TietgenSkolen
EDB-Skolen

Datamatiker uddannelsen

Noter til faget

Maskinarkitektur & Operativsystemer

Udgave 1.1

Samlet og redigeret af
Bjørk Busch

januar 1996

Forord.

Notesamlingen er opbygget af en række afsnit, der hver indeholder en eller flere noter vedrørende emnet.

Der er til hvert afsnit en detaljeret indholdsfortegnelse.

Der er ikke fortløbende sidenummerering, idet hver note betragtes som selvstændig.

Det er hensigten, at notesamlingen udbygges løbende, og at man desuden selv kan supplere med relevant materiale.

Indholdsfortegnelse:

1.

Operativsystemets brugerfaciliteter

Indhold:

1.1.a

Oversigt over DOS kommandoer

(De med * markerede kommandoer forventes at kunne anvendes middelbart af de studerende på Datamatiker uddannelsens første semester. De øvrige forventes kendt)

<Ansi.sys> *
<Append>
<Attrib> *
<Batch cmds> *
<Break>
<Buffers>
<Call> *
<Cd> *
<Chcp>
<Chdir>
<Chkdsk> *
<Chkstate.sys>
<Choice>
<Cls> *
<Command> *
<Config.sys cmds> *
<Copy> *
<Country> *
<Ctty>
<Date> *
<Dblspace>
<Dblspace Tips>
<Dblspace.sys>
<Debug>
<Defrag>
<Del> *
<Deltree>
<Device>
<Device drivers>
<Devicehigh>
<Dir> *
<Diskcomp>
<Diskcopy> *
<Display.sys> *
<Dos>
<Doskey>
<Dosshell>
<Driver.sys>
<Drivparm>
<Echo> *
<Edit>
<EGA.SYS>
<Emm386>
<Emm386.exe>
<Erase>
<Exit> *
<Expand>
<Fasthelp>
<Fastopen>
<Fc>
<Fcbs>
<Fdisk>
<Files>
<Find> *
<For> *
<Format> *
<Goto> *
<Graphics>
<Help>
<Himem.sys>
<If> *
<Include>
<Install>
<Interlnk>
<Intrelnk.exe>
<Internat. cmds> *
<Intersvr>
<Keyb> *
<Label> *
<Lastdrive>
<Lh>
<Loadfix>
<Loadhigh>
<Md> *
<Mem>
<Memmaker>
<MenuColor>
<MenuDefault>
<MenuItem>
<Mkdir>
<Mode Commands>
<More> *
<Move>
<Msav>
<Msbackup>
<Mscdex>
<Msd>
<Multi-config>
<Nlsfunc>
<Numlock>
<Path> *
<Pause> *
<Power>
<Power.exe>
<Print> *
<Prompt> *
<Qbasic>
<Ramdrive.sys>
<Rd> *
<Rem> *
<Ren> *
<Rename>
<Replace>
<Restore>
<Rmdir>
<ScanDisk>
<Set> *
<Setver>
<Setver.exe>
<Share>
<Shell> *
<Shift> *
<Sizer.exe>
<Smartdrv>
<Smartdrv.exe>
<Sort> *
<Stacks>
<Submenu>
<Subst>
<Switches>
<Sys>
<Time> *
<Tree> *
<Type> *
<Undelete>
<Unformat>
<Ver>
<Verify>
<Vol>
<VSafe>
<Xcopy> *

1.2.a

Dos batch programmering

Bjørk Busch

BAT-programmer er alm. tekstfiler (ascii) der bliver fortolket af kommandofortolkeren direkte. De oversættes således ikke først som f.eks. et turbopascal-program og adskiller sig herved fra programmer med extention COM og EXE. De startes dog fra DOS på samme måde, ved at angive navnet på filen (extention ikke nødv.).

BAT-programmer består primært af opstart af alm. programmer, der startes på samme måde som direkte i DOS-SHELLEN.

Fordelen er at man kan lave en sekvens af kommandoer på forhånd, hvilket er interessant ved opgaver der hyppigt gentages.

Der findes udover de alm. DOS-kommandoer nogle ekstra muligheder således at BAT-programmerne kan gøres fleksible. Der er således mulighed for at overføre informationer (argumenter/parameter) til BAT-programmet ligesom der findes enkelte muligheder for at afteste og styre udførslen af kommandoer.

Eksempel på et BAT-program uden parameter (DOKBKUP.BAT):

```
 1:  @ECHO OFF
 2:  REM  RUTINE TIL SIKKERHEDSKOPIERING AF DOKUMENTER
 3:  CLS
 4:  ECHO  INDSÆT DISKETTE TIL DOKUMENT BACKUP I DREV A:
 5:  ECHO  OG TRYK PÅ EN TAST   (CTRL+C for afbryd)
 6:  PAUSE > NUL
 7:  MD A:\DOKBKUP
 8:  COPY C:\DOK1\*.WP* A:\DOKBKUP\*.*  > A:\DOKBKUP.LOG
 9:  COPY C:\DOK2\*.WP* A:\DOKBKUP\*.*  >> A:\DOKBKUP.LOG
10:  ECHO  KOPIERING AFSLUTTET >> A:\DOKBKUP.LOG
11:  ECHO  KOPIERING AFSLUTTET
```

NB Linienumre medtages ikke i programfilen.

BAT-programmet startes ved at skrive:

DOKBKUP

Forklaring til rutinen:

1: normalt vil kommandolinier blive udskrevet før udførslen. Denne linie bevirker at DE EFTERFØLGENDE ikke vil blive udskrevet FØR udførslen.
For at denne kommando ikke selv udskrives sættes der et @ foran kommandoen.

2: Der kan (og bør) indlægges kommentar i BAT-programmer, disse indledes med REM og vil ikke blive udført.

3: Skærmen blankes

4+5: Udskrift af tekst på skærmen. Hvis der ikke var ECHO OFF ville først kommandoen blive udskrevet (ECHO + tekst) og herefter teksten.

6: BAT-program standses. Bemærk at den normale tekst på engelsk bliver undertrykt ved at redirigere den over i den logiske enhed NUL

7: Katalog oprettes på diskette drev A. Hvis det allerede findes vil der femkomme en meddelelse.

8: Filer kopieres og kopieringsmeddeleser omdirigeres til filen A:\DOKBKUP.LOG (hvis den fantes vil den blive overskrevet)

9: Der kopieres igen filer til disketten og kopieringsmeddelelser omdirigeres til LOG-filen, idet der skrives vidre på den eksisterende.

10: Afslutningsmeddelelse tilføjes til LOG-filen.

11: Afslutningsmeddelelse udskrives på skærmen.

Eksempel på BAT-program med parameter (DELBAK.BAT):

```
1:  DEL %1\*.BAK
```

Eksempler på brug:

```
a:  DELBAK
        DEL \*.BAK
        Sletter i root på aktuelt drev
b:  DELBAK .
        DEL .\*.BAK
        Sletter i aktuelt katalog på aktuelt drev
c:  DELBAK A:
        DEL A:\*.BAK
        Sletter i root på A:
d:  DELBAK PASCAL
        DEL PASCAL\*.BAK
        Sletter i PASCAL underkatalog på aktuelt drev
```

NB! Det skal bemærkes at %0 angiver navn på programfilen, der startes op.

Eksempel på BAT-program med brug af SHIFT kommandoen (DIR2.BAT):

```
1:  DIR %1
2:  DIR %2
3:  SHIFT
4:  DIR %1
5:  DIR %2
```

Eksempel på start:

VIS2 *.TXT *.DOK

bliver til:

```
1:  DIR *.TXT
2:  DIR *.DOK
4:  DIR *.DOK
5:  DIR
```

Når SHIFT anvendes rykkes parametre en plads, herved kan nye nåes men den foreste mistes.
Hvis man ønsker at gemme parametre kan SET kommandoen anvendes.

Eksempel på brug af IF og GOTO i batfiler (FMT.BAT):

```
 1:  @ECHO OFF
 2:  IF "%1"=="a:" GOTO OK
 3:  IF "%1"=="A:" GOTO OK
 4:  IF "%1"=="b:" GOTO OK
 5:  IF "%1"=="B:" GOTO OK
 6:  ECHO Der skal angives drevnavn A: eller B:
 7:  GOTO SLUT
 8:  :OK
 9:  C:\DOS\FORMAT.COM %1 %2 %3 %4 %5 %6 %7 %8 %9
10:  :SLUT
```

Eksempel på start:

FMT A:

Her vil A: blive formateret.

FMT eller
FMT C:

Her vil der komme en meddelelse (linie 6) og der vil ikke blive formateret.

Forklaring:

1-5: Her aftestes 1. parameteren. Da denne kan være "tom" (ingen parameter) er det nødvendig at tilføje 1 eller flere tegn så der ikke kommer til at stå IF == hvilket vil give syntaksfejl. Denne udvidelse med ekstrategn medtages så også i det der sammenlignes med.

9: FORMAT program startes, idet op til 9 parametre kan overføres (hvis flere ignoreres resten).

Eksempel på løkke i batfiler (VISFLERE.BAT):

```
 0:  @ECHO OFF
 1:  IF "%1"=="" GOTO VEJL
 2:  DEL DIRFIL$$.$$$
 3:  :IGEN
 4:    DIR %1 | FIND "." >> DIRFIL$$.$$$
 5:    SHIFT
 6:  IF NOT "%1"=="" GOTO IGEN
 7:  SORT < DIRFIL$$.$$$ | MORE
 8:  DEL DIRFIL$$.$$$
 9:  GOTO SLUT
10:  :VEJL
11:  ECHO Angiv mindst een parameter Eks:  %0 *.COM
12:  :SLUT
```

Eksempel på start:

VISFLERE *.COM *.EXE

Viser COM og EXE filer sorteret.

Eksempel på test om en fil findes (AKOPI.BAT):

```
1:  IF "%1"=="" GOTO SLUT
2:  IF EXIST %1 GOTO KOPIER
3:  ECHO %1 findes ikke
4:  GOTO SLUT
5:  :KOPIER
6:  COPY %1 A:\
7:  :SLUT
```

Eksempel på start:

AKOPI MINFIL.TXT

Forklaring:

Det er nødvendigt at 1. parametren ikke er tom i linie 2, da der ellers vil komme syntaksfejl.

Eksempel på test af returkode fra program:

Programmer har mulighed for at give en returkode tilbage til DOS, som så kan anvendes i BAT-programmering.

Der er enkelte DOS komandoer der udnytter denne mulighed men de fleste gør ikke.

```
1:  MITPROG
2:  IF ERRORLEVEL 4 GOTO LEV4
3:  IF ERRORLEVEL 1 GOTO LEV1
4:  :LEV0
.:  .......
.:  :LEV1
.:  .......
.:  :LEV4
```

Bemærk der også hoppes hvis ERRORLEVEL er større. Det er er derfor en fordel at teste i faldende orden.

Brug af systemvariable:

Det er muligt at gemme værdier i parameterblokken (environment).
Disse variable kan opsættes direkte fra DOS eller fra BAT-programmer og kan også bruges til at returnere værdier fra et BAT-program.
Der er 2 måder at anbringe værdier i parameterblokken. Den ene er med PATH kommandoen, hvor søgestien for programmer kan opsættes (tilsvarende med APPEND), den anden måde er med SET kommandoen. Ved opstarten indlægges desuden navnet på shellen.
Man kan få en oversigt over systemvariablerne med kommandoen SET uden parametre.
Systemvariable kan indeholde små bogstaver og specialtegn.

Eksempel på brug systemvarabel med SET:

a: SET DOSDIR=C:\DOS
Der oprettes en variabel med navnet DOSDIR og indholdet "C\DOS".
Denne linie kunne f.eks. indlægges i AUTOEXEC.BAT.

b: SET GEMP1=%1
Denne linie tænkes indlagt i et BAT-program, der startes med parametre.
1. parametren vil her blive gemt i variablen GEMP1 som så senere kan anvendes.

c: SET GEMP1=
Her slettes variablen GEMP1 igen. Det er vigtigt at der ikke er blanke efter lighedstegnet, da variable godt kan indeholde blanke. PAS PÅ!!!
Linien kunne ligge til slut i samme BAT-program som oprettede variablen (oprydning).

d: SET GEMPATH=%PATH%
Her oprettes en variabel med den aktuelle søgesti som indhold.

Eksempel på BAT-program med brug af systemvariable (DIR2KAT.BAT):

```
1:  SET P1=%1
2:  :IGEN
3:  SHIFT
4:  IF "%1"=="" GOTO SLUT
5:  DIR %P1%\%1
6:  GOTO IGEN
7:  :SLUT
8:  SET P1=
```

Eksempel på start:

DIR2KAT A: PASCAL WP51
Der bliver lavet 2 DIR kommandoer.
1. DIR A:\PASCAL
2. DIR A:\WP51

I det efterfølgende vises andre eksempler på BAT-programmer.

WP.BAT

```
 1:   REM Opstart til wordperfect med diverse parameter.
 2:   PUSHDIR                                shareware prog.
 3:   E:
 4:   CD E:\WP42
 5:   WP /D-D:\WPDOK /M-E:\WP42\STARTOP %1 %2 %3 %4 %5 %6 %7 %8 %9
 6:   CD E:\
 7:   CD D:\
 8:   POPDIR                                 shareware prog.
```

MTXBKUP.BAT

```
 1:   REM Sikkerheds kopiering af system.
 2:   ECHO OFF
 3:   CLS
 4:   ECHO Indsæt diskette til BACKUP af mattext-dokumenter i DREV A
 5:   ECHO Tast herefter RETURN  -  Der kan fortrydes med CTRL + C
 6:   PAUSE > NUL
 7:   MD A:\DOK1
 8:   MD A:\DOK2
 9:   COPY D:\MATTEXT\DOK1\*.MTX A:\DOK1\*.* /V
10:   COPY D:\MATTEXT\DOK2\*.MTX A:\DOK2\*.* /V
11:   ECHO BACKUP afsluttet
```

DIREXT.BAT

```
 1:   REM Filoversigt sorteret på EXTENTION
 2:   DIR %1 %2 | SORT /+10 | FIND "." | FIND " 0 " /V | MORE
```

CP.BAT

```
 1:   @ECHO OFF
 2:   REM Opsæt tegntabel for tastatur og skærm
 3:   IF NOT '%1'==''' GOTO SETCODE
 4:      ECHO Angiv nr. på tegntabel eks.    %0  865
 5:   GOTO SLUT
 6:   :SETCODE
 7:      KEYB DK,%1,C:\BIN\KEYBOARD.SYS
 8:      MODE CON CODEPAGE PREPARE=((%1)C:\BIN\EGA.CPI)
 9:      MODE CON CODEPAGE SELECT=%1
10:   :SLUT
```

SETSTI.BAT

```
 1:   REM Tilføj kataloger til default path
 2:   @ECHO OFF
 3:   PATH \;E:\BATFILER;E:\UTILITY;%DOS%
 4:   :LOOP
 5:   IF '%1'== '' GOTO SLUT
 6:      PATH %1;%PATH%
 7:      SHIFT
 8:   GOTO LOOP
 9:   :SLUT
10:   PATH
```

ASM.BAT

```
 1:   REM Rutine til assemblering og linkning - skrabet udgave
 2:   @ECHO OFF
 3:   D:\TASM\TASM %1,, /mu
 4:   IF ERRORLEVEL 1 GOTO SLUT
 5:      D:\TASM\TLINK %1
 6:   IF ERRORLEVEL 1 GOTO SLUT
 7:      ECHO Program %1 er assembleret og linket fejlfrit
 8:   :SLUT
```

ASM2.BAT

```
 1:   REM Rutine til assemblering og linkning - udbygget udgave
 2:   @ECHO OFF
 3:   IF '%1'=='' GOTO VEJL
 4:   IF EXIST %1.ASM GOTO TASM
 5:   ECHO %1.ASM findes ikke
 6:   :VEJL
 7:   ECHO Der skal angives filnavn uden .ASM
 8:   ECHO Eks.    %0  ASMPROG
 9:   GOTO SLUT
10:
11:   :TASM
12:   ECHO Assemblering af %1.ASM startet
13:   D:\TASM\TASM %1,, /mu
14:   IF ERRORLEVEL 1 GOTO ASMERR
15:   ECHO Link af %1.OBJ startet
16:   D:\TASM\TLINK %1
17:   IF ERRORLEVEL 1 GOTO LINKERR
18:   ECHO Program %1 er assembleret og linket fejlfrit
19:   GOTO SLUT
20:
21:   :ASMERR
22:   ECHO Fejl i assemblering af %1
23:   GOTO SLUT
24:   :LINKERR
25:   ECHO Fejl i linkning af %1
26:
27:   :SLUT
```

123.BAT

```
 1:   REM Opstart af lotus-123 i eget katalog uafhængig af drev
 2:   IF EXIST E:\123\123start.BAT GOTO EDREV
 3:   IF EXIST D:\123\123start.BAT GOTO DDREV
 4:   GOTO EJINST
 5:   :EDREV
 6:      E:
 7:      \123\123start.BAT %1 %2 %3 %4 %5 %6 %7 %8 %9
 8:   GOTO SLUT
 9:   :DDREV
10:      D:
11:      \123\123start.BAT %1 %2 %3 %4 %5 %6 %7 %8 %9
12:   GOTO SLUT
13:   :EJINST
14:      ECHO Lotus 123 er ikke instaleret
15:   :SLUT
```

123START.BAT

```
 1:   @ECHO OFF
 2:   IF "%1"=="?" GOTO VEJL
 3:
 4:   IF "%S123%"==""  SET S123=VGA        Sæt default første gang
 5:   IF NOT "%1"==""  SET S123=?               Hvis parameter
 6:   IF "%1"=="V"     SET S123=VGA
 7:   IF "%1"=="V25"   SET S123=VGA13225
 8:   IF "%1"=="V28"   SET S123=VGA13228
 9:   IF "%1"=="V44"   SET S123=VGA13244
10:   IF "%1"=="V60"   SET S123=VGA8060
11:
12:   IF NOT %S123%"=="?" GOTO START
13:
14:   :VEJL
15:   ECHO ---------------------------------
16:   ECHO Parametre for opstart af Lotus 123
17:   ECHO    V       VGA       80 x 25
18:   ECHO    V25     VGA      132 x 25
19:   ECHO    V28     VGA      132 x 28
20:   ECHO    V44     VGA      132 x 44
21:   ECHO    V60     VGA       80 x 60
22:   ECHO ---------------------------------
23:   GOTO SLUT
24:
25:   :START
26:   PUSHDIR
27:   CD \123                 aktuelt drev forudsættes
28:   123 %S123%              start med udvalgt driver %S123%
29:   CD \
30:   POPDIR
31:
32:   :SLUT
```

SPILINST.BAT

```
 1:   @ECHO OFF
 2:   IF '%1'==''  GOTO VEJL
 3:   IF '%2'==''  GOTO VEJL
 4:   IF EXIST %2.ARJ GOTO INSTARJ
 5:   IF EXIST %2.ZIP GOTO INSTZIP
 6:   IF EXIST %2.LZH GOTO INSTLZH
 7:
 8:   ECHO Fejl: filen %2 findes ikke
 9:   ECHO ----------------------------------------------
10:   FOR %%F IN (*.ARJ *.ZIP *.LZH) DO ECHO %%F
11:   GOTO SLUT
12:
13:   :VEJL
14:   ECHO Instalation af pakket spil.
15:   ECHO Der skal startes fra drev / katalog med pakket spil.
16:   ECHO Der skal være path til udpakke-programmet:
17:   ECHO     ARJ.EXE / PKUNZIP.EXE / LHARC.EXE
18:   ECHO ----------------------------------------------
19:   ECHO Instalationsrutinen opstartes med følgende parametre.
20:   ECHO    1. d1:\path (drev og katalog for spil)
21:   ECHO    2. spil (navn  spilfil uden  .ARJ / .ZIP / .LZH)
22:   ECHO eksempel
23:   ECHO    1.      %0  C:\SPIL   KEEN
24:   ECHO    2.      %0  D:\       KEEN
25:   ECHO ----------------------------------------------
26:   ECHO Tryk på en tast for oversigt over pakkede spil
27:   PAUSE > NUL
28:   FOR %%F IN (*.ARJ *.ZIP *.LZH) DO ECHO %%F
29:   GOTO SLUT
30:
31:   :INSTARJ
32:   @ECHO ON
33:   MD %1\%2
34:   ARJ x -r %2.ARJ %1\
35:   REM ARJ x -r %2.ARJ %1\%2
36:   GOTO SLUT
37:
38:   :INSTZIP
39:   @ECHO ON
30:   MD %1\%2
31:   PKUNZIP -o %2.ZIP %1\%2
32:   GOTO SLUT
33:
34:   :INSTLZH
35:   @ECHO ON
36:   MD %1\%2
37:   LHARC x -p %2.LZH %1\%2
38:   GOTO SLUT                          ikke nødvendig.
39:
30:   :SLUT
```

1.3.a

Eksempel på config.sys og autoexec.bat

Bjørk Busch

CONFIG.SYS

```
DEVICE=C:\XSYS\FASTBIOS.SYS
FILES=30
FCBS=1
BUFFERS=10
LASTDRIVE=E
DEVICE=C:\SYS\HIMEM.SYS
DEVICE=C:\SYS\EMM386.EXE noems
DOS=HIGH,UMB
DEVICEHIGH=C:\XSYS\EANSI.SYS
DEVICEHIGH=C:\XSYS\GMOUSE.SYS *4
DEVICEHIGH=C:\SYS\DISPLAY.SYS CON:=(EGA,865,1)
DEVICEHIGH=C:\SYS\SMARTDRV.SYS 1024
COUNTRY=045,865,C:\SYS\COUNTRY.SYS
SHELL=C:\SYS\COMMAND.COM C:\SYS /E:512 /P
```

AUTOEXEC.BAT

```
ECHO OFF
SET DOS=E:\MSDOS
PATH %DOS%
SET COMSPEC=%DOS%\COMMAND.COM
PROMPT $e[41;30m$P$G$e[43;30m        prompt i anden farve
LOADHIGH KEYB DK,865,C:\SYS\KEYBOARD.SYS
MODE CON CODEPAGE PREPARE=((865)C:\SYS\EGA.CPI)
MODE CON CODEPAGE SELECT=865
LOADHIGH FASTOPEN C:=10 D:=50 E:=50
CLS
DATE
TIME
D:
CALL E:\BATFILER\SETSTI.BAT                    se tidligere
ECHO Tast CTRL+C hvis virustest ikke skal køre
PAUSE > NUL
SCAN2 C:
PAUSE > NUL
SCAN2 E:
```

2.

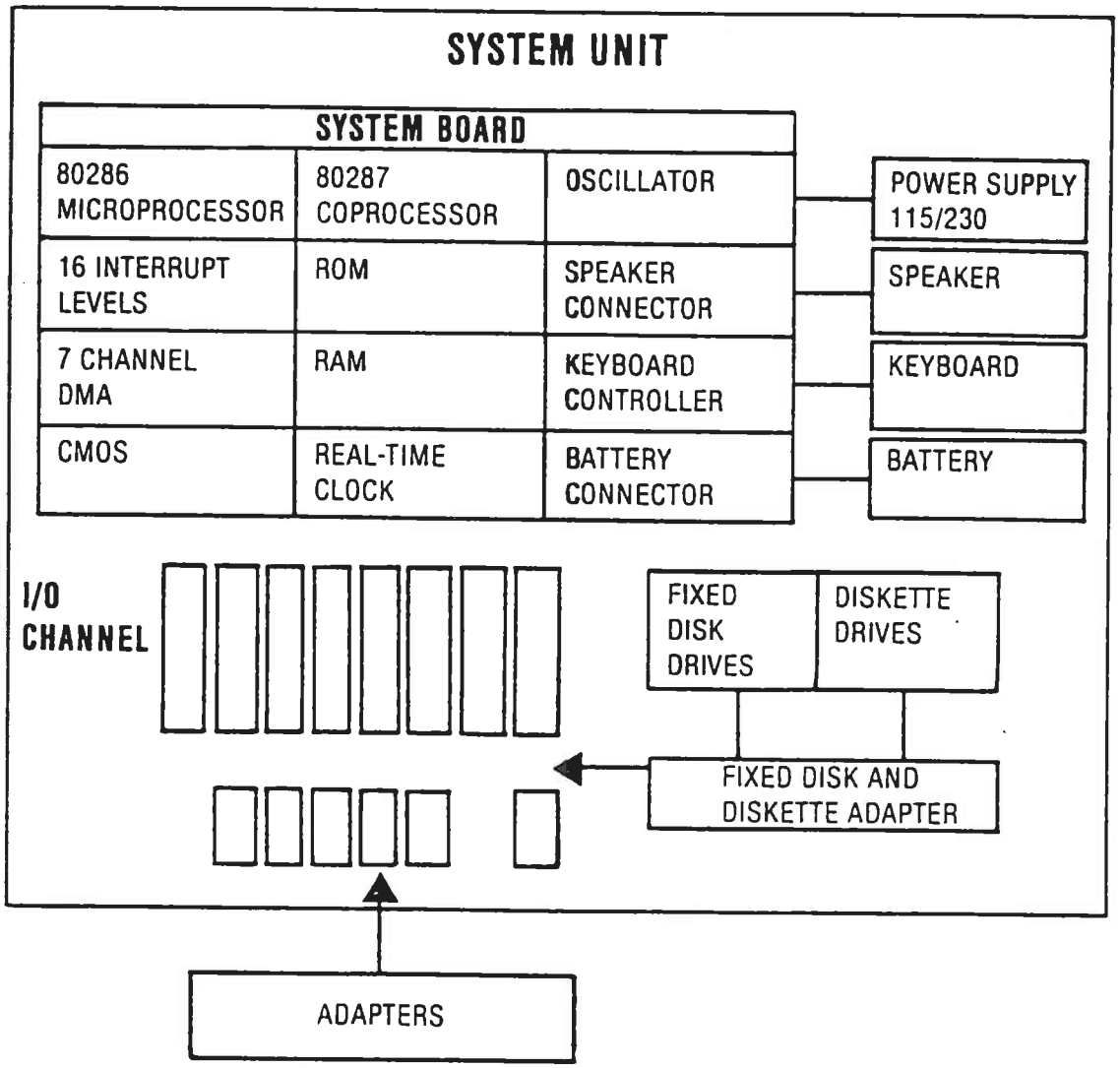
Maskinelle enheder

Indhold:

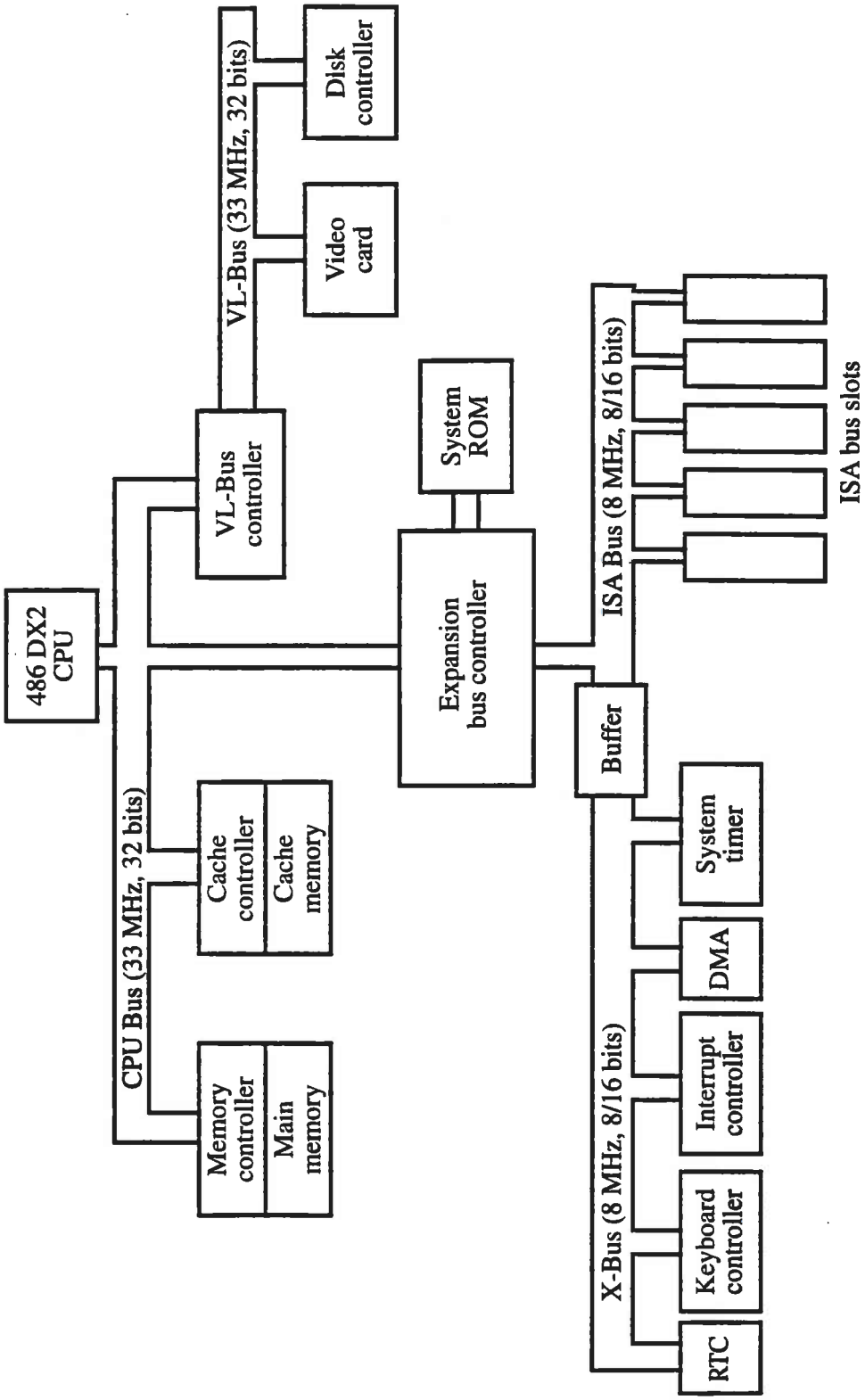
2.1.a

Systemenhedens komponenter

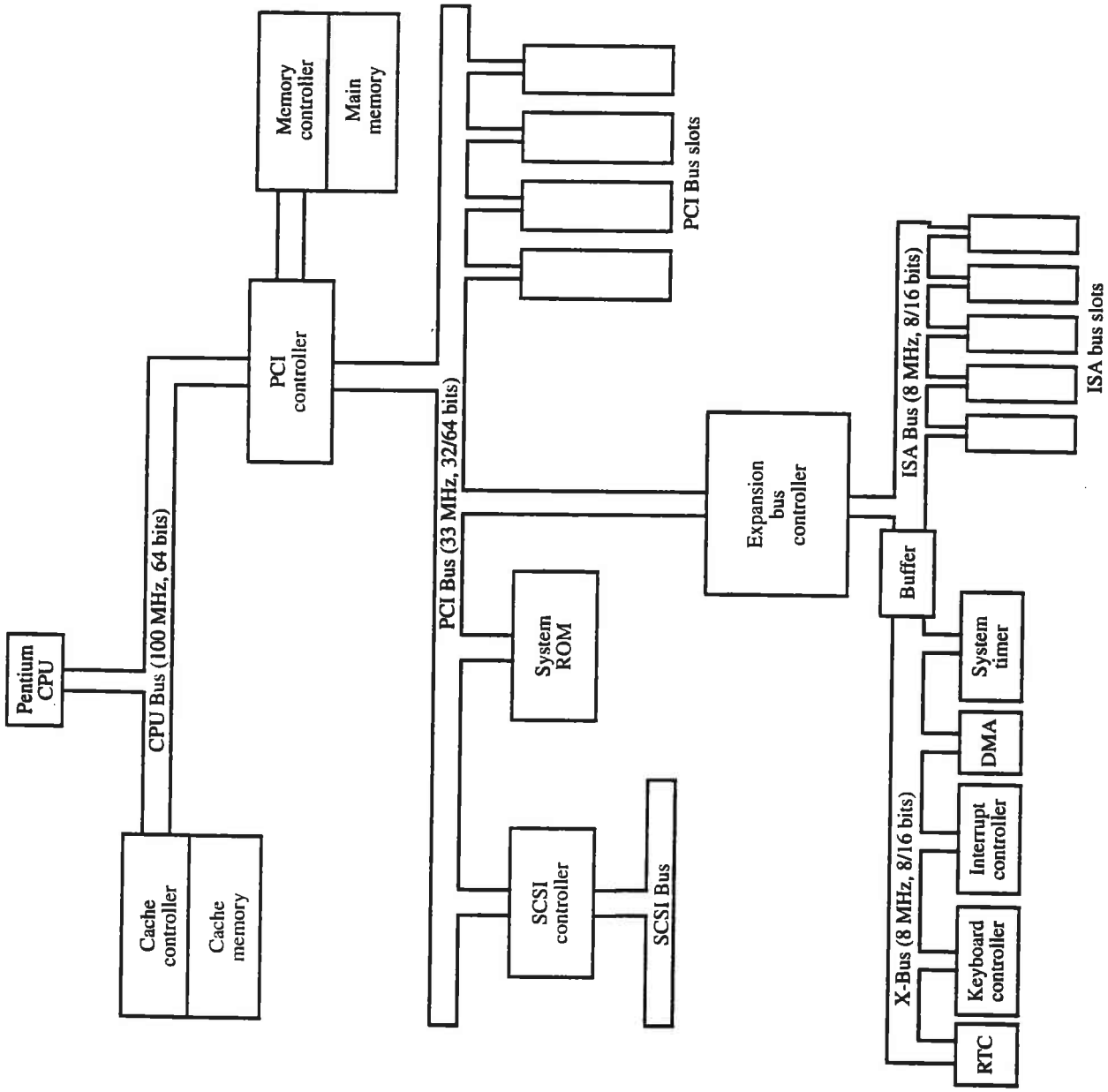
System Block Diagram



Block diagram showing the architecture of a basic ISA bus PC with a VL-Bus added to it



Block diagram showing the architecture of a PC that incorporates a PCI local bus.



