KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

ASSEMBLER

ALEŠ KEPRT

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc, verze 2.4.2008

Abstrakt

Tento text má za cíl posloužit jako učebnice programování v assembleru. Popisuje 32bitový assembler procesorů Intel (řady označované jako IA32 či x86) s cílem umožnit čtenářům či studentům základní pochopení principů, na kterých funguje assembler a potažmo mikroprocesory v současných počítačích. Ve výuce je assembler obvykle součástí cvičení některého z kurzů v řadě „Operační systémy“.

Cílová skupina

Text je primárně určen pro studenty oboru Aplikovaná informatika uskutečňovaného v kombinované formě na Přírodovědecké fakultě Univerzity Palackého v Olomouci a dále pro studenty oborů Informatika a Aplikovaná informatika uskutečňovaného v prezenční formě tamtéž. Všichni jmenovaní studenti mají programování v assembleru jako povinný předmět svého bakalářského studia.

Poznámka

Tato publikace obsahuje řadu obrázků (technických diagramů) popisujících různé detaily týkající se procesorů řady x86. Většina těchto jich je převzata z publikace [IA32], kterou vydala společnost Intel jako úplný manuál ke svým procesorům. Všechny svazky je možno volně stáhnout z internetu.

Obsah

1 Motivace

Studijní cíle: Prostudováním kapitoly získá student motivaci k dalšímu studiu. Dozví se, co to assembler je, proč se učí a jaká je jeho pozice v džungli programovacích jazyků.

Klíčová slova: assembler, strojový kód, instrukce, vyšší a nižší jazyky

Potřebný čas: 25 minut.

V úvodu kurzu assembleru je vhodné věnovat jistý čas vysvětlení, co nás vlastně motivuje učit se tento programovací jazyk. Řada z vás se s ním zřejmě doposud „na vlastní kůži“ vůbec nesetkala a toto může být pociťováno jako jistý signál, že assembler nebude až tak důležitým jazykem.

Skutečně, v dnešní době se v assembleru příliš neprogramuje. Jedná se o historický programovací jazyk, který je z dnešního pohledu poměrně zvláštní a pro tvorbu programů, jaké se dnes běžně dělají, je překonaný či zcela nevhodný. Ve výuce informatiky má však významné postavení právě pro svou zvláštnost a historickou povahu. Assembler je stejně starý jako sám mikroprocesor a pochopíme-li, jak funguje assembler, pochopíme tím současně, jak funguje samotný mikroprocesor.

Prvotní motivací k učení assembleru tedy je snaha o pochopení principů, na kterých „opravdu“ stojí počítačové programy. Ať už napíšete váš program v kterémkoliv jazyce, ať už použijete při vaší práci kterékoliv programovací paradigma, váš program nakonec vždy vykonáván nějakým mikroprocesorem a má podobu posloupnosti primitivních instrukcí, které odpovídají programu zapsaném v assembleru.

1.1 Procesor, strojový kód a assembler

Průvodce studiem

V minulosti bylo mnoho knižního papíru zbytečně popsáno na téma prvních počítačů a prvních mikroprocesorů. Každé takové prvenství však trvalo jen do okamžiku, kdy byly po letech odtajněny ty či ony vojenské záznamy. Pochopitelně, nic tak důležitého není jako první použito v civilní sféře a veřejně.

Mikroprocesory, či jednodušeji procesory, v počítačích i mnoha jiných elektronických zařízeních jsou nejčastěji von Neumannova či Harvardského typu (podle příslušné počítačové architektury). Tyto typy procesorů se vyznačují tím, že procesor vykonává program, který je uložen v paměti. Procesor tedy zná či umí vykonávat určité příkazy – říkáme jim odborně **instrukce**. Každá instrukce je přitom z lidského pohledu jedno či několik čísel uložených za sebou v paměti. Každý počítačový program je tedy de facto jen pole čísel.

Instrukce je základní primitivní příkaz, kterému procesor rozumí.

Průvodce studiem

Víte jaký je rozdíl mezi CPU a mikroprocesorem? CPU je centrální zpracovací/procesní jednotka, hlavní mozek počítače. Mikroprocesor je jednou z mnoha možných implementací CPU. Počítače a CPU existovaly již před vynálezem mikroprocesorů. Vynález mikroprocesoru však výrazně redukoval velikost i cenu počítačů a dnes je běžné, že každý počítač obsahuje mnoho mikroprocesorů. CPU je naopak v každém počítači obvykle pouze jedna, i když v poslední době roste obliba víceprocesorových počítačů či vícejádrových procesorů, kde pojem „procesor“ či „jádro“ se vztahuje právě k počtu CPU. Jak je vidět, pojmy CPU a procesor často splývají či bývají zaměňovány a z hlediska softwaru z toho nevzniká zásadní problém.

Tento způsob zápisu programu v řeči počítače se nazývá „strojový kód“. Pro lidi to je jen posloupnost čísel, jedniček a nul, či jiných symbolů, kterými zobrazíme paměť počítače do člověku čitelné podoby. To, že tato čísla umíme číst, samozřejmě ještě neznamená, že umíme porozumět tomu, co takový program dělá.

Strojový kód je jediným jazykem, kterému procesor přímo rozumí.

Kromě číselného zápisu vytvářejí výrobci ke každému svému procesoru také předpis pro lidský zápis instrukcí. Každá instrukce má nějaké jméno, obvykle zkratku z anglického slovesa. Případně je doplněná o nějaké další předpony či přípony. Namísto dlouhé řady čísel pak program ve strojovém kódu má podobu řady slovních zkratek, které zapisujeme na řádky pod sebe.

Jako příklad si uveďme program pro výpočet absolutní hodnoty celého čísla. V C++ může vypadat například takto:

```
int a;
  ...
if(a<0)  a=-a;
```

První řádek deklaruje proměnnou a samotný kód programu má pak délku jednoho jediného řádku, jehož smysl je na první pohled jasný. Ve strojovém kódu tento program vypadá takto: A1 78 71 41 00 83 F8 00 7D 02 F7 D8. Takto je zapsán v šestnáctkové soustavě, má délku 12 bajtů.

V tomto 12bajtovém programu nám čísla na druhé až páté pozici udávají, ze které proměnné absolutní hodnotu počítáme. Že nevíte, proč to jsou zrovna tato čísla? Je to číslo paměťové buňky s naší proměnnou. Když programujeme ve strojovém kódu, tato čísla musíme skutečně sami řešit a to dělá programy velmi nepřehledné. Proto programování ve strojovém kódu bylo možné jen v dávných dobách, kdy počítače měly velmi malé paměti.

Náš program můžeme namísto čísel zapsat v instrukčních kódech takto:

```
mov eax,dword ptr 417178h
cmp eax,0
jge 4113B4h
neg eax
```

První slovo na každém řádku je instrukce a za ním pak následuje upřesnění k instrukci.

Jednou z důležitých vlastností strojového kódu je, že každý procesor má svůj vlastní strojový kód. Rozdíly mezi jednotlivými procesory přitom mohou být poměrně značné. Programátorům jsou však tyto rozdíly obvykle skryty díky tomu, že používají tzv. „vyšší programovací jazyky“.

Průvodce studiem

Pojem „výšky“ jazyka souvisí s tím, zda je jazyk bližší člověku (pak je nazýván vyšším), či stroji (pak je nazývám nižším či nízkoúrovňovým). Různé programovací jazyky přitom můžeme považovat za vyšší či nižší podle toho, z jakého úhlu pohledu je hodnotíme. Například C++ je objektově orientovaný jazyk, který je z mnoha hledisek vyšším jazykem, avšak z pohledu práce s pamětí a bezpečnosti jde o jazyk spíše nižší.

Jak již možná tušíte, assembler je jedním z nejnižších jazyků, protože se velmi podobá strojovému kódu. Slovo assembler se většinou píše s malým a na začátku, neboť obvykle nejde o jméno nějakého úplně konkrétního jazyka, ale právě pouze o zkratku pro jistý způsob zápisu strojového kódu se symbolickými adresami. Slovo assembler je odvozeno od programu, který převádí zdrojový kód tohoto jazyka do strojového kódu (v angličtině: sloveso assemble, od něj odvozené podstatné jméno assembly). Říkat jazyku názvem programu, který s ním pracuje, se může zdát matoucí či chybné, ale je to velmi zažité a nebudeme se tomu bránit.

Jazyk assembler mezi vyšší programovací jazyky nepatří, je to totiž jen jakási jiná varianta strojového kódu, kde kromě slovních názvů instrukcí ještě nahradíme ostatní zbylá čísla. Assembler býval v minulosti v češtině také nazýván opisem „Jazyk symbolických adres“ a tento dnes již prakticky nepoužívaný název přesně vystihuje podstatu assembleru: Adresy (nesrozumitelná čísla) totiž assembler nahrazuje symbolickými jmény.

Podívejme se tedy znovu na předchozí příklad a zapišme jej v assembleru:

```
   mov eax, a
   cmp eax, 0
   jge skip
   neg eax
skip:
```

Ačkoliv pro pochopení tohoto programu stále musíme znát význam jednotlivých instrukcí „zkratek“ v prvním sloupci, takto zapsaný program je již pro člověka přehledný.

Pojmy strojový kód a assembler často splývají, neboť si odpovídají téměř 1:1 a člověk obvykle píše program v assembleru proto, aby jej stroj přesně tímto způsobem vykonával ve strojovém kódu. Velmi často tedy tyto dva pojmy splývají a neuděláme chybu, budeme-li jej v dalším textu považovat za jedno a totéž.

Průvodce studiem

Možná vás nyní zajímá, jak moc se liší strojové kódy či assemblery jednotlivých procesorů. Bohužel, rozdíly jsou často obrovské. Jména instrukcí bývají jiná a zápis jejich parametrů také. Samotný princip fungování procesorů je však vždy stejný. Zajímavou pozici zde zaujímají jazyky moderních výpočetních strojů, jako je například bytecode jazyka Java. Je to strojový jazyk, ke kterému byl teprve později vytvořen hardwarový procesor. Takový procesor pak vlastně přirozeně vykonává program v jazyku Java. Na zkoumání, jak moc je takový „pozpátku“ vytvořený výpočetní stroj podobný klasickým procesorům, je však zde na začátku studia příliš brzy. K této otázce se vrátíme později.

1.2 Co se naučíme a proč

Zdá se vám, že stačí mít po ruce tabulku vysvětlující význam jednotlivých instrukcí a můžeme směle programovat v assembleru? Do jisté míry ano. Zřejmě vás ale v následujících kapitolách překvapí, jak primitivní tyto instrukce jsou a jak málo jedna taková instrukce umí. Assembler například nezná příkaz `if`. Stejně tak nezná `for`, `while` či procedury a funkce. Žádný z těchto prvků celkem běžných ve vyšších jazycích ve skutečnosti procesor přímo nezná. Naučit se programovat v assembleru tedy kromě učení dlouhé řady jmen a významů instrukcí znamená také naučit se postupy, jak z těchto primitivních instrukcí sestavovat smysluplné programy.

Jedním ze způsobů, jak v assembleru programovat, je naučit se pomocí jeho instrukcí sestavit kód odpovídající zmíněným příkazům `if` či `for`, funkcím apod. To je taky zřejmě nejvhodnější způsob, jak s assemblerem začít.

Průvodce studiem

Je s podivem, že klasické učebnice programování v assembleru nejdou touto cestou. Namísto toho, aby vysvětlily, jak v assembleru udělat program pomocí prvků, které člověk zná z jiných jazyků, příliš času věnují vyjmenovávání jednotlivých instrukcí procesoru. K vysvětlení „jak programovat“ je věnováno naprosté minimum prostoru a zkáza je doplněna tím, že namísto úloh, ve kterých má assembler reálný smysl, je tento procvičován na primitivních a nudných příkladech.

V praxi se assembler používá jednak ve velmi specifických situacích, kde není možné použít jiný programovací jazyk (například některé části jádra operačního systému), nebo tam, kde programy napsané v jiných jazycích nejsou dostatečně rychlé. Ačkoliv například C++ zvládne 99% úloh stejně efektivně nebo alespoň uspokojivě dobře (ve srovnání s assemblerem), například výpočty v oblasti počítačové grafiky jsou v assembleru obvykle výrazně rychlejší než v C++ či jiných jazycích. Obecně je program v assembleru rychlejší při splnění těchto dvou podmínek:

1. Pracujeme s velkým polem čísel.
2. S každým prvkem pole uděláme stejnou operaci.

Konkrétní příklady si ukážeme v dalších kapitolách.

K procvičování assembleru se všeobecně hodí úlohy pracující s polem čísel či znaků. Nejčastěji budeme pracovat s textovými řetězci, tedy poli znaků. I takové programy jsou v assembleru obvykle rychlejší než v jiných jazycích, avšak tyto operace jsou celkově časově nenáročné, takže i kdyby byl program v assembleru třeba $2\times$ rychlejší, ušetříme tím v praxi několik mikrosekund, což je nepodstatné. S poli se v assembleru přitom dá pracovat pouze pomocí pointerů. Zde je zajímavé, že zatímco v C++ mívají studenti obvykle s pointery problémy, díky assembleru pak systém pointerů v jazyku C++ pochopí.

Kromě práce s poli využijeme assembler také k pochopení principu volání funkce (či procedury), předávání parametrů a souvisejících věcí. Tyto zdánlivě triviální operace jsou totiž v assembleru doslova odhaleny ve své nahotě, což nám odhalí princip fungování lokálních proměnných a programového zásobníku, který znáte z přednášek Operační systémy či Struktura počítačů.

Shrnutí

V této úvodní kapitole jsme získali motivaci k dalšímu studiu assembleru. Vysvětlili jsme si, co to je assembler a proč se jej máme učit. Zároveň jsme se seznámili s některými základními pojmy, jako instrukce.

Pojmy k zapamatování

- Strojový kód
- Assembler
- Vyšší a nižší jazyky

Kontrolní otázky

1. *Co je to instrukce?*
2. *Proč je assembler nižším programovacím jazykem?*
3. *Jaký je hlavní rozdíl mezi strojovým kódem a assemblerem?*
4. *Proč se u strojového kódu a assembleru velmi často používá právě šestnáctková číselná soustava?*
5. *Z dřívějšího studia byste měli znát jazyky Scheme a C++ nebo C#. Který z nich je vyšší a proč?*

Cvičení

1. Převeďte první čtyři čísla zápisu programu v našem příkladě ze šestnáctkové do desítkové soustavy.

Úkoly k textu

1. Pro další studium assembleru je velmi vhodné ovládat dvojkovou a šestnáctkovou číselnou soustavu. Zopakujte si je.

2. Zopakujte si také algoritmy pro převod mezi desítkovou, dvojkovou a šestnáctkovou soustavou. To vám ušetří mnoho času při práci s assemblerem.

Řešení

1. 161 120 113 65. Všimněte si, že šestnáctková čísla jsou vždy dvojciferná a reprezentují hodnotu jednoho bajtu. Každá cifra může nabývat jedné z šestnácti hodnot (10 čísel + 6 prvních písmen abecedy). Převod do desítkové soustavy uděláme pomocí vzorce: 16× první cifra + druhá cifra. Pro převod opačným směrem je dobré znát zpaměti násobky čísla 16, díky čemuž velmi rychle zjistíme první cifru. Druhou cifru pak snadno dopočítáme tak, aby součet byl roven původnímu číslu.

2 Úvodní krůčky s assemblerem

Studijní cíle: V této kapitole začneme programovat v assembleru. Jelikož je to složitý jazyk, nebudeme látku probírat lineárně a některé detaily odložíme na později. Vyzkoušíme si napsat první malý prográmek v assembleru a na konci kapitoly si připomeneme zásady strukturovaného programování. Tato teorie nám při další práci hodně pomůže.

Klíčová slova: instrukce, operandy, registry, vyšší jazyky, strukturované programování

Potřebný čas: 55 minut.

2.1 Strategie programování v assembleru

Aby vám assembler nezlomil vaz, je velmi důležité seznámit se hned na začátku se správnou strategií, jak v assembleru programovat. Máme-li řešit nějaký úkol v assembleru, nejprve mu musíme rozumět a musíme umět jej algoritmicky vyřešit (na úrovni slovního popisu algoritmu). Namísto slovního popisu řešení často poslouží přímo řešení v jiném programovacím jazyce, například C++ nebo C#. V tom případě se samozřejmě musíme omezit na ty konstrukty, které z těchto jazyků umíme převést do assembleru.

Obráceně, pokud nevíme, jak zadanou úlohu řešit, nevyřešíme ji ani v assembleru. Je-li například zadána úloha „Změňte textový řetězec na tentýž řetězec pozpátku a udělejte to přímo v místě původního řetězce.“, pak tuto úlohu buď umíme vyřešit v jiném jazyku a v assembleru můžeme postupovat stejně, nebo ji vyřešit neumíme a pak se studium programovacího jazyka spíše mění ve studium algoritmizace a dokud správný algoritmus nesestavíme, lopota s assemblerem je zbytečná. U zadání složitějších úloh proto bude uvedeno i ukázkové řešení v jiném jazyce.

2.2 Assembler a vyšší programovací jazyky

Jak jsme viděli výše, náš první program v assembleru nebyl obvyklý „Hello World“[1], ale mnohem jednodušší program na výpočet absolutní hodnoty celého čísla. Hello World v assembleru by pro nás totiž na úvod byl poněkud složitý – assembler totiž neumí pracovat přímo s textem a hlavně nemá žádný příkaz pro výpis na obrazovku.

Právě proto, že v assembleru jsou i zdánlivě jednoduché věci velmi složité a pracné, naše studium začneme na tzv. **inline assembleru**. To je assembler, který vpisujeme do vyššího programovacího jazyka. V našem případě to bude jazyk C++. Stručný úvod do jazyka C++ pro programátory v C# najdete v příloze C. Kromě C++ lze použít i jazyk C, Turbo Pascal nebo jiné starší jazyky. Naopak většina současných jazyků, včetně C#, Visual Basicu a Javy, vkládání assembleru neumožňuje. V dalším textu bude často zmiňován jazyk C++, avšak to, co o něm bude napsáno, často platí i o dalších vyšších jazycích. Proto si za heslo C++ obvykle můžete dosazovat váš oblíbený jazyk.

Inline assembler se vkládá do jiného jazyka.

Průvodce studiem

Nejčastější chyba, kterou začátečníci v assembleru dělají, je, že se úlohy snaží řešit různými krkolomnými způsoby, jakoby neznali C++. Přitom v assembleru by 90% kódu mělo být napsáno stejným způsobem jako v jiných jazycích, pouze s jinou syntaxí.

[1] Hello World = Ahoj světe

2.3 Struktura programu v inline assembleru

Inline assembler je assembler vepsaný do kódu jiného jazyka, v našem případě to bude Visual C++ 2005. Jako příklad uveďme nám známý výpočet absolutní hodnoty, tentokrát bude napsán jako funkce v C++ a vlastní výpočet bude v assembleru:

```
int abs(int value) {
  _asm {
    mov eax,value
    cmp eax,0
    jge skip
    neg eax
skip:
    mov value,eax
  }
  return value;
}
```

Na příkladu vidíme, že kód assembleru je uzavřen do bloku s hlavičkou `_asm`. Čára (podtržítko) před slovem asm ve starších verzích překladačů být nemusela, některé překladače naopak vyžadují dokonce dvě čáry.

Dále vidíme, že kód assembleru může používat proměnné z C++, naopak v assembleru nemůžeme přímo definovat celé funkce či vracet návratovou hodnotu. Kód programu se nám o jeden řádek prodloužil (oproti kódu absolutní hodnoty v předchozí kapitole), důvodem je předání výsledku zpět do proměnné `value`.

Jednotlivé řádky představují instrukce assembleru. Slovem instrukce přitom označujeme jak úvodní slovo na řádku, tak celý řádek. Parametry, které jsou za mezerou, nazýváme **operandy** Jak vidíme, jednotlivé instrukce mají různý počet operandů – obecně jich může být nula až tři, ale u každé instrukce jich musí být správný počet, aby program měl smysl. (Toto naštěstí kontroluje překladač.) Výpočetní instrukce vždy uloží výsledek do prvního operandu. U dvojoperandových instrukcí obvykle první operand slouží zároveň jako vstupní parametr operace, takže provedením operace přepíšeme první operand novou hodnotu a jeho původní hodnotu tak ztratíme.

Pojem instrukce má v assembleru dva podobné významy.

Parametry instrukcí nazýváme operandy.

Nyní jsme se tedy seznámili se syntaxí inline assembleru. Bohužel, tato syntaxe se v jednotlivých překladačích assembleru liší. Stejně tak se liší i syntaxe vkládání assembleru do vyšších jazyků. My se budeme držet této syntaxe, kterou zavedli či přijali především Intel, Microsoft a Borland.

2.4 Naše první instrukce

Naučme se tedy první instrukce. Začneme samozřejmě s těmi nejjednoduššími, jsou uvedeny v tabulce 1. Dodejme jen, že velká a malá písmena se v assembleru standardně nerozlišují. Lze to však zapnout, případně lze zapnout částečné rozlišování u identifikátorů sdílených s částmi programu napsanými v jiném jazyce (to se hodí právě při spolupráci s C++).

Všimněte si, že všechny instrukce v tabulce jsou aritmetické operace obsahující také přiřazování. Toto jsou totiž téměř jediné instrukce, jejichž význam a použití odpovídá vyšším programovacím jazykům. Navíc násobení a dělení je dosti komplikované z hlediska operandů. K tomu se ale ještě dostaneme později.

Důvodem toho, že ve všech těchto základních instrukcích je přiřazení, je způsob, jakým procesor pracuje. Zatímco pro člověka je přirozený například vzorec $a = b + c$, procesor takto výpočet provést nemůže. Měl by totiž problém, co dělat s mezivýsledkem $b+c$ v tom krátkém okamžiku, než se uloží do a. Navíc všechny proměnné jsou uloženy ve stejné (tedy společné) paměti

C++	Assembler	operandy	název instrukce
=	mov	2	move
+=	add	2	add
−=	sub	2	subtract
*=	mul	1	multiply (unsigned)
*=	imul	1,2,3	multiply (signed)
/=	div	1	divide (unsigned)
/=	idiv	1	divide (signed)
++	inc	1	increment
−−	dec	1	decrement
>>=	shr	2	shift right (unsigned)
>>=	sar	2	shift arithmetic right (signed)
<<=	shl	2	shift left
&=	and	2	(bitwise) and
|=	or	2	(bitwise) or
^=	xor	2	(bitwise) xor
∼=	not	1	(bitwise) not
−	neg	1	negate (změní znaménko čísla na opačné)

Tabulka 1: Nejjednodušší instrukce.

počítače a v jednom okamžiku tedy lze provádět jen jednu operaci (čtení či zápis jedné paměťové buňky).

Nelze v jedné instrukci dvakrát adresovat paměť.

Zde jsme narazili na jednu důležitou poučku: **Procesor může v jedné instrukci jen jednou adresovat paměť.** Jinými slovy, jedna instrukce může paměť číst i zapisovat, ale jen na stejném místě. Příkladem takové instrukce je `inc`, která přečte hodnotu z paměti, zvýší o jedničku a uloží zpět na stejné místo.

Registr je paměťová buňka uvnitř procesoru.

Řada dalších instrukcí paměť buď jen přečte, nebo jen zapíše. Hodnoty přečtené z paměti můžeme naštěstí dočasně uložit v lokální paměti uvnitř procesoru. Máme tam k dispozici několik málo paměťových buněk, které nazýváme **registry**. Registr je tedy něco jako proměnná, která je však uvnitř procesoru. Ve zdrojovém kódu assembleru rozlišíme registry od proměnných pomocí speciálních názvů – základní čtyři registry používané pro běžné výpočty se nazývají `eax`, `ebx`, `ecx` a `edx`. Podrobněji se k registrům vrátíme později.

Nyní tedy můžeme napsat program pro výpočet součtu $a = b + c$, použijeme k tomu operace z tabulky 1 a jeden registr.

```
mov eax, b
add eax, c
mov a, eax
```

Všimněte si, že výsledek střádáme do registru `eax` a teprve na konci výpočtu jej uložíme do proměnné a. Tento postup je v assembleru běžný – každá práce s pamětí je vždy „dražší“, než práce s registry. Registry navíc nemají ono omezení jedné adresace paměti, takže zatímco instrukce `mov a,b` není možná, protože by měla dvě adresace paměti, tatáž instrukce se dvěma registry, např. `mov eax,edx`, funguje.

2.5 První cvičení

Se znalostí těchto základních instrukcí už můžete napsat váš první vlastní program. Napište funkce pro výpočet obsahu a obvodu obdélníka. Vzorové řešení obsahu obdélníka následuje.

```
int ObsahObdelnika(int a, int b) {
```

```
  _asm {
    mov eax,a
    imul eax,b
    mov a,eax
  }
  return a;
}
```

Návratovou hodnotu vracíme tak, že výsledek uložíme do proměnné a a použijeme příkaz return v C++. Můžeme také využít znalosti toho, že hodnoty z funkcí se nativně vracejí v registru `eax`. Co to pro nás znamená? Když funkci deklarujeme jako vracející int, ale nenapíšeme do ní žádný příkaz return, funkce vrátí tu hodnotu, kterou necháme v registru `eax`. (Tento princip je samozřejmě podpořen překladačem C++, který běžně u zapomenutého příkazu return hlásí chybu. Nedělá to však u funkcí obsahujících kód v assembleru.) Vyzkoušejte.

2.6 Další základní konstrukty

Kromě základních výpočetních instrukcí v tabulce 1 si uveďme také několik dalších konstrukcí, které fungují stejně či podobně jako v C++. Najdete je v tabulce 2.

C++	Assembler	poznámka
//	;	jednořádková poznámka
goto	jmp	jump
label:	label:	
&	offset	reference (adresa proměnné)
*	[ ]	dereference (proměnná z adresy)
(char)&	byte ptr	(tvrdé) přetypování na 1bajtovou hodnotu
(short)&	word ptr	(tvrdé) přetypování na 2bajtovou hodnotu
(long)&	dword ptr	(tvrdé) přetypování na 4bajtovou hodnotu

Tabulka 2: Základní konstrukty.

