Inline assembler, registry a základní aritmetické operace na platformě Intel x86

23. února 2021

Pomocí assembleru jsme schopni vytvářet programy na té nejnižší možné úrovni, tj. na úrovni jednotlivých instrukcí procesoru. V minulosti nebylo neobvyklé, že programátoři přepisovali kritické rutiny svých programů do assembleru, aby dosáhli maximálního výkonu. Díky masivnímu pokroku v oblasti překladačů tato doba dávno minula a moderní překladače jsou schopny vytvořit efektivnější kód než programátor. Jsou však oblasti kdy, použití assembleru má svůj nezastupitelný význam a to je systémové programování (operační systémy, překladače), systémy s omezenými prostředky (vestavné systémy, spotřební elektronika, řídící systémy) a aplikace využívající specializované instrukce procesoru (zpracování obrazu, kryptografie). V tomto a následujících cvičeních si na instrukční sadě procesorů rodiny Intel x86 ukážeme činnost procesoru a jak je běh programu realizován pomocí jednotlivých instrukcí procesoru.

1 Inline assembler

Vytvářet program nebo jeho části pomocí assembleru můžeme dvěma způsoby. (i) Buď můžeme všechen kód napsat v assembleru a přeložit jej pomocí assembleru[1] do strojového kódu, který lze spustit jako samostatný program, případně volat z vyššího programovacího jazyka. (ii) Můžeme také kombinovat kód ve vyšším programovacím jazyce (např. C) s kódem v assembleru pomocí tzv. *inline assembleru*.

Vytvořit rozumný program v assembleru od začátku do konce vyžaduje relativně široké znalosti. Proto, aby učící křivka nebyla příliš strmá, budeme využívat ve cvičeních inline assembler, kdy překladač jazyka C za nás spoustu věcí vyřeší a umožní nám soustředit se jen na řešení konkrétních problémů.

Většina překladačů jazyka C umožňuje nějakou formou propojit kód v jazyce C s kódem v assembleru. My budeme používat 32bitový překladač z Microsoft Visual Studia (MSVC)[2], protože ten nabízí velmi komfortní, téměř bezešvé, propojení C a assembleru, jak ukazuje následující příklad.

```
1 int inc(int n) {
2     _asm {
```

[1] Původně slovo *assembler* označovalo nástroj, který vzal program v tzv. *jazyce symbolických adres* a přeložil jej do strojového kódu. Postupně se označení assembler přeneslo i na jazyk symbolických adres a dnes naprosto běžně pojem assembler označuje jak nástroj, tak i jazyk popisující program na úrovni jednotlivých instrukcí.

[2] Visual Studio je k dispozici v rámci univerzitní licence, případně můžete použít katederní Terminal Server. Pokud byste chtěli použít jiný překladač, můžete, ale nebude k němu existovat podpora ze strany vyučujícího, tj. pokud narazíte na nějaký specifický problém vztahující se k překladači, budete muset nalézt řešení sami.

```
3       mov eax, n  // do registru eax nacteme hodnotu argumentu
4       add eax, 1  // k hodnote pricteme jednicku
5       mov n, eax  // hodnotu vratime do promenne n
6     }
7     return n;
8   }
```

V tomto příkladu máme jednoduchou funkci, která má jeden argument typu `int` a vrací hodnotu stejného typu. Uvnitř této funkce na řádcích 2 až 6 je blok určený direktivou `_asm`, která umožňuje vkládat kód přímo v assembleru. Na řádku č. 7 pokračuje opět kód v jazyce C.

Když se podíváme podrobněji na kód v assembleru, vidíme, že nejdříve vložíme do registru `eax` hodnotu proměnné (argumentu) `n` (řádek č. 3), pak tuto hodnotu zvýšíme o 1 (řádek č. 4) a pak tuto hodnotu uložíme zpět do proměnné `n` (řádek č. 5).

Na tomto příkladu si povšimněme také komentářů, ty jsou uvozeny `//`[3]. Psaní komentářů ke kódu v assembleru je nanejvýš důležité, dalo by se říct, že až nutné. Bez nich je kód pro ostatní programátory nesrozumitelný a s časovým odstupem se stane nesrozumitelným i pro samotného autora programu. Obvykle se do komentářů píše, co daná část kódu provádí a co obsahují jednotlivé registry.

V našem ukázkovém příklad je zajímavé i to, co není na první pohled vidět. Například nikde není uvedena adresa proměnné `n` (tu za nás doplní překladač), nemuseli jsme se starat ani o to, jakým způsobem byly předány argumenty funkci a jak vrácena návratová hodnota. V tomto nám inline assembler značně usnadňuje práci. V neposlední řadě při ladění programu můžeme používat *breakpoints* a *watches* i na kód v assembleru a sledovat tak, co se v programu děje, stejným způsobem jako by to byl program v C.

Princip, kdy v jazyce C napíšeme deklaraci funkci a vrácení hodnoty a samotné tělo funkce napíšeme v inline assembleru, budeme používat pro řešení jednotlivých úloh.

