1. Architektura superskalárních procesorů, algoritmy pro zpracování instrukcí mimo pořadí, predikce skoků.

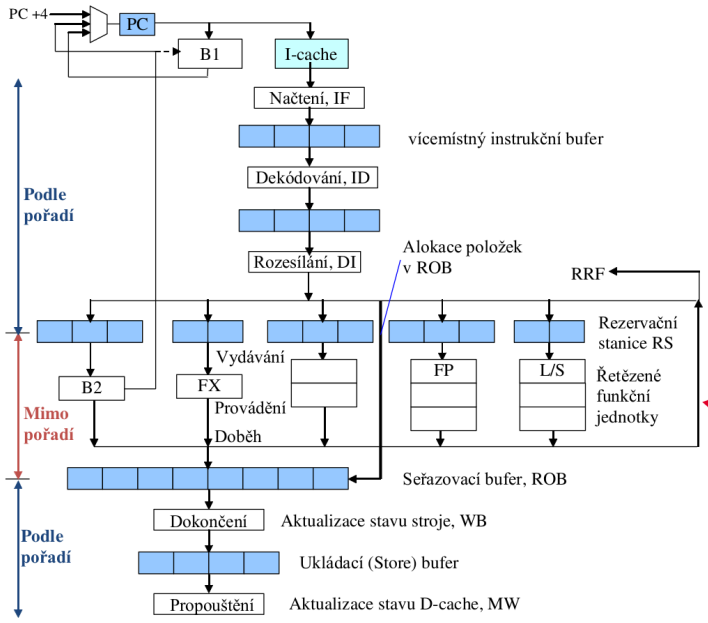
Vezmeme-li v úvahu standardní skalární procesor a jeho vykonávání, je patrný jasný rozdíl mezi využitím jednotlivých fází. Zatímco fáze Instruction Fetch a Instruction Decode se provádějí vždy pro každou instrukci, různé instrukce obvykle nevyužívají všechny pozdější fáze (Execute (EX), Memory access (MA), Write back (WB)). Při snaze o výkonnější návrh si uvědomíme, že funkční paralelismus mezi instrukcemi - některé instrukce používají FPU, jiné ALU nebo jen načítají z paměti - je zdrojem, který je možné využít. Architektura superskalárních procesorů se proto skládá z následujících dvou částí, které lze chápat jako stupně skalárního procesoru seskupené podle toho, zda je každá instrukce využívá, nebo ne:

- ***Front-end*** (všechny instrukce musí projít)
    - odpovídá fázím Instruction Fetch (IF) a Instruction Decode (ID) v (sub)skalárním procesoru
    - načítá a dekóduje několik instrukcí najednou
    - dva typy podle pořadí, v jakém instrukce opouštějí front-end:
        - ***In-order*** - instrukce opouštějí frontend podle pořadí programu
        - ***Out-of-order*** - instrukce opouštějí frontend v pořadí, které nemusí respektovat pořadí programu. Tento přístup obvykle umožňuje vyšší výkon a lepší využití HW, ale přináší nové problémy, se kterými je třeba se vypořádat (WAW, WAR konflikty, viz otázka 2.)
    - asi nejdůležitějším parametrem frontendu je počet instrukcí, které lze dekódovat za jeden takt - pokud je maximální počet instrukcí, které mohou opustit front-end **m**, říkáme, že jedná o **m-cestný superskalár**
- ***Back-end*** (různé instrukce využívají různé části)
    - odpovídá fázím Execute (EX), Memory access (MA), Write back (WB)
    - provádí paralelně několik dekódovaných instrukcí, ukládá jejich výsledky

Rysy superskalárních procesorů (vlastnosti označené OoO jsou specifické pro out-of-order):

- *Paralelní řetězené linky* - Časový i prostorový paralelismus (paralelní načítání, dekódování,vydávání instrukcí do FJ, jejich paralelní provádění a dokončování),
- *(OoO) Přejmenování registrů v HW* - Odstraní konflikty WAR a WAW
- *(OoO) Dynamické plánování instrukcí out-of-order* - Instrukce, včetně přístupů do paměti, jsou zpracovávány v jiném pořadí oproti pořadí v programu - instrukce jsou spuštěny jakmile jsou operandy připraveny,
- (OoO) *Seřazovací paměť* - Stupeň WB pomocí ní zajišťuje ukládání výsledků v pořadí určeném zdrojovým kódem.
- (OoO) *Spekulativní zpracování instrukcí* - Spekulace výsledků podmíněných skoků; spekulace, že dopředu načtená data nezmění.

