Cíl přednášky:

- Vysvětlit principy práce s registry v architekturách RISC a CISC, upozornit na rozdíly.
- Vysvětlit možnosti využívání sad registrů.
- Zabývat se principy využívanými v procesorech Intel.
- Zabývat se stručně postupy uplatňovanými při tvorbě překladačů kódu pro procesory RISC a CISC.

Další aspekty architektur CISC a RISC

Aktuálnost obsahu registru

- Problém s aktuálností obsahu registru existuje i v architekturách CISC.
- Předpokládejme posloupnost těchto instrukcí:

  ADD reg1,reg2,reg7

  AND reg6,reg1,reg3

  Instrukce *ADD* - má se sečíst obsah registrů *reg2* a *reg7* a výsledek se má uložit do *reg1*.

  Instrukce *AND* – provést logický součin mezi obsahy registrů *reg1* a *reg3* a výsledek uložit do *reg6*.
- Podmínka korektní realizace instrukce *AND* – obsah *reg1* musí být platný – pozor: **zápis výsledku se odehrává až v poslední fázi provádění instrukce**, **instrukce *ADD* je v tom okamžiku ve fázi provádění a obsah *reg1* nemusí být ještě zapsán.**
- Tento problém je řešitelný metodami již popsanými (Berkeley, Stanford).
- Stanford – vkládání NOP

  ADD reg1,reg2,reg7

NOP

NOP

AND reg6,reg1,reg3

- Posouzení výkonnosti – hodnota MIPS nezachytí tento stav, protože *NOP* je také instrukce – je potřeba takový algoritmus posuzovat podle **efektivity výpočtu** (jakoby instrukce NOP nebyly zařazeny do posloupnosti instrukcí).
- **Problém správného obsahu registru se umocňuje u superskalárních procesorů (Pentium)** – tento problém může vzniknout nejenom mezi fázemi v rámci jedné fronty ale také mezi fázemi různých front.
- **Závěr**: uplatnění principů zřetězeného zpracování způsobuje vznik problémů, které je nutno následně řešit.

Formáty instrukcí procesorů RISC, srovnání s CISC

- Výrazně kratší instrukce, všechny mají stejnou délku.
- Kódování registrů v RISC procesorech: 8 registrů – 3 bity.

- Kódování registrů v CISC procesorech (v začátcích): 8 registrů – 8 bitů (každý registr je reprezentován jedním bitem) – při malém počtu registrů snadno realizovatelné (méně náročné dekódování).
- Instrukce RISC – nedestruktivní (3 registry – 2 operandy, výsledek => hodnota žádného operandu se nepřepíše).
- Nedestruktivní (2 zdrojové registry *src1*, *src2*a jeden cílový registr *dest*).

  Pro všechny registry musí být v binární formě instrukce pozici.

  Registry jsou adresovány např. 5 bity – 32 registrů – v binární reprezentaci instrukce mají registry *dest, src1, src2* pevnou pozici.

  Všechny instrukce mají stejnou délku – např. dvouregistrová instrukce *MOV dest,src1* bude mít nulové pole odpovídající druhému operandu, platí to i o instrukci *NOP* – bude mít všechna pole volná.
- Instrukce CISC – obsáhlá instrukce se značným množstvím informace (parametrů).

  Přeložené instrukce mikroprocesoru I80386 měly různou délku: 1 – 15 slabik.

Takový složitý kód je možné rozumným způsobem realizovat pouze kombinací mikroprogramu a hardwarových dekodérů.

Dekódování takového typu instrukce je náročné na podporu (hardware i mikroprogram), je ale i časově náročné.

- **Závěr:** principy konstrukce instrukcí pro procesory RISC jsou výrazně jednodušší (průhlednější) ve srovnání s principy využívanými v architekturách CISC – to má další pozitivní dopady při konstrukci obvodů procesorů RISC.

