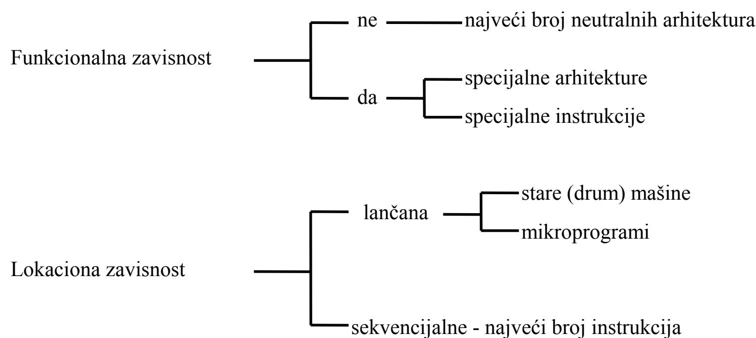
7. UPRAVLJANJE TOKOM PROGRAMA

7.1. Linearni redosled instrukcija

Program se sastoji od iskaza, koji se standardno izvršavaju u linearnom redosledu. Svaki iskaz se obično prevodi u jednu ili veći broj sukcesivnih mašinskih instrukcija, što ukazuje da postoji direktni odnos između lokacije pojedine instrukcije i one koja sledi posle nje.

Po analogiji sa HLL iskazima, i računarske instrukcije su u opštem slučaju funkcionalno nezavisne. To znači da se svaka instrukcija može posmatrati kao izdvojena sintaktička i semantička celina, različita od one koja joj prethodi, i one koja sledi. Na slici 7.1 je prikazan pregled arhitektura u odnosu na funkcionalnu i lokacionu zavisnost instrukcija.



Sl. 7.1. Funkcionalna i lokaciona zavisnost instrukcija.

7.1.1. Funkcionalna zavisnost

U opštem slučaju svaka instrukcija kod računara je nezavisna sintaktička i semantička celina. Pri ovome dve osobine pojednostavljaju implementaciju. Naime, imajući u vidu da postoji N različitih instrukcija (za datu arhitekturu) ukupno razlikujemo N*(N-1) parova instrukcija, N*(N-1)*(N-2) tripleta instrukcija, itd. Ako se imaju u vidu ponavljanja, tada dobijamo N^2 i N^3 kombinacija, respektivno. Kod nekih arhitektura postoje zavisne instrukcije, ali je broj zavisnih instrukcija ograničen - na primer, na parove ili triplete. Da bi ukazali na ove tipove instrukcija, analiziraćemo dve instrukcije koje poseduju funkcionalnu zavisnost od strane drugih instrukcija: to su instrukcije REPEAT i EXECUTION.

- Funkcija REPEAT instrukcije je da ukaže da jedan ili veći broj uzastopnih instrukcija mora da se ponovi specificirani broj puta. Instrukcija REPEAT specificira broj ponavljanja uzimajući u obzir i specijalni uslov završetka.
- EXECUTION instrukcija specificira izvršenje druge instrukcije koja je locirana na adresi specificiranoj operandom instrukcije EXECUTION. Tipičan primer je instrukcija CALLSUB.

Problem kod funkcionalne zavisnosti instrukcija je u tome što se svaka instrukcija ne može prihvatiti kao druga instrukcija para instrukcija. Na primer, ponavljanje REPEAT instrukcije se ne može prihvatiti jer se sa brojem koji ukazuje na uslov završetka mora manipulisati kako sa REPEAT instrukcijom internom hardveru (neophodno je ugraditi dodatni hardver) tako i sa statusom mašine koji mora biti obiman (po sadržaju). Drugi razlog zašto druga instrukcija u paru nije prihvatljiva ogleda se u tome što; koja će biti naredna instrukcija često puta nema smisla rezonovati. Na primer, razmatrajmo ponavljanje JUMP ili HALT instrukcije. Drugim rečima, potrebna je ugradnja hardvera koji će detektovati i signalizirati ovaj nevažeći par. Ova objašnjenja su osnovni razlog zašto se REPEAT i EXECUTION instrukcije ne sreću kod velikog broja arhitektura.

Instrukcije REPEAT i EXECUTE obično formiraju metainstrukcije.

7.1.2. Lokaciona nezavisnost

Lokacija naredne instrukcije zavisi od tekuće instrukcije na dva načina:

(1) **Ulančavanje** - ovo znači da svaka instrukcija specificira lokaciju naredne. Ovaj način je bio korišćen kod nekih starih računara koji su koristili bubanj kao proširenje glavne memorije veoma malog kapaciteta, sa ciljem da se dobije optimalno preklapanje između izvršenja tekuće instrukcije i lokacije na bubnju koja je ukazivala na narednu instrukciju. Kod ovakvog slučaja, naredna instrukcija se mogla čitati u trenutku kada je tekuća instrukcija završila sa izvršenjem. Ulančavanje se takođe koristi kod mikroprogramiranja, posebno kod instrukcija i dekodiranja operanada, jer se kodovi instrukcija i adresni načini rada koriste za aktiviranje različitih mikrorutina. Drugi, manje važan, razlog korišćenja ulančavanja je popunjenje praznina (gapova) koje su uslovljene neiskorišćenim (rezervnim) opkodovima ili adresnim načinima rada. Ovi "gapovi" mogu takođe biti uslovljeni određenim kombinacijama koje se ne mogu javiti. Na primer, kod višestrukog grananja (rezultat je aritmetičkog poređenja), od ukupno osam tipova grananja koristi se samo šest kombinacija ($<, \leq, =, \neq, >, \geq$). Kada je lokacija naredne instrukcije specificirana u okviru mikroinstrukcije, lokacije kombinacija koje se ne koriste, gapovi, se takođe mogu koristiti.

