B35APO: Architektury počítačů

Lekce 06. Superskalární architektura a prediktory skoků

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

26. března, 2023

Obsah

Cíl dnešní přednášky

- Podívat se na další možné zrychlení procesoru, které navazuje na pipelining – superskalární architekturu
- Predikce skoků jako velmi důležitá vlastnost superskalárních procesorů
- Obě tyto techniky se využívají jak v RISC-V procesorech, tak i ve všech současných procesorech

Opakování - kvíz

V kterém případě instrukce obsahují datový hazard?

a)

```
addi t0, s1, 4
add  t1, s1, s0
add  s1, s2, x0
```

b)

```
addi t0, s1, 4
add  t1, s2, s3
add  t2, t0, t1
```

A ani v jednom sloupci

B hazard je pouze v případě a)

C hazard je pouze v případě b)

D hazard je v případě a) i b)

Opakování - kvíz

Jak lze vyřešit následující datový hazard?

```
lw   s2, 10(s0)
addi s1, s2, -1
```

A tento hazard nelze vyřešit, musí ho řešit překladač nebo programátor

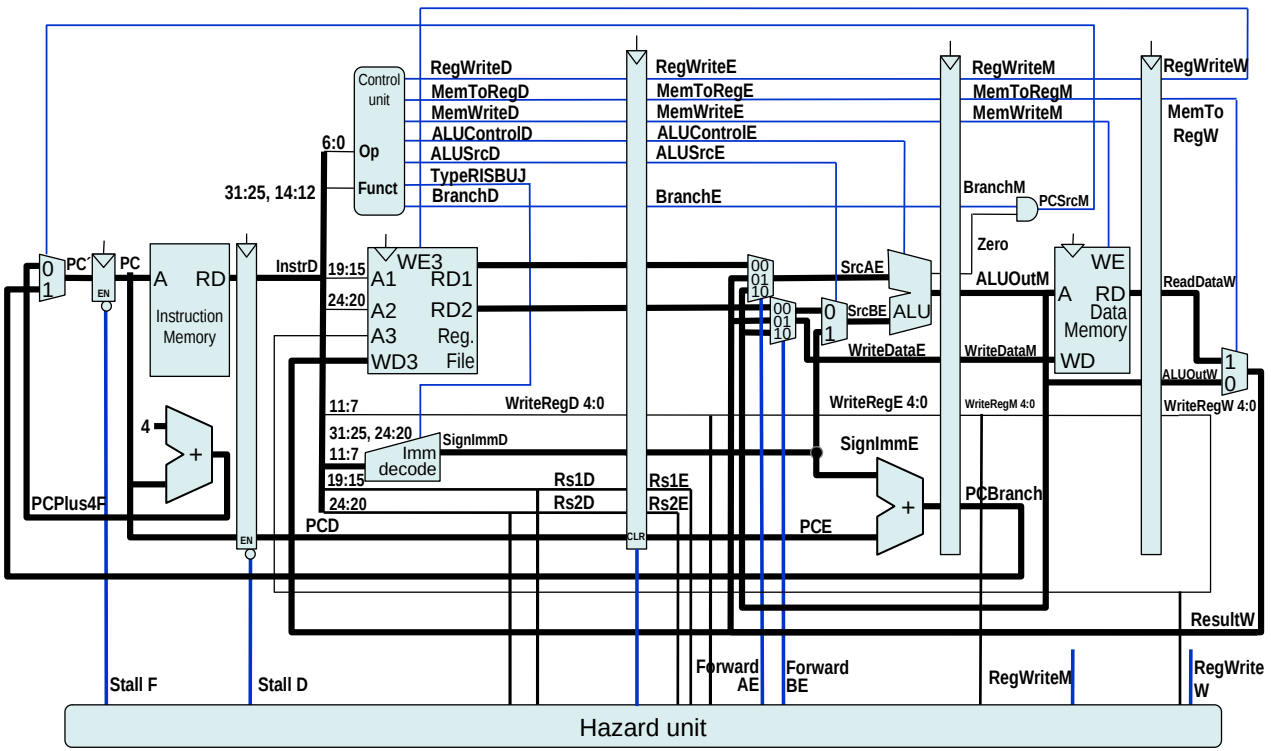
B lze ho vyřešit pomocí data forwarding

C musí se řešit pouze pomocí stall

D lze ho vyřešit kombinací stall a data forwarding

Procesor s pipeline (z přednášky 5)

Jak může Hazard Unit rozhodnout o datovém hazardu, když to dělá problémy studentům? Jednoduše:



- Pokud je RegWriteM==1, MemToRegM==0 a WriteRegM==RsE1 nebo RsE2 pak nastav ForwardAE na 2 nebo ForwardBE na 2
- Pokud je RegWriteW==1, MemToRegW==0 a WriteRegW==RsE1 nebo RsE2 pak nastav ForwardAE na 1 nebo ForwardBE na 1

- Pokud MemToRegM==1 a WriteRegW==Rs1E nebo Rs2E pak nastav pozastavení - STALL

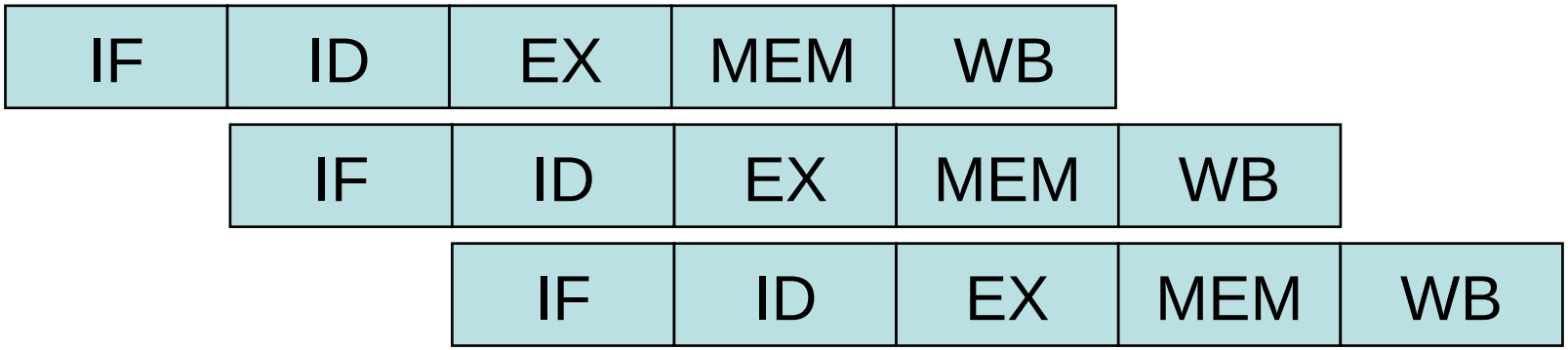
Paralelismus na úrovni instrukcí

Paralelní zpracování na úrovni instrukcí (Instruction Level Parallelism, ILP)

