8. STRUKTURE HLL-ova

Veliki broj programa napisanih na HLL je suvuše veliki. Obimni programi su teško razumljivi kao jedinstvena celina. Zbog toga se oni razlažu na veći broj manjih jedinica ili modula, kod kojih se mogu jasno sagledati izdvojene funkcije. Ovi moduli se dalje razlažu na funkcije, ili procedure, ili, kada se govori o mašinskim jezicima, na potprograme. Ove jedinice su lakše za razumevanje, lakše se može sagledati način njihovog aktiviranja, a jedinicu ispod sebe vide kao komandu. Drugim rečima programi na višem nivou se implementiraju kao hijerarhija komandi, pri čemu se svaka komanda implementira kao lista komandi za nivo ispod sebe. Ovo se završava na hardverskom nivou, gde se komande direktno izvršavaju od strane hardvera. Kod programa na mašinskom jeziku, algoritmi se razlažu na manje jedinice sve dok se ne dostigne nivo mašinske instrukcije, ali kod programa na HLL proces razlaganja se zaustavlja na nivou iskaza. Uloga kompilatora ogleda se u tome što prevodi ove iskaze u sekvencu instrukcija.

U daljem tekstu analiziraćemo funkcije (procedure se mogu smatrati kao funkcije koje ne vraćaju kao rezultat vrednost), njihov način predstavljanja, način izvršenja i podršku od strane MC68020. U daljoj analizi ukazaćemo na ulogu modula i njhovu podršku od strane hardvera.

8.1. Funkcije

Kod savremenih programskih jezika, kao što je Pascal, korišćenje funkcija (i procedura) se podstiče, tako da je povećan procenat vremena izvršenja programa koji se u programskom kodu troši na aktiviranje funkcija (procedura). Ispitujući statičku frekventnost iskaza koji deklarišu i koriste funkcije i procedure) kod FORTRAN-a (1% i 8%, respektivno) (Knuth 1971), i Pascal-a (2,6% i 31,6%, respektivno) (Shimasaki 1980), pokazalo se da je funkcija (procedura) jedna od najčešće korišćenih konstrukcija kod HLL-ova.

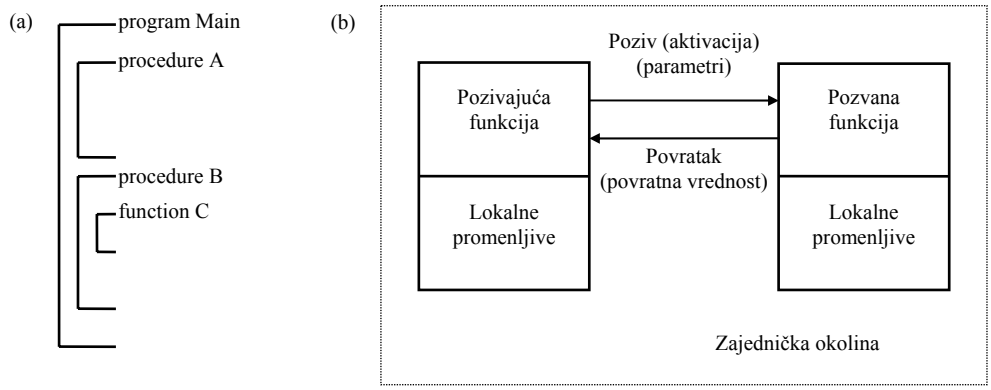
U opštem slučaju, funkcije se koriste za sledeće namene:

- **Smanjenje dužine koda** - ako se odeđena sekvenca iskaza u programu javlja na nekoliko mesta, veoma je korisno ovoj sekvenci dodeliti ime, tako da se ona mora specificirati samo jedanput, čime se obezbeđuje redukcija koda. Mehanizam funkcije se može koristiti za aktiviranje ovakvih sekvenci iskaza.
- **Skrivanje informacija** - specifikacija funkcije može (i treba) da se izvede tako da onaj koji je poziva ne treba da bude svestan njenih unutrašnjih detalja. Ovo se odnosi kako na primenjeni algoritam, tako i na internu strukturu podataka.
- **Kreiranje novih nivoa apstrakcije** - mehanizam funkcije se može koristiti za proširenje nivoa programiranja tako da on, što je moguće brže, odražava rad strukture podataka koja se zahteva od strane aplikacije. Ovo u stvari predstavlja kreiranje nove virtuelne mašine koja je prilagođena specifičnoj aplikaciji.

8.1.1. Terminologija

Na slici 8.1a prikazana je blok struktura programa na HLL-u (kao što je ALGOL ili Pascal). Blok je sekvenca iskaza i može sadržati deklaracije o lokalnim podacima. Unutar bloka, vidljive su sopstvene lokalne stavke. Stavkama se može pristupiti preko njihovih imena. Često je potrebno, ali ne uvek neophodno, da stavke budu vidljive i van bloka. Kod Pascala blokovi su povezani sa funkcijama ili procedurama: telo funkcije ili procedure je blok i to predstavlja jedini način da se kreira blok.

Blokovi se mogu definisati unutar drugih blokova. Na slici 8.1a program "Main" je spoljni blok koji obuhvata procedure A i B i funkciju C, dok su A, B i C unutrašnji blokovi programa "Main". Funkcija C je sa druge strane unutrašnji blok procedure B. Program koji sledi je na Pascalu, ima blok strukturu kao što je ona sa slike 8.1a, a prezentiran je sa ciljem da se upoznamo sa terminologijom koju ćemo pažljivo proučiti. Celokupni program, program "Main", formira spoljni blok prikazanog programa. Na slici 8.1b prikazana je terminologija aktiviranja funkcije.




```
program Main(input,output);                (*Glavni program, najspoljniji blok*)
  var I, J, K : integer; R : real;         (*Deklaracija globalnih promenlivih*)
  procedure A(var X : integer);            (*Deklaracija funkcije*)
            ⋮                              (*X je formalni parametar prenet po referenci*)
    var J, L, M : integer;                 (*Deklaracija lokalnih promenljivih*)
    begin
            ⋮                      2
    end;
  procedure B(Y : integer);                (*Deklaracija procedure*)
            ⋮                              (*Y je formalni parametar prenet po vrednosti*)
    var K, L, P : integer;                 (*Deklaracija lokalnih promenljivih*)
            ⋮
    function C(Z : real):integer;          (*Deklaracija ugnježđene funkcije*)
          var J, L, M L integer;(*Deklaracija lokalnih promenljivih*)
          begin
                  C:=...           5       (*Povratna vrednost dodeljena funkciji C*)
          end;
          begin
             ⋮                     4
           J:=C(R)      6                  (*Poziv funkcije, R je stvarni parametar*)
             ⋮                            (*povratna vrednost se dodeljuje promenljivoj J*)
          end;
begin                                      (*Početak glavnog programa*)
  I:=1;
  K:=2;                                    (*Inicijalizacija I i K*)
    ⋮                            1
  A(K);                          8         (*Poziv funkcije*)
    ⋮                            3
  B(I);                                    (*Poziv funkcije*)
    ⋮                            7
end.
```

Sl. 8.1. Struktura bloka i aktivacija funkcija.

Aktiviranje funkcije zovemo **function call** (linija 6 u programu), funkciju koja se aktivira zovemo **called fuction** (funkcija C), dok funkciju koja aktivira poziv funkcije zovemo **calling function** (procedura B). Kada se izvršenje pozvane funkcije završi, obavlja se povratak (**return**), kojim se upravljanje ponovo predaje pozivajućoj funkciji. Veliki broj jezika omogućava gnježđenje funkcija. Na primer na slici 8.1a, funkcija C je ugnježđena u proceduru B. Gnježđenjem se ostvaruje nevidljivost informacije, što znači da ugnježđena funkcija nije vidljiva van funkcije u kojoj je ona deklarisana. Funkcija C, na primer, nije vidljiva za proceduru A u programu "Main".

Kao što je već naglašeno, korišćenje funkcija omogućava programeru da izvrši podelu programa na manje potprograme, koji su konceptualno nezavisni jedan od drugog. Funkcije se izoluju od ostatka programa, jer one mogu imati svoje sopstvene *lokalne podatke* (local data), koji mogu biti bilo kog tipa - skalarnog ili strukturnog. Lokalni podaci se mogu deklarisati na početku funkcije, a njima se ne mogu obraćati globalne funkcije u odnosu na

tu funkciju. Lokalni podaci spoljnog bloka (program "Main") se zovu *globalni podaci* (global data). Mehanizam lokalnih podataka omogućava nevidljivost (skrivenost) informacije.

Primer 8.1:
Vidljivost imena u toku izvršenja prethodnog programa napisanog na Pascalu je prikazana u Tabeli 8.1.

Tab. 8.1. Vidljivost (v) imena tokom izvršenja Pascal programa Main.

	Promenljive																		
Linija	**program Main**					**procedura A**					**procedura B**					**funkcija C**			
	I	**J**	**K**	**R**	**A**	**B**	**J**	**L**	**M**	**X**	**K**	**L**	**P**	**C**	**Y**	**J**	**L**	**M**	**Z**
1	v	v	v	v	v	v	-	-	-	-	-	-	-	-	-	-	-	-	-
2	v	-	v	v	v	v	v	v	v	v	-	-	-	-	-	-	-	-	-
3	v	v	v	v	v	v	-	-	-	-	-	-	-	-	-	-	-	-	-
4	v	v	-	v	v	v	-	-	-	-	v	v	v	v	v	-	-	-	-
5	v	-	-	v	v	v	-	-	-	-	v	-	v	v	v	v	v	v	v
6	v	v	-	v	v	v	-	-	-	-	v	v	v	v	v	-	-	-	-
7	v	v	v	v	v	v	-	-	-	-	-	-	-	-	-	-	-	-	-

Ovaj deo programskog teksta za koji je dato ime vidljivo zove se *oblast važenja* (scope) tog imena. Kada su imena u određenim blokovima nevidljiva, program u tom bloku ne može koristiti ta imena.