2.7 Práce s různě velkými čísly

Zastavme se chvíli u přetypování, které má v assembleru několik důležitých odlišností od C++. Assembler především u datových typů rozlišuje jen jejich velikost. Je mu tedy jedno, když pomícháte dva různé typy, pokud oba mají stejný počet bajtů/bitů. Díky tomu většinou vystačíme se třemi typy uvedenými v tabulce: `byte`, `word` a `dword`. Přetypování pomocí `ptr` funguje jen pro práci s pamětí (nelze použít u registrů) a měli bychom ho používat jen pro změnu typu z většího na menší. V opačném případě totiž čteme či zapisujeme víc bajtů, než ve skutečnosti máme k dispozici, čímž nejspíš dojde k paměťové chybě. Tento fenomén si můžete vyzkoušet na následujícím příkladě. Co tento program vypíše? (V C++ dosáhneme stejného efektu příkazem ∗(**long**∗)&b=512;.

Tři základní typy: byte, word, dword.

```
char b,c;

void main() {
  _asm {
    mov word ptr b,512
  }
  printf("b=%i c=%i\n", b, c);
}
```

V případě celých čísel, se kterými pracujeme i v assembleru, jde při tzv. přetypování de facto jen o rozšíření či zúžení čísla na jiný počet bitů. V případě rozšíření na více bitů vždy chceme, aby rozšířená hodnota byla rovna hodnotě původní. V případě zúžení toto požadujeme pouze tehdy, pokud se hodnota do menšího počtu bitů vejde; v opačném případě dochází k ořezání. Jelikož tyto operace jsou prováděny přímo v procesoru, assembler se chová stejně jako vyšší jazyky (např. převodem čísla -1000 na typ `byte` dostaneme vždy číslo $+24$).

2.7.1 Zúžení na menší typ

Zúžení na menší typ provádíme bez ohledu na to, zda je číslo znaménkové, či neznaménkové. Při práci s registry jednoduše použijeme menší část registru. Při práci s proměnnou použijeme operátor `ptr` (viz tabulka 2). Máme-li tři následující proměnné

```
char p1;
short p2;
int p4;
```

pak můžeme konverze provádět například takto:

```
mov al, byte ptr p2      ; byte <-- short
mov al, byte ptr p4      ; byte <-- int
mov ax, word ptr p4      ; word <-- int
mov ebx, VelkéČíslo
mov p1, bl               ; char <-- word/dword
mov p2, bx               ; short <-- dword
```

Všimněte si, že když máme pro příklad nějaké velké číslo v registru `ebx`, jeho uložení do menší proměnné musíme provést pomocí menšího jména téhož registru `bl`. Totéž platí pro 2bajtový `bx` a samozřejmě taktéž pro `eax`, `ecx` a `edx`. U registrů `esi` a `edi` máme jen 2bajtovou verzi, ale ne 1bajtovou.

2.7.2 Rozšíření na větší typ

U neznaménkových čísel provádíme rozšíření doplněním nul do všech nově přidaných bitů. Toto je velmi jednoduchá operace a lze ji provést dvěma způsoby:

- Instrukce `movzx` je obdobou instrukce `mov` umožňující, aby první operand byl větší než druhý. Všechny vyšší bity jsou doplněny nulami.
- U registrů je možno provést převod dvoukrokově: Nejprve vynulujeme větší registr, pak do jeho menší části uložíme menší hodnotu. Například přiřazení `eax` ⟵ `bl` provedeme takto:

  ```
  mov eax,0
  mov al, bl   ; eax <-- bl
  ```

U znaménkových čísel rozšíření provádíme jinak. Důvodem je, že používáme doplňkový kód, kde u záporných čísel je jednička v nejvyšším bitu a hlavně také ve všech nepoužitých nejvyšších bitech. (Dokažte!) Algoritmus rozšíření je tedy dán doplňkovým kódem: Kladná čísla a nulu rozšiřujeme stejně jako u neznaménkových doplněním nuly, zatímco záporná čísla rozšiřujeme doplněním jedničky do všech přidaných bitů. De facto tedy provedeme okopírování nejvyššího bitu původní hodnoty do všech přidaných bitů. Toto by nebylo úplně jednoduché udělat, máme však k dispozici několik speciálních instrukcí. Od procesoru 80386 je k dispozici univerzální instrukce `movsx`, která funguje podobně jako `movzx`, ale všechny vyšší bity prvního operandu místo nulování nastavuje na hodnotu nejvyššího bitu druhého operandu. Uveďme několik příkladů.

```
movsx p4, al        ; int <-- sbyte
movsx p2, al        ; short <-- sbyte
movsx p4, ax        ; int <-- sword
movsx eax, p1       ; sdword <-- char
movsx ax, p1        ; sword <-- char
movsx eax, p2       ; sdword <-- short
```

Pro neznaménkové konverze platí totéž (pouze použijeme instrukci `movzx` místo `movsx`).

2.8 Popis instrukcí

V dalších kapitolách se budeme průběžně seznamovat s jednotlivými instrukcemi procesoru, budeme si je přitom obvykle popisovat podrobněji, než dosud. U každé instrukce bude uvedeno:

- Počet a typy operandů. Rozlišujeme přitom tři základní typy operandů: reg (registr), mem (adresa paměti) a imm (přímá hodnota). Adresou paměti rozumíme případ, kdy operandem je číslo určující adresu paměti. Přímou hodnotou rozumíme případ, kdy operandem je číslo.
- Popis instrukce. Z popisu je jasné, co přesně instrukce dělá, včetně případných zvláštností a omezení.
- Ovlivněné aritmetické příznaky. Některé instrukce po sobě zanechávají jisté „stopy“ v podobě příznaků. Vliv instrukce na základní příznaky si vždy uvedeme. (A to i přesto, že v počátcích ani ještě nebudeme rozumět, k čemu je to dobré.)

Typy operandů lze nejlépe pochopit na příkladě. Komentář na každém řádku popisuje, jakého typu jsou operandy v daném příkladě. V závorce je uveden tentýž kód v jazyku C++ (fungoval by za předpokladu, že by v C++ bylo možno přímo pracovat s registry).

```
mov eax,23              ; reg,imm  (eax = 23)
mov eax, x              ; reg,mem  (eax = x)
mov eax, [x]            ; reg,mem  (eax = x)
mov eax, offset x       ; reg,imm  (eax = &x)
mov eax,dword ptr x     ; reg,mem  (eax = *(int*)&x)
mov eax, ebx            ; reg,reg  (eax = ebx)
mov x,23                ; mem,imm  (x = 23)
mov [eax],0             ; mem,imm  (*eax = 0)
mov a, b                ; mem,mem - toto nelze!
```

Příklad také ukazuje význam operátorů reference a dereference a fakt, že pokud je operandem globální proměnná, hranaté závorky můžeme vynechat. (Některé alternativní překladače assembleru však toto zjednodušení nepodporují.)

Na úvod si představíme jednu instrukci, která se nám bude velmi hodit v úvodních cvičeních – instrukci podmíněného skoku podle hodnoty registru `ecx`.

2.9 Podmíněné skoky

Skoky dělíme na nepodmíněné a podmíněné. Nepodmíněný skok odpovídá příkazu `goto` známému z vyšších jazyků. Podmíněný skok je odlišný tím, že se provede jen při splnění určité podmínky. Pomocí podmíněných skoků lze tedy dosáhnout stejného efektu, jako při použití příkazu `if` ve vyšších jazycích.

U všech skokových instrukcí označujeme cíl skoku stejným způsobem: Kdekoliv v programu vytvoříme **návěští** uvedením jména a dvojtečky. Toto jméno je pak parametrem skokové instrukce. Nejjednodušší skokovou instrukci si hned představíme, na další se podíváme až později.

2.9.1 Instrukce `jecxz` imm

Instrukce `jecxz` provede skok na adresu danou operandem, pokud je registr `ecx` roven nule. Operand je přímá adresa (cíl skoku) a musí být v okolí $+/-$ 128 bajtů od místa instrukce skoku. (Kontrolu, že se nesnažíme skákat moc daleko, zajistí každý solidní překladač. Některé však při pokusu o příliš daleký skok vytvoří chybný kód a bez varování.)

Příznaky: neovlivňuje

Z popisu instrukce `jecxz` je vidět, že jméno návěští uvedené jako parametr se fyzicky přeloží na číslo – přímou hodnotu (immediate value). Toto číslo je číslem paměťové buňky, kde je umístěn příkaz programu následující za označeným návěštím. Podmíněné skoky neumožňují skákat jiným způsobem než přes přímou hodnotu. U nepodmíněného skoku máme daleko širší možnosti – můžeme totiž skákat i na hodnoty registrů, čili cílovou adresu skoku můžeme vypočítat při běhu programu.

V případě, že potřebujeme opačný test a skok, když je ecx nenulové, pomůžeme si opisem s nepodmíněným skokem.

```
  jecxz pokracuj
  jmp pryc          ;skočí pryč, když je ecx nenulové
pokracuj:
```

2.10 Zásady strukturovaného programování

V úvodním povídání o assembleru jsme se mj. dozvěděli, že v assembleru, přestože vypadá na první pohled zvláštně, se programuje stejně jako v jiných imperativních jazycích (C++, C# apod.). Tyto stejné rysy imperativního programování lze pojmenovat jako **strukturované programování**. Tento styl programování přináší přehlednost a snižuje chybovost programů, je proto vhodné jej dodržovat. V C++ nebo C# je přitom povinný a nelze se mu vyhnout. V assembleru však lze programovat i jinak než strukturovaně, proto je velmi důležité si hned v úvodu připomenout, jak vlastně správně psát programy, aby strukturované byly. (Přehlednost a bezchybnost programů přece je naším cílem.)

Imperativní programovací jazyky vyjadřují program jako posloupnost příkazů, které jsou vykonávány postupně po sobě. Dále máme k dispozici různé řídící příkazy, kterými lze pořadí vykonávání příkazů změnit. Příkladem takového řídícího příkazu je dobře známý příkaz `if`, který může přeskočit část kódu podle toho, zda platí, nebo neplatí nějaká podmínka. Dalším příkladem je příkaz `goto`, kterým lze přímo přesunout vykonávání příkazů na jiné místo. Tyto dva příkazy se v minulosti obvykle používaly dohromady takto: Program měl očíslované řádky. Příkazem ve tvaru `goto n` přešlo vykonávání na řádek `n`. Podmíněný příkaz měl tvar `if` podmínka then n+ a přešel na řádek `n` v případě, že podmínka platila. Toto muselo stačit k psaní celých programů.

Strukturované programování přineslo do imperativního programování pořádek tím, že zakázalo používat příkaz skoku `goto`. Namísto toho jsou definovány tři základní stavební „kameny“, pomocí kterých program skládáme:

1. Podmíněný příkaz (`if`), který podle splnění, či nesplnění podmínky vykoná jeden nebo druhý blok kódu. Tento příkaz může být v kombinaci s else, nebo bez – to není podstatné. Hlavní je, že po otestování podmínky program neskáče nikam pryč, ale vykoná, nebo nevykoná označený blok kódu, který přímo následuje za podmíněným příkazem.

2. Příkaz cyklu (`while`), který opakuje blok kódu tak dlouho, dokud je splněna podmínka. Toto je obdoba předchozího příkazu, rovněž jde o jistý druh podmíněného vykonání kódu. Není přitom podstatné, zda je podmínka testována před, nebo po provedení bloku.

3. Podprogramy (procedury či funkce), které můžeme zavolat. Po provedení kódu podprogramu se řízení vrací zpět na místo volání a vykonávání programu pokračuje příkazem následujícím za zavoláním podprogramu. Není přitom podstatné, zda podprogram vrací nějakou hodnotu.

Přínos této trojice pravidel se ukázal tak velký, že nově vznikající jazyky dokonce ani jinak než strukturovaně programovat neumějí. Řada těchto jazyků přitom podporuje i příkaz nestrukturovaného skoku `goto`, avšak lze jím skočit jen na jiné místo téže funkce.

Naopak assembler jakožto jazyk historický nevynucuje strukturované programování. Příkazy `if` nebo `while` zde vůbec nejsou a nestrukturovaným skokem můžeme skákat bez omezení na libovolné místo programu. Tato možnost skákání kamkoliv může dosti zkomplikovat první kroky začínajícího programátora, proto je vhodné na podobné možnosti raději zapomenout a snažit se dodržovat zásady strukturovaného programování z vlastní vůle.

Assembler má ještě jednu zvláštnost pramenící ze stejného důvodu: Ačkoliv zde existují procedury, které lze zavolat a vrátit se z nich tak, jak to popisuje strukturované programování, má to od strukturovaných jazyků jednu zvláštnost: **Na konci každé procedury musíme explicitně uvést instrukci ret.** V opačném případě procedura „neskončí“, jak bychom čekali, ale program pokračuje dalším příkazem v paměti – čili obvykle začne vykonávat následující proceduru. Zatím nás procedury nemusejí trápit, protože budeme používat pomoc jazyka C++. Později si toto pravidlo znovu připomeneme.

Průvodce studiem

Nutnost explicitně ukončovat procedury a možnost nestrukturovaně skákat na libovolné místo programu jsou vlastnosti vyplývající z technické realizace programu v paměti. Nejen hardwarové procesory, ale i moderní uměle vytvořené assemblery, např. nativní jazyk objektově orientované platformy .NET, mají tyto vlastnosti. To, že programátor nebude nestrukturovaně skákat do nepatřičných míst svého programu, musí zajistit překladač konkrétního vyššího jazyka, např. C#.

Shrnutí

V této kapitole jsme začali programovat v inline assembleru, který vkládáme do kódu jazyka C++. Seznámili jsme se s řadou nových pojmů, jako operand a adresování paměti. Představili jsme si sadu aritmetických instrukcí. Tyto mají jména v podobě zkratek z anglických sloves a bohužel si je budete muset zapamatovat zpaměti. Bez dobré znalosti názvů instrukcí se totiž programuje velmi pomalu. Naučili jsme se také vracet hodnoty z funkcí, použili jsme to při výpočtu obsahu a obvodu obdélníka. V závěru kapitoly jsme si zopakovali zásady strukturovaného programování a vysvětlili si jeho dopad na programování v assembleru.

Pojmy k zapamatování

- Inline assembler
- Instrukce a operand
- Adresování paměti
- Registry
- Strukturované programování

Kontrolní otázky

1. *Které tři programové konstrukty jsou základními kameny strukturovaného programování?*
2. *Kam se ukládá výsledek aritmetických operací (např. když provedeme součet pomocí instrukce* `add`*)?*

3. *Snad úplně nejzákladnější instrukcí je přiřazení hodnoty. Jak se jmenuje?*
4. *Co je to operand?*
5. *Proč přiřazení $a = b$ nelze provést v jedné instrukci?*

Cvičení

1. V sekci byl uveden program pro výpočet obsahu obdélníka. Napište ještě program pro výpočet obvodu obdélníka. (Obvod je součtem délek stran, čili $O = 2(a + b)$.)
2. Napište program pro výpočet obsahu trojúhelníka dle délky základny a jí příslušné výšky. (Vzorec je $S = \frac{a \cdot v_a}{2}$.)
3. Podmíněnými skoky nelze skákat na hodnotu registru, ale jen na přímou hodnotu. Ukažte, jak lze toto omezení vyřešit či obejít.

Úkoly k textu

1. Pokud jste tak ještě neudělali, spusťte si Visual Studio 2005 a ověřte, že vám správně funguje, včetně assembleru. Založte nový projekt typu „C++ / Win32 Console Application“ a vyzkoušejte ukázkové programy z této kapitoly.
2. Pro výměnu dvou hodnot existuje instrukce `xchg`, která funguje podobně jako `mov`, ale nastavuje oba svoje operandy na hodnotu opačného operandu. Jak byste vyměnili hodnoty dvou registrů, kdybyste neměli k dispozici `xchg`, ani další registr, proměnnou či zásobník? Nápověda: Použijte tento algoritmus: 1. A+=B, 2. B–=A, 3. A+=B, 4. B=0–B. Realizujte tento algoritmus v assembleru.
3. Lze předchozí algoritmus použít i pro případ, kdy potřebujeme vyměnit hodnotu mezi registrem a proměnnou či mezi dvěma proměnnými? Zdůvodněte.
4. Napište program pro výpočet obsahu trojúhelníka dle vzorce $S = a \cdot v_a$. Proměnné a a v_a budou typu `unsigned int`.
5. Předchozí program upravte na typ `unsigned short`.
6. Předchozí program upravte na typ `unsigned char`.
7. Napište program pro výpočet aritmetického průměru tří čísel (dle vzorce $(a+b+c)/3$).

Řešení

1. V programu využijeme vracení hodnoty v registru `eax` a tento registr použijeme také pro samotný výpočet. Pro násobení dvěma pak využijeme instrukci `shl`, což je pro tento účel nejelegantnější řešení.

```
int ObvodObdelnika(int a, int b) {
  _asm {
    mov eax, a
    add eax, b
    shl eax, 1
  }
}
```

   Alternativně bychom mohli použít např. instrukci `add eax,eax`, naopak použití pravého násobení není příliš vhodné, protože jde o velmi náročnou operaci. (O náročnosti násobení a dělení ještě bude řeč v jedné z dalších kapitol.)

2. Tentokrát budeme potřebovat opravdové násobení, takže použijeme `imul`. Pro dělení dvěma nám ale postačí `sar` (případně `shr` – v tomto případě funguje obojí stejně).

```
int ObsahTrojuhelnika(int a, int va) {
  _asm {
    mov eax,a
    imul eax,va
    sar eax,1
  }
}
```

3. Pomůžeme si nepodmíněným skokem, který umí skákat na hodnotu registru. Dejme tomu, že chceme udělat podmíněný skok na adresu v registru edx při ecx rovném nule (čili jakoby `jecxz edx`). Program, který toto dělá, může vypadat například takto:

```
  jecxz plati
  jmp konec
plati:
  jmp edx
konec:
```

3 Registry a adresování paměti

Studijní cíle: V této kapitole se zaměříme na data – vysvětlíme si, jak fungují registry a jak se pracuje s globálními proměnnými. Představíme a vysvětlíme pojmy adresace paměti a paměťové modely. Naopak lokální proměnné odložíme na následující kapitolu.

Klíčová slova: registr, adresace paměti, přímé adresování, nepřímé adresování

Potřebný čas: 80 minut.

3.1 Registry

Jak již víme z úvodní kapitoly, každý procesor má vnitřní paměťové buňky zvané registry. Ty, se kterými jsme se již setkali (eax, ebx, ecx, edx) jsou základní pracovní registry nejrozšířenější řady procesorů x86 ve 32bitovém režimu a fungují takto na všech procesorech typu 80386 nebo novějších. Podívejme se nyní na další detaily k registrům.

Průvodce studiem

První otázka, která vás asi napadne, je: „Kolik těch registrů vlastně v procesoru je?“ Celkový počet registrů se v jednotlivých procesorech liší, poměrně mnoho z nich však používá jen operační systém k řízení běhu počítače. Pro nás však budou podstatné jen výpočetní registry a registr příznaků, o kterých bude řeč dále v této kapitole. A ty jsou počínaje procesorem 80386 stále stejné.

Průvodce studiem

Jelikož registr není v paměti, nelze vytvořit pointer na registr. Pointer totiž vždy ukazuje někam do paměti.

Každý registr má 32 bitů.

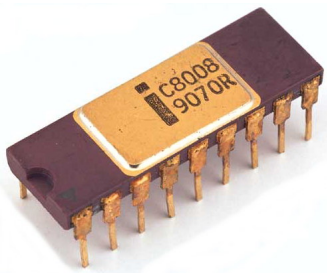
Každý registr má velikost 32 bitů, umí tedy uložit až 32bitové číslo (tedy v rozsahu zhruba 0 až 4 miliardy bez znaménka, či -2 až +2 miliardy se znaménkem). Je to výhoda, protože nám to zjednodušuje práci s procesorem. V některých systémových registrech jsou některé bity nevyužité nebo naopak navíc, to nás však nemusí trápit.

Jak již víme, základní čtveřice registrů se jmenuje `eax`, `ebx`, `ecx`, `edx`. Historické procesory měly pouze jeden takový registr jménem A, jako akumulátor. Později se logicky přidávaly další registry pojmenované dle abecedy. Každý z nich má i mnemotechnickou pomůcku pro zapamatování jeho významu (dle písmen a–b–c–d), ale tím se nebudeme trápit, neboť tyto registry jsou si ve většině případů rovnocenné.

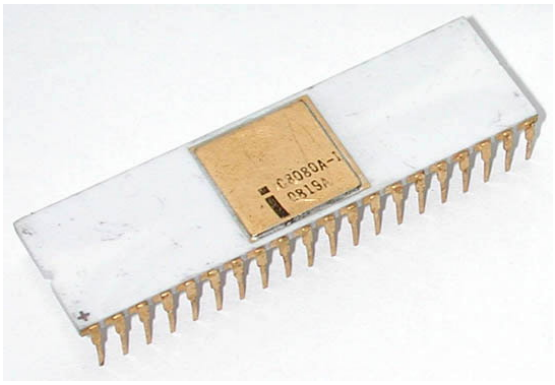
Písmeno `e` na začátku názvu určuje, že jde o 32bitový registr. Vynecháme-li jej, dostaneme 16bitovou verzi téhož registru (ax, bx, cx, dx). Jde tedy stále o jeden a tentýž registr, pouze pomocí písmena `e` určíme, zda jej chceme použít celý, nebo jen jeho první (dolní) polovinu. K horní polovině 32bitového registru se takto jednoduše nedostaneme, můžete ale velmi elegantně vyměnit dolní a horní polovinu registru – slouží k tomu instrukce `rol reg,16` (prvním operandem je název registru, např. eax).

Průvodce studiem

Důvodem toho, že každý registr má různá jména podle toho, s kolika jeho bity (a kterými) chceme pracovat, je historický. Dnešní procesory jsou odvozené od 32bitového 80386, ten od 16bitového 8086 a ten zas od 8bitového 8080 a jeho předchůdce 8008. Designéři



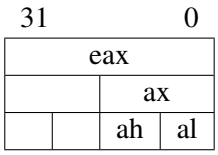
Obrázek 1: Procesor Intel 8008 (rok 1972).



Obrázek 2: Procesor Intel 8080 (rok 1974).

Intelu se s každou novou generací procesorů snažili vycházet z té stávající, takže vlastně „nabalovali“ další prvky ke stávající architektuře a stejně tomu bylo i u registrů.

Registry můžeme také používat jako 8bitové, tentokrát můžeme dokonce používat dolní dva bajty (čili obě poloviny 16bitového registru) tak, že příponu `x` nahradíme příponou `h` (high – vyšší) nebo `l` (low – nižší). Všechny možnosti pojmenování registru eax ukazuje obrázek 3, u registrů ebx, ecx a edx to platí stejně.



Obrázek 3: Přehled registru eax.

Průvodce studiem

Všimněte si, že dolní část registru je na obrázcích vždy značena vpravo a horní je vlevo. Toto odpovídá arabskému zápisu čísel, který se všeobecně používá (jednotky, čili dolní část je tam také vpravo). S tímto zápisem zprava doleva se setkáme i v dalších kapitolách.

esi a edi jsou indexové registry.

Další dva registry, které lze použít při běžné práci, se jmenují `esi` a `edi`. Můžete je opět používat k libovolnému účelu. Jejich nestandardní jména pocházejí z historických procesorů, kde sloužily pouze pro adresování paměti (si = source index, di = destination index). Tyto registry ale neumožňují pracovat s 8bitovými částmi. Také další dva adresovací registry `ebp` a `esp` mají stejné vlastnosti, jsou však používány k jiným účelům, takže při běžných výpočtech práci je většinou využívat nemůžeme. (O tom se ještě dozvíte více později.)

Na závěr zmíníme ještě dva systémové registry: V registru `eip` je udržována adresa právě vykonávaného kódu. Postupně se zvyšuje, jak procesor provádí jednotlivé instrukce, nebo se změní při volání procedur atp. Tento registr ale nemůžeme přímo číst ani zapisovat instrukcemi jako `mov`, změnit ho však můžeme velmi snadno pomocí podmíněných či nepodmíněných skoků (skok `jmp imm` je totiž totéž jako `mov eip,imm`). Registr `eflags` je příznakový registr, jeho 32 bitů slouží jako příznaky různého dění v procesoru či jako konfigurace jeho chování. O příznacích budeme mluvit později.

eip ukazuje na aktuální pozici vykonávání kódu.

3.2 Výběr registrů

K určitým operacím se doporučuje používat přednostně určité konkrétní registry. Není to sice technicky nutné, ale zpřehledňuje to program. Snažte se při práci s registry dodržet tato doporučení:

- Při kopírování a další práci s bloky paměti používejte esi jako pointer čtení a edi jako pointer zápisu, ebx pak přednostně jako třetí registr (je-li potřeba). Nelze-li rozlišit čtení a zápis, použijte opět přednostně tyto tři registry v uvedeném pořadí. Pokud vaše funkce nemá lokální proměnné, ani vstupní parametry, můžete jako čtvrtý v řadě použít registr ebp.
- Pro indexaci polí použijte přednostně ebx. Obvykle se pak sčítá esi+ebx či edi+ebx.
- Pro všechny ostatní operace kromě adresace paměti používejte esi, edi a ebp až jako poslední v řadě.
- Jako počítadlo cyklů (počet opakování) použijte ecx, pro dvojí do sebe vnořené smyčky použijte ecx přednostně pro vnitřní smyčky a pro vnější použijte kterýkoliv jiný registr, proměnnou nebo u delšího kódu opět ecx chráněný uložením na zásobník.
- Pro matematické operace použijte eax, jako druhý pak edx.
- Pracujete-li pouze s bajty (typ char apod.), nebojte se využívat každou část registru samostatně. Čili např. místo al a bl použijte raději al a ah, ušetříte tak celý jeden registr.
- Potřebujete-li ještě další registry, pomocí instrukce `rol reg32,16` můžete vyměnit spodní a horní polovinu registru. Tímto způsobem získáte dvojnásobný počet 16bitových registrů.
- Proměnné používejte, až když jsou všechny registry obsazené.
- Zásobník a push–pop používejte pouze tehdy, když už nelze použít ani proměnné. (Například při ukládání dat dynamických rozměrů.)
- Segmentové registry pro ukládání dat používat nelze (v MS-DOSu ano, ale je to poněkud pomalé).
- Pro zkušenější: Jako počítadla cyklů můžete použít ecx rozdělený na dvě nebo i tři části. Vnitřní smyčku počítejte pomocí cl nebo cx. Můžete mít i dvě vnořené vnitřní smyčky a počítat je pomocí cl a ch. Vnější smyčku pak počítejte ve zbytku registru, tj. ch, pokud je volný, nebo horní polovinu ecx, pokud je celá spodní polovina obsazená vnitřní smyčkou/smyčkami.

