Programiranje procesora FRISC

Prijenos stogom

Potprogrami - Prijenos stogom

- Pozivatelj prije poziva mora staviti argumente na stog
  - **Oprez: naredba CALL stavlja povratnu adresu na stog**
- Potprogram polazi od pretpostavke da su na stogu ispod povratne adrese argumenti i koristi ih pri izračunavanju rezultata **(kako?)**
- Potprogram mora staviti rezultat na stog **(kako?)**
  - **Oprez: naredba RET uzima s vrha stoga povratnu adresu**
- Pozivatelj, nakon povratka iz potprograma, pretpostavlja da se na vrhu stoga nalaze rezultati koje uzima sa stoga i koristi ih

Potprogrami - Prijenos stogom

- Problem je povratna adresa koja se automatski stavlja i uzima na stog naredbama CALL i RET.

- Ova povratna adresa:
  - smeta dohvatu argumenata sa stoga
  - smeta stavljanju rezultata na stog

- Ispravno baratanje sa stogom moguće je na više načina, ali treba poznavati kako radi stog i naredbe CALL i RET

Potprogrami - Prijenos stogom

Napisati potprogram koji izračunava logičku operaciju NILI. Parametri i rezultat se prenose stogom. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat sprema natrag u memoriju.

Rješenje:

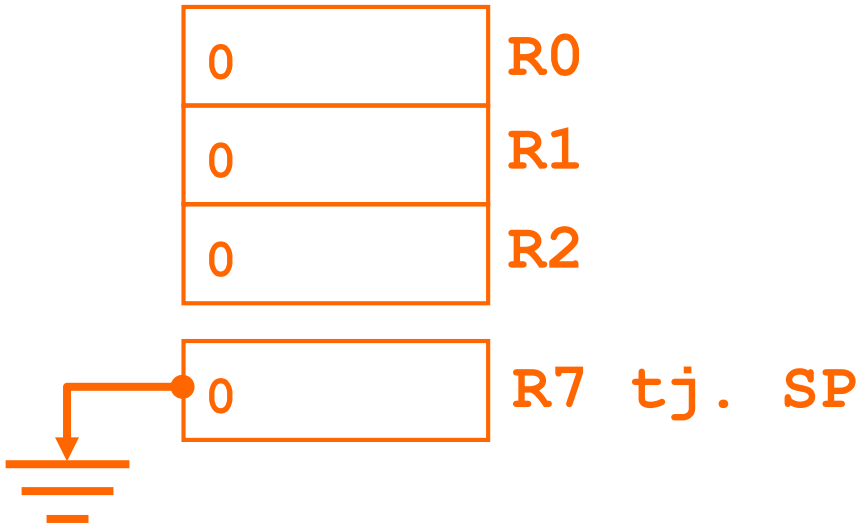
- U ovom rješenju potprogram mijenja registre R0, R1 i R2, ali **ne čuva njihove vrijednosti (LOŠE!!!)**

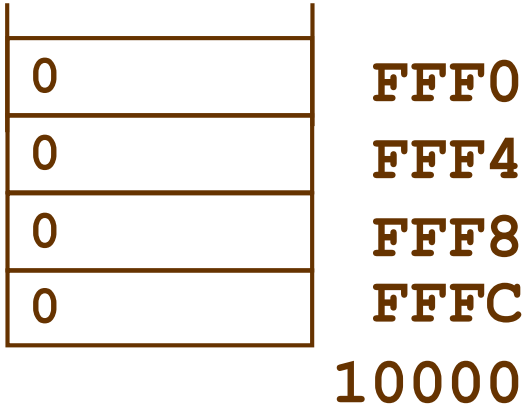
- Kasnije ćemo pokazati bolje rješenje (ovo je samo primjer kako se može zaobići povratna adresa pri slanju parametara i povratu rezultata)

>>>>

<<<< Izvođenje programa:

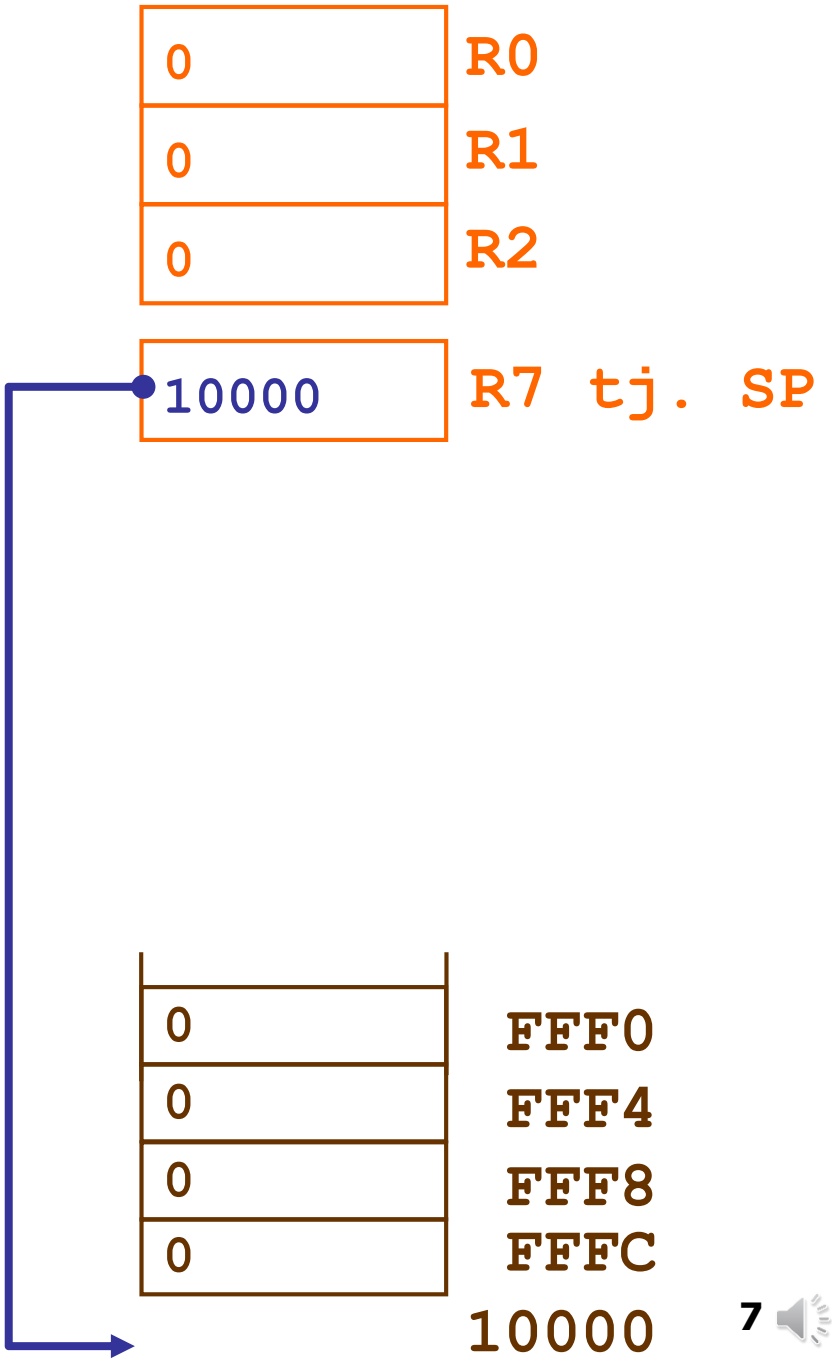
```
GLAVNI MOVE  10000, SP

       LOAD  R0, (PRVI)
       PUSH  R0

       LOAD  R0, (DRUGI)
       PUSH  R0

       CALL  NILI

       POP   R0
       STORE R0, (REZ)

       HALT
```





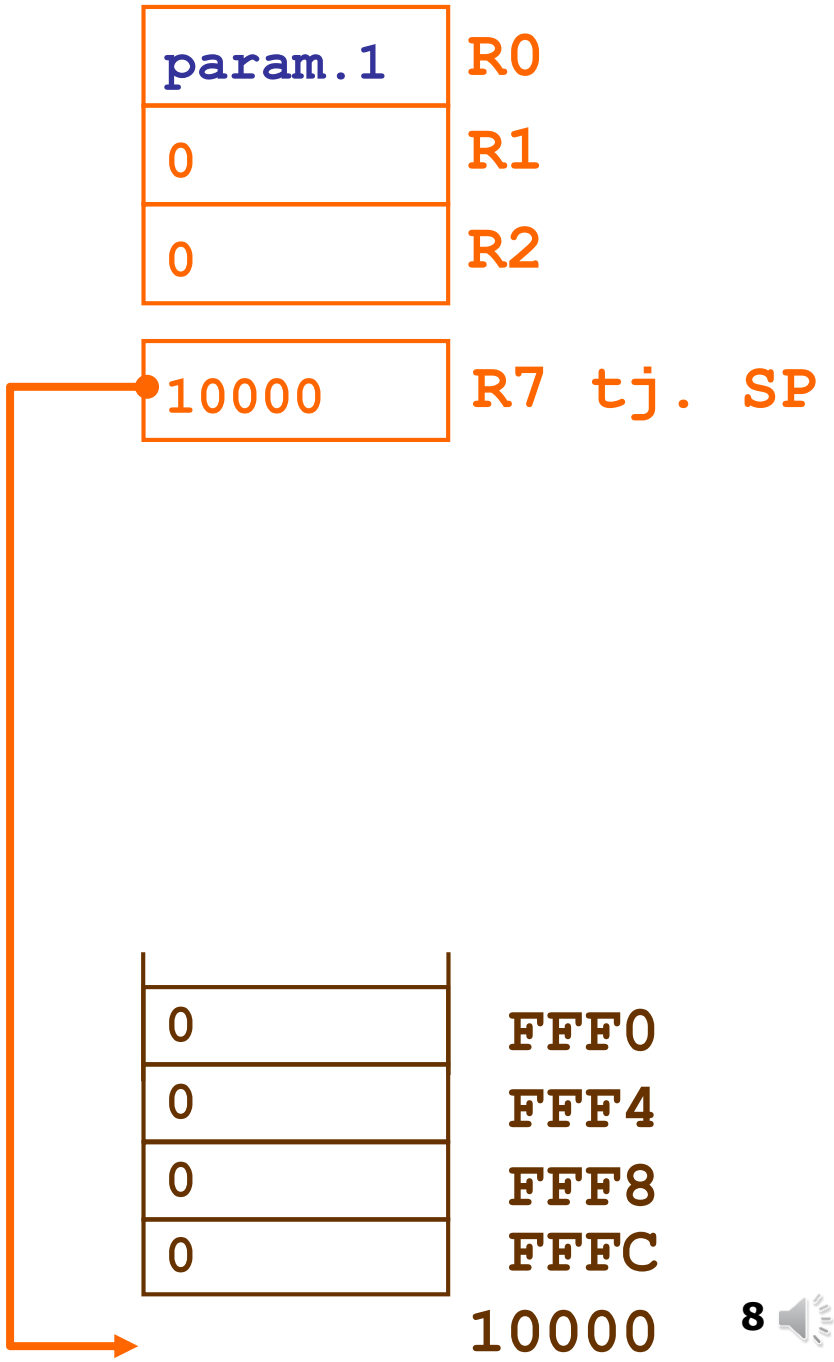
<<<< Izvođenje programa:

```
GLAVNI MOVE  10000, SP
       LOAD  R0, (PRVI)
       PUSH  R0
       LOAD  R0, (DRUGI)
       PUSH  R0
       CALL  NILI
       POP   R0
       STORE R0, (REZ)
       HALT
```



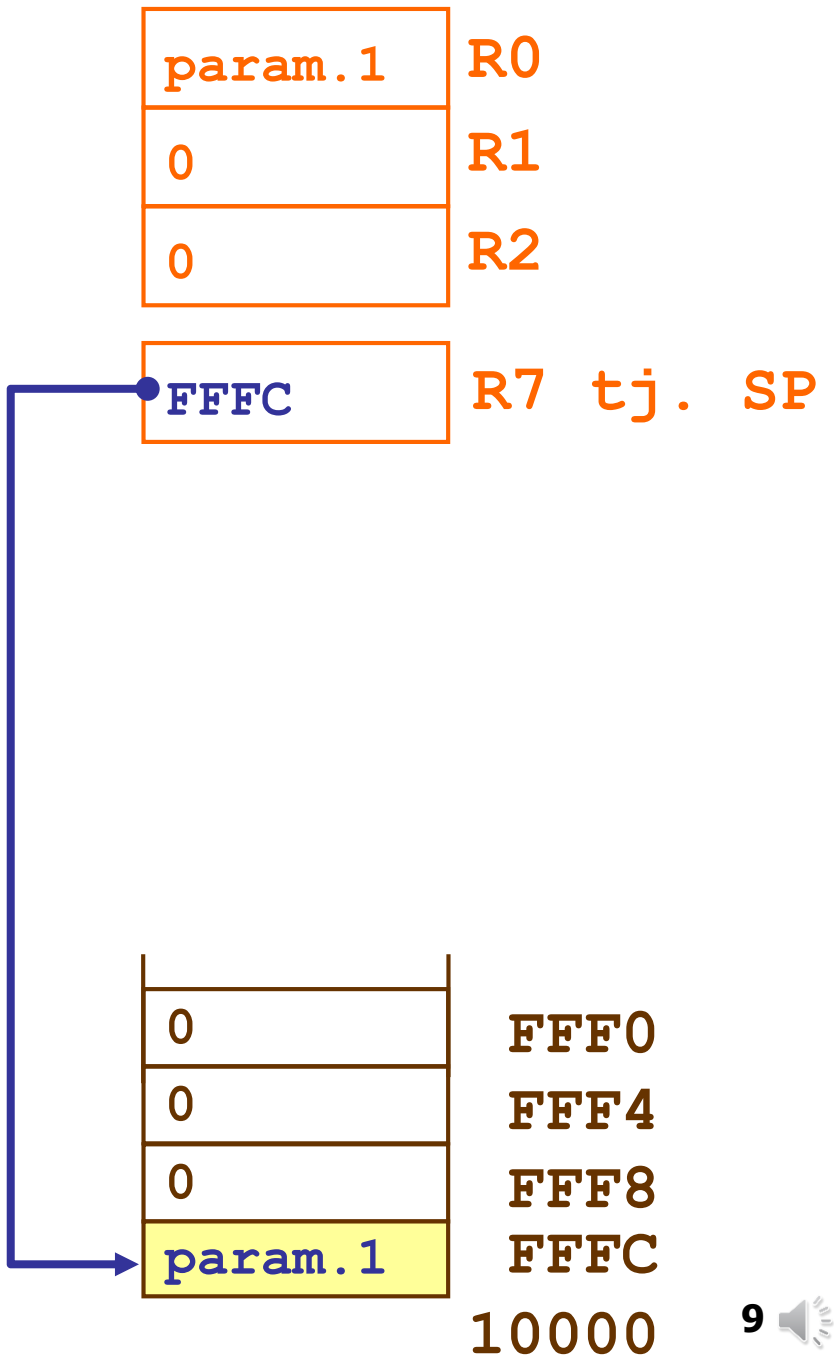
<<<< Izvođenje programa:

```
GLAVNI MOVE  10000, SP

       LOAD  R0, (PRVI)
       PUSH  R0

       LOAD  R0, (DRUGI)
       PUSH  R0

       CALL  NILI

       POP   R0
       STORE R0, (REZ)

       HALT
```



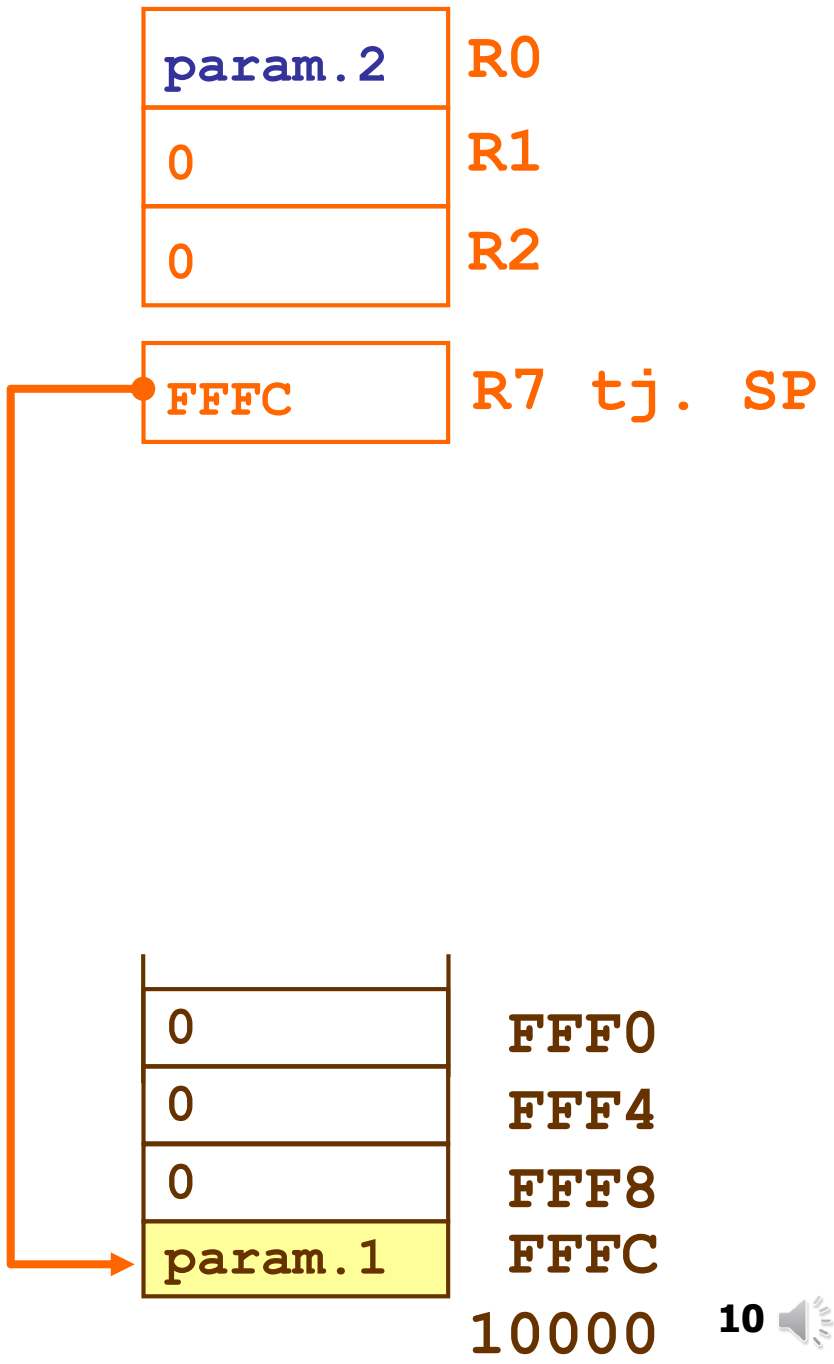
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       POP    R0
       STORE  R0, (REZ)

       HALT
```



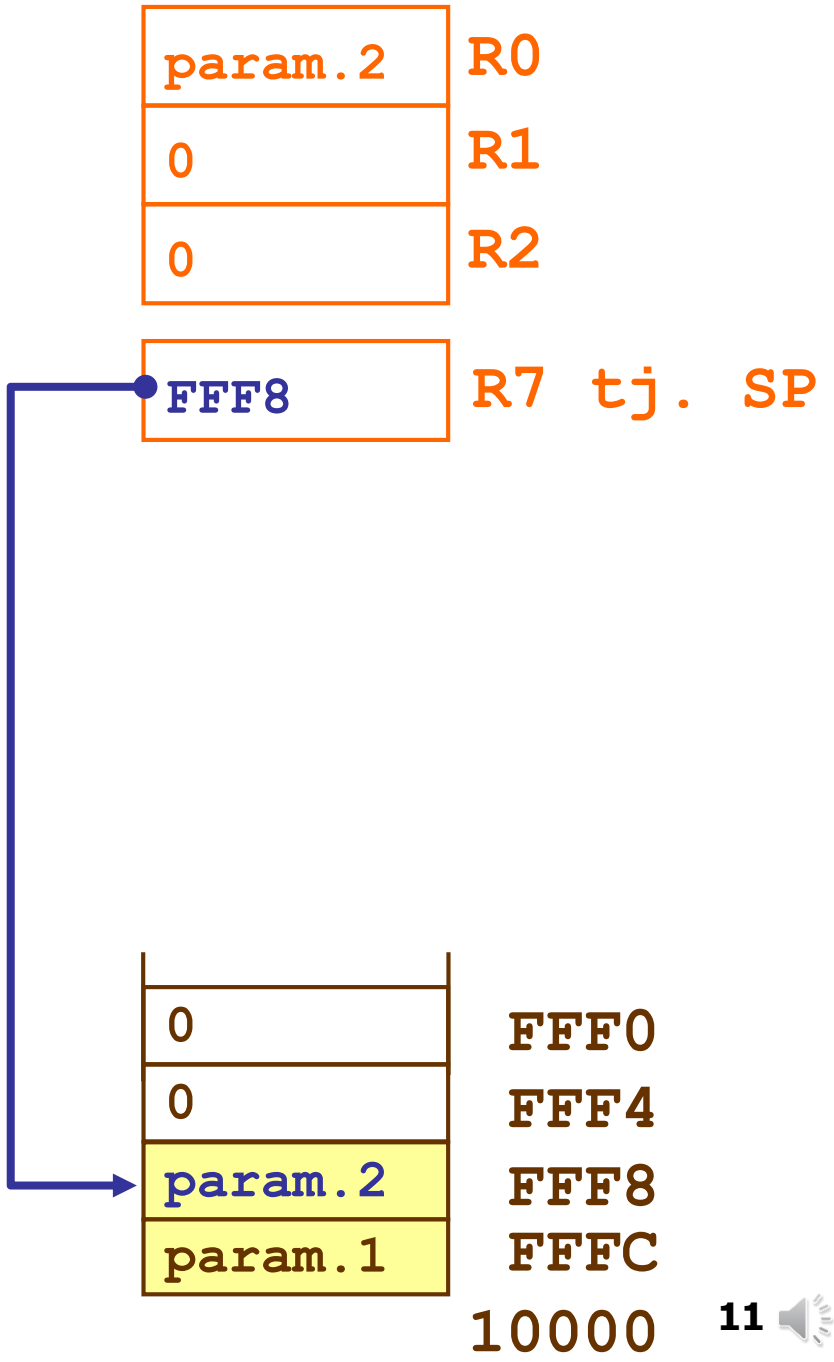
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       POP    R0
       STORE  R0, (REZ)

       HALT
```



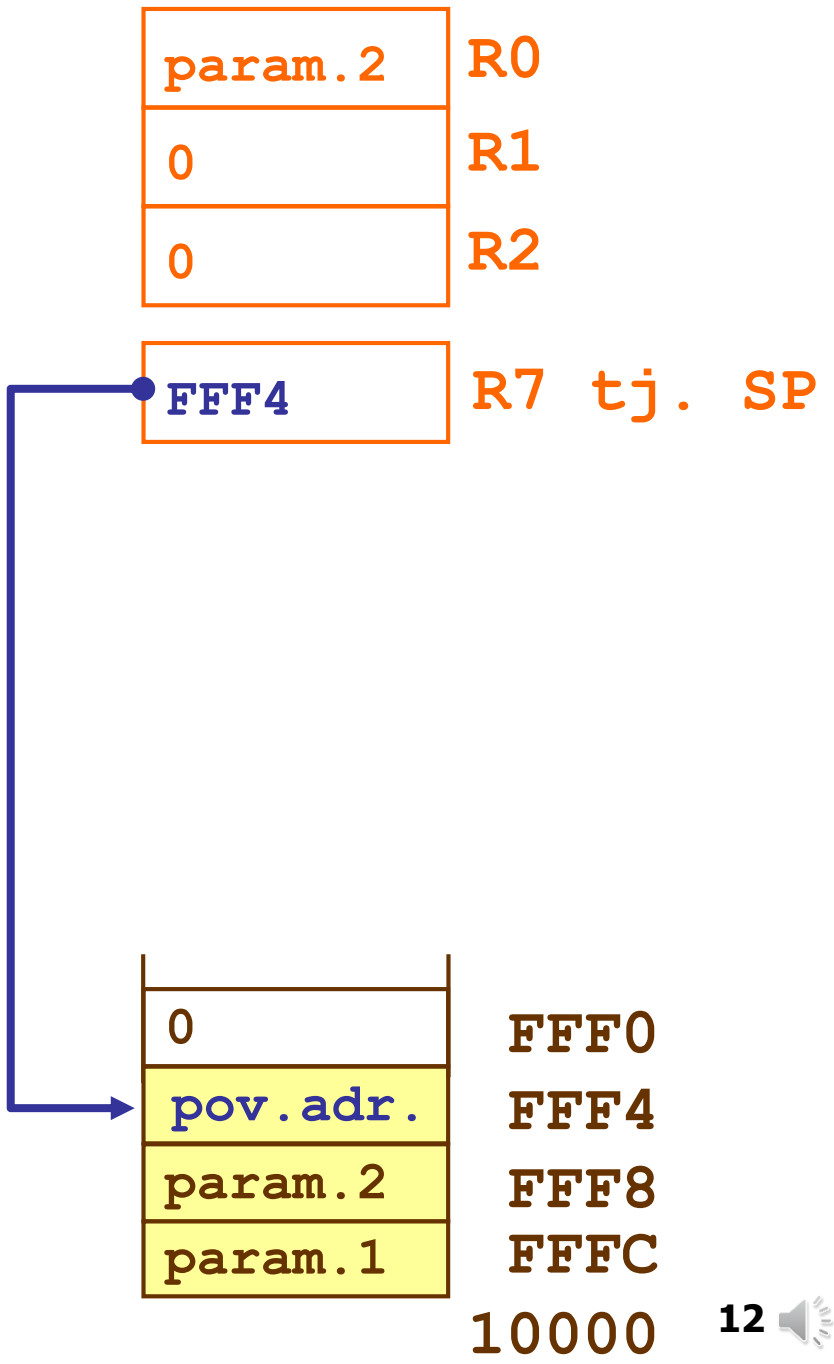
<<<< Izvođenje programa:

```
GLAVNI MOVE  10000, SP

       LOAD  R0, (PRVI)
       PUSH  R0

       LOAD  R0, (DRUGI)
       PUSH  R0

       CALL  NILI

       POP   R0
       STORE R0, (REZ)

       HALT
```



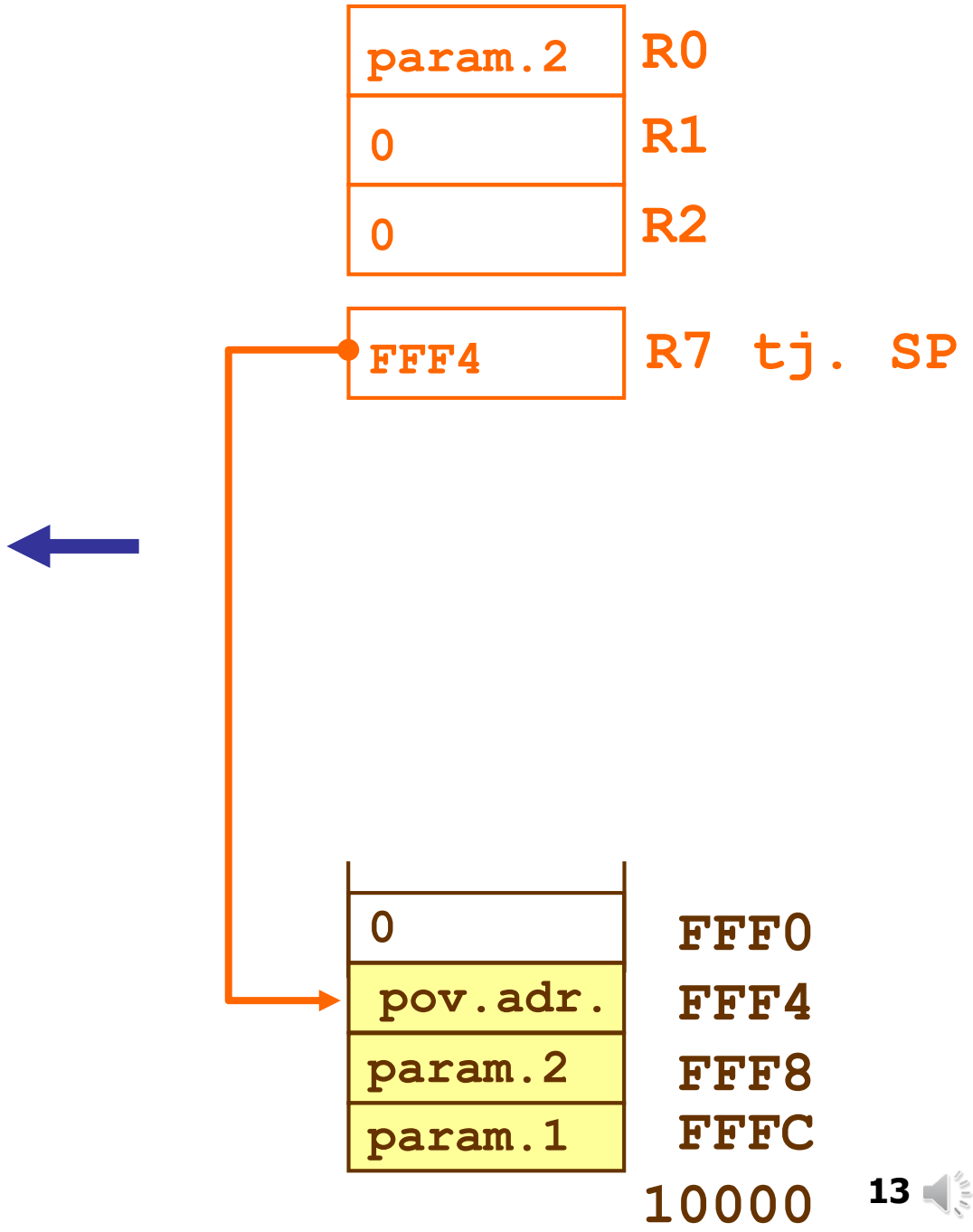
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       POP    R0
       STORE  R0, (REZ)