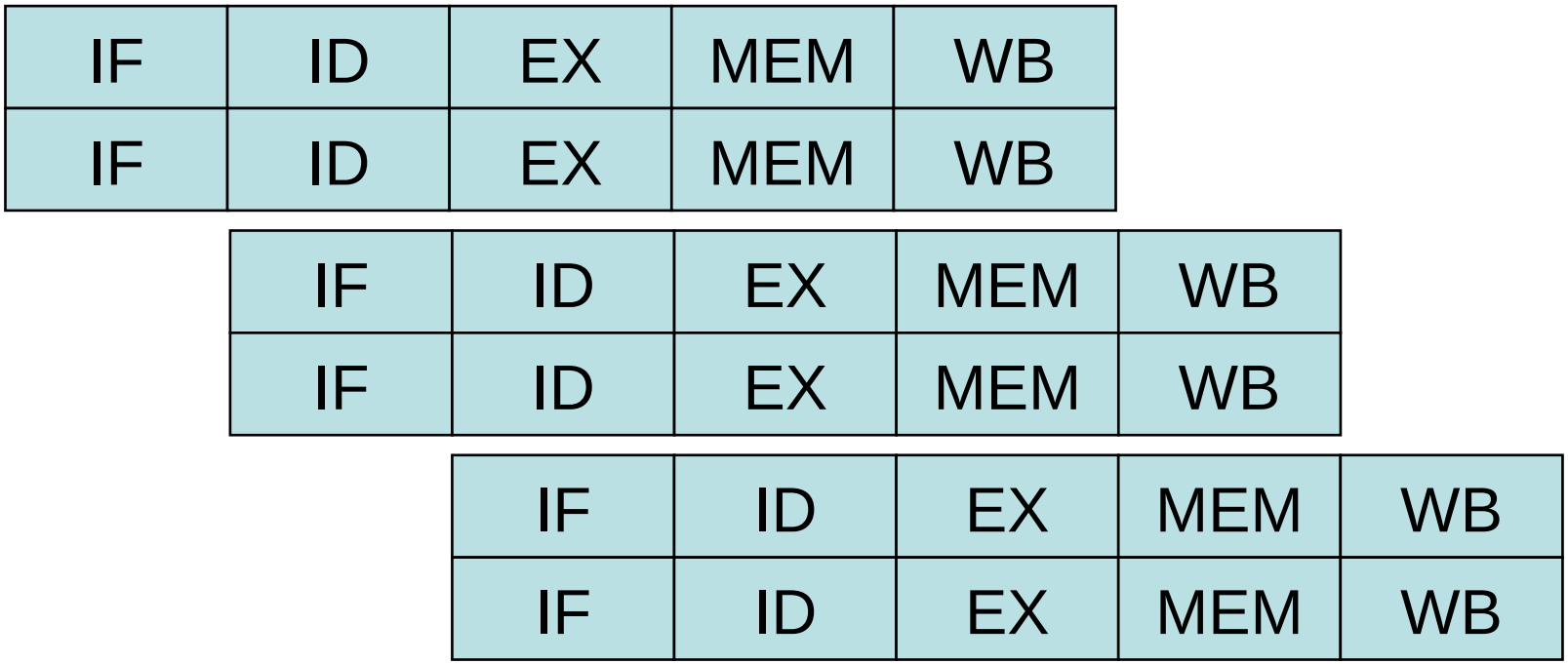
- Pipelining – paralelně se zpracovávají různé fáze různých instrukcí
- Superpipelining – za superpipelining se označuje pipelining s více než 10 kroky. Pomalejší fáze pipeliningu se rozdělí na více částí, tím dojde k možnosti zvýšit frekvenci procesoru a tím i jeho výkon.
- Superskalární procesor – paralelně se zpracovávají stejné fáze různých instrukcí
  - můžeme mít více jednotek ALU a provádět paralelně fáze EX více různých instrukcí
  - ve fázi fetch můžeme paralelně načítat více instrukcí jdoucích po sobě, tedy např. instrukce z adresy PC, PC+4, PC+8 a PC+12
  - jednotlivé fáze instrukcí jsou navíc zřetězené, takže po paralelním načtení 4 instrukcí, rovnou načítáme další 4 instrukce

Paralelismus na úrovni instrukcí

Zřetězený procesor – piplened procesor



Superskalární procesor – super scalar procesor



Superskalární procesory

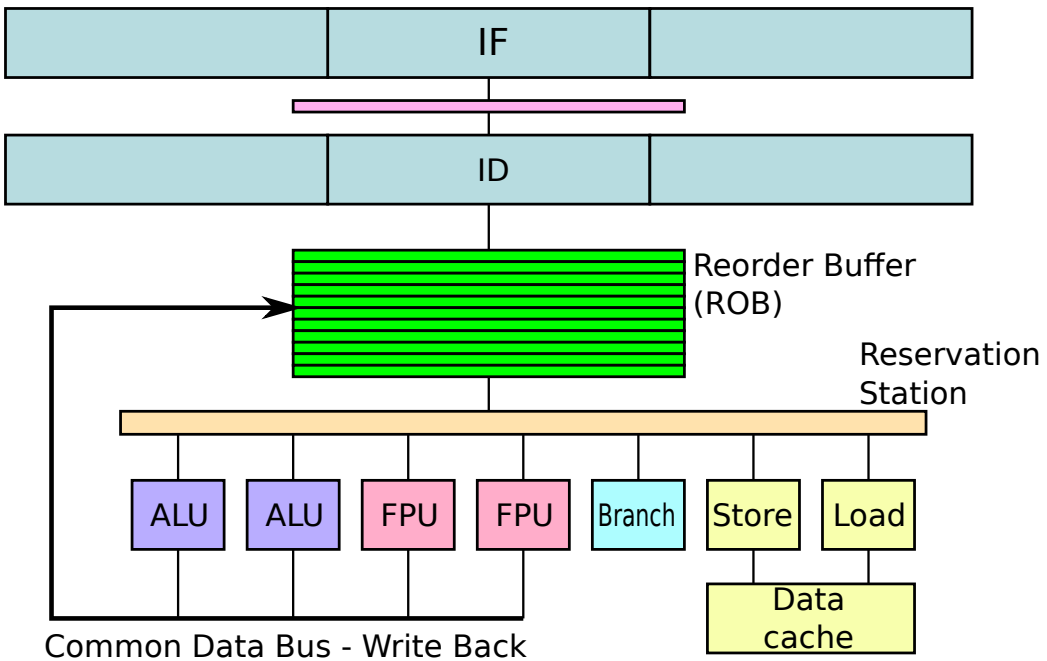
- Superskalární procesory mají IPC (Instruction Per Clock) větší než 1.
  - Normální i pipelined procesor mají neustále IPC=1
- Počet paralelních větví se nazývá šířka zřetězení (instruction pipeline width)
- Existují dvě základní varianty:
  - **Statická** superskalární architektura – paralelně mohou být spuštěny jen instrukce jdoucí v programu za sebou.
    - Pokud na sobě instrukce závisí vede to k pozastavení procesoru (stall).
  - **Dynamická** superskalární architektura – paralelně mohou být spuštěny libovolné instrukce, které jsou přiravené k vykonávání.
    - Umožňuje předbíhání instrukcí tzv. out-of-order execution.
    - Lépe využívá HW prostředky procesoru.

Superskalární procesory

- Jednotlivé paralelní větve mohou být **unifikované** – tedy všechny větve jsou stejné a mohou provádět všechny typy operací
  - V praxi by to byl zbytečně složitý procesor – nepoužívá se
- Jednotlivé paralelní větve jsou **specializované** – každá větev umí jen některé instrukce:
  - instrukce pracující pouze s registry – výpočty, porovnání
  - instrukce pracující s pamětí – načítání/ukládání dat z/do paměti
  - instrukce skoku – instrukce měnící PC

Obsah

Superskalární architektura



- Základem architektury je ReOrder Buffer (ROB), který pomocí přejmenování registrů umí zlepšit paralelizaci instrukcí.
- Reservation Station rozšiřuje možnosti ukládání operandů pro operace a organizuje jejich vykonávání
- Common Data Bus zajišťuje zapsání vypočtených hodnot do skutečných registrů i pro přejmenované registry.

Datové hazardy v superskalární architektuře

Pro instrukce pracující pouze s registry:

- je jasné, že není možné paralelně vykonávat všechny instrukce
- pomocí skryté sady registrů je možné paralelizaci zvýšit.

```
1: slli t1, s1, 4
2: add  t0, t1, s2
3: addi s2, t0, 8
4: mult t1, s0, s0
5: addi t3, t1, 100
```

Tento program lze paralelizovat s použitím skryté sady registrů pro přejmenování:

```
1: slli RN0, s1, 4
2: add  RN1, RN0, s2
3: addi RN2, RN1, 8
```

```
4: mult RN3, s0, s0
5: addi RN4, RN3, 100
```

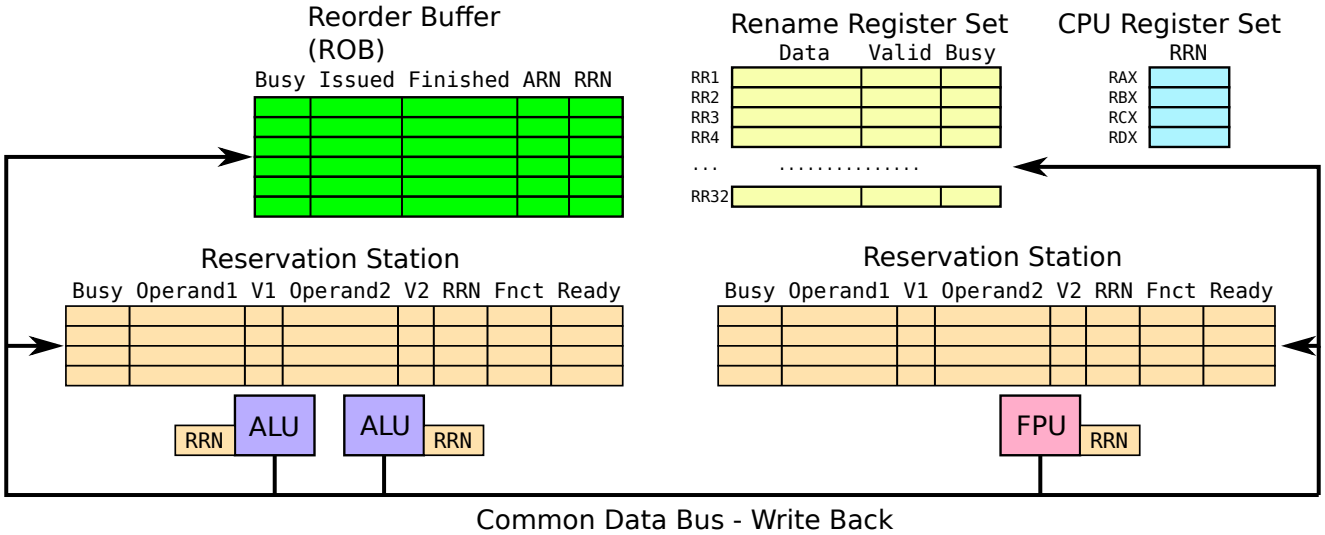
Tomasulo algoritmus

Robert Tomasulo z IBM vymyslel v roce 1967 algoritmus pro out-of-order vykonávání výpočtů na FPU. V současné době je jeho modifikace základem architektury všech moderních procesorů. Základní fáze jsou:

- Zavedení instrukce do ROB a přejmenování registru výsledku operace - získání čísla registru ze souboru záložních registrů.
- Zavedení instrukce do Reservation Station a čekání na připravená data
- Provedení výpočtu a zapsání výsledku do přejmenovaného registru přes Common Data Bus
- Dokončení nejstarší instrukce (kruhová fronta – FIFO) z ROB a aktualizace systémového registru.