Kada Main program startuje sa izvršenjem (linija 1), globalne promenljive I, J, K i R su vidljive, kao i imena procedura A i B. Zatim se vrši poziv procedure A. Kada se dođe do tačke 2, nevidljiva je globalna promenljiva J, jer se procedurom A deklariše lokalna promenljiva koja ima isto ime J. Ovo se zove *senčenje* (shadowing). Globalne promenljive I i K su vidljive u A, jer one nisu osenčene. Lokalne promeljive procedure A, kao i formalni parametar X procedure A, su vidljive. Kada izvršenje programa dođe do tačke 3 obavlja se povratak iz procedure A pa globalne promenljive I, J i K postaju ponovo vidljive, kao i kod linije 1. Promenljive lokalne proceduri A nisu više vidljive, što znači da globalna promenljiva J koja je bila osenčena prethodno, postaje ponovno vidljiva, a vrednost koju je imala pre poziva procedure A (linija 1) je ponovo dostupna. Slična situacija se javlja kada se pozove procedura B (linija 4), ali u ovom slučaju globalna promenljiva K je osenčena od strane procedure B. Procedura C poziva funkciju C. Kod linije 5, lokalne promenljive funkcije C senče lokalnu promenljivu L iz procedure B, i globalnu promenljivu J.

Za slučaj da se zahteva prenos podataka ka/iz pozvane funkcije, prenos se mora naznačiti u programskom tekstu koristeći parametre i povratnu (return) vrednost. *Parametri* su identifikatori koji se koriste za prenos objekata određenog tipa ka i iz pozvane funkcije. Vrednost "return" je rezultat koji se vraća natrag pozivajućoj funkciji. Kod Pascala, ova "return" vrednost se dedeljuje imenu funkcije. Drugi jezici, kao što su C i Modula-2 koriste eksplicitni "return" iskaz.

Neophodno je praviti razliku između formalnih i stvarnih parametara. *Formalni parametri* su identifikatori koji se koriste kod definicije funkcija sa ciljem da identifikuju podatke i elemente programa koji se koriste za komunikaciju sa pozivajućom funkcijom. Formalni parametri se obično specificiraju u listi formalnih parametara, i nalaze se na početku deklaracije funkcije - na primer, za proceduru A, X je formalni parametar. Stvarni parametri su specifični podaci i elementi programa koji se moraju prenositi. Opšta je praksa da se takođe specificiraju i stvarni parametri, koji se pridružuju pozivu funkcije. Na primer, kada se pozove procedura A (posle linije 1), K se specificira kao stvarni parametar.

Parametri se mogu prenositi na nekoliko načina, a metod koji se koristi je određen u zavisnosti od toga kako se izražava izraz za stvarni parametar od strane pozivajuće funkcije. Dve najčešće korišćene metode za prenos parametara su:

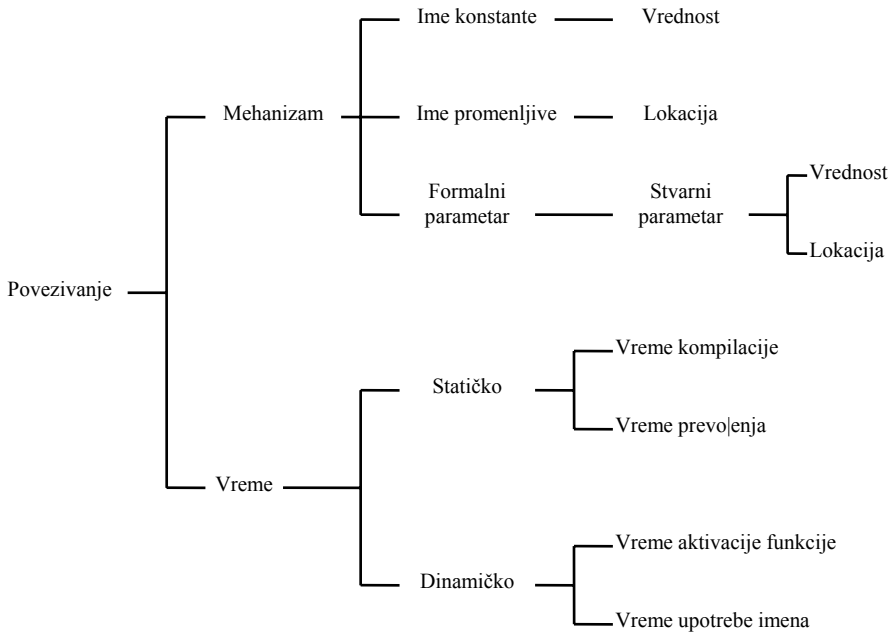
1. **Po referenci** (by reference): U toku izvršenja pozivajućeg programa, ali pre aktiviranja pozvane funkcije, adresa stvarnog parametra (na primer, adresa promenljive I ili A[J]) se predaje kao parametar. Na ovaj način parametar tipa "by reference" se uvek predaje (prenosi) kao adresa. Kao posledica ovoga, dodela ovakvog parametra nema efekat van ove funkcije. Kod Pascal-a, parametar tipa "by reference" se označava iskazom **var** u listi formalnih parametara, kao u proceduri A.
2. **Po vrednosti** (by value): Svrha tipa prenosa "by value" je da preda vrednost izraza koji se izračunava. Status parametra "by value" unutar pozvane funkcije se može smatrati identičnim statusu lokalne promenljive. Kod Pascal-a, parametri koji se ne deklarišu kao var po definiciji su (by default) su tipovi "by value".

8.1.2. Povezivanje

Koncept povezivanja (binding) čine mehanizam povezivanja i vreme povezivanja. (slika 8.2).

Mehanizam povezivanja

Mehanizam povezivanja određuje kako se pravi korespondencija (binding) između simboličkih imena (identifikatora) i vrednosti. Povezivanje je, u opštem slučaju, važeće samo na ograničenom delu programa koji se zove *oblast povezivanja*.



Sl. 8.2. Aspekti povezivanja.

Identifikatori koji se moraju ograničavati ne mogu označavati formalne parametre, imena konstanti i imena promenljivih (imena koja se koriste kod HLL-ova).

- **Povezivanje formalnih parametara** - čine ga dva dela: povezivanje formalnog parametra sa stvarnim parametrom i povezivanje stvarnog parametra sa vrednošću. Svaki programski jezik ima određena pravila u odnosu na korespondenciju formalnih sa stvarnim parametrima. Najpoznatije pravilo se zove *poziciona korespondencija* (positional correspondance), a to je, u trenutku aktiviranja pozvane funkcije, prvi stvarni parametar liste je povezan sa prvim formalnim parametrom. Nakon toga, stvarni parametri se moraju povezati sa vrenostima koje mogu biti numeričke vrednosti ili lokacije. Parametri koji povezuju ime-vrednost se pozivaju "by-value", a parametri koji povezuju ime-lokaciju se pozivaju "by reference".
- **Povezivanje konstanti i imena promenljivih**: Identifikatori imena konstanti u toku kompilacije su ograničeni na vrednosti, dok identifikatori za imena promenljivih su ograničeni na lokacije u kojima se smeštaju vrednosti. U svakoj poziciji u programu moguće je izlistati imena konstanti i promenljivih koje su vidljive u bloku (Tabela 8.1). Lista imena se zove *okruženje* (važi samo u toku izvršenja) ili *kontekst bloka*. Okruženje može biti različito za svako aktiviranje funkcije. Ako funkcija aktivira samu sebe (rekurzija), svako aktiviranje ima svoje okruženje. *Lokalno okruženje* funkcije sadrži lokalna imena te funkcije: to su imena koja se deklarišu u tekućoj funkciji. U opštem slučaju, lokalno okruženje se kreira (alocira) kada se funkcija pozove, a dealocira kada postoji povratak iz funkcije. Veći broj jezika takođe omogućava obraćanje konstantama i promenljivama u blokovima okruženja. Ove konstante ili promenljive zovu se *nelokalne* ili *slobodne*. Lista mogućih referenciranja van bloka se zove *nelokalno okruženje* bloka ili funkcije. Promenljive i konstante deklarisane u spoljnom bloku su *globalne*.

Vrste povezivanja

Vreme povezivanja određuje gde i u kom trenutku se čini da simboličko ime odgovara određenoj vrednosti. Ovo se može izvoditi u različitim trenucima. Pre izvršenja programa (statičko povezivanje) ili u toku izvršenja (dinamičko povezivanje).

- **Statičko povezivanje** - u slučaju kada se koristi identifkator, uloga kompilatora je da:
  a) "spoljašnje" potraži u tekstu programa blok koji sadrži deklaraciju za taj identifikator, i
  b) da koristi ovu deklaraciju.

  Ovo povezivanje se zove *statičko* ili *leksičko* povezivanje (leksičko, jer se pretražuju tekstualno zaokruženi blokovi). Statičko povezivanje se može izvesti kada se program kompiluje, ili kada se program puni, ali pre izvršenja.
- **Dinamičko povezivanje** - ovaj tip povezivanja koristi vrednosti promenljive čije se ime nalazi u najskorije aktiviranom bloku koji sadrži deklaraciju imena promenljive (konstante se uvek ograničavaju na statičko povezivanje). Dinamičko povezivanje se može izvesti nakon aktiviranja funkcije ili bloka, ili svaki put kada se koristi identifikator.

Najveći broj povezivanja uključuje neka izračunavanja, koja su rezultat, na primer, pretraživanja u simboličkim tabelama. Izvršenje programa će, zbog toga, biti efikasnije ako se povezivanje obavlja od strane kompilatora, a zove se "early binding" (ranije povezivanje). Zbog ovakvog praktičnog rezona, najveći broj jezika se definiše tako da se povezivanje konstanti i imena promenljivih obavlja u toku kompilacije. Obično je pravilo da kompilatorski jezici koriste statičko povezivanje, a interpretatorski jezici dinamičko povezivanje (mada postoje izuzeci u odnosu na ova pravila). Razlog za ovo je da se, kod interpretatora povezivanje identifikatora sa imenom promenljive vrši nakon svakog obraćanja imenu promenljive, dok se kod kompilatora ovo povezivanje izvodi jedanput u toku kompilacije. Ako interpreter treba da podržava statičko povezivanje, on mora da pretraži leksičku oblast važenja kod svakog obraćanja konstanti ili imenu promenljive, što ukazuje da treba obavljati analizu strukture programa kod svakog obraćanja. Ovo ukazuje na očiglednu neefikasnost. Kompilator ne može da podrži dinamičko povezivanje, jer to zahteva dostupnost sekvence aktiviranja za sve blokove, a to je jedino dostupno u toku izvršenja programa.

8.2. Model izvršenja

Svaka funkcija, koja se koristi kod HLL, ima svoje sopstveno okruženje. Koncept različitih okruženja je teško prevesti na mašinski jezik konvencionalnih računara, jer se za njih usvaja da postoji samo jedno okuženje za ceo program - a to je, bilo koja instrukcija može pristupiti bilo kojoj lokaciji. U tom cilju interesantno je proučiti koncept *upravljanja memorisanjem* kojim se podržavaju različita okruženja, i model zasnovan na magacinu pomoću koga se podržava statičko i dimaničko povezivanje.