Vývoj architektur RVP v souvislosti s existencí architektur RISC

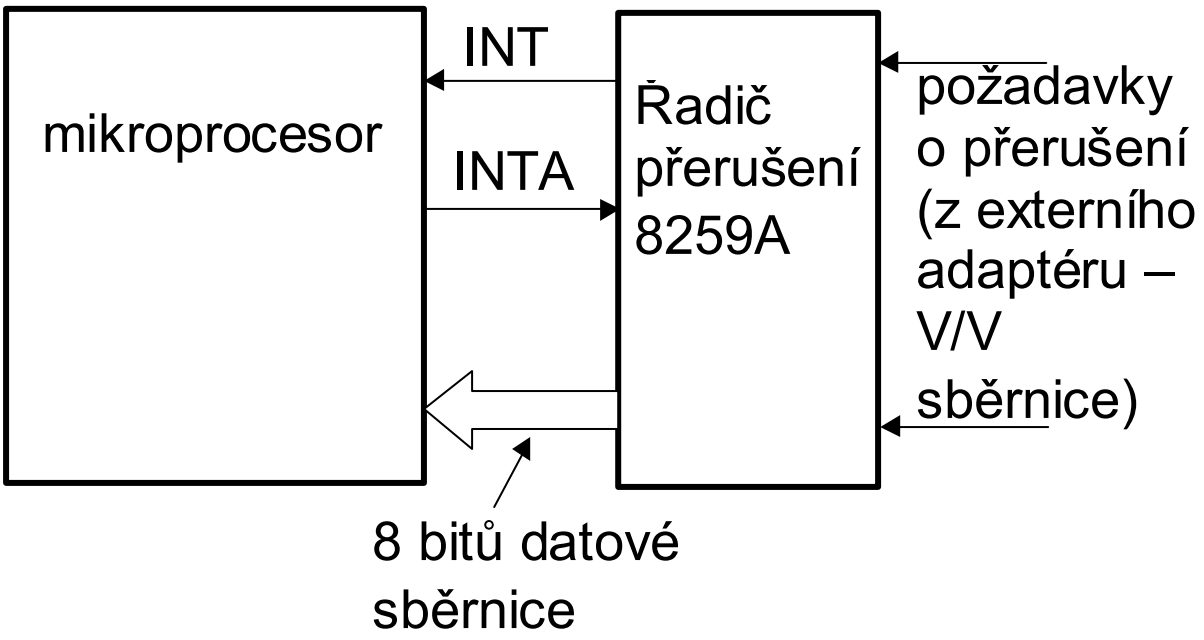
- Hlavní (operační) paměť je v architekturách RISC chápána jako „úzké místo“ – platí to i o architekturách CISC.
- Řešení: RVP nejprve L2 (mimo procesor), později zákonitě L1 (on-die – na stejném čipu jako procesor, in-package – ve stejném pouzdru jako procesor).
- RVP typu L1 je zákonitě rychlejší než RVP typu L2 (zásadní výhoda – komunikace mezi

procesorem a RVP L1 může být synchronizována vyššími kmitočty).

- Procesory RISC mají dvě oddělené RVP: pro kód a pro data (totéž uplatňuje i firma Intel).
- Čím více typů RVP, tím větší problém může nastat s *konzistencí,* (zajištěním informace o tom, obsah které RVP je správný – aktuální).

Vývoj architektur řadičů přerušení v souvislosti s architekturami RISC

- Stav, který existoval ještě v I80486: řadič přerušení 8259A.
- Architektura typická pro CISC:



Obr. 1 Princip generování a obsluhy žádosti o přerušení v architektuře CISC – do úrovně I80486

- Řadič přerušení sdružuje požadavky na přerušení od všech možných zdrojů – pokud je jich více, tak rozhodne, který bude mít prioritu.
- Do procesoru pak řadič přerušení generuje signál INT, procesor odpovídá dvěma cykly INTA směrem do řadiče přerušení, během druhého vloží řadič přerušení na datovou sběrnici 8 bitový vektor přerušení, které bylo vybráno pro obsluhu.
- Řadič přerušení 8259 byl vyvinut na míru sběrnici ISA, tzn. v sestavách, kde je sběrnice ISA, tam je k nalezení.
- **V souvislosti s architekturami RISC – snaha o zabudování řadiče přerušení do procesoru**

