evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Programování s ohledem na hardware

Martin Mareš[1]

Doprovodný text k sérii přednášek o architektuře počítačů a programování s ohledem na hardware, kterou jsem v únoru 2012 přednesl na Fakultě přírodovědně-humanitní a pedagogické Technické univerzity v Liberci.

Úvodem

Architektura počítačů se během posledních dvou desetiletí výrazně změnila. Od jednoduchých strojů vykonávajících instrukce programu jednu po druhé jsme dospěli k vnitřně paralelním procesorům vybaveným mnoha implementačními triky, které zrychlují provádění „průměrných“ programů.

Cílem tohoto textu je přiblížit čtenáři, jak dnešní počítače pracují, jaký to má vliv na rychlost různých programových konstrukcí a jak toho využít k optimalizaci programů. Nejde nám přitom o přesný popis hardwaru (ten v mnoha případech výrobci tají), nýbrž o to, jak se počítač chová k programu.

Vzhledem k šíři tématu se omezíme na současné počítače třídy PC (32-bitovou architekturu i386 a 64-bitovou AMD64), operační systém Linux a programovací jazyk C. Ostatní dnes vyráběné počítače, systémy i kompilované jazyky se nicméně chovají obdobně.

Architektura i386

Architektura dnešních PC prošla spletitou evolucí od 16-bitového procesoru Intel 8086 (1978) až k současným 64-bitovým procesorům. Během tohoto vývoje se vždy dodržovala zpětná kompatibilita strojového kódu, takže vzniká džungle plná slepých uliček a přepínatelných módů.

Stručně popíšeme, jak vypadá 32-bitový uživatelský mód s plochou adresací. To odpovídá běžnému prostředí uživatelských programů pod Linuxem. Omezíme se na vlastnosti nutné k pochopení přeložených programů.

Datové typy

- Základní adresovatelnou jednotkou je 1 *byte* (8 bitů).
- Celá čísla: byte, *word* (16 bitů), *long* neboli *doubleword* (32); záporná čísla reprezentujeme dvojkovým doplňkem. V paměti uložena v *little endian* pořadí – od nejnižšího bytu k nejvyššímu.

[1] Katedra Aplikované Matematiky MFF UK Praha, *mj@ucw.cz, http://mj.ucw.cz/*

- Floating-point (FP) čísla: *half* (16 bitů/z toho 11 mantisa), *single* (32/24), *double* (64/53), *extended* (80/64). Vše podle IEEE 754. Též nekonečna, znaménkové nuly, `NaN`y, nenormalizovaná čísla.
- 128-bitové vektory (novější i 256-bitové) – např. $4\times$single float. Složky mohou být libovolného typu mimo extended floatů.

Paměť

- Adresní prostor velikosti 4 GiB (32-bitové adresy).
- Společně kód i data.
- Obsazení a přístupová práva určuje OS.

Registry

- `EAX`, `EBX`, `ECX` a `EDX` – 32-bitové univerzální. Lze je dělit: `AX` (dolních 16 bitů), `AL` (dolních 8), `AH` (8.–15. bit).
- `ESI` (source index), `EDI` (destination index), `EBP` (base pointer), `ESP` (stack pointer) – další 32-bitové.
- `EFLAGS` – stavový registr obsahující 32 příznaků (flags). Například: `Z` (zero), `C` (carry), `S` (sign), `O` (overflow).
- `EIP` – ukazatel na právě prováděnou instrukci (instruction pointer).
- `ST0`–`ST7` – 80-bitové FP registry, používané jako zásobník.
- `FPUCR` – řídící registr pro FP: přesnost (24/53/64), mód zaokrouhlování (nejbližší/0/$+\infty$/$-\infty$), generování výjimek.
- `FPUSR` – stavový registr FP: příznaky pro přetečení, podtečení, výsledek porovnání, chybný argument apod.
- `XMM0`–`XMM7` – 128-bitové vektorové registry.
- `YMM0`–`YMM7` – jejich 256-bitové rozšíření (poprvé 2011).
- `MXCSR` – řídící a stavový registr vektorové jednotky.

Instrukce

V paměti kódovány posloupností bytů *(strojový kód),* obvykle se zapisují v *assembleru.* Existují dvě různé syntaxe: Intel (běžná pod Windows) a AT&T (zbytek světa), pozor, liší se pořadím operandů.

Typická instrukce v AT&T syntaxi vypadá takto:

```
addl $1,%ebx
```

Čteme v pořadí typ operace (`add` – sečtení celých čísel), velikost operandů (`l` – long), zdroj, cíl (zde slouží i jako druhý zdroj).

Možné typy operandů:

- `$`*číslo* – literál (konstanta, jež je součástí kódu instrukce)
- `%`*registr* – obsah registru
- *adresa* – hodnota uložená v paměti, *adresa* je literál
- `(%`*registr*`)` – adresace paměti registrem

- *offset*`(%`*registr*`)` – k adrese přičteme literál (adresace struktur)
- *offset*`(%`*reg1*`,%`*reg2*`,`*násobitel*`)` – k adrese v *reg1* navíc přičteme *reg2* násobený konstantou 1, 2, 4 nebo 8. Slouží k indexaci polí; offset či násobitele lze vynechat.
- implicitní operand – nepíše se, často je to třeba `ESP`, `EIP` nebo `EAX`.

Mimo „počítacích" instrukcí existují ještě řídící, které mění chod programu: skoky (na danou adresu), podmíněné skoky (skočí při určitě kombinaci příznaků v `EFLAGS`), volání podprogramu a návrat z něj.

Volací konvence a ABI

Pro každý kompilovaný programovací jazyk existuje souhrn zvyklostí *(ABI – Application Binary Interface),* které určují, jak se tento jazyk překládá do strojových instrukcí. Patří sem zejména *volací konvence,* což je způsob předávání parametrů a výsledků mezi podprogramy. ABI je závislé na architektuře počítače, na OS a někdy i na překladači. Podívejme se, jak je tomu v jazyce C pod Linuxem na i386 (trochu zjednodušeně).

