Řízení výpočtu

4

V tomto cvičení si ukážeme, jak jsou na úrovni procesoru řešeny konstrukce pro řízení výpočtu, které můžeme znát z vyšších programovacích jazyků. Zejména si ukážeme, jak lze realizovat podmínky a cykly.

1 Skoky

Společným nástrojem pro implementaci řídících struktur jsou tzv. *skoky*, které mohou přenést provádění výpočtu na zadanou adresu. Připomeňme, že prováděný program je uložen v paměti jako data a registr `rip` ukazuje na adresu instrukce, která má být provedena jako další. Standardně jsou instrukce prováděny v řadě za sebou, avšak změnou hodnoty v registru `rip` můžeme určit, jaký kód se má provádět jako další. Ze své povahy registr `rip` patří mezi řídící registry a není možné jej měnit přímo, např. instrukcemi typu `mov`. Je proto potřeba použít speciálních instrukcí, které se označují jako skoky a které, obrazně řečeno, provedou „skok" na zadanou adresu v programu, odkud se bude další kód provádět. Skoky se dělí na dva základní typy: (i) *nepodmíněné* a (ii) *podmíněné*.

1.1 Nepodmíněné skoky

Nepodmíněné skoky vždy převedou řízení výpočtu na zadanou adresu. Na platformě x86 se pro nepodmíněné skoky používá instrukce `jmp r/m/i`, která má právě jeden operand, kterým je adresa, na kterou má být proveden skok.

Nejtypičtěji se instrukce `jmp` používá s operandem, kterým je konstanta udávající konkrétní adresu cíle skoku. Avšak při zápisu kódu v assembleru nevíme, na jakých adresách jsou (resp. budou) jednotlivé instrukce uloženy. Proto se namísto konkrétní adresy uvádí tzv. *návěští* (anglicky *label*), které představuje symbolické pojmenování adresy, kam může být provedený nějaký skok, a ve fázi překladu se assembler postará o korektní doplnění adresy.[1] Ukažme si to na příkladu.

```
1      mov eax, 0x42
2  foo:
3      inc eax
4      jmp foo
```

Na prvním řádku jsme do registru `eax` uložili hodnotu, ta pro tento příklad není důležitá. Na druhém řádku máme deklarováno návěští `foo`. Deklarace je ve tvaru *jméno následované dvojtečkou*.[2] Samotná de-

[1] S návěštími jsme se již setkali při vytváření funkcí, kdy návěští identifikují začátek (vstupní bod) funkce. V obou případech (ať už se jedná o skoky, či volání funkcí) návěští zastupují adresu v kódu programu.

[2] Pro lepší čitelnost se návěští píšou jako předsazená jednotlivým instrukcím

klarace návěští nemá žádný konkrétní vliv na vygenerovaný kód, je to jen pojmenování místa v kódu. Ve skutečném programu by mělo být pojmenování návěští voleno tak, aby bylo jasné, jaký význam má daný úsek kódu.[3]

Na řádku číslo 3 máme instrukci, která zvýší hodnotu registru `eax` o 1, opět to v našem ukázkovém příkladu nehraje zásadní roli. Na čtvrtém řádku máme instrukci `jmp`, která provede skok na adresu, která je dána návěštím `foo`.[4] Tj. znovu se provede instrukce `inc` a skok `jmp`. To se bude opakovat do doby, než program násilně ukončíme. Jak ukončit smyčku si ukážeme v následující kapitole.

Jak jsme již zmínili, nejčastěji se používá skok na konkrétní adresu, tj. kdy se instrukce `jmp foo` převede na tvar `jmp 0x12`. V případě architektury AMD64 se používají tzv. relativní skoky, kdy cílová adresa je vztažena k aktuální adrese, tj. k registru `rip`. Máme-li například instrukci `jmp 0x12`, bude skok provedený na adresu `rip + 0x12`.

Je možné použít variantu instrukce `jmp qword [rbx]`, kdy nejdříve ze zadané adresy (uložené v registru `rbx`) přečteme adresu, kam se má provést skok, a až na takto získanou adresu skok provedeme. Tato varianta umožňuje efektivně implementovat konstrukce typu `switch-case` nebo virtuální metody v objektově orientovaných jazycích.

Všimněme si, že instrukce nepodmíněného skoku umožňuje provést skok na libovolné místo v programu. To dává programátorovi velice široké možnosti pro jeho použití. Avšak při použití skoků je dobré se držet logického členění programu, které rámcově kopíruje bloky kódu, jak je známe z vyšších programovacích jazyků. V opačném případě hrozí, že program bude buď špatně srozumitelný nebo bude obsahovat chyby.

