Procesory RISC (a CISC)

Studijní materiál pro předmět
Architektury počítačů a paralelních systémů

Ing. Petr Olivka, Ph.D.
katedra informatiky FEI VŠB-TU Ostrava
email: petr.olivka@vsb.cz

Ostrava, 2020

1 Procesory RISC a CISC

V dnešní době se ustálilo dělení počítačů do dvou základních kategorií podle typu použitého procesoru:

- **CISC** - počítač se složitým souborem instrukcí (Complex Instruction Set Computer)
- **RISC** - počítač s redukovaným souborem instrukcí (Reduced Instruction Set Computer)

Ať už je toto dělení jakkoli nepřesné, je třeba jej považovat za obvyklé. V dnešní době totiž prakticky neexistují procesory, které by nesly „čisté“ rysy RISC a CISC, vždy jde o kompromis mezi oběma směry.

Z hlediska historického vývoje je důležité se věnovat okamžiku, kdy se vývoj procesorů rozdělil do dvou větví. Vývoj procesorů, které zpětně dostaly označení CISC, směřoval na konci 70. let k nezadržitelnému růstu jejich složitosti a tak se objevily první pokusy o celkové zjednodušení struktury procesorů. Vznikla tak zcela nová kategorie procesorů, dnes označovaná jako RISC.

1.1 Vznik procesorů RISC

V 70. letech výzkumy četnosti výskytu instrukcí ukázaly, že programátoři a kompilátory používají strojové instrukce velmi nerovnoměrně. Počet nejfrekventovanějších instrukcí je omezen na velmi úzkou část instrukčního souboru.

		%
LOAD	čtení z paměti	26.6
STORE	zápis do paměti	15.6
Jcond	podmíněný skok	10.0
LOADA	načtení adresy	7.0
SUB	odčítání	5.8
CALL	volání podprogramu	5.3
SLL	bitový posun vlevo	3.6
IC	vložení znaku	3.2

Tabulka 1: Statická četnost instrukcí

V tabulce 1 je vidět statickou četnost (podle výskytu instrukcí v programu) a v tabulce 2 dynamickou četnost (podle frekvence vykonávání) instrukcí programů počítače IBM System/360.

V tabulce je vidět, že v 50% případů se vyskytují pouze 3 instrukce a v 75% případů jen 8 instrukcí! Četnost výskytu ostatních instrukcí je velmi malá, v některých případech jen zlomky promile. Je tedy na zvážení, zda

		%
LOAD	čtení z paměti	27.3
Jcond	podmíněný skok	13.7
STORE	zápis do paměti	9.8
CMP	porovnání	6.2
LOADA	načtení adresy	6.1
SUB	odčítání	4.5
IC	vložení znaku	4.1
ADD	sčítání	3.7

Tabulka 2: Dynamická četnost instrukcí

některé instrukce vůbec do instrukčního souboru zařadit a rozšiřovat kvůli nim řadič procesoru. Obdobná statistika, jako pro četnost instrukcí, platí i pro stupeň využití obvodů řadiče.

Uvedené výzkumy vedly na konci 70. let ke snaze o nalezení optimálního instrukčního souboru jeho omezením. Tím vznikly počítače RISC. Hlavními protagonisty byly v té době zejména dvě kaliforské univerzity: Univerzita státu Kalifornie v Berkeley a Stanfordova Univerzita. A také firma IBM.

Na univerzitě v Berkely vznikl v roce 1982 procesor *RISCI*, od jehož názvu je odvozen celý směr vývoje počítačů. V roce následujícím byl na Stanfordově univezitě realizován procesor *Mips* (Microprocessor without interlocked pipelining stages - procesor bez vzájemně se blokujících sekcí proudového zpracování). Z těchto projektů pak vzniklo několik generací průmyslově vyráběných RISC procesorů.

První sériové počítače RISC byly na trh dodávány až v polovině 80. let, ale vyráběly se technologií se stupněm integrace MSI a byly velmi rozměrné. Skutečný rozvoj nastal až s využitím vyššího stupně integrace v sériové výrobě.

