6.

Assemblermaskinen

Indhold:

6.1 Assembleringsprocesen
a) Uddrag: Assembleringen, Bjarne L. & Jon J.

6.2 8086 assemblerprogrammer
a) Note: 8086 assembler med debug, masm, link og codeview
b) Modeller for assemblerprogrammer

6.1.a

Datamat arkitektur,
Bjarne Larsen & Jon Jonsen
1991

Kapitel 10. Assembler.

I de foregående kapitler har vi flere gange vist eksempler på programmer i maskinkode. Programmerne i maskinkode har ofte været ledsaget af de samme programmer skrevet på en eller anden symbolsk form. Dette gjorde vi især, fordi sådanne symbolske programmer er betydelig lettere at læse, end de samme programmer skrevet i maskinkode.

Denne erkendelse kom man allerede til engang i midten af 1950'erne. Man fik ydermere den ide, at man kunne lave et program til computeren, som kunne oversætte et program fra en eller anden symbolsk notation til maskinkode.

Sådanne oversættere blev snart en realitet under navnet assemblere. Processen består i at omforme et kildeprogram (sourceprogrammet) skrevet i et såkaldt symbolsk maskinsprog eller assemblersprog til et objektprogram i maskinsprog. Det er stort set den samme proces, der anvendes ved mange højniveau sprog.

Det oversatte program er i maskinkode format, og er stort set klar til at køre på maskinen.

Fortolkede og oversatte programmer.

I de foregående kapitler har vi utallige gange brugt ordet program, uden at vi nærmere har defineret, hvad dette begreb dækker over. Et program er defineret som en sekvens af instruktioner, som er sammensat på en bestemt måde med henblik på løsning af en konkret opgave.

Der er stor forskel på de programmer, som vi hidtil har arbejdet med, og de programmer, som vi nu skal til at arbejde med. Forskellen ligger i, at mikroprogrammerne var fortolkede, hvorimod de programmer, som vi nu skal arbejde med, er oversatte.

Mikroprogrammet består ligesom et assemblerprogram af instruktioner; men mikroprogrammets enkelte delkoder føres typisk frem til dekodere og multiplexere, hvor den egentlige oversættelse eller fortolkning af delkoden udføres. Det vil sige, at hver enkelt instruktion i mikroprogrammet bliver fortolket pány, hver gang instruktionen udføres.

Det samme princip findes anvendt i forbindelse med visse højniveau sprog (f.eks. BASIC), som ligeledes kan være fortolkede. Sådanne sprogoversættere kaldes undertiden for fortolkere.

Assemblersprog og langt de fleste højniveau sprog er imidlertid oversatte, hvilket vil sige, at programmets kildetekst er oversat til objektkode (maskinkode) en gang for alle.

Efter oversættelsen har programmets kildetekst ikke nogen betydning i forbindelse med programmets udførelse, bortset fra at det kan være en hjælp ved fejlfinding i programmet. Hvis programmet skal rettes, sker det ved at ændre i kildeteksten, og så oversætte den til maskinkode igen.

Eksempel på assemblersprog.

I kapitel 8 definerede vi et maskinsprog, som vi brugte som basis for gennemgangen af mikroprogrammer. Dette maskinsprog var baseret på følgende maskininstruktioner:

0	NOP	No operation
1	LOAD	Hent data til akkumulator
2	ADD	Adder data til akkumulator
3	SUBTRACT	Subtraher data fra akkumulator
4	DIVIDE	Divider akkumulator med data
5	MULTIPLY	Multiplicer data med akkumulator
6	STORE	Gem akkumulators værdi i lageret
7	JUMP	Ubetinget hop
8	TESTZERO	Test om akkumulator er nul
9	TESTGREATER	Test om data er større end akkumulator
15	STOP	

Udover disse egentlige assembler instruktioner er der også et antal pseudo instruktioner, som ikke vil resultere i maskinkode; men som derimod er instruktioner til oversætter programmet - eventuelt med besked om, at der skal afsættes plads til data celler i lageret.

Med henblik på udformning af en fiktiv assembler til ovenstående maskinkode suppleres assembler instruktionerne med følgende pseudo instruktioner:

START	Programmets startadresse
DATA	Afsættelse af en celle med en data konstant
RESERVE	Afsættelse af et antal celler til data
END	Afslutning af programmet

I dette tilfælde har vi valgt uforkortede navne på de enkelte operationer; men i mange assemblersprog foretrækker man en kortere form, hvor man anvender en mnemoteknisk forkortelse for operationen. Det kan være "L" i stedet for "LOAD", "A" i stedet for "ADD" osv.

Opbygningen af en assembler instruktion.

Stort set alle assembler sprog til virkelige computere er opbygget efter samme princip. Hver linie består af fire felter: et label felt, et operationskode felt, et operand felt og et kommentar felt.

De enkelte felter på linien er adskilt fra hinanden med et eller flere blanktegn. Label feltet er valgfrit, og kan udelades. Hvis assembler instruktionen indeholder en label, skal denne starte i liniens position 1. Hvis en linie altså starter med et blanktegn, er det et tegn om, at der ikke er nogen label i denne linie.

En labels funktion er, at den skal agere hopadresse for eventuelle hopinstruktioner og variabelnavn for eventuelle dataceller.

Label	Operation	Operand	Kommentarer
BEGYND	START		
	LOAD	TAL1	Hent Tal1 til akk.
	ADD	TAL2	Adder Tal2 til akk.
	STORE	RESULTAT	Gem resultat i Resultat
	STOP		
TAL1	DATA	17	
TAL2	DATA	13	
RESULTAT	RESERVE	1	
	END	BEGYND	

Figur 1. Assemblerprogram til addition af to tal.

Assembleringen.

Selve den proces at oversætte kildeprogrammet i assembler sprog til et objekt program i form af maskinkode kaldes assemblering.

Oversættelsen sker i to delprocesser eller passager, hvilket er årsagen til, at sådanne oversættere kaldes two-pass assemblere. Årsagen til, at man er nødt til at lave to passager af programmet, er, at man ved første passage støder på variabelnavne og labels, som endnu ikke er defineret i label feltet. Det forekommer f.eks., hvor der er referencer til data celler, som først defineres senere i programmet, eller hvor der er hop til adresser, som først defineres senere i programmet.

Visse højniveau sprog (f.eks. Pascal og C) er principielt one-pass oversættere. I sådanne sprog er det et krav, at alle variabler samt procedure- og funktionsnavne er defineret, før de refereres i programmet.

Første passage (pass one).

Hovedformålet med første passage er at opbygge en såkaldt symbol tabel. Symbol tabellen er en fortegnelse over alle labels (hopadresser og variabelnavne) i programmet.

Hvis man ved gennemlæsning af programmet støder på en label, tilføjes navnet på denne label til symbol tabellen sammen med den adresse, som den pågældende label vil få i det færdige program.

Adressen holder assembleren hele tiden styr på ved hjælp af en tæller kaldet instruction location counter. Denne tæller tælles op, efterhånden som man behandler linier, som skal omsættes til maskinkode, eller hvor der afsættes plads til data celler.