Instrukce se mohou vypočítávat out-of-order, ale dokončení instrukcí je v pořadí programu.

Superskalární architektura



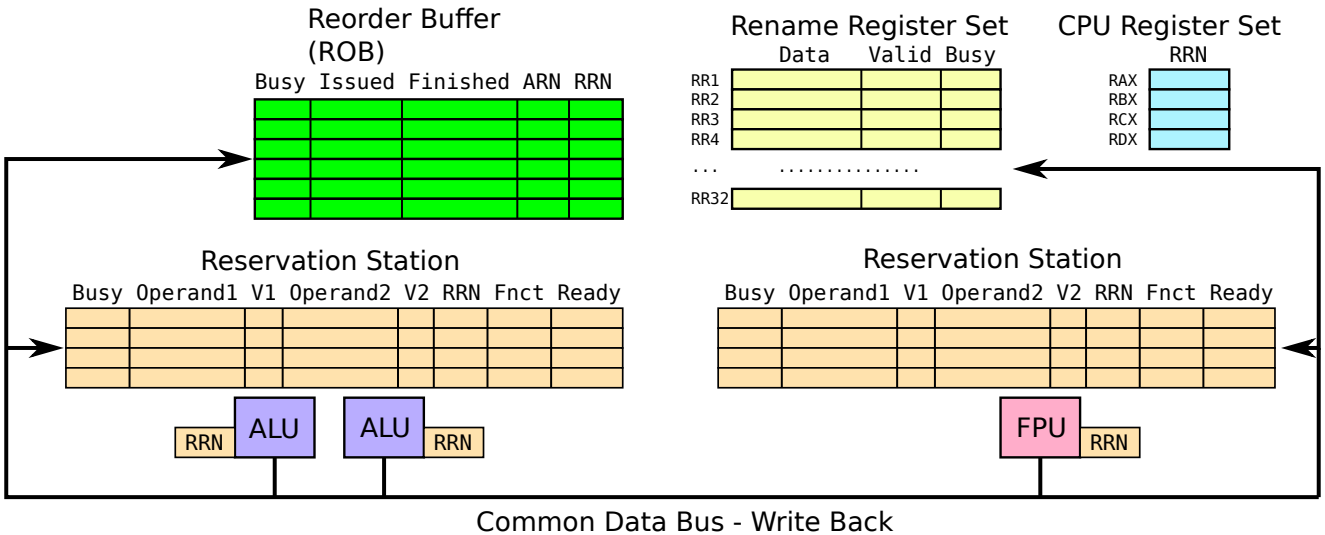
Rename Register Set:

- Množina pomocných registrů, často i mnohokrát větší než počet CPU registrů.
- Příznak Busy značí, že registr je využíván, Příznak valid značí, že hodnota je vypočtena a je validní. Při dokončení instrukce se nastaví CPU registr na RRN.

ReOrder Buffer:

- Cyklická fronta obsahuje instrukce vložené v pořadí programu
- Fronta zaručuje, že instrukce jsou dokončovány v programovém pořadí
- Provázání mezi Reservation Station přes číslo přejmenovaného registru (RRN Rename Register Number), který bude obsahovat výsledek operace
- Přes Common Data Bus se dozví, že registr RRN byl vypočten a instrukce dostane příznak Finished
- Dokončení instrukce - odstraní z ROB pouze nejstarší instrukci, až získá příznak finish.

Superskalární architektura



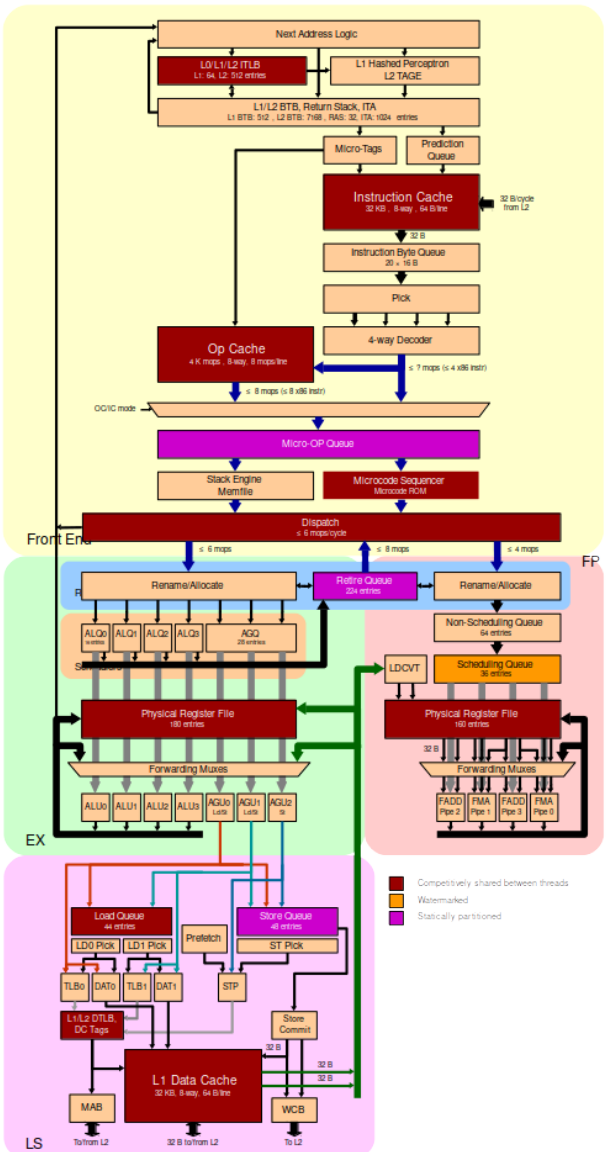
Reservation Station:

- Obsahuje dva operandy, pokud je nastaven příznak V1/2 pak je hodnota operandu validní. Pokud není příznak V1/2 nastaven, pak operand obsahuje číslo RRN na jehož hodnotu se čeká a která přijde z CDB
- Číslo RRN značí, do kterého registru se má zapsat výsledek.
- Pokud jsou oba operandy validní a je volná ALU/FPU, tak se zadá provedení operace a položka se odstraní z Reservation Station (RRN výsledku se zapamatuje, aby ho ALU/FPU mohla poslat na Common Data Bus)

Datové hazardy při čtení z paměti

- Pokud instrukce lw a sw využívají jiné adresy, tak je můžeme prohazovat.
- Pokud lw následuje po sw ze stejné adresy, lze implementovat přeposlání dat, data forwarding.
- V praxi ale může jedna instrukce předběhnout druhou tak, že ještě není vypočteno, kam se data uloží, tedy zda dojde k hazardu.
  - Řešení - spekulativní vykonání instrukce lw, tedy vykonání, i když není jasné, zda data budou správná
  - Při dokončování instrukce sw se zkontrolují všechny spekulativně prováděné lw
  - Při nalezení konfliktu se zruší spekulativní provádění instrukce lw
- Provedení se tváří, jako by bylo zachováno pořadí.
- Velký problém v multiprocesorových systémech – konzistence paměti při paralelních výpočtech na různých jádrech procesoru.¨

AMD Zen2 - Mikroarchitektura



- 7 nm process (from 12 nm), I/O die utilizes 12 nm
- Core (8 cores on CPU chiplet), 6/8/4 µOPs in parallel
  - Frontend, µOP cache (4096 entries)
  - FPU, 256-bit Eus (256-bit FMAs) and LSU (2x256-bit L/S), 3 cycles DP vector mult latency
  - Integer, 180 registers, 3x AGU, scheduler (4x16 ALU + 1x28 AGU)
  - Reorder Buffer 224 entries
- Memory subsystem
  - L1 i-cache and d-cache, 32 KiB each, 8-way associative
  - L2 512 KiB per core, 8-way,
  - L2 DTLB 2048-entry
  - 48 entry store queue

Autor: QuietRub
Source: `https://en.wikichip.org/wiki/amd/microarchitectures/zen_2`

Opakování - kvíz

Co je to řídicí hazard (control hazard)?

