Zvyšování výkonu procesorů

11. listopadu 2008
12:30

Sériové provádění
Pipeline [linka proudového zpracování dat]
Myšlenka hodně stará -- už v 50. letech byly náznaky
Umožňuje rychlejší provádění instrukcí, že nezávislé výkon zároveň
Na různých úrovních se dá provádět, a na podřazených se dá také využít

Když budu dělat pipeline, udělám kroků co nejvíc, to jde jen do určité míry, protože řízení okolo také něco stojí, a to už se zrychlit nedá

Problémy:
- Zavádíme paralelismus
  Např. druhá instrukce používá výsledek předchozí, proto musím počkat na výsledek první
- Procesor se navenek musí tvářit striktně sekvenčně

Problémy při proudovém zpracování:
- Strukturální hazard
- Datový hazard
  - Nejčastější problém, abych mohl vykonat instrukci, potřebuji všechna data, a musím počkat na vykonání předchozích
- Řídící hazard
  - Nutno vykonat rozhodnutí před vykonáním instrukce .. Typický problém je při zpracování skoků.. Ještě nevím jestli skočím nebo ne, a nevím jestli mám předem zpracovávat před skokem nebo po skoku

Hazardy:
- Přístup k datům:
  - Čtení z externí paměti způsobí zpoždění ve fázi fetch .. Řešení: zahájím čtení tak dopředu, abych ji opravdu měl k dispozici.. Pokud to jde
- Závislosti
  - V tabulce položky s pomlčkou jsou díry, tedy místa, kde se nic nedá provádět a čekáme na provedení
  - Nenecháme, aby to nastalo, tím, že vložím ke zpracování něco dalšího, a využije výkon k tomu, k čemu už mám prostředky
  - Nebo přerovnáme průběh
- Závislost skoků
  - Zpracovávat během toho něco, co s tím nic nesouvisí
- Skoky
  - Skok si vynutí vyprázdnění pipeliny a začnu načítat nové instrukce
  - Řešení:
    - **Multiple streams** - používá se pro skoky - zjistím, že jde o skok, proto tedy zpracuji pro obě varianty a která nastane, tak tu použiji a druhou zahodím; jednoduché technické řešení; technicky mám vedlejší pomocnou pipeline, ze které to když tak zkopíruji do hlavní; vyžaduje tedy tu vedlejší pipeline
    - **Look ahead / look behind buffer** - "Tak nebo tak" .. Paměť skoku
      - Pro řešení těchto výpadků máme paměť na několik instrukcí, a pokud se skákalo, tak se ukládaly do paměti (x posledních instrukcí) instrukce skoku a pokud byl další skok, tak se procesor podíval do paměti, jestli už jsme ji nevykonávali předtím
    - **Delayed branch** - než skočím, tak ty instrukce rozpracované v pipeline nechám dojet a pak skočím
      - Procesory: SPARC (1 instrukce), [CDC GaAs (2 instrukce) - ve své době uvedení 1989 měl taktovací frekvenci 1GHz - založen na Galiu Arsenidu místo křemíku; značně nedostupný; pro superpočítače]
    - **Branch prediction** - finta, jak vrovnat s vyprázdněním pipeliny.. Mám zapsáno

jak asi ten skok skončí, odhad.. Pokud vyjde, tak ok, pokud ne, tak tam zpracuji něco jiného; v instrukci mám napsanou nápovědu jak ten skok asi skončí a podle toho předzpracovávám

- ◇ Predikce: 1) kompilátor pozná, 2) procesor pozná
- ◇ Bez historie programu - statická
- ◇ Dynamická predikce - ukládání predikce do bufferu; (2-bitová historie) na začátku mám odhad, procesor se podle toho zařídí, pokud správně, OK, pokud špatně, tak si procesor poznamená, že predikce byla špatně, pokud narazím podruhé, a predice zase špatně, tak si ji už změním
- ◇ Na začátku jsem ve stavu, kdy predikce říká, že skok nastane, nebo nenstane.. Máme hint že skočíme, .. [viz obr na slidech .. Diagram začíná vlevo nahoře]

