7. Programiranje u asembleru

Šta ćemo učiti

- **Arhitektura procesora familije x86**
- **Skup instrukcija i načini adresiranja**
- **Kodiranje i i izvršavanje instrukcija**
- **Emulator x86 procesora**
- **Razlike izmedu x86 procesora**

Kako je familija mikroprocesora 80x86 veoma kompleksna, poznavanje svih njenih članova predstavlja velik napor za početnika. Zato ćemo karakteristike članova ove familije ilustrovati na primeru hipotetičkih procesora - nastavnih modela računara 886, 8286, 8486 i 8686 koji predstavljaju pojednostavljene verzije 80x86 čipova, ali omogućavaju da se ilustruju razne arhitektonske specifičnosti bez zalaženja u sve detalje ogromnog CISC skupa instrukcija. Ovo poglavlje koristi x86 hipotetički procesor za opis koncepata *kodiranja instrukcija, načina adresiranja, sekvencijalnog izvršavanja, bafera naredbi (prefetch queue), tekućih linija (pipelines) i superskalarnih operacija*.

Arhitektura procesora familije x86

CPU registri su specijalne memorijske lokacije konstruisane od flip-flop kola. Oni nisu deo memorije već su ugrađeni u CPU čip. Razni članovi 80x86 familije imaju registre različite veličine. NARx86 (nadalje x86) imaju tačno 4 registra svaki veličine 16-bitova. Sve aritmetičke i logičke 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 sledeću instrukciju i flag registar koji čuva rezultate poređenja (da li je neka vrednost manja, jednaka ili veća od druge vrednosti).

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

Aritmetičko - logički uređaj (ALU) je mesto gde se dešava većina procesorskih akcija. Npr. ako želite da dodate broj 5 na sadržaj registra AX, CPU vrši sledeće akcije:

kopira vrednost iz AX u ALU
šalje broj 5 u ALU
naređuje ALU da sabere dve poslate vrednosti
vraća 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 takođe odgovoran za pristup podacima u kešu.

Upravljački 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 korišćenjem logičkih kola za izvršavanje ovih instrukcija. Poželjno je da broj tih logičkih 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) obično su se programirali spolja (hard-wired). Povezivanjem

odgovarajućih komponenti definisalo se šta će program rešavati. Kada se prelazilo na rešavanje drugog zadatka, moralo se vršiti povezivanje na neki drugi način što je znatno otežavalo korišćenje računara. Pojava programibilnih kompjuterskih sistema sa kontrolnim panelom sa rupicama u koje su mogle da se utaknu žice i na taj način izvrši programiranje značila 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 poteškoća u ovakvom programiranju je što je broj mogućih instrukcija jako ograničen brojem rupica koje su fizički mogle da se smeste u svakom redu. Ovaj problem su projektanti relativno brzo prevazišli dodatnom logikom i omogućili su da se sa zadavanja n instrukcija pomoću n rupica pređe na zadavanje n instrukcija pomoću log(n) rupica. To su ostvarili dodeljivanjem numeričkog koda svakoj instrukciji, a zatim su dekodirali ovu instrukciju kao binarni broj korišćenjem u $\log_2 n$ rupica. Ovo je zahtevalo 8 dodatnih logičkih 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 obično dekodiraju izvorni i odredišni argument kao deo mašinske instrukcije, odgovarajuća rupica odgovara izvornom a druga 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 instrukcija koje se mogu zadati ograničena brojem redova na panelu. Džon Fon Nojman i drugi programeri tog vremena uočili su relaciju između 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 logičkim kolima *dekodera operacija* u CPU. Sve ovo podrazumeva još više logike u CPU. Upravljačka 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 uvećava IP tako da on ukazuje na sledeću instrukciju koju bi trebalo uzeti iz memorije.

Kada projektanti biraju skup instrukcija, obično uzimaju da je opkod neki umnožak 8 bitova, tako da CPU lako uzima kompletnu instrukciju iz memorije. Cilj CPU dizajnera je da dodele odgovarajući broj bitova za *klasu instrukcija* (MOV, ADD i sl.) i za *polja argumenata*. Više bitova u polju klase instrukcija omogućava da repertoar instrukcija bude veći, dok dodatni bitovi za argumente omogućavaju da se argumenti biraju iz većeg 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 neiskorišćene bitove, projektanti polje argumenata koriste za zadavanje opkodova novih instrukcija, što je opet povezano sa ugrađivanjem 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 početnike Intelova shema dekodiranja instrukcija prevelik zalogaj, mi smo hipotetičke procesore x86 snabdeli samo malom i znatno jednostavnijom shemom dekodiranja koja ipak može dati osećaj šta se u stvari dešava unutar procesora prilikom izvršavanja programa.

