Predikce skoků, přednačítání instrukcí a dat

AVS – Architektury výpočetních systémů

Týden 4, 2023/2024

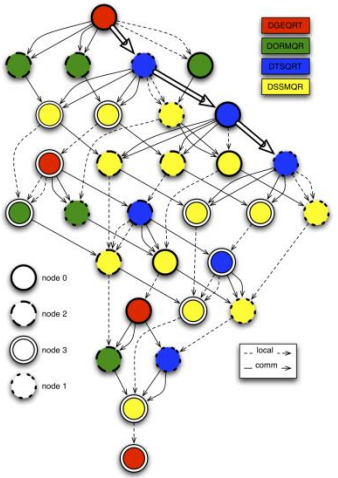
Jirka Jaroš

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

OPAKOVÁNÍ

Dynamické plánování instrukcí

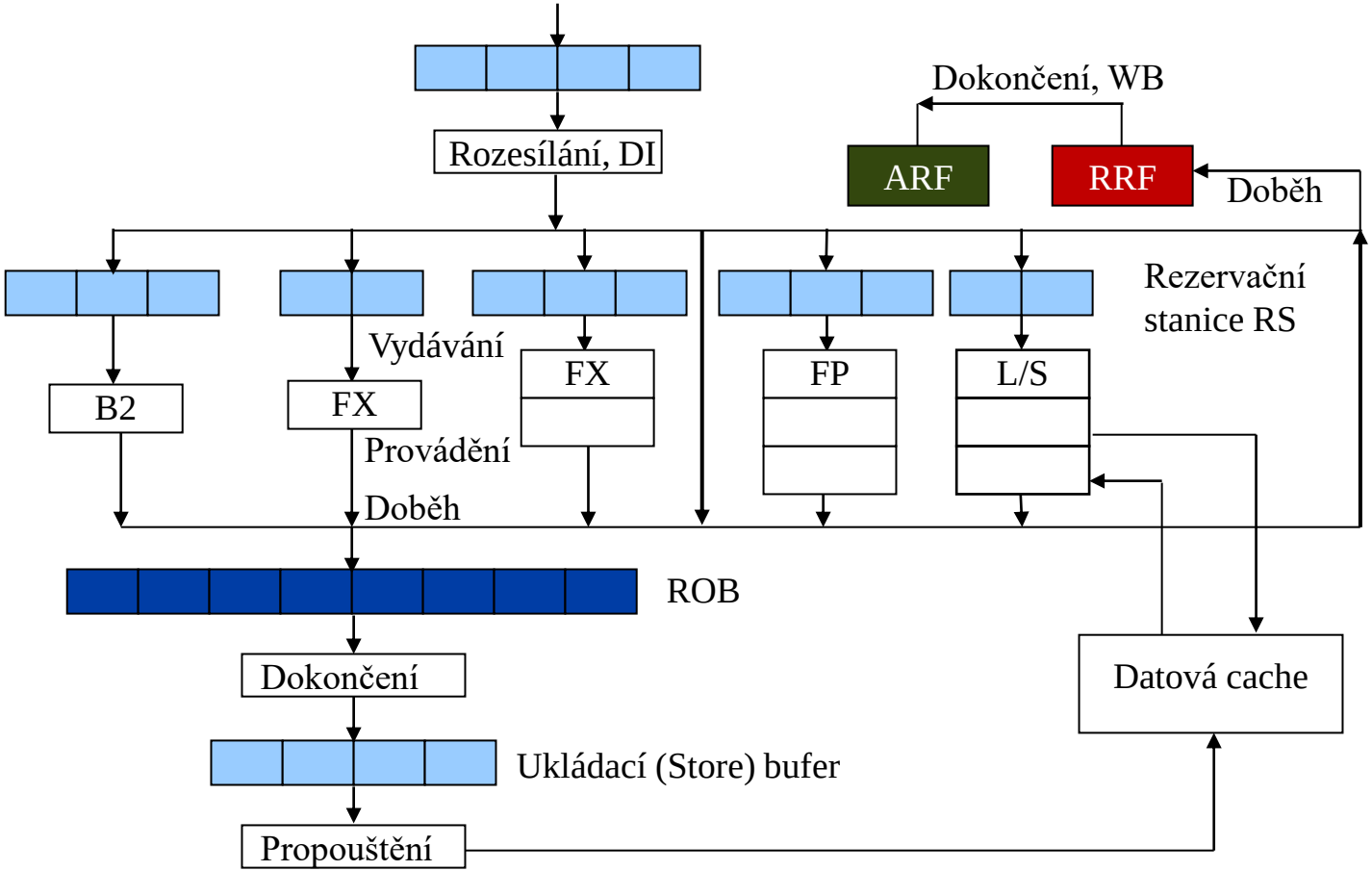
Instrukce jsou vydávány do FJ a prováděny **mimo pořadí** v programu, pokud mezi nimi **nejsou konflikty** a **FJ** jsou **volné**.



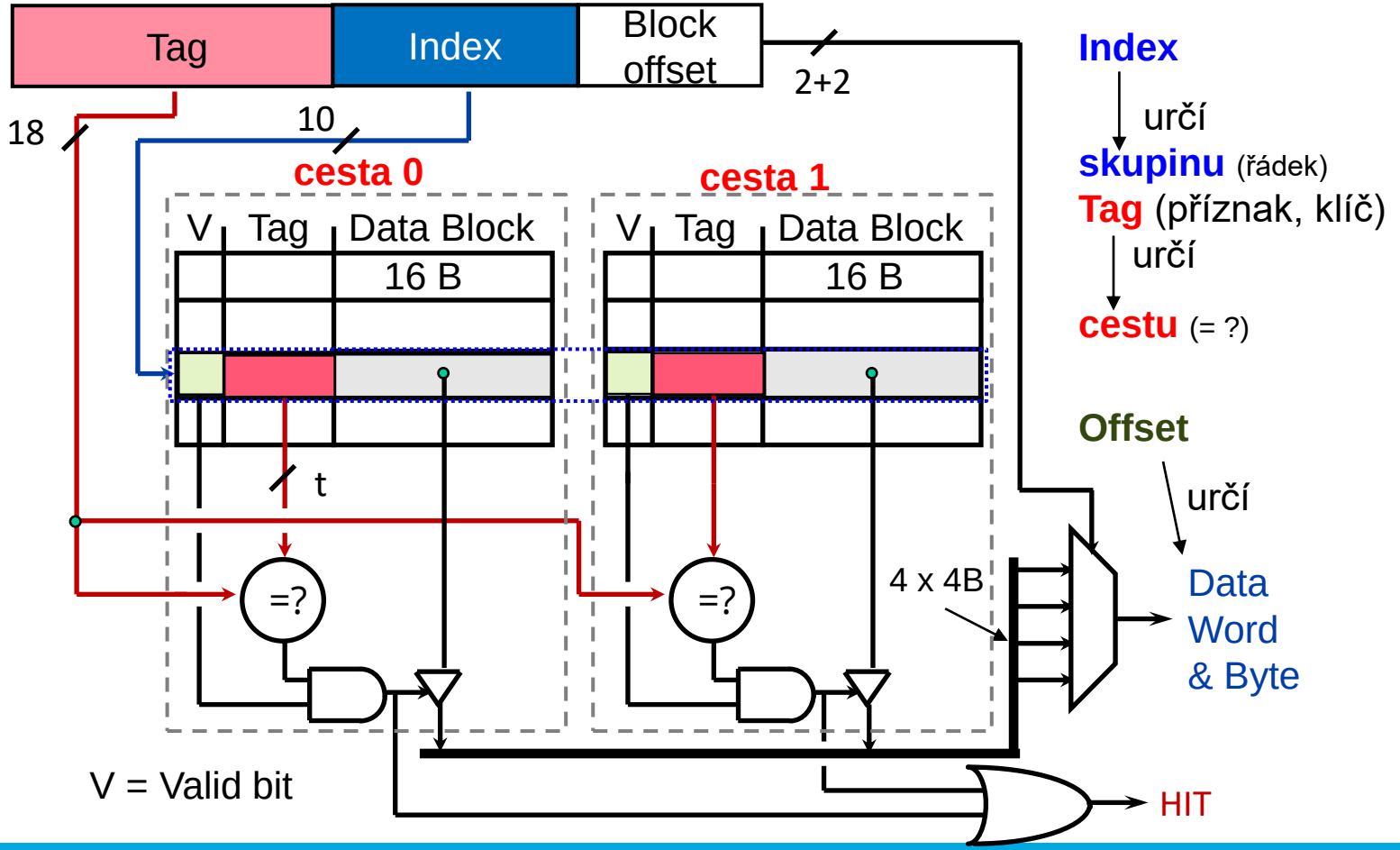
1. **ScoreBoarding** (Thorntonův algoritmus, 1964)
   - Registruje všechny **konflikty** (RAW, WAW, WAR) v **tabulce rozpracovaných instrukcí** a udržuje jejich skóre (SB).
   - SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB. **Přejmenování registrů neprobíhá**.
2. **Rezervační 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

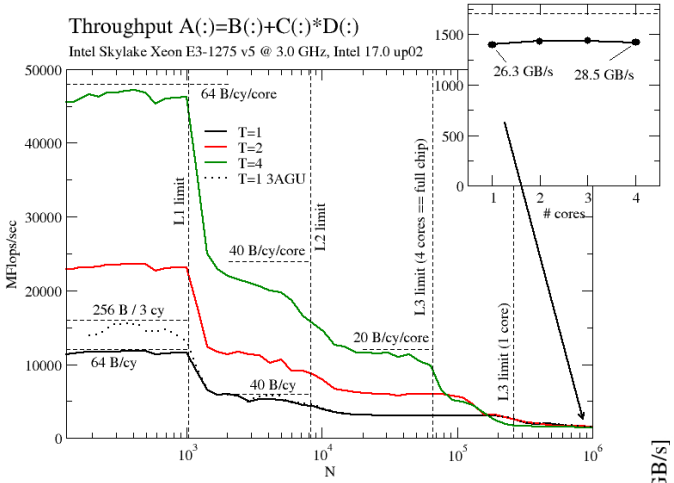
Superskalární procesor – Back end

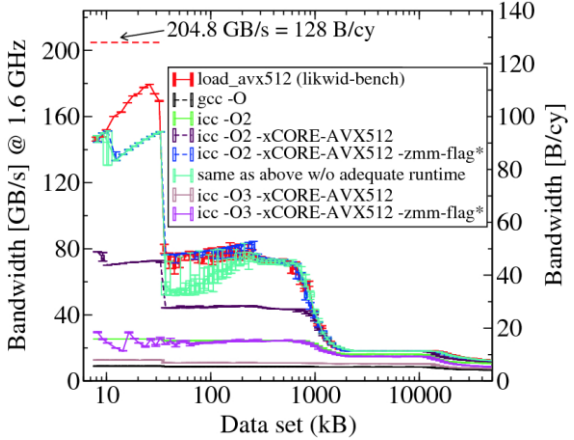


Skupinově asociativní (SA) cache

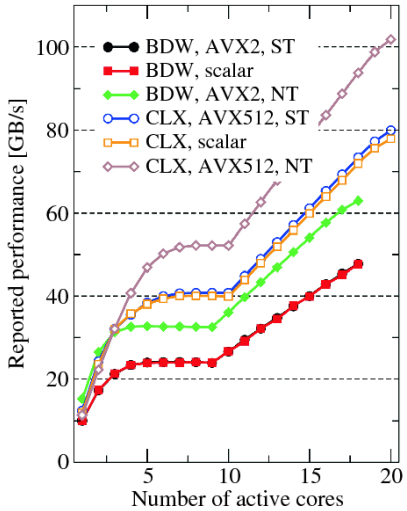


Propustnosti a latence cache

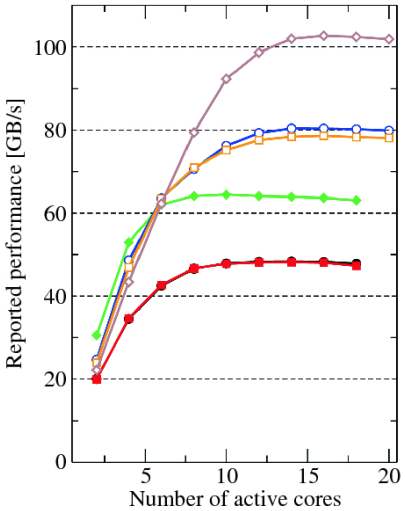




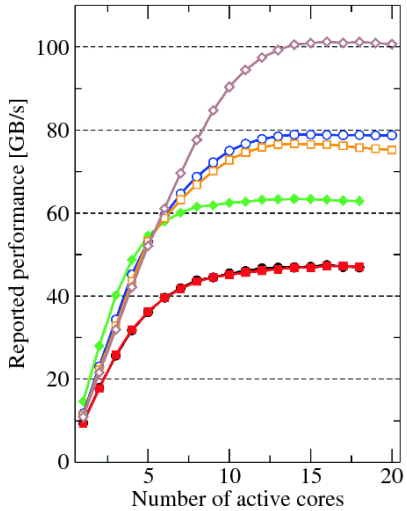
- Stream benchmark testující propustnost paměti a cache
- Vlevo – jedno jádro
- Vpravo – více jader
- https://link.springer.com/chapter/10.1007/978-3-030-50743-5_21