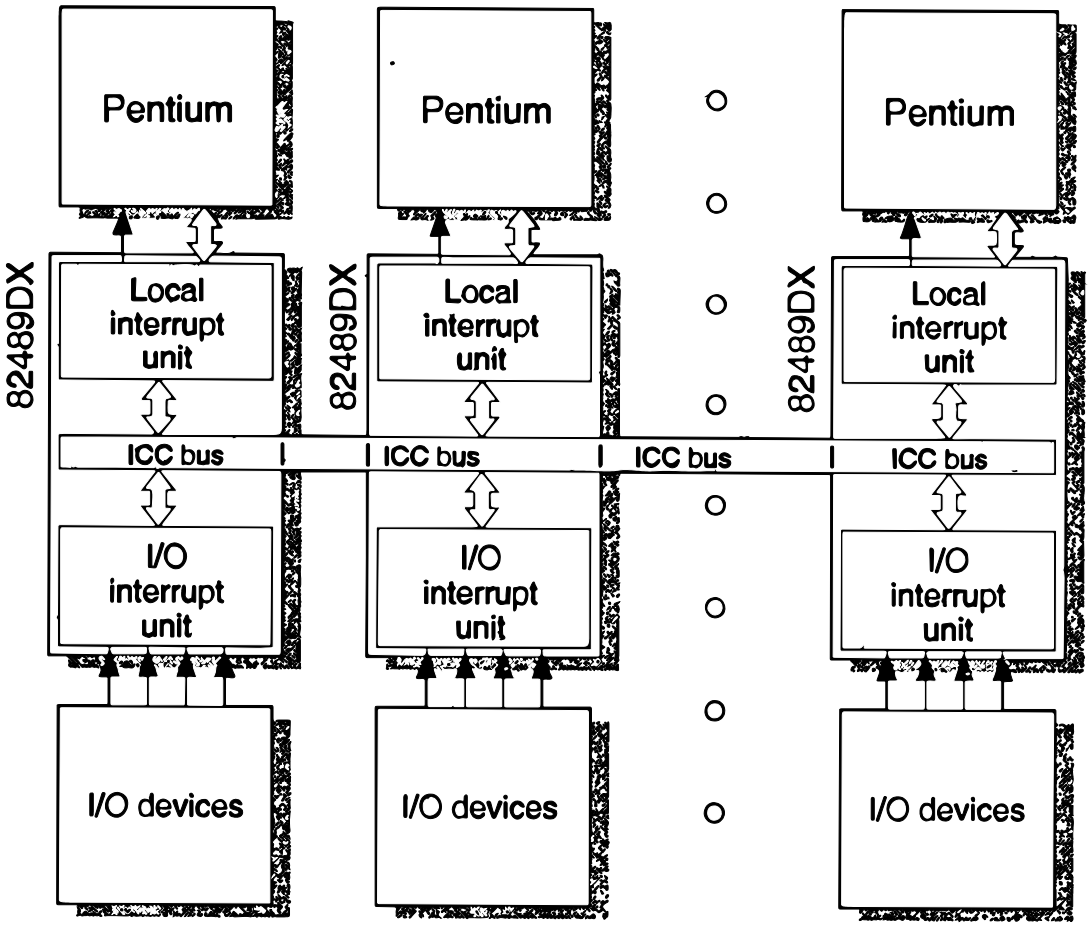
(již v době, kdy personální počítače byly stále budovány na sběrnici ISA) – zřejmá snaha o zrychlení počítače.

- INTEL – byl vyvinut nový typ řadiče přerušení – 82489DX, obsluha požadavků o přerušení se realizuje výrazně rychleji.

