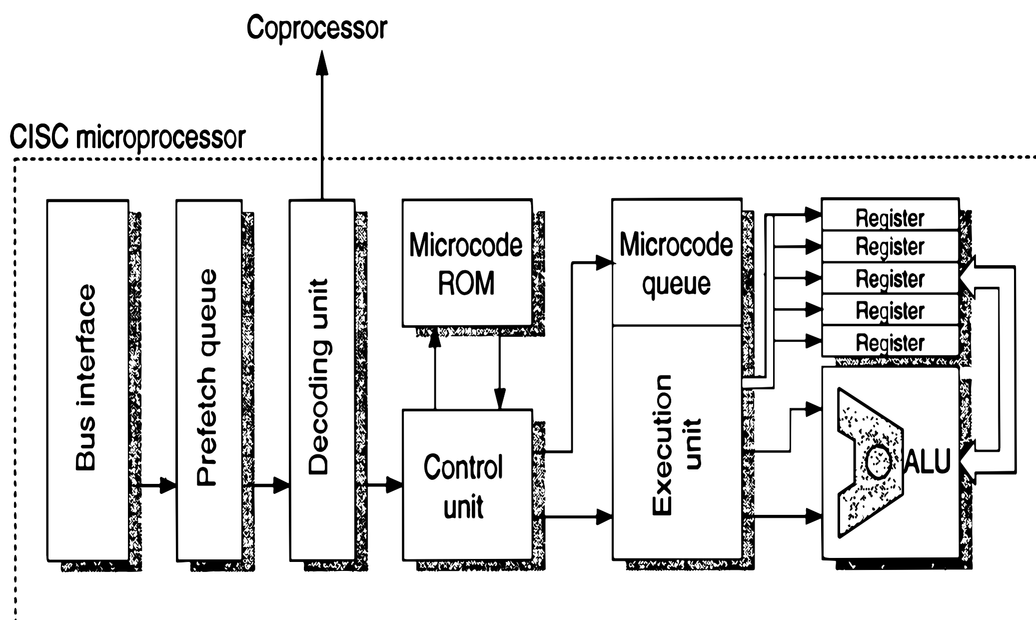
Cíl přednášky

- Seznámit se s charakteristickými rysy architektur CISC a RISC.
- Ukázat, jak tyto rysy postupně pronikaly do architektur procesorů Intel.
- Ukázat, jak se vyvíjely principy zřetězeného zpracování.
- Ukázat na problémy, které přitom vznikaly a jak byly řešeny.
- Zabývat se principy architektur dnešních procesorů.

CISC A RISC PROCESORY

Jak pracují procesory CISC:

- Výraznou vlastností procesorů CISC (Complex Instruction Set Computer) je existence **paměti mikroprogramů**, v níž jsou uloženy mikroprogramy jednotlivých instrukcí.
- Procesor vykonává rozsahem složité **instrukce** (complex), ty **jsou implementovány formou mikroprogramu**.



Obr. 1 Mikroprocesor CISC řízený mikroprogramem

- Posloupnost činností při provádění instrukce:
  1. **Čtení instrukcí** do *fronty instrukcí* (Prefetch Queue).

2. **Dekódování instrukcí** v dekodéru (Decoding Unit).
3. **Provedení instrukce** = provedení mikroprogramu sestávajícího z mikroinstrukcí.
4. Mikroinstrukce jsou **řazeny do *fronty mikroinstrukcí*** (Microcode Queue).
5. Mikroinstrukce pak vstupují do *prováděcí jednotky* (Execution Unit).

- Vyvolání mikroprogramu – řídicí jednotka (control unit) na základě informace z dekodéru (tato informace reprezentuje konkrétní instrukci).
- Všechny tyto činnosti je možné považovat za přípravné činnosti, které jsou poměrně časově náročné a srovnatelné s dobou provádění instrukce.
- **Každé instrukci odpovídá posloupnost mikroinstrukcí uložená v paměti ROM**.
- Role jednotky Prefetch Queue:
  - Při zpracovávání instrukce *n* v Execution Unit (EU) je v Prefetch Queue již načtena instrukce *n+1* a v dekodéru probíhá její rozdekódování na množinu budoucích příznaků, na jejichž základě bude

v následujícím kroku zahájeno provádění jiného mikroprogramu.
- Čím vyšší úroveň rozpracování instrukce, tím více „práce“ procesoru přijde nazmar.