K předávání argumentů a návratové adresy a k uchovávání lokálních proměnných se využívá zásobník. Každá funkce na něm vytvoří svůj *stack frame,* který vypadá následovně:

	...	↑ *vyšší adresy*
`EBP` + 12	argument 2	
`EBP` + 8	argument 1	*(odstraňuje volající)*
`EBP` + 4	návratová adresa	*(odstraňuje volaný)*
`EBP`	předchozí hodnota `EBP`	
`EBP` − ...	lokální proměnné	
`ESP`	pracovní prostor	↓ *nižší adresy*

Pomocí `EBP` adresujeme argumenty (kladnými offsety) a lokální proměnné (zápornými), pomocí `ESP` různé mezivýsledky (mezi něž patří i argumenty vnořených funkcí).

Argumenty se předávají v 32-bitových slotech (80-bitový double tedy zabere 3 sloty). Pokud je výsledek funkce skalár, vrací se v `EAX` (je-li celočíselný či ukazatel), v `ST0` (je-li to float) nebo v dvojici `EDX:EAX` (64-bitové celé číslo). Pokud vracíme strukturu, rozhoduje její velikost: malá se vrací také v `EAX`, pokud je větší, volající na ni vyhradí místo a jako nultý argument předá adresu tohoto prostoru; volaný tam strukturu zkopíruje a v `EAX` vrátí její adresu.

Volací konvence také specifikuje, které registry je volaná funkce povinna zachovat a které může libovolně měnit. V našem případě lze měnit `EAX`, `ECX`, `EDX`, `ST0`–`ST7` a některé příznaky v `EFLAGS`.

Pokud je pracovní prostor funkce konstantně velký, lze se na celý stack frame odkazovat relativně vůči `ESP`. Registr `EBP` je pak volný pro jiné účely.

Architektura AMD64

Firma AMD v roce 2000 s rozšířením architektury i386 o 64-bitový mód. Mezi datové typy přibyla 64-bitová celá čísla, která slouží i jako adresy v paměti. Registry byly rozšířeny následovně:

- `RAX`–`RDX`, `RSI`, `RDI`, `RBP`, `RSP` (souhrnně `R0`–`R7`), `RIP` – 64-bitové.
- Dalších 8 univerzálních registrů `R8`–`R15`. Dělitelné: `R8D`, `R8W`, `R8B`.
- Dalších 8 vektorových registrů `XMM8`–`XMM15`, stále 128-bitové.

Navíc k adresním módům přibyla adresace relativní k `RIP`. Naopak byla odbourána velká část historického balastu.

Volací konvence a ABI

Volací konvence je o něco složitější. Vzhledem k velkému počtu registrů předáváme část argumentů v registrech a teprve ty, co se nevejdou nebo jsou příliš složité, ukládáme na zásobník po 8-bytových slotech. Opět pravidla ABI poněkud zjednodušujeme.

Registry jsou přiděleny následovně:

`RAX`	var/res1	`R8`	arg5	`XMM0`	farg1/fres1	`XMM8-15`	scratch
`RBX`	save	`R9`	arg6	`XMM1`	farg2/fres2	`EFLAGS`	část scratch,
`RCX`	arg4	`R10`	scratch	`XMM2`	farg3		část save
`RDX`	arg3/res2	`R11`	scratch	`XMM3`	farg4	`ST0`	xres1
`RSI`	arg2	`R12`	save	`XMM4`	farg5	`ST1`	xres2
`RDI`	arg1	`R13`	save	`XMM5`	farg6	`ST2-7`	scratch
`RBP`	save	`R14`	save	`XMM6`	farg7		
`RSP`	save	`R15`	save	`XMM7`	farg8		

V této tabulce arg*i* značí *i*-tý celočíselný argument v pořadí, res*i* je *i*-tá část celočíselného výsledku, farg*i* a fres*i* jsou jejich floatové obdoby a xres*i* extended float výsledek. Registr označený jako *scratch* smí být funkcí přepsán; *save* píšeme, musí-li být jeho hodnota zachována; *var* viz níže.

Struktury se předávají v registrech, jsou-li dost malé a dost jednoduché, jinak na zásobníku. Návratová hodnota se předává v registrech, velké či složité struktury v paměti stejně jako na i386.

Pokud funkce má proměnný počet argumentů, předáváme navíc v `AL` počet `XMM` registrů použitých pro předávání argumentů.

Stack frame vypadá podobně jako na i386, jen OS navíc zaručuje dostupnost tzv. *červené zóny* v podobě 128 bytů pod `ESP`. Tu mohou funkce využívat jako pracovní prostor, aniž by musely pohybovat stack pointerem.

Hardware procesoru

Dřívější procesory vykonávaly program instrukci po instrukci, každá z nich zabrala jeden nebo více taktů hodinového signálu. Dnes se využívá mnoha různých implementačních triků, které mají výpočet zrychlit:

- *Pipelining* neboli *zřetězené zpracování* – instrukce prochází několika fázemi vykonávání, jakmile vstoupí do další fáze, může do té současné nastoupit další instrukce. Řetězec můžeme popsat *latencí* (za jak dlouho dostaneme výsledek) a *propustností* (za jak dlouho můžeme zadat další instrukci). Pokud instrukce nejsou nezávislé, musíme čekat na dokončení a vznikají *bubliny.*

- *Superskalární zpracování* – pořídíme si více jednotek pracujících současně (často jsou specializované, např. některé na celočíselné operace, jiné na floaty, další na práci s pamětí). Můžeme vykonat více instrukcí za jeden takt, pokud jsou na sobě nezávislé.
- *Rozklad na mikrooperace* – složité instrukce (a těch není málo) rozložíme na několik jednodušších (μop), které přiřazujeme jednotkám samostatně. Tím se i386 přiblíží symetrickým architekturám RISC, ale zůstane zachována kompaktnost strojového kódu; platíme za to složitými dekodéry v procesoru.
- *Přejmenovávání registrů* – v posloupnosti instrukcí typu

```
movl $1, %eax
movl %eax, (%edi)
movl $2, %eax
movl %eax, 4(%edi)
```

