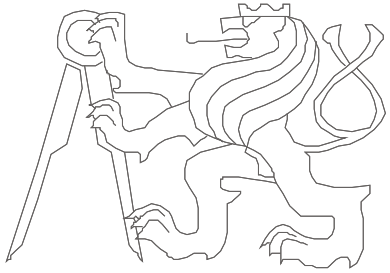
Pokročilé architektury počítačů

Paměť – část první
- úvod, realizace pamětí, cache, virtuální paměť



České vysoké učení technické, Fakulta elektrotechnická
Autor materiálu: Michal Štepanovský

Motivace pro přednášku z pohledu programátora?

Otázka 1.: Vykonávají programy to samé?

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

A:

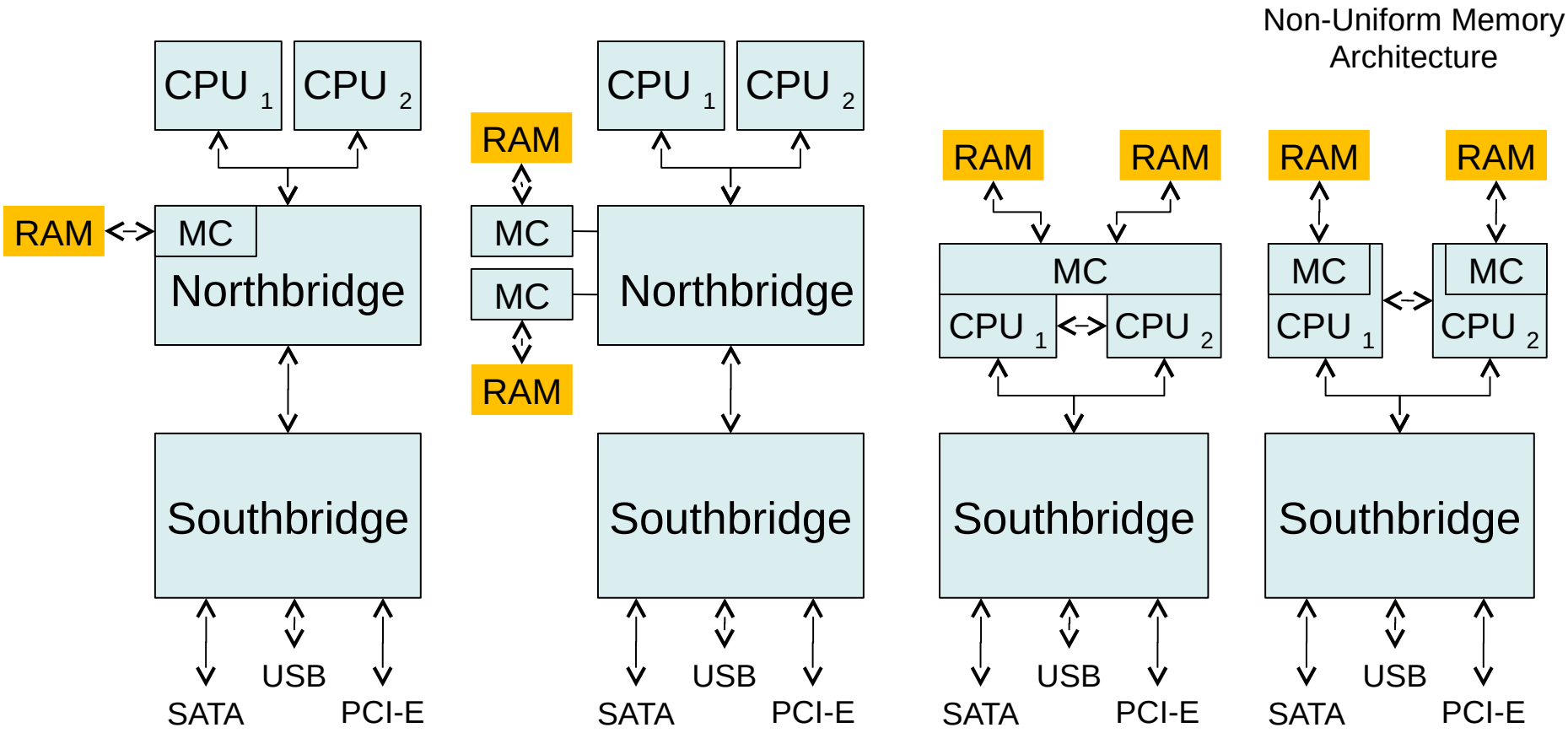
```
int matrix[M][N];
int i,j,sum=0;
…
for(i=0;i<M;i++)
  for(j=0;j<N;j++)
    sum+=matrix[i][j];
```