Dataskærme

2.2.a

Af Kaj Riis

Den store vækst i salget af edb-løsninger til såvel privat som kommerciel anvendelse, har naturligvis også ført et stigende salg af ydre enheder til installationerne med sig. Da den egentlige kontakt med datamaten sker gennem blandt andet dataskærme, er det ikke underligt at der lægges større og større vægt på design og kvalitet af disse. På samme måde stilles stadig større krav til de andre ydre enheder, der knytter sig til enhver form for databehandling

Fra de første dataskærmes fremkomst i starten af 60'erne og indtil i dag, er der sket en kolossal udvikling. En udvikling, der primært har givet sig udslag i et bedre billede, både hvad angår farver, opløsning og stabilitet.

Prisudviklingen er gået som inden for al anden elektronik, nemlig nedad. Dog ikke helt i samme tempo – et forhold, der også ses på fjernsynspriserne, der i forhold til f.eks. lønudviklingen ikke er faldet ret meget de seneste 10 år.

Som man ofte sammenligner kr. pr. Byte for maskiner og diskettedrev, kr. pr. tegn/sek. ved printere, sammenligner man nu kr. pr. pixel ved farveskærme.

I denne konstruerede måleenhed ses udviklingen i pris og ydeevne allerbedst.

Og hvad er så en monitor?

Hovedparten af mikroer til privat benyttelse er tilsluttet husets fjernsyn. Mange mikroer er lavet specielt til dette, idet de højst arbejder med 40 tegn på hver linie. Og ca. 40 tegn er netop hvad et almindeligt fjernsyn kan gengive uden større problemer med at læse teksten. Samtidig er fjernsynets opløsning tilstrækkelig til den grafik, der er indbygget i de fleste hjemmecomputere.

Problemerne dukker op, når der skal tilsluttes en mikro, der giver 80 tegn pr. linie, hvilket er det normale i al databehandling. Her er der simpelt hen ikke punkter nok på fjernsynsskærmen til, at de enkelte bogstaver kan dannes tydeligt nok. Punkterne opgives også som pixel eller linier.

Da UHF-signalet til et fjernsyn går igennem både modulator (i mikroen) og demodulator (i fjernsynet), før det når video-forstærkeren, sker der en svækkelse af signalets kvalitet. Samtidig er standards for et fjernsynsbillede lagt for mange år siden, således at opløsningen (antal punkter) ikke kan ændres.

Derfor benyttes specielle datamonitorer til f.eks. administrative opgaver og andre opgaver, hvor der stilles store krav til billedkvaliteten.

Monochrome monitorer

Monochrome monitorer har en opløsning på 18-25 MHz og er normalt baseret på videosignal som input. Den mest anvendte fosforbelægning er stadig væk P4 (sort/hvid) og P31 (sort/grøn). Den ravfarvede skærm er i stadig fremgang, idet mange eksperter mener, det er bedre for øjnene ved længere tids skærmarbejde.

Hvor grafisk anvendelse kommer ind i billedet, benyttes ofte en fosfor med længere efterglød, kaldet P39 eller P42.

Farvemonitorer

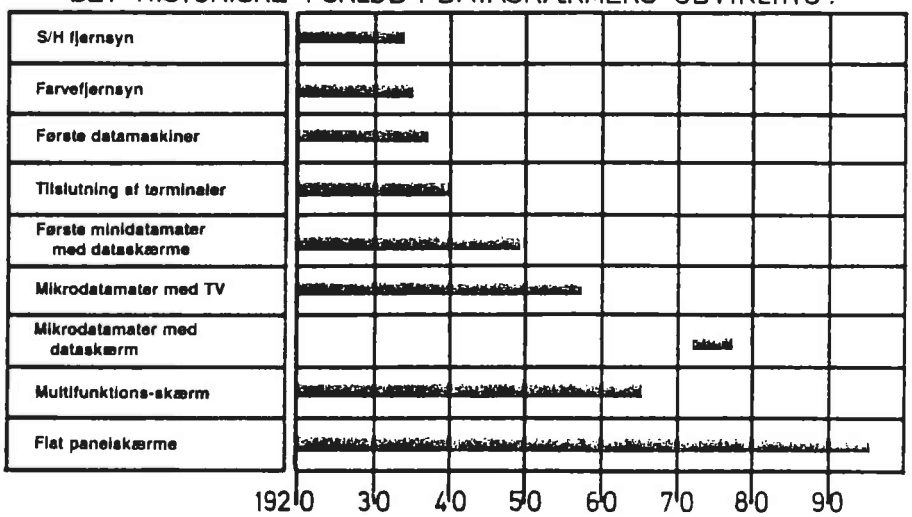
Farvemonitorer findes i et noget større spekter end de monochrome. En opløsning svarende til lidt over farve-TV er på 380 × 280 punkter. Næste spring er normalt en fordobling af den horisontale opløsning. På skærme til grafisk databehandling er opløsningen ofte 1280 × 1024. Samtidig benyttes 60 Hz scanningfrekvens og non-interlaced billedoverførelse for at sikre et helt stabilt billede.

Normalt stiger antallet af punkter naturligvis med skærmstørrelsen. For farvemonitorer på 9" til 20" ligger båndbredden på 15-40 MHz, oftest dog på 18-20 MHz, hvilket giver et godt billede med op til 2000 tegn. Skal man højere op, kræves 40 MHz til 4000 tegn eller f.eks. 960 × 720 punkter.

Et tredje væsentligt element er størrelsen af de enkelte punkter samt afstanden mellem disse. Dette måles i mm og kaldes pitch. Oftest mindskes pitch-størrelsen samtidig med forøgelsen af antal punkter. Mest anvendte størrelse er 0,31 mm.

Når der tale om datamonitorer, kommer der uvægerligt en masse udtryk og begreber ind i billedet. Begreb, de fleste ikke er bekendte med, eller ikke anvender. I det følgende gives derfor en gennemgang af grundprincipperne i opbygning af en monitor.

DET HISTORISKE FORLØB I DATASKÆRMENS UDVIKLING:



Dataskærme...

Lidt monitorteknik

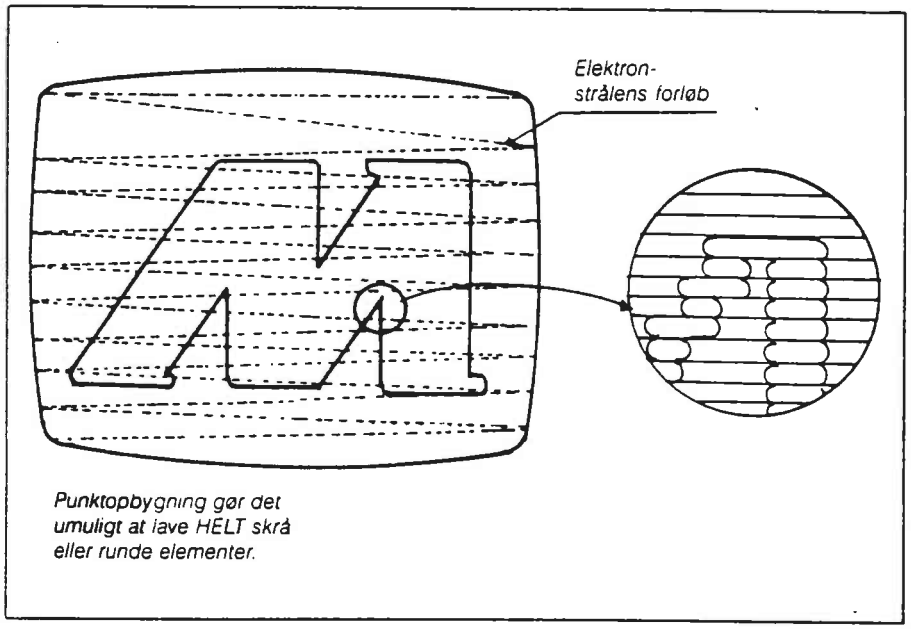
En monochrom monitor består af et billedrør med tre hovedelementer: 1) En elektronkanon, der afsender en hurtig stråle af elektroner mod billedrørets inderside, 2) en *afbøjningsspole*, der afbøjer elektronstrålen mod et givet punkt på et givet tidspunkt og 3) en *skærm*, der er belagt med én af de nævnte fosfortyper, som gløder i en brøkdel af et sekund, på det sted elektronstrålen rammer.

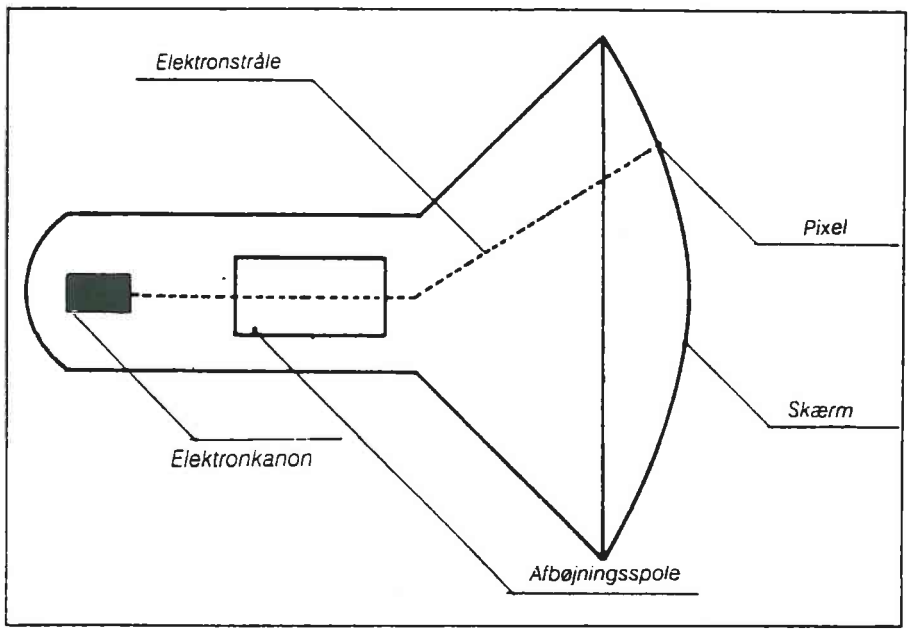
Det sted, hvor belysning finder sted, kaldes et *punkt* eller en *pixel* (picture element). Det er antallet af disse, der giver opløsningen. Styrken for belysning afhænger af den spænding, der afgives til katodestrålerøret. Da strålen gennem afbøjningen kommer over hele skærmen, skal dens intensitet ved hver pixel med kolossal præcision sættes ON og OFF, afhængig af det billede, der skal dannes på skærmen.

Da opglødningen af fosforbelægningen hurtigt forsvinder, skal elektronstrålen løbe gennem hver pixel på hele skærmen konstant. Dette sker 30-60 gange pr. sek.

Positioneringen af strålen styres af afbøjningsspolen, der består af to magnetiske spoler, der »kontrollerer« henholdsvis horisontal og vertikal retning. I et RASTER-SCAN system, som næsten alle monitorer er baseret på, starter strålen i øverste venstre hjørne og bevæger sig derefter mod højre og lidt nedad, indtil højre kant rammes. Samme bevægelse følges nu den anden vej og der fortsættes indtil nederste højre hjørne er nået, hvorefter strålen igen starter forfra. Dette gennemløb sker som nævnt mindst 30 gange pr. sek. (= 30 Hz).

Hvis denne »refresh rate« er for lille, hvad 30 gange/sek. er, ses en tydelig blinken på skærmen. Til data- og tekstbehandling benyttes derfor oftest en hastighed på 50-60 Hz eller mere.





Og så til farvemonitorer

I farvemonitorer indgår væsentlig flere elementer og mere styring end i de monochrome. En betydelig forskel er, at der indgår *tre elektronkanoner* i stedet for én. En for hver af farverne rød, grøn og blå. Heraf betegnelsen RGB-monitor. En meget vanskelig proces danner den vigtige fosforbelægning, hvor hver pixel – punkt – dannes og består af tre »dots«, én for hver basisfarve. Afhængig af hvilken farve, der skal komme til udtryk i det enkelte punkt, styres intensiteten på hver af de tre elektronstråler, der rettes mod det aktuelle punkt. For at alle tre stråler skal ramme præcis det samme sted, ligger der bag skærmen en såkaldt *»shadow-mask«*, som er en ganske tynd metalfilm med gennemperforerede huller.

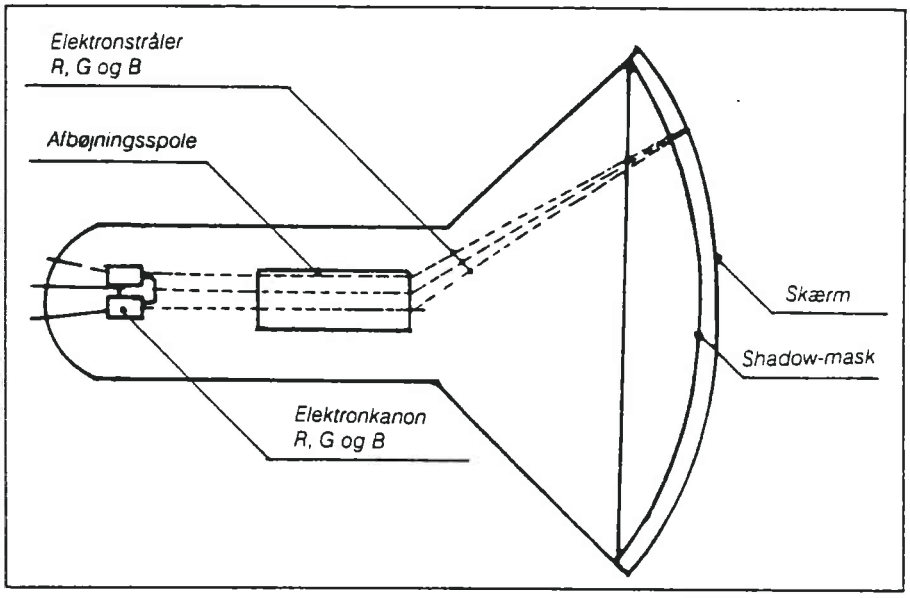
Denne lidt mere komplicerede opbygning medfører desværre, at farvemonitorer er dybere end tilsvarende størrelse s/h.

Afhængig af katodestrålerørets placering i billedrøret, skelnes mellem delta-gun og in-line gun, hvor sidstnævnte type er langt den mest udbredte.

Dataskærme . . .

Som det ses af fig. 4 er det primært pitch-størrelsen i shadow-masken, der afgør monitorens opløsning. Men som tidligere nævnt betyder også båndbredden (i MHz) en masse. Båndbredden er den hastighed, hvormed de enkelte punkt-oplysninger modtages af monitoren til styring af de enkelte farver og til fastsættelse af hastigheden for strålernes gennemløb af alle punkter på skærmen.

Hvis en monitor med opløsning på 1024 × 1024 punkter skal gennemløbes 30 gange pr. sek., kræves en båndbredde på mere end 30 MHz. D.v.s. mere end 30 millioner »bevægelser« pr. sek.

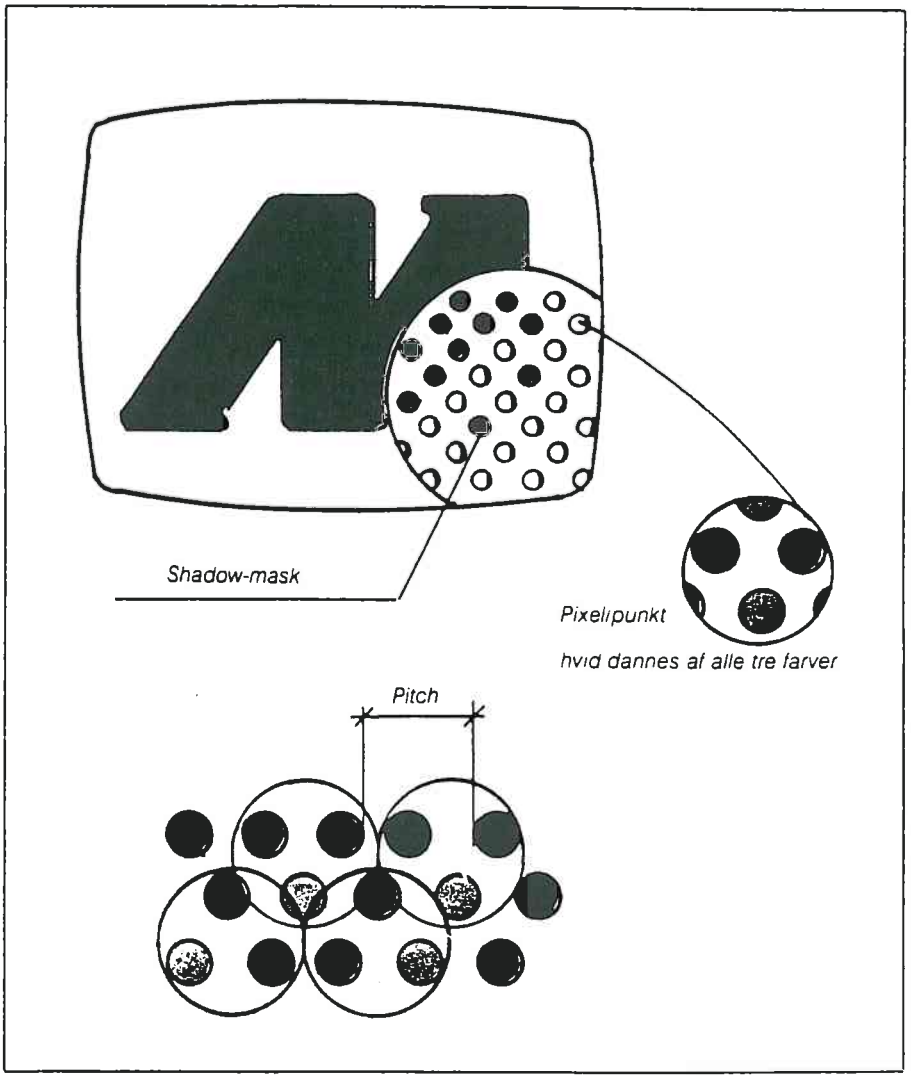


Mere end én mill. farver

Hvor mange farver, der kan vises på en skærm, afhænger af hvorledes de tre elektronstrålers intensitet kan styres. I de enkleste tilfælde opnås otte farver, styret af tre informationsbits pr. punkt: Ingen farve, rød, grøn, blå, rød/grøn, rød/blå, blå/grøn og rød/blå/grøn (= hvid).

Kræves 16 farver, hvilket betragtes som minimum til mange opgaver, skal man bruge fire bits til styringen.

Til monitorer til grafisk databehandling anvendes ofte 8-bits koder til styring af HVER af de tre farvekanoner. Dette giver 256 × 256 × 256 gradueringer (lysstyrker) på farveskalaen (mere end 16 mill.).



RGB og Composite Video

Hovedforskellen mellem de to typer er om de tre elektronstråler kontrolleres af ét sammensat (composite) videosignal eller om de kontrolleres af tre signaler (RGB). Composite video svarer meget til den metode, der anvendes i et almindeligt farve-TV. Det siger næsten sig selv, at en større præcision kan opnås ved RGB-udgaven, som da også benyttes ved stort set alle professionelle systemer.

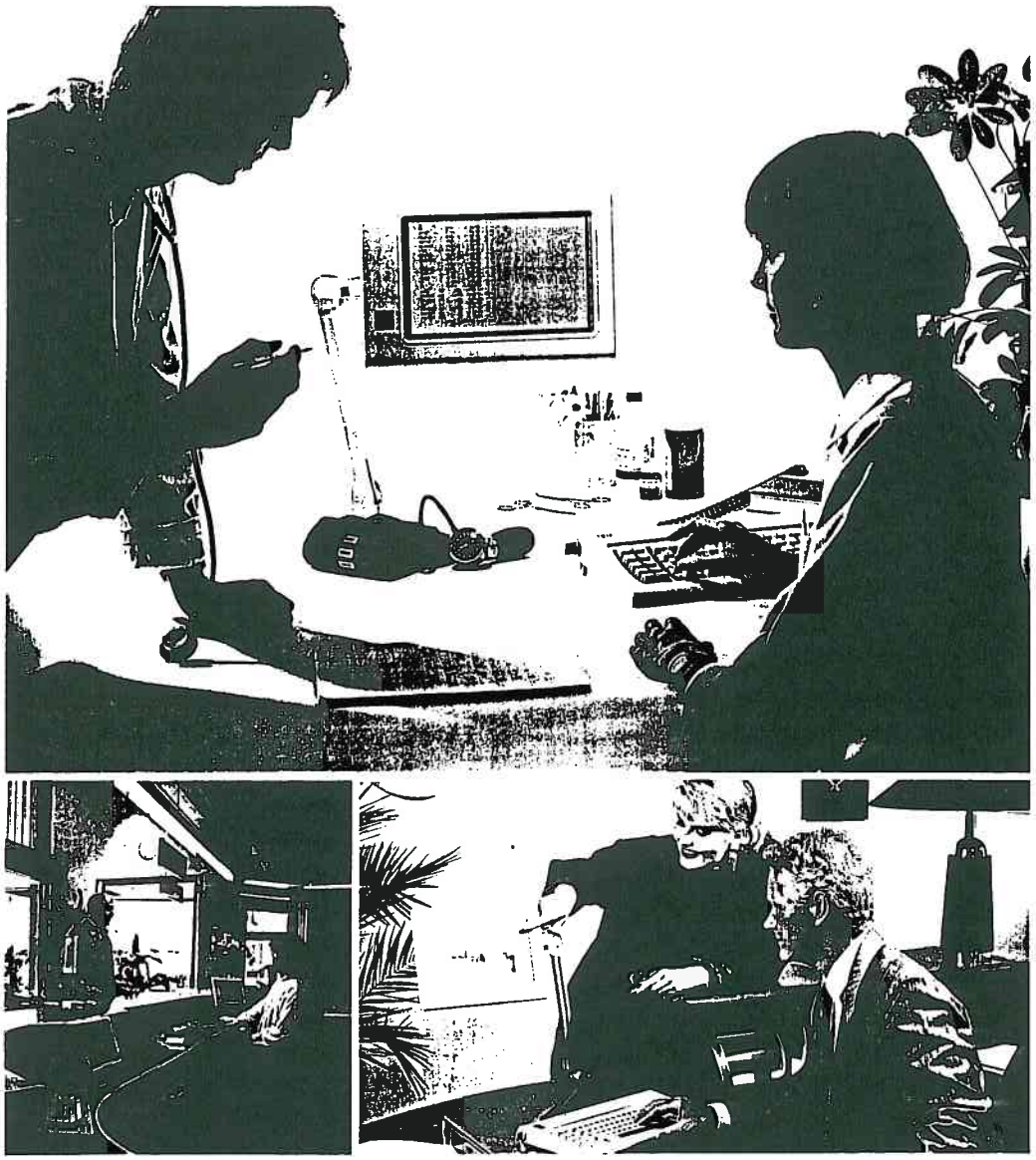
Priser

Farvemonitorer fås fra ca. kr. 3000 og opefter. Jo flere farver og jo større opløsning, desto højere pris. En ultra-high resolution monitor kan koste op til knap 100.000 kr. □

2.2.b

LCD
Flat-screen

CTC
COMPUTER



- Et alternativ til CRT skærme
- Pladsbesparende
- Ergonomisk
- Skånsom mod øjene

Spar plads

LCD-skærmen fylder kun 5% af en standard CRT monitor og vil gøre ethvert kontormiljø mere raffineret. Den enkle elegante arm, der holder LCD-skærmen, gør det muligt at placere skærmen netop der hvor det er mest behageligt. LCD-skærmen kan også ideelt monteres uden arm direkte på et kontormøbel.

Skånsom mod dine øjne

Når du begynder at arbejde med LCD-skærmen, vil du blive overrasket over hvor hurtigt du vender dig til den. En konstant klage fra CRT-brugerne er anstrengelsen af øjnene på grund af flimmer fra displayet og det store kontrastniveau mellem tekst og baggrund. CTC's LCD-skærm er fri for flimmer og har en kontrast som er nær den man er vant til når man læser en bog.

Ingen stråling

Undersøgelser af sundhedsrisikoen ved stråling fra traditionelle CRT-skærme er endnu uklaret.
Sundhedsspørgsmålet er af stigende interesse og et diskussionsobjekt for både brugere, producent, medier og relevante autoriteter.

LCD-skærme giver ingen stråling.

2.2.c

Skærmopløsninger

en oversigt over nogle af de almindeligste korttyper og de tilsvarende monitorer. Oversigten er langtfra fuldstændig, da der er adskillige computerfabrikanter, som har deres egne videokort og skærme som passer til, men hvor det er ganske besværligt at finde ud af, hvilke data grejet overholder.

