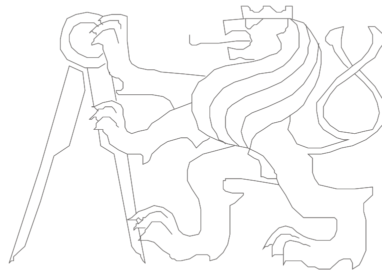
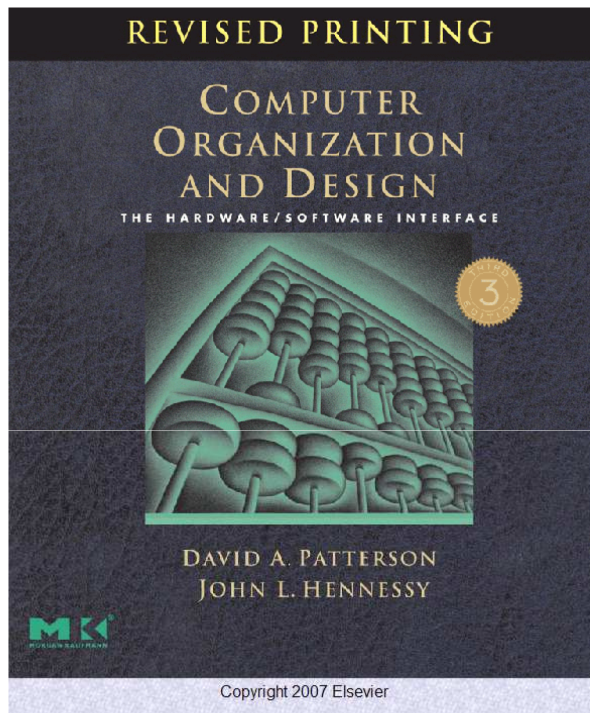
Architektury počítačů

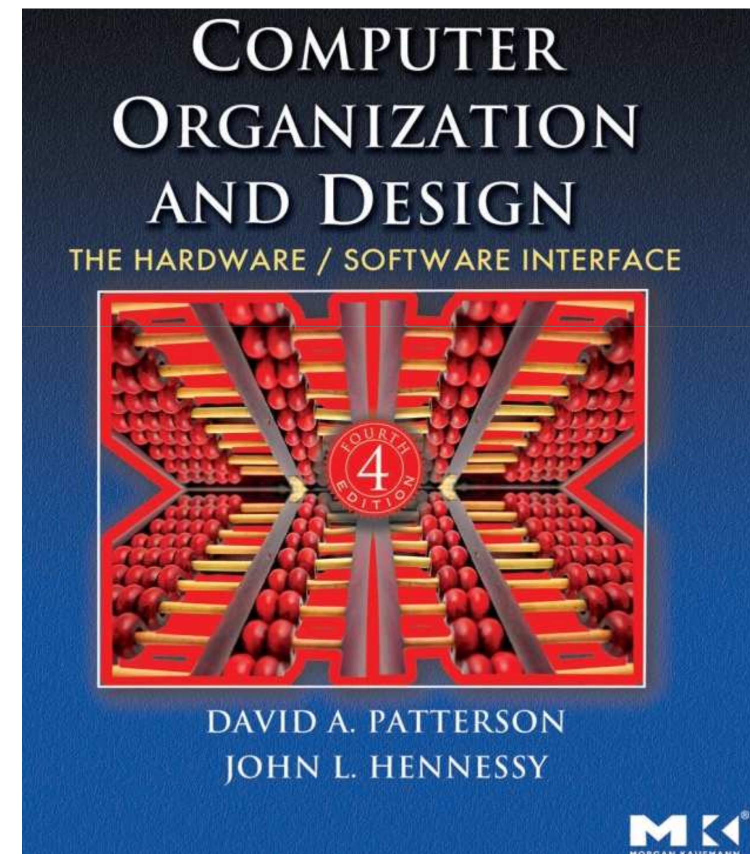
Paměť – část první
- úvod, realizace pamětí, cache



České vysoké učení technické, Fakulta elektrotechnická

Literatura k předmětu





Motivace pro přednášku z pohledu programátora?

Otázka 1.: Vykonávají programy to samé?

Otázka 2.: Který program je rychlejší (pokud některý)?

A:

```
int matrix[M][N];
int i,j,sum=0;
…
for(i=0;i<M;i++)
  for(j=0;j<N;j++)
    sum+=matrix[i][j];
```

B:

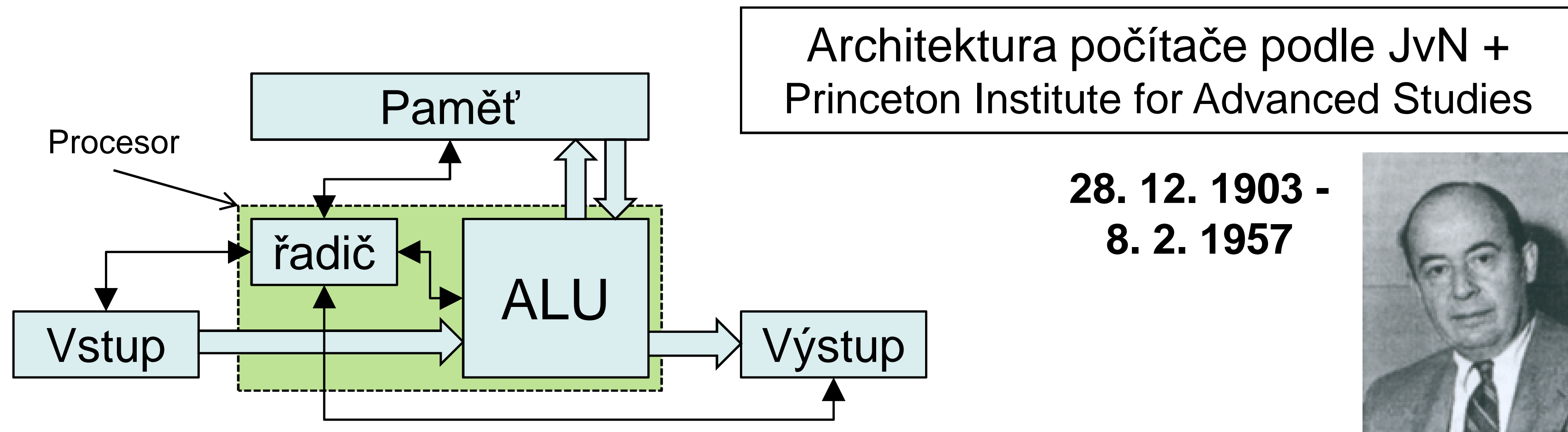
```
int matrix[M][N];
int i,j,sum=0;
…
for(j=0;j<N;j++)
  for(i=0;i<M;i++)
    sum+=matrix[i][j];
```

Lze doporučit výhodnější způsob procházení matice?

Osnova přednášky

- Paměťová terminologie
- Hierarchie pamětí
- SP (skrytá paměť), resp. cache
  - Více realistické SP
  - Víceúrovňové SP
- Virtuální paměť
- Problémy hierarchických pamětí
- Realizace pamětí - paměťové čipy
- Jiné principy vedlejších pamětí

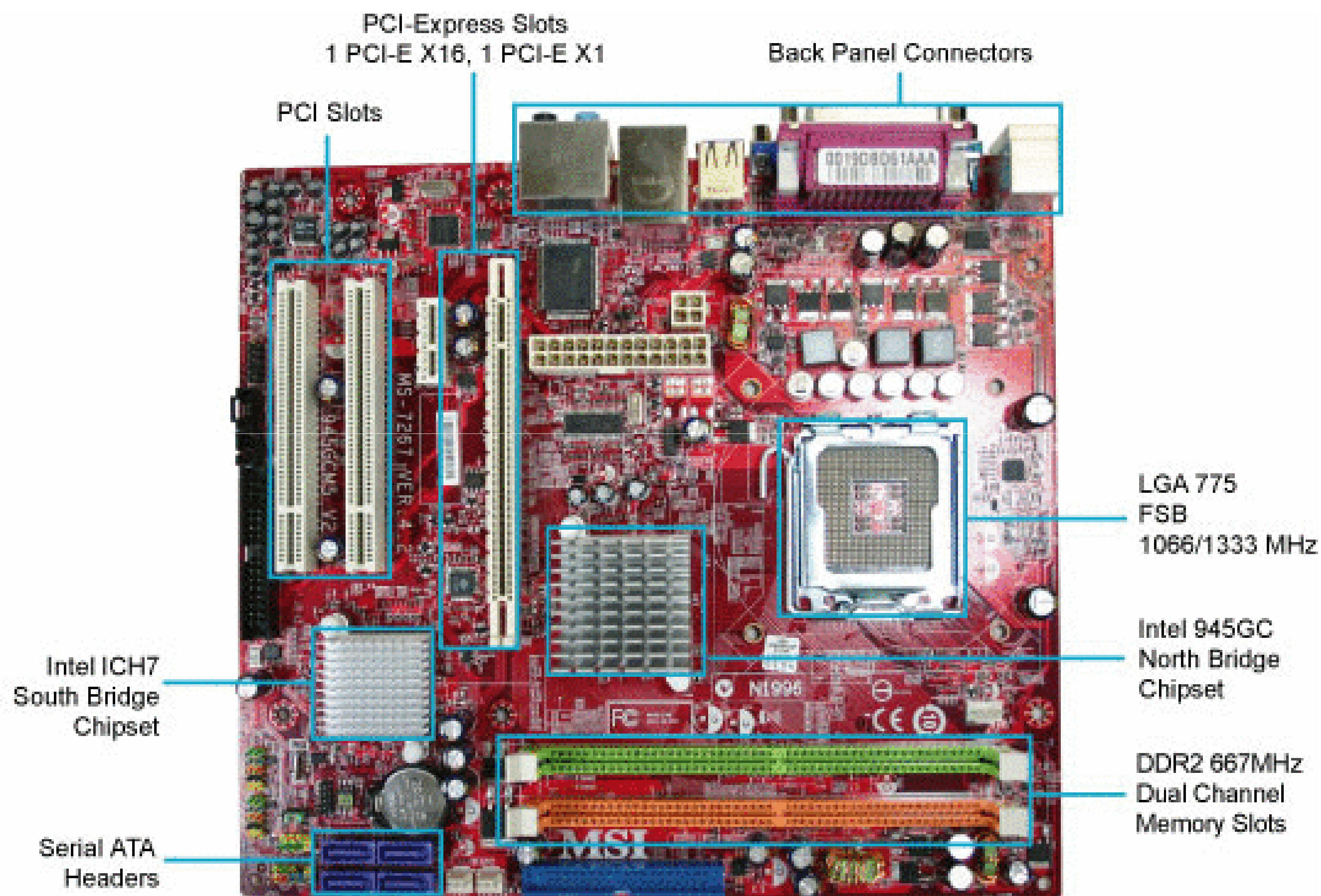
John von Neumann, maďarský fyzik



28. 12. 1903 - 8. 2. 1957

- 5 funkčních jednotek – řídicí jednotka (řadič), aritmeticko-logická jednotka, paměť, vstupní zařízení, výstupní zařízení
- Nezávislost struktury počítače na zpracovávaných problémech. Musí se zavést program a musí se uložit do paměti. Ten řídí činnost počítače.
- **Programy a výsledky (data) se ukládají do téže paměti.** Ta je rozdělena na stejně velké části (buňky), které jsou průběžně očíslované – adresa.
- Po sobě jdoucí instrukce se ukládají do po sobě jdoucích buněk.
- Existují instrukce aritmetické, logické, přenosu, skokové a ostatní.

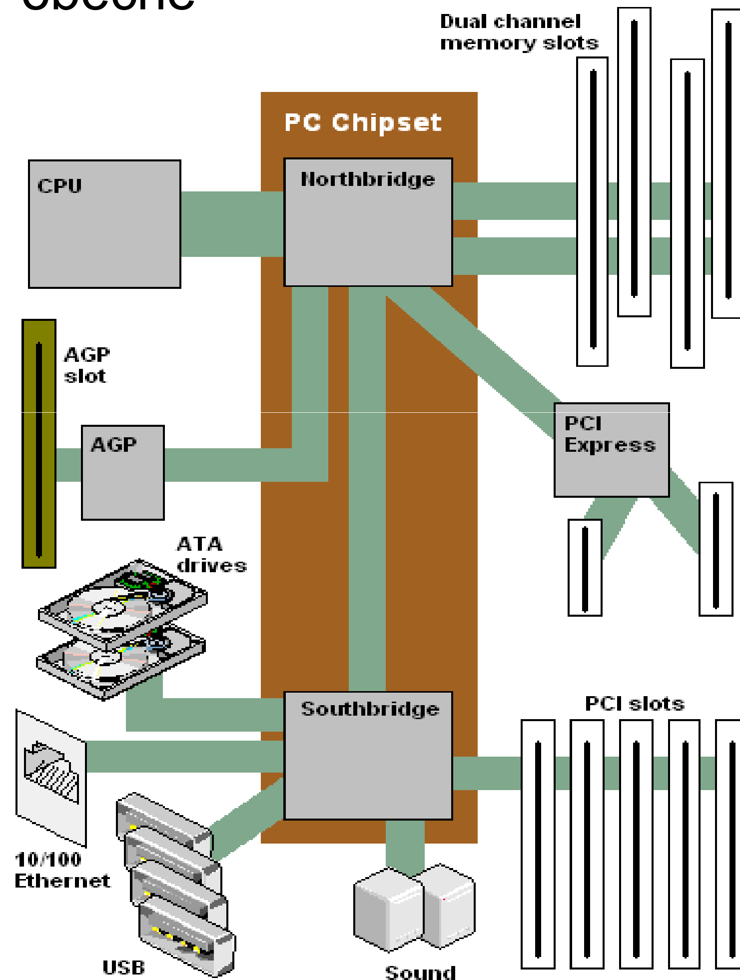
Architektura počítače



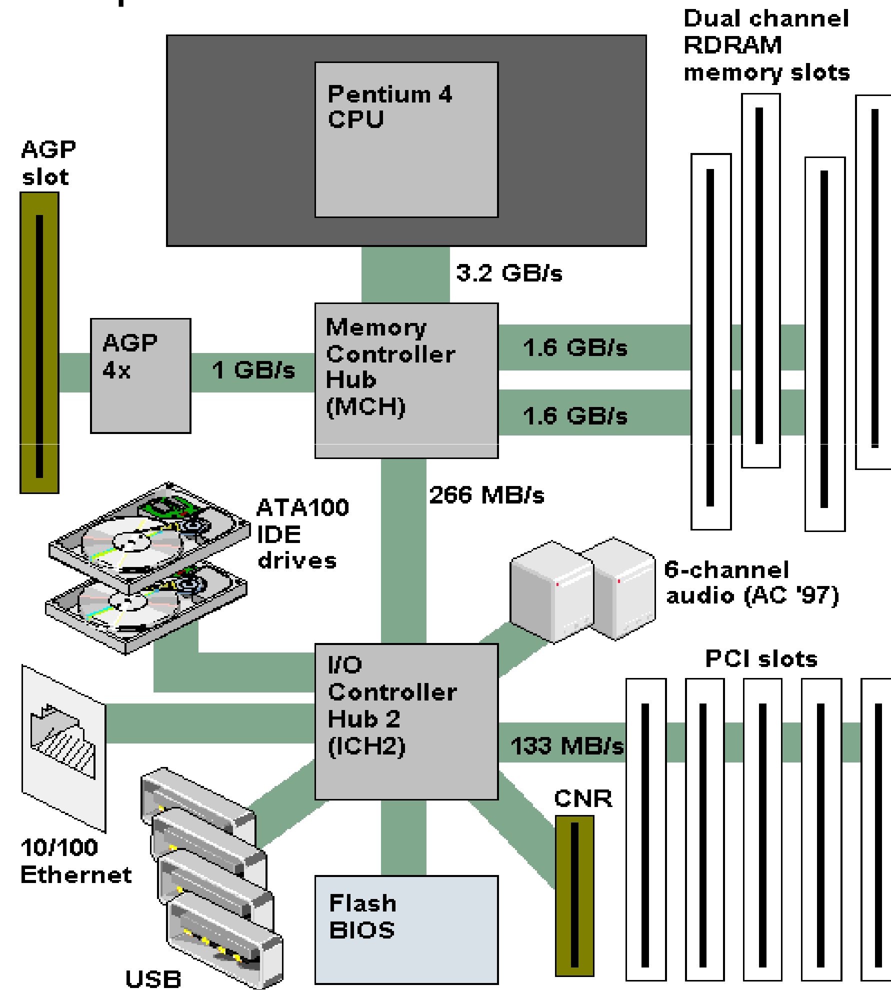
http://www.pcplanetsystems.com/abc/product_details.php?item_id=3263&category_id=208