A hazard, který se musí řešit pomocí data forwarding
B hazard, který se musí řešit pomocí stall
C situace, kdy je nutné zahodit rozpracované instrukce
D problém nestabilních výsledků logických operací

Obsah

Řídicí hazardy v superskalární architektuře

- Skoky v programu mění, které instrukce se vykonají.
- Při podmíněných skocích není jasné, které instrukce se budou vykonávat.
- Výpočet podmínky pro skok může trvat dlouho, je rozpracováno mnoho instrukcí.
  - Řešení - spekulativní nahrání dalších instrukcí
  - Po dokončení všech výpočtů nutných pro rozhodnutí skoku se ověří, zda se mělo skákat nebo ne.
  - Při chybné predikci se musí zahodit mnoho rozpracovaných, nebo i spekulativně vykonaných instrukcí.
- I nepodmíněné skoky mají problém, vypočítat kam se skočí. Cíl skoku může záviset na výpočtu předchozích instrukcí, a proto nejde jednoduše určit při načítání instrukcí.
  - Řešení - spekulativně odhadneme, na jakou adresu se asi bude skákat, podle historie minulých skoků.
  - Po dokončení všech výpočtů se zkontroluje, zda se odhadla správná adresa.
  - Důležité zvláště pro návrat z funkce.

Řídicí hazardy v superskalární architektuře

- Skok je v programu statisticky každá 4 až 7 instrukce
- 20% skoků je nepodmíněných – skočí se vždy, není potřeba rozhodovat
- 80% skoků je podmíněných
  - asi 66% je skoků na vyšší adresu, neboli dopředu
    - tyto skoky odpovídají větvení typu `if`
    - z nich statisticky asi 60% se neskočí – budeme značit **NT** (Not Taken)
  - zbytek asi 34% je skoků na nižší adresu, neboli dozadu
    - tyto skoky odpovídají větvení typu `for`, `while` a `do ... while`
    - z nich statisticky 99% (téměř všechny) se skočí – budeme značit **T** (Taken)

Statické prediktory

Statické prediktory mají pro danou skokovou instrukci vždy stejný výsledek:

- Prediktor, který by vždy odhadoval, že se vždy skočí
  - Podle statistiky z minulé stránky by měl úspěšnost: $p_{taken} = (0.66 * 0.4 + 0.34 * 0.99) = 0.60$
  - Statisticky se ukazuje, že hodnota p_{taken} je pro většinu programů v rozmezí 60 – 70%.
- Prediktor BTFNT – Backwards Taken / Forwards Not Taken
  - Podle znaménka relativního skoku - skok zpět se skočí, skok dopředu se neskočí
  - Podle statistiky z minulého stránky by měl úspěšnost: $p_{taken} = (0.66 * 0.6 + 0.34 * 0.99) = 0.73$

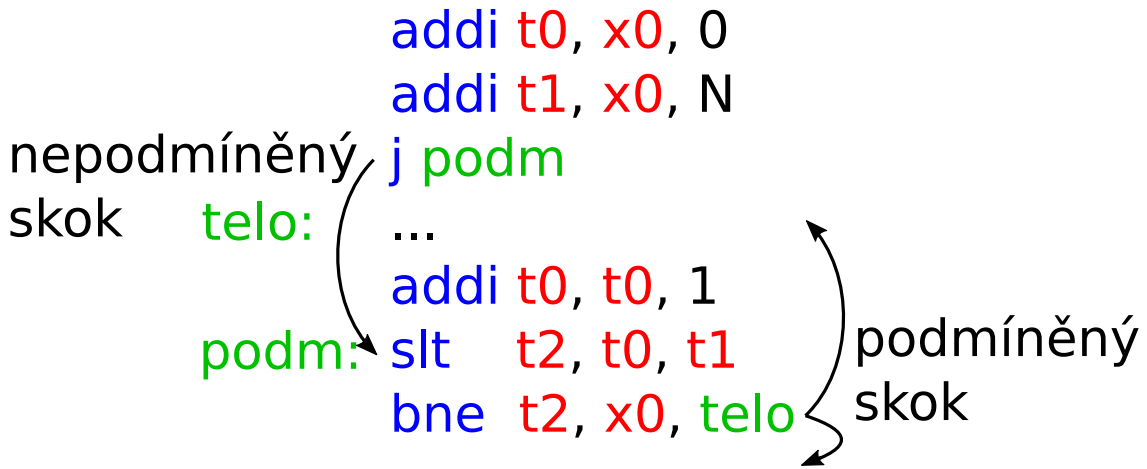
Statický prediktor BTFNT

Příklad překladu `for` cyklu:

```
for (i=0; i<N; i++) {

   ...

}
```



Pro podmíněný skok platí, že se $N-1$ krát skočí a 1 neskočí.
Skáče se vzad a pravděpodobnost správné předpovědi je tedy $\frac{N-1}{N}$

Statický prediktor BTFNT

Příklad překladu `if else` konstrukce:

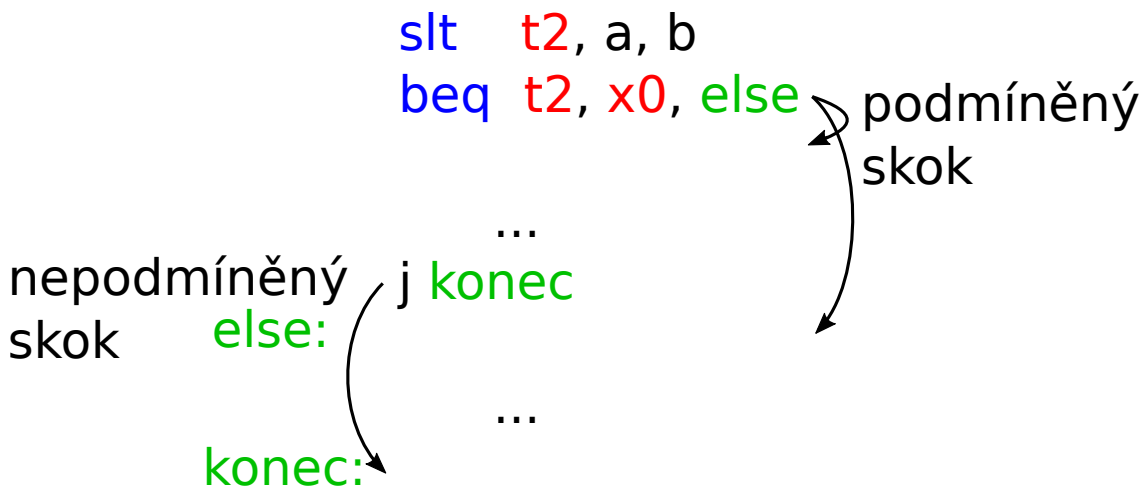
```
if (a<b) {

   ...

} else {

   ...

}
```



Nepodmíněný skok závisí na hodnotách a, b, nelze tedy o něm obecně nic prohlásit, jedině skáče se vždy dopředu.
Statistické chování záleží na typu programu, ale pro směs různých programů se ukazuje, že pravděpodobnost skoku je 0.4.

