UNIVERZITET UNION „NIKOLA TESLA“-BEOGRAD
POSLOVNO PRAVNI FAKULTET

ARHITEKTURA I ORGANIZACIJA RAČUNARSKIH SISTEMA

SKRIPTA-Prvo izdanje

Dušan Regodić

Beograd, 2018.godina

Predgovor

Ova skripta je nastala kao rezultat potrebe za odgovarajućim pisanim materijalom iz predmeta **Arhitektura i organizacija računarskih sistema**.

Pri pisanju je učinjen napor da skripta bude prihvatljiva za čitaoce bez nekog većeg predznanja u oblasti računarstva, ali uz posedovanje osnovnog znanja iz matematike. Namenjena je i prilagođena prosečnom studentu, jer je osnovni cilj autora bio da svi studenti koji slušaju predmet Arhitektura i organizacija računarskih sistema mogu na razumljiv i lak način da savladaju predviđeno gradivo. Upravo iz tog razloga, knjiga ne sadrži značajnija teorijska razmatranja, niti prikazuje punu širinu i složenost razmatranih problema, već je prvenstveno orijentisana ka praktičnim aspektima računarske tehnike koji su ilustrovani brojnim primerima.

Skripta je podeljena u sedam poglavlja: *Matematičke osnove računarske tehnike*, *Logički elementi*, *Memorijski elementi*, *Logičke funkcije*, *Logičke mreže*, *Osnovi organizacije računara* i *Personalni računar*.

U prvom poglavlju izložen je matematički aparat na kome se zasniva rad svakog računarskog sistema. S obzirom da se podaci u računaru predstavljaju i obrađuju u binarnom obliku, najveća pažnja posvećena je binarnom brojnom sistemu. Osim njega, razmatran je i heksadecimalni brojni sistem. U poglavlju su dati postupci konverzije brojeva između binarnog, decimalnog i heksadecimalnog sistema, kao i osnovne aritmetičke operacije nad binarnim brojevima. Takođe su opisani načini predstavljanja različitih vrsta podataka u računaru, uključujući cele brojeve, realne brojeve i podatake znakovnog tipa.

Osnovne logičke operacije, koje se intenzivno koriste na najnižem nivou obrade podataka u računaru, tema su drugog poglavlja. One su praćene odgovarajućim logičkim elementima koji služe za njihovu implementaciju u prekidačkim mrežama.

U trećem poglavlju obrađene su različite vrste flip-flopova kao osnovnih memorijskih elemenata koji se koriste u izgradnji sekvencijalnih prekidačkih mreža. Ukazano je na njihovu povezanost sa logičkim elementima. Posebna pažnja je posvećena taktovanim flip-flopovima i njihovom učešću u analizi i sintezi prekidačkih mreža.

Četvrto poglavlje uvodi pojam logičke funkcije kojom se mogu opisati složene strukture koje obezbeđuju potrebnu funkcionalnost računarskom sistemu. Prikazana su tri načina za predstavljanje logičke funkcije: pomoću kombinacionih tablica, u algebarskom obliku i pomoću Karnoove karte. Opisan je postupak

realizacije logičkih funkcija korišćenjem prekidačkih mreža. Na kraju je izložen metod minimizacije logičkih funkcija primenom Karnoove karte u cilju smanjenja složenosti njihove realizacije.

U petom poglavlju opisane su razne prekidačke mreže koje predstavljaju standardne module kombinacionog (koderi, dekoderi, multiplekseri, ...) i sekvencijalnog (registri, brojači, ...) tipa u realizaciji računarskog sistema. Za svaki modul dati su njegova funkcionalnost, osnovne osobine, mogućnosti primene i unutrašnja realizacija.

Opšta organizacija računarskog sistema i osnovni principi njegovog funkcionisanja opisani su u šestom poglavlju. Osim osnovnih pojmova, u poglavlju su izloženi i pojedini koncepti koji doprinose većoj efikasnosti rada računara: *pipeline*, DMA prenos i mehanizam prekida. U drugom delu poglavlja, organizacija računara je detaljno objašnjena na praktičnom primeru računarskog okruženja u kome centralno mesto zauzima procesor Intel 8086.

Poslednje poglavlje posvećeno je predstavljanju najbitnijih komponenata personalnog računara: matične ploče, procesora, memorija i ulazno/izlaznih uređaja. Naglašena je centralna uloga matične ploče i opisani su njeni delovi. Posebna pažnja posvećena je memorijama i to operativnoj memoriji, raznim vrstama spoljašnjih memorija i keš memoriji. Od ulazno/izlaznih uređaja predstavljeni su monitori i štampači.

Na kraju svakog poglavlja, u okviru *Vežbanja*, data su pitanja i zadaci za samostalno rešavanje koji studentima treba da posluže kao provera znanja na temu koja je razmatrana u poglavlju. Pitanja i zadaci su odabrani tako da u potpunosti pokrivaju predviđeno gradivo, pa se mogu iskoristiti za pripremanje ispita.

S A D R Ž A J

1 Uvod u računarske sisteme

Računarski sistem (računar) je digitalni elektronski uređaj koji služi za automatsku obradu podataka. Koristi se prilikom rešavanja različitih problema kada postoji potreba da se ulazne informacije, koje predstavljaju polazno stanje problema, obrade po određenom postupku i generišu izlazne informacije, koje predstavljaju rešenje tog problema. Za izvođenje obrade, neophodni su hardver (opipljive, fizičke komponente računara) i softver (programi koje računar može da izvrši na postojećem hardveru). Računar može da obrađuje samo podatke koji su predstavljeni u binarnom obliku (nizovima nula i jedinica).

1.1 Osnovni pojmovi

Obrada podataka podrazumeva izvršavanje niza **operacija** na koje se može razložiti postupak rešavanja problema. U računaru, operacije se predstavljaju pomoću **instrukcija** (naredbi, komandi), koje su, kao i podaci, u binarnom obliku. Uređeni niz instrukcija, naziva se **program**. Instrukcija se izvršava nad podacima koji se nazivaju **operandi**.

Za izvršavanje programa zadužen je hardver računara. Osnovne hardverske komponente su:

- **memorija**, koja služi za čuvanje podataka i programa
- **procesor**, koji obavlja obradu podataka izvršavanjem programskih instrukcija
- **ulazno/izlazni uređaji**, koji omogućavaju unos ulaznih podataka u računar, kao i prikaz rezultata obrade

- **magistrala**, koja služi za povezivanje i razmenu sadržaja između ostalih hardverskih komponenata

Poznavanje računarskog sistema se može posmatrati sa dva aspekta: sa aspekta arhitekture i sa aspekta organizacije računara.

Arhitektura računara obuhvata sve ono što je potrebno znati o računaru da bi za njega mogli da se pišu i na njemu izvršavaju programi. Može se reći da arhitekturu računara čini skup atributa koji je vidljiv za programera. Da bi programer napisao program koji može da se izvrši na računaru, treba da poznaje, na primer, tipove podataka, formate instrukcija, instrukcijski set, načine adresiranja, skup programski dostupnih registara, itd.

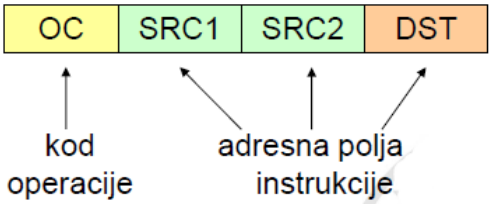
Organizacija računara se bavi načinima realizacije pojedinih delova računara. To je skup atributa transparentan (nevidljiv) za programera. Organizaciji pripadaju kontrolni signali, korišćene memorijske tehnologije, tehnike realizacije upravljačkih jedinica, tehnike realizacije keš memorija, veze računara sa periferijama i dr.

U narednim poglavljima biće detaljnije opisani pojedini atributi kako arhitekture, tako i organizacije računara.

1.2 Memorija

Memorija predstavlja skup **lokacija** (ćelija) u kojima se čuvaju binarne reči koje predstavljaju instrukcije i operande. Ako u jednu ćeliju memorije može da se smesti *n* bitova, to znači da svaka binarna reč predstavlja niz od *n* nula i jedinica. Posmatranjem ovog niza, ne može se zaključiti da li on predstavlja instrukciju ili podatak. Značenje binarne reči zavisi od toga kako je računar interpretira, tj. tumači.

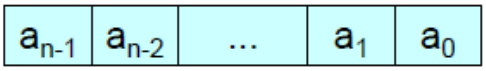
Binarnu reč koja predstavlja instrukciju, računar interpretira tako što je podeli u dva dela kao na slici 1.1.



Slika 1.1: *Jedna interpretacija instrukcije*

Prvi deo je **kod operacije** (OC) koji specificira vrstu operacije, na primer, sabiranje. Drugi deo sadrži **adresna polja instrukcije**. U zavisnosti od vrste operacije, broj polja u adresnom delu može biti različit. To mogu biti polja izvorišnih i odredišnog operanda. **Polja izvorišnih operanada** (SRC1 i SRC2) sadrže adrese lokacija u kojima se nalaze operandi nad kojima se operacija izvršava, ili same operande. **Polje odredišnog operanda** (DST) uvek sadrži adresu lokacije u koju treba smestiti rezultat operacije.

Binarnu reč koja predstavlja operand, računar može da interpretira na različite načine u zavisnosti od tipa operanda. Na slici 1.2 predstavljen je *n*-bitski operand koji se može interpretirati kao: neoznačeni ceo broj, označeni ceo broj u drugom komplementu, veličina u pokretnom zarezu, niz alfanumeriških znakova, itd. Navedeni tipovi podataka detaljno su opisani u [1].



Slika 1.2: *Operand*

1.3 Procesor

Procesor (CPU – *Central Processing Unit*) je osnovna komponeneta računara namenjena obradi podataka. Centralni deo procesora koji služi za izvršavanje instrukcija programa naziva se **aritmetičko-logička jedinica** (ALU – *Arithmetic Logic Unit*). Osim ALU, u procesoru se nalazi i određen broj tzv. **procesorskih registara** sa različitim namenama u kojima se privremeno čuvaju male količine podataka.

Da bi ALU mogla da izvrši instrukciju, potrebno je najpre da se ta instrukcija prenese iz memorije u procesor. Adresa memorijske lokacije u kojoj se nalazi instrukcija nalazi se u **programskom brojaču** (PC – *Program Counter*). PC je procesorski registar u kome se čuva adresa naredne instrukcije koju treba izvršiti. PC se realizuje kao inkrementirajući brojač zato što se instrukcije programa obično u memoriji čuvaju na sukcesivnim adresama.

Procesor ostvaruje komunikaciju sa memorijom samo preko dva svoja registra:

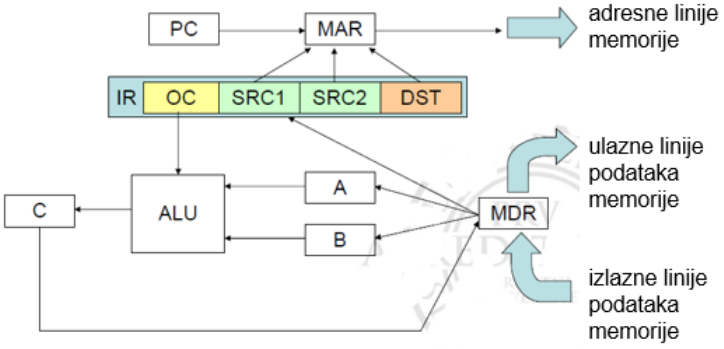
- **adresnog registra memorije** (MAR – *Memory Address Register*), koji sadrži adresu memorijske lokacije kojoj treba pristupiti u cilju upisa/čitanja; ovaj registar je magistralom povezan sa adresnim ulazom memorije
- **prihvatnog registra podatka memorije** (MDR – *Memory Data Register*), koji prihvata podatak pročitan sa izlaznih linija podataka memorije ili

prihvata podatak koji treba proslediti do ulaznih linija podataka radi upisa u memoriju

Prenos instrukcije iz memorije se vrši tako što se adresa iz PC prenese u MAR (a samim tim i do adresnog ulaza memorije), zatim se instrukcija pročita, tj. pojavi na izlaznim linijama podataka memorije, pa prosledi do MDR. S obzirom da MDR služi za prihvatanje različitih vrsta podataka, on nije pogodan za duže čuvanje instrukcije. Zato je uveden poseban procesorski registar u koji se smešta instrukcija iz MDR. To je **instrukcijski registar** (IR – *Instruction Register*). ALU izvršava instrukciju koja se nalazi u IR.

Dovođenje instrukcije u IR nije dovoljan uslov da ona bude izvršena. Iz instrukcije, ALU saznaje o kojoj instrukciji se radi (na osnovu OC), ali, mnoge instrukcije zahtevaju operande za svoje izvršavanje (na primer, instrukcija sabiranja obično zahteva dva operanda koje je potrebno sabrati). Operandi se, takođe, nalaze u memoriji, pa je i njih potrebno preneti u procesor pre izvršenja instrukcije. Prenos operanada se odvija na isti način kao i prenos instrukcije, korišćenjem registara MAR i MDR. Kada operandi stignu u MDR, šalju se u posebne procesorske **prihvatne registre za izvorišne operande** koji su direktno povezani sa ulazima ALU. Sada postoje svi uslovi da ALU izvrši instrukciju. Nakon izvršenja, rezultat se pojavljuje u **prihvatnom registru rezultata** i prosleđuje do MDR. Time je okončan proces izvršavanja instrukcije.

Na slici 1.3 data je struktura procesora iz koje se vide unutrašnje veze koje postoje između ALU i procesorskih registara, kao i spoljašnje veze procesora sa memorijom. Registri za izvorišne operande su označeni sa A i B, dok je registar rezultata označen sa C.



Slika 1.3: *Struktura procesora*

Detaljan redosled aktivnosti koje treba obaviti prilikom izvršavanja instrukcije nad dva operanda dat je tabeli 1.1. Adresni ulaz memorije označen je sa M(A),

ulazne linije podataka memorije sa M(DI), a izlazne linije podataka memorije sa M(DO).

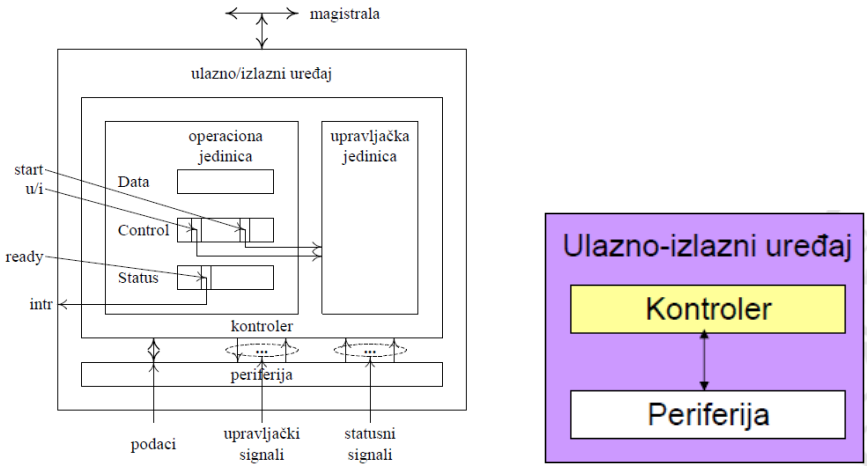
Tabela 1.1: *Primer izvršavanja instrukcije*

Proces	Aktivnosti
Prenos instrukcije	PC→MAR→M(A), M(DO)→MDR→IR
Prenos prvog operanda	SRC1→MAR→M(A), M(DO)→MDR→A
Prenos drugog operanda	SRC2→MAR→M(A), M(DO)→MDR→B
Izvršavanje instrukcije	ALU(IR(OC), A, B)→C→MDR→M(DI)
Upis rezultata u memoriju	DST→MAR→M(A), (M(A), M(DI))→M

1.4 Ulazno/izlazni uređaji

Osim unosa podataka u računar i prikaza dobijenih rezultata, ulazno/izlazni uređaji imaju još jednu važnu ulogu. Oni omogućavaju korisnicima da specificiraju ulazne/izlazne podatke u obliku koji im najviše odgovara. S obzirom da računar obrađuje samo binarne podatke, ulazno/izlazni uređaji konvertuju korisničke formate u binarni oblik i obrnuto.

U/I uređaj se sastoji od dve komponente: periferije i kontrolera (slika 1.4).



Slika 1.4: *Struktura U/I uređaja*

Periferija se realizuje kao standardni uređaj, na primer, tastatura, miš, štampač, itd. Između periferije i računara obavlja se kontrolisana razmena podataka. Zato svaka periferija mora da ima:

- linije za prenos podataka (podaci su u obliku specifičnom za datu periferiju)
- upravljačke linije, kojima se šalju komande periferiji
- statusne linije, kojima se šalje trenutni status periferije

Način prenosa podataka od/do periferije definisan je protokolom koji je specifičan za svaku periferiju.

Periferija može biti ulazna (podaci se sa periferije prosleđuju do procesora ili memorije) i izlazna (podaci iz memorije ili procesora se prosleđuju u periferiju).

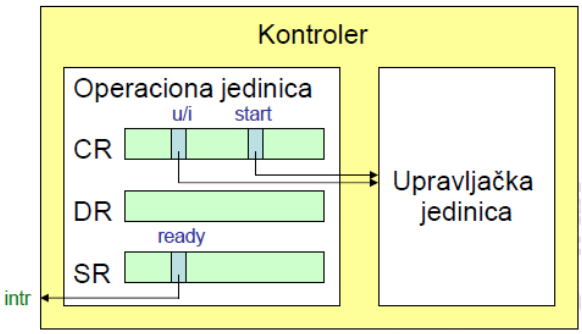
Kontroler upravlja radom periferije. Za svaku periferiju definiše se poseban kontroler. Osnovne uloge kontrolera su:

- fizičko povezivanje periferije sa memorijom i procesorom putem magistrale
- omogućavanje programske kontrole prenosa podataka od/do periferije

Programska kontrola omogućava da se:

- startuje i zaustavi kontroler programskim putem
- dobije informacija o tome da li je podatak prenet iz ulazne periferije u kontroler
- dobije informacija o tome da li podatak prenet iz kontrolera u izlaznu periferiju
- izvrši prenos podatka iz kontrolera ulazne periferije u memoriju ili procesor
- izvrši prenos podatka iz memorije ili procesora u kontroler izlazne periferije

Kontroler periferije se sastoji iz operacione jedinice i upravljačke jedinice (slika 1.5). **Upravljačka jedinica** generiše signale za prenos podataka iz ulazne periferije u kontroler ili iz kontrolera u izlaznu periferiju.



Slika 1.5: *Struktura kontrolera periferije*

Operaciona jedinica obuhvata tri registara kojima se pristupa programski tokom izvršavanja instrukcija:

- registar podatka (DR – *Data Register*)
- upravljački registar (CR – *Control Register*)
- statusni registar (SR – *Status Register*)

DR služi za prihvatanje podatka iz ulazne periferije, ili za prihvatanje podatka koji treba da se prosledi izlaznoj periferiji.

U sadržajima CR i SR svaki bit, tj. fleg (*flag*) ima drugačije značenje. U CR se nalazi fleg *start* koji omogućava aktiviranje (*start* ima aktivnu vrednost) ili zaustavljenje kontrolera (*start* ima neaktivnu vrednost). Nakon aktiviranja kontrolera, na osnovu vrednosti flaga *u/*i određuje se smer prenosa podatka, tj. da li je periferija ulazna (*u/i* ima aktivnu vrednost) ili izlazna (*u/i* ima neaktivnu vrednost). SR sadrži bit *ready* koji indicira da je podatak prenet iz ulazne periferije u DR ili da je podatak iz DR prenet u izlaznu periferiju. U oba slučaja treba generisati signal prekida *intr* koji kontroler šalje procesoru. Kada procesor primi *intr* signal, prekida izvršavanje tekućeg programa i prelazi na izvršavanje prekidne rutine u okviru koje se sadržaj iz DR prenosi u odgovaruću lokaciju u memoriji ili u procesor, ili obrnuto; nakon toga procesor nastavlja sa izvršavanjem tekućeg programa.

