Architektura Souborů Instrukcí (2)

Ing. Miloš Bečvář

ISA – Shrnutí první přednášky

Střadačově-orientovaná ISA je nejstarší, ale stále užívaná v některých vestavných aplikacích

Zásobníkově-orientovaná ISA umožňuje psaní jednoduchých překladačů. Využívána je v některých řídicích aplikacích a v emulačních systémech jako ISA virtuálních procesorů.

ISA s univerzálními registry (GPR ISA) je dnes nejběžnější.

Registr-Registr (3,0) GPR ISA je typická pro RISC

Registr-Paměť (2,1) GPR ISA je typická pro CISC

Typický RISC má Harvardskou architekturu, pevnou délku kódu instrukce a podporuje pouze zarovnaná data.

Osnova dnešní přednášky

- **Implementace řídicích instrukcí**
- **HW podpora volání podprogramů**
- **Operace vs Instrukce, RISC vs CISC**
- **Současné trendy v architekturách souborů instrukcí**

Implementace řídicích instrukcí

- **Skoky podmíněné a nepodmíněné**
- **Volání a návraty z podprogramů**

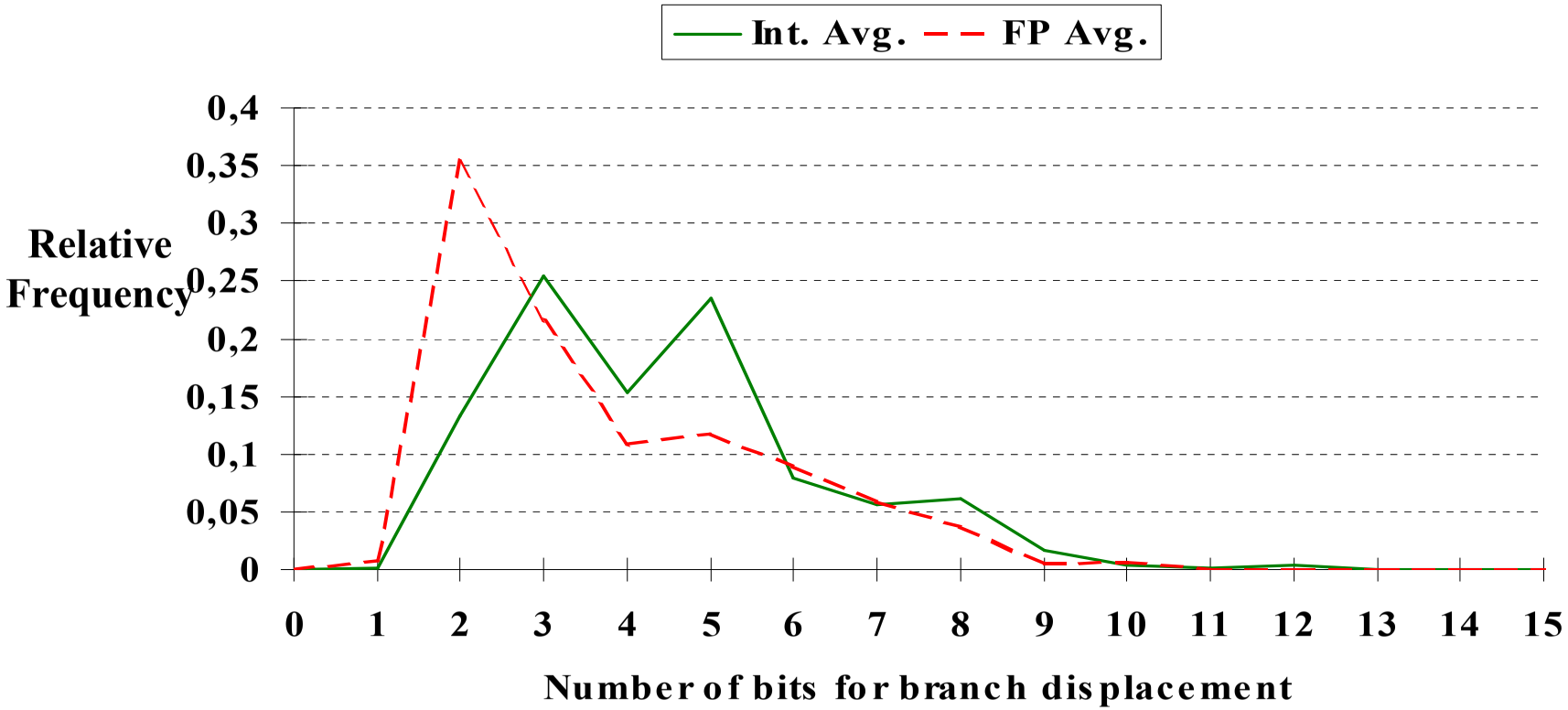
Způsob adresace cíle skoku

– PC-relativní (krátké podmíněné skoky)

– Absolutní (dlouhé skoky a volání podprogramů)

– Registrová nepřímá (tzv. “computed gotos”)
  – podpora „case“ příkazů a dynamického volání
  – (objektové jazyky, DLL)