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];
```

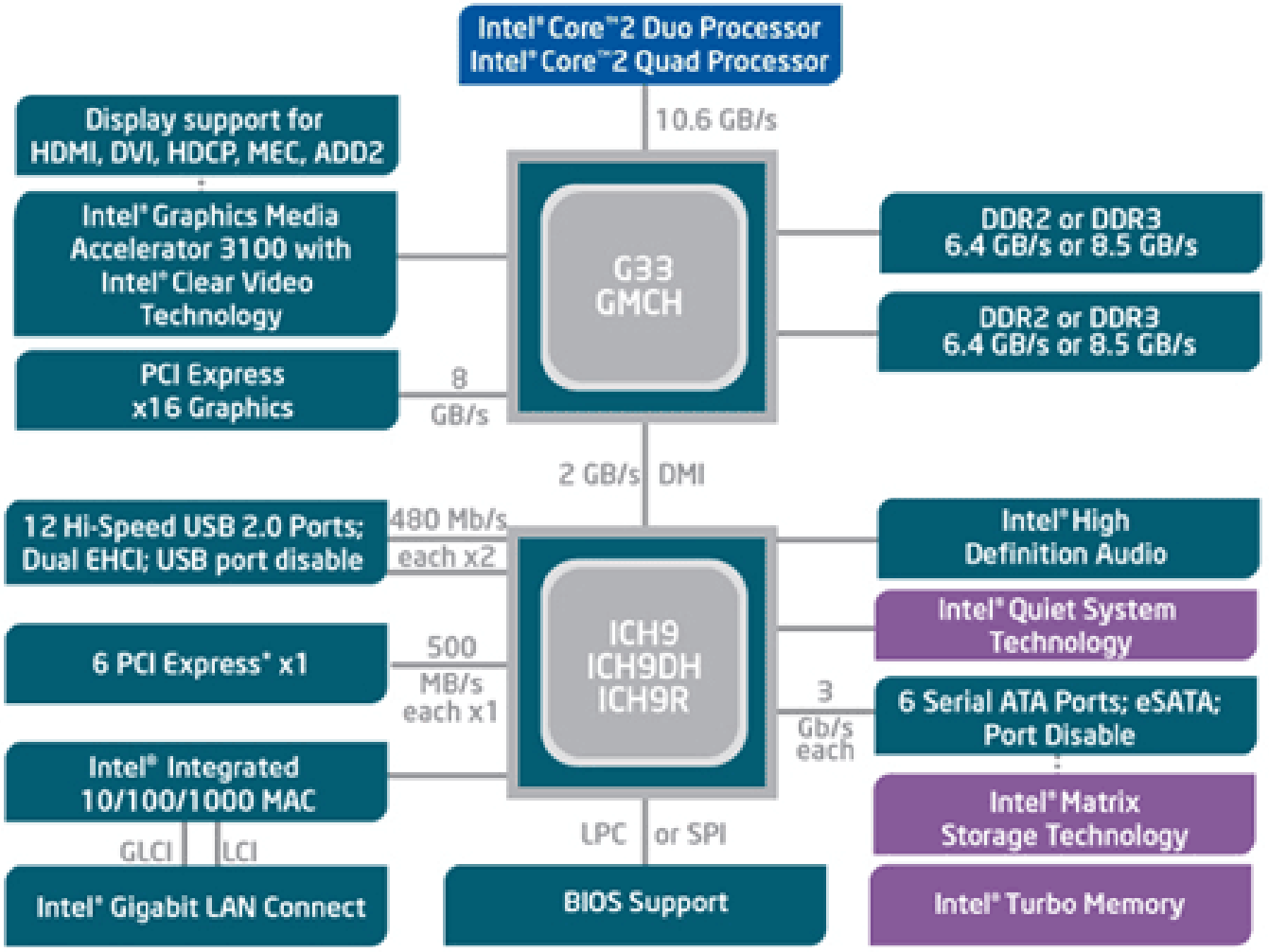
Lze doporučit výhodnější způsob procházení matice?

Architektura počítače



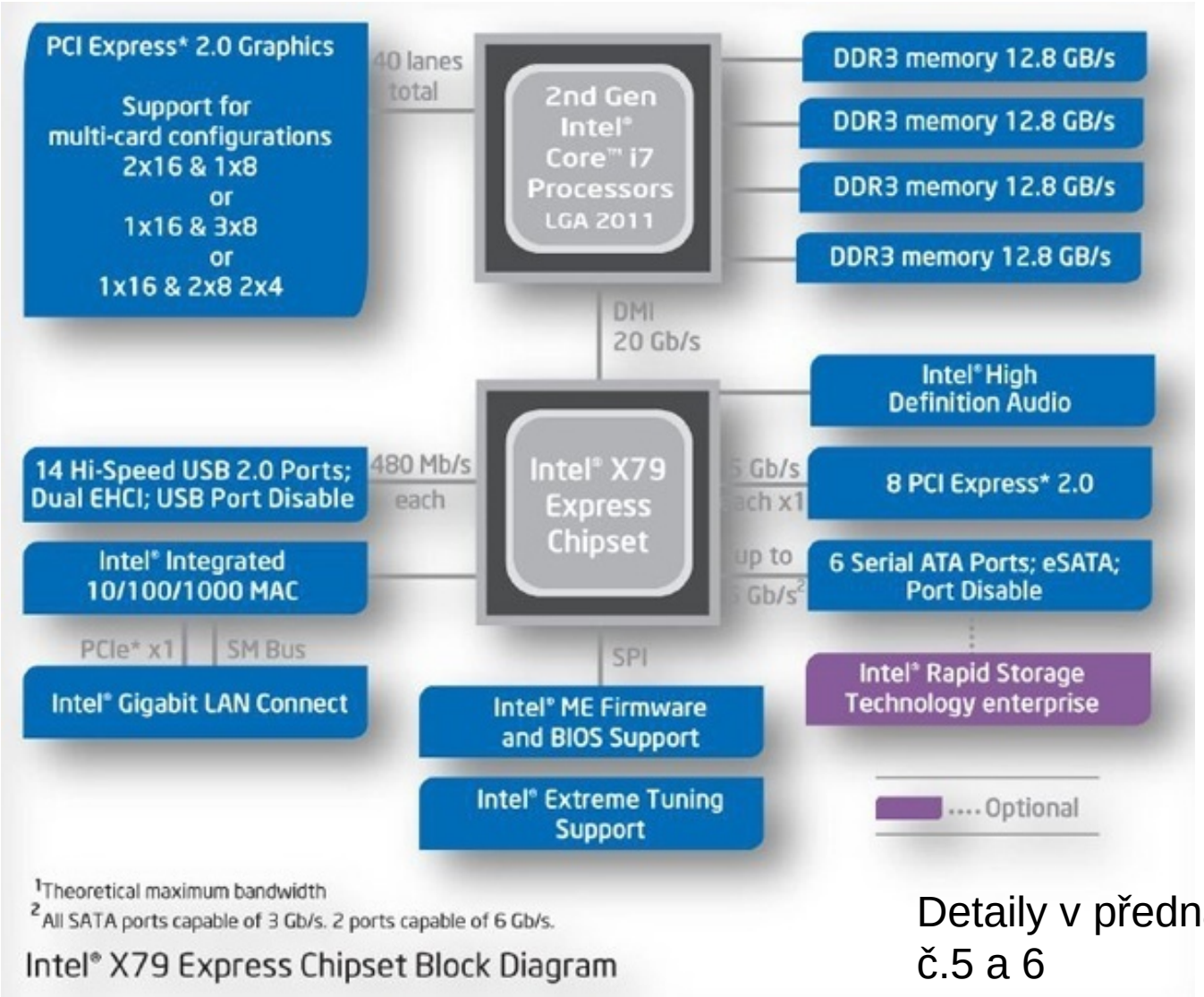
MC - Memory controller – obsahuje obvody pro zajištění operace čtení a zápisu z/do paměti. Také se stará o udržení obsahu paměti – refreshing každých několik desítek ms

Konkrétněji…



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

Konkrétněji…



Detaily v přednáškách č.5 a 6

Terminologie kolem pamětí

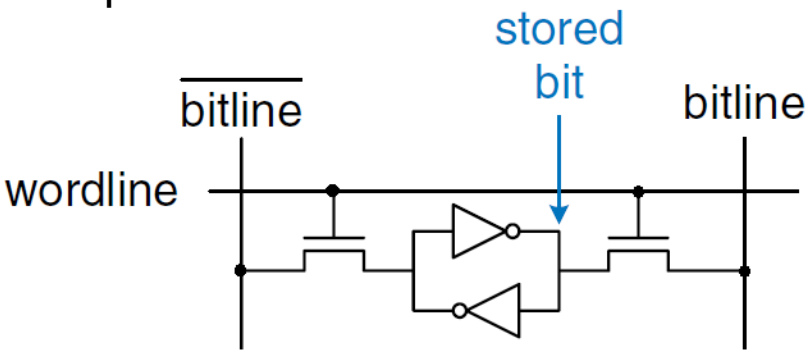
- Typy pamětí RWM (RAM), ROM, FLASH,
- Provedení RAM pamětí:
  SRAM (statická), **DRAM** (dynamická).

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

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

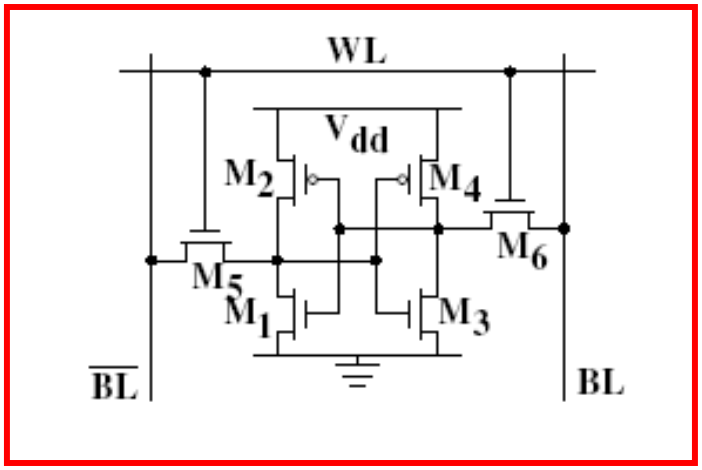
Typický čip a buňka SRAM

Princip:

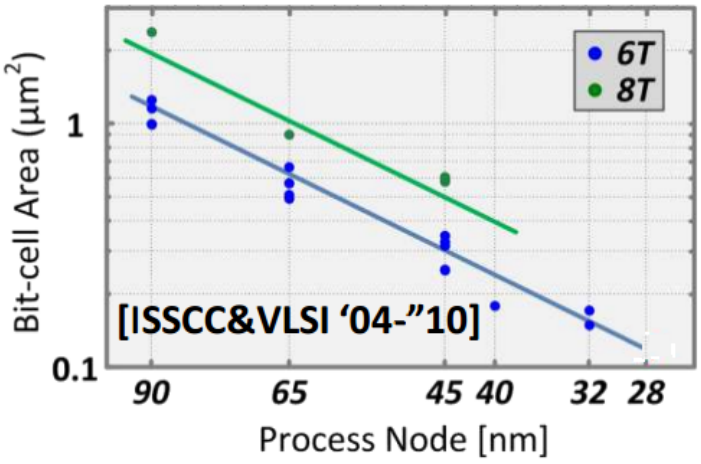


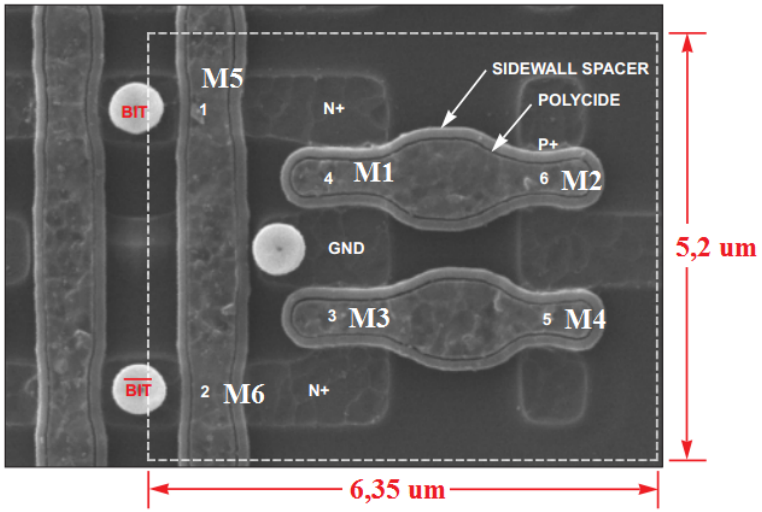
SRAM paměťová buňka

technologie CMOS



Plocha paměťové buňky:

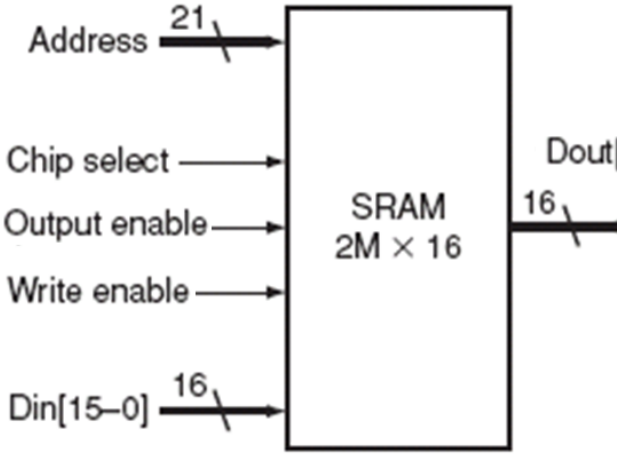




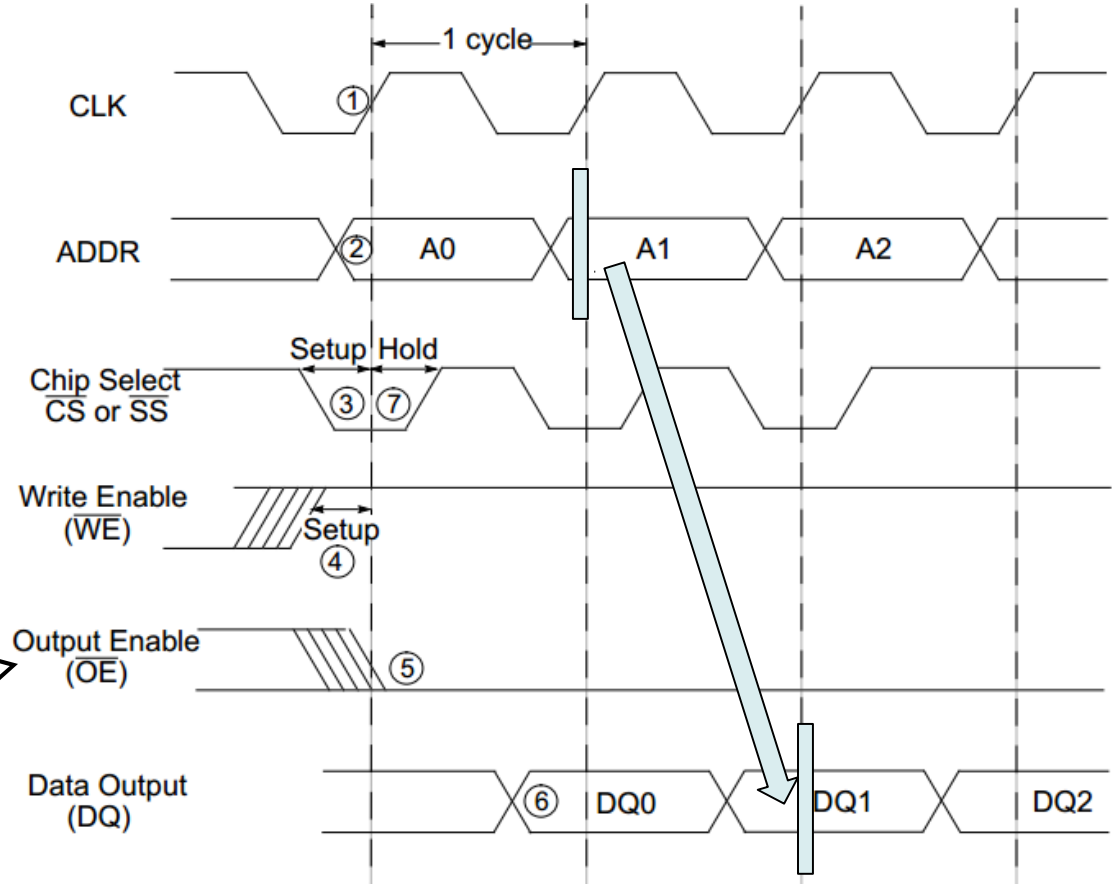
http://educypedia.karadimov.info/library/SEC08.PDF

Typický čip a buňka SRAM

Typický SRAM čip



Příklad čtení – typicky synchronní :

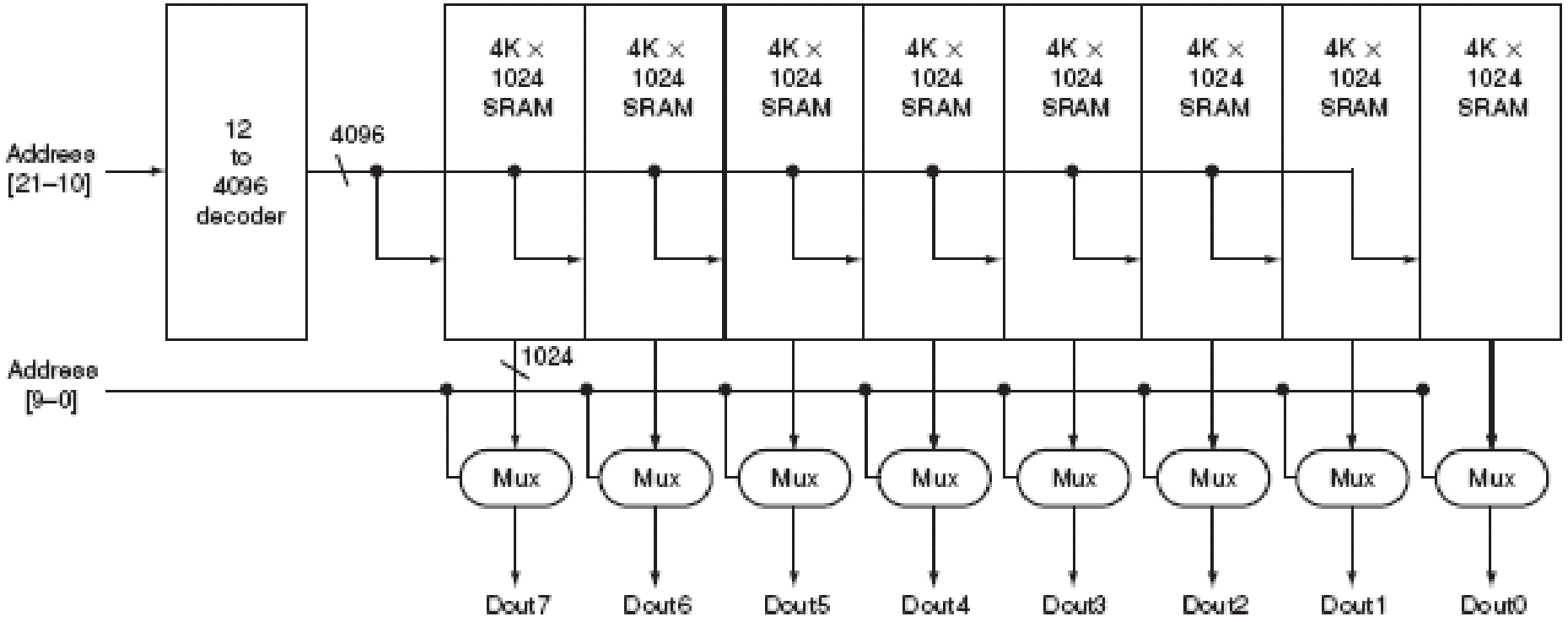


OE signál je asynchronní

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

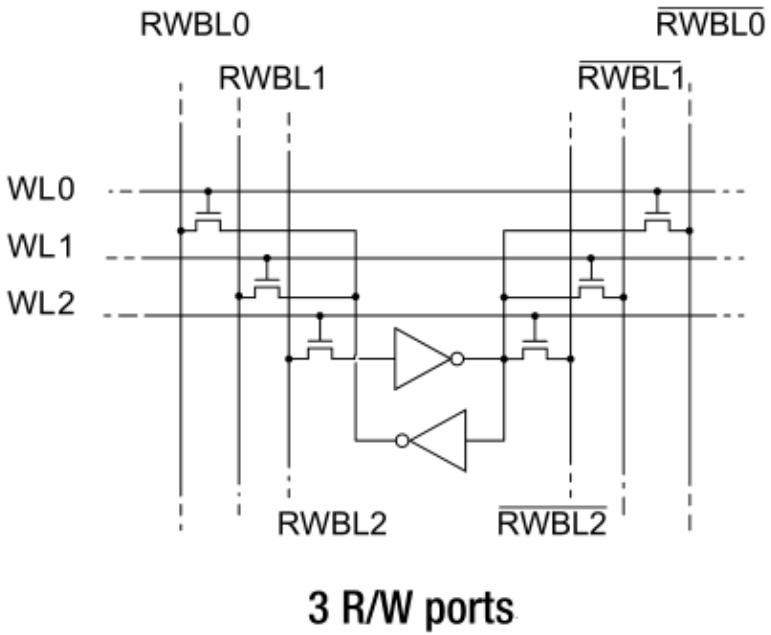
Typický čip a buňka SRAM

Větší paměť?

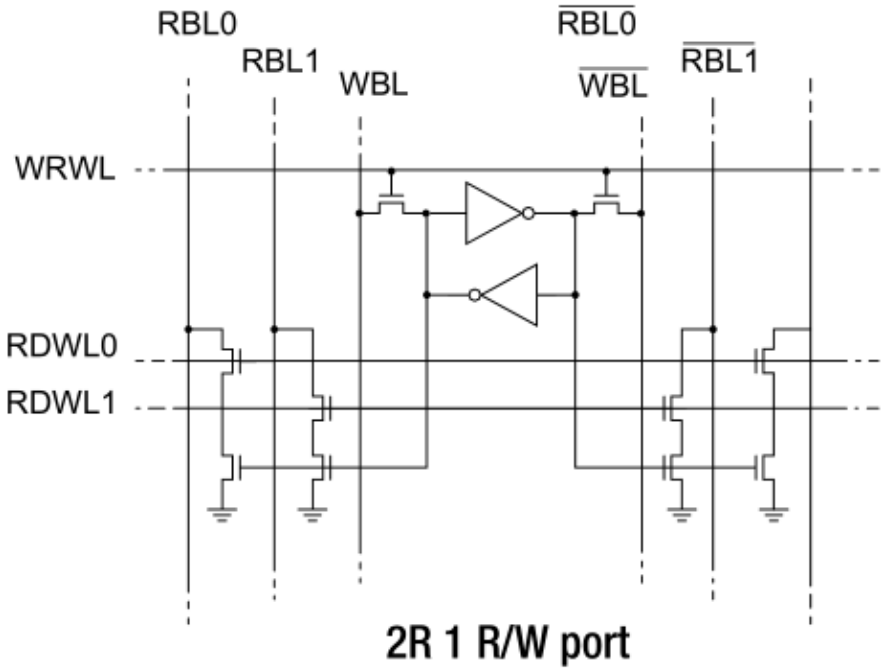


Více-portová cache?

- Pro superskalární CPU nezbytnost
- Rovněž pokud je cache sdílená



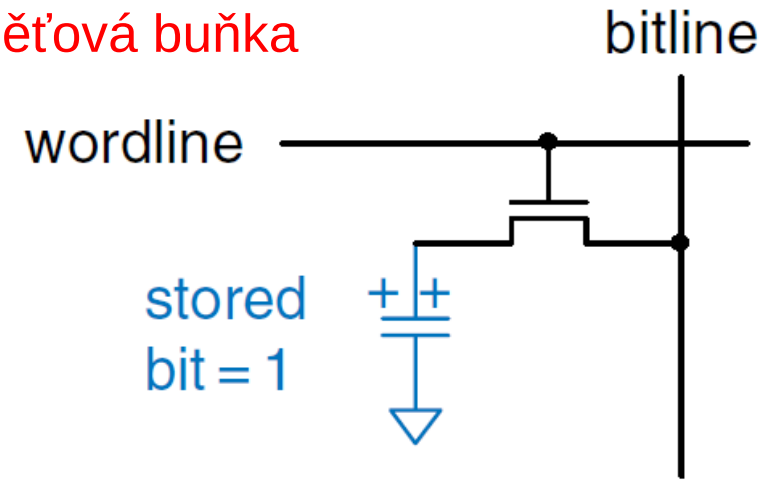
3 R/W ports

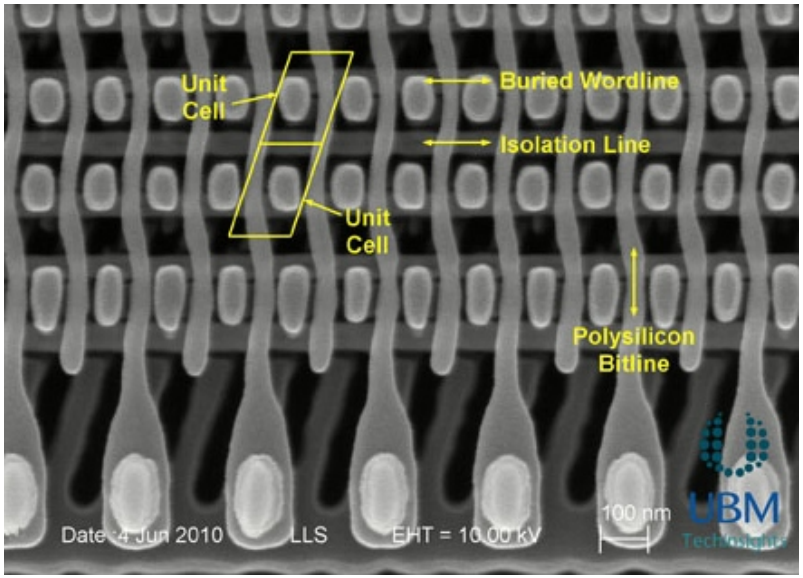


2R 1 R/W port

Detail paměťové buňky dynamické paměti

Jednotranzistorová dynamická paměťová buňka





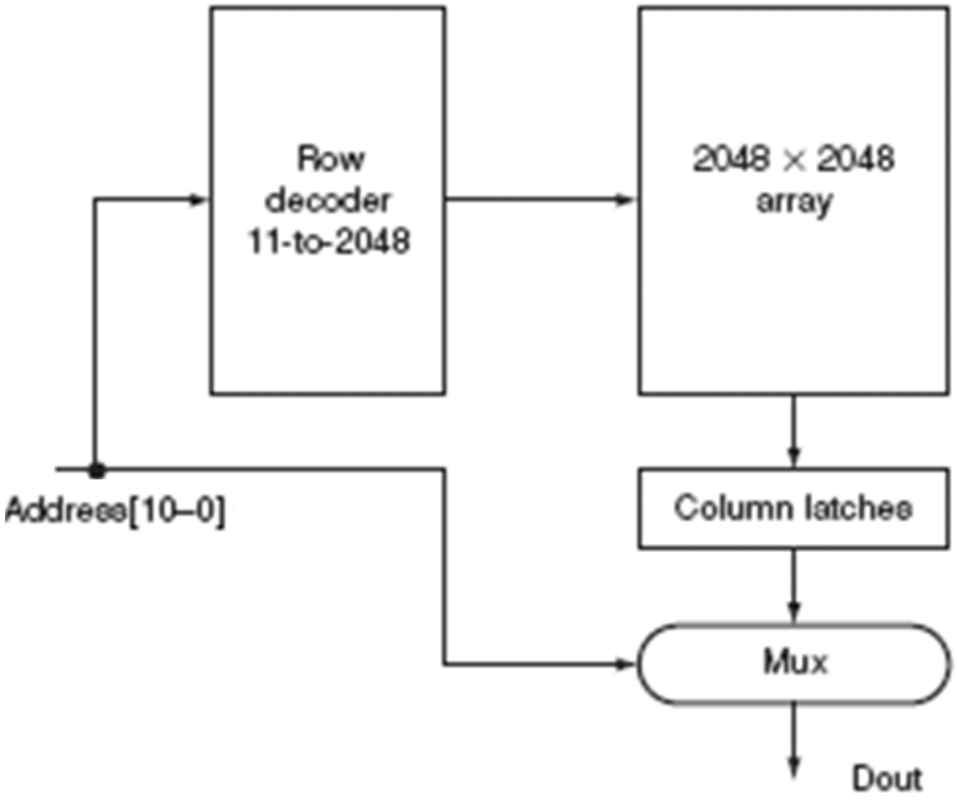
nMOS tranzistor představuje přepínač, který připojí (nebo ne) kondenzátor na vodič „bitline“. Připojení je řízeno vodičem „wordline“.

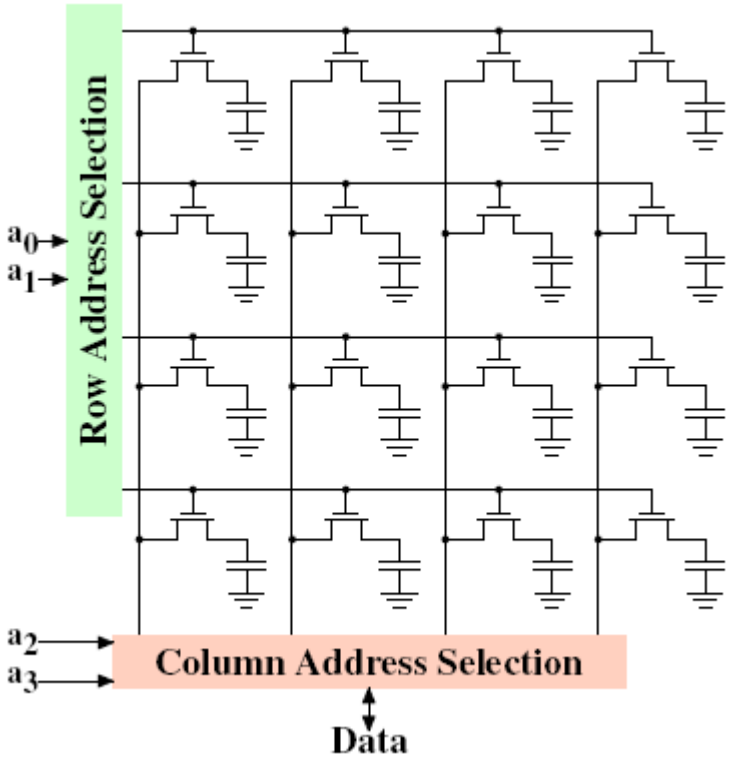
Proces čtení kondenzátor vybíjí. Proto musí být poté obnoven.

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

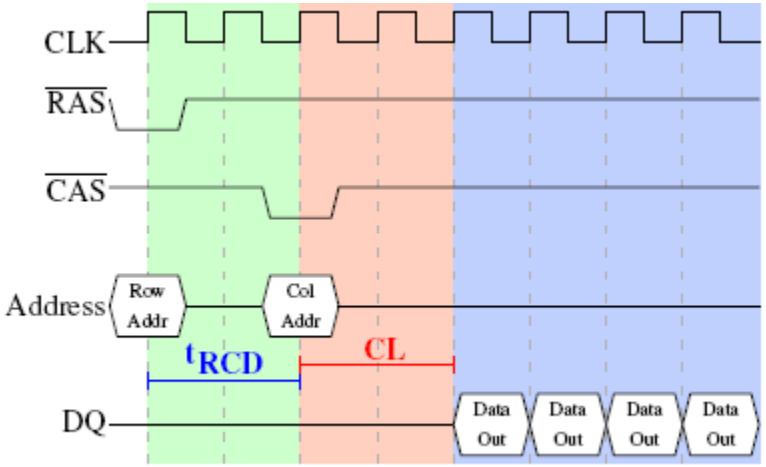




4M × 1 DRAM čip je uvnitř realizován
jako pole 2048 × 2048 1b paměťových buněk

Vývoj DRAM paměťových čipů v čase

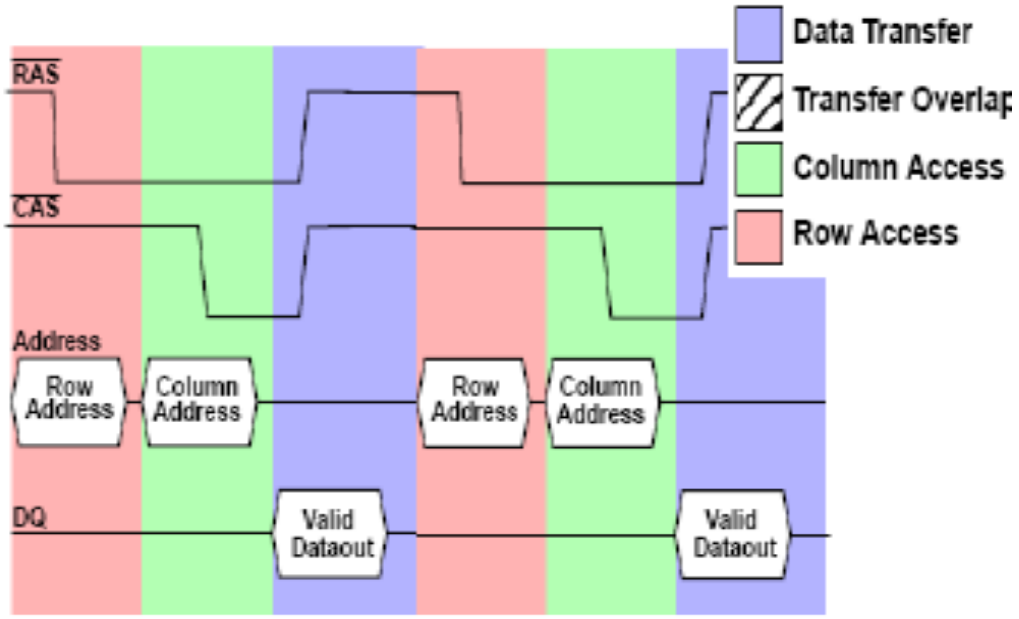
Rok	Kapacita	Cena[$]/GB	Doba přístupu [ns]
1980	64 Kb	1 500 000	250
1983	256 Kb	500 000	185
1985	1 Mb	200 000	135
1989	4 Mb	50 000	110
1992	16 Mb	15 000	90
1996	64 Mb	10 000	60
1998	128 Mb	4 000	60
2000	256 Mb	1 000	55
2004	512 Mb	250	50
2007	1 Gb	50	40



RAS – Row Address Strobe,
CAS – Column Address Strobe

Klasická DRAM – asynchronní rozhraní

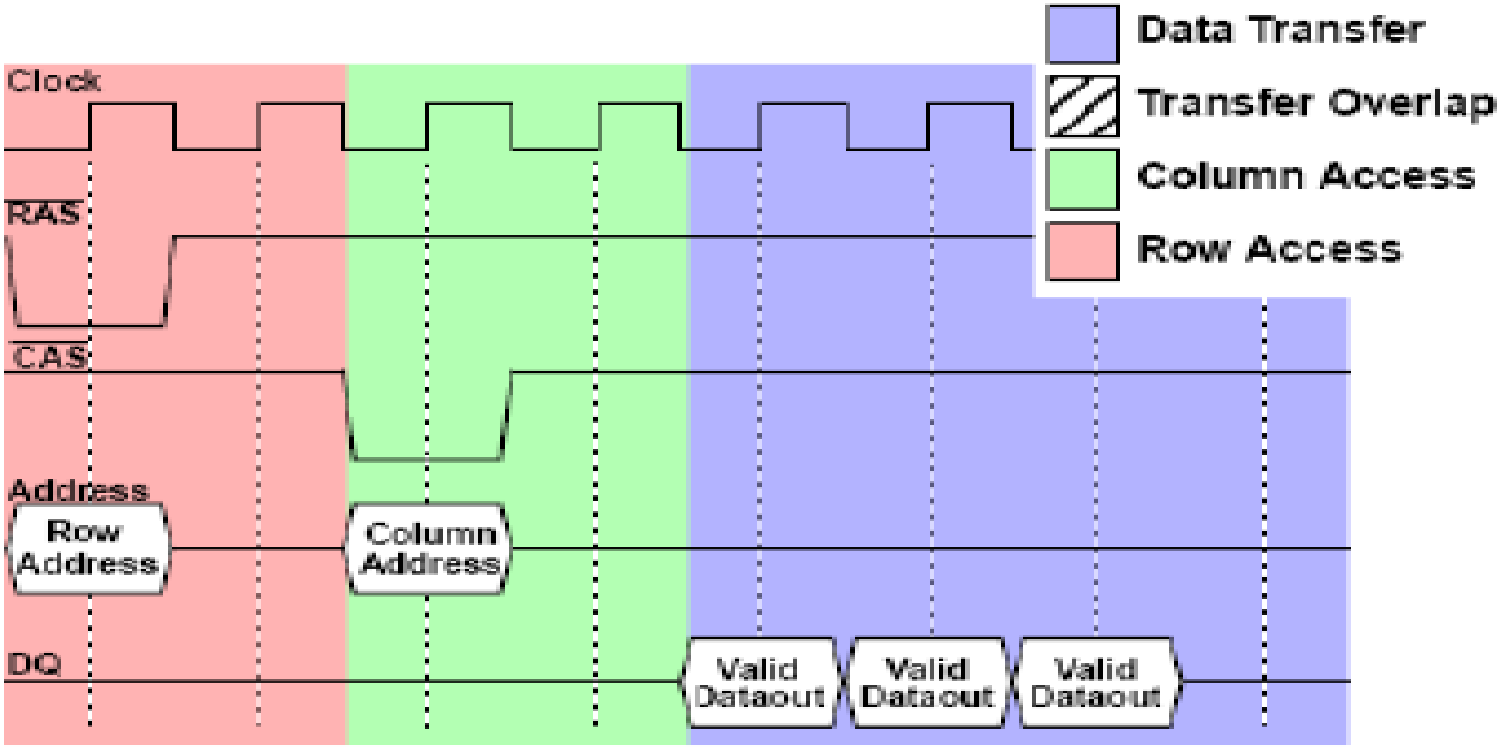
- Důvod rozdělení adresy na 2 části byl dán malým počtem pinů původních DRAM pouzder.
- Toto rozdělení se dodnes zachovává, ačkoli pouzdro už není problém. Uneslo by více vývodů…



RAS – Row Address Strobe,
CAS – Column Address Strobe

SDRAM – konec 90.let – synchronní DRAM

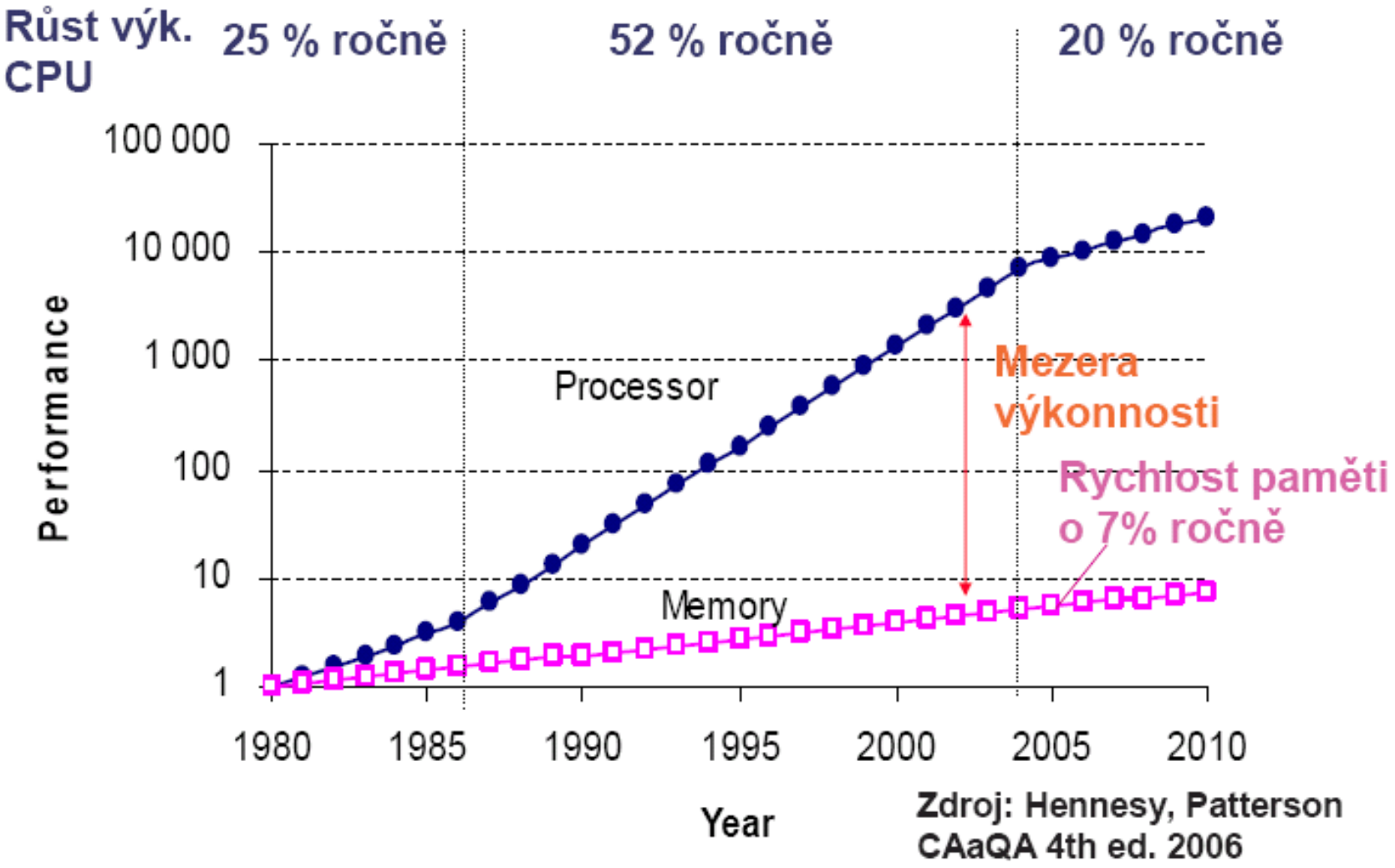
- SDRAM čip obsahuje čítač, který umožňuje nastavit délku souvislého (burst) čtení dat.



SDRAM – paměť současnosti

- SDRAM – frekvence až 100 MHz, 2.5V.
- DDR SDRAM – použití obou hran CLK při přenosu dat, 2.5V.
- DDR2 SDRAM – snížení spotřeby použitím napětí 1.8V, frekvence až 400 MHz.
- DDR3 SDRAM – snížení spotřeby použitím napětí 1.5V, frekvence až 800 MHz.
- DDR4 SDRAM …
- 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?

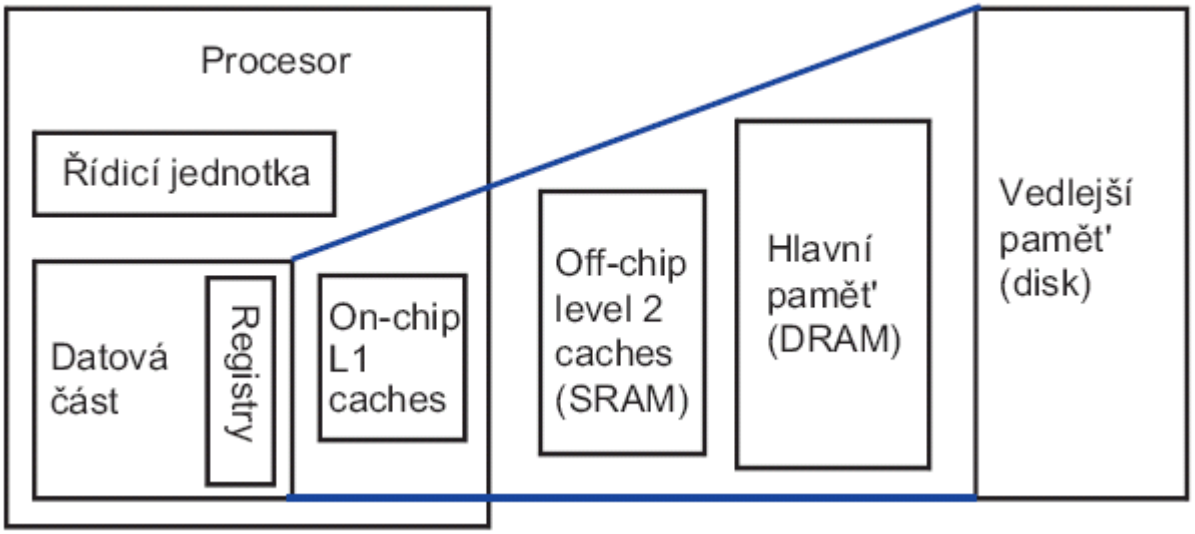
Paměťová hierarchie – základní principy

- 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



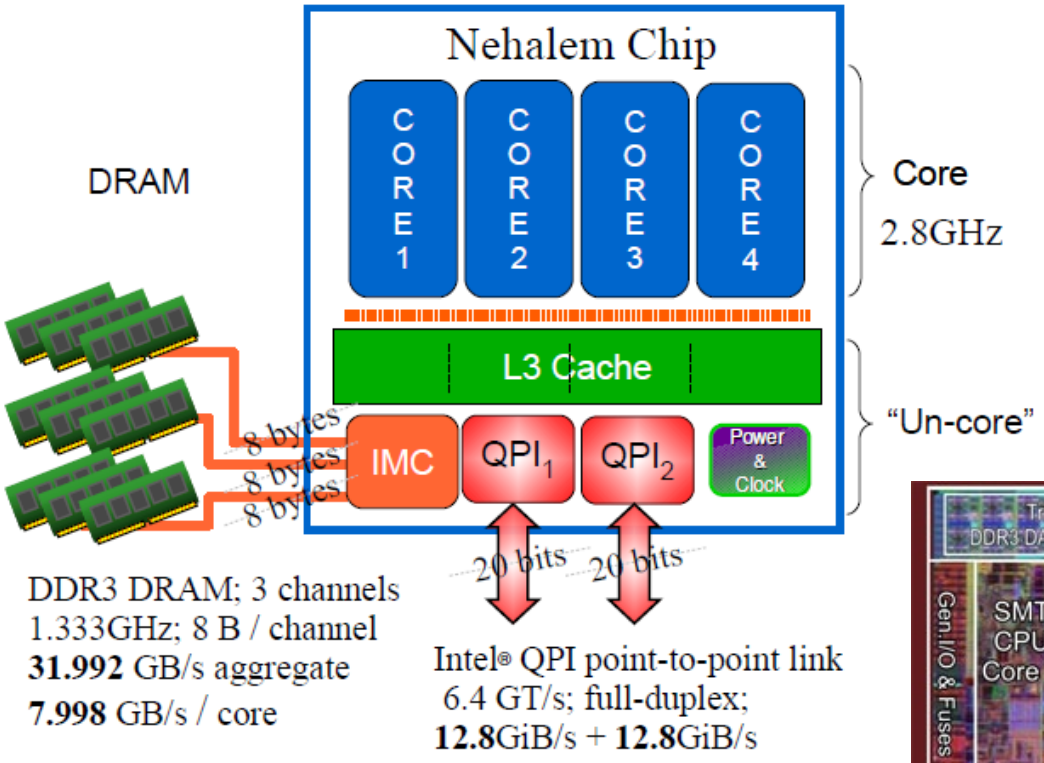
Současná rychlost (2008)	0,5-2,5 ns	50 – 70 ns	5 – 20 ms
cena/GB	$2k-5k	$20 – 75	$0,2 - 2

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

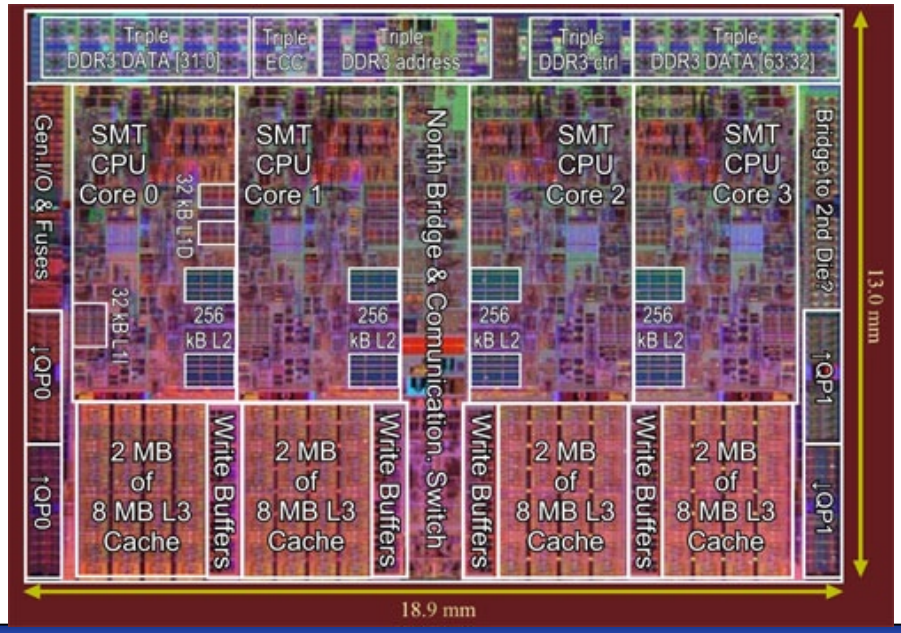
Jak a kde pak ale hledanou informaci najdeme?

- 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.

Příklad procesoru včetně cache
Harvardská architektura - Intel Nehalem



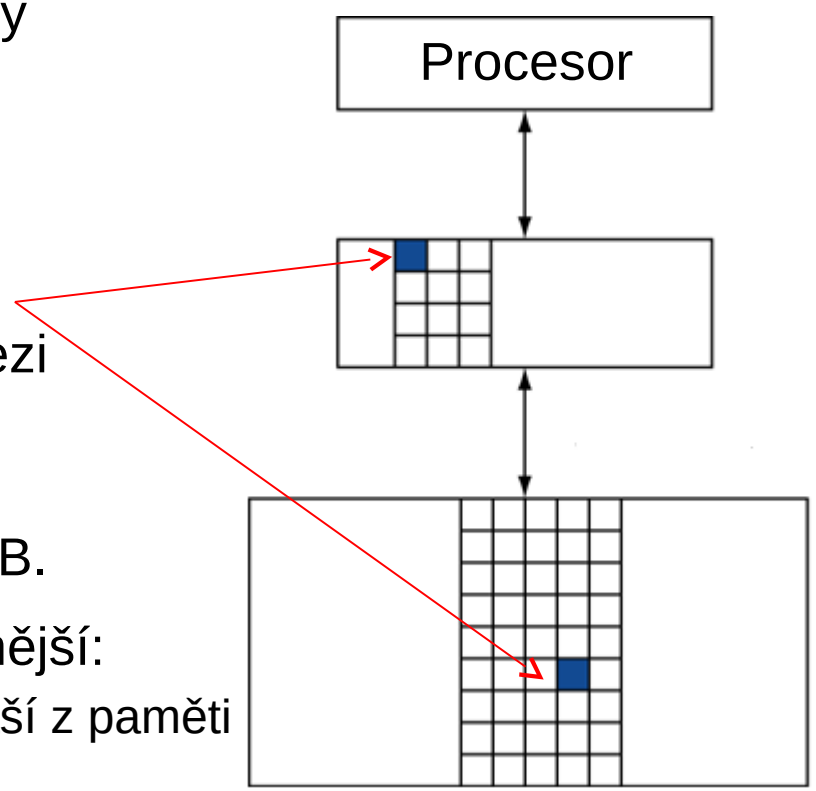
- IMC: integrated memory controller with 3 DDR3 memory channels,
- QPI: Quick-Path Interconnect ports
- Podívejte se na velkosti jednotlivých cache!!!



Řešení disproporce rychlosti? Cache.

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 block pohybuje od 8B do 1KB, typicky 64B.

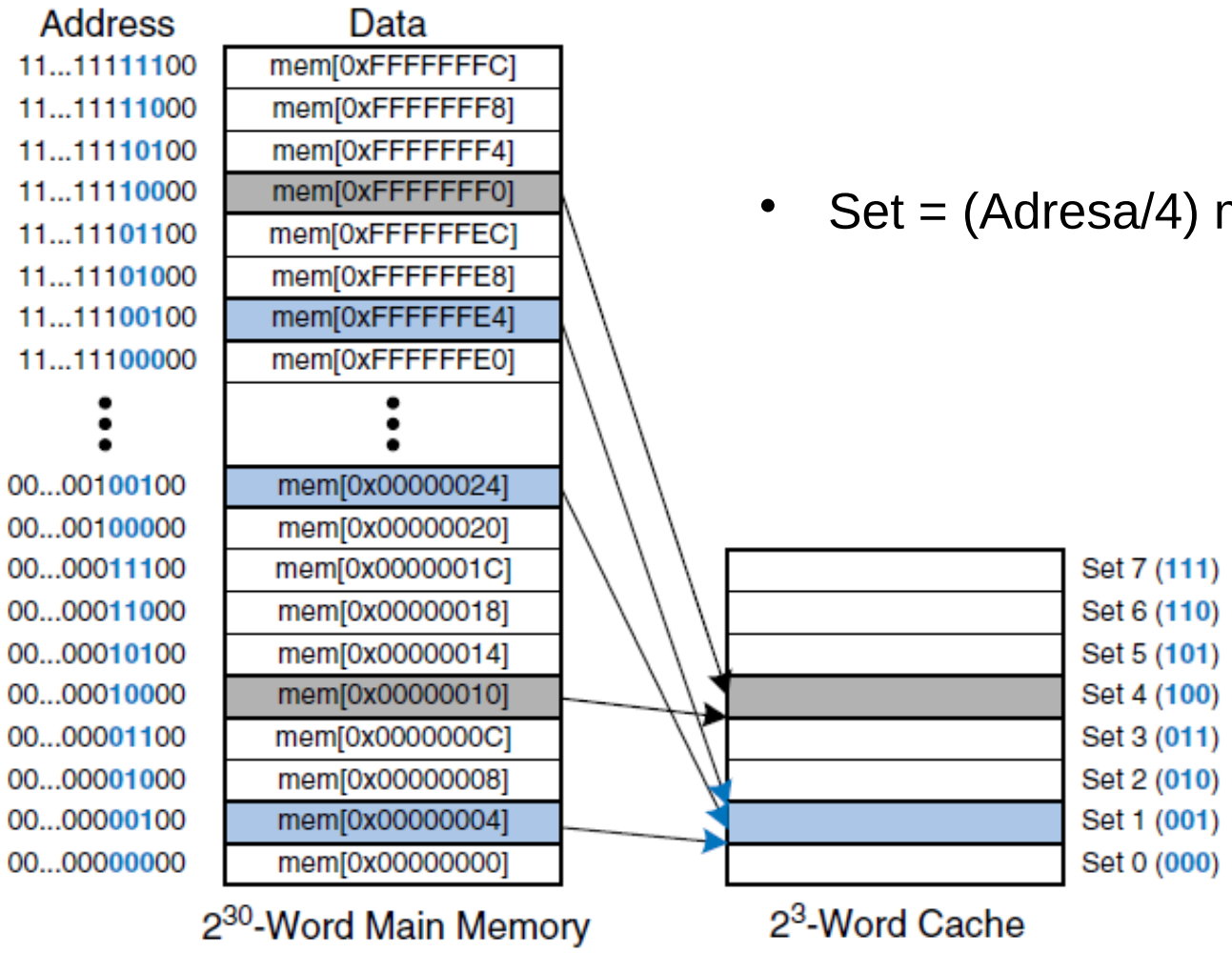


- V rámci A4M36PAP buďme důkladnější:
  - Cache block – data, která se přenáší z paměti do cache a naopak.
  - Cache line navíc musí obsahovat doplňující údaje o každém bloku (Tag, valid, dirty, atd. – záleží na protokolu)
  - Cache row – souvisí s vnitřní organizací cache

Příklad

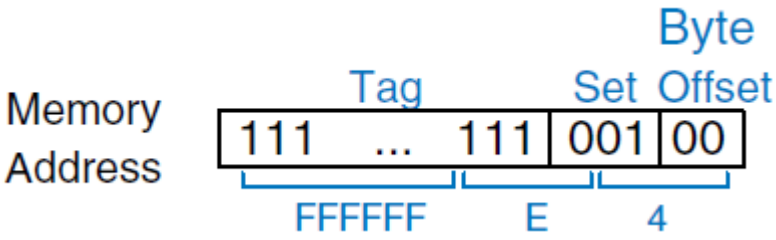
- Mějme cache o velikosti 8-mi bloků. Kam se do ní umístí data z adresy 0xF0000014?
  - Plně asociativní,
  - Přímo mapované, nebo
  - S omezeným stupněm asociativity N=2 (2-cestná, 2-way cache).

Přímo mapovaná cache

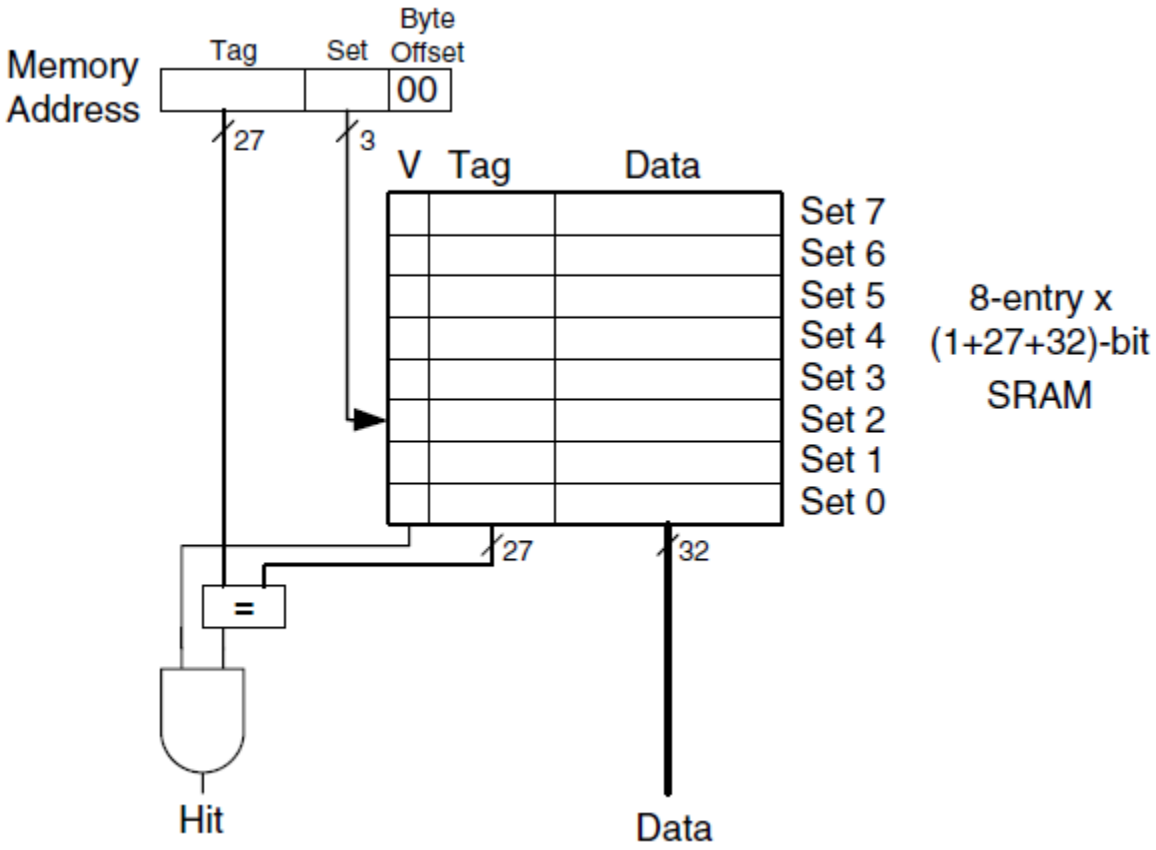


- Set = (Adresa/4) mod 8

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),

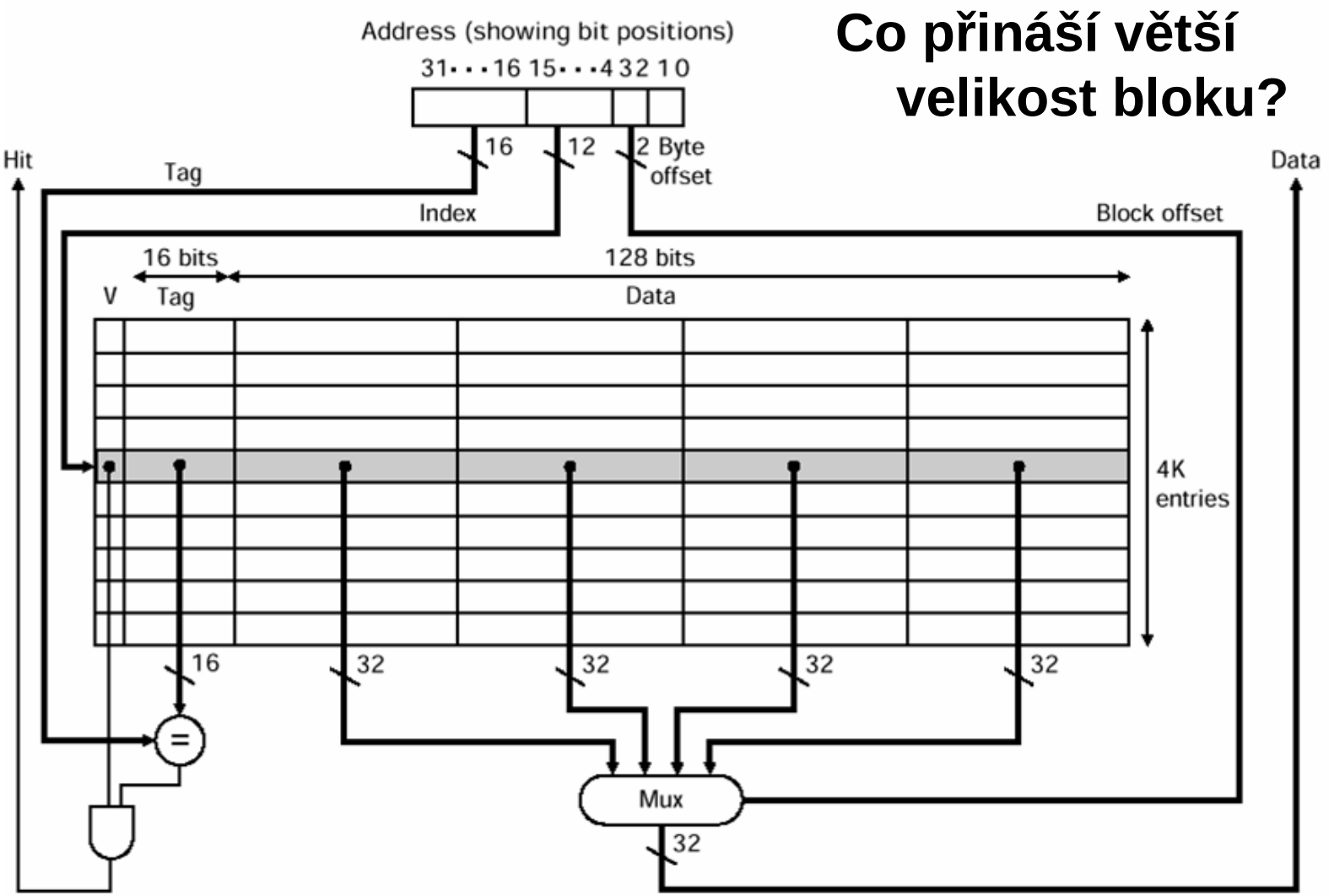
N = 1

