Podprogrami v RISC-V

Podprogrami

Podprogram je del programa, ki opravlja specifično nalogo na osnovi parametrov oz. argumentov, ki mu jih poda klicoči program oz. klicatelj.

- Podprogramu rečemo tudi *procedura*, v jeziku C pa se imenuje *funkcija*

Pri klicu podprograma so potrebni naslednji koraki:

1. Klicatelj postavi parametre nekam, kjer podprogram lahko dostopa do njih.
2. Klicatelj prenese kontrolo na podprogram.
3. Podprogram pridobi potrebne pomnilniške vire.
4. Podprogram izvede želeno nalogo.
5. Podprogram da rezultat nekam, kjer ga lahko klicatelj dobi.
6. Vrnitev na naslednji ukaz klicatelja.

Sklad

- Struktura, ki je najbolj primerna za ‘razlivanje’ (spill) registrov v pomnilnik, je sklad.
  - Sklad je podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
  - Za delo s skladom potrebujemo **skladovni kazalec** (SP, stack pointer): kazalec, ki kaže na vrh sklada.
  - Pri RISC-V ima vlogo skladovnega kazalca register **x2**, njegovo ABI ime je **sp**

- Sklad največ uporabljamo v podprogramih za:
  - shranjevanje povratnega naslova
  - prenos parametrov v podprograme
  - shranjevanje lokalnih spremenljivk
  - shranjevanje registrov

Dogovor o klicu podprogramov (calling convention)

- 8 registrov (x10–x17) uporablja za parametre oz. argumente procedur:
  - x10-x17 (a0-a7) za argumente
  - x10-x11 (a0-a1) za vrnjene vrednosti
  - Če prevajalnik potrebuje več kot 8 registrov za argumente, jih lahko poda prek sklada.
- Ukaz JAL je primeren za podprograme. Shrani povratni naslov v register za povratni naslov (x1 oz. ra, return address) in skoči na podani naslov

```
jal x1, NaslovProcedure
```

- Za vrnitev se uporablja indirektni skok JALR:

```
jalr x0, 0(x1)    # skoči na naslov v x1 oz. ra
```

(Ripes: jalr x0, ra, 0)

isto dela tudi prevdoukaz `ret`

- *Dogovor o klicih je potreben predvsem zaradi združljivosti (kompatibilnosti) z drugimi programi*

- Običajno sklad raste iz višjih naslovov proti nižjim (ni pa nujno - lahko gre k višjim, kopica pa k nižjim).
- Poleg tega SP lahko kaže na prvo prosto mesto na skladu, ali pa na zadnji podatek.

Zato so možne 4 variante (kombinacije gornjih 2 opcij):

1. Sklad narašča v smeri naraščajočih naslovov in SP kaže na prvo prosto mesto na skladu:
2. Sklad narašča v smeri naraščajočih naslovov in SP kaže na zadnji podatek na skladu:
3. Sklad narašča v smeri padajočih naslovov in SP kaže na prvo prosto mesto na skladu:
4. Sklad narašča v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu:

- Dogovor o klicih podprogramov (calling convention) za RISC-V pravi, naj
  - sklad narašča v smeri padajočih naslovov in naj
  - skladovni kazalec kaže na zadnji podatek na skladu.

- Sklad se torej začne na dnu podatkovnega segmenta – na naslovu, ki sicer ni del sklada.
- Na začetku vsakega programa moramo nastaviti skladovni kazalec
  - npr.: `addi sp,x0,0x500` (oz. `li sp,0x500`), če sami pišemo program
  - Bolj običajno pa je, da začetna koda iz skripte povezovalnika (linker) prebere, kje je konec podatkovnega segmenta in inicializira skladovni kazalec tam

Operaciji PUSH in POP:

- PUSH ‘porine’ register na sklad
- POP ‘obnovi’ register s sklada

RISC- V nima ukazov PUSH/POP.