Řešení problému přerušení v Pentiu

- Po více jak 10 letech využívání řadiče přerušení 8259A (označovaný jako PIC - Programmable Interrupt Controller) vyvinula firma Intel řadič přerušení **Advanced Programmable Interrupt Controller – APIC 82498DX**.
- Řadič 8259A byl synchronizován kmitočtem 8 Mhz, řadič 82498DX byl v začátcích vyráběn pro frekvence **33 MHZ**, později **55 Mhz** (v současnosti i pro vyšší frekvence) – **rozdíl oproti frekvencím procesorů** (to je výrazný inovační krok ve srovnání s řadičem přerušení ve sběrnici ISA).
- Všechny vnitřní registry tohoto řadiče jsou 32 bitové (na rozdíl od 8 bitových registrů řadiče 8259A).
- Firma Intel doporučuje, aby registry APIC byly mapovány do adresového prostoru operační paměti – zvýší se tím výkon.

- Řadič APIC 82498DX je vyráběn v pouzdře se 132 vývody.
- Architektura řadiče 82498DX je založena na alternativě použití tohoto prvku v multiprocesorových aplikacích.



Obr. 2 Využití řadiče přerušení APIC 82498DX v multiprocesorové aplikaci

- V řadiči je **local interrupt unit** (lokální jednotka přerušení) a **I/O interrupt unit** (V/V jednotka přerušení).

- Lokální jednotka je napojena na lokální („své“) Pentium, V/V jednotka přerušení je přes sběrnici ICC spojena s ostatními lokálními jednotkami přerušení.
- Pokud není lokální jednotka přerušení v okamžiku vzniku přerušení volná, může V/V jednotka svůj požadavek na přerušení **přenést na jinou než svou lokální jednotku přes sběrnici ICC**.
- V takové konfiguraci není procesor obsluhující požadavek na přerušení přerušen, pokud vznikne další požadavek na přerušení v jeho řadiči přerušení, ten je obsloužen v jiném procesoru (jsou pochopitelně zvažovány priority).

Vývoj architektur koprocesorů

- Architektury RISC potřebují stejně jako architektury CISC mít vyřešenu podporu instrukcí s pohyblivou čárkou.
- Příklad: procesor SPARC MB86900 a koprocesor MB86910.
- Firma Intel: např. procesor I80486, koprocesor I80487.
- 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 existují i instrukce realizované mikrokódem.
- Nové generace architektur RISC **nepředpokládají použití klasického**

koprocesoru, využívají jednotku realizující instrukce pohyblivé čárky (FPU – Floating-point Unit).

- **Stejná terminologie byla přijata i pro Pentium**.
- Výsledek:

  Existují dvě fronty instrukcí – pro zpracování čísel typu integer a čísel reálných, jedna zpracovávaná procesorem, druhá jednotkou FPU.

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).
- Počty registrů: od 32 do 2048 a více (?), všechny mají stejné vlastnosti.
- Srovnání: I80386 měl 7 registrů, navíc destruktivní charakter operací.
- Důležité: technika založená na sadách registrů není k vidění ani u Pentia (cenové důvody – hardware + programová podpora obsluhy

registrů), je typická např. pro procesory SPARC (pracovní stanice).

- Obr. 7a) jednoduchá *sada registrů*:

  32 registrů (princip Stanford) – bylo využito v procesorech R2000, R3000.

- Jistá nevýhoda: skok do podprogramu – je nutno uložit obsahy všech registrů do paměti, *sada registrů* musí být k dispozici podprogramu.
- Řešení: **využití několika *sad registrů***, každá podmnožina registrů odpovídá původní samostatné sadě registrů (obr. 7b).
- 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: **ukládání hodnot uložených v registrech do paměti není v těchto situacích nutné.**
- Programy, kde se vyskytne 10 úrovní volání podprogramu – řídký jev => běžně 10 *sad registrů* (320 registrů).

