NSWI143

Zkouška 27.05.2015 10:30 v místnosti S3

18,5 – 19,5 ⇒ (1)

Petr Houška <houskape@gmail.com>

2) 137 353 43272

3) - počet a typ přístupů k I/O
- počet a typ operací ~> ovlivňuje
  - počet instrukcí ~ byť to dost může zachránit překladač optimalizací
  - typ instrukcí ~ např. alg. kde hodně nepravid. skoků bude mít i při stejném počtu instr. větší CPI než ten kde skáču predikovatelně
- chování ke cache – alg. může používat časově lok. data – naposled. použ... paměťově lok. data – vždy vedl...

a něco může náhodně skákat -> horší

4) - Je výrazně rychlejší ~ řádově zlomky ns oproti desítkám (DRAM)
- Není ji třeba obnovovat
- Tvořena jen tranzistory (5-6) tj. stejný výrob. proces jako samotné CPU ~> snadná integrace
- čtení není destruktivní; tj. není třeba znovu ukládat po přečtení ~> zrychluje čtení

1/3,5

Petr Moravec

5) Procesory Intel a Mips mají jinou endianitu; tj jeden ukládá nejdůležitější skup. bitů (tady konkr. po 4 B) vlevo, druhý vpravo.
Proto MIPS bral jako nejméně důl. skup bitů 12_{16} namísto 78_{16} a zvýšil ji (na 13_{16}). Intel se o tom, ale neměl jak dozvědět a přečetl číslo normálně => zvýšená byla "most sig." skupina.

Řešením je standardizovat typ čísel posílaných po síti; tj posílat např. čísla jen v MSB (most signif. bit first) a na jedn. arch. pak (když se jejich domovská endianita liší) provádět konverze.) ← většinou spec instrukce => vydělé

6) Von-Neuman je jedna ze dvou architektur počítačů (druha Harvardská
- má sdílenou paměť pro data & instrukce
=> data mohou být instrukce & vice versa
=> možný sebe generující se / sebe modifikující se kód
-> složitější návrh (musíme řešit co vykonávat jako instr., co ne)
-> ne tak bezpečné -> někdo vám může podsunout spec. data a přimět nás je spustit jako instrukce

+ princip fce => čtu když nevím, instr. / pokud nevedeo...

1,5 - 2

Na rozdíl od Harv, která má data & instr. (které se provádějí nad daty) separátně => instrukce nemá jak měnit).

instrukce vygen jako data a pak je spustím jako instrukce

2/4 Petr Mašek

7) Data se zapíší jen do cache a nastaví se příznak (dirty bit) na true, že bude předně data z cache propsat níže než se zahodí
↳ přepisování většinou přes write-buffer reg → tak to nezpomaluje

2

—

- operaci čtení ani zápis to výrazně nezrychlí / nezpomalí
  → teoreticky rychlejší zápisy protože při write-trough se může zahltit přenos. kapacita cache ⇔ RAM což v případě write-back spíš nenastane
- šetří to přenosovou kapacitu → data se mezi cache ⇔ RAM (případně úrovněmi cache) nepropisují při každé změně (tj často po málo) ale jen občas (při zahození dirty vždy / požadavku na konzist. dat) tj. větší objem méně často. Což je vzhledem k overheadu kom. lepší.
- způsobuje probl. s konzistencí. V RAM totiž nejsou akt. data (dokud se nepropíše), což je problém, když k těm datům přistupuje jiné CPU či DMA. ⇒ potřeba mech. který dá vědět, že je třeba propsat → lehké (tedy velké, ale ne časté) zpomalení.