- Možnost realizovat mnohé činnosti automaty – rys architektur RISC.

- Závěr:
  1. Jednotka EU (Execution Unit) řídí provádění mikroinstrukcí čtených z paměti ROM (konkrétní mikroprogram je dán kódem instrukce), na realizaci se podílí ALU, registry procesoru, ……
  2. Pokud by měly být tyto funkce realizovány obvodově (hardwired), tzn. z logických prvků, pak by taková struktura byla enormně složitá (nahrazovala by realizaci posloupnosti mikroinstrukcí).
  3. **Realizace instrukce mikroprogramem je vždy pomalejší než realizace instrukce logickými obvody** (platí obecně, že hardwarová realizace funkce je rychlejší než realizace cestou mikroprogramu) – režie s realizací mikroprogramu je vysoká.

4. Dnešní stav: většina dnešních mikroprocesorů je řízena „částečně“ mikroprogramově.
5. Stav v Pentiu: **jednoduché instrukce** jsou realizovány **obvodově**, **složité instrukce** jsou realizovány **mikroprogramem**.

- Výhoda řízení procesoru mikroprogramem: přechod na vyšší verzi mikroprocesoru – doplnění o nové instrukce a vytvoření nových mikroprogramů reprezentující činnosti, které mají tyto nové instrukce realizovat (**nová instrukce = nový mikroprogram**) - **extensivní rozvoj.**.
- Mikroprocesory CISC se proto vyvíjely především tak, že se rozšiřovaly množiny instrukcí a vytvářely se jim odpovídající mikroprogramy – extensivní rozvoj.

Možnosti, jak zvýšit výkon počítačů využívajících mikroprocesory CISC

Využití fronty instrukcí – řešení problémů s instrukcemi skoku (podmíněný i nepodmíněný)

- Existence instrukční fronty – Prefetch Queue – čtení instrukcí se neodehrává v rámci provádění instrukce ale je to samostatná fáze.
- I80386 – četlo se vždy celé dvouslovo (4 slabiky) z adresy, která reprezentuje začátek dvouslova.
- **Přístupy do paměti kvůli čtení operandů** mají přednost před čtením instrukcí - čtení instrukcí se pozastaví.
- **Výskyt instrukce CALL nebo JUMP** – posloupnost provádění instrukcí se musí změnit => fronta instrukcí se vymaže, z paměti se musí načíst nové instrukce, teprve pak je rozdekódována => zpomalení.
- Skoková instrukce se neuskuteční, pokud není podmínka podmíněného skoku splněna, pak se pokračuje v původní posloupnosti bez jakéhokoliv zpoždění.

- **Možnost předvídání** jak dopadne vyhodnocení podmínky, která je součástí skokové instrukce.
- **Řešení – vyšší kapacita fronty instrukcí** – zvýší se pravděpodobnost, že skok bude mířit na adresu paměti, jejíž obsah je již ve frontě instrukcí.
- **Zásadní nevýhoda: mikroprogramově řízený procesor je pomalejší než obvodově realizovaný procesor (hardwired).**