Vývoj mikroprocesorů typu RISC je neoddělitelný od vývoje řady CISC. Vyplývá to z letité zkušenosti vývojářů s vývojem a realizací CISC procesorů. Dnes neexistuje žádný čistý RICS nebo CISC procesor. Každý moderní procesor v sobě uplatní rysy z obou kategorií. Je spíše věcí rozhodnutí výrobce a jeho marketingu, kam bude jejich produkt zařazen. Může se rozhodovat podle toho, jakých vlastností má procesor více, ale často je to obtížné.

1.2 Vlastnosti architektury RISC

Bezesporu nejtypičtější vlastností charakterizující počítače RISC je malý instrukční soubor. Pro tuto vývojovou větev procesorů si ale vývojáři stanovili celou řadu dalších vcelku zásadních kritérií, charakterizujících metodiku návrhu nejen procesoru, ale celého počítače. Procesoru je třeba přenechat jen tu činnost, která je nezbytně nutná a přenést další potřebné funkce do

architektury počítače, programového vybavení a kompilátoru.

Výsledkem návrhu konstrukce procesoru RISC jsou zejména tyto vlastnosti:

- v každém strojovém cyklu by měla být dokončena jedna instrukce (pozor, to neznamená, že její vykonání trvalo jeden stroj. cyklus),
- mikroprogramový řadič může být nahrazen rychlejším obvodovým řadičem,
- používat zřetězené zpracování instrukcí,
- celkový počet instrukcí a způsobů adresování je malý,
- data jsou z hlavní paměti vybírána a následně ukládána výhradně jen pomocí dvou instrukcí LOAD a STORE,
- instrukce mají pevnou délku a jednotný formát, který vymezuje význam jednotlivých bitů,
- je použit vyšší počet registrů,
- složitost se z technického vybavení přesouvá částečně do optimalizujícího kompilátoru.

Všechny uvedené vlastnosti tvoří dobře promyšlený a provázaný celek. Např. navýšení počtu registrů souvisí s omezením komunikace s pamětí na dvě instrukce LOAD a STORE. Pokud ostatní instrukce nemohou používat paměťové operandy, je třeba mít v procesoru uloženo více dat. Další souvislost je patrná mezi zřetězeným zpracováním a formátem instrukcí. Jednotná délka instrukcí dovoluje rychlejší výběr instrukcí z paměti a tím zajišťuje lepší plnění fronty instrukcí (viz další kapitola). Jednotný formát pak zjednodušuje dekódování instrukcí.

Výsledné počítače vytvořené podle těchto pravidel přinášejí výhody jak pro uživatele, tak i pro výrobce. Zkracuje se vývoj procesoru a zpravidla již první realizované čipy fungují správně. Architektura RISC má i své nedostatky, přesto se většina výrobců CISC procesorů uchýlila při výrobě procesorů k realizaci stále většího počtu vlastností architektury RISC.

Mezi nevýhody RICS architektury patří třeba nutný nárůst délky programů, tvořených omezeným počtem instrukcí a také díky jednotné délce všech instrukcí. Zpomalení, které by z toho mělo nutně plynout, se ale v praxi nepotvrdilo. Je třeba si uvědomit, jak malé procento instrukcí muselo být rozepsáno a zřetězené zpracování instrukcí zásadním způsobem urychluje práci procesoru.

Obrázek 1: Procesor (CISC) jako sekvenční obvod

Krok		Význam
1.	**VI**	Výběr Instrukce
2.	**DE**	Dekódování
3.	**VA**	Výpočet Adresy
4.	**VO**	Výběr Operandu
5.	**PI**	Provedení Instrukce
6.	**UV**	Uložení Výsledku

Tabulka 3: Příklad možných kroků zpracování instrukce

1.3 Proudové zpracování instrukcí

Procesor si lze představit jako sekvenční obvod podle obrázku 1. Vstupem jsou strojové instrukce a data z paměti. Z výstupu se data ukládají zpět do paměti. Každá instrukce tedy musí projít celým obvodem a dokud se neuloží výsledky, nelze začít provádět instrukci následující.

Provedení instrukce musí projít vždy stejnými fázemi. Ve zjednodušené variantě je možno si představit, že instrukce se musí vybrat z paměti, dekódovat, vypočítat adresa operandu, připravit data, instrukci provést a nakonec uložit výsledky. Pro přehlednost jsou jednotlivé kroky zpracování instrukce názorně uvedeny v tabulce 3.