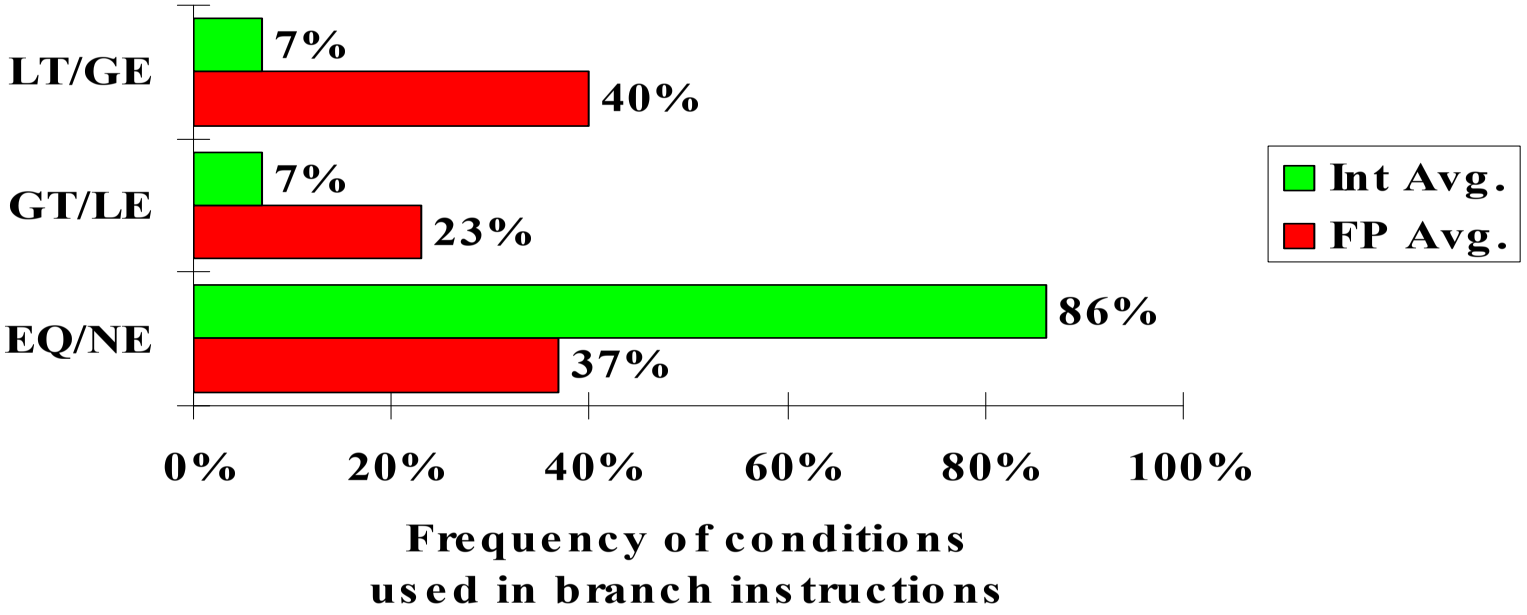
Jak daleko skáče podmíněný skok ?



- Většina podm. skoků je krátká => PC-relativní adresování
- Je třeba alespoň 8-bitů pro displacement
- Absolutní adresace není nutná a lze ji nahradit nepřímou registrovou adresací (RISC procesory)

Nejpoužívanější podmínky ve skocích

- Liší se celočíselné programy od programů v pohybl. čárce
- Srovnání na shodu (EQ/NE) je dominantní v celočíselných programech (86%).
- Většina porovnání je vůči konstantě (s přímým operandem)
- Nejběžnější konstantou je nula => BEQZ a BNEZ instrukce



Implementace podmíněných skoků v ISA

Jedna instrukce – „Compare and Branch“

Ex: BGT R1, R2, label

BLT R1, #57, label

Problémy :

- “Příliš mnoho práce na jednu instrukci”
  => problémy s proudovým zpracováním
- Problém jak zakódovat konstantu pro srovnání a offset skoku (*displacement*) do jednoho kódu instrukce

$\Rightarrow$ **Řešením je oddělení instrukce vyhodnocení podmínky od instrukce skoku.**

Implementace podmíněných skoků – 2 instrukce

Myšlenka :

….

instr1 – vyhodnotí podmínku a výsledek ***někam*** uloží

…. | výsledek podmínky

instr2 – provede podmíněný skok na základě výsledku

***Kam* uložit výsledek vyhodnocení podmínky ?**

- **Implicní místo** => typicky **registr příznaků** **(condition code register)**
- **Explicitní místo** => **dedikované registry** (**condition registers**)

nebo **univerzální registry**

Explicitní místo je specifikováno v instrukci vyhodnocující podmínku. Implicitní místo je pevně dáno pro všechny instrukce vyhodnocující podmínku.

Implementace podmíněných skoků – příznaky

Registr příznaků může být nastaven jako *postranní efekt* ALU instrukce nebo pomocí dedikované instrukce "compare/test". Podmíněný skok se rozhoduje dle nastavení bitů v registru příznaků.