Prvky typické pro RISC

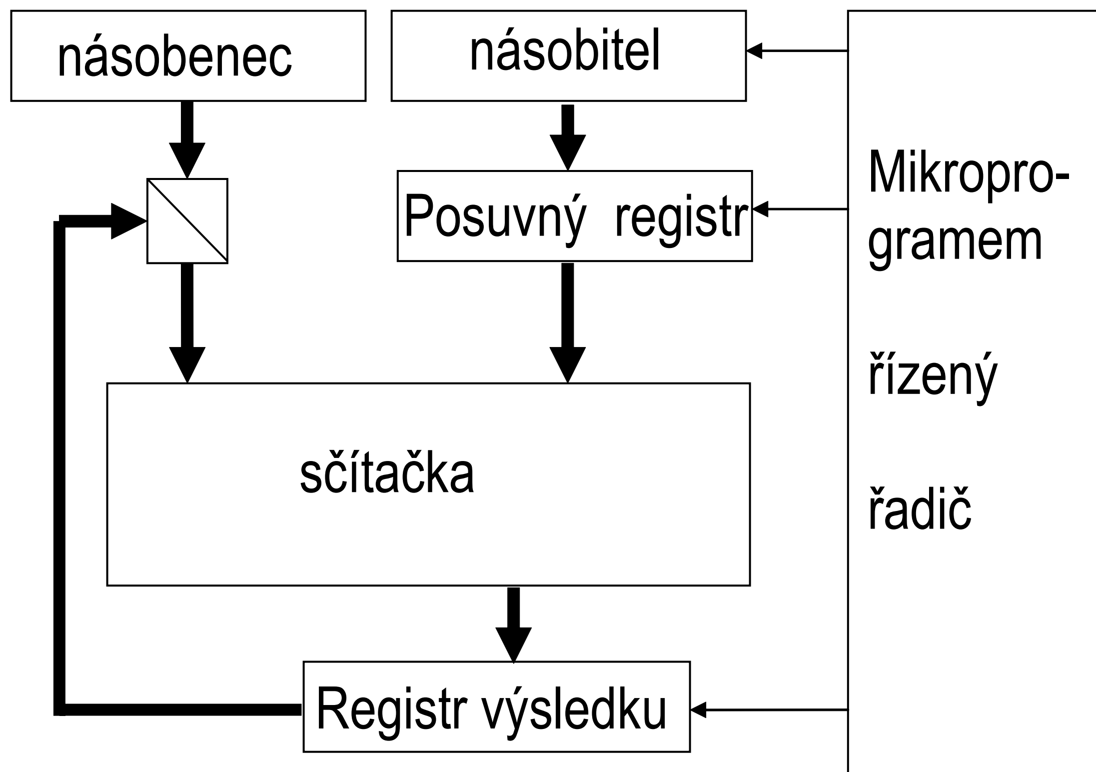
- Procesory CISC: neustále se zdokonalující technologie výroby integrovaných obvodů a také pamětí – do paměti ROM bylo možné vkládat mikroprogramy s narůstajícím rozsahem.
- Mikroprogramově řízené procesory – byl nastaven trend tvorby komplexních instrukcí, jim odpovídaly složité mikroprogramy.
- Důsledek: stav, kdy bylo nutné zvětšovat kapacitu ROM paměti se stal v jisté fázi neúnosný.
- Paměti ROM jsou prvky, které měly horší rychlostní parametry než paměti RAM (u

prvních verzí počítačů na bázi procesorů Intel řešeno stínováním paměti).

- Zjistilo se navíc, že na zpracování 20 % nejčastěji používaných instrukcí se spotřebuje 80 % strojového času => vznikla otázka optimalizace těchto instrukcí.
- Na době potřebné pro provedení instrukce se výrazným způsobem podílí doba na **dekódování** instrukce – u složitých instrukcí se to stává důležitým aspektem (dekódování složitých instrukcí je časově náročnější, několik stupňů dekódovacích obvodů).
- Základní vlastnost architektur RISC:
  1. Redukce počtu instrukcí.
  2. Zřetězení provádění instrukce.
  3. Složité a rozsáhlé instrukce (co do počtu kroků) neexistují, jsou nahrazeny jednoduššími instrukcemi.
  4. Omezení komunikace s pamětí.
  5. Implementace těchto instrukcí je realizována logickými obvody – např. sekvenčním automatem (výrazně rychlejší alternativa než mikroprogram).
  6. Nedestruktivní zpracování operandů.