3.3 Adresovací režimy

V předchozí kapitole jsme si vysvětlili možné kombinace operandů u instrukce `mov`, nyní si je rozebereme podrobněji. Instrukce ve tvaru `mov cíl,zdroj` je pro nás snadno srozumitelným zástupcem dalších instrukcí pracujících s pamětí (včetně všech aritmetických instrukcí z tabulky

1). Každý z operandů přitom může být v různém adresovacím režimu, všechny však musejí mít stejnou velikost (stejný počet bitů). Nejjednodušší dva typy operandů nepracují s pamětí:

Přímá hodnota (imm – immediate) je číselná konstanta. Použití přímých hodnot je velmi jednoduché a u většiny instrukcí bez omezení. Příklad: `mov a,23` – zde druhým operandem je přímá hodnota. Z principu věci vyplývá, že nikdy nejsou všechny operandy přímými hodnotami (taková instrukce by neměla smysl).

Registr (reg – register) je registr procesoru. Registry mají výsadní postavení a pracuje se s nimi velmi pohodlně, neboť většina instrukcí umožňuje bez omezení používat všechny registry jako jako první i druhý operand. Příklad: `mov eax,ebx` – zde oba operandy jsou registry.

Další typy operandů slouží k práci s pamětí (hovoříme o **adresování**, všechny dohromady značíme zkratkou mem).

Víme-li přesnou pozici v paměti, se kterou pracujeme (nejspíše jde o globální proměnnou), pak použijeme její adresu či jméno deklarované proměnné. Instrukce ve tvaru `mov a,23` má první operand v tomto tvaru – do strojového kódu se přeloží jako `mov [123456],23`, kde místo 123456 se dosadí adresa paměti, kde je proměnná `a`. Všimněte si, že `a` je totéž jako `[a]` – překladač si sám doplní hranaté závorky tam, kde je jen jméno proměnné. Pokud by hranaté závorky nedoplnil, z operandu by se vlastně stala přímá hodnota.

Pokud přesnou adresu nevíme, např. při práci s pointery, poli či objekty, použijeme **nepřímé adresování**. Nejprve adresu vypočítáme či načteme do některého registru, pak se na ni nepřímo odkazujeme opět pomocí hranatých závorek, např. `mov eax,[ebx]` načte do registru eax hodnotu z paměťového místa, na které ukazuje ebx (předpokládá se tedy, že ebx je pointer). Všimněte si zde, že assembler opravdu nerozlišuje mezi čísly a pointery – obé lze libovolně ukládat do stejných registrů a pouze pomocí použití hranatých závorek určíme, co má s registrem každá instrukce dělat.

Další variantou je **indexované adresování**, to je vhodné pro práci s poli. Příkladem je `mov eax,pole[ebx]` – pole je zde název globálního pole a ebx určuje pozici v tomto poli (posunutí od začátku pole). Identicky je možno použít registr pro označení počátku a přímou hodnotu pro posunutí – instrukce má tvar `mov eax,ebx[pole]` a je zcela totožná s předchozí variantou. Zápis adres proto bývá zvykem psát unifikovaně jako součet v hranatých závorkách v podobě `mov eax,[ebx+pole]`. Tento zápis také přesně vystihuje, co procesor dělá: nejprve sečte adresu pole a hodnotu registru a pak z této adresy přečte hodnotu a uloží ji do registru eax.

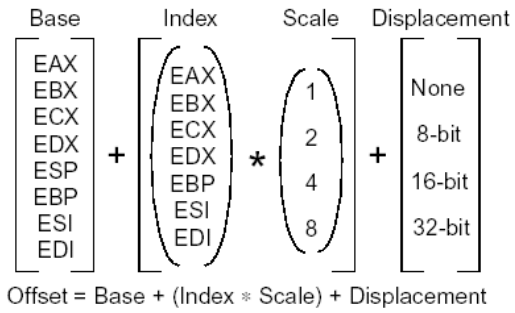
Indexované adresování je jen rozšířením nepřímého adresování a je možné použít také dva registry (např. `mov eax,[ebx+ecx]`).

Průvodce studiem

Syntaxe adresování pole je zde podobná jako v C++, assembler však počítá indexy pole vždy po bajtech, tedy bez ohledu na velikost jednotlivých prvků pole.

Možná nyní dumáte, proč a jak ten jinak celkem hloupý procesor dokáže v jedné instrukci provádět tak složité výpočty adres. Důvodem je, že výpočty adres patří k nejčastějším operacím, které program provádí, takže je vhodné právě tyto výpočty co nejvíce zrychlit. Proto má procesor vestavěnou schopnost skutečně velmi rychle provést vyhodnocení adresy paměti. Povolené samozřejmě nejsou všechny typy výpočtů, ale možnosti jsou poměrně bohaté: Můžete použít libovolnou kombinaci součástí ze vzorce

$$reg + reg * scale + imm$$



Obrázek 4: Možnosti výpočtu efektivní adresy (nepřímé adresování). [IA32]

Symbolicky lze toto schéma vyjádřit takto:

$$base + index * scale + displacement$$

Přesná definice možného tvaru nepřímé adresy je na obrázku 4. Displacement je libovolná přímá hodnota určující pevnou adresu paměti (např. globální proměnnou) či pevný index v poli (po bajtech). Base se obvykle používá při práci s lokálními proměnnými, kde určuje pozici dna zásobníku v aktuální funkci. Index je další registr použitý pro adresování pole – je to jednoduše index buňky v poli. Scale je násobící faktor při práci s poli – udává velikost jedné buňky pole. Může však nabývat jen hodnot 1, 2, 4 nebo 8; máme-li jinou velikost buněk v poli, musíme výpočet provést ručně.

Výpočty adres se provádějí během fáze dekódování instrukce, čili v ideálním případě trvají „nula“ cyklů procesoru (0T). Jedná se tedy o nejrychlejší možný výpočet. Pomocníkem je zde instrukce **lea**, která výsledek výpočtu adresy druhého operandu uloží do prvního operandu. Toho lze použít například pro velmi rychlé násobení devíti:

Instrukce lea vypočte adresu operandu.

```
mov eax, a
lea eax, [eax+8*eax]
```

V registru eax je nyní devítinásobek hodnoty proměnné `a`. Rychlejší způsob, jak dosáhnout téhož, neexistuje.

Jako další příklad si uveďme funkci, která přijme dva parametry: pointer na pole a index prvku, který má přečíst z pole a vrátit. V C++ je program jednoduchý:

```
int CtiPrvekPole(int *pole, int index) {
  return pole[index];
}
```

Pokuste se sami udělat totéž v assembleru. Vzorové řešení následuje.

```
int CtiPrvekPole(int *pole, int index) {
  _asm {
    mov eax, pole
    mov ebx, index
    mov eax, [eax+ebx*4]
  }
}
```

Všimněte si také, že v poslední instrukci používáme registr eax pro určení adresy a zároveň do něj zapisujeme výsledek operace. Tento postup funguje, protože procesor nejprve vypočte nepřímou adresu ve druhém operandu, pak z ní přečte hodnotu a až potom zapíše výsledek do eax.

Průvodce studiem

Hranaté závorky jsou u každého přístupu do paměti. Můžeme si to představit tak, že celá paměť je jedno velké pole, do kterého přistupujeme. Nejprve vypočteme index (číslo buňky) v tomto poli, a pak hranatými závorkami s polem pracujeme. Indexy přitom můžeme buď vypočítávat ručně pomocí instrukcí jako `mov` a `add`, nebo použít rozšířené možnosti nepřímého adresování.

Pro další procvičení adresace si nejprve vytvořte pole s prvky o délce 7 bajtů.

```
struct T {
  char a,b,c,d,e,f,g;
};
```

Nyní v assembleru napište funkci, která přijme dva parametry udávající pointer na pole a číslo prvku. Funkce vrátí v registrech edx:eax hodnotu daného prvku pole.

```
T CtiPrvekPole(T *pole, int index);
```

Dodejme ještě, že s lokálními proměnnými se ve skutečnosti pracuje jinak než s globálními. Jejich adresa se totiž mění při jednotlivých voláních (vzpomeňte na rekurzi), takže nemohou být přeloženy jako displacement. Naštěstí, inline assembler umožňuje načíst a uložit běžnou lokální proměnnou opět použitím jejího názvu jako operandu. Jak se to přesně přeloží, to si vysvětlíme později spolu s dalšími taji volání funkcí.

Průvodce studiem

Zkratky reg, mem, imm a také r/m (registr nebo paměť) používáme v popisu jednotlivých instrukcí k zdokumentování, jakého typu mohou být jednotlivé operandy u dané instrukce. Žádná instrukce totiž nefunguje úplně s libovolným operandem a pomocí těchto značek si můžeme snadno ověřit, jaké máme u každé instrukce možnosti.

S globálními konstantami se pracuje stejně jako s globálními proměnnými. Některé překladače (včetně Visual C++) sice konstanty ukládají do speciální sekce paměti, kde je hardwarově blokován zápis, jejich adresa je však opět stejná během celého běhu programu, takže se stejně jako globální proměnné mohou překládat na displacement.

3.4 Datové typy a přetypování

V předchozí kapitole jsme se již seznámili se dvěma operátory `offset` a `dword ptr`. Nyní si osvětlíme, jak přesně fungují.

Assembler typy stejné velikosti dále nerozlišuje.

Assembler a potažmo procesor na datové typy příliš nehledí, jedná se tedy o vyloženě slabě typovaný jazyk. Platí zde jednoduchá poučka: **Procesor nezajímají datové typy, zajímá ho jen počet bajtů (čili délka).**

Typování si popíšeme opět na příkladu instrukce `mov`. U každého použití instrukce s operandy musí být jasně uvedena velikost operandů. (Až na výjimky musí být všechny neadresovací operandy stejně velké.) Je-li některým operandem registr (samozřejmě bez hranatých závorek), pak velikost tohoto registru je daná jeho jménem. Použijeme-li současně více registrů různých velikostí, je to u většiny instrukcí chyba (např. `mov eax,bl` použít nelze). Použijeme-li adresování paměti v jednom operandu a zároveň registr v jiném operandu, překladač bohužel toto odmítne přeložit, ačkoliv dle jména registru by mohl vyvodit i velikost druhého operandu.

Velikost operandů však překladač umí vyvodit z typu proměnné. Např. instrukce `mov a,0` má za operandy proměnnou a číslo. Číslo nemá velikost, takže velikost obou operandů je určena typem proměnné a. Tabulka 3 ukazuje seznam základních typů známých v assembleru a jim odpovídající typy v C/C++. V tabulce jsou rozlišeny znaménkové a neznaménkové typy, v praxi však až na ojedinělé případy nemá smysl toto rozlišovat, protože registry procesoru mají velikost 1, 2 či 4 bajty a stačí nám tedy základní typy `byte`, `word` a `dword`.

Typ assembleru	velikost	Odpovídající typ C/C++
byte	1	unsigned char
word	2	unsigned short
dword	4	unsigned int, unsigned long, všechny typy pointerů
qword	8	unsigned long long
sbyte	1	char
sword	2	short
sdword	4	int, long
sqword	8	long long

Tabulka 3: Datové typy assembleru a jim odpovídající typy C/C++.

Použijeme-li za operand proměnnou, která nemá velikost odpovídající některému základnímu typu, program nepůjde přeložit (např. `mov pole,0`). Stejně zle dopadneme u instrukcí bez uvedení velikosti (např. `mov [eax],0`). V těchto případech je třeba explicitně uvést velikost některého z operandů právě pomocí operátoru `ptr` (jak jsme si ukázali již v tabulce 2 na straně 14). Správný tvar instrukce pak je např. `mov typ ptr pole,0` nebo `mov typ ptr[eax],0`. Za typ dosadíme dword či jiný typ, dle potřeby.

Uvádíme-li v instrukcích pouze jména proměnných, překladač je vždy musí převést do nějakého jiného tvaru, protože samotný název proměnné v assembleru nic neznamená. Každé uvedení pouhého jména se překládá na tvar `typ ptr jméno` nebo `offset jméno`. Tyto dvě varianty mají přesně opačný význam (jako dereference a reference), takže je samozřejmě nezbytné rozumět, podle čeho překladač zvolí variantu, kterou použije. Naštěstí to není složité: Je-li proměnná použita v instrukci skoku či volání funkce, pak je jméno přeloženo do tvaru `offset jméno`, zatímco ve všech ostatních případech do tvaru `typ ptr jméno`.

Použití operátoru ptr si ukážeme na příkladu práce s polem. Úloha zní: V poli bajtů překopírujte prvek č. 2 a 3 na pozici 7 a 9. Nejprve zkuste úlohu vyřešit sami.

Dvě vzorová řešení následují.

```
mov eax,offset pole
mov bl,[eax+2]
mov [eax+7],bl
mov bl,[eax+3]
mov [eax+9],bl
```

Toto řešení je jednoduché a dobře srozumitelné. Nyní si ukažme ještě trošku zajímavější řešení.

```
mov bx,word ptr[offset pole+2]
mov byte ptr[offset pole+7],bl
mov byte ptr[offset pole+9],bh
```

Analyzujme tento kód: Pomocí `offset pole` získáme adresu pole, pak se pomocí +2 posuneme na buňku č.2. Pomocí hranaté závorky provedeme adresaci, první instrukce mov tedy načte obsah paměti do registru bx. To je dvojbajtový registr, čili načetli jsme dvě buňky pole současně. Další dva řádky je podobným způsobem zapíší na pozice 7 a 9. Tyto instrukce vyžadují použití operátoru ptr, jelikož hranatá závorka pro přístup do paměti ctí typ pole, tj. překladač

ze zápisu `offset pole` uvnitř hranaté závorky odvodí, že celý výraz má mít stejná typ jako `pole` a to není `word`, který my potřebujeme

Operátor `offset` použijeme tehdy, chceme-li získat adresu jména. Ať už jde o jméno proměnné, funkce, návěští, či něčeho jiného, operátorem offset získáme jeho adresu. Pochopitelně jde ve skutečnosti jen o část celé adresy, a to konkrétně offset. Tento operátor používáme nejčastěji pro získání adresy pole (`offset pole`) či adresy funkce (`offset funkce`), kterou později můžeme zavolat přes registr.

Instrukce call volá funkci.

Rozdíl mezi operátory `offset` a `ptr` můžeme ukázat také na volání funkce. Z následujících dvou řádků kódu je jen jeden správný:

```
call offset printf
call dword ptr printf
```

Který je správný? To záleží na tom, je-li printf funkce, či proměnná typu pointer na funkci.

Visual Studio 2005/2008 používá obě varianty, přesněji řečeno jednu z nich podle nastavení překladače. Ve výchozím nastavení (když si založíte projekt a nebudete nic měnit) je dobře druhý řádek, neboť všechny funkce knihovny CRT se volají odkazem přes tabulku adres a printf je tedy vlastně proměnná typu pointer na funkci. Toto chování ale není intuitivní pro naše pokusy v assembleru, proto je vhodné nastavení překladače přepnout: V nastavení projektu (Klikněte na tučný řádek v Solution Exploreru a zvolte Properties) zvolte statickou CRT knihovnu (C/C++ Code Generation / Run Time Library → Multi-threaded Debug). Potom můžete volat printf pomocí obyčejného **call** printf (a totéž platí i pro další funkce CRT).

3.5 Programový zásobník

Každý procesor podporuje kromě přímého a nepřímého přístupu do paměti také programový zásobník. V této sekci se seznámíme s několika základními instrukcemi pro jeho používání a později se k zásobníku ještě vrátíme v souvislosti s voláním funkcí a organizací lokálních proměnných při rekurzivních výpočtech. Právě tam se totiž zásobník s výhodou využije.

Jak již víme, na zásobník ukazuje registr esp. Posledně vložená hodnota (vrchol zásobníku) je přímo na adrese [esp], předposlední je na adrese [esp+4], atd. Následující hodnota pak bude vložena na [esp-4]. Z toho vyplývá, že vrchol zásobníku můžeme velmi jednoduše přečíst instrukcí mov (např. `mov eax,[esp]`).

3.5.1 Instrukce push reg/mem/imm

Instrukce push vloží hodnotu na zásobník.

Push vloží hodnotu na zásobník. Bez ohledu na velikost operandu, na zásobník se vždy vloží 4 bajty. Vložení probíhá takto: Nejprve se esp sníží o 4, potom se do [esp] zapíše hodnota operandu instrukce.

Příznaky: Neovlivňuje

3.5.2 Instrukce pop reg/mem

Instrukce pop vyzvedne hodnotu ze zásobníku.

Vyzvedne hodnotu ze zásobníku a uloží ji do operandu (registru či paměti). Vyzvednutí proběhne takto: Nejprve je přečtena hodnota z [esp] a je uložena do operandu. Potom se esp zvýší o 4.

Příznaky: Neovlivňuje

3.6 Paměťové modely

Pojem paměťový model popisuje organizaci paměti.[2] Paměťové modely se liší u jednotlivých procesorů, ale liší se také v různých operačních systémech. Paměť každého procesu je obvykle rozdělena na tři části: kód, data a zásobník. Data a zásobník přitom mohou splývat, nebo být odděleny. Pro vyšší jazyky je však často nutné, aby splývaly, takže zásobník a data jsou na stejném místě paměti.

V systému Windows, se kterým pracujeme, se používá jediný paměťový model: Nazývá se **flat** a má velmi jednoduchou strukturu, kde každý proces má vlastní lineární paměť. Lineární paměť, jak už sám název napovídá, je nestrukturovaný blok paměti, se kterým se bez jakýchkoliv omezení může pracovat jako s polem. Současné verze Windows navíc oddělují paměť jednotlivých procesů, takže chyba v programu neovlivní ostatní programy. Procesy spolu naopak sdílejí kód knihoven, přitom ale samozřejmě nevědí, že část jejich paměti je sdílená s jinými procesy. (Takto sdílená paměť je vždy přístupná pouze pro čtení.)

Lineární organizace paměti především znamená, že máme-li pointer na první prvek pole a přičteme k němu např. číslo 20000, dostaneme vždy pointer na dvacátýtisící prvek (pokud takový existuje). Nelineární organizace paměti se používala v minulosti (před Windows 95) a posouvání pointerů na jiné prvky pole tam bylo složitější.

Paměťový model flat se také nazývá **small** (anglicky malý). Tyto dva názvy jsou identické. Malý je proto, že pointer má stejnou velikost jako registr, čili umožňuje adresovat jen 4GB (2^{32} bajtů) paměti. Naproti tomu jiné paměťové modely (např. large – velký) používají pointery větší, než je jeden registr. Umožňují tak adresovat víc než 4GB, ale práce v těchto modelech je složitější a nebudeme se jí zabývat. Ve Windows se tyto modely nepoužívají, pro adresaci paměti nad 4GB je vhodnější použít 64bitové Windows, takže registry jsou 64bitové a opět pracujeme v režimu flat/small.

Průvodce studiem

V 16bitových systémech, jako je Windows 3.1 či MS-DOS, je model small omezen na 64KB paměti. Větší modely tam tedy byly daleko častěji používány než dnes ve Windows. Kromě modelu large byl oblíbený také model compact, kde data jsou adresována dvěma registry, ale kód je jako v modelu small omezen na 64KB. Tento hybridní přístup byl výhodný z hlediska rychlosti programů i šetření paměti (adresy kódu jsou kratší, takže celý program je kratší).

3.7 Možnosti práce s poli a pointery

Pole a pointer není totéž.

Práci s poli a pointery jsme si již vysvětlili v předchozím textu, neuškodí nám však shrnout všechny známé poznatky. V assembleru je především nutno pečlivě rozlišovat mezi poli a pointery, navíc mezi globálními a lokálními. O poli hovoříme tehdy, máme-li v paměti přímo za sebou víc proměnných stejného typu, které nemají samostatná jména. Namísto pojmenovávání jednotlivých proměnných je pojmenujeme všechny jako celek. Pole jako celek není datovým typem, nelze tedy uvést pole jako vstupní parametr (ani návratový typ) nějaké funkce. Toto „nelze“ je třeba brát doslova – všimněte si následující dvě deklarace funkcí:

```
void funkce1(int a[]);
void funkce2(int *b);
```

[2] Toto platí pro sekvenční procesory. Při paralelním zpracování popisuje paměťový model i další vlastnosti paměti, které se u sekvenčního zpracování nemusejí řešit.

Na první pohled by se mohlo zdát, že proměnná `a` je pole a proměnná `b` je pointer. To je však omyl! Ve skutečnosti jsou obě deklarace zcela totožné a obě proměnné jsou pointery. Pole prostě jako parametr předat nejde.

Při práci s polem v assembleru stačí pamatovat si několik jednoduchých pravidel, jak získat adresu prvního prvku pole. To se liší podle toho, zda máme k dispozici opravdu přímo pole, nebo pouze pointer na pole. Zohlednit musíme také, zda se jedná o globální, či lokální proměnnou. (Parametry funkcí samozřejmě řadíme mezi lokální proměnné.)

Globální pole – Globální proměnné jsou vždy na stejném místě paměti. Chceme-li pracovat přímo s globálním polem, např. s deklarací `int pole[20];` v globálním prostoru, pak operátorem `offset` můžeme přímo získat adresu prvního prvku, např: `mov eax,offset pole`.

Globální pointer – Opět platí, že globální proměnná má vždy stejnou adresu. Tentokrát však musíme načíst její obsah a tím teprve získáme adresu onoho pole. Např. při deklaraci `int *pointer;` tedy použijeme příkaz `mov eax,dword ptr [pointer]`. Existuje i kratší zápis `mov eax,pointer`. Pozor: Jde o dva zápisy totožné instrukce (to první je skutečná podoba instrukce, to druhé je pomůcka, kterou nabízí překladač).

Lokální pointer – Tentokrát je pointer lokální proměnnou, jeho adresa v paměti se tedy mění při jednotlivých voláních naší funkce. Překladač zde však opět nabízí užitečnou pomůcku ve tvaru `mov eax,pointer`. Přístup k lokální proměnné je tedy díky této pomůcce stejný jako u globální proměnné. Fyzicky se instrukce přeloží na `mov eax,[ebp+x]`, kde za `x` se dosadí offset proměnné vzhledem ke dnu zásobníku. Toto ale zatím nebudeme prozkoumávat.

Lokální pole – Nejprve důležitou poznámku: Jak víme, vstupní parametr deklarovaný jako `int pole[]` je pointer, nikoliv pole! Pokud tedy nejde o tento případ a my máme opravdu pole jakožto lokální proměnnou, pak jeho adresa je opět vztažena ke dnu zásobníku. K načtení adresy prvního prvku pole použijeme instrukci `lea`, čili např. `lea eax,[pole]`. Alternativně je možno vynechat hranaté závorky a napsat `lea eax,pole`.

K lokálnímu poli se dostaneme pomocí instrukce lea.

Jakmile získáme adresu prvního prvku pole a máme ji uloženou v některém registru, všechna složitá práce je hotova. Dále již jen používáme tento registr a operátor přístupu do paměti `[ ]`, např. `mov ebx,[eax]` načte 4bajtovou hodnotu do registru `ebx`, zatímco `mov bl,[eax]` načte 1bajtovou hodnotu do registru `bl`. (Jak víme, 4bajtové registry používáme pro běžná čísla typu int, 1bajtové obvykle pro znaky.)

V některých případech přístupu do paměti musíme přidat ještě operátor `ptr` na určení typu. Např. vynulování prvku pole nelze provést instrukcí `mov [eax],0`, protože překladač zde vůbec netuší, kolik bajtů vlastně chceme vynulovat. Před hranatou závorku tedy musíme přidat určení velikosti, např. `mov dword ptr[eax],0`. Překladače assembleru bohužel co se týče datových typů nejsou zrovna dokonalé a toto určení velikosti vyžadují někdy i v situacích, kdy bychom očekávali, že velikost je z deklarace proměnné zřejmá. Tyto situace je potřeba se naučit rozpoznat – když překladač zahlásí chybu nekompatibilních operandů a je to u instrukce s hranatou závorkou, nejspíše tam chybí určení velikosti pomocí `ptr`.

3.8 Další poznámky k práci s pamětí

V závěru kapitoly nakousneme několik dalších otázek, které se práce s pamětí týkají.

Stejně jako vyšší jazyky, i assembler umožňuje pracovat se strukturovanými datovými typy (struct), také se sjednocenými typy (union) a výčtovými typy (enum). Těmito věcmi se zatím ale nebudeme trápit.

Assembler umožňuje také objektově orientované programování (OOP), práce je to ovšem podstatně méně snadná než v C++. Když se OOP začínalo rozšiřovat, řada překladačů assembleru přišla s jeho podporou. V praxi se však tyto prvky assembleru nikdy příliš nepoužívaly a programátoři se zájmem o OOP přešli na nové vyšší jazyky. Existenci podpory OOP v assembleru tedy berme za kuriozitu.

Některé operace nad poli, jako je kopírování polí, mazání pole či vyhledávání v poli, se používají tak často, že dostaly i vlastní instrukce. Tyto instrukce jsou v assembleru procesorů Intel dokonce již od dávných časů 8bitových procesorů. V současných procesorech se však již nepoužívají, neboť je paradoxně rychlejší napsat například kopírování pole pomocí načítání hodnot z jednoho pole do registrů a jejich následné ukládání do druhého pole. Přestože tedy tyto instrukce na rychlosti programu nepřidají, později se k nim ještě vrátíme a některé se naučíme používat. Zatím je ale potřebovat nebudeme.

Protože v úlohách k procvičení budete potřebovat napsat kód pro opakování (neboli smyčku, cyklus), bude se vám hodit instrukce loop, která vám tuto práci usnadní.

3.8.1 Instrukce loop imm

Instrukce loop slouží k opakování kódu.

Tato instrukce slouží k opakování kódu. Dekrementuje registr ecx a pokud je výsledek nenulový, skočí na adresu danou operandem. Pozor: Instrukce je omezena na blízké skoky ($\pm$128 bajtů), některé starší překladače nehlásí při překročení tohoto limitu chybu.

Příznaky: neovlivňuje

Obvyklé použití:

```
  mov ecx,<počet_opakování>
opakuj:
  ...zde libovolný kód...
  loop opakuj
```

Instrukce `loop <adresa>` je ekvivalentem ke dvojici instrukcí `dec ecx`, `jnz <adresa>`, ovšem s tím rozdílem, že loop neovlivňuje příznaky.