Skup instrukcija i načini adresiranja

Procesori x86 imaju 20 osnovnih instrukcija. Sedam instrukcija ima 2 argumenta, 8 jedan i 5 je bez argumenta. To su sledeće: 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 numerička konstanta.

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 određenje, jer oni koriste samo heksadekadni brojni sistem.

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

Aritmetičke i logičke instrukcije

mogu se pojaviti u sledećim formatima:

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

Dejstvo instrukcija je sledeće:

ADD d,s	d:= d + s	AND d,s	d:= d and s (bit po bit)
SUB d,s	d:= d - s	OR d,s	d:= d or s (bit po bit)
CMP d,s	Poredi d sa s i pamti rezultat poređenja	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 neoznačeni. Procesori x86 imaju sledeće instrukcije za predaju upravljanja:

JA dest	skok ako je prvi veći (Above)
JAE dest	skok ako je prvi veći 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 različiti (Not Equal)
JMP dest	bezuslovni skok
IRET	povratak iz interapta (Interrupt RETurn)

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

Ostale instrukcije

Instrukcije **GET** i **PUT** omogućavaju učitavanje i ispisivanje celobrojnih heksadekadnih vrednosti. Još dve instrukcije **HALT** i **BRK** su bez argumenta. HALT prekida izvršavanje programa dok BRK privremeno zaustavlja izvršavanje programa dajući mogućnost korisniku da ga nastavi.

Načini adresiranja

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

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 praktično 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. Treća 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 naći 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 načini adresiranja

Hipotetički procesori x86 mogu da koriste tri načina 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 čija 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 kraće napisati MOV AX, [1000]. Medutim, ovaj kraći način nije i bolji jer omogućava upis samo jedne vrednosti, a prethodni samo promenom sadržaja registra BX daje mogućnost da istom instrukcijom MOV AX, [BX] pristupamo različitim memorijskim lokacijama.

Poslednja instrukcija koristi *indeksni* (relativni) način adresiranja. Ovaj način adresiranja zgodan je za pristupanje različitim 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 čija se adresa nalazi u BX.

Kodiranje i izvršavanje instrukcija

Mada ne možemo proizvoljno da dodeljujemo opkodove instrukcijama x86, imajte na umu da CPU koristi logička kola za dekodiranje opkodova i proizvodi upravljačke akcije u skladu s njima. Tipičan CPU opkod koristi odreden broj bitova u opkodu za zadavanje klase instrukcije i odgovarajući broj bitova za dekodiranje svakog od argumenata. Neki sistemi (CISC) dekodiraju ova polja na vrlo kompleksan način što rezultuje kompaktnim instrukcijama. Drugi sistemi (RISC) dekodiraju opkodove na vrlo jednostavan način, što podrazumeva da se često neki bitovi uopšte i ne koriste (npr. kad je instrukcija bez argumenata bitovi za zadavanje argumenata) i time je ograničen (redukovan) broj instrukcija koje se mogu koristiti. Familija Intel 80x86 je pravi predstavnik CISC koncepcije i ima

jedan od najkompleksijnih sistema za dekodiranje. Budući da je namena hipotetičke familije x86 jedino da prezentira koncept dekodiranja instrukcija bez zalaženja u specificnosti karakteristične za realnu familiju 80x86 mi ćemo sada na pojednostavljen način prikazati CISC dekodiranje.

Instrukcije sa dva argumenta

Tipična 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 koristićemo trikove (što Intelovi projektanti obožavaju). Kao što sa slike možete videti, dve od mogućih 8 vrednosti ovog polja iskorišćene su za naredbu MOV, 5 za instrukcije obrade (aritmetičke i logičke) sa dva argumenta, a vrednost 000 rezervisana je za specijalne instrukcije. Specijalne instrukcije imaju mehanizam koji omogućava da se polje za nedostajući argument koristi za zadavanje o kojoj se od specijalnih instrukcija radi.

