

Funkcije

- Namjena funkcija* u asemblerskom programiranju ista je kao i u višim programskim jezicima (modularnost, ponovna upotreba koda, lakše programiranje i održavanje programa itd.)
- U višim programskim jezicima mnogi implementacijski detalji vezani uz funkcije su za programera skriveni
- U asembleru moramo detaljno poznavati sve mehanizme koji se koriste za funkcije

* Terminologija nije posve ujednačena. Mi ćemo uglavnom koristiti i termin funkcija i potprogram (rjeđe). Često se (prema jeziku Pascal) kaže da funkcije vraćaju vrijednost, a procedure ne (jedno i drugo su potprogrami ili rutine).

Funkcije u višem jeziku

Pozivatelj funkcije:

```
main () {  
    int i, j;  
    ...  
    i = f1(j, 8);  
    ...  
}
```

poziv

argumenti ili
stvarni parametri

Funkcija:

ime funkcije

parametri ili
formalni parametri

tijelo

```
int f1 ( int a, int b ) {  
    int v;  
    v = a+b;  
    ...  
    return v/(a-b);  
}
```

lokalna
varijabla

povratak

povratna
vrijednost

- Svi **označeni** elementi s prethodne slike postojat će i u asemblerskom programu:
 - neki (plavo označene) su podržani samim asemblerskim jezikom (**poziv i povratak, ime funkcije**)
 - neke (**zeleno označene**) moramo „ručno isprogramirati” (**prijenos argumenata i rezultata, stvaranje lokalnih varijabli**)
- Neki dijelovi slike **ne postoje**:
 - vitičaste zagrade, jer assembler nije strukturirani jezik
 - tipovi povratne vrijednosti i parametara te tipovi varijabli, jer assembler nema tipove
- I obrnuto, neke stvari ne postoje u višem jeziku (tj. nisu vidljive),
ali ih u assembleru moramo **eksplicitno napisati**
 - čuvanje konteksta

Pozivatelj funkcije:

```
MAIN ...  
    ... ; pripremi argumente  
    BL F1 ; poziv  
    ... ; dohvati rezultat  
    ...
```

Funkcija:

ime funkcije

tijelo

```
F1 ... ; spremi kontekst  
    ... ; stvori lokalne varijable  
    ... ; dohvati argumente  
  
    ... ; korisne naredbe  
  
    ... ; pripremi rezultat  
    ... ; ukloni lokalne varijable  
    ... ; obnovi kontekst  
    MOV PC, LR ;povratak izfunkcije
```

Funkcija - poziv i povratak

- Funkcija se poziva* naredbom **BL (Branch and Link)**:
BL IME_FUNKCIJE
- Slična naredbi B, ali dodatno sprema sadržaj **PC-4** u Link Register - LR (R14). To je spremanje **povratne adrese**.
 - Uočite da procesor sprema PC – 4, razlog za ovo je protočna struktura tako da za sad zapamtite a objasniti ćemo kasnije
- Odredište skoka zadaje se labelom, pa je ime funkcije zapravo obična **labela**
 - Asembler zamjenjuje tu labelu nekim od dozvoljenih načina adresiranja (npr PC + odmak)
- Povratak se ostvaruje* tako da se pohranjena **povratna adresa vrati iz LR u PC** naredbom:
MOV PC, LR

* Napomena: mnogi procesori imaju naredbe s intuitivnijim imenima CALL i RET

Parametri i povratna vrijednost

- **Parametri** se mogu prenositi:
 - preko **registara**
 - preko **fiksnihi memorijskih lokacija**
 - preko **stoga**
 - preko **lokacija iza naredbe BL**
- **Povratna vrijednost** se može vraćati na **jednake načine** kao i **parametri**, ali preko **stoga** je **nepraktično**
- Za svaku funkciju **mora se odabrati** točan način prijenosa parametara i vraćanja rezultata

Parametri i povratna vrijednost

- Za pojedinu funkciju mora se koristiti **isključivo način prijenosa odabran** za tu funkciju:
 - 1) Pozivatelj mora **upisati argumente** na zadana mjesta
 - 2) Pozivatelj poziva funkciju (BL)
 - 3) Funkcija **čita argumente** sa zadanih mjesta
 - 4) Funkcija izvodi naredbe u tijelu
 - 5) Funkcija **upisuje rezultat** na točno zadana mjesta
 - 6) Funkcija izvodi povratak u pozivatelja (MOV PC,LR)
 - 7) Pozivatelj nakon povratka **čita rezultat** sa zadanog mjesta
- **VAŽNO** - zbog nezavisnosti svih funkcija u programu:
 - Pozivatelj može pretpostaviti da funkcija **ne mijenja ništa** osim lokacija/registara preko kojih se vraćaju **rezultati**

Prijenos registrima

Prijenos registrima

- Za funkciju se točno definira
 - koje registre će se koristiti za prijenos parametara
 - koje registre će se koristiti za vraćanje rezultata

Prijenos registrima - primjer

Funkcija treba računati $R0 = R0^2 - R1^2$. Funkcija prima parametre preko registara R0 i R1 i vraća rezultat registrom R0.

Glavni program treba izračunati $35^2 - 12^2$ i spremiti rezultat u lokaciju REZ.

Klasično C-ovsko rješenje s parametrima i povratnom vrijednošću – nije isto kao zadani zadatak:

```
main () {  
    int rez;  
  
    rez = razl_kvad( 35, 12 );  
}  
  
int razl_kvad(int a, int b) {  
    return a*a - b*b;  
}
```

U C-u nemamo pristup registrima. Program „ekvivalentniji” assembleru koristio bi **globalne varijable**:

```
int r0, r1, rez;  
  
main() {  
    r0 = 35;  
    r1 = 12;  
    razl_kvad();  
    rez = r0;  
}  
  
void razl_kvad() {  
    r0 = r0*r0 - r1*r1;  
    return;  
}
```

Prijenos registrima - primjer

```
MAIN  MOV  R0, #35      ; pripremi argumente u R0 i R1
      MOV  R1, #12
      BL   RAZ_KVAD     ; poziv funkcije
      STR  R0, REZ       ; dohvat i upotreba rezultata
      SWI  0x123456
```

```
REZ   DW   0           ; mjesto za rezultat
```

```
RAZ_KVAD
      MUL  R2, R0, R0    ; izravna upotreba parametara iz R0 i R1
      MUL  R0, R1, R1
      SUB  R0, R2, R0    ; izravno spremanje rezultata u R0
      MOV  PC, LR       ; povratak u pozivatelja
```

Problem!!!

```
RAZL_KVAD  MUL  R2, R0, R0    ; izravna upotreba parametara iz R0 i R1
            MUL  R0, R1, R1
            SUB  R0, R2, R0    ; spremanje rezultata u R0
            MOV  PC, LR        ; povratak u pozivatelja
```

- Uočite da ova funkcija **mijenja sadržaj registra R2**.
 - Uočite da se u prvoj naredbi mijenja i R0, ali to nije problem jer se preko njega ionako vraća rezultat pa se očekuje da će se R0 sigurno promijeniti.
- Promjena sadržaja registra R2 je **vrlo loša** jer:
 - prekršeno je pravilo da funkcija ne smije mijenjati ništa osim rezultata
- Rješenje ovog problema je **spremanje konteksta**.

- **Kontekstom** se nazivaju oni registri koje **funkcija mijenja, ne računajući** registre za koje se **zna da će ih promijeniti, a to su:**
 - registri preko kojih se vraća **rezultat**
 - registar **R15** koji se ionako stalno mijenja
 - registar **CPSR** jer mnoge naredbe na njega utječu pa se očekuje da će biti promijenjen
- Prema ovoj definiciji, **dio konteksta su i registri kojima se prenose parametri**
- U kontekst bi spadale i **memorijske lokacije** koje bi funkcija koristila kao globalne varijable, ali takve lokacije ionako **nećemo** koristiti u funkciji jer takvu programersku praksu valja **izbjegavati**.
- **Kontekst ćemo spremati na stog** jer je tako najpraktičnije
- **Kontekst funkcije obavezno moramo uvijek spremati!!!**

VAŽNO



Kontekst - primjer

Prethodni primjer riješen sa **spremanjem konteksta**

```
MAIN  MOV    SP, #0x10000

      MOV    R0, #35
      MOV    R1, #12
      BL     RAZL_KVAD

      ; tu je R2 i dalje 0
      STR    R0, REZ
      SWI    0x123456

REZ    DW     0
```

```
RAZL_KVAD

      STMFD  SP!, {R2} ; pohrani

      MUL    R2, R0, R0
      MUL    R0, R1, R1
      SUB    R0, R2, R0

      LDMFD  SP!, {R2} ; obnovi
      MOV    PC, LR
```