(a) CoD/SNC enabled, compact



(b) CoD/SNC enabled, scatter



(c) CoD/SNC disabled

Názvosloví

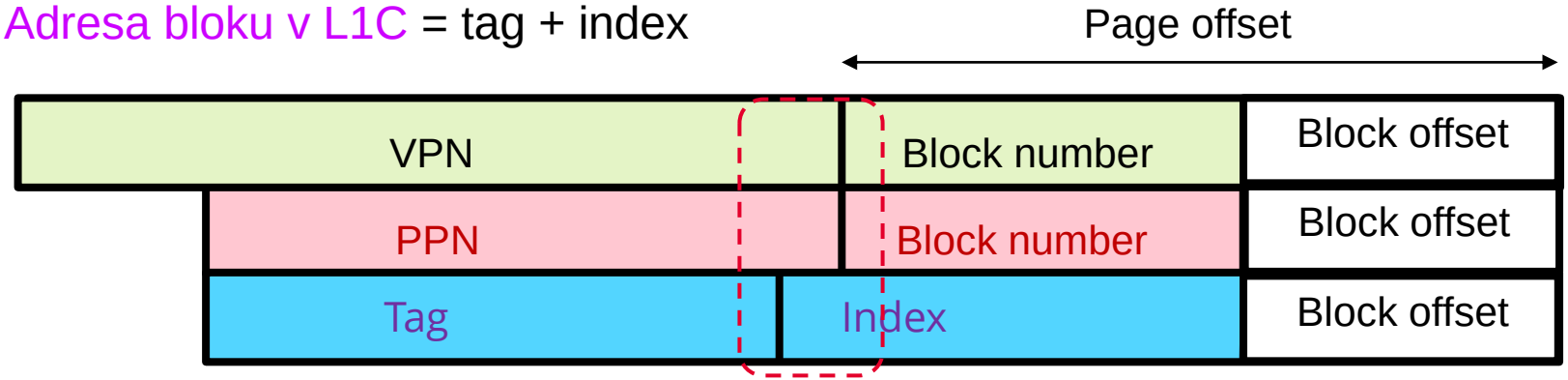
VA = Virtual Address = virtual page number **VPN** + **Pg. offset**

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

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.

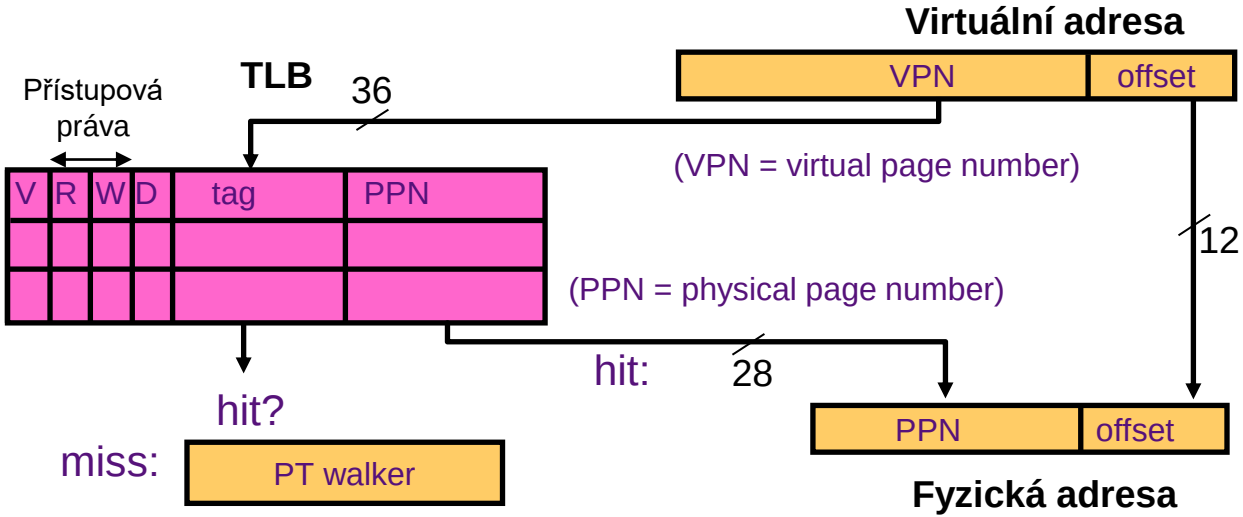
Translation Lookaside Buffer (TLB)

Překlad adresy je velmi drahý!

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

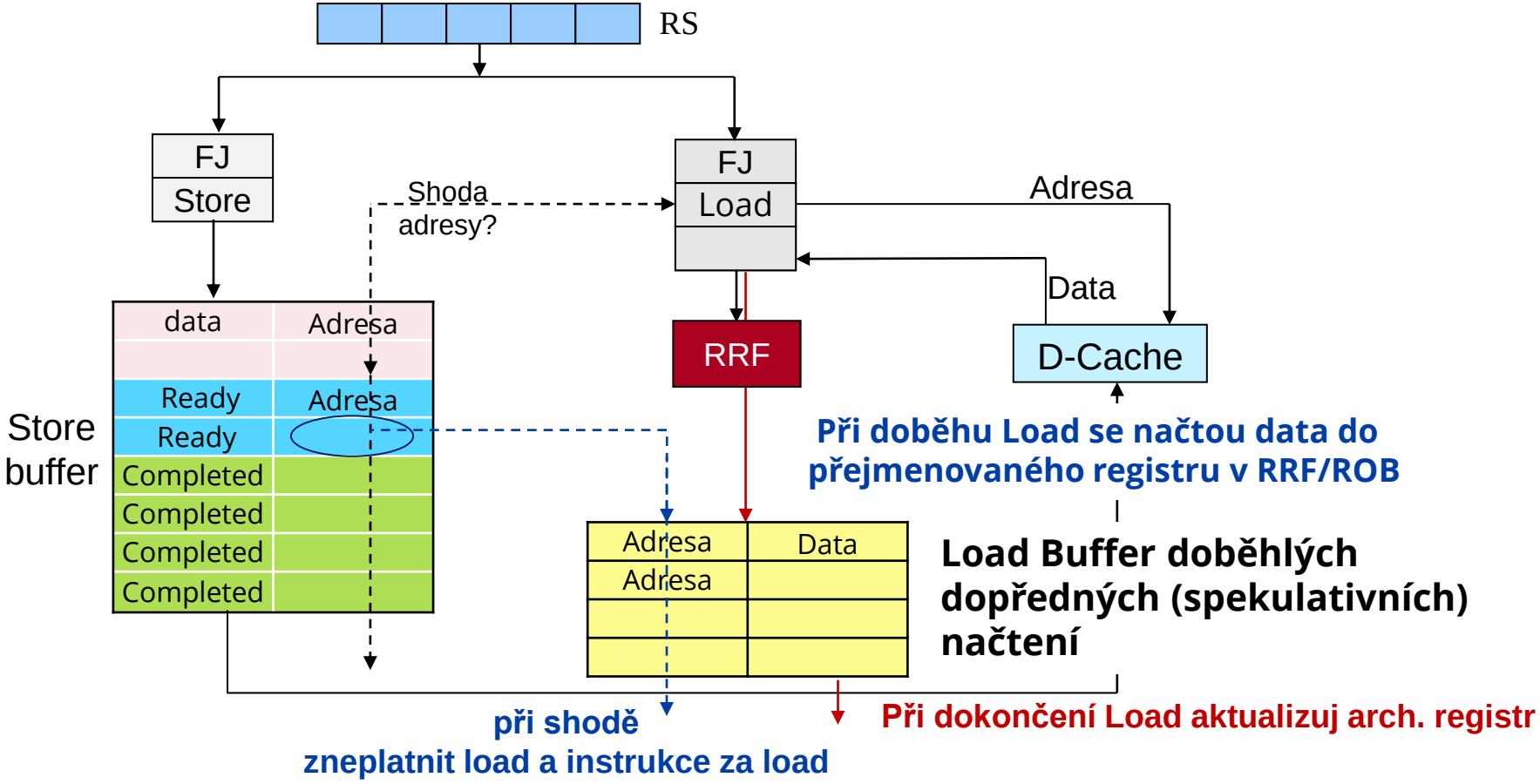
Řešení: ***Překladová tabulka TLB na čipu*** obsahuje sadu aktuálně používaných dvojic {VPN, PPN}

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



Překládá se jen číslo stránky, offset stránky zůstává stejný!

Out-of-order Load/Store jednotka



PREDIKCE SKOKŮ

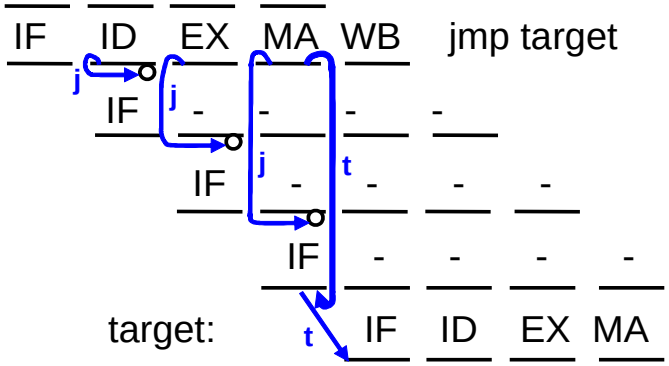
Řídící konflikty

- V průměru je každá 6. – 9. instrukce skoková
  - nepodmíněný skok - jump j, jump register jr
    - do podprogramu: jump and link, jal
  - podmíněný skok
    - bnez/beqz r1, target (test 1 registru ve stupni ID)
    - bne/beq r1, r2, loop (test 2 registrů ve stupni EX)

- Jaká data skok potřebuje:
  - nepodmíněný: op-kód = **j**, PC, rel. adresu (pole Imm 26 bitů) nebo obsah registru
  - podmíněný: op-kód = **b**, PC, rel. adresu (pole Imm 16 bitů), vyhodnocenou podmínku.

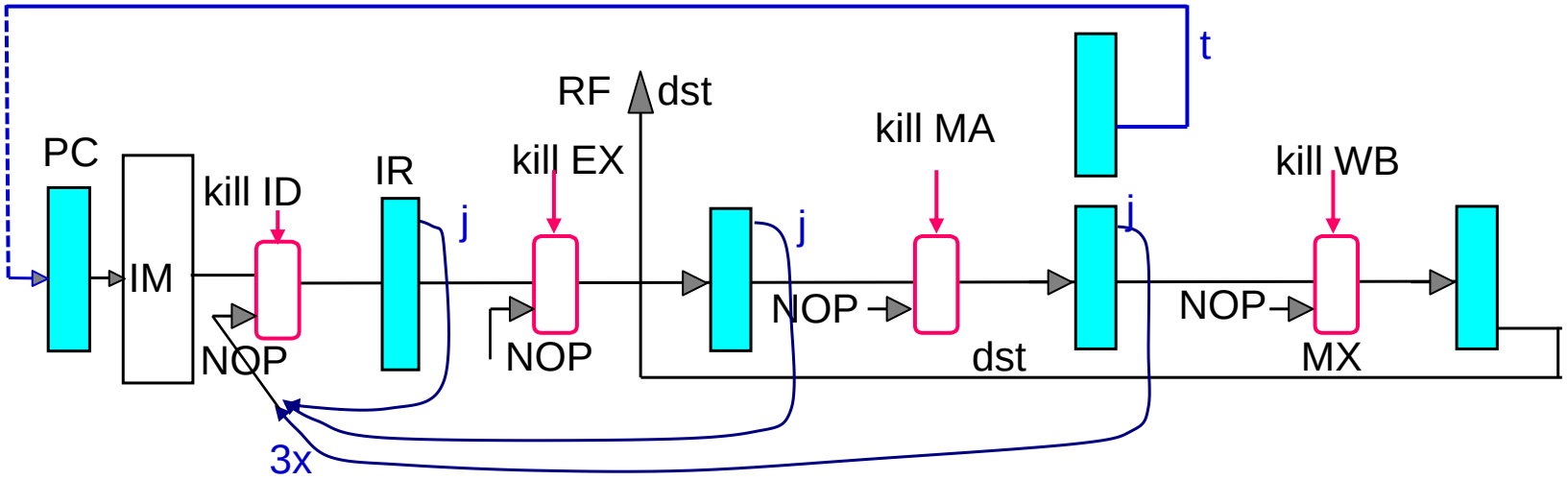
Výpočet cílové adresy (PC + rel. adresa) a podmínky je třeba co nejvíce urychlit!