i	i	i	r	r	m	m	m

OPKOD

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

iii	**rr**	**mmm** indirektno ili indeksno adresiranje
000 = spe cial	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 9.1 - Kodiranje osnovnih instrukcija procesora x86

Da bi se odredio pojedinačni opkod, treba da selektujete odgovarajuće 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 sadržajem memorijske lokacije 1000. Njen opkod biće 11000110 odnosno 0C6h. I da smo kodirali instrukciju MOV AX, [2000] njen prvi bajt takođe bi bio 0C6h, jer u njemu ne zadajemo konkretnu memorijsku lokaciju već samo način adresiranja memorijskog argumenta. Sledeća dva bajta definišu o kojoj lokaciji se radi. 16-bitna adresna konstanta koja sledi neposredno za bajtom sa opkodom u mlađem bajtu (prvom koji ide iza opkoda) imaće vrednost 00h, a u starijem 10h (ili 20h za zadavanje adrese 2000).

Instrukcije sa jednim argumentom

Specijalni opkodovi omogućavaju da se proširi skup raspoloživih instrukcija. Ovaj opkod sa tri nule na početku 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 obično ostavljaju mogućnost da proširе reporoar instrukcija na novim procesorima, recimo Intelovi projektanti su ovo maksimalno iskoristili kada su sa 80286 prelazili na 80386).

i	i	i	r	r	m	m	m

0	0	0	i	i	m	m	m

OPKOD

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

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

Slika 9.2 - Kodiranje instrukcija sa jednim argumentom

Instrukcije bez argumenata

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

0	0	0	0	0	i	i	i

OPKOD

iii

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

Slika 9.3 - Kodiranje instrukcija bez argumenata

Instrukcije skoka

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

Jcc adresa

JMP upisuje 16-bitnu vrednost koja sledi neposredno iza opkoda u IP - to je adresa instrukcije na koju će 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 sledeće instrukcije na koju već 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 odgovarajući način postavi flegove. Napominjemo da se može zadati 8 opkodova (jer odgovarajuće 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.

i	i	i	r	r	m	m	m

0	0	0	0	1	i	i	i

OPKOD

iii

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

000 = JE 100 = JA

001 = JNE 101 = JAE

010 = JB 110 = JMP

Slika 9.4 - Kodiranje instrukcija skoka

Izvršavanje instrukcija korak po korak

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

K1 - Donošenje bajta sa opkodom instrukcije iz memorije;
K2 - Uvećanje IP da pokazuje na sledeći bajt;
K3 - dekodiranje instrukcije;
K4 - Ako je potrebno, donošenje 16-bitnog izvornog argumenta iz memorije;
K5 - Ako je potrebno, uvećanje IP da pokazuje na bajt iza argumenta;
K6 - Ako je potrebno, izračunavanje 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 pomoći 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 čita bajt sa te adrese. To traje jedan takt (radi jednostavnijeg objašnjenja smatraćemo da memorijski ciklus traje jedan takt.) Posle uzimanja opkoda, CPU uvećava IP tako da pokazuje na sledeću memorijsku lokaciju. Ako tekuća instrukcija zauzima više bajtova, IP sada ukazuje na argument instrukcije. Ako tekuća instrukcija zauzima samo 1 bajt, IP će pokazivati na opkod sledeće instrukcije. Ovo se izvršava u sledećem 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 utvrđuje 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, trajaće 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 uvećati IP za 2 jer sve instrukcije x86 rade sa dvobajtnim argumentima. Ova operacija takođe traje 0 ili 1 takt u zavisnosti ima li ili nema argumenta.

Zatim CPU izračunava adresu memorijskog argumenta. Ovaj korak se izvršava jedino ako mmm polje instrukcije sadrži vrednost 101 ili 100. Ako sadrži 100, CPU izračunava 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 izračunavanje 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 traje 0 ciklusa, jer je konstanta u prethodnom koraku već doneta iz memorije. Ako je argument u registru (mmm=000, 001, 010 ili 011), onda ovaj korak traje jedan takt. ako se radi o reči koja je poravnata na parnu adresu potrebna su dva, ako je reč 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 instrukcije3 traje izmedu 5 i 11 taktova, zavisno od tipa argumenta i njegovog poravnanja u memoriji.

