

PB111 Nízkoúrovňové programování (výpočetní stroj)

P. Ročkal

Část A: Úvod 1

Část 1: Výpočetní stroj 3

Část 2: Lokální proměnné, řízení toku 5

Část A: Úvod

Stav výpočetního stroje, se kterým budeme v tomto předmětu pracovat, je velmi jednoduchý. Skládá se z:

- šestnácti registrů, každý o šířce 16 bitů:
 - registr `rv` (return value),
 - registry `l1` až `l7` (local),
 - registry `t1` až `t6` (temporary),
 - registry `bp` a `sp`,
- speciálního 16bitového registru `pc` (program counter),
- 64 KiB paměti adresované po slabikách (bajtech) – adresa je tedy 16bitové celé číslo (bez znaménka), které přesně určuje právě jednu paměťovou buňku, přitom každá taková buňka obsahuje celé číslo v rozsahu 0 až 255.

Sémanticky speciální jsou pouze registry `pc` a `sp` – všechny ostatní jsou z pohledu stroje ekvivalentní a jejich jména nemají pro samotný výpočet žádný speciální význam – jedná se pouze o konvenci, která nám usnadní čtení (a psaní) programů.

Výpočet stroje probíhá takto:

- z adresy uložené v registru `pc` se načtou dvě šestnáctibitová slova – `hi` z adresy `pc` a `lo` z adresy `pc + 2` – která kódují jednu instrukci,
- instrukce je strojem dekódována a provedena:
 - slovo `hi` kóduje operaci (vyšší slabika), cílový registr a první registrový operand,
 - slovo `lo` kóduje přímý (immediate) operand, nebo druhý registrový operand (v nejvyšší půlslabice),
 - provede se efekt instrukce (tento efekt samozřejmě závisí jak na operaci, tak na operandech) – obvykle je součástí tohoto efektu změna hodnoty uložené v registru `pc`,
- nebyl-li výpočet zastaven, pokračuje bodem 1.

Registry jsou očíslovány v pořadí uvedeném výše, totiž `rv` je registr číslo 0 a `sp` je registr číslo 15. Je vidět, že číslo registru lze zakódovat do jedné půlslabiky (registr `pc` operandem být nemůže).

Následuje výčet všech operací, které umí stroj provést. Nebudeme všechny operace potřebovat hned, a nebudeme se tedy zatím ani podrobněji zabývat

jejich sémantikou – tu si rozebereme vždy na začátku kapitoly, v níž začínou být tyto operace relevantní.

- speciální operace:
 - práce se zásobníkem (`push`, `pop`),
 - nastavení registru na konstantu (`put`),
 - nastavení registru na hodnotu z jiného registru (`copy`),
 - znaménkové rozšíření bajtu (`sext`),
- operace pro práci s pamětí:
 - kopírování dat z paměti do registru (`ld`, `ldb`),
 - kopírování z registru do paměti (`st`, `stb`),
- aritmetické operace:
 - aditivní – bez rozlišení znaménkovosti (`add`, `sub`),
 - násobení `mul`,
 - dělení se znaménkem (`sdiv`, `smod`),
 - dělení bez znaménka (`udiv` a `umod`),
- operace pro srovnání dvou hodnot:
 - rovnost (`eq`, `ne`),
 - znaménkové ostré nerovnosti (`slt`, `sgt`),
 - znaménkové neostré nerovnosti (`sle`, `sge`),
 - bezznaménkové ostré (`ult`, `ugt`), a konečně
 - bezznaménkové neostré (`ule`, `uge`),
- bitové operace:
 - logické operace `and`, `or` a `xor` aplikované po bitech,
 - bitové posuvy `shl` (levý), `shr` (pravý) a aritmetický `sar`,
- řízení toku:
 - nepodmíněný skok `jmp`,
 - podmíněné skoky `jz` (jump if zero) a `jnz` (if not zero),
 - volání a návrat z podprogramu (`call`, `ret`),
- ovládání stroje:
 - `halt` zastaví výpočet,
 - `asrt` zastaví výpočet s chybou, je-li operand nulový.