8.2.1. Upravljanje memorisanjem

Pod ovim se podrazumeva da program za upravljanje radom memorije (storage manager) kada se funkcija aktivira, mora da obezbedi memorijski prostor, sa ciljem da se sačuva njeno okruženje. Kada se funkcija izvrši, njeno okruženje se može dealocirati a memorijski prostor ovog okruženja se ponovo vraća programu za upravljanje radom memorije na ponovno korišćenje.

Glavni problem kod upravljanja memorisanjem je *alokacija podataka* (data alocation). U toku izvršenja programa, imena tog programa moraju da su povezana sa memorijskim lokacijama koje moraju biti dostupne (ili alocirane) za ovu namenu. Alokacija se obično izvodi u blokovima. Svaki blok čine uzastopne memorijske lokacije, tako da se konstante i promenljive sa istom oblašću važenja (deklarisane u okviru iste funkcije) u izvornom programu nalaze u istom memorijskom bloku. Ovakav blok čine uzastopne memorijske lokacije a zove aktivacioni zapis (activation record). On se često kreira (i alocira) kada se aktivira blok ili funkcija.

Alokacija podataka se izvodi na dva načina:

1. **Statička alokacija podataka** - u ovom slučaju povezivanje ime-lokacija se obavlja u toku kompilacije. Kompilator dodeljuje memorijski prostor u toku kompilacije, a upravljanje memorisanjem nije potrebno u toku izvršenja programa. Ovo znači da se mogu koristiti apsolutne adrese, jer se adrese u toku izvršenja programa ne menjaju. Kod statičke dodele, za sve funkcije programa se obezbeđuje fiksno lokalno okruženje. I pored toga što neki jezici kao što je FORTRAN koriste ovaj metod, on nije opšte prihvaćen jer funkcije ne mogu biti "reentrant" ili rekurzivne. Ovo je posledica činjenice da kod višestrukog aktiviranja funkcije, svako aktiviranje zahteva svoje sopstveno lokalno okruženje, a pomoću ovog metoda obezbeđuje se prostor samo za jedno okruženje.
2. **Dinamička alokacija prostora** - blokovi se alociraju u toku izvršenja programa. Ovo se koristi, na primer, kod blok strukturnih jezika kao što su ALGOL i Pascal. Dinamička dodela se izvodi u dva različita trenutka:
   - **U trenutku aktiviranja funkcije** - Aktivacioni zapisi (za ALGOL blokove i funkcije, i procedure kod Pascala) se alociraju kada se aktivira funkcija, a dealociraju se kada se obavlja povratak (function return). Ako se može garantovati da će memorijski blok koji je bio zadnji zahtevan da se oslobodi prvi (to je slučaj kod "reentrance" i rekurzije) tada je moguće koristiti magacin za ovaj tip memorijske

alokacije. Kada se aktivira funkcija, novi aktivacioni zapis mora da se kreira, a on se smešta na vrh magacina. Kada se vrši povratak vrši se oslobađanje aktivacionog zapisa.

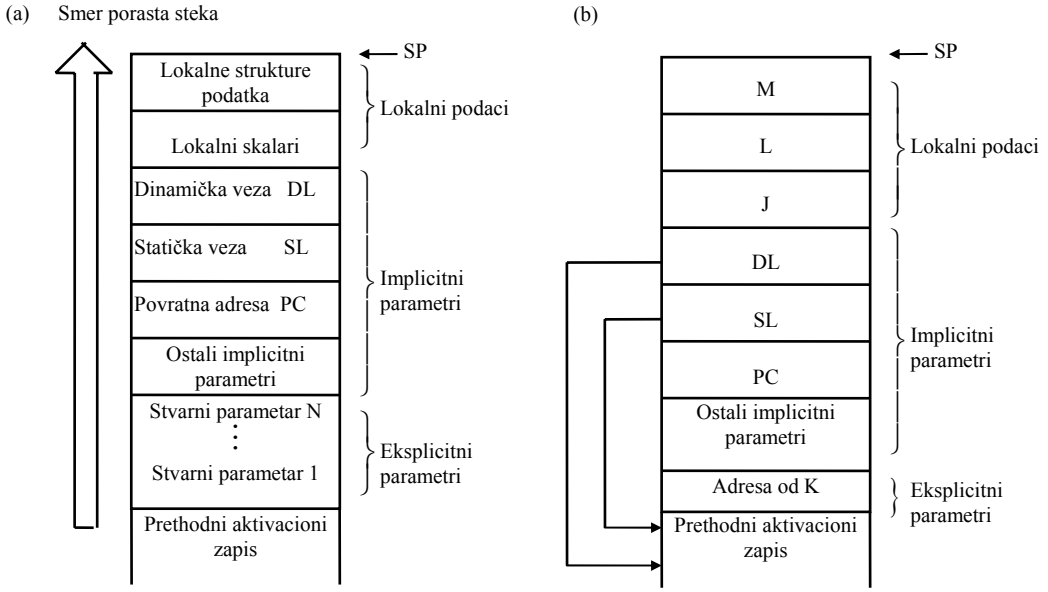
- **U trenutku korišćenja identifikatora** - ovo omogućava da se promenljivoj obezbedi memorijski prostor svaki put kada joj se dodeljuje vrednost. Shodno tome, neophodno je ugraditi generalnije algoritme za upravljanje memorijom. Kako su ovi algoritmi komplikovani, a za njihovo izvršenje je potreban dosta dug period, u principu oni imaju slabije performanse u odnosu na magacinsko-zasnovano upravljanje memorijom. Ipak, prednost se sastoji u tome što je tipovima podataka i strukturama objekata dozvoljeno da se menjaju od iskaza do iskaza. Shodno tome, samo jezici koji zahtevaju ovakvu fleksibilnost (kao što je LISP) koriste ovaj metod. No, i kod drugih blok strukturnih jezika ponekad postoji potreba za alokacijom podataka u trenutku korišćenja objekta. Da bi rešili ove probleme, najveći broj jezika koji koriste magacin radi alokacije podataka, obezbeđuju (imaju ugrađene) specijalne rutine za alokaciju podataka.

8.2.2. Podrška povezivanju

Kao što smo prethodno naglasili aktivacioni zapisi su blokovi uzastopnih memorijskih lokacija koje se alociraju kada se aktivira blok ili funkcija. Kako aktivacioni zapis ima ugnježđenu strukturu (kod blok strukturnih jezika kao što su ALGOL i Pascal) za memorisanje aktivacionog zapisa se koristi magacin.

Aktivacioni zapisi

Aktivacioni zapis čine sledeći delovi (slika 8.3):



Sl. 8.3. Struktura aktivacionog zapisa.

- **Implicitni parametri**: To je oblast koja se koristi za implicitne parametre. Poznata je kao administrativna oblast, jer se koristi od strane sistema za administriranje informacijom koja se odnosi na "run-time" model. Čine je sledeći delovi:
  a) **Statički "link"** (SL) - to je pokazivač na aktivacioni zapis za funkciju koju razmatramo. Ovaj pokazivač se koristi za statičko povezivanje.
  b) **Dinamički "link"** (DL) - to je pokazivač aktivacioni zapisa pozivajuće funkcije nakon povratka (return), a može se koristiti za dinamičko povezivanje. U najvećem broju slučajeva, DL i SL su identični. Ali kod rekurzivnih funkcijskih poziva, i kada funkcija na istom leksičkom nivou zove drugu funkciju, vrednosti SL-a i DL-a su različite.
  c) **Povratna adresa**: sačuvano stanje programskog brojača (PC), koji ukazuje gde se sa izvršenjem programa mora produžiti nakon povratka iz funkcije.
  d) **Drugi implicitni parametri**: U ovoj oblasti se čuvaju sadržaji registara opšte namene, reči stanja procesora, i dr. Koji će se registar čuvati zavisi od toga da li će se i kada isti menjati od strane pozvane funkcije.

- **Lokalni podaci**: To je oblast alocirana lokalno deklarisanim skalarima i strukturama podataka. Kada se funkcija pozove novi aktivacioni zapis se kreira u magacinu. Ovo se izvodi na sledeći način: Upravljač memorisanjem ima specijalni pokazivač - pokazivač magacina - koji se koristi za aktivacioni zapis. Kada se funkcija pozove, kreira se novi aktivacioni zapis. Njegova dužina (sa mogućim izuzećem lokalne strukture podataka) može se odrediti u toku kompilacije. Ova dužina se oduzima od pokazivača magacina, pa nova vrednost pokazivača magacina pokazuje na aktivacioni zapis. Kada se obavi povratak iz funkcije, aktivacioni zapis se dealocira, kada se pokazivaču magacina doda njegova vrednost. Naglasimo da se, tradicionalno, magacin povećava naniže prema nižim adresama. Na slici 8.3b prikazan je aktivacioni zapis koji odgovara liniji 2 procedure A, koja je pozvana u liniji 8. Predaje se samo jedan eksplicitni parametar (adresa K, jer je K parametar tipa "by-reference"). Povratna adresa (vrednost PC-a koja se čuva) biće adresa linije 3. DL i SL pokazuju na aktivacioni zapis programa "Main". Lokalne promenljive J, L i M smeštaju se na vrh magacina.

Adresiranje objekata u aktivacionom zapisu

Lokalni podaci funkcije smeštaju se u aktivacionom zapisu za funkciju, a nelokalni podaci se smestaju u drugim aktivacionim zapisima. Da bi pristupili nelokalnim podacima, mogu se koristiti SL i DL. Obično registar koji se zove *baza lokalnog imena* (LNB-local name base) ili *pokazivač okvira* (FMP - frame pointer), ukazuje na najskorije kreirani aktivacioni zapis. FMP se može koristiti kao bazna adresa za pristup lokalnim podacima i parametrima. Kada se kreira novi aktivacioni zapis, stari FMP se smešta kao implicitni parametar (DL).

Kod jezika sa dinamičkim povezivanjem (binding), pristupi konstantama i promenljivama se obavljaju pomoću DL. DL-ovi formiraju povezanu listu, tako da se dinamičko povezivanje konstanti i promenljivih vrši sledeći ovu listu.