Prijenos parametara registrima

- Vrlo **jednostavan** i vrlo **brz**
- Ograničenje je **broj slobodnih registara** koji nam stoji na raspolaganju
- **Nepraktičan** za **ugniježđene pozive**,
 - registri bi se vrlo brzo „potrošili” u svega nekoliko poziva
 - ako se koriste isti registri za parametre i u ugniježđenom pozivu funkcije, onda ih treba pohranjivati (nepraktično)
- **Onemogućuje** korištenje **rekurzivnih funkcija**
 - noviji rekurzivni poziv bi uništio parametre iz prethodnog poziva *
- Praktičan za „vršne funkcije”

* Može se izbjeći tako da se parametri u funkciji spremaju na stog, ali to onda ispadne kao da smo ih preneli preko stoga, ali s malom „odgodom” (+ takav rad se registrima je pomalo nepraktičan)

- **Najbrži** i **najpraktičniji** način vraćanja rezultata
- Najviše se koristi u praksi
- Viši jezici obično vraćaju samo jedan rezultat pa se koristi samo jedan registar. Kompajleri obično po konvenciji rezerviraju jedan registar za vraćanje rezultata (obično R0, pa ćemo tako i mi najčešće raditi)
- Ne sprječava rekurziju i ne radi probleme kod ugniježđenih poziva

Prijenos fiksnim lokacijama

- Sve je analogno kao za registre, jedino se umjesto registara koriste **fiksne lokacije u memoriji**
- Pod „fiksnim lokacijama” misli se na obične **memorijske lokacije kojima se pristupa pomoću labela** i koje se koriste samo za ovu svrhu
 - Efektivno su to globalne varijable, ali im se pristupa „kontrolirano”
 - Svi pozivatelji neke funkcije koriste iste fiksne lokacije
- U jednoj funkciji mogu se miješati različiti načini prijenosa parametara i vraćanja rezultata, ali to mora biti točno tako zadano
 - Na primjer: dva parametra mogu se prenijeti registrima, a tri se mogu prenositi fiksnim lokacijama itd.

Prijenos fiksnim lokacijama - primjer

Napisati funkciju `strchr(st,ch)` koji u stringu `st` (pokazivač na početak stringa) traži prvo pojavljivanje ASCII-znaka `ch`. Funkcija vraća pokazivač na pronađeni znak, ili `NULL` ako znak nije pronađen.

Parametre treba prenijeti fiksnim lokacijama `ST` i `CH`, a rezultat treba vratiti fiksnom lokacijom `REZ`.

Glavni program treba u nekom stringu zamijeniti prvo slovo 'a' slovom 'b', a ako slovo 'a' nije pronađeno, onda ne treba napraviti ništa.

```
main () {  
    char[] string1 = "fgafasdf";  
    char *rez;  
    rez = strchr(string1, 'a');  
    if( rez != NULL )  
        *rez = 'b';  
}
```

```
char* strchr (char* st, char ch) {  
    while( *st != '\0' ) {  
        if( *st == ch )  
            return st;  
        ++st;  
    }  
    return NULL;  
}
```



```
MAIN  MOV    SP, #0x10000    ; inicijalizacija stoga
      MOV    R0, #STRING1    ; pripremi argumente
      STR    R0, ST
      MOV    R0, #0x61        ; ASCII-kôd slova 'a'
      STR    R0, CH
      BL     STRCHR            ; poziv funkcije
      LDR    R0, REZ           ; dohvat i upotreba rezultata
      CMP    R0, #0           ; je li vraćen NULL pokazivač?
      BEQ    KRAJ             ; ako 'a' nije pronađen => gotovo

A_U_B MOV    R1, #0x62        ; ascii-kod slova 'b'
      STRB   R1, [R0]         ; zamijeni 'a' sa 'b'
KRAJ   SWI    0x123456
```

STRING1 DSTR "fgafasdf" ; upis stringa u memoriju

* DSTR je pseudonaredba slična kao DB, ali njome se definira string. String će biti upisan u memoriju, svaki ASCII-znak u jedan bajt, i bit će automatski terminiran znakom \0



```
ST      DW      0          ; prvi parametar - pokazivač na string
CH      DB      0          ; drugi parametar - znak koji tražimo
REZ     DW      0          ; rezultat - pokazivač na pronađeni znak

STRCHR  STMFD    SP!, {R0-R2} ; spremi kontekst
        LDR      R0, ST      ; dohvati parametara iz fiksnih lokacija ST i CH
        LDRB     R1, CH

LOOP    LDRB     R2, [R0]    ; čitaj znak po znak iz stringa

        CMP      R2, #0      ; provjeri kraj stringa '\0'
NEMA    MOVEQ    R0, #0      ; došli smo do kraja - znak nije pronađen
        BEQ      VAN         ; zapamti NULL kao rezultat i prekini petlju

        CMP      R2, R1      ; usporedi trenutni znak sa zadanim
ISTI    BEQ      VAN         ; isti su - R0 je adresa pronađenog znaka

DALJE   ADD      R0, R0, #1   ; nisu isti - traži dalje
        B        LOOP

VAN     STR      R0, REZ     ; spremi rezultat u fiksnu lokaciju REZ

        LDMFD    SP!, {R0-R2} ; obnova konteksta i povratak
        MOV      PC, LR
```

- Praktički **neograničen broj** parametara i rezultata
- Prednost je da se čuvaju registri ako u njima korisnih međurezultata
- **Sporiji i dulji način** jer traži dodatne naredbe LDR/STR
- Fiksne lokacije moraju biti „blizu” funkciji, ali i svim pozivateljima što nije izvedivo za velike programe
- **Onemogućuje** korištenje **rekurzivnih funkcija**
 - noviji rekurzivni poziv uništio bi parametre iz prethodnog poziva
- Prijenos registrima i fiksnim lokacijama efektivno je prienos „globalnim varijablama” jer su i registri i lokacije dostupni svim funkcijama. Odgovornost je programera da registre i fiksne lokacije koristi na „konzistentan” način.

Prijenos stogom

- Sve je analogno kao za registre/fiksne lokacije, jedino se koristi **stog**
- Korišćenje **stoga** omogućuje jednostavne **rekurzivne pozive**
- **Pozivatelj** prvo stavi na stog parametre, a zatim **pozove** funkciju
- **Funkcija čita** parametre sa stoga, ali ih **ne uklanja**
- Nakon povratka iz funkcije, **pozivatelj** je dužan **ukloniti** parametre sa stoga
- Ako se i rezultat vraća stogom, onda on može biti ili iznad ili ispod parametara, ali takav način vraćanja rezultata nije praktičan pa se **u praksi koristi vraćanje registrom**

Uobičajena organizacija memorije – ili Kako MAKSIMALNO iskoristiti memoriju ?

- Kod pokretanja nekog programa sustav ne može znati koliko dinamičke memorije će program alocirati i/ili koliko će podataka biti spremljeno na stog !!
 - Ne mogu se odrediti fiksna područja memorije za gomilu i stog

0000 0000

Efikasno rješenje:

- Na najnižim adresama se nalazi program i varijable/konstante.
- Iza toga se nalazi područje gomile (heap). Gomila dinamički raste prema najvišim adresama. (nema veze sa strukturom podataka koja se također naziva heap - iz "Algoritama i struktura podataka").
- Na najvišim adresama se nalazi stog koji raste prema nižim adresama.

PODSJETNIK: Upravo zbog ovoga kod većine procesora se koriste naredbe za spremanje na stog koje smanjuju SP !!

FFFF FFFF

**Program i
varijable/konst.
(fiksna veličina)**

**Gomila (heap):
malloc i free**



**Stog (stack):
push i pop**

Prijenos stogom - primjer

Napisati funkciju `strcmp(st1,st2)` koja leksički uspoređuje dva stringa (`st1` i `st2` su pokazivači na početke stringova). Funkcija vraća 0 ako su stringovi isti, <0 ako je prvi leksički manji, ili >0 ako je drugi leksički manji.