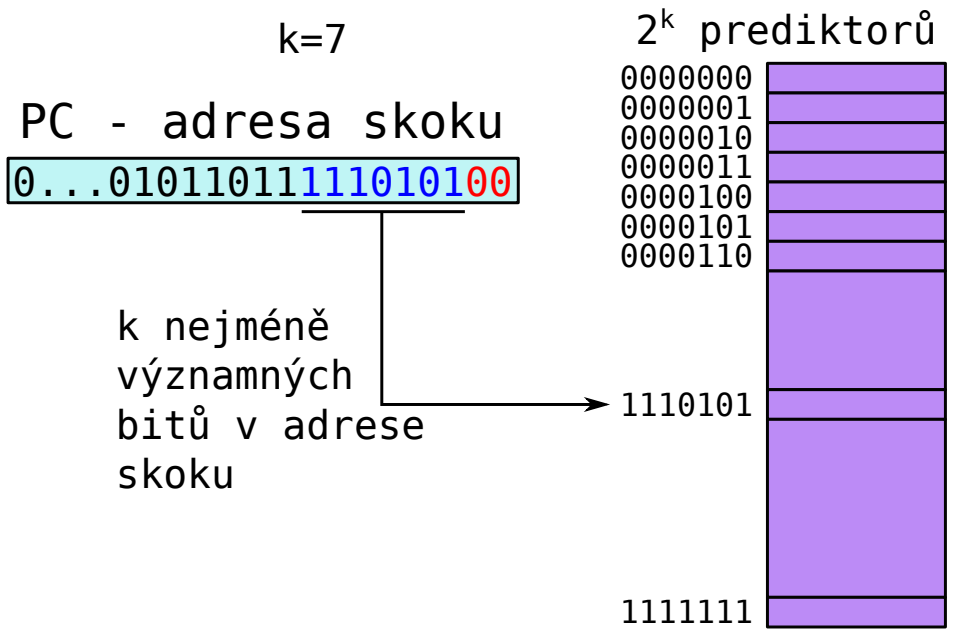
Dynamické prediktory

Dynamické prediktory se snaží odhadnout, zda se skočí na základě minulého chování dané instrukce skoku:

- Bylo by ideální, aby každá instrukce skoku měla svůj prediktor
  - To ale není možné, skoková instrukce může být na kterémkoliv místě paměti 4GiB

Řešení: budeme mít 2^k prediktorů a podle k nejnižších bitů adresy instrukce skoku vybereme prediktor

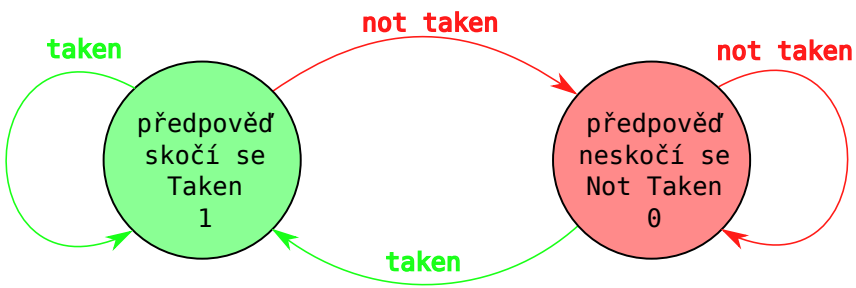
- Některé instrukce skoku na rozdílných adresách, ale se stejnými nejnižšími k bity adresy se budou ovlivňovat – **interferovat**.
- Interference mohou velmi nepříznivě ovlivnit úspěšnost predikce.



1-bitový Smithův prediktor

Nejjednodušší prediktor je 1-bitový Smithův prediktor

- Má pouze dva stavy, přepíná se podle minulého chování
- Předpovídá, že to dopadne, jak to dopadlo minule.
  - velmi jednoduchá implementace, vyhodnocení i úprava podle skutečnosti



1-bitový Smithův prediktor

Predikce pro for cyklus:

```
for (i=0; i<5; i++) {
  ...
}
```

```
        addi t0, x0, 0
        addi t1, x0, 5
        j podm
telo:   ...
        addi t0, t0, 1
podm:   slt  t2, t0, t1
        bne  t2, x0, telo
```

Pokud prediktor neinterferuje s jinými skoky, pak začíná ve stavu 0 – NT (Not taken).

Skutečné chování	T	T	T	T	T	NT
Predikce	NT	T	T	T	T	T

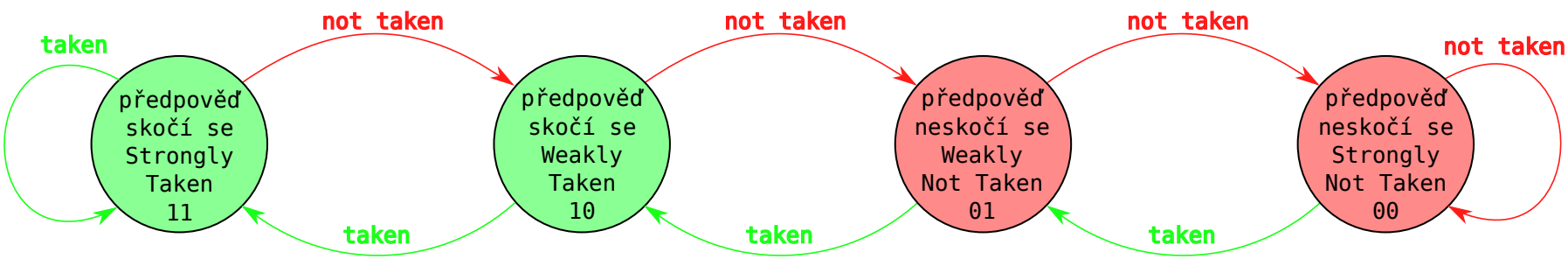
Vidíte, že prediktor není úspěšný na počátku for cyklu a na jeho konci.

Úspěšnost 1-bitového Smithova prediktoru pro cyklus s r iteracemi je $\frac{r-2}{r}$.

2-bitový Smithův prediktor

2-bitový Smithův prediktor patří stále k těm nejjednodušším

- Má již 4 stavy, reprezentované dvěma bity.
- 2 stavy předpovídají skok, 2 stavy předpovídají neskočení
- Předpověď závisí na minulém chování, ale toleruje jednu odchylku od pravidelnosti
- Velmi jednoduchá implementace, vyhodnocení i úprava podle skutečnosti



2-bitový Smithův prediktor

Predikce pro for cyklus:

```
for (i=0; i<5; i++) {
  ...
}
```

```
        addi t0, x0, 0
        addi t1, x0, 5
        j podm
telo:   ...
        addi t0, t0, 1
podm:   slt  t2, t0, t1
        bne  t2, x0, telo
```

Pokud prediktor neinterferuje s jinými skoky, pak začíná ve stavu 10 – WT (Weakly taken).

Skutečné chování	T	T	T	T	T	NT	
Stav	WT	ST	ST	ST	ST	ST	WT
Predikce	T	T	T	T	T	T	

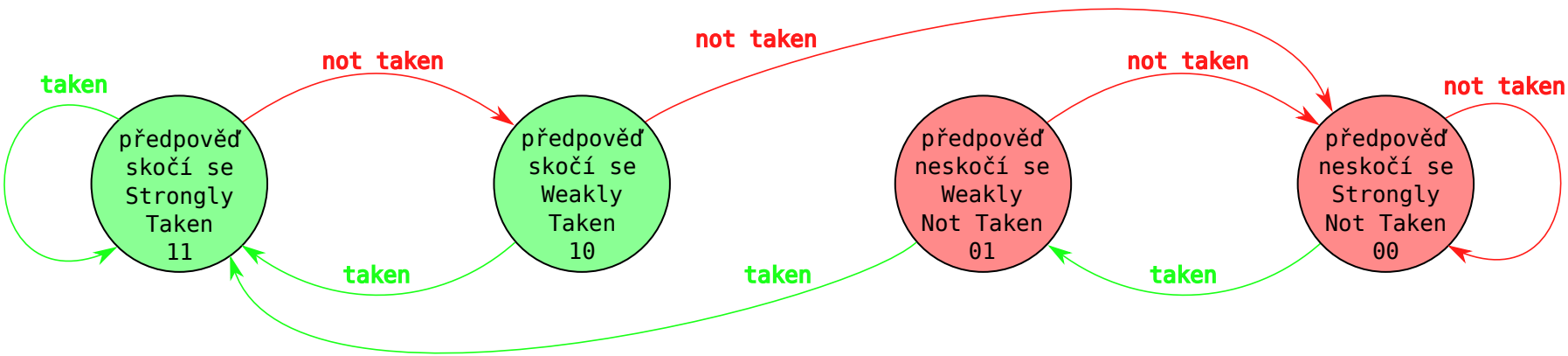
Vidíte, že prediktor není úspěšný pouze na konci for cyklu.

Úspěšnost 2-bitového Smithova prediktoru pro cyklus s r iteracemi je $\frac{r-1}{r}$.

2-bitový Smithův prediktor s hysterezí

Obdoba 2-bitového Smithova prediktoru

- Při dvou změnách po sobě se přejde rovnou do stavu Strongly a musí přijít další dvě opačné chování po sobě aby se prediktor překlopil do nového stavu.



Hodnocení prediktorů

- Je nemožné rozhodnout obecně, kdy je který prediktor lepší. Vždy lze najít protipříklady, kdy je každý z uvedených prediktorů chová nejlépe
- Jediná možnost je statistická analýza různých programů:

Statický prediktor - vždy se skočí	59.25
1-bitový Smithův prediktor	68.75
2-bitový Smithův prediktor s hysterezí	81.75

Zdroj: `https://ieeexplore.ieee.org/document/6918861`
H. Arora, S. Kotecha and R. Samyal, "Dynamic Branch Prediction Modeller for RISC Architecture," 2013 International Conference on Machine Intelligence and Research Advancement, Katra, 2013, pp. 397-401.