8) Výjimky a vnější přerušení jsou mechanismus jak přerušit norm. běh programu. Prakticky jiné věci řešené stejným mechanismem.
Výjimky: nastává ze samotného programu / procesoru
- výjimka v programu
- dělení nulou
- na někt. procs / v někt. jazycích přetečení

Přerušení: - vyvolává je ext (externí procesoru) zařízení (síť. karta, GPU, ...)
- způsob komunikace, že se něco stalo
- většinou skrze spec. vodiče na sběrnici k CPU
- např: - časovač ~ vyvolá přeruš - jednou za X ns
  - když disk dokončí DMA
  - když selže nějaký HW

Řešení: - procesor dodělá právě rozdělanou instrukci (zbytek pipeline anuluje; v příp. out of order ty provedené, co ještě hot. být neměly)

- uloží IP - instruction pointer - do reg. / na zásobník (dokumentace daného) & skočí na obsluhu výjimek -> předem daná adresa / tabulka

tam většinou je obsluha OS -> ta uloží registry, kontext ...
-> zpracuje výjimku
-> většinou se vrátí za instr. která výjimku vyvolala / za instr. před kterou bylo vyvolané přerušení

2

---

a) Pokud jinak cache zůst. stejně velká a stejně asociativní, tak se zvětšením řádku

- zlepší prost. lok. cache -> méně cold missů -> & hodn. se rovnou načte víc,
- zhorší konfliktnost cache -> víc conflict missů -> méně různ. příhodných sousedů -> víc se vyhazují navzájem

=> třeba najít zlatý střed

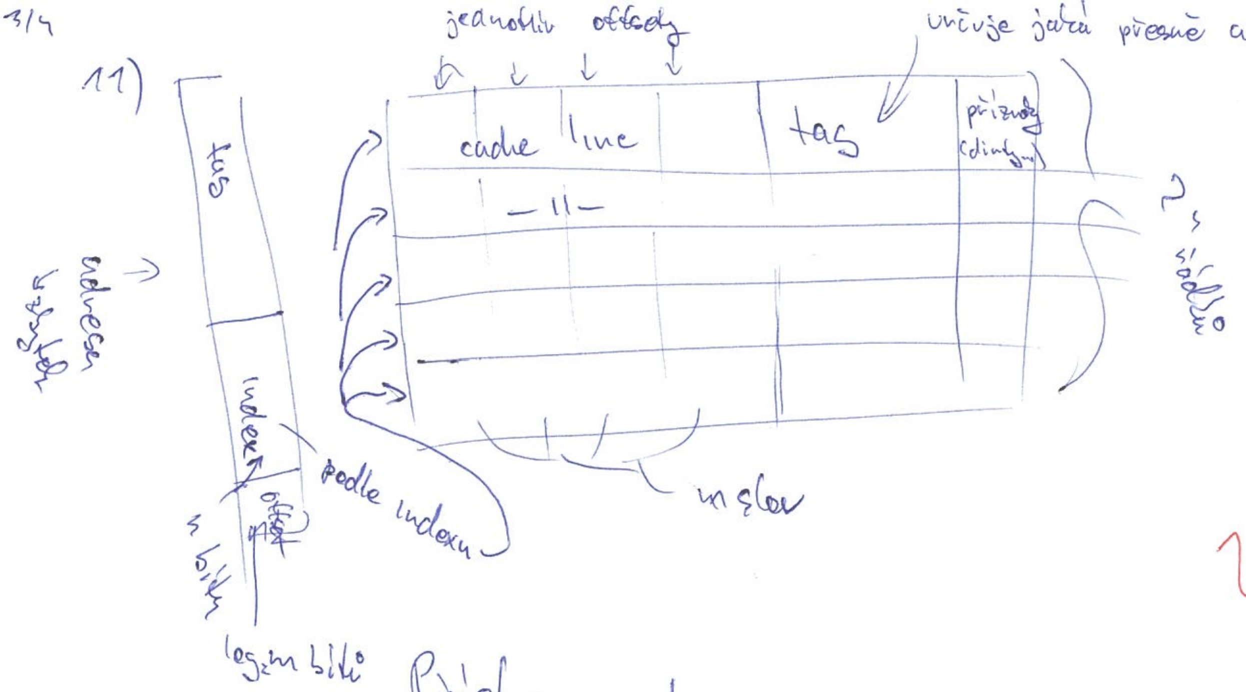
10) - do reg načíst hodnotu z adr jiného reg

```
load R1 [SP]        <- načte do R1 adresu kde je pole A
load R2 [SP]
add SP "-4" SP      <- přičtení záp. čísla k hodnotě v reg (dekrementace) (počítám že pro B)
store R3 "4"        <- uložit do reg konstantu
mult R3 R3*R2       <- vynásobit reg s hodn. v jiném reg
add R3 R1 R3        <- uložit do reg součet reg
~~load R5 [R3]~~
~~add R5 R2 R5 "1"  <- uložit do reg součet reg a konstanty~~
store [R3] R5       <- uložit na addr z reg. hodnotu z reg.
load R5 [R3]
add R5 R5 1         <- sečíst reg s konstantou
add R3 R3 "4"
store [R3] R5       <- uložit na addr. z registru hodnotu z reg.
```