Architektura počítače

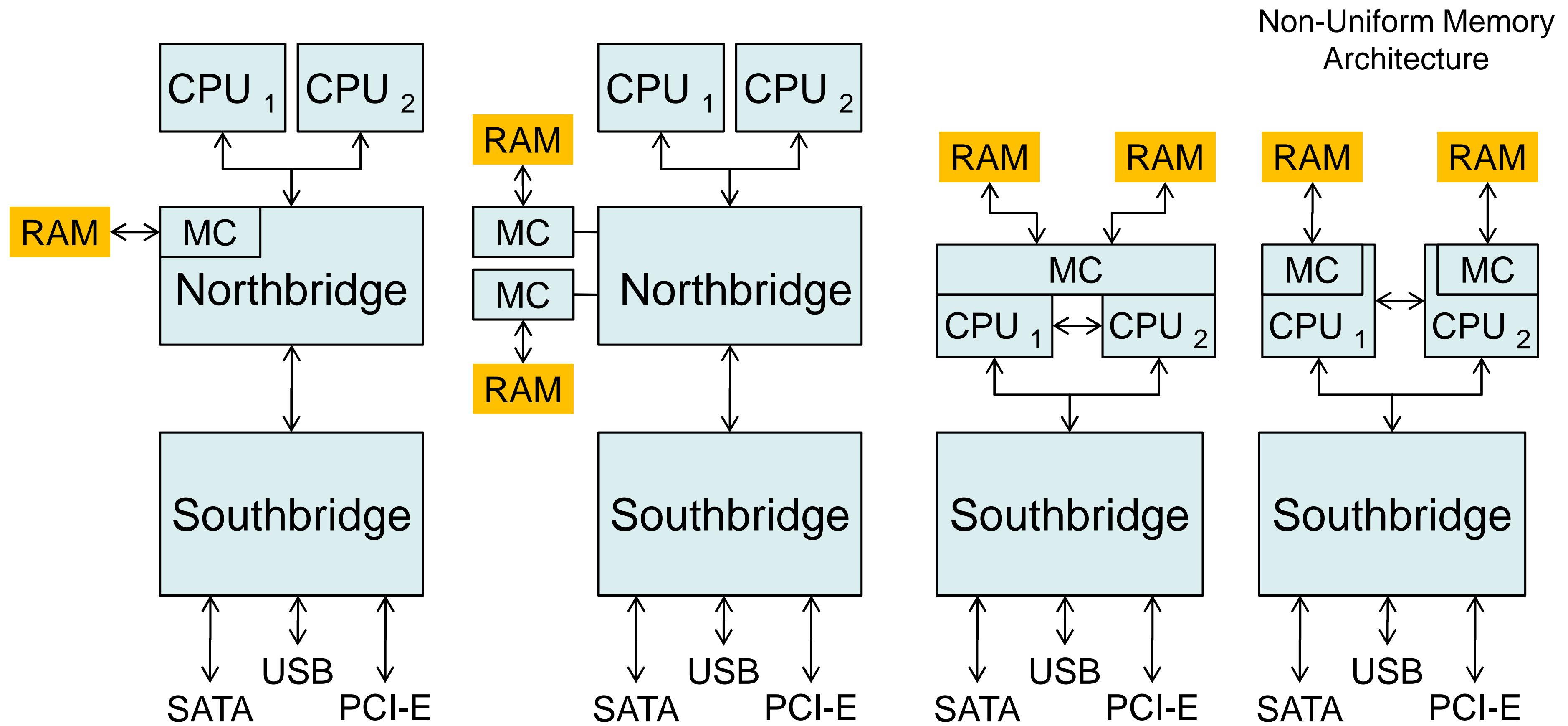
obecně



příklad



Architektura počítače



MC - Memory controller – obsahuje obvody pro zajištění operace čtení a zápisu z/do paměti. Také se stará o udržení obsahu paměti – refreshing každých několik desítek ms

Konkrétněji…

Intel® Core™2 Duo Processor
Intel® Core™2 Quad Processor

10.6 GB/s

Display support for HDMI, DVI, HDCP, MEC, ADD2

Intel® Graphics Media Accelerator 3100 with Intel® Clear Video Technology

PCI Express x16 Graphics

8 GB/s

G33 GMCH

DDR2 or DDR3 6.4 GB/s or 8.5 GB/s

DDR2 or DDR3 6.4 GB/s or 8.5 GB/s

2 GB/s DMI

12 Hi-Speed USB 2.0 Ports; Dual EHCI; USB port disable

480 Mb/s each x2

6 PCI Express* x1

500 MB/s each x1

Intel® Integrated 10/100/1000 MAC

GLCI LCI

Intel® Gigabit LAN Connect

ICH9 ICH9DH ICH9R

LPC or SPI

BIOS Support

Intel® High Definition Audio

Intel® Quiet System Technology

3 Gb/s each

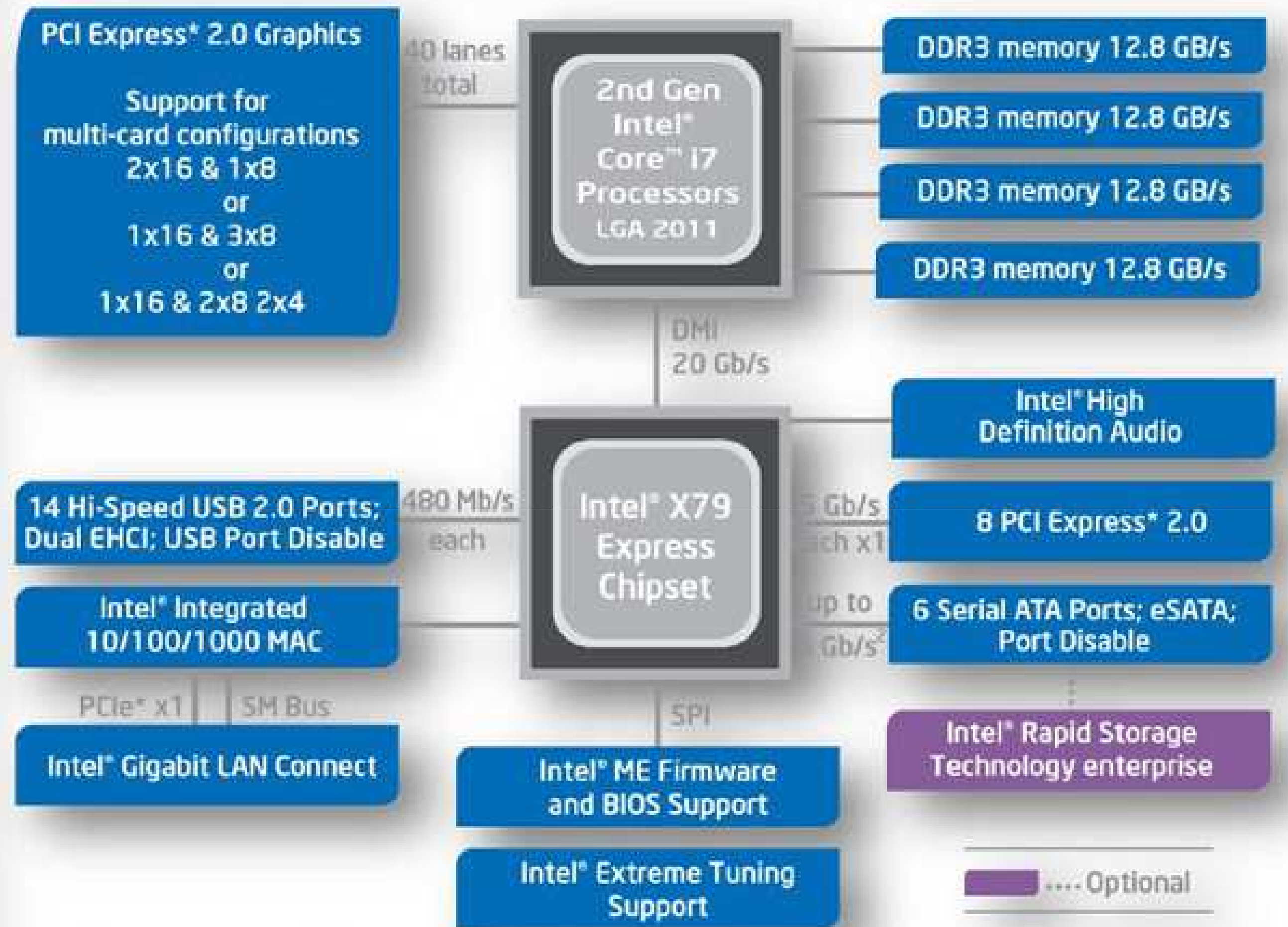
6 Serial ATA Ports; eSATA; Port Disable

Intel® Matrix Storage Technology

Intel® Turbo Memory

Nyní je Northbridge jako Graphics and Memory Controller Hub (GMCH)

Konkrétněji…



Intel® X79 Express Chipset Block Diagram

Detaily v přednáškách č.5 a 6

Terminologie kolem pamětí

- Adresa, pojem snad není třeba vysvětlovat.
- Hodnota, vlastní informace. Paměťová buňka však může obsahovat i další informaci (třeba o platnosti hodnoty, apod.).

- Parametry pamětí:
  - Vybavovací doba paměti, kritický parametr. Délka časového intervalu mezi objevením se požadavku a okamžikem, kdy jsou data k dispozici.
  - Doba přístupu, zastaralý parametr; vybavovací doba + obnovení obsahu po destruktivním čtení.
  - Propustnost, výkonový parametr. Schopnost zpracovat uvedené množství za jednotku času.
  - Latence = zpoždění, podobně jako vybavovací doba.

Terminologie kolem pamětí

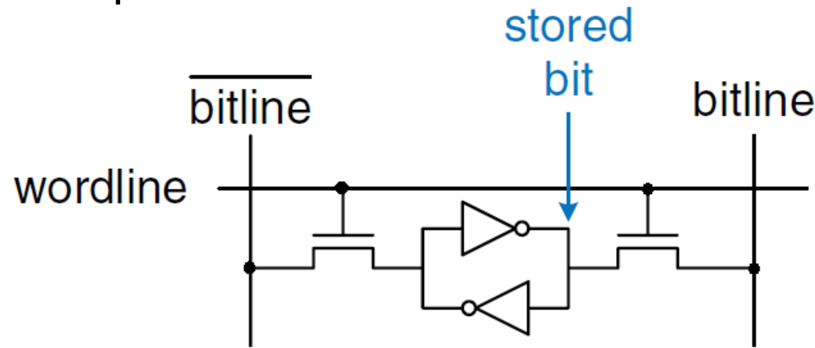
- Typy pamětí RWM (RAM), ROM, FLASH,
- Provedení RAM pamětí:
  SRAM (statická), **DRAM** (dynamická).

- RAM = *Random Access Memory – paměť s* ***libovolným*** *přístupem*

typ paměti	počet tranzistorů	plocha na 1 bit	dostupnost dat	latence
SRAM	cca 6	$< 0{,}1\ \mu m^2$	vždy	< 1ns – 5ns
DRAM	1	$< 0{,}001\ \mu m^2$	potřebuje refresh	desítky ns

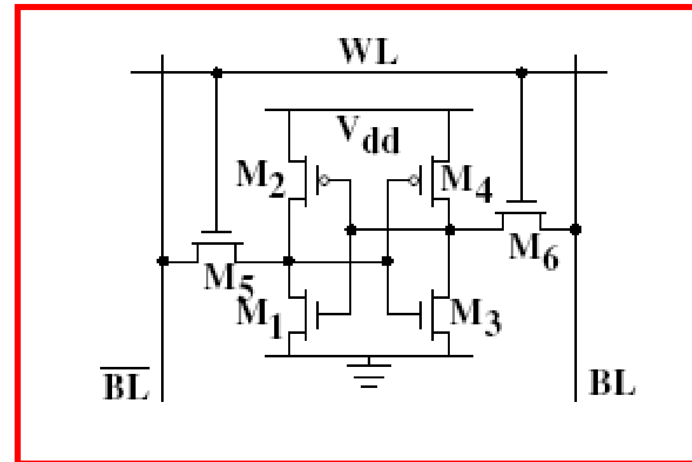
Typický čip a buňka SRAM

Princip:

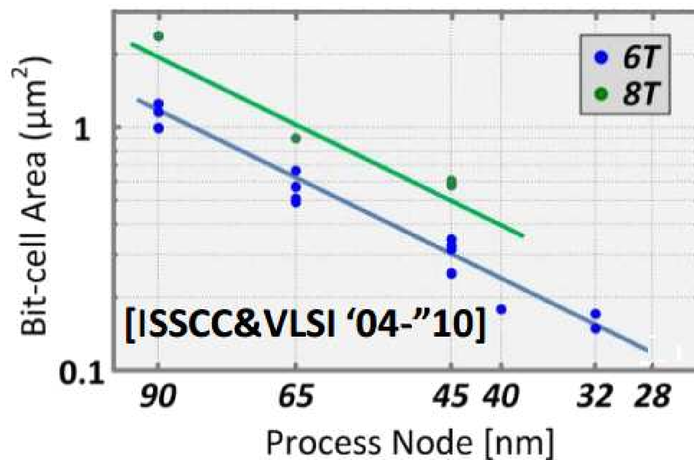


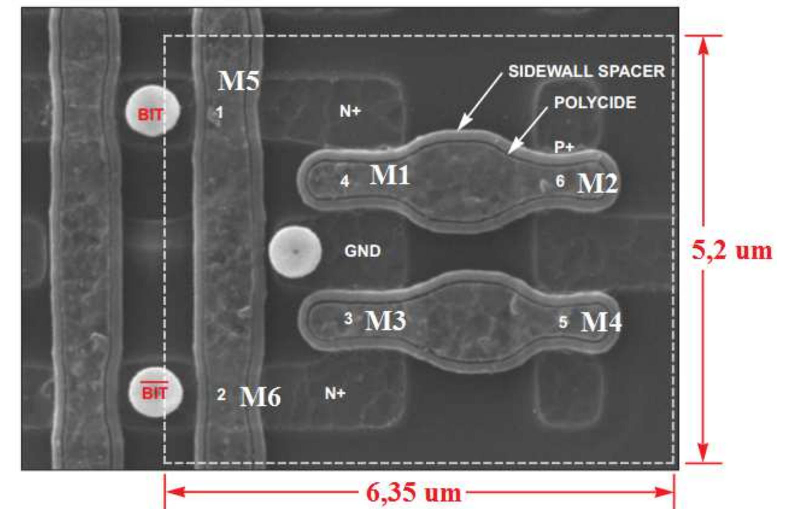
SRAM paměťová buňka

technologie CMOS



Plocha paměťové buňky:

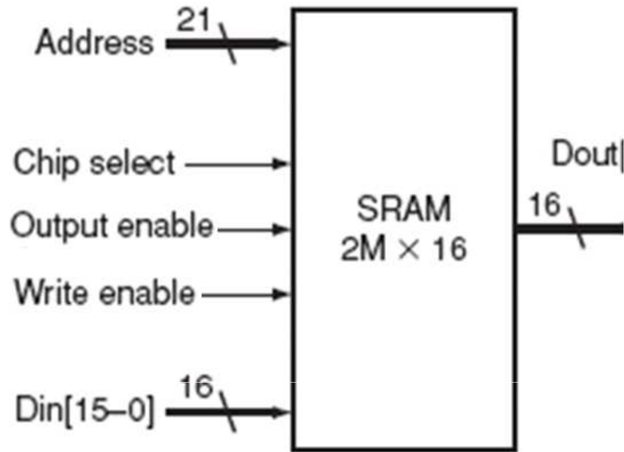




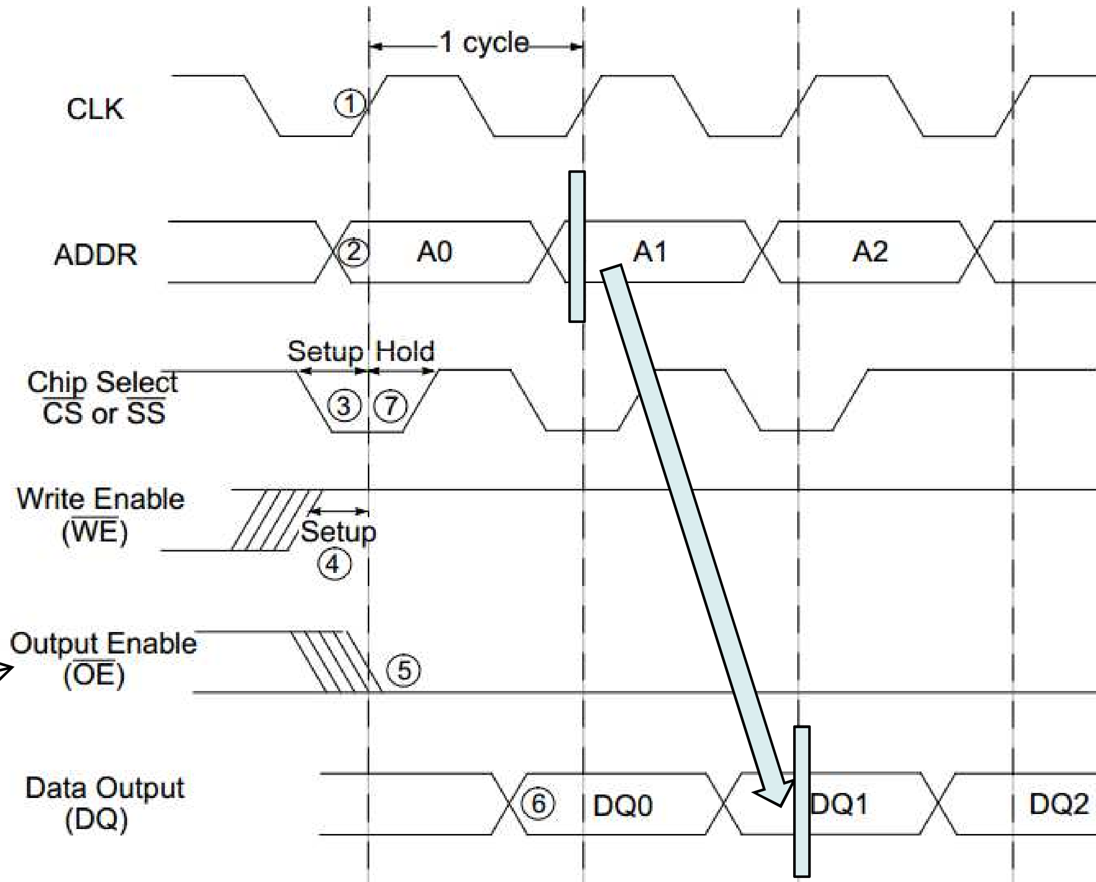
http://educypedia.karadimov.info/library/SEC08.PDF

Typický čip a buňka SRAM

Typický SRAM čip



Příklad čtení – typicky synchronní :

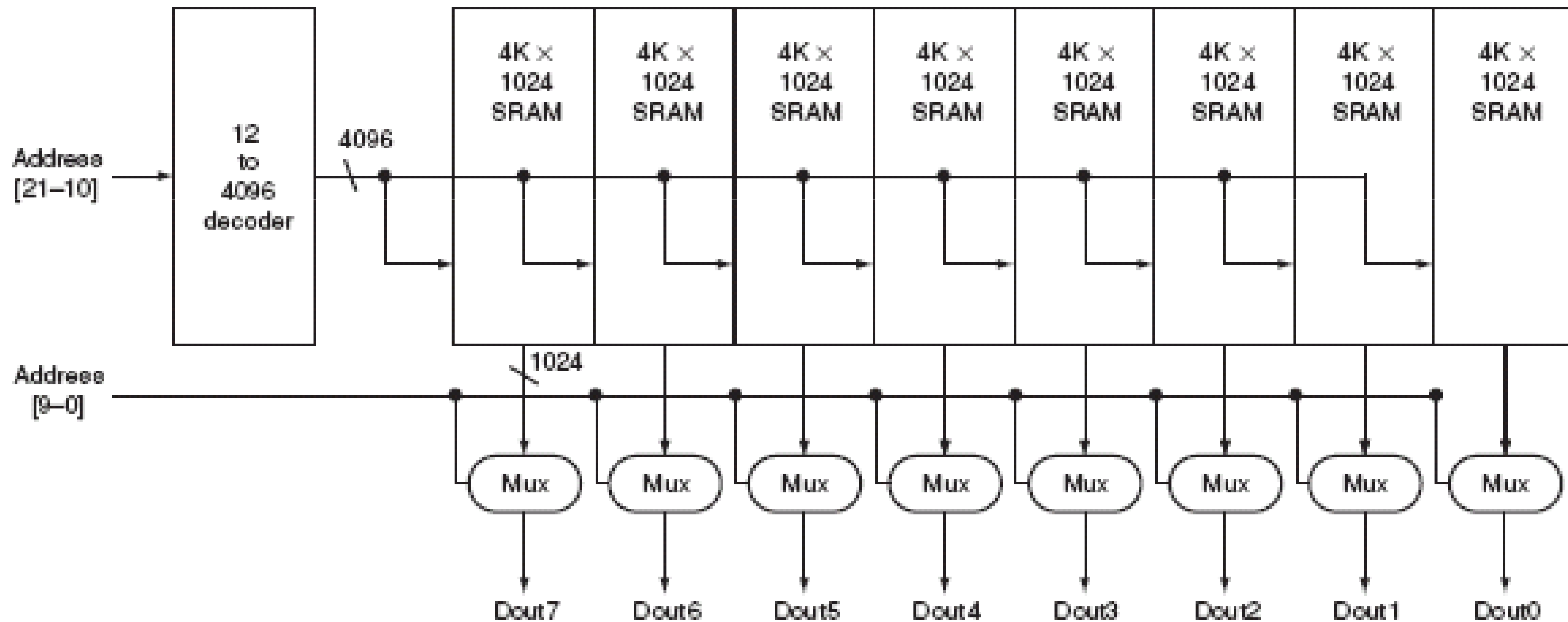


OE signál je asynchronní

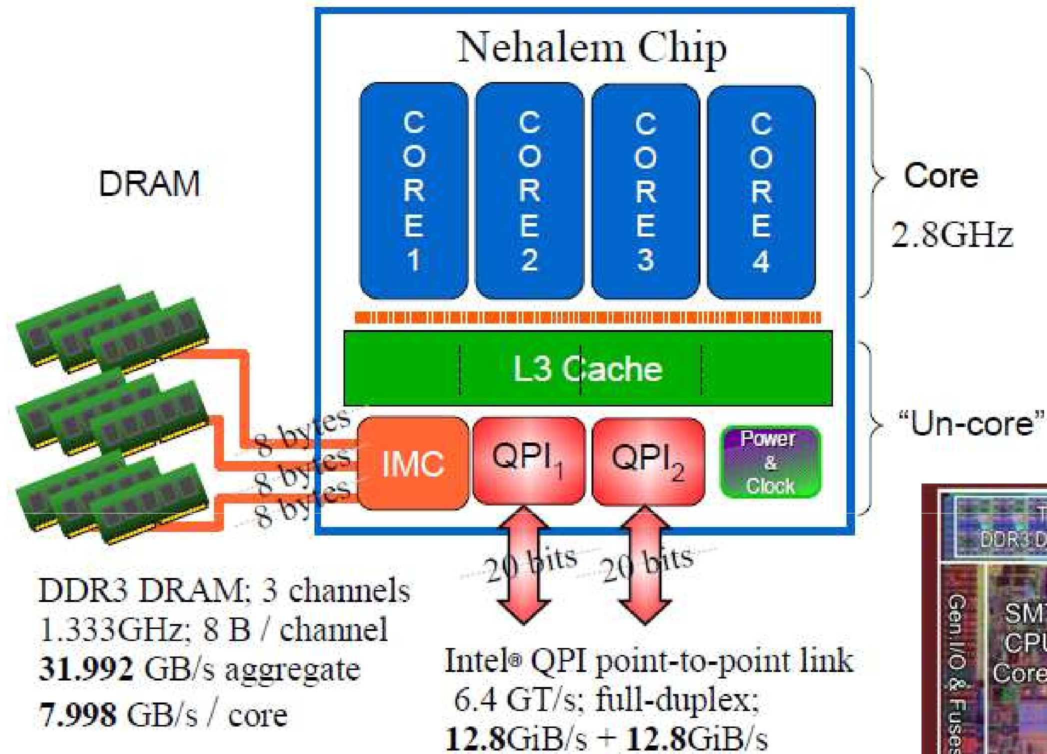
https://www.ece.cmu.edu/~ece548/localcpy/sramop.pdf

Typický čip a buňka SRAM

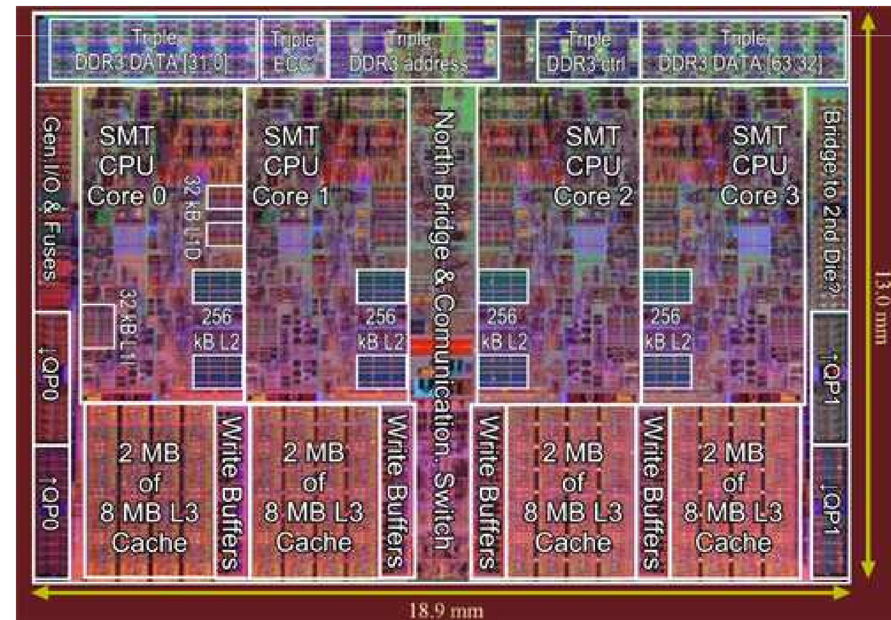
Větší paměť?



Příklad procesoru včetně cache
Harvardská architektura - Intel Nehalem

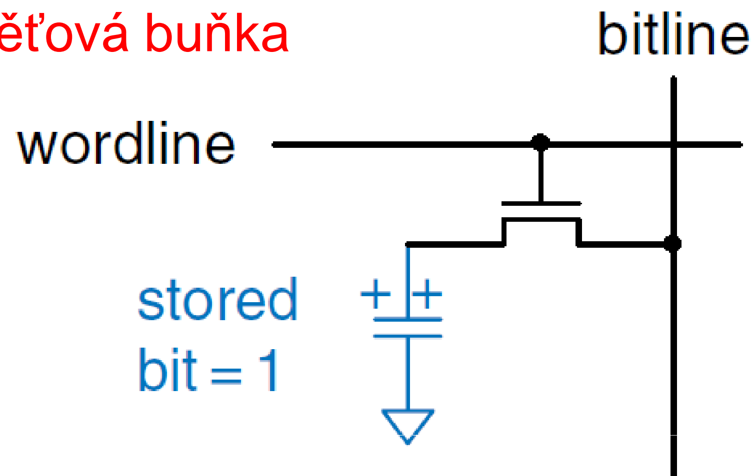


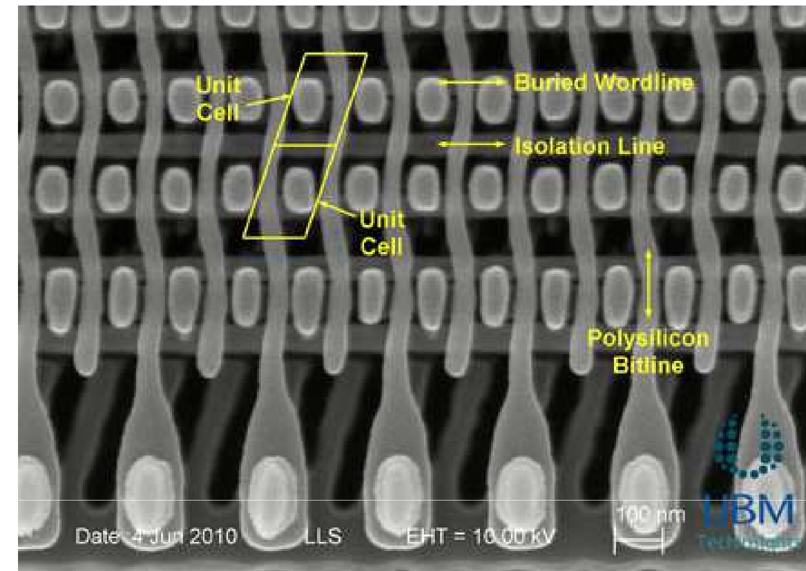
- IMC: integrated memory controller with 3 DDR3 memory channels,
- QPI: Quick-Path Interconnect ports
- Podívejte se na velkosti jednotlivých cache!!!