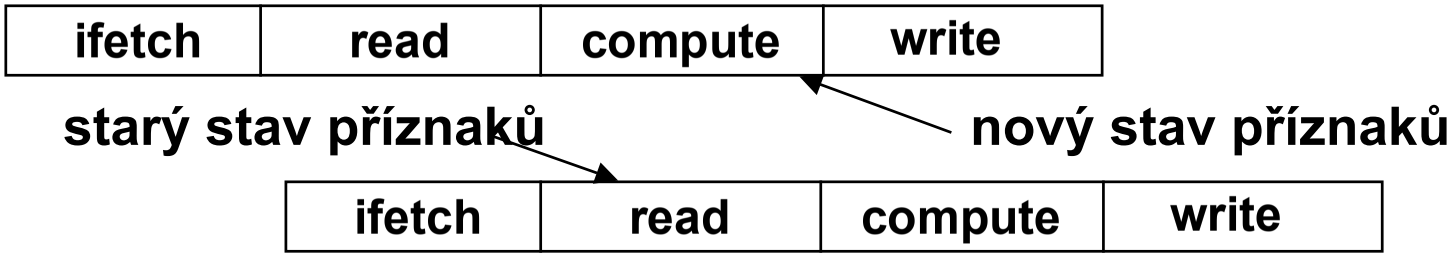
```
X:    .
      .
      .
      SUB  r7, r5, r4
      BRP  X
```

NEBO

```
X:    .
      .
      .
      SUB  r7, #1, r5
      CMP  r7, #0
      BRP  X
```

Nevýhody příznaků

- Ne všechny instrukce nastavují příznaky stejným způsobem
- Jeden registr příznaků se může stát úzkým hrdlem pro proudové zpracování instrukcí



Implementace podm. skoků – podmínkové registry

Power PC

- 8 nezávislých “příznakových registrů” – tzv. „**Condition Registers**“
- Každá ALU instrukce a instrukce srovnání či testu specifikuje jeden z 8 podmínkových registrů
- Složitější podmínky je možné vyhodnocovat bitovými operacemi mezi podmínkovými registry

IA 64 (Itanium)

- 128 dedikovaných 1-bitových podmínkových registrů
  - tzv. **predikátové registry**
- Instrukce porovnání / test může nastavit až dva predikátové registry (jeden při splnění a druhý při nesplnění podmínky)
- IA64 umožňuje **podmíněné provádění každé instrukce** v závislosti na stavu **predikátového registru**
- Podmíněný skok je jen zvl. případ podmíněné instrukce

Implementace podm. skoků – univerzální registry

Podmíněný skok procesoru DLX (podobně MIPS)

BEQZ R5, label **BNEZ R5, label**

= if R5 = ≠ 0 then goto label

- Složitější podmínky mohou být testovány pomocí instrukce srovnání/testu následované podmíněným skokem na základě nulovosti/nenulovosti registru .

Příklad instrukce srovnání procesoru DLX

SLEI R5, R4, 0x15

= if R4<=0x15 then R5:=1;

else R5=0;

SLEI = Set if Less or Equal to Immediate

Shrnutí implementace podmíněných skoků

Existují čtyři základní způsoby implementace:

- Jedna instrukce srovnání a skoku
- Řešení na bázi registru příznaků
- Podmínkové a predikátové registry
- Řešení na bázi univerzálních registrů

Volání a návraty z podprogramů

Kam uložit návratovou adresu ?

Jak předávat parametry a návratové hodnoty ?

Kde alokovat místo pro lokální proměnné, mezivýsledky ?

