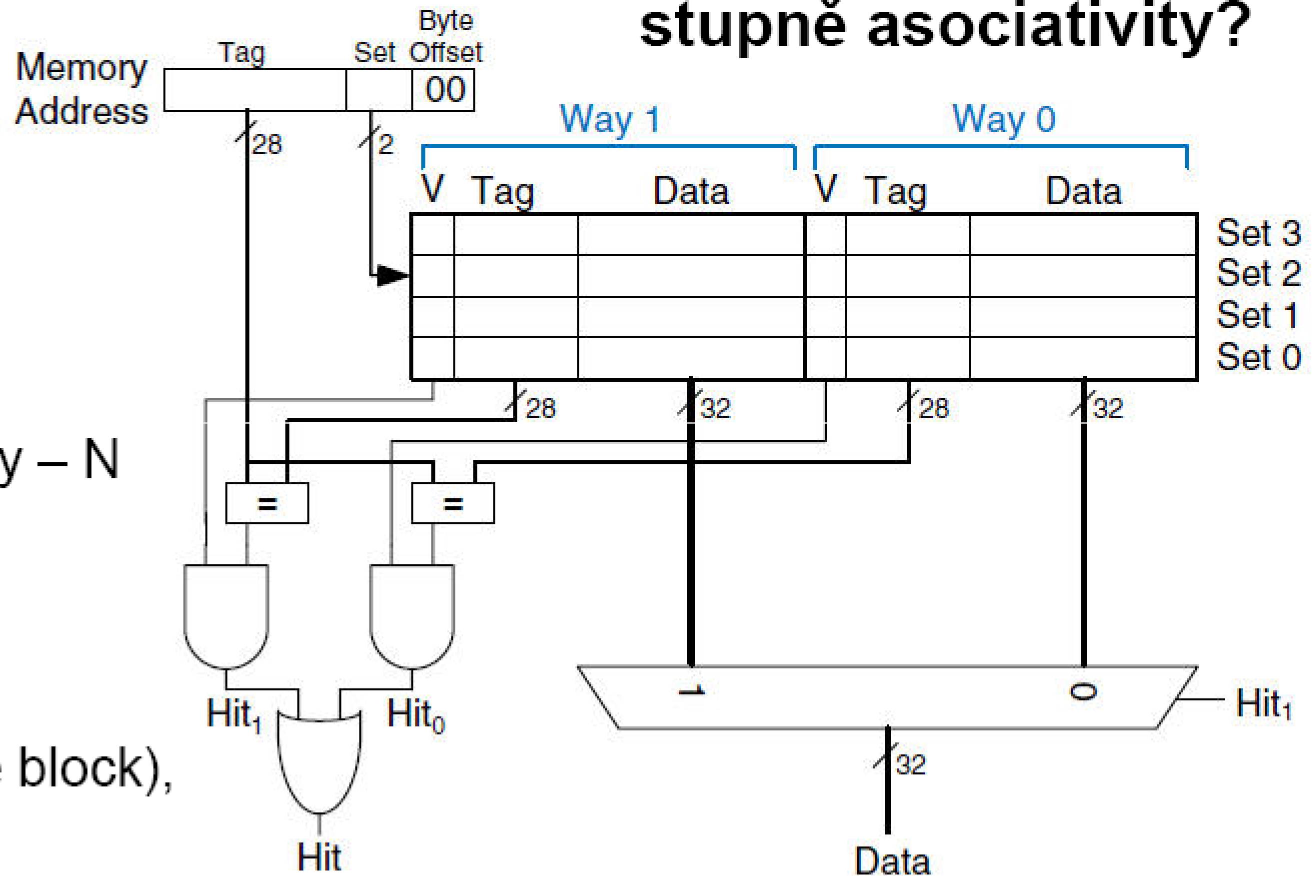
Capacity – C

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),

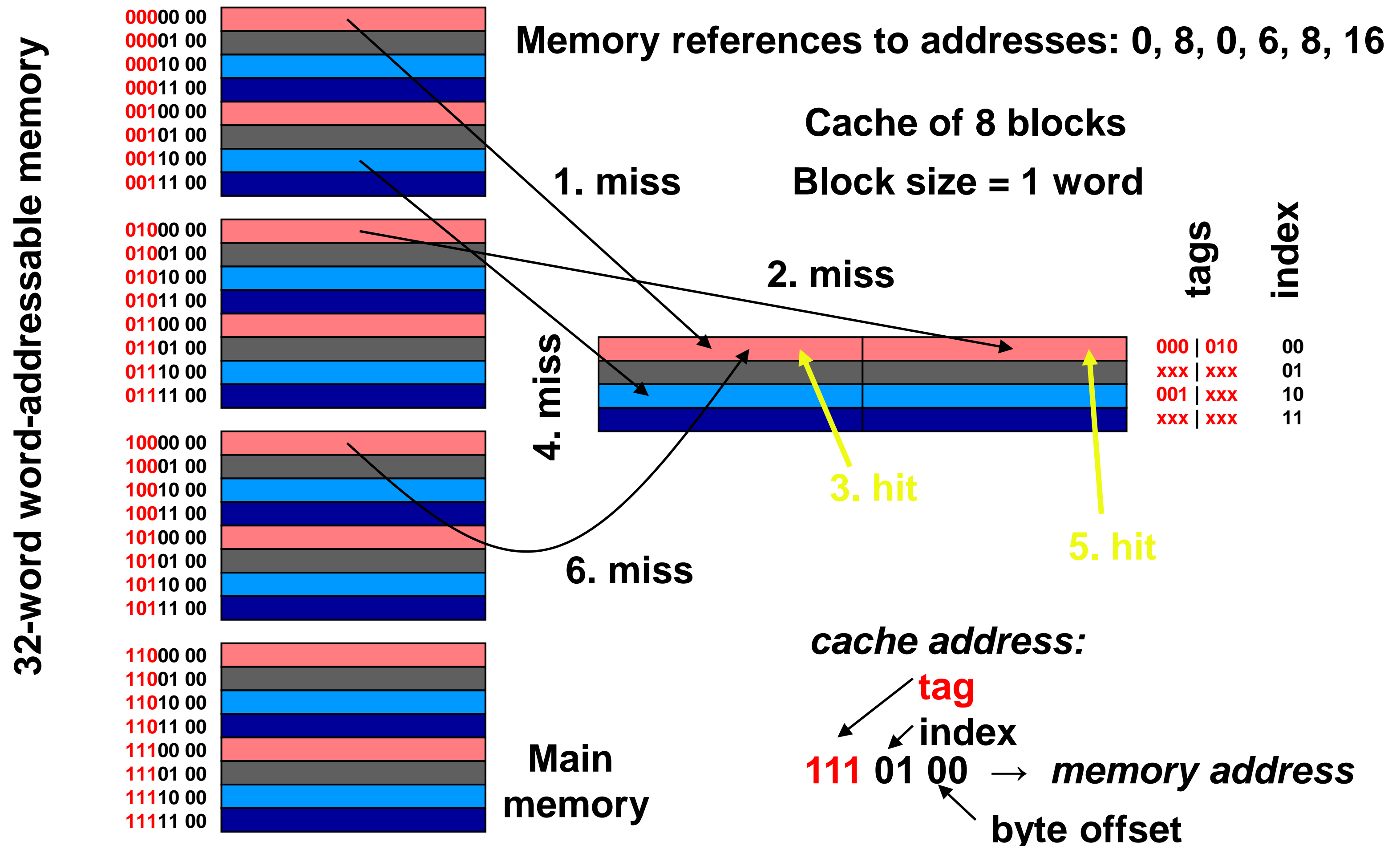
B = 8

N = 2

32 bit processor, its 1 word = 4 bytes

Miss Rate: Two-Way Set-Associative Cache

128 byte memory addressable as 4 byte words



LRU example in 2 way-cache

Tag 0, 2, 0, 1, 4

0: miss, bring into set 0 (loc 0)

2: miss, bring into set 0 (loc 1)

