Externí assembler, registry a základní aritmetické operace na platformě Intel x86/AMD64

2. března 2022

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 kde 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 (resp. AMD64) ukážeme činnost procesoru a jak je běh programu realizován pomocí jednotlivých instrukcí.

1 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*, jak ukazuje následující příklad pro překladač MSVC (Visual Studio):

```
int inc(int n) {
   _asm {
        mov eax, n  // do registru eax nacteme hodnotu argumentu
        add eax, 1  // k hodnote pricteme jednicku
        mov n, eax  // hodnotu vratime do promenne n
    }
    return n;
}
```

[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.

Protože překladač MSVC nepodporuje v inline assembleru jinou architektu než i386 a inline assembler v překladači GCC není úplně intuitivní, budeme v tomto a několika následujících cvičeních používat externí assembler *nasm* způsobem, jak jsme si ukázali v předchozím cvičení.

Připomeňme klíčovou část, tj. příklad funkce vracející hodnotu `42`.

```
; soubor demo.asm
global foo
section .text
foo:
        mov eax, 42
        ret
```

Na začátku máme komentář vyznačený znakem středník. Následuje direktiva `global`, která označuje jaké symboly (v terminologii C: funkce a proměnné) daný zdrojový soubor poskytuje, v našem případě to bude funkce `foo`. Následuje vyznačení sekce `.text`, která obsahuje kód programu zapsaný v jazyce symbolických adres. Naše funkce je vyznačena pomocí návěstí `foo:` (identifikátor + dvojtečka) a následuje kód samotné funkce.

Funkce se skládá ze dvou instrukcí. První instrukce uloží do registru `eax` návratovou hodnotu. Jedná se o konvenci, kdy celočíselné návratové hodnoty jsou předávány registrem `rax`, resp. `eax`, dle velikosti návratové hodnoty. Návrat z funkce je realizován instrukcí `ret`. Podrobněji se volání funkcí budeme věnovat na přednášce a v následujících cvičeních.

Překlad program v jazyce symbolických adres zajistíme příkazem `nasm`.

```
nasm -f elf64 demo.asm
```

2 Instrukční sada x86/AMD64

Instrukční sada procesorů x86/AMD64 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.[2]
- Brandejs M. Mikroprocesory Intel Pentium. Brno: Fakulta informatiky, Masarykova univerzita, 2010.[3]
- Přehled všech operací procesorů rodiny x86[4]. (Není potřeba znát.)

2.1 Registry

Procesory rodiny AMD64 nabízí následující registry:

[2] `http://phoenix.inf.upol.cz/~krajcap/courses/2021LS/OS1/Assembler.pdf`
[3] `http://www.fi.muni.cz/usr/brandejs/Brandejs_Mikroprocesory_Intel_Pentium_2010.pdf`
[4] `http://ref.x86asm.net/coder64.html`

- 64bitové: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` až `r15`,
- 32bitové: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `r8d` až `r15d`,
- 16bitové: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `r8w` až `r15w`,
- 8bitové: `ah`, `al`, `bh`, `bl` `ch`, `cl`, `dh`, `dl`, `r8b` až `r15b`.

Přičemž vzájemně si odpovídající skupiny registrů (např. `rax`, `eax`, `ah`, `al`) sdílí stejnou paměť.

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

2.2 Přehled základních aritmetických instrukcí

Instrukce mají obvykle tvar:

$$\langle \textit{ název instrukce } \rangle \langle \textit{ cílový operand } \rangle [\ , \langle \textit{ další operand } \rangle, \ldots\ ]$$

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[5].
- `i` – přímé hodnoty (konstanty)

Každá instrukce připouští jen určité kombinace operandů[6], 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
mul r/m               ; rdx:rax := rax * op1
imul r, r/m           ; op1 := op1 * op2
```

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

```
imul r, r/m, i        ; op1 := op2 * op3

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

Instrukce pro násobení a dělení jsou atypické v tom, že mají určené registry, se kterými pracují a ty jsou ještě ovlivněny velikostí operandu dané instrukce. To znamená, že pokud máme operand instrukce `mul` 32bitový, bude výsledek násobení uložen do dvojice registrů `edx` a `eax`, kde registr `eax` obsahuje spodních 32 bitů výsledku a registr `edx` horních 32 bitů. Máme-li operand 64bitový, bude výsledek uložen do dvojice registrů `rax`, `rdx`.