Parametre treba prenijeti stogom (`st1` na nižoj adresi, a `st2` na višoj), a rezultat treba vratiti registrom `R0`.

```
int strcmp (char* st1, char* st2) {  
    while(*st1 && (*st1 == *st2)) {  
        ++st1; ++st2;  
    }  
    return *st1-*st2;  
}
```

relacija leksički
manji ili veći:

"a" < "b"

"ab" < "abc"

"abc" < "abx"

"ab" = "ab"

"b" > "a"

"abc" > "ab"

Prijenos stogom - primjer

```
MAIN  MOV    SP, #0x10000    ; inicijalizacija stoga
      MOV    R1, #ST1        ; pripremi argumente
      MOV    R2, #ST2
      STMFD  SP!, {R1, R2}   ; stavi argumente na stog
                                   ; (R1 je na nižoj, a
                                   ; R2 je na višoj adresi)

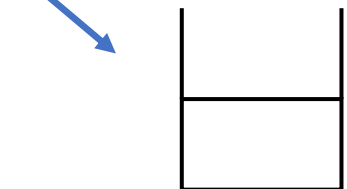
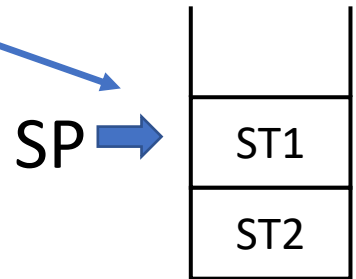
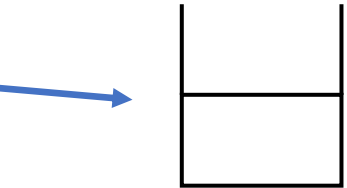
      BL     STRCMP          ; poziv funkcije
      ADD    SP, SP, #8      ; čišćenje stoga

      CMP    R0, #0          ; korištenje
      BEQ    ISTI_SU         ; rezultata
```

RAZLICITI ...

ISTI_SU ...

```
ST1   DSTR  "abcasdf"        ; leksički je manji ST1 jer
ST2   DSTR  "abcqwert"       ; na indeksu 3 ima 'a'
                                   ; dok ST2 ima 'q'
```



Prijenos stogom - primjer

```
STRCMP STMFD SP!, {R1, R5, R6} ; spremi kontekst

      LDR    R5, [SP, #12] ; parametar ST1 u R5
      LDR    R6, [SP, #16] ; parametar ST2 u R6

LOOP  LDRB   R0, [R5]      ; dohvat oba znaka u
      LDRB   R1, [R6]      ; R0 i R1

      CMP    R0, #0        ; provjera kraja stringa ST1
      BEQ    VAN

      CMP    R0, R1        ; usporedba znakova
      BNE    VAN

ISTI  ADD    R5, R5, #1 ; pomak pokazivača ST1...
      ADD    R6, R6, #1 ; i ST2 na sljedeće znakove
      B      LOOP

VAN   SUB     R0, R0, R1 ; leksička razlika => rezult.

      LDMFD  SP!, {R1, R5, R6} ; obnovi kontekst
      MOV    PC, LR
```

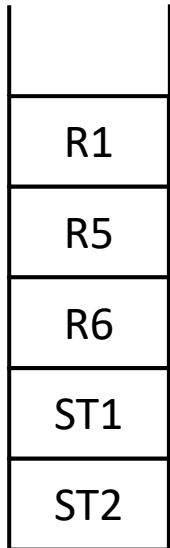
SP →

SP+4

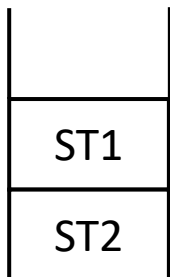
SP+8

SP+12

SP+16



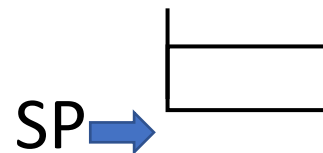
SP →



Vraćanje rezultata stogom - primjer

Kao prošli zadatak, ali i rezultat treba vratiti stogom (verzija s rezultatom ispod parametara).

```
MAIN MOV SP, #0x10000
```



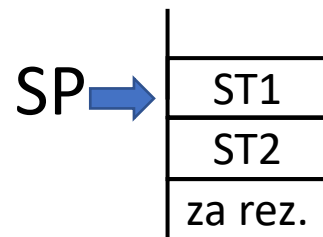
```
SUB SP, SP, #4 ; mjesto za rezultat
```



```
MOV R1, #ST1
```

```
MOV R2, #ST2
```

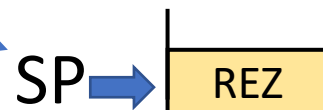
```
STMFD SP!, {R1, R2}
```



```
BL STRCMP
```

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

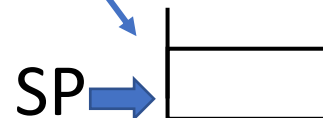
```
LDMFD SP!, {R0} ; dohvat rezultata
```



```
CMP R0, #0
```

```
BEQ ISTI_SU
```

```
...
```



Vraćanje rezultata stogom - primjer

STCMP STMFD SP!, {R1, R5, R6} ; spremi kontekst

LDR R5, [SP, #12] ; parametar ST1 u R5

LDR R6, [SP, #16] ; parametar ST2 u R6

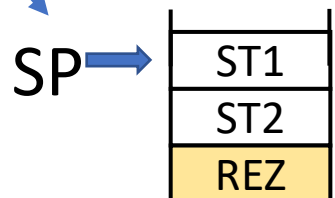
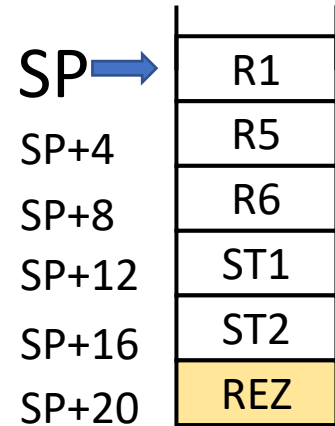
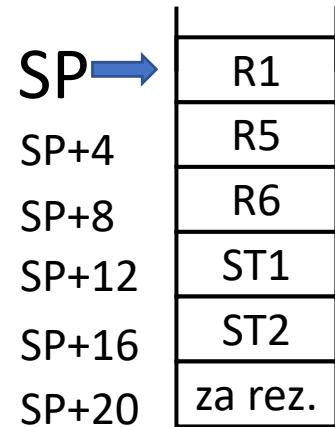
... usporedba stringova ...

VAN SUB R0, R0, R1 ; leksička razlika

STR R0, [SP, #20] ; spremi rezultat

LDMFD SP!, {R1, R5, R6} ; obnovi kontekst

MOV PC, LR



- **Sporiji i dulji način** jer traži dodatne naredbe LDM/STM
 - ali ipak **brže od fiksnih lokacija** jer tamo treba izvoditi nekoliko naredaba LDR/STR, a ovdje je to „sažeto” u jednu naredbu LDM ili STM
- Praktički **neograničeni** broj parametara i rezultata
- Jedini praktičan način za prijenos parametara u **rekurzivne funkcije**
- Vraćanje rezultata nije toliko praktično, pa se u praksi za tu namjenu koriste registri

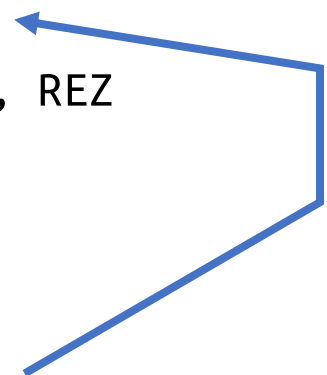
Prijenos lokacijama iza naredbe BL

Prijenos lokacijama iza BL

- Neposredno iza naredbe BL umetnu se podatci označeni labelama
- **Pozivatelj** koristi **labele** da upiše argumente
 - jednako kao pristup fiksnim lokacijama, ali je njihov položaj drugačiji – nalazi se unutar naredaba i to „blizu” pozivatelju

```
    ; POZIVATELJ
    ...
    STR R5, PAR ; šalji argument
    BL  FUNKC
PAR  DW  0
    STR R0, REZ
    ...

FUNKC ; u LR je povratna adresa
      ; dohvat parametra
      ; pomak LR za 4 mjesta
      ; tijelo
      MOV PC, LR
```



Prijenos lokacijama iza BL

- **Funkcija** čita parametre tako da za njihovo adresiranje koristi **povratnu adresu** zapisanu u LR jer ona upravo **pokazuje na parametre**
 - udaljenost parametara od funkcije je nebitna jer se LR koristi kao bazni registar
- Na isti način može se vratiti i rezultat
- Programer treba paziti da ne dođe do greške u programu, jer sada su „podatci izmiješani s naredbama”

```
    ; POZIVATELJ
    ...
    STR R5, PAR ; šalji argument
    BL  FUNKC
PAR  DW  0
    STR R0, REZ
    ...

FUNKC ; u LR je povratna adresa
      ; dohvat parametra
      ; pomak LR za 4 mjesta
      ; tijelo
      MOV PC, LR
```

Prijenos lokacijama iza BL - primjer

Napisati funkciju `bzp2k(bzp)` koja u dvostrukoj preciznosti pretvara zapis BZP (zapis s bitom za predznak) u zapis 2^k (dvojnim komplementom). Parametar `bzp` i rezultat treba prenositi lokacijama iza BL.

Glavni program treba pretvoriti broj `-0x35` iz jednog zapisa u drugi i spremiti rezultat u memoriju na lokaciju REZ.