Generická OOO superskalární architektura



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

RRF – registry pro přejmenování (cca 256)

RS – zde individuální rezervační stanice

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

Algoritmy pro zpracování instrukcí mimo pořadí

Provádění instrukcí mimo pořadí zavádí nepravé konflikty – **write after read (WAR)**, **write after write (WAW)**. Jediný skutečný konflikt **read after write (RAW)** znamená, že instrukce fyzicky potřebuje výsledek předchozí instrukce a pro okamžité získání výsledku nelze nic udělat než počkat na dokončení předchozí instrukce.

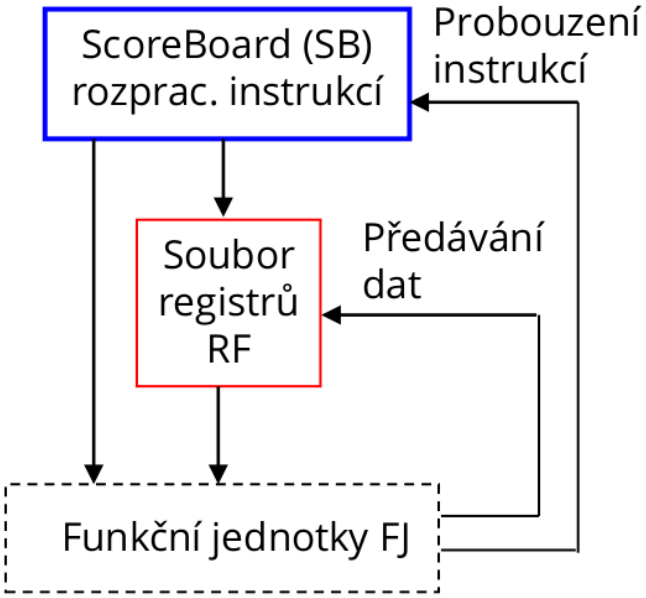
Falešné konflikty jsou ve své podstatě konflikty jmen. Například pokud n-tá instrukce zapisuje do registru RBX a (n+k)-tá instrukce také zapisuje do registru RBX, a tedy tato dvojice instrukcí tvoří konflikt WAW. Pokud však změníme cíl (n+k)-té instrukce na nějaký jiný registr (a odpovídajícím způsobem upravíme všechny následující instrukce tak, aby používaly nový cílový registr), konflikt WAW zmizí. V ideálním případě bychom chtěli, aby hardware automaticky prováděl přejmenování. Hardware schopný automatického přejmenování je však složitý a drahý, což znamená možnost kompromisu mezi tím, jak moc je procesor mimo pořadí, a tím, jak složitý je základní algoritmus plánování instrukcí.

Existují dva hlavní algoritmy plánování instrukcí:

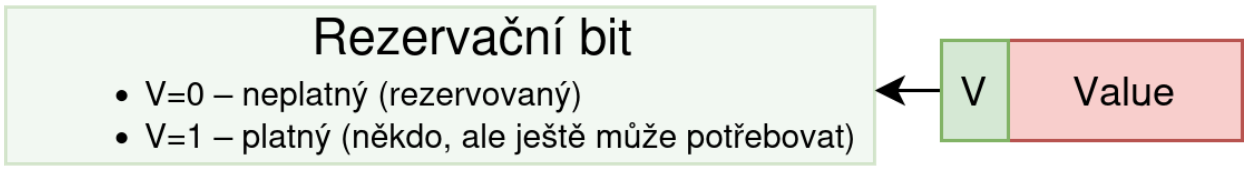
- ***ScoreBoarding (Thorntonův algoritmus, 1964)***
- **Rezervační stanice (Tomasulův algoritmus, 1967)**
    - Rezervační stanice (bufery) umožňují odložit čekající instrukce a pracovat dopředu na dalších – tím řeší RAW. Konflikty WAW a WAR se řeší přejmenováním.