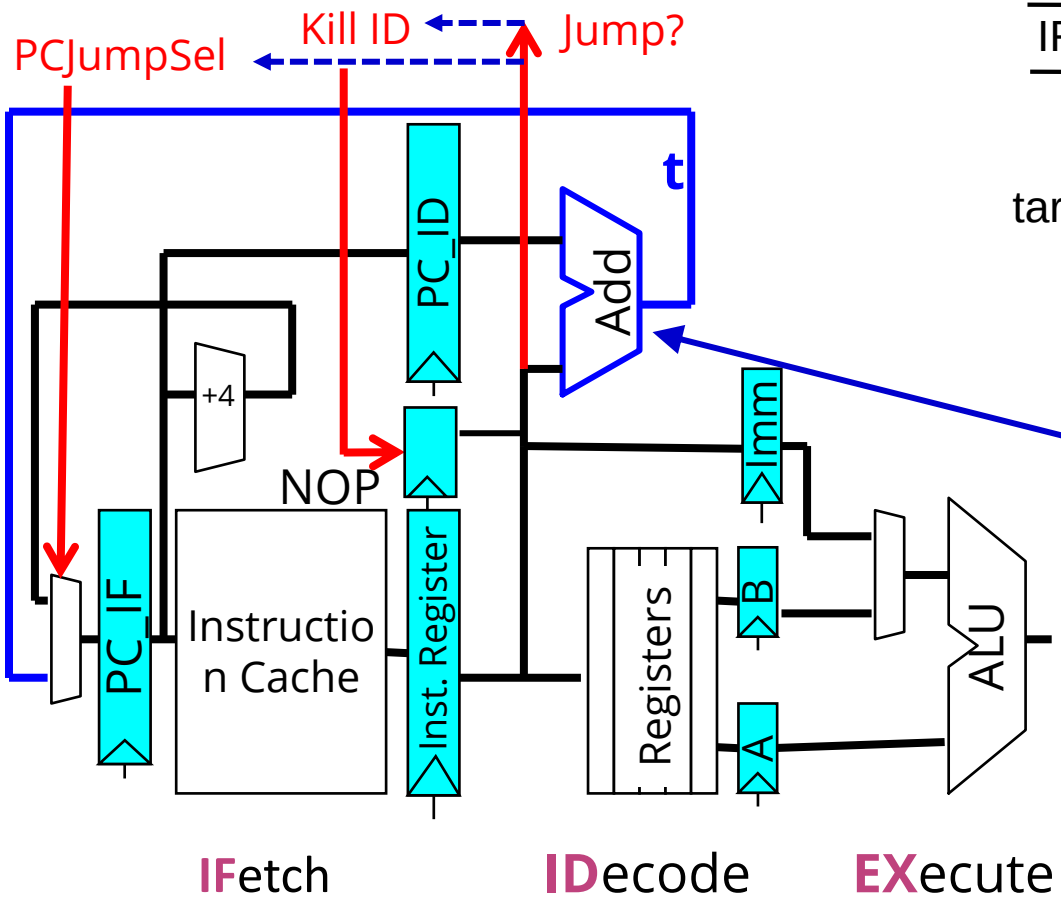
Skalární procesor – Nepodmíněný skok j relativně k PC

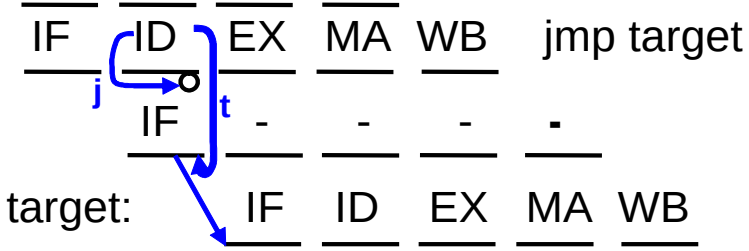


Adresa se spočte v EX, během MA se přepíše do PC → pokuta 3 takty.

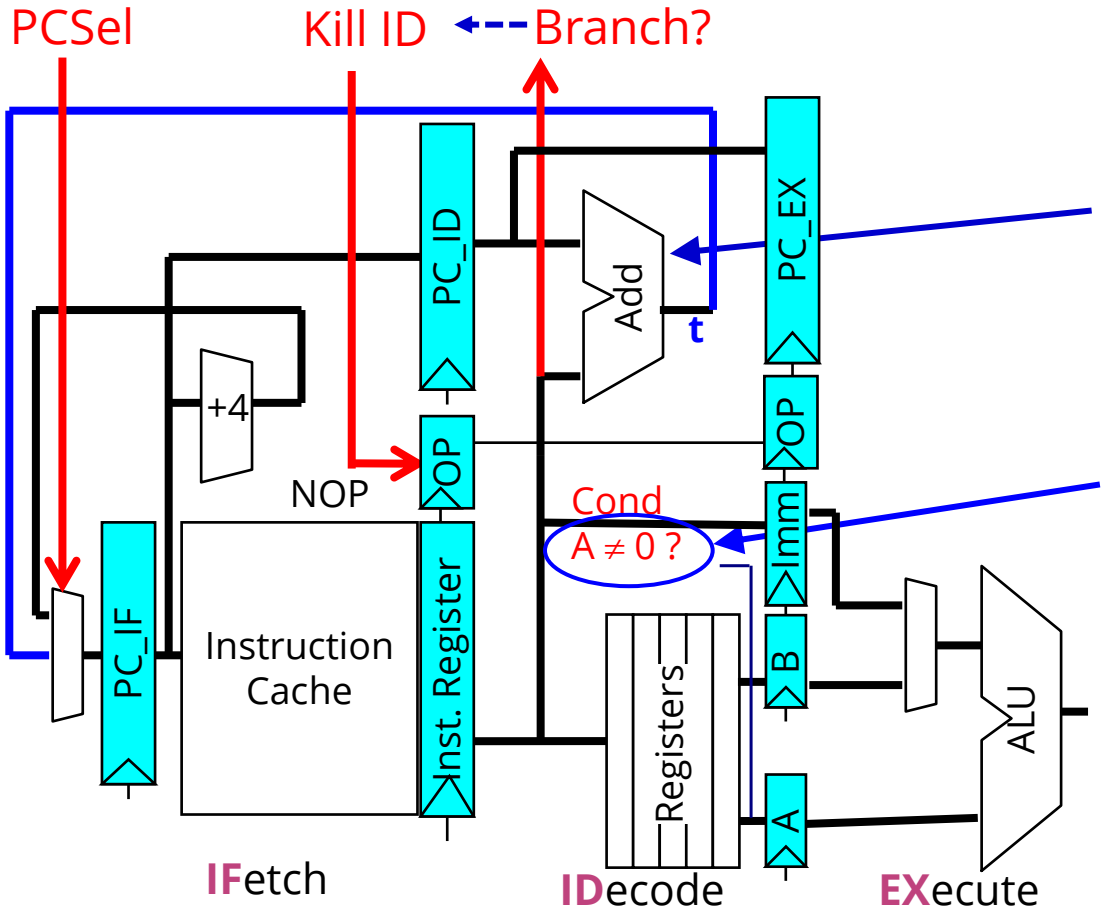
Storno již načtené nebo rozpracované instrukce (kill): injekce NOP do IR.







Nová sčítačka pro výpočet adresy **t** ve stupni ID: pokuta jen 1 takt



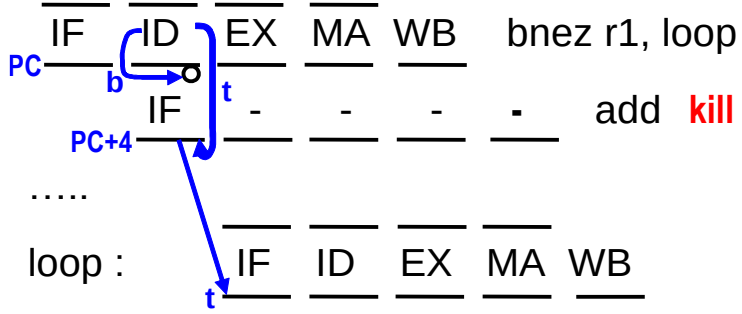
Stejná sčítačka pro výpočet adresy bnez a beqz jako pro jmp

Detekce nuly ve stupni ID

Pokuty u instrukcí bnez/beqz (test na nulu)

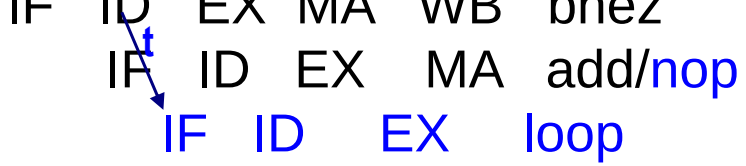
Pokud je testovaný registr zapsán s předstihem, je **pokuta 0 nebo 1 takt**:

```
loop:   …
        bnez  r1, loop
        add   …
```

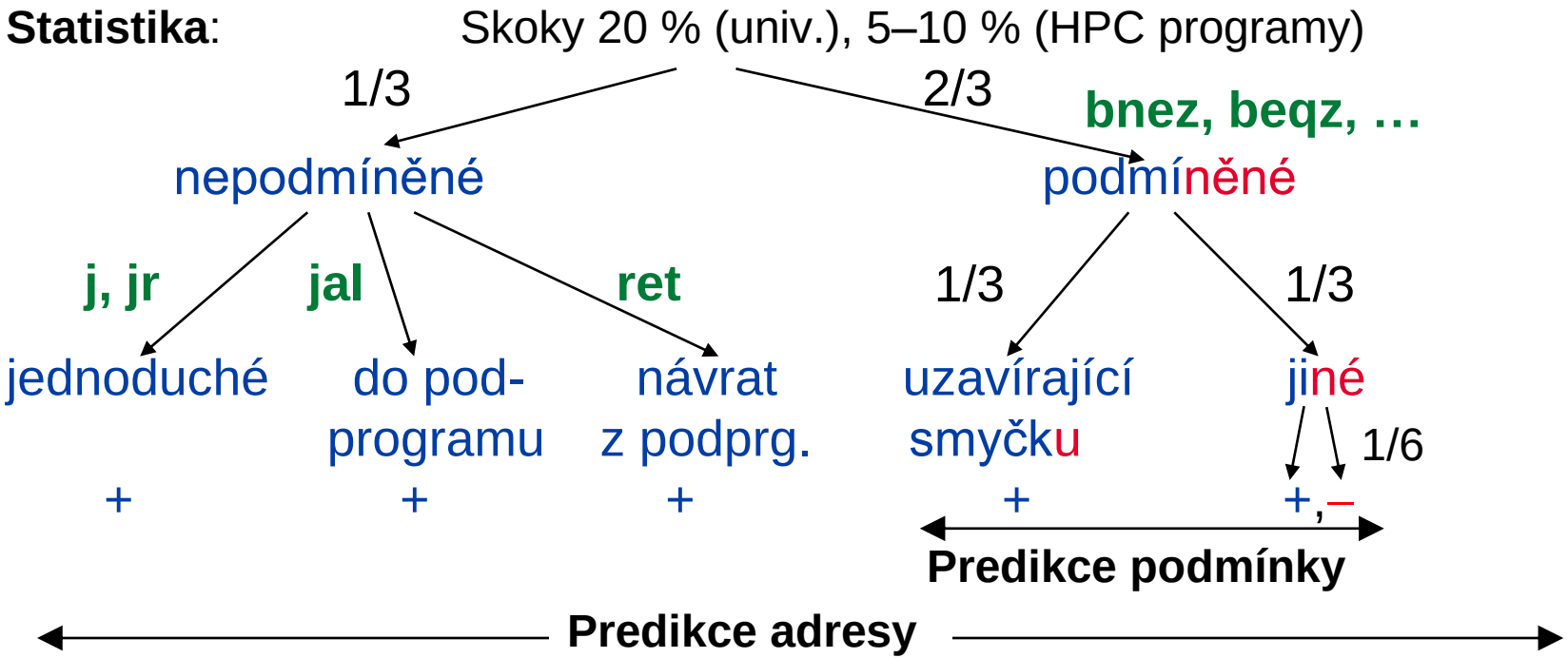


Fixní negativní predikce: jedeme dál s add. Pouze je-li test true, stornujeme add a aktivujeme bypass pro cílovou adresu *t* (skok na loop). Pokuta pak bude:

- 1 takt když se skočí na loop
- 0 když pokračuje add.