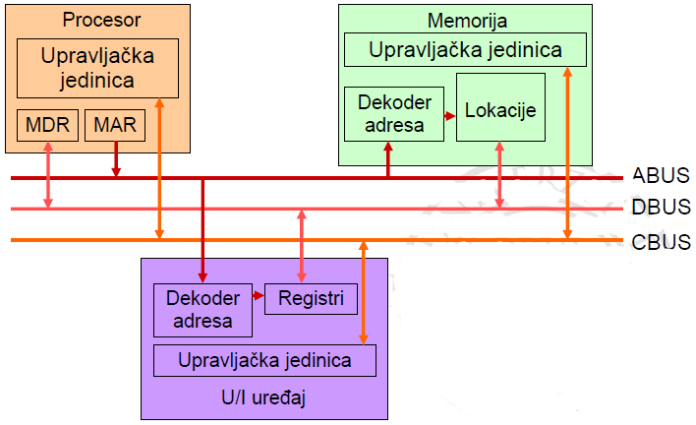
1.5 Magistrala

Magistrala predstavlja uređeni skup linija kojima se povezuju komponente rečunara (memorija, procesor, U/I uređaji). Ceo tok prenosa sadržaja između dve komponente naziva se **ciklus na magistrali**.

Komponenta koja započinje ciklus je **gazda** (*master*) na magistrali. Komponenta sa kojom se realizuje ciklus naziva se **sluga** (*slave*). Ulogu gazde najčešće ima procesor, a ulogu sluge memorija ili U/I uređaj.

Skup linija po kojima gazda šalje adresu memorijske lokacije ili adresu registra U/I uređaja naziva se **adresna magistrala** (ABUS). Skup linija po kojima se prenose podaci naziva se **magistrala podataka** (DBUS). DBUS je bidirekciona (dvosmerna) zato što prilikom upisa gazda šalje slugi podatak koji treba upisati, a prilikom čitanja sluga šalje pročitani podatak gazdi. Skup linija po kojima gazda šalje signale za upis ili čitanje sadržaja, ili sluga šalje signale da su upis ili čitanje realizovani naziva se **upravljačka magistrala** (CBUS).

Na slici 1.6 prikazano je povezivanje procesora (gazda), kao i memorije i U/I uređaja (sluge) na magistralu.



Slika 1.6 Priključivanje komponenata na magistralu

Svaka komponenta ima upravljački deo koji se priključuje na CBUS. Sa DBUS dvosmernu vezu ostvaruju MDR procesora, linije za podatke memorije i registri operacione jedinice kontrolera U/I uređaja. Sa ABUS su povezani MAR procesora i dekoderi adresa memorije i U/I uređaja. Smerovi povezivanja sa ABUS zavise od uloge komponente, tj. da li je ona gazda ili sluga.

Vežbanja

1. Šta je pozicioni brojni sistem? Na koji način se bilo koji pozitivan ceo broj može predstaviti u pozicionom brojnom sistemu?

2. Naći decimalne i heksadecimalne vrednosti sledećih binarnih brojeva:

   a) $100111_{(2)}$
   b) $11010_{(2)}$
   c) $1000101_{(2)}$
   d) $1010001_{(2)}$
   e) $101001110_{(2)}$

2 Programske instrukcije

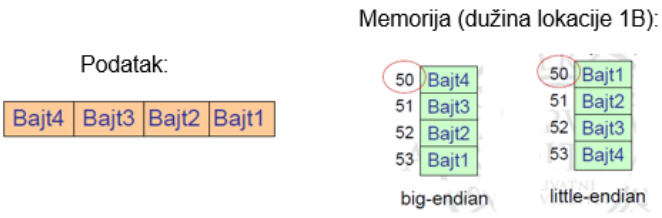
Proces obrade podataka u računaru sprovodi se izvršavanjem programskih instrukcija u zadatom redosledu. Instrukcije se obično izvršavaju nad nekim operandima. Svaka instrukcija ima precizno definisan format (izgled), u okviru koga je zadato kog tipa moraju da budu operandi. U ovom poglavlju biće predstavljeni formati instrukcija koji se najčešće koriste, a koje podržavaju savremeni procesori.

2.1 Tipovi podataka

U računaru, binarni nizovi nula i jedinica mogu da predstavljaju podatke različitih tipova: neoznačene cele brojeve, cele brojeve sa znakom, brojeve u drugom komplementu, brojeve u pokretnom zarezu, alfanumeričke znakove, itd. Na primer, niz 100011 može da se tumači kao 35 (neoznačeni ceo broj), -3 (ceo broj sa znakom) ili -29 (broj u drugom komplementu). Postupci predstavljanja podataka detaljno su opisani u [1].

Podaci bilo kog tipa mogu biti različite dužine, na primer 8, 16, 32, 64 bita. Dužina podataka je vrlo bitna zato što su prostori (memorijske lokacije, registri) za smeštaj podataka u računaru ograničeni. Tako se može desiti da je podatak duži od veličine memorijske lokacije u koju treba da se upiše. U ovom slučaju, podatak se mora smestiti u više sukcesivnih memorijskih lokacija, što komlikuje rad sa tim podatkom.

Pretpostavimo da podatak od 4 bajta treba da se upiše u memoriju čije lokacije imaju dužinu 1 bajt (slika 2.1).



Slika 2.1: *Upis podatka u memoriju*

Zbog svoje dužine, podatak mora biti upisan u četiri uzastopne memorijske lokacije, na primer sa adresama 50, 51, 52 i 53. Postoje dva načina upisa ovog podatka:

- da se bajt najveće težine (B4) upiše u lokaciju sa najmanjom adresom (50), a zatim da se redom upisuju ostali bajtovi; ovaj postupak se naziva **big-endian**
- da se bajt najmanje težine (B1) upiše u lokaciju sa najmanjom adresom (50), a zatim da se redom upisuju ostali bajtovi; ovaj postupak se naziva **little-endian**

Ovakvo smeštanje podataka zahteva složeniju realizaciju procesora, zato što procesor mora da zna da kada se traži podatak sa adrese 50, treba očitati četiri bajta sa adresa 50, 51, 52 i 53 i od pročitanih sadržaja formirati podatak.

2.2 Formati instrukcija

Svaka instrukcija u računaru predstavljena je nizom nula i jedinica. Značenje bitova u ovom nizu određeno je formatom instrukcije. Format sadrži sledeće informacije:

- vrstu operacije koju treba izvršiti
- tip operanada nad kojima se izvršava operacija
- specifikaciju izvorišnih i odredišnih operanada

Dužina instrukcije zavisi od veličine memorije i njene organizacije, širine magistrale, kao i složenosti i brzine procesora. Odluka o dužini instrukcije bitno utiče na fleksibilnost računara sa stanovišta programera. Programerima više odgovaraju raznovrsnije i moćnije instrukcije, zato što su tada programi koje pišu kraći. Međutim, ovakve instrukcije su duže (po broju bitova), pa je potrebno više prostora za njihov smeštaj u memoriji. Pošto su ovi zahtevi oprečni, obično se pravi balans (*trade-off*) između njih.

Kao što je rečeno u 1.2, format instrukcije obuhvata kod operacije i adresni deo instrukcije. Kodom operacije se specificira vrsta operacije i tip operanda. Tip operanda određuje iz koliko susednih memorijskih lokacija treba pročitati operand, ili u koliko susednih lokacija treba upisati operand i kako ga interpretirati.

Ako se ista vrsta operacije primenjuje nad operandima različitih tipova ili različitih dužina, neopohodno je za svaki operand definisati poseban kod operacije. Na primer, ako procesor podržava rad sa neoznačenim celim brojevima dužine 8 i

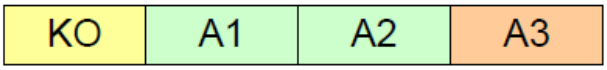
16 bitova i brojevima predstavljenim u pokretnom zarezu dužine 32 i 64 bita, za operaciju množenja neophodno je definisati četiri različita koda operacije. Primena jednog od ovih kodova operacije, direktno određuje tip operanda nad kojim će operacija biti izvršena.

Adresni deo instrukcije obično sadrži 3, 2, 1 ili 0 polja za specifikaciju operanada. Prema broju polja u adresnom delu instrukcija koje mogu da izvršavaju, procesori mogu biti troadresni, dvoadresni, jednoadresni ili nulaadresni. Procesor jednog tipa može da izvršava i instrukcije sa manje operanada (na primer, troadresni procesor može da izvršava i dvoadresne instrukcije). Procesor uvek mora da zna kako da interpretira sva polja u formatu instrukcije koju izvršava.

Za operande koji su zadati u adresnom delu instrukcije kaže se da su **eksplicitno definisani**. Međutim, ne moraju svi operandi da budu specificirani u adresnim poljima instrukcije. Na primer, jednoadresni format ne pruža mogućnost zadavanja dva operanda. U ovom slučaju se unapred zna gde se nalazi drugi operand. Za njega se kaže da je **implicitno definisan**.

2.2.1 Troadresni format

Troadresni format instrukcije prikazan je na slici 2.2.



Slika 2.2: *Troadresni format instrukcije*

Osim koda operacije, u formatu se nalaze još tri adresna polja za operande koji su eksplicitno definisani. Interpretacija ovih polja može biti različita. Obično se u poljima A1 i A2 nalaze adrese izvorišnih operanda, a u polju A3 adresa odredišnog operanda. Međutim, postoje i druge interpretacije, na primer, može u polju A1 da bude adresa odredišnog operanda, a u poljima A2 i A3 adrese izvorišnih operanada. U poljima namenjenim izvorišnim operandima, umesto adresa mogu da se nalaze i sami operandi. U polju koje odgovara odredišnom operandu uvek mora da bude adresa, jer rezultat operacije nije unapred poznat.

Dobra strana ovog formata je u tome što se binarna operacija (nad dva operanda) izvršava jednom instrukcijom. Loša strana formata je u velikoj dužini instrukcije.

Procenu dužine troadresne instrukcije možemo ilustrovati na primeru korišćenja memorije kapaciteta 1GB. Ova memorija ima oko 2^{30} lokacija dužine 1B ($1GB = 10^9B \approx 2^{30}B$). Dakle, za adresiranje jedne lokacije ove memorije potrebno je oko 30 bitova. Tri adrese u formatu instrukcije imaju dužinu oko 90 bitova. Ako je kod operacije 8-bitni, dužina cele instrukcije je oko 100 bitova. To znači da je za smeštaj jedne instrukcije potrebno bar 12 memorijskih lokacija, tj.

prilično veliki prostor. Problem nije samo u prostoru, već i u brzini čitanja ove instrukcije. Da bi se kompletirala ova instrukcija, potrebno je 12 obraćanja memoriji, što usporava njeno izvršavanje.

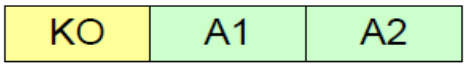
Izvršavanje instrukcije u troadresnom procesoru biće ilustrovano na primeru izračunavanja izraza A + B → C. Za računanje ovog izraza dovoljna je samo jedna instrukcija:

ADD A B C

Data instrukcija se izvršava tako što se izvorišni operandi koji se nalaze u memorijskim lokacijama čije su adrese A i B saberu, i dobijeni zbir smesti u memorijsku lokaciju čija je adresa C. Operacija sabiranja definisana je kodom operacije ADD.

2.2.2 Dvoadresni format

Dvoadresna instrukcija prikazana je na slici 2.3.



Slika 2.3: *Dvoadresni format instrukcije*

Ovaj format se interpretira tako što se smatra da oba adresna polja predstavljaju izvorišne operande. Lokacija odredišnog operanda se zadaje implicitno kao jedna od izvorišnih lokacija (za svaki procesor se definiše koja je to lokacija). Na primer, može se uzeti da se rezultat operacije smešta u memorijsku ćeliju sa adresom prvog izvorišnog operanda. To znači da se pre izvršenja instrukcije u datoj ćeliji nalazi izvorišni operand, a nakon izvršenja instrukcije odredišni operand (izvorišni operand se gubi).

U poređenju sa troadresnim formatom, dobra strana dvoadresnog formata je u tome što je instrukcija kraća, a loša strana u tome što je potrebno više instrukcija za izračunavanje izraza.

Povećanje broja instrukcija biće ilustrovano na ranijem primeru izračunavanja izraza A + B → C. Kod dvoadresnih procesora, za računanje ovog izraza potrebne su dve instrukcije:

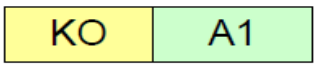
MOV C A
ADD C B

Kod operacije MOV definiše operaciju prenosa, a kod operacije ADD operaciju sabiranja. Neka u datom procesoru lokacija odredišnog operanda kod sabiranja odgovara adresi prvog izvorišnog operanda. Prva instrukcija se izvršava tako što se sadržaj lokacije na adresi A pročita i smesti u lokaciju sa adresom C.

Ovo je bilo potrebno da bi se za drugu instrukciju obezbedilo da prvi izvorišni operand bude na odgovarajućem mestu. U drugoj instrukciji, izvorišni operandi iz lokacija sa adresama C i B se sabiraju i rezultat smešta u lokaciju sa adresom C.

2.2.3 Jednoadresni format

Format jednoadresne instrukcije prikazan je na slici 2.4.



Slika 2.4: *Jednoadresni format instrukcije*

U ovom formatu, eksplicitno je definisan samo jedan izvorišni operand, dok su drugi izvorišni operand i odredišni operand implicitno definisani. Podrazumeva se da se oni čuvaju u posebnom procesorskom registru koji se naziva **akumulator**.

Dobra strana ovog formata je u maloj dužini instrukcije, a loša strana je u tome što je potrebno još više instrukcija za izračunavanje izraza nego u slučaju dvoadresnog formata.

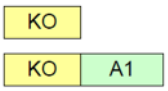
Za raniji primer računanja izraza A + B → C, kod jednoadresnih procesora potrebne su tri instrukcije:

```
LOAD A
ADD B
STORE C
```

Kod operacije LOAD definiše operaciju prenosa u akumulator, kod operacije ADD operaciju sabiranja, a kod operacije STORE operaciju prenosa iz akumulatora. Prva instrukcija se izvršava tako što se sadržaj lokacije na adresi A pročita i smesti u akumulator. Druga instrukcija čita sadržaj iz lokacije sa adresom B, sabira ga sa sadržajem akumulatora i rezultat smešta u akumulator. Treća instrukcija sadržaj akumulatora prenosi u lokaciju sa adresom C.

2.2.4 Nulaadresni format

Nulaadresni procesori podržavaju rad sa instrukcijama u dva formata prikazana na slici 2.5.



Slika 2.5*: Formati instrukcija kod nulaadresnih procesora*

Prvi format je nulaadresni jer nema adresnih polja. Oba izvorišna i odredišni operand su implicitno definisani. Za njihovo smeštanje se koristi **stek**.

Drugi format je jednoadresni i predstavlja izuzetak kod nulaadresnih procesora. Zbog čuvanja operanada na steku, bilo je neophodno uvesti dve jednoadresne instrukcije za rad sa stekom. Instrukcija PUSH upisuje na vrh steka sadržaj lokacije čija je adresa data u adresnom polju, dok instrukcija POP sadržaj sa vrha steka upisuje u lokaciju čija je adresa navedena u adresnom polju.

U odnosu na ostale formate, dužina instrukcije kod nulaadresnog formata je najmanja, ali je potrebno najviše instrukcija za izračunavanje izraza.

Izraz iz ranijeg primera, A + B → C se kod nulaadresnih procesora računa pomoću četiri instrukcije:

```
PUSH A
PUSH B
ADD
POP C
```

Prva instrukcija se izvršava tako što se sadržaj lokacije na adresi A pročita i smesti na vrh steka. Druga instrukcija čita sadržaj iz lokacije sa adresom B i takođe ga smešta na vrh steka (prethodni sadržaj sa vrha steka se pomera za jedno mesto niže u steku). Treća instrukcija čita dva operanda sa vrha steka, sabira ih i rezultat smešta na vrh steka. Četvrta instrukcija čita sadržaj sa vrha steka i upisuje ga u lokaciju sa adresom C.

Analizom opisanih postupaka izračunavanja izraza iz primera pomoću procesora različite adresnosti, mogu se izvesti sledeći zaključci:

- sadržaji memorijskih lokacija na adresama A, B i C u svim slučajevima ostaju nepromenjeni
- u svim slučajevima se na drugačiji način formira traženi zbir u lokaciji sa adresom C
- svaki od slučajeva ima drugačije memorijske i vremenske zahteve

2.3 Lociranje operanada

Operandi nad kojima se izvršavaju programske instrukcije mogu se nalaziti:

- u memorijskim lokacijama
- u procesorskim registrima
- u adresnim poljima formata instrukcije

U zavisnosti od toga gde su smešteni operandi, razlikuju se tri vrste arhitektura:

- memorija – memorija
- memorija – registar
- registar – registar

Kod arhitekture memorija – memorija, za smeštaj operanada (izvorišnih i odredišnih) koriste se isključivo memorijske lokacije. U ovoj arhitekturi se mogu primenjivati svi navedeni formati instrukcija.

Arhitektura memorija – registar podrazumeva da se jedan izvorišni operand nalazi u registru, drugi u memoriji, a odredišni operand bilo u memoriji ili u registru. U ovoj arhitekturi se obično koristi dvoadresni format.

U arhitekturi registar – registar, svi operandi se nalaze isključivo u registrima, dok se memorijskim lokacijama pristupa samo pomoću instrukcija LOAD i STORE. U ovoj arhitekuti se obično koristi troadresni format.

Vežbanja

1. Prikazati kombinacione tablice za sledeće logičke operacije:
   a) logičko množenje
   b) logičko sabiranje

3 Procesorski registri

U svakom procesoru postoji skup registara koji služe za privremeno čuvanje malih količina podataka neophodnih za pravilan rad procesora. Skup procesorskih registara definiše projektant procesora u zavisnosti od korišćenog načina projektovanja. Prema mogućnostima pristupa, procesorski registri se mogu svrstati u dve grupe: **interni registri** i **programski dostupni registri**.

3.1 Interni registri

Interni registri su registri procesora kojima se ne može programski pristupiti (ne postoje instrukcije koje mogu da upišu ili modifikuju njihov sadržaj). Ovim registrima se pristupa samo iz ugrađenih algoritama po kojima se instrukcija izvršava, pa stoga pripadaju organizaciji računara. Služe za čuvanje sadržaja u različitim fazama izvršavanja neke instrukcije, a njihov sadržaj se može koristiti samo u okviru te instrukcije.

Interni registri obuhvataju **kontrolne** i **statusne registre**.

3.1.1 Kontrolni registri

Tokom izvršavanja instrukcije, za prenos podataka između memorije i procesora koriste se sledeći kontrolni registri:

- programski brojač (PC) – sadrži adresu naredne instrukcije
- instrukcijski registar (IR) – sadrži instrukciju
- adresni registar memorije (MAR) – sadrži adresu memorijske lokacije

- registar podatka memorije (MDR) – sadrži podatak pročitan iz memorije ili podatak koji treba upisati u memoriju

Treba napomenuti da se PC inkrementira implicitno nakon svake instrukcije, međutim, može se postaviti i eksplicitno instrukcijama skoka. To znači da mu se može pristupiti i programski, tj. da se u slučaju skokova ponaša kao programski dostupan registar.

Da bi bili procesirani, podaci se moraju dovesti na ulaze aritmetičko-logičke jedinice. MDR može direktno da bude priključen na ALU.

3.1.2 Statusni registri

Statusni registar se sastoji od određenog broja indikatora, tj. flegova (*flags*) kojima se opisuje trenutno stanje procesora. Indikatori odgovaraju pojedinačnim bitovima u registru. Postavljaju se nezavisno jedan od drugog i predstavljaju različite statusne informacije.

Često se statusni registar naziva PSW (*Program Status Word*) registrom, što je preuzeto iz IBM arhitekture.

Indikatori se mogu svrstati u dve grupe:

- statusni indikatori
- upravljački indikatori

Statusni indikatori se postavljaju hardverski na osnovu rezultata dobijenog izvršavanjem instrukcije, a proveravaju se instrukcijama uslovnog skoka. **Upravljački indikatori** se postavljaju softverski tokom izvršavanja posebnih instrukcija programa, a proveravaju se hardverski. Zbog upravljačkih indikatora, može se reći da su statusni registri delom programski dostupni.

Uobičajeni statusni indikatori dati su u tabeli 3.1.

Tabela 3.1: *Statusni indikatori*

Indikator	Opis
N (*negative flag*)	postavlja se na 1 ako je rezultat operacije negativan
Z (*zero flag*)	postavlja se na 1 ako je rezultat operacije 0
C (*carry flag*)	postavlja se na 1 ako ima prenosa/pozajmice pri aritmetičkim operacijama nad celobrojnim veličinama bez znaka
V (*overflow flag*)	postavlja se na 1 ako ima prekoračenja pri aritmetičkim operacijama nad celobrojnim veličinama sa znakom

Uobičajeni upravljački indikator je I (*interrupt flag*) koji se postavlja na 1 ako su dozvoljeni maskirajući prekidi.