- Makro ukaza PUSH in POP naj imata samo en eksplicitni registrski operand.
- Izvedemo ju kot zaporedje dveh ukazov procesorja:

```
PUSH: addi sp, sp, -4        # oz. addi x2,x2,-4
      sw reg, 0(sp)

POP:  lw reg, 0(sp)
      addi sp, sp, 4
```

Če porinemo na sklad več vrednosti, lahko to naredimo z enim samim pomikom sp.

- Npr., za 3 vrednosti:

```
addi sp, sp, -16   # sp mora biti poravnan na 16B (po dogovoru o klicih)
sw reg1, 12(sp)
sw reg2, 8(sp)
sw reg3, 4(sp)
```

Primer 1

Funkcija v jeziku C:

```
int sum(int a, int b)
{
   int c;

   c = a + b;

   return c;
}
```

- Kako izgleda prevedena zbirniška koda za RISC-V?
  - Za parametre a in b uporabimo registra a0-a1 (x10-x11), za izračun pa npr. register s4 (x20).

- Na začetku je oznaka procedure:

  sum:

- Registre, ki jih podprogram uporablja (x1, x8 in x20), je treba shraniti.

  Porinemo (push) njihove stare vrednosti na sklad:

```
addi sp,sp,-16      # prostor za 3 registre, poravnan na 16 B
```

- V splošnem je treba shraniti povratni naslov, ki je v x1 (ra)
  - Izjema so primeri procedur, ki ne kličejo nobene druge procedure (kot v gornjem primeru)
  - V takih primerih prevajalnik ugotovi, da gre za ‘leaf-proceduro’ in upravičeno domneva, da noben drug klic ne bo ‘povozil’ vrednosti v ra

```
# Celotna koda:
        .text
        li  sp, 0x500
        li  a0, 5
        li  a1, 6
        jal ra, sum
        ret      # jalr x0,ra,0

   sum:
        # Vstop:
        addi sp,sp,-16   # prostor za 3 registre (in poravnan na večkratnik 16)
        sw   ra,12(sp)   # x1 (povratni naslov)
        sw   s0,8(sp)    # x8 (kazalec na okvir)
        sw   s4,4(sp)    # x20 (s4)
        addi s0,sp,16    # kazalec na okvir gre na začetek okvira

        # Procedura:
        add  s4,a0,a1    # s4 <- a+b
        sw   s4,-12(s0)  # c <- s4
        lw   a0,-12(s0)  # izhodna vrednost

        # Izstop:
        lw    s4,4(sp)   # obnovimo register x20
        lw    s0,8(sp)
        lw    ra,12(sp)
        addi  sp,sp,16   # ‘zbrišemo’ registre s sklada
        ret              # vrnitev
```

Primer 2

Podprogram, ki kliče drug podprogram:

```
int fun1(int a, int b);

int main() { // funkcija, ki kliče drugo funkcijo
   fun1( 5, 6);

   return 0;
}

int fun1(int a, int b) { // ta funkcija je podprogram-list
   return(a+b);
}
```