Shrnutí

V této kapitole jsme se podrobněji seznámili s registry a adresováním paměti. Zaměřili jsme se především na obecné registry využitelné při běžných výpočtech. Dozvěděli jsme se, že ačkoliv dnes již jsou jednotlivé registry mezi sebou většinou zaměnitelné, z historických důvodů a pro lepší přehlednost programů bývá doporučováno používat určité registry stále k těm účelům, ke kterým byly určeny v historických verzích procesorů.
Ve druhé části kapitoly jsme se naučili používat přímé a nepřímé adresování paměti. Na rozdíl od vyšších jazyků, kde výpočty efektivních adres v rámci nepřímého adresování zajišťuje překladač a programátor je této práce ušetřen, v assembleru to patří k základním dovednostem, které programátor musí ovládat.

Pojmy k zapamatování

- Registr
- Přímá hodnota
- Offset
- Nepřímé adresování
- Programový zásobník
- Paměťový model
- Pole a pointer

- Instrukce lea
- Instrukce loop

Kontrolní otázky

1. *K čemu slouží registr eip?*
2. *Které registry jsou indexové a co to znamená?*
3. *Jakým způsobem můžeme využít 32bitové registry, když potřebujeme jen 8 bitů?*
4. *Vysvětlete, co dělá operátor* `offset`.
5. *Vysvětlete, co dělá operátor* `[]`.
6. *K čemu slouží operátor* `ptr` *ve spojení s operátorem* `[]`*?*
7. *Popište smysl instrukce* `lea`*, uveďte vhodný příklad použití.*

Cvičení

1. Napište obdobu CRT funkce memcpy pro kopírování bloku paměti. Funkci implementujte tímto způsobem (pomocí pole):

```
void memcpy1(void *dest, void *src, int count) {
  char *_dest = (char*)dest;
  char *_src = (char*)src;
  for(int i=0; i<count; i++) {
    _dest[i] = _src[i];
  }
}
```

2. Napište opět CRT funkci memcpy, tentokrát však tímto způsobem (pomocí pointeru):

```
void memcpy2(void *dest, void *src, int count) {
  char *_dest = (char*)dest;
  char *_src = (char*)src;
  while(count>0) {
    *_dest = *_src;
    _dest++;
    _src++;
    count--;
  }
}
```

3. Napište program, který sečte hodnoty všech prvků v poli.

```
int c_soucet(int *pole, int delka) {
  int soucet = 0;
  for(int i=0; i<delka; i++) soucet += pole[i];
  return soucet;
}
```

4. Zatím neumíme větvit kód ve stylu příkazu if. Pomocí instrukce `jecxz` však umíme testovat nulovou hodnotu registru `ecx` a pomocí aritmetických instrukcí umíme dělat různé bitové operace. Napište kostru programu, který provede větvení programu ve stylu příkazu if podle toho, zda je hodnota registru `ecx` kladná/záporná/nulová.
   Nápověda: Může se vám hodit například znaménkový bitový posun `sar`.

Úkoly k textu

1. Napište funkci clearmem, která vynuluje blok paměti. V C++ tato funkce vypadá takto:

```
void clearmem(void *p, int count) {
  for(int i=0; i<count; i++) p[i] = 0;
}
```

2. Napište funkci indexarray, která vloží do každého prvku pole číslo jeho indexu. V C++ tato funkce vypadá takto:

```
void indexarray(int *array, int count) {
  for(int i=0; i<count; i++) array[i] = i;
}
```

3. Předchozí program upravte tak, aby sudé prvky nastavoval na kladnou a liché prvky na zápornou hodnotu dle indexu (tj. 0, –1, +2, –3, +4, atd.).

4. Předchozí program upravte tak, aby na hodnotu indexu nastavoval jen sudé prvky, zatímco liché prvky vynuluje. (První prvek pole je sudý.)

5. Napište CRT funkci `strcpy`, která okopíruje řetězec. Tato funkce se od `memcpy` liší tím, že délka není předem daná, ale určuje ji pozice (binární) nuly v řetězci. V C++ tato funkce vypadá takto:

```
void strcpy(char *dest, const char *src) {
  while(true) {
    *dest = *src;
    if(!*src) break;
    dest++;
    src++;
  }
}
```

   nebo srozumitelněji také takto:

```
void strcpy(char *dest, char *src) {
  int i = -1;
  do {
    i++;
    dest[i] = src[i];
  } while(src[i] != 0);
}
```

6. Ve cvičení 4 jste vytvořili program testující registr `ecx` vůči nule. Nyní program upravte tak, aby umožňoval testovat vůči jiným hodnotám. Nápověda: Na začátek přidejte odečet dvou hodnot, a pak porovnávejte opět s nulou (je-li druhé číslo větší, pak je výsledek odečtu záporný). Další práci si pak ale musíte dát s řešením aritmetického přetečení (je-li první číslo hodně malé a druhé číslo hodně velké, tak může být výsledek jejich odečtu kladný).

Řešení

1. Jedná se jednu ze základních úloh na procvičení adresování paměti. Řešení je poměrně snadné – vzpomeneme si přitom na správné používání registrů: použijeme `ecx` jako počítadlo opakování, `esi` jako adresu čtení, `edi` jako adresu zápisu a `ebx` jako index pole. (Fungovalo by to i s jinými registry, toto je jen konvence pro přehlednost.)

```
void asm_memcpy1(void *dest, void *src, int count) {
  _asm {
    mov esi,src
    mov edi,dest
    mov ecx,count
    mov ebx,0
opakuj:
    mov al,[esi+ebx]
    mov [edi+ebx],al
    inc ebx
    loop opakuj
  }
}
```

Drobnou úpravou lze dosáhnout i řešení se třemi registry – `ecx` totiž můžeme použít jako počítadlo a zároveň index v poli:

```
void asm_memcpy1b(void *dest, void *src, int count) {
  _asm {
    mov esi,src
    mov edi,dest
    mov ecx,count
    dec esi
    dec edi
opakuj:
    mov al,[esi+ecx]
    mov [edi+ecx],al
    loop opakuj
  }
}
```

2. Druhá varianta zadání vede k jinému zajímavému řešení, kdy indexy do pole nepoužíváme vůbec a namísto toho přímo posouváme registry čtení a zápisu.

```
void asm_memcpy2(void *dest, void *src, int count) {
  _asm {
    mov esi,src
    mov edi,dest
    mov ecx,count
opakuj:
    mov al,[esi]
    inc esi
    mov [edi],al
    inc edi
    loop opakuj
  }
}
```

3. Úloha má opět mnoho variant řešení. Ukážeme si jednu z nich:

```
int asm_soucet(int *pole, int delka) {
  _asm {
    mov esi,pole
    mov ecx,delka
    mov eax,0
opakuj:
    add eax,[esi]
```

```
        add esi,4
        loop opakuj
    }
}
```

Poznámka: Výsledek zde vracíme přímo v registru eax. Mohli bychom také použít příkaz C/C++ return, ale když jej vynecháme, tak překladač automaticky předpokládá, že výsledek pro vrácení je připraven v registru eax. Na to samozřejmě myslíme už v návrhu použití registrů – do registru eax průběžně ukládáme mezisoučty, takže pro vrácení hodnoty pak již nemusíme nic dalšího dělat.

4. Můžeme použít `sar`.

```
jecxz nula     ;skok při nule
sar ecx,31     ;okopírujeme nejvyšší bit do celého registru
jecxz kladne   ;skok pro kladné (ecx=0)
;pro záporné neskáčeme (ecx=-1)
```

Taktéž je možno použít `shr`.

```
jecxz nula     ;skok při nule
shr ecx,31     ;posuneme nejvyšší bit do nejnižšího
dec ecx        ;nyní bude 0 pro záporné, -1 pro kladné
jecxz zaporne  ;skok pro záporné (ecx=0)
;pro kladné neskáčeme (ecx=-1)
```

4 Příznaky a podmíněné vykonávání kódu

Studijní cíle: V této kapitole se seznámíme s příznaky a ukážeme si, kdy a jak je můžeme používat. Naučíme se v assembleru psát programy obsahující podmíněné příkazy pro větvení a opakování částí výpočtu. Tyto programové konstrukty se v assembleru píší velmi odlišně od vyšších programovacích jazyků, proto jim je věnována celá samostatná kapitola.

Klíčová slova: příznak, flag, podmíněný skok

Potřebný čas: 95 minut.

4.1 Příznaky

4.1.1 Seznámení s příznaky

Konstrukce jazyka assembler, jak víme, odpovídá strojovému kódu příslušného procesoru. Na rozdíl od většiny vyšších jazyků jde tedy o jazyk, jehož podoba je silně ovlivněna faktem, že procesory jsou (většinou) řešeny hardwarově. Jednou ze základních vlastností hardwaru je, že tam lze velmi jednoduše realizovat paralelní vykonávání více jednoduchých operací. Jedním z programových konstruktů, které toto využívají, jsou příznaky (anglicky flags).

Každý příznak je jednobitový registr v procesoru. Všechny příznaky jsou pak obvykle sdruženy do jednoho většího registru, kde zaujímají jednotlivé jeho bity. Např. v procesorech x86 jsou všechny příznaky umístěny ve 32bitovém registru `eflags`. Tento můžeme libovolně číst, ale měnit jeho hodnotu lze jen omezeně – některé z příznaků jsou totiž z bezpečnostních důvodů chráněné a může je měnit jen operační systém.

Příznaky můžeme rozdělit do dvou skupin:

Řídicí příznaky ovlivňují (řídí) chování procesoru. Nastavit je až na výjimky může jen operační systém. Čtení jejich hodnot je povoleno všem, ovšem není to většinou k užitku.

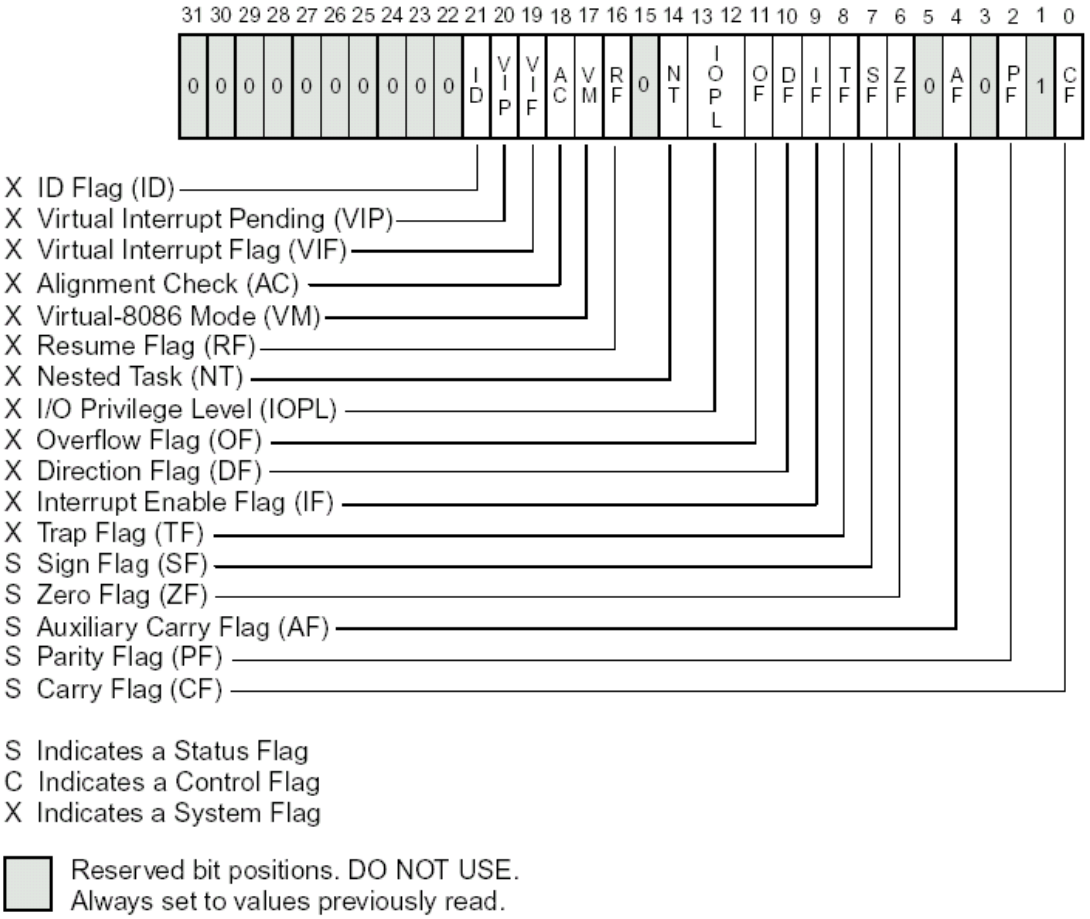
Aritmetické příznaky jsou naopak určeny primárně ke čtení, nikoliv k nastavování hodnot. Jejich hodnoty nastavuje sám procesor během vykonávání kódu.

Nás budou samozřejmě zajímat především příznaky aritmetické. Ty jsou nastavovány dle výsledku aritmetických i řady dalších instrukcí. Pomocí dalších instrukcí můžeme pak hodnoty těchto příznaků zjišťovat a řídit se jejich hodnotami v dalším výpočtu pomocí instrukcí podmíněných skoků (a nejen jich). Na některé z těchto příznaků a instrukcí jsme již dříve narazili, např. hned v první kapitole u příkladu výpočtu absolutní hodnoty. Ukažme si znovu tento příklad:

```
int abs(int hodnota) {
  _asm {
    mov eax,hodnota
    cmp eax,0
    jge skip
    neg eax
skip:
  }
}
```

Algoritmus podmíněného vykonání kódu je vždy dvoukrokový: Nejprve provedeme nějakou instrukci, která mj. nastaví aritmetické příznaky. V našem příkladě je to instrukce `cmp`. Pak instrukcí podmíněného skoku otestujeme vybraný příznak či příznaky. V případě, že testované příznaky jsou nastaveny na jedničku, procesor provede skok na adresu uvedenou v operandu. V opačném případě skok neproběhne a výpočet pokračuje normálním způsobem na následující instrukci.

Seznam všech příznaků procesoru x86 a struktura registru eflags je zobrazena na obrázku 5. Popis, jak jednotlivé instrukce ovlivňují jednotlivé příznaky je v tabulce 4 a tabula 5 pro úplnost uvádí seznam instrukcí, které příznaky nijak neovlivňují (i když některé z nich příznaky čtou).



Obrázek 5: Příznaky procesoru řady x86 (Pentium 3). [IA32]

4.1.2 Aritmetické příznaky

Aritmetické příznaky, jak jejich název napovídá, jsou ovlivněny výsledky aritmetických operací. Nyní se s každým z nich podrobně seznámíme.

CF (Carry Flag) je nastaven na jedničku, když dojde k přenosu z nejvyššího bitu. Při sčítání (add) je to tehdy, když je výsledek větší než největší možné číslo. Při odčítání (sub, cmp) je to tehdy, když je výsledek naopak menší než nejmenší možné číslo. CF se nastavuje také při dalších instrukcích, např. bitových posunech, se kterými se seznámíme později. Pozor však na to, že instrukce `inc` a `dec` tento příznak neovlivňují(!). Tento příznak má ještě druhou značku B (Below), protože při porovnávání (cmp) signalizuje případ, že první hodnota je menší než druhá (neznaménkově).

Příklady (sčítání a odčítání 8bitových hodnot):
2+5=7, CF=0
250+10=4, CF=1
5-2=3, CF=0
10-250=16, CF=1

PF (Parity Flag) je nastaven na jedničku při sudé paritě a na nulu při liché paritě. (Parita je počet jedničkových bitů.) Pozor na to, že tento příznak sleduje pouze dolních osm bitů.

Instrukce	OF	SF	ZF	AF	PF	CF
ADC	M	M	M	M	M	M
ADD	M	M	M	M	M	M
AND	0	M	M	-	M	0
BSF/BSR	?	?	M	?	?	?
BT/BTS/BTR/BTC	?	?	?	?	-	M
CMP	M	M	M	M	M	M
CMPS	M	M	M	M	M	M
DEC	M	M	M	M	M	
DIV	?	?	?	?	?	?
IDIV	?	?	?	?	?	?
IMUL	M	?	?	?	?	M
INC	M	M	M	M	M	
MUL	M	?	?	?	?	M
NEG	M	M	M	M	M	M
OR	0	M	M	?	M	0
POPF	R	R	R	R	R	R
RCL/RCR 1	M					M
RCL/RCR count	?					M
ROL/ROR 1	M					M
ROL/ROR count	?					M
SAHF		R	R	R	R	R
SAL/SAR/SHL/SHR 1	M	M	M	-	M	M
SAL/SAR/SHL/SHR n	?	M	M	-	M	M
SBB	M	M	M	M	M	M
SCAS	M	M	M	M	M	M
SHLD/SHRD	?	M	M	?	M	M
SUB	M	M	M	M	M	M
TEST	0	M	M	?	M	0
XOR	0	M	M	?	M	0

M = modifikuje dle výsledku, R = nastavuje celý registr
0/1 = nastavuje na 0/1, ? = nedefinovaná hodnota

Tabulka 4: Instrukce ovlivňující příznaky.

BSWAP, CALL, CBW, CWD, CDQ, CWDE, ENTER, INT, Jcc, JCXZ, JECXZ, JMP, LEA, LEAVE, LODS*, LOOP*, MOV*, NOP, NOT, POP* (kromě POPF), PUSH*, REP*, RET, SETcc, STOS*, XCHG

Tabulka 5: Instrukce neovlivňující příznaky.

Jeho smysl je dnes téměř nulový, měl však velké využití v minulosti, kdy se 8bitová parita používala pro sledování chyb při komunikaci.

Příklady:
0=00b: P=1
1=01b: P=0
2=10b: P=0
3=11b: P=1
257=100000001b: P=0, protože se započítává jen dolních 8 bitů

AF (Auxiliary Carry Flag) je doplňkový příznak přenosu. Nastavuje se na jedničku při přenosu z dolní poloviny bajtu (ze čtvrtého do pátého bitu). Tento příznak interně používají

instrukce pro práci s BCD čísly, čili opět jde o příznak, jehož praktické využití je velmi malé. (Nepleťte si však tento příznak s podmíněnými skoky s písmenem A – instrukce ja/jna/jae/jnae s ním nemají nic společného.)

ZF (Zero Flag) je příznak nuly. Nastavuje se na jedničku, když je výsledkem operace nula. Je vedle CF zřejmě nejužitečnějším příznakem. Tento příznak má ještě druhou značku E (Equal), protože při porovnávání hodnot (cmp) signalizuje rovnost operandů.

SF (Sign Flag) je příznak znaménka. Obsahuje vždy kopii nejvyššího bitu výsledku (tam je znaménko). SF je 1 pro záporná čísla a 0 pro kladná čísla a nulu.

OF (Overflow Flag) je příznak aritmetického přetečení. Jde o poměrně komplikovaný flag pomáhající při sledování přetečení u výpočtů se znaménkovými hodnotami. OF=1 když došlo k přenosu do nejvyššího bitu výsledku, ale nikoliv z nejvyššího bitu, nebo opačně. OF=0 ve všech ostatních případech.

Zjednodušeně lze OF popsat takto: OF=1 když v operaci došlo k přetečení znaménkové hodnoty - čili znaménkový výsledek je mimo rozsah platných hodnot.

Posledně uvedený příznak OF je zcela klíčový, stojí na něm totiž porovnávání znaménkových čísel a podmíněné skoky typu G–L (větší–menší), se kterými pracujeme velmi často. Programátor si naštěstí obvykle nemusí přesně pamatovat, jak tento příznak funguje, pokud jej umí používat. Rozdíl mezi CF a OF si ještě ukažme na dvou konkrétních příkladech obyčejného sčítání, viz tabulka 6.

operace	výsledek	CF	OF
255+1	0	1	0
127+1	128	0	1

Tabulka 6: Srovnání rozdílu mezi OF a CF při sčítání.

4.1.3 Řídicí příznaky

Řídicí příznaky až na výjimky v běžných programech nepotřebujeme. Někdy se však mohou hodit, proto si několik základních z nich představíme.

TF (Trap Flag) je příznak zastavení (doslova „pasti“). Používá se ke krokování programů (např. klávesami F10/F11 ve Visual Studiu). Je-li nastaven na 1, procesor se po každé instrukci přeruší. Obvykle je tedy tento příznak nulován, ale nespoléhejte na to.

IF (Interrupt Enable Flag) je příznak povolující přerušení. Je-li nulován, říkáme, že přerušení je maskováno, a všechna tzv. maskovatelná přerušení jsou ignorována. Tento příznak může nastavit jen operační systém.

DF (Direction Flag) je příznak směru posunu adres blokových instrukcí. Ve výchozím stavu je nulován. Význam tohoto příznaku si vysvětlíme později, jakmile se seznámíme s algoritmy a instrukcemi, které jej používají.

IOPL (I/O Privilege Level) je jediným 2bitovým příznakem. Jeho hodnota v rozsahu 0–3 určuje oprávnění procesu. Ve Windows mají všechny procesy oprávnění 3 (nejnižší), kromě jádra systému a některých ovladačů, které běží v úrovni oprávnění 0 (nejvyšší). Tento příznak nemůže přímo nastavit ani operační systém, podrobněji je systém oprávnění procesů popsán v [Kep07].

4.2 Podmíněné skoky

Instrukce podmíněných skoků se souborně označují `jcc`. Na rozdíl od instrukce skoku `jmp`, podmíněné skoky jsou provedeny jen při splnění (platnosti) určité podmínky. Podmínku ke skokové instrukci nezadáváme v operandu, ale je přímo součástí jména instrukce.[3] Kromě instrukce `jecxz`, kterou jsme si představili již v kapitole 2.9.1 na straně 17 (a její 16bitové sestry `jcxz`), se všechny ostatní podmínky vztahují k příznakům. Seznam všech instrukcí podmíněných skoků je uveden v tabulkách 7 a 8.

instrukce	příznaky	popis
ja/jnbe	(CF or ZF) = 0	Větší / není menší nebo roven
jae/jnb	CF=0	Větší nebo roven / není menší
jb/jnae	CF=1	Menší / není větší nebo roven
jbe/jna	(CF or ZF) = 1	Menší nebo roven / není větší
jc	CF = 1	Přenos
jnc	CF = 0	Není přenos
jz/je	ZF = 1	Nula / roven
jnz/jne	ZF = 0	Není nula / není roven
jnp/jpo	PF = 0	Není parita / parita lichá (odd)
jp/jpe	PF = 1	Parita / parita sudá (even)
jcxz	cx = 0	Registr cx je nulový
jecxz	ecx = 0	Registr ecx je nulový

Tabulka 7: Neznaménkové podmíněné skoky.

instrukce	příznaky	popis
jg/jnle	((SF xor OF) or ZF) = 0	Větší / není menší nebo roven
jge/jnl	(SF xor OF) = 0	Větší nebo roven / není menší
jl/jnge	(SF xor OF) = 1	Menší / není větší nebo roven
jle/jng	((SF xor OF) or ZF) = 0	Menší nebo roven / není větší
jno	OF = 0	Není přetečení
jo	OF = 1	Je přetečení
js	SF = 1	Je znaménko (je záporný)
jns	SF = 0	Není znaménko (je nezáporný)

Tabulka 8: Znaménkové podmíněné skoky.

Bez ohledu na složitost testů (nejvíc příznaků testujeme v instrukcích `jg`/`jng`) jsou všechny podmíněné skoky stejně rychlé. Obvykle se snažíme programy psát tak, aby v nich bylo co nejméně pojmenovaných návěští (labelů).

4.3 Další instrukce

Nyní se podíváme na několik dalších instrukcí, které se týkají příznaků. Instrukce `cmp`, `test` a `bt` slouží k porovnávání a testování hodnot. Instrukce `adc` a `sbb` přičítají resp. odečítají kromě druhého operandu i carry flag. Sada instrukcí `cl` a `st` explicitně mění hodnoty jednotlivých příznaků.

[3] Toto je specifická vlastnost procesorů řady x86. Na některých jiných procesorech jsou podmínky zadávány v operandech.

4.3.1 Instrukce cmp op1,op2

(Compare) Provede odečtení op1–op2 bez uložení výsledku. Dle hodnoty výsledku se nastaví příznaky. (Tato instrukce se chová stejně jako `sub`, ale nepřepisuje první operand výsledkem operace.)

Příznaky: Nastavuje AF, CF, OF, PF, SF, ZF dle výsledku operace.

4.3.2 Instrukce test op1,op2

Provede op1 and op2 bez uložení výsledku. Dle hodnoty výsledku se nastaví příznaky. (Tato instrukce se chová stejně jako `and`, ale nepřepisuje první operand výsledkem operace.)

Příznaky: Nastavuje CF, OF, PF, SF, ZF dle výsledku operace. Stav AF je nedefinován.

Poznámka: Pomocí `and` či `test` lze provést efektivní test, zda je hodnota registru nulová. Stačí daný registr dosadit za první i druhý operand instrukce, výsledkem je pak opět tatáž hodnota a zároveň je nastaven příznak ZF dle výsledku, tedy podle hodnoty tohoto registru. (Tato operace hodnotu registru nezmění, proto je možno použít `and` i `test`. Naopak pro proměnné to takto udělat nejde.)

4.3.3 Instrukce bt op1,op2

Hodnotu bitu číslo op2 v op1 přenese do CF. (Bity jsou číslovány od nuly.)

Příznaky: nastavuje CF dle výsledku operace.

4.3.4 Instrukce adc op1,op2

(Add with carry) Součet op1+op2+CF uloží do op1. Nastavuje všechny příznaky, stejně jako `add`. Používá se pro implementaci vícebitových operací.

Příznaky: Nastavuje AF, CF, OF, SF, PF, ZF dle výsledku operace.

4.3.5 Instrukce sbb op1,op2

(Subtract with borrow) Rozdíl op1–op2–CF uloží do op1. Nastavuje všechny příznaky, stejně jako `sub`. Používá se pro implementaci vícebitových operací.

Příznaky: Nastavuje AF, CF, OF, SF, PF, ZF dle výsledku operace.

4.3.6 Instrukce pro explicitní změnu příznaků

Instrukce pro explicitní změnu příznaků můžeme použít pro nastavení některých vybraných příznaků samostatně. Všechny tyto instrukce shrnuje tabulka 9. Významy zkratek jsou clear (cl), set (st), complement (cm).

Jak vidíme v tabulce, pouze příznaky CF, DF a IF umožňují explicitní nastavení.

4.3.7 Instrukce neměnící příznaky

Řídicí instrukce příznaky nikdy nemění (pokud nedojde k nějaké chybě). Z těch, které už známe, to jsou všechny varianty či typy přiřazení, skoků, volání podprogramů a návratů z něj.

instrukce	popis
clc	Nuluje CF (CF = 0)
stc	Nastaví CF (CF = 1)
cmc	Invertuje CF (CF = 1 - CF)
cld	Nuluje DF (DF = 0)
std	Nastaví DF (DF = 1)
cli	Nuluje IF (IF = 0)
sti	Nastaví IF (IF = 1)

Tabulka 9: Instrukce pro explicitní změnu příznaků.