3.2 Programski dostupni registri

Programski dostupan registar je registar procesora čiji se sadržaj može upisivati i čitati programskim putem, tj. tokom izvršavanja instrukcija. Registru se može pristupati na dva načina:

- eksplicitno, specificiranjem njegove adrese u adresnom polju instrukcije
- implicitno, izborom koda operacije instrukcije koji autoamtski uključuje specifikaciju registra

Za razliku od internih registara koji pripadaju organizaciji računara, programski dostupni registri su deo arhitekture računara. Oni služe za čuvanje rezultata izvršavanja neke instrukcije koji se kasnije mogu koristiti pri izvršavanju drugih instrukcija. Broj i uloga ovih registara razlikuje se od procesora do procesora.

Najčešće korišćenji programski dostupni registri su:

- registri podataka – DR (*Data Registers*)
- adresni registri – AR (*Address Registers*)
- registri opšte namene – GPR (*General-purpose Registers*)

Osim navedenih, ovoj grupi registara mogu da pripadaju programski brojač (kod instrukcija uslovnog skoka) i statusni registar (zbog upravljačkih indikatora).

Registri podataka služe samo za čuvanje podataka i ne mogu se koristiti pri računanjima adresa operanada. Procesor ovim podacima pristupa znatno brže nego podacima koji se nalaze u memoriji jer je pristup memoriji skoro za red veličine sporiji. U registre podataka se obično smeštaju međurezultati obrade, kao i podaci koji se više puta koriste prilikom nekih izračunavanja.

Razlozi za uvođenje registara podataka su:

- sekvencijalna priroda obrade, pri kojoj se često rezultat jedne operacije koristi kao ulazni podatak za narednu operaciju (rezultat prve operacije se smešta u DR, a ne u memoriju, pa mu se pri izvršavanju naredne operacije brže pristupa)

- pri obradi je velika verovatnoća da ako se jednom pristupi nekom podatku, isti podatak će biti uskoro opet korišćen

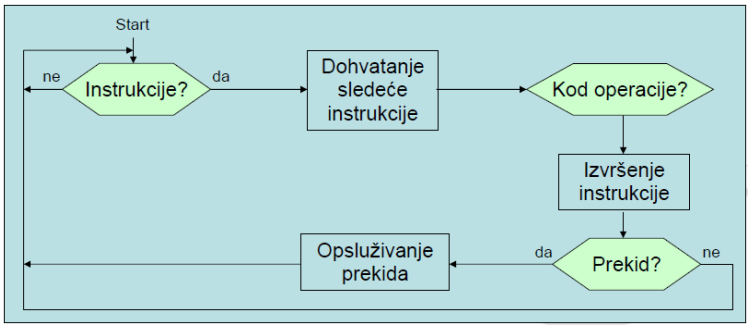
Adresni registri omogućavaju brže generisanje adresa, jer se tokom izvršavanja programa adrese (ili njihovi delovi) uzimaju iz ovih registara, a ne iz memorije, što bi bilo znatno sporije. Adresni registri mogu biti opšteg karaktera, ali mogu biti i specijalizovani u zavisnosti od primenjenog načina (moda) adresiranja. Adresni modovi su detaljno opisani u poglavlju 4. Specijalizovni adresni registri su, na primer, bazni i indeksni registri.

Kao što im ime kaže, **registri opšte namene** mogu da imaju različite uloge. U njima se mogu čuvati podaci (kao u registrima podataka) ili adrese (kao u adresnim registrima). U zavisnosti od načina projektovanja, postoje procesori koji:

- imaju specijalizovane registre (registre podataka, adresne registre, bazne registre i indeksne registre)
- imaju samo registre opšte nemene, koji preuzimaju uloge specijalizovanih registara

3.3 Instrukcijski ciklus

Procesor izvršava program tako što iz memorije dohvata jednu po jednu njegovu instrukciju dok god ih ima. Na osnovu operacionog koda instrukcije, ALU izvršava odgovarajuću operaciju. Po završetku izvršenja svake instrukcije, procesor proverava da li ima zahteva za prekidom. Ukoliko ima, procesor opslužuje prekid izvršavanjem konkretne prekidne rutine. Nakon toga, dohvata sledeću instrukciju iz memorije. Ako nema zahteva za prekidom, procesor odmah dohvata narednu instrukciju. Ovaj proces se naziva instrukcijskim ciklusom i prikazan je na slici 3.1. To je osnovni ciklus rada računara.



Slika 3.1: *Instrukcijski ciklus*

Dohvatanje instrukcije iz memorije se obavlja sledećom sekvencom mikro operacija:

- sadržaj programskog brojača se prenosi u adresni registar memorije, PC → MAR
- inkrementira se sadržaj programskog brojača (ova mikro operacija može da se radi istovremeno sa pristupom memoriji), PC + 1 → PC
- pročitani sadržaj memorije predstavlja instrukciju koja se smešta u memorijski registar podatka, M → MDR
- sadržaj memorijskog registra podatka se penosi u instrukcijski registar, MDR → IR

Sekvenca mikro operacija koje je potrebno obaviti prilikom **izvršavanja instrukcije** zavisi od same instrukcije i biće prikazna na dva primera instrukcija sabiranja.

Primer 1: ADD R1, R2, R0 (R1) + (R2) → R0

Sekvenca mikro operacija:

- adrese registara R1, R2 i R0 se uzimaju direktno iz formata instrukcije
- sadržaji registara R1 i R2 se dovode na ulaze ALU radi sabiranja
- izlaz iz ALU se prenosi u registar R0

Primer 2: ADD X, R0 M(X) + (R0) → R0

Sekvenca mikro operacija:

- adresa memorijske lokacije X se uzima iz formata instrukcije i smešta u MAR
- podatak pročitan iz memorije (sadržaj na adresi X) se smešta u MDR
- izlaz iz ALU se prenosi u registar R0
- sadržaji registara MDR i R0 se dovode na ulaze ALU radi sabiranja
- izlaz iz ALU se prenosi u registar R0

Opsluživanje prekida podrazumeva izvšavanje sledeće sekvence mikro operacija:

- sadržaj programskog brojača se prenosi u memorijski registar podatka (mora se sačuvati zbog povratka u glavni program), PC → MDR
- u memorijski adresni registar se upisuje adresa A na kojoj će se sačuvati sadržaj programskog brojača, A → MAR
- u programski brojač se upisuje adresa prve instrukcije u prekidnoj rutini A1, A1 → PC
- stara vrednost programskog brojača iz MDR se smešta u memoriju na adresu iz MAR, MDR → M(MAR)

Vežbanja

1. Koje vrste signala postoje u digitalnim kolima?

4 Adresni modovi

Adresni modovi predstavljaju načine adresiranja operanada instrukcije. U zavisnosti od toga gde se operandi nalaze, definišu se različiti adresni modovi. Ako se operand nalazi u nekoj memorijskoj lokaciji, adresni mod specificira kako se formira adresa te lokacije. Jednostavniji slučaj je kada se operand nalazi u nekom procesorskom registru, jer tada adresa registra istovremeno predstavlja i adresu operanda. Do operanda se najlakše dolazi ako je on direktno upisan u polje unutar formata instrukcije.

Adresni modovi programerima pružaju brojne mogućnosti kao što su: korišćenje pokazivača, korišćenje brojača za kontrolu petlji, indeksiranje podataka, itd. Zahvaljujući njima, redukovan je broj bitova u adresnim poljima instrukcija, čime je smanjena veličina instrukcija, a samim tim i prostor potreban za njihovo smeštanje.

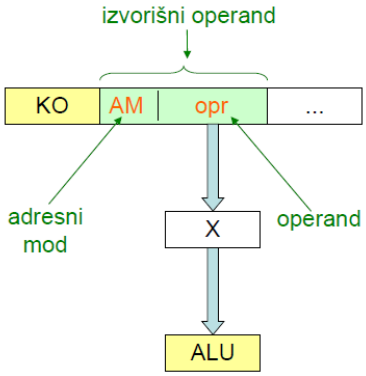
U ovom poglavlju, opisani su najčešće korišćeni adresni modovi: neposredno adresiranje, direktno adresiranje, indirektno adresiranje i adresiranja sa pomerajem, kao i njihove varijante.

4.1 Neposredno adresiranje

Neposredno adresiranje je najjednostavniji adresni mod koji podrazumeva da se operand nalazi unutar same instrukcije. Pošto svaka promena operanda zahteva izmene u svim instrukcijama u kojima se taj operand koristi, ovaj adresni mod ima malu fleksibilnost.

Neposredno adresiranje se može koristiti samo za adresiranje izvorišnih operanada, jer zadavanje odredišnog operanda na ovaj način nema smisla.

Na slici 4.1 prikazan je postupak neposrednog adresiranja.



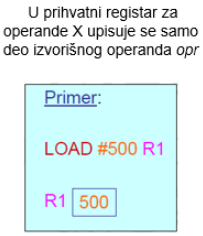
Slika 4.1: *Neposredno adresiranje*

Adresno polje u formatu instrukcije koje specificira izvorišni operand je podeljeno u dva dela. Jedan broj bitova definiše adresni mod (AM), dok ostali bitovi predstavljaju sam operand (*opr*). AM ukazuje da je izvorišni operand neposredno adresiran, pa binarni niz *opr* treba tumačiti kao vrednost operanda. Da bi se instrukcija izvršila, potrebno je da se vrednost *opr* upiše u prihvatni registar podatka (X) na ulazu u ALU.

Sledi primer neposrednog adresiranja operanda.

LOAD #500 R1 (# označava da je operand neposredno adresiran)

Operand 500 se direktno preuzima iz formata i upisuje u registar R1.



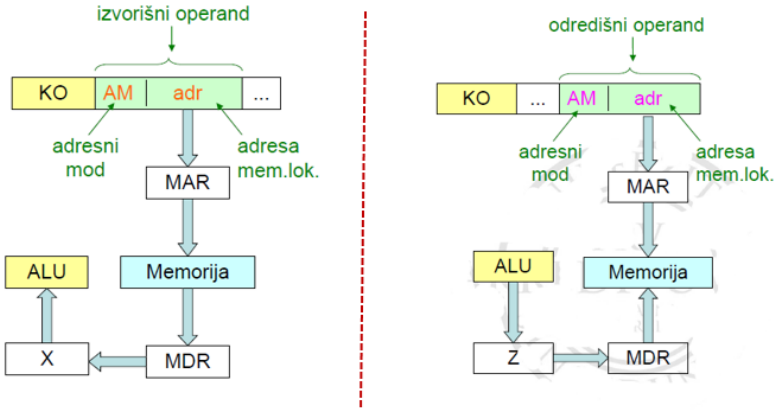
4.2 Direktno adresiranje

Direktno adresiranje je adresni mod koji podrazumeva da se u adresnom polju instrukcije navodi adresa na kojoj se nalazi operand. U zavisnosti od toga da li ta adresa predstavlja adresu memorijske lokacije ili adresu procesorskog registra, postoje dve varijante direktnog adresiranja: memorijsko direktno adresiranje i registarsko direktno adresiranje.

4.2.1 Memorijsko direktno adresiranje

Memorijsko direktno adresiranje je adresni mod kod koga se adresa memorijske lokacije u kojoj se nalazi operand navodi u formatu instrukcije. Ovaj mod je fleksibilniji od neposrednog adresiranja zato što se vrednost operanda može promeniti samo izmenom sadržaja određene memorijske lokacije, bez intervenisanja na instrukcijama. Na ovaj način se mogu specificirati i izvorišni i odredišni operandi.

Na slici 4.2 prikazan je postupak memorijskog direktnog adresiranja.



Slika 4.2: *Memorijsko direktno adresiranje izvorišnog (levo) i odredišnog (desno) operanda*

Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od adresnog moda (AM) i adrese (*adr*). AM ukazuje da se koristi memorijsko direktno adresiranje, pa binarni niz *adr* treba tumačiti kao adresu memorijske lokacije u kojoj se čuva operand. Da bi se instrukcija izvršila, najpre je potrebno je da se vrednost *adr* upiše u MAR i prosledi do memorije. Ako se adresira izvorišni operand, njegova vrednost se čita iz memorije, prosleđuje u MDR i dalje u prihvatni registar X na ulazu ALU, koja izvršava instrukciju. U slučaju odredišnog operanda, sadržaj u MAR predstavlja adresu ćelije u memoriji u koju treba upisati rezultat operacije. Dakle, kada ALU generiše rezultat, on se pojavljuje u prihvatnom registru Z na njenom izlazu, odakle se prosleđuje do MDR. Na kraju se sadržaj MDR upisuje u memorijsku ćeliju sa adresom iz MAR.

Sledi primer memorijskog direktnog adresiranja operanda.

LOAD 500 R1

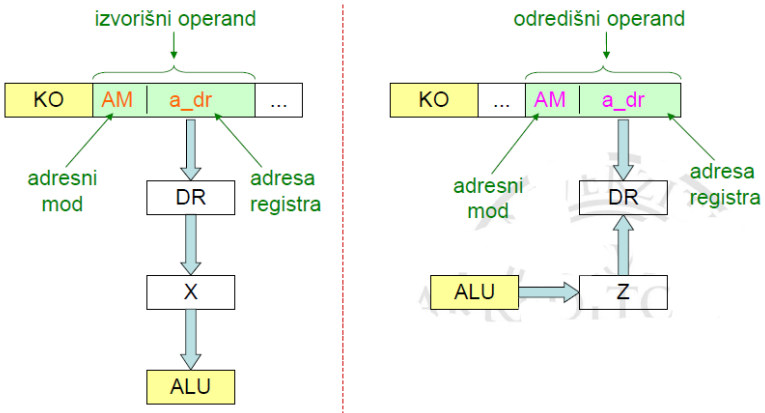
Operand se čita iz memorijske lokacije sa adresom 500 i upisuje u registar R1.

SLIKA

4.2.2 Registarsko direktno adresiranje

Registarsko direktno adresiranje je adresni mod kod koga se u formatu instrukcije navodi adresa registra (to može da bude jedan od registara podataka ili neki registar opšte namene) u kome se nalazi operand. Ovaj mod je, takođe, fleksibilniji od neposrednog adresiranja, a omogućava adresiranje i izvorišnih i odredišnih operanada.

Na slici 4.3 prikazan je postupak registarskog direktnog adresiranja.



Slika 4.3: *Registarsko direktno adresiranje izvorišnog (a) i odredišnog (b) operanda*

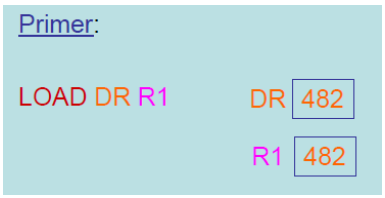
Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od adresnog moda (AM) i adrese registra (*a_reg*). AM ima istu ulogu kao i kod prethodno opisanih adresnih modova. Ako se adresira izvorišni operand, iz registra sa adresom *a_reg* pročita se vrednost operanda i prosledi u prihvatni registar X na ulazu ALU, koja zatim izvršava instrukciju nad tim operandom. U slučaju adresiranja odredišnog operanda, rezultat koji je generisala ALU se pojavljuje u izlaznom prihvatnom registru Z, a zatim se upisuje u registar čija je adresa *a_reg*.

Sledi primer registarskog direktnog adresiranja operanda.

LOAD DR R1

Operand se čita iz registra podatka DR i upisuje u registar R1.

SLIKA



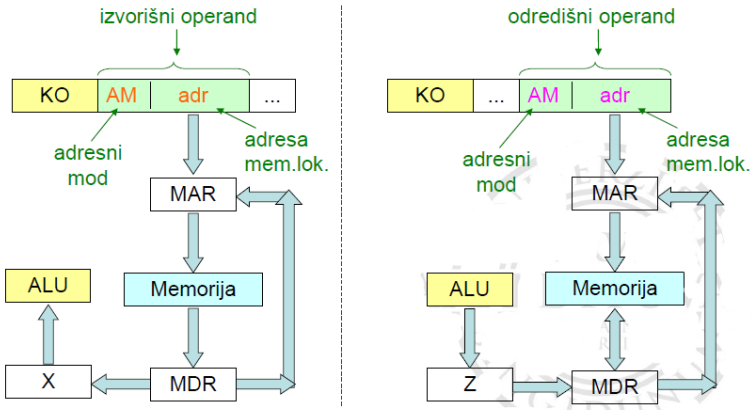
4.3 Indirektno adresiranje

Indirektno adresiranje je adresni mod koji podrazumeva da se u adresnom polju instrukcije navodi adresa na kojoj se nalazi adresa operanda. Adresa u polju instrukcije može da predstavlja adresu memorijske lokacije ili adresu procesorskog registra, pa postoje dve varijante indirektnog adresiranja: memorijsko indirektno adresiranje i registarsko indirektno adresiranje. U oba slučaja, operand se fizički nalazi u nekoj memorijskoj lokaciji.

4.3.1 Memorijsko indirektno adresiranje

Memorijsko indirektno adresiranje je adresni mod kod koga se u formatu instrukcije navodi adresa memorijske lokacije koja sadrži adresu memorijske lokacije u kojoj se nalazi operand. Ovaj mod se koristi za spcificiranje i izvorišnih i odredišnih operanada.

Na slici 4.4 prikazan je postupak memorijskog indirektnog adresiranja.



Slika 4.4: *Memorijsko indirektno adresiranje izvorišnog (levo) i odredišnog (desno) operanda*

Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od adresnog moda (AM) i adrese memorijske lokacije (*a_mem*). Da bi se instrukcija izvršila, najpre je potrebno je da se vrednost *a_mem* upiše u MAR i prosledi do memorije. Sa ove adrese, iz memorije se pročita sadržaj *a_op* i prosledi u MDR. Vrednost *a_op* predstavlja adresu memorijske lokacije za smeštaj operanda. Zatim se sadržaj MDR upisuje u MAR i prosleđuje do adresnih linija memorije. Ako se adresira izvorišni operand, sa adrese u MAR, iz memorije se čita operand koji se smešta u MDR i dalje prosleđuje u prihvatni registar X na ulazu ALU, koja zatim izvršava instrukciju. U slučaju odredišnog operanda, sadržaj u MAR predstavlja adresu

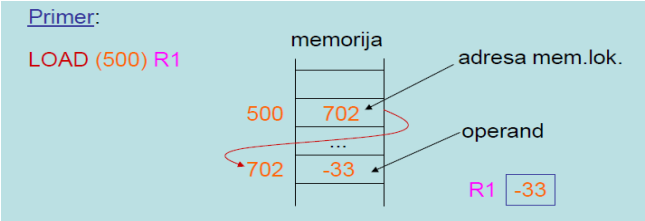
ćelije u memoriji u koju treba upisati rezultat operacije. Dakle, kada ALU generiše rezultat, on se pojavljuje u prihvatnom registru Z na njenom izlazu, odakle se prosleđuje u MDR. Sadržaj MDR se zatim upisuje u memorijsku ćeliju sa adresom u MAR.

Sledi primer memorijskog indirektnog adresiranja operanda.

LOAD (500) R1 (zagrade označavaju da je operand indirektno adresiran)

Operand se čita iz memorijske lokacije čija se adresa nalazi u memoriji u lokaciji sa adresom 500 i upisuje u registar R1.

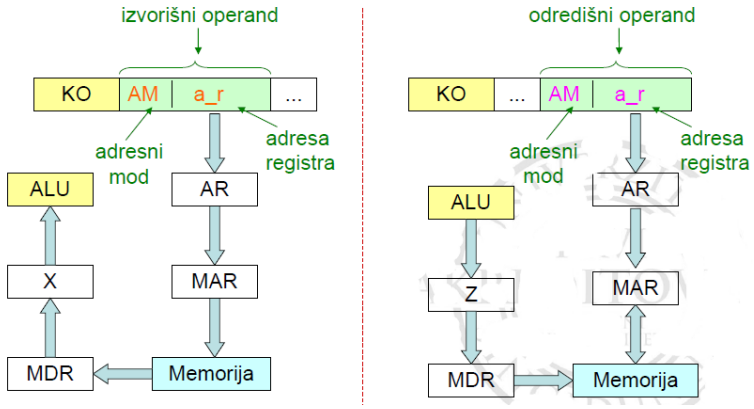
SLIKA



4.3.2 Registarsko indirektno adresiranje

Registarsko indirektno adresiranje je adresni mod kod koga se u formatu instrukcije navodi adresa registra koji sadrži adresu memorijske lokacije u kojoj se nalazi operand. Registar može da bude jedan od adresnih registara ili neki registar opšte namene. Ovaj mod se koristi za spcificiranje i izvorišnih i odredišnih operanada.