Če program prevedemo z *riscv32-unknown-elf-gcc -static -o prog1 prog1.c* in pogledamo disassembly, ki ga izpiše *riscv32-unknown-objdump -d prog1* (-d ... disassembly), vidimo nekaj takega:

```
<main>:
   1018c:       ff010113      addi    sp,sp,-16 # Kazalec na sklad se zmanjša za 4 besede (za manj se ne sme)
   10190:       00112623      sw      ra,12(sp) # Tudi funkcija main shrani povratni naslov na sklad,
   10194:       00812423      sw      s0,8(sp)  #   shrani staro vrednost x8(s0 oz. fp)
   10198:       01010413      addi    s0,sp,16  #   in nastavi fp (frame pointer) na začetek sklada
   1019c:       00600593      li      a1,6      # 2. argument <- 6 (x11)
   101a0:       00500513      li      a0,5      # 1. argument <- 5 (x10)
   101a4:       01c000ef      jal     ra,101c0 <fun1> # Klic funkcije fun1, v x1(ra) se shrani povratni naslov 101a8
   101a8:       00000793      li      a5,0      # a5 <- 0 (main vrne 0)
   101ac:       00078513      mv      a0,a5     # a0 <- a5 (=0)
   101b0:       00c12083      lw      ra,12(sp) # x1(ra) <- stara vrednost
   101b4:       00812403      lw      s0,8(sp)  # x8(s0/fp) <- stara vrednost
   101b8:       01010113      addi    sp,sp,16  # Kazalec na sklad se vrne v prejšnje stanje
   101bc:       00008067      ret

<fun1>:
   101c0:       fe010113      addi    sp,sp,-32 # Kazalec na sklad se zmanjša za 8 besed
   101c4:       00812e23      sw      s0,28(sp) # fp <- sp + 28 (1. lokacija (slot) sklada)
   101c8:       02010413      addi    s0,sp,32  # fp <- sp + 32 (začetna vrednost sp)
   101cc:       fea42623      sw      a0,-20(s0) # shrani 1. argument (x10(a0)=5) na sklad (lokacija 5)
   101d0:       feb42423      sw      a1,-24(s0) # shrani 2. argument (x11(a1)=6) na sklad (lokacija 6)
   101d4:       fec42703      lw      a4,-20(s0) # a4 <- 5
   101d8:       fe842783      lw      a5,-24(s0) # a5 <- 6
   101dc:       00f707b3      add     a5,a4,a5  # a5 <- 5 + 6
   101e0:       00078513      mv      a0,a5     # x(10)a0 <- a5 (=11)
   101e4:       01c12403      lw      s0,28(sp) # fp <- stara vrednost registra (iz 1. slota sklada)
   101e8:       02010113      addi    sp,sp,32  # Kazalec na sklad se vrne v prejšnje stanje
   101ec:       00008067      ret               # = jalr x0, 0(x1), vrnitev v klicoči program
```

Primer 3

Podprogram, ki kliče drug podprogram:

```
#include <stdio.h>

int fun1(int a, int b);

int main() {
    fun1( 5, 6);
    return 0;
}

int fun1(int a, int b) {
    int c;
    c = a+b;
    printf("%d", c);            // fun1 kliče funkcijo printf
    return(c);
}
```

```
<main>:
...
   101a4:  01c000ef          jal ra,101c0 <fun1>  # povratni naslov ra <- 101a8
...

<fun1>:
   101c0:  fd010113          addi sp,sp,-48 # malo večji okvir (3*16B)
   101c4:  02112623          sw  ra,44(sp)  # povratni naslov (101a8) se tu shrani na sklad,
                                            # ker ga JAL prepiše!
   101c8:  02812423          sw  s0,40(sp)
   101cc:  03010413          addi           s0,sp,48
   101d0:  fca42e23          sw  a0,-36(s0)
   101d4:  fcb42c23          sw  a1,-40(s0)
   101d8:  fdc42703          lw  a4,-36(s0)
   101dc:  fd842783          lw  a5,-40(s0)
   101e0:  00f707b3          add a5,a4,a5
   101e4:  fef42623          sw  a5,-20(s0)
   101e8:  fec42583          lw  a1,-20(s0)
   101ec:  000257b7          lui a5,0x25
   101f0:  07878513          addi           a0,a5,120 # 25078 <__clzsi2+0x4c>
   101f4:  1d4000ef          jal ra,103c8 <printf>    # ra je zdaj 101f8!
   101f8:  fec42783          lw  a5,-20(s0)
   101fc:  00078513          mv  a0,a5
   10200:  02c12083          lw  ra,44(sp)  # obnovi s sklada povratni naslov 101a8
   10204:  02812403          lw  s0,40(sp)
   10208:  03010113          addi           sp,sp,48
   1020c:  00008067          ret
```