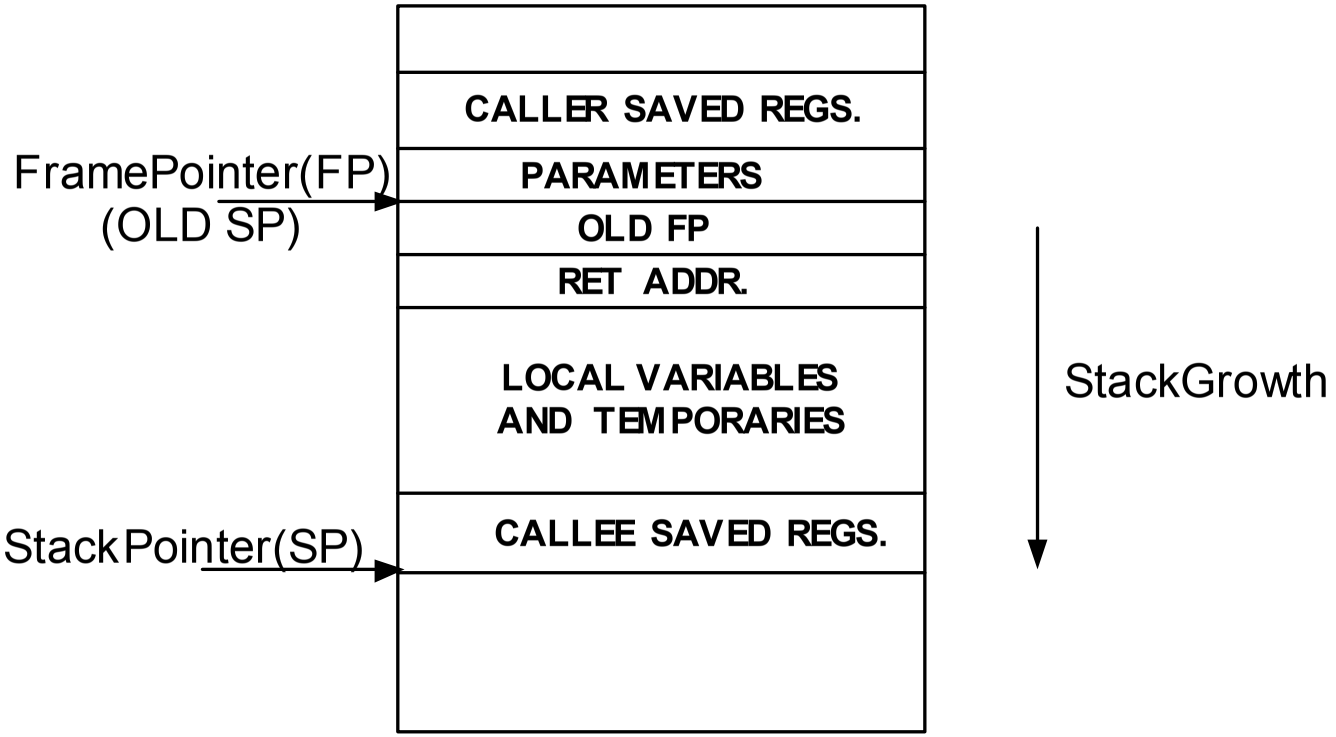
Tradiční (CISC) odpověď

- Všechno na **zásobník** (beztak máme málo registrů) !

Moderní (RISC) odpověď

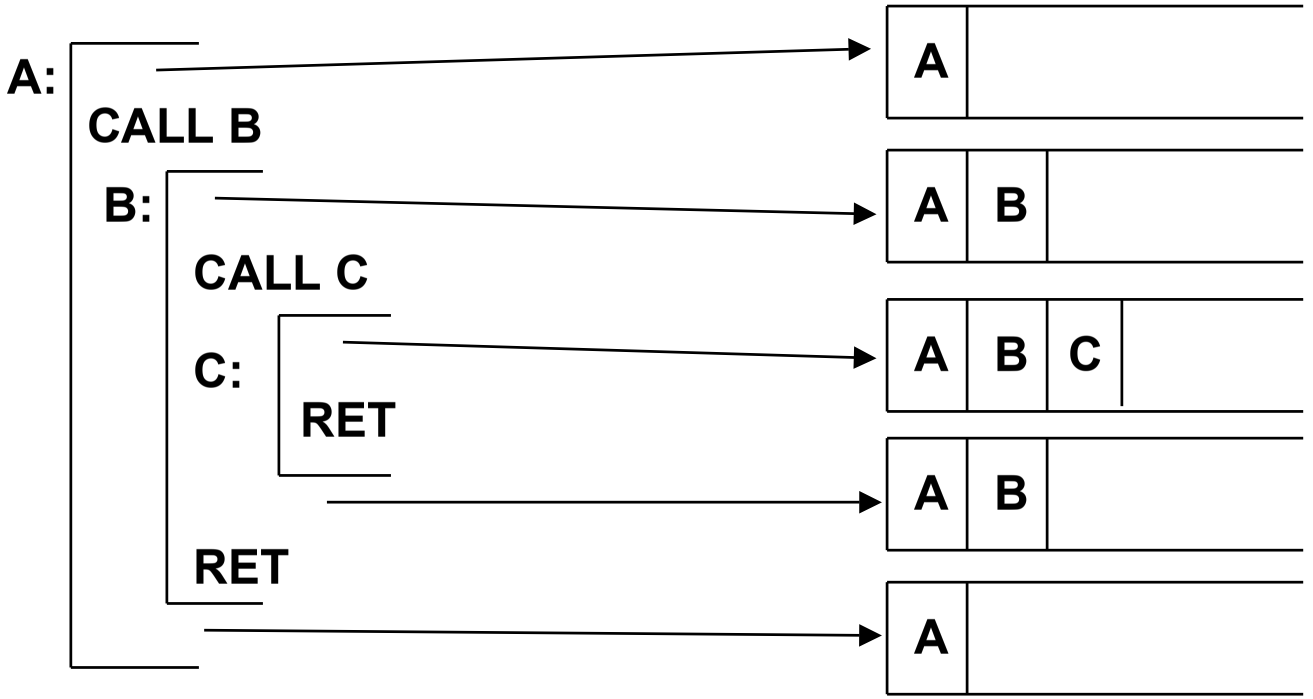
- Především zvaž úplnou eliminaci volání pomocí *inliningu*
- Především používej registry, jen pokud to nelze jinak, použij **zásobník**.

Volání podprogramů – aktivační záznamy



– Přesné rozložení položek na zásobníku závisí na programovacím jazyku a systémových konvencích (tzv. Application Binary Interface = ABI).
– Data v aktivačním záznamu můžeme adresovat relativně vůči SP nebo FP
– Zásobník může též obsahovat odkazy na hierarchicky nadřazený aktivační záznam (Tak se zpřístupní viditelné lokální proměnné hierarchicky nadřazených procedur např. v Pascalu)

Volání podprogramů – zásobník aktivačních záznamů



Tradiční architektury obsahují dedikovaný registr SP a někdy i FP a související instrukce PUSH a POP (např. VAX, x86, DOP)

Některé RISC architektury implementují zásobník jako SW konvenci (např. MIPS, DLX). Jeden z univerzálních registrů slouží jako „SP“. „PUSH“ je instrukce STORE , kde adresa je v tomto registru.