Detail paměťové buňky dynamické paměti

Jednotranzistorová dynamická paměťová buňka





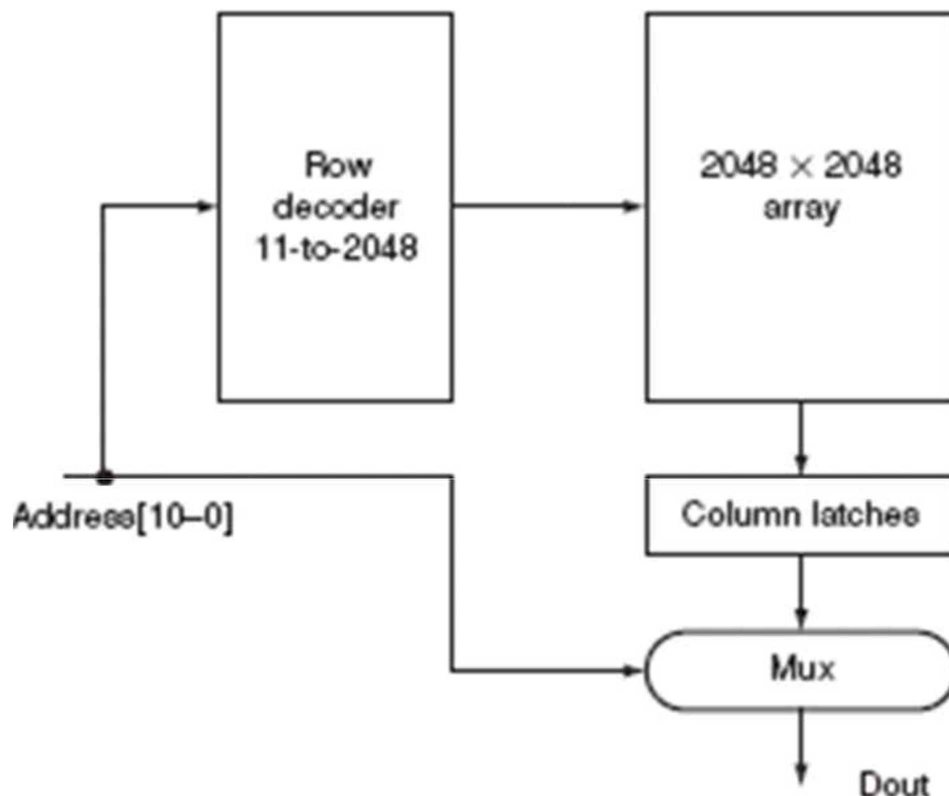
nMOS tranzistor představuje přepínač, který připojí (nebo ne) kondenzátor na vodič „bitline“. Připojení je řízeno vodičem „wordline“.

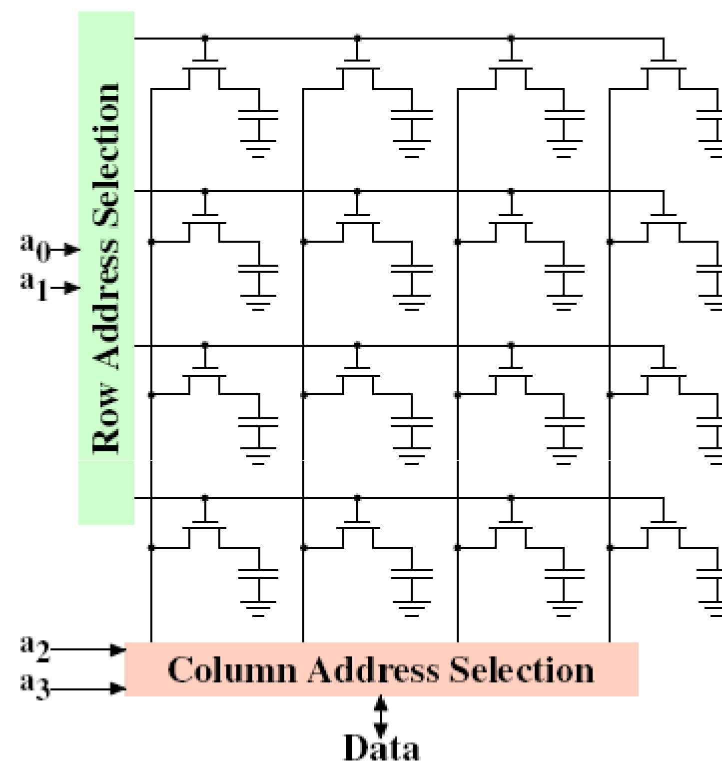
Proces čtení kondenzátor vybíjí. Proto musí být poté obnoven.

Občerstvování pamětí (refresh) – náboj se z kondenzátoru samovolně ztrácí. Nezbytná pracovní fáze dynamické paměti. Negativně ovlivňuje (prodlužuje) průměrnou vybavovací dobu.

Obrázek: http://www.eetimes.com/document.asp?doc_id=1281315

Vnitřní organizace čipu DRAM paměti





4M × 1 DRAM čip je uvnitř realizován
jako pole 2048 × 2048 1b paměťových buněk

Vývoj DRAM paměťových čipů v čase

Rok	Kapacita	Cena[$]/GB	Doba přístupu [ns]
1980	64 KB	1 500 000	250
1983	256 KB	500 000	185
1985	1 MB	200 000	135
1989	4 MB	50 000	110
1992	16 MB	15 000	90
1996	64 MB	10 000	60
1998	128 MB	4 000	60
2000	256 MB	1 000	55
2004	512 MB	250	50
2007	1 GB	50	40
2015	16 GB	20	10

Parametry pamětí

- Značení: DDRx-yyyyy/PCx-zzzz CL-tRCD-tRP-tRAS-(CMD)

8GB KIT DDR3
1600MHz CL10 Blue Series
Operační paměť 2x4GB, PC3-12800,
CL10-10-10-30, napětí 1.5V

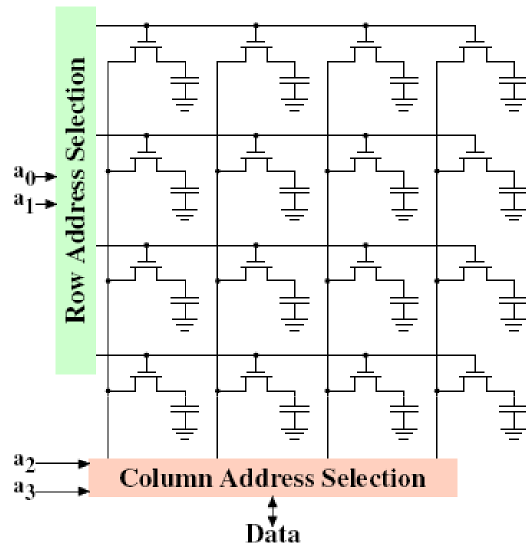
-15% ~~2 304,-~~
1 969,-
bez DPH 1 652,-

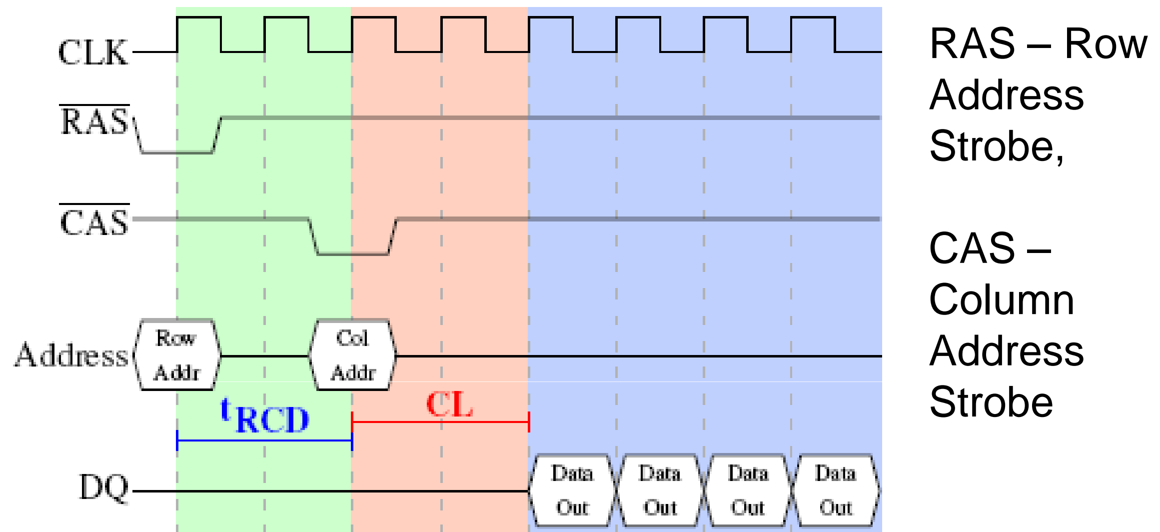
- Max Theoretical Transfer rate = clock * number of bits / 8
- Protože DIMM moduly přenášejí 64 bitů najednou, pak:
- Max Theoretical Transfer rate = clock * 8 (MB/s)

Parametry pamětí

- Značení: DDRx-yyyyy/PCx-zzzz CL-tRCD-tRP-tRAS-(CMD)

DDR3-1600/PC3-12800 CL10-10-10-30





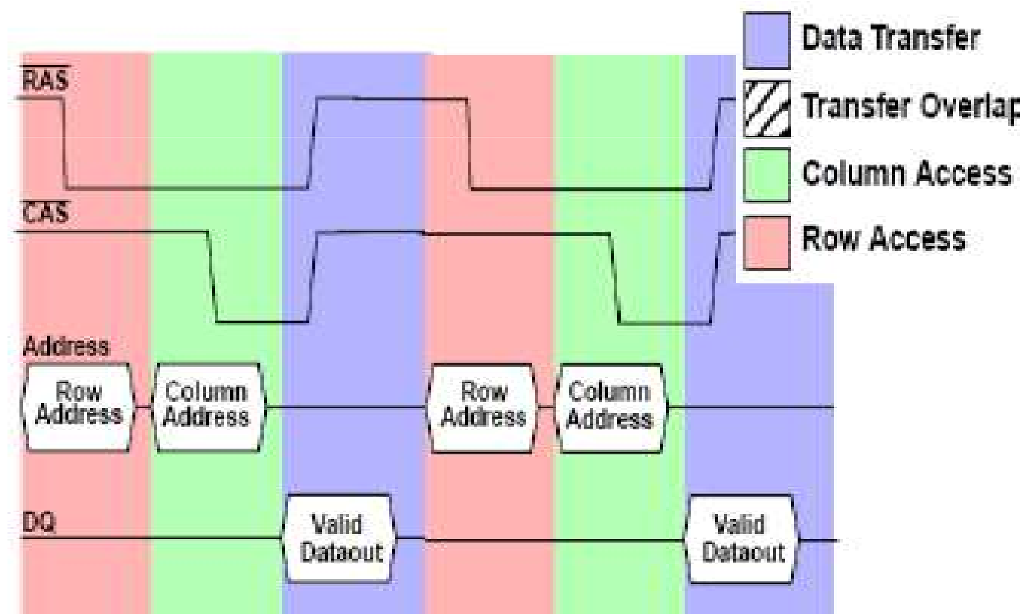
RAS – Row Address Strobe,

CAS – Column Address Strobe

- CL: CAS latency – viz obrázek
- tRCD: RAS to CAS delay – čas nutný mezi aktivací řádky a sloupce – viz obrázek
- tRP: RAS Precharge – čas mezi uzavřením jednoho řádku a aktivací nového
- tRAS: Active to Precharge delay – jak dlouho musí čekat než může být iniciován další přístup do paměti
- CMD: command rate – čas mezi aktivací paměti a prvním možným příkazem

Klasická DRAM – asynchronní rozhraní

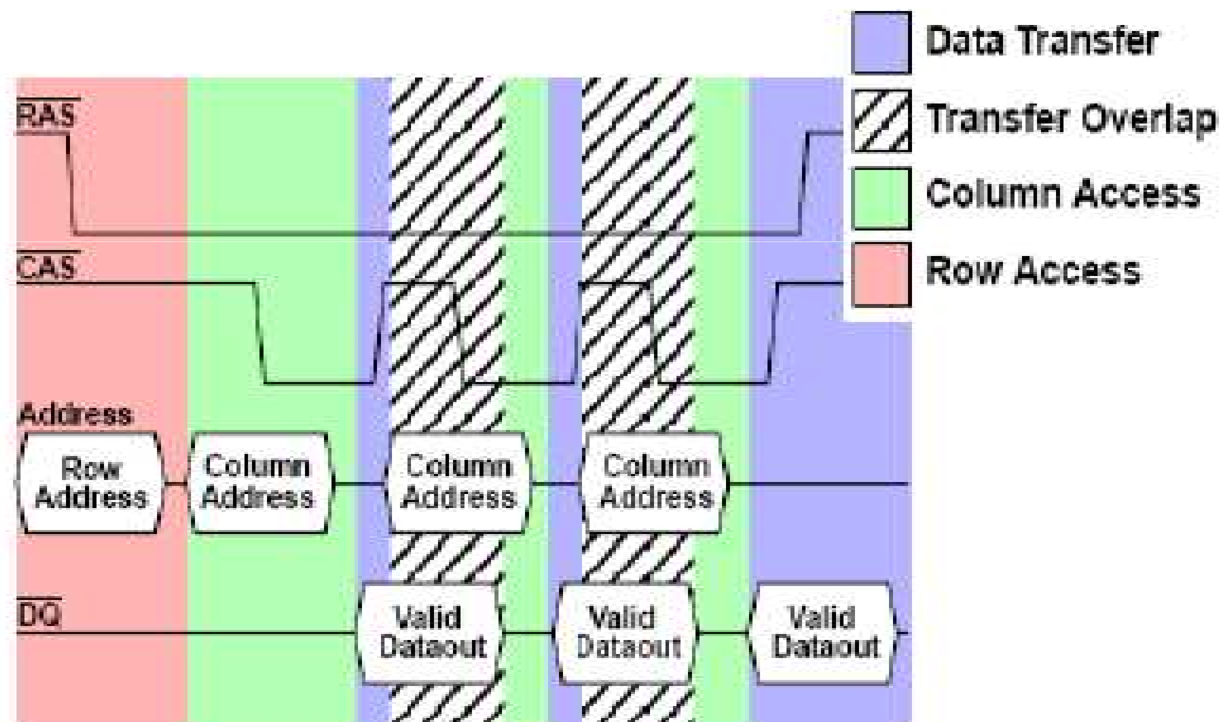
- Důvod rozdělení adresy na 2 části byl dán malým počtem pinů původních DRAM pouzder.
- Toto rozdělení se dodnes zachovává, ačkoli pouzdro už není problém. Uneslo by více vývodů…