I de følgende figurer 2 til 5 vises indholdet af symbol tabellen og location counteren på forskellige trin i assemblering.

```
BEGYND     START
           LOAD       TAL1      Hent Tal1 til akk.
           ADD        TAL2      Adder Tal2 til akk.
           STORE      RESULTAT  Gem resultat i Resultat
           STOP
TAL1       DATA       17
TAL2       DATA       13
RESULTAT   RESERVE    1
           END        BEGYND
```

Symbol	Adresse
BEGYND	0

Instruction Location Counter: 1

Figur 2. Symbol tabel efter behandling af 2 linier.

```
BEGYND     START
           LOAD       TAL1      Hent Tal1 til akk.
           ADD        TAL2      Adder Tal2 til akk.
           STORE      RESULTAT  Gem resultat i Resultat
           STOP
TAL1       DATA       17
TAL2       DATA       13
RESULTAT   RESERVE    1
           END        BEGYND
```

Symbol	Adresse
BEGYND	0

Instruction Location Counter: 3

Figur 3. Symbol tabel efter behandling af 4 linier.

```
BEGYND     START
           LOAD       TAL1      Hent Tal1 til akk.
           ADD        TAL2      Adder Tal2 til akk.
           STORE      RESULTAT  Gem resultat i Resultat
           STOP
TAL1       DATA       17
TAL2       DATA       13
RESULTAT   RESERVE    1
           END        BEGYND
```

Symbol	Adresse
BEGYND	0
TAL1	4

Instruction Location Counter

5

Figur 4. Symbol tabel efter behandling af 6 linier.

```
BEGYND     START
           LOAD       TAL1      Hent Tal1 til akk.
           ADD        TAL2      Adder Tal2 til akk.
           STORE      RESULTAT  Gem resultat i Resultat
           STOP
TAL1       DATA       17
TAL2       DATA       13
RESULTAT   RESERVE    1
           END        BEGYND
```

Symbol	Adresse
BEGYND	0
TAL1	4
TAL2	5
RESULTAT	6

Instruction Location Counter

7

Figur 5. Symbol tabel efter behandling af 9 linier.

Anden passage (pass two).

Opgaven i anden passage af assembler programmet er at generere den maskinkode, som udgør objekt programmet eller resultatet af assembleringsprocessen. Faktisk er objekt programmet sjældent det eneste resultat, idet der som regel også dannes en listning, som dokumenterer processen. Af denne listning vil også fremgå de fejl, som assembleren opdagede under assembleringen.

De fejl, der kan være tale om, er f.eks.:

Labelnavne, der er defineret flere gange (duplikater)

Henvisninger til labelnavne, som ikke er defineret

Angivelse af instruktionskoder, som ikke eksisterer

Under anden passage trækkes der oplysninger fra en operationskode tabel, hvor man på grundlag af assemblerinstruktionens mnemotekniske operationskode kan slå op og finde den tilsvarende talværdi.

Location counteren nulstilles, inden anden passage påbegyndes.

Af nedenstående figur 6 og 7 fremgår indholdet af location counter på forskellige stadier af anden passage. Man kan også se, hvordan opbygningen af objektprogrammet skrider frem.

```
BEGYND     START
           LOAD       TAL1      Hent Tal1 til akk.
           ADD        TAL2      Adder Tal2 til akk.
           STORE      RESULTAT  Gem resultat i Resultat
           STOP
TAL1       DATA       17
TAL2       DATA       13
RESULTAT   RESERVE    1
           END        BEGYND
```

Symbol	Adresse
BEGYND	0
TAL1	4
TAL2	5
RESULTAT	6

Instruction Location Counter

3

Objekt program i maskinkode (hexadecimal/binær)

Hex	Binær	
1004	0001 0000 0000 0100	LOAD
2005	0010 0000 0000 0101	ADD

Figur 6. Objekt program efter oversættelse af 3 linier.

```
BEGYND     START
           LOAD       TAL1      Hent Tal1 til akk.
           ADD        TAL2      Adder Tal2 til akk.
           STORE      RESULTAT  Gem resultat i Resultat
           STOP
TAL1       DATA       17
TAL2       DATA       13
RESULTAT   RESERVE    1
           END        BEGYND
```

Symbol	Adresse
BEGYND	0
TAL1	4
TAL2	5
RESULTAT	6

Instruction Location Counter

7

Objekt program i maskinkode (hexadecimal/binær):

Hex	Binær	
1003	0001 0000 0000 0011	LOAD
2004	0010 0000 0000 0100	ADD
6005	0110 0000 0000 0101	STORE
F000	1111 0000 0000 0000	STOP
0011	0000 0000 0001 0001	TAL1
000D	0000 0000 0000 1101	TAL2
0000	0000 0000 0000 0000	RESULTAT

Figur 6. Objekt program efter oversættelse af 9 linier.

Principopbygning for assembler.

```
Første passage:

Klargør(Source input fil)
ILC:= 0    (* Instruction Location Counter *)
Sålænge der er sourcelinier i input fil udfør:
  Læs næste sourcelinie fra source input fil
  Er der et symbol (label) i sourcelinien?
    ja:
    Indsæt symbol i symboltabel
  Er der en instruktion i sourcelinien?
    ja:
    ILC:= ILC + 1
  slut-udfør
Luk(Source input fil)


Anden passage:

Klargør(Source input fil)
Klargør(Object output fil)
ILC:= 0
Sålænge der er sourcelinier i input fil udfør:
  Læs næste sourcelinie fra source input fil
  Er der en instruktion i sourcelinien?
    ja:
    CASE instruktionstype
    instruktion uden datareference:
      Dan objectlinie af instruktionskode
      Skriv objectlinie på objectfil
      ILC:= ILC + 1
    instruktion med datareference:
      Find adresse i symboltabel
      Dan objectlinie
      Skriv objectlinie på fil
      ILC:= ILC + 1
    instruktion med datadefinition:
      Dan objectlinie med dataindhold
      Udskriv objectlinie på objectfil
      ILC:= ILC + 1
    instruktion med reservation af celler:
      Udfør i:= 1 til dataindhold
        Dan objectlinie med indhold 0
        Skriv objectlinie på objectfil
        ILC:= ILC + 1
      Slut-udfør
    Slut-CASE
Slut-sålænge
Luk(Object output fil)
Luk(Source input fil)
```

Figur 7. Pseudokode visende principopbygningen af en assembler oversætter.

Resume af kapitel 10, assembler.

I dette kapitel blev en særlig gruppe programmer, de såkaldte assemblere, behandlet. En assembler er et program, som kan oversætte et program skrevet i et symbolsk og dermed lettere forståeligt sprog til maskinkode, som udelukkende består af talkoder.

Der skelnes mellem to forskellige principper, hvoraf det ene kaldes fortolkning og det andet oversættelse. Man kalder det fortolkning, når hver enkelt linie oversættes og udføres, hver gang en instruktion i programmet skal behandles. Når man anvender oversættelse, oversættes hele programmet til maskinkode en gang for alle, inden programmet udføres. I så fald omtales programmet før oversættelsen som kildeprogrammet, hvorimod det efter oversættelse kaldes objektprogrammet.

Et assemblersprog består af et antal instruktioner, som kan oversættes direkte til maskininstruktioner. Herudover findes et antal pseudoinstruktioner, som er instruktioner til assembleren (oversætteren).

En assemblerinstruktion består af fire dele, som er adskilt af mindst et blanktegn: label feltet, operationskode feltet, operand feltet og kommentar feltet.

Selve assembleringen (oversættelsesprocessen) deles i to faser eller passager, hvor man i første passage opbygger en såkaldt symbol tabel, og i anden passage opbygger selve maskinkoden.