MIPS - softwarové konvence použití registrů

0	zero	constant 0 (HW)
1	at	reserved for ASM
2	v0	expression evaluation
3	v1	return values
4	a0	parameters
5	a1	
6	a2	
7	a3	
8	t0	temporaries
….	(caller saves)	
15	t7	

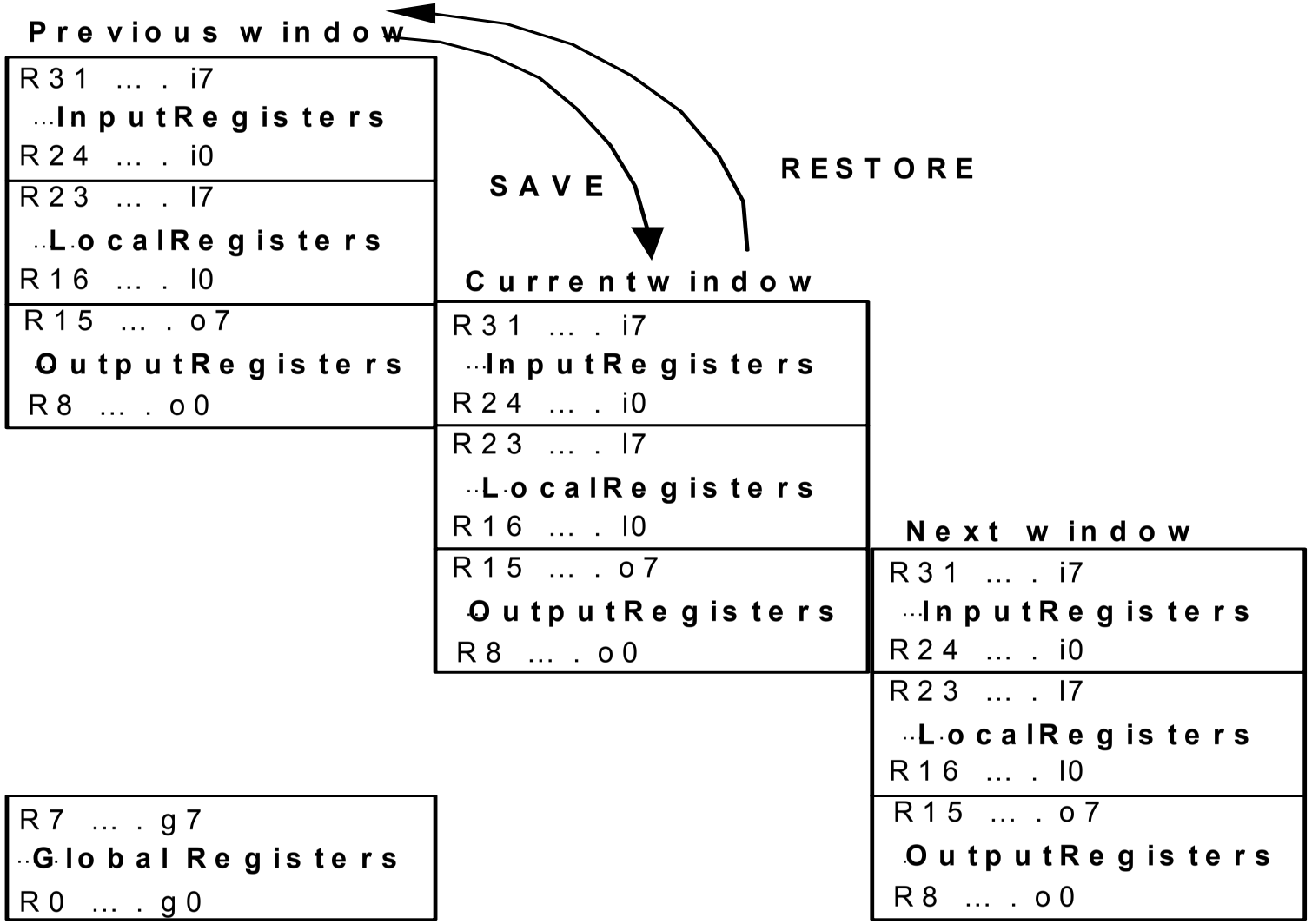
16	s0	
. . .	(callee saves)	
23	s7	
24	t8	temporaries (cont.)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to globals
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

SPARC: registrová okna

SPARC dělí GPR do následujících skupin
– 8 globální registry g0 … g7 – g0=0
– 8 vstupní registry i0 .. i7
– 8 výstupní registry o0 …o7
– 8 lokální registry l0 … l7

Princip použití registrových oken:
- Výstupní registry volajícího se stanou vstupními registry volaného
- Volaný získá nové výstupní a lokální registry
- o6 = stack pointer SP
- i6 = frame pointer FP
  (díky mechanismu oken je přiřazení FP=old SP automatické)
- i7 obsahuje návratovou adresu (na volajícího)
- i0 slouží k předání návratové hodnoty volajícímu