       HALT
```



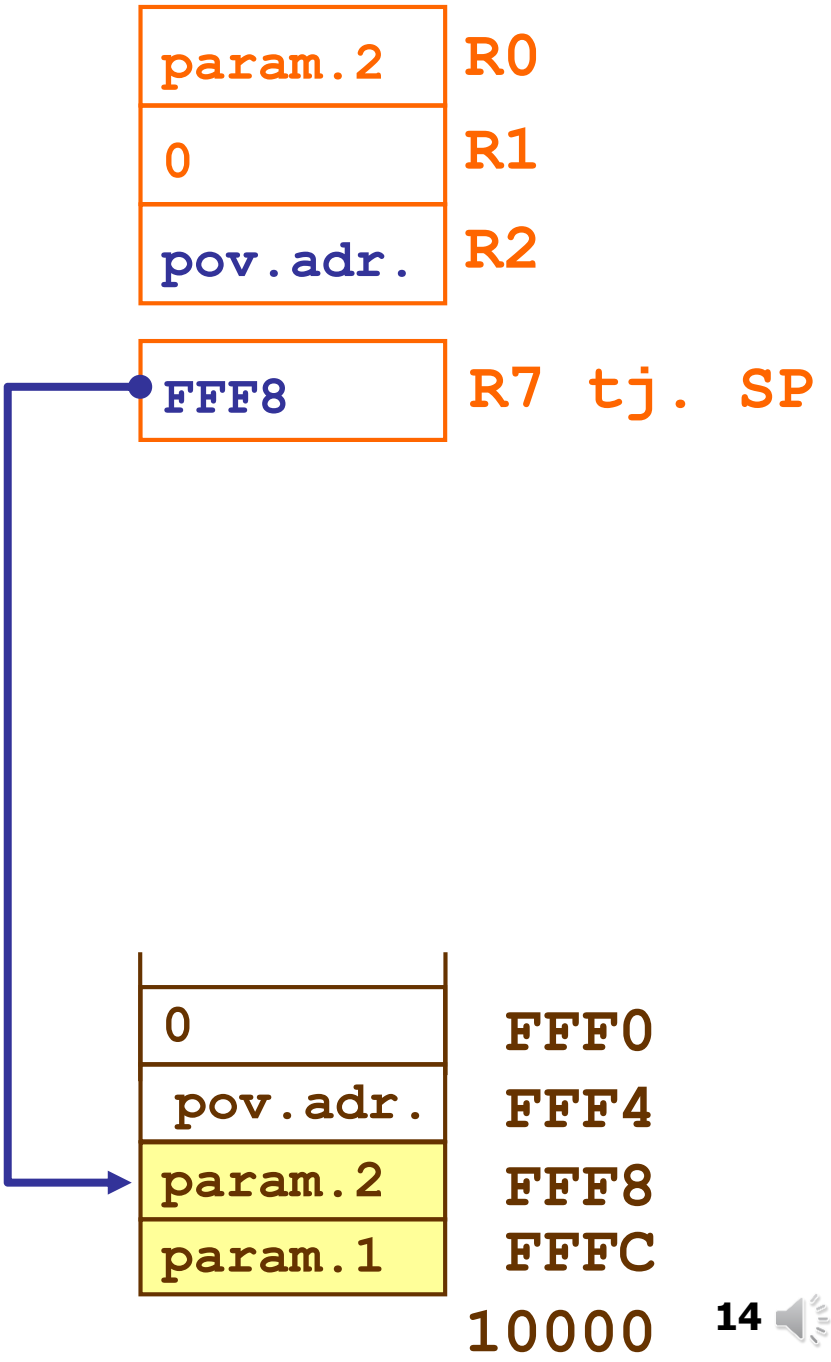
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



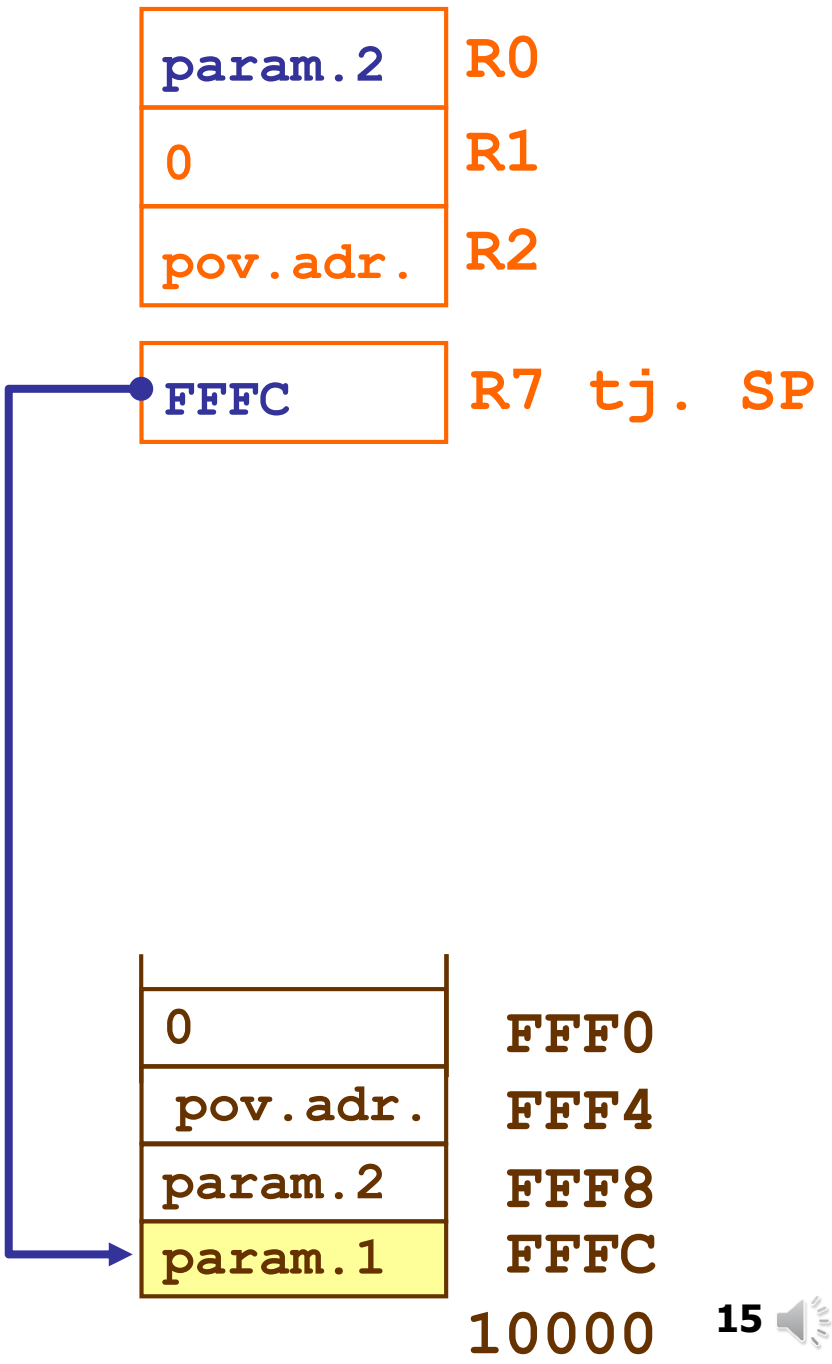
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



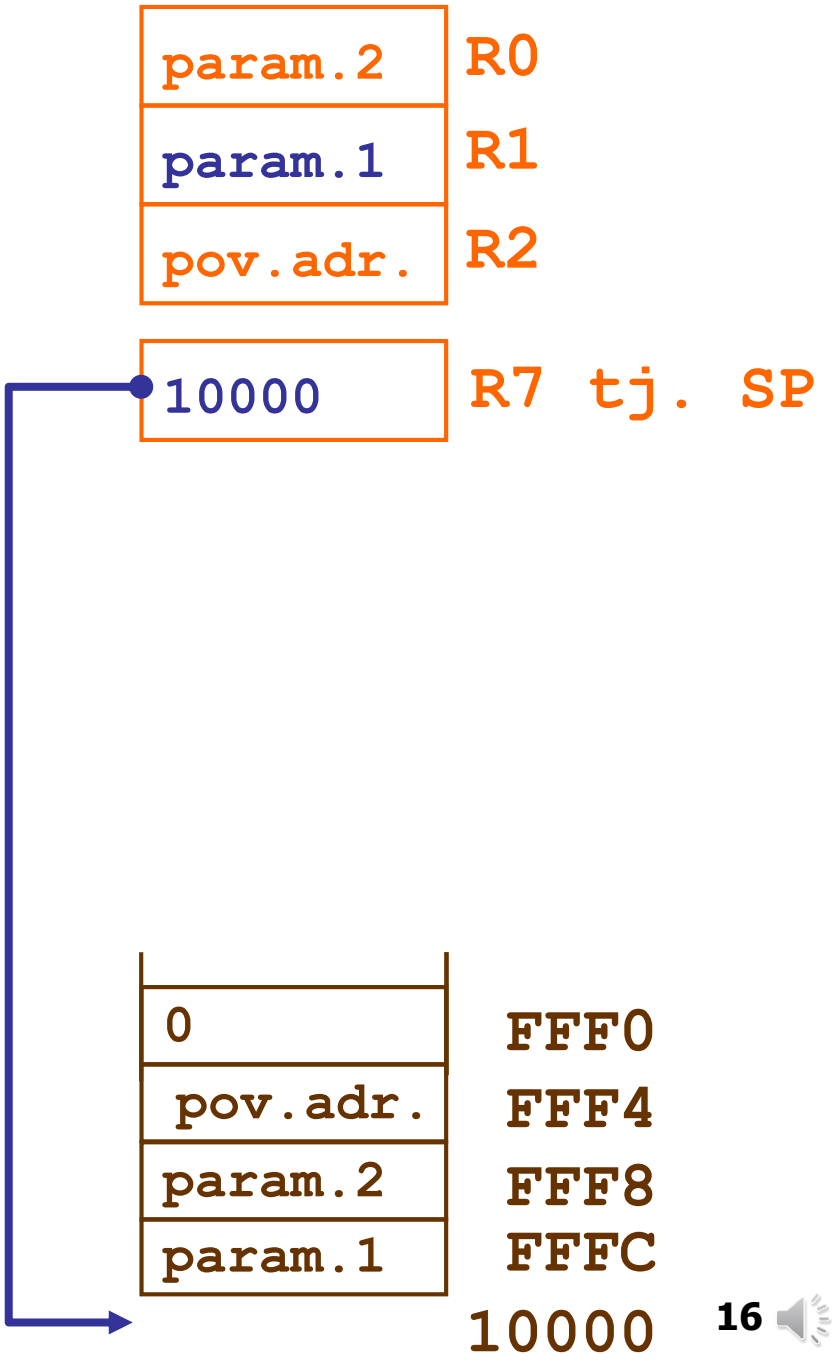
<<<< Izvođenje programa:

```
NILI    POP   R2
        POP   R0
        POP   R1

        OR    R0, R1, R0
        XOR   R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



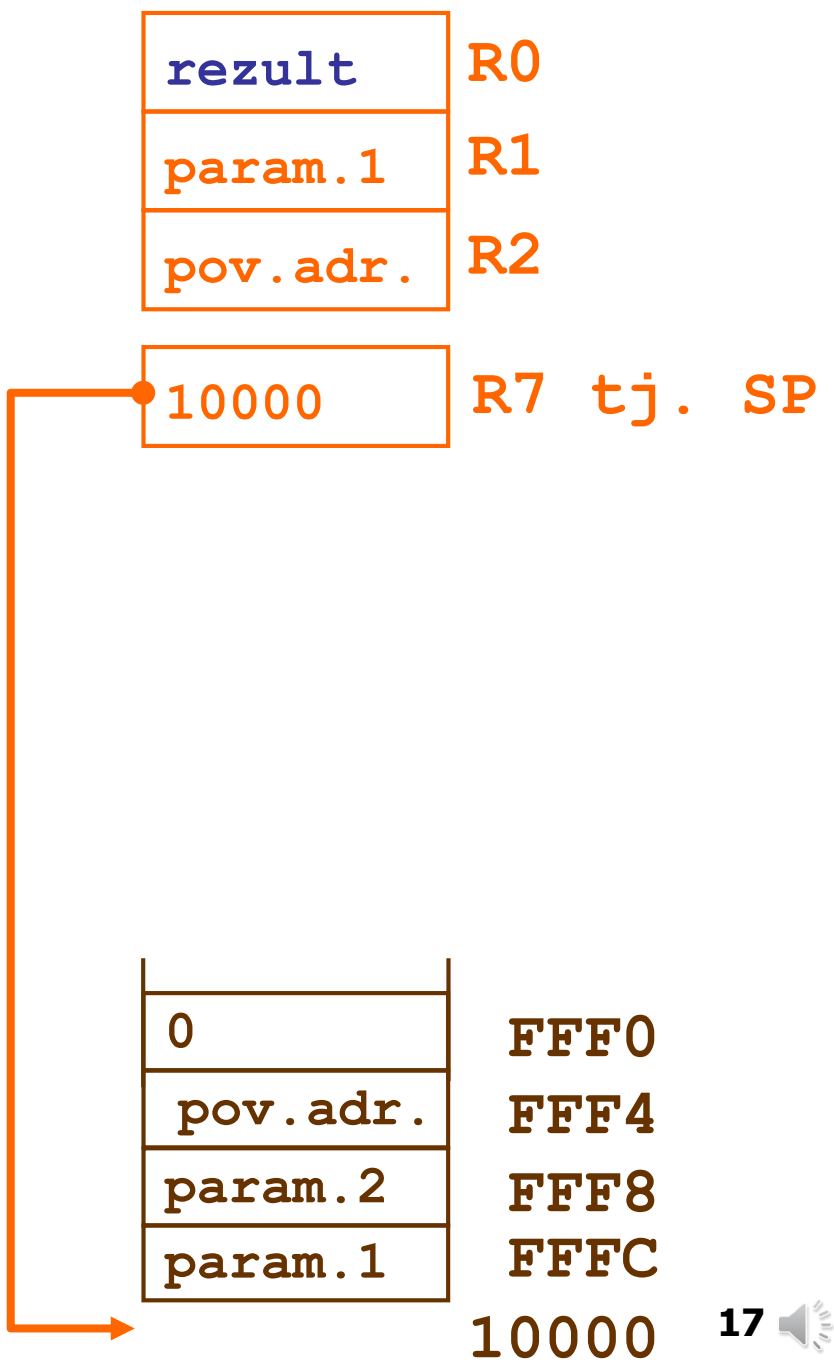
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



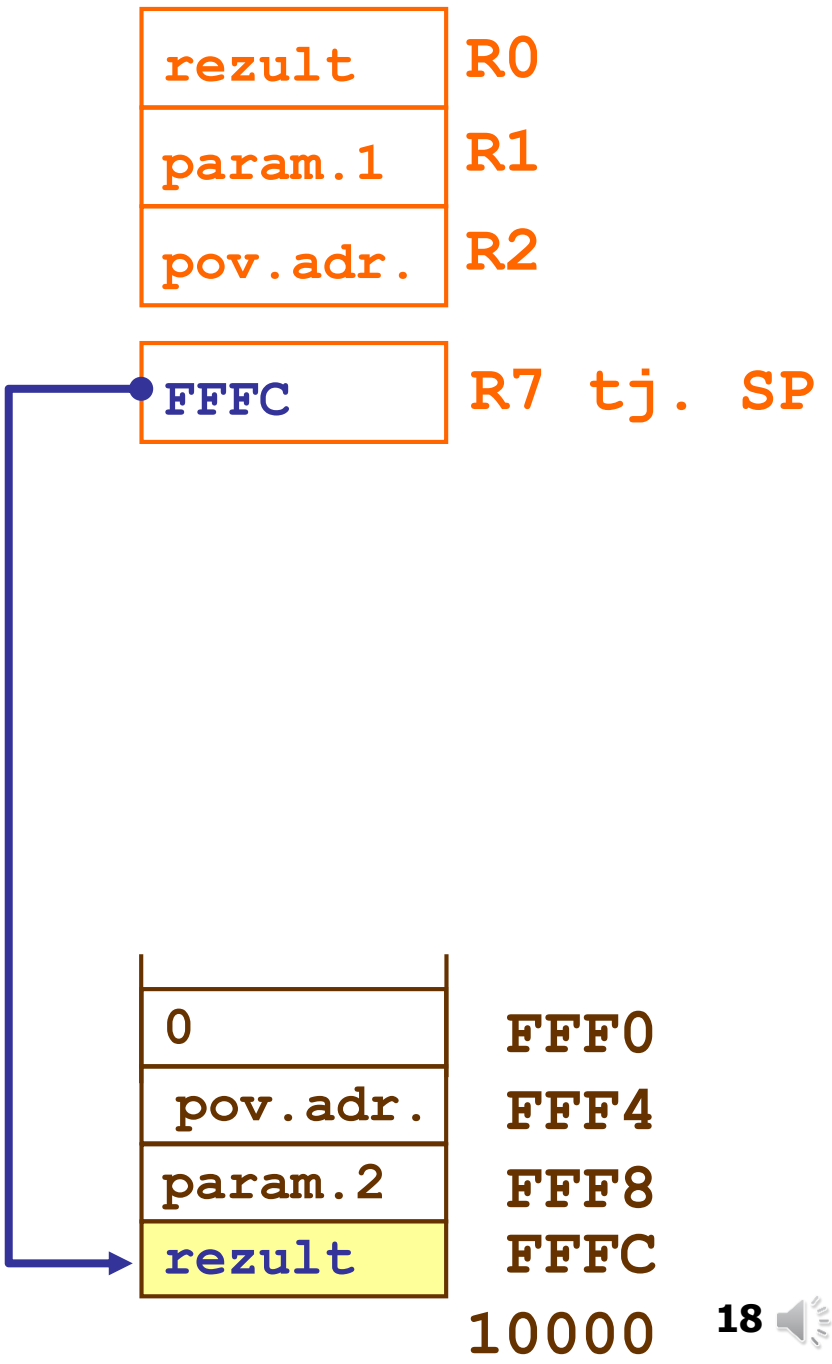
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



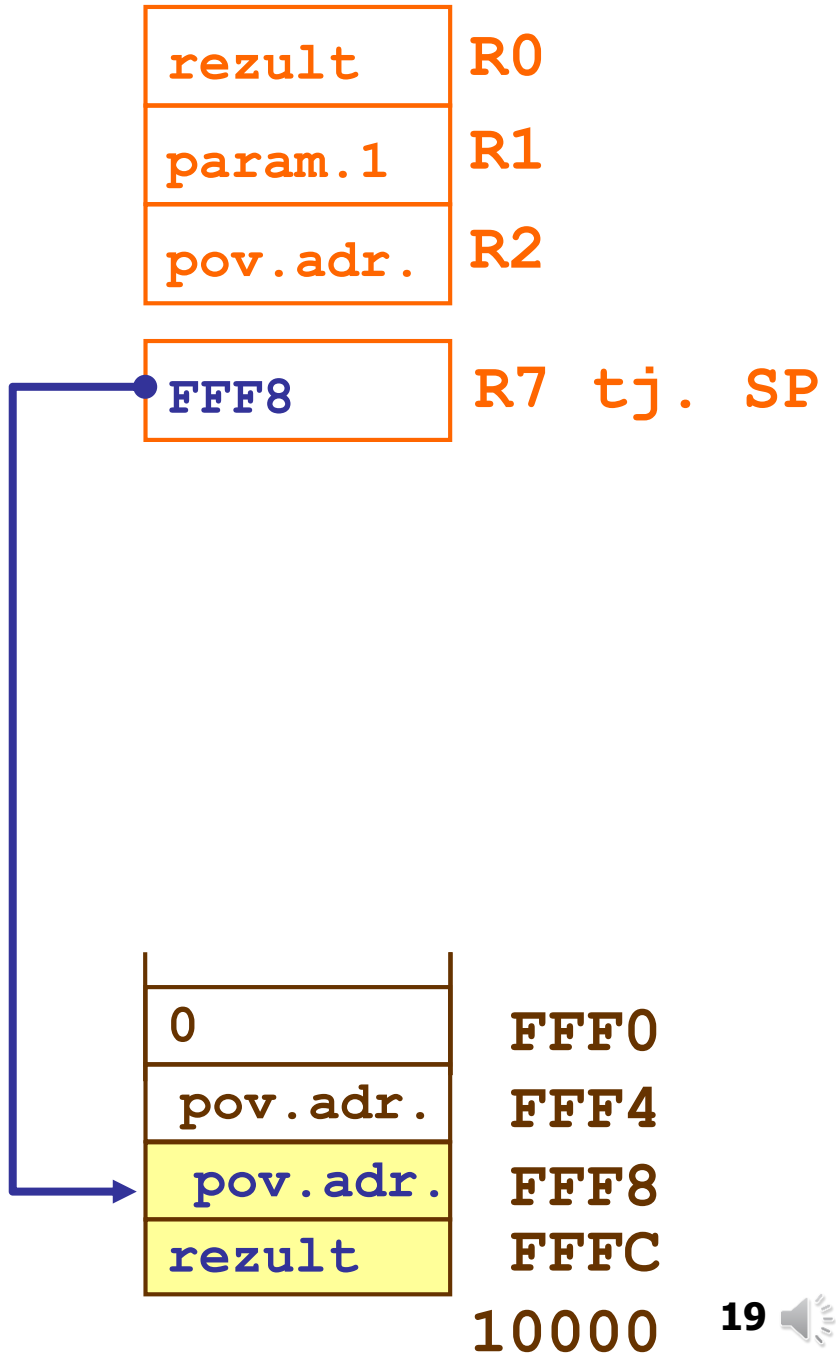
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



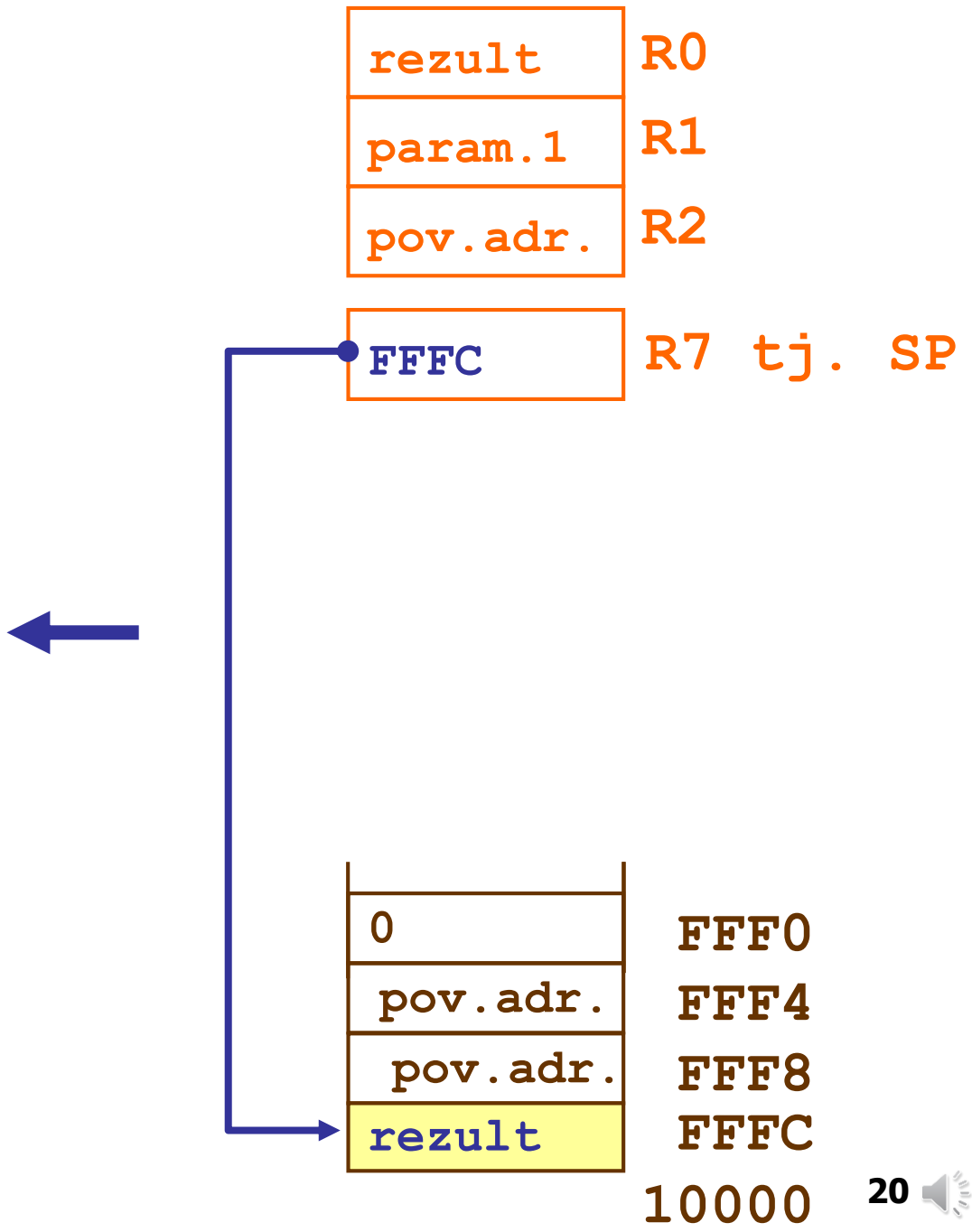
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



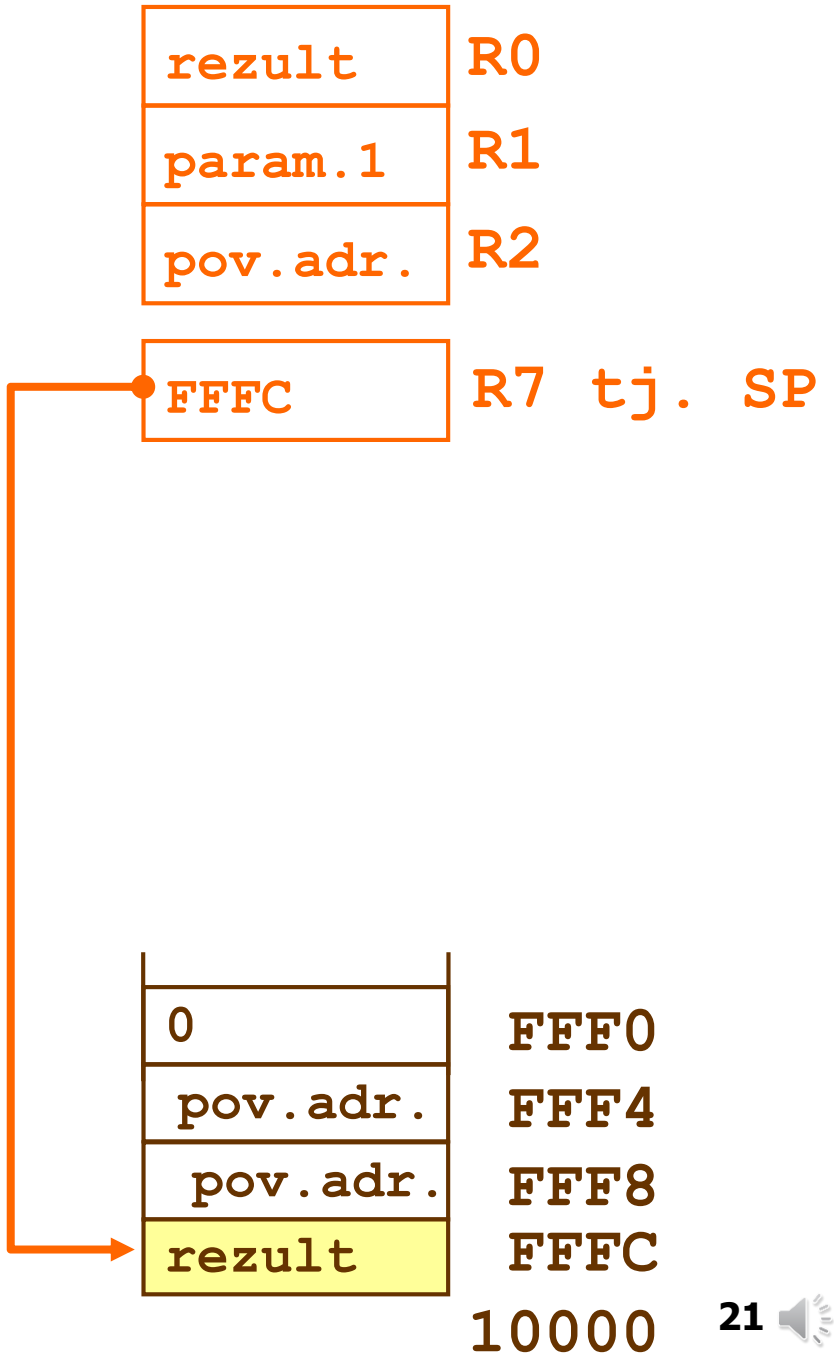
<<<< Izvođenje programa:

```
NILI    POP    R2
        POP    R0
        POP    R1

        OR     R0, R1, R0
        XOR    R0, -1, R0

        PUSH R0
        PUSH R2
        RET
```