6.2.a

8086 assembler med debug, masm, link og codeview

Introduktion incl. øvelser

Bjørk Busch

Denne note har til hensigt at indføre i 8086 assembler, afviklet på PC ved brug af DEBUG, MASM, LINK og CODEVIEW.

Der startes med at introducere assembler direkte fra DEBUG gennem 3 konkrete øvelser.

Når de første 3 øvelser er overstået gås der over til at bruge en egentlig oversætter, der også giver mulighed for at bruge symbolske navne (MASM).

Der vil her være en kort indføring i brug af MASM, LINK og CODEVIEW, idet der også her vil være 2 konkrete øvelser.

Man kan afprøve øvelserne på maskinen parallelt med, at man læser denne note.

Vi starter med DEBUG.

DEBUG er et af de programmer, der følger med DOS'en og giver mulighed for at rette direkte i det interne lager, afvikling af programmer i maskinkode instruktion for instruktion, indlæse og udskrive direkte på disk/diskette såvel på filniveau som på sectorniveau.

Øvelserne med DEBUG tager udgangspunkt i formatet for **COM-programfiler**. I dette format er **CODE**-, **DATA**- og **STAK**-segmentet pladseret samme sted i lageret (**ET**-segment på **64KB**).

I COM-programfil formatet **reserver** DOS de første **256 bytes** (100 hex) af segmentet til noget, der kaldes et **PSP** (program-segment-prefix).

Den **første** ledige celle til programmet bliver derfor **celle 256** (100 hex).

I COM-formatet forventes det endvidere at den **første instruktion** ligger på denne **celleadresse** (**100 hex**).

Øvelse 1.

Denne opgave tager udgangspunkt i en opgave, der tidligere er blevet afviklet på modelmaskinen.

Udtrykt i pascal-notation:

```
RES := TAL1 + TAL2 - TAL3;
```

I denne øvelse vælger vi at anvende en byte til hver variabel.

```
Valgte adresser:
TAL1: 110
TAL2: 111
TAL3: 112
RES:  113
```

Som regneregister vælges **AL**-registeret (8-bit).

Start DEBUG op fra DOS ved at skrive

DEBUG

Du vil nu få en ny prompt fra DEBUG i form af en bindestreg, der viser at DEBUG er klar til at modtage kommandoer.

Vi er nu klar til at indtaste vores program, idet vi ønsker at gøre dette med symbolske maskinkoder og overlade det til DEBUG at omsætte disse til den egentlige maskinkode. Dette sker med brugen af kommandoen ***A*** (assemble).

Tast

A 100

100 angiver første celleadresse der skal anvendes til programkoden.

DEBUG vil nu svare med følgende:

```
1922:0100
```

Hvor 1922 er segmentadressen for programmet, og tildeles af DOS udfra hvor der er ledig plads i lageret.

Indtast nu programmet linie for linie med retur mellem hver. Der afsluttes med retur uden andet, hvorefter den normale prompt (bindestreg) fra DEBUG kommer igen.

```
1922:0100        MOV AL,[110]
1922:0103        ADD AL,[111]
1922:0107        SUB AL,[112]
1922:010B        MOV [113],AL
1922:010E        RET
1922:010F
```

Programmet ligger nu i celle 0100 til 010E.

Vi skal nu have indlæst nogle testdata i vores variable. Dette kan ske med komandoen E (enter).
Indtast testdata (39, 26 og 28) med følgende kommandoer.

```
E 110 27
E 111 1A
E 112 1C
|   |   |
|   |   +---- celleindhold (hex)
|   +-------- celleadresse (hex)
+------------ kommando (enter data)
```

Vi vil nu prøve, at se hvad der ligger i lageret. Dette kan ske med kommandoen D (dump).
Tast:

```
D 100 11F
|   |   |
|   |   +---- sidste celleadresse (hex)
|   +-------- første celleadresse (hex)
+------------ kommando (enter data)
```

Dump sker pr. default med udgangspunkt i datasegmentet, men her skal det erindres at i det aktuelle tilfælde ligger alle segmenter samme sted.

Der kan eller tilføjes en absolut segmentadresse eks. 40:100 eller refereres til et segmentregister eks. CS:100

DEBUG svarer herefter med udskriften:

```
1922:0100  A0 10 01 02 06 11 01 2A-06 12 01 A2 13 01 C3 77   .......*.......w
1922:0110  27 1A 1C 06 78 03 A0 78-03 A2 77 03 34 00 11 19   '...x..x..w.4...
```

Hvor ***1922*** er den aktuelle segmentadresse.

Programmet ligger i adresse 1922:0100 til 1922:010E, men de er jo ikke ligefrem nemt at læse.

Vores variable ligger i adresse 1922:110, 1922:111 og 1922:112. Her kan vi genkende indholdet.

Adresse 1922:0113 er variablen til resultatet og indholdet er det, der tilfældigt lå her fra tidligere. De øvrige adresser hører ikke til vores program.

Kommandoen

D 110 113

Vil alene vise indholdet af vores variable.

Det er muligt at få DEBUG til at omsætte fra maskinkode til symbolske maskinkoder, dette kan ske med kommandoen ***U***.

Omsæt det indtastede med kommandoen:

```
U 100 10E
|   |   |
|   |   +---- sidste celleadresse (hex)
|   +-------- første celleadresse (hex)
+------------ kommando (enter data)
```

DEBUG viser nu følgende:

```
1922:0100   A01001      MOV   AL,[0110]
1922:0103   02061101    ADD   AL,[0111]
1922:0107   2A061201    SUB   AL,[0112]
1922:010B   A21301      MOV   [0113],AL
1922:010E   C3          RET
```

Adresse	***Maskinkode***	***Symbolske kode***
Segnent:	*hex som i*	
Offset	*dumpet*	

Hvor ***1922*** er den aktuelle segmentadresse.

Sammenlign med det "rå" dump.

Bemærk at der er forskel på hvor meget de enkelte maskininstruktioner fylder.

Det betyder også at programtælleren ikke tælles op med 1 altid ved udførslen, men derimod med længden af den iganværende instruktion.

Tager vi instruktionen MOV AL,[0110] er maskininstruktionen opbygget af følgende komponenter:

```
A0 1001
 |    +---  Adresse på 2 operant (bemærk adresse er
 |          "baglæns")
 +-------   Operationskode for MOV AL,celleadresse
```

Tager vi instruktionen ADD AL,[0111] er maskininstruktionen opbygget af følgende komponenter:

```
0206 1101
 |      +---      Adresse på 2 operant (bemærk adresse er
 |                "baglæns")
 +--------- Operationskode for ADD AL,celleadresse
```

Operationskoden kan her yderlig opsplittes i selve operationen ADD samt en adressebeskrivende del, der beskriver hvilket register der indgår m.m. Dette vil vi vende tilbage til senere.

Vi vil nu prøve at køre vores program, idet vi afvikler en instruktion af gangen.

Før vi starter vil vi lige se indholdet af registrene. Det kan ske med kommandoen R (register).
Tast

R

Herefter vises indhold af registrene.

