Hipoteticki procesori x86

Materijal koji treba da pomogne pri savladavanju osnova programiranja u asembleru rešavanju treceg domaceg zadatka (DZ3) nacinjen je na osnovu treceg poglavlja (System Organization) knjige The Art of Assembly Language koju je napisao i stavio svim zainteresovanima na raspolaganje u elektronskoj formi profesor

Randall Loren Hyde
Department of Computer Science
University of California, Riverside
Riverside, Ca 92521
(909) 787-3759
Internet: rhyde@cs.ucr.edu
WWW Home page: www.cs.ucr.edu
(under faculty/lecturer)

Kako je familija mikroprocesora 80x86 veoma kompleksna, poznavanje svih njenih clanova predstavlja velik napor za pocetnika. Zato cemo karakteristike clanova ove familije ilustrovati na primeru hipotetickih procesora - nastavnih modela racunara 886, 8286, 8486 i 8686 koji predstavljaju pojednostavljene verzije 80x86 cipova, ali omogucavaju da se ilustruju razne arhitektonske specificnosti bez zalaženja u sve detalje ogromnog CISC skupa instrukcija. Ovo poglavlje koristi x86 hipoteticki procesor za opis koncepata *kodiranja instrukcija, nacina adresiranja, sekvencijalnog izvršavanja, bafera naredbi (prefetch queue), tekucih linija (pipe-lines) i superskalarnih operacija.*

Arhitektura procesora familije x86

CPU registri su specijalne memorijske lokacije konstruisane od flip-flop kola. Oni nisu deo memorije vec su ugradeni u CPU cip. Razni clanovi 80x86 familije imaju registre razlicite velicine. NARx86 (nadalje x86) imaju tacno 4 registra svaki velicine 16-bitova. Sve aritmeticke i logicke operacije vrše se u procesorskim registrima.

Kako x86 ima vrlo malo registra, svakome od njih pristupamo na osnovu imena, a ne na osnovu adrese. Nazivi registara su:

AX - Akumulator

BX - Bazni registar

CX - Count

DX - Data

Osim ovih registara koji su vidljivi za programere imamo još IP - instruction pointer koji svojim sadržajem ukazuje na sledece instrukciju i flag registar koji cuva rezultate poredenja (da li je neka vrednost manja, jednaka ili veca od druge vrednosti)

Buduci da su ugradeni u CPU mnogo im se brže pristupa nego memorijskim registrima - trenutno, bez cekanja, dok je za pristup memorijskim registrima potreban jedan ili više taktova. Zato se treba truditi da se što više podataka cuva u procesorskim registrima. Skup registara je vrlo mali i mnogi od njih imaju i specijalne namene, ali su i dalje idealno mesto za privremeno cuvanje podataka.

Aritmeticko - logicki uredaj (ALU) je mesto gde se dešava vecina procesorskih akcija. Npr. ako želite da dodate broj 5 na sadržaj registra AX, CPU vrši sledece akcije:

- kopira vrednost iz AX u ALU
- šalje broj 5 u ALU
- nareduje ALU da sabere dve poslate vrednosti
- vraca rezultat u AX registar

Bus interface unit (BIU) zadužen je za kontrolu adresne i magistrale podataka kada se pristupa glavnoj memoriji. Ako postoji keš u CPU, onda je BIU takode odgovoran za pristup podacima u kešu.

Upravljacki organ i skup instrukcija

Glavno pitanje na koje želimo da damo odgovor u ovom trenutku je "Kako egzaktno CPU izvršava instrukcije?" Ovo se obavlja zadavanjem CPU fiksnog skupa komandi ili instrukcija. Imajte na umu da projektanti CPU konstruišu procesore korišcenjem logickih kola za izvršavanje ovih instrukcija. Poželjno je da broj tih logickih kola bude što manji, pa projektanti obavezno moraju da redukuju broj i složenost komandi koje CPU prepoznaje. Ovaj mali skup komandi naziva se *skup instrukcija*.

Prvi kompjuteri (pre Nojmanovi) obicno su se programirali spolja (hard-wired). Povezivanjem odgovarajucih komponenti definisalo se šta ce program rešavati. Kada se prelazilo na rešavanje drugog zadatka, moralo se vršiti povezivanje na neki drugi nacin što je znatno otežavalo korišcenje racunara. Pojava programibilnih kompjuterskih sistema sa kontrolnim panelom sa rupicama u koje su mogle da se utaknu žice i na taj nacin izvrši programiranje znacila je izvestan napredak. Kompjuterski program sastojao se od niza redova rupica od kojih je svaki predstavljao jednu operaciju. Programer je mogao da selektuje jednu ili više instrukcija umetanjem žice u pojedine rupice. Razume se, velika poteškoca u ovakvom programiranju je što je broj mogucih instrukcija jako ogranicen brojem rupica koje su fizicki mogle da se smeste u svakom redu. Ovaj problem su projektanti relativno brzo prevazišli dodatnom logikom i omogucili su da se sa zadavanja n instrukcija pomocu n rupica prede na zadavanje n instrukcija pomocu logn rupica. To su ostvarili dodeljivanjem numerickog koda svakoj instrukciji, a zatim su dekodirali ovu instrukciju kao binarni broj korišcenjem u logn rupica. Ovo je zahtevalo 8 dodatnih logickih funkcija za dekodiranje ABC bitova sa kontrolnog panela, ali ekstra kola malo koštaju zato što redukuju broj rupica koji mora postojati za svaku instrukciju.

Razume se, mnogi opkodovi nisu instrukcije sami za sebe. Npr. MOV instrukcija zahteva dva argumenta: *izvorni* i *odredišni*. Projektanti CPU obicno dekodiraju izvorni i odredišni argument kao deo mašinske instrukcije, odgovarajuca rupica odgovara izvornom a odgovarajuca odredišnom argumentu.

Glavni napredak u kompjuterskom dizajnu koji je dala Fon Nojmanova arhitektura je koncept *unutrašnjeg programa.* Jedan veliki problem u spoljašnjem programiranju je da je broj

1 U svom istorijskom radu publikovanom 1945. godine Džon fon Nojman izdvojio je i detaljno opisao kljucne komponente onoga što se danas naziva "Fon Nojmanova arhitektura". Da bi kompjuter bio efikasna i univerzalna mašina mora imati sledece komponente: aritmeticko-logicki uredaj (ALU) koji vrši aritmeticke i logicke operacije; upravljacki uredaj (CU - Control Unit) koji "diriguje" izvršavanjem naredbi; unutrašnju memoriju u kojoj bi se cuvali program i podaci; ulazni uredaj za prihvatanje i izlazni uredaj za izdavanje informacija. Fon Nojman je smatrao da taj sistem mora da radi sa binarnim brojevima, da bude elektronski a ne mehanicki i da izvršava instrukcije jednu za drugom. Fon Nojmanovi principi postali su opštepoznati samo zato što su se široko primenjivali sve vreme - oni su bili ugradeni u osnovu kako velikih elektronskih racunara prve generacije, tako i racunara narednih generacija ukljucujuci i mini i mikro racunare. Mada se danas koncept unutrašnjeg programa pripisuje Nojmanu, treba znati da su tvorci prvog elektronskog racunara Mauchly i Eckert govorili o programima zapisanim u memoriji racunara pola godine pre no što se Nojman pridružio njihovoj radnoj grupi, a Alan Tjuring je unutrašnju memoriju dodelio svojoj hipotetickoj univerzalnoj mašini još 1936. godine.

instrukcija koje se mogu zadati ogranicena brojem redova na panelu. Džon Fon Nojman i drugi programeri tog vremena uocili su relaciju izmedu rupica na kontrolnom panelu i bitova u memoriji. Došli su na ideju da da smeste binarni ekvivalent programa u memoriju i odatle uzimaju instrukciju po instrukciju, dovode je u registar za dekodiranje (*registar naredbe*) koji je direktno povezan sa logickim kolima *dekodera operacija* u CPU. Sve ovo podrazumeva još više logike u CPU. Upravljacka kola kontrolne jedinice (Control Unit) uzimaju kodove instrukcija (za koje se koriste termini *operacioni kodovi* ili *opkodovi*) iz memorije i upisuju ih u registar za dekodiranje. CU sadrži specijalan registar koji se naziva *instruction pointer* (IP) koji sadrži adresu izvršne instrukcije. Po izvršenju instrukcije, kontrolna jedinica uvecava IP tako da on ukazuje na sledecu instrukciju koju bi trebalo uzeti iz memorije.

Kada projektanti biraju skup instrukcija, obicno uzimaju da je opkod neki umnožak 8 bitova, tako da CPU lako uzima kompletnu instrukciju iz memorije. Cilj CPU dizajnera je da dodele odgovarajuci broj bitova za *klasu instrukcija* (MOV, ADD i sl.) i za *polja argumenata*. Više bitova u polju klase instrukcija omogucava da repertoar instrukcija bude veci, dok dodatni bitovi za argumente omogucavaju da se argumenti biraju iz veceg adresnog prostora (npr. sa memorijskih lokacija). Ovde se pojavljuju dodatni problemi. Neke instrukcije imaju samo jedan argument ili uopšte nemaju argumenata. Kako ne bi gubili neiskorišcene bitove, projektanti polje argumenata koriste za zadavanje opkodova novih instrukcija, što je opet povezano sa ugradivanjem dodatne logike. Kod familije Intel 80x86 ovo je dovedeno do ekstrema jer dužina instrukcije može da iznosi od 1 do 10 bajtova. Kako je za pocetnike Intelova shema dekodiranja instrukcija prevelik zalogaj, mi smo hipoteticke procesore x86 snabdeli samo malom i znatno jednostavnijom shemom dekodiranja koja ipak može dati osecaj šta se u stvari dešava unutar procesora prilikom izvršavanja programa.

Skup instrukcija x86

Procesori x86 imaju 20 osnovnih instrukcija. Sedam instrukcija ima 2 argumenta, 8 jedan i 5 je bez argumenta. To su sledece: MOV (sa dve forme), ADD, SUB, CMP, AND, OR, NOT, JE, JNE, JB, JBE, JA, JAE, JMP, BRK, IRET, HALT, GET i PUT.

Instrukcija za prenošenje podataka MOV

može se pojaviti u dve forme:

- MOV reg, reg/mem/imm
- MOV mem, reg

gde je:

reg - oznaka za neki od registara AX, BX, CX ili DX;

mem - 16-bitna memorijska lokacija;

imm - neposredan argument - ovde heksadekadna numericka konstanta.