(2) **Sekvenciranje** - Ovo znači da naredna instrukcija koja treba da se izvrši ima implicitnu adresu o narednoj memorijskoj lokaciji. Ovim se eliminiše potreba za specifikacijom lokacije naredne instrukcije. Šta više, važno je da se kod brze implementacije lokacija naredne instrukcije može unapred predvideti (bez znanja tekuće instrukcije) tako da se naredna instrukcija može unapred pribaviti (prefetch). Ali pri tome dolazi do promene programske sekvence koja se javlja kada se izvršavaju instrukcije tipa BRANCH/JUMP čime se narušava sekvencijalni redosled izvršenja. Određivanje lokacije na osnovu lančanja se u opštem slučaju ne primenjuje na konvencionalnom mašinskom nivou, jer zahteva dodatni operand. Sekvencijalni metod je stoga poželjniji, i pored toga što zahteva da se arhitektura proširi sa BRANCH/JUMP instrukcijama. Primer sekvencijalne lokacione zavisnosti je instrukcija zakašnjeno grananje (delayed branch). To je (uslonva) BRANCH instrukcija kod koje se grananje obavlja nakon izvršenja naredne sekvencijalne instrukcije. Razlog ovome je da je, u opštem slučaju, naredna instrukcija već spremna za izvršenje, tako da je sa te tačke gledišta bolje da se ta instrukcija takođe završi. Ovo rezultira poboljšanju performansi jer se inače diskontinuitet kod jedinice koja vrši pribavljanje instrukcija narušava zbog instrukcije grananja. Instrukcija "delayed BRANCH", uprkos svojoj pogodnosti, se ne koristi tako često zbog dodatne složenosti kod izvođenja kompilatora (potrebna je dodatnu instrukciju alocirati i pozicionirati posle BRANCH). Najveći broj RISC arhitektura koristi ovu tehniku.

7.2. Nelinearni redosled izvršenja

Kod programiranja, redosled instrukcija nije uvek linearan. Kod najvećeg broja slučajeva, moraju se doneti odluke na osnovu kojih se ukazuje koji će se put u programu, u daljem toku, izvršiti. Za izbor jednog od mogućih puteva koriste se uslovne operacije.

7.2.1. Operacije

Donošenje odluka u programu se obično izvodi promenom programskog toka koji se zasniva na nekom prethodno generisanom rezultatu. Ovo ukazuje da postoji potreba za instrukcijama pomoću kojih se može menjati linearni redosled izvršenja instrukcija na uslovni način, a ponekad i na bezuslovni. Ovim instrukcijama se obezbeđuje željena fleksibilnost i generalnost programskih jezika i skupova naredbi. Neki primeri iz HLL-a, kao što je Pascal, su: **if** ... **then** ... **else**, **case** i **goto** iskazi. Ove konstrukcije programskog jezika zahtevaju podršku od strane arhitekture računara. Ovu podršku čini veći broj testova i jedna ili veći broj operacija (JUMP ili BRANCH) koje se zasnivaju na rezultatima ovih testova. U ovom slučaju, test se može smatrati evaluacijom (procenom) podataka sa ciljem da se on preslika na skup uzajamno isključivih kriterijuma odlučivanja. Obično su to uslovni kodovi (markeri). Shodno tome, test je preslikavanje ulaznog domena, koga čine sve moguće ulazne vrednosti, na izlazni domen koga čini vrednost za svaku klasu ulaznih vrednosti.

Odluke se najčešće zasnivaju na poređenju brojeva (vrednosti), Najjednostavniji slučajevi su poređenja vrednosti sa nulom ili poređenje dve vrednosti. Izazovniji slučaj je onaj kada treba odrediti da li se vrednost nalazi unutar opsega vrednosti.

Poređenje dve vrednosti

Veoma često se obavlja poređenje dve vrednosti. U principu, poređenje čini upoređenje dve operandske vrednosti koje se ne menjaju u toku procesa poređenja. Rezultat je jedan od sledeća tri uslova: $<$, $=$ ili $>$. Često, korisnik želi da raspolaže i sa složenijim a ne samo sa prostim izborom, koji rezultira u istovremenom izboru sa sledećim mogućnostima:

$$A < B \text{ ili } A \geq B$$

A ≤ B ili A > B
A = B ili A ≠ B

Moguća implementacija je preko instrukcije SUBTRACT. Na žalost, u ovom slučaju javlja se sporedni efekat jer se razlika smešta na mesto odredišta, što dovodi do neželjene destrukcije tog operanda. Iz ovog razloga kod najvećeg broja arhitektura postoji instrukcija COMPARE, kod koje se ne javlja takav sporedni efekat, jer se razlika pravi samo sa ciljem da se generiše izvedeni rezultat.

Veoma često je neophodno uporediti operand sa nulom. Operand nula je, kod velikog broja arhitektura, implicitno specificiran opkodom u slučaju kada se koristi instrukcija TEST.

Ređe se podržavaju HLL funkcije **max** i **min**. One su specijalni slučaj poređenja dve operandske vrednosti, kombinovane sa instrukcijom koja se direktno izvršava nakon toga Na primer, funkcija

A:=**max**(A,B)

može da se obavi od strane sekvence mašinskih instrukcija mikroprocesora MC68020 na sledeći način:

```
MOVE.L    A,D0     ; smesti A u D0
CMP.L     B,D0     ; izračunaj A-B
BGE       DALJE    ; if A>B then DALJE
MOVE.L    B,A      ; else B>A, pa A:=B
```

Ova sekvenca se može obaviti sledećom novom instrukcijom

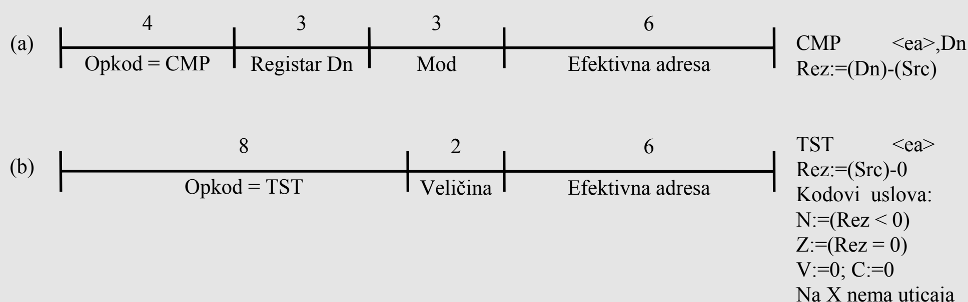
```
MAX.L     B,A
```

Uvođenje novih instrukcija je korisno u sledećim slučajevima:

- Kada je frekvencija korišćenja dovoljno velika ili kada ne postoji drugi metod za implementaciju. Ovo je razlog što se primitivne operacije obavljaju pomoću instrukcija kao što su WAIT, TRAP i RESET.
- Kada je potrebno uštedeti memorijski prostor i vreme izvršenja u odnosu na programsku implementaciju. Ovo vodi poboljšanju vremena izvršenja instrukcije zbog toga što se eliminiše BRANCH instrukcija koja može imati negativne posledice na protočnu obradu.
- Kada se ugradi u postojeće arhitekture pa ne dolazi do značajnih promena u postojećim formatima instrukcija.

