Organizace paměti cache, L/S jednotky

AVS – Architektury výpočetních systémů

Týden 3, 2024/2025

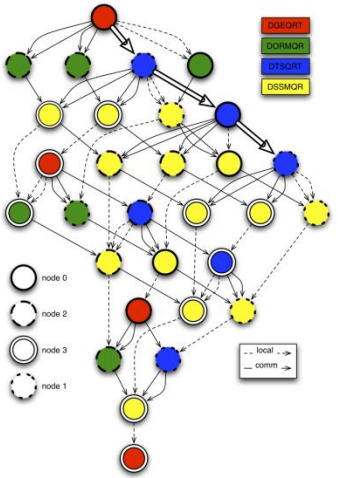
Jirka Jaroš

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole
jarosjir@fit.vutbr.cz

OPAKOVÁNÍ

Dynamické plánování instrukcí

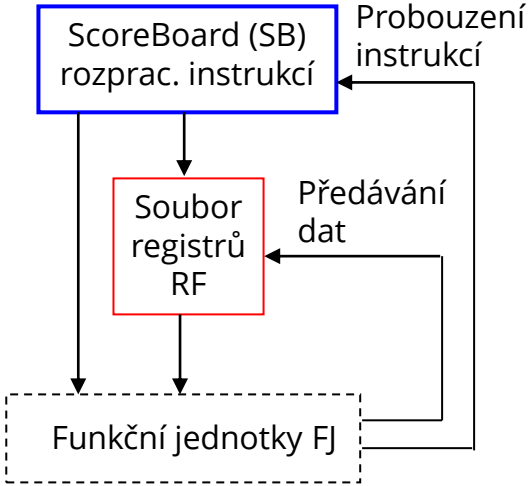
Instrukce jsou vydávány do FJ a prováděny **mimo pořadí** v programu, pokud mezi nimi **nejsou konflikty** a **FJ** jsou **volné**.



1. **ScoreBoarding** (Thorntonův algoritmus, 1964)
   - Registruje všechny **konflikty** (RAW, WAW, WAR) v **tabulce rozpracovaných instrukcí** a udržuje jejich skóre (SB).
   - SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB. **Přejmenování registrů neprobíhá**.
2. **Rezervační stanice** (Tomasulův algoritmus, 1967)
   - Konflikty WAW a WAR se řeší přejmenováním
   - **Rezervační stanice RS** (bufery) umožňují odložit čekající instrukce a pracovat dopředu na dalších – tím řeší RAW.
   - Rezervační stanice centrální (instruction window) nebo individuální u FJ či skupinové pro skupiny FJ.

http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf

Dynamické plánování – ScoreBoarding



Formát jedné položky **ScoreBoard (SB):**

- **stav** instrukce (vydána do FJ, operandy načteny, hotová)
- funkční jednotka **FJ busy**?
- **operace** FJ
- **dst** (**adresa** cílového registru)
- **src1** (**adresa** zdrojového reg. 1)
- bit **V1** (operand 1 platný?)
- **src2** (**adresa** zdrojového reg. 2)
- bit **V2** (operand 2 platný?)

Formát **registrů v RF**(dst):

Rezervační bit **V** | value

0 – neplatný (**rezervovaný**)
1 – platný (někdo, ale ještě může potřebovat)

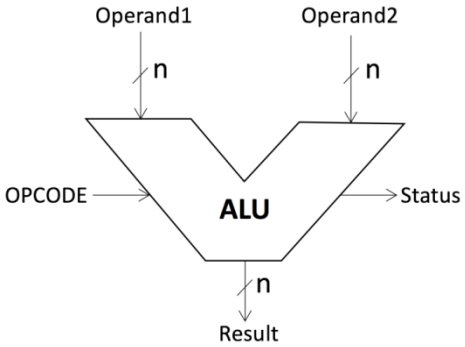
Valid bit V1 a V2
0 – neplatný **nebo** použitý
1 – platný, ale ještě nepoužitý

ScoreBoarding – příklad

Registry

Name	Valid	Value
R1	0	100
R2	0	200
R3	1	300
R4		
R5		
R6		

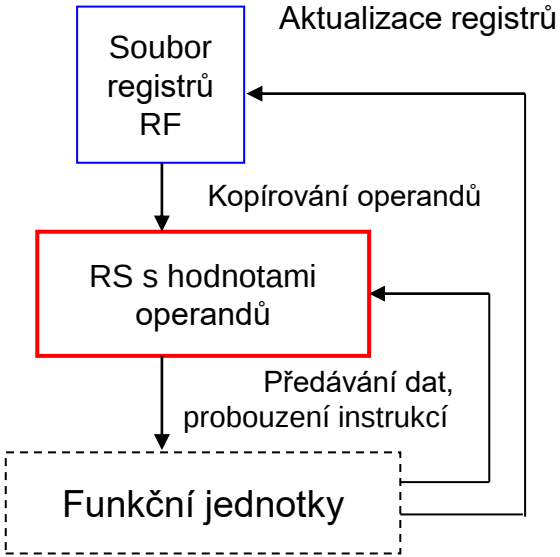
op1 r1, r2, r3
op2 r2, r1, r4
op3 r6, r3, r1
op4 r1, r2, r3
op5 r7, r8, r1
op6 r1, r5, r4



ScoreBoard

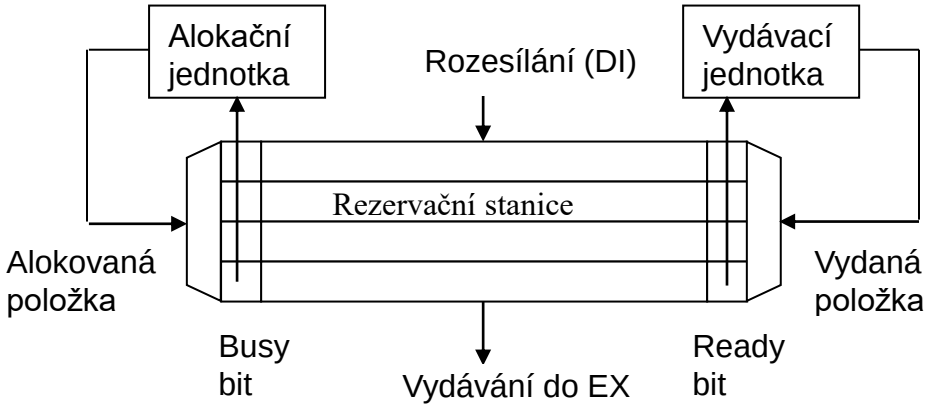
Dst	S1	V1	S2	V2

Rezervační stanice RS, zapojení a formát



Formát instrukcí v RS(i):

- busy bit
- operace
- src1 (hodnota, tag1, valid bit V1)
- src2 (hodnota, tag2, valid bit V2)
- dst (adresa, tag)
- ready bit



Formát **registrů v RF**(dst):

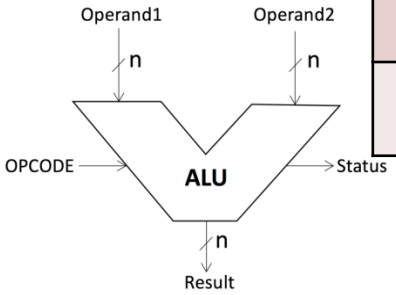
RF(dst) | V | tag | hodnota |

Tomasulo – příklad

Registry

Name	Valid	Tag	Value
R1	0	T1	100
R2	0	T2	200
R3	1	T3	300
R4	1	T4	400
R5			
R6			

```
op1  r1, r2, r3
op2  r2, r1, r4
op3  r6, r3, r1
op4  r1, r2, r3
op5  r7, r8, r1
op6  r1, r5, r4
```



Rezervační stanice

DST	Tag	V	Tag	Val	V	Tag	Val

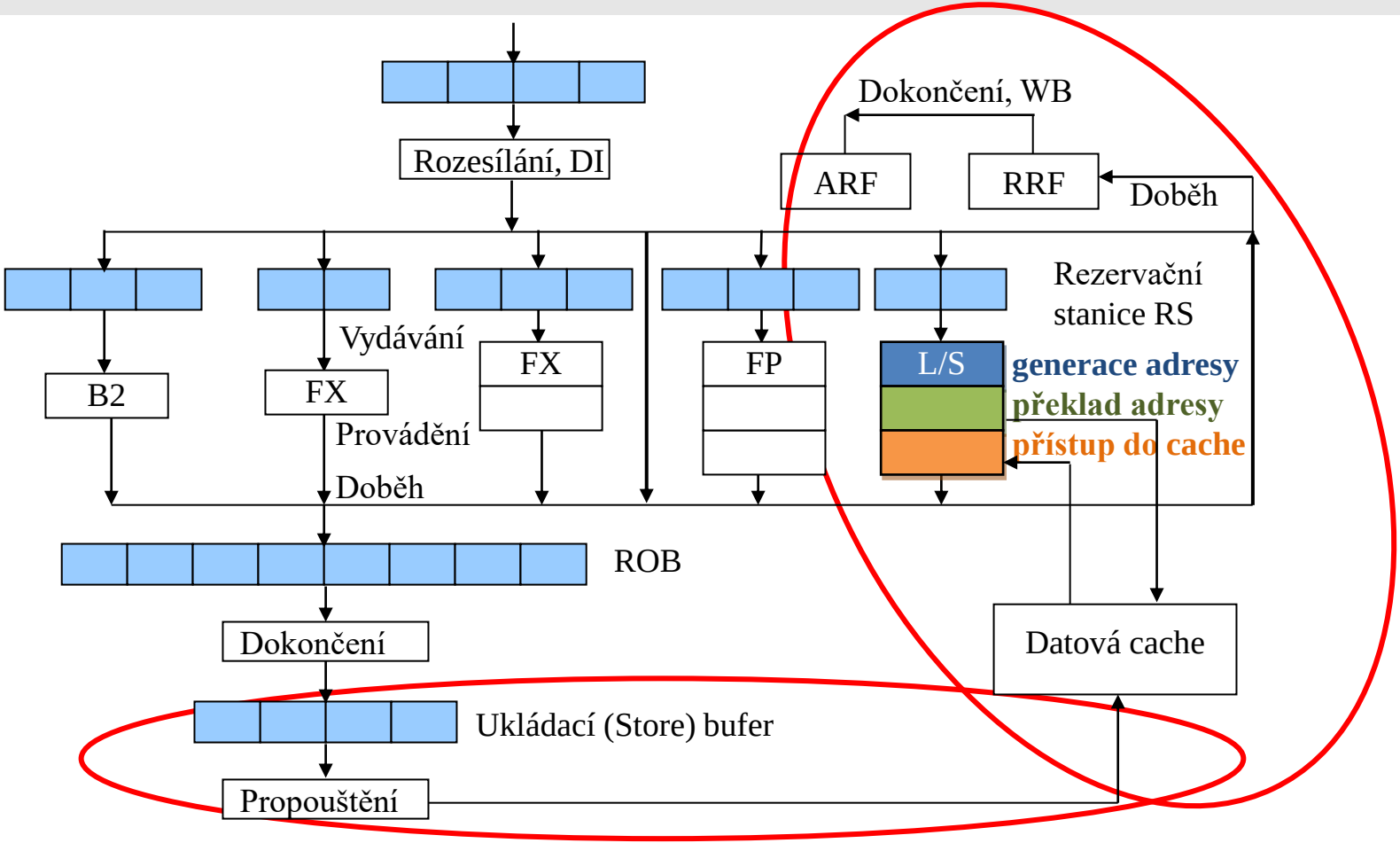
CDB

LOAD/STORE ČÁST PROCESORU

Motivace pro optimalizaci toku dat přes paměť

- V průměru každá 3. instrukce je L/S
  - Několik % z nich generuje výpadek v D-cache
  - Vyřízení výpadku může podle úrovně cache trvat mnoho taktů CPU
- Přistupovat do paměti v programové sekvenci s čekáním na vyřízení výpadků je neúnosné – brání rozbalování smyček.
- Výjimky je třeba zpracovávat podle původního pořadí L/S instrukcí v programu.
- Zisk v IPC překrytím paměťových operací je 35–100%.

