VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Viktor Kiško

Kompiuterių architektūra

Praktinių užsiėmimų konspektas

Vilnius
2005 m.

Turinys

TURINYS **2**

ĮVADAS **3**

1. KOMPIUTERIŲ ARCHITEKTŪROS SĄVOKA **4**

1.1. KOMPIUTERIŲ ISTORIJA 4
1.2. FON NEIMANO PRINCIPAI 6
1.3. FON NEIMANO PROCESORIAUS STRUKTŪRA IR VEIKIMAS 7
1.4. ARCHITEKTŪRŲ APŽVALGA 9

2. DUOMENŲ FORMATAI **22**

2.1. FIKSUOTO KABLELIO FORMATAS 22
2.2. SIMBOLINIS FORMATAS 28
2.3. DEŠIMTAINIAI SKAIČIAI 29
2.4. MATEMATINIO PROCESORIAUS DUOMENŲ FORMATAI 32

3. I8088 ARCHITEKTŪRA **35**

3.1. VIDINĖ MIKROPROCESORIAUS STRUKTŪRA 35
3.2. ATMINTIES SEGMENTACIJA 39
3.3. KOMANDOS STRUKTŪRĄ 42
3.4. ADRESAVIMO BŪDAI 50
3.5. STEKAS 51
3.6. PERTRAUKIMŲ SISTEMA 56

4. KOMANDŲ SISTEMA **64**

4.1. BENDROSIOS KOMANDOS 64
4.2. ARITMETINĖS KOMANDOS 68
4.3. DEŠIMTAINIAI SKAIČIAI 70
4.4. KONVERTAVIMAS 72
4.5. LOGINĖS KOMANDOS 73
4.6. KOMANDOS DARBUI SU EILUTĖMIS 75
4.7. VALDYMO PERDAVIMO KOMANDOS 78
4.8. SĄLYGINIS VALDYMO PERDAVIMAS 80
4.9. CIKLŲ KOMANDOS 83
4.10. PERTRAUKIMŲ KOMANDOS 84
4.11. PROCESORIAUS BŪSENOS VALDYMAS 85

5. PROGRAMAVIMO SISTEMA **86**

5.1. ASEMBLERIO KALBA 86
5.2. PROGRAMŲ TRANSLIAVIMAS 92
5.3. PROGRAMŲ DERINIMAS 95

6. MIKROPROGRAMINIS LYGIS **98**

6.1. FIZINIO LYGIO KOMPONENTĖS 98
6.2. INTERPRETUOJAMAS LYGIS 101
6.3. KOMANDŲ REALIZACIJA 102
6.4. KOMANDOS TEST IR GATE 103
6.5. MIKROPROGRAMAVIMO KALBA 104

KONTROLINIAI DARBAI **107**

ATSAKYMAI IR SPRENDIMAI **112**

LITERATŪRA **118**

Įvadas

Šis konspektas skirtas palengvinti asistento darbą praktinių užsiėmimų metu. Taip pat juo turėtu būti naudinga naudotis ir studentams.

Iki šiol itin daug dėmesio buvo skiriama asemblerio kalbai ir programavimo ypatumams MS-DOS operacinėje sistemoje. Pagrindinis šio konspekto tikslas – padėti studentams per praktinius užsiėmimus suformuoti bendrą vaizdą apie kompiuterių architektūrą ir išvengti situacijos, kai "už medžių nematyti miško".

Praktinių užsiėmimų pradžioje bandoma pateikti medžiagą bendrai, neprisirišant prie konkrečių architektūrų. Vėlesnėse pratybose pateikiami pavyzdžiai, kaip viena ar kita architektūros savybė realizuota Intel 8086 architektūroje.

Šiame kurse kompiuterio architektūra išdėstoma kartu su asemblerio kalba. Tai sąlyguota tuo, kad kompiuteris yra algoritmo, užrašyto mašinos kalba, vykdytojas. Todėl žemesnio lygio programavimo kalba reikalinga kompiuterio architektūros supratimui. Reikia pabrėžti, kad asemblerio kalba yra tik pagalbinė priemonė.

Šio konspekto pagrindas yra dr.doc.A.Mitašiūno mokymo priemonė "Kompiuterių architektūrą".

1. Kompiuterių architektūros sąvoka

1.1. Kompiuterių istorija

Kompiuterių istorija prasideda 1946 m., kai John Mauchley ir Presper Eckert sukūrė elektroninį kompiuterį ENIAC (*Electronic Numerical Integrator And Computer*). Tuo metu pagrindinis kompiuterio elementas buvo vakuuminė lempa. Kompiuteris ENIAC buvo sukonstruotas iš 18 000 lempų, 1500 relė ir svėrė apie 30 tonų. Programavimas buvo vykdomas jungiklių pagalba (kompiuteryje buvo apie 6000 jungiklių). Kompiuteryje buvo naudojama dešimtainė skaičiavimo sistema. Taip prasidėjo elektroninių skaitmeninių kompiuterių istorija.

Kartu su Mauchley ir Eckert dirbo žymus matematikas John von Neumann. Po kompiuterio ENIAC sukūrimo, John von Neumann pradėjo kurti kompiuterį IAS (*Immediate Address Storage*). John von Neumann iš esmės pakeitė kompiuterių organizavimo principus – jis pasiūlė saugoti programą skaitmeniniu pavidalu atmintyje kartu su duomenimis. Taip pat buvo siūloma atsisakyti dešimtainės sistemos ir pereiti prie dvejetainės. Pirmas kompiuteris, kuris realizavo von Neumann principus buvo EDSAC (1949). John von Neumann architektūra turėjo 5 elementus – atmintį, aritmetinį loginį įrenginį, valdymo įrenginį, įvedimo ir išvedimo įrenginius.

Antra kompiuterių karta pasirodė, kai buvo išrastas tranzistorius. Pirmas kompiuterius pagamintas tranzistorių pagrindu buvo TX-0. TX-0 buvo naudojamas kompiuterio TX-2 testavimui. Po kelerių metų kompanijoje DEC buvo sukurtas kompiuteris PDP-1, TX-0 ir TX-2 analogas. Nuo PDP kompiuterių prasidėjo serijinė elektroninių skaitmeninių kompiuterių gamyba. Korporacija IBM išleido tranzistorinius kompiuterius – IBM 7090 ir IBM 7094.

Trečia kompiuterių karta pasirodė maždaug nuo 1965m. Trečios kartos kompiuteriuose buvo naudojamos integralinės schemos, jos leido įdėti dešimtis tranzistorių į vieną mikroschemą. Korporacija IBM išleido kompiuterių seriją – System/360. System/360 yra kompiuteriai su vienoda asemblerio kalba, bet skirtingomis galimybėmis. Programos, parašytos vienam serijos kompiuteriui teoriškai turėjo be jokių apribojimų veikti ir kituose serijos kompiuteriuose.

Ketvirta kompiuterių karta pasirodė XX amžiaus 80-ais metais. Tai buvo personalinių kompiuterių karta. Pagrindiniai tos kartos kompiuteriai – IBM PC ir APPLE. Plačiau apie kompiuterių istoriją galima paskaityti [Tan98].

Pirmieji kompiuteriai buvo labai brangūs ir nestabilūs. Aparatūrinė įranga buvo žymiai brangesnė už programinę. Todėl buvo ieškoma būdų, kaip programinės įrangos sąskaita sumažinti kompiuterių kainą. Tam, kad sumažinti kompiuterio kainą, reikėjo supaprastinti komandų sistemą. Bet komandų sistemos supaprastinimas reiškė sudėtingumus programuotojams – reikėjo programuoti algoritmus, turint mažai operacijų. Kaip išeitis buvo sugalvota naudoti komandų interpretatorių mikroarchitektūriniame lygmenyje. T.y. kompiuterio komandų sistema buvo praplečiama. Bet komandos buvo interpretuojamos vidine programa. Interpretacijos dėka greitai atsirado daug labai sudėtingų komandų. Komandos turėjo skirtingus formatus. Taip buvo bandoma priartinti kompiuterių galimybes prie esamų reikalavimų iš vartotojų.

Bet kai kurie specialistai pastebėjo, kad reikia ieškoti kitų būdų kompiuterių galimybių gerinimui. 1980 m. Berkli universitete mokslinė grupė pradėjo kurti VLSI procesorius, kurie neturėjo interpretavimo programos. Šitai sąvokai jie sugalvojo terminą RISC (Reduced Instruction Set Computer) ir pavadino naują procesorių RISC I. Vėliau buvo išleista mikroschema MIPS, veikiantį analogiškais principais. Pavadinimas RISC atsirado dėl to, kad tuo metų komerciniai procesoriai turėjo labai daug instrukcijų – apie 200-300, o nauji RISC procesoriai vos ~50. Atitinkamai, procesoriai su interpretavimo programa buvo klasifikuojami kaip CISC (Complex Instruction Set Computer).

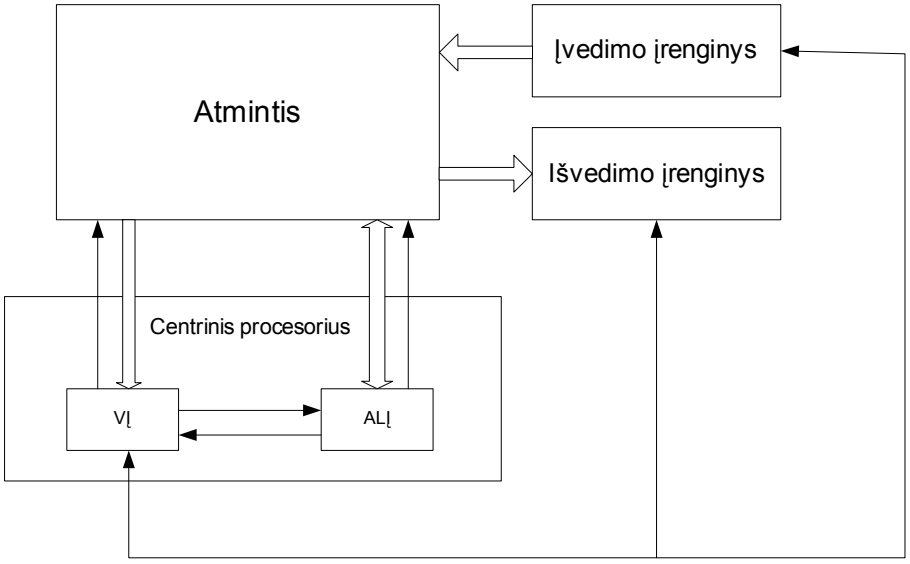
Pagrindinė RISC kūrėjų idėja – pasinaudoti taisykle 80/20, t.y. 80% visų procesoriaus vykdomų instrukcijų yra iš aibės, kurioje yra tik 20% instrukcijų. Tam reikėjo sukurti procesoriaus komandų sistemą, kurioje būtų pakankamai mažai komandų, bet kurios būtų vykdomos per vieną kompiuterio ciklą [Mit03, p.8]. T.y. aritmetinis loginis įrenginys paima du registrus, įvykdo operacija su jais ir padeda rezultatą į registrą. Kitos komandos buvo realizuotos, panaudojant mažą aibę labai greitų komandų. Dar vienas RISC procesorių skirtumas nuo CISC – didelis registrų skaičius (~32-128 RISC ir 8-16 CISC).

Išvardinkime principus, kurie būdingi RISC architektūroms. **Visos komandos vykdomos aparatūriškai –** visos komandos vykdomos aparatūrine įranga. RISC architektūrose nėra interpretavimo mechanizmo. CISC architektūrose komandos interpretuojamos mikroarchitektūriniame lygmenyje, t.y. komandos skaidomos į dalis, kurios vykdomos, kaip mikroprogramų seka. **Komandos tūri būti vykdomos per**

fiksuotą taktų skaičių (geriausia, per vieną procesoriaus taktą). Komandos turi būti lengvai dekoduojamos valdymo įrenginiu. Kuo greitesnis yra komandų dekodavimas, tuo daugiau komandų galima įvykdyti per tam tikrą laiko tarpą. Dekodavimo metu nustatoma kokie resursai reikalingi komandai įvykdyti (registrai, kanalai ir t.t.) ir kokius veiksmus reikia atlikti. Komandų dekodavimui supaprastinti naudojami skirtingi mechanizmai, pvz., komandos yra fiksuoto ilgio su nedideliu kiekiu operandu, bandoma sumažinti komandų formatų skaičių. **Į atmintį gali kreiptis tik specialios komandos**. Vienas iš komandų suskirstymo į atskirus žingsniu būdų – pareikalauti, kad daugelio komandų operandai būtų registruose. Pagrindinė priežastis – skirtumas tarp procesoriaus ir pagrindinės atminties veikimo greičių. **Procesoriuje turi būti daug registrų**. Skaitymas iš atminties yra lėtas procesas, todėl reikia turėti pakankamai registrų, tam kad saugoti reikalingus operandus procesoriuje (ir tuo sumažinti skaitymo iš atminties operacijų skaičių) [Tan98, pp.64-65].

1.2. Fon Neimano principai

1946 m. Džon fon Neiman (John von Neumann) paruošė ataskaitą, kurioje aprašė abstraktaus kompiuterio, dabar vadinamo fon Neimano mašina, veikimo principus. Tai buvo pirmas darbas apie skaitmeninius elektroninius kompiuterius. Ataskaita padarė didelę įtaką kompiuterių pramonei. Pasiūlytus principus galima rasti beveik visuose kompiuteriuose. Fon Neimano mašina susideda iš centrinio procesoriaus, atminties ir įvedimo/išvedimo įrenginių (1.2.1 pav.).



1.2.1 pav. Fon Neimano mašinos schema

Centrinis procesorius susideda iš valdymo įrenginio ir aritmetinio loginio įrenginio.

Fon Neimano mašinoje atmintis yra tiesinė homogeninė ląstelių seka. Mašinos įrenginiai gali nuskaityti ir įrašyti duomenis. Duomenų nuskaitymo iš bet kurios atminties ląstelės laikas yra vienodas visoms ląstelėms. Įrašymo į ląstelę greitis taip yra vienodas visoms ląstelėms. Tai yra **homogeninės atminties principas**. Visoms ląstelėms priskiriami skaičiai nuo 0 iki tam tikro N (N > 0). Toks skaičiaus vadinamas **ląstelės adresu**. Kiekviena ląstelė susideda iš mažesnių dalių, vadinamų skiltimis. Kiekviena skiltis saugo skaitmenį tam tikroje skaičiavimo sistemoje. Ląstelės turinys vadinamas **mašinos žodžiu**. Mašinos žodis yra minimalus duomenų kiekis kuriuo gali apsikeisti fon Neimano mašinos mazgai (žr. [Bau03]).

Mašinos žodis yra arba procesoriaus komanda, arba skaičius, skirtas apdorojimui. Tai yra **duomenų ir komandų neskiriamumo principas** – skaičiai ir komandos atmintyje užrašomi mašinos žodžių rinkiniais. Atmintyje **saugomos programos principas** reiškia, kad programa saugoma atmintyje kartu su skaičiais ir gali keistis vykdymo metu.

Centrinio procesoriaus valdymo įrenginys valdo visus kompiuterio įrenginius. Valdymo įrenginys siunčia valdymo signalus kitiems kompiuterio įrenginiams, pagal kuriuos jie vykdo tam tikrus veiksmus. Kiti įrenginiai gali valdyti tik atmintį, siunčiant signalus duomenims nuskaityti arba įrašyti. **Automatinio darbo (programinio valdymo) principas** reiškia, kad kompiuteris, vykdomas programą atmintyje, veikia be vartotojo įsikišimo (jeigu tik pati programa nereikalauja dialogo su vartotoju). Programa – tai atmintyje įrašyta mašininių komandų seka, kuri nusako programos algoritmą. Valdymo įrenginys pagal tam tikrą formulę nuskaito komandą iš atminties, įvykdo ją ir pereina prie kitos komandos. Komandos atmintyje nebūtinai turi išsidėstyti viena po kitos. **Tai yra nuoseklaus komandų vykdymo principas.** Čia reikia pabrėžti, kad kompiuteris architektūriniame lygmenyje nieko kito nedaro, išskyrus komandų vykdymą viena po kitos [Mit03, p.3].

1.3. Fon Neimano procesoriaus struktūra ir veikimas

Fon Neimano procesorius susideda iš aritmetinio loginio įrenginio ir valdymo įrenginio. Valdymo įrenginys valdo kitus įrenginius, o aritmetinis loginis įrenginys vykdo ”tikrus” skaičiavimus. Tarp ALĮ vykdomų funkcijų yra – nuskaityti tam tikros

atminties ląstelės turinį, įrašyti į atminties ląstelę tam tikrą reikšmę, įvykdyti tam tikrą operaciją su nuskaitytom reikšmėmis.

ALĮ susideda iš registrų – atminties ląstelių, kurios yra ne pagrindinėje atmintyje, o kituose kompiuterio įrenginiuose. Kai aritmetinis loginis įrenginys nuskaito atminties ląstelės turinį, reikšmė patalpinama į vieną iš registrų. Kai reikia įrašyti reikšmę į atminties ląstelę – ji paimama iš registro. Aritmetinio loginio įrenginio registrus žymėsime R1 ir R2. Dar išskirsime specialų registrą akumuliatorių S. Registras akumuliatorius saugo operacijos su registrais R1 ir R2 rezultatą (registrų gali būti ir daugiau, bet mūsų pavyzdžiuose užteks čia pateiktų).

Valdymo įrenginys taip pat turi registrus. Registras RK – komandų registras, saugo einamąją vykdomą komandą. Registras RA – komandų skaitliukas, saugo sekančios vykdomos komandos adresą atmintyje.

Dabar galime schematiškai parodyti kaip fon Neimano architektūroje vykdoma programa. Tarkime programa apskaičiuoja dviejų reikšmių, esančių adresuose x ir y, sumą ir įrašo ją adresų z, t.y. [z]:=[x]+[y]. Užrašas [A] nusako reikšmę, kurios adresas atmintyje yra A. Valdymo įrenginys įvykdys tokias komandas:

RK := [RA];
RA := RA + 1;
”Įvykdyti gautą komandą”;
”Pereiti prie pirmos komandos”;

Pakomentuokime šitą seką: pirmoje eilutėje valdymo įrenginys nuskaito komanda iš adreso RA, antroje koreguojama komandų skaitliuko reikšmę, trečioje vykdoma gautą komandą. Paskutinį veiksmą reiktų detalizuoti – tam, kad įvykdyti gautą komandą, reikia atlikti kelis veiksmus. Valdymo įrenginys dekoduoja komandą ir perduoda signalą ALĮ. Aritmetinis loginis įrenginys gauna dekoduotą komandą ir įvykdo veiksmus, kurie atitinka gautą komandą. Mūsų atveju tai yra sudėties komanda. ALĮ vykdomų veiksmų seka atrodys taip:

R1 := MEM[x];
R2 := MEM[y];
S := R1 + R2;
MEM[z] := S;

Pirmoje eilutėje ALĮ į registrą R1 nuskaitys reikšmę iš atminties adresu x, antroje į registrą R2 bus įrašyta reikšmė iš atminties adresu y, trečioje bus apskaičiuota

nuskaitytų reikšmių suma ir patalpinta į registrą akumuliatorių S. Paskutinėje eilutėje rezultatas patalpinamas į atminties ląstelę adresų z. Toliau vėl pradeda dirbti valdymo įrenginys. Veiksmas paskutinėje valdymo įrenginio veikimo algoritmo eilutėje ”Pereiti prie kitos komandos” reiškia sekančios komandų vykdymo ciklo iteracijos pradžia. Šitas ciklas bus vykdomas tol, kol nebus aptikta speciali sustojimo komanda, arba neįvyks neeilinis įvykis (pvz., elektros dingimas).

Algoritme sudėties komanda buvo pažymėta mums įprastu būdu – simbolių ”+”. Bet procesoriui toks užrašymo būdas nėra patogus. Procesorius turi aibę komandų, kurias gali vykdyti. Kiekviena komanda turi savo kodą, pvz. sudėties – 0, atimties – 1 ir t.t. Procesoriaus komandos koduojamos skaičiais (mašininiais kodais) ir vien pagal atminties ląstelės turinį neįmanoma nustatyti ar tai komanda, ar duomuo (komandų ir duomenų neskiriamumo principas). Kai ląstelės turinis patenka į valdymo įrenginį – reikšmė traktuojama, kaip komanda, o kai reikšmė patenka į aritmetinį loginį įrenginį – kaip duomenis.Valdymo įrenginys dekoduoja ląstelės turinį, nuskaitytą iš atminties tam tikru adresu ir toliau (jeigu yra būtinybė) dekoduotą komandą vykdo ALĮ.

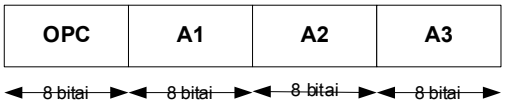
1.4. Architektūrų apžvalga

Mūsų anksčiau parodytoje komandos vykdymo schemoje liko neaišku, kaip atrodo vykdoma komanda. Pavyzdyje parodyta sudėties komanda turi du operandus – reikšmės, kurios atitinkamai įrašytos ląstelėse su adresais *x* ir *y*. Dabar panagrinėkime sudėties komandą iš kitos pusės – kaip ji atrodo atmintyje. Komanda reikalauja dar trijų operandų – pirmojo ir antrojo sudėties operandų ir rezultato. Pagal tai, kaip komandoje nurodomi operandai galima išskirti skirtingas architektūras – triadresinė, dviadresinė, vienadresinė ir beadresinė. Reikia pabrėžti, kad tokia architektūrų klasifikacija yra sąlyginė. Nagrinėjant kompiuterius kitu kampu, architektūrų klasių skaičius greičiausia pasikeis. Toliau panagrinėsime, kokie yra komandų formatai aukščiau išvardintose architektūrose [Bau03, pp. 16-17].

Laikysime, kad atmintis turi 2^{24} ląstelių, ląstelės dydis – 8 bitai. Bet kurios atminties ląstelės adresui užrašyti užtenka 3 baitų. Tegul komandos mašininis kodas užima vieną baitą. Pavaizduokime, kaip triadresinėje architektūroje užrašoma komanda:

OPC A1 A2 A3.

Čia OPC yra komandos mašininis kodas, A1, A2 ir A3 – komandos operandai (žr. pav.).



Komandos vykdymo schema tokioje architektūroje jau nagrinėta aukščiau:

R1 := [A2];

R2:= [A3];

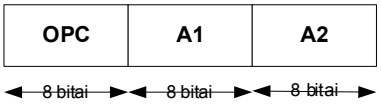
S := OPC(R1,R2);

[A1] := S;

Užrašas OPC(R1,R2) reiškia, kad procesorių vykdoma komanda su mašininiu kodu OPC, kurios operandai yra registruose R1 ir R2. Jeigu OPC nusako sudėties komandą, šita eilutė atrodys taip – S := R1 + R2.

Dviadresinėje architektūroje rezultato operandas nurodomas netiesiogiai – rezultatas užrašomas į pirmąjį operandą, sunaikinant prieš tai buvusią reikšmę. Komandos formatas atrodo taip:

OPC A1 A2.



Komandos vykdymo schema dviadresinėje architektūroje atrodo taip:

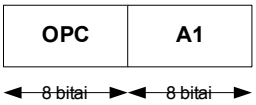
R1 := [A1];

R2 := [A2];

S := OPC(R1,R2);

[A1] := S;

Vienadresinėje architektūroje komandoje tiesiogiai užrašomas tik vienas operandas.



Bet į komandų sistemą įvedamos dvi specialios komandos – IN ir OUT. Komanda IN nuskaito atminties ląstelės turinį ir patalpina jį į akumuliatorių S. Komandos formatas – IN A1, vykdymo schema – S := [A1]. Komanda OUT įrašo akumuliatoriaus reikšmę komandoje nurodytu adresu. Komandos formatas analogiškas komandai IN – OUT

A1, vykdymo schema – [A1] := S. Tada operacijai OPC(A1,A2) įvykdyti tokioje architektūroje reikia trijų komandų:

IN A2;

OPC A1;

OUT A1;

Tokių komandų vykdymo schema:

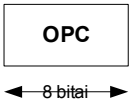
S := [A2];

R1 := [A1];

S : = OPC(S,R1);

[A1] := S;

Beadresinėje architektūroje naudojamas aparatūrinis stekas (todėl šita architektūra dar vadinama stekine architektūra). Visi veiksmai atliekami su steko viršūne.



Vienadresinėje architektūroje reikalingos dvi vienadresinės architektūros formato komandos – PUSH ir POP. PUSH A1 įrašo atminties ląstelės turinį į steko viršūnę, o komanda POP A1 – iš steko viršūnės į atminties ląstelę. Operacijai OPC(A1,A2) stekinėje architektūroje reikia 4 komandų:

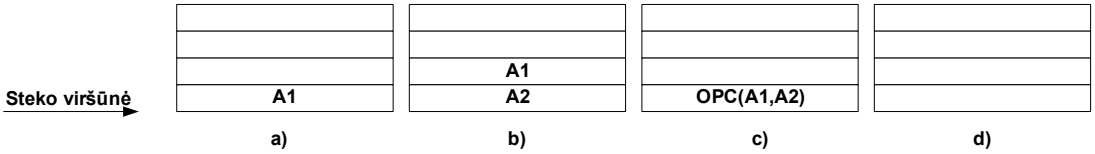
PUSH A1;

PUSH A2;

OPC;

POP A1;

Pakomentuokime vykdymo schemą: pirmoje eilutėje į steko viršūnę (SP) įrašomas atminties ląstelės adresu A1 turinis (žr. pav.1.4.1a), antroje – į steko viršūnę įrašoma reikšmė iš atminties adresu A2 (atkreipkite dėmesį, kad prieš įrašyta reikšmė iš adreso A1 nesugadinama, nes įrašymo į steką operacija automatiškai koreguoja steko viršūnės poziciją, 1.4.1b). Trečioje eilutėje vykdoma operacija su steko viršūnėje įrašyta reikšme ir viena reikšme aukščiau steke. Rezultatas įrašomas į steko viršūnę (žr. pav.1.4.1c). Paskutinėje eilutėje reikšmė iš steko viršūnės įrašoma į atminties ląstelę adresu A1 (žr. pav.1.4.1d).



1.4.1 pav. Operacijos vykdymo schema beadresinėje architektūroje

Panagrinėkime, kaip pateiktose architektūrose užrašomos programos. Tegul visose architektūrose apibrėžtos sudėties, atimties, daugybos, dalybos ir persiuntimo komandos. Taip pat tarkime, kad visose architektūrose šitos komandos turi tokius kodus: sudėties – 00, atimties – 01, daugybos – 02, dalybos – 03, persiuntimo – 04. Užrašykime programą, kurį apskaičiuoja tam tikrą reikšmę pagal formulę:

$$x := (a^2 + b^2)/(a+b)^2 .$$

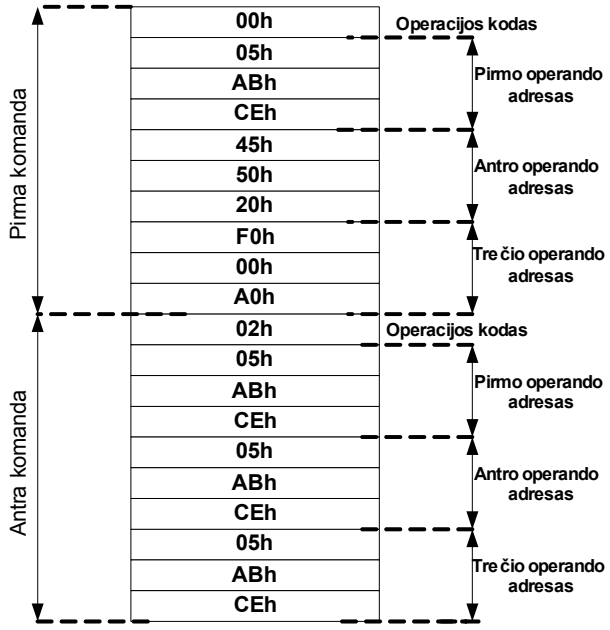
Tegul kintamojo x adresas yra 05ABCEh, kintamojo a adresas – 455020h, o kintamojo b – F000A0h. Triadresinėje architektūroje programa gali atrodyti taip:

00 05ABCEh 455020h F000A0h 02 05ABCEh 05ABCEh 05ABCEh

02 455020h 455020h 455020h 02 F000A0h F000A0h F000A0h

00 455020h 455020h F000A0h 03 05ABCEh 455020h 05ABCEh.

Atmintyje programos pirmos dvi komandos išsidėsto taip, kaip pavaizduota 1.4.2 pav.:



1.4.2 pav. Programos komandos atmintyje

Kaip matome, toks užrašymo būdas yra natūralus procesoriui, bet labai nepatogus žmogui. Programavimas mašininiais kodais neretai yra labai lėtas procesas, nes sunku išvengti klaidų. Programavimui pagreitinti buvo įvestos **mnemonikos** – simboliniai

mašininių kodų vardai. Pavyzdžiui, daugelyje architektūrų mūsų pateiktos komandos turi tokias mnemonikas: 00 – ADD, 01 – SUB, 02 – MUL, 03 – DIV, 04 – MOV. Dabar užrašykime tą pačią programą, naudojant mnemonikas:

```
ADD  x,a,b
MUL  x,x,x
MUL  a,a,a
MUL  b,b,b
ADD  a,a,b
DIV  x,a,x
```

Kaip matome, programa tapo žymiai lengviau skaitoma. Tam, kad programa, užrašyta mnemonikomis, būtų suprantama procesoriumi, ji yra apdorojama specialia programa, vadinama asembleriu. Tokia kalba – mnemonikos ir tam tikros pagalbinės konstrukcijos – vadinama asemblerio kalba. Asemblerio kalbą plačiau nagrinėsime penktame skyriuje.

Pabandykime užrašyti tą pačią programą dviadresinėje architektūroje ir parašykime ją asemblerio kalba:

```
MOV  x, a
ADD  x, b
MUL  x,x
MUL  a,a
MUL  b,b
ADD  a,b
DIV  a,x
MOV  x,a
```

Čia reikia pakomentuoti tik pirmą ir paskutines eilutes – reikšmė persiunčiama iš vienos atminties ląstelės į kitą. Triadresinėje architektūroje mes sugebėjom apsieiti be persiuntimo komandos, nes komandos formatas numato tiesioginį rezultato adresą.

Dabar pateiksim programą, kaip tai reikalauja vienadresinės architektūros taisyklės:

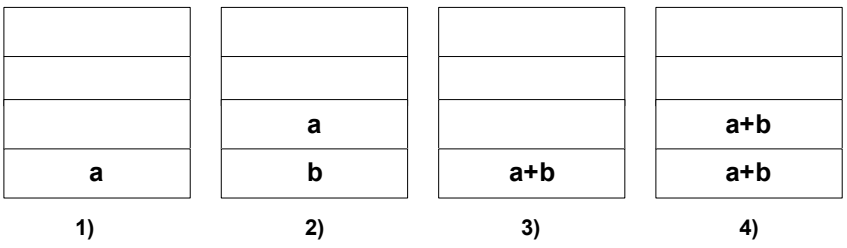
```
IN   a
ADD  b
OUT  b
MUL  b
```

```
OUT   x
IN    a
MUL   a
OUT   a
IN    b
MUL   b
ADD   a
DIV   x
OUT   x
```

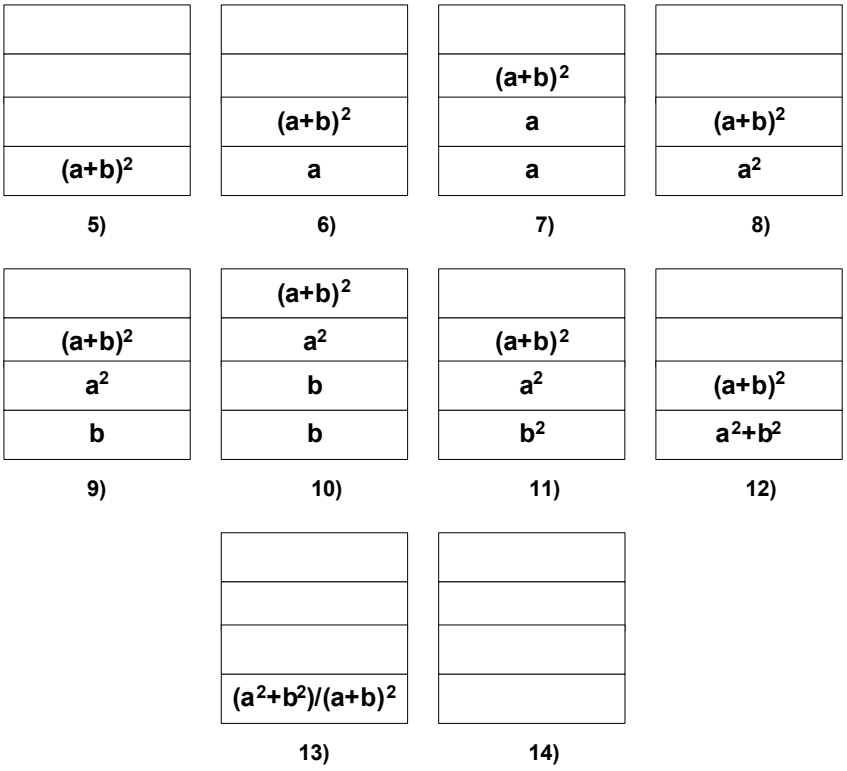
Beadresinėje architektūroje ta pati programa gali atrodyti taip:

```
PUSH a
PUSH  b
ADD
PUSH[1]
MUL
PUSH  a
PUSH  a
MUL
PUSH  b
PUSH  b
MUL
ADD
DIV
POP   x
```

Pateiktoje schemoje galima pamatyti, kaip keičiasi stekas, vykdant šias komandas viena po kitos.



[1] Komanda PUSH be operandų reiškia steko viršūnės dubliavimas



1.4.3 pav. Steko būsena

Be aukščiau išvardintų architektūrų dar yra architektūra, kurioje komandų formatas numato keturi operandai. Ketvirtas operandas skirtas laikyti sekančios komandos adresui.

Dabar panagrinėkime architektūrą, kuri vadinama architektūra su adresuojamais registrais. Šita architektūra nesilaiko vieno iš fon Neimano principų – homogeninės atminties principo. Architektūroje su adresuojamais registrais atmintis suskaidyta į dvi dalis. Kiekviena atmintis turi atskirą ląstelių numeraciją. Pirma atmintis – registrų atmintis, kurios dydis labai mažas, apie 10-20 ląstelių. Antra – pagrindinė atmintis, kurios dydis apie 1M – 100M ląstelių (gali būti ir žymiai daugiau). Dėl to, kad dviejų atminčių ląstelių numeracijos yra nepriklausomos, registrų atminties adresui užrašyti pakanka mažiau bitų. Pvz., jeigu mes turime 16 registrų, tai kiekvienam jų adresuoti užtenka keturių bitų. Registrų atmintis yra žymiai greitesnė už pagrindinę atmintį (10-40 kartų). Todėl programavimui šitoje architektūroje patartina kuo dažniau naudoti registrus.

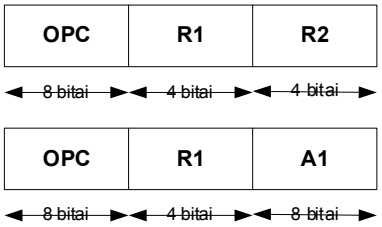
Panagrinėkime dviadresinę architektūrą su adresuojamais registrais, kurioje yra 16 registrų ir 2^{20} pagrindinės atminties ląstelių [Bau03, pp.19-20]. Kiekvienos ląstelės dydis – 20 dvejetainių skilčių. Tokioje architektūroje yra keli komandų formatai – viename abu operandai yra registrai, kitame registras ir operandas atmintyje. Pirmas

formatas yra registras-registras (RR), kitas registras-atmintis (RX). Šiuo atveju procesoriaus komandą galime užrašyti dviejuose formatuose:

OPC R1 R2

OPC R1 A1

Architektūros su adresuojamais registrais pagrindinis privalumas yra tai, kad komandų sistema yra lankstesnė ir komandos užima mažiau vietos atmintyje.



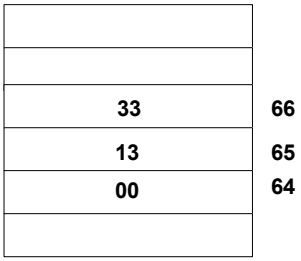
Bet valdymo įrenginys tampa sudėtingesniu (ir tuo pačiu brangesnis), nes vienai komandai reikia palaikyti kelis formatus. Pažiūrėkime, kaip šioje architektūroje gali atrodyti aukščiau pateiktas programos pavyzdys:

MOV *R1,a*
MOV *R3,R1*
MOV *R2,b*
ADD *R1,R2*
MUL *R1,R1*
MUL *R2,R2*
MUL *R3,R3*
ADD *R3,R2*
DIV *R3,R1*
MOV *x,R3*

Įveskime sąvoką - adresavimo būdas [Bau03, pp. 20-21][Mit03, pp.45-47]. Adresavimo būdas – operandų mašinos komandoje užrašymo būdas, t.y. taisyklė, pagal kurią nustatomi komandos operandų adresai ir reikšmės. Dažniausia adresavimo būdas nustatomas pagal operacijos kodą. Sudėties komandos pavyzdžiu parodysime adresavimo būdus vienadresinėje architektūroje. Tegul mnemonikos ADD_0, ADD_1 ir ADD_2 atitinkamai žymi sudėties operacijas su tiesioginiu, betarpišku ir netiesioginiu adresavimu. **Tiesioginis adresavimas**:

ADD_0 *x*

Šitas adresavimo būdas nustato, kad operandas, esantis komandoje yra adresas to skaičiaus, kurį reikia sudėti su akumuliatoriumi, t.y. S := S + [*x*]. T.y. jeigu *x* reikšmė būtų 65, tai operandas būtų atminties ląstelėje su adresu 65, operando reikmė būtų 13 (žr. pav.1.4.4).