A.1: Jazyk symbolických adres

Stroj jako takový pracuje pouze s **číselnými** adresami – instrukce, která

obsahuje adresu, ji vždy obsahuje jako číslo. To při programování představuje značný problém, protože adresy jednotlivých částí programu závisí na tom, kolik instrukcí se nachází v části předchozí. Uvažme třeba tento program (uložený v paměti od adresy nula):

```
put 0    → rv ; vynuluj registr rv
add 1, rv → rv ; do registru rv přičti 1
jnz rv, 0x0004 ; je-li rv nenulové, skoč na adresu 4
```

Protože každá instrukce je kódována do 4 bajtů, adresa druhé instrukce (operace `add`) je 4 (její kódování je uloženo na adresách 4, 5, 6 a 7). Program jak je napsaný provede prázdný cyklus 65535× (v poslední iteraci je v registru `rv` hodnota `ffff`, přičtením jedničky se změní na nulu, podmíněný skok „není-li `rv` nula“ se neprovede a cyklus tak skončí).

Uvažme nyní situaci, kdy do programu potřebujeme (na začátek) zařadit další instrukci, např. nastavení registru 11:

```
put 0    → l1 ; vynuluj registr l1
put 0    → rv ; vynuluj registr rv
add 1, rv → rv ; do registru rv přičti 1
jnz rv, 0x0004 ; je-li rv nenulové, skoč na adresu 4
```

Tím se ale posunuly všechny další instrukce v programu na jiné adresy – proto adresa skoku předaná operaci `jnz` neodpovídá původnímu programu – tento nový program bude cyklit donekonečna (rozmyslete si proč).

Je asi zřejmé, že kdyby měla každá změna programu (přidání nebo odebrání instrukce) znamenat, že musíme opravit všechny adresy ve všech ostatních instrukcích, moc dobře by se nám neprogramovalo. Proto pro zápis strojového kódu používáme tzv. jazyk **symbolických adres**. Ten nám umožňuje místa v programu – adresy – pojmenovat **symbolem** – textovým názvem, podobně jako nazýváme třeba proměnné v jazyce Python. Symbol zavedeme tzv. **návěstím** a použijeme v zápisu instrukce¹ na místě adresy:

¹ Striktně vzato se v takové chvíli nejedná o zápis instrukce, pouze o předpis, jak konkrétní instrukci dopočítat – protože je to ale výpočet velmi jednoduchý, nebudeme obvykle tyto případy rozlišovat (tzn. návěstí budeme přímo interpretovat jako adresu, kterou reprezentuje v daném programu).

```
put 0 → rv ; vynuluj registr rv
loop:      ; návěstí pro první instrukci cyklu
add 1, rv → rv ; do registru rv přičti 1
jnz rv, loop ; je-li rv nenulové, skoč na začátek cyklu
```

Když nyní přidáme na začátek programu instrukci, nic špatného se nestane – při sestavení (angl. **assembly**) programu se pak do podmíněného skoku místo adresy 4 doplní adresa 8 – totiž adresa instrukce, která bezprostředně následuje za návěstím.

Část 1: Výpočetní stroj

V této kapitole budeme potřebovat 2 typy instrukcí – výpočetní (aritmetické, logické, atp.) a instrukce pro řízení toku (nepodmíněné a podmíněné skoky). Zejména prozatím nebudeme potřebovat pracovat s adresami, pamětí obecně, ani zásobníkem.

1 Kopírování hodnot Nejzákladnější operací, kterou můžeme v programu potřebovat, je nastavení registru, a to buď na předem známou konstantu, nebo na hodnotu aktuálně uloženou v některém jiném registru.

K nastavení registru na konstantu můžeme použít operaci `put`, která nastaví výstupní registr na hodnotu přímého operandu. Zápis této instrukce bude vypadat např. takto:

```
put 13 → rv
put 0x70 → l1
halt
```

Tento program nastaví registr `rv` na hodnotu 13 a registr `l1` na hodnotu 112.

