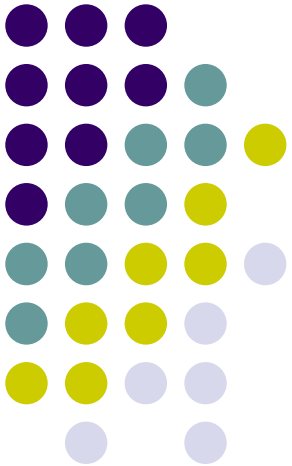
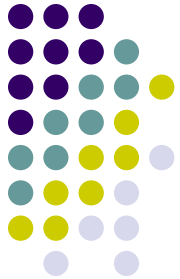
RISC x CISC pipelining

Hardware

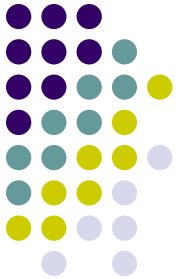


RISC x CISC



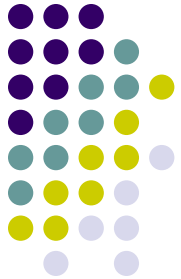
- Vývoj mikroprocesorů po roce 1980 postupně nezadržitelně vedl k jejich **nadměrné složitosti**
- Okolo roku 1985 let vznikl nový směr vývoje mikroprocesorů, jehož snahou bylo zjednodušit strukturu procesorů a přitom zvýšit jejich výkon
- Vzniká nová kategorie **RISC** (**R**educed **I**ntruction **S**et **C**omputer) - tj. počítače (nebo spíš procesory) s redukovaným souborem instrukcí
- Tento směr se stal populárním, ale měl i odpůrce, kteří dál prosazovali tradiční architekturu, kterou obhajovali pod označením **CISC** (**C**omplex **I**ntruction **S**et **C**omputer) - tj. počítač se složitým souborem instrukcí

CISC



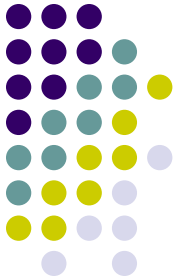
- Původní směr vývoje procesorů a počítačů
- Návrháři se neustále od počátku vzniku procesorů (1970) snažili zlepšovat jejich vlastnosti a zvyšovat výpočetní výkon
- Zlepšení často spočívalo v zavedení **nových složitějších instrukcí**
- Idea: Jestliže programátor bude mít k dispozici v instrukční sadě procesoru **složité výkonné instrukce**, pak programy zapsané takovými dokonalými instrukcemi budou jednoduché, krátké a obsadí malý objem paměti – Program bude krátký a napsaný rychle pomocí výkonných instrukcí
- Tyto úvahy se ale v praxi **nepotvrdily**
- Instrukční sada procesorů se dále rozšiřovala také z toho důvodu, že každý nový procesor přidává nějaké nové instrukce a kvůli **zpětné kompatibilitě** zachovává i ty staré
- Instrukční sada procesorů typu **CISC** obvykle obsahuje více než 200 instrukcí
- Instrukce **CISC** procesoru jsou v něm obvykle prováděny pomocí **mikroprogramu** (často dost složitého)
- Doba provádění jednotlivých instrukcí **CISC** procesoru se vzájemně značně **liší** podle jejich složitosti (každá instrukce je provedena během jiného počtu taktů)

RISC



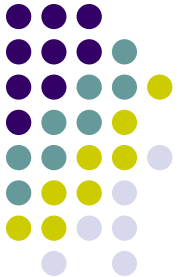
- Praktické výzkumy ukázaly, že množina skutečně používaných instrukcí se omezuje na malou část instrukční sady
- Zjistilo se například, že 85% programu je obvykle tvořeno pouze kombinací 7 nejčastějši používaných instrukcí
- četnost výskytu ostatních instrukcí je minimální (procento a méně)
- To vzbuzuje pochybnosti, zda takové instrukce má vůbec význam zařazovat do instrukční sady – instrukce, které se málo používají vlastně jen komplikují strukturu procesoru a prodražují ho
- IBM v roce 1979 realizovala minipočítač **IBM801** s redukovaným souborem instrukcí
- Počítač byl pouhým experimentem a byl zdrojem cenných zkušeností
- Na univerzitě Berkeley byl v roce 1982 navržen procesor **RISC I**
- O rok později na univerzitě Stanford mikroprocesor **Mips** (MIPS = million instructions per second)
- Filozofie RISC se ukázala jako životaschopná

RISC



- Hlavní idea RISC filozofie:
- Mikroprocesor umí vykonávat pouze **velmi malý počet nezbytně nutných instrukcí**
- Takový mikroprocesor je **jednoduché navrhnout**
- Mikroprocesor bude **levný**, protože jeho návrh je snadný (v ceně mikroprocesoru se musí zaplatit náklady na jeho vývoj a návrh)
- Mikroprocesor bude obsahovat **malý počet tranzistorů**
- **Výroba** mikroprocesoru bude tedy snadná
- Mikroprocesor, který obsahuje malý počet tranzistorů spotřebuje **menší množství energie**
- Mikroprocesor, který spotřebuje menší množství energie, se **méně zahřívá** a půjde **snadno chladit** – bude například stačit pasivní chladič
- Mikroprocesor bude fungovat lépe i na **vysokých taktovacích frekvencích** – malý počet tranzistorů a logických obvodů má menší celkové zpoždění a procesor se také méně zahřívá, takže může pracovat rychleji

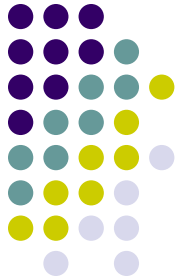
RISC



- **Příklad „CISC robota“**
- Robot se umí pohybovat v prostoru
- V jeho instrukční sadě je mnoho složitých povelů - KROKVPŘED, KROKVZAD, KROKDOPRAVA, KROKDOLEVA, OTOČITDOPRAVA, OTOČITDOLEVA, OTOČITVZAD
- Napsat program, který bude pohybovat robotem bude docela snadné, program bude krátký, protože k dispozici je řada instrukcí
- Robot bude složitý, drahý, jeho řídící jednotka obsahuje mnoho tranzistorů, které spotřebují mnoho energie

- **Příklad „RISC robota“**
- Robot se umí pohybovat v prostoru
- V jeho instrukční sadě jsou pouze dvě instrukce – KROKVPŘED, OTOČITDOPRAVA
- Jak otočit robota čelem vzad? – OTOČITDOPRAVA, OTOČITDOPRAVA
- Jak otočit robota doleva? – OTOČITDOPRAVA, OTOČITDOPRAVA, OTOČITDOPRAVA
- Jak udělat krok doprava? – OTOČITDOPRAVA, KROKVPŘED
- Napsat program pro takového robota vyžaduje více kroků, program bude delší
- Robot bude levnější a jednodušší, jeho řídící jednotka bude mít nižší spotřebu energie a bude se méně zahřívat. S robotem lze nakonec pohybovat úplně stejně jako se složitým CISC robotem a při tom ho lze pořídit za nižší cenu a i jeho provoz je levnější

Vlastnosti RISC procesoru



- Celkový **počet instrukcí je malý**
- Provedení všech instrukcí **trvá stejně dlouho**
- Počet instrukcí vykonaných za sekundu je stále stejný
- Instrukce také mají **pevnou délku** v bajtech **strojového kódu** – kompaktní strojový kód (všechny instrukce jsou např. dvoubajtové) – 1 KB strojového kódu obsahuje vždy stejný počet zakódovaných instrukcí
- Instrukce se provádějí v obvodově navrženém řadiči (nepoužívá se mikroprogram)
- **Adresace paměti je jednoduchá**, počet způsobů adresace paměti je malý (obvykle pouze přímá a jednoduchá nepřímá adresace)
- Pro přístup do paměti jsou k dispozici pouze dvě instrukce - zápis do paměti a čtení z paměti (**LOAD / STORE**)
- Operandy běžných instrukcí mohou být **pouze registry**, instrukce neumějí pracovat přímo s daty v pamětí
- Procesor má **vysoký počet registrů** nebo více skupin registrů (např. 32-64 registrů)
- Procesor **nemá střadač** – všechny registry jsou **univerzální**

Srovnání

2,5 MIPS je pouze **průměrná hodnota**. Počet provedených instrukcí je závislý na konkrétním programu. Složitější instrukce potřebují více taktů, jednoduché méně.

