Architektury CISC a RISC, uplatnění rysů architektur RISC v personálních počítačích - pokračování

Cíl přednášky

- Vysvětlit další rysy architektur CISC a RISC, upozornit na rozdíly.
- Upozornit, jak se typické rysy obou typů architektur projevily v konstrukci PC.

Architektury CISC a RISC – cenové aspekty

- Architektury RISC – výrazná snaha o modifikaci architektury tak, aby vyhovovala požadavkům proudového zpracování – hnací motor zavádění nových rysů do architektur RISC.
- Architektury CISC - snaha o totéž – **využívání technik nasazených v architekturách RISC, zohlednění cenových aspektů**.
- Cenové aspekty (co zvyšuje cenu?):

  technická/technologická náročnost implementace konkrétní techniky,

  náklady na vývoj techniky (lidský a technický potenciál),

  konkrétní technika zvyšuje výkon (výrobce si to nechá zaplatit).
- Výsledek – existují počítače na bázi architektur CISC i RISC:

  CISC – levnější, širší uplatnění na trhu s počítači (mají více potenciálních zákazníků),

  RISC – nákladnější, nasazeny v jiných aplikacích (servery, pracovní stanice pro CAD, …)

CISC a RISC architektury – vývoj koprocesorů

- Další problém: architektury RISC i CISC – nutno řešit podporu instrukcí s pohyblivou čárkou.
- Příklad: procesor SPARC MB86900 a koprocesor MB86910.
- Firma Intel: např. procesor I80486, koprocesor I80487, až do úrovně I80386 byl koprocesor samostatným prvkem - koprocesory jsou implementovány buď jako samostatné prvky (čipy) nebo jsou na stejném čipu jako procesor.
- Výhody druhého řešení:

  Možnost realizovat vyšší rychlosti přenosu a snadnější synchronizace.
- Kromě tzv. „čistých“ architektur RISC existují i takové, kde kromě hardwarově realizovaných instrukcí existovaly/existují i instrukce realizované mikrokódem (bylo k vidění i u dřívějších architektur RISC).

CISC a RISC architektury – vývoj koprocesorů

- Nové generace architektur RISC nepředpokládají použití koprocesoru jako pojmu, využívají jednotku realizující instrukce pohyblivé čárky (FPU – Floating Point Unit) – různá úroveň složitosti FPU.
- Stejná terminologie byla přijata i pro Pentium a vyšší typy procesorů.
- Možná reflexe v architekturách počítačů:

  Existují dvě fronty instrukcí – pro zpracování čísel typu integer a čísel reálných, jedna zpracovávaná procesorem, druhá jednotkou FPU.
- Dvě fronty instrukcí – vznikly zcela nové problémy, např. **párování instrukcí**.
- Rozšiřování množiny funkcí, které byly v FPU k dispozici: kromě klasických aritmetických operací také např. exponenciální nebo trigonometrické funkce.
- FPU je integrována na stejném plátku jako procesor – rozlišení od koprocesoru (odlišení této skutečnosti novým označením).
- Další rozvoj tohoto rysu: existují i architektury s více FPU, např. PowerPC 970, architektury na bázi Intel Netburst Microarchitecture a AMD 64.

Sady registrů (Register Files)

- Jeden z rysů architektur RISC – snaha o co nejmenší počet přístupů do paměti => součástí procesorů RISC je velký počet registrů (architektury CISC – tento problém není řešen):

  omezení komunikace s pamětí při realizaci aritmeticko-logických operací (operand v paměti – nesmí být) – tato snaha má svůj začátek v proudovém zpracování instrukcí (sady registrů nejsou využívány pro aritmeticko/logické operace).
- Počty registrů: od 32 do 2048 a více (?), specifický způsob využití.
- Srovnání: procesory v PC – pouze universální registry pro realizaci aritmeticko-logických operací, destruktivní charakter operací.
- Důležité: technika založená na sadách registrů není k vidění ani u moderních procesorů v PC (cenové důvody – hardware + programová podpora obsluhy registrů), je typická např. pro procesory SPARC (pracovní stanice).

