Esipuhe

Tämä moniste on syntynyt Tietokonejärjestelmät kurssin konekielisen ohjelmoinnin osan pohjalta. Moniste ei ole sellaisenaan mikään konekielisen ohjelmoinnin itseopiskeluopas, vaan lukijan täytyy itse osata lukea oikeat asiat ja tarvittaessa palata aikaisemmin lukemaansa asiaan.

Enemminkin moniste on eräänlainen hakuteos konekielisestä ohjelmoinnista, hyvän itseopiskelumonisteen vaatima asteittainen asioiden esittely puuttuu; samat asiat on pyritty sanomaan vain yhden kerran. Lisäksi monisteessa on useissa paikoissa esitetty enemmän asiaa kuin Tietokonejärjestelmät kurssilla edellytetään osattavaksi.

Esimerkkiprosessoriksi on valittu monisteen kirjoittamisen aikana yksi maailman eniten käytetyistä prosessoreista: Intel 8086. Valinta ei sinänsä tee monistetta hyödyttömäksi muidenkaan prosessoreiden konekielen opettelemisessa, erityisesti Intelin uudemmissa prosessoreissa 8086:n käskyt ovat osajoukkona.

Tietokoneiden kehityksen vauhtia kuvaa hyvin se, että tämän monisteen syntymiseen tarvitun parin vuoden aikana on konekielisen ohjelmoinnin merkitys vähentynyt mikrotietokoneissakin. Koneiden tehot ovat kasvaneet niin, että yleensä C-kielisillä ohjelmilla saavutetaan riittävä suorituskyky. Toivottavasti moniste kuitenkin auttaa osaltaan ymmärtämään tietokoneen toimintaa.

Lopuksi kiitokset Tapani Tarvaiselle monisteen tekstin kriittisestä oikolukemisesta.

Jyväskylässä 16. tammikuuta 1990

Monisteen 2. painokseen on korjattu edellisessä monisteessa olleita painovirheitä sekä lisätty C-kieleen liittyvien esimerkkien lukumäärää.

Palokassa 23. joulukuuta 1991

Vesa Lappalainen

Luku 1

Tiedon esittäminen tietokoneessa

1.1 Lukujärjestelmät

1.1.1 10-järjestelmä.

Käytämme 10-järjestelmää, jossa on 10 eri symbolia lukuarvon esittämiseen (0-9). Lisäksi käyttämämme järjestelmä on paikkajärjestelmä, eli kun yksittäisiä lukuja vastaavat symbolit laitetaan peräkkäin, määrää myös niiden paikka muodostuvan luvun suuruuden.

Esimerkki:

$$
\begin{aligned}
175 &= 1 \text{ sataa} + 7 \text{ kymmentä} + 5 \text{ ykköstä} \\
&= 1*100 + 7*10 + 5 \\
&= 1*10^2 + 7*10^1 + 5*10^0
\end{aligned}
$$

Siis jos luvussa olevat symbolien paikat numeroidaan oikealta vasemmalle alkaen nollasta, saadaan luvun arvo selville summaamalla kussakin paikassa oleva arvo kerrottuna kantaluku potenssiin paikan numero.

Aivan samalla tavalla tulkitaan myös muut lukujärjestelmät. Kantalukuna voi olla mikä tahansa kokonaisluku joka on suurempi kuin 1. Jos kantaluku on suurempi kuin 10, joudutaan symbolin 9 jälkeen keksimään uusia symboleja.

1.1.2 2-järjestelmä eli binäärijärjestelmä.

Koska nykyisissä tietokoneissa talletetaan eri suuruisia sähköisiä varauksia, jännitteitä, erilaisia magneettikentän suuntia jne., ei kymmenjärjestelmä ole kovinkaan luonnollinen tiedon talletustapa. Eräs luonnollisimmista tiedon esitystavoista tietokoneessa on tulkita jännite jossakin paikassa vastaamaan lukuarvoa 1 ja jännitteen puuttuminen lukuarvoa 0. Jos normaalina jännitteenä pidetään esimerkiksi + 5 volttia, voidaan tarvittaessa jännitteet 0-2.5 volttia tulkita lukuarvoksi 0 ja jännitteet 2.5-5 volttia tulkita lukuarvoksi 1. Häiriövara on siis varsin suuri.

Tietokoneissa käytetään siis yleensä järjestelmää, jossa on kaksi symbolia lukuarvon esittämiseen. Tätä nimitetään 2-järjestelmäksi. Vaikka symboli (0-2.5 volttia esimerkiksi) ei välttämättä olekaan kovin täsmällinen, voidaan tietyn rajan sisään sattuvat arvot tulkita samaksi symboliksi. Näinhän itse asiassa ihmisetkin tekevät lukiessaan toistensa kirjoittamia numeroita. Ihmiset käyttävät näistä tietokoneen symboleista nimiä 0 ja 1.

Kuten kymmenjärjestelmässäkin, saadaan 2-järjestelmässä luvun arvo selville summaamalla vastaavat potenssit.

Esimerkki:

$$
\begin{aligned}
&76543210 \\
&11101101_2 = 1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \\
&\qquad = 128 + 64 + 32 + 8 + 4 + 1 = 237_{10}
\end{aligned}
$$

Koska alaindeksin kirjoittaminen esimerkiksi ohjelmakoodiin on hankalaa, korostetaan binäärilukua usein ohjelmassa kirjoittamalla B-kirjain bittisarjan perään, eli edellisen esimerkin luku voitaisiin kirjoittaa: 11101101B

1.1.3 8-järjestelmä eli oktaalijärjestelmä

Ihmiselle pitkän binääriluvun tulkinta on hankalaa. Ryhmittelemällä binääriluku oikealta alkaen 3-bitin ryhmiin saadaan luvulle uusi esitys jossa kullakin numerolla on kahdeksan eri vaihtoehtoa. Kun kutakin kolmen bitin ryhmää merkitään vastaavalla 10-järjestelmän luvulla 0-7, saadaan binääriluvulle vastaava 8-järjestelmän esitys. Oktaalilukua korostetaan usein O kirjaimella tai joskus Q kirjaimella.

Esimerkki:

$$11101101B = 11\ 101\ 101_2 = 3\ 5\ 5_8$$
$$= 3*8^2 + 5*8^1 + 5*8^0 = 3*64 + 5*8 + 5$$
$$= 237_{10}$$

1.1.4 16-järjestelmä eli heksajärjestelmä

Kuten 8-järjestelmääkin, käytetään 16-järjestelmää sen vuoksi, ettei tarvitsisi kirjoittaa pitkiä binäärilukuja. Nykyisissä tavujakoisissa prosessoreissa heksajärjestelmänä kuvaaminen on luonnollista, koska 8 bitin tavu on helppo jakaa kahdeksi 4 bitin ryhmäksi, jotka sitten kumpikin ryhmä tulkitaan vastaavalla heksasymbolilla.

Koska kantalukuna on 16, tarvitaan 10-järjestelmän 9 jälkeen 6 uutta symbolia kuvaamaan lukuarvoja 10-15. Valitaan näiksi symboleiksi kirjaimet A-F.

Seuraavassa taulukossa on esitetty 2-, 8-, 16- ja 10-järjestelmän välinen riippuvuus:

B	O	H	D
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Siis esimerkiksi 1011B = 13Q = 0BH = 11D.

Edellisestä esimerkistä huomaamme, että myös lukujärjestelmää vastaavan kirjaimen kirjoittaminen saattaa aiheuttaa sekaannusta, mikäli ei olla varmoja siitä onko kirjain kirjoitettu luvun perään vai ei. Esimerkiksi edellä saatettaisiin tulkita että 1011B on viisinumeroinen heksaluku ja 11D 3-numeroinen heksaluku. Yleensä tämä sekaannus vältetään siten, että heksaluvun perässä PITÄÄ OLLA H (tai TURBO PASCALissa edessä $ tai C:ssä 0x).

Tehtävä 1.1 Lukujen lukumäärä

Montako eri lukua voidaan esittää k-järjestelmän n:llä numerolla?

1.2 Lukujärjestelmien väliset muunnokset

Siirtyminen binääriluvusta vastaavaan heksalukuun voidaan siis tehdä ryhmittelemällä bittijono oikealta alkaen 4 bitin ryhmiin, jotka sitten tulkitaan vastaavaksi heksasymboliksi. Vastaavasti voitiin binääriluku muuttaa oktaaliluvuksi 3 bitin ryhmiä käyttämällä.

Käänteinen muunnos saadaan kirjoittamalla heksalukua vastaavat 4 bitin ryhmät tai oktaalilukua vastaavat 3 bitin ryhmät. Myös oktaali-heksa ja heksa-oktaali muunnokset kannattaa käsin muunnettaessa tehdä tätä kautta.

Esimerkki:

```
103Q = 001 000 011 B = 0 0100 0011 B = 0000 0100 0011 B
     = 043H
```

Lukujärjestelmästä 10-järjestelmään siirtyminen tehtiin määritelmän mukaan summaamalla kantaluvun vastaavien potenssien lukumäärät.

Kymmenjärjestelmästä voidaan siirtyä k-järjestelmään mekaanisesti seuraavalla algoritmilla:

0. Laita tulos tyhjäksi.
1. Jaa luku kantaluvulla k.
2. Lisää jakojäännös tulokseen VASEMMAKSI numeroksi.
3. Laita kokonaisosa luvuksi.
4. Jos luku<>0 niin jatka kohdasta 1.
5. Tulos on valmiina.

Esimerkki: 175_{10} 8-järjestelmään:

```
175/8  = 21 7/8
 21/8  =  2 5/8
  2/8  =  0 2/8
```

Luku on siis 257_8

Esimerkki: 175_{10} 2-järjestelmään:

```
175/2  =  87 1/2
 87/2  =  43 1/2
 43/2  =  21 1/2
 21/2  =  10 1/2
 10/2  =   5 0/2
  5/2  =   2 1/2
  2/2  =   1 0/2
  1/2  =   0 1/2
```

Luku on siis 10101111B, mikä olisi saatu myös edellisen esimerkin 8-järjestelmän luvusta kirjoittamalla vastaavat kolmen bitin ryhmät.

Tehtävä 1.2 Lukujärjestelmien väliset muunnokset

Esitä seuraavat 10-järjestelmän luvut 2-,8- ja 16-järjestelmässä:

127 128 255 256 1111

1.3 Desimaaliluvut

Kymmenjärjestelmässä desimaalipisteen oikealla puolella olevat numerot tarkoittavat 1/10, 1/100 jne., eli 10^{-1}, 10^{-2} jne.

Vastaavasti voidaan tulkita muissakin lukujärjestelmissä:

Esimerkki:

$$12.45_8 = 1*8^1 + 2*8^0 + 4*8^{-1} + 5*8^{-2}$$
$$= 10 + 0.5 + 0.078125 = 10.578125_{10}$$

Kymmenjärjestelmän luku muutetaan k-järjestelmän desimaaliluvuksi muuttamalla kokonaisosa ja desimaaliosa erikseen. Kokonaisosa muutetaan edellä kerrotulla menetelmällä. Kymmenjärjestelmän luvun desimaaliosa voidaan muuttaa k-järjestelmän desimaaliosaksi seuraavalla algoritmilla:

0. Laita tulokseen 0. (nolla ja .).
1. Kerro luku kantaluvulla.
2. Laita kokonaisosa tuloksen seuraavaksi desimaaliksi.
3. Laita lukuun 0 kokonaisosan paikalle.
4. Jos luku<>0, niin jatka kohdasta 1.
5. Tulos on valmis.

Esimerkki. 0.625_{10} 2-järjestelmään:

```
0.625*2 = 1.25
0.250*2 = 0.5
0.500*2 = 1.00
```

Tulos on siis 0.101_2.

Esimerkki. 0.1_{10} 2-järjestelmään:

```
0.100*2 = 0.2
0.200*2 = 0.4
0.400*2 = 0.8
0.800*2 = 1.6
0.600*2 = 1.2
0.200*2 = 0.4
0.400*2 = 0.8
0.800*2 = 1.6
0.600*2 = 1.2
       ...
```

Huomaamme, että algoritmissa tulee toistuvasti vastaan sama silmukka 0.2*2, 0.4*2, 0.8*2, 0.6*2. Voimme siis päätellä, ettei algoritmi tule koskaan loppumaan. Tulos on siis päättymätön 2-järjestelmän desimaaliluku 0.00011001100110011... Tämä on esimerkki siitä, ettei kaikkia 10-järjestelmän päättyviä desimaalilukuja voida esittää päättyvinä k-järjestelmän desimaalilukuina. Tämä on eräs syy myöhemmin esitettäviin pyöristysvirheisiin.

Tehtävä 1.3 Desimaaliluvut

Muuta seuraavat 10-järjestelmän luvut binääriluvuiksi:

1.1 10.10 12.12 12.75

1.4 BCD-luvut

Joissakin, erityisesti kaupallis-hallinnollisissa, sovellutuksissa saattaa olla etua 10-järjestelmän käyttämisestä. Usein 10-järjestelmää käytetään myös taskulaskimissa. Tällöin muutettaessa 10 järjestelmän luku BCD-koodiin (Binary Coded Decimal), tulkitaan kukin numero vastaavana 4 bitin ryhmänä. BCD voi olla pakattu, jolloin 8 bitissä on kaksi numeroa tai pakkaamaton, jolloin tavussa on vain yksi BCD-numero.

Esimerkki: 2175_{10} on 0010 0001 0111 0101 pakattuna BCD:nä.

BCD-esityksen haittana on tilan tuhlaaminen, eli 8 bitillä voidaan esittää vain luvut 0-99 (kun binäärisenä saatiin luvut 0-255). Toinen haitta on vaikeammat laskutoimitukset, tämä haitta on kuitenkin osittain hoidettu eräiden prosessoreiden käskykannassa.

Esityksen etuna on eräiden pyöristysvirheiden välttäminen, koska laskut tehdään ’10-järjestelmässä’. Tosin tavallisilla liukuluvuillakin voidaan välttää esim. pennejä koskevat pyöristysvirheet käyttämällä markkojen sijasta pennejä ja riittävän monta bittiä mantissalle.

1.5 2-järjestelmän lukujen yhteenlasku

10-järjestelmässä yhteenlasku tapahtuu seuraavasti. Yhteenlaskettavat luvut kirjoitetaan allekkain ja summaa lähdetään muodostamaan oikealta laskemalla yhteen vastaavissa paikoissa olevat luvut. Mikäli summa ylittää 10, siirretään yli menevä osa muistinumeroksi seuraavaan lukuun. Aivan vastaavasti voidaan laskea yhteen 2-järjestelmän luvut.

Seuraavassa on **totuustaulu** lukujen yhteenlaskemista varten. Edellisestä paikasta saapuvalle muistinumerolle käytetään nimitystä c_i ja seuraavaan paikkaan siirtyvästä muistinumerosta nimitystä c_o (carry in ja carry out). Yhteenlaskettavat ovat a ja b sekä niiden summa on s.

ci	a	b	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Esimerkki. 3+6:

```
c    0110
3     0011
6     0110
s    01001 = 9_10
```

1.6 Kerto- ja jakolasku

Kuten yhteenlaskukin, voidaan kertolasku tehdä koulussa opitun algoritmin mukaisesti. Kerrottaessa kaksi n-bittistä lukua keskenään, on kuitenkin huomattava, että tulos voi olla 2n-bittiä.

Esimerkki: 3*9 4-bitillä:

```
 3              0011
*9              1001
-----------------------
                0011    1*3
        +      00000    0*3
        +     000000    0*3
        +    0011000    1*3
-----------------------
        =   00011011
```

Huomattakoon, että yhteenlaskun helpottamiseksi kertolaskulle voidaan kirjoittaa seuraava algoritmi:

0. Laita tulos nollaksi.
1. Mikäli kertoja on nolla jatka kohdasta 7.
2. Ota kertojan oikeanpuoleinen bitti ja poista se. (Rullaa kertojaa askel oikealle.)
3. Mikäli bitti oli 0, jatka kohdasta 5.
4. Lisää tulokseen kerrottava.
5. Lisää kerrottavaan bitti 0 oikeaan laitaan. (Eli rullaa kerrottavaa vasemmalle.)
6. Jatka kohdasta 1.
7. Tulos on valmis.

Myös jakolasku voidaan tehdä koulusta tutulla ’jakokulmalla’:

0. Laita osamäärä nollaksi.
1. Lisää jakajaan nollia oikealle, kunnes yhtä pitkä kuin jaettava.
2. Lisää osamäärään 0 bitti oikealle.
   (Rullaa osamäärää vasemmalle.)
3. Mikäli jaettava < jakaja, niin jatka kohdasta 5.
4. Vähennä jakaja jaettavasta. Lisää osamäärään 1.
5. Poista jakajan oikeanpuoleinen bitti.
   (Rullaa jakajaa oikealle.)
6. Mikäli jakaja >= alkup. jakaja, niin jatka kohdasta 2.
7. Osamäärä on valmis ja jaettava on jakojäännös.

Esimerkki: 27/4:

```
                                 Jakaja  jaettava   osamäärä
Tehdään kohta 0:                   100     11011      0
tehdään kohdat 1 ja 2:             10000   11011      00
tehdään kohdat 3, 4 ja 5:          1000     1011      01
tehdään kohdat 6, 2, 3, 4 ja 5:    100       011      011
tehdään kohdat 6, 2, 3, 5:         10        011      0110
tehdään kohdat 6 ja 7:                jakojäännös=3  osamäärä=6
```

Käytännössä edellinen algoritmi vaatii yhden ’ylimääräisen’ rekisterin (prosessorin sisäinen nopea muistipaikka) alkuperäisen jakajan tallettamiseen. Mikäli rekistereitä ei ole käytössä, voidaan algoritmia ’viilata’ siten, että osamäärään piilotetaan ylimääräinen bitti, jonka ’valuessa’ vasemmalta ulos, voidaan algoritmi lopettaa.

Tehtävä 1.4 Kerto- ja jakolasku

Laske seuraavat laskut 2-järjestelmässä:

10*10 15*15 175/11 175/15

1.7 Negatiiviset luvut

Kymmenjärjestelmässä negatiiviset luvut ilmaistaan laittamalla miinusmerkki luvun eteen. Tämä kuitenkin vaatii taas uuden symbolin käyttöönottoa. Tietokoneessa sovittiin, että on vain 2 symbolia 0 ja 1, joilla kaikki asiat pitää voida esittää. Voitaisiin tehdä sellainen sopimus, että bittijonon 1. merkki tarkoittaa etumerkkiä (vaikkapa 0=+ ja 1=-). Joissakin koneissa menetelläänkin näin, mutta myöhemmin huomaamme, että mekaanisesti laskettaessa jokin muu menetelmä saattaa olla parempi.

Tähän saakka on käsitelty lukuja, joissa voi olla mielivaltainen määrä numeroita. Käytännön tietokoneessa täytyy täsmälleen tietää, kuinka monta bittiä kukin lukua vastaava muistipaikka tulee viemään, jotta sille voidaan etukäteen varata tilaa muistista. Tavujakoisissa koneissa on yleensä 8-bitin, 16-bitin ja 32-bitin kokonaisluvut sekä 32-bitin ja 64 bitin liukuluvut (esitys reaaliluvuille, palataan myöhemmin).

1.7.1 Suora tulkinta

Sovitaan seuraavassa yksinkertaisuuden vuoksi, että koneessa on 3 bitin kokonaisluvut sekä yksi bitti etumerkille. Tällaisessa koneessa voitaisiin tosin esittää vain luvut -7..+7.
Tällöin luku $+3_{10}$ olisi 0011 ja -3_{10} olisi 1011.

1.7.2 1-komplementti

Edellisellä tulkinnalla tulisi tiettyjä ongelmia laskutoimituksia suoritettaessa. Tämän takia joissakin koneissa käytetään negatiivisille luvuille 1-komplementti esitystä:

Mikäli luku on positiivinen, kirjoitettaan se sellaisenaan. Mikäli luku on negatiivinen, käännetään jokaisessa paikassa oleva bitti päinvastaiseksi.

Esimerkki: $+3_{10}$ = 0011 ja -3_{10} = 1100.

1.7.3 2-komplementti

1-komplementtiesityskään ei aina ole riittävän yksinkertainen. Tämän vuoksi suurimmassa osassa nykyisiä tietokoneita käytetään 2-komplementtiesitystä negatiivisille luvuille:
Positiiviset luvut esitetään sellaisinaan ja negatiivisista otetaan ensin 1-komplementti ja lisätään tulokseen sitten 1.

Esimerkki: $+3_{10}$ = 0011 ja -3_{10} saadaan seuraavasti:

```
+3              0011    otetaan ensin
1-komplementti  1100    ja kun tähän lisätään 1 saadaan
2-komplementti  1101    = -3.
```

1.7.4 Eri menetelmien vertailua:

Seuraavassa on taulukko, jossa 4 bittinen luku on tulkittu 4 eri tavalla:

bitit	ei etum.	suora tulk.	1-k	2-k
0000	0	+0	+0	+0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Kaikilla menetelmillä saadaan luku itse takaisin, kun etumerkin vaihtamismenetelmää sovelletaan 2 kertaa peräkkäin. Tämä on tietysti välttämätön ominaisuus, mikäli lukuja halutaan käyttää laskemiseen.

Suorassa tulkinnassa ja 1-komplementissa on 2 eri esitystä 0:lle. Tämä ei yksikäsitteisyyden takia ole toivottava ominaisuus. Mikäli lukua joskus joudutaan vertaamaan 0:aan, täytyy tarkistaa onko se +0 tai -0.

Suoran tulkinnan yksi harvoista eduista on se, että itseisarvon ottaminen on helppoa: laitetaan etumerkkibitti 0:ksi. Muihin tulkintoihin tämä ei päde, vaan niissä joudutaan aina suorittamaan testi: Positiivinen jätetään koskemattomaksi ja negatiiviseen sovelletaan etumerkin vaihtomenetelmää.

Jokaisessa esitystavassa negatiivinen luku alkaa 1-bitillä.

Esimerkki yhteenlaskusta kullakin menetelmällä: 3+(-4)

	suora tulkinta	1-k	2-k
c	0000	0011	0000
+3	0011	0011	0011
-4	1100	1011	1100
	1111	1110	1111
s	-7	-1	-1

Edellä 1-komplementilla ja 2-komplementilla saatiin oikea tulos.

Esimerkki: -1-4

	suora tulkinta	1-k	2-k
c	1000	1110	1100
-1	1001	1110	1111
-4	1100	1011	1100
	0101	1001	1011
s	+5	-6	-5

Edellä vain 2-komplementilla saatiin oikea tulos. Todettakoon kuitenkin, että 1-komplementillakin päästäisiin oikeaan tulokseen, mikäli muistinumero lisättäisiin vielä kerran lopulliseen tulokseen.

Voidaan kuitenkin todeta, että 2-komplementille jää selvät edut muihin menetelmiin verrattuna: yksi ainoa esitys 0:lle ja yhteenlasku menee automaattisesti oikein.

Tästä syystä käytämmekin jatkossa vain 2-komplementtiesitystä negatiivisille luvuille. (Tosin esimerkiksi yliopiston vanhassa Sperry 1100-koneessa oli 1-komplementtiesitys negatiivisille kokonaisluvuille. Samoin on vasta Suomeen ostetussa CRAY X/MP -supertietokoneessa.)

1.8 Lukualue ja ylivuoto

Kiinteällä määrällä bittejä voidaan esittää tietenkin vain tietyn suuruisia lukuja. 4-bitillä suurin mahdollinen luku on 1111, mikä on yhtä vailla 10000 (eli, 16. Vrt. 999 ja 1000). Siis suurin 4 bitillä esitettävä luku on 2^4-1. Vastaavasti n:llä bitillä voidaan esittää luvut 0..2^n-1.

Jos myös etumerkki tulkitaan mukaan, jää n:stä bitistä n-1 luvun lukuarvon osoittamiseen. 2-komplementtia käytettäessä lukualue on -2^{n-1}..2^{n-1}-1.

Katsotaan esimerkin vuoksi muutama lasku käyttäen 2-komplementtia negatiivisille luvuille:

-1-4:			1+4		
c	**11**00		c	**00**00	
-1	1111		+1	0001	
-4	1100		+4	0100	
	1011	= -5		0101	= +5
-4-5:			**4+5**		
c	**10**00		c	**01**00	
-4	1100		+4	0100	
-5	1011		+5	0101	
	0111	= +7		1001	= -7

Kaksi ensimmäistä laskua (-1-4 ja 1+4) menivät oikein aivan kuten olettaa saattoikin, koska 4 bitillä esitettävä lukualue on [-8,7] ja tulokset mahtuvat sinne. Jälkimmäisten laskujen tulokset olisivat olleet -9 ja +9, jotka ovat jo esitettävän lukualueen ulkopuolella. Tapahtui siis **ylivuoto** (eli luku on itseisarvoltaan liian suuri).

Mekaanisesti kokonaislukujen yhteenlaskussa ylivuoto todetaan siten, että merkkibittiin tuleva muistinumero **EI** ole sama kuin sieltä lähtevä. Siis kun KAIKKI MUISTINUMEROT kirjoitetaan näkyviin, on 2 vasemmanpuoleisen oltava samat, jottei ylivuotoa tulisi. Käytännön piiritasolla tämä voidaan todeta käyttämällä XOR-piiriä.

Tehtävä 1.5 2-komplementti

Suorita seuraavat laskut 8 bitin koneessa. Negatiivisille luvuille 2-komplementtiesitys. Luvut on annettu 10-järjestelmässä.

10+81 81-10 81+81 -81-81

1.9 Liukuluvut

Vaikka käyttäisimme 64 bitin kokonaislukuja, olisi suurin esitettävä luku 2^{63}-1, eli noin 10^{18}. Tämä ei tietenkään riitä likimainkaan kaikkiin luonnontieteen vaatimiin laskuihin. Toisaalta kokonaisluvuilla ei voida esittää 1 pienempiä lukuja. Tämä vuoksi kokonaislukujen lisäksi käytetään liukulukuja. Usein liukuluvut ovat vielä normeerattuja.

Liukuluku koostuu kolmesta osasta: luvun **etumerkistä**, **mantissasta** ja kantaluvun **eksponentista**.

etum.	mantissa	eksponentti

Esimerkki: $3.5*10^3 = 3500$. Tässä 3 on kantaluvun 10 eksponentti ja 3.5 luvun mantissa.

Mantissalla ja eksponentilla on tietty kiinteä pituus. Liukuluvun **normeeraus** tarkoittaa sitä, että luvun mantissa on aina välillä [1/k,1[. Käytännössä tämä tarkoittaa sitä, että luvun mantissa on aina muotoa 0.mxxxx, missä m<>0. Koska 2-järjestelmässä kukin numero voi olla vain 0 tai 1, on normeeratun 2-järjestelmän liukuluvun mantissa aina muotoa 0.1xxxx. Normeeraus voidaan määritellä tehtäväksi myös välille [1,k[.

Koska mantissassa alkuosan (0.) kirjoittaminen vaatisi jälleen uuden symbolin (.) ja alkuosa on aina sama, ei alkuosaa merkitä mitenkään, vaan mantissa alkaa suoraan pisteen jälkeisellä 1 bitillä. Näin säästetään myös tilaa. Toisaalta myös pisteen jälkeinen bitti on normeerauksen takia aina 1, joten usein sitäkään ei merkitä, vaan vastaava bitti tarkoittaa mantissan etumerkkiä tai se jätetään kokonaan kirjoittamatta.

Toinen syy normeeraukseen on se, että -0:aa lukuunottamatta kullakin luvulla on vain yksi esitystapa.

Eksponenttiin lisätään positiivinen luku (nollataso, bias), esimerkiksi 2^{e-1}, missä e on eksponentin bittien lukumäärä. Näin eksponentti voidaan aina tulkita positiiviseksi ja peräkkäiset eksponentit ovat myös peräkkäisiä binäärilukuja. (2-komplementissa yllä olevassa esimerkissä on +7:n jälkeen -8).

Esimerkki: Olkoon koneessa 4 bittiä eksponentille, 5 mantissalle ja bitti luvun etumerkille. Normeeraus välille [0.5,1[ja bias 2^{4-1}.

$10_{10} = 10 * 2^0 = 0.625 * 2^4$

Lasketaan aluksi mantissa:

```
0.625 * 2 = 1.25
0.250 * 2 = 0.5
0.500 * 2 = 1.00
```

$0.625_{10} = 0.101_2$

etu	eksp. $4+2^{4-1}$	mantissa 0.625
0	1100	01000

Huomattakoon, että mantissan 1 bitti on jätetty kirjoittamatta, koska se olisi aina 1. Normeeraus voitaisiin suorittaa myös 2-järjestelmässä:

$10_{10} = 10 * 2^0 = 1010.00000_2 * 2^0$
$= 0101.00000_2 * 2^1$
$= 0010.10000_2 * 2^2$
$= 0001.01000_2 * 2^3$
$= 0000.10100_2 * 2^4$

Käytännössä ovat yleisiä seuraavat liukulukuesitykset:

Normeeraus [1,2[, bias 2^{e-1}-1:

Short Real	32 bittiä, joista 1 etum, 8 eksp. ja 23 mantissalle
Long Real	64 bittiä, joista 1 etum, 11 eksp. ja 52 mantissalle
Temporary Real	80 bittiä, joista 1 etum, 15 eksp ja 63 mantissalle (Tässä myös bitti 2^0 kirjoitetaan, koska luku ei aina ole normeerattu!)

Normeeraus [0.5,1[, bias 128 (tai [1,2[, bias 127!):

Turbo Pascalin Real	48 bittiä, joista 8 eksp, 40 mantissalle, mantissan bitti 2^{-1} on varattu etumerkille.

8086-pohjaisissa prosessoreissa on lisäksi peräkkäiset tavut talletettu muistiin siten, että vähiten merkitsevä tavu on ensimmäisenä. Poikkeuksena TURBO PASCALin REAL-tyyppi, jossa eksponentti on ensimmäisenä.

Tehtävä 1.6 Liukuluvut

Turbo Pascalin reaaliluvulle on varattu 6 tavua. Päättele mitä seuraavat Turbo Pascalin reaaliluvut ovat:

```
 1      81 00 00 00 00 00  = 1
 2      7D CC CC CC CC 4C  = 0.1
 3      84 00 00 00 00 20  = 10
 4      82 00 00 00 00 00
 5      7E CC CC CC CC 4C
 6      85 00 00 00 00 20
 7      82 00 00 00 00 40
 8      7F 99 99 99 99 19
 9      85 00 00 00 00 70
10      87 00 00 00 00 48
11      91 00 00 00 50 43
12      C3 AC C5 EB 78 2D
13      7A 70 3D 0A D7 23
14      70 1B 47 AC C5 27  = 1.0E-5;
15      3E 92 64 08 E5 3C
16      81 00 00 00 00 80  = -1
17      82 00 00 00 00 80
18      82 00 00 00 00 C0  = -3
```

1.10 Ylivuoto ja alivuoto

Mikäli eksponentissa on e bittiä, on pienin mahdollinen eksponentti -2^{e-1}. Siis itseisarvoltaan pienin mahdollinen esitettävä luku vaikkapa 8 bitin eksponentilla on $0.5*2^{-128}$ eli noin $1.5*10^{-39}$. Mikäli laskutoimituksissa luvun itseisarvo menee tätä pienemmäksi, on tulos 0. Tätä sanotaan **alivuodoksi**.
Suurin mahdollinen eksponentti on vastaavasti 2^{e-1}-1. Siis suurin 8 bitin eksponentilla esitettävä luku on $1*2^{127}$ eli noin $1.7*10^{38}$. Mikäli laskutoimituksen tulos ylittää tämän, tulee **ylivuoto**.

1.11 Pyöristysvirheet

Esimerkki:

$0.1_{10} = 0.0001100110011.._2 * 2^0 = 0.1100110011.. * 2^{-3}$

etu	eksp. $-3+2^{4-1}$	mantissa 0.625
0	0101	10011

Luku aukilaskettuna on itseasiassa

$$= +(1*2^{-1} + 1*2^{-2} +0*2^{-3} +0*2^{-4} + 1*2^{-5} + 1*2^{-6})*2^{-3}$$
$$= +(0.5+0.25+0.03125+0.015625)*2^{-3}$$
$$= +0.796875*2^{-3}$$
$$= +0.099609375$$

Kuten odottaa saattoikin, ei (2-järjestelmän) päättymätöntä desimaalikehitelmää voi esittää tarkasti kiinteällä bittimäärällä.
Mikäli normeeratussa mantissassa on m bittiä, merkitsee mantissan viimeinen bitti lukua $2^{-(m+1)}$. $2^{-(m+2)}$ ja sitä pienemmät eksponentit eivät enää mahdu mantissaan. Näiden summaksi tulee $2^{-(m+1)}$. Tämä on siis arvio mantissan mahdolliselle virheelle. Tämä suhteellinen virhe pitää vielä kertoa eksponenttiosalla, mikäli halutaan arvio absoluuttiselle virheelle.

Edellisessä esimerkissä suhteellinen virhe on noin $2^{-(5+1)}$ eli 0.015625. Absoluuttiseksi virhe saadaan kertomalla 2^{-3} eli absoluuttinen virhe on noin 0.002. Aivan näin suuri ei edellisen esimerkin virhe ollut, koska kaksi ensimmäistä mantissasta pois jäänyttä bittiä olivat nollia.

Joka tapauksessa AINA tietokoneella laskettaessa on syytä varautua siihen, että esimerkiksi **0.1*10 <> 1**!

1.12 Kirjainten esittäminen

Kirjaimet esitetään tietokoneissa numeroilla. Aluksi täytyy vain sopia mikä kirjain vastaa mitäkin numeroa. Koodilta toivottavia ominaisuuksia olisivat seuraavat:

- peräkkäisillä kirjaimilla peräkkäinen numero
- isojen ja pienten kirjaimien välinen muunnos yksinkertainen
- paikka myös erikoismerkeille ja numeroita vastaaville kirjaimille
- kansalliset merkistöt voidaan ottaa huomioon

Tutkittaessa tarvittavien merkkien lukumäärää, todetaan, että jo isot ja pienet kirjaimet vievät yhteensä yli 50 paikkaa. Numeroista ja tärkeimmistä erikoismerkeistä tulee noin 30 lisää. Tarvitaan siis vähintään 7 bittinen luku tämän määrän esittämiseen.

ASCII-koodi on alunperin 7 bittinen koodi, jossa koodin 32 ensimmäistä merkkiä on varattu erilaisille erikoistoiminnoille, kuten kuvaamaan rivin vaihtoa (LF, koodi 10), telan palautusta (CR, koodi 13) jne. Seuraavaksi tulevat erikoismerkit ja numerot. Sitten on peräkkäin isot kirjaimet (alkaen A:sta, koodi 65) ja lopuksi pienet kirjaimet (alkaen a:sta, koodi 97).

Osittain ASCII-koodi täyttääkin asetetut vaatimukset. Kuitenkin kansalliset merkistöt on siitä unohdettu täysin. Suomessa on jouduttu luopumaan mm. haka-ja aaltosuluista Ä:n, Ö:n ja Å:n hyväksi.

Toinen ASCII-koodin puute ohjelmoijan kannalta on aakkosjärjestäminen. Raa'alla kirjainten koodien vertailuun perustuvalla algoritmilla esimerkiksi apua on aakkosissa Zorron jälkeen.

Ison ja pienen kirjaimen välinen ero ASCII-koodissa on $97\text{-}65 = 32^{10}$ eli 20H. Siis pienen kirjaimen muuttaminen isoksi tapahtuu vähentämällä pienen kirjaimen koodista 20H. Vastaavasti iso kirjain voidaan muuttaa pieneksi lisäämällä koodiin 20H. Sama muunnos voidaan tehdä myös muuttamalla bitin 5 arvoa.

Eräät tietokonevalmistajat ovat ottaneet käyttöön laajennettuja ASCII-koodeja, joissa merkit, joiden koodi on yli 127 on varattu osittain kansallisia merkkejä varten ja osittain muihin erikoistarkoituksiin. Merkkien paikoista ei kuitenkaan ole tullut mitään yleismaailmallista standardia. Tunnetuimpia koodeja on IBM:n PC-mikroissa käyttämä koodi (katso kohta 1.15) sekä Digitalin ja Hewlett-Packardin koodit.

Mikään näistä laajennetuista koodeista ei enää täytä kunnolla koodin vaatimuksia. Erityisesti kirjainten aakkosjärjestys ei säily. Edellä olevista syistä aakkostamista ei koskaan saisikaan tehdä käyttäen suoraa vertailua. Aluksi pitäisi muuttaa aakkostettavat avaimet sopivalla aliohjelmalla siten, että kaikki kirjaimet muutetaan isoiksi ja kansalliset kirjaimet siirretään aakkostamisen ajaksi omalle paikalleen. Usein samaistetaan joitakin merkkejä (saks. ä->a), tai jopa merkkiyhdistelmiä (Mc->Mac). Siis itse asiassa aakkostamisen aikana käytetään omaa tilapäistä koodia.

Koodimuunnosta ei useinkaan voida enää tehdä aivan yhtä yksinkertaisesti kuin ASCII-koodin muunnosta isot-pienet. Kuitenkin käyttämällä apuna sopivaa muunnostaulukkoa, päästään lähes yhtä nopeaan, joskus jopa nopeampaan, tulokseen.

On myös olemassa muita koodeja, esimerkiksi IBM:n käyttämä EBDIC-koodi.

1.13 Muistin sisällön tulkitseminen

Kuten edellä todettiin, ei ole aivan selvää tarkoittaako bittijono 1100 lukua 12 vai -3 vai -4. Tulkinta riippuu siitä mikä esitys negatiiville luvuille on sovittu.

Seuraavassa on esimerkki 8086 prosessorilla varustetun koneen 8 peräkkäisestä (tavu)muistipaikasta, jotka on tulkittu eri tavoin:

```
8 heksa byte:       41 50 55 41 61 70 75 61
ASCII-jonona:       APUAapua
2 Short Real:       13.3320932388
                    2.82971878229e+20
1 Long Real:        3.01412940137e+161
4 Heksa word:       5041 4155 7061 6175
4 Integer:          20545 16725 28769 24949
7 ohjelma askelta:  41 =   inc    cx
                    50 =   push   ax
                    55 =   push   bp
                    41 =   inc    cx
                    61 =   popa
                    7075 = jo     007C
                    61 =   popa
```

Siis täsmälleen sama muistipaikka voi tarkoittaa aivan eri asioita riippuen siitä miten sisältö tulkitaan. Useinkaan ei ole mitään tapaa tietää sitä, mikä olisi oikea tulkinta. Mikäli muistipaikkojen sisältö näyttää järkevältä tekstiltä, saattaa ASCII-jono olla oikea tulkinta. Käytännössä ohjelmoijan on itse huolehdittava siitä, että muistin sisältöä tulkitaan oikein. Onneksi korkeamman tason kielissä kääntäjä (PASCAL, C, FORTRAN jne.) tai tulkki (BASIC, APL, LISP jne.) huolehtii muistipaikkojen organisoinnista ja tulkinnasta. ASSEMBLER-ohjelmoijan on kuitenkin itse pidettävä huoli muistin tulkinnasta. Väärä tulkinta johtaakin varsin yleiseen ohjelmointivirheiden luokkaan.

Tehtävä 1.7 Muistin tulkinta

Seuraavassa on kahdeksan tavua 8086-prosessorin muistia (heksana):

4B 65 73 84 20 79 94 21

Tulkitse muistin sisältö seuraavin eri tavoin (muista että ko. prosessorissa esitetään vähiten merkitsevä tavu ensin):

a) 8 merkin ASCII-jono.
b) 4 kpl 16-bitin heksalukua.
c) 2 kpl 32 bitin liukulukua (etum, 8 bitin eksp, bias 127, normeeraus [1,2[).
d) 1 kpl 64 bitin liukuluku (etum, 11 bitin eksp, bias 1023, normeeraus [1,2[).
e) 4 kpl 2 tavun positiivista kokonaislukua.
f) 4 kpl 2 tavun kokonaislukua, negatiivisille luvuille 2-komplementti.
g) 4 kpl 2 tavun kokonaislukua, negatiivisille luvuille 1-komplementti.
h) Luvusta 65H alkaen 14 numeron pakatuksi BCD-luvuksi.

Tehtävä 1.8 Binääri- ja oktaaliluvut

Kirjoita edellisen tehtävän muistin sisältö bittijonona.

Kirjoita edellisen tehtävän muistin sisältö 3 numeron oktaaliluvuilla.

1.14 Input ja output

Syötettäessä esimerkiksi päätteeltä numeerisia arvoja, ei pidä luulla, että syöttö menisi numeerisena koneeseen. Näppäimen 5 painaminen lähettää koneella **kirjaimen 5** ASCII-koodin eli 35H. Vastaavasti tulostettaessa ei voida suoraan tulostaa numeerisia arvoja, vaan pitää tehdä muunnos: numeerinen arvo -> ASCII-merkkijono.

1.14.1 Merkkijono numeeriseksi arvoksi

Olkoon meillä merkkijono 175, eli heksakoodina 31H 37H 35H. Tämä muutetaan numeeriseksi arvoksi seuraavasti:

```
Tulos:=0
31H  ->  01H  Tulos:=10*0+1
37H  ->  07H  Tulos:=10*1+7
35H  ->  05H  Tulos:=10*17+5=175
```

Siis algoritmina:

0. Tulos:=0.
1. Mikäli kirjaimet loppu, niin jatka kohdasta 6.
2. Ota seuraava kirjain.
3. Muuta kirjain numeroksi nollaamalla 4 vasemmanpuoleista bittiä.
4. Tulos:=10*Tulos+numero.
5. Jatka kohdasta 1.
6. Tulos on valmis.

Mikäli syötössä esiintyy myös desimaalipiste ja mahdollisesti eksponentti, pitää algoritmia parantaa. Myös mahdollinen miinusmerkki pitäisi ottaa huomioon (miten?).

Onneksi tavallisen ohjelmoijan kannalta esimerkiksi PASCALin READ(kokonaisluku-arvo) huolehtii tarvittavasta muunnoksesta.

1.14.2 Numeerinen arvo merkkijonoksi

Kokonaisluku voitaisiin muuttaa merkkijonoksi siten, että pidettäisiin aluksi laskuria 10000:ille ja vähennettäisiin luvusta 10000 niin kauan kuin voidaan. Sitten siirryttäisiin tuhansiin jne. Lopuksi laskureiden arvot (välillä [0,9]) muutettaisiin merkeiksi lisäämällä luku 30H.

Yleisempi algoritmi saadaan seuraavasta:

0. Laita tulosjono tyhjäksi.
1. Jaa luku 10:llä. Luku:=kokonaisosa.
2. Lisää jakojäännökseen 30H ja laita se tulosjonoon vasemmalle.
3. Mikäli luku<>0, niin jatka kohdasta 1.
4. Tulosjono on valmis.

Käytännössä tulosjonoon vasemmalle lisääminen on työlästä. Tämän takia jono rakennetaan aluksi väärinpäin ja lopuksi käännetään. Usein tämä voidaan tehdä esimerkiksi pinon avulla.

Negatiiviset luvut voidaan käsitellä seuraavasti:

1. Laita etumerkki = +.
2. Mikäli luvun vasemmanpuoleinen bitti = 0, jatka kohdasta 5.
3. Laita etumerkki = -.
4. Ota luvusta 2-komplementti.
5. Jatka edellisellä algoritmilla.

Myös luvun tulostamisesta huolehtii käytännössä kääntäjä. Esim PASCALin WRITE(kokonaislukuarvo) tulostaa päätteelle merkkijonon. Liukuluvun muuttaminen merkkijonoksi on jälleen työläämpää.

Kaikkein yksinkertaisin on muunnos kokonaisluku <-> heksamerkkijono.

1.15 IBM PC laajennettu merkkivalikoima

Luku 2

Konekieli

2.1 Johdanto

Aina kun tietokone suorittaa jotakin ohjelmaa, suoritetaan alimmalla tasolla konekielisiä käskyjä. Edelleen konekieliset käskyt koostuvat useista prosessorin sisäisistä mikrokäskyistä (nouda seuraava käsky, tulkitse käsky jne.). Normaalisti valmisohjelman käyttäjän ei tarvitse huolehtia konekielestä mitään. Korkeamman tason kielillä kirjoittava ohjelmoija kääntää kyseisen kielen kääntäjäohjelmalla alkuperäisen tekstitiedostonsa suorituskelpoiseksi konekieliseksi ohjelmaksi.

Assembler-kieli on symbolista konekieltä, joka assembler-kääntäjällä käännetään konekieliseksi ohjelmaksi. Vaikka jatkossa joskus puhutaankin konekielisestä ohjelmoinnista, tarkoitetaan usein kuitenkin assembler-kielistä ohjelmointia.

2.2 Mihin konekieltä tarvitaan?

2.2.1 Ohjelman toiminnan ymmärtäminen

Aina ei ohjelmoijankaan tarvitse välittää konekielestä yhtään mitään. Joskus kuitenkin hyvin pieni muutos ohjelmakoodissa saattaa tuottaa valtavan nopeuden lisäyksen ohjelman suoritusaikaan tai toisaalta pienentää lopullisen koodin kokoa. Tällaisen muutoksen teko vaatii kuitenkin käytettävän laitteiston sisäisen toiminnan ymmärtämistä.

Tyypillisiä esimerkkejä tällaisista muutoksista ovat:

- virtuaalisten taulukoiden käyttö; taulukon läpikäyminen väärässä järjestyksessä saattaa tehdä ohjelman toimintakelvottomaksi
- parametrin välittäminen muuttuja- tai arvoparametrina; ohjelman tarvitsema muistitila saattaa laskea huomattavasti oikean valinnan mukaan
- muuttujan saaminen sopivalle muistirajalle, esimerkiksi Intel 8086 pohjaisissa koneissa sanatyyppinen muuttuja pitäisi alkaa parillisesta muistiosoitteesta

2.2.2 Kielen ominaisuuksien ymmärtäminen

Esimerkiksi C-kieli on jo hyvin koneenläheinen kieli. Siinä voidaan muuttujille tehdä konekielen tapaan loogisia operaatioita, käyttää suoraan osoittimia muistipaikkoihin ja suorittaa laskutoimituksia osoitteilla. Näiden operaatioiden ymmärtämiseksi on hyvä ymmärtää myös niiden konekieliset vastikkeet.

2.2.3 Ohjelman toiminnan nopeuttaminen

Vaikka nykyisin on saatavana eri kielille hyvinkin optimoivia kääntäjiä, tulee joskus erikoistilanteita, joissa taitava ohjelmoija pystyy tekemään nopeampaa konekielistä koodia kuin optimoiva kääntäjä. Tällöin ohjelmoija kirjoittaa aikakriittisimmän ohjelman osan suoraan konekieliseksi aliohjelmaksi. Joihinkin ohjelmointikieliin voidaan

INLINE-komennolla laittaa suoraan konekielisiä käskyjä normaalin ohjelmakoodin sekaan.

Keskeytyspalvelijoiden (interrupt server = koodi, joka suoritetaan keskeytyspyynnön tultua) tulee olla nopeita jotta muuta koneen toimintaa häirittäisiin mahdollisimman vähän.

Graafisten perusrutiinien, kuten pisteen, viivan ja ympyrän piirron, täytyy olla hyvin nopeita jotta valmis CAD- (Computer Aided Design) tai vastaava graafinen ohjelma olisi riittävän nopea käyttää. Nämä joudutaan lähes poikkeuksetta ohjelmoimaan konekielellä. Onneksi kuitenkin rutiinit saadaan usein valmiina kirjastoaliohjelmina eri laitetyypeille.

2.2.4 Tarvittavan muistitilan pienentäminen

Joissakin tapauksissa kääntäjä kääntää suhteellisen monta konekielistä käskyä toimenpidesarjaan, joka todellisuudessa menisi muutamalla, ehkä jopa yhdellä, konekielisellä käskyllä. Esimerkiksi MS-DOSin muistiin jäävissä (TSR, Terminate and Stay Resident) ohjelmissa on ohjelman pieni koko erittäin tärkeää.

Myös ohjelman käynnistymisaika (latautumisaika) on lähes suoraan verrannollinen ohjelmakoodin pituuteen. Erityisesti levykekoneissa isojen ohjelmien lataaminen tapahtuu tuskastuttavan hitaasti.

Joissakin käyttöjärjestelmissä ohjelmia voidaan käynnistää toisen ohjelman sisältä. Tällöin myös käynnistävä ohjelma jää muistiin minkä lisäksi muistiin tulee käynnistettävä ohjelma eli muisti saattaa loppua. Vaikka useissa koneissa käytetäänkin virtuaalista muistia, pitenee ohjelman suoritusaika aina huomattavasti, mikäli virtuaalisivunvaihtoja joudutaan tekemään.

Mikroprosessoripohjaisissa ohjausjärjestelmissä ohjelmakoodi on usein sijoitettu ROM (tai EPROM) muistiin, ja käytössä olevan muistin koko saattaa olla hyvinkin rajallinen.

Muistin halvennuttua on sitä usein paljonkin käytössä, mutta vastaavasti ohjelmien koot ovat kasvaneet ja näyttääkin siltä, että muistipula on pysyvä ilmiö. Pienen muistin aikana tehtiin hienoja konekielisiä ohjelmia. Esimerkkinä mainittakoon Turbo Pascal 3.0 -kääntäjä, jossa ohjelmaeditori ja Pascal-kääntäjä oli saatu alle 40 kilotavuun. Myös kotimainen TEKO III -tekstinkäsittelyohjelma on aivan kilpailukykyinen muiden tekstinkäsittelyohjelmien rinnalla, ohjelman koko on kuitenkin vain 45 kB (WordPerfect 4.2 270 kB, tosin ominaisuuksia on enemmän).

2.2.5 Tehtävää ei voida muuten suorittaa

Valitulla kielellä ei ehkä voida edes suorittaa jotakin tarvittavaa tehtävää. Tällöin joudutaan jälleen kirjoittamaan korkeamman tason kielen rinnalla konekielistä koodia. Tällaisia ongelmia tulee eteen tosin lähinnä systeemiluontoisten ohjelmien yhteydessä.

Konekielellä voi tehdä mitä osaa, muilla kielillä sen mitä niillä voi!

2.3 Haittapuolet

2.3.1 Vaikeus?

Eräänä konekielisen ohjelmoinnin haittapuolena pidetään sen vaikeutta. Tämä ei tosin aivan pidä paikkaansa; mikäli tarvitsisi osata vain yksi konekieli, ei sillä ohjelmoimisessa ole vaikeuksia. Käytännössä kuitenkin prosessorit vaihtuvat jatkuvasti, samoin siis niiden konekieli. Tällöin joudutaan eri prosessoria varten opettelemaan uusi kieli. Tosin nykyprosessoreissa konekielet muistuttavat niin paljon toisiaan, että uuden kielen omaksuminen ei ole ylivoimainen tehtävä.

2.3.2 Laiteriippuvuus

Koska konekieli on prosessoririippuvaista, ei yhdelle prosessorille kirjoitettu ohjelma toimi toisessa prosessorissa. Lisäksi konekielellä voidaan käsitellä suoraan muitakin laitteiston oheispiirejä, joten sama ohjelma ei välttämättä toimi muissakaan pelkästään samalla prosessorilla varustetuissa laitteistoissa; myös oheispiirien pitää olla täsmälleen samoja.

Pitää tietenkin muistaa, että millä tahansa kielellä (ei tulkittavat kielet) kirjoitetut ohjelmat käännetään konekielelle, jolloin käännetty binääritiedosto toimii vain täsmälleen saman yhteensopivan laitteiston alaisuudessa. Korkeamman tason kielellä kirjoitettu ohjelma voidaan kuitenkin kääntää uudestaan toisessa ympäristössä ja tällä tavalla saadaan ohjelmat laiteriippumattomiksi.

Prosessorikohtaista riippuvuutta on osin koetettu poistaa RISC-prosessoreiden luokkaan (Reduced Instruction Set Computer) kuuluvassa SPARC-arkkitehtuurissa (Scalable Processor Architecture, esim. SUN 4). SPARCissa on määritelty prosessoreiden rekisterirakenne ja käskykanta, mutta fyysinen toteutus ja rekistereiden todellinen lukumäärä on valmistajan itse päätettävissä.

Joidenkin prosessoreiden (esim. 8086) ympärille on kehitetty niin paljon ohjelmia, että ohjelmistovalmistajat ovat tehneet kyseisiä prosessoreita simuloivia ohjelmia. Näin toisellakin prosessorilla voidaan ajaa toisen prosessorin konekielisiä ohjelmia. Esimerkkinä SoftPC-ohjelma (valmistaja Insignia Solutions, käytetään esim. Apple MacIntosh IIX:ssä), jolla voidaan ajaa PC-yhteensopivissa koneissa toimivia ohjelmia.

On myös kääntäjäohjelmia, jotka muuttavat toisen prosessorin konekieltä toisen prosessorin konekieleksi. Yksi tällainen ympäristö on XDOS (Hunter Systems), jossa 8086 prosessorin käskyjä muutetaan esim. Motorolan 68020-prosessorin konekäskyiksi. Tällainen tuote on tietysti edellä mainittua simulointia nopeampi, mutta täytyy nostaa hattua ohjelman tekijälle, mikäli kääntäjä selviää myös itseään muuttavista 8086-kielisistä ohjelmista.

Kun prosessorivalmistaja tekee uuden prosessorin, on se yleensä alaspäin yhteensopiva saman valmistajan vanhemman prosessorin kanssa (esim. Intel 8086-sarja, Motorola 68000-sarja). Näin ollen vanhalle prosessorille tehtyjä ohjelmia voidaan ajaa uudemmallakin prosessorilla. Tällöin uuden prosessorin ominaisuudet eivät ehkä kuitenkaan tule täysin hyödynnetyksi (vrt. 8086 ja 80386).

2.3.3 Ohjelmoinnin hitaus

Konekielellä ohjelmoiminen on usein paljon vastaavaa korkeamman tason kielellä ohjelmoimista hitaampaa. Tosin hyvällä makrokääntäjällä jatkuvasti konekieltä kirjoittava ohjelmoija voi päästä aivan tyydyttäviin tuloksiin esimerkiksi Pascal-kieleen verrattuna.

Lisäksi konekielisen ohjelman testaaminen on usein työläämpää kuin korkeamman tason kielellä tehdyn ohjelman testaaminen. Yksi puuttuva tai väärä konekielinen käsky saattaa helpostikin aiheuttaa piileviä vikoja, jotka paljastuvat vasta ohjelman ollessa tuotantokäytössä.

Tosin puuttuvat muuttujan alustukset aiheuttavat pahoja piileviä vikoja myös korkeamman tason kielillä tehtyihin ohjelmiin. Hyvillä analysointiohjelmilla ja kääntäjillä saadaan kuitenkin paljon korkeamman tason kielten ohjelmissa olevia virheitä karsituksi.

Laiteriippumattomuuden takia korkeamman tason kielten testausvälineiden kehittelyyn voidaan uhrata enemmän aikaa kuin jonkin tietyn prosessorin konekielen tarkistamiseen.

2.4 Yhteenveto

Tavallisen ohjelmoijan on hyvä tuntea konekielisen ohjelmoinnin perusteet. Itse konekielistä ohjelmointia ei pidä kuitenkaan lähteä tekemään suinpäin. Varsinainen ohjelman runko kannattaa lähes aina tehdä jollakin korkeamman tason kielellä. Mikäli ohjelma kirjoitetaan alunperin C-kielellä, ei siihen tarvitse ehkä edes lisätä konekielisiä aliohjelmia. Tosin C-kielen ominaisuuksien käyttö vaatii jo lähes konekielimäistä ajattelua.

Jopa konekieliseksi tarkoitetut rutiinit kannattaa ehkä aluksi tehdä korkeamman tason kielellä mikäli mahdollista. Mikäli valmis ohjelma osoittautuu hitaaksi, lähdetään analysoimaan ohjelman suoritusta. Tässä voi usein käyttää apuna jotakin monitorointiohjelmaa, jolla voidaan selvittää missä ohjelman osissa suoritusaika kuluu. Usein kriittiset kohdat voi löytää myös pöytätestaamalla tai muuten päättelemällä.

Kun kriittiset kohdat on löydetty kannattaa harkita riittääkö esimerkiksi C-kielinen aliohjelma ohjelmakoodin nopeuttamiseksi. Mikäli konekieltä päätetään käyttää, täytyy konekieliset ohjelmat dokumentoida vielä normaaliakin tarkemmin, jotta myöhemmin ohjelma olisi helposti kirjoitettavissa toiselle prosessoriperheelle.

2.5 Kääntäjän tekemä koodi

Tutkitaan seuraavaa Pascal-kielistä ohjelmaa:

```
PROGRAM eka;
VAR k,i,j:INTEGER;
BEGIN
  i:=4;
  j:=3;
  k:=i+j;
END.
```

Ohjelma ei selvästikään tee mitään näkyvää. Seuraavassa on levyn hakemisto EKA.* kun ohjelma on käännetty Turbo Pascal 3.0 (EKA.COM) ja Turbo Pascal 5.0 (EKA.EXE) kääntäjillä:

```
EKA      PAS       76   6.02.89  14.23
EKA      EXE     1328   6.02.89  14.24
EKA      COM    11455   6.02.89  14.25
```

Vastaavasta C-ohjelmasta:

```
int k,i,j;
main()
{
  i=4;
  j=3;
  k=i+j;
}
```

Turbo C 2.0 kääntää 2.3 kB - 5.5 kB koodia kääntäjän muistimallista ja optioista riippuen. Tosin C-kääntäjä varoittaa, että muuttujaa k ei käytetä koskaan.

Siis ohjelma, joka ei tee mitään muuta kuin käy muistissa ja muuttaa muutamaa muistipaikkaa, vie konekielisenä kääntäjästä riippuen 1328 - 11455 tavua muistia?

Kääntäjät ottavat ohjelmaan mukaan yleensä tarvittavia kirjastorutiineja. Edellisissä esimerkeissä ei mitään kirjastorutiineja tarvita, mutta siitä huolimatta ne tulevat mukaan. Käytännön ohjelmissa vastaavaa hukkatilaa ei tule suhteellisesti yhtä paljon.

Perehdytään nyt siihen, mitä kääntäjä joutuu tekemään muodostaakseen esimerkkiohjelmista toimivan ohjelman:

1. rivi ei merkitse Pascal-kääntäjälle oikeastaan muuta kuin ohjelman alkua.

Seuraavalla rivillä ilmoitetaan, että ohjelmassa halutaan käyttää kolmea kokonaisluvun kokoista muistipaikkaa, joille varataan nimet k,i ja j. Tällöin kääntäjä varaa aluksi tyhjästä datamuistialueesta kolme (usein peräkkäistä) 2 tavun aluetta, joista 1. viitataan nimellä k, seuraavaan i ja viimeiseen j.

Osoite	76543210	
003C	00000111	k bitit 7-0
003D	00000000	k bitit 15-8
003E	00000100	i bitit 7-0
003F	00000000	i bitit 15-8
0040	00000011	j bitit 7-0
0041	00000000	j bitit 15-8

Kun myöhemmin viitataan vaikkapa muuttujaan i, tarkoittaa tämä tosiasiassa sitä, että viitataan niihin kahteen muistiosoitteeseen (esimerkissä 003E ja 003F), jotka oli varattu muuttujalle i. Seuraavassa esimerkissä on Turbo Pascal 5.0:lla käännetystä koodista Turbo Debugger-ohjelmalla otettu olennainen osa konekieliseksi (Intel 8086):

```
konekieli      assembler                ; pascal
-------------------------------------------------
C7063E000400   mov    word ptr [I],0004  ; i:=4
C70640000300   mov    word ptr [J],0003  ; j:=3
A13E00         mov    ax,[I]             ; k:=i+j;
03064000       add    ax,[J]
A33C00         mov    [K],ax
```

Siis sijoitus `i:=4` on konekielisenä C7 06 3E 00 04 00. Tämä tarkoittaa, että laitetaan sanan kokoiseen (C7 06) i:lle varattuun muistipaikkaan 003E vakioluku 0004. Vastaavasti seuraavalla rivillä sijoitetaan j:lle varattuun muistipaikkaan 0040 vakioluku 0003.

A1 tarkoittaa sanan kokoisen (i:lle varatun 003E) muistipaikan siirtämistä prosessorin rekisteriin AX. 03 06 tarkoittaa sanankokoisen (j:lle varatun 0004) muistipaikan lisäämistä rekisteriin AX. Lopuksi A3 tarkoittaa rekisterin AX siirtämistä sanankokoiseen (k:lle varattuun 003C) muistipaikkaan.

Turbo C 2.0 kääntää vastaavasta ohjelman osasta täsmälleen saman konekielisen koodin muistipaikkojen osoitteita lukuunottamatta. Emme siis täällä kertaa puutu enempää C-kieliseen koodiin.

Edellinen esimerkki ei ole paras mahdollinen esimerkki siitä miten koodia voitaisiin parantaa kääntäjän tekemään koodiin verrattuna. Hyvä optimoiva kääntäjä kyllä saattaisi edellä huomata, että i ja j eivät ole sijoituksen jälkeen muuttuneet ja osaisi sijoittaa suoraan muistipaikkaan k luvun 7. No onneksi edellä ollut koodi oli lähes optimaalista, muutenhan emme voisi lainkaan luottaa kääntäjien tekemään koodiin, mikäli jo perustoimitukset kääntyisivät huonosti.

Yksinkertainen kääntäjä, kuten Turbo Pascal, käy käännettävän koodin vain kerran lävitse, joten käännöksen täytyy syntyä tarkasti siten, että jokainen Pascal-rivi vastaa täsmälleen tiettyä joukkoa konekielisiä käskyjä. Koodin optimoimiseksi lähdekoodi täytyisi käydä useita kertoja lävitse ja tällöin vastaavasti käännösaika kasvaisi.

Luku 3

8088 ja 8086-rakenne

3.1 iAPX86-perhe

3.1.1 **8086** (1978)

Mikropiirien valmistaja Intel julkaisi 70-luvun loppupuolella aidon 16 bitin mikroprosessorin 8086.

8086 prosessorissa on 20 bitin osoiteväylä, jolloin se siis pystyy osoittamaan 1 megatavun muistialueeseen. Muistiosoitus on segmentoitu 16 bitin offset osoitteen avulla. Tällöin yhden segmentin koko voi siis olla 64 kilotavua. Segmentit voivat mennä myös päällekkäin.

8086 prosessori on mm. Olivetti M24:ssä ja uudemmissa PC-klooneissa.

3.1.2 **8088** (1979)

Tuolloin vielä kalliiden muistihintojen takia julkaistiin myös yhteensopiva 8088 prosessori, joka on sisäisesti kuten 8086, mutta sen dataväylä on vain 8-bittinen. Tällaisista prosessoreista käytetään usein merkintää 16/8-bittinen prosessori.

8088 prosessorista tuli IBM PC mikrotietokoneen ja sen kloonien mukana nopeasti eräs kaikkein yleisimmistä prosessoreista.

8088 prosessorin suorituskyky muistinhaussa on periaatteessa vain puolet 8086 prosessorin suorituskyvystä. Käytännössä hyvin paljon käsiteltävästä datasta on 8 bittistä joten prosessoreiden suorituskyky ei ole aivan suoraan verrannollinen dataväylän leveyteen.

3.1.3 **80186** (1983)

8086 tarvitsee lisäkseen useita oheispiirejä, esimerkiksi piirinvalinta-, kello-, keskeytyskontrolleri- ja laskuripiirin. Yhteensä yleensä noin 15-20 eri komponenttia. Ohjauslaitteiden kontrollerilla tarvitaan tällöin paljon piirilevytilaa. Tämän vuoksi Intel julkaisi 80186 ja 80188 piirit, jotka vastaavat 8086 ja 8088 prosessoreita sekä niiden 10 yleisintä lisäpiiriä. Näiden avulla ohjausmikro voidaan toteuttaa yhdelle piirikortille.

Lisäksi 80186 prosessorin käskykantaa on hieman laajennettu 8086 prosessoriin verrattuna. Esimerkiksi koko rekisterirakenne voidaan tallettaa pinoon yhdellä PUSHA-käskyllä. Myös osoitteen muodostusta ja eräitä sisäisiä laskutoimituksia on nopeutettu (16 bitin kertolasku 8086: 128-154 kellokierrosta, 80186 34-37 kellokierrosta).

80186 prosessori on esimerkiksi kotimaisessa MikroMikko 2A:ssa. Itse asiassa MikroMikossa on kaksi 80186 prosessoria: toinen pääprosessorina ja toinen näytön ohjaimena.

3.1.4 **80286** (1983)

IBM PC/AT (1984) oli ensimmäinen yleistynyt mikrotietokone, jossa käytettiin Intelin uutta 80286 prosessoria. 80286 voi toimia kahdessa eri moodissa:

- **REAL MODE**, jossa 80286 toimii aivan kuten 8086 (tosin laajennetulla käskykannalla ja nopeutetulla osoitteen muodostuksella).
- **PROTECTED MODE**, jossa 80286 pystyy osoittamaan 24 bitin virtuaaliseen osoiteavaruuteen. Tätä moodia käytetään hyväksi UNIX- ja OS/2-käyttöjärjestelmissä. Normaalit MS-DOS-koneet eivät hyödy 80286:sta muutoin kuin sen nopeuden puolesta. Myös tässä tilassa on edelleen 64 kilotavun segmentit.

PC/AT-yhteensopivien mikrojen myötä 80286 prosessori on levinnyt hyvin laajalle.

3.1.5 **80386** (1985)

Intelin julkaistua aidon 32-bittisen prosessorinsa ei IBM tällä kertaa ehtinytkään julkaista ensimmäistä tämän prosessorin varaan tehtyä mikrotietokonetta. Kloonivalmistajien hetken odotellessa IBM:n julkistusta rupesivat eräät valmistajat tekemään koneita 'omin päin', etunenässä COMPAC. Ilman tien näyttäjää kukaan ei uskaltanut ottaa mullistavaa harppausta, vaan koneista tehtiin PC/AT-yhteensopivia nopeammalla prosessorilla.

80386 prosessori voi toimia useassa eri tilassa:

- **REAL 86 MODE** on kuten 80286 prosessorin vastaava nopea 8086-tila. Tosin myös 32-bittiset operaatiot ovat mahdollisia, mutta MS-DOS ei tue niitä.
- **16 BIT PROTECTED MODE** on sama kuin 80286:en protected mode. Ensimmäisissä OS/2:n versiossa 80386:sta käytettiin vain tässä tilassa.
- **32 BIT PROTECTED MODE (NATIVE MODE)** on tehokkain 80386:n tiloista. Siinä on käytössä 32 bitin offset-osoitteet ja tehokas moniajo. Jotkut UNIX toteutukset käyttävät jo tätä tilaa hyväkseen.
- **VIRTUAL 86 MODE** mahdollistaa useiden virtuaalisten 8086 prosessien suorittamisen. Tämän tila on parempi kuin OS/2:n Dos Compatibility Box, koska virtuaalisen prosessin kaatuminen ei kaada koko konetta kuten OS/2:ssa tapahtuu.

3.1.6 **80386SX** (1987)

Kaupallisista syistä ihmisille uskotellaan, että 16-bittinen prosessori on 32-bittistä halvempi. Tämän takia 80386 prosessorista valmistettiin 32/16-bittinen yhteensopiva versio 80386SX. Näin vanhojen 80386 koneiden hinta voidaan säilyttää ennallaan ja myydään 'halvemmalla' prosessorilla varustettuja koneita halvemmalla. Tosiasiassa materiaalikustannukset muodostavat vain pienen osan mikrotietokoneen hinnasta, johon prosessorin vaihto ei siis käytännössä merkitse mitään.

3.1.7 **80486** (1989)

Keväällä 1989 julkaistiin 80386 piirin seuraaja 80486. Uudessa piirissä on käskyjä tehostettu siten, että samallakin kellotaajuudella toiminta on 3-4 kertaa nopeampaa kuin 80386-piirillä. Lisäksi aritmetiikkaprosessori ja käteismuisti on integroitu samalle sirulle.

Prosessoriin pohjautuvia mikroja julkistettiin jo loppukesästä muutamien valmistajien toimesta, mutta alkuperäisessä prosessorissa olevien virheiden takia tuotantoversiota saatiin ulos vasta alkuvuodesta 1990. Näin ollen lähes kaikki mikrovalmistajat ehtivät tällä kertaa samanaikaisesti mukaan uuden prosessorin käytössä.

3.1.8 Klooniprosessorit

Kuten yleensäkin menestyville tuotteille ilmestyi myös iAPX-perheelle kloonivalmistajia. Näistä tunnetuimpia on NEC prosessoreillaan V20 (8088), V30 (8086) ja V40. Esimerkiksi NECin prosessorit ovat hieman alkuperäisiä suorituskykyisempiä.

3.1.9 Prosessorin nopeus

Prosessorin suorituskyvyn ratkaisee väylän leveyden lisäksi käytettävä kellotaajuus. Kellotaajuus määrää yhteen kellokierrokseen kuluvan ajan. 8088 prosessorin käskyjen suoritusaika vaihtelee yksinkertaisten käskyjen kahdesta kellokierroksesta jakolaskun 180 kellokierrokseen. Siis kellotaajuutta nostamalla prosessorin teho kasvaa.

Alkuperäisiä 8088 prosessoreita ajettiin 4.77 MHz taajuudella. Uusissa 80386 koneissa käytetään jo 33 MHz taajuutta.

Myös oheislaitteiden nopeus vaikuttaa laitteistojen suorituskykyyn. Suurestakaan kellotaajuudesta ei ole apua, mikäli käytetään hitaita muistipiirejä. Odottaessaan muistin saantiajan, joutuu prosessori tekemään tyhjiä kellokierroksia. Tällöin puhutaan wait state-tiloista, joita tyypillisesti on 0-3.

Eri arkkitehtuurilla toteutettujen prosessoreiden suorituskyvyssä on myös eroja. Toisen arkkitehtuurin prosessori saattaa samallakin kellotaajuudella olla huomattavasti nopeampi. Esimerkiksi RISC-prosessoreissa on pyritty optimoimaan kaikkein tärkeimpien käskyjen suorituksen nopeutta.

3.1.10 Apuprosessorit

iAPX-perheen prosessoreihin voidaan liittää useita apuprosessoreita, jotka toimivat synkronissa pääprosessorin kanssa. Tunnetuimpia apuprosessoreita ovat aritmetiikkaprosessorit: 8087, 80287 ja 80387.

8087 suorittaa esimerkiksi 5 MHz:n kellolla varustetussa koneessa reaalilukujen kertolaskun 19 mikrosekunnissa 8086 prosessorin 1600 mikrosekunnin sijaan. e^x:llä vastaavat ajat ovat 100 ja 17100 mikrosekuntia. Ero on erittäin tuntuva erityisesti graafisissa sovelluksissa (esim. CAD) sekä myös taulukkolaskennassa. Lisäksi laskentatarkkuus paranee 80 bitin sisäisen esityksen myötä.

Käytännössä ero ei tosin ole aivan näin suuri, koska aritmetiikkaprosessorille täytyy myös siirtää luvut ja tämäkin vie aikaa. Esimerkiksi 4.77 MHz:n 8088-prosessorilla varustetussa koneessa 1000 e^x:n laskeminen kestää 15 sekuntia ilman apuprosessoria ja 0.6 sekuntia 8087-prosessorin kanssa (mittauksessa tosin osa ajasta tulee silmukan kiertämisestä, jota ei ole vähennetty).

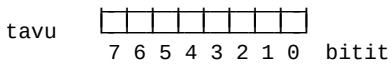
8087-sarjan aritmetiikkaprosessorit ovat pinoprosessoreita, joissa tulos lasketaan pinossa olevien lukujen kesken ja tulos palautetaan pinoon. Laskennassa käytetään sisäistä 80 bitin esitystä. Operaatiot ovat kertolaskusta neliöjuureen, tangenttiin ja eksponenttifunktioihin.

Muita iAPX-sarjan apuprosessoreita on esimerkiksi 8089 I/O-prosessori.

3.2 Muisti

3.2.1 8-bitinen muisti

Loogisesti 8088 ja 8086 prosessoreiden muisti koostuu tavuista. Tavun pituus on kahdeksan bittiä.



Itse asiassa RAM-muistissa on vielä 9. bitti/tavu, joka on pariteettibitti. Ohjelmoijan kannalta tätä bittiä ei ole, mutta muistinhallintayksikkö antaa prosessorille NMI-keskeytyksen (Non Maskable Interrupt), mikäli muistissa tapahtuu pariteettivirhe. Lisäksi tämä on muistettava muistipiirejä hankittaessa. 256 kilotavua lisämuistia saadaan seuraavasti: 9 kappaletta 256 kilobitin muistipiirejä asennetaan rinnan, jolloin muodostuu 256 kilotavua (8 bittiä) pariteettitarkistettua (9. bitti) muistia.

Koska prosessoreiden osoiteväylä on 20-bittinen, voidaan osoittaa 1 megatavu muistia.

3.2.2 8088:n ja 8086:n fyysinen ero

Fyysisesti 8086:en muisti on 16-bittistä ja 20 bitin osoiteväylästä käytetään vain 19 ylintä bittiä. Siis vain 512 kilosanaa muistia pystytään osoittamaan. Prosessoriin on kuitenkin järjestetty mahdollisuus osoittaa 512 kilosanan lohkosta joko parillista tai paritonta 512 kilotavun lohkoa.

8088-prosessorissa muisti on myös fyysisesti 8-bittistä ja kaikki 20 osoitelinjaa käytetään osoitteen muodostamiseen.

	8086 muisti odd || even	hex
osoite	5432109876543210	
00000	0101100101101111	596F
00002	1010011101010110	A756
00004	0111110110101010	7DAA
00006	1010111010101010	AEAA
00008	1010010101010001	A551

	8088	hex
osoite	76543210	
00000	01101111	6F
00001	01011001	59
00002	01010110	56
00003	01111101	7D
00004	10101010	AA
00005	01111101	7D
00006	10101010	AA
00007	10101110	AE
00008	01010001	51
00009	10100101	A5

Edellisessä esimerkissä on kummankin prosessorin 10 ensimmäistä muistipaikkaa. Esimerkiksi 8088 prosessorin muistipaikasta 00004 alkava sana 7DAA on myös 8086 prosessorilla sanana muistipaikassa 00004. Kuitenkin muistipaikasta 00007 alkava sana 51AE on 8086 prosessorilla muistissa seuraavasti. Sanan alkuosa AE muistipaikan 00006 parittomassa puoliskossa (bitit 8-15) ja sanan loppuosa 51 muistipaikan 00008 parillisessa (even) puoliskossa.

Esimerkissä muistin sisältö on kirjoitettu binäärisenä, jotta lukija varmasti muistaa tietokoneen muistin olevan binääristä. Jatkossa lukemisen helpottamiseksi kirjoitamme kuitenkin muistin sisällön heksadesimaalisena. Pitää kuitenkin muistaa että heksaesitys on vain lyhenne binääriesitykselle!

Noudettaessa tavuja muistista aktivoi 8086 joko vastaavan osoitteen parillisen (even) tai parittoman (odd) puoliskon (osoiteväylän bitillä 0) sekä erityisellä BHE-signaalilla (Bus High Enable) tiedon siitä että noudettava tieto on tavu. Sanan nouto parillisesta osoitteesta tapahtuu aktivoimalla parillinen puolisko ja BHE signaalilla ilmoitetaan, että noudetaan sana. Parittomasta osoitteesta sana täytyy hakea kahtena tavun noutamisena. Sanan noutaminen parittomasta osoitteesta vie siis kaksinkertaisen ajan verrattuna sanan noutamiseen parillisesta osoitteesta.

8088-prosessorilla noudetaan tavut tavuina ja sanat aina kahtena tavun noutona. Hakuaika on siis riippumaton sanan osoitteesta (edellyttäen, että muistipiirit ovat samaa teknologiaa kussakin osoitteessa!).

Itse noutamistekniikasta ei ohjelmoijan tarvitse normaalisti välittää yhtään mitään, loogisesti muisti voidaan siis ajatella tavujakoiseksi, jonka jokaisessa osoitteessa voidaan käsitellä joko sanoja tai tavuja.

Tekniikan seuraukset pitää kuitenkin muistaa. Laskettaessa 8086-prosessorilla tehdyn ohjelman suoritusaikoja, pitää jokaiseen parittomasta muistiosoitteesta alkavaan sanaoperaatioon lisätä 2 kellokierrosta. Erään MS-DOSin virheen takia voidaan tehdä levyä puskuroidusti lukeva ohjelma, joka kestää 5 kertaa kauemmin, mikäli levypuskuri alkaa parittomasta osoitteesta (suhteessa levyltä luettavan tavun järjestysnumeroon)!

Seuraavassa on esimerkki Turbo Pascal 5.0:lla tehdystä ohjelmasta joka kuvaa eroa aivan normaalissa sijoituslauseissa:

```
PROGRAM tavuraja;
{ Ohjelmalla demonstroidaan mitä vaikuttaa ohjelman suoritusaikaan     }
{ muuttujien sijoittuminen muistissa joko tavurajalle tai sanarajalle. }
{ Vesa Lappalainen 8.1.1989                                            }
USES timer;                  { Tapani Tarvaisen tekemä kello.          }
CONST n=400;                 { Suoritettavien kierrosten lukumäärä.    }
VAR ie,je,ke,le,me:INTEGER;  { Parillisista os. alkavat muuttujat.     }
    dummy:BYTE;              { Tavumuuttuja sanajaon sotkemiseksi.     }
    io,jo,ko,lo,mo:INTEGER;  { Parittomista os. alkavat muuttujat.     }
    t:time;                  { Kellonaika.                             }
BEGIN
  StartTimer(1);             { Aloitetaan ajanotto parillisille os.    }
  FOR je:=1 TO n DO FOR ie:=1 TO n DO BEGIN
    me:=le; le:=ke; ke:=je; me:=ie;
   END;
  StopTimer(1,t);            { Parillisten testisilmukka valmis.       }
  WRITELN('Parillinen osoite ',t,' sadasosasekuntia.');
  StartTimer(2);             { Aloitetaan ajanotto parittomille os.    }
  FOR jo:=1 TO n DO FOR io:=1 TO n DO BEGIN
    mo:=lo; lo:=ko; ko:=jo; mo:=io;
   END;
  StopTimer(2,t);            { Parittomien testisilmukka valmis.      }
  WRITELN('Pariton osoite    ',t,' sadasosasekuntia.');
END.
```

Ohjelmassa on muuttujalla dummy sotkettu muuttujien muistiin sijoittuminen. Ilman tätä muuttujaa kaikki muuttujat alkaisivat parillisesta osoitteesta. Nyt tavun kokoisen muuttujan jälkeen osoitteet alkavat parittomasta osoitteesta, edellyttäen että ohjelmaa käännettäessä on kääntäjän asetus ALIGN asennossa BYTE (asento WORD pakottaa jokaisen tavua suuremman muistikokonaisuuden alkamaan parillisesta osoitteesta). Tällöin saadaan seuraavat tulokset:

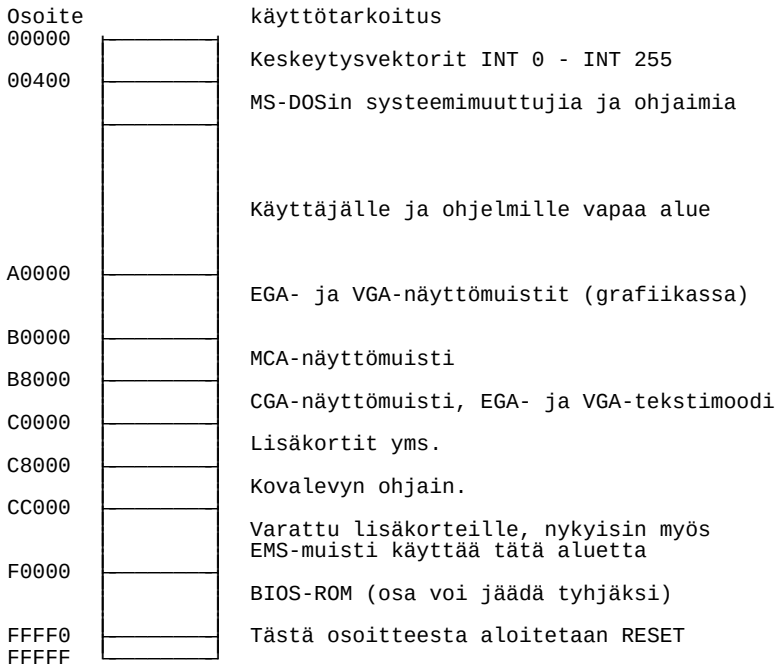
```
20 MHz 80386 kone (Osborne 386E)
  Parillinen osoite 77 sadasosasekuntia.
  Pariton osoite    117 sadasosasekuntia.
10 MHz 80286 kone (Osborne 6T):
  Parillinen osoite 176 sadasosasekuntia.
  Pariton osoite    230 sadasosasekuntia.
8 MHz 8086 kone (Olivetti M24):
  Parillinen osoite 330 sadasosasekuntia.
  Pariton osoite    440 sadasosasekuntia.
```

Esimerkissä ohjelman suoritusaika kasvaa noin 33% mikäli muuttujat alkavat parittomista osoitteista! 8088 prosessorilla varustetussa IBM PC:ssä molempien osien suoritusaika on sama.

Ohjelmoijan on siis pyrittävä huolehtimaan siitä, että sekä koodi että muuttujat sijoittuvat edullisesti muistissa. Tosin monesti tämä voidaan järjestää kääntäjän optioiden avulla (data alignment).

3.2.3 Muistin jako MS-DOS koneessa

MS-DOS -käyttöjärjestelmän alla prosessorin osoitettavissa oleva 1 megatavun muisti on jaettu seuraavasti:



3.3 Rekisterirakenne

8086 prosessorissa on 14 rekisteriä: 4 yleiskäyttöistä rekisteriä, kaksi indeksirekisteriä 4, segmenttirekisteriä, pino-osoitin, kantaosoitin, lippurekisteri ja ohjelmalaskuri.

3.3.1 Yleisrekisterit

8086-prosessorissa on 4 yleiskäyttöistä rekisteriä (AX, BX, CX ja DX) joita voidaan käyttää joko 16 bitin rekistereinä tai 8 bitin rekistereinä (AX 16 bittiä, AH ja AL 8 bittiä, High ja Low).

Neljällä yleiskäyttöisellä rekisterillä on lisäksi omat erikoistarkoituksensa:

- AX on akku, johon voidaan suorittaa kerto- ja jakolaskuja
- BX toimii kantaosoittimena
- CX toimii laskurina silmukoissa
- DX on apurekisteri kerto- ja jakolaskuissa

3.3.2 Indeksirekisterit

Kahta indeksirekisteriä (DI, Destination Index ja SI, Source Index) voidaan käyttää indeksoituun osoitukseen sekä merkkijono-operaatioihin.

3.3.3 Muut rekisterit

Kantaosoitinta (BP, Base Pointer) käytetään yleensä parametrin välitykseen. Muut rekisterit ovat pino-osoitin (SP, Stack Pointer), ohjelmalaskuri (IP, Instruction Pointer) sekä lippurekisteri (FL, Flags).

3.3.4 Segmenttirekisterit

Lisäksi on 4 segmenttirekisteriä: DS (Data Segment), ES (Extra Segment), SS (Stack Segment) ja CS (Code Segment). Segmenttirekistereitä käytetään hyväksi osoitteen muodostuksessa. Seuraavaksi suoritettava ohjelma-askel on aina osoitteessa CS:IP.

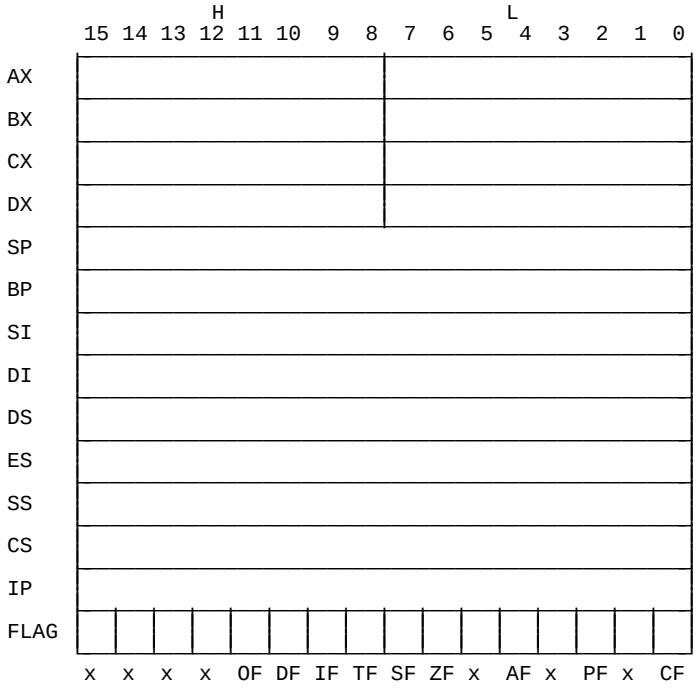
3.3.5 Liput

Lippurekisterin kukin bitti vastaa yhtä lippua.

Liput ovat:

- OF (Overflow Flag) ylivuoto
- DF (Direction Flag) merkkijono-operaation suunta
- IF (Interrupt Flag) keskeytyslippu
- TF (Trap Flag) yhden askeleen suorittamislippu
- SF (Sign Flag) etumerkkilippu
- ZF (Zero Flag) nollalippu, =1 jos tulos on 0,
- AF (Auxiliary carry Flag) välimuistinumero bitistä 3
- PF (Parity Flag) pariteettilippu, asetetaan sanaoperaatioissa vain 8 alimman bitin mukaan! =1 jos parillinen määrä 1 bittejä.
- CF (Carry Flag) muistinumerolippu

3.3.6 Rekisterit



3.4 Muistiosoitukset

8086-prosessorissa on useita eri muistinosoitustapoja. Tosin kaikki osoitustavat eivät ole käytössä kaikille rekistereille. Tätä on pidettävä eräänä prosessorin huonona puolena.

Kaikkiin osoitustapoihin liittyy lisäksi segmenttirekistereiden käyttö. Segmenttirekistereitä käsitellään toisaalla.

3.4.1 Välitön operandi

Rekisteriin tai muistipaikkaan voidaan siirtää vakioluku:

```
MOV  AX,5              ; Luku 5 rekisteriin AX
MOV  [003EH],5         ; Luku 5 muistipaikkaan 003E
```

3.4.2 Suora osoitus

Muistipaikan sisältö voidaan siirtää rekisteriin tai rekisterin sisältö muistipaikkaan:

```
MOV  AX,[003EH]        ; Rekisteriin AX muistipaikan 003E sisältö
MOV  [003EH],CX        ; Muistipaikkaan 003E rekisterin CX arvo
```

3.4.3 Indeksoitu osoitus

Indeksirekisterin avulla voidaan osoittaa lisäsiirtymä muistilohkon alusta:

```
MOV  AX,[003EH+SI]     ; AX:ään luku muistipaikasta 003E+SI
MOV  taulukko[DI],DX
```

Voidaan käyttää myös kahta indeksiä:

```
MOV  AX,[003EH+BX+SI]
MOV  taulukko[BX][SI],DX
```

XLAT-käskyä voidaan käyttää merkkimuunnoksissa. Käsky siirtää AL-rekisteriin tavun muistipaikasta [AL+BX].

3.4.4 Epäsuora osoitus

Epäsuoraa osoitusta voidaan käyttää vain hyppykäskyissä:

```
JMP  [003E]
```

Edellä on oikeastaan kyseessä suora osoitus:

```
MOV  IP,[003E]
```

jota tosin ei ole syntaktisesti oikein.

Muulloin epäsuora osoitus pitää tehdä kahdella käskyllä:

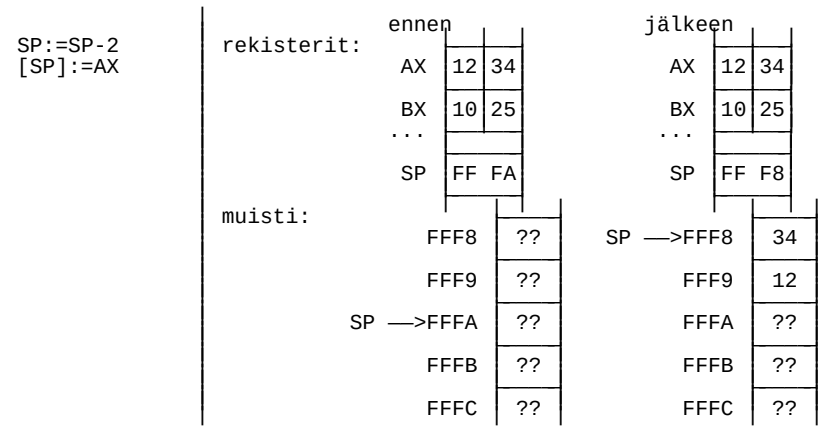
```
MOV  SI,[003E]
MOV  AX,[SI]
```

3.4.5 Pino-osoitus

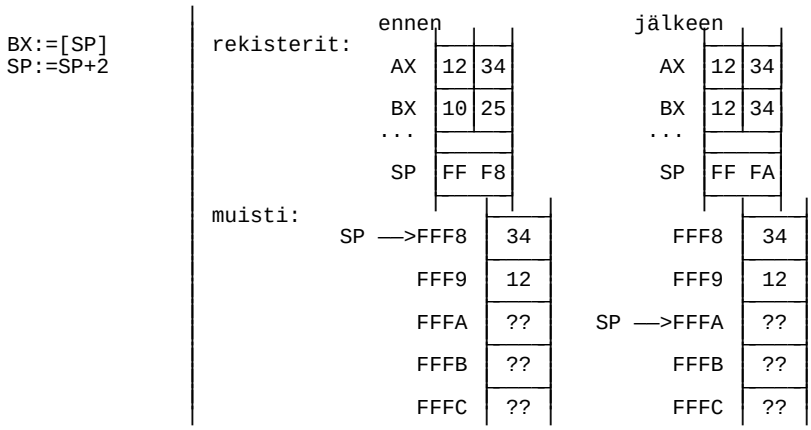
Pinoa käsitellään PUSH ja POP -käskyillä:

```
PUSH AX                 ; Rekisterin AX arvo pinoon
POP  DX                 ; Rekisteriin DX pinon pinnalla oleva arvo
```

Pino kasvaa pienempää osoitetta kohti, eli PUSH AX-käsky aiheuttaa seuraavat muutokset:



ja POP BX vastaavasti:



Myös aliohjelmakutsut ja RET-käsky käyttävät pinoa:

```
CALL 1234H    ==>  "IP:=IP+3   PUSH IP  IP:=1234"
RET           ==>  "POP IP"
```

Parametrin välityksessä BP-rekisteriä käytetään pinossa olevien parametrien osoittamiseen:

```
 MOV  AX,[BP+08]
```

3.4.6 Suhteellinen osoitus

Suhteellinen osoitus itse käskyn suhteen on mahdollista vain hyppykäskyille ja aliohjelmakutsuille:

```
 CALL laske
  ...
 JZ   liian_pieni
  ...
 JMP  lopeta_ohjelma
```

Tosin aliohjelmakutsun avulla voidaan rakentaa tarvittaessa suhteellinen osoitus:

```
 CALL seuraava
seuraava:
 POP  SI
```

Edellä `CALL`-käsky laittaa käskyä seuraavan osoitteen pinoon ja laittaa käskyssä olevan osoitteen ohjelmalaskurin arvoksi. Koska osoite on sama kuin seuraava käsky, jatkuu suoritus käskystä `POP SI`. Näin rekisteriin SI on saatu **suoritusaikainen** osoite nimiölle `seuraava`.

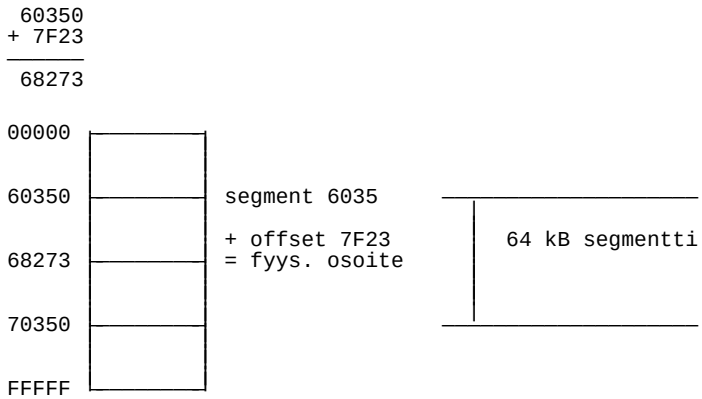
3.5 Segmentit

Eräs 8086-prosessorin parjatuista puolista on sen segmentoitu muistirakenne. Mikäli segmentit olisivat suurempia kuin 64kB, olisi niistä myös selvää etua. Näin on jo 80386-prosessorissa, jonka vaihtuvan kokoisia segmenttejä voidaankin pitää parempana kuin joidenkin prosessoreiden kiinteän mittaisia virtuaalisivuja.

Kuten edellä todettiin, on 8086 osoiteväylä 20-bittinen. Toisaalta kaikki rekisterit ovat 16-bittisiä. Fyysisessä muistiosoiteessa on kaksi osaa:

- SEGMENT joka osoittaa 64 kilotavun muistilohkon alun. Segmentti osoittaa aina johonkin 16:lla jaolliseen osoitteeseen, joten segmentille riittää varata 16 bittiä (ts. 4 alimmaista 0 bittiä jätetään merkitsemättä).
- OFFSET joka osoittaa siirtymän segmentin alusta

Fyysinen muistiosoite on siis segment+offset. Esimerkiksi osoite, jonka segment on 6035H ja offset on 7F23H on fyysisenä osoitteena



Koska segmenttiosoite ja offset-osoite osittain menevät päällekäin, ei tietylle fyysiselle osoitteelle ole yksikäsitteistä segment+offset -esitystapaa. Edellisen esimerkin osoite voitaisiin esittää mm. muodoissa:

```
segment  offset
 6827     0003
 6820     0073
 6800     0273
 6111     7163
 6000     8273
```

Tämä ei-yksikäsitteisyys aiheuttaa tiettyjä ongelmia mm. osoitteiden vertailuissa (esim. Pascalin pointer-tyypit). Osoite voidaan tietysti aina normeerata vaikkapa edellisen esimerkin ensimmäiseen muotoon, jossa osoitteesta siirretään mahdollisimman suuri osa segment-osaan.

Tehtävä 3.1 Osoitteen eri esitystavat

Kirjoita 10 erilaista esitystä osoitteelle 0175:1234.

Neljällä segmenttirekisterillä on kullakin oma erikoistarkoituksensa.

3.5.1 Data Segment, DS

Datasegmenttiä käytetään oletuksena kaikissa dataviittauksissa pinoon kohdistuvia viittauksia lukuunottamatta. Ainoan poikkeuksen tekee merkkijono-operaatiot, joihin palataan myöhemmin.

Jos esimerkiksi rekisterit ovat

```
BX = 0123
DI = 0E00
SI = 7F23
DS = 6035
```

niin silloin seuraavissa käskyissä ladataan rekisteriin AX muistipaikassa 68273H oleva sana:

```
MOV  AX,[7F23H]
MOV  AX,[SI]
MOV  AX,[7000H+BX+DI]
LODSW
```

Viimeinen käsky on merkkijono-operaatio, jossa AX:ään ladataan muistipaikassa DS:[SI] oleva sana. Latauksen jälkeen SI lisääntyy tai vähenee kahdella riippuen DF-lipun asennosta.

3.5.2 Stack Segment, SS

Pinosegmenttiä käytetään kaikissa pinoon kohdistuvissa operaatioissa. Näitä ovat esimerkiksi:

```
MOV  AX,[BP+06H]
CALL laske_erotus
PUSH AX
POP  DS
RET
```

3.5.3 Code Segment, CS

Seuraavaksi suoritettava ohjelma-askel otetaan aina paikasta CS:[IP]. Tarvittaessa voidaan dataakin ottaa koodisegmentistä esimerkiksi käskyllä:

```
MOV  AX,CS:[7F23H]
```

Koodisegmenttiin voidaan myös kirjoittaa dataa, mutta tätä on vältettävä, koska tällaiset ohjelmat eivät toimi esimerkiksi 80286-prosessorin protected modessa, eivätkä näin ollen OS/2-käyttöjärjestelmässä.

3.5.4 Extra Segment, ES

Lisäsegmentti ES:n ainoa oletuskäyttö on merkkijono-operaatioissa kohdesegmenttinä. Näitä käskyjä ovat

```
STOSB   ; AL paikkaan ES:[DI], DI:=DI+/-1
MOVSB   ; tavu DS:[SI] paikkaan ES:[DI], SI ja DI etenevät
CMPSB   ; verrataan DS:[SI] ja ES:[DI]:ssä olevia tavuja
        ; SI ja DI etenevät
SCASB   ; verrataan ES:[DI]:ssä olevaa tavua ja AL:ää
        ; DI etenee
```

Kaikki käskyt voivat olla myös W-loppuisia, jolloin toiminta tapahtuu sanoille ja tarvittaessa AX-rekisterille. Etenemissuunta riippuu suuntalipun DF asennosta.

Muulloin lisäsegmentin käyttötarve pitää ilmoittaa erikseen (segment override).

3.5.5 Muun kuin oletussegmentin käyttö

Tarvittaessa voidaan erikseen ilmoittaa (segment override), mikäli operaatioon halutaan käyttää muuta kuin oletussegmenttiä. Koodia voidaan kuitenkin suorittaa ainoastaan koodisegmentistä. Segmentin tilapäinen vaihto tehdään segment override -käskyllä, joka assembler-kielissä yleensä ilmaistaan kirjoittamalla käytettävän segmentin nimi osoitteen eteen. Esimerkiksi

```
MOV  AX,ES:[SI]
MOV  AX,CS:[0029H]
```

Pitää kuitenkin muistaa, että todellisuudessa segmentin vaihto on oma konekielinen käskynsä, joka vie 2 kellokierrosta lisää. Mikäli usein tarvitaan muuta kuin oletussegmenttiä, kannattaa segmentin arvo siirtää siihen segmenttirekisteriin, jossa sitä oletuksena käytettäisiin. Alkuperäinen arvo talletetaan esimerkiksi pinoon käyttämisen ajaksi. Tosin 8086-prosessorissa ei ole käskyä segmenttirekistereiden välisille siirroille, joten siirto täytyy tehdä jotain muuta kautta.

Olkoon seuraavassa esimerkissä ES-segmentissä dataa, jota tarvitsee käsitellä paljon:

```
PUSH DS       ; talletetaan DS:än alkuperäinen arvo pinoon
MOV  AX,ES    ; siirretään ES DS:ään AX:än kautta
MOV  DS,AX
MOV  AX,[SI]  ; nyt AX:ään tulee muodollisesti ES:[SI]
 ...          ; muita alkup. ES:ään kohd. käskyjä
POP  DS       ; palautetaan alkuperäinen DS paikalleen
```

Edellä sijoitus DS:=ES voitaisiin hoitaa myös pinon kautta ilman apurekisteriä AX, mutta tämä olisi hieman hitaanpaa:

```
PUSH DS       ; talletetaan DS:än alkuperäinen arvo pinoon
PUSH ES       ; siirretään ES DS:ään pinon kautta
POP  DS
MOV  AX,[SI]  ; nyt AX:ään tulee muodollisesti ES:[SI]
 ...          ; muita alkup. ES:ään kohd. käskyjä
POP  DS       ; palautetaan alkuperäinen DS paikalleen
```

8086-prosessorin segmenttirekisterit riittävät juuri datan siirtämiseen paikasta toiseen (lähde DS, kohde ES), mutta eivät esimerkiksi datan lomitukseen. Lomituksessa (merge) joudutaan käyttämään apurekisteriä, jossa esimerkiksi lähdesegmenttejä vaihdellaan keskenään:

```
; Seuraavassa ohjelmassa lomitetaan CX:än (>0) mittaiset
; muistilohkot DS:[SI] ja DX:[BX] yhdeksi muistilohkoksi
; alkaen paikasta ES:[DI]
 CLD             ; SI:in ja DI:n kasvatukset ylöspäin
lomita:          ; silmukan jatkamiskohta
 MOVSB           ; ES:[DI] := DS:[SI] ja SI,DI +1
 MOV  AX,DS      ; vaihdetaan DX ja DS AX:än kautta
 XCHG DX,AX      ;  tämä vaihtaa DX:än ja AX:än keskenään
 MOV  DS,AX      :  nyt DS:ssä on alkup. DX
 MOV  AL,[BX]    ; AL:=DX:[BX]
 INC  BX         ; BX:=BX+1
 STOSB           ; ES:[DI] = AL ja DI:=DI+1;
 MOV  AX,DS      ; vaihdetaan DS takaisin alkuperäiseen
 XCHG DX,AX
 MOV  DS,AX
 LOOP lomita     ; CX:=CX-1 ja jatketaan lomita kohdasta,
                 ;  mikäli CX<>0.
```

Edellä vaihto DS <-> DX voitaisiin suorittaa myös pinon kautta:

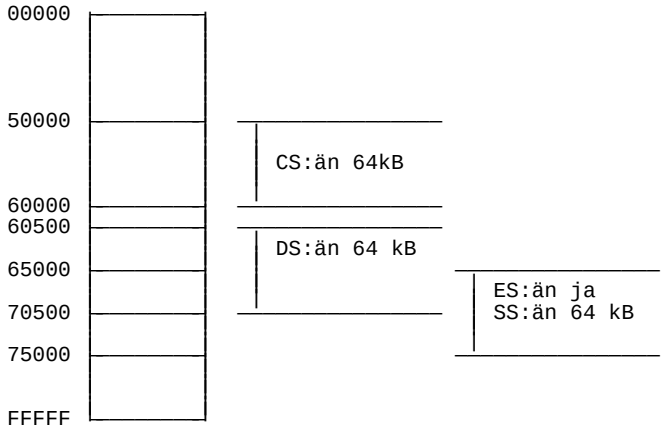
```
 PUSH DS
 MOV  DS,DX
  ...          ; operaatiot
 POP  DS
```

3.5.6 Segmenttien päällekäisyys

Segmentit eivät ole mitenkään toisiaan poissulkevia alueita kuten joidenkin prosessoreiden sivutus. Segmentit voivat mennä joko kokonaan päällekäin, olla osittain toistensa päällä tai olla kokonaan erillisiä. Olkoon segmenttirekistereillä seuraavat arvot:

```
DS = 6050
ES = 6500
SS = 6500
CS = 5000
```

Tällöin segmenttien osoittamat muistialueet jakautuvat seuraavasti:



Siis CS:än muistialue on erillään muista segmenteistä. DS:än loppuosan päällä on ES:än ja SS:än yhteinen muistialue.

3.6 Konekielisen käskyn koodaus

Seuraavassa käytetään esimerkkinä MOV-käskyn koodaamista. Vastaavasti koodataan esimerkiksi ADD-käsky.

3.6.1 Rekisteri-muisti -siirrot

Siirrettäessä rekisteristä muistiin tai muistista rekisteriin on konekielisen käskyn muoto seuraava:

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
1	0	0	0	1	0	d	w	mod		reg			r/m		

Vasemmanpuoleisessa tavussa on kaksi bittiä, joita voidaan muuttaa:

- d (direction) siirron suunta:
  0 = rekisteristä muistiin
  1 = muistista rekisteriin
- w (word size) siirron koko:
  0 = tavu (8 bittiä)
  1 = sana (16 bittiä)

Oikeanpuoleinen tavu ilmaisee mistä siirretään ja mihin rekisteriin. Siirtoon liittyy usein lisäksi siirtymä (displacement), joka lyhennetään jatkossa **disp**. Tarvittaessa disp-tavu tai -tavut seuraavat näitä kahta tavua.

Kenttien tulkinnat:

reg-kenttä 3 bittiä, lähde-/kohderekisteri:

reg	w=1	w=0	Seg
000	AX	AL	ES
001	CX	CL	CS
010	DX	DL	SS
011	BX	BL	DS
100	SP	AH	
101	BP	CH	
110	SI	DH	
111	DI	BH	

mod-kenttä 2 bittiä, disp-osan koko:

mod	DISP
00	DISP = 0, eli DISP tavuja ei ole, paitsi jos r/m = 110, jolloin osoite on seuraava sana
01	DISP = seuraava tavu laajennettuna 16 bittiseksi
10	DISP = seuraava sana
11	r/m-kenttä tulkitaan reg-kentäksi

r/m-kenttä 3 bittiä, indeksointi:

r/m	indeksointi
000	[BX+SI+DISP]
001	[BX+DI+DISP]
010	[BP+SI+DISP]
011	[BP+DI+DISP]
100	[SI+DISP]
101	[DI+DISP]
110	[BP+DISP], tai [DISP] jos mod=00
111	[BX+DISP]

Esimerkkejä:

konekieli	assembler	d w	mod reg r/m	disp
89D3	MOV BX,DX	0 1	11 010 011	
88D3	MOV BL,DL	0 0	11 010 011	
8A163412	MOV DL,[1234]	1 0	00 010 110	
89163412	MOV [1234],DX	0 1	00 010 110	1234H
8B163412	MOV DX,[1234]	1 1	00 010 110	1234H
892F	MOV [BX],BP	0 1	00 101 111	0000H
891E1200	MOV [0012],BX	0 1	00 011 110	0012H
896F12	MOV [BX+12],BP	0 1	01 101 111	0012H
89AF1200	MOV [BX+0012],BP	0 1	10 101 111	0012H
896FEE	MOV [BX-12],BP	0 1	01 101 111	FFEEH
89AF3412	MOV [BX+1234],BP	0 1	10 101 111	1234H
89AFCCED	MOV [BX-1234],BP	0 1	10 101 111	EDCCH
89063412	MOV [1234],AX	0 1	00 000 110	1234H

Huomattakoon edellisestä esimerkistä, että jollekin käskylle ei välttämättä ole yksikäsitteistä esitystapaa. Esimerkkinä rivit 896F12 ja 89AF1200.

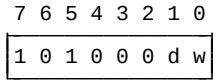
Tehtävä 3.2 Koodaus konekielelle

Kirjoita konekielisenä seuraavat käskyt:

```
MOV DX,BX
MOV DL,BL
MOV CL,[12]
MOV [12],CL
MOV SI,[12+SI+BX]
```

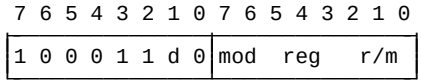
3.6.2 Siirrot akkuun tai akusta

Edellisen esimerkin viimeinen rivi `MOV [1234],AX` voitaisiin esittää (ja yleensä esitetäänkin) erityisesti akkuun kohdistuvalla käskyllä `A33412`. Akkuun kohdistuvan muistisiirron yleinen muoto on:



3.6.3 Segmenttirekisterit

Mikäli siirto kohdistuu segmenttirekisteriin, on konekielisen käskyn muoto:

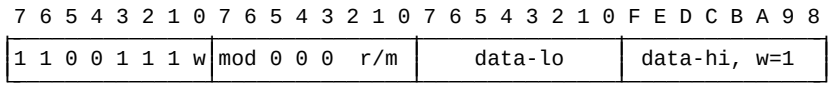


Edellä on kuitenkin MOV CS, -muotoon johtavat käskyt (d=1 ja reg=001) ovat määrittelemättömiä. reg-kentästä tulkitaan vain kaksi viimeistä bittiä.

konekieli	assembler	d w	mod reg r/m	disp
8E063412	MOV ES,[1234]	1 0	00 000 110	1234H
8C0F	MOV [BX],CS	0 0	00 001 111	0000H
8C163412	MOV [1234],SS	0 0	00 010 110	1234H
8CD8	MOV AX,DS	0 0	11 011 000	
8CF8	MOV AX,DS	0 0	11 111 000	

3.6.4 Välitön operandi muistiin tai rekisteriin

Välittömän operandin siirrolle on oma käskynsä, jonka muoto on:

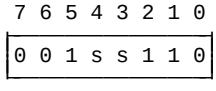


Mahdollisesti reg-kentässä olevat tavut jätetään tulkitsematta. Mikäli DISP-tavut tulevat, ovat ne ennen datatavuja.

Esimerkkejä:

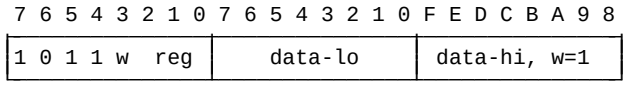
konekieli	assembler	d w	mod reg r/m	disp
C70634127856	MOV [1234],5678	1 1	00 000 110	1234H
C606341256	MOV [1234],56	1 0	00 000 110	1234H
C7075600	MOV [BX],0056	1 1	00 000 111	0000H
C7C35600	MOV BX,0056	1 1	11 000 011	
C7C17856	MOV CX,5678	1 1	11 000 001	
B97856	MOV CX,5678	1	001	

Esitysmuoto `MOV [1234],5678` tuottaa pisimmät käskyt (6 tavua) 8086 konekielessä. 7-tavuinen käsky (26C70634127856) tulee, mikäli käytetään vielä oletussegmentin vaihtoa `MOV ES:[1234],5678`. Segmentin vaihtokäskyn muoto on:



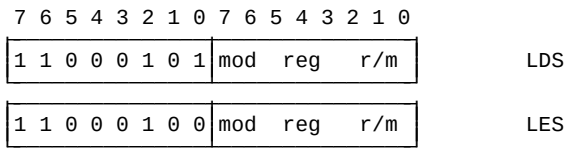
3.6.5 Välitön operandi rekisteriin

Edellisessä esimerkissä `MOV CX,5678` on esitetty kahdella tavalla. Jälkimmäinen tapa on erityisesti välittömän operandin rekisteriin laittamiseksi:



3.6.6 Osoittimen lataaminen

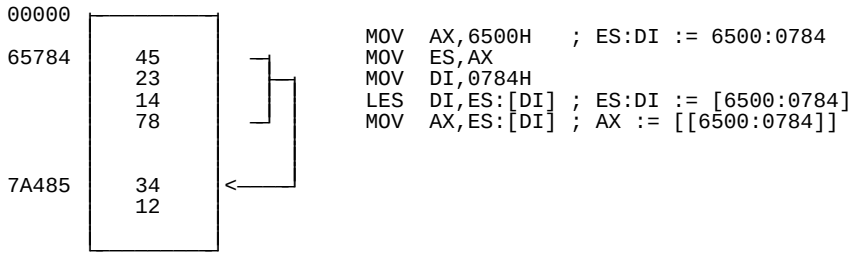
Prosessorissa on kaksi käskyä LDS ja LES joiden avulla voidaan ladata kokonainen osoite tarvitsematta ladata segmentti- ja offset-osaa erikseen. Käskyjen muoto:



Esimerkkejä:

konekieli	assembler	d w	mod reg r/m	disp
C57012	LDS SI,[BX+SI+12]		01 110 000	0012H
26C43D	LES DI,ES:[DI]		00 111 101	0000H

Viimeksi mainittua käskyä käytetään erityisesti epäsuoran osoituksen muodostamiseen, koska prosessorissa ei ole varsinaista epäsuoraa osoitusta. Seuraavassa esimerkissä haetaan epäsuoralla osoituksella rekisteriin AX muistipaikan 6500:0784 osoittamassa muistipaikassa oleva sana.



Käskyä käytetään myös joskus 4-tavuisen kokonaisluvun (long integer) lataamiseksi:

```
LES  AX,pitka_luku      ; DX AX := pitka_luku
MOV  DX,ES
```

3.7 Ohjelman suoritusaika

Kun konekielellä ruvetaan ohjelmoimaan, täytyy ohjelmoinnista olla jotakin hyötyä. Yksi hyöty on suoritusnopeuden kasvaminen. Jotta suoritusnopeutta voidaan kasvattaa, täytyy ohjelmoijan tietää kuinka kauan minkäkin käskyn suorittaminen tulee kestämään.

Eri käskyjen suoritusajat löytyvät prosessoreiden manuaaleista. 8086-prosessorissa on huomattava, että suoritusajat on annettu kellokierroksina. Ohjelmoijan riittää siis minimoida suoritukseen tarvittavien kellokierrosten lukumäärä. Nopein tapa tehdä jokin käsky ei aina välttämättä ole lyhin tai selvin (vrt. esimerkki *80 sivu 45). Tässä on pidettävä järki päässä ja vältettävä monimutkaisia yhden tai kahden kellokierroksen säästöjä ohjelman selvyyden tai pituuden kustannuksella.

3.7.1 Osoitteen laskeminen (EA)

8086 prosessorissa on ilmoitettu esimerkiksi MOV-käskylle eri suoritusaikoja riippuen mistä siirrosta on kyse. Rekisteri-rekisteri -siirron pituudeksi ilmoitetaan 2 kellokierrosta, kun taas rekisteri-muisti -siirron pituudeksi ilmoitetaan 8+EA kellokierrosta. Tässä EA tarkoittaa osoitteen laskemiseen kuluvaa aikaa (Effective Address calculation). Laskemiseen kuluva aika riippuu osoitustavasta.

Esimerkkejä:

```
MOV  BX,[1234H]           ; 8 + 6  (mem->reg)
MOV  AX,[1234H]           ; 10     (direct mem->ac)
MOV  AX,[SI]              ; 8 + 5
MOV  AX,[SI+5]            ; 8 + 9
MOV  AX,[SI+BX]           ; 8 + 7
MOV  AX,[SI+BX+13H]       ; 8 + 11
MOV  AX,ES:[BX+DI+14H]    ; 2 + 8 + 12
```

Viimeisessä lauseessa tulee 2 lisäkellokierrosta segmentin vaihdosta.

Huomattakoon, että 80186 ja sitä uudemmissa prosessoreissa osoitteenlaskemista on tehostettu ja EA:n laskemiseen kuluvaa aikaa ei tarvitse lisätä!

3.7.2 Muut tekijät

Edellisten lisäksi ohjelman suoritusaikaan vaikuttaa onko muistiosoite parillinen vai pariton sekä wait-state -tilojen määrä. Näiden tarkka laskeminen on usein varsin työlästä ja tämän takia teoriassa lasketut suoritusajat eivät aivan täsmää todellisen suoritusajan kanssa.

Ehdollisissa hypyissä suoritusaikaan vaikuttaa se tehdäänkö hyppy (16 kellokierrosta) vai jatketaanko seuraavasta lauseesta (4 kellokierrosta). Yleensäkin hypyt, aliohjelmakutsut ja paluut aliohjelmista sotkevat prosessorin sisäisen jonon, johon on ladattu valmiiksi seuraavaksi suoritettavia käskyjä.

Silmukoista täytyy ohjelman suoritusaikaa laskettaessa laskea jokin sopiva keskiarvo/kierros, jota käytetään jatkossa arviona silmukan kestolle.

Lisäksi suoritusaikaan vaikutaa ohjelman ajon aikana tulleiden keskeytysten lukumäärä. Luotettava ajoitus saataisiin vain mikäli kaikki keskeytykset kiellettäisiin.

3.7.3 Vakiotemppuja

Seuraavassa on esitetty eräitä vakiotemppuja jonkin tehtävän suorittamiseksi:

```
;                              toiminta           clocks
 MOV  AX,0                   ; AX:=0                4
 XOR  AX,AX                  ; AX:=0                3
 SUB  AX,AX                  ; AX:=0                3
 ADD  AX,AX                  ; AX:=2*AX             3
 SHL  AX,1                   ; AX:=2*AX             2
 SAR  AX,1                   ; AX:=AX DIV 2         2
 OR   AX,DX                  ; verrataan DX AX=0    2
 JZ   pitka_nolla            ; eli ed tulok. perust.
 INC  AX                     ; AX:=AX+1             2
 DEC  AX                     ; AX:=AX-1             2
 SHL  AX,1
 SHL  AX,1                   ; AX:=4*AX             4
```

Tehtävä 3.3 Suoritusaika

Laske seuraavan ohjelmanpätkän suorittamiseen kuluva aika kellokierroksina kun muistipaikassa 0003 on luku 61H. Kauanko suoritus kestäisi 10MHz 0-wait state 8086 koneessa?

```
 MOV  AL,[0003]
 CMP  AL,'a'
 JB   ei_muutosta
 CMP  AL,'z'
 JA   ei_muutosta
 SUB  AL,'a'-'A'
 MOV  [0003],AL
ei_muutosta:
```

3.8 Käskykanta

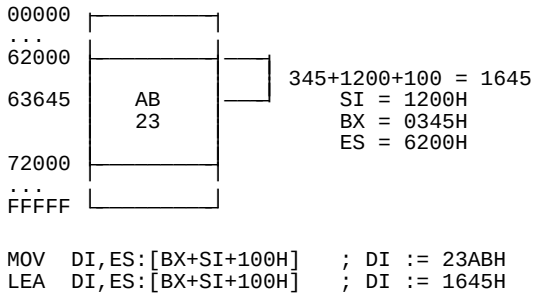
Tässä kappaleessa käsitellään edellisissä kappaleissa käsittelemättä jäänyttä 8086-prosessorin käskykannan osaa. Täydellinen käskykanta löytyy prosessoria käsittelevästä kirjallisuudesta.

3.8.1 Informaation siirto

Datan siirto ja muistin osoitustavat on käsitelty edellä jo varsin perusteellisesti. Näitä käskyjä on mm.

```
LDS,LES       ; osoittimen lataus
              ; LDS SI,[1234]   =  MOV  SI,[1234]  MOV  DS,[1236]
              ; LES DI,[4321]   =  MOV  DI,[4321]  MOV  ES,[4323]
LODSB,LODSW   ; AL (tai AX) <- DS:[SI], SI etenee
MOV           ; kopioidaan operandi
MOVSB,MOVSW   ; ES:[DI] <- DS:[SI]  SI,DI etenee
PUSH          ; operandi pinoon
POP           ; operandi pois pinosta
PUSHF         ; liput pinoon
POPF          ; liput pois pinosta
STOSB, STOSW  ; ES:[DI] <- AL (tai AX), DI etenee
XCHG          ; vaihdetaan operandit keskenään
XLAT          ; AL <- [BX+AL]
LEA           ; ladataan osoite (Load Efective Address)
```

LEA-käsky lataa osoitteen osoitteen sisällön sijasta. Tätä valaisee seuraava esimerkki:




```
MOV  DI,ES:[BX+SI+100H]   ; DI := 23ABH
LEA  DI,ES:[BX+SI+100H]   ; DI := 1645H
```

Käskyä käytetään mikäli samaa osoitetta tarvitaan jatkossa useasti, jolloin seuraavissa osoituksissa osoitteen laskemiseen kuluva aika jää pois ja toisaalta mahdolliset indeksirekisterit vapautuvat muuhun käyttöön.

3.8.2 Lippujen käsittelykäskyt

Lippujen arvoja käytetään ehdollisissa hypyissä. Lippujen arvoja voidaan muuttaa joko suoraan tietyillä käskyillä, tai ne muuttuvat eräiden käskyjen (aritmeettiset-, loogiset- ja pyörityskäskyt) jälkeen vastaamaan operaation tulosta.

Esimerkiksi lauseiden

```
                        ;   00110 111
MOV  AL,15H             ;    0001 0101
MOV  BL,37H             ;    0011 0111
ADD  AL,BL              ;    0100 1100
```

jälkeen tulee lipuille seuraavat arvot:

```
OF  0             - ei ylivuotoa
DF  ei muutu
IF  ei muutu
TF  ei muutu
SF  0             - etumerkki positiivinen
ZF  0             - tulos ei ole nolla
AF  0             - välimuistinumero 0
PF  0             - tuloksessa pariton määrä 1-bittejä
CF  0             - ei muistinumeroa
```

HUOM! Paritettilippu asetetaan aina vain operandien 8 alimman bitin mukaan. Mikäli halutaan testata 16-bitin paritetti, pitää tämä tehdä erikseen.

Tehtävä 3.4 Sanan pariteetti

Miten testaat AX-rekisterin pariteetin?

Suorat käsittelykäskyt:

```
CLC           ; nollataan muistinumero
CMC           ; muistinumero päinvastaiseksi
CLD           ; muistisiirrot (STOS, MOVS yms.) ylöspäin
CLI           ; kielletään keskeytykset
STC           ; asetetaan muistinumero
STD           ; muistisiirrot alapäin
STI           ; sallitaan keskeytykset
POPF          ; liput pinosta
```

3.8.3 Yhteenlaskukäskyt

Yhteenlaskukäskyissä tulos tulee aina vasemmalla ilmoitettuun operandiin.

```
ADD           ; laskee operandit yhteen
ADC           ; laskee operandit yhteen käyttäen aikaisempaa
              ; muistinumeroa
INC           ; lisää operandin arvoa yhdellä
AAA           ; korjaa ASCII-yhteenlaskun tuloksen
DAA           ; korjaa pakatun BCD-yhteenlaskun tuloksen
```

Esimerkki:

```
; Lisätään rekisteriparin DX AX muistipaikassa [SI]
; oleva 32 bittinen luku
 ADD  AX,[SI]   ; alaosien summa
 INC  SI        ; ei muuta Carry-lippua!
 INC  SI
 ADC  DX,[SI]   ; yläosien summa huomioiden edellisen carry
```

Mikäli osoittimen SI ei tarvitse kasvaa edellisessä esimerkissä, voitaisiin käyttää myös muotoa:

```
ADD  AX,[SI]   ; alaosien summa
ADC  DX,[SI+2] ; yläosien summa muistinumero huomioiden
```

Tehtävä 3.5 Liput yhteenlaskussa

Muuta seuraavat heksalukujen laskut binäärisiksi ja laske yhteenlaskut käsin merkiten jokainen muistinumero ylös. Mitkä olisivat lippujen arvot kunkin yhteenlaskun jälkeen?

```
a) 89+35 b) 89+89 c) 39+39  d) 1+1 e) 1+2
```

3.8.4 Vähennyslaskukäskyt

Vähennyslaskukäskyt toimivat kuten yhteenlaskukin. Muistinumero edustaa lainausta. Negatiivisilla luvuilla on 2-komplementtiesitys.

```
SUB           ; vähennetään operandit toisistaan
SBB           ; vähennetään huomioiden lainaus edellisestä
DEC           ; vähennetään operandia yhdellä
AAS           ; ASCII-vähennyslaskun korjaus
DAS           ; pakatun BCD-vähennyslaskun korjaus
NEG           ; 2-komplementti operandista
CMP           ; kuten SUB, mutta asettaa vain liput
```

3.8.5 Kertolaskukäskyt

8086-prosessorissa kertolasku on eräs hitaimmista käskyistä. Usein kertolasku voidaan suorittaa nopeammin sarjana siirtoja ja yhteenlaskuja.

```
MUL           ; kertoo AX:än operandilla. Tulos DX AX:ään
              ; tai AL*operandi AH AL:ään
IMUL          ; kuten edellä, mutta etumerkki huomioiden
AAM           ; pakkaamattoman BCD-kertolaskun korjaus
```

Esimerkiksi merkin paikan laskemiseksi näyttömuistissa pitää rivinumero kertoa sarakkeiden lukumäärällä: 80:llä. Seuraavassa 2 tapaa suorittaa kertolasku, kun rivinumero on rekisterissä AL. Myös ohjelmanpätkien koko ja suoritusajat on laskettu:

```
;                                                  bytes clocks
 MOV  CL,80    ;  kertoja CL:ään                     2    4
 MUL  CL       ;  tulos AX:ään                       2   80-98
               ;                    Yhteensä:        4   84-102

; 80 = 64 +16
 XOR  AH,AH    ; AH:=0                               2    3
 SHL  AX,1     ; = *2                                2    2
 SHL  AX,1     ; = *4                                2    2
 SHL  AX,1     ; = *8                                2    2
 SHL  AX,1     ; = *16                               2    2
 MOV  CX,AX    ; *16 talteen                         2    2
 SHL  AX,1     ; = *32                               2    2
 SHL  AX,1     ; = *64                               2    2
 ADD  AX,CX    ; = *64 + *16 = *80                   2    2
               ;                    Yhteensä:       18   19
```

Ensimmäinen tapa vie 4 tavua muistia ja kestää yli 80 kellokierrosta. Jälkimmäinen tapa vie 18 tavua muistia ja suoritus kestää 19 kellokierrosta. Ensimmäinen tapa on muunneltavampi, koska myös sarakkeiden lukumäärä voitaisiin välittää rekisterissä. 80186-prosessorissa ensimmäinen tehtävä veisi vain 30 kellokierrosta, joten jälkimmäinen tapa ei ehkä olisi kannattava.

3.8.6 Jakolaskukäskyt

Jakolaskukäskyissä jaettava on kooltaan kaksi kertaa suurempi kuin jakaja. Siis 16 bittisessä jakolaskussa jaettavan täytyy olla rekisteriparissa DX AX ja 8 bittisessä jakolaskussa rekisterissä AX. Tulokset tulevat vastaavasti siten, että jaon kokonaisosa tulee rekisteriin AX (AL) ja jakojäännös rekisteriin DX (AH). Mikäli jaon kokonaisosa ei mahdu sille varattuun rekisteriin, tulee 0:lla jakamista vastaava keskeytys.

Käytännössä tämä tarkoittaa sitä, että ennen jakolaskun suorittamista pitää tarkistaa, että DX (tai AH) on pienempi kuin jakaja.

```
DIV           ; jakaa DX AX:än jakajalla (rekisteri), tulos
              ; AX:ään ja jakojäännös DX:ään (tai AX -AL,AH)
IDIV          ; kuten edellä, mutta etumerkki huomioiden
AAD           ; pakkaamattoman BCDjakolaskun korjaus
CBW           ; muuttaa etumerkin laajentaen AL:ssä olevan
              ; tavun sanaksi AX:ään
CWD           ; kuten edellä AX -> DX AX
```

Esimerkki:

```
; Lasketaan millä rivillä ja sarakkeessa on näytön
; merkki, jonka järjestysnumero on AX
 MOV  CX,80   ; näytössä 80 saraketta
 XOR  DX,DX   ; DX:=0
 DIV  CX      ; nyt AX:ssä on rivi ja DX:ssä sarake
```

Edellä tulos voitaisiin laskea myös rekisteripariin AH AL, mutta tällöin täytyy varmistua jakolaskun onnistumisesta:

```
 MOV  CL,80      ; sarakkeita 80
 CMP  AH,CL      ; voidaanko jakaa?
 JAE  liian_iso  ; ei voida
 DIV  CL         ; AL:ssä rivi ja AH:ssa sarake
 ...             ; normaali käsittely jatkuu
liian_iso:       ; tästä jatketaan, jollei voida jakaa
```

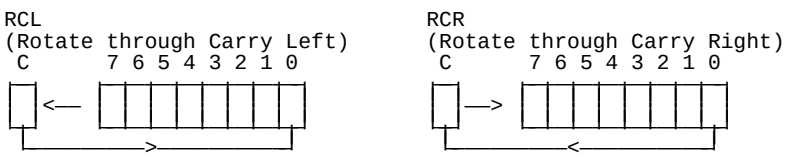
Myös jakolaskun suorittaminen kestää noin 100 kellokierrosta, joten joissakin erikoistapauksissa (esimerkiksi 2:en potenssit) sekin kannattaa korvata sarjalla siirtoja.

3.8.7 Pyöritys- ja siirtokäskyt

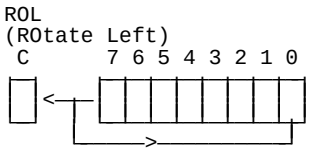
Bittipyörityksiä ja -siirtoja käytetään informaation pakkaamiseen ja 2:n potensseilla kertomiseen sekä jakamiseen.

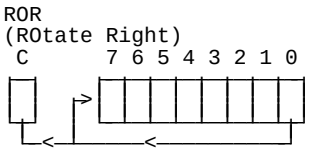
Seuraavassa pyöritysoperaatiot esitetään tavuille, mutta ne voidaan tehdä myös sana-operandeille (rekisteri tai muisti)

Pyöritys muistinumerolipun kautta:

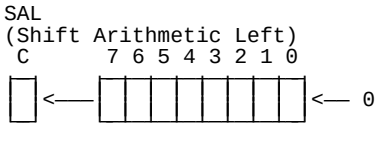


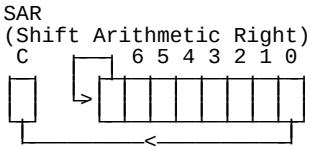
Pyöritys:



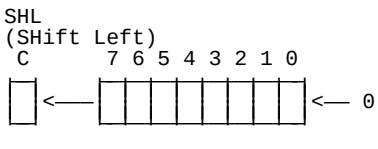


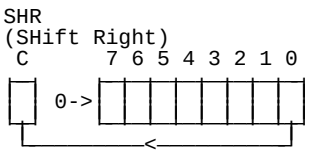
Aritmeettinen siirto:





Looginen siirto:





Aritmeettisen ja loogisen siirron ero on siinä, että aritmeettinen siirto vastaa etumerkillisen luvun jakamista tai kertomista kahdella (eli operaatiossa etumerkki säilyy), kun taas looginen siirto vastaa etumerkittömän (positiivisen) luvun kahdella kertomista tai jakamista. SHL ja SAL ovat tosin samoja käskyjä; ero tulee vain jakolaskussa (SHR ja SAR).

Siirtokäskyt muuttavat myös muita lippuja kuin muistinumerolippua (CF, joskus kirjoitetaan pelkkä C).

Pyöritysaskelten lukumäärä määrätään käskyssä olevalla toisella operandilla, joka 8086-prosessorissa voi olla vain 1 tai rekisteri CL. Uudemmissa prosessoreissa myös muut vakioaskeleet ovat sallittuja.

Esimerkkejä:

```
MOV  AX,5
SHR  AX,1       ; AX <- 2
MOV  CL,3
SHL  AL,CL      ; AX <- 16  ( 2*2^3 )
ROR  [1234H],1  ; kierretään muistipaikan 1234H sisältöä
                ; Kerrotaan rekisteripari DX AX 2:lla:
CLC             ; Nollataan muistinumero
RCL  AX,1       ; AX*2
RCL  DX,1       ; DX*2 + muistinumero
```

Viimeinen kertolasku voitaisiin suorittaa myös seuraavasti:

```
SHL  AX,1       ; AX*2, bitti 15 muistinumeroksi
RCL  DX,1       ; DX*2 + muistinumero
```

Tehtävä 3.6 Pyörityskäskyt

Mikä on rekisterin AX arvo seuraavan ohjelmanpätkän jälkeen?

```
MOV  AX,1234H
SHL  AL,1
MOV  CL,2
ROR  AX,CL
RCL  AH,1
```

3.8.8 Ehdottomat hypyt

Hyppykäskyillä asetetaan ohjelmalaskurille (IP) uusi arvo. Näin voidaan vaikuttaa ohjelman suoritusjärjestykseen. Hyppykäskyjä on neljä erilaista:

```
JMP disp8        ; ohjelmlaskuriin lisätään käskyn perässä
                 ; oleva 8 bittinen luku 16 bittiseksi
                 ; etumerkki huomioiden täydennettynä
JMP disp16       ; lisätään seuraava sana
JMP absolut      ; CS:IP on seuraava osoitin (32 bit)
JMP mem/reg      ; IP:=[mem/reg]
```

Ensin mainitulla voidaan suorittaa +/- 127 tavun siirtymiä. Pitempiin hyppyihin pitää käyttää toisena esitettyä käskyä. Käskyn valitsemisesta huolehtii useimmiten kääntäjä. Eteenpäin viittaavissa hypyissä kääntäjä ei kuitenkaan voi tietää riittääkö 8-bittinen osoitteen muutos vai tarvitaanko 16 bittiä. Ilman eri kehoitusta käännetään eteenpäin suunnatut hypyt 16-bittisiksi.

Kaksi ensimmäistä hyppyä ovat suhteellisia. Siis hypyt suuntautuvat oikeaan paikkaan riippumatta siitä, mihin ohjelmakoodi sijoittuu muistissa. Kaksi seuraavaa ovat absoluuttista hyppyjä. Näitä ei useinkaan voida käyttää .COM-tyyppisiksi käännettävissä ohjelmissa eikä mielellään aliohjelmissa. .EXE-tyyppisissä tiedostoissa ohjelman lataaja huolehtii osoitteiden muuttamisesta ennen ohjelman käynnistämistä. Absoluuttisilla hypyillä päästään segmentistä toiseen, suhteelliset pysyvät saman segmentin sisällä.

Hypyn kokoon voidaan vaikuttaa SHORT ja LONG määrityksillä.

3.8.9 Ehdolliset hypyt

Kaikki ehdolliset hypyt ovat suhteellisia ja niissä siirtymä on 8 bittiä, eli hypyn pituus voi olla +/-127 tavua. Hypyn suoritus päätetään lippurekistereiden arvon mukaan. Usein hyppyä edeltää joko CMP-käsky tai jokin looginen tai aritmeettinen operaatio, joka muuttaa lippujen arvoja.

	ehto	loog.	hypyn ehto
JA	Above	> +	CF=0 ja ZF=0
JNBE	Neither Below nor Equal	> +	CF=0 ja ZF=0
JAE	Above or Equal	>= +	CF=0
JNB	Not Below	>= +	CF=0
JNC	No Carry	>=	CF=0
JB	Below	< +	CF=1
JNAE	Neither Above nor Equal	< +	CF=1
JC	Carry	<	CF=1
JBE	Below or Equal	<= +	CF=1 tai ZF=0
JNA	Not Above	<= +	CF=1 tai ZF=0
JE	Equal	=	ZF=1
JZ	Zero	=	ZF=1
JNE	Not Equal	<>	ZF=0
JNZ	Not Zero	<>	ZF=0
JG	Greater	> e	ZF=0 ja SF=OF
JNLE	Neither Less nor Equal	> e	ZF=0 ja SF=OF
JGE	Greater or Equal	>= e	SF=OF
JNL	Not Less	>= e	SF=OF
JL	Less	< e	SF<>OF
JNGE	Neither Greater nor Equal	< e	SF<>OF
JLE	Less or Equal	<= e	SF<>OF tai ZF=1
JNG	Not Greater	<= e	SF<>OF tai ZF=1
JS	Sign	-	SF=1
JNS	No Sign	+	SF=0
JO	Overflow		OF=1
JNO	No Overflow		OF=0
JP	Parity		PF=1
JPE	Parity Even		PF=1
JNP	No Parity		PF=0
JPO	Parity Odd		PF=0
JCXZ	CX Zero		CX=0
LOOP	Loop		CX:=CX-1, CX<>0
LOOPE	Loop Equal		CX:=CX-1, CX<>0 ja ZF=1
LOOPZ	Loop Zero		CX:=CX-1, CX<>0 ja ZF=1
LOOPNE	Loop Not Equal		CX:=CX-1, CX<>0 ja ZF=0
LOOPNZ	Loop Not Zero		CX:=CX-1, CX<>0 ja ZF=0

Edellä on ehdollisista hypyistä merkitty +-merkillä ne hypyt, joiden ehto voidaan tulkita etumerkittömien (positiivisten) lukujen vähennyslaskusta tulleeksi. Vastaavasti e-kirjaimella on merkitty ehdot, joiden tulos on etumerkillisten lukujen vähennyslaskusta.

```
 MOV  AL,5
 CMP  AL,0FEH  ; vertaus joko etumerkillinen -2 tai etumerkitön 254
 JGE  yli      ; hyppää jos AL>=-2 eli esimerkissä hyppää
 JAE  yli      ; hyppää jos AL>=254 eli esimerkissä ei hyppää
 ...
yli:
 ...           ; toiminta jatkuu normaalisti
```

Mikäli ehdollisella hypyllä pitäisi päästä kauemmaksi kuin 127 tavua, pitää käyttää apuna päinvastaista hyppyehtoa ja tavallista hyppykäskyä:

```
 CMP  AX,500
 JG   yli_500  ; korvataan hyppy "JLE kauas" kahdella
 JMP  kauas    ; lauseella
yli_500:
 ...           ; toiminta jatkuu normaalisti
```

LOOP-käskyillä voidaan toteuttaa FOR-silmukan kaltaisia rakenteita.

3.8.10 Aliohjelmakutsut

Kuten korkeamman tason kielissäkin, kannattaa konekielessä jaksottaa ohjelma pienempiin kokonaisuuksiin, aliohjelmiin, jotka kukin toteuttavat oman erikseen määrätyn tehtävänsä.

CALL-käsky toimii kuten JMP-käskykin, mutta käskyä seuraava osoite laitetaan pinoon, josta se aikanaan RET-käskyllä voidaan palauttaa ohjelmalaskurin arvoksi, ja näin palata pois aliohjelmasta.

RET-käskyyn voidaan tarvittaessa liittää myös parametrejä pinosta poistava lisäyskäsky:

```
 RET  10
```

Edellisestä seuraa seuraavat muutokset:

```
IP:=[SP]
SP:=SP+2+10
```

CALL-käsky voidaan tehdä myös toiseen segmenttiin, jolloin pinoon laitetaan sekä paluu segmentti, että paluu offset-osoite. Vastaavasti tällöin pitää käyttää pitkää paluukäskyä (RET FAR), jolloin vastaavat osoitteet otetaan pinosta.

Tehtävä 3.7 Rekisteriparin arvon kääntäminen

Kirjoita aliohjelma joka 'kääntää' rekisteriparissa DXAX olevan 32-bittisen luvun toisinpäin, eli DH->AL, DL->AH jne. Kirjoita myös aliohjelmaa kutsuva testiohjelma.

3.8.11 Loogiset operaatiot

Loogiset operaatiot suorittavat operandeille bitti bitiltä tapahtuvan loogisen operaation. Loogisia operaatioita ovat

```
AND
OR
XOR
NOT
TEST          ; kuten AND, mutta vaikuttaa vain lippuihin
```

Seuraavassa totuustaulussa on kuvattu kahden vastinbitin A ja B arvoilla tulevia tuloksia kustakin operaatiosta:

A B	AND	OR	XOR	NOT A
0 0	0	0	0	1
0 1	0	1	1	1
1 0	0	1	1	0
1 1	1	1	0	0

Loogisissa operaatiossa tulee usein vastaan käsite **maski**. Maski on sana tai tavu, jossa on 1 bitit päällä niissä kohdissa, joihin jokin operaatio halutaan kohdistuvaa.

AND-operaatiota käytetään maskin avulla bittien nollaamiseen. Niistä kohdissa joissa maskissa on 1 bitti, jää alkuperäiset bitin arvot paikalleen ja muista kohdista bitit nollautuvat.

```
; ZF=1 jos mikään rekisterin AL parillisista biteistä
; ei ole päällä (AND muuttaa ZF lippua)
 AND  AL,01010101B  ; nollataan AL:än parittomat bitit
```

Olkoon edellisessä esimerkissä AL=4BH. Tällöin AND-operaation jälkeen AL:ssä on 41H:

```
4B    01001011
maski 01010101
AND   01000001
```

Vastaavasti OR-operaatiolla laitetaan bittejä päälle. XOR-operaatiolla voidaan kääntää bittejä päinvastaiseksi halutuista paikoista.

```
; Muutetaan isot kirjaimet pieniksi ja päinvastoin.
; Muutettavan merkin ASCII-koodi on AL:ssä.
 CMP  AL,'A'     ; onko edes kirjain
 JB   muutettu   ; alle 'A' kirjaimen
 TEST AL,80H     ; onko ylin bitti päällä?
 JNZ  muutettu   ; jos on, niin ei normaali ASCII-merkki
 XOR  AL,20H     ; vaihdetaan kirjaimen case-bitti
muutettu:
  ...
```

Tehtävä 3.8 Loogiset operaatiot

Mikä on rekisterin AL ja lippujen arvo kunkin seuraavan ohjelmanpätkän lauseen jälkeen?

```
MOV  AL,95H
OR   AL,AL
OR   AL,20H
XOR  AL,20H
TEST AL,20H
AND  AL,0FH
XOR  AL,AL
```

3.8.12 Keskeytykset

8086-prosessorissa on useita eri tapoja suorittaa keskeytys (interrupt):

- virhe jakolaskussa (Divide Overflow)
- NMI (Non Maskable Interrupt)
- Single Step
- laiton käsky (Unused Opcode)
- ulkoinen keskeytys
- pehmokeskeytys

Keskeytys tarkoittaa toiminnaltaan sitä, että keskeytyksen tullessa nykyinen ohjelman osan suoritus lopetetaan laittamalla pinoon lippujen arvot sekä ohjelmaosoitin (CS ja IP). Tämän jälkeen ohjelmaosoittimelle etsitään uusi arvo keskeytysvektori taulukosta, joka on osoitteessa 00000-00400. Keskeytysvektori on keskeytystä vastaavan numeron osoittamassa paikassa oleva osoitin (32 bittiä).

Keskeytysvektori osoittaa siis keskeytyksen palveluohjelmaan, joka suorittaa keskeytyksen vaatimat toimenpiteet ja tämän jälkeen palaa takaisin alkuperäisen ohjelman suoritukseen IRET-käskyllä.

Esimerkki ulkoisesta keskeytyksestä MS-DOS koneessa on näppäimen painaminen:

- painettaessa näppäintä, lähettää näppäimistön prosessori keskeytyspyynnön 9 pääprosessorille
- kun pääprosessori on suorittanut kesken olevan käskyn loppuun, tarkistaa se onko käskyn suorituksen aikana tullut keskeytyspyyntöjä. Mikäli pyyntöjä on tullut ja keskeytysten palveluja ei ole kielletty CLI-käskyllä, niin suoritetaan keskeytyksen palvelu.
- kielletään muut keskeytykset
- pinoon laitetaan lippurekisterin arvo, CS ja IP
- otetaan vastaava keskeytysvektori. Tässä tapauksessa osoitteesta (9*4) 00024H luetaan uudet arvot IP:lle ja CS:lle.
- näppäimistön palveluohjelma tallettaa kaikkien tarvitsemiensa rekistereiden arvot pinoon

- palveluohjelma vapauttaa keskeytykset STI-käskyllä (jos on oikein tehty!)
- tämän jälkeen luetaan näppäimistöprosessorilta näppäimistön tila, esimerkiksi alas painetun näppäimen SCAN-koodi
- näppäimen koodi muutetaan vastaavaksi ASCII-koodiksi ja talletetaan näppäinpuskuriin, josta se myöhemmin ohjelmallisesti voidaan lukea (BIOS-kutsu INT 16H).
- palautetaan rekistereiden arvot pinosta
- palataan IRET-käskyllä
- alkuperäinen ohjelma jatkaa toimintaansa ikäänkuin mitään ei olisi tapahtunut

NMI-keskeytys on keskeytys numero 2 ja sen palvelemista ei voida ohjelmallisesti estää. Tätä keskeytystä käytetään usein vaikeiden laitevirheiden jälkeen. Esimerkiksi muistin pariteettitarkistuksen epäonnistuttua.

Ulkoisia keskeytyslinjoja on rajallinen määrä, mutta kytkemällä linjoihin erillinen keskeytysohjain, voidaan linjojen lukumäärää kasvattaa. Keskeytykset voidaan priorisoida siten, että kaikkein tärkein keskeytys on suurimmalla prioriteetilla. Tämä tarkoittaa sitä, että alemman prioriteetin keskeytys ei pääse katkaisemaan ylemmän suoritusta vaikka keskeytykset olisi vapautettu.

Jakolaskun keskeytys tulee, mikäli jakolaskua suoritettaessa tulos ei tule mahtumaan sille varattuun tilaan. Tämä ei siis ole ulkoinen keskeytys, vaan se tapahtuu ainoastaan DIV tai IDIV-käskyn jälkeen.

Pehmokeskeytykset suoritetaan `INT nro` -käskyllä. Nro ilmaisee mikä keskeytys halutaan suorittaa. Myös ulkoisen keskeytyksen pyyntö voidaan haluttaessa antaa tällä käskyllä, mutta se ei ole suositeltavaa.

MS-DOS -käyttöjärjestelmässä kaikki systeemikutsut suoritetaan pehmokeskeytyksillä. Esimerkiksi suurin osa MS-DOSin kutsuista voidaan suorittaa keskeytyksen `INT 21` kautta. Halutun kutsun numero sijoitetaan AH-rekisteriin ja muut tarvittavat parametrit kutsun kuvauksen mukaisesti.

Esimerkiksi merkin tulostaminen kuvaruutuun:

```
MOV  AH,02H   ; Display character
MOV  DL,'A'   ; tulostetaan A-kirjain
INT  21H      ; kutsutaan käyttöjärjestelmää
  ...         ; muut käskyt
```

Yleensä pehmokeskeytykset vievät muistista 2 tavua. Poikkeuksen tekee keskeytys INT 3, joka on Breakpoint Interrupt. Tämä keskeytys vie vain yhden tavun (0CCH) ja debuggerit käyttävät tätä keskeytystä katkokohtien asettamiseen tutkittavaan ohjelmaan.

3.8.13 I/O-käskyt

Muihin oheislaitteisiin kommunikoimista varten prosessorissa on I/O-käskyt:

```
IN  akku,portti   ; luetaan portista arvo (portti<=255)
IN  akku,DX       ; luetaan DX osoittamasta portista
OUT portti,akku   ; kirjoitetaan porttiin (<=255)
OUT DX,akku       ; kirjoitetaan DX:än osoittamaan porttiin
```

Esimerkki I/O piirien käytöstä:

```
;*****************************************************************************
;
; INTEL 8255A-5 Programmable Peripheral Interface
; INTEL 8253-5  Programmable Interval Timer
;
;***************************** beep ****************************************
beep PROC NEAR
;
; input:   -
; muuttuu: FL
;
; Aliohjelmalla piipataan IBM:n kaiutinta.
;
 push_reg <AX,CX>
 MOV  AL,10110110B     ; kanava 2 = taajuusjakaja
                       ;  10 = select counter 2
                       ;  11 = lue lo ja hi peräkkäin
                       ; 011 = mode 3 = jakaja
                       ;   0 = 16 bit jakaja
                       ; ks. IBM-PC TR SDLC Adapter 1-241
 OUT  43H,AL           ; timer/counter 8253:en ohjausrekisteri (TR 1-8 SU)
 MOV  AX,2000          ; 1.19 MHz /AX => 595 Hz
 OUT  42H,AL           ; timer 2 low
 MOV  AL,AH
 OUT  42H,AL           ; timer 2 hi
 IN   AL,61H           ; luetaan alkuperäinen arvo (8255 output ks. TR 1-9 SU)
 MOV  AH,AL            ; ja talteen AH:hon
 OR   AL,000000011B    ; kun viimeinen bitti 0, niin timer 2 -> kaiuttimelle
 OUT  61H,AL           ; kaiutin päälle
 MOV  CX,8000
soi:
 LOOP soi              ; tapetaan hetki aikaa
 MOV  AL,AH
 OUT  61H,AL           ; kaiutin pois päältä
 pop_reg <CX,AX>
 RET
beep ENDP
```

Luku 4

Assembler-kieli

4.1 Yleistä

Konekielellä ohjelmoiminen olisi ihmiselle varsin vaikeata. Kuitenkin kehittämällä kutakin konekielistä käskyä vastaten oma muistisana, muistikas (mnemonic), saadaan kieli, jota on jo huomattavasti helpompi ymmärtää. Tällaista kieltä nimitetään assembler-kieleksi.

Ensimmäisten mikroprosessoreiden käskykanta oli varsin suppea. Näin myös muistikkaiden määrä oli pieni, samoin erilaisten osoitustapojen. Tällaisen prosessorin assembler-kääntäjän tekeminen on varsin helppo tehtävä.

Yleensä assembler-kielelle on tyypillistä se, että kutakin ohjelmariviä vastaa aina tietty konekielinen käsky. Ohjelmointi tapahtuu siis joka tapauksessa konekielen tarkkuudella.

4.2 8086-assemblerin vaikeudet

8086-sukuisilla prosessoreilla käskykanta on jo varsin laaja. Lisäksi erilaisia osoitustapoja on useita eivätkä kaikki osoitusvaihtoehdot edes tule kyseeseen. Kun käännetään esimerkiksi lausetta

```
MOV  AX,[BX+SI+12]
```

pitää aluksi selvittää mistä käskystä on kyse. Vaihtoehtoina on

- siirto muistista rekisteriin (sana/tavu?)
- siirto muistista segmenttirekisteriin
- siirto akkuun (sana/tavu?)
- välittömän operandin siirto rekisteriin
- rekisterin siirto rekisteriin

Seuraavaksi pitää selvittää miten DISP-osa voidaan muodostaa (riittääkö tavu tai tarvitaanko sana) ja onko edes osoitus BX+SI+12 mahdollinen. Rekisterin nimestä AX selviää, että siirrossa on kyse sanan siirtämisestä. Vielä hankalampi on käskyn

```
MOV [1234],12
```

kääntäminen. w-bitin arvon selvittämiseksi pitäisi pystyä päättämään onko kyseessä tavun 12H vai sanan 0012H siirtäminen. Ilman lisätietoa tätä ei edes pystytä päättämään ja siksi käsky usein kirjoitetaan muotoon

```
MOV  WORD PTR [1234],12
```

missä WORD PTR tarkoittaa, että kyseessä on sijoittaminen sanankokoiseen muistipaikkaan.

```
MOV  BYTE PTR [1234],12
```

tarkoittaisi vastaavasti siirtämistä tavun kokoiseen muistipaikkaan.

Tulkintavaihtoehtoja voitaisiin vähentää keksimällä eri muistikkaita eri MOV-käskyille. Esimerkiksi MOVWREG, MOVWMEM, MOVBMEM jne. Prosessorin valmistaja on kuitenkin määritellyt käytettävät muistikkaat ja assembler-kääntäjän valmistajan on tyydyttävä näihin. Kielen käyttäjän kannalta on toki helpompi mikäli muistikkaita on vähän. 8086-assemblerissa tosin muistikkaiden käyttöä vaikeuttaa epäsymmetrisyys.

Erityisesti edellä mainitut vaikeudet tulevat esiin käytettäessä eteenpäin viittaavia symbolisia nimiä. Esimerkiksi viittauksesta

```
MOV  jemma,lkm
```

ei voi päättää mitään ennenkuin sekä jemma että lkm on esitelty.

4.3 Muita assembler-kielen ominaisuuksia

Assembler-kääntäjän valmistajasta riippuu mitä muita ominaisuuksia kielessä on muistikkaiden ja rekistereiden symbolisten nimien lisäksi.

Jatkossa käsitellään Microsoftin Macro Assembler-kielen ominaisuuksia.

4.3.1 Kommenttimerkki

Lähes kaikissa kääntäjissä on kommenttimerkki. Usein kommenttimerkkinä on puolipiste (;), jonka jälkeinen osa rivillä jätetään huomioimatta.

```
MOV  AX,5  ; laitetaan yhden käden sormien lukumäärä akkuun
```

4.3.2 Nimiö

Myös nimiön käyttö on yleistä. Yleensä nimiö (label) merkitään rivin alkuun siten, että nimiö loppuu kaksoispisteeseen (:). Nimiöihin voidaan kohdistaa esimerkiksi hyppykäskyjä.

```
 CMP  AL,'a'  ; onko AL pienempi kuin a-kirjain
 JL   alle_a  ; jos on, niin jatketaan alle_a-kohdasta
  ...
alle_a:       ; täältä jatketaan, mikäli AL oli alle 'a':n
 MOV  AL,'B'  ; vaihdetaan isoksi B:ksi
```

Nimiölle voidaan antaa myös tyyppi LABEL-käskyllä (directive) esimerkiksi myöhempää sijoitusta varten

```
sanat LABEL WORD  ; seur. paikalle käyt. myös. nimitystä sanat
tavut DB  10,20
 ...
 MOV  tavut,30    ; koska tavut BYTE-tyyppiä, niin sijoittaa
 ...              ; tavun 30
 MOV  AX,sanat    ; MOV  AX,tavut olisi laiton!
```

4.3.3 Symbolisen nimen määritys

EQU-käskyllä voidaan antaa symbolinen nimi jollekin merkkijonolle. Nimen esiintyessä korvataan tämä myöhemmin vastaavalla merkkijonolla ennen käännöksen suorittamista.

```
sormia    EQU 10     ; ihmisellä on 10 sormea
varpaita  EQU 10     ; myös varpaita on yleensä 10
 ...
 MOV  AX,sormia      ; lasketaan sormet ja varpaat yhteen
 ADD  AX,varpaita    ; tämä kääntyy ADD  AX,10
```

```
p         EQU  [BP+04H]
 ...
 MOV  AX,p           ; tämä kääntyy MOV  AX,[BP+04H]
```

4.3.4 Muistipaikan varaaminen

DB, DW ja DD-käskyillä voidaan varata vastaavan kokoiset muistialueet, joilla on jatkossa myös varausta vastaava tyyppi. Lauseessa voidaan myös antaa muistipaikalle alkuarvo. Mikäli alkuarvoja annetaan useampia, varataan vastaavasti useita peräkkäisiä muistipaikkoja.

```
; Varataan kokonaislukumuistialue, jolla on nimi paivia_kk
; ja jonka kussakin paikassa on valmiina vastaavan kuukauden
; paivien lukumäärä.
;            ta he ma hu to ke he el sy lo ma jo
paivia_kk DW 31,28,31,30,31,30,31,31,30,31,30,31
; muuta varattavat muistipaikat:
kirjain   DB 'a'     ; tavunkokoinen muistipaikka alkuarvona 61H
numero    DW 123     ; sanankokoinen muistipaikka alkuarvona 007BH
tavu      DB 41H     ; tavunkokoinen muistipaikka alkuarvona 41H
viesti    DB 'Anna kuukauden numero >','$'
tulos     DW ?       ; sanankokoinen muistipaikka, arvo tuntematon
 ...
 MOV  SI,CX               ; CX:ssä kk:n numero
 DEC  SI                  ; SI kk: järj. nro alk. 0
 SHL  SI,1                ; kerrotaan SI 2:lla
 MOV  AX,paivia_kk[SI]    ; AX:ään päivien lukumäärä ko. kuussa
 MOV  kirjain,'B'         ; muutetaan kirjain B:ksi
 MOV  numero,1234H        ; siirretään sanaan numero 1234H
```

DUP-käskyn avulla voidaan muistipaikan varauksessa monistaa haluttua alkuarvoa useampaan muistipaikkaan:

```
alkupaivat DW 12 DUP(?)    ; varataan tila kuun alkup.
 ...
 MOV  alkupaivat[SI],AX    ; talletetaan alkupäivän nro
```

Muistin varauksessa kannattaa käyttää aina alkuarvoa ?, mikäli se on mahdollista. Tämä korostaa ohjelman lukijalle, että arvo on tuntematon ja toisaalta vie ohjelmaa levylle talletettaessa vähemmän tilaa.

Käännöksessä varatun muistipaikan nimi korvataan hakasuluilla ja paikan osoitteella. Mikäli halutaan viitata nimenomaan muistipaikan osoitteeseen, käytetään muotoa

```
 MOV  SI,OFFSET alkupaivat
```

Muistipaikan varaamisen jälkeen paikalle varattu nimi kertoo myös muistipaikan tyypin. Edellä esimerkiksi sijoitus

```
 MOV  alkupaivat[SI],AL
```

johtaisi vähintäänkin kääntäjän varoitukseen, koska `alkupaivat` oli varattu sana-tyyppiseksi.

Tehtävä 4.1 Heksamuunnos taulukon avulla

Kirjoita muunnostaulukon avulla aliohjelma, joka muuttaa rekisterissä AL olevan luvun 0-15 vastaavaksi heksamerkiksi.

4.3.5 Käännösaikana laskettavat lausekkeet

Assembler-kielen ominaisuuksiin kuuluu käännösaikana laskettavat lausekkeet. Näiden avulla voidaan helpottaa erilaisten vakioiden ja osoitteiden kirjoittamista ja näin ollen vähentää ohjelman muutoksissa tarvittavaa työtä.

Lausekkeiden on kuitenkin ehdottomasti oltava sellaisia, jotka pystytään laskemaan käännösaikana. Muuttujia niissä ei siis voi olla, mutta EQU-lauseella annettuja vakioita saa käyttää.

```
tun_vrk    EQU 24
min_tun    EQU 60
min_vrk    EQU tun_vrk*min_tun
```

Operaattorit suoritusjärjestyksessä ovat:

```
<>,(),[], LENGTH, MASK, SIZE, WIDTH
.          (struktuureja käytettäessä)
HIGH, LOW
+,-        (etumerkkinä)
:          (segmentin vaihto)
OFFSET, PTR, SEG, THIS, TYPE
*,/, MOD, SHL, SHR
+,-
EQ, GE, GT, LE, LT, NE
NOT
AND
OR, XOR
LARGE, SHORT, SMALL, TYPE
```

4.3.6 Makrot

Usein assembler-kääntäjiin liittyy myös EQU-käskyyn verrattava tekstinkäsittelymakro monirivisille komentosarjoille. Makroja käyttämällä voidaan muutoin raskasta konekielistä ohjelmointi keventää huomattavasti. Sopivilla makroilla voidaan luoda jopa oma kieli.

Makroista on aina muistettava, että ne EIVÄT ole aliohjelmia, vaan makrojen kutsut korvataan täsmälleen niiden alkuperäisellä tekstillä ENNEN käännöksen suorittamista.

Makron sijainnilla ei ole väliä ohjelmakoodiin nähden, kunhan se on esitelty ennen makron kutsua.

```
; Makro lopettaa ohjelma suorituksen ja palauttaa kontrollin
; käyttöjärjestelmälle.  Käytetään MS-DOSin funktiokutsua
; 4CH.  Parametrinä kutsulle välitetään MS-DOSin ERRORLEVEL
lopeta MACRO errorlevel
 MOV  AH,4CH
 MOV  AL,errorlevel
 INT  21H
ENDM
 ...
 lopeta 0       ; ohjelma päättyi oikein
 ...
```

Edellisessä esimerkissä korvataan käännösaikana teksti `lopeta 0` tekstillä

```
 MOV  AH,4CH
 MOV  AL,0
 INT  21H
```

Makrossa voi olla myös useita parametrejä. Edellinen makro voitaisiin kirjoittaa myös seuraavasti:

```
lopetus EQU 4CH
; Seuraavalla makrolla kutsutaan MS-DOSia funktiolla
; funktio.  Parametri siirretään rekisteriin AL.
MS_DOS MACRO funktio,parametri
 MOV  AH,funktio
 MOV  AL,parametri
 INT  21H
ENDM
 ...               ; Muita ohjelman lauseita kunnes
 MS_DOS lopetus,0  ; ohjelma päättyi oikein
 ...
```

Myös ehdollinen kääntäminen on mahdollista. Mikäli edellisen esimerkin makroa kutsuttaisiin vaikkapa siten, että ohjelman aikaisemmissa vaiheissa virhekoodi olisi jo tullut rekisteriin AL, niin kutsu

```
MS_DOS lopetus,AL
```

kääntäisi tarpeettoman lauseen

```
MOV  AL,AL
```

Tämä voitaisiin välttää ehdollisella kääntämisellä. Jätetään MOV-lause kääntämättä, mikäli parametri on valmiiksi AL:

```
lopetus EQU 4CH
; Seuraavalla makrolla kutsutaan MS-DOSia funktiolla
; funktio.  Parametri siirretään rekisteriin AL.
MS_DOS MACRO funktio,parametri
 MOV  AH,funktio
 IFDIF <parametri>,<AL>
  MOV  AL,parametri     ;; käännetään vain tarvittaessa
 ENDIF
 INT  21H
ENDM
```

Edellä IFDIF vertaa kahta pilkulla erotettua kulmasuluissa olevaa lauseketta ja suorittaa käännöksen, mikäli lausekkeet ovat erisuuria. Ehdollisessa käännöksessä voi olla myös ELSE-osa.

Ehdollisen käännöksen käskyt ovat:

IF lauseke	käännetään, mikäli lauseke<>0
IF1	käännetään, mikäli 1. läpikäynti menossa
IF2	käännetään, mikäli 2. läpikäynti menossa
IFB <arvo>	käännetään, mikäli arvo on tyhjä
IFDEF symboli	käännetään, mikäli symboli määritelty
IFDIF <l1>,<l2>	käännetään, mikäli l1<>l2
IFDIFI <l1>,<l2>	käännetään, mikäli l1<>l2 kun isot ja pienet kirjaimet samaistetaan
IFE lauseke	käännetään, mikäli lauseke=0
IFIDN <l1>,<l2>	käännetään, mikäli l1=l2
IFIDNI <l1>,<l2>	käännetään, mikäli l1=l2 kun isot ja pienet kirjaimet samaistetaan
IFNB <arvo>	käännetään, mikäli arvo ei ole tyhjä
IFNDEF symboli	käännetään, mikäli symbolia ei ole määritelty

Symboli saadaan määritellyksi joko EQU-lauseella tai jopa käännöskäskyssä optiolla /Dsymboli. Näin ohjelmaan voidaan tehdä osia, joita käytetään testaustarkoituksissa, versioriippuvaisissa osissa tai esimerkiksi tulosteiden kielen vaihtamiseksi:

```
kieli = 0
IFDEF suomi
 tulosta 'Syötä arvo >'
 kieli = 1
ENDIF
IFDEF englanti
 tulosta 'Enter value >'
 kieli = 2
ENDIF
IFE kieli ; Mikäli kieltä ei ole määritelty
          ; tulostetaan virheilmoitus ja kaadetaan käännös
%OUT Käytä käännöksessä joko optiota /Dsuomi
%OUT tai /Denglanti
.ERR
ENDIF
```

Edellä = -merkki on kuten EQU, mutta sijoituksen arvoa voi muuttaa kesken käännöksen. Sijoitus toimii tosin vain numeerisille arvoille.

Makroissa voidaan käyttää myös erinäisiä toistorakenteita, joiden avulla koodia saadaan lyhyemmäksi:

```
;------------------------------ push_reg  -----------------------------
push_reg MACRO rekisterit
;
; Input:   lista rekistereistä
; Muuttuu: SP
;
; Makrolla laitetaan listassa luetellut rekisterit pinoon.
; Kutsu esim. push_reg <AX,BX,CX,DX>
;
  IRP reg,<rekisterit>
   PUSH reg
  ENDM
 ENDM
```

Edellisessä esimerkissä esitetty kutsu kääntyisi sarjaksi PUSH-käskyjä:

```
 PUSH AX
 PUSH BX
 PUSH CX
 PUSH DX
```

Tehtävä 4.2 pop_reg

Kirjoita edellistä makroa vastaava makro `pop_reg`.

IRPC-makrolla voidaan toistaa kutsun perässä olevaa merkkijonoa merkki kerrallaan:

```
; Rekisterissä AL on käyttäjän vastaus.
; Verrataan onko vastaus k K y tai Y
 IRPC kylla,kKyY
  CMP  AL,'&kylla&'
  JZ   kylla_vastaus
 ENDM
 JMP ei_vastaus
kylla_vastaus:
```

&-merkit muuttujan ympärillä tekevät siitä muuttujan myös lainausmerkkien sisällä, joten se korvautuu ensimmäisellä kerralla k-kirjaimella ja viimeisellä Y-kirjaimella. Edellinen esimerkki kääntyisi siis seuravasti:

```
 3C 6B                 CMP  AL,'k'
 74 0F                 JZ   kylla_vastaus
 3C 4B                 CMP  AL,'K'
 74 0B                 JZ   kylla_vastaus
 3C 79                 CMP  AL,'y'
 74 07                 JZ   kylla_vastaus
 3C 59                 CMP  AL,'Y'
 74 03                 JZ   kylla_vastaus
 EB 85 90              JMP ei_vastaus
                      kylla_vastaus:
```

Seuraavassa esimerkissä varataan kokonaislukutaulukko, jossa sanoissa 0,1,2,...,8 on vastaavien lukujen kertomat.

```
; Tähän tulee taulukko 1,1,2,6,24,120,720,5040,40320
kertoma  = 1
n        = 1
kertomataulu LABEL WORD
 REPT 9                ;; Toistetaan 9 kertaa (viimeinen 8 kertoma)
 DW kertoma            ;; varataan n-1:en kertomalle tila
  kertoma = n*kertoma  ;; lasketaan seuraava kertoma
  n       = n+1        ;; siirrytään seuraavaan n:än arvoon
 ENDM
 ...
 MOV  SI,AX            ; Ladataan AX:ään AX:än kertoma
 SHL  SI,1             ; Kertomataulu oli sanoina
 MOV  AX,kertomataulu[SI]
```

Makroissa voidaan myös käyttää lokaaleja nimiöitä, jolloin makron uudelleen käyttäminen ei aiheuta nimiön toistumista:

```
;****************************** jif ************************************
; Input:    x1,x2,hyppy
; Output:   IP
; Muuttuu:  liput,IP
;
; Esimerkki:  jif  AX,5,on_vitonen
;
; Makrolla suoritetaan pitkä hyppy paikkaan hyppy mikäli
; x1 ja x2 ovat yhtäsuuria.  Jos kutsussa on vakio, pitää sen olla
; x2.  Molemmat (x1 ja x2) eivät saa olla muistipaikkoja.
;
jif MACRO x1,x2,hyppy
LOCAL yli
 CMP  x1,x2
 JNZ  yli
 JMP  hyppy
yli:
ENDM
```

Ilman LOCAL määritystä ei makroa `jif` voitaisi käyttää kuin yhden kerran, määrityksen kanssa nimiö `yli` korvataan jokaisella esiintymiskerrallaan eri nimiöllä. Makro on kätevä mikäli joudutaan tekemään yhtäsuuruusvertailu ja pitkä hyppy, muussa tapauksessa jouduttaisiin aina keksimään uusia nimiöitä nimiön `yli` tilalle.

Viimeisenä esimerkkinä makroista esitetään kertolaskua optimoiva makro.

```
;***************************** kerro ***********************************
; Input:    AX,kerroin,merkki
;           I286
;           OPTSIZE
; Output:   DX AX
; Muuttuu:  liput, DX, AX
;
; Esimerkki:  kerro  16,e
;
; Makrolla kerrotaan AX:ssä oleva etumerkitön (merkki<>e) tai etumerkillinen
; (merkki=e) luku kutsussa olevalla positiivisella kertoimella.
; Suoritusaikaa optimoidaan siten, että mikäli kerroin on 2 potenssi,
; suoritetaan kertominen siirtoina oikealle.  Muuten käytetään
; MUL-käskyä.  Siiräminen tehdään joko oikealle tai vasemmalle riipuen
; siitä kumpi on optimoinnin kannalta parempi.
;
; Lasketaan kannattaako kääntää siirtoja vai kertolasku.
; Mikäli kääntäjän kutsussa on määritelty parametri I286, käytetään
; kellokierroksia laskettaessa 80286-prosessorin aikoja.
;
; Mikäli parametri OPTSIZE on määritelty, käytetään kertolaskua
; muulloin paitsi jos siirtely vie vähemmän tilaa.
;
; Nollalla kertominen käsitellään aina erikseen.
;
kerro MACRO kerroin,merkki
IF kerroin NE 0             ;; Eihän kerrota nollalla?
;; --------------------- Optimointiin liittyvät muuttujat -------------
shift_time   = 2         ; Siirtojen kesto kellokierroksina
shift_size   = 2         ; Siirtojen koko tavuina
AXDX_size    = 2+2       ; AX->DX, AX<-0 koko
AXDX_time    = 2+3       ; AX->DX, AX<-0 aika
IFDIFI <merkki>,<e>      ; Jos etumerkitön luku, nollataan DX
 DX0_size    = 2         ; käskyllä SUB DX,DX
 DX0_time    = 3
ELSE
 DX0_size    = 1         ; Muuten CWD-käskyllä
 IFDEF I286              ; Jonka suoritusaika riippuu prosessorista
  DX0_time   = 4
 ELSE  DX0_time   = 5
 ENDIF
ENDIF
IFDEF I286               ; Kertolaskun ja siirron suoritusaika kellokierroksina
 mul_time    = 35+4      ; 80286
ELSE
 mul_time    = 115+4     ; 8086
ENDIF
mul_size     = 3+2       ; MOV DX,kerroin ja MUL DX
;; --------------------- Makron apumuuttujat --------------------------
on_2_potenssi = 0        ; =1 jos potenssi löytyy
n             = 0        ; luku jonka potenssiksi epäillään
```

```
potenssi      = 1        ; 2 potenssiin n
;; ------------------ Tutkitaan onko 2 potenssi -----------------------
 REPT 16                      ;; Etsitään potenssia kaikilla n:än arvoilla.
   IF potenssi EQ kerroin     ;; Onko 2 potenssiin n ?
     on_2_potenssi = 1        ;; On, laitetaan muuttuja todeksi
     EXITM                    ;; ja lopetetaan potenssin etsiminen.
    ENDIF
   n          = n+1           ;; muuten siirrytään seuraavaan n:än arvoon,
   potenssi   = potenssi SHL 1                          ;; ja potenssiin.
  ENDM                        ;; REPT 16 - potenssin etsiminen

;---------- Tutkitaan kannattaako siirtely ja kumpaan suuntaan? ------
oikealle MACRO
  vasemmalle = 0
  total_time = rigth_time
  total_size = rigth_size
  siirtoja   = 16-n
ENDM
siirtelemalla = 0             ;; =1 jos siiretely kannattaa. Olet. ei siirtoa
vasemmalle    = 1             ;; Oletetaan siirro vasemmalle
siirtoja      = n             ;; Vas. siirrettäessä n kpl siirtoja
  IF on_2_potenssi            ;; Tutkitaan siirtojen viemät ajat, koko ja suunta
    left_time   = DX0_time  + (n*2*shift_time)
    rigth_time  = AXDX_time + ((16-n)*2*shift_time)
    left_size   = DX0_size  + (n*2*shift_size)
    rigth_size  = AXDX_size + ((16-n)*2*shift_size)
    total_time  = left_time
    total_size  = left_size
    IFDEF OptSize             ;; Jos optimoidaan kokoa, asetetaan koko etus.
      IF left_size GT rigth_size   ;; Oikealle koko pienempi?
        oikealle
       ENDIF                      ;; Oikealle koko pienempi?
      IF total_size LE mul_size    ;; Kannattaako siirtaminen koon puolesta?
        siirtelemalla = 1
       ENDIF                      ;; Koon puolesta?
     ELSE                     ;; Aika etusijalla
      IF left_time GT rigth_time   ;; Oikealle aika pienempi?
        oikealle
       ENDIF                      ;; Oikealle aika pienempi?
      IF total_time LT mul_time    ;; Kannattaako siirtäminen ajan puolesta?
        siirtelemalla = 1
       ENDIF                      ;; Ajan puolesta?
     ENDIF                    ;; Jos optimoidaan kokoa
   ENDIF                      ;; Tutkitaan siirtoja

  IF siirtelemalla            ;; Mikäli oli 2 potenssi ja siirtely kannatti.
    IF vasemmalle             ;; Kannattaako siirtää oikealle vai vasemmalle?
      IFDIFI <merkki>,<e>     ;; Etumerkitön?
        SUB  DX,DX            ;  DX := 0;
       ELSE
        CWD                   ;  AX:än etumerkki DX:ään
       ENDIF                  ;; Etumerkitön?
      REPT siirtoja           ;; Siirretään siirtoja kertaa vasemmalle
        SHL  AX,1             ;; Ylimäärinen bitti carryyn ja nolla sisään.
        RCL  DX,1             ;; Carry sisään oikealta.
       ENDM                   ;; REPT n - n vasemmalle siirtoa
     ELSE
      MOV  AX,DX              ;; AX DX:ään ja kierretään oikealle
      XOR  AX,AX              ;; nollataan AX
      REPT siirtoja           ;; Siirretään siirtoja kertaa vasemmalle
        IFDIFI <merkki>,<e>   ;; Etumerkitön?
          SHR  DX,1           ;; Ylimäärinen bitti carryyn ja nolla sisään.
         ELSE
          SAR  DX,1           ;; Etumerkkiä monistaen ja ylim. carryyn.
         ENDIF                ;; Etumerkitön?
        RCR  AX,1             ;; Carry sisään vasemmalta.
       ENDM                   ;; REPT n - n vasemmalle siirtoa
     ENDIF                    ;; Siirto vasemmalle?
   ELSE                       ;; Ei siirtoa, kertolasku on parempi.
    total_time   = mul_time   ;; kertolaskuun kuluva aika
    total_size   = mul_size   ;; kertolaskun viemä tila
    siirtoja     = 0          ;; Ei siis myöskään siirtoja
    MOV  DX,kerroin
    IFDIFI <merkki>,<e>       ;; Etumerkitön?
      MUL  DX
     ELSE                     ;; Etumerkillinen
      IMUL DX
     ENDIF                    ;; Etumerkitön?
   ENDIF                      ;; Siirtelemällä?
 ELSE                         ;; 0:lla kertominen käsitellään erikseen:
   total_time = 3 + 2
   total_size = 2 + 2
   XOR  AX,AX                 ;; Eli sekä AX := 0
   MOV  DX,AX                 ;; että DX := 0
 ENDIF                        ;; 0:lla kertominen?

ENDM                          ;; Kerro
```

```
;************************************************************************
```

Makron kutsut kääntyisivät alla olevan taulukon mukaisesti:

kerro 10	kerro 0	kerro 1	kerro 16384	kerro 2,e
MOV DX,10 MUL DX	XOR AX,AX MOV DX,AX	SUB DX,DX	MOV DX,AX XOR AX,AX SHR DX,1 RCR AX,1 SHR DX,1 RCR AX,1	CWD SHL AX,1 RCL DX,1

Mikäli käännös suoritettaisiin kutsulla `MASM kerto /DOPTSIZE;`, kääntyisi myös kutsu `kerro 16384` kertolaskuksi.

4.3.7 Include

Käytännössä koko ohjelmaa ei kannata kirjoittaa samaan tiedostoon. Tällöin voidaan tehdä ’päätiedosto’, jossa INCLUDE-käskyillä kutsutaan muut tiedostot mukaan. Esimerkiksi omat makrot ja muut hyvät vakiomääritykset kannattaa kasata omaksi tiedostokseen, jotka sitten käännösaikana liitetään varsinaiseen ohjelmaan:

```
INCLUDE makrot.asm
 ...                ; ohjelmalauseita
```

4.3.8 Struktuurit

Usein korkemman tason kielissä määritellään yksinkertaisia muuttujia monimutkaisempia tietorakenteita. Esimerkiksi Turbo Pascalissa voitaisiin määritellä henkilöille seuraava tietue-tyyppi:

```
TYPE henkilo = RECORD
                 nimi      : STRING[30];
                 sotu      : STRING[10];
                 ika       : INTEGER;
                 pituus_cm : INTEGER;
                 paino_kg  : INTEGER;
               END;
```

Vastaava tyyppi voitaisiin tehdä myös assembler-kielessä STRUC-käskyn avulla:

```
henkilo STRUC
 nimi      DB 31 DUP(?)  ; 1 tavu varataan nimen pituudelle
 sotu      DB 11 DUP(?)  ;
 ika       DW  ?
 pituus_cm DW  ?
 paino_kg  DW  ?         ; Yhteensä 48 tavua/henkilö
henkilo ENDS
```

Tällä tavalla luodulla tietotyypillä voidaan varata myös muistipaikkoja:

```
vanha    henkilo <,,99,,>   ; ikä kenttään arvo 99
pitka    henkilo <,,,205,>  ; pituudeksi 205 cm
pullukka henkilo <,,,,130>  ; painoksi 130 kg
muut     henkilo 10 DUP(<>) ; loput 10 henkilöä
```

Muistipaikkaa varattaessa ne kentät, joille ei ole annettu arvoa saavat oletusarvon. Mikäli nimi- ja sotu-kenttiin haluttaisiin sijoittaa edellä arvoja, olisi ne pitänyt varata merkkijonotyyppisinä:

```
 nimi      DB ’oletusnimi’
```

Käännöksessä kentän nimeen viittaaminen kääntyy pelkästään tavujen lukumääräksi rakenteen alusta lukien.

Mikäli esimerkin muistivarauksessa `vanha` menisi osoitteeseen 0H, olisi `pitka` osoitteessa 30H (=48) ja `pullukka` osoitteessa 60H sekä `muut` alkaisivat osoitteesta 90H. Tällöin käännöksessä tulisi seuraavat vastaavuudet:

```
MOV  AX,vanha.ika            ; MOV  AX,[002AH]
MOV  AX,pullukka.pituus_cm   ; MOV  AX,[60H+2CH]
MOV  CL,pitka.sotu           ; MOV  AX,[30H+1FH]
```

Edelleen olkoon rekisterissä AX sen henkilön järjestysnumero (alkaen 1), johon `muut` muistipaikassa halutaan viitata. Ko. henkilön paino voitaisiin siirtää rekisteriin DX seuraavilla lauseilla:

```
MOV  DX,SIZE henkilo         ; tavujen määrä tietueessa
DEC  AX                      ; koska numerointi alkoi 1
MUL  DX                      ; nro*tavujen määrä
MOV  SI,AX                   ; SI osoittaa oikeaan henkilöön
MOV  DX,muut[SI].paino_kg    ; ja henkilön paino DX:ään
```

4.3.9 Bittikentät

STRUC-rakenteella voitiin kasata tavuista koostuvia kokonaisuuksia. Usein tietoa täytyy kuitenkin pakata bittitasolla yhden tavun tai sanan sisällä.

Esimerkiksi MS-DOS-koneissa on muistipaikassa 00410H tavu, johon on talletettu koneessa olevien laitteiden lukumäärä seuraavasti:

```
7 6 5 4 3 2 1 0
┌─┬─┬─┬─┬─┬─┬─┬─┐
│l│l│v│v│r│r│m│f│
└─┴─┴─┴─┴─┴─┴─┴─┘
```

Edellä kentillä on seuraava tarkoitus:

ll	levykeasemien lukumäärä - 1
vv	näytön tyyppi: 00 = ei näyttöä 01 = 40 x 25 väri 10 = 80 x 25 väri 11 = 80 x 25 mustavalko
rr	varattu
m	matematiikkaprosessori asennettu
f	onko levykeasemia

Mikäli rekisterissä AL olisi laitetavun arvo, pitäisi näytön tilan kokonaisluvuksi muuttamiseksi tehdä seuraavat operaatiot:

```
AND  AL,00110000B    ; poistetaan tavusta ylimääräinen osa
MOV  CL,4            ; siirron määrä
SHR  AL,CL           ; kierretään video-moodi laitaan
```

Edellisessä täytyy ohjelmoijan kuitenkin itse laskea maski `00110000` ja siirron pituus 4. RECORD-käskyllä laskut voidaan jättää kääntäjän tehtäväksi ja ohjelman muuttaminen helpottuu:

```
video_mode RECORD levyja:2,naytto:2,varattu:2,matem:1,floppy:1
;
;---------------------- video moodi ------------------------
;
; Input:
;
; Output: AL = 0, jos ei valittua näyttömoodia
;              1, jos 40 x 25 väri
;              2, jos 80 x 25 väri
;              3, jos 80 x 25 mustavalko
;
; Muuttuu: liput,AL
;
; Aliohjelmalla palautetaan rekisterissä AL kutsuhetkellä
; valittuna oleva videomoodin arvo.
;
video_moodi PROC NEAR
 push_reg <CX,ES>
 MOV  CX,0040H
 MOV  ES,CX             ; ES := BIOS segmentti
 MOV  AL,ES:[10H]       ; laitetavu
 AND  AL,MASK naytto    ; poistetaan muut paitsi video-moodin bitit
 MOV  CL,naytto
 SHR  AL,CL             ; siirretään videomoodi AL:än oikeaan reunaan
 pop_reg <ES,CX>
 RET
video_moodi ENDP
```

RECORD-käskyllä määritellyssä rakenteessa korvataan kenttien nimet käännöksen aikana vastaavilla siirtymillä. `MASK ja nimi` korvataan vastaavalla bittimaskilla ja `WIDTH ja nimi` kentän pituudella. Esimerkin käännöksessä seuraisi seuraavat vastaavuudet:

nimi	nimi	MASK nimi	WIDTH nimi
levyja	6	11000000	2
naytto	4	00110000	2
varattu	2	00001100	2
matem	1	00000010	1
floppy	0	00000001	1

Kentän nimi vastaa siis suoraan kentän oikeaan reunaan siirtämiseen tarvittavien askelten lukumäärää.

Kentille voidaan antaa myös alkuarvoja, joita käytetään myöhemmin tilavarauksissa apuna:

```
pakattu RECORD k1:2=1,k2:5=7,k3:1=1
;         seuraa määritys k1   k2  k3
;                         01 00111 1
p1 pakattu <>           ; 01 00111 1
p2 pakattu <3,8,0>      ; 11 01000 0
p3 pakattu <,12,><2,,0> ; 01 01100 1  10 00111 0
```

4.3.10 Yhteiset muistialueet

UNION-käskyllä voidaan luoda tietuetyyppi, jossa samassa muistipaikassa on useita erityyppisiä muuttujia (vrt. Pascalin RECORD CASE tyyppi OF -rakenne). Tämä käsky on tosin käytössä mm. Turbo Assembler -kääntäjässä (ei MASM 5.0:ssa).

```
ryhma UNION
sana  DW 2
tavu  DB ?
ryhma ENDS
 ...
oma ryhma <>
 ...
 MOV  AX,oma.sana  ; muuttujaan oma voidaan viitata sanana
 MOV  BL,oma.tavu  ; tai tavuna.
```

Tyypille varataan tilaa suurimman yksittäisen tietotyypin varaaman tilan verran.

4.3.11 Segmentit

Koska 8086-prosessori on segmenttijakoinen, täytyy myös assembler-ohjelmoijan huolehtia siitä, mistä segmentistä mikäkin muistipaikka otetaan. Kullekin muistialueelle (myös ohjelmakoodille) ilmoitetaan ohjelmakoodissa SEGMENT-käskyllä mihin segmenttiin se kuuluu.

ASSUME-käskyllä ilmoitetaan mitkä arvot ohjelmoija 'kuvittelee', segmenttirekistereillä olevan ko. ohjelmalohkon suorituksen aikana. ASSUME ei siis missään tapauksessa aiheuta esiintymiskohtaansa mitään kääntyvää koodia. Ohjelmoijan on itse pidettävä huoli siitä, että segmenttirekistereille tulevat oikeat arvot.

```
DATA SEGMENT           ; Tämän alle määritellään muuttujat.
terve DB 'Terve!','$'
DATA ENDS

CODE SEGMENT           ; Tähän kirjoitetaan ohjelmakoodi.
ASSUME CS:CODE,DS:DATA ; Jatkossa oletetaan eo. segmentit.

ohjelma PROC NEAR
 MOV  AX,DATA          ; Ilman tätä ei DS:llä ole oikeata arvoa.
 MOV  DS,AX
 MOV  AH,09H           ; Tulostetaan merkkijono terve
 MOV  DX,OFFSET terve
 INT  21H              ; käyttöjärjestelmän kutsulla.
 MOV  AX,4C00H
 INT  21H              ; Lopetetaan ohjelma.
ohjelma ENDP

CODE ENDS
     END ohjelma       ; Suoritus aloitetaan paikasta ohjelma.
```

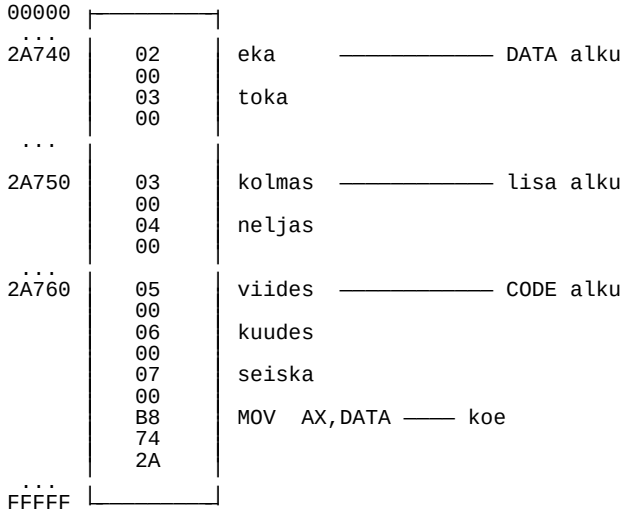
Seuraava esimerkki kuvastaa ASSUME-käskyn merkitystä:

```
DATA SEGMENT
eka    DW 2
toka   DW 3
DATA ENDS

lisa SEGMENT
kolmas DW 3
neljas DW 4
lisa ENDS

CODE SEGMENT
ASSUME CS:CODE
viides DW 5
kuudes DW 6
seiska DW 7
koe PROC NEAR
  MOV  AX,DATA
  MOV  DS,AX     ; DS paikalleen
 ASSUME DS:DATA
  MOV  AX,lisa
  MOV  ES,AX     ; ES paikalleen
 ASSUME ES:lisa
  MOV  AX,toka
  MOV  CX,neljas
  MOV  DX,seiska
koe ENDP
CODE ENDS
  END koe
```

ASSUME -lauseet voivat olla koodissa missä tahansa, mutta yleensä paras paikka on käyttää niitä sitten, kun segmentille on varmasti sijoitettu haluttu arvo (kuten edellisessä esimerkissä).

Käännettynä ja ajettuna muistipaikkaan 10H saakka näyttäisi ohjelma DEBUG-ohjelmalla seuraavalta:

```
-g 10
AX=2A75  BX=0000  CX=003D  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=2A74  ES=2A75  SS=2A74  CS=2A76  IP=0010   NV UP EI PL NZ NA PO NC
2A76:0010 A10200         MOV    AX,[0002]                          DS:0002=0003
-u 10,18
2A76:0010 A10200         MOV    AX,[0002]
2A76:0013 268B0E0200     MOV    CX,ES:[0002]
2A76:0018 2E8B160400     MOV    DX,CS:[0004]
```



Siis ASSUME-käskyn ansiosta kääntäjä pystyy laittamaan segmentin vaihtokäskyt automaattisesti paikalleen. Mikäli edellä olisi puuttunut `ASSUME ES:lisa`, olisi käännös päättynyt virheeseen lauseen `MOV CX,neljas` osalta, koska mikään oletettu segmenttirekisteri ei olisi osoittanut segmenttiin josta muistipaikka `neljas` löytyy.

Kääntäjä, linkittäjä ja ohjelman lataava ohjelma huolehtivat segmenttien sijoittumisesta muistiin.

Edellisen esimerkkiohjelman kääntäminen aiheuttaa linkityksessä varoituksen pinosegmentin puuttumisesta. Yleinen tapa on kirjoittaa segmentit seuraavasti:

```
DOSSEG
STACK SEGMENT PARA STACK 'STACK'
 DB   400H DUP(?)
STACK ENDS

_DATA SEGMENT WORD PUBLIC 'DATA'
terve DB 'Terve!','$'
_DATA ENDS

_TEXT SEGMENT WORD PUBLIC 'CODE'
ASSUME CS:_TEXT,DS:_DATA
ohjelma PROC NEAR
 MOV  AX,_DATA
 MOV  DS,AX             ; DS := _DATA
 MOV  AH,09H
 MOV  DX,OFFSET terve
 INT  21H
 MOV  AX,4C00H
 INT  21H               ; ohjelman lopetus
ohjelma ENDP
_TEXT ENDS
 END ohjelma
```

Käskyllä `DOSSEG` (ei ehkä toimi kääntäjän vanhoissa versioissa) ilmoitetaan kääntäjälle, että segmentit järjestetään MS-DOSin mukaiseen järjestykseen. Sanat `PARA` ja `WORD` ilmoittavat, että segmenttien alku pakotetaan 16:lla (paragraph) tai 2:lla (word) jaollisiin osoitteisiin. Lainausmerkeissä oleva sana ilmoittaa luokan, johon segmentti kuulluu.

PUBLIC ilmoittaa, että segmentti on julkinen ja myös muista ohjelmista tulevat saman nimiset segmentit sijoitetaan samaan segmenttiin (Pascal, C, Assembler).

Edellisen esimerkin mukainen runko kannattaa kirjoittaa vaikkapa tiedostoon ALKU.ASM. Uuden assembler-ohjelman kirjoittaminen aloitetaan kopioimalla ALKU.ASM uuteen tiedostoon. Tämän jälkeen uutta koodia täydennetään kopioituun tiedostoon.

Mikäli kääntäjästä on käytössä uudempi versio, voidaan segmenttimääritykset lyhentää. Edellinen esimerkki voitaisiin kirjoittaa tällöin muotoon:

```
DOSSEG
.MODEL SMALL
.STACK
.DATA
terve DB 'Terve!','$'
.CODE
ohjelma PROC NEAR
 MOV  AX,_DATA
 MOV  DS,AX             ; DS := _DATA
 MOV  AH,09H
 MOV  DX,OFFSET terve
 INT  21H
 MOV  AX,4C00H
 INT  21H               ; ohjelman lopetus
ohjelma ENDP
 END ohjelma
```

Yleensä tosin DATA-segmenttiin viitataan muodossa MOV AX,@DATA.
Edellä kerrottu pätee, mikäli assembler-ohjelma käännetään .EXE-suorituskelpoiseksi ohjelmaksi. .COM-tiedostoksi käännettäessä täytyy segmentit määritellä hieman toisin. Samoin on, mikäli kirjoitetaan aliohjelmia. Näihin palataan myöhemmin.

4.4 Ohjelman kääntäminen

4.4.1 Ohjelman kirjoittaminen

Assembler-ohjelma on siis vasta ASCII-muotoinen esitys konekieliselle ohjelmalla. ASCII-tiedosto, jonka tarkennin on yleensä .ASM, voidaan kirjoittaa millä tahansa editorilla, jolla teksti voidaan tallettaa ASCII-muotoisena. Esimerkkinä Emacs, SideKick, Turbo Pascal- tai Turbo C -editori. Joissakin tekstinkäsittelyohjelmissa skandinaaviset merkit eivät talletu oikein (esim. WordStar) ja talletuksen jälkeen tiedosto täytyy ajaa jollakin muunnosohjelmalla.

4.4.2 Kääntäminen

Jotta tiedostosta saataisiin toteutuskelpoinen ohjelma, pitää tiedosto antaa assembler-kääntäjä -ohjelmalle käsiteltäväksi. Kääntäjä tekee tiedostosta .OBJ -tiedoston, joka sisältää konekielisen koodin lisäksi linkittämisessä tarvittavat tiedot segmenttien nimistä ja joidenkin muistipaikkojen sijainnista.

4.4.3 Linkittäminen

Toteteutuskelpoiseksi .OBJ tiedosto saadaan linkittäjä ohjelmalla. Tällöin muodostuvassa .EXE tiedostossa on ohjelman käynnistämistä varten tiedot aloitusosoitteesta, pino-osoittimen arvosta sekä muistipaikkojen relokoimiseksi tarvittava informaatio.

4.4.4 Muistinkuva

Joissakin erikoistapauksissa .EXE ohjelma voidaan vielä muuttaa täsmälleen ohjelman ajonaikaiseksi muistinkuvaksi, .COM -tiedostoksi. Muunnos voidaan tehdä joko EXE2BIN (Exe To Bin) -ohjelmalla tai esimerkiksi Turbo Assemblerin TLINK-ohjelmalla.

Tämä muunnos on kuitenkin mahdollista vain, mikäli .EXE-tiedosto ei sisällä tietoa aloitusosoitteesta, ohjelman koko on alle 64K tavua eikä DATA-segmentti sisällä alkuarvoja. Ohjelma on lisäksi täytynyt kirjoittaa mahdollisimman vapaasijoitteiseksi, sillä .COM tiedoston käynnistyksessä ohjelmalaskurin arvoksi tulee aina 100H .EXE tiedoston 0H:n sijaan.

4.4.5 MicroSoft Macro Assembler

MicroSoftin Macro Assembler on kaksivaiheinen kääntäjä. Tämä tarkoittaa sitä, että lähdekielinen tiedosto käydään kaksi kertaa lävitse. Ensimmäisellä läpikäynnillä pyritään kääntämään kaikki käskyt lukuunottamatta symbolisia viittauksia. Kaikkien nimiöiden paikat selvitetään. Toisella läpikäynnillä symbolisten viittausten paikalle sijoitetaan vastaavat osoitteet.

Eteenpäin viittauksissa tulee joskus hankaluuksia, mikäli kääntäjää ei avusteta muuttujien kokoa yms. kertomalla.

Mikäli edellisen kappaleen esimerkkiohjelma olisi kirjoitettu tiedostoon TERVE.ASM käännettäisiin ohjelma ajokelpoiseksi tiedostoksi seuraavasti:

```
MASM TERVE /Z;
LINK TERVE;
```

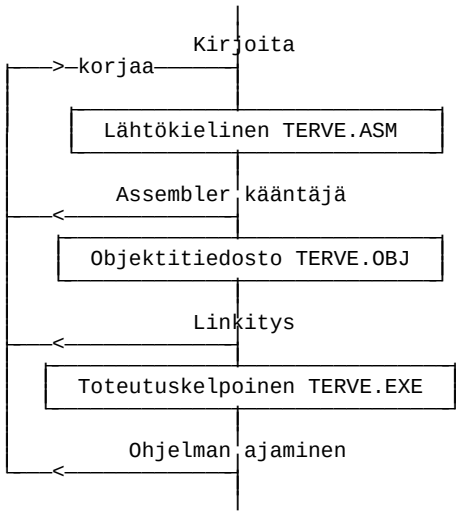
Ohjelma voitaisiin tämän jälkeen ajaa komennolla

```
TERVE
```

Edellä kääntäminen edellyttää, että assembler-kääntäjän hakemisto on ilmoitettu hakupolussa (PATH). Laskentakeskuksen koneissa tämä saadaan aikaan komennolla

```
ASETA MASM
```

Optiolla /Z saadaan virheelliset rivit tulostumaan näytölle käännöksen aikana. Puolipiste rivin lopussa estää lisäkysymysten (objektitiedoston nimi, list-tiedoston nimi ja map-tiedoston nimi) tulemisen ja käyttää oletusarvoja.
Optiolla /L voidaan haluttaessa tulostaa lisäksi .LST -tiedosto. .LST-tiedostosta näkyy käännetty konekielinen koodi, makrojen laajentuminen ja lista symbolisista nimistä aakkosjärjestyksessä. .LST-tiedoston avulla on helppo kirjoittaa INLINE-koodia. Muutenkin .LST-tiedoston tutkiminen on erittäin opettavaista.



4.4.6 Ajojono

Kääntämistä varten voidaan kirjoittaa EXE.BAT tiedosto:

```
ECHO OFF
REM    exe         - käännetään .ASM tiedosto .EXE tiedostoksi
REM ---------------------------------------------------------------------
REM    Kutsu:
REM      EXE tiedosto
REM
REM =====================================================================
REM
REM *********************************************************************
ECHO %ECHO%
IF NOT EXIST %1.ASM GOTO EI_ASM
C:\KIELET\MASM50\MASM %1,d:apu.obj/B30/Z %2 %3 %4 %5 %6 %7 %8;
IF ERRORLEVEL 1 GOTO POIS
C:\KIELET\MASM50\LINK d:apu.obj,%1;
IF ERRORLEVEL 1 GOTO POIS
DEL d:apu.OBJ
GOTO END
:EI_ASM
 ECHO EI .ASM tarkenninta
GOTO POIS
:POIS
  ECHO   ****** ERRORLEVEL 1 ******
:END
```

Ajojonon avulla kääntäminen voidaan kokonaisuutena suorittaa komennolla:

```
EXE terve
```

Esimerkin ajojono tekee välitiedoston .OBJ RAM-levylle D:. Välitiedosto voidaan ohjata myös muuallekin.

4.4.7 Borland Turbo Assembler

Borlandin Turbo Assembler on Macro Assemblerin kanssa yhteensopiva yksivaiheinen kääntäjä. Näin ollen se on aavistuksen nopeampi kuin Macro Assembler. Suurin etu Turbo Assemblerista saadaan kuitenkin sen helppokäyttöisyydestä yhdessä Turbo Debuggerin kanssa.

Kääntäminen Turbo Assembler-kääntäjällä:

```
TASM terve /z /zi;
TLINK terve /v;
```

Optiolla /zi ja /v saadaan tiedostoihin täydellinen symbolinen informaatio ohjelmasta Turbo Debugger-ohjelmaa varten. Lisätietoa kääntäjän ja linkkerin optioista saa kirjoittamalla pelkän ohjelman nimen. Käännöksen jälkeen ohjelmaa voidaan ajaa Turbo Debugger-ohjelmalla komennolla:

```
TD terve
```

Yliopiston mikroissa Turbo Assembler saadaan polkuun komennolla

```
ASETA TASM
```

ja Turbo Debugger komennolla

```
ASETA TD
```

4.5 Muistimallit

C-ohjelmoija joutuu 8086-prosessoreita ohjelmoidessaan päättämään mitä muistimallia hänen ohjelmansa vastaa. Yksinkertaistettua segmenttirakennetta käytettäessä sama päätös pitää tehdä myös assembler-kielessä. Muistimalli tarkoittaa sitä, minkälaisia osoittimia pitää käyttää eri muistialueisiin viitattaessa.

Muistimallista riippumatta dynaamisia muuttujia voidaan varata ja käyttää.

Yleensä esiintyy 6 erilaista muistimallia.

4.5.1 Tiny

Sekä koodi että data mahtuvat samaan 64 kilotavun segmenttiin. .COM-ohjelmien täytyy olla tätä tyyppiä.

4.5.2 Small

Data mahtuu omaan 64 kilotavun segmenttiin ja koodi toiseen 64 kilotavun segmenttiin. Lyhyet osoittimet riittävät molempien osoittamiseen. Eräs yleisimmistä muistimalleista pienissä ohjelmissa. Erikoisjärjestelyin tämäkin muistimalli voidaan kääntää .COM-ohjelmaksi.

4.5.3 Medium

Data 64 kilotavua. Koodi yli 64 kilotavua. Siis datalle riittää lyhyet osoittimet, mutta koodissa siirtymisessä tarvitaan `FAR CALL` ja `FAR JMP` -käskyjä.

4.5.4 Compact

Koodi 64 kilotavun segmentissä. Data yli 64 kilotavua. Siis normaalit hypyt riittävät ohjelmassa, mutta datan osoittamiseen tarvitaan pitkiä osoittimia. Staattisten muuttujien yhteinen koko täytyy kuitenkin olla alle 64 kilotavua.

4.5.5 Large

Sekä koodi että data yli 64 kilotavua. Staattisten muuttujien yhteinen koko täytyy kuitenkin olla alle 64 kilotavua.

4.5.6 Huge

Kuten edellä, mutta staattisia muuttujia voi olla yli 64 kilotavua.

Luku 5

Ohjelman suunnittelu

Assembler-kielisessä ohjelmassa joudutaan käyttämään yhtä toimenpidettä varten useita konekielisiä käskyjä. Tämän takia ohjelmoijan täytyy aina ennen ohjelman kirjoittamista hahmottaa tarkasti ohjelmoitava kokonaisuus.

5.1 Tehtävän tarkennus

Ensimmäinen vaihe on tehtävän määrityksen tarkennus siten, ettei määritykseen jää mitään epämääräisyyksiä. Myös esille tulevat erikoistapaukset on syytä miettiä tarkoin.

Usein kannattaa ennen ohjelmoinnin aloittamista kirjoittaa esimerkkejä ohjelman suorituksesta eri syöttöarvoilla. Näin saadaan kuva ratkaisualgoritmin luonteesta ja erikoistapauksista.

5.2 Algoritmi

Kun ratkaisualgoritmi on selvillä, kirjoitetaan se yksityiskohtaisesti näkyville luonnollisella kielellä. Tässä vaiheessa ei vielä tarvitse välittää lainkaan siitä, mitä prosessoria tullaan käyttämään. Mikäli ongelma voidaan ratkaista selvästi erilaisilla algoritmeilla, voidaan kirjoittaa näkyville myös muita vaihtoehtoisia algoritmeja.

Luonnollisella kielellä kirjoitettu algoritmi kannattaa säästää ja kirjoittaa jopa ohjelman kommentteihin. Samoin esimerkit eri syöttöaineistosta ja tuloksista voidaan liittää kommentteihin.

5.3 Muuttujat

Ratkaisualgoritmissa tulee usein joukko muuttujia joita tarvitaan. Osa muuttujista on syöttönä tulevia muuttujia, osa tulostusmuuttujia sekä osa ongelman ratkaisussa tarvittavia apumuuttujia. Kun ongelmaa ruvetaan kirjoittamaan tietylle prosessorille, mietitään miten on edullisinta käyttää muuttujia. Aluksi tutkitaan mitä rekistereitä ongelman ratkaisun aikana tarvitaan. Mikäli rekisterit riittävät, kannattaa kaikki apumuuttujat säilyttää rekistereissä. Rekistereitä käytettäessä on syytä muistaa, että niiden muuttaminen saattaa aiheuttaa virheitä kutsuvan ohjelman toimintaan. Tämän takia on huolellisesti kommentoitava ne rekisterit, joiden arvo muuttuu suorituksen aikana. Mahdollisesti rekistereiden arvot kannattaa tallettaa suorituksen ajaksi esimerkiksi pinoon.

5.4 Kirjoittaminen

Kun ongelma ja sen tarvitsemat muuttujat on dokumentoitu algoritmiseen muotoon, voidaan aloittaa ongelman koodaaminen assembler-kielelle. Kukin algoritmin askel kirjoitetaan omaksi lohkokseen ohjelmassa tai jopa omaksi aliohjelmakseen tai makroksi. Mikäli selvästi osoittautuu, että pieni algoritmin muutos saattaa johtaa selvästi lyhyempään tai no-

peampaan koodiin käytettävällä prosessorilla, voidaan algoritmia tietysti hieman modifioida.

5.5 Testaaminen

Mikäli ongelma on jäsennetty hyvin, voidaan kukin kokonaisuus testata omana yksikkönään pienen pääohjelman avulla ja näin varmistua palasten toimivuudesta ennen kokonaisuuden kasaamista.

5.6 Esimerkki

5.6.1 Tehtävä

Tutkitaan esimerkiksi seuraavaa ongelmaa: Etsittävä kokonaislukutaulukosta suurin poikkeama taulukon keskiarvosta.

5.6.2 Algoritmi

Tehtävää analysoitaessa on ratkaisualgoritmi suhteellisen selkeä: eniten keskiarvosta poikkeaa joko taulukon suurin tai pienin luku.

Koska
keskiarvo <= suurin ja
keskiarvo >= pienin,
on suurin poikkeama joko (suurin-keskiarvo) tai (keskiarvo-pienin). Tämä voidaan kirjoittaa selkeäksi algoritmiksi:

```
1. Lasketaan keskiarvo.
2. Etsitään suurin ja pienen alkio.
3. Vastaus := suurempi luvuista suurin-ka ja ka-pienin
```

5.6.3 Algoritmin tarkistus

Edellinen algoritmi olisi jo varsin hyvä alku tehtävän ratkaisulle. Tarkennettaisiin vain kukin askel vielä erikseen omaksi algoritmikseen. Nopeimpaan mahdolliseen koodiin pyrittäessä huomattaisiin kuitenkin, että sekä kohdassa 1. että kohdassa 2. täytyisi kummassakin käydä taulukko kokonaan lävitse. Tämän takia algoritmin kohdat 1. ja 2. voitaisiin ehkä yhdistää:

```
1.  Lasketaan summa, etsitään suurin ja pienin.
2.  KA:=summa/lkm
3.  vastaus := MAX(suurin-KA,KA-pienin)
```

5.6.4 Erikoistapaukset

Algoritmissa tulee muutamia erikoistapauksia. Mikäli alkioita ei ole, mikä on vastaus? Voidaan sopia, että tällöin poikkeama on 0.

Kohdassa 2 suoritetaan jakolasku. Korkeamman tason kielellä jakolasku ei aiheuta mitään ongelmaa, koska tulos voitaisiin laskea reaalilukumuuttujaan. Konekielellä sen sijaan ongelmia aiheuttaa tuloksen pyöristys. Katkaistaanko tulos kokonaisluvuksi vai pyöriste-

täänkö ylöspäin? Voidaan yksinkertaisuuden vuoksi sopia, että katkaisu riittää. Ilman tarkempaa katkaisun aiheuttaman virheen analysointia täytyy kuitenkin todeta, että tulokseen saattaa tulla +/-1 kokoinen virhe.

Lisäksi summa pitää ehkä laskea 32-bittisenä, jollei ole tarkempaa tietoa lukujen suuruudesta ja lukumäärästä. Luvut tulee käsitellä etumerkillisinä, koska toisin ei ole sovittu.

5.6.5 Algoritmin tarkennus

Algoritmin kohta 1. kaipaa ehkä vielä hieman tarkentamista:

```
1.1  suurin := 1. alkio;
     pienin := 1. alkio;
     summa  :=      0;
1.2. lopetetaan mikäli taulukko on tyhjä
1.3. summa  := summa+kohdalla oleva alkio;
     suurin := MAX(suurin,alkio);
     pienin := MIN(pienin,alkio);
1.4. siirrytään seuraavaan alkioon
1.5. jatketaan kohdasta 1.3, mikäli alkioita jäljellä
```

5.6.6 Muuttujat

Nyt voidaan tutkia muuttujien käyttöä. Toistaiseksi on määrittelemättä mistä edes syöttö saadaan. Tämä riippuu tietenkin siitä, onko ohjelma tulossa jonkin korkeamman tason kielellä kirjoitetun ohjelman aliohjelmaksi vai pelkästään itsenäisesti toimivaksi assembler-ohjelmaksi. Ongelma voidaan siirtää myöhäisempään vaiheeseen, mikäli keskitytään kirjoittamaan assembler-kielistä aliohjelmaa, joka saa tietyn syötön ja palauttaa tuloksen. Tällöin voimme itse määritellä mistä syöttö saadaan ja mihin tulos laitetaan. Todelliseen ympäristöön aliohjelma saadaan kirjoittamalla sitä kutsuva apuohjelma.

Sovitaan, että syöttövektori alkaa DS:SI:n osoittamasta paikasta ja että siinä on CX kappaletta alkioita. Tulos palautetaan rekisterissä AX.

Algoritmin aikana tarvitaan seuraavat muuttujat:

```
 1. osoitin kohdalla olevaan alkioon
 2. kohdalla oleva alkio
 3. summa
 4. suurin
 5. pienin
 6. laskuri montako alkiota on käsitelty
 7. lkm
 8. keskiarvo
 9. suurin-ka
10. ka-pienin
11. ero
```

5.6.7 Muuttujat rekistereille

Jaetaan prosessorin rekisterit edellä oleville muuttujille:

```
 1. osoittimena  DS:SI
 2. alkiona      AX
 6. laskurina    CX
 3. summana      DX BX (32 bit)
 4. pienin       DI
 5. suurin       BP
 7. lkm          CX (kun laskuria ei enää tarvita)
 8. keskiarvo    AX (kun alkiota ei enää tarvita)
 9. suurin-ka    BP
10. ka-pienin    AX (kun keskiarvoa ei enää tarvita)
11. ero          AX (lopussa)
```

Siis prosessorin rekisterit riittävät juuri ja juuri kaikkien muuttujien käsittelyyn, kun otetaan huomioon, että kaikkia muuttujia ei tarvita yhtäaikaa. Nytkin joudutaan toisin esimerkiksi lukumäärää vastaava CX tallettamaan osaksi ajaksi johonkin apupaikkaan. Pino on luonnollinen paikka tällaisille hetken käyttämättömille rekistereille. Mikäli rekisterit eivät olisi riittäneet, pitäisi ehkä pinoon allokoida tilaa apumuuttujille. Pinon käytöstä on toisaalla eri ohjelmointikielien kohdalla.

5.6.8 Koodaus

Nyt algoritmia voidaan ruveta koodaamaan assembler-kielelle. Kussakin vaiheessa seuraavan alkion ottaminen ja uuteen siirtyminen on nopeinta tehdä LODSW-käskyllä. LOOP-käsky sopii ehkä parhaiten silmukan toteuttamiseen. Osaksi käytettävät rekisteritkin valittiin näitä käskyjä silmällä pitäen.

5.6.9 MIN ja MAX

MAX ja MIN operaatiot muodostavat algoritmissa selvän oman kokonaisuutensa. Ne siis kannattaa kirjoittaa joko omaksi aliohjelmakseen tai omiksi makroikseen. Tässä tapauksessa makro on ehkä käyttökelpoisempi, koska tällöin tulos saadaan helposti mihin rekisteriin tahansa. Kirjoitetaan aluksi vaikkapa MAX-makro:

```
;********************** MAX ******************************
MAX MACRO x1,x2,tulos
LOCAL suurempi
;
; Makrolla sijoitetaan rekisteriin tai muistipaikkaan tulos
; suurempi etumerkillisistä kokonaisluvuista  x1 ja x2.
;
; Input:   x1,x2  (rekisteri, muistipaikka, vakio)
; Output:  tulos  (rekisteri tai muistipaikka)
; Muuttuu: tulos,liput
;
; Esimerkki: MAX AX,BX,AX        kääntyy:  CMP  AX,BX
;                                          JGE  suurempi
;                                          MOV  AX,BX
;                                         suurempi:
;
;            MAX ES:[DI],10,DX   kääntyy:  MOV  DX,ES:[DI]
;                                          CMP  DX,10
;                                          JGE  suurempi
;                                          MOV  DX,10
;                                         suurempi:
;
; HUOM! Mikäli jompi kumpi verrattavista on jo tulos, täytyy
; se laittaa kutsussa x1:en paikalle!
;
 IFDIFI <x1>,<tulos> ;; mikäli x1 ei ole valmiiksi jo tulos
  MOV  tulos,x1      ;; alkuarvaus:  tulos:=x1
 ENDIF
 CMP  tulos,x2       ;; tulos=x1, onko tulos >= x2
 JGE  suurempi       ;; jos on, niin valmis
 MOV  tulos,x2       ;; muuten tulos:=x2
suurempi:
ENDM                 ;; MAX
```

Tehtävä 5.1 Minimi

Kirjoita edellistä vastaava makro `MIN`.

Tehtävä 5.2 Sijoita

Makrot `MIN` ja `MAX` ovat tosin niin samanlaisia, että voitaisiin kirjoittaa makro, jota kutsuttaisiin seuraavasti:

```
sijoita JGE,AX,10,BX
```

Kirjoita makro `SIJOITA`.

5.6.10 Makrojen testaus

Mikäli makrot MIN ja MAX kirjoitettaan tiedostoon MIN_MAX.ASM, voidaan ne myöhemmin ottaa ohjelmaan INCLUDE-käskyllä. Makrojen oikeellisuus voidaan testata vaikkapa seuraavalla pienellä pääohjelmalla:

```
INCLUDE MIN_MAX.ASM
DOSSEG
.MODEL SMALL
.STACK
.CODE
 MAX AX,BX,AX
 MIN 10,20,DX
 MAX ES:[DI],DS:[SI],AX
 END
```

Edellinen pääohjelma ei tietenkään tee mitään, eikä sitä saa ajaa! Oikeellisuus tarkistetaankin joko .LST-tiedostosta tai debuggerilla käännetystä koodista.

5.6.11 Poikkeama

Nyt voidaan kirjoittaa varsinainen aliohjelma POIKKEAMA:

```
INCLUDE MIN_MAX.ASM
;****************************** poikkeama ********************************
poikkeama PROC NEAR
;
; Aliohjelmalla etsitään paljonko kokonaislukutaulukon
; suurin tai pienin alkio poikkeaa taulukon keskiarvosta.
;
;  Input:     DS:SI   osoittaa taulukon ensimmäiseen alkioon
;             CX      taulukon alkioiden lukumäärä
;
;  Output:    AX      MAX(suurin-KA,KA-pienin)
;
;  Muuttuu:   AX, liput
;
;  Algoritmi: 1. Lasketaan summa, etsitään suurin ja pienin:
;                  1.1  alustetaan suurin := 1. alkio;
;                                  pienin := 1. alkio;
;                                  summa  :=      0;
;                                  ero    :=      0;
;                  1.2. mikäli taulukko on tyhjä jatketaan kohdasta 4.
;                  1.3. summa  := summa+kohdalla oleva alkio;
;                       suurin := MAX(suurin,alkio);
;                       pienin := MIN(pienin,alkio);
;                  1.4. siirrytään seuraavaan alkioon
;                  1.5. jatketaan kohdasta 1.3,
;                       mikäli alkioita jäljellä
;             2. KA  := summa DIV lkm;
;             3. ero := MAX(suurin-KA,KA-pienin);
;             4. palautetaan ero
;
;  Rekistereiden käyttö:
;                osoittimena  DS:SI
;                alkiona      AX
;                laskurina    CX
;                summana      DX BX (32 bit)
;                pienin       DI
;                suurin       BP
;                lkm          CX (kun laskuria ei enää tarvita)
;                keskiarvo    AX (kun alkiota ei enää tarvita)
;                suurin-ka    BP
;                ka-pienin    AX (kun keskiarvoa ei tarvita)
;                ero          AX (lopussa)
;
; Vesa Lappalainen 27.2.1989
;====================================================================
 push_reg <BX,DX,DI,SI,BP>
;
;====================================================================
; Lasketaan summa ja etsitään pienin sekä suurin alkio
;----------------------------; Alustetaan muuttujat:
 XOR  AX,AX                  ; ero := 0
 MOV  BP,[SI]                ; Suurin
 MOV  DI,BP                  ; ja pienin := 1. alkio
 MOV  DX,AX                  ; summa := 0.
```

```
 MOV  BX,AX
;--------------------------- ; Jollei alkioita ole, on ero valmis.
 JCXZ ero_valmis
 PUSH CX                    ; lkm pinoon talteen.
 CLD                        ; Taulukkoa ylöspäin.
;--------------------------- ; Lasketaan summa, etsitään ja edetään:
summaa_ja_etsi:
 LODSW                      ; Otetaan alkio ja siirrytään seuraavaan.
 ADD  BX,AX                 ; summa:=summa + alkio.
 ADC  DX,0
 MAX  BP,AX,BP              ; suurin:=MAX(suurin,alkio).
 MIN  DI,AX,DI              ; pienin:=MIN(pienin,alkio).
 LOOP summaa_ja_etsi        ; Jos alkiota, jatketaan.
;=============================================================
; Lasketaan keskiarvo:
 POP  CX                    ; lkm pois pinosta
 MOV  AX,BX                 ; DX AX = summa, eli jako menee oikein!
 IDIV CX                    ; AX = keskiarvo, DX jakojäännös
;=============================================================
; Lasketaan ero:
 SUB  BP,AX                 ; BP := suurin-ka
 SUB  AX,DI                 ; AX := ka-pienin
 MAX  AX,BP,AX              ; AX := MAX(suurin-ka,ka-pienin)
;=============================================================
ero_valmis:                 ; palautetaan AX:ssä ero
 pop_reg  <BP,SI,DI,DX,BX>
 RET
poikkeama ENDP
;*****************************************************************
```

Huomattakoon, että ohjelman kirjoittaminen on aloitettu kirjoittamalla alkukommentit, push_reg ja pop_reg makrojen kutsut (parametrit helppo kirjoittaa kun kutsut vielä lähekkäin) sekä RET ja ENDP rivit. Tämän jälkeen koodiin on täydennetty kaksoisviivojen alla olevat algoritmin mukaiset kommentit ja vasta sitten täydennetty näiden kommenttien väliin varsinaisia konekielisiä käskyjä.

Ohjelmoijan on pysyttävä selvillä rekistereiden arvoista ja käytöstä koko ohjelman kirjoittamisen ajan. Kirjoitettaessa esimerkiksi aliohjelma- tai makrokutsua, pitää vastaavasta esittelystä tarkistaa, ettei aliohjelma tai makro tee yllättäviä muutoksia rekistereihin tai muistipaikkoihin.

Haluttaessa voitaisiin myös määritellä rekistereille symboliset nimet:

```
alkio      EQU  AX
laskuri    EQU  CX
summa_lo   EQU  BX
summa_hi   EQU  DX
pienin     EQU  DI
suurin     EQU  BP
lkm        EQU  CX
 ...
 ADD  summa_lo,alkio
 ADC  summa_hi,0
 MAX  suurin,alkio,suurin
```

Tässä on kuitenkin se vaara, että samaa rekisteriä käytetään mahdollisesti eri tarkoituksiin **samanaikaisesti**. Ohjelmakoodi tosin tulee enemmän itsedokumentoivaksi. Päällekkäiskäytön vaaraa pienennetään, mikäli esimerkiksi määritys `lkm EQU CX` tehtäisiin vasta lauseen `POP CX` jälkeen.

5.7 Esimerkin testaaminen

5.7.1 Apualiohjelmat

Edellinen aliohjelma vaatii vielä testaamisen. Tätä varten on jo aiemmin kirjoitettu ja testattu tiedosto WORD.ASM, jossa on kokonaisluvun tulostamiseen ja lukemiseen liittyviä aliohjelmia:

```
;*************************** muuta AX ASCII ****************************
muuta_AX_ASCII PROC NEAR
;
; Aliohjelmalla muutetaan AX:ssä oleva positiivinen kokonaisluku
; 5 merkkiseksi ASCII-jonoksi alkaen paikasta ES:DI.
;
; Input:      AX       - muutettava kokonaisluku
;             ES:DI    - osoitin tuloksen alkuun
; Output:     ES:[DI]  - 5 merkkiä
; Muuttuu:    ES:[DI],liput
;
; Tulos on aina 5 merkkiä ja etunollat ovat mukana.
;
; Algoritmi:  1. Aloitetaan viimeisestä merkistä.
;             2. Jaetaan luku 10:llä.
;             3. Muutetaan jakojäännös ASCII-merkiksi.
;             4. Talletetaan merkki.
;             5. Jatketaan 1:stä kunnes tehty 5 kertaa.
;
  push_reg <AX,BX,CX,DX>
  MOV  CX,5                           ; Silmukkalaskuri
  ADD  DI,CX                          ; Aloitetaan viimeisestä merkistä
  MOV  BX,10                          ; Jakaja 10
edellinen_merkki:
  DEC  DI                             ; Edellinen merkki
  CWD                                 ; AX -> DX AX
  DIV  BX                             ; jakojäännös DX:ään kok.osa AX:ään
  OR   DL,30H                         ; DL kirjaimeksi '0'-'9'
  MOV  ES:[DI],DL                     ; Talletetaan kirjain
  LOOP edellinen_merkki               ; Toistetaan 5 kertaa.
  pop_reg <DX,CX,BX,AX>
  RET
muuta_AX_ASCII ENDP
;***********************************************************************

;*************************** lue int ***********************************
lue_int PROC NEAR
;
; Aliohjelmalla luetaan päätteeltä kokonaisluku rekisteriin AX.
; Luvun lukeminen lopetetaan, kun tulee ensimmäinen ei numero.
;
; Input:      pääte
; Output:     AX,         - tulos AX:ään
;             pääte       - merkit kaiutetaan
;             BL          - viimeksi luettu merkki
; Muuttuu:    AX, BX, näyttö
;
; Mikäli merkkejä annetaan siten, että niistä muodostuisi yli 0FFFFH
; ei aliohjelma toimi oikein!
;
; Algoritmi:       1. Asetetaan luku:=0.
;                  2. Luetaan numero.
;                  3. Jollei numero laillinen, niin lopetetaan.
;                  4. luku:=luku*10+numero.
;                  5. Jatketaan 2.
;
 push_reg <CX,DX>
 MOV  CX,10                  ; Kerroin.
 MOV  BX,0                   ; Luku:=0;
 MOV  AX,0
seuraava_numero:
 MS_DOS read_kbd_and_echo    ; Luetaan merkki AL.ään.
 XCHG BX,AX                  ; Nykyinen luku AX:ään ja merkki BL:ään.
 CMP  BL,'0'                 ; Onko kelvollinen numero
 JB   vaara_merkki
 CMP  BL,'9'
 JA   vaara_merkki
 AND  BX,000FH               ; On kelvollinen, muutetaan luvuksi
 MUL  CX                     ; Luku:=luku*10
 ADD  BX,AX                  ;               + numero
 JMP  seuraava_numero
vaara_merkki:
 pop_reg <DX,CX>
 RET
lue_int ENDP
;***********************************************************************

;*************************** lue vektori *******************************
lue_vektori PROC NEAR
;
; Aliohjelmalla luetaan päätteeltä paikkaan ES:DI kokonaislukuvektori,
; johon tulee korkeintaan CX kappaletta alkioita.  Lukeminen lopetetaan
; RETURN merkkiin.  Alkioiden väliin mikä tahansa ei numero.  Alkioiden
; lukumäärä palautetaan rekisterissä CX.
;
; Input:      ES:DI          - vektorin alkuosoite
;             CX             - alkioiden maksimimäärä
```

```
; Output:     ES:[DI]        - vektorin arvot
;             CX             - luettujen alkioiden lukumäärä
;             näyttö         - arvot kaiutetaan näytölle
; Muuttuu:    ES:[DI], CX, näyttö
;
; Palautetaan aina vähintää yksi arvo (=0 jos heti RETURN).
;
; Algoritmi:  1.  Nollataan laskuri.
;             2.  Luetaan luku.
;             3.  Talletetaan luku.  Lisätään laskuria.
;             4.  Mikäli viimeinen merkki CR lopetetaan.
;             5.  Toistetaan kohdasta 2. kunnes korkeintaan max.määrä luettu.
;
 push_reg <AX,BX,DX,DI>
 MOV  DX,0                  ; Laskuri:=0;
lue_seuraava:
 CALL lue_int
 CLD                        ; Varulta nollataan suuntalippu (MS-DOS?)
 STOSW                      ; Talletetaan luku ja siirrytään seuraavaan.
 INC  DX                    ; Lisätään laskuria.
 CMP  BL,cr                 ; Oliko viimeinen merkki RETURN?
 LOOPNE lue_seuraava        ; jos oli tai tehty CX kertaan niin loppu.
 MOV  CX,DX                 ; Lukujen määrä CX:ään
 pop_reg  <DI,DX,BX,AX>
 RET
lue_vektori ENDP
```

Edellä olevat ohjelmat eivät ole täydellisiä, ne eivät esimerkiksi käsittele etumerkkiä lainkaan. Myös etunollien poisjättäminen puuttuu aliohjelmasta muuta_AX_ASCII. Kuitenkin niiden avulla voidaan lukea ja kirjoittaa kokonaislukuja.

5.7.2 Pääohjelma

Aliohjelma poikkeama voidaan testata vaikkapa seuraavalla pääohjelmalla:

```
DOSSEG
INCLUDE MAKROT.ASM
.MODEL TINY
.STACK
.DATA
max_koko  EQU 10
vektori   DW  max_koko DUP(?)
alkioita  DW  max_koko
viesti    DB  cr,lf,'Suurin poikkeama on '
max_ero   DB  '   '
loppu     DB  ' +/-1.',cr,lf,'$'
.CODE
INCLUDE POIKKEAMA.ASM
INCLUDE WORD.ASM
poikkeama_testi PROC NEAR
 MOV  AX,@DATA              ; DS,ES := DATA
 MOV  DS,AX
 MOV  ES,AX
 LEA  DI,vektori            ; Vektorin alkuosoite DI:hin
 MOV  CX,alkioita           ; ja alkioiden lukumäärä CX:ään.
 CALL lue_vektori           ; Luetaan vektoriin alkiot päätteeltä.
 LEA  SI,vektori            ; Vektorin alkuosoite SI:hin
 CALL poikkeama             ; Lasketaan poikkeama keskiarvosta.
 LEA  DI,max_ero            ; Muutetaan AX merkkijonoksi
 CALL muuta_AX_ASCII        ; paikkaan max_ero.
 tulosta_jono DS,viesti     ; Tulostus: Suurin poikkeama 00018 +/-1.
 MS_DOS lopetus,0           ; Ei virhettä.
poikkeama_testi ENDP
 END poikkeama_testi
```

Koodin lyhentämiseksi edellä on käytetty yksinkertaistettua segmenttien määritystä. Mikäli käytössä on vanhempi kääntäjä, täytyy segmenttimääritykset kirjoittaa täydellisesti. MAKROT.ASM tiedostoon on kasattu joukko jo aiemmin esitettyjä ja myöhemmin esitettäviä makroja.

Koska ohjelma on puhtaasti testiohjelma, ei se ole suinkaan täydellisesti dokumentoitu.

Ohjelmaa ei ehkä kannata heti kääntämisen jälkeen ajaa, vaan sitä kannattaa tutkia jollakin debugger-ohjelmalla. Tätä käsitellään seuraavassa luvussa.

Luku 6

Debuggerit

6.1 Yleistä

Debug tarkoittaa "bug"ien poistamista. Vapaata käännöstä "luteen poistin" ehkä parempi suomenkielinen termi on virheenjäljitin. Yleensä debug-ohjelmilla voidaan ajaa ohjelmaa käsky kerrallaan (trace), tulostaa (dump) ja muuttaa (enter) muistipaikkojen arvoja sekä laittaa pysäytyskohtia (breakpoint) koodin sekaan. Näin vältytään tulostuslauseiden lisäämisestä koodiin ja mahdollisesti pitkällisestä uudelleen kääntämisestä.

Symbolinen debuggeri tarkoittaa sitä, että muistipaikoille voidaan käyttää niille annettuja symbolisia nimiä suorien osoitteiden sijasta (joita voidaan myös käyttää tarvittaessa). Mikäli symbolista debuggeria ei ole käytössä, käytetään yleensä apuna kääntäjän tekemää .LST-tiedostoa.

Debuggerin käyttö ei rajoitu pelkästään virheen etsimiseen valmiista ohjelmasta, vaan sillä voidaan myös testata hyvinkin epätäydellisiä ohjelman osia, mikäli ne ovat syntaktisesti oikein ja näin ollen pystytään kääntämään.

6.2 Debug

Debug on MS-DOSin mukana tuleva yksinkertainen (ja pieni) virheenjäljitin. Lukujärjestelmänä käytetään ainoastaan heksalukuja. Ohjelma on rivipohjainen ja näin ollen sopii käytettäväksi millä laitteella tahansa.

6.2.1 Komennot

Komentohoputteena (prompt) on miinus-merkki (-). Ohjelman komennot ovat yksikirjaimisia:

```
A [osoite]            Assemble - käännä koodia muistiin
C alue osoite         Compare - vertaa muistialueita
D [alue]              Dump - tulosta muistia
E alue [lista]        Enter - sijoita arvoja muistiin
F alue lista          Fill - täytä muistia arvoilla
G[=osoite]{osoite}    Go - ajaa ohjelmaa
H arvo1 arvo2         Hex - tulostaa arvo1+arvo2 ja arvo1-arvo2
I portti              Input - lukee portin arvon
L [osoite[l:t1 t2]]   Load - lukee levyltä muistiin
M alue osoite         Move - kopioi muistia paikasta toiseen
N nimi {nimi}         Name - seuraavaksi käytettävä nimi
O arvo portti         Output - sijoittaa arvon porttiin
Q                     Quit - poistuu ohjelmasta
R [rekisteri]         Register - tulostaa kaikkien rekistereiden arvon tai pelkästään yhden rekisterin, jolloin arvoa voidaan myös muuttaa
S alue lista          Search - etsii arvoja muistista
T[=osoite][määrä]     Trace - ajaa ohjelmaa askel kerrallaan
U [alue]              Unassemble - kääntää assembler koodiksi muistia
W [osoite[l:t1 t2]]   Write - kirjoittaa muistia levylle
```

Edellä hakasulut tarkoittavat, että niitä ei kirjoiteta, vaan niiden välinen osa joko kirjoitetaan tai voidaan jättää kirjoittamatta. Aaltosulut tarkoittavat, että niiden välinen osa voidaan jättää kirjoittamatta tai kirjoittaa useita kertoja. Muut selitykset:

	syntaksi	esimerkki		
osoite	[segment:]offset	100	DS:100	1234:100
alue	osoite,offset	100,120	CS:120,12F	400:10,20
alue	osoite Lpituus	100 L20	CS:120 L10	400:10 L11
lista	{tavu|mjono}	10 20	'Apua.'	'Apu' 61 '.'
arvo	tavu	10		
määrä	arvo	20		
nimi	tiedoston nimi	oma.dat		
portti	portin numero	61		

Alla on esimerkki debug-ohjelman käytöstä. Käyttäjän kirjoittama osa on vahvennettu. Puolipiste ja sen jälkeinen osa on kirjoitettu jälkeenpäin kommentiksi, eikä sitä saa kirjoittaa oikeasti.

6.2.2 Ajoesimerkki

```
E:\TKJ\MONISTE\ASM>debug
-d     ; tulostuu 128 tavua muodossa: osoite heksana ja asciina
2515:0100  24 89 16 14 24 A3 12 24-EB 5F E8 C7 FE 0B C0 75   $...$..$._.....u
2515:0110  58 9A AA 0C 78 31 EB B5-EB 4F E8 B7 FE 0B C0 74   X...x1...O.....t
2515:0120  41 C6 06 D4 20 50 C6 06-D5 20 67 C6 06 D6 20 2E   A... P... g... .
2515:0130  BF D7 20 B8 DE 20 3B C7-76 22 8B 1E 8D 4E 80 3F   .. .. ;.v"...N.?
2515:0140  20 72 19 8A 07 88 05 47-FF 06 8D 4E A1 8D 4E 3B    r.....G...N..N;
2515:0150  06 91 3D 72 DE 9A 7D 0D-78 31 EB D7 C6 05 00 EB   ..=r..}.x1......
2515:0160  08 9A 95 0D 78 31 E9 3A-FF 8D 46 F6 50 9A BA 22   ....x1.:..F.P.."
2515:0170  78 31 59 C6 06 FD 22 01-C7 06 7A 21 FF FF B8 0F   x1Y..."...z!....
-e 100 12 13       ; sijoitetaan muistiin tavut 12H ja 13H alk. 100H:sta
-f 102 L11 FF 00   ; sijoitetaan 17 (11H) tavua jonoa FF 00
-d 100,11f         ; tulostetaan muistia
2515:0100  12 13 FF 00 FF 00 FF 00-FF 00 FF 00 FF 00 FF 00   ................
2515:0110  FF 00 FF 0C 78 31 EB B5-EB 4F E8 B7 FE 0B C0 74   ....x1...O.....t
-e 102             ; jos listaa ei anneta, kysytään kukin tavu erikseen
2515:0102  FF.     00.2    FF.1 ; välilyönnillä seuraavaan, ENTER lopettaa
-f 120,13f 'Apua.' ; täytetään väli 120H-13FH jonolla Apua.
-d100,14F          ; tulostetaan muutos
2515:0100  12 13 FF 02 01 00 FF 00-FF 00 FF 00 FF 00 FF 00   ................
2515:0110  FF 00 FF 0C 78 31 EB B5-EB 4F E8 B7 FE 0B C0 74   ....x1...O.....t
2515:0120  41 70 75 61 2E 41 70 75-61 2E 41 70 75 61 2E 41   Apua.Apua.Apua.A
2515:0130  70 75 61 2E 41 70 75 61-2E 41 70 75 61 2E 41 70   pua.Apua.Apua.Ap
2515:0140  20 72 19 8A 07 88 05 47-FF 06 8D 4E A1 8D 4E 3B    r.....G...N..N;
-c 100 L5 120      ; verrataan aluetta 100-104 ja 120-124
2515:0100  12  41  2515:0120  ; tulostuu eroavat tavut
2515:0101  13  70  2515:0121
2515:0102  FF  75  2515:0122
2515:0103  00  61  2515:0123
2515:0104  FF  2E  2515:0124
-c 120 L5 125      ; verrataan aluetta 120-124 ja 125-129. Ei eroja!
-i 61              ; luetaan portin 61H arvo
38
-h 38 03           ; Tulostetaan 38H+03H ja 38H-03H
003B  0035
-o 61 3b           ; Jos portin 61 bitit 1 ja 0 päällä, niin kaiutin piippaa
-o 61 38           ; Lopetetaan piippaus
-s 100,140 'Apua.' ; Etsitään väliltä 100-140 jonoa Apua.
2515:0120          ; Tulostuu löytöjen alkupaikat
2515:0125
2515:012A
2515:012F
2515:0134
2515:0139
-a 100                  ; Käännetään muistiin
2515:0100 MOV  AX,FFFF  ; AX kaikki bitit päälle
2515:0103 PUSH AX       ; AX pinoon
2515:0104 POPF          ; eli liput:=AX
2515:0105               ; kääntäminen lopetetaan ENTER tyhjällä rivillä
-r                      ; tulostetaan rekistereiden arvot
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2515  ES=2515  SS=2515  CS=2515  IP=0100   NV UP EI PL NZ NA PO NC
2515:0100 B8FFFF        MOV     AX,FFFF
-                               ; liput OF DF IF SF ZF AF PF CF
-t                      ; Suoritetaan MOV AX,FFFFF

AX=FFFF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2515  ES=2515  SS=2515  CS=2515  IP=0103   NV UP EI PL NZ NA PO NC
2515:0103 50            PUSH    AX
-t

AX=FFFF  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=2515  ES=2515  SS=2515  CS=2515  IP=0104   NV UP EI PL NZ NA PO NC
```

```
2515:0104 9D            POPF
-t                       ; Nyt kaikki liput tulevat päälle

AX=FFFF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2515  ES=2515  SS=2515  CS=2515  IP=0105   OV DN EI NG ZR AC PE CY
2515:0105 00FF          ADD     BH,BH
-f 100,120 'Kukkuu'      ; Sijoitetaan muistiin jonoa Kukkuu
-n koe.dat               ; Annetaan käsitteltäväksi nimeksi KOE.DAT
-rcx                     ; BX CX määrää W-käskyssä kirj. tavujen lkm
CX 0000
:6                       ; CX:=6
-r
AX=FFFF  BX=0000  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2515  ES=2515  SS=2515  CS=2515  IP=0105   OV DN EI NG ZR AC PE CY
2515:0105 754B          JNZ     0152
-w                       ; Kirjoitetaan 6 tavua nimelle KOE.DAT koska CX=6
Writing 0006 bytes
-q                       ; Lopetetaan DEBUG-ohjelma
E:\TKJ\MONISTE\ASM>type koe.dat
Kukkuu
E:\TKJ\MONISTE\ASM>
```

Debug-ohjelmassa ei voida asettaa keskeytyskohtia pysyvästi, mutta GO-käsky asettaa väliaikaisia keskeytyskohtia.

GO-käskyn käytöstä esimerkki SYMDEB-ohjelman yhteydessä.

6.3 Symdeb

Microsoftin Symdeb on DEBUG-ohjelman kanssa yhteensopiva symbolinen virheenjäljitin. Siis kaikki DEBUG-ohjelman komennot toimivat sellaisenaan. Lisänä on joukko uusia komentoja, joista tässä käsitellään vain muutamia.

6.3.1 Kääntäminen

Symbolisen informaation saamiseksi pitää käännös suorittaa seuraavasti (käännetään ohjelma PT.ASM):

```
MASM PT;
LINK PT /MAP;
MAPSYM PT
```

Lisäksi .ASM tiedostossa jokainen nimiö pitää ilmoittaa PUBLIC-komennolla, mikäli sen halutaan näkyvän debuggerissa:

```
PUBLIC vektori,alkioita
```

6.3.2 Uudet komennot

Ilman symbolistakin informaatiota Symdeb on Debug-ohjelmaa käyttökelpoisempi uusien käskyjensä ansiosta:

```
BP [numero] osoite [n] ["komennot"]
                        Breakpoint - asettaa keskeytyskohdan
BC numero|*             Clear - poistaa keskeytyskohdan
BD numero|*             Disable - poistetaan väliaikaisesti keskeytyskohta
BE numero|*             Enable - otetaan keskeytyskohta jälleen käyttöön
BL                      List - tulostaa keskeytyskohdat
?lauseke                laskee lausekkeen arvon
DA [alue]               Dump ASCII- tulosta muistia ASCIIna
DB [alue]               Dump Bytes - tulosta muistia tavuina
DW [alue]               Dump Words - tulosta muistia sanoina
DD [alue]               Dump Doublewords - tulosta muistia osoitteina
DS [alue]               Dump Short Reals - tulosta muistia 32 bitin reaalilukuina
DL [alue]               Dump Long Reals - tulosta muistia 64 bitin reaalilukuina
DT [alue]               Dump Ten-Byte Reals - tulosta muistia 80 bitin reaalilukuina
EA,EB,EW,ED,ES,EL,ET    kuten E-komento, mutta eri kokoisille alkioille
```

```
P[=osoite][määrä]      PTrace - kuten Trace, mutta suorittaa aliohjelmat yhtenä käs-
                       kynä.  Samoin REP ja LOOP-käskyt.
R[rekisteri[[=]arvo]]  muuttaa rekisterin arvon
```

6.3.3 Lukujärjestelmät

Lisäksi lukuja voidaan käyttää eri kantajärjestelmissä:

```
 Kirjain    järjestelmä     esimerkit
 Y          Binary          10010Y
 O,Q        Octal           74O    24Q
 T          Decimal         10T
 H          Hexadecimal     123FH
```

Oletuksena luvut ovat 16-järjestelmässä, ellei toisin mainita.

6.3.4 Esimerkkiajo

Seuraavassa esimerkissä tutkitaan aiemmin esitettyä POIKKEAMA-aliohjelman testausohjelmaa PT.ASM. Puolipiste (;) on Symdeb ohjelmassa erotinmerkki, jonka avulla voidaan antaa useita komentoja samalle riville. Esimerkissä kuitenkin puolipiste ja sen jälkeinen osa on tulkittava kommentiksi, jota ei kirjoiteta:

```
E:\TKJ\MONISTE\ASM>masm pt;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Käännetään NEAR PROG tyyppiseksi

  51312 + 286544 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

E:\TKJ\MONISTE\ASM>link pt;

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.


E:\TKJ\MONISTE\ASM>symdeb pt.exe
Microsoft (R) Symbolic Debug Utility  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.

Processor is [80286]
-u        ; Käännetään koodia assembleriksi
2A64:00A8 B8712A          MOV    AX,2A71
2A64:00AB 8ED8            MOV    DS,AX
2A64:00AD 8EC0            MOV    ES,AX
2A64:00AF 8D3E0600        LEA    DI,[0006]
2A64:00B3 8B0E1A00        MOV    CX,[001A]
2A64:00B7 E8D5FF          CALL   008F
2A64:00BA 8D360600        LEA    SI,[0006]
2A64:00BE E84FFF          CALL   0010
-u        ; Lisää, koska pääohjelma ei vielä mahtunut kokonaan
2A64:00C1 8D3E3200        LEA    DI,[0032]
2A64:00C5 E884FF          CALL   004C
2A64:00C8 BA1C00          MOV    DX,001C
2A64:00CB B409            MOV    AH,09
2A64:00CD CD21            INT    21
2A64:00CF B44C            MOV    AH,4C                          ;'L'
2A64:00D1 B000            MOV    AL,00
2A64:00D3 CD21            INT    21
-d 2a71:0,3f ; Tulostetaan Data-segmentin sisältöä
2A71:0000  4C B0 00 CD 21 00 00 00-00 00 00 00 00 00 00 00  L0.M!...........
2A71:0010  00 00 00 00 00 00 00 00-00 00 0A 00 0D 0A 53 75  ..............Su
2A71:0020  75 72 69 6E 20 70 6F 69-6B 6B 65 61 6D 61 20 6F  urin poikkeama o
2A71:0030  6E 20 20 20 20 20 20 20-2B 2F 2D 31 2E 0D 0A 24  n       +/-1...$
-g=a8 ba     ; Koska syöttö testattu aiemmin, voidaan ajaa sen ohi
12 5 3       ; Syöttö.  Oikeasti seuraava rivi tulostuisi päälle!
AX=2A71  BX=0000  CX=0003  DX=0000  SP=0400  BP=0000  SI=0000  DI=0006
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=00BA   NV UP EI PL ZR NA PE NC
2A64:00BA 8D360600        LEA    SI,[0006]                      DS:0006=000C
-t           ; Suoritetaan seuraava käsky
AX=2A71  BX=0000  CX=0003  DX=0000  SP=0400  BP=0000  SI=0006  DI=0006
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=00BE   NV UP EI PL ZR NA PE NC
```

```
2A64:00BE E84FFF         CALL   0010
-t           ; Siirrytään aliohjelmaan
AX=2A71  BX=0000  CX=0003  DX=0000  SP=03FE  BP=0000  SI=0006  DI=0006
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0010   NV UP EI PL ZR NA PE NC
2A64:0010 53             PUSH   BX
-u           ; Käännetään assembler-kieliseksi
2A64:0011 52             PUSH   DX
2A64:0012 57             PUSH   DI
2A64:0013 56             PUSH   SI
2A64:0014 55             PUSH   BP
2A64:0015 33C0           XOR    AX,AX
2A64:0017 8B2C           MOV    BP,[SI]
2A64:0019 8BFD           MOV    DI,BP
2A64:001B 8BD0           MOV    DX,AX
-u           ; Lisää, koska kaikki ei mahtunut vielä
2A64:001D 8BD8           MOV    BX,AX
2A64:001F E325           JCXZ   0046
2A64:0021 51             PUSH   CX
2A64:0022 FC             CLD
2A64:0023 AD             LODSW
2A64:0024 03D8           ADD    BX,AX
2A64:0026 83D200         ADC    DX,+00
2A64:0029 3BE8           CMP    BP,AX
-u           ; Lisää, koska kaikki ei mahtunut vielä
2A64:002B 7D02           JGE    002F
2A64:002D 8BE8           MOV    BP,AX
2A64:002F 3BF8           CMP    DI,AX
2A64:0031 7E02           JLE    0035
2A64:0033 8BF8           MOV    DI,AX
2A64:0035 E2EC           LOOP   0023
2A64:0037 59             POP    CX
2A64:0038 8BC3           MOV    AX,BX
-u           ; Lisää, koska kaikki ei mahtunut vielä
2A64:003A F7F9           IDIV   CX
2A64:003C 2BE8           SUB    BP,AX
2A64:003E 2BC7           SUB    AX,DI
2A64:0040 3BC5           CMP    AX,BP
2A64:0042 7D02           JGE    0046
2A64:0044 8BC5           MOV    AX,BP
2A64:0046 5D             POP    BP
2A64:0047 5E             POP    SI
-u           ; Lisää, koska kaikki ei mahtunut vielä
2A64:0048 5F             POP    DI
2A64:0049 5A             POP    DX
2A64:004A 5B             POP    BX
2A64:004B C3             RET
2A64:004C 50             PUSH   AX
2A64:004D 53             PUSH   BX
2A64:004E 51             PUSH   CX
2A64:004F 52             PUSH   DX
-bp 24       ; Asetetaan kaksi katkokohtaa
-bp 37
-bl          ; Tulostetaan pysäytyskohdat
0 e 2A64:0024
1 e 2A64:0037
-g 1f        ; Ajetaan ohjelmaa CX:än vertailuun saakka
AX=0000  BX=0000  CX=0003  DX=0000  SP=03F4  BP=000C  SI=0006  DI=000C
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=001F   NV UP EI PL ZR NA PE NC
2A64:001F E325           JCXZ   0046
-g           ; CX<>0 kuten haluttiin, voidaan ajaa katkokohtaan
AX=000C  BX=0000  CX=0003  DX=0000  SP=03F2  BP=000C  SI=0008  DI=000C
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0024   NV UP EI PL ZR NA PE NC
2A64:0024 03D8           ADD    BX,AX                    ;BR0
-g           ; AX:ssä 12 kuten piti, ajetaan eteenpäin
AX=0005  BX=000C  CX=0002  DX=0000  SP=03F2  BP=000C  SI=000A  DI=000C
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0024   NV UP EI PL ZR NA PE NC
2A64:0024 03D8           ADD    BX,AX                    ;BR0
-g           ; AX:ssä 5, pienin ja suurin OK, ajetaan eteenpäin
AX=0003  BX=0011  CX=0001  DX=0000  SP=03F2  BP=000C  SI=000C  DI=0005
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0024   NV UP EI PL NZ NA PO NC
2A64:0024 03D8           ADD    BX,AX                    ;BR0
-g           ; AX=3 OK, suurin 12 ja pienin 5 OK!
AX=0003  BX=0014  CX=0000  DX=0000  SP=03F2  BP=000C  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0037   NV UP EI PL NZ NA PO NC
2A64:0037 59             POP    CX                       ;BR1
-t8          ; Luvut käsitelty, pienin ja suurin OK.  8 askelta eteenpäin
AX=0003  BX=0014  CX=0003  DX=0000  SP=03F4  BP=000C  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0038   NV UP EI PL NZ NA PO NC
2A64:0038 8BC3           MOV    AX,BX
AX=0014  BX=0014  CX=0003  DX=0000  SP=03F4  BP=000C  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=003A   NV UP EI PL NZ NA PO NC
2A64:003A F7F9           IDIV   CX
AX=0006  BX=0014  CX=0003  DX=0002  SP=03F4  BP=000C  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=003C   NV UP EI NG NZ AC PE CY
2A64:003C 2BE8           SUB    BP,AX
AX=0006  BX=0014  CX=0003  DX=0002  SP=03F4  BP=0006  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=003E   NV UP EI PL NZ NA PE NC
```

```
2A64:003E 2BC7          SUB    AX,DI
AX=0003  BX=0014  CX=0003  DX=0002  SP=03F4  BP=0006  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0040   NV UP EI PL NZ NA PE NC
2A64:0040 3BC5          CMP    AX,BP
AX=0003  BX=0014  CX=0003  DX=0002  SP=03F4  BP=0006  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0042   NV UP EI NG NZ AC PO CY
2A64:0042 7D02          JGE    0046
AX=0003  BX=0014  CX=0003  DX=0002  SP=03F4  BP=0006  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0044   NV UP EI NG NZ AC PO CY
2A64:0044 8BC5          MOV    AX,BP
AX=0006  BX=0014  CX=0003  DX=0002  SP=03F4  BP=0006  SI=000C  DI=0003
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=0046   NV UP EI NG NZ AC PO CY
2A64:0046 5D            POP    BP
-g 4b        ; Tähän saakka kunnossa, ajetaan paluuseen saakka
AX=0006  BX=0000  CX=0003  DX=0000  SP=03FE  BP=0000  SI=0006  DI=0006
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=004B   NV UP EI NG NZ AC PO CY
2A64:004B C3            RET
-t           ; Suoritetaan paluu aliohjelmasta
AX=0006  BX=0000  CX=0003  DX=0000  SP=0400  BP=0000  SI=0006  DI=0006
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=00C1   NV UP EI NG NZ AC PO CY
2A64:00C1 8D3E3200      LEA    DI,[0032]                      DS:0032=2020
-t
AX=0006  BX=0000  CX=0003  DX=0000  SP=0400  BP=0000  SI=0006  DI=0032
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=00C5   NV UP EI NG NZ AC PO CY
2A64:00C5 E884FF        CALL   004C
-p           ; Suoritetaan aliohjelma menemättä sinne (testattu aik.)
AX=0006  BX=0000  CX=0003  DX=0000  SP=0400  BP=0000  SI=0006  DI=0032
DS=2A71  ES=2A71  SS=2A75  CS=2A64  IP=00C8   NV UP EI PL NZ NA PE NC
2A64:00C8 BA1C00        MOV    DX,001C
-d 0,3f      ; Katsotaan muuttunut muistin sisältö.
2A71:0000  4C B0 00 CD 21 00 0C 00-05 00 03 00 00 00 00 00  L0.M!...........
2A71:0010  00 00 00 00 00 00 00 00-00 00 0A 00 0D 0A 53 75  ..............Su
2A71:0020  75 72 69 6E 20 70 6F 69-6B 6B 65 61 6D 61 20 6F  urin poikkeama o
2A71:0030  6E 20 30 30 30 30 36 20-2B 2F 2D 31 2E 0D 0A 24  n 00006 +/-1...$
-g           ; Suoritetaan ohjelma loppuun.

Suurin poikkeama on 00006 +/-1.

Program terminated normally (0)
-q           ; poistutaan tyytyväisenä Symdeb-ohjelmasta

E:\TKJ\MONISTE\ASM>
```

6.3.5 Historia talteen

Vaikka kuvaruutupohajiset debuggerit kuten CodeView ja Turbo Debugger ovat mukavia käyttää, ei niillä saada ajosta samanlaista historiaa kuin Symdeb- tai Debug-ohjelmilla. Tämän takia jomman kumman rivipohjaisen debuggerin käyttö on toisinaan suositeltavaa.

6.3.6 INT 3, 0CCH

Debuggerit tekevät keskeytyskohdat seuraavalla tavalla:

- kun ohjelma käynnistetään, otetaan sisäisestä taulukosta kaikkien keskeytyskohtien osoitteet
- kussakin osoitteessa oleva tavu talletetaan muualle
- kuhunkin osoitteeseen laitetaan tavu 0CCH (INT 3)
- ohjelmalaskuri asetetaan sille kuuluvaan arvoonsa ja näin suoritetaan käyttäjän ohjelmaa
- jos joskus ohjelmalaskuri osoittaa tavuun, jonka sisältö on 0CCH, suoritetaan keskeytys 3
- debugger korvaa jokaisen muuttamansa 0CCH tavun alkuperäisellä arvolla
- käyttäjälle näytetään rekisterit kuten ne olivat ennen tavua 0CCH

Edellä mainitusta seuraa muutamia sivuvaikutuksia. Ensinnäkin ohjelma ei ole debuggerin alla ajettuna ajon aikana täsmälleen sama kuin se on normaalisti.

Erityisesti väärään osoitteeseen asetettu katkokohta saattaa aiheuttaa ohjelman väärän toiminnan, ei suinkaan keskeytystä. Esimerkiksi muistipaikassa 100H oleva käsky `MOV AX,1234H` (B8 34 12) muuttuu käskyksi `MOV AX,12CCH` (B8 CC 12), mikäli keskeytyskohta asetetaan vahingossa osoitteeseen 101H (`G 101` tai `BP 101`)!

6.3.7 Turbo Pascal 3.0

Toinen seikka jota voidaan käyttää hyväksi, on se että ohjelman suoritus katkaistaan aina 0CCH lauseeseen ja siirrytään debuggerin alaisuuteen. Tällä tavalla voidaan käsitellä esimerkiksi Turbo Pascal 3.0 ohjelmia. Ohjelmakoodiin kirjoitetaan lause `INLINE($CC)` epäilyttäviin kohtiin.

Käynnistetään aluksi Turbo Pascal 3.0 Symdeb-debuggerin alaisuuteen:

```
E:\TKJ\MONISTE\ASM>symdeb c:\sys\bin\turbo.com
Microsoft (R) Symbolic Debug Utility  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.

Processor is [80286]
-g

; Turbo Pascal käynnistyy. Kirjoitetaan vaikkapa seuraava ohjelma:

VAR i,j:INTEGER;
BEGIN
  i:=2;
  INLINE($CC);
  j:=i+5;
END.

; ja ajetaan se Turbo Pascalin Run-komennolla:

>R

Compiling
  6 lines

Code:       0004 paragraphs (    64 bytes), 0D24 paragraphs free
Data:       0003 paragraphs (    48 bytes), 0FD9 paragraphs free
Stack/Heap: 6808 paragraphs (426112 bytes)

Running
AX=0002  BX=2D9C  CX=2DB7  DX=0000  SP=FFFE  BP=FFFE  SI=2D9C  DI=2D9C
DS=37D1  ES=34F5  SS=9000  CS=34F5  IP=2DA5   NV UP EI PL NZ NA PE NC
34F5:2DA5 CC            INT    3
-g 2DA6          ; ohitetaan INT 3 -lause. Myös  RIP=IP+1  kävisi
AX=0002  BX=2D9C  CX=2DB7  DX=0000  SP=FFFE  BP=FFFE  SI=2D9C  DI=2D9C
DS=37D1  ES=34F5  SS=9000  CS=34F5  IP=2DA6   NV UP EI PL NZ NA PE NC
34F5:2DA6 A16002        MOV    AX,[0260]
-u               ; Nyt voitaisiin käyttää debuggeria normaalisti
34F5:2DA9 050500        ADD    AX,0005
34F5:2DAC A36202        MOV    [0262],AX
34F5:2DAF E90000        JMP    2DB2
34F5:2DB2 33C0          XOR    AX,AX
34F5:2DB4 E8D2DE        CALL   0C89
34F5:2DB7 0000          ADD    [BX+SI],AL
34F5:2DB9 897EFE        MOV    [BP-02],DI
-               ; Ohjelman ajaminen palauttaisi Turbo Pascaliin!
```

6.4 CodeView

Microsoftin CodeView on Symdebistä kehitetty symbolinen kuvaruutupohjainen debuggeri. Sillä voidaan käsitellä Microsoftin kääntäjillä käännettyjä C-, Pascal-(?), Basic- ja assembler-ohjelmia. Ilman symbolista informaatiota voidaan käsitellä mitä tahansa koodia.

Käynnistetään komennolla `CV tiedoston_nimi`

Komentoja voidaan antaa esimerkiksi C-kielen syntaksin mukaisesti.

CodeView:n parhaita puolia on hiirellä asetettavat katkokohdat ja yleensäkin hiiren käyttö.

Muuten CodeView on hyvin samanlainen seuraavana käsiteltävän Turbo Debuggerin kanssa.

6.5 Turbo Debugger

Borlandin Turbo Debugger on symbolinen debuggeri erityisesti Borlandin kielillä tehtyjen ohjelmien käsittelemiseksi. Kääntäjissä ja linkkerissä on valmiina optiot symbolisen informaation tuottamiseksi ja erillistä MAPSYM- tai vastaava ohjelmaa ei tarvita. Turbo Debugger sisältää hyvät avustukset ja on helppo käyttää. Lisäksi siinä on lähes kaikki hyvältä symboliselta debuggerilta vaadittavat ominaisuudet.

Turbo Debuggerin ainoat huonot puolet ovat sen suuri koko, siinä ei ole hiiriliityntää sekä ajosta ei jää mitään historiaa.

Debuggerin kokoa voidaan toisin pienentää mukana seuraavan TDREMOTE-ohjelman avulla (alle 20 kilotavua). Tämä ohjelma käynnistetään toisessa koneessa ja TD-toisessa. Koneet yhdistetään toisiinsa sarjaporttien välityksellä. Itse debuggaus suoritetaan toisessa koneessa ja ohjelma pyörii TDREMOTEn päällä toisessa. Näin voidaan käsitellä hyvinkin suuria ohjelmia.

6.5.1 Kääntäminen

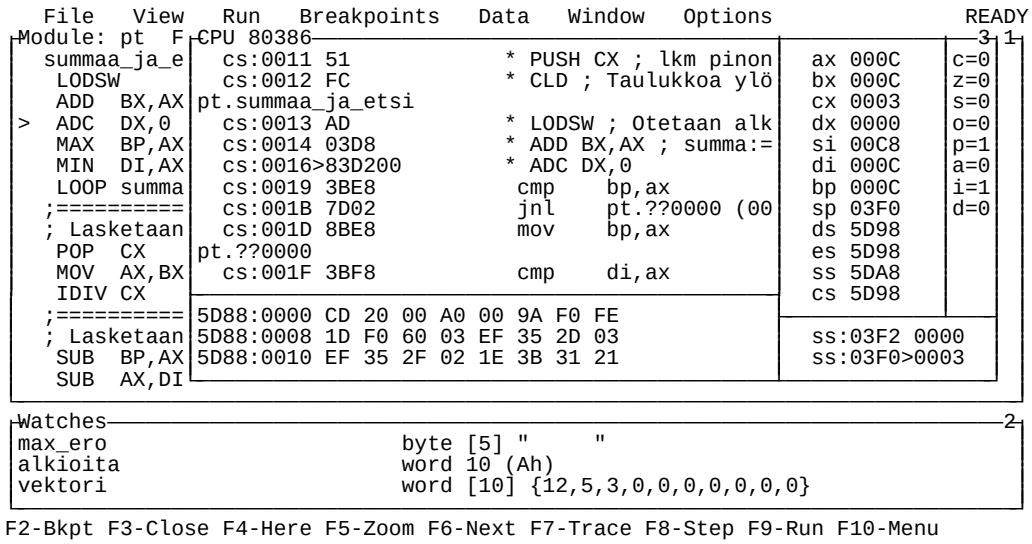
Borlandin kielien integroiduissa editoreissa valitaan kääntäjän optio erillisen debuggauksen tukemiseksi ja käännös levylle (TURBO PASCAL 5.0:ssa [ALT-D] [S] [ALT-C] [D]). Ulkoisilla kääntäjillä käännös saadaan seuraavasti:

Turbo Pascal 5.0:	`TPC oma /v`
Turbo C 2.0:	`TCC -v oma`
Turbo Assembler 1.0:	`TASM oma /zi/z;`
	`TLINK oma /v;`

Kun käännös on valmis, käynnistetään Turbo Debugger komennolla `TD tiedoston_nimi`.

6.5.2 Esimerkki

Seuraavassa esimerkki ohjelman PT.EXE ajosta, kun ohjelmaa on ajettu osoitteeseen 16H saakka sekä Watch-ikkunaan on laitettu tutkittavaksi muuttujien `vektori`, `alkiota` ja `max_ero` arvot (esim. näppäilyllä `[CTRL-F7]vektori[RETURN]` jne.).



Näyttöön on lähdekielisen koodin lisäksi otettu näkymä CPU-ikkunasta ([ALT-V][C]).

6.5.3 Komentoja

Kunkin ikkunan oikeassa yläkulmassa on merkittynä ikkunan numero. Kursori voidaan siirtää ikkunasta toiseen [ALT-numero] -näppäilyllä (numero kirjainnäppäimistöltä, ei erilliseltä numeronäppäimistöltä). Mikäli ikkunassa on useampia ruutuja, voidaan [TAB] ja [SHIFT-TAB] näppäimillä siirtyä ruudusta toiseen. Ikkunoiden kokoa ja paikkaa voidaan muuttaa painamalla [Scroll Lock]-näppäintä. Kun ikkuna on halutussa paikassa painetaan uudestaan [Scroll Lock]-näppäintä.

Kussakin paikassa saa [F1]-näppäimellä toimintoon liittyvää apua. [F10] näppäimellä pääsee valitsemaan näytön yläreunassa olevasta menusta toimintoja. [ALT-X] lopettaa ohjelman toiminnan.

Ohjelman ajaminen haluttuun paikkaan on helppoa: Viedään kursori halutun käskyn kohdalle ja painetaan Here-näppäintä ([F4]). Myös katkokohdat asetetaan vastaavasti. Kursori halutulle riville ja [F2]. Kaikki toiminnot voidaan tehdä tietysti myös osoitteiden avulla.

Ohjelmaa voidaan aina käsitellä joko lähdekielisessä ikkunassa tai CPU-ikkunassa.

Siirtymällä Watch-ikkunaan, voidaan muuttaa muuttujien arvoja: Siirretään kursori muuttujan päälle ja painetaan [CTRL-I] (Inspect). Uusi arvo kirjoitetaan vanhan tilalle.

Arvoja voidaan muuttaa myös [CTRL-F4] ja muuttujan nimi sekä siirtymällä [TAB]-näppäimellä alimpaan ruutuun, johon arvo kirjoitetaan. Tosin vektoreiden arvoja ei aina voida muuttaa tällä valinnalla.

6.5.4 Lukujärjestelmä

Lukujärjestelmä riippuu käytettävästä kielestä, joka voidaan vaihtaa milloin tahansa. Seuraavassa esimerkki luvun 10H syöttämisestä eri kielissä:

```
C:          0x10
Pascal:     $10
Assembler   10H
```

Lukuja syötettäessä on oltava tarkkana, koska assembler-moodissa syötetty 10 tarkoittaa 10H ja Pascal-moodissa syötetty 10 tarkoitaa 10_{10}. Kieli on automaatisesti pääohjelman kieli, ellei erikseen toisin mainita.

Luku 7

Assembler-kieliset aliohjelmat

Suurin hyöty konekielestä saadaan, kun sillä tehdyt ohjelmat yhdistetään oikeassa suhteessa korkeamman tason kielillä tehtyihin ohjelmiin. Ainoastaan ohjelmat, joilta vaaditaan erittäin pientä kokoa kannattaa kirjoittaa kokonaan assemblerilla.

Assembler-kielisen ohjelman liittäminen toiseen kieleen vaatii käytettävän kielen parametrinvälityksen ymmärtämistä. Eri kielissä parametrin välittäminen on toteutettu eri tavoin. Sama ohjelma ei ehkä toimi sekä Pascalin että C:n aliohjelmana.

7.1 TURBO PASCAL 5.0

7.1.1 Aliohjelmat

Tutkitaan seuraavaa Turbo Pascal-ohjelmaa Turbo Debuggerin avulla:

```
VAR k,i,j:INTEGER;
PROCEDURE summa(i,j:INTEGER; VAR k:INTEGER);
BEGIN
  k:=i+j;
END;
BEGIN
  i:=4;
  j:=3;
  summa(i,j,k);
  i:=k;
END.
```

Käännöksessä on käytetty pinon tarkistuksen estävää optioita:

```
PROGRAM.SUMMA: BEGIN
  cs:0000 55            push   bp
  cs:0001 89E5          mov    bp,sp
PROGRAM.4:  k:=i+j;
  cs:0003 8B460A        mov    ax,[bp+0A]
  cs:0006 034608        add    ax,[bp+08]
  cs:0009 C47E04        les    di,[bp+04]
  cs:000C 268905        mov    es:[di],ax
PROGRAM.5: END;
  cs:000F 89EC          mov    sp,bp
  cs:0011 5D            pop    bp
  cs:0012 C20800        ret    0008
PROGRAM.7: BEGIN
  cs:0015>9A00009D5D    call   5D9D:0000
  cs:001A 55            push   bp
  cs:001B 89E5          mov    bp,sp
PROGRAM.8:  i:=4;
  cs:001D C7063E000400  mov    word ptr [003E],0004
PROGRAM.9:  j:=3;
  cs:0023 C70640000300  mov    word ptr [0040],0003
PROGRAM.10:  summa(i,j,k);
  cs:0029 FF363E00      push   [003E]
  cs:002D FF364000      push   [0040]
  cs:0031 BF3C00        mov    di,003C
  cs:0034 1E            push   ds
  cs:0035 57            push   di
  cs:0036 E8C7FF        call   PROGRAM.SUMMA
PROGRAM.11:  i:=k;
  cs:0039 A13C00        mov    ax,[003C]
  cs:003C A33E00        mov    [003E],ax
PROGRAM.12: END.
  cs:003F 89EC          mov    sp,bp
  cs:0041 5D            pop    bp
  cs:0042 31C0          xor    ax,ax
  cs:0044 9AD8009D5D    call   5D9D:00D8
```

Aliohjelmakutsu `summa(i,j,k);` kääntyy siis seuraavasti:

```
cs:0029 FF363E00       push   [003E]         ; i:n arvo pinoon
cs:002D FF364000       push   [0040]         ; j:n arvo pinoon
cs:0031 BF3C00         mov    di,003C        ; DI:= k:n offset
cs:0034 1E             push   ds             ; k:n segmentti pinoon
cs:0035 57             push   di             ; k:n offset pinoon
cs:0036 E8C7FF         call   PROGRAM.SUMMA  ; kutsutaan aliohjelmaa
```

Pinoon laitetaan ensin muuttujan i arvo, sitten muuttujan j arvo ja lopuksi k:n osoite (double word). Pino aliohjelmaan tultaessa on seuraavan näköinen (esitetään sanoina):

```
          ┌──────┐
SP    ->  │ 0039 │   paluuosoite
          ├──────┤
          │ 003C │   k:n offset osoite
          ├──────┤
          │  DS  │   k:n segment osoite
          ├──────┤
          │ 0003 │   j:n arvo
          ├──────┤
          │ 0004 │   i:n arvo
          └──────┘
```

Aliohjelman `BEGIN`-lauseesta kääntyi seuraava koodi:

```
cs:0000 55             push   bp         ; vanha BP pinoon
cs:0001 89E5           mov    bp,sp      ; BP:=SP
```

`BEGIN`-lauseen jälkeen on pino seuraavan näköinen:

```
          ┌──────┐
SP = BP ->│ a.BP │   alkuperäinen BP
          ├──────┤
    BP+02 │ 0039 │   paluuosoite
          ├──────┤
    BP+04 │ 003C │   k:n offset osoite
          ├──────┤
    BP+06 │  DS  │   k:n segment osoite
          ├──────┤
    BP+08 │ 0003 │   j:n arvo
          ├──────┤
    BP+0A │ 0004 │   i:n arvo
          └──────┘
```

Rekisteri BP siis osoittaa pinossa siihen kohtaan, mihin ennen aliohjelmakutsua ollut BP:n arvo on talletettu. Näin BP:n avulla voidaan nyt osoittaa pinossa oleviin parametreihin. Esimerkiksi i:n arvo saadaan rekisteriin AX käskyllä `MOV AX,[BP+0A]`.

Koska tulos sijoitetaan muuttujaan k, on siitä välitetty osoite pinossa. Osoite saadaan rekisteripariin ES:DI käskyllä `LES DI,[BP+04]`. Näin ollen tulos voidaan sijoittaa muuttujaan k lauseella `MOV ES:[DI],AX`.

Aliohjelmasta poistuttaessa `END`-lauseesta kääntyy koodi:

```
cs:000F 89EC           mov    sp,bp
cs:0011 5D             pop    bp
cs:0012 C20800         ret    0008
```

Tässä tapauksessa sijoitus `SP:=BP` on turha, koska niillä on muutenkin sama arvo. Näin ei kuitenkaan ole aina, esimerkiksi mikäli jouduttaisiin allokoimaan lokaaleja muuttujia pinosta.

Käskyllä `POP BP` palautetaan BP:n kutsua edeltänyt arvo ja pino on siis samassa kunnossa kuin aliohjelmaan tultaessa. `RET 0008` -käsky ottaa pinosta ohjelmalaskurille paluuosoitteen. Tämän jälkeen pino-osoittimeen lisätään 8 ja pino on täsmälleen samassa kunnossa kuin ennen aliohjelman kutsua. Siis pinoon lisäämisellä poistetaan kutsuvan ohjelman pinoon laittamat parametrit. Huomattakoon, että Turbo Pascalissa aliohjelma

huolehtii pinon siivoaminen pinon siivoamisesta, kun taas C-kielessä kutsuva ohjelma siivoaa pinon. Kummallakin tavalla on omat etunsa.

7.1.2 Funktiot

Muutetaan edellinen aliohjelma vastaavaksi funktio-aliohjelmaksi:

```
VAR k,i,j:INTEGER;
FUNCTION summa(i,j:INTEGER):INTEGER;
BEGIN
  summa:=i+j;
END;
BEGIN
  i:=4;
  j:=3;
  i:=summa(i,j);
END.
```

Tämä ohjelma kääntyy konekielelle seuraavasti:

```
PROGRAM.SUMMA: BEGIN
  cs:0000 55             push   bp
  cs:0001 89E5           mov    bp,sp
  cs:0003 83EC02         sub    sp,0002
PROGRAM.4:  summa:=i+j;
  cs:0006 8B4606         mov    ax,[bp+06]
  cs:0009 034604         add    ax,[bp+04]
  cs:000C 8946FE         mov    [bp-02],ax
PROGRAM.5: END;
  cs:000F 8B46FE         mov    ax,[bp-02]
  cs:0012 89EC           mov    sp,bp
  cs:0014 5D             pop    bp
  cs:0015 C20400         ret    0004
PROGRAM.7: BEGIN
  cs:0018>9A00009D5D     call   5D9D:0000
  cs:001D 55             push   bp
  cs:001E 89E5           mov    bp,sp
PROGRAM.8:  i:=4;
  cs:0020 C7063E000400   mov    word ptr [003E],0004
PROGRAM.9:  j:=3;
  cs:0026 C706400000300  mov    word ptr [0040],0003
PROGRAM.10:  i:=summa(i,j);
  cs:002C FF363E00       push   [003E]
  cs:0030 FF364000       push   [0040]
  cs:0034 E8C9FF         call   PROGRAM.SUMMA
  cs:0037 A33E00         mov    [003E],ax
PROGRAM.11: END.
  cs:003A 89EC           mov    sp,bp
  cs:003C 5D             pop    bp
  cs:003D 31C0           xor    ax,ax
  cs:003F 9AD8009D5D     call   5D9D:00D8
```

Voidaan todeta, että vastaavaan aliohjelmarakenteeseen verrattuna ainoana erona on se, että nyt ei muuttujaparametriä k ole. Siis sitä vastaavaa osoitetta ei laiteta pinoon. Aliohjelma laskee tuloksen rekisteriin AX ja palaa kuten edellisessäkin esimerkissä. Pääohjelma käyttää rekisteriä AX funktion tuloksena.

Funktion laskemisessa käytetään apuna lokaalia muuttujaa pinosta (joka yo. esimerkissä on turha).

7.1.3 Lokaalit muuttujat

Muutetaan edelleen SUMMA-aliohjelmaa siten, että se käyttää lokaaleja muuttujia:

```
PROCEDURE summa(i,j:INTEGER; VAR k:INTEGER);
VAR a,b:INTEGER;
BEGIN
  a:=1;
  b:=2;
  k:=i+j;
END;
```

Tämä näyttää konekielisenä seuraavalta:

```
PROGRAM.SUMMA: BEGIN
  cs:0000 55             push   bp
  cs:0001 89E5           mov    bp,sp
  cs:0003 83EC04         sub    sp,0004
PROGRAM.5:  a:=1;
  cs:0006 C746FE0100     mov    word ptr [bp-02],0001
PROGRAM.6:  b:=2;
  cs:000B C746FC0200     mov    word ptr [bp-04],0002
PROGRAM.7:  k:=i+j;
  cs:0010 8B460A         mov    ax,[bp+0A]
  cs:0013 034608         add    ax,[bp+08]
  cs:0016 C47E04         les    di,[bp+04]
  cs:0019 268905         mov    es:[di],ax
PROGRAM.8: END;
  cs:001C 89EC           mov    sp,bp
  cs:001E 5D             pop    bp
  cs:001F C20800         ret    0008
```

Pääohjelma näyttäisi aivan samalta kuin ennenkin, ainoana erona on, että se on siirtynyt 13 tavua eteenpäin. Sijoituksen `b:=2` jälkeen on pino seuraavan näköinen:

SP ->BP-04	0002	lokaali b
BP-02	0001	lokaali a
BP ->	a.BP	alkuperäinen BP
BP+02	0046	paluuosoite
BP+04	0031	k:n offset osoite
BP+06	DS	k:n segment osoite
BP+08	0003	j:n arvo
BP+0A	0004	i:n arvo

Nyt siis lauseella `MOV SP,BP` on selvä merkitys: lokaalit muuttujat poistetaan pinosta. Lokaalit muuttujat tehtiin pinoon siirtämällä pino-osoitinta tarvittavan tavumäärän verran ylöspäin (`SUB SP,4`).

7.1.4 Lokaalit aliohjelmat

Pascalissa on sallittua, että aliohjelman sisällä on toinen aliohjelma. Kukin aliohjelma voi käyttää kaikkia niitä muuttujia, jotka on esitelty sitä ympäröivissä lohkoissa tai lohkossa itsessään. Katsotaan vielä esimerkki tästä:

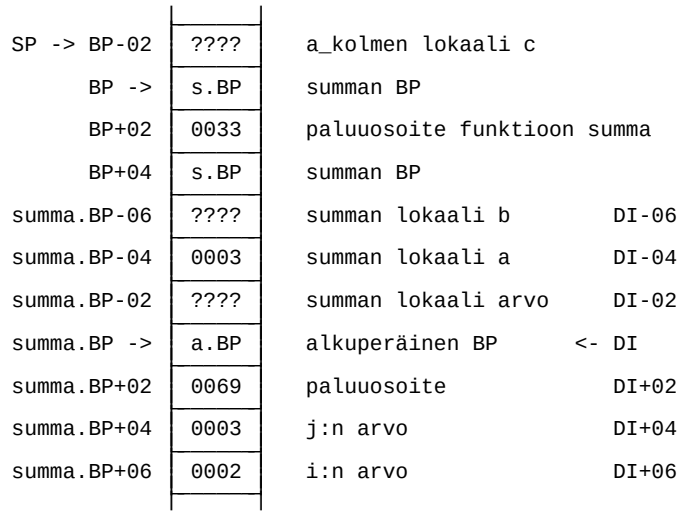
```
VAR k,i,j:INTEGER;
FUNCTION summa(i,j:INTEGER):INTEGER;
VAR a,b:INTEGER;
 PROCEDURE a_kolme;
 VAR c:INTEGER;
 BEGIN   { a_kolme                                    }
   k:=1; { sijoitus globaaliin muuttujaan k           }
   i:=2; { sijoitus summan tilapäiseen parametriin i }
   a:=3; { sijoitus summan lokaaliin muuttujaan a     }
   c:=4; { sijoitus a_kolmen lokaaliin muuttujaan c   }
 END;    { a_kolme                                    }
BEGIN    { summa                                      }
  a_kolme;
  b:=2;
  summa:=i+j;
END;     { summa                                      }
BEGIN    { pääohjelma                                 }
  i:=4;
  j:=3;
  k:=summa(i,j);
END.     { pääohjelma                                 }
```

Lopuksi funktio `summa` konekielisenä:

```
PROGRAM.SUMMA.A_KOLME:  BEGIN { a_kolme }
  cs:0000 55             push   bp
  cs:0001 89E5           mov    bp,sp
  cs:0003 83EC02         sub    sp,0002
PROGRAM.7:  k:=1; { sijoitus globaaliin muuttujaan k }
  cs:0006 C7063C000100   mov    word ptr [003C],0001
PROGRAM.8:  i:=2; { sijoitus summan tilapäiseen parametriin i }
  cs:000C 8B7E04         mov    di,[bp+04]
  cs:000F 36C745060200   mov    ss:word ptr [di+06],0002
PROGRAM.9:  a:=3; { sijoitus summan lokaaliin muuttujaan a }
  cs:0015 8B7E04         mov    di,[bp+04]
  cs:0018 36C745FC0300   mov    ss:word ptr [di-04],0003
PROGRAM.10:  c:=4; { sijoitus a_kolmen lokaaliin muuttujaan c }
  cs:001E C746FE0400     mov    word ptr [bp-02],0004
PROGRAM.11:  END; { a_kolme }
  cs:0025 5D             pop    bp
  cs:0026 C20200         ret    0002
PROGRAM.SUMMA: BEGIN { summa }
  cs:0029 55             push   bp
  cs:002A 89E5           mov    bp,sp
  cs:002C 83EC06         sub    sp,0006
PROGRAM.13:  a_kolme;
  cs:002F 55             push   bp
  cs:0030 E8CDFF         call   PROGRAM.SUMMA.A_KOLME
PROGRAM.14:  b:=2;
  cs:0033 C746FA0200     mov    word ptr [bp-06],0002
PROGRAM.15:  summa:=i+j;
  cs:0038 8B4606         mov    ax,[bp+06]
  cs:003B 034604         add    ax,[bp+04]
  cs:003E 8946FE         mov    [bp-02],ax
PROGRAM.16: END; { summa }
  cs:0041 8B46FE         mov    ax,[bp-02]
  cs:0044 89EC           mov    sp,bp
  cs:0046 5D             pop    bp
  cs:0047 C20400         ret    0004
```

Huomattakoon, että käännöksessä on joitakin turhia rekistereiden sijoituksia.

Pino aliohjelman `a_kolme` lauseen `a:=3` jälkeen:



Siis lokaalia aliohjelmaa kutsuva ohjelma välittää pinossa osoittimen lokaaliin muuttujiin (frame pointer). Globaaliin muuttujaan viittaaminen on kaikkein suoraviivaisinta. Omaan lokaaliin viittaaminen toiseksi helpointa. Kaikkein hankalinta on viittaaminen ylempien tasojen lokaaleihin muuttujiin. Tehtävä hankaloituu vielä, mikäli tasoja on useampia.

Onkin suositeltavaa viitata ainoastaan globaaleihin muuttujiin (joihin viittaaminen taas ohjelmointiteknisistä syistä ei ole suositeltavaa!) ja aliohjelman omiin lokaaleihin muuttujiin sekä välitettyihin parametreihin.

7.1.5 Parametrin välittäminen

Siis Turbo Pascalissa kutsuva ohjelma laittaa parametrit pinoon. Arvoparametri laitetaan pinoon sellaisenaan ja muuttujaparametreistä välitetään osoite. Pinossa alimmaksi jää ensimmäinen parametri ja päällimmäiseksi viimeinen parametri (C-kielessä päinvastoin).

Tavukokoiset arvoparametritkin laitetaan pinoon sanana. Reaalilukutyyppejä lukuunottamatta kaikki 3-tavuiset ja yli 4-tavuiset tyypit välitetään osoitteen avulla myös arvoparametreinä ollessaan. Tällöin aliohjelman täytyy allokoida väliaikainen tila, johon aliohjelma kopioi välitettävän arvon. Merkkijonot välitetään aina osoitteen avulla (myös STRING[3]).

Funktio-aliohjelma palauttaa arvon rekistereissä lukuunottamatta merkkijonofunktiota. Tavukokoiset tyypit (CHAR, BOOLEAN, BYTE,...) palautetaan AL:ssä, sanakokoiset (INTEGER, WORD,...) AX:ssä, reaaliluvut (REAL-tyyppi) DX:BX:AX:ssä ja muut reaalilukutyypit 8087-prosessorin pinon päällä. Osoittimet ja pitkät kokonaisluvut palautetaan DX:AX:ssä.

Merkkijonofunktiossa kutsuva ohjelma tallettaa ennen kutsua muiden parametrien alapuolelle pinoon osoitteen väliaikaiseen merkkijonoon, johon tulos kopioidaan. Tätä osoitetta EI poisteta pinosta, kun aliohjelmasta poistutaan.

Hyvä nyrkkisääntö pinon siivoamiselle on siis se, että pinosta poistetaan ainoastaan ne parametrit, jotka on esitelty kutsussa olevien sulkujen välissä.

7.1.6 Merkkijonot

Turbo Pascalissa on merkkijonotyyppi `STRING`. Esimerkiksi

```
VAR nimi:STRING[30];
```

Merkkijonotyyppi varaa tilaa merkkijonon pituuden + 1 tavun todellisen pituuden talletta-mista varten. Pituustavu on merkkijonossa ensimmäisenä. Esimerkiksi sijoituksen

```
nimi:='Uuno';
```

jälkeen on muistipaikan `nimi` sisältö seuraava:

0	1	2	3	4	5	6	7
04	55	75	6E	6F	??	??	...

Merkkijonoja käsiteltäessä välitetään siis aina parametrinä jonon ensimmäisen tavun (siis pituuden) osoite. Merkkijonolle todellisuudessa varattua pituutta ei missään välitetä, tästä täytyy ohjelmoijan huolehtia itse. Periaatteessahan jokainen eri kokoiseksi varattu merkkijono on eri tyyppiä!

7.1.7 Taulukot

Taulukot talletetaan peräkkäisinä alkion kokoisina muistipaikkoina. Esimerkiksi määrityksestä:

```
TYPE coord=RECORD
             x      : INTEGER;
             y      : INTEGER;
             inside : BOOLEAN;
            END;
VAR  pisteet : ARRAY[1..10] OF coord;
...
 pisteet[1].x := 10;  pisteet[1].y:=256; pisteet[1].inside:=TRUE;
 pisteet[2].x :=  3;  pisteet[2].y:=513; pisteet[2].inside:=FALSE;
```

seuraa seuraavan näköinen muistin varaus:

```
            ┌────┐
pisteet +0  │ 0A │      pisteet[1].x lo
        +1  │ 00 │      pisteet[1].x hi
        +2  │ 00 │      pisteet[1].y lo
        +3  │ 01 │      pisteet[1].y hi
        +4  │ 01 │      pisteet[1].inside
        +5  │ 03 │      pisteet[2].x lo
        +6  │ 00 │      pisteet[2].x hi
        +7  │ 01 │      pisteet[2].y lo
        +8  │ 02 │      pisteet[2].y hi
        +9  │ 00 │      pisteet[2].inside
            └────┘
            ...          ....
```

Vastaavasti useampi ulotteiset taulukot talletetaan taulukkoina joiden alkioina ovat taulukot. Esimerkiksi seuraavat määritykset ovat identtisiä (tosin Pascalin kannalta eri tyyppiä!):

```
TYPE rivi   = ARRAY [1..2] OF INTEGER;
VAR  taulu1 : ARRAY [1..3] OF rivi;
     taulu2 : ARRAY [1..3,1..2] OF INTEGER;
BEGIN
  taulu1[1,1]  :=1;
  taulu1[2][1] :=5;
  taulu2[1,1]  :=1;
  taulu2[2][1] :=5;
END.
```

Huomattakoon, että Pascal-kielessä lause `taulu1[2][1]` samaistetaan lauseeseen `taulu1[2,1]`, kun taas assembler-kielessä se samaistettaisiin lauseeseen `taulu2[2+1]`.

Seuraavassa vielä näkymä Turbo Debuggerilla edellisen ohjelman ajoon:

```
┌CPU 80286──────────────────────────────────────────────────────────────┬──────────────┬─3─┐
│PROGRAM.5:  taulu1[1,1] :=1;                                           │ ax 0000      │c=0│
│  cs:0008 C7063C000100   mov    word ptr [PROGRAM.TAULU1],000          │ bx 6931      │z=1│
│PROGRAM.6:  taulu1[2][1] :=5;                                          │ cx 044D      │s=0│
│  cs:000E C70640000500   mov    word ptr [0040],0005                   │ dx 80D3      │o=0│
│PROGRAM.7:  taulu2[1,1] :=1;                                           │ si 0207      │p=1│
│  cs:0014 C70648000100   mov    word ptr [PROGRAM.TAULU2],000          │ di 0154      │a=0│
│PROGRAM.8:  taulu2[2][1] :=5;                                          │ bp 3FFC      │i=1│
│  cs:001A C7064C000500   mov    word ptr [004C],0005                   │ sp 3FFC      │d=0│
│PROGRAM.9: END.                                                        │ ds 697A      │   │
│  cs:0020>89EC           mov    sp,bp                                  │ es 697A      │   │
│  cs:0022 5D             pop    bp                                     │ ss 69A4      │   │
│  cs:0023 31C0           xor    ax,ax                                  │ cs 692E      │   │
├───────────────────────────────────────────────────────────────────────┤ ip 0020      │   │
│  ds:0000 00 00 00 00 00 00 00 00                                      ├──────────────┴───┘
│  ds:0008 A4 6D A4 6D A4 6D 00 00                                      │ ss:4000 0D20
│  ds:0010 00 00 00 00 00 00 A4 6D                                      │ ss:3FFE 0000
│  ds:0018 00 00 A4 6D 00 00 00 90                                      │ ss:3FFC>0000
└───────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────────────2─┐
│taulu2          ((1,0),(5,0),(0,0)) : ARRAY [1..2] OF ARRAY [1..3] OF INTEGER     │
│taulu1                        ((1,0),(5,0),(0,0)) : ARRAY [1..3] OF RIVI          │
└─────────────────────────────────────────────────────────────────────────────────┘
```

Taulukon alkiot ovat nollia, koska ne on selvyyden vuoksi nollattu debuggerissa. Ohjelmoijalla ei ole mitään lupaa olettaa, että alkiot olisivat nollia ilman alustamista.

Esimerkissä kääntäjä on voinut laskea kaikki osoitteet valmiiksi jo käännösaikana. Muutoin matriisin paikkaan (r,s) viitattaisiin kaavalla:

osoite = alkuosoite + k*((r-r_0)*sarakkeiden_lkm + (s-s_0)),

missä r_0 ja s_0 ovat ensimmäisen rivin ja ensimmäisen sarakkeen indeksit. k on yhden alkion koko tavuina.

Aivan vastaavalla periaatteella käsitellään myös useampiulotteiset taulukot.

7.1.8 Rekisterit jotka täytyy säilyttää

Kutsuva ohjelma olettaa aina, että kutsun jälkeen rekistereillä

BP, SP, SS ja DS

on niiden alkuperäiset arvot. Siis aliohjelma on velvollinen huolehtimaan siitä, ettei mainittuja rekistereitä muuteta.

7.1.9 Assembler-aliohjelmat

Seuraavassa on edellisten esimerkkien pääohjelma muutettu kutsumaan assembler-kielistä funktiota summa (Pascal-ohjelma voidaan kääntää komennolla `TPC summa /v` tarvitsematta aina mennä integroituun editoriin):

```
VAR k,i,j:INTEGER;
{$L summa} { Täytyy luetella tiedostot, joista .OBJ ohjelmia löytyy }
FUNCTION summa(i,j:INTEGER):INTEGER; EXTERNAL;
{ Ulkoiset aliohjelmat täytyy esitellä EXTERNAL }
BEGIN
  i:=4;
  j:=3;
  i:=summa(i,j);
END.
```

Funktioaliohjelma `summa` on kirjoitettu tiedostoon `SUMMA.ASM` ja käännetty MASM- tai TASM-kääntäjällä nimelle `SUMMA.OBJ`:

```
PUBLIC summa            ; Ulospäin näkyvät nimiöt esitetään julkisiksi
pino SEGMENT STACK      ; Tarvitaan vain Turbo Pascal 3.0 aliohjelmia
pino ENDS               ; varten jottei linkkeri anna virheilmoitusta
CODE SEGMENT            ; TURBO 5.0:ssa täytyy koodi olla segmentissä CODE
                        ;
ASSUME CS:CODE          ; Ei muita oletuksia, koska ei DATA segmenttiä
                        ;
; Funktio summa(i,j:INTEGER):INTEGER laskee yhteen  kaksi kokonaislukua
summa PROC NEAR
 PUSH BP
 MOV  BP,SP
 MOV  AX,[BP+04]        ; j
 ADD  AX,[BP+06]        ; +i
 POP  BP
 RET  4                 ; tulos jätetään AX rekisteriin
summa ENDP
;
CODE ENDS
 END
```

TURBO PASCAL 5.0:aa varten ei voi käyttää lyhennettyä segmenttimääritystä, joka toimisi myös MASM-kääntäjällä. TASM-kääntäjässä on määritys `.MODEL TPASCAL`, jonka avulla myös parametrinvälitys helpottuisi, mutta toistaiseksi kannattaa pyrkiä säilyttämään yhteensopivuus MASM-kääntäjään.
Mikäli halutaan viitata pääohjelman globaaleihin muuttujiin, pitää ne esitellä EXTRN-käskyllä.

Lokaalit staattiset muuttujat pitää sijoittaa segmenttiin DATA, ja niille EI SAA ANTAA alkuarvoja. Muut lokaalit muuttujat allokoidaan pinoon. Alkuarvoja vaativat muuttujat pitää laittaa koodisegmenttiin (tämä ei taas ole yhteensopivaa 80286 Protected-moden kanssa, ei siis toimi välttämättä OS/2:ssa).

Muutettaessa arvoparametrejä jotka on välitetty osoitteina (merkkijonot, taulukot yms.), täytyy parametristä tehdä kopio esimerkiksi pinoon ennen muuttamista.

7.1.10 Pascal-koodin nopeuttaminen

Koska arvoparametreille allokoidaan pinosta tilaa ja arvo kopioidaan pinoon, kannattaa joskus myös arvoparametrit välittää muuttajaparametreinä, mikäli niitä ei muuteta aliohjelmassa. Näin säästetään kopiointiin kuluva aika sekä vastaava tila pinosta. Yleensäkin kannattaa välttää turhia lokaaleja muuttujia sekä arvoparametrejä. Myös aliohjelmien sisäiset aliohjelmat generoivat lisää koodia ja pinon tarvetta.

Osa edellä mainituista sekoista on kuitenkin ristiriidassa hyvän Pascal-ohjelmointitavan kanssa, joten muutoksia kannattaa tehdä harkiten sekä kommentoida huolellisesti, miksi hyvästä ohjelmointitavasta on poikettu.

Turbo Pascal 5.0 kääntää kuhunkin aliohjelmaan pinon riittävyyden tarkistavan koodin, joka täydellisesti testatusta ohjelmasta voidaan tarvittaessa jättää pois kääntäjän optiolla pinon tarkistus $S-.

Kääntäminen kannattaa yleensä tehdä siten, että aritmetiikkaprosessorin emulointikirjasto tulee mukaan. Mikäli aritmetiikkaprosessori on koneessa ohjelman suoritusaikana, nopeuttaa tämä reaalilukulaskuja noin 10 kertaisesti.

7.1.11 Esimerkki

Aikaisemmin on kirjoitettu assembler aliohjelma POIKKEAMA. Tämä voitaisiin liittää Pascal-aliohjelmaksi seuraavan tiedoston avulla:

```
;*****************************************************************************
; Kopioi katkoviivojen välinen osa Turbo Pascal 5.0 pääohjelmaan.
COMMENT %
;-----------------------------------------------------------------------------
{$L poik.obj}
PROCEDURE poikkeama_ka(lkm:INTEGER; VAR vektori:taulukko;
                                    VAR max_poikkeama:INTEGER);  EXTERNAL;
;-----------------------------------------------------------------------------
%
;*****************************************************************************
CODE     SEGMENT
ASSUME   CS:CODE
include makrot.asm
PUBLIC poikkeama_ka
```

```
;*************************** poikkeama_ka *********************************
poikkeama_ka PROC NEAR
;
; PROCEDURE poikkeama_ka(lkm:INTEGER;
;                        VAR vektori:'ARRAY OF INTEGER';
;                        VAR max_poikkeama:INTEGER);
;
; Aliohjelmalla lasketaan vektorissa olevien alkioiden suurin poikkeama
; alkioiden keskiarvosta (tarkkuudella +/-1).
;
;
; Parametripino käskyn MOV BP,SP jälkeen
;
;    BP    ->   old BP
;    BP+02      paluuosoite
;    BP+04      max_poikkeaman osoite
;    BP+08      vektorin osoite
;    BP+0C      lkm
;
 PUSH BP
 MOV  BP,SP        ; Alustetaan pino.
 PUSH DS           ; Turbo Pascalissa täytyy säilyttää DS.
 LDS  SI,[BP+08H]  ; Vektorin osoite DS:SIhin.
 MOV  CX,[BP+0CH]  ; Alkioiden lukumäärä CX:ään.
 CALL poikkeama    ; Kutsutaan työn tekevää aliohjelmaa.
 LES  DI,[BP+04H]  ; Max_poikkeaman osoite ES:DIhin
 MOV  ES:[DI],AX   ; ja siirretään suurin poikkeama paikalleen.
 POP  DS           ; Palautetaan pino ennalleen.
 POP  BP
 RET  10
poikkeama_ka ENDP
INCLUDE  poikkeam.asm

CODE ENDS
 END
```

Lopuksi vielä esimerkki Pascal-pääohjelmasta:

```
PROGRAM testi(INPUT,OUTPUT);
TYPE taulukko=ARRAY [1..100] OF INTEGER;
VAR  lkm,max_poikkeama,i:INTEGER;
     vektori:taulukko;

{$L poik.obj}
PROCEDURE poikkeama_ka(lkm:INTEGER; VAR vektori:taulukko;
                                    VAR max_poikkeama:INTEGER);  EXTERNAL;

BEGIN
  WRITELN('Tulostetaan taulukon alkioiden suurin poikkeama keskiarvosta.');
  WRITE('Alkioiden lukumäärä >'); READLN(lkm);
  WRITELN('Paina kunkin alkion jälkeen [RETURN]:');
  FOR i:=1 TO lkm DO BEGIN
    WRITE(i,'. alkio >'); READLN(vektori[i]);
   END;
  poikkeama_ka(lkm,vektori,max_poikkeama);
  WRITELN('Suurin poikkeama keskiarvosta on ',max_poikkeama,'.');
END.
```

7.1.12 Parametrien nimeäminen

Edellä esitetyn aliohjelman vika on siinä, että itse ohjelmakoodissa esiintyy viittauksia pinoon. Mikäli itse ohjelman parametrien lukumäärää muutettaisiin, pitäisi jokainen lause `[BP+??]` käydä lävitse. Usein tämä unohtuu. Tästä ongelmasta selvittäisiin esimerkiksi seuraavasti:

Esitellään pinon kommentoimisen jälkeen parametrit nimellä ja viitataan niihin myöhemmin tällä esitellyllä nimellä.

```
...
;    BP+0C      lkm
;
; Parametrit:
max_poikkeama EQU [BP+04H]
vektori       EQU [BP+08H]
lkm           EQU [BP+0CH]
;-------------------------------------------------------
 PUSH BP
 MOV  BP,SP        ; Alustetaan pino.
 PUSH DS           ; Turbo Pascalissa täytyy säilyttää DS.
 LDS  SI,vektori   ; Vektorin osoite DS:SIhin.
 MOV  CX,lkm       ; Alkioiden lukumäärä CX:ään.
...
```

Nyt parametrien lukumäärän tai järjestyksen muuttuessa ei tarvitse muuttaa muita kuin EQU lauseita.

7.1.13 UNIT ja NEAR/FAR

Turbo Pascal 5.0:ssa voidaan aliohjelmia kirjoittaa omiksi yksiköikseen, UNITeiksi.

Seuraavassa alkeellinen kompleksilukuja käsittelevä ’aliohjelmakirjasto’, joka on kirjoitettu tiedostoon COMPLEX.PAS:

```
UNIT complex;
{************************************************************************}
INTERFACE     { Tämän alle kirjoitetut määritykset näkyvät niissä
                lohkoissa, joissa tämä UNIT on esitelty }

TYPE kompleksi_luku = RECORD
                        reaaliosa     : REAL;
                        imaginaariosa : REAL;
                      END;
VAR i,vakio:kompleksi_luku;
PROCEDURE yhteen(a,b:kompleksi_luku; VAR c:kompleksi_luku);
PROCEDURE lisaa_vakio(VAR c:kompleksi_luku);
PROCEDURE sijoita(ro,io:REAL; VAR c:kompleksi_luku);

{************************************************************************}
IMPLEMENTATION { Tämän alle kirjoitetut määritykset näkyvät vain tässä   }
VAR yksi:kompleksi_luku;

PROCEDURE summa(c1,c2:kompleksi_luku; VAR c3:kompleksi_luku);
BEGIN
  c3.reaaliosa:=c1.reaaliosa+c2.reaaliosa;
  c3.imaginaariosa:=c1.imaginaariosa+c2.imaginaariosa;
END;

PROCEDURE yhteen(a,b:kompleksi_luku; VAR c:kompleksi_luku);
BEGIN
  summa(a,b,c);
END;

PROCEDURE lisaa_vakio(VAR c:kompleksi_luku);
BEGIN
  summa(c,vakio,c);
END;

PROCEDURE sijoita(ro,io:REAL; VAR c:kompleksi_luku);
BEGIN
  c.reaaliosa:=ro;
  c.imaginaariosa:=io;
END;

{************************************************************************}
BEGIN                  { BEGIN-END. välinen osa suoritetaan 1 kerran
                         ohjelman käynnistyksen yhteydessä, käytetään
                         yleensä alustusten tekemiseen!                 }
  sijoita(0,1,i);      { Imaginääriyksiköksi i (0,1)                    }
  sijoita(1,0,yksi);   { Alustetaan yksi  (1,0):ksi                     }
  vakio:=yksi;
END.
```

’Kirjastoa’ voitaisiin kutsua vaikkapa seuraavasti:

```
PROGRAM kompleksi; { Testiohjelma complex-UNITia varten }
USES complex;
VAR  x,y:kompleksi_luku;
BEGIN
  sijoita(0,2,x);
  yhteen(x,i,y);
  lisaa_vakio(x);
  vakio:=y;
  lisaa_vakio(x);
END.
```

Turbo Pascal 5.0:ssa kaikki INTERFACE-osassa esitellyt aliohjelmat ovat FAR-tyyppisiä, eli kutsut niihin suoritetaan laittamalla pinoon sekä paluuosoite että segmentti. Muut aliohjelmat ovat NEAR-tyyppisiä.

Siis assembler-ohjelmoijan on jo ohjelman kirjoitusvaiheessa päätettävä esitelläänkö aliohjelmat INTERFACE-osassa vai ei. Mikäli esitellään, pitää aliohjelmat esitellä assemblerissa PROC FAR -lauseella ja pinossa pitää ottaa huomioon ylimääräinen sana.

Makro-esimerkissä tämä on otettu huomioon kirjoittamalla makrot siten, että ne voivat toimia kummassakin muistimallissa. Vasta käännösvaiheessa päätetään kumpaako mallia käytetään joko antamalla optio /Dfarprog tai jättämällä se pois. Esimerkki:

```
TASM poik /z /zi /Dfarprog;
```

7.1.14 MODEL TPASCAL

Turbo Assemblerin MODEL TPASCAL -komennolla voidaan myös helpottaa parametrien osoitteiden laskemista ja lokaalien muuttujien varaamista.

Poikkeama-aliohjelmaa kutsuva ohjelma voitaisiin kirjoittaa seuraavasti (tilan säästämiseksi kommentteja on vähennetty):

```
DOSSEG
.MODEL TPASCAL
INCLUDE  makrot.asm
.CODE

;***************************** poikkeama_ka *********************************
poikkeama_ka PROC NEAR lkm:WORD,vektori:DWORD,max_poikkeama:DWORD
PUBLIC      poikkeama_ka
;
; PROCEDURE poikkeama_ka(lkm:INTEGER; VAR vektori:'ARRAY OF INTEGER';
;                                     VAR max_poikkeama:INTEGER);
 PUSH DS
 LDS  SI,vektori
 MOV  CX,lkm
 CALL poikkeama
 LES  DI,max_poikkeama
 MOV  ES:[DI],AX
 POP  DS
 RET
poikkeama_ka ENDP
INCLUDE  poikkeam.asm

;***************************** lajittele ************************************
lajittele PROC NEAR lkm:WORD,vektori:DWORD
PUBLIC lajittele
;
; PROCEDURE lajittele(lkm:INTEGER; VAR vektori:'ARRAY OF INTEGER');
;
 PUSH DS
 LDS  SI,vektori
 MOV  CX,lkm
 CALL lajittelu
 POP  DS
 RET
lajittele ENDP
INCLUDE lajittelu.asm
;
 END
```

Siis MODEL TPASCAL -käsky generoi jokaisen PROC-sanan jälkeen normaalin pinon alustamisen. Vastaavasti jokainen RET-käsky aliohjelman sisällä korvataan pinon siivoamisella.

Menetelmän haittana on se, että pinon käsittely käännetään myös aliohjelmiin, joissa sitä ei tarvita.

Myös lokaalien muuttujien varaus helpottuu LOCAL-käskyllä:

```
LOCAL a:WORD, b:BYTE
```

Tällöin PROC sanan jälkeiseen pinon alustamiseen tulee automaattisesti SUB SP,4-lause.

7.1.15 Alustetut muistipaikat

Valmiiksi alustetut muistipaikat täytyy varata koodisegmentistä. Näitä käytettäessä on muistettava myös viitata koodisegmenttiin. 80286-prosessorin protected mode kieltää koodisegmenttiin kirjoittamisen. Yhteensopivuuden saamiseksi OS/2-käyttöjärjestelmään kannattaa siis välttää koodisegmenttiin kirjoittamista. Mikäli muuttujiin tarvitsisi kirjoittaa, voi tehdä alustusohjelman, joka kopioi koodisegmentissä olevat muuttujat datasegmentistä varattuun tilaan.

Seuraavassa esimerkissä on muunnostaulu, johon on talletettu eri kirjaimia vastaavat isot kirjaimet. Tämä taulu voidaan tallettaa ainoastaan koodisegmenttiin.

```
PUBLIC iso,isoksi
;*****************************************************************************
; Kopioi katkoviivojen välinen osa Turbo Pascal 5.0 pääohjelmaan.
COMMENT %
;-----------------------------------------------------------------------------
{$L isoksi.obj}
FUNCTION iso(c:CHAR):CHAR;            EXTERNAL;
FUNCTION isoksi(s:STRING):STRING;     EXTERNAL;
;-----------------------------------------------------------------------------
%
;INCLUDE  makrot.asm
CODE     SEGMENT
ASSUME   CS:CODE

;*************************** iso ************************************
iso PROC NEAR
;
; FUNCTION iso(c:CHAR):CHAR;
;
; Funktio-aliohjelmalla muutetaan merkki isoiksi kirjaimeksi
; käyttäen apuna muunnostaulukkoa.
;
; Input:   c               ; muutettava merkki
; Output:  funktion arvo   ; merkki muutettuna isoksi
; Muuttuu: liput,AL,BX
;
; VL 8.3.1989
;
; Pinon sisältö kutsun MOV BP,SP jälkeen:
;  BP ->       old_BP
;  [BP+02]     paluu_os
;  [BP+04]     c
;
c EQU [BP+04]
;
 PUSH BP
 MOV  BP,SP
 MOV  BX,OFFSET isona     ; Muunnostaulukon alkuosoite BX:ään.
 MOV  AL,c                ; Muunnettava merkki AL:ään.
 XLAT CS:[BX]             ; AL:ään merkki muunnostaulukosta
 POP  BP
 RET 2
iso ENDP

;************************** isoksi ************************************
```

```
isoksi PROC NEAR
;
; FUNCTION isoksi(s:STRING):STRING;
;
; Funktio-aliohjelmalla merkkijonon merkit isoiksi kirjaimiksi
; käyttäen apuna muunnostaulukkoa.
;
; Input:   s               ; muutettava merkkijono
; Output:  funktion arvo   ; merkkijono muutettuna isoksi
; Muuttuu: liput,AX,BX,CX,DI,SI
;
; Algoritmi:
;  Otetaan merkki kerrallaan ja muutetaan taulukon avulla isoksi.
;
; Rekistereiden käyttö:
;               CX       - laskuri
;               DS:SI    - osoitin seuraavaan merkkiin
;               ES:DI    - osoitin seuraavaan talletuspaikkaan
;               CS:BX    - osoitin muunnostaulukon alkuun
;               AL       - muunnettava merkki
;
; VL 8.3.1989
;
; Pinon sisältö kutsun MOV BP,SP jälkeen:
;  BP ->       old_BP
;  [BP+02]     paluu_os
;  [BP+04]     s:n osoite
;  [BP+08]     funktion arvon osoite (koska merkkijono)
;
s             EQU [BP+04]
funktion_arvo EQU [BP+08]
;
 PUSH BP
 MOV  BP,SP
 PUSH DS
 MOV  BX,OFFSET isona     ; Muunnostaulukon alkuosoite BX:ään.
 CLD                      ; Merkkijono-operaatiot ylöspäin.
 LDS  SI,s                ; Osoitin merkkijonon pituuteen DS:SI:hin.
 LES  DI,funktion_arvo    ; Osoitin funktion tulokseen ES:DI:hin.
 LODSB                    ; Merkkijonon pituus AL:ään.
 STOSB                    ; Ja talletetaan myös funktion arvoksi pituus.
 XOR  AH,AH               ; Merkkijonon pituus CX:ään.
 MOV  CX,AX
 JCXZ muunnos_valmis      ; Jos nollamittainen, niin ei muuntamista.
muunna:
 LODSB                    ; Seuraava merkki AL:ään
 XLAT CS:[BX]             ; AL:ään merkki muunnostaulukosta
 STOSB                    ; Ja talletetaan funktion tulokseksi
 LOOP muunna
muunnos_valmis:
 POP  DS
 POP  BP
 RET  4

;*************************** isona ***************************************
; Taulukko, jossa kutakin paikkaa vastaa vastaavan ison kirjaimen
; ASCII-koodi.
isona DB 000H,001H,002H,003H,004H,005H,006H,007H    ; 00-07
      DB 008H,009H,00AH,00BH,00CH,00DH,00EH,00FH    ; 08-0F
      DB 010H,011H,012H,013H,014H,015H,016H,017H    ; 10-17
      DB 018H,019H,01AH,01BH,01CH,01DH,01EH,01FH    ; 18-1F
      DB 020H,021H,022H,023H,024H,025H,026H,027H    ; 20-27
      DB 028H,029H,02AH,02BH,02CH,02DH,02EH,02FH    ; 28-2F
      DB 030H,031H,032H,033H,034H,035H,036H,037H    ; 30-37
      DB 038H,039H,03AH,03BH,03CH,03DH,03EH,03FH    ; 38-3F
      DB 040H,041H,042H,043H,044H,045H,046H,047H    ; 40-47
      DB 048H,049H,04AH,04BH,04CH,04DH,04EH,04FH    ; 48-4F
      DB 050H,051H,052H,053H,054H,055H,056H,057H    ; 50-57
      DB 058H,059H,05AH,05BH,05CH,05DH,05EH,05FH    ; 58-5F
      DB 060h,041h,042h,043h,044h,045h,046h,047h    ; 60-67 a-
      DB 048h,049h,04Ah,04Bh,04Ch,04Dh,04Eh,04Fh    ; 68-6F
      DB 050h,051h,052h,053h,054h,055h,056h,057h    ; 70-77
      DB 058h,059h,05Ah,07BH,07CH,07DH,07EH,07FH    ; 78-7F  -z
      DB 080H,09Ah,082H,083H,08Eh,085H,08Fh,087H    ; 80-87  ü ä å
      DB 088H,089H,08AH,08BH,08CH,08DH,08EH,08FH    ; 88-8F
      DB 090H,091H,092H,093h,099h,095H,096H,097H    ; 90-97  ö
      DB 098H,099H,09AH,09BH,09CH,09DH,09EH,09FH    ; 98-9F
      DB 0A0H,0A1H,0A2H,0A3H,0A4H,0A5H,0A6H,0A7H    ; A0-A7
      DB 0A8H,0A9H,0AAH,0ABH,0ACH,0ADH,0AEH,0AFH    ; A8-AF
      DB 0B0H,0B1H,0B2H,0B3H,0B4H,0B5H,0B6H,0B7H    ; B0-B7
      DB 0B8H,0B9H,0BAH,0BBH,0BCH,0BDH,0BEH,0BFH    ; B8-BF
      DB 0C0H,0C1H,0C2H,0C3H,0C4H,0C5H,0C6H,0C7H    ; C0-C7
      DB 0C8H,0C9H,0CAH,0CBH,0CCH,0CDH,0CEH,0CFH    ; C8-CF
      DB 0D0H,0D1H,0D2H,0D3H,0D4H,0D5H,0D6H,0D7H    ; D0-D7
      DB 0D8H,0D9H,0DAH,0DBH,0DCH,0DDH,0DEH,0DFH    ; D8-DF
      DB 0E0H,0E1H,0E2H,0E3H,0E4H,0E5H,0E6H,0E7H    ; E0-E7
      DB 0E8H,0E9H,0EAH,0EBH,0ECH,0EDH,0EEH,0EFH    ; E8-EF
```

```
      DB 0F0H,0F1H,0F2H,0F3H,0F4H,0F5H,0F6H,0F7H    ; F0-F7
      DB 0F8H,0F9H,0FAH,0FBH,0FCH,0FDH,0FEH,0FFH    ; F8-FF
isoksi ENDP
;*************************************************************************

CODE ENDS
     END
```

Edelliset aliohjelmat voitaisiin testata vaikkapa seuraavalla pääohjelmalla:

```
VAR s:STRING;
{$L isoksi.obj}
FUNCTION iso(c:CHAR):CHAR;            EXTERNAL;
FUNCTION isoksi(s:STRING):STRING;     EXTERNAL;
BEGIN
  WRITE  ('Anna merkkijono  > '); READLN(s);
  WRITELN('Alkaa merkillä   : ',iso(s[1]));
  WRITELN('Isoina kirjaimina: ',isoksi(s));
END.
```

7.2 Turbo C 2.0

7.2.1 Kääntäjän tekemä koodi

Edellisen kappaleen summa-esimerkki C-kielellä:

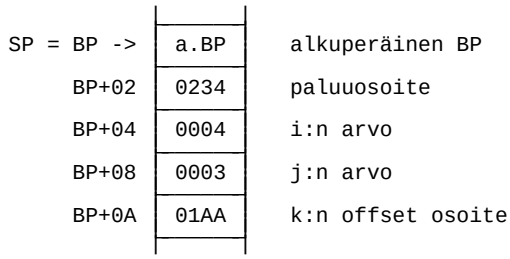
```
int k,i,j;
void summa(int i,int j,int *k)
{
  int a,b;
  a=1;
  b=2;
  *k = i + j;
}
int main(void)
{
  i=4;
  j=3;
  summa(i,j,&k);
  i=k;
  return 0;
}
```

Sekä vastaavasti assembleriksi käännettynä:

```
_summa: void summa(int i,int j,int *k)
  cs:0239 55             push   bp
  cs:023A 8BEC           mov    bp,sp
  cs:023C 83EC04         sub    sp,0004
#SUMMA#4:  a=1;
  cs:023F C746FE0100     mov    word ptr [bp-02],0001
#SUMMA#5:  b=2;
  cs:0244 C746FC0200     mov    word ptr [bp-04],0002
#SUMMA#6:  *k = i + j;
  cs:0249 8B4604         mov    ax,[bp+04]
  cs:024C 034606         add    ax,[bp+06]
  cs:024F 8B5E08         mov    bx,[bp+08]
  cs:0252 8907           mov    [bx],ax
#SUMMA#7: }
  cs:0256 5D             pop    bp
  cs:0257 C3             ret
_main: int main(void)
  cs:0258>55             push   bp
  cs:0259 8BEC           mov    bp,sp
#SUMMA#13:  i=4;
  cs:025B C7068A020400   mov    word ptr [_i],0004
#SUMMA#14:  j=3;
  cs:0261 C7068C020300   mov    word ptr [_j],0003
#SUMMA#15:  summa(i,j,&k);
  cs:0267 B88E02         mov    ax,028E
  cs:026A 50             push   ax
  cs:026B FF368C02       push   word ptr [_j]
  cs:026F FF368A02       push   word ptr [_i]
  cs:0273 E8C3FF         call   _summa
  cs:0276 83C406         add    sp,0006
#SUMMA#16:  i=k;
```

```
  cs:0279 A18E02          mov    ax,[_k]
  cs:027C A38A02          mov    [_i],ax
#SUMMA#17:  return 0;
  cs:027F 33C0            xor    ax,ax
  cs:0281 EB00            jmp    #SUMMA#18 (0283)
#SUMMA#18: }
  cs:0283 5D              pop    bp
  cs:0284 C3              ret
```

Edellinen käännös on tehty Borland C++ 1.0:n optioilla `-ms -r-`. Siis parametrin välitys on juuri päinvastoin kuin Pascalissa.



7.2.2 Parametrien järjestys

Pinoon jää päällimmäiseksi kutsun 1. parametri. Tällä on se etu, että voidaan välittää muuttuva määrä parametrejä, kun kutsun 1. parametrinä kerrotaan parametrien lukumäärä.

7.2.3 Pinon siivoaminen

Pinon siivoamisesta huolehtii kutsuva ohjelma, eikä aliohjelma kuten Pascalissa.

7.2.4 Assembler-aliohjelmat

Pääohjelma kirjoitetaan omaan tiedostoonsa, seuraavassa esimerkissä tiedostoon `SUM.C`:

```
int summa(int,int);

int main(void)
{
  int i,j;
  i = 4;
  j = 3;
  i = summa(i,j);
  return i;
}
```

Funktioaliohjelma `summa` kirjoitetaan tiedostoon `SUMMA.ASM`:

```
PUBLIC _summa
.MODEL SMALL
.CODE
; int summa(int i,int j);
_summa PROC NEAR
 PUSH BP
 MOV  BP,SP
 MOV  AX,[BP+06]   ; j
 ADD  AX,[BP+04]   ; i
 POP  BP
 RET               ; Tulos jätetään AX-rekisteriin
_summa ENDP
 END
```

Koska kääntäjä tekee myös pääohjelman tiedostosta `.OBJ` -tiedoston, ei pääohjelman tiedostolla ja aliohjelman tiedostolla voi nyt olla samaa nimeä kuten Turbo Pascal -esimerkissä!

Tulos voidaan kääntää esimerkiksi komentorivikääntäjällä

```
TCC sum.c summa.asm
```

7.2.5 Muistimalli

Pinoon talletettavien osoitteiden koko riippuu käytettävästä muistimallista. Esimerkissä on käytetty SMALL-mallia, jolloin lyhyet osoittimet riittävät sekä datalle että koodille. Tällöin on muistettava käyttää MOV-käskyä LDS käskyn sijasta (vain OFFSET). Kaikki data sijaitsee DS:än osoittamassa segmentissä. Lisäksi osoitteet vievät pinossa 2 tavua eikä 4 tavua kuten Pascal-aliohjelmissa.

muistimalli	paluuosoite	osoitin
tiny	2 tavua NEAR	2 tavua
small	2 tavua NEAR	2 tavua
medium	4 tavua FAR	2 tavua
compact	2 tavua NEAR	4 tavua
large	4 tavua FAR	4 tavua
huge	4 tavua FAR	4 tavua

7.2.6 Merkkijonot

C-kielessä merkkijonot päättyvät aina NULL-merkkiin (ASCII 0).

```
char st[8];
strcpy(st,"Uuno");
```

0	1	2	3	4	5	6	7
55	75	6E	6F	00	??	??	??

7.2.7 Esimerkki

Seuraavassa C-pääohjelma poik.c:

```
#include <stdio.h>

void poikkeama_ka(int lkm,int vektori[],int *max_poikkeama);
void lajittele(int lkm,int vektori[]);

int main (void)
{
  int lkm,max_poikkeama,i;
  int vektori[100];

  printf("Tulostetaan taulukon alkioiden suurin poikkeama keskiarvosta.\n");
  printf("Alkioiden lukumäärä >"); scanf("%d",&lkm);
  printf("Paina kunkin alkion jälkeen [RETURN]:\n");
  for (i=0;i<lkm;i++){
    printf("%2d. alkio >",i+1); scanf("%d",&vektori[i]);
   }
  poikkeama_ka(lkm,vektori,&max_poikkeama);
  printf("Suurin poikkeama keskiarvosta on %2d.\n",max_poikkeama);
  lajittele(lkm,vektori);
  printf("Alkiot suuruusjärjestyksessä:\n");
  for (i=0;i<lkm;i++) printf("%5d",vektori[i]); printf("\n");
  return 0;
}
```

POIKKEAMA-aliohjelmaa kutsuva assembler-ohjelma poikc.asm SMALL-muistimallin mukaan kirjoitettuna:

```
PUBLIC _poikkeama_ka,_lajittele
;*****************************************************************************
; Kopioi katkoviivojen välinen osa Turbo C 2.0 pääohjelmaan.
COMMENT %
;-----------------------------------------------------------------------------
extern void poikkeama_ka(int lkm,int vektori[], int *max_poikkeama);
extern void lajittele(int lkm,int vektori[]);
;-----------------------------------------------------------------------------
%
;*****************************************************************************
INCLUDE  makrot.asm
.MODEL SMALL
.CODE

;***************************** poikkeama_ka *********************************
_poikkeama_ka PROC NEAR
;
; void poikkeama_ka(int lkm,int vektori[], int *max_poikkeama);
;
; Aliohjelmalla lasketaan vektorissa olevien alkioiden suurin poikkeama
; alkioiden keskiarvosta (tarkkuudella +/-1).
;
;
; Pinon sisältö kutsun MOV BP,SP jälkeen:
;  BP ->       old_BP
;  [BP+02]     paluu_os
;  [BP+04]     lkm
;  [BP+06]     vektorin osoite
;  [BP+08]     max_poikkeaman osoite
 PUSH BP
 MOV  BP,SP
 PUSH SI
 MOV  SI,[BP+06]
 MOV  CX,[BP+04]
 CALL poikkeama
 MOV  SI,[BP+08]
 MOV  [SI],AX
 POP  SI
 POP  BP
 RET
_poikkeama_ka ENDP
INCLUDE  poikkeam.asm

;***************************** lajittele ************************************
_lajittele PROC NEAR
;
; void lajittele(int lkm,int vektori[]);
;
; Aliohjelmalla lajitellaan vektori nousevaan järjestykseen.
;
; Input:   lkm, vektori
; Output:  vektori
; Muuttuu: CX
; Käyttää: lajittelu-aliohjelmaa
;
; Pinon sisältö kutsun MOV BP,SP jälkeen:
;  BP ->       old_BP
;  [BP+02]     paluu_os
;  [BP+04]     lkm
;  [BP+06]     vektorin osoite
;
 PUSH BP
 MOV  BP,SP
 PUSH SI
 MOV  SI,[BP+06]
 MOV  CX,[BP+04]
 CALL lajittelu
 POP  SI
 POP  BP
 RET
_lajittele ENDP
INCLUDE lajittelu.asm

 END
```

7.2.8 Kääntäminen

Ohjelmat voidaan kääntää yhdeksi kokonaisuudeksi vaikkapa komennolla (-ms = model SMALL):

```
TCC -v -ms poik poikc.asm
```

Huomattakoon, että C-kielessä nimiöiden eteen tulee aina alleviivausmerkki.

Mikäli assembler-tiedosto käännetään erikseen, täytyy muistaa käyttää optiota /MX , joka säilyttää pienien ja isojen kirjaimien välisen eron. C-kielessä ei samaisteta isoja ja pieniä kirjaimia kuten PASCAL-kielessä.

Tällöin valmis objektitiedosto saadaan käännökseen mukaan komennolla (s=SMALL):

```
TCC -v -ms poik poikc.obj
```

Muistimalleissa, joissa DATAlle käytetään pitkiä osoitteita, pitää toimia kuten TURBO PASCAL- esimerkeissä, eli osoitteet vievät 4 tavua ja osoiteet siirretään `LDS` tai `LES`-käskyllä.

7.2.9 Säilytettävät rekisterit

Turbo C käyttää DI- ja SI-rekistereitä muuttujina, mikäli tämä on kääntäjän optiolla sallittu. Siis nämä rekisterit kannattaa säilyttää.

Luonnollisesti myös DS ja BP sekä SP tulee säilyttää.

7.2.10 Pascal-tyylinen parametrin välitys

Parametrin välitys saadaan samanlaiseksi kuin Turbo Pascalissa käyttämällä avainsanaa `pascal` aliohjelman esittelyn yhteydessä.
Aikaisemmin Pascalia varten tehty tiedosto `poik.asm` kelpaisi sellaisenaan, mikäli segmentin nimi `CODE` vaihdettaisiin `_TEXT` ja aliohjelmat esiteltäisiin C:ssä seuraavasti:

```
void pascal poikkeama_ka(int lkm,int far *vektori, int far *max_poikkeama);
void pascal lajittele(int lkm,int far *vektori);
```

Tällöin POIK.ASM on varminta kääntää erikseen ilman /MX-optiota, jollei globaaleja nimiä ole kirjoitettu isoilla kirjaimilla.

7.3 Turbo Pascal 4.0

Turbo Pascal 4.0 toimii kuten Turbo Pascal 5.0. Siis samalla tavalla kirjoitettu koodi kelpaa kummallekin kääntäjälle.

Aliohjelmien lokaalit muuttujat varataan kuitenkin pinosta päinvastaisessa järjestyksessä kuin Turbo Pascal 5.0:ssa. Varausjärjestykseen ei siis saa luottaa. Tämä koskee myös Pascal-ohjelmoijaa ja ABSOLUTE-käskyn käyttöä!

7.4 Turbo Pascal 5.5

Turbo Pascal 5.5 on Turbo Pascal 5.0:an oliokeskeiseen ohjelmointiin (object-oriented programming, oop) tarkoitettu versio. Mikäli olioita ei määritellä, toimii 5.5 aivan kuten 5.0.

Oliot välitetään parametrina osoitteen avulla. Mikäli kirjoitetaan olion metodia (olion määrittelyn sisällä esitelty aliohjelma), täytyy muistaa, että tällöin aina viimeisenä parametrina välitetään itse olion (self) osoite (on siis pinossa 1., eli osoitteessa `[BP+06]`). Metodeja kutsutaan aina pitkillä (`CALL FAR`) kutsuilla. Mikäli olio sisältää virtuaalisia metodeja, on olion muistialueella osoite olion metodeihin.

7.5 TURBO PASCAL 3.0

7.5.1 Erot Turbo Pascal 5.0:aan

Turbo Pascal 3.0 poikkeaa hieman Turbo Pascal 5.0:sta. Tärkeimmät erot ovat seuraavat:

- kaikki arvoparametrit välitetään pinossa
- ulkoinen aliohjelma ladataan .BIN-muodossa, siis muistin kuvana ja se täytyy olla täysin vapaasti sijoittuva
- kutsut ovat aina NEAR-tyyppiä
- aliohjelman suoritus aloitetaan tiedoston alusta ellei toisin ole mainittu
- ei globaaleja symboleja assembler-ohjelmasta tai assembler-ohjelmaan

7.5.2 Kutsu Pascal-ohjelmasta

Konekieliset aliohjelmat esitellään Pascal-ohjelmassa seuraavasti:

```
PROGRAM testi(INPUT,OUTPUT);
TYPE taulukko=ARRAY [1..100] OF INTEGER;
VAR  lkm,max_poikkeama,i:INTEGER;
     vektori:taulukko;

PROCEDURE poik; EXTERNAL 'poik3.bin';
PROCEDURE poikkeama_ka(lkm:INTEGER; VAR vektori:taulukko;
                       VAR max_poikkeama:INTEGER);  EXTERNAL poik[0];
PROCEDURE lajittele(lkm:INTEGER; VAR vektori:taulukko); EXTERNAL poik[3];

BEGIN
  WRITELN('Tulostetaan taulukon alkioiden suurin poikkeama keskiarvosta.');
... loppu kuten Turbo Pascal 5.0 esimerkissä.
```

Aliohjelmat on siis esitelty löydettäväksi tiedostosta `poik3.bin` ja tiedoston alusta käytetään jatkossa nimeä `poik`. Kunkin aliohjelman sijainti tiedoston alusta on kerrottu `EXTERNAL`-lauseen perässä olevalla tiedoston nimellä ja suhteellisella siirtymällä.

7.5.3 Esittely Assembler-aliohjelmassa

Assembler-aliohjelma on varminta kirjoittaa siten, että tiedoston ensimmäisiksi suoritettaviksi lauseiksi tulevat hypyt varsinaisiin aliohjelmiin. Tällöin hypyistä tulee 3 tavuisia joten Pascal-ohjelmassa kukin hyppy on 3 tavua kauempana kuin edellinen hyppy.

Turbo Pascal 5.0:aa varten kirjoitettu tiedosto muutettaisiin alusta seuraavan näköiseksi:

```
;*****************************************************************************
;
; Kopioi katkoviivojen välinen osa Turbo Pascal 3.0 pääohjelmaan.
COMMENT %
;-----------------------------------------------------------------------------
PROCEDURE poik; EXTERNAL 'poik3.bin';
PROCEDURE poikkeama_ka(lkm:INTEGER; VAR vektori:taulukko;
                       VAR max_poikkeama:INTEGER);  EXTERNAL poik[0];
PROCEDURE lajittele(lkm:INTEGER; VAR vektori:taulukko); EXTERNAL poik[3];
;-----------------------------------------------------------------------------
%
;*****************************************************************************
;
INCLUDE  makrot.asm
CODE     SEGMENT
ASSUME   CS:CODE

;***************************** hypyt aliohjelmiin ***************************
;
JMP poikkeama_ka   ;   [0]
JMP lajittele      ;   [3]

;***************************** poikkeama_ka *********************************
;
poikkeama_ka PROC NEAR
;
; PROCEDURE poikkeama_ka(lkm:INTEGER;
... loppu kuten aikaisemmin
```

Turbo Pascal 5.0:aan verrattuna lisää on tullut vain hypyt varsinaisiin aliohjelmiin. Näin ollen samaa tiedostoa voitaisiin käyttää myös Turbo Pascal 5.0 aliohjelmana.

7.5.4 Kääntäminen

Mikäli esimerkin Assembler-aliohjelma olisi kirjoitettu nimelle `poik3.asm` voitaisiin se kääntää binääritiedostoksi `poik3.bin` vaikka seuraavilla komennoilla:

```
TASM  poik3 /z;
TLINK poik3;
EXE2BIN poik3
```

Kääntämisen jälkeen kannattaa tuhota tarpeettomat tiedostot `poik3.exe` ja `poik3.obj`. Mikäli linkitysvaiheessa tuleva varoitus pinon puutteesta häiritsee, voidaan pinosegmentti lisätä assembler-ohjelmaan.

7.5.5 Vapaasti sijoittuva koodi

Erittäin tärkeätä Turbo Pascal 3.0 -aliohjelmia tehtäessä on muistaa, että koodin tulee olla täysin vapaasti sijoittuvaa. Kääntäminen ja linkittäminen on tehty jo ennen Pascal-ohjelman kääntämistä, joten käytettäviin osoitteisiin ei enää voida vaikuttaa. Hyppyjen osalta tämä ei häiritse, koska hypyt ovat muutenkin suhteellisia.

Eniten hankaluuksia tulee koodisegmenttiin sijoitettujen muuttujien käytöstä. Esimerkiksi seuraavat viittaukset eivät toimi:

```
CODE SEGMENT
ASSUME CS:CODE
lkm   DW ?
isona DB 000H,001H,002H,003H,004H,005H,006H,007H    ; 00-07
 ...
 MOV  AX,lkm    ; kääntyy MOV  AX,CS:lkm
 MOV  BX,OFFSET isona
```

Assembler-kääntäjä ja linkkeri kääntävät eo. esimerkeissä muistipaikkojen `lkm` ja `isona` sijainnin siten, kuten ne ovat käännösaikana. Kuitenkin Turbo Pascal 3.0 saattaa sijoittaa koodin alkamaan vaikkapa osoitteesta 175H. Tällöin kukin muistipaikkaviittaus on vastaavasti 175H liian pieni. Tämä ongelma voidaan poistaa esimerkiksi seuraavan makron avulla:

```
;--------------------------- ota os -------------------------------------
otaos MACRO muuttuja,reg
 LOCAL next
;
; Makrolla lasketaan koodisegmentissä olevan muuttujan ajonaikainen osoite
; rekisteriin.
;
; Input:    muuttuja        - muuttuja, jonka osoite lasketaan
;           reg             - rekisteri, johon tulos laitetaan
; Output:   reg
; Muuttuu:  liput,reg
;
 CALL next                                ; next:in ajonaikainen osoite pinoon.
next:
 POP reg                                  ; => saadaan tämän käskyn osoite
 ADD reg,(OFFSET muuttuja)-(OFFSET next) ; + muuttujan ja POP reg:in väli.
 ENDM
```

Nyt edelliset kutsut voitaisiin suorittaa seuraavasti:

```
 otaos lkm,BX                          ; BX:ään lkm:n AJONAIKAINEN OSOITE
 MOV  AX,CS:[BX]
 ADD  BX,(OFFSET isona - OFFSET lkm)  ; muutetaan BX isona osoitteeksi
```

Siis jotta aikaisemmin esitetty `isoksi` -aliohjelma toimisi Turbo Pascal 3.0:ssa, pitää siinä oleva lause

```
MOV  BX,OFFSET isona
```

muuttaa lauseeksi

```
otaos isona,BX
```

Tietysti pitää muistaa kirjoittaa myös tarvittava hyppytaulukko tiedoston alkuun.

7.5.6 Parametrien välitys

Edellä esitetyllä tavalla kirjoitettu aliohjelma toimii myös Turbo Pascal 5.0:ssa. Mikäli kuitenkin välitetään arvoparametrejä, pitää muistaa, että Turbo Pascal 3.0:ssa arvoparametrit välitetään pinossa arvoina, ei osoitteena.

Mikäli halutaan kirjoittaa koodia, joka on käytettävissä kummassakin kääntäjän versiossa, kannattaa kirjoittaa kummankin kielen vaatimat tavat sekä käyttää ehdollista kääntämistä valitsemaan käytettävä kieli. Esimerkiksi `isoksi`-aliohjelmassa voitaisiin kirjoittaa:

```
IFDEF turbo3     ; Mikäli kääntäjän optio /Dturbo3
 MOV  SI,SS
 MOV  DS,SI
 LEA  SI,[BP+04] ; Osoitin merkkijonon pituuteen DS:SI:hin
ELSE             ; Muuten TURBO 5.0-koodia
 LDS  SI,[BP+04] ; Osoitin merkkijonon pituuteen DS:SI:hin
ENDIF
```

Edellä myös pinon esittely pitää muistaa kirjoittaa ehdollisesti s:n osoitteeksi tai s:n arvoksi (merkkijonon määrittelyn pituus +1 tavua).

7.6 INLINE-koodi

Mikäli assembler-aliohjelma halutaan toimivan mahdollisimman monen eri kielisen aliohjelman osana, kannattaa itse aliohjelma kirjoittaa assemblerilla kutsuttavaksi aliohjelmaksi kuten `poikkeama`-aliohjelman yhteydessä tehtiin. Kutakin tarvittavaa ohjelmointikieltä varten kirjoitetaan sitten lyhyt kutsuva aliohjelma.

Ohjelman kutsuminen toisella aliohjelmalla hidastaa tietysti hieman ohjelman suoritusta, mutta useimmissa tapauksissa itse aliohjelman suoritus kestää kuitenkin niin kauan, että kutsun tuoma lisäaika on varsin merkityksetön kokonaisajan rinnalla.

Mikäli itse aliohjelma on kuitenkin hyvin lyhyt, kannattaa ehkä harkita konekielisen koodin sijoittamista suoraan ohjelmakoodin sekaan. Tämä onnistuu usein `INLINE`-käskyllä.

7.6.1 Turbo Pascal

Kaikissa Turbo Pascalin versioissa on käytettävissä INLINE-käsky, jolla voidaan kirjoittaa konekielistä koodia suoraan INLINE-käskyn osoittamaan paikkaan. Tästä on jo aikaisemmin ollut esimerkkinä `INLINE($CC)` -käsky Turbo Pascal 3.0 ohjelmien debuggauksen yhteydessä.

Oletetaan, että ongelmana on vaikkapa kokonaislukupisteiden (x_1,y_1) ja (x_2,y_2) kautta kulkevan janan y-koordinaatin laskeminen pisteessä x. Tämähän saadaan kaavasta:

$$(y-y_1) = (y_2-y_1)/(x_2-x_1) * (x-x_1).$$

Ongelmaksi tuleekin lähinnä laskemisen nopeus, mikäli laskemisessa käytetään apuna reaalilukuaritmetiikkaa. Toisaalta kokonaislukuaritmetiikassa tarkkuus häviää jakolaskussa. Tarkkuus säilyisi, mikäli laskut ryhmitellään uudestaan siten, että ensin suoritetaan kertolasku ja vasta sitten jakolasku. Tosin tällöin kertolaskun tulos saattaa tuottaa ylivuodon kokonaislukualueelta.

Ongelma voidaan ratkaista laskemalla kertolaskun välitulos 32 bittiseksi kokonaisluvuksi, jonka jakaminen tuottaa jälleen 16 bittisen luvun, mikäli oletetaan, etteivät pisteet x_1 ja x_2 ole samoja ja piste x on niiden välissä.

Ilman pyöristystä koodi olisi assemblerilla kirjoitettuna suurinpiirtein seuraava:

```
MOV  AX,y2
MOV  BX,y1
SUB  AX,BX   ; AX:=(y2-y1)
MOV  DX,x
MOV  DI,x1
SUB  DX,DI   ; DX:=(x-x1)
MOV  CX,x2
SUB  CX,DI   ; CX:=(x2-x1)
IMUL DX      ; DX AX := (y2-y1)*(x-x1)
IDIV CX      ; AX    := (y2-y1)*(x-x1)/(x2-x1)
ADD  AX,BX   ; AX    := (y2-y1)*(x-x1)/(x2-x1) + y1
MOV  y,AX    ; Janan y-koordinaatti paikalleen
```

Oletetaan, että muuttujat ovat kaikki pääohjelman globaaleja muuttujia. INLINE-koodin kirjoittamista varten koodi kirjoitetaan assemblerilla ja käännetään siten, että tulokseksi saadaan myös .LST-tiedosto:

```
Turbo Assembler  Version 1.0        15.03.89 22.20.52         Page 1
JANA.ASM
      1 0000                         .model small
      2 0000                         .data
      3 0000  ????                   x1 DW ?
      4 0002  ????                   y1 DW ?
      5 0004  ????                   x2 DW ?
      6 0006  ????                   y2 DW ?
      7 0008  ????                   x  DW ?
      8 000A  ????                   y  DW ?
      9 000C                         .code
     10 0000  A1 0006r                MOV  AX,y2
     11 0003  8B 1E 0002r             MOV  BX,y1
     12 0007  2B C3                   SUB  AX,BX   ; AX:=(y2-y1)
     13 0009  8B 16 0008r             MOV  DX,x
     14 000D  8B 3E 0000r             MOV  DI,x1
... jne
```

Kaikki osoitteet korvataan heksakoodissa vastaavilla muuttujien nimillä ja näin tulokseksi saadaan seuraava koodi, joka sijoitetaan PASCAL-ohjelmaan:

```
VAR x,y,x1,y1,x2,y2:INTEGER;
...
y2:=5;
INLINE(     { Lasketaan y:=y1+(y2-y1)/(x2-x1)*(x-x1)                    }
 $A1/y2     { MOV  AX,y2                                               }
/$8B/$1E/y1 { MOV  BX,y1                                               }
/$2B/$C3    { SUB  AX,BX   ; AX:=(y2-y1)                               }
/$8B/$16/x  { MOV  DX,x                                                }
/$8B/$3E/x1 { MOV  DI,x1                                               }
/$2B/$D7    { SUB  DX,DI   ; DX:=(x-x1)                                }
/$8B/$0E/x2 { MOV  CX,x2                                               }
/$2B/$CF    { SUB  CX,DI   ; CX:=(x2-x1)                               }
/$F7/$EA    { IMUL DX      ; DX AX := (y2-y1)*(x-x1)                   }
/$F7/$F9    { IDIV CX      ; AX    := (y2-y1)*(x-x1)/(x2-x1)           }
/$03/$C3    { ADD  AX,BX   ; AX    := (y2-y1)*(x-x1)/(x2-x1) + y1      }
/$A3/y      { MOV  y,AX    ; Janan y-koordinaatti paikalleen           }
);
...
```

7.6.2 Turbo Pascal 5.0

Turbo Pascal 5.0:ssa on normaalin INLINE-koodin lisäksi mahdollisuus kirjoittaa INLINE-makroja, jotka siis korvaavat käännösaikana makron kutsun vastaavalla INLINE-koodilla. Parametrin välitys INLINE-makroille tapahtuu kuten aliohjelmillekin, paitsi ettei pinossa ole paluuosoitetta. Paluutahan ei varsinaisesti ole, vaan suoritus jatkuu makron jälkeisestä paikasta aivan normaalisti.

Janan pisteen laskeminen voitaisiin suorittaa esimerkiksi seuraavan makron avulla:

```
FUNCTION MulDiv(a,b,c:INTEGER):INTEGER; INLINE(
{ Makro-funktiolla lasketaan kerto ja jakolasku (a*b)/c          }
{ siten, että välitulos lasketaan 32 bittiseksi.  Tulos katkais- }
{ 16 bittiseksi kokonaisluvuksi.                                 }
 $59               {  POP    CX          ; CX:=c                 }
/$58               {  POP    AX          ; AX:=b                 }
/$5A               {  POP    DX          ; DX:=a                 }
/$F7/$EA           {  IMUL   DX          ; DX AX := a*b          }
/$F7/$F9           {  IDIV   CX          ; AX := (a*b)/c         }
);
...
  y2:=5;
  y:=MulDiv(y2-y1,x-x1,x2-x1);
...
```

INLINE-makron etu tavalliseen INLINE-koodiin verrattuna on se, että parametrejä on helpompi vaihtaa. Tavalliseen aliohjelmaan verrattuna makro on nopeampi, koska kutsua ja paluuta ei tehdä.

Lyhyitä INLINE-koodeja kirjoitettaessa nopein tapa heksakoodin saamiseksi voi olla käskyjen kirjoittaminen debuggerissa, josta U-komennolla saadaan sitten heksalistaus.

7.6.3 Turbo Pascal 6.0

Turbo Pascal 6.0:ssa on sisäänrakennettu, osittain TASM-yhteensopiva, assembler-kääntäjä. Lisäksi Turbo Pascal 6.0:n integroituun debuggeriin on lisätty rekisteri-ikkuna (saadaan käyttöön [ALT-W][R]). Ikkunoiden kokoa voidaan muuttaa myös hiirellä oikean alanurkan "kulmamerkistä" tarttumalla.

Intergoidun debuggerin ja Turbo Pascal 6.0:n ASM-lohkon ansioista Turbo Pascal 6.0 on ehkä tämän hetken helpoin väline assemblerin kirjoittamiseen ja varsinkin kokeilemiseen:

```
PROGRAM tp6_malli(INPUT,OUTPUT);

TYPE koord = RECORD
  x,y :INTEGER;
 END;

VAR p1,p2:koord; y:INTEGER;

PROCEDURE laske_y(p1,p2:koord; x:INTEGER; VAR y:INTEGER);
{ Aliohjelmalla lasketaan kokonaislukupisteiden
 p1 ja p2 määräämän suoran pistettä x vastaava lähin
 kokonaisluku y-arvo.
 Algoritmi:
 y:=y1+(y2-y1)/(x2-x1)*(x-x1)
}
BEGIN
 ASM
   MOV AX,p2.y    ;
   MOV BX,p1.y    ;
   SUB AX,BX      ; { AX:=(y2-y1)                          }
   MOV DX,x       ;
   MOV DI,p1.x    ;
   SUB DX,DI      ; { DX:=(x-x1)                           }
   MOV CX,p2.x    ;
   SUB CX,DI      ; { CX:=(x2-x1)                          }
   IMUL DX        ; { DX AX := (y2-y1)*(x-x1)              }
   IDIV CX        ; { AX := ( ) * ( ) / (x2-x1)            }
   ADD AX,BX      ; { AX := ( ) * ( ) / ( ) + y1           }
   LES DI,y       ; { ES:[DI]:hin y:n osoite               }
   MOV ES:[DI],AX ; { y:= tulos                            }
 END;
END;

BEGIN
 p1.x:=1; p1.y:=2;
 p2.x:=6; p2.y:=9;
 laske_y(p1,p2,4,y);
 WRITELN('y=',y);
END.
```

Edellinen esimerkki on kirjoitettu proseduuri-aliohjelmaksi osoittamaan parametrin välitystä paremmin. Siis arvoparametreihin voidaan viitata suoraan nimellä, mutta muuttujaparametreihin viitataan osoitteen avulla.

Mikäli aliohjelma olisi kirjoitettu funktio-aliohjelmaksi, korvattaisiin kaksi viimeistä assembler-riviä lauseella

```
MOV @Result,AX
```

Koodia voidaan vielä hieman parantaa, mikäli kyseessä on puhdas assembler-aliohjelma (kuten edellä), jättämällä aliohjelman BEGIN ja END rivit pois sekä esittelemällä aliohjelma tyyliin:

```
PROCEDURE laske_y(p1,p2:koord; x:INTEGER; VAR y:INTEGER); ASSEMBLER;
ASM
  { .. lauseet ... }
END;
```

Assembleriksi esitellyssä funktiossa 16-bittinen tulos palautetaan AX-rekisterissä, @Result ei ole käytössä!

Kommentit kannattaa kirjoittaa esitetyllä tavalla, jolloin sama koodi kelpaa sekä TASM-kääntäjälle, että TURBO 6.0:lle.

ASM-koodin kirjoittajan on kuitenkin muistettava huolehtia BP,SP, DS ja SS -rekistereiden arvon säilyttämisestä!

Nimiöiden pitää olla esiteltynä Pascalin LABEL-osassa. Mikäli nimiö alkaa @-merkillä, on kyseessä lokaali nimiö, jota EI tarvitse esitellä LABEL-osassa. Lisäksi nimiöllä alkavat `DB` -yms. lauseet ovat kiellettyjä:

```
{ Sallittu: }
  ASM
    DB 5
  END;
{ Ei sallittu: }
  ASM
    luku DB 5
  END;
```

7.6.4 Turbo C 2.0

Turbo C:ssä INLINE-koodia vastaa `__emit__(arvo,...)` funktio. Tämä funktio toimii kuten Turbo Pascalin INLINE-koodi, eli arvot sijoitetaan sellaisenaan lopulliseen ohjelmakoodiin.

```
 ...
 __emit__(0xFA); /* CLI - kielletään keskeytykset */
 ...
 __emit__(0xFB); /* STI - sallitaan keskeytykset  */
```

Koodin sekaan voidaan kirjoittaa myös suoraan assembler-kielisiä rivejä asm-komennon perään. Käännösvaiheessa käytetään sitten apuna TASM-kääntäjää assembler-kielisten rivien kääntämisessä. Seuraavassa C-ohjelma, jossa lasketaan janan pisteen y-koordinaatti:

```
int x,y,x1,y1,x2,y2;
main () {
 ...
 y2 = 5;
 /* Lasketaan y:=y1+(y2-y1)/(x2-x1)*(x-x1)                          */
 asm MOV  AX,y2   ; /*                                              */
 asm MOV  BX,y1   ; /*                                              */
 asm SUB  AX,BX   ; /* AX:=(y2-y1)                                  */
 asm MOV  DX,x    ; /*                                              */
 asm MOV  DI,x1   ; /*                                              */
 asm SUB  DX,DI   ; /* DX:=(x-x1)                                   */
 asm MOV  CX,x2   ; /*                                              */
 asm SUB  CX,DI   ; /* CX:=(x2-x1)                                  */
 asm IMUL DX      ; /* DX AX := (y2-y1)*(x-x1)                      */
 asm IDIV CX      ; /* AX    := (y2-y1)*(x-x1)/(x2-x1)              */
 asm ADD  AX,BX   ; /* AX    := (y2-y1)*(x-x1)/(x2-x1) + y1         */
 asm MOV  y,AX    ; /* Janan y-koordinaatti paikalleen              */
 y2 = y;
... jne.
}
```

Koodia voidaan kirjoittaa myös seuraavasti (ainakin kääntäjän uudemmissa versioissa, Turbo C++ 1.0 eteenpäin):

```
asm {
  MOV AX,y2;
  MOV BX,y1;
  ...
  MOV y,AX;
}
y2 = y;
```

Kääntäjä tunnistaa koodista SI ja DI rekistereiden käytön, joten niiden tallettamisesta ei tarvitse huolehtia.