  vznikají falešné závislosti mezi první a druhou dvojicí instrukcí. Odstraníme ji tak, že do procesoru zabudujeme větší množství registrů, kterým budeme jména určená architekturou přiřazovat dynamicky. První dvojice instrukcí tedy bude pracovat s jiným registrem než druhá dvojice a závislost nevznikne.
- *Spojování mikrooperací* – někdy se naopak hodí dvojici μop, které vznikly z různých instrukcí, spojit do jedné, kterou zpracujeme dohromady. Takto se kombinuje třeba instrukce pro porovnání s instrukcí podmíněného skoku.
- *Cache* – jelikož přístup k paměti je o několik řádů pomalejší než přístup do registrů, vybavují se procesory vyrovnávací pamětí neboli cache. V ní se uchovávají často používané hodnoty z hlavní paměti. Cache je součástí procesoru a je řádově rychlejší a řádově menší než hlavní paměť.
- *Spekulativní vyhodnocování* – pokud procesor při provádění instrukcí narazí na podmíněný skok, musí počkat, až bude podmínka vyhodnocena. Tím se celý řetězec jednotek vyprázdní, a teprve pak se začne plnit dalšími instrukcemi, čímž vznikne obrovská bublina. Abychom tomu zamezili, vsadíme si na jednu z větví výpočtu a předpokládáme, že je to ta, která se provede. Se zápisy do paměti nebo do registrů architektury přitom čekáme, až bude jasné, jak podmínka dopadla, a pak buďto zápisy provedeme nebo celý spekulativní výpočet zahodíme. K tomu potřebujeme mechanismus pro *predikci skoků.*

Při programování si musíme dávat pozor na:

- Pomalé instrukce – některé instrukce se dnes udržují jen pro zpětnou kompatibilitu a procesor na ně nebývá optimalizován.

- Kombinace instrukcí – vzhledem k asymetrii jednotek se často vyplácí některou operaci vyjádřit jinou ekvivalentní instrukcí nebo dokonce více instrukcemi. Velmi časté je to u násobení a dělení.
- Závislosti mezi instrukcemi – je lepší prokládat více nezávislých částí výpočtu, aby byly jednotky lépe vytíženy.
- Predikovatelnost skoků – raději se vyhnout skokům, které se chovají „chaoticky“, i za cenu většího počtu aritmetických operací.

Většinu z toho za nás zařídí překladač lépe, než bychom zvládli ručně, ale zejména u závislostí a skoků je potřeba mu pomoci.

Cache

Pro účely cacheování rozdělíme paměť na *řádky,* obvykle velké 32 nebo 64 bytů. Cache si pak pamatuje nějakou množinu dvojic (číslo řádku, obsah řádku). Každý řádek přitom buďto je v cachi uložen celý, nebo není uložen ani zčásti.

Pokud by program z paměti jenom četl, je ovládání cache jednoduché. Jakmile chce program číst z nějaké adresy, zjistíme, ve kterém leží řádku, a vyhledáme tento řádek v cachi. Pokud se v ní vyskytuje, vrátíme obsah z cache. Pokud nikoliv, přečteme ho z hlavní paměti a na nějaké místo v cachi ho uložíme. Dochází-li volné místo, uvolňujeme přednostně ty řádky, ke kterým se už dlouho nepřistupovalo; obvykle se k tomu používá strategie LRU *(least-recently used)* nebo její aproximace.

Zápisy situaci trochu komplikují. Obvykle jsme ochotni zapisovat pouze do řádků, které už v cachi leží; pokud by program modifikoval nějaká necacheovaná data, nejprve je do cache načteme. Pak zapamatovaný řádek upravíme a buďto ho do hlavní paměti rovnou zapíšeme (tomu se říká *write-through cache*), nebo si poznamenáme, že obsahuje změněná data, a fyzicky ho zapíšeme při první vhodné příležitosti (*write-back cache*), nejpozději v okamžiku, kdy potřebujeme místo v cachi využít pro jiný řádek. Druhý způsob je složitější na implementaci, ale výrazně efektivnější – dlouhý, ale paměťově lokální výpočet si vystačí s jedním přečtením dat z hlavní paměti a jedním zápisem zpět.

Vidíme, že na rozdíl od hlavní paměti nehledáme v cachi podle pozice, nýbrž podle části obsahu. Takovým pamětem se říká *asociativní* a obvykle se hardwarově realizují jedním z těchto způsobů (demonstrujeme na 32-bitových adresách a 32 KiB cachi složené z 64-bytových řádků):

- *Plně asociativní* – při dotazu na řádek hardware prohledá celou cache. To je účinné, leč pro netriviální velikosti cache příliš pomalé.
- *Přímo mapované* – zvolíme si nějakou hashovací funkci, která každé adrese přiřadí jednu z pozic v cachi, na které jediné se daný řádek může vyskytovat. To je velice jednoduché na implementaci, ale efektivitu nám kazí *kolize* hashovací funkce.

  Pro náš modelový příklad by to mohlo vypadat takto: cache se skládá z 512 pozic, nejnižších 6 bitů adresy udává *polohu* v řádku, dalších 9 udává *index* pozice a zbylých 17 slouží jako *tag,* pomocí nějž ověříme, zda na dané pozici skutečně leží hledaný řádek (ostatní bity adresy nemusíme porovnávat).

- *Množinově asociativní* – to je kompromis mezi oběma předchozími metodami: index určí nikoliv konkrétní pozici, ale množinu několika pozic, mezi nimiž je cache už plně asociativní.

  Pro naši ukázkovou cache zvolíme množiny velikosti 4 (tomu se říká *4-cestná asociativita*), bude jich tedy 128. Tím pádem dolních 6 bitů adresy určí polohu v řádku, dalších 7 určí index množiny a zbylých 19 tag.

V dnešních procesorech se obvykle používají množinově asociativní cache a najdeme jich tam hned několik zřetězených za sebe. Typická *hierarchie cachí* vypadá takto:

- `L1` *(level 1)* – velikost řádově desítky KiB, obvykle rozdělena na instrukční `L1I` a datovou `L1D`. Instrukční cache slouží pouze pro čtení, datová i pro zápisy (používá write-back). K `L1D` je možné během jednoho taktu přistupovat i vícekrát současně.
- `L2` – řádově stovky KiB, společná pro kód a data, pomalejší než `L1`. Obvykle se do ní ukládají data vyřazená z `L1`.
- `L3` – řádově jednotky MiB, společná pro kód a data, opět trochu pomalejší.

Dodejme ještě, že latence instrukcí pro čtení z paměti se obvykle snižuje speciální jednotkou pro *prefetch*. Ta má na starost předpovídat, jaká data bude program v blízké budoucnosti potřebovat, a v předstihnu zahájit jejich načítání do některé z cachí. Prefetch obvykle funguje velice dobře pro sekvenční přístup (popředu, někdy i pozpátku), v ostatních případech si jej lze vyžádat ručně speciálními instrukcemi.