Primer 7.1:

Kod MC68020 postoje dve instrukcije koje se koriste za poređenje vrednosti: CMP i TST. Instrukcijom CMP kompariraju se operandi opšteg tipa sa registrom, dok TST komparira opšti operand sa nulom. Instrukcijom CMP A,B izračunava se B-A. Rezultat CMP i TST instrukcija ima uticaja na markere registra CCR. Dinamika korišćenja CMP i TST kod MC68020 je 7,86% i 3,09%, respektivno (MacGregor 1985). Ukupni procenat od 10,95% pokazuje relativnu važnost ovih instrukcija. Na slici 7.2 je prikazan format CMP i TST instrukcije kod MC68020.



Sl. 7.2. Format instruckija CMP i TST mikroprocesora MC68020.

Poređenje opsega

Poređenje opsega predstavlja rangiranje elemenata (tj. podataka) u odnosu na dva elementa. Može se primeniti i za N elemenata. Pomenuta operacija se sastoji u određivanju korektne pozicije elemenata (tj. podataka), usvajajući da se vrši sortiranje N elemenata po nekom ključu. Najveći broj arhitektura poseduje specijalne instrukcije za ovo poređenje. Operacija se često obavlja repetitivnim rangiranjem dva elementa.

Kod ovog poređenja, neophodno je klasifikovati objekat tako da postoji određena relacija sa drugim objektima; na primer:

'A' ≤ Char ≤ 'Z'	alfabetski znaci
'0' ≤ Char ≤ '9'	numerički znaci
donja_granica ≤ indeks gornja_granica	indeks polje

Poređenje A ≤ X ≤ B može da generiše (u zavisnosti od rezultata):

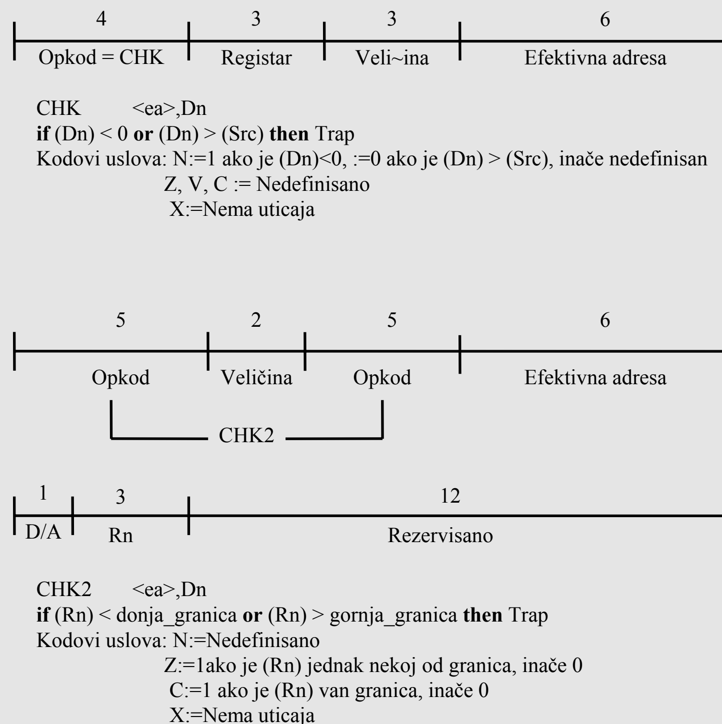
- dva moguća odgovora : $A \le X \le B$; $X<B$ ili $X>B$
- tri moguća odgovora : $A \le X \le B$; $X<A$; $X>B$
- pet mogućih odgovora: $A<X<B$; $X=A$;$X=B$;$X<A$;$X>B$

Poređenje opsega je važna funkcija kod adresiranja polja. U ovom slučaju važne su samo dve funkcije, u opsegu i van opsega, pa shodno tome rezultat ima uticaj na uslovne kodove (markere), koji se u daljem programskom toku koriste (testiraju) od strane instrukcija koje obavljaju uslovno grananje.

Primer 7.2:

Na slici 7.3 je prikazan format instrukcija CHK i CHK2 procesora MC68020 pomoću kojih se obavlja poređenje opsega. Operand koji se poredi specificira se u registarskom polju; kod instrukcije CHK on se uvek nalazi u registru za podatke, a kod CHK2 D/A poljem se specificira da li se on nalazi u registru za podatke ili u adresnom registru.

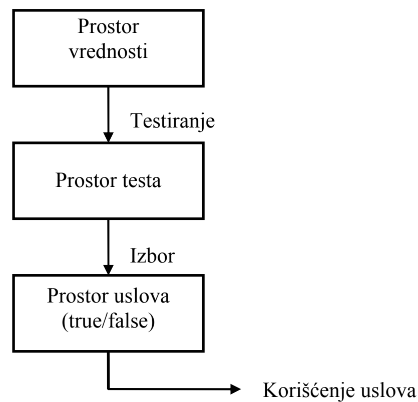
Kod CHK instrukcije implicitno se specificira donja granica na nulu, dok je gornja granica specificirana efektivnom adresom. Efektivna adresa kod CHK2 specificira adresu nižeg-višeg para granice. Instrukcija postavlja uslovne markere, koji odražavaju rezultat poređenja (slika 7.3), i izazivaju trap (sinhroni prekid programa u odnosu na njegovo izvršenje) kada je operand van opsega.



Sl. 7.3. Format instrukcija CHK i CHK2 mikroprocesora MC68020.

7.2.2. Uslovne operacije

Uslovne operacije se mogu koristiti za promenu programskog toka. Njih čine tri dela: *testiranje*, *izbor* i *korišćenje uslova* (slika 7.4).



Sl. 7.4. Delovi uslovnih operacija/

Testiranje

Predstavlja generisanje izvedenog rezultata (koji formira test prostor - test space) od operacija koje se obavljaju nad vrednostima iz prostora vrednosti (value space). Na primer, uslovni markeri N, Z, V i C zauzimaju test prostor od 16 elemenata.