0: hit

1: miss, bring into set 1 (loc 0)

4: miss, bring into set 0 (loc 1, replace 2)

0: hit

	loc 0	loc 1
set 0	0	lru
set 1		

	loc 0	loc 1
set 0	lru 0	2
set 1		

	loc 0	loc 1
set 0	0	lru 2
set 1		

	loc 0	loc 1
set 0	0	lru 2
set 1	1	lru

	loc 0	loc 1
set 0	lru 0	4
set 1	1	lru

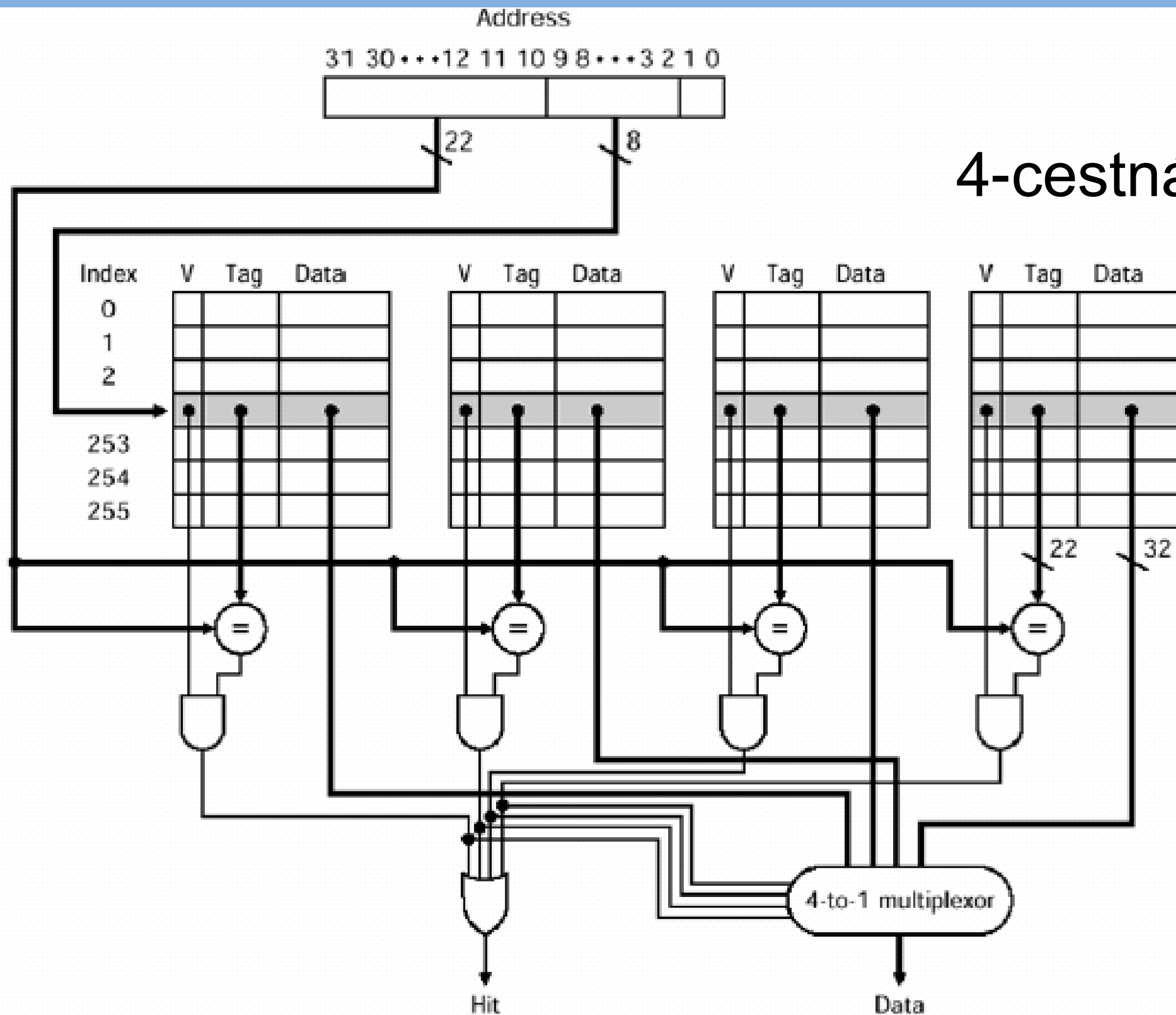
	loc 0	loc 1
set 0	0	lru 4
set 1	1	lru

[Source: Garcia, UCB]