Prikaz sa bitom za predznak u dvostrukoj preciznosti:

Najviši 64. bit čuva predznak, a niža 63 bita čuvaju iznos broja:

$$-0x35 = 0x\ 8000\ 0000\ 0000\ 0035$$

Pretvorba: Zapis u dvojnem komplementu jednak je za pozitivne brojeve. Za negativni broj treba prvo obrisati bit za predznak, a zatim 64-bitni broj treba dvojno komplementirati.

```
MAIN MOV R13, #0x10000 ; inicijalizacija stoga

      MOV R1, #0x80000000
      MOV R0, #0x35
      STR R1, BZPH      ; spremi viši dio parametra
      STR R0, BZPL      ; spremi niži dio parametra

      BL  BZP2K          ; poziv funkcije
      BZPL DW 0           ; 2 mjesta za parametar
      BZPH DW 0
      REZL DW 0           ; 2 mjesta za rezultat
      REZH DW 0

      ; mjesto nastavka izvođenja
      LDR R0, REZL      ; kopiraj rezultat u memoriju
      LDR R1, REZH
      MOV R2, #REZ
      STMIA R2, {R0-R1}
      SWI 0x123456

      REZ  DW 0, 0       ; mjesto za pohranu rezultata
```

BZP2K STMFD SP!, {R0-R1}

LDR R0, [LR], #4 ; dohvat parametara + pomak LR

LDR R1, [LR], #4 ; LR sada pokazuje na REZL

; ispitaj predznak

TST R1, #0x80000000

BEQ POZ

NEG ; briši predznak

BIC R1, R1, #0x80000000

; operacija dvojnog komplementa

; u dvostrukoj preciznosti

MVN R0, R0

MVN R1, R1

ADDS R0, R0, #1

ADC R1, R1, #0

POZ ; spremi rezultat + pomak LR

STR R0, [LR], #4

STR R1, [LR], #4

; LR sada pokazuje na mjesto povratka

LDMFD SP!, {R0-R1}

MOV PC, LR

U pozivatelju:

...

BL BZP2K

BZPL DW 0

BZPH DW 0

REZL DW 0

REZH DW 0

; nastavak izvođenja

...

Primjer dijeljenja NBC brojeva

Napisati funkciju koji cjelobrojno dijeli dva NBC broja. Parametri se šalju lokacijama iza BL. Rezultat i ostatak se vraćaju registrima R0 i R1. Zanimajte slučaj dijeljenja nulom.

Dijeljenje ostvariti metodom uzastopnog oduzimanja:

$$7/2 \Rightarrow 7-2-2-2$$

\Rightarrow 3 uspješna oduzimanja \Rightarrow rezultat je 3, a ostatak je 1

```
uint Rez; // 2 globalne varijable za rezultat, zato jer
uint Ost; // C-ova funkcija može vratiti samo jedan rezultat

void dijeli(uint A, uint B) { // dijeli se: A/B
    Rez = 0;
    while ( A >= B ) {
        A = A - B;
        Rez = Rez + 1;
    }
    Ost = A;
}
```

Primjer dijeljenja NBC brojeva

```
DIJELI STMFD SP!, {R2}      ; spremanje konteksta i dohvat parametara
    LDR    R1, [LR],#4      ; R1 je parametar A (ujedno i ostatak)
    LDR    R2, [LR],#4      ; R2 je parametar B
    ; gornje dvije naredbe mogu i kraće: LDMIA LR!, {R1, R2}

    MOV    R0, #0           ; Rez = 0

LOOP  CMP    R1, R2          ; while( A >= B )
      BLO   KRAJ
      SUB   R1, R1, R2       ; A = A - B
      ADD   R0, R0, #1       ; Rez = Rez + 1
      B     LOOP

KRAJ  LDMFD  SP!, {R2}       ; obnova konteksta s povratkom
      MOV   PC, LR
```

Zadatak: Modificirajte program tako da radi za brojeve u formatu 2'k.

Prijenos lokacijama iza naredbe BL

- Po karakteristikama sličan prijenosu fiksnim lokacijama
- Prednost mu je da položaj lokacija za prienos parametara ne mora biti blizak i pozivatelju i funkciji

Ugniježđene funkcije

Ugniježdene funkcije - problem

- Funkcije pozivaju druge funkcije (često) ili rekurzivno same sebe (rijeđe)
- ARM-ov mehanizam spremanja povratne adrese u LR to onemogućuje jer **ugniježđeni poziv će prepisati LR novom povratnom adresom**
- Većina procesora automatski sprema povratnu adresu na stog tako da se funkcije mogu bez problema gnijezditi
- ARM-ovo rješenje je zapravo jednostavno i čak je bolje od standardnog spremanja na stog (iako sve treba „ručno isprogramirati“).

Ugniježdene funkcije - rješenje

- Za svaku funkciju ionako uvijek moramo **odrediti kontekst.**
 - Kontekst ovisi o naredbama upotrijebljenima u tijelu – ako naredba mijenja neki registar, on postaje dio konteksta i **mora se spremiti na stog**
- Ako **u tijelu postoji naredba BL**, tj. ugniježđeni poziv, a znamo da BL mijenja registar LR, znači da LR postaje dio konteksta => **LR treba spremiti na stog**
- Ovo je vrlo efikasno jer se **izvodi samo tamo gdje je nužno**
 - sve „vršne” funkcije (a takvih ima puno) imaju brzo spremanje povratne adrese u LR (za razliku od ostalih procesora koji imaju sporije spremanje na stog za svaku funkciju)

Primjer gniježđenja funkcija

Napisati funkcije `isupper(ch)` i `islower(ch)` koje ispituju je li ASCII-znak `ch` veliko odnosno malo slovo.

Koristeći prethodne dvije funkcije treba napisati funkciju `isalpha(ch)` koja ispituje je li ASCII-znak `ch` slovo. Sve tri funkcije primaju parametar `ch` preko `R1` i vraćaju rezultat preko `R0` (0=false, 1=true).

Glavni program mora u nekom stringu sva slova zamijeniti razmacima.

```
ISUPPER  CMP R1, #65    ; 'A'=65
          BLO NIJE1
          CMP R1, #90    ; 'Z'=90
          BHI NIJE1
```

```
JE1      MOV R0, #1
          MOV PC, LR
```

```
NIJE1    MOV R0, #0
          MOV PC, LR
```

```
ISLOWER  CMP R1, #97    ; 'a'=97
          BLO NIJE2
          CMP R1, #122   ; 'z'=122
          BHI NIJE2
```

```
JE2      MOV R0, #1
          MOV PC, LR
```

```
NIJE2    MOV R0, #0
          MOV PC, LR
```

Primjer gniježđenja funkcija

```
ISALPHA STMFD SP!, {LR}
```

```
    BL    ISUPPER
```

```
    CMP   R0, #1
```

```
    BEQ   VAN
```

```
    BL    ISLOWER
```

```
VAN    LDMFD SP!, {LR}
```

```
    MOV   PC, LR
```

```
MAIN   MOV   SP, #0x10000
```

```
        MOV   R2, #STRN
```

```
LOOP   LDRB   R1, [R2], #1
```

```
        CMP   R1, #0 ; kraj stringa?
```

```
        BEQ   KRAJ
```

```
        BL    ISALPHA
```

```
        CMP   R0, #1
```

```
        BNE   LOOP
```

```
SLOVO  MOV   R0, #32 ; razmak=32
```

```
        STRB  R0, [R2, #-1]
```

```
        B     LOOP
```

```
KRAJ   SWI   0x123456
```

```
STRN    DSTR "As8 5kLvb7(Jd.$u"
```

Primjer gniježđenja funkcija

- Zadatci:
 - Preradite rješenje tako da funkcija isupper vraća rezultat preko R1.
 - Preradite rješenje tako da funkcija isupper vraća rezultat preko R1, a prima parametar preko R2.
 - Zašto je praktičnije da sve funkcije vraćaju rezultat preko istog registra?
 - Teži zadatak: Napišite funkciju isupper tako da kao vrijednost true umjesto 1 vraća bilo koji broj različit od 0. Zatim skratite ovu funkciju na samo 5 naredaba.