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1922  ES=1922  SS=1922  CS=1922  IP=0100   NV UP EI PL NZ NA PO NC

1922:0100 A01001        MOV   AL,[0110]                        DS:0110=27
```

Bemærk af de 4 segmentregistre har samme indhold, samt at IP (programtælleren - instruction pointer) indeholder 100 hex.

Ud over reistrene vises den instruktion, der vil blive udført næste gang og for instruktioner der referer til lagerceller vises også aktuel indhold af disse lagerceller før udførslen.

Vi vil nu afvikle programmet med en instruktion af gangen.
Dette kan ske med kommandoen T (trace).

Tast

T

Herefter vises registrenes indhold efter udførslen af en maskininstruktion, herunder hvilken instruktion, der næste gang vil blive udført.

DEBUG viser

```
AX=0027  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=1922  ES=1922  SS=1922  CS=1922  IP=0103   NV UP EI PL NZ NA PO NC
1922:0103 02061101      ADD   AL,[0111]                        DS:0111=1A
```

Tast

T

DEBUG viser nu

```
AX=0041  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1922  ES=1922  SS=1922  CS=1922  IP=0107   NV UP EI PL NZ AC PE NC
1922:0107 2A061201      SUB   AL,[0112]                        DS:0112=1C
```

Tast

T

DEBUG viser nu

```
AX=0025  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1922  ES=1922  SS=1922  CS=1922  IP=010B   NV UP EI PL NZ AC PO NC
1922:010B A21301        MOV   [0113],AL                        DS:0113=06
```

Tast

T

DEBUG viser nu

```
AX=0025  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1922  ES=1922  SS=1922  CS=1922  IP=010E   NV UP EI PL NZ AC PO NC
1922:010E C3            RET
```

Nu skulle vores resultat være i adresse **0113** og vi kan nu se indeholdet med kommandoen ***D*** (dump).

Tast

D 110 113

DEBUG viser nu alle vores variable

```
1922:0110  27 1A 1C 25
                      +---- indhold i adresse 113
```

Vi kan nu lade programmet køre til ende og returnere, idet det ikke er interessant at følge med i programafviklingen, der så fortsætter i selve DOS/DEBUG programmet.

Vi kan lade programmet køre til ende med kommandoen ***G*** (go).

Tast

G

DEBUG vil så svare med følgende:

Program terminated normally

Vi kunne nu prøve vores program igen, men ved afslutningen vil segmentregistrene blive ændret og disse samt IP registret skal så først reetaberes.

Indtil nu har programmet **kun** ligget i den interne hukommelse og mistes når vi forlader debug eller slukker.
Det er dog muligt at gemme et lagerindhold på en disk/diskette og herfra senere hente det ind i lageret igen.
For at gemme lagerindhold i en fil skal DEBUG vide, hvad vi ønsker filen skal hedde. Det kan ske med kommandoen ***N*** (name).

Tast

```
N  DEBOPG1.COM
|       +------      Filnavn
+--------------      Kommando
```

Vi skal desuden fortælle hvor meget der skal gemmes (programmets størrelse).
Dette skal stå i CX registeret, der kan ændres med ***R*** kommandoen.

Tast

R CX

Herefter vises aktuelt indhold af CX-registeret som så kan ændres:

```
CX 0000
:
```

Herefter indtastes programstørrelse (her 14 hex bytes - fra 100 hex til og med sidste adresse i program 113).

Tast

14

Lagerindholdet kan nu gemmes med kommandoen ***W*** (write).

Tast

W

Debug svarer herefter

Writing 00014 bytes

Programmet er nu gemt.

Hvis man ønsker at køre programmet igen kan man indlæse det med kommandoen ***L*** (load). Herved vil segmentregistre og instruktionspointer samtidig blive reetableret.

Hvis man ønsker at indlæse en ny fil, skal man først angive navnet med ***N*** kommandoen. Det er dog også muligt at angive navnet på filen direkte ved opstart af DEBUG.

Eksempel på direkte opstart:

Tast

DEBUG DEBOPG1.COM

Øvelse 2.

Denne tager udganspunkt i samme program som den foregående, idet vi denne gang dog ønsker, at vores variable er på 16 bit istedet for 8.

Valgte adresser:

```
TAL1: 110
TAL2: 112
TAL3: 114
RES:  116
```

Som regneregister vælges **AX**-registeret (16-bit).

Opstart DEBUG og indtast programmet

```
A 100
MOV AX,[110]
ADD AX,[112]
SUB AX,[114]
MOV [116],AX
RET
```

Vi skal nu have indlæst nogle testdata i vores variable.
Indtast testdata (39, 26 og 28) med følgende kommandoer.

Indlæggelse:

```
E 110 27 00
E 112 1A 00
E 114 1C 00
|   |   |   |
|   |   |   +- celleindhold (hex) mest betydende byte
|   |   +----- celleindhold (hex) mindst betydende byte
|   +--------- celleadresse (hex)
+------------- kommando (enter data)
```

Vi erindre at tal ligger "baglæns" i lageret.

Variablerne kan ses ved udskrift af lageret med kommandoen D.

Tast

D110 116

Variablerne vises så som:

```
1922:0110  27 00 1A 00 1C 00 A0 78-
```

Vi kunne også have oprette data med kommandoen A.

Tast

```
A  110
DW 0027
DW 001A
DW 001C
```

Tast retur uden kommando for at afslutte.

På denne måde kan data oprettes **uden** vi selv skal ombytte, det foretages så automatisk.

Kontroler ved at dumpe igen.

Tast

D110 116

Sammenlign med foregående dump.

Afprøv programmet instruktion for instruktion ligesom i foregående øvelse.

Øvelse 3.

Denne opgave tager ligesom de foregående udgangspunkt i en opgave, der tidligere er blevet afviklet på modelmaskinen.

Udtrykt i pascal-notation:

```
RES := TAL1 * TAL2;
```

Programmet skal løse opgaven uden brug af multiplikations instruktion, men ved brug af addition i en løkke.

```
Valgte adresser:
TAL1: 120
TAL2: 122
RES:  124
```

Som regneregister vælges AX-registeret (16-bit).
TAL2 holdes i BX for at øge hastighed og som tælleregister anvendes CX.

Indtast programmet i DEBUG. (kommentar indtastes ikke)

```
adr.    A 100
100     MOV AX,0
103     MOV BX,[120]
107     MOV CX,[122]
10B     SUB CX,1          Cx:=Cx-1 , sæt flag
10E     JL  114           Hop hvis sidste beregning < 0
110     ADD AX,BX
112     JMP 10B
114     MOV [124],AX
117     RET
```

Det er her nødvendig at kende længden af de enkelte instruktioner der ligger efter JL for at få adressen, der skal hoppes til. Dette kan ske ved først at indsætte en foreløbig værdi og så rette den når den rigtige adresse kendes.
Rettelse kan ske med følgende kommandoer:

```
A 10E          (assembler adr. 10E)
JL  114        (kommando)
```

Afluttes med **retur** uden kommando.

Indlæg testværdier i Tal1 (120-121) og Tal2 (122-123) med E-kommandoen, idet du skal huske, at de 2 byte i hvert tal ligger "ombyttet".
Du skal nok vælge nogle små værdier for Tal2, så der ikke bliver for mange gennemløb.
Afprøv nu program trin for trin.

Her stopper vi med brugen af DEBUG til udvikling af assemblerprogrammer, idet vi vil gå over til at bruge en "rigtig" assembler, så vi kan anvende symbolsk kode.

Vi vil samtidig forlade DEBUG og bruge en bedre debugger der giver flere faciliteter.

Det efterfølgende vil give en indføring i brugen af MASM, LINK og CODEVIEW.

Introduktion til brug af MASM:

Kildeteksten til assemblerprogrammer kan skrives med en hvilken som helst ASCII editor, eks. EDLIN, EDIT, TURBO pacal editoren eller sågar WP (hvis man gemmer som DOS-tekst, funktion CTRL+F5)

Assemblerdiriktiver:

Assemblerdiriktiver er kommandoer til oversætteren, der ikke resulterer i maskinkode men bruges til styring af oversættelsen. I det efterfølgende gennemgås en række centrale assemblerdiriktiver.

PAGE 66,120

Dette angiver formatet på den liste, der kan dannes i forbindelse med oversættelsen. Der angives antal udskriftslinier pr. side og antal tegn pr. linie.

TITLE xxxxxxxxxxxxxxxxxxxxx

Angiver overskrift for den liste, der kan dannes i forbindelse med oversættelsen.

;

Fra semikolon og til linien slutter er kommentar