Discussion: LRU Implementations

- For 2-way set associative cache, it is easy to keep track because we have only one LRU bit.
- With 4-way or greater, LRU requires more complicated hardware and much time. The number of states for full LRU implementation increases dramatically by factorial of the number of ways. It increases not only storage overhead but also time for updating!
- 4-way cache has 24 states and requires 5 bits per set. 8-way cache needs 40320 states and 16 bits per set.
- Therefore, LRU replacement policy is usually implemented by simpler mechanisms as **binary tree pseudo-LRU** but its algorithm is outside of APO topic, see e.g.: https://en.wikipedia.org/wiki/Pseudo-LRU , or 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.

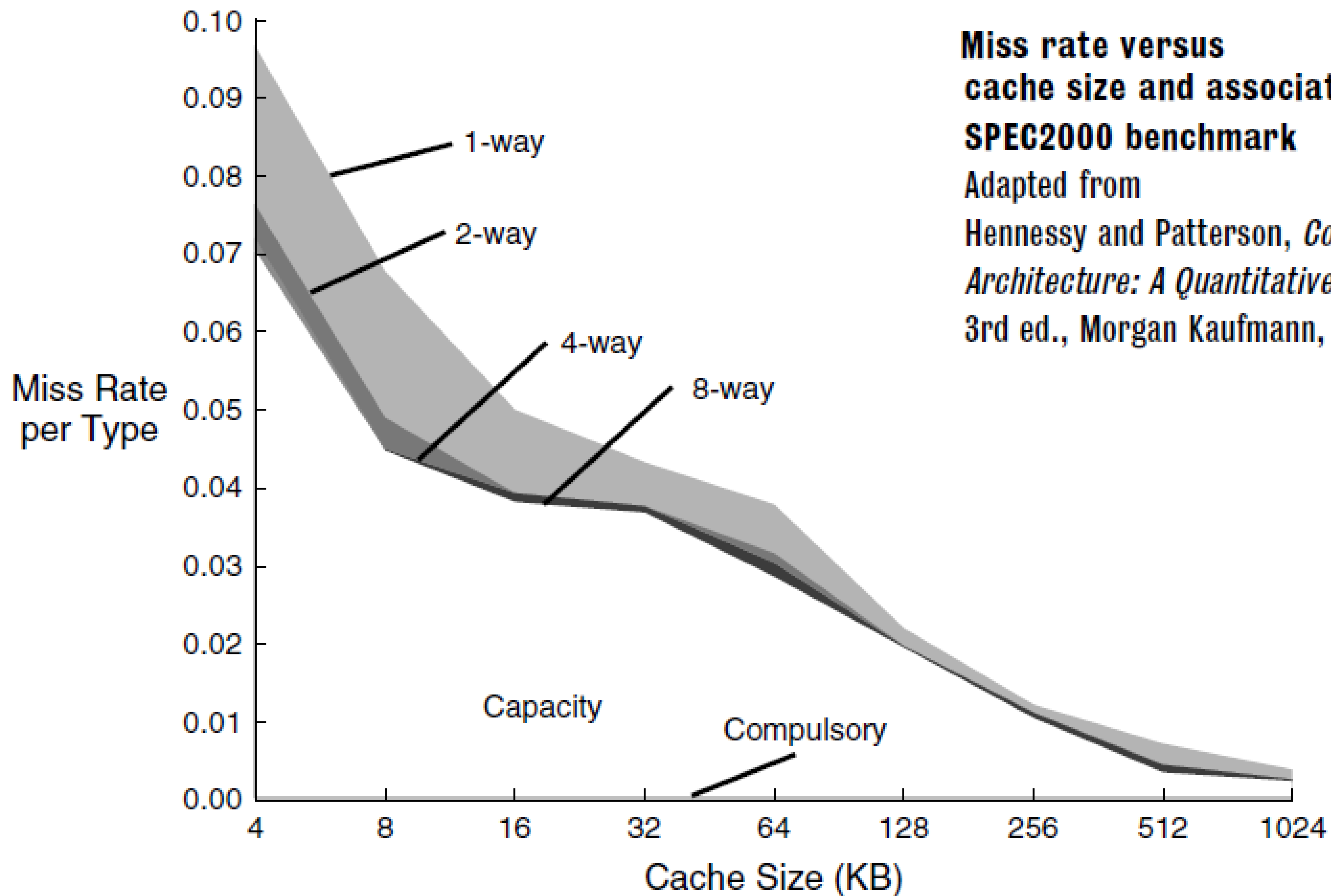
Cache s omezeným stupněm asociativity N=4