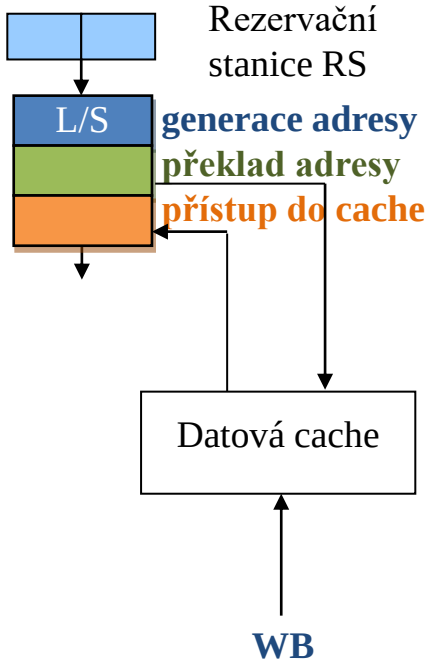
Jednotka L/S – Load/Store



Stupně jednotky L/S

Bez výpadku:

1. generace adresy (sčítačka)
2. překlad adresy (DTLB)
3. čtení z cache (zápis není vidět)



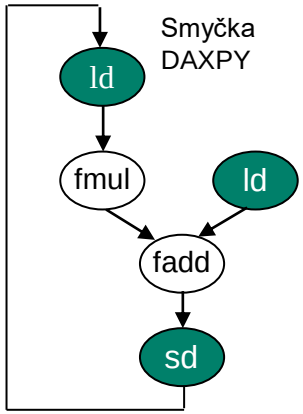
- Překlad adresy v TLB a přístup do fyzicky adresované (P/P) cache probíhají většinou sériově, což znamená součet zpoždění (dva oddělené stupně v lince L/S).

- U cache s virtuálním indexem (V/V a V/P) je možný souběžný přístup do TLB i do cache, který dovoluje překrytí obou zpoždění.

Pořadí přístupů do paměti

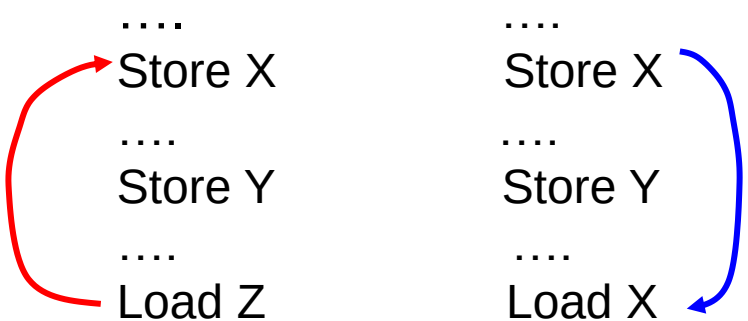
- Mezi instrukcemi **Load a Store se stejnou adresou** existují datové závislosti, podobně jako u registrů.
  - RAW, WAR a WAW.
  - Není je možné odhalit dříve než jsou spočteny adresy paměťových operandů.
  - Tyto závislosti musí být respektovány, aby se zachovala sémantika programu.
- **Instrukce Load a Store je možné**
  - Vykonávat v programovém pořadí (pomalé)
  - Instrukce Load a Store lze za jistých okolností provádět OOO.
    - **RPR** Read can Pass Read
    - **RPW** Read can Pass Write
    - **WPR** Write can Pass Read
    - **WPW** Write can Pass Write

Čtení předbíhá zápis (Read can Pass Write, RPW)



- OOO procesor v podstatě provádí HW rozbalení smyčky.
- Nový load může předběhnout aktuální store, iterace smyčky se mohou částečně překrýt!
- RPW je hlavním zdrojem lepší výkonnosti, načítání bývá totiž na začátku těla smyček se závislými instrukcemi
2 způsoby:

Load z adresy ***Z*** předběhne Store s jinou adresou ***X***, které ještě nezačalo (*bypassing*)



Load načte data z ještě nedokončené instrukce Store se stejnou adresou ***X*** (load from store *forwarding*, předávání).

Technika RPW vyžaduje **dynamické rozlišování adres**:

adresa (R) ≠ adresa (W) ?

Ano: neshoda adres, (není konflikt RAW);
R (load) nečeká na W (store) a předběhne jej
načte data z D-cache do dst reg (samozřejmě v RRF nebo ROB).

Ne: shoda adres, R (load) načte data do dst reg (v RRF nebo ROB).
z čekajícího nedokončeného zápisu (store buferu)

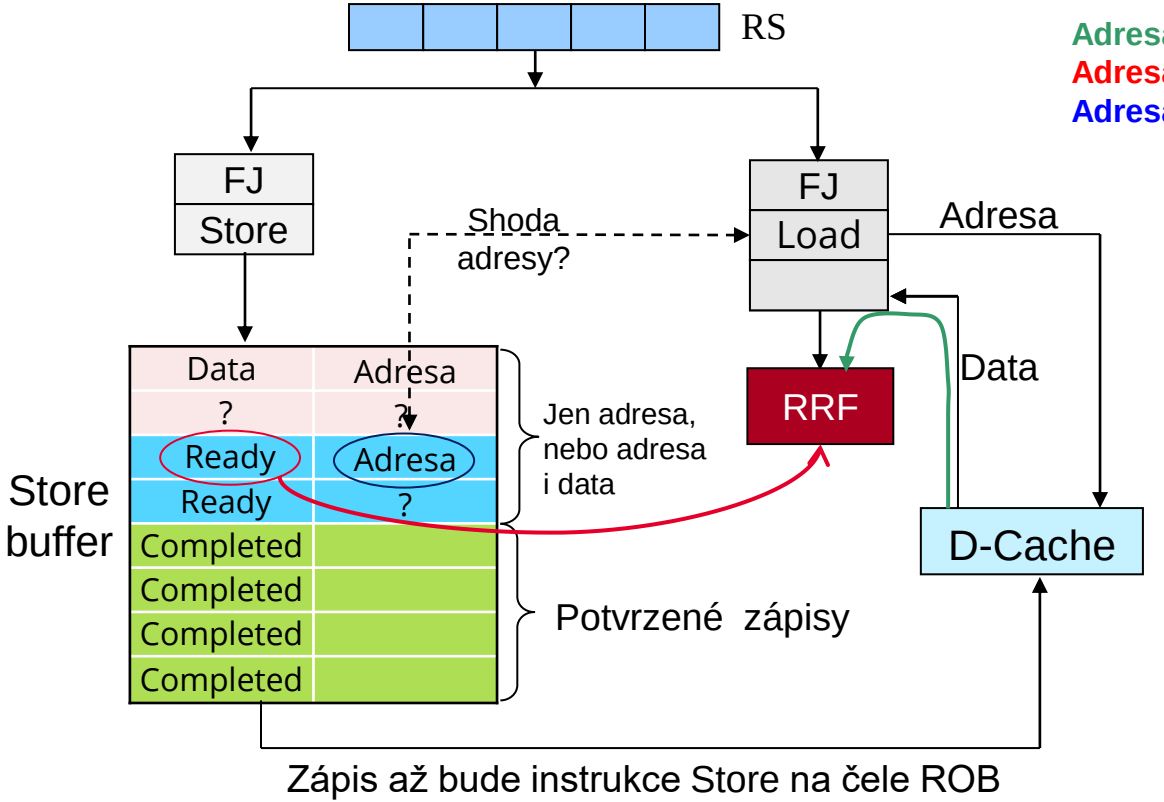
Neví: čekej (nebo spekuluj, že se adresy liší).

RPW může zvýšit výkonnost o 11–19% (bypassing) a o 1–4% (forwarding).