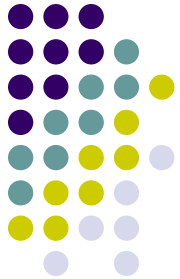
- Koncem osmdes… …h procesorů čipy M… (CISC) a ARM (RISC)
- Zajímavé je jejich srovnání

- **Motorola - MC 68020** - 192000 tranzistorů, doba návrhu 20 měsíců, výkon 2,5 MIPS
- **Acorn - ARM** - 25000 tranzistorů, doba návrhu 5 měsíců, výkon 3 MIPS

3 MIPS je absolutní pevná hodnota, protože všechny instrukce jsou stejně složité a trvají stejně dlouho.

- ARM je jednodušší, podstatně levnější a jeví se na první pohled i výkonnější
- Ve skutečnost… …e umožňuje složitější adresaci a má bohatší instrukční sadu (některé instrukce bychom na ARMu museli rozložit na více jednoduchých)
- Z mikroprocesorů ARM se postupným vývojem staly nejpoužívanější mikroprocesory v mobilních telefonech a tabletech

Nejznámější historické procesory RISC



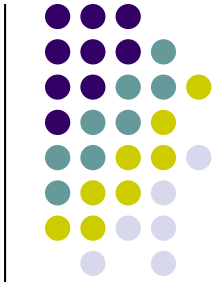
- **Alpha** - mikroprocesory Alpha vyráběla firma DEC (Digital). Jedná se o první 64-bitové procesory, které byly na trhu

- **SPARC** - procesory RISC, které se nejčastěji používali v pracovních stanicích firmy SUN Microsystem

- **MIPS** - Procesory MIPS se používali v pracovních stanicích Silicon Graphics, na kterých se zpracovává většina animovaných sekvencí pro filmový průmysl

- **PowerPC –** Výsledek spolupráce IBM+Apple+Motorola. Záměrem bylo vytvořit nástupce původních PC, další etapu vývoje osobních počítačů – nenavazovat na 80486, ale vydat se novou RISC cestou. Používali se ale nakonec pouze v počítačích firmy Apple.

RISC a přístup do paměti



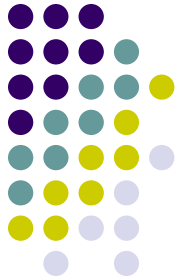
- **RISC** procesory jsou typické tím, že mají pouze dvě instrukce pro práci s pamětí - **čtení** a **zápis** (LOAD / STORE)
- **CISC** procesory umožňují použití adresy jako operandu běžných instrukcí
- Příklad programu pro RISC

```
LOAD R2,500
LOAD R1,502
ADD R1,R2
STORE 600,R1
```

- Příklad programu pro CISC

```
ADD    A,500
INC    501
MOV    502,501
```

Výhody a nevýhody RISC



- **Malý počet instrukcí** vede ke vzniku **delších programů** - programy musí být zapsány pomocí většího počtu instrukcí, protože některé operace je třeba „rozepsat“

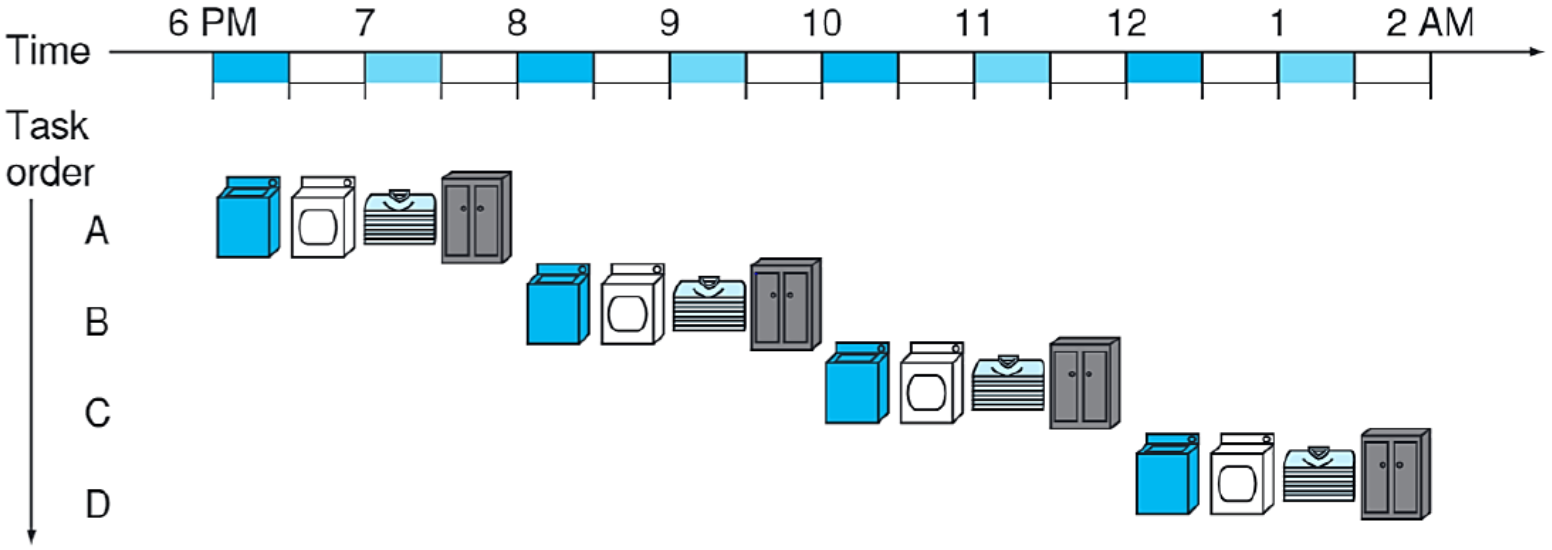
- **Vysoký počet registrů** minimalizuje počet nutných přístupů do paměti, usnadňuje práci s daty a mezivýsledky

- RISC instrukce jsou obvykle **stejně časové náročné** a mají **stejně velký strojový kód**, takže jsou snadno umísťovány do fronty instrukcí přímo v procesoru a snadno lze realizovat **pipelining** (viz o pár stránek dále)

- Díky pipelinigu je pak mikroprocesor schopen dokončit v každém taktu jednu instrukci – při dokonalém pipelingu se potom počet instrukcí vykonaných za sekundu rovná taktovací frekvenci (např. mikroprocesor s f=100 MHz má výkon 100 MIPS)