4.3.8 Bitové rotace

Instrukce pro bitové rotace `rol` (rotate left) a `ror` (rotate right) fungují podobně jako instrukce bitových posunů `shl` a `shr`, jenže bity rotují „kolem dokola“ a nikam se neztrácejí. Proto například jednorázovou nebo i postupnou rotací o 32 bitů máme vždy zpět původní hodnotu. Stejně jako u bitových posunů se rotující bit dostane také do CF.

Další rotační instrukce jsou `rcl` (rotate with carry left) a `rcr` (rotate with carry right), které fungují obdobně, ale k registru přidávají ještě CF a rotují místo 8/16/32 bitů 9/17/33 bitů. Tyto instrukce umožňují snadněji provádět některé operace s CF, které by jinak vyžadovaly delší kód.

4.4 Vícebitové operace

Pomocí příznaku přenosu CF (carry) lze snadno implementovat vícebitové operace. Zkusíme pro procvičení naimplementovat 32bitový součet pomocí 16bitových registrů. Použijeme při tom výše uvedenou instrukci `adc`.

```
mov bx,word ptr A
add bx,word ptr B
mov word ptr C,bx
mov bx,word ptr A+2
adc bx,word ptr B+2
mov word ptr C+2,bx
```

Všimněte si také, že hodnotu CF nastavenou ve druhém řádku použijeme až v 5.řádku. Instrukce `mov` totiž příznaky neovlivňuje, takže toho zde s výhodou využijeme.

4.5 Vyhýbání se podmíněným instrukcím (optimalizace)

Zajímá-li nás rychlost výpočtu, upřednostňujeme případ neplatné podmínky (neskočit je rychlejší než skočit) nebo se podmíněným skokům přímo vyhýbáme. Pro příklad si uveďme kód, který zjistí, zda je hodnota v registru `eax` sudá, či lichá, a nastaví dle toho registr `ebx` na jedničku (lichá), či nulu (sudá). Základní implementace může vypadat například takto:

```
  mov ebx,0
  test eax,1
  jz konec
  mov ebx,1
konec:
```

Tento algoritmus však lze přepsat tak, aby nepoužíval podmíněný skok. Přitom můžeme zavést i další optimalizace.

```
  xor ebx,ebx
  shr eax,1
  adc ebx,0
  rcl eax,1
```

První řádek pomocí instrukce `xor` vynuluje registr. Nulování pomocí `xor` je rychlé a zabírá i méně paměti (ušetříme 4 bajty s číslem). Potom posuneme `eax` doprava, čímž dojde k přenosu nejnižšího bitu do CF. Pak můžeme přičíst CF k `ebx` a na závěr pomocí rotace doleva přes carry vrátíme `eax` do původního stavu. Navíc, nebude-li hodnota `eax` již potřeba, můžeme poslední dva řádky nahradit elegantním **rcl ebx**,1. Rotace a posuny jsou lepší než sčítání, protože číslo ve druhém operandu nezabírá 4 bajty.

4.6 Větvení kódu

Jak už víme, větvení kódu, které ve vyšších jazycích provádíme především pomocí příkazu `if`, v assembleru provádíme pomocí podmíněných skoků. Zastavme se nyní podrobněji u tohoto tématu.

4.6.1 Konstrukce if–then–else

Příkaz `if` se ve vyšších jazycích používá ve dvou variantách:

```
if A then B
if A then B else C
```

Nejprve se vyhodnotí predikát (podmínka) A. Pokud je výsledek true, provede se B. Je-li výsledek false, provede se C. Blok kódu C je přitom nepovinný. V assembleru se tato konstrukce implementuje dle této šablony:

```
  ... A ...
  jcc neplati
  ... B ...
  jmp konec
neplati:
  ... C ...
konec:
```

Zkrácená verze bez bloku C pak vypadá takto:

```
  ... A ...
  jcc neplati
  ... B ...
neplati:
```

Použití šablony si ukážeme na následujícím příkladu: Funkce akceptuje dvě čísla `a` a `b` a vypíše pozdrav, když `a` je kladné a `b` je sudé. Zvládnete-li vyřešit tuto úlohu sami, zkuste to!

```
void testab_c(int a, int b, const char *pozdrav) {
  if(a>0 && b%2==0) printf(pozdrav);
}
```

Vzorové řešení následuje.

```
void testab(int a, int b, const char *pozdrav) {
  const char *pozdrav = "Ahoj";
  _asm {
```

```
    cmp a,0
    jng konec
    test b,1
    jnz konec
    push pozdrav
    call dword ptr printf
    add esp,4
konec:
  }
}
```

4.6.2 Konstrukce switch–case–break

Příkaz `switch` provede jeden z bloků kódu dle hodnoty výrazu. Tento příkaz se jmenuje i zapisuje v různých jazycích různě, proto se tentokrát budeme odkazovat přímo na syntaxi C/C++.

```
switch(A) {
case B1:
   ...C1...
  break;
case B2:
   ...C2...
  break;
default:
}
```

Jednou z možností, jak tuto konstrukci realizovat, je použít stejný postup jako u příkazu `if`. Tuto variantu nebudeme dále zkoumat, je naprosto triviální.

Další možností je implementace pomocí tabulky skoků. Tuto variantu také nyní nebudeme zkoumat, protože v inline assembleru ji nelze jednoduše realizovat (vyžaduje totiž pomocné pole, které musíme deklarovat v assembleru, což ale inline assembler neumí).

4.7 Cykly

V této sekci se podíváme, jak lze v assembleru realizovat cykly. Vycházíme přitom z předpokladu, že studenti znají nějaký vyšší programovací jazyk a není důvod v assembleru nepoužít některé již naučené vzory programování, proto se podíváme, jak v assembleru realizujeme cykly v podobě klasických vysokoúrovňových příkazů repeat, while, for a do–while.

Některé tyto příkazy lze v assembleru výhodně realizvoat také pomocí maker, tuto alternativu budeme diskutovat v kapitole A.1 na straně 76.

4.7.1 Konstrukce repeat

Příkaz repeat se ve vyšších jazycích používá pro opakování části kódu, kde počet opakování je předem daný. Tvar pseudokódu je následující:

```
repeat A: B
```

A je číslo, počet opakování. Může to být jak konstanta, tak hodnota vypočítaná, ale během vykonávání kódu, který se opakuje, se již nedá změnit. B je blok kódu.

V assembleru toto implementujeme nejsnáze pomocí instrukce `loop`:

```
  mov ecx,A
zacatek:
  ...B...
  loop zacatek
```

Jak vidíme, uvnitř opakovaného bloku máme v registru ECX přístupné počítadlo opakování, které má při jednotlivých průchodech hodnotu A až 1. Po skončení opakování platí $ecx = 0$.

Alternativně lze použít opis instrukce `loop`, kdy pomocí kombinace `dec` a `jnz` provedeme stejnou operaci.

```
  mov ecx,A
zacatek:
  ...B...
  dec ecx
  jnz zacatek
```

Výhodou tohoto řešení je, že místo registru `ecx` můžeme použít libovolný, který máme volný. Je-li počet opakování malý, může to být i některý menší registr. Naopak u instrukce `loop` není volba registru možná.

4.7.2 Konstrukce do–while

Cyklus typu do–while se liší od repeat tím, že počet opakování nemusí být předem znám. Příkaz má tento tvar:

```
do B while A
```

Kde B je opět blok kódu, který se má opakovat, zatímco A je podmínka, jejíž platnost je testována na konci každého průchodu cyklem. Cyklus tedy proběhne minimálně jednou na začátku, a pak probíhá znovu tak dlouho, dokud platí podmínka A.

Realizace v assembleru je stejná jako druhá varianta repeat – na konci cyklu testujeme libovolnou podmínku a dokud platí, podmíněným skokem se vracíme zpět na začátek. Samotné testování podmínky je přitom analogické řešení příkazu if, viz výše. Můžeme tedy testovat i složenou podmínku apod.

4.7.3 Konstrukce while

Cyklus typu while je obdobou předchozího, tentokrát se však platnost podmínky testuje na začátku cyklu. Příkaz má tvar:

```
while A: B
```

Až na pořadí je příkaz stejný jako do–while. Tím, že podmínku testujeme na začátku **před** provedením B, je docíleno toho, že při neplatnosti podmínky se cyklus neprovede ani jednou.

Implementace je obdobou příkazu if: Nejprve otestujeme podmínku a pokud neplatí, přeskočíme blok B. Použijeme proto stejnou šablonu, pouze na konec bloku B přidáme nepodmíněný skok na začátek příkazu.

```
zacatek:
  ... A ...
  jcc konec
  ... B ...
  jmp zacatek
konec:
```

Na jednu věc je třeba dát pozor: Podmínku musíme testovat obráceně, protože pokud bude splněn test, skáčeme na konec. Tzn. musíme test postavit tak, aby podmínka v assembleru byla splněna právě tehdy, když není splněna podmínka A, a obráceně.

4.7.4 Konstrukce for

Cyklus typu for je zobecněním while či do–while (podle toho, zda je prováděn test před prvním průchodem). Kód tohoto typu není pro assembler příliš vhodný, protože není zcela přehledný a nepřináší žádný velký přínos oproti jednodušším příkazům výše. Přesto se na něj podíváme.

Příkaz for má v různých jazycích různou podobu, my použijeme základní model:

```
for A = D to H
  B
end
```

B je opět blok kódu, který se bude opakovat (tělo cyklu). A je počítadlo cyklu, P je počáteční hodnota A, H je horní hodnota A. A je nejprve inicializováno na D, potom se cyklus opakuje tak dlouho, než je A rovno H.

```
  mov A,D
zacatek:
  cmp A,H
  jg konec
  ... B ...
  inc A
  jmp zacatek
konec:
```

Tento vzor můžeme libovolně upravit, např. pro odečítání počítadla nebo úplně volné úpravy počítadla po každém cyklu.

Shrnutí

V této kapitole jsme se věnovali příznakům. Aritmetické příznaky, které jsou jejich hlavní skupinou, slouží k implementaci rozhodovacích příkazů. Příznaky jsou (až na pár výjimek) jediným způsobem, jak lze větvit kód. Jelikož assembler nemá ani nejběžnější příkazy strukturovaných jazyků jako if nebo while, je v praxi potřeba používat příznaky poměrně často. Představili jsme si proto instrukce podmíněných skoků, které s příznaky pracují, a také jsme probrali realizaci běžných konstrukcí pro rozhodovací a opakovací příkazy známé ze strukturovaných jazyků.
Další skupina, systémové příznaky, slouží k nastavení činnosti procesoru. Jejich použití v praxi je daleko méně časté než u příznaků aritmetických.

Pojmy k zapamatování

- řídící příznaky
- aritmetické příznaky
- CF (carry flag)
- PF (parity flag)
- AF (auxiliary carry flag)
- ZF (zero flag)
- SF (sign flag)
- OF (overflow flag)
- TF (trap flag)

- IF (interrupt enable flag)
- DF (direction flag)
- IOPL (I/O privilege level)
- podmíněný skok
- větvení kódu
- opakování kódu (cyklus)

Kontrolní otázky

1. *Které dvě kategorie příznaků rozlišujeme? Charakterizujte je.*

Cvičení

1. Napište funkci strlen zjišťující délku řetězce. (V jazyce C/C++ je délka řetězce dána pozicí binární nuly, která jej ukončuje. Do délky není tato nula zahrnuta.)

   ```
   int strlen(const char *text) {
     int len=0;
     while(text[len]) len++;
     return len;
   }
   ```

   Jedná se o velmi jednoduchou funkci, takže se snažte najít nějaké elegantní řešení.
2. Napište funkci, která zjistí, zda je v poli víc kladných nebo záporných čísel. Funkce bude vracet: -1 když převládají záporná, 0 když je jich stejně, +1 když převládají kladná.

   ```
   int KladnaZaporna_C(int *pole, int delka) {
     int pocitadlo = 0;
     for(int i=0; i<delka; i++) {
       if(pole[i]>0) pocitadlo++;
       else if(pole[i]<0) pocitadlo--;
     }
     if(pocitadlo>0) return +1;
     if(pocitadlo<0) return -1;
     return 0;
   }
   ```

3. Stejnou úlohu řešte bez použití porovnávací instrukce `cmp` a také bez odčítání.
4. Instrukce `cdq` provádí znaménkové rozšíření `eax` na `edx:eax`. Jak byste ji nahradili pomocí jiných instrukcí s využitím příznaků?

Úkoly k textu

1. Napište funkci, která v řetězci změní všechna velká písmena na malá. Funkce bude měnit text v místě (čili nebude alokovat novou paměť). Písmena abecedy jsou `'A'` až `'Z'` a `'a'` až `'z'`. Znaky v apostrofech jsou literály, které můžete používat jako čísla. (Toto platí pro C/C++ i assembler.) České znaky neuvažujte, řešte pouze anglickou abecedu.

   ```
   void tolower(char *text) {
     for(int i=0; text[i]; i++) {
       if(text[i]>='A' && text[i]<='Z') text[i] += 'a'-'A';
     }
   }
   ```

2. Napište obdobnou funkci jako v předchozím úkolu, ale znaky měňte tak, že každé sudé písmeno abecedy bude velké a každé liché bude malé.

3. Sestavte šablonu pro příkaz `for`. (Vaším úkolem je napsat vzorový příklad v assembleru, na kterém ukážete, jak lze zapsat kód ekvivalentní konstrukci `for` známé z jazyků C/C++.)

4. Sestavte šablonu pro příkaz `do–while`. (Je to obdoba cyklu typu `while`, kde se podmínka testuje na konci cyklu, tj. poprvé až po prvním průchodu.)

5. Napište funkce pro součet, rozdíl, součin a podíl dvou znaménkových 64bitových čísel. Nápověda: Použijte klasické algoritmy ze základní školy („sčítání pod sebou“ atd.) s 4bajtovou číselnou soustavou (tj. každé 4 bajty představují jednu cifru čísla a vy máte udělat triviální dvojciferný součet, rozdíl, součin a podíl).

6. Napište funkci pro součin dvou 1024bitových neznaménkových čísel. Vstupními parametry budou tři pointery – dva ukazující na 1024bitové činitele a třetí ukazující na 2048bitový buffer pro výsledek. Pro jednoduchost se můžete omezit na pouze neznaménková čísla.
   Nápověda: Úlohu lze řešit klasickým algoritmem ze základní školy „násobení pod sebou“. Řešení v C++ je zde jiné než v assembleru, neboť v C++ nemůžeme používat CF a rozšíření výsledku do 2 registrů. Následující vzorové řešení ukázka je tedy jen v pseudo–C++.

```
void Multiply(unsigned *a, unsigned *b, unsigned *result) {
  for(int i=0; i<128; i++) {
    for(int j=0; j<128; j++) {
      (unsigned long long)result[i+j] = b[i] * a[j];
    }
  }
}
```

7. Napište CRT funkci memmove, která je obdobou memcpy, ale funguje i v případě překrývajících se bufferů. Tato funkce tedy má řešit i případ, kdy cílový buffer začíná za začátkem zdrojového a překrývá jeho konec. V tom případě totiž memcpy přepíše konec zdrojového bufferu daty z jeho začátku. Řešení je však jednoduché: Nejprve zjistíme, zda je cílový buffer za, nebo před zdrojovým. Pokud bude za ním, budeme kopírovat pozpátku.

8. Napište funkci strmove, která bude mít stejný význam jako memmove, ale bude fungovat pro řetězce. Tato funkce tedy v případě, že cílový buffer je před zdrojovým, funguje stejně jako strcpy. V opačném případě ale nejprve zjistí délku řetězce, a pak jej okopíruje pozpátku.

9. Jak víme, rozšíření hodnoty na větší počet bitů u znaménkových čísel provádíme instrukcí `movsx`. Tato instrukce se však objevila až v procesoru 386, starší procesory měly jen dvě instrukce omezené na konkrétní dvojice registrů (`cbw` rozšiřuje `ax⟵al`, `cwd` rozšiřuje `dx:ax⟵al`. Nyní si ale představme, že žádnou takovou pomocnou instrukci nemáme. Jakým náhradním způsobem provedete rozšíření 16bitové znaménkové hodnoty na 32bitovou? Napište příslušný program. Nápověda: Mohou se hodit bitové posuny.

10. Napište funkci pro součet čísel v poli. Prvky pole budou typu `int`, výsledek typu `long long` (64bitový se znaménkem). V této funkci využijete aritmetické instrukce pracující s příznakem přenosu. (Toto jste už jednou dělali, nyní ale musíte výsledek spočítat bez ořezání tj. 64bitově.)

```
long long soucetpole(int *pole, int delka);
```

Řešení

1. Jedná se o triviální funkci, která jednoduše najde nulu v poli znaků a vrací její index. Řešení může vypadat třeba takto:

```
int strlen(const char *text) {
  _asm {
    mov eax,text
    mov ecx,0
dalsi:
    cmp byte ptr[eax+ecx],0
    jz konec
    inc ecx
    jmp dalsi
konec:
    mov eax,ecx
  }
}
```

   Vzpomeňme si však, že je vhodné vyhýbat se podmíněným skokům. Právě zde lze jeden ze skoků ušetřit, stejně tak můžeme ušetřit jeden registr, protože k posouvání po znacích v poli nám bude stačit i jeden.

```
int strlen(const char *text) {
  _asm {
    mov eax,text
    dec eax       ;posuneme se jeden znak před začátek textu
dalsi:
    inc eax       ;posuneme se na další znak
    cmp byte ptr[eax],0
    jnz dalsi
    sub eax,text ;máme adresu konce, odečteme od ní adresu začátku
  }
}
```

2. Řešení je poměrně jednoduché. Využijeme také toho, že v assembleru lze test na kladnou–zápornou–nulovou hodnotu provést pomocí jediného použití `cmp` (zatímco v C/C++ to jsou dva příkazy `if`).

```
int KladnaZaporna(int *pole, int delka) {
  _asm {
    mov esi, pole
    mov ecx, delka
    mov eax, 0
opakuj:
    cmp dword ptr[esi],0
    je dalsi
    jg kladne
    ;záporné
    dec eax
    jmp dalsi
    ;kladné
```

```
kladne:
    inc eax
dalsi:
    add esi,4
    loop opakuj

    cmp eax,0
    je konec
    mov eax,1
    jg konec
    mov eax,-1
konec:
  }
}
```

3. Tuto úlohu lze samozřejmě řešit různými způsoby. Například lze využít příznaku znaménka SF. Instrukci `cmp` můžeme nahradit např. instrukcí `test` či `and`.

```
int KladnaZaporna2(int *pole, int delka) {
  _asm {
    mov esi, pole
    mov ecx, delka
    mov eax, 0
opakuj:
    mov bl,[esi]
    and bl,bl
    je dalsi
    jns kladne
    ;záporné
    dec eax
    jmp dalsi
    ;kladné
kladne:
    inc eax
dalsi:
    add esi,4
    loop opakuj

    and eax,eax
    je konec
    mov eax,1
    jns konec
    mov eax,-1
konec:
  }
}
```

4. Hodnotu nejvyššího bitu (tj. znaménko) nejprve přesuneme do CF, a potom pomocí odečtení 0–CF jej přeneseme do všech bitů registru.

```
    xor edx,edx
    bt eax,31
    sbb edx,0
```

5 Zásobník a podprogramy

Studijní cíle: V této kapitole se naučíme používat programový zásobník a externí assembler, čímž se dostaneme k celé řadě dalších prostředků v inline assembleru nedostupných. Téma programového zásobníku studujeme proto, abychom pochopili, jak v procesoru ve skutečnosti funguje volání podprogramů a rekurze a externí assembler je nástrojem, který k tomu budeme potřebovat. Pro studenty s větším zájmem o assembler bude externí assembler zajímavý i tím, že teprve s ním lze využít plnou škálu programových konstruktů tohoto jazyka. (Pokročilejší témata z externího assembleru přitom budou tématem i následující kapitoly.)

Klíčová slova: zásobník, podprogram, volání, externí assembler, MASM, parametry volání

Potřebný čas: 120 minut.

5.1 Programový zásobník

Jak již víte z kurzu funkcionálního programování, rekurze je velmi mocný programátorský nástroj. A rekurze je také jedním ze základních prvků procesorů, či přesněji každý procesor pracuje s programovým zásobníkem díky němuž umožňuje rekurzi poměrně snadno používat.

Zásobník je datová struktura typu LIFO (last in first out – co poslední uložíme, to první přečteme), kterou můžeme velmi snadno použít k uložení aktuálního stavu (operace push) a později k jeho obnovení (operace pop).

Procesor nativně používá jeden zásobník, máme proto k dispozici dvě jednoduché instrukce s jedním operandem: `push` a `pop`. Programový zásobník má podobu pole 4bajtových čísel a nic jiného, než 4bajtové hodnoty na něj ukládat nejde. Implementace zásobníku pomocí pole je přitom velmi jednoduchá. Procesor používá registr `esp` (který jsme nenápadně již zmínili v předchozí sekci) jako ukazatel na vrchol zásobníku. Hodnotou esp je tedy vždy adresa posledně uloženého prvku v zásobníku. Přečtení hodnoty na vrcholu zásobníku je tedy možné například takto: `mov eax,[esp]`. (Použili jsme zde poprvé hranatou závorku pro přečtení hodnoty z pointeru. Je to operátor dereference, byl již v tabulce 2 a podrobněji se s ním seznámíme dále v této kapitole.)

Registr esp ukazuje na vrchol zásobníku.

Dodejme ještě, že registr `esp` nemá nejnižší dva bity, jeho hodnota je tedy vždy násobkem čtyř. Toto je ale vlastnost, se kterou bychom neměli mít problémy, protože registr esp nebudeme nesystematicky měnit. Důležitější vlastností zásobníku je, že roste směrem dolů a obvykle je umístěn v paměti za globálními proměnnými programu. Pokud tedy budete na zásobník ukládat příliš mnoho hodnot, mohlo by dojít k přepsání globálních proměnných. Toto ale býval problém spíše v MS-DOSu, protože moderní procesory umějí s podporou operačního systému kontrolovat a zjistit naplnění zásobníku dříve, než dojde k jeho přetečení a porušení nějakých dat.

5.1.1 Instrukce `push` reg/mem/imm

Instrukce `push` uloží na zásobník hodnotu operandu: Nejprve se sníží hodnota `esp` o 4, pak se do `[esp]` uloží operand. Operandem může být přímá hodnota (imm), adresa paměti (mem) nebo registr (reg).

Jelikož je zásobník nativně 4bajtový, ukládáme na něj vždy 4bajtové hodnoty. Existují i 16bitové instrukce jako `push ax`, ale na zásobníku se vždy objeví 4 bajty.

Příznaky: neovlivňuje

5.1.2 Instrukce `pop` reg/mem

Instrukce pop vyzvedne ze zásobníku hodnotu a uloží ji do operandu: Hodnota z `[esp]` se uloží do operandu a hodnota esp se zvýší o 4. Operandem může být registr nebo adresa paměti.

Příznaky: neovlivňuje

5.1.3 Instrukce `call` reg/mem/imm

Instrukce `call` zavolá podprogram (proceduru či funkci): Uloží na zásobník aktuální hodnotu `eip` a předá řízení na adresu danou operandem.

Příznaky: neovlivňuje

5.1.4 Instrukce `ret` / `ret` imm

Instrukce `ret` bez operandu ukončí provádění podprogramu a vrátí řízení zpět do volajícího programu: Vyzvedne ze zásobníku hodnotu a uloží ji do `eip`.

Tato instrukce má ještě variantu s operandem, která po nastavení `eip` ještě zvýší esp o hodnotu uvedenou v operandu, čímž ze zásobníku uklidí parametry volání funkce. (Toto ale mohou používat jen jazyky, které nepodporují proměnlivý počet parametrů u procedur a funkcí, jako např. Turbo Pascal. Naopak C++ používá jen `ret` bez operandu.)

Příznaky: neovlivňuje

Průvodce studiem

Pozor! `ret` je ve skutečnosti pseudoinstrukce, tj. nejde o skutečnou instrukci, ale jen dohodnutý zápis tvářící se jako instrukce. Konkrétně u této pseudoinstrukce lze tento fakt zanedbat, neboť `ret` je pouze zkráceným zápisem instrukcí `retn` a `retf`, které se liší pouze typem paměťového modelu. Poslední písmeno názvu znamená n jako near pro blízký návrat, či f jako far pro vzdálený návrat. V našem programování ve Windows se `ret` vždy přeloží jako `retn`; druhá varianta by se mohla použít např. v MS–DOSu či pro návrat ze systémového volání, kdy je potřeba obnovit ze zásobníku také segmentový registr. Typ instrukce návratu musí přesně odpovídat typu volání – zajímavé přitom je, že blízké i vzdálené volání se provádí stejně pojmenovanou instrukcí `call`, protože typ volání pozná překladač dle hodnoty operandu této instrukce. Návratová instrukce ale žádný takový operand nemá a nepotřebuje, a tak musí mít dva názvy. Při běžném programování jsou všechna volání stejného typu a tak překladač sám doplní správnou variantu návratové instrukce.

5.1.5 Význam zásobníku při volání podprogramů

Zásobník má klíčový význam při volání podprogramů. Zavolání podprogramu funguje v těchto krocích:

1. Volající vloží na zásobník postupně všechny parametry (push).
2. Volající zavolá volaného (call).
3. Volaný vytvoří rámec volání podprogramu (tzv. „entry code“ či „prolog“):
   (a) Uloží ebp (push ebp) – ten doposud ukazoval na lokální proměnné volajícího.
   (b) Nastaví ebp na esp (mov ebp,esp)– takže bude ukazovat na vlastní lokální proměnné.

   (c) Sníží esp tak, aby vytvořil na zásobníku místo pro své lokální proměnné (sub esp,x).

4. Volaný uloží aktuální stavy všech pracovních registrů, které bude během výpočtu měnit (push).
5. Volaný vykoná vlastní kód podprogramu.
6. Volaný obnoví stavy všech pracovních registrů, které dříve uložil (pop).
7. Volaný uklidí rámec volání podprogramu (tzv. „exit code“ či epilog):
   (a) Vrátí esp na hodnotu ebp, čímž uklidí své lokální proměnné (mov esp,ebp).
   (b) Vyzvedne původní ebp ze zásobníku (pop ebp).
8. Volaný vrátí řízení do nadřazeného bloku (ret).
9. Volající ze zásobníku uklidí parametry volání (push nebo add esp).