4-cestná, 4-way cache

32 bit processor 1 word = 4 bytes

Porovnání

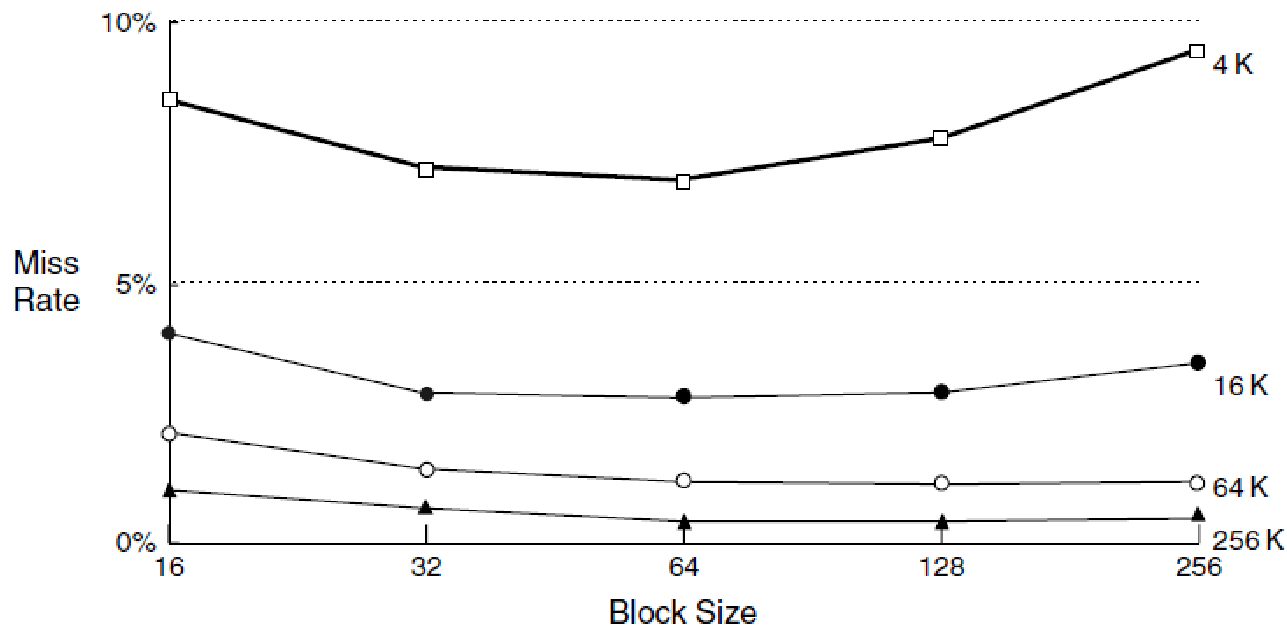


Pamatujte: 1. miss rate není vlastností cache!

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 dané velikosti cache rovněž znamená snižování počtu setů – to se projeví nárůstem konfliktů (nárůstem miss rate)…



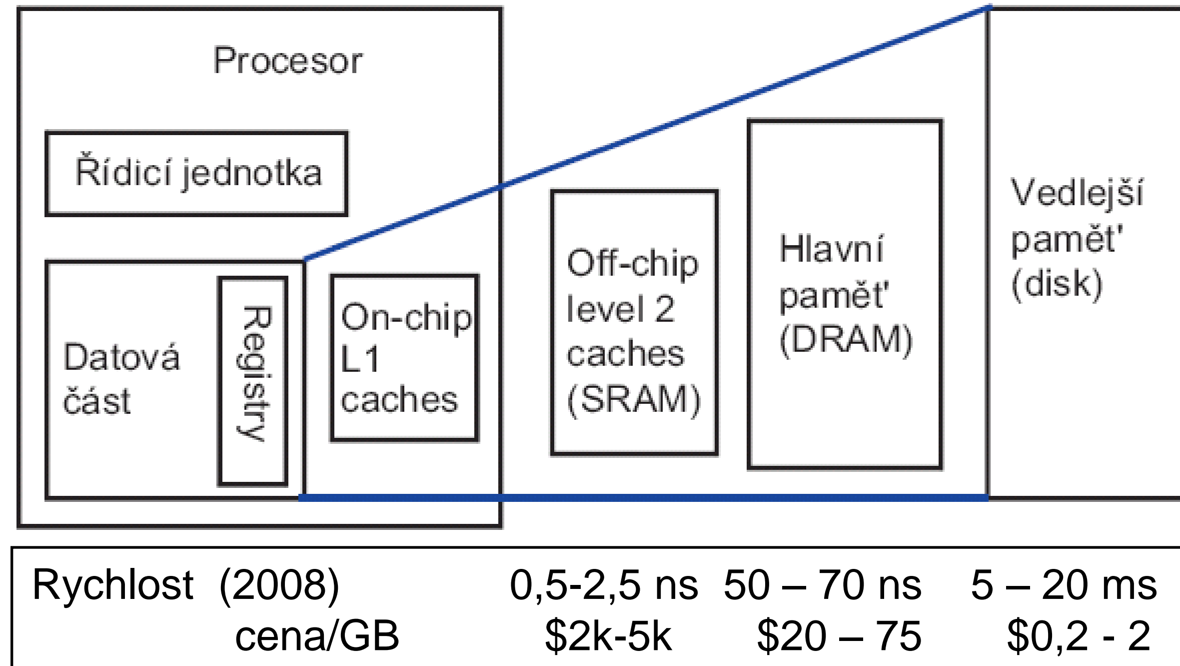
Miss rate versus block size and cache size on SPEC92 benchmark Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Trend - **Víceúrovňové SP**

- Primární SP je bezprostředně připojena k procesoru
  - Rychlá, malá. Nejdůležitější: minimální Hit Time, bývá 2-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

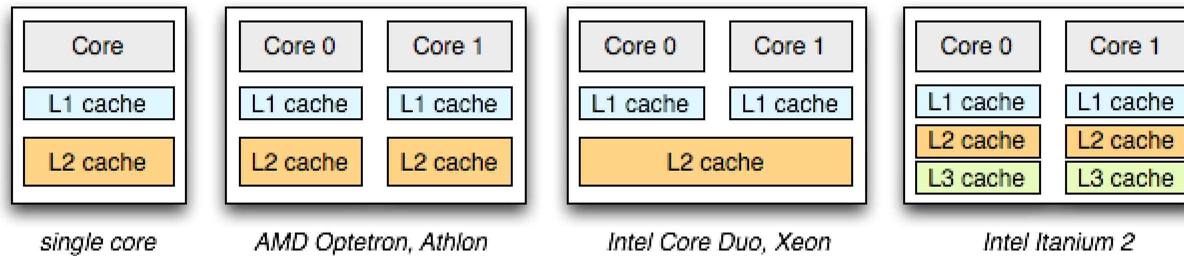
	Typicky pro L1	**Typicky pro L2**
Počet bloků	250-2000	15 000-250 000
KB	16-64	2 000-3 000
Velikost bloku v B	16-64	64-128
Miss penalty (v hod)	10-25	100-1 000
Miss rates	2-5%	0,1-2%

Paměťová hierarchie



- To se jedna a tatáž informace může objevit na více místech hierarchické paměti? Ano.