Vývoj pipeline

- Myšlenka pipeline už je dlouho, 40 let minimálně
- MIPS R4000: 8 kroků (deep pipelining)
  - [popis kroků ve slidu] - načítání po 2 částech,
- Superscalar pipelining
  - Idea: když budu mít jednu nebo 2 pipeliny, paralelně zpracovávající tok informací, tak na tom 2x ušetřím
  - RISC Systém 6000 - větvění po dekódování
    - □ Různé cesty pro int nebo float
  - Můžeme mít i více než 2 pipeliny.. Třeba pro float výpočtu dám víc, protože počítání ve floatu je časově náročnější
  - Problém může opět nastat v závislosti na datech, to by ale měl pohlídat kompilátor
  - I vícecelých pipeline.. Sdílené výpočty
- Dynamic pipeline
  - Na začátku nějak načítám instrukce, pak se podívám co je za typ, pak se podívám jestli je jednotka, která to má na práci, pokud není k dispozici, tak počkáme až taková bude k dispozici, nakonci je odkládací, která z plůvodního sériového provádění zase zpět seřadí do původního pořadí.. Navenek se tváří sériově, ač uvnitř se to možná provádí mimo pořadí

Další zrychlování

- Simultánií multithreading
  - Do pipeliny dáváme instrukce různých vláken, které na sobě skoro jistě nebudou závislé
- Predikce hodnoty - např. instrukce load ukazuje, če často zavádí stejná data, takže nečekám na načtení a použiju z minula, ale pokud jsme nahráli jiný, tak výpočet zahodím a provedu znovu s těmi daty.. Nedochází ke zpomalení, ale může pomoci ke zrychlení

Statistika užívání instrukcí

- Ve vykonávání bežného programu se skoro polovina instrukcí týká přesunu dat
  - Pro zrychlení: optimalizace instrukcí, které sep používají nejčastji, klidně na úkor nějaké instrukce, která se používá minimálně
- 50% konstant se vejde do 5 bitů, 98% konstant se vejde do 10 bitů,
  - instrukce pracující s konstantami se dají smrsknout na menší pamět, a pokud náhodou větší, tak ji holt hodíme do paměti

Instrukční sady

- Do 80. let … [viz slide]
- Do té doby se používal vesměs jen assembler nebo zdrojový kód..
- Snaha nejen zrychlit HW, ale i po stránce SW a logického využití
- = změna priorit
  - Důraz na to, co je nejvíce potřeba a co trvá nejdéle
  - Implementace méně používaných instrukcí může zhoršit celkový návrh
    - □ Když by to mělo zhoršit, tak takovou instrukci tam nedám
  - Spolehnout se na vyšší jazyky a optimalizující kompilátory
    - □ Člověk by to technicky nemusel umět zvládnout, a kompilátor se s tím popere
  - Paměť je dnes rychlejší a levnější
    - □ Mnoho paměti ve stroji, proto se mohla použít delší sekvence
- MIPS -

  - John Hennessy - byl pomocníkem na Standford University - škola, která měla dobré zázemí v kompilátorech
  - Vývoj procesoru, jehož architektura by vyjadřovala snížení kompilátoru na úroveň HW (místo obvyklého povyšování HW na úoven SW)
- CISC vs. RISC - historické označení complex vs. Reduced
- CISC - proč vůbec byl?
  - Stroje vznikaly postupně, takže ze začátku to nebylo třeba
  - Výrazně se zlevňoval HW na úkor SW, proto se lidská práce měla zjednodušit
  - Pokud bude dražší HW, tak ušetřím, protože nemusím ještě víc platit programátorovi
  - Méně instrukcí pro daný úkol, méně přístupů do paměti, která tehdy byla drahá!
  - Implementace pomocí mikrokódu se dá snadno měnit
    - Umožní dělat složité věci, proto toho využívám
  - Méně instrukcí ale neznamená, že výsledek bude menší, protože se musí ošetřit všechny eventuality, při zápisu do strojivého kódu
- RISC
  - Pro ty instrukce, které se používají nejčastěji, tak se implementují aby byly co nejrychlejší
  - Přidám další instrukce, pokud nezpomalí procesor
  - Přesunutí komplexních činností do kompilátoru
    - Složité operace musí zvládnout kompilátor za použití jednoduchých instrukcí - např. instrukce násobení bez použití instrukce pro násobení
  - Charakteristika
    - Jedna instrukce na cyklus - problém s aritmetikou, u hodně procesorů pak nebyly složité matematické operace (např. SPARC neměl násobení)
    - Operace typu registr - registr - pokud mají zacházet z pamětí, tak je to pomalé, pokud něco počítám, tak můžu pracovat s více registrama a až výsledek uložím
    - Architektura load - store - ukládám až výsledek
    - Malý počet jednoduché adresovací režimy -žádné kombinace různých adresování
    - Pevný formát instruckce - jednodušší práce pro dekodér
    - Malý počet a jednoduché instrukce
    - Velké množství registrů - abychom epracovali s pomalou pamětí
    - Použití linky proudového zpracování (pipeline) - značné zrychlené
    - Zvláštní zpracování skoků (hazard….)
    - Hardwired řadič - není potřeba obecných mechanismů, založeno na obvodovém řešení = značně rychlejší, jedu sekvenční logikou
    - Silná závislost na kompilátoru - který zajišťuje "luxus" pro programátora
  - Původně vzniklo jako návrh školního procesoru, ale ujal se i komerčně
  - = pokusy o definici:
    - N.J. Davis:
      - omezená a jednoduchá instrukční sada
      - Velké množství obecných registrů
      - Dúraz na optimalizace pipeline
    - Colwell:
      - Prováděni instrukcí v jednom taktu (což mikrokód nedovoloval)
      - … viz [slide]
    - Filozofie přístupu k procesoru, důležitější než velikost programu je jeho rychlost! = optimalizující kompilátor.. Technické prostředky - pevná délka instrukce, zpožděné větvení atp…
  - První procesory:
    - Výzkumné procesory:
      - MIPS (Standford)
      - RISC 1, RISC 2 (Berkeley)
      - IBM 801
  - Typičtí představitelé:
    - Jak v desktop/server řešeních tak i v embedded
    - V této řadě byly architektury strašně podobné od různých výrobců
    - Desktop: PowerPC (IBM + Motorola) - původně mělo nahradit Intel
    - Embedded: ARM, Thumb, SuperH [Hitachi], MIPS16
- Návrh procesoru
  - [viz slide] - nad čím se zamyslet při návrhu
- RISC nebo CISC?