Na slici 4.5 prikazan je postupak registarskog indirektnog adresiranja.



Slika 4.5: *Registarsko indirektno adresiranje izvorišnog (a) i odredišnog (b) operanda*

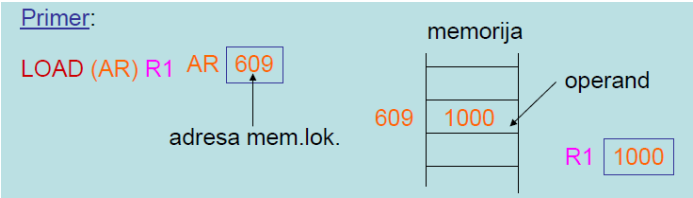
Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od adresnog moda (AM) i adrese registra (*a_reg*). Da bi se instrukcija izvršila, najpre se pročita sadržaj registra čija je adresa *a_reg*, upiše u MAR i prosledi do memorije. Ako se adresira izvorišni operand, sa adrese u MAR, iz memorije se pročita operand koji se smešta u MDR i dalje prosleđuje u prihvatni registar X na ulazu ALU, koja zatim izvršava instrukciju. U slučaju odredišnog operanda, sadržaj u MAR predstavlja adresu ćelije u memoriji u koju treba upisati rezultat operacije. Dakle, kada ALU generiše rezultat, on se pojavljuje u prihvatnom registru Z na njenom izlazu, odakle se prosleđuje u MDR. Sadržaj MDR se zatim upisuje u memorijsku ćeliju sa adresom u MAR.

Sledi primer registarskog indirektnog adresiranja operanda.

LOAD AR R1

Operand se čita iz memorijske lokacije čija se adresa nalazi u registru AR i upisuje u registar R1.

SLIKA



U nekim primenama, na primer pri realizaciji steka, postoji potreba da se adresiraju operandi na uzastopnim memorijskim lokacijama. Da bi se olakšalo ovakvo adresiranje, uvedene su dve varijante registarskog indirektnog adresiranja: adresiranje sa autoinkrementiranjem i adresiranje sa autodekrementiranjem.

Registarsko indirektno adresiranje sa autoinkrementiranjem

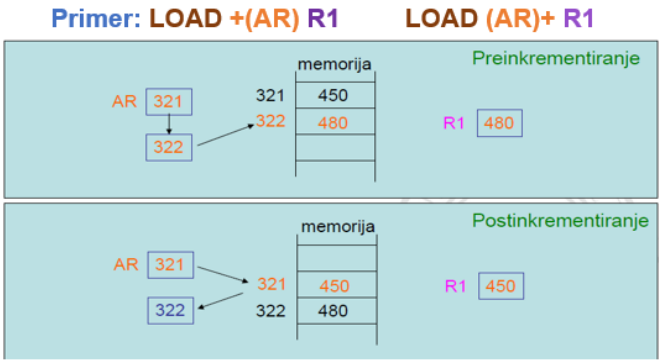
Kod adresiranja sa autoinkrementiranjem, u formatu instrukcije se navodi adresa registra koji sadrži adresu memorijske lokacije koja se automatski inkrementira (uvećava za 1). To znači da je nakon adresiranja prvog operanda promenjena vrednost u registru, pa se on može koristiti za adresiranje operanda iz naredne memorijske lokacije. U zavisnosti od trenutka u kome se vrši inkrementiranje, postoje dve mogućnosti:

- preinkrement adresiranje – sadržaj registra se najpre inkrementira, pa se nova vrednost uzima kao adresa memorijske lokacije operanda

- postinkrement adresiranje – sadržaj registra se uzima kao adresa memorijske lokacije operanda, pa se onda inkrementira

Mogućnosti adresiranja sa autoinkrementiranjem su ilustrovane na primeru instrukcije LOAD AR R1.

SLIKA



Kao što se vidi, sadržaji registra pre (321) i posle (322) adresiranja su isti u oba slučaja, iako su adresirani različiti operandi (u prvom slučaju 480, a u drugom 450).

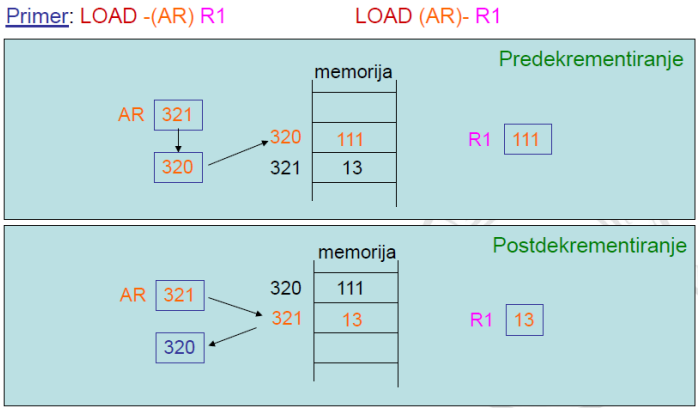
Registarsko indirektno adresiranje sa autodekrementiranjem

Kod adresiranja sa autodekrementiranjem, u formatu instrukcije se navodi adresa registra koji sadrži adresu memorijske lokacije koja se automatski dekrementira (umanjuje za 1). U zavisnosti od trenutka u kome se vrši dekrementiranje, postoje dve mogućnosti:

- predekrement adresiranje – sadržaj registra se najpre dekrementira, pa se nova vrednost uzima kao adresa memorijske lokacije operanda
- postdekrement adresiranje – sadržaj registra se uzima kao adresa memorijske lokacije operanda, pa se onda dekrementira

Mogućnosti adresiranja sa autodekrementiranjem su ilustrovane na primeru instrukcije LOAD AR R1.

SLIKA



Slično kao i kod adresiranja sa autoinkrementiranjem, sadržaji registra pre (321) i posle (320) adresiranja su isti u oba slučaja, iako su adresirani različiti operandi (u prvom slučaju 111, a u drugom 13).

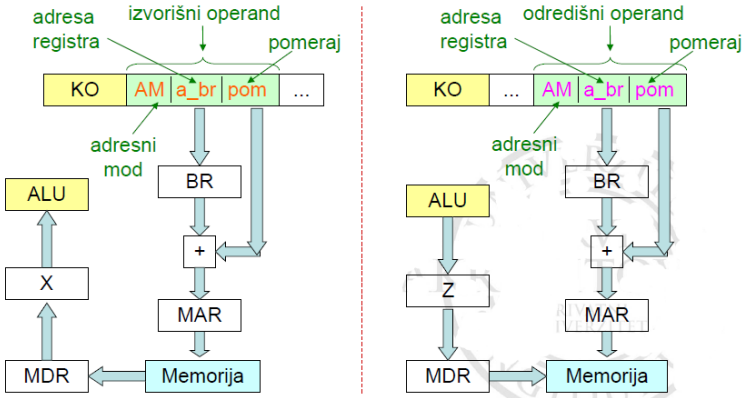
4.4 Adresiranja sa pomerajem

Načini adresiranja operanada kod kojih se, osim adresa registara, u adresnim poljima instrukcije navode i fiksne vrednosti, tzv. pomeraji, nazivaju se adresiranja sa pomerajem. Pri generisanju adrese operanda, pomeraj se dodaje (sabira) sa sadržajem registara. U zavisnosti od korišćenih registara, postoje različite vrste adresiranja sa pomerajem: bazno adresiranje, indeksno adresiranje, bazno-indeksno adresiranje, relativno adresiranje i registarsko indirektno adresiranje.

4.4.1 Bazno adresiranje

Bazno adresiranje sa pomerajem je adresiranje kod koga se u formatu instrukcije navode adresa baznog registra i pomeraj. Adresa memorijske lokacije u kojoj se nalazi operand se formira sabiranjem sadržaja baznog registra i pomeraja. Ovaj adresni mod se može koristiti za adresiranje i izvorišnih i odredišnih operanada kod procesora koji imaju bazne registre.

Na slici 4.6 prikazan je postupak baznog adresiranja sa pomerajem.



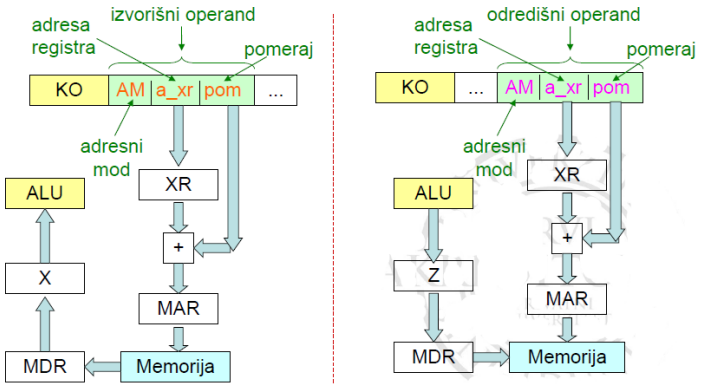
Slika 4.6: *Bazno adresiranje sa pomerajem*

Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od tri komponente: adresnog moda (AM), adrese baznog registra (*a_br*) i pomeraja (*pom*). Adresa memorijske lokacije za smeštaj operanda se dobija tako što se pročita sadržaj baznog registra čija je adresa *a_br*, sabere sa *pom*, upiše u MAR i prosledi do adresnih linija memorije. Ako se adresira izvorišni operand, sa adrese u MAR, iz memorije se pročita operand koji se smešta u MDR i dalje prosleđuje u prihvatni registar X na ulazu ALU, koja zatim izvršava instrukciju. U slučaju odredišnog operanda, sadržaj u MAR predstavlja adresu ćelije u memoriji u koju treba upisati rezultat operacije. Dakle, kada ALU generiše rezultat, on se pojavljuje u prihvatnom registru Z na njenom izlazu, odakle se prosleđuje u MDR. Sadržaj MDR se zatim upisuje u memorijsku ćeliju sa adresom u MAR.

4.4.2 Indeksno adresiranje

Indeksno adresiranje sa pomerajem je adresiranje kod koga se u formatu instrukcije navode adresa indeksnog registra i pomeraj. Adresa memorijske lokacije u kojoj se nalazi operand se formira sabiranjem sadržaja indeksnog registra i pomeraja. Ovaj adresni mod se može koristiti za adresiranje i izvorišnih i odredišnih operanada kod procesora koji imaju indeksne registre.

Na slici 4.7 prikazan je postupak indeksnog adresiranja sa pomerajem.



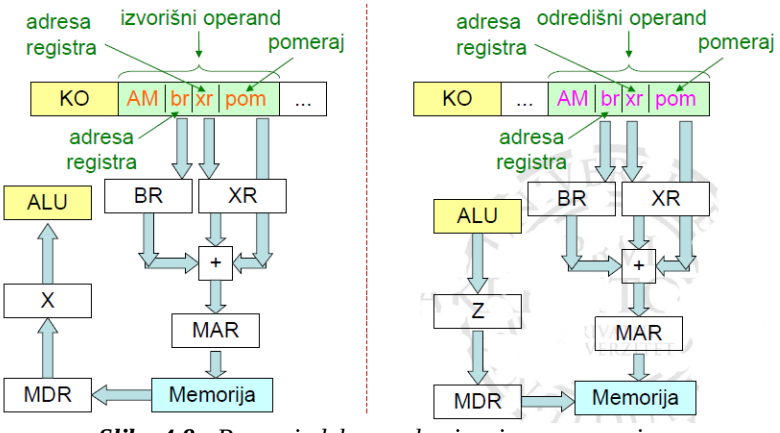
Slika 4.7: *Indeksno adresiranje sa pomerajem*

Kod ovog načina adresiranja, postupak je isti kao kod baznog adresiranja sa pomerajem, s tim što se umesto baznog, koristi indeksni registar.

4.4.3 Bazno-indeksno adresiranje

Bazno-indeksno adresiranje sa pomerajem je adresiranje kod koga se u formatu instrukcije navode adresa baznog registra, adresa indeksnog registra i pomeraj. Adresa memorijske lokacije u kojoj se nalazi operand se formira sabiranjem sadržaja baznog registra, sadržaja indeksnog registra i pomeraja. Ovaj adresni mod se može koristiti za adresiranje i izvorišnih i odredišnih operanada u slučaju procesora koji imaju bazne i indeksne registre. Kod procesora koji imaju samo registre opšte namene, sa pomerajem se sabiraju sadržaji dva registra opšte namene.

Na slici 4.8 prikazan je postupak bazno-indeksnog adresiranja sa pomerajem.



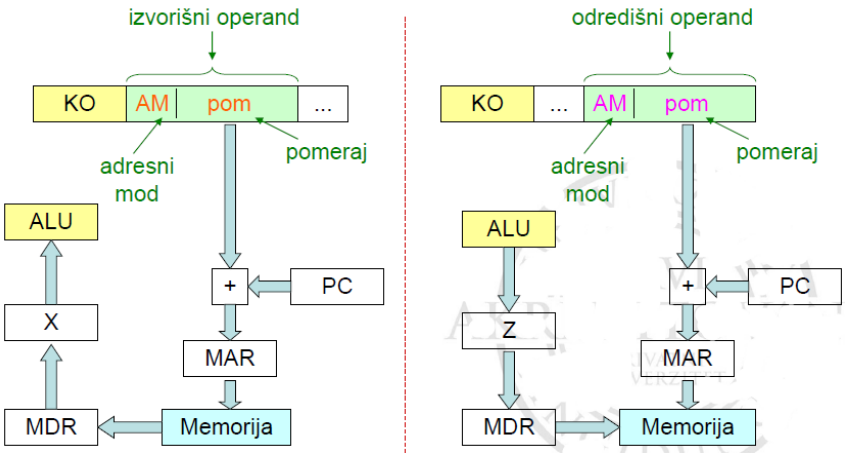
Slika 4.8: *Bazno-indeksno adresiranje sa pomerajem*

Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od četiri komponente: adresnog moda (AM), adrese baznog registra (*a_br*), adrese indeksnog registra (*a_xr*) i pomeraja (*pom*). Adresa memorijske lokacije rezervisane za operand se dobija tako što se pročitaju sadržaji baznog registra čija je adresa *a_br* i indeksnog registra čija je adresa *a_xr*, saberu sa *pom*, pa se rezultat upiše u MAR i prosledi do adresnih linija memorije. Ako se adresira izvorišni operand, sa adrese u MAR, iz memorije se pročita operand koji se smešta u MDR i dalje prosleđuje u prihvatni registar X na ulazu ALU, koja zatim izvršava instrukciju. U slučaju odredišnog operanda, sadržaj u MAR predstavlja adresu ćelije u memoriji u koju treba upisati rezultat operacije. Dakle, kada ALU generiše rezultat, on se pojavljuje u prihvatnom registru Z na njenom izlazu, odakle se prosleđuje u MDR. Sadržaj MDR se zatim upisuje u memorijsku ćeliju sa adresom u MAR.

4.4.4 Relativno adresiranje

Relativno adresiranje sa pomerajem je adresiranje kod koga se u formatu instrukcije navodi samo pomeraj, a adresa memorijske lokacije u kojoj se nalazi operand se formira sabiranjem pomeraja sa sadržajem programskog brojača. Ovaj adresni mod se može koristiti za adresiranje i izvorišnih i odredišnih operanada.

Na slici 4.9 prikazan je postupak relativnog adresiranja sa pomerajem.



Slika 4.9: *Relativno adresiranje sa pomerajem*

Adresno polje operanda (izvorišnog ili odredišnog) sastoji se od dve komponente: adresnog moda (AM) i pomeraja (*pom*). Adresa memorijske lokacije za smeštaj operanda se dobija tako što se pročita sadržaj programskog brojača, sabere sa pomerajem *pom*, upiše u MAR i prosledi do adresnih linija memorije.

Ako se adresira izvorišni operand, sa adrese u MAR, iz memorije se pročita operand koji se smešta u MDR i dalje prosleđuje u prihvatni registar X na ulazu ALU, koja zatim izvršava instrukciju. U slučaju odredišnog operanda, sadržaj u MAR predstavlja adresu ćelije u memoriji u koju treba upisati rezultat operacije. Dakle, kada ALU generiše rezultat, on se pojavljuje u prihvatnom registru Z na njenom izlazu, odakle se prosleđuje u MDR. Sadržaj MDR se zatim upisuje u memorijsku ćeliju sa adresom u MAR.

4.4.5 Registarsko indirektno adresiranje

Registarsko indirektno adresiranje sa pomerajem je adresiranje kod koga se u formatu instrukcije navode adresa registra opšte namene i pomeraj. Adresa memorijske lokacije u kojoj se nalazi operand se formira sabiranjem sadržaja registra opšte namene i pomeraja. Ovaj adresni mod se može koristiti za adresiranje i izvorišnih i odredišnih operanada kod procesora koji imaju registre opšte namene.

Postupak adresiranja kod ovog adresnog moda je isti kao što je opisano u slučaju baznog ili indeksnog adresiranja sa pomerajem, s tim što se umesto baznog, odnosno indeksnog registra koristi registar opšte namene.

4.5 Primena adresnih modova

Sve češća potreba za obradom velikih količina raznovrsnih podataka, dovela je do povećanja složenosti softvera. Postalo je veoma važno da se podacima što brže pristupa, tj. da se oni adresiraju na što efikasniji način. To je uslovilo uvođenje velikog broja adresnih modova primenljivih u različitim situacijama.

Najjednostavniji adresni mod, neposredno adresiranje, pogodan je za definisanje konstanti.

Direktno memorijsko adresiranje se koristi za adresiranje jednostavnih varijabli, tako što imena promenljivih direktno adresiraju memorijske lokacije.

Relativno adresiranje je pogodno pri programskim skokovima, jer se tipični skokovi izvode na obližnje instrukcije. Zadavanjem pomeraja, mogu se ostvariti skokovi i unapred i unazad po instrukcijama programa. Zahvaljujući relativnom adresiranju, programski kod je poziciono nezavisan, tj. može se smestiti bilo gde u memoriji bez potrebe za podešavanjem adresa.

Registarsko indirektno adresiranje se koristi za adresiranje nizova, tabela i sličnih struktura podataka, zato što pruža mogućnost automatskog inkrementiranja ili dekrementiranja adresa. Zbog manjeg broja registara, adrese registara su kraće

od adresa memorijskih lokacija, pa način adresiranja pomoću registara štedi memoriju jer je broj bitova u instrukcijama manji.

Indeksno adresiranje je efikasno pri indeksiranju članova niza, dok se bazno adresiranje koristi za pristup poljima neke strukture ili nekog objekta (adresa strukture se smešta u bazni registar, a pomeraj određuje polje unutar strukture).

Bazno-indeksno adresiranje je pogodno pri radu sa dvodimenzionalnim nizovima. U ovom slučaju, adresa niza se zadaje pomerajem, dok se adresa vrste smešta u bazni registar, a adresa elementa vrste, tj. indeks kolone u indeksni registar. Ovaj adresni mod se može koristiti i pri radu sa nizovima čiji su elementi strukture ili objekti. Tada se obično adresa niza smešta u bazni registar, relativna adresa strukture u indeksni registar, a pomeraj predstavlja adesu polja u strukturi.

Vežbanja

1. Logičke funkcije (Y) prikazane datim kombinacionim tablicama predstaviti:

5 Instrukcijski set

Instrukcijski set je skup instrukcija kojima se specificiraju operacije koje procesor može da izvrši. Izbor instrukcija u setu zavisi od vrsta aplikacija za koje je procesor namenjen. Procesori opšte namene uglavnom podržavaju standardne instrukcije, dok procesori specijalne namene implementiraju i nestandardne instrukcije.

Zbog velikog broja, standarne instrukcije su tematski svrstane u sledeće grupe: aritmetičke instrukcije, logičke instrukcije, pomeračke instrukcije, instrukcije prenosa i instrukcije skoka. Standarne instrukcije su detaljno predstavljene u ovom poglavlju.

Nestandardne instrukcije se definišu prema konkretnim potrebama. To su, na primer, instrukcije nad celobrojnim veličinama promenljive dužine, instrukcije za rad sa stringovima, instrukcije kontrole petlji, itd.

5.1 Aritmetičke instrukcije

Aritmetičke instrukcije su instrukcije koje realizuju standardne aritmetičke operacije: sabiranje, oduzimannje, množenje i deljenje.