Korttype	*Mode*	*Opløsning*	*Antal farver*	*Linie-/billedfrekvens*
MA	tekst	720 × 350	2	18,4 kHz/50 Hz
HM	tekst	720 × 350	2	18,8 kHz/50 Hz
HM	grafik	720 × 348	2	18,8 kHz/50 Hz
CGA	tekst	320 × 200	16	15,7 kHz/60 Hz
CGA	grafik	320 × 200	4	15,7 kHz/60 Hz
EGA	tekst	720 ×350	2	21,8 kHz/60 Hz
EGA	grafik	640 × 350	16	21,8 kHz/60 Hz
VGA	tekst	720 × 400	16	31,5 kHZ/70 Hz
VGA	grafik	640 × 480	16	31,5 kHz/60 Hz
8514/A	grafik	1024 × 780	256	35,5 kHz/43 Hz

VGA og 8514/A hører til IBM's PS/2-serie

Oversigt over forskellige grafik-modes

2.2.d

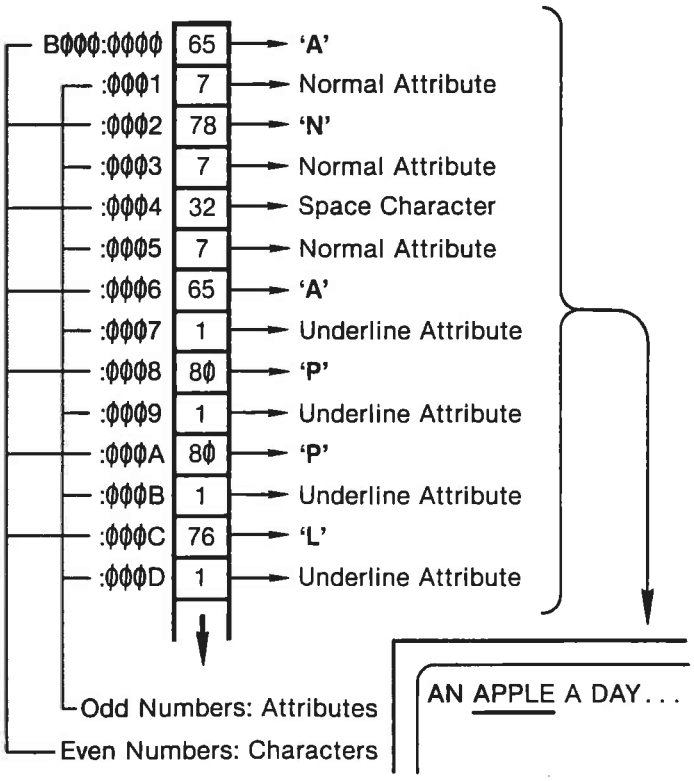
Skærmhukommelsen

Fra programkode til skærmbillede

Der er to principielt forskellige metoder til at overføre information til en skærm. Den ene metode er at udstede kommandoer (typisk som escape sekvenser) i stil med: Vis følgende tekst: "Velkommen til ..." på linie 3, søjle 5 på skærmen i omvendt video. Den anden er, at lade adapteren skandere et lagerområde, som et program kan skrive i, og vise det fundne på monitoren ("Bit mapping"). Det sidste er det almindeligste til PCere.

Ethvert program har normalt fuld adgang til det lager (skærmbufferen), der skanderes, da det er en del af 1 MB adresserummet. Der benyttes op til 128 KB i adresserummet i området A000:0000 - C000:0000. Den monokrome buffer begynder således på adresse B000:0000 og CGA bufferen på adresse B800:0000 (se fig. 2.3.1). EGA og senere standarder benytter også B800:0000 til tekst, men til grafik er det mere indviklet, da der måske skal bruges mere end 128 KB.

Arbejdes med tekst, fylder hvert tegn 2 byte i bufferen, nemlig en byte til tegnets ASCII kode på lige adresse og en byte til tegnets attributter på efterfølgende ulige



Memory mapping on the monochrome adaptor.

				0	0	0	0	Sort
				0	0	0	1	Understregning
				0	0	1	0	Grøn
				0	0	1	1	
				0	1	0	0	
				0	1	0	1	
				0	1	1	0	
	Forgrunds-			0	1	1	1	
	farver			1	0	0	0	Sort
				1	0	0	1	Understreget,høj intensitet
				1	0	1	0	Grøn, høj intensitet
				1	0	1	1	
				1	1	0	0	
				1	1	0	1	
	Blink-----1			1	1	1	0	
	Ikke blink-----0			1	1	1	1	

	R	G	B	I	R	G	B
Sort	0	0	0				
Grøn, omvendt video (dog sort ved understregning)	0	0	1				
	0	1	0				
	0	1	1		Baggrunds-		
	1	0	0		farver		
	1	0	1				
	1	1	0				
	1	1	1				

Figur 3.3.2. Attribut-bittenes betydning for IBMs monokrome skærm. Bemærk, at kombinationen: understreget forgrund + grøn baggrund, ikke giver omvendt video, men blot understregning.

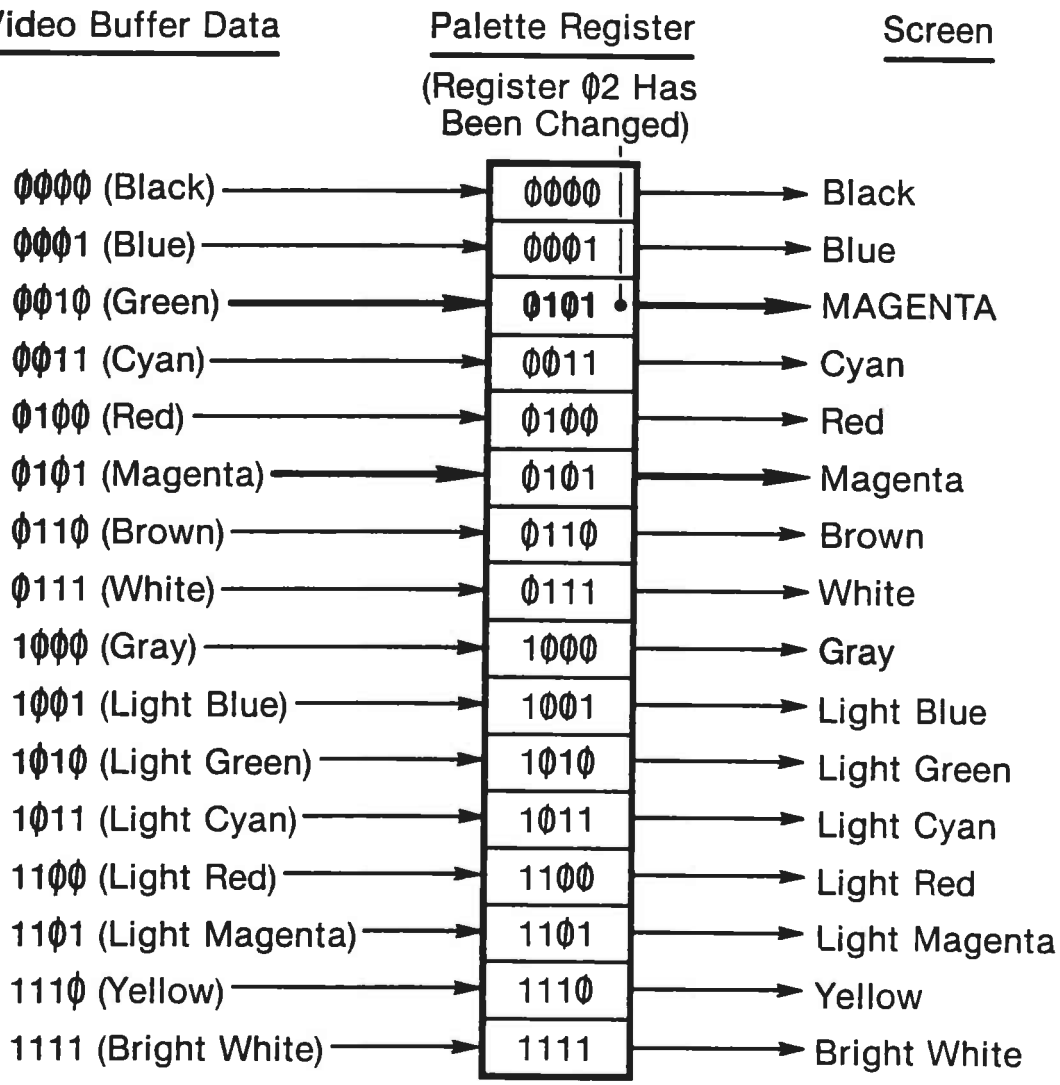
				0	0	0	0	Sort
				0	0	0	1	Blå
				0	0	1	0	Grøn
				0	0	1	1	Cyan
				0	1	0	0	Rød
				0	1	0	1	Magenta
				0	1	1	0	Brun
	Forgrunds-			0	1	1	1	Hvid
	farver			1	0	0	0	Grå
				1	0	0	1	Lyseblå
				1	0	1	0	Lysegrøn
				1	0	1	1	Lys cyan
				1	1	0	0	Lyserød
				1	1	0	1	Lys magenta
	Blink-----1			1	1	1	0	Gul
	Ikke blink-----0			1	1	1	1	Kraftigt hvid

	R	G	B	I	R	G	B
Sort	0	0	0				
Blå	0	0	1				
Grøn	0	1	0				
Cyan	0	1	1		Baggrunds-		
Rød	1	0	0		farver		
Magenta	1	0	1				
Brun	1	1	0				
Hvid	1	1	1				

Figur 3.3.1. Attribut-bittenes betydning ved tekst på en RGBI monitor. Under VGA standarden kan I-bitten benyttes til noget helt andet end intensitet, nemlig til at vælge mellem to tegnsæt indlæst i RAM; hvert tegn på skærmen kan altså hentes fra en af to tegntabeller, svarende til 512 forskellige tegn. Det samlede antal tegntabeller, der kan vælges fra er 8.

Farvespalette

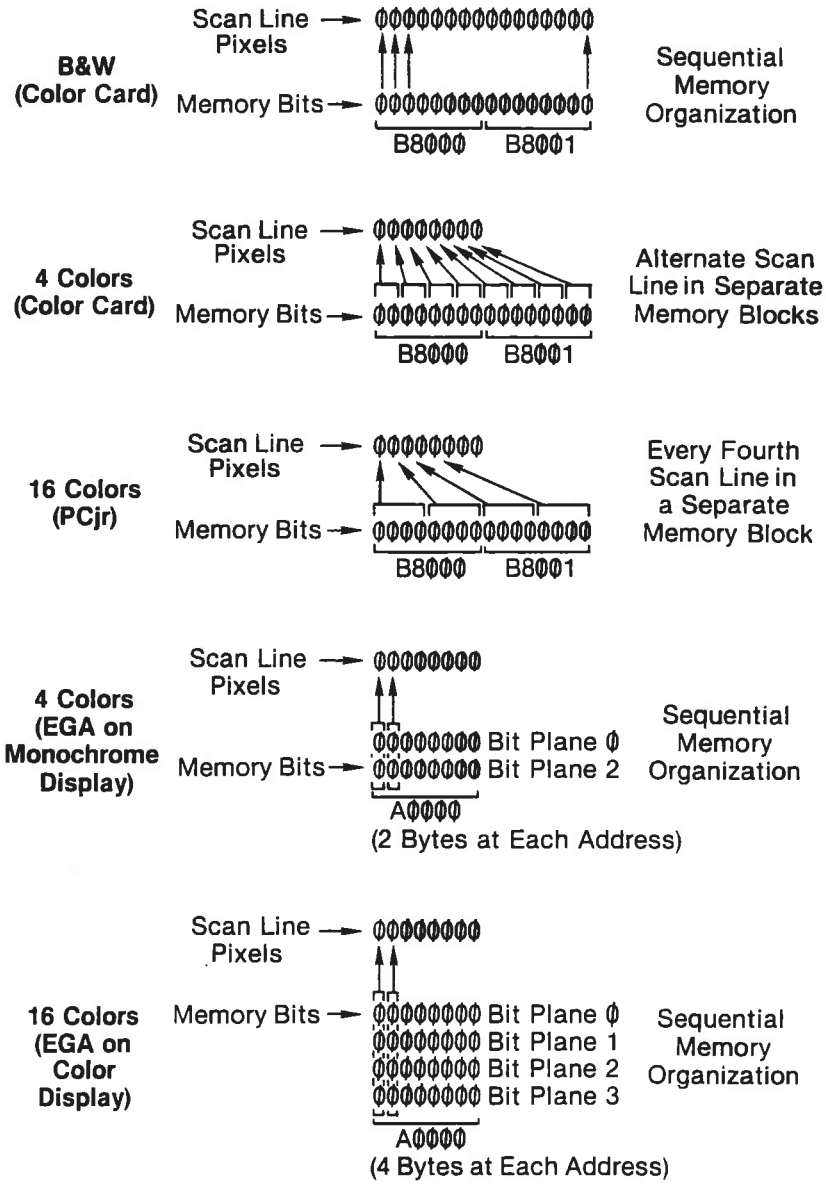
På EGA og VGA skærmkort anvedes skærmattibutter ikke direkte, men der sker et farvevalg ud fra en palette. Der kan ved ændring af paletten vælges andre farver f.eks 7 røde nuancer istedet for rød, grøn, blå o.s.v.



Displaying "green" as magenta.

Grafikskærmen

Der findes flere måder, at opbygge skærmhukommelsen på, når skærmen skal anvedes til grafik.



Four strategies for graphics layout in memory.

2.3.a

Tastaturet

Bjørk Busch

Keyboard'et har sin egen indbyggede microprocessor, på IBM-PC'er og mange compatible maskiner en INTEL 8048 chip.

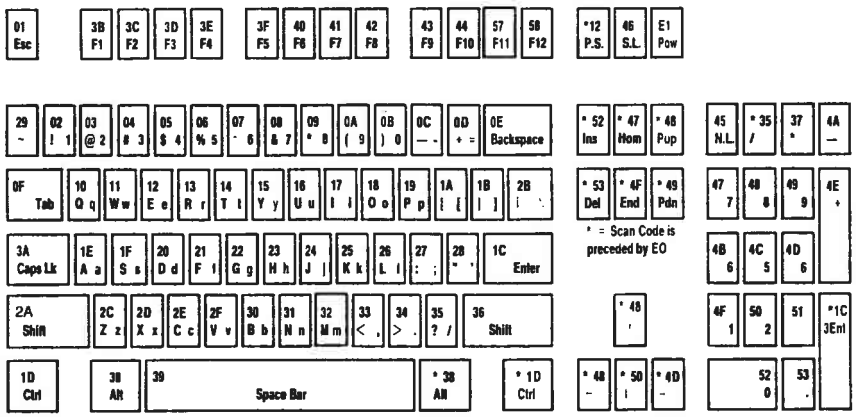
Denne processor har til opgave, kontinuerligt at "scanne" tastaturet for at opdage, om et tastetryk har fundet sted (eller om en taste er blevet "sluppet" igen).

Når et tastetryk (eller -slip) er konstateret, genererer 8048 en SCANKODE for den pågældende taste og anbringer denne kode i en buffer i selve tastaturet. Denne buffer har plads til max. 19 scan-koder plus en "stopklods-kode", som er FFH. Når alle pladser er fyldt, og der finder yderligere indtastninger sted, så ignoreres disse tastninger.

En SCAN-KODE er et simpelt løbenummer for tastaturets taster. Som regel starter nummereringen med tasten "Esc" som scan-kode nr 1, tallet 1 og tegnet ! (udråbstegn) er fælles om scan-kode nr 2 og tallet 2, tegnet " og tegnet @ deler alle 3 samme taste og er fælles om scan-kode nr. 3. Kun undtagelsesvis genereres der forskellige scan-koder for samme taste.
Taster, som har samme tegn eller funktion, genererer hver sin scan-kode.
Tallet 1 fra hoved-tastaturet giver en anden kode end tallet 1 fra nummer-tastaturet, og tegnet '*', som ofte findes i flere taster, genererer også hver sin scan-kode.

O.s.v. Der findes naturligvis optegnelser om disse scan-koder i computerens tekniske manualer.
Nedenfor er vist en oversigt over en tastatur med tilhørende scankoder.
Der gøres op mærksom på, at tildelingen af scan-koder til de enkelte taster i høj
grad er fabrikat-afhængig. Scan-koden for et taste-"slip" sættes lig med tastens scan-kode + 128.

Hexadecimal scan codes for an Enhanced Keyboard.



Samtidig med, at 8048 anbringer scan-koden på plads i bufferen, så udsteder den et hardware-interrupt (nr. 9H) til hoved-processoren 8088.

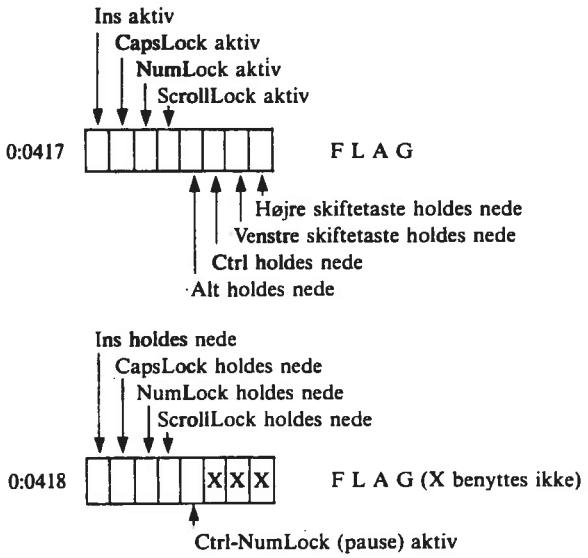
Den bios-rutine, som dette interrupt initierer, udfører nu flere på hinanden følgende opgaver:

- Der sættes et "READY" -signal op på en af I/O portene (portene 60H til 63H er på IBM-pc reserveret til keyboardet) hvilket betydere "Kom!, hvad har du at sende".

- 8048 svarer med at sætte den scan-kode op, hvis tur det er til at blive behandlet, op i en af de andre i/o-porte.

- Bios-rutinen tager nu scankoden ind via systembussen og analyserer den. Har den værdien FFH, så er det tegn på, at tastaturets buffer er løbet fuld, og bipperen aktiveres som en advarsel til brugeren.

- Hvis scan-koden betyder, at en af følgende taster har været aktiveret:
  <venstre SHIFT>, <højre SHIFT>, <CAPS LOCK>,<CTRL>, <ALT>, <NUM LOCK>, <SCROLL LOCK>, <PAUSE> el. evt. tilsvarende taster (fabrikat-afhængigt) - så opdateres en særlig bitmaske på adresse 0:417H-418H (IBM), som indeholder keyboardets STATUS-FLAG.
  Status-flagenes har følgende betydning:



- Bemærk INSERT tastet, som er lidt speciel, idet den BÅDE sætter et flag og giver en SCAN-kode i tastaturbufferen. Ins-flaget bruges sjældent, idet programmerne selv tager hånd om denne funktion, når de indlæser scan-koden fra tastaturbufferen.
  For nogle programmer er det antallet af tryk på INSERT, der afgør hvor mange pladser, der skal rykkes. Denne funktion kan ikke opnås ved, at teste den aktuelle status.

- Udfra alle andre scan-koder generer bios-rutinen nu en 2-bytes kode, hvor scan-koden anbringes i HI-order-byten og det tilsva rende ascii-tegns værdi anbringes i LOW-order byten. Ascii-tegn værdien findes ved opslag i systemenhedens skrift-tegn tabel med et index, der først er blevet beregnet udfra scan-koden og udfra det bitmønster, som fremgår af status-flagene. Hvis der er tale om tastetryk, der stammer fra tastaturets special-taster (F-tasterne, piltaster mv.), så har disse ingen ascii-repræsentation, og 2-bytes-kodens Low-order-byte får derfor værdien 00H.

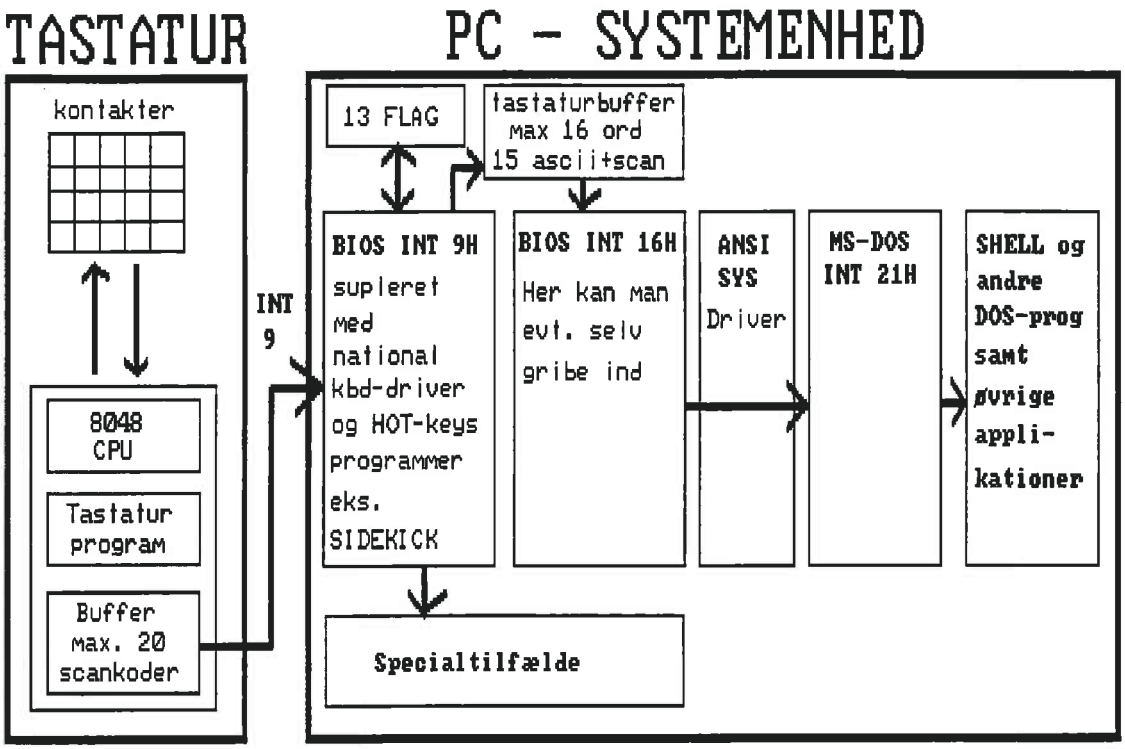
- Bios-rutinen noterer det nøjagtige tidspunkt for hvert eneste tastetryk. Hvis et tastetryk ikke er blevet ophævet (ved modtagelsen af den tilsvarende scan-kode for "slip" af tasten) inden et halvt sekund, så vil bios-rutinen af sig selv automatisk generere en scan-kode for taste-tryk for hver tiendedel sekund, der går, inden taste-slippet konstateres. Dette gælder for langt de fleste af tastaturets taster, men ikke for dem alle. Prøv dig frem og lav en liste over de taster, der ikke kan "repetere".

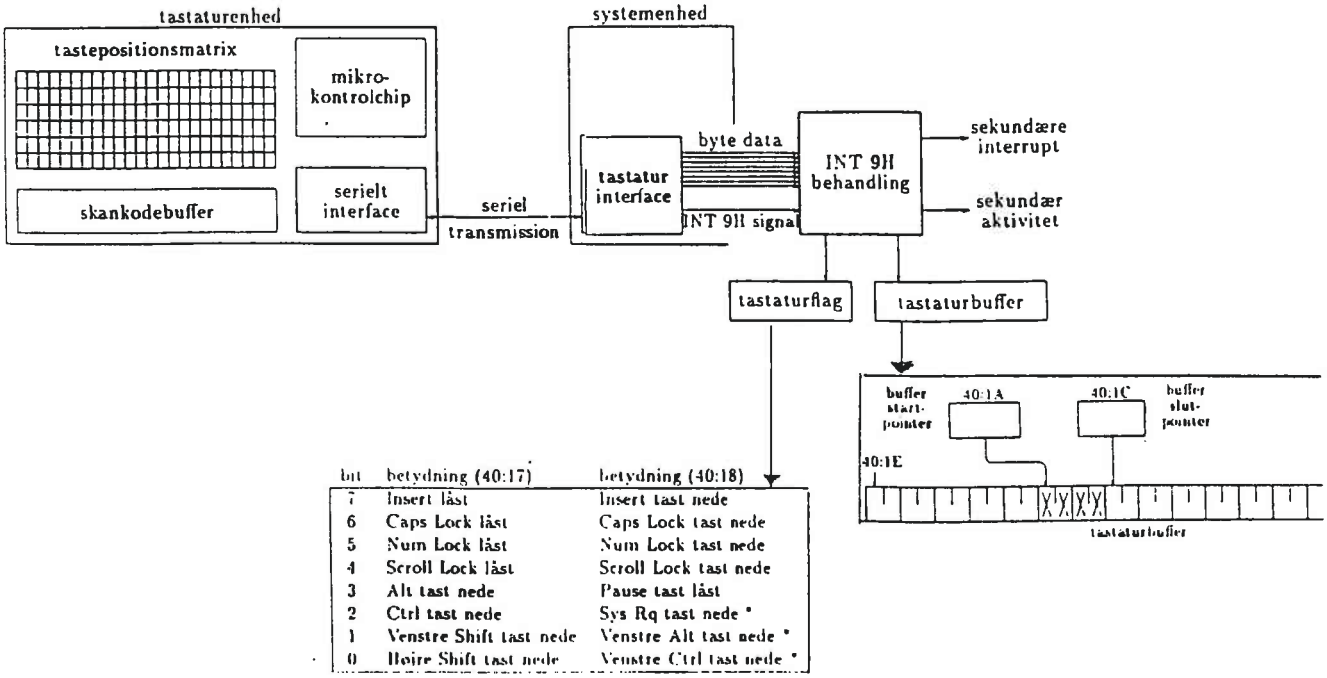
- De genererede 2-bytes koder anbringes i en key-board-buffer, der er 16 ord lang og organiseret som en cirkulær kø (ring-kø).
  Ringen styres af to pointere på hver et ord et HOVED og en HALE.
  HOVEDET peger på det næste tegn, som ligger klar til modtagelse af programmet.
  HALEN peger på det sted hvor et nyt indtastet tegn skal placeres.
  Hvis HOVED og HALE peger på samme tegn er bufferen tom, hvilket betyder, at der KUN kan placers 15 tegn i bufferen. Hvis der blev placeret 16 tegn ville HOVED og HALE nemlig igen pege på samme tegn, og der kunne ikke skelnes mellem en fuld og en tom buffer. Der kan altså "huskes" ialt 15 tastetryk, som er til rådighed for DOS eller for et brugerprogram i "FIFO"-rækkefølge.
  Princip og operationer for en cirkulær kø er ilustreret senere.

- Hermed er bios-rutinen, der kaldtes fra interrupt nr. 9H, færdig med sit arbejde og kommandoen returneres til programmet.

Når brugerprogrammet (eller DOS) ønsker at "se" og anvende de tegn koder, der er placeret i keyboard-bufferen, anvender det (via dets læse-instruktion) en anden bios-rutine, som kaldes fra software-interruptet 16H.

Tastaturet og behandlingen frem til modtager

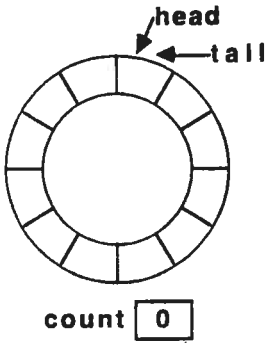




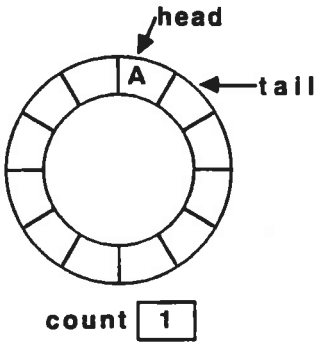
bit	betydning (40:17)	betydning (40:18)
7	Insert låst	Insert tast nede
6	Caps Lock låst	Caps Lock tast nede
5	Num Lock låst	Num Lock tast nede
4	Scroll Lock låst	Scroll Lock tast nede
3	Alt tast nede	Pause tast låst
2	Ctrl tast nede	Sys Rq tast nede *
1	Venstre Shift tast nede	Venstre Alt tast nede *
0	Høire Shift tast nede	Venstre Ctrl tast nede *

Tastaturbufferen fungerer som en cirkulær kø:

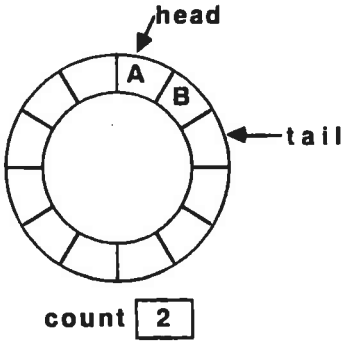
1/ Initial situation (empty queue)



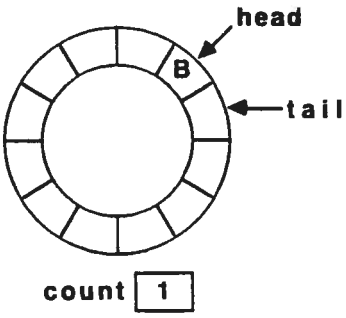
2/ After enqueue (value = A)



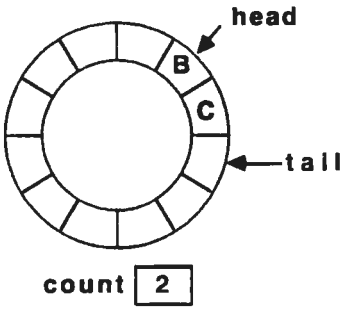
3/ After enqueue (value = B)



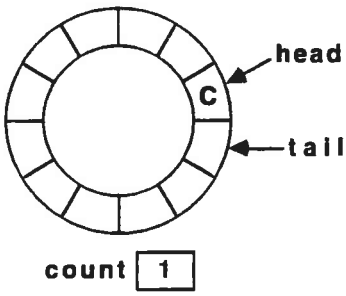
4/ After dequeue (A dequeued)



5/ After enqueue (value = C)



6/ After dequeue (B dequeued)



Enqueue Operation	Dequeue Operation
ring[tail] := value advance tail count := count + 1	value = ring[head] advance head count := count - 1

Circular Queue Operation

2.3.b

Vejledning til tastatur-demo-programmet DEMOKBD.COM

Bjørk Busch

```
          Demo af BIOS-ens keyboardbuffer og tastaturflag (BBB 09/88)

Programmet viser hvorledes, der arbejdes med keyboardbufferen - opsamlingen
af indtastninger - tidsforskydningen - og den senere læsning fra et program

Programmet standses ved at trykke på CTRL + ALT + SHIFT(en af dem) samtidig.
Ved tryk samtidig på CTRL + ALT vil programmet foretage en læsning af et tegn.

Lad være med at taste for hurtigt, da programmet ikke reagerer med det samme.
Tegn, som ligger klar til indlæsning fra program, blinker med normal farve.
Tegn, som er under indlæsning / lige indlæst, har en anden farve.
```

```
Indtast hastighed   1-9 (norm 5)
```

```
hast:1    Demo af BIOS-ens keyboardbuffer og tastaturflag (BBB 09/88)      369

Programmet viser hvorledes, der arbejdes med keyboardbufferen - opsamlingen
af indtastninger - tidsforskydningen - og den senere læsning fra et program

Programmet standses ved at trykke på CTRL + ALT + SHIFT(en af dem) samtidig.
Ved tryk samtidig på CTRL + ALT vil programmet foretage en læsning af et tegn.

Lad være med at taste for hurtigt, da programmet ikke reagerer med det samme.
Tegn, som ligger klar til indlæsning fra program, blinker med normal farve.
Tegn, som er under indlæsning / lige indlæst, har en anden farve.
```

```
Hoved:0022 Hale:0028                    Flag1: 00001110  Flag2: 00000011
Indhold af Keyboard Buffer
ASCII kode                              b  d  .     1  .  j  j  j
ASCII kode  HEX          00 00 00 00 00 62 64 0D 00 31 0D 6A 6A 6A 00 00
SCAN/extented kode       19 19 19 19 19 30 20 1C 19 02 1C 24 24 24 19 19
                         Hoved !
                                 Hale !
```

2.3.c

Sådan kommer trætte PC-fingre i fin form

Tusindvis af mennesker bliver invalideret af at lave forkerte og ensformige bevægelser, når de skriver ved computer-tastaturet

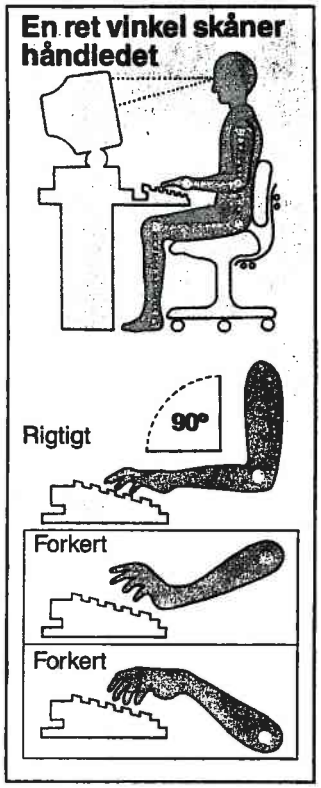
Arbejdsskader er noget, de fleste forbinder med hårdt fysisk arbejde, tunge løft eller indånding af sundhedsskadelige stoffer på arbejdspladsen. Men et stort antal mennesker, hvis daglige arbejde hovedsagelig består i at skrive på et computer-tastatur, plages af svære lidelser i fingre og håndled. De ensformige, små bevægelser ved tastaturet kan forårsage lammelser, der kan forkrøble hænderne i en sådan grad, at man ikke kan bruge dem normalt længere.