Podoba zásobníku při volání je zobrazena na obrázku 6. Tento příklad platí pro případ volací konvence C. Přesná podoba zásobníku však závisí na několika faktorech (například také na použití lokálních proměnných, o kterých jsme zatím nemluvili), u některých se ještě zastavíme později.

...	
Parametr n	
...	
Parametr 1	⟵ ebp+8
Návratová adresa	
Původní ebp	⟵ ebp
První lokální proměnná	⟵ ebp–4
...	
Poslední lokální proměnná	⟵ esp
...	

Obrázek 6: Data na zásobníku v okamžiku volání podprogramu. [Kep07]

Další poznámky:

- V případě, že podprogram nepoužívá lokální proměnné, nemusí vytvářet rámec (stack frame), takže lze vynechat entry code a exit code.
- Povinnost ukládat registry, které chceme měnit, lze také převelet na volajícího. Ten ale nemůže vědět, které registry bude volaný potřebovat, takže musí při každém volání ukládat všechny. Proto je tato varianta méně efektivní.
- Při vkládání inline assembleru do cizího kódu nemusíme registry EAX, EBX, ECX, EDX, ESI a EDI ukládat, protože Visual C++ to dělá za nás. Je však potřeba myslet na to, že v jiném překladači se může inline assembler chovat jinak.
- Vstupní parametry volání lze ukládat na zásobník například postupně zleva doprava – tak to dělá např. Turbo Pascal. Parametry jsou pak ale na zásobníku pozpátku, protože zásobník roste dolů.
- Rovněž za úklid vstupních parametrů může odpovídat volající nebo volaný. Přednostně používáme model jazyka C, kde parametrů může být proměnlivý počet a jen volající ví, kolik jich ve skutečnosti bylo. Proto je po sobě uklízí on sám. Naopak Turbo Pascal používá k úklidu instrukci ret s operandem, takže za úklid zodpovídá volaný.

5.2 Cvičení s rekurzí

Nyní známe princip fungování zásobníku a můžeme si jej vyzkoušet na rekurzi. Mohli bychom jako úvodní příklad použít třeba výpočet faktoriálu, ale nevýhodou je, že faktoriály jsou velmi velká čísla – již pro poměrně malé vstupní hodnoty se výsledek nevejde do 32bitové proměnné. Proto místo násobení budeme čísla v řadě 1..n sčítat. Vypočítáme tak částečný součet číselné řady $1 + 2 + \cdots + n$.

V C++ je kód velmi jednoduchý:

```
int SoucetRady(int n) {
  return (n<=1) ? n : n+SoucetRady(n-1);
}
```

K realizaci rekurze v assembleru nám však ještě chybí nějaký nástroj pro ošetření limitního případu rekurze (v tomto případě ukončení rekurze pro $n \leq 1$). Nejjednodušší instrukcí, kterou k tomu lze použít, je podmíněný skok `jecxz`.

Řešení

Zde je vzorové řešení:

```
int SoucetRady(int n) {
  _asm {
    mov ecx,n
    jecxz konec  ;skok pro n==0
    sub ecx,1
    jecxz konec  ;skok pro n==1
    push ecx
    call SoucetRady
    pop ecx
    add n,eax
konec:
  }
  return n;
}
```

Projděte si tento program příkaz po příkazu. Měli byste jej snadno pochopit.

5.3 Externí assembler

Nyní musíme od výkladu látky odbočit k čistě technickému tématu. Látka probíraná zde a v následujících sekcích již v inline assembleru nejde dělat a je tedy nejvyšší čas přejít k externímu assembleru, což je termín označující práci s plnohodnotným překladačem assembleru a soubory neobsahujícími C/C++. Zatím jsme sice dávali přednost inline assembleru pro jeho jednoduchost, pouze externí assembler nám však umožní vytvářet v assembleru celé podprogramy a také proměnné, konstanty či makra.

5.3.1 Historické souvislosti

V další práci budeme opět používat překladač od Microsoftu, který jakožto výrobce operačního systému MS–DOS (a později MS–Windows) od začátku vyvíjel a poskytoval překladač MASM (Microsoft Macro Assembler) pro procesory x86 a svoje operační systémy. V assembleru je napsán i samotný operační systém MS-DOS a zejména v 80.letech, kdy počítače nebyly tak

rychlé jako dnes, patřil assembler mezi nejdůležitější programovací jazyky. (Na drtivé většině tehdejších počítačů, které dlouhou dobu úspěšně konkurovaly dnes převažujícím počítačům PC měl assembler dokonce zcela dominantní roli, na PC také patřil mezi hlavní jazyky, měl však i konkurenci.)

Na začátku 90.let 20.století se objevovaly i konkurenční překladače, v tehdejším Československu byl v oblibě zejména TASM (Turbo Assembler) firmy Borland, který byl dodáván jako součást Turbo Pascalu a Turbo C. Tento překladač se navíc snaží být kompatibilní s MASM, takže přechod programátorů mezi nimi dvěma je poměrně snadný. Během 90.let došlo na PC k masívnímu nástupu vyšších programovacích jazyků na jedné straně a vzniku několika nekomerčních projektů s cílem vytvořit konkurenční překladače. Zpočátku byl velmi oblíbený například NASM (Netwide Assembler). Microsoft se na konci 90.let opět zapojil do této „bitvy“, když uvolnil MASM k používání zdarma a začal jej dodávat spolu s Visual Studiem a Visual C++. Encyklopedie Wikipedia [Wiki] k tomu uvádí, že MASM od začátku byl a dodnes je číslem 1 mezi assemblery na platformě x86 a MS–DOS/Windows. (Odhad, kdo dnes zaujímá další přední místa v oblíbenosti, [Wiki], ani jiné dostupné zdroje neuvádějí. Můžeme tedy pouze z historických souvislostí odhadnout, že to jsou právě zmíněné NASM a/nebo TASM.)

MASM je vyšší assembler (high level), neboť poskytuje podporu maker a široké spektrum již zabudovaných vyšší příkazů, které napodobují chování vyšších programovacích jazyků (jako if, while, funkce s parametry apod.). Pro srovnání: Schopnosti maker jsou v MASM na daleko vyšší úrovni, než v jazyce C, ovšem třeba konkurenční NASM jako svou hlavní devizu uvádí právě ještě lepší schopnosti maker ve srovnání s MASM. Zmíněný NASM je také asi nejoblíbenějším konkurentem MASM, jeho výhodou je použitelnost i na jiných operačních systémech, zatímco MASM je možno spouštět jen ve Windows (a případně v emulátorech Win32, starší verze pak také v MS-DOSu a IBM OS/2).

Zájemci o historii najdou další poznámky o MASM v příloze B.

5.4 Microsoft Macro Assembler (MASM)

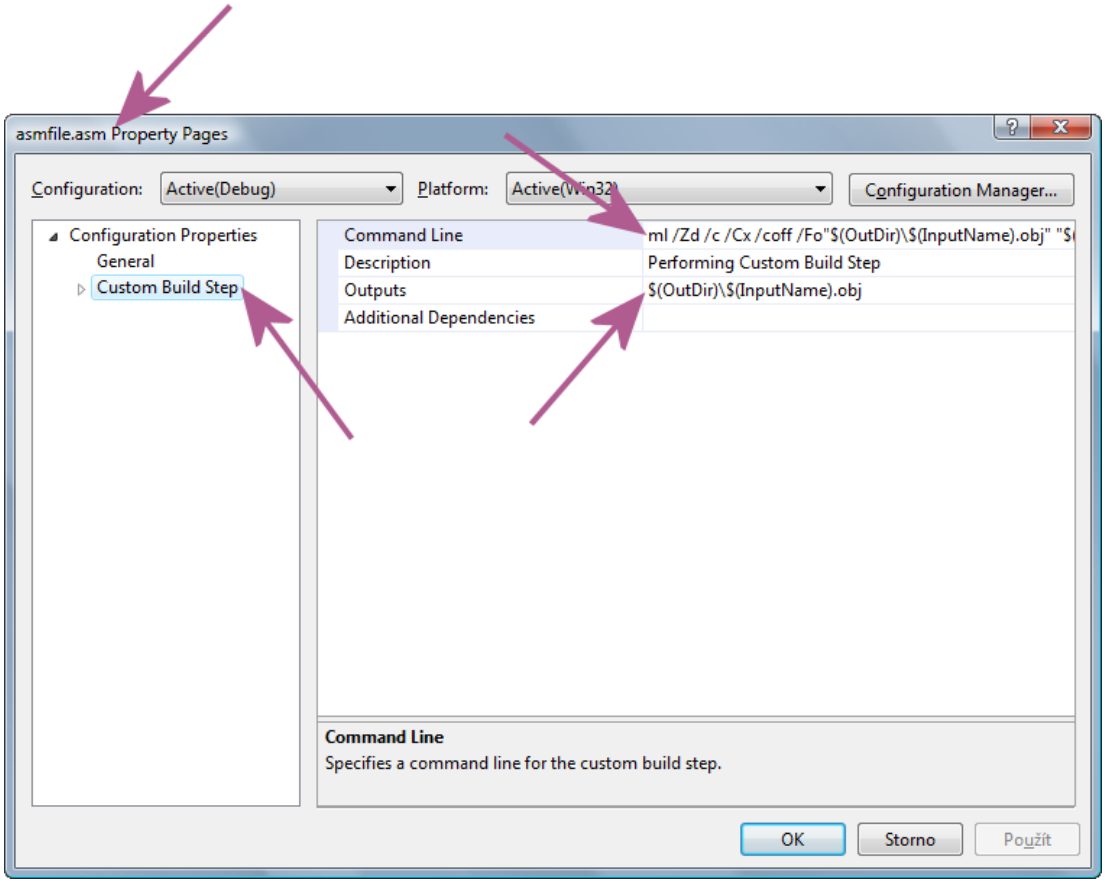
Jak bylo řečeno, MASM je nyní součástí Visual Studia a Visual C++. Najdete jej jako `ml.exe` v adresáři `VC/bin` a má stejné číslo verze jako příslušná edice překladače C/C++. Programy v externím assembleru se nejčastěji píší tak, že se založí projekt C/C++ a přidá se do něj jeden nebo více souborů s příponou `.asm`. K dispozici pak máme jak plnohodnotný assembler, tak knihovnu CRT (C Runtime library), která může být k užitku a její přítomnost nemá (resp. neměla by mít) žádný negativní dopad. Otázkou vlastních preferencí pak je, zda spouštěcí funkci `main()` napíšeme do assembleru, nebo ji necháme v C/C++. Popsaný postup je obecně používaný a platí i pro konkurenční prostředí, ne jen Visual Studio a MASM. V dobách MS-DOSu bývala obvyklá i tvorba celých programů bez CRT, ale vzhledem k tomu, jak složitý kód je nutný k inicializaci a spuštění programu ve Windows a jak složité je volat funkce Windows API, ve Windows se CRT v programech běžně nechává. CRT je navíc jako DLL soubor distribuováno přímo s Windows, není třeba jej mít u každého programu znovu. (Kód vykonávaný před a po volání `main()`, který normálně nevidíme, je obvykle také součástí CRT a obsahuje mj. inicializaci struktur používaných v CRT. Program při spuštění startuje právě zde a pak teprve volá naši funkci `main()`.) Zde je potřeba dát pozor na dvě důležité věci:

1. V projektu musí zůstat alespoň jeden C/C++ soubor, jinak linker nepřidá CRT knihovnu.
2. Visual Studio sice zná `.asm` soubory, má pro ně ikonu apod., ale nechce je automaticky kompilovat pomocí MASM. U každého `.asm` souboru je tedy potřeba nastavit kompilaci ručně.

Kompilace se ve Visual Studiu nastavuje takto: Přidejte do projektu nový soubor (v menu Project – Add New Item..., případně v Solution Exploreru kontextové menu a vybrat Add – New Item...), zvolte si nějaké jméno a přidejte příponu `.asm`. Potom na soubor klikněte v

Solution Exploreru a v kontextovém menu zvolte Properties. Otevře se okno vlastností souboru (Property Pages), kde je třeba nastavit následující (viz také obrázek 7):

- Na kartě General nastavit Tool na Custom Build Step.
- Na kartě Custom Build Step – General nastavit:
  - Command line na
    `ml /Zd /Zi /c /Cx /coff /Fo"$(OutDir)\$(InputName).obj" "$(InputPath)"`
  - Outputs na `$(OutDir)\$(InputName).obj`



Obrázek 7: Konfigurace překladu externím assemblerem (Visual Studio 2008)

Průvodce studiem

Pozor! V projektu nesmějí být dva stejně pojmenované soubory, protože by nebylo možno je oba zkompilovat do jednoho adresáře. Při kompilaci se ztrácí původní přípona, takže máte-li v projektu .cpp a .asm lišící se jen příponou, nebude to fungovat.

Příkazem `ml /?` je možno zobrazit seznam všech podporovaných přepínačů na příkazové řádce. Vysvětlení námi použitých je zde:

- `/Zd` připojí debug info s čísly řádků
- `/Zi` připojí debug info se jmény všech symbolů
- `/c` pouze kompiluje, ale nelinkuje
- `/Cx` u veřejných a externích symbolů rozlišuje velikost písmen
- `/coff` kompiluje do formátu COFF

- `/Fo` nastaví jméno výstupního souboru

Tzv. „debug info“ jsou ladicí informace, které dokáže pak Visual Studio využít k pohodlnému ladění. Čísla řádků slouží ke krokování v běžícím programu, tzv. „symboly“ jsou všechny pojmenované prvky ve zdrojovém kódu, které se překládají do kódu. Všechny tyto informace jsou uloženy přímo ve výstupním `.obj` souboru, jeho formát COFF je standardním formátem `.obj` souborů používaný ve Windows.

Průvodce studiem

Překladač assembleru ve Visual Studiu 2008 (ML.EXE verze 9.00) má chybu způsobující, že v okně Error List se nezobrazují správně všechny chyby nalezené ve zdrojovém kódu během překladu. Microsoft doporučuje použít ML.EXE z předchozí edice Visual Studia 2005 (ML.EXE verze 8.00), kde se chyba nevyskytuje. Ačkoliv je tento postup dosti nestandardní, je skutečně funkční: Používáte-li Visual Studio 2008, okopírujte si do něj tedy (obvykle do cesty ”c:\Program Files\Microsoft Visual Studio 9.0\VC\bin\ml.exe”) funkční soubor z Visual Studia 2005 (obvykle ”c:\Program Files\Microsoft Visual Studio 8\VC\bin\ml.exe”). [bug report autora tohoto učebního textu, viz Microsoft Connect feedback ID 331784]

V edici Visual C++ 2008 Express překladač assembleru dokonce vůbec není [Microsoft Connect feedback ID 291199], opět by však mohl pomoci výše uvedený postup. Microsoft k Express edici uvádí, že v následující verzi (pravděpodobně již v rámci SP1) tam překladač assembleru přidá.

5.5 Kostra programu v assembleru

V assembleru máme poměrně hodně možností, jak psát zdrojový kód. Pro začátek a hlavně pro účely výuky budeme používat tuto jednotnou kostru programu, do které budeme vpisovat naše programy – viz obrázek 8.

```
.486                    ;budeme používat procesor 486
.model flat,c           ;paměťový model flat, jazyk C

.const                  ;začátek konstant

;deklarace konstant

.data                   ;začátek inicializovaných dat

;deklarace proměnných

.data?                  ;začátek neinicializovaných dat

;deklarace proměnných

.code                   ;začátek kódu

;kód programu

end                     ;konec souboru
```

Obrázek 8: Kostra souboru v externím assembleru.

Na začátku souboru máme direktivu `.486` nastavující typ procesoru. Pojem direktiva znamená, že jde o nějaký pokyn pro překladač (tedy ne o instrukci). Číslo v této direktivě omezuje podporované instrukce na procesor typu 486 a starší, toto omezení se používá pro zajištění, že program bude minimálně na uvedeném procesoru fungovat. Podobných direktiv má MASM hodně, pro naše programování je `.486` vyhovující (v podstatě je to totéž jako `.386`), pro pokročilejší programátory se případně může hodit také direktiva `.686`, která povoluje až Pentium Pro/Pentium II (od verze MASM 6.12).

Další direktiva `.model flat,c` nastavuje paměťový model. Ve 32bitových operačních systémech na platformě x86 (Windows 9x/NT, Linux apod.) se používá jen model flat, takže jej zde uvedeme. Za čárkou je vyšší programovací jazyk, se kterým budeme assembler kombinovat. Toto lze i vynechat, ale nám se samozřejmě spolupráce s jazykem C bude hodit. Model jazyka C předepisuje assembleru především dvě věci:

- Způsob předávání parametrů při volání funkcí.
- Způsob dekorování veřejných jmen.

Předávání parametrů při volání funkcí si vysvětlíme později. Dekorování jmen je nutné k tomu, aby byl program v jazyce C vůbec přeložitelný, neboť historicky se používal především překlad přes assembler v textové formě. Každé symbolické jméno (proměnná, funkce atp.) tedy musí být nějak dekorováno, aby se nekrylo s klíčovým slovem assembleru, protože jinak by nešlo používat jména jako mov, eax apod. Překladače C na platformě x86 obvykle dekorují všechna jména přidáním čáry (podtržítka) na začátek názvu. Tím, že uvedeme `.model něco,c` v assembleru, zajistíme konzistentní pojmenovávání symbolů v assembleru a C souborech v rámci našeho projektu a můžeme je pak libovolně kombinovat.

Pozor! Je potřeba dodržet pořadí prvních dvou direktiv. MASM podle něj totiž určuje adresovací režim procesoru. Toto nemá žádný logický důvod, je to jen historická konvence: Nastavíme-li nejprve procesor na 386 či vyšší a pak teprve paměťový model, překladač emituje kód ve 32bitovém segmentu, v opačném případě překladač emituje kód v 16bitovém segmentu. Jelikož procesory 386 a vyšší podporují 16 i 32bitové instrukce v 16 i 32bitových kódových segmentech a není žádná direktiva, která by je rozlišila, MASM používá pro určení adresovacího režimu právě pořadí těchto dvou direktiv.

5.6 Globální proměnné a konstanty

Průvodce studiem

Pro jména symbolů platí následující pravidla: Prvním znakem může být písmeno (anglické abecedy) nebo jeden z těchto čtyř znaků: @, _, $, ?. Pro další znaky platí totéž a navíc tam lze použít i čísla. Maximální délka symbolů je 247 znaků (údaj berte spíše s rezervou, u MASM 5 platí jen 31 znaků a všeobecně u historických překladačů se používal princip ignorování konců jmen – to platí i pro překladače jiných jazyků (jako C apod.)). Jméno symbolu nesmí být stejné jako klíčové slovo assembleru (tj. jména instrukcí, registrů, direktiv apod.). Tato pravidla platí pro všechny symboly, ne jen proměnné.

5.6.1 Definice

Zdrojový soubor assembleru obsahuje v libovolném pořadí konstanty, proměnné a kód, které umisťujeme pomocí příslušných direktiv do svých segmentů. Kód je vždy v sekci `.code`, data jsou tří typů:

- V sekci `.const` jsou konstanty. Ve Windows je tato sekce hardwarově chráněna proti zápisu.

- V sekci `.data?` jsou proměnné s nulovou počáteční hodnotou. Tyto proměnné se neukládají do souboru s programem, pouze mají vymezenou paměť a při startu programu se tento úsek paměti vynuluje.
- V sekci `.data` jsou inicializované proměnné, které se ukládají do souboru s programem.

Na konci souboru vždy musí být direktiva `end`. (Tato direktiva nemá na začátku tečku. MASM má z historických důvodů řadu direktiv s tečkou a jiné pro změnu bez tečky na začátku, nehledejte v tom žádnou logiku.)

Globální proměnné a konstanty zakládáme uvedením jména na začátku řádku (všimněte si, že v assembleru se vždycky píše jako první nově definované jméno, v jazycích typu C je tomu naopak), za které píšeme definici.

```
p1    byte 7       ;1 bajt
p2    word 4000    ;2 bajty
p4    dword ?      ;4 bajty
```

V příkladu jsme založili tři proměnné o délce 1, 2 a 4 bajty. Každé proměnné přitom musíme uvést hodnotu nebo otazník jako vyjádření nedefinované hodnoty. Otazníky uvádíme jen v sekci `.data?`, zatímco sekce `.const` a `.data` musí mít všechny hodnoty uvedené. Na rozdíl od vyšších jazyků v assembleru takto definujeme i pole, kde hodnoty oddělujeme čárkami. U polí se může hodit speciální direktiva `dup`, která zopakuje definice hodnot (dup = duplicate). U této direktivy si všimněte poměrně dost netradiční způsob zápisu.

```
pole1 byte 1,2,3,4,5   ;pole 5 prvků
pole2 byte ?,?,?,?,?   ;pole 5 prvků bez uvedení hodnot
pole3 byte 5 dup (?)   ;pole 5 prvků pomocí dup
pole4 byte 2 dup (3 dup (4,5),6) ;pole hodnot 4,5,4,5,4,5,6,4,5,4,5,4,5,6
```

Stejným způsobem definujeme i řetězce, přičemž v assembleru se nedoplňuje koncová nula automaticky jako v jazyce C – doplníme ji tam tedy ručně. Assembler také nepoužívá escape kódy jako `\r\n` pro ukončení řádku, opět je třeba zadat ručně kódy 13,10. Příklad následuje, zároveň ukazuje možnost víceřádkových definic polí: Když je na konci řádku čárka, seznam hodnot pokračuje na dalším řádku.

```
t1 byte "Hello␣World",
        13,10,0
```

Ještě jedna poznámka k otazníkům v definicích hodnot: Jejich používání není vyloženě omezeno na sekci `.data?`, nicméně sekce `.data` a `.const` se ukládají do výsledného "exe" souboru, zatímco `.data?` ne. Použijete-li naopak v sekci `.data?` neotazníkové definice, ve skutečnosti tam budete mít nuly (paměť globálních proměnných se nuluje při startu programu).

Čísla lze zadávat v desítkové soustavě, v šestnáctkové soustavě s příponou `h` či ve dvojkové soustavě s příponou `b`. (Je možno použít i osmičkovou soustavu či úplně předefinovat výchozí číselnou soustavu, ale to se nebudeme učit.)

5.6.2 Export a import symbolů

Pojmy export a import označují vzájemné zpřístupnění symbolů mezi soubory v projektu. Export/import symbolů funguje nezávisle na programovacím jazyce, je to záležitost linkeru. Pro pochopení linkeru a principu linkování je dobré si alespoň stručně popsat, jak probíhá vytvoření "exe" souboru ze zdrojových textů programu.

Máme-li příslušné (kompatibilní) překladače, je teoreticky možné psát jednotlivé soubory programu v různých jazycích. Už víme, že můžeme libovolně kombinovat C, C++ a Assembler, totéž však platí o Pascalu či Fortranu a dalších klasických procedurálních jazycích. Podmínkou

je mít vhodný překladač. Překladač překládá programy po jednotlivých souborech a ke každému zdrojovému souboru „vyrobí“ samostatný výstup – je to vždy soubor se stejným jménem a příponou ”.obj” či ”.o”. Visual Studio tyto soubory dává do adresáře Debug či Release, kde potom najdete i hotový program. V okamžiku, kdy překladače příslušných programovacích jazyků vyrobily ”obj” soubory, přichází ke slovu linker – program, který všechny obj soubory propojí (odtud název linker – anglicky „spojovač“) neboli „slinkuje“. Výstupem linkeru je ”exe” soubor, případně jiný vhodný spustitelný soubor (např. v Linuxu se nevyrábí ”exe”, ale jiný typ souborů).

Na celém procesu kompilace si všimněte, že překladače jazyků se nedívají do ostatních souborů v projektu, starají se jen o ten jeden. Ani kdybyste napsali celý program v jednom jazyce, překladač si nebude ostatních souborů v projektu všímat a vždy kompiluje jen jeden. Z toho pak přímo plyne způsob, jakým je zajištěno propojení jednotlivých souborů: Každý soubor projektu musí explicitně definovat, které své symboly chce zveřejnit a pod jakým jménem (tedy co chce „exportovat“) a naopak které dodatečné (de facto neznámé ci cizí) symboly potřebuje, aby program fungoval (tedy co potřebuje „importovat“). Linker pak projde všechny soubory a propojí mezi nimi exporty a importy.

Poznámka: Principiálně není zakázáno, aby více souborů exportovalo stejně pojmenovaný symbol. Když pak ale jiný soubor bude tento symbol chtít importovat, linker mu dá jeden z těch dvou stejně pojmenovaných a vy nedokážete ovlivnit, který to bude. Jednotlivé linkery tyto situace řeší různě (buď berou vždy první, nebo vždy poslední, takže závisí na pořadí souborů v projektu; nebo mají možnost dalších speciálních nastavení, kterými to lze upravit), je dobré se stejně pojmenovaným exportům vyhnout.

Export se v assembleru provádí velmi jednoduše direktivou public s uvedením symbolu k exportu. Překladač přitom nevyžaduje nějaké speciální pořadí v kódu, obvykle se všechny exporty uvádějí na začátek souboru pro přehlednost či ke každému exportovanému symbolu. Příklad následuje.

```
public MojePromenna
public MojeProcedura
public MojeKonstanta
public MujLabel
```

Exportovat se takto dá jakýkoliv symbol, který je někde v paměti při běhu programu, tedy třeba jen místo ke skoku (label). Platí to i pro procedury (zakládání procedur si popíšeme v následující sekci).

Import funguje podobně jako export, avšak jelikož MASM je typovaný assembler, u každého importovaného symbolu musíme uvést typ. (Pozor! Pro potřeby linkeru se skutečně exportují jen symboly, tedy jména a jejich adresy, zatímco jejich typy či jakékoliv další informace součástí exportu nejsou!) Definice importu je patrná z následujících příkladů.

```
extern CiziInteger:dword
extern CiziZnak:byte
extern CiziProcedura:proc
```

V tabulce je opět uveden i import procedury. Můžeme takto importovat například `printf` a pak jej zavolat pomocí `call printf`. Je však potřeba se ještě naučit předávat parametry při volání a to se naučíme za okamžik.

Průvodce studiem

MASM z historických důvodů umožňuje označovat importy direktivami `extern` i `extrn` (v tom druhém chybí jedno `e`). Obě direktivy dělají přesně totéž, je tedy jedno, který zápis budete používat.