RAS – Row Address Strobe,
CAS – Column Address Strobe

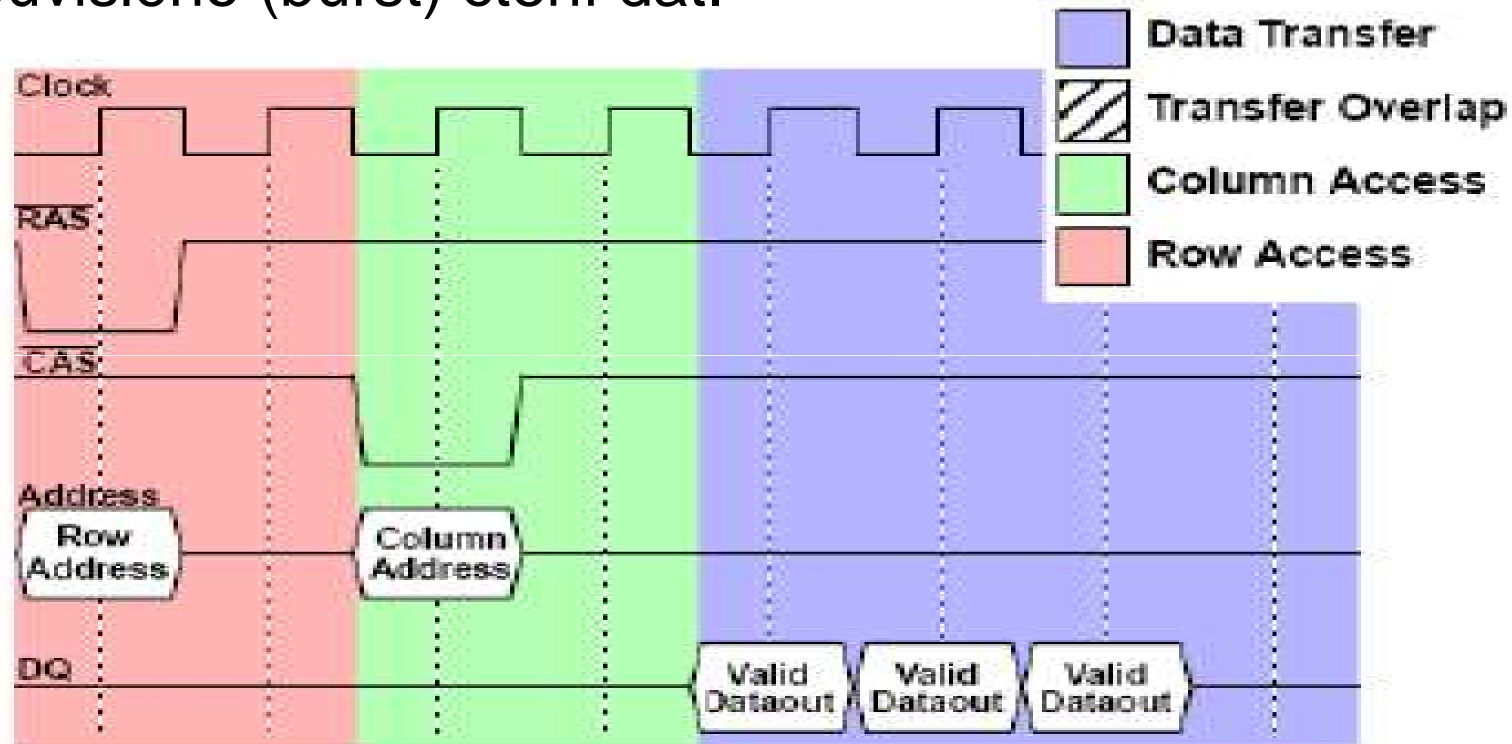
EDO DRAM – cca 1995

- EDO DRAM má registr na výstupu, což umožní překrýt následující CAS s čtením předchozích dat.



SDRAM – konec 90.let – synchronní DRAM

- SDRAM čip obsahuje čítač, který umožňuje nastavit délku souvislého (burst) čtení dat.

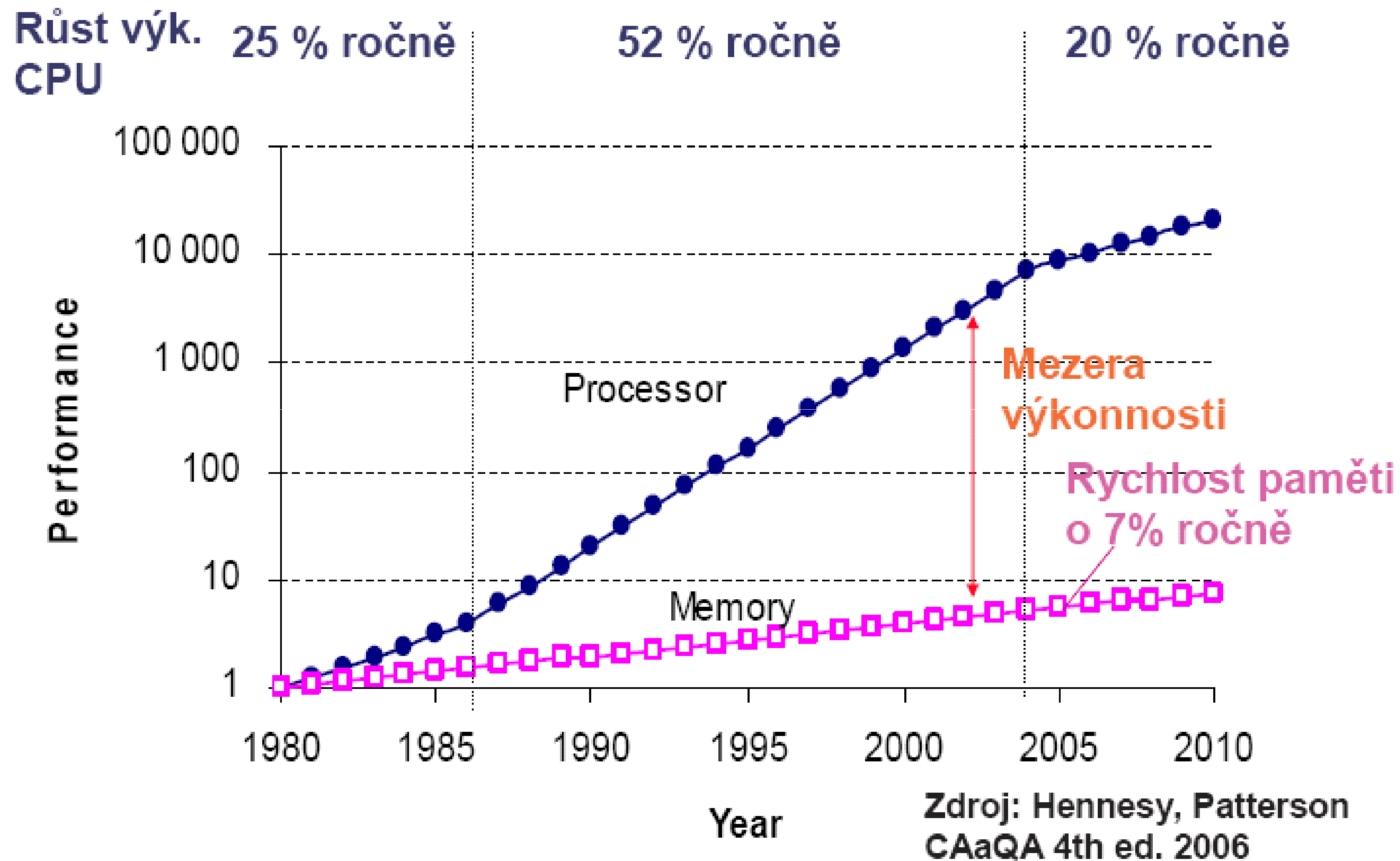


- Co z toho plyne? Náhodné „poskakování“ po paměti není zrovna optimální…

SDRAM – paměť současnosti

- SDRAM – frekvence **I/O bus** až 100 MHz, 2.5V.
- DDR SDRAM – použití obou hran CLK při přenosu dat, 2.5V. (odteď začíná být pojem „frekvence“ zavádějící)
- DDR2 SDRAM – snížení spotřeby použitím napětí 1.8V, frekvence až 400 MHz.
- DDR3 SDRAM – snížení spotřeby použitím napětí 1.5V (1.35V), frekvence 800-2400 MHz.
- DDR4 SDRAM - napětí 1.2V , frekvence 1600-3200 MHz
- Dále ještě existují paměti a moduly RAMBUS, které ovšem mají zcela odlišné rozhraní i způsob použití.
- Všechny tyto inovace vylepšují propustnost čtení dat, nikoli latenci čtení první položky.

Disproporce ve výkonu proc x pam, Moorův zákon



Bubble sort – již znáte z cvičení

```
int pole[5]={5,3,4,1,2};
int main()
{
    int N = 5,i,j,tmp;
    for(i=0; i<N; i++)
        for(j=0; j<N-1-i; j++)
            if(pole[j+1]<pole[j])
            {
                tmp = pole[j+1];
                pole[j+1] = pole[j];
                pole[j] = tmp;
            }
    return 0;
}
```

Jakou využitelnou vlastnost mají naše programy?

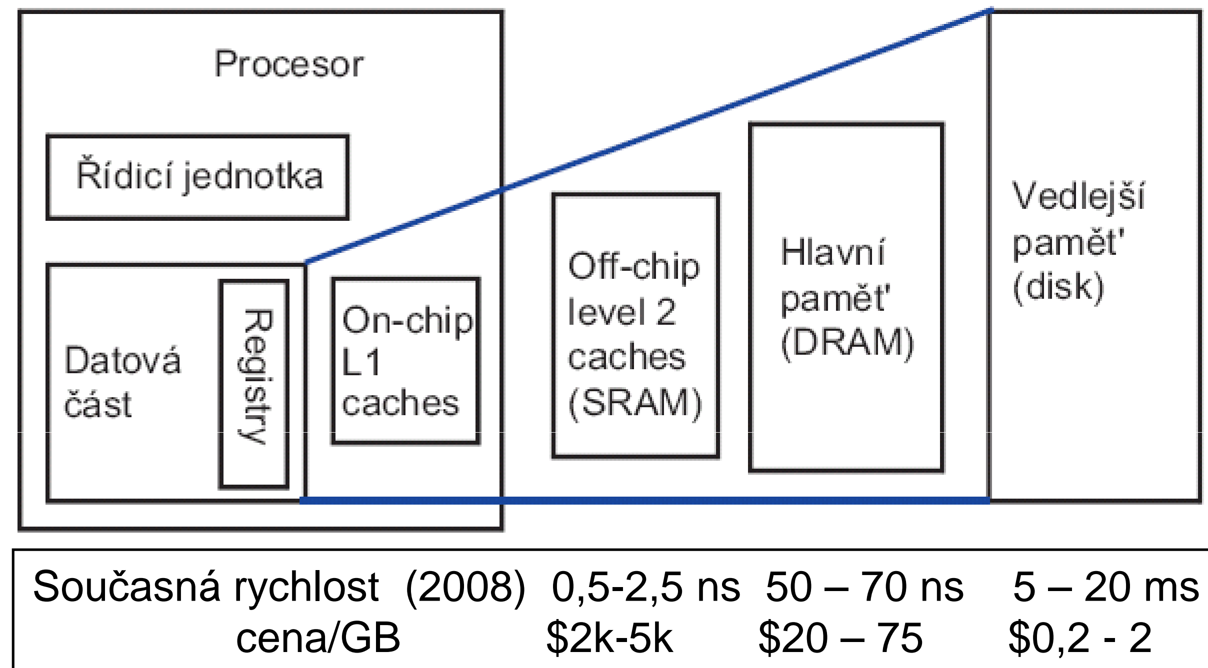
Paměťová hierarchie – základní principy

- 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).

Co z uvedeného plyne?

- Je výhodné uspořádat paměťový prostor hierarchicky – paměťová hierarchie.
- Všechny potřebné informace uchovávejte v sekundární paměti.
- Položky nedávno používané a blízké zkopírujte do (menší) paměti implementované DRAM.
  - Operační paměť.
- Položky ještě častěji používané (i ty jim blízké) zkopírujte do menší a rychlejší SRAM.
  - Skrytá paměť.

Paměťová hierarchie



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

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.

Řešení disproporce rychlosti? Skrytá paměť (SP) – cache.

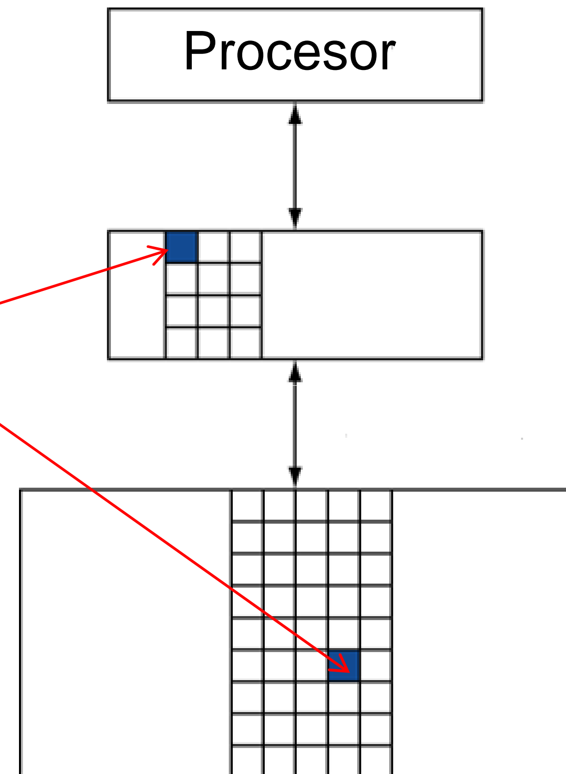
Cache, česky keš

- Nebo-li **skrytá paměť**
- je označení pro vyrovnávací paměť používanou ve výpočetní technice.

- Zařazujeme ji mezi dva subsystémy s různou rychlostí. Vyrovnává se jí rychlost přístupu k informacím.
- Účelem skryté paměti je urychlit přístup k často používaným datům na „pomalých“ médiích jejich překopírováním na média rychlá.