Cache Organization



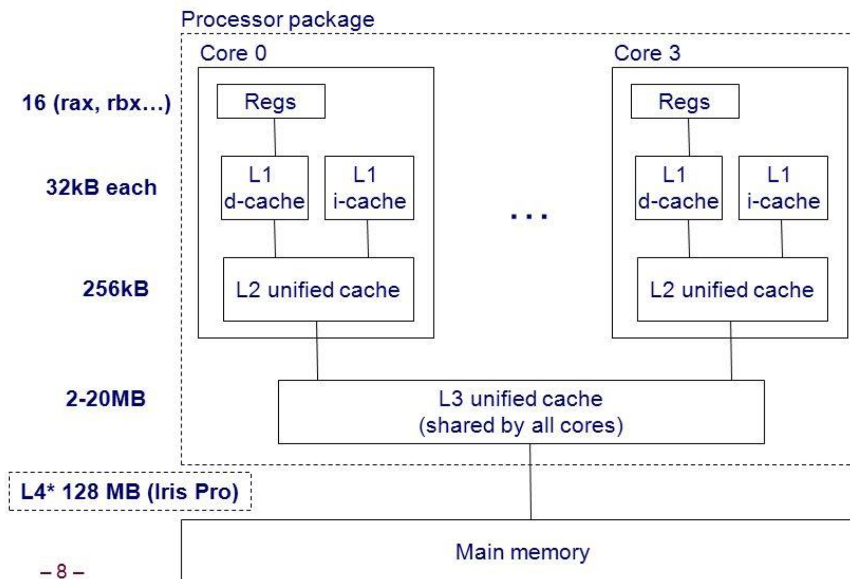
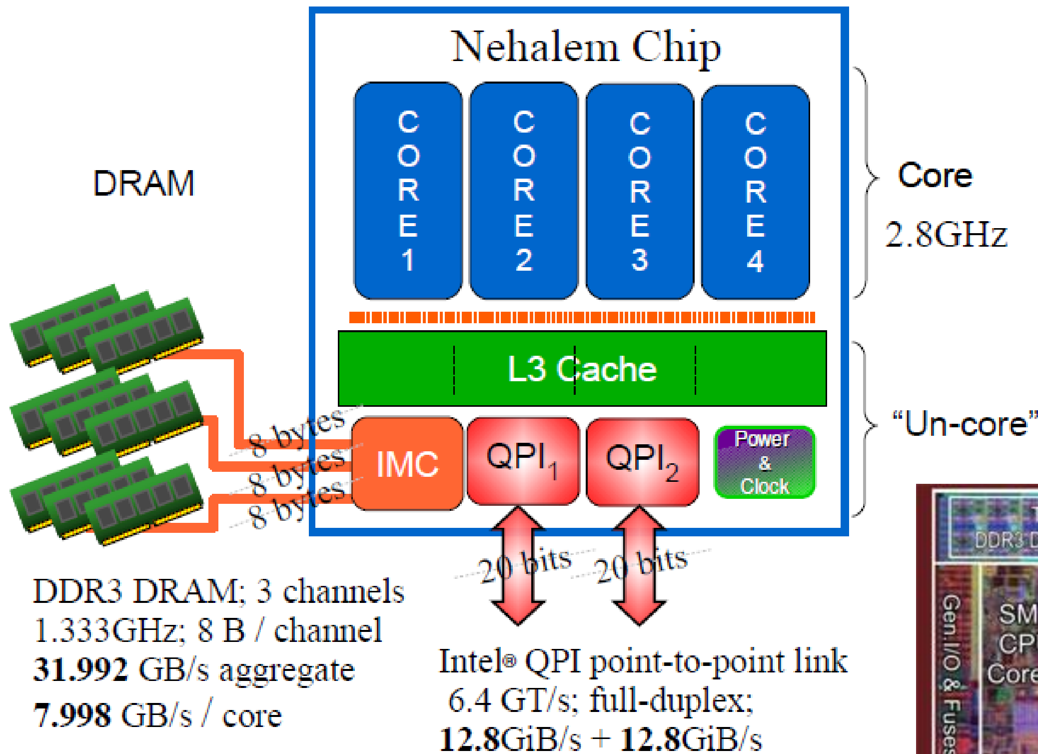
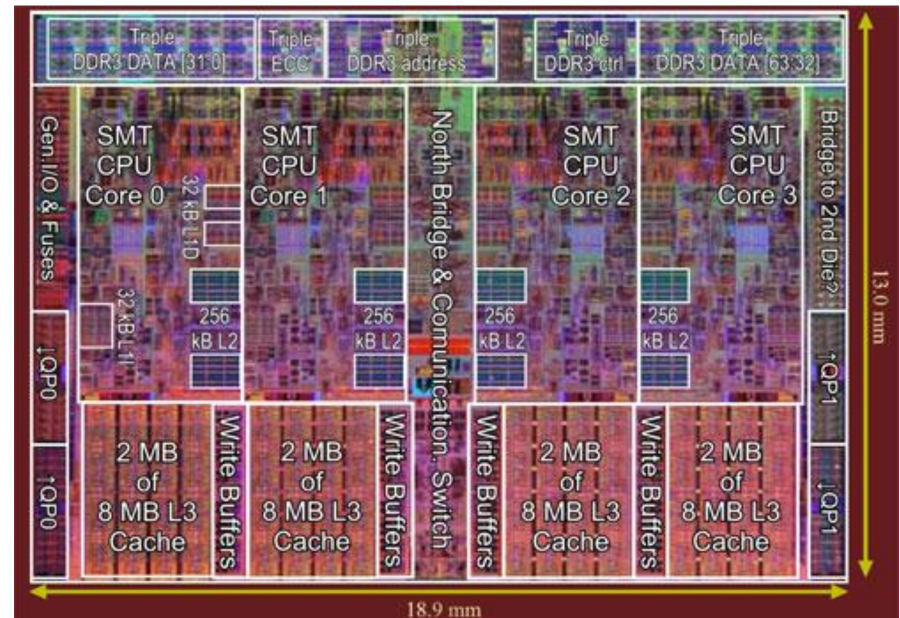