- V splošnem je treba za sabo pospraviti
  - torej, če uporabimo znotraj procedure neke registre, je treba njihove stare vrednosti shraniti in jih pred izstopom iz procedure obnoviti, ker jih klicatelj morda uporablja.
- Da pa se lahko izognemo shranjevanju in obnavljanju registrov, katerih vrednosti ne bodo nikoli uporabljene, kar se tipično zgodi z začasnimi registri, RISC-V loči 19 registrov v dve skupini:
  - **x5–x7 (t0-t2) in x28–x31 (t3-t6): začasni registri**, ki jih pri klicu procedure klicani podprogram ne ohrani,
  - **x8 (s0/fp), x9 (s1) in x18–x27 (s2-s11): registri, ki jih je treba shraniti** pri klicu podprograma (če se uporablja, jih klicani program shrani in obnovi).
- S tem preprostim dogovorom se zmanjša 'razlivanje' registrov.
  - V primeru zgoraj, ker klicatelj ne pričakuje, da bosta registra x5 in x6 ohranjena po klicu procedure, lahko prištedimo dva ukaza store in dva ukaza load. Še vedno pa moramo shraniti in obnoviti x20 - klicani program mora domnevati, da klicatelj potrebuje njegovo vrednost.

Podprogrami-listi in gnezdeni podprogrami

- Podprogram, ki ne kliče nobenega drugega podprograma (niti printf), se imenuje list (leaf)
  - Takemu načelno ni treba shraniti povratnega naslova, ki je v x1(ra), na sklad, saj ga ne bo prepisal povratni naslov kakega drugega podprograma
- Kadar podprogram kliče drug podprogram, ali celo samega sebe (rekurzija), so stvari malo bolj zapletene.
  - Npr.: glavni program kliče podprogram A z argumentom 3, (x10 ← 3) in z uporabo jal x1, A.
  - Predpostavimo, da podprogram A kliče podprogram B prek jal x1, B z argumentom 7, prav tako postavljenim v x10 (=a0).
  - Ker A še ni končal svoje naloge, pride do konflikta glede uporaba registra x10.
  - Podobno obstaja konflikt glede povratnega naslova v registru x1, saj je v x1 zdaj povratni naslov za B.
  - Če ničesar ne storimo, se podprogram A ne bo mogel vrniti k klicatelju.

```
                addi x10, x0, 3   # x10 = 3
0x40            jal x1, A         # x1 = 0x44, skok na A
0x44
                ...
0x64    A:      ...
0x68            addi x10, x0, 7
0x6c            jal x1, B         # x1 = 0x70, skok na B
```

Rešitev je, da potisnemo vse druge registre, ki morajo biti ohranjeni, na sklad, tako kot smo storili s shranjenimi registri.

- Klicatelj potisne vse registre argumentov (x10–x17, tj. a0-a7) ali začasne registre (x5-x7, tj. t0-t2 in x28-x31), ki so potrebni po klicu.
- Klicani potisne register x1 (=ra) s povratnim naslovom in vse ‘saved’ registre (x8-x9, tj. s0 in s1, in x18-x27, tj. s2-s11), ki jih uporablja.
- Skladovni kazalec sp se prilagodi tako, da upošteva število registrov, postavljenih na sklad.
- Po vrnitvi se registri obnovijo iz pomnilnika in skladovni kazalec je ustrezno postavi.

Okvir procedure (frame)

- Kazalec na okvir (frame pointer, fp) si zapomni stanje kazalca na sklad (sp) pred shranjevanjem argumentnih registrov
  - sp se bo kasneje morda spreminjal
    - zato je spremenljivke lažje referencirati preko fp (ki se tekom procedure ne spreminja)
  - pri RISC-V temu služi x8 (s0/fp)
    - Po dogovoru o klicih sicer ni nujno potreben
  - Po dogovoru o klicih je velikost okvira poravnana na 16 bajtov
    - pri 'čistem' zbirnem programu, ki ne uporablja C funkcij, je to sicer vseeno

Okvir znotraj sklada →

SP →	Lokalna polja in strukture (če jih je kaj)
	Shranjeni 'shranjeni' (saved) registri (če jih je kaj)
	Shranjen povratni naslov
FP →	Shranjeni argumentni registri (če jih je kaj)