Kod jezika sa statičkim povezivanjem, FMP se takođe može koristiti kao bazni registar za pristup lokalnim podacima i parametrima. Da bi pristupio nelokalnim podacima, SL se koristi da puni bazni registar. Ovaj registar ukazuje na aktivacioni zapis koji sadrži objektno nelokalne podatke kojima se pristupa. Ime objekta se može zameniti pomoću *nivosko-razmeštajnog para* (level-displacement pair), koji se predstavlja pomoću stavke (L,D). L predstavlja broj leksičkog nivoa koji se mora retrasirati iz tekućeg leksičkog nivoa (koji odgovara broju koraka u lancu SL pokazivača), a D predstavlja relativni razmeštaj objekta u aktivacionom zapisu. Obe ove vrednosti se određuju za vreme kompilacije.

Za povezivanje parametara, koristi se ponekad specijalni pokazivač sličan FMP-u. Ovaj pokazivač se obično zove *argument pokazivač* (AP), a ukazuje na prvi eksplicitni parametar eksplicitnog parametarskog bloka (slika 8.3a). Ovaj parametarski blok može biti deo aktivacioni zapisa, ali i može biti lociran bilo gde u memoriji. Lociranje vrednosti u memoriji se sreće kod FORTRAN-a kod koga se omogućava da se alokacija svih objekata podataka obavi u toku kompilacije tako da nema potrebe da se podaci smeštaju i izbavljaju iz magacina (troši se mnogo vremena). Nakon poziva, AP tekućeg aktivacionog zapisa se smešta u magacin kao implicitni parametar.

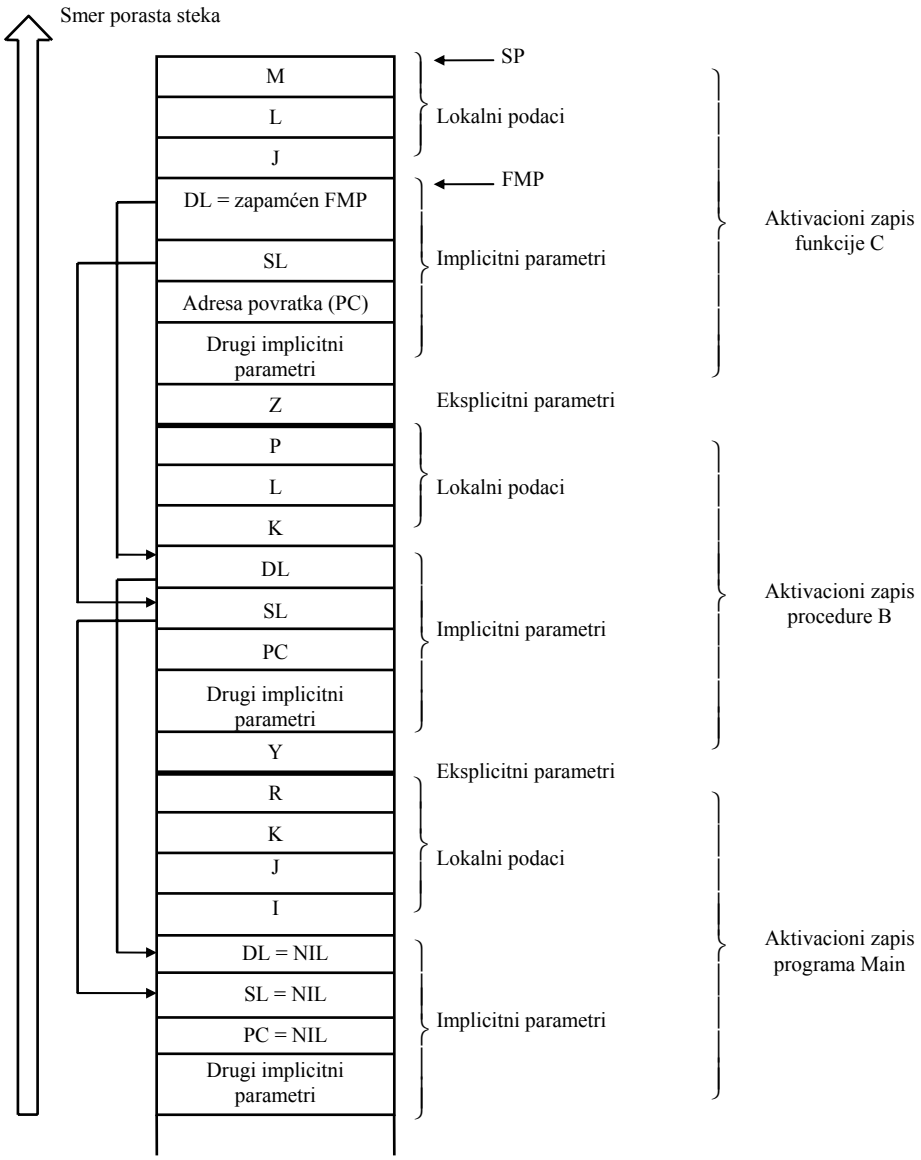
8.2.3. Model izvršenja kod ALGOL-a

Sa ciljem da opišemo model izvršenja ALGOL-a, koristićemo se programom napisanom na Pascalu, koga smo već analizirali, jer je Pascal sličan ALGOL jeziku. Na slici 8.4 prikazan je magacin sa aktivacionim zapisima programa "Main", procedure B i funkcije C (linija 5).

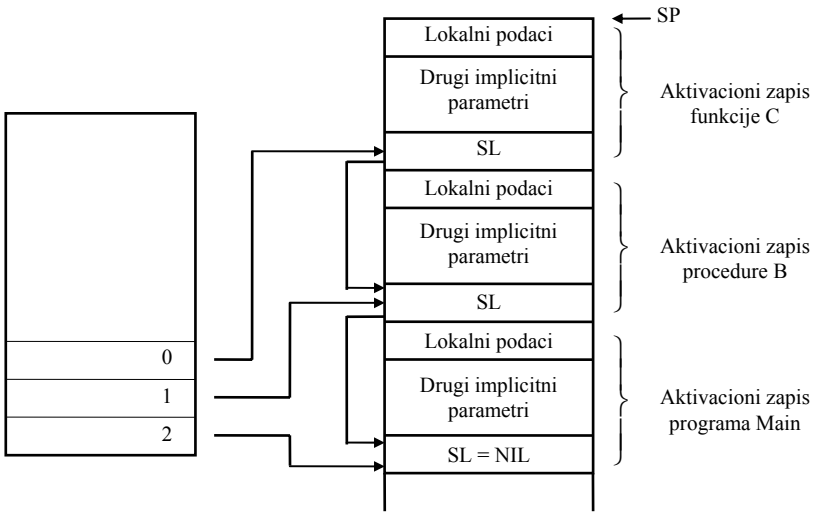
Za adresiranje u toku izvršenja lokalnih skalara, nelokalnih promenljivih, parametara, lokalnih struktura podataka i povratne vrednosti, važi sledeće:

1. **Lokalni skalari** - na prvi pogled, jasna stvar koju treba uraditi je da se koristi pokazivač magacina "SP" kao bazni registar za adresiranje lokalnih skalara. Glavni problem kod ovog rešenja ogleda se u tome što se magacin vrlo često koristi i u okviru same funkcije (na primer,u toku određivanja izraza). Ovo čini da adresiranje lokalnih promenljivih bude teško, jer razmeštaj koji je relativan u odnosu na SP nije isti od iskaza do iskaza, što čini da objektni program bude teško razumljiv. Kada je lokalnim strukturama podataka dozvoljeno da imaju dinamički obim (na primer, polje A[1..N] kod ALGOLA), tada se razmeštaj ne može odrediti u toku kompilacije, što čini da bazno adresiranje sa razmeštajem lokalnih skalara relativno na SP bude nemoguće. Zbog toga se adresiranje izvodi pomoću razmeštaja koji je relativan u odnosu na FMP. Na primer, za lokalnu promenljivu M kod funkcije C (slika 8.4) ovaj razmeštaj je -3.
2. **Nelokalne promenljive** - za nelokalne promenljive adresiranje se obavlja pomoću SL lanca (ALGOL i Pascal koriste statičko povezivanje). Da bi ukazali na statičko ugnježđeni nivo "L", vrednost "0" treba da se koristi za lokalne podatke, a vrednost "1" za celoviti (zaokruženi) blok, itd. Globalna promenljiva I treba da ima nivo-razmeštajnu vrenost od (2,-1). Dva SL pokazivača moraju da postoje da bi odredili baznu adresu aktivacionog zapisa programa "Main". Zatim se mora dodati ofset od -1 baznoj adresi da bi se odredila korektna vrednost za I. Kako "ulančavanje pokazivača" kod statičkih veza može usloviti gubljenje mnogo vremena, često se kao rešenje koristi *prikaz* (displej). Umesto da se u registru čuvaju samo najskorije statičke veze, u njemu se smešta

grupa statičkih veza. Na slici 8.5 je prikazan ovakav princip rada kod koga se koriste tri displej registra za pristup leksičkim nivoima. Na ovaj način, obraćanje nelokalnoj promenljivoj tekućeg aktivacionog zapisa je znatno brže, jer je korektna statička veza trenutno dostupna. Kod savremenih jezika, displej mehanizam je manje važan jer je dubina statičkog ugnježđenja veoma mala (tipično 1 do 3). Obično se za simulaciju prikaza koriste adresni registri.

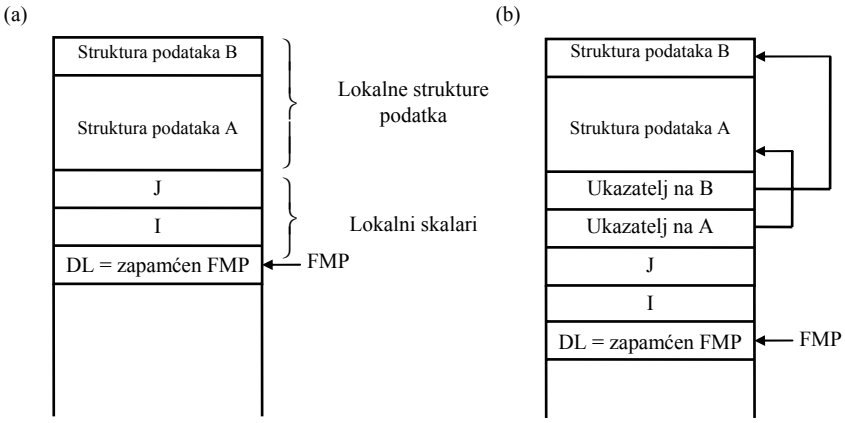


Sl. 8.4. Izgled steka za program Main (linija 5).



Sl. 8.5. Displej mehanizam.