Instrukcije sabiranja

U tabeli 5.1 prikazane su standardne instrukcije sabiranja:

- ADD, koja aritmetički sabira dva izvorišna operanda i zbir smešta u odredišni operand
- INC, koja inkrementira (uvećava za 1) vrednost izvorišnog operanda i rezultat smešta u odredišni operand

U tabeli 5.1 (kao i u ostalim tabelama u ovom poglavlju) su korišćene sledeće oznake:

n.a. – neposredno adresiran
ACC – akumulator (procesorski registar)
TS – vrh steka (*top of stack*)

Tabela 5.1: *Instrukcije sabiranja*

Operacija	Instrukcija	Izvorišni operandi	Odredišni operand	Napomena
sabiranje	ADD a, b, c	a i b	c	c nije n.a.
	ADD a, b	a i b	a (ili b)	a (ili b)nije n.a.
	ADD a	ACC i a	ACC	
	ADD	TS i TS	TS	
inkrementiranje	INC a, b	a	b	b nije n.a.
	INC a	a	a	a nije n.a.
	INC	TS	TS	

Instrukcije oduzimanja

U tabeli 5.2 prikazane su standardne instrukcije oduzimanja:

- SUB, koja aritmetički oduzima drugi izvorišni operand od prvog i razliku smešta u odredišni operand
- DEC, koja dekrementira (umanjuje za 1) vrednost izvorišnog operanda i rezultat smešta u odredišni operand
- CMP, koja poredi izvorišne operande tako što od prvog oduzima drugi, pa dobijeni rezultat nigde ne upisuje, već ga samo proverava i na osnovu njega postavlja indikatore N, Z, C i V statusnog registra; indikator N ima vrednost 1 ako je rezultat negativan, indikator Z ima vrednost 1 ako je rezultat 0, indikator C ima vrednost 1 ako je bilo pozanjmice pri oduzimanju, a indikator V ima vrednost 1 ako je došlo do prekoračenja opsega; svrha postavljanja indikatora je da se omogući realizacija uslovnih skokova u programu (instrukcija uslovnog skoka proverava indikatore i utvrđuje da li je uslov za skok ispunjen ili nije; ako je uslov ispunjen, ostvaruje se skok, a ako nije, nastavlja se sa sekvencijalnim izvršavanjem instrukcija programa).

Tabela 5.2: *Instrukcije oduzimanja*

Operacija	Instrukcija	Izvorišni operandi	Odredišni operand	Napomena
oduzimanje	SUB a, b, c	a i b	c	c nije n.a.
	SUB a, b	a i b	a (ili b)	a (ili b)nije n.a.
	SUB a	ACC i a	ACC	
	SUB	TS i TS	TS	
dekrementiranje	DEC a, b	a	b	b nije n.a.
	DEC a	a	a	a nije n.a.
	DEC	TS	TS	
aritmetičko poređenje	CMP a, b, c	a i b		c se ne koristi
	CMP a, b	a i b		
	CMP a	ACC i a		
	CMP	TS i TS		

Instrukcija množenja

U tabeli 5.3 prikazana je standardna instrukcija množenja:

- MUL, koja aritmetički množi izvorišne operande i proizvod smešta na odredište

Operacija	Instrukcija	Izvorišni operandi	Odredišni operand	Napomena
množenje	MUL a, b, c	a i b	c	c nije n.a.
	MUL a, b	a i b	a (ili b)	a (ili b)nije n.a.
	MUL a	ACC i a	ACC	
	MUL	TS i TS	TS	

Tabela 5.3 Instrukcija množenja

Instrukcija deljenja

U tabeli 5.4 prikazana je standardna instrukcija deljenja:

- DIV, koja deli prvi izvorišni operand sa drugim i količnik smešta na odredište.

Tabela 5.4: *Instrukcija deljenja*

Operacija	Instrukcija	Izvorišni operandi	Odredišni operand	Napomena
deljenje	DIV a, b, c	a i b	c	c nije n.a.
	DIV a, b	a i b	a (ili b)	a (ili b)nije n.a.
	DIV a	ACC i a	ACC	
	DIV	TS i TS	TS	

5.2 Logičke instrukcije

Logičke instrukcije su instrukcije koje realizuju standardne logičke operacije: logičko množenje (I operacija), logičko sabiranje (ILI operacija), ekskluzivno sabiranje (ekskluzivno ILI operacija) i negaciju (NE operacija). Osim ovih, biće predstavljena još jedna logička instrukcija, a to je logičko upoređivanje.

U tabeli 5.5 prikazane su sledeće logičke instrukcije:

- AND, koja logički množi dva izvorišna operanda i rezultat smešta u odredišni operand
- OR, koja logički sabira dva izvorišna operanda i rezultat smešta u odredišni operand
- XOR, koja ekskluzivno sabira dva izvorišna operanda i rezultat smešta u odredišni operand
- NOT, koja komplementira vrednost izvorišnog operanda i rezultat smešta u odredišni operand
- TST, koja izvršava operaciju logičkog množenja nad izvorišnim operandima, dobijeni rezultat nigde ne upisuje, već ga samo proverava i na osnovu njega postavlja indikatore N, Z, C i V statusnog registra

Sastavni deo izvršavanja logičkih instrukcija AND, OR, XOR i NOT je postavljanje indikatora statusnog registra na osnovu dobijenog rezultata. Indikator N se postavlja na vrednost najstarijeg bita rezultata operacije, indikator Z ima vrednost 1 ako je rezultat operacije 0, dok se indikatori C i V postavljaju na fiksnu vrednost 0. Indikatori se koriste kada se u programu iza logičke instrukcije nalazi instrukcija uslovnog skoka. Pri tome, od instrukcija uslovnog skoka, ima smisla koristiti samo skok na jednako (Z = 1) i nejednako (Z = 0). Ostale instrukcije uslovnog skoka interpretiraju rezultat kao celobrojnu vrednost bez znaka ili

vrednost sa znakom u drugom komplementu, a on to nije, pa ovi skokovi ne mogu da se koriste.

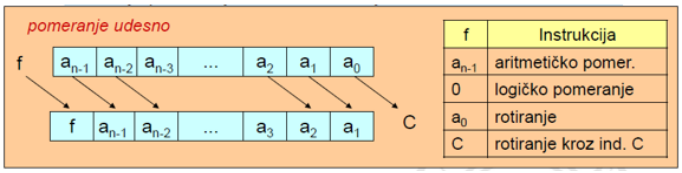
Tabela 5.5: *Logičke instrukcije*

Operacija	Instrukcija	Izvorišni operandi	Odredišni operand	Napomena
logičko množenje	AND a, b, c	a i b	c	c nije n.a.
	AND a, b	a i b	a (ili b)	a (ili b)nije n.a.
	AND a	ACC i a	ACC	
	AND	TS i TS	TS	
logičko sabiranje	OR a, b, c	a i b	c	c nije n.a.
	OR a, b	a i b	a (ili b)	a (ili b)nije n.a.
	OR a	ACC i a	ACC	
	OR	TS i TS	TS	
ekskluzivno logičko sabiranje	XOR a, b, c	a i b	c	c nije n.a.
	XOR a, b	a i b	a (ili b)	a (ili b)nije n.a.
	XOR a	ACC i a	ACC	
	XOR	TS i TS	TS	
negacija	NOT a, b	a	b	b nije n.a.
	NOT a	a	a	a nije n.a.
	NOT	TS	TS	
logičko upoređivanje	TST a, b, c	a i b		c se ne koristi
	TST a, b	a i b		
	TST a	ACC i a		
	TST	TS i TS		

5.3 Pomeračke instrukcije

Pomeračke instrukcije su instrukcije koje realizuju pomeranje binarne reči za jedno mesto ulevo ili udesno.

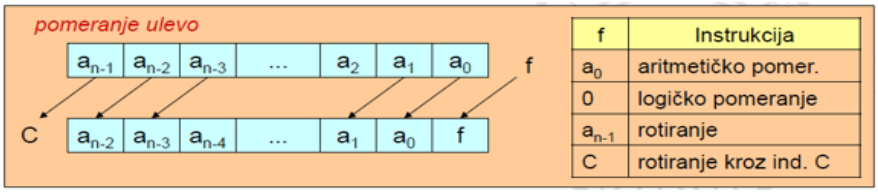
Na slici 5.1 prikazan je postupak pomeranja binarne reči $a_{n-1}...a_0$ udesno.



Slika 5.1: *Pomeranje binarne reči udesno*

Kao što se vidi, bitovi $a_{n-1}...a_1$ se pomeraju za poziciju niže. Bit a_0 se upisuje u inidkator C statusnog registra. Najstariji bit u rezultatu, f, može da dobije jednu od četiri vrednosti: a_{n-1}, 0, a_0 ili vrednost indikatora C (pre upisa a_0). U zavisnosti od toga koja od četiri vrednosti se upisuje u f, postoje četiri instrukcije pomeranja udesno: aritmetičko pomeranje udesno, logičko pomeranje udesno, rotiranje udesno i rotiranje udesno kroz indikator C (vrednost f za svaku vrstu instrukcije data je u tabeli na slici).

Na sličan način se realizuje i pomeranje binarne reči ulevo (slika 5.2).



Slika 5.2: *Pomeranje binarne reči ulevo*

Bitovi $a_{n-2}...a_0$ se pomeraju za poziciju više. Bit a_{n-1} se upisuje u inidkator C statusnog registra. Najniži bit u rezultatu, f, može da dobije jednu od četiri vrednosti: 0, 0, a_{n-1} ili vrednost indikatora C (pre upisa a_{n-1}), tako da postoje četiri instrukcije pomeranja ulevo: aritmetičko pomeranje ulevo, logičko pomeranje ulevo, rotiranje ulevo i rotiranje ulevo kroz indikator C (vrednost f za svaku vrstu instrukcije data je u tabeli na slici).

U tabeli 5.6 prikazane su sledeće pomeračke instrukcije:

- ASR (*Arithmetic Shift Right*), LSR (*Logic Shift Right*), ROR (*Rotate Right*) i RORC (*Rotate Through Carry Right*), koje izvorišni operand pomeraju za jedno mesto udesno i rezultat smeštaju u odredišni operand; pri izvršenju instrukcija, na osnovu rezultata pomeranja, postavljaju se indikatori statusnog registra; indikator N ima vrednost najstarijeg bita rezultata, indikator Z ima vrednost 1 ako je rezultat 0, indikator C ima vrednost najnižeg bita izvorišnog operanda, a indikator V ima vrednost 0
- ASL (*Arithmetic Shift Left*), LSL (*Logic Shift Left*), ROL (*Rotate Left*) i ROLC (*Rotate Through Carry Left*), koje izvorišni operand pomeraju za jedno mesto ulevo i rezultat smeštaju u odredišni operand; pri izvršenju instrukcija, na osnovu rezultata pomeranja, postavljaju se indikatori statusnog registra; indikator N ima vrednost najstarijeg bita rezultata, indikator Z ima vrednost 1 ako je rezultat 0, indikator C ima vrednost najstarijeg bita izvorišnog operanda, a indikator V ima vrednost 0.

Tabela 5.6: *Pomeračke instrukcije*

Operacija	Instrukcija	Izvorišni operandi	Odredišni operand	Napomena
aritmetičko pomeranje udesno	ASR a, b, c	a	b	c se ne koristi
	ASR a, b	a	b	
	ASR a	a	a	
	ASR	TS	TS	
logičko pomeranje udesno	LSR a, b, c	a	b	c se ne koristi
	LSR a, b	a	b	
	LSR a	a	a	
	LSR	TS	TS	
rotacija udesno	ROR a, b, c	a	b	c se ne koristi
	ROR a, b	a	b	
	ROR a	a	a	
	ROR	TS	TS	
rotacija udesno kroz indikator C	RORC a, b, c	a	b	c se ne koristi
	RORC a, b	a	b	
	RORC a	a	a	
	RORC	TS	TS	
aritmetičko pomeranje ulevo	ASL a, b, c	a	b	c se ne koristi
	ASL a, b	a	b	
	ASL a	a	a	
	ASL	TS	TS	
logičko pomeranje ulevo	LSL a, b, c	a	b	c se ne koristi
	LSL a, b	a	b	
	LSL a	a	a	
	LSL	TS	TS	
rotacija ulevo	ROL a, b, c	a	b	c se ne koristi
	ROL a, b	a	b	
	ROL a	a	a	
	ROL	TS	TS	
rotacija uvelo kroz indikator C	ROLC a, b, c	a	b	c se ne koristi
	ROLC a, b	a	b	
	ROLC a	a	a	
	ROLC	TS	TS	

5.4 Instrukcije prenosa

Instrukcije prenosa su instrukcije koje realizuju prenos podatka sa jednog mesta u računaru na drugo. Podaci se mogu naći na različitim mestima:

- u memorijskim lokacijama - ML
- u procesorskim registrima - PR
- u instrukciji (neposredno adresiranje) - NAI
- u registrima kontrolera periferija i to
  - registrima koji se tretiraju isto kao memorijske lokacije (U/I adresni prostor je memorijski preslikan) - RKP
  - registrima koji se tretiraju posebno (U/I i memorijski adresni prostor su razdvojeni) - RKR
- u akumulatoru (kod jednoadresnih instrukcija) - AK
- na steku (kod nulaadresnih instrukcija) - ST

U tabeli 5.7 prikazane su standarnde instrukcije prenosa:

- MOV, koja izvorišni operand prenosi na odredište; izvorišni operanad može da bude u ML, PR, RKP ili NAI, a odredišni operand se nalazi u ML, PR ili RKP
- IN, koja vrši prenos izvorišnog operanda iz RKR na odredište; koristi se kod 2-adresnih i 1-adresnih procesora sa razdvojenim U/I i memorijskim adresnim prostorima; odredišni operand je u PR (ako je instrukcija 2-adresna) ili u AK (ako je instrukcija 1-adresna)
- OUT, koja vrši prenos izvorišnog operanda na odredište u RKR; koristi se kod 2-adresnih i 1-adresnih procesora sa razdvojenim U/I i memorijskim adresnim prostorima; izvorišni operand je u PR (ako je instrukcija 2-adresna) ili u AK (ako je instrukcija 1-adresna)
- LOAD, koja vrši prenos izvorišnog operanda u akumulator (AK); izvorišni operanad može da bude u ML, PR, RKP ili NAI
- STORE, koja obavlja prenos sadržaja akumulatora na odredište; odredišni operand se nalazi u ML, PR ili RKP

- PUSH, koja vrši prenos izvorišnog operanda na stek (ST); izvorišni operanad može da bude u ML, PR, RKP ili NAI
- POP, koja sadržaj sa vrha steka prenosi na odredište; odredišni operand može da bude u ML, PR ili RKP

Tabela 5.7: *Instrukcije prenosa*

Operacija	Instrukcija	Izvorišni operand	Odredišni operand	Napomena
prenos u/iz memorije/registara	MOV a, b	a	b	a→b
prenos do/od periferije	IN a, b	a	b	RKR→PR
	OUT a, b	a	b	PR→RKR
	IN a	a	ACC	RKR→ACC
	OUT a	ACC	a	ACC→RKR
prenos u/iz akumulatora	LOAD a	a	ACC	a→ACC
	STORE a	ACC	a	ACC→a
prenos na/sa steka	PUSH a	a	TS	a→TS
	POP a	TS	a	TS→a

5.5 Instrukcije skoka

Instrukcije skoka su instrukcije koje modifikuju sadržaj programskog brojača (PC) i na taj način omogućavaju nesekvencijalno izvršavanje programa, što je korisno u mnogim situacijama. Novi sadržaj PC predstavlja adresu memorijske lokacije u kojoj se nalazi naredna instrukcija koju treba izvršiti. Nakon izvršenja ove instrukcije, nastavlja se sa sekvencijalnim izvršavanjem programa od date adrese uz inkrementiranje PC.

Instrukcije skoka se mogu klasifikovati u sledeće grupe:

- instrukcije bezuslovnog skoka
- instrukcije uslovnog skoka
- instrukcije za rad sa potprogramima
- instrukcije za rad sa prekidnim rutinama

Instrukcije bezuslovnog skoka

Instrukcije bezuslovnog skoka modifikuju PC uvek, bez postavljanja bilo kakvih uslova. To su sledeće instrukcije:

JMP *a*

Ova instrukcija se izvršava tako što se u PC upisuje vrednost adresnog polja *a*. Vrednost *a* predstavlja adresu memorijske lokacije u kojoj se nalazi instrukcija koju treba sledeću izvršiti. Instrukcija se koristi kada je tokom prevođenja (kompajliranja) programa poznata adresa instrukcije na koju treba skočiti.

JMPIND *a*

U ovoj instrukciji, polje *a* specificira adresu memorijske lokacije u kojoj se nalazi instrukcija koju treba sledeću izvršiti. Specifikacija zavisi od korišćenog adresnog moda. Instrukcija se izvršava tako što se najpre računa adresa instrukcije, a zatim se rezultat upisuje u PC. Instrukcija se koristi ukoliko u vreme prevođenja programa nije poznata adresa instrukcije na koju treba skočiti, već se ona prethodno računa tokom izvršavanja programa.

Sledi primer koji ilustruje razliku između navedenih instrukcija. Ako je adresa instrukcije na koju treba skočiti poznata, na pr. 145, koristi se instrukcija JMP 145. Po izvršenju ove instrukcije skoka, u PC će biti upisana vrednost 145 i procesor će preći na izvršavanje instrukcije na adresi 145. Ukoliko adresa instrukcije na koju treba skočiti nije poznata, mora se koristiti instrukcija skoka JMPIND a. U ovom slučaju, adresa se mora izračunati izvršavanjem instrukcija koje u programu prethode instrukciji JMPIND i smestiti na odgovarajuće mesto, na pr. u adresni registar AR3 ili u memorijsku lokaciju 1030. Neka je izračunata adresa 889. Ako se ona upiše u AR3, onda se poljem a u JMPIND instrukciji mora specificirati registarsko indirektno adresiranje uz primenu AR3. Ako se 889 upiše u memorijsku lokaciju 1030, onda se poljem a mora specificirati memorijsko indirektno adresiranje i adresa 1030. U oba slučaja, nakon izvršenja instrukcije skoka, u PC je upisana vrednost 889.

Instrukcije uslovnog skoka

Instrukcije uslovnog skoka modifikuju PC samo ukoliko je ispunjen neki uslov. Opšti format ovih instrukcija je:

CO *p*

CO označava kod operacije (ovo je samo opšti oblik, dok su konkretni kodovi operacija dati u tabeli 5.8), a *p* vrednost pomeraja sa kojim treba napraviti relativan skok u odnosu na tekući sadržaj PC, samo ukoliko je uslov ispunjen. Na primer,

ako je trenutna vrednost u PC 625, a p ima vrednost 12, to znači da se sledeća instrukcija koju treba izvršiti nalazi u memorijskoj lokaciji sa adresom 637.

Pošto je vrednost *p* u drugom komplementu, skok može biti unapred (ako je *p* pozitivan broj) ili unazad (ako je *p* negativan broj). Uslov za skok je specificiran kodom operacije i predstavlja proveru indikatora N, Z, C i V statusnog registra.

Instrukcije uslovnog skoka se izvršavaju u tri koraka:

- sadržaj PC se sabira sa *p* i tako dobija adresa memorijske lokacije u kojoj se nalazi instrukcija koju treba sledeću izvršiti (na koju treba skočiti)
- na osnovu vrednosti indikatora N, Z, C i V proverava se da li je uslov za skok ispunjen ili nije
- ako je uslov ispunjen, adresa izračunata u prvom koraku se upisuje u PC

U tabeli 5.8 prikazane su instrukcije uslovnog skoka:

Instrukcija	Opis	Uslov za skok
BEQL *p*	skok na jednako	Z = 1
BNEQ *p*	skok na nejednako	Z = 0
BGRTU *p*	skok na veće nego (bez znaka)	C v Z = 0
BGREU *p*	skok na veće nego ili jednako (bez znaka)	C = 0
BLSSU *p*	skok na manje nego (bez znaka)	C = 1
BLEQU *p*	skok na manje nego ili jednako (bez znaka)	C v Z = 1
BGRT *p*	skok na veće nego (sa znakom)	(N ⊕ V) v Z = 0
BGRE *p*	skok na veće nego ili jednako (sa znakom)	N ⊕ V = 0
BLSS *p*	skok na manje nego (sa znakom)	N ⊕ V = 1
BLEQ *p*	skok na manje nego ili jednako (sa znakom)	(N ⊕ V) v Z = 1
BNEG *p*	skok na N = 1	N = 1
BNNG *p*	skok na N = 0	N = 0
BOVF *p*	skok na V = 1	V = 1
BNVF *p*	skok na V = 0	V = 0

Tabela 5.8 Instrukcije uslovnog skoka

Osim navedenih, postoje i instrukcije uslovnog skoka kod kojih se adresa instrukcije na koju treba skočiti direktno navodi u formatu instrukcije skoka, umesto pomeraja. Ove instrukcije realizuju tzv. apsolutni skok. One imaju vrlo sličan format kao instrukcije koje realizuju relativan skok (tabela 5.8) uz dve razlike:

- kodovi operacije ne počinju slovom B (*Branch*), već slovom J (*Jump*)
- *p* ne predstavlja pomeraj, već adresu instrukcije na koju treba skočiti

Primeri ovih instrukcija su: JEQL *adr*, JLSS *adr*, JNVF *adr*, itd.