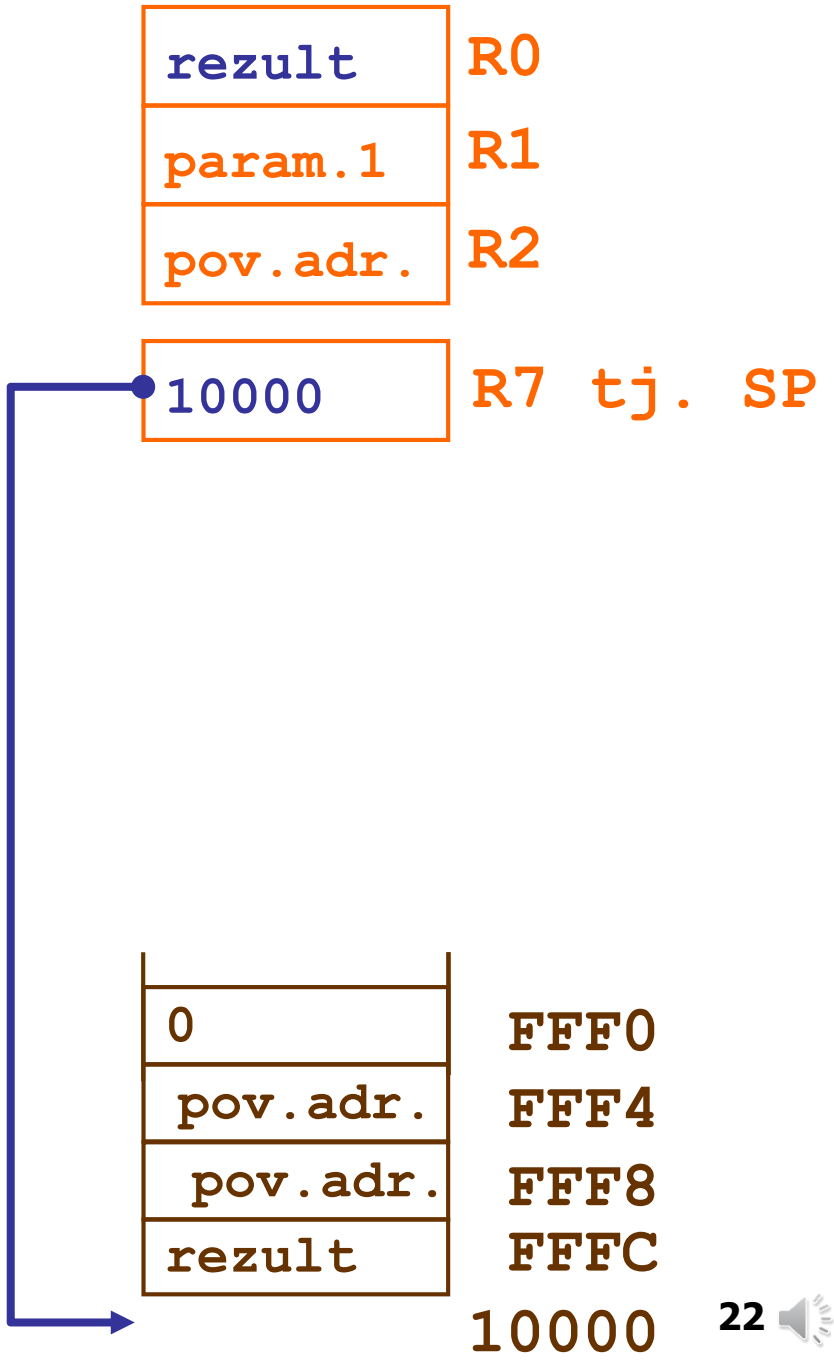
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       POP    R0
       STORE  R0, (REZ)

       HALT
```



<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       POP    R0
       STORE  R0, (REZ)

       HALT
```



<<<< Izvođenje programa:

- dohvaćeni rezultat se koristi
- stog je u početnom stanju

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       POP    R0
       STORE  R0, (REZ)

       HALT
```

<<<< (kompletan listing s komentarima)

```
        ; glavni program
GLAVNI  MOVE   10000, SP    ;važno: inicijaliziraj SP !!!

        ; stavi vrijednost prvog parametra na stog
        LOAD   R0, (PRVI)
        PUSH   R0

        ; stavi vrijednost drugog parametra na stog
        LOAD   R0, (DRUGI)
        PUSH   R0

        CALL   NILI         ;poziv potprograma

        ; uzmi rezultat sa stoga i spremi ga
        POP    R0
        STORE  R0, (REZ)     ;spremanje rezultata

        HALT
```

>>>>

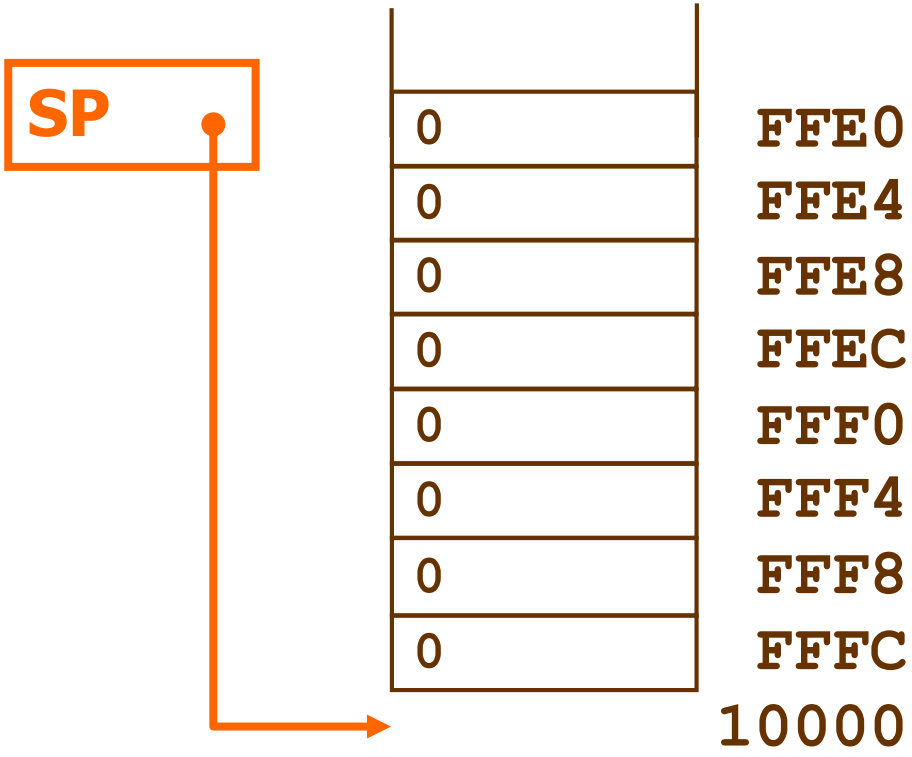
```
<<<<
        ; potprogram NILI
NILI    POP   R2       ; uzmi povratnu adresu sa stoga
        POP   R0       ; dohvat prvog parametra
        POP   R1       ; dohvat drugog parametra

        OR    R0, R1, R0    ; Izračunavanje
        XOR   R0, -1, R0    ; rezultata.

        PUSH R0        ; stavi rezultat na stog
        PUSH R2        ; vrati povratnu adresu na stog
        RET


        ; podatci i mjesto za rezultat
PRVI    DW  1234ABCD
DRUGI   DW  22445599
REZ     DW  0
```

Potprogrami - Prijenos stogom

- Pokušajmo riješiti prethodni zadatak tako da se čuvaju stanja registara koje potprogram mijenja

- Već smo vidjeli da je najzgodnije **registre spremiti na stog**

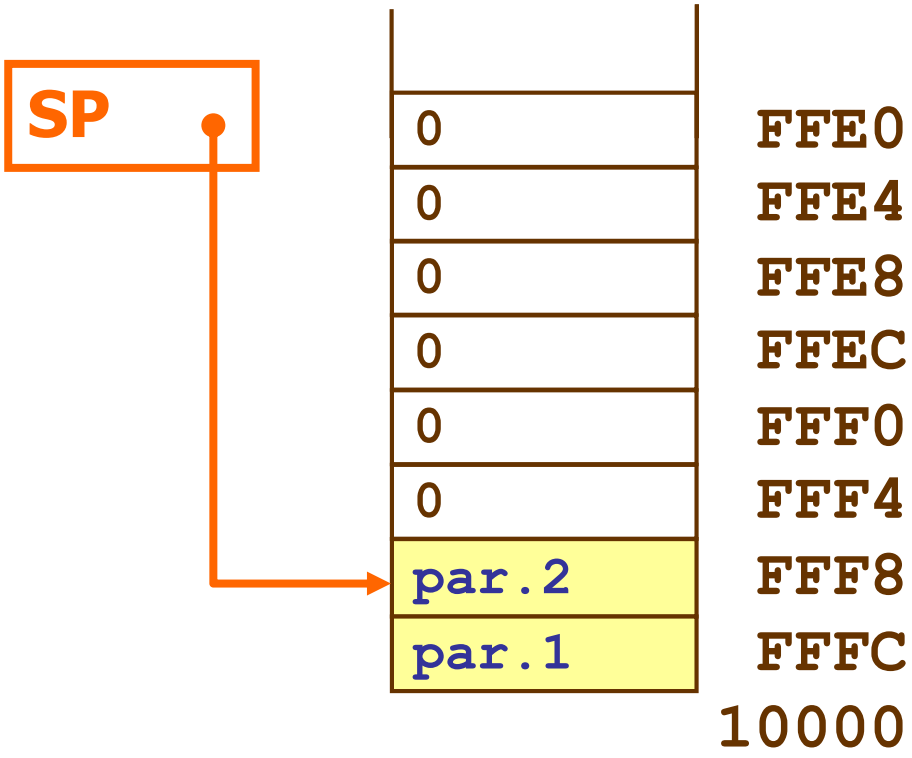
Potprogrami - Prijenos stogom



Potprogrami - Prijenos stogom

Izvodi se:

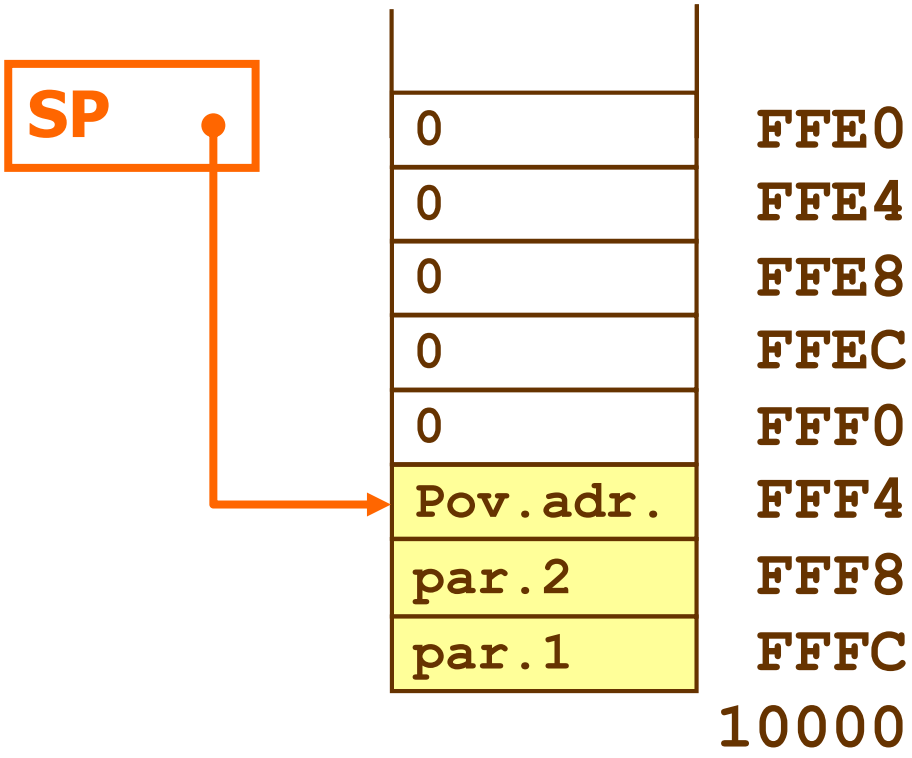
Glavni program stavlja dva parametra na stog



Potprogrami - Prijenos stogom

Izvodi se:

Glavni program poziva potprogram

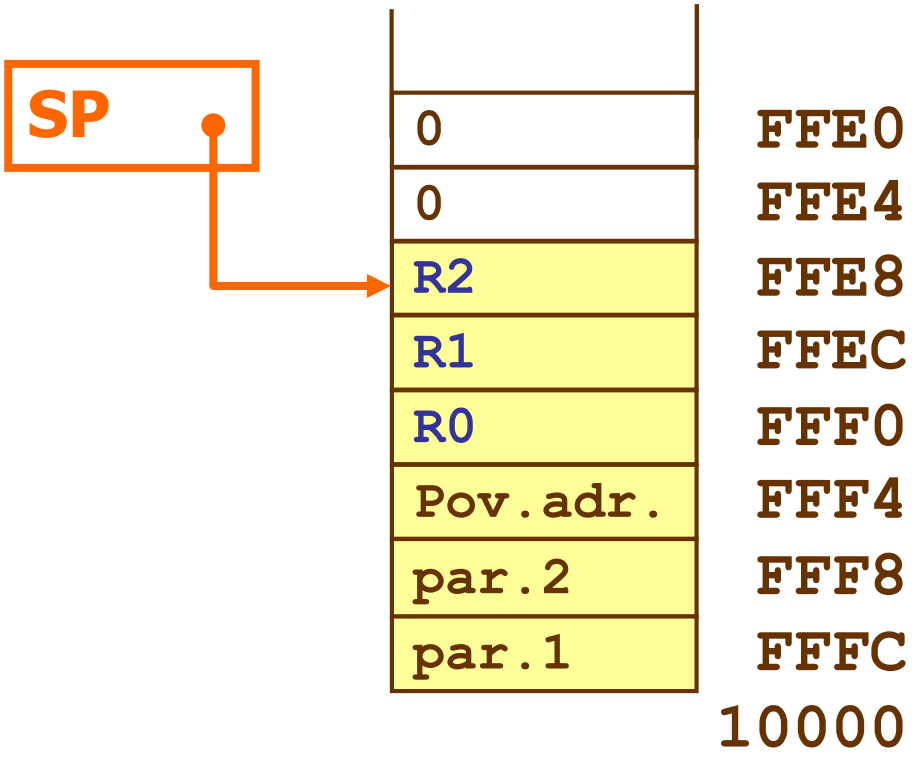


Potprogrami - Prijenos stogom

Izvodi se:

Potprogram sprema registre (koje će mijenjati) na stog

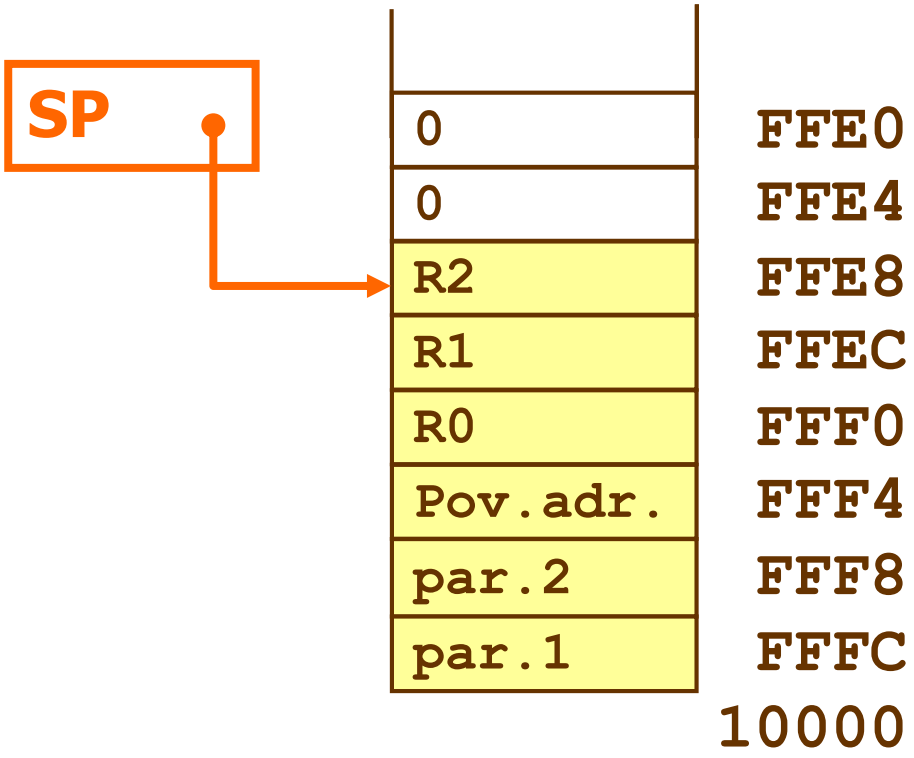
(npr. tri registra R0, R1, R2)



Potprogrami - Prijenos stogom

?

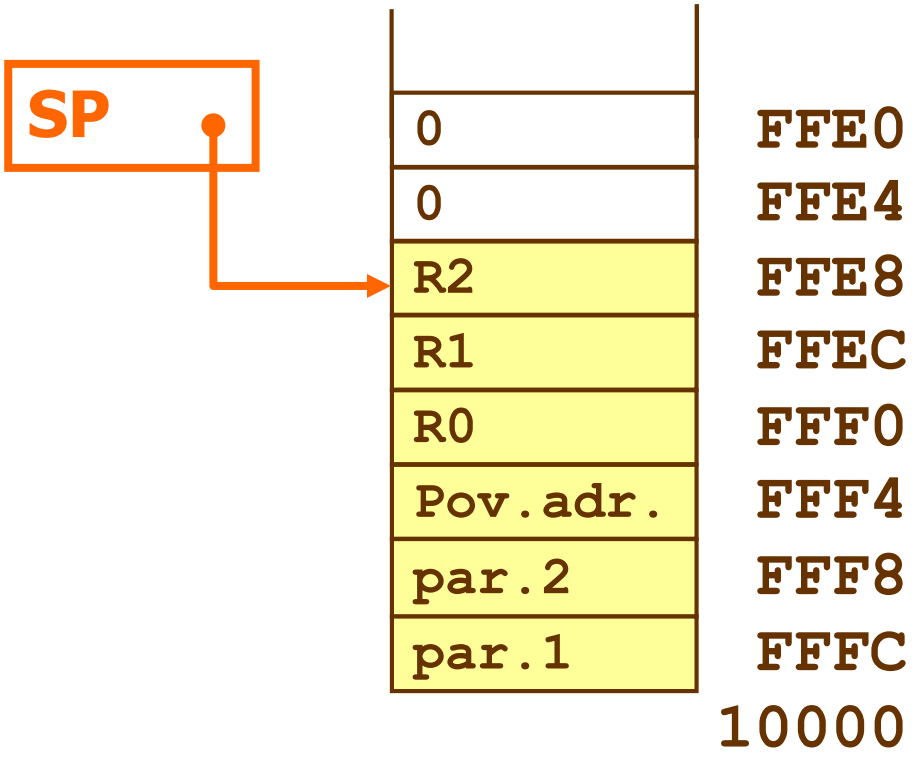
Kako dohvatiti parametre ?



Potprogrami - Prijenos stogom

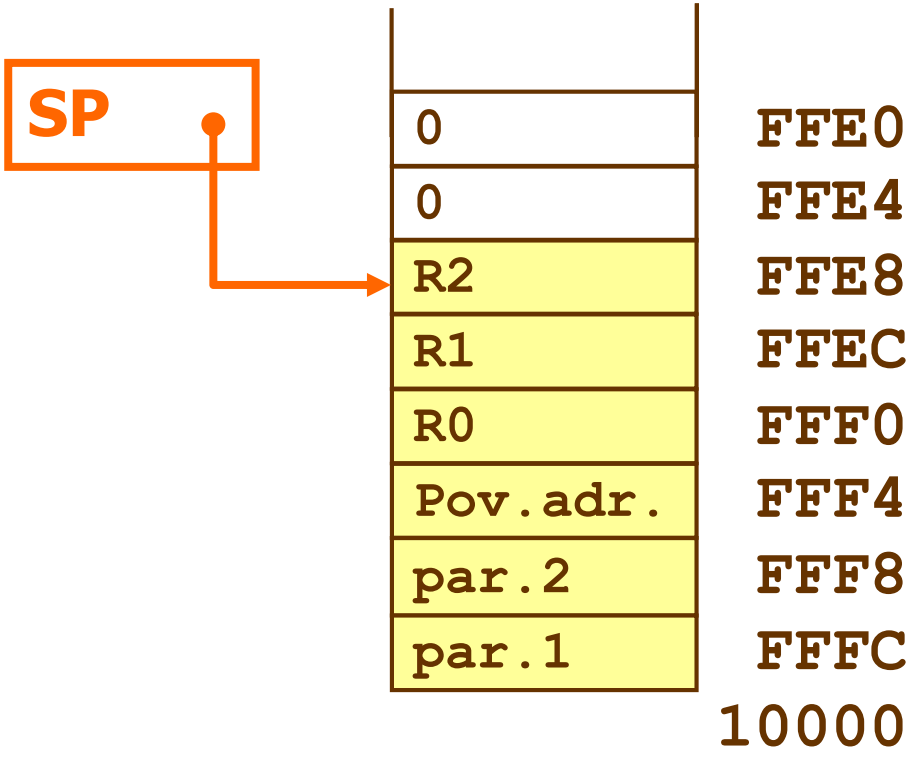
Da smo prije spremanja registara dohvatili parametre, onda bi uništili registre prije nego ih spremimo:

?



Potprogrami - Prijenos stogom

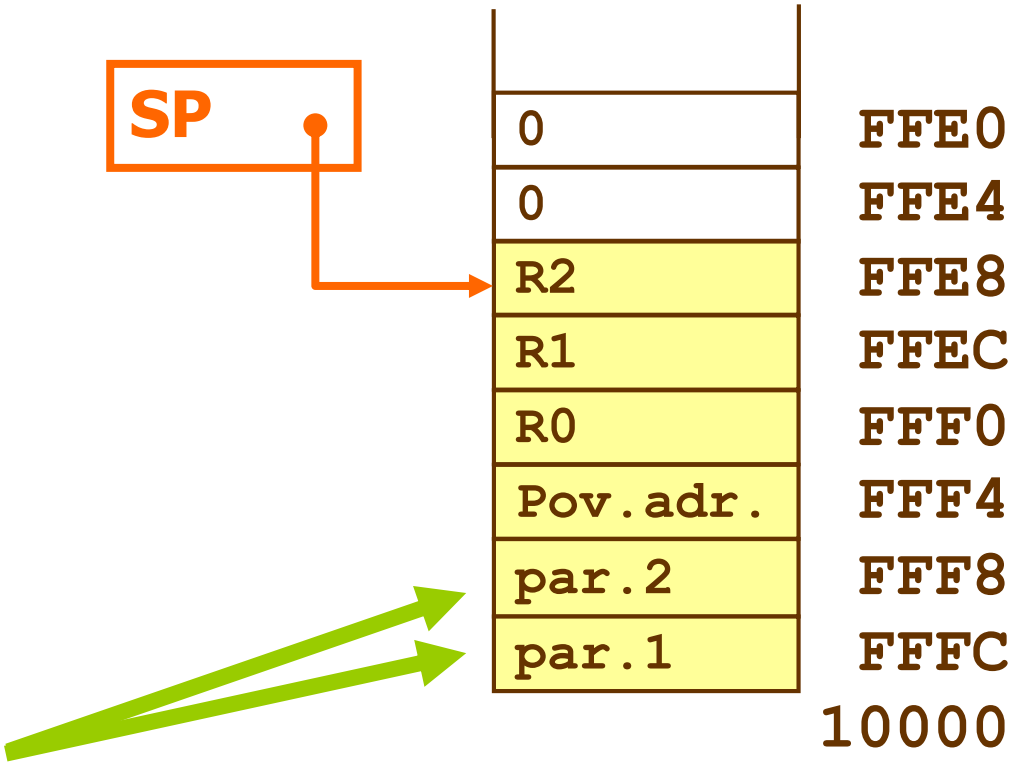
Jedna mogućnost (LOŠA!!!) je da presložimo podatke na stogu korištenjem fiksnih memorijskih lokacija (npr. tako da parametri dođu na vrh stoga)



Potprogrami - Prijenos stogom

Rješenje bi bilo kad ne bi morali pristupati podatcima na stogu samo pomoću PUSH i POP.

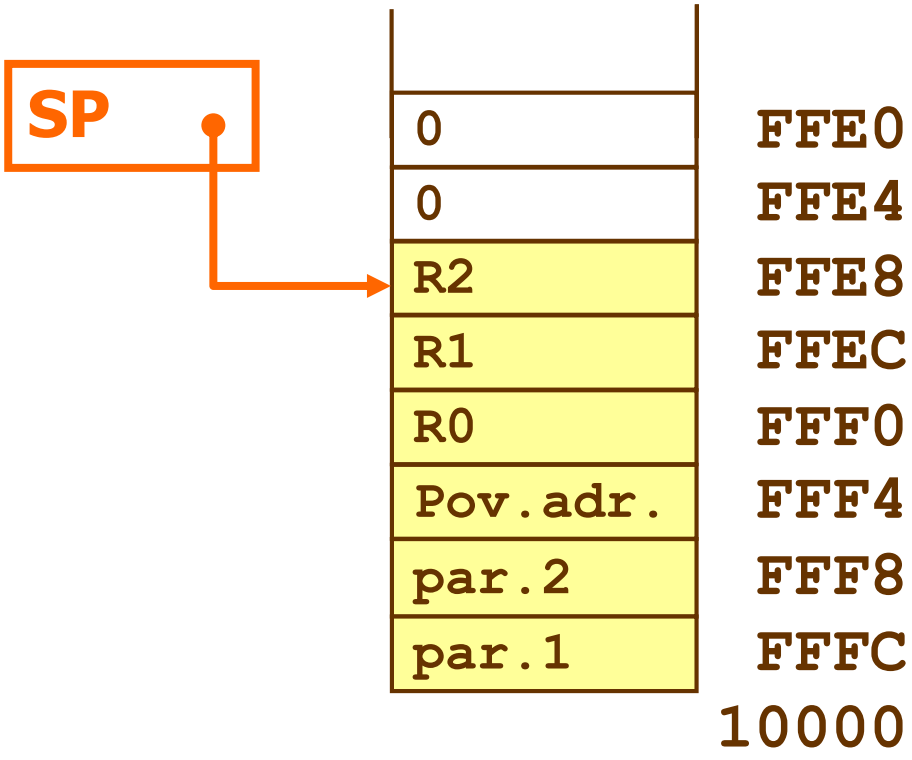
Kako možemo pristupiti podatcima "ispod" vrha stoga, bez pomicanja vrha stoga?



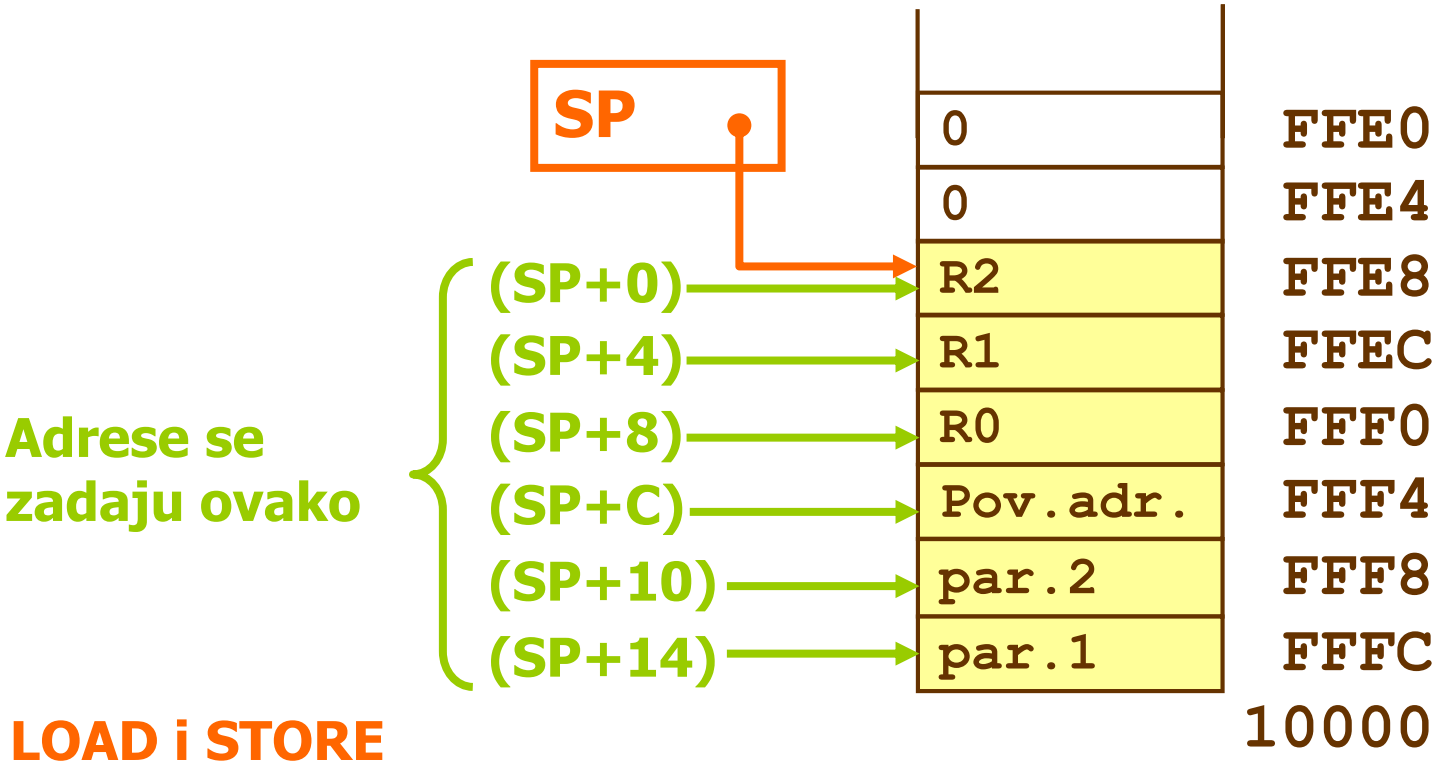
Potprogrami - Prijenos stogom

Rješenje je jednostavno:

Registarsko indirektno adresiranje s odmakom, pri čemu kao adresni registar koristimo SP



Potprogrami - Prijenos stogom



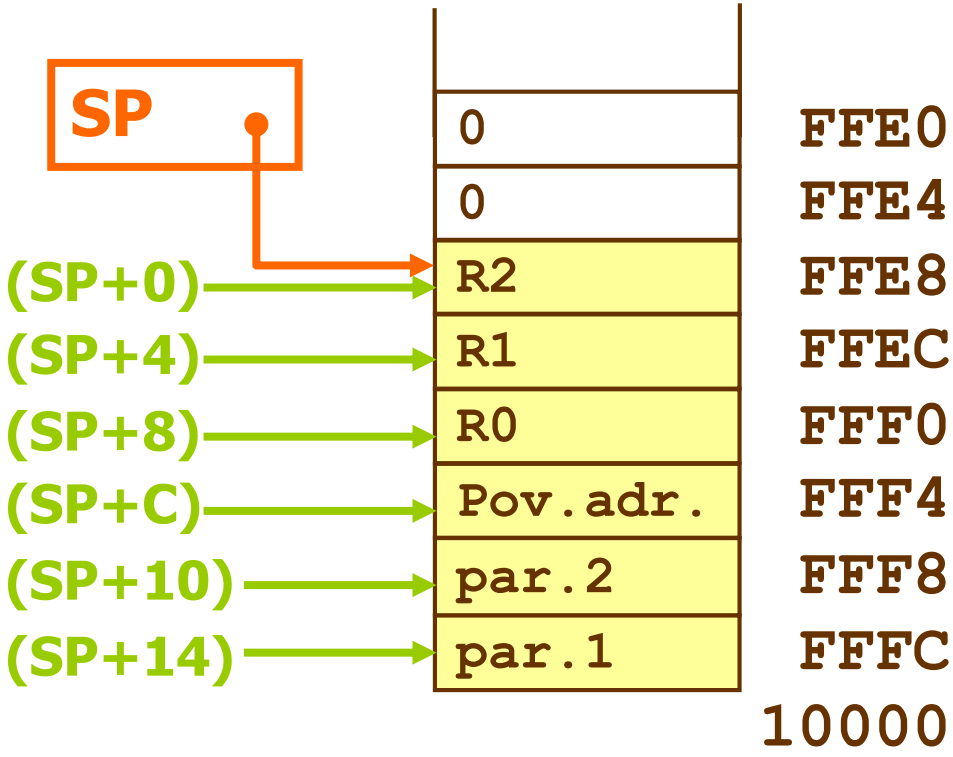
Adrese se zadaju ovako

Naredbama LOAD i STORE možemo pristupati podatcima "unutar" stoga.

Potprogrami - Prijenos stogom

- Dok pišemo potprogram, točno znamo koliko on parametara ima i koliko registara mora spremiti na stog.

- Zato lako možemo izračunati odmake pojedinih parametara u odnosu na SP (vrh stoga).



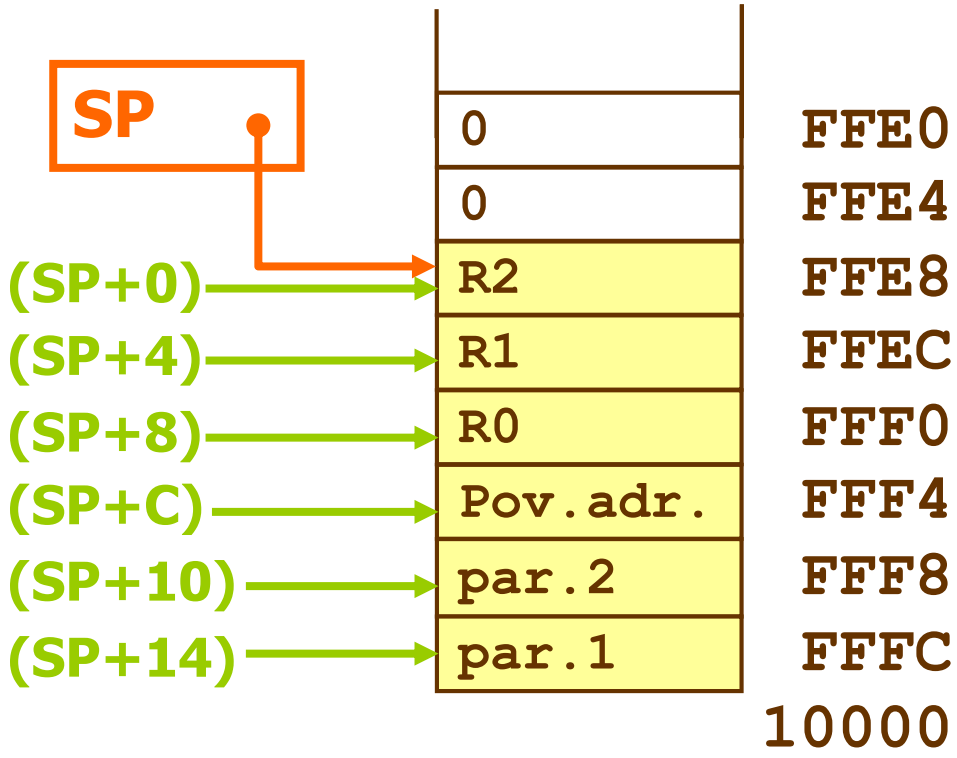
Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Zadnji problem je povratna vrijednost.

Neka je upravo završeno izračunavanje rezultata potprograma i neka je rezultat npr. u registru R0.

Želimo se vratiti iz potprograma:



Potprogrami - Prijenos stogom

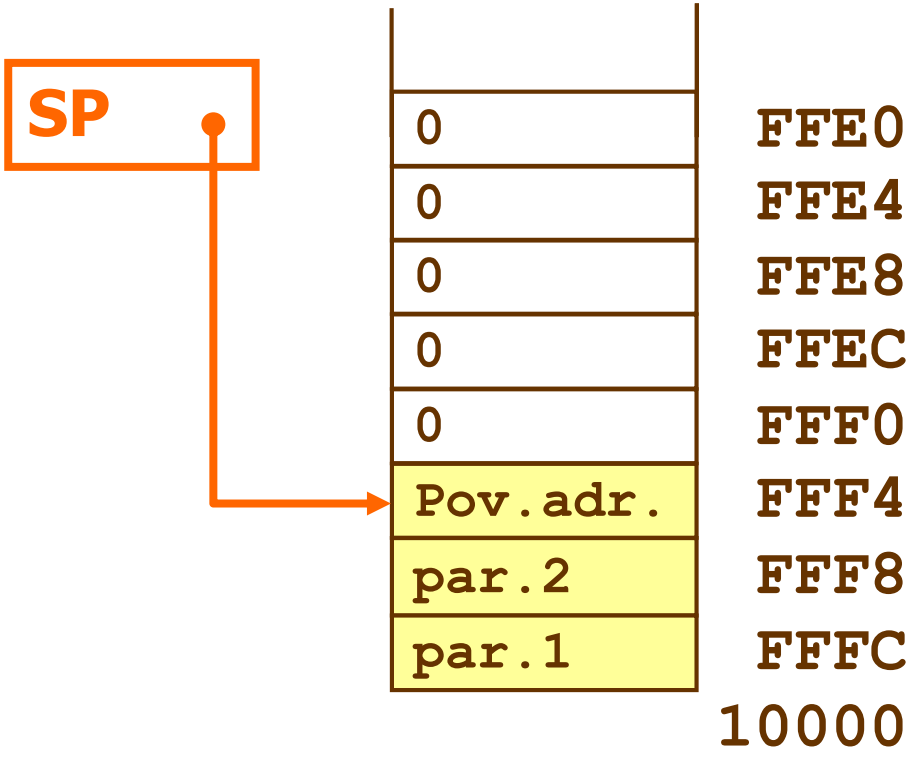
Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Izvodi se:

Obnavljaju se registri R0, R1 i R2.

Povratak se lako ostvaruje sa RET

Ali, obnavljanjem R0 gubi se rezultat !!!



Potprogrami - Prijenos stogom

Rješenje koje se koristi u praksi:

- Za **povratak rezultata** iz potprograma koristi se **prijenos registrom**, a ne stogom.

- Ovo rješenje je efikasno i jednostavno. Ograničenje je da se može vratiti samo jedan rezultat, ali je u praksi to obično dovoljno.

- Obično se odabire jedan registar koji svi potprogrami koriste za povratak vrijednosti, a na mjestu pozivanja se zna da u tom registru ne smije biti koristan podatak.

Potprogrami - Prijenos stogom

Napisati potprogram koji izračunava logičku operaciju NILI. Parametri se prenose stogom, a rezultat se vraća registrom R0. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat sprema natrag u memoriju.

Potprogram mora čuvati vrijednosti svih registara koje mijenja.

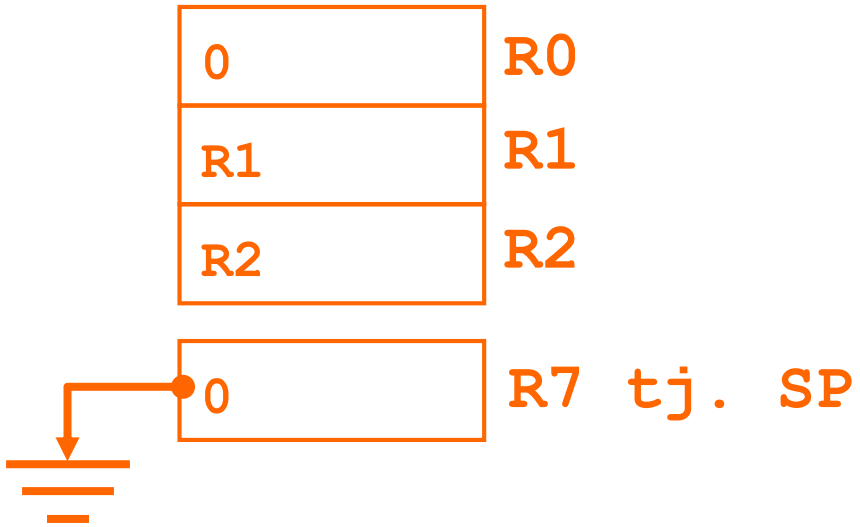
Rješenje:

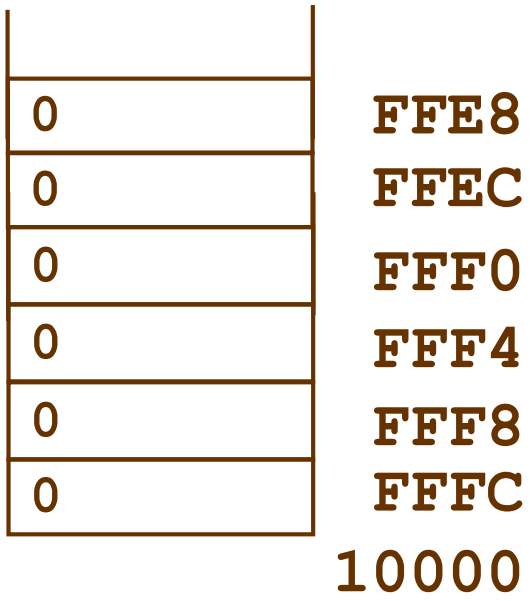
- U ovom rješenju potprogram mijenja registre R1 i R2 i sprema ih na stog (R0 se također mijenja, ali preko njega se ionako vraća povratna vrijednost)
- Parametrima se pristupa registarskim indirektnim adresiranjem s odmakom (korištenjem SP-a kao adresnog registra)

>>>>

<<<< Izvođenje programa:

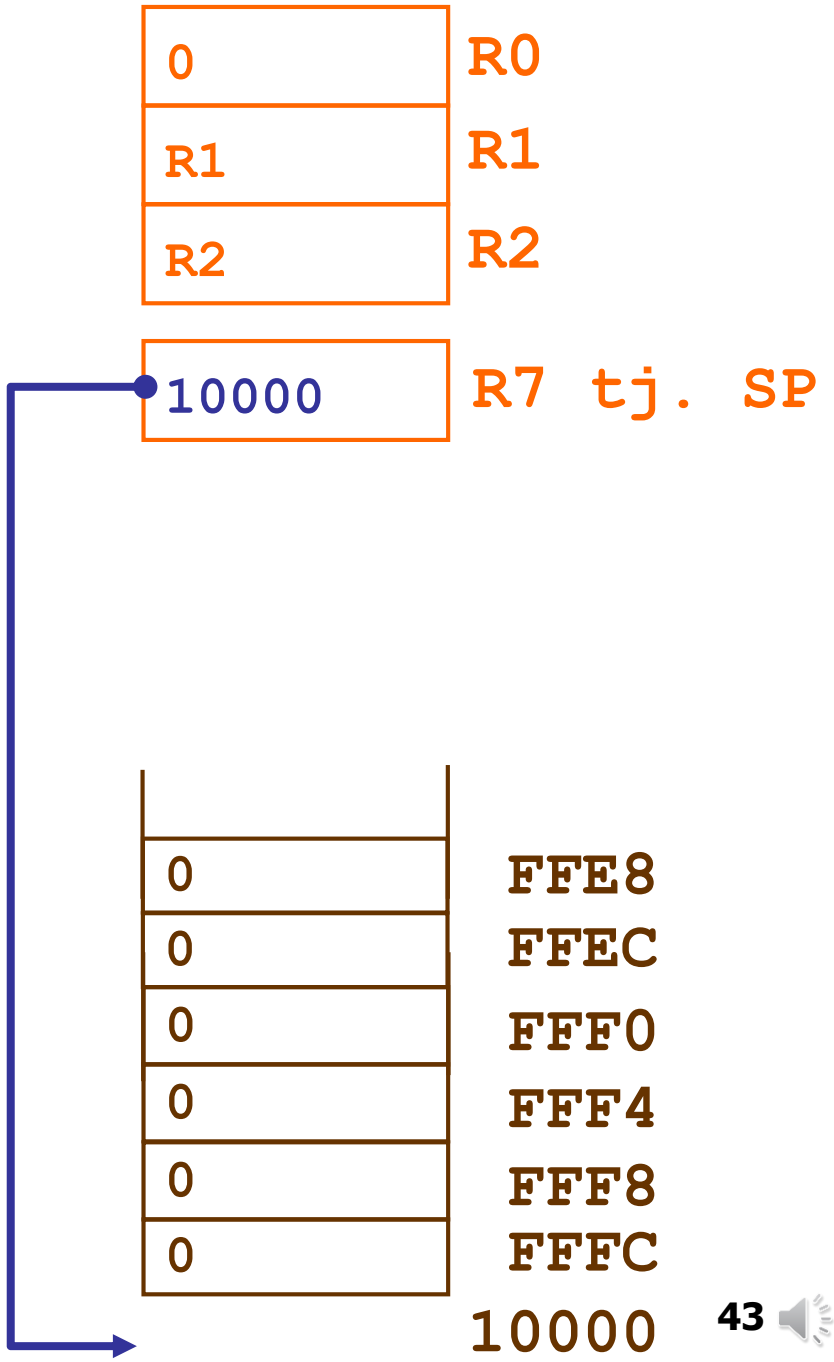
```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       STORE  R0, (REZ)