Image credit: Intel

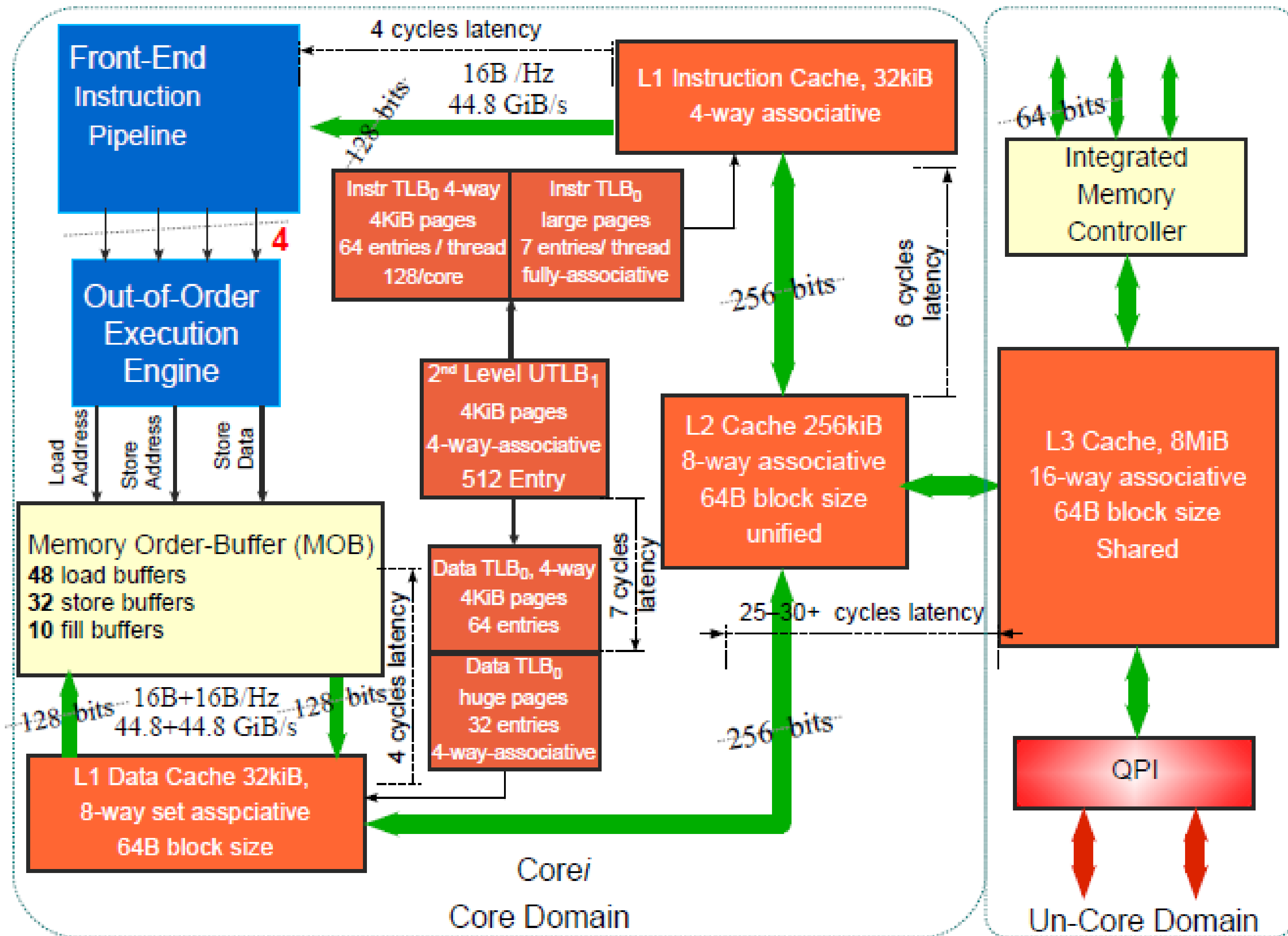
Příklad procesoru včetně cache
Harvardská architektura - Intel Nehalem (2008)



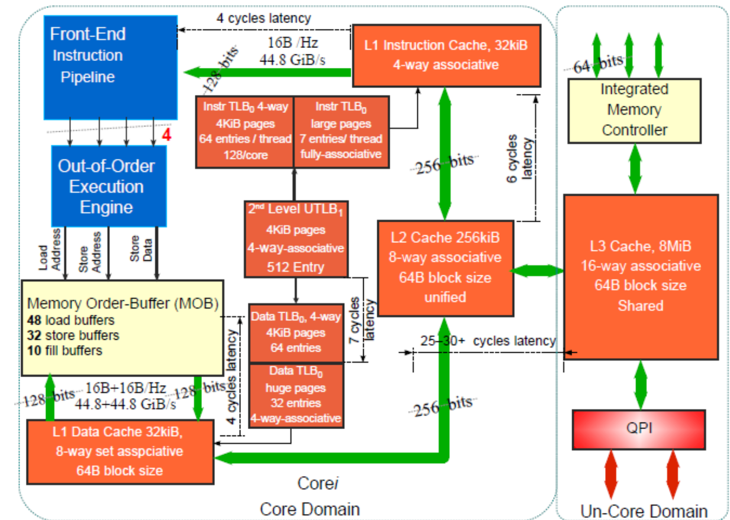
- IMC: integrated memory controller with 3 DDR3 memory channels,
- QPI: Quick-Path Interconnect ports
- Podívejte se na velkosti jednotlivých cache!!!