Terminologie kolem skryté paměti

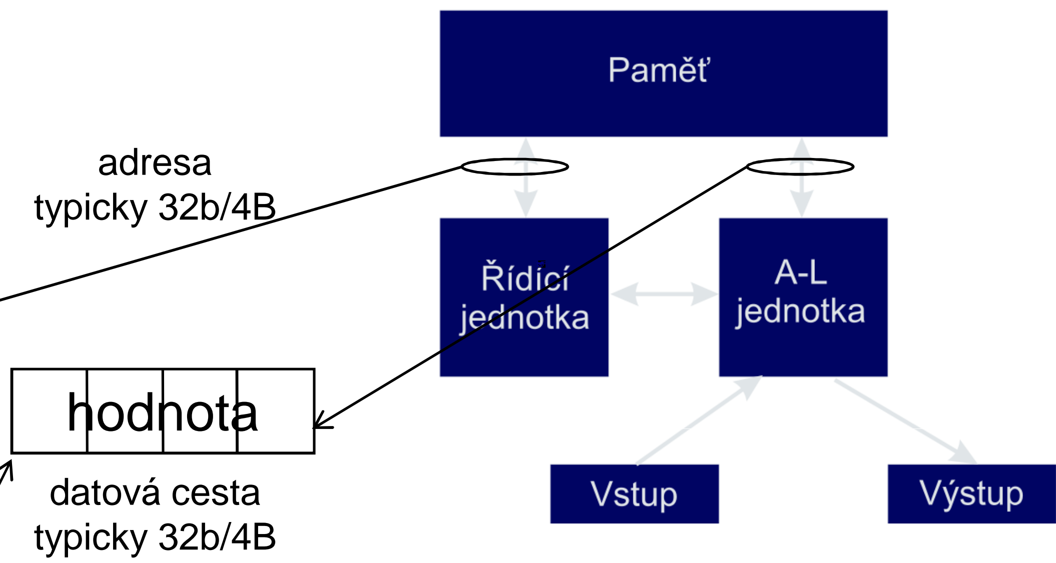
- **Cache hit** pojmenování situace, kdy požadovaná hodnota ve skryté paměti (cache) **je**.
- **Cache miss**, opak. **Není** tam.
- **Cache line** nebo **Cache block** – základní kopírovatelná jednotka mezi hierarchickými úrovněmi.
- V praxi se velikost Cache Line pohybuje od 8B do 1KB, typicky 64B.



Co je to adresní prostor? Adresa/hodnota

Address	Data
11...11111100	mem[0xFFFFFFFC]
11...11111000	mem[0xFFFFFFF8]
11...11110100	mem[0xFFFFFFF4]
11...11110000	mem[0xFFFFFFF0]
11...11101100	mem[0xFFFFFFEC]
11...11101000	mem[0xFFFFFFE8]
11...11100100	mem[0xFFFFFFE4]
11...11100000	mem[0xFFFFFFE0]
⋮	⋮
00...00100100	mem[0x00000024]
00...00100000	mem[0x00000020]
00...00011100	mem[0x0000001C]
00...00011000	mem[0x00000018]
00...00010100	mem[0x00000014]
00...00010000	mem[0x00000010]
00...00001100	mem[0x0000000C]
00...00001000	mem[0x00000008]
00...00000100	mem[0x00000004]
00...00000000	mem[0x00000000]

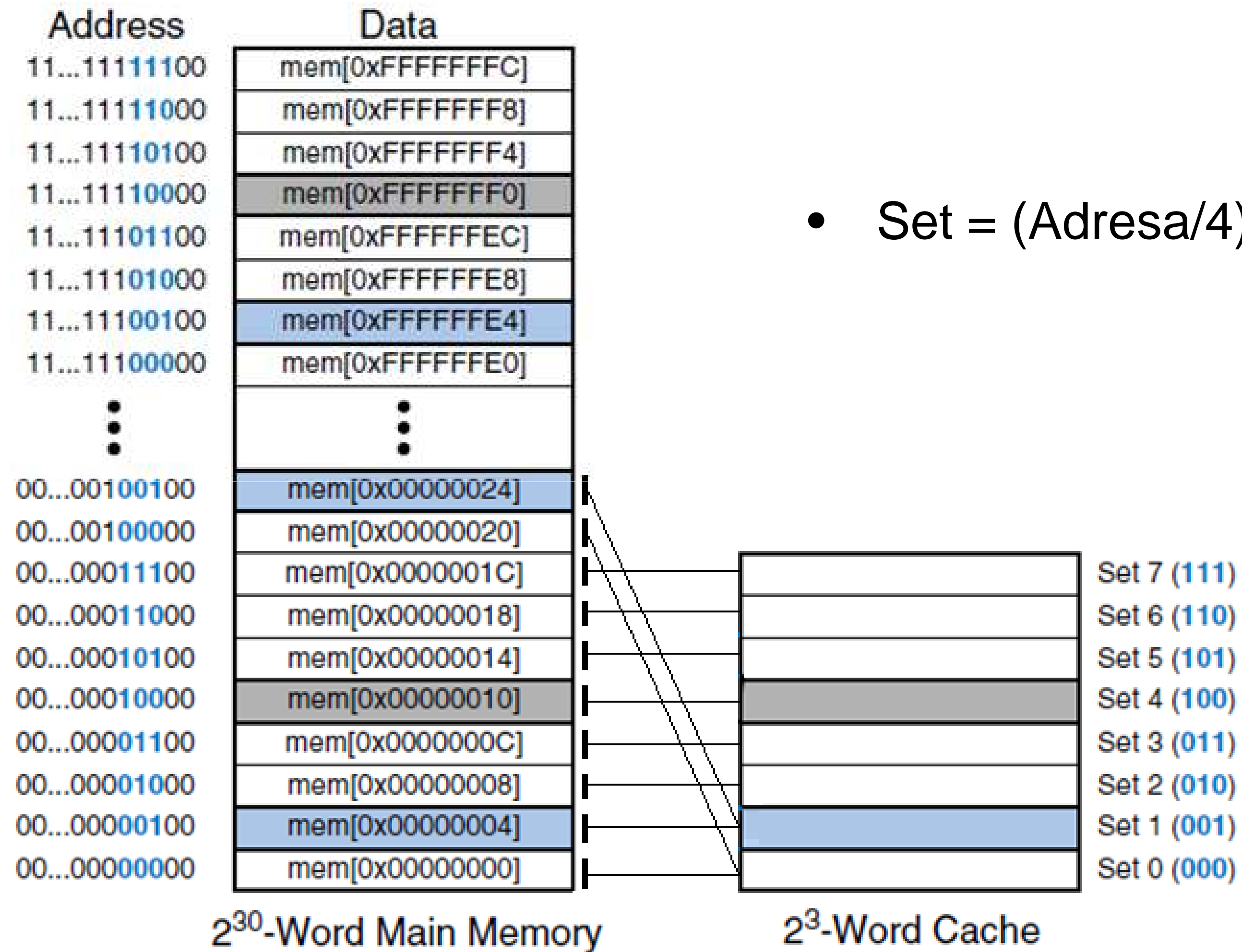
2^{30}-Word Main Memory



n	2↑n
8	256 různých buněk
16	64K (K=1024)
…	……
32	4G (4096M, M=K↑2)

velikost paměťové buňky typicky 1B

Přímo mapovaná cache

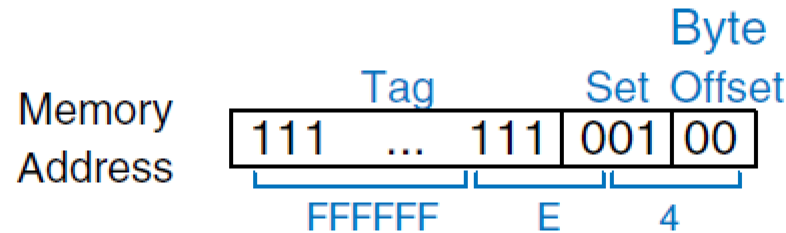


- Set = (Adresa/4) mod 8

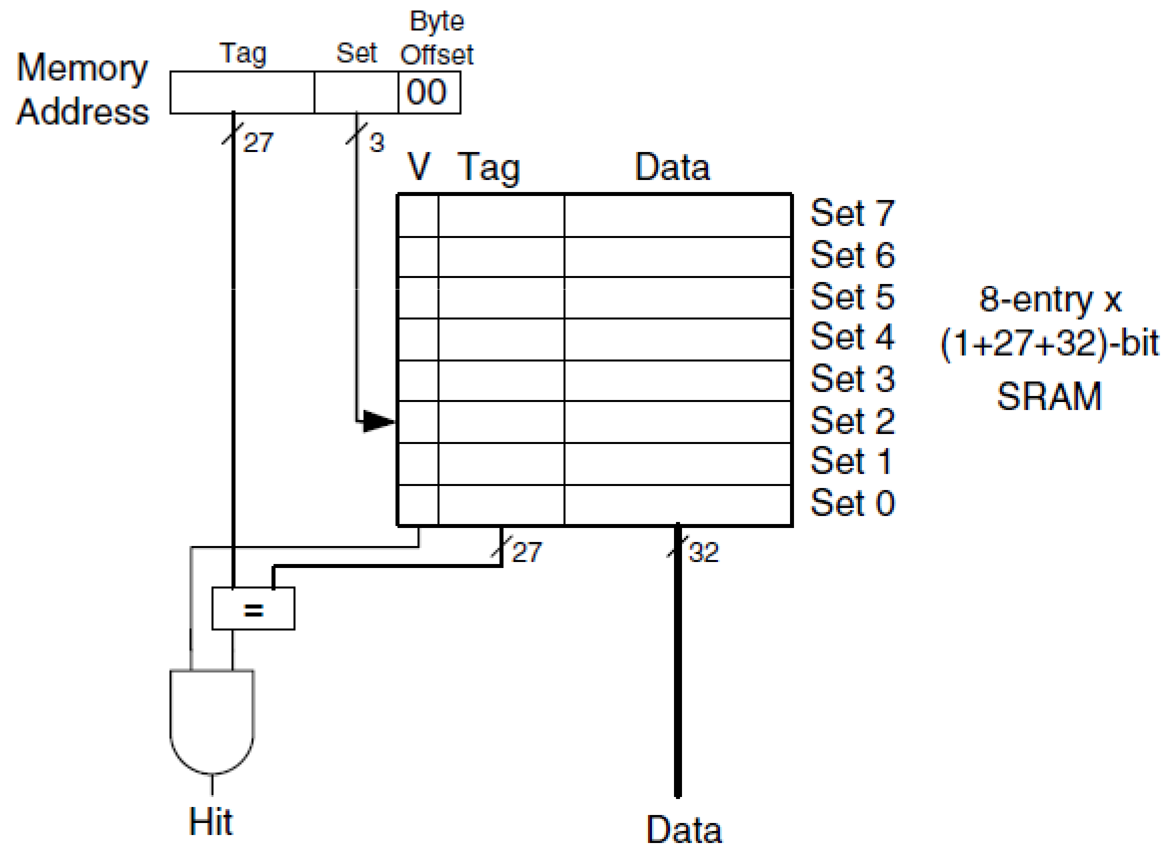
Příklad

- Mějme cache o velikosti 8-mi bloků. Kam se do ní umístí data z adresy 0xF0000014?
  - Přímo mapované, nebo
  - S omezeným stupněm asociativity N=2 (2-cestná, 2-way cache), nebo
  - Plně asociativní.

Přímo mapovaná cache



přímo mapovaná cache: one block in each set



Capacity – C

Number of sets – S

Block size – b

Number of blocks – B

Degree of associativity – N

C = 8 (8 words),

S = B = 8,

b = 1 (one word in the block),

N = 1

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 příslušné/ných adrese/ách.
- **Validity bit** – bit platnosti. Indikuje, zda je obsah pole Data vůbec platný.
- **Dirty bit** – rozšiřující pole v obsahu paměti. Indikuje, že v cache (cache) je **jiná hodnota**, než v paměti hlavní.

V	Další bity, např. D	Tag	Data

Přímo mapovaná cache – větší velikost bloku???

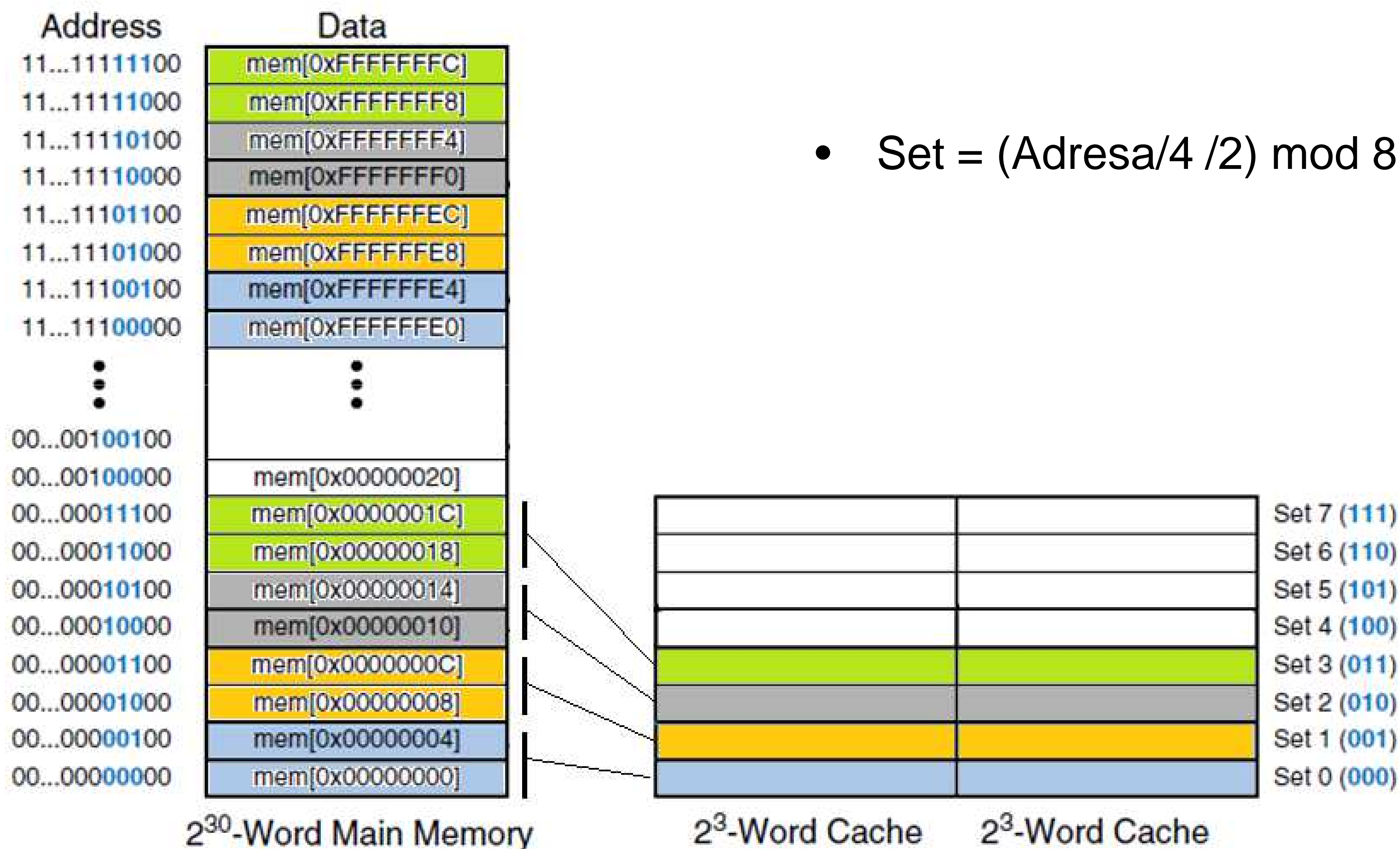
přímo mapovaná cache: one block in each set