Store buffer – rozpracované zápisy před fází WB

- Položka je ve store bufferu alokována v době dekódování (DI). Pokud je store buffer plný, musí se čekat.
- Adresy instrukcí Store jsou ve bufferu v programovém pořadí.
- Nové adresy Load se kontrolují s čekajícími adresami Store na shodu / neshodu.
- **Stavové bity** v každé položce indikují:
  - Available – položka je volná, k dispozici
  - Adr only – adresa je již v bufferu, data ještě ne
  - Ready – adresa i data jsou již v bufferu
  - Completed – potvrzené zápisy, čekají až instrukce Store bude na čele ROB
- Když je instrukce Store na čele ROB i store buferu, dojde k propuštění Store, tj. k zápisu do D-cache. Položka přejde do stavu Available.
- V terminologii x86 nazýván Store Queue

Organizace jednotky L/S podporující RPW



Adresa (R) ≠ Adresa (W)
Adresa (R) = Adresa (W)
Adresa (R) ? Adresa (W)

Bypassing
Forwarding

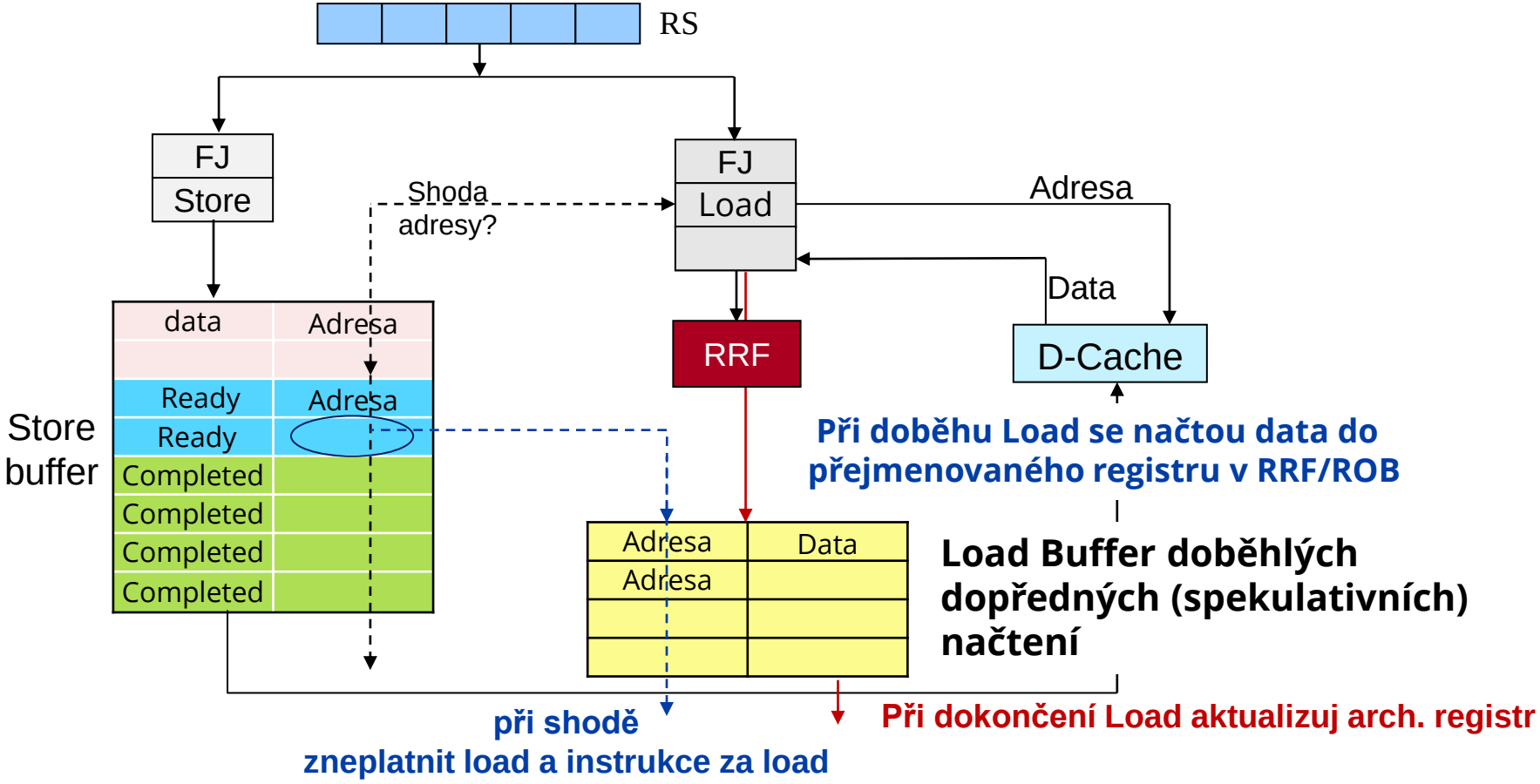
Kvůli porovnávání load a store adres je kruhový store buffer typicky organizován jako plně asociativní fronta (komparátor u každé položky).

Spekulativní načítání – Load buffer

Pokud některé adresy Store nejsou známy, je třeba spekulovat, že se budou lišit od adresy Load.

- **Pro ověření spekulace:**
  - jsou adresy doběhlých spekulativních načítání uloženy do nového L-bufferu.
  - Každá potvrzená instrukce Store pak musí ověřit, že nemá adresu shodnou s nějakou položkou L-bufferu, tj. s nějakým spekulativním čtením, které Store předběhlo.
    - Při shodě je potřeba postiženou instrukci Load a další za ní následující instrukce zrušit a opakovat.
    - Pokud k žádné shodě nedošlo až do dokončení Store, je proveden přepis Load dat z RRF do ARF.
- V terminologii x86 nazýván Load Queue

Out-of-order Load/Store jednotka



Relaxovaná paměťová konzistence

- **Sekvenční konzistence,** která zachovává pořadí přístupů do paměti, není v současnosti zajímavá. Je totiž překážkou modernímu hardware a optimalizujícím kompilátorům.
- Předbíhání RPW je jen jedna možnost uvolnění pořadí přístupů do paměti, i když pro výkonnost nejvýznamnější.
- Existuje však řada ještě volnějších modelů se zcela volným pořadím čtení a zápisů.
- Moderní procesory vykazují **relaxovanou** (uvolněnou) **paměťovou konzistenci**, kdy se čtení a zápisy mohou vzájemně předbíhat, pokud to nemění správnost programu.

Dopady relaxované paměťové konzistence

- **Relaxovaná paměťová konzistence**
  - Lepší výkonnost a jednodušší HW implementace
  - Je ponecháno na programátorovi, aby identifikoval a označil spec. instrukcemi (např. paměťovými bariérami) ty instrukce L/S, které musí být uspořádány.
  - Všechny ostatní instrukce se mohou provádět mimo pořadí.
- Na vyšší úrovni **musí programátor použít synchronizační příkazy** pro vymezení oblastí předepsaného pořadí L/S.
  - Direktivou flush v OpenMP
  - proměnnými volatile aj.
- **Nevýhodou relaxovaných modelů je břímě navíc pro programátora, možnost záludných chyb.**

```
data = gendata(…)
ready = 1;
```

```
while (!ready) ;
usedata(data)
```

Paměťové zábrany (memory barriers, fences)

- Speciální instrukce **paměťových zábran fence** F (ohrádka, zábrana) zabraňují přeskládání L/S tam, kde je to nežádoucí.
- Všechny instrukce L/S v programu před fence musí dokončit a teprve po fence mohou začít další, takže v sekvenci WFR nemůže nastat RPW.
  - Plná zábrana (viz výše)
  - Částečné zábrany (se týkají jen čtení nebo jen zápisu)
  - Jednosměrné zábrany: dvojice instrukcí acquire (brání přesunům L/S nahoru) a release (brání přesunům L/S dolů).
- Deklaraci proměnné „**volatile**“ interpretuje kompilátor jako **zápis s release a čtení s acquire.**

```
data = gendata(…)
ready = 1;
```

```
while (!ready) ;
usedata(data)
```

Možné optimalizace toku dat přes paměť

- Sekvenci po sobě jdoucích zápisů na čele fronty do téhož bloku D-cache lze provést současně.
- Důležitější je optimalizovat čtení než zápisy, protože mohou bránit postupu výpočtu. Čtení se vyskytuje 2x častěji než zápis a četnost výpadků je zhruba stejná.
- Při výpadku čtení v běžné (blokující) cache L1 se **zastaví linka L** a další instrukce se nevydávají až do vyřízení výpadku.
- Jednotka L/S dovolující jen jedno čtení nebo zápis v jednom taktu je značně omezující u shluků L instrukcí. Proto se používá více jednotek L/S, neblokující cache, dvoubránová nebo levnější prokládaná paměť cache.
- **Výhody:**
  - Další přístupy po výpadku čtení, které nepotřebují data z výpadku, nejsou blokovány
  - Dovolují zpracování několika souběžných výpadků a zásahů.
  - Mnoho L1C a většina L2C jsou neblokující D-cache.
  - Výpadky při zápisu jsou ošetřeny zápisovými bufery.

PAMĚTI CACHE
(RYCHLÉ VYROVNÁVACÍ PAMĚTI)

Paměti cache (Rychlé Vyrovnávací Paměti)

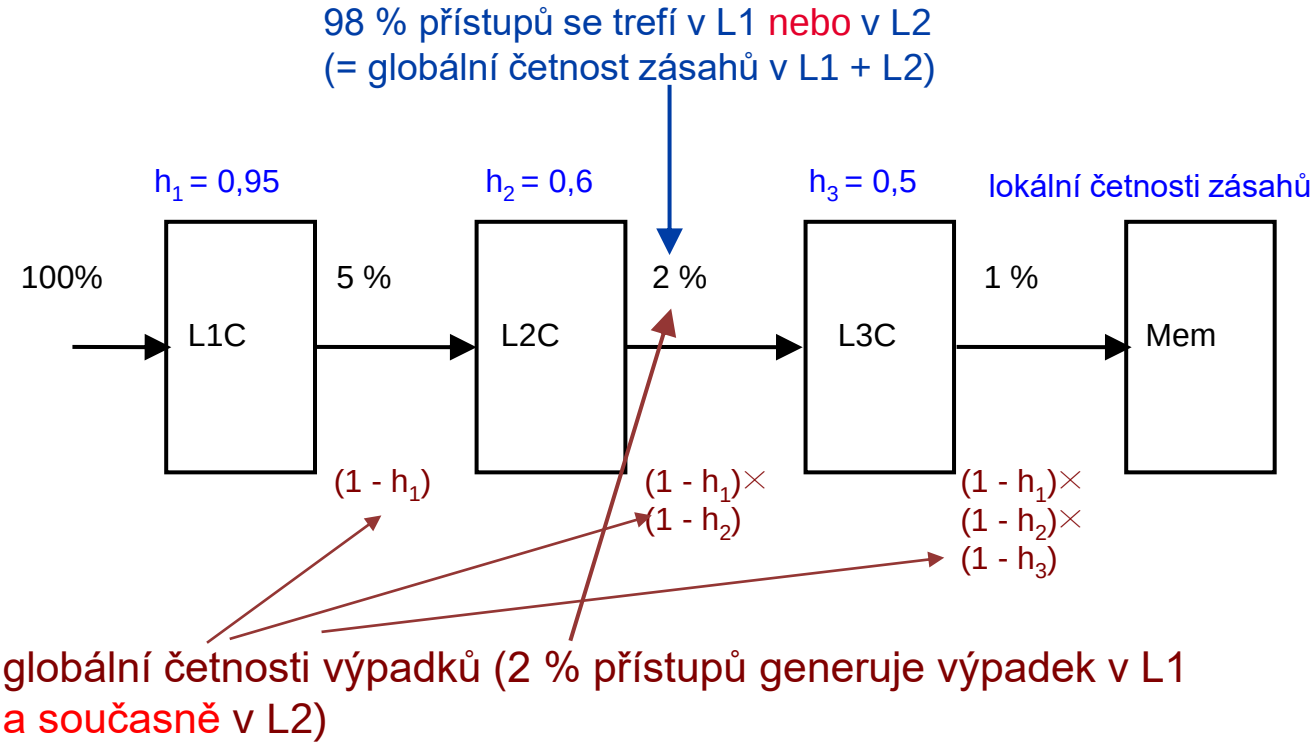
- Doposud jsme uvažovali:
  - jen cache L1,
  - reálnou paměť,
  - fyzickou adresu PA,
  - četnost zásahů (hit rate) h = 100 %.
- Nyní budeme uvažovat:
  - L1C, L2C, L3C = cache úrovně L1, L2, L3,
  - virtuální paměť, virtuální adresa VA, $h_{1,2,3} < 100$ %.