5.7 Podprogramy, procedury a funkce

5.7.1 Vysvětlení pojmu

Pojmy podprogram, procedura a funkce označují totéž: Jedná se část kódu, kterou můžeme zavolat, případně s nějakými parametry, a po skončení volání pokračuje vykonávání kódu na původním místě. Podprogram je tedy jistou obdobou softwarového přerušení – vykonávání kódu je přerušeno a po vykonání podprogramu výpočet pokračuje na místě, kde byl přerušen.

Průvodce studiem

Ptáte se, proč vlastně má tento pojem tři názvy? Původně se používal pouze pojem podprogram a formálně vzato bychom při práci s assemblerem měli používat pouze tento termín. Procedury a funkce byly zavedeny ve strukturovaných jazycích. Mají některé dobře známé vlastnosti a navzájem se liší především tím, že procedura nevrací návratovou hodnotu, zatímco funkce ano. Některé pozdější jazyky tyto dva pojmy opět slily (např. C), protože vracení či nevracení hodnoty není rozhodující. Makro assemblery podprogramům pro změnu říkají obvykle procedury a mají k tomu příslušná klíčová slova. (Z hlediska moderního assembleru je také nepodstatné, zda podprogram vrací či nevrací hodnotu.)

5.7.2 Definice a export procedury

Naučíme se nejprve používat čistý assembler bez maker, který přímo procedury nezná. Chceme-li mít možnost něco zavolat, označíme si příslušný řádek kódu návěštím (jméno a dvojtečka). Chceme-li, aby tento symbol byl vidět z jiných souborů projektu, musíme jej explicitně exportovat direktivou `public`. Pro srovnání: V jazyce C jsou všechny symboly exportovány implicitně (a lze to u jednotlivých součástí zakázat označením `static`). Příklad následuje.

```
public VratJednicku
VratJednicku:
  mov eax,1
  ret
```

Jak už víme z kapitoly 2.10, na konci procedury musí být instrukce `ret` a návratovou hodnotu vracíme v registru `eax`. Tato procedura tedy jednoduše vždy vrací číslo 1.

Pozor! Assembler standardně nerozlišuje velikost písmen, při kompilaci s projektem C/C++ však máme nastaveno, aby exportované symboly velikost písmen ctily dle definice (viz předchozí sekce, konfigurace překladu v MASM). Direktivu `public` lze uvádět i k neexistujícím symbolům, proto když v `public` uvedete jinou velikost písmen, než u symbolu v programu, tak vlastně označujete export neexistujícího symbolu.

5.7.3 Import procedury do jazyka C

V C/C++ si proceduru zpřístupníme pomocí prototypu. (Exportováním jsme symbol zveřejnili v rámci projektu, v každém jednotlivém C/C++ souboru jej však musíme importovat pomocí prototypu. Princip, že „všechno je vidět všude“, který známe třeba z C# a jiných moderních jazyků, zde tedy ani zdaleka neplatí.) Nejprve verze pro jazyk C:

```
int VratJednicku(void);
```

A nyní totéž v jazyce C++:

```
extern "C" int VratJednicku();
```

Typ	Registr
byte	al
word	ax
dword	eax
qword	edx:eax (registrový pár)

Tabulka 10: Registry používané pro vracení hodnot z procedur.

Jak vidno, rozdíl je v tom, že v jazyce C++ musíme před každou takovou deklaraci přidat `extern "C"` a naopak v C musíme u funkcí bez parametrů psát do závorek slovo `void`.

Náš program pak můžeme zavolat a otestovat (vypsat vrácené číslo na obrazovku apod.). Všimněte si, že z assembleru se exportuje jen jméno a nic víc. Formát vstupních parametrů či typ návratové hodnoty tedy nejsou součástí exportních informací. Toto je klasická vlastnost jak assembleru, tak jazyka C. Uděláme-li při psaní prototypů chybu, překladač ji nezjistí. (Můžeme dokonce zaměnit funkci a proměnnou, opět to ve většině překladačů nejde zjistit.)

5.7.4 Import C funkce do assembleru

Podobným způsobem jako volání assemblerovské procedury z C lze také volat C funkce z assembleru. Jelikož C vše exportuje implicitně, na straně C není třeba dělat žádné úpravy kódu. V assembleru pak pouze importujeme příslušný symbol direktivou `extern`. Díky direktivě `.model něco,c` se překladač navíc sám postará o dekorování importovaných jmen. Příklad následuje.

```
extern VratDvojku:proc
```

Nyní můžete zkusit napsat v C funkci `VratDvojku()`, která vrátí dvojku a z procedury `VratJednicku` ji zavolat pomocí instrukce `call`. Pak znovu otestujte proceduru `VratJednicku`, nyní by měla vracet dvojku.

5.7.5 Návratová hodnota

Každá procedura může mít návratovou hodnotu. Jak už jsme uvedli výše, při spolupráci C a assembleru se typ návratové hodnoty nehlídá a závisí jen na našem napsání prototypu. V kapitole 2.5 na straně 14 víme, že hodnota typu `int` se vrací v registru `eax`. Tabulka 10 ukazuje, v jakých registrech se vracejí jednotlivé datové typy assembleru.

Pro zajímavost dodejme, že hodnoty typů s plovoucí řádovou čárkou se vracejí na FPU zásobníku a větší datové struktury lze vracet jen tak, že volající poskytne předem buffer pro zapsání výsledku. (Psát takové volání v assembleru je tedy dosti komplikované.)

5.7.6 Předávání parametrů při volání

Nyní se dostáváme k poměrně složité věci: předávání parametrů. Ve vyšších jazycích se s parametry obvykle pracuje velmi jednoduše a chovají se jako lokální proměnné. V assembleru se také parametry volání a lokální proměnné chovají stejně, ale jejich používání v obyčejném assembleru bez nějakých pomůcek od překladače je složitější. Později se naučíme používat pseudoinstrukce, které vše zjednodušší, nejprve se ale musíme naučit, jak věci doopravdy fungují (proto přece assembler studujeme).

O předávání parametrů byla řeč již na začátku této kapitoly v souvislosti s organizací zásobníku a také na přednášce. Nyní tedy již známé věci jen převedeme do praxe. Standardně v assembleru používáme volací konvenci C, jinými variantami se nyní zabývat nebudeme. Volání si předvedeme na příkladu vypsání formátovaného textu s číslem na obrazovku.

```
.const                  ;začátek konstant

cislo    dword 37
pozdrav  byte "Hello World!",0
format   byte "cislo: %i, pozdrav: %s",13,10,0

.code                   ;začátek kódu

extern printf:proc

public testuj
testuj:
  push offset pozdrav
  push dword ptr cislo
  push offset format
  call printf
  add esp,12
  ret
```

Příklad ukazuje volání `printf` se třemi parametry (formátovací řetězec, číslo a další řetězec). (Pozor! Pro vyzkoušení tohoto příkladu je nutno ve Visual Studiu přepnout na statickou CRT knihovnu (tedy ne DLL).) Parametry pozpátku (zprava doleva) vložíme na zásobník instrukcí `push`, všimněte si přitom, že můžeme na zásobník přímo ukládat i hodnoty proměnných (neboť máme CISC procesor). Důležité také je uvědomit si, že u řetězců ukládáme na zásobník jejich adresu (offset), zatímco u integerů jejich hodnoty (dword ptr). Pozor také na rozdíly mezi lokálními a globálními proměnnými, pointery a poli.

Po skončení volání má volající povinnost uklidit zásobník, tedy uvést ho do stavu před voláním `push`. To samozřejmě lze provést pomocí tří volání `pop`, nebo lépe přímým přičtením příslušné hodnoty k registru `esp`. Programový zásobník funguje na x86 tak, že všechny jeho buňky jsou stejně velké, dle typu kódového segmentu. V našem případě tedy má každé buňka 4 bajty, proto pro uklizení 3 hodnot ze zásobníku přičteme k `esp` číslo $3 \cdot 4 = 12$.

Poznámka: Budete-li potřebovat volat krátce po sobě funkce s podobnými parametry, můžete hodnoty na zásobníku použít. Lze je totiž nechat i pro více volání, či dokonce měnit pomocí `mov [esp+n],něco` apod. Můžete však narazit na problém, že zavolaná funkce hodnotu na zásobníku změní (je to přece její lokální proměnná, takže ji má právo měnit), proto tento druh optimalizace není zrovna bezpečný.

5.7.7 Použití předaných parametrů

Podívejme se nyní na opačnou situaci: My chceme vytvořit funkci s parametry tak, aby byla zavolatelná z jazyka C. Jak na to jsme si už teoreticky popsali na začátku této kapitoly, proto rovnou uvedeme příklad:

```
public soucet
soucet:
  ;entry
  push ebp
  mov ebp,esp
  ;tělo
  mov eax,[ebp+8]
  add eax,[ebp+12]
  ;exit
  pop ebp
  ret
```

Funkce `soucet` sečte dvě čísla předaná jako parametry volání. Ve funkci jsou také vyznačeny tzv. „entry“ a „exit“ kódy nutné k používání parametrů volání a lokálních proměnných. Teoreticky by šlo k proměnným přistupovat i přímo přes `[esp+n]` ale měli bychom problémy s přístupem na zásobník v okamžiku volání dalších funkcí (každé `push` posune `esp`, čímž minimálně vzniká velký chaos ve zdrojovém kódu).

Pozor! Jelikož instrukce `push`/`pop` pracují vždy s 4bajtovými buňkami, předání třeba 10 1bajtových proměnných zabere ve skutečnosti 40 bajtů.

5.7.8 Lokální proměnné

Lokální proměnné se v assembleru používají naprosto sporadicky. Jejich účel a smysl použití je sice úplně stejný jako ve vyšších jazycích, v assembleru ale máme k dispozici registry procesoru, se kterými se lépe pracuje a obvykle nám stačí. Používat můžeme minimálně 6 obecných a indexových registrů eax, ebx, ecx, edx, esi a edi. Oželíme-li lokální proměnné a stack frame, pak máme ještě ebp. Je-li registrů nedostatek, často se hodí ukládat více hodnot do jednoho registru a pomocí bitových rotací (instrukce `rol` a `ror`) si je zpřístupňovat. Dolní dva bajty základních 4 registrů dokonce lze používat i přímo.

Použití lokálních proměnných opět plyne z faktů uvedených na začátku této kapitoly. Entry kód doplníme o vytvoření prostoru na zásobníku posunutím `esp`, exit kód pak musí `esp` obnovit na původní hodnotu. Pro příklad si uveďme úpravu předchozího programu tak, aby se oba vstupní parametry nejprve okopírovaly do lokálních proměnných, a až pak se sečetly.

```
public soucet
soucet:
  ;entry
  push ebp
  mov ebp,esp
  sub esp,8
  ;přeneseme parametry do lokálních proměnných
  mov eax,[ebp+8]
  mov [ebp-4],eax
  mov eax,[ebp+12]
  mov [ebp-8],eax
  ;tělo
  mov eax,[ebp-4]
  add eax,[ebp-8]
  ;exit
  add esp,8
  pop ebp
  ret
```

Používání parametrů a lokálních proměnných tedy spočívá ve vytvoření rámce na zásobníku a odkazováním se přes `ebp`. Největší bolístkou programátora v praxi je pak neexistence názvů, kód je nepřehledný a je zde větší riziko chyb.

5.7.9 Vnořené funkce

Assembler pochopitelně umožňuje i definovat a používat vnořené funkce, tedy funkce definované uvnitř jiných funkcí. Toto se hodí například při realizaci rekurzivních algoritmů, kdy se jakýkoliv algoritmus tváří jako jedna funkce a všechen kód, který se v něm používá, tedy včetně případných rekurzivních funkcí, je umístěn uvnitř oné jedné veřejné funkce. Výhodou tohoto řešení je, že u vnitřních funkcí nemusíme dodržovat konvenci volání C a můžeme si

předávat parametry v registrech dle aktuální potřeby. Jinými slovy: Vnořené funkce jsou jedním z nástrojů optimalizace kódu pro rychlost. Ve vyšších jazycích vnořené funkce obvykle nejdou vytvářet (např. C nebo C#), nebo jde o konstrukt vedoucí naopak ke zpomalení programu (např. Pascal).

5.8 Konstanty

Konstanty (neplést s proměnnými v segmentu `.const`) lze vytvářet dvěma způsoby. Číselné konstanty vytváříme velmi intuitivně pomocí obyčejné značky =. Tyto konstrukce jsou velmi mocné, neboť je lze používat pro makrovýpočty, předefinovávat i pomocí sebe sama. Uveďme několik příkladů

```
jméno = 120
nula = 0
jméno = jméno*2   ;změníme definici konstanty
```

kromě toho MASM umožňuje pomocí direktivy `equ` definovat obyčejné literální konstanty. Tentokrát se jedná o čistě textový konstrukt, který jednou nadefinovaný již nelze nikdy změnit. (Chová se de facto stejně jako `#define` v jazyce C.)

```
jmeno equ 120
nula equ 0
jmeno equ jmeno*2 ;chyba: překladač toto chápe jako "120 equ 120*2"
```

Shrnutí

Inline assembler vepsaný do zdrojových kódu jazyka C/C++ je jednoduchý způsob, jak assembler použít a přitom vynechat složitosti spojené se syntaxí globálních konstruktů, má to však jistá omezení. Například tímto způsobem nelze zakládat celé funkce a také nelze používat prostředky makroassembleru. V této kapitole jsme si podrobně vysvětlili fungování programování zásobníku a jeho význam při volání funkcí (či obecně řečeno podprogramů). Za účelem práce se zásobníkem jsme se naučili tzv. externí assembler, tedy „opravdový“ assembler, který se již nevpisuje do jiných programovacích jazyků.

Pojmy k zapamatování

- programový zásobník
- instrukce push a pop
- instrukce call a ret
- externí assembler
- MASM
- paměťový model
- podprogram, procedura, funkce
- dekorování jmen
- předávání parametrů (při volání)
- globální proměnná
- export a import symbolů
- lokální proměnná
- vnořená funkce
- konstanta

Kontrolní otázky

1. *Kolik programových zásobníků v počítači je? Diskutujte, co by přineslo či jaké by byly problémy, kdyby zásobníků bylo méně či více.*
2. *Co je to paměťový model? Jaký má při programování význam a jak souvisí s operačními systémy?*
3. *Vysvětlete motivaci k použití a princip fungování dekorování jmen symbolů a jejich exportů a importů.*
4. *Většina běžných programovacích jazyků neumožňuje používat vnořené funkce. Pokuste se navrhnout, jak by se vnořená funkce dala implementovat v assembleru. Chceme přitom, aby vnořená funkce měla přístup k lokálním proměnným lexikálně nadřazené funkce (tedy k proměnným funkce, ve které je vepsaná).*

Cvičení

1. Zkuste si převést některé vaše programy (z předchozích kapitol) do externího assembleru. Procvičíte si tak syntaxi.

Úkoly k textu

1. Napište v externím assembleru proceduru pro součet libovolného počtu čísel. Posledním argumentem volání bude vždy nula, podle toho bude určeno, kolik čísel se má sečíst.

```
int soucet(int cislo, ...);
```

2. Napište v externím assembleru proceduru, která vypíše hodnoty prvků v poli. Vstupním parametrem bude velikost pole a adresa začátku pole. Funkce hodnoty vypíše voláním printf.

```
void vypispole(int pocet, int *pole) {
  for(int i=0; i<pocet; i++) {
    printf("%i ", pole[i]);
  }
  printf("\n");
}
```

3. Napište funkci, která se bude chovat jako printf a skutečně zavolá opravdové printf, před tím ale převede formátovací řetězec na velká písmena (kromě formátovacích znaků). Přitom nesmí změnit původní formátovací řetězec.
   Nápověda: Nejprve si okopírujte formátovací řetězec do pomocné proměnné (alokujte si potřebnou paměť pomocí malloc či přímo strdup), pak nahraďte parametry na zásobníku a volejte printf. Nechte si printf vrátit do vaší funkce, abyste mohli uvolnit alokovanou paměť (pomocí funkce free).

4. Napište funkci PrintfKrizek, která se bude chovat stejně jako pravé printf, ale před i za to, co vypíše, přidá ještě křížek (#). Nápověda: Parametry do printf předáte přímo na zásobníku tak, jak jste je dostali. Jen se musíte zásobník upravit tak, aby funkce printf „neviděla“, že je volána přes prostředníka. Stejně tak můžete upravit adresy návratu z volání. Křížek před i za vlastní text vložíte dalšími dvěma voláními printf.

Řešení

1. Ukažme si například zápis strlen (tuto funkci jsme řešili ve cvičení v minulé kapitole). Jedná se o velmi jednoduchou funkci bez lokálních proměnných či volání jiných funkcí, takže můžeme zkusit ji realizovat bez vytváření rámce. Navíc si vystačíme s jediným registrem, ve kterém potom také vrátíme výsledek.

```
public strlen2ext
strlen2ext:
    mov eax,[esp+4]
    dec eax         ;posuneme se jeden znak před začátek textu
dalsi:
    inc eax         ;posuneme se na další znak
    cmp byte ptr[eax],0
    jnz dalsi
    sub eax,[esp+4] ;máme adresu konce, odečteme od ní adresu začátku
    ret
```

6 Prostředky makroassembleru

Studijní cíle: Tato kapitola poskytuje náhled do možností makroassembleru jakožto nástroje pro zpřehlednění a zjednodušení programování v assembleru. Zde diskutovaná témata jsou již jen doplněním studia pro studenty, kteří mají o problematiku a studium assembleru hlubší zájem.

Klíčová slova: makro, direktiva, procedura, podmíněný blok, opakování bloku

Potřebný čas: 80 minut.

Průvodce studiem

Na programování v assembleru existují mezi odborníky různé pohledy. Jedna skupina používá makroassemblery a tvoří v nich high–level kód s tím, že je kratší, přehlednější a méně chybový. Druhá skupina však oponuje, že to už přece není opravdový assembler a používá pouze čistý assembler bez maker. Další skupina zase odmítá typování v assembleru a používá assembler bez datových typů, i když třeba s makry. Autor tohoto učebního textu zcela jednoznačně inklinuje k první zmíněné skupině, nicméně v zájmu studia vnitřního chování CPU bylo nutné studentům řadu užitečných maker ve výkladu látky zatajit. Většina programátorů patřící do zmíněné druhé skupiny nepoužívá high–level konstrukce jednoduše z toho důvodu, že je vůbec nezná. Hlavním úskalím jsou totiž nekvalitní učebnice, které se obvykle pitvají v detailech jednotlivých instrukcí, ale vůbec neřeší, jak v assembleru tvořit větší programové celky. Třetí zmíněná skupina pak obvykle nepoužívá typy, protože jejich překladače assembleru jednoduše typy vůbec neumožňují používat. Například NASM je velmi populárním assemblerem v Linuxu, kde MASM od Microsoftu použít nelze. NASM je překladač s bohatou funkcionalitou maker, ale je beztypový, takže programátory nutí pracovat bez typů ať se jim to líbí, nebo ne.

Tato kapitola čerpá výhradně z manuálu k MASM 6 [Masm]. Dalším zdrojem může být také seznam novinek v MASM 8 [Masm8].

6.1 Direktivy, pseudoinstrukce a makra

Direktivy, pseudoinstrukce a makra jsou prostředky, kterými nám MASM pomáhá zjednodušit si život při práci s assemblerem. Assembler se tak de facto stává o něco vyšším jazykem, než jen přepisem strojového kódu se symbolickými adresami. Celou řadu konstrukcí makroassembleru již známe z předchozích kapitol, nyní se seznámíme s několika dalšími. Nebudou to však všechny, které MASM nabízí, neboť jich existuje opravdu velké množství a jejich zvládnutí je nad rámec našeho studia.

6.2 Export/import a hlavičkové soubory

Místo deklarací public a extern můžeme použít univerzální direktivu `externdef`, která se používá stejně jako `extern`, ale funguje jinak: Je-li symbol v souboru definován, `externdef` se chová jako `public`. Je-li symbol v souboru používán, ale není definován, `externdef` se chová jako `extern`. V ostatních případech se direktiva ignoruje.

Tato direktiva se často používá společně z tzv. include soubory, což jsou soubory speciálně určené k definici veřejných součástí programu. Pomocí direktivy `include` a uvedení jména se include soubor připojí na dané místo zdrojového kódu assembleru. Include soubor má stejný formát jako .asm soubory, ale příponu .inc (což není technicky nutné, ale je to zvykem). V jazyce C se těmto souborům říká „hlavičkové soubory“. (V assembleru je to totéž, pouze místo `#include` píšeme jen `include` a jméno souboru uvádíme bez uvozovek.)

6.3 Výrazy

Do zdrojového textu MASM lze psát libovolné výrazy, které lze spočítat při kompilaci. Viz příklad:

```
add a,ebx+ecx                    ;chyba! taková instrukce není
add a,120*1000 shr 12 xor 10     ;OK
```

Součet dvou registrů uvedený na první řádku není možno spočítat v čase překladu, takže první řádek je chybný. Slova `shr` a `xor` na druhém řádku však nejsou instrukce assembleru, nýbrž aritmetické operátory (s intuitivním významem) a celá pravá strana (vše za čárkou mezi operandy) je tedy konstanta, kterou překladač spočítá při překladu a dosadí jako přímou hodnotu (immediate). Tabulka 11 shrnuje základní operátory MASM.

Operátor	Popis
+, -, *, /	klasické binární aritmetické operátory
+, -	klasické unární aritmetické operátory
shr, shl	bitové posuny
mod	zbytek po celočíselném dělení
not, and, or, xor	klasické bitové operátory
eq, ne	pravdivostní – rovnost, nerovnost
lt, le	pravdivostní – menší než, menší nebo rovno
gt, ge	pravdivostní – větší než, větší nebo rovno
length	počet prvků v poli
size	velikost proměnné či celého pole v bajtech

Tabulka 11: Vybrané základní operátory MASM.

6.4 Definice vlastních typů

Definice vlastních typů direktivou `typedef` má stejný význam jako stejnojmenná konstrukce v jazyce C a používá se nejčastěji k pojmenování typových pointerů. Příklad následuje.

```
char typedef sbyte      ;char je znaménkový bajt
pchar typedef ptr char  ;pchar je pointer na char
```

Poznámka: Jak je vidět na příkladu, slovo `ptr` následované typem má lehce odlišný význam než námi již známé použití opačné, tj. typu následovaného slovem `ptr`.

Vlastní typy také můžeme definovat jako struktury pomocí `struct` či unie pomocí `union`. Tyto konstrukce fungují stejně jako v jazyce C. Ukážeme si jeden příklad za všechny:

```
dwb union
  d dword ?
  w word ?
  b byte ?
dwb ends          ;ends = end structure
```

Jména členských položek nemusejí být jednoznačná v rámci souboru (což možná působí jako samozřejmá věc, ale v assemblerech je toto poměrně nová funkcionalita). Z slovem `struct` lze ještě uvést požadované zarovnání (alignment, opět stejný význam jako v jazyce C), aktuální verze MASM podporuje zarovnání na 1, 2, 4, 8 a 16 bajtů. (Zarovnání struktur na více bajtů zrychluje program, používáte-li však jen 1, 2 a 4bajtové proměnné, nemá zarovnání na více než 4 bajty smysl. Aktuální překladače C obvykle struktury zarovnávají na 8 bajtů, kvůli proměnným typu `double` apod.)

```
ZarovnanaStruktura struct 4
  a1 byte ?  ;začíná na pozici 0
  a2 byte ?  ;začíná na pozici 1
  a3 dword ? ;začíná na pozici 4
ZarovnanaStruktura ends

X ZarovnanaStruktura ;založení proměnné X typu ZarovnanaStruktura

  mov eax, X.a3    ;ukázka přístupu k položce struktury
```

V ukázkovém příkladě je vidět také intuitivní způsob zakládání proměnných strukturovaných typů a přístup k jednotlivým položkám.

Další možností jsou bitová pole pomocí direktivy `record` či typování registrů procesoru pomocí `assume`, např. označíme, že některý registr má být chápán jako pointer na konkrétní datový typ. Ve všech instrukcích odkazujících na paměť tímto registrem se pak předpokládá, že jde o hodnotu (či pole) daného typu.

6.5 Podmíněný překlad

Podmíněný překlad funguje opět podobně jako v jazyce C, používáme direktivy `if`, `elseif`, `else` a `endif`. (Podobnost assembleru s jazykem C samozřejmě není náhodná.) Dále je možno použít `ife` pro překlad při nesplnění podmínky, `ifdef` či `ifndef` pro překlad je-li definován či naopak nedefinován nějaký symbol.

Dále máme k dispozici nástroje pro hlášení chyb. Direktiva `.err` zahlásí chybu danou jako argument vždy, direktivy `.erre`, `.errnz`, `.errdef`, `.errndef` a další se chovají stejně jako příslušné výše uvedené direktivy podmíněného překladu, ale při splnění dané podmínky zahlásí chybu danou jako jejich argument. Zahlášením chyby překlad v daném místě končí a dále nepokračuje.

6.6 FP čísla

Čísla s plovoucí řádovou čárkou jsou jedním z možná i tajemných témat, kterým jsme se záměrně vyhýbali. Důvodem je, že historicky procesory x86 FP čísla nepodporovaly přímo, ale jen pomocí tzv. koprocesoru – samostatného čipu. V instrukční sadě byla dohodnutá značka a když na ni CPU narazil, předal instrukci k vykonání do koprocesoru nebo vyvolal výjimku, když koprocesor nebyl přítomen (což vzhledem k ceně byla nejčastější situace). Instrukci pak mohl místo koprocesoru vykonávat třeba speciální program (nebylo to rychlé, ale možné náhradní řešení). Matematický koprocesor se obvykle označuje FPU (Floating Point Unit).

Průvodce studiem

Wikipedia [Wiki] uvádí, že na původním IBM PC znamenal koprocesor řádově 50násobné zrychlení matematických výpočtů. U jiných procesorů je výkon FPU jednotky často ještě mnohem vyšší. (Intel se během historického vývoje procesorů vždycky zaměřoval spíše na optimalizace výpočtů v ALU či vektorové výpočty v MMX či SIMD.)

FPU má přímý přístup do paměti, používá však vlastní sadu registrů, vlastní registr příznaků a vlastní instrukce. Registrů je 8 a každý má 80 bitů (10 bajtů), všechny výpočty jsou prováděny na těchto 80bitových registrech. Při práci s pamětí umí FPU konvertovat hodnoty mezi svým 80bitovým formátem a kratším 32 a 64bitovým formátem, který se běžněji používá například ve vyšších jazycích. Osmice registrů je organizována jako zásobník, kde vrchol je `st(0)` či

krátce `st` a poslední registr je `st(7)`. S registry je však kromě zásobníkových operací možno pracovat i přímo.