Test funkcija se ponekad obavlja implicitno; naime, kada se uslovni markeri kao rezultat izvedenog rezultata postave ili obrišu od strane druge operacije - na primer, instrukcijom ADD. U ostalim slučajevima se mora izvesti eksplicitno, specijalnom test instrukcijom, kao što je CMP ili TST.

Kod procesora BS32000 izvedeni rezultat je podeljen na dva dela:

(1) **Implicitna grupa** koju čine sledeći izvedeni rezultati koji su posledica operacije: bit prenosa (C) i bit premašaja (F).
(2) **Eksplicitna grupa** koju čine izvedeni rezultati koji su posledica rezultata operacije: Z bit (rezultat jednak nuli), N bit (rezultat u označenoj prezentaciji brojeva je negativan) i L (operand, posmatran kao neoznačen broj, je manji od drugog operanda).

Da bi se generisali eksplicitno izvedeni rezultati, potrebne su posebne **compare** i **test** instrukcije, dok se implicitni uslovni markeri postavljaju ili brišu iz očitih razloga.

Izbor

Kao što se vidi sa slike 7.4, operacijom selekcije (izbora) vrši se preslikavanje test prostora u prostor uslova (condition space). On proverava da li test prostor sadrži elemente specificirane operandom za izbor, pa shodno tome izlaz može biti "true" (istinit) ili "false" (nije istinit). Na primer, izbor operanda GE (Greater or Equal) kod instrukcije grananja proverava da li test prostor sadrži elemenat GE.

Obično, operand izbora, zbog ograničenog prostora opkoda, ne može da specificira svaki elemenat test prostora. Kod MC68020, jedinstveni podskup, kao što je GE, moguće je izabrati od ukupnog test prostora koga čini 16 elemenata. Zbog velikog broja podskupova, na primer podskup GE i "not C" - izbor zahtevanog podskupa (u ovom slučaju GE i "not C") se ne može izvesti pomoću jedne instrukcije. Samo najčešće korišćeni podskupovi iz ukupnog test prostora se mogu birati jednom instrukcijom (Tab. 7.1). Ostali skupovi se mogu birati izvršenjem jedne ili većeg broja instrukcija, naravno njihovom pogodnom kombinacijom. Na primer, test "not equal to zero" kod MC68020 se izvodi jednom instrukcijom (uslov NE), dok se test "greater or equal to zero" i "not carry" mora se izvesti sekvencom od dve instrukcije (uslov GE i CC).

Tab. 7.1. Uslovni testovi kod MC68020.

Mnemonik	Naziv	Formula	Uslov
T	tačno		1
F	netačno		0
Jednobitni kod uslova			
CC	prenos obrisan		~C
CS	prenos postavljen		C
NE	nejednako		~Z
EQ	jednako		Z
VC	prekoračenje obrisano		~V
VS	prekoračenje postavljeno		V
PL	plus		~N
MI	minus		N
[1]Neoznačen			
HL	više		~C~Z
LS	manje ili isto		C+Z
HS	više ili isto		~C
LO	manje		C
[2]Označen			
GE	veći ili jednak	N EOR ~V	NV+~N~V
LT	manji od	N EOR V	N~V+~NV
GT	veći od	(N EOR ~V) AND ~Z	NV~Z+~N~V~Z
LE	manji ili jednak	(N EOR V) OR Z	Z+N~V+~NV

[1] Aritmetika sa neoznačenim brojevima: **Manje**: rezultat operacije upoređenja "je manji od" ako je bit prenosa postavljen (C=1). **Više ili isto**: Ovo je inverzija od **Manje** (C=0). **Manje ili isto**: Istiniti je ako je **Manje** istinito (C=1) ili ako je rezultat nula (Z=1). **Više**: predstavlja inverziju od **Manje ili isto** (~(C+Z)=~C~Z).

[2] Aritmetika sa neoznačenim brojevima: Označeni rezultat (na primer u dvojičnom komplementu) je ≥0 kod sledeća dva slučaja: (a) ako se bit znaka briše bez detekcije premašaja (N=0 i V=0); (b) ako, da tako kažemo, sabiranje dva pozitivna broja generiše rezultat koji uslovljava da se bit znaka promeni, tako da se bit premašaja postavi na (N=1 i V=1). Formula za **Veće ili jednako** postaje N EOR ~V. Ostale formule mogu se izvesti na sličan način kao kod aritmetike sa neoznačenim brojevima.

Korišćenje uslova

Ovo je deo koji koristi Boolean rezultat prostora uslova. Obično, uslov prati deo (adresu skoka) koji se obavlja operacijom grananja, u slučaju kada se želi ostvariti promena toka izvršenja programa. Kod HLL-ova, Boolean rezultat izbora (selekcije) može se kao vrednost dodeliti operandu, kao što je, na primer, B:=(X≤Y).

7.3. Arhitekturna podrška kod uslovnih operacija

Uslovne operacije se mogu obaviti pomoću jedne, dve ili tri instrukcije, koje su na određeni način međusobno povezane. Arhitekturna podrška uslovnim operacijama se pokriva u zavisnosti od sva tri dela sa slike 7.4. Specifikacija ovih delova (testiranje, izbor, korišćenje uslova) može se izvesti, kao što je prikazano u Tabeli 7.2, na četiri načina, a da bi se to sprovelo potrebne su jedna, dve ili tri instrukcije.

Tab. 7.2. Podrška uslovnim operacijama.

Slučaj	Testiranje	Izbor	Korišćenje uslova
A	1	2	3
B	1	2	2
C	1	1	2
D	1	1	1

Kada uslovnu operaciju čine više od jedne instrukcije (slučajevi A, B i C), neophodno je vršiti prenos informacija između instrukcija. Obično se za prenos informacija koriste uslovni markeri (condition code). Kod alternative A (Tabela 7.2) zahtevaju se tri posebne instrukcije radi izvršenja uslovnih operacija (instrukcija 1 za testiranje, instrukcija 2 za izbor uslova, a instrukcija 3 za korišćenje uslova). U svetlu čestog korišćenja uslovnih operacija, ovo je veoma neefikasno i zbog toga se ova alternativa ne koristi kod komercijalno raspoloživih mašina.

Alternativa B prikazuje klasično rešenje registarski orijentisanih arhitektura. Uslovni markeri se koriste za promenu informacije između dve instrukcije. Prva instrukcija obavlja test, dok druga instrukcija vrši izbor i koristi generisani uslov.