2 Za razliku od realnih procesora kod kojih oznakom h moramo naglasiti da je konstanta heksadekadna kako bi se razlikovala od dekadne, kod procesora x86 uz heksadekadne konstante ne moramo da stavljamo nikakvo bliže odredenje, jer oni koriste samo heksadekadni brojni sistem.

Oznaka *reg/mem/imm* kazuje da odgovarajuci argument može biti u procesorskom registru, memoriji ili je konstanta. Sve instrukcije menjaju odredišni argument, a izvorni uvek zadržava svoju prvobitnu vrednost.

Aritmeticke i logicke instrukcije

mogu se pojaviti u sledecim formatima:

- ADD reg, reg/mem/imm
- SUB reg, reg/mem/imm
- CMP reg, reg/mem/imm
- AND reg, reg/mem/imm
- OR reg, reg/mem/imm
- NOT reg/mem

Dejstvo instrukcija je sledece:

ADD d,s	d:= d + s
SUB d,s	d:= d - s
CMP d,s	Poredi d sa s i pamti rezultat poredenja
AND d,s	d:= d and s (bit po bit)
OR d,s	d:= d or s (bit po bit)
NOT d	d:= not d (bit po bit)

Instrukcije za predaju upravljanja

prekidaju sekvencijalno izvršavanje instrukcija i predaju upravljanje nekoj drugoj instrukciji bilo bezuslovno bilo u zavisnosti od ostvarenja nekog uslova (koji se proverava sa CMP). Brojevi se porede kao neoznaceni. Procesori x86 imaju sledece instrukcije za predaju upravljanja:

- JA dest skok ako je prvi veci (Above)
- JAE dest skok ako je prvi veci ili jednak (Above or Equal)
- JB dest skok ako je prvi manji (Below)
- JBE dest skok ako je prvi manji ili jednak (Below or Equal)
- JE dest skok ako su jednaki (Equal)
- JNE dest skok su razliciti (Not Equal)
- JMP dest bezuslovni skok
- IRET povratak iz interapta (Interrupt RETurn)

Prvih 6 instrukcija ove klase proveravaju rezultat prethodne CMP instrukcije. Ako je uslov ispunjen, upravljanje se predaje instrukciji sa adresom *dest*, inace se nastavlja sa sekvencijalnim izvršavanjem instrukcija. JMP bezuslovno predaje upravljanje na labelu *dest*, a IRET vraca kontrolu sa rutine za opsluživanje prekida, o cemu cemo detaljnije govoriti kasnije.

Ostale instrukcije

Instrukcije **GET** i **PUT** omogucavaju ucitavanje i ispisivanje celobrojnih heksadekadnih vrednosti. Još dve instrukcije **HALT** i **BRK** su bez argumenta. HALT prekida izvršavanje programa dok BRK privremeno zaustavlja izvršavanje programa dajuci mogucnost korisniku da ga nastavi.

Nacini adresiranja

Procesori x86 koriste *registarske*, *neposredne* (konstante) i *memorijske* argumente. Za zadavanje adresa memorijskih argumenata na ovim procesorima koriste se tri *nacina adresiranja: direktan, indirektan* i *indeksni* nacin.

Registarsko adresiranje

Najlakše se koriste argumenti koji se nalaze u procesorskim registrima, tj. argumenti zadati registarskim adresiranjem. Na primer:

```
MOV AX,AX
MOV AX,BX
MOV BX,AX
```

Prva instrukcija prakticno ništa ne radi - kopira vrednost iz AX na to isto mesto. Druga instrukcija kopira vrednost iz izvornog registra BX u odredište AX. Napominjemo da ova instrukcija *ne menja* sadržaj izvornog registra. Treca instrukcija prenosi podatak u suprotnom smeru - AX je izvorni, a BX odredišni registar.

Neposredno adresiranje

Za konstante, odnosno argumente koji se nalaze u okviru same naredbe koristi se neposredno adresiranje. Po prirodi stvari, neposredni argumenti se mogu se naci samo na mestu izvornog argumenta. Na primer:

```
MOV AX, 25
MOV BX,195
```

Prva instrukcija upisuje hekasadekadnu konstantu 25 (dekadnu 37) u registar AX, a druga 195 (dekadno 405) u registar BX.

Memorijski nacini adresiranja

Hipoteticki procesori x86 mogu da koriste tri nacina adresiranja za zadavanje memorijskih argumenata:

```
MOV AX, [1000]
MOV AX, [BX]
MOV AX, [1000+BX]
```

Prva instrukcija koristi *direktno* adresiranje - ona vrednost sa memorijske lokacije sa adrese 1000 (hex) upisuje u registar AX. Druga instrukcija uzima podatak sa lokacije cija je adresa prethodno upisana u registar BX i upisuje ga u registar AX. Ovde se koristi *indirektno* adresiranje. Primetimo da smo umesto dve instrukcije:

```
MOV BX,1000
MOV AX,[BX]
```

mogli krace napisati MOV AX, [1000]. Medutim, ovaj kraci nacin nije i bolji jer omogucava upis samo jedne vrednosti, a prethodni samo promenom sadržaja registra BX daje mogucnost da istom instrukcijom MOV AX, [BX] pristupamo razlicitim memorijskim lokacijama.

Poslednja instrukcija koristi *indeksni* (relativni) nacin adresiranja. Ovaj nacin adresiranja zgodan je za pristupanje razlicitim elementima niza, poljima slogova i komponentama drugih strukturiranih podataka. Konkretno, naredba MOV AX, [1000+BX] upisuje u AX podatak sa lokacije koja je za 1000 (hex) bajtova udaljena od lokacije cija se adresa nalazi u BX.

Kodiranje instrukcija

Mada ne možemo proizvoljno da dodeljujemo opkodove instrukcijama x86, imajte na umu da CPU koristi logicka kola za dekodiranje opkodova i proizvodi upravljacke akcije u skladu s njima. Tipican CPU opkod koristi odreden broj bitova u opkodu za zadavanje klase instrukcije i odgovarajuci broj bitova za dekodiranje svakog od argumenata. Neki sistemi (CISC) dekodiraju ova polja na vrlo kompleksan nacin što rezultuje kompaktnim instrukcijama. Drugi sistemi (RISC) dekodiraju opkodove na vrlo jednostavan nacin, što podrazumeva da se cesto neki bitovi uopšte i ne koriste (npr. kad je instrukcija bez argumenata bitovi za zadavanje argumenata) i time je ogranicen (redukovan) broj instrukcija koje se mogu koristiti. Familija Intel 80x86 je pravi predstavnik CISC koncepcije i ima jedan od najkompleksijnih sistema za dekodiranje. Buduci da je namena hipoteticke familije x86 jedino da prezentira koncept dekodiranja instrukcija bez zalaženja u specificnosti karakteristicne za realnu familiju 80x86 mi cemo sada na pojednostavljen nacin prikazati CISC dekodiranje.

Instrukcije sa dva argumenta

Tipicna x86 instrukcija uzima jedan od formata prikazanih na slici 1. Osnovne instrukcije su bilo 1 bilo 3 bajta duge. Opkod se nalazi u jednom bajtu i sastoji se od 3 polja. Prvo polje (tri najstarija bita) definiše klasu instrukcija. Razume se, to nije dovoljno da se kodira 20 instrukcija, ali koristicemo trikove (što Intelovi projektanti obožavaju). Kao što sa slike možete videti, dve od mogucih 8 vrednosti ovog polja iskorišcene su za naredbu MOV, 5 za instrukcije obrade (aritmeticke i logicke) sa dva argumenta, a vrednost 000 rezervisana je za specijalne instrukcije. Specijalne instrukcije imaju mehanizam koji omogucava da se polje za nedostajuci argument koristi za zadavanje o kojoj se od specijalnih instrukcija radi.

OPKOD

16-bitno polje koje je prisutno jedino kod JMP instrukcija ili ako se za memorijski argument koristi neposredno, indirektno ili indeksno adresiranje

iii	**rr**	**mmm**
000 = special	00 = AX	000 = AX
001 = OR	01 = BX	001 = BX
010 = AND	10 = CX	010 = CX
011 = CMP	11 = DX	011 = DX
100 = SUB		100 = [BX]
101 = ADD		101 = [xxxx+BX]
110 = MOV reg, reg/mem/imm		110 = [xxxx]
111 = MOV mem, reg		111 = const

Slika 1: Kodiranje osnovnih instrukcija procesora x86

Da bi se odredio pojedanicni opkod, treba da selektujete odgovarajuce bitove iz *iii, rr* i *mmm* polja. Npr, pri kodiranju MOV AX, BX prvo se beleži iii=110 (MOV reg,...) zatim rr=00 (AX) i mmm=001 (BX). Tako ovu instrukciju kodiramo u jednom bajtu: 11000001 odnosno 0C0h.

Neke od instrukcija zahtevaju više od jednog bajta. Npr. MOV AX,[1000] koja puni registar AX sadrž ajem memorijske lokacije 1000. Njen opkod bice 11000110 odnosno 0C6h. I da smo kodirali instrukciju MOV AX, [2000] njen prvi bajt takode bi bio 0C6h, jer u njemu ne zadajemo konkretnu memorijsku lokaciju vec samo nacin adresiranja memorijskog argumenta. Sledeca dva bajta definišu o kojoj lokaciji se radi. 16-bitna adresna konstanta koja sledi neposredno za bajtom sa opkodom u mladem bajtu (prvom koji ide iza opkoda) imace vrednost 00h, a u starijem 10h (ili 20h za zadavanje adrese 2000).

✍ 1. Kodirajte sledece instrukcije:

AND CX,BX _____ ADD AX,14_____

SUB AX,DX _____ CMP BX, 0 _____

Instrukcije sa jednim argumentom

Specijalni opkodovi omogucavaju da se proširi skup raspoloživih instrukcija. Ovaj opkod sa tri nule na pocetku koristi se za zadavanje instrukcija sa jednim argumentom ili bez argumenata. (sl.2 i 3). Imamo 4 klase instrukcija sa 1 argumentom. 00 je rezervisano za dalju ekspanziju, opkod 01 ukazuje na JMP instrukciju, 10 na NOT, a 11 se ne koristi i možemo mu (potencijalno) dodeliti neku novu instrukciju sa jednim argumentom, ali za sada bi se ovaj opkod proglasio kao greška. Projektanti procesora obicno ostavljaju mogucnost da prošire reporoar instrukcija na novim procesorima, recimo Intelovi projektanti su ovo maksimalno iskoristili kada su sa 80286 prelazili na 80386).