       ADD    SP, 8, SP

       HALT
```





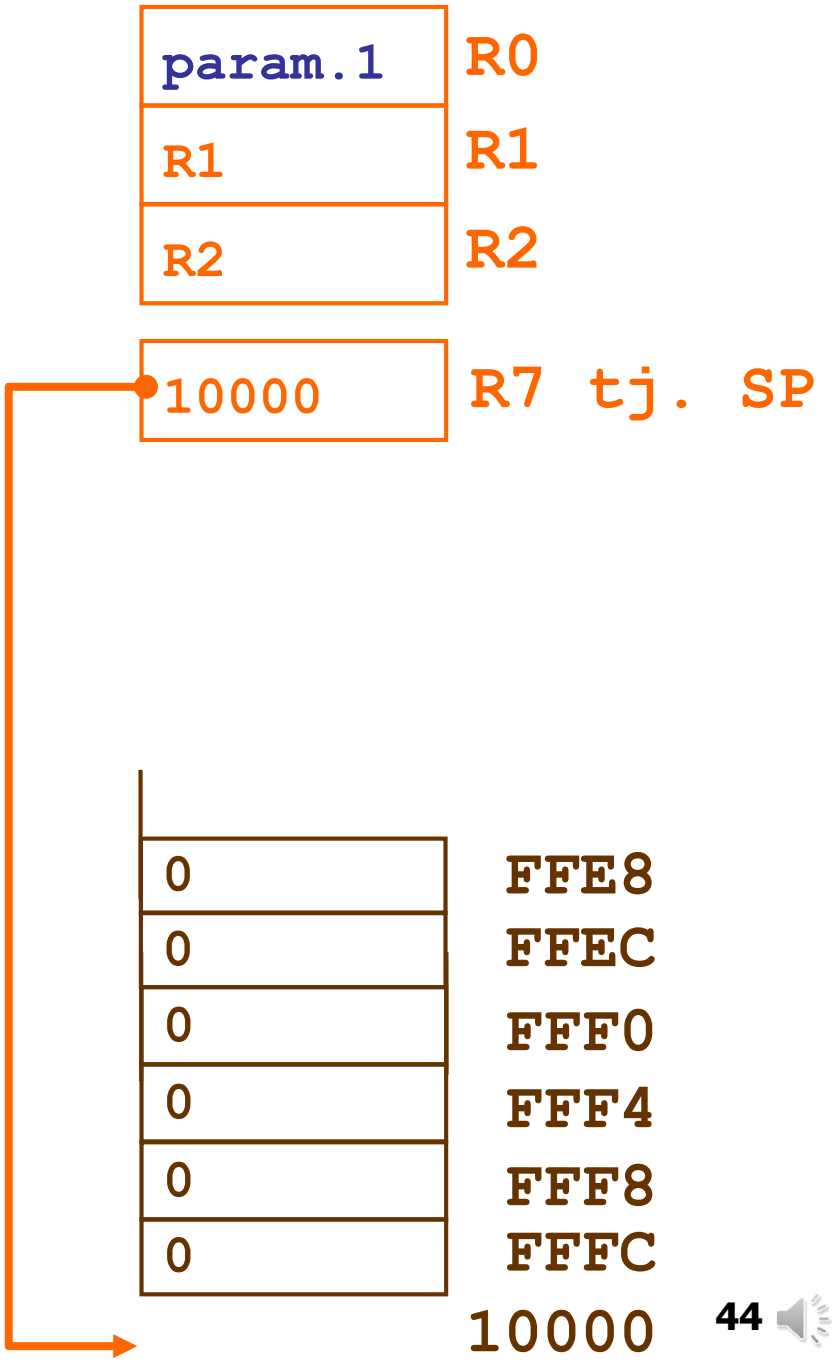
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP
       LOAD   R0, (PRVI)
       PUSH   R0
       LOAD   R0, (DRUGI)
       PUSH   R0
       CALL   NILI
       STORE  R0, (REZ)
       ADD    SP, 8, SP
       HALT
```



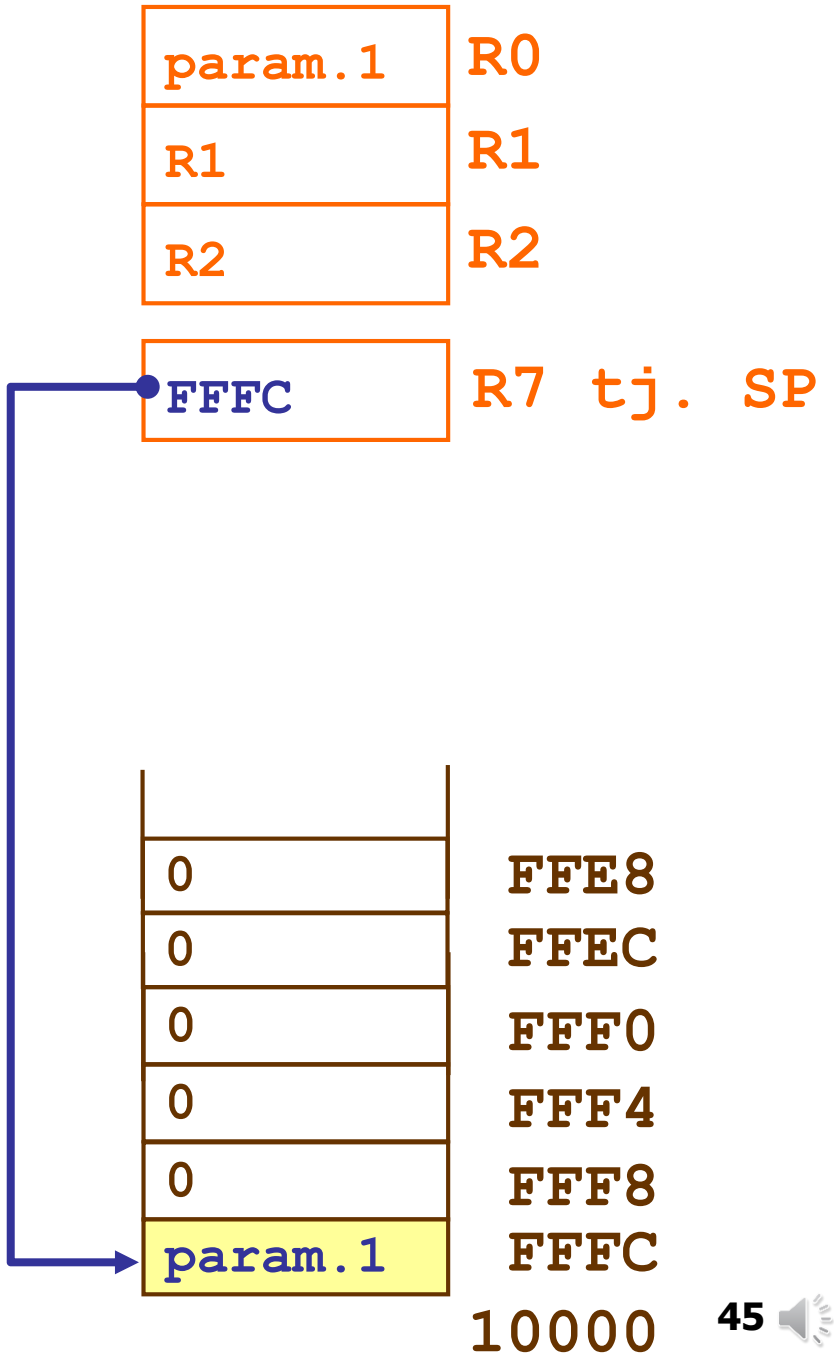
<<<< Izvođenje programa:

```
GLAVNI MOVE  10000, SP

       LOAD  R0, (PRVI)
       PUSH  R0

       LOAD  R0, (DRUGI)
       PUSH  R0

       CALL  NILI

       STORE R0, (REZ)

       ADD   SP, 8, SP

       HALT
```



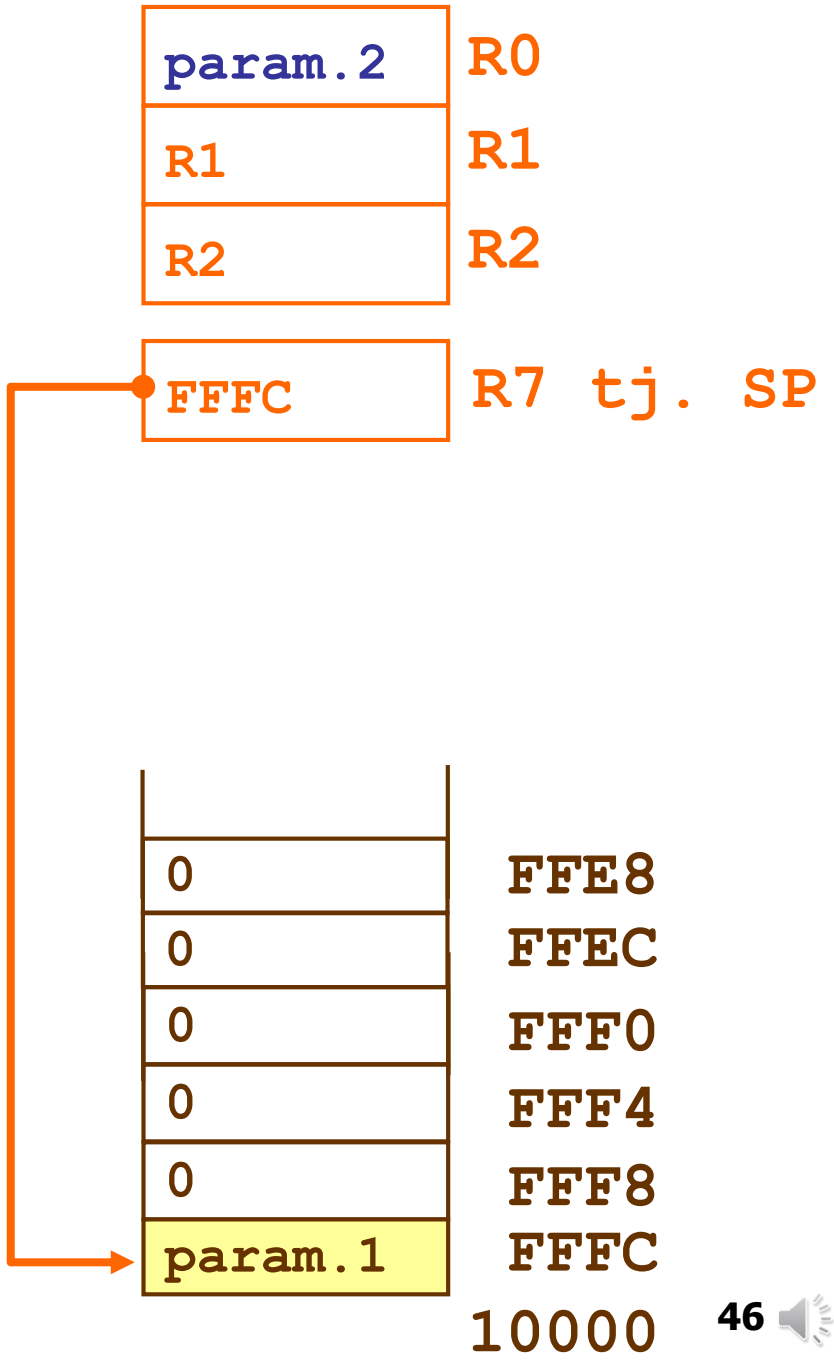
<<<< Izvođenje programa:

```
GLAVNI MOVE  10000, SP

       LOAD  R0, (PRVI)
       PUSH  R0

       LOAD  R0, (DRUGI)
       PUSH  R0

       CALL  NILI

       STORE R0, (REZ)

       ADD   SP, 8, SP

       HALT
```