```
xxxx    Segment
        ......  (segmentets kode pladseres her)
xxxx    EndS
```

Angiver henholdsvis start og slut på et segment med navnet ***xxxx***.

Et **EXE**-program kan bestå af mange segmenter, idet det som regel består af mindst **3** (***CODE***, ***DATA*** og ***STACK***).

Et segment kan være på **max. 64 K byte**.

Der kan med de 4 segmentregistre (CS, DS, SS og ES) arbejdes "samtidig" i 4 segmenter.

Segmentdefinitionerne fylder ikke noget når programmet er indlæst til kørsel i lageret, men der gemmes informationer til linker og til DOS'es loader, der indlæser programmet i lageret og starter det.

xxxx Segment Para Public 'nnnn'

Udvidet beskrivelse af segment. Dette er kun nødvendigt når et program er opsplittet i flere selvstændige dele, der oversættes hver for sig.

Para angiver at segmentet skal starte på en adresse, der ender på 0 hex (paragraf-grænse).

Public 'nnnn' gør det muligt for linkeren, at samle segmenter med samme navn (nnnn - f.eks = CODE) i eet.

Sseg Segment Stack 'STACK'

Stack fortæller, at det er et stack-segment.

Sseg er det navn vi har valgt at give vores stack-segment.

Der skal afsættes plads til een stack, da operativ-systemet bruger denne.

Stack'en anvendes også ved kald af underrutiner og som "gemmeareal" i forbindelse med PUSH og POP instruktionerne.

```
pppp    PROC    tttt
        ................   (procedurens kode)
pppp    EndP
```

Angiver henholdsvis start og slut på en procedure med navnet ***pppp***.

Typen på procedure angives som ***tttt*** og kan være NEAR eller FAR.

Hvis der ikke angives en type antages NEAR.

Procedurenavnet kan anvendes som en almindelig label, men da der knyttes en type til, således at referencer til en FAR procedure giver fuld 32-bits adresse (segment+offset), hvorimod en NEAR kun giver et 16-bits offset (inden for aktuel code-segment).

Når der returneres fra en procedure med RET-instruktionen vil oversætteren vælge henholdsvis en 16-bits og en 32-bits adresse returnering, svarende til procedurens type.

Brug aldrig jump-instruktioner over til en anden procedure, da denne kan være en anden type og der så kan returneres forkert.

Brugen af procedurer gør selvsagt programmer nemmere af læse, så det er også derfor en god ide at bruges disse.

Assume Cs:Cseg,Ds:Dseg,Es:nothing,Ss:Sseg

Denne kommando fortæller oversætteren, hvilket segmentregister, der skal anvendes ve referencer i det enkelte segment (her navngivet Cseg, Dseg og Sseg).

Man skal selv sikre, at registrene har den rigtige adresse som indhold, idet loaderen dog normalt opsætter segmentregistrene Cs og Ss for os.

Oversætteren udregner selv offset på symbolske navne ud fra segmenterne.

Der kan udstedes flere assume-komandoer, idet der hermed skiftes offset beregning og dertil knyttet segmentregister efter hver ny assume.

zzzzz:

Her angives en symbolsk adresse (label), der kan anvendes f.eks i forbindelse med jump-instruktionerne eller END statmentet.

aaa EQU bbb

EQU kan anvendes til angive erstatningssymboler, således at et program bliver mere læsbart.
Oversætteren erstatter udtrykket bbb med aaa alle steder i programmet i forbindelse med oversættelsen.
En EQU-sætning "*fylder*" ikke selv noget i der oversatte program.

Eksempel på brug af EQU:
Der ønskes symbolnavn for carrige-return (ascii-tegn 13).

```
CR      EQU     13

        MOV     Ah,CR
```

Bliver under oversættelsen omskrevet til:

```
        MOV     Ah,13
```

End

Denne kommando fortæller oversætteren at der ikke er mere, der skal oversættes. Står der noget efter END vil det ikke blive oversat.

BEMÆRK !!!!!! der kan for noge oversættere (MASM) være problemer, hvis END står i den sidste linie og denne ikke er afsluttet med et linieskift.

End zzzzz

zzzzz er en label eller procedurenavn, der angiver hvilken instruktion (adresse) programmet skal starte i (hvad programtælleren skal sættes til ved start).
Dette er ikke nødvendigvis første instruktion i programmet.

Oversættelse af assemblerprogram til maskinkode:

Når programmet er skrevet med en editor skal det oversættes til maskinkode. Dette sker ved først at assemblere til objectkode og herefter linke til et eksekverbart modul.

Assembleringen kan foretages med programmet MASM og linkningen med programmet LINK.

Oversigt over de direkte anvendet programmer i forbindelse med programmering i PC-assembler:

1.	MASM	Oversætter til objektkode
2.	LINK	Binder / (linker) objektkode sammen til færdigt program.
3.	DOS-loader	Indlæser det færdige program i lageret og overgiver kontrollen til det. DOS-loader'en er en integreret del af DOS-kernen og bliver ikke beskrevet her.
4.	DEBUG	Linieorinteret debugger, som følger med DOS. Indgik i tidligere øvelser og beskrives ikke her.
5.	CODEVIEW	Fuld-skærms debugger, der følger med Micro Soft's macro-assembler fra version 5.0

Assemblering:

Først skal programmet oversættes til objectkode med programmet MASM.

Dette gøres fra DOS med følgende kommando:

MASM asmprog

Hvor asmprog er navnet på filen med kildeteksten (.ASM behøves ikke).

Man bliver nu "promptet" for navn på OBJ-fil, LST-fil og CRF-fil.

Hvis man blot taster retur oprettes kun OBJ fil idet NUL som filnavn betyder, at filen ikke laves.

Hvis man ønsker alle filerne dannet kan man skrive:

MASM asmprog,,,,

Man bliver så ikke "prompt'et" og alle filer oprettes med samme "fornavn".

Hvis der er syntaksfejl kan med fordel udskrive listen for det oversatte program, idet denne er påført fejl-meddelelserne.

Udskriften kan med fordel foretages via WordPerfect for at få sat liniebredde korrekt.

Start WP, Vælg 15/17 tegn pr. tomme (CTRL + F8 , grundskrift), indlæs .LST fil (CTRL + F5, Dos tekst, Hent) og udskriv (SHIFT + F7).

Linkning:

Når programmet er fri for syntaksfejl og oversat til objectkode skal det linkes med programmet LINK, for at blive til et program der kan køre.

Dette gøres fra DOS med følgende kommando:

LINK asmprog

Hvor *asmprog* er navnet på object-filen (.OBJ behøves ikke).

Man bliver nu "promptet" for navn på EXE-fil, samt navne på en række filer til udskrift m.m.

TIPS ! Skriv LINK og tryk F3, så kaldes resten af foregående (MASM) kommando frem og man slipper for skrivearbejde (risiko for skrivefejl).

Hvis alle filer ønskes, kan der ligesom for MASM anvendes en række kommaer efter asmprog. Der skrives altså:

LINK asmprog,,,

Man bliver så ikke "prompt'et" og alle filer oprettes med samme "fornavn".

Krydsreferencen, er **ikke** direkte klar til udskrift.
Hvis der ønskes en krydsreference-udskrift skal man først anvende programmet CREF.EXE, for at danne en printfil. Denne kan så udskrives.

Objekt-bibliotek med selvstændige oversatte underprogrammer:

Det er muligt at oversætte underprogrammer særskilt til objektkode og så medtage dem ved linkningen.

Det normale er i sådanne tilfælde at oprette et specielt bibliotek med underprogrammerne (objekt-modulerne). Når man så linker, kan man referere til biblioteket, hvorved linkeren selv medtager de moduler, der er reference til.

Objekt-biblioteker oprettes med LIB.EXE programmet.

Ved brug af selvstændige objekt-moduler, skal der defineres *eksterne* referencer, når der refereres til symboler i andre moduler, og i disse moduler skal disse referencer være defineret på en særlig måde.

For yderlig beskrivelse henvises til diverse assemblerbøger og manualer.

Programmet (EXE-fil) skulle nu være klar til afprøvning.

Programmet kan startes direkte fra DOS ved at skrive navn på EXE-fil.

Hvis programmet ikke udskriver noget på skærmen eller kører i en uendelig løkke (maskinen hænger) får man imidlertid ingen tegn fra programmet.
Vi må derfor benytte os af et program, der giver mulighed for, at se indhold i lager, registre m.m. Dette beskrives i det efterfølgende.

Afprøvning af programmer med DEBUG / CODEVIEW:

Sammen med DOS leveres et program med navnet DEBUG.EXE, der kan anvendes til at se indhold af lager, ændring af lagerindhold, indtastning af programmer direkte i assembler, omsætning fra maskinkode til symbolsk maskinkode (disassemblering/unassemble), afvikling af programmer i maskinkode instruktion for instruktion (single step) og meget mere.

Vi har imidlertid købt en egentlig oversætter MASM 5.1 og med den følger en langt bedre debugger med navnet CODEVIEW og det følgende vil beskrive brugen af dette program.
Kommandoerne fra DEBUG kan anvendes i CODEVIEW, så man kan derfor godt have interesse i at læse om DEBUG selvom man anvender CODEVIEW.
CODEVIEW giver mulighed for at arbejde med de symbolske navne fra assemblerprogrammet. Dette kræver dog nogle ekstra parametre til MASM og LINK, hvilket dog ikke vil blive beskrevet her.

Opstart af CODEVIEW sker fra DOS med følgende kommando:

CV prog

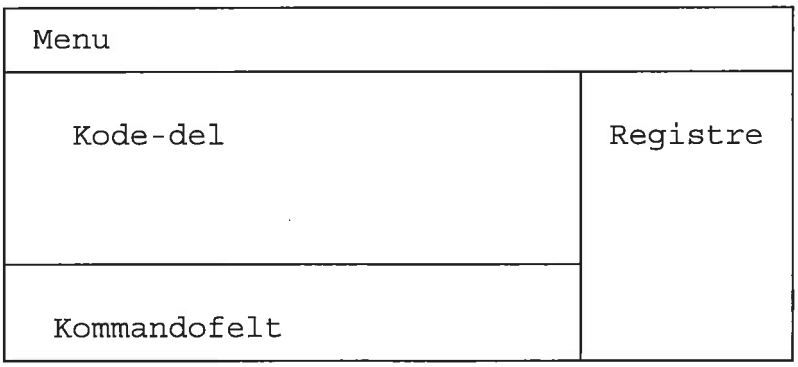
Prog er navnet på det færdige programmet (EXE-fil).

Der kan sættes nogle ekstra parametre på, så man f.eks kan få flere linier end 25 på en VGA skærm.

Man kan få en oversigt ved at skrive: CV /HELP

Parameteren /HELP kan også anvendes i forbindelse med MASM og LINK.

CODEVIEW har følgende opdeling af skærmen:



Menu'en aktiveres ved at taste ALT.

Register-delen vises ikke ved opstart, men kan fåes frem under menu-punkt view eller ved brug af F2.

Der kan skiftes til "bruger"-skærbilledet under menu-punkt view eller direkte med F4.

Der kan skiftes mellem kode-del og kommandofelt med F6.

Udførsel af program:

Når man ønsker at afvikle programmet kan det ske på flere måder:

Ønskes programmet kørt til ende uden stop, kan man vælge menu-punkt RUN og START eller taste **F5** for **GO**.

Ønsker man kun at afvikle en enkelt maskininstruktion af gangen (single step), kan man taste **F8** (**TRACE**). Det er så muligt at se / ændre registre, lager m.m. efter hver instruktion.

Ønsker man at afvilke programmet instruktion for instruktion, dog således at indtruktioner i underprogrammer ikke vises kan man istedet anvende **F10**.

Ønsker man at afvikle et bestemt antal instruktioner, kan man i komandofeltet skrive:

P n

hvor n er antallet af instruktioner før næste stop.

Skal programmet køres forfra vælges funktionen restart under menu-punkt RUN.

Display af memory:

Det er muligt udskrive indholdet af memory (lager-dump) ved at skrive følgende i kommandofeltet:

D seg:ofs

Seg er segmentadressen, enten angivet som en direkte adresse (hex) eller ved reference til segmentregister.
Ofs er offsetadresssen angivet hexadecimalt.
Eks:

D DS:0

Dumper starten af datasegmentet.
BEMÆRK at DS først skal sættes til vores datasegment før vi får det "rigtige" indhold. Det sker normalt i de 2 første instruktioner i vores program: (Mov Ax,Dseg og MOV Ds,Ax) Start derfor med 2 gange F8 for at få opsat datasegment.

Hvis man ønsker at se mere, behøver man blot at skrive:

D

idet der så fortsættes, hvor der sidst blev stoppet.

Dump af memory fortsat:

Det er muligt at få udskrevet memory-dump i flere formater med CODEVIEW.
Det gøres på følgende måde:

Df *seg:ofs*

Hvor ***f*** er det ønskede format og ***seg:ofs*** som før er adressen.
Der kan blandt andet angives følgende formater:

A Ascii
B Byte
I Integer med fortegn
W Word

Ændring af registerindhold:

Registrenes indhold kan ændres direkte ved at skrive følgende i kommandolinien:

R xx

Hvor xx er navnet på registeret. Herefter vises det gamle indhold og der kan skrives et nyt (hex).
Hvis indhold ikke ønskes ændret, tastes blot retur.

Flagene kan ændres ved at skrive følgende i kommandolinien:

RF

Herefter vises flagene og de kan ændres.

Ændring af memory:

Hvis man ønsker at ændre i memory direkte, kan dette gøres ved at skrive følgende i kommandolinien:

E ***seg:ofs***

Hvor ***seg:ofs*** er en adresse på samme måde, som for dump kommandoen.
Det gamle indhold vises og nyt indhold kan tildeles (hex).
Hvis man taster blank fortsættes med mulighed for at ændre næste lagercelle og hvis man ønsker at stoppe tastes retur.

Afslutning af CODEVIEW:

CODEVIEW kan afsluttes fra menu-punkt file, eller ved at skrive **Q** i kommandolinien.

Øvelser assemblerprogrammering med MASM og CODEVIEW.

De efterfølgende øvelser har til hensigt, at introducere hvorledes man opbygger 8086-assembler programmer med MASM samt afprøver disse med CODEVIEW.

Øvelse 4.

Denne opgave tager udgangspunkt i et program (DEBUG øvelse 2), der tidligere er blevet udviklet direkte fra DEBUG.

Udtrykt i pascal-notation:

```
RES := TAL1 + TAL2 - TAL3;
```

Løsning direkte fra DEBUG:

Valgte adresser til variable:

```
TAL1:     110
TAL2:     112
TAL3:     114
RES:      116
```

Tilhørende assemblerprogram:

```
MOV Ax,[110]
ADD Ax,[112]
SUB Ax,[114]
MOV [116],Ax
RET
```

Vi vil nu opbygge programmet i "rigtig" assembler, der senere kan oversættes til maskinkode med MASM.

I det efterfølgende skal programmet indtastes i en tekst-editor og gemmes med ASM som extention.

Programmet opbygges som et **EXE**-program og kommer til at bestå af **3 selvstændige segmenter:**

1) **Data** (variable)
2) **Code** (maskin-instruktioner)
3) **Stack** (anvendes bl.a. af systemet)

Det skal bemærkes at assembleren er en 2-pass assembler og segmenters rækkefølge let kan ombyttes.

Definition af data-segment (variable):

Vi definer først vores data i et segment vi her kalder **Dseg**.

```
Dseg   Segment          ; start på datasegment

Tal1 Dw   39            ; 39 angivet decimalt
Tal2 Dw   1Ah           ; 26 angivet hexadecimalt
Tal3 Dw   11100b        ; 28 angivet binært
Res  Dw   ?             ; plads afsat uden initiering

Dseg EndS               ; slut på data-segment
```

Vi har nu defineret vores data. Oversætteren vil sørge for at der afsættes plads samt give data det angivne startindhold.

Oversætteren vil klare udregningen af adresserne gennem opbygningen af en symboltabel.

Der vil udover adresserne også blive knyttet en datatype til symbolerne.

Datatypen anvendes dels af oversætteren til valg af den rigtige maskininstruktion, dels til kontrol af om maskininstruktion og datatype passer, idet oversætteren hermed kan give fejlmeddelelser.

Man kan selv påføre datatype i forbindelse med instruktionerne hvis man ønsker en anden type end den, der knyttes til vores data.

Definition af code-segment (instruktioner):

Vi vil nu opbygge selve programmet (instruktionerne).

I forhold til da programmet var et COM-program skrevet direkte i DEBUG vil der være 2 ænringer:

1) Vi skal selv sørge for, at datasegment-registeret (Ds) er opsat korrekt under selve programafviklingen (indeholder startadressen på datasegmentet.
Dette kan ske med følgende instruktioner:

```
Mov     Ax,Seg Dseg
Mov     Ds,Ax
```

Hvor Dseg er det navn vi aktuelt har givet vores datasegment.
Da der ikke findes en maskininstruktion, der kan flytte tal dirrekte til Ds-registeret, er det nødvendigt at flytte via. et andet register (her er anvendt Ax-registeret).

2) Ved afslutningen skal programmet returnere til DOS ved systemkald 21h, som er den korrekte fremgangmetode fra og med DOS version 3.0
Dette kan ske ved følgende instruktioner:

```
Mov     Al,0
Mov     Ah,4Ch
Int     21h
```

Hvor Al-registeret sættes til den ønskede returkode (kan aftestes i DOS som errorlevel), der her sættes til 0 (normal afslutning).
Ah-registeret sættes til 4Ch, som er koden for funktionen "afslutning af program" til DOS's funktionskald (INT 21h).
DOS-funktioner udføres normalt, ved at angive en kode i Ah-registeret, samt evt. yderligere informationer gennem andre registre afhængig af funktionen. Herefter "kaldes" DOS gennem interrupt 21h.

Den centrale del af programmet bliver følgende, idet der anvendes symbolske navne.

```
Mov Ax,Tal1
Add Ax,Tal2
Sub Ax,Tal3
Mov Res,Ax
```

Vi er nu klar til at skrive det samlede code-segment, idet det her er nødvendigt, at fortælle oversætteren hvilket segment-register, der skal anvendes i forbindelse med de enkelte segmenter og symbolske navne i disse.
Dette sker med compiler-direktivet "ASSUME".

Det samlede code-segment incl. "ASSUME" kommer hermed til at se således ud:

```
Cseg Segment                ; start på code-segment

     Assume Cs:Cseg,Ds:Dseg,Es:nothing,Ss:Sseg

Start:
     Mov  Ax,Seg Dseg       ; etabler
     Mov  Ds,Ax             ;     datasegment

     Mov  Ax,Tal1           ; Ax := Tal1
     Add  Ax,Tal2           ; Ax := Ax + Tal2
     Sub  Ax,Tal3           ; Ax := Ax - Tal3
     Mov  Res,Ax            ; Res := Ax

     MOV  Al,0              ; returkode = 0
     MOV  Ah,4Ch            ; opsæt funktion for
     INT  21h               ; terminate og udfør

Cseg EndS                   ; slut på code-segment
```

Bemærk lablen "***Start:***" der er adressen på den første instruktion, der skal udføres i vores program.
Vi har brug for denne label, idet vi i "***END***" sætningen skal angive hvad programtælleren skal sættes til når programmet indlæses af DOS for opstart.
Vi kunne have defineret programstumpen som en procedure, hvorved lablen "***Start:***" kunne undværes, da vi så kunne have anvendt procedurenavnet (der er startadressen på denne) istedet for.

Programmet mangler endnu et segment, nemlig et staksegment.
Alle normale programmer har brug for et sådanne også selvom de ikke selv gemmer data på stakken.
Stakken anvendes dels i forbindelse med kald til DOS'en.
Herudover vil ydre enheder som f.eks tastaturet kune afbryde det kørende program midlertidig og i den forbindelse bruges stakken også.

Staksegmentet defineres på følgende måde:

```
Sseg Segment Stack 'STACK'     ;start på stack-segment
     dw   128 dup(?)           ; der er sat 128 ord af
Sseg EndS                      ; slut på stack-segment
```

Vi mangler nu blot at fortælle oversætteret, at der ikke er mere der skal oversættes, samt hvad programtælleren skal sættes til når programmet skal køre.

Det samlede program kommer hermed til at se således ud, idet vi ønsker at påvirke programudskriftens udseende:

```
     PAGE 66,120            ; 66 linier med 120 tegn.
     TITLE  ET LILLE ASSEMBLER PROGRAM

Dseg   Segment              ; start på datasegment
;-----------------------------------------------------------
;    Datasegment
;-----------------------------------------------------------
Tal1 Dw   39                ; 39 angivet decimalt
Tal2 Dw   1Ah               ; 26 angivet hexadecimalt
Tal3 Dw   11100b            ; 28 angivet binært
Res  Dw   ?                 ; plads afsat uden initiering
;-----------------------------------------------------------
Dseg EndS                   ; slut på data-segment