Rješenje težeg zadatka:
nemojte gledati nego
probajte riješiti sami :)

```
SUBS R0, R1, #64  
MOVLO R0, #0  
CMPHI R0, #26  
MOVHI R0, #0  
MOV PC, LR
```

Ugniježdene funkcije i prijenos lokacijama iza BL

- Kad se u nekoj funkciji F parametri ili rezultati prenose lokacijama iza BL, i ako ta funkcija F poziva daljnje funkcije...
 - onda treba paziti kako se radi s registrom LR
- Ako funkciju pišemo kao do sada, funkcija će raditi pogrešno

- PRVI PROBLEM
- Ako se na početku LR kao dio konteksta spremi na stog, a zatim se koristi za pristup parametrima i pomiče se...
 - ...onda će nakon obnove konteksta opet pokazivati na parametre umjesto na povratnu adresu → KRIVO

Primjer - Pogrešno

Funkcija $F(X,Y)$ računa i vraća $G(X)+Y+123$. F prima parametre preko lokacija iza BL i vraća rezultat preko $R0$. Pretpostavite da $G(X)$ prima parametar i vraća rezultat preko $R0$ i da već postoji u memoriji. Glavni program treba pozvati $F(77,88)$ i spremiti rezultat u REZ .

```
MAIN    MOV    SP, #10000
        MOV    R0, #77
        STR    R0, P1
        MOV    R1, #88
        STR    R1, P2
        BL     F
P1       DW     0
P2       DW     0
POVADR   STR    R0, REZ
        SWI    0x123456
REZ      DW     0
```

; „naivna” i kriva verzija:

```
F    STMFD    SP!, {R1,LR}    ; LR pokazuje na P1
        LDR    R0, [LR], #4    ; LR := P2
        LDR    R1, [LR], #4    ; LR := POVADR

        BL     G

        ADD    R0, R0, R1
        ADD    R0, R0, #123

        LDMFD    SP!, {R1,LR} ; LR := P1
        MOV     PC, LR
```

KRIVO: LR je obnovljen
tako da opet pokazuje na
parametar P1

Primjer - Ispravno

U ovom slučaju kad se svi parametri čitaju **prije** poziva funkcije G, **rješenje** je jednostavno - treba korigirati iznos LR nakon obnove konteksta:

```
MAIN  MOV  SP, #10000
      MOV  R0, #77
      STR  R0, P1
      MOV  R1, #88
      STR  R1, P2
      BL   F
P1     DW   0
P2     DW   0
POVADR STR  R0, REZ
      SWI  0x123456

REZ    DW   0
```

```
F  STMFD SP!, {R1,LR} ; LR pokazuje na P1
   LDR   R0, [LR]
   LDR   R1, [LR, #4]

   BL    G

   ADD   R0, R0, R1
   ADD   R0, R0, #123

   LDMFD SP!, {R1,LR} ; LR opet pokazuje na P1
   ADD   LR, LR, #8
   MOV   PC, LR
```

Znamo da će nakon obnavljanja
LR opet pokazivati na prvi
parametar pa ga lako korigiramo

- DRUGI PROBLEM
- Ako se parametrima ili rezultatu (koji su iza BL) treba pristupati **nakon** daljnjeg poziva funkcije, onda imamo dodatne probleme...
 - zato jer naredba BL za poziv funkcije mijenja LR i on više ne pokazuje na željene parametre ili rezultat

Primjer - Pogrešno

Funkcija F(X) računa i vraća $G(X)+123$. F prima parametar i vraća rezultat preko lokacija iza BL. Pretpostavite da G(X) prima parametar i vraća rezultat preko R0 i da već postoji u memoriji. Glavni program treba pozvati F(77) i spremiti rezultat u REZUL.

```
MAIN    MOV    SP, #10000
        MOV    R0, #77
        STR    R0, PAR
        BL     F
PAR      DW     0
REZ      DW     0
POVADR  LDR     R0, REZ
        STR    R0, REZUL
        SWI    0x123456

REZUL   DW     0
```

; „naivna” i kriva verzija:

```
F  STMFD SP!, {R0, LR}      ; LR == PAR
   LDR    R0, [LR]
   BL     G
X  ADD    R0, R0, #123       ; LR == X
   STR    R0, [LR, #4]      ; upis na X+4
   LDMFD SP!, {R0, LR}      ; LR := PAR
   ADD    LR, LR, #8
   MOV    PC, LR
```

KRIVO: LR više ne
pokazuje na REZ, jer ga
je poziv „BL G” pokvario

Primjer - Ispravno

Vjerojatno je najjednostavnije **rješenje** zapamtiti LR u pomoćnom registru (ovdje je to R8), koji time također postaje dio konteksta:

```
F  STMFD SP!, {R0,R8,LR}      ; LR pokazuje na PAR
    MOV  R8, LR                ; kopiramo LR u R8, R8 pokazuje na PAR
    LDR  R0, [R8]              ; dohvat parametra sa R8
    BL   G
X  ADD  R0, R0, #123           ; LR pokazuje na X nakon povratka iz G
    STR  R0, [R8, #4]          ; upis rezultata na adresu REZ == R8+4
    LDMFD SP!, {R0,R8,LR}     ; LR sada opet pokazuje na PAR
    ADD  LR, LR, #8            ; korekcija LR na POVADR nakon obnove
    MOV  PC, LR
```

Rješenje oba problema

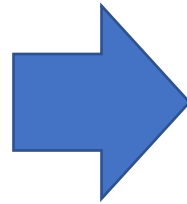
- Ako funkcija koja koristi lokacije iza BL ne poziva daljnje funkcije
 - onda ne treba raditi ništa dodatno
- Ako se LR koristi isključivo prije poziva daljnjih funkcija
 - onda treba samo korigirati LR nakon obnove konteksta, a prije povratka
- Ako se LR koristi i nakon poziva daljnjih funkcija
 - onda LR treba dodatno spremiti u pomoćni registar koji postaje dio konteksta
 - pomoćni registar se koristi za pristup parametrima/rezultatu
 - također treba korigirati LR kao i u prethodnom slučaju

- BL funkcija može skočiti na labelu koja je $\pm 32\text{MB}$ od trenutne pozicije PC
- Da li se može pozvati funkcija koja se nalazi na većoj udaljenosti?
- Ovaj problem ne može riješiti procesor nego ga rješava assembler/linker kod generiranja koda tako da u program ubacuje dio koda koji se naziva “veneer”
- Ideja je da se veneer nalazi unutar $\pm 32\text{MB}$ tako da naredba BL skoči na njega a onda se od tamo skače u bilo koji dio memorije (nešto kao “odskočna daska” 😊)

Napredno gradivo

```
ORG 0x100  
BL func1
```

```
ORG 0x100  
BL v_func1
```



```
ORG 0x1000  
v_func1 LDR PC, [PC, #-4]  
DW 0x20000000
```

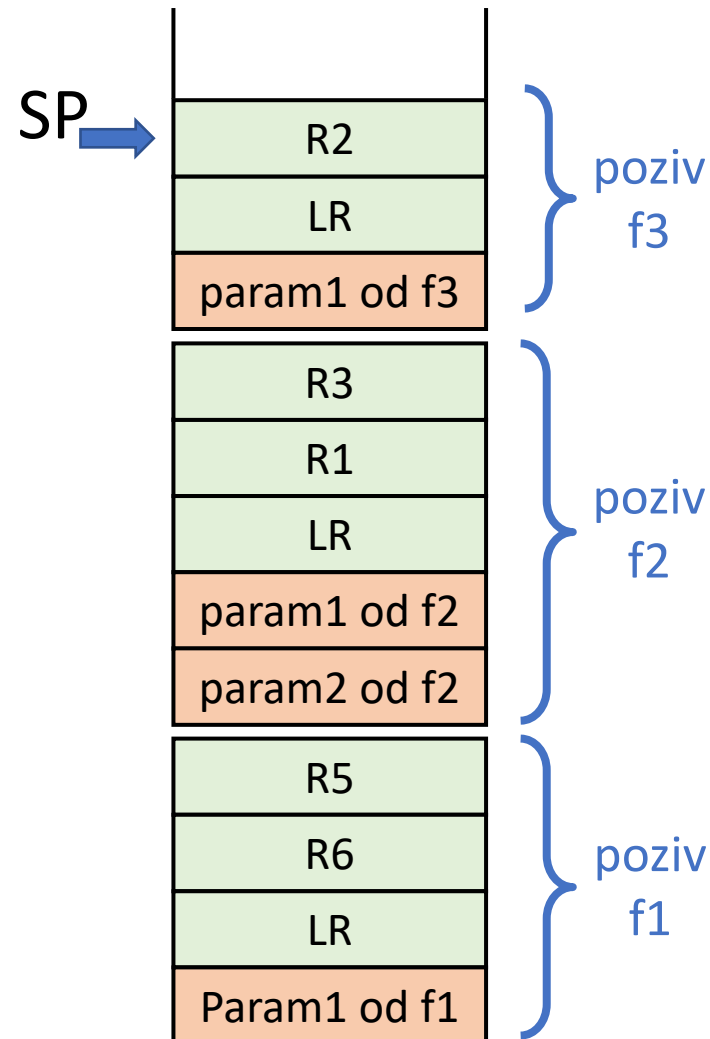
```
ORG 0x20000000  
func1 ADD ...  
MOV PC,LR
```

```
ORG 0x20000000  
func1 ADD ...  
MOV PC,LR
```