b = 2

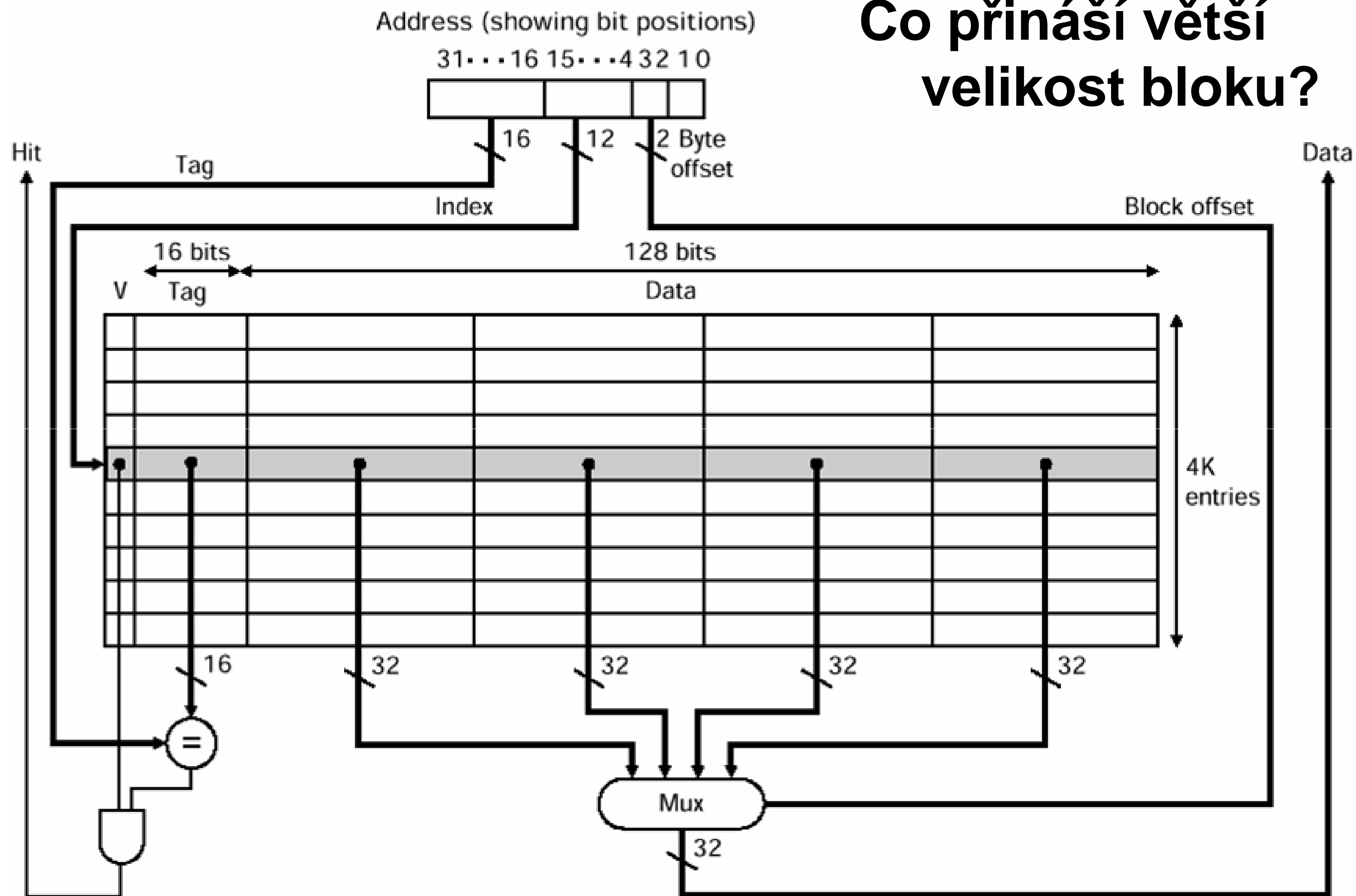
(two words in the block),

Přímo mapovaná cache – větší velikost bloku???

- Set = (Adresa/4 /2) mod 8



Přímo mapovaná skrytá paměť – velikost bloku 4 slova



Co přináší větší velikost bloku?

Cache s omezeným stupněm asociativity N=2

Co přináší zvětšení stupně asociativity?

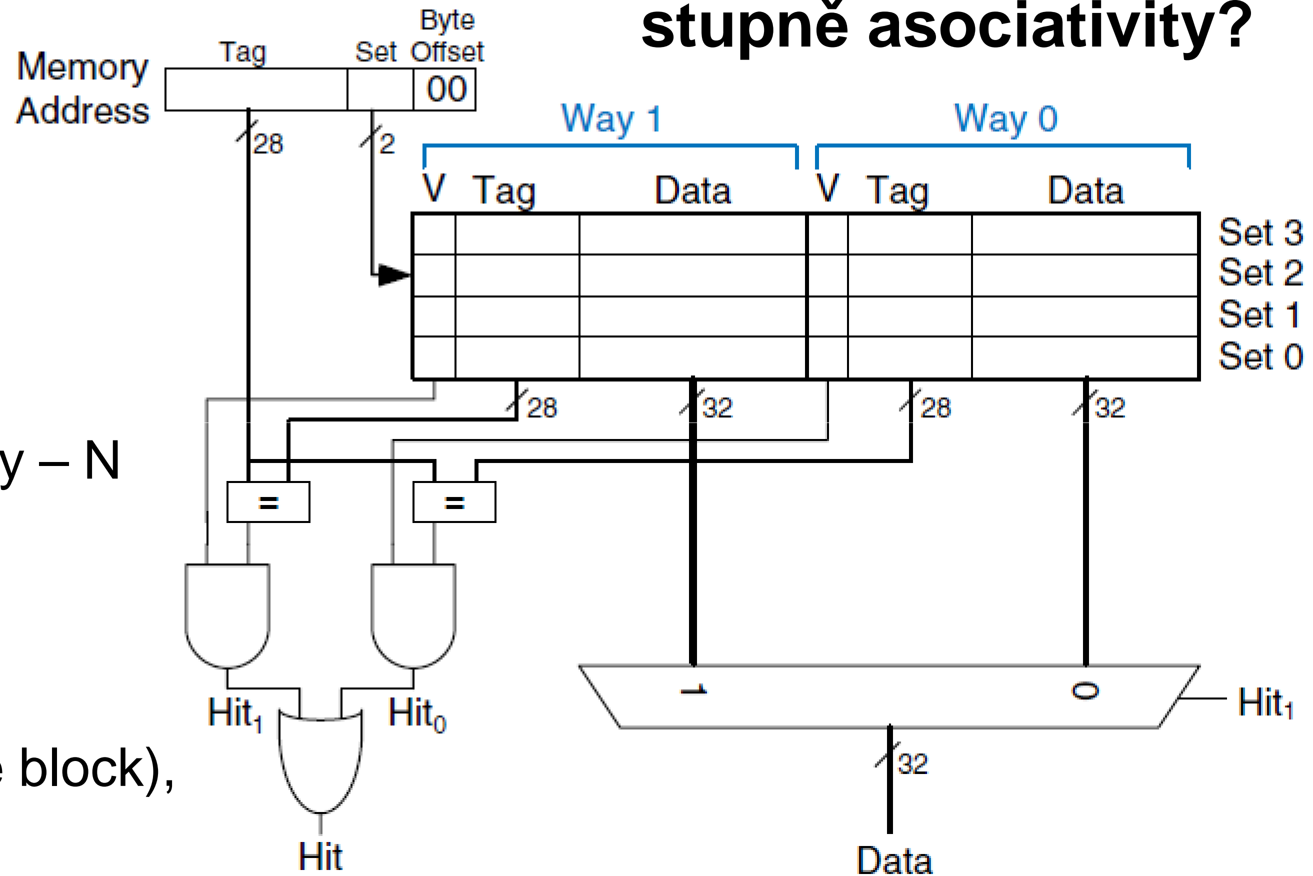
Capacity – C

Number of sets – S

Block size – b

Number of blocks – B

Degree of associativity – N



C = 8 (8 words),

S = 4,

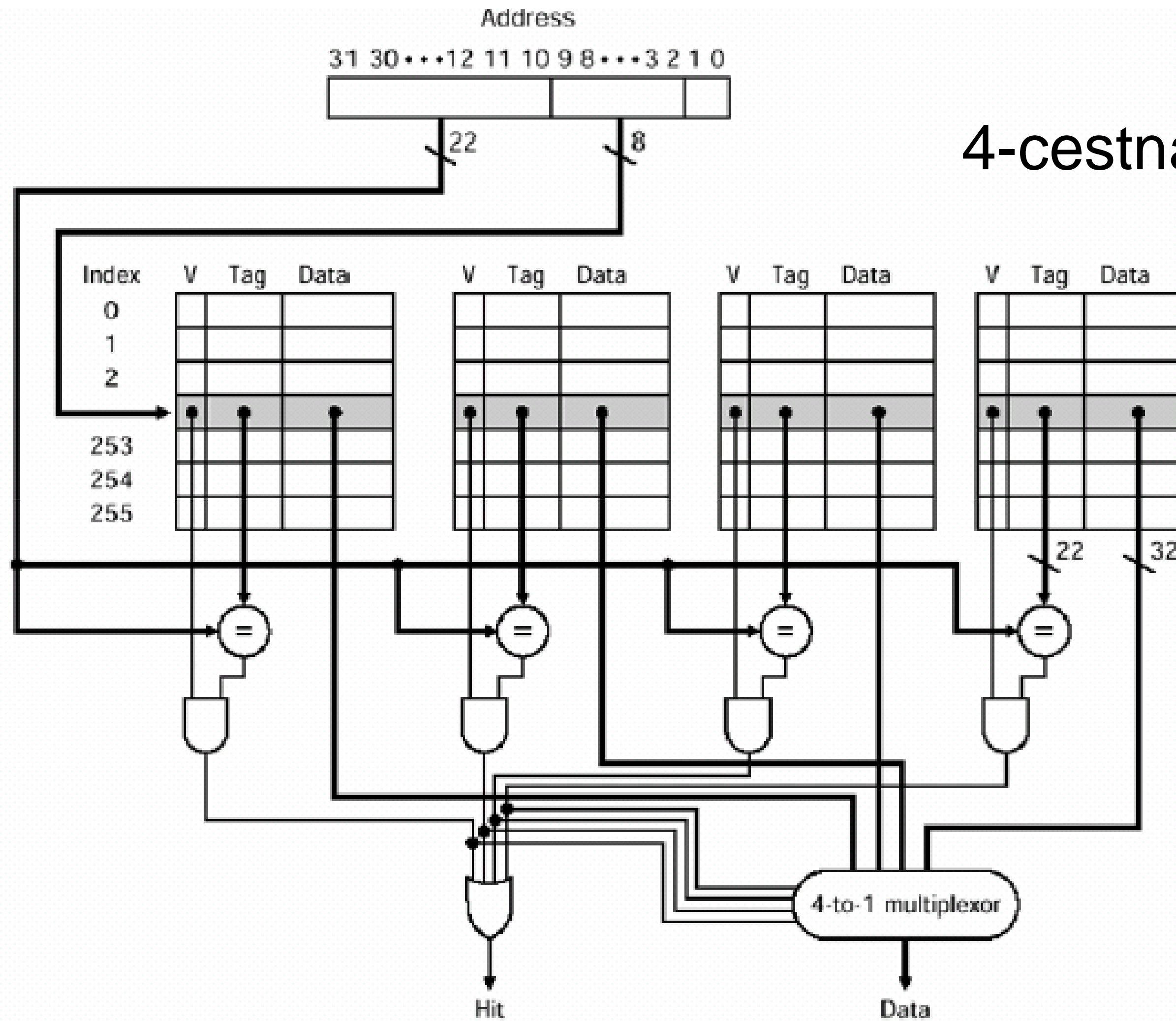
b = 1 (one word in the block),

B = 8

N = 2

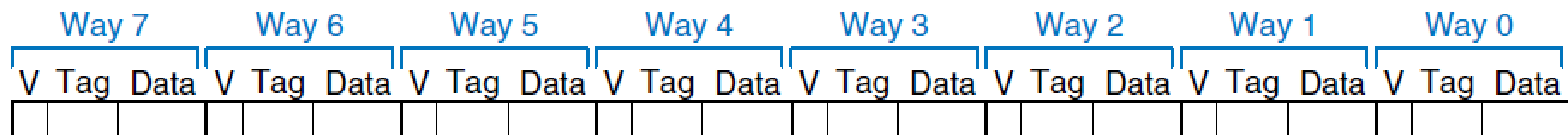
Cache s omezeným stupněm asociativity N=4

4-cestná, 4-way cache



Plně asociativní cache

- Plně asociativní cache obsahuje jenom jeden set, stupeň asociativity je roven počtu bloků (N=B). Adresa paměti se může mapovat kamkoliv.
- …je jiné pojmenování pro B-cestně asociativní cache s jedním setem
- … 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.

Diskuze k plně asociativní cache

- Šířka pole Tag odpovídá šířce adresy,
- Každý řádek cache obsahuje tolik jednobitových kompa-rátorů, 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.

- Takováto cache je ale velmi drahá.
- Proto existují a používají se cache
  - Přímo mapované,
  - S omezeným stupněm asociativity.

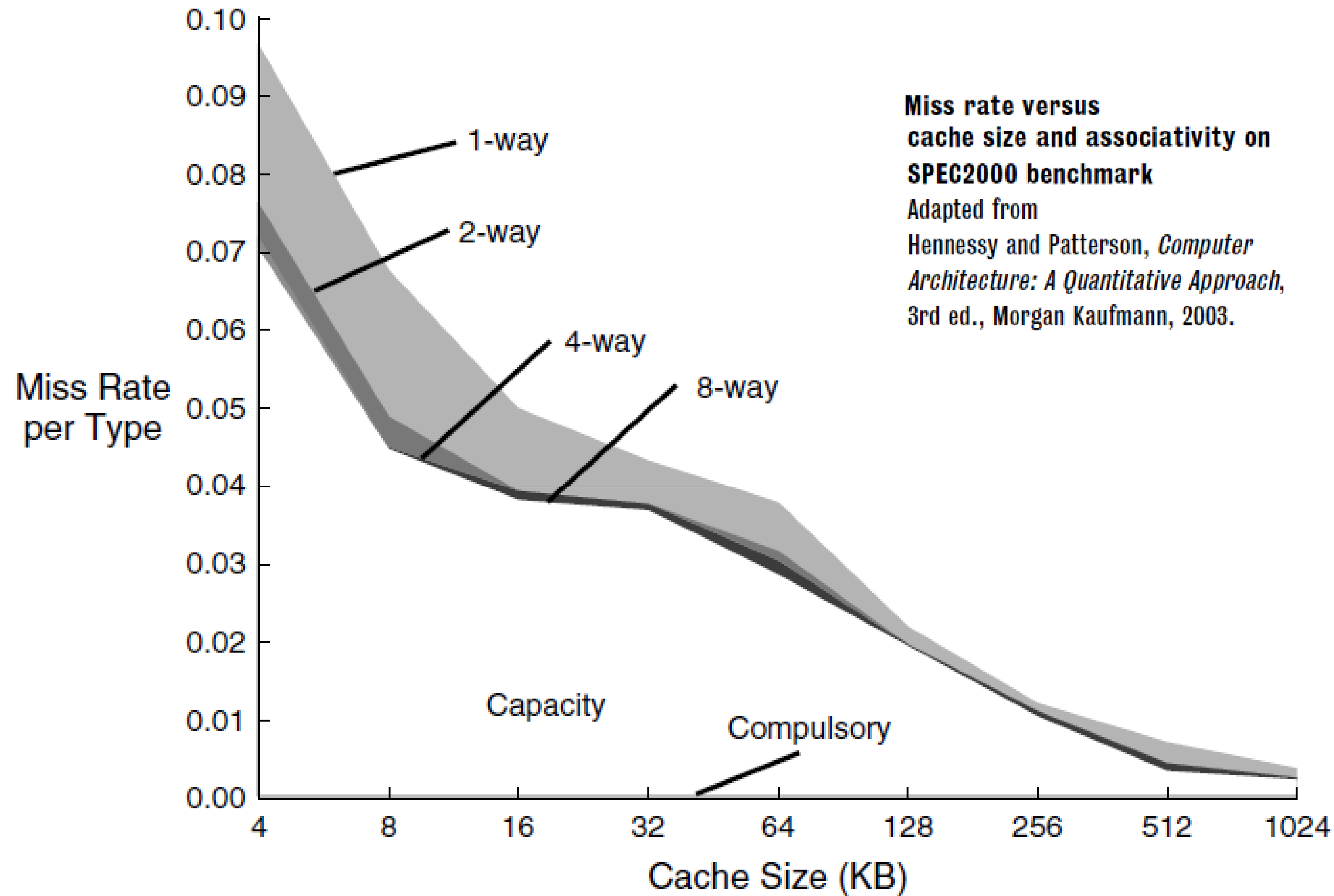
Terminologie kolem skryté paměti II.