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 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

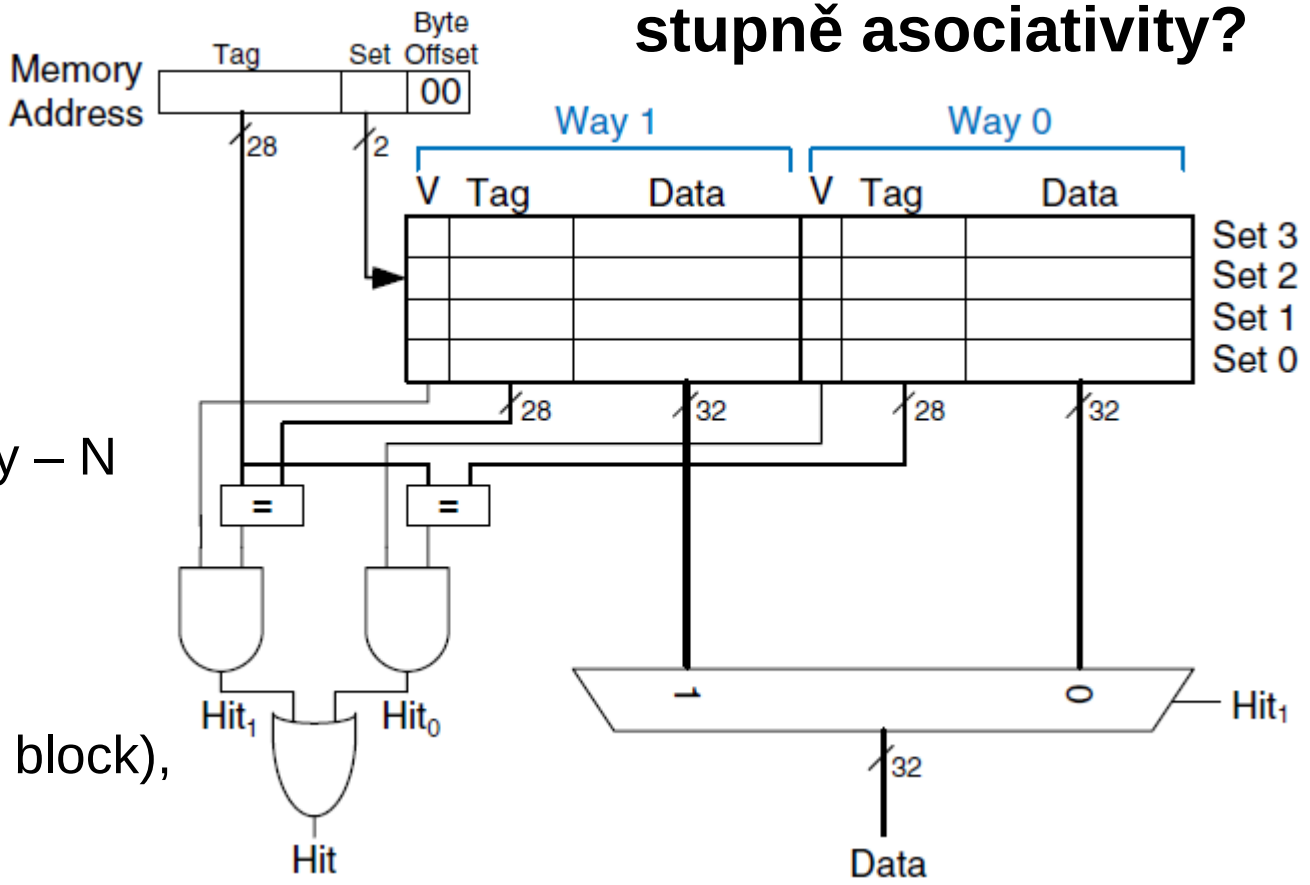
C = 8 (8 words),

S = 4,

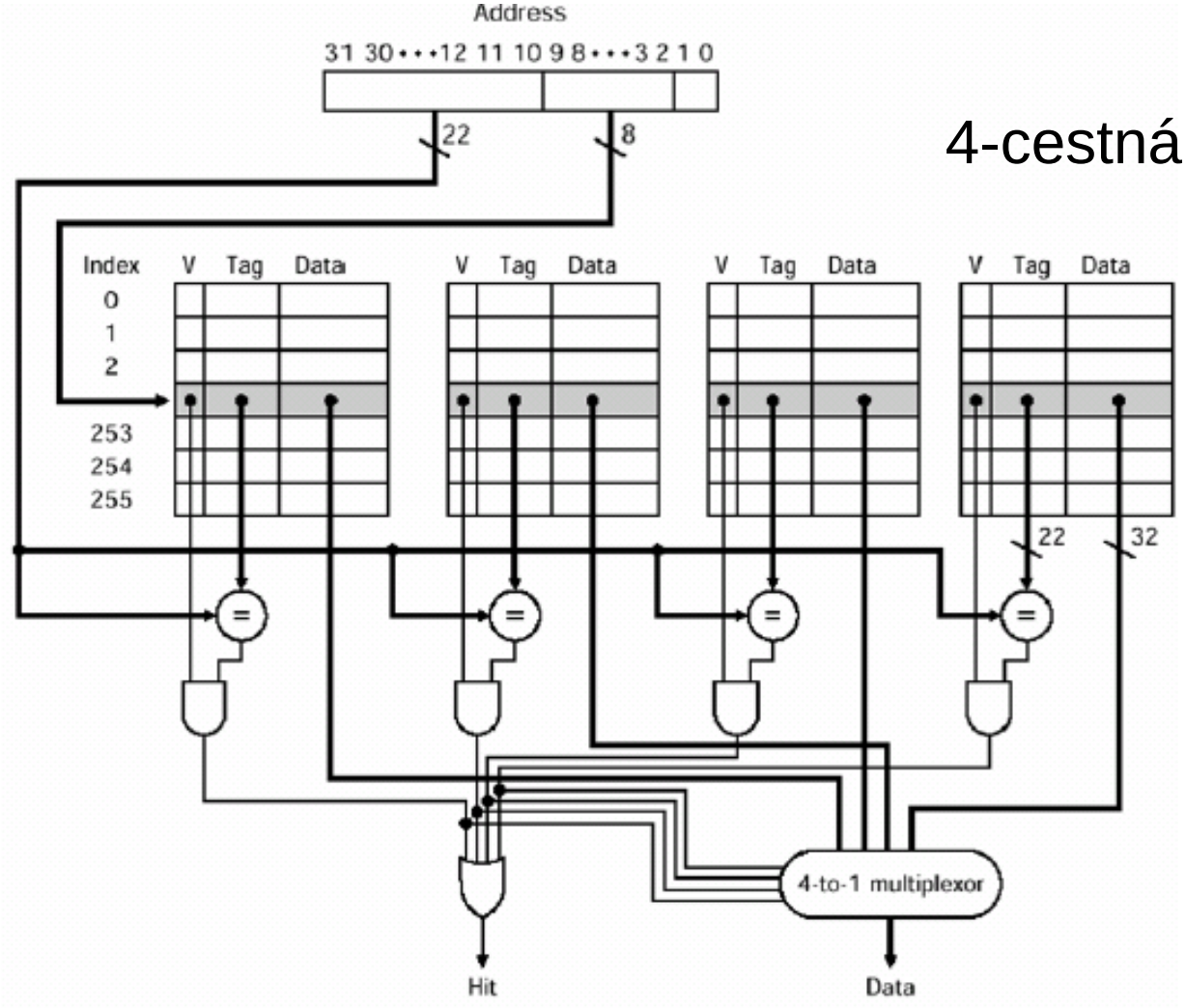
b = 1 (one word in the block),

B = 8

N = 2



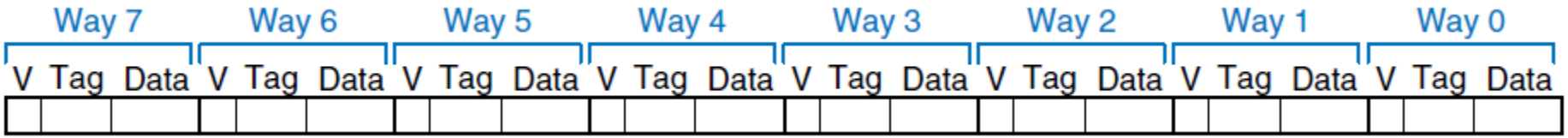
Cache s omezeným stupněm asociativity N=4



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.

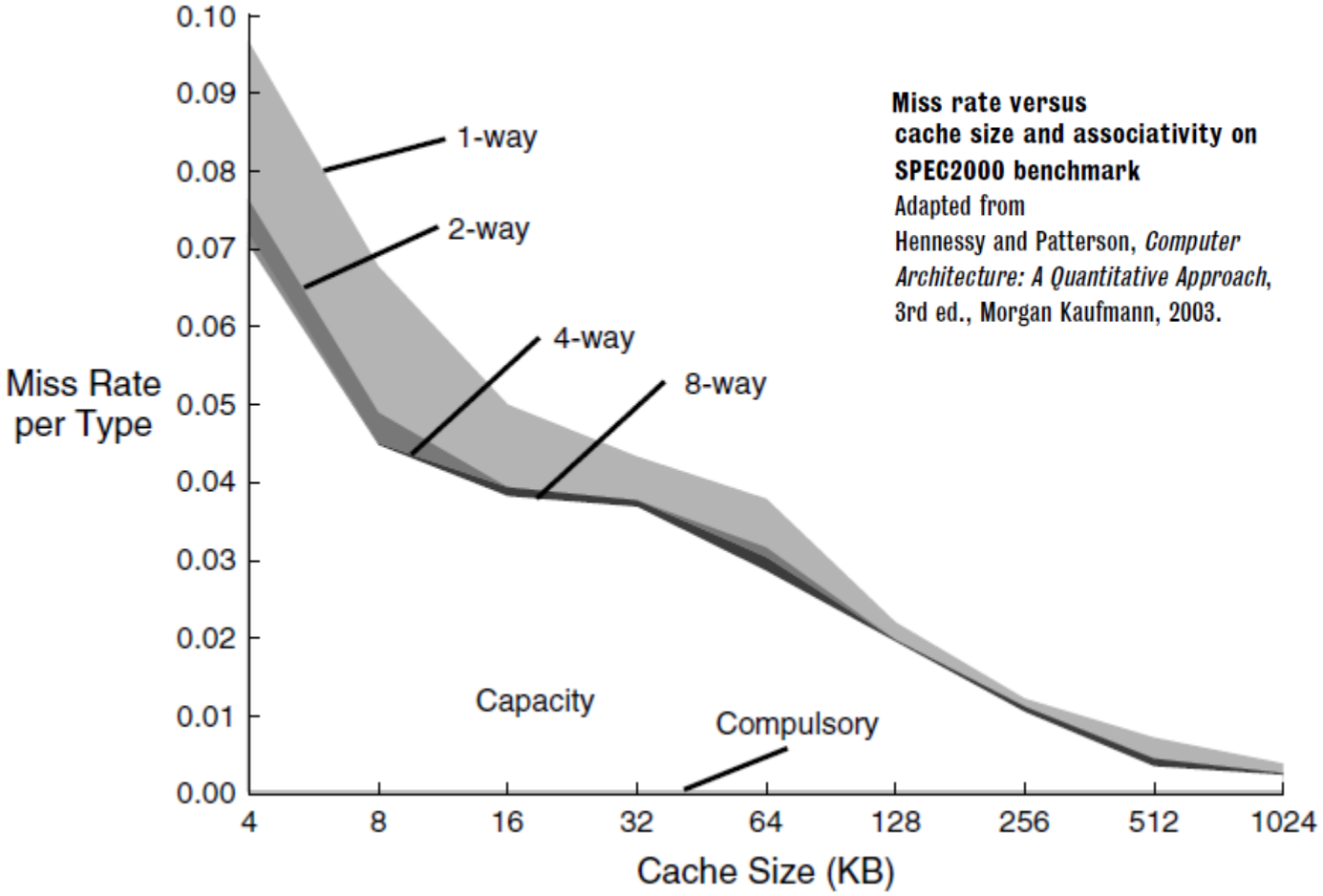
Terminologie kolem skryté paměti II.

- **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)**

  AMAT = HitTime + MissRate×MissPenalty

  MissPenalty – může být vypočtena rekurentně

Porovnání



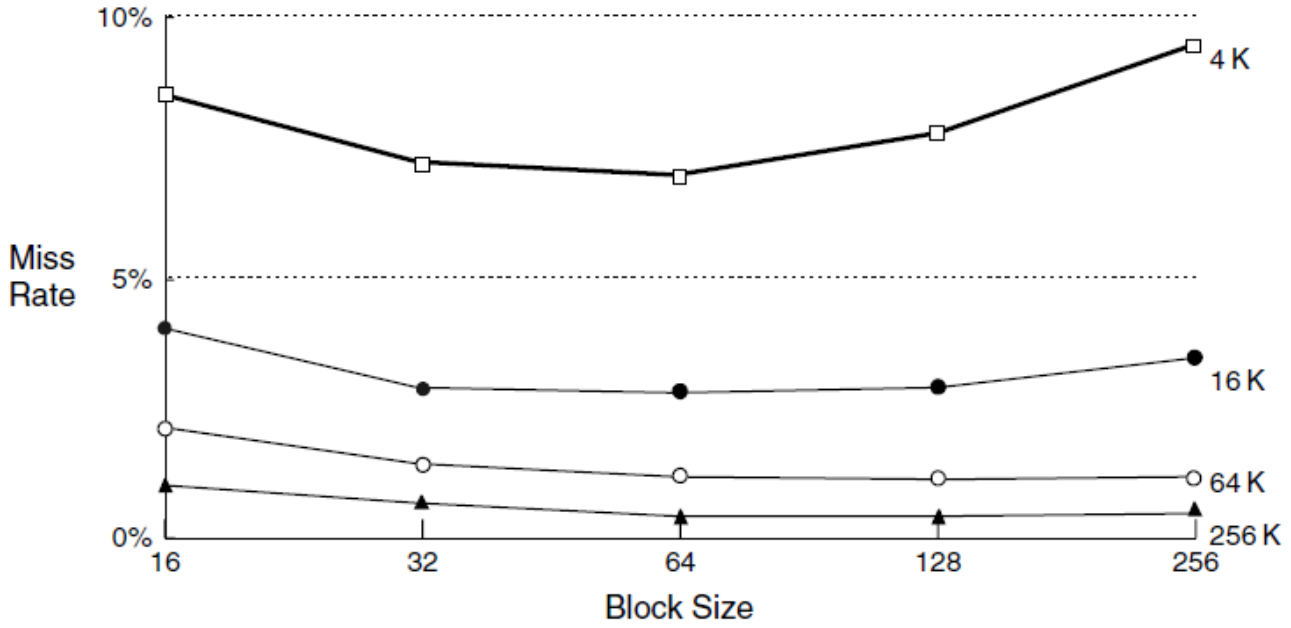
Miss rate versus cache size and associativity on SPEC2000 benchmark
Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Pamatujte: 1. miss rate není vlastností 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

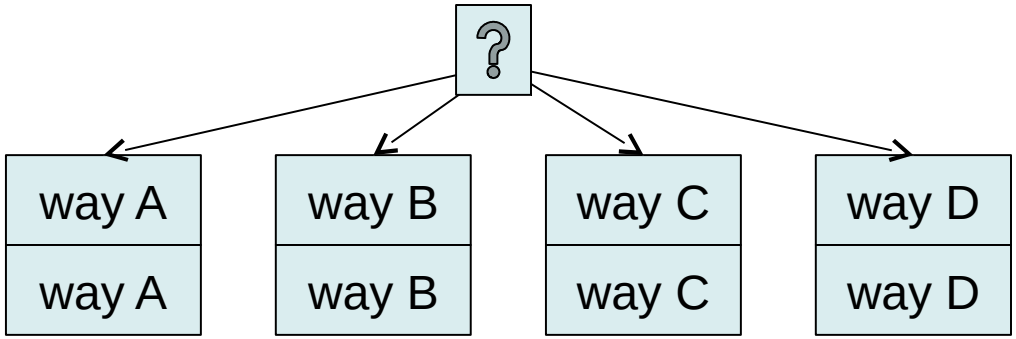
- Data se nejprve musí z hlavní paměti přečíst. Jenže: co když je cache plná?

- 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.

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

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

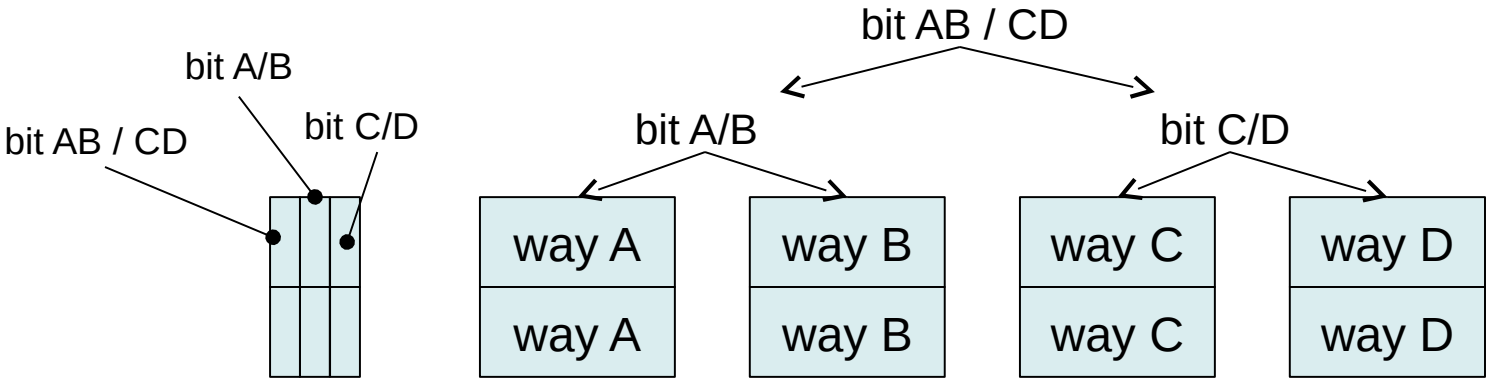
- Jak byste implementovali LRU (Least Recently Used) ???
- Představte si 4-cestně asociativní cache!



- Není to tak snadné pokud má být LRU rychlý…
- Intel používá pseoudo-LRU pro 4-way, kde každý set má navíc 3 bity
- Na činnost cache můžeme nahlížet jako na případy kdy v cache data nalezneme (hit) a na případy kdy ne (miss)
- V případě hitu musíme uchovat informaci o tom, která cesta data poskytla – zcela jistě nebude nejméně používanou (least recently used)
- V případě missu se musíme rozhodnout kam nová data uložit

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

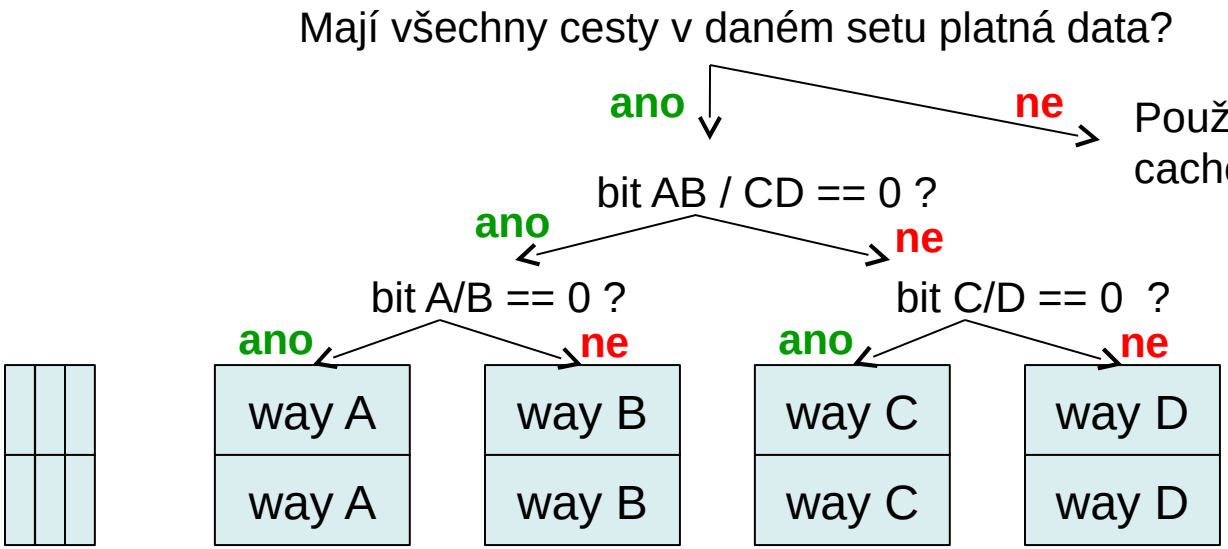
- Jak byste implementovali LRU (Least Recently Used) ???



- Nejvyšší bit „AB/CD“ se nastavuje pokud je hit v A nebo B. Tento bit je nulován pokud je hit v C nebo D. Pokud je cache miss, nic se nezapisuje. Tento bit „AB/CD“ nám tedy dává informaci o tom, ve které „půlce“ cache byl naposled hit.
- Rekurentně, bit „A/B“ se nastavuje pokud byl hit v A, nuluje pokud v B.
- Podobně bit „C/D“.

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

- Jak byste implementovali LRU (Least Recently Used) ???
- Která data nahradíme v případě cache miss???



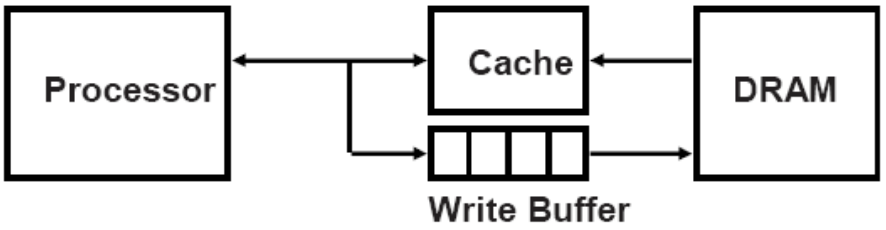
state	replace	next state
00x	way A	11u
01x	way B	10u
1x0	way C	0u1
1x1	way D	0u0

x – don't care
u - unchanged

- **Sumarizace:** Výhodou pseudo-LRU je, že v případě cache hit jsou všechny bity „AB/CD“, „A/B“, „C/D“ pouze zapisovány. Zpomalení (čtení) nastává pouze v případě cache miss. Použití binárního stromu pro implementaci pseudo-LRU je snadno rozšířitelné i na 8, 16, atd. cest.

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 (avní.

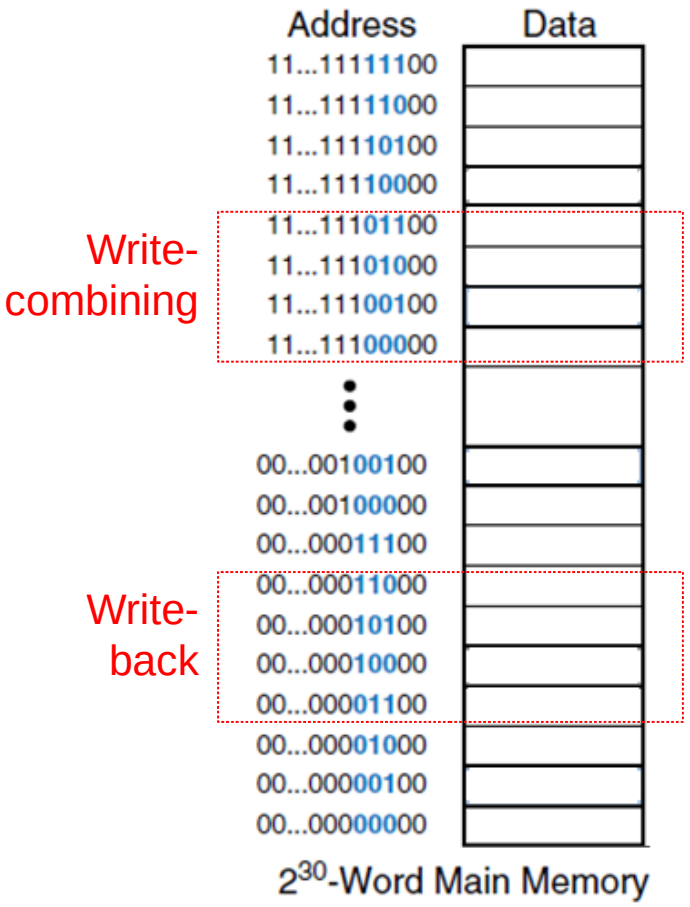


V	Další bity, např. D	Tag	Data

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**

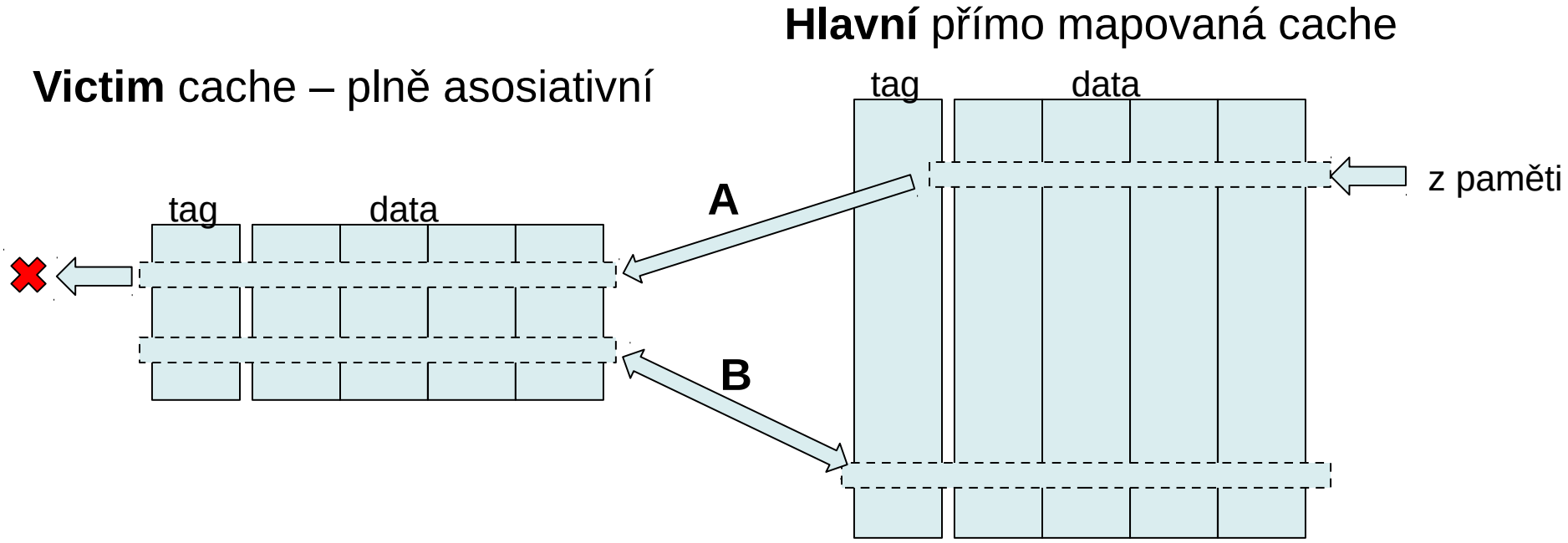
- Primární SP je bezprostředně připojena k procesoru
  - Rychlá, malá. Nejdůležitější: minimální Hit Time
- L2 SP ošetřuje výpadky primární SP
  - Větší, pomalejší, ale stále rychlejší než hlavní paměť. Nejdůležitější: low Miss Rate
- Hlavní paměť ošetřuje výpadky L2
- Současné nejvýkonnější systémy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

Victim cache

- Přímo mapovaná cache je levná a rychlá
- Problémem přímo mapované cache je, že v případě konfliktu (stejné mapování dvou různých adres) jsou starší (mnohdy stále užitečné) data/instrukce nahrazeny novějšími
- Řešením tohoto problému byla N-cestně asociativní, resp. plně asociativní cache.
- **Je to jediné řešení? Ne!** Ještě lze použít tzv. Victim cache.
- PRINCIP: Používejme rychlou přímo mapovanou cache. V případě, že data z této cache odstraňujeme, uložme je do Victim cache. Při cache miss se pak máme šanci data najít v této Victim cache.

Victim cache

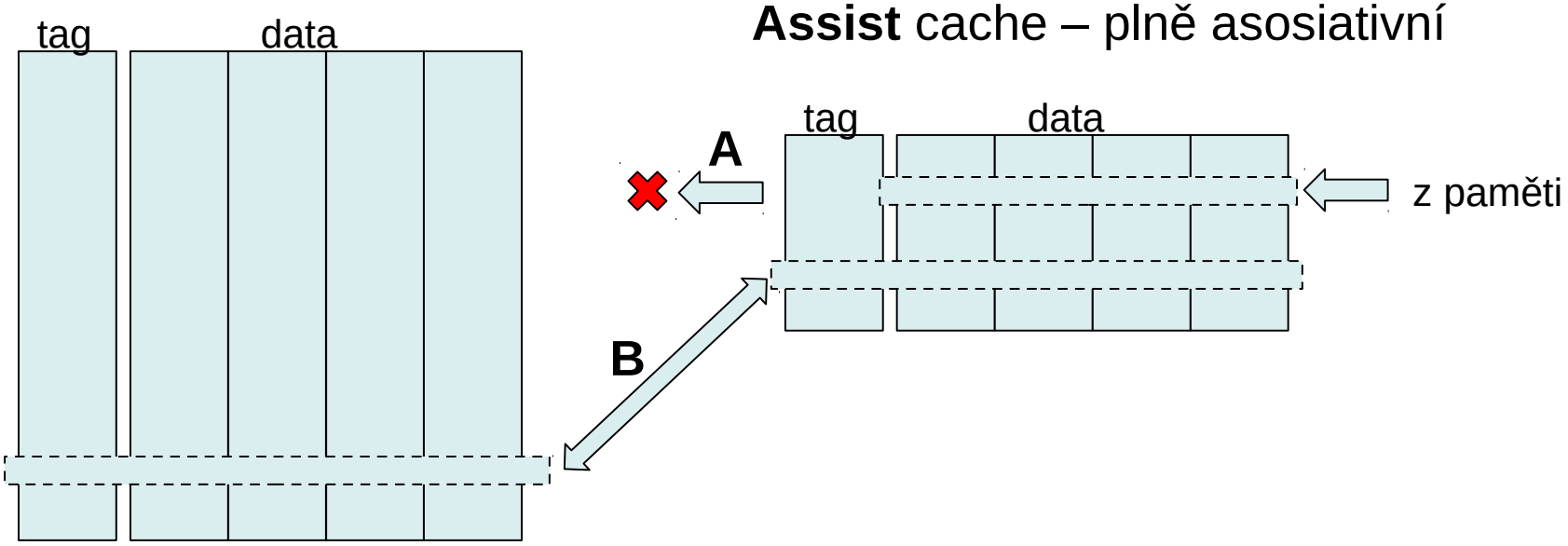


- **A**: Přicházející blok je umístěn do hlavní cache a „vytlačený“ blok do Victim cache (FIFO strategie nad Victim cache postačuje k realizaci LRU – důsledek pravidla B)
- **B**: Pokud dojde k miss v hlavní cache a hit ve Victim cache, prohoď cache lines mezi těmito cache
- **Je to jediné řešení? Ne!** Ještě lze použít tzv. Assist cache.

Assist cache

Hlavní přímo mapovaná cache

Assist cache – plně asosiativní



- **A**: Přicházející blok je uložen do Assist cache (FIFO)
- **B**: Pokud dojde k miss v hlavní cache a hit ve Assist cache, prohoď cache lines mezi těmito cache.
- Poznámka: Data se přesouvají do hlavní cache až po hitu v Assist cache, to znamená, až po opakované žádosti na tu samou adresu. Do hlavní cache tedy jdou data vykazující časovou lokalitu

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…
- 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:

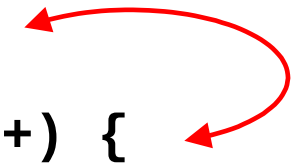
```
/* Před optimalizací */
for (i = 0; i < SIZE; i++)
   for (j = 0; j < SIZE; j++)
     a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < SIZE; i++)
   for (j = 0; j < SIZE; j++)
     d[i][j] = a[i][j] - c[i][j];