3. **Eksplicitni parametri**: Da bi se stvari pojednostavile kompilator tretira eksplicitne parametre upravo kao lokalne promenljive. Parametrima se verovatno može vršiti obraćanje pomoću razmeštaja, koji je relativan u odnosu na FMP. Specijalni argument pokazivač (AP) je potreban kada se parametri ne prenose preko magacina, jer tada ovaj AP pokazuje na blok parametara koji su smešteni bilo gde u memoriji (sam AP je takođe smešten u magacin).
4. **Lokalne strukture podataka**: kod lokalne strukture podataka, memorijski prostor za (verovatno) veliki broj elemenata podataka treba da se alocira u magacin. Dve različite lokalne strukture podataka se mogu razlikovati:
    a) Struktuirani podaci statičkog oblika: Obim ovih struktura podataka se može odrediti u toku kompilacije. Strukture se mogu memorisati na dva načina:
        i) direktno u magacin; Na slici 8.6a prikazane su celobrojne vrednosti A[1..100] i B[2..20]. Prednost ove metode je što se podaci mogu direktno adresirati koristeći se FMP registrom kao baznim. Nedostatak je što razmeštaji mogu biti veliki, zauzimaju dodatni instrukcioni prostor i vreme za pribavljanje.
        ii) indirektno u magacin, preko pokazivača (slika 8.6b): U ovom slučaju, adresiranje elementa strukture podataka je znatno komplikovanije. Na primer, da bi se pristupilo B[x], adresa operanda od B[x]=(M[(FMP)-4])-x.
    b) Struktuirani podaci sa dinamičkim granicama: Ovim podacima, na primer, poljima celobrojnih vrednosti A[1..N] i B[2..M] ne može se pristupiti direktno, jer obim polja A nije poznat u toku kompilacije, što znači da se početna adresa B ne može odrediti u toku kompilacije. Zbog toga se mora koristiti indirektni metod prikazan na slici 8.6b.



Sl. 8.6. Adresiranje strukturnih tipova podataka.

5. **Povratne vrednosti** - kada se obrada funkcije završi, njen rezultat, koji se zove *povratna vrednost*, je potrebno negde smestiti. U principu se može koristiti magacin, što ima prednost da se vrednosti bilo koje dužine mogu dobiti kao povratne vrednosti. Pri ovome treba uočiti da se povratna vrednost smešta na vrh magacina (jer to je bila zadnja aktivnost funkcije), dok operacija povratka zahteva da se aktivacioni zapis (koji je ispod povratne vrednosti) mora dealocirati. Jedno od rešenja je da se kopira povratna vrednost ponovo u magacin kada se aktivacioni zapis dealocira, ali to dovodi do vremenskog prekoračenja, i zahteva da dužina povratne vrednosti bude poznata rutini koja vrši kopiranje. Druga situacija se javlja kada treba smestiti povratnu vrednost u registar. Ovo je slično drugom rešenju sa izuzetkom da su registri deo stanja funkcije, a njihovi sadržaji se pamte kada se pozove druga funkcija. Šta više, mogući tipovi podataka koji predstavljaju povratne vrednosti se ograničavaju na tipove podataka koji se mogu smestiti u registar, na primer, na celobrojne vrednosti, znakove, i nestrukturne tipove podataka. Najveći broj jezika, ipak, samo omogućava funkcijama da povrate vrednost skalarnog tipa, koja ima fiksnu dužinu (verovatno zbog prethodno pomenutog problema). Zbog toga najveći broj aplikacija koristi interne CPU-ove registre za smeštaj povratnih vrednosti pošto se taj tip prenosa ostvaruje za najkraći vremenski period, tj. registarski prenos je najbrži.

8.3. Arhitekturna podrška funkcijama

Kao što smo već ukazali, jedna od najvažnijih konstrukcija kod savremenih HLL-ova je mehanizam funkcija. Računarske arhitekture treba zbog toga da obezbede podršku ovom mehanizmu. Podrška se pre svega ogleda kroz sledeće aspekte: predaja parametara, mehanizam poziva, lokacija adrese podataka, čuvanje stanja i obnavljanje stanja, rezervacija prostora lokalnim podacima i prostor gde se smeštaju rezultati nakon povratka iz funkcije.

Prostor za smeštaj aktivacionog zapisa može se obezbediti od strane pozivajuće funkcije, pozvane funkcije ili sistema, preko programa za upravljanje radom memorije (storage manager). Maksimalan iznos memorije za smeštaj koji se može obezbediti od strane pozivajuće ili pozvane funkcije je ograničen, jer je taj prostor deo njegovog sopstvenog lokalnog prostora. Sistem, ipak, može da obezbedi bilo kakav prostor do veličine memorijskog podsistema. Način kako se obezbeđuje prostor koji se zahteva za aktivacioni zapis ima efekat na ponovno korišćenje (reusability) funkcije. Funkcije koje same sebe menjaju mogu se koristiti samo jednom. Bilo bi dobro, kao pravilo, da se od funkcije zahteva da se može ponovo koristiti, jer bi se inače, kod svakog poziva, zahtevalo da se pravi nova kopija pozvane funkcije (i njenih podataka). Funkcije mogu da se učine pogodnim za ponovno korišćenje na jedan od sledećih načina: serijski da se ponovo koriste, da su "reentrant" tipa, ili da su rekurzivne.

Funkcije koje se *serijski ponovo koriste* (serially reusable) mogu imati samo jedan aktivacioni zapis u datom trenutku. Ovo je slučaj kada pozvana funkcija oslobađa prostor za aktivacioni zapis. Sledeći aktivacioni zapis može početi samo kada se prethodni završi. Funkcije se zovu "*reentrant*" ako se one tako definišu da se mogu istovremeno (simultano) aktivirati od strane dve ili većeg broja pozivajućih funkcija. Ovaj tip funkcija može da ne menja sam sebe i zahteva poseban prostor za svaki aktivacioni zapis. Prostor se može obezbediti od strane pozivajuće funkcije ili sistema.

Na ovaj način, kada sistem obezbedi prostor za aktivacioni zapis, "reentrance" i rekurzija su moguće. Zbog toga, najveći broj arhitektura se tako projektuje da se alokacija memorije obavlja od strane sistema.

8.3.1. Prenos parametara

Za predaju parametara koriste se sledeća mesta:

- **Magacin**: pre nego što se izvrši poziv funkcije parametri se smeštaju u magacin. Parametri se adresiraju kao podaci, lokalno u odnosu na pozvanu funkciju. Ovaj metod je veoma fleksibilan, jer omogućava "reentrancy" i rekurziju.
- **Lista parametara**: kod ovog metoda, predaje se pokazivač (obično u registar, AP), koji pokazuje na blok gde se smeštaju parametri. Kod jezika bez rekurzije, ovaj metod ima prednost jer se samo jedan parametar, pokazivač, mora predati.
- **U registrima**: parametri se pune u registre (ili, kao što je često slučaj, već su prisutni u registrima). Ovi registri se mogu koristiti bilo od strane pozivajuće ili pozvane funkcije. Parametri tipa "by value" ili "by reference" mogu se predavati ovim načinom (za obraćanje parametrima, neophodno je koristiti registarski indirektni adresni način rada). Prednost ove metode je brzina. Nedostataka je što broj parametara ne može premašiti broj dostupnih registara. Obično je ipak, broj parametara mali, tipično 1-4.

Korišćenje magacina za predaju parametara je najfleksibilnije jer automatski obezbeđuje rekurziju i ne postavlja granice što se tiče broja parametara. Zbog toga se koristi od strane najvećeg broja kompilatora. Prednost predaje parametara preko registara je povećana brzina rada, ali to zahteva dodatno usložnjavanje kompilatora zbog problema registarske alokacije.

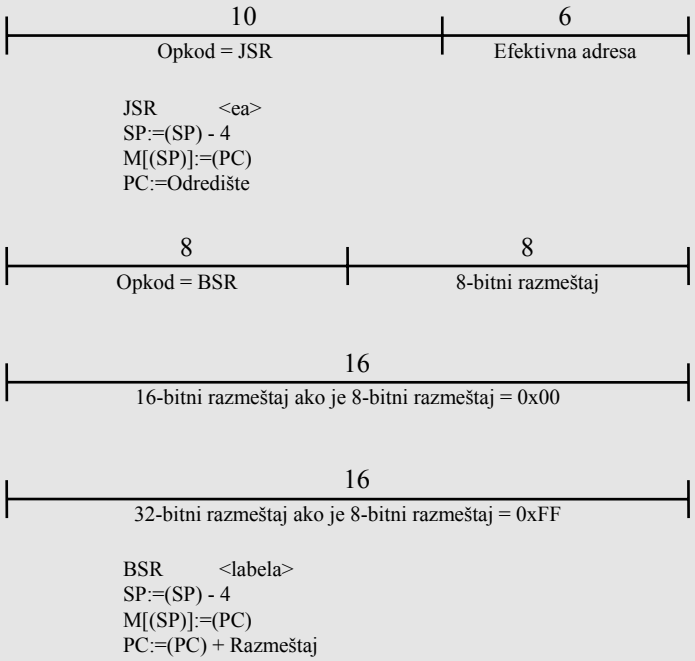
8.3.2. Mehanizmi poziva

Pozivi funkcija se često javljaju, veliki broj arhitektura podržava rad dva tipa poziva:

1. **Poziv potprograma** (subroutine call): ovi pozivi se obično koriste za poziv rutina koje ne zahtevaju svoje sopstveno okruženje (svoj sopstveni aktivacioni zapis). Kada koriste okruženje pozivajuće funkcije, one ne mogu biti rekurzivne. Tipičan primer rutine koja zahteva samo jedan prosti poziv je rutina za množenje: njen prostor memorisanja je ograničen, a rekurzija nije neophodna. Motorola MC68020 podržava poziv potprograma preko instrukcija JSR/BSR. Imajući u vidu njihovu jednostavnost one se mogu izvršiti veoma brzo.
2. **Poziv funkcija** (function call): ovi pozivi se koriste kada pozvana funkcija zahteva svoje okruženje, ipa su zbog toga kompleksniji. Najveći broj starijih arhitektura podržava, zbog razloga jednostavnosti, samo jednostavne pozive potprograma: kompleksniji pozivi funkcija su bili simulirani. Arhitekture, kao što je MC68020 imaju instrukciju tipa poziv funkcije koja se takođe koristi za podršku rada sa modulima.

Primer 8.2:

MC68020 poseduje JSR (Jump to Subroutine) i BSR (Branch to Subroutine) instrukcije (slika 8.7).



Sl. 8.7. Instrkcije JSR i BSR mikroprocesora MC68020.