- Názvosloví:
  - Sjednocená/rozdělená cache pro data a instrukce
  - Cache privátní/sdílené
  - Strategie zápisu:
    - write through nebo write back (při zásahu)
    - write around nebo write allocate (při výpadku)

Hierarchie pamětí cache

- **2 úlohy pamětí cache:**
  - Snížení objemu komunikace s hlavní pamětí on 2 až 3 řády.
  - Snížení průměrné doby přístupu o 1 až 2 řády.
- Jelikož malé paměti jsou rychlejší a dražší, velké paměti pomalejší a levnější, je optimální paměťový systém heterogenní, s několika úrovněmi pamětí cache
- **Parametry pamětí cache**:
  - (lokální) četnost zásahů h_i: počet zásahů v cache úrovně *i* dělený počtem přístupů do **této** cache
  - (globální) četnost výpadků v úrovni *i*: $(1 - h_1) \times (1 - h_2) \times \ldots (1 - h_i)$ znamená četnost výpadků generovaných **současně** v úrovni 1, 2… *i*
  - latence zásahu t_i: doba od vyslání adresy do návratu dat z cache *i.*
  - latence výpadku (pokud procesor při výpadku blokuje) na úrovni *i*: doba přenosu bloku z cache úrovně *i* + 1 do úrovně *i* a načtení slova z cache *i*.
- **U procesorů OOO nelze latenci výpadku přímo měřit.**

Lokální a globální četnost výpadků a zásahů



Uvedené parametry se týkají konkrétního programu nebo skupiny programů.

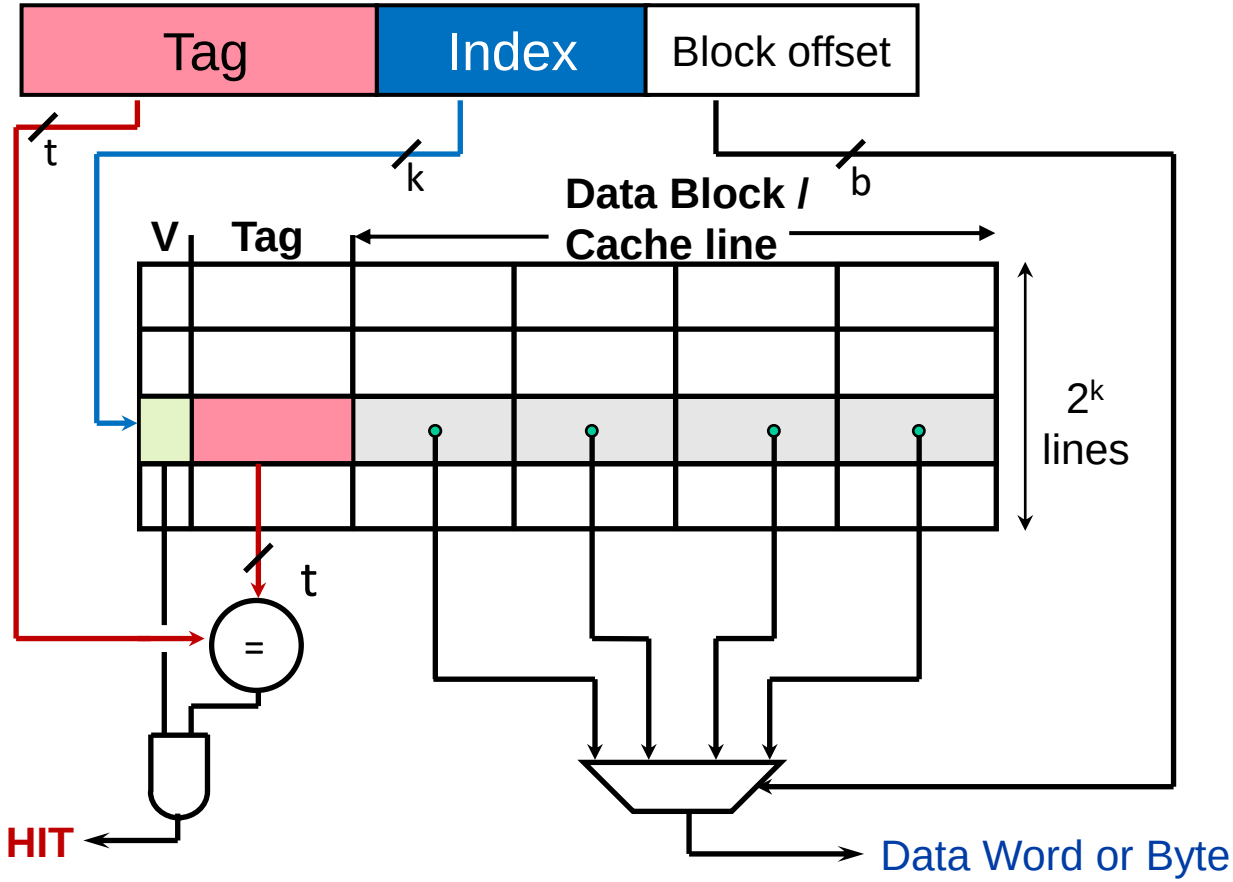
Organizace pamětí cache

- Výběr dat po blocích na základě
  - obsahu (příznaku bloku, tag)
  - nebo i adresy (index).
- Asociativita ***m* = počet cest (ways), komparátorů** pro tag.
  Přístup do bloku *i*:
  - shodný tag existuje: hit,
  - neexistuje: miss.
- **Zjednodušení HW:**
  - počet bloků *C* celé paměti
  - lze rozdělit do *S* skupin (sets)
  - a pro výběr skupiny použít adresu (index).

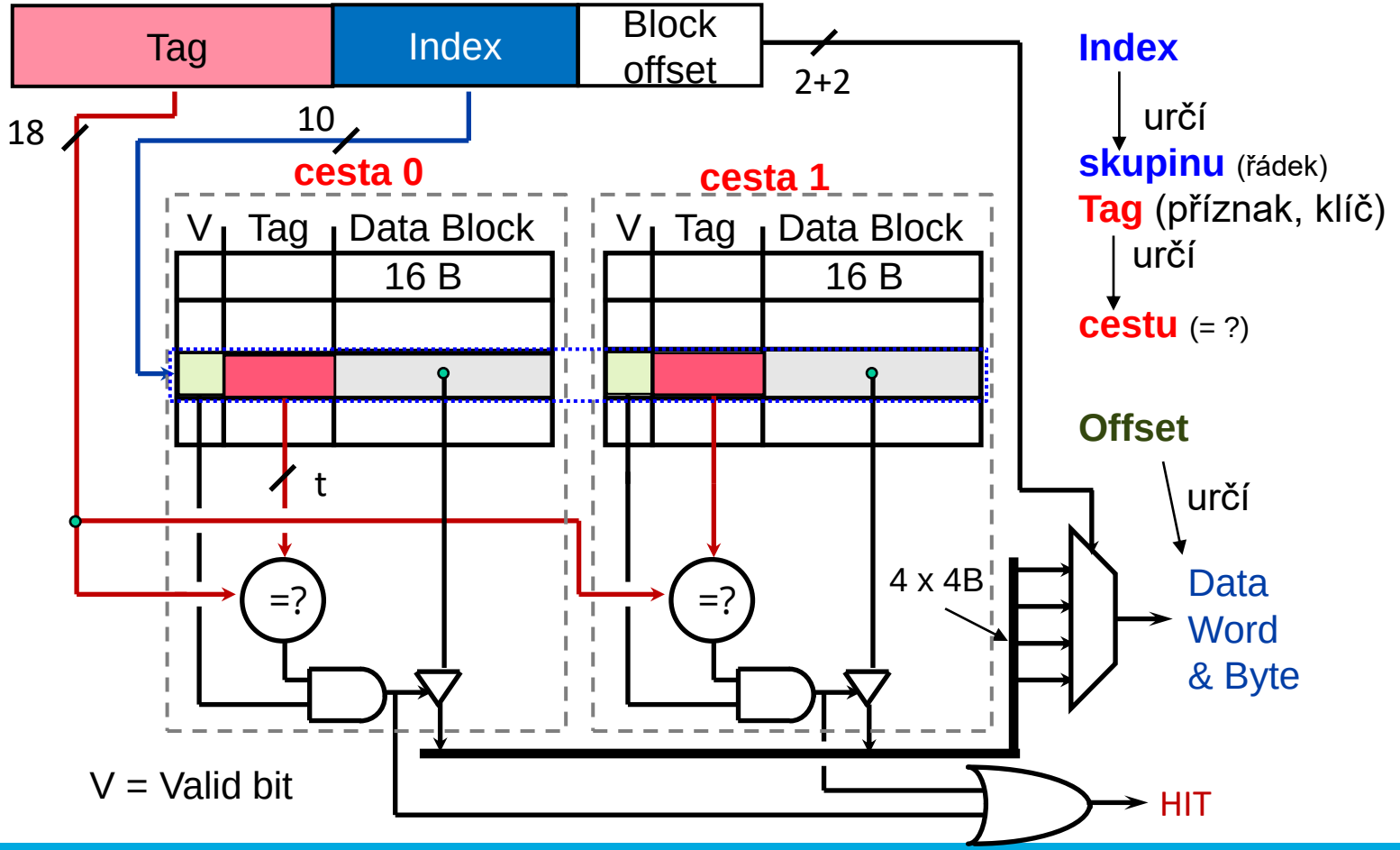
- Podle toho máme paměti cache
  - Přímo mapované (PM): $m = 1$, $S = C$ (skupina = 1 blok)
    - `index = i mod S` vybírá 1 blok cache
    - shoda tagů se hledá jen v nalezeném bloku.
  - Plně asociativní (PA): $C = m$, $S = 1$ skupina, jen tag
    - hledá se shodný tag v celé paměti (tj. skupině).
  - Skupinově asociativní (SA): index + tag
    - index určuje skupinu
    - `index = i mod S` (má $\log_2 S$ bitů)
    - shodný tag se hledá v indexované skupině.