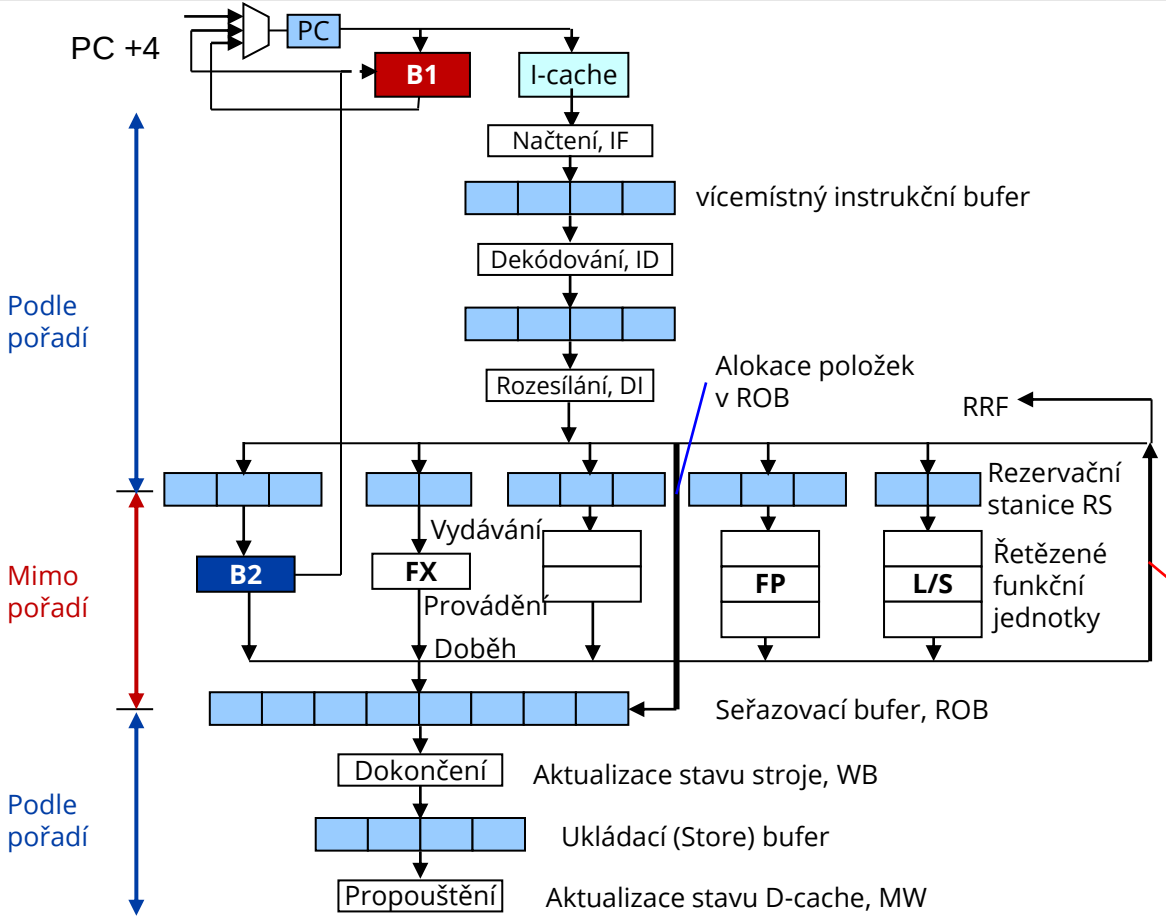
Skoky jsou vysoce předvídatelné



Predikce podmínky:

- statická (podle testu $\neq 0$, >0, ≥ 0, směru skoku kompilátorem – predict bit)
- dynamická (za běhu programu)

Superskalární procesor



- B, Branch unit – jednotka pro zpracování skoků
  - B1 – spekulativní
  - B2 – reálné

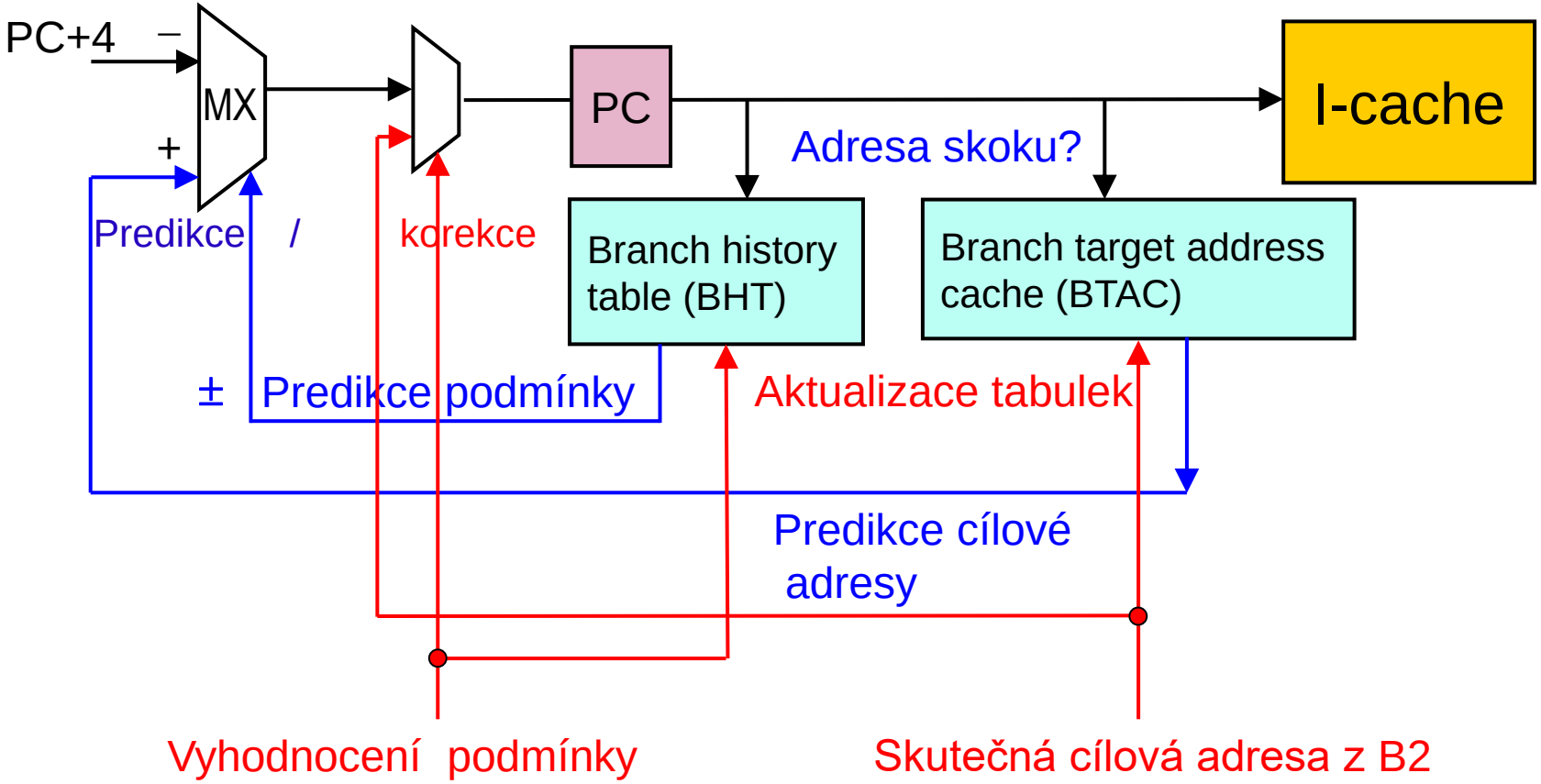
Předávání výsledků do RS a RF přes **CDB** (společná datová sběrnice)

Dynamická predikce skoků

Pro provedení skoku potřebujeme předpovědět dvě věci:

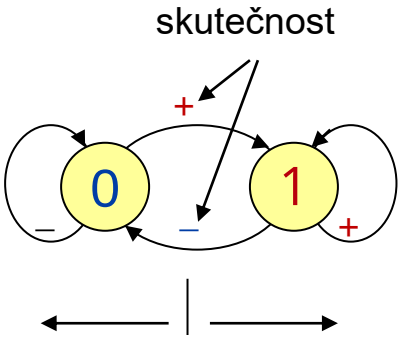
- Predikce **podmínky** skoku – jen u podmíněných skoků
  - 1 bitový prediktor
  - 2 bitový prediktor spolupracující s BHT Branch History Table
  - 2 bitový prediktor spolupracující s PHT Pattern History Table
  - Korelační prediktory
- Predikce **cílové adresy** (cílové instrukce) u všech skoků
  - BTB Branch Target Buffer
  - BTAC Branch Target Address Cache
  - BTIC Branch Target Instruction Cache
  - RSB Return Stack Buffer (prediktor návratových adres)

Predikce podmínky a cílové adresy skoku (B1)



Jednobitový prediktor podmínky skoku

1 skok = 1 bit v BHT,
Branch History Table



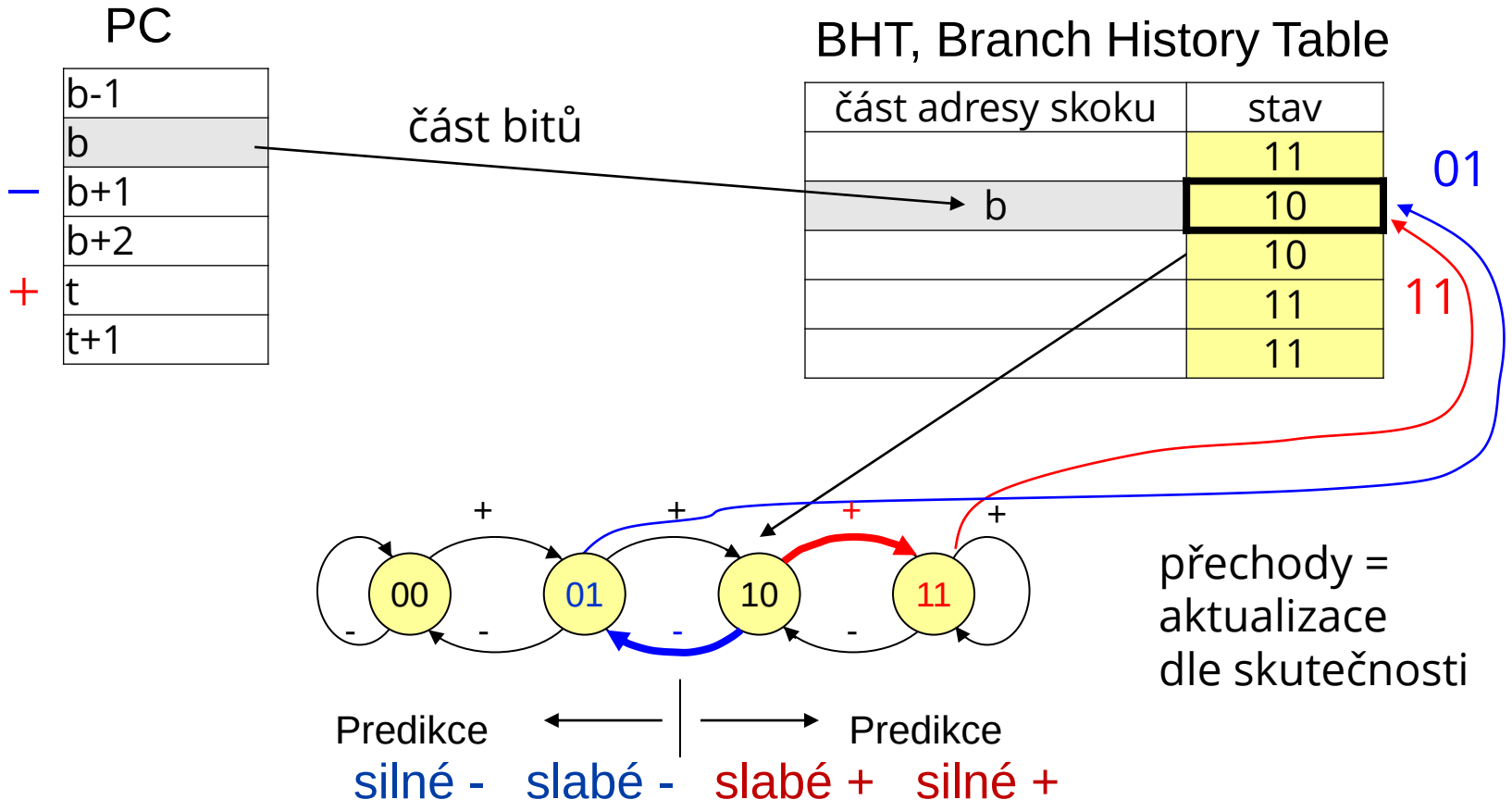
1 bitová predikce skoků:

- Když se skočilo, očekávám že se zase skočí (stav 1)
- Když se neskočilo, očekávám že se zase neskočí (stav 0).

Implementace:

- Pro každý skok je třeba uložit stav (1 bit) v tabulce BHT.
- **Kvůli rozměrům** je tabulka indexovaná jen vybranými bity PC nebo nějakou hashovací funkcí nad PC.