Obe instrukcije smeštaju povratnu (return) adresu u magacin. Nakon toga, izvršenje programa produžava na efektivnoj adresi specificiranoj od strane instrukcije. Kod BSR instrukcije, odredišna adresa nije određena od strane efektivne adrese operanda, nego od strane 8- ili 16-bitnog razmeštaja, slično kao kod instrukcije za vrlo kratak poziv potprograma. Kod JSR Instrukcije, opšti operand se koristi za specifikaciju efektivne adrese pozvanog potprograma. Povratne adrese se izbavljaju pomoću komplementarne RTS (Return from Subroutine) instrukcije. Ova nultooperandska instrukcija izbavlja povratnu adresu iz magacina i smešta je u registar PC.

8.3.3. Lokacija povratne adrese

Nakon izvršenja pozvane funkcije, pozivajuća funkcija ili program treba da produži sa one tačke sa koje je bio učinjen poziv. Zbog toga pozvana funkcija mora da čuva informaciju o tački na koju se mora obaviti povratak. Ovo se izvodi na sledeći način:

- **U registru**: kod ovog metoda povratna adresa se automatski smešta u registar. Ovo je opšte prihvaćeno rešenje kod registarsko orijentisanih arhitektura, jer se poziv uklapa sa formatom instrukcije kod aritmetičkih i logičkih operacija.
- **U magacinu** - instrukcijom CALL smešta se stari PC u magacin (ima prednost što čini mogućim izvođenje rekurzivnih funkcija), a takođe se obezbeđuje i mehanizam za smeštaj parametara kao i lokalnih podataka u magacin. Ovaj princip se koristi od strane instrukcija JSR i BSR.

8.3.4. Čuvanje stanja i obnavljanje

U toku izvršenja potprograma ili funkcije može se zahtevati korišćenje određenih resursa, kao što su SP ili registri opšte namene. Ovi resursi mogu do tog trenutka biti korišćeni od strane onog koji je izvršio poziv pa se zbog toga njihovo stanje mora zapamtiti. Poželjno je da pozvana funkcija umesto pozivajuće funkcije obavi ovu aktivnost, jer pozvana funkcija može da odluči koliko se od stanja mašine može sačuvati, a programski kod koji će to obavljati je potrebno napisati samo jedanput (u pozvanoj funkciji a ne sve u pozivajućoj). Promene u funkciji kao što je korišćenje dodatnog registra, transparentne su u odnosu na pozivajuću funkciju. Dva aspekta su veoma važna u ovom slučaju: naime, gde se i koliko od stanja mašine mora zapamtiti.

Stanje se obično pamti u magacinu, jer on omogućava rekurzivnost. Ali kod jezika kao što je FORTRAN, ovo nije neophodno. U ovom slučaju, stanje se može zapamtiti bilo u prostoru pozvane funkcije, ili u onom prostoru koji pripada pozivajućoj funkciji.

Koliko će se od stanja zapamtiti (sačuvati) zavisi od arhitekture. Kod nekih arhitektura, kao što je MC68020, postoje specijalne instrukcije za čuvanje i izbavljanje sadržaja bilo kog registra, dok kod drugih arhitektura postoje instrukcije za eksplicitno čuvanje i izbavljanje većeg broja registara ili svih registara. Instrukcija MOVEM (Move Multiple Registers) procesora MC68020 ima masku kod koje svaki bit specificira registar, tako da se određeni broj registara može smestiti u magacin jednom instrukcijom. Svi registri i drugi delovi stanja mašine, koji su bili sačuvani moraju se izbaviti kada se vrši povratak iz pozvane funkcije. Za registre se ovo izvodi, ako je moguće, pomoću MOVEM instrukcije u obrnutom smeru, ili pomoću niza POP instrukcija.

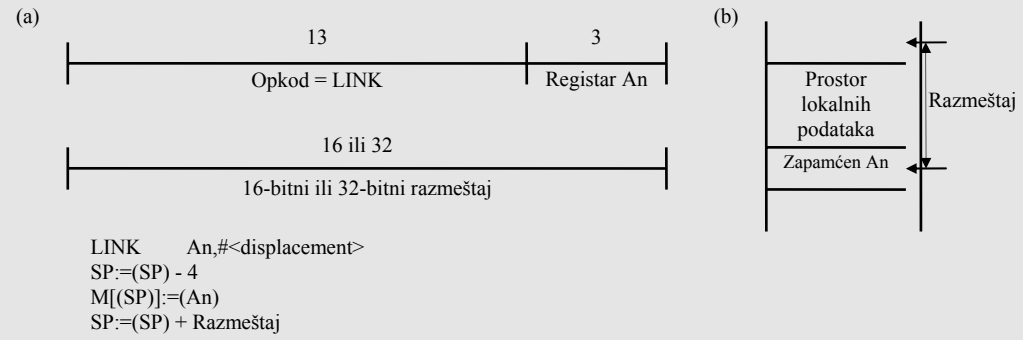
8.3.5. Rezervacija prostora za lokalne podatke

Kod onih poziva funkcija kod kojih je potrebno kreiranje aktivacionih zapisa, neophodno je da se rezerviše prostor za lokalne podatke. Tradicionalne instrukcije koje se koriste za podršku rada FMP-ova i rezervaciju prostora lokalnim podacima su LINK i njen komplement UNLINK instrukcija.

Primer 8.3:

Na slici 8.8a prikazana je LINK (Link and Allocate) instrukcija mikroprocesora MC68020. Na slici 8.8b prikazana je njena aktivnost koju čine sledeći koraci:

- smešta tekući FMP (specificiran registrom An) u magacin,
- kreira novi FMP kopiranjem tekuće vrednosti pokazivača magacina u registar An,
- rezerviše prostor u magacinu inkrementiranjem SP-a. Obim prostora koji treba da se rezerviše specificiran je negativnim 16- ili 32-bitnim razmeštajem, definisanim argumentom instrukcije.




```
LINK    An,#<displacement>
SP:=(SP) - 4
M[(SP)]:=(An)
SP:=(SP) + Razmeštaj
```

Sl. 8.8. Instrukcija LINK mikroprocesora MC68020.

8.3.6. Povraćaj memorijskog prostora

Kada se obavlja povratak iz pozvane funkcije, memorijski prostor koji se koristio za njegov aktivacioni zapis mora da se dealocira. U slučaju kada se ostvaruje poziv potprograma, a aktivacioni zapis mora da se kreira, uobičajeno je da se parametri i lokalni podaci i dalje mogu smeštati u magacin. Ovi parametri se mogu jednostavno izbavljati iz magacina podešavanjem SP-a, a to se izvodi dodavanjem konstantne vrednosti. Ovo se realizuje jednom od najstandardnijih aritmetičkih instrukcija (ADD #8,SP) ili specijalnom "return" instrukcijom. Kod MC68020 za ovu svrhu se koristi instrukcija koju zovemo RTD.

Primer 8.4:

Kod mikroprocesora MC68020 instrukcija RTD (Return and Deallocate Parameters) je kombinacija instrukcije RTS i instrukcije pomoću koje se podešava magacin sa ciljem da se dealocira memorijski prostor koji se koristi od strane parametara. Funkcije sa promenljivim brojem parametara (najveći broj jezika ne omogućava ovakve konstrukcije), ili sa lokalnim strukturama podataka, ne podržavaju se od strane instrukcije RTD. Asemblerska sintaksa je RTD #<razmeštaj>. Operacija RTD se obavlja na sledeći način:

PC:=(M[(SP)]
SP:=(SP)+4+Displacement

Kod MC68020 LINK instrukcija se koristi za kreiranje novog okruženja funkcije, a UNLINK instrukcija se koristi za obavljanje aktivnosti u obrnutom redosledu.

Primer 8.5:

UNLINK instrukcija kod MC68020 (slika 8.9) obavlja obrnute akcije od ovih predviđenih za LINK kopiranjem tekućeg FMP-a (specificiran sa An) u SP i izbavljanjem starog FMP-a iz magacina (vidi sliku 8.8b).


```
UNLK    An
SP:=(An)
An:=(M[(SP)])
SP:=(SP) + 4
```

Sl. 8.9. Instrukcija UNLK mikroprocesora MC68020.

8.4. Podrška funkcijama od strane mikroprocesora MC68020

Podršku koju komercijalni procesori daju funkcijama sagledaćemo kroz korišćenje mikroprocesora MC68020. Podršku ćemo opisati u zavisnosti od sledećih aspekata:

1. predaja parametara,
2. čuvanje stanja i obnavljanje,
3. rezervacija prostora za lokalne podatke,
4. vraćanje prostora za pamćenje,
5. mehanizam poziva i lokacija povratne adrese.

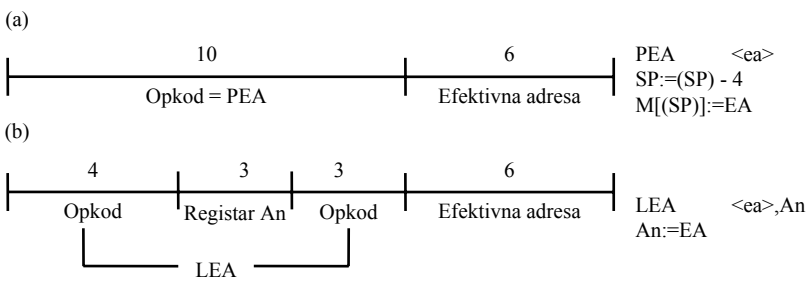
Analizirajmo redom nabrojane aspekte:

1. **Predaja parametara**: kada se vrši predaja parametara preko magacina koriste se sledeće instrukcije:

```
MOVE    par,-(SP)    ; za "by value" parametre
PEA     par          ; za "by reference" parametre (push efektivne adrese)
```

Instrukcija PEA (Push Effective Address) je prikazana na slici 8.10a. Ona smešta adresu operanda, koji je duga reč, u magacin. Kada efektivna adresa specificira neposrednu vrednost, ili Dn ili An registar, generiše se ilegalna instrukcija **trap**. Kada se prenose parametri preko registara, koriste se sledeće konstrukcije:

```
MOVE    par,Dn       ; za "by value" parametre
LEA     par,An       ; za "by reference" parametre
```



Sl. 8.10. Instrukcije za efektivne adrese mikroporcesora MC68020. (a) PEA (Push Effective Address). (b) LEA (Load Effective Address).