SPARC: registrová okna



SPARC: registrová okna - realizace

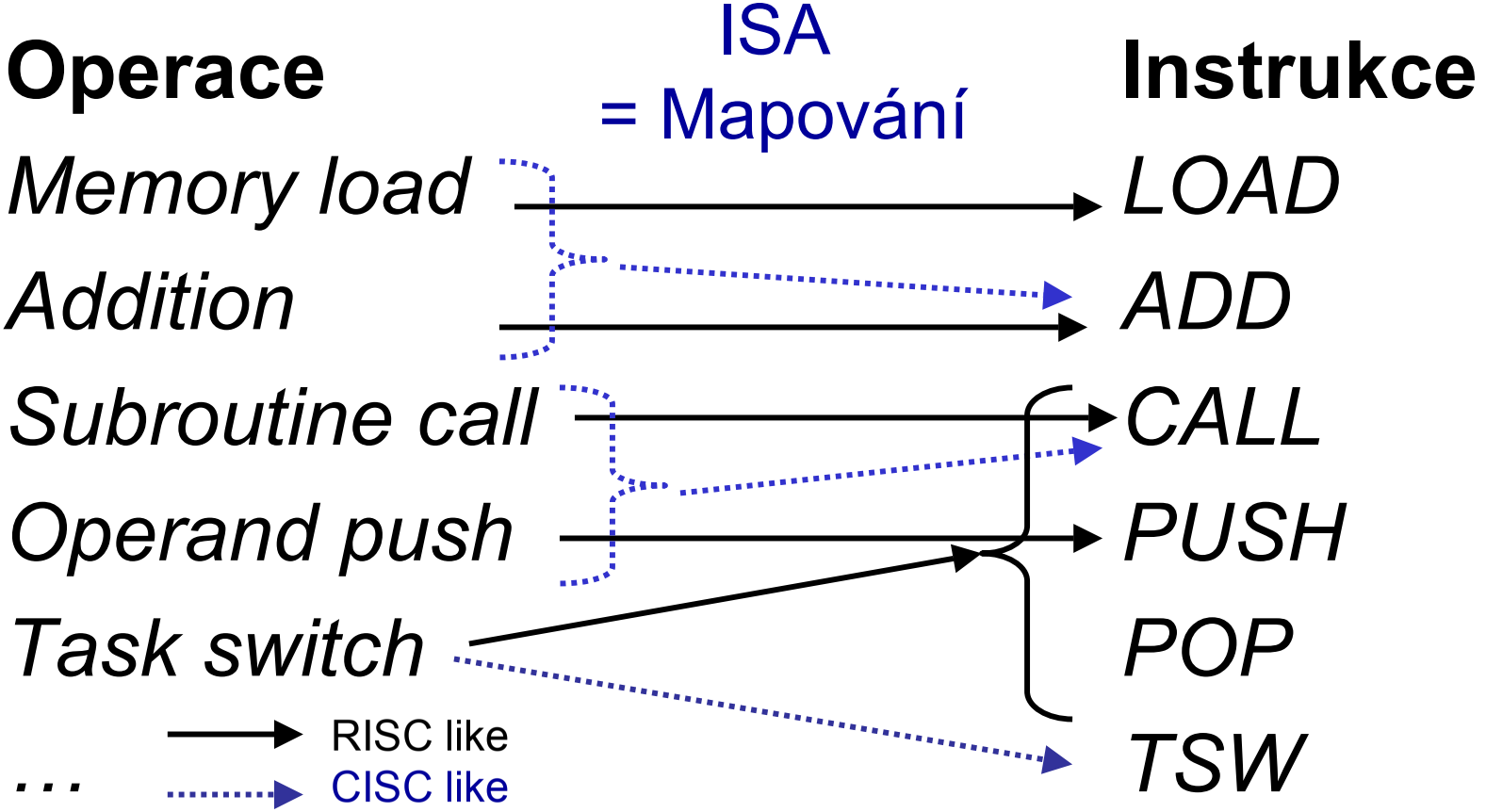
o Registrová okna jsou mapována na fyzické registrové pole organizované jako kruhový buffer
o Počet fyzických registrů závisí na implementaci (dle normy 40 – 520) – typicky 128-144 registrů
o Rámec na zásobníku je automaticky vytvořen k případné úschově při přetečení registrových oken a pro další parametry a lokální proměnné.

Problémy registrových oken

- Přetečení registrových oken je řešeno jako SW přerušení => doba provádění programu není deterministická
- Registrové pole s mechanismem registrových oken může vést ke snížení dosažitelné T_{clk}

IA64 –Itanium používá podobný mechanismus (ještě složitější)

Operace vs Instrukce



Instrukce kóduje jednu či více operací.
Komplexní operace je mapována na jednu **komplexní instrukci** nebo na **posloupnost jednoduchých instrukcí.**
ISA definuje toto **mapování** mezi **operacemi** a **instrukcemi**

RISC vs CISC

Jak složitá operace má být zakódovánav jedné instrukci ?

Je lepší mít jednu komplexní instrukci nebo operaci skládat pomocí více jednoduchých instrukcí ?

Odpověď na tyto otázky se v čase měnila (a mění?) v závislosti na:
- výzkumu v **architektuře počítačů**
- možnostech **implementační technologie**
- schopnostech a úloze **překladačů**

Komplexní ISA dominuje v 70. letech

Technologické parametry:

- hlavní paměť je pomalá a drahá
- ROM mikroprogramu je levnější a mnohonásobně rychlejší než hlavní paměť

=> Doba načtení instrukce (instruction fetch time) je dominující složkou doby provádění instrukce

Komplexní instrukce v takové situaci

- zmenší počet instrukcí (IC) v programu (ušetří paměť)
- urychlí dobu provádění programu (ušetří dobu načítání a dekódování instrukcí)

Komplexní ISA dominuje v 70. letech

Výzkum architektury počítačů v 70. letech:

- Hardware by měl uzavřít "sémantickou mezeru" vůči vyšším programovacím jazykům (tzv. High-level Language Computer Architecture)
- Budoucí počítače budou vykonávat přímo instrukce vyšších programovacích jazyků
- Toto vyřeší též problém "programové záplavy" a rostoucí cenu vývoje software => "softwarovou krizi"
- Vše vyřeší "chytřejší" hardware
- ISA je studována teoreticky bez vyhodnocení složitosti její implementace a reálného chování skutečných počítačů

Komplexní instrukce tedy

- uzavřou "sémantickou mezeru"
- zabrání "programové záplavě"
- zjednoduší překladače a ladění programů
- vyřeší „softwarovou krizi"

Použití instrukcí u x86 – kde jsou ty komplexní ?