ScoreBoarding

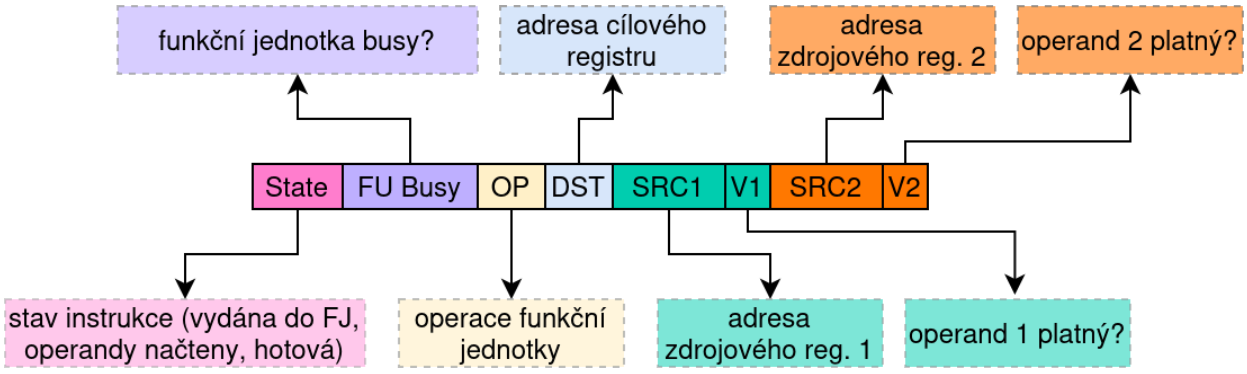
Registruje všechny konflikty (RAW, WAW, WAR) v tabulce rozpracovaných instrukcí (= ScoreBoard) a udržuje jejich skóre. SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB. Přejmenování registrů neprobíhá, **konflikty se řeší čekáním.**



Formát registrů v RF (register file):



Formát jedné položky ScoreBoard:



Algoritmus scoreboard (SB):

1. **Vydání nové instrukce – Fáze ID**

1.1. **Rezervace cílového registru v RF** - pokud je valid bit cílového registru 1, lze registr rezervovat nastavením hodnoty platného bitu na 0. Pokud je valid bit již nastaven na 0, je registr rezervován nějakou jinou instrukcí, která do něj vygeneruje svůj výsledek. Frontend proto musí počkat, dokud tato jiná instrukce neskončí.

1.2. Vydání instrukce

1.2.1. Jsou-li oba zdrojové operandy platné (V1 a V2 = 1), tak odešli op-kód instrukce do FJ, odešli adresy SRC1 a SRC2 do RF a odtud jejich hodnoty do FJ, vynuluj příznaky V1 a V2 ve SB a změň stav instrukce (operandy načteny).

1.2.2. Pokud není některý operand platný, čekej

2. Vykonání instrukce – Fáze EX - Výsledek a adresa dst registru se objeví na výstupu FJ

3. Zápis výsledků do registru – Fáze WB

3.1. Dokud se shoduje dst výsledku se src1 nebo src2 v nějaké instrukci v SB s bitem V1 = 1 nebo V2 = 1 („platný a ještě nepoužitý“) - **čekej**. Stávající obsah RF(dst) totiž ještě nepřečetla nějaká čekající instrukce

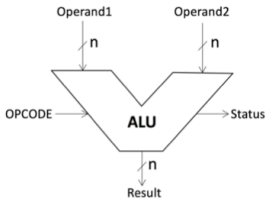
3.2. Jakmile jsou relevantní bity V1 a V2 v SB vynulovány - zapíšeme výsledek a V = 1 do RF(dst). Prohledáme SB a změníme bit V1 nebo V2 na 1 ve všech položkách SB, které čekali na výsledek (src1 | src2 = dst).