Instrukcije za rad sa potprogramima

Instrukcije za rad sa potprogramima čine instrukcija skoka na potprogram (JSR) i instrukcija povratka iz potprograma (RTS), čiji su opisi dati u nastavku:

JSR *a*

Vrednost adresnog polja *a* predstavlja adresu memorijske lokacije u kojoj se nalazi prva instrukcija potprograma. Instrukcija JSR se izvršava tako što se najpre tekući sadržaj PC prenese na stek, a zatim se vrednost *a* upiše u PC. Prenos na stek je neophodan zbog ispravnog povratka iz potprograma.

RTS

Ova instrukcija mora da bude poslednja instrukcija u potprogramu. Izvršava se tako što se sadržaj sa steka upisuje u PC. Očigledno je da ovaj sadržaj mora da odgovara sadržaju koji je JSR instrukcijom prenet na stek.

Instrukcije za rad sa prekidnim rutinama

Instrukcije za rad sa prekidnim rutinama su instrukcija skoka na prekidnu rutinu (INT) i instrukcija povratka iz prekidne rutine (RTI) koje imaju sledeće formate:

INT *a*

Vrednost adresnog polja *a* predstavlja adresu memorijske lokacije u kojoj se nalazi prva instrukcija prekidne rutine. Instrukcija INT se izvršava tako što se tekući sadržaji PC i statusnog registra prenesu na stek, a zatim se vrednost *a* upiše u PC. U nekim realizacijama mehanizma prekida, adrese prekidnih rutina se nalaze u tabeli prekida. U ovom slučaju, *a* predstavlja broj ulaza u tabeli prekida iz koga treba pročitati adresu prekidne rutine i smestiti je u PC.

RTI

Ova instrukcija mora da bude poslednja instrukcija u prekidnoj rutini. Izvršava se tako što se sadržaji sa steka upisuju u statusni registar i PC. Ovi sadržaji moraju da odgovaraju sadržajima koji su INT instrukcijom preneti na stek.

Vežbanja

1. Šta su koderi i čemu služe? Objasniti princip rada kodera. Koje vrste kodera postoje i u čemu se one razlikuju?

6 Programiranje

Računarski program može biti napisan na različitim nivoima apstrakcije. Najvišem nivou apstrakcije odgovaraju programi pisani na višim programskim jezicima kao što su Java ili C++. Ovi programi su mašinski nezavisni i za njihovo pisanje nije neophodno detaljno poznavanje arhitekture računara. Međutim, procesor ne može direktno da izvršava ove programe. Svaki procesor može da izvršava isključivo programe pisane na mašinskom jeziku koji je specifičan za njegovu arhitekturu. Programi na mašinskom jeziku predstavljaju najniži nivo apstrakcije u odnosu na arhitekturu sistema (nulti nivo). Zbog problema pri pisanju programa na mašinskom jeziku (brojni su detalji o kojima se mora voditi računa), uveden je asemblerski jezik kao simbolička reprezentacija mašinskog jezika. On je jednostavniji za programiranje, a programi pisani na asemblerskom jeziku odgovaraju nešto višem nivou apstrakcije u odnosu na programe pisane na mašinskom jeziku.

6.1 Mašinski jezik

Mašinski jezik je kolekcija mašinskih instrukcija predstavljenih u binarnom obliku, tj. u vidu nizova nula i jedinica. Programi pisani na mašinskom jeziku su prilagođeni konkretnom hardveru (arhitekturi sistema) i izvršavaju se bez prevođenja. Da bi mašinski kod mogao da se izvrši, dovoljno je samo da bude smešten u memoriju i aktiviran.

Pisanje i razumevanje programa na mašinskom jeziku je teško i podložno greškama. S obzirom da se ovi programi sastoje samo od binarnih nizova, svaka modifikacija predstavlja rizik od pojave grešaka jer se izvodi na nivou bitova. Danas se praktično ne programira na mašinskom jeziku, ali je u nastavku opisan

ovaj proces da bi se bolje razumeo značaj mašinskog jezika u kontekstu savremenog programiranja.

Pisanje programa na mašinskom jeziku je moguće samo ukoliko su poznati elementi arhitekture računara, kao što su: veličina i organizacija memorije, procesorski registri, formati instrukcija i instrukcijski set. Ilustracije radi, neka je na raspolagnju jednostavan hipotetički računar koji ima memoriju M od 4096 8-bitnih lokacija i procesor koji među svojim 16-bitnim registrima ima akumulator (ACC) i registar podataka (DR), a podržava instrukcijski set u kome se, između ostalih, nalaze 16-bitne instrukcije čiji je opis dat u tabeli 6.1.

Tabela 6.1: *Deo instrukcijskog seta*

Kod operacije	Mnemonik	Operand	Opis
0000	STOP		zaustavljanje izvršenja
0001	LD	a	M → ACC
0010	ST	a	ACC → M
0011	MOVAC		ACC → DR
0101	ADD		ACC + DR → ACC

Pretpostavimo da je za ovaj računar potrebno napisati program na mašinskom jeziku koji sabira sadržaj memorijske lokacije čija je adresa 12 sa sadržajem memorijske lokacije čija je adresa 14 i dobijeni zbir smešta u memorijsku lokaciju sa adresom 16. Traženi program ima izgled kao na slici 6.1.

```
0001 0000 0000 1100
0011 0000 0000 0000
0001 0000 0000 1110
0101 0000 0000 0000
0010 0000 0001 0000
0000 0000 0000 0000
```

Slika 6.1 Primer mašinskog koda

Kao što se vidi, program na mašinskom jeziku je težak za razumevanje i debagovanje (ispravljanje grešaka). Na slici 6.2 je prikazan isti program smešten u memoriju, počeviši od lokacije sa adresom 0. Uzeto je da se, inicijalno, u memorijskim lokacijama sa adresama 12, 14 i 16 (poslednje tri vrste) nalaze vrednosti 350, 96 i 0, respektivno.

Slika 6.2: *Mašinski kod u memoriji*

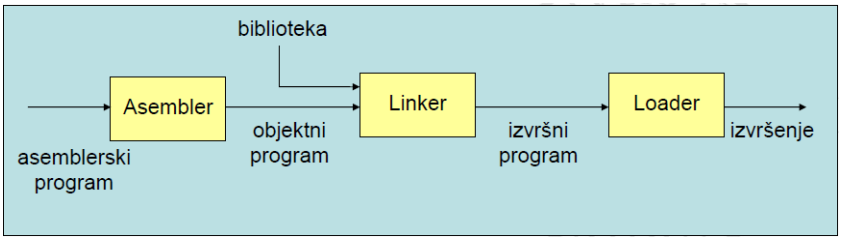
Adresa	Instrukcija	Opis	KO
0000 0000 0000	0001 0000 0000 1100	M(12) → ACC	LD: 0001
0000 0000 0010	0011 0000 0000 0000	ACC → DR	MOVAC:0011
0000 0000 0100	0001 0000 0000 1110	M(14) → ACC	LD: 0001
0000 0000 0110	0101 0000 0000 0000	ACC + DR → ACC	ADD: 0101
0000 0000 1000	0010 0000 0001 0000	ACC → M(16)	ST: 0010
0000 0000 1010	0000 0000 0000 0000	Stop	STOP: 0000
0000 0000 1100	0000 0001 0101 1110	350	
0000 0000 1110	0000 0000 0110 0000	96	
0000 0001 0000	0000 0000 0000 0000	0	

Radi lakšeg razumevanja programa, u desnom delu slike 6.2 dati su kratak opis odgovarajućih instrukcija programa i kodovi operacija tih instrukcija. Adrese memorijskih lokacija su 12-bitne zato što je kapacitet memorije $4096=2^{12}$ lokacija. Pošto su memorijske ćelije 8-bitne, svaka 16-bitna instrukcija je smeštena u dve uzastopne ćelije, što se vidi iz redosleda navedenih adresa. Ovakav program procesor može da izvrši. Rezultat izvršavanja bi bio promena sadržaja u lokaciji sa adresom 16 na novu vrednost 446.

6.2 Asemblerski jezik

U cilju lakšeg programiranja, nakon mašinskog, uveden je asemblerski jezik. Kao i mašinski jezik, asemblerski jezik je hardverski zavisan, što znači da svaki tip procesora ima svoj asemblerski jezik. Asemblerske instrukcije su jednostavnije za razumevanje od mašinskih, zato što su predstavljene mnemonicima i simboličkim adresama kojima čovek može lakše da upravlja.

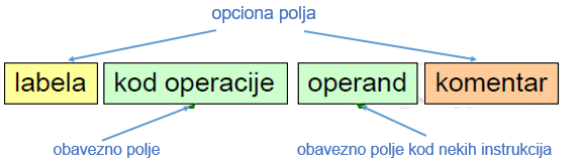
Da bi program napisan na asemblerskom jeziku mogao da se izvrši, neophodno je svaku njegovu instrukciju prevesti u odgovarajuću mašinsku instrukciju. Za prevođenje se koristi poseban program koji se naziva asembler (*assembler*). Uloga asemblera je da sve simboličke adrese u programu zameni numeričkim, simboličke kodove operacije mašinskim kodovima, rezerviše mesto za instrukcije i podatke u memoriji, predstavi konstante u mašinskom obliku, itd. Rezultat rada asemblera je odgovarajući mašinski kod. Ukoliko ima potrebe, dobijeni kod treba povezati sa bibliotekama ili drugim programima pomoću posebnog programa koji se naziva povezivač (*linker*). Ovako dobijeni izvršni kod se pomoću punjača (*loader*) smešta u memoriju i zatim se aktivira njegovo izvršavanje. Opisani postupak je prikazan na slici 6.3.



Slika 6.3: *Proces pripreme asemblerskog programa za izvršenje*

Asemblerski program se sastoji od niza asemblerskih instrukcija - iskaza. Svaki asemblerski iskaz se piše u posebnoj liniji programa i u opštem slučaju se sastoji od četiri polja:

SLIKA



Labela služi za simboličko imenovanje memorijskih adresa ili podataka. To je identifikator koji se može koristiti u drugim linijama koda za skok na liniju sa labelom. Komentar omogućava pisanje objašnjenja. Ova dva polja su opcionog karaktera. Obavezno polje predstavlja kod operacije koji ima već poznato značenje. Polje operand je obavezno kod nekih instrukcija i označava podatak ili dodatnu informaciju za kod operacije. Sledi primer dve asemblerske instrukcije:

```
START   LD    X
        BRA   START
```

Prva instrukcija (čiji je kod operacije LD) kopira sadržaj iz lokacije sa adresom X u akumulator. START predstavlja labelu. Druga instrukcija podrazumeva skok (kod operacije BRA) na iskaz sa labelom START.

Asemblerski program može da sadrži i iskaze koji se ne izvršavaju, već samo služe asembleru kao komande koje utiču na njegov način rada pri prevođenju asemblerskog koda u mašinski. Ovi iskazi se nazivaju **direktivama** ili pseudo-operacijama. To nisu asemblerske instrukcije i ne generišu nikakav mašinski kod. Direktive omogućavaju da se isti program prevodi na različite načine zavisno od parametara koje zadaje programer. Primenjuju se, na primer, za rezervaciju ili inicijalizaciju prostora za smeštaj promenljivih ili za kontrolu smeštaja programskog koda. Primer direktive W koja rezerviše 16-bitnu reč u memoriji za labelu X i inicijalizuje je na 120 je:

```
X  W      120
```

Sledi primer programa napisanog na asemblerskom jeziku (slika 6.4). To je program koji može da se izvrši (nakon asembliranja) na istom hipotetičkom računaru koji je već opisan u poglavlju 6.1, a rešava i isti zadatak (sabiranje sadržaja iz dve memorijske lokacije i smeštanje rezultata u treću lokaciju).

```
     LD       X        \ ACC ← X
     MOVAC             \ DR ← ACC
     LD       Y        \ ACC ← Y
     ADD               \ ACC ← ACC + DR
     ST       Z        \ Z ← ACC
     STOP
X    W        350      \ rezervacija reci inicijalizovane na 350
Y    W        96       \ rezervacija reci inicijalizovane na 96
Z    W        0        \ rezervacija reci inicijalizovane na 0
```

Slika 6.4: *Primer asemblerskog koda*

Kao što se vidi, ovaj program je jednostavniji za razumevanje i debagovanje od ranije datog programa na mašinskom jeziku. Jedina razlika je u tome što ovde nije eksplicitno definisano u kojim lokacijama se nalaze podaci (ranije su to bile fiksne lokacije sa adresama 12, 14 i 16), već se lokacije rezervišu tokom asembliranja. Posebnu pogodnost kod asemblerskog programiranja predstavlja mogućnost pisanja komentara.

Asemblerski jezici imaju primenu pri izradi sistemskih programa gde je potrebna direktna interakcija sa hardverom, pri izradi aplikacija sa vrlo ograničenim resursima, kod sistema sa posebnom namenom (*embedded systems*), itd. Programiranje u asemblerskom jeziku može rezultovati mašinskim kodom koji je kraći i brži od koda nastalog primenom viših programskih jezika.

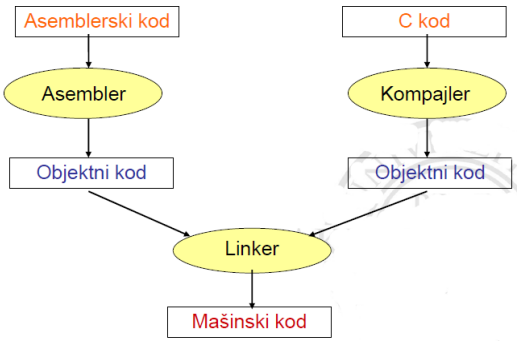
Prednosti programiranja u asemblerskom jeziku su:

- jasniji proces izvršavanja instrukcija
- prikaz načina na koji se podaci čuvaju u memoriji
- vidljiva interakcija između programa i operativnog sistema, procesora, U/I jedinica,...
- lakše kasnije programiranje na višim programskim jezicima zbog poznavanja procesa u računaru.

6.3 Viši programski jezici

Viši programski jezici, kao što su C, Visual Basic, Pearl, PHP, Python i drugi, podrazumevaju korišćenje varijabli, nizova, objekata, složenih aritmetičkih i logičkih izraza, petlji, funkcija, potprograma i sl. Po tome se razlikuju od mašinskih ili asemblerskih jezika koji se direktno obraćaju hardveru računara, na primer, memorijskim lokacijama, registrima, steku, itd.

Da bi program napisan na nekom višem programskom jeziku mogao da se izvrši, najpre se mora prevesti u objektni kod pomoću posebnog programa koji se naziva prevodilac (*compiler*). Moduli objektnog koda se zatim povezuju u celinu pomoću linkera. Tako se dobija mašinski (izvršni) kod koji se može uneti u memoriju i izvršiti. Na slici 6.5 dat je proces pripreme programa pisanog na višem programskom jeziku za izvršenje.



Slika 6.5: *Priprema programa pisanog na višem programskom jeziku za izvršenje*

Viši programski jezici imaju sintaksu koja poseduje elemente prirodnog jezika, pa su zato jednostavni za programiranje. Na primer, ranije razmatrano sabiranje sadržaja iz dve memorijske lokacije i smeštaj rezultata u treću, u programskom jeziku C se realizuje na sledeći način:

```
int a = 360, b = 96;
int c = a + b;
```

Prednosti programiranja u višim programskim jezicima su:

- veća ekspresivnost i konciznost
- potrebno je manje vremena za razvoj softvera
- lakša verifikacija koda, otkrivanje i ispravljanje grešaka

- lakše održavanje programskog koda
- veća mogućnost prenosivosti koda na druge platforme
- veća pouzdanost i sigurnost.

Vežbanja

1. U kom obliku se podaci predstavljaju u računaru? Šta su unarne, a šta binarne operacije? Koje klase operacija postoje? Ukratko opisati svaku od njih.

7 CPU arhitektura

Sa pojavom i razvojem viših programskih jezika, sve je postajala uočljivija razlika između operacija koje realizuju ovi jezici i operacija koje podržava računarska arhitektura. Mera ove različitosti je u literaturi poznata pod nazivom semantički gap. Posledice postojanja semantičkog gapa su: neefikasno izvršavanje programa, predugački mašinski programi, velika složenost kompajlera, itd.

Pokušaj projektanata da prevaziđu navedene probleme najpre je bio usmeren ka usložnjavanju arhitekture procesora u smislu specificiranja složenijih instrukcija u setu i uvođenja većeg broja adresnih modova. To je dovelo do pojave CISC (*Complex Instruction Set Computing*) CPU arhitekture.

Sa povećanjem obima i složenosti softvera, problem efikasnog izvršavanja programa postajao je sve izraženiji. Ključno pitanje je bilo kakav treba da bude instrukcijski set da bi se program efikasno izvršavao. Kod procesora sa minimalnim setom instrukcija, realizacija i vrlo jednostavnih operacija zahteva izvršavanje velikog broja instrukcija, što usporava rad računara. Kod procesora sa obimnim setom instrukcija, broj instrukcija je manji, ali zbog složenosti instrukcija, potrebno je više vremena za njihovo dekodiranje, pa je rad računara, takođe, usporen. Na projektantima je bilo da nađu balans između ove dve alternative.

Rađene su brojne studije u cilju utvrđivanja strukture karakterističnih (*banchmark*) programa po instrukcijama. Ispitivano je koje operacije se u programima najčešće izvršavaju, na koje operacije se troši najviše vremena, koji tipovi operanada se najčešće koriste i dr. Deo razultata ovih studija je dat u tabelama 7.1 i 7.2.

Tabela 7.1: *Udeo operacija u programu*

Operacija	Procena učešća
iskazi	35 %
petlje	5 %
pozivi procedura	15 %
uslovni skokovi	40 %
bezuslovni skokovi	3 %
ostalo	2 %

Tabela 7.2: *Udeo tipova podataka u programu*

Tip operanda	Procena učešća
skalarne promenljive	> 60 %
lokalne promenljive (od skalarnih)	> 80 %

Zaključci sprovedenih studija su bili sledeći:

- jednostavan prenos podataka (predstavljen iskazima) je mnogo češći od složenih operacija, pa ga treba optimizovati
- uslovni skokovi su dominantni, pa treba posvetiti punu pažnju podeli instrukcija na faze pri izvršavanju (neophodan je *pipeline* mehanizam)
- pozivi procedura su česti, a troše mnogo vremena, pa treba smisliti mehanizam prenosa parametara koji omogućava izvršavanje najmanjeg broja instrukcija
- mehanizam za smeštaj i pristup lokalnim skalarnim varijablama treba optimizovati
- umesto prilagođavanja instrukcijskog seta višim programskim jezicima (HLL), bolje je optimizovati performanse svojstava HLL programa na koje se troši najviše vremena

Kao konačan zaključak, utvđeno je da je bolje da arhitektura procesora bude jednostavnija nego složenija. Tako je nastala RISC (*Reduced Instruction Set Computing*) CPU arhitektura.

Danas je RISC arhitektura zastupljenija, ali mnogi prozvođači ne prave striktnu granicu po pitanju arhitekture, već u svoje procesore ugrađuju osobine i jedne i druge arhitekture koje su korisne za neku primenu.

7.1 CISC arhitektura

CISC arhitektura je nastala početkom 70-tih godina prošlog veka, iako je naziv kasnije uveden. Ona podrazumeva instrukcijski set sa relativno malim brojem složenih instrukcija čije je vreme izvršenja dugo. Zbog složenosti instrukcija, kontrolna jedinica za dekodovanje i izvršavanje instrukcija mora da bude, takođe, složena, što zahteva više vremena za njeno projektovanje, čime se povećava verovatnoća greške u njenom dizajnu. Ukoliko dođe do pojave grešaka, one se vrlo teško otkrivaju, a kad se otkriju, njihovo ispravljanje prouzrokuje značajne troškove. U CISC arhitekturi se koristi relativno veliki broj adresnih modova, kao i različiti formati instrukcija.