2 Instrukční sada x86

Instrukční sada procesorů x86 je velmi košatá a není možné ji v rámci jednotlivých cvičení popsat celou. Proto u jednotlivých cvičení budou popsány jen určité tématické části a pro další informace odkazujeme laskového čtenáře k dalším materiálům:

- Keprt, A. Assembler.[4]
- Brandejs M. Mikroprocesory Intel Pentium. Brno: Fakulta informatiky, Masarykova univerzita, 2010.[5]
- Přehled všech operací procesorů rodiny x86[6]. (Není potřeba znát.)

[3] V assembleru je zvykem uvazovat komentář znakem `;` (středník), to lze i v inline assembleru MSVC, ale editor Visual Studia má různé problémy s odsazením.

[4] `http://phoenix.inf.upol.cz/~krajcap/courses/2021LS/OS1/Assembler.pdf`

[5] `http://www.fi.muni.cz/usr/brandejs/Brandejs_Mikroprocesory_Intel_Pentium_2010.pdf`

[6] `http://ref.x86asm.net/coder32.html`

2.1 Registry

Procesory x86 nabízí čtveřici obecně použitelných 32bitový registrů `eax`, `ebx`, `ecx`, `edx`, které se dále dělí na 16bitové registry `ax`, `bx`, `cx`, `dx`, které se dále dělí na dvojice osmibitových registrů `ah`, `al`, `bh`, `bl`, `ch`, `cl` a `dh`, `dl`. Za obecně použitelné lze považovat i 32bitové registry `esi` a `edi`, které se dále dělí jen na 16 bitové registry `si` a `di`.

Vedle těchto registrů existují a jsou přístupné ještě registry `esp` a `ebp`, které ale mají specifickou funkci a nelze je použít libovolně. Další registry jako jsou `eip` a `ef(lags)` mohou být měněny jen vybranými instrukcemi.

2.2 Přehled základních aritmetických instrukcí

Instrukce mají obvykle tvar:

⟨ *název instrukce* ⟩ ⟨ *cílový operand* ⟩ [, ⟨ *další operand* ⟩, ...]

Například instrukce ščítání `add` má právě dva operandy, kdy k prvnímu operandu je přičtena hodnota operandu druhého, tzn. máme-li instrukci `add eax, ebx`, znamená to, že k hodnotě v registru `eax` je přičtena hodnota `ebx`. To odpovídá výrazu `eax += ebx`, jak jej známe z vyšších programovacích jazyků.

Operandy instrukcí mohou být

- `r` – registry
- `m` – adresa místa v paměti[7].
- `i` – přímé hodnoty (konstanty)

Každá instrukce připouští jen určité kombinace operandů[8], význam jednotlivých základní aritmetických instrukcí a jaké jsou přípustné operandy ukazuje následující výčet.

```
mov r/m, r/m/i        ; op1 := op2
add r/m, r/m/i        ; op1 := op1 + op2
sub r/m, r/m/i        ; op1 := op1 - op2
neg r/m               ; op1 := - op1

inc r/m               ; op1 := op1 + 1
dec r/m               ; op1 := op1 - 1

mul r/m               ; edx:eax := eax * op1
imul r, r/m           ; op1 := op1 * op2
imul r, r/m, i        ; op1 := op1 * op2 * op3

div r/m               ; eax := edx:eax / op1; edx := edx:eax % op1  (neznaménkové dělení)
idiv r/m              ; eax := edx:eax / op1; edx := edx:eax % op1  (znaménkové dělení)
```

[7] V našem případě proměnné odpovídají adresám v paměti.
[8] Navíc paměť lze v jedné instrukci adresovat pouze jednou.

Instrukce pro násobení a dělení jsou atypické v tom, že mají určené registry, se kterými pracují. V případě instrukce `mul` je výsledek 64 bitový a je uložen do dvojice registrů `edx` a `eax`, kde registr `eax` obsahuje spodních 32 bitů výsledku a registr `edx` horních 32 bitů. Ještě ošemetnější situace je při dělení. Jednak dělí se 64 bitová hodnota ve dvojici registrů `edx` a `eax`. Z toho plyne, že i kdybychom chtěli dělit pouze 32bitovou hodnotu v registru `eax`, vždy musíme korektně nastavit i hodnotu v registru `edx`[9]. Současně je výsledek zapisován do registrů `eax` (podíl) i `edx` (zbytek po dělení). V důsledku toho, i když nás zbytek po dělení velice často nezajímá, je toto hodnota uložena do registru `edx` a je to potřeba brát v potaz.

Při popisu instrukcí násobení a dělení jsme uvažovali, že operand je 32bitový. Z historických důvodů a z důvodů zpětné kompatibility, pokud je operand instrukcí `mul`, `div` nebo `idiv` 16bitový, pracují jednotlivé instrukce s dvojící registrů `ax` a `dx`.

[9] Pokud pracujeme s kladnými čísly, stačí zajistit, že obsah `edx` bude 0.