Závislost předpovědi

V praxi se ukazuje, že predikce závisí na předchozím chování program.

```
if (x==2) {  // skok s1
}
if (y==2) { // skok s2
}
if (x!=y) { // skok s3
}
```

Pokud se proměnné x, y nemění v tělech podmínek s1 a s2, pak tady máme silnou závislost skoku s3:

s1	s2	$\Longrightarrow$ s3	vysvětlení
neskočí	skočí	neskočí	x==2 a y!=2 tedy platí x!=y
skočí	neskočí	neskočí	x!=2 a y==2 tedy platí x!=y
neskočí	neskočí	skočí	x==2 a y==2 tedy neplatí x!=y
skočí	skočí	nevíme	x!=2 a y!=2 nevíme zda platí x!=y

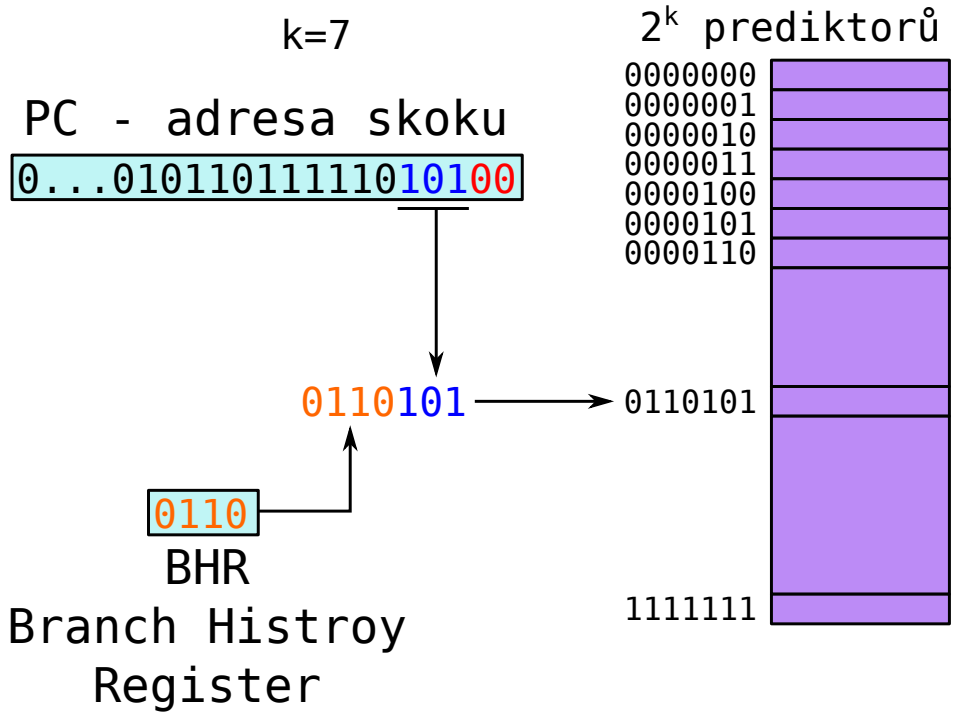
Historie skoků – korelované prediktory

Registr BHR (Branch Histrory Register) obsahuje informaci, zda posledních m skoků provedlo skok:

- Pokud se skočilo pak obsahuje 1, pokud se neskočilo pak obsahuje 0
- Nová informace se vloží na nejnižší bit, nejstarší informace se vyhodí z nejvyšší pozice, ostatní informace se posune.

Pro zjištění indexu prediktoru se vezme $k - m$ nejnižších bitů adresy instrukce skoku a tato informace se spojí s bity z BHR

- Výhoda – podle předchozích skoků se vybere jiný prediktor.
- Nevýhoda – některé kombinace se v BHR nevyskytnou, a proto se tyto prediktory nepoužijí.



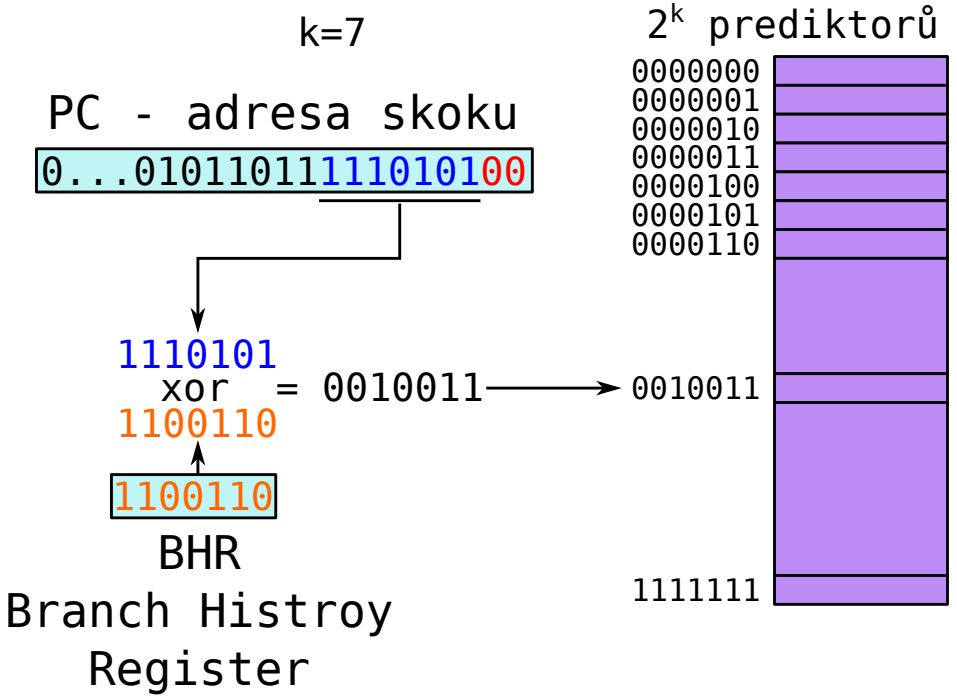
GShare prediktory

GShare přístup je podobný předchozímu:

- Také využívá BHR registr, který v této implementaci má přímo k bitů

Pro zjištění indexu prediktoru se vezme k nejnižších bitů adresy instrukce skoku a provede se xor s bity z BHR.
Výhody:

- lépe statisticky pokryje všechny prediktory.
- umožňuje použít delší BHR registr.