/* 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

```
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

  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?

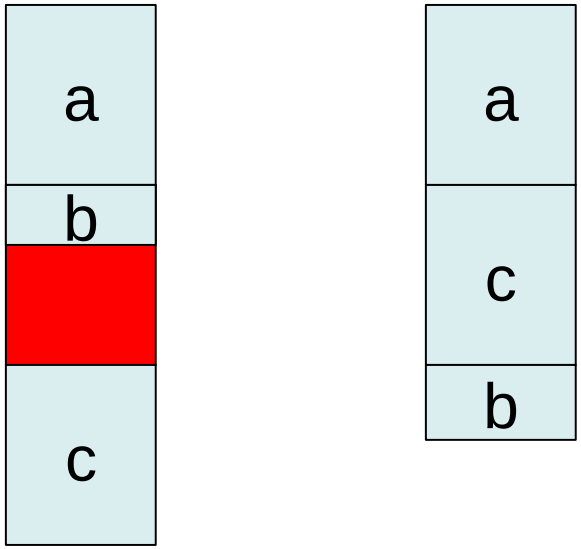
- Neplýtvejme pamětí – používejme minimální množství paměti
- Spatřujete rozdíl v těchto deklaracích?

- `/* 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=i;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

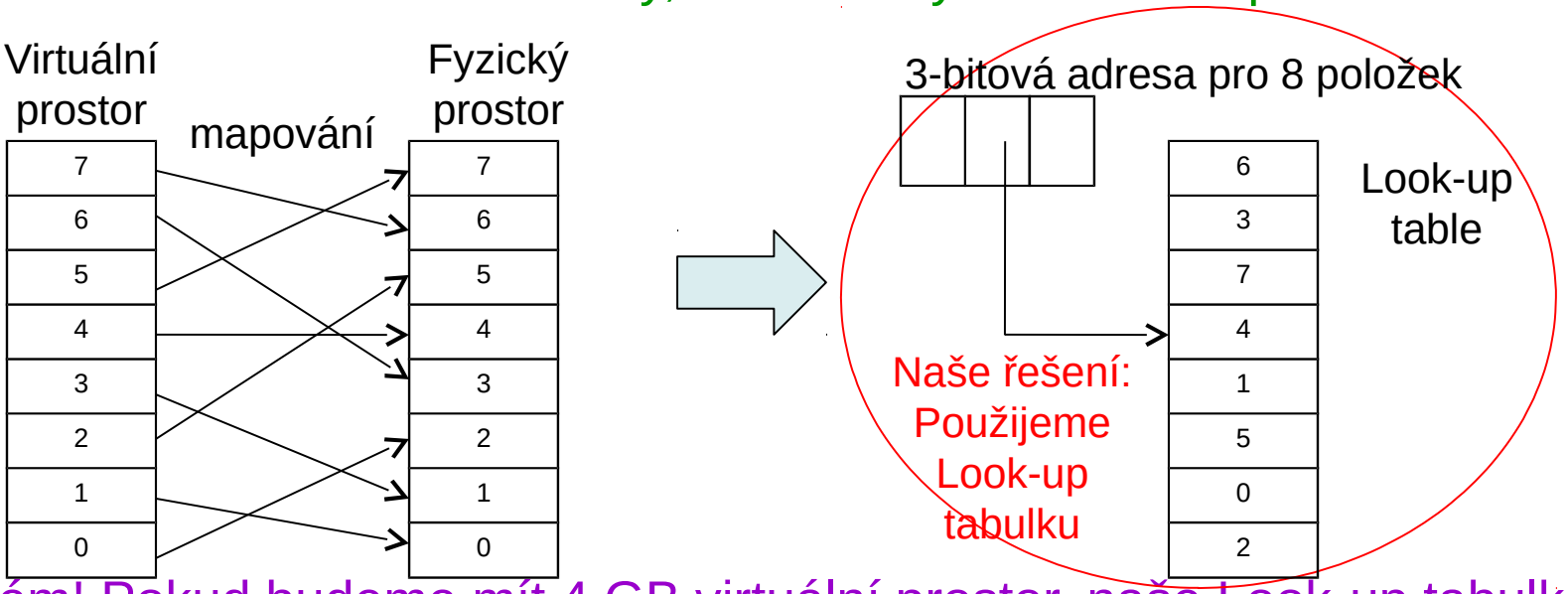
Virtuální paměť.

Motivace k virtuální paměti..

- **Běžně máme na počítači spuštěno několik desítek/stovek procesů…**
- **Umíte si představit situaci, kdy bychom rozdělili fyzikou paměť (například 1 GB) mezi tyto procesy? Jak veliký kus paměti by pak patřil jednomu procesu? Jak bychom řešili kolize – kdy nějaký program úmyslně (například virus) nebo neúmyslně (chybou programátora – práce s ukazateli) by chtěl zapisovat do kusu paměti, který jsme vyhradili jinému procesu?**
- **Řešením je právě virtuální paměť…**
- **Každému procesu vytvoříme iluzi, že celá paměť je pouze jeho a může se v ní libovolně zcela bezpečně pohybovat.**
- **Dokonce každému procesu dále vytvoříme iluzi, že má k dispozici např. 4GB paměti i když je fyzická paměť mnohem menší. Proces pak nerozlišuje mezi fyzickou pamětí a diskem (disk se mu jeví jako paměť).**
- **Základní idea: Proces adresuje ve virtuální paměti pomocí virtuálních adres. Ty pak musíme nějak přeložit na adresy fyzické.**

Motivace k virtuální paměti..

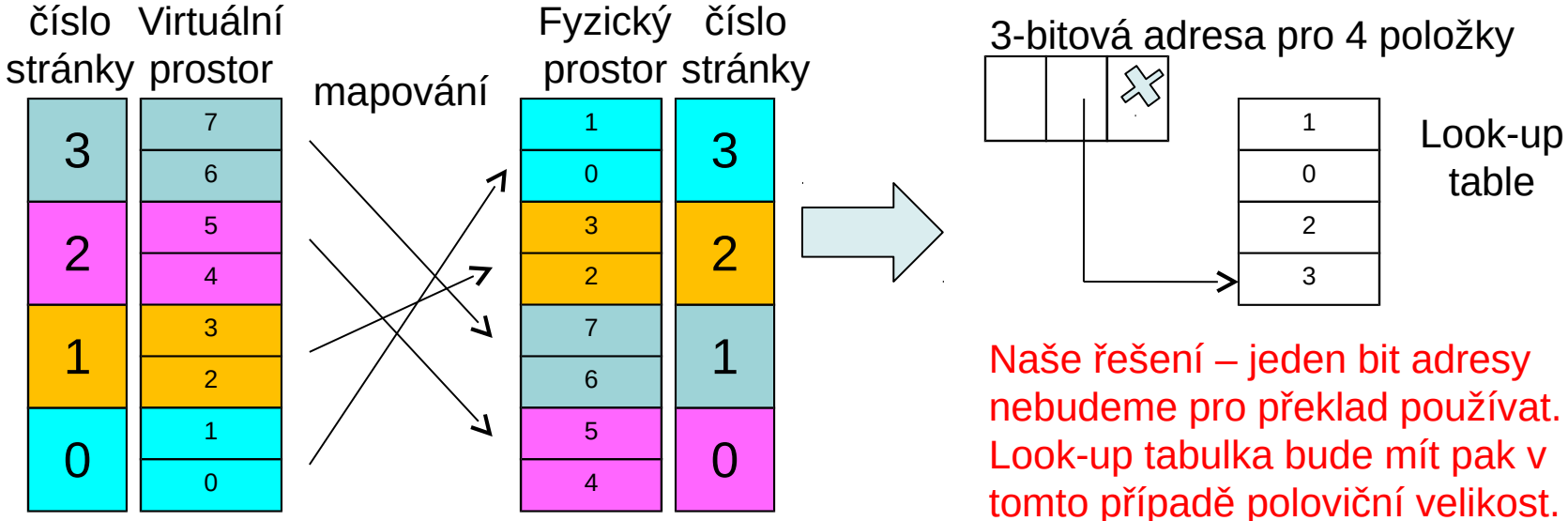
- Představme si, že máme 8B (Bajtů) virtuální prostor a 8B fyzické paměti…
- **Jak zabezpečíme překlad adres?** Předpokládejme **adresaci po bajtech**.
- **Zde je jedno řešení:** Chceme přeložit libovolnou virtuální adresu na libovolnou fyzickou adresu. Máme 3-bitovou virtuální adresu, a tu chceme přeložit na 3-bitovou fyzickou adresu. K tomu stačí tabulka o 8 záznamech, kde jeden záznam bude mít 3 bity, dohromady 8x3=24bitů/proces.



- Problém! Pokud budeme mít 4 GB virtuální prostor, naše Look-up tabulka bude zabírat 2^{32}x32 bitů = 16GB/proces!!! To je poněkud hodně…

Motivace k virtuální paměti.. - Ponaučení z předchozího slide:

- **Mapování z libovolné virtuální adresy na libovolnou fyzickou adresu je prakticky nerealizovatelný požadavek!**
- **Řešení:** Rozdělme virtuální prostor na stejně velké části – virtuální stránky, a fyzickou paměť na fyzické stránky. Ať je velikost virtuální a fyzické stránky stejná. V našem příkladu máme stránku o velikosti 2B.



Naše řešení – jeden bit adresy nebudeme pro překlad používat. Look-up tabulka bude mít pak v tomto případě poloviční velikost.

- Naše řešení tedy překládá virtuální adresy po skupinách… Uvnitř dané stránky se pak pohybujeme za pomocí právě toho bitu, který jsme při překladu ignorovali.. Tím jsme schopni využít celý adresní prostor.

Příklad č.1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, b[4], *c, d;
    c = (int*)malloc(4*sizeof(int));
    printf("%p %p %p %p\n",&a,&b,&c,&d);
    printf("%p %p %p\n",&b[0],&b[1],&b[2]);
    printf("%p %p %p\n",&c[0],&c[1],&c[2]);
    free(c);
    return 0;
}
```