Rank	*Instruction*	*Frequency*
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
Total		96%

10 instrukcí tvoří **96%** instrukcí v programech.
(výsledky z měření SPECInt95)

Adresní módy – Které z nich má ISA podporovat ?

Adresní mód	Syntaxe	Sémantika
Registrový	ADD R4,R3	R4← R4+R3
Přímý operand	ADD R4,#3	R4 ← R4+3
Bázovaný s offsetem	ADD R4,100(R1)	R4 ← R4+Mem[100+R1]
Registrový nepřímý	ADD R4,(R1)	R4 ← R4+Mem[R1]
Bázovaný/indexovaný	ADD R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Přímá/absolutní adresa	ADD R1,(1001)	R1 ← R1+Mem[1001]
Paměťový nepřímý	ADD R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
S post-inkrementací	ADD R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
S pre-dekrementací	ADD R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Škálovaný	ADD R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Který adresní mód je nejpoužívanější ?

SPECInt měření na počítači DEC VAX který podporuje téměř všechny adresní módy

- **Bázovaný s ofsetem:** 42% prům. 32% - 55%
- **Přímý operand:** 33% prům., 17% - 43%
- **Registrový nepřímý:** 13% prům., 3% - 24%
- **Škálovaný:** 7% prům., 0% - 16%
- **Paměťový nepřímý:** 3% prům., 1% - 6%
- **Ostatní:** 2% prům., 0% - 3%

75%

88%

Velikost přímého operandu :
- 50% - 60% se vejde do 8 bitů
- 75% - 80% se vejde do 16 bitů

Velikost ofsetu :
- pouze 1% offsetu se nevejde do 16 bitů

Argumenty pro RISC

Kvantitativní studie existujících počítačů nepotvrzuje, že cesta směrem ke komplexním instrukcím je správná:

- Komplexní instrukce a adresní módy jsou málokdy využívány (komplexnější instrukce = méně používaná instrukce)
- 10 % instrukcí je využíváno 90 % času

  => Amdahlův zákon doporučuje soustředit se na těchto 10 % instrukcí
- Spousty chyb ve složitých mikroprogramech implementujících komplexní instrukce (nutno distribuovat záplaty mikrokódu)
- Málo používané komplexní instrukce vyžadující kódování instrukcí proměnlivé délky zpomalující provádění používaných jednoduchých instrukcí
- Paměťová hierarchie (cache) umožní zkrátit dobu načítání instrukcí
- Překladače ne hardware uzavřou "sémantickou mezeru" !

Počítače s redukovanou složitostí instrukcí jsou nazývány RISC.
Tradičním architekturám je následně vymyšlen název CISC.

Proč jsou (potenciálně) RISC efektivnější ?

- **Mikroprogramování není třeba, řídící paměť uvolní místo více registrům či skryté paměti (cache)**
- **Snadnější implementace proudového zpracování (*pipeliningu*)**
- **Více registrů umožní efektivnější využití překladačem**
- **Jednodušší návrh je snadno a dříve dokončen**
- **Jednodušší procesor může běžet na vyšší hodinové frekvenci**

RISC vs CISC (zjednodušené srovnání výkonnosti)

Tradiční CISC mají průměrné CPI mezi 5 – 10 takty,

RISC mají průměrné CPI mezi 1.3 a 2 takty (díky *pipeliningu*).
Počet instrukcí (IC) programů pro RISC je pouze o málo vyšší než u CISC procesorů

=> Tento fakt dává RISC procesorům výkonnostní náskok oproti tradičním CISC procesorům.

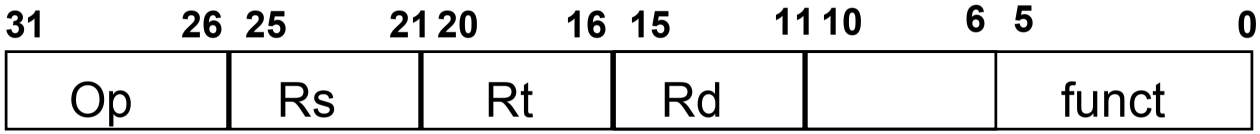
“Typický 32-bitový RISC” - DLX

- 32-bit kódování instrukcí pevné délky (3 formáty)
- 32 32-bitových univerzálních registrů (R0=0)
- 32 FP registrů, registry dvojnásobné přesnosti v párech
- Registr – Registr (Load-Store) (3,0) GPR ISA
- ALU operace jsou typu registr-registr a registr-přímý operand (16-bitů přímý operand)
- Jeden adresní mód pro instrukce load/store: bázovaný + 16-bitový offset (base/displacement)
- Jednoduché PC-relativní podmíněné skoky, 16-bitový offset (displacement)