OPKOD

16-bitno polje koje je prisutno jedino kod JMP instrukcija ili ako se argument NOT instrukcije zadaje indirektnim ili indeksnim adresiranjem

ii	**mmm**
00 = inst. bez argumenata	000 = AX
01 = JMP	001 = BX
10 = NOT	010 = CX
011 = ilegalna (rezervisana)	011 = DX
	100 = [BX]
	101 = [xxxx+BX]
	110 = [xxxx]
	111 = const

Slika 2: Kodiranje instrukcija sa jednim argumentom

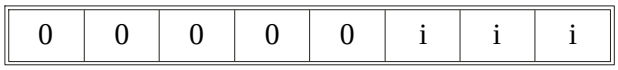
✍ 2. Kodirajte sledece instrukcije:

NOT BX _____ NOT [BX] _____

NOT [1000] _____ NOT [1000+BX] _____

Instrukcije bez argumenata

Poslednja grupa instrukcija je bez argumenata. Ona u prva dva polja ima nule (000 00), a trece polje determiniše o kojoj od njih se radi. BRK instrukcija zaustavlja CPU dok ga korisnik rucno ne resetartuje, što je zgodno kada želimo da analiziramo medurezultate. IRET vraca kontrolu iz rutine za opsluživanje prekida (Interrupt Service Routine) i o njoj cemo govoriti kasnije. HALT prekida izvršavanje programa. GET cita heksadekadnu vrednost sa ulaza i upisuje je u AX, a PUT vrednost iz AX prikazuje na izlazu.



iii

000 = ilegalna 100 = IRET
001 = ilegalna 101 = HALT
010 = ilegalna 110 = GET
011 = BRK 111 = PUT

Slika 3: Kodiranje instrukcija bez argumenata

Instrukcije skoka

Na procesorima x86 imamo 7 instrukcija skoka (sl.4). One imaju sledecu formu:

Jcc adresa

JMP upisuje 16-bitnu vrednost koja sledi neposredno iza opkoda u IP - to je adresa instrukcije na koju ce CPU preneti upravljanje. JMP je primer instrukcije koja vrši bezuslovnu predaju upravljanja. Ostalih 6 su instrukcije uslovne predaje upravljanja. One testiraju jedan od flegova i ako je uslov koji pretpostavljaju ispunjen, vrše prenos upravljanja, a u suprotnom se nastavlja sa izvršavanjem sledece instrukcije na koju vec ukazuje IP. To su instrukcije JA, JAE, JB, JBE, JE i JNE koje testiraju uslove ">", ">=", "<", "<=", "=", "< >" respektivno. Ove instrukcije pišu se neposredno iza CMP koja na odgovarajuci nacin postavi flegove. Napominjemo da se može zadati 8 opkodova (jer odgovarajuce polje ima 3 bita), ali x86 procesori koriste samo 7 od njih, dok se vrednost 111 u ovom polju tretira kao ilegalna i dekoder prijavljuje grešku.

OPKOD
iii

16-bitno polje koje je uvek prisutno i sadrži adresu na koju ce skok predati upravljanje (odnosno adresu koja ce biti upisana u IP)

000 = JE 100 = JA
001 = JNE 101 = JAE
010 = JB 110 = JMP
011 = JBE 111 = ilegalan

Slika 4: Kodiranje instrukcija skoka

✍ 3. Ako labeli a odgovara adresa 0003, kodirajte sledece instrukcije

JB a _____ JMP a_____

4. Kodirajte sledece instrukcije uz pretpostavku da program pocinje na adresi 0.

	Binarni opcod	Heksadekadni argument
mov cx, 0	_____	
a: get	_____	
put	_____	
add ax, ax	_____	
put	_____	
add ax, ax	_____	
put	_____	
add ax, ax	_____	
put	_____	
add cx, 1	_____	_____
cmp cx, 4	_____	
jb a	_____	_____
halt	_____	

Izvršavanje instrukcija korak po korak

Procesori x86 nemaju hardverski izvedene naredbe i zato nisu u mogucnosti da izvršavaju instrukcije u jednom taktu. CPU za svaku instrukciju izvršava niz koraka - mikroradnji kojima realizuje instrukciju. Npr. CPU koristi sledece korake za realizaciju **MOV reg, reg/mem/imm** instrukciju:

- ✍K1 - Donošenje bajta sa opkodom instrukcije iz memorije;
- ✍K2 - Uvecanje IP da pokazuje na sledeci bajt;
- ✍K3 - dekodiranje instrukcije;
- ✍K4 - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije;
- ✍K5 - Ako je potrebno, uvecanje IP da pokazuje na bajt iza argumenta;
- ✍K6 - Ako je potrebno, izracunavanje adrese u memoriji za odredišni argument
- ✍K7 - Donošenje izvornog argumenta;
- ✍K8 - Upisivanje donetog argumenta na odredište.

Objašnjenje kako se izvršava instrukcija korak po korak može pomoci da se razume šta CPU u stvari radi. U prvom koraku CPU uzima bajt sa opkodom instrukcije iz memorije. Ovo radi tako što šalje vrednost iz IP na magistralu adresa i zatim cita bajt sa te adrese. To traje jedan takt (radi jednostavnijeg objašnjenja smatracemo da memorijski ciklus traje jedan takt.) Posle uzimanja opkoda, CPU uvecava IP tako da pokazuje na sledecu memorijsku lokaciju. Ako tekuca instrukcija zauzima više bajtova, IP sada ukazuje na argument instrukcije. Ako tekuca instrukcija zauzima samo 1 bajt, IP ce pokazivati na opkod sledece instrukcije. Ovo se izvršava u sledecem taktu.U narednom koraku dekodira se instrukcija kako bi se ustanovilo šta treba

raditi. Ovo kazuje CPU pored ostalih stvari da li treba donositi neki argument iz memorije. I ovaj korak traje jedan takt.

Za vreme dekodiranja CPU utvrduje tip argumenata koje instrukcija zahteva. Ako instrukcija koristi 16-bitnu konstantu (tj. ima polje mmm 101, 110 ili 111) onda CPU donosi konstantu iz memorije. Ovaj korak može trajati 0, 1 ili 2 takta. Ako nema argumenta, trajace 0 taktova, ako je 16-bitni podatak na parnoj adresi - 1 takt, a ako je na neparnoj adresi - 2 takta.

Ako pak CPU treba da donese 16-bitni memorijski argument, mora uvecati IP za 2 jer sve instrukcije x86 rade sa dvobajtnim argumentima. Ova operacija takode traje 0 ili 1 takt u zavisnosti ima li ili nema argumenta.

Zatim CPU izracunava adresu memorijskog argumenta. Ovaj korak se izvršava jedino ako mmm polje instrukcije sadrži vrednost 101 ili 100. Ako sadrži 100, CPu izracunava zbir BX registra i 16-bitne konstante što traje 2 ciklusa - jedan da se uzme vrednost iz BX i jedan da se sabere sa sa xxx. Ako mmm polje sadrži 100, CPU uzima vrednost sa memorijske adrese što zahteva jedan takt. Ako mmm ne sadrži ni 100 ni 101, onda se za izracunavanje EA ne troši ni jedan ciklus.

Uzimanje argumenta može da traje 0, 1 2 ili tri takta, zavisno od argumenta. Ako je argument konstanta (mmm=111), onda ovo treje 0 ciklusa, jer je konstanta u prethodnom koraku vec doneta iz memorije. Ako je argument u registru (mmm=000, 001, 010 ili 011), onda ovaj korak traje jedan takt. ako se radi o reci koja je poravnata na parnu adresu potrebna su dva, ako je rec na neparnoj adresi - tri ciklusa.

Poslednji korak u realizaciji MOV instrukcije je upis vrednosti na odredišnu lokaciju. Kako je kod procesora x86 odredište uvek procesorski registar, ovaj korak izvršava se u jednom taktu.

Kao što ste videli, izvršavanje MOV instrukcije traje izmedu 5 i 11 taktova, zavisno od tipa argumenta i njegovog poravnanja u memoriji.

Instrukcija **MOV mem,reg** realizuje se kroz sledece korake:

- ✍K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)
- ✍K2 - Uvecanje IP da pokazuje na sledeci bajt (1 takt);
- ✍K3 - dekodiranje instrukcije (1 takt);
- ✍K4 - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije (0-2 takta);
- ✍K5 - Ako je potrebno, uvecanje IP da pokazuje na bajt iza argumenta (0-1 takt);
- ✍K6 - Ako je potrebno, izracunavanje adrese u memoriji za odredišni argument (0-2 ciklusa)
- ✍K7 - Donošenje izvornog argumenta iz registra (1 takt);
- ✍K8 - Upisivanje donetog argumenta na odredište (1-3 takta).

Tajming za poslednje dve instrukcije razlikuje se od drugih MOV instrukcija zato što može da cita podatke iz memorije. Ova verzija instrukcije puni podatke iz registra. Izvršavanje instrukcije traje 5 do 11 taktova.

ADD, SUB, CMP, AND i OR naredbe izvršavaju se u sledecim koracima:

3 Videcete uskoro da instrukcija MOV na stvarnim procesorima 80x86 ima mnogo više formi, jer postoje i druge vrste adresiranja, odredišni argument može biti i u memoriji i postoji mnogo više opštih i specijalizovanih procesorskih registara.

- ✍K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)
- ✍K2 - Uvecanje IP da pokazuje na sledeci bajt (1 takt);
- ✍K3 - dekodiranje instrukcije (1 takt);
- ✍K4 - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije (0-2 takta);
- ✍K5 - Ako je potrebno, uvecanje IP da pokazuje na bajt iza argumenta (0-1 takt);
- ✍K6 - Ako je potrebno, izracunavanje adrese u memoriji za odredišni argument (0-2 ciklusa)
- ✍K7 - Donošenje izvornog argumenta i upucivanje u ALU (0-3 takta);
- ✍K8 - Naredivanje ALU da izvrši zadatu operaciju (sabere, oduzme, uporedi izvrši logicko sabiranje ili množenje) 1 takt
- ✍K9 - Upisivanje donetog argumenta u odredišni procesorski registar (1 takt).

Ova grupa instrukcija izvršava se za 8 do 17 taktova.

NOT instrukcija slicna je prethodnim, ali se izvršava brže zato što ima samo jedan argument.