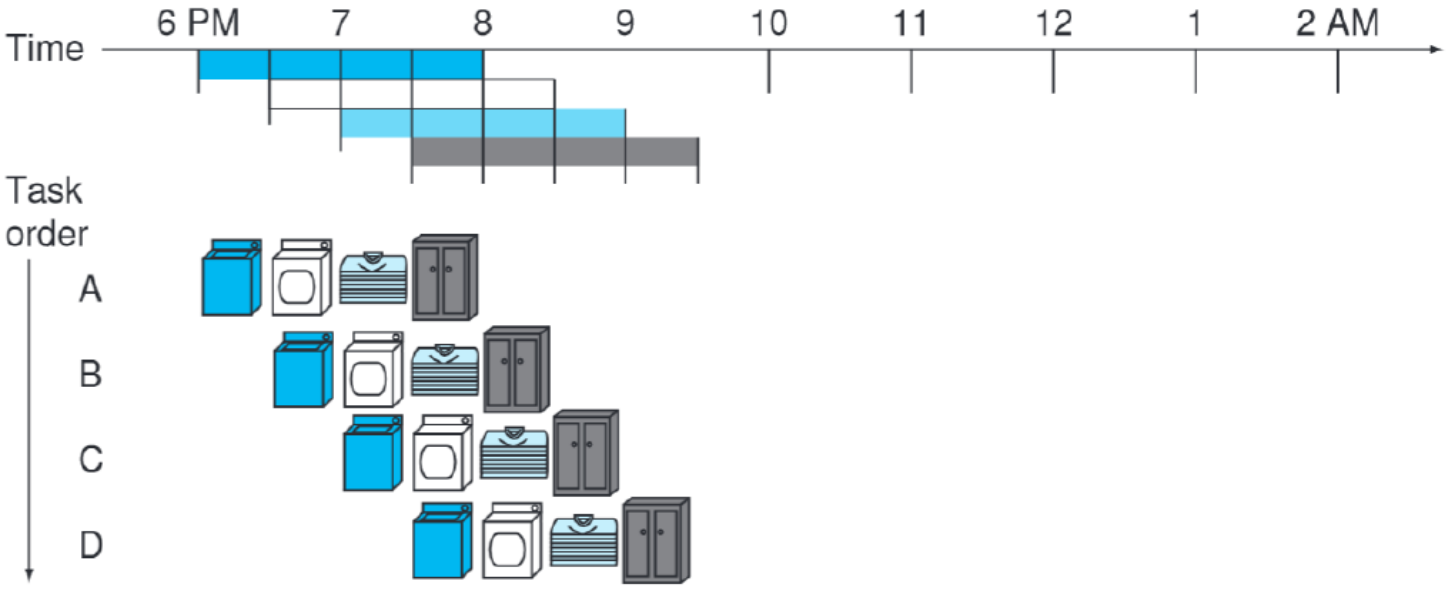
Pipelining – proudové zpracování

- Příklad:
- Je třeba vyprat, usušit, vyžehlit a uložit zpět do skříní 4 dávky prádla
- Praní trvá 30 minut
- Sušení trvá 30 minut
- Žehlení trvá 30 minut
- Úklid do skříní trvá 30 minut
- Zpracování jedné dávky prádla tedy trvá 4 x 30 minut, tj. 2 hodiny
- Zpracování všech dávek pak bude trvat **8 hodin**

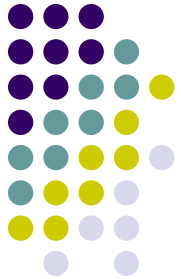


Pipelining – proudové zpracování

- Příklad:
- Jednotlivé úkony je možné provádět **proudově** (částečně paralelně) – zatímco se bude první dávka prádla sušit, může se další dávka prát. Až se bude první dávka žehlit, může se druhá dávka sušit a třetí dávka prát atd…
- Zpracování jedné dávky stále trvá 4 x30 minut, tj. 2 hodiny
- Ale jednotlivé fáze se mohou v čase překrývat – pračka, sušička a žehlička mohou pracovat současně a při tom další pracovník může hotové prádlo uklízet
- Po zpracování první dávky, pak bude každá další dávka hotová po půl hodině
- Zpracování všech dávek pak bude trvat **3,5 hodiny**



Pipelining

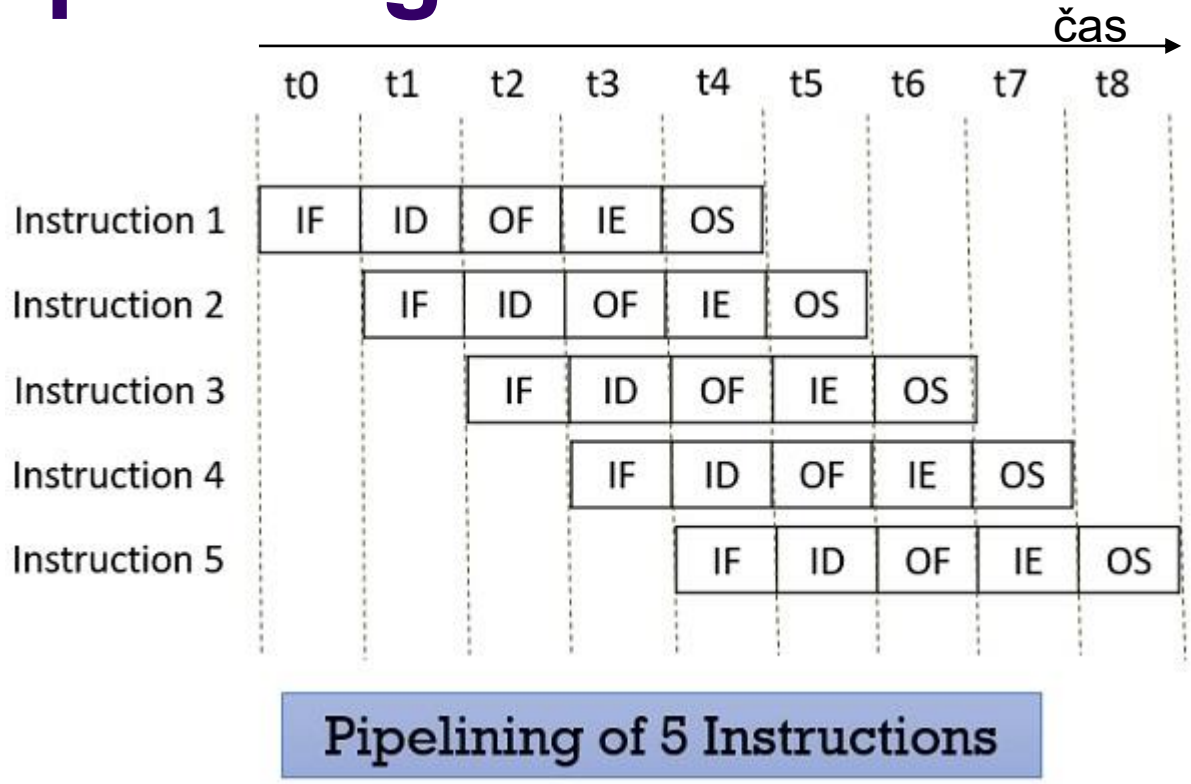


- **Proudové zpracování** - provedení každé instrukce je mikroprocesorem realizováno v několika krocích a lze ho tedy rozložit na několik standardních úkonů (fází)

- Typické **fáze** provádění jedné instrukce jsou například tyto:
  - Výběr strojového kódu instrukce z paměti
  - Dekódování strojového kódu
  - Výběr operandů
  - Provedení operace
  - Zápis výsledku

- **Hlavní idea:** procesor lze rozdělit na **jednotlivé sekce** vykonávající **jednotlivé fáze**
- Každá sekce procesoru umí vykonávat jinou fázi
- Procesor pracuje podobně jako výrobní linka v továrně (nebo jako příklad s prádlem)
- Procesor pak dokáže v jeden okamžik **pracovat paralelně** na pěti (nebo více, pokud je rozdělen na více sekcí) rozpracovaných instrukcích (každá je v jiné fázi)
- **Hlavní přínos: v každém taktu** je zahájeno zpracování nové instrukce, v každém taktu je **dokončena jedna instrukce**

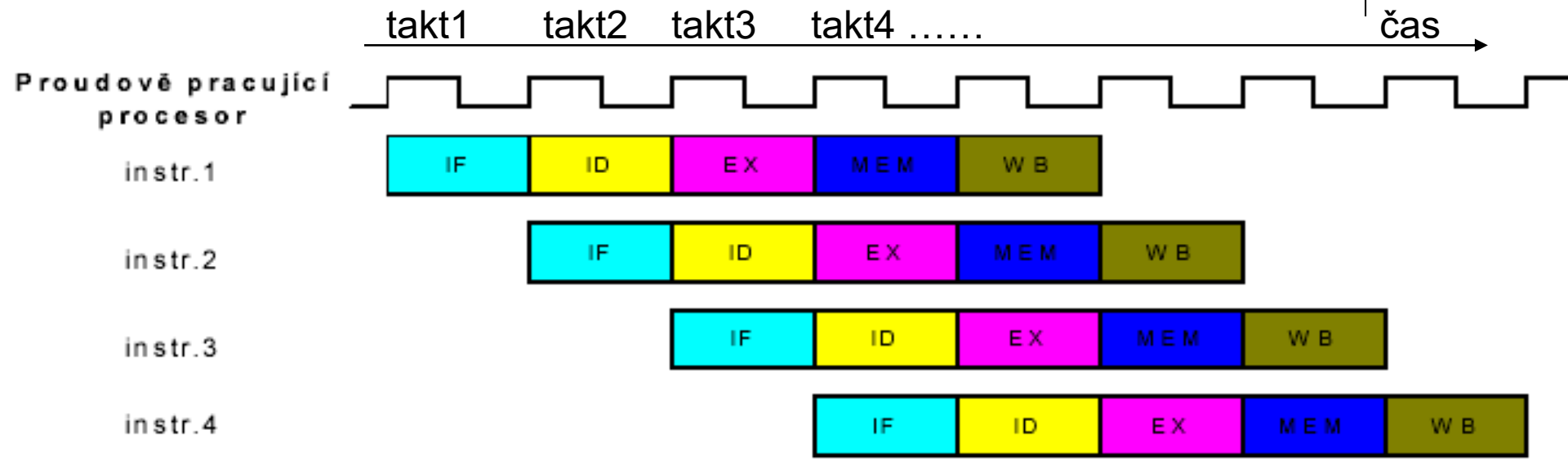
Pipelining



Pipelining of 5 Instructions

IF – Instruction feed – načtení strojového kódu instrukce z paměti
ID – Instruction decode – dekódování strojového kódu
OF – Operands feed - načtení operandů, se kterými instrukce provádí výpočet
IE – Instruction Execution – vykonání výpočtu
OS – Operands Store – uložení výsledků výpočtu