4. Jakmile je instrukce dokončena, její záznam je smazán z tabulky score

Registry

Name	Valid	Value
R1		
R2		
R3		
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

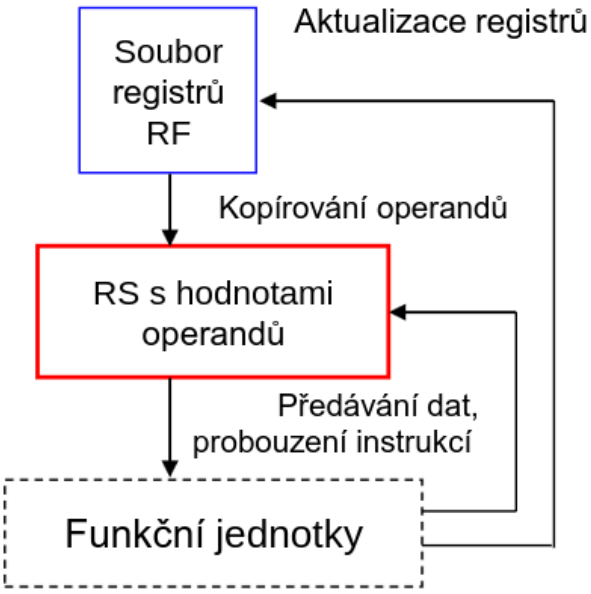


ScoreBoard

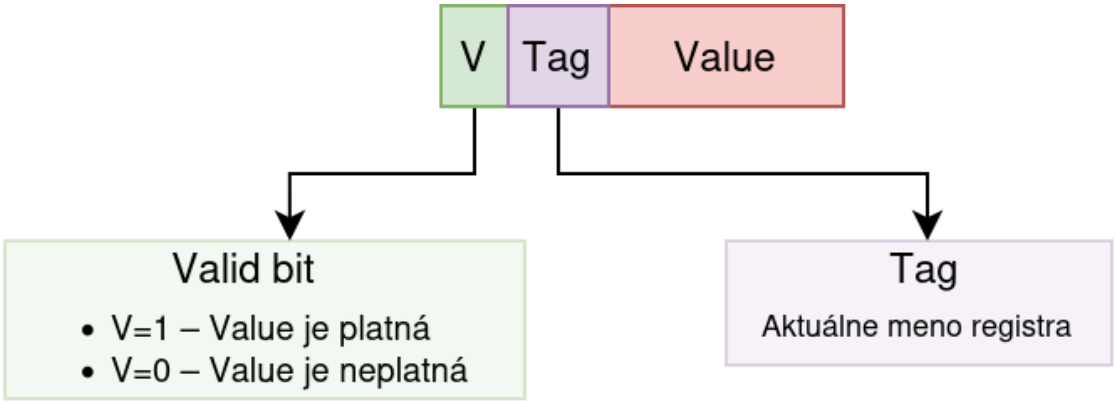
Dst	S1	V1	S2	V2

Rezervační stanice

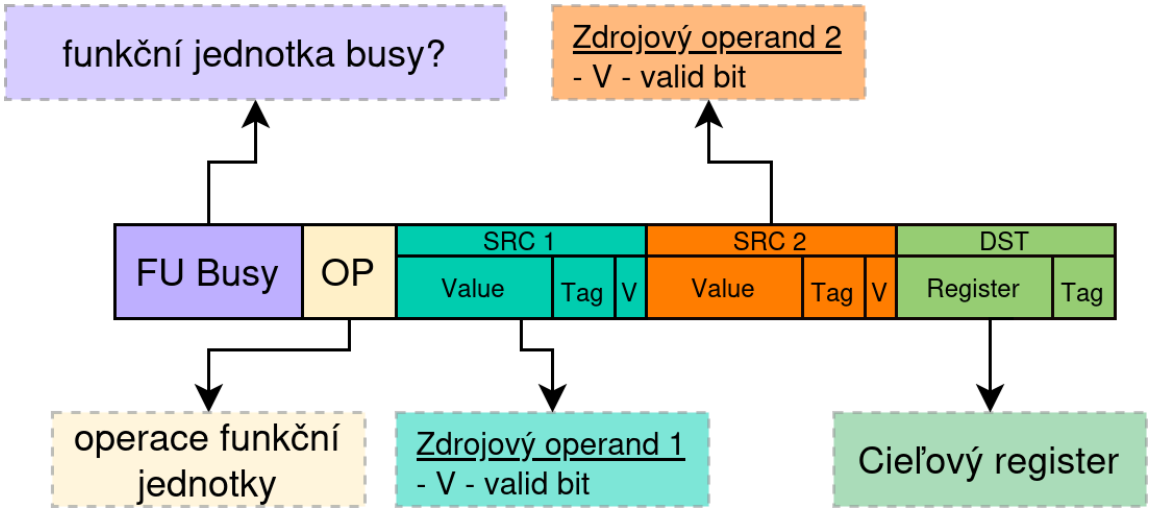
- převážně používaný algoritmus v moderních procesorech, nepravé konflikty se řeší přejmenováním, pravý konflikt RAW se řeší odložením čekající instrukce.