Celý mechanismus cacheování je řízen hardwarem, ani programy, ani operační systém o něm nijak nerozhodují. Tím spíš je potřeba si rozmyslet, jak bude náš program s cachemi interagovat:

- K datům je vhodné přistupovat co nejlokálněji. Například chceme-li uložit pole komplexních čísel, je lepší použít pole struktur než dvě samostatná pole. Procházet velkou matici po sloupcích je daleko pomalejší než po řádcích.
- Data se hodí zarovnávat, aby pokud možno nepřecházela přes hranici řádků cache.
- Je důležité šetřit pamětí – složitější program, kterému stačí méně pracovní paměti, může být v praxi efektivnější, jelikož se všechny pracovní proměnné vejdou do cache. Z téhož důvodu také někdy uškodí rozbalování smyček.
- Samostatné zápisy jsou dražší než samostatná čtení.
- Jelikož cache nebývá plně asociativní, mohou vznikat nečekané kolize *(cache aliasing)*. Například v 4-cestné cachi z našeho příkladu skončí v téže množině libovolné dvě adresy, které se liší o násobek 8 KiB. Pokud tedy máme matici, jejíž rozměry jsou dělitelné vysokými mocninami dvojky, snadno se stane, že se prvky ležící pod

sebou navzájem vyhazují z cache. Bývá lepší matici trochu zvětšit, byť to zkomplikuje výpočet adres.

- Je třeba dávat pozor na případy, kdy zpracováváme velká necacheovatelná data, a přitom si potřebujeme udržovat malé množství často používaných mezivýsledků. Tehdy se běžně děje, že se při přístupu k velkým datům vyhazují z cache ta malá. Tomu lze předejít použitím speciálních instrukcí pro *proudový (streaming) přístup* do paměti, které hierarchii cachí obcházejí.

Překlad adres a TLB

Krátce zmiňme mechanismus, který operační systémy používají pro překlad *virtuálních adres* v adresním prostoru jednotlivých procesů na *fyzické adresy* skutečné operační paměti. Oba adresní prostory jsou rozděleny na stejně velké *stránky* (typicky 4 KiB), jednotka pro správu paměti *(Memory Management Unit)* pracuje na úrovni jednotlivých stránek a zajišťuje překlad čísla virtuální stránky na číslo fyzické stránky a přístupová práva (operační systém si například může objednat, že stránku bude možné jenom číst).

Detaily překladového mechanismu ponechme stranou, důležité je, že je to poměrně pomalý proces, který vyžaduje nahlédnutí do speciálních datových struktur (stromově uspořádaných tabulek stránek). Výsledek překladu se proto také cacheuje ve speciální paměti nazývané TLB (Translation Look-aside Buffer). O implementaci TLB toho není mnoho známo, můžeme předpokládat, že obsahuje řádově stovky položek a že je podobně jako cache rozdělena na `L1I`, `L1D` a `L2`.

Přistupujeme-li tedy do paměti příliš na přeskáčku, bude nás zdržovat nejen nepřítomnost řádku v cachi, ale také nepřítomnost překladu adresy stránky v TLB. Lokalita přístupů je tedy o to důležitější.

Náhodné přístupy k velkým blokům paměti lze zrychlit použitím speciálních velkých stránek (na i386 jsou velké 4 MiB, na AMD64 2 MiB). Ty je ovšem na Linuxu zatím potřeba alokovat speciálním způsobem (pomocí `hugetlbfs`).

Predikce skoků

Podívejme se nyní na implementaci prediktoru skoků. To je jednotka, která má za úkol sledovat skoky v programu a podle minulého chování předpovídat, jak se skok zachová příště – tedy zda skočí a pokud ano, tak kam. Tabulce, v níž si procesor chování skoků pamatuje, se říká BTB *(Branch Target Buffer)*. Běžně se používají tyto metody predikce:

- *Statická predikce* – pokud o předchozím chování skoku nic nevíme (například není vůbec přítomen v BTB), můžeme použít nějakou primitivní heuristiku typu „skok dopředu obvykle neskočí, skok dozadu obvykle skočí“. To vychází z běžné struktury podmínek a smyček v přeložených programech.
- *Saturující čítač* – v každé položce BTB máme uložen čítač, který může nabývat hodnot od $-k$ do k (typicky $k = 2$) bez nuly. U nové položky ho inicializujeme podle statické predikce. Pokud se skok provede, čítač zvýšíme; pokud už byl na maximu, zůstane tam.

Pokud neskočíme, čítač naopak snižujeme. Znaménko čítače určuje předpověď.

- *Pole čítačů* – pro každý skok si pamatujeme jeho historii (h-bitové číslo, které popisuje, jak se skok choval při posledních h návštěvách) a pro každou z možných historií si udržujeme jeden saturující čítač. Tím zvládneme předpovídat skoky s krátkoperiodickým chováním, ale potřebujeme 2^h čítačů, takže si můžeme dovolit jen velmi malé h.
- *Globální historie* – místo historie každého skoku si pamatujeme společnou historii všech skoků a udržujeme společné pole čítačů (hashovací tabulku indexovanou historií). Tím pádem si můžeme dovolit delší historii, ale mohou nám ublížit falešné závislosti a také kolize hashovací funkce.
- *Agree predictor* – zkombinujeme předchozí přístupy. Pro každý skok si budeme udržovat jeden čítač a k tomu si ještě pořídíme globální tabulku (indexovanou globální historií), ale nebudeme do ní ukládat výsledky predikce, nýbrž zda se základní čítač strefil.
- *Detektor smyček* – přídavný prediktor, který se snaží odhalit smyčky s pevným počtem opakování, a příště předpovědět stejný počet iterací. Někdy si procesor umí u krátké smyčky zapamatovat i to, ze kterých μop se skládá, takže není potřeba instrukce při každém průchodu znovu dekódovat.
- *Nepřímé skoky* – občas se vyskytují skoky, jejichž cílová adresa není konstantní (např. příkaz `switch`, ukazatele na funkce, tabulky virtuálních metod v C++). Některé procesory obsahují speciální jednotku, která předpovídá cíl takových nepřímých skoků.
- *Zásobníkový prediktor* – speciálním případem nepřímého skoku je návrat z podprogramu (instrukce `ret`), který skáče na adresu vybranou ze zásobníku. Často se proto implementuje speciální jednotka, která si pamatuje nedávnou historii instrukcí `call` a `ret` a předpovídá, kam skočí příští `ret`.
- *Hybridní prediktory* – kombinují více různých metod a pamatují si u každého skoku, která metoda dává nejlepší výsledek (například statická predikce, dynamická predikce, zásobník a detektor cyklů).