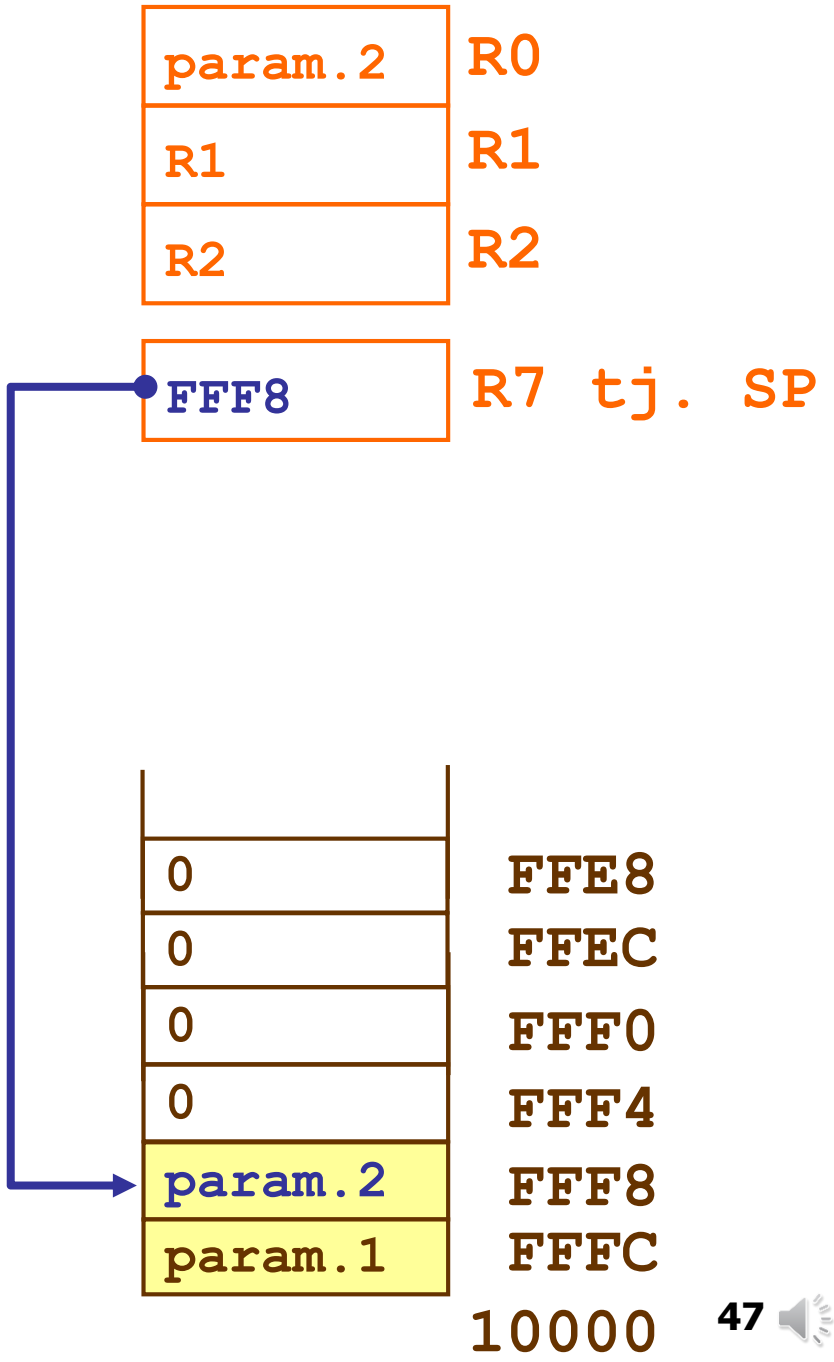
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       STORE  R0, (REZ)

       ADD    SP, 8, SP

       HALT
```



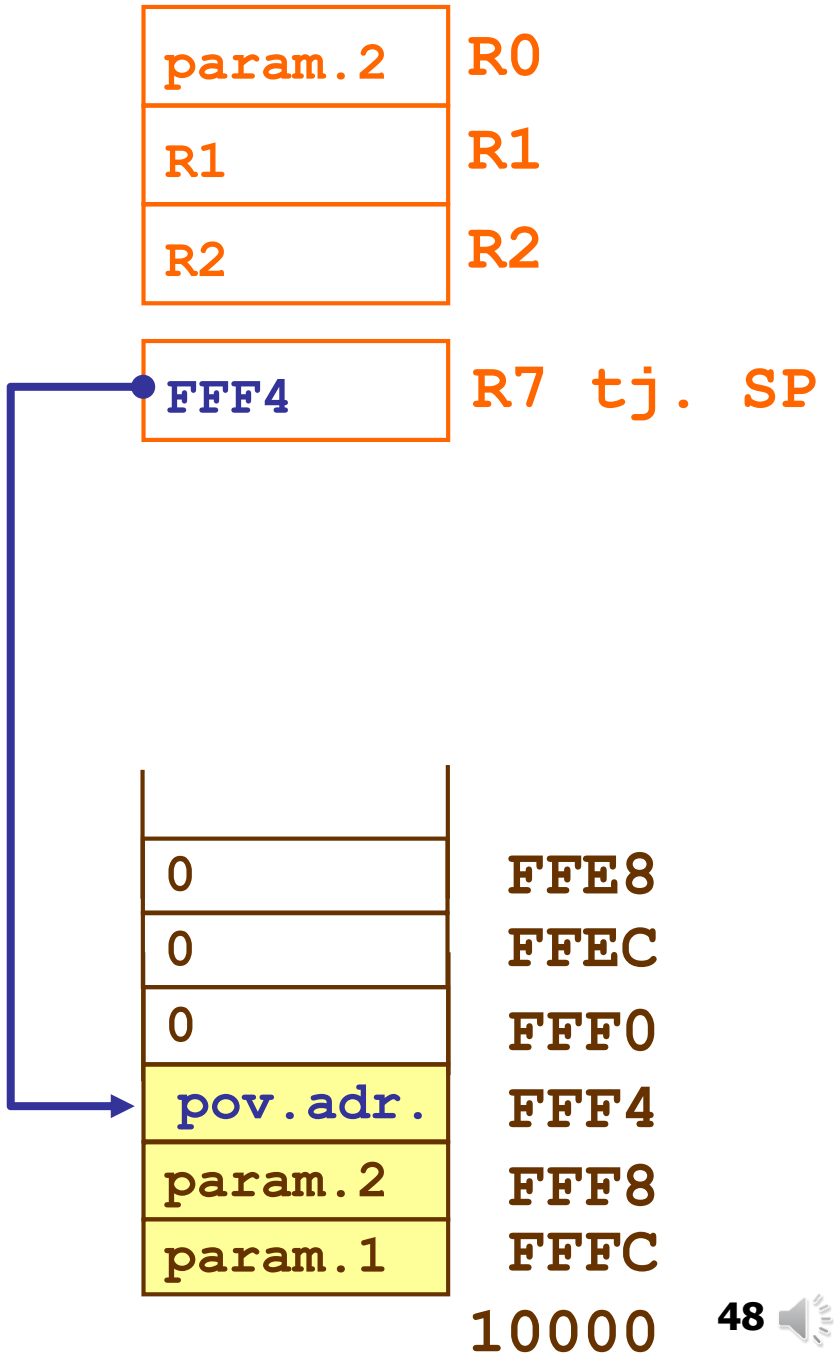
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       STORE  R0, (REZ)

       ADD    SP, 8, SP

       HALT
```



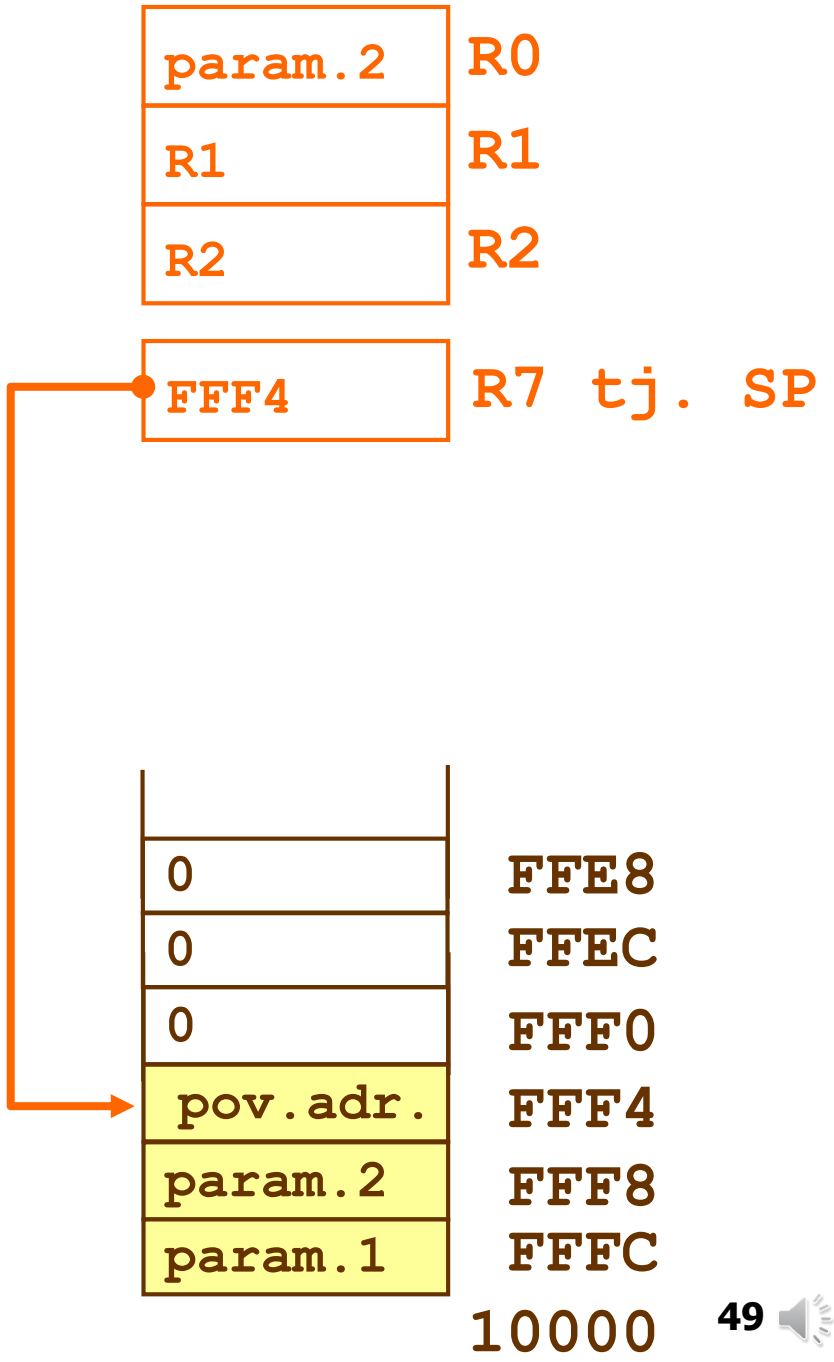
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       STORE  R0, (REZ)

       ADD    SP, 8, SP

       HALT
```



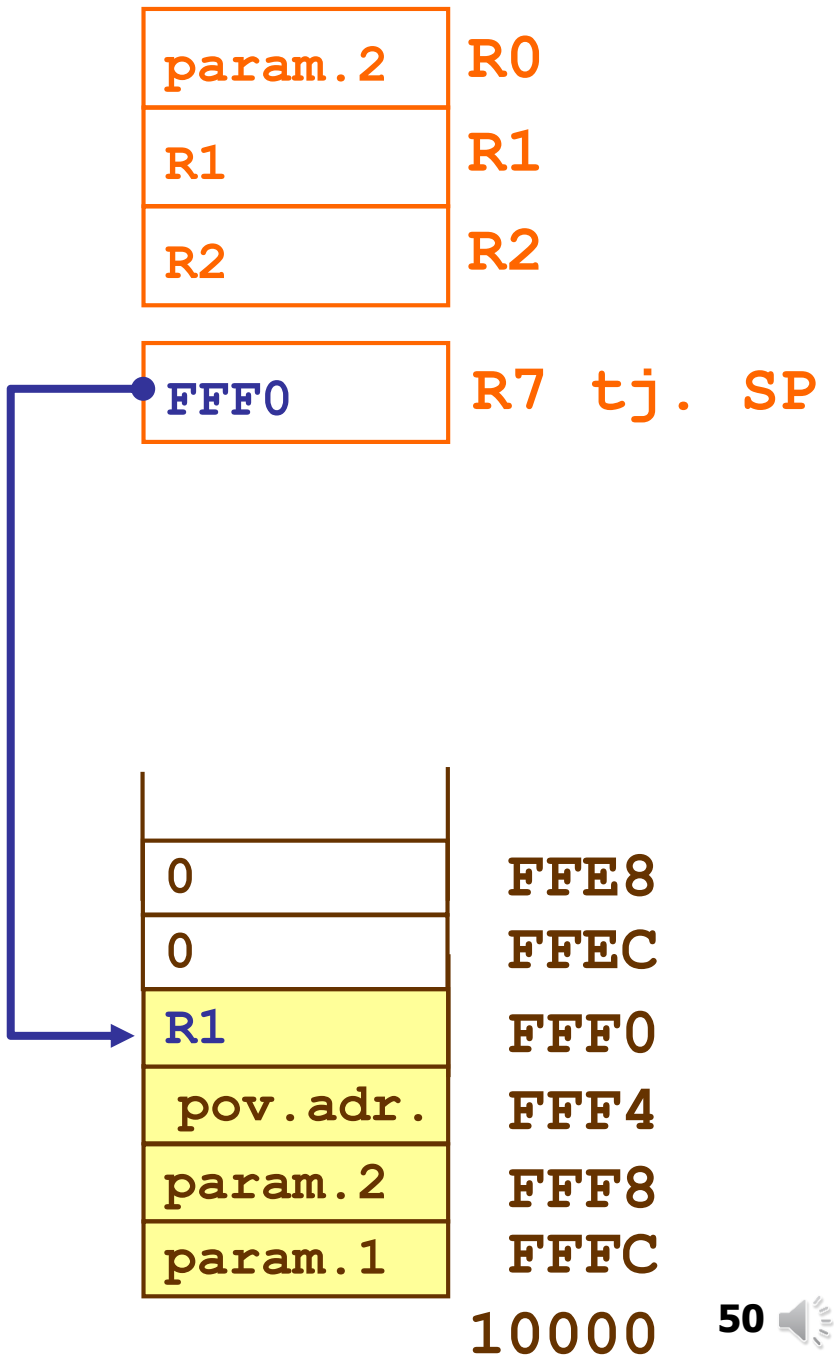
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



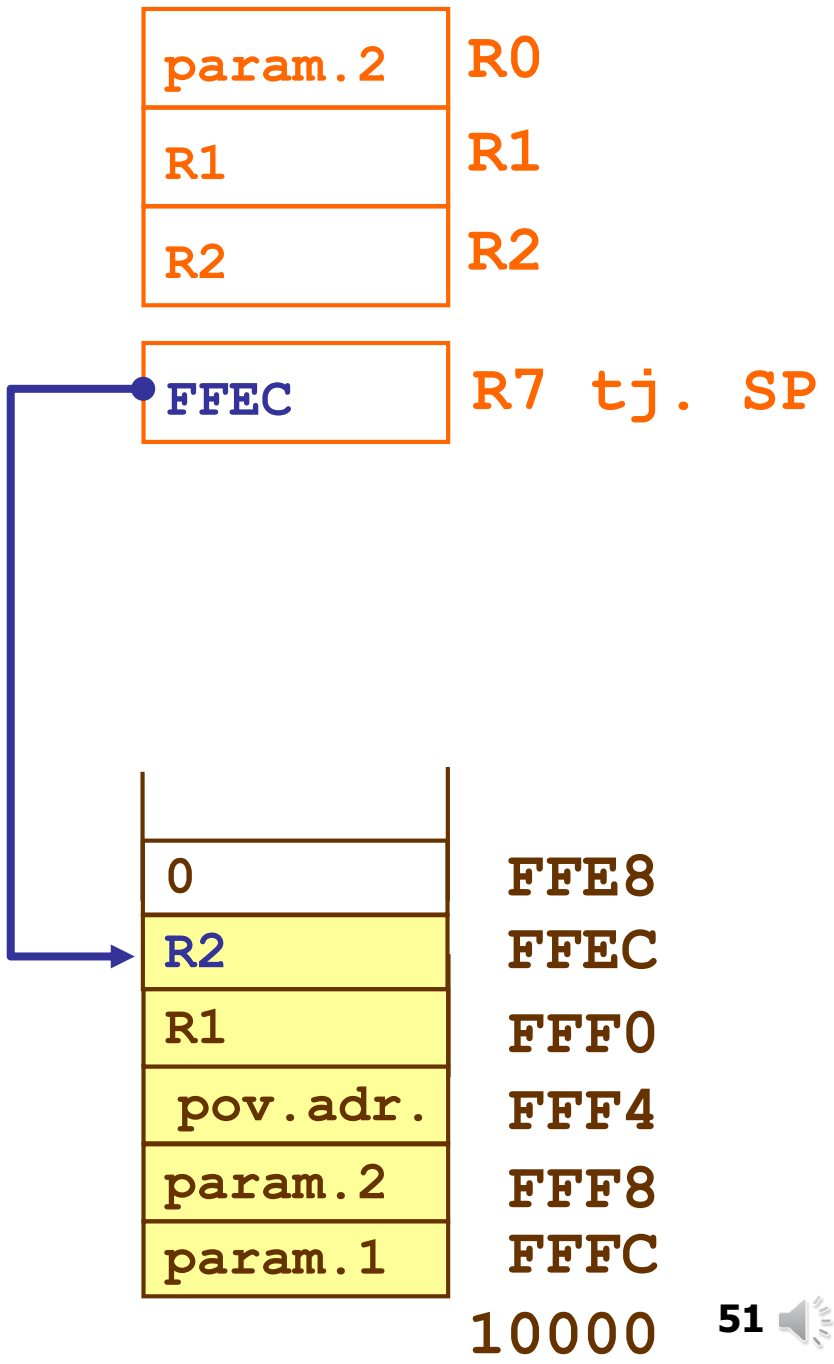
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



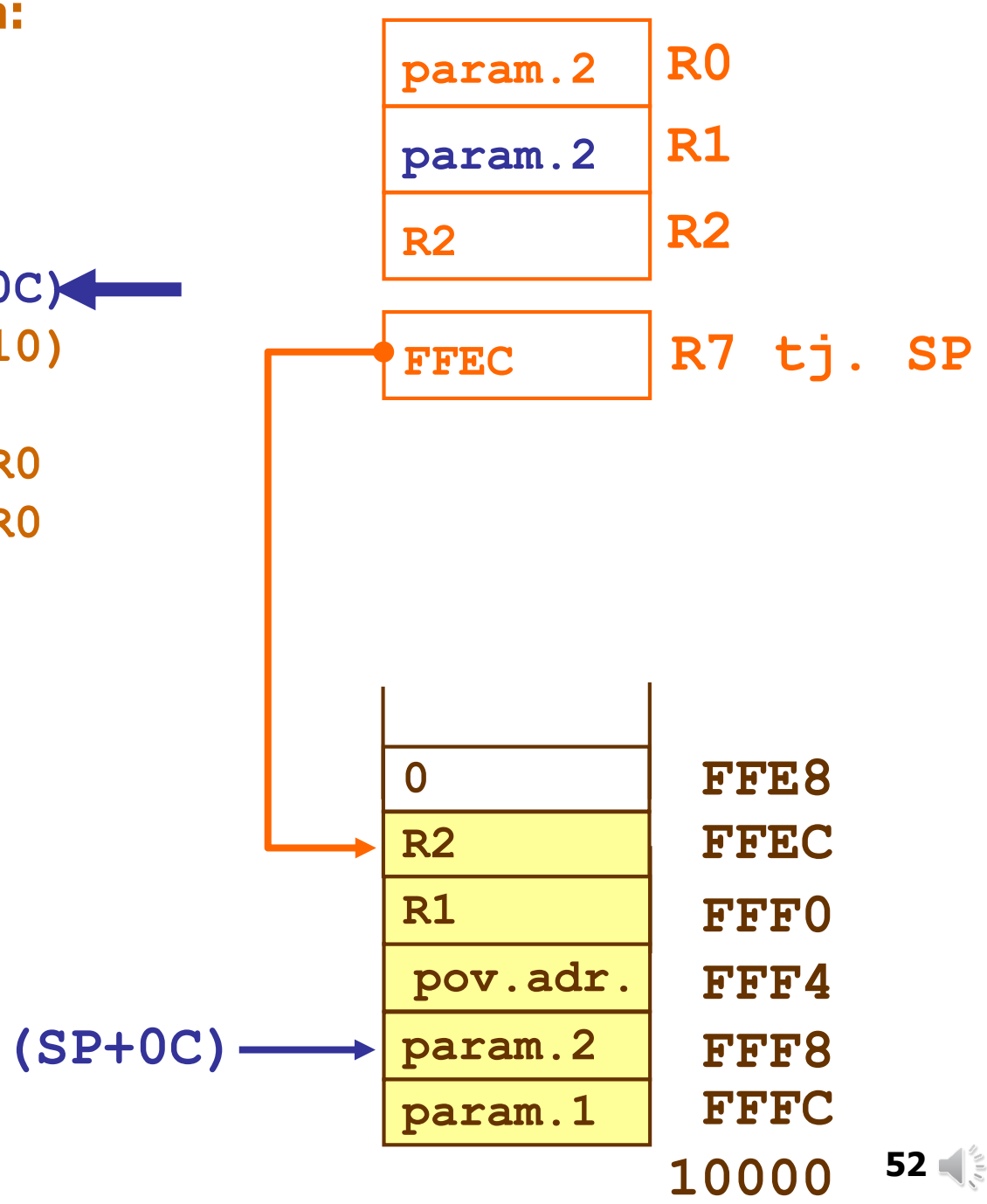
<<<< Izvođenje programa:

```
NILI    PUSH  R1
        PUSH  R2

        LOAD  R1, (SP+0C)
        LOAD  R2, (SP+10)