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

K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)

K2 - Uvećanje IP da pokazuje na sledeći 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, uvećanje IP da pokazuje na bajt iza argumenta (0-1 takt);

K6 -Ako je potrebno, izračunavanje 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 čita 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 sledećim koracima:

K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)

K2 - Uvećanje IP da pokazuje na sledeći 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, uvećanje IP da pokazuje na bajt iza argumenta (0-1 takt);

K6 - Ako je potrebno, izračunavanje adrese u memoriji za odredišni argument (0-2 ciklusa)

K7 - Donošenje izvornog argumenta i upućivanje u ALU (0-3 takta);

K8 - Naređivanje ALU da izvrši zadatu operaciju (sabere, oduzme, uporedi izvrši logičko 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 slična je prethodnim, ali se izvršava brže zato što ima samo jedan argument.

K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)

K2 - Uvećanje IP da pokazuje na sledeći 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, uvećanje IP da pokazuje na bajt iza argumenta (0-1 takt);

K6 - Ako je potrebno, izračunavanje 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 sledeći način:

K1 - Donošenje bajta sa opkodom instrukcije iz memorije (1 takt)

K2 - Uvećanje IP da pokazuje na sledeći bajt (1 takt);

K3 - Dekodiranje instrukcije (1 takt);

K4 - Donošenje 16-bitne odredišne adrese iz memorije (1-2 takta);

K5 - Uvećanje IP da pokazuje na bajt iza argumenta (1 takt);

K6 - Testiranje "manje od" i "jednako" flega (1 takt)

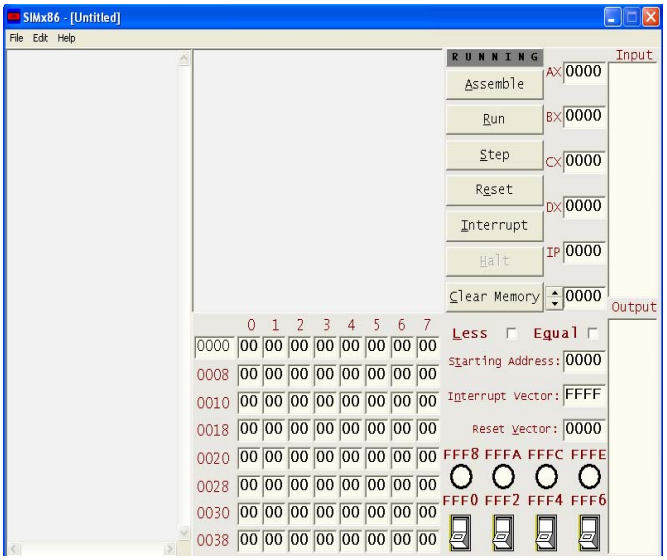
K7 - Ako je uslov ispunjen, CPU kopira donetu 16-bitnu konstantu u IP (0 ciklusa ako nema grananja, 1 ako treba predati upravljanje)

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

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

Emulator procesora x86

Program SIMx86.exe koji ćemo koristiti za upoznavanje sa programiranjem na simboličkom nivou predstavlja modifikaciju istoimenog programa prof. R. Hajda koju je uradio bivši učenik Matematičke gimnazije Darko Pešikan.



Ovaj program sadrži ugrađen editor, asembler, dibager i interrupter za x86 hipotetičke procesore. Uz njegovu pomoć ilustovaćemo kako se pišu elementarni x86 simbolički programi, kako se prevode - asembliraju, kako se pregleda i menja sadržaj memorijskih lokacija i kako se korak po korak izvršavaju mašinski programi. Za razliku od emulatora za uIAS računar koji nije prihvatao simboličke adrese (ni promenljive ni labele) ovde možemo da koristimo jednoslovne sim-boličke adrese. Ilustovaćemo kako se koristi memorijski mapirani ulaz/izlaz, DMA i pojednostavljeni sistem prekida. Pokaza-ćemo takođe kako se vrši programska modifikacija i kako bi se mogle uvesti nove instrukcije u repertoar naredbi.