Pipelining



IF – Instruction Feed – Načtení str. kódu instrukce z paměti

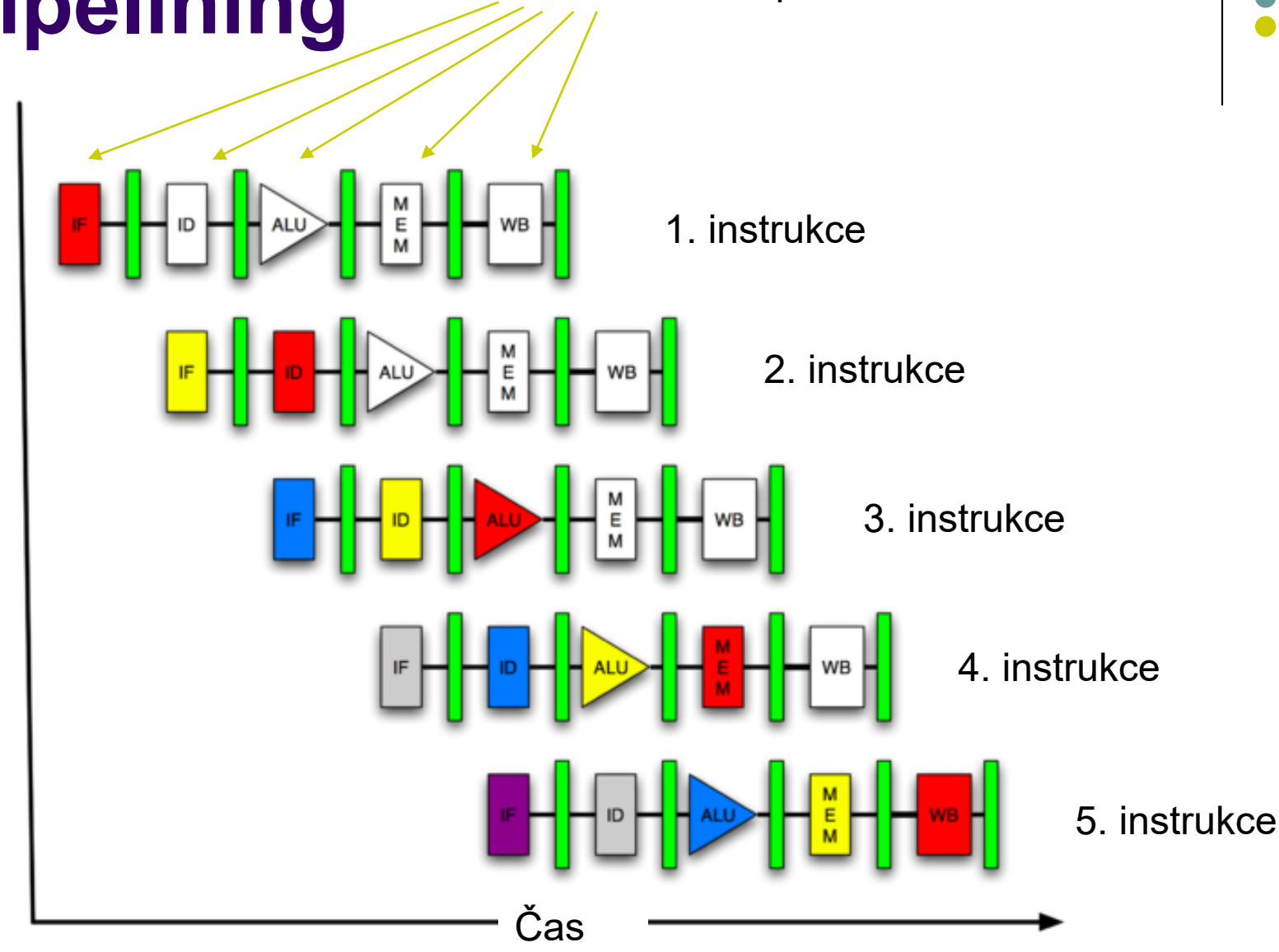
ID – Instruction Decode – Dekódování instrukce

EX – Execution – Provedení instrukce

MEM – Memory – Takt rezervovaný pro přístup do paměti

WB – Write back – Zápis výsledku

Pipelining


Jednotlivé „sekce“ procesoru
IF
ID
ALU
MEM
WB
1. instrukce
2. instrukce
3. instrukce
4. instrukce
5. instrukce
Čas

Pipelining

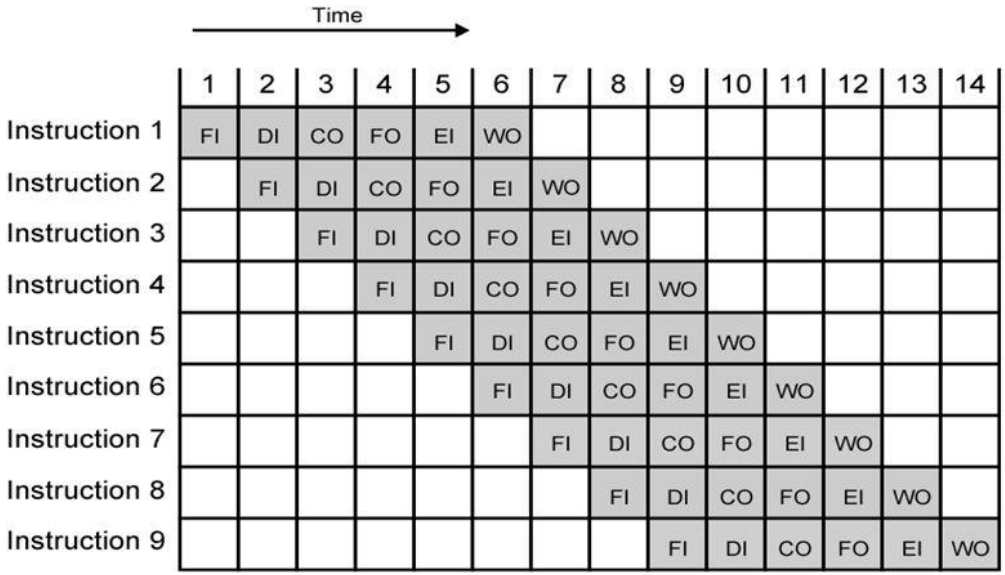
Time Slot ->	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	ID	OF	EX	SR						
Instruction 2		IF	ID	OF	EX	SR					
Instruction 3			IF	ID	OF	EX	SR				
Instruction 4				IF	ID	OF	EX	SR			
Instruction 5					IF	ID	OF	EX	SR		
Instruction 6						IF	ID	OF	EX	SR	
Instruction 7							IF	ID	OF	EX	SR

Instruction Pipeline

IF – Instruction feed – načtení strojového kódu instrukce z paměti
ID – Instruction decode – dekódování strojového kódu
OF – Operands feed - načtení operandů, se kterými instrukce provádí výpočet
EX – Execution – vykonání výpočtu
SR – Store result – uložení výsledku výpočtu