Pokud bude pro provedení jednoho elementárního úkonu potřeba jeden strojový cyklus, je možno provádění instrukcí názorně zobrazit v tabulce 4. Instrukce I_2 se nezačne vykonávat dříve, než je uložen výsledek instrukce I_1. Bude-li jeden časový cyklus označen jako T_M, je potřeba pro vykonání jedné instrukce 6 cyklů. Provedení každé další instrukce pak tedy bude vyžadovat opět 6 cyklů. Zatím ovšem nepožadujeme, aby měly všechny cykly stejnou délku.

Kdyby se nám tedy podařilo osamostatnit jednotlivé části sekvenčního obvodu z obrázku 1 na samostatné obvody, aby každému obvodu odpovídala jedna fáze zpracování instrukce, mohli bychom si jej představit jako posloupnost na sebe navazujících zřetězených jednotek (v podstatě jde o princip výrobní linky, jak je známo z mnoha jiných odvětví). Obvod by pak mohl být realizován podle schématu na obrázku 2.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}
VI	I_1						I_2						...
DE		I_1						I_2					
VA			I_1						I_2				
VO				I_1						I_2			
PI					I_1						I_2		
UV						I_1						I_2	

Tabulka 4: Postup provádění instrukcí procesorem CISC

Mezi jednotlivé fáze zpracování však musí být pro mezivýsledky zařazen registr, který slouží jako předávací místo mezi po sobě jdoucími obvody. Z tohoto předávání mezivýsledků plyne jisté malé zpoždění. To lze ovšem v celkovém přínosu zřetězení zanedbat.

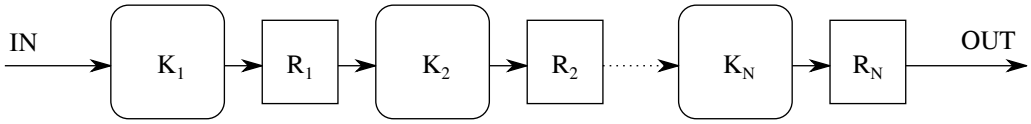
Ještě ovšem nebyl vysloven jeden důležitý předpoklad. Aby uvedený princip zřetězení měl co největší přínos, musí být všechny fáze zpracování stejně časově náročné. Jinak je jasné, že nejpomalejší článek zřetězení bude brzdit všechny ostatní.

Když se podaří rozdělit vykonávání instrukce na jednotlivé úkony, přičemž časový cyklus potřebný pro každou fázi zpracování bude stejný, je možné zpracovávání instrukcí $I_1 \div I_7$ znázornit v tabulce 5. Je vidět, že pro vykonání 7 instrukcí stačí 12 cyklů. V tabulce 4 byly za stejný počet cyklů vykonány pouze 2 instrukce.

Teoreticky to znamená, že v nekonečném čase nám N-úrovní zřetězení zrychlí vykonávání instrukcí $N\times$. V praxi je ovšem celkový přínos zřetězení limitován mnoha dalšími hledisky. Jednak je omezena rychlost na vstupu a výstupu zřetězené jednotky a taky dochází během vykonávání k problémům s používáním omezeného množství pomocných obvodů (registry, sběrnice, atd.). Zmíněno bylo i zpomalení předáváním mezivýsledků.

Jako nejkritičtější se ovšem ukazuje problém plnění zřetězené jednotky - a tím vzniklé fronty rozpracovaných instrukcí. Zejména podmíněné skoky znehodnocují frontu rozpracovaných instrukcí (instrukce se zpracovávají sekvenčně a pokud se vykonávání programu přesune na jinou adresu podmíněným skokem, můžeme již rozpracované instrukce zahodit). A čím je hloubka zřetězení větší, tím se zvyšují i ztráty (zde je vidět, jak může být větší hloubka zřetězení kontraproduktivní). Z tabulky 2 víme, že počet skokových instrukcí v programech je velmi vysoký, prakticky každá šestá instrukce představuje podmíněný skok.

Na základě známých technický parametrů lze hledat optimální hloubku zřetězení. Tímto problémem se ale zde zabývat nebudeme.