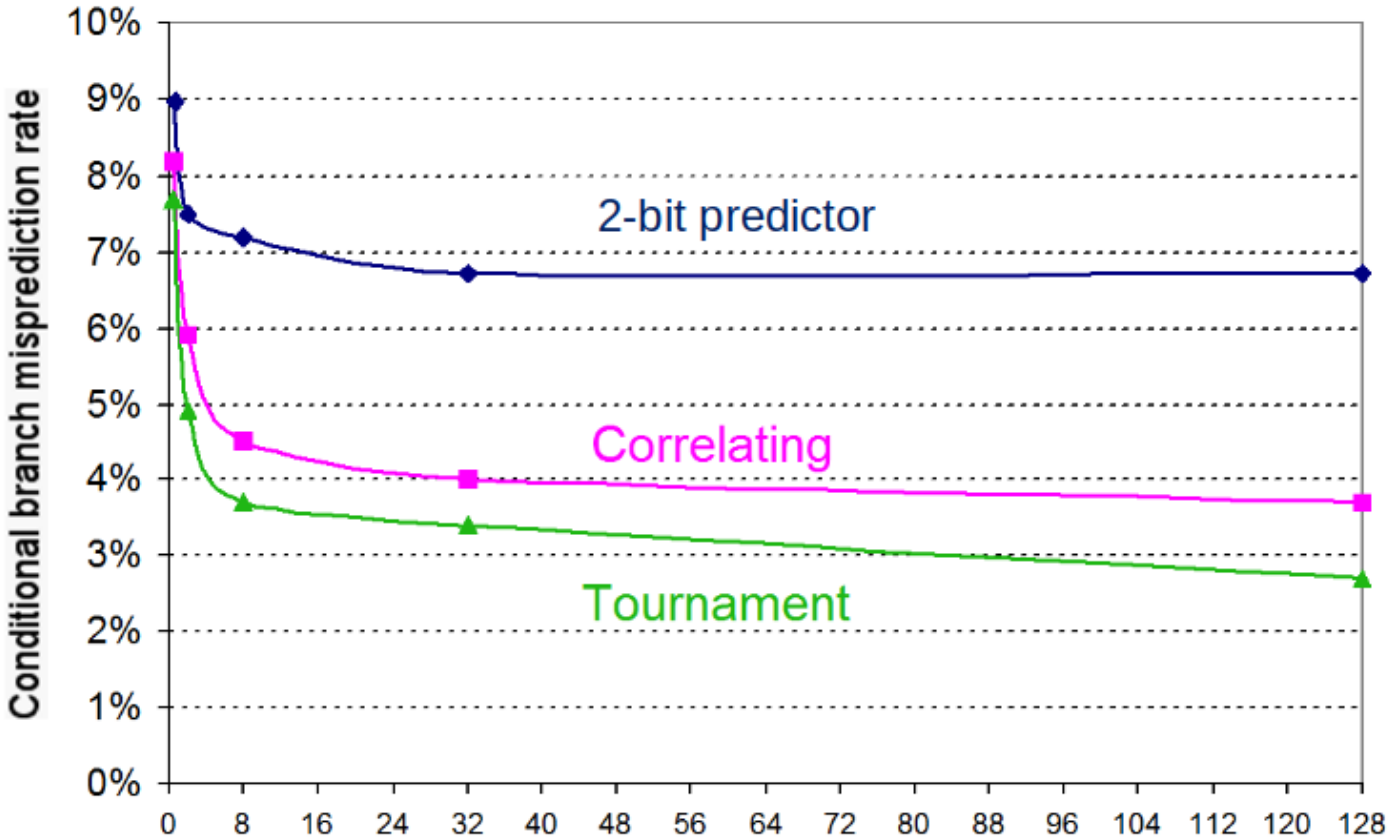
Turnajové prediktory

Základem turnajového prediktoru je soutěž dvou jiných, jednodušších prediktorů. Pro výběr, který prediktor je lepší a tedy, který prediktor bude na výstupu, lze použít 1-bitový, nebo 2-bitový prediktor.

Jak funguje 1-bitový turnajový prediktor

- Vypočti predikci prediktory P1 a P2.
- Pokud se predikce shodují je výsledkem tato predikce.
- Pokud se predikce neshodují:
  - Výsledná predikce je podle toho, který prediktor byl v minulosti úspěšný. Tato informace je uložena v 1-bitovém stavovém automatu.
  - Podle skutečnosti vyber prediktor P1, nebo P2 pro příští predikci.

Turnajové prediktory



Opakování - kvíz

Který prediktor nejlépe zvládne predikovat následující skutečnost skoků pokud začíná ve stavu Taken, nebo Weakly Taken:

T	T	T	NT	NT	NT	T	T	T

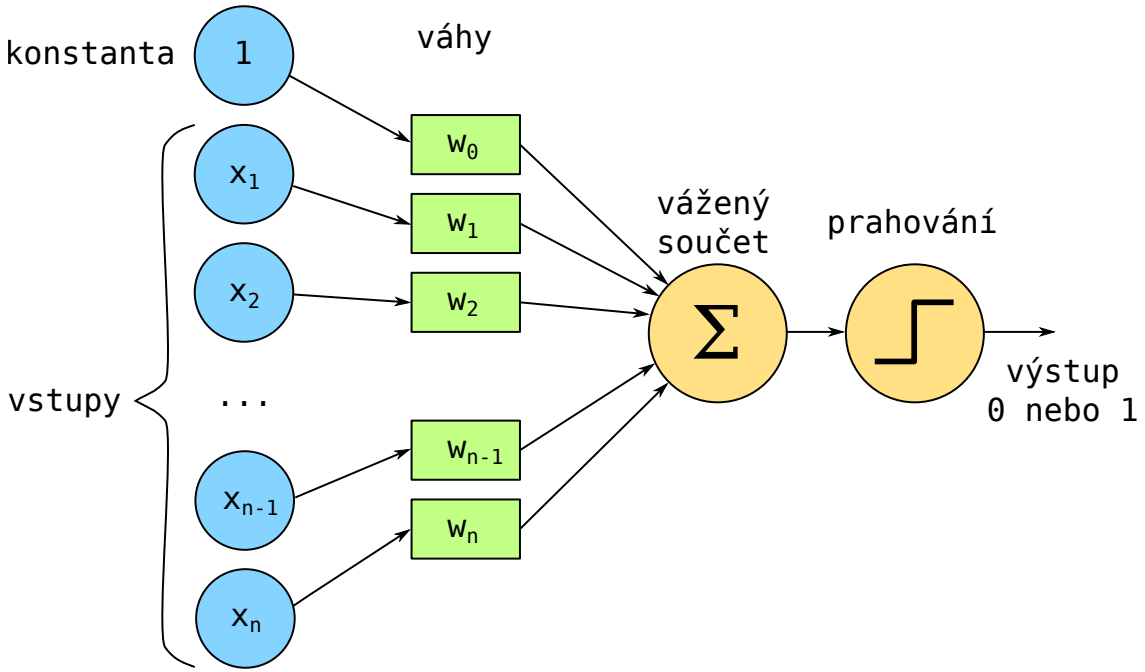
A 1-bitový Smithův prediktor

B 2-bitový Smithův prediktor

C 2-bitový Smithův prediktor s hysterezí

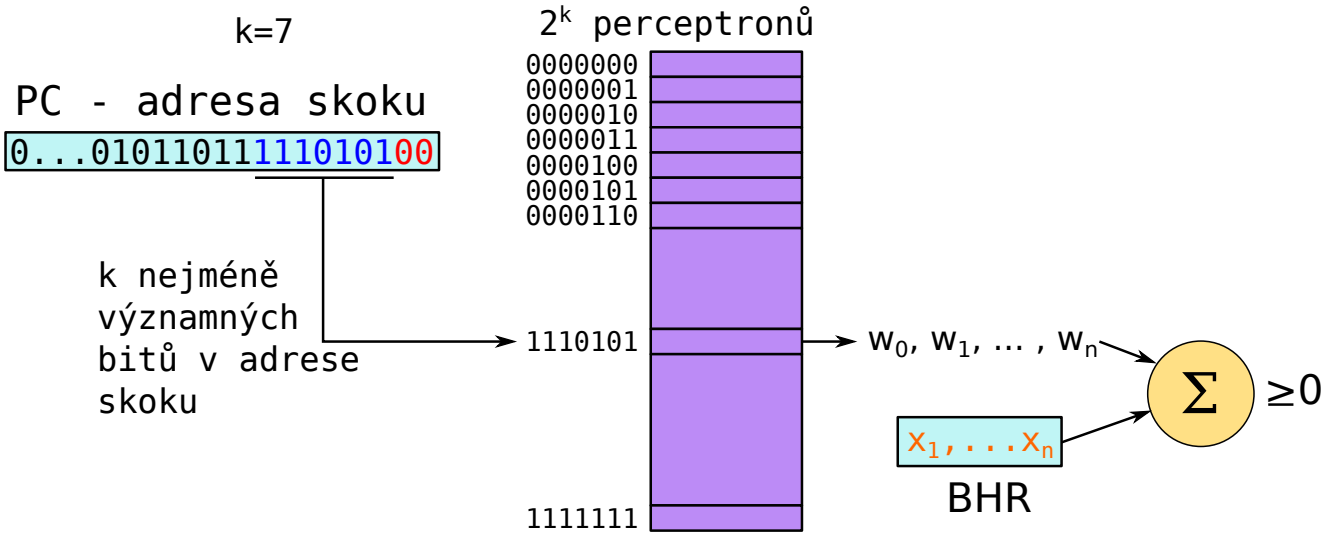
D všechny udělají stejný počet chyb

Perceptron



Pro výpočet hodnoty se použije vzoreček $\xi = \sum_{i=0}^{n} x_i \cdot w_i$. Závěrečné vyhodnocení je pomocí přenosové funkce, v našem případě skokové funkce, která má tvar: $f(\xi) = \begin{cases} 1 & \text{pro } \xi \geq 0 \\ 0 & \text{pro } \xi < 0 \end{cases}$