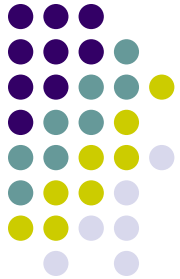
Pipelining

Timing Diagram for Instruction Pipeline Operation



FI – Fetch instruction – načtení strojového kódu instrukce z paměti
DI – Decode instruction – dekódování strojového kódu
CO – Calculate operands – výpočet potřebných operandů (např fyzické adresy)
FO – Fetch operands - načtení operandů
EI – Execute instruction – provedené výpočtu
WO – Write output – zápis výsledků

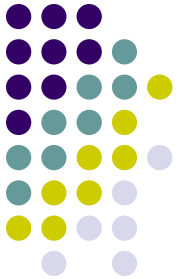
Pipelining



- V případě proudového zpracování instrukcí v pěti sekcích, trvá zpracování každé instrukce 5 taktů
- V každém taktu je ovšem dokončena jedna instrukce a z paměti je čtena jedna nová instrukce
- Takže vlastně během provozu vychází, že je **jedna instrukce provedena během jednoho taktu**
- Ve skutečnosti se pracuje na pěti instrukcích naráz v pěti **paralelně** pracujících sekcích procesoru

- Různé procesory používají různý počet kroků, na které se instrukce dělí
- Například *Pentium 1* rozdělí provedení instrukce na 6 fází, procesor je rozdělen do 6 sekcí – šestistupňový pipelining
- *Pentium 4* používá 20-stupňový pipelining – provedení každé instrukce je rozděleno do dvaceti fází

Pipelining - příklad

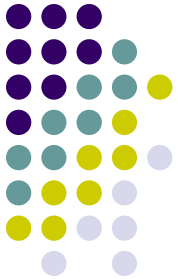


- **RISC** Mikroprocesor s **pětistupňovým** pipeliningem má taktovací frekvenci **100 MHz**
- Jak dlouho trvá vykonání jedné instrukce?
- Kolik instrukcí vykoná procesor za sekundu?
- V kolikátém taktu po spuštení dokončí procesor desátou instrukci?

- Provedení jedné instrukce je rozděleno do 5 fází. Každá fáze je provedena během jednoho taktu. Vykonání instrukce tedy trvá 5 taktů. $t_{instrukce} = 5\ takt = 5T = 5\frac{1}{f} = 5\frac{1}{100\,000\,000} = \mathbf{50\ ns}$
- Díky pipeliningu je procesor schopen v každém taktu dokončit jednu rozpracovanou instrukci. Počet instrukcí vykonaných za sekundu (IPS – instructions per second) tedy bude
- $IPS = f = \mathbf{100\,000\,000\ Instrukcí\ za\ sekundu}$

- Po spuštění procesoru trvá 5 taktů, než bude vykonána první instrukce. Provádění druhé instrukce bude dokončeno hned poté v 6. taktu. Třetí instrukce bude hotová v 7. taktu atd… Desátá instrukce bude tedy dokončena v **14. taktu**

Pipelining - příklad



- **RISC** Mikroprocesor s **10-stupňovým** pipeliningem má taktovací frekvenci **100 MHz**
- Jak dlouho trvá vykonání jedné instrukce?
- Kolik instrukcí vykoná procesor za sekundu?
- V kolikátém taktu po spuštení dokončí procesor desátou instrukci?

- Provedení jedné instrukce je rozděleno do 10 fází. Každá fáze je provedena během jednoho taktu. Vykonání instrukce tedy trvá 10 taktů. $t_{instrukce} = 10\ takt = 10T = 10\frac{1}{f} = 10\frac{1}{100\ 000\ 000} = \mathbf{100\ ns}$
- Díky pipeliningu je procesor schopen v každém taktu dokončit jednu rozpracovanou instrukci. Počet instrukcí vykonaných za sekundu (IPS – instructions per second) tedy bude
- $IPS = f = \mathbf{100\ 000\ 000\ Instrukcí\ za\ sekundu}$

- Po spuštění procesoru trvá 10 taktů, než bude vykonána první instrukce. Provádění druhé instrukce bude dokončeno hned poté v 11. taktu. Třetí instrukce bude hotová v 12. taktu atd… Desátá instrukce bude tedy dokončena v **19. taktu**

Pipelining - příklad

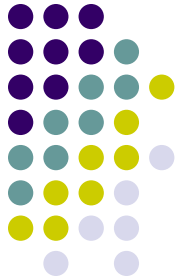
- **CISC** Mikroprocesor bez pipeliningu má taktovací frekvenci **100 MHz**
- Vykonání instrukce trvá v průměru **10 taktů**
- Jak dlouho trvá v průměru vykonání jedné instrukce?
- Kolik instrukcí vykoná průměrně procesor za sekundu?
- V kolikátém taktu po spuštění dokončí procesor desátou instrukci?

- Instrukce jsou časově různě náročné. Provedení jedné instrukce vyžaduje v průměru 10 taktů. Vykonání instrukce tedy průměrně trvá $t_{instrukce} = 10\ takt = 10T = 10\frac{1}{f} = 10\frac{1}{100\ 000\ 000} = \mathbf{100\ ns}$
- Procesor nemá pipelining. Instrukce se provádějí postupně. Provedení instrukce vyžaduje v průměru 10 taktů. Oscilátor během jedné sekundy generuje 100 000 000 taktů.
- $IPS = \frac{f}{10} = \frac{100\ 000\ 000}{10} = \mathbf{10\ 000\ 000\ Instrukcí\ za\ sekundu}$

- Po spuštění procesoru trvá 10 taktů, než bude vykonána první instrukce. Provádění druhé instrukce bude dokončeno za dalších 10 taktů, tedy ve 20. taktu. Třetí instrukce bude hotová v 30. taktu atd… Desátá instrukce bude tedy dokončena v **100. taktu**

- **Situace se ale může každou sekundu měnit, podle toho jak složité instrukce mikroprocesor právě vykonává – provedení každé instrukce na CISC procesoru trvá jinak dlouho**

Pipelining - příklad

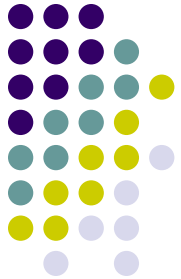


- **RISC** Mikroprocesor s **5-stupňovým** pipeliningem má taktovací frekvenci **10 MHz**
- Za jak dlouho po spuštění programu, který obsahuje 20 instrukcí, bude program dokončen?

- Provedení instrukce je rozděleno do pěti fází, vykonání instrukce vyžaduje pět taktů.
- První instrukci mikroprocesor dokončí v pátém taktu.
- Následně druhou instrukci dokončí v 6. taktu, třetí instrukci v 7. taktu, čtvrtou instrukci v 8. taktu…
- 20. instrukci tedy procesor dokončí v 24. taktu
- Provedení programu vyžaduje 24 taktů
- $t_{program} = 24\ takt = 24T = 24\frac{1}{f} = 24\frac{1}{10\,000\,000} = \mathbf{2400\ ns}$

- **Program bude hotový za 2400 nanosekund.**

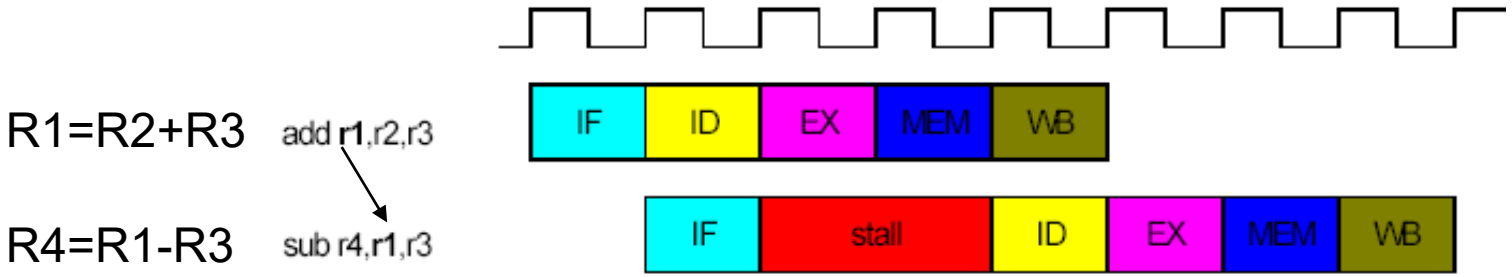
Problémy proudového zpracování



- Ideální využití procesoru mohou narušit
  - Instrukce, které vyžadují pro své provedení více taktů (obvyklé u CISC)
  - Instrukce, které pracují s výsledkem předchozí operace (datové závislosti)
  - Podmíněné skoky (větvení programu)
  - Pomalá paměť, čekání na data z paměti
- Tyto problémy se odborně nazývají **hazardy** (nemá to nic společného s hazardy v logických obvodech)
- **Strukturní hazardy** – daný zdroj není k dispozici, kolize při přístupu ke sdíleným prostředkům (paměť, registr, funkční jednotka)
- **Datové hazardy** – datové závislosti mezi po sobě jdoucími instrukcemi
- **Řídící hazardy** – Skoky měnící stav programového čítače
- Základní a neefektivní řešení: vložení jalového čekacího taktu v pipeliningu