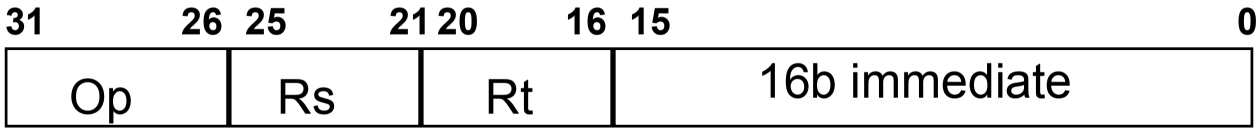
Podobně : SPARC, MIPS, HP PA-RISC, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Kódování instrukcí DLX (MIPS)

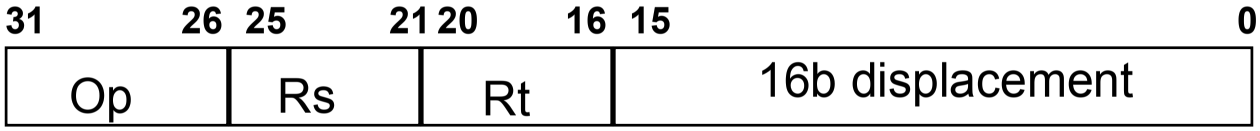
Register-Register (R type)



Register-Immediate (I Type)



Branch, Load, Store (I Type)



Jump / Call (J type)

Chronologie RISC vs CISC

Konec 70.let - počátek 80.let : „RISC movement“

- John L. Henessy, David A. Patterson a další
- RISC I, RISC II, MIPS (projekty na UCB a Stanfordu)

80. léta – polovina 90.let: Komercializace RISC

- RISC architektury jsou výrazně výkonnější než CISC
- RISC ovládají trh pracovních stanic, prvních serverů a vestavné aplikace. Jedním z posledních CISC je x86 …
- Všechny nové počítače navrženy jako RISC

Konec 90. let a současnost : “Post-RISC era”

- x86 dosahuje srovnatelnou i vyšší výkonnost než RISC při nižší ceně => vytlačuje RISC počítače z „low end“ trhu a tlačí se stále výše …
- „PowerPC G5 má více instrukcí a je stejně složité jako Pentium 4“
- “Na ISA nezáleží.”,“RISC a CISC architektury konvergovaly. ”
- „Na obou přístupech je něco pravdy“ ?

Co se stalo koncem 90. let ?

Pokroky v technologii i architektuře superskalárních procesorů umožnily

- implementaci superskalárních CISC s využitím technik RISC
- moderní CISC procesor se od RISC odlišuje zejména v tzv. "frontendu" kde dochází k překladu komplexních instrukcí na „RISCovské" "mikroinstrukce", tyto mikroinstrukce jsou prováděny na "high-performance substrate" (superskalárním RISC procesoru)
- technologie tedy umožňuje minimalizovat vliv některých inherentních nevýhod CISC architektury

Záleží tedy na ISA ?

- x86 stále zůstává méně efektivní z hlediska optimalizace překladače
- Velký podíl na trhu ovšem umožňuje implementovat x86 ve velkých objemech na pokročilejších technologiích než konkurenční RISC architektury a tím nevýhody kompenzovat
- Na ISA stále záleží ve vestavných aplikacích, které jsou dominovány RISC procesory. Luxus hardwarového překladu instrukcí není zde možný (spotřeba a efektivita implementace je zde více kritická).

Poslední trendy v ISA

Nové oblasti aplikací (90.léta)

- multimédia, grafika – speciální aritmetické instrukce
- počítačová bezpečnost
- podpora virtualizace počítače (Java, .NET, více OS)

Podpora ISA pro větší výkonnost – datový paralelismus

"SIMD instrukce"

- jedna instrukce pracující s krátkými datovými vektory
- 128b = 16 x 8b, 8 x 16b, 4 x 32b, 2 x 64b
- operace podporující multimediální aplikace (saturační aritmetika, celočíselná i pohyblivá řád. čárka)
- efektivnější využití registrů a datové cesty
- podpora knihoven a také překladačů

Komerční příklady:

MMX, 3DNow! ™, SSE, SSE2, PowerPC AltiVec™, Sun VIS™

Poslední trendy v ISA

Podpora ISA pro větší výkonnost – instrukční paralelismus (ILP)

"VLIW instrukce"

- VLIW = Very Long Instruction Word
- jedna "dlouhá instrukce" = povel k více (4-16) operacím, které mohou být provedeny paralelně
- VLIW může potenciálně využít více instrukčního paralelismu (ILP) než stávající superskalární procesory s tradiční ISA
- VLIW by měl umožnit implementaci jednoduššího procesoru než je stávající superskalární RISC/CISC
- VLIW ISA se používají úspěšně v DSP a Media procesorech (vestavné aplikace)

Myšlenky VLIW v univerzálních počítačích – úspěch či neúspěch ?

-Transmeta Crusoe (VLIW s JIT překladačem x86)

-Intel IA64 (EPIC) – komplexní ISA kombinující VLIW s predikátováním a dalšími technikami softwarové spekulace

Architektury souborů instrukcí - shrnutí

- Základní třídy ISA jsou **střadačově-orientovaná, zásobníkově-orientovaná** a **GPR**

- GPR ISA je dnes v počítačích převažující

- Střadačová a zásobníková ISA jsou stále používány v určitých aplikacích

- Ukázali jsme si různé možnosti při návrhu ISA a jejich vlastnosti

- RISC ISA je podložena kvantitativní analýzou využití souboru instrukcí

- Soudobým trendem v ISA je podpora nových aplikací a vyšší výkonnosti počítače