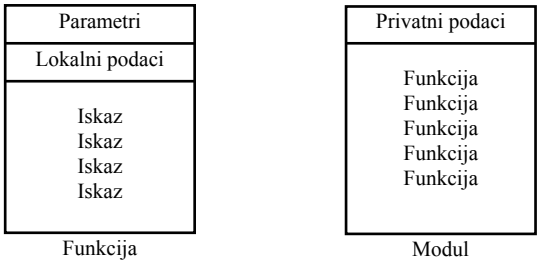
2. **Čuvanje stanja i obnavljanje**: Za push/pop većeg broja registara iz/ka magacina koristi se MOVEM instrukcija.
3. **Rezervacija prostora za lokalne podatke**: za ovo se koristi LINK instrukcija.
4. **Povraćaj prostora za čuvanje**: za ovo se koriste RTD i UNLINK instrukcije.
5. **Specifikacija pozivnog mehanizma i povratne adrese**: Kod MC68020 postoje JSR i BSR instrukcije za poziv potprograma, pomoću kojih se specificira memorijska lokacija pozvanog potprograma i smešta povratna adresa u magacin. Instrukcijom RTS vraća se upravljanje pozivajućoj funkciji.

8.5. Moduli

Jedan od metoda koji se koriste za upravljanje kompleksnošću velikih programa je modularizacija; to predstavlja deljenje programa na veći broj nezavisnih modula. Kada se ovo uradi, svaki modul je kao mali program koji se može nezavisno implementirati od drugih modula. Svaki modul se može prečistiti od grešaka, razumeti i nezavisno održavati.

Kada je broj funkcija veliki, poželjno je grupisati srodne (uzajamno povezane) funkcije i podatke zajedno, i njih tretirati kao jedan objekat. Upravo kao što je funkcija namenjena da skupi i da objedini grupu povezanih iskaza i podataka, tako je modul namenjen da skupi i objedini grupu povezanih funkcija i podataka (slika 8.11).

Termin "modul" bio je uveden kod nekoliko programskih jezika pod različitim imenima, kao što je modul u Modula-2, segment u ALGOL68 i paket kod Ada. Modul ne samo da podržava mogućnost "povezivanja" programskih konstrukcija (konstante, tipovi, promenljive i funkcije) nego takođe i podržava posebne osobine modula koje se odnose na *javni deo* (public part) i *privatni deo* (private part).



Sl. 8.11. Funkcije i moduli.

Javni deo specificira imena objekata, tipova i funkcija koje su spolja dostupne. Privatni deo sadrži telo modula, koje mogu činiti lokalni podaci i funkcije. Ovaj deo je interni što se modula tiče i nevidljiv (skriven) je od spoljnog sveta. Obezbeđeno je da funkcije van modula ne mogu menjati podatke ili koristiti funkcije modula, ako modul to ne omogućava (dozvoljava). Kada se funkcije ili podaci učine dostupnim drugim modulima, oni se *eksportuju* (izvezeni simboli) od strane modula koji je njih deklarisao (prisvojio). Sa druge strane, modul može uvoziti funkcije i podatke (uvezeni simboli) koji su bilo eksportovani od strane drugih modula.

8.6. Arhitekturna podrška modulima

Kod koji se generiše za program napisan na jeziku koji podržava koncept modula ne razlikuje se mnogo od koda generisanog za sličan program napisan na jeziku bez modula. Mehanizmi potprograma i funkcija koji se koriste za podršku funkcija mogu se takođe koristiti i za podršku modula. Nezavisno od toga, loadovanje (punjenje) i linkovanje (povezivanje) programa koga čine moduli je znatno komplikovanije, a arhitekture treba da obezbede sledeću podršku:

- **Spoljnu adresnu nezavisnost**: kod unutar modula treba da bude nezavisan od adresa uvezenih podataka i koda; tj. modul ne treba da se menja ako se promene spoljne adrese modula. Ovo se može izvesti uvođenjem specijalnih tabela, nazvanih *deskriptori funkcija*, pomoću kojih se opisuju uvezeni simboli od drugih modula. Shodno tome, kada se adrese uvezenih podataka i koda menjaju, menja se samo opisivač funkcije, dok kod modula ostaje nepromenjen. Kao rezultat, modul se može smestiti u ROM i koristiti od strane drugih modula koji nemaju fiksne lokacije.
- **Linkovanje (povezivanje) modula**: glavni problem je obezbeđivanje podrške rada samodulima na osnovu koje bi se uspešno izvršilo linkovanje većeg broja modula sa ciljem da se formira kompletan program. Zbog toga, kompilator (ili posebno "linkage editor") mora generisati module koji se mogu povezivati. Ovi moduli treba da

sadrže dovoljno informacija koje će obezbediti korektno linkovanje sa drugim modulima, sa ciljem da se kreira konačni program. Da bi ovo bilo moguće, link modul mora da sadrži listu uvezenih simbola i listu svih izvezenih simbola. Program "linkage editor" prihvata listu svih modula i povezuje eksportovane simbole modula sa importovanim simbolima drugih modula preko istih imena. On takođe proverava da vidi da li se tipovi simbola slažu.

- **Stanje čuvanje/obnavljanje**: kada funkcija pozove funkciju u drugom modulu, ona prvo mora da sačuva staro okruženje, a zatim da se napuni novim okruženjem. Ova operacija se podržava od strane specijalnih instrukcija koje mogu da sačuvaju/izbave stanje, koje je deo okruženja.

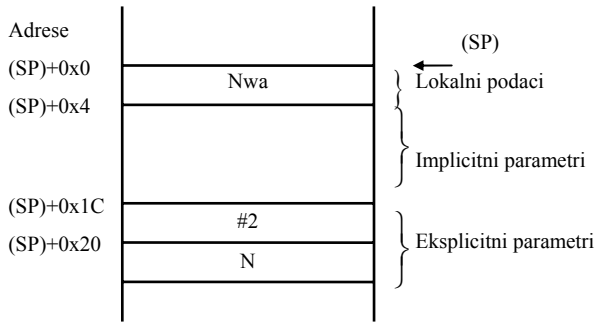
8.7. Podrška modula od strane MC68020

Kako MC68020 podržava rad sa modulima biće dato kroz dva nivoa. Na prvom nivou dat je program napisan na Pascalu i njegov odgovarajući kod na asemblerskom jeziku (mikroprocesora MC68020). Drugi nivo pokazuje detalje kako se vrši podrška modula od strane MC68020. On opisuje instrukcije CALL i RETURN zajedno sa njihovim modelom izvršenja.

8.7.1. Programiranje modula

Analizirajmo sledeći Pascal program koji se koristi za sabiranje vredosti S sa "by reference" parametrom A:

```
procedure Inc (var A:integer;S:integer);
var Nwa:integer;
begin
        Nwa:=A+S;
        A:=Nwa;
end;
```



Sl. 8.12. Aktivacioni zapis za Inc(N,2).

Kod generisan za ovaj program koji koristi podršku modula za mikroprocesor MC68020 ima sledeći oblik. Slika 8.12. prikazuje aktivacioni zapis kreiran kao posledica poziva.

Prvi deo koda prikazuje poziv Inc(N,2) zajedno sa načinom prevođenja: parametri se predaju preko magacina, a na poziv ima uticaj instrukcija CALLM koja ima operand pomoću kojeg se specificira broj neophodnih bajtova koji se zahtevaju od strane parametara.

Drugi deo sadrži kod generisan od strane tela procedure Inc. Ovaj deo čine sledeće tri komponente:

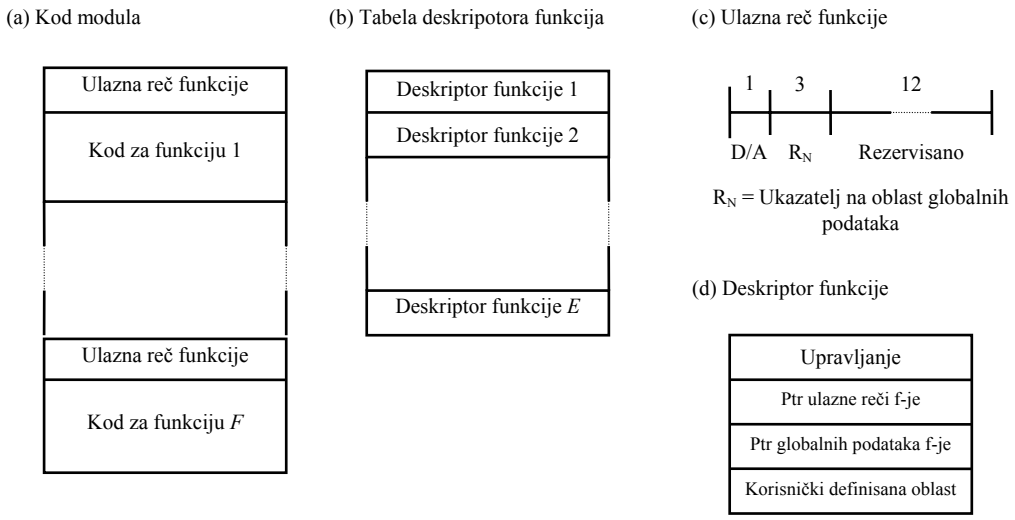
1. **Deskriptor funkcije Inc**: precizni format deskriptora funkcije dat je na slici 8.13d. Deskriptor ima upravljačku reč koja je 0, ukazujući da ne postoje posebne osobenosti, kao što je prenos parametara preko argument pokazivača. Druga reč opisivača sadrži pokazivač koji pokazuje na ulaznu reč funkcije Inc. Treća reč, koja je 0, ukazuje da ova funkcija nema globalne podatke.
2. **Ulazna reč funkcije**: ova reč prethodi kodu tela funkcije. Ona specificira da nije potreban globalni pokazivač oblasti podataka, jer funkcija ne poseduje bilo kakav globalni podatak.
3. **Kod procedure tela Inc**: zbog jednostavnosti funkcije, generisani kod je trivijalan.

Poziv procedure Inc(N,2); na Pascalu se prevodi u

```
MOVE.L      N,-(SP)  ; Push N
MOVE.L      #2,-(SP) ; Push 2
CALLM       #8,Inc   ; Pozovi Inc sa 8 bajtova za parametre
```

Telo procedure se prevodi u:

```
                                        ; 1, opisivač funkcije za "Inc"
Inc:    defl            0               ; upravljačka informacija
        defl    -       inc             ; pokazivač ulazne reči funkcije
        defl            0               ; pokazivač globalnih podataka funkcije
-inc:                                   ; 2, ulazna reč funkcije "Inc"
        defl            0xF000          ; nije potreban pokazivač globalne oblasti podataka
                                        ; kod za telo procedure Inc
        SUBQ            #4,SP           ; obezbedi radni prostor za lokalnu promenljivu Nwa
        MOVE.L          (0x1C,SP),D0    ; pribavi vrednost parametra S
        ADD.L           ([0x20,SP],0,0),D0  ; saberi referencirani parametar sa D0
        MOVE.L          D0,(SP)         ; smesti lokalnu promenljivu u Nwa
        MOVE.L          (SP),([0x20,SP],0,0)  ; A:=Nwa
        ADDQ #4,SP                      ; oslobodi lokalni radni prostor
        RTM                             ; povratak iz modula
```



Sl. 8.13. Informacije o modulima mikroporcesora MC68020.