Primer 1 - Jednostavan program bez I/Opodataka
Sledeći program EX1.x86 izračunava vrednost izraza AX:=3*(a+b-1) gde su a i b simbolička imena za memorijske reči sa adresa 1000h i 1002h.

```
mov ax, [1000]          ; upisuje podatak sa memorijske lokacije 1000h u registar AX
mov bx, [1002]
add ax, bx              ; AX:=AX+BX
sub ax, 1               ; AX:=AX-1
mov bx, ax
add bx, ax
add ax, bx              ; AX:=3*AX
halt                    ; zaustavlja izvrsavanje programa
```

Primer 2 - Jednostavan program sa I/O instrukcijama get i put
Sledeći program EX2.x86 učitava niz celih brojeva koje upisuje se na uzastopne memorijske lokacije počev od adrese 1000h, a zatim izračunava i prikazuje njihov zbir. Unos vrednosti 0 znači da nema više ulaznih podataka.

```
        mov bx, 1000
a:      get
        mov [bx], ax
        add bx, 2
        cmp ax, 0
        jne a
        mov cx, bx
        mov bx, 1000
        mov ax, 0
b:      add ax, [bx]
        add bx, 2
        cmp bx, cx
        jb b
        put
        halt
```

Primer 3 - Program koji ilustruje memorijski mapiran I/O
Sledeći program EX3.x86 prihvata ulazne podatke - vrednosti dve logičke promenljive sa prekidača povezanih sa memorijskim lokacijama 0FFF0h i 0FFF2h i za njih izračunava vrednosti logičkih funkcija AND, OR, XOR i EQU (ekvivalencije) koje prikazuje preko LED dioda povezanih sa memorijskim lokacijama 0FFF8h, 0FFFAh, 0FFFCh i 0FFFEh. U trenutku kada se uključi treći prekidač, povezan sa lokacijom 0FFF4h, program prekida sa radom.

```
a:      mov ax, [fff0]
        mov bx, [fff2]
        mov cx, ax          ; Izracunavanje Sw0 AND Sw1
        and cx, bx
        mov [fff8], cx
        mov cx, ax          ; Izracunavanje Sw0 OR Sw1
        or cx, bx
        mov [fffa], cx
        mov cx, ax          ; Izracunavanje Sw0 XOR Sw1
        mov dx, bx          ; A XOR B = AB' + A'B
        not cx
        not bx
        and cx, bx
        and dx, ax
        or cx, dx
```

```
mov [fffc], cx
not cx          ; Izracunavanje Sw0 EQU Sw1
mov [fffe], cx  ; EQU = NOT XOR
mov ax, [fff4]  ; Cita stanje treceg prekidaca
cmp ax, 0               ; Proverava da li je ukljucen
je a                    ; Ponavlja izracunavanja ako je iskljucen
halt
```

Primer 4 - Program koji ilustruje DMA

Sledeći program EX4.x86 startuje od labele d upisivanjem 0 na lokaciju 1000. Zatim ulazi u petlju u kojoj proverava dva uslova - da li je uključen prekidač FFF0 i da li je korisnik promenio vrednost na lokaciji 1000. Uključivanje prekidača prekida izvršavanje programa, a promena vrednosti na lokaciji 1000 predaje upravljanje na na sekciju sa labelom c u kojoj se sabira vrednost n reči gde je n nova vrednost sa lokacije 1000. Program sabira vrednosti sa uzastopnih lokacija počev od adrese 1002, prikazuje zbir i vraća upravljanje na labelu d.

```
d:  mov cx, 0           ;Brise lokaciju 1000h pre no sto je testiramo
    mov [1000], cx
                        ;U sledećoj petlji proveravamo da li je promenjen sadrzaj
                        ; sa lokacije 1000h i da li je ukljucen prekidac FFF0
a:  mov cx, [1000]
    cmp cx, 0
    jne c               ; Skok ako je promenjena vrednost na lokaciji 1000h
    mov ax, [fff0]      ; Ako nije citamo stanje prekidaca FFF0
    cmp ax, 0
    je a                ; Skok ako je iskljucen
    halt                ; Ako je ukljucen, prekida se izvrsavanje programa
                         ; Sledeći fragmaent sabira "cx" uzastopnih reci od kojih je prva
                        ; na adresi 1002h, a po zavrsenom sabiranju prikazuje zbir.
C:   mov bx, 1002       ;Inicijalizuje BX da pokazuje na pocetak niza
    mov ax, 0           ;Inicijalizuje zbir
b:  add ax, [bx]        ;Dodaje sledeći podatak.
    add bx, 2           ;pomera pokazivac BX na sledeći elemnt niza
    sub cx, 1           ;Smanjuje brojac
    cmp cx, 0
    jne b
    put                 ;Prikazuje zbir i vraca se na pocetak
    jmp d
```