        OR    R1, R2, R0
        XOR   R0, -1, R0

        POP R2
        POP R1

        RET
```



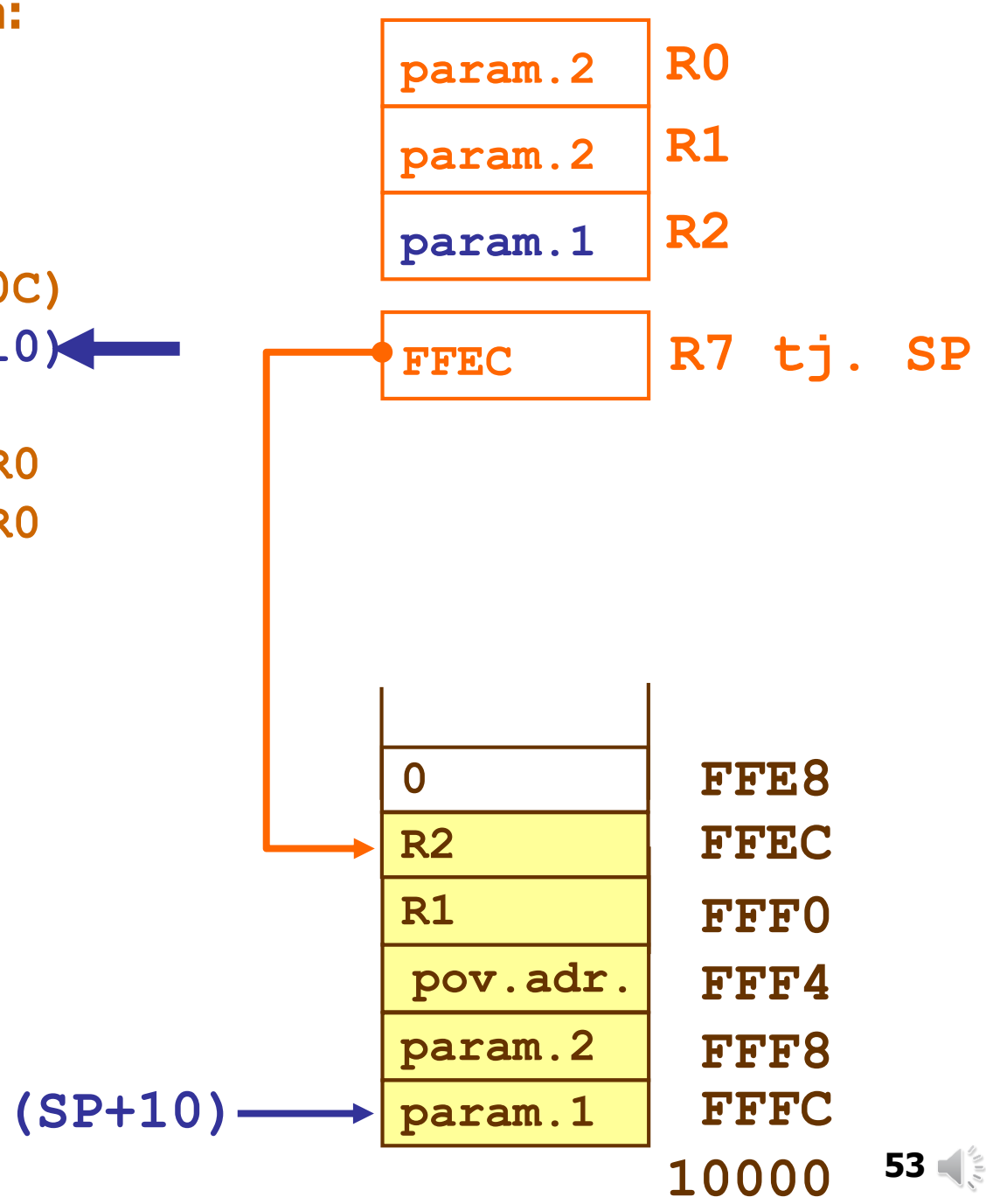
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



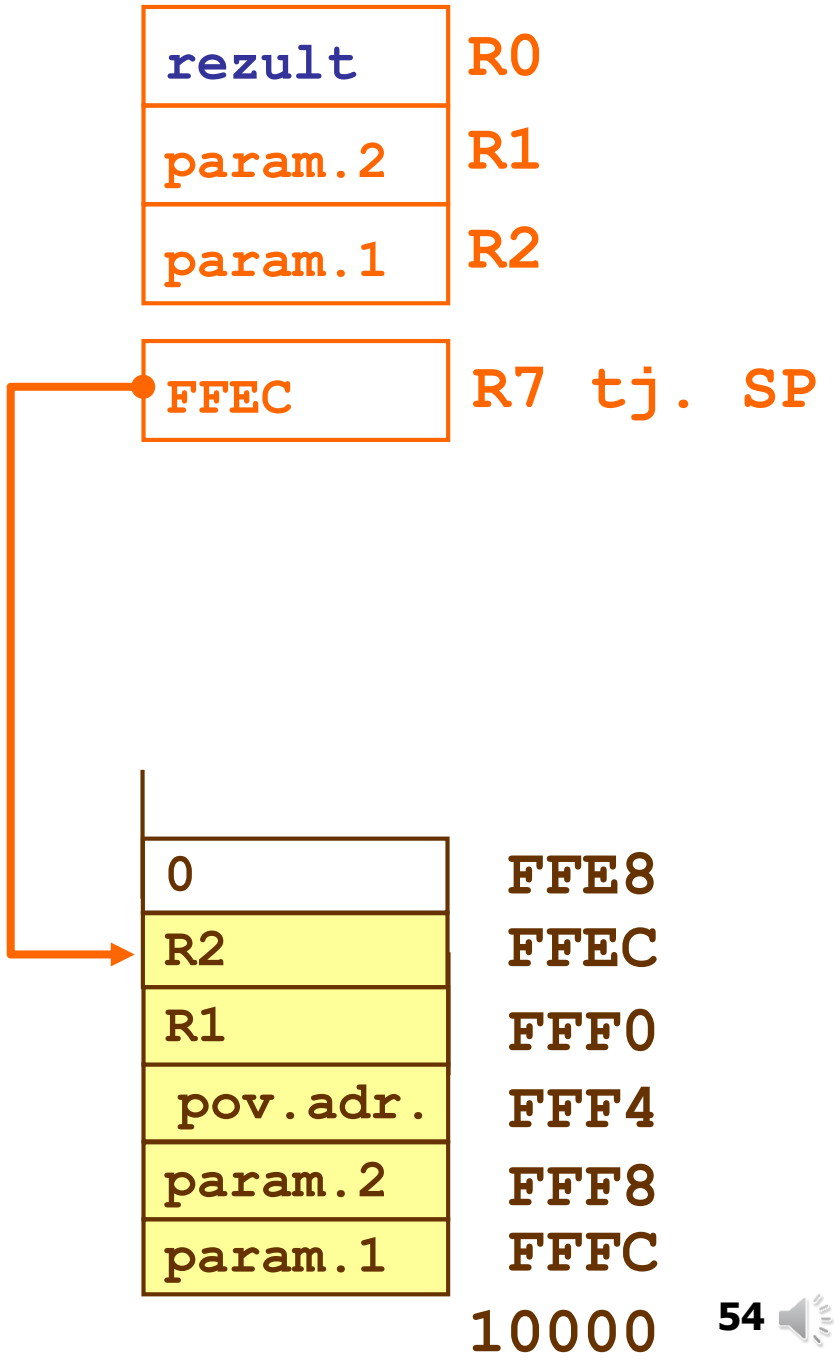
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



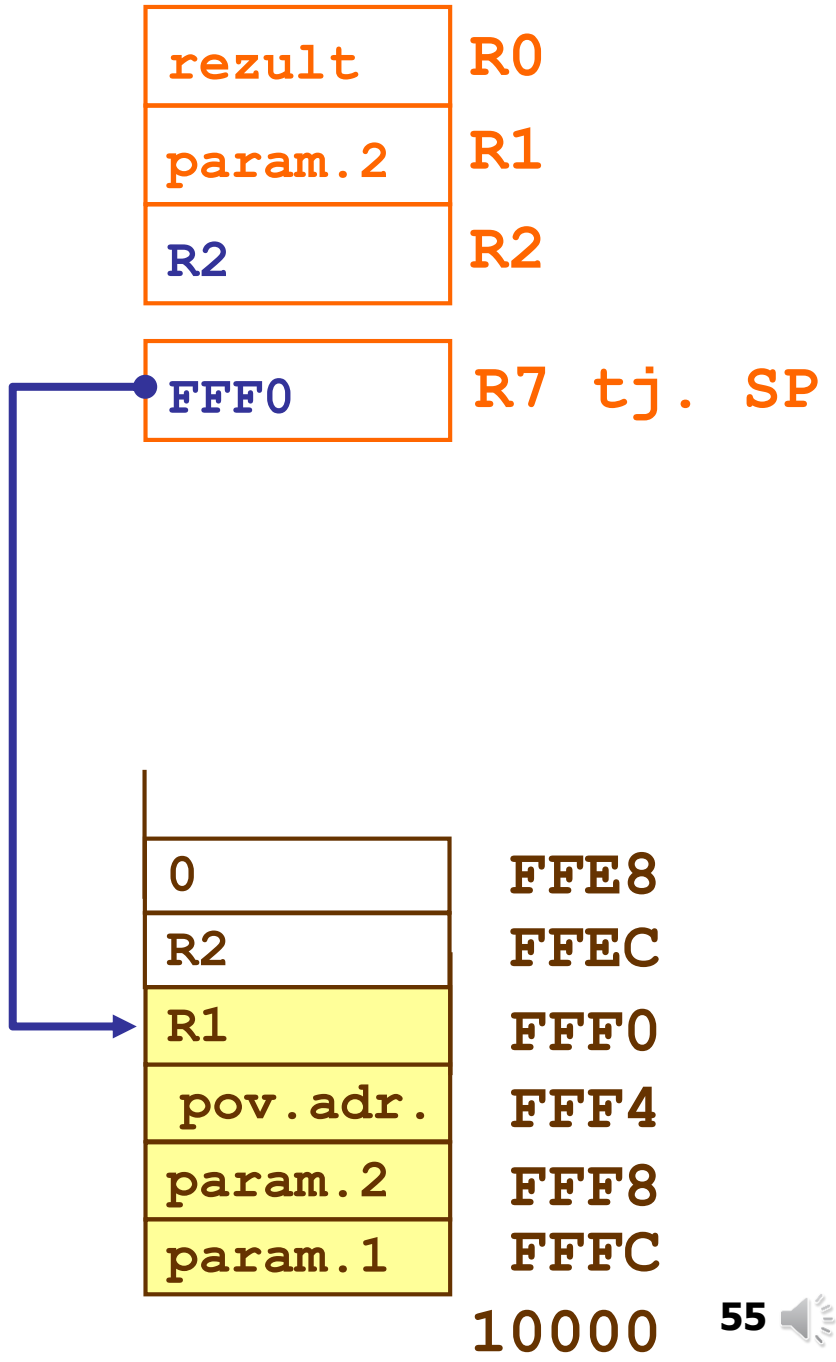
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



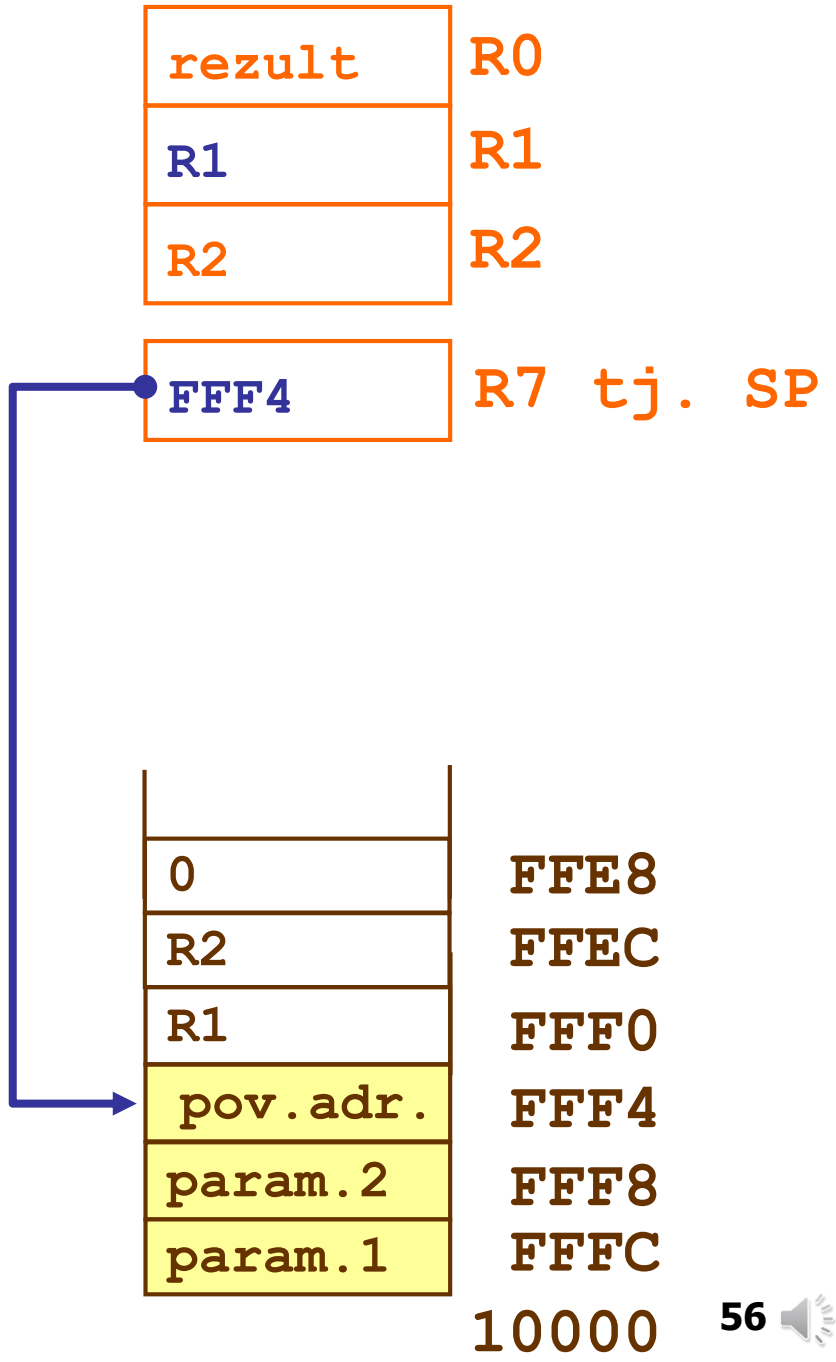
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



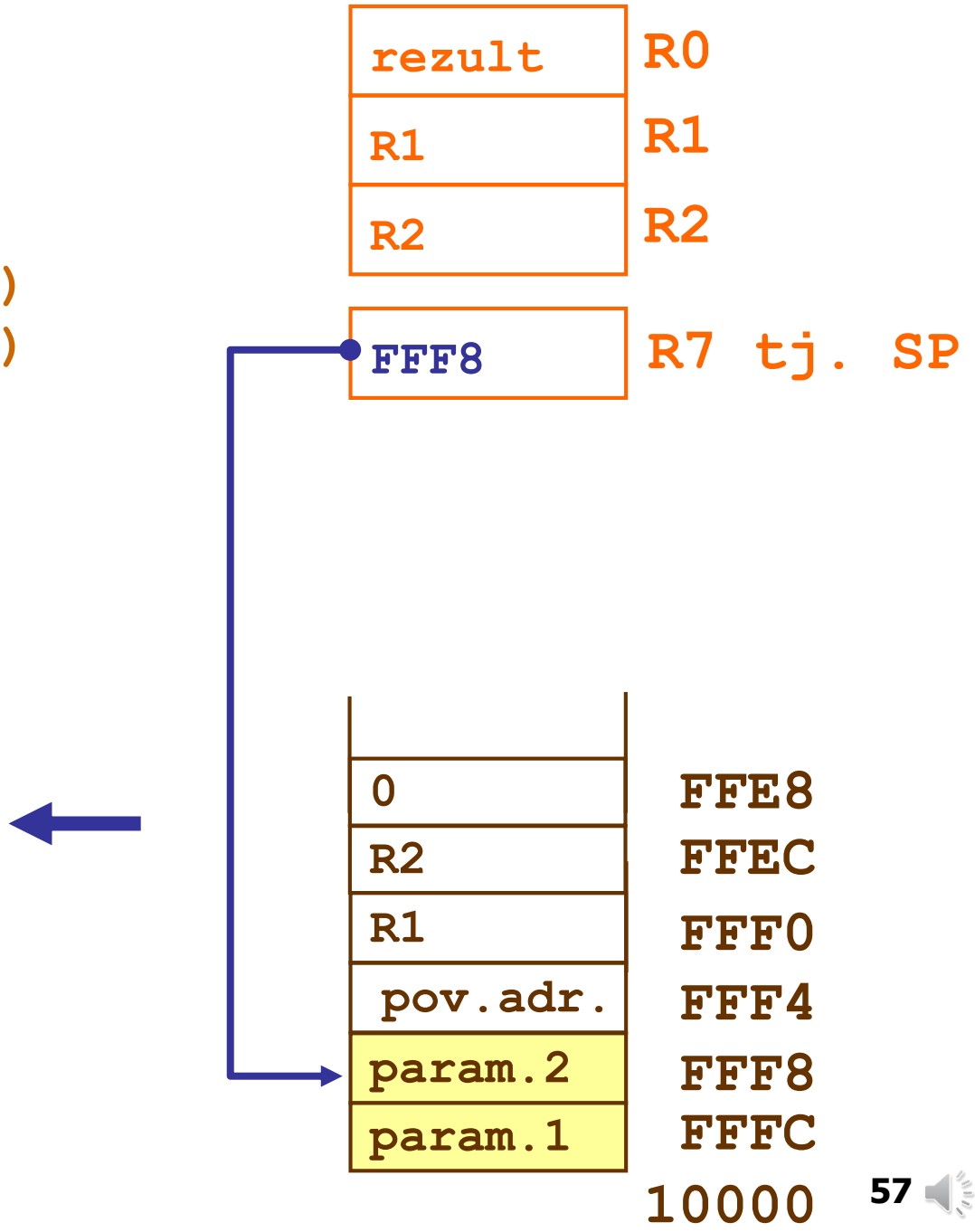
<<<< Izvođenje programa:

```
NILI    PUSH  R1
        PUSH  R2

        LOAD  R1, (SP+0C)
        LOAD  R2, (SP+10)

        OR    R1, R2, R0
        XOR   R0, -1, R0

        POP R2
        POP R1

        RET
```



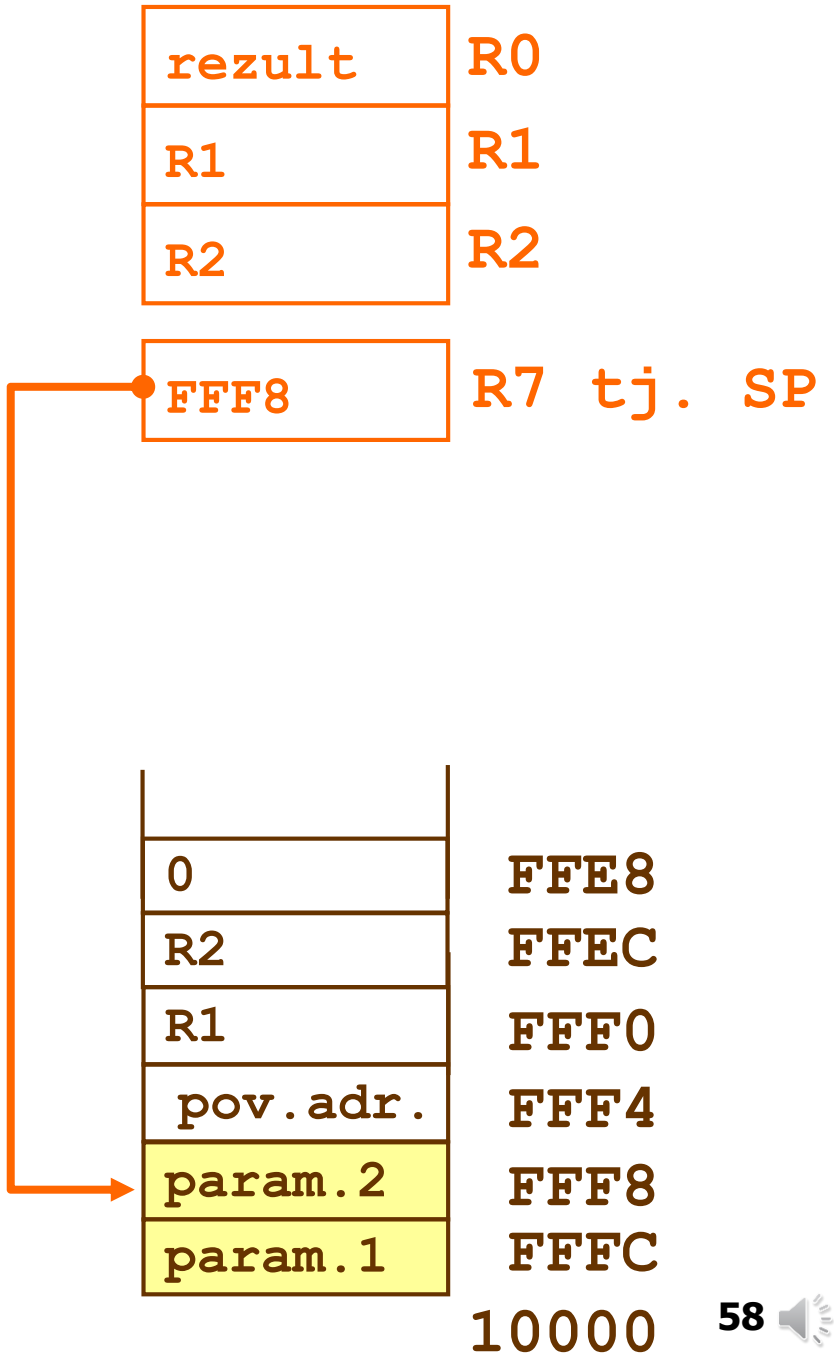
<<<< Izvođenje programa:

```
NILI    PUSH   R1
        PUSH   R2

        LOAD   R1, (SP+0C)
        LOAD   R2, (SP+10)

        OR     R1, R2, R0
        XOR    R0, -1, R0

        POP R2
        POP R1

        RET
```



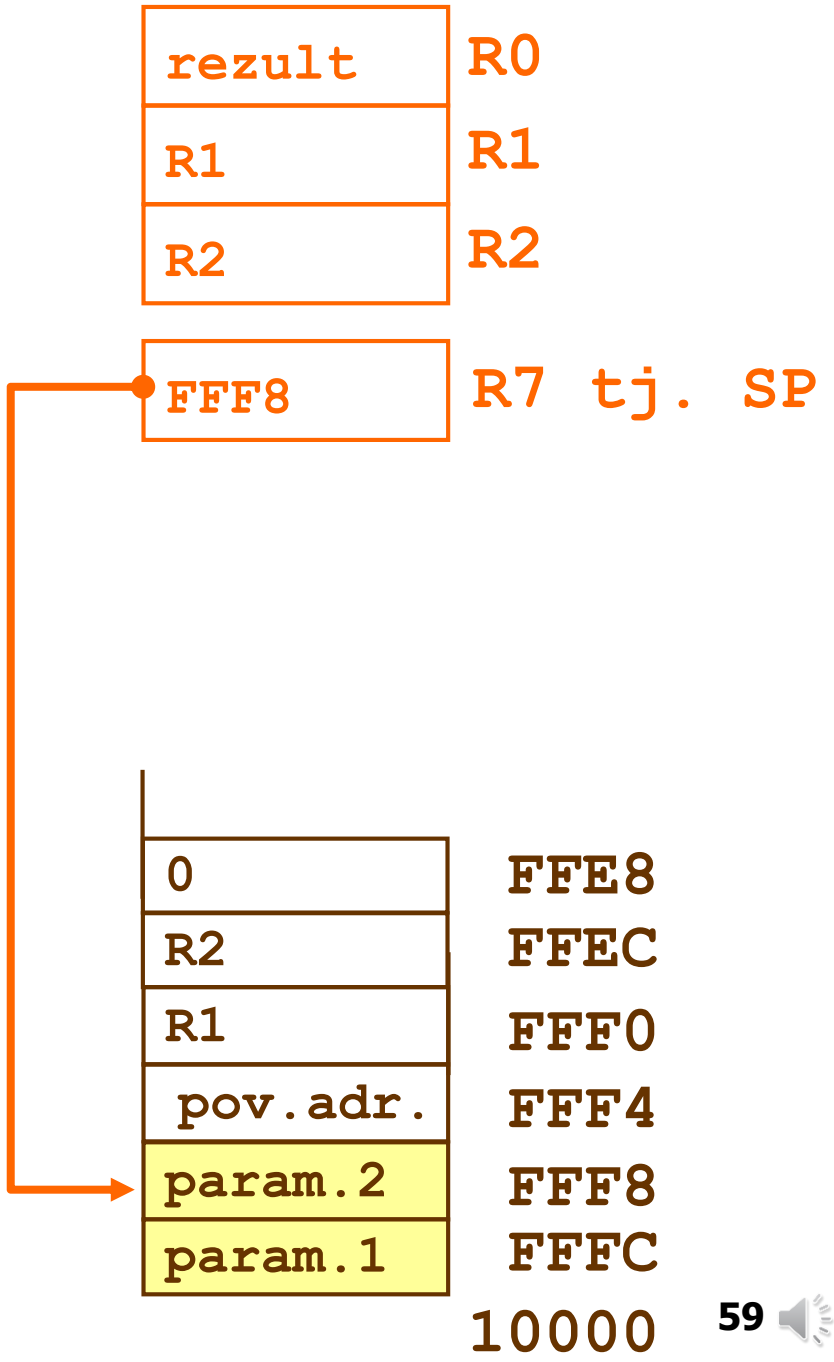
<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       STORE  R0, (REZ)

       ADD    SP, 8, SP

       HALT
```



<<<< Izvođenje programa:

```
GLAVNI MOVE   10000, SP

       LOAD   R0, (PRVI)
       PUSH   R0

       LOAD   R0, (DRUGI)
       PUSH   R0

       CALL   NILI

       STORE  R0, (REZ)

       ADD    SP, 8, SP

       HALT
```

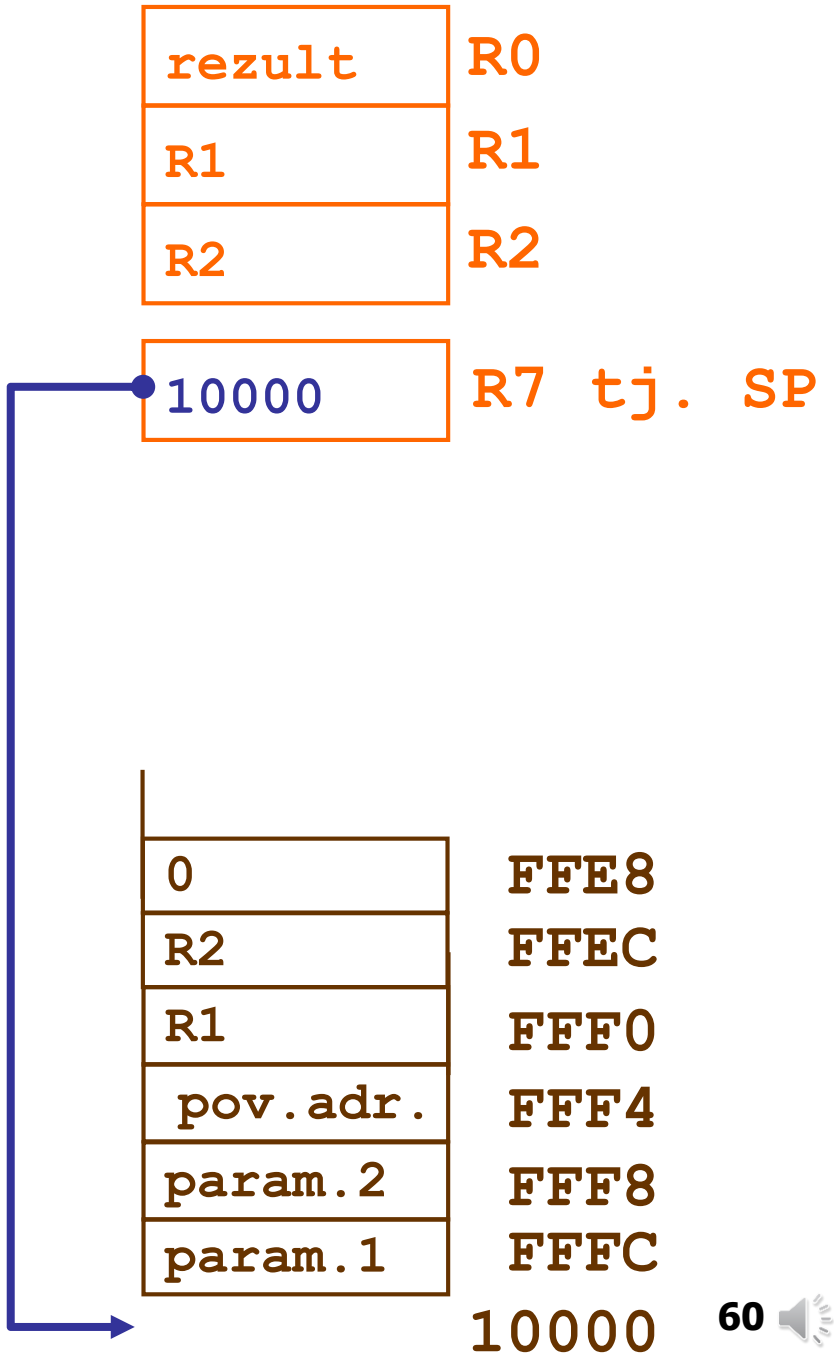


rezultat se sprema izravno iz R0

<<<< Izvođenje programa:

```
GLAVNI MOVE  10000, SP

       LOAD  R0, (PRVI)
       PUSH  R0

       LOAD  R0, (DRUGI)
       PUSH  R0

       CALL  NILI

       STORE R0, (REZ)

       ADD   SP, 8, SP

       HALT
```

glavni program uklanja parametre sa stoga



Komentari:

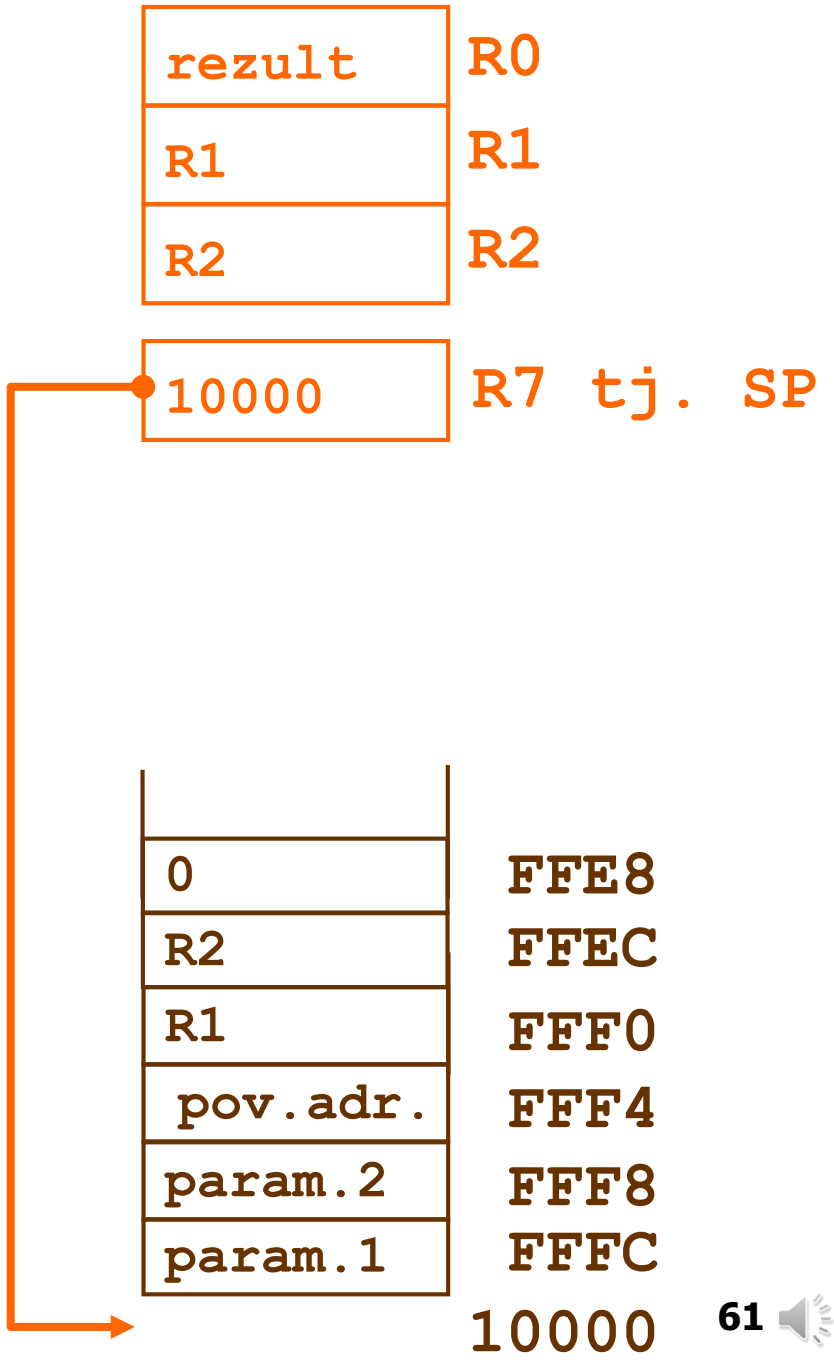
• Za razliku od prethodnih primjera, gdje je potprogram pomoću naredaba POP "potrošio" svoje parametre, ovdje parametre uklanja pozivatelj.

• Ovo je "čišće" rješenje, jer je logičnije da je parametre sa stoga dužan ukloniti onaj tko ih je i stavio na stog.