AMD Zen2 prediktor

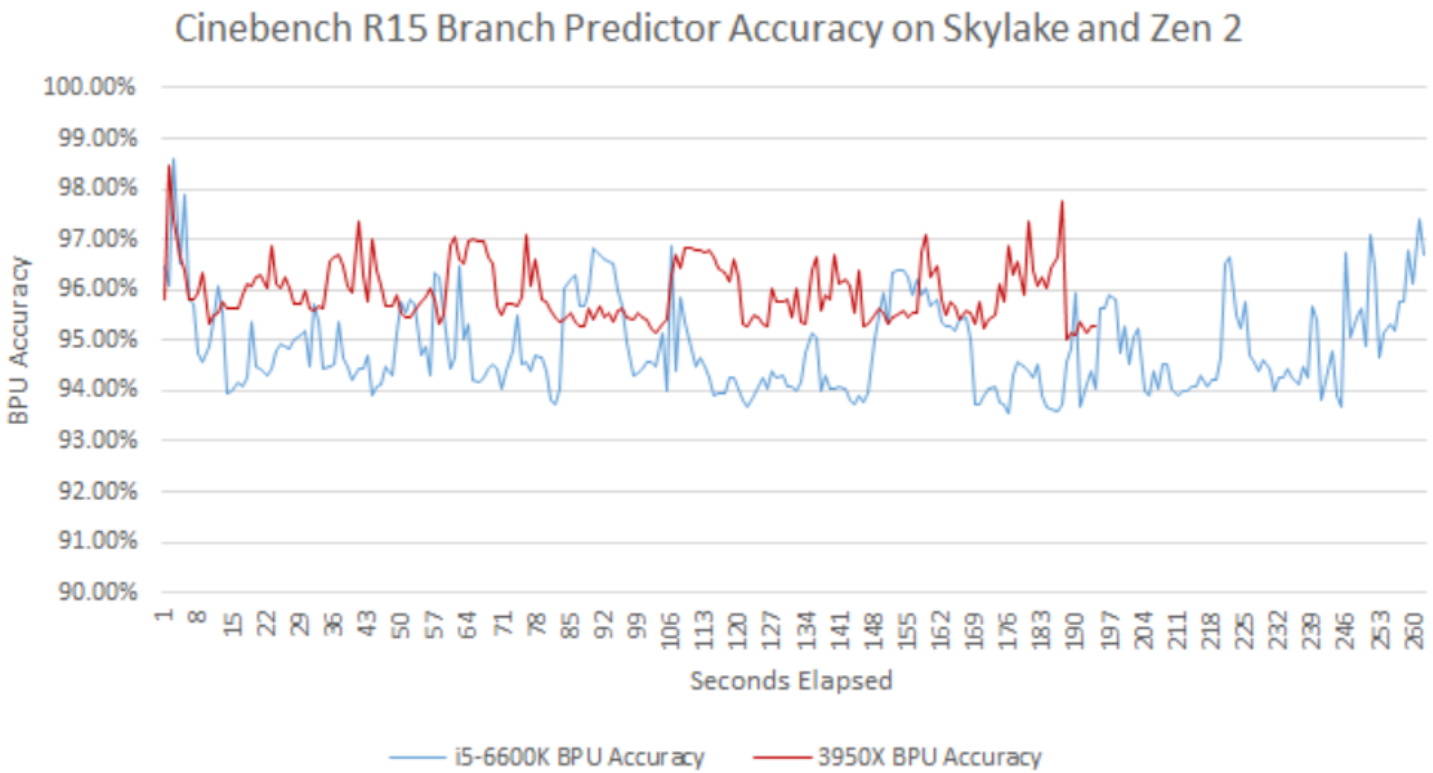


Podle adresy skoku se vybere jeden perceptron z tabulky.
Perceptron je definován vahami w_i.
Výsledek predikce je znaménko váženého součtu $\xi = \sum_{i=0}^{n} x_i \cdot w_i$, kde x_i jsou bity z registru historie skoků BHR.
Výhoda – lepší výsledky než gshare, lze použít i pro dlouhé registry historie BHR, nevýhoda – složitý výpočet, nelze získat výsledek v jednom taktu procesoru.

AMD Zen2 – Predikce skok

- Výpočet výstupu perceptronu je relativně pomalý, normální perceptrony používají reálná čísla s plovoucí desetinnou čárkou. Je možné urychlit 16-ti bitovými reálnými čísly, nebo použitím pevné desetinné čárky.
- Skutečná implementace prediktorů skoků má tři úrovně
  - úroveň 1 – 16 velmi rychlých prediktorů, rozhodnou ještě v tom samém hodinovém cyklu, které další instrukce načítat
  - úroveň 2 – 512 prediktorů, které v dalším hodinovém cyklu upřesní předpověď, buď se zahodí načítané instrukce, nebo se potvrdí
  - úroveň 3 – 7168 prediktorů, za 4 hodinové cykly zpřesní předpověď. Opět se dosavadní načtené instrukce buď uchovají, nebo zahodí.
- Průměrná časová prodleva při špatné konečné predikci je přibližně 18 hodinových cyklů.
- Při rychlosti 4GHz a pokud je každá desátá instrukce skok, tak 1% chybné predikce vede ke snížení výkonu o téměř 2%.

AMD Zen2 prediktor



Zdroj: Analyzing Zen 2's Cinebench R15 Lead By clamchowder from `https://chipsandcheese.com`

Obsah

Predikce cíle skoku

Skoky mají jak v RISC-V tak v ostatních procesorech různý formát cílové adresy skoku:

- skok na pevnou adresu – cíl skoku je adresa přímo uvedená v instrukci skoku (není v RISC-V, protože má pevnou délku instrukce a 32-bitová adresa se do kódu instrukce nevejde)
- skok relativně vůči adrese skokové instrukce – cíl skoku se vypočte jako součet PC v okamžiku načtení skokové instrukce a zadaná konkrétní hodnota v instrukci.
- skok na pozici uvedenou v registru, nebo v paměti – RISC-V má instrukci `jalr`, tedy skok na adresu uvedenou v registru, x86 obsahuje instrukce `ret` – návrat z podprogramu podle adresy uvedené v paměti na zásobníku a instrukci `jump indirect`, kdy adresa ukazuje do paměti, kde je uvedena cílová adresa skoku (tuto instrukci lze použít při skoku podle tabulky např. při překladu `switch` konstrukce, nebo při volání funkcí dynamických knihoven).

Predikce cíle skoku

- Cíl skoku je potřeba zjistit již ve fázi načítání instrukcí.
- Lze mít speciálně dedikované sčítačky na cílové adresy skoku, ale i tam součet nějakou dobu trvá.
- Proto je důležité odhadnout cíl skoku:
  - BTB (Branch Target Buffer) je buď plně asociativní paměť nebo částečně asociativní s daným stupněm asociativity
  - Řádky jsou dvojice:
    - klíč – BIA (Branch Instruction Address) – tedy hodnota PC v okamžiku skoku
    - BTA (Branch Target Address) – cílová adresa skoku
- Pokud se PC shoduje s hodnotou BIA v BTB a současně prediktor předpoví, že se skočí, změní se PC spekulativně na hodnotu příslušné BTA

Predikce návratu z funkce

Nejčastější skok podle adresy v paměti je návrat z funkce:

- Pro rychlou predikci adresy návratu z funkce obsahují moderní CPU – Return Address Stack (RAS)
  - Jedná se o rychlou paměť typu zásobník – pamatující si omezené množství (až 32) návratových adres
  - Hodnota se do RAS uloží při volání funkce, při načtení instrukce ret pak vrchol RAS slouží jako prediktor cíle skoku
- Funguje spolehlivě pro úrovně vnoření funkcí podle velikosti RAS

Obsah

Ukázka, jak odstranit skok v programu

Na internetu můžete najít mnoho triků, jak odstranit podmínku a tedy podmíněný skok ve Vašich programech.
Zde si ukážeme odstranění if z výpočtu absolutní hodnoty celého čísla:

program v C	program v RISC-V	komentář
`if (x<0) {`	`slt t0, s0, x0`	porovná, zda x<0
`  x = -x;`	`beq t0, x0, skip`	podle výsledku skočí, nebo ne
`}`	`sub s0, x0, s0`	vypočte -x
	`skip:`	

Odstranit podmíněný skok, který by se velmi špatně odhadoval lze následující konstrukcí, využívající nejvyšší znaménkový bit:

program v C	program v RISC-V	komentář
`int tmp = x>>31;`	`srai t0, s0, 31`	tmp bude buď 0, nebo 0xFFFFFFFF
`x ^= tmp;`	`xor  s0, s0, t0`	nedělá nic, nebo bitovou inverzi
`x -= tmp;`	`sub  s0, s0, t0`	pro tmp==0 nedělá nic, jinak odečte -1, tedy přičte 1

Ukázka, jak odstranit skok v programu

Pokud je hodnota b rovna 0, nebo 1, pak lze následující C program:

```
a = ( (b!=0) ? c : d);
```

změnit na program:

```
static const int lookup_table[] = {d,c};
a = lookup_table[b];
```

Odstranit lze i více skoků najednou, pokud opět b1, b2, b3 budou nabývat pouze hodnot 0, nebo 1:

```
a = ( b1 ? c : ( b2 ? d : (b3 ? e : f)));
```

lze změnit na program:

```
static const int lookup_table[] = { f, e, d, d, c, c, c, c };
a = lookup_table[b1 * 4 + b2 * 2 + b3];
```

Ukázka, jak odstranit skok v programu

Obdobně převod čísla od 0 do 15 na hex znak lze buď

```
if (a<10) {
  ch = '0'+a;
} else {
  ch = 'A'+(a-10);
}
```

lze změnit na program:

```
static const int hex_c[] = {'0','1','2','3','4','5','6','7',
                            '8','9','A','B','C','D','E','F'};
ch = hex_c[a];
```