Primer 5 - Program koji ilustruje korišćenje sistema prekida

Sledeći program sastoji se iz dva modula - EX5a.x86 i EX5b.x86. Glavni program EX5a.x86 poredi memorijske lokacije 1000h i 1002h. Ako njihovi sadržaji nisu jednaki glavni program prikazuje vrednost sa adrese 1000h, kopira tu vrednost na lokaciju 1002h i ponavlja ceo proces sve dok se ne uključi prekidač FFF0.

```
a:      mov ax, [1000]      ;Uzima podatak sa lokacije1000h
        cmp ax, [1002]      ;Poredi fga sa podatkom na lokaciji 1002h
        je b                ;Ako su jednaki proverava stanje prekidaca FFF0h.
        put                 ;Ako su razliciti prikazuje vrednost sa lokacije 1000h
                            (AX)
        mov [1002], ax      ;upisuje prikazanu vrednost na 1002h
```

```
b:      mov ax, [fff0]      ;Testira prekidac FFF0
        cmp ax, 0
        je  a
        halt
```

Rutina za opsluživanje prekida (interrupt service routine - ISR EX5b.x86) nalazi se na lokaciji 100h, za razliku od glavnog programa koji se nalazi na adresi 0. Kada se primi signal za prekid, odgovarajuća ISR uvećava vrednost na adresi 1000h i vraća upravljanje glavnom programu.

```
mov ax, [1000]      ;Uvecava sadrzaj sa adrese lo ca tion 1000h za 1
add ax, 1
mov [1000], ax
iret                ;vraca upravljanje prekinutom programu
```

Primer 6 - Program koji ilustruje samomodifikaciju

Sledeći program EX6.x86

```
        sub ax, ax
        mov [100], ax       ;Postavlja 0 na lokaciju 100h. Promena ove vrednosti
a:      mov ax, [100]       ;biće  indikator da je izvrsen novogenerisan kod
        cmp ax, 0
        je b                ;Ako je [100h]=0, prelazi se na generisanje koda
        halt                ;U suprotnom se zavrsava sa radom programa
b:       mov ax, 00c6       ;Pocev od ove naredbe, upisuju se kodovi masinskih
        mov [100], ax        instrukcija u memoriju pocev od lokacije 100h
        mov ax, 0710
        mov [102], ax
        mov ax, a6a0
        mov [104], ax
        mov ax, 1000
        mov [106], ax
        mov ax, 8007
        mov [108], ax
        mov ax, 00e6
        mov [10a], ax
        mov ax, 0e10
        mov [10c], ax
        mov ax, 4
        mov [10e], ax
        jmp 100             ;predaje upravljanje generisanom kodu
```

Upisuje sledeći kod na lokaciju 100 i izvršava ga.

```
mov ax, [1000]      ;Generisan kod prikazuje broj sa lokacije 1000h
put add ax, ax
add ax, [1000]
put                 ; i njegovu trostruku vrednost
sub ax, ax
mov [1000], ax      ;a zatim upisuje 0 na lokaciju 1000h
jmp 0004            ;0004 je adresa labele a na koju se vraca upravljanje
```

Primer 7 - Samomodifikacija koja realizuje poziv procedure

Skup instrukcija hipotetičkih procesora ne sadrži naredbu za poziv procedure. Međutim programi EX7a.x86 i EX7b.x86 simuliraju poziv potprograma i povratak upravljanja glavnom programu korišćenjem samomodifikacije. Program prevodi neoznačen ceo dekadni broj u binarni brojni sistem.

Očekuje broj u AX, prevodi ga u niz nula i jedinica koje upisuje na uzastopne memorijske lokacije počev od adrese 1000h.

```
; Potprogram koji treba upisati u memoriju pocev od adrese 100h
        mov bx, 1000    ;pocetna adresa stringa
        mov cx, 10              ;brojac=16 (10h), tj. broj bitova u reci
a:      mov dx, 0               ;pretpostavljamo da je tekući bit 0
        cmp ax, 8000    ;proveravamo da li je H.O. bit 0 ili 1
        jb b
        mov dx, 1               ;ako je H.O. bit u AX jednak 1
b:      mov [bx], dx    ;upisujemo 0 ili 1 na tekuću memorijsku lokaciju.
        add bx, 1               ;pomeramo pointer BX na sledeću lokaciju
        add ax, ax              ;AX = AX *2 (shift left)
        sub cx, 1               ;
        cmp cx, 0               ;ponavljamo 16 puta
        ja a
        jmp 0           ;povratak upravljanja glavnom programu koji pre poziva potprograma
                        ;na adresni deo ove instrukcije upisuje adresu povratka.
```