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

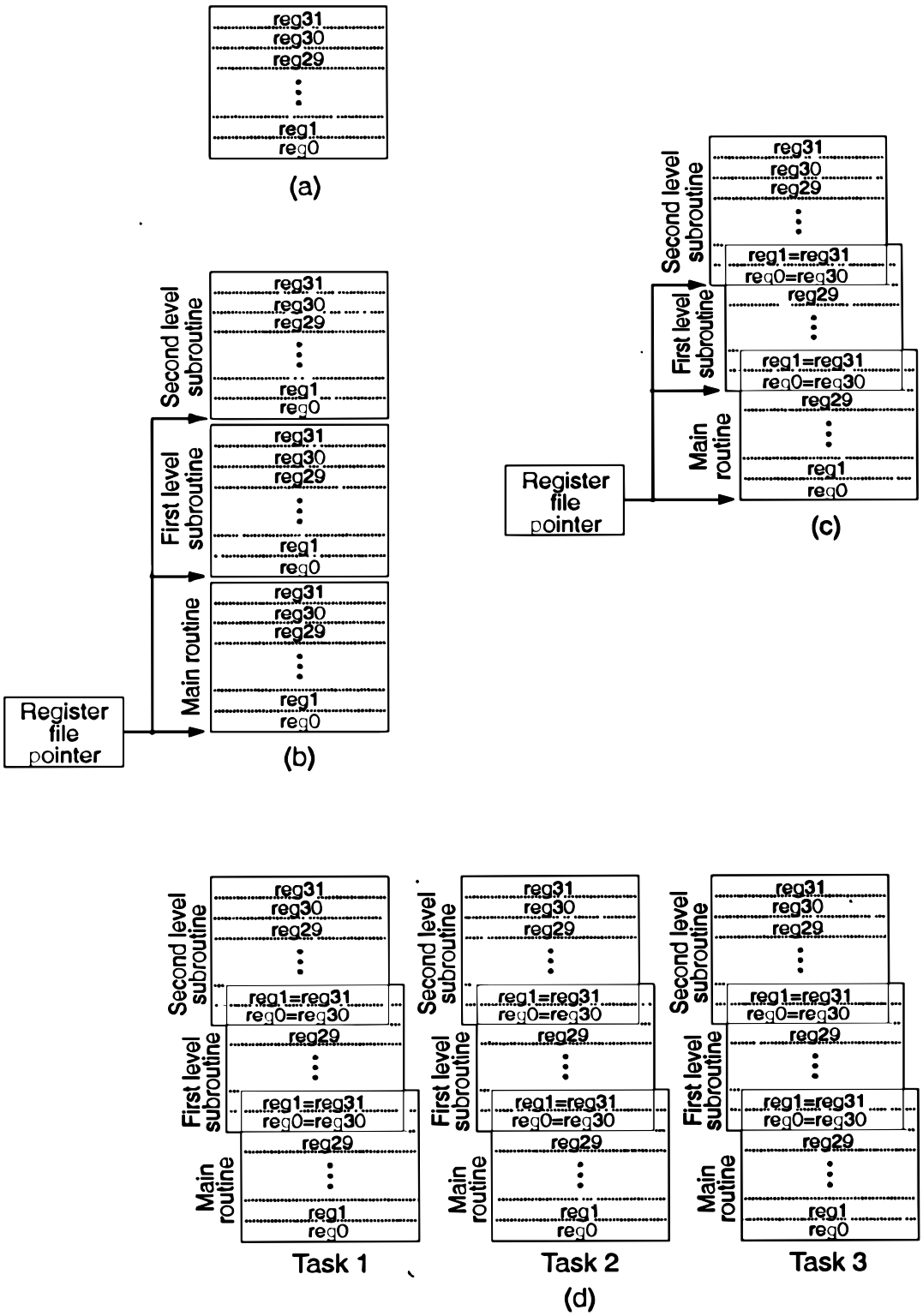
  Sada registrů je řízena ukazateli na začátek/konec *sady registrů*.