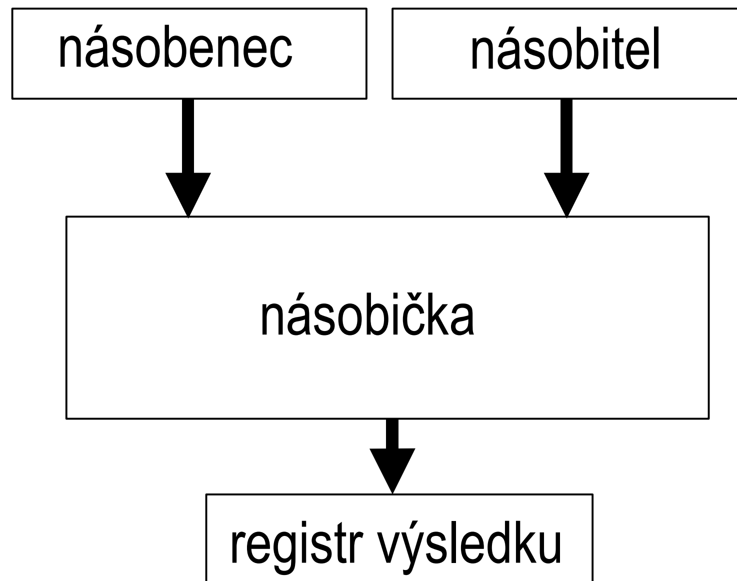
- Realizace násobení dvou operandů typu „integer“ v architekturách CISC a RISC.

- Násobení v procesoru CISC:
  1. Násobení je možné v procesoru CISC realizovat tak, že jeden operand (násobitel) posuneme o jednu pozici doleva a přičteme jej k mezivýsledku, pokud je odpovídající bit násobitele roven 1 – pokud je rovna 0, provede se další posun doleva.
  2. Doba potřebná pro realizaci instrukce bude záviset na tom, kolik bitů v násobiteli bude mít hodnotu rovnu 0 (tzn. nebude pevná délka provádění instrukcí).
  3. Pro realizaci násobení dvou operandů bude potřebná sčítačka, patřičný počet registrů a řadič řízený mikroprogramem provádějící jednotlivé kroky násobení.
  4. Pro každý krok podle bodu 1) bude realizováno několik mikroinstrukcí.



Obr. 2 Realizace násobení v mikroprocesoru CISC

- Násobení v procesoru RISC:
  1. Místo sčítačky se využívá specielní obvod (násobička), v němž je operace násobení realizována hardwarově a rychleji.
  2. Doba potřebná pro realizaci násobení v násobičce se vždy provádí stejně dlouho, tzn. stejným počtem kroků (cyklů).
  3. To je výhodné, protože při řetězení instrukcí je dobře, když se v okamžiku zahájení nějaké činnosti přesně ví, kdy skončí.



Obr. 3 Realizace násobení v mikroprocesoru RISC

- Odraz těchto principů v architekturách procesorů Intel:

  Procesory I8086 – I80386: převažovala implementace mikroprogramem uloženém v paměti ROM.

  Nevýhoda: proces, v němž byla ve hře paměť ROM, je od samého začátku pomalý.

  Doba procesorů I80286 – I80486: vybavovací doba RAM – 60 ns, ROM asi 200 ns.

  Řešení: stínování ROM – obsah ROM se při zavádění systému přenesl do paměti RAM.

  Dnes ROM BIOS je v paměti Flash – vybavovací doba 15 – 20 ns (první cyklus je výrazně pomalejší – kolem 100 ns), principielně je to paměť EEPROM (electrically

erasable programmable read only memory), energeticky nezávislá.

Se zvyšujícím se tlakem na zřetězené zpracování instrukcí (stejná doba trvání násobení nezávislá na hodnotách operandů) – realizace procesu násobení pomocí hardwarových automatů.

Nedestruktivní operace

- RISC procesory jsou tzv. nedestruktivní, což znamená, že obsahy paměťových míst, v nichž jsou uloženy operandy, se nemění.
- Stav v I80386 (CISC) – pokud operandy jsou registry nebo paměťová místa, pak jsou jejich obsahy provedením instrukce likvidovány.

  Příklad: ADD *eax*, *mem32*

  Tato instrukce přičte obsah *mem32* k obsahu akumulátoru *EAX* a výsledek uloží zpět do *EAX* => původní hodnota uložená v *EAX* se přepíše novou hodnotou.

  Totéž platí pro instrukce s operandy registr – registr, registr – paměť nebo paměť – paměť.

  Důsledek: v konečném dopadu vyšší objem komunikace s pamětí.

- Architektury RISC – nedestruktivní režim, operandy zůstanou po provedení instrukce zachovány => zkratka RISC je mnohými překládána jako *Reusable Information Storage Computer.*
- Důsledek – operandy mohou být používány opakovaně a nemusejí se tudíž znovu přenášet z paměti.
- Běžně pracují procesory RISC procesory se 3 operandy, dvěma operandy reprezentujícími vstup a jedním pro uložení výsledku.