Sady registrů – skoky do podprogramů

- Využití sady registrů při skocích do podprogramů

  Architektura CISC - skok do podprogramu – je nutno uložit obsahy všech registrů do paměti.

  Řešení v architekturách RISC: **využití několika sad registrů**, každá podmnožina registrů odpovídá původní samostatné (jediné) sadě registrů.

  Pracuje se s tzv. **ukazatelem na sadu registrů**, při výskytu instrukce ***CALL*** se ukazatel posune na další pozici a ukazuje tak na další sadu registrů - volnou.

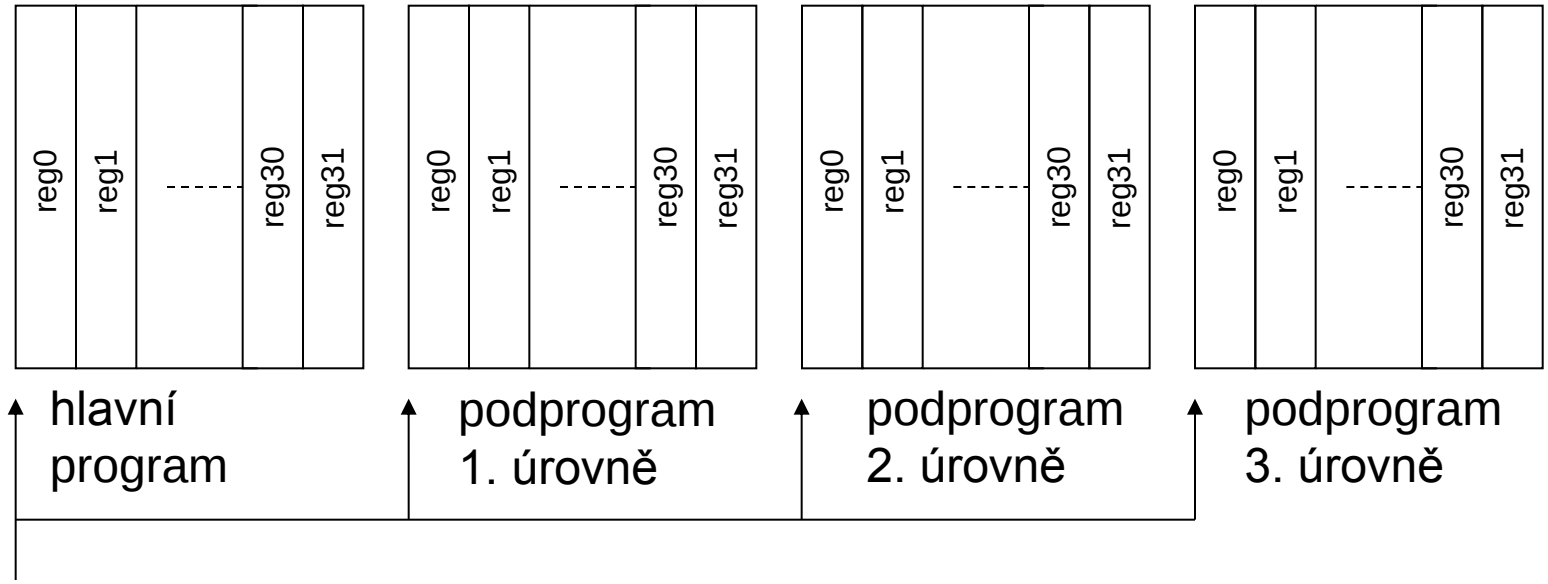
  Instrukce ***RETURN*** (návrat z podprogramu) – hodnota ukazatele se sníží o 1.

  Výsledek: **uklízení hodnot uložených v registrech do paměti není nutné v situacích, kdy počet sad registrů pro konkrétní aplikaci postačuje**.

  Programy, kde se vyskytne **10 úrovní volání podprogramu** – řídký jev => běžně např. **10 sad registrů** (320 registrů).