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

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

  MissPenalty – může být vypočtena rekurentně

Porovnání

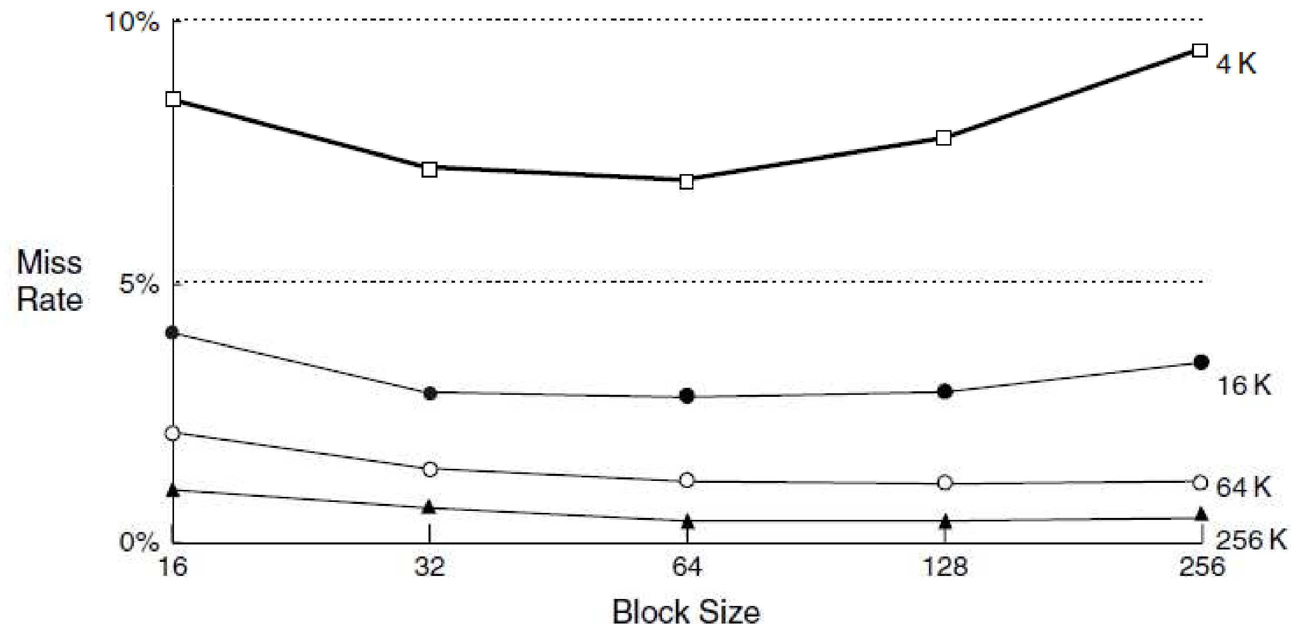


Pamatujte: 1. miss rate není vlastností cahce!

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.

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

- Data se nejprve musí z hlavní paměti přečíst. Jenže: co když je cache plná?

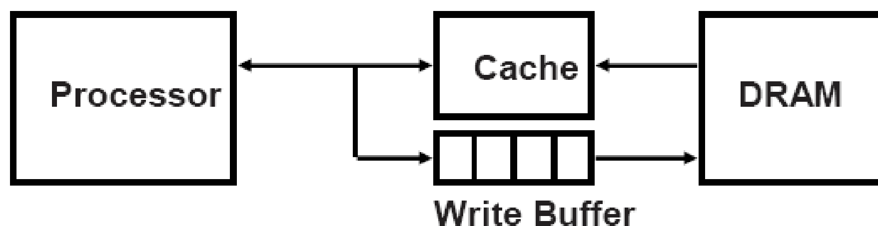
Strategie uvolňování bloků/řádek cache

- **Náhodná** (Random) – vybere se libovolný blok. Snadné, ale hloupé.
- **LRU** (Least Recently Used) musíme znát informace o posledním použití tohoto bloku (jedná se o celé číslo).
- **LFU** (Least Frequently Used), ke každému bloku si pamatujeme informace o tom, jak často byl blok požadován.
- **ARC** (Adaptive Replacement Cache), ve které se vhodným způsobem kombinuje strategie LRU a LFU.
- Pozn: U přímo-mapované cache nemáme na výběr…

- **Tím jsme zodpověděli otázku KTERÝ blok se odstraní… Ale KDY dojde k zápisu do paměti? – viz další slide**

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 Inf pole). Ke skutečnému zápisu dat do hlavní paměti dojde až v okamžiku případného rušení příslušného řádku cache, kdy hrozí jejich ztráta.
- **Dirty bit** – rozšiřující pole v obsahu paměti. Indikuje, že v cache (cache) je **jiná hodnota**, než v paměti hlavní.

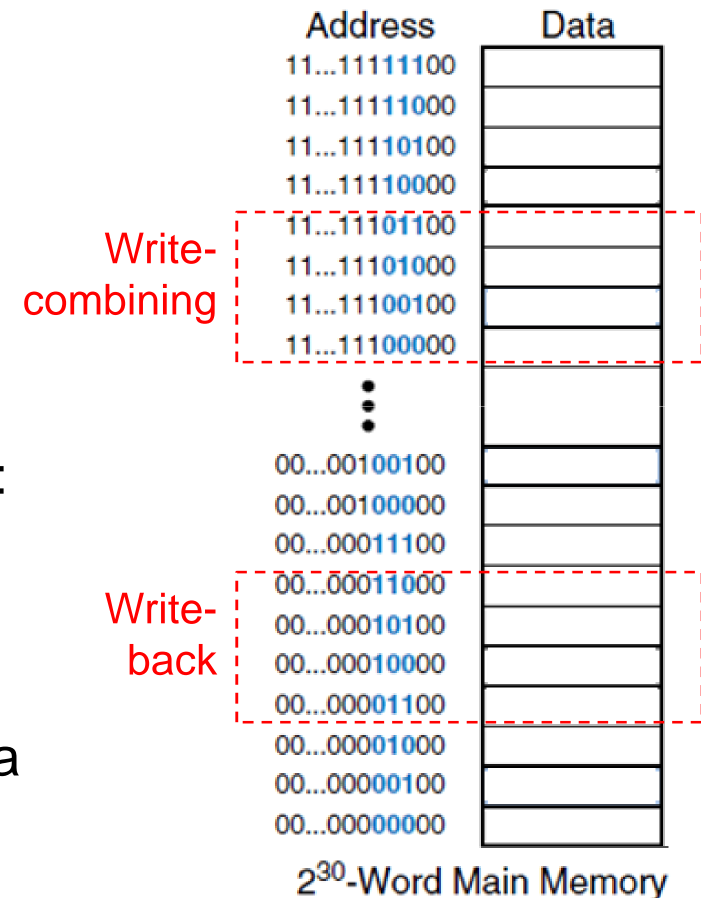


V	Další bity, např. D	Tag	Data

Řešení situace **Zápis dat** procesorem do paměti

Ještě existují další tři důležité strategie:

- **Write-combining** (data jsou posílána do **write combine bufferu** aby mohla být později najednou zapsána; negarantuje pořadí (weakly ordered memory); příklad: při zápisu do RAM grafické karty)
- **Uncacheable** (typicky když daná adresa není adresou do RAM => nechceme zapisovat do RAM, ale do jiného zařízení, př: PCIe karta, kterému je dána daná adresa)
- **Write-protect**
- na x86 k určení dané strategie používáme **Memory Type Range Registers** (MTRR), na novějších CPU pak **Page Attribute Table** (PAT) - umožňuje nastavit režim pro každou tabulku stránek zvlášť



Trend - **Víceúrovňové SP**

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

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?

- 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++)
    …
```

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

X = Y * Z

```
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 úvodní příklad…)

Pochopili jste tuto přednášku?

- Dalším příkladem je násobení matic

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++)
     zT[i][j] = z[j][i];

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

Vygenerování transponované matice nás něco stojí.. Nicméně vyplatí se to!

Pochopili jste tuto přednášku?

- Dalším příkladem je násobení matic
  Ještě lépe je však použít tzv. blokové násobení.
  Idea: Rozdělme výpočet na submatice BxB, které se vejdou do cache.. => eliminace „capacity misses“

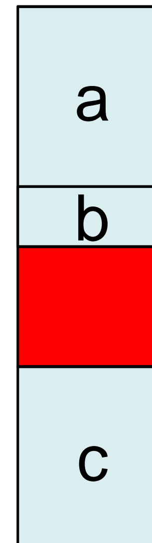
```
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i++)
      for (j = jj; j < min(jj+B-1,N); j++) {
      tmp = 0;
        for (k = kk; k < min(kk+B-1,N); k++)
           tmp += y[i][k]*z[k][j];
        x[i][j] = x[i][j] + tmp;
      }
```

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

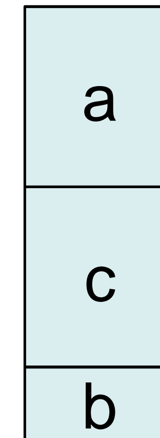
Pochopili jste tuto přednášku?

- **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';
```



Co je špatně? – viz. pahole

```
struct cheese {
	char name[17]; /* 0 17 */
		/* XXX 1 byte hole, try to pack */
	short age; /* 18 2 */
	char type; /* 20 1 */
		/* XXX 3 bytes hole, try to pack */
	int calories; /* 24 4 */
	short price; /* 28 2 */
		/* XXX 2 bytes hole, try to pack */
	int barcode[4]; /* 32 16 */
};		/* size: 48, cachelines: 1 */
		/* sum members: 42, holes: 3 */
		/* sum holes: 6 */
		/* last cacheline: 48 bytes */
```


Arnaldo Carvalho de Melo: The 7 dwarves: debugging information beyond gdb

Ponaučení

- Dávejte pozor na uspořádání prvků struktury
- Na začátek struktury dávejte nejkritičtější prvky (nejčastěji používané)
- Pokud přistupujete k prvkům struktury, snažte se zachovat pořadí v jakém jsou ve struktuře definovány
- Pro větší struktury, pravidla platí rovněž a lze je aplikovat nad velikostí cache line
- Další otázkou je, jaké položky vůbec mají být ve struktuře: OOP princip vs. rychlost

Pochopili jste tuto přednášku?

- **Data, ke kterým přistupujete ve stejnou dobu (krátce za sebou) uložte vedle sebe (seskupte).**
- **Data, ke kterým přistupujete často uložte vedle sebe (seskupte).**
- Někdy se je potřeba rovněž zamyslet nad zarovnáním dat v paměti – buď přímo v assembleru nebo v jazyce C – zkontrolujte si jestli Váš kompilátor zarovnává double na 8-byte hranici, pokud ne:
  - alokujte si kolik potřebujete + 4B (nebo i víc – dle zarovnání)
  - pomocí ANDu získejte zarovnanou adresu pro svá data, příklad:

```
double a[5];
double *p, *newp;
p = (double*)malloc ((sizeof(double)*5)+4);
newp = (p+4) & (-7);
```

- Viz také int posix_memalign(void **memptr, size_t align, size_t size);

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; /*přenos z paměti do cache a zápis 0*/
```

Přenos nastáva pouze při cache miss

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) /*přenos z paměti do cache a čtení*/
        array[j] = 0; /*zápis 0 pouze někdy*/
```

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

Obcházení cache může rovněž urychlit Vaše programy

- Pokud vyprodukujete data, která nejsou ihned použita (*non-temporal* write operation) není důvod je cacheovat
- To je mnohdy případ velkých datových struktur (matice apod.)
- Proč to vede k urychlení programu?

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);        A další…
```

Uloží data obsažena v „a“ na adresu „p“ bez vynucení účasti cache. Nicméně pokud již „p“ existuje v cache, cache bude aktualizována.

-> viz strategie **Write-combining**;

-> o finální vyprázdnění WC buffru se stará programátor, jinak HW

- Podrobnosti v: “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Optimalizujte často volané funkce

- Pokud často a zejména v rychlém sledu po sobě voláte tutéž funkci, optimalizujte ji! **Využijte k tomu i cache…**
- Příklad: Víme, že budeme potřebovat počítat odmocniny pouze celých čísel, ale velmi často pouze od 0 do 10.

```
double sqrt10(int i) {
    static const double lookup_table[] = {0, 1,
        sqrt(2), sqrt(3), 2, sqrt(5), sqrt(6),
        sqrt(7), sqrt(8), 3, sqrt(10)   };

    if(0 <= i && i <= 10)
        return lookup_table[i];
    else
        return sqrt(i);
}
```

Optimalizujte často volané funkce

- Příklad: Budeme volat funkci, která je často krát po sobě volaná se stejnými parametry…

```
double f(double x, double y) {
   return sqrt(x * sin(x) + y * cos(y)); }
```

Po optimalizaci:

```
double f(double x, double y) {
    static double prev_x = 0, prev_y = 0, result = 0;

    if (x == prev_x && y == prev_y)
        return result;
    prev_x = x;
    prev_y = y;
    result = sqrt(x * sin(x) + y * cos(y));
    return result;
}
```

Jak zjistit paramatry cache?

- Linux

```
#include <unistd.h>
long sysconf (int name);
```

Kde name:

_SC_LEVEL1_ICACHE_SIZE
_SC_LEVEL1_ICACHE_ASSOC
_SC_LEVEL1_ICACHE_LINESIZE   atd.

- Windows

GetLogicalProcessorInformation() -> SYSTEM_LOGICAL_PROCESSOR_INFORMATION a ta obsahuje CACHE_DESCRIPTOR