1.4.4 pav. Tiesioginis adresavimas

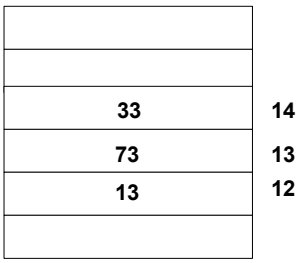
Betarpiškas adresavimas:

$ADD_1\ x$

Tokiu adresavimo būdu komandos operandas yra nurodytas pačioje komandoje. T.y. jeigu *x* reikšmė būtų 12, tai operando reikšmė būtų 12 – S:= S + *x*. Netiesioginis adresavimas:

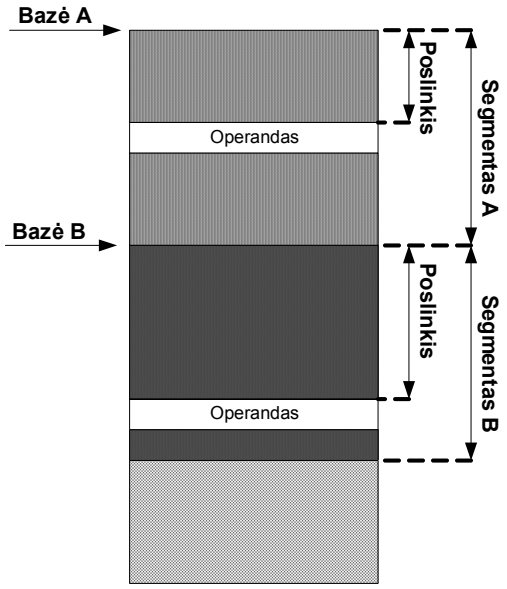
$ADD_2\ x$

Tokiu adresavimo būdu komandoje esantis skaičius traktuojamas, kaip ląstelės adresas, kurioje saugomas skaičius savo ruožtu yra adresas, kuriame yra komandos operandas. Pvz., jeigu *x* reikšmė būtų 12, o ląstelėje su adresu 12 būtų skaičius 13, tai komandos operandas būtų ląstelėje su adresu 13 – S := S + [[*x*]], operando reikšmė būtų 73 (žr. pav. 1.4.5).



1.4.5 pav. Netiesioginis adresavimas

Įveskime dar vieną sąvoką – adresų bazavimas. Adresų bazavimas – tai adreso užrašymas pavidalu β + Δ, kur β yra **bazinis adresas** (arba tiesiog bazė), o Δ - **poslinkis** nuo bazės. Pilnas adresas β + Δ vadinamas **absoliučiu adresu,** poslinkis nuo bazės dar vadinamas **vykdomuoju** arba **efektyviu adresu** (žr. pav. 1.4.6).



1.4.6 pav. Adresų bazavimas

Paaiškinsime tai tokiu pavyzdžiu – tarkime, dviadresinėje architektūroje mums reikia užprogramuoti tokią formulę $x := (a - b)/(a+b)$:

MOV x,a

MOV z[2],a

SUB x,b

ADD z,b

DIV x,z

Tarkime, kad mūsų kompiuteris turi 2^{20} (~ 1 milijonas) atminties ląstelių ir a,b,z ir x reikšmės atitinkamai yra A0000h, A0001h, A0002 ir A0003h. Tada mašininiais kodais aukščiau pateiktas fragmentas atrodys taip:

04 A0003h A0000h

04 A0002h A0000h

01 A0003h A0001h

00 A0002h A0001h

03 A0003h A0002h

Čia aiškiai matome perteklinę informaciją. Mums nebūtina komandoje laikyti visą adresą, užtektų tik poslinkio Δ, bazę galime laikyti specialiame registre (baziniame) BR. Tada, įvesdami dar vieną komandą LDB (LoaD Base register), galime sutrumpinti kodą:

[2] z yra papildomas kintamasis

LDB A0000h

…

04 003h 000h

04 002h 000h

01 003h 001h

00 002h 001h

03 003h 002h

Bet tokiu atveju, kaskart kai procesoriui reikia paimti operandą iš atminties, procesorius turi apskaičiuoti operando adresą pagal formulę $A = \beta + \Delta$. Adreso skaičiavimas vykdomas valdymo įrenginyje. Poslinkio dydis skirtingose architektūrose gali būti skirtingas. Mūsų atveju poslinkis yra 12 bitų (3 šešioliktainiai skaitmenys). Reiškia, maksimalus poslinkis nuo bazės yra 2^{12}. Kai turime adresų bazavimą, sakoma, kad atmintis suskirstyta į **segmentus**. Segmento pradžią nurodo bazinis registras, dar vadinamas segmentiniu registru. Toliau adresų bazavimą vadinsime **segmentavimu**.

Kaip matome, vienas iš segmentavimo privalumų – atminties taupymas. Bet pagrindinis privalumas yra tai, kad programos tampa nepriklausomos nuo padėties atmintyje. Be segmentavimo mes turėjome nurodyti absoliučius adresus pačioje programoje ir programa visada turėjo prasidėti nuo tam tikro adreso. Segmentavimas leidžia programose naudoti tik poslinkius. Kai programa yra kraunama į atmintį, operacinė sistema nustato segmentinį registrą ir programa gali prasidėti bet kurioje atminties ląstelėje.

Pratimai

Tarkime, kad visose nagrinėjamose architektūrose operacijos ADD,SUB,MUL,DIV,MOV atitinkamai turi tokius mašininius kodus – 00h,01h,02h,03h ir 04h. Operacijos kodas užima vieną baitą, atminties adresui saugoti reikia trijų baitų.

1 Koks yra galimas maksimalus adresas?

2 Kiek bitų užima komandos triadresinėje, dviadresinėje, vienadresinėje ir beadresinėje architektūrose?

3 Pateikite programos fragmentą, kuris apskaičiuoja formulę
$x := (a^2 + b^2)/(a^2 - a*b + b^2)$

3.1 triadresinėje architektūroje;

3.2 dviadresinėje architektūroje;

3.3 vienadresinėje architektūroje;
3.4 beadresinėje architektūroje:
3.5 architektūroje su adresuojamais registrais, kai yra 16 registrų;
3.6 tuos pačius fragmentus mašininiais kodais.

4 Pateikite programos fragmentą, kuris apskaičiuoja formulę $x := (a^2-b^2)/(a^2+b^2)$

4.1 architektūroje, kurios komandos turi 4 operandus (ketvirtas operandas yra sekančios komandos adresas);
4.2 tą patį fragmentą mašininiais kodais (programa prasideda adresu A00000h).

5 Užrašykite programos fragmentą beadresinei architektūrai, kuris apskaičiuoja formulę

5.1 $x := (a^2 + b^2)/(a^3 - a^2*b^2 + b^3)$ ir parodykite, kaip kinta stekas fragmento vykdymo metu.

6 Duotas programos fragmentas mašininiais kodais (šešioliktaine sistema) parašytas triadresinėi architektūrai, užrašykite jį asemblerio kalbą

6.1 02A00002A00000A0000002A00003A00001A00001
00A00002A00002A0000302A00005A00000A00000
02A00004A00005A0000302A00005A00005A00000
02A00003A00003A0000100A00005A00005A00003
01A00005A00005A0000303A00002A00002A00005;
6.2 Užrašykite tą patį fragmentą mašininiais kodais beadresinei architektūrai ir parodykite, kaip kinta stekas fragmento vykdymo metu;
6.3 Užrašykite tą patį fragmentą mašininiais kodais vienadresinei architektūrai;
6.4 Užrašykite tą patį fragmentą mašininiais kodais architektūrai su adresuojamais registrais.

7 Tarkime, sudėties komandos, kuri naudoja tiesioginį adresavimo būdą, mnemonika yra ADDT, sudėties komandos, kuri naudoja netiesioginį adresavimo būdą, mnemonika yra ADDN ir betarpiško adresavimo būdo atveju – ADDB. Tegu pradinė sumatoriaus reikšmė yra 0. Apskaičiuokite sumatoriaus reikšmę po įvykdyto programos fragmento, kai nurodytos atminties ląstelių reikšmės (žr. pav.1.4.7).

11	21
21	20
10	19
16	18
18	17
21	16
66	15
19	14
13	13
21	12
12	11
11	10

1.4.7 pav. Adresų bazavimas

7.1 ADDT 11
ADDT 10
ADDN 10
ADDN 11
ADDB 10

7.2	ADDN	20
	ADDN	12
	ADDT	12
	ADDB	21
7.3	ADDB	37
	ADDT	21
	ADDN	16
	ADDN	14
7.4	ADDN	10
	ADDN	11
	ADDN	12
	ADDN	13
	ADDN	14
	ADDT	15
	ADDN	16
	ADDN	17
	ADDB	18
7.5	ADDB	19
	ADDB	20
	ADDN	21
	ADDT	10
	ADDT	16
	ADDT	15
7.6	ADDN	11
	ADDN	12
	ADDB	66
	ADDT	15
7.7	ADDN	16
	ADDT	16
	ADDT	21
	ADDN	21
	ADDB	21

2. Duomenų formatai

2.1. Fiksuoto kablelio formatas

Aukščiau pateiktuose pavyzdžiuose ir pratimuose yra skaičiai užrašyti skirtingose skaičiavimo sistemose. **Skaičiavimo sistema** – tai taisyklės, kurios nusako kaip užrašyti skaičių. Pvz., skaičius 10 dešimtainėje sistemoje užrašomas simboliais ”1” ir ”0”, šešioliktainėje sistemoje skaičius 10 užrašomas simboliu ”A”, o dvejetainėje sistemoje tokia ”0” ir ”1” kombinacija - 1010. Tam, kad atskirti, kokioje sistemoje užrašytas skaičius, po skaičiaus rašysime indeksą, kuris reikš skaičiavimo sistemos pagrindą, pvz., 1010_2 – dvejetainėje sistemoje, 10_{10} – dešimtainėje sistemoje, A_{16} – šešioliktainėje sistemoje. Dažniausia šešioliktainei sistemai žymėti naudosime raidę h, pvz. 0Ah.

Skaičiavimo sistemos konstruojamos pagal vieną bendrą principą [Fom87, pp.11-12], [Mit03, pp. 4-5]. Išrenkamas skaičius p – sistemos pagrindas. Kiekvienas skaičius N skaičiavimo sistemoje užrašomas kaip pagrindo laipsnių kombinacija su koeficientais nuo 0 iki p-1, t.y. pavidalu

$$a_k \cdot p^k + a_{k-1} \cdot p^{k-1} + \ \ldots + a_1 \cdot p + a_0, \text{ kur } \ 0 \le a_i < p, \text{ kai i=0..k.}$$

Toks skaičius užrašomas trumpesniu pavidalu $(a_k a_{k-1} \ldots a_1 a_0)_p$. Šitame užraše kiekvieno skaitmens reikšmė priklauso nuo pozicijos. Pvz., skaičiuje 555_{10} penketas dalyvauja 3 kartus. Bet dešinioji reiškia penkis vienetus, antra iš dešinės – penkios dešimtis, o trečia iš dešinės penkis šimtus. Jeigu mes naudotume kokia nors kitą skaičiavimo sistema p, penketai atitinkamai reikštų 5, $5p$ ir $5p^2$. Skaičiavimo sistemos, sukonstruotos tokiu principu vadinamos **pozicinėmis**. Yra kitos sistemos – **nepozicinės** skaičiavimo sistemos. Nepozicinės sistemos pavyzdys – romėniški skaitmenys. Šitoje sistemoje yra specialių simbolių rinkinys, I – vienetas, V – penki, X – dešimt, L – penkiasdešimt, C – šimtas ir t.t. Kiekvienas skaičius užrašomas kaip šių simbolių kombinacija. Pavyzdžiui skaičius 77 užrašomas LXXVII. Nepozicinėje sistemoje kiekvieno simbolio reikšmė nepriklauso nuo pozicijos. Pavyzdyje simbolius X dalyvauja du kartus, bet visada jis reiškia viena dydį – dešimt vienetų.

Pozicinės skaičiavimo sistemos patogios tuo, kad jos leidžia užrašyti didelius skaičius, naudojant mažai simbolių. Kitas privalumas – aritmetinių operacijų paprastumas.

Panagrinėkime, kaip atliekama sudėties operacija pozicinėse skaičiavimo sistemose. Kaip dešimtainėje sistemoje, pirma mes sudedam vienetus, paskui

pereinam prie sekančios skilties ir t.t, kol neprieisime prie vyriausios skilties. Būtina turėti omenyje, kad kai skiltyje gaunasi suma didesnė už sistemos pagrindą, reikia padaryti pernešimą į sekančią skiltį. Pavyzdžiui,

$$\begin{array}{ll} & (23651)_8 \\ + & (\underline{17043})_8 \\ & (42714)_8 \end{array}$$

$$\begin{array}{ll} & (423)_6 \\ + & (\underline{1341})_6 \\ & 2204_6 \end{array}$$

Vienas iš svarbesnių uždavinių, susietas su pozicinėmis skaičiavimo sistemomis, yra pervedimas iš vienos sistemos į kitą. Detaliai pervedimo algoritmas nagrinėjamas [Mit03, pp.5-6]. Čia pateiksime pavyzdžius ir tam tikras sistemų pervedimo iš ir į dvejetainę sistemą gudrybes.

Pagrindinės skaičiavimo sistemos, kurios bus naudojamos šitame kurse, yra dvejetainė, dešimtainė ir šešioliktainė. Verčiant skaičių iš dvejetainės sistemos į šešioliktainę, reikia dvejetainėje išraiškoje išskirti grupes po 4 bitus, pradedant nuo mažiausiai reikšmingo bito (pirmo iš dešinės). Kiekviena tokia grupė šešioliktainėje sistemoje užrašoma vienu šešioliktainiu simboliu, pvz., 1010 – Ah. Ir atvirkščiai, kiekvienas skaitmuo šešioliktainėje sistemoje atvaizduojamas į 4 bitus. Norint išversti skaičių dešimtainėje sistemoje į dvejetainę sistemą, reikia išskirti dvejeto laipsnius., pvz., skaičius 47_{10} gali būti užrašytas kaip suma 32+8+4+2+1, t.y. 101111_2. Verčiant į dvejetainę sistemą visada verta padaryti paprastą patikrinimą, jeigu verčiamas skaičius lyginis – mažiausiai reikšmingas bitas turi būti 0. Norint tą patį skaičių išversti į šešioliktainę sistemą, reikia išskirti 16 laipsnius, pvz., skaičius 87_{10} gali būti užrašytas kaip suma $16 \cdot 5+7$, t.y. 57h.

Duomenų formatas yra tam tikras informacijos kodavimo būdas, kuris priklauso nuo procesoriaus komandų sistemos. Duomenų formatai yra komandų sistemos atributas, apibrėžimo sudėtinė dalis[Mit03, p.55].

Reikia atkreipti dėmesį į tai, kad duomenų formatas tai ne tas pats, kas duomenų tipas. Duomenų tipai nusako reikšmių aibę, kuriai užkoduoti gali būti panaudoti skirtingi formatai.

Mūsų nagrinėjamoje architektūroje (Intel 8088) yra tokie duomenų formatai – **skaičiai be ženklo**, **skaičiai su ženklu**, **dešimtainiai supakuoti skaičiai**, **dešimtainiai išpakuoti skaičiai**, **simbolinis formatas**. Iš pradžių, panagrinėkime fiksuoto kablelio formatą, t.y. skaičius be ir su ženklu.

Kaip jau buvo minėta, bitai atmintyje gali būti traktuojami skirtingai. Valdymo įrenginys traktuoja baitą, kaip procesoriaus komandą, o aritmetinis loginis įrenginys – kaip skaičių. Taip pat, kiekvienas atmintyje esantis baitas gali būti traktuojamas programa kaip skaičius su ženklu arba be ženklo. Bet vien pagal baito reikšmę negalima nustatyti koks saugomas skaičius – su ženklu, ar be ženklo. Tik programa žino, kas yra atmintyje. T.y. kompiuteryje apibrėžtos dvi skaičiavimo sistemos skaičiams su ženklu ir skaičiams be ženklo atvaizduoti.

Skaičių be ženklo formatu visi skaičiaus bitai interpretuojami kaip dvejetainis skaičius. Toks atvaizdavimas vadinasi tiesioginiu kodu. Pavyzdžiui, skaičius 17_{10} tiesioginiu kodu viename baite bus užrašytas kaip 00010001_2.

Jeigu invertuoti[3] tiesioginį kodą, tai gausime atvirkštinį skaičiaus kodą. Pavyzdžiui, skaičiaus 17_{10} atvirkštinis kodas viename baite yra 11101110_2. Neigiamiems skaičiams atvaizduoti naudojamas **papildomas kodas**, kurį galima gauti iš atvirkštinio, pridėjus vienetą. Pavyzdžiui, skaičiaus 17_{10} papildomas kodas yra 11101111_2 – taip skaičiaus -17 būtų užrašytas atmintyje. Toks neigiamų skaičių kodavimas leidžia apsieiti be atimties operacijos, realizuotos aparatūriškai. Vietoj atimties operacijos galima naudoti sudėtį ir neigiamų skaičių kodavimą.

Paimkime paprasta pavyzdį: tarkime, reikia apskaičiuoti 25 + (-17). Jeigu operaciją reikia atlikti moduliu 100 (mod 100), tai užkodavę -17 skaičiumi 83, gausime (25 + 83) mod 100 = 8. Šiuo atveju, skaičius 83 yra atvirkštinis skaičiui 17 sudėties moduliu 100 atžvilgiu. Neigiamiems skaičiams koduoti parenkamas toks teigiamas skaičius, kuris yra atvirkštinis operacijos moduliu 2^n atžvilgiu [Mit03, pp. 55-56].

Jeigu pažiūrėsime į papildomą skaičiaus 17_{10} kodą, pamatysime, kad vyriausias bitas yra 1. Kai kalbame apie skaičius su ženklu, šitas bitas vadinamas ženklo bitu. Jeigu jis nustatytas į 1, skaičius yra neigiamas, jeigu į 0 – teigiamas. Bet iš tikrųjų,

[3] Pakeisti 0 į 1, o 1 į 0

jokio ženklo nėra, juk reikšmės atmintyje yra tos pačios ir ženklas šiuo atveju tik interpretavimo klausimas.

Galime pateikti dar vieną papildomo skaičiaus kodo skaičiavimo algoritmą. Skaičiaus *k* papildomam kodui rasti reikia tiesioginiame kode užrašyti reikšmę 2^n-|*k*|, kur *n* yra 8 arba 16 (ir skaičiai su ženklu, ir skaičiai be ženklo gali užimti vieną arba du baitus).

Dvejetainis formatas yra ekvivalentus šešioliktainiam formatui, todėl nėra atskirai kalbama apie šešioliktainius skaičius su ženklu ar be ženklo. Ar skaičius su ženklu viename baite užrašytas šešioliktainėje sistemoje yra neigiamas galima nustatyti pagal vyriausią šešioliktainį skaitmenį – jeigu skaitmuo ≤ 7h, skaičius yra teigiamas, kitaip – neigiamas.

Kai mes atliekame aritmetines operacijas su fiksuoto kablelio formatu galime gauti nekorektiškus rezultatus. Pavyzdžiui, sudėdami du skaičius viename baite be ženklo 11111010_2 ir 00001101_2, gausime $100000111_2 = 263$, bet mes atliekam skaičiavimus moduliu 2^8, todėl vyriausią bitą turime atmesti $00000111_2 = 7_{10}$. Jeigu skaičius interpretuosime kaip skaičius be ženklo, tai mūsų rezultatas 250 + 13 = 263 yra nekorektiškas (nes išeina už lestinas ribas 0..255). Jeigu skaičius interpretuosime kaip skaičius su ženklu, tai rezultatas –6 + 13 = 7 yra korektiškas. Vienu atveju rezultatas yra korektiškas, kitu ne – viskas priklauso nuo interpretacijos. Procesorius nežino kaip komandoje interpretuojamos reikšmės, todėl turi būti pranešta apie galimą nekorektiškumą. Tam naudojama požymių sąvoka – procesoriuje išskiriami specialūs požymiai operacijos rezultato korektiškumui užfiksuoti. Tai yra vieno bito laukai, mūsų nagrinėjamoje architektūroje šitie požymiai vadinami CF (*Carry Flag*) ir OF (*Overflow flag*) ir saugomi specialiame registre FLAGS. Požymis CF žymi operacijos su skaičiais be ženklo rezultato korektiškumą, o požymis OF – skaičių su ženklu. Kai požymio reikšmė yra 0, rezultatas yra korektiškas, o kai 1 – įvyko pernešimas arba perpildymas.

Pateiksime algoritmą [Bau03, pp.26-27], pagal kurią galima nustatyti CF ir OF požymių reikšmes sudėties ir atimties operacijų rezultatams. Norėdami apskaičiuoti skaičių *x* ir *y* sumą bei nustatyti požymius OF ir CF nubraižykime tokią lentelę:

	Be ženklo	Su ženklu
x *y*		
rezultatas		

kur pirmame stulpelyje antroje eilutėje užrašysime skaičius x ir y vienas po kito dvejetainėje sistemoje, antrame stulpelyje skaičius *x* ir *y* dešimtainėje sistemoje, interpretuodami juos kaip skaičius be ženklo. Trečiame stulpelyje skaičius x ir y dešimtainėje sistemoje, interpretuodami juos kaip skaičius su ženklu. Trečios eilutės stulpeliuose užrašysime rezultatą atitinkamai dvejetainėje sistemoje, dešimtainėje sistemoje (skaičiai be ženklo), dešimtainėje sistemoje (skaičiai su ženklu).

Tarkime, skaičiai *x* ir *y* yra 11100100_2 ir 00010011_2, tada lentelė atrodys taip:

	Be ženklo	Su ženklu
11100100 00010011	228 19	-28 19
11110111	247	-9

Matome, kad interpretuodami reikšmes tiek kaip skaičius su ženklu, tiek be ženklo gavo korektiškus rezultatus. Reiškia požymiai CF ir OF nustatomi į 0. Paimkime kitą pavyzdį, $x=11111111_2$, $y=00001001_2$. Šiuo atveju lentelė atrodys taip:

	Be ženklo	Su ženklu
11111111 00001001	255 9	-1 9
100001000	264	8

Šiuo atveju rezultatas skaičiams be ženklo yra nekorektiškas, o skaičiams su ženklu korektiškas, reiškia CF=1, OF=0. Panagrinėkime dar vieną pavyzdį, $x=01111111_2$ ir $y=00000011_2$:

	Be ženklo	Su ženklu
01111111 00000011	127 3	127 3
10000010	130	-126

Čia matome, kad skaičių be ženklo atveju rezultatas yra korektiškas, o skaičių su ženklu ne, reiškia CF=0, OF=1. Analogiškais samprotavimais galima nustatyti požymių OF ir CF reikšmes ir atimties operacijai.

Pratimai

1 Užrašykite dešimtaines reikšmes dvejetainėje sistemoje.

1.1	11	1.12	93	1.23	128
1.2	12	1.13	71	1.24	127
1.3	37	1.14	76	1.25	129
1.4	45	1.15	120	1.26	511
1.5	79	1.16	197	1.27	513
1.6	87	1.17	225	1.28	1024
1.7	54	1.18	252	1.29	1023
1.8	47	1.19	245	1.30	661
1.9	48	1.20	255	1.31	1025
1.10	98	1.21	256		
1.11	96	1.22	512		

2 Užrašykite dešimtaines reikšmes šešioliktainėje sistemoje.

2.1	12	2.11	256	2.21	97
2.2	17	2.12	254	2.22	34
2.3	15	2.13	1024	2.23	37
2.4	32	2.14	65536	2.24	69
2.5	31	2.15	65535	2.25	198
2.6	33	2.16	544	2.26	177
2.7	127	2.17	512	2.27	215
2.8	128	2.18	511	2.28	656
2.9	129	2.19	513		
2.10	255	2.20	64		

3 Užrašykite dvejetainės reikšmes šešioliktainėje sistemoje.

3.1	1110111	3.7	0101	3.13	11111100001111
3.2	11111100101010	3.8	1010	3.14	111101011111
3.3	11111111111101	3.9	1011	3.15	11101011111111
3.4	11101	3.10	111111	3.16	11100101111101
3.5	1111	3.11	100010101100	3.17	11111100110011
3.6	10000	3.12	01101111010111		

4 Užrašykite šešioliktaines reikšmes dvejetainėje sistemoje.

4.1	0Fh	4.5	54848h	4.9	11248FFh
4.2	0EAh	4.6	54542324h	4.10	0FFFFh
4.3	0AEDAh	4.7	445477ECDh	4.11	6698EADh
4.4	745FECh	4.8	DE55847Ah	4.12	E45457DBh

5 Užrašykite šešioliktaines reikšmes dešimtainėje sistemoje.

5.1	0FFFFh	5.5	6658h	5.9	7747h
5.2	0FA00h	5.6	54ABh	5.10	65EBh
5.3	0FECAh	5.7	9AB0h	5.11	7FFFh
5.4	4587h	5.8	CAD31h	5.12	7FF

6 Apskaičiuokite dešimtainių reikšmių viename baite sumą ir nustatykite požymius CF ir OF.

6.1	128 ir 253	6.6	–128 ir 68	6.11	97 ir 133
6.2	128 ir 250	6.7	–56 ir 87	6.12	13 ir 255
6.3	–128 ir –6	6.8	–88 ir 101	6.13	64 ir 64
6.4	–128 ir –125	6.9	96 ir 56	6.14	37 ir 115
6.5	–128 ir 124	6.10	189 ir 14	6.15	110 ir 111

2.2. Simbolinis formatas

Intel 8088 architektūros kompiuteriuose naudojamas **ASCII** (American Standard Code for Information Interchange) simbolinis formatas simboliams užkoduoti. Kiekvienam simboliui šitame formate priskiriamas vieno baito kodas, t.y. iš viso galima užkoduoti 256 simbolius. Kodai nuo 00h iki 1Fh nėra atvaizduojami kokiu nors simboliu. Dažniausiai naudojamų simbolių lentelė pateikta [Mit03, pp.58-59]. Jeigu pažiūrėsime į šios lentelės dalį, kur pateikti skaitmenų kodai, tai pastebėsime, kad iš skaitmens gauti atvaizduojantį simbolį galima tiesiog pridėjus 30h prie skaitmens. Pvz., jeigu norime gauti skaitmens 4 simbolį ASCII lentelėje, turime prie skaičiaus 4 pridėti 30h ir gausime 34h, o tai yra reikalingas simbolis. Taip reikia atkreipti dėmesį, kad skirtumas tarp abėcėlės didžiųjų ir mažųjų raidžių kodų yra pastovus. T.y. norint paversti simbolius ‘a’ ir ‘b’ simboliais ‘A’ ir ‘B’ atitinkamai, reikia iš jų ASCII kodų atimti reikšmę 20h (žr. [Mit03 pp.58-59]).

Panagrinėkime, kaip galima fiksuoto formato skaičius užrašyti simboliniame formate. Tarkime, norime užrašyti skaičių 45_{10} simbolinių formatų – tam reikia užrašyti kiekvieno skaičiaus skaitmens simbolį, 34h 35h.

0 NUL	16 DLE	32	48 0	64 @	80 P	96 `	112 p
1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
4 EOT	20 DC4	36 $	52 4	68 D	84 T	100 d	116 t
5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
9 TAB	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124 |
13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 DEL

2.2.1 pav. ASCII lentelė

Pratimai

1 Užkoduoti simbolių seką ASCII kodais

1.1	‘AaLKJSDaa’	1.4	‘C++’	1.7	‘Aaaa bCdr0\x’
1.2	‘0123456789’	1.5	‘if(a>9){b++;}’	1.8	‘a1 b2 c3 d4’
1.3	‘Pascal’	1.6	‘365aAdDvV’	1.9	‘AAEERRCCSS’

2 ASCII kodus paversti simboliais

2.1	30h,31h,32h,34h,	2.3	67, 43,43
2.2	56h,68h,65h	2.4	40,97,43,66,41

2.3. Dešimtainiai skaičiai

Fiksuoto kablelio formato skaičiai yra iš labai siauro diapazono – skaičius be ženklo dvejuose baituose kinta nuo 0 iki 65535, o skaičiai su ženklu nuo -32768 iki 32767. Akivaizdu, kad šitų reikšmių neužtenka daugeliui uždavinių. Kompiuterių architektūrose didesniam reikšmių diapazonui gauti naudojami slankaus kablelio formato skaičiai ir **dešimtainiai skaičiai**. Slankaus kablelio formato skaičius nagrinėsime sekančiame poskyryje, čia apžvelgsime dešimtainius skaičius.

Dešimtainiai formato skaičiai (anglų literatūroje žymimi **BCD** – binary coded decimals) yra viename baite. Dešimtainiai skaičiai yra **supakuoti** ir **išpakuoti**. Išpakuotas dešimtainis skaičius kinta nuo 0 iki 9 ir užrašomas viename baite – 0000XXXX, kur XXXX kinta nuo 0000_2 iki 1001_2 (žr. [Mit03, p.56]). Supakuotas dešimtainis formatas išnaudoja visą baitą ir kinta nuo 0 iki 99 – XXXXYYYY, kur XXXX ir YYYY kinta 0000_2 iki 1001_2. Veiksmai su dešimtainiais skaičiais yra mažiau efektyvūs, negu su fiksuoto kablelio formato skaičiais. Bet dešimtainius skaičius yra žymiai lengviau konvertuojami į simbolinį formatą. Išpakuoti dešimtainiai skaičiai naudojami kaip tarpinis formatas tarp dešimtainio supakuoto formato ir simbolinio formato. Pvz., norėdami užrašyti supakuotą dešimtainį skaičių 56 (01010110_2), mums reikia užrašyti skaičių išpakuotame formate – 00000101_2 00000110_2 ir pridėti prie kiekvieno baito reikšmę 30h (žr. ASCII kodų lentelę, [Mit03,pp.58-59]). Fiksuoto kablelio skaičiams vertimas į simbolinį formatą užima daugiau veiksmų, tarp kurių yra brangiau kainuojanti dalybos operacija.

Panagrinėkime, kaip galime užrašyti skaičių 37 dešimtainiame formate. Išpakuotame formate skaičiui pavaizduoti reikia dviejų baitų – 00000011_2 00000111_2. Supakuotame formate skaičius užima vieną baitą, po vieną skaitmenį pusbaityje – 00110111_2.

Dešimtainiams skaičiams taip pat apibrėžtos ir aritmetinės operacijos. Bet su jomis reikia atlikti papildomus veiksmus. Sudedami skaičius mes gausime nekorektiškus rezultatus (žr. [Mit03, p. 58]). Pavyzdžiui,

$$+\frac{05}{06}=0\text{Bh}.$$

Gautą rezultatą reikia pakoreguoti, pridedant 6 (dirbame šešioliktainėje sistemoje, o vaizduojame dešimtainius skaičius). 0Bh+06h = 11h, t.y. 00010001_2 dešimtainio supakuoto formato atveju, arba 0000000100000001_2 išpakuoto formato atveju. Plačiau aritmetines operacijas su dešimtainiais skaičiais panagrinėsime poskyryje 4.3.

Pažiūrėkime, kaip nustatyti tą atvejį, kada reikia atlikti rezultato korekciją. Pirma, panagrinėkime korekcijos taisykles išpakuotiems dešimtainiams skaičiams. Akivaizdu, kad korekciją reikia atlikti, kai mažiausiam pusbaityje *LN* (Low Nibble) esanti reikšmė yra didesnė už 1001_2 (nes to reikalauja formato apibrėžimas). Pavyzdžiui, sudedame dvi dešimtainio išpakuoto formato reikšmes – 07+08. Rezultate gauname šešioliktainę reikšmę 0Fh. Akivaizdu, reikalinga rezultato korekcija 0Fh+06h=15h. Bet gauta reikšmė dar nėra korektiška, nes vyriausiam pusbaityje *HN* (High Nibble) reikšmė neatitinka mūso formato. Dar reikia išvalyti vyriausią pusbaitį ir atitinkamai pakoreguoti vyresnįjį rezultato baitą HB (High Byte, nes rezultatas gali netilpti į vieną baitą). T.y. LB:=LB & 0Fh, HB:=HB+1. Mūsų pavyzdyje tai atrodytų taip:

1. 07h+08h = 0Fh, LB=0Fh, LN=Fh, HN=0h
2. 0Fh + 06h = 15h, LB=15h, LN=5h, HN=1h
3. LB:=15h & 0Fh, HB:=HB+1, LB=05h, HB=01h (jeigu pradinė HB reikšmė buvo 00h)

Rezultatas saugomas dviejuose baituose HB ir LB.

Panagrinėkime kitą atvejį, kai mažesniojo pusbaičio reikšmė yra leistinose ribose, bet reikia atlikti korekciją. Tarkime, turime sudėti dvi dešimtainio išpakuoto formato reikšmes 09h + 08h. Rezultate gauname reikšmę 11h, t.y. LN=1h, HN=1h. Šiuo atveju taip pat turime atlikti rezultato korekciją. Šiam atvejui užfiksuoti procesoriuje yra požymis AF (*Auxiliary Flag*). Jeigu AF=1, reikia atlikti korekciją, priešingu atveju – ne. Požymis nustatomas procesoriumi po operacijos rezultato suformavimo. Požymis AF panašus į požymį CF tuo, kad užfiksuoja pernešimą. Bet AF užfiksuoja pernešimą iš jaunesniojo pusbaičio į vyresnįjį, o ne iš baito į baitą. Pavyzdžiui,

$$+\frac{1000}{0100}=1100_2,\ \text{pernešimo iš jaunesniojo pusbaičio nėra, AF=0,}$$

$$+\frac{1000}{1001}=10001_2$$, yra pernešimas iš jaunesniojo pusbaičio, AF=1.

Užrašykime dabar korekcijos algoritmą pseudokodu:

```
if( (LB & 0Fh)>09h  || AF=1)
{
        LB := LB + 06h;
        HB := HB + 1;
        LB := LB  & 0Fh;
        AF := 1;
        CF := 1;
}
```

Dabar panagrinėkime rezultato korekciją dešimtainiams supakuotiems skaičiams. Korekcijos algoritmas supakuotiems skaičiams panašus į analogišką algoritmą išpakuotiems skaičiams, tik supakuotų skaičių atveju reikia atskirai koreguoti vyresnįjį ir jaunesnįjį pusbaičius.

```
if( (LN>9h) || AF=1 )
{
    LB := LB + 06h;
    AF := 1;
}
if( (HN)>9h || CF=1 )
{
    HN := HN + 60h;
    CF := 1;
}
```

Pakomentuokime algoritmą tokiais pavyzdžiais – 38h+34h ir 91h+83h.

$$+\frac{00111000}{00110100}=01101100_2$$

Šiuo atveju AF=0, CF=0. Pirma algoritmo sąlyga patenkinta, todėl įvyksta jaunesniojo pusbaičio korekcija: LB:=6Ch+06h, t.y. LB=72h, AF=1. Antra sąlyga nėra patenkinama, todėl neįvyksta vyresniojo pusbaičio korekcija.

$$+\frac{10010001}{10000100}=100010100_2$$

Šiuo atveju AF=0, CF=1. Pirma algoritmo sąlyga yra neteisinga, todėl neįvyksta jaunesniojo pusbaičio korekcija. Antra sąlyga yra teisinga, todėl įvyksta vyresniojo pusbaičio korekcija: LB:=LB+60h, CF:=1. T.y. LB=74h, CF=1.

Reikia pabrėžti kad programuojant, programuotojui nereikia pačiam nustatyti, ar reikalinga rezultato korekcija. Tam architektūroje egzistuoja specialios komandos, kurios pagal požymius ir rezultatą atlieka korekciją, programuotojui reikia tik užrašyti tokią komandą iškart po aritmetinės operacijos.

Pratimai

1 Užrašyti fiksuoto kablelio formato skaičius dešimtainiu supakuotu formatu:

1.1 12; 1.5 255; 1.9 874;
1.2 65; 1.6 0; 1.10 555.
1.3 253; 1.7 128;
1.4 127; 1.8 657;

2 Užrašyti fiksuoto kablelio formato skaičius dešimtainiu išpakuotu formatu;

2.1 98; 2.5 128; 2.9 1111;
2.2 54; 2.6 333; 2.10 25;
2.3 11; 2.7 847; 2.11 0.
2.4 2; 2.8 647;

3 Atlikti aritmetinius veiksmus su dešimtainiais supakuotais skaičiais:

3.1 56+45; 3.3 75+11; 3.5 05+87;
3.2 12+55; 3.4 02+09; 3.6 33+55.