Konkrétní implementace: Intel Core2

Podívejme se alespoň na jeden konkrétní procesor, totiž Intel Core2. Výrobce bohužel zveřejňuje jen základní informace, mnohé detaily byly odvozeny dodatečně autory překladačů měřením chování procesoru. Nemusejí proto přesně odpovídat reálnému hardwaru, ale pro náš programátorský pohled jsou snad dostatečně přesné.

Zpracovaní instrukcí probíhá v následujících krocích:

- *Prediction* – nejprve pomocí hybridního prediktoru skoků předpovíme, kde leží následující instrukce. (Mispredikce skoku stojí v nejlepším případě 15 taktů.)

- *Fetch* – nyní vyzvedneme z `L1I` cache blok obsahující instrukce (16 bytů za takt).
- *Predecode* – určíme hranice instrukcí. Procesor si umí zapamatovat až 64 předdekódovaných instrukcí a vrátit se k nim ve smyčce.
- *Decode* – dekódujeme instrukce na μop.
- *Rename* – přejmenování registrů architektury na hardwarové registry (odhad: 64 celočíselných registrů). Speciální logika detekuje idiomy typu `xorl %eax,%eax` a nevytváří u nich závislosti.
- *Reorder* – μop se ukládají do ROB *(Re-Order Buffer)*. Ten obsahuje 40 položek, každá z nich má až 2 zdrojové a 2 cílové operandy (to mohou být i části `EFLAGS`). Vstupy μop mohou být konstanty, hodnoty získané z registrů nebo informace o tom, který výstup jiné μop se stane vstupem této, až bude spočítán.
- *Reservation station* – jakmile μop v ROB dostane všechny vstupy, postoupí do další fáze, která μop předává výpočetním jednotkám. Pamatuje si až 20 μop, je vybavena 6 porty, na každém z nich vyčkává několik jednotek. V každém taktu může na každý port poslat jednu μop.

  Celočíselné jednotky jsou připojeny na porty 0, 1 a 5; floatové jednotky na porty 0 a 1; jednotky pro práci z pamětí na 2, 3 a 4; skoky jdou přes port 5. Obvyklá latence je 1 takt, celočíselné násobení má 3 takty, floatové sčítání také 3, floatové násobení 4–5, dělení je ještě pomalejší. Operace s pamětí mají obvykle latenci 2–3 takty. Propustnost všech jednotek činí 1 μop/takt.
- *Retirement* – μop se do ROB ukládají v pořadí původních instrukcí, ale vyhodnocují se na přeskáčku. V této poslední fázi se hotové μop odebírají opět v původním pořadí a potvrzují se jejich výsledky (zapisují do registrů apod.).

Dekodér je schopen přeložit až 3 instrukce za takt. Fáze Rename, Reorder a Retirement jsou dimenzovány na zpracování 3 μop za takt, Reservation jich dokáže odeslat i více, má-li volné porty. Procesory novější generace (Nehalem) zvyšují tyto počty na 4 za takt.

Ještě novější generace (Sandy Bridge) je vybavena speciální cachí, ve které se uchovává až 1 536 μop vystupujících z dekodéru. Tato cache nahrazuje speciální paměť těl smyček a kompenzuje relativní pomalost dekodérů.

Mimo to se již v Core2 uplatňuje *Stack Engine,* který během dekódování každé instrukce stanoví hodnotu `ESP` pro tuto instrukci. Tím obchází závislosti na předchozí hodnotě `ESP` u instrukcí pro práci se zásobníkem (`push`, `pop`, `call`, `ret`). Pokud ale `ESP` využijeme k nepřímé adresaci nebo v aritmetických instrukcích, je potřeba SE synchronizovat s výkonnými jednotkami vložením speciální μop.

Core2 je vybaven následujícími cachemi, všechny mají 64-bytové řádky:

- `L1I` – 32 KiB, 8-cestná, latence 3 takty.

- `L1D` – 32 KiB, 8-cestná, latence 3 takty.
- `L2` – 2 MiB 16-cestná až 6 MiB 24-cestná (podle modelu), 15 taktů.

TLB je rozdělen na tři části: `L1I` se 128 položkami, `L1D` s 256, `L2` s 512.

Všechny cache pracují s fyzickými adresami (po překladu pomocí MMU), aby se zamezilo problémům s více virtuálními stránkami namapovanými na jednu fyzickou. Obě `L1` cache mají přitom parametry zvoleny tak, aby bylo možné index vypočíst i bez znalosti překladu a vyhledávat pak současně v cachi i v TLB.

Paralelní výpočty

Programy můžeme dále zrychlovat tím, že je budeme paralelizovat. Toho můžeme dosáhnout na několika různých úrovních:

- *Bitový paralelismus* – využijeme toho, že aritmetické a logické operace pracují paralelně na jednotlivých bitech čísla. Například můžeme malé množiny kódovat pomocí čísel a množinové operace převést na bitové. Jiný příklad: `x&(x-1) == 0` právě tehdy, když `x` je mocnina dvojky nebo 0.
- *Vektorové instrukce* – architektury i386 i AMD64 jsou vybaveny speciálními instrukcemi, které dovedou vykonat jednu operaci současně na všech složkách 128-bitového vektoru. Tedy například provést najednou 4 násobení 32-bitových floatů. Tomu se říká paralelismus typu *SIMD* – Single Instruction Multiple Data. Též lze kombinovat s bitovým paralelismem uvnitř složek vektorů.
- *Víceprocesorové počítače* – postavíme počítač s více procesory, které budou sdílet paměť. Na každém procesoru poběží jiná část programu, komunikovat spolu budou prostřednictvím sdílené paměti. Tím můžeme výpočet výrazně zrychlit, pokud nás nebude příliš brzdit komunikace. Také je nutno řešit synchronizaci datových struktur.
- *Clustery* – využijeme více počítačů propojených rychlou sítí. Jednotlivé části výpočtu komunikují pomocí zpráv. Výhodou je takřka neomezený výpočetní výkon, nevýhodou nutnost starat se o komunikaci explicitně, často s ohledem na propustnost a zpoždění sítě.
- *Grafické procesory* – herní průmysl přiměl výrobce grafických procesorů (GPU) stále zvyšovat výkon, takže GPU jsou dnes pro některé typy výpočtů (třeba některé vektorové) výrazně rychlejší než hlavní procesor. Dnes už se vyrábějí i specializované karty s více GPU pro vědecké výpočty.