zde by se hodila adresace báze + offset (registr)

2

11)



2

Přístup: adresa se rozdělí na 3 části: index, tag, a offset. Podle indexu se vybere řádek. Následně se zkontroluje jestli sedí tag + příznaky (např. jestli je cache line validní). Pokud ano => podle offsetu se vybere jaké přesně slovo z cache line chceme.

12) Asociativní paměť si pamatuje vždy hodnotu – klíč, tj. může uchovávat cca adresy.
→ v procesoru taková HW hash tabulka, většinou indexovaná částí adresy.

Např. cache – asociovaná index. částí adresy
paměť branch predikce ~ adres. částí IP adresy...

1

7)

$$\frac{1 \leftarrow \text{původní trvání}}{1 - \text{poměr zrychl podproblému} + \dfrac{1}{\text{zrychlení podproblému}}} = \text{cfsl.}$$

0 – 0,5

jaká je to část celého programu

jaká část se urychlí

zrychl. program

vysvětlení? jak rychle aplikace např. na 7x zrychlení násobení, které se vyskytuje v celém progr.

Petr Mošťek

13) Většina alg. využívá prost. a časovou lokalitu přístupů k paměti tj. "v jeden" moment často přistupují k datům ke kterým jsem už přistoupil dříve / která jsou okolo dat ke kterým jsem přistoupil. => V jeden moment by mi mohlo stačit malé množství velmi rychlé paměti a _hodně_ by to pomohlo.

A právě takhle dělá hierarchie cachí.

Nejmenší a nejrych. těsně u procesoru (kB)
Pomalejší pod tím (vyvarává kom. rychlé cache <=> RAM)
RAM
:
Disk
:

Výkonost závisí na minimalizaci $t_{hit} + \%_{miss} \cdot t_{miss}$ tj min.
šance že tam zrovna ty data nebudou a rychl. přístupu když tam
jsou * rych. když tam nejsou.
Problémy: tyhle požadavky jsou lehce protichůdné + SRAM je drahá
=> kvůli tomu je to pyramida a ne jedna velká superrychlá cache
musí se najít zlatý střed pro: velikost
velikost bloku
stupeň asociativity

1)

4/4 Petr Mareš

14) forwarding a bypassing je mechanismus, který dokáže předat mezivýsl. instrukce co je dál v pipelině nějaké, co je v začátku přímo na vstup.

Tj. když op. 1 add R_1 R_2 R_3
store [25] R_1 ← tak forwarding dokáže bypass

předat výsl R_2+R_3 instrukci store aniž by ho ještě uložil do reg R_1 a aniž by ho store z R_1 načetl.

← to se pak samozřejmě stane ale díky forwardingu nemusí store čekat ⇒ pipeline zůstane plná

pokud load probíhá v posl. části pipeline ⇒ není jak ho přeskočit a uspořit

↑ nejsem si jist

2