4 Atlikti aritmetinius veiksmus su dešimtainiais išpakuotais skaičiais:

4.1 05+07; 4.3 09+09; 4.5 04+08.
4.2 08+09; 4.4 00+07;

2.4. Matematinio procesoriaus duomenų formatai

Mūsų nagrinėjamas procesorius negali apdoroti slankaus kablelio formato skaičius. Veiksmus su slankaus kablelio skaičiais gali vykdyti tik matematinis procesorius Intel 8087 (nuo Intel i486DX matematinis procesorius yra pagrindinės mikroschemos viduje).

Kiekvieną realų skaičių α (išskyrus 0.0) galima užrašyti pavidalu

$$\alpha=\pm 1.M*2^E,$$

kur M yra skaičiaus mantisė, E – eilė. Tokie skaičiai vadinami normalizuotais – pirmas daugiklis tenkina nelygybę $1.0 \le 1.M \le 2.0$. Pavyzdžiui, užrašykime skaičių 7.375. Pervesim skaičių į dvejetainę sistemą – 111.011_2 (žr. [Mit03, pp.5-6]). Normalizuotas skaičius yra $1.11011*2^2$. T.y. skaičiaus mantisė yra 11011_2, o eilė 2.

Matematinio procesoriaus Intel 8087 duomenų formatai atitinka tarptautinį standartą IEEE 754. Standarte apibrėžti du formatai – trumpas realus ir ilgas realus. Trumpas realus užrašomas keturiuose baituose (32 bitai) – bitas 31 yra ženklo bitas, bituose nuo 30 iki 23 imtinai (8 bitai) saugoma charakteristika, bituose nuo 0 iki 22 imtinai saugoma skaičiaus mantisė (23 bitai):

±	charakteristika	mantisė
1 bitas	8 bitai	23 bitai

Ilgas realus užrašomas aštuoniuose baituose (64 bitai) – 63 bitas yra ženklo bitas, bituose nuo 62 iki 52 imtinai saugoma charakteristika (11 bitų), bituose nuo 51 iki 0 saugoma skaičiaus mantisė (52 bitai):

±	charakteristika	mantisė
1 bitas	11 bitų	52 bitai

Normalizuoto skaičiaus užraše dalyvauja mantisė ir eilė, o duomenų formatuose mantisė ir charakteristika. Charakteristika yra pastumta eilė, trumpo realaus formato atveju eilė pastumta per 127, t.y. charakteristika = E+127, o ilgo realaus formato atveju per 1023, t.y. charakteristika = E+1023. Eilė kinta nuo -127 iki 127 trumpo realaus atveju, ir nuo –1023 iki 1023 ilgo realaus formato atveju. Charakteristika naudojama skaičių palyginimui supaprastinti (žr. [Mit03, p.101]).

Pažiūrėkime, kaip aukščiau pavyzdyje nagrinėtas skaičius 7.375 užrašomas matematinio procesoriaus formatuose. Normalizuotas skaičius yra $1.11011*2^2$, kur mantisė yra 11011_2, o eilė yra 2. Charakteristika yra pastumta eilė, t.y. charakteristika = eilė + 127 = 2 + 127 = 129_{10} = 10000001_2. Užrašykime visą skaičių:

±	charakteristika	mantisė
0	10000001	11011000000000000000

Ženklo bitas yra 0 (nes skaičius yra teigiamas), mantisė papildoma nuliais iš dešinės iki reikiamo bitų skaičiaus (23 bitai). Užrašykime gautą skaičių šešioliktainėje sistemoje 40 EC 00 00h.

Užrašykime tą patį skaičių ilgo realaus formatu – charakteristika = eilė + 1023 = $2+1023=1025_{10}=10000000001_2$:

±	charakteristika	mantisė
0	10000000001	1101100000000000000000000000000000000000000000000000

Ženklo bitas yra 0, mantisė papildoma nuliais iš dešinės iki reikiamo bitų skaičiaus (52 bitai). Šešioliktainėje sistemoje skaičius atrodys taip: 40 1D 80 00 00 00 00 00h.

Pastebėkime, kad realių skaičių aibė nagrinėjamoje architektūroje (ir šiaip visuose diskretiniuose kompiuteriuose) nesudaro kontinuumo. Todėl mes negalime atvaizduoti visų įmanomų realių skaičių, pvz., negalime tiksliai atvaizduoti skaičiaus $\frac{1}{3}$ nei trumpo, nei ilgo realaus formatuose. Tokiu atveju reikia imti artimiausią skaičių

iš tų, kuriuos galima atvaizduoti viename iš slankaus kablelio formatų. Toks veiksmas vadinamas **apvalinimu**.

Matematiniame procesoriuje papildomai apibrėžtas vidinis realus formatas, skirtas skaičiams kurie netelpa į formatą ”ilgas realus” (žr., [Mit03, p.102]).

Pratimai

1 Normalizuoti dešimtaines reikšmes.

1.1	15	1.6	20,1	1.11	0,0025
1.2	31	1.7	–4,25	1.12	8,18
1.3	32	1.8	–56,75	1.13	99,375
1.4	87	1.9	–50,5	1.14	127,55
1.5	12,12	1.10	1023	1.15	128,25

2 Užrašyti dešimtaines reikšmes formatu ”trumpas realus”.

2.1	19	2.11	–128,45	2.21	–67,1
2.2	5	2.12	656,25	2.22	109
2.3	97	2.13	457,12	2.23	–227
2.4	64	2.14	–188,2	2.24	85,350
2.5	127	2.15	78,87	2.25	–333
2.6	127,25	2.16	–65,125	2.26	–1024
2.7	127,63	2.17	–225,65	2.27	–1023
2.8	10,1	2.18	–256,6	2.28	1025
2.9	–54	2.19	511,3	2.29	65535
2.10	–65,375	2.20	–513,8	2.30	65536

3 Užrašyti dešimtaines reikšmes formatu ”ilgas realus”.

3.1	1023	3.8	5	3.15	87,375
3.2	1024	3.9	6	3.16	998
3.3	–1025	3.10	128	3.17	65536
3.4	511	3.11	–127	3.18	–65535
3.5	–512,22	3.12	–0,05	3.19	65,56
3.6	–513,6	3.13	–0,25	3.20	127,350
3.7	98,55	3.14	–0,01	3.21	-222
3.22	–56	3.25	–12,2	3.28	6,025
3.23	551	3.26	17,25	3.29	31,6
3.24	–12,1	3.27	–33,025	3.30	63,2

4 Pervesti skaičių iš formato ”trumpas realus” į dešimtainę sistemą.

4.1	41CA0000h	4.3	425C0000h	4.5	C2820000h
4.2	C0A00000h	4.4	42F00000h	4.6	C0C00000h

5 Pervesti skaičių iš formato ”ilgas realus” į dešimtainę sistemą.

5.1	4018000000000000h	5.6	402E000000000000h
5.2	C018000000000000h	5.7	4040000000000000h
5.3	401C000000000000h	5.8	C040800000000000h
5.4	C01C000000000000h	5.9	403F000000000000h
5.5	4020000000000000h		

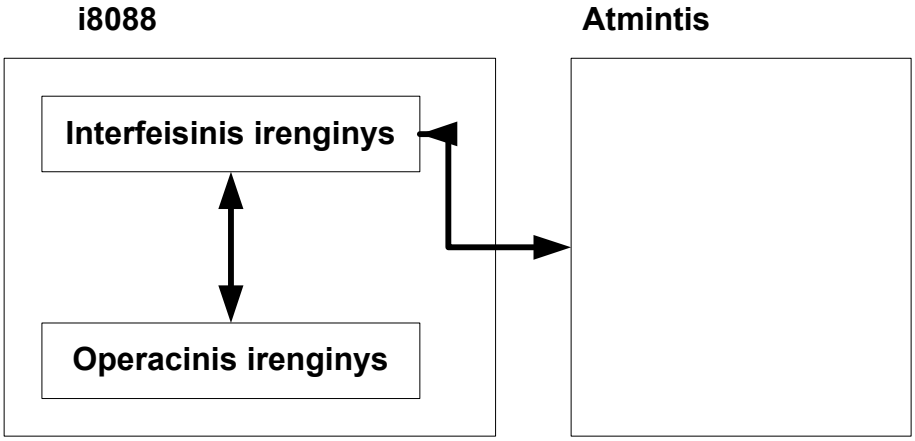
3. i8088 architektūra

Mikroprocesoriaus Intel 8088 architektūra yra su adresuojamais registrais. T.y. dalinama į dvi dalis – pagrindinę atmintį ir registrų atmintį (toliau sakysime tiesiog registrai). Pagrindinė atmintis turi 2^{20} ląstelių. Architektūroje naudojamas atminties segmentavimas (kitaip dar bazavimas, žr. 1.4). Mašinos žodis yra 8 bitų (atkreipkite dėmesį, kad mašinos žodis nėra tas pats, kas duomenų tipas žodis - word).

3.1. Vidinė mikroprocesoriaus struktūra

Mikroprocesorius Intel 8088 vykdomas programas turi 1)nuskaityti komandas iš atminties, 2) įvykdyti nuskaitytas komandas. Procesoriaus architektūra orientuota į šių 2 pagrindinių funkcijų vykdymą.

Bendrai procesoriaus i8088 struktūrinė schema (žr. 3.X) susideda iš dviejų įrenginių – operacinio įrenginio ir interfeisinio įrenginio. Operacinis įrenginys skirtas komandų dekodavimui ir vykdymui, interfeisinis įrenginys vykdo komandų nuskaitymą iš atminties ir kreipiasi į atmintį ir išorinius įrenginius.



3.1.1 pav. Struktūrinė i8088 schema

Operacinis įrenginys (žr. pav. 3.1.2) susideda iš aritmetinio loginio įrenginio, bendro naudojimo registrų ir požymių registro. Aritmetinis loginis įrenginys susideda iš 3 laikinų registrų operandų ir rezultato saugojimui, 16 bitų sumatoriaus. Duomenų apsikeitimas vyksta per bendro naudojimo registrus. Valdymas vyksta mikroprograminiame valdymo įrenginyje, kuris dekoduoja komandas ir generuoja valdymo signalus.

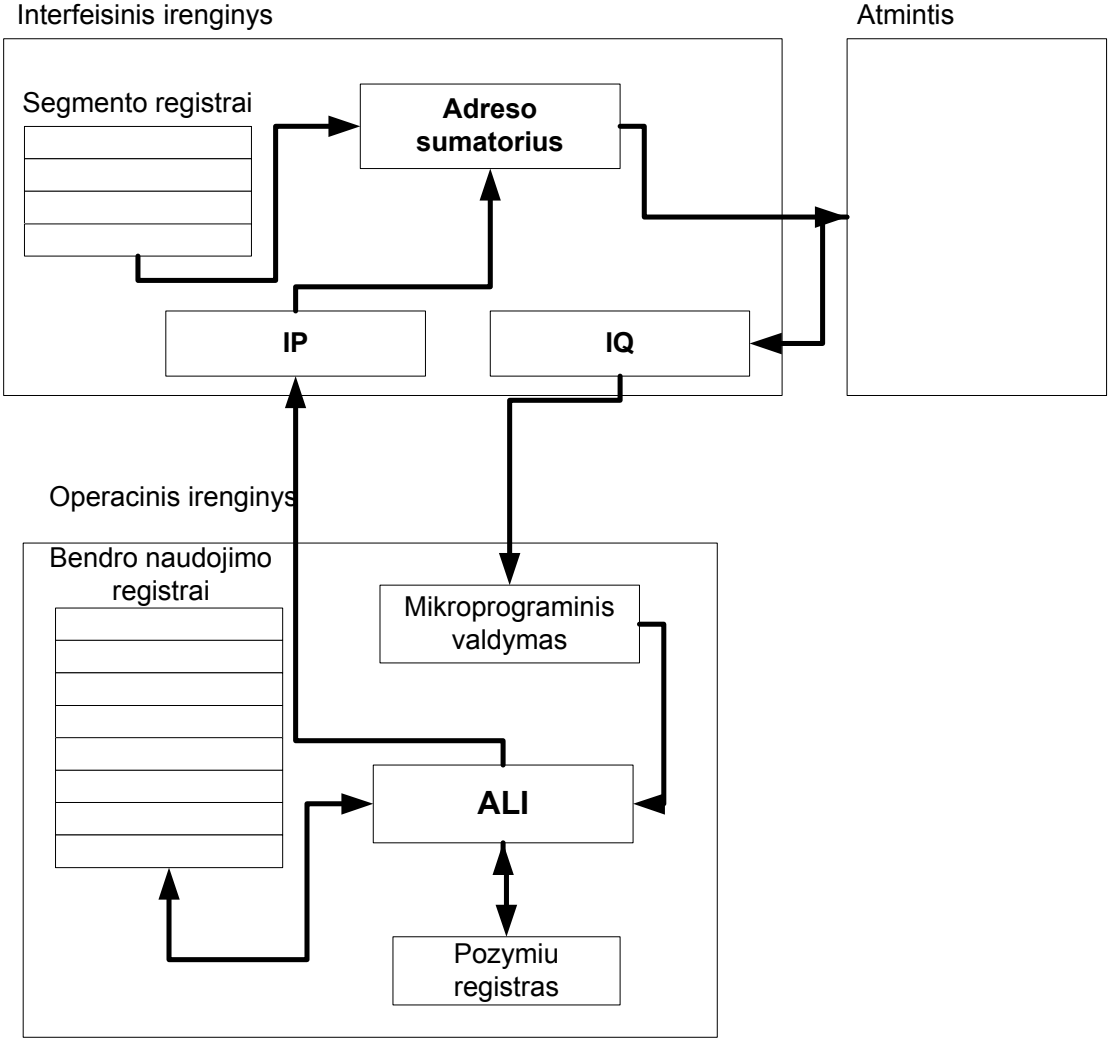
Operacinis įrenginys neturi priėjimo prie atminties ir išorinių įrenginių. OĮ su išore suriša interfeisinis įrenginys. IĮ susideda iš absoliutaus adreso formavimo aparato, segmento registrų bloko, komandų rodiklio, adresų sumatoriaus ir išrinkimo eilės.

Kai OĮ vykdo komandą, IĮ lygiagrečiai vykdo komandų kodų išrinkimą iš atminties į išrinkimo eilę.

Išrinkimo eilė yra keturių baitų rinkinys, kuris organizuotas FIFO principu ir iš esmės yra komandų registras, kuriame saugomi komandos iš atminties.

Išrinkimo ir vykdymo funkcijų priskyrimas skirtingiems įrenginiams leidžia procesoriui dirbti lygiagrečiai, padidina darbo našumą ir magistralės apkrovimą.

Segmentiniai registrai saugo bazinius atminties segmentų adresus. CS – kodo segmento, kuriame yra programos kodas, adresą. DS – duomenų segmento, SS – steko segmento, ES – papildomo duomenų segmento. Segmentiniai registrai yra būtini, nes architektūroje naudojama atminties segmentacija. Adresų sumatorius apskaičiuoja 20 bitų fizinius adresus. Komandų rodiklis IP saugo sekančios komandos poslinkį (efektyvų adresą) einamajame kodo segmente, t.y. rodo į sekančią komandą atmintyje.

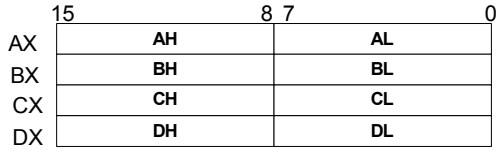


3.1.2 pav. Vidinė i8088 struktūra

Procesoriaus registrai sugrupuoti į tris kategorijas – duomenų, adresiniai ir segmento registrai. Kol kas mes naudosime registrų mnemonikas, vėliau parodysime, kaip registrai koduojami komandose.

Procesoriuje yra keturi **duomenų registrai** – AX, BX, CX ir DX. Tokios registrų mnemonikos kilo iš procesoriaus Intel 8080, kur buvo keturi 8 bitų registrai A (*accumulator* – akumuliatorius), B (*base* – bazė), C (*counter* – skaitliukas), D (*data* - duomenys). Akumuliatorius buvo naudojamas sudėties arba atimties rezultatui saugoti, bazė – duomenų atmintyje pradiniam adresui (bazei) saugoti, skaitliukas – ciklo skaitliukas, nurodantys ciklo iteracijų skaičių, duomenys – saugoti duomenims. Intel 8088 duomenų registrai yra 16 bitų, todėl jie buvo pavadinti A eXtended, B eXtended, C eXtended, D eXtended. Registrai AX, BX, CX ir DX išsaugojo prasminius pavadinimus, bet niekas netrukdo saugoti sudėties rezultatą registre BX, arba duomenis registre CX. Bet yra tam tikros procesoriaus komandos, kurios reikalauja reikšmių tam tikruose registruose. Pagal nutylėjimą laikysime, kad komandoje galima naudoti bet kokį duomenų registrą, o kai reikės – nurodysime kokio registro arba registrų reikalauja komanda.

Kiekvienas duomenų registras logiškai padalintas į dvi dalis – vyriausią ir mažiausią. Tai pavaizduota pav. 3.1.3:



3.1.3 pav. Duomenų registrai

Registrų AX,BX, CX, DX bitai numeruojami iš dešinės nuo 0 iki 15. Registrai AL, BL, CL, DL užima atitinkamai registrų AX, BX, CX, DX bitus nuo 0 iki 7, o AH, BH, CH, DH nuo 8 iki 15. Tai leidžia adresuoti vyriausią ir mažiausią duomenų registro baitus. Įrašius reikšmę į vieno baito registrą, keičiama viso registro reikšmė, pvz., jei registro AX reikšmė yra 5466h, tai įrašius reikšmę 70h į AH, registro AX reikšmė taps 7066h. Apsikeitimas galimas tik tarp vieno dydžio registrų, pvz., AX ir BX, AH ir BL, bet ne DX ir CL.

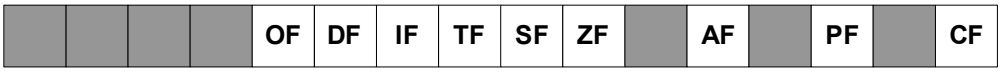
Adresiniai registrai susideda iš dviejų pogrupių – **indeksiniai registrai** ir **nuorodų registrai** (registrai-rodyklės). Indeksiniai registrai SI (*Source Index* – šaltinio indeksas) ir DI (*Destination Index* – gavėjo indeksas) yra 16 bitų, nėra

galimybės adresuoti vyriausią ir mažiausią registrų baitus. Komandos darbui su eilutėmis reikalauja indeksinių registrų, taip indeksiniai registrai naudojami vykdomojo adresų formavimui (žr. poskyrį 3.2).

Registrai-rodyklės SP (Stack Pointer – steko rodyklė) ir BP (Base Pointer – bazės rodyklė) naudojami darbui su steku, registrai yra 16 bitų. Steko rodyklė SP visada saugo steko viršūnės poslinkį steko segmente. Bazės rodyklė BP naudojama operandams steke (ne steko viršūnėje) adresuoti.

Segmentiniai registrai CS, DS, SS, ES naudojami atminties segmentavimui ir saugo segmento pradžią. Segmento registrai yra 16 bitų. Registras CS (*Code Segment* – kodo segmentas) nurodo segmento, kuriame saugoma programa, numerį. Segmento numeris vadinamas paragrafu, t.y. segmento pradžios adresas padalintas iš 16. Registras DS (*Data Segment* – duomenų segmentas) nurodo segmento, kuriame yra programos duomenys, paragrafą. Registras SS (*Stack Segment* – steko segmentas) skirtas steko segmento paragrafui saugoti. Registras ES (Extra Segment – papildomas segmentas) dažniausia naudojamas komandose darbui su eilutėmis.

Taip pat procesoriuje yra du programiškai neadresuojami registrai – komandų skaitliukas ir požymių registras. **Komandų skaitliukas** IP (*Instruction Pointer*) kartu su segmento registru CS naudojamas sekančios komandos absoliutaus adreso formavimui. IP yra sekančios vykdomos komandos poslinkis kodo segmente, registras yra 16 bitų. Požymių registras FLAGS (dar žymimas SF – *Status Flag*) taip pat yra 16 bitų, bet jame naudojami tik kai kurie bitai, kiekvienas iš jų turi savo prasmę:



1 pav. Požymių registras

- Požymis CF (*Carry Flag*) naudojamas pernešimui, nustatomas į 1, jeigu po paskutinės loginės aritmetinės operacijos vykdymo atsirado rezultato pernešimas (žr. 2.1, [Mit03, p.60]);
- PF (*Parity Flag*) – lyginimo požymis, nustatomas į 1, kai vienetinių rezultato bitų skaičius yra lyginis;
- AF (*Auxiliary Flag*) - papildomo pernešimo požymis, naudojamas dešimtainio formato skaičiams, nustatomas į 1, kai įvyksta pernešimas iš jaunesniojo pusbaičio į vyresnįjį (žr. [Mit03, pp. 60-61]);

- ZF (*Zero Flag*) – nulio požymis, nustatomas į 1, kai visi rezultato bitai yra nuliniai;
- SF (*Sign Flag*) – ženklo požymis, naudojamas skaičiams su ženklu, nustatomas į 1, kai ženklo bitas yra 1;
- TF (*Trap Flag*) – ”spąstų” požymis, naudojamas pertraukimų apdorojimo mechanizme (plačiau apie pertraukimų apdorojimo mechanizmą žr. 3.6);
- IF (*Interrupt Flag*) – pertraukimo požymis, jeigu reikšmė yra 1 – leidžiami maskuojami pertraukimai (plačiau žr. 3.6);
- DF (*Direction Flag*) – krypties požymis, naudojamas komandose darbui su eilutėmis, kai reikšmė yra 0, simbolių eilutė apdorojama iš kairės į dešinę, kai reikšmė yra 1 - atvirkščiai;
- OF (*Overflow Flag*) – perpildymo požymis, naudojamas perpildymui pažymėti, nustatomas į 1, jeigu po paskutinė aritmetinės komandos vykdymo rezultate įvyko perpildymas.

Pratimai

1 Nurodykite atskirų požymių reikšmes, kai duota požymių registro reikšmė

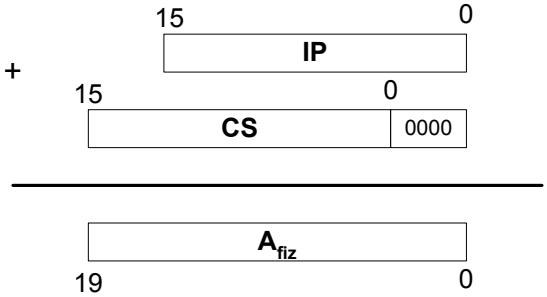
1.1	0FFA3h	1.6	0BBBBh	1.11	0B8C9h
1.2	000Fh	1.7	0CCCCh	1.12	7C7Fh
1.3	0FA38h	1.8	148Bh	1.13	1819h
1.4	0F00Fh	1.9	FA81h	1.14	3257h
1.5	0AAAAh	1.10	3A4Bh	1.15	0DE85h

3.2. Atminties segmentacija

Atmintis – baitų masyvas, kur kiekvienas baitas turi 20 bitų adresą, nuo 00000h iki 0FFFFFh. Bet kurie du šalia esantys baitai gali būti traktuojami kaip vienas 16 bitų žodis. Mūsų architektūroje naudojama duomenų atmintyje saugojimo schema Little endian. T.y. jaunesnysis baitas turi mažesni adresą. Pavyzdžiui, jeigu mes įrašysime žodį 1234h adresu 12450h, tai baito 12h adresas bus 12451h, o baito 34h adresas – 12450h. Kai kuriuose architektūrose naudojama schema Big Endian, o SPARC V9 architektūros procesoriai palaiko abi schemas. Žodžio adresas yra žodžio jaunesniojo baito adresas. Fizinis 20 bitų adresas gali būti traktuojamas kaip baito, arba kaip 16 bitų žodžio adresas.

Intel 8088 architektūros kompiuteriuose atmintis logiškai suskirstyta į paragrafus. Segmento dydis yra 64K, paragrafo dydis yra 16 baitų, poslinkis segmente yra 16 bitų (taip pat 64K). Absoliutus adresas formuojamas pagal vykdomąjį adresą (efektyvų

adresą) ir vieną iš segmento registrų. Absoliutus adresas yra 20 bitų (tiek yra atminties ląstelių), o mūsų registrai 16 bitų. Todėl segmento registras dauginamas iš 16 (tas pats, kas pastumti reikšmę per 4 bitus į kairę) ir sudedamas su efektyviu adresu. Bendra formulė A_{abs}=(Seg*16+EA) mod 2^{20}, kur Seg – segmento paragrafas, EA – efektyvus adresas. Pavyzdžiui, pavaizduokime kaip apskaičiuojamas absoliutus sekančios vykdomos komandos (iš tikrųjų ne sekančios vykdomos, o sekančios skaitomos iš atminties, žr. [Mit03, p. 39])) adresas. Kaip jau žinome, vykdomos komandos adresas nusakomas dviem registrais – CS ir IP. CS saugo segmento paragrafą, kuriame yra vykdomą programą, o IP saugo sekančios komandos poslinkį kodo segmente:



3.2.1 pav. Absoliutaus adreso formavimas

Analogiška schema egzistuoja ir bet kokiam kitam absoliučiam adresui suformuoti, tik naudojami kiti segmento registrai ir efektyvūs adresai, pvz., steko viršūnė nusakoma pora SS:SP. Dažniausia absoliutų adresą žymėsime pora ”Segmento paragrafo numeris:Poslinkis”.

Operandų atmintyje absoliutus adresas nusakomas pora DS:Efektyvus adresas, bet čia yra kelios išimtys. Laikoma, kad operandas, adresuojamas registru BP, yra steko segmente. Komandų darbui su eilutėmis rezultato absoliutus adresas formuojamas pagal porą ES:DI. Lentelėje 3.2.2 pateiktos taisyklės, kuriomis vadovaujasi interfeisinis įrenginys, kai formuoja absoliutų adresą.

Kreipimosi į atminties tipas	Segmentas pagal nutylėjimą	Variantai	Efektyvus adresas
Komandos išrinkimas	CS	-	IP
Steko operacija	SS	-	SP
Kintamasis	DS	CS,SS,ES	EA
[BP]	SS	CS,DS,ES	EA

3.2.2 pav. Taisyklės segmentinių registrų naudojimui

Pakomentuokime šitas taisykles – komandos išrinkimui naudojama pora CS:IP. Šita pora nusako sekančios išrenkamos iš atminties komandos fizinį adresą. Variantų šiam veiksmui nėra. Steko viršūnės absoliutus adresas nusakomas registrų pora SS:SP, čia taip nėra variantų. Duomenų nuskaitymas iš atminties arba įrašymas į atmintį pagal nutylėjimą reiškia segmentinio registro DS panaudojimą, efektyvus adresas nusakomas pačioje komandoje. Šitam veiksmui gali panaudoti ir kitus segmentinius registrus – CS,SS arba ES. Kai adresuojamas operandas atmintyje ir efektyviam adresui apskaičiuoti naudojamas registras BP bet kokioje kombinacijoje, pagal nutylėjimą naudojamas segmentinis registras SS. T.y. registras BP skirtas duomenims steke adresuoti (ne steko viršūnėje).

Vykdomas operando atmintyje adresas formuojamas pagal komandoje esantį operandą. Operando atmintyje efektyvus adresas komandoje gali būti užrašytas pagal tokius šablonus – [A], [A+IR], [A+RR], [A+IR+RR], kur IR yra bet koks indeksinis registras, RR – BP arba BX registras (baziniai registrai), A – 8 arba 16 bitų reikšmė. Tuo atveju, kai A užima 8 bitus, reikšmė traktuojama kaip skaičius su ženklu. Formulė efektyviam adresui apskaičiuoti yra $EA=(A+RR+IR^4) \bmod 2^{16}$. Pavyzdžiui, komandoje operandas gali būti nusakomas tokiomis kombinacijomis – [3030h], [1105h+SI], [45A3h+BP], [55A0h+DI+BX]. O tokios kombinacijos nėra leistinos – [SI+DI], [BP+BX], [SP], [BL+SI]. Apskaičiuokime absoliutų operando adresą aukščiau pateiktoms kombinacijų pavyzdžiams. Tarkime, registro DS reikšmė yra 1200h, registro SS 1400h, SI=4545h, BP=2020h, DI=8FA7h, BX=0BD0h. Kai operandas komandoje yra užrašytas kaip [3030h], absoliutus adresas formuojamas pagal duomenų segmentą – DS*16+poslinkis=1200h*10h+3030h=15030h. Absoliutus adresas pagal [1105h+SI] – DS*16+efektyvus adresas=(1200h*10h)+(1105h+4545h)=1764Ah. Absoliučiam adresui pagal [45A3h+BP] apskaičiuoti reikia naudoti segmentinį registrą SS (nes registras adresuoja operandus steko segmente). A_{abs}=SS*16+(45A3h+BP), A_{abs}=(1400h*10h)+(45A3h+2020h), A_{abs}=1A5C3h. Absoliutus adresas pagal [55A0h+DI+BX] apskaičiuojamas, naudojant segmento registrą DS, A_{abs}=21117h.

[4] Sudėtyje dalyvauja tik komandoje nurodyti operandai, pvz., jei komandoje yra šablonas [A+IR], tai ir formulėje būtų $(A+IR) \bmod 2^{16}$

Pratimai

1 Apskaičiuokite operando absoliutų adresą, kai žinomas segmento paragrafo numeris ir poslinkis

1.1	4578h:0020h	1.6	000Fh:0AB0h	1.11	6EACh:6600h
1.2	0F0F:0FFF0h	1.7	5555h:01ABh	1.12	0AFFFh:0FFEAh
1.3	6000h:7000h	1.8	99EAh:66FFh	1.13	0ECA9h:0F009h
1.4	98FAh:0FFFEh	1.9	3127h:0EE0Ah	1.14	4512h:6658h
1.5	0000h:FAB0h	1.10	3200h:44F0h	1.15	8000h:08FCh

2 Pagal absoliutų adresą nustatykite bent dvi galimas poras Segmento paragrafas:poslinkis

2.1	01000h	2.6	0FFFFFh	2.11	21EFDh
2.2	45112h	2.7	65432h	2.12	DC560h
2.3	04545h	2.8	1324h	2.13	77F99h
2.4	0FFAEFh	2.9	55FACh	2.14	45052h
2.5	00000h	2.10	998E0h	2.15	44142h

3 Išbraukite neteisingas kombinacijas

3.1	[3245Eh]	3.6	[SS+SP]	3.11	[BP+DI]
3.2	[AX+BP]	3.7	[DI+04h]	3.12	[CS+IP]
3.3	[CX]	3.8	[BP+80h]	3.13	[DS+0EAh]
3.4	[DS]	3.9	[DX+BP]	3.14	[34h + SI]
3.5	[ES+DI]	3.10	[650h+SI+BX]	3.15	[228h+IP]

3.3. Komandos struktūrą

Bendru atveju komandos struktūrą galima užrašyti taip:

Prefix-OpCode-MODR/M-Offset-Operand,

kur *Prefix* yra komandos prefiksas, *OpCode* – operacijos kodas, *MODR/M* – adresavimo baitas, *Offset* – operando atmintyje poslinkis, *Operand* – betarpiškas komandos operandas. Komandos prefiksas yra vieno baito dydžio ir nusako segmento, naudojamo pagal nutylėjimą, keitimą, arba tai yra pasikartojimo prefiksas komandoms darbui su eilutėmis. Operacijos kodas yra vienintelis būtinas komandos struktūros elementas, jis nusako kokią komandą reikia atlikti procesoriui. Operacijos kodas yra vieno baito dydžio. Elementas *MODR/M* vadinamas adresavimo baitu. Pagal adresavimo baitą nustatoma, kur yra komandoje dalyvaujantys operandai, kaip apskaičiuoti operandų efektyvų adresą. Poslinkis gali būti vieno, dviejų, arba keturių baitų. Poslinkis naudojamas operando efektyviam adresui apskaičiuoti. Betarpiškas operandas yra operandas, kuris tiesiogiai dalyvauja komandos apibrėžtoje operacijoje. Betarpiškas operandas gali būti vieno arba dviejų baitų.

Operacijos kodas yra vienintelė privaloma komandos dalis. Visų kitų dalių komandoje būvimas priklauso nuo komandos formato. Pasakymas, kad operacijos kodas užima vieną baitą nėra visiškai tikslus. Tiksliau yra pasakyti, kad bet kuri komanda negali būti trumpesnė už vieną baitą. Bet operacijos kodas gali būti koduojamas ir baito dalimi, kur viena dalis vienareikšmiškai nusako operaciją, o kita dalis nusako komandos operandą (žr. [Mit03,p.42]). Mūsų nagrinėjamoje architektūroje gali būti dviadresinės, vienadresinės ir beadresinės komandos. Pavyzdžiui, beadresinės komandos INC AX mašininis kodas yra 40H, t.y. 01000000_2. Šiuo atveju operacijos kodas užima 5 bitus – 01000_2 (koduojamas baito dalimi), o likusieji 3 bitai – 000_2 – nusako registrą, kuris dalyvauja komandoje. Jeigu pakeisime vieną iš mažiausių bitų, pvz., 01000001_2 gausime komandą INC CX. O jeigu pakeisime viena iš vyriausių bitų (operacijos kodą) – gausime kitą komandą, pvz., 01001000_2 yra DEC AX. Intel 8088 procesoriaus operacijų kodai pateikti mokymo priemonėje [Mit03, p.65-98].

Intel 8088 procesoriaus komandos gali adresuoti iki dviejų operandų. Dviadresinės komandos, kuriose nenaudojamas betarpiškas adresavimo būdas, yra simetrinės, t.y. operacijos rezultatas gali būti įrašytas į bet kurį komandoje dalyvaujantį operandą. Bet vienas iš operandų būtinai yra registras, nes architektūroje yra komandų formatai registras-registras, registras-atmintis ir atmintis-registras, bet nėra komandų formato atmintis-atmintis.

Schematiškai komandų su vienu ir dviem operandais formatai pateikti pav. 3.3.1.

a) OPC d w Mod Reg R/M DispL DispH
b) OPC s w Mod OPC R/M DispL DispH DataL DataH
c) OPC w Mod OPC R/M DispL DispH

3.3.1 pav. Komandų formatai

Pirmas komandos baitas susideda iš operacijos kodo OPC ir dviejų konfigūracinių bitų – krypties *d* ir žodžio *w*. Kai krypties bitas d=1, operandas arba operacijos rezultatas įrašomas į registrą, kuris nusakomas antro komandos baito lauku REG. Kai d=0, vykdomas duomenų paėmimas iš nurodyto registro. Konfigūracinis bitas *w* apibrėžia operandų tipą: kai *w*=1, komanda dirba su žodžiais, kai *w*=0 – su baitais.

Antras komandos baitas yra adresavimo baitas. Adresavimo baitas nurodo komandoje dalyvaujančius registrus, arba registrą ir operandą atmintyje. Adresavimo baitas susideda iš trijų laukų MOD – režimas, REG - registras ir R/M –

registras/atmintis. Laukas MOD užima 2 bitus, REG ir R/M užima po 3 bitus. REG laukas nurodo registrą, kuris dalyvauja komandoje, R/M – registrą, arba operandą atmintyje. Čia reikia pabrėžti, kad laukas REG naudojamas tik dviejų operandų komandose, komandose su vienu operandu registras nurodomas lauku R/M, o laukas REG yra operacijos kodo išplėtimas. Laukas MOD nurodo, kaip reikia interpretuoti lauką R/M, apskaičiuojant vieno iš operandų adresą. Kai MOD=11, operandas yra registre, kitais atvejais operandas yra atmintyje. Panagrinėkime adresavimo baito dekodavimo lentelę:

	MOD				
R/M	00	01	10	11	
				w=1	*w*=0
000	BX+SI	BX+SI+DISP8	BX+SI+DISP16	AX	AL
001	BX+DI	BX+DI+DISP8	BX+DI+DISP16	CX	CL
010	BP+SI	BP+SI+DISP8	BP+SI+DISP16	DX	DL
011	BP+DI	BP+DI+DISP8	BP+DI+DISP16	BX	BL
100	SI	SI+DISP8	SI+DISP16	SP	AH
101	DI	DI+DISP8	DI+DISP16	BP	CH
110	DISP16	BP+DISP8	BP+DISP16	SI	DH
111	BX	BX+DISP8	BX+DISP16	DI	BH

3.3.2 Lentelė Adresavimo baito dekodavimas

DISP8 yra baitas, kurio reikšmę nurodo baitas DispL, DISP16 yra žodis, kurio reikšmę nurodo baitai DispH DispL (būtent tokia seka). Reikia pabrėžti, kad kai MOD yra 01, komandoje nurodytas baitas DispL interpretuojamas kaip skaičius su ženklu. Baitų DispL ir DispH nėra komandoje, kai MOD yra 11 ir 00 (išskyrus atvejį, kai laukas R/M yra 110, žr. lentelę 3.3.2). Kai laukas MOD yra 01 komandoje dalyvauja tik baitas DispL, kai MOD yra 10 komandoje dalyvauja ir DispL ir DispH.