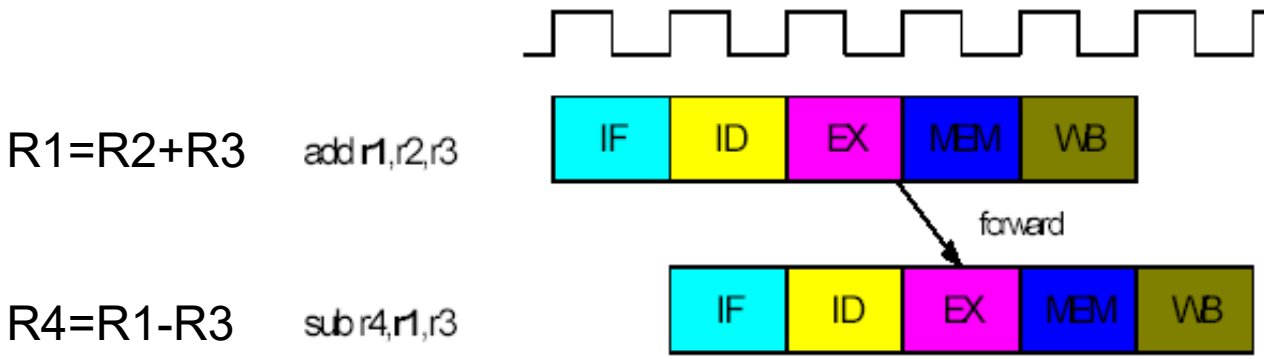
Datové hazardy

- Problém nastává, pokud se instrukce snaží přečíst data, která jsou výsledkem předchozí instrukce, ale ještě nebyla zapsána do cílového registru ve fázi WriteBack



Při dekódování strojového kódu instrukce bylo zjištěno, že pracuje s registrem, jehož hodnotu mění předchozí instrukce, proto byly vloženy čekací cykly (stall)

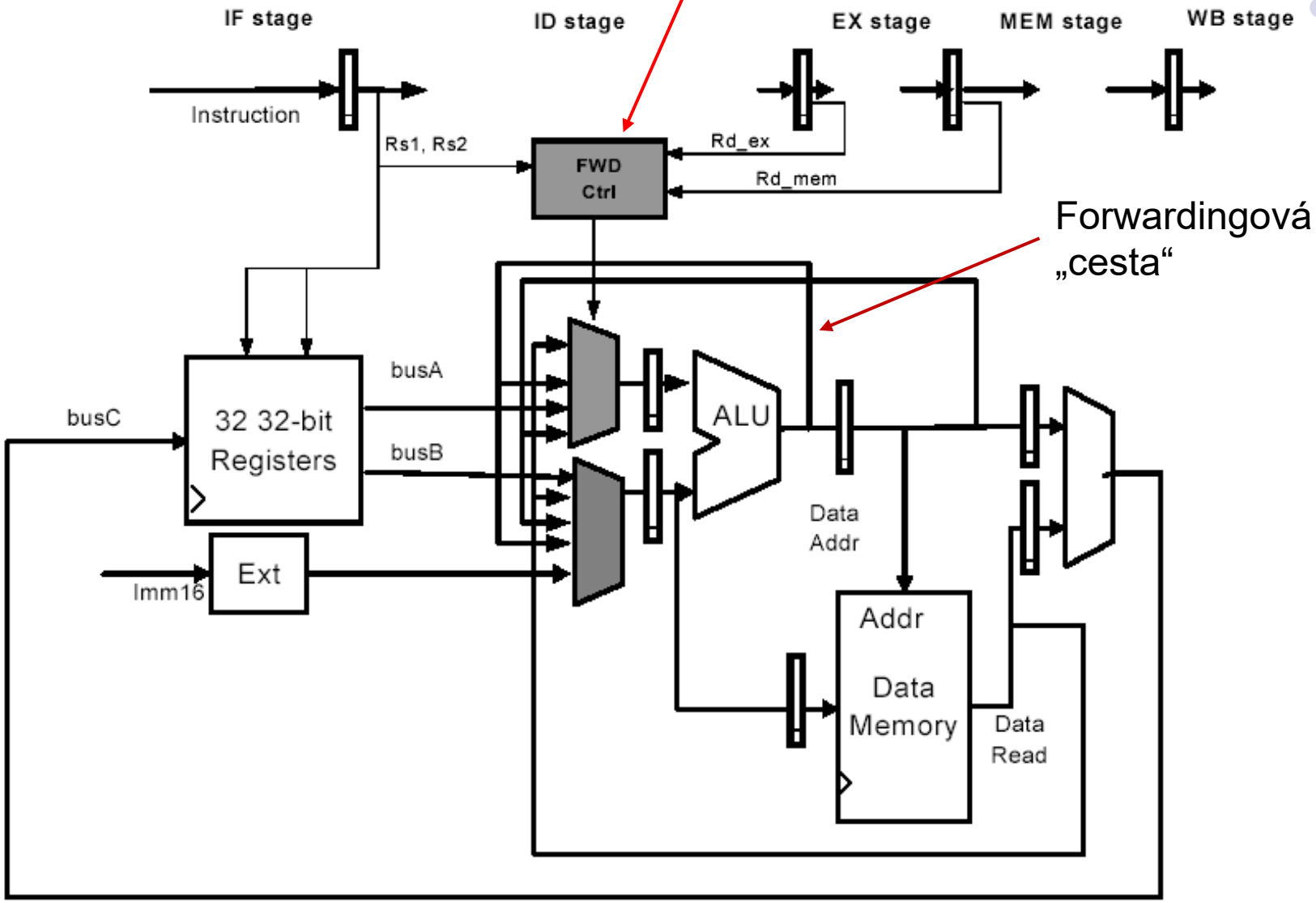
Řešení - forwarding



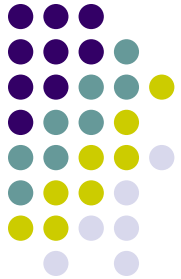
Forwarding umožňuje použití vypočítaného výsledku následující operací ještě před tím, než byl výsledek zapsán do cílového registru

Forwarding

Forwarding control
Test shody cílového registru a zdrojových registrů příští operace



Datové závislosti
Forwarding



```
ADD R1,R5,R6      (R1 ← R5+R6)
INC R1            (R1 ← R1+1)
```