  Do paměti se v případě potřeby uloží obsah té *sady registrů*, která je označena jako počáteční.

  Obsah s*ady 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).**
- Další problém – **předávání parametrů mezi volajícím a volaným programem**.
- Možnost řešení – kopírování parametrů z jedné sady registrů (volající) do jiné (volaný) – nepříliš vhodné.
- Jiné řešení – překrývání *registrových sad* (obr. 3c).

- Princip: množina *sad registrů* je rozdělena na registry pro uložení **vstupních parametrů**, **lokálních proměnných**, **výstupních parametrů** a **globálních parametrů**.
- Výsledek: pokud se zajistí, že se množina registrů pro uložení výstupních parametrů volajícího programu překrývá s množinou registrů pro uložení vstupních parametrů volaného programu => není nutné provádět přenosy dat mezi *sadami registrů*.

  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*.
- 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).



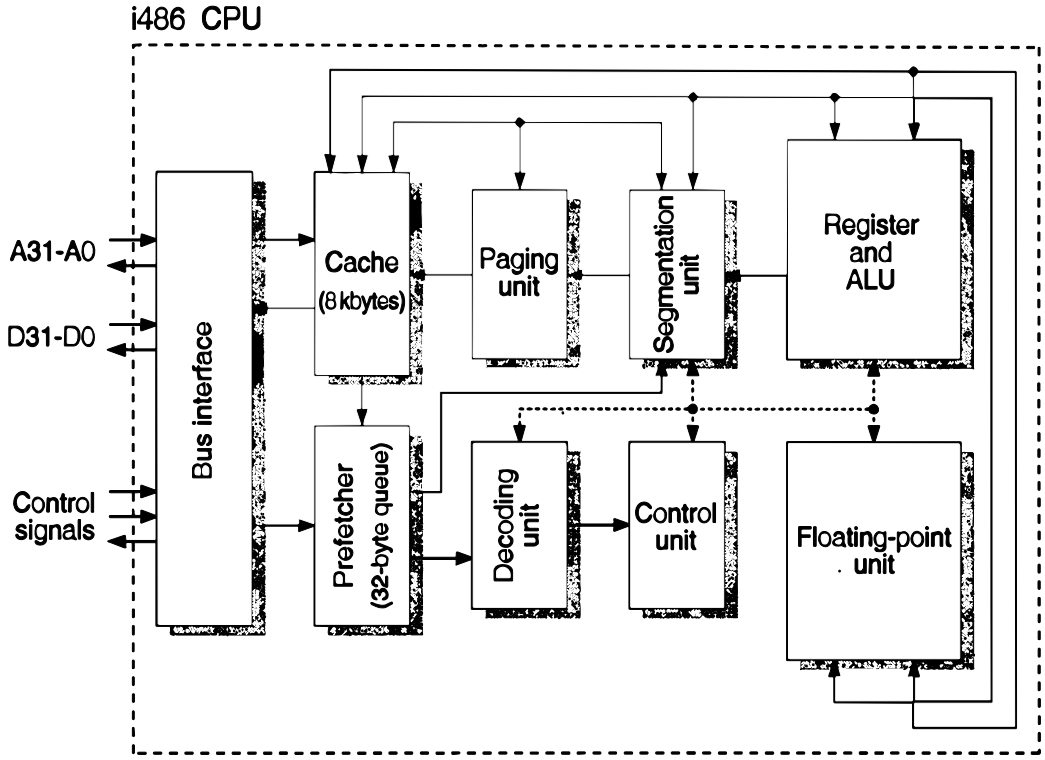
Obr. 3 Příklady *sad registrů* v architekturách RISC

Využití *sad registrů* při přepínání úloh

- 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 – mají 2048 registrů uspořádaných do mnoha *sad registrů*.
- **V procesorech Intel včetně Pentia 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 zásadním způsobem zrychlit počítač jako celek.**