Přímo mapovaná (PM) cache

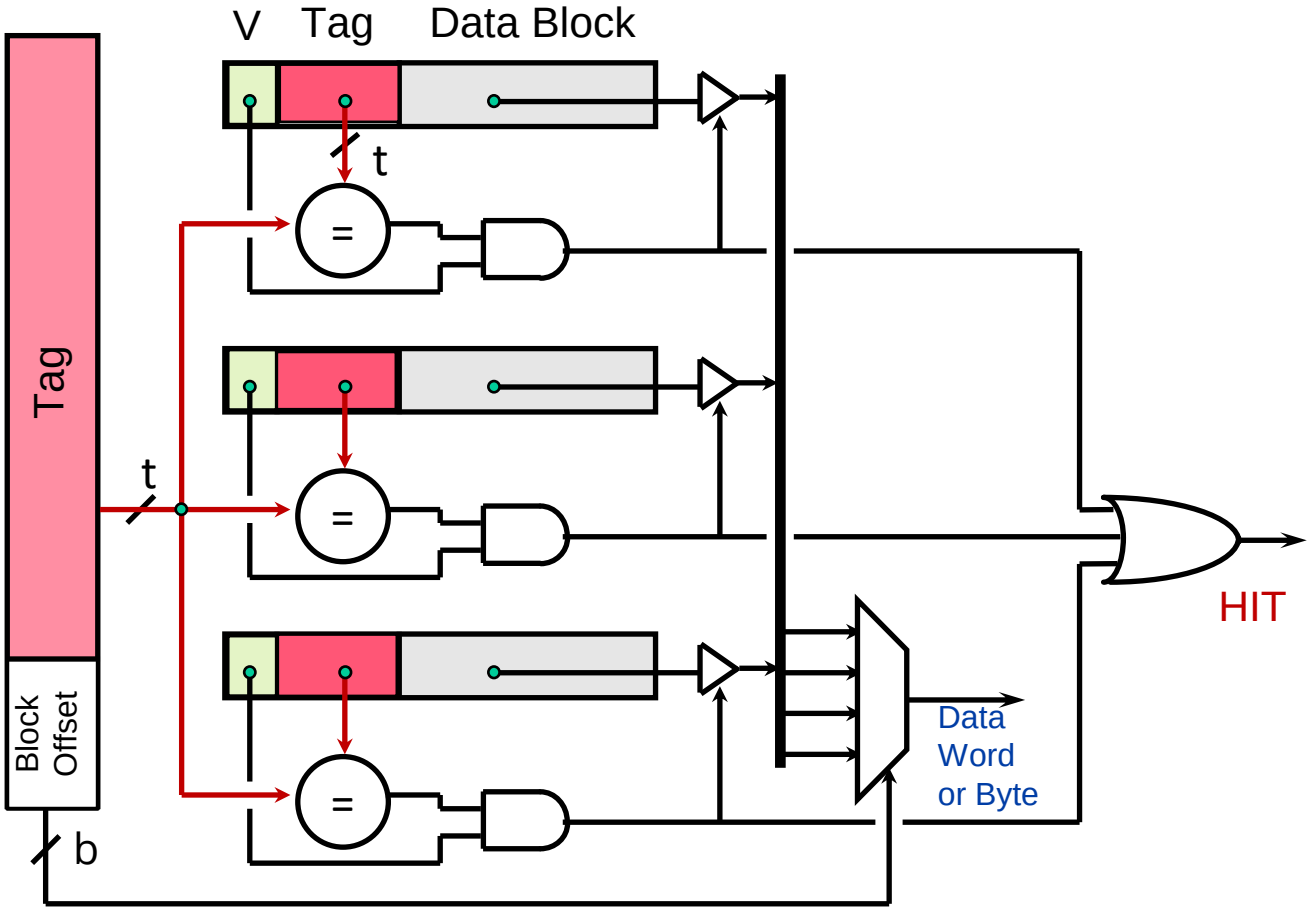
Adresa:



Skupinově asociativní (SA) cache 32 kB

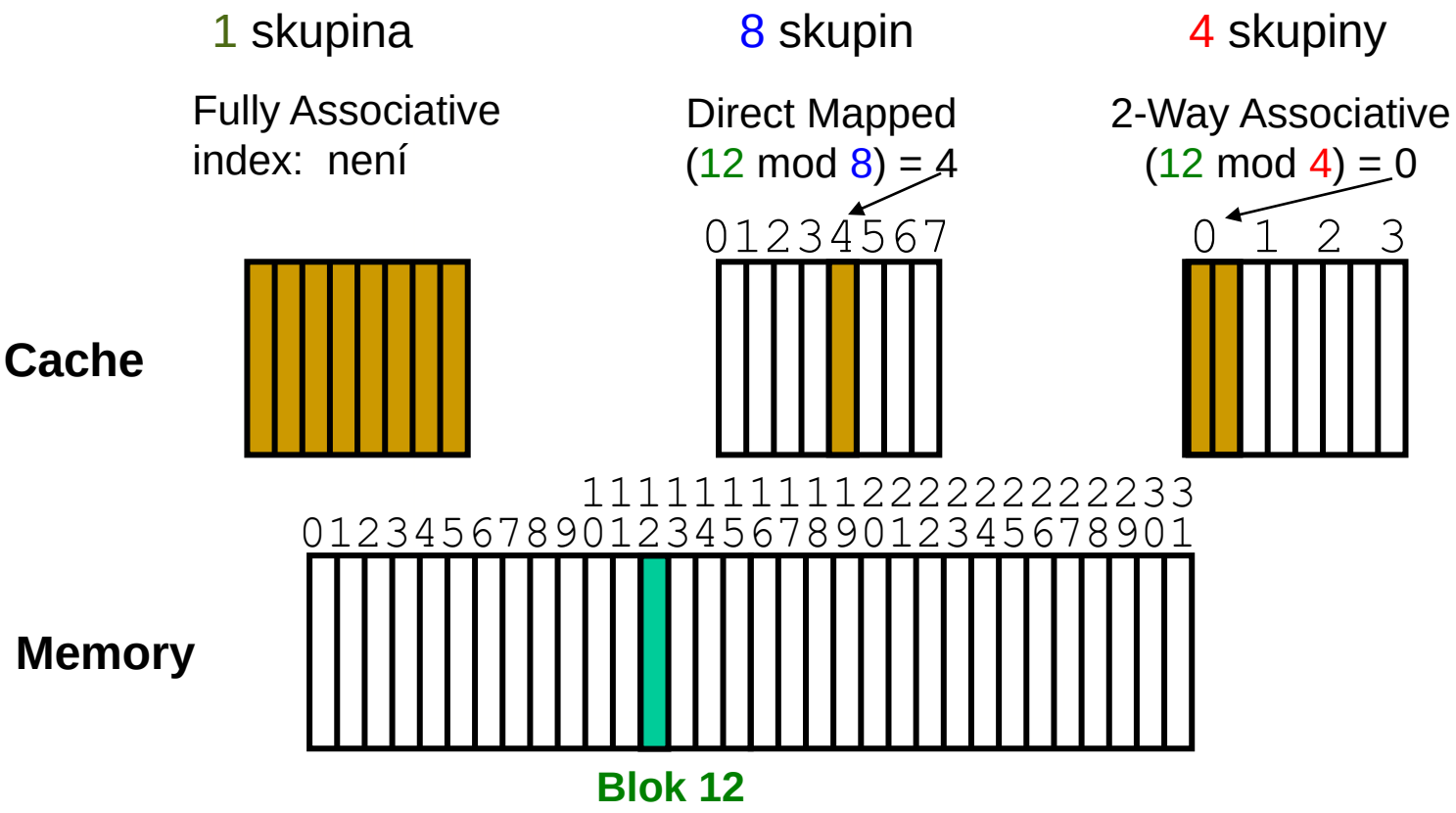


Plně asociativní (PA) cache



Příklad mapování bloků paměti do bloků cache

Umístění bloku i = 12 do cache o velikosti 8 bloků (kam mohu vložit):



Výpadky v paměti cache

- **Výpadek při zápisu slova do cache** (write allocate):
  - Najde se volný blok a provede se zápis.
  - Není-li v cache volné místo, musí se některý blok z cache přemístit do paměti (block replacement).
- **Strategie výběru bloku pro přemístění:**
  - LRU (blok nejdéle nepoužitý, zaznamenává se počet přístupů),
  - FIFO (nejstarší blok),
  - nebo náhodný blok.
- **Pokud je vyměněný blok později potřeba**
  - nastane výpadek kvůli kolizi (jen jedna cesta u PM cache nebo málo cest u SA cache)
  - kapacitě (PA cache).
- **Výpadky vznikají na začátku**, když je cache prázdná (povinné výpadky)
- **Výpadky udržováním koherence** v multiprocesorovém systému (koherenční).

Inkluzivní a exkluzivní paměti cache

- U vícejádrových procesorů jsou **cache kromě poslední úrovně privátní** a **LLC** (Last Level Cache) **sdílená**.
- **Inkluzivní paměti cache (Intel do Skylake-X)**
  - **Vlastnost inkluze** mezi úrovněmi cache (obsah L2C) je nadmnožina obsahu L1C) zajišťuje, že L2C odfiltruje zbytečné požadavky na L1C:
    - invalidace bloků, které nejsou v L2C, nepostupují do L1C
    - je-li blok v L2C, **inclusion bity** u něj ukazují, zda je i v některé L1C.
      - 1 : je v dané L1C, musí se zneplatnit,
      - 0 : žádná akce.
  - **Udržování inkluze**:
    - Při výpadku v L1C musí být blok načten do všech úrovní L_i
    - Když je blok z úrovně L_i přemístěn do paměti, musí být odstraněn ze všech úrovní pod L_i

Víceúrovňová hierarchie pamětí cache

- Exkluzivní paměti cache (AMD):
  - Je-li blok v úrovni L_i, pak není v žádné jiné úrovni
  - Při výpadku v L1C se tam **přesune** (nekopíruje) blok z některé vyšší úrovně
  - Když je blok z úrovně L_i **přepsán** (zničen) blokem z paměti, je před tím přesunut na úroveň L_i +1
  - V systému L1C-L2C je pak L2C velká cache „obětí“.
- **Cache obětí (Victim Cache, VC)**
  - Přidá se malý buffer pro umístění (LRU) bloků vyhozených z L1C
  - Tato PA **cache obětí** již jen se 4–16 položkami může efektivně eliminovat mnoho výpadků PM cache.
  - Používá se mezi úrovněmi L1 a L2. Dojde-li v L1C k výpadku, tak
    - při zásahu ve VC se nalezený blok vymění s LRU blokem v PM cache
    - při výpadku ve VC se v L1C udělá místo pro nový blok. Oběť (LRU blok v L1C) se zapíše do VC. Je-li VC plná, pak do další úrovně. Nový blok je pak zaslán do L1C.