8.7.2. CALL i RETURN instrukcije kod MC68020

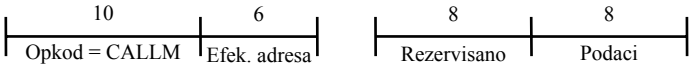
U daljem tekstu opisaćemo CALLM i RTM instrukcije, deskriptor funkcije kao i model izvršenja kojeg one podržavaju.

Mehanizam podrške modulu, kod MC68020, se zasniva na prethodno opisanom modelu izvršenja. Pored ostalog, on koristi nekoliko deskriptora koji omogućavaju pozivanje spoljnih funkcija. Ove deskriptore, zajedno sa globalnim pogledom na modul, ćemo najpre objasniti, koristeći sliku 8.13.

Na slici 8.13a prikazan je modul koga čini funkcija F. Svaka funkcija se predstavlja ulaznom funkcijskom rečju, iza koje sledi kod za tu funkciju. Na slici 8.13c prikazan je format funkcijske reči. Ona specificira koji se registar koristi od strane odgovarajuće funkcije da ukaže na oblast globalnih podataka za tu funkciju.

Funkcija F modula sa slike 8.13a može referencirati spoljnu funkciju E specificiranu u tabeli deskriptora (slika 8.13b). Ove spoljne funkcije se moraju indirektno referencirati preko deskriptora instrukcija. Izgled deskriptora funkcije za funkciju X je prikazan na slici 8.13d. Njega čine: upravljačka informacija, pokazivač ulazne reči funkcije, koji pokazuje na ulaznu reč funkcije za funkciju X, globalni pokazivač podataka funkcije, koji pokazuje na globalne podatke funkcije X, i korisničko definisane oblasti (koje su opcione).

Pozivanje funkcije koja nije lokalna u odnosu na tekući modul može se izvršiti pomoću instrukcije CALLM (slika 8.14). Efektivna adresa ove instrukcije specificira deskriptor funkcije koja treba da se pozove.



Sintaksa: CALLM #<data>,<ea>
#<data> opisuje dužinu (u bajtovima) oblasti koje se koristi za eksplicitne parametre
<ea> specificira deskriptor funkcije za pozvanu funkciju
Efekat: Pamti se stek pozivajućeg modula
Kreira se aktivacioni zapis za pozvani modul, specificiran efektivnom adresom

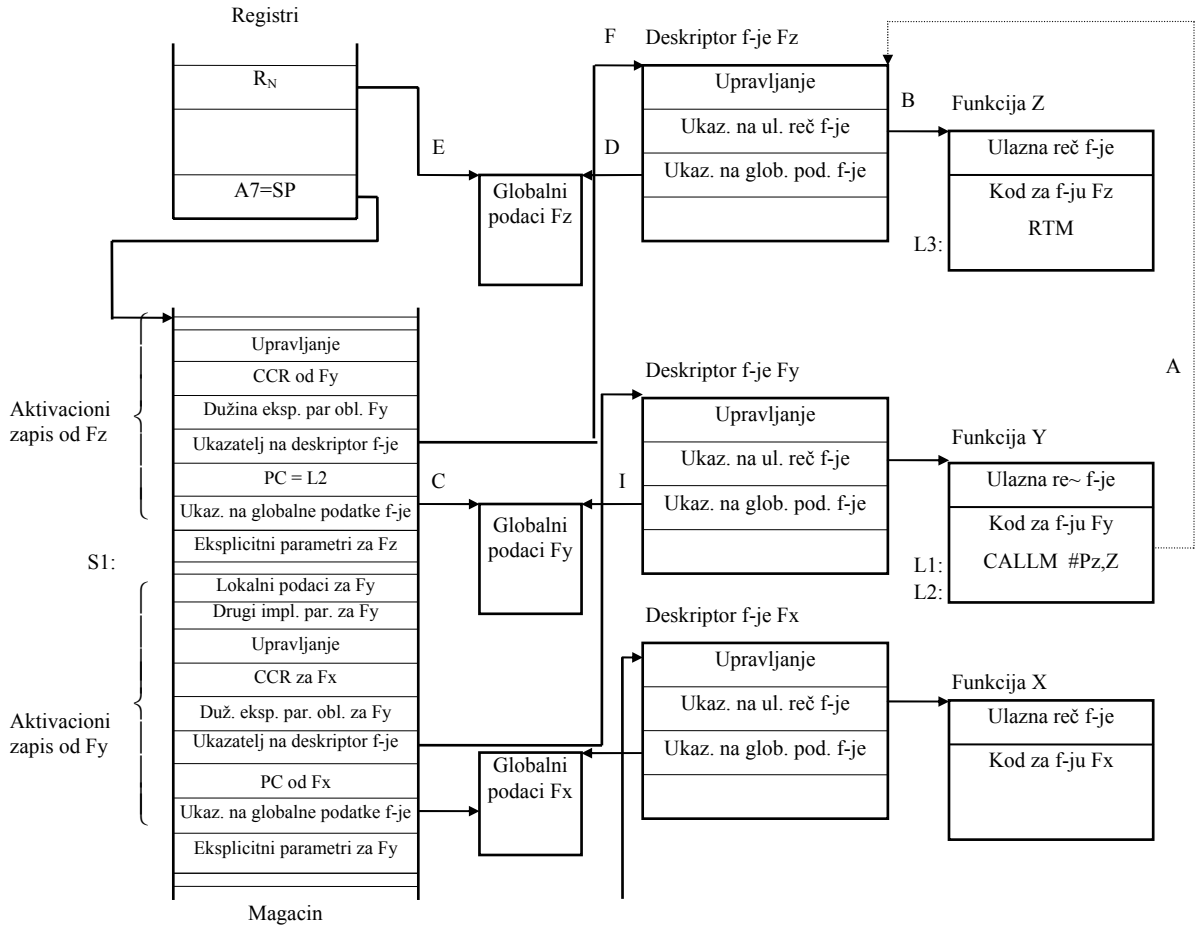
Sl. 8.14. Instrukcija CALLM mikroprocesora MC68020.

Neposredna vrednost "podatka" specificira dužinu (u bajtovima) oblasti koja je zauzeta od strane eksplicitnih parametara. Upravljačko polje deskriptora funkcije (slika 8.13d) ima marker koji specificira kada treba predati parametre preko magacina ili preko argument pokazivača. Kada se parametri predaju preko magacina, pozivajuća funkcija je odgovorna za predaju parametara u magacin.

Na slici 8.15 je prikazana situacija kod koje funkcija Fy, koja je bila pozvana od strane funkcije Fx, je upravo pozvala funkciju F2 u liniji L1 u kodu funkcije Fy.

Instrukcija poziva na liniji L1 "CALLM #Pz,Z" specificira da oblast zauzeća od strane eksplicitnih parametara bude "#Pz" a "Z", funkcija koja treba da se pozove. Z pokazuje na deskriptor funkcije F2 (isprekidana linija A na slici 8.15). Upravo pre CALLM instrukcije od Fy na L1, magacin koji sadrži aktivacioni zapis funkcije Fy i SP je ukazivao na lokaciju S1. Aktivnost CALLM instrukcije (usvajamo da su parametri predati preko magacina u oblasti označenoj sa "eksplicitni parametri za F2") je sledeća:

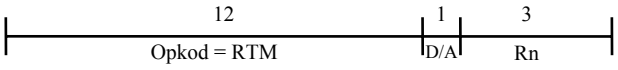
1. Postavi pokazivač globalne oblasti podataka: Efektivna adresa CALLM instrukcije specificira deskriptor funkcije Fz-a (isprekidana linija A na slici 8.15). Drugi ulaz deskriptora funkcije (slika 8.13d) je pokazivač ulazne reči funkcije koji pokazuje na ulaznu reč funkcije koja pripada kodu Fz(B). Ova ulazna reč (slika 8.13c) specificira da registar Rn treba da se koristi kao globalni pokazivač oblasti podataka. Registar se postavlja na sledeći način:
   a) Push Rn u magacin (C). Na slici 8.15 usvojeno je da se Rn koristi da ukaže na globalnu oblast pokazivača Fy.
   b) Punjenje Rn sa trećim ulazom deskriptora funkcije Fz-a, to je pokazivač globalne oblasti podataka za Fz(D). Rn sada pokazuje na globalnu oblast podataka Fz(F).
2. Sačuvaj PC: vrednost adrese povratka , koja je adresa linije L2 u kodu funkcije Fy, se smešta u magacin.
3. Adresa deskriptora funkcije Fz se smešta u magacin (F).
4. Dužina eksplicitne parametarske oblasti Fz se smešta. Ova dužina se predaje kao parametar od strane instrukcije CALLM (slika 8.14).
5. CCR (Condition Code Register) funkcije Fy se smešta.
6. Upravljačka informacija prvog ulaza deskriptora funkcije Fz (slika 8.13d) se smešta u magacin. Sa ovim se završava operacija instrukcije CALLM, nakon čega se izvršava kod Fz sve dok de ne naiđe na instrukciju RTM u liniji L3 koda funkcije Fz.



Sl. 8.15. Poziv funkcije kod mikroprocesora MC68020.

Izvršenjem instrukcije RTM u suštini se obavljaju obrnute aktivnosti od CALLM instrukcije (slika 8.16):

1. puni se CCR sa Fy.
2. puni se PC, koji sada ukazuje na liniju L2 koda funkcije Fy.
3. puni se Rn, specificiran kao operand RTM (slika 8.16) sa ciljem da ukaže na globalne podatke Fy.
4. poništava se oblast koja se koristila za eksplicitne parametre Fz. Dužina ove oblasti je bila sačuvana u magacinu.



Sintaksa: RTM Rn
Rn specificira registar koji se koristi da ukaže na oblast globalnih podataka
Efekat: Obnavlja zapamćeno stanje modula na koji se vraća

Sl. 8.16. Instrukcija RTM mikroprocesora MC68020.

Nakon ovoga završava se instrukcija RTM. SP ukazuje ponovo na labelu S1, a izvršenje instrukcija produžava sa linije L2 koja pripada Fy.