2 bitový prediktor podmínky skoku



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    do_work(i, j);
```

Vnější smyčka, jiný skok

Skutečnost:

+++++– ….+….+++++–….+…. +++++–….+….+++++–….

- **1 bitová predikce:**

–+++++……–+++++ ……. –+++++……–+++++….

2 chyby na 1 průchod vnitřní smyčky

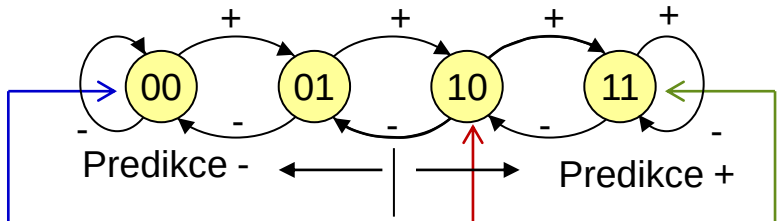
- **2 bitová predikce** (má jistou setrvačnost):

– –+++++…… +++++++ ……. +++++++……+++++

1 chyba na 1 průchod vnitřní smyčky (zanedbáme-li 2 počáteční chyby – význam počátečního nastavení prediktoru!)

Skutečnost:

–+–+–+–+–+–+– +–+–….



2 bitová predikce, počáteční stav silné –, levá smyčka

– – – – – – – – – – – – …. 50% úspěšnost

2 bitová predikce, počáteční stav silné +, pravá smyčka

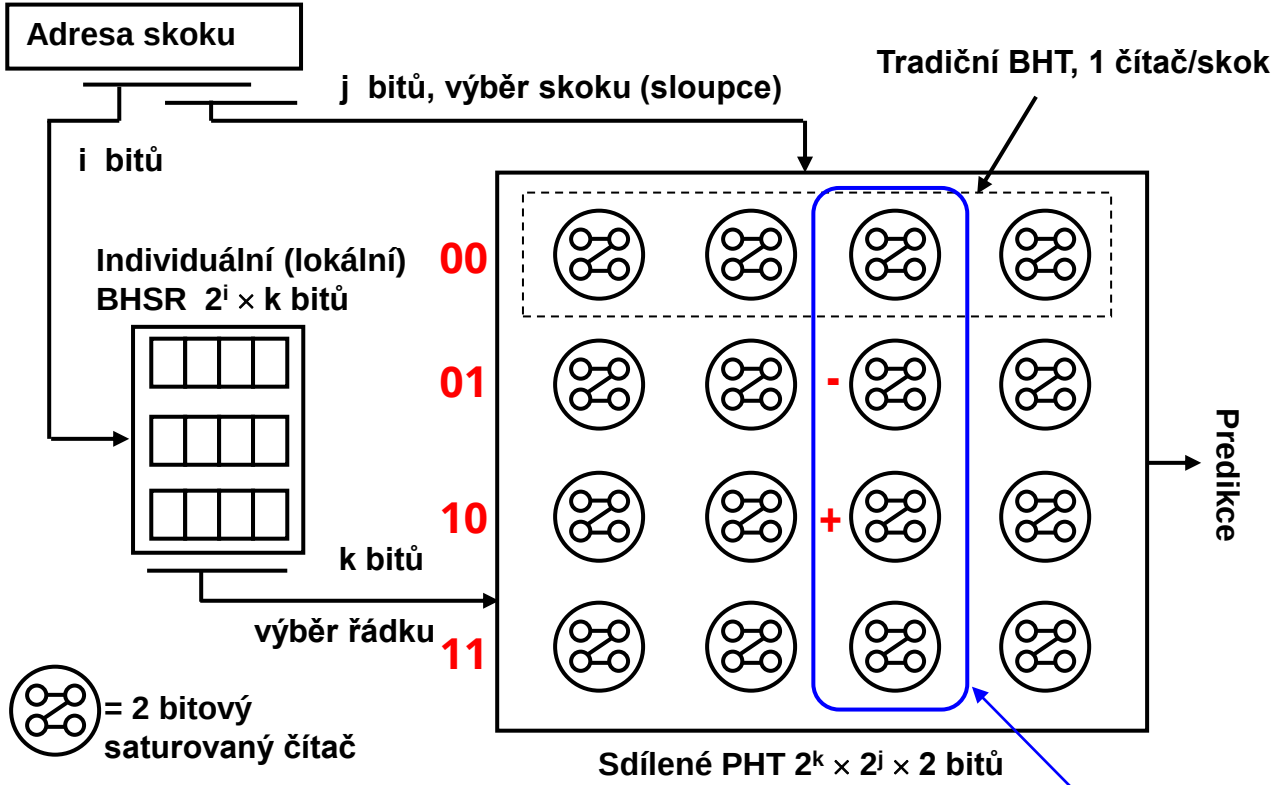
++++++++++++++++…. 50% úspěšnost

2 bitová predikce, počáteční stav slabé +, střední smyčka

+–+–+–+– +–+– +–+–….nulová úspěšnost

Řešení (Yeh & Patt) adaptivní prediktor – podle posledních ***k*** výsledků každého skoku vybrat jeden z 2^k prediktorů a ten použít. (*k* = 2: prediktor **10** by předpovídal +, prediktor 01 by předpovídal -, další 2 prediktory 00 a 11 by se neuplatnily).

Adaptivní prediktor s lokálními BHSR (Branch History Shift Register)

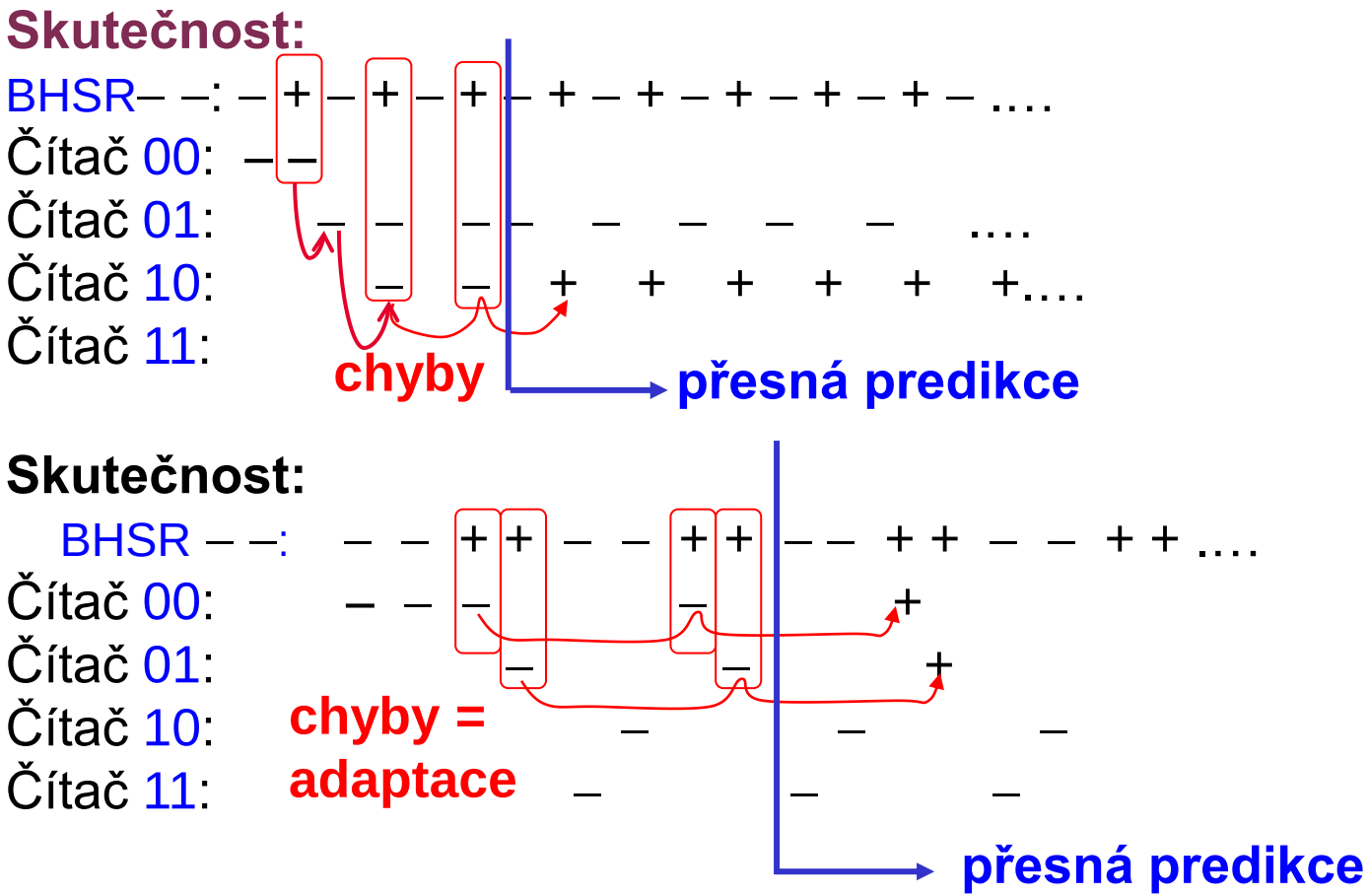


čítače odpovídající 2 bitovým vzorkům historie jednoho skoku

Adaptivní prediktor podmínky, BHSR *k* bitů

- Používá 2^k tradičních tabulek BHT. Tato 2D struktura čítačů se nazývá **Pattern History Table, PHT**.
- Pro daný podmíněný skok (identifikovaný obecně i bity PC) a podle k předchozích výsledků tohoto skoku, ukládaných do k-bitového posuvného registru historie skoků (**BHSR – Branch History Shift Register**) se vybere jeden BHT (řádek PHT); každý skok má svůj lokální BHSR.
- Část adresy skoku (j bitů) představuje index do vybrané BHT (sloupec PHT) a určí příslušný 2 bitový čítač.
- Nový obsah BHSR se získá vložením výsledku (1 bitu) příslušného skoku. PHT se aktualizuje jen u podmíněných skoků.

Příklad: Adaptivní prediktor $k = 2$ bity historie



Adaptivní prediktor s lokálními BHSR délky *k*

Umí se naučit přesně predikovat každou opakující se sekvenci
(adaptuje se na určitý vzorek)

- délky $\leq k + 1$ nejvýš po **třech** pokusech,
- délky $k + 1$ až 2^k, pokud všechny pod-sekvence délky k jsou různé.
- nepravidelné sekvence mají více špatných predikcí než u prediktoru bez BHSR.

Příklad: $k = 4$
000011 umí, protože 0000, 0001, 0011, 0110, 1100, 1000 jsou různé,
00001 umí, protože 0000, 0001, 0010, 0100, 1000 jsou různé,
000001 neumí, 0000, 0000, 0001 … nejsou různé

Korelační prediktory

- **Problémy adaptivních prediktorů:**
  - Velikost prediktorů roste exponenciálně s počtem bitů historie.
  - Neberou v úvahu možnou korelaci mezi provedením uvažovaného skoku a skoků na cestě k němu (např. vnořené smyčky).
- **Korelační prediktory**
  - Nepoužívají lokální BHSR, ale jeden globální GBHSR pro všechny skoky.
  - Tím zohledňují dynamický kontext posledně provedených skoků programu (a nikoliv výsledky jen jednoho skoku v čase).
  - Při k = 8 až 16 je jejich přesnost lepší než 95 %.
  - Nevýhoda: PHT je příliš velké a využití položek nerovnoměrné.

Predikce skoků prakticky

- **Statická** – Založena pouze a jen na instrukci (bit v OP code)
- **Dynamická** – Založena na historii vykonání (x-bitový)
- **Lokální** – Historie udržována jen pro danou instrukci (adaptivní)
- **Globální** – Historie udržována přes sekvenci skoků (korelační)

```
// generates random numbers from uniform distribution [1, 10]
for (size_t i = 0; i < data.size(); i++){
  data[i] = dist(rng);
}