  Stav, kdy **úrovní je více** – **rekurzivní výpočty** => nastane **přetečení sad registrů** => musí se začít **využívat paměť**.

Sady registrů – skoky do podprogramů



ukazatel na sadu registrů

Sady registrů – skoky do podprogramů

- **Mechanismus přetečení do paměti (sady registrů nestačí svým objemem)**:

  Sada registrů je řízena ukazateli na začátek/konec sady registrů.

  Do paměti se v případě potřeby uloží nejprve obsah té sady registrů, která je označena jako počáteční – při návratech z podprogramů se bude s touto informací pracovat až jako s poslední.

  Obsah sady registrů je přenesen z paměti zpět, jakmile se instrukcemi RETURN (každá z nich realizována v jiné úrovni) vrátíme na úroveň pouze o jeden stupeň vyšší než je úroveň sady registrů, jejíž obsah byl přenesen do paměti.

  Evidentní snaha o to, aby se zmírnil dopad paměťových operací na dobu trvání výrazného počtu činností realizovaných procesorem (omezení komunikace s pamětí) – souvisí s optimalizací proudového zpracování programu – zkrácení doby trvání procesů, pro něž se dříve musela využívat paměť.

Sady registrů – předávání parametrů

- Další problém – předávání parametrů mezi volajícím a volaným programem – přesuny dat.

  Možnost řešení – kopírování parametrů z jedné sady registrů (volající) do jiné (volaný) – nepříliš vhodné.

  Jiné řešení – **překrývání registrových sad**.

  Princip: **množina sad registrů je rozdělena na registry pro uložení vstupních parametrů, lokálních proměnných, výstupních parametrů a globálních parametrů**.

  Výsledek: když se množina registrů pro uložení výstupních parametrů volajícího programu překrývá s množinou registrů pro uložení vstupních parametrů volaného programu => přenosy dat mezi sadami registrů nejsou nutné.

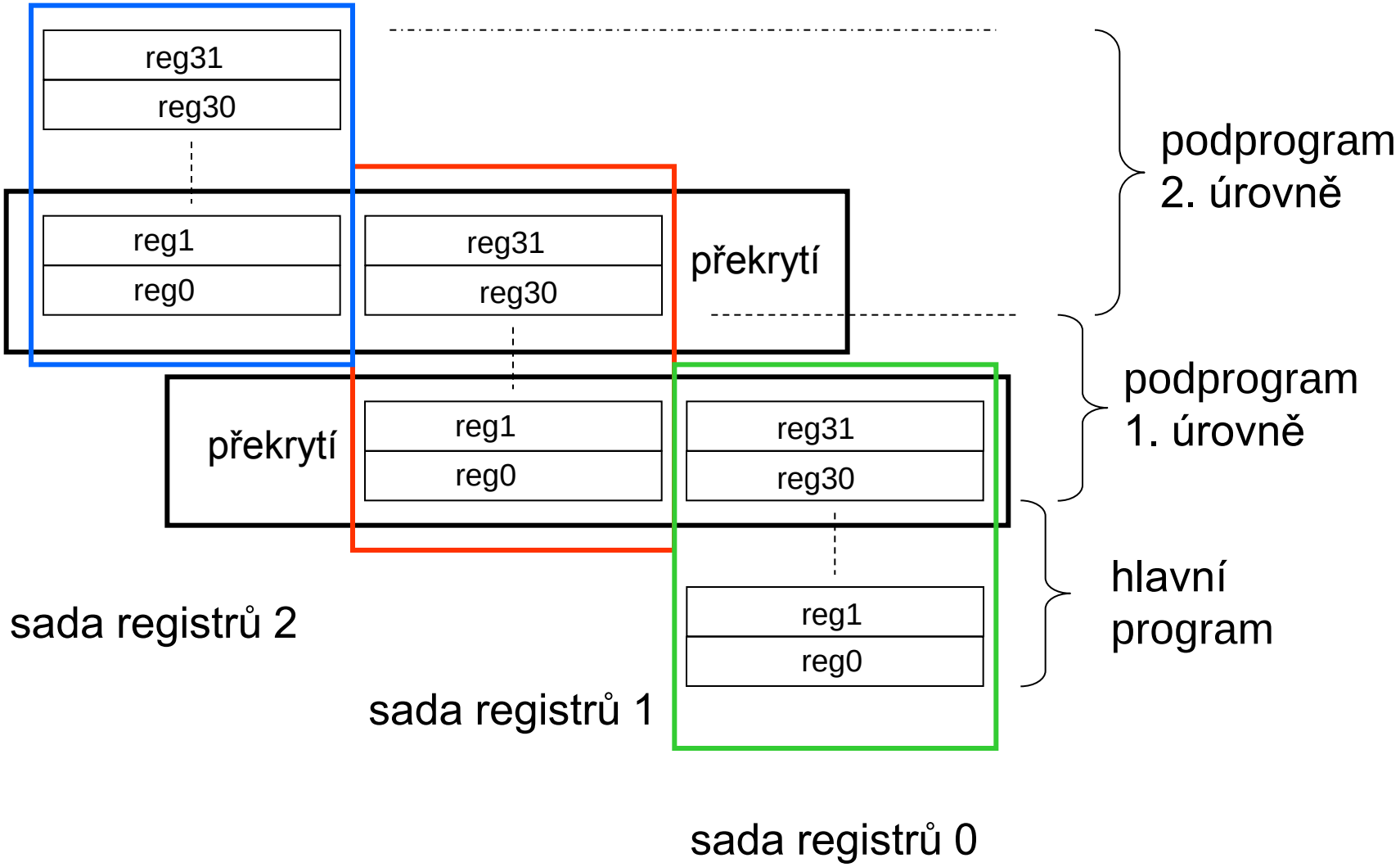
  Obdobným způsobem si předají výsledky volaný a volající program při návratu z podprogramu.