Pro kopírování hodnot mezi registry použijeme operaci `copy` – ta nastaví výstupní registr na tutéž hodnotu, jakou má registr vstupní. Například:

```
put 13 → rv
put 17 → l1
copy rv → l2 ; sets l2 = 13
copy l1 → rv ; sets rv = 17
halt
```

Po provedení tohoto programu budou hodnoty registrů `rv = 17`, `l1 = 17` a `l2 = 13`.

2 Aritmetika Další důležitou kategorií jsou aritmetické instrukce. Následující tabulka shrnuje operace, které máte k dispozici. Registr `l1` odpovídá proměnné `a`, registr `l2` proměnné `b`, registr `rv` pak proměnné `x`.

název	python	tiny
sčítání	<code>x = a + b</code>	<code>add l1, l2 → rv</code>
odečítání	<code>x = a - b</code>	<code>sub l1, l2 → rv</code>
násobení	<code>x = a * b</code>	<code>mul l1, l2 → rv</code>
dělení	<code>x = a // b</code>	<code>sdiv l1, l2 → rv</code> <code>udiv l1, l2 → rv</code>
zbytek	<code>x = a % b</code>	<code>smod l1, l2 → rv</code> <code>umod l1, l2 → rv</code>

Všimněte si, že operaci celočíselného dělení a zbytku po dělení odpovídají

dvě různé instrukce. Je to proto, že fyzicky jsou registry realizované jako sekvence binárních přepínačů – každý přepínač reprezentuje jeden bit. Tyto binární sekvence lze interpretovat různými způsoby, nicméně b -bitový registr obvykle chápeme jako:

- celé číslo n bez znaménka v rozsahu $(0, 2^b)$ – pak sekvence bitů přímo odpovídá binárnímu zápisu tohoto čísla,
- jako celé číslo s se znaménkem v rozsahu $(-2^{b-1}, 2^{b-1})$, a to tak, že:
 - je-li nejvyšší bit nastaven na 1, $s = n - 2^b$,
 - jinak $s = n$

Podmínku z bodu (a) můžeme také chápat jako $[n \geq 2^{b-1}]$.

Pro 16bitová čísla, která budeme v tomto předmětu používat zdaleka nejčastěji, to jsou tyto rozsahy:

- `(0, 65535)` (nebo `0xffff` v šestnáctkovém zápisu) pro reprezentaci bez znaménka,
- `(-32768, 32767)` (nebo `-8000` až `7fff` šestnáctkově) pro reprezentaci se znaménkem.

Tato reprezentace má tu vlastnost, že sčítání, odečítání a násobení používá na úrovni bitů stejný algoritmus v obou případech – proto operace `add` funguje stejně dobře bez ohledu na to, chápeme-li operandy jako znaménkové nebo bezznaménkové.

To ale neplatí pro dělení (a nebude to platit ani pro srovnání, jak uvidíme za chvíli) – výsledek se bude lišit v závislosti na tom, je-li operace znaménková (`sdiv`, `smod`) nebo nikoliv (`udiv`, `umod`).

3 Srovnání Prakticky každý vyšší programovací jazyk má nějakou formu **podmíněného příkazu**. Aby byla tato konstrukce užitečná, potřebujeme mít k dispozici **predikáty** – operace, kterých výsledkem je pravdivostní hodnota. Ty nejběžnější již dobře znáte – jsou to celočíselné srovnávací operátory. V Pythonu je zapisujeme jako `a == b`, `a < b`, atp.