Lammelserne skyldes bl.a., at muskler og væv omkring håndroden hæver og klemmer de nerver, som har forbindelse til fingrene. De varige skader kan heldigvis forebygges, inden de bliver invaliderende, bl.a. ved hjælp af en række øvelser, der styrker hænder og håndled. Mange kontorarbejdere bliver nærmest en slags tastatur-atleter, der skriver i rasende fart mange timer i træk. Men kun de færreste holder fingrene i form og sørger for at strække de overanstrengte muskler.

Tastatur-atleter bør holde håndled og hænder lige i en 90° vinkel fra albuen og undgå at bøje og vride håndledene, mens de skriver. Det belaster også fingrene unødvendigt, hvis man har for vane at slå hårdt på tasterne. De fleste tastaturer reagerer nemlig på ganske blide anslag.

Hvis det er muligt, bør man tilrettelægge sit arbejde, så man veksler mellem at skrive på tastaturet og udføre andre funktioner – hvor man vel at mærke også sørger for at undgå at belaste håndledene.

Dr. Marvin J. Dainoff, der leder et ergonomisk forskningscenter i Ohio, USA, har udviklet træningsprogrammer for folk, som tilbringer mange timer dagligt ved et tastatur. Ifølge Dainoff bliver mange overraskede over, hvor plagsomme skader arbejdet ved en PC kan medføre. □



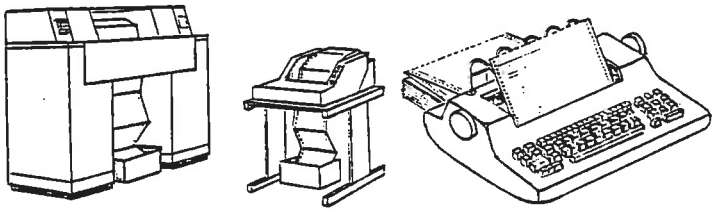
Hænder og håndled belastes mindst, hvis de holdes i en 90° ret vinkel ud fra albuen.

Simple øvelser udført dagligt kan skåne edb-arbejdere, sekretærer og skribenter for varige skader i hænder og håndled.

2.4.a

Printere & plottere
fra
Grundlæggende databehandling
John Hansen & Carl Tange

Linieskriver (engelsk: Line Printer)



"Normal" Linieskriver | Matrix-skriver | Elektrisk skrivemaskine

Symbol:

Linieskriveren er den ydre enhed, der sørger for uddata i skriftlig form. Der findes en stor mængde forskellige linieskrivere på markedet, varierende fra enkle, elektriske skrivemaskiner, der kan skrive 8-10 tegn pr. sekund, og op til avancerede linieskrivere baseret på elektroniske principper eller laserskrivere, der kan skrive op til 20.000 linier pr. min.

Anvendelse

Hvilken type linieskriver, der skal vælges til en aktuel maskinkonfiguration, afhænger helt af anvendelsesområderne for det pågældende anlæg. Hvis anlægget skal bruges til informationssøgning, som ved politiets motorregister, eller til beregningsopgaver med en lille ind- og uddatamængde, er en elektrisk skrivemaskine tilstrækkelig. Drejer det sig derimod om et edb-servicebureau, der f.eks. kører bogholderiopgaver og lønsystemer, er der behov for en af de hurtige linieskrivere.

Fælles for alle typer linieskrivere er, at de er istand til at producere skriftlige uddata styret af programmet i centralenheden. Dette betyder, at man er nødt til selv at fastlægge, hvilke uddata man ønsker skrevet, og den form, de ønskes skrevet på, d.v.s. opbygningen af den enkelte linie med mellemrum, tekster og variable oplysninger. Man er selv nødt til at definere overskriftslinier, hvilke blanke linier der ønskes i udskriften o.s.v. Alle disse oplysninger skal så kodes i programmet, der så kan styre udskrivningen på linieskriveren i overensstemmelse med brugerens ønsker.

Linieskriveren kan anvendes til udskrivning på blanke lister (Leporello-lister), hvor man selv skal sørge for tekster og lign., samt på fortrykte formularer (lønsedler, kontoudtog og lign.), hvor den faste tekst er fortrykt. Normalt kan en linieskriver skrive 10 tegn pr. tomme (vandret), og 6 linier pr. tomme (lodret).

Tekniske krav

Set fra et brugersynspunkt, er de tekniske krav til en linieskriver forholdsvis enkle:

- Linieskriveren skal være udstyret med det ønskede skriftbillede.
- Linieskriveren skal evt. kunne skrive med både store og små bogstaver (normalt skrives udelukkende med store bogstaver).
- Linieskriveren skal kunne skrive med en hastighed, der står i forhold til den forventede uddatamængde.
- Skrivebredden (antal anslag pr. linie) skal være tilstrækkelig (normalt 120-132 anslag pr. linie).
- Mulighed for at kunne skrive på løsblade istedet for endeløse baner. Der findes dog speciellle formularsæt, der giver denne mulighed.
- Mulighed for at kunne udskrive med det ønskede antal kopier.

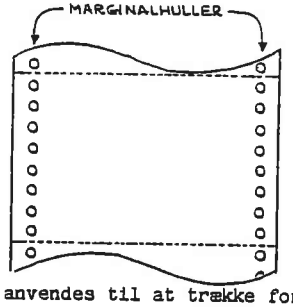
Hvorledes linieskriveren rent teknisk er opbygget, er set fra brugerens synspunkt, ret ligegyldigt, blot
den opfylder de kapacitetsmæssige krav, der stilles i form af ønsker til udskriftens udseende, mængden af udskrift m.v.
Gennemgang af nogle typer linieskrivere findes på de efterfølgende sider.

Linieskrivere, teknik

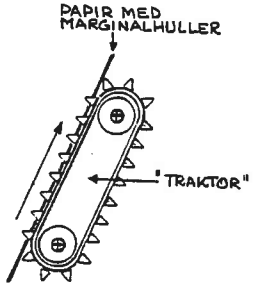
Som tidligere nævnt findes der et meget stort antal forskellige linieskrivertyper, varierende fra en elektrisk skrivemaskine til meget avancerede "sideskrivere" baseret på anvendelse af laser-stråler . Der findes nogle egenskaber fælles for en række linieskrivere, og disse egenskaber skal gennemgås nedenfor inden gennemgangen af de specifikke skrivetyper

Papir-fremføring

Papir til udskrivning på linieskrivere er normalt i endeløse baner, og er udstyret med maginalhuller.

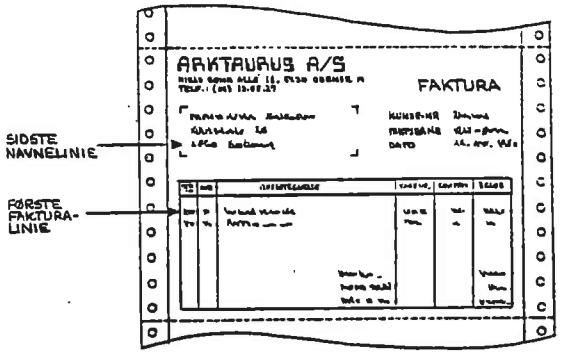


Disse huller anvendes til at trække formularen gennem linieskriveren, idet linieskriveren er udstyret med et eller flere sæt "traktorer", der griber fat i formularens maginalhuller, og dermed styrer formularen under udskrivningen.



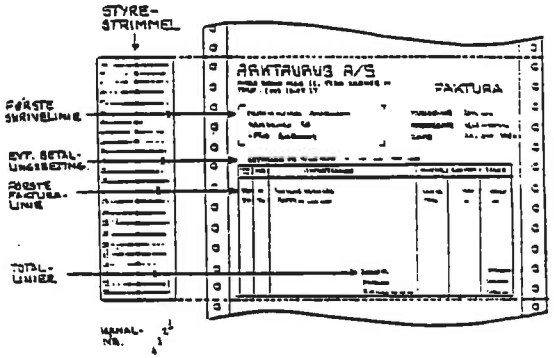
Skip-funktion

Mange af de hurtige linieskrivere er udstyrede med en "skip-funktion", hvilket vil sige en mulighed for, med høj hastighed, at køre formularen frem til den næste linie, der ønskes skrevet på.



Man skal f.eks. i ovenstående faktura springe over 5 linier fra sidste navnelinie til første linie på selve fakturaen. Man kan lade linieskriveren skrive 5 blanke linier, men man kan også lade den "skippe" frem, hvilket er en langt hurtigere funktion. Skip-mekanismen ligger i linieskriveren, men styres fra programmet i centralenheden. Til linieskriveren fremstilles enten en styre-strimmel (carriage-tape) svarende til den konkrete formulartype, der skal udskrives, og hvori der er angivet hvormange linier, der skal springes over ved hvert hop, eller linieskriveren kan programmeres til at foretage disse hop. I dette tilfælde findes der et program for hver formulartype.

Carriage-tape

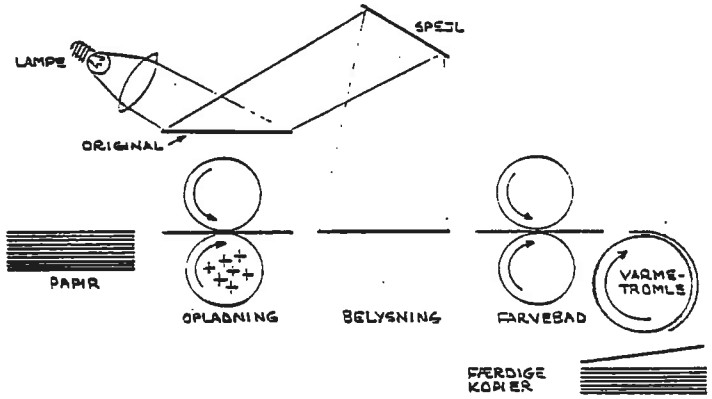


Elektrostatisk kopiering

Det elektrostatiske princip er en metode, der anvendes ved nogle af de moderne, hurtige linieskrivere (og iøvrigt ved nogle typer fotokopieringsmaskiner).

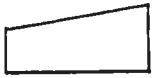
Grundprincippet er baseret på det forhold, at papir der er opladet med en statisk, elektrisk spænding, aflades hvis det belyses.

Man oplader et stykke papir med en elektrisk spænding. Dernæst projiceres et billede over på det opladede papir af den original, der ønskes kopieret (bogstaver, tegninger o.s.v.). Hvor originalbilledet er mørkt (bogstaver, rammer m.v.) sker der intet med den elektriske spænding, mens papiret aflades alle andre steder. Tilbage bliver et billede på papiret af originalen i form af den elektriske spænding. Papiret sendes nu gennem et farvebad, hvor et meget fint, sort farvepulver "klæber" fast til de elektriske spændinger, og dermed danner en kopi af originalen. Endelig brændes farvepulveret fast til papiret, hvorefter kopien er færdig.



Skrivemaskinen Den enkleste type linieskriver er en elektrisk skrivemaskine.

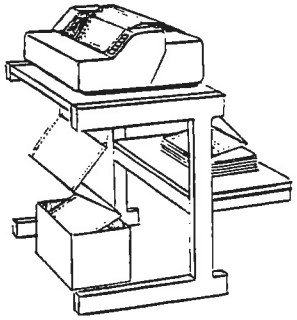
Symbol:



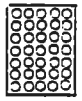
Udskrivningskapaciteten er ret ringe, ca. 8-10 tegn pr. sekund, således at denne type skriveenhed kun kan anvendes ved relativ små uddatamængder. Det skal dog anføres, at denne skriveenhed kan skrive med både store og små bogstaver, et forhold der langtfra gælder alle de hurtige linieskrivere. Den producerer et pænt tryk, og kan skrive med 3-4 læselige gennemslag. Den kan i flere tilfælde fås med udskriftlige skriftbilleder (kuglehovedmaskinen), hvilket gør den velegnet til flere formål, bl.a. ved elektronisk tekstbehandling (ETB).

Matrix-printeren

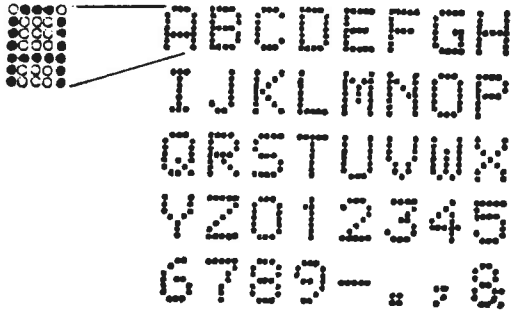
Matrix-printeren er en linieskriver-type med en udskrivningskapacitet på 80-500 linier/minut.



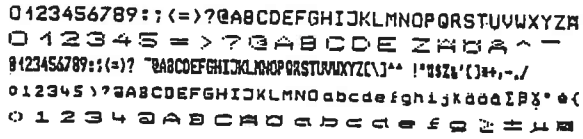
Teknikken bag en matrix-printer er et skrivehoved, der indeholder en række "nåle" siddende i en firkantet ramme (matrix).



Ved at presse et antal af disse nåle frem mod et farvebånd, ind mod papiret fremkommer de enkelte tegn.



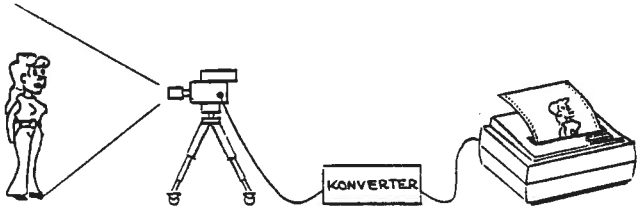
Som følge af sin konstruktion giver matrix-printeren ikke alene mulighed for at skrive med store og små bogstaver, men også mulighed for forskellige skriftbilleder i samme udskrivning, idet det udelukkende er et spørgsmål om, at den "tegngenerator", der presser nålene frem, kan rumme de nødvendige kombinationer.



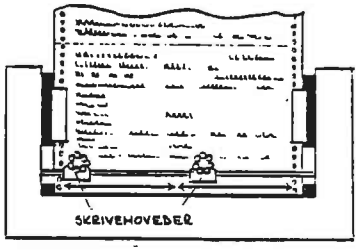
Eksempler på forskellige skriftbilleder
fra en matrix-printer

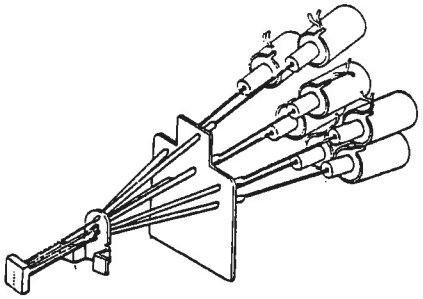
Billed-fremstilling

Udover forskellige skriftbilleder kan matrix-printeren også, ved hjælp af sine nåle, "tegne" billeder. Et billede kan fremstilles som en kombination af sorte og hvide prikker i forskellig tæthed (jf. avisbilleder). Matrix-printerens nåle kan anvendes i vilkårlige kombinationer, og dermed frembringe det ønskede billede. Teknikken kan anvendes både i forbindelse med et kamera, og udfra lagrede data på f.eks. magnetbånd.



Ved skrivning af en linie bevæger skrivehovedet sig hen forbi linien, og skriver de enkelte tegn i de ønskede positioner undervejs. Ved ønske om større skrivehastigheder kan matrix-printeren udstyres med to skrivehoveder, der hver skriver 1/2 linie, og hvor skrivningen af de to halvdele sker samtidigt. Matrix-printeren kan også indrettes således, at der skrives under både frem- og tilbageløb af skrivehovedet.





Skrivehoved fra matrix-printer

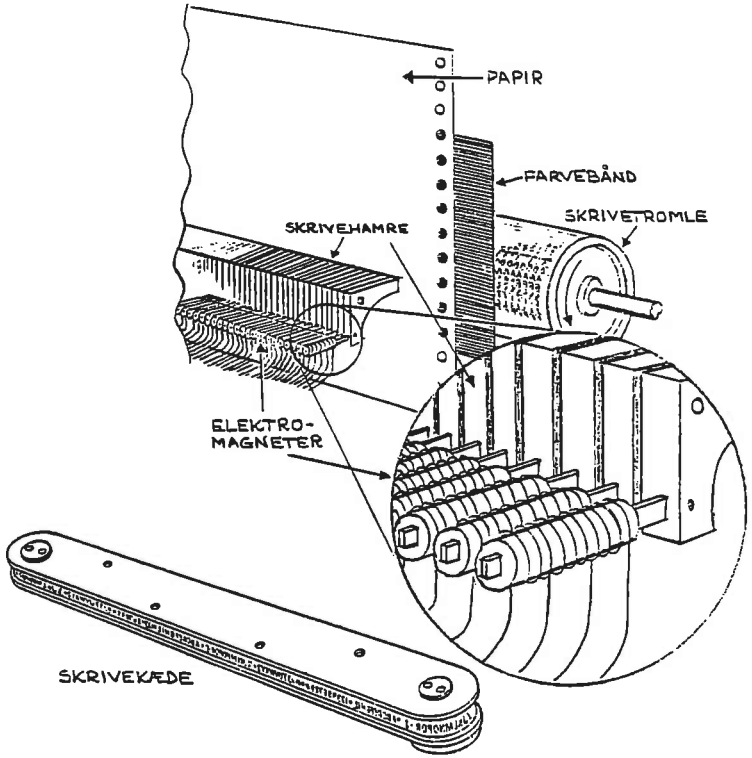
Udskrivningen sker normalt på papir i endeløse baner, idet formularen trækkes igennem linieskriveren ved hjælp af papirets marginalhuller.

Tekniske data

Matrix-printeren kan skrive fra 80 til 132 tegn pr. linie, og skriver normalt 6 linier pr. tomme. Skrivehastighederne varierer fra 80 til 500 linier pr. minut. Skrivekvaliteten er ikke overvældende god i øjeblikket, men da printertypen er meget populær p.g.a. prisbillighed og sin robusthed, kan det forventes, at den gradvis vil blive bedre. Der vil normalt kunne udskrives formularer med 1-2 kopier.

Tromle- og kædeskrivere

De to "klassiske" former for hurtige linieskrivere er kæde- og tromleskrivere. Begge typer skriver med hastigheder på 1.000 - 2.000 linier pr. minut og 132 - 150 anslag pr. linie. Begge printertyper er baseret på den teknik, at papiret trykkes ind mod et farvebånd og videre ind mod selve typen ved hjælp af en elektrisk styret hammer. Forskellen mellem de to typer er, at selve tryktyperne er monteret på henholdsvis en skrivetromle og en skrivekæde.



Skrivning med tromleskrivere

Skrivningen sker på den måde, at der fra centralenheden afleveres en linie tekst til linieskriveren. Denne kontrollerer om papiret er fremme ved den korrekte linie, eller om det skal køres frem. Når papiret er i den rigtige position og tromlen som roterer konstant har A-rækken udfor skrivelinien, udløses de hamre, der svarer til de positioner på linien, hvor der skal skrives A. Dernæst kommer B-rækken og de relevante hamre udløses o.s.v. Når tromlen er drejet en omgang, har alle tegn, bogstaver, tal og specialtegn passeret skrivelinien, og er blevet trykt, hvis de har været i den aktuelle linie.

Skrivning med kædeskrivere

Ved kædeskriveren, hvor kæden roterer vandret forbi skrivelinien, skrives de enkelte tegn efterhånden som de passerer de positioner, hvor det aktuelle tegn skal skrives. Der udløses da en hammer, hvorved tegnet trykkes på papiret. Først passerer A'et forbi linien, dernæst B o.s.v. Når kæden har roteret en omgang er linien færdig. For at sætte skrivehastigheden i vejret, indeholder nogle skrivekæder alle tegn tre gange således, at kæden kun skal dreje 1/3 omgang for at skrive en linie.

Karaktersæt

Ved tromleskriveren er karaktersættet fast (antal typer og udseende (skriftbillede)), idet det er meget vanskeligt at udskifte selve tromlen. Ved kædeskriveren kan det relativt let lade sig gøre at udskifte kæden med en anden, hvorved man kan skrive med forskellige skriftbilleder og evt. med forskellige tegn. Begge skrivertyper anvender i øjeblikket fortrinsvis store bogstaver som skriftbillede, men det må forventes, at der i tiden fremover vil ske en udvikling i retning af mulighed for både store og små bogstaver.

```
0123456789012345678901234567890**SEN NO. 05
```

Karaktersæt fra en tromleskriver

Farvebånd

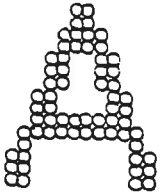
Det anvendte farvebånd er indfarvet silkebånd, ligesom ved en alm. skrivemaskine. Farvebåndet skal have samme bredde som antallet af skrivepositioner. Ved udskrivning af optisk læselig skrift (OCR) anvendes et farvebånd af en særlig god kvalitet, der kan give et meget præcist aftryk (engangsfarvebånd). Ved udskrivning af magnetisk læselig skrift (MICR) anvendes et farvebånd, hvor tryksværten indeholder et magnetiserbart materiale. Sidstnævnte anvendes dog kun meget sjældent herhjemme.

Kopier

Mekaniske linieskrivere kan udskrive formularer med 3-4 læselige kopier, hvor kopieringen sker ved hjælp af engangskarbonpapir indlagt mellem de enkelte blanketter, eller ved hjælp af en kemisk præparering af blanketterne (NCR-papir).

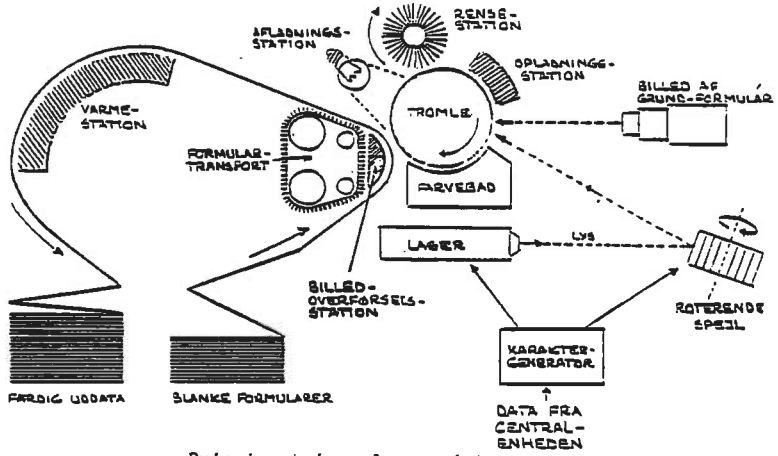
Laserprinteren En af de nyeste og hurtigste printertyper er laserprinteren. Den er baseret på det elektrostatiske kopieringsprincip, hvor den anvendte lyskilde er en laserstråle.

Hjertet i en laserprinter er en tromle, der oplades elektrostatisk. På denne tromle opbygges den ønskede tekst ved hjælp af en laserstråle, styret fra en "karaktergenerator". De enkelte tegn er opbygget af punkter, som laserstrålen gennem en række impulser "tegner" på tromlen.



Disse belyste punkter aflades elektrisk, og når hele siden er færdig, passerer tromlen gennem et farvetonebad, hvor farven lægger sig på de afladede områder mens de ikke-belyste (=elektriske) områder af tromlen frastøder farven . Billedet fra tromlen overføres nu til papiret, hvor farven brændes fast. Til slut aflades papir og tromle for den resterende elektriske spænding, og tromlen renses for farve inden overførsel af næste billede.

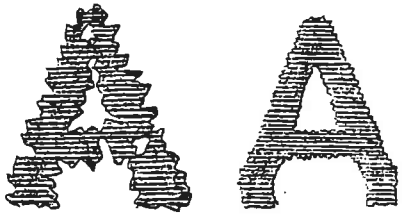
Samtidig med overførslen af de enkelte tegn i udskriften til tromlen via laserstrålen, kan man fra en anden lyskilde via et negativ overføre et billede af selve formularen (streger, fortekster m.v.) til tromlen således, at resultatet fremtræder som trykt på en fortrykt formular. Dette kan reducere omkostningerne til fortrykte formularer ganske væsentligt, samt forøge fleksibiliteten i anvendelse af disse, idet man kan ændre formularudseendet efter behov og ikke efter hvor stort lager man har liggende.



Princippet i en laserprinter

Skriftbillede I kraft af sin konstruktion kan laserprinteren anvende vilkårlige skriftbilleder. Det er et spørgsmål om nødvendig programmering af karaktergeneratoren, der danner de enkelte tegn ved hjælp af punkter.

Kvalitet Skrivekvaliteten af en laserprinter er bedre end for de traditionelle linieskrivere. Farvetætheden pr. tegn er væsentlig større, hvilket resulterer i et lettere læseligt skriftbillede.



Mekanisk printer — Laser printer

Hastighed

Laserprinteren kan udskrive 20.000 linier pr. min. hvilket er mere end 10 gange hurtigere end de normalt anvendte linieskrivere. Samtidig er den, som nævnt, istand til at producere fortrykte formularer udfra blanke lister, hvilket gør formularprisen billig. Den kan dog ikke, i sagens natur, udskrive med kopier. Man er her nødt til at skrive det antal eksemplarer, der ønskes, som originaler.

Andre typer linieskrivere

Udover de nævnte linieskrivertyper findes der en række andre af forskellige konstruktioner:

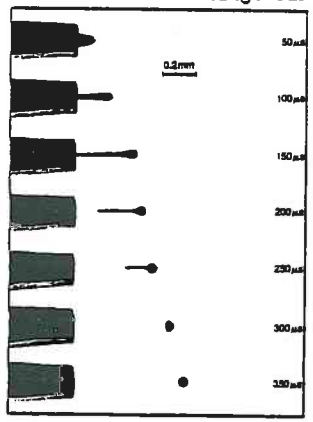
Typehjulsskriveren, baseret på et eller flere typehjul, hvert indeholdende hele tegnsættet. Efterhånden som der skrives hen ad linien, flyttes typehjul og hammer til de aktuelle positioner.

Typestangsskriveren, indeholdende hele karaktersættet på en typestang der enten vandret eller lodret passerer skrivelinien.

Den elektrostatiske printer, hvor der fra en filmmatrix overføres billeder af de enkelte tegn til en elektrisk opladet formular, hvorefter der påføres farve, som slutteligen brændes fast til formularen.

Termoskriveren, der ved hjælp af et skrivehoved med varmetråde overfører skriften til varmefølsomt papir.

Blækskriveren. (Jet Ink Printer) Ved hjælp af en dyse udsendes en fin stråle blækpartikler. Denne stråle styres elektrisk, og kan derigennem bringes til at tegne de enkelte tegn på papiret. Der udsendes ca 100.000 blækpartikler pr. sekund, og skriveren kan skrive op til 1000 tegn/min. Da styringen af blækpartiklerne sker via et program, kan skriveren arbejde med alle størrelser (i det omfang programmet kan styre det). Skriveren har også den store fordel at arbejde omtrent lydløst. Overflødige blækpartikler suges automatisk tilbage igen. Blækskriveren kan skrive med forskellige farver.