- ✍K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)
- ✍K2 - Uvecanje IP da pokazuje na sledeci bajt (1 takt);
- ✍K3 - dekodiranje instrukcije (1 takt);
- ✍K4 - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije (0-2 takta);
- ✍K5 - Ako je potrebno, uvecanje IP da pokazuje na bajt iza argumenta (0-1 takt);
- ✍K6 - Ako je potrebno, izracunavanje adrese u memoriji za odredišni argument (0-2 ciklusa)
- ✍K7 - Donošenje izvornog argumenta iz registra (1 takt);
- ✍K8 - Upisivanje donetog argumenta na odredište (1-3 takta).

Izvršavanje instrukcije traje 6-15 taktova.

Uslovni skokovi realizuju se na sledeci nacin:

- ✍K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)
- ✍K2 - Uvecanje IP da pokazuje na sledeci bajt (1 takt);
- ✍K3 - Dekodiranje instrukcije (1 takt);
- ✍K4 - Donošenje 16-bitne odredišne adrese iz memorije (1-2 takta);
- ✍K5 - Uvecanje IP da pokazuje na bajt iza argumenta (1 takt);
- ✍K6 - Testiranje "manje od" i "jednako" flega (1 takt)
- ✍K7 - Ako je uslov ispunjen, CPU kopira donetu 16-bitnu konstantu u IP (0 ciklusa ako nema grananja, 1 ako treba predati upravljanje)

Bezuslovni skok JMP identican je instrukciji MOV reg,xxx osim što odredišni registar nije AX, BX, CX ni DX vec specijalizovan registar IP.

BRK, IRET, HALT, PUT i GET u ovom trenutku nisu za nas interesantne jer nismo u mogucnosti da procenimo koliko ce taktova trajati buduci da su mnogo složenije od prethodno analiziranih.

Emulator x86 procesora

Program SIMx86.exe koji cemo koristiti za upoznavanje sa programiranjem na simbolickom nivou predstavlja modifikaciju istoimenog programa prof. R. Hajda (*rhyde@cs.ucr.edu)* koju je prethodne školske godine uradio ucenik 3b Matematicke gimnazije Pešikan Darko. Ovaj program sadrži ugraden editor, asembler, dibager i interrupter za x86 hipoteticke procesore. Uz njegovu pomoc ilustovacemo kako se pišu elementarni x86 simbolicki programi, kako se prevode - asembliraju, kako se pregleda i menja sadržaj memorijskih lokacija i kako se korak po korak izvršavaju mašinski programi. Ilustovacemo kako se koristi memorijski mapirani ulaz/izlaz, DMA i pojednostavljeni sistem prekida. Pokazacemo takode kako se vrši

programska modifikacija i kako bi se mogle uvesti nove instrukcije u repertoar naredbi.

Razlike izmedu x86 procesora

Svi x86 procesori imaju isti skup instrukcija, iste nacine adresiranja i izvršavaju instrukcije koristeci iste sekvence mikro naredbi. Pa u cemu se onda razlikuju? Zašto se ne upoznaje jedan procesor umesto cetiri?

Glavni razlog za ovu vežbu je da se objasne razlike u performansama koje se postižu sa 4 hardverske mogucnosti: *bafer naredbi* (prefetch queue), *ugradene keš memorije* (chache), *tekuce linije* (pipelines) i *superskalarna arhitektura*. Procesor 886 je jednostavan uredaj koji nema implementirane ove izvanredne mogucnosti. Procesor 8286 ima implementiran bafer naredbi, 8486 ima bafer naredbi, keš i tekuce linije, dok procesor 8686 ima superskalarnu arhitekturu. Analiziranje organizacije svih ovih procesora omogucice vam da uvidite prednosti odgovarajucih rešenja i da prilikom prograrimaranja realnih procesora Intelove familije 80x86 na najbolji moguci nacin iskoristite te prednostii.

instrukcija ✍ nacin adresiranja✍	MOV oba formata	ADD, SUB, CMP, AND, OR	NOT	JMP	Jcc
reg, reg	5	7			
reg, xxxx	6-7	8-9			
reg, [BX]	7-8	9-10			
reg, [xxxx]	8-10	10-12			
reg, [xxxx+BX]	10-12	12-14			
[BX], reg	7-8				
[xxxx], reg	8-10				
[xxxx+BX], reg	10-12				
reg			6		
[BX]			9-11		
[xxxx]			10-13		
[xxxx+BX]			12-15		
xxxx				6-7	6-9

Slika 5: Trajanje izvršavanja instrukcija procesora 886

Procesor 886

Procesor 886 je najsporiji procesor x86 familije. Faza izvršavanja svake instrukcije vec je prodiskutovana. Npr, MOV traje izmedu 5 i 12 clck ciklusa u zavisnosti od argumenata. Tabela na sl. 1 ilustruje vreme trajanja svih instrukcija.

Moramo da napomenemo tri važne stvari:

1) duže instrukcije traže duže vreme izvršavanja

2) instrukcije koje se ne obracaju memoriji u opštem slucaju se brže izvršavaju; ovo je specijalno naglašeno kada se prilikom obracanja memoriji mora umetati više ciklusa cekanja (u tabeli se pretpotstavlja da se koristi idealna memorija sa 0 ciklusa cekanja)

3) instrukcije koje koriste složenije nacine adresiranja rade sporije.

Instrukcije koje koriste procesorske registre su krace, ne pristupaju memoriji i ne koriste komplikovane nacine za adresiranje memorije. To je razlog zašto se maksimalno treba truditi da se argumenti cuvaju u procesorskim registrima.

Procesor 8286

Kljuc za povecanje brzine procesora je paralelno izvršavanje instrukcija. Kada bi izvršavanje trajalo kao u tabeli 1, procesor koji bi mogao istovremeno da izvršava 2 instrukcije radio bi na istom taktu u proseku dvostruko brže. Medutim, rešenje da se po dve instrukcije izvršavaju u istom taktu nije nimalo jednostavno. Mnogi koraci u izvršavanju svake instrukcije dele funkcionalne jedinice u CPU. Funkcionalna jedinica je grupa logickih kola koja izvršavaju neku operaciju, npr. ALU i CU. Funkcionalne jedinice mogu da izvršavaju samo po jednu operaciju u jednom taktu. Dakle, nije moguce realizovati da dve instrukcije dele jednu funkcionalnu jedinicu u istom taktu (npr. da povecavaju IP ili koriste sabirac). Drugi problem koji može da nastupi u vreme kada dve instrukcije koriste razlicite funkcionalne blokove je što je moguce da izvršavanje jedne instrukcije zavisi od rezultata koji je proizvela ona druga. Npr. poslednja dva koraka u instrukciji ADD ukljucuju sabiranje dve vrednosti i pamcenje zbira. Postoje takode neki drugi resurski koje CPU ne može da deli izmedu koraka neke instrukcije. Na primer,. postoji samo jedna magistrala podataka i CPU ne može donositi naredbu u isto vreme kada se u memoriju upisuje rezultat druge instrukcije. Trik pri projektovanju CPU koji omogucava da se izvršava više koraka paralelno je da se preklapaju oni koraci koji ne proizvode konflikte ili se uvodi dodatna logika da bi se dve ili više instrukcija izvršavale u razlicitim funkcionalnim blokovima.

Navedimo opet korake u izvršavanju MOV reg, mem/reg/imm instrukcije:

- ✍K1 - Donošenje bajta instrukcije iz memorije;
- ✍K2 - Uvecanje IP da pokazuje na sledeci bajt;
- ✍K3 - dekodiranje instrukcije;
- ✍K4 - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije;
- ✍K5 - Ako je potrebno, uvecanje IP da pokazuje na bajt iza argumenta;
- ✍K6 - Ako je potrebno, izracunavanje adrese u memoriji za odredišni argument
- ✍K7 - Donošenje izvornog argumenta;
- ✍K8 - Upisivanje donetog argumenta na odredište.

Prva operacija koristi vrednost IP registra (pa zato ne možemo da je preklopimo sa uvecanjem IP) za izbor memorijskog registra u kome se nalazi opkod instrukcije. Svi sledeci koraci zavise od opkoda koji se donosi iz memorije, pa ga zato ne možemo preklopiti ni sa jednim od njih.

Drugi i treci korak ne dele ni jedan funkcionalan blok, dekodiranje nema nikave veze sa uvecanjem sadržaja brojaca naredbi. To znaci da se jednostavnom modifikacijom kontrolne jedinice može postici da se oni relaizuju uporedo. Ovo može uštedeti jedan takt u realizaciji MOV instrukcije.

Treci i cetvrti. korak ne mogu se izvršavati paralelno jer se prvo mora dekodirati instrukcija da bi se utvrdilo da li CPU treba da donosi 16-bitni argument iz memorije. Istina, možemo projektovati CPU tako da donosi argument uvek, pa ako bude potreban da može da se koristi. Postoji jedan problem u vezi ove ideje - moramo imati adresu argumenta koji se donosi (vrednost iz IP) i moramo da cekamo dok se ne završi uvecanje IP. Kada bismo inkrementirali IP u isto vreme kada se instrukcija dekodira, ne bismo morali da cekamao sledeci ciklus za donošenje argumenta.

Kako su koraci K4, K5 i K6 neobavezni, postoji više razlicitih sekvenci izvršavanja:

#1: K4, K5, K6, K7 (npr. MOV AX,[1000+BX])
#2: K4, K5, K7 (npr. MOV AX,[1000])
#3: K6, K7 (npr. MOV AX,[BX])
#4: K7 (npr. MOV AX,BX)

U navedenim sekvencama K7 uvek završava skup prethodnih mikro operacija. Prema tome, K7 se ne može izvršavati uporedo sa koracima od 4 do 6 . K6 takode sledi za K4. K5 ne može se izvršavati paralelno sa K4, jer K4 koristi vrednost IP registra, ali može da se izvršava paraleno sa bilo kojim drugim korakom. Na osnovu ove kratke analize vidimo da možemo da skratimo trajanje izvršavanja u prva dva primera za 1 takt na sledeci nacin:

#1: K4, K5/6, K7
#2: K4, K5/7
#3: K6, K7
#4: K7

Razume se, ne možemo da prekrijemo izvršavanje K7 i K8 jer ce sigurno uzimati vrednosti pre no što ih smeštaju. Kombinovanjem ovih koraka, možemo doci do sledecih koraka u realizaciji MOV naredbe:

- K1' - Donošenje bajta instrukcije iz memorije;
- K2' - Dekodiranje instrukcije i ažuriranje IP;
- K3' - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije;
- K4' - Ako je potrebno, izracunavanje adrese u memoriji za odredišni argument
- K5' - Donošenje izvornog argumenta i ažuriranje IP;
- K6' - Upisivanje donetog argumenta na odredište.