• Umjesto niza naredaba POP, parametri se uklanjaju naredbom:

```
ADD    SP, 8, SP
```

što je ne samo brže, nego dodatno čuva vrijednosti svih registara.



<<<< (kompletan listing s komentarima)

```
        ; glavni program
GLAVNI  MOVE  10000, SP     ;važno: inicijaliziraj SP !!!
        LOAD  R0, (PRVI)    ; stavi vrijednost
        PUSH  R0            ; prvog parametra na stog
        LOAD  R0, (DRUGI)   ; stavi vrijednost
        PUSH  R0            ; drugog parametra na stog
        CALL  NILI          ; poziv potprograma
        STORE R0, (REZ)     ; spremi rezultat iz R0
        ADD   SP, 8, SP     ; ukloni parametre sa stoga
        HALT

        ; podatci i mjesto za rezultat
PRVI    DW  1234ABCD
DRUGI   DW  22445599
REZ     DW  0
```

>>>>

<<<<

```
        ; potprogram NILI
NILI    PUSH  R1      ; Spremanje
        PUSH  R2      ; registara.

        LOAD  R1, (SP+0C)     ; Čitanje parametara
        LOAD  R2, (SP+10)    ; u registre R1 i R2

        OR    R1, R2, R0    ; Izračunavanje
        XOR   R0, -1, R0    ; rezultata.

        POP R2        ; Obnovi
        POP R1        ; registre.

        RET
```

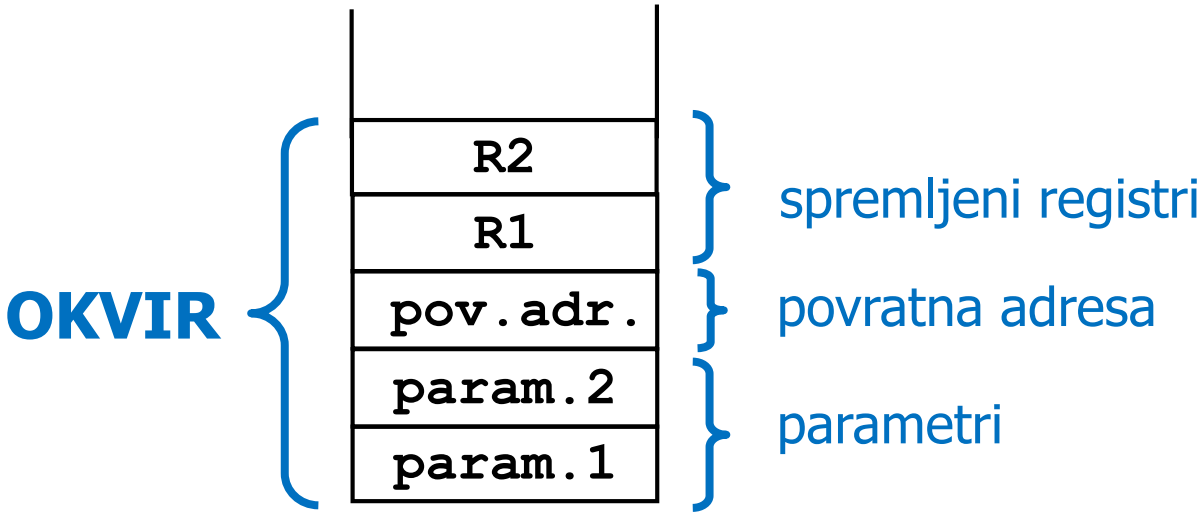
>>>>

Komentari:

• Način rada s parametrima i način vraćanja rezultata pokazan u ovom primjeru vrlo je sličan stvarnim potprogramima dobivenim prevođenjem viših programskih jezika u asembler.

• Razlog za ovakvu organizaciju je njena praktičnost i efikasnost te općenitost koja omogućuje korištenje rekurzivnih i nerekurzivnih potprograma uz čuvanje stanja svih registara.

• Podatci na stogu su uniformno organizirani za svaki potprogram i takav **niz podataka za jedan potprogram** naziva se **okvir stoga** ili kraće okvir (frame, stack frame, activation record). Svaki potprogram ima svoj okvir.



- Do sada nismo vidjeli kako u asembleru ostvariti **lokalne varijable**

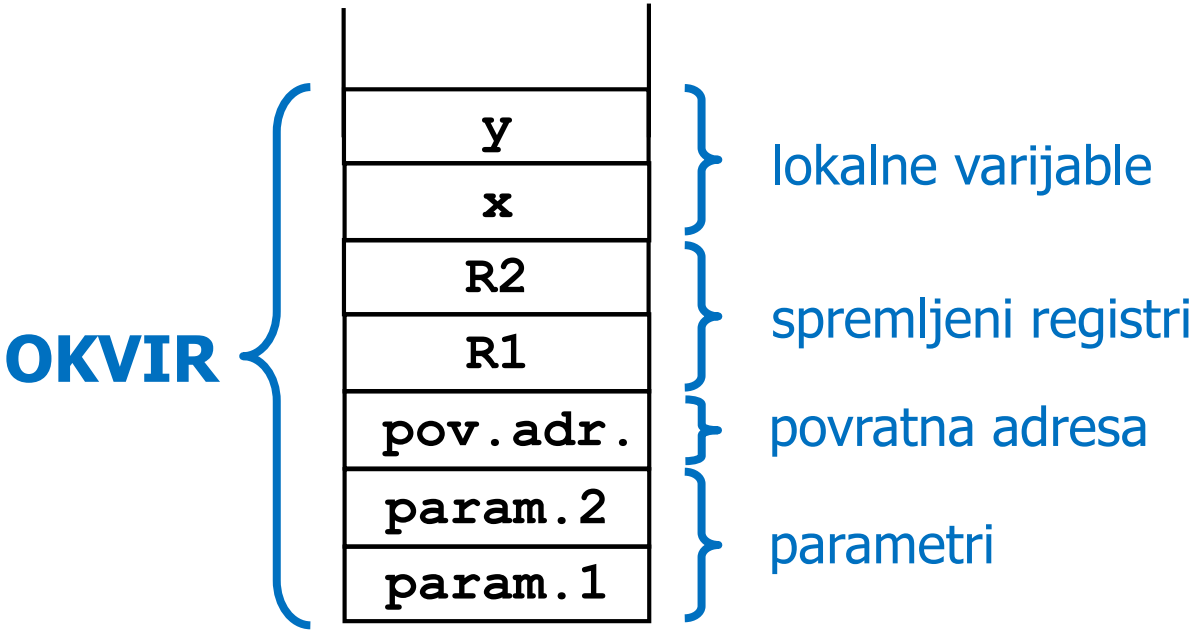
- Lokalne varijable imaju sljedeća svojstva:
  - za svaki poziv potprograma postoje vlastite lokalne varijable
  - vidljive su samo unutar potprograma u kojem su definirane
  - stvaraju se prilikom poziva potprograma
  - nestaju prilikom povratka iz potprograma

- Vidimo da su lokalne varijable po svemu slične parametrima potprograma. Jedina je razlika u početnoj vrijednosti koja se za parametar definira od strane pozivatelja potprograma

- Dakle, prirodno je rješenje da se i lokalne varijable čuvaju na stogu, ili točnije u okviru stoga

<<<<<

- Potpunija verzija stogovnog okvira uključuje i lokalne varijable
- Lokalne varijable se stavljaju na stog nakon ulaska u potprogram, a mogu se staviti "ispod" ili "iznad" spremljenih registara. Ovdje je odabrano da se stave "iznad" njih:

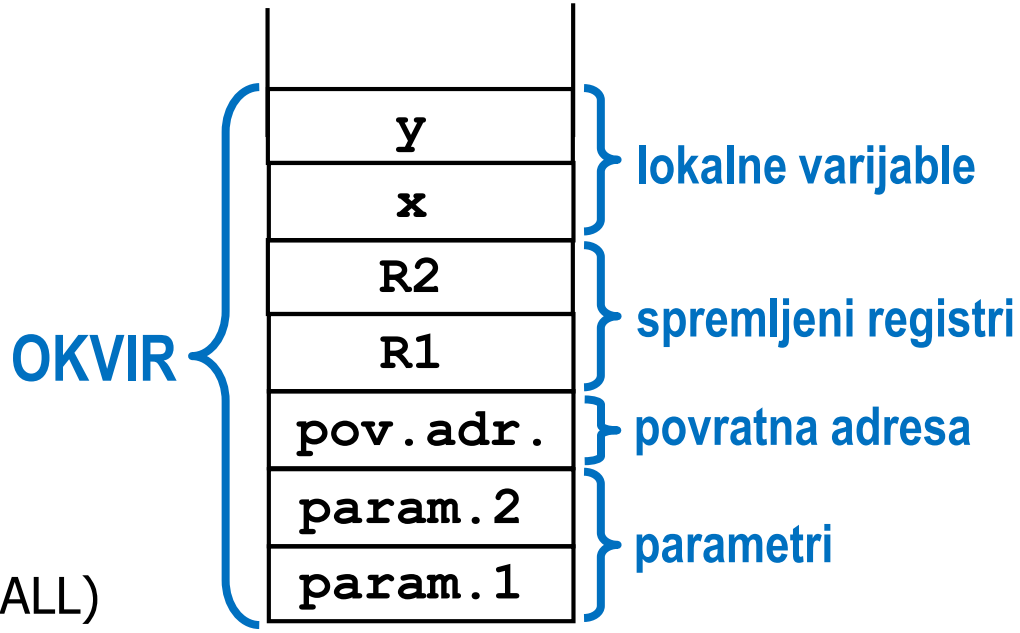


Važna napomena:

Prevoditelj u stvarnosti obično pokušava staviti lokalne varijable u registre. Tek kad se oni napune, stavlja lokalne varijable na stog

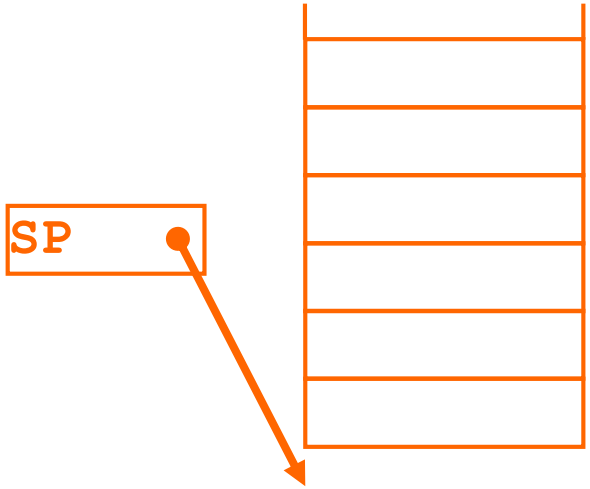
Rekapitulacija okvira stoga

- Okvir stoga sadrži:
  - parametre
  - povratnu adresu
  - spremljene registre
  - lokalne varijable
- Način rada s okvirom:
  - Glavni program:
    - stavlja parametre
    - stavlja povratnu adresu (CALL)
    - uklanja parametre
  - Potprogram:
    - sprema i obnavlja registre
    - stvara i uklanja lokalne varijable
    - uklanja povratnu adresu (RET)



Rekapitulacija rada s okvirom stoga:

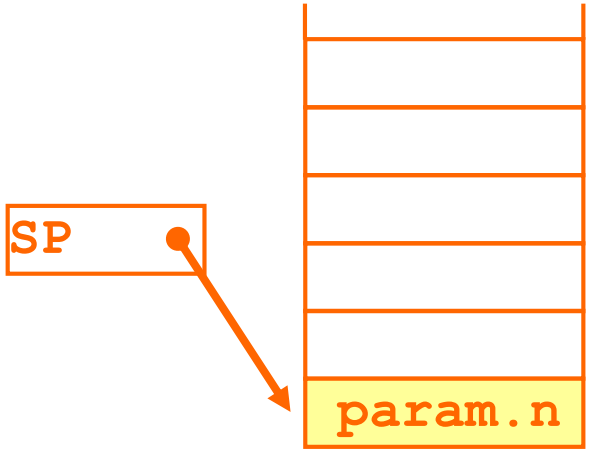
Pozivatelj: Potprogram: Okvir:



Rekapitulacija rada s okvirom stoga:

Pozivatelj: Potprogram: Okvir:

Stavi parametre na stog

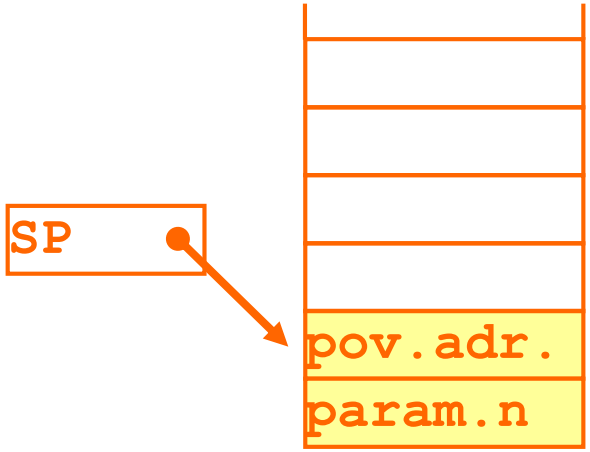


Rekapitulacija rada s okvirom stoga:

Pozivatelj: | Potprogram: | Okvir:

Stavi parametre na stog

Pozovi potprogram



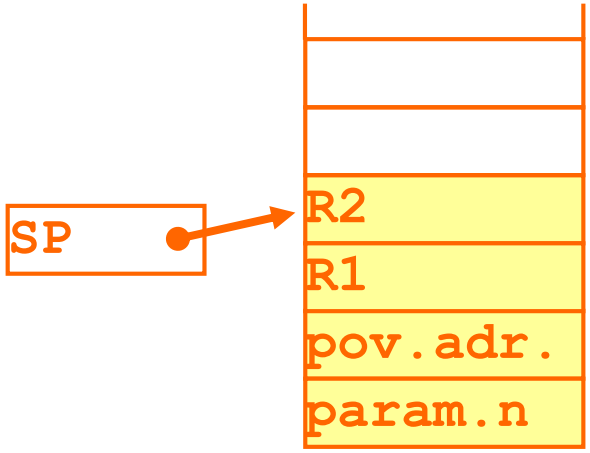
Rekapitulacija rada s okvirom stoga:

Pozivatelj: | Potprogram: | Okvir:

Stavi parametre na stog

Pozovi potprogram

Spremi registre



Rekapitulacija rada s okvirom stoga:

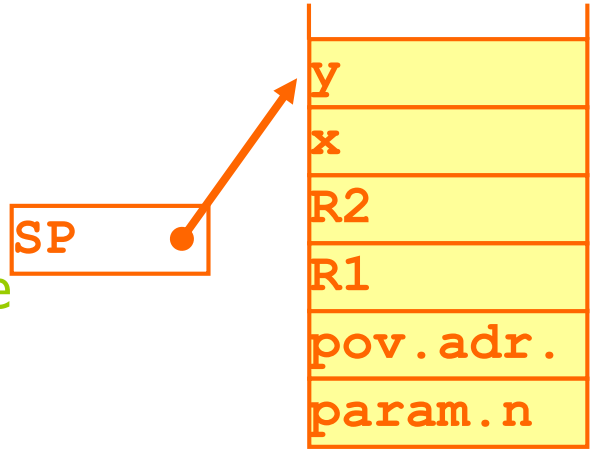
Pozivatelj: | Potprogram: | Okvir:

Stavi parametre na stog

Pozovi potprogram

Spremi registre

Stvori lokalne varijable



Rekapitulacija rada s okvirom stoga:

Pozivatelj: | Potprogram: | Okvir:

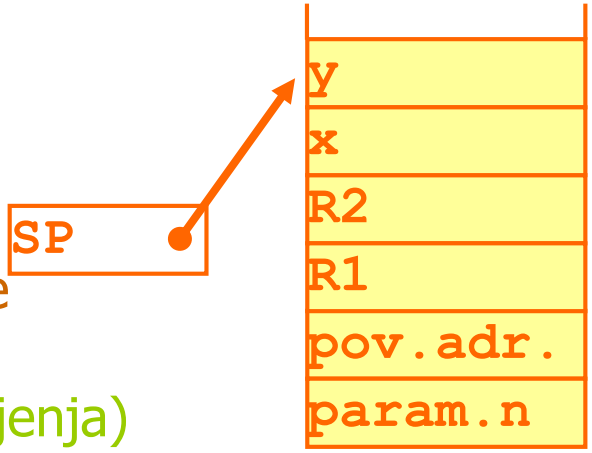
Stavi parametre na stog

Pozovi potprogram

Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)
(ali se mogu mijenjati podatci u okviru,
npr. parametri i lokalne varijable)



Rekapitulacija rada s okvirom stoga:

Pozivatelj: | Potprogram: | Okvir:

Stavi parametre na stog

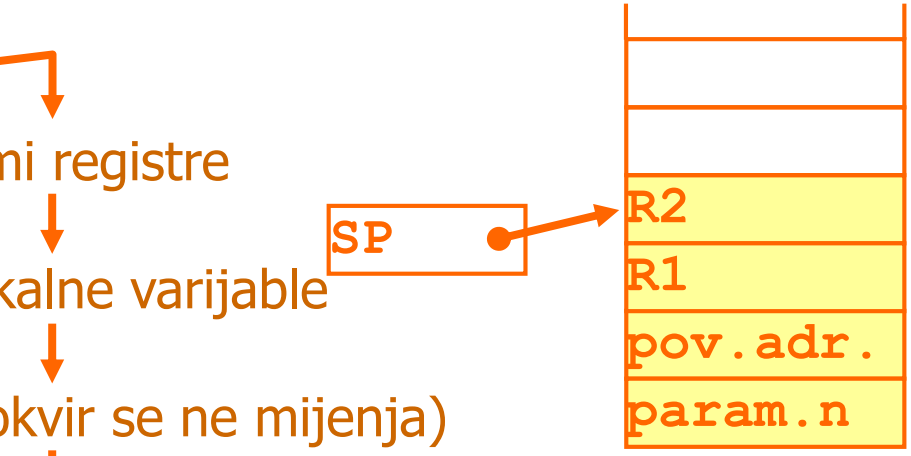
Pozovi potprogram

Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne



Rekapitulacija rada s okvirom stoga:

Pozivatelj: | Potprogram: | Okvir:

Stavi parametre na stog

Pozovi potprogram

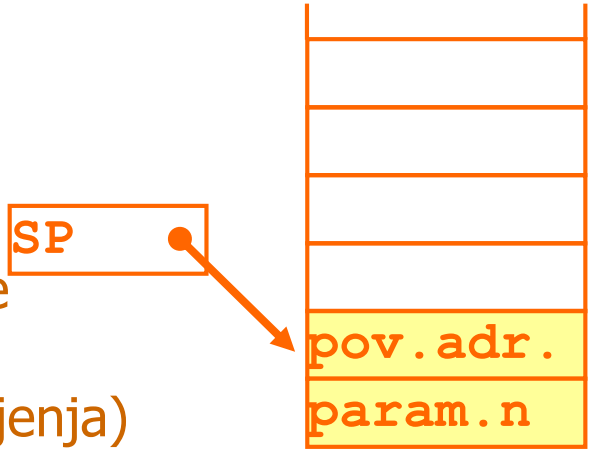
Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)

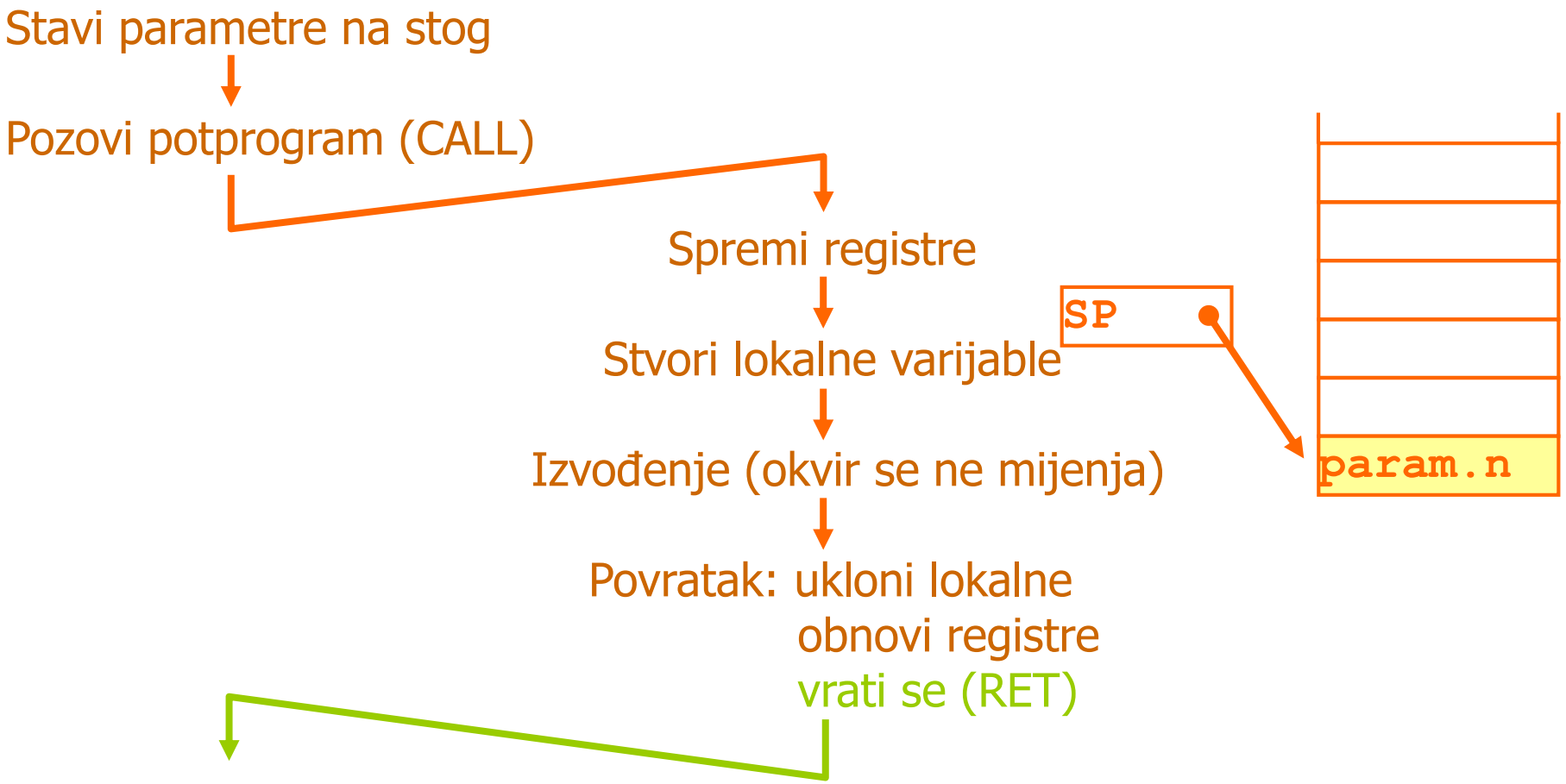
Povratak: ukloni lokalne

obnovi registre



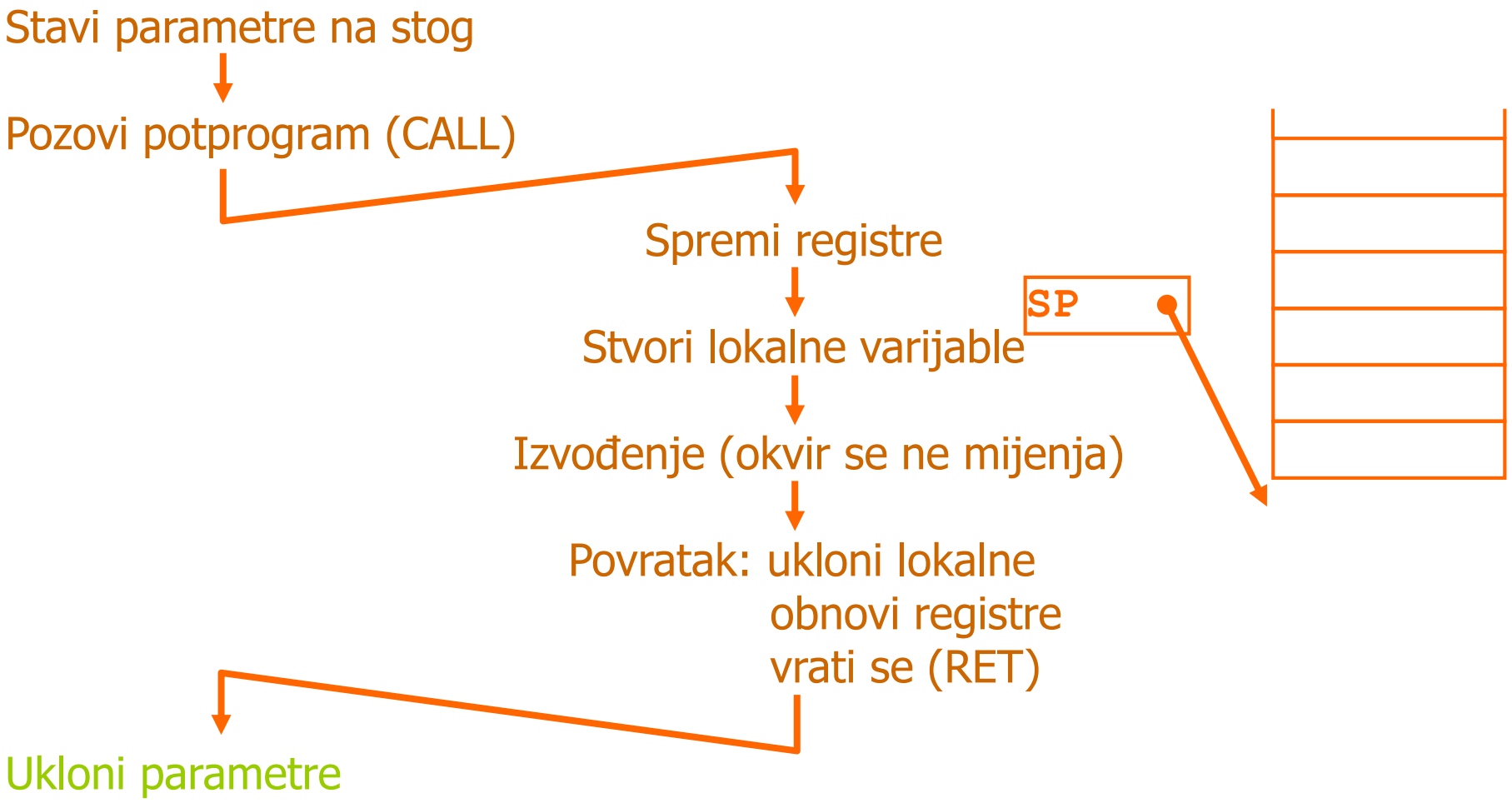
Rekapitulacija rada s okvirom stoga:

Pozivatelj: Potprogram: Okvir:



Rekapitulacija rada s okvirom stoga:

Pozivatelj: | Potprogram: | Okvir:



Rekurzivni potprogrami

Potprogrami - Prijenos stogom

- Rekurzivni potprogrami su oni koji mogu pozvati sami sebe
  - (izravno, ili neizravno - preko drugih potprograma)
- Budući da imamo višestruki poziv istog potprograma, to je isto kao da je rekurzivni potprogram istovremeno "više puta aktiviran"
  - Zato se parametri ne mogu prenositi registrima i fiksnim lokacijama (npr. prvi poziv bi napunio vrijednosti u parametre, a već bi drugi poziv prepisao preko njih svoje vrijednosti, čime bi prve vrijednosti za prvi poziv bile izgubljene)
- Zato rekurzivni potprogrami koriste prijenos stogom i okvir stoga
- Uočite da povratna vrijednost nije problem i za njeno vraćanje se koristi prijenos registrom

Potprogrami - Rekurzija

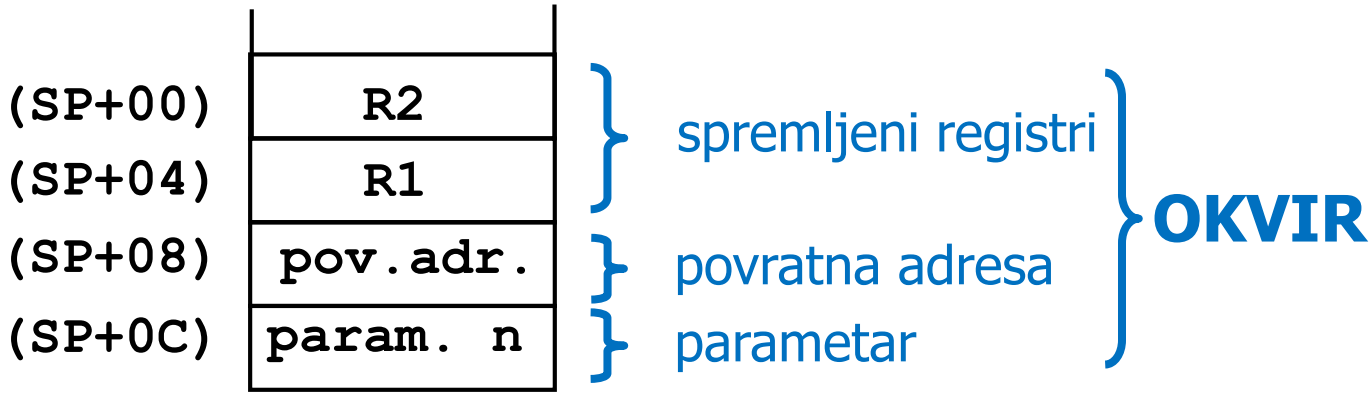
Treba napisati rekurzivni potprogram za računanje faktorijela:

fakt(1) = 1
*fakt(n) = fakt(n-1)*n za n= 2, 3, 4, ...*

Parametar se prenosi stogom, a rezultat se vraća registrom R0.