Intel Nehalem – memory subsystem structure



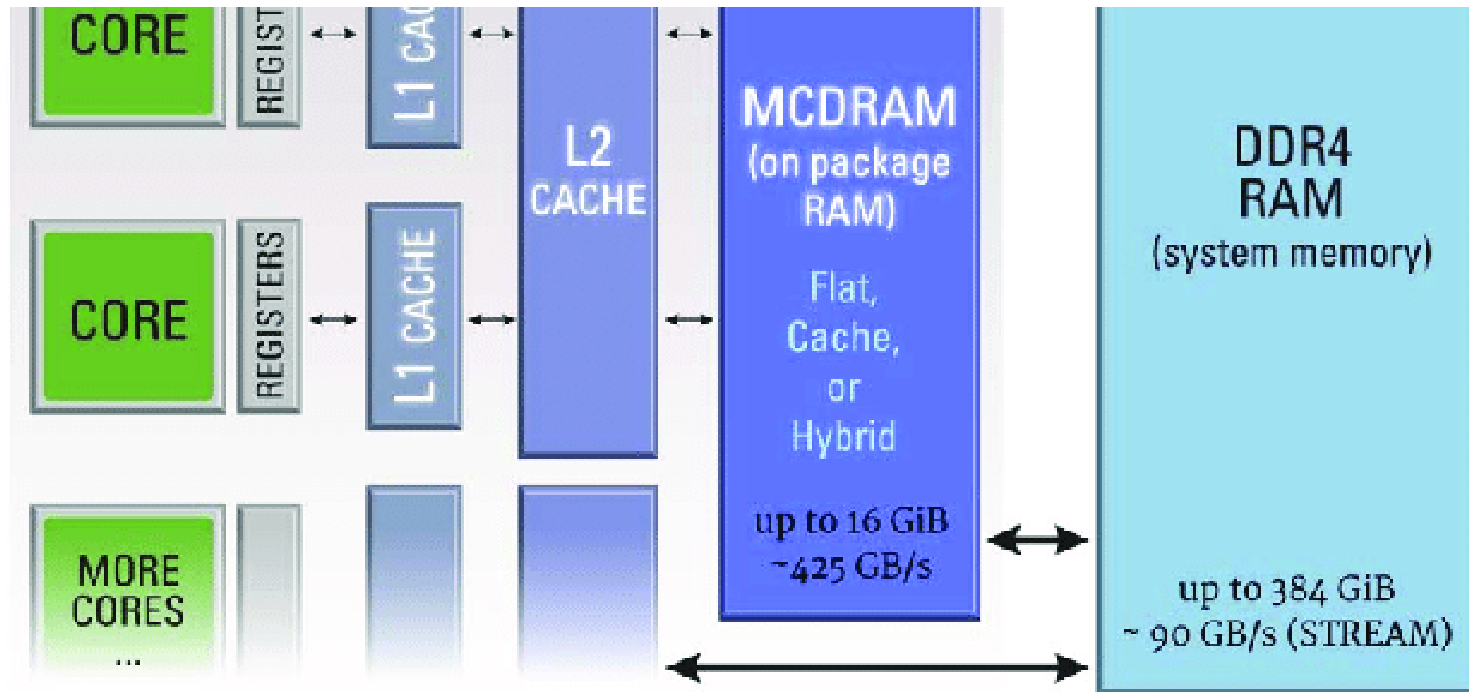
Notes for Intel Nehalem example



- Block size: 64B
- CPU reads whole cache line/block from
- main memory and each is 64B aligned
- (6 LS bits are zeros), partial line fills allowed
- L1 – Harvard. Shared by two (H)threads
- instruction – 4-way 32kB, data 8-way 32kB
- L2 – unified, 8-way, non-inclusive, WB
- L3 – unified, 16-way, inclusive (each line stored in L1 or L2 has copy in L3), WB
- Store Buffers – temporal data store for each write to eliminate wait for write to the cache or main memory. Ensure that final stores are in original order and solve “transaction” rollback or forced store for:
- - exceptions, interrupts, serialization/barrier instructions, lock prefix,..
- TLBs (Translation Lookaside Buffers) are separated for the first level
- Data L1 32kB/8-ways results in 4kB range (same as page) which allows to use 12 LSBs of virtual address to select L1 set in parallel with MMU/TLB

Intel Knights Landing Processors (from 2013)

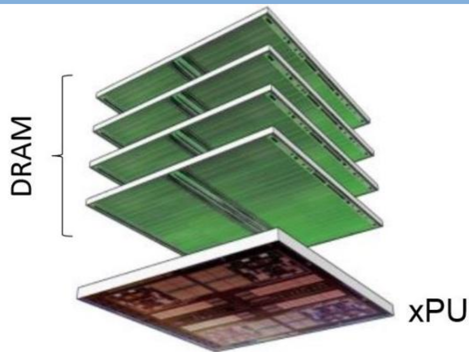
Knights Landing generation contains up to 72 Airmont (Atom) cores with four threads per core, manufactured as processors of series Xeon Phi 72xx

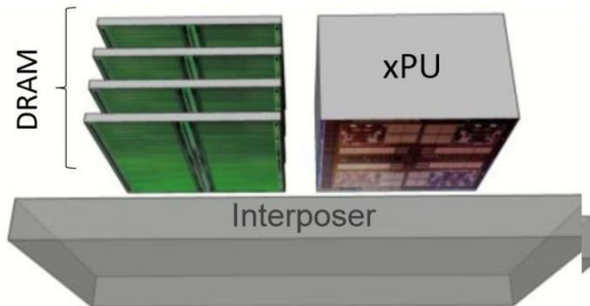


MCDRAM - Multi-Channel DRAM

Source: Haidar, Azzam & Jagode, Heike & Yarkhan, Asim & Vaccaro, Phil & Tomov, Stanimire & Dongarra, Jack. (2017): Power-aware computing: Measurement, control, and performance analysis for Intel Xeon Phi. 1-7, *2017 IEEE High Performance Extreme Computing Conference (HPEC)*