Dodavanjem samo malo logike u CPU možemo uštedeti jedan ili 2 ciklusa u izvršavanju naredbe MOV. Slican nacin optimizacije može dosta da doprinese i za preostale instrukcije.

Drugi problem vezan za izvršavanje instrukcije MOV je poravnavanje (alignment) koda. Pretpostavimo da se instrukcija MOV AX,[1000] nalazi na lokaciji 100 u memoriji. CPU troši jedan ciklus za donošenje opkoda i posle dekodiranja instrukcije ako se utvrdi da je potreban 16-bitni argument troši još 2 dodatna ciklusa da donese argument iz memorije (zato što je argu-

ment na neparnoj adresi 101). Najtužnije je što se jedan ciklus gubi uzalud jer se mladi bajt mogao doneti skupa sa operacionim kodom (x86 je 16-bitni procesor sa 16-bitnom magistralom podataka). Kada bi u procesoru postojao samo jednobajtni bafer uz malo dodatne logike prilikom svakog donošenja argumenta štedeo bi se po jedan ciklus.

Kada dodamo registar za prihvatanje podataka (bafer), razmotrimo neke dodatne optimizacije koje mogu da koriste istu logiku. Npr. šta se dešava sa istom MOV instrukcijom posle izvršavanja. Ako smo doneli opkod i mladi bajt argumenta u prvom ciklusu, prilikom donošenja starijeg bajta argumenta donecemo i opkod naredne instrukcije. Ako sacuvamo ovaj opkod do kraja izvršavanja tekuce instrukcije, opet cemo uštedeti jedan ciklus jer nece biti potrebno donošenje novog opkoda. I više od toga, mogli bismo da organizujemo dekodiranje naredne instrukcije paralelno sa izvršavanjem prethodne i tako uštedeti još jedan ciklus. Ako bismo prilikom svakog obracanja memoriji donosili po dva bajta, proseku bismo štedeli po 2 ciklusa za vreme izvršavanja 50% instrukcija.

[ta mislite, možemo li nešto da radimo i za vreme izvršavanja onih drugih 50% instrukcija? Odgovor je - da. Imajte na umu da se za vreme izvršavanja instrukcije MOV ne pristupa memoriji u svakom taktu. Na primer, takt u kome se vrši upis u procesorski registar predstavlja prazan hod za BIU. U vreme dok je BIU besposlen, mogli bismo unapred donositi (pre-fetch) opkodove i argumente i privremeno ih odlagati u prihvatne registre procesora.

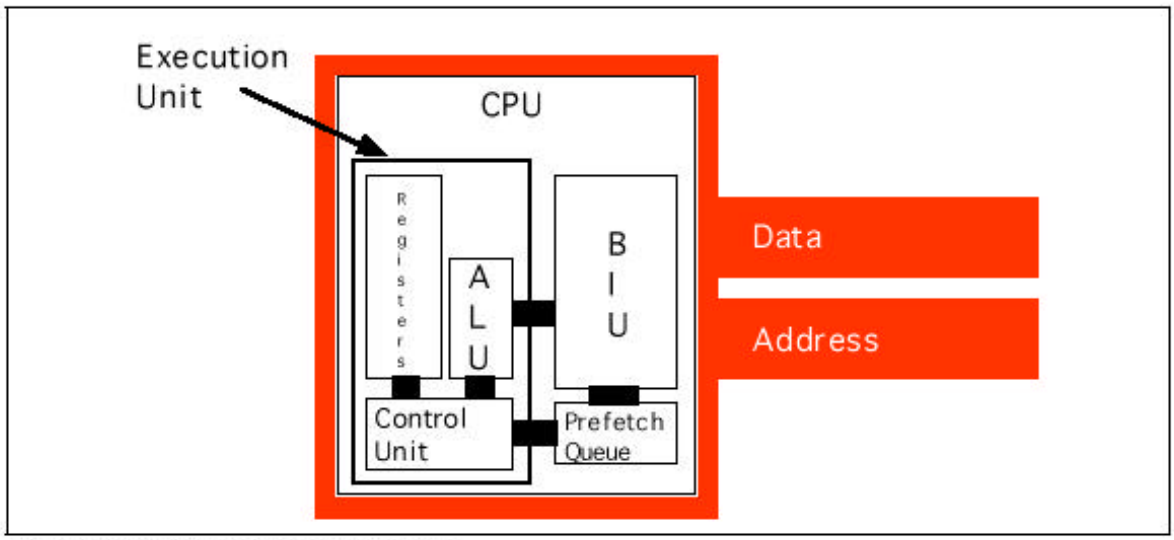
Najveca prednost koju ima 286 procesor u odnosu na x86 je *prefetch queue* - bafer za cuvanje unapred donetih naredbi koji cemo krace nazivati *bafer naredbi*. Uvek kada CPU ne koristi BIU, BIU donosi dva dodatna bajta u bafer naredbi. Uvek kada je CPU potreban opkod ili argument, uzima tekuci bajt iz bafera naredbi. Kako BIU donosi po dva bajta u jednom koraku, a CPU generalno koristi po manje od dva bajta u jednom taktu, svaki bajt koji je procesoru potreban obicno je vec u baferu instrukcija i na raspolaganju je procesoru. Napominjemo da ne postoji garancija da ce svi opkodovi i argumenti biti u baferu naredbi kadgod su potrebni procesoru. Na primer, instrukcija JMP 1000 cini da sadržaj bafera naredbi postane nevažeci. Ako se ova instrukcija nalazi na adresi 400, 401 i 402 u memoriji, bafer naredbi ce u vreme njenog izvršavanja imati vrednosti sa memorijaskih lokacija 403, 404, 405 itd. Posle upisa vrednosti 1000 u IP, bajtovi sa lokacija 403, 404 itd. nisu potrebni. Zato sistem mora da izgubi izvesno vreme dok se bafer instrukcija opet ne napuni.

Drugo poboljšanje postižemo preklapanjem koraka dekodiranja sa poslednjim korakom u fazi izvršenja prethodne naredbe. Pošto procesor iskoristi argument, prvi naredni bajt u baferu naredbi sadrži opkod i CPU može da ga dekodira pre no što dode na bafer njegovo izvršavanje, Razume se, ako tekuca instrukcija izmeni sadržaj IP, vreme utrošeno na dekodiranje nabaferne instrukcije bice izgubljeno, ali buduci da se dekodiranje obavljalo upobafero sa izvršavanjem tekuce instrukcije, sistem se nece usporiti.

Sekvenca optimizacije zahteva neke izmene u hardveru. Blok dijagram sistema dat je na sl. .2. Izvršavanje instrukcije sada podrazumeva sledece dogadaje koji se dešavaju u pozadini.

Dogadaji vezani za opsluživanje bafera nabaferbi

Ako bafer naredbi nije prazan (u opštem slucaju može da cuva izmedu 8 i 32 bajta, zavisno od procesora) i BIU je besposlen u tekucem taktu, uzima sledecu rec iz memorije sa adrese na koju ukazuje IP na pocetku takta. (ako je u IP neparan broj, ova operacija donosi samo jedan bajt)



Slika 6: Procesor koji koristi bafer naredbi (Prefetch Queue)

Ako je dekoder instrukcija besposlen i tekuca instrukcija ne zahteva argumente, pocinje sa dekodiranjem opkoda koji se nalazi na pocetku bafera instrukcija (ako je prisutan). U suprotnom, pocinje sa dekodiranjem treceg bajta iz bafera instrukcija (ako je prisutan). U slucaju da odgovarajuci opkod nije prisutan u baferu naredbi, ovaj dogadaj ne nastupa.

Vreme izvršavanja instrukcija ima nekoliko optimistickih pretpostavki, recimo da se svi potrebni opkodovi i argumenti instrukcija nalaze u u baferu instrukcija i da uvek može da se dekodira opkod tekuce instrukcije. Ako ovo nije tacno izvršavanje instrukcija na 8286 ce cekati dok sistem ne donese podatke iz memorije ili dekodira instrukcije. Slede koraci za svaku od 8286 instrukcija:

MOV reg,mem/reg/imm

- ✍K1 - Ako je potrebno, izracunava se zbir [xxx+BX] - 1 ciklus
- ✍K2- Uzima se izvorni argument 0 ciklusa ako je konstanta (vecje doneta u bafer naredbi), 1 takt ako je registarski 2 takta ako je memorijska rec sa parne adrese ili 3 takta ako je memorijska rec sa neparne adrese
- ✍K3- Upis rezultata u odredišni registar - 1 takt

MOV mem,reg

- ✍K1 - Ako je potrebno, izracunava se zbir [xxx+BX] - 1 ciklus
- ✍K2- Uzima se izvorni argument iz registra - 1 takt
- ✍K3- Upis rezultata na odredište - 2 takta ako je odredište memorijska rec sa parne adrese ili 3 takta ako je memorijska rec sa neparne adrese

INSTR reg, mem/reg/imm (INSTR=ADD, SUB, CMP, AND, OR)

- ✍K1 - Ako je potrebno, izracunava se zbir [xxx+BX] - 1 ciklus
- ✍K2- Uzima se izvorni argument 0 ciklusa ako je konstanta (vec je doneta u bafer naredbi), 1 takt ako je registarski 2 takta ako je memorijska rec sa parne adrese ili 3 takta ako je memorijska rec sa neparne adrese
- ✍K3- Uzimanje vrednosti prvog argumenta (registar) 1 takt
- ✍K4- Izracunavanje (sume, razlike itd.)1 takt
- ✍K5 Upis rezultata u odredišni registar - 1 takt

NOT mem/reg

- ✍K1 - Ako je potrebno, izracunava se zbir [xxx+BX] - 1 ciklus
- ✍K2- Uzima se izvorni argument 0 ciklusa ako je konstanta (vec je doneta u bafer), 1 takt ako je registarski 2 takta ako je memorijska rec sa parne adrese ili 3 takta ako je memorijska rec sa neparne adrese
- ✍K3- Izracunavanje logicke NOT vrednosti 1 takt
- ✍K4 Upis rezultata na odredište - 1 takt ako je procesorski registar, - 2 takta ako je odredište memorijska rec sa parne adrese ili 3 takta ako je memorijska rec sa neparne adrese

Jcc xxx (uslovni skokovi cc= A, AE, B, BE, E, NE)

- ✍K1- Testiraju se tekuci flegovi (manje od i jednako) - 1 takt
- ✍K2- Ako vrednost flegova odgovarajuca CPU kopira 16-bitni argument u IP registar (1 takt)

JMP xxxx

- ✍K1- CPU kopira 16-bitni argument u IP registar (1 takt)

Kao i kod 886, za ostale instrukcije ne možemo da procenimo vreme izvršavanja.