Z pohledu programování mají takto široce pojaté skoky koncepční nedostatky, viz legendární článek *E. Dijkstra: Go To Statement Considered Harmful*,[5] a proto dnešní programavocí jazyky skoky téměř nepoužívají nebo používají skoky jen s omezenými možnostmi. Na úrovni jednotlivých instrukcí procesoru se však skokům vyhnout nedá.

1.2 Podmíněné skoky

Druhou variantou, jak realizovat řízení výpočtu, jsou podmíněné skoky. To jsou skoky, k jejichž provedení dojde pouze v případě, že je splněna zadaná podmínka, jinak program pokračuje následující instrukcí. Podmínky, jež se využívají u podmíněných skoků, jsou určeny registrem `rf`, kde jsou mimo jiné čtyři jednobitové příznaky ZF, CF, SF, OF, které jsou nastaveny po provedení aritmetických operací.[6] Tyto příznaky mají následující význam:

- ZF (zero flag) – výsledek byl nula,
- SF (sign flag) – výsledek je nezáporný (0) nebo záporný (1),[7]
- CF (carry flag) – výsledek je větší nebo menší než největší/nejmenší možné číslo,
- OF (overflow flag) – příznak přetečení znaménkové hodnoty mimo daný rozsah.

[3] Všimněme si analogie s pojmenováváním proměnných ve vyšších programovacích jazycích.

[4] Při překladu assembler doplní konkrétní adresu.

[5] `https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf`

[6] Pozor, ne všechny instrukce nastavují všechny příznaky. Informace o tom, jaké příznaky jsou nastavovány, je potřeba čerpat z dokumentace procesoru.

[7] Odpovídá kopii nejvyššího bitu výsledku.

Pro každý z těchto čtyř příznaků existují dvě instrukce, které provedou skok, pokud je příznak nastaven, nebo nenastaven. Ukažme si to na příznaku ZF. Pro tento příznak existuje instrukce `jz`[8], která provede skok, pokud je příznak ZF nastaven na 1 (tj. předchozí operace skončila nulou), a dále existuje instrukce `jnz`[9], která skok provede, pokud je příznak ZF nastaven na 0 (tj. předchozí operace neskončila nulou). Použití ukazuje následující kód.

```
1       sub eax, 1
2       jz foo
3       inc ebx
4   foo:
```

Na prvním řádku jsme provedli aritmetickou operaci (odečetli jsme jedničku od registru `eax`). Tato operace nastavila příznaky v registru `rf` a pomocí instrukce `jz` můžeme otestovat, zda byl výsledek nula. Pokud ano, instrukce provede skok na místo v programu dané návěštím `foo`. V opačném případě se pokračuje následující instrukcí, což je v tomto případě instrukce na třetím řádku, která zvýší hodnotu v registru `ebx` o jedna.

Podobně existují podmíněné skoky pro příznaky SF (`js`, `jns`), CF (`jc`, `jnc`), OF (`jo`, `jno`).

Práce na úrovni jednotlivých příznaků a aritmetických operací není úplně komfortní a není s to podchytit řadu běžných situací, které v programech nastávají. Proto instrukční sada procesorů x86 obsahuje instrukci `cmp r/m, r/m/i`, která porovná dvě hodnoty tak, že je od sebe odečte a nastaví příznaky v registru `rf`.[10]

Vedle této instrukce existují dvě sady instrukcí podmíněných skoků, které slouží k porovnání celočíselných hodnot tak, jak se běžně používá ve vyšších programovacích jazycích. Dvě sady potřebujeme, protože musíme rozlišovat mezi tím, zda porovnáváme znaménkové nebo neznaménkové hodnoty.

Následující tabulka ukazuje instrukce pro porovnání znaménkových hodnot.

instrukce	alt. jméno	příznaky	podmínka
`jg`	`jnle`	(SF = OF) & ZF = 0	$A > B$
`jge`	`jnl`	(SF = OF)	$A \geq B$
`jl`	`jnge`	(SF $\neq$ OF)	$A < B$
`jle`	`jng`	(SF $\neq$ OF) nebo ZF = 1	$A \leq B$

Písmeno `g` v názvu instrukce značí *greater* (větší) a `l` značí *lesser* (menší), tj. `jg` odpovídá jump-if-greater apod.

Podobnou sadu podmíněných skoků pro porovnání neznaménkových hodnot popisuje následující tabulka.

instrukce	alt. jméno	příznaky	podmínka
`ja`	`jnbe`	(CF or ZF) = 0	$A > B$
`jae`	`jnb`	CF = 0	$A \geq B$
`jb`	`jnae`	CF = 1	$A < B$
`jbe`	`jna`	(CF or ZF) = 1	$A \leq B$

[8] jump-if-zero
[9] jump-if-not-zero
[10] Chová se podobně jako instrukce `sub`, avšak nemění žádný ze svých operandů.