Laukas REG dekoduojamas pagal tokią lentelę:

REG	*w*=0	*w*=1	REG	*w*=0	*w*=1
000	AL	AX	100	AH	SP
001	CL	CX	101	CH	BP
010	DL	DX	110	DH	SI
011	BL	BX	111	BH	DI

3.3.3 Lent. Adresavimo baito dekodavimas

Čia reikia pabrėžti, kad komandoje dalyvaujantis poslinkis – baitai DispL ir DispH – nėra poslinkis nuo segmento pradžios. Todėl susitarkime baitus DispL ir DispH vadinti poslinkiu, o poslinkį nuo segmento pradžios efektyviu adresu.

Dviadresinės komandos su betarpišku operandų formatas pateiktas pav. 3.3.1. b). Šiuo atveju mums nebereikia adresuoti vieno iš operandų (nes jis yra komandos sudėtinė dalis). Todėl laukas REG naudojamas operacijos kodui praplėsti. Taip pat nebėra krypties bito *d*, nes mes negalime padėti operacijos rezultatą į betarpišką operandą. Vietoj krypties bito atsiranda konfigūracinis bitas *s*, pagal kurį nusprendžiama, koks yra betarpiškas operandas:

sw	Betarpiškas operandas
X0	Vieno baito betarpiškas operandas, dataL
01	Dviejų baitų betarpiškas operandas, dataH dataL
11	Vieno baito operandas, kuris automatiškas išplečiamas iki žodžio, pagal ženklo plėtimo taisyklę

3.3.4 Lent. Konfigūracinių bitų SW dekodavimas

Komandos su vienu operandu formatas bendru atveju pateiktas pav. 3.3.1 c). Visus šitame formate dalyvaujančius laukus mes jau išnagrinėjome.

Reikia pastebėti kad adresavimas pagal adresavimo baitą yra universalus, nes leidžia adresuoti tiek bendrus registrus, tiek operandus atmintyje. Bet kai komandoje adresuojami tik registrai, adresavimo baitas gali būti nereikalingas, jeigu lauką REG laikyti pirmame komandos baite, t.y. kartu su operacijos kodu. Tokia galimybė realizuota specialiuose komandų formatuose, kuriuose yra minimaliai reikalingas baitų skaičius. Pavyzdžiui, operacijai INC architektūroje apibrėžti du formatai – įprastas ir specialus:

- 1111111*w* MOD 000 R/M DispL dispH – įprastas formatas, INC R/M;
- 01000reg – specialus formatas, INC REG

Iš viso Intel 8088 architektūroje galima išskirti devynis komandų formatus (bet tokai klasifikacija yra sąlyginė). Lentelėje 3.3.5 pateikti komandų formatai, *kursyvu* pažymėti specialūs (sutrumpinti) komandų formatai, simboliais b_i, kur $0 \le i \le 7$, žymėsime pirmo komandos baito (kuriame saugomas operacijos kodas) bitus. Reikia pastebėti, kad į lentelę neįtraukti dvi retai naudojamos komandos be operandų AAM ir AAD, jų operacijų kodai užima po du baitus.

Komandos formatas	Formato variantai	Komandų pavyzdžiai
OPC	$b_0 = w$ $b_2b_1b_0$=REG b_7-b_5b_2-b_0=OPC b_4b_3=SR	CLC;CLD;CLI;HLT; CMPS;LODS;IN pt,DX, INC reg;DEC reg; PUSH SR; POP SR;
OPC MOD REG R/M [disp L][disp H]	b_0=w b_1=d, b_0=w	LEA; LDS;LES TEST r,r; TEST r,m; MOV r,r; ADD r,m;
OPC MOD OPC R/M [disp L][disp H]	b_0=w b_1=v b_0=w	PUSH r; PUSH m; JMP [r]; JMP [m] INC m; DEC m; NEG; NOT; MUL; RCL;ROR;SAL;SHR;
OPC MOD OPC R/M [disp L][disp H] [data L] [data H]	b_0=w	MOV m,d; AND r,d; TEST m,d;
OPC disp L [disp H]	b_0=w	JMP disp16;JMP disp8;Jcond disp8 MOV acc,m; MOV m,acc;

OPC data L [data H]	b_0=w
	b_3=w b_2-b_0=reg
		RET data
		ADD acc,data; CMP acc,data;
		MOV reg,data;

OPC MOD 0SR R/M [disp L][disp H]		MOV sr, m; MOV r,sr;
OPC addr8	b_0=w	IN acc,addr8; OUT acc,addr8;
OPC offs L offs H seg L seg H		JMP addr; CALL addr;

3.3.5 Lent. Intel 8088 komandų formatai

Panagrinėkime mašininių kodų dekodavimo pavyzdžius. Tarkime turime duomenų persiuntimo komandą MOV. Komandos mašininis kodas yra 100010*dw* MOD REG R/M. Pažiūrėkime, kaip mašininiais kodais atrodys komanda MOV [BP+DI+03], CX. Pirma, matome kad kryptis yra iš registro į atmintį, reiškia krypties bito *d* reikšmė yra 0. Dirbama su žodžiais (žodinis registras CX), reiškia operandų tipo bitas *w* yra 1. Taigi, pirmas komandos baitas yra 10001001_2. Pagal pirmą komandos operandą galime nustatyti adresavimo baito reikšmę – MOD yra 01, REG yra 001, R/M yra 011, reiškia antras komandos baitas yra 01001011_2. Paskutinis komandos baitas yra poslinkis 3. Taigi visa komanda mašininiais kodais yra: 10001001 01001011 00000011_2, arba šešioliktainėje sistemoje 89 4B 03h.

Pateiksime komandų pavyzdžius visuose 9 formatuose. Pirmo formato komandos susideda iš vieno baito, kurio visi bitai gali koduoti operaciją, arba tik bitų dalis koduoja operaciją, o kita dalis nusako operandą. Pvz., komanda INC reg (registro reikšmės inkrementas), kur reg yra vienas iš 16 bitų bendro naudojimo registrų. Mašininiais kodais komanda atrodo taip – $10000b_2b_1b_0$, kur bitai $b_2b_1b_0$ nusako registrą, pvz., INC CX mašininiais kodais yra 10000001_2.

Antro formato komandos turi adresavimo baitą. Taip pat komandoje gali būti ir vieno arba dviejų baitų poslinkis, priklausomai nuo adresavimo baito lauko MOD.

Pavyzdžiui, paimkime komandą AND [BX+3],AL. Komandos AND tokio formato mašininis kodas yra 001000dw_2, kur *d* ir *w* yra krypties ir žodžio bitai atitinkamai. Užrašykime pilną komandos mašininį kodą – bitas *d* yra 0 nes kryptis ”iš registro”, bitas w yra 0, nes dirbama su baitas (registras AL), reiškia pirmas komandos baitas yra 00100000_2. Po pirmo baito eina adresavimo baitas, kuris susideda iš laukų MOD, REG ir R/M. Mūsų komandoje dalyvauja poslinkis, kuris telpa viename baite, reiškia lauko MOD reikšmė yra 01_2, lauko REG reikšmė yra 000_2 (registras AL), lauko R/M reikšmė yra 111_2 (BX+poslinkis), t.y. adresavimo baitas yra 01000111_2. Po adresavimo baito eina vieno arba dviejų baitų poslinkis. Mūsų atveju yra vieno baito poslinkis – 00000011_2. Visa komanda mašininiais kodais yra 00100000 01000111 00000011_2, arba šešioliktainėje sistemoje 20h 47h 03h.

Trečio formato komandos susideda iš operacijos kodo, adresavimo baito ir nebūtino poslinkio. Bet adresavimo baito laukas REG naudojamas kaip operacijos kodas praplėtimas, o nenusako registrą. Kaip pavyzdį paimkime komandą PUSH [BP+SI]. Komandos mašininis kodas yra 11111111 MOD 110 R/M. Mūsų atveju lauko MOD reikšmė yra 00 (nėra poslinkio). Lauko R/M reikšmė yra 010. T.y. komandos mašininis kodas yra 11111111_2 00110010_2, arba šešioliktainėje sistemoje FF32h.

Ketvirto formato komandos susideda iš operacijos kodo, adresavimo baito, kuriame laukas REG yra operacijos kodas praplėtimas, nebūtino poslinkio ir betarpiško operando. Pavyzdžiui paimkime komanda MOV [SI],FF45h. Komandos MOV tokiame formate mašininis kodas yra 1100011*w* MOD 000 R/M. Žodžio bitas *w* šiuo atveju yra 1 (dirbame su žodžiu FF45h), lauko MOD reikšmė yra 00, lauko R/M reikšmė yra 100. Taigi pirmi du komandos baitai yra 11000111_2 00000100_2. Lauko MOD reikšmė 00, reiškia komandoje nėra poslinkio ir po adresavimo baito eina betarpiškas operandas. Pirmas eina mažiausias baitas, t.y. 45h 0FFh. Komanda šešioliktainėje sistemoje atrodo taip C7h 04h 45h 0FFh.

Penkto formato komandos susideda iš operacijos kodo ir betarpiško operando viename arba dviejuose baituose. Kaip pavyzdį paimkime sudėties komanda akumuliatoriui – ADD AX, 0045h. Komandos mašininis kodas yra 0000010*w* data L [data H]. Bito *w* reikšmė yra 1. Po operacijos kodo seka mažiausias betarpiško operando baitas. T.y. komanda mašininiais kodais atrodo taip – 00000101_2 01000101_2 00000000_2, arba šešioliktainėje sistemoje 054500h.

Šešto formato komandos susideda iš operacijos kodo ir poslinkio. Poslinkis yra būtinas. Kaip pavyzdį, paimkime komandą MOV [1234h],AX. Operacijos kodas yra 1010001*w*. Mūsų atveju žodžio bito reikšmė yra 1. Po operacijos kodo eina poslinkis. Užrašykime visą komandą – 10100011_2 00110100_2 00010010_2, arba šešioliktainėje sistemoje A33412h.

Septinto formato komandos skirtos darbui su segmento registrais. Komandos susideda iš operacijos kodo, adresavimo baito, kuriame lauko REG vyriausias bitas yra 0, o likę bitai nusako segmento registrą, ir nebūtino poslinkio. Kaip pavyzdį paimkime komandą MOV DS,[BX+DI+1254h]. Operacijos mašininis kodas yra 10001110_2. Po operacijos kodo seka adresavimo baitas – lauko MOD reikšmė yra 10 (dviejų baitų poslinkis), lauko REG reikšmė yra 011, kur 11 nusako registrą DS, lauko R/M reikšmė yra 001. Po adresavimo baito seka poslinkio mažiausias ir vyriausias baitai 54h ir 12h atitinkamai. Visa komanda atrodo taip – 10001110_2 10011001_2 01010100_2 00010010_2, arba šešioliktainėje sistemoje 8E995412h.

Aštunto formato komandos dirba su portais. Komanda susideda iš operacijos kodo ir porto numerio. Kaip pavyzdį paimkime komandą IN AL, 70h. Komandos operacijos kodas yra 1110010*w*. Mūsų atveju bitas *w* yra 0. Visa komanda atrodo taip – 11100100_2 01110000_2, arba šešioliktainėje sistemoje E470h.

Devinto formato komandos yra valdymo perdavimo komandos, kurios naudoja absoliutų adresavimą. Komandos susideda iš operacijos kodo ir tiesioginio dviejų žodžių adreso. Pavyzdžiui paimkime komandą JMP 0120h:4568h. Operacijos kodas yra 11101010_2. Visa komanda atrodo taip – 11101010_2 01101000_2 01000101_2 00100000_2 00000001_2, arba šešioliktainėje sistemoje EA68452001h.

Prieš operacijos kodą gali būti nurodytas prefiksas. Intel 8088 architektūroje yra apibrėžti kartojimo, magistralės blokavimo ir segmento keitimo prefiksai. Panagrinėkime segmento keitimo prefiksą. Prefiksas yra 8 bitų dydžio ir dvejetainėje sistemoje atrodo taip 001SR110, kur bitai SR nusako segmentinį registrą: 00-ES, 01-CS, 10-SS, 11-DS. Segmento keitimo prefiksas turi įtakos vykdomos komandos operando atmintyje absoliutaus adreso skaičiavimui. Kai nurodytas segmento keitimas, absoliutus adresas apskaičiuojamas pagal prefikse nurodytą registrą. Pavyzdžiui, tarkime turime komandą 26h OPC 00h, kur 26h yra segmento keitimo prefiksas, OPC – operacijos kodas ir 00h – adresavimo baitas. Pagal adresavimo baitą galima pasakyti, kad operando atmintyje absoliutus adresas apskaičiuojamas pagal

formulę DS*16+[BX+SI]. Bet prieš operacijos kodą stovi segmento keitimo prefiksas 26h, t.y. 00100110_2. SR yra 00, kas reiškia segmentinį registrą ES, absoliutus operando adresas bus apskaičiuojamas pagal formulę ES*16+[BX+SI]. Asemblerio kalboje segmento keitimo prefiksas užrašomas prie R/M operando, pvz.:

ADD CX,ES:[SI]; MOV CS:[BX+DI+22h],1234h

Pratimai

1 Duotai komandai nurodykite adresavimo baitą ir baitus kurie eina po jo (operacijos kodo nurodyti nereikia)

1.1 MOV AX,BX
1.2 MOV DX,DX
1.3 MOV BL,[SI+BX]
1.4 MOV [BX+03h],SI
1.5 MOV [BP+02H],DL
1.6 MOV [SI+BX+33ABH],BH
1.7 MOV SP,[3AB0H]
1.8 MOV [BP+DI+3353H]
1.9 MOV CX,[SI+33H]
1.10 MOV DI,[BP+1600H]
1.11 MOV BX,[BP+DI+32H]
1.12 MOV [SI],BX
1.13 MOV DX,[DI+0FF0AH]
1.14 MOV [DI+BX+33AFH],SP
1.15 MOV AL,[BP+127]
1.16 MOV [BP+SI+128],CH
1.17 MOV [SI+138],BP
1.18 MOV [DI-128],SP
1.19 MOV [SI+BX-129],BX
1.20 MOV [BP+255],DH

2 Duota 3 baitų seka, kur pirmas baitas – adresavimo baitas. Komandos mnemonika MOV REG⇔R/M. Užrašykite pilną komandą.

2.1 *d=1,w=0*, 73h 04h 55h.
2.2 *d=1,w=1*, 0B3h 0FFh 32h.
2.3 *d=1,w=1*, 01h 33h 7Eh.
2.4 *d=1,w=0*, 38h 38h 38h.
2.5 *d=1,w=0*, 1Bh 0FAh 0AFh.
2.6 *d=1,w=0*, 51h 33h 4Fh.
2.7 *d=1,w=1*, 0AEh 0ABh 33h.
2.8 *d=0,w=1*, 20h 45h 0F0h.
2.9 *d=0,w=1*, 71h 0FEh 0A3h.
2.10 *d=0,w=1*, 92h 0FFh 33h.
2.11 *d=0,w=1*, 0ABh 0FFh 0F3h.
2.12 *d=0,w=0*, 0B0h 0FFh 0FFh.
2.13 *d=1,w=0*, 23h 43h 33h.
2.14 *d=0,w=0*, 0Dh 0Ah 3Fh

3 Duota persiuntimo komanda MOV REG⇔R/M. Pagal pateiktą komandos mašininį kodą apskaičiuokite operando R/M fizinį adresą. Registrų reikšmės yra CS=3FAEh, DS=0EAFDh, SS=12FFh, ES=87Fah, BX=0FF4Dh, BP=0010h, SI=0F00h, DI=0F000h. Komandos MOV šitame formate operacijos kodas yra 100010dw.

3.1 8Ah 56h 7Fh 03h
3.2 89h 60h 0FCh 33h
3.3 89h 0Fh 5Fh 7Eh
3.4 89h 0BDh 0Fh 0F0h
3.5 26h 8Ah 26h 75h 12h
3.6 2Eh 89h 0Eh 0Eh 0FFh
3.7 3Eh 89h 7Eh 00h
3.8 88h 37h 5Ah 7Ch
3.9 26h 88h 9Ch 81h 00h
3.10 8Ah 60h 0E0h 0Fh
3.11 8Ah 47h 0FEh 0Fh
3.12 26h 88h 26h 21h 12h
3.13 36h 89h 26h 01h 00h
3.14 8Bh 0Dh 77h 45h
3.15 2Eh 88h 23h 4Fh
3.16 8Ah 4Eh 00h 44h
3.17 89h 93h 3Ah 0FFh
3.18 36h 8Ah 75h 0FFh
3.19 88h 0AAh 12h 0FFh
3.20 89h 6Eh 00h 05h

4 Persiuntimo komandos MOV formato REG⇔R/M mašininis kodas yra 100010dw. Užrašykite duotą asemblerinę komandą mašininiais kodais.

4.1 MOV AX, [BX]
4.2 MOV BX, [BP]
4.3 MOV CL, [SI]
4.4 MOV [SI+BP],DX
4.5 MOV ES:[BP],CH
4.6 MOV AX,SS:[DI+37]
4.7 MOV DL,[BP+DI]
4.8 MOV CS:[DI],DH
4.9 MOV [SI+BP+33FEH],BH
4.10 MOV DS:[BX],BX
4.11 MOV SS:[123H],SP
4.12 MOV BP,[BP]
4.13 MOV CS:[BX-2],DL
4.14 MOV AH,[BP+129]
4.15 MOV CL,SS:[BP+SI-129]
4.16 MOV DX,CS:[BP]
4.17 MOV AL,[BX-12H]
4.18 MOV SS:[BX],BP
4.19 MOV [02H],BX
4.20 MOV ES:[-12],AL

3.4. Adresavimo būdai

Adresavimo būdo sąvoka suprantama kaip taisyklės, pagal kurias reikšmės, esančios komandoje, identifikuoja komandos operandą. Mūsų nagrinėjamoje architektūroje priimti 3 adresavimo būdai – tiesioginis, betarpiškas ir netiesioginis.

Tiesioginio adresavimo atveju, reikšmė, esanti komandoje, traktuojama kaip operando efektyvus adresas. Tiesioginis adresavimo būdas yra tokių rūšių [Mit03,p.43]:

1) Pagal adresavimo baitą;
2) Pagal operacijos kodą;
   a) paprasta tiesioginė;
   b) santykinė tiesioginė;
   c) absoliuti tiesioginė.

Pateiksime tiesioginio adresavimo pavyzdžius. Tiesioginis adresavimas pagal adresavimo baitą, pavyzdžiui, naudojamas komandoje su mašininiais kodais 8B1Bh, kurios mnemonika yra MOV BX,[BP+DI]. Šiuo atveju, pagal adresavimo baitą 1Bh suformuojamas efektyvus adresas, ir reikšmė, esanti šiuo adresu, nusiunčiama į registrą BX.

Paprasto tiesioginio adresavimo būdo pavyzdžiui galima išnagrinėti komandą E90004h, kurios mnemonika yra jmp label. Šiuo atveju, po operacijos kodo E9h įrašytas tiesioginis adresas, kurio reikšmė yra 0004h. Tiesioginis adresas pridedamas prie registro IP.

Santykinis tiesioginis adresavimas naudojamas, pavyzdžiui, komandoje 7405h, kurios mnemonika yra je label. Šiuo atveju komandoje esantis baitas traktuojamas

kaip skaičius su ženklu intervale [-128..127]. Vykdomas adresas apskaičiuojamas pagal formulę IP+05h.

Tiesioginio absoliutaus adresavimo pavyzdžiu yra komanda EA23000000h, kurios mnemonika yra JMP 0000h:0023h. Šiuo atveju pagal reikšmę, esančią komandoje, suformuojamas absoliutus operando adresas.

Betarpiškas adresavimas nustatomas pagal operacijos kodą. Čia svarbu skirti betarpišką adresavimą nuo paprastos tiesioginės. Betarpiško adresavimo pavyzdžiui pateiksime komandą B84532h, kurios mnemonika yra MOV AX,3245h. Šiuo atveju komandoje nurodytas betarpiškas operandas, kuris nusiunčiamas į registrą AX.

Netiesioginis adresavimas yra toks adresavimo būdas, kai reikšmė, esanti komandoje traktuojama, kaip efektyvus adresas, kuriuo esanti reikšmė ir yra komandos operandas. Netiesioginis adresavimas taip nustatomas pagal operacijos kodą. Netiesioginės adresavimui pavaizduoti galime pateikti komandą FF 50 03h, kurios mnemonika yra CALL [BX+SI+03].

3.5. Stekas

Programuotojui Intel 8088 architektūroje yra prieinamas stekas. Stekas yra atminties segmentas, kurio pradžią nurodo segmentinis registras SS. Darbui su steku architektūroje yra dvi vienadresinės komandos PUSH ir POP. Komanda PUSH įrašo operandą į steko viršūnę, o komanda POP įrašo steko viršūnės reikšmę į komandoje nurodytą operandą. Steko viršūnės poslinkį nuo steko segmento pradžios nurodo registras-rodyklė SP. Steko viršūnės absoliutus adresas formuojamas pagal formulę $A_{abs}=(SS*16+SP) \bmod 2^{20}$.

Reikšmės į steką įrašomos iš apačios į viršų, t.y. adresų mažėjimo tvarka. Pakomentuokime tai detaliau. Komandos PUSH op1 vykdymo schema atrodo taip:

1. Steko viršūnės reikšmė sumažinama dvejetu, SP:=SP-2 (t.y. į steką galima įrašyti tik 2 baitų reikšmes).
2. Adresu SS:SP patalpinama operando op1 reikšmė.

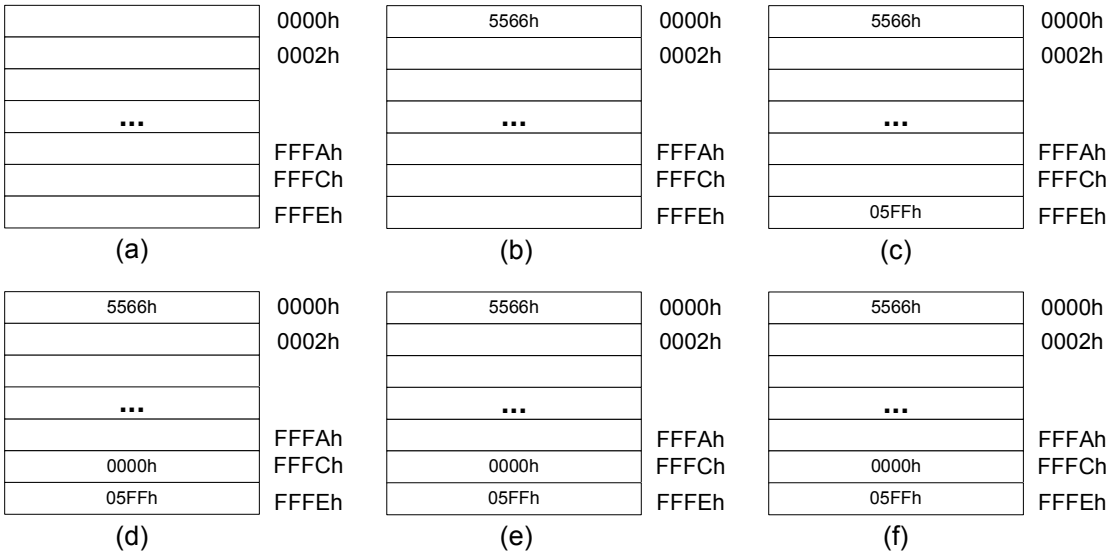
Komanda POP op1 vykdoma atvirkštine tvarka:

1. Į operandą op1 patalpinama reikšmė iš atminties adresu SS:SP.
2. Steko viršūnė padidinama dvejetu, SP:=SP+2.

Komandų PUSH ir POP operandas op1 gali būti arba vienas iš 16 bitų registrų, arba operandas atmintyje dvejuose baituose. Parodykime, kaip kinta stekas po programos fragmento:

```
PUSH    AX
PUSH    BX
PUSH    CX
POP     ES
POP     [BX+SI]
POP     AX
```

kai SS reikšmė yra 2240h, SP reikšmė prieš vykdant fragmentą yra 0002h, AX=5566h, BX=05FFh, CX=0000h, SI=0012h. Prieš vykdant fragmentą, stekas atrodo taip, kaip pavaizduota pav. 3.5.1 (a).



3.5.1 pav. Steko būsenos kitimas

Vykdant pirmą fragmento komandą, sumažinama SP reikšmė SP:=SP-2, ir adresu SS:SP įrašoma operandas. Šiuo atveju operandas yra registras AX, t.y. adresu 2240h:0000h įrašoma reikšmė 5566h (žr. pav. 3.5.1(b)). Vykdant antrą komandą, sumažinama SP reikšmė SP:=SP-2 = 0FFFEh, ir adresu SS:0FFFEh įrašoma reikšmė 05FFh (žr. (c)). Po trečios fragmento komandos SP reikšmė tampa 0FFFCh, o adresu SS:0FFFCh įrašoma reikšmė 0000h (žr. 3.5.1(d)). Vykdant ketvirtą fragmento komandą, SP reikšmė padidinama dvejetu ir tampa 0FFFEh (žr. 3.5.1(e)). Registras ES įgyja reikšmę 0000h. Vykdant penktą komandą, SP reikšmė didinama dvejetu ir tampa 0000h. Šiuo atveju komandos operandas yra atmintyje. Operando vykdomas adresas yra registrų BX ir SI suma, o absoliutus adresas formuojamas, naudojant

segmentinį registrą DS, t.y. reikšmė 05FFh įrašoma adresu DS:0611h (žr.3.5.1(f)). Vykdant paskutinę komandą, SP įgyja reikšmę 0002h, o į registrą AX įrašoma reikšmė iš atminties adresu SS:0000h (žr. 3.5.1(g)).

Maksimalus steko dydis yra 64K, t.y. kai stekui išskirtas visas segmentas. Tokiu atveju kai stekas yra tuščias, SP reikšmė yra 0000h. Kai stekas yra pilnas, SP reikšmė taip pat yra 0000h. Tai yra dėl to, kad vykdomas adresas steko segmente (ir šiaip bet kokiame segmente) apskaičiuojamas moduliu 2^{16}. Tuo atveju, kai bandysime įrašyti reikšmę į pilną steką, mes sugadinsime prieš tai buvusias reikšmes.

Registras BP naudojamas steko operandams ne steko viršūnėje adresuoti. Jeigu efektyvaus adreso formavime dalyvauja registras BP ir nėra segmento keitimo prefikso, tai absoliučiam adresui suformuoti naudojamas segmentinis registras SS. Pavyzdžiui, registro SS reikšmė yra 0540h, registro BP reikšmė yra 0110h, tada komanda MOV AX, [BP+02] į registrą AX įrašo reikšmę, kurios absoliutus adresas yra 05400h+0112h=05512h.

Stekas aktyviai naudojamas paprogramių iškvietimo mechanizme. Kviečiant paprogramę, į steką įrašomas grįžimo adresas. T.y. į steką įrašomas registro IP reikšmė, arba registro CS ir registro IP reikšmės (žr. 4.7). Grįžimo iš paprogramės komanda interpretuoja reikšmes, esančias steko viršūnėje kaip adresą, kur reikia grąžinti valdymą. Taip pat stekas naudojamas pertraukimų apdorojimo mechanizme.

Aukšto lygio programavimo kalbos naudoja steką paprogramių parametrams perduoti, o taip pat ir paprogramių laikinų kintamųjų saugojimui. Prieš iškviečiant paprogramę, kviečianti programa į steką įrašo faktinius parametrus. Jeigu vyksta parametro perdavimas pagal reikšmę, į steką įrašomos parametrų reikšmės. Jeigu vyksta parametrų perdavimas pagal nuorodą, į steką įrašomi parametrų adresai. Lokalūs paprogramės kintamieji taip pat suformuojami steke. Steko dalis, kur saugomi paprogramės parametrai ir lokalūs kintamieji, vadinama steko kadru (*stack frame*). Pavaizduokime, kaip atrodo steko kadras ir kaip vyksta paprogramės iškvietimas C kalboje. Tarkime, turime tokią programą C kalba:

```
int max(int a, int b);                  (1)

int main()                              (2)
{
        int c = max(2,3);               (3)
        return 0;                       (4)
}
```

```
int max(int a, int b)                    (5)
{
        return (a>b)?a:b;                (6)
}
```

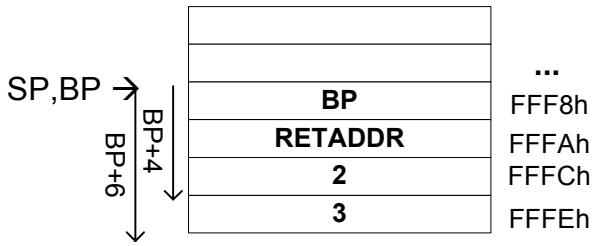
C kompiliatoriumi (Borland C 3.1) programą bus sutransliuota į tokį asemblerinį kodą:

```
_main   proc    near                     (1)
        push    bp                       (2)
        mov     bp,sp                    (3)
;
; {
;   int c = max(2,3);
;
        mov     ax,3                     (4)
        push    ax                       (5)
        mov     ax,2                     (6)
        push    ax                       (7)
        call    near ptr _max            (8)
        pop     cx                       (9)
        pop     cx                       (10)
        mov     word ptr [bp-2],ax       (11)
;
;   return 0;
;
        xor     ax,ax                    (10)
;
; }
;
        mov     sp,bp                    (11)
        pop     bp                       (12)
        ret                              (13)
_main   endp                             (14)
;
; int max(int a, int b)
;

_max    proc    near                     (15)
        push    bp                       (16)
        mov     bp,sp                    (17)
        mov     dx,word ptr [bp+4]       (18)
        mov     bx,word ptr [bp+6]       (19)
;
; {
;   return (a>b)?a:b;
;
        cmp     dx,bx                    (20)
        jle     short @2@86              (21)
        mov     ax,dx                    (22)
        jmp     short @2@114             (23)
@2@86:                                   (24)
        mov     ax,bx                    (25)
@2@114:                                  (26)
;
; }
;
        pop     bp                       (27)
```

```
    ret                         (28)
_max    endp                    (29)
```

C programoje yra dvi funkcijos – ***int*** *main()* ir ***int*** *max(**int**,**int**)*. Funkcija main turi vieną lokalų kintamąjį – ***int*** *c* (žr. (3) eilutę). Asemblerio programoje kintamasis suformuojamas main funkcijos steko kadre (žr. (11) eilutę). Taip pat matome, kaip suformuojamas funkcijos max steko kadras, eilutėse (4-7) ir (16-17). Pavaizduokime funkcijos *max* steko kadrą:



3.5.2 pav. Funkcijos MAX steko kadras

Pratimai

1 Pagal pateiktą programos fragmentą ir registrų reikšmes paaiškinti kaip kinta stekas. CS=1200h, DS=54FFh, SS=0998h, ES=0000h, AX=5545h, BX=0000h,CX=66FFh, DX=0EACh, SI=8800h, DI=7784h, BP=0000h, SP=0004h.

1.1
```
PUSH BP
MOV BP,SP
PUSH AX
PUSH CX
MOV AX,[BP-4]
POP  DX
POP  BX
```

1.2
```
PUSH BP
MOV  BP,SP
PUSH CX
MOV  SI,[BP-2]
POP  DI
POP  BP
```

1.3
```
PUSH  SP
PUSH  BP
POP   AX
PUSH  DX
POP   [BX+SI]
POP   BP
```

1.4
```
PUSH  AX
PUSH  DX
PUSH  CS
POP   DS
POP   SI
POP   DI
```

2 Pagal pateiktą fragmentą ir registrų reikšmes apskaičiuoti visų keičiamų atminties ląstelių absoliučius adresus. CS=1200h, DS=54FFh, SS=0998h, ES=0000h, AX=5545h, BX=0000h,CX=66FFh, DX=0EACh, SI=8800h, DI=7784h, BP=0000h, SP=0004h.

2.1
```
PUSH  AX
PUSH  [BX+SI]
PUSH  BP
PUSH  CX
PUSH  DX
POP   DI
POP   [BP+189]
POP   SS:[BX+DI-3]
POP   CX
```

2.2
```
PUSH  AX
PUSH  BX
PUSH  CX
PUSH  DX
PUSH  CX
POP   BX
POP   [BX]
POP   SI
POP   [SI+130]
POP   SS:[SI+BP]
```

2.3
```
PUSH  AX
PUSH  BX
PUSH  CX
PUSH  DX
POP   BP
POP   [BP+SI]
POP   DI
POP   [DI+BX]
```

2.4
```
PUSH  CS
PUSH  DS
PUSH  ES
PUSH  SS
MOV   BP,SP
MOV   BX,[BP+2]
MOV   DI,[BP+4]
POP   [BP+DI]
POP   [DI]
POP   [SI+BX]
POP   [BP]
```

3 Perrašykite aukščiau pateiktus fragmentus, nenaudojant komandų PUSH ir POP

4 Į steką įrašyti ne mažiau nei 2 žodžiai. Sukeiskite du žodžius steko viršūnėje ir sekančioje ląstelėje vietomis. Nenaudoti kintamųjų ir išsaugoti pradines registrų reikšmes.

3.6. Pertraukimų sistema

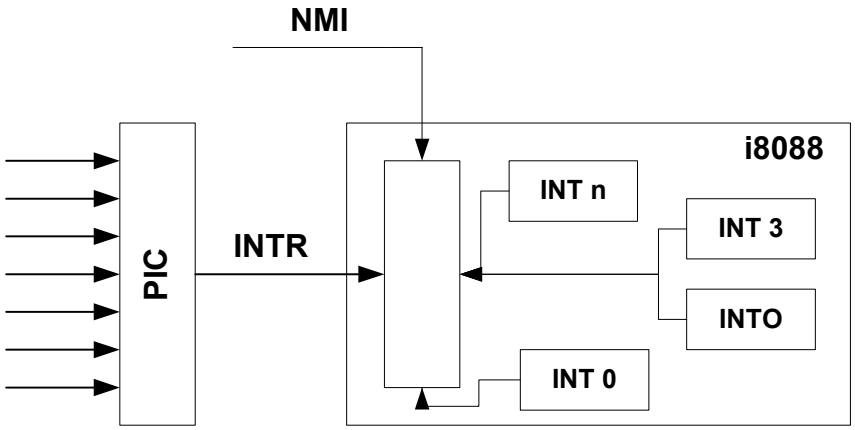
Pertraukimas – tai kompiuterio galimybė reaguoti į tam tikrus įvykius. Įvykiai gali įvykti kaip procesoriaus viduje, taip ir išoriniuose įrenginiuose. Kai įvyksta įvykis, programos tęsti arba negalima (įvyko kritinė klaida), arba neverta (pvz., reikia apdoroti įvedimą iš klaviatūros), nes reikia kažkaip sureaguoti į įvykį. Kompiuterio architektūroje numatyta, kad kiekvienas įrenginys, kuriame įvyksta įvykis generuoja pertraukimo signalą – elektrinį signalą, kuris ateina į specialią mikroschemą (kontrolerį). Pertraukimo signalas kiekvienam įvykiui turi fiksuotą numerį. Pagal pertraukimo numerį procesorius reaguoja į įvykį. Pertraukimo signalai būna vidiniai ir išoriniai procesoriaus atžvilgiu. Reikia pastebėti, kad pertraukimų mechanizmas atsirado trečios kartos kompiuteriuose. Kompiuteriui pertraukimai yra tas pats, kas telefonui yra skambutis. Jeigu nebūtų pertraukimų, procesoriui reikėtų pastoviai apklausinėti išorinius įrenginius apie galimus įvykius. Pertraukimų mechanizmo

buvimas kompiuterio architektūroje yra būtina multiprograminių operacinių sistemų veikimo sąlyga.

Pertraukimo signalas priverčia procesorių laikinai sustabdyti einamosios programos vykdymą ir pereiti prie pertraukimo apdorojimo procedūros, kuri laikoma žymiai svarbesne. Sustabdytos programos vykdymas turi būti atnaujintas taip, lyg jokio pertraukimo ir nebuvo. Tam reikia atmintyje (mūsų atveju steke) išsaugoti programai kritinę informaciją.

Kaip jau buvo minėta, pertraukimai gali būti išoriniai ir vidiniai. Išoriniai pertraukimai savo ruožtu sugrupuoti į maskuojamus ir nemaskuojamus pertraukimus. Architektūroje maskuojamiems pertraukimams numatyta galimybė juos programiškai laikinai uždrausti. Negalima uždrausti nemaskuojamų pertraukimų apdorojimą.

Procesoriuje fiziškai išskirtos dvi įėjimo linijos išoriniams pertraukimams – INTR ir NMI (žr. 3.6.1).



3.6.1 pav. Intel 8088 pertraukimų įvadai

Įėjimas INTR prijungimas prie programuojamo pertraukimų kontrolerio (*Programmable Interrupt Controller - PIC*) mikroschemos išėjimo. Prie programuojamo pertrukimų kontrolerio prijungimai išoriniai įrenginiai, kuriems reikalingas pertraukimų apdorojimas. Pagrindinės pertraukimų kontrolerio funkcijos – pertraukimų užklausų nuo išorinių įrenginių priėmimas, įrenginio su aukščiausių prioritetų nustatymas ir magistralės INTR aktyvacija. Pertraukimus, kurie ateina į procesorių per magistralę INTR, galima uždrausti, nustačius požymio IF reikšmę į 0.