Rješenje:

Pokažimo prvo kako će izgledati okvir stoga:



```
<<<<      ; potprogram fakt(n)

FAKT      PUSH  R1
          PUSH  R2

          LOAD  R0, (SP+0C)
          CMP   R0, 1
          JR_EQ VRATISE

          SUB   R0, 1, R0
          PUSH  R0
          CALL  FAKT
          ADD   SP, 4, SP

          LOAD  R1, (SP+0C)
          MOVE  0, R2
MNOZI     ADD   R0, R2, R2
          SUB   R1, 1, R1
          JR_NZ MNOZI

          MOVE  R2, R0

VRATISE   POP   R2
          POP   R1
          RET
```

Spremi registre

Ako je n jednak 1, onda vrati 1 u R0

Pozovi fakt(n-1) ovako:
- izračunaj n-1 i stavi ga na stog
- pozovi fakt(n-1), rezultat će biti u R0
- ukloni parametar (tj. n-1) sa stoga

Brojač R1=n; početni umnožak R2=0

Množenje uzastopnim pribrajanjem:
R1 je brojač, umnošku R2 pribraja se R0 u kojem je fakt(n-1)

Umnožak iz R2 treba vratiti preko R0

Obnovi registre i vrati se

<<<<

Pokažimo još samo kako bi mogao izgledati glavni program koji poziva fakt(3) i sprema rezultat na memorijsku lokaciju REZULT.

```
GLAVNI   MOVE 10000, SP

         MOVE 3, R0
         PUSH R0
         CALL FAKT
         STORE R0, (REZULT)
         ADD  SP, 4, SP

         HALT

REZULT   DW  0
```

Potprogrami - Rekurzija

Treba napisati rekurzivni potprogram za računanje niza Fibonaccijevih brojeva:

Fib(1) = 1
Fib(2) = 1
Fib(n) = Fib(n-1) + Fib(n-2) *za n= 3,4, ...*

Parametar se prenosi stogom, a rezultat se vraća registrom R0. Glavni program treba izračunati *Fib(8)*.

Rješenje:

• Radi lakšeg objašnjenja, pokažimo prvo kako bi rješenje moglo izgledati u programskom jeziku C

>>>>

```
int fib ( int n ) {

   int x, y;     // lokalne varijable za međurezultate

   if ( n == 1   ||   n == 2 )   // Fib(1) i Fib(2) = 1
      return ( 1 );

   x = fib ( n-1 );
   y = fib ( n-2 );                // Fib(n) = Fib(n-1) + Fib(n-2)
   return ( x+y );
}

main () {

    int rez;
    rez = fib ( 8 );
}
```


Rješenje u C-u se moglo napisati i nešto kraće:

```
int fib ( int n ) {

   if ( n == 1  ||  n == 2 )
      return ( 1 );

   return ( fib ( n-1 ) + fib ( n-2 ) )
}
```

• Ovdje nema lokalnih varijabli x i y. Međutim, treba voditi računa da rezultat od fib(n-1) mora biti negdje pohranjen dok se izračunava fib(n-2).

• Zato će C-prevodilac stvoriti asemblerski program koji je sličniji verziji s prethodnog slajda, iako je to za programera u C-u nevidljivo.

Mogući prijevod C-programa u asembler:

```
        ; glavni program

MAIN    LOAD  SP, 10000  ; inicijalizacija stoga

        MOVE   8, R0       ; Stavi željenu vrijednost
        PUSH   R0          ; parametra (n=8) na stog.

        CALL   FIB      ; poziv potprograma

        ADD    SP, 4, SP   ; ukloni parametar sa stoga
        STORE  R0, (REZ)   ; spremi rezultat iz R0

        HALT

REZ     DW  0              ; mjesto za rezultat
```

>>>>

```
        ; potprogram FIB

FIB     PUSH   R1     ; Spremanje
        PUSH   R2     ; registara na stog

        ; Stvaranje lokalnih varijabli x i y
        SUB    SP, 8, SP

        ; if( n==1  ||  n==2 ) return (1);

        MOVE   1, R0 ; Pripremi rez. za slučaj povratka

        LOAD   R1, (SP+14) ; dohvati parametar n
        CMP    R1, 1       ; je li n==1 ?
        JR_EQ  VRATI_SE    ; Da: idi na dio za povratak

        CMP    R1, 2       ; Ne: ispitaj je li n==2 ?
        JR_EQ  VRATI_SE    ; Da: idi na dio za povratak
```

>>>>

```
; n nije ni 1 ni 2 - nastavak izvođenja

; x = fib(n-1)

LOAD   R1, (SP+14) ; dohvati parametar n
SUB    R1, 1, R1   ; Oduzmi n-1 i stavi na stog
PUSH   R1          ; za prvi rekurzivni poziv.

CALL   FIB         ; pozovi fib(n-1)
ADD    SP, 4, SP   ; ukloni n-1 sa stoga
STORE  R0, (SP+4)  ; spremi rezultat u x

; y = fib(n-2)

LOAD   R1, (SP+14) ; dohvati parametar n
SUB    R1, 2, R1   ; Oduzmi n-2 i stavi na stog
PUSH   R1          ; za drugi rekurzivni poziv.

CALL   FIB         ; pozovi fib(n-2)
ADD    SP, 4, SP   ; ukloni n-2 sa stoga
STORE  R0, (SP+0)  ; spremi rezultat u y
```

>>>>

```
        ; return ( x + y )

         LOAD   R1, (SP+4)   ; dohvati x
         LOAD   R2, (SP+0)   ; dohvati y
         ADD    R1, R2, R0   ; izračunaj povratnu vrijednost

VRATI_SE ADD    SP, 8, SP    ; ukloni x i y sa stoga

         POP    R2           ; Obnovi sadržaje
         POP    R1           ; spremljenih registara.

         RET                 ; povratak iz FIB
```


- Pokažimo ponašanje okvira na stogu ako je u glavnom programu pozvano Fib(4).

- Nećemo pratiti pojedine podatke u okvirima na stogu već promatramo pojedine **okvire kao cjeline**.

okviri na stogu

main

izvodi se main

okviri na stogu

main

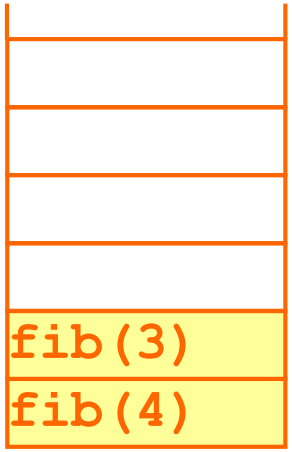
main → fib(4)

main poziva fib(4)

okviri na stogu

main
main → fib(4)
main fib(4) → fib(3)

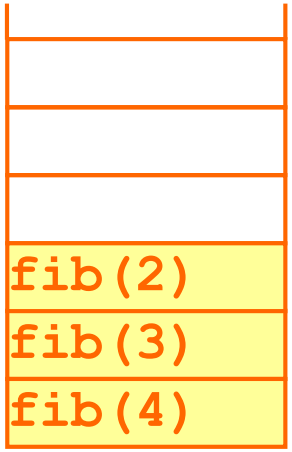
fib(4) poziva fib(3)



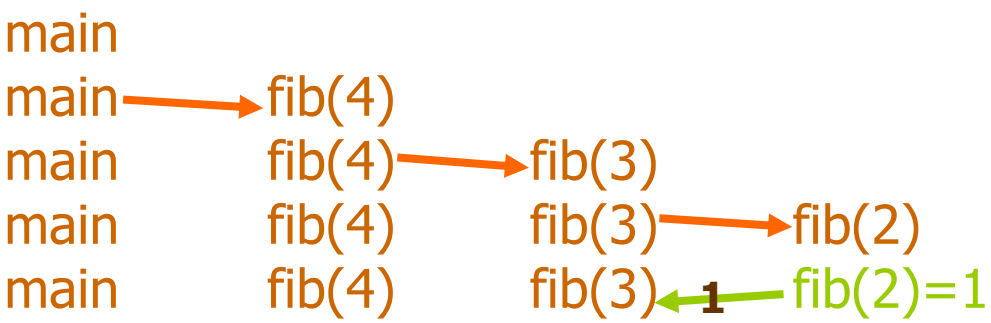
okviri na stogu

main
main → fib(4)
main fib(4) → fib(3)
main fib(4) fib(3) → fib(2)

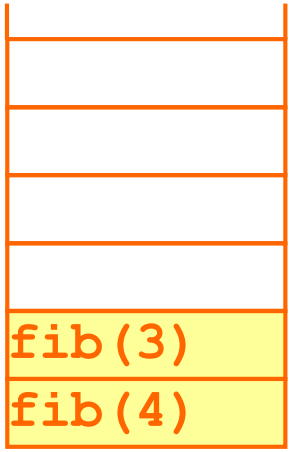
fib(3) poziva fib(2)



okviri na stogu



fib(2) vraća 1



okviri na stogu

main

main → fib(4)

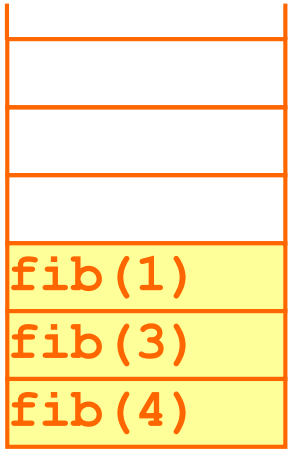
main fib(4) → fib(3)

main fib(4) fib(3) → fib(2)

main fib(4) fib(3) ← 1 fib(2)=1

main fib(4) fib(3) → fib(1)

fib(3) poziva fib(1)



okviri na stogu

main

main → fib(4)

main fib(4) → fib(3)

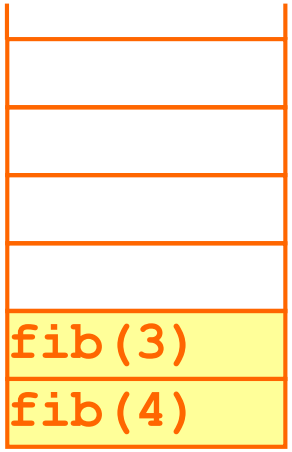
main fib(4) fib(3) → fib(2)

main fib(4) fib(3) ← 1 fib(2)=1

main fib(4) fib(3) → fib(1)

main fib(4) fib(3) ← 1 fib(1)=1

fib(1) vraća 1



okviri na stogu

main

main → fib(4)

main fib(4) → fib(3)

main fib(4) fib(3) → fib(2)

main fib(4) fib(3) ← 1 fib(2)=1

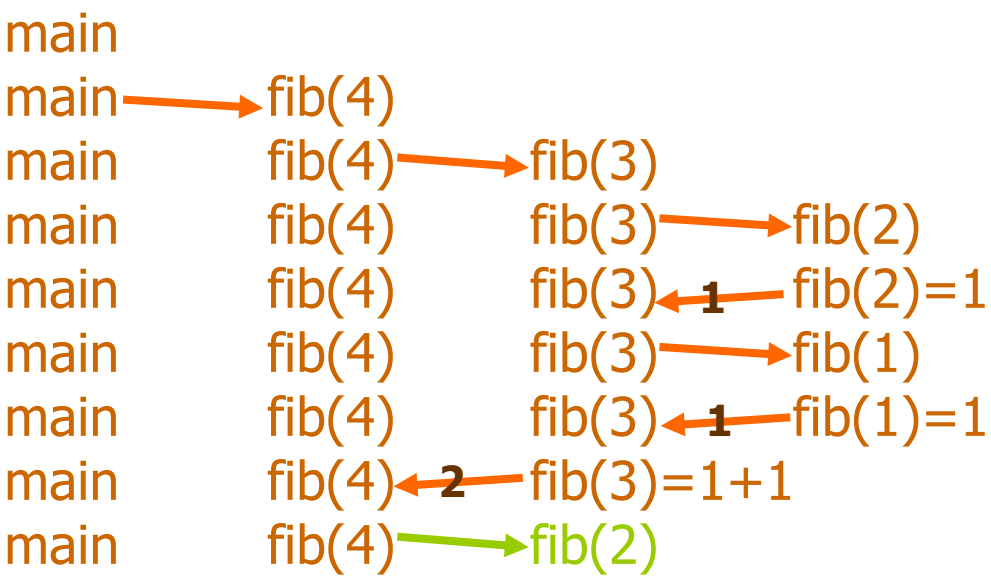
main fib(4) fib(3) → fib(1)

main fib(4) fib(3) ← 1 fib(1)=1

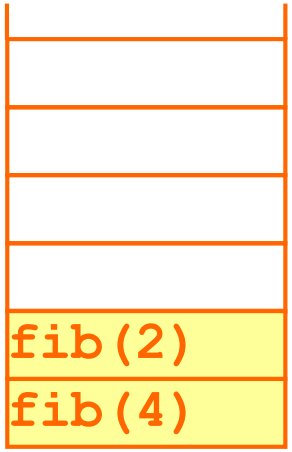
main fib(4) ← 2 fib(3)=1+1

fib(3) vraća fib(2)+fib(1)

okviri na stogu



fib(4) poziva fib(2)



okviri na stogu

main

main → fib(4)

main fib(4) → fib(3)

main fib(4) fib(3) → fib(2)

main fib(4) fib(3) ← 1 fib(2)=1

main fib(4) fib(3) → fib(1)

main fib(4) fib(3) ← 1 fib(1)=1

main fib(4) ← 2 fib(3)=1+1

main fib(4) → fib(2)

main fib(4) ← 1 fib(2)=1

fib(2) vraća 1

okviri na stogu

main

main → fib(4)

main fib(4) → fib(3)

main fib(4) fib(3) → fib(2)

main fib(4) fib(3) ←1– fib(2)=1

main fib(4) fib(3) → fib(1)

main fib(4) fib(3) ←1– fib(1)=1

main fib(4) ←2– fib(3)=1+1

main fib(4) → fib(2)

main fib(4) ←1– fib(2)=1

main ←3– fib(4)=2+1

main

fib(4) vraća fib(3)+fib(2)

okviri na stogu

main

main → fib(4)

main fib(4) → fib(3)

main fib(4) fib(3) → fib(2)

main fib(4) fib(3) ←1 fib(2)=1

main fib(4) fib(3) → fib(1)

main fib(4) fib(3) ←1 fib(1)=1

main fib(4) ←2 fib(3)=1+1

main fib(4) → fib(2)

main fib(4) ←1 fib(2)=1

main ←3 fib(4)=2+1

main

nastavlja se izvoditi main

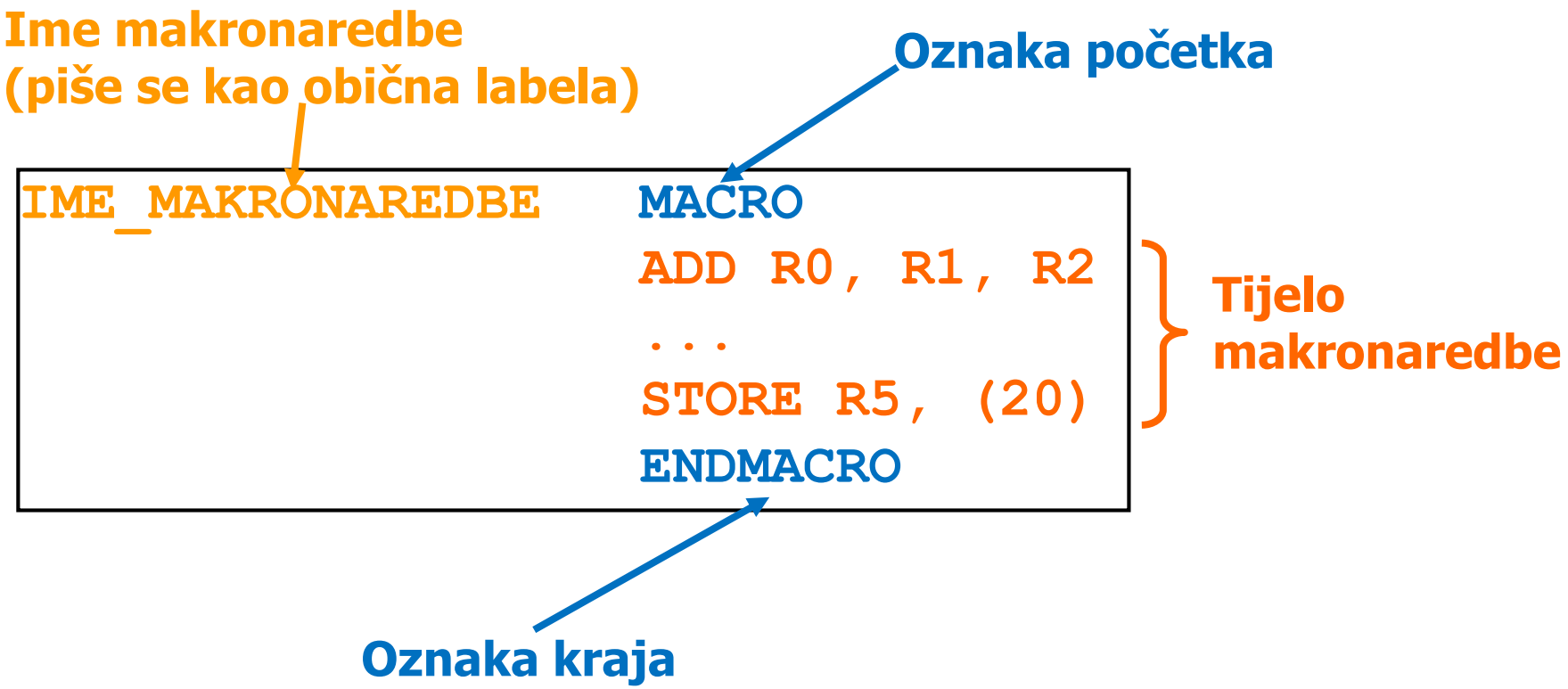
okviri na stogu

Makronaredbe

Makronaredbe

- Makronaredbe imaju istu namjenu kao i potprogrami
- Razlika je u načinu njihove izvedbe
- Potprogrami su podržani od strane procesora, pomoću naredaba CALL i RET kojima se obavljaju poziv i povratak iz potprograma
- Makronaredbe su podržane od strane asemblerskog prevoditelja koji ih prevodi u obične naredbe procesora

Makronaredbe - definicija



Makronaredbe - Primjer

Napisati makronaredbu koja izračunava logičku operaciju NILI između sadržaja registara R0 i R1 i vraća rezultat u registru R2. Napisati i glavni program koji će pozvati makronaredbu za dva podatka iz memorije i rezultat također upisati u memoriju.

Rješenje:

```
          ; DEFINICIJA MAKRONAREDBE

NILI      MACRO
          OR    R0, R1, R2
          XOR   R2, -1, R2
          ENDMACRO
```

>>>>

```
;;;;;;  GLAVNI PROGRAM

        LOAD    R0, (PRVI)   ; DOHVATI PODATKE
        LOAD    R1, (DRUGI)  ; U R0 i R1

        NILI                 ; POZOVI MAKRONAREDBU

        STORE   R2, (REZ)    ; SPREMI REZULTAT
        HALT

;;;;;;  PODATCI I MJESTO ZA REZULTAT
PRVI    DW   81282C34
DRUGI   DW   29A82855
REZ     DW   0
```

Napomena: Poziv se ostvaruje navođenjem imena makronaredbe. Poziv **nije naredba procesora**, već je sličniji pseudonaredbi asemblerskog prevoditelja.

Makronaredbe - Način prevođenja

- Asembleri koji podržavaju makronaredbe moraju biti troprolazni ili četveroprolazni i nazivaju se makroasemblerima
- U troprolaznom asembleru se svaka definicija makronaredbe mora nalaziti **ISPRED** njenog pozivanja
- U četveroprolaznom asembleru se definicija makronaredbe smije nalaziti **IZA** njenog pozivanja
- Objasnimo kako rade ove dvije vrste asemblera . . .

Makronaredbe - Troprolazni asembler

Prvi prolaz:

- Asembler čita redak po redak datoteke:
  - Ako naiđe na definiciju makronaredbe, onda u tablici makronaredbi zapamti njeno ime i tijelo (slično kao što dvoprolazni asembler nailaskom na labelu u tablici labela pamti njeno ime i vrijednost)
  - Ako naiđe na poziv makronaredbe, zamjenjuje ga njenim tijelom koje je zapamćeno u tablici (ovo je uvijek moguće napraviti zbog obaveznog redoslijeda u kojem definicija uvijek prethodi pozivu makronaredbe)
  - Prevoditelj ne mijenja ostale retke mnemoničkog programa
- Na kraju prvog prolaska, mnemonička datoteka više nema ni definicija makronaredaba ni poziva makronaredaba, već sadrži samo obične naredbe procesora i pseudonaredbe

Makronaredbe - Troprolazni asembler

<<<<

Drugi i treći prolaz:

- Budući da je rezultat prvog prolaska obični mnemonički program, drugi i treći prolaz u potpunosti odgovaraju radu običnog dvoprolaznog asemblera:
  - Drugi prolaz: ekvivalentan prvom prolasku dvoprolaznog asemblera
  - Treći prolaz: ekvivalentan drugom prolasku dvoprolaznog asemblera

>>>>

Makronaredbe - Troprolazni asembler

Kako izgleda prevođenje prethodnog primjera?

```
NILI   MACRO
       OR   R0, R1, R2
       XOR  R2, -1, R2
       ENDMACRO

GLAVNI LOAD   R0,(PRVI)
       LOAD   R1,(DRUGI)
       NILI
       STORE  R2,(REZ)
       HALT
```

originalni program s makronaredbama i njihovim pozivima

prvi prolaz

```
GLAVNI LOAD   R0,(PRVI)
       LOAD   R1,(DRUGI)

       OR   R0, R1, R2
       XOR  R2, -1, R2

       STORE  R2,(REZ)
       HALT
```

rezultat prvog prolaza: obični mnemonički program

Makronaredbe - Četveroprolazni asembler

Četveroprolazni asembler dozvoljava da poziv makronaredbe prethodi njenoj definiciji. Zato je potreban dodatni prolaz (kao što je u simboličkom dvoprolaznom asembleru potreban dodatni prolaz zbog labela koje se koriste prije nego što su definirane)

Prvi prolaz:

- Asembler čita redak po redak datoteke:
  - Ako naiđe na definiciju makronaredbe, onda u tablici makronaredbi zapamti njeno ime i tijelo (slično kao što dvoprolazni asembler nailaskom na labelu u tablici labela pamti njeno ime i vrijednost)
  - Prevoditelj ne mijenja ostale retke mnemoničkog programa
- Na kraju prvog prolaska, mnemonička datoteka više nema definicija makronaredaba, ali sadrži pozive makronaredaba.

Makronaredbe - Četveroprolazni asembler

<<<<

Drugi prolaz:

- Drugi prolaz zamijenjuje sve pozive makronaredba njihovim tijelima što daje obični mnemonički program

Treći i četvrti prolaz:

- Treći i četvrti prolaz u potpunosti odgovaraju prvom i drugom prolazu običnog dvoprolaznog asemblera (odnosno odgovaraju radu drugog i trećeg prolaza troprolaznog asemblera)

>>>>

Makronaredbe - Parametri

- Jedna od mogućnosti koju makroasembleri obično nude je i korištenje **parametara makronaredaba**
- Parametri se obično navode kao lista imena iza *MACRO*
- Unutar tijela se parametri koriste na bilo kojim mjestima u naredbama
- Kod poziva makronaredbe programer zadaje argumente, tj. navodi npr. registre, adrese i sl. Ovi argumenti zamjenjuju parametre prilikom proširivanja makronaredbe.
- Ovisno o mjestima korištenja parametara, programeru je pri pozivanju ograničena njihova upotreba

Makronaredbe - Parametri

Primjer:

Napišite makronaredbu koja će izračunati logičku operaciju NILI. Ulazni podatci i rezultat mogu biti smješteni na bilo kojim memorijskim lokacijama koje se zadaju kao parametri makronaredbe.

Napišite glavni program koji će izračunati NILI između podataka s lokacija PRVI i DRUGI, a rezultat će spremiti na REZULT.

Rješenje:

na sljedećem slajdu

>>>>

<<<<

```
        ; DEFINICIJA MAKRONAREDBE
NILI    MACRO P1, P2, REZ
        LOAD  R0, (P1)
        LOAD  R1, (P2)
        OR    R0, R1, R2
        XOR   R2, -1, R2
        STORE R2, (REZ)
        ENDMACRO

        ; GLAVNI PROGRAM
        NILI PRVI, DRUGI, REZULT    ;;;; POZIV
        HALT

        ; PODATCI I MJESTO ZA REZULTAT
PRVI     DW  81282C34
DRUGI    DW  29A82855
REZULT   DW  0
```

>>>>

<<<<

Nakon prvog prolaza prevođenja, glavni program izgleda ovako:

```
        LOAD   R0, (PRVI)
        LOAD   R1, (DRUGI)
        OR     R0, R1, R2
        XOR    R2, -1, R2
        STORE  R2, (REZULT)

        HALT

PRVI    DW   81282C34
DRUGI   DW   29A82855
REZULT  DW   0
```

NILI PRVI,DRUGI,REZULT

Usporedba makronaredaba i potprograma

- Makronaredbe općenito troše više memorije jer njihovo tijelo u memoriji postoji u više primjeraka (onoliko koliko ima poziva). Potprogrami se u memoriji nalaze samo u jednom primjerku.

- Potprogrami su sporiji jer se troši vrijeme na njihovo pozivanje i povratak, a također dio vremena troši i prijenos parametara i povratne vrijednosti. Makronaredbe se ne pozivaju nego se jednostavno izvode na mjestu na kojem je to potrebno.

- Što odabrati? Ovisno o tome što je kritično u pojedinom dijelu programa - brzina ili zauzeće memorije.

Napomena: Mjesto upotrebe parametara u tijelu makronaredbe ograničava argumente koje možemo slati prilikom poziva. U ovom primjeru kao argumente možemo navoditi samo adrese zadane apsolutnim ili simboličkim adresiranjem.

Dodatno se smije (iako to nije bila namjera pri pisanju makronaredbe) kao argument poslati i registar opće namjene, jer se u naredbama STORE i LOAD u zagradama smije koristiti i indirektno registarsko adresiranje s odmakom (odmak će u ovom slučaju biti 0)

```
NILI    MACRO P1, P2, REZ
        LOAD  R0, (P1)
        LOAD  R1, (P2)
        OR    R0, R1, R2
        XOR   R2, -1, R2
        STORE R2, (REZ)
        ENDMACRO
```

>>>>

<<<<

Napomena: Uočite da ovako definirana makronaredba mijenja sadržaje registara R0, R1 i R2 što nije poželjno (kao što nije bilo ni kod potprograma).

Registri se slično potprogramima mogu spremati na stog (ali i na fiksne lokacije jer se makronaredbe ne mogu pozivati rekurzivno*)

```
NILI    MACRO P1, P2, REZ
        LOAD  R0, (P1)      <-- mijenja R0
        LOAD  R1, (P2)      <-- mijenja R1
        OR    R0, R1, R2    <-- mijenja R2
        XOR   R2, -1, R2
        STORE R2, (REZ)
        ENDMACRO                          >>>>
```

---

* Neki makroasembleri dozvoljavaju rekurzivno i uvjetno pozivanje makronaredaba, ali to prelazi opseg gradiva na ovom predmetu

Budući da se ovoj makronaredbi smiju poslati registri kao argumenti, što bi se dogodilo da se pozove ovako:

```
NILI PRVI, DRUGI, R0
```

Zašto makronaredba ne bi radila ispravno?

```
NILI    MACRO P1, P2, REZ
        LOAD  R0, (P1)
        LOAD  R1, (P2)
        OR    R0, R1, R2
        XOR   R2, -1, R2
        STORE R2, (REZ)
        ENDMACRO
```