  Tato procedura se nazývá **procedure window method** (okno do sady registrů), okno se přesouvá dynamicky přes celou sadu registrů.

Sady registrů – předávání parametrů

- Stupeň překrývání (tzn. počet předávaných parametrů a proměnných) nemusí být pevný, bude určen při kompilování => **register windows with variable size** (okno do sady registrů s proměnnou délkou).
- **Rys architektur RISC - optimalizace běhu programu prováděná při překladu.**

Sady registrů – předávání parametrů



Využití sad registrů při přepínání úloh

- Klasické řešení: při přepínání úloh se **obsahy všech sad registrů staré úlohy ukládají do paměti**, v opačném směru se přenáší obsah sady registrů úlohy, která bude odstartována.
- Pokud by byl k dispozici dostatečný počet sad registrů, pak by se přepínání úloh mohlo realizovat bez komunikace s pamětí.
- Přepínání úloh se pak odehrává pomocí změny hodnoty ukazatele na aktivní sadu registrů.
- Procesory SPARC – 2048 registrů uspořádaných do sad registrů.
- V procesorech využívaných v PC není tato technika využita, využívá se klasický způsob přepínání úloh.
- Závěr: **technika založená na využívání sad registrů vyřazuje ze hry zásadním způsobem paměť a může výrazným způsobem zrychlit počítač jako celek** – jasný trend v architekturách RISC.

Sady registrů – závěr

- Sady registrů jsou využitelné v těchto třech technikách:
  - volání podprogramu,
  - předávání parametrů,
  - přepínání úloh.

- Techniky založené na využití sad registrů vyřazují ze hry paměť v situacích, kde právě paměť byla důležitou komponentou při realizaci těchto procesů.