  - Není jednoznačná odpověď
  - Kvantitativní přístup:
    - Chceme malou velikost na úkor rychlosti? CISC
    - Chceme rychlost na úkor velkého návrhu? RISC
  - Kvalitativní - vyhodnotit podporu vyšších jazyků
  - Většina dnešních návrhů bere z obou kategorií.. Na stejnou plochu se vejde víc tranzistorů
    - Intel P4 - bere jednoznačně z obou kategorií
- Vznik post-RISC - kombinují oba přístupy s metodami, které nejsou použity v žádné z těchto kategorií
- Post risc
  - Větší míra paralelismu
    - Víc jednotek co umí to samé, tak to můžu zrychlit (CISC neměl, RISC omezeně)
    - Nove uspořádaní nové jednotky
    - Agresivní přerovnávání instrukcí v průběhu zpracování - "out of order execution", "speculative execution" (odklon od závislosti na kompilátoru)
      - O tom, jak se to bude vykonávat bude rozhodovat ne kompilátor, ale až HW (kompilátor mu to jen vhodně předpřipraví)
    - Opět přibyli další registry, cache přímo na procesoru, floating point jednotky
  - postup
    - Data z paměti se načítají do Data Cache
    - Jednotky: Fetch/flow a (decode/branch - zjistí co je za instrukci a kdo by ji měl zpracovat - a pak ji hodíme do bufferu)
    - A execution units.. Ty zpracovávají, pokud nemají co dělat, tak se zeptají, jestli není něco, co by mohly dělat a pak to hodí do bufferu)
  - komponenty
    - Predecode unit, I- cache
      - Instrukce načítány po blocích
      - Částečné dekódovní zjisti vlastnosti instrukcí a uloří je do vyrovnávací paměti instrukcí
      - Dodatečné příznaky určují:
        - -
        - -
        - -
    - Příklad záznamu v I-Cache
      - Ve vyrovnávací paměti uloženo:
        - Blok instrukcí
        - Historie skoků
        - Predekódované příznaky
        - Historie vykonávání bloku
    - Fetch flow
      - Chystá a udržuje predikce, bloky si nějak poznačuje a dělá si mezi nima návaznosti, tam se projevují vlastnosti skoků, jestli byl nebo nebyl vykonán
      - Má vliv na instruction cache
      - Ukáže-li se odkaz jako špatný, je změněn
    - Decode / branch
      - Vlastní dekódování instrukce
      - Rozdělení na dva proudy do doby, než je znám skutečný cíl skoku
      - "barvení instrukcí" - jako ilustrace
        - Třídy které se provádí společně .. Např. na pozici 5 zjistím, že celá červená sekce se nemá vykonávat.. [viz obr slidu]
        - Při barvení vím, jak můžu zpřeházet
    - Výsledky z branch unit
      - Fetch/flow unit
        - Aktualizuje flow history
        - Načte správné instrukce
      - Branch/decode
        - Aktualizac branch history