Kod mikroprocesora MC68020, test se obavlja pomoću instrukcija tipa ADD, MUL, CMP i TST, dok se delovi koji se odnose na izbor i korišćenje uslova obavljaju od strane instrukcija Bcc ili Scc. Glavni razlog ovakvog izvođenja ogleda se u tome što se deo koji se odnosi na test često izvodi implicitno (izvedeni rezultati se generišu od strane ADD, MUL i drugih instrukcija), i što kod registarski orijentisanih mašina test instrukcije koriste instrukcioni format aritmetičkih operacija (na primer CMP ima isti format kao ADD i SUB), a to ne ostavlja dovoljno prostora u formatu instrukcije za delove koji se odnose na izbor i korišćenje uslova. Alternativa C se koristi kod magacinski orijentisanih mašina i kod softverski orijentisanih arhitektura (Pascal P-kod). Kod ovih arhitektura ne postoje uslovni markeri, ali postoje specijalne instrukcije pomoću kojih se realizuje deo testa i deo izbora. Ovo se izvodi jednom instrukcijom, pošto ove instrukcije ne specificiraju bilo kakve operande (operandi se implicitno specificiraju preko magacina). Shodno tome postoji dovoljno prostora za specifikaciju dela koji se odnosi na izbor u delu koji propada testu. Lokacija rezultata dela, koji pripada izboru, se takođe određuje implicitno - to je vrh magacina.

Primer 7.3:

Instrukcija GRTR (deo testiranja i izbora) a pripada skupu naredbi mašine B6700 se izvršava na sledeći način:

if (M[(SP)])>(M[(SP)-1]) **then**
(M[(SP)-1]:=True
else
(M[(SP)-1]:=False;
SP:=(SP)-1; (*dekrementira se pokazivač magacina*)

Instrukcije koje koriste deo korišćenja uslova su:

BRTR/DBTR	Branch if True
BRFL/DBFL	Branch if False
BRUN/BBUN	Branch Unconditionally

Grupa BR kod B6700 u samoj instrukciji specificira adresu grananja (kao relativni razmeštaj), dok DB grupa specificira implicitno adresu grananja na vrh magacina. Operacije BRTR i DBFL imaju sledeće značenje:

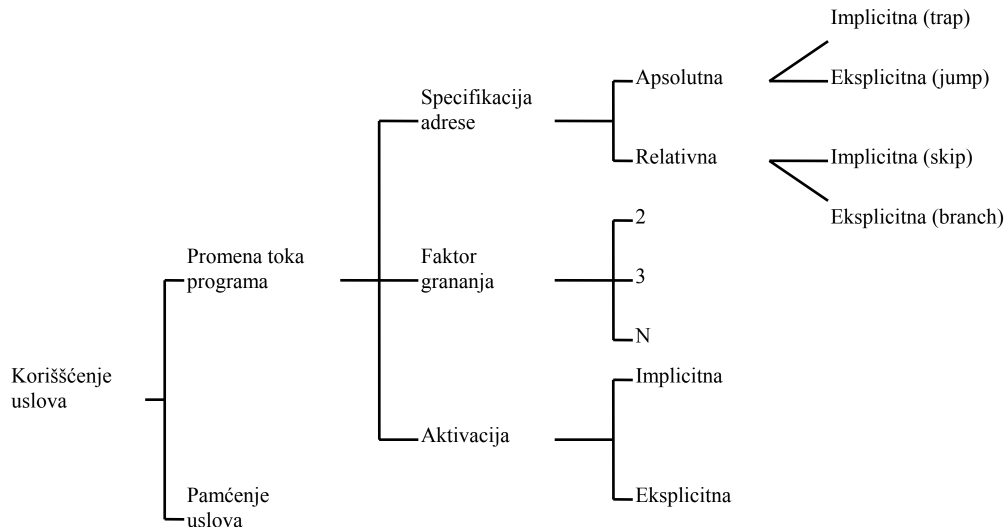
- BRTR omogućava relativno grananje na odredište specificirano u Displ polju instrukcije
  if (M[(SP)]) = True **then** PC:=(PC)+Displ;
  SP:=(SP)-1;
- DBFL obezbeđuje odredište kod izračunatog grananja. Odredište se određuje, kao apsolutna adresa, iz magacina:
  if (M[(SP)]) = False **then** PC:=(M[(SP)-1]);
  SP:=(SP)-2;

Kod alternative D, da bi se obavila uslovna operacija, potrebna je samo jedna instrukcija. Korišćenje uslovnih markera je redundantno kod ove arhitekture. Tipičan primer registarsko orijentisane arhitekture koja kombinuje sva tri dela u jednoj instrukciji je MIPS procesor. Kod ovog procesora nema uslovnih markera, a koristi se eksplicitna compare-and-branch instrukcija za četiri operanda (uslov, dva izvorna operanda koji se porede, i razmeštaj za mesto grananja), gde se uslov specificira kao deo instrukcije.

Kada se javi prekid (interrupt) neophodno je zapamtiti uslovne markere ili izabrani uslov, jer oni pripadaju statusu prekinutog procesa. Alternativa B, shodno tome, zahteva prostor u statusnom registru (SR) za pamćenje uslovnih markera (bitovi 0-4 SR registra procesora MC68020). Kod alternative C, rezultat selektovane operacije se pamti u magacin, tako da se ne preduzimaju posebne mere kada se javi prekid. Kod alternative D, ne postoje uslovni markeri tako da nema potrebe da se bilo šta pamti.

7.4. Korišćenje uslova

Uslov koga generiše izborni (selekcioni) deo može se koristiti na dva različita načina: da se pamti uslov, ili da se promeni programski tok (slika 7.5).



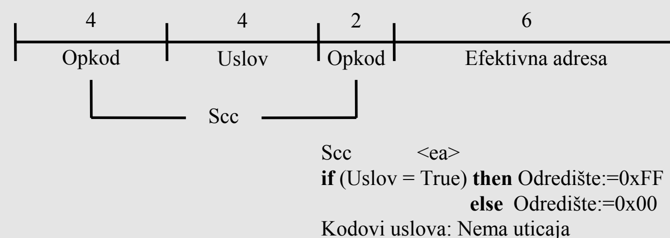
Sl. 7.5. Korišćenje uslova.