  Příklad: *ADD dest, src1, src2*

  Dest – destination (určení)

  Src – source (zdroj)

  Procesor RISC touto instrukcí sečte dva operandy *src1*, *src2* a výsledek uloží do *dest*.

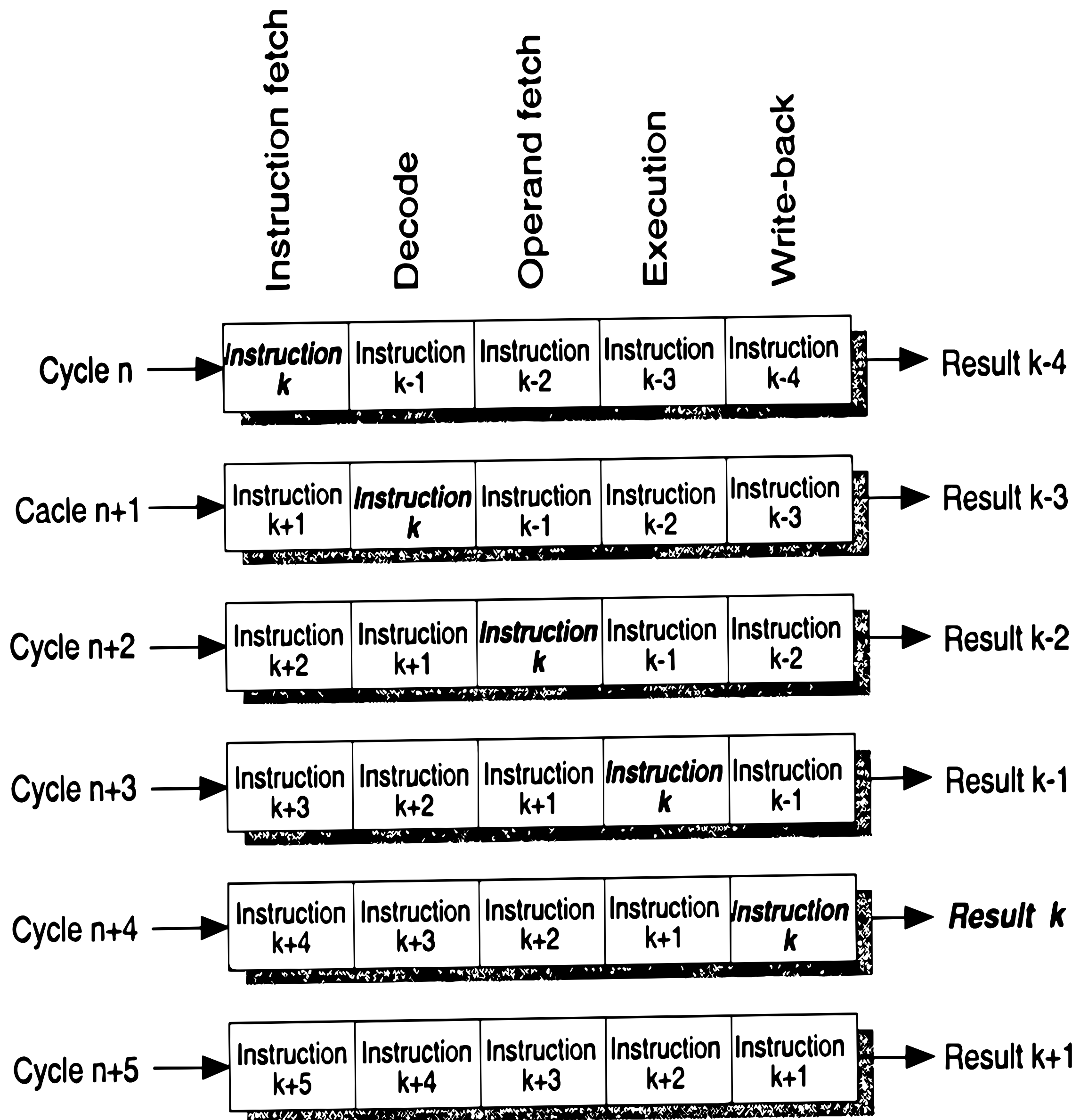
  => hodnoty uložené do *src1*, *src2* zůstanou zachovány.
- Výsledek: **procesor** RISC nepotřebuje tak často komunikovat s **pomalou pamětí**.
- Uplatnění architektur RISC – nepracuje se s jedním či dvěma universálními registry ale je k dispozici tzv. *register file* (*sada registrů*) – takto nejsou vybaveny procesoru Intel – cenové důvody.