Jedina instrukcija koju program modifikuje u potprogramu je poslednja naredba jmp. Ovaj skok treba da vrati kontrolu na instrukciju neposredno iza JMP kojim je predato upravljanje potprogramu. Da bi se to omogućilo glavni program treba da upamti adresu povratka na mestu argumenta u poslednjoj instrukciji sledećeg koda. Ako kod počinje na lokaciji 100h odgovarajuća JMP instrukcija biće na adresi 120h. Dakle, glavni program adresu povratka mora da upiše na adresu 121h.

```
;Glavni program koji treba upisati pocev od adrese 0
        mov ax, 000c            ;Adresa prve instrukcije BRK
        mov [121], ax           ;upis adrese povratka (prve instrukcije BRK) adr. deo JMP
        mov ax, 1234            ;Predaja broja 1234h koji će biti preveden u binarni
        jmp 100                 ;"Call" potprograma
        brk                     ;Pauza da korisnik proveri sadrzaj niza iz memorije
        mov ax, 0019            ;Adresa sledeće BRK instrukcije
        mov [121], ax
        mov ax, fdeb            ;Prevođenje broja 0FDEBh u binarni zapis.
        jmp 100
        brk
        mov ax, 26
        mov [121], ax
        mov ax, 2345            ;Prevođenje broja 2345h u binarni zapis.
        jmp 100
        halt
```

Ključni pojmovi

kodiranje instrukcija načini adresiranja registarsko adresiranje neposredno adresiranje direktno adresiranje	indirektno adresiranje indeksno memorijski mapiran UI direktan memorijski pristup sistem prekida	vektor prekida samomodifikacija programa poziv procedure povratak iz procedure prenos argumenta

Pitanja i zadaci za vežbu

1. Kodirajte sledeće instrukcije:

```
AND CX,BX
ADD AX,14
NOT [BX]
NOT [1000+BX]
```

2. Ako labeli a odgovara adresa 0003, kodirajte sledeće instrukcije

JB a
JMP a

3. Napisati program za SIMx86 koji izračunava vrednost izraza: a) AX:= 2*(2a+3b-c) b) BX:=2a-b+3c gde su a, b i c simbolička imena za memorijske reči koje se nalaze na adresama 1000h, 1002h i 1004h.

4. Napisati program za SIMx86 koji za ulazne podatke x1, x2 i x3 izračunava i prikazuje vrednost izraza:

y = max(x1, x2, x3)

5. Napisati program za SIMx86 koji prihvata vrednosti dve logičke promenljive sa prekidača povezanih sa memorijskim lokacijama 0FFF0h i 0FFF2h i za njih izračunava vrednosti logičkih funkcija $\Rightarrow$ (implikacija), $\Leftarrow$, NAND i NOR koje prikazuje preko LED dioda povezanih sa memorijskim lokacijama 0FFF8h, 0FFFAh, 0FFFCh i 0FFFEh. U trenutku kada se uključi treći prekidač, povezan sa lokacijom 0FFF4h program prekida sa radom.

6. Napisati program za SIMx86 koji prihvata n - broj članova niza i upisuje ga na adresu 1000, a zatim i n članova niza koje upisuje na adrese počev od 1002 (n <= 20) i na kraju

a) prebrojava koliko je među njima manjih od 10 i prikazuje taj rezultat;

b) prebrojava koliko je među njima parnih i prikazuje taj rezultat Uputstvo: AND AX,1 daće kao rezultat 0 ako je broj paran, jer se završava sa 0b;

c) izračunava njihov zbir i prikazuje ga,

7. Skup instrukcija x86 ne sadrži naredbu za množenje. Napišite program MUL.x86 koji učitava dve vrednosti i izračunava i prikazuje njihov proizvod. Uputstvo - množenje se može predstaviti kao ponovljeno sabiranje.

8. Napisati proceduru za SIMx86 koja nalazi i štampa maksimum niza reči koje su upisane u memoriju počev od lokacije 1000h. Broj elemenata niza nalazi se u registru CX. Glavni program treba da poziva ovu proceduru više puta sa različitim vrednostima CX. Koristiti programsku samomodifikaciju za realizaciju poziva procedure i povratka u glavni program.