Formát registrů v RF (register field):

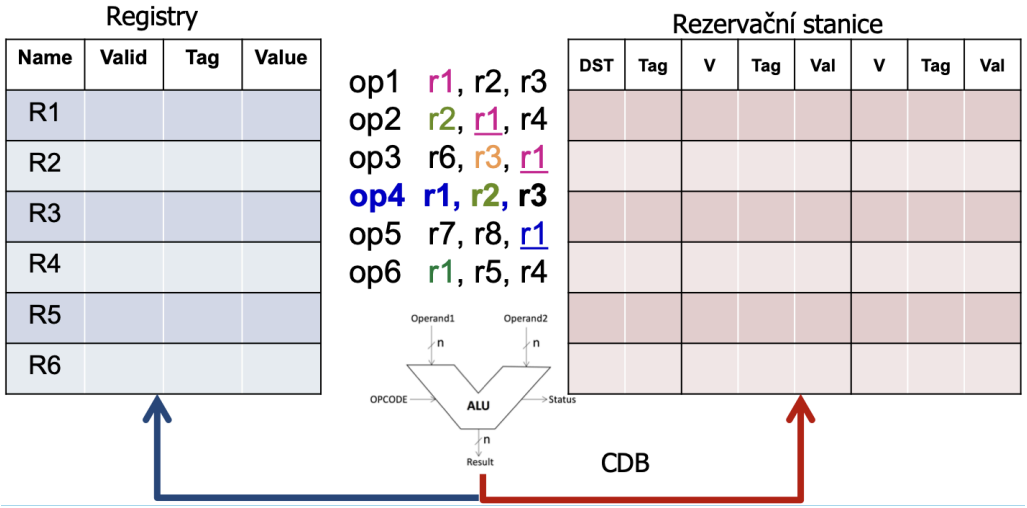


Formát jedné položky rezervační stanice:



Tomasulo algoritmus dynamického plánování

1. Instrukce opouští frontend
   1.1. cílovému registru je přiřazen nově vygenerovaný tag, který je zapsán do RF a do přiřazené položky RS
   1.2. bit platnosti cílového registru je v RF nastaven na 0
   1.3. platné hodnoty (V = 1) zdrojových operandů z RF včetně tagu se načtou do RS současně s instrukcí (valid bit v RS je 1)
   1.4. zatím ještě neplatné operandy (V = 0) - načte se pouze tag a valid bit = 0.
2. Instrukce opouští RS
   2.1. Pokud src operandy instrukce v RS ready (V1 & V2 = 1) & FJ volná, odešle se instrukce z RS (včetně dat, dst addr, tag) do FJ
3. FJ vytvoří výsledek
   3.1. FJ rozhlásí hodnotu výsledku na CDB (common data bus) (pokud je volný) spolu s adresou dst i tagem dst registru
4. RS monitoruje CDB
   4.1. Instrukce čekající v RS monitorují CDB, porovnávají tagy operandů s tagem na CDB. V případě shody zachytí hodnotu z CDB do příslušného políčka RS, případně do store bufferu (instrukce store) (eliminace WAR, WAW);
5. RF monitoruje CDB
   5.1. Při shodě tagů na CDB i v reg. RF(dst) se hodnota zapíše paralelně i do reg. RF(dst) a nastaví se v něm V = 1. Při neshodě tagů (tj. RF(dst) byl již znovu přejmenován) se hodnota v RF(dst) vůbec neobjeví, je zachycena pouze v polích RS, které tak slouží jako tagem přejmenované registry.