PODPORA VIRTUÁLNÍ PAMĚTI NA PROCESORU

Názvosloví

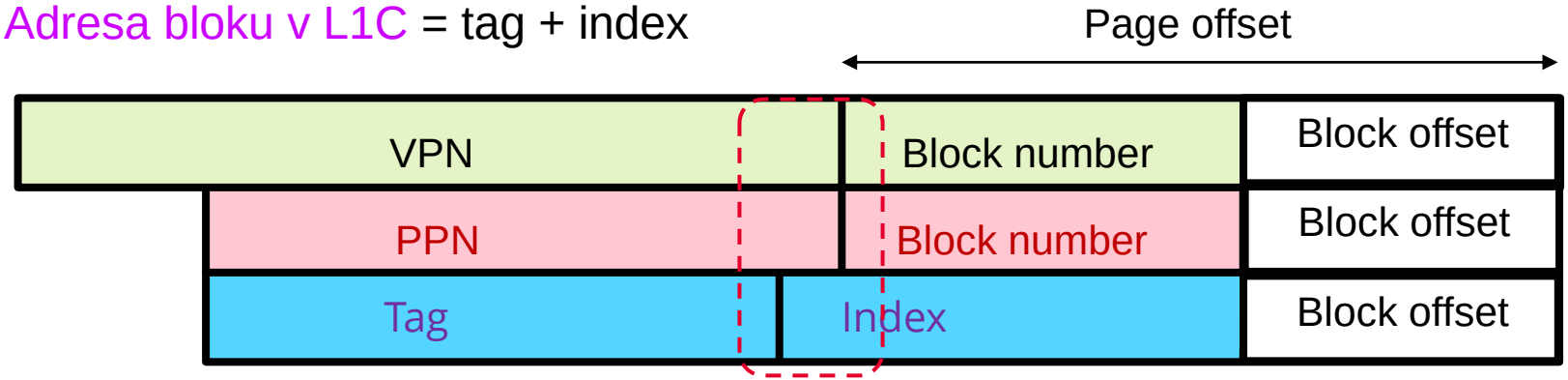
VA = Virtual Address = virtual page number **VPN** + **Pg. offset**

PA = Physical Address = phys. page number **PPN** + **Pg. offset**

Pg.offset = Block number (číslo bloku na stránce) + **block offset** (které slovo, byte)

Adresa bloku v paměti = PPN + Block number

Adresa bloku v L1C = tag + index



Velikost stránky: 4, 8 kB, ale i 64 kB, 2 MB, 4 MB

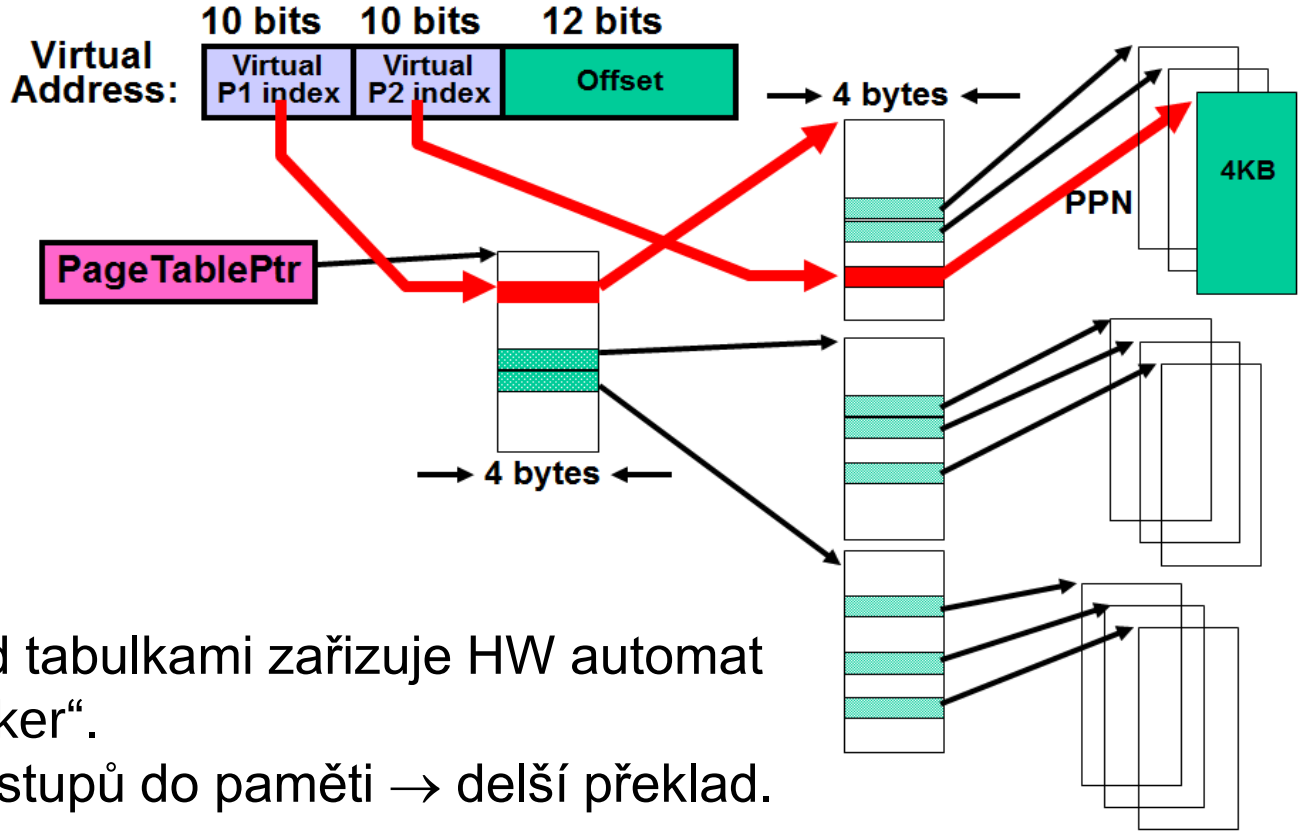
Cache index: 16 kB, 32 kB, 512 kB, 16 MB

Velikost lineární tabulky stránek (Page Table)

- Položka PT (PTE, Page Table Entry) mapuje číslo virtuální stránky na číslo fyzické stránky, VPN → PPN; Vyhledání je snadné!
- **PT je v paměti.**
- Pro 32 bitové adresy, stránky 4 KB a PTEs 4 byte:
  - $2^{32}/2^{12} = 2^{20}$ PTEs, tj. **tabulka stránek 4 MB** na 1 proces
  - až 2^{32} = 4 GB dat v celém virtuálním prostoru na 1 proces
- **Větší stránky?**
  - Vnitřní fragmentace (celá stránka se neužije) ☹
  - Větší pokuta při výpadku stránky (delší čas čtení z disku) ☹
  - Méně překladu při zpracování velkých dat (matice) ☺
- **A co teprve 64 bit virtuální adresový prostor???**
  - Dokonce při velikosti stránek 1MB bychom potřebovali $2^{64}/2^{20} = 2^{44}$ PTEs 8 byte (2^7 TB!)

- **Naštěstí je obsazení virtuálního adresového prostoru řídké**

Víceúrovňová tabulka stránek



Průchod tabulkami zařizuje HW automat „PT walker“.
Více přístupů do paměti → delší překlad.

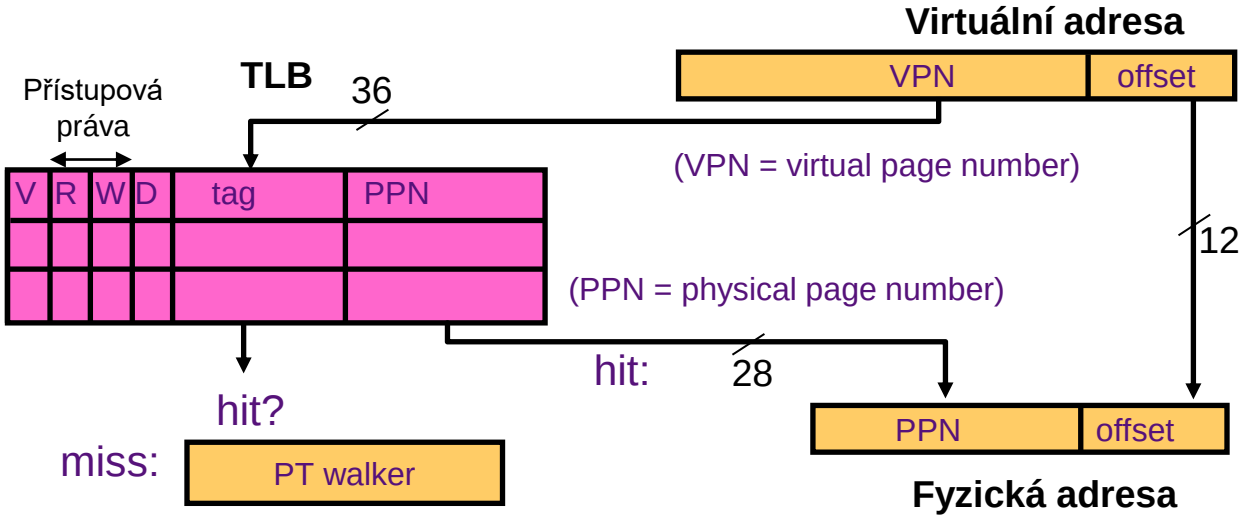
Translation Lookaside Buffer (TLB)

Překlad adresy je velmi drahý!

V hierarchické tabulce stránek stojí každý překlad několik přístupů do paměti.

Řešení: ***Překladová tabulka TLB na čipu*** obsahuje sadu aktuálně používaných dvojic {VPN, PPN}

- TLB hit ⇒ *překlad za 1 takt,*
- TLB miss ⇒ *průchod PT k doplnění TLB*



Překládá se jen číslo stránky, offset stránky zůstává stejný!