- Architektura CISC (např. I80386) – pro ukládání výsledků aritmetických operací je k dispozici pouze *střadač*.

Uplatnění paralelismu na úrovni provádění instrukcí

- Většina instrukcí se provádí v těchto krocích:
  1. **čtení instrukce** z paměti (instruction fetching)
  2. **dekódování instrukce** (decoding phase)
  3. **čtení operandu** (operandů), pokud to instrukce vyžaduje (operand fetching phase)
  4. **provedení instrukce** (execution phase)
  5. **uložení výsledku** (write-back phase)

- Obr. 4 – provádění instrukcí je rozděleno na 5 fází.
- Počítač nezačne provádět instrukci *k*, dokud instrukce *k – 1* neopustí první stupeň.
- Obecně – procesor zahájí čtení instrukce z paměti, jakmile předcházející instrukce přejde do fáze dekódování.



Obr. 4 Uplatnění paralelismu na úrovni provádění instrukcí

- Vysvětlení pojmů (fází realizace instrukce):

  Instruction Fetch – čtení instrukce z paměti (rychlá vyrovnávací paměť - cache (RVP), L1, L2, operační paměti)

  Decode - dekódování instrukce

  Operand Fetch – čtení operandu

Execution – provedení instrukce

Write back – zápis (uložení) výsledku

- Ideální případ – v každém stupni se provádí jedna fáze jedné z 5 instrukcí, pro realizaci jedné fáze je zapotřebí jeden cyklus => na provedení instrukce je zapotřebí 5 synchronizačních pulsů a s každým synchronizačním pulsem se na výstupu objeví jeden výsledek.
- Mezi jednotlivé stupně je vložen registr, který pracuje jako výstupní pro jeden stupeň a jako vstupní pro následující.
- Takové uplatnění paralelismu je typické pro RISC procesory.
- První typy CISC procesorů – neměly takovou strukturu => instrukce *k + 1* se zahájila poté, co skončila instrukce *k*.
- Pokud by byly 2 procesory - jeden RISC a druhý CISC - realizovány na shodné technologii (pracovaly by stejně rychle), pak by se instrukce v procesoru RISC prováděla 5 x rychleji než v původním procesoru CISC.
- Pozdější verze procesorů CISC – zřetězené zpracování.

- Důležité: zatímco některé prvky se v architekturách CISC objevovaly pouze jednou (např. sčítačka), pak v architekturách RISC mohou být zapotřebí v každé fázi provádění instrukce, tzn. vyskytují se násobně.
- **Příklad**:

  Instrukce *ADD eax, [ebx + ecx]* – je zapotřebí provést **2 součty**: určit adresu operandu a realizovat vlastní realizaci instrukce ADD.

  Původní procesory CISC – **vše bylo realizováno jednou sčítačkou**, bylo to možné, protože každý krok byl realizován v časově disjunktních okamžicích.

  Procesor RISC – každá jednotka realizující jednotlivé fáze provádění instrukce potřebuje své vlastní vybavení, což při současném stavu integrace není problém (v tomto případě musí sčítačka být v dekódovací jednotce a v prováděcí jednotce).

  Není důvod, aby tyto principy nebyly uplatněny také v moderních verzích procesorů CISC.

  Příklad:

  Za sebou následují tyto instrukce:

  ADD eax, [ebx + ecx]

  MOV edx, [eax + ecx]

V jistém okamžiku je instrukce *ADD* v prováděcí fázi => potřebuje sčítačku, instrukce *MOV* je ve fázi dekódování => potřebuje sčítačku na určení adresy operandu.

Jiná alternativa – mít pouze jedinou sčítačku => dekódování instrukce *MOV* by se muselo zpozdit, dokud se nedokončí provedení sčítání (instrukce *ADD*).