7.4.1. Pamćenje uslova

Pamćenje uslova selekcionog dela je važno kod HLL gde se ovaj rezultat dodeljuje Boolean promenljivoj. Analizirajmo, na primer, iskaz B:=(X≤Y). Scc instrukcijom u primeru koji sledi na korektan način se podržavaju HLL iskazi.

Primer 7.4:

Kod MC68020 na elegantan način se podržava HLL iskaz B:=(X≤Y) koristeći Scc (Set Acording to Condition) instrukciju (slika 7.6).



Scc <ea>
if (Uslov = True) **then** Odredište:=0xFF
else Odredište:=0x00
Kodovi uslova: Nema uticaja

Sl. 7.6. Format Scc instrukcije mikroporcesora MC68020.

Ova instrukcija postavlja sve bitove odredišnog bajta operanda na "1" ako je uslov istinit, i na "0" ako je uslov pogrešan. Uslovi mogu biti bilo koji od onih definisanih u Tabeli 7.1.

7.4.2. Promena redosleda instrukcija (programskog toka)

Korišćenje uslova da se promeni redosled instrukcija (tok programa) uključuje sledeća tri aspekta (vidi sliku 7.5):

1. **specifikacija adrese**
2. **faktor grananja** - broj uslova koji se koriste (dvo-, tro- i N-struki skokovi pri čemu se dvostruki skok najčešće implementira).
3. **aktiviranje** (implicitno ili eksplicitno) koristi deo uslova.

Specifikacija adrese

Specifikacija adrese se, shodno slici 7.5, može izvesti u zavisnosti od apsolutne ili relativne adrese koja se može specifirati implicitno ili eksplicitno.

Apsolutna specifikacija adrese je jednostavna, i pored toga što zahteva punu adresu, a ne obezbeđuje relokatibilni kod.

Apsolutna eksplicitna specifikacija adrese se javlja kada je apsolutna adresa specificirana kao operand instrukcije, kao kod JUMP instrukcije. Ovo nije uobičajeni način za implementaciju BRANCH instrukcija, jer se ne obezbeđuje relokatibilnost koda (programa).

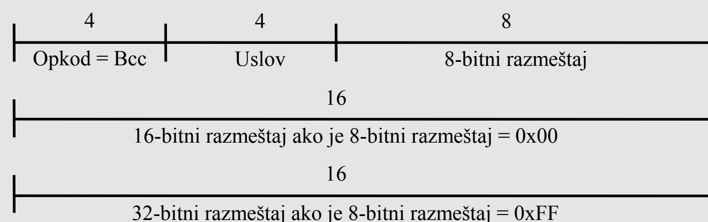
Apsolutna implicitna specifikacija adrese se javlja kada je apsolutna adresa specificirana pre njenog korišćenja. Zbog toga, u trenutku korišćenja, nije potrebna adresna specifikacija jer se koristi unapred specificirana apsolutna adresa. Trapovi i prekidi obično koriste apsolutne implicitne adrese, koje omogućavaju automatske skokove na rutine koje su smeštene na fiksnim lokacijama i poznate su kao rituine za obradu izuzetaka (exception handlers). Ovi skokovi su u najvećem broju slučajeva posledica: premašaja, deljenja nulom, ili spoljnih prekidia i dr.

Relativna specifikacija adrese predstavlja oblik baznog adresiranja. Ipak, kako se programski brojač (PC) obično koristi za specifikaciju bazne adrese, adresiranje se ipak zove *PC-relativno adresiranje*. Relativno adresiranje je obično poželjnije u odnosu na apsolutno adresiranje, jer automatski generiše relokatibilni kod, a za specifikaciju relativne adrese je obično potrebno nekoliko bitova.

Relativna, eksplicitna specifikacija adrese je ona kod koje instrukcijom specificira relativna adresa. Ova specifikacija se koristi od strane najvećeg broja arhitektura u obliku BRANCH instrukcije. Njihova prednost je u tome što su bit efikasne i obezbeđuju relokatibilni kod (slika 7.5).

Primer 7.5:

Instrukcija Bcc (Branch According to Condition) kod MC68020 (slika 7.7) je dvooperandska instrukcija: prvi operand specificira uslov koji se bira, a drugi operand specificira relativni razmeštaj kao neposredni deo instrukcije.



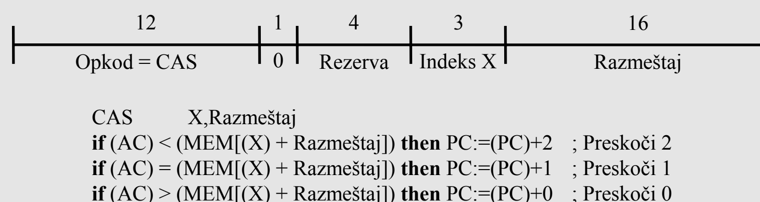
Sl. 7.7. Format instrukcije Bcc mikroprocesora MC68020.

BRANCH instrukcije su veoma često korišćene upravljačke instrukcije - ranije smo ukazivali da one čine 23,4% svih instrukcija koje se izvršavaju. Najveći broj arhitektura, uključujući i MC68020, ima specijalni oblik ove instrukcije, kao što je relativni veliki razmeštaj koji je deo instrukcije (prva reč instrukcije).

Relativna, implicitna specifikacija adrese znači da je relativni razmeštaj implicitno specificiran. Ovo je obično fiksni razmeštaj čija je vrednost "1". Koristi se, na primer, kada se delovi za test, izbor i korišćenje uslova moraju izvesti jednom instrukcijom (alternativa D iz Tabele 7.1), jer se često ne određuje operand za eksplicitnu specifikaciju adrese koji pripada delu za korišćenje uslova. Ovakav princip je bio primenjivan kod nekih starih arhitektura, kod kojih su sve instrukcije iste dužine, a relativna lokacija je bila jednu lokaciju dalje u odnosu na narednu instrukciju (tj. PC se povećava za 1 ako je uslov istinit). Efekat ovoga je da instrukcija u narednoj poziciji (koja mora biti dužine jedne reči) može se preskočiti uslovno. Zbog toga se ove instrukcije zovu SKIP.