Type sum = 0;
for (int r = 0; r < REPETITIONS; r++)
  for (int i = 0; i < SIZE; i++)
    if (data[i] < 6)
    {
      sum += data[i];
    }
```

Počet chybných predikcí

- **Perf**
  - `$ perf stat -e branch-misses ./filter`

    212 565 554 branch-misses
  - `$ perf list # list all available counters`
- **Intel Vtune**
  - `$ ml VTune`
  - `$ amplxe-cl -collect uarch-exploration -- ./filter`
  - Branch Mispredict: 33.7% of Pipeline Slots
- **Programmatically PAPI**

Jak zredukovat počet chybných predikcí

- Více predikovatelná data
  - Seřadit pole
  - 212M branch-misses -> 7M
- Odstranění nebo náhrada skoků
  - `sum += data[i] * (a < 6);`
  - `sum += (a < 6) ? data[i] : 0;`
- Compiler hints
  - `if (__builtin_expect(a < 6, 1)`
- PGO (profile-guided optimization)

- **Co když neznáme cílové adresy?**
  - Virtuální funkce, ukazatele na funkce, návraty z volání rutin

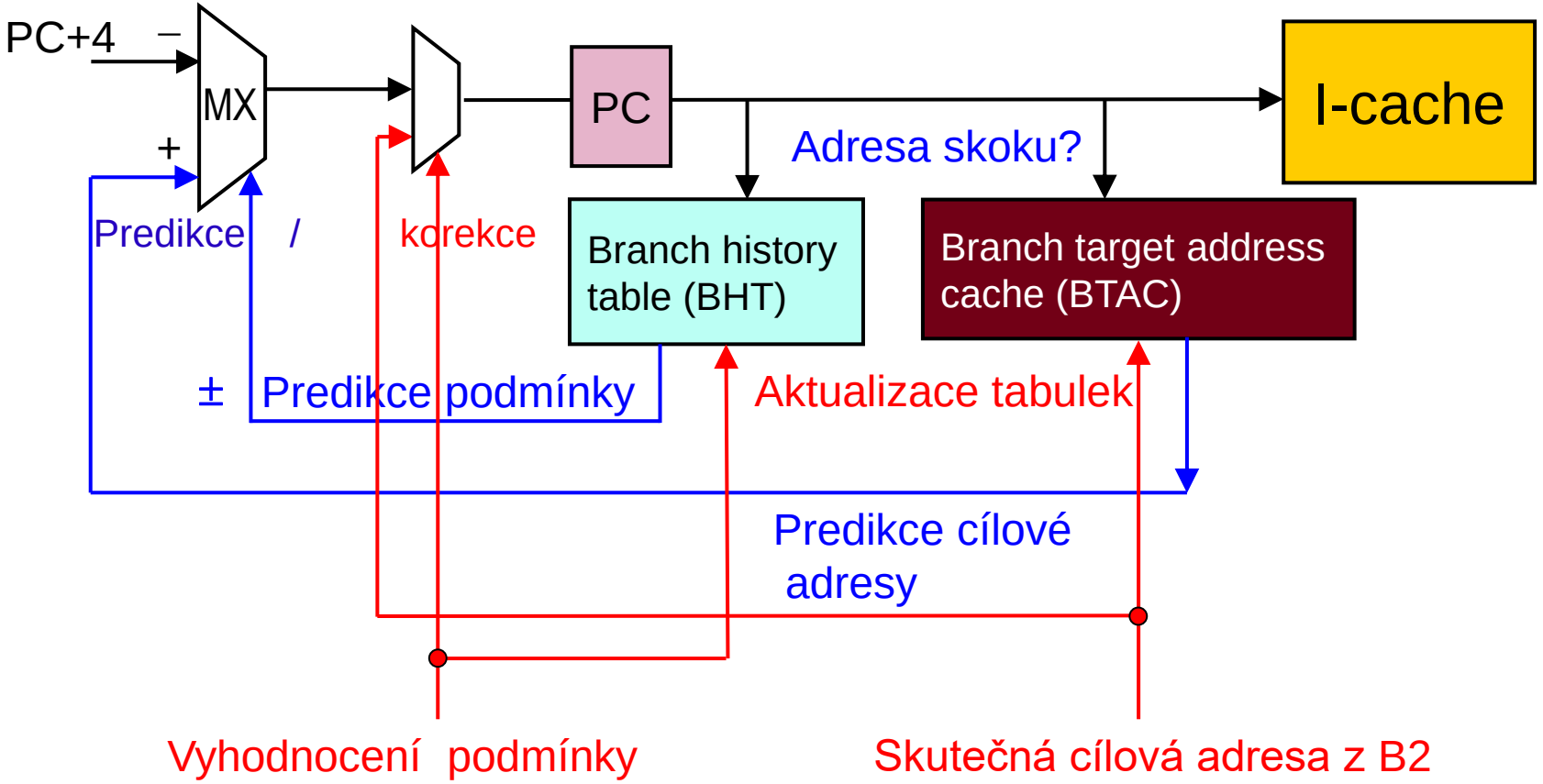
Predikce cílové adresy skoku

- Hned ve fázi IF, jakmile se rozpozná že PC ukazuje na skokovou instrukci (**b** nebo **j**), dá se část adresy skoku použít jako tag a index do malé cache, ve které jsou uloženy **předchozí cílové adresy skoků** (**t**, target = cíl).
- Tato malá cache se označuje jako
  - BTB (Branch Target Buffer) nebo
  - BTAC (Branch Target Address Cache)

Velikost BTB:
512 až 8192 položek

- **Nepřímé skoky** (**jr**): skáče se na různá *t*, v BTB může být
  - jen poslední *t*, takže dost špatných predikcí nebo
  - všechna *t* (Core 2) – oddělený BTB využívající GBHSR
- **Návraty z funkcí** (ret) se řeší jinak, viz dále (RSB)
- **Návraty z konce smyček** – oddělený BTB a loop counter

Predikce podmínky a cílové adresy skoku (B1)



Spojená predikce cílové adresy i podmínky

Pokud jsou v BTAC pouze **cílové adresy t** uskutečněných skoků, je s predikcí adresy automaticky spojena 1 bitová predikce podmínky. (skok se udělal: položka v BTAC existuje → příště skoč; skok se neudělal: položka se vyřadí → příště neskoč).

Častěji se k BTAC připojuje BHT se 2 bity.

- Když BHT říká „**skákat se bude**“ a ***t* existuje** v BTAC, **skočí se**
- Podobně „skákat se nebude“ a *t* neexistuje…, neskočí se
- Když BHT říká „skákat se nebude“ a ***t* existuje** v BTAC, neskočí se, pokračuje se na *b* + 4 (predikce BHT je silnější);
- Když BHT říká „**skákat se bude**“ a *t* neexistuje v BTAC:
  nevíme, že jde o skok (alias v tabulce)
  musí se čekat, až se spočítá efektivní adresa skoku a podmínka.

Aktualizace BHT a následně BTAC podle skutečnosti.

Zotavení ze špatné predikce

Predikce **podmíněných** skoků má 2 kroky:

- spekulaci (jednotka B1) v době IF:

  $b \rightarrow \text{BTB},\ b + 4 \rightarrow \text{PC}$ (predikce –)

  $b \rightarrow \text{BTB},\ t = \text{BTB}(b) \rightarrow \text{PC}$ (predikce +)
- její ověření: adresa a podmínka platí? (jednotka B2).

Než se skok ověří, provádějí se další instrukce včetně skoků s predikcí.

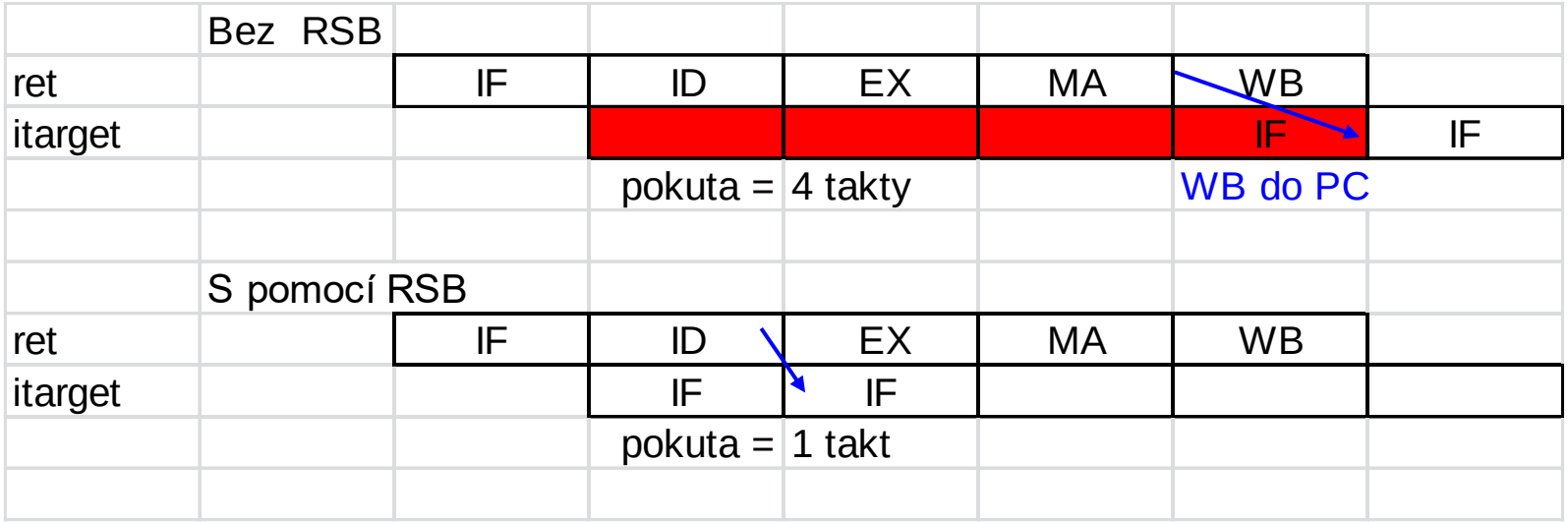
- Tyto **spekulativní instrukce** (jejichž provedení nebo storno závisí na správnosti predikce) je třeba identifikovat příznaky.
- Při **správné predikci** je ze spekulativních instrukcí příznak odstraněn.
- Při **špatné predikci** je špatná větev ukončena a příznaky se použijí k odstranění těchto instrukcí z buferů ROB i RS.

Prediktory návratových adres

- 85 % nepřímých skoků jsou **nepodmíněné skoky *ret*** z konce volané funkce zpět do hlavního programu.
  Pokud je adresa návratu na zásobníku v paměti D-cache, instrukce *ret* provádí její načtení a WB do PC.
- Běžná predikce pomocí BTB by byla nepřesná (funkce se volá z více míst → více návratových adres, takže při návratu jinam než na předešlou adresu → BTB miss)

- **Řešení**: ukládáním návratových adres do **malého zásobníku přímo v CPU** (Return Stack Buffer, RSB nebo Return Address Stack, RAS)
  - Při zjištění v ID, že instrukce je návrat, načte se adresa návratu z RSB, hned v dalším taktu je již v PC a pokračuje hlavní program.
  - Tím se současně redukuje zpoždění při načtení adresy návratu z D-cache.

Prediktor návratových adres, příklad zrychlení u skalárního procesoru



Za předpokladu, že instrukcí návratu je v programu 5 %
a neuvažujeme-li pokuty u dalších instrukcí (CPI = 1), je zrychlení **skalární** CPU dosažené pomocí RSB ~ 14 % :

$$S = \frac{CPI + 5\% * 4 takty}{CPI + 5\% * 1} = \frac{1 + 0{,}2}{1{,}05} = 1{,}14$$

Loop exit prediktor (Intel Core)

- Skok na konci smyčky se chová tak, že z *n* průchodů
  - jde *n*-1 jedním směrem (na začátek)
  - 1 průchod na další instrukci.
- Skoky na konci (vnořených) smyček je možné predikovat samostatně pomocí počítadla iterací – **loop counter (LC)** a oddělené cache cílových adres skoků (BTB) jen pro smyčky s přídavnou informací o skoku (*n*).
- Při prvním průchodu LC napočítá do *n*, při dalších průchodech se LC porovnává s *n* a predikuje se exit pro *n*-tý běh.

- Např. 6 bitový čítač LC umí přesně predikovat skoky až do 64 iterací smyčky.

Spekulativní zpracování skoků

Instrukce za podmíněným skokem jsou vydávány, dynamicky plánovány i prováděny, jako kdyby predikce byla správná.

Spekuluje se i s více skoky za sebou, i se 2 skoky v 1 taktu. Vyžaduje to:

- dynamickou predikci skoků
- místo pro spekulativní výsledky (hodí se ROB! nebo RRF),
  ROB: [typ instrukce, dst reg., stav instrukce, hodnota]
- označení operandů a instrukcí „spekulativní/potvrzený“,
- storno špatně spekulovaného úseku a repete,
- ignorování výjimek dokud není jasné, že k nim dojde.

- **Spekulativní zpracování instrukcí po predikci skoku:**
  - Spekulativní instrukce v ROB jsou označeny (1 bit) a nemohou být propuštěny, dokud se nerozhodnou příslušné predikované skoky.
  - CPU zpracuje výjimku pouze u potvrzené instrukce na čele ROB.
- **Spekulativně lze také načítat data dopředu:**
  - Spekuluji, že se předem načtená data do použití již nezmění.
  - Jsou použity instrukce nevyvolávající výjimky, např. sld (spekulativní load) a existence výjimek se testuje odděleně později.

PŘEDNAČÍTÁNÍ INSTRUKCÍ

Načítání instrukcí

Fakta:

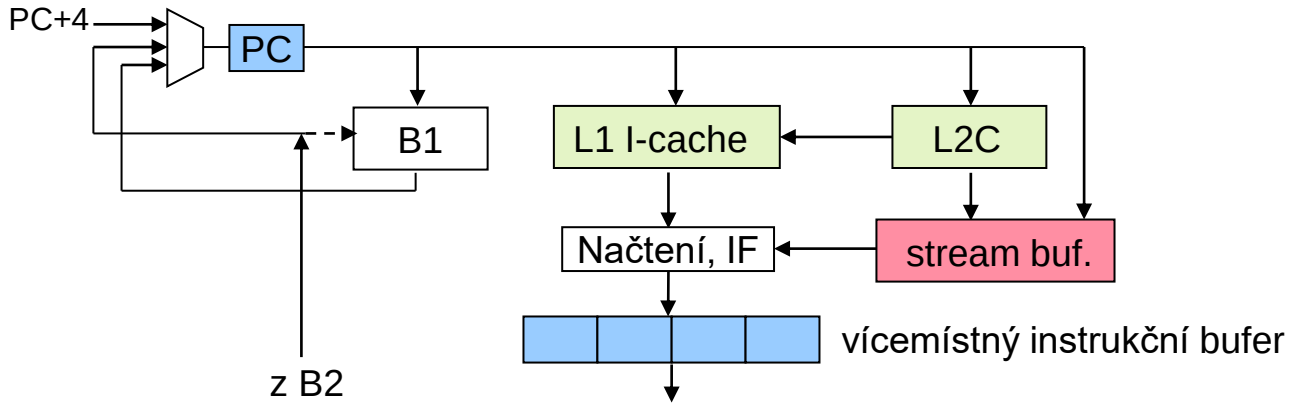