Náš výpočetní stroj má pro tento účel sadu operací – jsou shrnuty v tabulce níže. Jak již bylo výše naznačeno, s výjimkou rovnosti musíme rozlišovat znaménkovou a bezznaménkovou verzi. Na rozdíl od Pythonu (nebo jazyka C) nemá strojový kód složené výrazy, proto musíme výsledek srovnání vždy uložit do registru (analogem v Pythonu je booleovská proměnná – budeme ji zde opět značit `x`).

python	tiny	
<code>x = a == b</code>	<code>eq l1, l2 → rv</code>	equal
<code>x = a != b</code>	<code>ne l1, l2 → rv</code>	not equal
<code>x = a < b</code>	<code>slt l1, l2 → rv</code>	signed less than
	<code>ult l1, l2 → rv</code>	unsigned less than
<code>x = a > b</code>	<code>sgt l1, l2 → rv</code>	signed greater than
	<code>ugt l1, l2 → rv</code>	unsigned greater than
<code>x = a <= b</code>	<code>sle l1, l2 → rv</code>	signed less or equal
	<code>ule l1, l2 → rv</code>	unsigned less or equal
<code>x = a >= b</code>	<code>sge l1, l2 → rv</code>	signed greater or equal
	<code>uge l1, l2 → rv</code>	unsigned greater or equal

Výsledek uložený do výstupního registru (v příkladech výše `rv`) je u instrukcí z této rodiny vždy 1 (pravda) nebo 0 (nepravda). To zejména znamená, že je možné tyto výsledky kombinovat operacemi `and`, `or` a `xor` a výsledek bude vždy opět 0 nebo 1, v souladu s definicí příslušné logické operace (k těmto se vrátíme níže).

4 Řízení toku Abychom mohli realizovat podmíněné příkazy a cykly, budeme k tomu potřebovat speciální operace – podobně jako příslušným příkazům ve vyšším jazyce jim budeme říkat **řízení toku**.

Výpočetní stroj `tiny` obsahuje 3 operace tohoto typu:

- `jmp addr` způsobí, že výpočet bude pokračovat od adresy `addr` – bez ohledu na aktuální stav registrů; adresu můžeme (a typicky budeme) zadávat jako **symbol** (jméno **návěští** – viz též část B.3),
- `jz reg, addr` (jump if zero) nejprve ověří, je-li hodnota registru `reg` nulová – pokud ano, provede skok stejně jako `jmp addr`, v případě opačném pokračuje na další instrukci bez jakéhokoliv dalšího efektu,
- `jnz reg, addr` (jump if not zero) se chová stejně, ale skok provede pouze je-li hodnota uložená v `reg` nenulová.

V kombinaci s aritmetickými a srovnávacími operacemi popsanými výše dokážeme zapsat jednoduchou podmínku např. takto (odpovídající program v Python-u je uveden v komentářích):

```
put 1 → l1 ; a = 1
slt l1, 3 → t1 ; t = a < 3
jz t1, else ; if t:
then:
    put 2 → l2 ; b = 2
    jmp endif ; else:
else:
```

```
put 3    → 12 ;    b = 3
endif:
halt
```

Zkuste si program spustit pomocí `tinym.py` z kapitoly B, a také upravit první instrukci na `put 5 → 11` a srovnajte výsledek. Podobně můžeme zapsat také `while` cyklus (cykly `for` do strojového kódu přímo přepsat nemůžeme, ale jak jistě víte, je vždy možné nejprve je přepsat na cykly `while`). Uvažme tento velmi jednoduchý program v Pythonu:

```
a = 1
while a < 3:
    a += 1
```

Přepis do strojového kódu bude opět vyžadovat určitou kreativitu, protože máme pouze instrukce skoku, nikoliv instrukce cyklu. Stačí si ale uvědomit, že `while True` se realizuje snadno: pomocí nepodmíněného skoku zpět (na nižší adresu).

```
put 1    → 11 ; a = 1
loop:    ; while True:
slt 11, 3 → t1 ; t = a < 3
jz t1, end ; if not t: break
add 11, 1 → 11 ; a += 1
jmp loop
end:
halt
```

Cyklus `while podmínka` jsme přepsali na `while True` a podmíněný `break` – ekvivalenci těchto dvou zápisů si rozmyslete.

5 Bitové logické operace XXX

6 Bitové posuvy XXX

Část 2: Lokální proměnné, řízení toku

V této kapitole žádné nové operace potřebovat nebudeme – budeme se soustředit na jazyk C a jak se jeho základní konstrukce přeloží na operace, které známe z předchozí kapitoly.