Až na triviální případy zatím neumíme paralelizovat automaticky, u některých úloh ani není jasné, zda efektivní paralelní algoritmus existuje (viz slavný problém $L = P$ v teorii složitosti).

Vektorové instrukce

Na i386 i AMD64 jsou k dispozici 128-bitové registry, do kterých můžeme ukládat vektory v různých formátech. Složky mohou být celá čísla o 8, 16, 32 nebo 64

bitech, případně 32b nebo 64b floaty. Počet složek je vždy takový, aby celková velikost vektoru vyšla 128 bitů. V poslední generaci procesorů přibylo rozšíření AVX (Advanced Vector Extensions), s nímž je délka vektorů dvojnásobná.

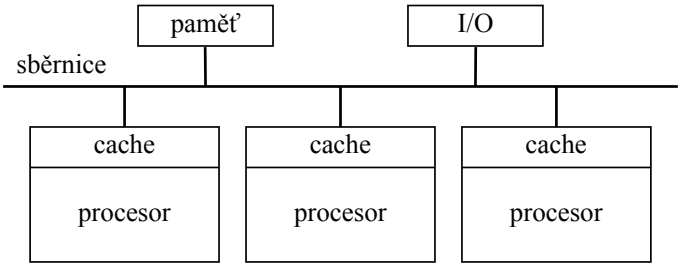
Vektorové instrukce tvoří poměrně složitou džungli, instrukční sada se rozšiřovala postupně a některé operace jsou evidentně motivované konkrétními aplikacemi, často zpracováním obrazu a videa. K dispozici jsou například:

- *Celočíselné operace:* sčítání, odčítání, násobení, minimum, maximum, průměr, bitové operace. Pozor, dělení a modulo chybí. Aritmetika může fungovat jak klasicky (modulo 2^w), tak saturačně (při přetečení dosadíme min/max reprezentovatelné číslo). Porovnávání vydá bitovou masku, tu je možné použít k podmíněnému přiřazení.
- *Fixed-point operace:* extrakce bitových polí, násobení kombinované s extrakcí.
- *Floatové operace:* konverze (int $\leftrightarrow$ float, single $\leftrightarrow$ double), sčítání, odčítání, násobení, dělení, minimum, maximum, aproximace $\sqrt{x}$ a $1/\sqrt{x}$. I na floatech fungují bitové operace (práce se znaménky apod.), porovnávání opět používá masky.
- *Horizontální instrukce:* Některé aritmetické operace existují i v horizontální podobě, která pracuje s různými složkami téhož vektoru.
- *Permutace složek:* pro 32-bitové složky můžeme předepsat libovolné zobrazení zdrojových pozic na cílové (permutace, rozkopírování, ...), pro 16-bitové složky si musíme vybrat horní/dolní polovinu registru a v ní permutovat.
- *Přístup do paměti:* čtení/zápis celých vektorů, speciální instrukce pro nezarovnaný přístup (pomalejší), streaming (obchází cache), manuální prefetch.
- *Různé:* skalární součin, čtení/zápis jedné složky, řetězcové operace.

Mimo přirozeně vektorových výpočtů lze vektorizovat i operace s bloky paměti, případně provádět tentýž skalární výpočet několikrát paralelně s různými vstupy (to se typicky hodi v kryptografii).

Symetrický multiprocesing (SMP)

Symetrický multiprocesorový počítač se skládá z několika procesorů připojených ke společné paměti:



Problémy přitom působí existence cache – je potřeba zajistit její *koherenci* (cache se chová sémanticky transparantně, stejně jako ve stroji s jediným procesorem). Obvykle se používá protokol MESI, který každému řádku každé cache přiřadí jeden z následujících stavů:

- *Exclusive* – řádek se nachází pouze v této cachi a je aktuální (odpovídá stavu v hlavní paměti).
- *Shared* – řádek se může nacházet i v jiné cachi a je aktuální.
- *Modified* – řádek pouze v této cachi a obsahuje novější data, než jsou v hlavní paměti.
- *Invalid* – řádek se v této cachi nenachází.

Pokud tedy více procesorů nějaká sdílená data pouze čte, může je každý z nich mít nacacheovaná a všechny cache jsou ve stavu Shared. Chce-li některý procesor do sdílených dat zapisovat, nejprve požádá ostatní procesory, aby mu předaly vlastnictví řádku (tím se stane Exclusive), a teprve pak data upraví a přejde do stavu Modified. Možné scénáře:

- Chci data načíst do cache: vyšlu na sběrnici žádost o čtení. Pokud ostatní cache řádek neznají, vydá mi ho paměť a přejdu do stavu Exclusive. Pokud ho někdo má jako Exclusive, předá mi ho a oba přejdeme do Shared. Pokud ho má jako Shared, já přejdu také do Shared. Pokud ho někdo měl Modified, zapíše ho do paměti, já si ho přečtu a oba přejdeme do Shared.
- Chci do dat zapisovat: pokud je můj řádek Modified nebo Exclusive, zapíši a přejdu do Modified. Je-li Shared, vyšlu žádost o přivlastnění řádku. Ostatní procesory přejdou do Invalid, já přejdu do Modified.
- Chci řádek odstranit z cache: je-li Shared nebo Exclusive, mohu data zapomenout a přejít do Invalid. Je-li řádek Modified, zapíši jej do paměti.

Nevýhodou tohoto mechanismu je, že při každém předání změněného řádku mezi procesory musí dojít jeho k zápisu do hlavní paměti. To zdržuje, jelikož sběrnice je sice pomalejší než vnitřek procesoru, ale stále rychlejší než paměť. Rozšíříme tedy protokol, aby připustil i případ, kdy data jsou sdílená a novější než v hlavní paměti. Některý z procesorů si musí zapamatovat, že je data časem potřeba do hlavní paměti zapsat. Oddělíme tedy od stavu Shared nový stav Owned, který tuto potřebu popisuje. Právě jeden procesor se pak bude nacházet ve stavu Owned a ostatní mohou mít tentýž řádek Shared. Přechody mezi stavy se změní takto:

- Jsem-li ve stavu Modified a někdo jiný si chce řádek přečíst, předám mu ho, přejdu do Owned a on do Shared.
- Odstraňuji-li řádek z cache a je Shared, zahodím ho. Je-li Modified nebo Owned, zapíši ho do hlavní paměti.