Okvir stoga

Okvir stoga

- U dosadašnjim primjerima smo vidjeli:
 - Na stog se sprema **kontekst** (može uključivati i povratnu adresu)
 - Na stog se spremaju **parametri** (ako se prenose stogom)
- Ovakav **niz podataka na stogu** naziva se **okvir stoga** (stack frame, call frame, activation record)
- **Okvir stoga** stvara se za svaki **poziv funkcije**
- Okvir stoga omogućuje međusobnu nezavisnost funkcija, njihovo gniježđenje i rekurzivne pozive



- **Lokalne varijable** u funkcijama su uobičajene u višim programskim jezicima
- U assembleru zbog efikasnosti pokušavamo **držati što više varijabli „u registrima“** (slično rade i kompajleri koji u današnje vrijeme imaju dobre optimizacije)
- Ako „potrošimo“ sve registre, morat ćemo dio podataka čuvati u lokalnim varijablama
 - Do toga dolazi u složenijim funkcijama
 - **Lokalne varijable** se smještaju u **okvir stoga**

Primjer – Lokalne varijable

Napisati funkciju `funk(x,y)` koja prima dva NBC podatka `x` i `y` pomoću stoga. Funk(`x,y`) računa rezultat $x/y + y/x$ i vraća ga pomoću registra `R0`. Funkcija treba koristiti lokalne varijable za pohranu međurezultata x/y i y/x .

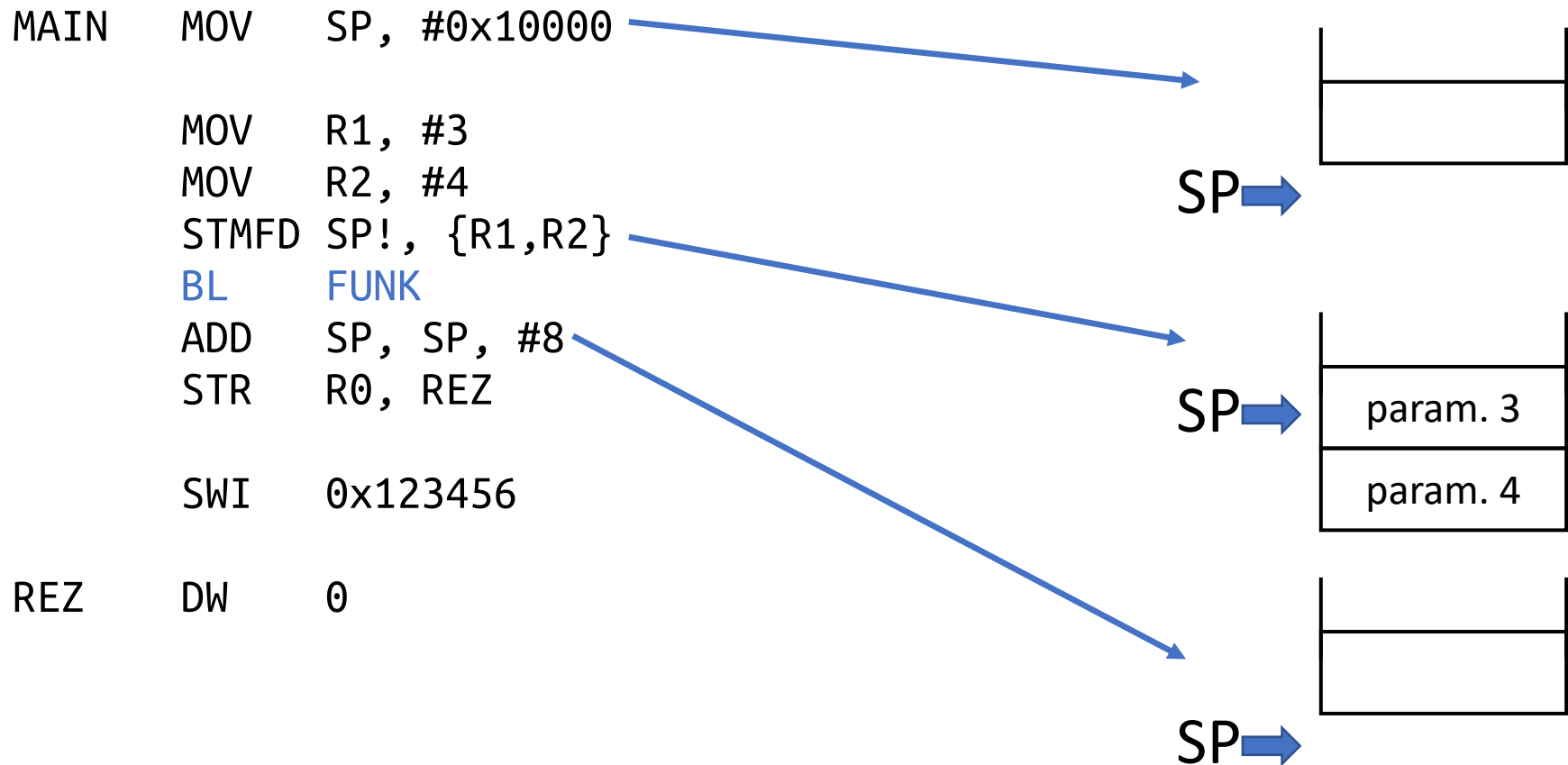
Za dijeljenje treba upotrijebiti funkciju `dijeli()` koju smo pokazali desetak slajdova ranije (parametri su iza `BL`, rezultat je u `R0`, ostatak je u `R1`).

Glavni program treba izračunati `funk(3,4)` i spremiti rezultat na memorijsku lokaciju `REZ`.

```
int funk( int x, int y ) {  
    int a, b; // lokalne varijable  
    a = dijeli(x,y);  
    b = dijeli(y,x);  
    return a + b;  
}
```

Primjer - Lokalne varijable

Rješenje ćemo, jer je tako zadano, napraviti korištenjem lokalnih varijabli na stogu (iako bismo u ovom jednostavnom primjeru mogli koristiti registre jer na raspolaganju ima dovoljno slobodnih).



Primjer - Lokalne varijable

```
FUNK STMFD SP!, {R1, LR} ; kontekst*
```

```
; stvori prostor za  
; lokalne varijable a i b
```

```
SUB SP, SP, #8
```

```
; dohvat 1. parametra u registar R0
```

```
LDR R0, [SP, #16]
```

```
; dohvat 2. parametra u registar R1
```

```
LDR R1, [SP, #20]
```

SP →

R1
LR
param. 3
param. 4

SP → SP+0

SP+4

SP+8

SP+12

SP+16

SP+20

a
b
R1
LR
param. 3
param. 4

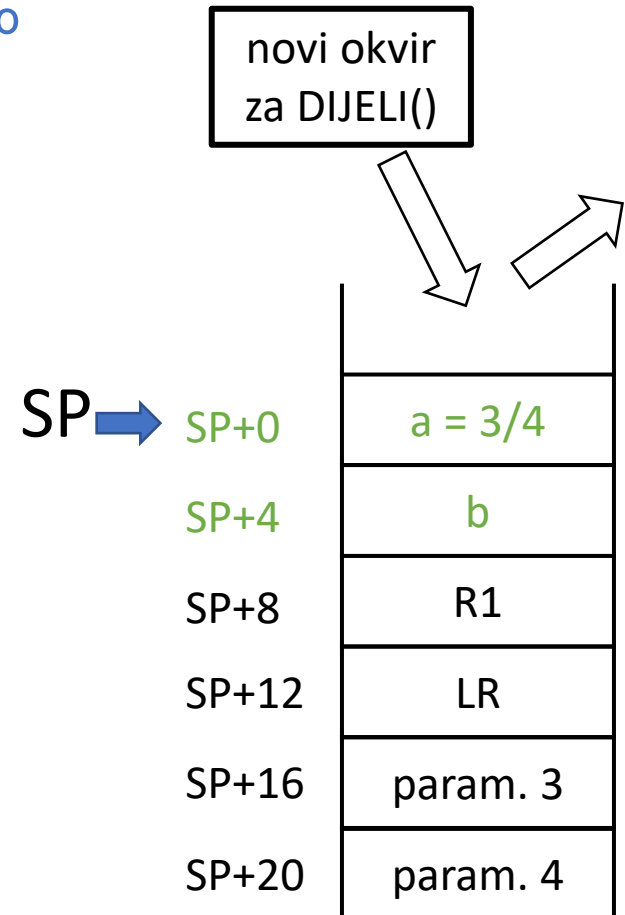
(nastavak na sljedećem slajdu)

* R1 je dio konteksta jer ga mijenja DIJELI. Istovremeno i u ovoj funkciji koristimo R1 da imamo što manji kontekst za spremati.

