Nízkoúrovňové programování

Instrukční sada ARMv7 (volání funkcí)

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

Volání funkcí

- návratová adresa uložena v běžném registru (R14, LR)
- optimalizace pro listové funkce (nevolají další funkci, není potřeba ukládat navrátovou adresu na zásobník)
- lze použít cokoliv, co odpovídajícím způsobem nastavuje registry (LR a PC)
- pozor, nejnižší bit adresy udává typ kódování instrukce (ARM/Thumb)
- instrukce `BL` (branch with link) a `BLX` (branch with link and exchange)
- `BLX` mění typ kódování instrukcí
- `BL` má smysl v rámci vlastního kódu, kde garantujeme stejné kódování instrukcí

Volací konvence

- první čtyři argumenty předávány registry R0, R1, R2, R3
- další argumenty přes zásobník zprava doleva
- návratové hodnota v R0 (příp. i v R1)
- R0, . . . , R3 (caller-saved, scratch registry)
- R4,. . . , R8, R10 (callee-saved, obecně použitelné pro uložení proměnných)
- R9 – platformě specifický registr
- R11 (FP) – ukazatel na rámec (callee-saved, možné použít i pro proměnné)
- R12 (IP) – určen pro použití linkerem (lze použít i jako scratch registr)
- R13 (SP) – stack pointer
- R14 (LR) – link register

Zásobník

- zásobník roste zhora dolů (od vyšších adres k nižším)
- hodnota v registru SP musí být násobkem 4
- pro veřejné rozhraní musí platit, že SP je násobkem 8
- přístup k zásobníku musí být z rozsahu [SP, stack-base - 1] (tj. nelze přistupovat na adresu nižší než SP)
- instrukce `PUSH {Ri, Rj, ...}` a `POP {Ri, Rj, ...}` uloží a načtou ze zásobníku zvolené registry
- speciální případ instrukcí `LDMxx` a `STMxx`
- registry ukládány na zásobník podle svých čísel zprava doleva
- `push {r0, r1, r2}`

  =

  ```
  push {r2}
  push {r1}
  push {r0}
  ```

Volání funkcí

```
@ foo(1, 2, 3, 4, 5, 6, 7)

sub     sp, sp, #16      @ prostor pro argumenty (zarovnani na 8B)
mov     r3, #7
str     r3, [sp, #8]     @ posledni argument (7)
mov     r3, #6
str     r3, [sp, #4]     @ predposledni argument (6)
mov     r3, #5
str     r3, [sp]         @ pred-predposledni argument (5)
mov     r3, #4           @ argumenty predavane pres registry
mov     r2, #3
mov     r1, #2
mov     r0, #1
bl      funkce           @ funkce
add     sp, sp, #16      @ odstraneni argumentu (5-7) ze zasobniku
```

Prolog a epilog funkcí

- ve srovnání s jinými architekturami volnější pravidla (platforma si může definovat striktnější nebo volnější požadavky)
- jednodušší funkce nemusí ukládat a nastavovat FP
- listové funkce nemusí ukládat LR
- obecně se předpokladájí dvě hodnoty (FP a LR) na zásobníku
- umožňuje to procházet, jak jsou jednotlivé funkce volány

Prolog

- uložení registrů:
  `push {fp, lr}` // nelistová funkce mající rámec
  `push {r4, r5, r6, r7}` // listová funkce pracující s registry R4, . . . , R7
  `push {r4, r5, fp, lr}` // nelistová funkce pracující s registry R4 a R5

- FP ukazuje na začátek aktuálního rámce na zásobníku (FP = SP + počet uložených registrů $\times 4 - 4$)
  `add fp, sp, #0` // pokud se ukládal jen FP
  `add fp, sp, #4` // pokud se ukládaly jen FP a LR
  `add fp, sp, #12` // pokud s ukládaly např. R4, R5, FP a LR