Išoriniai nemaskuojami pertraukimai patenka į procesorių per magistralę NMI. Nemaskuojami pertraukimai turi aukštesnį prioritetą negu maskuojami pertraukimai. Pertraukimai, ateinantys per magistralę NMI signalizuoja apie tokius kritinius įvykius, kaip elektros dingimas, atminties mikroschemų klaidos ir t.t. Nemaskuojami

pertraukimai negali būti uždrausti. Išorinio nemaskuojamo pertraukimo signalas fiksuojamas procesoriuje (atvirkščiai nuo maskuojamų pertraukimų) ir aptarnaujamas po einamosios komandos įvykdymo. Nemaskuojamiems išorinėms pertraukimams iš anksto nustatytas pertraukimo numeris 2, todėl nereikia siųsti pertraukimo signalo numerį.

Vidiniai pertraukimai charakterizuojami pertraukimo numeriu, kuris arba nurodytas komandoje, arba apibrėžtas ir žinomas.

Dalybos klaidos pertraukimas (pertraukimo numeris 0) generuojamas procesoriumi iškart po dalybos operacijos (DIV, IDIV) tuo atveju, kai dalybos rezultatas netelpa į registrą-rezultatą, arba įvyko dalyba iš nulio.

Perpildymo pertraukimas (pertraukimo numeris 4) generuojamas pagal vieno baito dydžio komandą INTO tuo atveju, kai požymis OF lygus 1. Toks pertraukimas reikalingas saugioms programoms rašyti. Iš esmės, pertraukimo komanda INTO yra trumpiausia sąlyginio valdymo perdavimo komanda, kuri naudoja netiesioginį adresavimo būdą.

Žingsninio režimo pertraukimas (pertraukimo numeris 1) generuojamas procesoriumi po kiekvienos įvykdytos komandos[5], kai požymis TF lygus 1. Dažniausia šitas pertraukimas naudojamas derinimo programose (*debuggers*).

Sustojimo taško pertraukimas pagal komanda INT (pertraukimo numeris 3) taip pat reikalingas derinimo programoms. Sustojimo taško pertraukimo komanda užima vieną baitą.

Programiniai pertraukimai INT N (kur N užima vieną baitą) nepriklauso nuo požymių registro. Pertraukimo numeris tokiems pertraukimams užkoduotas komandoje.

Komanda INT N panaši į komandą CALL, kuri naudoja netiesioginį adresavimo būdą. Programinius pertraukimus labai patogu naudoti išoriniams pertraukimams emuliuoti. Taip programiniai pertraukimai naudojami bendrų procedūrų iškvietime – daugeliui programų reikalinga vykdyti dažnai pasikartojančius veiksmus (duomenų įvedimas ir išvedimas, standartinių paprogramių iškvietimas ir pan.). Procedūros, vykdančios tokius veiksmus, apiforminamos kaip standartinių procedūrų bibliotekos ir pastoviai yra pagrindinėje atmintyje. Programos neturėtų naudotis šių bibliotekų

[5] Yra išimtinės komandos, kurioms pertraukimo apdorojimas prasideda po sekančios įvykdytos komandos

tiesioginiais adresais, nes jie gali būti pakeisti naujose sistemos versijose. Todėl standartinėms bibliotekoms pasiekti naudojamas netiesioginis adresavimo būdas. Pavyzdžiui, operacinės sistemos MS-DOS servisai prieinami per programinį pertraukimą su numeriu 21h. Tokios pertraukimo komandos dar vadinamos sisteminiais iškvietimais (operacinės sistemos funkcijomis), o standartinių procedūrų bibliotekos – bazine įvedimo-išvedimo procedūrų sistema (*BIOS – basic input output system*).

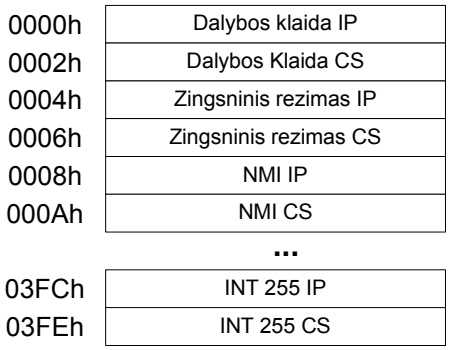
Pertraukimams (tiek išoriniams, tiek vidiniams) priskiriami prioritetai. Pertraukimų prioritetai naudojami išrinkti pertraukimą apdorojimui, kai vienu metu ateina keli pertraukimų signalai. Lentelėje 3.6.2 pateikti pertraukimų prioritetai – kuo mažesnis skaičius, tuo aukštesnis pertraukimo prioritetas.

Prioritetas	Pertraukimas
1	Dalyba iš nulio (pertraukimo numeris 0)
2	Programinis vidinis pertraukimas pagal komandą INT
3	Pertraukimas pagal komandą INTO (pertraukimo numeris 4)
4	Nemaskuojamas išorinis pertraukimas (pertraukimo numeris 2)
5	Maskuojamas išorinis pertraukimas
6	Žingsninio režimo pertraukimas (pertraukimo numeris 1)

3.6.2 Lent. Pertraukimų prioritetai

Mes išnagrinėjome pertraukimų atsiradimo šaltinius, dabar panagrinėkime kaip procesorius reaguoja į pertraukimą. Žingsniai, kuriuos vykdo procesorius, reaguodamas į pertraukimų signalus yra vienodi išoriniams ir vidiniams pertraukimams.

Kiekvienam pertraukimui egzistuoja pertraukimo apdorojimo procedūra (*ISR – interrupt service routine*). Ryšys tarp pertraukimo numeriu ir pertraukimo apdorojimo procedūros organizuotas per vektorių lentelę. Pertraukimų vektorių lentelė yra masyvas (žr. pav. 3.6.3), kurio kiekvienas elementas užima po 4 baitus.



3.6.3 pav. Vektorių lentelė

Lentelė prasideda fiziniu adresu 00000h ir užima 1KB. Kiekvienas masyvo elementas – vektorius – saugo ISR adresą. Pirmieji du vektoriaus baitai – ISR efektyvus adresas, sekantys du baitai – segmento paragrafo numeris. Pavyzdžiui, tegu atmintyje fiziniu adresu 0000Ch yra baitų seka 32h 57h C2h 03h. Fiziniu adresu 0000Ch yra pertraukimo su numeriu 3 vektorius (12 / 4 = 3). Pagal vektoriaus reikšmę galime nustatyti pertraukimo su numeriu 3 ISR fizinį adresą – 03C2h:5732h.

Kai į procesorių patenka pertraukimo signalas su numeriu N, procesorius įvykdo einamąją komandą ir jeigu einamojo komanda nėra išimtis, tai procesorius įvykdo tokius veiksmus:

1. Į steko viršūnę įrašoma požymių registro reikšmė.
2. Požymiai IF ir TF nustatomi į 0.
3. Į steko viršūnę įrašoma registro CS reikšmė.
4. Apskaičiuojamas pertraukimo vektoriaus adresas – 4*N.
5. Į registrą CS įrašomas žodis iš atminties fiziniu adresu 4*N+2.
6. Į steko viršūnę įrašoma registro IP reikšmė.
7. Į registrą IP įrašomas žodis iš atminties fiziniu adresu 4*N.

Po to, kai į registrą IP yra įrašyta nauja reikšmė, procesorius pradeda vykdyti pirmą ISR komandą. Paskutinė ISR komanda visada turi būti komanda IRET. Šita komanda atlieka aukščiau pateiktus žingsnius atvirkštine tvarka – atstato registrų IP ir CS reikšmes, atstato požymių registro reikšmę.

Pertraukimo iškvietimas labai panašus į paprogramių iškvietimą, bet papildomai steke išsaugoma požymių registro reikšmė. Tai reikalinga tam, kad pertraukta programa būtų vykdoma toliau lyg jokio pertraukimo ir nebuvo. Pavyzdžiui, jeigu pertraukimo signalas įvyks komandos ADD vykdymo metu ir operacijos rezultate bus

perpildymas (t.y. OF=1) ir procesorius neišsaugos požymių registro reikšmės, tai grįžus iš ISR programa gali veikti nekorektiškai.

Reenterabilios programos – svarbi sisteminio programavimo sąvoka. Programa yra reenterabili, jeigu leidžia pakartotinai įėjimą į programos pradžia, dar iš jos neišėjus.

Pakartotinis įėjimas atliekamas ne pačios programos, o kitomis programomis arba aparatūriniais įrenginiais. Kaip pavyzdį galima pateikti pertraukimo apdorojimo procedūras. Pertraukimai gali kilti bet kuriuo momentu. Ir net tada, kai vykdoma kokio nors pertraukimo ISR. Jeigu pakartotinai kyla pertraukimas su tuo pačiu numeriu, kai vykdoma ISR, tai pertraukimo apdorojimo procedūra bus pristabdyta einamajame taške ir bus pradėta vykdyti nuo įėjimo taško. Jeigu pertraukimo padorojimo procedūra netenkins reenterabilumo savybių, tai programos vykdymo rezultatas nėra apibrėžtas. Išvardinkime programų reenterabilumo savybes:

1. Programa nekeičia kodo segmento.
2. Programa arba nenaudoja duomenų segmento, arba kiekvienu įėjimu gauna naujas segmentų kopijas.
3. Kiekvienu įėjimu programa gauna naują steko segmentą.

Visa šiuolaikinė sisteminė programinė įranga yra reenterabili.

Steko perjungimas pertraukimo apdorojimo procedūrose

Vienas iš programos reenterabilumo reikalavimų yra steko keitimas. Jeigu pertraukimo apdorojimo procedūra nepakeis steko segmento, ji naudos pristabdytos programos steką. Tokiu atveju pristabdyta programa gali būti sugadinta, jeigu ISR aktyviai naudos steko steką. Todėl kiekviena ISR prieš pradedant dirbti su steku turi pakeisti steko segmentą. Čia reikia pastebėti, kad jei ISR ir tiesiogiai nenaudoja komandų darbui su steku vis tiek pristabdytas programos stekas nėra apsaugotas – bet kuriuo metu gali kilti pertraukimas su tuo pačiu numeriu ir į pristabdytos programos steką bus įrašytos požymių, CS ir IP registrų reikšmes.

ISR turi išsaugoti originalias SS ir SP reikšmes, išskirti vietą naujam stekui ir nustatyti savo steko segmento registrų reikšmes.

Ryšys su operacine sistema

Kaip jau buvo minėta aukščiau, programinius pertraukimus patogu naudoti sisteminių programų bibliotekoms iškviesti. Kaip pavyzdį galime pateikti operacinę sistemą. Į operacinę sistemą galima žiūrėti, kaip sisteminių programų bibliotekų rinkinį, kurį naudoja taikomosios programos.

Operacinės sistemos modulių adresai atmintyje gali pasikeisti, todėl logiška naudoti netiesioginį sisteminių paprogramių adresavimo būdą. Tokiam adresavimui labai tinka pertraukimų apdorojimo mechanizmas. Pavyzdžiui, operacinė sistema MS-DOS teikia savo servisus per programinį pertraukimą su kodu 21h. T.y. vektoriuje su numeriu 21h yra operacinės sistemos sisteminių bibliotekų adresas. MS-DOS servisai realizuoti funkcijų pavidalu. Funkcijoms priskirti numeriai, pertraukimo su numeriu 21h apdorojimo procedūra ištraukia funkcijos numerį iš registro AH. Pavyzdžiui, pertraukimo generavimas komanda INT 21h, kai AH reikšmė yra 4Ch iškviečia operacinės sistemos funkciją "Valdymo perdavimas operacinei sistemai", o funkcija su numeriu 09h – "Eilutės išvedimas į ekraną". Funkcijų parametrai perduodami per registrus, pavyzdžiui, funkcija su numeriu 09h išveda į ekraną simbolių eilutę, kurios adresas nurodytas registruose DS:DX.

Aukščiau pateiktoje pertraukimų apdorojimo schemoje buvo pasakyta, kad procesorius pradeda reaguoti į pertraukimą įvykdžius einamąją komandą. Bet šiai taisyklei egzistuoja išimtys. Tuo atveju, kai pertraukimas kyla procesoriui vykdant komandą, pateiktą lentelėje 3.6.X, reaguoti į pertraukimą pradedama įvykdžius sekančią komandą.

Komanda	**Pastabos**
Segmento keitimo prefiksas	Segmento keitimo prefiksas iš esmės pakeičia vykdomos komandos logiką (operando efektyvaus adreso skaičiavimą), todėl negalima pertraukti programos šioje vietoje
Pasikartojimo prefiksas	Taisyklė negalioja pasikartojimo prefiksams su eilutinėmis komandomis
Magistralės blokavimo prefiksas LOCK	
MOV	Kai keičiama segmentinio registro reikšmė. Šita taisyklė

	skirta užtikrinti saugiam segmento keitimui, pvz., steko
POP	Kai keičiama segmentinio registro reikšmė.
STI	
IRET	

3.6.4 lent. Išimtinės komandos

Bendra pertraukimo apdorojimo schema pateikta [Mit03, p.54].

Pratimai

1 Užrašyti programos fragmentą, kuris keičia pertraukimo su kodu N fizinį adresą į A_{fiz}.

1.1 N=0, A_{fiz}=0F3ABh.
1.2 N=21h, A_{fiz}=3ABCDh.
1.3 N=2Eh, A_{fiz}=07321h.
1.4 N=71h, A_{fiz}=1EA00h.
1.5 N=0F0h, A_{fiz}=3FF0Eh.
1.6 N=47h, A_{fiz}=0E0003h.
1.7 N=7Dh, A_{fiz}=7FBC3h.
1.8 N=0FFh, A_{fiz}=6F320h.
1.9 N=0FCh, A_{fiz}=90000h.
1.10 N=1Fh, A_{fiz}=87F0Eh.
1.11 N=2Ch, A_{fiz}=32780h.
1.12 N=66h, A_{fiz}=8F300h.

2 Duotas programos fragmentas:

```
        XOR   AX,AX
        PUSH  AX
        POP   ES
        MOV   DI,4
        MOV   CX,150h
        MOV   AX,START
        MOV   BX,STEP
a2:
        MOV   ES:[DI],AX
        ADD   AX,BX
        ADD   DI,2
        LOOP  A2
```

Apskaičiuokite pertraukimo su numeriu N ISR fizinį adresą A_{fiz}, kai duoti START ir STEP reikšmės

2.1 N=1, START=2, STEP=1.
2.2 N=1, START=2, STEP=-4.
2.3 N=9, START=15, STEP=5.
2.4 N=15, START=15, STEP=11.
2.5 N=21h,START=15h,STEP=0Ch.
2.6 N=99, START=7, STEP=23.
2.7 N=12, START=13, STEP=-12.
2.8 N=2Eh, START=77h, STEP=32.
2.9 N=5, START=100h, STEP=17.
2.10 N=8, START=-5, STEP=6.

3 Duotas programos fragmentas:

```
        MOV   AX,VALUE      (1) CS:00040h
        PUSH  AX            (2) CS:00043h
        POPF                (3) CS:00044h
        MOV   AX,0          (4) CS:00045H
        PUSH  AX            (5) CS:00048h
        POP   DS            (6) CS:00049h
        XOR   CX,CX         (7) CS:0004Ah
```

Apskaičiiuokite komandos, kuri bus vykdoma po komandos CMD, kai nurodyta VALUE reikšmė. CS=045Eh, SS=55AFh. Atminties baitai su adresais nuo 0 iki 15 užpildyti Fibonači skaičiais (1,1,2,3,5,8,…).

3.1 CMD=(1), VALUE=0F301h.
3.2 CMD=(4), VALUE=0F5FFh.

3.3 CMD=(4), VALUE=0F601h.
3.4 CMD=(6), VALUE=0791h.
3.5 CMD=(7), VALUE=0A55h.
3.6 CMD=(2), VALUE=0B60h.
3.7 CMD=(4), VALUE=1D32h.
3.8 CMD=(5), VALUE=2FF0h.
3.9 CMD=(3), VALUE=2CC3h.
3.10 CMD=(3), VALUE=9901h.
3.11 CMD=(7), VALUE=9703h.
3.12 CMD=(6), VALUE=7203h.

4 Duotam programos fragmentui apskaičiuokite sekančios vykdomos komandos fizinį adresą, jeigu vykdant pirmąją fragmento komandą kyla pertraukiams su numeriu 3. Atminties baitai su adresais nuo 0 iki 26 užpildyti seka 37h,36h,35h, ir t.t. CS=420Fh

4.1
```
CS:0000 MOV SP,AX
CS:0002 MOV SS,BX
```
4.2
```
CS:0000 MOV [BX],CX
CS:0002 MOV DX,AX
```
4.3
```
CS:0000 MOV SS,AX
CS:0002 MOV SP,BX
```
4.4
```
CS:0000 MOV DS,AX
CS:0002 MOV DX,09h
```
4.5
```
CS:0000 XOR  AX,AX
CS:0002 MOV ES,AX
```
4.6
```
CS:0000 PUSH AX
CS:0001 POP  DX
```
4.7
```
CS:0000 POP  ES
CS:0001 POP  AX
```
4.8
```
CS:0000 MOV ES,DX
CS:0002 INT  21h
```

5 Duotas programos fragmentas:

```
MOV   AX,0   (1) CS:0000
MOV   ES,AX  (2) CS:0003
PUSH  DX     (3) CS:0005
POP   AX     (4) CS:0006
```

Vykdant komandą CMD, kyla pertraukimas su numeriu N. Nurodykite baito fiziniu adresu A_{fiz} reikšmę, kai duotos registrų reikšmės.

5.1 CMD=(1), N=12h, CS=320Fh,SS=1200h,SP=0FF00h, A_{fiz}=21EFDhh.
5.2 CMD=(1), N=13h, CS=6650h,SS=0EACh,SP=0002h, A_{fiz}=1EABDh.
5.3 CMD=(2), N=09h, CS=440Fh,SS=00ABh,SP=0004h, A_{fiz}=10AAEh.
5.4 CMD=(3), N=01h, CS=1230h,SS=99FFh,SP=0012h, A_{fiz}=99FFFh.
5.5 CMD=(2), N=05h, CS=3350h,SS=74FEh,SP=0000h, A_{fiz}=84FDBh.
5.6 CMD=(1), N=03h, CS=5545h,SS=2345h,SP=4567h, A_{fiz}=279B4h.
5.7 CMD=(3), N=04h, CS=32FEh,SS=0EFCh,SP=0002h,A_{fiz}=1EFBDh.
5.8 CMD=(2), N=02h, CS=665Eh,SS=0AB7h,SP=0000h,A_{fiz}=1AB6Dh.
5.9 CMD=(3), N=00h, CS=0AFEh,SS=1FFFh,SP=0002h,A_{fiz}=2FFEEh.

6 Duotas programso fragmentas:

```
mov   ax,200h(1) CS:0000h
mov   bx,1    (2) CS:0003h
div   bl      (3) CS:0006h
aad           (4) CS:0008h
```

Apskaičiuokite komandos, kuri bus vykdoma po (3) komandos, fizinėį adresą kai atminties baitai su adresais nuo 0 iki 12 užpildyti seka 23h,22h,21h,20h,1Fh ir t.t. CS=124Fh, SS=44FEh,SP=0002h.

4. Komandų sistema

4.1. Bendrosios komandos

Bendrosios mikroprocesoriaus Intel 8088 komandos sudaro 3 pogrupius – persiuntimo, įvedimo-išvedimo ir steko komandos.

Persiuntimo komanda **MOV DEST,SRC** persiunčia operandą SRC į operandą DEST. Komandos mašininiai kodai ir formatai pateikti [Mit03, p.65-66]. Komandos operandui, esančiam atmintyje, procesorius formuoja fizinį adresą segmente, kurį nurodo registras DS (jeigu nenurodytas segmento keitimo prefiksas, žr. 4.10). Operandu DEST gali būti segmentiniai registrai DS,ES arba SS, bet ne CS – kodo segmentinį registrą galima keisti tik valdymo perdavimo komandomis. Reikia pastebėti, kad komandai neapibrėžtas formatas, kai DEST yra segmentinis registras, o SRC – betarpiškas operandas. Norint įrašyti konstantą į segmentinį registrą, reikia dviejų komandų:

```
MOV  REG,SRC;
MOV SEGREG, REG
```

kur REG yra vienas iš bendro naudojimo registrų, o SEGREG yra ES,DS arba SS.

Komanda MOV nekeičia požymių registro. Pateiksime MOV komandos užrašymo pavyzdžius asemblerio kalba:

```
MOV BYTE PTR [BX],DL
MOV DL,CH
MOV WORD PTR [DI+7Eh]
MOV SS,AX
MOV DI,-15
MOV B,AL
MOV CX,W
MOV DX,OFFSET STR
```

Komanda **XCHG DEST,SRC** skirta duomenų apsikeitimui tarp DEST ir SRC, t.y. SRC reikšmė įrašoma į operandą DEST, o operando DEST reikšmė įrašoma į SRC (žr. [Mit03, p.66]). Komanda XCHG AX,AX naudojama kaip tuščios operacijos komanda NOP. Komandos operandu negali būti betarpiškas operandas. Komanda nekeičia požymių registro.

Vieno baito komanda be operandų **XLAT** skirta kodų perkodavimui (žr. [Mit03, p.67]). Registre AL esanti reikšmė traktuojama kaip poslinkis lentelėje, kurios fizinį adresą nusako pora DS:BX. Baitas iš lentelės, nurodyto registru AL, persiunčiamas į registrą AL. T.y. AL:=DS:[BX+AL]. Komanda nekeičia požymių registro.

Komandos **LES REG,MEM**; **LEA REG,SRC**; ir **LDS REG,MEM** skiriasi nuo kitų persiuntimo komandų tuo, kad į registrus patenka ne duomenys iš atminties, o operandų adresai (žr. [Mit03, p.68]). Pagrindinis šių komandų pritaikymas – registrų nustatymas prieš vykdant komandas darbui su eilutėmis arba kviečiant paprogrames. Komandos nekeičia požymių registro.

Komanda **LEA REG,MEM** apskaičiuoja atminties ląstelės MEM efektyvų adresą ir užrašo jį į registrą REG. Ši komanda gali būti užrašyta komanda MOV – **MOV REG, OFFSET M**.

Komandos **LES** ir **LDS** nustato segmentinį ir bendro naudojimo registrus. Komandoje nurodytam operandui apskaičiuojamas efektyvus adresas. Pagal apskaičiuotą EA ir segmentinį registrą DS apskaičiuojamas fizinis adresas. Tuo adresu esantis žodis persiunčiamas į bendro naudojimo registrą REG, o sekantis po jo žodis į segmentinį registrą DS **LDS** komandos atveju ir ES **LES** komandos atveju.

Įvedimo-išvedimo komandos skirtos darbui su portais (žr. [Mit03, p.36-37]). Komanda **IN** persiunčia baitą arba žodį iš įvedimo-išvedimo porto į registrą AL arba AX. Porto numeris gali būti nurodytas komandoje arba registre DX. Komanda OUT persiunčia baitą arba žodį iš registro AL arba AX į įvedimo-išvedimo portą. Porto numeris gali būti nurodytas komandoje, arba registre DX. Komandos nekeičia požymių registro.

Komandos darbui su steku **PUSH**, **POP**, **PUSHF**, **POPF** dirba su steko viršūne, kurios fizinis adresas nusakomas pora SS:SP.

Komanda **PUSH OP** sumažina registro SP reikšmę dvejetu ir į atmintį adresu SS:SP įrašo operandą OP. Operandas OP būtinai yra 16 bitų ir gali būti operandas atmintyje (duomenų segmente), bendro naudojimo registras arba segmentiniu registras. Komanda nekeičia požymių registro.

Komanda **POP OP** įrašo į operandą OP steko viršūnėje esančią reikšmę ir padidina registro SP reikšmę dvejetu. Operandu OP gali būti operandas atmintyje (duomenų segmente), bendro naudojimo registras arba segmentinis registras. Pastebėkime, kad komanda PUSH, kai operandas yra registre SP veikia nekorektiškai. Pvz., tegu SP pradinė reikšmė yra 100h, tada po komandų

```
PUSH SP
POP  AX
```

Registro SP reikšmė bus 100h, o registro AX reikšmė – 0Feh. Šita klaida buvo ištaisyta vėlesniuose procesorių modeliuose. Komanda nekeičia požymių registro.

Komandos **PUSHF** ir **POPF** yra be operandų ir veikia kaip ir komandos PUSH ir POP, tik šiuo atveju operandas yra požymių registras. Šitos komandos naudojamos tuo atveju, kai reikia sužinoti požymių registro reikšmę, arba pakeisti visus požymius, o ne po vieną bitą. Pavyzdžiui, komandos

```
PUSHF;
POP AX;
```

į registrą AX įrašo požymių registro reikšmę. O komandos

```
PUSH AX;
POPF;
```

Į požymių registrą įrašo reikšmę iš registro AX. Komanda POPF keičia visus požymius.

Komandos **LAHF** ir **SAHF** taip pat yra be operandų ir skirti darbui su požymių registrų. Komanda LAHF į registrą AH įrašo jaunesnįjį požymių registro baitą. Komanda SAHF baito AH reikšmę įrašo į požymių registro jaunesnįjį baitą. Komanda SAHF keičia požymius CF, PF, AF, SF ir ZF.

Pratimai

1 Užrašyti pateiktus fragmentus, nenaudojant komandos MOV

1.1
```
MOV AX,BX
MOV BX,CX
MOV CH,AL
```

1.2
```
MOV AX,CX
MOV CX,DX
MOV AX,DX
```

1.3
```
MOV AL,BL
MOV BL,CL
MOV CL,CH
```

1.4
```
MOV AH,AL
MOV AL,DH
MOV DL,DH
```

2 Užrašyti pateiktus fragmentus, nenaudojant komandos XCHG

2.1
```
XCHG AX,BX
XCHG BX,CX
```

2.2
```
XCHG CX,DX
XCHG CH,AL
XCHG DH,BL
```

2.3
```
XCHG AH,AL
XCHG BL,BH
```

2.4
```
XCHG CL,DL
XCHG DH,BL
XCHG SI,DI
```

3 Užrašyti pateiktus fragmentus, nenaudojant komandų PUSH ir POP

3.1
```
PUSH AX
PUSH DX
POP AX
POP CX
```

3.2
```
PUSH AX
POP [BX+SI]
```

3.3
```
PUSH [SI-2]
PUSH [BX]
POP BX
POP AX
```

3.4
```
PUSH [BP-2]
POP BP
```

4 Užrašyti komandas SAHF ir LAHF komandomis MOV,PUSHF ir POPF.

4.2. Aritmetinės komandos

Aritmetinės komandos dirba su sveikais skaičiais be ženklo ir su ženklu. Skaičiai be ženklo gali būti 8 arba 16 bitų ir yra atitinkamai ribose 0..255 ir 0..65535. Skaičiai su ženklu taip pat gali būti 8 arba 16 bitų. Skaičiai su ženklu kinta ribose atitinkamai –128..127 ir –32768..32767.

Skaičiai užrašomi papildomu kodu – tai leidžia naudoti tas pačias sudėties ir atimties komandas tiek skaičiams be ženklo, tiek skaičiams su ženklu. Daugybos ir dalybos operacijų komandos skaičiams be ženklo ir su ženklu skiriasi.

Komanda **NEG OP** keičia operando ženklą. T.y. suformuoja operandą papildomu kodu OP:=0-OP. Operandas gali būti 8 arba 16 bitų ir yra bendro naudojimo registre arba atmintyje duomenų segmente. Pavyzdžiui, kai AL=FFh, komanda NEG AL į registrą AL įrašys 1. Komanda keičia požymių registrą.

Komanda **ADD DEST,SRC** įrašo operandų sumą į operandą DEST, DEST:=DEST+SRC. Operandais gali būti bendro naudojimo registrai, operandai atmintyje ir betarpiški operandai. Komanda keičia požymių registrą. Požymių OF ir CF reikšmių nustatymo taisyklės pateiktos poskyryje 2.1.

Komanda **ADC DEST,SRC** analogiška komandai ADD, bet papildomai prie sumos prideda požymio CF reikšmę – DEST:=DEST+SRC+CF. Jeigu ADD arba ADC komandose SRC yra betarpiškas operandas, tai priklausomai nuo konfigūracinių bitų *s* ir *w* keičiasi operando reikšmė (žr. [Mit03, pp.69-70]). Jei s=1 ir w=1, tai 8 bitų operandas išplečiamas iki žodžio pagal ženklo plėtimo taisyklę – vyriausias baito bitas užpildo vyresnįjį žodžio baitą. Pavyzdžiui, baitas 7Fh, kuris dvejetainėje sistemoje yra 01111111_2 išplečiamas į 00000000 01111111_2, arba 007Fh, o baitas 80h (10000000_2) išplečiamas į FF80h (11111111 10000000_2).

Komanda **SUB DEST,SRC** įrašo operandų skirtumą į operandą DEST, DEST:=DEST-SRC. Komandos operandai gali būti bendro naudojimo registrai, operandai atmintyje arba betarpiškas operandas (SRC). Komanda keičia požymių registrą.

Komanda **SBB DEST,SRC** analogiška komandai SUB, tik dar atima požymio CF reikšmę, DEST:=DEST-SRC-CF. Šioms komandoms taip pat yra formatas, kuriame naudojami konfigūraciniai bitai *s* ir *w*. Kai s ir w yra 1, 8 bitų operandas išplečiamas iki žodžio pagal ženklo plėtimo taisyklę. Komanda keičia požymių registrą.

Komandos **INC OP** ir **DEC OP** padidina ir sumažina operandą OP vienetu atitinkamai. Operandas OP gali būti operandas atmintyje arba bendro naudojimo 16 bitų registras. T.y. OP:=OP±1.

Komanda **CMP DEST,SRC** analogiška komandai SUB DEST,SRC, tik neišsaugo rezultato operande DEST. Komanda skirta operandų palyginimui (CMP - CoMPare). T.y. komanda tik suformuoja požymių registrą pagal operacijos rezultatą. Komanda dažniausiai naudojama su sąlyginio valdymo perdavimo komandomis. Priklausomai nuo operandų komanda suformuoja požymių registro reikšmę (žr. [Mit03, pp.71-74]).

Komanda **MUL OP** įrašo operando ir akumuliatoriaus daugybos rezultatą į registrą AX, jeigu dauginami baitai, arba į registrus [DX,AX], jei dauginami žodžiai. 8 bitų operandams komandos veikimas atrodo taip AX:=OP*AL, 16 bitų operandams taip – [DX,AX] := AX*OP (žr. [Mit03, p.74]). Komanda skirta skaičiams be ženklo dauginti.

Komanda **DIV OP** įrašo akumuliatoriaus ir operando OP dalybos rezultatą į registrus AH ir AL 8 bitų operandams, ir DX ir AX žodžiams. 8 bitų operandų atveju į registrą AL įrašomas AX/OP dalmuo, o į registrą AH AX/OP liekana. 16 bitų operandams į registrą AX įrašomas [DX,AX]/OP dalmuo, o į registrą DX įrašoma [DX,AX]/OP liekana. Čia labai svarbu pastebėti, kad komanda generuoja pertraukimą su numeriu 0 ne tik tuo atveju, kai OP reikšmė yra 0, bet ir tada, kai dalmuo netelpa į akumuliatorių. Pavyzdžiui, tarkime turime tokį fragmentą:

```
mov     ax,100h         (1)
mov     bl,1            (2)
div     bl              (3)
```

Eilutėje (3) dalybos komanda bandys įrašyti dalybos rezultatą (100h) į registrą AL. Bet rezultatas šiuo atveju netelpa į akumuliatorių, todėl bus sugeneruotas pertraukimas su numeriu 0.

Komanda **IMUL OP** skirta skaičiams su ženklu dauginti. Komanda veikia analogiškai komandai MUL OP (žr. [Mit03, p.74]). Komanda nustato požymius CF ir OF į 0, jeigu rezultatas tilpo jaunesniojoje rezultato pusėje, CF=OF=1 priešingu atveju.

Komanda **IDIV OP** skirta skaičiams su ženklu dalinti. Komanda veikia analogiškai komandai DIV OP. Šiai komandai generuojamas dalybos klaidos pertraukimas, kai dalybos rezultatas netelpa į skaičių su ženklu ribas. Pastebėkime,

kad komanda IDIV skirta skaičių su ženklu dalybai, todėl čia lengva padaryti klaidą. Jei norėdami padalinti, pvz. skaičių –5 iš 2, į registrą AL įrašysime 0FBh (5 papildomu kodu), o registre AH paliksime 0, tai gausime nekorektišką rezultatą, nes 00FBh nėra skaičius –5 dvejuose baituose. Teisingai būtų į registrą AX įrašyti reikšmę 0FFFBh. Dalybos korektiškumui užtikrinti architektūroje yra konvertavimo komandos (žr. 4.4).

Pratimai

1 Duotas programos fragmentas:

```
MOV   AL,VAL1
MOV   AH,VAL2
ADD   AH,AL
```

Nurodykite naują požymių registro reikšmę, kai duoti VAL1, VAL2 ir požymių registro SF reikšmė prieš vykdant fragmentą.

1.1 VAL1=100, VAL2=127, SF=FF01h.
1.2 VAL1=37, VAL2=96, SF=F002h.
1.3 VAL1=-12, VAL2=123, SF=0000h.
1.4 VAL1=-5, VAL2=-124, SF=0000h.
1.5 VAL1=8, VAL2=-128, SF=0000h.
1.6 VAL1=9, VAL2=123, SF=0000h.
1.7 VAL1=64, VAL2=87, SF=0001h.
1.8 VAL1=12, VAL2=97, SF=00FFh.
1.9 VAL1=-64, VAL2=-98, SF=00F0h.
1.10 VAL1=3, VAL2=2, SF=0FFFFh.
1.11 VAL1=98, VAL2=-23, SF=0003h.
1.12 VAL1=37, VAL2=-15, SF=0005h.

.

4.3. Dešimtainiai skaičiai

Dešimtainio formato skaičiams naudojamos tos pačios aritmetinės komandos, kaip ir skaičiams be ženklo, bet papildomai į komandų rinkinį įvestos korekcijos komandos.

Išpakuotiems dešimtainiams skaičiams apibrėžtos korekcijos komandos sudėties, atimties, daugybos ir dalybos operacijoms. Korekcijos komandos turi eiti po aritmetinių komandų, ir operacijos rezultatas turi būti užrašytas registre AL.

Komanda **AAA** yra be operandų ir skirta koreguoti dviejų dešimtainių išpakuotų skaičių sudėties rezultatą. Komandos AAA veikimas priklauso nuo AL registre esančios reikšmės ir požymio AF. Komandos veikimo algoritmas mums jau yra žinomas (žr. poskyrį 2.3). Detalizuokime jį, nes šiuo atveju keičiami baitai turi būti fiksuotuose registruose.

```
if((AL & 0Fh)>09h || (AF=1))
```

```
{
   AL:=AL+6;
   AH:=AH+1;
   AF:=1;
   CF:=1;
}
else
{
   AF:=0;
   CF:=0;
}
AL := AL & 0Fh;
```

Kaip jau buvo minėta aukščiau, požymis AF nustatomas į 1, kai įvyksta pernešimas iš jaunesniojo pusbaičio į vyresnįjį.

Komanda **DAA** koreguoja supakuotą dešimtainį skaičių registre AL. Komandos veikimo algoritmas panašus į komandos AAA veikimo algoritmą:

```
if((AL and 0Fh) > 9 || AF=1)
{
   AL := AL + 6;
   AF := 1;
}
if(AL > 9Fh || CF=1)
{
   AL := AL + 60h;
   CF := 1;
}
```

Čia atskirai koreguojami jaunesnysis ir vyresnysis pusbaičiai.

Komanda **AAS** skirta koreguoti dešimtainių išpakuotų skaičių atimties rezultatą. Koreguojama reikšmė yra registre AL. Komandos veikimo algoritmas panašus į komandos AA algoritmą, bet kai reikia koreguoti reikšmę – iš registro AL reikia atimti 6:

```
if((AL and 0Fh)>9 || AF=1)
{
   AL := AL - 6;
   AH := AH - 1;
   CF := 1;
   AF := 1;
}
AL := AL and 0Fh;
```

Komanda **DAS** skirta koreguoti dešimtainių supakuotų skaičių atimties rezultatą, kuris yra registre AL. Komandos veikimas panašus į komandos DAA. Pateiskime komandos veikimo algoritmą:

```
if((AL and 0Fh)>9 or AF=1)
{
   AL := AL -6;
   AF := 1;
}
if(AL>9Fh or CF=1)
{
```

```
    AL := AL - 60h
    CF := 1
}
```

Komanda **AAM** skirta dešimtainių išpakuotų skaičių daugybos rezultatui koreguoti. Koreguojamas registras AL. Padalinus registro reikšmę iš 10, komandą įrašo dalmenį į registrą AH, o liekaną į registrą AL. Komandos \veikimo algoritmas:

AL := AL mod 10;

AH := AL div 10;

Komanda **AAD** skirta dešimtainių išpakuotų skaičių dalybos operandams koreguoti, t.y. atvirkščiai nuo kitų koregavimo komandų, AAD taikoma prieš aritmetinę operaciją, o ne po jos. Komandos veikimo algoritmas:

AL := AH*10 + AL;

AH := 0;

Pavyzdžiui, norime padalinti dešimtainę išpakuotą reikšmę registre AX=0102h iš keturių. Jeigu neatliksime korekcijos gausime nekorektišką rezultatą, nes dalinsime skaičių 268, o ne 12.

Pratimai

1 Apskaičiuokite registrų AL ir AH bei požymių AF ir CF reikšmes po nurodyto fragmento.

1.1
```
MOV AL,06
SUB AH,AH
ADD AL,03
AAA
```

1.2
```
MOV AL,05
MOV AH,65h
ADD AL,05
AAA
```

1.3
```
MOV AL,05
MOV AH,87h
ADD AL,07
AAA
```

1.4
```
MOV AX,8407h
ADD AL,08h
AAA
```

1.5
```
MOV AX,0009h
ADD AL,09h
AAA
```

1.6
```
MOV AL,08
SUB AL,05
AAS
MOV AL,06
SUB AL,07
AAS
```

1.7
```
MOV AL,02
SUB AL,09
AAS
```

1.8
```
MOV AL,06
SUB AL,19
AAS
```

4.4. Konvertavimas

Konvertavimo komandos **CBW** ir **CWD** dažniausia naudojamos kartu su dalybos operacija. Komanda **CBW** (Convert Byte to Word) užrašo vyriausią registro AL bitą į visus registro AH bitus. Tai yra ženklo išplėtimas. Jeigu, pavyzdžiui AL reikšmė prieš komandą CBW buvo 7Fh, tai po komandos registro AX reikšmė bus 007Fh, o jei AL

reikšmė buvo 80h, tai AX bus FF80h. T.y. užrašoma ta pati reikšmė kaip skaičius su ženklu dviejuose baituose. Komanda reikalinga teisingam komandos IDIV rezultatui, kai dalinami 8 bitų skaičiai.

Komanda **CWD** (Convert Word to Double) veikia panašiai – vyriausias registro AX bitas užpildo visus registro DX bitus. Komanda reikalinga, kai dalinami 16 bitų skaičiai. Pavyzdžiui, jeigu AX reikšmė yra 7F0Eh, tai po komandos CWD registro DX reikšmė bus 0000h, o jeigu AX reikšmė yra A065h, tai DX reikšmė po komandos bus 0FFFFh. T.y. užrašoma ta pati reikšmė kaip skaičius su ženklu keturiuose baituose.

Pratimai

1 Nurodykite registro AX reikšmę po komandos CBW.

1.1 AX=13.
1.2 AX=123.
1.3 AX=54.
1.4 AX=7Fh.
1.5 AX=356.
1.6 AX=0AE56h.
1.7 AX=0FFFFh.
1.8 AX=0FEEFh.
1.9 AX=0FF80h.
1.10 AX=00F7Fh.
1.11 AX=545Fh.
1.12 AX=777Fh.
1.13 AX=FF8Fh.
1.14 AX=FFF3h.
1.15 AX=99FDh.
1.16 AX=47FEh

2 Nurodykite registrų AX ir DX reikšmes po komandos CWD

2.1 AX=67.
2.2 AX=456.
2.3 AX=512.
2.4 AX=16384.
2.5 AX=32789.
2.6 AX=8000h.
2.7 AX=0FE00h.
2.8 AX=00FFh.
2.9 AX=9876h.
2.10 AX=8765h.
2.11 AX=0ABCFh.
2.12 AX=F0FFh.
2.13 AX=6FFFh.
2.14 AX=7FFFh.
2.15 AX=8745h.
2.16 AX=32768

.

4.5. Loginės komandos

Visos loginės komandos veikia kiekvieną operando bitą atskirai.

Komanda **NOT OP** – operando OP neigimas, t.y. visi operando bitai invertuojami. Komanda nekeičia požymių registro. Operandas gali būti baitas arba žodis registre arba atmintyje.

Komanda **AND DEST,SRC** – loginė operandų DEST ir SRC daugyba. Komandos veikimas:

DEST := DEST and SRC.

CF:= 0;

OF:= 0;

t.y. po komandos įvykdymo požymiai CF ir OF nustatomi į 0.

Komanda **OR DEST,SRC** – loginė operandų DEST ir SRC sudėtis. Komandos veikimo principas:

DEST:= DEST or SRC;

OF:=0;

CF:=0;

t.y. po komandos požymiai CF ir OF nustatomi į 0.

Komanda **XOR DEST,SRC** yra loginė operacija "išimtinis arba". Operandai gali būti 8 arba 16 bitų dydžio. Komandos veikimo principas:

DEST:=DEST⊕SRC;

CF:=0;

OF:=0;

Požymiai OF ir CF po komandos nustatomi į 0.

Komanda **TEST DEST,SRC** veikia analogiška komandai AND, bet neišsaugo rezultato operande DEST, t.y. apskaičiuojamas operacijos rezultatas DEST and SRC, nustatomi požymiai, bet DEST operandas nekeičiamas. Komanda nustato CF ir OF į 0, AF reikšmė yra neapibrėžta, PF,SF ir ZF reikšmės nustatomos pagal rezultatą.

Komanda **SAL/SHL DEST,SRC** operandą DEST pastumia per SRC pozicijas į kairę (šiuo atveju vienai komandai galimos dvi mnemonikos). SRC gali būti arba 1, arba CL. Iš esmės, komanda daugina operandą DEST iš 2^{SRC}. Jeigu postūmis vykdomas per vieną poziciją, komanda užrašoma SHL/SAL DEST,1, jeigu postūmis yra per kelias pozicijas, komanda užrašoma SHL/SAL DEST, CL, t.y. 2 laipsnis nurodomas registre CL. Pavyzdžiui, tarkime AL pradinė reikšmė yra 03h, tada po komandos SHL AL,1 AL reikšmė bus 06h, o po komandos SHL Al,CL kai CL yra 02h, AL reikšmė bus 0Ch. Postūmis per vieną bitą keičia požymio OF reikšmę. Jaunesnieji atsilaisvinę bitai užpildomi nuliais.

Komanda **SHR DEST,SRC** operandą DEST pastumia į dešinę per SRC bitų. Iš esmės, komanda SHR dalina operandą DEST iš 2^{SRC}. SRC gali būti arba 1, arba CL. Šiai komandai taip pat postūmis per kelias pozicijas nurodomas registre CL. Vyresnieji atsilaisvinę bitai užpildomi nuliais. Pavyzdžiui, tarkime pradinė registro AL reikšmė yra 0FFh, tada po komandos SHR AL,1 registro AL reikšmė bus 7Fh. Postūmis per vieną bitą keičia požymio OF reikšmę.

Komanda **SAR DEST,SRC** veikia analogiškai komandai SHR, bet postūmio metu vyresnieji atsilaisvinę bitai gali būti užpildyti vienetais, tam kad išsaugoti skaičiaus ženklą. T.y. atsilaisvinę bitai užpildomi ženklo bitu. SRC gali būti arba 1, arba CL. Pavyzdžiui, tarkime pradinė AL reikšmė yra 0FCh, tada po postūmio per vieną bitą nauja reikšmė bus 0FEh. O jeigu pradinė reikšmė buvo 7Fh, tai po postūmio per 2 pozicijas AL reikšmė bus 1Fh.

Komanda **ROL DEST,SRC** yra ciklinis DEST postūmis į kairę per SRC pozicijas. SRC gali būti arba 1, arba CL. Jaunesnieji atsilaisvinę bitai užpildomi išstumtais bitais (todėl postūmis yra ciklinis). Pavyzdžiui, tarkime pradinė AL reikšmė yra 80h, po komandos ROL AL,1 AL reikšmė bus 01h. Postūmis per vieną bitą keičia požymio OF reikšmę.

Komanda **ROR DEST,SRC** yra ciklinis DEST postūmis į dešinę per SRC pozicijas. SRC gali būti arba 1, arba CL. Vyresnieji atsilaisvinę bitai užpildomi išstumtais jaunesniais bitais. Pavyzdžiui, tarkime pradinė AL reikšmė yra 01h, po komandos ROR AL,1 AL reikšmė bus 80h. Postūmis per vieną bitą keičia požymio OF reikšmę.

Komanda **RCL DEST,SRC** yra ciklinis DEST postūmis per SRC pozicijas, vykdant vyresniųjų bitų postūmį per požymį CF. T.y. jaunesniojo bito pozicija užpildoma reikšme iš požymio CF, o vyresnysis išstumtas bitas patenka į požymį CF. Pavyzdžiui, tarkime AL pradinė reikšmė yra 7Fh, CF yra 0, tada atlikus postūmį per 3 pozicijas AL reikšmė bus F9h, o CF = 1. Postūmis per vieną bitą keičia požymio OF reikšmę.

Komanda **RCR DEST,SRC** veikia panašiai kaip ir komanda RCL, bet stumia bitus į dešinę. Šita komanda taip pat stumia bitus per požymį CF. Pavyzdžiui, jei pradinė AL reikšmė yra 01h, o CF yra 0, tai postūmis per vieną poziciją į registrą AL įrašys 0, o į CF 1. Postūmis per vieną bitą keičia požymio OF reikšmę.

4.6. Komandos darbui su eilutėmis

Intel 8088 mikroprocesoriuje numatytos komandos darbui su eilutėmis, kur eilutė yra baitų arba žodžių seka. Reikšmės eilutėje eina viena po kitos. Prieš komandą darbui su eilutėmis gali būti nurodytas pasikartojimo prefiksas. Maksimalus eilutės ilgis yra 64K. Pasikartojimų skaičius įrašytas registre CX. Kai vykdoma eilutinė

komanda su prefiksu, ji gali būti pertraukta (atvirkščiai, pavyzdžiui, nuo segmento keitimo prefikso).

Yra 2 pasikartojimo prefiksų komandos, kurios apibrėžtos tokios mnemonikos REP/REPE/REPZ ir REPNE/REPNZ. Pirmas prefiksas kartoja eilutinė komanda CX kartų arba kol ZF=1. Prefiksas REPNE/REPNZ kartoja komandą CX kartų arba kol ZF=0. Kai kurios eilutinės komandos nepriklauso nuo požymio ZF. Bendrai eilutinės komandos vykdymo schema pavaizduota [Mit03, pp.85-86].

Komandos darbui su eilutėmis naudoja požymį DF – Direction Flag. Kai DF yra 0 eilutės nagrinėjamos iš kairės į dešinę, o kai DF yra 1 iš dešinės į kairę.

Komanda **MOVS** persiunčia baitą arba žodį iš eilutės-šaltinio į eilutę-gavėją. Eilutės-šaltinio fizinis adresas nusakomas registrais DS:SI, o eilutės-gavėjo registrais ES:DI. Komanda gali dirbti su baitų arba žodžių eilutėmis. Kai dirbama su baitas, komanda atrodo **MOVSB**, o kai su žodžiais – **MOVSW** (t.y. operacijos kodas užima 7 bitus, jaunesnysis bitas yra pločio bitas *w*). Komanda persiunčia elementą (baitą arba žodį) iš atminties adresu DS:SI į atmintį adresu ES:DI. Jeigu DF yra 0, tai MOVSB po elemento persiuntimo padidina registrus SI ir DI vienetu, komanda MOVSW padidina SI ir DI dvejetu. Jeigu DF yra 1 (t.y. eilutės nagrinėjamos iš dešinės į kairę), komanda MOVSB sumažina SI ir DI vienetu, o MOVSW dvejetu. Komandos veikimas su pasikartojimo prefiksu nepriklauso nuo požymio ZF. Pavyzdžiui, tegu SI pradinė reikšmė yra 0012h, o DI 45FFh. Po komandos MOVSB (DF yra 0) registrų reikšmės bus 0013h ir 4600h atitinkamai. Komandos MOVSW atveju registru reikšmės bus 0014h ir 4601h atitinkamai.

Komanda **CMPS** palygina du eilučių elementus. Kaip ir MOVS komandai, šitai komandai yra du variantai – **CMPSB** ir **CMPSW**. Komanda atima iš šaltinio gavėjo elementą ir suformuoja požymių registrą ir pagal požymį DF pakoreguoja registrus SI ir DI. T.y. CMPS panaši į paprastą palyginimo komanda, bet papildomai dar keičia registrus SI ir DI. Kai komandai nurodytas pasikartojimo prefiksas, komanda priklauso nuo požymio ZF.

Komanda **SCAS** yra elemento paieškos komanda. Komandai apibrėžti du variantai – su baitais **SCASB** ir su žodžiais **SCASW**. Ieškomas elementas įrašomas į akumuliatorių (AL komandos SCASB atveju, AX – SCASW atveju). Paieška vykdoma atėmimo būdu, t.y. iš akumuliatoriaus atėmimas eilutės-gavėjo elementas. Pagal atimties rezultatą suformuojamas požymių registras. Komanda pagal požymį

DF keičia registrą DI. Kai komandai nurodytas pasikartojimo prefiksas, komandos veikimas priklauso nuo požymio ZF.

Komanda **LODS** yra elemento oš eilutės-šaltinio persiuntimas į akumuliatorių. Komandai apibrėžti du variantai – **LODSB** dirba su baitais, **LODSW** su žodžiais. Komanda keičia registrą SI priklausomai nuo požymio DF reikšmės. Komandos veikimas nepriklauso nuo požymio ZF, kai naudojamas pasikartojimo prefiksas.

Komanda **STOS** yra elemento, esančio akumuliatoriuje (AL arba AX) persiuntimas į eilutę-gavėją. Komandai apibrėžti du variantai – **STOSB** baitų eilutėms ir **STOSW** žodžių eilutėms. Komanda keičia registrą DI priklausomai nuo DF. Kai nurodytas pasikartojimo prefiksas, požymis ZF neturi reikšmės komandos veikimui. Pavyzdžiui, tarkime AL reikšmė yra 00h, CX yra 5, DF yra 0, komanda REP LODSB įrašo 5 baitus su reikšme 00h į atminties ląsteles su adresais nuo ES:[DI] iki ES:[DI+4].

Pratimai

1 **Užrašykite komandas, kurios įvykdo tuos pačius veiksmus, kaip ir komanda REP MOVSB, nenaudodami pakartojimo prefiksų ir komandų darbui su eilutėmis.**

2 **Užrašykite komandas, kurios įvykdo tuos pačius veiksmus, kaip ir komanda REP SCASW, nenaudodami pakartojimo prefiksų ir komandų darbui su eilutėmis.**

3 **Užrašykite komandas, kurios įvykdo tuos pačius veiksmus, kaip ir komanda REP MOVSB, nenaudodami pakartojimo prefikso.**

4 **Duotas programos fragmentas:**

```
MOV     AX,080h
MOV     CL,3
SHL     AX,CL
PUSH    BP
MOV     BP,SP
PUSHF
OR      [BP-2],AX
POPF
POP     BP
PUSH    DS
POP     ES
MOV     SI,03Eh
MOV     DI,04Eh
MOV     CX,0A
REP     MOVSB
```

Apskaičiuokite registrų DI ir SI sumą po fragmento įvykdymo.

5 **Duotas programos fragmentas:**

```
PUSH DS
POP ES
MOV SI,01Ah
MOV DI,SI
INC SI
```

```
MOV CX,6
CLD
REP CMPSB
```

Duomenų segmente ląstelėse su adresais nuo 01Ah iki 01Fh yra reikšmės 1,2,4,5,6. Apskaičiuokite registrų CX ir DI sumą po paskutinės fragmento komandos.

4.7. Valdymo perdavimo komandos

Valdymo perdavimo komandos naudojamos programų išsišakojimui ir ciklų organizavimui, o taip pat ir paprogramių iškvietimui bei grįžimui iš jų. Kaip jau žinote, Intel 8088 architektūroje naudojama atminties segmentacija. Pagal tai valdymo perdavimai skiriami į perdavimus segmento viduje ir perdavimus už segmento ribų. Kai vykdomas valdymo perdavimas segmento viduje, koreguojamas tik registras IP. Kai vykdomas valdymo perdavimas už segmento ribų, kartu su IP koreguojamas segmentinis registras CS.

Komanda **CALL OP** išsaugo grįžimo adresą steke ir perduoda valdymą paprogramei, kurios adresą nusako operandas OP. Komanda gali vykdyti artimą arba tolimą valdymo perdavimą, atitinkamai grįžimo adresas yra IP arba CS:IP. Tiek tolimo valdymo perdavimo komanda, tiek artimo valdymo perdavimo komanda gali naudoti tiesioginį arba netiesioginį adresavimo būdą. Pirma, panagrinėkime komandą, kuri naudoja tiesioginį valdymo perdavimą. Tokia komanda, užrašyta asemblerio kalbam, atrodo taip **CALL LABEL**, kur LABEL nusako iškviečiamą paprogramę. Direktyva **FAR** prieš nusako asembleriui generuoti tolimą valdymo perdavimo komandą. Panagrinėkime, kaip vykdoma komanda:

1. Jeigu vykdomas tolimas valdymo perdavimas, į steką įrašoma CS reikšmė.
2. Į steko viršūnę įrašoma registro IP reikšmė.
3. Jeigu vykdomas tolimas valdymo perdavimas, į registrą CS įrašomas antras žodis, esantis komandoje.
4. Jeigu vykdomas tolimas valdymo perdavimas, pirmas žodis, esantis komandoje, nusiunčiamas į registrą IP. Jeigu vykdomas artimas valdymo perdavimas, žodis, esantis komandoje, pridedamas prie registro IP.

Dabar panagrinėkime paprogramių iškvietimo komandą, kuri naudoja netiesioginį adresavimo būdą. Komandos mnemonika yra ta pati – CALL. Netiesioginio adresavimo atveju šiek tiek keičiasi komandos veikimo algoritmas. Komandoje nurodytas operandas traktuojamas, ne kaip CS arba IP reikšmė, bet kaip atminties

ląstelės adresas, kur saugomas kviečiamos paprogramės adresas. Paprogramės iškvietimo komanda, kuri naudoja artimą netiesioginį valdymo perdavimą, pagal adresavimo baitą apskaičiuoja efektyvų adresą ir tuo adresų esantį žodį įrašo į registrą IP (aišku, prieš tai komanda išsaugo grįžimo adresą steke).

Paprogramės tolimo iškvietimo komanda su netiesioginiu adresavimo būdu steke išsaugo grįžimo adresą (CS ir IP) ir pagal adresavimo baitą apskaičiuoja efektyvų adresą. Adresu EA esantis žodis nusiunčiamas į registrą IP, o adresu EA+2 esantis žodis nusiunčiamas į registrą CS. Pateiksime komandos CALL užrašymo pavyzdžius asemblerio kalboje:

CALL FAR PTR PROC1 – tolimas tiesioginis iškvietimas;

CALL NEAR PROC2 – artimas tiesioginis iškvietimas

CALL SI – artimas netiesioginis iškvietimas (registro SI reikšmė nusiunčiama į IP);

CALL WORD PTR [BP+02] – artimas netiesioginis iškvietimas;

CALL DWORD PTR [BX+DI] – tolimas netiesioginis iškvietimas.

Komanda **RET** skirta grįžimui iš paprogramės. Komandai RET yra keturi skirtingi variantai – du tolimiems iškvietimams ir du artimiems. Komandos RET atveju nėra tiesioginio arba netiesioginio varianto – grįžimo adresas yra ten, kur jį įrašė kviečianti programa, t.y. steke. Panagrinėkime grįžimą iš artimos paprogramės. Pirmas variantas – tiesiog **RET**. Šiuos komandos veikimas yra priešingas CALL komandai, komanda iš steko viršūnės paima žodį ir įrašo jį į registrą IP. Antras variantas - **RET OP**, kur OP yra betarpiškas operandas, 16 bitų dydžio. Antras variantas vadinamas grįžimu su steko išlyginimu. Po to kai iš steko viršūnės buvo paimtas grįžimo adresas, komanda prie SP prideda operandą OP. Šitie variantai gali būti naudojami aukšto lygio programavimo kalbų kompiliatoriams. Pavyzdžiui, PASCAL kalboje už steko išlyginimą atsako iškviečiama procedūra arba funkcija, todėl grįžimui iš paprogramės kompiliatorius panaudos komanda **RET OP**. O, pavyzdžiui, C kalboje už steko išlyginimą atsako kviečianti programa, todėl C kompiliatorius grįžimui iš paprogramės panaudos komanda **RET**.

Tolimui iškvietimui skirtos RET komandos taip pat yra su arba be steko išlyginimo. Komandos veikia panašiai į savo artimus analogus, tik po IP nuskaitymo iš steko viršūnės dar nuskaito CS reikšmę.

Besąlyginio valdymo perdavimo komandos užrašomos mnemonika **JMP LABEL**. Komanda perduoda valdymą komandai, kurią nurodo komandoje esantis operandas. Kai komandoje naudojamas tiesioginis adresavimo būdas, betarpiško operando dydis gali būti 1, 2 arba 4 baitai. Kai operandas yra 1 baito, komanda vadinama – "vidinis artimas perdavimas", t.y. baitas komandoje traktuojamas kaip skaičius su ženklu, kuris pridedamas prie registro IP. Tokiu būdu valdymą galima perduoti per 126 atgal, arba per 127 į priekį. Kai reikia perduoti valdymą segmento viduje, naudojamas vidinis tiesioginis perdavimas – komandoje nurodytas betarpiškas operandas yra 16 bitų ir jo reikšmė pridedama prie registro IP. Išorinis tiesioginis naudojamas tuo atveju, kai reikia perduoti valdymą į kitą segmentą. Šiuo atveju komandoje esantis antras žodis įrašomas į registrą CS, o pirmas į registrą IP.

Taip pat besąlyginio valdymo perdavimo komanda JMP naudoja ir netiesioginį adresavimo būdą. Vidinis netiesioginis ir išorinis netiesioginis valdymo perdavimai veikia panašiai, kaip ir paprogramių netiesioginiai iškvietimai, tik nėra išsaugomas grįžimo adresas.

4.8. Sąlyginis valdymo perdavimas

Sąlyginio valdymo perdavimo komandos dažnai naudojamos ciklų organizavime. Sąlyginio valdymo perdavimo komandos vykdo vidinį artimą valdymo perdavimą priklausomai nuo tam tikrų požymių. T.y. komandoje yra betarpiškas 8 bitų operandas, kuris traktuojamas kaip skaičius su ženklu. Jeigu sąlyga (tam tikrų požymių kombinacija) yra teisinga, komanda prie registro IP prideda skaičių, priešingu atveju vykdoma sekanti komanda. Daugeliui sąlyginio valdymo perdavimo komandoms yra dvi mnemonikos.

Mnemonika	Paaiškinimas	Sąlyga
JG/JNLE	Daugiau/Ne (mažiau arba lygu) skaičiams su ženklu	(SF xor OF) and ZF=0
JNG/JLE	Ne daugiau/Mažiau arba lygu skaičiams su ženklu	(SF xor OF) and ZF=1
JL/JNGE	Mažiau/Ne (daugiau arba lygu) skaičiams su ženklu	SF xor OF=1
JNL/JGE	Ne mažiau/Daugiau arba lygu skaičiams su ženklu	SF xor OF=0

JA/JNBE	Daugiau/Ne (mažiau arba lygu) skaičiams be ženklo	CF and ZF=0
JNA/JBE	Ne daugiau/mažiau arba lygu skaičiams be ženklo	CF and ZF=1
JB/JNAE/JC	Mažiau/Ne (daugiau arba lygu)	CF=1
JNB/JAE/JNC	Ne mažiau/Daugiau arba lygu skaičiams be ženklo	CF=0
JE/JZ	Lygu	ZF=1
JNE/JNZ	Nelygu	ZF=0
JS	Neigiamas rezultatas	SF=1
JNS	Teigiamas rezultatas	SF=0
JO	Yra perpildymas	OF=1
JNO	Nėra perpildymo	OF=0
JP/JPE	Lyginis rezultatas	PF=1
JNP/JPO	Nelyginis rezultatas	PF=0
JCXZ	Nulinis skaitliukas	CX=0

Sąlyginio valdymo perdavimo komandos dažniausia naudojamos iškart po komandos CMP. Šių komandų pagalba (sąlyginis valdymo perdavimas ir palyginimas) galima sukonstruoti sąlygos operatorių, atitinkantį PASCAL kalbos sakinį **IF** *sąlyga* **THEN** *veiksmai* **ELSE** *veiksmai* ir ciklų sakinius. Sąlygos operatorių galime užrašyti taip:

```
        CMP    AL,0
        JZ     IsZero
        ...
        JMP    out1
    IsZero:
        ...
    Out1:
```

Šiuo atveju sąlyga yra “registro AL reikšmė nelygi nuliui”.

Ciklo sakinys **WHILE** *sąlyga* **DO** *veiksmai* gali būti užrašyta tokiomis komandomis:

```
    START:
        CMP    AL,0
        JZ     OUT1
        ...
        JMP    START
OUT1:
```

Šiuo atveju sąlyga taip yra “registro AL reikšmė nelygi nuliui”.

Ciklas **FOR** i:=1 **TO** 10 **DO** *veiksmai* asemblerio kalboje gali atrodyti taip:

```
    MOV   AL,1
START:
    CMP   AL,10
    JA    OUT1
    ...
    INC   AL
    JMP   START
OUT1:
```

Pratimai

1 **Duotas programos fragmentas**:

```
    MOV   AL,VAL1
    MOV   AH,VAL2
    CMP   AL,AH
    JP    OUT1
    MOV   AL,0
OUT1:
```

Apskaičiuokite registro AL reikšmę, kai duotos VAL1 ir VAL2 reikšmės.

1.1 VAL1=13, VAL2=15.
1.2 VAL1=127, VAL2=128.
1.3 VAL1=236, VAL2=110.
1.4 VAL1=-15, VAL2=-122.
1.5 VAL1=-66, VAL2=37.
1.6 VAL1=-127, VAL2=129.
1.7 VAL1=80h, VAL2=33h.
1.8 VAL1=212, VAL2=121.
1.9 VAL1=65, VAL2=-9.
1.10 VAL1=210, VAL2=198.
1.11 VAL1=255, VAL2=54.
1.12 VAL1=-118, VAL2=93.
1.13 VAL1=220, VAL2=127.
1.14 VAL1=33, VAL2=31.
1.15 VAL1=68, VAL2=-94.
1.16 VAL1=178, VAL2=187.
1.17 VAL1=19, VAL2=-17.
1.18 VAL1=66, VAL2=77.
1.19 VAL1=0, VAL2=84.
1.20 VAL1=36, VAL2=-55.

2 Duotas programos fragmentas:

```
    MOV   AL,VAL1
    MOV   AH,VAL2
    CMP   AL,AH
    JNLE  OUT1
    MOV   AL,0
OUT1:
```

Apskaičiuokite registro AL reikšmę, kai duotos VAL1 ir VAL2 reikšmės.

2.1 VAL1=33, VAL2=12.
2.2 VAL1=47, VAL2=-66.
2.3 VAL1=-12, VAL2=-128.
2.4 VAL1=64, VAL2=112.
2.5 VAL1=78, VAL2=21.
2.6 VAL1=-11, VAL2=-21.
2.7 VAL1=-64, VAL2=-34.
2.8 VAL1=66h, VAL2=82.
2.9 VAL1=37h, VAL2=55.
2.10 VAL1=48h, VAL2=80h.
2.11 VAL1=198, VAL2=133.
2.12 VAL1=129, VAL2=127.
2.13 VAL1=166, VAL2=132.
2.14 VAL1=199, VAL2=13.
2.15 VAL1=88, VAL2=164.
2.16 VAL1=25, VAL2=-128.
2.17 VAL1=127, VAL2=144.
2.18 VAL1=255, VAL2=120.
2.19 VAL1=55, VAL2=221.
2.20 VAL1=121, VAL2=176.

3 Užrašykite ciklą REPEAT veiksmai UNTIL sąlyga, kur sąlyga yra "registro AL reikšmė nelygi nuliui".

4 Realizuokite komandą AAA kitomis komandomis.

5 Realizuokite komandą DAA kitomis komandomis.

6 Duotas programos fragmentas

```
CS:0000: B00C      MOV     AL,VAL1
CS:0002: B420      MOV     AH,VAL2
CS:0004: 3AC4      CMP     AL,AH
CS:0006: 7CF8      JL      OUT1
```

Apskaičiuokite sekančios vykdomos komandos po paskutinės komandos fizinį adresą.

6.1 VAL1=13,VAL2=80,CS=1200h.
6.2 VAL1=65,VAL2=78, CS=32EFh.
6.3 VAL1=123,VAL2=129,CS=0665h.
6.4 VAL1=127,VAL2=129,CS=8847h.
6.5 VAL1=41,VAL2=198,CS=22ADh.
6.6 VAL1=22,VAL2=98,CS=782Eh.
6.7 VAL1=198,VAL2=-2,CS=6652h.
6.8 VAL1=84,VAL2=120,CS=1142h.
6.9 VAL1=64,VAL2=127,CS=44AFh.
6.10 VAL1=197,VAL2=-16,CS=111Ah.
6.11 VAL1=88,VAL2=12h,CS=6654h.
6.12 VAL1=33h,VAL2=81h,CS=245Eh.
6.13 VAL1=12,VAL2=21,CS=77EFh.
6.14 VAL1=12,VAL2=-7,CS=3321h.
6.15 VAL1=166,VAL2=61,CS=33C1h.
6.16 VAL1=5,VAL2=123,CS=945Eh.
6.17 VAL1=55,VAL2=135,CS=774Bh.
6.18 VAL1=222,VAL2=-12,CS=1111h.
6.19 VAL1=166,VAL2=66,CS=66FFh.
6.20 VAL1=151,VAL2=78,CS=881Dh.

4.9. Ciklų komandos

Ciklų komandos naudojamos **FOR** tipo ciklams organizuoti. Ciklų komandos labai panašios į sąlyginio valdymo perdavimo komandas – vykdomas vidinis artimas valdymo perdavimas, kuris priklauso nuo tam tikrų sąlygų. Ciklų komandos yra **LOOP LABEL**, **LOOPE/LOOPZ LABEL** ir **LOOPNE/LOOPNZ LABEL**. Komanda LOOP priklauso tik nuo registro CX – ciklo skaitliuko. Komandos veikimas atrodo taip:

1. Registro CX reikšmė sumažinama vienetu.
2. Jeigu CX<>0, sekančios komandos adresas gaunamas pridedant komandos operandą (8 bitų skaičius su ženklu) prie IP.
3. Jeigu CX reikšmė yra 0, vykdoma sekanti komanda.

Čia svarbu pastebėti, kad kai CX reikšmė yra 0 prieš vykdant komandą LOOP, ciklas vykdomas 0FFFFh kartų. Tokiai situacijai išvengti naudojama sąlyginio valdymo perdavimo komanda JCXZ.

Komandos LOOPE ir LOOPNE veikia panašiai, tik prie sąlygos CX<>0 dar prisideda sąlygos atitinkamai ZF=1 ir ZF=0.

Komandos LOOP pagalba galima nesudėtingai realizuoti ciklą **FOR** I:=N **DOWNTO** 1 **DO** *veiksmai*, kur N įrašomas į registrą CX. Pavyzdžiui,

```
    MOV   CX,N
START:
    ...
    LOOP  START
```

Čia reikia atkreipti dėmesį į tai, kad kai CX reikšmė prieš komandą LOOP yra 0, tai ciklo kūnas bus įvykdytas ne 1 kartą, o 10000h kartų – pirma sumažinamas registras CX ir tik po to jo reikšmė lyginama su nuliu. Šiuo atveju sumažinus CX vienetu gausime reikšmę 0FFFFh. Tokiai situacijai pastebėti naudojama sąlyginio valdymo perdavimo komanda JCXZ, kuri iškviečiama prieš ciklą.

Pratimai

1 Duotas programos fragmentas:

```
    MOV   CX,VAL1
    MOV   AX,VAL2
    SUB   CH,AL
    MOV   CL,8
A2:
    INC   AL
    LOOP  A2
```

Apskaičiuokite registro AL reikšmę po ciklo.

1.1 VAL1=32h,VAL2=56h.
1.2 VAL1=100h,VAL2=50h.
1.3 VAL1=0FFFFh,VAL2=0FFh.
1.4 VAL1=657,VAL2=321.
1.5 VAL1=4415,VAL2=774.
1.6 VAL1=1124,VAL2=336.
1.7 VAL1=4477,VAL2=-654.
1.8 VAL1=87,VAL2=12554.
1.9 VAL1=2214,VAL2=11.
1.10 VAL1=125,VAL2=256.
1.11 VAL1=665,VAL2=124.
1.12 VAL1=255,VAL2=128.
1.13 VAL1=256,VAL2=127.
1.14 VAL1=6541,VAL2=60541.
1.15 VAL1=1471,VAL2=112.
1.16 VAL1=2214,VAL2=33654.
1.17 VAL1=2214,VAL2=2254.
1.18 VAL1=587,VAL2=1452.
1.19 VAL1=1144h,VAL2=3321h.
1.20 VAL1=4475,VAL2=124.

4.10. Pertraukimų komandos

Pertraukimo komandos skirtos programiniams pertraukimams generuoti. Kai procesorius aptinka pertraukimo komandą, generuojamas vidinis pertraukimas su nurodytu kodu. Pertraukimo komanda atrodo taip – **INT N**, kur N yra pertraukimo numeris. Pertraukimas su kodu 3 (sustojimo taško pertraukimas) asemblerio kalboje užrašomas **INT 3**, o asembleriu verčiamas į mašininį kodą 0CCh. Pertraukimas su kodu 4 asemblerio kalboje užrašomas **INTO**.

Panagrinėkime komandos **INT N** veikimo schemą:

1. Į steko viršūnę įrašoma požymių registro reikšmė.

2. Į steko viršūnę įrašoma registro CS reikšmė.
3. Į steko viršūnę įrašoma registro IP reikšmė.
4. Požymiai IF ir TF nustatomi į 0.
5. Į registrą CS įrašoma reikšmę iš atminties adresu N*4+2.
6. Į registrą IP įrašoma reikšmė iš atminties adresu N*4.

Po 5 ir 6 žingsnių perduodamas valdymas pertraukimo apdorojimo procedūrai. Paskutinė ISR komanda yra **IRET** – grįžimas iš pertraukimo apdorojimo procedūros. Komanda IRET veikia priešingai – grįžimo adresą įrašo į registrus CS ir IP ir atstato požymių registro reikšmę.

4.11. Procesoriaus būsenos valdymas

Kaip jau minėjome nėra galimybės tiesiogiai pakeisti požymių registro reikšmės, tačiau galima tai padaryti per steką (POPF pagalba), arba naudojant komandas, kurios keičia tam tikrus registro bitus. Tokios komandos priskiriamos prie procesoriaus būsenos valdymo komandų.

Komanda **CLC** nustato požymį CF į 0.

Komanda **STC** nustato požymį CF į 1.

Komanda **CMC** keičia CF reikšmę į priešingą

Komanda **CLD** nustato požymį DF į 0.

Komanda **STD** nustato požymį DF į 1.

Komanda **CLI** nustato požymį IF į 0, t.y. uždraudžia išorinius maskuojamus pertraukimus.

Komanda **STI** nustato požymį IF į 1.

Kitos procesoriaus būsenos valdymo komandos nekeičia požymių registro. Komanda **HLT** perveda procesorių į sustojimo būseną. Kai iškviesta komanda ir IF yra 0, procesorių iš būsenos gali išvesti tik nemaskuojamas pertraukimas arba RESET signalas. Pavyzdžiui, po komandų **CLI;HLT;** procesorių galima “atgaivinti” tik per RESET signalą.

Komanda **WAIT** skirta procesoriaus sinchronizavimui su kitais įrenginiais.

Komanda **LOCK** skirta naudoti kompiuteriuose su keliais procesoriais – komanda uždraudžia priėjimą prie magistralės kitiems procesoriams.

5. Programavimo sistema

5.1. Asemblerio kalba

Asemblerio kalba – žemesnio lygio programavimo kalba, maksimaliai priartinta prie mašininių kodų. Asemblerio kalboje naudojamos mnemonikos vietoj mašininių kodų ir simboliniai vardai duomenims ir ląstelių adresams užrašyti.

Iki šiol pateikti programų fragmentai buvo aprašomi tik mnemonikomis. Išbaigtoms programoms užrašyti reikia papildomai naudoti specialias konstrukcijas. Pilnos programos tekstas (arba kodas) apdorojamas asembleriu, kuris perveda mnemonikas į mašininius kodus, parenka geriausiai tinkančius komandų formatus, apskaičiuoja poslinkius valdymo perdavimo komandoms ir t.t. Rezultate gauname modulį, užrašytą specialia objektine kalba – objektinį modulį. Objektinis modulis tai dar nėra programa, paruošta vykdymui. Objektiniame module tam tikrų komandų operandai lieka neužpildyti (vėliau panagrinėsime kokie yra tie operandai ir kodėl asembleris negali juos užpildyti). Objektinis modulis susideda iš galvos ir kūno – galvoje yra modulį aprašanti informacija (kiek yra segmentų, kokie segmentų ilgiai, neužpildyti operandai ir t.t.), kūne yra modulio segmentai su komandomis ir duomenimis.

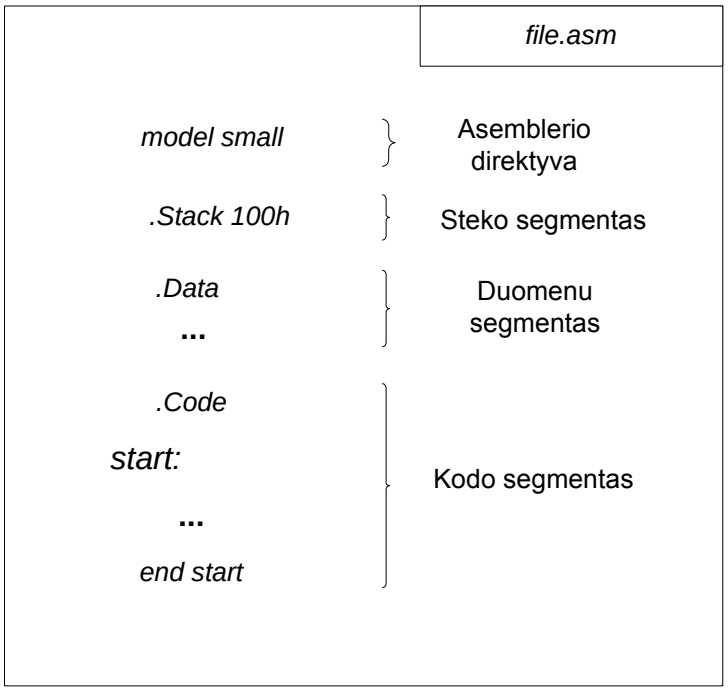
Ryšių redaktorius iš vieno arba kelių modulių suriša vykdomą failą. Ryšių redaktorius apdoroja programoje aprašytus segmentus ir priklausomai nuo objektiniame modulyje esančių nurodymų išdėsto programos segmentus. Vykdomas failas savo ruožtu susideda iš galvos ir kūno. Galvoje esanti informacija aprašo vykdomojo failo parametrus.

Tam, kad pradėti vykdyti programą, reikia perduoti vykdomą failą programai-pakrovėjui (loader). Pakrovėjas yra operacinės sistemos dalis. Pakrovėjas nuskaito vykdomo failo galvą ir nustato kiek atminties reikia programai ir į kokią atminties vietą reikia įrašyti programą. Pakrovėjas įrašo programą į atmintį, nustato segmentinių registrų reikšmes ir perduoda valdymą pirmai programos komandai.

Bendra programos struktūra

Kaip jau žinome, Intel 8088 architektūroje naudojama atminties segmentacija. Programos atvaizduoja šitą architektūros savybę programos kode. Programa asemblerio kalba susideda iš segmentų, asemblerio direktyvų, žymių ir mnemonikų. Programos tekstai saugomi failuose su išplėtimu *asm*. Direktyvos nurodo asembleriui,

kaip sugeneruoti objektinį modulį. Programoje gali būti vienas, arba daugiau segmentų. Bendra programos struktūra atrodo taip:



5.1.1 pav. Bendra programos struktūra

Programa logiškai suskirstyta į 3 segmentus – kodo segmentą, duomenų segmentą ir steko segmentą. Programuojant operacinėje sistemoje MS-DOS parenkamas tam tikras atminties modelis. Pavyzdžiui, .exe tipo programoms dažniausia parenkamas modelis *small*. Šitame modelyje programa gali naudotis atskirais segmentais kodui, duomenims ir stekui. Modelyje *tiny* programos duomenis, kodas ir stekas yra viename segmente. Toks modelis naudojamas .com tipo programose.

Steko segmentas dažniausia naudojamas grįžimo ir paprogramių adresams laikyti. Bet jeigu programoje ir nėra paprogramių iškvietimo komandų (*call*) ir komandų, dirbančių su steku (push, pop ir t.t.), reikia apibrėžti steko segmentą. Bet kurio programos vykdymo metu gali įvykti pertraukimas ir į steką bus įrašytas grįžimo adresas ir požymių registro reikšmė. Steko segmentas deklaruojamas direktyva *.Stack*, po kurios eina steko dydis baitais. Šita reikšmė bus įrašyta į registrą SP, kai pakrovėjas ruošia programą vykdymui. Pavyzdžiui, direktyva *.Stack 100h* apibrėžia steko segmentą, kurio dydis yra 256 baitų.

Duomenų segmentas deklaruojamas direktyva *.Data*. Duomenų segmente apibrėžiami duomenys, kuriais operuoja programa. Duomenys apibrėžiami

specialiomis duomenų apibrėžimo direktyvomis. Bendras duomenų deklaracijos sakinys:

[žymė] dx reikšmė[,reikšmė]*,

kur dx viena iš db,dw,dd,dq,dt direktyvų. Direktyvos antroji raidė atitinka reikšmės dydį: b – baitas, w – žodis, d – dvigubas žodis (keturi baitai), q – aštuoni baitai, t – dešimt baitų. Pavyzdžiui,

```
(1)          db    1
(2)a         db    ?
(3)b         db    1,2,3
(4)msg1      db    'eilute'
(5)array1    db    10 dup (0).
```

Eilutėje (1) duomenų segmente deklaruojamas vienas baitas, kurio pradinė reikšmė yra 1; eilutėje (2) apibrėžiamas vienas baitas, kurio pradinė reikšmė nėra apibrėžta, programoje šitą baitą galima adresuoti pagal vardą *a*; eilutėje (3) apibrėžiama 3 baitų seka, kurios elementų pradinės reikšmės yra 1,2 ir 3, pirmą sekos baitą galima adresuoti pagal vardą *b*; eilutėje (4) apibrėžiama 6 baitų seka ASCII kodais; eilutėje (5) apibrėžiama 10 baitų seka, kurios elementų pradinės reikšmės yra 0.

Direktyva dw apibrėžia atmintyje dviejų baitų dydžio elementą, direktyva dd – keturių baitų, pvz.:

```
(1)a   db    1
(2)b   dw    1
(3)c   dd    1
```

adresu a yra vienas baitas, kurio pradinė reikšmė yra 01h, adresu b apibrėžiamas žodis, kurio reikšmė yra 0001h, adresu c apibrėžiamas dvigubas žodis, kurio pradinė reikšmė yra 00000001h.

Kodo segmente yra komandos, kurios bus vykdomos po to, kai pakrovėjas perduos valdymą į programos įėjimo tašką.

Pateiktame pavyzdyje yra viena asemblerio direktyva ir 3 segmentai. Pirmas programos segmentas yra steko segmentas. Steko segmentas deklaruojamas direktyva *.Stack*. Skaičius, esantis po direktyvos nurodo programos steko dydį. Šita reikšmė bus įrašyta į registrą SP, kai pakrovėjas pakraus vykdomą failą į atmintį. Steko dydis negali viršyti 64K.

Duomenų segmentas deklaruojamas asemblerio direktyva *.Data*. Duomenų segmento dydį apskaičiuoja asembleris programos transliavimo į objektinį modulį metu. Duomenų segmente aprašomi programos duomenys. Asemblerio kalboje nėra

mums įprastų aukšto lygio programavimo kalbų duomenų tipų (int, float ir t.t.). Visi kintamieji duomenų segmente apibrėžiami direktyvomis db (*define byte*), dw (define word), dd (define double word), dq (define quadword), dt (define ten bytes).

Direktyva *db* yra viena iš dažniausiai naudojamų duomenų deklaravimo direktyvų. Direktyva deklaruoja viena baitą arba baitų seką. Prieš direktyvą galima nurodyti žymę, tokiu atveju programoje galima kreiptis į atmintyje išskirtą baitą pagal žymės vardą. Bendras direktyvos užrašymo būdas – [žymė] db []

Kodo segmente užrašomos komandos, kurios ir sudaro programą. Kodo segmentas deklaruojamas direktyva *.Code*. Bendras komandų užrašymo būdas yra [žymė] komanda [operandas1[,operandas2]] [;komentaras]. Kiekvienoje eilutėje gali būti tik viena komanda. Pavyzdžiui,

```
        stc       ; set carry      (1)
        ret       4                (2)
        mov       ax,0245h         (3)
label1: add       al,3             (4)
```

Eilutėje (1) pateikta komanda be operandų su komentaru, eilutėje (2) – komanda su vienu operandu, eilutėje (3) yra komanda su dviem operandais, eilutėje (4) yra žymė *label1* ir komanda su dviem operandais. Komandos yra mašininių kodų mnemonikos, operandai – registrai, betarpiški operandai arba operandai atmintyje.

Kodo segmente reikia nurodyti programos įėjimo tašką, t.y. komandą kuriai pakrovėjas perduoda valdymą. Įėjimo taškui nurodyti reikia dviejų žingsnių – 1) prie komandos, kuri turi būti vykdoma pirma užrašoma žymė, pvz., *_start:*, 2) programos teksto gale užrašoma direktyva *end* su žyme, kuri rodo į įėjimo tašką, pvz., *end _start*.

Užrašykime pilna paprastos programos kodą:

```
.model small                                                    (1)
.Stack 100h                                                     (2)

.Data                                                           (3)
        msg1    db      'Labai paprasta programa',13,10,'$'     (4)

.Code                                                           (5)

_start:                                                         (6)
        mov     ax,@data        ;                               (7)
        mov     ds,ax           ;       Init data segment       (8)
        mov     ah,9            ;                               (9)
        mov     dx,offset msg1  ;       call output string      (10)
        int     21h             ;                               (11)
        Exit:                                                   (12)
        mov     ah,4Ch          ;       exit to DOS             (13)
        int     21h                                             (14)
END     _start                                                  (15)
```

Pateiktoje programoje duomenų segmente apibrėžiama simbolių seka (4). Programos įėjimo taškas apibrėžiamas žyme *_start* (6),(15).

Ruošiant programą vykdymui, pakrovėjas įrašo reikšmes į registrus SS,SP, CS ir IP. Registrui DS reikšmė priskiriama programos viduje. Tam panaudojamas užrezervuotas žodis *@data* (7-8). Komanda (7) įrašo paragrafo, kuriame prasideda duomenų segmentas, numerį į registrą AX. Šita reikšmė taps žinoma tik pakrovėjo veikimo metu. Todėl asembleris, transliuojant šitą komandą, paliks komandos operandą neužpildytu ir objektinio modulio galvoje pažymės, kad šitą lauką reikia užpildyti pakrovėjui. Jeigu iš programos pašalinti eilutes (7-8), tai programai taps nebeprieinamas duomenų segmentas, t.y. negalėsime adresuoti duomenų.

Atkreipkime dėmesį į eilutes (13-14). Šitos komandos atsako už valdymo perdavimą operacinei sistemai. Fragmente iškviečiama operacinės sistemos funkcija su numeriu 9h, kas baigti programos vykdymą. Jeigu šitų komandų nebus programoje, tai programa nebus užbaigta ir procesorius vykdys visas komandas iš eilės kodo segmente. Pavyzdžiui, tarkime programos kodo segmentas yra paragrafe su numeriu 3A00h.

```
3A00:0000 : B8003B          mov     ax,3B00 ;"  "
3A00:0003 : 8ED8            mov     ds,ax
3A00:0005 : B409            mov     ah,009 ;" "
3A00:0007 : BA0000          mov     dx,00000 ;"  "
3A00:000A : CD21            int     021
3A00:000C : B44C            mov     ah,04C ;"L"
3A00:000E : CD21            int     021
3A00:0010 : 4C              dec     sp
3A00:0011 : B004            mov     al,004 ;" "
3A00:0013 : 41              inc     cx
3A00:0014 : C3              retn
3A00:0015 : 207061          and     [bx][si][00061],dh
```

Čia matome programą atmintyje. Paskutinė programos komanda prasideda adresu 3A00:000Eh, po jos eina baitų seka, kurią procesorius dekoduoja kaip komandas. Bet iš tikrųjų tai yra ne komandos, o "šiukšlės". Jeigu bus pašalintos komandos nuo 3A00:000C iki 3A00:0011, tai procesorius po komandos INT 21h (3A00:000Ah) pradės vykdyti komandą DEC SP (3A00:0010) ir t.t.

Paprogramės

Paprogramės naudojamos programavimo kalbose dažnai pasikartojančioms operatorių sekoms atikirimui. Paprogramių panaudojimas padeda sukurti gerai

struktūruotas programas, o tai padaro programos tekstą lengviau skaitomu ir sumažina programos palaikymo kaštus.

Į mikroprocesoriaus Intel 8088 komandų rinkinį įeina komandos darbui su paprogramėmis. Tai jau nagrinėtos komandos *call* ir *ret*.

Dabar panagrinėkime kokiomis konstrukcijomis asemblerio kalboje apibrėžiamos paprogramės. Asemblerio kalboje yra keli paprogramių apibrėžimo būdai. Dažnai, skirtumai tarp šitų būdų yra tik sintaksėje. Todėl apžvelgsime tik pagrindinius bruožus.

Paprogramės iškvietimui naudojama komanda *call*, kurioje nurodomas iškviečiamos paprogramės adresas (nebūtinai tiesioginis paprogramės adresas, pvz., kai naudojamas netiesioginis adresavimo būdas). Reiškia, naudoti kokį nors adresą, tam kad iškviesti paprogramę. Paprasčiausias variantas – naudoti programos žymes. Pavyzdžiui:

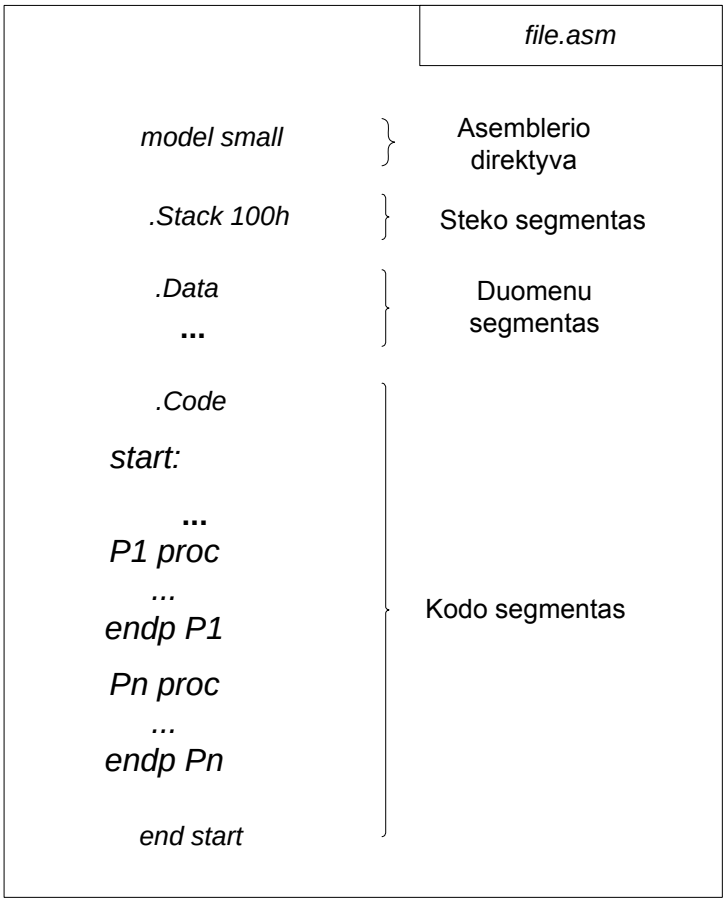
```
call        label1
...
label1:
{paprogramės kūnas}
ret
```

Čia labai svarbu nepamiršti komandos *ret*.

Kitas būdas – naudoti užrezervuotus žodžius *proc* ir *endp*. Šiuo būdu paprogramė apibrėžiama tokia konstrukcija:

```
name        proc
{paprogramės kūnas}
ret
name        endp
```

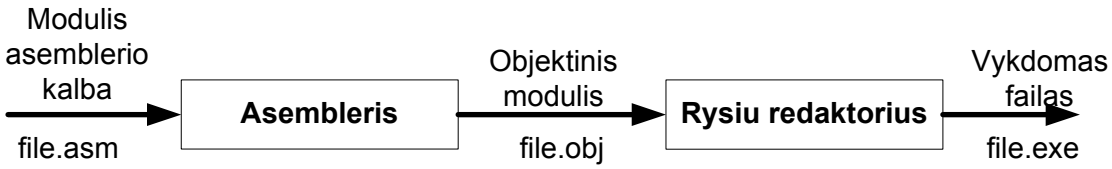
Toks būdas yra žymiai lankstesnis, nes leidžia naudoti modulinį programavimą ir ryšį su aukšto lygio programavimo kalbomis. Bendra programos struktūra su paprogramėmis atrodo taip:



5.1.2 pav. Bendra programos struktūra su paprogramėmis

5.2. Programų transliavimas

Programų, parašytų asemblerio kalba, transliavimas vykdomas pagal 5.2.1 schemą.



5.2.1 pav. Programos asemblerio kalba transliavimo schema

Programa asemblerio kalba saugoma faile (dažniausia su išplėtimu *asm*). Failas paduodamas asembleriui. Asembleris generuoja objektinį modulį. Objektinis modulis apdorojamas ryšių redaktoriumi, kuris generuoja vykdomą failą.

Panagrinėkime asemblerio ir ryšių redaktoriaus darbą detaliau. Asembleris nuskaito programos tekstą iš failo. Asembleris analizuoja kiekvieną eilutę ir konstruoja atitinkamus mašininius kodus, kuriuos užrašo į kitą failą – objektinį modulį. Po šito žingsnio asembleris baigia darbą.

Pagrindinė transliavimo schema yra paprasta. Transliavimo metu asembleriui reikia:

1) pakeisti mnemonikas mašininiais kodais;
2) pakeisti kintamųjų ir žymių vardus į atitinkamus adresus;
3) užrašyti duomenis dvejetainiu pavidalu.

Pavyzdžiui, panagrinėkime, kaip asembleris perveda komandą MOV B,67 į mašininius kodus. Tegu duomenų segmentas apibrėžtas taip:

```
.Data
a db 0
b db 2
```

Asembleris sugeneruos operacijos kodą C6h, adresavimo baitą 06h, poslinkį 0100 ir betarpišką operandą 43h.

Operacijos kodas parinktas pagal mnemoniką ir komandos formatą, t.y. adresavimo baito laukas REG naudojamas operacijos kodo išplėtimui – 1100011*w*. Šiuo atveju konfigūracinis bitas nustatytas į 0. Adresavimo baitas atrodo taip – MOD 000 R/M. Šiuo atveju 00000110_2.

Pažiūrėkime, kaip asembleris apskaičiuoja operando efektyvų adresą. Kintamojo b vardas turi būti pakeistas į poslinkį tame segmente, kur kintamasis yra apibrėžtas. Žymės adresas skaičiuojamas nuo 0. Mūsų atveju b adresas yra 0001h, nes adresu 0000h apibrėžtas kintamasis a, kurio dydis yra 1 baitas.

Programa gali susidaryti iš kelių objektinių modulių. Tam, kad programą sudarančius modulius surišti į vieną vykdomą failą naudojamas ryšių redaktorius. Pagrindinės ryšių redaktoriaus funkcijos yra:

1) Modulių apjungimas. Ryšių redaktorius turi apibrėžti kokia tvarka vykdomajame faile turi išsidėstyti programos segmentai. Ryšių redaktorius apjungia segmentus viename faile.
2) Ryšių tarp modulių redagavimas. Ryšių redaktorius turi pakeisti išorines nuorodas adresais. Tai yra pagrindinis redaktoriaus uždavinys.
3) Vykdomojo failo galvos konstravimas. Ryšių redaktorius negali pilnai sugeneruoti vykdomojo failo, nes kai kuri informacija yra žinoma tik programos pakrovimo į atmintį metu. Vykdomas failas susideda iš kūno ir galvos. Kūnas yra programos mašininiai kodai, o galva – informacija apie programą, kuri reikalingą pakrovėjui.

Programai įvykdyti reikia ją užrašyti į pagrindinę atmintį ir perduoti jai valdymą. Šituo uždaviniu užsiima pakrovėjas. Pagrindinės pakrovėjo funkcijos:

1) Programos pakrovimas. Pakrovėjas turi rasti vietą atmintyje ir užrašyti programos kodą į tą vietą.
2) Programos priryšimas. Pakrovėjas turi užbaigti programos transliavimą, t.y. užpildyti tuos laukus, kuriuos negali užpildyti ryšių redaktorius.
3) Programos paleidimas. Po priryšimo pakrovėjas perduoda valdymą į programos įėjimo tašką.

Pakrovėjas apskaičiuoja programos ilgį ir nustato, ar pakanka atminties programai pakrauti. Jeigu atminties neužtenka, pakrovėjas praneša apie tai ir baigia darbą. Jeigu atminties užtenka, pakrovėjas kopijuoja programos segmentus į atmintį. Po programos pakrovimo į atmintį pakrovėjui žinomas pradinis programos adresas, t.y. galima užbaigti programos transliavimą. Pakrovėjas apskaičiuoja segmentų pradžios adresus ir užrašo juos į neužpildytus laukus.

Po laukų užpildymo pakrovėjui reikia perduoti valdymą į programos įėjimo tašką. Tam reikia 1) nustatyti registrų SS ir SP reikšmes, kad programai būtų prieinamas steko segmentas, 2) užrašyti programos įėjimo taško adresą į registrus CS:IP.

Registro SP reikšmė ištraukiama iš vykdomojo failo galvos. Registro SS reikšmė apskaičiuojama pagal pakrautos programos adresą.

Perėjimas į programos įėjimo tašką įvykdomas pagal komandą *jmp far*. Įėjimo taško adresas taip pat apskaičiuojamas pagal vykdomojo failo galvoje esančias reikšmes ir pakrautos į atminį programos adresą.

Programa asemblerio kalba transliuojama į vykdomą failą naudojant asemblerį TASM ir ryšių redaktorių TLINK. Mūsų naudojami asembleris ir ryšių redaktorius yra konsolės interfeiso. Norint sutransliuoti programa asemblerio kalba, programoms komandinėje eilutėje reikia pateikti parametrus – failo vardą ir transliavimo parametrus.

Panagrinėkime, kaip naudotis asembleriu TASM. Bendrai parametrai, kuriuos reikia pateikti asembleriui atrodo taip: [PARAM] FILE.ASM [,OBJECT] [,LIST], kur [PARAM] yra nurodymas asembleriui kaip sukurti objektinį modulį, FILE.ASM – failas, kuriame įrašyta programa asemblerio kalba. [OBJECT] – objektinio modulio failo vardas, jeigu nenurodytas, tai failo vardas yra FILE.OBJ. [LIST] – objektinio

modulio transliavimo informacijos failo vardas, jeigu parametras nenurodytas, failo vardas yra FILE.LST. Pateiksime svarbiausių nurodymų asembleriui sąrašą:

/l – sukurti transliavimo informacijos failą;

/z – rodyti klaidingą eilutę

/zi – įjungti į objektinį failą derinimo informaciją

Pavyzdžiui, komanda >TASM FILE.ASM objektinį modulį FILE.OBJ, komanda >TASM FILE.ASM /l , OBJFILE.DDD sukurs objektinį modulį OBJFILE.DDD ir transliavimo informacijos failą FILE.LST.

Ryšių redaktoriui taip reikia pateikti parametrus komandinėje eilutėje. Ryšių redaktoriaus parametrų sąrašas atrodo taip: [PARAM] OBJFILE.OBJ [,EXEFILE], kur OBJFILE.OBJ yra objektinio modulio failo vardas. [PARAM] yra ryšių redaktoriaus nustatymai ir yra nebūtinas parametras. [EXEFILE] – yra generuojamo vykdomo failo vardas, jeigu parametras nenurodytas, tai vykdomo failo vardas bus OBJFILE.EXE (arba OBJFILE.COM).

Panagrinėkime svarbiausius nustatymus ryšių redaktoriui:

/t – generuoti COM tipo vykdomą failą;

/v – įjungti į vykdomą failą derinimo informaciją.

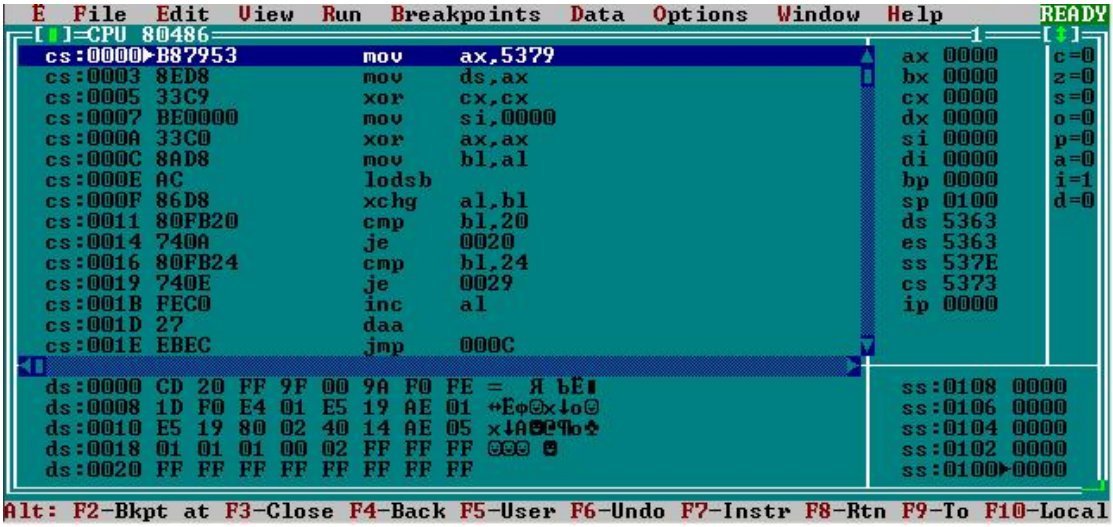
Pavyzdžiui, komanda >TLINK A1 sukurs vykdomą failą A1.EXE, o komanda >TLINK /t A1 sukurs vykdomą failą A1.COM.

5.3. Programų derinimas

Labai retai būna tokių paprastų programų, kurias galima parašyti be klaidų. Kai kurios klaidos aptinkamos asembleriu ir gali būti ištaisytos dar iki programos vykdymo. Tai yra sintaksės klaidos. Bet yra ir loginės klaidos, kai programos sintaksė yra taisyklinga, o programa veikia nekorektiškai. Tokio tipo klaidoms taisyti naudojamos derinimo programos (*debuggers*). Derinimo programos leidžia vykdyti programą po žingsniniu režimu ir stebėti programos ir aplinkos būsenos pasikeitimus.

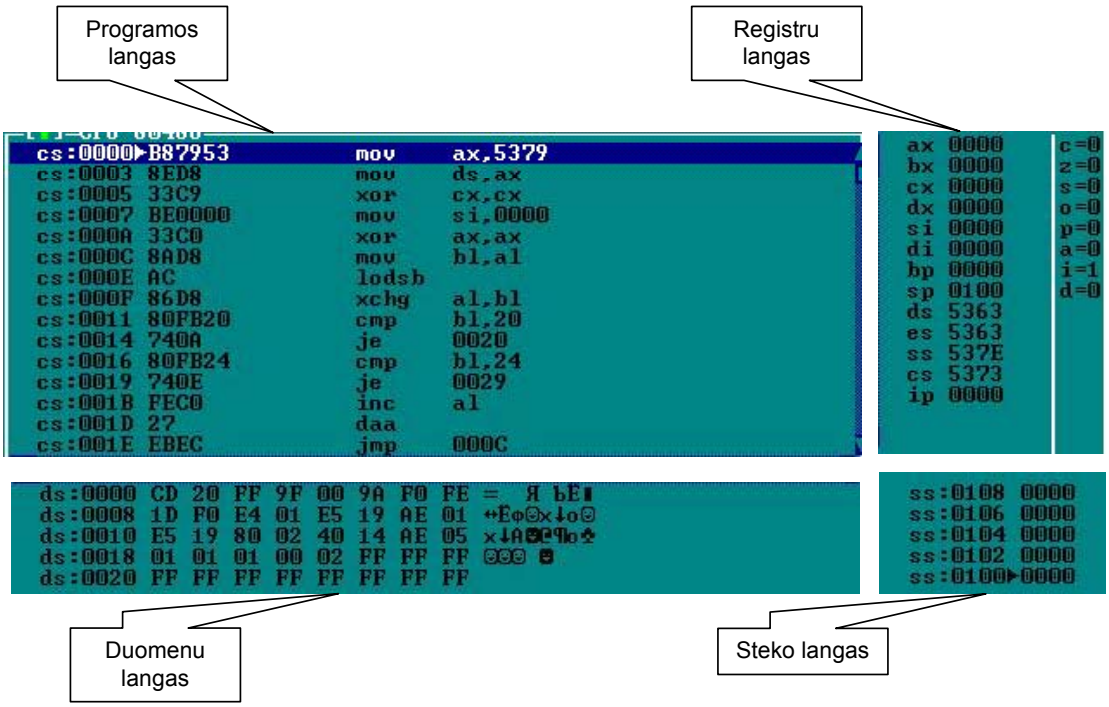
Asemblerio programoms mes naudosime derinimo programa TD (*Turbo Debugger*). Programa turi patogu vartotojo interfeisą ir turi daug naudingų funkcijų.

Programa paleidžiama iš komandinės eilutės, kur po programos pavadinimo TD reikia parašyti derinamos programos failo vardą. Derinimo programos langas atrodo taip:



5.3.1 pav. Bendras TD vaizdas

ir susideda iš keturių atskirų langų – programos kodo, registrų, steko ir duomenų:



5.3.2 pav. TD langų išsidėstymas

Programos lange matomas programos kodas, iš esmės tai yra einamojo kodo segmento turinys. Programos lango kairiajame stulpelyje pateikti programos mašininiai kodai su poslinkių kodo segmente, o dešiniajame stulpelyje – komandų mnemonikos. TD leidžia keisti programos kodo segmentą – paspaudus tarpą, atsidaro pažymėtos komandos keitimo langas. Lange įvedus komandą asemblerio kalba, TD pakeičia pažymėtą komandą į įvestą ir pažymi sekančią komandą. Toks funkcionalumas yra patogus tuo atveju, kai derinimo metu atsiranda poreikis pakeisti programos logiką, netransliuojant programos.

Registrų lange pateiktos registrų reikšmės ir požymių reikšmės. Kiekvieną registro reikšmę galima pakeisti – padidinti vienetu, sumažinti vienetu, įrašyti naują reikšmę arba įrašyti nulį. Taip galima keisti kiekvieno požymio reikšmę.

Steko lange rodoma steko viršūnė ir šalia esančios steko ląstelės. Steko lange galima pakeisti duomenis, esantis steke, pereiti prie kitų steko elementų. Steko langas yra patogus įrankis tuo atveju, kai reikia stebėti steko kitimą.

Duomenų langas rodo atminties ląstelių turinį duomenų segmente. Lange galima peržiūrėti ląsteles, esančias nurodytu poslinkiu duomenų segmente, galima peržiūrėti ląsteles ir kituose segmentuose, keisti atminties ląstelių reikšmes.

Panagrinėkime bendrą supaprastintą uždavinio sprendimo procesą. Pirma, suformuluojamas uždavinys, kuriame turi būti apibrėžti reikalavimai programai, nustatyti pradiniai duomenis ir programos veikimo rezultatas. Kai nustatyti programos reikalavimai, priimamas sprendimas – kaip išspręsti uždavinį. Kai sugalvotas sprendimo algoritmas, jis užrašomas programavimo kalba ir transliuojamas į vykdomą failą. Vykdomas failas praeina pro testų seką, kuriuos metu gali būti aptiktos programos klaidos. Šiame žingsnyje ir naudojamos derinimo programos. Reikia pastebėti, kad derinimo programos, šiuo atveju TD, neranda klaidų, o tik padeda jų ieškoti.

Derinimo programa leidžia vykdyti programą, lyg ji būtų paleista iš komandinės eilutės, arba vykdyti kiekvieną žingsnį atskirai. Kai paleidžiamas TD ir jam yra nurodyta programą, kurią reikia derinti, TD atlieka pakrovėjo funkcijas ir sustoja ties programos įėjimo taško. Nuo šito žingsnio prasideda programuotojo darbas su TD. Jeigu bus paleista vykdymui (F9 arba Run→Run), TD tiesiog įvykdys programą. Norint vykdyti po vieną programos komandą, reikia pasinaudoti žingsniniu režimu – kiekviena komanda įvykdoma paspaudžius F8 arba Run→”Step over”. Įvykdžius vieną komandą, TD sustoja ties sekančios komandos.

Tais atvejais, kai programos kodas yra pakankamai didelis (virš 1000 komandų), verta pasinaudoti sustojimo taškais (*Breakpoint*). Sustojimo taškai leidžia vykdyti programą įprastu režimu ir pristabdyti vykdymą nurodytoje vietoje. Norint apibrėžti sustojimo tašką komandai, reikia ją pažymėti ir įjungti sustojimo tašką (F2 arba Breakpoints→Toggle). Paleidus programą, ji bus pristabdyta šitame taške.

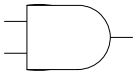
6. Mikroprogramininis lygis

Kaip jau minėjome, Intel 8088 mikroprocesorius yra CISC architektūros atstovas. T.y. asemblerio lygio komandos realizuotos ne aparatūriškai, o interpretuojamos mikroprogramomis. Jeigu pažiūrėsime į mikroprocesoriaus vidinę struktūrą (žr. 3.1.X), tai interpretavimo procesas vykdomas operacinio įrenginio mikroprogramininio valdymo įrenginiu. Mikroprogramos sudarytos iš mikrokomandų, kurios yra labai primityvios. Tai leidžia įvesti sudėtingas komandas į architektūrą, smarkiai nedidinant jos kainos.

6.1. Fizinio lygio komponentės

Šiuolaikinių kompiuterių aparatūra susideda iš elementų, kurie vadinami ventiliais. Ventilis realizuoti vieną iš loginių operacijų – OR, AND ir t.t. Dažniausia, ventilis turi du įėjimus ir vieną išėjimą. Bet aparatūroje naudojamas ir ventilis su vienu įėjimu – invertorius. Ventiliai realizuojami tranzistoriais. Pavaizduokime schematiškai ventilius ir pakomentuokime jų darbą.

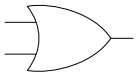
Ventilis AND realizuoja loginę operaciją AND ir schematiškai dažniausia žymimas tokiu simboliu:



6.1.1 pav. AND ventilio simbolis

Ventilis turi du įėjimus ir du išėjimus. Į įėjimus paduodami elektriniai signalai, išėjime taip pat yra elektrinis signalas. Aukštos įtampos signalą žymėsime vienetu, žemos įtampos signalą – nuliu. Tokiu žymėjimu patogu aprašyti logines funkcijas. Mūsų AND ventilis realizuoja loginės daugybos operaciją – kai bent vienas įėjimo signalas yra žemos įtampos – išėjime turime žemos įtampos signalą. Kai abu įėjimo signalai yra aukštos įtampos – išėjime turime aukštos įtampos signalą. Tokį ventilį galime sukonstruoti iš dviejų tranzistorių, sujungtų nuosekliai.

Ventilis OR realizuoja loginę operaciją OR ir schematiškai dažniausia žymimas tokiu simboliu:

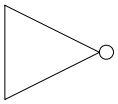


6.1.2 pav. OR ventilio simbolis

Ventilis taip pat turi du įėjimus ir vieną išėjimą. OR ventilis realizuoja loginę sudėtį, t.y. kai bent vienas įėjimo signalas yra aukštos įtampos išėjime turime aukštos

įtampos signalą. Kai abejuose įėjimuose yra nuliai, tai ir išėjime turime nulį. Tokį ventilį galime sukonstruoti iš dviejų lygiagrečiai sujungtų tranzistorių.

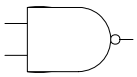
Ventilis NOT, dar vadinamas invertoriumi, schemose dažniausia žymimas simboliu:



6.1.3 pav. NOT ventilio simbolis

Invertorius turi vieną įėjimą ir vieną išėjimą. Invertorius pakeičia įėjimo signalo įtampą – kai įėjime yra 0, išėjime gauname vienetą ir atvirkščiai.

Dar vienas svarbus ventilis yra ventilis, kuris realizuoja loginę operaciją XOR ir ventilis NAND (NOT AND). Ventilių NAND kombinacija galima išreikšti visas svarbiausias logines funkcijas (žr. [Mit03, p.9]). Pavaizduokime ventilį NAND:



6.1.4 pav. NAND ventilio simbolis

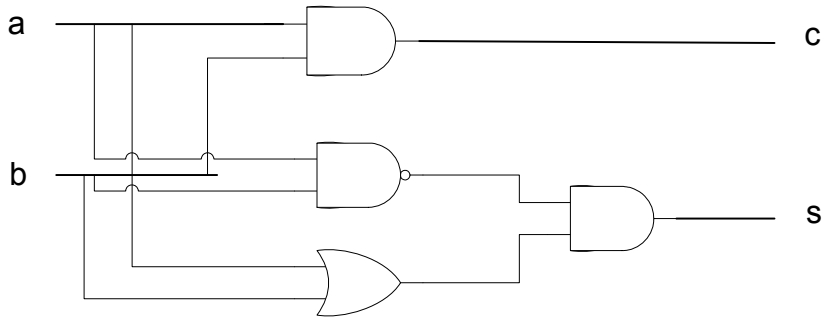
Iš ventilių galima sukonstruoti aukštesnio lygio komponentus. Panagrinėkime, kaip sukonstruoti sumatorių. Sumatorius yra labai svarbus kompiuterių architektūros elementas. Mūsų nagrinėjamoje architektūroje sumatorius naudojamas tiek interfeisiniame įrenginyje (fiziniam adresui formuoti), tiek operaciniame įrenginyje (aritmetinio loginio įrenginio sumatorius).

Pirma, pažiūrėkime kaip sukonstruoti sumatorių, kuris sudeda du dvejetainius bitus. Nors mes sudedame tik du bitus, t.y. pradinės reikšmės yra vieno bito dydžio, bet rezultatas gali gautis dviejuose bituose – kai abi sudedamos reikšmės yra 1, gauname pernešimą. Todėl mūsų sumatorius turės du įėjimus ir du išėjimus – rezultato bitui ir pernešimo požymiui. Užrašykime galimas įėjimo ir išėjimo signalų kombinacijas. Įėjimo signalus žymėsime A ir B, o išėjimus S rezultatui ir C – pernešimo požymiui:

A	**B**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

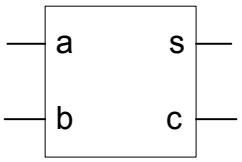
6.1.5 pav. Sumatoriaus teisingumo lentelė

Signalui S suformuoti mums reikia ventilių OR,AND ir NAND – S:=AND(OR(A,B), NAND(A,B), arba S:= (A OR B)AND(A NAND B). Išėjimui C reikia ventilio AND – C:=OR(A,B). Pavaizduokime tai schematiškai:



6.1.6 pav. Paprasčiausio sumatoriaus schema ventiliais

Kaip matome, turint ventilius, sukonstruoti sumatorių skaičiams, kurių dydis yra 1 bitas yra nesudėtingas uždavinys. Bet mūsų naudojami skaičiai dažniausia užima 8 bitus, ar net daugiau. Sukonstruoti dvejetainį sumatorių, kuris sudėtu du 8 bitų skaičius yra sudėtingesnis uždavinys, nes sudedant vyresniuosius bitus reikia atkreipti dėmesį į pernešimo požymį iš jaunesniųjų bitų sumos. Įrenginys, kuris 3 bitus – du bitus ir pernešimą – vadinamas pilnu sumatoriumi. Panagrinėkime, kaip galime panaudoti prieš tai sukonstruotą sumatorių bitams 8 bitų skaičių pilno sumatoriaus konstravime. Toliau sumatorių žymėsime tokiu simboliu:



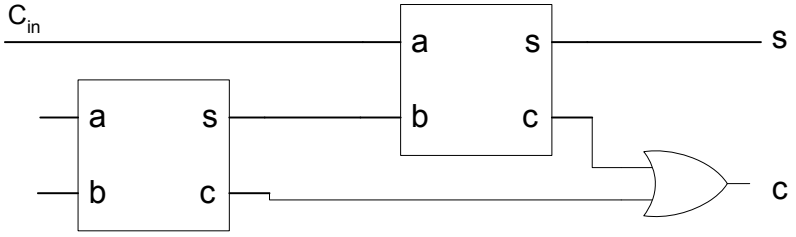
6.1.7 pav. Sumatoriaus vaizdavimas

kur *a* ir *b* yra sudedami bitai, o *s* ir *c* yra suma ir pernešimas.

Pastebėkime, kad pilnas sumatorius turi 3 įėjimus – du skaičių bitams ir vienas pernešimo bitui. Tarkime, turime sudėti du dvejetainius skaičius 11_2 ir 11_2:

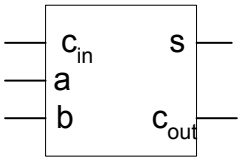
$$\begin{array}{r} 11 \\ 11 \\ \hline 110 \end{array}.$$

Kaip matome, gavome skaičių trejuose bituose. Sudedant du jaunesniuosius bitus, gavome 0 sumoje ir 1 pernešime. Sudedant du vyresniuosius bitus ir pernešimą, gavome 1 sumoje ir 1 pernešime. T.y. pilnas sumatorius turi susidėti iš dvejų sumatorių – vienas skirtas bitų A ir B sumai, kitas – pernešimui ir A ir B sumai sudėti. Pavaizduokime pilną sumatorių:



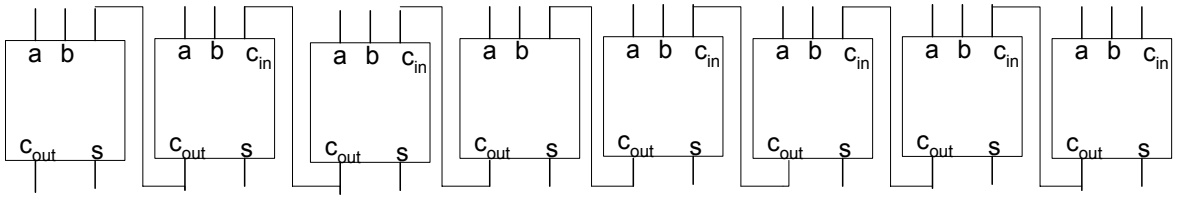
6.1.8 pav. Pilnas sumatorius

Jis susideda iš dvejų sumatorių ir vieno OR ventilio. Taip sukonstruotas pilnas sumatorius jau gali sudėti du bitus ir pernešimo bitą. Toliau vaizduosime pilną sumatorių tokiu simboliu:



6.1.9 pav. Pilno sumatoriaus vaizdavimas

Dabar nesunku sukonstruoti pilną sumatorių 8 bitų skaičiams sudėti – tam reikia sujungti aštuonis pilnus sumatorius taip, kad jaunesniojo sumatoriaus pernešimo išėjimas susijungtu su vyresniojo sumatoriaus pernešimo įėjimu.



6.1.10 pav. 8 bitų skaičių sumatorius

Ventiliai dirba nepastoviai, o tik tada kai į jį patenka taktinis impulsas. Kompiuteriai, kurie dirba pagal tokį principą, vadinami diskretiniais kompiuteriais. Analoginiuose kompiuteriuose ventiliai veikia pastoviai. Dauguma šiuolaikinių kompiuterių yra diskretiniai, todėl mes nagrinėjame tik juos.

Iš ventilių sudaromos integralinės schemos. Integralinė schema tai ventilių, sujungtų tarpusavyje laidais, rinkinys. Integralinė schema turi įėjimus ir išėjimus, kiekviena integralinė schema realizuoja tam tikrą funkciją. Aukščiau nagrinėti sumatoriai yra integralinės schemos.

Integralinės schemos veikimo greitis priklauso nuo taktinių impulsų gavimo dažnio – taktinio dažnio. Sumatoriaus integralinė schema (žr. pav. 6.1.X) susumuoja bitus per du taktus.

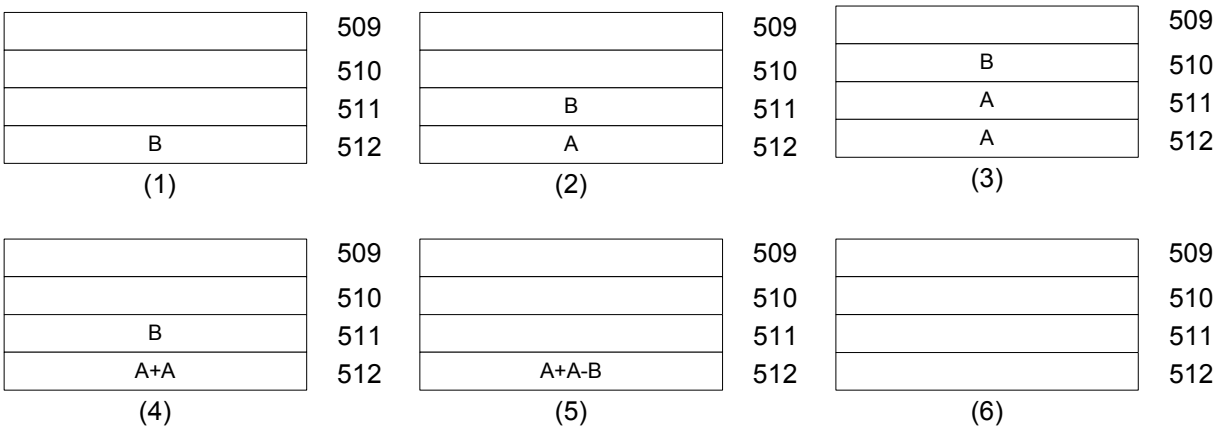
6.2. Interpretuojamas lygis

Čia mes nagrinėjame mikroprogramino valdymo įrenginį, kuris interpretuoja tam tikrą virtualią mašiną (žr. [Mit03, pp.11-13]). Stekinė architektūra mums jau žinoma, nagrinėjome ją 1.4 skyriuje. Naudosime tas pačias mnemonikas – PUSH, POP, ADD ir SUB.

Tarkime, mums reikia sudaryti programą, kuri apskaičiuoja formulę X:=2*A-B, kur X,A ir B yra atminties ląstelių adresai. Sudarykime programą pateiktai virtualiai mašinai:

```
PUSH    B       (1)
PUSH    A       (2)
PUSH    A       (3)
ADD             (4)
SUB             (5)
POP     X       (6)
```

Pavaizduokime steko kitimą programos vykdymo metu:



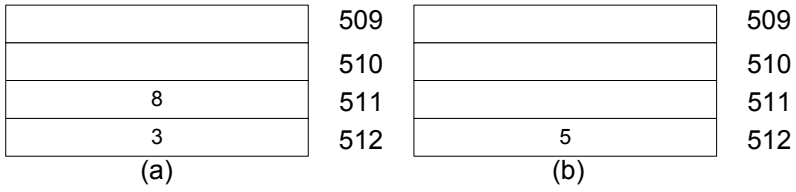
6.2.1 pav. Steko kitimas

6.3. Komandų realizacija

Interpretuojamo lygio programos yra pakankamai suprantamos ir jas yra nesudėtinga sukonstruoti, nes mums nereikia rūpintis žemesnio lygio komponentais – ventiliais, sumatoriais ir pan. Bet mikroprogramininiame lygyje tokios programos sudarymas reikalauja truputi daugiau pastangų.

Asemblerio lygio komandos vykdymas mikroprogramininiame lygyje iš esmės yra tam tikrų magistralių valdymas. Komandos ADD realizacija pateikta [Mit03, pp.17-18]. Pabandykime čia realizuoti SUB komandą – atimties operaciją.

Komandos veikimo logika suprantama – iš steko viršūnėje esančio skaičiaus reikia atimti sekantį steko skaičių ir užrašyti rezultatą į steko viršūnę. Pavyzdžiui, tarkime turime tokią situaciją:



6.3.1 pav. Steko būsena a) prieš SUB b) po SUB

Pažiūrėkime, kaip galime realizuoti atimties komandą magistralių valdymu. Pirma, pastebkime, kad atimtį galime realizuoti sudėties operacijos pagalba, antra

operandą užrašius papildomu kodu. Iš esmės, komandos SUB realizacija tik truputi skiriasi nuo komandos ADD realizacijos. Komandai reikia tik pakeisti trečio ciklo pirmą pociklį. Į sumatorių antras operandas (iš registro MAR) turi patekti papildomu kodu. Tam reikia įjungti 29 ventilį – papildomą kodą. Tokiu atveju, pilna komandos SUB realizacija pagal ciklus gali atrodyti taip:

Ciklas	Pociklis	SP	MAR	MBR	A	Ventiliai
1	1	511	?	?	?	20,2,14
	2	511	511	?	?	34
	3	512	511	?	?	38
2	1	512	511	8	?	20,23
	2	511	512	8	8	
	3	511	512	8	8	38
3	1	511	512	3	8	6,14,28
	2	511	512	3	8	
	3	511	512	3	8	37
4	1	511	512	-3	8	4,15
	2	511	512	-3	8	37
	3	511	512	5	8	39

6.3.2 pav. Komandos SUB realizacija

Komandų MUL ir DIV realizacija bus sudėtingesnė, jeigu naudosime ADD komanda cikle.

6.4. Komandos TEST ir GATE

Mikroprograminiame lygyje veiksmai atliekami mikrokomandomis. Pateiktoje mikroprogramėje architektūroje užtenka dviejų komandų TEST ir GATE (žr. [Mit03, pp.19-20]). Komandos užima 40 bitų. GATE komanda skirta ventilių valdymui. Jaunesnysis komandos bitas reiškia operacijos kodą – 1 skirtas komandai GATE, 0 komandai TEST. Komandos GATE likusieji 39[6] bitai skirti ventilių būsenai pažymėti – atidarytas (bito reikšmė 1) arba uždarytas (bito reikšmė 0).

Komanda TEST skirta komandų GATE vykdymo nuoseklumui pakeisti. Iš esmės tai yra sąlyginio valdymo perdavimo komanda. Komandoje nurodytas registras, kurį reikia palyginti (komandos bitai 1-8), bito, kurį reikia lyginti, pozicija (bitai 9-24),

[6] 39 – tiek mikroprograminiame lygyje yra ventilių.

reikšmė, su kuria reikia lyginti registro bitą (bitas 25), ir adresas, kuriuo reikia perduoti valdymą, jeigu reikšmės yra lygios (bitai 26-33). Likusieji bitai nenaudojami (plačiau žr. [Mit03, pp.19-21]).

6.5. Mikroprogramavimo kalba

Kaip matome, programuoti mikroarchitektūros lygmenyje komandomis GATE ir TEST yra pakankamai sudėtinga. Todėl verta įvesti mikroprogramavimo kalbą – MPL. MPL kalba susideda iš mikrokomandų, kurios atitinka komandas GATE ir TEST. Mikrokomandos, savo ruožtu, susideda iš operatorių, kurie nusako atidaromus ventilius. Viena mikrokomanda gali susidėti iš kelių operatorių, kurie užrašyti vienoje eilutėje. Jeigu tie patys operatoriai yra skirtingose eilutėse – jie apibrėžia kelias komandas.

Panagrinėkime kalboje naudojamus operatorius. Priskyrimo operatorius naudojamas duomenų persiuntimui tarp registrų. Pavyzdžiui, operatorius MAR = IR reiškia, 21 magistralė yra atidaryta, o operatoriai MAR = IR;PC = IR; reiškia, kad atidarytos magistralės 21 ir 18.

Sudėties operacija + pažymi sumatoriaus panaudojimą. Sudėties nariai yra registrai. Pavyzdžiui, operatoriai MBR = 1 + 1 reiškia konstantinių registrų reikšmių siuntimą į sumatorių ir rezultato persiuntimas į registrą MBR, mikrokomanda reiškia 7,14 ir 37 magistralių atidarymą, mikrokomandos pabaigoje MBR registro reikšmė bus 2.

Sudėtis su konstantiniu registru 0 reiškia reikšmės persiuntimą iš vieno registro į kitą. Atvirkštinio kodo operatorius yra COM. Pavyzdžiui, operatoriai MBR = COM(MBR) į registrą MBR įrašo invertuotą registro MBR reikšmę.

Sumatorius taip pat sugeba pastumti dviejų reikšmių sumą per vieną poziciją į dešinę arba į kairę operatoriais RIGHT_SHIFT ir LEFT_SHIFT. Šių operatorių veikimas analogiškas Intel 8088 komandų SHL ir SHR veikimui.

38 ir 39 magistralei pažymėti naudojamas operatorius MEMORY(MAR), kai operatorius stovi iš dešinės nuo priskyrimo operatoriaus, atidaroma 38 magistralė, o kai iš kairės – 39 magistralė. Plačiau apie mikroprogramavimo kalbą MPL skaitykite [Mit03, pp.22-24].

Iš esmės, programavimas mikrokomandomis yra tam tikrų reikšmių užrašymas iš vieno registro į kitą. Tam naudojamos primityvios operacijos, prieinamos

mikroarchitektūros lygmenyje – registro sumažinimas vienetu (sudėtis su konstantiniu registru (-1)), registro padidinimas vienetu (sudėtis konstantiniu registru 1), reikšmės persiuntimas iš vieno registro į kitą (sudėtis su konstantiniu registru 0), reikšmės daugyba arba dalyba iš 2 (atitinkamai operatorių LEFT_SHIFT arba RIGHT_SHIFT panaudojimas).

Panagrinėkime pavyzdžius, tarkime turime užrašyti reikšmę 16383 į registrą MBR. Pastebėkime, kad 16383_{10} yra 11111111111111_2, arba maksimali teigiama reikšmė, pastumta per vieną poziciją į dešinę. Apskaičiuoti maksimalią teigiamą reikšmę galime tokiu operatoriumi COM(SIGN). Postūmiui į dešinę naudojame operatorių RIGHT_SHIFT, visa komanda atrodys taip MBR=RIGHT_SHIFT(COM(SIGN)+0).

Paimkime kitą pavyzdį, tarkime reikia užrašyti reikšmę –40 į registrą MBR. Pabandykime išspręsti uždavinį nuo kito galo, t.y. jeigu turime registre MBR reikšmė 40, kaip ją paversti -40. Prisiminkime, kad skaičiai su ženklu užrašomi papildomu kodu – visi bitai invertuojami ir pridedamas vienetas. Tokią komandą mes galime atlikti tokiais operatoriais MBR=COM(MBR)+1. Dabar pabandykime gauti reikšmę 40. Užrašykime reikšmę 40 tokia suma 40=32+8. Skaičius 32 yra skaičius 16 pastumtas per vieną poziciją į kairę, o 8 – 16 per vieną poziciją į dešinę. Skaičių 16 lengva gauti sudėjus konstantinius registrus 15 ir 1. Komandos, kurios užrašo skaičių –40 į registrą MBR yra tokios:

```
X=15;MBR=LEFT_SHIFT(X+1);        // X:15, MBR:32
A = RIGHT_SHIFT(X+1);            // X:15, MBR:32, A:8
MBR = A + MBR;                   // X:15,MBR:40, A:8
MBR = COM(MBR)+1                 // X:15, MBR:-40, A:8
```

Pratimai

1. **Užrašykite nurodytą dešimtainę reikšmę į registrą MBR**

1.1	0	1.10	8	1.19	–2
1.2	1	1.11	7	1.20	32766
1.3	2	1.12	30	1.21	32752
1.4	–1	1.13	28	1.22	32753
1.5	4	1.14	–32768	1.23	–32754
1.6	15	1.15	32767	1.24	31
1.7	16	1.16	–32767	1.25	–31
1.8	14	1.17	16384	1.26	29
1.9	32	1.18	32753	1.27	–29

1.28 –15
1.29 –16
1.30 –14
1.31 44
1.32 –44
1.33 43
1.34 –43
1.35 45
1.36 –45
1.37 34
1.38 –34
1.39 22
1.40 –22
1.41 6
1.42 –6
1.43 90
1.44 –90
1.45 48
1.46 –48
1.47 96
1.48 –96
1.49 60
1.50 75
1.51 –60
1.52 –75
1.53 56

Kontroliniai darbai

Kontrolinis darbas 1

1. Užrašyti dešimtainį skaičių 363.12125 slankaus kablelio formatu 4 baituose šešioliktaine sistema.
2. Požymių registro reikšmė SF = 0000h. Baitų sudėties komanda prie dešimtainės reikšmės 111 pridedama dešimtainė reikšmė 37. Užrašyti naują požymių registro reikšmę.
3. Apskaičiuoti nurodytos komandos operando absoliutų adresą pagal adresavimo baitą 82h. Po adresavimo baito eina baitai 34h 12h 54h 32h. Prieš operacijos kodo stovi baitas 26h. Duotos registrų reikšmės: AX=1234h, BX=2345h, CX=3456h, SI=54Afh,DI=FFAFh, BP=100h, CS=22AFh,DS=13B0h, SS=12A0h, ES=554Ah.
4. Duotas programos fragmentas:

MOV	SP,AX	*CS:0AB4h*
MOV	SS,BX	*CS:0006h*

   Vykdant pirmąją komandą kyla pertraukimas su kodu 6. Nustatyti kokios reikšmės bus įrašytos į steką, ir kokia bus požymių registro reikšmė prieš vykdant pirmąją ISR (pertraukimo apdorojimo procedūra) komandą. Duotos registrų reikšmės: AX=1234h, BX=2345h, SF=FF00h, SP=100h, CS=12ACh, DS=22ACh, SS=32ACh.
5. Duotas programos fragmentas. Nustatyti komandos, kuri bus vykdoma po (3) komandos, fizinį adresą:

MOV	AL,25h	*CS:0005h*
SUB	AL,85h	*CS:0007h*
JBE	Label1	*CS:0009h*

   Paskutinės komandos mašininiai kodai yra 76AFh. Duotos registrų reikšmės DS=0235h, CS=5564h, SS=8842h.
6. Duotos registrų reikšmės SI=000Eh, DI=FA0Eh, CX=000Dh, SF=FC00h. Apskaičiuoti registrų CX ir DI sumą, kai bus įvykdyta komanda REPNZ STOSB.

7. Mikrokomandomis užrašykite skaičių 12 į registrą MBR

Kontrolinis darbas 2

1. Užrašyti dešimtainį skaičių -167.275 slankaus kablelio formatu 4 baituose šešioliktaine sistema.
2. Duotas programos fragmentas:

```
        MOV     AL,55h
        SUB     AL,85h
        JNLE    Lab1
        MOV     AL,1
        JMP     Out1
Lab1:
        MOV     AL,2
Out1:
```

   Nustatyti AL registro reikšmę.

3. Apskaičiuoti nurodytos komandos operando absoliutų adresą pagal adresavimo baitą 0Dh. Po adresavimo baito eina baitai 0Dh 0Ah. Duotos registrų reikšmės: AX=1234h,BX=2345h,CX=3456h, SI=54AFh, DI=FFAFh, BP=0100h, CS=22AFh,DS=13B0h,SS=12A0h, ES=554Ah.
4. Užrašyti programos fragmentą, po kurio 67h pertraukimo ISR (pertraukimo apdorojimo procedūra) prasidėtu fiziniu adresu 3ABFFh.
5. Iš dešimtainės skaičiaus 63 atimti dešimtainį skaičių 135 ir suformuoti požymių registro reikšmę. SF=0000h.
6. Užrašyti komandas REPNZ SCASB, nenaudojant pasikartojimo prefiksų ir komandų darbui su eilutėmis, kai DF=1.
7. Mikrokomandomis užrašykite skaičių –16 į registrą MBR.

Kontrolinis darbas 3

1. Užrašyti dešimtainį skaičių -812.9513125 slankaus kablelio formatu 4 baituose šešioliktaine sistema.
2. Duota registro AX reikšmė 0ABCDh, nustatyti registro reikšmę po komandos AAM.
3. Apskaičiuoti komandos operando absoliutų adresą pagal adresavimo baitą 72h. Po adresavimo baito eina baitai 03h 0FFh. Duotos registrų reikšmės: AX=1234h,BX=2345h,CX=3456h, SI=54AFh, DI=FFAFh, BP=0100h, CS=22AFh,DS=13B0h,SS=12A0h, ES=554Ah.
4. Duotas programos fragmentas,

```
        xor     ax,ax           (1)
        push    ax              (2)
        pop     es              (3)
        mov     di,4            (4)
        mov     cx,150h         (5)
a2:
        mov     es:[di],cx      (6)
        add     di,2            (7)
        loop    a2:             (8)
```

   Apskaičiuoti pertraukimų su kodais 4,5 ir 6 ISR fizinius adresus.
5. Duotas programos fragmentas:

```
        MOV     AL,55h
        SUB     AL,63h
        JLE     Lab1
        MOV     AL,1
        JMP     Out1
Lab1:
        MOV     AL,2
Out1:
```

   Nustatyti registro AL reikšmę.
6. Užrašyti komandas REPZ CMPSW, nenaudojant pasikartojimo prefiksų ir komandų darbui su eilutėmis, kai DF=1.
7. Mikrokomandomis užrašykite skaičių 32 į registrą MBR.

Kontrolinis darbas 4

1. Užrašyti dešimtainį skaičių -362.14825 slankaus kablelio formatu 4 baituose šešioliktaine sistema.
2. SF=0000h. Iš dešimtainio skaičiaus 1 atimti dešimtainį skaičių 63 ir suformuoti požymių registro reikšmę.
3. Apskaičiuoti komandos operando absoliutų adresą pagal adresavimo baitą B1h. Po adresavimo baito eina baitai 33h 22h. Duotos registrų reikšmės: AX=1234h,BX=2345h,CX=3456h, SI=54AFh, DI=FFAFh, BP=0100h, CS=22AFh,DS=13B0h,SS=12A0h, ES=554Ah.
4. Užrašyti programos fragmentą, po kurio 3Fh pertraukimo ISR (pertraukimo apdorojimo procedūra) prasidėtu fiziniu adresu 8EBFEh
5. Duotas programos fragmentas:

```
        MOV     AL,65h
        SUB     AL,88h
        JBE     Lab1
        MOV     AL,1
        JMP     Out1
Lab1:
        MOV     AL,2
Out1:
```

   Nustatyti registro AL reikšmę.

6. Apskaičiuoti SI ir DI registrų sumą po komandų REPE MOVSW. SI=0112h, DI=0012h, CX = 23h, SF=FF00h
7. Mikrokomandomis užrašykite skaičių 37 į atminties ląstelę adresu 36.

Kontrolinis darbas 5

1. Užrašyti dešimtainį skaičių -703.573 slankaus kablelio formatu 4 baituose šešioliktaine sistema.
2. SF=0000h. Prie dešimtainės reikšmės 250 pridėti dešimtainę reikšmę 51 ir suformuoti naują požymių registro reikšmę.
3. Apskaičiuoti komandos operando absoliutų adresą pagal adresavimo baitą 73h. Po adresavimo baito eina baitai 33h FFh. Duotos registrų reikšmės: AX=1234h,BX=2345h,CX=3456h, SI=54AFh, DI=FFAFh, BP=0100h, CS=22AFh,DS=13B0h,SS=12A0h, ES=554Ah.
4. Duotas programos fragmentas:

```
MOV   AX,F301h      (1) CS:0040
PUSH  AX            (2) CS:0043
POPF                (3) CS:0044
MOV   AX,0          (4) CS:0045
PUSH  AX            (5) CS:0048
POP   DS            (6) CS:0049
XOR   CX,CX         (7) CS:004A
```

   Apskaičiuoti komandos, kuri bus vykdoma po (4) komandos, fizinį adresą. Atminties baitai su adresais nuo 0 iki 15 užpildyti Fibonači skaičiais (1,1,2,3,5,8 ir t.t.). CS=045Eh, SS=55AFh

5. Duotas programos fragmentas:

```
        MOV     AL,EFh
        SUB     AL,7Fh
        JNAE    Lab1
        MOV     AL,1
        JMP     Out1
Lab1:
        MOV     AL,2
Out1:
```

   Nustatyti registro AL reikšmę.

6. Apskaičiuoti SI ir DX registrų sumą po komandų REPE LODSB. SI=0112h, DI=0012h, CX = 13h, DX=FFEEh, SF=0000h
7. Mikrokomandomis užrašykite skaičių –16383 į atminties ląstelę adresu 43.

Atsakymai ir sprendimai

1. Kompiuterių architektūros sąvoka

1.4 Architektūrų apžvalga

1 2^{24}-1.

2 80, 56, 32 ir 8.

3 Užrašykite programos fragmentą, kuris apskaičiuoja formulę
***x* := ($a^2 + b^2$)/($a^2 – a*b + b^2$)**

```
3.5   MOV   R1,a
      MOV   R2,b
      MOV   R3,R1
      MOV   R4,R2
      MUL   R3,R3
      MUL   R4,R4
      MOV   R5,R3
      ADD   R3,R4
      MUL   R1,R2
      SUB   R5,R1
      ADD   R5,R4
      DIV   R3,R5
      MOV   x,R3
```

7 Apskaičiuokite sumatoriaus reikšmę.

7.1 66

7.2 64

7.3 69

2. Duomenų formatai

2.1 Fiksuoto kablelio formatai

1 Užrašykite dešimtaines reikšmes dvejetainėje sistemoje

1.1 1011
1.2 1100
1.3 100101
1.4 101101
1.5 1001111
1.6 1010111
1.15 1111000
1.16 11000101
1.17 11100001
1.18 11111100
1.20 11111111
1.21 100000000
1.22 1000000000
1.23 10000000

1.24 01111111
1.25 10000001
1.26 111111111
1.27 1000000001

3 Užrašykite dvejetaines reikšmes šešioliktainėje sistemoje

3.1 77h
3.2 3F2Ah
3.3 3FFDh
3.4 1Dh
3.5 0Fh
3.6 10h
3.7 5h
3.8 0Ah
3.9 0Bh
3.10 3Fh
3.11 8Ach
3.12 1BD7h

6 Apskaičiuokite dešimtainių reikšmių viename baite sumą ir nustatykite požymius CF ir OF.

6.1 1111101, CF=1, OF=1
6.2 1111010, CF=1, OF=1
6.5 11111100, CF=0, OF=0
6.6 11000100, CF=0, OF=0
6.12 00001100, CF=1, OF=0
6.13 10000000, CF=0, OF=1

2.3 Dešimtainiai skaičiai

1 Užrašyti fiksuoto kablelio formato skaičius dešimtainiu supakuotu formatu

1.1 12h
1.2 65h
1.3 02h 53h
1.4 01h 27h
1.5 02h 55h
1.6 00h
1.7 01h 28h
1.8 06h 57h
1.9 08h 74h
1.10 05h 55h

2 Užrašyti fiksuoto kablelio formato skaičius dešimtainiu išpakuotu formatu

2.1 09h 08h
2.2 05h 04h
2.3 01h 01h
2.4 02h
2.5 01h 02h 08h
2.6 03h 03h 03h
2.7 08h 04h 07h
2.8 06h 04h 07h
2.9 01h 01h 01h 01h

2.4 Matematinio procesoriaus duomenų formatai

1 Normalizuoti dešimtaines reikšmes

1.1 $1{,}111*2^3$
1.2 $1{,}1111*2^4$
1.3 $1{,}0*2^5$
1.10 $1{,}111111111*2^9$
1.14 $1{,}11111110(0011)*2^6$

2 Užrašyti dešimtaines reikšmes formatu ”trumpas realus”

2.1 41980000h
2.2 40A00000h
2.5 42FE0000h
2.6 42FE8000h
2.7 42FF428Fh
2.8 41219999h
2.9 C2580000h
2.10 C282C000h
2.12 44241000h

2.17 C361A666h
2.18 C3804CCCh
2.21 C2863333h
2.27 C47FC000h
2.30 47800000h

3 **Užrašyti dešimtaines reikšmes formatu ”ilgas realus”.**

3.1 408FF80000000000h
3.2 4090000000000000h
3.3 C090040000000000h
3.5 C08001C28F5C28F5h
3.6 C0800CCCCCCCCCCCh
3.7 4058A33333333333h
3.12 BFA9999999999999h
3.13 BFD0000000000000h
3.14 BF847AE147AE147Ah
3.18 C0EFFFE000000000h
3.20 405FD666666666666h
3.26 4031400000000000h
3.28 4018199999999999h
3.30 404F999999999999h

3. i8088 architektūra

3.2 Atminties segmentacija

1 **Apskaičiuokite operando absoliutų adresą, kai žinomas segmento paragrafo numeris ir poslinkis**

1.1 457A0h
1.2 1F0E0h
1.8 0A059Fh
1.11 750C0h
1.13 0FBA99h
1.15 808FCh

3.3 Komandos struktūra

1 **Duotai komandai nurodykite adresavimo baitą ir baitus kurie eina po jo (operacijos kodo nurodyti nereikia)**

1.3 18h
1.4 77h 03h
1.5 56h 02h
1.6 0B8h 0ABh 33h
1.14 0A1h 0AFh 33h
1.15 46h 7Fh
1.16 0AAh 80h 00h
1.17 0ACh 8Ah 00h
1.18 65h 80h

2 **Duota 3 baitų seka, kur pirmas baitas – adresavimo baitas. Komandos mnemonika MOV REG⇔R/M. Užrašykite pilną komandą.**

2.1 MOV DH,[BP+DI+04]
2.2 MOV SI,[BP+DI+32FFh]
2.3 MOV AX,[BX+DI]
2.5 MOV BL,[DI+BP]
2.9 MOV [BX+DI+0FEh],SI
2.11 MOV [BP+DI+0F3FFh],CH
2.12 MOV [BX+SI+0FFFFh],DH
2.14 MOV [DI],CL

3.5 Stekas

4 **Į steką įrašyti ne mažiau nei 2 žodžiai. Sukeiskite du žodžius steko viršūnėje ir sekančioje ląstelėje vietomis. Nenaudoti kintamųjų ir išsaugoti pradines registrų reikšmes.**

```
PUSH  AX
PUSH  BX
PUSH  BP
MOV   BP,SP
```

```
MOV   AX,[BP+6]
MOV   BX,[BP+8]
MOV   [BP+8],AX
MOV   [BP+6],BX
POP    BP
POP    BX
POP        AX
```

3.6 Pertraukimai

1 Užrašyti programos fragmentą, kuris keičia pertraukimo su kodu N fizinį adresą į A_{fiz}.

1.2

```
PUSH    DS
XOR     AX,AX
MOV     DS,AX
MOV     DI,21h*4
MOV     [DI],000Dh
MOV     [DI+2],3ABCh
POP     DS
```

3 Apskaičiiuokite komandos, kuri bus vykdoma po komandos CMD, kai nurodyta VALUE reikšmė. CS=045Eh, SS=55AFh. Atminties baitai su adresais nuo 0 iki 15 užpildyti Fibonači skaičiais (1,1,2,3,5,8,…).

Pastaba: Atkreipkite dėmesį į požymio TF reikšmę.

6 Apskaičiuokite komandos, kuri bus vykdoma po (3) komandos, fizinėį adresą kai atminties baitai su adresais nuo 0 iki 12 užpildyti seka 23h,22h,21h,20h,1Fh ir t.t. CS=124Fh, SS=44FEh,SP=0002h.

Pastaba: prisiminkite, kur įrašomas grįžimo adresas ir požymio registro reikšmė pertraukimo metu.

4. Komandų sistema

4.1 Bendrosios komandos

4 Užrašyti komandas SAHF ir LAHF komandomis MOV,PUSHF ir POPF.

4.1 LAHF

```
PUSHF
POP     AX
AND     AX,00FFh
```

4.2 SAHF

```
AND     AX,00FFh
PUSH    AX
POPF
```

4.2 Aritmetinės komandos

1 Nurodykite naują požymių registro SF reikšmę.

1.1 SF=0FF90h.

1.2 SF=0F882h.

1.3 SF=0005h.

1.4 SF=0801h.

4.3 Dešimtainiai skaičiai

1 Apskaičiuokite registrų AL ir AH bei požymių AF ir CF reikšmes

1.1 AF=0,CF=0,AL=09h,AH=00h
1.2 AF=1,CF=1,AL=00h,AH=66h
1.4 AF=1,CF=1,AL=05h,AH=85h
1.5 AF=1,CF=1,AL=08h,AH=01h

4.6 Komandos darbui su eilutėmis

4 Apskaičiuokite registrų DI ir SI sumą po fragmento įvykdymo

4.1 SI+DI=78h

4.8 Sąlyginis valdymo perdavimas

1 Apskaičiuokite registro AL reikšmę

1.1 AL=00h
1.2 AL=7Fh
1.3 AL=0Ech
1.17 AL=13h

2 Apskaičiuokite registro AL reikšmę

2.1 AL=21h.
2.2 AL=2Fh.
2.10 AL=48h.
2.12 AL=00h.
2.14 AL=00h.
2.20 AL=79h.

4.9 Ciklų komandos

1 Apskaičiuokite registro AL reikšmę po ciklo.

1.1 AL=5Eh.
1.2 AL=58h.
1.3 AL=07h.
1.18 AL=0B4h.
1.19 AL=29h.
1.20 AL=84h.

6. Mikroprogramininis lygis

6.5 Mikroprogramavimo kalba

1 Užrašykite nurodytą dešimtainę reikšmę į registrą MBR

1.1 MBR=0+0
1.2 MBR=1+0
1.3 MBR=1+1
1.4 MBR=(-1)+0
1.10 X=15;MBR=RIGHT_SHIFT(X+1)
1.11 X=15;MBR=RIGHT_SHIFT(X+(-1))
1.12 X=15;MBR=LEFT_SHIFT(X+0)
1.13 X=15;MBR=LEFT_SHIFT(X+(-1))

Literatūra

[Mit03] A.Mitašiūnas, Kompiuterių architektūra, 2003

[Tan98] Andrew Tanenbaum, Structured Computer Organization, Prentice Hall, 669 pages, 1998

[Bau03] В.Г.Баула, Введение в архитектуру ЭВМ и системы программирования, МГУ ВМК, 2003

[Fom87] С.В.Фомин, Системы счисления, Наука, 1987

[Shn98] В.З. Шнитман, Архитектура современных компьютеров, Москва, 1998

[Kaz90] Под редакцией Ю.М. Казаринова, Микропроцессорный комплект К1810, Высшая школа, 1990

[RF97] П.И. Рудаков, К.Г. Финогенов, Программируем на языке ассемблера IBM PC, Обнинск, 1997