Primjer - Lokalne varijable

```
STR    R0, P11 ; pripremi argumente
STR    R1, P21
; R0 = P11/P21, ostatak u R1 zanemarujemo
BL     DIJELI
P11 DW    0
P21 DW    0
STR    R0, [SP, #0] ; a = P11/P21 = 3/4
```

(nastavak na sljedećem slajdu)



Primjer - Lokalne varijable

; ponovni dohvat parametara u R0 i R1

; jer je DIJELI promijenio R0 i R1

LDR R0, [SP, #16]

LDR R1, [SP, #20]

STR R0, P22 ; pripremi argumente

STR R1, P12

; R0 = P12/P22, ostatak u R1 zanemarujemo

BL DIJELI

P12 DW 0

P22 DW 0

STR R0, [SP, #4] ; b = P12/P22 = 4/3

SP → SP+0

SP+4

SP+8

SP+12

SP+16

SP+20

a = 3/4

b = 4/3

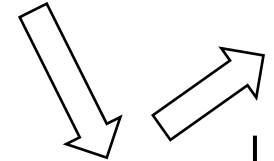
R1

LR

param. 3

param. 4

novi okvir
za DIJELI()



(nastavak na sljedećem slajdu)

Primjer - Lokalne varijable

```
; return a + b;  
LDR    R0, [SP, #0]  
LDR    R1, [SP, #4]  
ADD    R0, R0, R1
```

SP → SP+0

SP+4

SP+8

SP+12

SP+16

SP+20

a
b
R1
LR
param. 3
param. 4

```
; ukloni lokalne varijable a i b
```

```
VAN ADD    SP, SP, #8
```

```
LDMFD SP!, {R1, LR}  
MOV    PC, LR
```

SP →

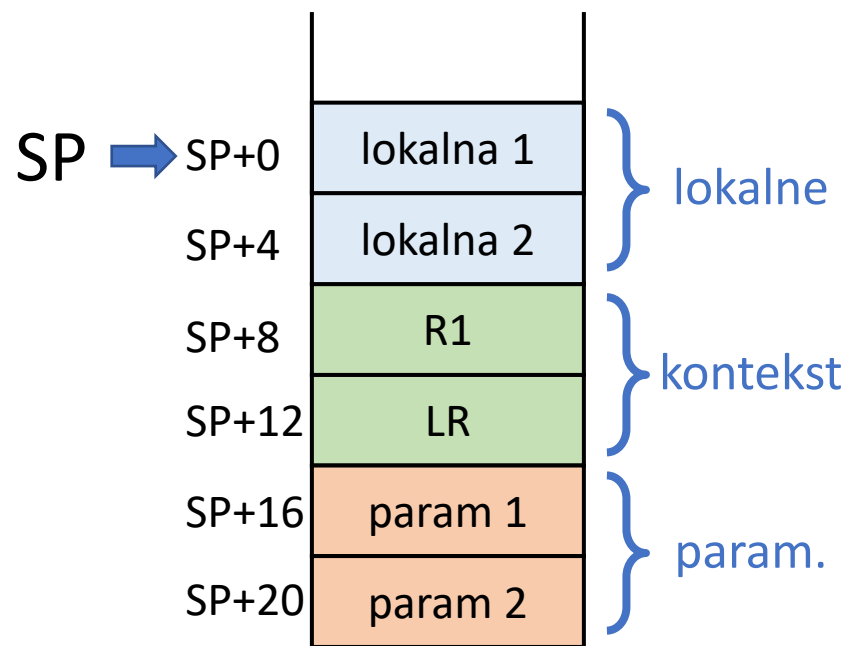
R1
LR
param. 3
param. 4

SP →

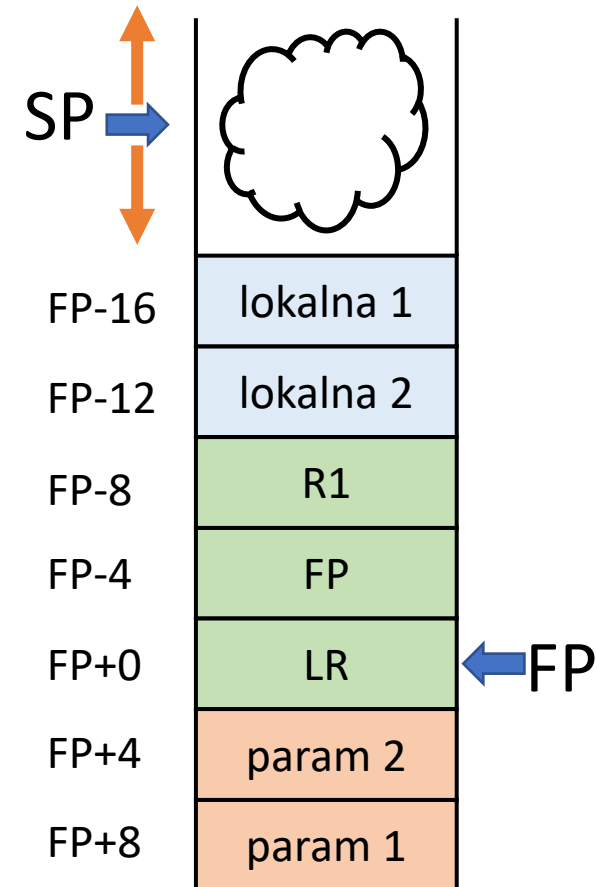
param. 3
param. 4

Okvir stoga s lokalnim varijablama

- Cjeloviti okvir stoga u općenitom slučaju sadrži:
 - lokalne varijable
 - kontekst
 - parametre
- Pristup svim podacima na okviru lako se ostvaruje jer su odmaci od SP-a fiksni
- Lokalne varijable se stvaraju jednostavnim pomicanjem SP-a
 - Lokalne varijable mogu se stvoriti ispod ili iznad konteksta



- U stvarnosti situacija je malo složenija jer funkcija može privremeno stavljati i skidati podatke sa stoga (oblačić na slici):
- Tada se SP-u mijenja vrijednost pa ofseti podataka u okviru više „ne vrijede”
- Zato se koristi dodatni registar (postoji u nekim procesorima) koji se zove **pokazivač okvira FP (frame pointer)**
- FP uvijek pokazuje na točno određeno mjesto u okviru i **ne mijenja se**
=> **podacima u okviru ne pristupa se više SP-om, nego pomoću FP-a**
- **FP je također dio konteksta** jer se pozivom funkcije mijenja njegova vrijednost



Razni primjeri

Primjer pretvorbe broja u string

Napisati pojednostavljenu funkciju itoa() koji pretvara broj u opsegu od 0 do 99 u dekadski zapis u obliku ASCII stringa (terminiranog s \0). Broj se prenosi kao parametar registrom R0. String treba zapisati od lokacije 0x100. Za dijeljenje treba upotrijebiti funkciju DIJELI() koju smo već objasnili ranije.

Rješenje: npr. broj 35 se podijeli s 10 što daje 3 i ostatak 5, a to su vrijednosti dekadskih znamenaka. String treba biti „35”.

```
ATOI      STMFD SP!, {R0, R1, R2, R5, LR}

          ; dijeljenje R0 s 10:
          ; parametri iza naredbe BL
          ; R0=rezultat (desetica), R1=ostatak dijeljenja (jedinica)
          STR    R0, P1
          BL     DIJELI
P1        DW     0    ; mjesto za prvi parametar
P2        DW     10   ; drugi parametar je uvijek 10
```

(nastavak funkcije na sljedećem slajdu)

Primjer pretvorbe broja u string

(nastavak funkcije s prethodnog slajda)

```
KRAJ  ADD    R0, R0, #48      ; pretvorba desetica u ASCII znamenku
      ADD    R1, R1, #48      ; pretvorba jedinica u ASCII znamenku

      MOV    R5, #0x100       ; adresa rezultata, tj. stringa

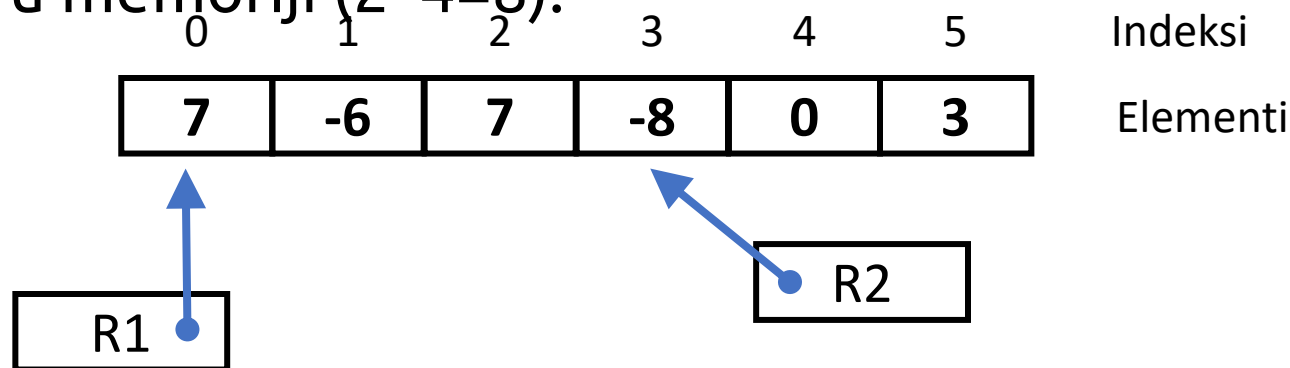
      ; „stvaranje” stringa
      STRB   R0, [R5], #1     ; upis ASCII-desetica
      STRB   R1, [R5], #1     ; upis ASCII-jedinica

      MOV    R0, #0           ; terminiranje NUL-znakom
      STRB   R0, [R5]

      LDMFD  SP!, {R0, R1, R2, R5, LR}
      MOV    PC, LR
```