Uplatnění principů procesorů RISC v architekturách CISC

- Prvky architektur RISC se začaly jistým (nepříliš výrazným) způsobem uplatňovat v I80486.
- Často se vyskytující instrukce nejsou implementovány mikroprogramově ale obvodově (např. instrukce MOV).
- Složité a méně časté instrukce jsou realizovány mikroprogramově.
- Důsledek: provedení posloupnosti jednoduchých instrukcí implementovaných obvodově může trvat kratší dobu než složitá instrukce realizující totéž mikroprogramově.
- Procesor integruje do jednoho čipu mikroprocesor, vylepšený koprocesor (ve srovnání s I80387), RVP a řadič RVP – vše motivováno snahou o zvýšení rychlosti a o redukci objemu komunikace s operační pamětí (typické pro architektury RISC).



Obr. 4 Vnitřní struktura mikroprocesoru I80486

- Procesor I80386 byl k dispozici 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 zřetězení na úrovni provádění instrukcí byl procesor I80486 asi 3x rychlejší než I80386.

Vnitřní struktura I80486

- Výraznější komplikovanější struktura než I80386 – v jednom pouzdře je procesor,

koprocesor, řadič RVP a RVP (3 samostatné procesory) – obr. 8.

- RVP L1 je 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.
- Kapacita fronty instrukcí je 32 B.

Zřetězení na úrovni provádění instrukcí v procesoru I80486

- Velmi odlišné doby provádění instrukcí => těsnější propojení jednotlivých stupňů nebylo možné.
- Těsné propojení jednotlivých stupňů (rys architektur RISC) – objevilo se až na úrovni Pentia.
- **Instruction Fetch** – čtení instrukce: čtou se dva 16 bytové 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.
- Důsledek – **v architekturách CISC (např. I80486) se velmi obtížně dosahuje stavu, kdy doba provádění jednotlivých fází realizace operace trvá stejnou dobu** – zásadní odlišnost od architektur RISC (obr. 4), kde v každém z 5 stupňů je jedna instrukce (5 instrukcí je současně zpracováváno) – takto je to řešeno až v Pentiu.

Charakteristika architektur RISC a CISC z hlediska optimalizace toku instrukcí

- Pro architektury RISC jsou vyvíjeny specielní překladače, jsou vyvíjeny pro konkrétní procesory, kvalita obou je důležitá => přeložené kódy jsou optimalizovány.
- Důležitý aspekt – zásah do množiny instrukcí (doplnění o nové instrukce) znamená, že se musí 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.
- Obecně platí (je statisticky dokázáno):
  - struktura strojového kódu souvisí silně s konkrétním překladačem.
- Pro konkrétní situace bylo zjištěno, že:
  - 10 nejčastěji používaných instrukcí ve Fortranu tvoří pak 60% běhu programu,

- překladač COBOLu – obdobný počet instrukcí tvoří 8% běhu programu.

=> to vedlo k vývoji konkrétních RISC procesorů pro konkrétní jazyky (*SOAR – Smalltalk On A RISC)* pro Smalltalk, *SPUR* (*Symbolic Processing Using RISC*) a *COLIBRI* (*Coprocessor for LISP on the Basis of RISC)* – obojí pro LISP.

- Pro procesory CISC to pochopitelně neplatí – překladače se vyvíjejí pro procesory, nikoli naopak.
- 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:
  - eliminace instrukcí, které nemají efekt,
  - analýza smyček a jejich odstranění, pokud je to možné,
  - 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 (včetně architektur procesorů Intel) nesledují.

Analýza některých situací překladačem RISC

- 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.
- Pětistupňová fronta – provádění dalších instrukcí se musí zpozdit o 4 cykly.
- Další možnost – přeorganizovat posloupnost instrukcí tak, aby se počet instrukcí NOP minimalizoval, tzn. optimalizace při překladu.
- Příklad: bez optimalizace by 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*.

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 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.