Co z toho vyplývá?

- Data jsou v poli uložena za sebou.

Otázky, které se nabízí..

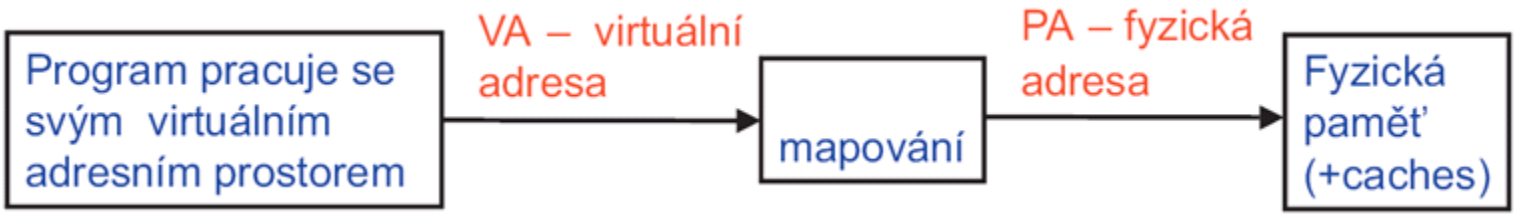
- Ale co je to za adresu?
- Kam do cache se tyto data namapují?

Výpis programu:

```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

Virtualizace paměti

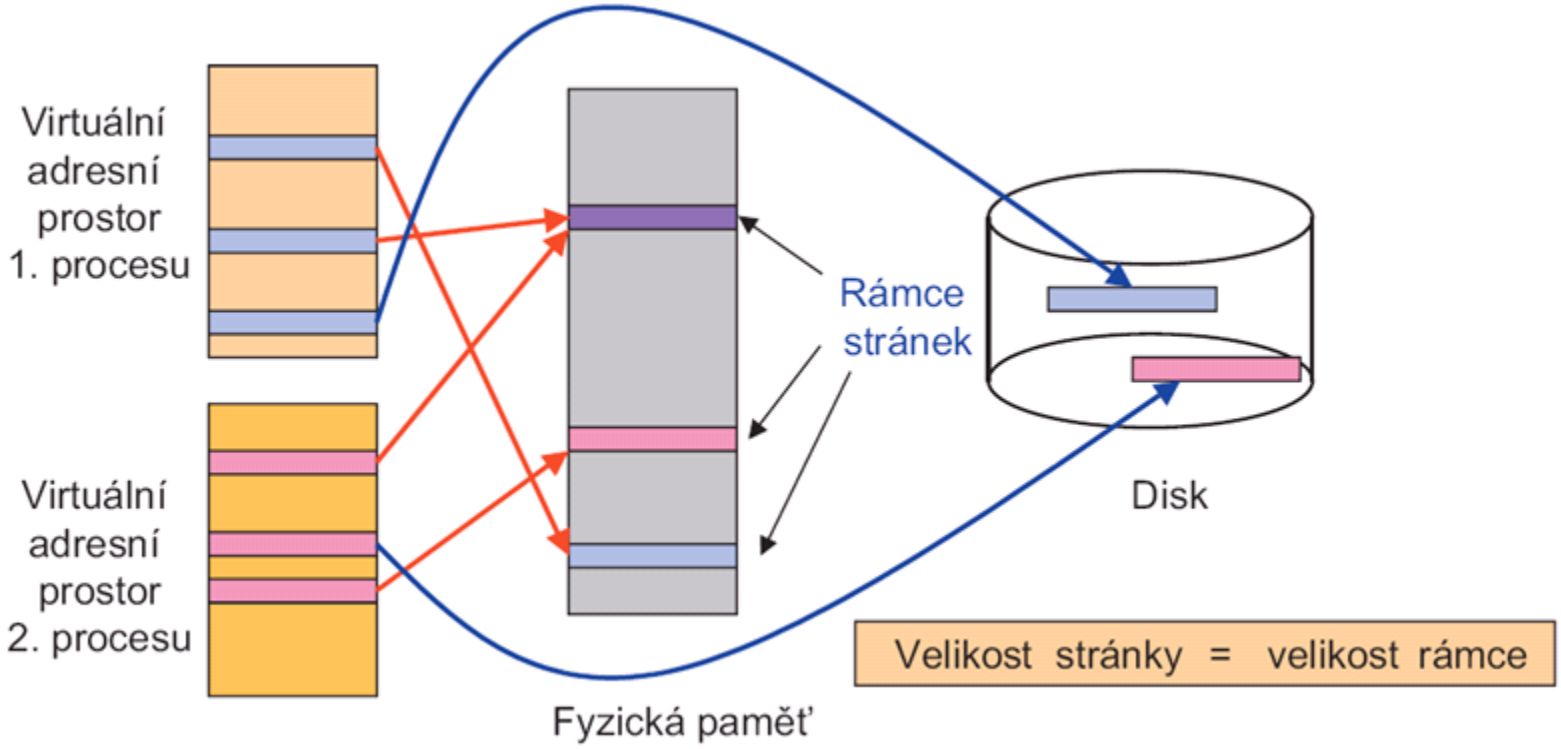
- **VP** je způsob správy operační paměti umožňující běžícímu procesu zpřístupnění paměťového prostoru, který je uspořádán jinak, nebo je dokonce větší, než je fyzicky připojená operační paměť.
- Převod mezi virtuální VA a fyzickou PA adresou může podporovat procesor (HW mapováním TLB, viz dále).
- V současně běžných operačních systémech je virtuální paměť implementována pomocí stránkování paměti spolu se stránkováním na disk, které rozšiřuje operační paměť o prostor na disku.



* R. Lórenc, X36APS, 2005

Virtuální paměť - stránkování

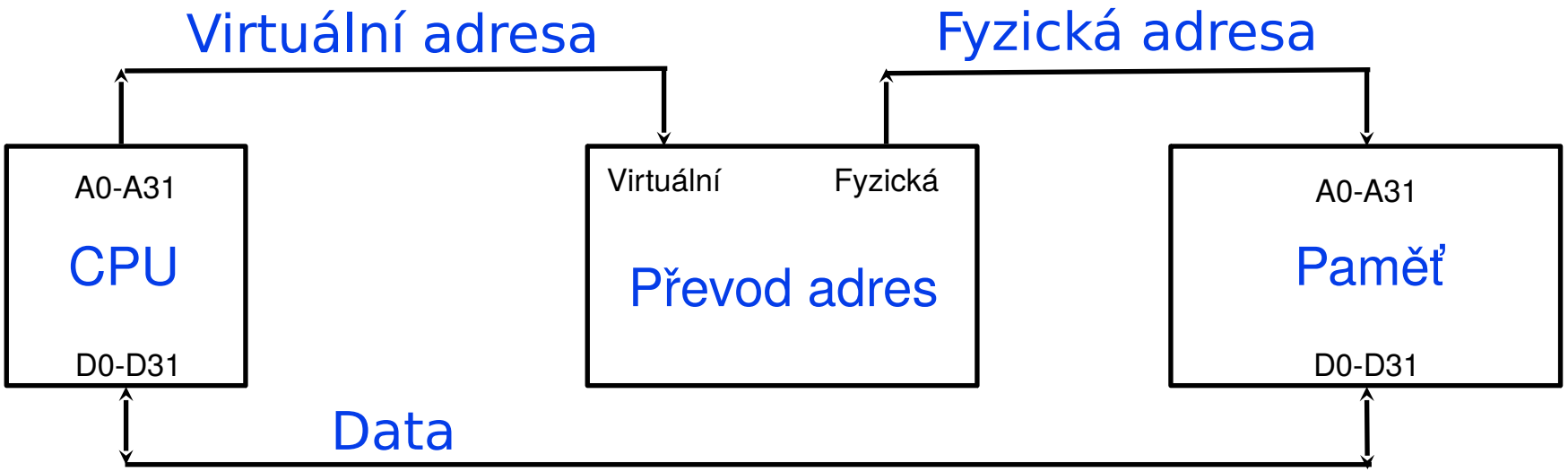
- Virtuální prostor tvoří stejně velké stránky (pages, virtual pages), které se přiřazují jednotlivým běžícím procesům.
- Fyzickou paměť tvoří stejně velké rámce (frames, physical pages).



Virtuální paměť - stránkování

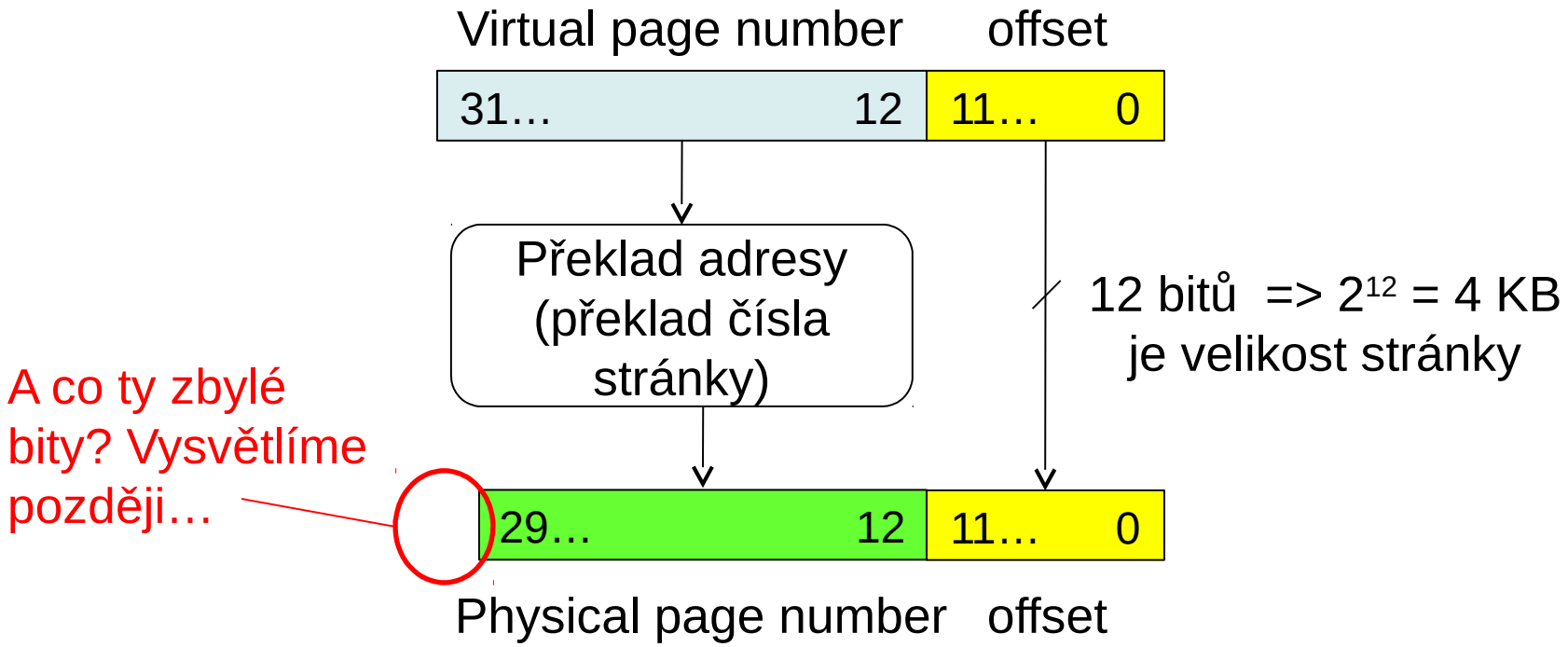
- Každé virtuální stránce může odpovídat nejvýš jedna fyzická stránka, obráceně to neplatí, takže:
- Na jednu konkrétní fyzickou stránku může být namapováno několik virtuálních stránek. Co to přináší?
- Můžeme sdílet paměť napříč různými procesy nebo vlákny (data nebo kód – OS načte sdílené knihovny jenom jednou), můžeme poskytnout jiná oprávnění (přístupová práva).
- Pokud se program snaží přistoupit do stránky způsobem, který neodpovídá jeho oprávněním, CPU generuje *General protection fault*
- handler pro General protection fault – typická reakce je ukončení procesu

Virtuální a fyzické adresování



Virtuální a fyzické adresování - detailněji

- Předpokládejme virtuální adresu o délce 32 bitů, 1GB fyzické paměti a velikost stránky 4 KB



- Jaký **velmi důležitý** praktický důsledek má toto uspořádání překladu (tj. nejnižší bity adresy zůstávají zachovány) ?

Vraťme se k příkladu č.1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, b[4], *c, d;
    c = (int*)malloc(4*sizeof(int));
    printf("%p %p %p %p\n",&a,&b,&c,&d);
    printf("%p %p %p\n",&b[0],&b[1],&b[2]);
    printf("%p %p %p\n",&c[0],&c[1],&c[2]);
    free(c);
    return 0;
}
```

Výpis programu:

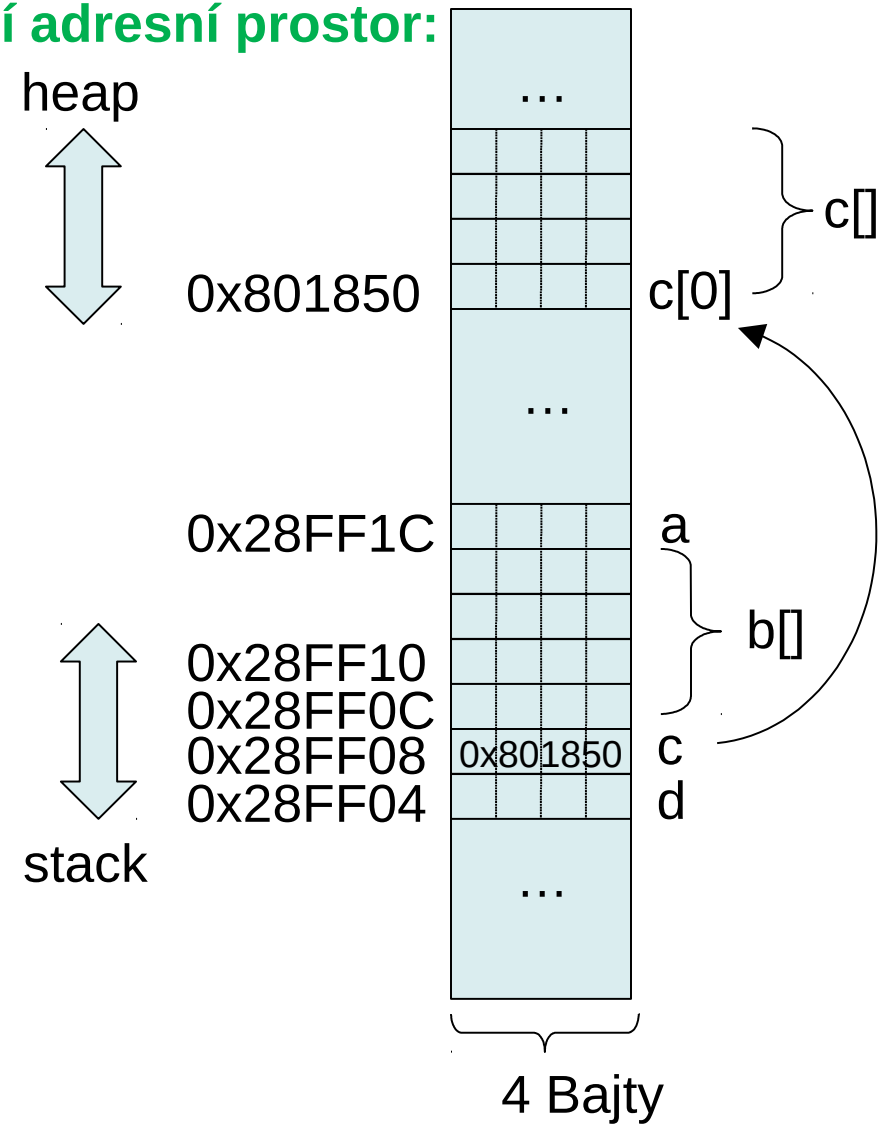
```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

Vraťme se k příkladu č.1

- Všimli jste si adres, na kterých se nachází proměnné ***a***, **c**, ***d*** a pole ***b***?
- Co když budeme chtít rozšířit náš program třeba o příkazy:

```
a = 1;
b[0] = a+1;
b[1] = b[0]+1;
d = b[2];
```

//b[2] neinicializováno..



Vraťme se k příkladu č.1

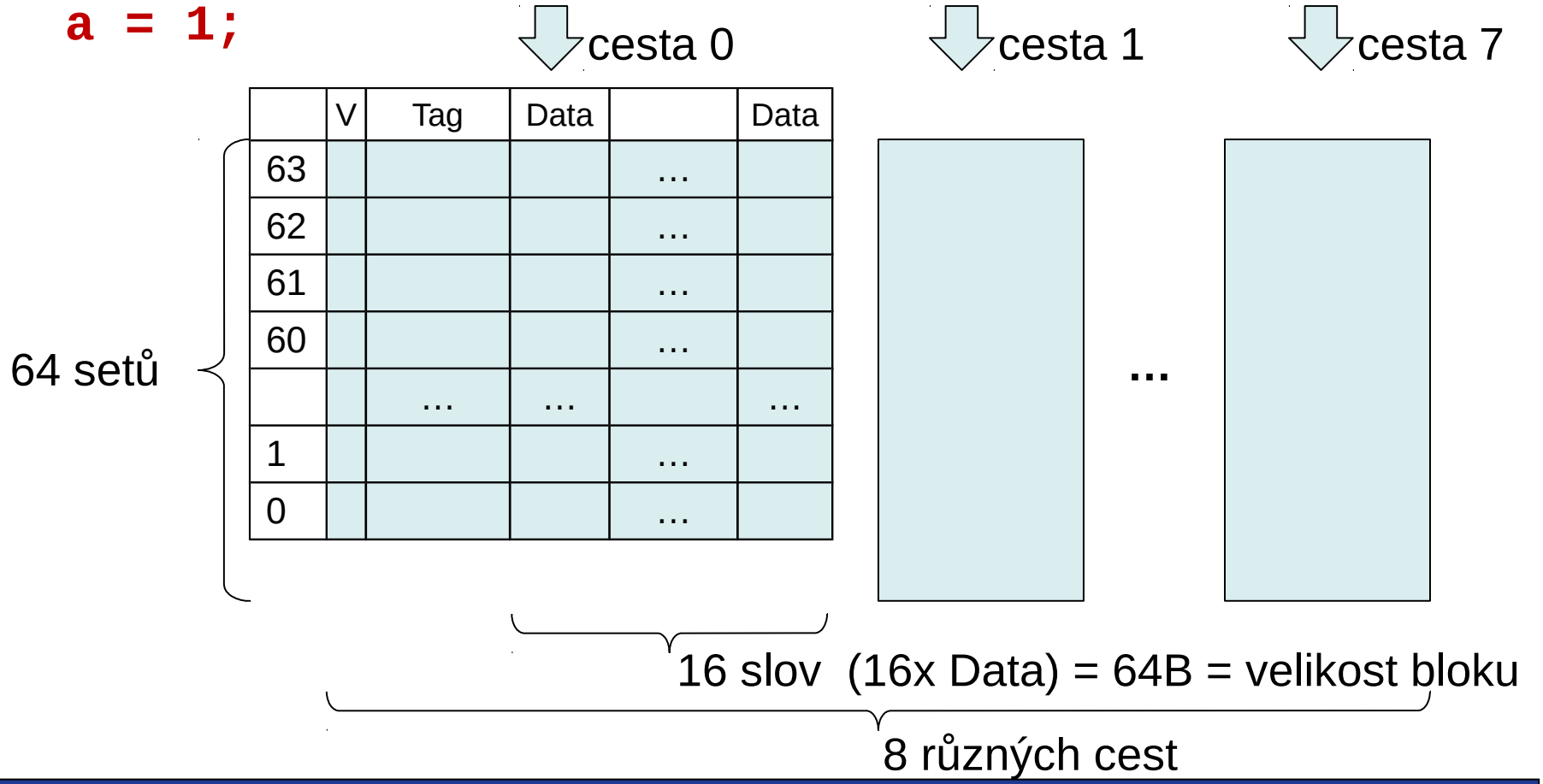
- Předpokládejme L1 datovou cache o velikosti 32kB se stupněm asociativity 8, a velikostí bloku 64B. Cache je na počátku prázdná.
- Co všechno se stane když vykonáme první řádek programu?