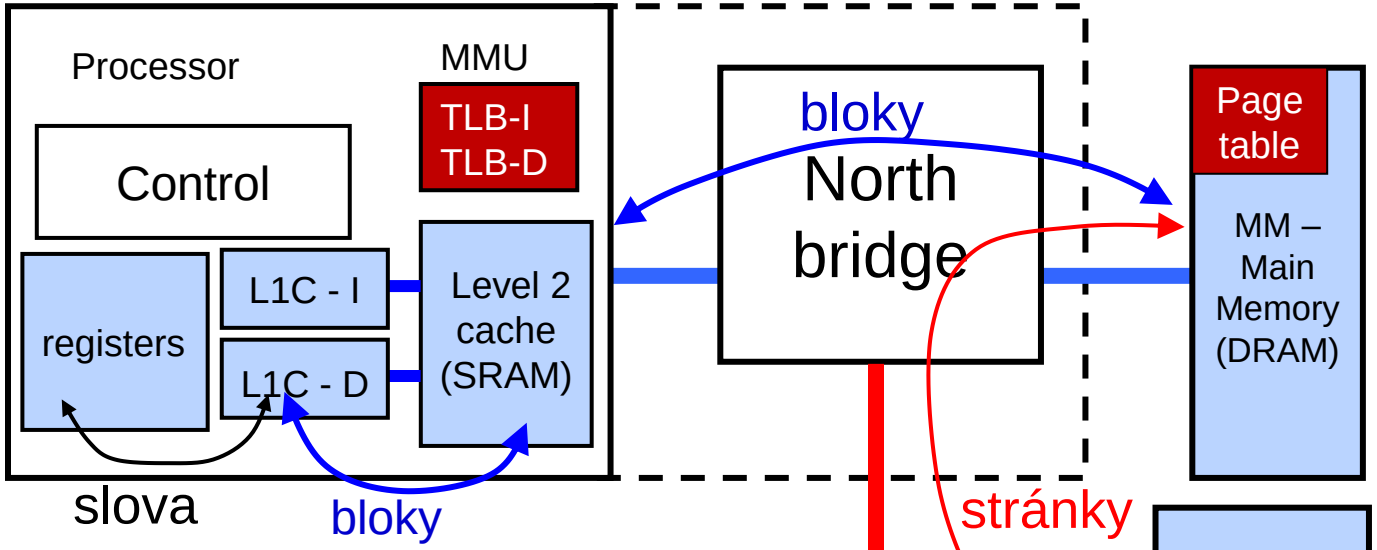
Překladová tabulka TLB

- **Velmi malá** (32–128 položek) a tak velmi rychlá cache.
- **Plně asociativní.** Někdy jsou větší TLBs (256–512 položek) skupinově asociativní, 4–8 cestné
- Může být rozdělena na instrukční a datovou část.
- **Strategie výměny**: LRU, pseudo LRU, random nebo FIFO
- **Správa TLB**: HW nebo SW.
- Větší systémy někdy mívají více-úrovňové TLB (L1 a L2).
  TLB L2 bývá sjednocená.

Intel Haswell:

- **I-TLB**
  - 4 kB stránky
    - 128 položek, 4-way asociativita, dynamicky rozdělena mezi 2 vlákna.
  - 2/4 MB stránky
    - 8 položek, plně asociativní. Každé vlákno má svoji tabulku.
- **D-TLB**
  - 4 kB stránky
    - 64 položek, 4-way asociativita, fixně rozdělena mezi vlákna.
  - 2/4 MB stránky
    - 32 položek, 4-way asociativita.
  - 1 GB stránky
    - 4 položky, 4-way asociativita.
- **STLB**
  - 4 kB + 2 MB stránky
    - 1024 položek, 8-way asociativita.
    - Sdílená pro data i instrukce na úrovní L2.