Písmeno `a` v názvu instrukce značí *above* (větší) a `b` značí *below* (menší), tj. `ja` odpovídá jump-if-above apod.

Následující příklad ukazuje použití podmíněných skoků pro výpočet absolutní hodnoty.

```
1  ;;
2  ;; funkce pro vypocet absolutni hodnoty
3  ;; nazev `absi' zvolen z duvodu kolize s klicovym slovem `abs'
4  ;;
5  ;; int absi(int n);
6  ;;
7  absi:
8          mov eax, edi   ; precteni argumentu n
9          cmp eax, 0     ; porovnani s 0
10         jge konec      ; skok na konec funkce
11         neg eax        ; otoceni znameka
12 konec:
13         ret            ; navrat z funkce
```

Nejdříve si do registru `eax` uložíme argument funkce, číslo `n`, to je uloženo v registru `edi`. Následně hodnotu tohoto argumentu porovnáme s 0. Pokud je hodnota větší nebo rovna 0, provedeme skok na návěští `konec`. Připomeňme, že pracujeme se znaménkovou hodnotou, a proto jsme použili instrukci `jge`. Pokud je hodnota menší než 0, pokračujeme další instrukcí a provedeme negaci (řádek s číslem 11). Na konci se nám obě větve výpočtu opět spojí a provedeme návrat z funkce instrukcí `ret`.

V předchozí kapitole jsme viděli, jak lze pomocí instrukce nepodmíněného skoku implementovat nekonečnou smyčku. Pomocí podmíněného skoku však můžeme do takto sestrojené smyčky vložit koncovou podmínku a cyklus ukončit.[11] Jak může vypadat spojení podmíněného a nepodmíněného skoku ukazuje následující příklad funkce počítající faktoriál čísla `n`.

```
;;
;; funkce pro vypocet faktorialu
;; int fact(int n);
;;
fact:
     mov ecx, edi    ; ecx -- vstupni argument (n)
     mov eax, 0x1    ; eax -- strada vyslednou hodnotu tj. n * (n - 1) * (n - 2) * ... * 1
fact_loop:
     cmp ecx, 0x0
     jle konec       ; narazili jsme na konec, ukoncime funkci
     imul ecx        ; vynasob eax hodnotou n
     sub ecx, 0x1    ; opakuj pro n - 1
     jmp fact_loop
```

[11] Někdy se též říká „vyskočit z cyklu".

```
konec:
     ret
```

Praktická poznámka závěrem

Instrukce (podmíněných) skoků zásadním způsobem ovlivňují naplnění instrukční pipeline a tím rychlost zpracování programů, proto pokud můžeme instrukci skoku ušetřit, je to vždy vítané.

U začínajících programátorů[12] je možné najít konstrukce typu:

```
    cmp eax, 0
    jge foo
    jmp bar
foo:
    ;; nejaky kod
bar:
```

Takto napsaný kód sice nejspíš bude provádět, co jeho autor zamýšlel, ale není úplně dobrý, protože jednak bude tento kód špatně čitelný a také obsahuje instrukci navíc. Můžeme jej totiž přepsat do podoby.

```
    cmp eax, 0
    jl bar
foo:
    ;; nejaky kod
bar:
```

2 Úkoly k procvičení

1. Napište funkci `int sgn(int i)`, která vrací hodnoty -1, 0, 1 v závislosti na tom, zda-li je hodnota i záporná, nulová nebo kladná.
2. Napište funkci `char max2c(char a, char b)`, která vrací největší hodnotu. Vyzkoušejte, že funkce funguje správně pro kladné i záporné argumenty, i jejich kombinaci.
3. Napište funkci `unsigned short min3us(unsigned short a, unsigned short b, unsigned short c)`, která vrací nejmenší hodnotu ze zadaných parametrů. Vyzkoušejte, že funkce funguje správně i pro hodnoty větší než 32768.
4. Napište funkci `int kladne(int a, int b, int c)`, která vrací 1, pokud jsou všechny argumenty kladné, jinak 0.
5. Napište funkci `int mocnina(int n, unsigned int m)` vracející mocninu n^m.
6. Do registrů `al`, `bl` vložte vhodné hodnoty, proveďte s nimi operace add a sub a pomocí instrukcí `jz`, `js`, `jc` a `jo` ověřte, zda byl nastavený příznak, nebo ne.

[12] Resp. těch, kteří teprvé získávají zkušenost s programovaním v assembleru.