- vytvoření prostoru pro lokální proměnné
  `sub sp, sp, #8` // velikost + prostor pro zarovnání zásobníku na 8 B

Struktura zásobníku

```
                        ...
FP + 8                  arg #6
FP + 4          -->     arg #5
                        ------------
FP              -->     návratová adresa (LR)
FP - 4          -->     předchozí FP
FP - 8          -->     uložený registr Rj
FP - 12         -->     uložený registr Ri
                        ...
FP - n * 4      -->     lokální proměnné
```

Epilog

- odstranění lokálních proměnných ze zásobníku (opak prologu)
  `add sp, sp, #n`
- obnovení registrů, analogicky `PUSH` v prologu
- návrat z funkce – uložíme LR do PC
  - pokud je LR na zásobníku, lze řešit pomocí `POP`
  - jinak použijeme `BX LR`, (příp. `B LR`)

Příklad 0: Faktoriál

```
fact:
     push {fp, lr}       @ vytvoreni ramce
     add fp, sp, #4      @ nastaveni zacatku ramce (obsahuje FP a LR)
     sub sp, sp, #8      @ vytvoreni prostoru pro lok. promenne

     cmp r0, #0          @ koncova podminka
     beq fact_trivial

     str r0, [fp, #-8]   @ ulozeni argumentu n na zasobnik
     sub r0, r0, #1      @ zavolani pro n-1
     bl fact

     ldr r1, [fp, #-8]   @ obnova hodnoty ze zasobniku
     mul r0, r1, r0      @ n * fact(n - 1)
     b fact_done         @ ukonceni funkce
fact_trivial:
     mov r0, #1          @ navratova hodnota pro koncovou podminku
fact_done:
     add sp, sp, #8      @ odstraneni hodnot ze zasobniku
     pop {fp, pc}        @ navrat z funkce
```

Příklad 1: Faktoriál

```
fact:
    push {r4, lr}        @ ulozeni pracovniho registru a navrat adresy
    cmp r0, #0           @ koncova podminka
    beq fact_trivial

    mov r4, r0           @ ulozeni argumentu n
    sub r0, r4, #1       @ zavolani pro n - 1
    bl fact

    mul r0, r0, r4       @ n * fact(n-1)

    pop {r4, pc}         @ navrat

fact_trivial:            @ navratova hodnota pro koncovou podminku
    mov r0, #1
    pop {r4, pc}
```

Příklad 2: Faktoriál

```
fact:
    cmp r0, #0          @ test na koncovou podminku
    beq fact_trivial    @ pro n == 0, listova funkce

    push {r4, lr}       @ ulozeni pracovniho registru a navratove adresy

    mov r4, r0          @ ulozeni argumentu n
    sub r0, r4, #1      @ zavolani pro n - 1
    bl fact

    mul r0, r0, r4      @ n * fact(n-1)
    pop {r4, pc}        @ navrat

fact_trivial:
    mov r0, #1          @ navratova hodnota pro koncovou podminku
    bx lr
```

Hodnocené úkoly

- Vyzkoušejte si různé strategie pro vytvoření prologu a epilogu funkcí.
- Napište funkci void `void print_row(int n, char c)`, která s pomocí volání funkce `putchar` ze standardní knihovny vypíše na standardní výstup řádek skládající se z `n` opakování znaku `c`. Výpis by měl být ukončen znakem `\n`.
- Napište funkci void `print_rect(int rows, int cols)`, která s pomocí volání funkce `print_row` vykreslí na standardní výstup vyplněný obdélník skladající se ze znaků `*` mající `rows` řádků a `cols` sloupců.
- Napište funkci `int fib(int n)`, která rekurzivně vypočítá hodnotu n-tého fibonacciho čísla.
- Napište funkci `int fact_tc(int n, int r)`, která bude počítat faktoriál koncově rekurzivní (iterativní) metodou. Místo volání rekurzivního kroku použijte skok.