- OOO procesory načítají více instrukcí najednou.
- Instrukce jsou v I-cache nebo v L2C (nebo výš).
  - V bloku I-cache 64 B je např. sekvence 8 instrukcí po 8 B.
  - Přes hranice bloků je třeba načítat bez dalšího zpoždění (i při výpadku).
- Fragmentace bloků I-cache
  - Načítání sekvenčního sledu instrukcí končí skokem.
  - Z výchozího a cílového bloku I-cache se tak při skoku použije jen fragment, při špatné predikci dojde k výpadku.
- Instrukce načtené v 1 taktu mohou obsahovat více skoků, jejichž predikce ovlivní další načítání.

Závěr:

- K vyloučení výpadků v L1C je třeba načítat instrukce z L2C do L1C dopředu před jejich použitím – **přednačítat**.

Jednotka načítání instrukcí (IF)

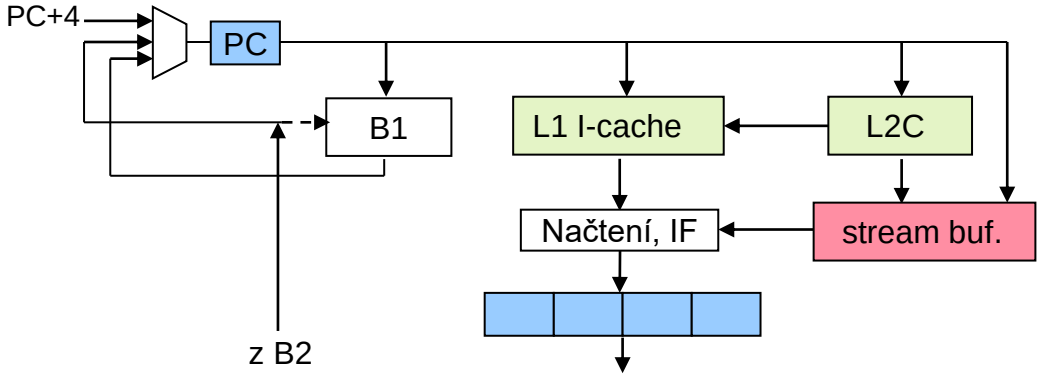
- Autonomní jednotka, která zásobuje instrukcemi další stupně linky
- **Tato jednotka integruje více funkcí:**
  1. predikci skoků
  2. přednačítání instrukcí
  3. přístupy do více bloků cache, jejich spojování
  4. dočasné uložení instrukcí v buferu



HW přednačítání instrukcí

- **Typicky procesor při výpadku v I-cache načítá z L2C 2 bloky:**
  - žádaný blok do I-cache
  - s předstihem následující blok na adrese získané inkrementací PC nebo predikcí do **instrukčního streamového buferu.**

- **Je-li žádaný blok ve stream buferu:**
  - žádost do cache je zrušena
  - blok je načten ze stream buferu
  - je vydána žádost na další přednačtení (jen při 1. přístupu do bloku).



- **Načítání do vícemístného instrukčního buferu má ještě další účel:**
  - jednotka načítání může pospojovat a zkombinovat instrukce z bloků cache do front (buferů) a vytížit tak lépe dekodéry (načítání přes hranice bloků, fragmenty bloků vlivem skoků).

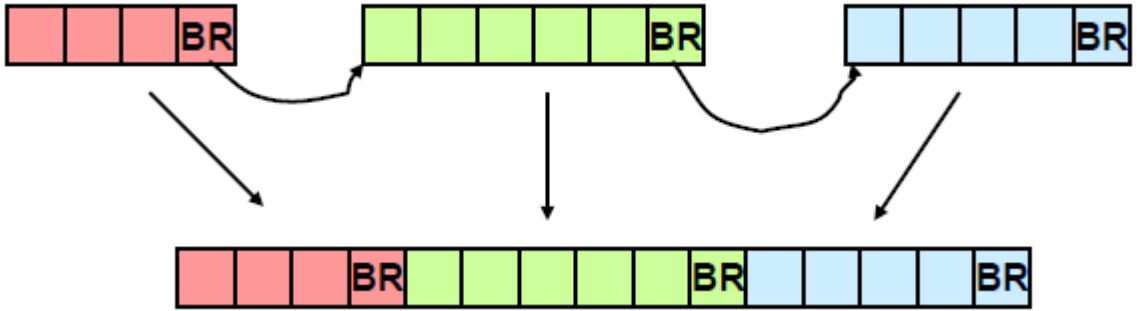
Nový způsob načítání instrukcí – „Trace Cache“

Problémy s načítáním instrukcí:

- Spojování instrukcí z různých bloků zvyšuje latenci.
- Při načítání skupin více než 4 instrukcí za takt je často ve skupině více skoků. Je třeba je predikovat v 1 taktu.

Řešení:

- Statické uspořádání instrukcí programu (v I-cache) nahradit **dynamickým pořadím** (v doplňkové Trace cache). Tj. sbalit několik nenavazujících základních bloků do jednoho souvislého bloku (stopy) v **Trace cache**.

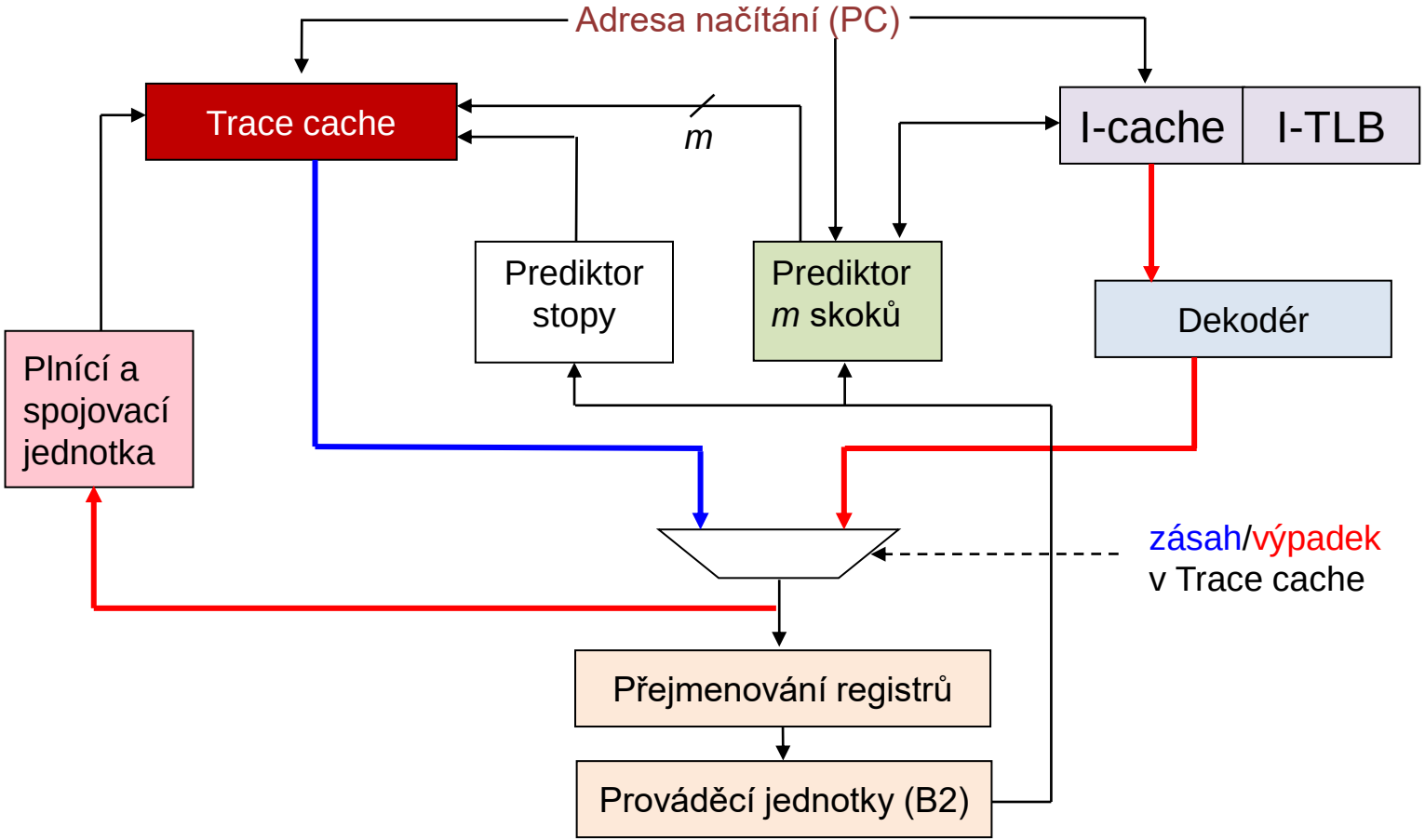


Paměť stop „Trace Cache“ TC

- **Základní bloky instrukcí (zakončené skokem) se načítají**
  - na začátku programu z I-cache pomocí jednotky načítání, přičemž se pomocí plnící jednotky spojují a zapisují i do TC
  - při zásahu v TC se načítá stopa z TC
  - při výpadku v TC postup jako na začátku.

- Přístup do TC probíhá paralelně s přístupy do I-cache a do BTB s použitím aktuální adresy načítání.
- Predikce několika skoků + adresa načítání se porovnává s reálnou sekvencí uloženou v TC.
- Podle výsledku porovnání máme zásah nebo výpadek v TC.

Načítání instrukcí s využitím Trace cache



Stopy v Trace cache

- Jedno načtení stopy dodá hned několik základních bloků.
- **Délka stopy v TC je omezena**
  - jistým max. počtem instrukcí (např. $n = 16$)
  - počtem skoků predikovatelných v 1 taktu m (např. $m = 3$)
  - nepřímým skokem, návratem nebo výjimkou.

- Blok paměti **trace cache** tak uchovává segment dynamické instrukční stopy přes několik provedených větvení.
- Na rozdíl od I-cache, uchovává **trace cache logicky navazující instrukce ve fyzicky navazujících paměťových místech.**

Zásah a výpadek v TC

- **Zásah v trace cache nastává když**
  1. adresa načítání se shoduje s tagem stopy
  2. a současně predikce skoků se shodují s flagy skoků (tj. s tím jak již byly skoky reálně ne/provedeny)
- **Při zásahu v TC** se celá stopa instrukcí nemusí dekódovat, cache instrukcí se neúčastní.
- **Při výpadku v TC**
  - se načítání provádí z I-cache do plnící jednotky, jeden základní segment za druhým.
  - Plnění skončí po *n* instrukcích nebo když se detekoval *m*-tý skok, nepřímý skok či návrat.
  - Jsou generovány postupové adresy stopy, flagy a masky skoků a obsah plnící jednotky je přepsán do TC.

PŘEDNAČÍTÁNÍ DAT

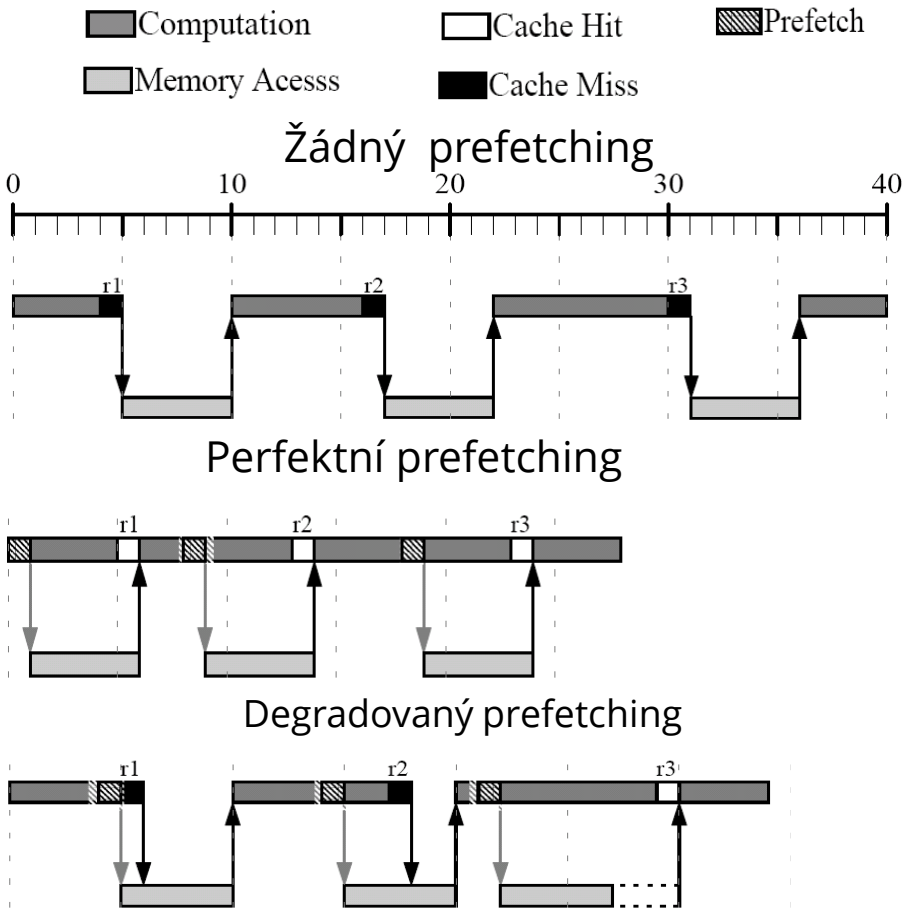
Přednačítání dat

- Podobně jako u instrukcí, než aby se čekalo na vyřízení výpadku (tj. načtení dat z paměti do cache), přednačítání provede načtení dopředu, čímž se přístup do paměti překryje s výpočtem.
- Může být implementováno v **HW nebo v SW**
- **Otázky:**
  - **kdy** je přednačtení iniciováno
  - **kam** jsou data přednačtena
    - do které úrovně cache, prefetch buferu nebo i registru
  - **jaká** je velikost dat u přednačítání:
    - bloky z paměti do D-cache
    - bloky s rozestupem (HW), slova spekulativně (SW)

Načasování přednačítání

- Aby bylo užitečné, musí být **just-in-time**.
  - je-li příliš brzy, může být předem načtený blok ještě před použitím vyměněn nebo modifikován jinou CPU; (tj. neužitečné přednačítání)
  - je-li příliš pozdě, musí se na data nějakou, i když kratší dobu, čekat.

- **Špatné načasování** přednačítání má vliv jen na výkonnost, ale ne na správnost
  - přednačtená data mohou v cache nahradit data, která ještě procesor užívá → výpadek navíc
  - snaha využít všechna slova z bloku.



Přednačítání dat iniciované SW

- Mikroprocesory mají pro **přednačítání strojové instrukce**
  - mohou být vloženy kompilátorem nebo
  - programátorem při optimalizaci
- Instrukce přednačítání nemůže vyvolat výjimku (přednačtení nepokračuje).
- Přednačítání může předbíhat regulérní rozpracované operace v cache.
- Přednačítání má také svou režii (přídavné instrukce, použití jistých registrů pro adresy, zabírá část propustnosti paměti), která znamená vyšší provoz mezi CPU a pamětí.
- U přednačítání se testuje, zda daný blok v cache už je nebo ne.
  Provede se jen když tam není.

- **Nejčastěji** se používá u smyček počítajících **s velkými poli** (vědecké výpočty s predikovatelnými vzory přístupů do polí).
- Existuje též přednačítání ukazatelů (průchod stromy, seznamy).

- SW přednačítá data nikoliv slepě, ale jen ta data, která budou velmi pravděpodobně potřeba.
- Vyžaduje neblokující cache.
- Může se aplikovat mezi libovolnými úrovněmi paměťové hierarchie. My si ukážeme jen do L1C.

Bez přednačítání