- Instruction despatch nd reorde buffer
  - Čekají zde až je někdo bude moci vykonat
  - K provedení dojde pokud jsou k dispozici
    - Vstupní hodnoty
    - Výstup
    - Exekuční jednotka
  - Přednost mají
    - Starší instrukce - kdyby mladší commit buffer by na ně stejně čekal
    - Instrukce load - poskytují data jiným instrukcím
- Datové závislosti při přerovnání
  - True data dependency - RAW read after write
    - Výstup instrukce je použit jako vstup následující
  - Outpt dependency WAW - write after write
    - Dvě instruce zapisují na stejné místo
  - Anti dependency Write after Rread WAR
    - Zatímco jednainstrukce zpracovává data, další instrukce tato data změní
  - WAW A WAR jde vyřešit přeznačením
    - [viz slide Eliminace WAW]
    - Je třeba rozhodovat co vykonat a co nevykonat
  - Out of order execution
    - [vizz slide]
    - Vykonávám ne jak je psáno v programu,ale když pro ně mám data
  - Execution units
    - Podobně jako u RISCu
    - Jednotky, které nepočítají v 1 cyklu použijí pipeline
    - Jednotky load/store obvykle umožnují rozpraovat více přístupů najednou
  - Completed Instruction Buffer, Retire Unit
    - Znovu složí instrukce, aby vypadaly jako seriově zpracované
    - V bufferu jich může čekat docela dost čekající na poslední pomalou, a pak se všechny uvolní najednou.. Může vypadat, že za jeden takt provedeno více instrukcí
    - Ukládání vykonaných instrukcí spolu s:
      - Příznak vyjímky
        - Nemůžu vykonávat vyjimky když nastanou, ale když na ně přijde řada.. Protože mezitím třeba dokončuji výpočet v pořádku, který měl vyjít dřív
        - A při výpisu více instrukcí najednou vypíšu všechny do první vyjímky
        - Ošetření vyjímek je docelanáročné
      - Mapováním (přejmenováním použitých registrů)

    (viz. Další stránka o přerušení)

      - U postrisců se řeší tak, že odkládací jednotce se řekne, ať případně smaže obsah, když má skončit např.
    - Precizní přerušení vyjímka.. Viz slidy..

RISC vs. Post-RISC

- RISC: - výkon je dán stupněm paralelismu a tedy délky pipeline
- Post-RISC: výkon je dán tím, jak rychle se mi daří dané instrukce odkládat

Proudová linka u postrisců

- Viz slide..

Příklady procesorů post-RISC

- Ojedínělé znaky
  - DEC Aplha 21164
    - Používá pouze minimum z návrhu postrisc

        - Vysokou taktovací frekvenci
        - Používá predikci skoku -> tabulka historie skoků
    - SUN UltraSparc
        - Instrukce už v balíku po 4
        - Některé instrukce odkládány out-of-order
            - U věci na sobě nezávislých

- Typická organizace postrisc
    - IBM PowerPC 604
        - Použití přejmenovaných registrů; registry podmínkových kódů...
    - MIPS R10000
        - Plně 4cestně superskalární architektura
        - Tři plánovací fronty
        - ...
- Smíšené organizace
    - Intel Pentium P6
        - Instrukční sada CISC interpretován mikrooperacemi na jádře post-RISC
        - Instrukce rozkládány na mikroinstrukce
        - Pipeline provádí mikroinstrukce
        - Třícestnž superskalární návrh
        - Spekulativní provádění
        - Historické dědictví podporovat staré procesory
        - Dekódování makroinstrukcí
            - Maximálně 3 makroinstrukce (= to co leze do procesoru) na jeden cyklus
            - Maximálně 6 mikrooperací na jeden cyklus
            - … viz slide

Dataflow CPU vs. Post-Risc

- Dataflow - architektura umožňuje za optimalizace prostředků množství úloh řešit jednoduše, protože je založena na sekvenci instrukcí, na základě příchozích dat vybírám co provedu.. Tedy řízeno tokem dat, místo toku instrukcí
    - Výsledek jsou zpracovaná data, tak, v jakém pořadí instrukce byly