V ostatních případech se Owned chová stejně jako Shared.

Protokol M(O)ESI je tedy plně transparentní, ale rychlost přístupu do paměti velmi závisi na tom, jak často se řádky přehazují mezi jednotlivými procesory. Je potřeba dávat pozor na správné zarovnání dat a vyhnout se tomu, aby data s různými druhy přístupu sdílela tentýž řádek.

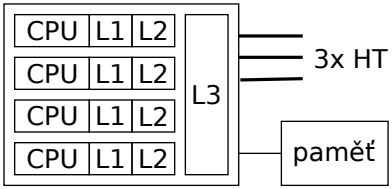
Implementace SMP

Ve světě PC se donedávna více procesorů nacházelo jen ve velkých serverech. Dnes můžeme běžně potkat:

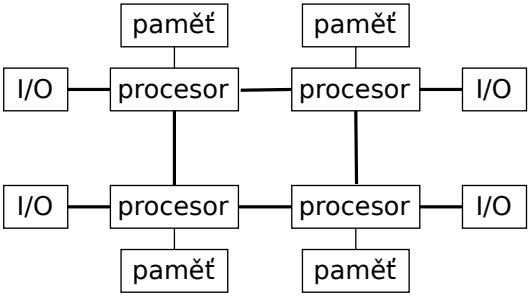
- *Vícejádrové procesory* – více výpočetních jader *(cores),* de facto samostatných procesorů v jednom pouzdře. Často jsou některé úrovně cacheové hierarchie sdílené mezi více jádry.
- *Hyperthreading* – jedno jádro je rozděleno na dvě logická, každé má svůj stav programu (registry apod.), ale výkonné jednotky jsou společné. Tím se elegantně zaplní část bublin v pipelinách a využijí se pauzy, v nichž jeden z programů čeká na data z paměti. Nevýhodou je efektivně poloviční velikost cache, takže někdy může dojít i ke zpomalení.
- *Více procesorů* – u serverů se dnes běžně vyskytují základní desky se dvěma nebo čtyřmi vícejádrovými procesory. Starší systémy připojují všechny procesory ke společnému řadiči, který zařizuje jak jejich vzájemnou komunikaci, tak přístup k paměti. Novější stroje obsahují paměťové řadiče přímo v jednotlivých procesorech a procesory spolu komunikují rychlými dvojbodovými sběrnicemi. Přístup k lokální paměti je pak rychlejší než k paměti sousedního procesoru (tomu se říká NUMA – Non-Uniform Memory Architecture).

Skutečnou konfiguraci svého počítače se pod Linuxem dočtete ve virtuálních souborech `/proc/cpuinfo` a `/sys/devices/system/cpu/`.

Jako příklad uveďme procesor AMD Phenom II X4. V něm najdeme 4 jádra, každé z nich má 64 KiB 2-cestné `L1I` cache, 64 KiB 2-cestné `L1D` cache a 512 KiB 16-cestné `L2` cache. Všecha jádra sdílejí 6 MiB 48-cestné `L3` cache, paměťový řadič a 3 kanály sběrnice HyperTransport pro připojení vstupně-výstupních zařízení.



Serverová verze téhož procesoru umožňuje použít všechny 3 kanály i pro propojení s ostatními procesory v systému. Typická konfigurace pak vypadá třeba takto (tučné spoje jsou kanály HyperTransportu):



Programování v C

Nejběžnějším překladačem pod Linuxem je GCC (GNU Compiler Collection) – původně překladač jazyka C, dnes překládá i C++, Fortran, Pascal, Adu a řadu dalších jazyků. Mimo to se používá též ICC (Intel C Compiler, pro procesory Intel někdy generuje lepší kód než GCC, pro AMD typicky horší) a LLVM (Low-level Virtual Machine, výzkumný překladač).

Zajímavé přepínače při překladu

- *Typ výstupu:* `-S` (assembler), `-fverbose-asm` (přidat komentáře), `-E` (výstup preprocesoru), `-c` (object file).
- *Informace pro debugger:* `-g`, `-ggdb` (pozor, po použití některých optimalizací debugger nemusí fungovat).
- *Optimalizační módy:* `-O2` (základní), `-O3` (maximální, zahrnuje i kontroverzní optimalizace typu rozbalování smyček), `-Os` (na velikost), `-Ofast` (zapne navíc optimalizace, které v některých případech porušují standardy; například předpokládá, že sčítání floatů je asociativní).
- *Volba typu procesoru:* `-march=` (ovlivňuje používanou instrukční sadu), `-mtune=` (ovlivňuje optimalizaci). Na výběr je buďto konkrétní rodina procesorů nebo `generic` (kompromisní optimalizace pro všechny rodiny), případně `native` (řídí se podle procesoru, na kterém překládáme).
- *Typ generovaného kódu:* `-m32` (32-bitový pro i386), `-m64` (64-bitový pro AMD64), `-mfpmath=sse` (lze zapnout i na i386).
- *Jednotlivé optimalizace:* `-fomit-frame-pointer` (na AMD64 automaticky), `-funroll-loops` (rozbalování smyček), různé optimalizace smyček: `-floop-interchange`, `-floop-block`, `-floop-flatten` a mnoho dalších.
- *Optimalizace řízená profilem:* `-fprofile-generate`, `-fprofile-use`.
- *Globální optimalizace:* `-flto`, `-fwhole-program`.

Běžné optimalizace

- *Inlining* – triviální funkce jsou zkopírovány na místo volání. To si

lze vyžádat i explicitně pomocí `inline`, ale překladač to nemusí vyslyšet.

- *Propagace konstant* – konstantní podvýrazy jsou vypočetny v čase překladu.
- *Udržování proměnných v registrech* – překladač se snaží co nejvíce aktivních proměnných uchovávat v registrech (klíčové slovo `register` nepoužívejte). Pozor na *aliasing ukazatelů,* překladač často potřebuje dokázat, že dva ukazatele neukazují na totéž místo.
- *Eliminace společných podvýrazů (CSE)* – pokud počítame totéž vícekrát, je to vypočteno jen jednou (např. `(x[1]+x[2])*(x[1]-x[2])` indexuje pole pouze dvakrát).
- *Přesouvání invariantního kódu* – pakliže ve smyčce počítáme nějaký podvýraz, který se mezi iteracemi nemění, je vysunut ven. Někdy se též transformuje řídící proměnná cyklu (například neiterujeme přes indexy v poli, ale přímo přes adresy v paměti).
- *Rozbalování smyček.*
- *Autovektorizace.*
- *Specializace funkcí* – od jedné funkce může vzniknout více kopií pro konkrétní hodnoty argumentů.