Uplatnění principů procesorů RISC v architekturách CISC

- Prvky architektur RISC se začaly jistým (nepříliš výrazným) způsobem uplatňovat v I80486:

  často se vyskytující instrukce nebyly implementovány mikroprogramově ale obvodově (např. instrukce MOV).
- Složité a méně časté instrukce byly implementovány mikroprogramově.
- Důsledek: provedení posloupnosti jednoduchých instrukcí implementovaných obvodově trvá kratší dobu než složitá instrukce realizující totéž mikroprogramově.
- Procesor: integroval do jednoho čipu mikroprocesor, vylepšený koprocesor (ve srovnání s I80387), RVP a řadič RVP – vše bylo motivováno snahou o zvýšení rychlosti a o redukci objemu komunikace s operační pamětí (typické pro architektury RISC).
- Nevýhodné v situaci, kdy operand aritmeticko-logické operace je uložen v paměti (více typů paměti).
- RISC – operand zásadně v registru.

Uplatnění principů procesorů RISC v architekturách CISC

- Procesor I80386 pracoval na kmitočtech 25 – 50 Mhz, procesor ve verzi I80486DX4 na 100 Mhz v nárazovém režimu komunikoval na rychlosti 160 MB/s (rozhraní mikroprocesoru).
- Výsledek z hlediska rychlosti: **díky hardwarové implementaci instrukcí a dokonalejší technice zřetězení na úrovni provádění instrukcí (vyšší granularita) byl procesor I80486 asi 3x rychlejší než I80386** (uvádělo se v době, kdy se objevily počítače na bázi I80486)**.**
- Výrazněji komplikovanější struktura než I80386 – v jednom pouzdře je procesor, koprocesor, řadič RVP a RVP (3 samostatné procesory).
- RVP L1 byla určena pro uložení kódu i dat (architektury RISC – odděleno, u vyšších verzí procesorů Intel rovněž).
- Přístup do RVP L2 – 2 synchronizační pulsy, přístup do RVP L1 – 1 synchronizační puls – účelné z hlediska zřetězeného zpracování instrukcí.
- Kapacita fronty instrukcí byla 32 B.

Zřetězení na úrovni provádění instrukcí v procesoru I80486

- Velmi odlišné doby provádění instrukcí => není naplněn základní požadavek proudového zpracování instrukcí.
- Instruction Fetch – čtení instrukce: čtou se dva 16 slabikové celky.
- Decoding Unit – dekódování instrukce: konvertuje jednoduché instrukce na povely pro CU (Control Unit) a složité instrukce na skoky na patřičný mikrokód, který se pak provádí v CU.
- Prováděcí fáze – jednotka CU interpretuje řídicí signály z dekodéru nebo skoky na patřičný mikrokód a řídí ALU, FPU a ostatní logiku.
- Tato činnost trvá jistou dobu (různě dlouhou) v závislosti na typu instrukce.

Zřetězení na úrovni provádění instrukcí v procesoru I80486

- Důsledek – v architekturách CISC (např. I80486) se velmi obtížně dosahovalo stavu, kdy doba provádění jednotlivých fází realizace operace trvala stejnou dobu – zásadní odlišnost od architektur RISC, kde v každém z X stupňů je jedna instrukce (X instrukcí je současně zpracováváno) – lepší situace nastala v Pentiu a vyšších typech procesorů (rozdělení provádění operací na větší počet kroků).

Charakteristika architektur RISC a CISC z hlediska optimalizace toku instrukcí

- Pro architektury RISC jsou vyvíjeny překladače „šité na míru“, jsou vyvíjeny pro konkrétní procesory (a naopak – vývoj procesoru pro konkrétní jazyk), kvalita obou je důležitá => přeložené kódy jsou optimalizovány.
- Důležitý aspekt – zásah do množiny instrukcí (doplnění o nové instrukce) => musí se změnit struktura CU (control unit – řadič) => nutnost společné tvorby hardware a množiny instrukcí, které tento hardware implementuje.
- Mikroprogramově řízené struktury – je třeba doplnit nový mikroprogram, využije se k tomu stávající množina mikroinstrukcí (nezměněná) => CU není třeba z hlediska struktury měnit.
- Platí: struktura strojového kódu souvisí silně s konkrétním překladačem => možnost úpravy strojového kódu při překladu.