```
a = 1;
b[0] = a+1;
b[1] = b[0]+1;
d = b[2];
```

Vraťme se k příkladu č.1

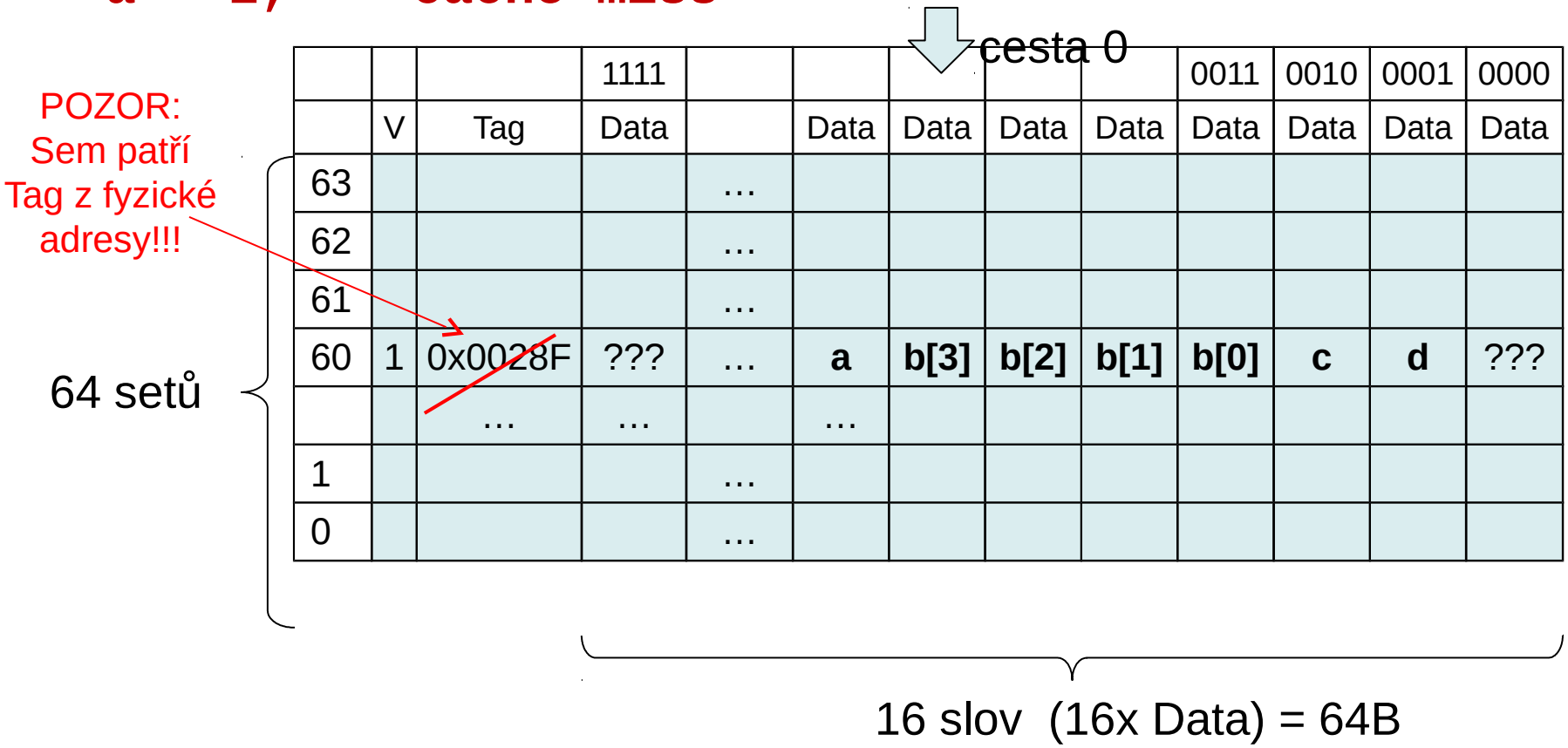
- Předpokládejme L1 datovou cache o velikosti 32kB se stupněm asociativity 8, a velikostí bloku 64B. Cache je na počátku prázdná.
- Co všechno se stane když vykonáme první řádek programu?