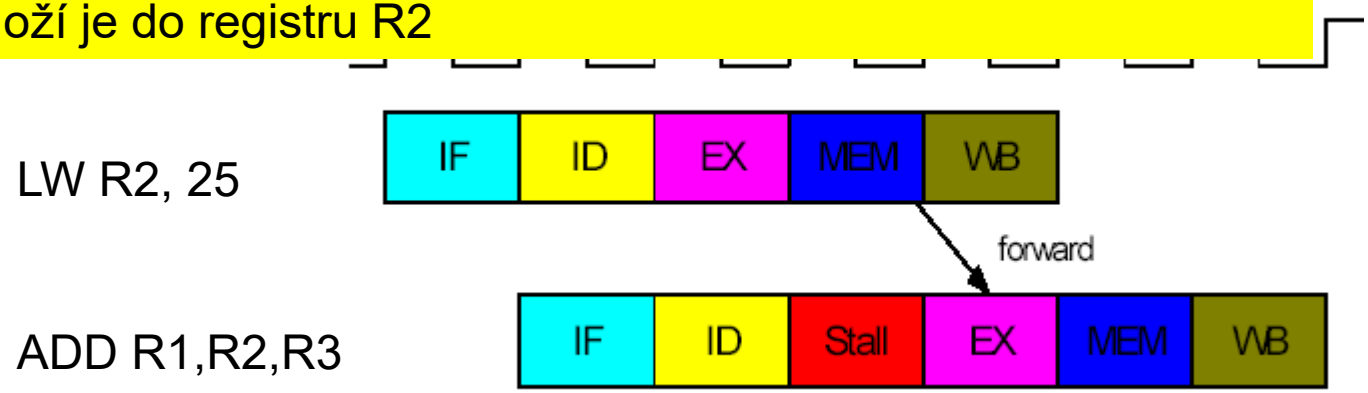
- Inkrementace registru R1 by bez forwardingu mohla proběhnout až po zápisu výsledku operace ADD do registru R1
- Díky forwardingu probíhá inkrementace ihned v návaznosti na ADD
- Výsledek operace ADD **zůstává v ALU** a je ihned inkrementován
- Výsledek operace ADD se ani nezapíše do registru R1
- Do registru R1 bude zapsán až výsledek inkrementace, která na ADD navazuje

Load-use delay

- Při čtení operandu z paměti dochází k problému – operand nelze použít hned v bezprostředně následující instrukci (tady forwarding nefunguje)

Jde o příklad RISC-ových instrukcí

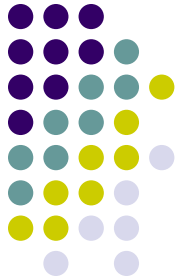
LW R2,25 – Load Word – přečte z paměti z adresy 25 dva bajty a uloží je do registru R2



ADD R1,R2,R3 – R1=R2+R3

ve fázi execute ještě není k dispozici obsah registru R2, protože neproběhla fáze MEM instrukce LW

Load-use delay



- Problém tedy spočívá v tom, že po instrukci LOAD (načtení dat z paměti do registru) nelze ihned v následující instrukci s načtenými daty pracovat, protože vlastně ještě načtená nejsou

- Eliminace load-use delay je možná vložením nezávislé instrukce mezi instrukci čtení z paměti a instrukci, která použije přečtený bajt jako operand

- To je typickým úkolem dobře pracujícího překladače s optimalizací kódu

Eliminace load-use delay

Příklad programu pro

a = b + c;

d = e - f;

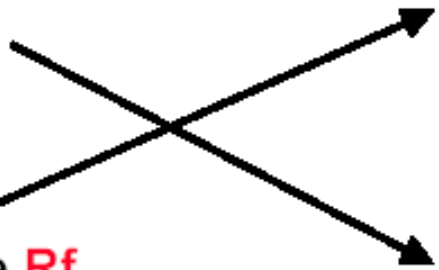
předpokládejme a, b, c, d ,e a f jsou v paměti.

Pomalý program:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Rychlý program:

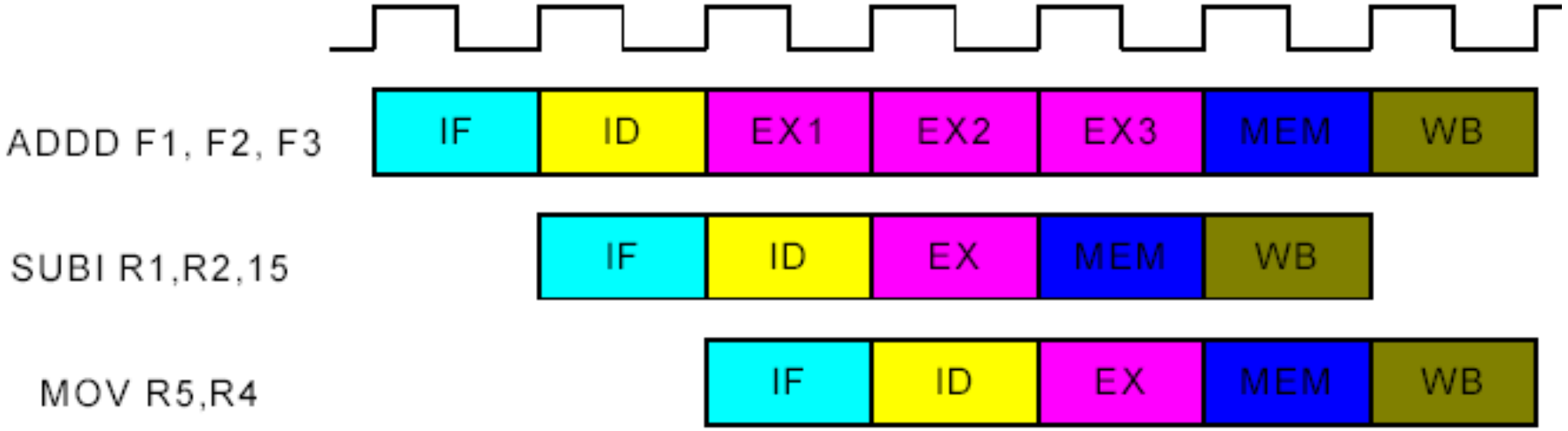
```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```



Strukturní hazardy

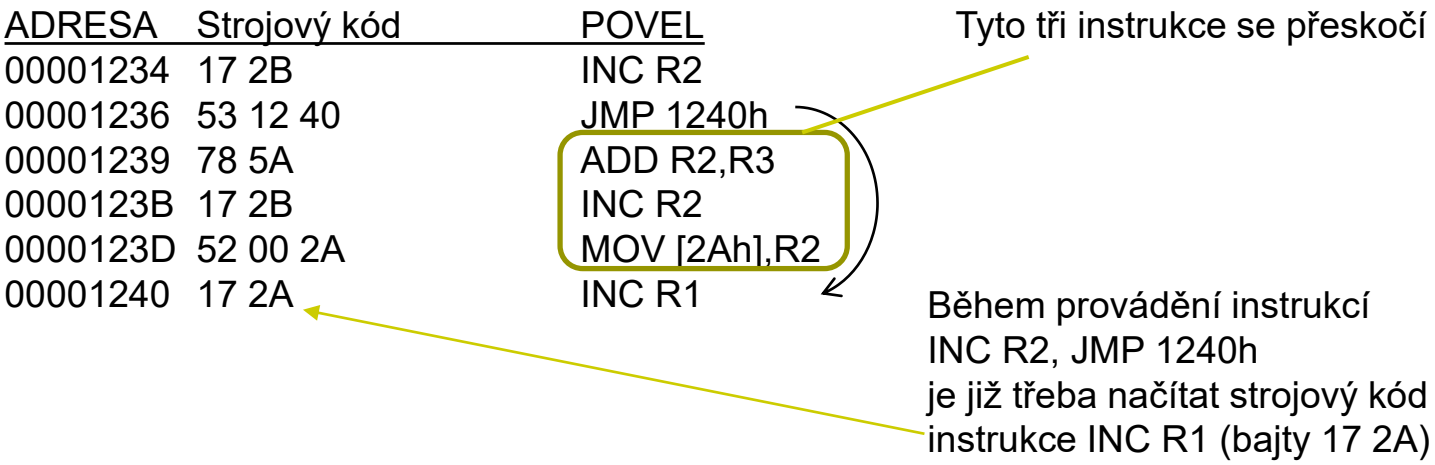
- Tento typ problému vzniká hlavně tehdy, když instrukce mají různou dobu zpracování (vyžadují různý počet taktů – typické pro CISC)
- V jednom taktu může skončit více než jedna instrukce
- V jednom taktu může více instrukcí požadovat provedení fáze ve stejné jednotce

- Dochází ke kolizi

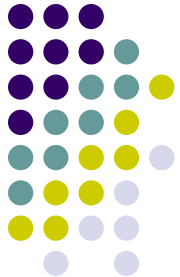


Fronta instrukcí a skok

- Moderní procesory s **proudovým zpracováním** pracují s **frontou instrukcí**, do níž jsou bajty **strojového kódu** vkládány v předstihu
- Při provedení **skoku** je obsah fronty instrukcí k ničemu, protože běh programu se přesouvá na jiné místo a načtené instrukce se přeskočí
- Skok ve strojovém kódu lze dopředu dekódovat a přizpůsobit podle něj načítání strojového kódu dalších instrukcí (tak aby se načetl strojový kód z místa, kam se skočí a ne strojový kód instrukcí, které se přeskočí)

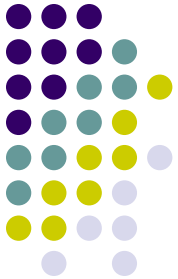


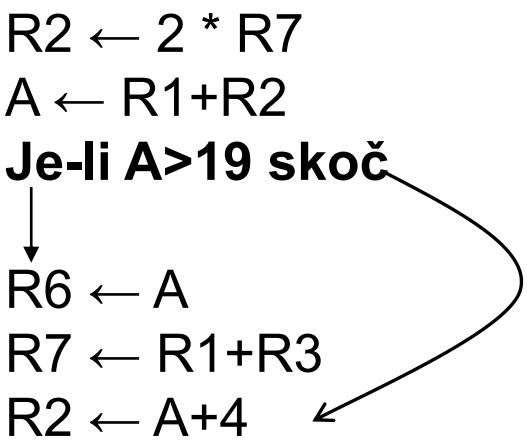
Podmíněný skok



- Bohužel cca 15% všech instrukcí jsou **podmíněné skoky**, které mění pořadí vykonávání instrukcí **nepředvídatelným** způsobem
- **Podmíněný skok** ve strojovém kódu vzniká například kompilací instrukcí if, while, switch-case….
- **Podmíněný skok** = Je-li splněna testovaná podmínka skočí na jiné místo programu; není-li podmínka splněna pokračuje se další instrukcí
- Protože výsledek podmínky **není dopředu znám**, nelze dopředu odhadnout, kterým směrem se bude program po podmíněném skoku ubírat
- Pokud nevíme, jak se bude po podmínce program dále chovat (kudy bude pokračovat), není možné dopředu plnit instrukční frontu a nelze rozpracovávat instrukce – pipelining se zadrhne
- Mikroprocesor tedy neví, jaké další instrukce se po podmíněném skoku budou provádět, dokud nevypočítá výsledek podmínky – ale během tohoto počítání výsledku podmínky je již třeba rozpracovávat nějaké další instrukce
- Mikroprocesor se pokusí odhadnout, kterým směrem bude pokračovat program, který se dle výsledku podmínky větví a rozpracovává instrukce, kterými bude program pravděpodobně pokračovat, ale může se zmýlit – pak se stav všech rozpracovaných instrukcí musí zahodit
- Existují metody, které zvyšují pravděpodobnost, že instrukce vložené do fronty nebudou kvůli podmíněnému skoku zahozeny
- O fungování BTB si povíme později v souvislosti s procesorem PENTIUM

Podmíněný skok





Dokud mikroprocesor nevyřeší, jestli je hodnota registru A vyšší než 19, není jasné, jak bude program dále pokračovat

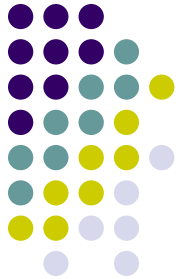
Pokud je A>19, bude následovat operace R2 ← A+4

V opačném případě se nikam neskáče a provádí se R6 ← A

Během řešení, zda je A>19, musí ale mikroprocesor s pipeliningem již jednu z těchto dvou operací rozpracovat!

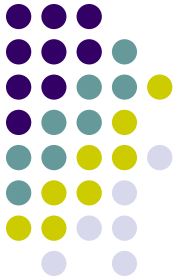
Než bude známo, zda platí A>19, musí být dokonce rozpracováno několik dalších instrukcí (podle toho, kolikastupňový se používá pipelining)

RISC x CISC a pipelining



- Na **CISC** procesorech je pipelining teoreticky možný, ale prakticky je jeho realizace nesmírně složitá
- Instrukce každého **CISC** procesoru trvají různě dlouho
- U procesorů **CISC** mohou být zdrojovými operandy adresy a data tedy musí být nejprve přečteny z paměti (v rámci provádění instrukce)
- Některé instrukce pak mohou požadovat více přístupů do paměti než jiné
- Pro proudové zpracování instrukcí (pipelining) je jasně výhodnější **RISC** architektura
- **Superskalární** implementace **CISC** je velmi komplikovaná

Další vývoj mikroprocesorů

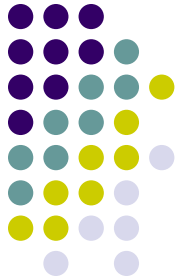


- Do konce osmdesátých let spočívalo **zdokonalování** mikroprocesorů a zvyšování jejich výkonu především v neustálém **zvyšování taktovací frekvence** a v **rozšiřování instrukční sady**

- U pozdějších moderních procesorů dochází ke zvýšení výkonu především hledáním **nových metod** práce procesoru a **nových architektur a koncepcí**

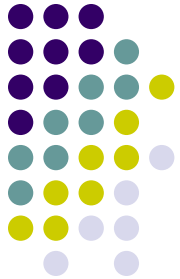
- Výkon procesoru je zvyšován nejen rozšiřováním datové sběrnice, zvyšováním frekvence a zdokonalováním instrukční sady, ale nově také vždy **vynalezením** nějaké **nové koncepce**

Některé rysy moderních procesorů



- **Pipelining** (zřetězené zpracování instrukcí) - instrukce se zpracovávají v několika funkčních blocích. Jakmile se jeden funkční blok uvolní, načte se do něj další instrukce. V jednom okamžiku může být rozpracováno více instrukcí – funguje na RISC procesorech
- **Superskalární architektura** - procesor má více proudově pracujících prováděcích jednotek (více pipeliningů vedle sebe), proto může dokončit v jednom taktu více instrukcí najednou
- **Out-of-Order** (provádění instrukcí mimo pořadí) - instrukce mohou být prováděny i v jiném pořadí, než ve kterém byly umístěny do instrukční fronty a v jakém je zapsal programátor
- **Speculative execution** (spekulativní provádění instrukcí) - procesor si spekulativně provádí některé instrukce předem (a zpětně může některé výsledky odvolat)
- **Register renaming** (přejmenování registrů) - procesor má více fyzických než logických registrů a může stejným jménem označit více fyzických registrů (vhodné pro spekulativní provádění)
- **Řízení spotřeby a výkonu** – různé výkonnostní režimy běhu procesoru
- **SIMD** – jedna instrukce provádí stejnou operaci s více daty naráz
- **Hyperthreading** – paralelní provádění více vláken
- **Multicore - Více jader** – uvnitř procesoru integrováno více nezávislých CPU

Kontrolní otázky



- Uveďte výhody koncepce RISC
- Uveďte nevýhody koncepce RISC
- Porovnejte koncepci RISC a CISC z hlediska možnosti práce s daty uloženými v paměti
- Proč lze na RISC procesoru snadno realizovat pipelining, zatímco na CISC procesoru je to prakticky nemožné ?
- Proč nelze porovnávat výkon dvou různých mikroprocesorů poměrem jejich taktovacích frekvencí ?
- Proč nelze porovnávat výkon dvou různých mikroprocesorů počtem instrukcí vykonaných za sekundu ?
- Vysvětlete, co je to pipeling
- Popište typické fáze proudového zpracování instrukcí
- Proč je podmíněný skok hlavním nepřítelem plynulého pipeliningu?
- Popište, jak funguje forwarding a jaký typ hazardu vyřeší
- Co je to load-use-delay a jak se dá vyřešit?
- Program pro RISC procesor ve strojovém kódu bude obsahovat více nebo méně instrukcí než stejný program napsaný pro CISC procesor ?
- Mikroprocesory používané v moderních chytrých mobilních telefonech jsou typu RISC nebo CISC ?
- Moderní vícejádrové mikroprocesory používané v současných počítačích PC jsou typu RISC nebo CISC ?
- Mikroprocesor 80486 je RISC nebo CISC ?