Paměti cache v systému virtuální paměti



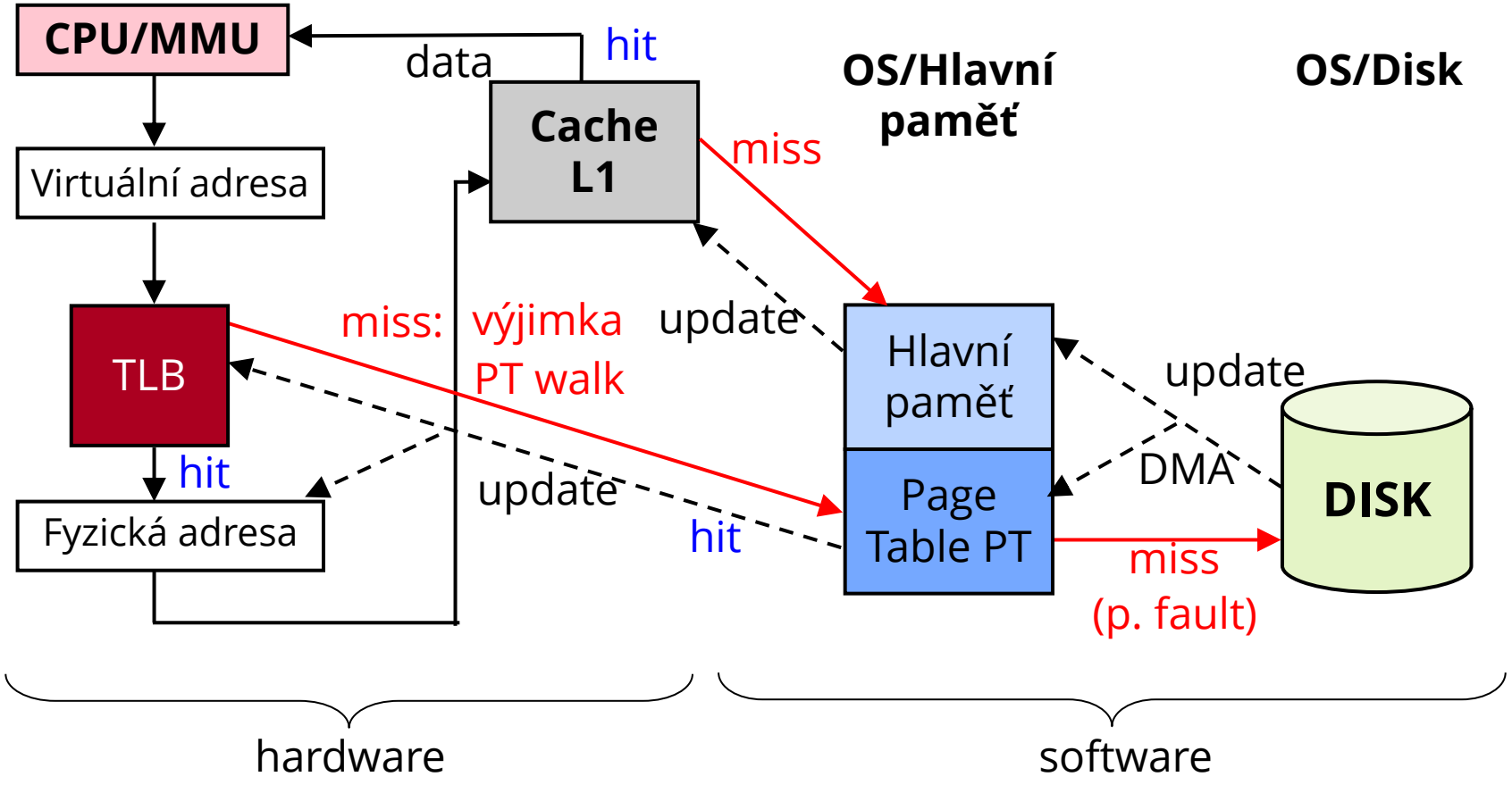
zásah v TLB i v L1C — 1–3 takty

výpadek v L1C, zásah v L2C — 8–12

výpadek v L2C, zásah v MM — 75–250

výpadek v TLB, zásah v PT — 2000 → výpadek v PT — 10^8 taktů

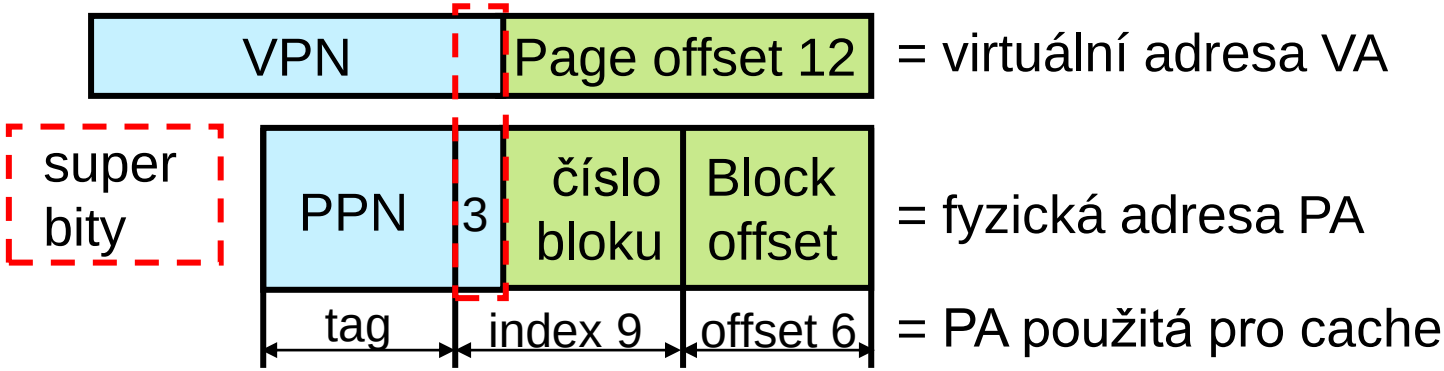
Schéma přístupu do paměti s L1C



Virtuální paměť a cache L1C

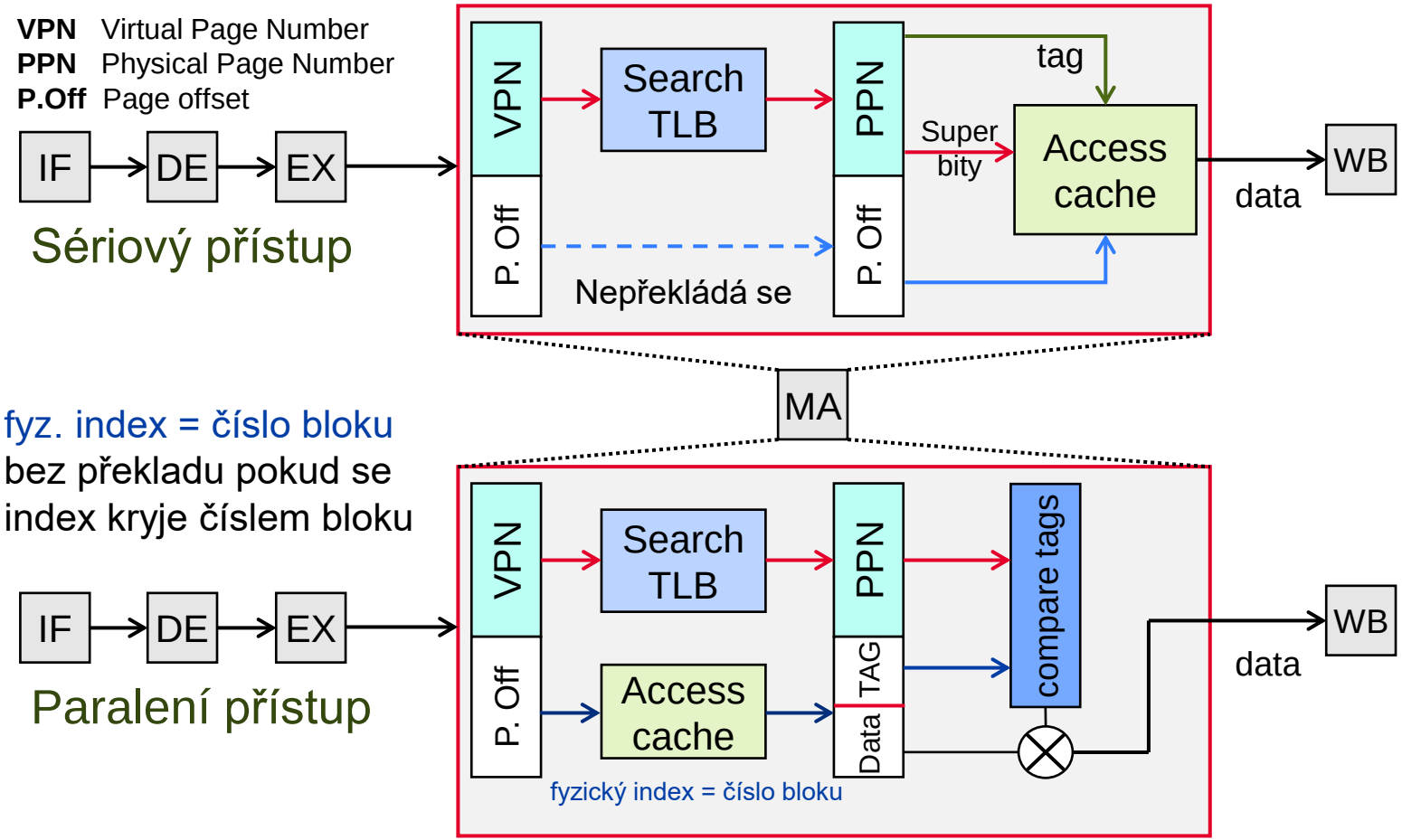
- Spolupráce CPU s L1C vyžaduje co nejrychlejší přístup do cache.
- Jak dospět od VA k adrese bloku (**index, tag**) v L1C?
  - VA = virtuální adresa je k dispozici ihned
  - PA = fyzická adresa je k dispozici až po překladu VA.
  - Která adresa by se mohla použít pro přístup?
- **Tři možnosti:**
  - P/P cache: fyzický index, fyzický tag
  - V/V cache: virtuální index, virtuální tag
  - V/P cache: virtuální index, fyzický tag
    virtuální index se dá použít hned, paralelně s překladem VA.
  - P/V cache: fyzický index, virtuální tag - nepoužívá se, index by se musel získat překladem a čas by se neuspořil.

Fyzicky adresované P/P cache

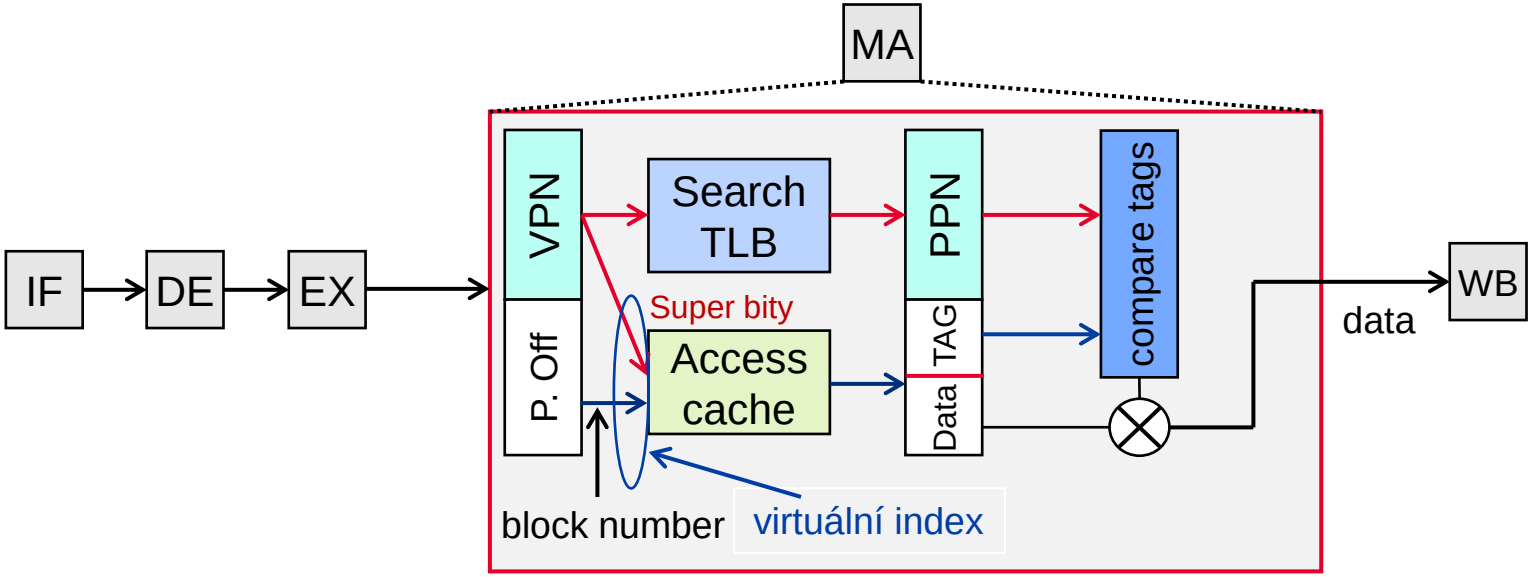


- **Sériový přístup** napřed TLB, pak cache L1 je pomalý.
- **Paralelní přístup** je možný

1. Když se **index** kryje s **číslem bloku**: nemusí se překládat, indexuje bloky na 1 stránce dané PPN.
   Např. pro stránku 4 kB, velikost bloku 64 B je na stránce 4 kB/64 B = 64 bloků a index má 6 bitů. Kapacita cache je zde ale omezena na *m* (cest) stránek. Např. 8 cestná cache má kapacitu 8 x 4 kB =32 kB (Intel Haswell)
2. Když se bity VPN a PPN použité v indexu (superbity) shodují (stránky mají stejnou „barvu“). To může zařídit např. OS/SW. Pak se také index získá bez překladu.

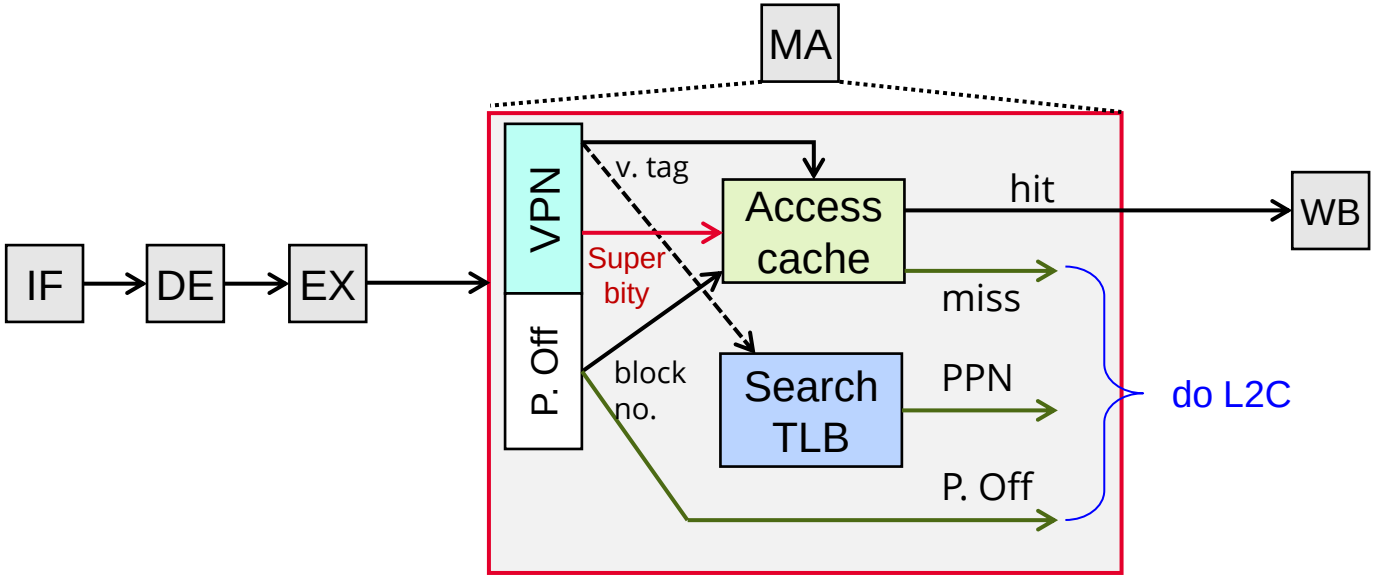

VPN Virtual Page Number
PPN Physical Page Number
P.Off Page offset
IF
DE
EX
Sériový přístup
VPN
P. Off
Search TLB
PPN
P. Off
tag
Super bity
Access cache
Nepřekládá se
data
WB
MA
fyz. index = číslo bloku
bez překladu pokud se
index kryje číslem bloku
IF
DE
EX
Paralení přístup
VPN
P. Off
Search TLB
Access cache
PPN
TAG
Data
compare tags
fyzický index = číslo bloku
data
WB

- **Indexování** L1C může začít hned, spolu s překladem v TLB.
- Následuje **porovnání tagu** z L1C s tagem získaným z TLB.



- **Problém Synonym**
  - 2 nebo víc VA v TLB se přeloží na stejný fyzický index

V/V cache (virtuální index i tag)



Virtuální index = block number + virtuální superbity

- **Při zásahu** v L1C se TLB neuplatní, **překlad není třeba**.
- **Výstup TLB** je použit jen **při výpadku v L1C**, tedy zřídka. Nalezené PPN se použije pro hledání v L2C.

Další komplikace:

- Jelikož TLB se většinou obchází, je třeba režijní bity stránky překopírovat z TLB do L1C.
- Kromě **synonym** je třeba ještě řešit **homonyma**: stejná VA se mapuje do rozdílných PA (přepnutí kontextu).

Pokračování příště