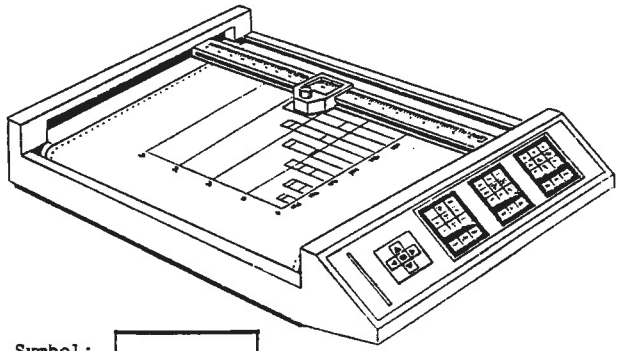


PT801 INK-JET PRINTER
I PRINT 270 CHARACTERS PER SECOND BI-DIREKTIONALLY AND EACH CAN BE *RIGHT-* OR LEFT-ITALICIZED, OR UNDERLINED.
I CAN PRINT ELONGATED CHARACTERS.
ALL OF THESE FEATURES ARE UNDER SOFTWARE CONTROL.

Eksempel på blækskriver (jet ink printer), teknik og skrivekvalitet.

Kurvetegner (engelsk: Plotter)

Kurvetegneren er en ydre enhed, der kan producere uddata i form af stregtegninger, kurver, diagrammer og lign. Kurvetegneren anvendes i forbindelse med grafisk databehandling, hvor man ønsker resultatet tegnet efter endt konstruktion via en dataskærm.



Symbol:

Princippet i en kurvetegner er, at en pen kan bevæge sig vandret og lodret hen over en papirflade, styret af de data, der ønskes afbildet. Resultatet af disse bevægelser er kurver af vilkårligt udseende, altså alle de former for grafisk afbildning, man kan ønske. Kurvetegneren kan også skrive den til afbildningen hørende tekst i samme arbejdsgang.

Mekanisk fås kurvetegnere i mange forskellige udgaver, lige fra ganske enkle enheder, der kan tegne i A4-format, til store kurvetegnere, der kan producere tegninger i 2x4 m (bruges f.eks. på Meterologisk Institut).

Tromletegner I den mest almindelige type, tromletegneren, ligger papiret på en tromle, der kan rotere. Pennen er monteret på en bjælke, og kan bevæge sig tværs over papirbanen. Ved at kombinere de to bevægelser, opnår man at kunne tegne i vilkårlige retninger. Under flytninger kan pennen hæves fra papiret.

Plantegnere I en anden konstruktion, plantegneren, er papiret placeret plant på et tegnebord. Pennen kan her bevæge sig i to retninger, vandret og lodret, og kan dermed generere den ønskede udskrift.

Elektrostatisk tegner Endelig kan kurvetegneren opbygges efter et elektrostatisk princip, hvor der ikke direkte tegnes på papiret, men hvor tegningerne opbygges af elektriske punkter, der så gennem en fremkalder-proces omdannes til en læsbar kurve. Denne type kurvetegner kræver specielt papir.

Pennen Selve pennen kan være en tuschpen, spritpen eller en avanceret udgave, en "blæk-sprøjte". Afhængig af størrelsen kan en kurvetegner arbejde med flere farver, ved faste penne op til 4 farver, ved blæksprøjten, 4 grundfarver der kan kombineres til over 100 nuancer. Pennen kan udskiftes/tilpasses den ønskede stregtykkelse.

Nøjagtighed Kurvetegneren kan arbejde med en nøjagtighed bedre en 0,1 mm, d.v.s. udstyret kan tegne mere nøjagtigt, end det er muligt manuelt.

Hastighed Tegnehastigheden afhænger stærkt af udstyrets art og størrelse, og af hvor indviklet tegningen er, men en "normal" tegning kan udskrives på 5-15 min. Det kan derfor være hensigtsmæssigt at udstyre kurvetegneren med et internt lager og en processor, således at den ikke skal belaste CPU'en under hele tegneprocessen.

Anvendelse Sammen med en grafisk dataskærm og relevant programmel udgør kurvetegneren et overordentligt godt udstyr til konstruktion og rentegning. Når man er færdig med at gennemarbejde en tegning på dataskærmen, kan den målsættes automatisk og udskrives på kurvetegneren. Tegninger og lign. kan lagres på et af datamatens ydre lagre, således at man ved ændringer kan kalde tegningen frem på skærmen, foretage ændringen, og få en ny rentegning udskrevet.

2.4.b

Control Extensions

1 Control Sequence

The BJ-300/330 printers support the following 10 extension control sequences.

Set Presentation Highlight	High Resolution Graphics
Set Vertical Units	Page Presentation Media
Select Code Page	Set Vertical Margins
Select Font	Score Selection
Set Initial Condition	Set Print Quality

All other extension control sequences are ignored.
The basic control sequence format is as shown below.

CSP	Code Count-Low Count-High [$Parm_1$ . . . $Parm_n$]
	or
ESC X'5B'	Code Count-Low Count-High [$Parm_1$ . . . $Parm_n$]
X'1B5B'	Code Count-Low Count-High [$Parm_1$ . . . $Parm_n$]

Where:
CSP (Control Sequence Prefix) is a 2-byte value that introduces the control sequence. CSP is assigned ASCII values 27 and 91 (1BH, 5BH), which represent ESC [.

Code is a 1-byte value that designates a specific control function.

Count-Low is a 1-byte value that is the least significant 8 bits of the parameter count bytes in the control sequence function.

Count-High is a 1-byte value that is the most significant 8 bits of the parameter count bytes in the control sequence function.

$Parm_1$ to $Parm_n$ are the parameters that contain the control settings. These parameters can be decimal or hexadecimal in format and each can consist of one or more bytes. Some controls do not have parameters.

1.1 Control Specifications

Format controls include the basic ASCII control codes from the C0 set, the escape sequence functions and the control sequence functions.

1.2 Single-byte Controls

The BJ-300/330 printers support the following single-byte controls.
When Character Set 1 is active, most of the control codes exist from ASCII 80H to 9FH. These control codes are enclosed in parentheses below.
The supported control codes are as follows.

Code	Decimal (*)	Hex (*)	Function
NUL	0, 127 (128)	00, 7F (80)	Null
BEL	7 (135)	07 (87)	Bell
BS	8 (136)	08 (88)	Backspace
HT	9 (137)	09 (89)	Horizontal Tab
LF	10 (138)	0A (8A)	Line Feed
VT	11 (139)	0B (8B)	Vertical Tab
FF	12 (140)	0C (8C)	Form Feed
CR	13 (141)	0D (8D)	Carriage Return
SO	14 (142)	0E (8E)	Shift Out, double-wide on
SI	15 (143)	0F (8F)	Shift In, condensed mode on
DC1	17 (145)	11 (91)	Device Control 1, printer select (XON for serial interface)
DC2	18 (146)	12 (92)	Device Control 2, cancel condensed mode and return to 10 CPI
DC3	19 (147)	13 (93)	Device Control 3, same as NUL (XOFF for serial interface)
DC4	20 (148)	14 (94)	Device Control 4, cancel double-wide if set with SO
CAN	24 (152)	18 (98)	Cancel
SP	32	20	Space
RSP	255	FF	Required Space

(*) The control codes in parentheses can only be used when character set 1 is selected. These characters represent text characters in character set 2.

Note: For PC compatibility, the above control codes from ASCII 00H to 1FH can also be invoked by preceding the control code with ESC (1BH).

2 Escape Sequence

This section describes the control syntax and semantics of the escape sequence functions.
The format of the escape sequence functions is as follows.

ESC F n1...n

"F" can be any one of the following:

ASCII 0 (00H) to 31 (1FH) (same as for single-byte control codes, see the previous section)
ASCII 45 (2DH)
ASCII 48 (30H) to 126 (7EH) (excluding 59 (3BH) and 91 (5BH)
ASCII 51 (5BH) (see section "Control Extensions: Control Sequence" on page 1-23).

If "F" is not one of the above, the escape sequence will terminate without performing any operation.

The BJ-300/330 printers support the following escape sequences.

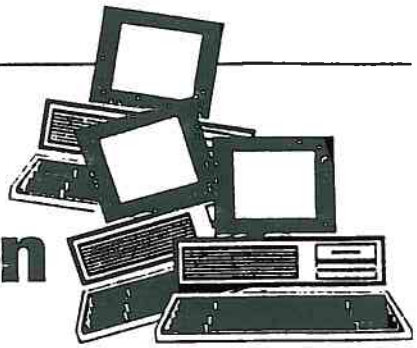
Escape Sequence	ASCII Code Dec	ASCII Code Hex	Function
ESC - n	45	2D	Auto Underscore (1=on, 0=off)
ESC 0	48	30	Set 1/8 Inch Line Spacing
ESC 1	49	31	Set 7/72 Inch Line Spacing
ESC 2	50	32	Invoke Variable Line Spacing
ESC 3 n	51	33	Set Graphics Line Spacing (n/216: changed with ESC [\)
ESC 4	52	34	Set Top of Form
ESC 5 n	53	35	Set LF Auto Mode on CR (1=on, 0=off)
ESC 6	54	36	Select Character Set 2
ESC 7	55	37	Select Character Set 1
ESC :	58	3A	Set 12 CPI
ESC =	61	3D	Character Font Image Download
ESC A n	65	41	Set Text Line Spacing (n/72)
ESC B n...n 0	66	42	Set Vertical Tabs n1...n
ESC C 0 m	67	43	Set Page Length (m inches)
ESC C n	67	43	Set Page Length (n lines)
ESC D n...n 0	68	44	Set Horizontal Tabs n...n
ESC E	69	45	Begin Emphasized Print
ESC F	70	46	End Emphasized Print
ESC G	71	47	Begin Double Strike Print
ESC H	72	48	End Double Strike Print
ESC I n	73	49	Change Font (n selects the font)
ESC J n	74	4A	Relative Move Base Line (n/216: changed with ESC [\)
ESC K n1 n_2 n...n	75	4B	Normal Density Bit Image Graphics (letter quality speed)
ESC L n1 n_2 n...n	76	4C	Dual Density Bit Image Graphics (letter quality speed)
ESC N n	78	4E	Set Skip Over Perforation
ESC O	79	4F	Reset Skip Over Perforation
ESC P n	80	50	Proportional Spacing (1=on, 0=off)
ESC Q n	81	51	Deselect on Positive Query Reply
ESC R	82	52	Set Default Tab Racks (vertical and horizontal)
ESC S n	83	53	Start Subscript (1) or Superscript (0)
ESC T	84	54	End Subscript or Superscript
ESC U n	85	55	Set Print Direction (0=bi-directional, 1= uni-directional)
ESC W n	87	57	Double-wide Continuous Mode (1=on, 0=off)
ESC X n1 n_2	88	58	Set Horizontal Margins (n_1=left margin, n_2=right margin)
ESC Y n1 n_2 n...n	89	59	Dual Density Bit Image Graphics (letter quality speed)
ESC Z n1 n_2 n...n	90	5A	High Density Bit Image Graphics (letter quality speed)
ESC [@ $n_1 n_2 m_1 ... m_4$	91 64	5B 40	Set Presentation Highlight

Escape Sequence	ASCII Code Dec	ASCII Code Hex	Function
ESC [F $n_1 n_2 m_1 m_2 m_3$	91 70	5B 46	Page Presentation Media
ESC [I $n_1 n_2$	91 73	5B 49	Select Font
ESC [K $n_1 n_2$	91 75	5B 4B	Set Initial Condition
ESC [S $n_1 n_2$	91 83	5B 53	Set Vertical Margins
ESC [T $n_1 n_2$	91 84	5B 54	Set Code Page
ESC [\ $n_1 n_2$	91 92	5B 5C	Set Vertical Units
ESC [d $n_1 n_2$	91 100	5B 64	Set Print Quality
ESC [g $n_1 n_2$	91 103	5B 67	High Resolution Graphics
ESC [- $n_1 n_2$	91 45	5B 2D	Score Selection
ESC\ $n_1 n_2$	92	5C	Print All Characters (including below 20H)
ESC ^	94	5E	Print Single Character Under Hex 20
ESC _ n	95	5F	Continuous Overscore Mode (1=on, 0=off)
ESC d $n_1 n_2$	100	64	Relative Move Inline Forward (n/120th inch)
ESC e $n_1 n_2$	101	65	Relative Move Inline Backward (n/120th inch)
ESC j	106	6A	Stop (and go offline)
ESC n m	110	6E	Select Aspect Ratio

Note: The BJ-300/330 printers support the original Canon commands, FS C B and FS C J.

2.5.a

TÆT PÅ PC'EN



Trimning af harddisken

Anvisninger på, hvordan man holder harddisken "velsmurt".

Af Martin Rørbye Angelo

Når man arbejder med sin PC, synes det i de første måneder som om alt går med en svimlende fart. Men snart vænner man sig til det, og inden længe synes alt at tage en forfærdende tid. Hvis ikke man vidste bedre, ville man tro, at ens PC ikke var den samme, som den man købte. Den er meget langsommere.

Der er naturligvis rigtigt, at man vænner sig til hastighed - *performance*, men det er også rigtigt, at ens PC bliver langsommere med tiden. Det er ikke fordi den regner langsommere, men fordi man, hvis man ikke tænker på det, kan benytte sin PC på en sådan måde, at der bliver mere og mere arbejde for PC'en for at udføre de samme opgaver.

Vi vil i denne artikel se på, hvorledes man kan trimme sin PC - dog mest i forbindelse med harddisken. Det tager tid at hente data ind fra harddisken, det tager tid at læse programmerne ind, og det er den værste slags tid, der findes: *ventetid*.

Øget hastighed uden omkostninger

Der er en række ting, man kan gøre for at bringe overførselstiden af data på sin harddisk ned uden at indkøbe nye omkostningskrævende komponenter. Noget kræver dog anvendelse af (ikke særlig dyr) software, men det meste er en række grundlæggende regler for god opførsel overfor sin harddisk.

De 14 faktorer, der bestemmer harddiskens hastighed

Der er 14 grundlæggende faktorer, der bestemmer den hastighed, hvormed harddisk-systemet henter information fra disken til arbejdslageret. De første fire er betinget af det "isenkram", der sidder i PC'en. Isenkrammet kan naturligvis udskiftes, men det koster penge.

De næste to er bestemt af *low level*-formateringen - dog begrænset af "isenkrammet". De sidste otte er kun bestemt af brugeren, og det er derfor her, vi skal finde den gratis hastighedsforøgelse.

Det tager tid at flytte data gennem elektronikken, men den er minimal. Det altafgørende tidsforbrug ligger i læse/skrivehovedets bevægelser. De fleste af de følgende råd og vejledninger omhandler derfor forskellige måder at begrænse læse/skrivehovedets bevægelser på.

CPU-hastighed (1). Højere CPU-hastighed øger hele PC'ens reaktionshastighed og derfor også dataoverførsel fra harddisken. Det kræver dog investeringer i *accelerator-boards* eller i udskiftning af CPU'en.

Hukommelses- og hjælpe-kredsene forbliver de oprindelige (langsomme) versioner, så man får ikke nær så meget ud af at udskifte fx en 80286-CPU med en 80386SX-ditto som ved at købe en 80386SX-maskine. Men det er stadigvæk en glimrende og prisbillig måde at skaffe sig ekstra hastighed på i en lidt ældre PC.

Middelsøgetid (2). Jo kortere middelsøgetid en harddisk har, jo hurtigere bevæger den læse/skrivehovedet, dvs. jo hurtigere finder den de enkelte sektorer. Der er dog intet, der kan ændre middelsøgetiden ved et givet drev. Hvis man ønsker at nedsætte middelsøgetiden, må man købe en ny, hurtigere harddisk.

Data transfer rate (3). En hurtig harddisk kan forsinkes ved langsom overførsel gennem harddisk-controlleren (se PC World nr. 6/92 si. 96f). Ændring af en given PC kræver derfor indkøb af et nyt controller-board. Udskiftning af controlleren fra fx en ST-506/412-type til en RLL-type kan dog øge hastigheden væsentligt, samtidigt med at harddisk-kapaciteten øges - ofte med 50 pct. Således kan en 40 MByte harddisk med ST-506-controller blive til en 60 MByte med RLL-controller - uden anden investering. Men det fordrer en *low level*-omformatering samt at harddisken er konstrueret med tilstrækkeligt gode tolerancer til at kunne lagre den øgede datamængde.

Interleave (4). Når der læses fra harddisken, kan systemet normalt læse data hurtigere end controllerkortet kan modtage og behandle dem. Bl.a. derfor anvendtes tidligere meget *interleave factor*, der er det antal sektorer, der springes over, hver gang der er læst én sektor. Nyere hurtigere maskiner har ikke det problem og anvender derfor ofte interleave 1:1. Ved at optimere interleave, kan den effektive søgetid bringes ned. Dette kan gøres ved fx Norton's funktion Calibrate.

Track buffering (5). Man kan læse et helt *track* (spor) ind i en buffer, selv om PC'ens controller ikke kan håndtere 1:1 interleaving. Da sporet da læses sekventielt, går det væsentlig hurtigere, end hvis læsningen skete i den rækkefølge, interleave-faktoren ellers foreskrev. Track-buffering er imidlertid i reglen en egenskab ved det enkelte controllerkort, hvorfor et nyt må købes, hvis man vil øge hastigheden således.

Antal sektorer pr. cylinder (6). Jo flere plader pr. cylinder der er, jo mindre skal læse/skrivehovedet (rettere: læse/skrivehovederne) bevæge sig under læsning og skrivning. Det betyder, at med en given kapacitet vil en harddisk med flere plader være hurtigere end én med færre, men større plader. Men med en given harddisk kan dette jo ikke ændres. Jo flere sektorer pr. spor, jo flere data kan læses uden at skulle flytte læse/skrivehovedet til næste spor. Vil man øge antal sektorer pr. spor, kræver det dog udskiftning af controllerkortet. ▷

RAM-disk support (7). Ofte læste filer kan lægges i et RAM-lager, derved nedsættes den mængde data, der skal læses fra harddisken.
DOS buffer-definition (8). Såfremt antallet af DOS-buffere er valgt optimalt, nedsættes behovet for at læse de samme filer igen og igen.
Caching (9). En mere avanceret metode til RAM-lagring af ofte benyttede databidder end DOS' "buffers" er data-caching. Caching-programmer leveres fx som standard med DOS 3.3 eller senere
Path-kommandoen (10). Ved omhyggelig (begrænset) anvendelse af DOS' path-kommando, kan søgetiden efter lokalbiblioteker holdes nede.
Fastopen-kommandoen (11). Ved at køre *fastopen*, vil DOS huske, hvor de sidst anvendte lokalbiblioteker er placeret og behøver derfor ikke at lede efter dem, men kan gå direkte til dem. Det er en ny kommando i DOS 3.3.
Design af træstrukturen (12). Ved at udforme sit bibliotekstræ med omtanke, kan man holde sin harddisk på samme hastighed, som da den var ny.
Lokalbibliotekslayout (13). Ved at sikre sig, at lokalbibliotekerne er lagt i en eller få cylindre, kan harddiskens hastighed ligeledes holdes på samme niveau, som da den var ny.
Fil-fragmentering (14). Jo mindre fragmenteret en fil er, jo færre cylindre den er spredt over, jo mindre bevægelse skal læse/skrivehovedet gennemføre for at læse den.

Hvorledes øger jeg min harddisk performance?

Af disse 14 faktorer kan man måske anvende de ti sidste på en eksisterende PC uden at skulle investere i nyt isenkram, og det er disse ti faktorer, vi vil behandle.
Alle reglerne i de følgende tre afsnit går derfor ud på ét eneste: at formindske antallet af mekaniske bevægelser, som læse/skrivehovedet skal foretage.

Optimering på low-level formateringsniveau

Når den "blanke" harddisk formate-

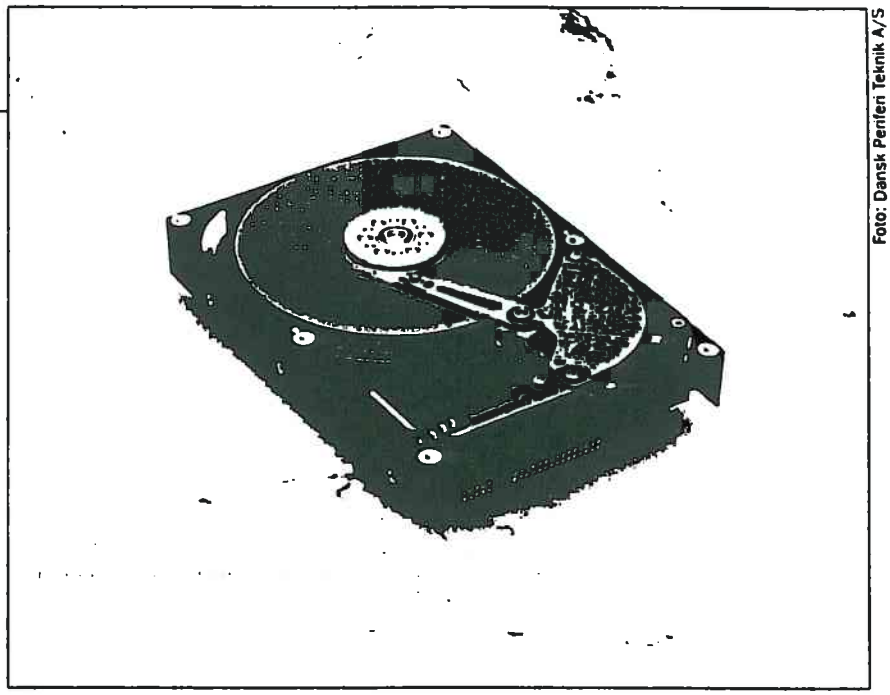


Foto: Dansk Periferi Teknik A/S

Harddiskene bliver mere og mere kompakte. Denne Maxtor MXT har en kapacitet på 1,2 Gb og en gennemsnitlig søgetid på 8 ms.

res til anvendelse under DOS-styresystemet, vælges en række parametre, der har væsentlig indflydelse på harddiskens tidsforbrug. Hvis hastighedsforøgelse er et krav eller hvis parametrene simpelt hen er valgt forkert fra starten, kan man forsøge sig med en ny low level-formatering.

Interleave

Sektorer, skrevet i rækkefølge på harddisken, kommer ikke til at ligge i rækkefølge, fordi systemet "fletter" disse sektorer for at give tid til at behandle en læst sektor, inden den næste sektor læses. Dette kaldes interleave. En interleave-faktor på 6:1 betyder, at der er fem "fremmede" sektorer mellem en bestemt sektor og den næste sektor, der skal læses, når hele den pågældende fil læses.

Såfremt faktoren er rigtigt valgt, betyder det, at den pågældende PC anvender samme tid til at behandle den læste sektor, som det tager harddisken at rotere en vinkel svarende til knapt fem sektorer og derfor er klar til at læse, når sektor nr. seks kommer ind under læse/skrivehovedet.

Såfremt interleave-faktoren er valgt for stor, skal systemet altså vente længere end nødvendigt. Såfremt interleave-faktoren er valgt for lille, skal harddisken dreje en hel omgang, før næste sektor læses med tilhørende tidsspild.

Hvis man har mistanke om, at harddiskens interleave-faktor er sat forkert, må man først konsultere den dokumentation, man fik med eller konsultere sin forhandler. En XT havde som standard 6:1, mens en AT har 3:1. Der er i dag så mange forskellige kombinationer af processorer og hurtige eller langsomme harddiske, at det ikke er til at sige, hvilken interleave-faktor man kan forvente en nyleveret PC har. Grundet den stadig større andel af hurtige harddiske og controllers, er andelen af PC'er med interleave 1:1 stadig stigende.

Vil man eksperimentere, må man low level-reformatere harddisken, high level (DOS) -formatere og så køre en hastighedstest (som Benchmark e.l.) for at se, om man øgede hastigheden.

Det er et næsten uoverkommeligt arbejde. Der findes programmer til optimering af interleave, der samtidig vil reformatere harddisken til den optimale interleave, såfremt det ønskes - uden at slette indlæste data. Den reformaterer én cylinder ad gangen og gemmer data imens i arbejdslageret. Et sådant program er fx Norton Utilities 5.0 eller senere, der indeholder et program kaldet *calibrate* som gør netop dette. Programmet skal dog genkende formateringen for at kunne gennemføre analysen og reformateringen, så det er ikke alle PC'er, der kan anvende dette eller tilsvarende programmer; men nok de fleste.

Som hovedregel er der kun væsentlige hastighedsforøgelser at hente her, hvis interleave-faktoren er sat for lavt. Hvis den er sat en eller to sektorer for høj, er det marginalt, hvad der indtjenes ved en ændring. Husk at tage backup, før disse øvelser begyndes.

Optimering i DOS-miljøet

I det følgende vil vi se på programmer og kommandoer, der under DOS-styresystemet vil kunne øge den hastighed, hvormed harddisken øjensynligt opererer.

Når vi siger øjensynligt, er det fordi disse hjælpemidler mindsker behovet for at læse fra og skrive til harddisken eller mindsker søgetiden efter et bestemt bibliotek. Den egentlige læse- og skrivetid ændres ikke herved.

RAM-disksupport

En RAM-disk er en del af PC'ens almindelig elektroniske arbejdslager, der er afsat til at opbevare data i en sådan form, at det for brugeren ligner en disk.

RAM-diske blev mere almindelige, da 80286-baserede PC'er kom frem, idet 80286 kan adressere 16 MByte, mens 8088 kun kan adressere 1 MByte. DOS, der oprindeligt var skabt til 8086 eller 8088, kan heller ikke adressere mere end den ene MByte, hvorimod 80286 har mulighed for at adressere de 15 MByte, der ligger udenfor DOS' rækkevidde.

Det er det, der kaldes *extended memory* - engelsk for udvidet arbejdslager. Den er extended (forlænget) ud over DOS' rækkevidde. Fra DOS 3.0 findes mulighed for at sætte en RAM-disk op i denne extended memory, kaldet *vdisk* (IBM DOS) for virtual disk eller *ramdisk* (Microsoft DOS).

Herudover kan en PC forsynes med hukommelse, der ikke direkte kan adresseres af CPU'en, men gennem et 64 KByte"-vindue", der ligger inden for den første ene MByte, og derfor kan ses af såvel 8088 som 80286. Denne form for hukommelse kaldes *expanded memory* - engelsk for ekstra arbejdslager. Kommunikationen gennem dette 64 KByte-vindue varetages af et hukommelsesstyringsprogram - ▷

TÆT PÅ PC'EN

ofte refereret til som EMMS (*Expanded Memory Manager System*).

Da denne form for hukommelse er tilgængelig via DOS, kan den derfor anvendes direkte af de fleste programmer, som almindelige PC-brugere anvender.

Også i Expanded Memory kan man sætte en RAM-disk op, det gøres dog ikke af DOS, men derimod af det pågældende EMMS, men for brugeren er de to systemer ikke til at skelne fra hinanden. Begge de to typer RAM-diske mister deres information, når der slukkes for PC'en, hvorfor de kun kan anvendes til program- og datastumper, der skal lagres midlertidigt under programafviklingen. Således opretter nogle en RAM-disk alene for at kunne lægge DOS' eksterne kommandoer herop, hvorved afviklingen af DOS-kommandoer bliver væsentlig hurtigere.

RAM-diske er derfor bedst egnet til lagring af dele af programmer eller hele programmer, mens de afvikles. Således vil regnskabsprogrammet PCPlus afvikles meget hurtigere fra RAM-disk, fordi programmet er opbygget på en sådan måde, at det ustandseligt skal ud på harddisken og hente nye dele af programmet.

Ved at lægge hele programmet og tilhørende data op i en 3 MByte RAM-disk, benyttes harddisken ikke under hele brugen, og det batch-program, der lægger program og data op i RAM-disken (når programmet startes), sørger også automatisk for at lægge data tilbage på harddisken ved afslutning.

Batch-programmet lægger ligeledes en kopi af data på et andet diskdrev ved afslutning af brugen, hvorved egentlig backup kan foretages relativt sjældent, idet headcrash på to forskellige harddiske er så usædvanligt, at man kan roligt kan se bort fra det.

Visse tekstbehandlingsprogrammer giver brugeren mulighed for at specificere, hvorledes programmet skal finde den midlertidige hukommelse, det har brug for under editering (kaldet fx Virtual Memory Setup). Default er ofte harddiskens rodbibliotek, mens det vil spare en del harddisk-læse- og skriveaktivitet, hvis en RAM-disk specificeres.

DOS buffer-definition

DOS har faktisk selv et system til minimering af harddisk (eller floppydisk) læsninger. Det er *buffers=nn*, i config.sys, hvor nn er et tal fra 1 til 99. Tallet angiver, hvor mange sektorer på 512 Byte, der gemmes i hukommelsen og derfor ikke behøver blive læst fra disken, når de skal anvendes igen. Tilsvarende vil dette antal sektorer ikke blive skrevet på disken, for buffer-pladsen skal benyttes af andre data eller en bestemt tid er forløbet uden diskaktivitet.

Herved undgås at en sektor, hvor der måske kun rettes nogle få Byte ad gangen, skrives på disken, hver eneste gang en rettelse finder sted; men først skrives, når programmet kalder nye sektorer ind.

Hvis kommandoen buffers= ikke anvendes, sætter XT'er den til 2 og AT'er den til 3. Dette afspejler hukommelsens begrænsning fra PC'ens barndom, hvor de var født med 64 KByte alt iberegnet.

I dag, hvor man næppe kan købe en PC med under et par MByte, er buffers=15 eller buffers=20 et mere optimalt valg. Hver buffer fylder 528 Byte - 512 Byte til hukommelsen og 16 Byte til at holde styr på den med.

Caching

Det bedste ved RAM-diske og DOS-buffers kombineres i *disk-caching*. Når et program giver besked om at læse en programdel eller datadel fra harddisken, søges i caching-området (i arbejdslageret) først. Disk-caching kan holde MegaByte af program og data i hukommelsen, såfremt extended eller expanded memory benyttes.

Forskellen mellem DOS-buffere og disk-cache er blot, at da disk-cache opbevarer hele programblokke og

ikke kun sektorer, er søgningen væsentlig hurtigere og mere logisk.

Derfor forsinker en disk-cache på 1 MByte ikke søgningen efter en bestemt programdel væsentligt, mens en tilsvarende DOS-buffer dels ikke kan lade sig gøre (maksimum er buffers=99 svarende til ca. 50 KByte) og dels ville tage evigheder, da buffere søger sektor for sektor.