Zadatak: Modificirajte funkciju tako da radi za brojeve u cijelom 32-bitnom opsegu. Pokušajte u rješenju upotrijebiti petlju koja se vrti potreban broj puta i izračunava pojedine znamenke (znamenke će se računati od jedinica ka višim znamenkama pa ih treba okrenuti u stringu)

- Često korišteni složeni tip podataka u višim programskim jezicima je **polje** (*array*).
- U assembleru se ostvaruje **blokom podataka u memoriji**.
- Treba poznavati **početnu adresu polja** (bazna adresa, R1), **indeks elementa** s kojim želimo raditi (npr. 2) i **širinu podataka u polju** (npr. 4 bajta) iz čega računamo **odmak** (*offset*) u memoriji ($2 \cdot 4 = 8$).



- **Pokazivač na elemente polja** (R2) ostvaruje se registrom ili lokacijom u kojoj se čuva adresa pojedinog elementa.

Primjer jednodimenzionalnog polja

Napisati funkciju koji radi s poljem 32-bitnih 2'k brojeva. Potrebno je sve negativne brojeve u bloku zamijeniti njihovim apsolutnim vrijednostima. Adresa polja prenosi se u funkciju registrom R1, a broj elemenata u polju registrom R2.

```
void abs_array(int *Adr, uint N) {    // ili int Adr[];

    for( uint i = 0; i < N; ++i ) {
        if( Adr[i] < 0 )
            Adr[i] = -Adr[i];
    }
}
```

Primjer jednodimenzionalnog polja

```
A_A  STMFD  SP!, {R3-R4}                ; spremanje konteksta

      MOV    R4, #0                      ; i = 0
LOOP  CMP    R4, R2                      ; i < N
      BHS    KRAJ

      ; bazna adresa Adr = R1, offset = i*4 = R4*4
      LDR    R3, [R1, R4, LSL #2 ]       ; čitanje Adr[i]

      CMP    R3, #0                      ; Adr[i] < 0
      RSBLT  R3, R3, #0                  ; Adr[i] = -Adr[i]
      STRLT  R3, [R1, R4, LSL #2 ]       ; pisanje Adr[i]

      ADD    R4, R4, #1                  ; ++i
      B      LOOP

KRAJ  LDMFD  SP!, {R3-R4}                ; obnova konteksta s povratkom
      MOV    PC, LR
```

- Za $\text{POLJE}[N][M]$ čiji element ima VEL bajtova, ofset pojedinog elementa $\text{POLJE}[i][j]$ računa se kao:

$$j * \text{VEL} + i * M * \text{VEL}$$

- Za $\text{POLJE}[N][M][Q]$ čiji element ima VEL bajtova, ofset pojedinog elementa $\text{POLJE}[k][i][j]$ računa se kao:

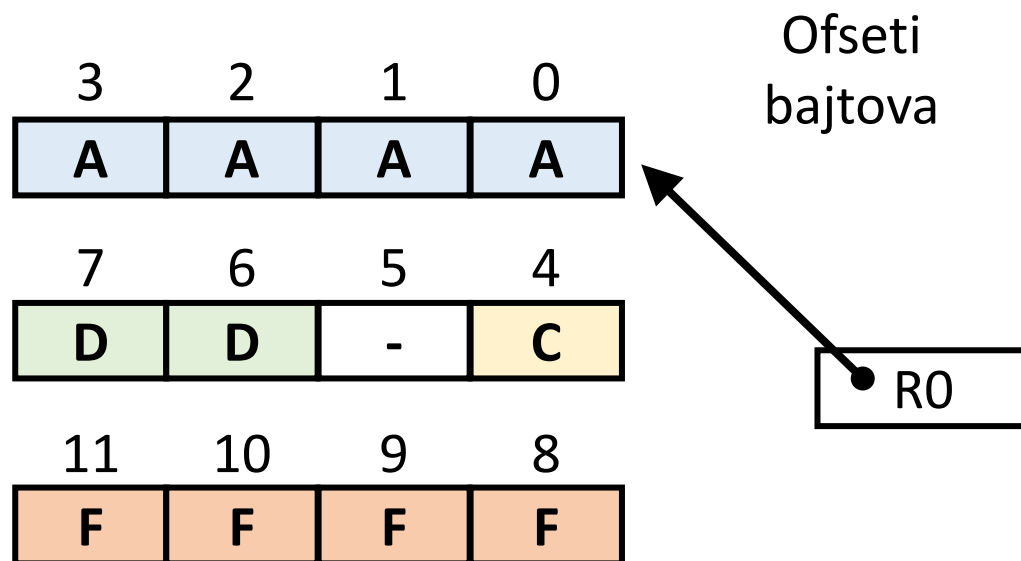
$$j * \text{VEL} + i * M * \text{VEL} + k * M * N * \text{VEL}$$

- Ovdje se vidi praktičnost početnog indeksa u C-u koji je nula, a ne jedinica što bi možda bilo intuitivnije

Strukture podataka

- Drugi složeni tip podataka koji se često koristi u višim programskim jezicima je **struktura** (*struct*).
- U assembleru se također ostvaruje **blokom podataka u memoriji**, ali pojedini elementi **nisu jednakih veličina**
- Treba poznavati **početnu adresu strukture** (bazna adresa, R0), te **ofset svakog elementa** koji se određuju na temelju **veličina svih prethodnih elemenata i adresnog poravnanja**.

```
struct S1 {    // offset
    int  A;    // 0
    char C;    // 4
    short D;   // 6
    float F;   // 8
}
```

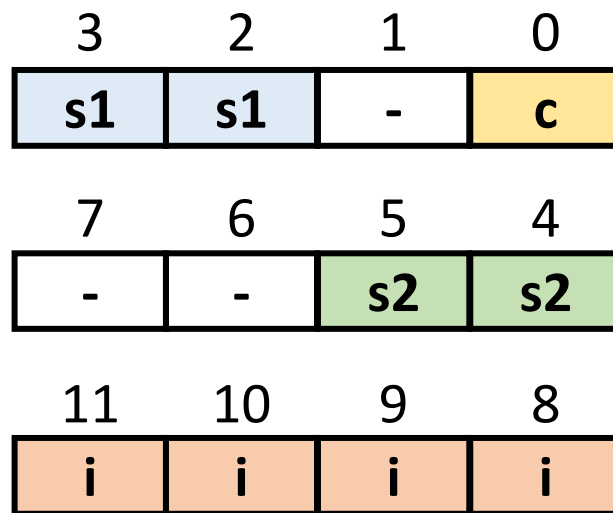


Primjer strukture

Napisati funkciju koja za zadanu strukturu zbraja vrijednosti elemenata **c**, **s1** i **s2** te zbroj sprema u **i**. Svi elementi su 2'k brojevi. Adresa polja se u funkciju prenosi registrom R0.

```
struct S2 {           // offset
    signed char  c;    // 0
    signed short s1;   // 2
    signed short s2;   // 4
    signed int   i;    // 8
}
```

```
void add_struct(Struct S2 * Adr) {
    Adr->i = Adr->c + Adr->s1 + Adr->s2;
}
```



Primjer strukture

```
AD_ST STMFD  SP!, {R1-R4}                ; spremanje konteksta

        ; učitavanje elemenata c, s1 i s2 s predznačnim proširivanjem
LDRSB   R1, [R0, #0]    ; čitanje c
LDRSH   R2, [R0, #2]    ; čitanje s1
LDRSH   R3, [R0, #4]    ; čitanje s2

ADD      R4, R1, R2      ; zbrajanje c+s1+s2
ADD      R4, R4, R3

STR      R4, [R0, #8]    ; spremanje u i

KRAJ    LDMFD  SP!, {R1-R4}                ; obnova konteksta s povratkom
MOV      PC, LR
```