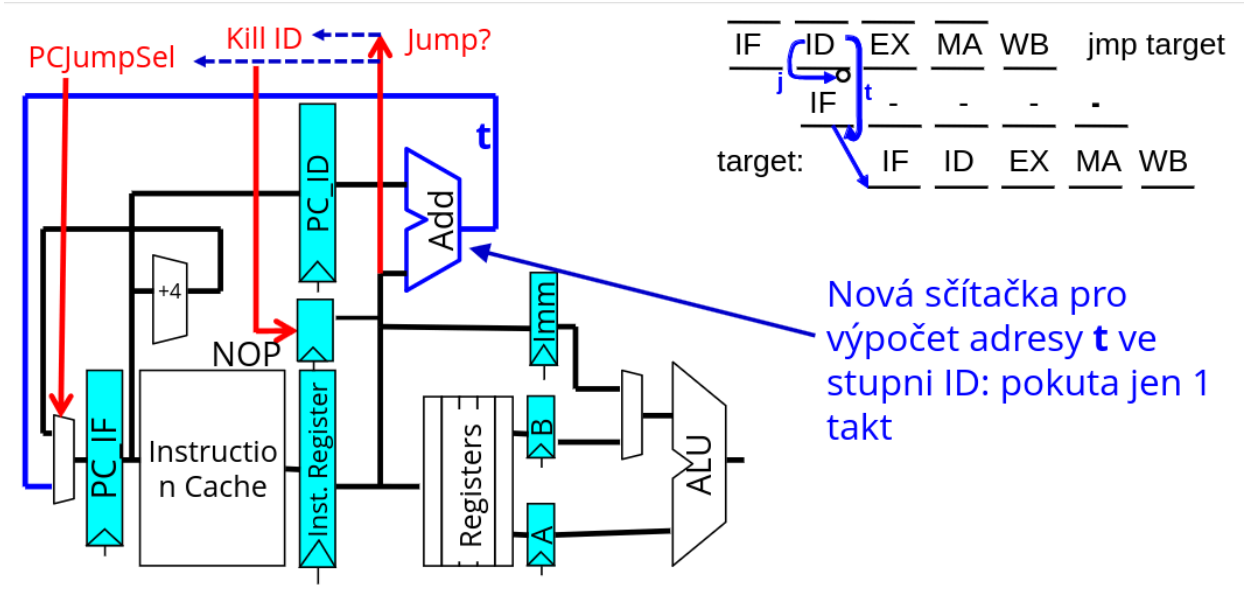
Řešení konfliktů v Tomasulo algoritmu:

- RAW - Instrukční závislosti RAW se detekují a řeší čekáním v RS
- WAR, WAW - registry viditelné programátorovi (architekturní, ARF) jsou mapovány (**přejmenovány** příznakem) na položky RS (15–60 celkem)