HBM and MCDRAM

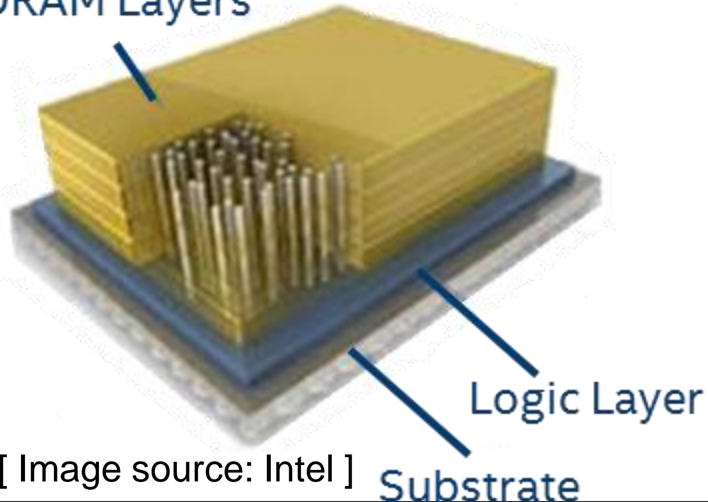




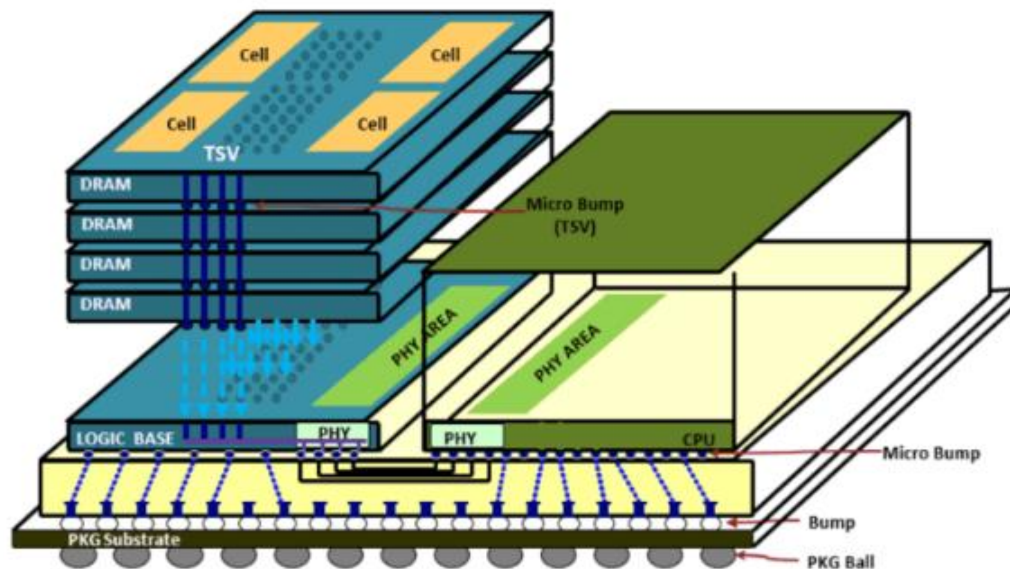
HBM =
High Bandwidth Memory

[Image source: Overclocking, HBM in AMD]

MCDRAM Multi-Channel DRAM



[Image source: Intel]



[Image source: **Michael Feldman**]

Intel Processors

Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second Sometimes shared by two cores
L3 cache	8 MB or more	10x slower than L2	>400 GB/second
MCDRAM		2x slower than L3	400 GB/second
Main memory on DDR DIMMs	4 GB-1 TB	Similar to MCDRAM	100 GB/second
I/O devices on memory bus	6 TB	100x-1000x slower than memory	25 GB/second
I/O devices on PCIe bus	Limited only by cost	From less than milliseconds to minutes	GB-TB/hour Depends on distance and hardware

[Source Intel]

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.

Pochopili jste tuto přednášku?

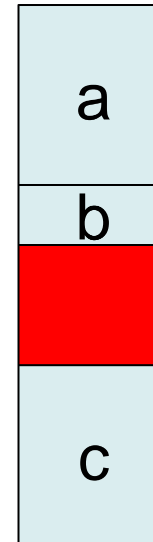
- Neplýtvejme pamětí – používejme minimální množství paměti
- Spatřujete rozdíl v těchto deklaracích?

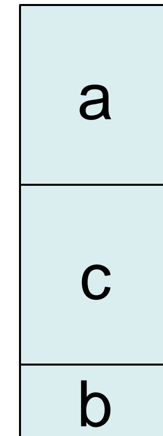
- /* Před optimalizací */

```
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, ke kterým 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?

- Dalším příkladem je násobení matic
  Lépe je však použít tzv. blokové násobení.
  Idea: Rozdělme výpočet 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;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;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)

Did you understand this lecture?

Sometimes it is also necessary to think about aligning data in memory *- either directly in assembler or in C - check if your compiler aligns double to 8-byte boundary, if not:*

- *allocate how much you need + 7 (or even more by alignment)*
- *use AND to get an aligned address for your data, example:*

```
const int SIZE_OF_ARRAY = 64;
// we define struct to see our array in debugger as an array
struct MyArr { double Item[SIZE_OF_ARRAY];  } *myArr;
double *p = (double*)malloc( sizeof(double) * (SIZE_OF_ARRAY+1) );
// align 3, i.e., the last tree bits are always zero, -8 = 0b1111 … 1111 1000
myArr = (MyArr*)( (long)((unsigned char *)p + 7) & -8 );
int i=0; double d=0.5; double * pd = myArr->Item;  // a usage of array
do { *pd++ = (d *= 2); } while (++i < SIZE_OF_ARRAY);
```