Nízkoúrovňové programování

Instrukční sada ARMv7 (další instrukce)

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

Kontext

- tradiční RISC – jednoduché instrukce, které dělají jednu jednoduchou operaci, ale mají více použití
- ARMv7 – relativně jednoduché instrukce, mohou provést více operací současně
- barrel shifter – před posledním operandem
- operace s pamětí – čtení + změna operandu
- více čtecích/zapisovacích operací současně (práce se zásobníkem)

Barrel shifter před posledním operandem

- viděli jsme
  - konstanty zakódovány jako 8 bitů hodnota (imm) + 4 bity rotace doprava (rr)
  - hodnota konstanty je dána jako $imm \text{ ROR } (rr \times 2)$
- podobný princip aplikován i na registry
- instrukce typu:
  - `INST Rd, Rn, Rm, <typ> imm5`
  - `INST Rd, Rn, Rm, <typ> Rs`
- `typ` může být:
  - LSL – logický bitový posun vlevo (logical shift left)
  - LSR – logický bitový posun vpravo (logical shift right); na uvolněné místo vloženy 0
  - ASR – aritmetický bitový posun vpravo (arithmetic shift right); na uvolněné místo vložena kopie nejvyššího bitu
  - ROR – rotace vpravo (rotate right)
- `imm5` a `Rs` určují počet bitů posunu/rotace

Použití bitových operací v posledním operandu

- posun o 1 bit vlevo = vynásobení dvěma
- posun o 1 bit vpravo = dělení dvěma
- více bitů odpovídá násobení/dělení mocninou dvou
- výpočet adres při přístupu do paměti
- násobení konstantami
- bitové operace (AND, OR, EOR)

Příklady

```
add r0, r1, r2, lsl #1  // r0 := r1 + (r2 * 2)
sub r0, r1, r2, asr #1  // r0 := r1 - (r2 / 2)
mov r0, r0, lsl #2      // r0 := r0 * 4
```

Násobení konstantou

- instrukce `smul`, `umul` relativně pomalé
- lze nahradit aritmetickými operacemi + bitovými posuny

Specifické případy

```
mov r0, r0, lsl #2      // r0 := r0 * 4
add r0, r1, r1, lsl #2  // r0 := r1 + (r1 * 4), resp. r0 := r1 * 5
```

Obecné řešení

- násobení vyjádříme jako postupné sčítání mocnin dvou
- např. $n \times 12 = n \times 8 + n \times 4 = n \times 2^3 + n \times 2^2$.

```
lsl r1, r0, #2          // bitový posun vlevo o 2 bity, tj. r1 := r0 * 4
add r1, r1, r0, lsl #3  // r1 := r1 + r0 * 8
```

Přístup do paměti

- *load and store* architektura
- přístup na nezarovnané adresy může skončit výjimkou (cf. x86)
- oddělené instrukce pro přístup do paměti (cf. x86)
- operace mohou měnit indexovací registr
- pracuje se s celými registry, tyto operace řeší rozšíření a zúžení typů
- instrukce `STRB`, `STRH`, `STR` – zapisují do paměti 1, 2, 4 B
- instrukce `LDR` – přečte celé 32bitové slovo
- instrukce `LDRSB`, `LDRSH` – přečte 1 nebo 2 byty a rozšíří je na 32bitovou znaménkovou hodnotu
- instrukce `LDRB`, `LDRH` – přečte 1 nebo 2 byty a rozšíří je na 32bitovou neznaménkovou hodnotu

Typy adres

Základní typy adres

- `[Rn]` – pracuje s hodnotou na adrese `Rn`
- `[Rn, +/- Rm]` – pracuje s hodnotou na adrese `Rn +/- Rm`
- `[Rn, +/- Rm, shift]` – pracuje se s hodnotou na adrese `Rn +/- (Rm << shift)`
- `[Rn, +/- imm12]` – pracuje se s hodnotou na adrese `Rn +/- imm12)`

Se změnou indexu

- pre-index:
  `[Rn, <varianta>]!` // jako předchozí varianty, ale adresa je uložena do registru `Rn`
- post-index:
  `[Rn], <varianta>` // pracuje s hodnotou v paměti na adrese `Rn` a následně změní hodnotu v tomto registru podle použité varianty

Příklady použití (1/3)

```
// void my_strcpy(char *dest, char *src);
my_strcpy:
     mov r2, #0          // r2: index
my_strcpy_loop:
     ldrb r3, [r1, r2]   // r3 := *(r1 + r2)
     strb r3, [r0, r2]   // *(r0 + r2) = r3
     cmp r3, #0          // konec retezce?
     beq my_strcpy_done
     add r2, r2, #1      // posun na dalsi znak
     b my_strcpy_loop
my_strcpy_done:
     bx lr
```

Příklady použití (2/3)

```
// void my_strcpy(char *dest, char *src);
my_strcpy:
    ldrb r2, [r1], #1 // r2 := *r1; r1 := r1 + 1
    strb r2, [r0], #1 // *r0 := r2; r0 := r0 + 1
    cmp r2, #0        // konec retezce?
    bne my_strcpy     // pokracuje
    bx lr
```

Příklady použití (3/3)

```
// short sum_short(int count, short *array);
sum_short:
    mov r2, #0          // r2 -- celkovy soucet
sum_loop:
    ldrsh r3, [r1], #2  // precte hodnotu a posune se na dalsi
    add r2, r2, r3      // prida k celkovemu souctu
    subs r0, r0, #1     // snizi pocitadlo
    bne sum_loop        // pokud neni 0, pokracuje
    mov r0, r2
done:
    bx lr
```

Hodnocené úkoly

- Napište funkci `int mul11(int x)`, která vrátí hodnotou svého argumentu vynásobenou jedenácti. Pro výpočet použijte instrukce bitových posunů.
- Napište funkci `char fnzb(int x)`, která vrátí číslo nejvyššího nenulového bitu.
- Napište funkci `void *my_memset(void *s, int c, size_t n)`, která se bude chovat jako standardní funkce `memset`.
- Napište funkci `int my_strlen(char *)`, která se bude chovat jako standardní funkce `strlen`.
- Napište funkci `int my_strcmp(char *, char *)`, která se bude chovat jako standardní funkce `strcmp`.
- Napište funkci `void multiples(short *shorts, short n)`, která do pole `shorts` uloží prvních deset násobků čísla `n`.
- Napište funkci `int minimum(int count, int *values)`, která vrací nejmenší prvek pole values obsahující `count` hodnot.