Derudover vil DOS-buffere forbruge af den maksimale 1 MByte, som DOS kan adressere, hvoraf vi som brugere kun kan anvende de 640 KByte, hvorimod en disk-cache kan lægges i såvel extended som expanded memory.

Eksempelvis leveres IBM-cache med nogle af PS/2-systemerne. Det ligger som en skjult fil på systemdisketten og kan sætte en disk cache på 16-512 KByte inden for DOS-hukommelsen (system memory) eller fra 16 KByte-15 MByte i extended memory (IBM-cache kan ikke håndtere expanded memory).

Cache-programmer leveres med andre hjælpeprogrammer som Windows og Norton Utilities. I DOS 5.0 introduceredes en cache kaldet smartdrv.sys. Man kan sætte cache-hukommelsen så stor man vil, men på et tidspunkt bliver den tid, der skal til at søge gennem cache-hukommelsen, hver gang man søger en information på harddisken, så stor, at den samlede hastighed falder. En værdi over 256 KByte, men under 1 MByte, placeret i extended hukommelse synes at være et godt startpunkt.

Path-kommandoen

Praktisk taget alle, der anvender harddisk, kender DOS-kommandoen *path*, der beder DOS om at lede i en række specificerede biblioteker, såfremt en kommando ikke findes i det bibliotek, kommandoen indtastes fra.

De fleste opbygger en under tiden særdeles kompliceret path (søgesti) i logisk rækkefølge: Først roden i C:, dernæst lokalbiblioteker - ofte i alfabetisk orden - ned ad én streng, ned ad den næste osv. Dernæst disk D:, hvis flere logiske drev anvendes, osv. De færreste tænker på, at DOS rent faktisk søger i samme orden. Så søgestien bør opbygges således, at de mest anvendte lokalbiblioteker ligger først fx den, der indeholder command.com og DOS' eksterne kommandoer. Sidst i søgestien kommer så de lokalbiblioteker, der indeholder "elegante", men sjældent brugte batch-filer. Søgestien må i øvrigt højst være 127 tegn lang.

Fastopen-kommandoen

Fastopen-kommandoen er en nyskabelse i DOS 3.3. Ved enten direkte eller via autoexec.bat at kalde kommandoen fastopen c: = nnn, vil DOS huske de sidste nnn-klyngenumre på drev c:. Første gang man beder om at læse filen C:bib1\bib2\bib3\bib4\-fil.dat, må DOS lede gennem hele denne path. Næste gang ved den, hvor på harddisken filen fil.dat ligger og kan derfor gå direkte til den første klynge og sparer derved en del søgetid.

I Fastopen er nnn et tal fra 10 til 999. Hvis Fastopen eksekveres uden angivelse af nnn, sætter DOS nnn = 034.

Bibliotekets træstruktur

Når DOS søger en fil, må den springe fra lokalbibliotek til lokalbibliotek i sin søgen ned igennem træet. Derfor kan følgende "leveregler" udledes: Først må filer, der ikke anvendes ret meget, lægges langt nede i træstrukturen, mens filer, der adresseres ofte fx data, bør ligge nærmere roden. Ofte lægges programmer i første niveau under roden, og data lægges et eller to niveauer herunder. Man burde, for at minimere søgetiden, gøre det modsatte.

Af hensyn til forenklet backup, kan det være en fordel at have sine programmer på fx logisk drev E: og alle data på logisk drev D:, idet logisk drev C: reserveres til systemfiler såsom DOS og hjælpeprogrammer, der ikke bruges dagligt. Herved mindskes risikoen for, at et vildtløbende program kan ændre noget og umuliggøre at harddisken boot'er.

Da data ikke fylder ret meget, såfremt, der ikke er tale om grafik-filer, kan mange klare sig med et mindre drev til data og et noget større til programmer.

Tekstbehandlingsprogrammer kan fx godt i dag fylde over 5 MByte, mens datafiler hertil næppe kommer op over nogle få MByte for de flestes vedkommende. Derfor foreslås det, at programmet lægges på fx e:\txtprg\.., mens data lægges på d:\txtdata\..

Da man ikke behøver at lave backup af programmerne, man har jo originaldisketterne, kan man nøjes med den simple kommando: lav backup af D:.

Dernæst må størrelsen på lokalbiblioteket holdes nede. Et bibliotek optager 32 Byte på harddisken. En 512 Byte-sektor kan derfor indeholde 16 biblioteker. En standard formateret harddisk under DOS 3.0 eller senere har klynger (harddiskens mindste adresserbare enhed) på fire sektorer. En klynge kan altså indeholde op til 64 navne på filer eller lokalbiblioteker i én klynge. To af disse går til DOS' egen information, kendt som "." og "..", så for at sikre sig, at alle navne i et bibliotek holdes inden for én klynge, bør man ikke have mere end 62 navne i sit bibliotek.

Ved at holde alle navne i én klynge, sikrer man sig, at læse/skrivehovedet ikke skal flyttes til en anden cylinder under søgningen og sparer derved søgetid. Sorteres bibliotekerne regelmæssigt, og holdes bibliotekerne først på harddisken - før program og datafiler, opnås optimal hastighed. Denne sortering kan foretages meget enkelt og hurtigt med hjælpeprogrammer som Norton Utilities. Dets Speed Disk-program spørger i hvilken rækkefølge, man vil have sine filer lagt efter defragmenteringen.

Endelig, som sidste regel: Hold antallet af niveauer så langt nede som muligt. Helst kun to eller tre niveauer. Hvis mange data skal hentes fem eller syv niveauer nede, kan læse/skrivehovedets søgetid alene løbe op i sekunder.

Ved at holde data og programmer adskilt på hver sit logiske drev, spares ofte et par niveauer, hvor man ellers ville fristes til at lægge databiblioteket som et underbibliotek under selve programmet.

Hvordan lægges lokalbiblioteker ind?

Hvis lokalbibliotekerne lægges fysisk samlet på harddisken - i kun et par cylindre langs harddiskens ydre kant ▷

- kan læse/skrivehovedets søgetid reduceres meget. Det betyder, at søgning fra lokalbibliotek til lokalbibliotek kan foregå inden for samme cylinder eller nogle få nabo-cylindre, hvilket mindsker den afstand, læse/skrivehovedet skal bevæge sig.

Man skal derfor skabe hele sit bibliotekstræ - just efter at harddisken er formateret - inden, der lægges data eller programmer ind. Derved lægges alle biblioteker automatisk i harddiskens kant, og lokalbiblioteker lægges først i bibliotekets navneliste. Når bibliotekstræet skabes, skal man derfor først oprette alle lokalbiblioteker i én streng og dernæst oprette hele den næste path.

Det er ikke den måde, man normalt tænker på. Ofte vil man først oprette sine hovedbiblioteker som fx c:\tekst, c:\regneark, c:\database, c:\regnskab, c:\pascal, c:\utility osv. for dernæst at oprette de lokalbiblioteker, der er nødvendige i hvert bibliotek. Vil man maksimere sin harddisk-hastighed, er det ikke vejen frem.

Fil-fragmentering - undgå at sprede filer for meget

Fragmentering og defragmentering er nok den del af harddisk-optimeringen ,de fleste har stiftet bekendtskab med gennem programmer som Nortons SpeedDisk og lignende.

Når data skrives på en nyformateret harddisk, lægges de i klynger, der ligger lige efter hinanden (eller i rækkefølge svarende til interleave-faktoren). Det næste sæt data lægges i klynger lige bag det første sæt.

Når nu det første sæt data vokser, lægges de overskydende data bag de nu skrevne data, dvs. at det første sæt data ikke længere ligger i samlet rækkefølge. Efterhånden som harddisken bruges, og data skrives og slettes, filer vokser og andre forsvinder, bliver det fænomen, at filer ikke ligger pænt i rækkefølge, mere og mere udbredt. Man taler om, at filerne er brudt op i stumper eller med et nudansk ord: fragmenteret.

Det siger sig selv, at jo flere cylindre en fil er spredt over, jo større læse- og/eller skrivetid er nødvendigt. Især for filer, der indlæses på én gang, er tidsforøgelsen åbenbar. Det gælder fx regneark, tekst-filer og programmer. Hvor stumper af filer hentes af og til som fx databaser, føles det derimod ikke så meget.

Fragmentering er dog den mest betydende faktor i forringet harddisk-hastighed. Defragmentering bør foretages regelmæssigt, hvis man vil holde sin harddisk "afkokset".

Ud over at holde harddisk-hastigheden oppe, opnås en større sandsynlighed for at kunne redde filer, der ved en fejltagelse er blevet slettet.

Diverse *unerase*-programmer virker perfekt på absolut ufragmenterede filer, så alene af den grund bør man holde sin harddisk opryddet.

Der er principielt to forskellige måder at defragmentere sin harddisk på: Den første er at lave backup til disketter eller en anden harddisk ved simpelt hen at anvende copy *.* eller backup.

Dernæst kan man evt. reformatere harddisken, hvilket dog ikke er strengt nødvendigt og så genindlæse filerne med copy eller restore afhængigt af, hvordan backup'en blev lavet. Det er en temmelig besværlig måde, men den kræver ikke andet end DOS og disketter.

Det kan dog gøres væsentlig lettere, hvis man råder over to PC'er med tilstrækkelig harddisk-kapacitet og overfører data fra den ene til den anden med FastLynx eller LapLink e.l. hjælpeprogrammer.

Den hurtige, enkle og "rigtige" måde er at anvende et af de mange hjælpeprogrammer, der har defragmenteringsprogrammer. Et eksempel er Norton Utilities Speed Disk-funktion.

Hvornår dette skal gøres, kan DOS-kommandoen chkdsk give en ide om. Ved at checke en af sine mest brugte store filer, gives bl.a. antallet af blokke denne er brudt i; fx med kommandoen chkdsk *c:\tekst\tolstoi.txt*.

Man kan ikke gøre ret meget for at hindre eller mindske fil-fragmentering. Arbejder man med DOS 2.x eller tidligere, er sandsynligheden for fragmentering væsentlig større end ved DOS 3.0 og senere. Årsagen er, at de tidligere versioner af DOS fylder disken op udefra og derfor skriver i de første klynger, der er ledige, også selv om det er en løstliggende klynge fra en enkelt lille fil, der er slettet, som kun giver én ledig klynge. DOS 3.0 og senere arbejder sig ind mod diskens midte, og først når disken er fyldt med sammenhængende filer, udnytter DOS 3.3 de mellemliggende klynger.

Man kan med andre ord mindske (eller måske undgå) fragmentering ved at anvende DOS 3.0 eller senere, og sikre sig rimelig reserveplads på sin harddisk.

Arbejder man med en 40 MByte harddisk og har 1 MByte ledig kapacitet, kan man være sikker på en voldsom fragmentering. Har man derimod 20 MByte ledig, er man rimelig sikret mod fragmentering.

Hvis man sletter gamle filer for at skaffe sig denne rimelige plads på harddisken, må man dog defragmentere harddisken, før man skriver på harddisken igen ellers vil de tilfældige huller fra de slettede filer blot blive fyldt med kraftig fragmentering til følge.

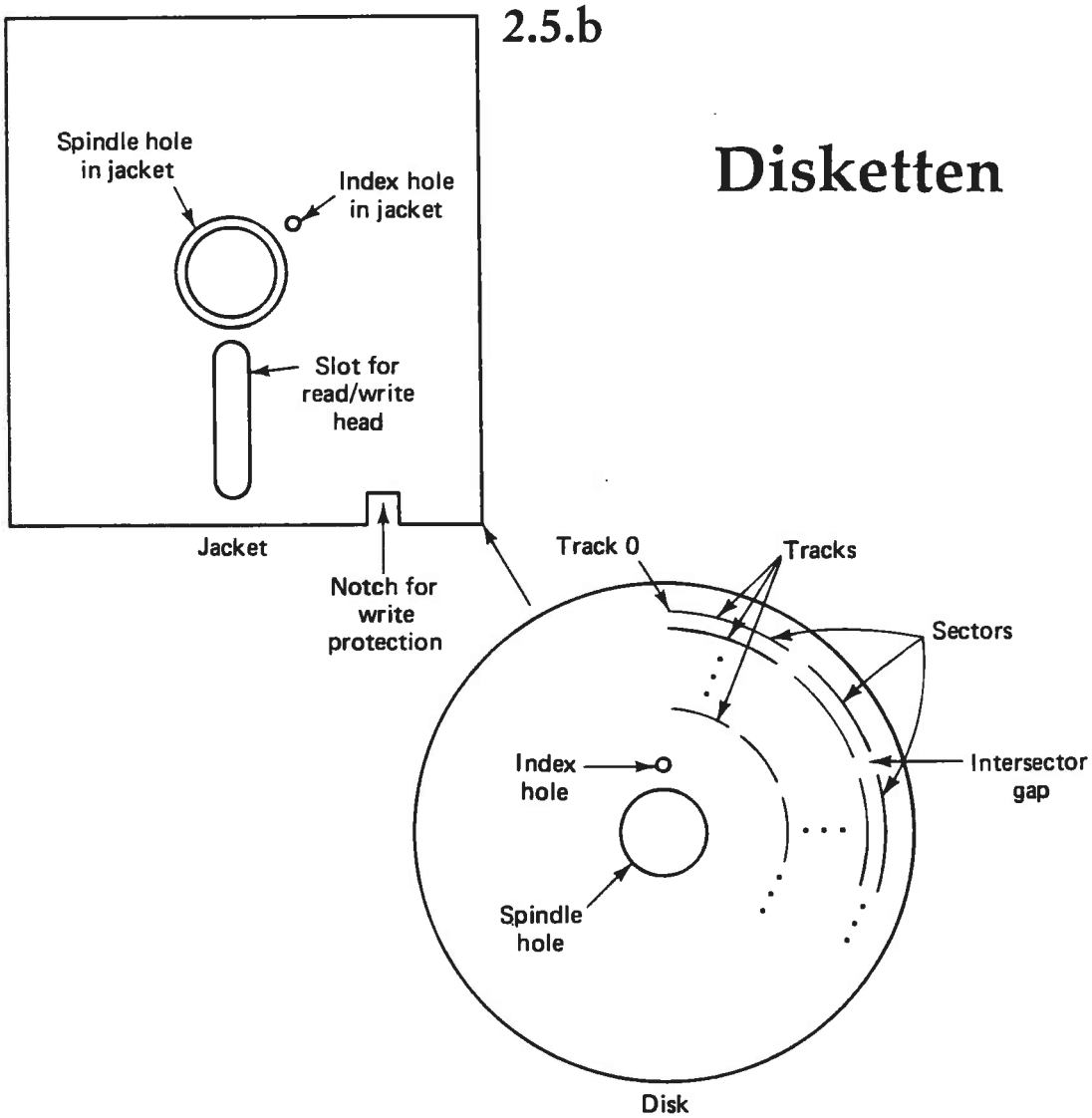
Afsluttende bemærkninger

Mange af ovenstående anvisninger kan ikke følges 100 pct. De er mest ment som en art rettesnor. Ved at henlede opmærksomheden på en række fænomener, man i almindelighed ikke går og tænker på, kan man komme langt uden de store tiltag. Og egentlig er det vel sjovere at få sin "gamle" PC til at virke fornuftig og kvik, end at ofre stakkevis af kroner på en ny.

□

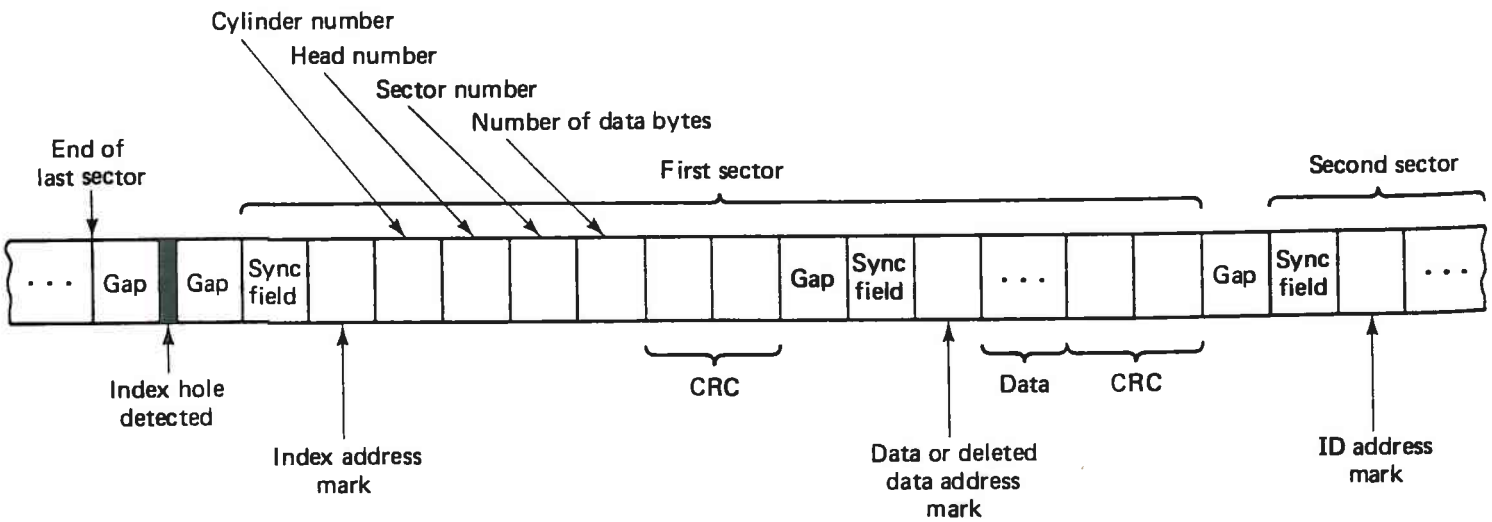
2.5.b

Disketten



Construction of a diskette.

Track and sector format for a soft-sectored diskette.



2.5.c

Harddisk og diskette under dos
DOS-filsystem

Bjørk Busch

Denne lille note har til hensigt, at udbygge forklaringen til hvorledes disketter og harddiske anvendes under DOS.

Princip for sektornummereringen under BIOS og DOS, eksemplificeret på en 5,25" 360KB (DSDD) diskette.

Bios nummerering af sektorer:
Afhængig af fysisk medie.
Opdelt i spor, side, sektor (3. dim. tabel)

Spor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
Side	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
Sektor	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	1	2
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Dos sektor-nummerering:
Uafhængig af fysisk medie.
Fortløbende med start i 0 (1. dim. tabel)

Princip for opdeling af diskette i områder, eksemplificeret ved en 5,25" 360KB (DSDD) diskette.

Brug af dos sektorer (fysisk pladsering):

0	1-2	3-4	5-11	12- ... (dec.)
BOOT	FAT1	FAT2	ROOT	DATA (filer/underkat.)

Den samme opdeling findes for andre disketter og for et logisk drev på en harddisk (eks C:), idet størrelsen af de 2 FAT-tabeller og ROOT kan variere, og dermed give andre sektor-numre.

BOOT-sektoren indeholde et opstarts-program, samt informationer om hvorledes den er formaterer og hermed opbygget.

På en data-diskette vil BOOT-programmet normalt blot udskrive en meddelelse om, at disketten/det logiske drev ikke indeholder systemfilerne (DOS-kernen). På en system-diskette/-drev vil programmet opstarte den første del af DOS systemet.

Strukturen opbygges med programmet FORMAT.

Opdeling af harddisk under DOS:

Partitions tabel	Primær DOS partition	Extended DOS partition

Partitionstabellen er placeret på harddiskens første spor, første side, første sektor. Her ligger også et lille opstarts program, men det er forskelligt fra DOS-BOOT programmet.

Partitionstabellen beskriver opdelingen af harddisken i områder, også kaldet partitioner. Det er herved også muligt at placere partitionerne i en anden rækkefølge fysisk på harddisken.

Den primære DOS partition bliver til drev C: og er opbygget på samme måde som disketten med DOS-BOOT sektor, FAT tabeller m.m.
De fysiske (BIOS) sektor adresser er selfølgelig anderledes.

Der behøver ikke være en Extended DOS-partition, idet den primære kan optage hele pladsen, eller der kan være ledig plads til evt. andre systemer.

Opdeling af Extended DOS-partion på harddisk:

Ext.Part. tabel	Logisk DOS Drev	Logisk DOS Drev

Den Extendede partition indeholder en ny partitionstabel, der beskriver hvorledes dette område er opdelt yderlig.
Opdelingen sker her i logiske drev, og hvert af disse er igen opbygget på samme måde som disketten, med DOS-BOOT sektor, FAT tabeller m.m.

Opdelingen af harddisken i partitioner kan ske med programmet FDISK.

Logisk sammenhæng mellem FAT, ROOT og DATA arealer:

KATALOG (ROOT)

Filnavn	Start
FIL1	002
FIL2	006
FIL3	005
UKAT	00A

FAT

000	
001	
002	003
003	004
004	FFF
005	007
006	FFF
007	008
008	FFF
009	000
00A	00B
00B	FFF
00C	000
00D	FF7
00E	000

2 første elm. i FAT bruges ikke normalt.
1. byte = mediedescritor

DATA CLUSTRE

		DOS Sekt. (dec.)
002	FIL1's 1. cluster	12,13
003	FIL1's 2. cluster	14,15
004	FIL1's sidste cluster	16,17
005	FIL3's 1. cluster	18,19
006	FIL2's eneste cluster	20,21
007	FIL3's 2. cluster	22,23
008	FIL3's sidste cluster	24,25
009	ledig	26,27
00A	UKAT's 1. cluster	28,29
00B	UKAT's sidste cluster	30,31
00C	ledig	32,33
00D	Fejlmarkeret BAD sekt.	34,35
00E	ledig	36,37

Her 2 sektor pr. cluster

UKAT er et underkatalog. Det administreres som en alm. fil med hensyn til pladsering.

Underkataloger er opbygget på samme måde som hovedkataloget (ROOT), idet der dog er følgende forskelle:

- Der optræder ingen LABEL i underkataloger.
- Der optræder altid 2 filnavne i starten af underkataloger:
  - . Der reffererer til underkataloget selv.
  - .. Der reffererer til kataloget lige over.

  Ved refference til ROOT-kataloget, der jo ikke ligger i nogen af data-clustrene, står der 0 i startadressen.

Der kan opstå fejl i FAT-tabeller og kataloger i spec. situationer. Der kan herved f.eks opstå "kæder" i FAT, uden tilhørende fil. Dos kommandoen CHKDSK kan blandt andet anvendes til kontrol af sådanne ting.

FIL1's fysiske pladsering (Sammenhængende):

Start -> 002 -> 003 -> 004 (stop)

Dos sektor (dec.)

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
XX	XX	XX	XX	XX	XX														
2		3		4		5		6		7		8		9		A		B	

Clusternr (hex.)

FIL2's fysiske pladsering (Sammenhængende):

Start -> 007 (stop)

Dos sektor (dec.)

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
										XX	XX								
2		3		4		5		6		7		8		9		A		B	

Clusternr (hex.)

FIL3's fysiske pladsering (Spredt):

Start -> 005 -> 006 -> 008 (stop)

Dos sektor (dec.)

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
						XX	XX	XX	XX			XX	XX						
2		3		4		5		6		7		8		9		A		B	

Clusternr (hex.)

UKAT's fysiske pladsering (Sammenhængende):

Start -> 00A -> 00B (stop)

Dos sektor (dec.)

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
																XX	XX	XX	XX
2		3		4		5		6		7		8		9		A		B	

Clusternr (hex.)

Opbygning af FAT-tabel.

Start af FAT-TABEL, som den ser ude i et dump:

0000: FD FF FF 03 40 00 FF 7F 00 FF 0F 08 FF 0F 00

Mediekode: FD svarende til 360KB, 40 spor, 2 side og 9 sektorer. Koden efterfølges af FF FF, som ikke anvendes til noget.

I DOS versioner før 3.30 blev data fra BOOT sektoren ikke anvendt for disketter, men formatet blev allene defineret ud fra 1. byte i FAT-tabellen.

Omsætning til 12 bits FAT tabel:

1. Det ønskede clusternr gamges med 1,5

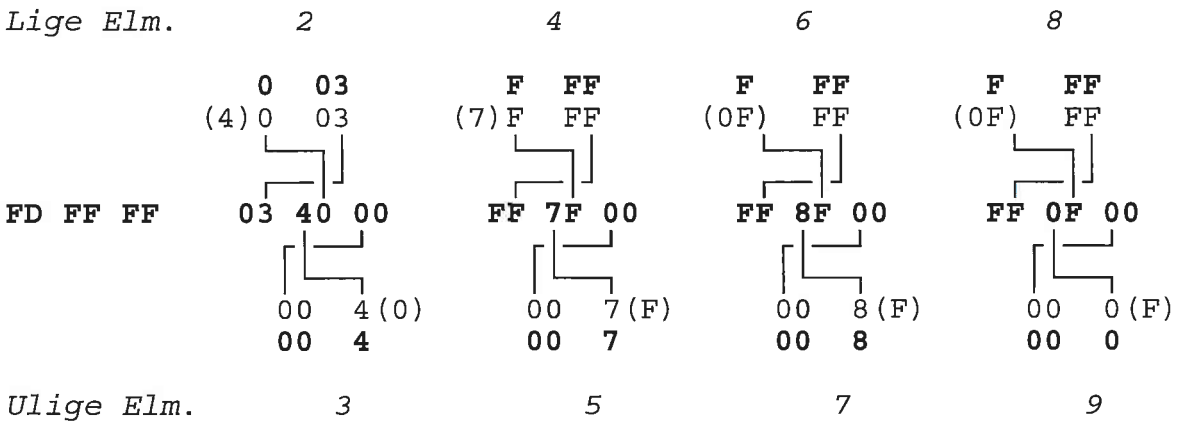
2. Der rundes ned, og man har så offset relativt til tabelstart.

3. Der hentes 2 bytes fra denne adresse (den beregnede og den efterfølgende.
   De 2 bytes ombyttes som normalt for word's.

4. For lige clustre anvendes de sidste 12 af de 16 bits, og for ulige clustre anvendes de første 12 af de 16 bits.

Hvis man fortolker FAT-tabellen fra star af, kan man tage 3 bytes af gangen, der så svarer til et lige og et ulige cluster.

a) De første 2 bytes af de 3 ombyttes og venstre halve bytes smides bort (*Left=Lige*) hvorved man får det lige cluster.
b) De sidste 2 bytes af de 3 ombyttes og højre halve bytes smides bort hvorved man får det ulige cluster.



Ved 16 bits FAT tabeller er der 2 byte pr. cluster, her springes de første 4 bytes over svarende til cluster 0 og 1.
FAT elementerne er alm. word så der skal stadig fortages alm. ombytning.

Opbygning af katalog-tabeller.

Hvert element i katalog-tabellen er på 32 byte, svarende til 2 linier i dumpet. Disse 32 bytes indeholder følgende:

Indhold	position	antal bytes
Filnavn	00 - 07	8
Filextention	08 - 0A	3
Attribute	0B - 0B	1
Reserveret	0C - 15	10
Tidspunkt	16 - 17	2
Dato	18 - 19	2
Clusternr	1A - 1B	2
Filstørrelse	1C - 1F	4

Attribute byten's 8 bit har følgende betydning:

Betydning / bit	7	6	5	4	3	2	1	0
Read only	.	.	.	.	.	.	.	1
Hidden file	.	.	.	.	.	.	1	.
System	.	.	.	.	.	1	.	.
Volume label	.	.	.	.	1	.	.	.
Subdirectory	.	.	.	1	.	.	.	.
Archive	.	.	1	.	.	.	.	.
Ubrugt	.	1	.	.	.	.	.	.
Ubrugt	1	.	.	.	.	.	.	.

Klokkeslettet er opdelt på følgende måde:

Bit 0 - 4	Sekunder/2 (0-29). Gang med 2 for sek.
Bit 5 - 10	Minutter (0-59)
Bit 11 - 15	Timer (0-23)

Bytes ligger som sædvanlig omvendt i lager og skal ombyttes.

Dato er opdelt på følgende måde:

Bit 0 - 4	Dag (1-31)
Bit 5 - 8	Måned (1-12)
Bit 9 - 15	Årstal - 1980. Adder 1980 til for år.

Bytes ligger som sædvanlig omvendt i lager og skal ombyttes.

Når en fil slettes ændres 1. bogstav i filnavn til **E5** og plads frigives i FAT-tabel. Pladsen i kataloget kan herefter genbruges. Nye filer indlægges på første ledige plads i kataloget.

Eksempel på FIL-element i ROOT og fortolkning af dette:

1. ellement i ROOT katalog, som det ser ud i dump.

```
0000:  4C 41 42 50 43 44 20 20 - 20 20 20 28 00 00 00 00
0020:  00 00 00 00 00 00 61 80 - 12 17 00 00 00 00 00 00
```

Fortolkning:

```
Filnavn:        4C 41 42 50 43 44 20 20 => LABPCD

Extention:      20 20 20                => (blank)

Attribute:      28                      => ARC + LABEL

Reserveret:     00 00 00 00 00 00 00 00 00 00

Tidspunkt:      61 80     => 80 61  =  1000 0000 0110 0001
                                       TTTT TMMM MMMS SSSS

                Time:     10000         => 16

                Min:      000011        => 3

                Sek:      00001 => 1*2  => 2

Dato:           12 17     => 17 12  = 0001 0111 0001 0010
                                      ÅÅÅÅ ÅÅÅM MMMD DDDD

                År:       0001011 => 11+1980  => 1991

                Måned:    1000          => 8

                Dag:      10010         => 18

Startcluster:   00 00

Filstørrelse:   00 00 00 00        En label optager
                                   kun plads i ROOT
```

2. ellement i ROOT katalog, som det ser ud i dump.

```
0030:  46 73 4C 32 20 20 20 20 - 54 58 54 21 00 00 00 00
0040:  00 00 00 00 00 00 20 17 - 5D 15 02 00 05 0A 00 00
```

Fortolkning:

```
Filnavn:        46 73 4C 32 20 20 20 20 => FIL1

Extention:      54 58 54                => TXT

Attribute:      21                      => ARC + READONLY

Reserveret:     00 00 00 00 00 00 00 00 00 00

Tidspunkt:      20 10     => 10 20  =  0001 0000 0010 0111
                                       TTTT TMMM MMMS SSSS

                Time:     00010         => 2

                Min:      000001        => 1

                Sek:      00111 => 7*2  => 14

Dato:           5D 15     => 15 5D  = 0001 0101 0101 1101
                                      ÅÅÅÅ ÅÅÅM MMMD DDDD

                År:       0001011 => 10+1980  => 1990

                Måned:    1010          => 10

                Dag:      11101         => 29

Startcluster:   02 00                   => 00 02

Filstørrelse:   05 0A 00 00 => 00 00 0A 05   => 2565
                                             Sidste cluster
                                             er ikke fyldt op.
```

De enkelte bytes i binære tal er lagt "baglæns" ud i lager.

Beskrivelse af boot-sector:

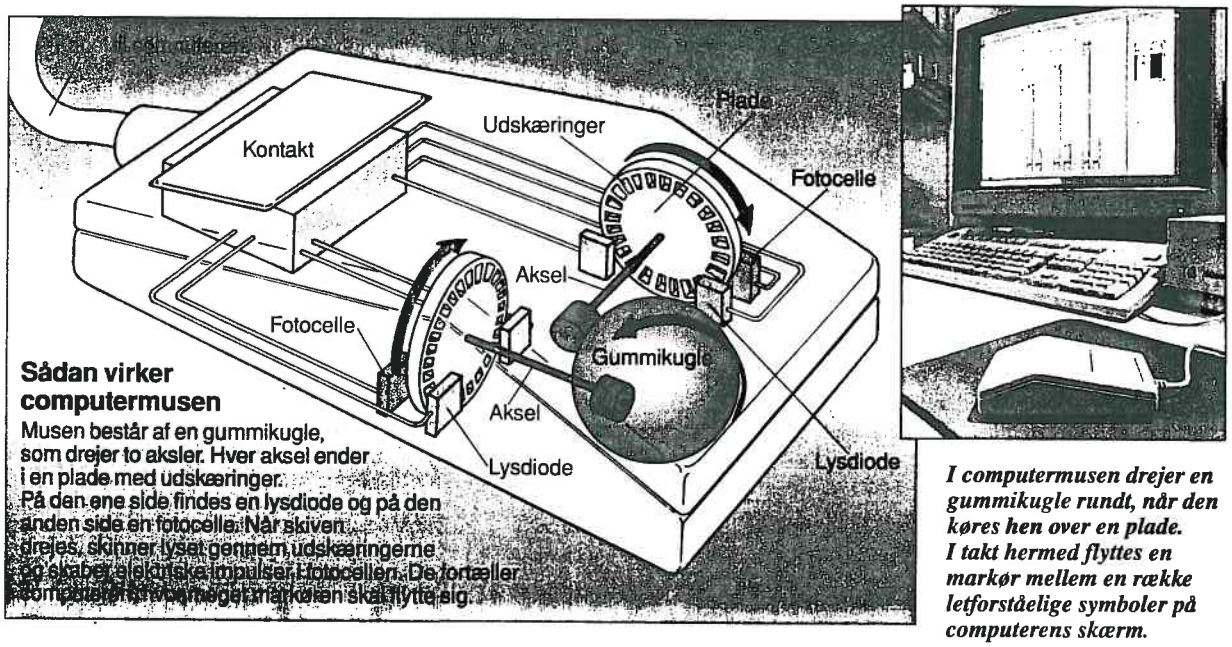
00 - 02	...	-	Jump unden om beskivelse af medie, til BOOT-PROGRAMMET.
03 - 0A	CHAR	-	Producentnavn i ASCII-format.
0B - 0C	WORD	-	Antal bytes pr. sektor.
0D - 0D	BYTE	-	Antal sektorer pr. cluster.
0E - 0F	WORD	-	Antal reserverde sektorer.
10 - 10	BYTE	-	Antal FAT tabeller
11 - 12	WORD	-	Antal entries i ROOT.
13 - 14	WORD	-	Antal sektorer ialt på medie.
15 - 15	BYTE	-	Medie descriptor - står også i første byte i FAT-tabel. Før DOS 3.30 anvendtes alene descriptor fra FAT-tabel når DOS skulle anvende en diskette og format var derfor fastlåst.
16 - 17	WORD	-	Antal sektorer pr. FAT-tabel.
18 - 19	WORD	-	Antal sektorer pr. spor.
1A - 1B	WORD	-	Antal sider.
1C - 1D	WORD	-	Antal skjulte sektorer (til DOS 3.xx)
1E - 1FF	...		Resten af BOOT-programmet.

Fra DOS 4.00 er der informationerne i BOOT-sektoren udvidet. Dette skyldtes blandt andet den begrænsning der ligger i at feltet med antal sektorer på mediet kun er et word, hvilket med en sektorstørrelse kun giver mulighed for 32 MB.

Det nye format bygger videre på det gamle og rummer også en beskrivelse (i klar tekst) af om FAT-tabellen er en 12-bits eller en 16-bits. BOOT-sektoren rummer også volume lablen, idet denne stadig også opbevares i ROOT-directory.

2.6.a

PC-MUS



Sådan virker computermusen
Musen består af en gummikugle, som drejer to aksler. Hver aksel ender i en plade med udskæringer. På den ene side findes en lysdiode og på den anden side en fotocelle. Når skiven drejes, skinner lyset gennem udskæringerne og skaber elektriske impulser i fotocellen. De tortæller computeren hvor meget markøren skal flytte sig.

I computermusen drejer en gummikugle rundt, når den køres hen over en plade. I takt hermed flyttes en markør mellem en række letforståelige symboler på computerens skærm.

Genvej til computerens hjerne

Teknik Det er langsommeligt og besværligt at styre et indviklet computerprogram ved hjælp af et tastatur. Opfindelsen af computermusen har gjort det meget nemmere at bruge en PC. I musen findes en gummikugle, som styrer to aksler. Den ene styrer de vandrette bevægelser, den anden de lodrette. Når musen kører hen over en plade, sendes elektriske impulser til computeren, og en lille pil flytter sig rundt på skærmen.

Pilen kan pege på forskellige symboler, som aktiveres ved hjælp af en trykknap på musen. Derefter udføres den bestemte del af computerprogrammet.

Betegnelsen mus blev patenteret allerede i 1964 som det engelske »mouse«. □

3.

Den lagdelte maskine

Indhold:

3.1.a

Structured computer organization,
Second edition,
Prentice Hall
Andrew S. Tannenbaum
1984

Uddrag af kapitel 1.

A digital computer is a machine that can solve problems for people by carrying out instructions given to it. A sequence of instructions describing how to perform a certain task is called a **program**. The electronic circuits of each computer can recognize and directly execute a limited set of simple instructions into which all its programs must be converted before they can be executed. These basic instructions are rarely much more complicated than:

Add 2 numbers.

Check a number to see if it is zero.

Move a piece of data from one part of the computer's memory to another.

Together, a computer's primitive instructions form a language in which it is possible for people to communicate with the computer. Such a language is called a **machine language**.

The people designing a new computer must decide what instructions to include in its machine language. Usually they try to make the primitive instructions as simple as possible, consistent with the computer's intended use and performance requirements, in order to reduce the complexity and cost of the electronics needed. Because most machine languages are so simple, it is difficult and tedious for people to use them.

This problem can be attacked in two principal ways: both involve designing a new set of instructions that is more convenient for people to use than the set of built-in

machine instructions. Taken together, these new instructions also form a language, which we will call L2, just as the built-in machine instructions form a language, which we will call L1. The two approaches differ in the way programs written in L2 are executed by the computer, which, after all, can only execute programs written in its machine language, L1.

One method of executing a program written in L2 is first to replace each instruction in it by an equivalent sequence of instructions in L1. The resulting program consists entirely of L1 instructions. The computer then executes the new L1 program instead of the old L2 program. This technique is called **translation**.

The other technique is to write a program in L1 that takes programs in L2 as input data and carries them out by examining each instruction in turn and executing the equivalent sequence of L1 instructions directly. This technique does not require first generating a new program in L1. It is called **interpretation** and the program that carries it out is called an **interpreter**.

Translation and interpretation are similar. In both methods instructions in L2 are ultimately carried out by executing equivalent sequences of instructions in L1. The difference is that, in translation, the entire L2 program is first converted to an L1 program, the L2 program is thrown away, and then the new L1 program is executed. In interpretation, after each L2 instruction is examined and decoded, it is carried out immediately. No translated program is generated. Both methods are widely used.

Rather than thinking in terms of translation or interpretation, it is often more convenient to imagine the existence of a hypothetical computer or **virtual machine** whose machine language is L2. If such a machine could be constructed cheaply enough, there would be no need for having L1 or a machine that executed programs in L1 at all. People could simply write their programs in L2 and have the computer execute them directly. Even though the virtual machine whose language is L2 is too expensive to construct out of electronic circuits, people can still write programs for it. These programs can either be interpreted or translated by a program written in L1 that itself can be directly executed by the existing computer. In other words, people can write programs for virtual machines, just as though they really existed.

To make translation or interpretation practical, the languages L1 and L2 must not be "too" different. This constraint often means that L2, although better than L1, will still be far from ideal for most applications. That L2 should be far from ideal is perhaps discouraging in light of the original purpose for creating it—namely, to relieve the programmer of the burden of having to express algorithms in a language more suited to machines than people. However, the situation is far from hopeless.

The obvious approach is to invent still another set of instructions that is more people-oriented and less machine-oriented than those of L2. This third set also forms a language, which we will call L3. People can write programs in L3 just as though a virtual machine with L3 as its machine language really existed. Such programs can either be translated to L2 or executed by an interpreter written in L2.

The invention of a whole series of languages, each one more convenient than its predecessors, can go on indefinitely until a suitable one is finally achieved. Each language uses its predecessor as a basis, so we may view a computer using this

technique as a series of **layers** or **levels**, one on top of another, as shown in Fig. 1-1. The bottom-most language or level is the simplest and the highest language or level is the most sophisticated.

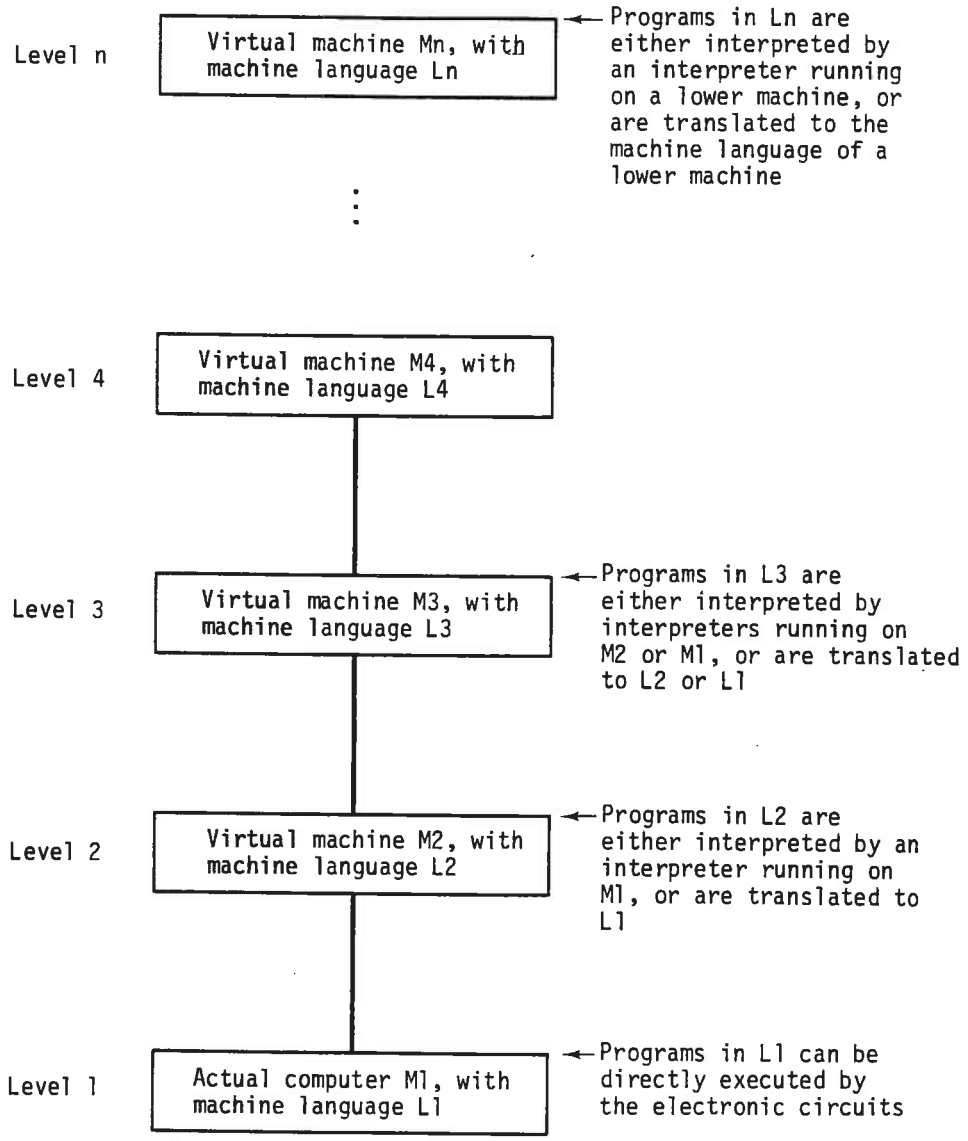


Fig. 1-1. A multilevel machine.

1.1. LANGUAGES, LEVELS, AND VIRTUAL MACHINES

There is an important relation between a language and a virtual machine. Each machine has some machine language, consisting of all the instructions that the

machine can execute. In effect, a machine defines a language. Similarly, a language defines a machine—namely, the machine that can execute all programs written in the language. Of course, the machine defined by a certain language may be enormously complicated and expensive to construct directly out of electronic circuits but we can imagine it nevertheless. A machine with Ada*, Pascal, or COBOL as its machine language would be a complex beast indeed but it is certainly conceivable, and perhaps in a few years such a machine will be considered trivial to build.

A computer with *n* levels can be regarded as *n* different virtual machines, each with a different machine language. We will use the terms "level" and "virtual machine" interchangeably. Only programs written in language L1 can be directly carried out by the electronic circuits, without the need for intervening translation or interpretation. Programs written in L2, L3, ..., L*n* must either be interpreted by an interpreter running on a lower level or translated to another language corresponding to a lower level.

A person whose job it is to write programs for the level *n* virtual machine need not be aware of the underlying interpreters and translators. The machine structure ensures that these programs will somehow be executed. It is of little interest whether they are carried out step by step by an interpreter which, in turn, is also carried out by another interpreter, or whether they are carried out directly by the electronics. The same result appears in both cases: the programs are executed.

Most programmers using an *n*-level machine are only interested in the top level, the one least resembling the machine language at the very bottom. However, people interested in understanding how a computer really works must study all the levels. People interested in designing new computers or designing new levels (i.e., new virtual machines) must also be familiar with levels other than the top one. The concepts and techniques of constructing machines as a series of levels and the details of some important levels themselves form the main subject of this book. The title *Structured Computer Organization* comes from the fact that viewing a computer as a hierarchy of levels provides a good structure or framework for understanding how computers are organized. Furthermore, designing a computer system as a series of levels helps to ensure that the resulting product will be well structured.

1.2. CONTEMPORARY MULTILEVEL MACHINES

Most modern computers consist of two or more levels. Six-level machines are not at all unusual, as shown in Fig. 1-2. Level 0, at the bottom, is the machine's true hardware. Its circuits carry out the machine language programs of level 1. For the sake of completeness, we should mention the existence of yet another level below our level 0. This level, not shown in Fig. 1-2, because it falls within the realm of electrical engineering (and is thus outside the scope of this book) is called the **device level**. At this level, the designer sees individual transistors, which are the lowest-level

*Ada is a trademark of the U.S. Department of Defense.

primitives for computer designers. (Of course, one can also ask how transistors work inside but that gets into solid-state physics.)

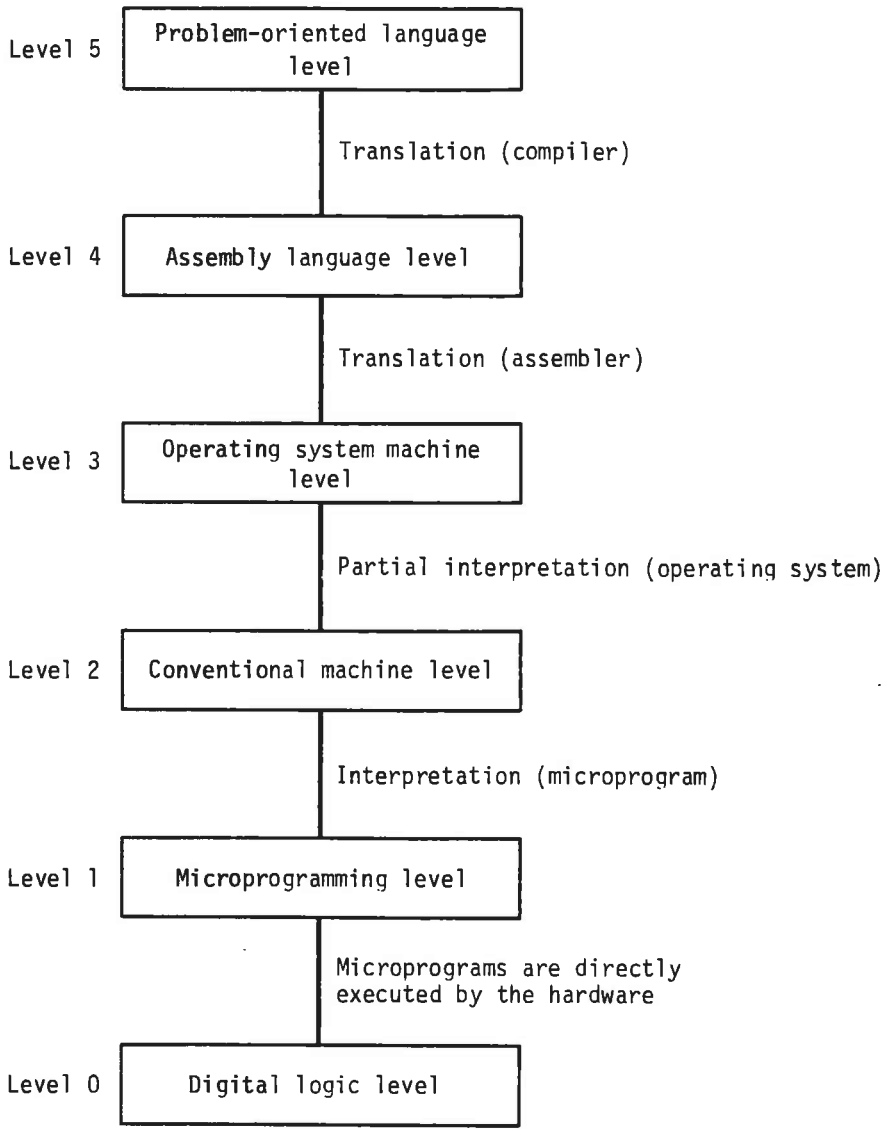


Fig. 1-2. Six levels present on most modern computers. The method by which each level is supported is indicated below it, along with the name of the supporting program in parentheses.

At the lowest level that we will study, the **digital logic level**, the interesting objects are called **gates**. These gates are digital, unlike transistors, which are analog. Each gate has one or more digital inputs (signals representing 0 or 1) and computes as output some simple function of these inputs, such as AND or OR. Each gate is built up

of at most a handful of transistors. We will examine the digital logic level in detail in Chap. 3. Although knowledge of the device level is something of a specialty, with the advent of microprocessors and microcomputers, more and more people are coming in contact with the digital logic level. For this reason we have included the latter in our model and devoted an entire chapter of the book to it.

The next level up is level 1, which is the true machine language level. In contrast to level 0, where there is no concept of a program as a sequence of instructions to be carried out, in level 1 there is definitely a program, called a **microprogram**, whose job it is to interpret the instructions of level 2. We will call level 1 the **microprogramming level**. Although it is true that no two computers have identical microprogramming levels, enough similarities exist to allow us to abstract out the essential features of the level and discuss it as though it were well defined. For example, few machines have more than 20 instructions at this level and most of these instructions involve moving data from one part of the machine to another, or making some simple tests.

Each level 1 machine has one or more microprograms that can run on it. Each microprogram implicitly defines a level 2 language (and a virtual machine, whose machine language is that language). These level 2 machines also have much in common. Even level 2 machines from different manufacturers have more similarities than differences. In this book we will call this level the **conventional machine level**, for lack of a generally agreed-upon name.

Every computer manufacturer publishes a manual for each of the computers it sells, entitled "Machine Language Reference Manual" or "Principles of Operation of the Western Wombat Model 100X Computer" or something similar. These manuals are really about the level 2 virtual machine, not the level 1 actual machine. When they describe the machine's instruction set, they are in fact describing the instructions carried out interpretively by the microprogram, not the hardware instructions themselves. If a computer manufacturer provided two interpreters for one of its machines, interpreting two different level 2 machine languages, it would need to provide two "machine language" reference manuals, one for each interpreter.

It should be mentioned that some computers, particularly older ones, do not have a microprogramming level. On these machines the conventional machine level instructions are carried out directly by the electronic circuits (level 0), without any level 1 intervening interpreter. As a result, level 1 and not level 2 is the conventional machine level. Nevertheless, we will continue to call the conventional machine level "level 2," despite these exceptions.

The third level is usually a hybrid level. Most of the instructions in its language are also in the level 2 language. (There is no reason why an instruction appearing at one level cannot be present at other levels as well.) In addition, there is a set of new instructions, a different memory organization, the ability to run two or more programs in parallel, and various other features. More variation exists between level 3 machines than between either level 1 machines or level 2 machines.

The new facilities added at level 3 are carried out by an interpreter running at level 2, which, historically, has been called an **operating system**. Those level 3

instructions identical to level 2's are carried out directly by the microprogram, not by the operating system. In other words, some of the level 3 instructions are interpreted by the operating system and some of the level 3 instructions are interpreted directly by the microprogram. This is what we mean by "hybrid." We will call this level the **operating system machine level**.

There is a fundamental break between levels 3 and 4. The lowest three levels are not designed for direct use by the average garden-variety programmer. They are intended primarily for running the interpreters and translators needed to support the higher levels. These interpreters and translators are written by people called **systems programmers** who specialize in designing and implementing new virtual machines. Levels 4 and above are intended for the applications programmer with a problem to solve.

Another change occurring at level 4 is the method by which the higher levels are supported. Levels 2 and 3 are always interpreted. Levels 4, 5, and above are usually, although not always, supported by translation.

Yet another difference between levels 1, 2, and 3, on the one hand, and levels 4, 5, and higher, on the other, is the nature of the language provided. The machine languages of levels 1, 2, and 3 are numeric. Programs in them consist of long series of numbers, which are fine for machines but bad for people. Starting at level 4, the languages contain words and abbreviations meaningful to people.

Level 4, the assembly language level, is really a symbolic form for one of the underlying languages. This level provides a method for people to write programs for levels 1, 2, and 3 in a form that is not as unpleasant as the virtual machine languages themselves. Programs in assembly language are first translated to level 1, 2, or 3 language and then interpreted by the appropriate virtual or actual machine. The program that performs the translation is called an **assembler**. Assembly language once was important but it is becoming less important as time goes on.

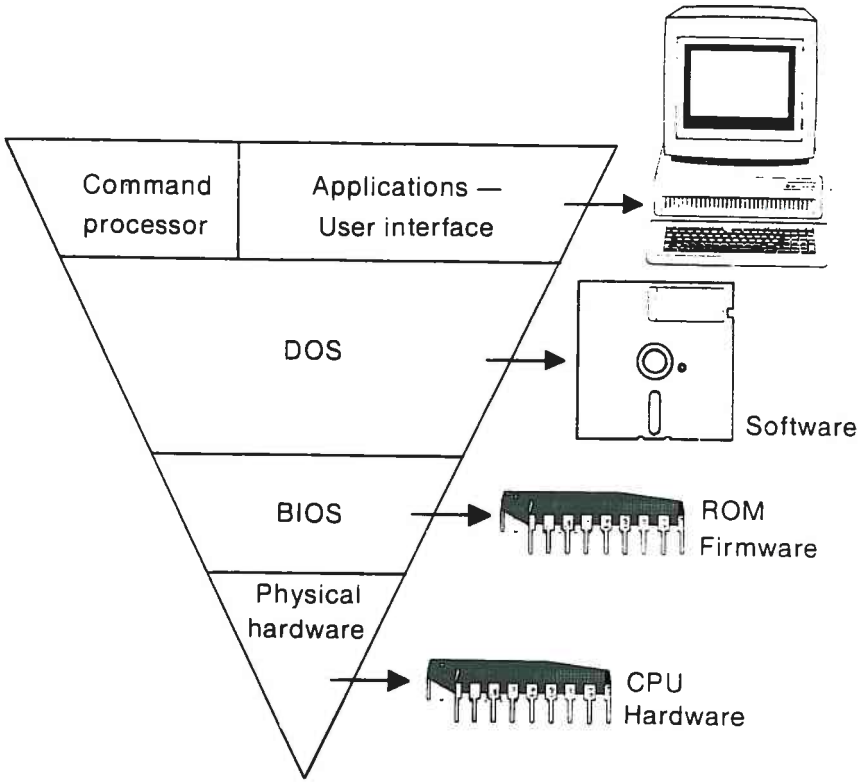
Level 5 consists of languages designed to be used by applications programmers with problems to solve. Such languages are called by many names, including **high-level languages** and **problem-oriented languages**. Literally hundreds of different ones exist. A few of the better known ones are Ada, ALGOL 68, APL, BASIC, C, COBOL, FORTRAN, LISP, Pascal, and PL/1. Programs written in these languages are generally translated to level 3 or level 4 by translators known as **compilers**, although occasionally they are interpreted instead.

Levels 6 and above consist of collections of programs designed to create machines specifically tailored to certain applications. They contain large amounts of information about that application. It is possible to imagine virtual machines intended for applications in administration, education, computer design, and so on. These levels are an area of current research.

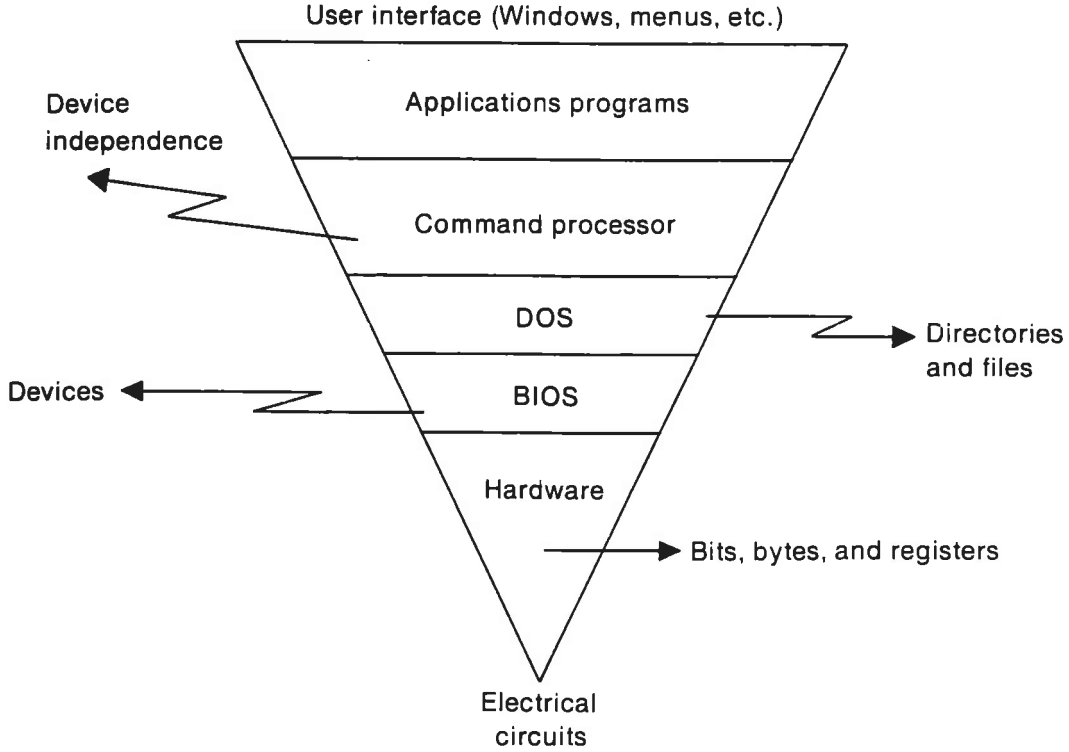
In summary, the key thing to remember is that computers are designed as a series of levels, each one built on its predecessor. Each level represents a distinct abstraction, with different objects and operations present. By designing and analyzing computers in this fashion, we are temporarily able to suppress irrelevant details and thus reduce a complex subject to something easier to understand.

3.1.b

The virtual machine hierarchy.



System layering.