Predikce skoků

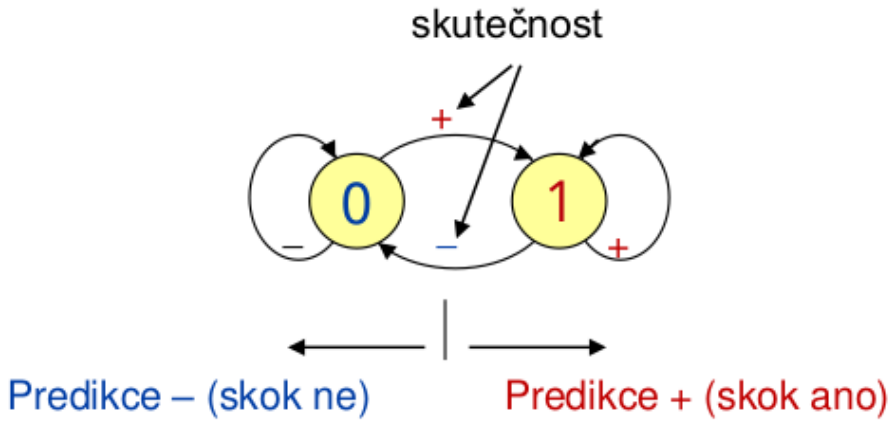
Proč jsou skoky problematické? Instrukce skoku v programu způsobují zastavení pipeline, protože se neví, jaká instrukce má následovat po instrukci skoku, dokud není instrukce skoku provedena. Délku zdržení pipeline lze zkrátit zavedením další ALU do fáze ID, což umožní vypočítat adresu bez čekání na to, až instrukce skoku projde fázemi EX a WB. Podobným trikem lze snížit cenu podmíněného skoku - pokračujeme normální inkrementací programového čítače, jako by podmínka skoku byla záporná (pevná záporná predikce).

Optimalizace (zkrácení) zpoždění pipeline způsobeného (podmíněnými) skoky:



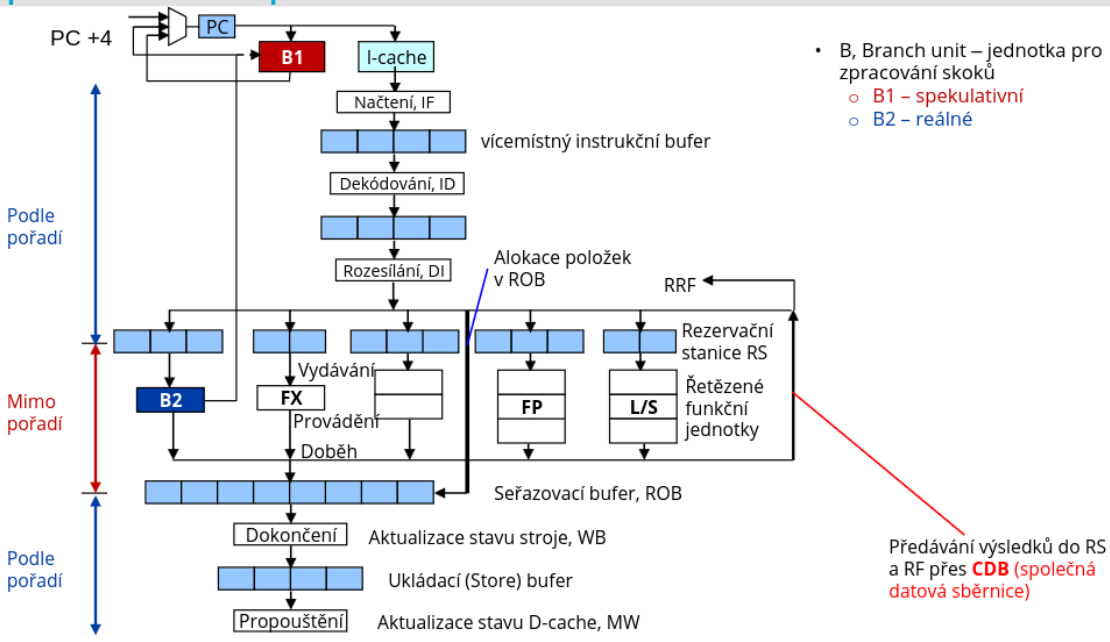
Dále lze zlepšit výkonnost tipováním účinku instrukce skoku. V případě, že úspěšně odhadneme výsledek, zcela se vyhneme zpoždění pipeline:

- ***Proč to funguje?*** Skoky jsou vysoce předvídatelné - napr. návrat z podprogramu, skoky uzavírající smyčky, skoky do podprogramu.
- ***Co potřebujeme predikovat?***
  - Predikce podmínky skoku – jen u podmíněných skoků
    - adaptivní prediktory - berou v úvahu pouze jeden skok izolovaně a ne to, jak jsme k němu dospěli (dynamický kontext předchozích skoků):
      - 1 bitový prediktor