```
a = 1;
```



Vraťme se k příkladu č.1

- Předpokládejme L1 datovou cache o velikosti 32kB se stupněm asociativity 8, a velikostí bloku 64B. Cache je na počátku prázdná.
- Co všechno se stane když vykonáme první řádek programu?
  `a = 1; -> cache miss`



			1111						0011	0010	0001	0000
	V	Tag	Data		Data	Data	Data	Data	Data	Data	Data	Data
63				…								
62				…								
61				…								
60	1	0x0028F	???	…	**a**	**b[3]**	**b[2]**	**b[1]**	**b[0]**	**c**	**d**	???
		…	…		…							
1				…								
0				…								

Vraťme se k příkladu č.1

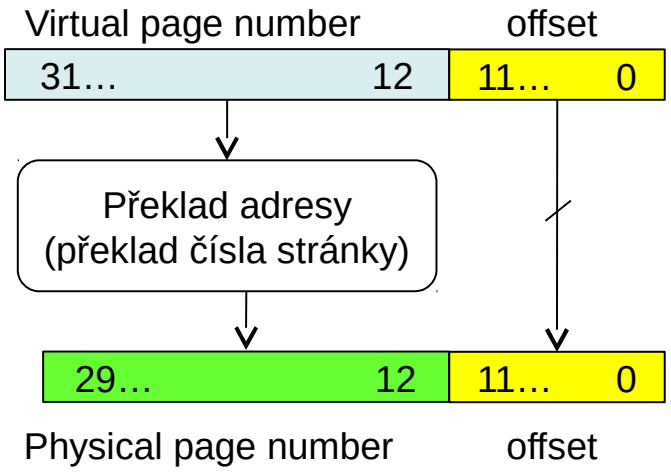
Závěry:

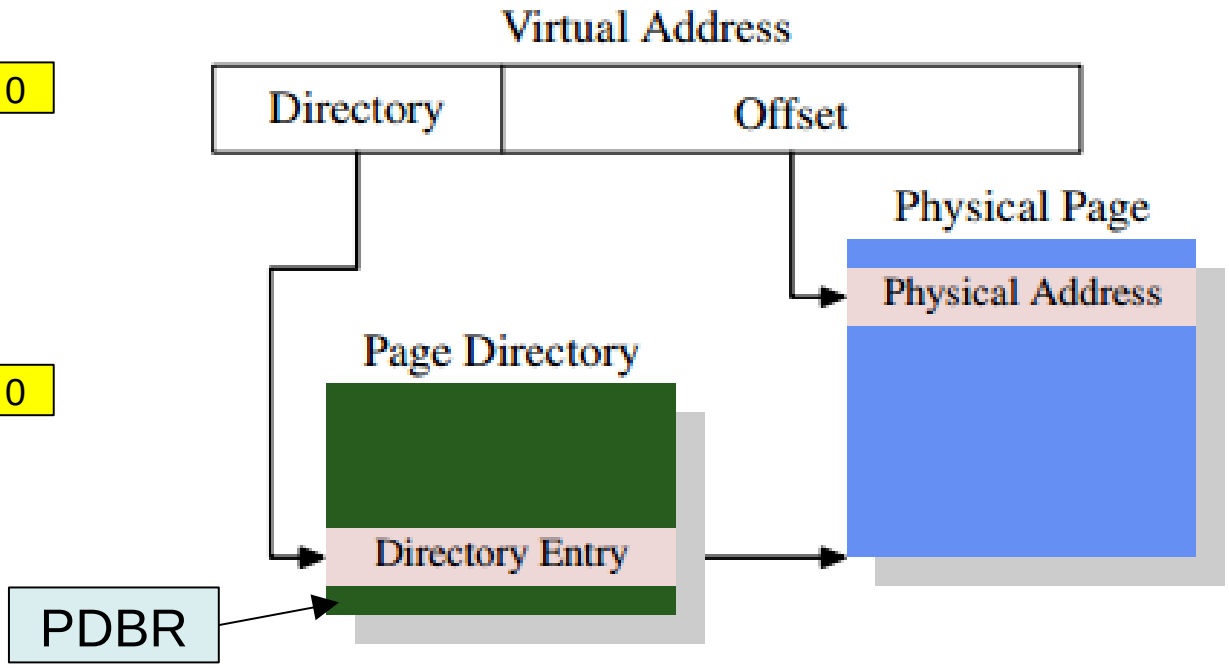
- Stránkování (realizace virtuální paměti) nenarušuje princip prostorové lokality => důležité pro cache.
- **Data na sousedících virtuálních adresách budou uloženy ve fyzické paměti vedle sebe (pokud nepřekročí hranici stránky).**
- Pokud nastane page fault (stránka je na disku) jako důsledek cache miss, pak se celá stránka z disku přesune do paměti a z té se pak celý blok (cache line) přesune do cache. Další cache miss uvnitř stránky již nevyvolá page fault (dokud nebude stránka nahrazena jinou stránkou).

Realizace převodu adres?

- Tabulka stránek, Page Table.
- Jednotkou mapování jsou stránky,
- Stránka je také jednotkou přenosu mezi vedlejší a hlavní paměti.
- Mapovací funkce se nejčastěji implementuje Look-up Table (vyhledávací tabulkou).
- O překlad virtuálních adres na fyzické se stará **Memory Management Unit** (**MMU**)
- MMU je součástí CPU
- Příklad:

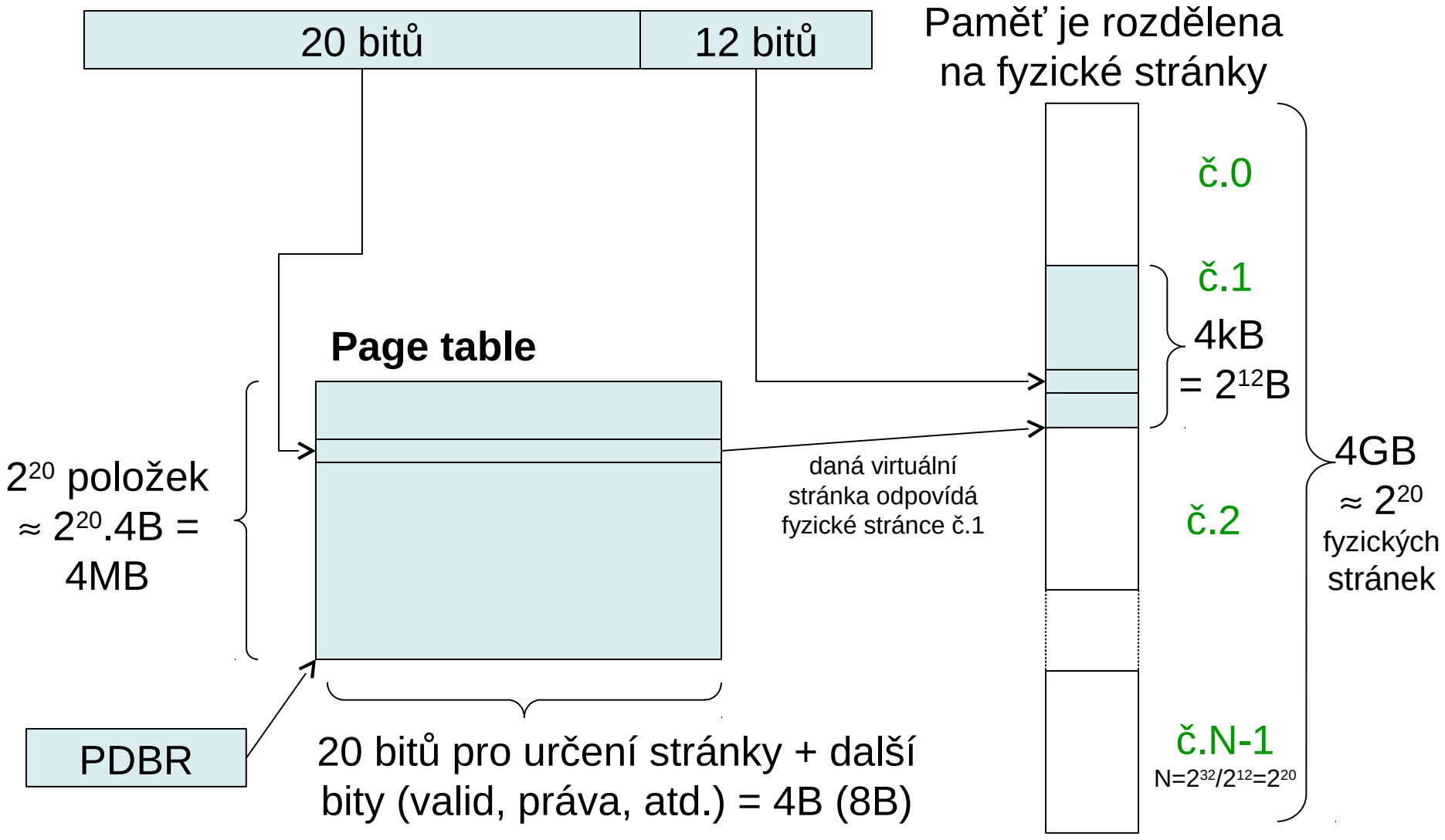
Realizace převodu adres?





- Datová struktura pro Page Directory (Page Table) je uložena v hlavní paměti. Úkolem operačního systému je alokovat souvislou oblast paměti a počáteční adresu této oblasti uložit do speciálního registru CPU.
- PDBR - page directory base register – v x86 v registru CR3 – obsahuje fyzickou adresu
- PTBR - page table base register – to samé…

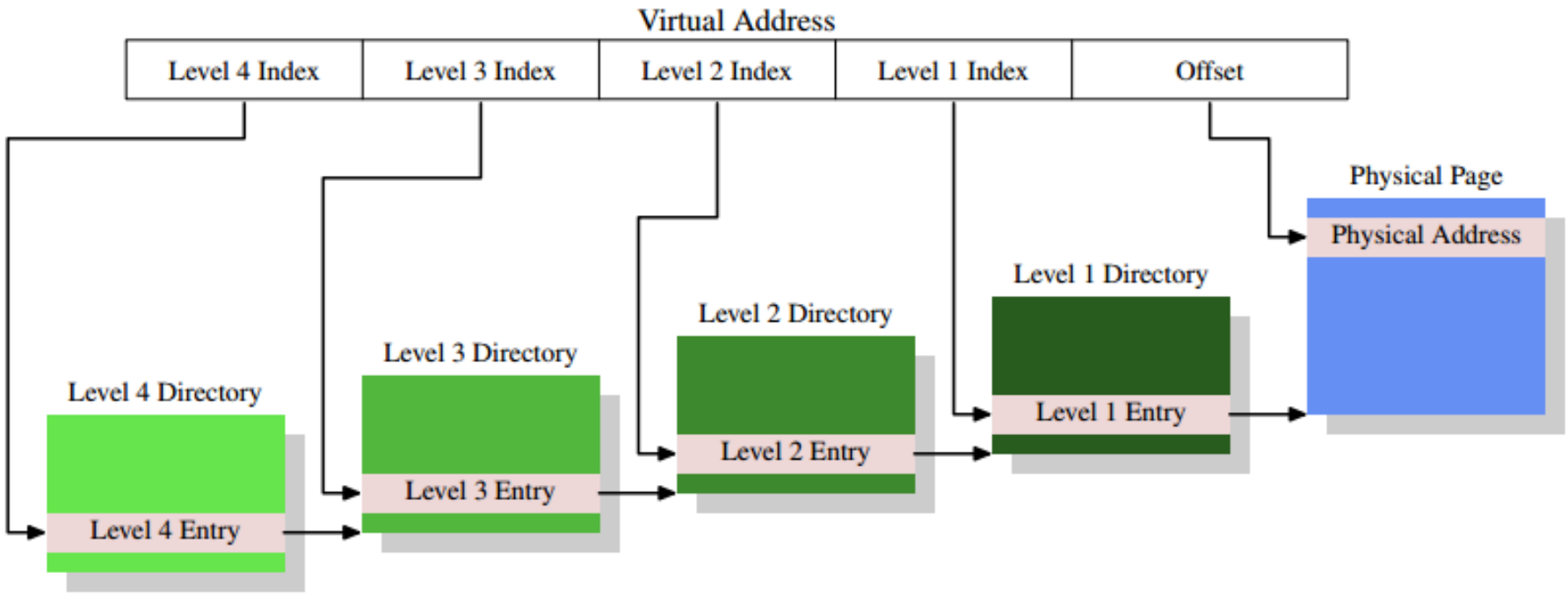
Realizace převodu adres?



Uvažujme…

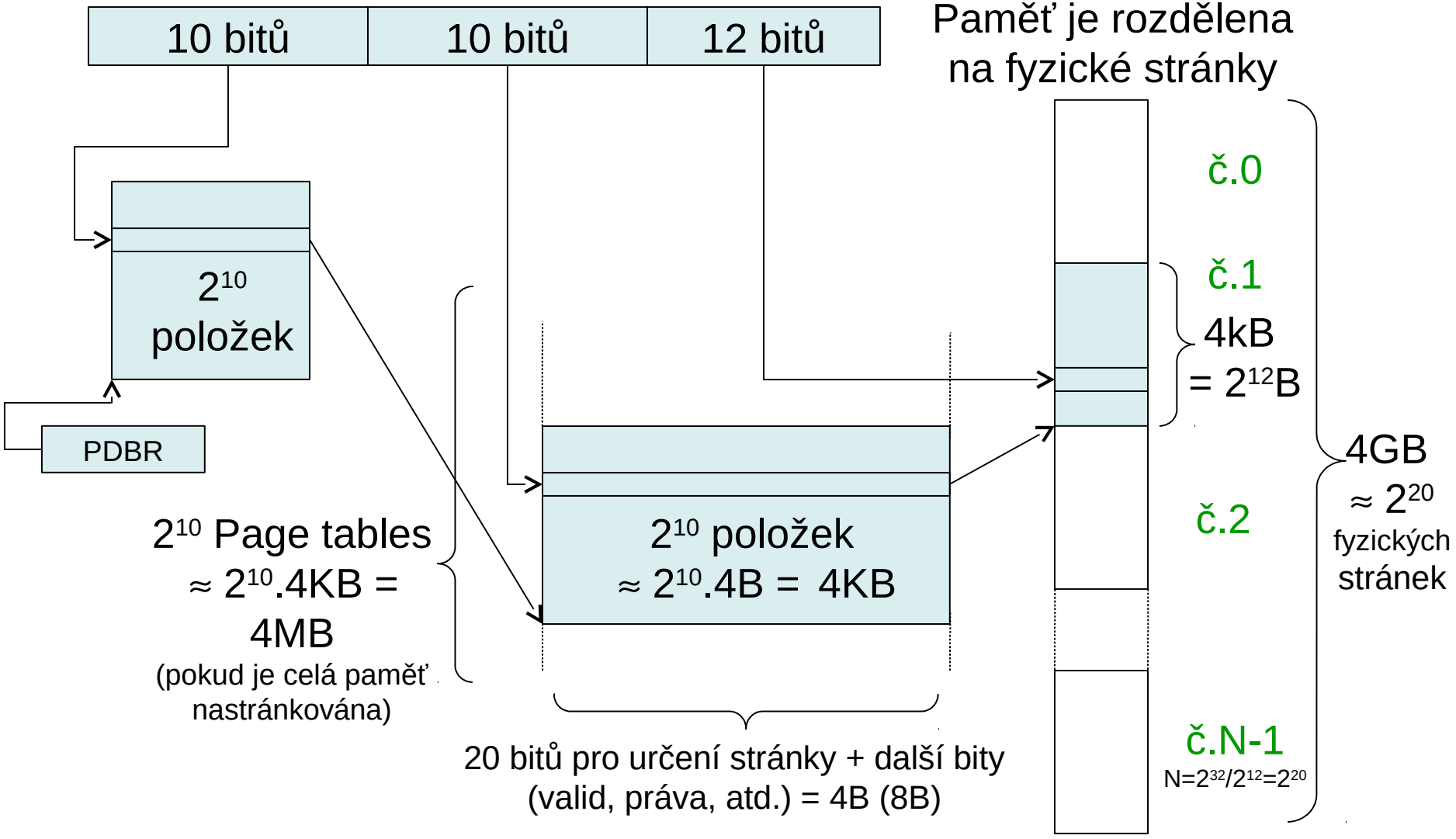
- Stránka je typicky 4 kB = 2^12
- Když budeme znát adresu stránky, postačuje nám tedy jenom 12 bitů na pohyb (adresaci) v ní. Zbývá 20 bitů (pro 32-bitovou adresu).
- Tudíž Page Directory (Page Table) by měl obsahovat 2^20 položek. To je nepraktické a přináší řadu nevýhod.
- Typický proces/vlákno se v daném „okamžiku“ pohybuje pouze v malé části svého adresního prostoru – princip časové a prostorové lokality…
- Řešením je více-úrovňové stránkování.

Více-úrovňové stránkování



4-Level Address Translation

Více-úrovňové stránkování – 2 úrovně



Více-úrovňové stránkování

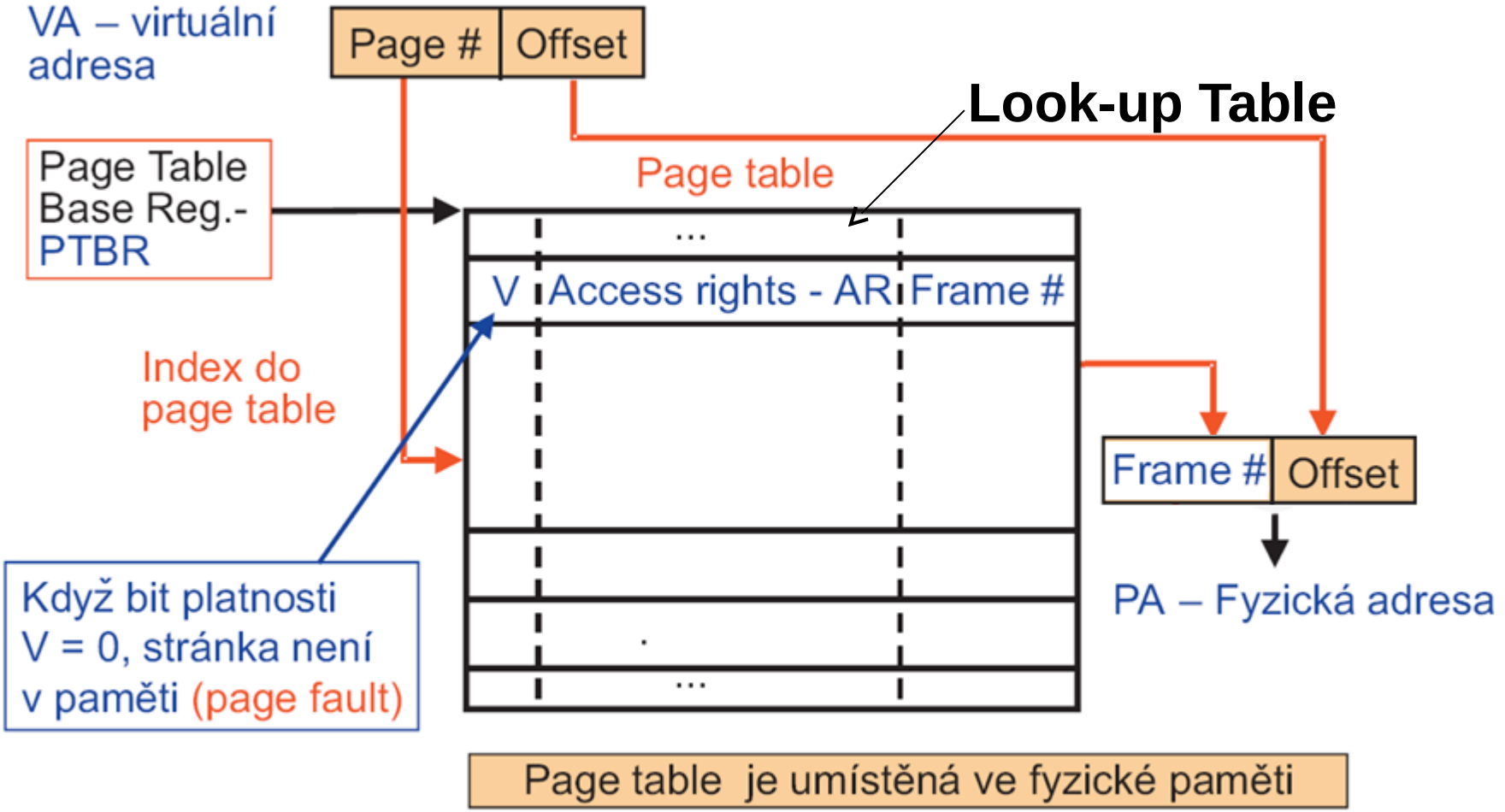
Poznámky k předchozímu slide:

- Ne každý proces využívá celý svůj adresní prostor => není nutné alokovat v druhé úrovni 2^{10} Page tables
- Tabulky stránek můžou být rovněž stránkovány

Obecné poznámky:

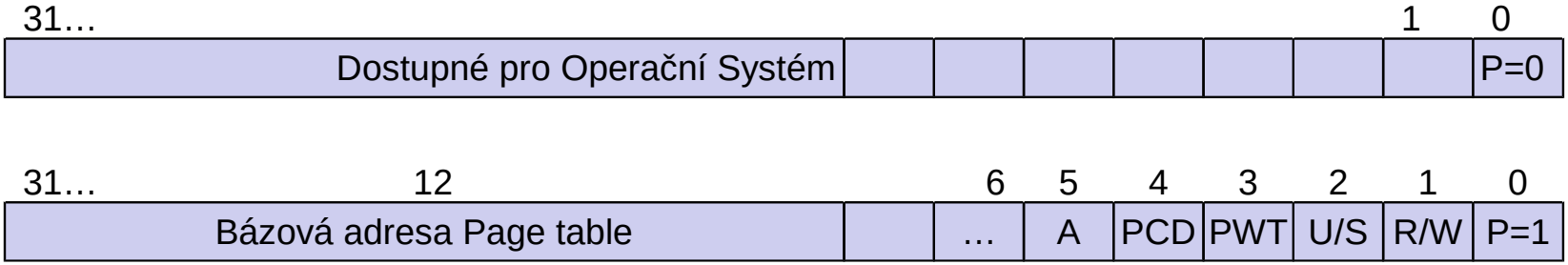
- Intel IA32 implementuje 2-úrovňové stránkování
  - Page Table v úrovni 1 označuje jako Page Direcory (10 bitů pro adresaci)
  - Page Table v úrovni 2 pak jako Page Table (10 bitů)
- V případě 64-bitové virtuální adresy je obvyklé používat méně bitů pro fyzickou adresu – například 48, nebo 40.
- Intel Core i7 používá 4-úrovňové stránkování a 48 bitový adresní prostor
  - Page Table v úrovni 1: Page global directory (9 bitů)
  - Page Table v úrovni 2: Page upper directory (9 bitů)
  - Page Table v úrovni 3: Page middle directory (9 bitů)
  - Page Table v úrovni 4: Page table (9 bitů)

Tabulka stránek – jak vypadají položky? Význam položek…



Tabulka stránek – jak vypadají položky? Význam položek…

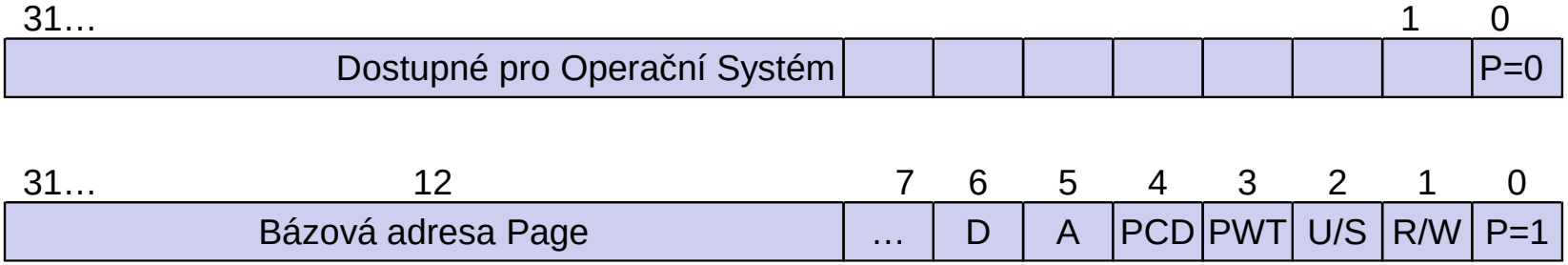
Podívejme se na položku Page Directory (Page Table na 1.úrovni)



- bit 0: Present bit – určuje zda je stránka v paměti (1), nebo na disku (0) Někdy se tento bit označuje V – valid.
- bit 1: Read/Write: pokud 1 – R/W; pokud 0 – jenom ke čtení
- bit 2: User/Supervisor: 1 – uživatelský přístup; 0 – pouze OS
- bit 3: Write-through/Write-back – zapisovací strategie pro stránku
- bit 4: Cache disabled/enabled – některé periferie jsou namapovány přímo do paměti (memory mapped I/O), čímž je umožněn zápis/čtení do/z dané periferie. Tyto adresy v paměti pak chápeme jako I/O porty. Nekešujeme.
- bit 5: Accessed – zda jsme četli/zapsali – pomáhá rozhodovat o tom, které stránky mají být odstraněny když potřebujeme uvolnit prostor v paměti

Tabulka stránek – jak vypadají položky? Význam položek…

Podívejme se na položku Page Table (Page Table na 2.úrovni)

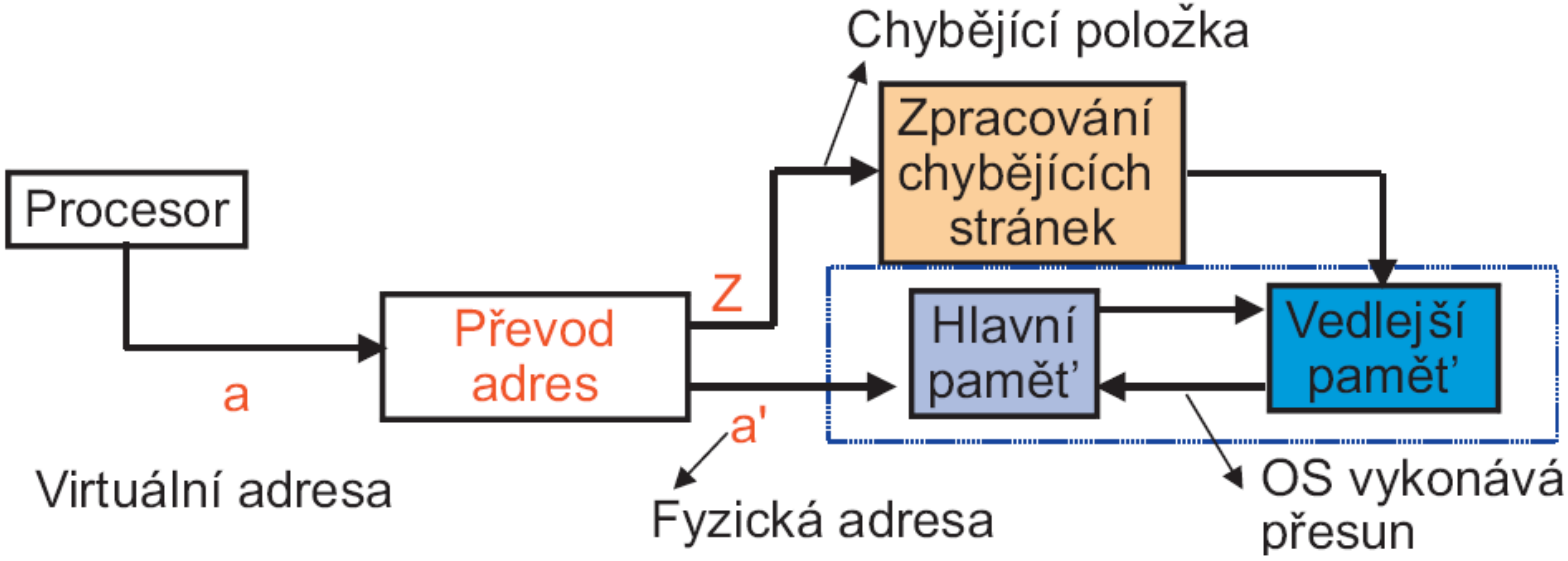


- bit 6: Dirty bit – Je nastaven pokud jsme do stránky zapsali.

Poznámky

- Každý proces má svou Tabulku stránek,
- Tedy i svou hodnotu PTBR (bázového registru).
- To, mimochodem, zajišťuje paměťovou bezpečnost procesů.
- Co chceme abyste si zapamatovali z Formátu položky Tabulky stránek?
  - V – Validity Bit. V=0 Stránka není platná (je na disku).
  - AR – Access Rights. Přístupová práva (Read Only, Read/Write, Executable, apod.),
  - Frame# - číslo rámce (bázová adresa do nižší úrovně),
  - Popřípadě další, např. Modified/Dirty, apod. (budeme dále podle potřeby doplňovat).

Virtuální paměť: spolupráce HW a SW



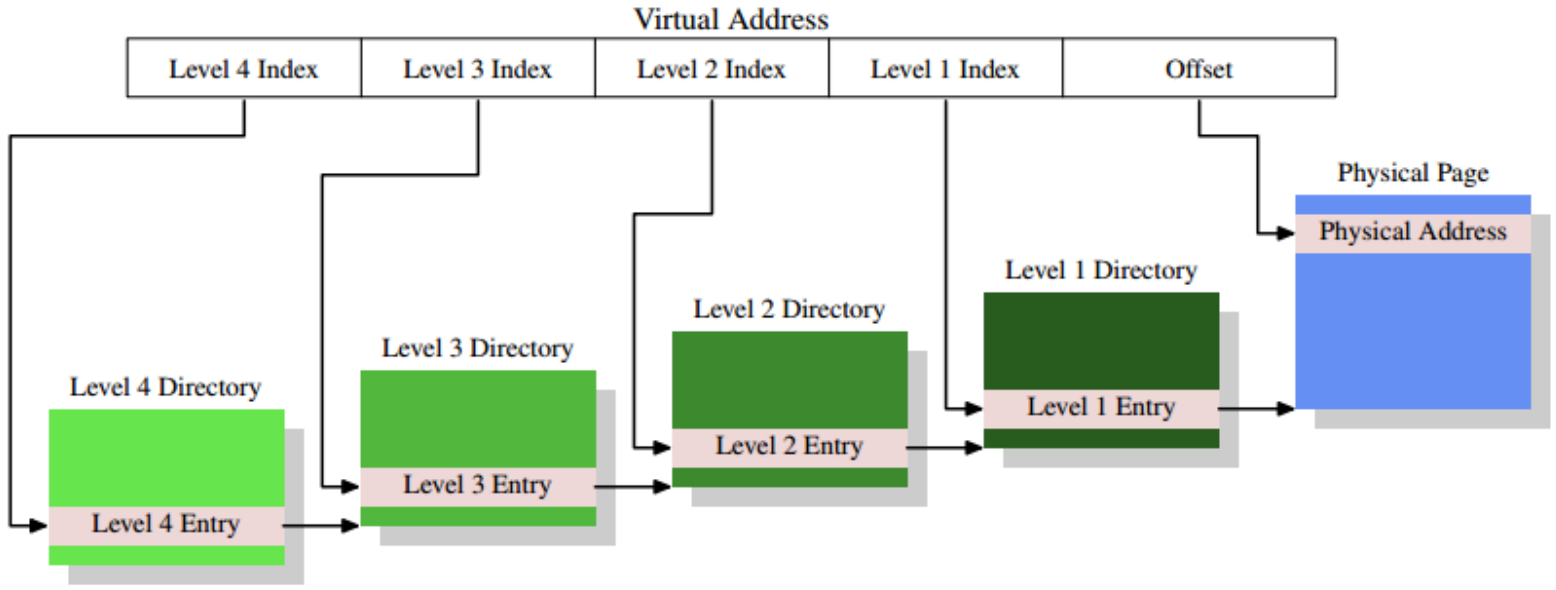
Co dělat, když je výpadek stránky – Page Fault?

- Fyzická paměť je volná, ale
  - Rámec je prázdný, data jsou ve vedlejší paměti (na disku).
  - Požadovaná stránka se „nějak“ (DMA, Direct Memory Access, přímým přístupem do paměti, ale to zde neřešíme) načítá do prázdného rámce. Přepne se na případně čekající proces, který může probíhat.
  - Po dokončení DMA přenosu se vyvolá přerušení, aktualizuje se Tabulka stránek procesu.
  - Přepne se zpět na původní proces.
- Paměti je nedostatek
  - Pomocí LRU najdeme rámec, který můžeme uvolnit.
  - Má-li nastaven Dirty bit, zapíšeme stránku do vedlejší paměti (na disk).
  - Aktualizuje se Tabulka stránek procesu.

Virtuální paměť a soubory na disku…

- Virtuální paměť rozšiřuje fyzickou paměť o prostor na disku tím, že automaticky odkládá/načítá stránky na disk (swapování). Toho lze využít…
- **Načtení programů a knihoven do paměti:**
  - Programy a knihovny jsou uloženy na disku jako binární soubory obsahující instrukce a data
  - Když chceme spustit nový program:
    - Jádro OS alokuje souvislou množinu virtuálních stránek (dostatečně velký prostor pro uchování vlastního programu a dat)
    - Poté OS aktualizuje Page table procesu (Page tables pak odkazují na soubory na disku)
    - Položky Page table jsou označeny jako Valid=0 (na disku)
  - Jakmile program běží, správa virtuální paměti načte program do paměti automaticky…
  - Viz **mmap()** – funkce alokuje virtuální stránky a nastaví položky Page table tak, aby odkazovaly na soubor na disku

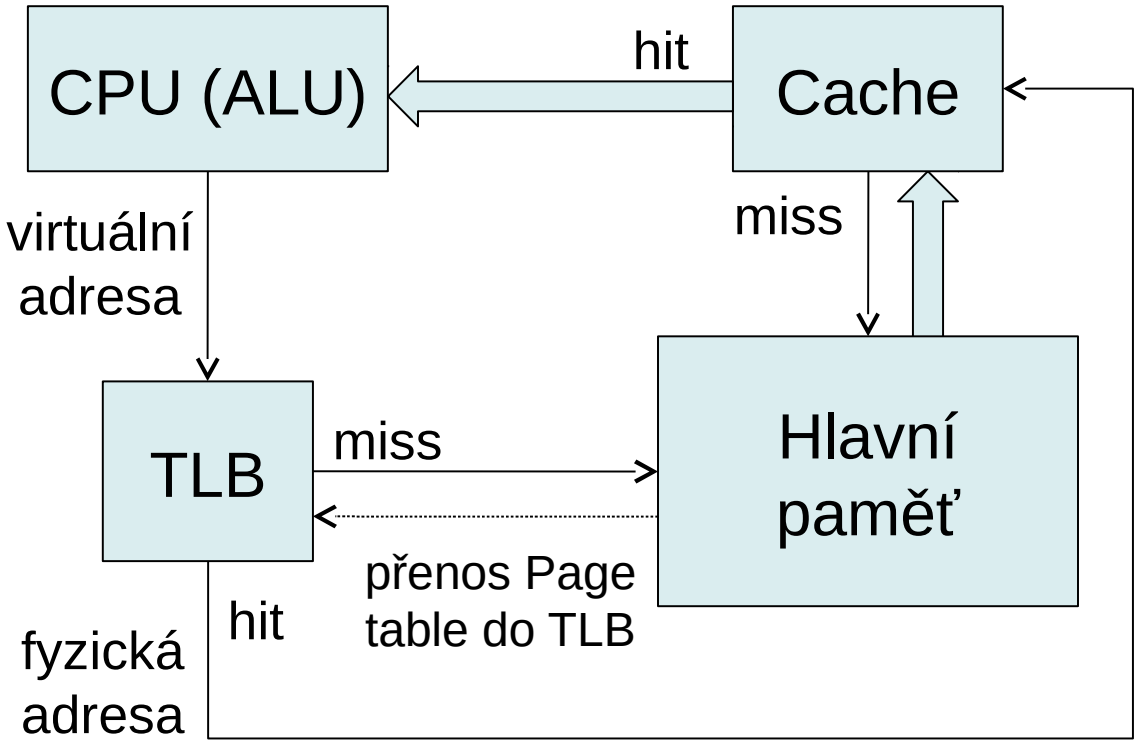
Více-úrovňové stránkování – **Problém rychlosti**



4-Level Address Translation

- Pokud bychom předpokládali, že všechny položky pro výpočet adresy máme již v cache, bude i tak výpočet adresy trvat velmi dlouhou (v závislosti od počtu úrovní – nelze paralelizovat).
- Výhodnější je přímo cachovat „vypočtené“ adresy.
- K tomu slouží Translation Look-Aside Buffer (TLB)
- Dnes se používají více-úrovňové TLB

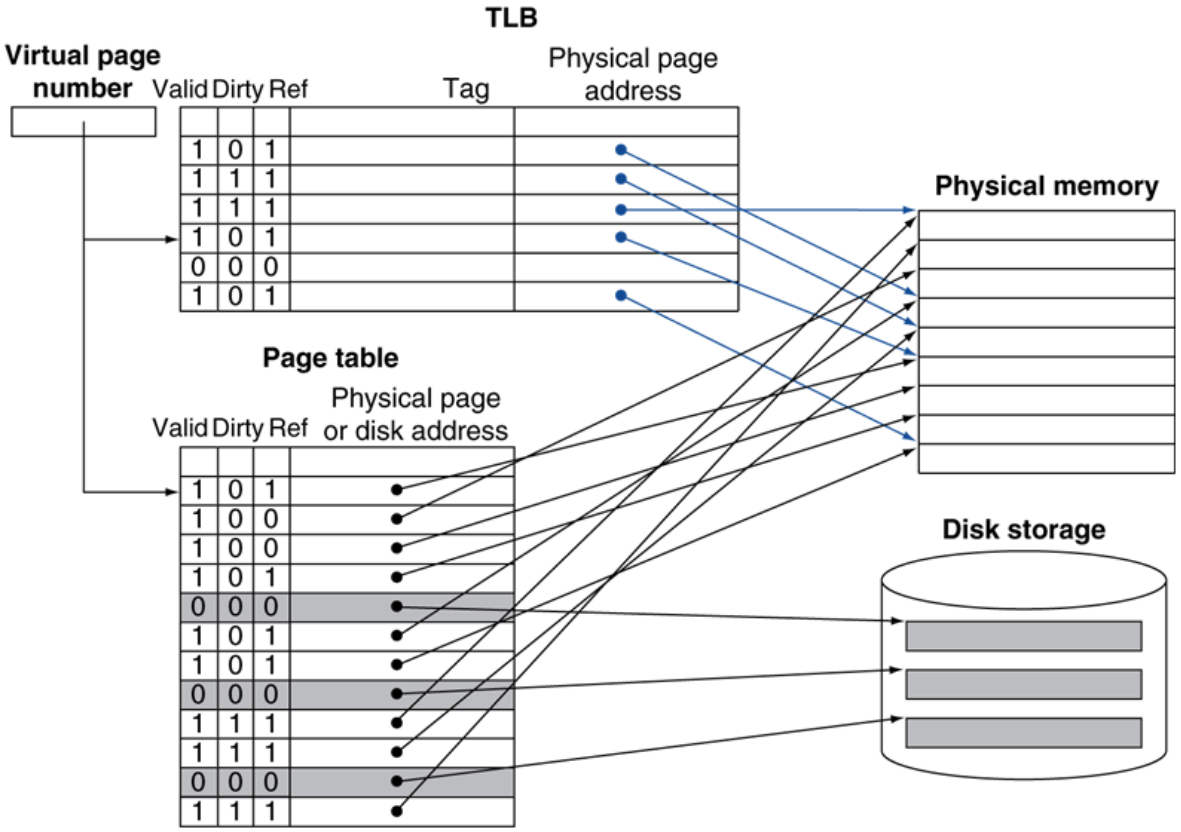
Idealizace překladu adres pomocí TLB - čtení



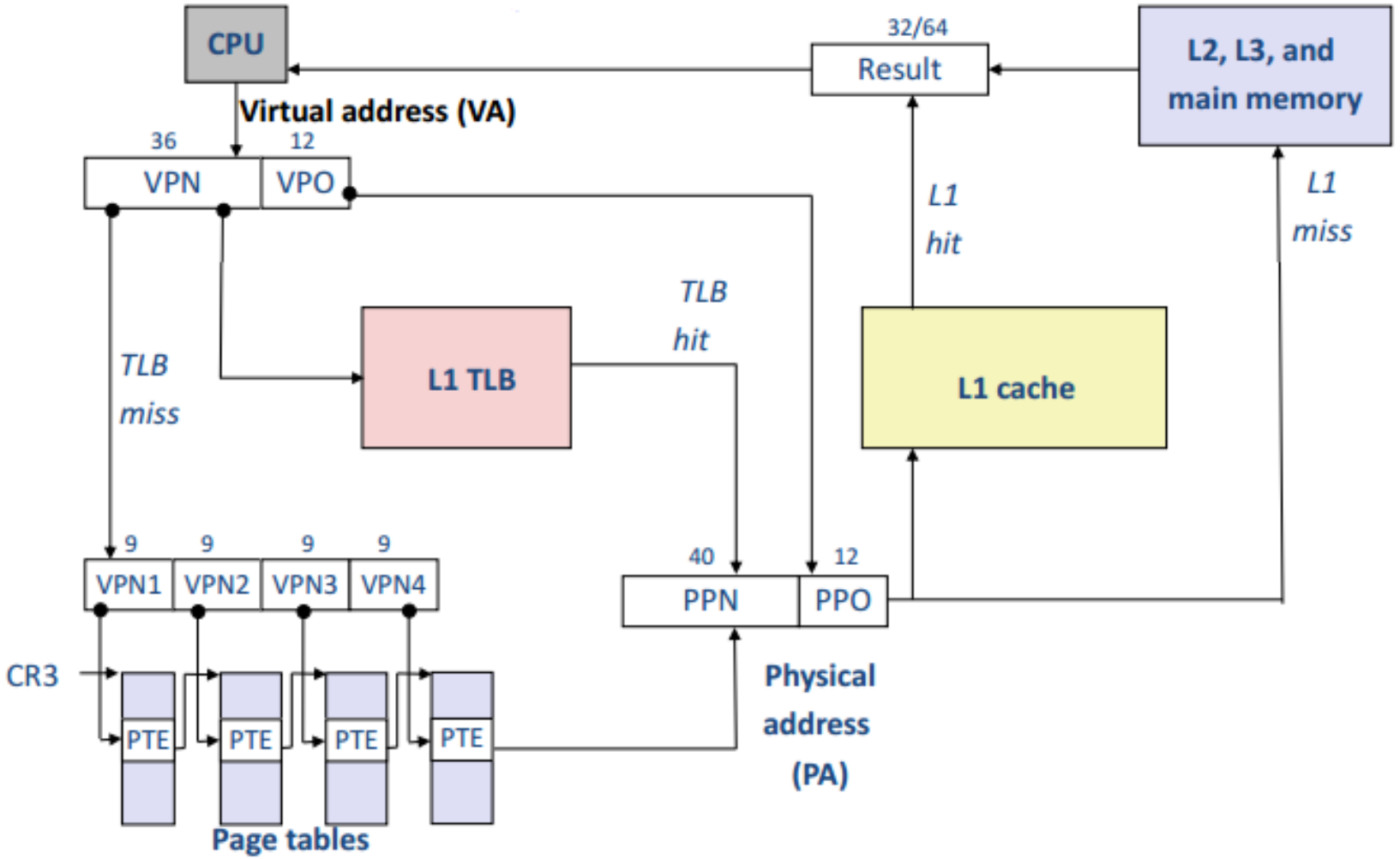
- Všimněte si, že může dojít k miss-u 2x
- Pokud nastane TLB miss, musíme vykonat tzv. *page walk*

Rychlá realizace Tabulky stránek - TLB

- Translation-lookaside Buffer, výstižnější by byl termín překládací keš (Translation Cache).
- Je vlastně skrytá paměť (cache) adres stránek.

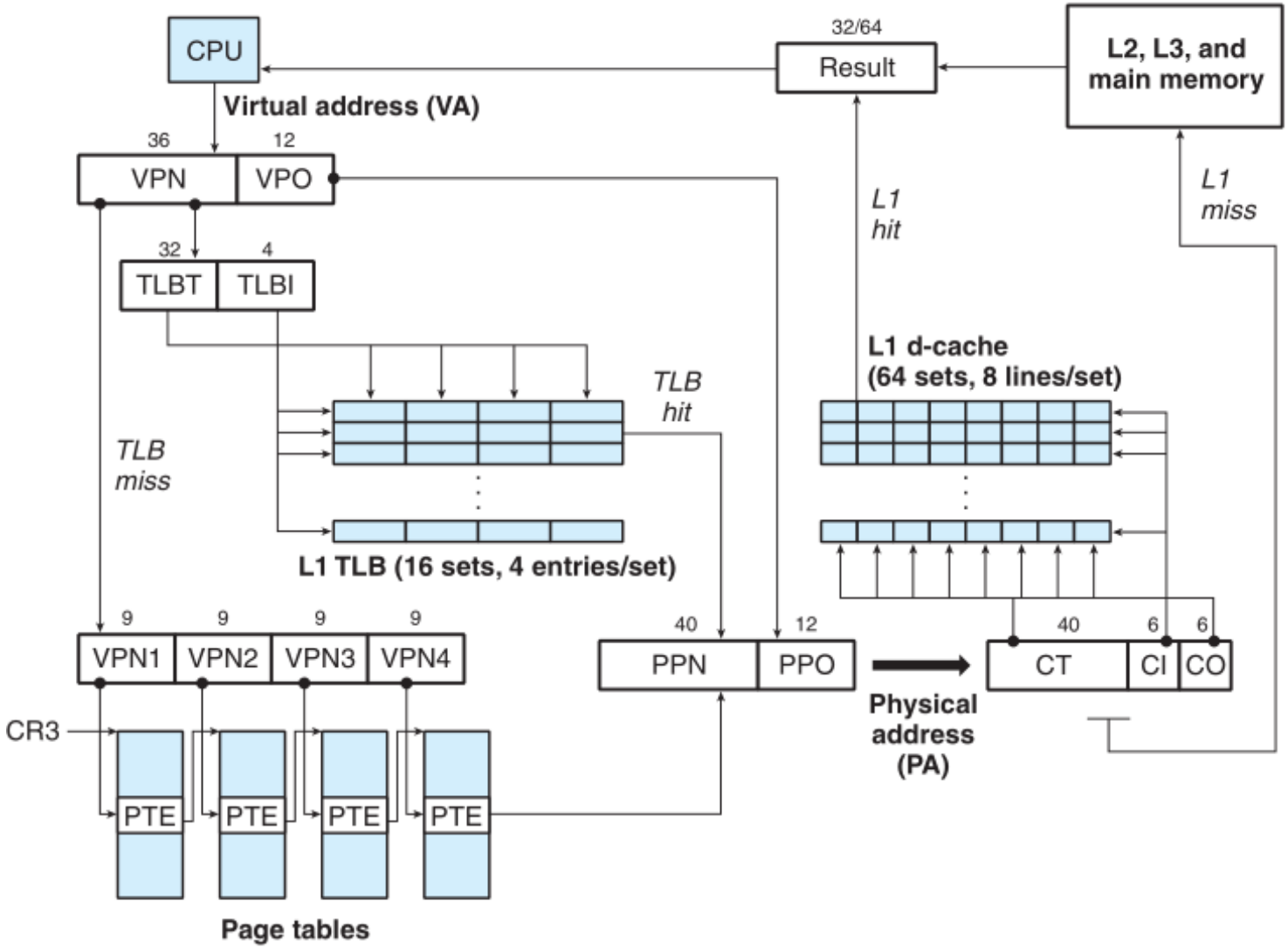


Překlad adres – Intel Nehalem (Core i7)

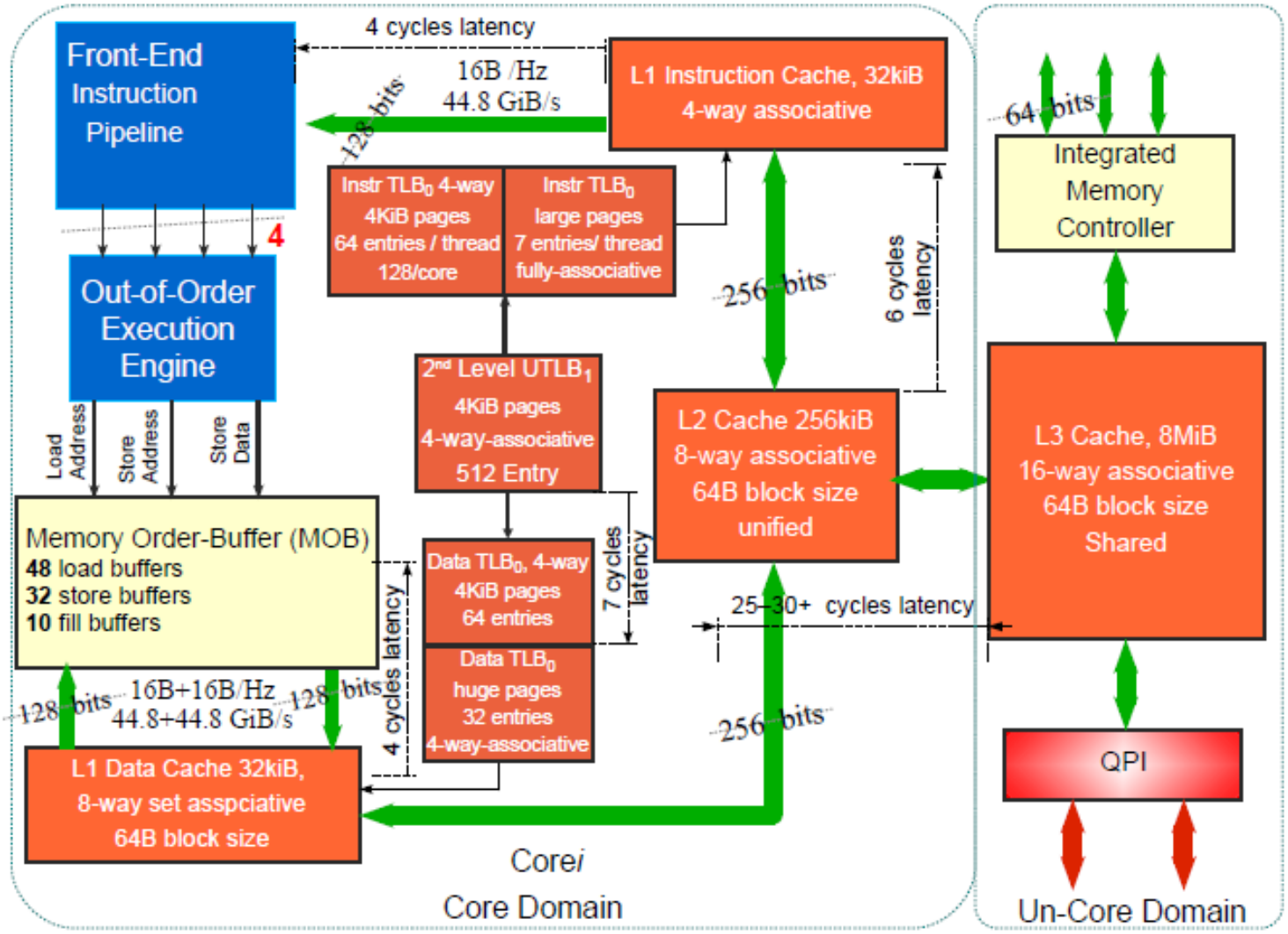


http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/lecture18.pdf

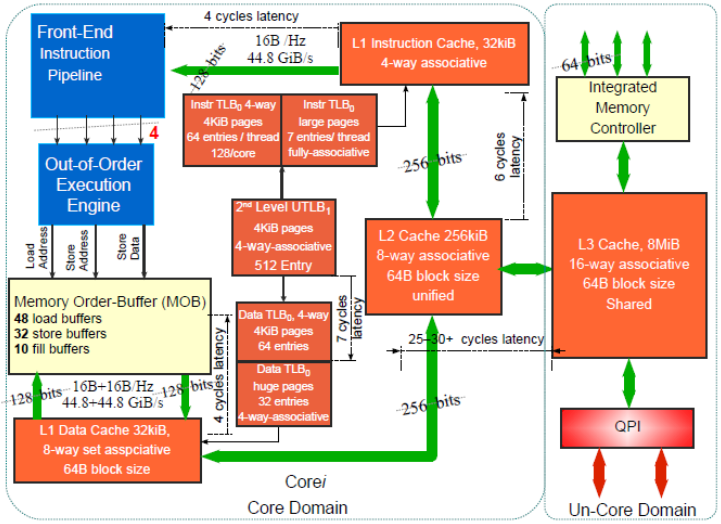
Překlad adres – Intel Nehalem (Core i7) – detailněji



Organizace paměti - Intel Nehalem (Core i7)

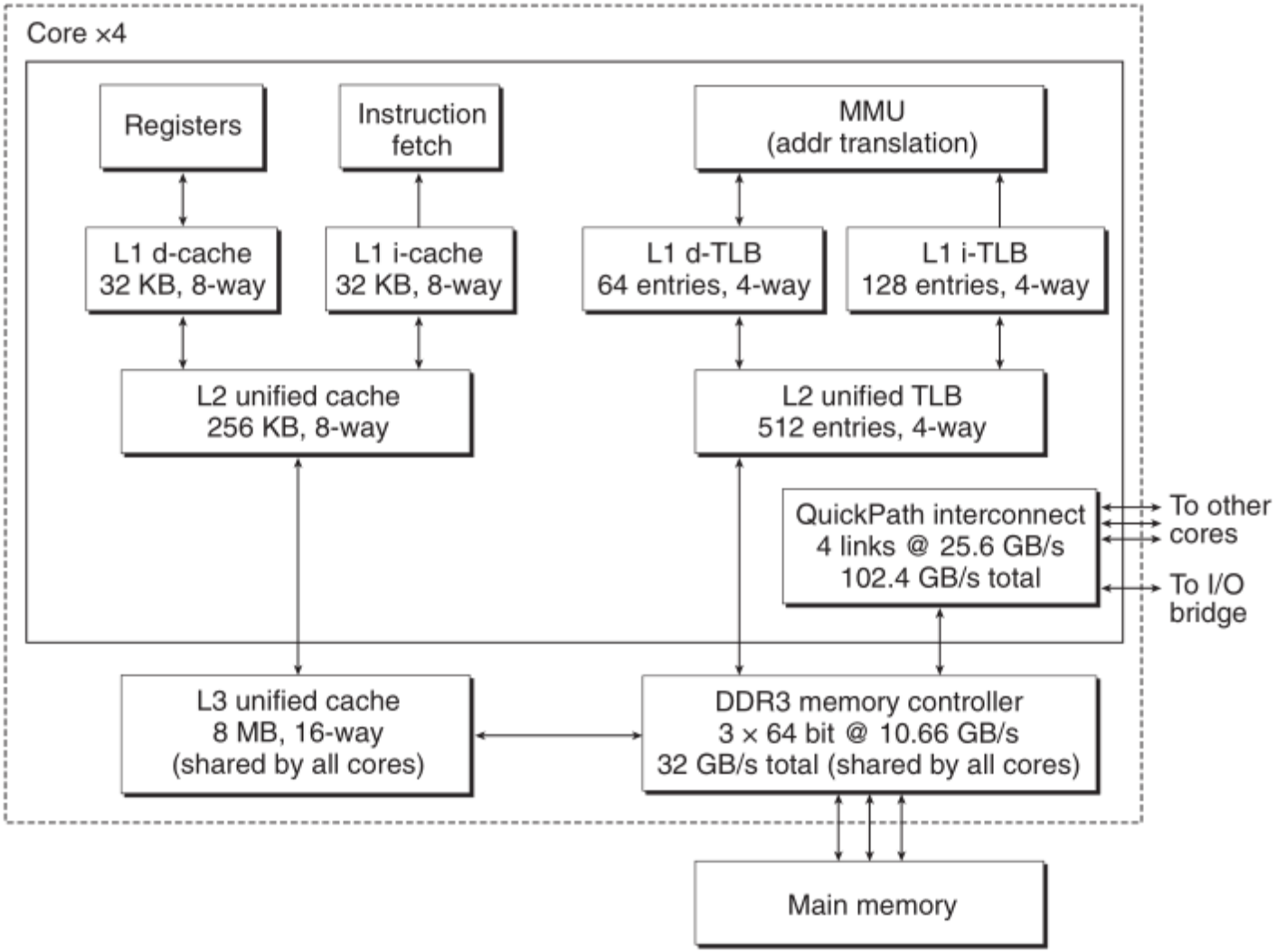


Organizace paměti - Intel Nehalem – několik poznámek



- Velkost bloku: 64B
- procesor vždy čte řádek cache ze systémové paměti zarovnán na 64B (6 LSb adresy jsou nuly) a nepodporuje částečně plněné řádky
- L1 – Harvard. V SMT sdílená oběma vlákny, Instrukční – 4-way, Datová 8-way.
- L2 – unifikovaná, 8-way, neinkluzivní, WB
- L3 – unifikovaná, 16-way, inkluzivní (řádek obsažen buď v L1 nebo L2 se nachází v L3), WB
- Store Buffers – dočasně uchovávají data pro každý zápis. Netřeba čekat na zápis do cache či paměti. Zajišťují, že zápisy jsou ve správném pořadí a také když je potřeba:
  - výjimka, přerušení, instrukce serializace, lock,..
- Můžete si také všimnout oddělených TLB (Translation Lookaside Buffer)

Intel Core i7 – to samé, jiný pohled



Pro vaší představu: typické hodnoty

	Typicky pro stránkované paměti	Typicky pro TLB
Velikost v blocích	16 000-250 000	40-1024
Velikost	500-1 000 MB	0,25-16 KB
Velikost bloku v B	4 000-64 000	4-32
Miss penalty (v hod)	10 000 000 – 100 000 000	10-1 000
Miss rates	0,00001-0,0001%	0,01-2

Efektivnější používáni paměti – prostředek zrychlení programu

Váš program může brát v potaz velikost stránky a používat paměť efektivněji – jednak zarovnáním alokací na násobek velikosti stránky a pak redukcí interní a externí fragmentace stránek.. (pořadí alokací atd. Viz také *memory pool*)

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
   printf(„Velikost stranky je: %ld B.\n",
          sysconf(_SC_PAGESIZE));
   return 0;
}
```