V případě delění je situace ošemetnější. Pokud je operand (tj. dělitel) 32bitový, dělí se obsah dvojice registrů `edx` a `eax`, podíl je uložen do registru `eax` a zbytek po dělení do registru `edx`. V případě, že je dělitel 64bitový, je postup analogický, jen s tím rozdílem, že jsou použity registry `rax` a `rdx`.

Z toho plyne, že při dělení nestačí nastavit jen obsah registru `eax` (resp. `rax`), vždy musíme korektně nastavit i hodnotu v registru `edx` (resp. `rdx`).[7]

3 Praktická práce s assemblerem

Programování na úrovni assembleru si můžeme vyzkoušet implementací jednoduchých funkcí, jak jsme si uvedli v části 1 a v předchozím cvičení. Princip implementace zůstává stejný, tj.

- Pomocí direktivy `global` určíme, jaké funkce jsou implementovány v assembleru.
- V sekci `.text` implementujeme jednotlivé funkce, které vyznačíme návěstím. Funkcí může být více.
- Funkce vrací výsledek v registru `eax` (resp. `rax`) a je ukončena instrukcí `ret`.
- V jazyce C definujeme prototypy daných funkcí a voláme je standardním způsobem.
- Při sestavování programu sloučíme kód v C a v assembleru.

Aby fukce mohly dělat něco smysluplného, je potřeba jim předat argumenty. Pokud budeme uvažovat unixový operační systém a funkce mající jen celočíselné argumenty, kterých není více než šest, můžeme předpokládat, že argumenty jsou uloženy v následujících registrech v tomto pořadí: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`. Dále platí, že obsah registrů: `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`, musí být na konci funkce stejný, jako na jejím začátku.

3.1 Ukázka použití

Nyní si ukážeme implementaci dvou funkcí, jedna bude inkrementovat hodnotu svého argumentu o jedna, druhá spočítá obvod obdélníka.

[7] Pokud pracujeme s kladnými čísly, stačí zajistit, že obsah `edx` (resp. `rdx`) bude 0.

3.1.1 Část v assembleru

Soubor: `tutorial03.asm`

```
global inc1
global rectangle_circumference

section .text

;;
;; funkce majici jeden argument, vracejici hodnotu o jedna vyssi
;;
inc1:
    mov eax, edi    ; presune prvni argument do registru eax
    add eax, 1      ; zvysi hodnotu o jedna
    ret             ; navrat z funkce

;;
;; vypocet obvodu obdelnika
;; funkce ma dva argumenty (velikost strany obdelnika)
;;
rectangle_circumference:
    mov eax, edi    ; ulozi do eax jednu stranu obdelnika
    add eax, esi    ; pricte druhou stranu
    add eax, eax    ; vynasobi dvema
    ret
```

3.1.2 Část v C

Soubor: `tutorial03-test.c`

```
#include <stdio.h>

/* prototypy funkci napsanych v assembleru */
int inc1(int arg);
int rectangle_circumference(int a, int b);

int main()
{
    printf("5 + 1: %i\n", inc1(5));
    printf("obvod obdelnika o stranach 4x5: %i\n", rectangle_circumference(4, 5));
    return 0;
}
```

3.1.3 Makefile

```
tutorial03-test: tutorial03-test.o tutorial03.o
    gcc -o tutorial03-test tutorial03-test.o tutorial03.o

tutorial03-test.o: tutorial03-test.c
    gcc -c tutorial03-test.c

tutorial03.o: tutorial03.asm
    nasm -f elf64 tutorial03.asm
```