    - 2 bitový prediktor spolupracující s BHT (Branch History Table - tabulka obsahující stav automatu)
    - 2 bitový prediktor spolupracující s PHT (Pattern History Table - 2D struktura o velikosti 2^k čítačů - BHT - schopná učit se složité skokové vzory)
  - Korelační prediktory - berou v úvahu dynamický kontext předchozích skoků. Používají PHT, ale namísto lokální historie skoku používají globální historii.
- Predikce cílové adresy
  - Branch Target Address Cache (BTAC) - obsahuje předchozí cílové adresy skoků. Pro návrat z podprogramů se používá jiný prediktor se strukturou podobnou zásobníku - Return Stack Buffer (RSB) (prediktor návratových adres).

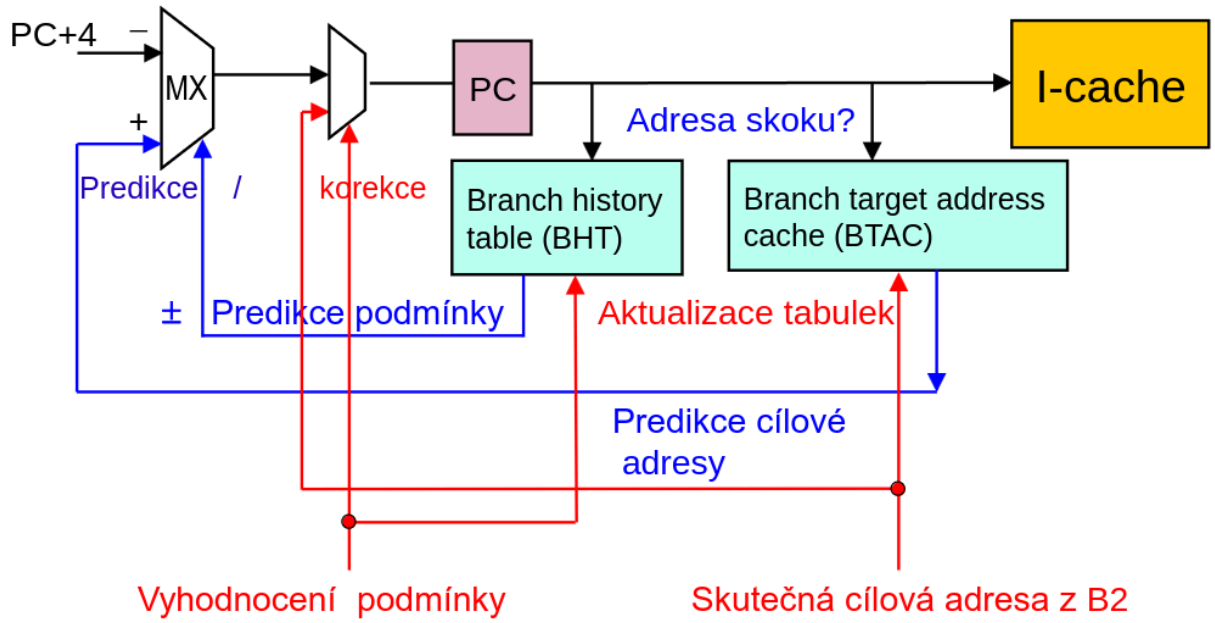
Superskalární procesor



- ***Kdy potřebujeme predikovat?***

- Adresa - Hned ve fázi IF, jakmile se rozpozná že PC ukazuje na skokovou instrukci (se část adresy skoku použije jako tag a index do BTAC
- Skákat/neskákat - pouze pro podmíněné skoky, obdobně jako při adrese (tj. fáze fetch) v BHT

Predikce podmínky a cílové adresy skoku (B1)



- ***Co když se předpověď nezdařila?***
  - Všechny instrukce, které byly provedeny na základě spekulace (jednotka B1 s BHT a BTAC), čekají v seřaďovací paměti, dokud se nepotvrdí, že je spekulace správná (ve fázi EX v jednotce B2). Pokud byla nesprávná, výsledky těchto instrukcí se zahodí a provede se správná větev a aktualizují se BHT a BTAC.