Obrázek 2: Zřetězené zpracování (v procesoru RISC)

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
VI	I_1	I_2	I_3	I_4	I_5	I_6	I_7	...				
DE		I_1	I_2	I_3	I_4	I_5	I_6	I_7				
VA			I_1	I_2	I_3	I_4	I_5	I_6	I_7			
VO				I_1	I_2	I_3	I_4	I_5	I_6	I_7		
PI					I_1	I_2	I_3	I_4	I_5	I_6	I_7	
UV						I_1	I_2	I_3	I_4	I_5	I_6	I_7

Tabulka 5: Zřetězené provádění instrukcí procesorem RISC

1.4 Problémy zřetězeného zpracování

Zřetězené zpracování instrukcí přináší nejen výrazné navýšení výkonu procesoru, ale nese s sebou i řadu nástrah, úskalí a problémů. Mezi hlavní problémy, které budou dále popsány, patří hazardy datové i strukturální a problémy s plněním fronty instrukcí.

1.4.1 Datové a strukturální hazardy

Datové hazardy vznikají např. tehdy, když některá rozpracovaná instrukce potřebuje mít k dispozici data předchozí instrukce, a ta ještě nejsou k dispozici. Stačí si představit podle tabulky 5 situaci, kdy instrukce I_5 potřebuje pro výpočet adresy v čase T_7 hodnotu, kterou instrukce I_4 uloží až v čase T_9. K řešení problému musí být uzpůsobena zřetězená jednotka svou konstrukcí, nebo se tyto problémy řeší už v překladači, aby se podobným situacím zabránilo.

Strukturální hazardy můžeme charakterizovat zjednodušeně jako problém omezených prostředků procesoru i počítače jako celku. Jako příklad si můžeme uvést problém, se kterým se budou potýkat některé části zřetězené jednotky: při výběru instrukce, při výběru operandu i při ukládání výsledku, bude potřeba komunikovat po sběrnici. Ta je ale k dispozici obvykle pouze jedna a nelze na ni povolit přístup více jednotkám současně. Přístup na sběrnici je tedy třeba koordinovat, což přinese zpomalení práce.

Uvedené problémy jen velmi okrajově naznačují, jak složitá je problematika zřetězeného zpracování a vznikajících hazardů.

1.4.2 Problémy plnění fronty instrukcí

Pro optimální činnost zřetězeného zpracování je důležitá reakce na skokové instrukce. Nejjednodušší je řešení nepodmíněných skoků a volání podprogramů s pevnou adresou. Zřetězená jednotka může být snadno upravena a začne vybírat další instrukce z cílové adresy.

Trochu komplikovanější je to v případě, že cílová adresa nepodmíněného skoku se musí vypočítat. Výsledek výpočtu může být dán již rozpracovanými instrukcemi a výsledek není k dispozici dostatečně brzo a hrozí výpadek fronty. Omezit četnost těchto případů lze např. optimalizací pořadí strojových instrukcí. To je úkol zejména pro optimalizující překladače z vyšších programovacích jazyků.

Vážný problém nastane ovšem v okamžiku, kdy se při zřetězením zpracování instrukcí narazí na podmíněný skok. Podobně jako u předchozího případu, kdy jsme neznali cílovou adresu, tady sice adresu (většinou) známe, ale nevíme, jestli se skok provede, či nikoliv. Nemá smysl zpracovávání zastavit a čekat na výsledek. Lepší je pokračovat ve zpracovávání sekvenčním způsobem a pokud se skok neprovede, tak se rozpracovaná fronta instrukcí použije. V opačném případě se rozpracované instrukce ignorují a fronta se začne plnit znovu.

Velmi dobré řešení se používá pro vysoce výkonné počítače, kde se implementují dvě paralelní fronty. Druhá fronta je obvykle kratší, podle našeho příkladu z tabulky 3 může být omezena jen na první čtyři kroky zpracování. Prováděcí jednotka pak může začít vybírat alternativně připravené instrukce z druhé fronty. Je jasné, že přepínání fronty zabere určitý čas. Tato ztráta je ovšem výrazně výhodnější, než výpadek celé fronty.