Cseg Segment                ; start på code-segment
     Assume Cs:Cseg,Ds:Dseg,Es:nothing,Ss:Sseg
;-----------------------------------------------------------
;    Codesegment
;-----------------------------------------------------------
Start:
     Mov  Ax,Seg Dseg       ; etabler
     Mov  Ds,Ax             ;     datasegment
     Mov  Ax,Tal1           ; Ax := Tal1
     Add  Ax,Tal2           ; Ax := Ax + Tal2
     Sub  Ax,Tal3           ; Ax := Ax - Tal3
     Mov  Res,Ax            ; Res := Ax
     MOV  Al,0              ; returkode = 0
     MOV  Ah,4Ch            ; opsæt funktion for
     INT  21h               ; terminate og udfør
;-----------------------------------------------------------
Cseg EndS                   ; slut på code-segment

Sseg Segment Stack 'STACK'      ; start på stack-segment
;-----------------------------------------------------------
;    Stacksegment
;-----------------------------------------------------------
     dw   128 dup(?)        ; der er sat 128 ord af
;-----------------------------------------------------------
Sseg EndS                   ; slut på stack-segment
;-----------------------------------------------------------
     End  Start             ; Slut og opsæt startadresse
```

Indtast programmet og oversæt med programmet MASM.

I forbindelse med oversættelsen til OBJECT-kode kan der også dannes en en liste. Udskriv denne og sammelign med kildeteksten samt med programmet skrevet direkte i DEBUG.

Hvis der var syntaksfejl i programmet skal disse rettes og programmet oversættes igen med MASM.
Dette gentages til der ikke er flere syntaksfejl.

Programmet skal nu linkes med programmet LINK.

Programmet bliver nu til en EXE-fil, der kan eksekveres (udføres).
Da det aktuelle program imidlertid ikke har nogen kommunikation med skærm og tastatur, er det do ikke interessant at køre programmet direkte fra DOS.

Programmet skal nu prøves. Dette kan ske ved at afvikle det under DEBUG, som i de foregående øvelser.
Du skal imidlertid denne gang istedet prøve den mere avancerede debugger CODEVIEW (se udlevede note).
De anvendte kommandoer fra DEBUG kan også anvendes under CODVIEW.

Afprøv programmet under CODEVIEW.

CODEVIEW-menuen aktiveres ved at taste ALT.

CODEVIEW afsluttes med kommandoen QUIT.

Øvelse 5.

I DEBUG øvelse 3 udviklede vi et assemblerprogram der løste følgende udtrykt i pascal-notation:

```
RES := TAL1 * TAL2;
```

Omskriv assemblerprogrammet så der anvendes symbolske datanavne og kildetekst format svarende til programmet i fortegående øvelse.

Oversæt programmet og link det med henholdsvis MASM og LINK.

Afprøv herefter programmet under CODEVIEW.

6.2.b

Modeller for assemblerprogrammer

Bjørk Busch

Model med brug af fuld segmentdefinition

```
        PAGE 66,120
        TITLE AsmModel - skelet til assemblerprogram i 8086 asm
;-----------------------------------------------------------------
;       Programmør:_____________________________ Dato: __________
;-----------------------------------------------------------------

Cseg    Segment Para Public 'CODE'
        Assume Cs:Cseg,Ds:Dseg,Es:nothing,Ss:Sseg
;-----------------------------------------------------------------
;       Hovedrutine
;-----------------------------------------------------------------
Start:
        Mov     Ax,Seg Dseg             ; etabler
        Mov     Ds,Ax                   ;     datasegment

        ;.........................
        ;.........................
        ;.........................
        ;. programinstruktioner ..
        ;.........................
        ;.........................
        ;.........................


        Mov     Al,0                ; returkode = 0
        Mov     Ah,4CH              ; opsæt funktion for
        Int     21H                 ; terminate og udfør (MsDos)
;-----------------------------------------------------------------
Cseg    EndS                            ; slut på code-segment

Dseg    Segment Para Public 'DATA'
;-----------------------------------------------------------------
;       Data definitioner m.m.
;-----------------------------------------------------------------
        ;......................
        ;......................
        ;. Datadefinitioner ..
        ;......................
        ;......................
;-----------------------------------------------------------------
Dseg    EndS                            ; slut på data-segment

Sseg    Segment Para Stack 'STACK'
;-----------------------------------------------------------------
;       Stack
;-----------------------------------------------------------------
        dw      256 dup(?)          ; 256 ord sat af til stack
;-----------------------------------------------------------------
Sseg    EndS                        ; slut på stack-segment

        End     Start               ; Slut og opsæt startadr.
```

Model med brug af simplificeret segmentdefinition

```
;-----------------------------------------------------------------
;       Model for EXE-program med brug af simplificerede
;       segmentdefinitioner
;-----------------------------------------------------------------

        .MODEL  SMALL               ; max 64 KB

        .STACK  256                 ; definer stak

        .DATA                       ; definer datasegment

        ;......................
        ;. Datadefinitioner ..
        ;......................

        .CODE                        ; definer codesegment

MAIN    PROC    FAR                  ; main-procedure
        Mov     Ax,@DATA             ; etabler
        Mov     Ds,Ax                ;     datasegment

        ;.........................
        ;. programinstruktioner ..
        ;.........................

        Mov     Al,0                 ; returkode = 0
        Mov     Ah,4CH               ; opsæt funktion for
        Int     21H                  ; terminate og udfør (MsDos)
MAIN    ENDP                         ; slut på main-procedure

        End     MAIN                 ; kildetekst slut / startadr.
```

Model for COM-programmer

```
;-----------------------------------------------------------------
;       Model for COM-program
;       Der må kun være et segment, og ikke anvendes
;       FAR procedurer.
;       Ved opstart af COM-programmer vil CS=DS=SS=ES, man
;       kan derfor undlade at opsætte datasegmentet selv, men
;       hvis man kører EXE-programmet går det galt.
;-----------------------------------------------------------------
COMSEG  SEGMENT

        Assume Cs:COMSEG,Ds:COMSEG,Es:nothing,Ss:COMSEG

        ORG     100h                ; program start i
                                    ; offset 100h
                                    ; I adr. 0-100h
                                    ; pladserer DOS PSP
STARTADR:                           ; COM-programmer starter
        JMP     MAIN                ; automatisk i adr. 100h
                                    ; Hvis data ønskes i start
                                    ; må "man" jumpe udenom
;-----------------------------------------------------------------
;       Data definitioner m.m.
;-----------------------------------------------------------------
        ;......................
        ;. Datadefinitioner ..
        ;......................
;-----------------------------------------------------------------

;-----------------------------------------------------------------
;       Hovedrutine
;-----------------------------------------------------------------
MAIN    PROC    NEAR
        Mov     Ax,Seg COMSEG           ; etabler
        Mov     Ds,Ax                   ;     datasegment

        ;..........................
        ;. programinstruktioner ..
        ;..........................


        Mov     Al,0                ; returkode = 0
        Mov     Ah,4CH              ; opsæt funktion for
        Int     21H                 ; terminate og udfør (MsDos)
;-----------------------------------------------------------------
MAIN    ENDP                        ; slut på main-procedure

;-----------------------------------------------------------------
COMSEG  ENDS                        ; slut på segment

        END     STARTADR            ; Slut og opsæt startadr.
```

Bemærk: **COM-programmer dannes efter oversættelse og linkning med programmet EXE2BIN.**