- Spremenljivka v jeziku C je na splošno lokacija v pomnilniku, njena interpretacija pa je odvisna od njene vrste in razreda shranjevanja (storage class).
  - Primeri tipov vključujejo cela števila in znake.
  - C ima dva razreda shranjevanja: automatic in static.
  - Avtomatske spremenljivke so lokalne za proceduro in se zavržejo, ko se procedura konča.
  - Statične spremenljivke pa se ohranijo.
  - Spremenljivke, deklarirane zunaj vseh funkcij (torej globalne), veljajo za statične, tako kot vse spremenljivke, eksplicitno deklarirane kot static. Ostale so avtomatske.
  - Za poenostavitev dostopa do statičnih podatkov nekateri prevajalniki RISC-V rezervirajo register x3 za uporabo kot globalni kazalec ali gp (global pointer).

Globalni kazalec gp je torej register, ki kaže na statično območje.

Kateri registri se ohranijo pri klicu procedure?

Se ohranijo	**Se ne ohranijo**
Shranjeni registri: x8-x9, x18-x27	Začasni registri: x5-x7, x28-x31
Kazalec na sklad: x2 (sp)	Argumentni registri: x10-x17
Kazalec na okvir: x8 (fp)	
Povratni naslov: x1 (ra)	

Poleg tega se na sklad shranjuje tudi lokalne spremenljivke, ki so prevelike za v registre, npr. lokalna polja in strukture. Del sklada, ki hrani shranjene registre in lokalne spremenljivke procedure, se imenuje okvir procedure.

- RISC-V teži k temu, da pogoste primere skuša izvesti hitro, zato je možno veliko funkcij izvesti z 8 argumentnimi registri, 12 shranjenimi reg. in 7 začasnimi, torej brez uporabe pomnilnika.
  - Kadar pa je parametrov več kot 8, jih klicatelj porine na sklad tik nad kazalcem na okvir (fp).
- Pri rekurzivnih funkcijah pa je sicer vedno tudi možnost iterativne izvedbe

Povzetek uporabe registrov pri klicih podprogramov:

Register	Uporaba	Ali se mora ob klicu ohraniti?
x0	0	
x1 (ra)	povratni naslov (return address)	da
x2 (sp)	kazalec na sklad (stack pointer)	da
x3 (gp)	global pointer	da
x4 (tp)	thread pointer	da
x5-x7 (t0-t2)	začasni (temporary) registri	ne
x8 (fp/s0), x9 (s1)	shranjeni* (saved) registri	da
x10, x11 (a0, a1)	argumenti/rezultati	ne
x12-x17 (a2-a7)	argumenti	ne
x18-x27 (s2-s11)	shranjeni* (saved) registri	da
x28-x31 (t3-t6)	začasni (temporary) registri	ne

* v smislu, da morajo biti shranjeni, če se jih uporablja, in nato obnovljeni

Kopica

- Nekatere podatkovne strukture (npr. povezani seznami ali drevesa) se med tekom programa glede na podatke lahko zelo spreminjajo
  - zato jih je težko umestiti med statične podatke, saj ni jasno vnaprej, koliko prostora bodo zasedli v (glavnem) pomnilniku
  - segment za dinamično alokacijo takih podatkov se imenuje *kopica* (*heap*)
    - običajno raste iz druge smeri kot sklad, tako da se medsebojno približujeta
    - malloc() je C-funkcija za dinamično alokacijo (zadaj je sistemski klic, npr. sbrk na Linuxu), free() pa za sprostitev zaseženega pomnilnika
    - C-program sam nadzoruje dodeljevanje pomnilnika, kar lahko vodi tudi do problemov (npr. odtekanje ('memory leak'), ali pa uporaba kazalca po sprostitvi ('dangling pointer'))
    - Java se temu izogne tako, da uporablja samodejno alokacijo pomnilnika in 'garbage collection'

Pomnilniška slika pri RISC-V: