1. Skaičiavimo sistemos

Skiriamos pozicinės ir nepozicinės skaičiavimo sistemos. Mes kasdieną susiduriame su dešimtaine skaičiavimo sistema, kuri yra pozicinė skaičiavimo sistema. Pozicinėje skaičiavimo sistemoje simbolio, reiškiančio skaitmenį, prasmė priklauso nuo jo vietos skaičiuje. Nepozicinėje skaičiavimo sistemoje tokio simbolio prasmė nepriklauso nuo jo vietos skaičiuje. Plačiau apie skaičiavimo sistemas galima paskaityti [1, 113 p.].

1.1 Dešimtainė sistema

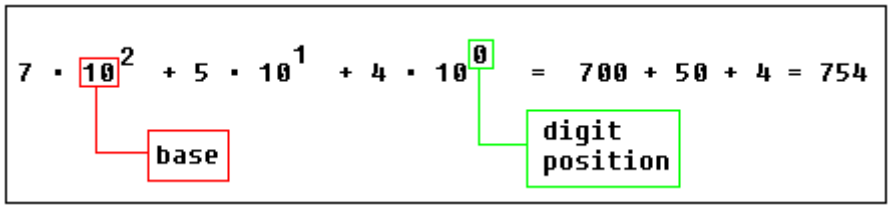
Skaičiuodami dažniausiai naudojame dešimtainę skaičiavimo sistemą. Joje yra dešimt skaitmenų:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Šiais skaitmenimis galime išreikšti bet kokią skaitinę reikšmę, pavyzdžiui:

754.

Reikšmė gaunama sumuojant kiekvieną skaitmenį, padauginus jį iš pagrindo (**base**), pakelto skaitmens pozicijos (**digit position**) laipsniu. Pozicijos skaičiuojamos nuo nulio, iš dešinės į kairę. Šiuo atveju pagrindas yra 10, todėl, kad yra 10 skaitmenų dešimtainėje sistemoje:



1.1 pavyzdys. Pademonstruosime kaip gaunama dešimtainio skaičiaus **1095** skaitinė reikšmė. Sumuojame

$$1 \times 10^3 + 0 \times 10^2 + 9 \times 10^1 + 5 \times 10^0 = 1000 + 0 + 90 + 5 = 1095$$

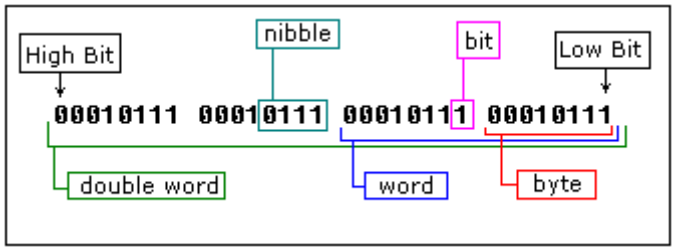
1.1 pratimas. Parodykite kaip gaunama dešimtainio skaičiaus **5268** skaitinė reikšmė.

1.2 Dvejetainė sistema

Elektroninės mašinos turi dvi būsenas: įjungta ir išjungta, arba 1 ir 0. Tokioms būsenoms atspindėti naudojama dvejetainė skaičiavimo sistema. Joje yra du skaitmenys:

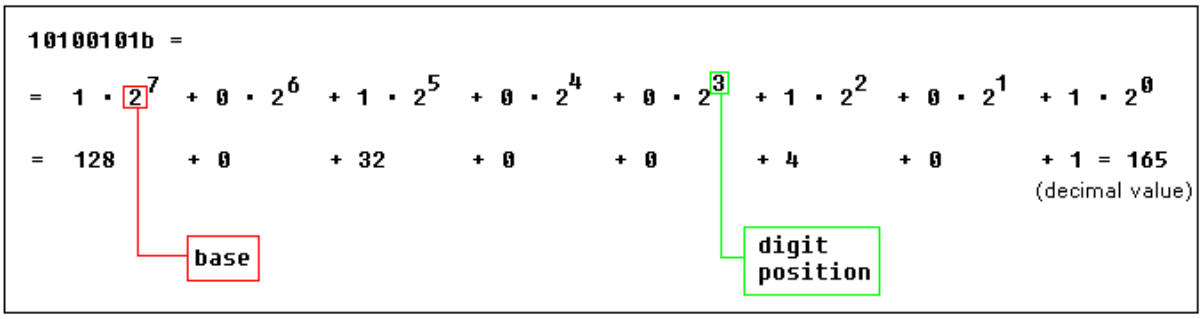
0, 1.

Taigi dvejetainės sistemos pagrindas yra 2. Kiekvienas skaitmuo dvejetainiame skaičiuje yra vadinamas bitu (**BIT**), 4 bitai sudaro pusbaitį (**NIBBLE**), 8 bitai - baitą (**BYTE**), 2 baitai - žodį (**WORD**), 2 žodžiai - dvigubą žodį (**DOUBLE WORD**):



Tam, kad atskirtume dvejetainius skaičius nuo kitų, laikomasi susitarimo dvejetainio skaičiaus gale prirašyti "**b**". Tokiu būdu žinome, kad **101b** yra dvejetainis skaičius su dešimtaine reikšme **5**.

Dvejetainis skaičius **10100101b** yra **165** dešimtainėje sistemoje:



1.2 pavyzdys. Dvejetainį skaičių **11011011b** užrašysime dešimtainėje sistemoje. Sumuojame

$$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 128 + 64 + 0 + 16 + 8 + 0 + 2 + 1 = 219$$

1.2 pratimas. Dvejetainį skaičių **01101110b** užrašykite dešimtainėje sistemoje.

1.3 Veiksmai su dvejetainiais skaičiais

Sudėtis. Atlikdami dviejų dvejetainių skaičių sudėtį laikomės šių taisyklių:

1. 0 + 0 duoda 0;
2. 0 + 1 duoda 1;
3. 1 + 0 duoda 1;
4. 1 + 1 duoda 0 ir 1 minty (kitaip tariant, gauname dvejetainį dvejetą „10“ nulį parašome, o vienetuką laikome mintyje).

Pavyzdžiui norėdami sudėti du dvejetainius skaičius **10110011b** ir **01101110b** užrašome

+	1	0	1	1	0	0	1	1
	0	1	1	0	1	1	1	0
1	0	0	1	0	0	0	0	1

Paryškintas vienetas duota baito perpildymą.

Asembleryje yra keletas komandų, galinčių realizuoti sudėties operaciją. Jos pateikiamos lentelėje.

Mnemonika	Formatas	Komentaras
ADD	ADD operandas1, operandas2	Operandais gali būti registrai, atminties adresas, tiesioginė reikšmė. Į pirmąjį operandą įrašoma pirmojo ir antrojo operandų suma.
ADC	ADC operandas1, operandas2	Operandais gali būti registrai, atminties adresas, tiesioginė reikšmė. Į pirmąjį operandą įrašoma pirmojo, antrojo operandų ir pernešimo žymės reikšmių suma.
AAA	Operandų neturi	ASCII kodo pakoregavimas po sudėties. Jei žemesnysis registro AL pusbaitis >9 arba AF (auxuliary flag) = 1 tada • AL = AL + 6 • AH = AH + 1 • AF = 1 • CF = 1 (carry flag) Priešingu atveju • AF = 0 • CF = 0 Abiem atvejais išvalomas aukštesnysis AL pusbaitis.
DAA	Operandų neturi	Dešimtainis pakoregavimas po sudėties. • Jei žemesnysis AL pusbaitis > 9 arba AF = 1 tada: • AL = AL + 6 • AF = 1 Jei AL > 9Fh arba CF = 1 tada • AL = AL + 60h • CF = 1
INC	INC operandas	Operandais gali būti registras arba atminties adresas. Nurodyto operando reikšmę didina vienetu.

1.3 pavyzdys. Sudėties komandų veikimą iliustruoja keletas pavyzdžių, kurie pateikiami kartu su mikroprocesoriaus emuliavimo programa Emu8086. Pavyzdžiai, kuriuos dabar reikėtų išnagrinėti yra programos

katalogo pakatalogyje „**Samples**“. Failų pavadinimai: **2_sample.asm**, **bcd_add.asm**, **bcd_aas.asm**, **bcd_aaa.asm**. Nagrinėjant šių programų kodą dabar daugiausiai kreipkite dėmesį į sudėties komandų veikimą.

Atimtis. Atlikdami dviejų dvejetainių skaičių atimtį laikomės šių taisyklių:

1. 0 – 0 duoda 0;
2. 1 – 0 duoda 1;
3. 0 – 1 duoda 1 ir prieš 0 stovintis vienetas tampa nuliu. Kaip pavyzdžiui iš 10b atimkime 01b

```
- 1 0   toliau paryškintas vienetas tampa   0 1
  0 1                                       0 1
    1                                       0 1
```

Taigi, atimkime du dvejetainius skaičius 1001010b ir 10100b. Užrašome:

```
- 1 0 0 1 0 1 0     1 0 0 0 1 1 0     1 0 0 0 1 1 0
      1 0 1 0 0           1 0 1 0 0         1 0 1 0 0
          1 0             0 1 1 0           1 0 1 1 0

  0 1 1 0 1 1 0
      1 0 1 0 0
    1 1 0 1 1 0
```

Asemblerio komandų, realizuojančių atimties operaciją, sąrašą pateikiame lentelėje:

Mnemonika	Formatas	Komentaras
SUB	SUB operandas1, operandas2	Operandais gali būti registras, atminties adresas ir tiesioginė reikšmė. Pirmajam operandui priskiria pirmojo ir antrojo operandų skirtumą.
SBB	SBB operandas1, operandas2	Operandais gali būti registras, atminties adresas ir tiesioginė reikšmė. Pirmajam operandui priskiria pirmojo, antrojo operandų ir pernešimo žymės (CF) skirtumą.
AAS	Neturi operandų	ASCII kodo pakoregavimas po atimties. Jei žemesnysis AL pusbaitis > 9 arba AF (auxiliary flag) = 1 tada: • AL = AL - 6 • AH = AH - 1 • AF = 1 • CF = 1 Priešingu atveju • AF = 0

		• CF = 0 Abiem atvejais išvalomas aukštesnysis AL pusbaitis.
DAS	Neturi operandų	Dešimtainis pakoregavimas po atimties. Jei žemesnysis AL pusbaitis > 9 arba AF (auxiliary flag) = 1 tada • AL = AL - 6 • AF = 1 Jei AL > 9Fh arba CF (carry flag) = 1 tada: • AL = AL - 60h • CF = 1
DEC	DEC operandas	Operandais gali būti registrai ir atminties adresas. Nurodytą operandą sumažina vienetu.
NEG	NEG operandas	Operandais gali būti registrai ir atminties adresas. • Invertuoja visus operando bitus • Prideda vienetą prie invertuoto operando
CMP	CMP operandas1, operandas2	Operandais gali būti registras, atminties adresas ir tiesioginė reikšmė. Palygina nurodytų operandų reikšmes. Iš pirmojo operando atima antrąjį, rezultatas niekur nesaugomas.

1.4 pavyzdys.

```
a) MOV AL, 5
SUB AL, 1        ; AL = 4

RET

b) STC ;nustatyti pernešimo žymę (carry flag)
MOV AL, 5
SBB AL, 3    ; AL = 5 - 3 - 1 = 1

RET

c) MOV AX, 02FFh  ; AH = 02, AL = 0FFh
AAS           ; AH = 01, AL = 09
RET
```

Užduotis. Čia pateiktus pavyzdėlius įvykdykite emu8086 programa. Instrukcijos **CMP** veikimą iliustruoja **4_sample.asm**.

Daugyba. Sudaugindami du dvejetainius skaičius laikomės šių taisyklių:

1. 0×0 duoda 0;
2. 0×1 duoda 0;
3. 1×0 duoda 0;
4. 1×1 duoda 1.

Dauginant dvejetainius skaičius veiksmų seka išlieka tokia pati kaip ir dešimtainių skaičių daugyboje. Imkime skaičius 1101b ir 101b ir sudauginkime juos:

```
        ×  1  1  0  1
              1  0  1
     +     1  1  0  1
        0  0  0  0
     1  1  0  1
  1  0  0  0  0  0  1
```

Asemblerio komandų, realizuojančių daugybos operaciją, sąrašą pateikiame lentelėje:

Mnemonika	Formatas	Komentaras
MUL	MUL operandas	Sudaugina neatsižvelgiant į ženklą. Kai operandas baito dydžio: AX = AL * operandas. Kai oprandas žodžio dydžio: (DX AX) = AX * operandas.
IMUL	IMUL operandas	Sudaugina atsižvelgiant į ženklą. Kai operandas baito dydžio: AX = AL * operandas. Kai oprandas žodžio dydžio: (DX AX) = AX * operandas.
AAM	Neturi operandų	ASCII kodo pakoregavimas po daugybos. • AH = AL / 10 • AL = liekana

1.5 pavyzdys.

```
MOV AL, -2
MOV BL, -4
IMUL BL      ; AX = 8
RET
```

Užduotis. Įvykdykite pateiktą kodo fragmentą emu8086 programa. Tada instrukciją IMUL pakeiskite instrukcija MUL. Pažiūrėkite kaip pasikeitė rezultatai.

Dalyba. Dvejetainės dalybos algoritmas sudėtingiausias iš čia aprašytų. Tiesa sakant yra du dalybos algoritmai. Galioja tokie pavadinimai dalinys, daliklis dalmuo. Pavyzdžiui 15/3 = 5. 15 yra dalinys, 3 yra daliklis, 5 yra dalmuo.

1 algoritmas. Dalyba kaip ciklinė atimtis

- Dalmeniui priskiriame nulį
- **Kartojame tol, kol** dalinys didesnis arba lygus už daliklį
  - Atimti daliklį iš dalinio
  - Pridėti vienetą prie dalmens
- **Kartojimo** bloko pabaiga
- Gautas dalmuo yra teisingas, o dalinys yra liekana
- STOP

Padalinkime **1001b** iš **11b**. **1001b** yra dalinys, o **11b** yra daliklis. Užrašome

Dalinys	Dalmuo
1 0 0 1 - 1 1	0
1 1 0 - 1 1	1
1 1 - 1 1	1 0
0	**1 1**

Gauname atsakymą **11b**, o liekana lygi **0b**.

2 algoritmas. Dalyba kaip perstūmimas ir atimtis

- Dalmeniui priskiriame nulį
- Išlygiuojame dalinį ir daliklį pagal kairį kraštą (patys kairiausi skaitmenys eina vienas po kitu)
- **Kartojame**
  - **Jei** virš daliklio esanti dalinio dalis yra didesnė arba lygi už daliklį
    - **Tada** atimti daliklį iš dalinio dalies, esančios virš daliklio **ir**
    - Prirašyti 1 prie dalmens iš dešinės
    - **Priešingu** atveju prirašyti 0 prie dalmens iš dešinės
  - Pastumti daliklį per vieną poziciją į dešinę
- **Tol, kol** dalinys tampa mažesnis už daliklį
- Gautas dalmuo yra teisingas, o dalinys yra liekana
- STOP

Dabar dvejetainius skaičių **1001b** padalinsime iš **11b** taikydami ką tik aprašytą algoritmą. Užrašome

Dalinys	Dalmuo
1 0 0 1 - 1 1	0
1 0 0 1 - 1 1	0 0
0 0 1 1 -	0 0 1

			1	1			
0	**0**	**0**	**0**	**0**	**0**	**1**	**1**

Gauname atsakymą **0011b**, o liekana lygi **0000b**. Kaip matome abiejų algoritmų atsakymai sutampa.

Asemblerio komandų, realizuojančių dalybos operaciją, sąrašą pateikiame lentelėje:

Mnemonika	Formatas	Komentaras
DIV	DIV operandas	Dalina neatsižvelgiant į ženklą. Kai operandas yra baito dydžio: AL = AX / operandas AH = liekana Kai operandas žodžio dydžio: AX = (DX AX) / operandas DX = liekana
IDIV	IDIV operandas	Dalina atsižvelgiant į ženklą. Kai operandas yra baito dydžio: AL = AX / operandas AH = liekana Kai operandas žodžio dydžio: AX = (DX AX) / operandas DX = liekana
AAD	Neturi operandų	ASCII kodo pakoregavimas po dalybos. • AL = (AH * 10) + AL • AH = 0

1.6 pavyzdys. Programoje emu8086 įvykdykite pavyzdį fahrenheit.asm. Stebėkite registrų turinius programos vykdymo metu. Ar vietoj instrukcijos IDIV galėtume čia panaudoti instrukciją DIV?

1.3 pratimas. Padauginkite ir dviem skirtingai būdais padalinkite dvejetainius skaičius **01101b** ir **01011b**.

1.4 Šešioliktainė sistema

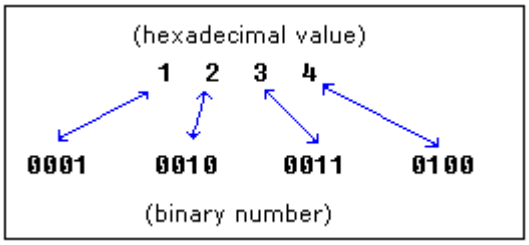
Šešioliktainė skaičiavimo sistema naudoja 16 skaitmenų:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Taigi tokios sistemos pagrindas yra 16. Šešioliktainiai skaičiai - kompaktiški ir lengvai skaitomi.

Lengva dvejetainius skaičius pervesti į šešioliktainę formą ir atvirkščiai. Kiekvienas pusbaitis (4 bitai) yra pervedamas į šešioliktainį skaitmenį pasinaudojus šia lentele:

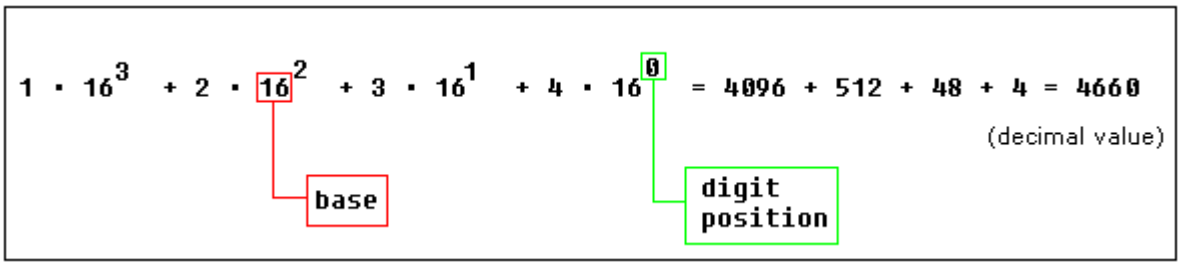
Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F



Šešioliktainio skaičiaus gale prirašome "**h**" tokiu būdu galime pasakyti, kad **5Fh** yra šešioliktainis skaičius, lygus **95** dešimtainėje sistemoje.

Taip pat, šešioliktainio skaičiaus priekyje prirašome "**0**" (nulį) , jei jis prasideda raide (**A..F**), pavyzdžiui **0E120h**.

Šešioliktainis skaičius **1234h** lygus **4660** dešimtainėje sistemoje:



1.7 pavyzdys. Šešioliktainį skaičių **5Fh** užrašysime dešimtainėje ir dvejetainėje sistemose. Pervesdami duotąjį skaičių į dešimtainę sistemą užrašome:

$$5Fh = 5 \times 16^1 + 15 \times 16^0 = 80 + 15 = 95$$

Čia dauginome iš šešiolikos, kadangi šešiolika yra šešioliktainės sistemos pagrindas. Skaičius 15 atsiranda todėl, kad jį žymi šešioliktainis skaičius F.

Pervesdami į dvejetainį skaičių, turime atsiminti, kad kiekviena šešioliktainio skaičiaus skiltis koduojama keturiais dvejetainiais skaitmenimis, todėl gauname

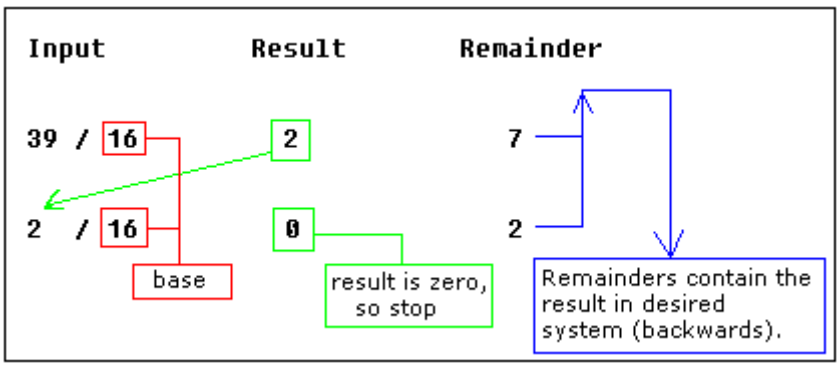
5Fh = 0101 1111b

1.5 pratimas. Šešioliktainį skaičių **0F9h** perveskite į dešimtainę ir šešioliktainę sistemą.

1.5 Dešimtainių skaičių vertimas į kitas sistemas

Pervesdami dešimtainį skaičių į bet kurią kitą skaičiavimo sistemą, dešimtainį skaičių daliname iš pagrindo (**base**) tos skaičiavimo sistemos, į kurią norime pervesti duotąjį dešimtainį skaičių. Padalinus, gautą rezultatą (**result**) ir dalybos liekaną (**remainder**) įsimename ir toliau daliname rezultatą iš naujos sistemos pagrindo tol, kol rezultatas tampa mažesnis už pagrindą. Dalybos liekanos, užrašytos atvirkščia tvarka, yra skaičius naujoje sistemoje.

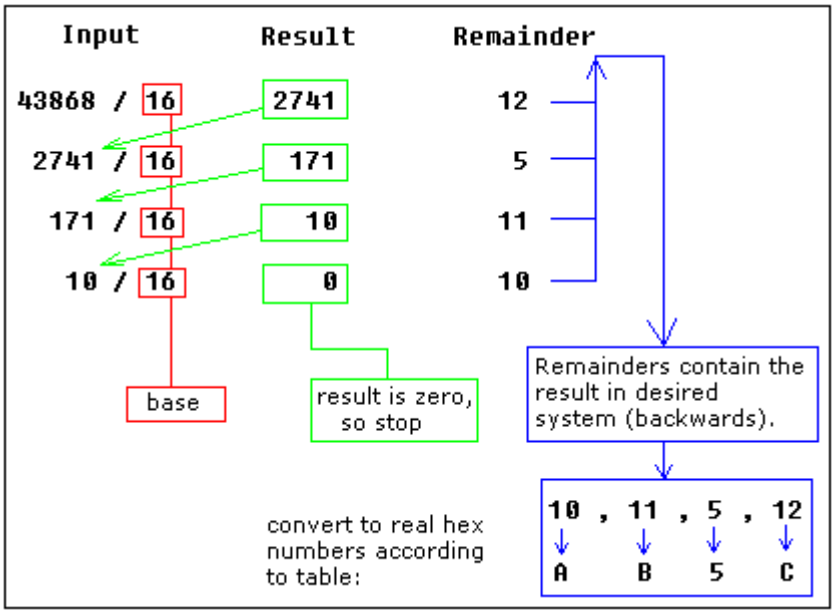
Perveskime dešimtainį skaičių 39 (pagrindas 10) į šešioliktainę sistemą (pagrindas 16):



Gauname šešioliktainį skaičių: **27h**.

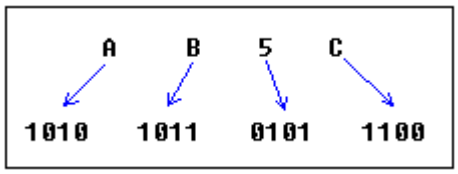
Kadangi, pavyzdyje visos dalybos liekanos buvo mažesnės nei 10, todėl šešioliktainiam skaičiui užrašyti nenaudojame jokių raidžių.

Kitas pavyzdys: perveskime dešimtainį skaičių **43868** į šešioliktainę sistemą:



Rezultatas - **0AB5Ch**. Liekanoms, didesnėms už 9, pervesti į atitinkamas raides naudojamės lentele.

Taikydami tuos pačius principus skaičių galime pervesti į dvejetainę formą (daliklis 2) arba galime skaičių pervesti į šešioliktainę formą, o tada naudojantis lentele pervesti jį į dvejetainę formą:



Gauname dvejetainį skaičių: **1010101101011100b.**

1.8 pavyzdys. Pervesime dešimtainį skaičių **58** į dvejetainę ir šešioliktainę sistemą. Gauname

įvedimas	rezultatas	liekana
58/2	29	**0**
29/2	14	**1**
14/2	7	**0**
7/2	3	**1**
3/2	1	**1**
1/2	0	**1**

Gauname dvejetainį skaičių **111010b**.

įvedimas	rezultatas	liekana
58/16	3	**10 arba A**
3/16	0	**3**

Gauname šešioliktainį skaičių **3Ah**.

1.6 pratimas. Perveskite dešimtainį skaičių **128** į dvejetainę ir šešioliktainę sistemą.

1.6 Teigiami ir neigiami skaičiai

Negalime užtikrintai pasakyti, kada šešioliktainis baitas **0FFh** yra teigiamas, o kada neigiamas. Jis gali atitikti dešimtainius skaičius "**255**" ir "**- 1**".

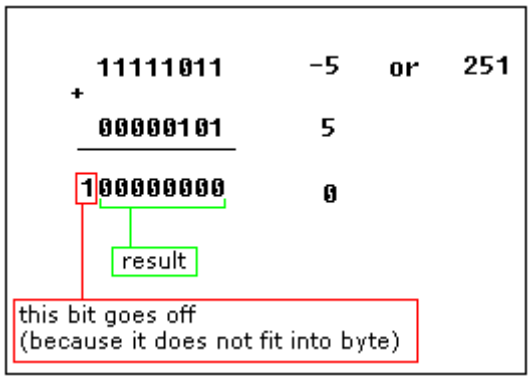
Naudodami 8 bitus galime sukurti **256** (įskaitant nulį) skirtingas kombinacijas, taigi paprasčiausiai laikome, kad pirmosios **128** kombinacijos (**0..127**) atspindi teigiamus skaičius, o likusios **128** kombinacijos (**128..255**) atspindi neigiamus skaičius.

Kad gautume "**- 5**", turime atimti **5** iš kombinacijų skaičiaus (**256**), taip gauname: **256 - 5 = 251**.

Toks sudėtingas neigiamų skaičių užrašymo būdas turi prasmę. Aritmetikoje, kai sudedame "**- 5**" ir "**5**" gauname nulį. Taip atsitinka, kai procesorius sudeda du baitus **5** ir **251**, rezultatas yra didesnis už **255**, dėl perpildymo procesorius grąžina nulį!

Jei dirbame su baito tipo kintamaisiais, tai skaičių sritis **128..256** yra laikoma neigiamų skaičių sritimi. Tai patogu dėl to, kad šių skaičių aukščiausias bitas visada **1**, taigi tai gali būti panaudota skaičiaus ženklui nustatyti.

Toks pat principas taikomas ir žodžiams (16 bitų reikšmėms), 16 bitų sukuria **65536** skirtingas kombinacijas, pirmosios 32768 kombinacijos (**0..32767**) naudojamos teigiamiems skaičiams žymėti, o likusios 32768 kombinacijos (**32767..65535**) neigiamiems skaičiams žymėti.



1.9 pavyzdys. Nustatysime koks šešioliktainis baito dydžio skaičius atitiktų dešimtainį neigiamą skaičių **-15**.

Atimame **256 – 15 = 241**. Dešimtainį skaičių **241** verčiame į šešioliktainį skaičių, gauname

įvedimas	rezultatas	liekana
241/16	15	**1**
15/16	0	**15 arba F**

Rezultatas **0F1h**.

1.7 pratimas. Nustatykite koks šešioliktainis baito dydžio skaičius atitiktų dešimtainį neigiamą skaičių **-36**.

1. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Kokia dešimtainio skaičiaus 900 išraiška šešioliktainėje sistemoje?	384h	Atsakymas teisingas.
		1604o	Atsakymas neteisingas. Tai šio skaičiaus atitikmuo aštuntainėje sistemoje.
		5Ah	Atsakymas neteisingas. Tai dešimtainio skaičiaus 90 atitikmuo šešioliktainėje sistemoje.
2.	Dvejetainio skaičiaus 1101011b išraiška dešimtainėje sistemoje yra	200	Atsakymas neteisingas. Paskaičiavę pagal nurodytą algoritmą turime gauti 107.
		107	Atsakymas teisingas.
		115	Atsakymas neteisingas. Paskaičiavę pagal nurodytą algoritmą turime gauti 107.
3.	Dvejetainis skaičius 1101011b aštuntainėje sistemoje atitinka skaičių	153h	Atsakymas neteisingas. Aštuntainį skaičių ženkliname simboliu „o“.
		153o	Atsakymas teisingas.
		107o	Atsakymas neteisingas. Turime gauti 153o, nes dvejetainį skaičių versdami į aštuntainį jį daliname į skiltis po tris bitus.

Papildomi skaitiniai

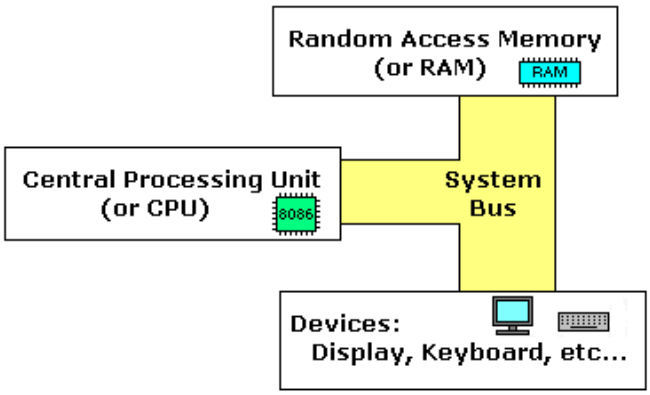
[1] A. Kiseliovas, Matematika. Vilnius: Mokslo ir enciklopedijų l-kla, 1994. – 296 p. (113 p. – 120 p.)
[2] A. Mitašiūnas, Kompiuterių architektūra, Mokymo priemonė. VU Informatikos katedra, 2003. – 126 p. (4 p. – 6 p.)
[3] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (59 p. – 65 p.)

2. Asemblerio kalbos ypatumai

Asembleris - tai žemo lygio programavimo kalba. Asembleriu vadinamas kompiliatorius, specialiu būdu sudarytas tekstines eilutes paverčiantis į mašinines instrukcijas. Kaip ir bet kuris kitas kompiliatorius, asembleris programos rašymo procesą palengvina sudarydamas programuotojui galimybę į mašinines instrukcijas ir operandus kreiptis naudojant jų simbolinius vardus.

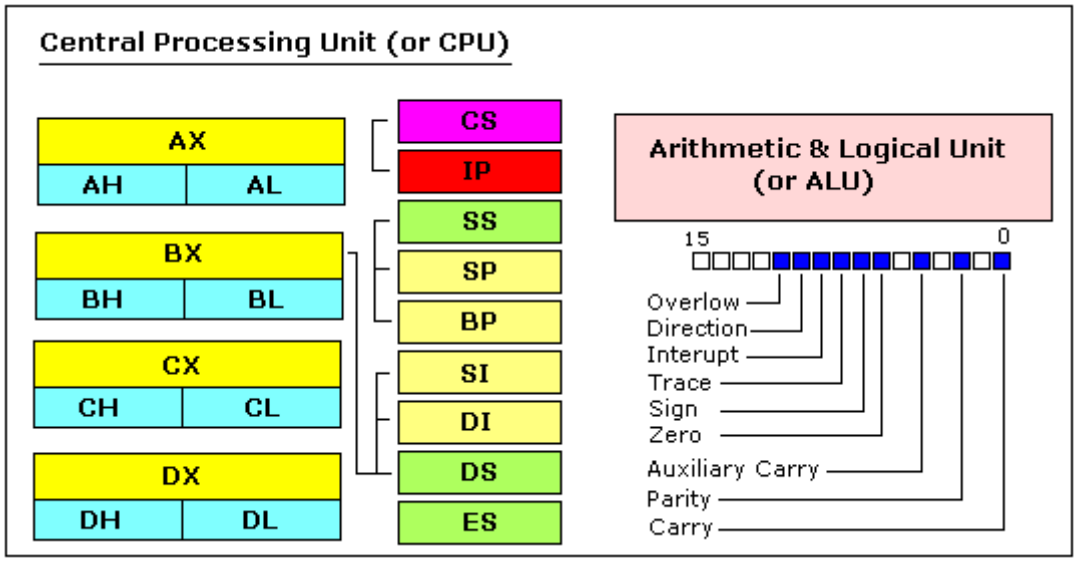
Programavimas asembleriu turi savų privalumų ir trūkumų. Privalumas būtų tas, kad rašydami programą asembleriu paprastai renkamės tą mašininių instrukcijų seką, kuri leidžia realizuoti reikiamus skaičiavimus ar kt. maksimaliu greičiu mažiausiomis atminties sąnaudomis, kai tuo tarpu aukšto lygio programavimo kalbos kompiliatorius neišvengiamai į mašininį kodą įnešą tam tikrą pertekliškumą, sumažinantį skaičiavimų greitį ir padidinantį atminties sąnaudas. Kita vertus, programavimas mašininių instrukcijų lygyje yra pernelyg skrupulingas užsiėmimas ir programų rašymo procesas nėra toks greitas kaip aukšto lygio programavimo kalbomis. Tai pagrindinis asemblerio trūkumas.

Tam, kad suprastume asemblerio kalbą, turime turėti tam tikrų kompiuterio sandaros žinių. Trumpai prisiminsime pagrindinius jos elementus. Paprasčiausias kompiuterio modelis pavaizduotas žemiau pateikiamame paveikslėlyje.



2.1 pav. Paprasčiausias kompiuterio sandaros modelis.

Sisteminė šyna arba **sisteminė magistralė** (system bus) jungia įvairias kompiuterio dalis. **CPU** (Central Processing Unit) - **procesorius** yra kompiuterio "širdis". Procesorius atlieka didžiausią dalį skaičiavimų. Į **RAM** (Random Access Memory) - **laisvosios kreipties atmintinę** įkraunamos programos, kurios turi būti įvykdytos.



2.2 pav. Centrinio procesorinio įrenginio sandaros schema.

8086/8088 šeimos mikroprocesorius turi 14 registrų (2.2 pav.). Pagal funkcionalumą jie skirstomi į tokias grupes:

- **bendrosios paskirties registrai AX, BX, CX, DX** skirti operandų saugojimui ir pagrindinių operacijų vykdymui; bet kuris iš jų gali būti naudojamas kaip dviejų – vyresniojo (AH, BH, CH, DH) ir jaunesniojo (AL, BL, CL, DL) baito, vienas nuo kito nepriklausančių 8 bitų registrų visuma;
- **segmentų registrai CS, DS, SS, ES** naudojami nurodyti segmentui atminties adresavimo metu;
- **registrai rodyklės SP, BP, IP** naudojami nurodyti postūmiui atminties adresavimo metu;
- **indeksiniai registrai SI, DI** naudojami indeksinėje adresacijoje;
- **žymių registras (flag register)** naudojamas žymių, nusakančių procesoriaus būseną, saugojimui.

Kiekvienos funkcinės grupės viduje registrai gali būti panaudoti įvairiais būdais.

2.1 Centrinis procesorinis įrenginys

Bendrosios paskirties registrai. Prie bendrosios paskirties registrų grupės dar priskiriami indeksiniai registrai SI, DI bei du registrai rodyklės SP, BP. Todėl sakoma, kad 8086 procesorius turi 8 bendrosios paskirties registrus, kiekvienas registras turi savo pavadinimą:

- **AX** - akumuliatorius, padalintas į **AH / AL** (accumulator regsiter). Šis registras yra pagrindinis sumatorius. Jis naudojamas visose aritmetinėse operacijose. Tik regsitro AX ir jo pusregistrių AH/AL dėka įmanomas duomenų apsikeitimas su įvedimo / išvedimo uostais.

- **BX** - bazės adreso registras, padalintas į **BH / BL** (base address register). Šis registras naudojamas kaip sumatorius aritmetinėse operacijose ir kaip bazinis registras indeksinėje adresacijoje.

- **CX** – registras skaitliukas, padalintas į **CH / CL** (count register). Šis registras pagrinde naudojamas kartojimo ir postūmio operacijose, taip pat gali dalyvauti ir aritmetinėse operacijose.

- **DX** - duomenų registras, padalintas į **DH / DL** (data register). Šis registras naudojamas duomenų įvedimo/išvedimo operacijose, taip pat kaip sumatorius kai dirbama su 32 bitų sveikaisiais skaičiais.

- **SI** - šaltinio indekso registras (source index register). Šis registras apibrėžia informacijos šaltinio adresą, kai naudojama indeksinė adresacija (pavyzdžiui, dirbant su masyvais). Paprastai naudojamas poroje su registru DS.

- **DI** - paskirties indekso registras (destination index register). Šis registras poroje su registru ES apibrėžia informacijos gavėją tarpsegmentinio duomenų apsikeitimo metu.

- **BP** - bazės rodyklė (base pointer). Šis registras palengvina lokalaus steko (steko, kuris naudojamas procedūros viduje) sukūrimą ir naudojimą.

- **SP** - steko rodyklė (stack pointer). Šis registras nurodo steko viršūnę. Tai yra, kartu su registru SS adresuoja atminties ląstelę, į kurią bus talpinamas operandas arba iš kurios jis bus išimamas. Šio registro turinys automatiškai mažėja, kai į steką patalpinamas eilinis operandas, ir automatiškai didėja, kai iš steko išimamas eilinis operandas.

Nepaisant registro pavadinimo, programuotojas nustato kam bus panaudoti bendrosios paskirties registrai. Pagrindinė registro paskirtis - saugoti skaičių (kintamąjį).

Visi čia išvardinti registrai yra 16 bitų dydžio, tai reiškia, kad juose gali būti saugomas pavyzdžiui toks skaičius:

0011000000111001b (dvejetainėje formoje), arba **12345** dešimtainėje formoje.

4 bendrosios paskirties registrai (AX, BX, CX, DX) sudaryti iš dviejų atskirų 8 bitų registrų, pavyzdžiui jei AX= **0011000000111001b**, tai AH=**00110000b** ir AL=**00111001b**. Taigi, modifikuojant kurį nors 8 bitų registrą, 16 bitų registro turinys taip pat atnaujinamas ir atvirkščiai.

Lygiai taip yra ir su kitais trimis registrais, "H" žymi vyresnįjį baitą "L" jaunesnįjį baitą.

Kadangi registrai yra procesoriaus viduje, jie daug greitesni nei atmintinė. Kreiptis į atmintinę vyksta per sistemos magistralę, taigi ji užtrunka daug ilgiau nei kreiptis į registrus. Duomenų nuskaitymas arba įrašymas į registrą paprastai neužima jokio laiko. Taigi, reikia stengtis saugoti kintamuosius registruose. Registrų rinkinys

negausus ir dauguma jų turi specialią paskirtį, kuri apriboja jų kaip kintamųjų naudojimą, tačiau jie visgi yra puiki vieta laikinų skaičiavimo duomenų saugojimui.

2.1 pavyzdys. Programoje emu8086 įvykdykite pavyzdį **5_sample.asm**. Stebėkite kaip keičiasi bendrosios paskirties registrų turinys programos vykdymo metu.

2.1 pratimas. Mokėkite paaiškinti kas atliekama kiekvienoje programos **5_sample.asm** kodo eilutėje.

Segmentų registrai. Kaip jau žinome yra 4 segmentų registrai:

- **CS** – kodo segmentas (code segment). Šis registras saugo atminties segmento numerį, kuriame įkrauta einamoji mašininė instrukcija. Sekančios komandos pilnojo adreso gavimui šio registro turinys pastumiamas į kairę per keturis bitus ir sudedamas su registru rodykle IP. Registro CS turinys automatiškai keičiasi tolimosios (tarpsegmentinės) kreipties ir procedūrų iškvietimo komandose.

- **DS** – duomenų segmentas (data segment). Šis registras saugo atminties segmento numerį, į kurį įkrauti duomenys (konstantos ir kintamieji).

- **ES** - papildomas segmento registras (extra segment). Šis registras naudojamas duomenų apsikeitimui tarp segmentų ir kai kuriose operacijose su eilutėmis.

- **SS** - steko segmentas (stack segment). Šis registras saugo steko segmento numerį. **Stekas** – tai automatiškai adresuojamos atminties sritis, skirta laikinam operandų saugojimui. Steko atmintis naudojama pagal **LIFO** (**L**ast **I**n **F**irst **O**ut) taisyklę. Tai reiškia, kad paskutinis į steką įkrautas operandas iš jo bus išimamas pirmas. Steko dėka yra organizuojamas duomenų apsikeitimas tarp programos ir procedūrų, saugomi lokalūs kintamieji.

Net jei yra įmanoma išsaugoti bet kokius duomenis segmentų registruose, to nevertėtų daryti. Segmentų registrai turi specifinę paskirtį - nurodyti prieinamus atmintinės blokus.

Segmento registrai dirba kartu su bendrosios paskirties registrais, kad būtų įmanoma pasiekti bet kurią atmintinėje esančią reikšmę. Pavyzdžiui, jei norime pasiekti atmintinę, kurios fizinis adresas **12345h**, turime įkrauti **DS = 1230h** ir **SI = 0045h**. Tokiu būdu galime pasiekti daugiau atmintinės adresų, nei su vienu registru, kuris apribotas 16 bitų reikšmėmis.

Procesorius nustato atmintinės **fizinį adresą** daugindamas segmento registro reikšmę iš 10h ir pridėdamas bendrosios paskirties registro reikšmę (1230h * 10h + 45h = 12345h):

```
 12300
+  0045
-------
 12345
```

Adresas, suformuotas dvejais registrais, vadinamas **efektyviu adresu**. Įprasta, kad registrai **BX, SI** ir **DI** dirba su **DS** segmento registru; **BP** ir **SP** su **SS** segmento registru.

Kiti bendrosios paskirties registrai negali sudaryti efektyvaus adreso! Taip pat, pastebėtina, kad BX gali sudaryti efektyvų adresą, tačiau BH ir BL negali!

2.2 pavyzdys. Programoje emu8086 įvykdykite pavyzdį **stack.asm**. Pažingsniui vykdydami programą stebėkite kaip keičiasi segmentų registrų ir turinys programos vykdymo metu (automatinį steko atminties adresavimą stebėkite įjungę mygtuką „Stack“).

2.2 pratimas. Mokėkite paaiškinti kas atliekama kiekvienoje programos **stack.asm** kodo eilutėje.

Specialiosios paskirties registrai. Specialiosios paskirties registrų grupei priskiriamas registras rodyklė IP ir žymių registras:

- **IP** - instrukcijų rodyklė (instruction pointer). Šis registras apibrėžia kodo segmento CS pradžios postūmį atsižvelgiant į eilinę vykdomą mašininę instrukciją. IP turinys automatiškai kinta instrukcijos vykdymo metu, tuo užtikrinant teisingą komandų išrinkimo iš atminties tvarką.

- **Flags Register** - žymių registras, parodo einamąją procesoriaus būseną.

IP registras visada dirba su **CS** segmento registru ir nurodo einamuoju momentu vykdomą instrukciją.

Flags Register - **žymių registras** yra atnaujinamas automatiškai (tai atlieka procesorius), įvykdžius matematines operacijas. Tai leidžia nustatyti rezultatų tipą ir valdymo perdavimo kitoms programos dalims sąlygas. Registras yra 16 bitų dydžio. Kiekvienas bitas vadinamas žyme (2.3 pav.) ir gali turėti reikšmę **1** arba **0**.

- **Carry Flag (CF)** - pernešimo žymė. Lygi **1**, kai stebimas **beženklis perpildymas**. Pavyzdžiui, sudedant baitus **255 + 1** (rezultatas viršija intervalą 0...255). Kai perpildymo nėra, tada žymė turi reikšmę **0**.

- **Zero Flag (ZF)** - nulinė žymė. Lygi **1**, kai rezultatas nulis. Esant nenuliniam rezultatui, žymės reikšmė **0**.

- **Sign Flag (SF)** - ženklo žymė. Lygi **1**, kai rezultatas **neigiamas**. Kai rezultatas **teigiamas**, tada reikšmė lygi **0**.

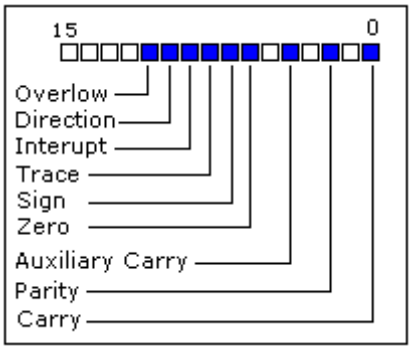
- **Overflow Flag (OF)** - perpildymo žymė. Lygi **1**, kai stebimas **ženklinis perpildymas**. Pavyzdžiui, kai sudedame baitus **100 + 50** (rezultatas nepakliūva į intervalą -128...127).

- **Parity Flag (PF)** - pariteto žymė. Lygi **1**, kai rezultatas turi lyginį vienetinių bitų skaičių ir lygi **0**, kai rezultatas turi nelyginį vienetinių bitų skaičių. Jei rezultatas yra žodžio ilgio, tada analizuojami tik 8 jaunesnieji bitai.

- **Auxiliary Flag (AF)** - pagalbinė žymė. Lygi **1**, kai stebimas **beženklis perpildymas** jaunesniajame pusbaityje (4 bituose).

- **Interrupt enable Flag (IF)** - pertraukimo galimumo žymė. Kai ši žymė lygi **1**, procesorius reaguoja į išorinių įrenginių pertraukimus.

- **Direction Flag (DF)** - krypties žymė. Ši žymė kai kuriose instrukcijose naudojama duomenų sekos apdorojimo tvarkai nurodyti. Kai ji lygi **0** - duomenų apdorojimas vykdomas einant į priekį (forward), kai ji lygi **1**, duomenų apdorojimas vykdomas atgaliniu būdu (backward).



2.3 pav. Žymių registro struktūra.

Bendru atveju negalime į šiuos registrus kreiptis tiesiogiai.

2.3 pavyzdys. Programoje emu8086 įvykdykite pavyzdį **4_sample.asm**. Pažingsniui vykdydami programą stebėkite kaip keičiasi žymių registro ir instrukcijų rodyklės turinys programos vykdymo metu (žymių registro bitų pasikeitimams stebėti įjunkite mygtuką „Flags“).

2.3 pratimas. Mokėkite paaiškinti kas lemia žymių registro bitų pasikeitimą kiekvienu atveju, kai vykdoma programa **4_sample.asm**.

2.2 Atminties adresavimas

8086/8088 architektūros mikroprocesoriuose bet kurio baito adresas užduodamas dviem 16 bitų žodžiais – segmentu ir postūmiu. Formuojant 20 bitų fizinį adresą, būtino 1Mbaito adresacijos ribose, segmentas pastumiamas į kairę per keturis bitus (padauginamas iš 16) ir sudedamas su postūmiu. Kadangi 16 bitų postūmio talpa yra 65536 reikšmės, vieno segmento ribose galima adresuoti iki 64 Kbaitų.

Mikroprocesoriaus architektūra leidžia naudoti septynis skirtingus atminties adresavimo būdus.

Registrinis adresavimas. Adresuojant šiuo būdu operandas yra išimamas iš registro arba įrašomas į jį. Pavyzdžiui,

```
MOV ax,bx       ;BX reikšmę įrašome į AX
ADD cx,ax       ;AX turinį pridedame prie CX
PUSH cx         ;CX turinį įdedame į steką
```

Tiesioginis adresavimas. Operandas (8 arba 16 bitų konstanta) būtinai yra komandos dalis. Pavyzdžiui,

```
MOV ax,100      ;į AX įrašome reikšmę 100
ADD ax,5        ;prie AX turinio pridedame 5
MOV cx,$FFFF    ;į CX patalpinam reikšmę 65535
```

Adresavimas naudojant kintamuosius. Operando postūmis užduodamas programos kūne ir sudedamas su registru DS. Pavyzdžiui,

```
MOV AX, X       ;kintamojo X reikšmę persiunčiame į registrą AX
ADD AH, B       ;prie registro AH turinio pridedame kintamojo B reikšmę
MOV X, AX       ;registro AX turinį persiunčiame į kintamojo X reikšmei
                ;saugoti skirtą atminties vietą

X DW 15         ;apibrėžiame 16 bitų kintamąjį X
B DW 5          ; apibrėžiame 16 bitų kintamąjį B
```

Netiesioginis adresavimas. Vykdomasis operando adresas (tiksliau, jo postūmis) saugomas viename iš registrų BX, BP, SI arba DI. Norėdami nurodyti netiesioginę adresaciją, registras turi būti užrašomas laužtiniuose skliaustuose. Pavyzdžiui,

```
MOV ax, [bx]    ;16 bitų žodžio turinys, saugomas atminties adresu DS:BX,
                ;persiunčiamas į registrą AX
```

Kiekvienas registras BX, BP, SI, DI pagal nutylėjimą dirba su savo segmento registru:

DS:BX, SS:BP, DS:SI, ES:DI

Leidžiama ir tiesiogiai nurodyti segmento registrą, jei jis skiriasi nuo nutylimojo. Pavyzdžiui,

```
MOV ax, es:[bx]
```

Bazinis adresavimas. Bazės registras BX (arba BP) saugo bazę (tam tikro atminties fragmento pradžios adresą), nuo kurios asembleris apskaičiuoja postūmį. Pavyzdžiui,

```
MOV ax, [bx]+10 ;į AX pakrauname dešimtą nuo bazės pradžios pagal eilę
                ;baitą, esantį adresu DS:BX
```

Indeksinis adresavimas. Vienas iš indeksinių registrų SI arba DI rodo elemento padėtį tam tikros atminties srities pradžios atžvilgiu. Pavyzdžiui, tegu AOB žymi baito tipo reikšmių masyvą. Tada galima naudoti tokius kodo fragmentus:

```
MOV si,15          ;į SI patalpiname konstantą 15
MOV ah, AOB[si]    ;į AH persiunčiame šešioliktą nuo masyvo pradžios baitą,
                   ;šešioliktas masyvo elelentas, skaičiuojant nuo 0
MOV si,0
MOV AOB[si],ah     ;gautą reikšmę įrašome į patį pirmą masyvo elementą
```

Bazinis indeksinis adresavimas. Tai indeksinės adresacijos variantas, kai indeksuojama atminties sritis užduodama savo baze. Pavyzdžiui,

```
MOV ax, [bx] [si]
```

Šis adresavimo būdas patogus dirbant su dvimačiais masyvais. Pavyszdžiui, jei AOB yra dvimatis baito tipo kintamųjų masyvas ir norime kreiptis į elementą AOB [2,3], tada galime užrašyti tokį kodo fragmentą:

```
MOV bx,20                ;antros eilutės bazė
MOV si,2                 ;trečio elemento numeris
MOV ax, AOB [bx] [si]    ;nurodytą elementą įrašome į AX
```

Kreipimuisi į atmintį naudojame šiuos keturis registrus: **BX, SI, DI, BP**.

Apjungdami šiuos registrus tarp laužtinių skliaustų **[ ]**, galime pasiekti skirtingas atminties vietas. Žemiau pateikiamos leistinos registrų kombinacijos - adresavimo režimai (addressing modes):

[BX + SI]	[SI]	[BX + SI] + d8
[BX + DI]	[DI]	[BX + DI] + d8
[BP + SI]	d16 (variable offset only)	[BP + SI] + d8
[BP + DI]	[BX]	[BP + DI] + d8
[SI] + d8	[BX + SI] + d16	[SI] + d16
[DI] + d8	[BX + DI] + d16	[DI] + d16
[BP] + d8	[BP + SI] + d16	[BP] + d16
[BX] + d8	[BP + DI] + d16	[BX] + d16

d8 - reiškia 8 bitų poslinkį (perkėlimą).
d16 - reiškia 16 bitų poslinkį (perkėlimą).

Poslinkis gali būti tiesioginė kintamojo reikšmė arba kintamojo postūmis (offset), arba abu. Kompiliatorius apskaičiuoja bendrą tiesioginę reikšmę.

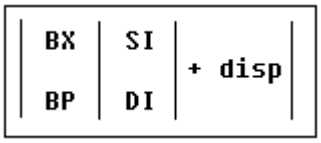
Poslinkis gali būti nurodomas laužtinių skliaustų **[ ]** viduje arba išorėje, kompiliatorius generuoja tokį pat mašininį kodą abiem atvejams. Poslinkis gali turėti tiek **teigiamą**, tiek **neigiamą ženklą**. Bendru atveju kompiliatorius rūpinasi skirtumu tarp d8 ir d16 ir generuoja reikalingą mašininį kodą.

Pavyzdžiui, tarkime , kad **DS = 100**, **BX = 30**, **SI = 70**.

Fiziniam adresui **100 * 16 + 30 + 70 + 25 = 1725** procesorius nustato tokį adresavimo režimą **[BX + SI] + 25**.

Įprasta **DS** segmento registrą naudoti visiems adresavimo režimams, išskyrus tuos kuriems naudojamas **BP** registras, šiems naudojamas **SS** segmento registras.

Toliau pateikiamas būdas kaip įsiminti visas įmanomas registrų kombinacijas:



Kombinacijos sudaromos imant po vieną elementą iš kiekvieno stulpelio, apjungiant jį su elementu (-ais) iš kitų stulpelių. Galima ir nejungti su jokiu kitu elementu.

Kaip matome **BX** ir **BP** niekada neina kartu. **SI** ir **DI** taip pat. **[BX+5]** yra galimo adresavimo režimo pavyzdys.

Reikšmė segmento registre (CS, DS, SS, ES) yra vadinama "**segmentu**", o reikšmė paskirties registre (BX, SI, DI, BP) yra vadinama "**postūmiu**".

Kai DS reikšmė **1234h,** o SI reikšmė **7890h**, tai galima taip pat užrašyti **1234:7890**. Fizinis adresas bus 1234h * 10h + 7890h = 19BD0h.

Kompiliatoriui užduodant duomenų tipą, naudojame tokius prefiksus:

BYTE PTR - baito tipui.
WORD PTR - žodžiui (du baitai).

Pavyzdžiui:
BYTE PTR [BX] ; baito kreiptis.
arba
WORD PTR [BX] ; žodžio kreiptis.
emu8086 palaiko ir trupmesnes priesagas:
b. - **BYTE PTR**
w. - **WORD PTR**

Kartais kompiliatorius nustato duomenų tipą automatiškai, bet negalime pasikliauti, kad vienas iš operandų yra tiesioginė reikšmė.

2.4 pavyzdys. Programoje emu8086 įvykdykite pavyzdį **1_sample.asm**. Pažingsniui vykdydami programą stebėkite kaip keičiasi registrų turinys programos vykdymo metu.

2.4 pratimas. Mokėkite paaiškinti visus atminties adresavimo būdus, kurie buvo panaudoti pavyzdyje **1_sample.asm** duomenims saugoti ir išvesti.

2.3 Instrukcija MOV

- Nukopijuoja **antrąjį operandą** (šaltinį) į **pirmąjį operandą** (paskirties vietą).
- Šaltinio operando reikšmė gali būti tiesioginė reikšmė, bendrosios paskirties registras arba atmintinės vieta.
- Paskirties operandas gali būti bendrosios paskirties registras, arba atmintinės vieta.
- Abu operandai turi būti tokio pat dydžio: baito arba žodžio dydžio.

Leistini operandų tipai:

```
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
```

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
memory: [BX], [BX+SI+7], kintamasis, ir pan...
immediate: 5, -24, 3Fh, 10001101b, t.t...

Segmento registrams komandos **MOV** leistinas pavidalas:

```
MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
```

SREG: DS, ES, SS, ir tik kaip antrasis operandas: CS.
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
memory: [BX], [BX+SI+7], kintamasis, ir pan...

Instrukcija MOV negali būti naudojama registrų CS ir IP reikšmių priskyrimui.

2.5 pavyzdys. Žemiau pateikiama programa demonstruoja MOV instrukcijos naudojimą:

```
#MAKE_COM#          ;direktyva kompiliatoriui sukurti COM bylą.
ORG 100h            ;direktyva reikalinga COM programai.
MOV AX, 0B800h      ;įkrauti AX šešioliktaine reikšme B800h.
MOV DS, AX          ;nukopijuoti AX reikšmę į DS.
MOV CL, 'A'         ;įkrauti CL ASCII kodu 'A', tai yra 41h.
MOV CH, 01011111b   ;įkrauti CH dvejetaine reikšme.
MOV BX, 15Eh        ;įkrauti  BX  15Eh.
MOV [BX], CX        ;nukopijuoti CX turinį į atminties vietą B800:015E.
RET                 ;grįžti į operacinę sistemą.
```

2.5 pratimas. Realizuokite 2.5 pavyzdžio programą emu8086 asembleryje.

2. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Tegu segmento registre DS yra reikšmė 1578h, o registre SI reikšmė 0850h. Apskaičiuokite fizinį adresą, kurį nustatys procesorius?	15EC0h	Atsakymas neteisingas. Fizinis adresas apskaičiuojamas padauginus segmento registro turinį iš 10h ir gautą rezultatą sudėjus su segmento registru poroje dirbančio registro turiniu.
		1515130h	Atsakymas neteisingas. Fizinis adresas apskaičiuojamas padauginus segmento registro turinį iš 10h ir gautą rezultatą sudėjus su segmento registru poroje dirbančio registro turiniu.
		15FD0h	Atsakymas teisingas.
2.	Kaip vadinamas šis adresavimo būdas MOV ax,bx ?	Netiesioginis adresavimas	Atsakymas neteisingas. Adresuodami operandą netiesioginės adresacijos būdu operandą užrašome laužtiniuose skliaustuose.

		Bazinis adresavimas	Atsakymas neteisingas. Adresuodami baziniu būdu bazės registras užrašomas laužtiniuose skliaustuose .
3.	Instrukcija MOV gali būti naudojama šių registrų reikšmių priskyrimui	CS, IP, CX	Atsakymas neteisingas. Instrukcija MOV negali būti naudojama registrų CS ir IP reikšmėms priskirti.
		CS, CX, AX	Atsakymas neteisingas. Instrukcija MOV negali būti naudojama registrų CS ir IP reikšmėms priskirti.
		CX, BX, AX	Atsakymas teisingas.

Papildomi skaitiniai

[1] B. B. Фаронов, Turbo Pascal 7.0. Начальний курс. Москва, 1997. – 616 c. (233 p. – 238 p.)
[2] A. Mitašiūnas, Kompiuterių architektūra, Mokymo priemonė. VU Informatikos katedra, 2003. – 126 p. (35 p. – 63 p.)
[3] B. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 c. (38 p. – 51 p.)

3. Kintamieji

Kintamasis - tai atminties vieta. Daug patogiau kintamojo reikšmę saugoti kaip kintamojo pavadinimą pvz. "**var1**", nei kaip adresą pvz. 5A73:235B, juolab kai turime daug kintamųjų.

Kompiliatorius su kuriuo dirbame palaiko dviejų tipų kintamuosius: **BYTE** ir **WORD**.

Kintamojo apibrėžimo sintaksė asembleryje yra tokia:

pavadinimas **DB** *reikšmė*
pavadinimas **DW** *reikšmė*

DB - Define Byte (apibrėžiamas baito tipo kintamasis).

DW - Define Word (apibrėžiamas žodžio tipo kintamasis).

pavadinimas - gali būti bet kokia raidžių ir skaičių kombinacija, prasidedanti raide. Įmanoma apibrėžti neįvardintus kintamuosius neapibrėžiant jiems pavadinimo. Tokie kintamieji turi adresą, bet neturi pavadinimo.

reikšmė - gali būti bet kokia skaitinė reikšmė bet kurioje palaikomoje skaičiavimo sistemoje arba "**?**" simbolis kintamiesiems, kurie nėra inicializuoti.

Kaip jau žinome instrukcija **MOV** naudojama kopijuoti reikšmėms iš vienos vietos į kitą. Kitas instrukcijos **MOV** naudojimo pavyzdys galėtų būti toks:

```
#MAKE_COM#
ORG 100h
MOV AL, var1
MOV BX, var2
RET ; sustabdo programą.
VAR1 DB 7
var2 DW 1234h
```

3.1 pratimas. Įvykdykite aukščiau pateiktą programą emu8086 asembleryje. Atkreipkite dėmesį į disasemblerio lange rodomas reikšmes, kai atliekama instrukcija **MOV AL, var1** bei **MOV BX, var2**. Paaiškinkite kodėl taip yra?

Kaip matote įvykdę programą, kodas beveik nesiskiria nuo pavyzdžio, išskyrus tai, kad kintamieji pakeisti tikromis atminties vietomis. Kada kompiliatorius kodą verčia į mašininį kodą, jis automatiškai kintamųjų pavadinimus pakeičia jų **postūmiais** (offsets). Pagal nutylėjimą segmentas užkraunamas į **DS** registrą (kada užkraunamas COM failas **DS** registro reikšmė sutapatinama su **CS** registro - kodo segmento - reikšme).

Kompiliatorius neskiria didžiųjų mažųjų raidžių, taigi "**VAR1**" ir"**var1**" yra tas pats kintamasis.

Kintamojo **VAR1** postūmis yra **0108h**, o pilnas adresas **0B56:0108**. Kintamojo **var2** postūmis **0109h**, o pilnas adresas **0B56:0109**, šis kintamasis yra **WORD** tipo, taigi jis užima **2 baitus**. Laikomasi taisyklės, kad jaunesnysis baitas yra išsaugomas žemesniame adrese, taigi **34h** yra patalpinamas prieš **12h**.

Matome, kad po instrukcijos **RET** yra dar ir kitos instrukcijos. Taip yra dėl to, kad procesorius tiesiog apdoroja reikšmes atmintyje ir supranta jas kaip galiojančias 8086 instrukcijas.

Tą pačią programą galime parašyti naudodami tik DB direktyvą:

```
#MAKE_COM#
ORG 100h
DB 0A0h
DB 08h
DB 01h
DB 8Bh
```

```
DB 1Eh
DB 09h
DB 01h
DB 0C3h
DB 7
DB 34h
DB 12h
```

3.2 pratimas. Įvykdykite aukščiau pateiktą direktyvų seką emu8086 asembleryje ir įsitikinkite, kad ji padaro tą patį kaip ir 3.1 pratimo programa.

Kompiliatorius tiesiog perverčia programos kodą į baitų aibę, kuri vadinama **mašininiu kodu**. Procesorius supranta **mašininį kodą** ir jį vykdo.

ORG 100h - kompiliatoriaus direktyva (ji pasako kaip traktuoti šaltinio kodą). Ši direktyva labai svarbi dirbant su kintamaisiais. Ji nurodo, kad paleidžiamasis failas bus užkraunamas postūmiu 100h (256 baitai), taigi kompiliatorius turi apskaičiuoti teisingą adresą visiems kintamiesiems kintamųjų pavadinimų pakeitimo jų **postūmiais** metu. Direktyvos niekada neverčiamos į jokį realų **mašininį kodą**.

Kodėl paleidžiamasis failas užkraunamas postūmiu 100h? Operacijų sistema saugo tam tikrus duomenis apie programą pirmuosiuose **CS** (kodo segmento) 256 baituose, tokius kaip komandų eilutės parametrus ir pan. Tačiau, tai galioja tik **COM** failams, **EXE** failai užkraunami postūmiu **0000** ir bendru atveju kintamiesiems naudoja specialų segmentą.

3.3 pratimas. Programą **2_sample.asm** modifikuokite taip, kad veiksmai būtų atliekami ne su tiesioginėmis reikšmėmis, o su kintamaisiais. Apibrėžkite baito tipo kintamuosius **a** ir **b**, jiems priskirdami atitinkamas reikšmes **5** ir **10** šešioliktainėje sistemoje.

3.1 Masyvai

Masyvus galime įsivaizduoti kaip kintamųjų grandines. Teksto eilutė yra baito masyvo pavyzdys, kur kiekvienas simbolis yra atvaizduojamas kaip ASCII kodo reikšmė (0..255). Masyvą galime apibrėžti taip:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

b yra tiksli **a** masyvo kopija. Kai kompiliatoriui pateikiama eilutė tarp kabučių, kompiliatorius automatiškai verčią ją į baitų aibę.

Į masyvo elementą ir jo reikšmę galima kreiptis naudojant laužtinius skliaustus, pavyzdžiui:

```
MOV AL, a[3]
```

Tai indeksinio adresavimo pavyzdys. Taip pat galima naudoti ir registrus **BX, SI, DI, BP**, pavyzdžiui:

```
MOV SI, 3
MOV AL, a[SI]
```

Jei reikia apibrėžti didelį masyvą galima naudoti operatorių **DUP**. Operatoriaus **DUP** sintaksė yar tokia:

skaičius DUP (reikšmė (s))

skaičius - reikalingų reikšmės kopijų skaičius (bet kokia konstanta).
reikšmė - išraiška, kuri bus pakartota DUP operatoriumi nurodytą kartų skaičių .

3.1 pavyzdys. Masyvus galime apibrėžti pavyzdžiui taip:

```
c DB 5 DUP(9)

yra tas pats kaip:
c DB 9, 9, 9, 9, 9


kitas pavyzdys:
d DB 5 DUP(1, 2)

yra tas pats kaip:
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Žinoma, galime naudoti **DW** vietoje **DB** jei reikalinga išsaugoti reikšmes, didesnes nei 255, arba mažesnes nei -128. **DW** negali būti naudojamas eilutėms apibrėžti!

DUP operatoriaus plotis negali viršyti 1020 simbolių. Paskutiniojo pavyzdžio plotis yra 13 simbolių. Jei reikia apibrėžti didžiulį masyvą, patartina apibrėžimą suskaidyti į dvi eilutes.

3.4 pratimas. Sudarykite programą, kuri paeiliui sumuoja masyvo narius. Masyvo nariai - dešimtainiai skaičiai. Gautą rezultatą išspausdinkite ekrane.

3.2 Kintamojo adreso nustatymas

Šiam tikslui naudojame **LEA** (Load Effective Address) instrukciją ir alternatyvų **OFFSET** operatorių. Abu **OFFSET** ir **LEA** gali būti naudojami kintamojo postūmio adresui gauti.

LEA turi daugiau privalumų, kadangi leidžia gauti indeksuotų kintamųjų adresus. Kintamojo adreso gavimas gali būti labai naudingas tam tikrose situacijose, pavyzdžiui kai reikia perduoti parametrus į procedūrą.

```
ORG 100h
MOV AL, VAR1          ;nustatyti VAR1 reikšmę kopijuojant jį į AL.
LEA BX, VAR1          ;įrašyti VAR1 adresą į BX.
```

```
MOV BYTE PTR [BX], 44h      ;redaguoti VAR1 turinį.
MOV AL, VAR1                ;patikrinti pakeitimus VAR1 kopijuojant jį į AL.
RET

VAR1 DB 22h
END
```

Kitas pavyzdys, kuriame vietoje **LEA** naudojame **OFFSET**:

```
ORG 100h
MOV AL, VAR1                ;nustatyti VAR1 reikšmę kopijuojant jį į AL.
MOV BX, OFFSET VAR1         ;įrašyti VAR1 adresą į BX.
MOV BYTE PTR [BX], 44h      ;redaguoti VAR1 turinį.
MOV AL, VAR1                ;patikrinti pakeitimus VAR1 kopijuojant jį į AL.
RET

VAR1 DB 22h
END
```

Abu pavyzdžiai vienodai funkcionalūs. Eilutės

```
LEA BX, VAR1
MOV BX, OFFSET VAR1
```

yra net gi perverčiamos į tokį pat mašininį kodą:

```
MOV BX, num
```

Čia **num** kintamojo postūmio 16 bitų reikšmė.

Reikia įsidėmėti, kad tik registrai BX, SI, DI, BP gali būti naudojami tarp laužtinių skliaustų (kaip atminties rodyklės)!

3.5 pratimas. Nukopijuokite abu šioje temoje pateiktus pavyzdžius į emu8086. Vykdydami programas pažingsniui stebėkite kaip veikia instrukcija **LEA** bei operatorius **OFFSET**. Mokėkite paaiškinti jų veikimo principą.

3.2 Konstantos

Konstantos apibrėžiamos panašiai kaip ir kintamieji, tačiau jos gyvuoja iki tol, kol programa sukompiliuojama. Apibrėžus konstantą jos reikšmės negalime pakeisti. Konstantai apibrėžti naudojame direktyvą **EQU**:

pavadinimas EQU < bet kokia išraiška >

Pavyzdžiui:

```
k EQU 5
MOV AX, k
```

Pateiktas pavyzdys ekvivalentus kodui:

MOV AX, 5

3.6 pratimas. Sudarykite programą, kuri iš klaviatūros įvestą skaičių pakelia **5** laipsniu. Laipsnį apibrėžkite kaip konstantą. Kėlimą laipsniu realizuokite kaip sandaugą cikle. Ciklui sudaryti naudokite instrukciją **Loop** kartu su registru **CX**.

3. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras

1.	Kintamieji k1 ir k2 apibrėžti tokiu būdu:
	
	k1 DB 10
	k2 DW 11
	
	kuris iš pateiktų programos fragmentų veiks be klaidų?
		MOV AL, k2
		MOV CX, k1
		M1:
		ADD AL, AL
		LOOP M1
			Atsakymas neteisingas. Pirmoji klaida pasirodys, kai į 8 bitų registrą AL bandysime įrašyti 16 bitų dydžio kintamojo k2 reikšmę. Antroji, kai į žodžio dydžio registrą CX, rašysime baito dydžio efektyvųjį adresą (kintamojo k1 reikšmę).

		MOV AX, k2
		MOV CX, k1
		M1:
		ADD AL, AL
		LOOP M1
			Atsakymas neteisingas. Kadangi į žodžio dydžio registrą CX rašome baito dydžio efektyvųjį adresą (kintamojo k1 reikšmę).

		MOV AX, k2
		MOV CL, k1
		M1:
		ADD AL, AL
		LOOP M1
			Atsakymas teisingas.

2.	Kuriuo sakiniu apibrėžiame masyvą mas1, sudarytą iš 10 kartų pasikartojančių	mas1 DW 10 (2,4)	Atsakymas teisingas.

	žodžio dydžio elementų 2,4?	mas1 DB 10 (2,4)	Atsakymas neteisingas. Šiuo sakiniu apibrėšime baito dydžio elementų 2,4 pasikartojantį masyvą.
		mas1 DW (2,4) 10	Atsakymas neteisingas. Šis užrašas sintaksiškai nekorektiškas. Pirmiau turime nurodyti kiek kartų pasikartoja, o tada kokie elementai.
3.	Turime kodo fragmentą: XOR AX,AX LEA BX, k1 MOV AX,BX k1 DW 5 Ir fragmentą: XOR AX,AX MOV BX, OFFSET k1 MOV AX,BX k1 DW 5 Kokia bus regsitro AX reikšmė po pirmojo ir antrojo kodo fregmentų įvykdymo?	Pirmuoju atveju reikšmė bus didesnė nei antruoju.	Atsakymas neteisingas, kadangi reikšmės sutaps. Tiek instrukcija LEA, tiek operatorius OFFSET į registrą įkraus kintamojo efektyvųjį adresą, kuriame išsaugota kintamojo k1 reikšmė.
		Reikšmės sutaps.	Atsakymas teisingas.
		Pirmuoju atveju reikšmė bus mažesnė nei antruoju.	Atsakymas neteisingas. Tiek instrukcija LEA, tiek operatorius OFFSET į registrą įkraus kintamojo efektyvųjį adresą, kuriame išsaugota kintamojo k1 reikšmė.

Papildomi skaitiniai

[1] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (38 p. – 51 p.)

4. Pertraukimai

Pertraukimai (angl.k. *interruption*) – gavo savo pavadinimą atsižvelgus į jų pagrindinę funkciją – pristabdyti pagrindinių skaičiavimų procesą centriniame procesoriuje, siekiant įvykdyti pagalbinius, technologinius ir tarnybinius veiksmus operacijų sistemoje.

Pertraukimus galime įsivaizduoti kaip tam tikrą eilę funkcijų. Šios funkcijos labai palengvina programavimą. Pavyzdžiui, vietoj kodo, kuris turėtų išspausdinti simbolį, rašymo galime paprasčiausiai iškviesti pertraukimą, kuris atliks tuos pačius veiksmus.

Pertraukimų apdorojimas IBM PC tipo kompiuteriuose susijęs su mašininės komandos **INT N** įvykdymu. Čia **N** pertraukimo vektoriaus numeris. Pagal komandos INT N išrinkimo procesoriumi iš duomenų šynos būdą, pertraukimai skirstomi į programinius (programinės įrangos pertraukimai) ir aparatūrinius (techninės įrangos pertraukimai).

Programinio (vidinio) pertraukimo metu komanda INT N yra įtraukta į programos kodą, kitaip sakant yra programos viduje. **Aparatūrinio** (išorinio) pertraukimo metu komanda INT N "įtraukiama" į mašininių instrukcijų seką aparatūrinėmis pertraukimų valdiklio (kontrolerio) priemonėmis, kuris savo ruožtu inicializuojamas vidiniu pertraukimo pareikalavimo signalu.

Taigi pertraukimų apdorojimo programos turi unikalią galimybę būti iškviestos iš bet kurios atmintinėje įkrautos programos, pasitelkiant komandą INT N. Iš tikrųjų, ši komanda perduoda valdymą adresu, kursi įrašytas viename iš 256 4 bitų dydžio vektorių, kurie savo ruožtu nuosekliai patalpinti ix86 procesoriaus atmintinės pirmajame kilobaite. Visi pertraukimai asembleryje programuojami kaip specialios procedūros su grįžimu į nutrauktą uždavinį tam tikslui naudojant komandą IRET. Kreiptis į DOS ir BIOS funkcijas vykdoma standartiniu būdu, valdymo perdavimui naudojant komandą INT.

Bet kuriuo atveju, procesorius pristabdo uždavinio sprendimą ir vykdo pertraukimą, o tada grįžta į prieš tai buvusią sprendimo vietą. Tam, kad procesorius turėtų galimybę tiksliai grįžti į reikiamą programos vietą, šios vietos adresas (CS:IP) įsimenamas steke kartu su žymiu registru. Tuomet į CS:IP pakraunamas pertraukimo apdorojimo programos adresas ir jai perduodamas valdymas. Pertraukimo apdorojimo programos dar vadinamos pertraukimų valdikliais (handler). Šios programos visada užsibaigia komanda IRET (grįžimas iš pertraukimo), kuri užbaigia pertraukimo iškvietimu pradėtą procesą, t. y. grąžina prieš tai buvusias CS:IP ir žymių registro reikšmes. Tokiu būdu duodama procesoriui galimybę toliau vykdyti programą nuo tos vietos, kurioje ji buvo nutraukta.

Didžiausią dėmesį skirsime programinės įrangos pertraukimams. Programinės įrangos pertraukimams iškviesti naudojame instrukciją **INT**, jos sintaksė labai paprasta:

INT N

N gali būti skaičius tarp 0 ir 255 (arba 0 ir 0FFh), bendru atveju naudosime šešioliktainius skaičius. Galime pamanyti, kad yra tik 256 funkcijos, tačiau tai

netiesa. Kiekvienas pertraukimas gali turėti sub-funkcijas. Sub-funkcija apibrėžiama įkraunant registrą **AH**, o tik tada iškviečiamas pertraukimas.

Kiekvienas pertraukimas gali turėti 256 sub-funkcijas (taigi turime 256 * 256 = 65536 funkcijas). Bendru atveju naudojame **AH** registrą, bet kartais gali būti panaudoti ir kiti. Bendru atveju kiti registrai naudojami parametrų ir duomenų perdavimui sub-funkcijoms. Net gi palyginti trumpa galimų pertraukimų suvestinė su minimaliu sintaksės ir jų vykdymo semantikos aprašymu užima pakankamos apimties dvitomį.

Žemiau pateiktas pavyzdys naudoja **INT 10h** sub-funkciją **0Eh** ir spausdina pranešimą "Hello!". Šios funkcijos spausdina simbolį ekrane, perkeldamos kursorių ir prasukdamos ekraną jei reikia.

```
#MAKE_COM#
ORG 100h
MOV AH, 0Eh       ;sub-funkcijos pasirinkimas. INT 10h / 0Eh sub-funkcija
                  ;gauna simbolio, kuris bus įkrautas į registrą AL, ASCII kodą.
MOV AL, 'H'       ;ASCII kodas: 72
INT 10h           ;išspausdinti
MOV AL, 'e'
INT 10h           ;išspausdinti
MOV AL, 'l'
INT 10h           ;išspausdinti
MOV AL, 'l'
INT 10h           ;išspausdinti
MOV AL, 'o'
INT 10h           ;išspausdinti
MOV AL, '!'
INT 10h           ;išspausdinti
RET               ;grįžti į operacijų sistemą
```

Su emu8086 palaikomais pertraukimus susipažinsime pasinaudodami emu8086 pagalbos sistemos byla **list of supported interrupts**.

4.1 pratimas. Emu8086 iš Samples katalogo atverkite bylas:

a) **int10_13.asm**. Šis pavyzdys demonstruoja **10h** pertraukimo 13h (spausdina ekrane eilutę) ir 0Eh (spausdina ekrane po vieną simbolį) sub-funkcijų veikimą;
b) **int21.asm**. Šis pavyzdys iliustruoja **21h** pertraukimo 0Ah (nuskaito eilutę), 9h (išspausdina eilutę) ir 4Ch (perduota valdymą OS) sub-funkcijų veikimą.

Paeiliui įvykdykite programas. Žinokite pertraukimų atributų (sub-funkcijų) prasmę, mokėkite paaiškinti programoje panaudotas instrukcijas, žinokite jų paskirtį.

4. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras

1.	Pateiktas kodo fragmentas:
	
	MOV AX, CS
	MOV DS, AX
	MOV DX, OFFSET string
	MOV AH, 09h
	INT 21h
	RET
	
	string DB ‘absc $’
	
	Kas įvyks po šio kodo fragmento įvykdymo?
		Bus nuskaityta ir adresu DS:DX išsaugota eilutė abc.	Atsakymas neteisingas. Norint nuskaityti simbolių eilutę reikėtų naudoti 21h pertraukimo 0Ah atributą.

		Ekrane bus išspausdinta eilutė abc.	Atsakymas teisingas.
		Ekrane bus išspausdintas simbolis a.	Atsakymas neteisingas. Notrėdami išspausdinti vieną simbolį turėtume pasinaudoti 21h pertraukimo atributu 02h. Tada turėtume pertvarkyti ir čia pateiktą programos fragmentą.

2.	Turime programos fragmentą:
	
	MOV AL, 66h
	MOV DH, 2
	MOV DL, 3
	MOV AH, 02h
	INT 10h
	MOV AH, 9h
	MOV BH, 0
	MOV BL, 0F2h
	MOV CX, 5
	INT 10h
	RET
	
	Į kokį registrą ir kokią reikšmę turime įrašyti, jei norime, kad simbolis “f “ ekrane būtų išspausdintas 10 kartų?
		Į registrą CX turime įrašyti reišmę 10h	Atsakymas neteisingas. Jei įrašysime reikšmę 10 h į regsitrą CX simbolis “f” bus išspausdintas 16 kartų.

		Į registrą BH turime įrašyti reikšmę 0Ah	Atsakymas neteisingas. Registre BH šiuo atveju nurodomas ekrano puslapio numeris. Leistini numeriai yra nuo 0 iki 7. Įvykdžius programą, nieko nepamatysime.
		Į registrą CX turime įrašyti reikšmę 0Ah.	Atsakymas teisingas.
3.	Kuri instrukcija visada užbaigia pertraukimo apdorojimo programą?	RET	Atsakymas neteisingas. Šia instrukcija programa grąžina valdymą operacijų sistemai.

		INT	Atsakymas neteisingas. Šia komanda iškviečiamo numerio pertraukimas.

Papildomi skaitiniai

[1] В. И. Пустоваров, Язык Ассемблер в програмировании информационных и управляющих систем. Москва, «ЭНТРОП», Киев, «Век» 1997. – 304 с. (113 p. – 142 p.) ISBN 5-88547-052-9

5. Procedūros

Procedūros yra kodo dalis, kuri gali būti iškviečiama pagrindinės programos kodo vykdymo metu. Procedūromis dažniausiai aprašomos tam tikros specifinės užduotys. Jos paverčia programą labiau struktūrizuota ir lengviau suprantama. Bendru atveju po procedūros iškvietimo ir jos įvykdymo grįžtama kodo eilute žemiau, kur buvo iškviesta procedūra. Procedūra apibrėžiama taip:

```
pavadinimas PROC
; čia rašome
; procedūros kodą ...
RET
pavadinimas ENDP
```

pavadinimas - tai procedūros pavadinimas. Toks pat pavadinimas turi būti procedūros pradžioje ties direktyva PROC bei procedūros pabaigoje ties direktyva ENDP. Tai reikalinga siekiant korektiškai užbaigti procedūrą. Jau žinome, kad instrukcija RET valdymą grąžina operacijų sistemai. Ta pati instrukcija procedūrose naudojama gražinti valdymą pagrindinei programai.

PROC ir **ENDP** yra kompiliatoriaus direktyvos, taigi jos netransliuojamos į jokį realų mašininį kodą. Kompiliatorius tiesiog įsimena procedūros adresą.

Instrukcija **CALL** naudojama procedūrai iškviesti. Pavyzdžiui:

```
CALL m1         ;kviečiame procedūrą m1
MOV AX, 2
RET             ;grįžti į OS

m1 PROC         ;procedūros m1 pradžia
MOV BX, 5
RET             ;grįžti iš procedūros
m1 ENDP         ;procedūros m1 pabaiga

END             ;programos pabaiga
```

Aukščiau pateiktas pavyzdys iškviečia procedūrą **m1**, įvykdo **MOV BX, 5** ir grįžta prie instrukcijos **MOV AX, 2**, kuri eina iškart po instrukcijos **CALL**.

Procedūrai parametrus perduoti galima keliais būdais. Lengviausias yra naudojantis registrais. Žemiau pateikiamas procedūros pavyzdys, kur ji parametrus gauna naudojantis registrais **AL** ir **BL**, sudaugina juos ir grąžina rezultatą į registrą **AX**.

```
ORG 100h
MOV AL, 1
MOV BL, 2
CALL m2
CALL m2
CALL m2
CALL m2
RET             ;grįžti į OS.

m2 PROC
MUL BL          ;AX = AL * BL.
RET             ;grįžti į iškvietimo vietą.
m2 ENDP

END
```

Šiame pavyzdyje registro **AL** reikšmė atnaujinama kiekvieną kartą, kai iškviečiama procedūra. **BL** registras išlieka nepakitęs. Taigi šis algoritmas apskaičiuoja dvejeto ketvirtąjį laipsnį ir galutinis rezultatas yra registre **AX**, t. y. **16** (arba **10h**).

5.1 pavyzdys. Sudarysime procedūrą, apskaičiuojančią nurodyto skaičiaus faktorialą (pavyzdžiui 5!=1*2*3*4*5). Paprastumo dėlei, skaičių, kurio faktorialą norime apskaičiuoti, nurodysime tiesiog programos kode.

```
ORG 100h
MOV BX, 5h      ;skaičius, kurio faktorialą skaičiuosime
CALL fakt       ;kviečiame procedūrą fakt
RET

fakt PROC       ;procedūros fakt apibrėžties pradžia
MOV AX, BX      ;AX=BX
DEC BX          ;BX=BX-1
Z1:             ;žymė Z1
MUL BX          ;AX=AX*BX
DEC BX          ;BX=BX-1
JNZ Z1          ;jei BX<>0, pereiti prie žymės Z1
RET
fakt ENDP       ;procedūros fakt apibrėžties pabaiga
```

5.1 pratimas. Remdamiesi pateiktais pavyzdžiais, sudarykite procedūrą realizuojančią nurodyto sveikojo skaičiaus kėlimą laipsniu. Laipsnis taip pat – sveikasis skaičius.

5. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Kuri iš pateiktų procedūros apibrėžčių yra teisinga?	PROC1 PROC MOV AX,BX DEC BX ENDP	Atsakymas neteisingas. Procedūros apibrėžties pabaigoje prieš direktyva ENDP trūksta procedūros pavadinimo PROC1.
		PROC PROC MOV AX,BX DEC BX PROC ENDP	Atsakymas neteisingas. Procedūros pavadinimas negali sutapti su direktyva.
		PROC1 PROC MOV AX,BX DEC BX PROC1 ENDP	Atsakymas teisingas.

2.	Kokia bus registro BL reikšmė po šios programos įvykdymo?
	
	MOV CX,5
	CALL p
	RET
	
	p PROC
	MOV BX,CX
	Z1:
	INC BX
	LOOP Z1
	RET
	p ENDP
		0Ah	Atsakymas teisingas.

		05h	Atsakymas neteisingas. Procedūra p registro BL reikšmę vis didins po vieną tol, kol CX reikšmė taps lygi 0. Vadinasi rezultate gausime BL=0Ah.
		00h	Atsakymas neteisingas. Procedūra p registro BL reikšmę vis didins po vieną tol, kol CX reikšmė taps lygi 0. Vadinasi rezultate gausime BL=0Ah.

3.	Kokia bus registro AX reikšmė po šios programos įvykdymo?
	
	MOV CX,5
	XOR AX,AX
	CALL p
	RET
	
	p PROC
	MOV BX,CX
	Z1:
	INC BX
	ADD AX,BX
	LOOP Z1
	RET
	p ENDP
		40h	Atsakymas neteisingas. Prie pradinės registro AX reikšmės 0 bus pridedamos reikšmės 6, 7, 8, 9, 10. Taigi, rezultate gausime 40 arba šešioliktainiu pavidalu 28h.

		28h	Atsakymas teisingas.
		00h	Atsakymas neteisingas. Prie pradinės registro AX reikšmės 0 bus pridedamos reikšmės 6, 7, 8, 9, 10. Taigi, rezultate gausime 40 arba šešioliktainiu pavidalu 28h.

Papildomi skaitiniai

[1] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (212 p. – 218 p.) ISBN 5-272-00040-4
[2] В. И. Пустоваров, Язык Ассемблер в програмировании информационных и управляющих систем. Москва, «ЭНТРОП», Киев, «Век» 1997. – 304 с. ISBN 5-88547-052-9

6. Makro komandos

Makro komandos labai panašios į procedūras, bet tai ne visiškai tas pats. Jos atrodo kaip procedūros, tačiau egzistuoja tik iki tol, kol kodas sukompiliuojamas. Po kompiliavimo visos makro komandos pakeičiamos realiomis instrukcijomis. Jeigu apibrėžėte makro ir niekur jo nepanaudojote savo kode, kompiliatorius paprasčiausiai praignoruos ją. **emu8086.inc <asm_tutorial_05.html>** pateikia pavyzdžių kaip galima panaudoti makro komandas. Makro komandos apibrėžimas:

```
pavadinimas MACRO [parametrai,...]
<instrukcijos>
ENDM
```

Priešingai nei procedūros, makro komandos turi būti apibrėžtos prieš kodą, kuris naudos makro komandą, pavyzdžiui

```
MyMacro MACRO  p1, p2, p3       ;makro komandos apibrėžties pradžia
MOV AX, p1
MOV BX, p2
MOV CX, p3
ENDM                            ;makro komandos apibrėžties pabaiga

ORG 100h
MyMacro 1, 2, 3
MyMacro 4, 5, DX
RET
```

Aukščiau pateiktas kodas yra išplečiamas taip:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

6.1 pavyzdys. Ankstesnę 5.1 pavyzdžio procedūrą, apskaičiuojančią nurodyto skaičiaus faktorialą pertvarkysime į makro komandą.

```
ORG 100h

fakt MACRO sk   ;makro komandos apibrėžties pradžia
```

```
MOV AX,sk       ;AX=sk
MOV BX,AX       ;BX=AX
DEC BX          ;BX=BX-1
Z1:             ;žymė Z1
MUL BX          ;AX=AX*BX
DEC BX          ;BX=BX-1
JNZ Z1          ;jei BX<>0, pereiti prie žymės Z1
ENDM

fakt 5          ;kreiptis į makro siunčiant parametro sk reikšmę 5
RET
```

Apie **makro komandas** ir **procedūras** svarbu atsiminti, kad

- norėdami iškviesti procedūrą turime naudoti instrukciją CALL. Pavyzdžiui CALL MyProc;
- norint iškviesti makro komandą, užtenka tiesiog užrašyti jos pavadinimą. Pavyzdžiui MyMacro;
- procedūros yra talpinamos tam tikru adresu atmintyje. Procesorius perduoda valdymą šiai atminties vietai. Valdymas bus grąžintas programai panaudojus instrukciją **RET**. Stekas naudojamas grįžimo adresui saugoti. Instrukcija CALL užima apie 3 baitus, taigi paleidžiamojo failo dydis auga labai nežymiai, visai nesvarbu kiek kartų programoje kreipiamės į vieną ir tą pačią procedūrą;
- makro komanda tiesiogiai išplečiama į programos kodą. Taigi, jei naudojate tą pačią makro 100 kartų, kompiliatorius išplečia makro į programos kodą 100 kartų, tuo pačiu paversdamas paleidžiamąjį failą vis didesniu ir didesniu. Kiekvieną kartą įterpiamos visos makro komandos instrukcijos;
- norėdami perduoti parametrus procedūrai, turime naudoti steką arba bet kurį bendrosios paskirties registrą;
- parametrų perdavimui į makro komandą užtenka juos surašyti už makro komandos pavadinimo, pavyzdžiui MyMacro 1, 2, 3;
- makro komandos pabaigą žymi direktyva **ENDM**;
- procedūros pabaigą žymi procedūros **pavadinimas** ir už jo einanti direktyva **ENDP**.

Makro komandos išplečiamos tiesiogiai programos kode. Be to, jei naudojame žymes makro komandos apibrėžimo viduje, galime susidurti su klaidos pranešimu "Duplicate declaration" (toks apibrėžimas jau yra), kai makro komanda yra naudojama du ar daugiau kartų. Tokių problemų išvengti padeda direktyvos **LOCAL** naudojimas, pavyzdžiui LOCAL MyMacro p1, p2. Pavyzdžiui;

```
MyMacro2 MACRO
LOCAL label1, label2
CMP AX, 2
JE label1
CMP AX, 3
JE label2
label1:
INC AX
label2:
ADD AX, 2
```

```
ENDM

ORG 100h
MyMacro2
MyMacro2
RET
```

6.1 pratimas. Pertvarkykite 5.1 pratimo skaičiaus kėlimo laipsniu procedūrą į makro komandą.

6. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Kuri iš pateiktų makro komandos apibrėžčių yra teisinga?	Makro MACRO p1 MOV BX,p1 DEC BX ENDM	Atsakymas teisingas.
		Makro MACRO MOV BX,p1 DEC BX Makro ENDM	Atsakymas neteisingas. Prieš direktyvą ENDM makro komandos pavadinimo rašyti nereikia.
		Macro MACRO MOV BX,p1 DEC BX ENDM	Atsakymas neteisingas. Makro komandos pavadinimas negali sutapti direktyva MACRO.

2.	Kokia bus registro AL reikšmė po šios programos įvykdymo?
	
	m MACRO sk
	MOV BX,sk
	DEC BX
	Z1:
	MUL BX
	DEC BX
	JNZ Z1
	ENDM
	
	MOV AX,0
	m 5
	RET
		05h	Atsakymas neteisingas. Kadangi registre AL pradžioje įrašomas 0, tai atlikus daugybos veiksmą 0 jame ir išliks.

		78h	Atsakymas neteisingas. Kadangi registre AL pradžioje įrašomas 0, tai atlikus daugybos veiksmą 0 jame ir išliks.
		00h	Atsakymas teisingas.

3.	Į kiek eilučių bus iškleistas programos kodas jį kompiliuojant jei makro komanda tokia kaip 2 klausime, o programos kodas toks: MOV AX,0 m 5 m 10 RET	8	Atsakymas neteisingas. Eilutė „m 5“ bus išskleista į 6 eilutes ir į dar tiek eilučių bus išskleista “m 10”. Todėl iš viso gausime 6+6+2=14 eilučių.
		14	Atsakymas teisingas.
		4	Atsakymas neteisingas. Eilutė „m 5“ bus išskleista į 6 eilutes ir į dar tiek eilučių bus išskleista “m 10”. Todėl iš viso gausime 6+6+2=14 eilučių.

Papildomi skaitiniai

[1] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (283 p. – 291 p.) ISBN 5-272-00040-4
[2] В. И. Пустоваров, Язык Ассемблер в програмировании информационных и управляющих систем. Москва, «ЭНТРОП», Киев, «Век» 1997. – 304 с. ISBN 5-88547-052-9

7. Stekas

Stekas - tai atminties vieta laikinų duomenų saugojimui. Stekas taip pat naudojamas procedūros **CALL** grįžimo adresams iš procedūrų saugoti. Instrukcija **RET** procedūros grįžimo adresą paima iš steko ir grįžta atitinkamu postūmiu. Tas pats vyksta ir kai instrukcija **INT** iškviečia pertraukimą, ji steko žymių registre išsaugo kodo segmentą ir postūmį. Instrukcija **IRET** naudojama registrų IP, CS bei žymių registro reikšmėms atstatyti.

Steką taip pat galima naudoti ir kitokių duomenų saugojimui. Tada naudojamos dvi instrukcijos:

PUSH - išsaugo 16 bitų reikšmę steke.
POP - paima 16 bitų reikšmę iš steko.

PUSH instrukcijos sintaksė:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate

REG: AX, BX, CX, DX, DI, SI, BP, SP.
SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bitų kintamasis ir pan.
immediate: 5, -24, 3Fh, 10001101b ir pan.

POP instrukcijos sisntaksė:

POP REG
POP SREG
POP memory

REG: AX, BX, CX, DX, DI, SI, BP, SP.
SREG: DS, ES, SS, (except CS).
memory: [BX], [BX+SI+7], 16 bitų kintamasis ir pan.

Pastabos:

- PUSH ir POP dirba tik su 16 bitų reikšmėmis.
- PUSH immediate veikia tik 80186 procesoriuose ir vėlesniuose!

Stekas naudoja **LIFO** (Last In First Out) algoritmą. Tai reiškia jei patalpinsime į steką reikšmes 1, 2, 3, 4, 5 vieną paskui kitą iš eilės, tai pirmoji reikšmė, kurią ištrauksime iš steko bus 5, tada 4, 3, 2 ir tik tada 1. Pavyzdžiui, norėdami registro AX turinį patalpinti į steką ir po to tą reikšmę priskirti registrui CX, galime atlikti tokią instrukcijų seką:

```
MOV AX, 10h       ;AX=10h
PUSH AX
POP CX            ;CX=10h
```

Steko atminties sritis nustatoma registru SS (steko segmentas) ir registru SP (steko rodyklė). Bendru atveju operacijų sistema priskiria šių registrų reikšmes programos vykdymo pradžioje.

Instrukcija "**PUSH šaltinis**" atlieka tokius veiksmus:

- atima **2** iš **SP** registro;
- įrašo šaltinio reikšmę adresu **SS:SP**.

Instrukcija "**POP paskirties vieta**" atlieka tokius veiksmus:

- reikšmę, esančią adrese **SS:SP**, įrašo į paskirties vietą;
- prideda **2** prie **SP** registro.

Einamasis adresas, nusakytas **SS:SP** yra vadinamas **steko viršūne**.

COM failams steko segmentas paprasčiausiai yra kodo segmentas, o steko rodyklei yra priskiriama reikšmė **0FFFEh**. Adresu **SS:0FFFEh** išsaugomas grįžimo adresas instrukcijai **RET**, kuri įvykdoma programos pabaigoje.

Emuliatoriuje galime vizualiai stebėti steko operacijas. Reikia paspausti mygtuką [**Stack**]. Steko viršūnė pažymėta ženklu "**<**".

7.1 pratimas. Emuliatoriuje iš "**Samples**" katalogo atverkite failą "**stack.asm**". pažingsniui vykdykite programą, atkreipkite dėmesį kaip keičiasi bendrosios paskirties registrų turiniai įvykdžius instrukcijas **PUSH** ir **POP**. Mokėkite paaiškinti steko veikimo principą.

7. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Steko veikimo principas pagrįstas algoritmu	FIFO (First In First Out)	Atsakymas neteisingas, kadangi stekas veikia LIFO algoritmu.
		LIFO (Last In First Out)	Atsakymas teisingas.
		LILO (Last In Last Out)	Atsakymas neteisingas, kadangi stekas veikia LIFO algoritmu.

2.	Tegu turime kodo fragmentą:
	
	MOV BX, 5
	MOV CX, 7
	PUSH CX
	PUSH BX
	POP CX
	POP AX
	ADD AX,CX
	MUL BX
	
	Kokia reikšmė bus registre AX po šio kodo fragmento įvykdymo?
		24	Atsakymas neteisingas. Įvykdžius instrukciją POP CX, CX = 5; įvykdžius POP AX, AX = 7. Sudėjus gauname AX = 12, padauginę AX = 60.

		12	Atsakymas neteisingas. Įvykdžius instrukciją POP CX, CX = 5; įvykdžius POP AX, AX = 7. Sudėjus gauname AX = 12, padauginę AX = 60.
		60	Atsakymas teisingas.
3.	Kuri iš pateiktų instrukcijų yra neteisinga?	PUSH 15h	Atsakymas neteisingas. Ši instrukcija korektiška, nes į steką galime įrašyti ir tiesioginę reikšmę.

		PUSH AX	Atsakymas neteisingas. Ši instrukcija yra korektiška, kadangi į steką galime įrašyti registro turinio reikšmę.

Papildomi skaitiniai

[1] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (283 p. – 291 p.) ISBN 5-272-00040-4
[2] В. И. Пустоваров, Язык Ассемблер в програмировании информационных и управляющих систем. Москва, «ЭНТРОП», Киев, «Век» 1997. – 304 с. ISBN 5-88547-052-9

8. Loginės instrukcijos

Greta aritmetinių komandų mikroprocesoriaus komandų sistema turi eilę komandų, kurios skirtos loginiam duomenų apdorojimui. Tokių instrukcijų veikimo pagrindas yra formaliosios (matematinės) logikos taisyklės. Ši logika operuoja dvejomis reikšmėmis - tiesa ir melas, kurios mikroprocesoriui suprantamos kaip 1 ir 0 atitinkamai. Kompiuteriui nulių ir vienetų kalba puikiai suprantama, tačiau minimalus duomenų vienetas, su kuriuo dirba mašina, yra baitas. Sisteminiame lygyje dažnai būtina turėti galimybę dirbti ir pakankamai žemame lygyje, t.y. bitų lygyje.

Kaip jau minėta, loginis duomenų apdirbimas remiasi matematinės logikos taisyklėmis. Kaip žinome iš matematinės logikos kurso yra keletas logikos sistemų. Viena iš jų – teiginių logika. Teiginių logikos skaičiavimai puikiai dera su kompiuterio darbo principais ir pagrindiniais jo programavimo metodais. Visi aparatūriniai kompiuterio komponentai sudaryti loginių mikroschemų pagrindu. Informacijos atvaizdavimo kompiuteryje sistema pačiame žemiausiame lygmenyje paremta bito samprata. Bitas, turėdamas tik dvi galimas būsenas 0 (melas) ir 1 (tiesa) puikiai įsikomponuoja į teiginių skaičiavimo teoriją.

Remiantis teiginių logikos skaičiavimų (propozicinių skaičiavimų) teorija, su bitais galime atlikti tokias logines operacijas:

AND operandas1, operandas2 – loginės daugybos operacija. Ši komanda atlieka konjunkcijos operaciją su kiekvienu abiejų operandų bitu. Rezultatas įrašomas į pirmąjį operandą.

OR operandas1, operandas2 – loginės sudėties operacija. Ši komanda atlieka disjunkcijos operaciją su kiekvienu abiejų operandų bitu. Rezultatas įrašomas į pirmąjį operandą.

XOR operandas1, operandas2 – griežtosios disjunkcijos operacija. Su kiekvienu abiejų operandų bitu atliekama griežtosios disjunkcijos operacija. Rezultatas įrašomas pirmajame operande.

TEST operandas1, operandas2 – patikrinimo operacija. Pabičiui atliekama konjunkcijos operacija su abiejų operandų bitais. Operandų būsena išlieka nepakitusi, keičiasi tik žymės ZF, SF, ir PF. Tai sudaro galimybę analizuoti atskirų operando bitų būseną, nepakeičiant pačių operandų būsenos.

NOT operandas – loginio neigimo operacija. Ši komanda pabičiui invertuoja operando bitų reikšmes. Rezultatas išsaugomas pačiame operande.

Svarbu suprasti loginių komandų taikymo sritį programuojant. Pasinaudodami loginėmis komandomis galime išskirti atskirus operando bitus siekdami nustatyti, invertuoti ar paprasčiausiai patikrinti reikiamo bito reikšmę. Tam tikslui antrasis operandas paprastai atlieka kaukės vaidmenį. Pasinaudojant kakės bitais, kuriems priskirta 1 reikšmė, ir nustatomi konkrečiai operacijai reikalingi pirmojo operando bitai. Pavyzdžiui,

- Norėdami nustatyti atitinkamų pirmojo operando bitų reikšmę į 1 naudosime komandą **OR operandas1, operandas2**. Šioje komandoje operandas2 yra kaukė, kurioje vieneto reikšmes turi tie bitai, kurie pirmajame operande turi įgyti reikšmę 1.

```
OR AX, 01b          ;nustatyti AX registro nulinio bito reikšmę į 1
```

- Norėdami nustatyti atitinkamo bito reikšmę į 0, naudosime komandą **AND operandas1, operandas2**. Šioje komandoje operandas2 yra kaukė, kurioje nulio reikšmes turi tie bitai, kuriuos norime nustatyti į 0 pirmajame operande.

```
AND AX, 0FFFDh    ;nustatyti į 0 pirmąjį bitą registre AX
```

- Komanda **XOR operandas1, operandas2** naudojame, kai
  - norime išsiaiškinti, kurie bitai pirmajame ir antrajame operande skiriasi savo reikšmėmis;
  - norime invertuoti pirmojo operando nurodytų bitų reikšmes.

Mus dominantys kaukės bitai komandos vykdymo metu turi būti lygūs 1, visi likę lygūs 0.

```
XOR AX, 10b         ;invertuoti pirmąjį registro AX bitą
```

- Komanda **TEST operandas1, operandas2** (tikriname pirmąjį operandą) naudojama nurodytų bitų būsenai patikrinti. Tikrinamieji pirmojo operando bitai kaukėje (antrasis operandas) turi turėti vienetines reikšmes. TEST komandos algoritmas panašus kaip ir AND komandos, tik šiuo atveju nėra pakeičiamos pirmojo operando bitų reikšmės. Komandos rezultatas atsispindi žymėje ZF.

Jeigu ZF lygi 0, tai atlikus loginę daugybą, buvo gautas nulinis rezultatas. T. y. vienas vienetinis kaukės bitas nesutapo su atitinkamu pirmojo operando bitu.

Jeigu ZF lygi 1, tai atlikus loginę daugybą buvo gautas nenulinis rezultatas. T. y bent vienas vienetinis kaukės bitas sutapo su atitinkamu vienetiniu pirmojo operando bitu.

```
TEST AX, 0010h
JZ M1                ;pereiti prie žymės M1, jei ketvirtasis bitas lygus 1
```

Kaip matome iš pavyzdžio kartu su TEST instrukcija prasminga naudoti peršokimo komandas **JNZ** (Jump if Not Zero) – pereiti, jei nulio žymė ZF lygi 1, arba atvirkštinę komandą **JZ** (Jump if Zero) - pereiti, jei nulio žymė ZF lygi 0.

Prie loginių instrukcijų taip pat priskiriamos ir postūmio komandos. Kai kurias iš jų ir aptarsime.

SHL operandas, postūmių_skaitliukas - loginis postūmis į kairę (Shift Logical Left). Operando turinys pastumiamas į kairę per postūmių skaitliuke nurodytą bitų skaičių. Iš dešinės (pradedant nuo jauniausiojo bito reikšmės) įrašomi nuliai.

SHR operandas, postūmių_skaitliukas - loginis postūmis į dešinę (Shift Logical Right). Operando turinys pastumiamas į dešinę per postūmių skaitliuke nurodytą bitų skaičių. Iš kairės (pradedant nuo vyriausiojo bito) įrašomi nuliai.

Ciklinio postūmio komandoms priklauso tos, kurios išsaugo pastumiamų bitų reikšmes. Kai kurias iš jų paminėsime.

ROL operandas, postūmių_skaitliukas – ciklinis postūmis į kairę (Rotate Left). Operando turinys pastumiamas per tiek bitų į kairę, kiek yra nurodyta postūmių_skaitliuke. Pastumiami į kairę bitai įrašomi į tą patį operandą iš dešinės.

ROR operandas, postūmių_skaitliukas – ciklinis postūmis į dešinę (Rotate Right). Operando turinys pastumiamas per tiek bitų į dešinę, kiek yra nurodyta postūmių_skaitliuke. Pastumiami į dešinę bitai įrašomi į tą patį operandą iš kairės.

Pavyzdžiui norint pakeisti dviejų registro AX pusregistrių reikšmes pakanka įvykdyti tokią instrukcijų seką:

```
MOV AX, FF00h
MOV CL, 8
ROL AX, CL
```

8.1 pratimas. Žemiau pateiktą programos fragmentą įvykdykite emuliatoriuje emu8086. Tai šviesoforo programos fragmentas. Pakeiskite jį taip, kad pradinėje būsenoje dega visos žalios šviesoforo lempos, o tada būsena keičiama cikliškai pastumiant reikšmes per tris bitus į kairę.

```
MOV AX, 1
next_situation:
OUT 4, AX
ROL AX, 1    ; pastumiame AX reikšmę per vieną bitą
JMP next_situation
```

8. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Norime registro BX trečiojo, ketvirtojo ir septintojo bitų reikšmes nustatyti į 1. Kurią instrukciją turime įvykdyti?	AND BX, 10011000b	Atsakymas neteisingas. Įvykdę šią instrukciją BX registro 0-2, 5, 6 bitus nustatysime į 0.
		OR BX, 98h	Atsakymas teisingas.
		OR AX, 10011000b	Atsakymas neteisingas. Įvykdę šią instrukciją atitinkamai nustatysime registro AX bitus.
2.	Turime tokį kodo fragmentą: TEST BX, 80h JNZ Z1 Kuriuo atveju bus pereinama prie žymės Z1?	Jei registro BX septintasis bitas bus lygus 0, tada bus pereinama prie žymės Z1.	Atsakymas teisingas.
		Jei registro BX septintasis bitas bus lygus 1, tada bus pereinama prie žymės Z1.	Atsakymas neteisingas. Šiuo atveju, turėtume panaudoti instrukciją JZ Z1.
		Jei registro BX aštuntasis bitas bus lygus 0, tada bus pereinama prie žymės Z1.	Atsakymas neteisingas. Bito, į kurį kreipiamas dėmesys numeris yra septintas.
3.	Registre AL įrašyta reikšmė 11000000b. Kokią instrukciją turime panaudoti norėdami cikliškai per dvi pozicijas į dešinę pastumti bitų reikšmes?	MOV CL, 10b ROL AL, CL	Atsakymas neteisingas. Panaudodami šias instrukcijas bitų reikšmes cikliškai pastumsime per dvi pozicijas į kairę.
		MOV CL, 02h SHR AL, CL	Atsakymas neteisingas, nes instrukcija SHR atlieka loginį postūmį į dešinę.
		MOV CL, 2 ROR AL, CL	Atsakymas teisingas.

Papildomi skaitiniai

[1] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (184. – 201) ISBN 5-272-00040-4

[2] В. И. Пустоваров, Язык Ассемблер в програмировании информационных и управляющих систем. Москва, «ЭНТРОП», Киев, «Век» 1997. – 304 с. ISBN 5-88547-052-9

9. Programos eigos valdymas

Komandos, su kuriomis susidūrėme iki šiol, atliko tam tikrus veiksmus su duomenimis arba juos perduodavo tam tikriems operandams. Po tokios komandos įvykdymo procesorius perduoda valdymą sekančiai komandai. Tačiau jau bent kiek sudėtingesnėje programoje tokio nuoseklaus komandų vykdymo nebepakanka. Paprastai programoje yra tam tikros vietos, kuriose privalu apsispręsti kokia komanda bus vykdoma sekančiame žingsnyje. Šis sprendimas gali būti

- besąlyginis – einamojoje programos vietoje valdymą reikia perduoti komandai, kuri nėra nuosekliai sekanti komanda, o yra aprašyta kitoje programos kodo vietoje;
- sąlyginis – sprendimas, apie tai kuri komanda bus vykdoma sekančiame žingsnyje, priimamas remiantis tam tikrų sąlygų ar duomenų analizės rezultatais.

Kaip žinome, programa yra komandų ir duomenų seka, užimanti tam tikrą operatyviosios atminties erdvę. Ši erdvė gali būti vientisa arba susidėti iš keletos fragmentų (programos kodo ir jos duomenų skaidymas į segmentus). Kokia programos komanda turi būti vykdoma sekančiame žingsnyje, mikroprocesorius sužino pagal registrų poros CS:IP turinį:

- CS – kodo segmentas. Jame patalpintas fizinis (bazinis) einamojo kodo segmento adresas;
- IP – instrukcijų rodyklė. Jame patalpinta reikšmė, reiškianti postūmį operatyviojoje atmintyje nuo segmento pradžios (adresą), kuriuo išsaugota tam tikra komanda.

Tokiu būdu programos eigos valdymo komandos keičia registrų CS ir IP turinius ir dėka to mikroprocesorius vykdymui išrenka ne nuosekliai sekančią programos kodo komandą, o komandą tam tikroje tolimesnėje programos vietoje.

Pagal veikimo principą programos eigos valdymo komandas galime suskirstyti į grupes:

- besąlyginės valdymo perdavimo komandos
  - besąlyginio peršokimo komanda JMP;
  - procedūros iškvietimo ir grįžimo iš jos komandos CALL ir RET;
  - programinio pertraukimo iškvietimo ir grįžimo iš jo komandos INT ir IRET;

- sąlyginės valdymo perdavimo komandos
  - peršokimo pagal palyginimo instrukcijos CMP rezultatą komandos;
  - peršokimo, atsižvelgiant į tam tikros žymių registro žymės būseną, komandos;
  - peršokimo, atsižvelgiant į registro CX turinį, komandos;

- ciklo valdymo komandos
  - ciklo sudarymo naudojant skaitliuką CX komanda;

- ciklo sudarymo naudojant skaitliuką CX su galimybe išeiti iš ciklo atsižvelgiant į papildomą sąlygą.

9.1 Besąlyginės valdymo perdavimo komandos

Besąlyginio peršokimo komanda JMP. Besąlyginio peršokimo komandos be informacijos į grįžimo vietą išsaugojimo sintaksė yra

JMP [modifikatorius] peršokimo_adresas

Peršokimo_adresas – tai adresas žymės pavidalu arba atminties srities adresas, kuriam reikia perduoti valdymą; modifikatorius gali įgyti reikšmes **near ptr**, **far ptr**, **word ptr**, **dword ptr**, tačiau jį nurodyti nebūtina, jei peršokimas vyksta to paties kodo segmento ribose.

Procedūros. Kviesdami procedūrą peršokame į tolimesnę programos kodo vietą. Šitam peršokimui įvykdyti nereikia jokių sąlygų, pakanka užrašyti instrukciją CALL, o grįžimui iš procedūros jos pabaigoje nurodyti instrukciją RET. Todėl procedūros taip pat priskiriamos prie besąlyginių programos eigos valdymo komandų. Apie procedūrų sudarymo taisykles kalbėjome 5 skyriuje.

Programiniai pertraukimai. Programiniams pertraukimams iškviesti kaip ir procedūroms naudojamės specialia instrukcija INT. Pertraukimui iškviesti taip pat nereikia papildomų sąlygų tenkinimo, todėl instrukcijos INT ir IRET yra besąlyginio programos valdymo perdavimo komandos. Plačiau apie pertraukimus kalbėjome 4 skyriuje.

9.1 pavyzdys. Naudodami besąlyginio peršokimo komandą **JMP** sudarysime ciklą, kuris sumuoja masyvo elementus.

```
MOV SI, 0              ;registrą SI naudojame masyvo indeksui nurodyti
JMP start:             ;peršokame į žymę start

ciklas:                ;žymė ciklas
ADD AX, lentele[SI]    ;prie AX turinio pridedame einamąjį masyvo
                       ;elementą
ADD SI, 2              ; didiname registro SI turinį

start:                 ;žymė start
JMP ciklas             ;peršokame prie žymės ciklas

lentele DW 1, 2, 3, 4, 5  ;masyvo apibrėžtis
```

Naudodami besąlyginio peršokimo komandą **JMP** galime sudaryti taip vadinamus “amžinus” ciklus, kaip kad yra ciklas „**JMP** ciklas“. Taip yra todėl, kad ši komanda netikrina jokių sąlygų, kurios galėtų nutraukti ciklo vykdymą. Kiekvieną kartą, kai valdymas perduodamas šiai instrukcijai, ji peršoka į tą atminties vietą, kurioje apibrėžta programos dalis “ciklas” ir programa kartojama iš naujo.

Šiame pavyzdyje reikėtų, kad sumavimo ciklas baigtųsi ties ketvirtuoju (skaičiuojant nuo nulio) masyvo elementu. Tačiau naudodami instrukciją

JMP nurodyti tokios sąlygos paprasčiausiai neturime galimybės.

9.2 Sąlyginės valdymo perdavimo komandos

Mikroprocesorius turi 18 sąlyginio peršokimo komandų. Šios komandos leidžia patikrinti

- santykį tarp operandų su ženklu (daugiau, mažiau);
- santykį tarp operandų be ženklo (aukščiau, žemiau);
- aritmetinių žymių registro žymių ZF, SF, CF, OF, PF būseną (bet ne AF).

Sąlyginio peršokimo komandos turi vienodą sintaksę

JCC žymė

Visų šių komandų mnemoninis kodas prasideda raide J (jump), o CC apibrėžia konkrečią sąlygą, kurią analizuoja komanda. Operandas *žymė* būtinai turi būti apibrėžtas einamajame kodo segmente.

Sprendimas, į kurią programos vietą bus perduotas valdymas sąlyginio peršokimo komanda, priimamas remiantis suformuota sąlyga, kurios analizės rezultatai ir įtakoja priimamą sprendimą. Minėtai sąlygai suformuoti galima pasinaudoti

- bet kuria komanda, kuri pakeičia aritmetinių žymių registro žymių būseną;
- palyginimo komanda CMP, lyginančia dviejų operandų reikšmes;
- registro CX turiniu.

Palyginimo komanda CMP. Ši komanda kaip ir atimties komanda SUB atlieka dviejų operandų atimtį, t. y. iš pirmojo operando atima antrąjį operandą ir atitinkamai nustato žymių registro žymes. Tačiau skirtingai nuo instrukcijos SUB, ji neišsaugo rezultato vietoje pirmojo operando.

Komandos CMP (compare) sintaksė tokia:

CMP operandas1, operandas2

1 lentelė. Sąlyginių peršokimo komandų mnemoninės žymės ir jų prasmė

Mnemoninė žymė	Prasmė	Operandų tipas
E	Equal (lygu)	Bet kokie
N	Not (ne)	Bet kokie
G	Greater (daugiau)	Skaičiai su ženklu
L	Less (mažiau)	Skaičiai su ženklu
A	Above (aukščiau, “daugiau” prasme)	Skaičiai be ženklo
B	Below (žemiau, “mažiau prasme”)	Skaičiai be ženklo

Kaip jau buvo minėta, ši komanda palygina du operandus ir pagal palyginimo rezultatus nustato atitinkamas žymių registro žymes. Žymes, kurias nustato instrukcija CMP, galima analizuoti specialiomis sąlyginio peršokimo komandomis. Atkreipkime dėmesį į šių komandų mnemoninius kodus (mnemonikas), surašytus 1 lentelėje. 1 lentelės stulpelyje “mnemoninė žymė” pateikiamos tik sąlyginės

peršokimo instrukcijos J**CC** paskutinės reidės, kurias apibrėždami pažymėjome kaip CC.

2 lentelė. Specialios sąlyginio peršokimo komandos

Operandų tipai	**Mnemonika**	**Sąlyginio peršokimo kriterijus**	**Žymių registro žymių reišmės, kad įvyktų peršokimas**
Bet kokie	JE	operandas1 = opernadas2	ZF = 1
Bet kokie	JNE	operandas1 <> operandas2	ZF = 0
Su ženklu	JL/JNGE	operandas1 < opernadas2	SF <> OF
Su ženklu	JLE/JNG	operandas1 <= operandas2	SF <> OF or Zf = 1
Su ženklu	JG/JNLE	operandas1 > operandas2	SF = OF and ZF = 0
Su ženklu	JGE/JNL	operandas1 => operandas2	SF = OF
Be ženklo	JB/JNAE	operandas1 < operandas2	CF = 1
Be ženklo	JBE/JNA	operandas1 <= operandas2	CF = 1 or ZF=1
Be ženklo	JA/JNBE	operandas1 > operandas2	CF = 0 and ZF = 0
Be ženklo	JAE/JNB	operandas1 => operandas2	CF = 0

9.2 pavyzdys. Naudodami sąlyginio peršokimo komanda JE modifikuosime 9.1 pavyzdžio ciklą, eliminuodami ankstesniame pavyzdyje paminėtus instrukcijos JMP trūkumus.

```
MOV SI, 0               ;registrą SI naudojame masyvo indeksui nurodyti
JMP start:              ;peršokame į žymę start

ciklas:                 ;žymė ciklas
ADD AX, lentele[SI]     ;prie AX turinio pridedame einamąjį masyvo
                        ;elementą
ADD SI, 2               ;didiname registro SI turinį
CMP SI, 10              ;tikriname ar jau susumavome visus masyvo
                        ;elementus
JE stop                 ;jei susumuoti visi elementai, tada peršokame į
                        ;žymę stop

start:                  ;žymė start
JMP ciklas              ;peršokame prie žymės ciklas

stop:
MOV AH, 4Ch             ;sustabdome programą ir grąžiname valdymą
INT 21h                 ;operacijų sistemai

lentele DW 1, 2, 3, 4, 5  ;masyvo apibrėžtis
```

Sąlyginio peršokimo komandos ir žymių registro žymės. Kai kurių sąlyginio peršokimo komandų mnemonika atspindi žymių registro žymių pavadinimus ir turi struktūrą: pirmas eina simbolis J (jump, peršokti), antras – arba žymės pavadinimo simbolis, arba neigimo simbolis N, po kurio eina žymės pavadinimo simbolis. Tokia komandos struktūra atspindi jos paskirtį.

Jei simbolio N nėra, tai tikrinama žymių registro žymės būsena ir jei ji lygi 1, tada vykdomas peršokimas į peršokimo žymę.

Jei mnemonikoje yra simbolis N, tada tikrinama žymių registro žymės būsena ir jei ji lygi 0, tada vykdomas peršokimas į peršokimo žymę.

Šios sąlyginio peršokimo komandos pateiktos 3 lentelėje, jas galima naudoti po bet kurios komandos, pakeičiančios tam tikras žymių registro žymių reikšmes.

3 lentelė. Sąlyginio peršokimo komandos ir žymių registro žymės

Žymės pavadinimas	**Bito numeris žymių registre**	**Sąlyginio peršokimo komanda**	**Žymės reikšmė, kad peršokimas įvyktų**
Pernešimo žymė CF	1	JC	CF = 1
Lygiškumo žymė PF	2	JP	PF = 1
Nulio žymė ZF	6	JZ	ZF = 1
Ženklo žymė SF	7	JS	SF = 1
Perpildymo žymė OF	11	JO	OF = 1
Pernešimo žymė CF	1	JNC	CF = 0
Lygiškumo žymė PF	2	JNP	PF = 0
Nulio žymė ZF	6	JNZ	ZF = 0
Ženklo žymė SF	7	JNS	SF = 0
Perpildymo žymė OF	11	JNO	OF = 0

Reikia atkreipti dėmesį į tai, kad dauguma komandų pateiktų 2 ir 3 lentelėse yra ekvivalentinės, nes analizuodamos rezultatą jos kreipia dėmesį į tas pačias žymes.

9.1 pratimas. Kurią iš 3 lentelėje įvardintų instrukcijų galime panaudoti vietoje instrukcijos JE, kad nepakistų 9.2 pavyzdžio programa?

Sąlyginio peršokimo komandos ir registras CX. Mikroprocesoriaus architektūra numato daugelio registrų specifinį panaudojimą. Pavyzdžiui registras AX naudojamas kaip akumuliatorius, o registrai BP ir SP darbui su steku. Registras CX taip pat turi savo specialią funkcinę paskirtį. Jis naudojamas kaip registras skaitliukas, pavyzdžiui ciklų valdymo komandose.

Šios sąlyginio peršokimo komandos sintaksė tokia:

```
JCXZ peršokimo_žymė          ;peršokti, jei CX nulis
```

9.3 Ciklų sudarymas

Ciklas – svarbi algoritminė struktūra, be kurios neapsieina nei viena rimtesnė programa. Ciklą sudaryti galime naudodami sąlygines arba besąlyginę peršokimo komandas. Konstruojant ciklą tokiu būdu visos jo kūrimo operacijos atliekamos “rankiniu būdu”, t. y programuotojas rūpinasi skaitliuko reikšmės didinimu ar mažinimu ir tikrinimu kada išeiti iš ciklo. Tačiau mikroprocesoriaus komandų sistemoje yra trijų komandų grupė, palengvinanti ciklų programavimą. Šios komandos naudoja registrą CX, kuris atlieka skaitliuko vaidmenį. Apžvelgsime šias instrukcijas.

LOOP peršokimo_žymė ;kartoti ciklą

Ši komanda leidžia sudaryti FOR tipo aukšto lygio programavimo kalbose ciklus, su automatiniu ciklo skaitliuko mažinimu. Komandos veikimas susideda iš tokių veiksmų:

- Mažinamas registro CX turinys vienetu;
- Registro CX turinys lyginamas su 0;
  - Jei CX > 0, tai peršokama į nurodytą žymę;
  - Jei CX = 0, tai vykdoma sekanti po instrukcijos LOOP programoje einanti instrukcija.

9.3 pavyzdys. Naudodami instrukciją **LOOP** modifikuosime anktesnių šio skyriaus pavyzdžių užduoties sprendimo variantą.

```
MOV CX, 5              ;skaitliukas lygus 5, nes penki elementai masyve
MOV SI, 0              ;registrą SI naudojame masyvo indeksui nurodyti
JMP start:             ;peršokame į žymę start

ciklas:                ;žymė ciklas
ADD AX, lentele[SI]    ;prie AX turinio pridedame einamąjį masyvo
                       ;elementą
ADD SI, 2              ;didiname registro SI turinį
LOOP ciklas            ;tikriname ar jau susumavome visus masyvo
                       ;elementus ir skaitliuko turinį mažiname vienetu
JMP stop               ;jei susumuoti visi elementai, tada peršokame į
                       ;žymę stop

start:                 ;žymė start
JMP ciklas             ;peršokame prie žymės ciklas

stop:
MOV AH, 4Ch            ;sustabdome programą ir grąžiname valdymą
INT 21h                ;operacijų sistemai

lentele DW 1, 2, 3, 4, 5  ;masyvo apibrėžtis
```

LOOPE/LOOPZ peršokimo_žymė ;kartoti ciklą tol kol CX <> 0 arba ZF = 0

Komandos LOOPE ir LOOPZ yra ekvivalenčios, todėl nėra jokio skirtumo kurią iš jų naudosime. Šios komandos atlieka tokius veiksmus:

- Mažinamas registro CX turinys vienetu;
- Registro CX turinys lyginamas su 0;
- Analizuojama nulio žymės ZF būsena;
  - Jei CX > 0 ir ZF = 1, tai peršokama į nurodytą žymę;
  - Jei CX = 0 arba ZF = 0, tai vykdoma sekanti po instrukcijos LOOPE/LOOPZ programoje einanti instrukcija.

LOOPNE/LOOPNZ peršokimo_žymė ;kartoti ciklą tol kol CX <> 0 arba ZF = 1

Komandos LOOPNE ir LOOPNZ taip pat ekvivalenčios. Jos veikia tokiu principu:

- Mažinamas registro CX turinys vienetu;
- Registro CX turinys lyginamas su 0;
- Analizuojama nulio žymės ZF būsena;
  - Jei CX > 0 ir ZF = 1, tai peršokama į nurodytą žymę;
  - Jei CX = 0 arba ZF = 1, tai vykdoma sekanti po instrukcijos LOOPE/LOOPZ programoje einanti instrukcija.

Komandos LOOPE/LOOPZ ir LOOPNE/LOOPNZ savo veikimo principu yra atvirkštinės viena kitai. Jos išplečia komandos LOOP galimybes tuo, kad papildomai analizuoja žymę ZF, o tai (jei reikia) leidžia prieš laiką išeiti iš ciklo, pasinaudojant šia žyme kaip indikatoriumi. Šios komandos leidžia atlikti peršokimus tik to paties kodo segmento ribose.

9.2 pratimas. Naudodami sąlyginio peršokimo komandas, o po to instrukcijas ciklams realizuoti parašykite programą kuri dvejetą kelia dešimtuoju laipsniu.

9. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Kurie iš žemiau išvardintų elementų yra besąlyginės valdymo perdavimo komandos?	Programiniai pertraukimai	Atsakymas teisingas tik iš dalies, nes prie besąlyginių valdymo perdavimo komandų grupės dar priskiriamos JMP ir procedūros.
		Instrukcija JMP	Atsakymas teisingas tik iš dalies, nes prie besąlyginių valdymo perdavimo komandų grupės dar priskiriami programiniai pertraukimai ir procedūros.

		Procedūros	Atsakymas teisingas tik iš dalies, nes prie besąlyginių valdymo perdavimo komandų grupės dar priskiriami programiniai pertraukimai ir instrukcija JMP.
		Visos išvardintos	Atsakymas teisingas.

2.	Turime kodo fragmentą:
	
	MOV BX, 3
	MOV AX, 0
	
	ciklas:
	ADD AX, 1
	DEC BX
	CMP BX, 0
	JNE ciklas
	
	Kiek kartų bus pakartotas grįžimas prie žymės ciklas?
		4	Atsakymas neteisingas, kadangi prie žymės ciklas bus grįžtama du kartus.

		3	Atsakymas neteisingas, kadangi prie žymės ciklas bus grįžtama du kartus.
		2	Atsakymas teisingas.

3.	Turime kodo fragmentą:
	
	MOV CX, 5
	MOV AX, 1
	MOV BX, 2
	
	ciklas:
	MUL BX
	LOOP ciklas
	
	MOV AH, 4Ch
	INT 21h
	
	Kam bus lygi registro AX reikšmė po šio kodo fragmento įvykdymo?
		32	Atsakymas teisingas.

		16	Atsakymas neteisingas. Šis kodo fragmentas 2 pakels penktuoju laipsniu.
		64	Atsakymas neteisingas. Šis kodo fragmentas 2 pakels penktuoju laipsniu.

Papildomi skaitiniai

[1] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. (202 p. – 228 p.) ISBN 5-272-00040-4

10. Darbas su disku fiziniame lygmenyje

Rašant tam tikras programas gali prireikti kreipties ne į failus, bet į patį diską kaip įrenginį, t. y. dirbti su disku fiziniame lygmenyje. Dirbant su disku fiziniame lygmenyje būtina aiškiai įsivaizduoti įrenginio struktūrą ir naudojimo ypatybes. Darbas fiziniame lygmenyje reikalauja itin aukšto programuotojo dėmesingumo, nes bet kokia klaidinga kreiptis gali negrįžtamai sugadinti disko failinę struktūrą.

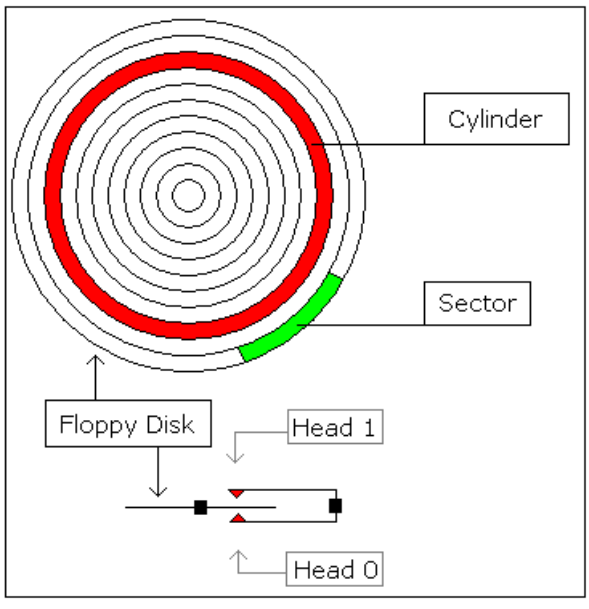
10.1 DOS tipo diskų struktūra

Diskinių kaupiklių veikimo principas. Bet kuriame diskiniame informacijos kaupiklyje naudojamas laisvasis kreipties į diską ir skaitymo/rašymo galvučių judėjimo principas. Diskas iš abiejų pusių yra padengtas specialiu feromagnetiniu paviršiumi, diską griežtai pastoviu greičiu suka specialus varikliukas. Kiekvienoje disko pusėje prie feromagnetinio sluoksnio tampriai priglūsta po vieną skaitymo/rašymo galvutę (10.1 pav.).

Galvutės gali judėti disko paviršiumi spindulio kryptimi ir būti užfiksuotos bet kurioje naujoje padėtyje. Sąveika tarp skaitymo/rašymo galvučių ir disko paviršiaus vyksta tik tuomet, kai galvutė užfiksuojama naujoje padėtyje. Todėl kiekviena nauja galvutės padėtis ant besisukančio disko paviršiaus palieka nematomą koncentrinio apskritimo pėdsaką. Šie koncentriniai apskritimai vadinami **takeliais** (tracks), o įrenginiuose, kurie turi kelis besisukančius diskus, **cilindrais** (cylinders). Kiekvienas takelis/cilindras dalijamas į vienodą skaičių **sektorių**, kurių ribose ir atliekamas informacijos skaitymas/rašymas. DOS pagrindu dirbančios diskinės operacinės sistemos į bet kurį sektorių įrašo ne daugiau kaip **512 baitų** informacijos. Tokiu būdu, bet kurio DOS tipo disko talpa **V** apskaičiuojama pagal formulę

$$\mathbf{V = H \cdot C \cdot S \cdot 512,}$$

čia **H** – galvučių (heads) skaičius; **C** – cilindrų (cylinder) skaičius; **S** – sektorių (sectors) skaičius; 512 yra maksimalus informacijos kiekis baitais sektoriuje.



10.1 pav. Idealizuota diskinio kaupiklio struktūra

Pavyzdžiui, 1440 kbaitų disketė (10.1 pav.)

- yra dvipusė, vadinasi yra dvi galvutės (0..1), kurios juda kiekvienos pusės paviršiumi;
- kiekvienoje disketės pusėje yra 80 cilindrų (numeruojami 0..79);
- kiekvienas cilindras turi 18 sektorių (1..18)
- kiekvienas sektorius talpina 512 baitų informacijos.

Tokiu būdų, galime patikrinti, kad 1440 kbaitų talpos disketės talpa yra

V = 2 · 80 · 18 · 512 = 1.474.560 baitų.

10.1 pratimas. Tarkime, kad diske yra 4 galvutės, 782 cilindrai, kiekviename cilindre yra po 27 sektorius. Kokia šio disko talpa Mbaitais?

Konstrukciniu požiūriu skiriami lankstūs ir kietieji diskiniai kaupikliai. Tai priklauso nuo medžiagos, iš kurios pagamintas pats diskas. Abiejų įrenginių veikimo principas vienodas, tačiau kietasis diskas pasižymi didesne talpa ir trumpesniu kreipties laiku (laiku, reikalingu surasti reikiamą sektorių ir apsikeisti informacija) nei lankstusis diskas. Kietojo disko didesnė talpa pasiekiama spartinant jo sukimosi greitį, tai leidžia duomenis jame įrašyti žymiai didesniu tankiu. Taip pat kietasis diskas turi daugiau cilindrų ir sektorių. Taip pat tokio tipo diskuose naudojami keli diskai, kurie tvirtinami ant vienos ašies. Su kiekvienu tokiu disku sąveikauja sava pora galvučių (po vieną galvutę kiekvienam paviršiui). Visos galvutės tvirtinamos ant vieno laikiklio ir disko paviršiumi juda kartu. Takelių ant visų diskų vienai galvutės padėčiai visuma vadinama **cilindru**.

Fizinės disko sandaros požiūriu visi sektoriai yra vienodi, jų talpa DOS tipo diskuose visada yra 512 baitų. Tačiau operacinės sistemos programos tam tikruose sektoriuose įrašo tam tikras apibrėžtas funkcijas, kurios sudaro tam tikrą disko loginę struktūrą.

Loginė diskų struktūra. Diskas turi keturias logines dalis: paleidžiamąjį sektorių, failų išdėstymo lentelę, šakninį katalogą ir archyvinę erdvę.

Paleidžiamasis sektorius - tai pats pirmasis sektorius nuliniame takelyje (cilindre). Šiame sektoriuje talpinama svarbi disko loginės struktūros informacija ir trumpa pradinio paleidimo programa. Paleidžiamojo sektoriaus turinys automatiškai nuskaitomas į operatyviąją atmintį po to kai įjungiame kompiuterį, tada valdymas perduodamas pradinio paleidimo programai. Ši programa nuskaito į atmintį reikalingas operacijų sistemos dalis ir joms perduoda valdymą. Paleidžiamasis sektorius visų pirma reikalingas pradiniam operacijų sistemos paleidimui.

Pradedant nuo paleidžiamojo sektoriaus 12 baito, jame talpinamas BIOS parametrų blokas (BPB), kuriame pateikiama informacija apie likusių loginių disko dalių dydį ir padėtį, o taip pat kai kurie kiti svarbūs parametrai. BPB bloke pateikiama informacija atitinka tokią duomenų struktūrą:

```
SectSiz     DW ?   ;baitų skaičius sektoriuje, DOS tipui visada 512 baitų
```

```
ClustSiz    DB ?    ;sektorių skaičius klasteryje
ResSecs     DW ?    ;sektorių skaičius prieš FAT
FatCnt      DB ?    ;FAT kopijų skaičius
RootSiz     DW ?    ;šakninio katalogo elementų skaičius
TotSecs     DW ?    ;sektorių skaičius diske
Media       DB ?    ;kaupiklio deskriptorius
FatSize     DW ?    ;sektorių skaičius FAT
```

Klasteris – tai minimali diskinio kaupiklio erdvės sritis, kurią operacinė sistema gali skirti failui.

Iš karto po BPB, pradedant 25 baitu, paleidžiamajame sektoriuje patalpinta tokia duomenų struktūra:

```
TrkSecs     DW ?    ;sektorių skaičius takelyje
HeadCnt     DW ?    ;galvučių skaičius
HidnSecLo   DW ?    ;paslėptų sektorių skaičius
HidnSecHi   DW ?
TotSecs     DD ?    ;bendras sektorių visame diske skaičius
```

HidnSec nurodo sektorių, nepriklausančių disko loginei struktūrai, skaičių.

Paleidžiamasis sektorius kompiuterio įkrovimo metu BIOS programomis yra nuskaitomas į buferį, kuris prasideda adresu **7C00:0000** ir šiuo adresu perduodamas valdymas.

Failų išdėstymo lentelė (FAT – File Allocation Table) talpinama už paleidžiamojo sektoriaus. Ši lentelė prasideda BPB sektoriuje ResSecs ir turi informaciją apie kiekvieno klasterio priklausomumą vienam ar kitam failui. Paprastai diske sukuriamos kelios identiškos FAT kopijos (dažniausiai dvi). FAT – tai vienarūšė elementų struktūra, kurių kiekvienas logiškai susijęs su vienu disko klasteriu. FAT elementai numeruojami nuo 0, tačiau pirmieji du turi specialią paskirtį, todėl klasterių numeriai prasideda nuo 2. Norint nustatyti ar N klasteris užimtas, reikia išanalizuoti FAT elementą. Jei elemento turinys lygus 0, vadinasi klasteris laisvas. Priešingu aveju klasteryje patalpinta nuoroda į kitą su juo surištą klasterį arba nuorodų grandinės pabaigos žymė.

Skiriant diskinę vietą naujam failui, FAT lentelėje ieškoma pirmo neužimto klasterio ir klasterio numeris talpinamas į su šiuo failu susijusį katalogą. Jei failas pilnai telpa viename klasteryje, tai atitinkamame FAT elemente talpinamas užimtumo požymis. Priešingu atveju, talpinamas to klasterio numeris, kuriame patalpintas failo tęsinys.

Katalogų struktūra. Iš karto už paskutinės FAT kopijos diske talpinamas šakninis katalogas. Šio katalogo turinys aprašo visus jame užregistruotus pakatalogius ir failus. Kiekvienas šakninio katalogo ir visų jo pakatalogių elementas vaizduojamas 32 baitų duomenų struktūra:

```
Name        DB 8 DUP(?)     ;failo pavadinimas
Ext         DB 3 DUP(?)     ;prievardis
FAttr       DB ?            ;failo atributai
Reserv      DB 10 DUP(?)    ;rezervas
```

```
Time    DW ?        ;sukūrimo laikas
Date    DW ?        ;sukūrimo data
FirstC  DW ?        ;pirmojo klasterio numeris
Size    DD ?        ;failo dydis baitais
```

Laukas **FirstC** nurodo klasterio numerį, kuriame patalpinta failo arba katalogo pradžia. Tuo pačiu šis numeris pradeda FAT nuorodų grandinėlę, jei failas ar katalogas netelpa viename klasteryje.

10.2 pratimas. Tarkime, kad operacijų sistema skiria minimalią 64 baitų sritį vienam failui saugoti. Kiek klasterių užims 100 Kbaitų dydžio failas?

10.2 BIOS priemonės darbui su disku

Į BIOS (Basic Input Output System) sudėtį įeina darbui su disku skirta programa, kurios funkcijomis galima pasinaudoti INT 13h dėka. Kaip ir kiti BIOS pertraukimai šis pertraukimas numato keletą funkcijų, susijusių su tam tikrais veiksmais. Šios funkcijos apibrėžiamos registro AH turiniu ir turi lentelėje nurodytą prasmę.

10.1 lentelė. Galimos INT 13h funkcijos.

Funkcija	Veiksmas
00h	Perkrauti disko sistemą
01h	Nuskaityti disko būseną
02h	Skaityti sektorių
03h	Įrašyti sektorių
04h	Patikrinti sektorių
05h	Formatuoti takelį (cilindrą)
06h	Formatuoti kietojo disko takelį (cilindrą)
07h	Formatuoti kietąjį diską
08h	Nuskaityti disko parametrus
09h	Inicializuoti disko kontrolerį disko lentelių pagrindu
0Ah	Skaityti sektorių taikant korekcijos kodą
0Bh	Rašyti sektorių taikant korekcijos kodą
0Ch	Pozicionuoti galvutes reikiamame cilindre
0Dh	Perkrauti disko sistemą
0Eh	Skaityti sektoriaus buferį
0Fh	Rašyti sektoriaus buferį
10h	Nuskaityti diskasukio būseną
11h	Įvykdyti diskasukio perkrovimą
12h	Patikrinti diskasukio atmintį
13h	Patikrinti diskasukį
14h	Patikrinti kietojo disko valdiklį
15h	Nuskaityti disko tipo ir dydžio parametrus
16h	Patikrinti disketės žymę
17h	Nustatyti disketės tipą
18h	Nustatyti aplinką disketės formatavimui
19h	Parkuoti kietojo disko galvutes
1Ah	Formatuoti kietąjį diską naudojant ESDI tipo valdiklį

Kadangi emuliatoriuje emu8086 kol kas palaikomos tik 00h, 02h ir 03h INT 13h funkcijos, tai smulkiau aptarsime tik jas.

INT 13h / AH = 00h

Pradiniai duomenys:

DL = disko numeris.

Įvykdžius šia funkciją disko galvutės nustatomos į nulinį cilindrą.

INT 13h / AH = 02h - skaityti disko sektorius į atmintį
INT 13h / AH = 03h - rašyti disko sektorius

Pradiniai duomenys:

AL = sektorių, kuriuos reikia nuskaityti/įrašyti, skaičius (> 0);
CH = cilindro numeris (0..79);
CL = sektoriaus numeris (1..18);
DH = galvutės numeris (0..1);
DL = diskasukio numeris (0..3, priklausomai nuo FLOPPY_? failų skaičiaus);
ES:BX rodo į atminties buferį.

Gražinami rezultatai:

CF nustatomas į 1, jei įvyko klaida;
CF lygus 0, jei užduotis atlikta sėkmingai;
AH = būsena (0 jei sėkmingai įvykdyta);
AL = persiųstų sektorių skaičius.

Čia nurodytas cilindrų, sektorių juose ir galvučių skaičius atitinka 1,4 Mbaito diskelį. Emuliatoriuje emu8086 fiziniame lygmenyje galime dirbti tik su virtualiu lanksčiojo diskelio įrenginiu.

10.1 pavyzdys. Norint nuskaityti iš lanksčiojo diskelio 2 sektorius (1 Kbaitą informacijos) patalpintus pirmajame cilindre turime atitinkamai nustatyti registrų reikšmes:

```
MOV AL, 2          ;nuskaitomų sektorių skaičius
MOV CX, 0101h      ;CH=1 cilindro numeris, CL=1 sektoriaus, nuo
                   ;kurio pradėsime skaityti informaciją, numeris
MOV DX, 0000h      ;DH=0 nulinė galvutė, DL=0 įrenginio numeris

MOV AH, 2          ;disko sektorių skaitymo į atmintį funkcijos
                   ;pakrovimas
INT 13h            ;pertraukimo iškvietimas
```

10. skyriaus savikontrolės klausimai

Nr.	Klausimas	Atsakymas	Komentaras
1.	Tarkime, kad diske yra 8 galvutės, 1024 cilindrai, kiekviename cilindre yra po 46 sektorius. Kokia šio disko talpa Mbaitais?	60	Atsakymas neteisingas. 1 Mbaitas yra 1024x1024 baitų, o bendra formulė yra galvutės x cilindrai x sektoriai x 512 baitų.
		184	Atsakymas teisingas.
		200	Atsakymas neteisingas. 1 Mbaitas yra 1024x1024 baitų, o bendra formulė yra galvutės x cilindrai x sektoriai x 512 baitų.
2.	Kuris iš išvardintų teiginių apie FAT yra neteisingas?	FAT saugo informaciją apie kiekvieno klasterio priklausomumą failui.	Atsakymas neteisingas. Nes šis teiginys apie FAT yra korektiškas.
		Diske sukuriamos kelios identiškos FAT kopijos.	Atsakymas neteisingas. Nes šis teiginys apie FAT yra korektiškas.
		FAT lentelės kiekvienas elementas yra susijęs su keliais disko klasteriais.	Atsakymas teisingas.

3.	Kas vyks, jei turėsime atitinkamai nustatytus registrus:
	
	MOV AL, 2
	MOV CX, 0101h
	MOV DX, 0000h
	MOV AH, 3
	INT 13h
		Pradedant nuo antro cilindro pirmo sektoriaus nulinį diskelį naudojant pirmą galvutę bus įrašyti du sektoriai informacijos.	Atsakymas neteisingas. Kadangi registruose esanti informacija turi tokią prasmę:
			
			AL - sektorių, kuriuos reikia nuskaityti/įrašyti, skaičius (> 0);
			CH - cilindro numeris (0..79);
			CL - sektoriaus numeris (1..18);
			DH - galvutės numeris (0..1);
			DL - diskasukio numeris (0..3, priklausomai nuo FLOPPY_? failų skaičiaus).

		Pradedant nuo nulinio cilindro pirmo sektoriaus į pirmą diskelį naudojant nulinę galvutę bus įrašyti du baitai informacijos.	Atsakymas neteisingas. Kadangi registruose esanti informacija turi tokią prasmę: AL - sektorių, kuriuos reikia nuskaityti/įrašyti, skaičius (> 0); CH - cilindro numeris (0..79); CL - sektoriaus numeris (1..18); DH - galvutės numeris (0..1); DL - diskasukio numeris (0..3, priklausomai nuo FLOPPY_? failų skaičiaus).
		Pradedant nuo pirmo cilindro pirmo sektoriaus į nulinį diskelį naudojant nulinę galvutę bus įrašyti du sektoriai informacijos.	Atsakymas teisingas.

Papildomi skaitiniai

[1] В. В. Фаронов, Turbo Pascal 7.0. Практика программирования. Москва, 1998. – 432 с. (54 p. – 83 p.) ISBN 5-89251-012-3

Mokymosi medžiagoje naudota literatūra

[1] A. Kiseliovas, Matematika. Vilnius: Mokslo ir enciklopedijų l-kla, 1994. – 296 p.
[2] A. Mitašiūnas, Kompiuterių architektūra, Mokymo priemonė. VU Informatikos katedra, 2003. – 126 p. (el. leidinys)
[3] В. Юров, Assembler, Учебник. СПб: Издательство «Питер», 2000. – 624 с. ISBN 5-272-00040-4
[4] В. И. Пустоваров, Язык Ассемблер в програмировании информационных и управляющих систем. Москва, «ЭНТРОП», Киев, «Век» 1997. – 304 с. ISBN 5-88547-052-9
[5] В. В. Фаронов, Turbo Pascal 7.0. Начальний курс. Москва, 1997. – 616 с. ISBN 5-89251-012-3
[6] В. В. Фаронов, Turbo Pascal 7.0. Практика программирования. Москва, 1998. – 432 с. ISBN 5-89251-012-3
[7] http://www.ziplib.com/emu8086help/index.html
Online Assembly Language Tutorial and Reference (p.k. žiūrėta 2004-04-30)