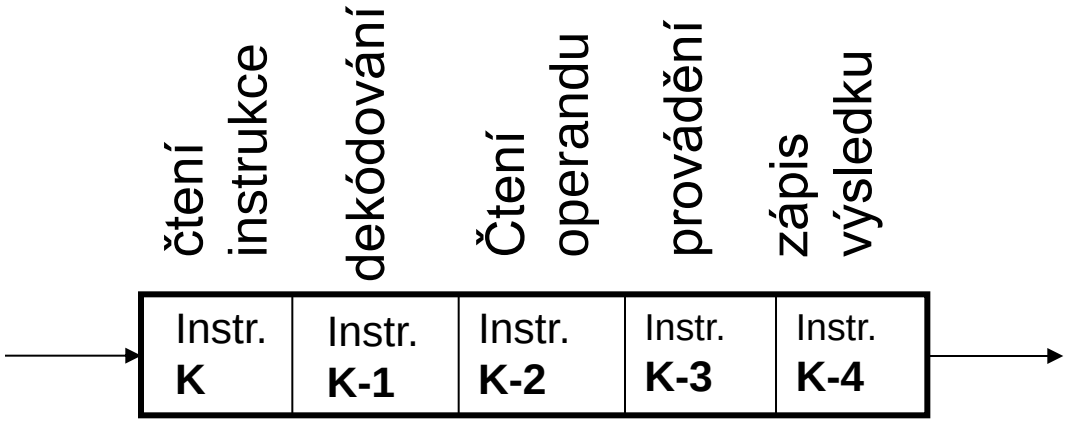
  => vývoj konkrétních RISC procesorů pro konkrétní jazyky: (SOAR – Smalltalk On A RISC), SPUR (Symbolic Processing Using RISC) a COLIBRI (Coprocessor for LISP on the Basis of RISC) – obojí pro LISP (takový přístup není využit pro procesory CISC).
- CISC architektury – optimalizace až při běhu programu.

Charakteristika architektur RISC a CISC z hlediska optimalizace toku instrukcí

- Menší počet instrukcí pro architektury RISC – výstavba překladačů je jednodušší, bez optimalizace by byl ale vygenerovaný kód obsáhlejší (překladač pro procesor RISC vygeneruje obsáhlejší kód než bude kód pro procesor CISC realizující stejnou činnost).
- Cíl optimalizace:
  - využít strukturu procesoru,
  - negenerovat rozsáhlé kódy.
- Způsob, jak se toho dosáhne:
  - optimalizace využití registrů,
  - prevence vazeb mezi instrukcemi a mezi frontami.
- Závěr: Pokud je překladač realizován tak, aby respektoval strukturu procesoru, zohledňoval výše uvedené aspekty, pak výsledný kód bude efektivnější než kód pro architekturu CISC. Tyto trendy klasické CISC architektury nesledují.

Analýza některých situací překladačem RISC a CISC

- Situace, kdy se v posloupnosti instrukcí vyskytuje instrukce podmíněného skoku – dokud se instrukce podmíněného skoku nevyhodnotí (podmínka je/není splněna), není jasné, kde bude provádění programu pokračovat.
- Nejjednodušší řešení – instrukce načtené do fronty se přestanou provádět na jistý počet cyklů (odpovídající počtu stupňů mezi prvním stupněm a prováděcí jednotkou), až se instrukce podmíněného skoku vyhodnotí, bude se pokračovat dál – buď na další adrese nebo na adrese, kam ukazuje skok - počet kroků, po něž se bude čekat, se vyplní instrukcemi NOP – je možno řešit při provádění instrukcí/při překladu.



Analýza některých situací překladačem RISC

- Další možnost – přeorganizovat posloupnost instrukcí tak, aby se počet instrukcí NOP minimalizoval, tzn. optimalizace při překladu.
- Příklad: překladač vygeneroval následující posloupnost instrukcí:

```
ADD r3,r2,r1
AND r0,r5,r6
JMPT r0,label
NOP
.
.
label: sub r1,5,r6
```

- Kvůli instrukci JMPT je nutné do posloupnosti instrukcí vložit instrukce NOP => snaha přeorganizovat posloupnost instrukcí tak, aby instrukce, jejichž výsledek není důležitý pro instrukci skoku, byly přesunuty za instrukci podmíněného skoku => instrukci ADD je možné přenést za instrukci JMPT => zredukuje se počet instrukcí NOP.

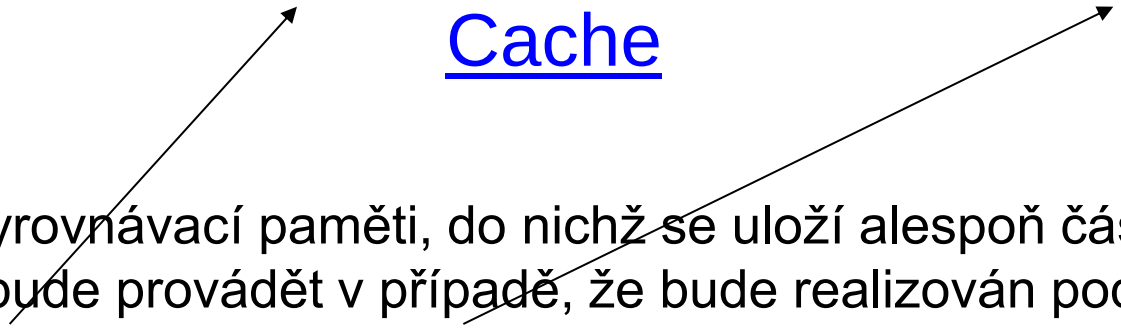
Analýza některých situací překladačem RISC

- Výsledek realizace předcházející úvahy:

```
AND         r0,r5,r6
JMPT        r0,label
ADD         r3,r2,r1
NOP
.
.
label: sub  r1,5,r6
```

- Instrukce ADD neovlivňuje parametr (podmínku) podmíněného skoku JMPT a proto je možné ji přenést za JMPT - **výsledek překladu.**

Pojem Branch Target Cache nebo Jump Target Cache



- Rychlé vyrovnávací paměti, do nichž se uloží alespoň část kódu, který se bude provádět v případě, že bude realizován podmíněný skok (branch) nebo skok (jump) => z RVP se bude tento kód číst a může se začít provádět, mezitím se z RAM přečte zbytek kódu.
- Jiná možnost – zvýšení kapacity fronty instrukcí Instruction Fetch tak, aby obsahovala podstatně vyšší počet instrukcí => zvýší se pravděpodobnost, že ve vstupní frontě bude i instrukce, na niž se bude uskutečňovat skok.
- Další možnost – využití techniky předvídání (využita i v Pentiu – samostatná přednáška).

Shrnutí – některé rysy moderních architektur RISC

- Instrukce jsou prováděny hardwarovými automaty, tzn. nejsou interpretovány mikroprogramem (posloupnostmi mikroinstrukcí).
- Zvyšování rychlosti, s níž jsou instrukce prováděny (úroveň technologie, nové typy architektur).
- Snadné dekódování instrukcí – definovaná délka instrukce, přesně definovaná struktura instrukce, definovaný (nízký) počet operandů.

  Různé formáty instrukcí – zcela špatně (problémy s dekódováním).
- Omezení komunikace s pamětí (pouze instrukce load a store),

  operandem instrukce zásadně není hodnota uložená v paměti.
- K dispozici jsou sady registrů – využití nejenom pro ukládání parametrů.
- Uplatnění paralelismu:

  na úrovni instrukcí (zřetězené zpracování),

  na úrovni procesorů (jeden problém řešen více procesory).

Shrnutí – rysy moderních architektur

- Uplatnění architektur RISC – náročnější aplikace.
- Na architektury RISC jsou kladeny vyšší požadavky než na architektury CISC.
- Některé z uvedených rysů jsou součástí procesorů nasazovaných do PC – cenové aspekty.
- Do ceny je nutné zahrnout nejenom implementační náklady, ale také cenu vývoje konkrétního rysu.
- Závěr: postupné uplatňování rysů architektur RISC v architekturách CISC.