Ako želite da napišete programe koji ce da rade brzo, izbegavajte naredbe skokova što više možete. Napominjemo da naredbe uslovne predaje upravljanja cine sadržaj bafera naredbi nevaljanim jedino ako se ostvari uslov za skok. Ako se uslov ne ostvari, nastavlja se sa izvršavanjem naredne instrukcije i sve one koje su unapred dekodirane, mogu se iskoristiti. Zato prilikom kodiranja stavite verovatnije instrukcije u ne-granu cime cete takode povecati (verovatno) brzinu izvršavanja programa.

instrukcija ✍ nacin adresiranja✍	MOV oba formata	ADD, SUB, CMP, AND, OR	NOT	JMP	Jcc
reg, reg	2	4			
reg, xxxx	1	3			
reg, [BX]	3-4	5-6			
reg, [xxxx]	3-4	5-6			
reg, [xxxx+BX]	4-5	6-7			
[BX], reg	3-4	5-6			
[xxxx], reg	3-4	5-6			
[xxxx+BX], reg	4-5	6-7			
reg			3		
[BX]			5-7		
[xxxx]			5-7		
[xxxx+BX]			6-8		
xxxx				1+pfd*	2 ** 2+pfd

Slika 7: Vreme izvršavanja instrukcija na procesoru 8286

Velicina instrukcije (u bajtovima) takode utice na performanse bafera naredbi. Nikad nije potrebno više od jednog takta da se dovede jednobajtna instrukcija iz memorije, ali su uvek potrebna dva takta za donošenje trobajtne instrukcije. Pri tom CPU nema dovoljno vremena da donese i dekodira drugu i trecu instrukciju dok se izvršava prva. Zato treba da se trudite da u svojim programima što je više moguce koristite kratke insrukcije, jer tako povecavate performanse koje nudi bafer instrukcija.

Tabela na slici 7 daje (optimisticko) vreme izvršavanja za 8286 instrukcije. Pri tome važe sledece oznake:

* trajanje uzimanja i dekodiranja sledece instrukcije
** ako se ne izvrši

Isticemo koliko je brže izvršavanje instrukcije MOV na 8286 u odnosu na 886. Do ovog ubrzanja dolazi zato što bafer naredbi omogucava procesoru preklapanje izvršavanja susednih instrukcija. Ova tabela oslikava preklapanje instrukcija. Napomena za predvidanje: "pretpostavljamo da je opkod prisutan u baferu naredbi i da može biti dekodiran". Pretpostavimo da imamo sledecu sekvencu instrukcija:

```
????:   JMP 1000
1000:   JMP 2000
2000:   JMP CX, 3000[BX]
```

Druga i treca instrukcija ne mogu se izvršiti brzinom koju navodi tabela. Kada modifikujemo IP registar, CPU prazni bafer naredbi, pa nije u mogucnosti da uzme i dekodira narednu instrukciju. U stvari, mora da uzima iz memorije, zatim dekodira udaljenu instrukciju. Jedino ubrzanje koje je moguce postici pri izvršavanju ove instrukcije je uvecanje IP uporedo sa nekim drugim korakom.

Obicno rad sa baferom naredbi poboljšava performanse. To je razlog što je Intel predvideo da svaki od modela procesora 80x86 ima bafer naredbi. Kod svih BIU konstantno donosi podatke u bafer naredbi uvek kada je magistrala podataka slobodna.

bafer naredbi bolje radi što je šira magistrala podataka. 8286 procesor radi mnogo brže nego 886 jer se trudi da uvek ima popunjen bafer naredbi. Pretpostavimo da imamo sledece instrukcije:

```
100:    MOV AX,[1000]
105:    MOV BX,[2000]
10A:    MOV CX, [3000]
```

Kako su AX, BX i CX 16-bitni evo šta se dešava uz pretpostavku da se kompletna prva instrukcija vec nalazi u baferu naredbi i da je dekodirana.

- K1 - Uzimanje opkoda iz bafera naredbi (0 taktova)
- K2 - Dekodiranje instrukcije(0 taktova)
- K3 - Uzima argument iz bafera naredbi (0 taktova)
- K4 - Uzima vrednost drugog argumenta (1 takt). Uvecava IP
- K5 - Upisuje donet argument u odredišni registar (1 takt). Donosi 2 bajta iz niza naredbi u memoriji bi upisuje ih u bafer naredbi. Dekodira sledecu instrukciju.

Ovde je kraj izvršenja prve naredbe. Trenutno su dva bajta u baferu naredbi.

- K6 - uzima opkod bajt iz bafera naredbi (0 taktova)
- K7 - dekodira instrukciju (0 taktova)

✍K8 - Ako instrukcija ima argument , uzima ga iz bafera naredbi (1 ciklus, jer nedostaje jedan bajt)
✍K9 - Uzima vrednost drugog argumenta (jedan ciklus). Uvecava IP
✍K10 Upisuje donetu vrednost u odredišni registar (jedan ciklus) Uzima dva bajta iz niza naredbi u memoriji. Dekodira narednu instrukciju.

Ovo je kraj druge instrukcije. Trenutno su tri bajta u baferu naredbi.

✍K11 - Uzima opkod iz bafera naredbi (0 taktova)
✍K12 - Dekodira instrukciju (0 taktova)
✍K13 - Ako instrukcija ima argument, uzima ga iz bafera naredbi (0 taktova)
✍K14 - Uzima vrednost drugog argumenta (1 takt) Uvecava IP
✍K15 - Upisuje donetu vrednost u odredišni registar (jedan ciklus) Uzima dva bajta iz niza naredbi u memoriji. Dekodira narednu instrukciju

Kao što možete videti, druga instrukcija zahteva jedan takt više od preostalih instrukcija. To je zato što BIU nije mogao da popuni bafer naredbi tako brzo kako CPU izvršava naredbe. Ovaj problem uznemirava kada je bafer naredbi samo nekoliko bajtova dug i mada ne postoji na hipotetickom 8286 procesoru, vrlo je izražen kod realnih 80x86 procesora.

Kao što cemo kasnije videti, 80x86 procesori imaju tendenciju da potroše bafer naredbi vrlo lako. Razume se, kada se rad naredbi isprazni, CPU mora da ceka da mu BIU donosi instrukcije iz memorije što znacajno usporava program. Izvršavanje kracih instrukcija pomaže da se bafer naredbi održi punim. Npr. 8286 može da donosi dve jednobajtne instrukcije u jednom memorijskom ciklusu, ali mu je neophodan 1.5 taktova da donese trobajtnu instrukciju. Obicno je potrebno više vremena za izvršavanje 4 jednobajtnih instrukcija nego za 1 trobajtnu. Ovo daje vremena da se bafer naredbi popuni.

Upamtite: Ako koristite procesor koji koristi bafer naredbi, u svojim programima što je više moguce koristite kratke insrukcije!

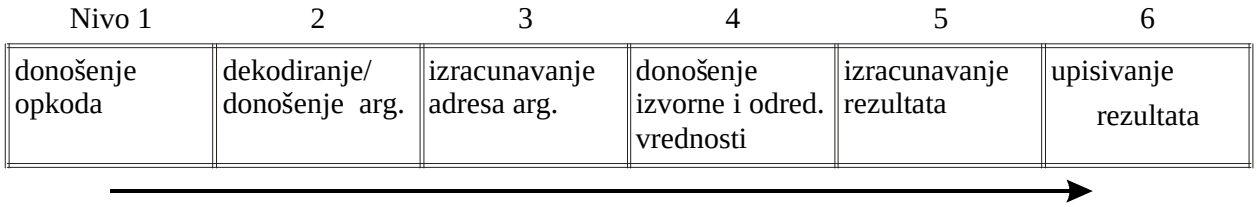
Procesor 8486

Prednost bafera naredbi je što CPU preklapa faze donošenje i dekodiranja sa izvršavanjem instrukcije. Ako se u procesor doda nov hardver, moci ce uporedo da se izvršava više instrukcija. Paralelno izvršavanje instrukcija korišcenjem BIU i EU je specijalan slucaj preklapanja instrukcija koje se postiže korišcenjem tekucih linija. Procesor 8486 ima ugradene tekuce linije koje mu poboljšavaju performanse toliko da u proseku u svakom taktu izvršava po jednu instrukciju.

Tekuce linije

Pretpostavimo da imamo sledece korake pri izvršavanju genericke instrukcije:

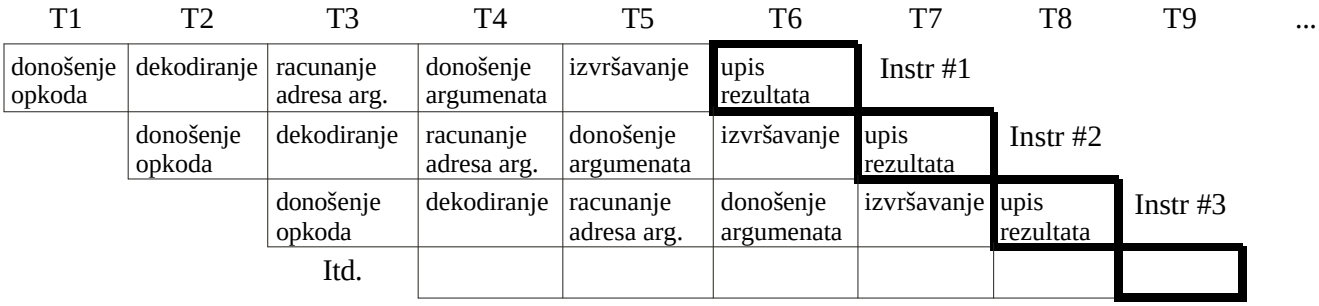
✍K1 -donošenje opkoda
✍K2- dekodiranje opkoda i (paralelno) donošenje potencijalnog 16-bitnog argumenta.
✍K3 - izracunavanje EA kompleksnog nacina adresiranja ako je potrebno
✍K4 -donošenje izvorne vrednosti iz memorije (ako je se radi sa memorijskim argumentom) i vrednost odredišnog registra (ako se koristi)
✍K5 - izracunavanje rezultata
✍K6 - upis rezultata u odredišni registar



Slika 8: Implementacija tekucih linija za izvršavanje instrukcija

Pretpostavimo da ste platili malo dodatne logike i imate ugraden "Miniprocesor" za rad sa navedenim koracima. Organizacija bi izgledala kao na sl. 8.