Akolace paměťově zarovnaného bloku:

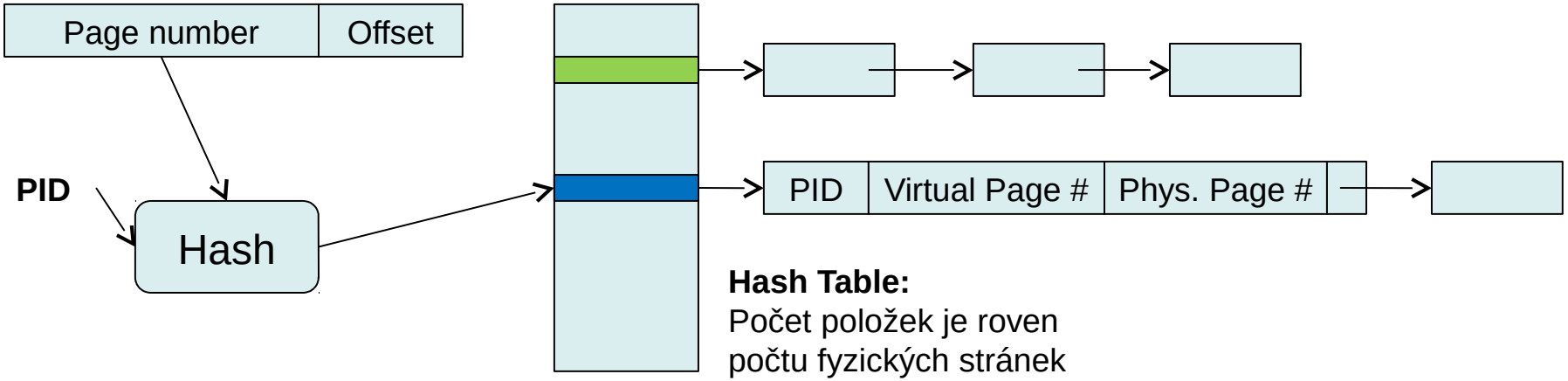
```
void * memalign(size_t size, int boundary)
void * valloc(size_t size)
```

windows

```
#include <stdio.h>
#include <windows.h>

int main(void) {
    SYSTEM_INFO s;
    GetSystemInfo(&s);
    printf("Velikost stranky je: %ld B.\n",
       ns.dwPageSize);
    printf("Rozsah adres pro aplikaci (a dll):
       0x%lx – 0x%lx\n",
       s.lpMinimumApplicationAddress,
       s.lpMaximumApplicationAddress);
    return 0;
}
```

Jsou **hierarchické stránkovací tabulky** jedinou možností?

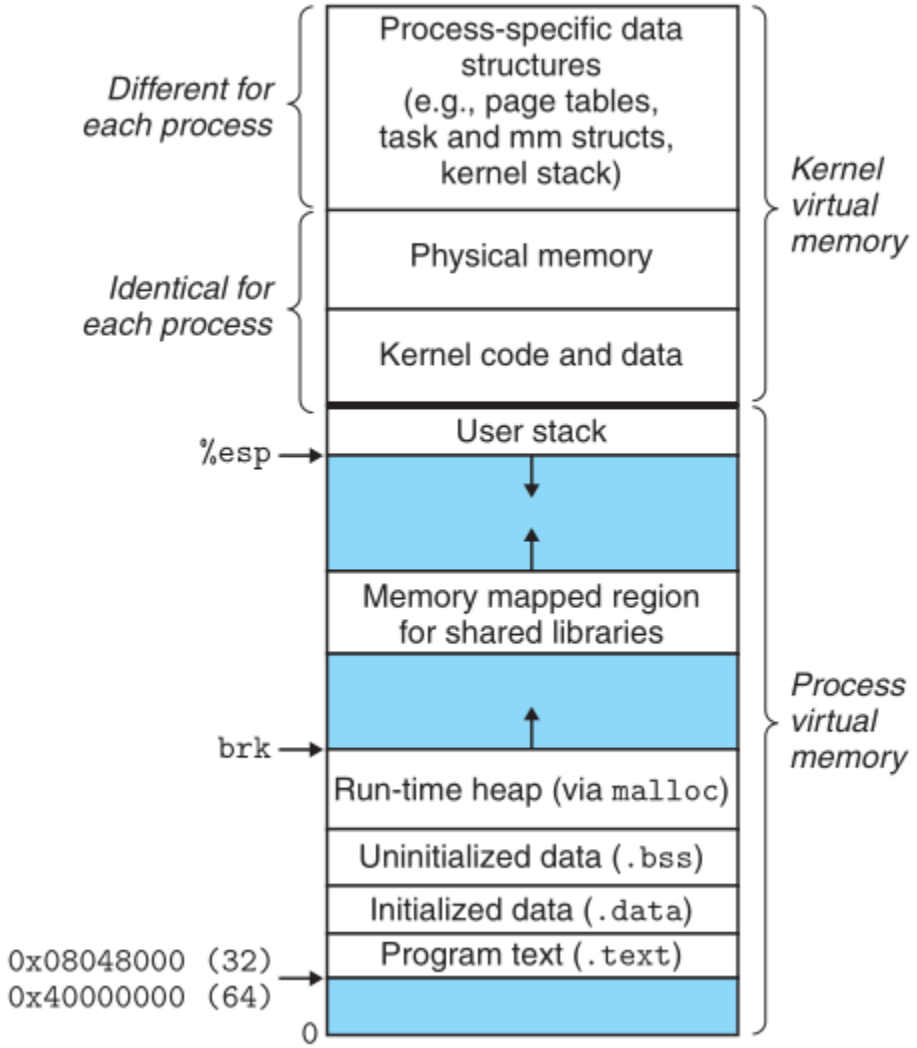
- Hierarchické stránkovací tabulky de facto představují stromovou strukturu, kterou je pak nutno prohledávat
- Ještě existuje alternativa: **Inverted Page Tables**
- 64-bitový virtuální adresní prostor je dosti velký, fyzická paměť mnohem menší -> výrazná disproporce
- **Idea:** Fyzická paměť je rozdělena na stránky. Abychom určili, které virtuální stránky jsou přirazeny dané fyzické stránce, postačuje nám tabulka o takovém počtu řádků, kolik je fyzických stránek
- Problémem je špatná prostorová lokalita (kešovatelnost) v důsledku hashe



Virtuální paměť z pohledu Linuxu

Pojmy – dejme věci do souvislostí

- Linux organizuje VP jako kolekci oblastí - **areas (segments)**
- **Area** je souvislý blok existující (alokované) virtuální paměti, který má nějaký význam. Příklad: code segmant, data segment, heap, shared library segment, user stack.
- **Každá existující virtuální stránka patří do nějaké oblasti.**
- Použití oblastí (areas/segments) umožňuje organizovat VP s „mezerami“ – segmenty nemusí nutně ležet vedle sebe

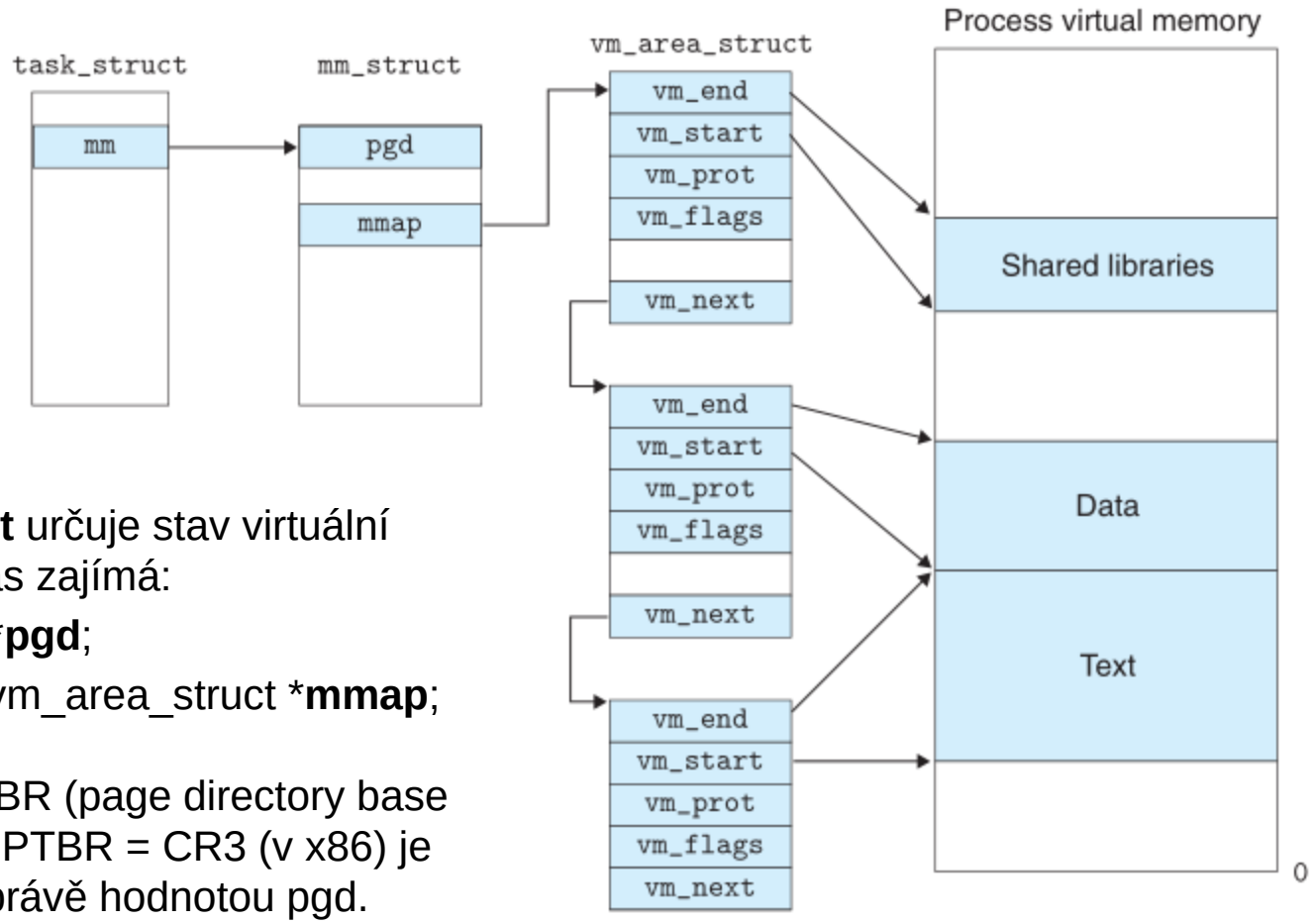


struct task_struct

```
struct task_struct {
  volatile long       state; /* -1 unrunnable, 0 runnable, >0 stopped */
  long                counter;
  long                priority;
  unsigned            long signal;
  unsigned            long blocked;   /* bitmap of masked signals */
  unsigned            long flags;     /* per process flags, defined below
        */
  int errno;
  long                debugreg[8];    /* Hardware debugging registers */
  struct exec_domain   *exec_domain;
  struct linux_binfmt  *binfmt;
  struct task_struct   *next_task, *prev_task;
  struct task_struct   *next_run,  *prev_run;
  unsigned long        saved_kernel_stack;
  unsigned long        kernel_stack_page;
  int                 exit_code, exit_signal;
  unsigned long        personality;
  int                 dumpable:1;
  int                 did_exec:1;
  int                 pid;
  int                 pgrp;
  int                 tty_old_pgrp;
  int                 session;
  int                 leader;
  int                 groups[NGROUPS];
struct task_struct   *p_opptr, *p_pptr, *p_cptr,
                    *p_ysptr, *p_osptr;
struct wait_queue    *wait_chldexit;
unsigned short       gid,egid,sgid,fsgid;
  unsigned long        timeout, policy, rt_priority;
  unsigned long        it_real_value, it_prof_value, it_virt_value;
  unsigned long        it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list    real_timer;
  long                utime, stime, cutime, cstime, start_time;
unsigned long        min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long        swap_address;
  unsigned long        old_maj_flt;    /* old value of maj_flt */
  unsigned long        dec_flt;        /* page fault count of the last time */
  unsigned long        swap_cnt; *number of pages to swap on next pass */
struct rlimit        rlim[RLIM_NLIMITS];
  unsigned short       used_math;
  char                comm[16];
int                 link_count;
  struct tty_struct    *tty;          /* NULL if no tty */
struct sem_undo      *semundo;
  struct sem_queue     *semsleeping;
struct desc_struct *ldt;
struct thread_struct tss;
struct fs_struct     *fs;
struct files_struct  *files;
struct mm_struct *mm; /* memory management info */
struct signal_struct *sig; /* signal handlers */
};
```

struct task_struct

- **task_struct** obsahuje (nebo odkazuje na) informace, které potřebuje kernel aby mohl vykonávat proces (PID, ukazatel na user stack,…). Nás zajímá **mm_struct *mm**.



mm_struct určuje stav virtuální paměti. Nás zajímá:

- pgd_t ***pgd**;
- struct vm_area_struct ***mmap**;

Obsah PDBR (page directory base register) = PTBR = CR3 (v x86) je nastaven právě hodnotou pgd.

Page Fault Exception Handling - zjednodušeně

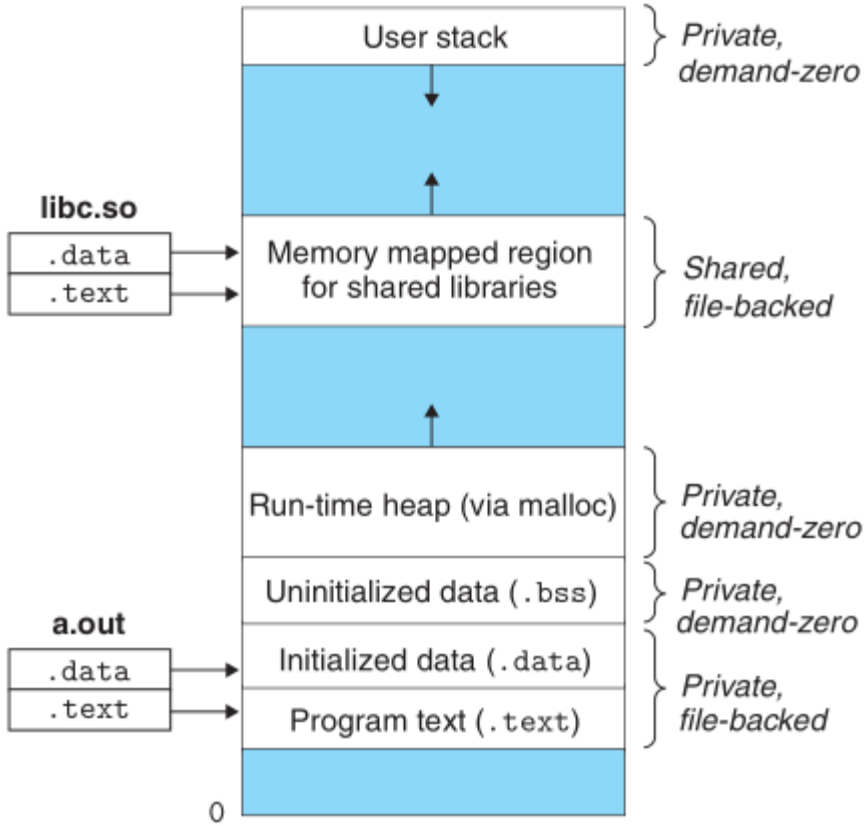
Předpokálejme, že MMU (Mem Manag Unit) spustí **Page Fault** během pokusu o překlad nějaké virtuální adresy A (nenakešováno v TLB). To vede k přenosu řízení na Page Fault Handler, který výkoná:

- Je virtuální adresa A legální? Tzn. leží A unvitř oblasti (area) definovanou některou ze struktur vm_area_struct? Porovnání na meze vm_start a vm_end. Sekvenční prohledávání seznamu je časově náročné => v praxi prohledávání stromu nad tímto seznamem. Pokud adresa není legální -> **Segmentation Fault** a ukončí proces
- Je pokus o přístup legální? Tzn. máme práva? (read, write, execute) Pokud ne -> **Protection Exception** a ukončení procesu
- Nyní může být přístup pouze legální a na legální adresu. Takže musíme vybrat oběť (v případě, že je dirty nakopírovat ji na disk), nahrát novou stránku (tu žádanou), a aktualizovat Page Table. Tím činnost Page Fault Handler končí. CPU restartuje instrukci, která spustila Page Fault. Nyní již MMU žádost o překlad adresy A obslouží normálně – bez generování Page Fault.

Memory Mapping

Linux inicializuje obsah oblasti (area) virtuální paměti asociací s:

- **Regular file** (čtení z dané oblasti paměti se projeví čtením souboru)
- **Anonymous file** – pokud CPU čte z dané oblasti paměti poprvé, kernel najde ve fyzické paměti oběť (pokud je dirty zapíše na disk do *swap file*), vynuluje obsah paměti, aktualizuje Page Table – označí ji jako residentní – bit Present. Oblasti stránek, které jsou mapovány na Anonymous files jsou někdy označovány jako *demand-zero pages*



Dále viz funkci *mmap()*

- která mapuje soubory nebo zařízení do paměti

Některá systémová volání z pohledu paměti

- **fork**() – vytvoří nový proces s novým adresním prostorem. Všechny stránky jsou označeny jako Copy-On-Write (COW) a jsou sdíleny mezi procesy – VMA pro daný region je označena jako writable, ale PTE ne => privátní kopie
- **clone**() – vytvoří nový proces, ale umožní sdílet části jeho kontextu s rodičovským procesem => vlákno
- **mmap**() – vytvoří nový region uvnitř lineárního adresního prostoru daného procesu
- **mremap**() – remapuje nebo mění velikost regionu paměti
- **munmap**() – ruší část nebo celý region. (Pokud je unmapováno někde uprostřed, pak se region rozdělí na dva)
- **shmat**() – připojí shared memory segment k adresnímu prostoru procesu
- **shmdt**() – opak shmat()
- **exit**() – zruší adresní prostor a všechny regiony

References

- Randal E. Bryant, David R. O'Hallaron: Computer Systems, A Programmer's Perspective.
- http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/lecture18.pdf
- David Money Harris and Sarah L. Harris: Digital Design and Computer Architecture, Second Edition. Morgan Kaufmann.