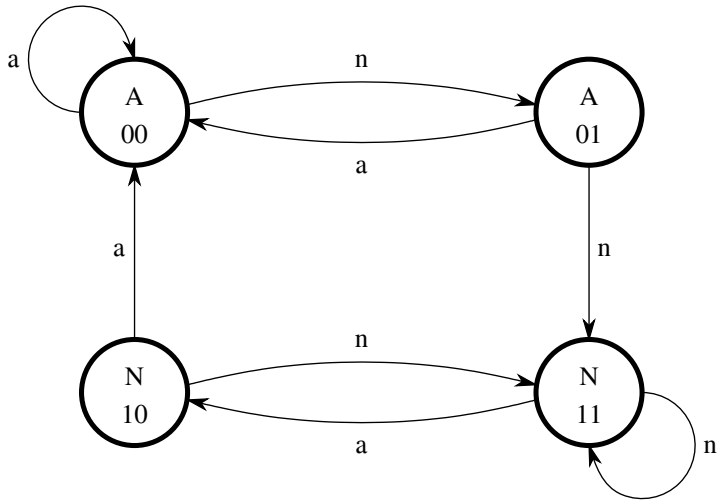
V procesorech RISC se kromě dvou uvedených krajních řešení používají i další metody, jako např. predikci skoků.

1.5 Metody pro zlepšení plnění fronty instrukcí

Procesory RISC používají vždy frontu instrukcí. Aby bylo riziko ztráty již vybraných instrukcí z paměti co nejmenší, je třeba omezit možnosti, jak sekvenční vykonávání instrukcí měnit.

V moderních systémech se předpokládá, že program nemůže modifikovat sám sebe. Dále pak podmíněné skoky jsou vždy na pevně danou adresu a pouze nepodmíněné skoky mohou používat registry pro určení cílové adresy skoku.

V praxi statistiky ukázaly, že z výše uvedených problémů se nejčastěji vyskytují podmíněné skoky. Jejich četnost je podle typu řešených úloh $15 \div 25\%$. Vyplatí se proto hledat jednoduchá a levná řešení, jak chování podmíněných skoků předpovídat.



Obrázek 3: Dvoubitová predikce jako stavový automat

1.5.1 Bit predikce skoku

Tato metoda předpokládá, že se ve formátu instrukce vyhradí jeden bit predikující, zda se skok provede či nikoliv. Jednotka výběru instrukcí pak vybírá instrukce z předpokládané adresy.

Predikce může být statická, nebo i dynamická. U statické metody se do instrukce vkládají příslušné bity predikce již při kompilaci, nebo přímo programátorem při tvorbě programu. Dynamická predikce si při každém provedeném podmíněném skoku zaznamená, jestli se skok provedl, či nikoliv. Dynamická metoda je výhodnější, protože se predikce přizpůsobí aktuálním podmínkám. Vyžaduje ovšem nutnost příslušný bit predikce zapisovat a to může být technicky komplikované, případně i nemožné (kód programu je chráněn proti přepisování, případně může být uložen v nepřepisovatelné paměti).

Jednobitová predikce je velmi jednoduchá, ale její použití v podmínce cyklu znamená, že dojde k jednomu selhání vždy na začátku cyklu, a k jednomu na konci (u dynamické metody). Pokud si ale představíme dva nebo i více vnořených cyklů, je četnost selhání výrazně vyšší.

Lepší chování nabízí predikce dvoubitová. Ta dokáže snížit u cyklů selhání predikce na jediné a to na konci cyklu. Chování dvoubitové predikce si můžeme znázornit stavovým automatem na obrázku 3.

Jde o čtyřstavový automat, kde stav A predikuje provedení skoku, zatím co stav N říká, že skok se provádět nebude. Přechody a a n označují, zda se naposledy skok prováděl, či nikoliv. Je tedy patrné, že pouze pokud se skok dvakrát po sobě provede, může se změnit predikce z N na A a obráceně. V ustáleném stavu se tedy predikce nastaví na $A - 00$ nebo $N - 11$.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
VID	I_1	I_2	I_3	I_4	I_5	X	X	I_6	I_7		
VVD		I_1	I_2	I_3	I_4	I_5	X	X	I_6	I_7	
PUV			I_1	I_2	I_3	I_4	I_5	X	X	I_6	I_7

Tabulka 6: Výpadek fronty po skokové instrukci

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
VID	I_1	I_2	I_5	I_3	I_4	I_6	I_7		
VVD		I_1	I_2	I_5	I_3	I_4	I_6	I_7	
PUV			I_1	I_2	I_5	I_3	I_4	I_6	I_7

Tabulka 7: Využití zpožděného skoku - změna pořadí provedení instrukcí

1.5.2 Zpoždění skokové instrukce

Představme si nejprve zjednodušenou zřetězenou jednotku pouze se třemi úrovněmi zřetězení. V první fázi se provede výběr instrukce a dekódování (VID), následuje výpočet adres a výběr dat (VVD) a poslední třetí krok je provedení instrukce a uložení výsledku (PUV).