Ako postoji zaseban hardver za svaki korak, svi mogu da se izvršavaju paralelno. Razume se, u istom trenutku se ne mogu izvršavati razliciti koraci iste instrukcije. Ako imamo n-nivosku tekucu liniju, uporedo se može izvršavati n instrukcija. 8486 ima 6-nivosku tekucu liniju.



Slika 9: Izvršavanje instrukcija na tekucoj liniji

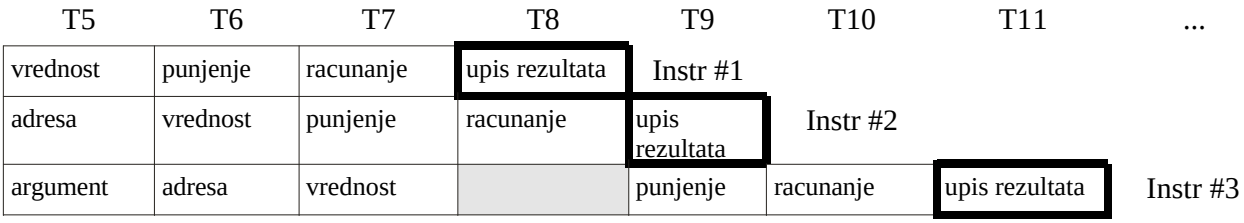
Sl. 9 ilustruje tekuce linije. T1, T2, ... su uzastopni taktovi. U vreme T=T1, CPU donosi opkod tekuce instrukcije iz memorije. U trenutku t=T2 CPU pocinje da dekodira opkod prve instrukcije i uporedo donosi 16-bitova iz bafera naredbi u slucaju da instrukcija ima jedan argument. Ako prva instrukcija nema argumente, CPU "nareduje" donošenje opkoda sledece instukcije. Napomenimo da postoji minimalni konflikt. CPU pokušava da donese sledeci bajt iz bafera naredbi da bi koristio jedan argument, u isto vreme donosi 16-bitova iz bafera naredbi da bi ga koristi kao opkod. Kako da se to uradi u isto vreme? Uskoro cemo videti rešenje.

U trenutku t=T3 CPU izracunava adresu argumenta prve instrukcije, ako treba. Ukoliko to nije potrebno, ne radi ništa vezano za prvu instrukciju, ali u istom trenutku dekorira drugu i donosi ako je potrebno, njene argumente.

U trenutku t=T6 CPU kompletira izvršavanje prve, izracunava rezultat druge itd. i konacno, donosi opkod 6. na tekucu liniju. Od tog trenutka, trenutka kada je napunjena tekuca linija, u svakom taktu kompletira po jednu instrukciju. Naglašavamo da je ovo tacno cak i kada se koriste složeni nacini adresiranja za memorijske argumente.

Prazni odeljci u tekucoj liniji

Nažalost, prethodno opisan scenario je malo pojednostavljen. Postoje usporavanja: može se zahtevati nesekvencijalo izvršavanje instrukcija i ili nastupiti blokiranje magistrale. Oba problema povecavaju prosecno vreme izvršavanja instrukcija.



Prazan odeljak pojavljuje se na ovom mestu zato što Instr #1 donosi podatak u isto vreme kada CPU želi da donese opkod iz memorije

Instr #3 potrebna su 2 takta za završetak izvršavanja zbog praznog odeljka u tek. liniji

Slika 10: Prazni odeljci na tekucoj liniji

Blokiranje magistrale dešava se uvek kada instrukcija traži pristup nekoj memorijskoj lokaciji. Npr. ako MOV mem,reg instrukcija treba da upiše podatak u memoriju i MOV reg, mem instrukcija treba da donese izvornu vrednost, nastupa problem jer CPU ne može u istom taktu da ostvari pristup do dve memorijske lokacije. Jedno jednostavno rešenje blokiranja magistrale je uvodenjem praznih odeljaka (boksova) u tekucu liniju. Kada je CPU suocen sa blokiranjem magistrale, daje prioritet instrukciji koja je više uznapredovala u tekucoj liniji. CPU suspenduje donošenje opkodova dok tekuca instrukcija donosi (ili upisuje) argument. To uzrokuje da nova instrukcija u tekucoj liniji troši 2 takta za izvršavanje umesto jednog (sl. 10).

Ovaj primer je samo jedan od uzroka "otimanja" magistrale. Postoje i mnogi drugi. Npr. kao što je receno ranije, donošenje opkodova instrukcija zahteva pristup baferu naredbi u isto vreme kada CPU treba da donese opkod. Više od toga, kod procesora naprednijih od 8486 (npr. 80486) postoji još razloga za ovo. Medutim, bez obzira na izneto razmatranje vecina instrukcija ce se izvršavati za 1 takt.

Inteligentno korišcenje keš sistema eliminisace mnoge prazne odeljeke u tekucim linijima. U sledecem tekstu posvecenom keširanju objasnicemo kako se to dogada. Ipak, nije moguce u potpunosti, cak i kad se koristi keširanje, izbeci prazne odeljke u tekucoj liniji. Ono što ne možemo hardverski rešiti, rešavamo softverski. Ako treba da koristite memoriju, redukujte potencijalne konflikte na magistrali i vaši programi izvršavace se brže. Korišcenje kracih instrukcija takode smanjuje mogucnost pojave praznih boksova u tekucoj liniji.

[ta se dešava ako instrukcija menja sadržaj IP? Npr. u trenutku završetka instrukcije JMP 100 imamo zapoceto izvršavanje 5 drugih naredbi i teba nam samo jedan takt da kompletiramo sledecu. Medutim , CPU uopšte ne treba da je izvrši, vec treba da donese na izvršavanje instrukciju s lokacije 100. Jedino razumno rešenje da se baci pogled (flash) po tekucoj liniji i pocne sa donošenjem potrebnog opkoda. Ovo znacajno usporava izvršavanje. Na 8486 potrošice 6 taktova pre no što se ta instrukcija izvrši. Razume se, u programima moramo da koristimo i ovakve instrukcije, ali treba da se trudimo da ih što više izbegavamo. Drugi problem je dužina tekuce linije (broj nivoa). Duža tekuca linija omogucava vecu paralelnost u izvršavanju, ali ima i veci gubitak kada se naide na prenos upravljanja.

Bafer naredbi, ugradeni keš i 8486

Projektanti sistema mogu da reše mnoge probleme konflikta na magistrali inteligentnim korišcenjem bafera naredbi i keš podsistema. Oni mogu da projektuju bafer naredbi kao bafer koji prihvata podatke iz niza instrukcija u memoriji i da dizajniru keš sa zasebnim zonama za podatke i naredbe.Obe tehnike unapreduju sistemske performanse i da eliminišu konflikte koji se dešavaju na magistrali.

Naž alost bafer naredbi na 8486 ne radi tako dobro kao na 8286, jer kod 8286 CPU ne pristupa memoriji u svakom taktu, pa se "prazan hod" BIU koristi za donošenje novih instrukcja u bafer naredbi. Kod 8486 medutim CPU stalno pristupa memoriji jer u svakom taktu donosi opkod bajt. Ipak, i kod 8486 se koristi bafer naredbi iz vrlo jednostavnog razloga: BIU u svakom obracanju donosi 2 bajta, ali postoje instrukcije koje su duge samo 1 bajt. Bez bafera naredbi sistem bi gubio donete podatke, cak i kada BIU "greškom" donese opkod naredne instrukcije. Kad je jednom donet u procesor, opkod se skladišti u baferu naredbi i spreman je za dekodiranje.

Npr. ako izvršavamo dve jednobajtne instrukcije zaredom, BIU može da donese oba opkoda u jednom taktu, pa je sledeci slobodan za donošenje dodatnih informacija iz memorije. Razume se, nisu sve instrukcije jednobajtne. 8486 ima dve velicine instrukcija: jednobajtne i trobajtne. Ako se izvršava nekoliko trobajtnih naredbi zaredom, program ce se sporije izvršavati. Npr.

```
MOV AX, 1000
MOV BX, 1000
MOV CX, 1000
ADD AX, 1000
```

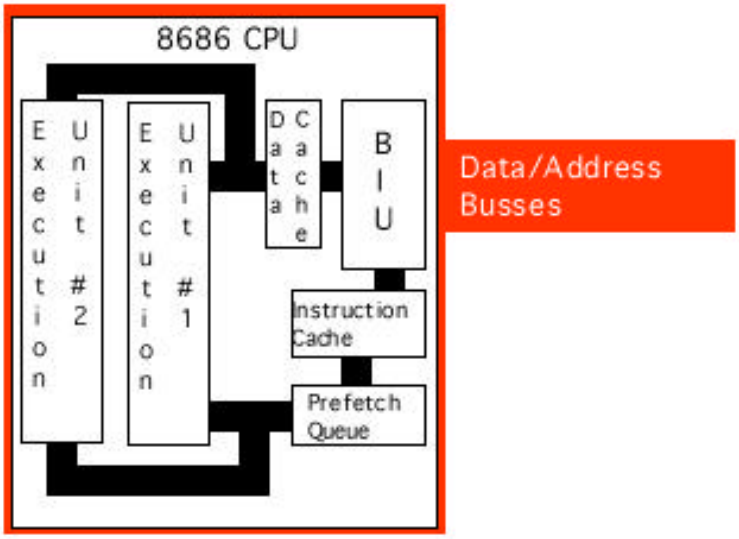
Svaka od ovih instrukcija zauzima po tri bajta - jedan za opkod i dva za 16-bitnu konstantu. Kako je u proseku neophodan po 1.5 takt za njihovo donošenje, to je ukupno potrebno 6 taktova da se sve one donesu u procesor (umesto 4 za koliko bi mogle da se izvrše da su vec donete u bafer naredbi). Još jednom naglašavamo da dobijamo brže programe ako koristimo krace instrukcije. Dejstvo ekvivalento dejstvu prethodnih instrukcija možemo ostvariti i na sledeci nacin:

```
MOV AX, 1000
MOV BX, AX
MOV CX, AX
ADD AX, AX
```

a ovaj kod je svega 6 bajtova dug (prethodni zauzima 12) i može se doneti u procesor za tri takta i brže izvršiti. **Upamtite: krace obicno podrazumeva brže**.