Rady do života

- Nedělejte práci za překladač. Výrazy typu `(x<<2)+x` sice mohou vypadat učeně, ale když napíšete `5*x`, překladač sám zvolí optimální způsob výpočtu (a ten obvykle závisí na kontextu). Pište přímočaře.
- Zarovnávejte data. Překladač to obvykle udělá za vás, ale pokud chcete například zarovnat na řádek cache, musíte se o to postarat sami. Pozor, bloky paměti alokované pomocí `malloc()` nebývají dostatečně zarovnané, použijte `posix_memalign()` nebo `mmap()`.
- Dávejte si pozor na aliasing ukazatelů. Někdy pomůže `restrict`.
- Nepočítejte v menších typech než `int` (stejně se takové hodnoty při prvním použití na `int` samočinně přetypují), ani je nepředávejte jako parametry funkcí.
- Dbejte na lokalitu dat. Bývá lepší pole struktur než několik samostatných polí.

Zajímavá rozšíření

GCC oproti standardu jazyka nabízí četná rozšíření. Pro optimalizaci kódu se hodí zejména atributy funkcí a typů, tedy konstrukce `__attribute__((...))`:

- `const` – funkce nemá vedlejší účinky a výsledek závisí pouze na hodnotách argumentů.
- `pure` – oproti `const` smí jestě číst globální paměť.
- `hot`, `cold` – označení horkých (často používaných) a studených míst.

- `optimize` – lokální nastavení optimalizačních přepínačů.
- `always_inline` – donutí k inlinování (pokud to opravdu nejde, ohlásí chybu).
- `noinline` – zabrání inlinování (hodí se při profilování).
- `flatten` – funkce se „zploští“ tzn. z ní volané funkce se zainlinují.
- `aligned` – u typu či proměnné si lze vybrat, jak moc je zarovnaný.

Vektorové operace

GCC disponuje datovými typy pro vektory:

```
typedef int __attribute__((vector_size(16))) v4i;
```

Vektory lze indexovat jako pole, přiřazovat, funguje na nich základní aritmetika (`+`, `*`, atd.), ostatní operace jsou k dispozici v podobě zabudovaných funkcí, např.:

```
__builtin_ia32_paddb(x, y);
```

Mimo to je překladač v jednoduchých případech schopen kód autovektorizovat.

Programování na SMP

Pokud je to možné, doporučujeme výpočet rozdělit do několika nezávislých *procesů,* které spolu komunikují jen minimálně (ať už pomocí souborů nebo rourami). Je-li nutné vyměňovat hodně dat, hodí se buďto procesy komunikující skrz *sdílenou paměť* nebo *vlákna* (sdílejí celý adresní prostor, viz `pthread.h`).

Při používání společné paměti je především potřeba dbát na konsistenci dat. Obvyklé způsoby řešení:

- *Atomické typy* – jednoduché operace (test, přiřazení) s některými datovými typy jsou zaručeně atomické (většinou spíš de facto než de iure), ale pozor, překladač může přístupy do paměti vyoptimalizovat nebo přerovnat, takže je třeba použít buď `volatile`, nebo bariéru.
- *Atomické instrukce* – některé instrukce zaručují atomické provedení páru čtení + zápis. Na i386 je k dispozici například atomický `inc`, `dec`, `xchg` a `cmpxchg`. V GCC dostupné jako `__sync_...` (i bariéra).
- *Atomická systémová volání* – například `write()` do souboru otevřeného s `O_APPEND` serializuje.
- *Synchronizační primitiva* – různé konstrukce pro vzájemné vyloučení procesů:
  - *Mutexy* (z Mutual Exclusion) – dvoustavové zámky (zamčeno může mít nejvýše jeden proces, ostatní na zámek čekají), operace `Lock` a `Unlock`. Existuje i read/write varianta (zamčeno pro zápis nejvýše jednou, pro čtení i vícekrát).
  - *Semafory* – počítadlo, které nesmí klesnout pod nulu. Operace `Down` (sníží, pokud už bylo nulové, čeká), `Up` (zvýší a případně probudí čekající).

- *Hashované mutexy* – hashovací funkce přiřadí adrese objektu jeden z pole mutexů. Řeší problémy s granularitou zamykání.
- *Zamykání souborů* a jejich intervalů – `flock()`, `fcntl()`. Hodí se, pokud nepoužíváme vlákna, ale procesy, mezi nimiž sdílíme bloky paměti či soubory.

Současná implementace mutexů a semaforů na Linuxu je pomalá pouze v případě, kdy čekáme (voláme jádro) nebo zámek předáváme mezi procesory (ping-pong s řádky cache).

Pokud potřebujeme od nějakých dat samostatnou instanci pro každé vlákno, hodí se `pthread_getspecific()`, případně `__thread`.

Pro „quick and dirty“ programy může mnoho práce ušetřit obecná paralelizační knihovna. Nejznámější jsou OpenMP pro C a Fortran a Thread Building Blocks pro C++. Výkon pravděpodobně nebude optimální, ale rozhodně lepší než u sekvenčního programu.

Prostředky pro sledování běhu programů

- `strace` – vypisuje provedená systémová volání
- `valgrind`, *http://www.valgrind.org/* – dynamický analyzátor programů (ve skutečnosti překladač, který do strojového kódu přidává instrumentaci). Umí odhalovat běžné chyby při práci s pamětí (např. použití neinicializovaných dat), simulovat cache a predikci skoků.
- `perf`, *https://perf.wiki.kernel.org/* – sbírá data z hardwarových počítadel událostí a umí zjistit, kolik ve které části programu nastalo cache missů, přístupů do paměti, mispredikovaných skoků apod. Možnosti závisí na schopnostech konkrétního procesoru.

Literatura

Odkazy na další zdroje informací najdete na *http://mj.ucw.cz/vyuka/aim/,* na stejném místě naleznete případnou novější verzi tohoto textu.