Podívejme se teď na tabulku 6. Pokud I_5 bude skoková instrukce a skok se provede, budou rozpracované instrukce označené jako X ignorovány.

Co by se však stalo, kdybychom činnost procesoru po skokové instrukci nezastavili? Rozpracované instrukce by se provedly a pokračovalo by se instrukcí I_6. Takové řešení by zjednodušilo logiku vnitřního řízení procesoru, ale po skokové instrukci by se provedlo ještě několik rozpracovaných instrukcí, což by mohlo narušit logiku programu.

Můžeme tedy místo za podmíněným skokem vyplnit prázdnými instrukcemi NOP. Tím se sice logika programu nezmění, ale vykonáváním prázdných instrukcí se efektivita činnosti procesoru nezvýší.

Pokud by se ale podařilo před skokovou instrukcí najít několik instrukcí nesouvisejících přímo s podmíněným skokem a vyhodnocením jeho podmínky, mohli bychom je zařadit na prázdné místo označené v tabulce 6 jako X.

Jak bude vypadat provádění upraveného kódu se můžeme podívat v tabulce 7. V optimalizujícím kompilátoru je poměrně snadné realizovat požadované přeskupení instrukcí a využít tak zpoždění skoku k vyšší efektivitě práce zřetězené jednotky.

Pro názornost si můžeme představit celé řešení na jednoduchém příkladu v programovacím jazyce C a jeho následném přepise do assembleru:

```
/* Příkaz v jazyce C */
```

```
  if ( i++ == j++ ) { /* blok příkazů ... */ }
  k++;

  ;Přepis do assembleru
  CMP i, j
  JNE pokracuj
  INC i
  INC j
  ; blok příkazů ...
pokracuj:
  INC k
```

Z příkladu je vidět, že zvýšení hodnoty proměnných i a j musí být provedeno ještě před vykonáním „bloku příkazů“. Přesto pokud víme, že skoková instrukce bude zpožděna, můžeme instrukce pro inkrementaci proměnných i a j nechat až za skokovou instrukcí. Pro analogii s tabulkou 7 si představme, že instrukce I_5 je podmíněný skok `JNE pokracuj` a inkrementace proměnných budou instrukce I_3 a I_4.

1.5.3 Použití paměti skoků

Velmi rozšířeným řešením se stala tabulka provedených podmíněných skoků, realizovaná přímo jako součást procesoru. Velikost tabulky je předem pevně stanovena a do tabulky se ukládají adresy posledních provedených skoků. K údajům v tabulce se může aplikovat jednobitová, nebo dvoubitová predikce skoků.

Tato metoda nevyžaduje úpravu formátu strojových instrukcí, není spojena s činností překladače a je možno ji použít i v nových generacích procesorů, kde je vyžadována zpětná kompatibilita strojového kódu.

2 Kontrolní otázky

1. Kdy a proč se začaly procesory dělit na RISC a CISC?
2. Jaké byly zásadní důvody, proč se začaly procesory RISC vyvíjet?
3. Jaké jsou základní konstrukční vlastnosti procesorů RISC?
4. Jak přispěly jednotlivé charakteristické vlastnosti procesorů RISC ke zvýšení výpočetního výkonu?
5. Jaký je princip zřetězeného zpracování instrukcí v RISC procesorech?
6. Jakého zrychlení lze zřetězeným zpracováním instrukcí dosáhnout?

7. Jaké problémy přináší zřetězené zpracování instrukcí v procesorech RISC?
8. Co to je predikce skoků, proč se používá a jaké způsoby predikce se využívají?
9. Co to jsou datové a strukturální hazardy v RISC procesorech? Co je způsobuje?
10. Jak funguje dvoubitová dynamická predikce skoků a proč se využívá?