Instrukce koprocesoru začínají písmenem `f`, takže jsou ve zdrojovém kódu snadno rozpoznatelné. Tyto instrukce nikdy nepracují s přímými hodnotami (immediate) a až na jednu speciální instrukci neumějí pracovat ani s běžnými registry CPU. Základní způsob práce s FPU je pomocí zásobníkových instrukcí. Tento způsob je odvozen od běžného způsobu implementace matematických výpočtů v počítači, kdy se systém používající zásobník osvědčil jako nejlepší (neboť nejlépe koresponduje s lidským způsobem zápisu a chápání matematických operací). Následující příklad sečte dvě čísla.

```
fld1    ; vloží 1 na zásobník
fldpi   ; vloží PI na zásobník
fadd    ; vytáhne ze zásobníku 1 a PI, sečte je a výsledek uloží zpět na zásobník
```

Koprocesor umí hardwarově velmi rychle počítat i složitější operace jako je sinus, kosinus či druhá odmocnina. Má však poměrně hodně instrukcí a pro nás není reálné je probrat a naučit se (a zřejmě by to ani nemělo nějaký praktický smysl).

6.7 Bezejmenná návěští

Jedním z klíčových high–level prvků jsou bezejmenná návěští. Definují se pomocí konstrukce `@@:` a odkazovat se dají jen nejbližší předchozí pomocí `@b` (back) a nejbližší následující pomocí `@f` (forward). Smyslem těchto konstrukcí je zbavit se či alespoň minimalizovat výskyt pojmenovaných návěští. Například cyklus může vypadat takto:

```
@@:
    ...
    ...
    ...
    loop @b
```

6.8 Příkazy pro blokové podmínky a opakování

6.8.1 Podmíněné vykonání bloku

Dalším velice užitečným prvkem jsou direktivy implementující rozhodovací příkazy ve stylu strukturovaného programování. Konstrukce `.if/.elseif/.else/.endif` nahradí větvení kódu podmíněnými skoky.

Pozor! Tyto příkazy jsou úplně odlišné od direktiv podmíněného překladu se stejnými jmény bez tečky na začátku. Zatímco direktivy podmíněného překladu umožňuje části kódu při překladu vynechat, tyto direktivy se přeloží do podmíněných skoků `jcc` a budou se vyhodnocovat až při běhu programu. Dalším důležitým rozdílem je, že se zde používají operátory v podobě jako ve vyšších jazycích typu C, podrobněji si je představíme níže v sekci 6.8.3. Direktiva však umí jen základní typy testů, které lze přímo převést na podmíněné skoky. Pro příklad si ukážeme přepis tohoto programu do assembleru:

```
int a,b,c;
if(a<=2 || b!=d) a=2; else a=1;
```

V assembleru totéž napíšeme takto:

```
cmp a,2
jle lab2
mov eax,b
```

```
   cmp eax,d
   je lab1
lab2:
   mov a,2
   jmp lab3
lab1:
   mov a,1
lab3:
```

Pomocí direktivy `.if` lze totéž napsat bez pojmenovaných návěští.

```
.if a<=2
   mov a,2
.else
   mov eax,b
.if eax!=d
   mov a,2
.else
   mov a,1
.endif
```

Z příkladu je vidět, že kód pro je sice bez pojmenovaných návěští a je plně strukturovaný, ale u složitějších podmínek není optimální co do efektivity kódu, ani není příliš přehledný, protože složené testy je třeba rozepsat na víc vnořených či po sobě jdoucích testů.

6.8.2 Opakování bloku

Po seznámení s `.if` zřejmě nikoho nepřekvapí, že MASM má i direktivy `.while` a `.repeat` pro opakování kódu. Tyto direktivy používají opět test podmínek pomocí speciálních operátorů popsaných samostatně v následující sekci.

Direktivy `.while` a `.endw` označují blok kódu, který se opakuje, dokud platí podmínka uvedená na začátku bloku. Podmínka se poprvé testuje ještě před prvním vykonání bloku. Jde tedy o klasickou konstrukci while známou z jazyka C.

Direktivy `.repeat` a `.until` označují blok kódu, který se opakuje, dokud nezačne platit podmínka na konci bloku. Od while se tedy liší dvěma vlastnostmi: Podmínka se poprvé testuje až po prvním vykonání bloku a podmínka se testuje opačně, tj. blok kódu se opakuje, dokud podmínka neplatí. U tohoto příkazu je možno alternativně použít závěrečnou direktivu `.untilcxz`, která generuje místo podmíněných skoků instrukci `loop` (takže vždy sníží hodnotu `ecx` o jedničku). Tato direktiva tedy nepotřebuje uvádět nějakou další podmínku, ale je to možné (pak se testuje daná podmínka a ještě se přidá `loop`).

U opakovacích konstrukcí je také k dispozici `.break` a `.continue` (význam je zřejmý z názvu). Tyto dvě direktivy je také možno kombinovat s direktivou `.if` a provést je tak jen při platnosti určité podmínky. Příklad následuje.

```
.while !carry?
   ...
.break .if ecx==0
   ...
.endw
```

6.8.3 Operátory podmíněného vykonání

Na příkladech direktiv `.if` a `.while` bylo vidět, že pro určení podmínek se zde používají jiné operátory, než jaké jsme si uváděli dříve v tabulce. MASM zde používá relační operátory

stejné jako v jazyce C, tj. ==, ! =, >, >=, <, <=, & (bitový test), ! (negace), && (logický součin) a || (logický součet). Dále lze testovat hodnoty příznaků pomocí slov $carry?$, $zero?$, $overflow?$, $sign?$ a $parity?$.

U aritmetických testů se samozřejmě rozlišují znaménkové a neznaménkové hodnoty, stejně jako to známe z klasických podmíněných skoků (na ně se ostatně tyto vyšší konstrukce nakonec přeloží). U operandů je proto velmi důležité hlídat nastavení znamének. Známým operátorem `ptr` lze typy upřesnit přímo v testech, pro určení znaménkovosti hodnoty lze `ptr` použít i u registrů procesoru(!).

Závěrem ještě dodejme, že překladač může při složitějších testech použít registry procesoru k uložení pomocné hodnoty, například proměnné atp. Pokud však lze podmínku vyhodnotit, bez použití dalších registrů, tak samozřejmě překladač vytvoří tento jednodušší kód.

6.9 Procedury

6.9.1 Definice a volání procedury

Jak víme, model C, který uvádíme na začátku každého zdrojového textu assembleru, zajišťuje správné dekorování exportovaných i importovaných symbolů. Další velmi důležitá funkcionalita, kterou jím získáme, je možnost definovat či importovat high level procedury a volat je pomocí pseudoinstrukce `invoke`. Nejprve si na příkladu `printf` ukážeme definici prototypu cizí funkce a jejího zavolání z naší funkce (`printf` má navíc proměnlivý počet parametrů, takže je to zvlášť zajímavý příklad).

```
.data

f byte "zkusebni cislo:%i a text: %s",13,10,0
t byte "Hello World!"

.code

printf proto format:ptr sbyte, args:vararg

public soucet
soucet proc
        invoke printf,offset f,50,offset t
        ret
soucet endp
```

Direktiva `proto` definuje prototyp funkce, tj. importuje daný symbol a zároveň definuje typy parametrů. Slovem `vararg` označujeme proměnlivý počet parametrů (konkrétně `printf` má jako první parametr `char*`, pak mohou následovat další parametry, viz dokumentaci CRT).

Definice procedury touto direktivou automaticky zajistí i entry a exit kód. Entry kód je vložen na začátek procedury, exit kód pak ke každé pseudoinstrukci `ret`. Znamená to, že `ret`můžete uvést kamkoliv do těla procedury, ne jen na její konec, ale ze stejného důvodu nelze vytvářet vnořené podprogramy. (Návrat z vnořeného podprogramu by měl taky exit kód, který by tam vadil. Potřebujete-li volat jiný (lokální) kód z takto definované procedury, můžete, ale příkaz návratu nesmí být řešen umístěním `ret` uvnitř procedury.)

Průvodce studiem

Jelikož překladač generuje epilog ke každému použití pseudoinstrukce `ret`, je často rozumnější použít jen jediné `ret` v každé proceduře a skočit pomocí nepodmíněného skoku

na toto místo skočit. Naopak se vyvarujte skákání na `ret` do jiné procedury, protože přesná podoba epilogu je v každé proceduře odlišná (přesněji řečeno závisí to na počtu volacích parametrů, lokálních proměnných a registrů v klauzuli `uses`).

Volací direktiva `invoke` umí i automatické rozšiřování parametrů (např. prototyp definuje parametr 4bajtový, a vy voláte s 1bajtovou hodnotou). Používá přitom registry `eax` a `edx`, takže není možno používat rozšiřování a zároveň se snažit v těchto registrech předávat hodnoty. (Poznámka autora: Překladač MASM se při testech choval poněkud nepředvídatelně. Někdy dokonce vyrobil docela nečekaný neplatný kód, když například dword hodnotu uložil na zásobník jako 6 bajtů.)

Procedury lze volat také odkazem, například máme-li sadu procedur o stejné signatuře, můžeme v nějakém registru spočítat adresu konkrétního volání a potom ji zavolat dle prototypu. Tato konstrukce je bohužel trochu složitá, vyžaduje definici pointeru na funkci pomocí `typedef`, pak teprve lze přetypovat registr při volání pomocí `ptr`. Alternativně lze totéž udělat pomocí direktivy `assume`, kterou lze registru natrvalo přiřadit daný typ. (To se hodí spíše při častějším používání.)

6.9.2 Uchování měněných registrů

Před seznam parametrů je možno ještě vložit seznam použitých registrů. Tento seznam je uveden slovem `uses` a jednotlivé registry jsou odděleny mezerami (tedy ne čárkami). Za čárkou pak následuje seznam parametrů. Příklad následuje.

```
mojefunkce proc uses esi edi, param1:dword, param2:dword
  ...
  ret
mojefunkce endp
```

Připomeňme, že Visual C++ v souladu s protokolem fastcall definuje (a to bez ohledu na použitou volací konvenci), že každá funkce může měnit `eax`, `ecx` a `edx`, zatímco ostatní registry je nutno obnovit do původního stavu. Typicky tedy používáme `uses ebx esi edi` (pokud tyto registry měníme) a `ebp` uchováváme a obnovujeme v rámci prologu/epilogu.

6.9.3 Lokální proměnné

Ve spojitosti s direktivou `proc` je možno také pohodlně vytvářet pojmenované lokální proměnné. Slouží k tomu direktiva `local`. Příklad následuje.

```
mojefunkce proc
local a:dword, b:dword, c:dword, pole:dword:100
  ...
mojefunkce endp
```

Pozor! Direktiva `local` musí být na řádku přímo následujícím za direktivou `proc` a jednotlivé proměnné se píší dohromady (na jeden řádek) oddělené čárkami. Poslední proměnná v příkladu ukazuje definice lokálních polí.

Shrnutí

V této kapitole jsme si velmi stručně představili možnosti programování s makry v makroassembleru MASM. Makra, direktivy a související konstrukty jsou v assembleru především nástroji pro zpřehlednění a zjednodušení programování, přitom při rozumném použití přinášejí

i dosti důležitou přehlednost. Při méně rozumném použití však makra mohou být i důvodem nepřehlednosti v kódu a je tedy čistě otázkou, jak je programátor bude používat. U složitějších konstrukcí často musíme obětovat přehlednost a používáme makra především pro zjednodušení zdrojového kódu (ve smyslu zkrácení na úkor přehlednosti). S využitím maker lze také naprogramovat řadu věcí, které bez nich ze syntaktických důvodů v assembleru ani mnohých jiných jazycích vůbec naprogramovat nejde.

Pojmy k zapamatování

- direktiva
- pseudoinstrukce
- makro
- hlavičkový soubor
- výraz
- vlastní (datový) typ
- podmíněný překlad
- bezejmenné návěští
- lokální proměnná

A Další témata assembleru

A.1 Co bylo ve starém online textu navíc

Ve starém online studijním textu byla ještě řada pokročilejších témat. Zmíněný materiál je stále k dispozici na adrese http://www.keprt.cz/vyuka/, uveďme si zde alespoň stručný seznam témat (u každého je číslo kapitoly, ve které je diskutováno):

- 1. Kombinace operandů u násobení a dělení
- 1. Disassembler
- 1. Ukázka noc200
- 3. Externí assembler – podrobnosti k dalším překladačům C a taky TASM
- 4. Přehled instrukcí dle kategorií
- 5. Další pseudoinstrukce a direktivy ($, org)
- 5. Konstanty (=, equ, rept)
- 5. Makra

V 6.části je ještě stručně probrán programový model v systému MS–DOS.

- Segmentové registry
- Segmentové direktivy
- Paměťové modely DOSu (tiny, small, large, compact, medium)
- Alokace paměti a další systémové funkce
- Vytváření samostatných EXE a COM souborů
- Čtení parametrů z příkazové řádky

A.2 Doporučené volby ve vlastnostech projektu

Následuje seznam doporučených voleb projektu ve Visual Studiu. První volba je důležitá při volání funkcí CRT z assembleru, další tři usnadňují disassemblování tím, že vypnou všelijaké kontrolní mechanizmy projevující se dodatečným kódem vloženým především na začátek a konec každé funkce. Poslední volba je důležitá při volání funkcí Windows API (z jazyka C i assembleru).

C/C++ / Code Generation / Run Time Library → Multi-threaded Debug
– vypne volání CRT funkcí přes DWORD – důležité!

C/C++ / Code Generation / Basic Runtime Checks → Default
– nekontroluje stack frame (přetečení polí atp.), nekontroluje uninitialized variable

C/C++ / Code Generation / Buffer Security Check → No
– vypne kontrolu přetečení bufferu (má efekt jen ve funkcích, kde jsou lokální pole)

C/C++ / Linker / General / Enable Incremental Linking → No
– vypne volání funkcí přes tabulku JMP skoků

General / Character Set → Use Multi–Byte Character Set
– vypne unicode

B Historie MASM

Studijní cíle: Tato příloha neslouží jako studijní materiál, pouze přináší několik zajímavostí z historie vývoje překladače MASM.

Klíčová slova: MASM, TASM, historie, Microsoft

Potřebný čas: 5 minut.

MASM je jeden z mála programů, který s námi byl po celou dobu existence počítačů PC. Na stránce [Harv] je povídání o vývoji MASM v posledních 20 letech. Pro zajímavost: V 80.letech vycházel MASM jako komerční produkt pro MS-DOS. Teprve ve verzi 5.00 se objevily pomocné direktivy `.model` či `.code` (pro nastavování segmentů je dodnes k dispozici i jiná hodně složitá direktiva). Verze 5.10 přidala lokální návěští začínající `@@` a podporu IBM OS/2. Poslední verzí starých časů byla 5.10B z roku 1989. Ta ještě běžela v 640KB RAM a v několika pozdějších verzích překladače ještě bývala přibalena jako `masm386.exe`. S těmito starými verzemi je také kompatibilní konkurenční překladač Borland TASM.

Současné verze MASM jsou odvozeny od verze 6.00, která vyšla roku 1992 ještě jako plně komerční vývojový nástroj. Tato verze se hodně lišila od předchozích verzí (a potažmo TASM) a přidala zejména `.if`/`.endif` direktivy umožňující strukturované programování téměř bez pojmenovaných návěští. Tato verze běžela v DOSu na procesorech 286 a 2MB RAM. Téhož roku následovaly další opravné verze ve formě záplat, např. 6.10 už běžela jen na 386 a 4MB RAM a měla přidánu podporu tehdy nového Visual C++ 1.0 (které zabíralo 50MB na disku, na tehdejší poměry docela hodně). Verze 6.10A byla poslední, která obsahovala integrované vývojové prostředí. (Editor Programmer's Workbench (PWB) je známý i ze starého Microsoft C a Basicu.)

Verze 6.11 vyšla roku 1993. Za cenu 250 dolarů to byla první verze pro 32bitové Windows, s přibaleným emulátorem Win32 pro MS-DOS (který samotný byl velmi zajímavý, ale skončil v propadlišti dějin). Tato verze podporovala instrukce Pentium (`.586`). Verze 6.11 vznikla v době betaverze prvního Windows NT a přechod na verzi jen pro NT byl zřejmě poněkud zbrklý. Při uvedení finální verze Windows NT na něm pak kvůli rozdílům od beta verze tento překladač paradoxně nefungoval, dokud Microsoft nevydal speciální záplatu, která simuluje chování beta verze na finální verzi NT. Verze 6.11C roku 1994 jako první podporovala tvorbu VxD ovladačů zařízení pro Windows 95.

Verze 6.12 v roce 1997 přidala podporu Pentium Pro (`.686` a `.686p`) a MMX instrukce (`.MMX`) a verze 6.13 téhož roku přidala ještě AMD 3D instrukce (`.K3D`). V roce 1999 Intel vydal sadu maker pro podporu MMX a nových SSE instrukcí v MASM 6.12 (zdarma).

Verze 6.14 v roce 1999 přidala podporu SSE instrukcí a k nim 128bitový datový typ `OWORD` (octet word). V roce 2000 Microsoft vydal tzv. „Processor Pack" jako záplatu pro tehdejší Visual Studio 6.0, ve které se objevil MASM 6.15 a po dlouhých osmi letech záplat také nová dokumentace k tomuto assembleru. Tato verze již byla uvolněna volně ke stažení, MASM byl poprvé zdarma.

V roce 2002 se objevil MASM 7.00 jako součást Visual Studia .NET. Interní číslo verze zůstalo 6.15, šlo tedy možná jen o sladění čísel verzí v balíku Visual Studia. Microsoft dává MASM nyní do každé edice Visual Studia (zatím to platí do verze 2008), MASM je tedy zdarma, ale je třeba si koupit Visual Studio (které stojí víc, než původně MASM). MASM překladač nepotřebuje ke spuštění .NET Framework a lze jej provozovat i na Windows 98 (i když Visual Studio ne). V době psaní tohoto textu Microsoft potvrdil, že MASM bude opět zdarma poskytován v rámci budoucích verzí Visual C++ Express (počínaje záplatou 2008 SP1). Mezitím Borland svůj konkurenční překladač TASM stále prodává, i když je poměrně obtížné na jejich webové stránce najít o něm aktuální informace.

C Úvod do jazyka C++

Studijní cíle: Tato příloha poskytne stručný úvod do jazyka C++ pro programátory v C#. Vzhledem k podobnosti různých programovacích jazyků bude stejně dobře srozumitelná i programátorům v Javě, PHP a některých dalších jazycích.

Klíčová slova: C++

Potřebný čas: 20 minut.

C.1 Úvod

Většina následujících informací platí stejně i pro jazyk C, neboť prvky, ve kterých se tyto dva jazyky liší, při studiu assembleru nepoužíváme. Při práci s assemblerem je často dokonce lepší omezit se jen na jazyk C, protože se tím vyhneme některých problémům způsobeným odlišným stylem dekorování jmen v C++ (viz také kap. 5.5 na straně 58 pro další informace o dekorování jmen).

C.2 Datové typy

Základní datové typy ukazuje tabulka 12. Základní celočíselný typ se jmenuje `int`. Jeho velikost se liší dle typu počítače a prostředí, protože je stejná jako velikost běžného registru či datové sběrnice počítače. V našem případě má tedy velikost 4 bajty.

typ	velikost	typ v assembleru
char	1	byte
short	2	word
int	dle počítače	–
long	4	dword
long long	8	qword

Tabulka 12: Základní datové typy jazyka C++.

Znaky jsou 1bajtové, čili nepoužívá se kód Unicode. Čísla v plovoucí čárce fungují stejně jako v C# (ale v assembleru se používají jinak, takže nás až tak zajímat nebudou).

Typ string sice v C++ je, ale nejde o základní typ a nedá se používat v assembleru, takže místo něj používáme náhradní řešení dle staršího jazyka C. Ten totiž typ string nemá vůbec a texty ukládá do polí znaků. Takový string především nemá nikde explicitně uloženu informaci o své délce – skutečnou délku stringu zjistíme tak, že projdeme jeho znaky a najdeme nulu. Funkce vracející délku stringu tedy může vypadat např. takto:

```
int strlen(char *text) {
  int i=0;
  while(text[i]!=0) i++;
  return i;
}
```

Parametr `text` je v této ukázce pointer. Je to 4bajtová proměnná, ve které je uložena adresa paměti (přesněji offset – číslo paměťové buňky od začátku paměti). Slovo „pointer“ překládáme jako ukazatel – ukazuje totiž někam do paměti. Pointery sice v jazyku C# jsou také, ale v drtivé většině případů je nepoužíváme, takže je to pro nás věc zcela nová. Při studiu assembleru se s nimi však velmi dobře seznámíme, jsou to skutečně jen proměnné obsahující adresu. Na rozdíl od assembleru, který nekontroluje u datových typů nic jiného než velikost (tj. kolik bajtů zabírají v paměti), C++ u pointerů také hlídá, na co ukazují. Hvězdičkou tedy označíme, že

proměnná je typu pointer, ale před hvězdičku musíme ještě napsat, na jaký typ hodnot ukazuje. V assembleru se všechny tyto „hvězdičkové“ proměnné ale používají stejně – nejsou to nic víc než 4bajtová čísla.

C.3 Umístění kódu a dat

Kód v C++ je uložen ve funkcích. Ačkoliv jazyk umožňuje používat i třídy a metody, v assembleru se toto použít nedá, takže pracujeme pouze z globálními funkcemi. V programu je tedy vynechána hlavička „class Jméno“ a všechen kód je na globální úrovni. Stejným způsobem umísťujeme i proměnné, které chceme sdílet mezi funkcemi – hovoříme pak o globálních proměnných. Proměnné mohou být i lokální, pak fungují stejně jako v C#.

Hlavičky funkcí jsou velmi podobné tomu, co známe ze C#. Pouze vynecháme údaj o viditelnosti (public/private).

C.4 Volání

Funkce voláme podobně jako v C#, pouze nemáme třídy a objekty, takže všechny funkce jsou globálně přístupné. C++ má také dvě knihovny – CRT je knihovna funkcí jazyka C, které C++ obsahuje také, knihovna STL obsahuje třídy, které ale až na výjimky používat nebudeme.

Funkce CRT lze volat přímo z assembleru, jednoduše příkazem `call jmeno` nebo `invoke jmeno`. Některé verze knihovny CRT však používají nepřímý model, kdy zpřístupňují jen pointery na funkce, a ne přímo funkce. Toto nemá žádný vliv na funkčnost a v C/C++ to ani nejde nijak poznat. V assembleru ale v těchto případech bohužel musíme použít zápis volání `call dword ptr jmeno` či `invoke dword ptr jmeno`. Parametry (obvykle) předáváme konvencí C, tj. zprava doleva je uložíme na zásobník a volající zásobník musí po sobě i uklidit.

C.5 Hlavičkové soubory

Funkce z knihoven lze v C++ používat až po „inkludování“ příslušného hlavičkového souboru, kde mají funkce deklaraci. Toto je věc, která v C# nemá ekvivalent – tam co máme v projektu, to můžeme použít přímo. V C/C++ ale musíme na začátku zdrojového souboru dát příkazy ve tvaru `#include <soubor>`, kterými do programu zahrneme („inkludujeme“) příslušný soubor. Informace o tom, která funkce je ve kterém hlavičkovém souboru, jsou k nalezení v helpu.

C.6 Některé užitečné funkce CRT

C.6.1 Psaní na obrazovku

Základní věcí, která se nám bude hodit, je výpis na obrazovku. To můžeme provést buď pomocí `printf`, nebo pomocí `cout`.

Funkce `printf` je v CRT (`#include <stdio.h>`), ve Visual Studiu ji můžeme v projektu založeném přes wizard používat přímo. Prvním parametrem této funkce je formátovací řetězec, kde znak procento signalizuje, že následující znak určí typ parametru. Další parametry jsou pak libovolné a dosazují se v přesném pořadí na místa procent. Nejlépe to pochopíme na příkladech vypsání základních datových typů, viz tabulka 13.

Jak je vidět v tabulce 13, konec řádku se označuje stejně jako v C# symbolem `\n`.

typ	příklad
int	`printf("hodnota = %i\n", i);`
char	`printf("znak = %i\n", c);`
char*	`printf("text = %s\n", s);`
různé	`printf("víc hodnot: %i %i %s\n", a, b, text);`

Tabulka 13: Příklady použití příkazu printf.

Druhou možností, jak něco vypsat na obrazovku, je objekt `cout` z knihovny STL. Jeho použití je poněkud netradiční, má totiž překrytý operátor <<. Výhodou je však jednodušší použití než u funkce `printf`.

```
cout << cokoliv << endl;
```

Konstanta `endl` značí konec řádku.

C.6.2 Alokace paměti

Operátor `new` nelze jednoduše zavolat z assembleru, proto pro alokaci paměti používáme následující funkce z knihovny CRT (`#include <malloc.h>`):

```
void* malloc(int size);
void free(void*);
```

První funkce alokuje paměť o daném počtu bajtů a vrací beztypový pointer na ni. Druhá funkce přijme tento pointer a paměť uvolní. Alokovanou paměť musíte vždy sami uvolnit, protože jazyky C/C++ automatickou správu paměti neprovádějí.

D Seznam obrázků

E Seznam tabulek

Reference

[Harv] R. E. Harvey. *Assemblers.* http://ourworld.compuserve.com/homepages/r_harvey/doc_book.htm

[Hyd96] Randall Hyde. *The Art of Assembly Language. – MASM verze* 1996. http://www.arl.wustl.edu/ lockwood/class/cs306/books/artofasm/toc.html

[Kep07] Aleš Keprt. *Operační systémy.* Univerzita Palackého, 2007. Studijní text pro distanční vzdělávání, dostupný studentům na adrese http://www.keprt.cz/vyuka/.

[IA32] *IA-32 Intel Architecture Software Developer's Manual.* Intel 2006. (Má několik svazků a existuje ve verzích pro jednotlivé procesory Intel, ke stažení na www.intel.com.)

[Joh04] Peter L. B. Johnson (Ed.) *Computer Engineering II – Laboratory Notes.* University of Illinois, Urbana-Champaign, IL (USA). http://courses.ece.uiuc.edu/ece390/books/labmanual/inst-ref-general.html

[Masm] *Microsoft MASM 6.1 Programmer's Guide.* Microsoft, 1992.

[Masm8] *New MASM Features [in Visual C++ 2005].* Microsoft, 2004. http://msdn2.microsoft.com/en-us/library/xw102cyh(VS.80).aspx

[Wiki] *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/