Mikäli asm-rivejä kirjoitetaan, ei integroitua kääntäjää pystytä käyttämään, vaan käännös on suoritettava komentorivikääntäjällä TCC. Uudemmissa versioissa on sisäänrakennettu assembler-kääntäjä ja käännös voidaan tehdä myös integroidusta ympäristöstä.

Koodin sekaan kirjoitetut asm-rivin muuttujat ovat tyypittömiä, joten esimerkiksi riviä

```
int i;
...
asm inc i;
```

ei pystytä kääntämään oikein, vaan se on kirjoitettava

```
asm inc WORD PTR i;
```

Inline assembler -komentoja kirjoitettaessa sotketaan C-kääntäjän kirjanpito rekistereiden yms. arvoista, joten koodin optimointi ei voi olla aivan parasta mahdollista. Tämän takia kannattaa joskus verrata debuggerilla koodia, jossa on asm-komentoja ja jossa ei ole. Muutaman kellokierroksen nopeutuksesta yhdessä tai kahdessa käskyssä ei ole paljon hyötyä, mikäli satoja kellokierroksia hävitään huonommin optimoidun koodin takia.

Luku 8

Tehtäviä

8.1 Pascalin ja C:n rakenteita

Kirjoita seuraavia Pascal-rakenteita vastaavat assembler-ohjelman pätkät:

8.1.1 FOR-silmukka

```
s:=0;                              s=0;
FOR i:=n1 TO n2 DO s:=s+i;         for (i=n1; i<=n2; i++) s+=i;

s:=0;                              s=0;
FOR i:=1 TO n DO s:=s+1;           for (i=1; i<=n; i++; s++);
```

8.1.2 WHILE-silmukka

```
s:=0; i:=1;                        s=0; i=1;
WHILE i<=n DO BEGIN                while ( i <= n ) {
  s:=s+i; i:=i+2;                    s += i; i += 2;
END;                               }
```

8.1.3 REPEAT-silmukka

```
s:=0; i:=1;                        s=0; i=1;
REPEAT                             do {
  s:=s+i; i:=i+2;                    s += i; i += 2;
UNTIL s>16*i;                      } while ( s <= 16*i )
```

8.2 Rekursio

Kirjoita rekursiivinen assembler-aliohjelma TULO, joka laskee tulon AX*BX kaavasta

```
TULO(AX,BX) = TULO(AX-1,BX) + BX , AX>0
TULO(AX,BX) = 0                  , AX=0
```

Aliohjelma palauttaa tulon rekisterissä AX.

8.3 Laajennettu kertolasku

Kirjoita assembler-aliohjelma joka kertoo 32-bittisen luvun 16-bittisellä luvulla. Vihje: Käytä hyväksesi valmista kertolaskua ja muistele alekkain kertomista 10-järjestelmässä.

8.4 Laajennettu jakolasku

Kirjoita assembler-aliohjelma joka jakaa 32-bittisen luvun 16-bittisellä luvulla siten, että tulos voi olla vielä 32-bittinen. Vihje: Käytä valmista jakolaskua ja muistele jakokulmassa jakamista 10-järjestelmässä.

8.5 Lajittelu

Kirjoita loppuun seuraava assembler-aliohjelma:

```
;--------------------------- lajittele --------------------
lajittele PROC NEAR
;
; Aliohjelmalla lajitellaan paikasta DS:SI alkava
; tavuvektori nousevaan järjestykseen
;
;  Input:   DS:SI          - vektorin alkuosoite
;           CX             - alkioiden lukumäärä
;  Output:  DS:[SI]        - lajiteltu vektori
;  Muuttuu: liput,DS:[SI]
;  Kutsuu:  etsi_suurin
;
;  Algoritmi:
;    1.  etsitään osoittimen kohdasta alkavan osavektorin suurin alkio
;    2.  vaihdetaan suurin osoittimen kohdalla olevan kanssa
;    3.  siirretään osoitinta eteenpäin
;    4.  mikäli ei vektorin lopussa, jatketaan 1:stä
;
;  Rekistereiden käyttö:
```

8.6 Harjoitustyö

Harjoitustyössä kirjoitetaan assembler-aliohjelma Turbo Pascal 5.0:aan tai vaihtoehtoisesti Turbo Pascal 3.0:aan tai Turbo C 2.0:aan tai uudempaan.

Harjoitustyössä listataan sekä tarvittavat assembler-aliohjelmat että Pascal- tai C-kielinen testiohjelma. Listaukset leikataan A4-kokoisiksi sivuiksi jotka niitataan vasemmasta yläkulmasta yhteen työn aiheella, tekijän nimellä ja päivämäärällä varustetun kansilehden kanssa.

Tässä monisteessa esitettyjä makroja ei tarvitse listata, mikäli niitä käytetään sellaisenaan. Muutetut tai omatekoiset makrot täytyy myös listata.

Erityistä työselostusta ei tarvitse tehdä, mutta ohjelmat tulee dokumentoida riittävän hyvin.

Jonkin assembler-aliohjelman kesto täytyy laskea kellokierroksina ja ilmoittaa osan kesto muodossa

```
suoritusaika = (lukujen_lkm * 105) + 23 kellokierrosta
```

Luku 9

Tyypilliset virheet

Assembler ohjelmoinnissa voi tehdä kahdenlaisia virheitä: loogisia virheitä ja syntaksivirheitä. Kääntäjä ilmoittaa useimmiten kielioppivirheistä. Joskus makrojen käyttäminen saattaa aiheuttaa odottamatonta koodia, joten koodin oikeellisuus kannattaa tarkistaa .LST-tiedostosta. Mikäli assembler-ohjelmassa ei ole käytetty PUBLIC ja EXTERNAL-lauseita, ei linkittäjä ehkä osaa käyttää tehtyjä aliohjelmia.

Loogiset virheet ovat kuitenkin huomattavasti hankalampia, koska kääntäjä ei pysty niistä ilmoittamaan. Osa loogisista virheistä saattaa jäädä jopa piileviksi ja ohjelma voi käyttäytyä satunnaisesti. Tässä luvussa käsitellään joitakin tyypillisiä ohjelmointivirheitä.

9.1 Alustamattomat

Ohjelman satunnaista käyttäytymistä aiheuttaa erityisesti alustamattomat liput, rekisterit ja muistipaikat.

9.1.1 Liput

Koska yleensä merkkijono-operaatiot (`STOS, LODS, MOVS` jne.) tehdään ylöspäin, on suuntalippukin valmiiksi oikeassa asennossa. Tällöin testattava ohjelma saattaa toimia oikein jopa ilman suuntalipun asettamista, mutta mikäli ohjelmaa kutsutaan merkkijonoja alaspäin käyttävän ohjelman jälkeen, ei ohjelma enää toimikaan. Siis `CLD` tai `STD` -käskyjä ei saa unohtaa.

Joskus käytetään jatkettua yhteenlaskua (`ADC`) tai vähennyslaskua (`SBB`). Tällöin lasku suoritetaan siis ottamalla huomioon edellä tullut muistinumeron arvo. Mikäli tällaista laskentaa suoritetaan silmukassa, täytyy ennen silmukan aloittamista varmistua muistinumeron oikeasta arvosta. Sama koskee muistinumerolipun kautta tapahtuvia pyöritysoperaatioita.

9.1.2 Rekisterit

Jakolasku (`DIV`, `IDIV`) tapahtuu aina rekisteriparille DX AX tai AH AL. Ennen jakolaskua täytyy jaettavan yläosaan saada oikea arvo. Tämä seuraa joko edellisistä laskuoperaatioista tai sitten täytyy käyttää `MOV DX,0` tai `CWD` tyyppisiä käskyjä.

Segmenttirekisterin väärä arvo on yksi hyvin yleinen virhe. Ohjelman lataaja laittaa vain koodisegmentin ja pinosegmentin kohdalleen. Kaikki muut segmentit täytyy ohjelmoijan alustaa itse. Aliohjelmiin tultaessa on segmenttirekistereissä yleensä kutsuvan ohjelman arvot. Tällöinkin segmenttirekisterit täytyy alustaa tarvittaessa.

Ei suinkaan saa olettaa, että kaikki aliohjelman tarvitsemat muuttujat olisivat samassa segmentissä. Esimerkiksi lomitteluun on helppo keksiä pääohjelma, jossa lomiteltavat vektorit ovat eri segmenteissä ja lomittelun tulos vielä omassa segmentissään. Tällöin 8086-prosessorin segmenttirekisterit eivät riitä ja apuna täytyy käyttää joitakin apumuistipaikkoja tai rekistereitä.

LOOP-käsky käyttää CX-rekisteriä. Usein kierrosmäärä saadaan kuitenkin AL-rekisteristä, joten pitää muistaa nollata CH-rekisteri tavalla tai toisella.

9.1.3 Muistipaikat

Joskus rekistereitä käytetään apumuistipaikkoina. Riippumatta siitä, onko muistipaikka rekisterissä vai todellisessa muistissa, pitää muistipaikat alustaa ennen käyttöä. (Jotkut korkeamman tason kielten kääntäjät jopa varoittavat alustamattomista muistipaikoista, mikäli niitä käytetään ennen alustamista.)

9.2 Hypyt

Lippujen asentoon perustuvia hyppyjä on sekä etumerkittömiä (`JA, JB, JAE ja JBE`) sekä etumerkillisiä (`JG, JL, JGE ja JLE`). Valitsemalla väärä hyppy, voidaan saada odottamattomia tuloksia.

9.3 Pino

Pinon väärä käyttö johtaa usein koneen täydelliseen jumiintumiseen.

9.3.1 RET

Aliohjelman lopusta voi unohtua kokonaan paluukäsky `RET`. Pascal-aliohjelman paluussa pitää muistaa siivota pino täsmälleen oikeankokoisella `RET koko` -käskyllä.

`RET`-käskyn koko täytyy myös täsmätä `CALL`-käskyyn (vrt. `PROC NEAR` ja `PROC FAR`).

9.3.2 PUSH ja POP

Pinosta otettavien tavaroiden määrä täytyy olla täsmälleen sama kuin sinne laitettavien. Siis ohjelman `PUSH` ja `POP` -käskyjen tulee täsmätä toisiinsa.

9.3.3 VAR

Pascal-pääohjelmassa jokin parametri on saatettu esitellä muuttujaparametriksi ja sitä käytetään kuitenkin kuten arvoparametriä tai päinvastoin. Siis Pascal-ohjelmassa `VAR`-käskyn käytön kanssa tulee olla tarkkana.

9.3.4 Ylivuoto

Pinon koko saattaa olla varattu niin pieneksi, ettei sinne mahdu kaikki aliohjelmien tarvitsema tila. Tällaisissa tapauksissa aliohjelmien täytyy itse tehdä oma pino, jossa on riittävästi tilaa. Pinon vaihdon aikana pitää kieltää keskeytykset.

Erityisesti keskeytyspalvelijoissa on oltava huolellinen pinon käytössä, koska pinoa saattaa olla jäljellä vain hyvin vähän.

9.4 Sivuvaikutukset

Varsin yleinen virhe on myös erilaiset sivuvaikutukset.

9.4.1 Rekisterit

Aliohjelma saattaa käyttää ja muuttaa kutsuvan ohjelman tarvitsemia rekistereitä. Hyvä tapa on tallettaa kaikki tarpeelliset rekisterit pinoon ja palauttaa ne sieltä aliohjelman lopuksi.

Kertolasku (`MUL`, `IMUL`) laittaa aina tuloksen rekisteripariin `DX AX` tai `AH AL`. Tällöin pitää muistaa, että vastaavasti menetetään joko rekisterin DX tai rekisterin AH alkuperäinen arvo.

Merkkijono-operaatiot muuttavat myös rekistereiden (`STOS -> DI, LODS -> SI, MOVS -> SI,DI` jne.) arvoja.

9.4.2 Liput

Jotkut käskyt (esim. `MOV AX,0`) eivät muuta lippujen arvoja kun taas jotkut toiset (esim. `ADD AX,5`) muuttavat.

Myös aliohjelmista palattaessa lippujen arvot ovat usein muuttuneet ja niihin ei saa luottaa.

9.4.3 Muistipaikat

Aliohjelmat saattavat muuttaa globaaleja muistipaikkoja. Tällaiset muutokset on hyvä kommentoida aliohjelmien esittelyissä.

9.5 Muita virheitä

9.5.1 Väärä sisääntulokohta

Aliohjelman suoritusta ei aina ehkä aloitetakaan ohjelmoijan olettamasta paikasta. Tämä koskee erityisesti Turbo Pascal 3.0 aliohjelmia ja assembler-kielisiä pääohjelmia. Muulloinkin väärään väliin luetellut muistipaikkojen varaukset saattavat aiheuttaa yllätyksiä.

VÄÄRIN:

```
oma_ali PROC NEAR
lkm    DB ?   ; kutsu CALL oma_ali  aloittaisi tästä!!!!!
suurin DW ?
 PUSH AX      ; eikä tästä kuten pitäisi.
 ...
oma_ali ENDP  ; Siis muuttujat ennen PROC sanaa!
```

9.5.2 Segmentit

Mikäli käskyssä käytetään muuta kuin käskyyn kuuluvaa oletussegmenttiä, täytyy käyttää segmentinvaihtokäskyä.

Erityisesti pitää muistaa, että STOS -tallettaa aina ES:DI:n osoittamaan paikkaan.

Paha virhe on myös segmentin ylitys. Esimerkiksi osoitteeseen FFFF tehtävän STOSB-käskyn jälkeen seuraava STOSB-käsky tallettaa tavun saman segmentin osoitteeseen 0000.

9.5.3 Käskyt

Eri prosessoreiden assembler-kielissä operandien järjestys vaihtelee. 8086-assemblerissa kohde on vasemmanpuoleinen operandi. Siis

```
MOV  AX,BX  ;  siirretään AX:ään BX:än sisältö
```

9.5.4 Ymmärtämättömyys

Ei pidä luulla, että tietokoneen muistissa olisi ohjelmaa, tavuja, kokonaislukuja, reaalilukuja tai merkkijonoja. Prosessorin kannalta muisti on aina täysin samanlaista sisällöstä riippumatta. Tiettyinä hetkinä jotakin muistin kohtaa saatetaan tulkita ohjelmaksi ja sitä suoritetaan kuten ohjelmaa. Vastaavasti jokin bittijono jossakin kohti muistia tulkitaan kokonaisluvuksi, jokin toinen ehkä merkkijonoksi jne.

Ohjelmoija on se, joka tekee muistille jonkin merkityksen tekemällä ohjelmaa, joka tulkitsee muistia halutulla tavalla!

9.5.5 Kirjoitusvirheet

Kirjoitusvirheetkin saattavat aiheuttaa loogisia virheitä. Tietysti useimmiten kirjoitusvirhe johtaa samalla syntaksivirheeseen, mutta kun käytetään lähes samannimisiä muistipaikkoja, aliohjelmia jne, on tämä mahdollista. Esimerkiksi rekistereiden nimissä voi helposti tapahtua sekaannus:

```
PUSH BX   ; vaikka pitäisi olla PUSH BP
```

Luku 10

Tehtävien vastauksia

10.1 Pascal-rakenteita

10.1.1 FOR-silmukka

Indeksi tarvitaan:

```
; s:=0;
; FOR i:=n1 TO n2 DO s:=s+i;
 XOR  AX,AX   ; AX:=s:=0
 MOV  SI,n1   ; SI:=i
 MOV  CX,n2   ; kierrosten lukumäärä = (n2-n1)+1
 SUB  CX,SI   ; (n2-n1)
 ADD  CX,1    ; +1. INC CX ei laittaisi C-lippua!
 JLE  ei_kierroksia
for_i_n1_to_n2:
 ADD  AX,SI   ; s:=s+i
 INC  SI      ; i:=i+1
 LOOP for_i_n1_to_n2
ei_kierroksia:
```

Vaihtoehtoisesti:

```
; s:=0;
; FOR i:=n1 TO n2 DO s:=s+i;
 XOR  AX,AX    ; AX:=s:=0
 MOV  SI,n1    ; SI:=i
for_i_n1_to_n2:
 CMP  SI,n2
 JG   ei_kierroksia
 ADD  AX,SI    ; s:=s+i
 INC  SI       ; i:=i+1
 JMP  for_n1_to_n2
ei_kierroksia:
```

Indeksiä ei tarvita:

```
; s:=0;
; FOR i:=1 TO n DO s:=s+1;
 XOR  AX,AX   ; s:=0;
 MOV  CX,n    ; kierrosten lukumäärä, n>0
for_i_1_to_n:
 INC  AX      ; s:=s+1
 LOOP for_i_1_to_n
```

10.1.2 WHILE-silmukka

```
; s:=0; i:=1;
; WHILE i<=n DO BEGIN s:=s+i; i:=i+2; END;
 XOR  AX,AX   ; s:=0;
 MOV  SI,1    ; i:=1;
while_i:
 CMP  SI,n    ; i>n?
 JG   while_end
 ADD  AX,SI   ; s:=s+i
 ADD  SI,2    ; i:=i+2
 JMP  while_i
while_end:
```

10.1.3 REPEAT-silmukka

```
; s:=0; i:=1;
; REPEAT s:=s+i; i:=i+2; UNTIL s>16*i;
 XOR  AX,AX  ; s:=0
 MOV  SI,1   ; i:=1
 MOV  CX,4   ; 16*i  = i SHL 4
repeat:
 ADD  AX,SI  ; s:=s+i
 ADD  SI,2   ; i:=i+2
 MOV  DI,SI
 SHL  DI,CL  ; 16*i
 CMP  AX,DI  ; s>16*i?
 JLE  repeat
```

10.2 Rekursio

```
;--------------------------- tulo ----------------------------
tulo PROC NEAR
;
; Aliohjelmalla lasketaan tulo rekusiivisesti kaavasta:
;
;  TULO(AX,BX) = TULO(AX-1,BX) + BX , AX>0
;  TULO(AX,BX) = 0                  , AX=0
;
; Input:    AX,BX       ; kerrottavat
; Output:   AX          ; tulos
; Muuttuu:  liput,AX
;
 CMP  AX,0
 JZ   paluu
 DEC  AX       ; TULO(AX-1,BX)
 CALL tulo
 ADD  AX,BX    ;  + BX
paluu:
 RET
tulo ENDP
```

10.3 Laajennettu kertolasku

```
;---------------------------- long mul ----------------------------------
long_mul PROC NEAR
;
; Aliohjelmalla kerrotaan DXAX:ssä oleva 32 bittinen etumerkitön luku
; CX:ssä olevalla 16 bittisellä luvulla.  32 bittinen tulos palautetaan
; DXAX:ssä ja 16 bittinen ylivuoto-osa BX:ssä.
;
; Input:    DXAX,CX
; Output:   DXAX,BX
; Muuttuu:  liput,AX,BX,DX,DI
;
; Algoritmi:
;   10-järjestelmässä kerrotaan esimerkiksi 29*7 seuraavasti:
;
;           2 9
;       *     7
;         1_____
; 7*9       6 3
; 7*2   + 1 4
;         _____
;         2 0 3     ; 2 on ylivuoto 2 numeroiseen tulokseen
;
;  Sama rekistereillä:              Muuttujat:
;                                   DXa AXa - alkuperäiset DX ja AX
;          DXa AXa                  DX1 AX1 - tulo AXa*CX
;  *            CX                  DX2 AX2 - tulo DXa*CX
;       c _________                 BX      - jemmataan DXa
;          DX1 AX1                  DI      - jemmataan DX1
;  +   DX2 AX2
;      ____________
;      yli DXt AX1
;
; Vesa Lappalainen 28.3.1989                          | Rekisterit:
;                                                     | AX  BX  DX  DI
;                                                     | AXa  ?  DXa ?
 MOV  BX,DX   ; Jemmataan DXa.                       | AXa DXa DXa ?
 MUL  CX      ; Kerrotaan ensin AXa CX:llä.          | AX1 DXa DX1 ?
 XCHG AX,BX   ; Jemmataan AX1 ja otetaan DXa.        | DXa AX1 DX1 ?
```

```
 MOV  DI,DX   ; Jemmataan DX1.                          | DXa AX1 DX1 DX1
 MUL  CX      ; Kerrotaan DXa  CX:llä.                  | AX2 AX1 DX2 DX1
 ADD  AX,DI   ; DX1+AX2                                 | DXt AX1 DX2 DX1
 ADC  DX,0    ; muistinumero lisätään DX2:een           | DXt AX1 yli DX1
 XCHG BX,DX   ; Ylivuoto DX2 BX:ään ja AX1 DX:ään.      | DXt yli AX1 DX1
 XCHG AX,DX   ; DX1+AX2 DX:ään ja AX1 AX:ään.           | AX1 yli DXt DX1
 RET
long_mul ENDP
```

10.4 Laajennettu jakolasku

```
;---------------------------- long div ----------------------------------
long_div PROC NEAR
;
; Aliohjelmalla jaetaan DXAX:ssä oleva 32 bittinen etumerkitön luku
; CX:ssä olevalla 16 bittisellä luvulla.  32 bittinen kokonaisosa
; palautetaan DXAX:ssä ja 16 bittinen jakojäännös BX:ssä.
;
; Input:    DXAX,CX
; Output:   DXAX,BX
; Muuttuu:  liput,AX,BX,DX
;
; Algoritmi:
;   10-järjestelmässä jaetaan esimerkiksi 95/7 seuraavasti:
;
;           1  3
;         ┌─────
;     7   │ 9  5
;       - 7
;         ---
;           2  5    (2  < 7 , itse asiassa 2 = 9 MOD 7)
;         - 2  1
;           ----
;              4    ( 4 = 25 MOD 7 )
;
;  Sama rekistereillä:              Muuttujat:
;                                   DXa AXa - alkuperäiset DX ja AX
;                                   AX1 DX1 - jaon DXa/CX tulos
;           AX1 AX2                 AX2 DX2 - jaon DX1 AXa/CX tulos
;         ┌─────────                BX      - jemmataan AXa ja AX1
;    CX   │ DXa AXa
;           DX1 AXa  (DX1 < CX !, DX1 = DXa MOD CX)
;               DX2
;
; Vesa Lappalainen 17.2.1988                            | Rekisterit:
;                                                       | AX  BX  DX
;                                                       | AXa  ?  DXa
 MOV  BX,AX   ; Jemmataan AXa.                          | AXa AXa DXa
 MOV  AX,DX   ; Jaetaan ensin DXa CX:llä,               | DXa AXa DXa
 XOR  DX,DX   ; jota varten DX pitää nollata.           | DXa AXa  0
 DIV  CX      ;                                         | AX1 AXa DX1
 XCHG AX,BX   ; Jemmataan AX1 ja otetaan AXa.           | AXa AX1 DX1
 DIV  CX      ; Jaetaan DX1 AXa CX:llä (DX1 < CX !!!)   | AX2 AX1 DX2
 XCHG BX,DX   ; Jakojäännös DX2->BX,kok.alkuosa->DX.    | AX2 DX2 AX1
 RET
long_div ENDP
```

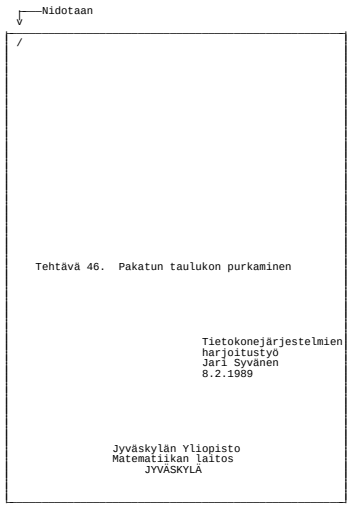
10.5 Lajittelu

```
;========================== lajittelu ==================================
lajittelu PROC NEAR
;
; Aliohjelmalla lajitellaan paikasta DS:SI alkava
; kokonaislukuvektori laskevaan järjestykseen.
;
;  Input:   DS:SI          - vektorin alkuosoite
;           CX             - alkioiden lukumäärä
;  Output:  DS:[SI]        - lajiteltu vektori
;  Muuttuu: liput,DS:[SI]
;  Kutsuu:  etsi_suurin
;
;  Algoritmi:
;    1.  Etsitään osoittimen kohdasta alkavan osavektorin suurin alkio.
;    2.  Vaihdetaan suurin osoittimen kohdalla olevan kanssa.
;    3.  Siirretään osoitinta eteenpäin.
;    4.  Mikäli ei vektorin lopussa, jatketaan 1:stä.
;
;  Rekistereiden käyttö:
;    DS:SI  -  osoitin osavektorin alkuun
```

```
;    DS:DI  -  osoitin osavektorin suurimpaan alkioon
;    CX     -  osavektorin alkioiden lukumäärä
;
; Vesa Lappalainen 21.3.1989
;
 push_reg <AX,CX,DI,SI,ES>
 JCXZ  valmis
osa_vektori:
 CALL etsi_suurin
 XCHG [SI],AX      ; Osavektorin 1. := suurin, AX:=1. alkio.
 MOV  [DI],AX      ; Suurimman paikalle osavektorin 1.
 INC  SI           ; Osoitin seuraavaan osavektorin alkuun.
 INC  SI
 LOOP osa_vektori
valmis:
 pop_reg  <ES,SI,DI,CX,AX>
 RET
lajittelu ENDP

;========================== etsi suurin ==================================
etsi_suurin PROC NEAR
;
; Aliohjelmalla etsitään paikasta DS:SI alkavan vektorin suurin alkio.
;
;  Input:   DS:SI          - vektorin alkuosoite
;           CX             - alkioiden lukumäärä
;  Output:  AX             - suurin alkio
;           DI             - osoitin suurimpaan alkioon
;  Muuttuu: liput,AX,DI
;  Kutsuu:  -
;
;  Algoritmi:
;    0.  asetetaan suurin = 1. alkio
;    1.  otetaan seuraava alkio
;    2.  mikäli suurempi kuin suurin, niin suurin:=alkio, osoitin alkioon
;    3.  mikäli ei vektorin lopussa, jatketaan 1:stä
;
;  Rekistereiden käyttö:
;    DS:SI  -  osoitin vektorin alkuun
;    DS:DI  -  osoitin vektorin suurimpaan alkioon
;    CX     -  vektorin alkioiden lukumäärä
;    BX     -  suurin alkio
;
 push_reg <BX,CX,SI>
 CLD
 MOV  BX,[SI]       ; Suurin := 1. alkio
 MOV  DI,SI
 ADD  DI,2          ; Osoitin tähän astista suurinta seuraavaan. (Lopuksi -2)
onko_tama_suurin:
 LODSW              ; Otetaan seuraava alkio.
 CMP  AX,BX         ; Onko suurin?
 JLE  ei_suurin
 MOV  DI,SI         ; On, osoitin siihen (tai oikeastaan sitä seuraavaan).
 MOV  BX,AX         ; Suurin:=alkio.
ei_suurin:
 LOOP onko_tama_suurin
 SUB  DI,2          ; Siirretään osoitin itse alkioon.
 MOV  AX,BX         ; Suurin palautetaan myös AX:ssä.
 pop_reg <SI,CX,BX>
 RET
etsi_suurin ENDP
```

10.6 Harjoitustyö

10.6.1 Kansilehti



10.6.2 Ohjelmalistaus

```
;*************************************************************************
; Tietokonejärjestelmät -kurssin harjoitustyö      Jari Syvänen 8.2.1989
;
;-------------------------------------------------------------------------
; Tehtävä 46 : Tee assembler-aliohjelma, joka purkaa pakatun 3-bittisiä
;              kokonaislukuja sisältävän taulukon.
;-------------------------------------------------------------------------
;
; Mahdollinen käytännön sovellutus on näyttömuistin pixelin väri, jolloin
; saadaan kolmella bitillä hallittua 8 väriä.
; Toisaalta tehtävä voidaan tulkita myös kokonaisluvun muuttamiseksi
; oktaaliluvuksi.
;
; Ohjaajan kanssa sovittiin, että taulukko on pakattu seuraavalla tavalla:
; Esimerkiksi 17 pakattua lukua (2,0,6,7,1,4,7,5,5,2,5,2,6,3,5,1,2)
; neljässä sanassa (21B9,4F6A,559D,1500):
;    1  2  3  4  5    6  7  8  9 10   11 12 13 14 15   16 17
;
;   | 2 | 0 | 6 | 7 | 1 |  | 4 | 7 | 5 | 5 | 2 |  | 5 | 2 | 6 | 3 | 5 |  | 1 | 2 |
;
;  |0010000110111001|0100111101101010|0101010110011101|0001010100000000|
;
;  5432109876543210 5432109876543210 5432109876543210 5432109876543210
;       1. sana          2. sana          3. sana           4. sana
;       21B9             4F6A             559D              1500
;
; Sanan bitit 15 eivät ole pakatussa taulukossa mukana, joten ne
; jätetään huomiotta.
;
; Alkioiden järjestys määräytyy kuvan yläpuolella annetulla tavalla.
; Esimerkissä on 15 pakattua lukua 3 kokonaisessa sanassa ja
; neljännessä sanassa on 2 lukua.  (3 = 17 DIV 5 ja 2 = 17 MOD 5).
;
;-------------------------------------------------------------------------
;Määritykset
;
CODE SEGMENT
ASSUME CS:CODE

;============================ purku ==================================
PUBLIC purku
purku PROC NEAR
;
; Procedure purku(n:integer;var a:pakattu;var b:purettu);
;
; Taulukossa a on pakattu taulukko 3 bittisiä kokonaislukuja.
; Taulukko on pakattu siten, että sanassa on viisi lukua ja
; sanan ylin bitti on merkityksetön, eli siis ei kuulu taulukkoon.
; Kokonaisluku n ilmoittaa pakattujen alkioiden määrän.
; Aliohjelma palauttaa puretun taulukon taulukossa b.
```

```
;
; Input:   a         - pakattu taulukko, (ylösp.pyör. n/5 kok.lukua)
;          n         - taulukon alkioiden lukumäärä
; Output:  b         - purettu taulukko (n kokonaislukua)
; Muuttuu: AX,BX,CX,DX,DI,SI,ES,liput
; Käyttää: PaloitteleJaTallenna -aliohjelmaa
;
; Algoritmi:
;   0.  Laitetaan osoitin pakattuun ja purettavaan taulukkoon.
;   1.  Lasketaan kokonaisten 5 lukua sisältävien lukumäärä (n DIV 5)
;       sekä viimeiseen sanaan jäävien lukumäärä (n MOD 5)
;   2.  Puretaan kokonaiset sanat.
;   3.  Puretaan viimeinen sana, jos siinä jotakin.
;   4.  Valmis
;
; Muuttujat:
;   DS:SI   - osoitin pakattuun taulukkoon
;   AX      - sana pakatusta taulukosta
;   ES:DI   - osoitin purettuun taulukkoon
;   CX      - sanasta talletettavien lukujen lukumäärä
;   DX      - kokonaisten sanojen lukumäärä
;   pinossa - viimeisessä sanassa olevien lukujen määrä
;
; Pinon sisältö kutsun MOV BP,SP jälkeen:
;  BP ->    old_BP
;  [BP+02]  paluu_os
;  [BP+04]  b:n osoite
;  [BP+08]  a:n osoite
;  [BP+0C]  n (arvo)
;-------------------------------------------------------------------------
PUSH BP                    ; Pinon alustus.
MOV  BP,SP
PUSH DS                    ; DS täytyy säilyttää.
;---------------------- Osoittimet paikalleen: ----------------------------
 LDS  SI,[BP+08H]          ; DS:SI osoittaa a:n ensimmäiseen alkioon.
 LES  DI,[BP+04H]          ; ES:DI osoittaa b:n ensimmäiseen alkioon.
;---------------------- Lasketaan kokonaisten lukumäärä: ------------------
 MOV  CX,[BP+0CH]          ; Alkioiden lukumäärä CX:ään.
 JCXZ valmis               ; Ellei alkioita ole, niin ei tehdä mitään.
 XOR  DX,DX                ; Nollataan DX jakolaskua varten.
 MOV  AX,CX                ; Lasketaan CX/5:
 MOV  BX,5                 ; Jaetaan sanaan pakattujen alkioiden määrällä.
 DIV  BX                   ; DX on jakojäännös, eli viim. sanassa olev. lkm.
 PUSH DX                   ; Viim. sanassa olevien määrä talteen
 MOV  DX,AX                ; Kokonaisten sanojen määrä laskuriin DX.
;---------------------- Puretaan kokonaiset sanat: ------------------------
 CLD                       ; Suunta +.
kokonaiset_sanat:          ;
 LODSW                     ; Siirretään purettava alkio DS:SIstä akkuun.
 MOV  CX,5                 ; Kerralla otettavien lukumäärä.
 CALL PaloitteleJaTallenna ; Puretaan sana kerrallaan.
 DEC  DX
 JNZ  kokonaiset_sanat     ; Kunnes DX on nolla.
;---------------------- Viimeinen sana: -----------------------------------
 POP  CX                   ; Viimeisessä olevien lukumäärä.
 JCXZ valmis               ; Jollei yhtään, niin lopetetaan.
 LODSW                     ; Otetaan viimeinen vajaa sana käsittelyyn.
 CALL PaloitteleJaTallenna ; Puretaan viimeisestä CX kappaletta lukuja.
;---------------------- Valmis: -------------------------------------------
valmis:                    ;
 POP  DS                   ; Palautetaan pinoon talletettu DS.
 POP  BP                   ; Palautetaan pino.
 RET  10                   ; Palataan ja siivotaan parametrit.
purku ENDP
;-------------------------------------------------------------------------

;======================== Paloittele ja tallenna ==========================
PaloitteleJaTallenna PROC NEAR
;
; Otetaan AX:stä CX-kertaa kolme bittiä ja siirretään muistipaikkaan johon
; ES:DI osoittaa sekä DI:=DI+2.
;
; Input:   AX        - pakattu sana
;          CX        - purettavien lukujen lukumäärä
;          ES:DI     - osoitin talletettavaan paikkaan
;          DF+       - suuntalippu täytyy olla +
; Output:  ES:[DI]   - purettu taulukko, CX kokonaislukua
;          ES:DI     - siirretty osoitin uutta kutsua varten
; Muuttuu: AX,BX,CX,DI,liput
; Käyttää: -
;
; VARO!!!  -  CX = 0 suorittaa 10000H kierrosta!
;
; Algoritmi:
;   0.  Poistetaan merkityksetön bitti vasemmasta reunasta.
;   1.  Kierretään vasemman reunan 3 bittiä oikeaan reunaan.
;   2.  Erotettaan 3 oikeanpuoleista bittiä.
```

```
;   3.  Talletetaan muodostunut kokonaisluku ja siirretään osoitinta.
;   4.  Tehdään kohdasta 1. yhteensä CX-kertaa.
;
; Rekistereiden käyttö:
;   AX      - talletettava kokonaisluku
;   BX      - luku, jossa otettavat bitit
;   CX      - kierroslaskuri
;   ES:DI   - osoitin talletuspaikkaan
;                                                                    kellok.
  MOV  BX,AX          ; Purettava talteen BX:ään                       2
  ROL  BX,1           ; Poistetaan merkityksetön bitti.                2
ota_ja_talleta:
  ROL  BX,1           ; Pyöritetään tutkittavat vasemman kautta        2
  ROL  BX,1           ; viimeisiksi oikeaan reunaan.                   2
  ROL  BX,1           ;                                                2
  MOV  AX,BX          ; Tutkittava sana AX:ään, jotta ylim. pois.      2
  AND  AX,00000111B   ; Vain kolme viimeistä bittiä merkitsee.         4
  STOSW               ; Talletetaan ja siirrytään seuraavaan.         11
  LOOP ota_ja_talleta ;                                              17/5
  RET                 ;                                               8
PaloitteleJaTallenna ENDP ;            Yht: 4 + (CX-1)*40 + 36 = CX*40
;-------------------------------------------------------------------------

CODE ENDS
  END                  ; muista <RET> End:n perään !!!!!!!!!
```

10.6.3 Pääohjelma

```
PROGRAM taulukonpurkaminen(input,output);
{ Tietokonejärjestelmät-kurssin esimerkkiohjelma Jari Syvänen  8.2.1989   }
{                                                                          }
{ Pääohjelma aliohjelman testaukseen. Aliohjelma purkaa pakatun            }
{ 3-bittisiä kokonaislukuja sisältävän taulukon.                           }
{ Aliohjelmalle välitetään integer taulukko, jonka alkiot sisältävät       }
{ heksaluvut 21B9, binäärisenä 0010 0001 1011 1001. Kun aliohjelma         }
{ jättää ylimmän bitin huomiotta pitäisi tulokseksi tulla numerosarjaa     }
{ 2,0,6,7,1,.... Ja alkioita niin monta kuin vakio pakattuja ilmoittaa,    }
{ loput taulukosta b ovat -1:iä.                                           }

CONST  pakatunpituus =  20;   { pakatun taulukon pituus sanoina            }
       puretunpituus = 100;   { puretun taulukon pituus                    }
       pakattuja     =  17;   { pakattujen alkioiden määrä                 }

TYPE   pakattu = array[ 1..pakatunpituus ] OF INTEGER;
       purettu = array[ 1..puretunpituus ] OF INTEGER;

VAR    a    : pakattu;
       b    : purettu;
       i    : integer;

{$L teht46} { Näin kun käytetään Turbo Pascal 4.0 tai 5.0 }
PROCEDURE purku(i:INTEGER; VAR a: pakattu; VAR b:purettu); EXTERNAL;

BEGIN { taulukonpurkaminen }
  WRITELN('Ohjelma purkaa pakatun 3-bittisiä kokonaislukuja sis. taulukon.');
  FOR i:=1 TO pakatunpituus DO a[i]:=$21B9;      { Alustetaan pakattu     }
  FOR i:=1 TO puretunpituus DO b[i]:=-1;         { Alustetaan purettu     }
  purku(pakattuja,a,b);                          { Puretaan a b:hen       }
  FOR i:=1 TO puretunpituus DO WRITE(b[i]:4);    { Tulostetaan b          }
  WRITELN;
END. { taulukonpurkaminen }
```

Purku-aliohjelma ilman makroja:

Tehtävä 10.1 Nimet parametreille

Kirjoita PURKU-aliohjelma myös kohdassa 7.1.12 esitetyllä tavalla.

10.7 Harjoitustyö C-versio

10.7.1 Ohjelmalistaus COMPACT-muistimalli

```
;*************************************************************************
; Tietokonejärjestelmät -kurssin harjoitustyö      Jari Syvänen 8.2.1989
;
;-------------------------------------------------------------------------
; Tehtävä 46 : Tee assembler-aliohjelma, joka purkaa pakatun 3-bittisiä
;              kokonaislukuja sisältävän taulukon.
;-------------------------------------------------------------------------
;
; Mahdollinen käytännön sovellutus on näyttömuistin pixelin väri, jolloin
; saadaan kolmella bitillä hallittua 8 väriä.
; Toisaalta tehtävä voidaan tulkita myös kokonaisluvun muuttamiseksi
; oktaaliluvuksi.
;
; Ohjaajan kanssa sovittiin, että taulukko on pakattu seuraavalla tavalla:
; Esimerkiksi 17 pakattua lukua (2,0,6,7,1,4,7,5,5,2,5,2,6,3,5,1,2)
; neljässä sanassa (21B9,4F6A,559D,1500):
;    1  2  3  4  5    6  7  8  9 10   11 12 13 14 15   16 17
;
;   ┌──┬──┬──┬──┬──┐ ┌──┬──┬──┬──┬──┐ ┌──┬──┬──┬──┬──┐ ┌──┬──┐
;   │ 2│ 0│ 6│ 7│ 1│ │ 4│ 7│ 5│ 5│ 2│ │ 5│ 2│ 6│ 3│ 5│ │ 1│ 2│
;   └──┴──┴──┴──┴──┘ └──┴──┴──┴──┴──┘ └──┴──┴──┴──┴──┘ └──┴──┘
;  ┌────────────────┬────────────────┬────────────────┬────────────────┐
;  │0010000110111001│0100111101101010│0101010110011101│0001010100000000│
;  └────────────────┴────────────────┴────────────────┴────────────────┘
;
;  5432109876543210 5432109876543210 5432109876543210 5432109876543210
;       1. sana          2. sana          3. sana           4. sana
;       21B9             4F6A             559D              1500
;
; Sanan bitit 15 eivät ole pakatussa taulukossa mukana, joten ne
; jätetään huomiotta.
;
; Alkioiden järjestys määräytyy kuvan yläpuolella annetulla tavalla.
; Esimerkissä on 15 pakattua lukua 3 kokonaisessa sanassa ja
; neljännessä sanassa on 2 lukua.  (3 = 17 DIV 5 ja 2 = 17 MOD 5).
;
;-------------------------------------------------------------------------
;Määritykset
;
; Tämä versio on tehty COMPACT -muistimallin mukaan
.model compact
.code
```

```
;============================ purku ====================================
PUBLIC _purku
_purku PROC NEAR
;
; void purku(int n, int *a; int *b);
;
; Taulukossa a on pakattu taulukko 3 bittisiä kokonaislukuja.
; Taulukko on pakattu siten, että sanassa on viisi lukua ja
; sanan ylin bitti on merkityksetön, eli siis ei kuulu taulukkoon.
; Kokonaisluku n ilmoittaa pakattujen alkioiden määrän.
; Aliohjelma palauttaa puretun taulukon taulukossa b.
;
; Input:   a         - pakattu taulukko, (ylösp.pyör. n/5 kok.lukua)
;          n         - taulukon alkioiden lukumäärä
; Output:  b         - purettu taulukko (n kokonaislukua)
; Muuttuu: AX,BX,CX,DX,DI,SI,ES,liput
; Käyttää: PaloitteleJaTallenna -aliohjelmaa
;
; Algoritmi:
;   0.  Laitetaan osoitin pakattuun ja purettavaan taulukkoon.
;   1.  Lasketaan kokonaisten 5 lukua sisältävien lukumäärä (n DIV 5)
;       sekä viimeiseen sanaan jäävien lukumäärä (n MOD 5)
;   2.  Puretaan kokonaiset sanat.
;   3.  Puretaan viimeinen sana, jos siinä jotakin.
;   4.  Valmis
;
; Muuttujat:
;   DS:SI   - osoitin pakattuun taulukkoon
;   AX      - sana pakatusta taulukosta
;   ES:DI   - osoitin purettuun taulukkoon
;   CX      - sanasta talletettavien lukujen lukumäärä
;   DX      - kokonaisten sanojen lukumäärä
;   pinossa - viimeisessä sanassa olevien lukujen määrä
;
; Pinon sisältö kutsun MOV BP,SP jälkeen:
;  BP ->   old_BP
;  [BP+02]  paluu_os
;  [BP+04]  n (arvo)
;  [BP+06]  a:n osoite
;  [BP+0A]  b:n osoite
;---------------------------------------------------------------------------
PUSH BP                     ; Pinon alustus.
MOV  BP,SP
PUSH DS                     ; DS, DI ja SI täytyy säilyttää.
PUSH DI
PUSH SI
;---------------------- Osoittimet paikalleen: -----------------------------
 LDS  SI,[BP+06H]           ; DS:SI osoittaa a:n ensimmäiseen alkioon.
 LES  DI,[BP+0AH]           ; ES:DI osoittaa b:n ensimmäiseen alkioon.
;---------------------- Lasketaan kokonaisten lukumäärä: -------------------
 MOV  CX,[BP+04H]           ; Alkioiden lukumäärä CX:ään.
 JCXZ valmis                ; Ellei alkioita ole, niin ei tehdä mitään.
 XOR  DX,DX                 ; Nollataan DX jakolaskua varten.
 MOV  AX,CX                 ; Lasketaan CX/5:
 MOV  BX,5                  ; Jaetaan sanaan pakattujen alkioiden määrällä.
 DIV  BX                    ; DX on jakojäännös, eli viim. sanassa olev. lkm.
 PUSH DX                    ; Viim. sanassa olevien määrä talteen
 MOV  DX,AX                 ; Kokonaisten sanojen määrä laskuriin DX.
;---------------------- Puretaan kokonaiset sanat: -------------------------
 CLD                        ; Suunta +.
kokonaiset_sanat:           ;
 LODSW                      ; Siirretään purettava alkio DS:SIstä akkuun.
 MOV  CX,5                  ; Kerralla otettavien lukumäärä.
 CALL PaloitteleJaTallenna ; Puretaan sana kerrallaan.
 DEC  DX
 JNZ  kokonaiset_sanat      ; Kunnes DX on nolla.
;---------------------- Viimeinen sana: ------------------------------------
 POP  CX                    ; Viimeisessä olevien lukumäärä.
 JCXZ valmis                ; Jollei yhtään, niin lopetetaan.
 LODSW                      ; Otetaan viimeinen vajaa sana käsittelyyn.
 CALL PaloitteleJaTallenna ; Puretaan viimeisestä CX kappaletta lukuja.
;---------------------- Valmis: --------------------------------------------
valmis:                     ;
 POP  SI                    ; Palutetaan pinoon talletetut rekisterit.
 POP  DI
 POP  DS
 POP  BP                    ; Palautetaan pino.
 RET                        ; Palataan.
_purku ENDP
;---------------------------------------------------------------------------

;======================== Paloittele ja tallenna ===========================
PaloitteleJaTallenna PROC NEAR

... Loppu kuten Pascal-versiossa!
```

10.7.2 C-kielinen pääohjelma

Kun aliohjelman muistimalliksi on valittu COMPACT, pitää pääohjelma muistaa kääntää tällä muistimallilla!

```
/*************************************************************************/
/* Tietokonejärjestelmät-kurssin esimerkkiohjelma Jari Syvänen  8.2.1989   */
/*                                                                         */
/* Pääohjelma aliohjelman testaukseen. Aliohjelma purkaa pakatun           */
/* 3-bittisiä kokonaislukuja sisältävän taulukon.                          */
/* Aliohjelmalle välitetään integer taulukko, jonka alkiot sisältävät      */
/* heksaluvut 21B9, binäärisenä 0010 0001 1011 1001. Kun aliohjelma        */
/* jättää ylimmän bitin huomiotta pitäisi tulokseksi tulla numerosarjaa    */
/* 2,0,6,7,1,.... Ja alkioita niin monta kuin vakio pakattuja ilmoittaa    */
/* loput taulukosta b ovat -1:iä.                                          */

#include <stdio.h>

#define PAKATUNPITUUS    20    /* pakatun taulukon pituus sanoina          */
#define PURETUNPITUUS   100    /* puretun taulukon pituus                  */
#define PAKATTUJA        17    /* pakattujen alkioiden määrä               */

typedef int pakattu[PAKATUNPITUUS];
typedef int purettu[PURETUNPITUUS];

pakattu a;
purettu b;
int     i;

void purku(int i, int *a,int *b);

int main(void)
{
  printf("Ohjelma purkaa pakatun 3-bittisiä kokonaislukuja sis. taulukon.\n");

  for (i=0; i<PAKATUNPITUUS; i++) a[i]=0x21B9;   /* Alustetaan pakattu     */
  for (i=0; i<PURETUNPITUUS; i++) b[i]=-1;       /* Alustetaan purettu     */

  purku(PAKATTUJA,a,b);                          /* Puretaan a b:hen       */

  for (i=0; i<PURETUNPITUUS; i++) printf("%4d",b[i]); /* Tulostetaan b     */
  printf("\n");

  return 0;
}
```

Kirjallisuutta

8086/8088 mikroprosessorit,Karel Åkerlund, Kari Rinne - Mik-Net Oy, 1985
The 8086 book, Rector Russel, George Alexy - McGraw-Hill, 1980
Advanced Microprocessor Architectures, Luigi Ciminiera & Adriano Valenzano - Addison-Wesley, 1987
Asiantuntijan Yritysmikrot 1/89, DOS-moniajo ja UNIX, Johan Helsingius - 1989
Assembly Language Programming for the IBM PC, David J. Bradley - Prentice_Hall, 1984
C-Ohjelmointikieli, Jukka Korpela, Timo Larmela - Ota Data, 1987
Digital Computer Fundamentals, Thomas C. Bartee - McGraw-Hill, 1985
iAPX 86/88, 186/188 User's Manual - Intel, 1985
Microsoft Macro Assembler 5.0 Mixed Language Programming Guide - Microsoft, 1987
Technical Reference Manual - IBM, 1983
The Peter Norton Programmer's Guide to the IBM PC, Peter Norton - Microsoft Press, 1985
The Visible Computer: 8088, Assembly Language Teaching System IBM PC, Charles Anderson - Software Masters, 1985
Turbo Assembler Version 1.0 Reference Guide - Borland, 1988
Turbo Assembler Version 1.0 User's Guide - Borland, 1988
Turbo Pascal Version 5.0 Reference Guide - Borland, 1988
Turbo Pascal Version 5.0 User's Guide - Borland, 1988
Turbo Pascal Version 5.5 Object-Oriented Programming Guide - Borland, 1989

Hakemisto

E

F

G

H

I

J

Q

R

S

T

U

V

W

X

Y

Z ZF 31, 44

Zero Flag 31

Sisällys

Tehtävät