Mada bafer naredbi može da oslobodi bus cikluse i eliminiše konflikte na magistrali, neki problemi ipak ostaju. Pretpostavimo da je prosecna dužina instrukcija za niz instrukcija 2.5 bajtova. (npr. 3 trobajtne i jedna jednobajtna). U tom slucaju magistrala ce biti sacuvana zazetom zbog uzimanja opkodova i argumenata. Nece biti slobodnog vremena za pristupanje memoriji. Uz pretpostavku da neke od ovih instrukcija zahtevaju pristup memoriji, tekuce linije imace prazne boksove i izvršavanje ce biti usporeno.

Pretpostavimo za momenat da CPU ima dva nezavisna memorijska prostora jedan za instrukcije drugi za podatke svaki sa sopstvenom magistralom. Ovaj koncept naziva se harvardska arhitektura jer je prvi put upotrebljen na racunaru koji je projektovan na Harvardu.



Slika 11: Struktura procesora 8486 sa razdvojenim L1 kešom na deo za instrukcije i deo za podatke.

Na harvardskim mašinama nema konflikta na magistrali iz prostog razloga što se jednim putem donose podaci, a drugim instrukcije. Slika 11 prikazuje strukturu procesora 8486 sa razdvojenim L1 kešom na deo za instrukcije i deo za podatke.

Svaki put unutar CPU predstavlja nezavisnu magistralu. Podaci mogu da idu svim putevima uporedo. To podrazumeva da bafer naredbi može da se puni opkodovima iz keša instrukcija i dok EU upisuje podatke u keš podataka. Sada BIU jedino donosi opkodove iz memorije kada ih ne nade u kešu. CPU koristi adresnu i magistralu podataka samo kada cita vrednost koja se ne nalazi u kešu ili kada prazni podatke nazad u operativnu memoriju.

Uzgred budi receno, 8486 radi sa argumentima/opkodovima na vrlo neobican nacin. Dodavanjem još jed- nog dekodera operacija, on dekodira 3 bajta sa pocetka bafera naredbi uporedo. Ako je instrukcija bez argumenata, CPU koristi rezultat prvog dekodera. Ako prva instrukcija koristi argumente, CPU uzima rezultat drugog dekodera.

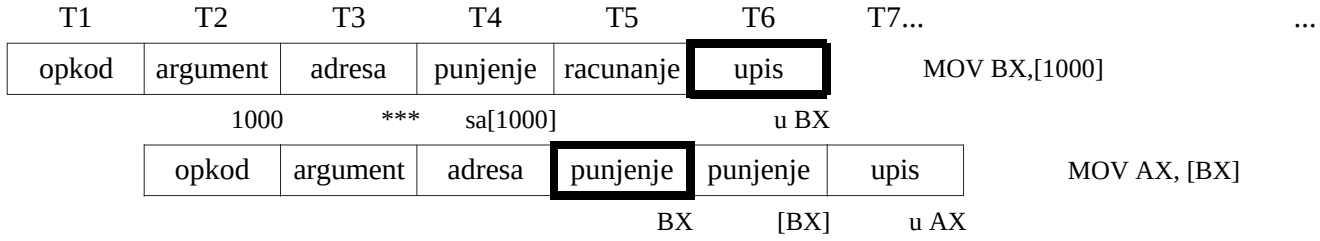
Mada ne možete da kontrolišete prisustvo, velicinu i tip keša u CPU, kao asemblerski programer morate da vodite racuna o tome kako keš radi jer jedino tako možete da pišete optimalne programe. U cip ugradeni keš instrukcija je veoma mali (8KB na 80486) pa što krace instrukcije koristite, bolje cete iskoristiti njegove mogucnosti. Što više instrukcija imate u kešu, manje cete imati konflikata na magistrali. Takode, ako koristite procesorske registre za cuvanje medurezultata, manje cete naprezati keš podatak i on ce rede imati potrebu da se prazni u memoriju. Dakle, **koristite registre kad god je to moguce**.

Meduzavisnost podataka na 8486

Postoji još jedan problem koji se pojavljuje kada se koriste tekuce linije - meduzavisnost podataka (data hazard). Analizirajmo izvršni profil sledece sekvence instrukcija:

```
MOV BX, [1000]
MOV AX, [BX]
```

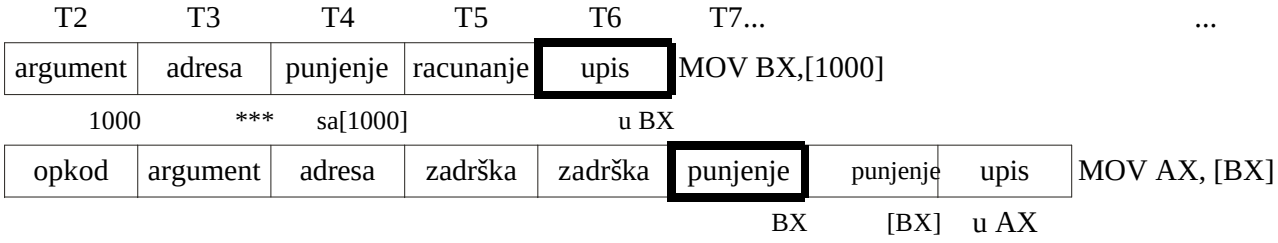
Pri izvršavanju ove dve instrukcije, tekuca linija radi kao na sl. 12. Uocite veliki problem. Ove dve instrukcije uzimaju 16 bitnu vrednost koja se nalazi na memorijskoj lokaciji 1000. Ali ova



Slika 12: Opasnost od meduzavisnosti podataka

sekvenca instrukcija nece raditi korektno. Na nesrecu, druga instrukcija koristi vrednost BX pre no što je prva upisala u BX vrednost sa memorijske lokacije 1000. (T4 i T6 na dijagramu).

CISC procesori kao što je 80x86 sprecavaju meduzavisnost podataka automatski. (RISC cipovi to ne rade. Ovakvu sekvencu instrukcija RISC procesor bi izvršio sa greškom.) Oni umecu prazne odeljke u tekucu liniju da bi sinhronizovali ove dve instrukcije. Pravo izvršenje izgledalo bi kao na dijagramu prikazanom na sl. 13.



Slika 13: Otklanjanje opasnosti od meduzavisnosti podataka

Zadržavanjem druge instrukcije za dva takta, 8486 garantuje da da ce prethodna instrukcija prvo upisati neophodnu vrednost u BX, pa ce tek potom druga instrukcija koristiti taj (korektan) sadržaj. Medutim, vreme izvršavanja druge instrukcije sada je za dva takta duže. Ukoliko ovaj problem imamo na umu, možemo pažljivim pisanjem programa da minimiziramo zadrške koje procesor automatski umece kako bi sprecio meduzavisnost podataka. Buduci da do mešanja podataka dolazi kada je izvorni argument neke instrukcije jednak odredišnom argumentu njoj prethodne. Zato ako imamo sledeci niz instrukcija:

```
MOV CX, 2000
MOV BX, [1000]
MOV AX, [BX]
```

ispisom instrukcija u drugacijem redosledu (razume se, ako tako ne kvarimo logiku programa):

```
MOV BX, [1000]
MOV CX, 2000
MOV AX, [BX]
```

eliminišemo potrebu da umecu zadrške u izvršavanje poslednje instrukcije.
Na procesorima sa tekucim linijima redosled instrukcija može dramaticno da pokvari (ili popravi) brzinu izvršavanja programa. Uvek **analizirajte da li su podaci u uzastopnim instrukcijama vašeg programa meduzavisni i eleminišite ovu situaciju drugacijim redosledom instrukcija**.

Procesor 8686

Uocili smo da procesori sa tekucim linijama poput 8648 izvršavaju prosecno u svakom taktu po jednu instrukciju. Da li se instrukcije mogu izvršavati i brže? Odgovor je - DA, ako procesor ima *superskalarnu* arhitekturu. Ovo karakteristiku ima hipoteticki procesor 8686 i realni procesori familije P6, kao i njihov prethodnik Pentium.

Superskalarni CPU ima više EU koji mogu da rade paralelno. Ako imamo dve ili više instrukcija u baferu instrukcija, one se mogu izvršavati istovremeno.

Postoji mnoštvo prednosti koje donosi superskalarnost. Pretpostavimo da imamo sledeci niz instrukcija:

```
MOV AX, 1000
MOV BX, 2000
```

U ovom primeru nema problema sa meduzavisnošcu podataka i svih šest bajtova nalazi se u baferu instrukcija, pa nema razloga da procesor ne obraduje obe instrukcije istovremeno. Razume se, pod uslovom da procesor ima dve nezavisne jedinice za izvršavanje. Osim ubrzavanja izvršavanja nezavisnih instukcija, superskalarni CPU takode ubrzavaju i izvršavanje delova programa u kojima se pojavljuje meduzavisnost podataka. Ogranicenje 8486 procesora bilo je da kada se otkrije meduzavisnost podataka, instrukcija koja ceka da se za nju proizvede podatak usporava tekucu liniju. Sve instrukcije koje slede moraju da cekaju da se ovaj konflikt reši. Na superskalarnim CPU instrukcije koje slede za onima koje imaju zavisne podatke mogu da nastave sa izvršavanjem na tekucoj liniji sve dok same ne zahtevaju nedostajuci podatak. Ovo olakšava, mada ne otklanja u potpunosti prethodne pobleme pod uslovom da se instukcije pažljivo rasporeduju.

Asemblerski programeri moraju da imaju na umu da nacin pisanja programa dramaticno utice na vreme izvršavanja programa na superskalarnim procesorima. Prvo pravilo koje ste verovatno do sada naucili je - **treba koristiti kratke instrukcije**. Mnogi superskalarni procesori nemaju kompletne dvostruke izvršne jedinice. Oni možda imaju više ALO, FPU i slicno. To podrazumeva da se neke instrukcije mogu izvršavati vrlo brzo, dok druge ne. Treba prostudirati od cega se sastoji konkretan procesor i shodno tome komponovati instrukcije na odgovarajuci nacin kako bi se mogao izvuci maksimum iz tog procesora.