Primer 7.6:

Interesantni primer relativne, implicitne specifikacije adrese srećemo kod IBM/704 CAS (Compare Accumulator with Storage) instrukcije (slika 7.8).




```
CAS    X,Razmeštaj
if (AC) < (MEM[(X) + Razmeštaj]) then PC:=(PC)+2   ; Preskoči 2
if (AC) = (MEM[(X) + Razmeštaj]) then PC:=(PC)+1   ; Preskoči 1
if (AC) > (MEM[(X) + Razmeštaj]) then PC:=(PC)+0   ; Preskoči 0
```

Sl. 7.8. Format CAS instrukcije procesora IBM/704.

Kod IBM/704 arhitekture koriste se instrukcije fiksnih dužina od po 36 bitova, koje omogućavaju da se CAS instrukcija izvrši kao trostruki SKIP. Ovaj primer je interesantan jer na jasan način pokazuje kako je ova arhitektura imala uticaja na definiciju programskog jezika FORTRAN, i da je FORTRAN, kao jezik, više mašinski orijentisan nego što bi trebalo da bude. CAS instrukcija se koristi kao polazna tačka za definiciju FORTRAN iskaza IF (A-B) 10,20,30, što znači **if** A<B **then goto** 10 **else if** A=B **then goto** 20 **else goto** 30, gde su 10, 20, 30 labele.

Faktor grananja

Veliki broj mašina podržava instrukcije koje nude dva alternativna načina da se produži sa programskim izvršenjem, bilo sekvencijalno ili pomoću grananja (jumping). Drugim rečima, faktor grananja je dva. Prethodnim primerom je opisana instrukcija grananja na tri različita mesta. Faktor grananja N je važan kod izvršenja iskaza **case**. Kod ovog tipa iskaza, izračunata vrednost (**case** selektor) određuje koji put treba izabrati. Kako kod najvećeg broja mašina ne postoji specijalna instrukcija za ovo, **case** izraz se obično izvršava koristeći se bezuslovnom instrukcijom "jump", sa adresom koja se izračunava (koristi se bazno adresiranje).

Primer 7.7:

Kod VAX-11 postoji specijalna CASE instrukcija. Tip podatka (bajt, reč ili duga reč) selektora, baza i granica operanda se određuje od strane opkoda, dok je tip podatka operanda koji ukazuje na razmeštaj uvek reč.

```
CASE selector,base,limit,displ[0] ... displ[limit]
Tmp:=selector-base;
PC:=(PC)+if Tmp≤limit
        then (disp[Tmp])
        else (2+2*limit);
```

Privremena vrednost Tmp se izračunava oduzimanjem baznog od selektorskog operanda. Vrednost Tmp se zatim poredi sa operandom "limit": ako je on manji ili jednak sa njim, dodaje se razmeštaj PC-u, koji se bira iz liste razmeštaja na osnovu Tmp vrednosti; ako je Tmp veći od operanda "limit", tada se obavlja skok preko liste razmeštaja. Na primer, FORTRAN iskaz **goto** (10,20,30),I se prevodi u

```
CASE  I,#1,#3
WORD  10      ; If I=1
WORD  20      ; If I=2
WORD  30      ; If I=3
              ; If I<1 ili I>3 tada se javlja greška
```

Aktiviranje i uslovne instrukcije

Aktiviranje dela korišćenja uslova se može izvesti na dva načina:

(1) **Eksplicitno aktiviranje** - specifične instrukcije (na primer, instrukcije uslovnog grananja) se koriste za određivanje dela korišćenja uslova. Ovo se uobičajeno izvodi standardno.

(2) **Implicitno aktiviranje** - upotreba dela **korišćenje_uslova** se javlja kao sporedni efekat drugih eksplicitno specificiranih instrukcija. Implicitno aktiviranje se koristi kada treba da se obavi funkcija "nadgledanja" sa ciljem da se detektuju izuzetni uslovi (kao što su na primer, premašaj, podbačaj ili deljenje nulom). Implicitno aktiviranje se može posmatrati kao uslovna operacija koja je aktivna (dozvoljena) u sklopu analize velikog broja instrukcija (na primer, kod situacije deljenje nulom analiza se odnosi na ceo program).

Instrukcija se može učiniti uslovnom ako se specificira da se proverava određeni uslov pre nego što se instrukcija izvrši. Ako uslov nije zadovoljen, instrukcija se ne izvršava. Ove instrukcije se zovu uslovne instrukcije. Primer arhitekture sa uslovnim instrukcijama je procesor ARM (Acorn RISC Machine), kod koje svaka instrukcija ima 4-bitno polje koje ukazuje na uslove (condition field). Ako je uslov istinit, instrukcija se izvršava, ako ne preskače se.

Drugi oblik uslovne instrukcije je "compare-and-test" instrukcija, kod koje rezultat (rezultujući uslovni kod) zavisi od uslovnog koda tekuće instrukcije. Ove instrukcije se zovu *instrukcije uslovnog testa*, i veoma su važne kod aritmetike povećane prezicnosti. Rezultat test instrukcije kada je operand obične preciznosti je <0, =0, ili >0; za operande povećane preciznosti, uslovni kodovi treba da reflektuju vrednost potpunih operanada u povećanoj preciznosti.

Druge instrukcije kod kojih postoje uslovni sporedni efekti su aritmetičke instrukcije, kao što su "add with carry" i "subtract with carry". Rezultujući sporedni efekti moraju odražavati status tekućih metacifara. Ovo znači da se formula za Z bit proširuje zahtevom da rezultujući Z bit može jedino biti postavljen u stanje "istinit", ako su

tekuća metacifra i sve prethodne metacifre jednake nuli. Na ovaj način marker Z se briše ako rezultat nije nula. U Tabeli 7.3 je prikazana implementacija HLL iskaza

if (A>B) **and** (C<D) **then** S1;
B:=(X≤Y);

za neke arhitekture i alternative B, C i D iz Tabele 7.2.

Tab. 7.3.

Iskaz	Alternativa A (MC68020)	Alternativa C (B6700)	Alternativa D (MIPS)
(10.1)	CMP.L B,A BLE end CMP.L D,C BGE end S1 end:	PUSH A PUSH B GRTR PUSH C PUSH D LESS AND BRFL end S1 end:	BLE A,B,end BGE C,D,end S1 end:
(10.2)	CMP.L Y,X SLE B	PUSH X PUSH Y LEQ POP B	SLE X,Y,B

7.5. Iteracija

Normalni mehanizam za realizaciju operacija nad elementima podataka koji pripadaju strukturi (na primer, svim elementima vektora) zove se *iteracija* i čini ga ponavljanje većeg broja instrukcija nad svakim elementom podataka. Najčešće korišćeni mehanizmi kod HLL-ova su **for**, **while** i **repeat** iskazi.