Ilustracije radi, data su dva primera procesora sa CISC arhitekturom.

DEC VAX 11/780 (1977.g.)

- 304 instrukcije u setu i 16 adresnih modova
- 16 32-bitna registra
- tipovi podataka: celi brojevi (6 tipova), pokretni zarez (4 tipa), nizovi karaktera (3 tipa), polja promenljive dužine, itd.
- instrukcije su mogle da imaju do 6 operanada
- dužina instrukcije od 2 do 14 bajtova

Motorola MC68020 (1984.g.)

- 101 instrukcija i 18 adresnih modova
- 16 registara opšte namene
- 7 tipova podataka
- dužina instrukcija od 1 do 11 16-bitnih reči

Nedostaci CISC arhitekture su vrlo brzo uočeni.

7.2 RISC arhitektura

Počeci RISC arhitekture se vezuju za IBM (IBM 801, 1975.g.), a kasnije za univerzitet Berkeley (RISC I, 1982.g.). Ova arhitektura promoviše jednostavnost umesto složenosti. RISC pristup pokušava da poboljša arhitekturu procesora dodavanjem resursa potrebnih za izvršavanje najčešće korišćenih operacija i operacija koje se najduže izvršavaju.

Osnovna svojstva RISC arhitekture su: instrukcijski set sa realativno malim brojem jednostavnijih instrukcija, mali broj adresnih modova i mali broj instrukcijskih formata. Zbog ovakvog instrukcijskog seta, kontrolna jedinica za dekodiranje i izvršavanje instrukcija ima jednostavan dizajn, a izvršavanje instrukcija je brzo (instrukcije se izvršavaju u jednom ciklusu takta). Pristup memoriji je moguć samo pomoću dve instrukcije (LOAD/STORE), što je dovoljno s obzirom da je skup procesorskih registara relativno veliki, pa se uglavnom koristi registar-registar arhitektura sa minimalnim pristupom memoriji. Sve instrukcije u setu su jednake dužine, što je vrlo pogodno za implementaciju *pipeline* mehanizma.

Primena procesora sa RISC arhitekturom je danas vrlo rasprostranjena. Ovi procesori se često koriste kao gradivni elementi u složenim multiprocesorskim sistema. Takođe, koriste se i u smart telefonima (*iPad*, *Android*), tabletima

(*Windows RT*), super-računarima (*Fujitsu K Computer* – najbrži računar u 2011, *IBM Sequoia* – najbrži u 2012, *Google Quantum Computer* – najbrži u 2014.g.). *Embedded* kontroleri koji rade sa aplikacijama visokih performansi, takođe, koriste RISC pristup.

U nastavku je dat primer računara sa RISC arhitekturom.

IBM SEQUOIA (2012.g.)

- brzina 16.32 petaFLOPS (*FLoating point Operations Per Second*)
- potrošnja 7,9 MW
- 1572864 procesorskih jezgara
- primarno namenjen za simulaciju nuklearnog oružja, ali se koristi i za naučna istraživanja na polju astronomije, energije, genetike, klimatskih promena

7.3 Poređenje arhitektura

Pri izboru CPU arhitekture, projektant mora da razmotri nekoliko faktora: veličinu sistema, složenost implementacije i brzinu izvršavanja programa. CISC i RISC arhitekture se razlikuju po strategiji pravljenja kompromisa (*trade-off*) između ovih faktora. Međutim, ako neka ideja poboljšava performanse RISC arhitekture, to ne znači da nije dobra u CISC pristupu. Za projektanta je važno da napravi procesor koji će efikasno da obavlja ono čemu je namenjen.

Najvažnija razlika između CISC i RISC pristupa nije u veličini i sastavu instrukcijskog seta (nigde nije specificirano koliko i kakvih instrukcija mora da bude u njemu), već u dizajnu kontrolne jedinice. Pojava VLSI (*Very Large Scale Integration*, 1970/80.g.) tehnologije omogućila je proizvodnju kompletnih procesora u čipu. Kod procesora sa CISC arhitekturom, kontrolne jedinice su zauzimale veliki deo čipa. Na primer, kontrolna jedinica procesora Motorola MC68020 zauzimala je 68% prostora u čipu. To je ostavljalo malo prostora za implementaciju drugih procesorskih funkcija u hardveru. S obzirom da RISC procesori imaju mnogo jednostavnije kontrolne jedinice, one zauzimaju manji prostor u čipu. Na primer, kontrolna jedinica procesora RISC I je zauzimala 6%, a RISC II 10% prostora u čipu. Tako je dobijeno više raspoloživog prostora za smeštaj drugih resursa, kao što su registri ili keš memorija, čime je smanjena komunikacija sa memorijom i ubrzan rad.

Različiti aspekti poređenja CISC i RISC arhitekture dati su u tabeli 7.3.

Tabela 7.3: *CISC/RISC poređenje*

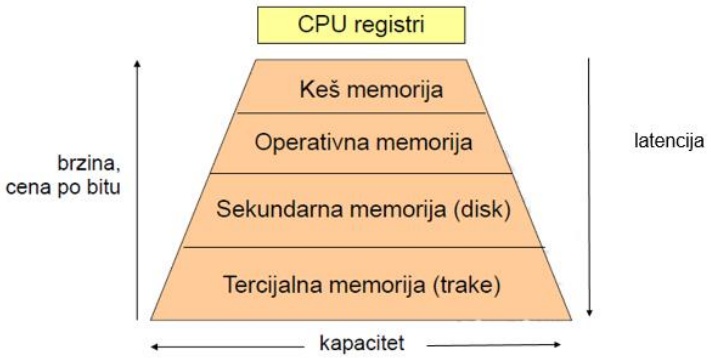
CISC	RISC
naglasak je na hardveru	naglasak je na softveru
programi su kraći i lakše se pišu	programi su duži i teže se pišu
izvršava se manje instrukcija	izvršava se više instrukcija
memorija-memorija instrukcije	registar-registar instrukcije
manje zauzeće memorije	veće zauzeće memorije
izvršavanje instrukcije u više taktova	izvršavanje instrukcije u jednom taktu
složeniji i skuplji kompajleri	jednostavniji i jeftiniji kompajleri
više tranzistora u realizaciji	manje tranzistora u realizaciji

Vežbanja

8 Memorijski sistem

Memorije, generalno, služe za smeštaj i čuvanje podataka i programa. U računaru postoji više vrsta memorija sa različitim ulogama. One čine memorijski sistem računara. Memorije su organizovane u hijerarhiju na čijem vrhu se nalaze brze i skupe memorijske jedinice malog kapaciteta, a na dnu sporije i jeftinije koje imaju veliki kapacitet. Ideja o hijerahijskoj organizaciji memorija potiče još iz 1946.g. a predložio ju je Von Neumann u okviru svoje arhitekture računara.

Tipična hijerarhijska organizacija memorijskog sistema prikazana je na slici 8.1. Na vrhu hijerarhije se nalaze procesorski registri, kao prvi nivo smeštanja podataka unutar samog procesora. U registrima se nalaze najčešće korišćeni podaci kojima se najbrže pristupa. Zatim sledi brza i skupa keš memorija relativno malog kapaciteta, kao prelaz ka operativnoj memoriji znatno većeg kapaciteta, ali manje brzine. Na sledećem nivou su sekundarne memorije, kao što su hard disk, optički diskovi (CD, DVD) i USB *flash*. Na dnu hijerahije su tercijalne memorije, obično magnetne trake.



Slika 8.1: *Memorijska hijerarhija*

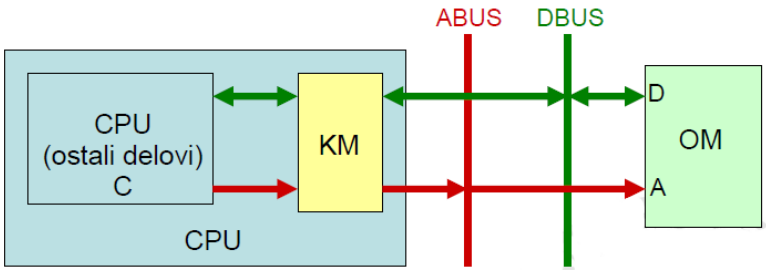
Na slici su strelicama označeni parametri po kojima je uspostavljena hijerarhija: brzina, kapacitet i cena. Smer strelice ukazuje na porast određenog parametra.

Poznavanje memorijske hijerarhije je važno zato što može značajno da utiče na performanse napisanog softvera. Na primer, ukoliko su podaci koje softver koristi smešteni u bržim memorijskim jedinicama, program će se brže izvršavati. Efikasnost hijerarhijske organizacije zasniva se na principu ne tako čestog prenošenja podataka u bržu memoriju uz njihovo višestruko korišćenje pre nego što se zamene novim podacima. Ovaj princip je ostvarljiv zahvaljujući fenomenu lokalnosti referenci. Lokalnost referenci podrazumeva da, u određenom vremenskom periodu, programi obično više puta pristupaju nekom ograničenom delu memorije. Postoje dva oblika lokalnosti: prostorna i vremenska lokalnost. Prostorna lokalnost je fenomen da ako se program referiše na jednu adresu, velika je verovatnoća da će se uskoro referisati na obližnje adrese u memoriji (na primer, na uzastopne instrukcije pri sekvencijalnom izvršavanju programa). Vremenska lokalnost je fenomen da ako se program referiše na neku adresu u memoriji, velika je verovatnoća da će joj uskoro opet pristupiti (na primer, instrukciji u programskoj petlji).

8.1 Keš memorija

Ideju o keš memoriji je prvi predstavio Wilkes 1965.g, dok je sam termin nastao kasnije. Uvođenje keš memorije je omogućilo brži pristup sadržaju operativne memorije, što je dovelo do značanog ubrzanja rada računara. Suština ideje je da se mali deo operativne memorije (za koji se očekuje da će uskoro biti korišćen), kopira u keš memoriju, koja je znatno brža od operativne, a nalazi se unutar procesora ili vrlo blizu njemu. Kada procesor zahteva neki podatak, najpre se proverava da li se on nalazi u keš memoriji, pa ako se nalazi, podatak se uzima iz keš memorije, bez obraćanja operativnoj memoriji, čime se postižu značajne vremenske uštede.

Na slici 8.2 prikazan je princip rada keš memorije (KM) koja se nalazi unutar procesora. Keš memorija je putem linija podataka i adresnih linija povezana sa ostalim komponentama procesora (označenim sa C), kao i sa operativnom memorijom (OM).



Slika 8.2: *Princip rada keš memorije*

Kada procesor želi da pristupi nekom podatku, C generiše adresu ćelije u OM u kojoj se nalazi taj podatak i prosleđuje je do KM, gde se proverava da li se traženi podatak nalazi u KM. Pošto je u početnom trenutku KM prazna, podatak se ne nalazi u KM, već u OM. Zato se taj podatak, kao i podaci na susednim adresama u OM, prenose (u vidu bloka podataka) u KM. Nakon toga, podatak se čita iz KM i prosleđuje procesoru. Ovaj postupak se ponavlja za sve podatke potrebne procesoru. Postepeno, KM se puni podacima, i u jednom trenutku će zahtevani podatak biti u KM. Tada će on moći direktno da se pročita iz KM, bez obraćanja OM, pa će pristup biti brži.

Kao što se vidi iz opisanog postupka, procesor generiše jednu adresu za pristup podatku. Međutim, podatak se može nalaziti ili u KM ili u OM, očigledno na različitim adresama zbog razlika u kapacitetima ove dve memorije. Prema tome, da bi se pristupilo podatku, potrebno je obezbediti prevođenje, tj. mapiranje adresa. Funkciju mapiranja obavlja jedinica za upravljanje memorijom (MMU – *Memory Management Unit*), koja se obično implementira kao deo CPU, mada može da bude i posebno integrisano kolo.

Da bi proces keširanja podataka bio moguć, potrebno je rešiti tri problema:

- problem evidencije blokova OM koji se trenutno nalaze u KM; da bi se po zadatoj adresi pristupilo podatku, neophodno je znati da li je blok OM u kome se podatak nalazi ranije prebačen u KM i, ako jeste, gde je smešten; ovaj problem je rešen uvođenjem tehnika preslikavanja koje precizno definišu sadržaj KM (po pitanju blokova podataka) u svakom trenutku
- problem odlučivanja o tome koji blok OM treba izbaciti iz pune KM da bi se oslobodio prostor za upis novog bloka OM; ovaj problem rešavaju tehnike zamene
- problem ažuriranja sadržaja OM u slučaju promene sadržaja KM; pošto se isti blok podataka nalazi i u OM i u KM, svaka izmena sadržaja u KM mora se reflektovati i na OM radi održanja njihove konzistentnosti; ovaj problem rešavaju tehnike ažuriranja

8.1.1 Tehnike preslikavanja

Mehanizam rada KM zahteva da ona bude podeljena u blokove koji se sastoje od određenog broja uzastopnih memorijskih lokacija. Broj lokacija u bloku se naziva dužinom bloka. Na sličan način, podeljena je i OM. Dužine blokova OM i KM su iste. Blokovi obe memorije su numerisani.

Tehnike preslikavanja definišu postupak upisa bloka podataka iz OM u blok KM. Pri tome se poštuje redosled lokacija u blokovima. U ovom poglavlju opisane

su tri tehnike preslikavanja: direktno preslikavanje, asocijativno preslikavanje i set-asocijativno preslikavanje.

Direktno preslikavanje

Direktno preslikavanje je najjednostavnija tehnika preslikavanja. Zasniva na tome da se isti blok OM smešta uvek u isti blok KM.

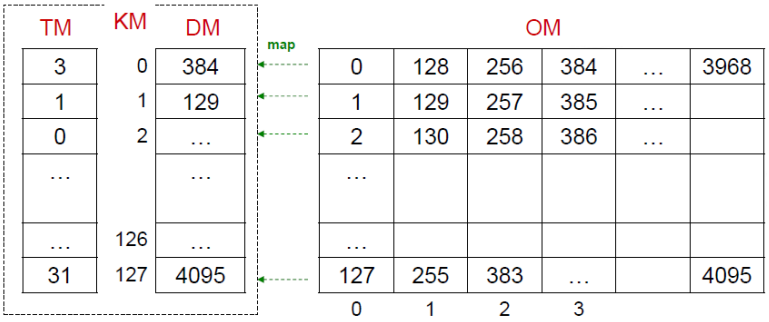
Pretpostavimo da je u KM potrebno upisati *i*-ti blok OM. Neka je NCB ukupan broj blokova u KM. Broj bloka *j* u KM u koji treba upisati dolazeći blok OM se računa po formuli:

$$j = i \bmod \text{NCB}$$

Kao što se vidi, više blokova OM se preslikava u isti blok KM, pa je ovo *više-na-jedan* tehnika mapiranja. Na primer, za NCB = 8, blokovi 14, 22 i 30 iz OM se upisuju u blok 6 u KM.

Pošto za funkcionisanje keš mehanizma nije dovoljno samo čuvati blokove podataka u KM, već se o njima mora voditi i evidencija, KM je podeljena na dva dela. U prvom delu, koji se naziva memorija tagova (TM – *Tag Memory*), čuvaju se podaci o evidenciji, dok se u drugom delu, koji se naziva memorija podataka (DM – *Data Memory*), čuvaju blokovi podataka iz OM.

Sledi primer direktnog preslikavanja. Neka OM sadrži NMB = 4096 blokova, a KM 128 blokova (NCB). Veličina bloka je B = 16 memorijskih lokacija (reči). Raspored blokova u memorijama prikazan je slici 8.3. Blokovi OM su organizovani u 128 vrsta po 32 bloka (4096/128 = 32). DM se sastoji od 128 blokova (označenih od 0 do 127) u koje su upisani blokovi OM 384, 129, ..., 4095.

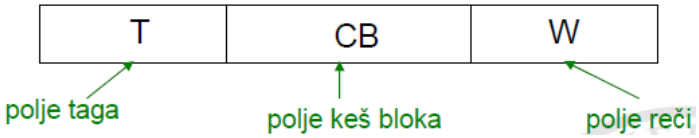


Slika 8.3: *Primer direktnog preslikavanja*

Prema ranije datoj formuli, svi blokovi iz prve vrste u OM se preslikavaju u blok 0 u DM. Takođe, svi blokovi iz druge vrste OM se preslikavaju u blok 1 u DM, itd. S obzirom da se više blokova OM iz jedne vrste može upisati u isti blok KM, neophodno je uvek imati evidenciju o tome koji blok iz vrste je trenutno

upisan. To se postiže pomoću TM. Za svaki blok u KM postoji odgovarajuća ćelija u TM koja sadrži redni broj bloka OM u odgovarajućoj vrsti koji se trenutno nalazi u KM (kolonu). Na primer, sadržaj prve ćelije u TM je 3 zato što se u bloku 0 u KM nalazi blok 384 iz OM koji je u trećoj koloni OM (brojanje kolona počinje od 0).

Kod direktnog preslikavanja, adresu podatka dobijenu od procesora, MMU interpretira tako što je podeli u tri polja (slika 8.4): polje taga (T bitova), polje bloka KM (CB bitova) i polje memorijske reči (W bitova).



Slika 8.4: *Interpretacija adrese kod direktnog preslikavanja*

Dužine polja (u broju bitova) određuju se na sledeći način:

$T = \log_2 (NMB/NCB)$

$CB = \log_2 NCB$

$W = \log_2 B$

Dužina adrese (A bitova) memorijske lokacije u OM koju je generisao procesor, računa se kao:

$A = \log_2 (B \cdot NMB)$

Primenom ovih formula u razmatranom primeru dobija se da je:

$T = \log_2 (4096/128) = 5$ $\quad CB = \log_2 128 = 7$ $\quad W = \log_2 16 = 4$
$A = \log_2 (16 \cdot 4096) = 16$

Kao što se vidi, dobijeno je da važi T + CB + W = A = 16, pa je adresa lokacije u OM 16-bitna.

U nastavku će, na razmatranom primeru, biti pokazano da je adresa koju generiše procesor istovremeno i adresa podatka u OM i adresa tog podatka u KM (ukoliko se on tamo nalazi).

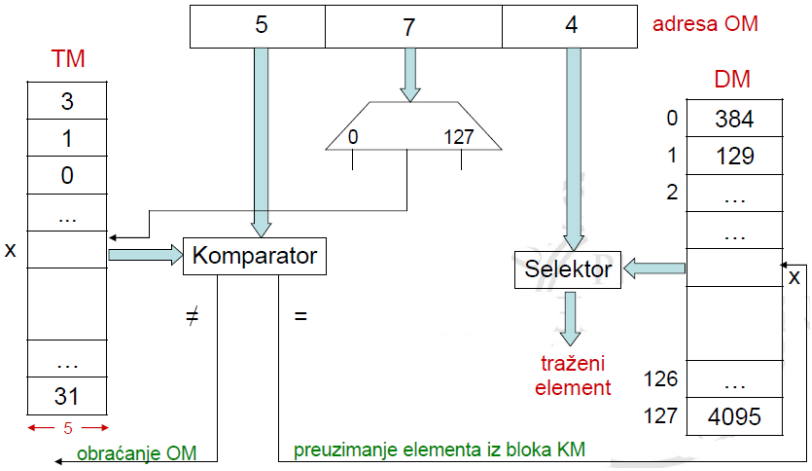
Neka je procesor generisao 16-bitnu adresu $1011010011000101_{(2)} = 46277_{(10)}$. U OM, ovaj podatak se nalazi u bloku 2892 na poziciji 5 unutar bloka od 16 reči, jer je 46277 / 16 = 2892 (5). Blok 2892 u OM se direktno preslikava u blok 76 u KM, zato što je 2892 mod 128 = 76. Vrednost taga za blok 2892 u OM je 22, jer je 2892/128 = 22 (76). S druge strane, ako se zadata adresa podeli u tri polja od 5, 7 i

4 bita, dobija se da polje taga sadrži vrednost $10110_{(2)} = 22_{(10)}$, polje bloka KM vrednost $1001100_{(2)} = 76_{(10)}$, a polje memorijske reči vrednost $0101_{(2)} = 5_{(10)}$. Kao što se vidi, ovako dobijene vrednosti odgovaraju prethodno izračunatim, što znači da se preko iste adrese pristupa podatku bilo u OM ili u KM.