- Další možná situace: předpokládáme tytéž instrukce jako v předcházejícím případě.

  ADD eax, [ebx + ecx]

  MOV edx, [eax + ecx]

  Konkrétní stav provádění této posloupnosti instrukcí: instrukce *ADD* je v prováděcí jednotce, instrukce *MOV* je v jednotce dekódovací.

  Problém: instrukce *MOV* potřebuje hodnotu uloženou v registru *eax*, ta je však známá až po provedení instrukce *ADD*.

  Řešení: 2 možnosti

  1. zajistit zpoždění výpočtu operandu v dekódovací jednotce (o jeden cyklus),
  2. ve fázi překladu – překladač buď vloží do posloupnosti instrukcí NOP anebo provede restrukturalizaci programu =>

v architekturách RISC tvoří hardware i software navzájem propojený celek, překladače musí respektovat problémy, které mohou nastat při implementaci hardware realizujícího instrukce.

- Realizace instrukce v jedné frontě – tzv. skalární struktura.
- Pětistupňová realizace instrukce je pouze jedna z možností – další možnosti:

  Dekódování a čtení operandů může být spojeno do jedné fáze => struktura je pak čtyřstupňová.

  Rozložení provádění instrukce do více fází => *superpipelined architecture* (superřetězení) – 10 a více fází.

  Jiná alternativa – front instrukcí je více => *superskalární architektura* (architektura s jednou frontou – *skalární architektura*).
- *Superskalární architektura* – problémy s koordinací činností v jednotlivých frontách se umocňují – musejí spolupracovat nejenom komponenty řazené za sebou ale také jednotlivé paralelní fronty – např. problém tzv. párování instrukcí.

- Další problém – *párování instrukcí* do front tak, že tyto instrukce mohou být prováděny paralelně.
- Jako *superskalární struktura* může být označen také multiprocesorový systém na jednom čipu.
- Příklady superskalárních struktur – Pentium má dvě fronty instrukcí.
- Dosahované rychlosti:

  CISC architektura – I80386, 33 Mhz, 15 MIPS

  Uplatnění prvků RISC architektury – hodnota v MIPS přibližně dvakrát vyšší než synchronizace – Pentium 100 Mhz => přibližně 160 MIPS.

Vzájemné vazby mezi fázemi provádění instrukce a jejich blokování

- Důležitý aspekt zřetězeného zpracování – všechny fáze provádění instrukce by měly končit ve stejný okamžik.
- Ve struktuře podle obr. 4 to pak znamená, že všechny instrukce by měly skončit po 5 cyklech.
- Stav, kdy operandem instrukce jsou data z paměti – **je potřeba je přečíst buď z rychlé**

vyrovnávací paměti (RVP) nebo z operační paměti (OP).

- Časové relace – přenos z RVP – 10 ns, jeden cyklus.
- Přenos operandu z OP – 60 ns => přechod z fáze „čtení operandu“ (operand fetch) do fáze „provádění instrukce“ (execution) se musí zpozdit => výrazným způsobem se naruší synchronizace jednotlivých kroků v jednotlivých fázích zpracování instrukce, vzniká problém se zajištěním stejné doby trvání zpracování instrukce v jednotlivých krocích.
- **Řešení**:

  Zpoždění provádění programu - vkládáním NOP – tzv. *delayed load* (zpožděné čtení).

  Jiná technika – *load forwarding* (zavedení operandu dopředu) – operand se z paměti nečte do registru ale přivádí se přímo na vstup ALU, tzn. registry se obejdou (proto *forwarding*).
- Snaha o vyřazení instrukcí, kde operandem je hodnota uložená v paměti => typická architektura RISC – s pamětí manipulují pouze instrukce typu *load/store*, žádné jiné instrukce s pamětí nemanipulují => instrukce *ADD* odkazuje pouze na interní registry procesoru => instrukce *ADD reg, mem* neexistuje.