7.5.1. For iskaz

Opšti oblik **for** iskaza je sledeći:

for <upravljačka promenljiva>:=<početna vrednost> **step** <vrednost koraka>
until <krajnja vrednost>
do <iskaz>

Kod Pascala **for** iskaz je manje opšti jer se upravljačka promenljiva (control value) može menjati samo za +1 ili -1, umesto za vrednost koraka (step value) koja može primiti proizvoljnu vrednost, kako pozitivnu, tako i negativnu.

Neki važni implementacioni aspekti **for** iskaza su:

1. Provera inicijalne vrednosti: Provera inicijalne vrednosti zajedno sa vrednošću koraka mora se izvoditi sa ciljem da se odredi kada je čak i prva iteracija petlje dozvoljena. Na primer, kod iskaza

   for I:=5 **step**+1 **until** 3

   petlja se neće izvršiti.
2. Stop kriterijum; petlja se može zaustaviti na dva načina:
   - Kada upravljačka promenljiva primi vrednost koja je van intervala određenog početnom i krajnjom vrednošću (FORTRAN i ALGOL 60 koriste ovaj kriterijum).
   - Kada je upravljačka promenljiva jednaka krajnjoj vrednosti (na primer, kod Pascala se zahteva ovo, jer se u ovom slučaju iteracije u podopsegu prekidaju sa vrednošću upravljačke promenljive koja je unutar opsega i zbog toga može biti istog podopsežnog tipa).

Kao i kod prethodnih alternativa, kada se HLL-ovi analiziraju, nije moguće raspolagati jedinstvenom mašinskom instrukcijom koja se može koristiti u svim slučajevima. Kod MC68020 **for** iskaz se podržava pomoću DBcc (test condition, decrement and branch) instrukcije. Kod VAX-11 pomoću tri instrukcije se podržava **for** iskaz.

Primer 7.8:

Instrukcija ACB (Add, Compare and Branch) kod VAX-11 mašine ima četiri operanda i funkcije koje su sledećeg oblika:

ACB limit, add, index, displacement
index:=index + add
if {(add ≥ 0) **and** (index ≤ limit)} **or** {(add < 0) **and** (index ≥ limit)} **then**
PC:=(PC) + SEXT(displacement) ; SEXT = Sign-extended (Znakovno proširen)

Instrukcija podržava opšti oblik **for** iskaza. Kako je veličina koraka promenljiva (kako po vrednosti tako i po znaku) ona se može menjati unutar petlje, kako se to specificira kod ALGOL-a. Ostale dve instrukcija AOB (Add One and Branch) i SOB (Subtract One and Branch) su pojednostavljeni slučajevi koji koriste veličinu koraka +1 i -1, respektivno, a SOB ima implicitnu krajnju vrednost 0. Ove instrukcije su pogodne za implementaciju, na primer, **for** iskaza kod Pascala.

7.5.2. While iskaz

Iskaz **while** ... **do** se obično ne podržava specijalnim instrukcijama. Blok na kome se vrši iteracija ima test na početku, koji uzrokuje skok (jump) "preko" bloka ako specificirani uslov nije istinit. Zadnja instrukcija bloka je bezuslovni skok na test na početku. Mogući oblik **while** ... **do** iskaza za mikroprocesor MC68020 ima oblik:

```
while A < 5 do          Lbeg:  CMP   5,A
  begin                        BGE   Lend    ; Uslovno grananje
     ⋮                          ⋮     ⋮
    A:=A + J;                  ADD   J,A
     ⋮                          ⋮     ⋮
  end;                         BRA   Lbeg    ; Bezulsovno grananje
                        Lend:
```

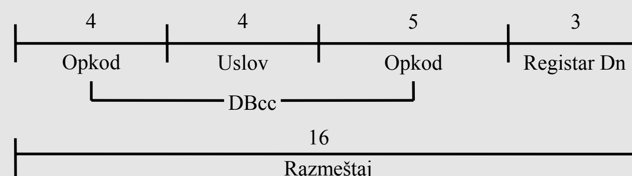
7.5.3. Repeat iskaz

Iteracioni mehanizam **repeat** ... **until**, se obično ne podržava specijalnim instrukcijama. Blok nad kojim se vrši iteracija ima test na kraju, tako da se blok izvršava najmanje jedanput. Ovo znači da ovaj iteracioni mehanizam zahteva manje instrukcija u odnosu na **while** ... **do** konstrukciju, jer ne postoji bezuslovna instrukcija skoka (jump). Veliki broj optimizirajućih kompilatora može prepoznati petlju tipa **while** ... **do** koja će se izvršiti najmanje jedanput i da prevede taj tip petlji u njihov **repeat** ... **until** ekvivalent. Moguće prevođenje **repeat** ... **until** jezičke konstrukcije kod MC68020 ima sledeći oblik:

```
Cnt:=100;           MOVE.L      #100,D0       ; D0:=Cnt
do           loop     ⋮
  ⋮
  read(x);          JSR         READ          ; Čita X
  ⋮                   ⋮                       ; Rezultat je u steku
```

Primer 7.10:

Kod MC68020 postoji instrukcija DBcc (Decrement and Branch According to Condition) kao što je prikazano na slici 7.9.



Sl. 7.9. Format instrukcije DBcc mikroprocesora MC68020.

DBcc instrukcija podržava tri osnovna iteraciona mehanizma. Ova instrukcija se postavlja na kraj kod **repeat** ... **until** i **for** petlji, a na početku kod **while** ... **do** petlji. DBcc prvo testira uslov da bi odredila kada je završeni uslov petlje istinit. Ako je istinit, operacije se ne izvršavaju i petlja završava. Ako je uslov završetka pogrešan, LS reč vrednosti koja se broji u Dn se dekrementira za 1 (brojač se definiše kao 16-bitni). Ako je rezultat -1 petlja se završava, a izvršenje se produžava sledećom instrukcijom. Ako rezultat nije jednak -1, izvršenje se produžava na lokaciji označenoj tekućom vrednošću PC-a, plus 16-bitni razmeštaj koji se znakovno proširuje (pozitivni za **while** ... **do** petlje, a negativan u ostalim slučajevima), a specificiran je DBcc instrukcijom.