Kada dobije adresu podatka, čija je struktura (T-polje, CB-polje, W-polje), od procesora, MMU izvršava sledeći protokol:

- na osnovu CB-polja određuje blok u KM u kome bi trebalo da se nalazi podatak
- poredi sadržaj T-polja sa vrednošću ćelije u TM koja odgovara ovom bloku; ako su vrednosti iste, podatak se nalazi u KM, a ako nisu, onda je podatak u OM
- ako je podatak u KM, na osnovu sadržaja W-polja određuje se njegova pozicija u bloku, pa mu se može da pristupi
- ako podatak nije u KM, potrebno je blok iz OM preneti u KM, podatak proslediti procesoru i ažurirati TM i DM

Za razmatrani primer, opisani protokol je dat na slici 8.5.



Slika 8.5: *Postupak direktnog preslikavanja - primer*

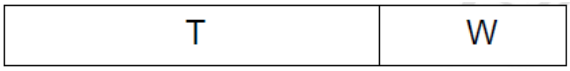
Kao što se vidi na slici, dekoder na osnovu sadržaja CB-polja određuje blok u KM (X) i njemu odgovarajuću ćeliju sa tagom u TM (X). Komparator poredi tag sa sadržajem T-polja. Ako su vrednosti jednake, podatak je u bloku KM. Selektor izdvaja traženi podatak iz bloka X u DM na osnovu sadržaja W-polja. Ukoliko podatak nije u KM, realizuje se obraćanje OM.

Prednost direktnog preslikavanja je u jednostavnosti postupka i direktnom određivanju mesta u KM gde će biti smešten blok OM. Nedostatak je u neefikasnom korišćenju KM. Naime, česte su situacije da više blokova konkuriše za isto mesto u KM, a da su neki drugi blokovi u KM prazni.

Asocijativno preslikavanje

Asocijativno preslikavanje je fleksibilnije od direktnog preslikavanja, zato što se kod ove tehnike blok OM može smestiti u bilo koji slobodan blok KM.

U slučaju asocijativnog preslikavanja, adresa koju generiše procesor se interpretira pomoću dva polja (slika 8.6): polja taga (T bitova) i polja memorijske reči (W bitova).



Slika 8.6: *Interpretacija adrese kod asocijativnog preslikavanja*

Polje taga jednoznačno identifikuje blok OM koji se nalazi u KM, dok polje memorijske reči identifikuje traženi podatak unutar bloka.

Dužine polja (u broju bitova) se određuju na sledeći način:

$T = \log_2 NMB$ (NMB je broj blokova u OM)

$W = \log_2 B$ (B je broj memorijskih reči u bloku)

Dužina adrese (A bitova) memorijske lokacije u OM koju je generisao procesor, računa se kao:

$A = \log_2 (B \cdot NMB)$

Primenom ovih formula u ranije razmatranom primeru dobija se da je:

$T = \log_2 4096 = 12$ $W = \log_2 16 = 4$

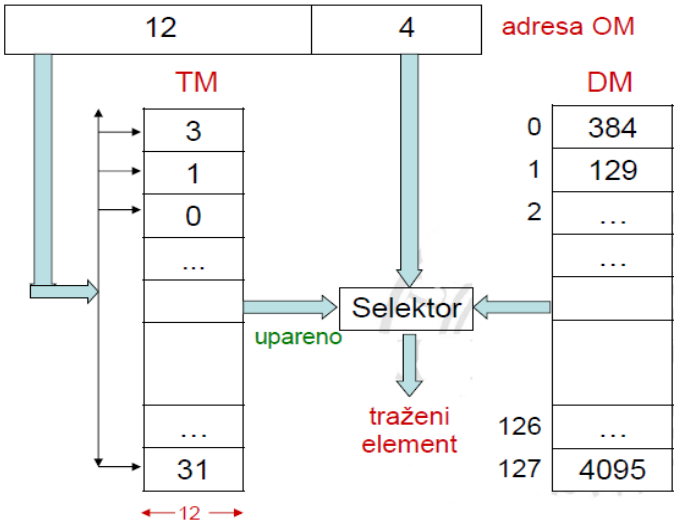
$A = \log_2 (16 \cdot 4096) = 16$

S obzirom da važi T + W = A = 16, adresa lokacije u OM je 16-bitna.

Kada od procesora dobije adresu podatka, čija je struktura (T-polje, W-polje), MMU izvršava sledeći protokol:

- ispituje da li se vrednost T-polja nalazi među postojećim tagovima u TM; ako se nalazi, podatak je u KM, inače je u OM
- ako je u KM, podatak se nalazi u bloku KM koji odgovara tagu, pa mu se može pristupiti na osnovu sadržaja W-polja
- ako podatak nije u KM, potrebno je blok iz OM preneti u KM, podatak proslediti procesoru i ažurirati TM i DM

Na slici 8.7 prikazan je navedeni protokol za razmatrani primer.



Slika 8.7: *Postupak asocijativnog preslikavanja - primer*

Suštinu asocijativnog preslikavanja predstavlja pretraživanje tagova u TM. Ukoliko bi ovo pretraživanje bilo sekvencijalno (redom), trajalo bi neprihvatljivo dugo. Zato se TM realizuje kao asocijativna memorija koja omogućava paralelno (istovremeno) pretraživanje, kao što je prikazano na slici. Paralelno pretraživanje zahteva dodatni hardver, što poskupljuje realizaciju i predstavlja glavni nedostatak tehnike asocijativnog preslikavanja.

Prednost asocijativnog preslikavanja je u efikasnom korišćenju KM, jer nema restrikcija po pitanju smeštaja dolazećeg bloka OM.

Set-asocijativno preslikavanje

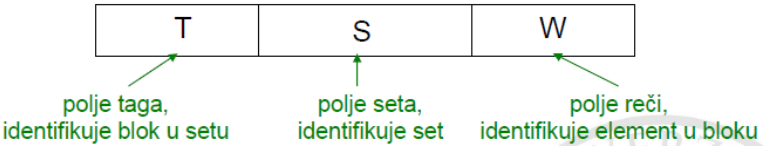
Set-asocijativno preslikavanje predstavlja kompromis između direktnog i asocijativnog preslikavanja. Ideja je da se zadrže jednostavnost direktnog i efikasnost asocijativnog preslikavanja tako što bi se KM podelila u skupove (setove) sa određenim brojem blokova. Blok OM bi se uvek preslikavao u isti set u KM (slično direktnom preslikavanju), ali u bilo koji slobodan blok u tom setu (slično asocijativnom preslikavanju).

Pretpostavimo da je KM podeljena u NS setova. Blok *i* u OM se mapira u set *s* KM prema formuli:

$$s = i \bmod NS$$

Unutar seta *s*, *i*-ti blok OM se može smestiti u bilo koji blok KM koji pripada tom setu.

Kod ove tehnike, adresa podatka koju generiše procesor se interpretira pomoću tri polja (slika 8.8): polja taga (T bitova), polja seta (S bitova) i polja memorijske reči (W bitova). Polje taga identifikuje blok u setu, polje seta identifikuje set, dok polje memorijske reči identifikuje traženi podatak unutar bloka.



Slika 8.8: *Interpretacija adrese kod set-asocijativnog preslikavanja*

Ako je broj blokova u setu BS, dužine polja (u broju bitova) određuju se na sledeći način:

$T = \log_2 (NMB \cdot BS/NCB)$ (NMB je broj blokova u OM, a NCB u KM)

$S = \log_2 NS$

$W = \log_2 B$ (B je broj memorijskih reči u bloku)

Dužina adrese (A bitova) memorijske lokacije u OM koju je generisao procesor, računa se kao:

$A = \log_2 (B \cdot NMB)$

Primenom ovih formula u razmatranom primeru, uz BS = 4 i NS = 128/4 = 32, dobija se da je:

$T = \log_2 (4096 \cdot 4/128) = 7$ $S = \log_2 32 = 5$ $W = \log_2 16 = 4$

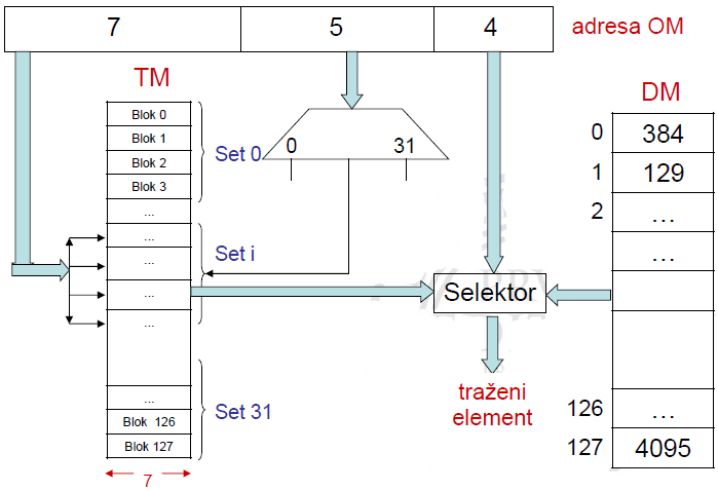
$A = \log_2 (16 \cdot 4096) = 16$

S obzirom da važi T + S + W = A = 16, adresa lokacije u OM je 16-bitna.

Kada dobije adresu podatka, čija je struktura (T-polje, S-polje, W-polje), MMU izvršava sledeći protokol:

- na osnovu sadržaja S-polja direktno određuje set u koji se blok OM mapira
- ispituje da li se sadržaj T-polja nalazi među postojećim tagovima u TM za taj set; ako se nalazi, podatak je u KM, inače je u OM
- ako je podatak u KM, pronalazi se unutar odgovarajućeg bloka u DM na osnovu vrednosti u W-polju
- ako podatak nije u KM, potrebno je blok iz OM preneti u KM, podatak proslediti procesoru i ažurirati TM i DM

Na slici 8.9 prikazan je opisani protokol za razmatrani primer.



Slika 8.9: *Postupak set-asocijativnog preslikavanja - primer*

Kao što se vidi na slici, dekoder na osnovu sadržaja S-polja određuje set i njemu odgovarajuće ćelije sa tagovima u TM. Zatim se asocijativnim pretraživanjem ispituje da li u ovim ćelijama postoji vrednost iz T-polja. Ako postoji, na osnovu taga se određuje blok u setu u kome se nalazi podatak. Selektor izdvaja traženi podatak iz bloka u DM na osnovu sadržaja W-polja. Ukoliko podatak nije u KM, realizuje se obraćanje OM.

Opisana tehnika nije toliko efikasna kao asocijativno preslikavanje, ali je preuzela jednostavnost direktnog preslikavanja.

8.1.2 Tehnike zamene

Tehnike zamene rešavaju problem izbora bloka koji će biti izbačen iz pune KM kako bi se na njegovo mesto upisao dolazeći blok OM. Potreba za tehnikama zamene postoji samo kod asocijativnog i set-asocijativnog preslikavanja, dok se u slučaju direktnog preslikavanja unapred zna lokacija upisa dolazećeg bloka.

Izbor bloka koji će biti zamenjen se može sprovesti:

- po slučajnom izboru
- po vremenu koje su upisani blokovi proveli u KM (FIFO tehnika)
- po trenucima poslednjeg korišćenja blokova upisanih u KM (LRU tehnika)

Tehnika slučajanog izbora

U računaru postoji generator slučajnih brojeva (*random generator*) koji po uključenju računara počne da generiše brojeve iz zadatog opsega. Tehnika

slučajnog izbora koristi izlaz ovog generatora u trenutku zamene na osnovu koga određuje broj bloka KM koga treba zameniti blokom OM. Ova tehnika je prvi put bila upotrebljena u Intel-ovoj iAPX seriji mikroprocesora. Tehnika je vrlo jednostavna, a njen glavni nedostatak je to što ne uzima u obzir lokalnost referenci, pa se može desiti da se izbaci blok koji će se uskoro koristiti.

FIFO tehnika

FIFO (*First-In-First-Out*) tehnika kao kriterijum uvodi vreme koje su blokovi proveli u KM od svog upisa to tekućeg trenutka. Izbor se pravi tako što se iz KM izbacuje onaj blok koji je najviše vremena proveo u njoj. Dakle, to je blok koji je, od svih trenutno raspoloživih blokova u KM, prvi upisan. Za sprovođenje ove tehnike, neophodno je vođenje evidencije o vremenima upisa blokova u KM, što ovu tehniku čini složenijom od tehnike slučajnog izbora. FIFO tehnika je pogodna za programe koji se pravolinijski izvršavaju, jer kod njih lokalnost referenci nije bitna.

LRU tehnika

Najefikasniji postupak zamene je LRU (*Least Recenlty Used*) tehnika. Ona se zasniva na praćenju istorije korišćenja blokova koji se nalaze u KM, za šta je zadužen keš kontroler. Kriterijum za izbacivanje bloka iz KM je vreme njegovog poslednjeg korišćenja. Naime, iz KM se izbacuje blok sa najranijim vremenom poslednjeg korišćenja.

Keš kontroler može pratiti istoriju korišćenja blokova na različite načine. Jedna moguća realizacija je da se svakom bloku KM pridruži brojač. Vrednost brojača je veća ako je blok ranije korišćen. Radi lakšeg razumevanja realizacije, može se posmatrati da se brojači nalaze u listi po redosledu korišćenja njima odgovarajućih blokova. Na čelu liste je brojač bloka koji je poslednji bio korišćen (ima najmanju vrednost). Kada procesor želi da pristupi nekom podatku, najpre se proverava da li se blok sa tim podatkom nalazi u KM. Ako se nalazi, pročita se trenutna vrednost njegovog brojača *b*. Nakon pristupa podatku, brojači blokova KM se ažuriraju na sledeći način:

- brojač bloka u kome se nalazi podatak se resetuje na 0 (time se brojač bloka sa podatkom stavlja na čelo liste jer se podatku upravo pristupilo)
- svi brojači sa vrednošću manjom od *b* se inkremetiraju za 1 (pomeraju se za jedno mesto od čela liste, jer je na čelo liste stavljen brojač bloka sa podatkom)
- svi brojači sa vrednošću većom od *b* se ne menjaju (oni su se u listi nalazili iza brojača bloka sa podatkom, pa promena na čelu liste nema uticaja na ove brojače)

U slučaju da se traženi podatak ne nalazi u KM, potrebno je uraditi sledeće:

- blok KM sa najvećom vrednošću brojača zameniti blokom OM u kome se nalazi podatak
- brojač zamenjenog bloka postaviti na 0 (time se on postavlja na čelo liste)
- vrednosti svih ostalih brojača inkrementirati za 1 (jer je brojač sa kraja liste prebačen na čelo liste)

LRU tehnika, kod koje je keš kontroler realizovan na opisani način, biće ilustrovana na jednom primeru. Neka se KM sastoji od 10 blokova koji su popunjeni blokovima OM 10, 3, 15, 1, 6, 9, 12, 5, 32 i 4, kao što je prikazano na slici 8.10. Svakom bloku KM pridružen je brojač sa odgovarajućom vrednošću. Procesor zahteva podatke iz blokova OM 6 i 11, redom.

Primer 4: iz OM dolaze blokovi 6, 11, 1 i 20

		promene vrednosti brojača			
B10	3	4	5	5	6
B3	4	5	6	6	7
B15	1	2	3	3	4
B1	0	1	2	0	1
B6	5	0	1	2	3
B9 →B20	10	10	11	11	0
B12	6	6	7	7	8
B5	2	3	4	4	5
B32	7	7	8	8	9
B4→B11	12	12	0	1	2
KM	brojači	Blok 6	Blok 11	Blok 1	Blok 20

Slika 8.10: *LRU tehnika - primer*

Prvi podatak (iz bloka 6) se već nalazi u 4. bloku KM, čiji brojač ima vrednost 5. Nakon pristupa podatku, ovaj brojač se resetuje na 0, brojači prva četiri bloka KM, kao i brojač 7. bloka se inkrementiraju (jer su imali vrednost manju od 5), dok brojači ostalih blokova ostaju nepromenjeni (jer su imali vrednost veću od 5).

Drugi podatak (iz bloka 11) se trenutno ne nalazi u KM, pa je potrebno blok 11 iz OM preneti u KM. Blok OM se upisuje u blok KM čiji brojač ima najveću vrednost (12), a to je blok 9. Nakon upisa, brojač 9. bloka se resetuje na 0, a svi ostali brojači se inkrementiraju (jer su imali vrednost manju od 12). Kao što se vidi, u ovom slučaju je urađena zamena bloka po kriterijumu koji podržava LRU tehnika.

8.1.3 Tehnike ažuriranja

Tehnike ažuriranja se bave problemom koherencije, tj. usklađivanja sadržaja KM i OM. U KM se u svakom trenutku nalazi određen broj kopija blokova OM. Procesor može da pristupa podacima u KM i da menja njihove vrednosti. Svaka promena, učinjena samo u bloku KM, dovodi do neusaglašenosti između sadržaja tog bloka i njegovog originala koji se nalazi u OM. Za korektan rad, neophodno je usklađivanje sadržaja OM sa promenama u KM, što omogućavaju tehnike ažuriranja.

Postoje četiri tehnike ažuriranja:

- tehnika upisa ako blok postoji u KM
- tehnika upisa ako blok ne postoji u KM
- tehnika čitanja ako blok postoji u KM
- tehnika čitanja ako blok ne postoji u KM

Tehnika upisa ako blok postoji u KM (*Cache Write Policy Upon a Cache Hit*) se bavi problemom ažuriranja u slučaju upisa podatka u neki blok KM. Ova tehnika pruža dve mogućnosti ažuriranja. Prva mogućnost je trenutni upis (*write-trough*) kod koga se svaka operacija upisa realizuje tako što se istovremeno podatak upisuje i u KM i u odgovarajući blok OM. Druga mogućnost je odloženi upis (*write-back*) kod koga se podatak upisuje samo u blok KM, dok se upis u OM odlaže sve dok se ne pojavi potreba da se taj blok izbaci iz KM. U trenutku zamene bloka KM, ispituje se da li je taj blok menjan. Ukoliko jeste, blok KM se upisuje u blok OM, a ako nije, onda se samo blok KM zameni dolazećim blokom OM. Da bi pomenuto ispitivanje bilo moguće, neophodno je svakom bloku KM pridružiti jedan bit (*dirty bit*) koji ima vrednost 1 ako je postojao bar jedan upis u taj blok, inače ima vrednost 0.

Tehnika upisa ako blok ne postoji u KM (*Cache Write Policy Upon a Cache Miss*) se bavi problemom ažuriranja u slučaju upisa podatka u blok OM koji nije ranije prenet u KM. Ovde, takođe, postoje dve mogućnosti ažuriranja. Prva je dovlačenje bloka (*write-allocate*) što podrazumeva prenos i upis bloka OM u KM, nakon čega se primenjuje prethodno opisana tehnika. Druga mogućnost je nedovlačenje bloka (*write-no-allocate*) kod koje se podatak upisuje samo u blok OM.

Tehnika čitanja ako blok postoji u KM (*Cache Read Policy Upon a Cache Hit*) je jednostavna jer se podatak direktno čita iz KM.

Tehnika čitanja ako blok ne postoji u KM (*Cache Read Policy Upon a Cache Miss*) se bavi problemom ažuriranja u slučaju čitanja podatka iz nekog bloka OM.

Tehnika pruža dve mogućnosti ažuriranja. Prva mogućnost je direktno prosleđivanje, kada se blok OM prenosi u KM, a podatak odmah po pristizanju u KM prosleđuje procesoru. Druga mogućnost je prosleđivanje sa zadrškom, pri čemu se blok OM prenosi u KM, ceo se smešta, pa se tek onda čita podatak i prosleđuje procesoru.

Literatura

1. Violeta Tomašević, *Osnovi računarske tehnike*, Univerzitet Singidunum, 2012.

2. Jovan Đorđević, *Osnovi računarske tehnike - Zbirka rešenih zadataka*, Viša poslovna škola, Blace, 2006.

3. Mostafa Abd-El-Barr and Hesham El-Rewini, *Fundamentals of Computer Organization and Architecture*, Wiley & Sons, Series on Parallel and Distributed Computing, 2004.

4. William Stallings, *Computer Organization and Architecture: Designing for Performance*, 8th Edition, Prentice Hall, 2010.

5. Sajjan Shiva, *Computer Organization, Design and Architecture*, 4th Edition, Taylor & Frances Group, 2007.

6. Carl Hammaher, Zvonko Vranešić and Safwat Zaky, *Computer Organization*, 5th Edition, Mc Graw-Hill, Inc. 2002.

7. Morris Mano, *Computer System Architecture*, ISBN: 978-0131755635. 1992.