- Pak se vyskytují např. takové posloupnosti:

  LOAD reg1,mem

  ADD dest,reg1,reg2
- Stav, kdy instrukce *LOAD* je ve fázi provádění, zatímco *ADD* je ve fázi čtení operandů => zákonitě musí vzniknout „*delayed load*“, protože *ADD* se nemůže provést, dokud není v *reg1* uložen operand z paměti.
- **Řešení 1** (Berkeley):
  - Je zaveden pojem „*scoreboarding*“ spočívající v:

    Každému registru procesoru je přidělen jeden bit, který reprezentuje to, zda informace v něm uložená je/není platná.

    Princip: instrukce, která operuje s registrem, nejprve na začátku provádění instrukce nastaví tento bit do „1“ (obsah je neplatný), jakmile se do registru během fáze provádění (execution) vloží data, pak se tento bit vynuluje.

    Instrukce, která potřebuje platný operand v *reg1* (instrukce *ADD*), je pak patřičným způsobem zpožděna, pokud je tento bit v 1.

    Toto všechno zařídí řadič, tzn. hardware a řeší se to až při provádění instrukce.

- **Řešení 2** (Stanford):
  - Je řešeno již při překladu programu tak, že do posloupnosti instrukcí vkládá překladač instrukce typu *NOP* => výsledkem je tento kód:

    LOAD reg1,mem

    NOP

    NOP

    ADD dest,reg1,reg2
  - Takové řešení má problémy, protože překladač nebere v úvahu tyto aspekty (v okamžiku překladu nejsou známy):
    zda je operand v RVP typu L1, L2 nebo hlavní paměti (OP),
    kmitočet, jímž budou tyto operace řízeny,
    kvalitu paměťových čipů (rychlost),
    režim činnosti paměti.
  - Kompilátor musí počítat spíše s horší alternativou, takže počet vložených NOP může být příliš vysoký => snaha o optimalizaci kompilátoru, kompilátor musí respektovat principy uplatněné v procesoru => bude souviset s typem procesoru.

Zpožděný skok a zpožděné větvení

- Instrukce skoku a podmíněného skoku tvoří asi 30 % všech instrukcí (?) – tyto skupiny instrukcí mají na běh programu nežádoucí efekt.
- V procesoru, v němž je provádění instrukcí rozděleno na jednotlivé fáze, pak může nastat tento stav:
  - Čítač adresy ukazuje průběžně na adresy instrukcí, která se načítají, ve frontě instrukcí (před instrukcí právě načítanou z paměti) je ale **instrukce skoku** nebo **podmíněného skoku (budou se provádět dříve než instrukce právě načítaná z paměti)**, **které potenciálně mohou změnit posloupnost provádění instrukcí**.
  - To, zda se bude měnit adresa (tzn. bude se přecházet na jinou instrukci) podle výsledku podmínky v instrukci podmíněného skoku, bude jasné, až se tato instrukce dostane do fáze provádění (execution) – proto **zpožděné větvení**.
  - Tzn. když se bude provádět instrukce skoku nebo podmíněného skoku, budou v jednotlivých fázích rozpracovány různé instrukce, které se budou potenciálně rušit

(podle výsledku instrukce podmíněného skoku).

- Tento stav je označován jako *delayed jump* (zpožděný skok) nebo *delayed branch* (zpožděné větvení).
- Pokud se má realizovat skok, pak se musí zrušit všechny instrukce, které jsou již rozpracovány v jednotlivých fázích.
- Nejjednodušší řešení – zařazení instrukcí NOP za instrukce skoku – musí se zařídit při překladu.
- Účinek vkládání NOP: neprovádějí se instrukce, které možná (podle výsledku instrukce podmíněného skoku – ten se ještě neví) bude zbytečné provádět.
- Výsledkem instrukce podmíněného skoku je vygenerování ukazatele na další instrukci (2 možné výsledky) v kroku zápis výsledku (write-back).
- Jiná možnost: jakmile se v dekodéru rozpozná některá z těchto instrukcí, tak se začnou načítat dva sledy instrukcí respektující oba výsledky podmíněného skoku.
- Další možnost řešení: dostatečně dlouhá fronta instrukcí, takže se zvyšuje

pravděpodobnost, že oba výsledky instrukce podmíněného skoku budou odkazovat na instrukci, která je ve frontě a nemusí se tudíž číst z paměti (RVP nebo OP).

- Progresivní způsob: **předvídání výsledku instrukce podmíněného skoku** (bude vysvětleno později).