```
for (i = 0; i < N; i++)
{
   sum += a[i] * b[i];
}
```

- Předpoklad:
  - v bloku cache jsou 4 prvky vektorů
  - kód generuje 2 výpadky na 4 iterace

Naivní přednačítání

```
for (i = 0; i < N; i++)
{
   fetch(&a[i+1]);
   fetch(&b[i+1]);
   sum += a[i] * b[i];
}
```

- V bloku cache jsou 4 prvky a[0], a[1], a[2], a[3]
- přednačítání každého zvlášť je proto zbytečné

SW přednačítání dat s rozbalenou smyčkou, první verze

```
for (i = 0; i < N; i+=4) {
  fetch(&a[i+4]);
  fetch(&b[i+4]);
  sum += a[i] * b[i];
  sum += a[i+1] * b[i+1];
  sum += a[i+2] * b[i+2];
  sum += a[i+3] * b[i+3];
}
```

- při první iteraci výpadky
- zbytečné přednačítání při poslední iteraci i = N - 4
- rozbalení tolikrát, kolik je slov v bloku (4).

```
      i = 0
fetch(&a[4]);
fetch(&b[4]);
sum += a[0]*b[0];
sum += a[1]*b[1];
sum += a[2]*b[2];
sum += a[3]*b[3];
```

```
i = 4
fetch(&a[8]);
fetch(&b[8]);
sum += a[4]*b[4];
sum += a[5]*b[5];
sum += a[6]*b[6];
sum += a[7]*b[7];
```

```
…  i = N- 4
   fetch(&a[N]);
   fetch(&b[N]);
   sum += a[N-4]*b[N-4];
   sum += a[N-3]*b[N-3];
   sum += a[N-2]*b[N-2];
   sum += a[N-1]*b[N-1];
```

SW přednačítání jednu iteraci dopředu

```
fetch (&sum);
fetch (&a[0]);
fetch (&b[0]);

for (i = 0; i < N-4; i+=4)
{
  fetch (&a[i+4]);
  fetch (&b[i+4]);
  sum += a[i]*b[i];
  sum += a[i+1]*b[i+1];
  sum += a[i+2]*b[i+2];
  sum += a[i+3]*b[i+3];
}

for (i = N-4; i < N; i++)
  sum = sum + a[i]*b[i];
```

- Instrukce prefetch nemají registrový operand
- Blok obsahující slovo s uvedenou adresou je načten z té úrovně, kde právě je, do D-L1C, případně do cache zadané úrovně
- Implicitní předpoklad, že 1 iterace (4 x výpočet sum) stačí překrýt latenci paměti
- Poslední 4 iterace pracují s daty přednačtenými již dříve

Přednačítání dat do cache iniciované HW

- Programátor ani kompilátor nemusí zasahovat.
- Není nutno modifikovat programy, žádné instrukce navíc.
- Může využít informací z běhu programu ke zefektivnění přednačítání.
- Generuje ale více zbytečných přednačtení než SW způsob (znečisťuje cache).

- **Varianty:**
  - sekvenční přednačítání (s rozestupem 1)
  - přednačítání s libovolným rozestupem

- **Neumí nepravidelné vzory přístupů k datům**

Sekvenční přednačítání dat iniciované HW

- Po načtení bloku *i* se přednačte sousední blok *i*+1 (OBL, One-Block Lookahead). Zvládne smyčky s `i++`, ale už né `i--`.
- Jestliže blok již v cache je, přednačtení není iniciováno.
- **Existují 2 implementace**

1. Přednačtení při výpadku:
   - Při výpadku v bloku *i* se načte blok *i* a přednačte *i*+1.
   - Jak se to liší od 2x většího bloku cache? – Ten by se musel přesunout nebo zneplatnit celý, větší možnost falešného sdílení.
2. Přednačtení s příznakem (jak se mě dotkneš, přednačti další):
   - při výpadku v bloku *i* se načte blok *i* (s příznakem 0) a přednačte se blok *i+1* (s příznakem 1)
   - při zásahu v bloku *i*+1:
     - s příznakem 1 (první zásah): změň příznak na 0 a přednačti blok *i+2* (s příznakem 1)
     - s příznakem 0 (druhý a další zásah): žádná akce

Porovnání obou přednačítání OBL

Přístupy do po sobě jdoucích bloků (vyžádaných, demand-fetched nebo přednačtených, prefetched)

1.

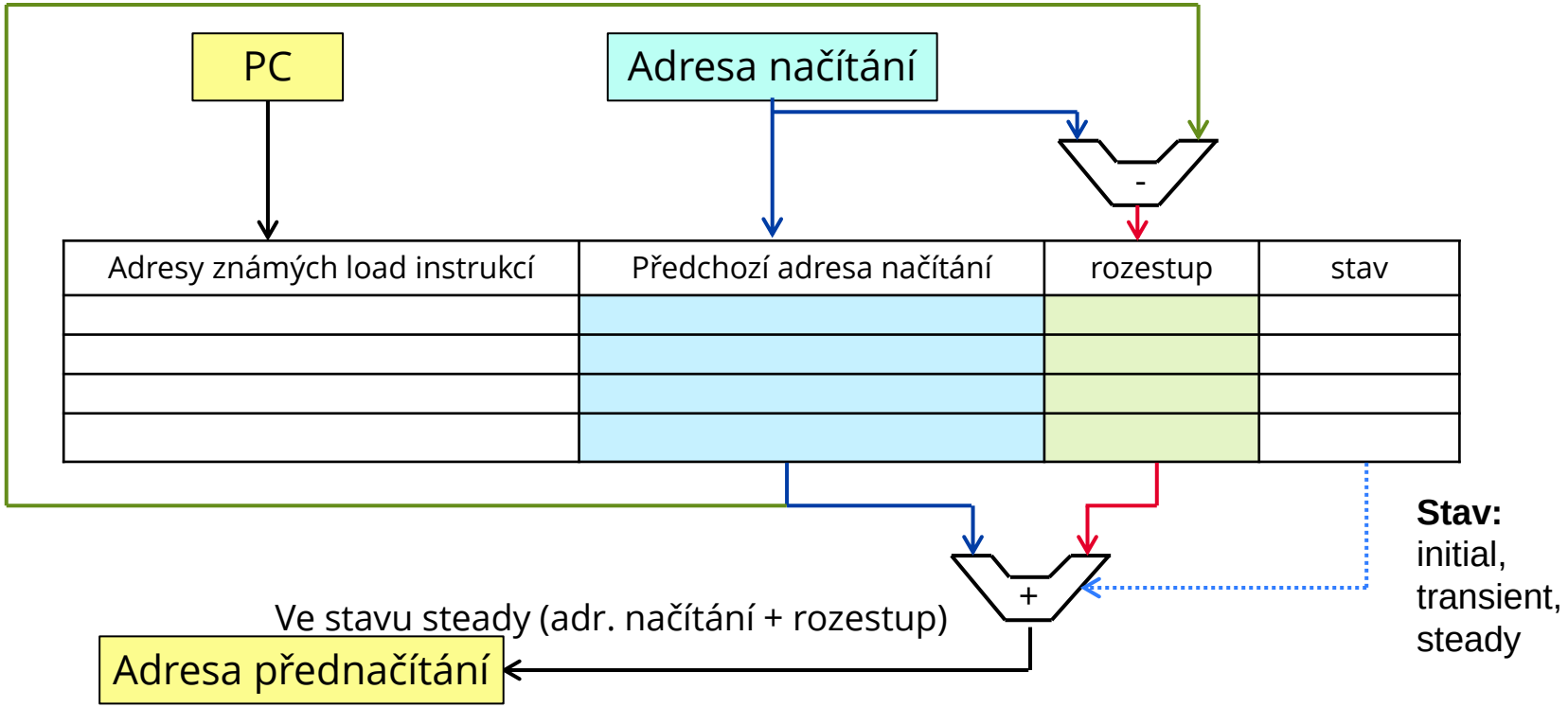
2.

Streamové přednačítání i pro data

- Je to analogie přednačítání instrukcí do instrukčního buferu.
- Fixní počet 4–8 bloků (např. mediální data) se sekvenčně načítá předem do FIFO stream buferu a ne do L1 D cache, aby se vyloučilo její znečistění, jelikož se data použijí jen 1x.
- Hledá se současně v cache i na čele stream buferu.
  - Když je nalezen blok v buferu, přesune se do cache, pointer na čelo se posune a nový blok se přednačte na konec buferu.
  - Při výpadku v cache a když blok není nalezen ani na čele buferu (konec sekvence) se bufer začne plnit daty od nové adresy výpadku.
- Výhoda stream buferu se ukáže jen když přístup k přednačítaným blokům je přísně sekvenční.

Přednačítání operandů s rozestupem (HW)

- Load History Table (LHT) = malá cache již provedených instrukcí L
- Rozestup se získá ze dvou L na téže adrese.



Pokračování příště