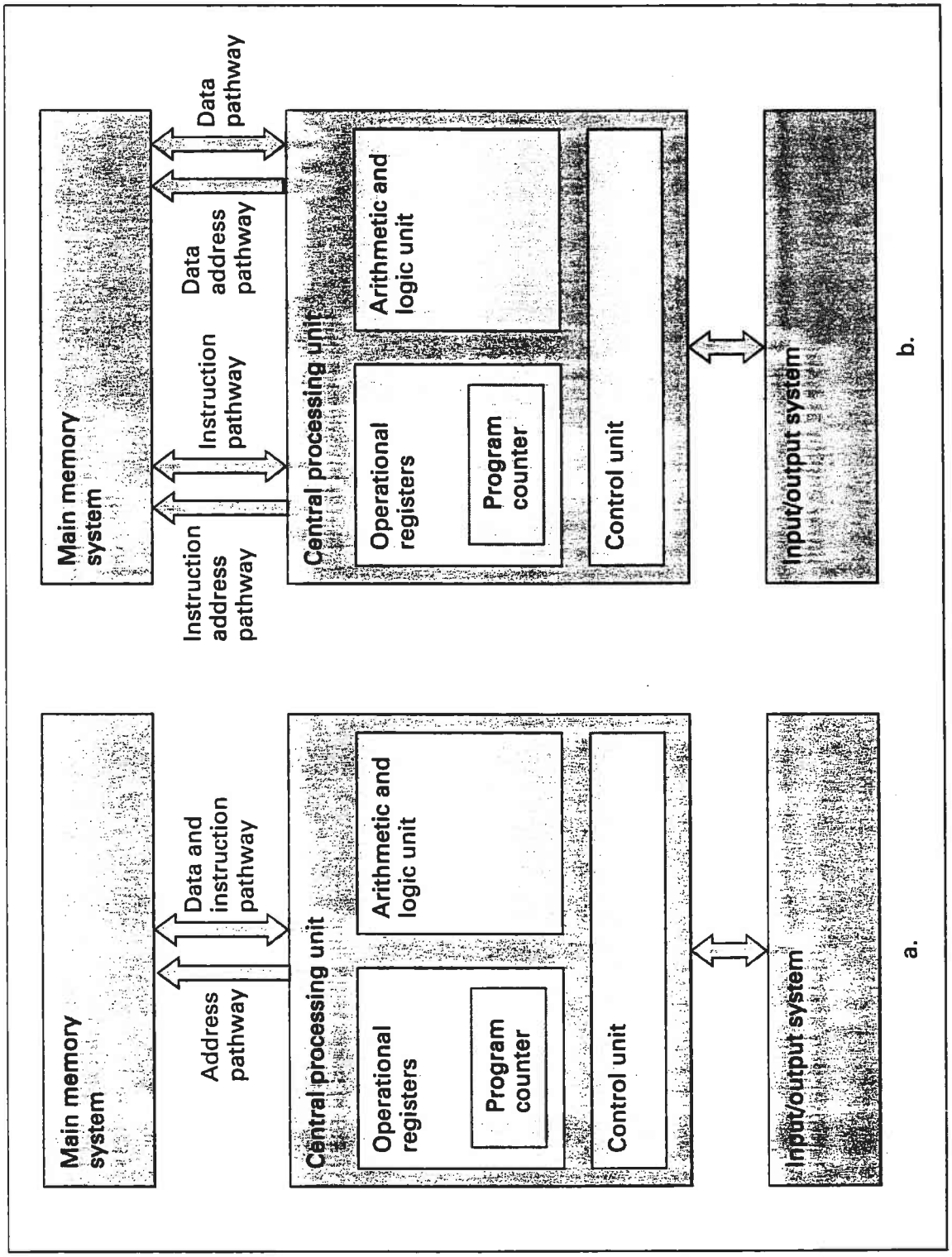
4.

Den konventionelle maskine

Indhold:

4.1.a

Von Neuman maskiner



Main components of typical von Neumann machines. (a) A conventional von Neumann architecture. (b) A Harvard architecture.

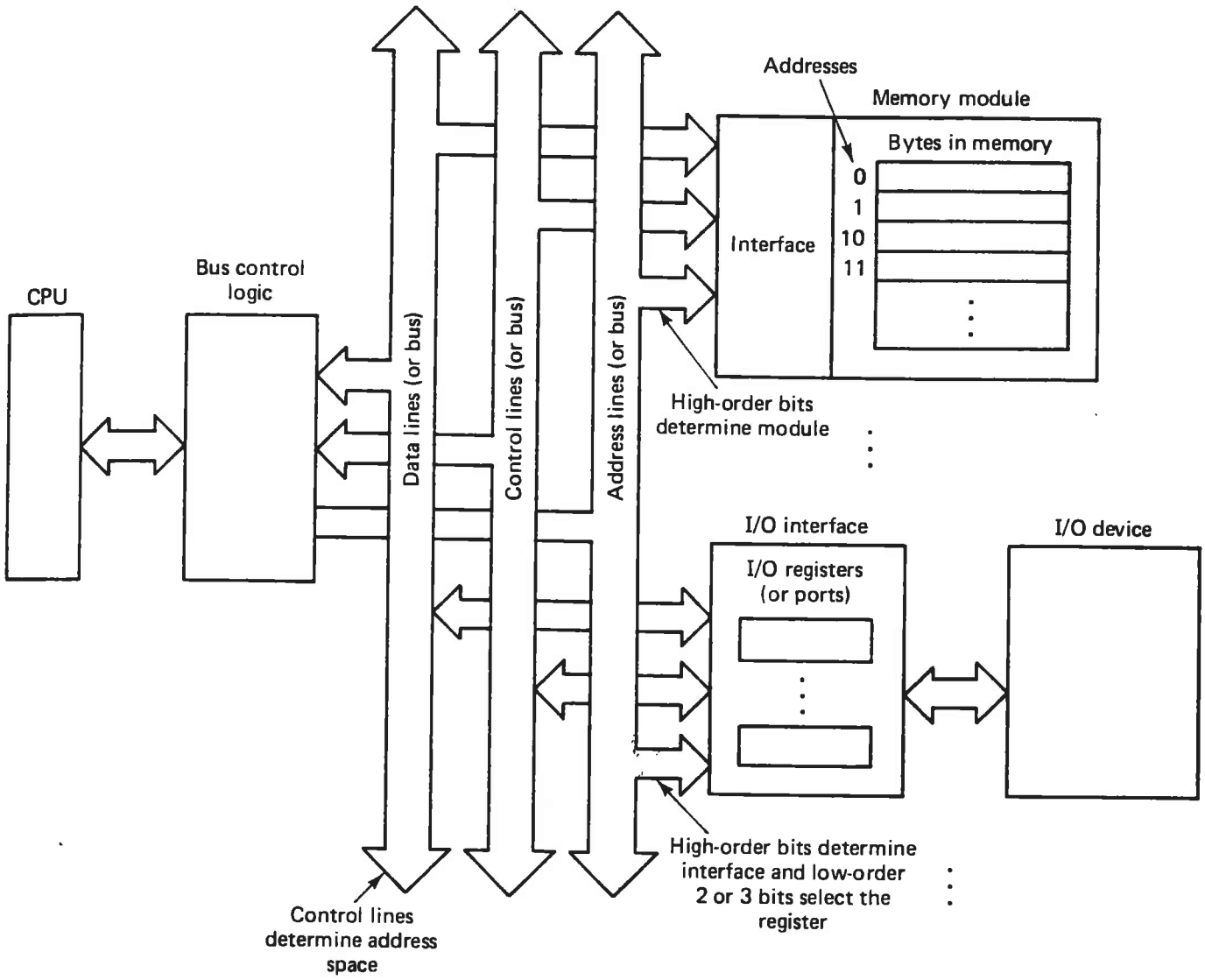
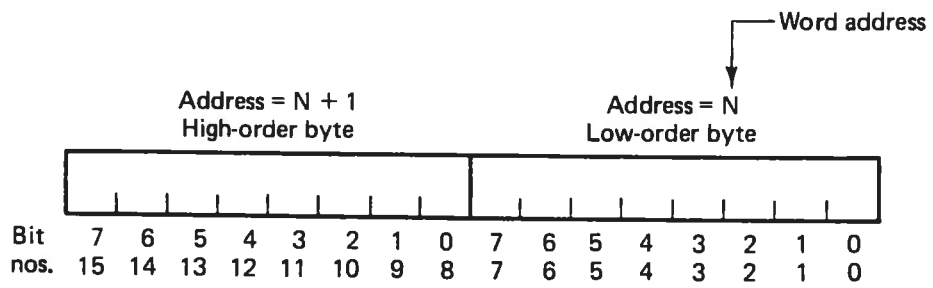


Figure 1-7 Memory and I/O register organization.

The bits are numbered with 0 being assigned to the least significant bit (LSB). In a byte the most significant bit (MSB) is numbered 7 and in a word the MSB is numbered 15. These conventions are summarized in Fig. 1-8.

Some computers require that words begin with addresses that are divisible by the number of bytes in a word and produce an alignment error if this rule is not followed. Others, such as the 8086, permit words to begin at any address;

Figure 1-8 Address and bit numbering conventions.



4.2.a

Modelmaskinen

Bjarne Larsen

INDHOLDSFORTEGNELSE

Modelmaskinens opbygning og virkemåde

Programmering af datamaskinen.

Ved programmering af datamaskinen er det programmørens opgave at sammensætte et program med instruktioner fra maskinens instruktionssæt. Instruktionerne skal sammensættes på en sådan måde, at de tilsammen og udført i den rigtige rækkefølge vil bevirke, at et givent problem vil blive løst.

Hver enkelt instruktion består af mindst to dele, en operationskode og en adresse. Operationskoden skal fortælle datamaskinen hvilken operation, der skal udføres. Adressen fortæller datamaskinen hvilke data, operationen skal udføres på.

Datamaskinens virkemåde.

Med henblik på at anskueliggøre datamaskinens virkemåde under et programs udførelse er der konstrueret en stærkt forenklet imaginær modelmaskine. Ved hjælp af denne modelmaskine kan man få et overblik over datamaskinens arbejde i de forskellige faser af en instruktions udførelse. Arbejdet i disse faser kan kort beskrives således:

1. Hent næste instruktion på den adresse, som programtælleren peger på. Programtælleren er et specialregister, som altid indeholder adressen på den næste instruktion, der skal udføres.

2. Den fundne instruktion overføres fra det interne lager til processoren, hvor operationskodedelen af instruktionen indsættes i operationskoderegistret og instruktionens adressedel indsættes i adresseregistret.

3. Der adderes 1 til programtælleren, som således kommer til at pege på den næste instruktion i det interne lager.

4. Operationskoden i operationskoderegistret fortolkes.

5. Operationen udføres. Denne fase er forskellig afhængig af den operationskode, som står i operationskoderegistret. Hvad der præcist sker i forbindelse med de enkelte oprationskoder, er omtalt i forbindelse med beskrivelsen af modelmaskinens instruktionssæt.

Modelmaskinens opbygning.

Modelmaskinens centralenhed består at et internt lager og en processor. Det interne lager, som skal indeholde programmet og dets data, er opbygget af 40 celler, som hver kan rumme 8 decimale cifre.

Modelmaskinens opbygning og virkemåde

Processoren består af en styreenhed, en arimetisk/logisk enhed samt et akkumulatorregister, som kan rumme 8 decimale cifre, et oprationskoderegister på 2 decimale cifre, et adresseregister og en programtæller på hver 6 decimale cifre.

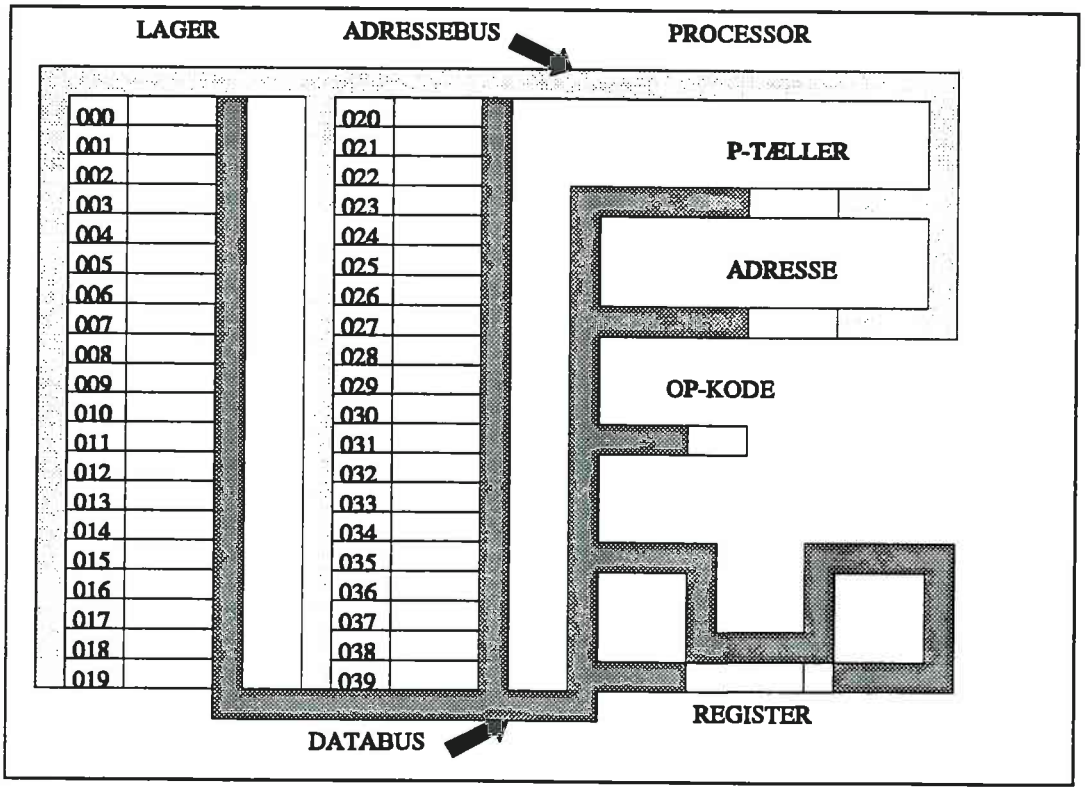
Imellem det interne lager og processoren er der dels en adressebus, som bruges til udpegning af den celle, der ønskes overført til eller fra processoren, dels en databus, hvorigennem indholdet af den udpegede celle i lageret overføres til eller fra processoren.

Positive og negative tal.

Af hensyn til en så forenklet fremstilling som muligt, er der ved konstruktionen af modelmaskinen kun åbnet mulighed for at arbejde med numerisk decimalt indhold i celler og registre. Til gengæld er det muligt at arbejde med såvel positive som negative decimale tal, idet der benyttes sædvanlig 2-komplement repræsentation af negative tal.

I det decimale talsystem betyder dette, at det første ciffer i en celles eller registrets indhold skal være 9, hvis tallet skal være negativt. Eksempelvis vil tallet -423 i en af det interne lagers celler eller akkumulatorregistret være repræsenteret ved tallet 99999577, og tallet -1 vil være repræsenteret ved tallet 99999999.

Principskitse over modelmaskinen.



Modelmaskinens opbygning og virkemåde

Modelmaskinens instruktionssæt.

Inden vi går over til en nærmere beskrivelse af modelmaskinens instruktionssæt, skal vi kort omtale instruktionsformatet. Instruktionsformatet fortæller, hvordan indholdet af en celle er opdelt i en oprationskodedel og en adressedel. Instruktionsformatet er nært forbundet med og afhængig af, hvordan processoren er konstrueret.

I modelmaskinen er der et 2-cifret operationskoderegister og et 6-cifret adresseregister. Det er derfor naturligt at lade modelmaskinens instruktionsformat diktere, at de to første cifre i de celler, som indeholder instruktioner, skal indeholde en to-cifret operationskode, og at de sidste seks cifre skal indeholde adressen på det datafelt, som instruktionen skal virke på.

En datamaskines instruktionssæt er summen af de instruktioner, som maskinen er bygget til at kunne udføre. I det følgende gennemgås de enkelte instruktioner, som modelmaskinen kan udføre. Instruktionerne gennemgås i nummerorden sorteret efter den to-cifret operationskode.

01 LOAD

Med denne instruktion henter man indholdet af den celle, som instruktionens adressedel peger på, til akkumalatorregistret.

02 ADD

Indholdet af den celle, som instruktionens adressedel peger på, hentes til processoren, hvor det adderes til indholdet af akkumulatorregistret. Resultatet af additionen anbringes i registret.

03 SUBTRACT

Indholdet af den celle, som instruktionens adressedel peger på, hentes til processoren, hvor det subtraheres fra indholdet af registret. Resultatet af subtraktionen anbringes i registret.

04 DIVIDE

Indholdet af registret divideres med indholdet af den celle, som instruktionens adressedel peger på. Kvotienten ved divisionen anbringes i registret.

05 MULTIPLY

Indholdet af den celle, som instruktionens adressedel peger på, hentes til processoren, hvor den multipliceres med indholdet af registret. Produktet anbringes i registret.

Modelmaskinens opbygning og virkemåde

06 STORE

Indholdet af registret gemmes i den celle, som instruktionens adressedel peger på.

07 JUMP

Med denne og de to næste instruktioner er det muligt at gribe ind i, fra hvilken celle i lageret den næste instruktion skal hentes. Denne instruktion har til formål at bevirke et hop i programmet til den celle, som instruktionens adressedel peger på. I praksis sker dette ved, at indholdet af adresseregistret overføres til programtælleren.

08 TESTZERO

Denne instruktion tester, om indholdet af registret er nul. Hvis udfaldet af testen er sandt (registret er nul), springes næste instruktion over ved at addere 1 til programtælleren. Ellers fortsættes på normal måde med næste instruktion.

09 TESTGREATER

Indholdet af den celle, som instruktionens adressedel peger på, overføres til processoren, hvor det sammenlignes med indholdet af registret. Hvis indholdet af cellen er større end indholdet af registret, er testen sand, og der adderes 1 til programtælleren. Ellers fortsættes på normal måde med næste instruktion.

99 STOP

Udførelsen af denne instruktion bevirker, at programmet stoppes.

Eksempel på et program.

Indholdet af celle 20 og 21 ønskes adderet, og resultatet anbragt i celle 20. Hvis de to tal, som ønskes adderet, for eksempel er 14 og 49, kan en løsning af problemet programmeres på denne måde:

```
000: 01000020     LOAD CELLE 20
001: 02000021     ADD CELLE 21
002: 06000020     STORE CELLE 20
003: 99000000     STOP
 :
 :
020: 00000014
021: 00000049
 :
```

Modelmaskinens opbygning og virkemåde

programafbrydelser (interrupts) i modelmaskinen.

Den normale måde at stoppe et program på er ved at udføre STOP-instruktionen. En sådan afbrydelse af programmet er tilsigtet og planlagt; men der er også mulighed for, at maskinen selv kan afbryde programmet på en mere eller mindre utilsigtet måde. En sådan situation, hvor maskinen selv stopper programmet, kalder man en afbrydelse eller et interrup.

Et interrupt forekommer, når maskinen bliver stillet overfor en uløselig opgave. Modelmaskinen kan afbryde et program som følge af tre forskellige fejl. For det første kan der opstå adresseringsfejl. En adresseringsfejl opstår, hvis en lagerreference enten i programtælleren eller i adresseregistret peger på en celle, som ikke findes i lageret.

For det andet kan der ske afbrydelse på grund af en invalid operationskode. Denne situation opstår, når styreenheden ikke kan genkende operationskoden i operationskoderegistret. En sådan fejl opstår typisk, når en celle som indeholder data, bringes til processoren, og ved et udheld bliver fortolket som om den indeholdt en instruktion.

Den tredje afbrydelse sker, hvis man i en divisionsinstruktion forsøger at dividere med nul, idet division med nul er og bliver umuligt. Programmeringsmæssigt kan man sikre sig imod, at dette problem opstår ved at teste divisor, inden divisionen udføres.

Endelig kan der opstå en fjerde situation, som dog ikke udløser nogen afbrydelse. Situationen kaldes overflow, op opstår, når resultatet af en addition eller multiplikation ikke kan stå i registret. Maskinen markerer, at situationen er opstået, hvorefter de mest betydende cifre i resultatet afskæres, og maskinen fortsætter.

Brugervejledning for modelmaskineprogrammet.

Med henblik på at illustrere principperne bag den foran beskrevne modelmaskines virkemåde, er modelmaskinen blevet programmeret, således at en arbejdende modelmaskine simuleres på en PC-farveskærm som en slags tegnefilm. Ved hjælp af forskellige kommandoer kan man styre simuleringen og på den måde få en klar fornemmelse af maskinens virkemåde i forskellige situationer.

Opstart af systemet.

Systemet kan køres på enhver IBM-kompatibel PC'er med farveskærm. Når maskinen er tændt, og operativsystemet er indlæst, startes modelmaskineprogrammet ved indtastning af kommandoen:

ASM

Brugervejledning

Hvis man ikke råder over en farveskærm, må man anvende den sort-hvide udgave af modelmaskineprogrammet, som startes ved indtastning af kommandoen:

ASMSH

Der er mindre maskinafhængige forskelle i den måde modelmaskinen præsenteres på skærmen; men selve programmets virkemåde er den samme.

Efter opstarten kommer et billede af modelmaskinen frem på skærmen, og samtidig opfordres man til at foretage et funktionsvalg.

Funktionsvalg.

Når man opfordres til at foretage et funktionsvalg, sker det med følgende udskrift i højre side af skærmen:

FUNKTIONSVALG

E - Exit
G - Gem program
H - Hastighedsændring
I - Indsæt i celler
L - Læs program ind
N - Nulstil lagret
S - Startadr. i P-tæller
T - Trinvis udførelse til/fra
U - Udfør programmet
X - Sætter lyd til/fra

INDTAST FUNKTION:

Selve funktionsvalget foretages ved indtastning af det ønskede bogstav på tastaturet. Man kan anvende såvel små som store bogstaver. Funktionen vil blive kaldt straks, når det ønskede bogstav er indtastet. Indtastningen skal altså ikke afsluttes med tryk på returknappen i forbindelse med funktionsvalget.

E - Exit

Bogstavet E indtastes, når man ønsker at afslutte simuleringsprogrammet. Kontrollen gives tilbage til operativsystemet.

G - Gem program

Denne funktion kan med fordel anvendes, når man har indlagt et program i modelmaskinens interne lager, idet funktionen kan bruges til at gemme en kopi af det interne lagers 40 celler på pladelager eller diskette, hvorfra de senere kan indlæses igen. Hvis man ønsker at gemme programmet i sin oprindelige form, bør man gemme det umiddelbart efter, at man har indtastet det, og før en eventuel udførelse påbegyndes, da udførelsen kan bevirke, at nogle af cellernes indhold ændres.

Brugervejledning

Når funktionen er blevet kaldt, udskrives følgende nederst til højre på skærmen:

Programmet gemmes

Indtast navn: [ ]

I det oplyste felt indtastes det navn, som man ønsker at gemme programmet under. Findes der i forvejen et program med dette navn, udskrives der en fejlmeddelse, og programmet gemmes ikke.

H - Hastighedsændring

Det er muligt at ændre den hastighed, hvormed simuleringen af datastrømmene i adresse- og databus foregår. Dette gøres ved at variere den forsinkelsesfaktor, som er skudt ind i simuleringen af datastrømmene, for at 'tegnefilmen' ikke skal køre for hurtigt. Denne forsinkelsesfaktor er som standardværdi blevet tildelt værdien 200 millisekunder mellem hvert nyt billede af datastrømmen. Gøres forsinkelsesfaktoren større, sættes arbejdshastigheden ned. Gøres forsinkelsesfaktoren mindre, øges arbejdshastigheden.

Når funktionen kaldes, udskrives følgende tekst til højre på skærmen:

Ændring af programmets arbejdshastighed under udførelsen - jo større tal der indtastes - jo langsommere

INDTAST FAKTOR: [200]

I det oplyste felt står standardværdien 200, som kan accepteres ved tryk på returknappen. Ønskes en anden værdi, indtastes en værdi mellem 1 og 9999.

I - Indsæt i celler

Denne funktion kaldes, hvis man vil indlægge program og data i det interne lagers celler. Hvis man gør dette umiddelbart efter, at man har startet programmet, er cellerne nulstillet; men hvis der allerede står noget i cellerne, bør man nulstille dem ved kald af nulstil-funktionen (se side 8), inden man begynder at indtaste et nyt indhold.

Når man har kaldt funktionen, skal man svare på følgende spørgsmål, som kommer frem nederst til højre på skærmen:

INDSÆT I CELLE: [99]

CELLEINDHOLD : [ ]

Brugervejledning

Det oplyste felt er det felt, hvor der skal indtastes nummeret på den celle, som skal have tildelt en værdi. Det indtastede cellenummer skal være mellem 0 og 39 eller 99. Indtastning af cellenummer 99 bevirker, at indtastningsfunktionen afbrydes, og der vendes tilbage til funktionsvalget. Cellenummer 99 er det cellenummer, som maskinen selv foreslår, og dette betyder, at indtastningen af celleindhold kan stoppes blot ved at trykke på returknappen. Hvis cellenummeret ligger uden for de acceptable grænser, bipper maskinen.

Når et acceptabelt cellenummer er indtastet, skal cellen i næste linie tildeles et indhold. Det ønskede celleindhold indtastes i det oplyste felt. Indholdet skal være numerisk, hvilket vil sige, at der kun må indtastes cifre.

Skulle man ved et uheld komme til at afbryde indtastningen af celleindhold, før man er helt færdig, eller skulle man få behov for at ændre allerede indtastede cellers indhold, kan man blot kalde indtastningsfunktionen igen.

L - Læs program ind

Denne funktion modsvarer den tidligere beskrevne funktion, hvormed det var muligt at gemme et indtastet program på pladelager eller diskette. Med denne funktion er det muligt at hente et tidligere gemt program og få det lagt ind i modelmaskinens lager - klar til udførelse.

Når funktionen er blevet kaldt, udskrives følgende nederst på skærmen:

Gemt program læses ind

Indtast navn : [ ]

I det oplyste felt indtastes navnet på det program, som man ønsker at indlæse. Hvis det ønskede program ikke findes, udskrives en fejlmeddelse, hvorefter man opfordres til at foretage fornyet funktionskald.

N - Nulstil lageret

Denne funktion kaldes, når modelmaskinens interne lager ønskes nulstillet. Dette behov opstår hovedsageligt, når et nyt program skal indtastes. Man skal være opmærksom på, at alle celler nulstilles. Hvis det kun er enkelte celler, som ønskes nulstillet, skal man anvende den normale indtastningsfunktion, hvormed man også kan indsætte nuller i enkelte celler.

S - Startadresse indsættes i P-tæller

Inden et program, som ligger i modelmaskinens interne lager, kan udføres, er det nødvendigt at indsætte programmets startadresse i programtælleren. Programmets startadresse er nummeret på den celle i det interne lager, hvor programmets første instruktion findes.

Brugervejledning

Når funktionen kaldes, udskrives følgende tekst nederst til højre på skærmen:

Programmets startadresse
indsættes i P-tæller

INDTAST P-TÆLLER: [0]

I det oplyste felt indtastes nummeret på den celle, hvor programmets første instruktion findes. Den foreslåede standardværdi for feltet er 0, som kan accepteres ved blot at trykke på returknappen. Samtidig med at P-tælleren bliver tildelt en værdi, nulstilles de øvrige registre.

T - Trinvis udførelse til/fra

Af pædagogiske årsager kan det være overordentligt hensigtsmæssigt at lade maskinen stoppe efter udførelse af hver enkelt instruktion. Dette kan lade sig gøre ved at slå faciliteten "trinvis udførelse" til ved tryk på T-knappen på tastaturet. Næste tryk på T-knappen slår trinvis udførelse fra igen og så fremdeles. Når trinvis udførelse er slået til, skal man kalde udfør-instruktionen hver gang, man ønsker næste instruktion udført. Standardværdien for trinvis udførelse ved opstart af programmet er "fra".

U - Udfør programmet

Når man ønsker at starte udførelsen af programmet, eller hvis faciliteten trinvis udførelse er slået til, og man ønsker at udføre den næste instruktion, kaldes udfør-funktionen. Det er i denne funktion, at datamaskinens funktion under udførelse af programmet simuleres. Når udfør-funktionen kaldes udskrives følgende tekst til højre på skærmen:

Programmet udføres

P - Pause i udførelsen

Nedenunder denne tekst, som er fast under hele udførelsen, fortæller skiftende tekster om, hvad maskinen er i gang med netop nu. Med hensyn til nærmere beskrivelse af modelmaskinens arbejdsmåde henvises til den tidligere gennemgåede og mere uddybende beskrivelse af modelmaskinens funktion og instruktionssæt.

Den ovenfor omtalte faste tekst, som står på skærmen under hele udførelsen hentyder til, at det er muligt midlertidigt at afbryde programudførelsen. Dette gøres ved at holde P-tasten nedtrykket, indtil en særlig "Pause menu" er kommet frem på skærmens højre side. I forbindelse med pause menuen har man mulighed for at ændre forskellige parametre, som har relation til udførelsen af programmet, såsom hastighed, trinvis udførelse og lyd. Når man har rettet parametrene, kan programudførelsen genoptages. Hele pause menuen er nærmere beskrevet i et senere afsnit(se side 10).

Brugervejledning

X - Sætter lyd til/fra

Som omtalt i forbindelse med udfør-funktionen fremkommer der under programmets udførelse tekster på højre side af skærmen, som beskriver, hvad modelmaskinen er i gang med netop nu. I forbindelse hermed er det muligt, at få maskinen til at udsende en bip-lyd hver gang der fremkommer en ny tekst. Dette kan være en hjælp, hvis man vil prøve at følge med i, hvad der foregår, ved at læse teksterne efterhånden som de kommer frem på skærmen. Bip-lyden slås til ved at trykke på X-tasten. Bip-lyden slås fra ved at trykke på X-tasten igen og så fremdeles. Som standardværdi ved opstart af programmet er bip-lyden slået fra.

Pause menuen

Som omtalt i forbindelse med udfør-funktionen er det muligt at foretage en midlertid afbrydelse af programudførelsen ved at hold P-tasten nedtrykket indtil nedenstående tekst kommer frem i højre side af skærmen:

UDFØRELSEN ER STOPPE

F - Fortsæt udførelse
H - Hastighedsændring
X - Sætter lyd til/fra
A - Afbryd efter udfø-
    relsen af igangvæ-
    rende ordre
T - Trinvis udf. til/fra

Nedenfor gennemgås de to funktioner, som ikke er kendt fra det allerede gennemgåede normale funktionsvalg.

F - Fortsæt udførelse

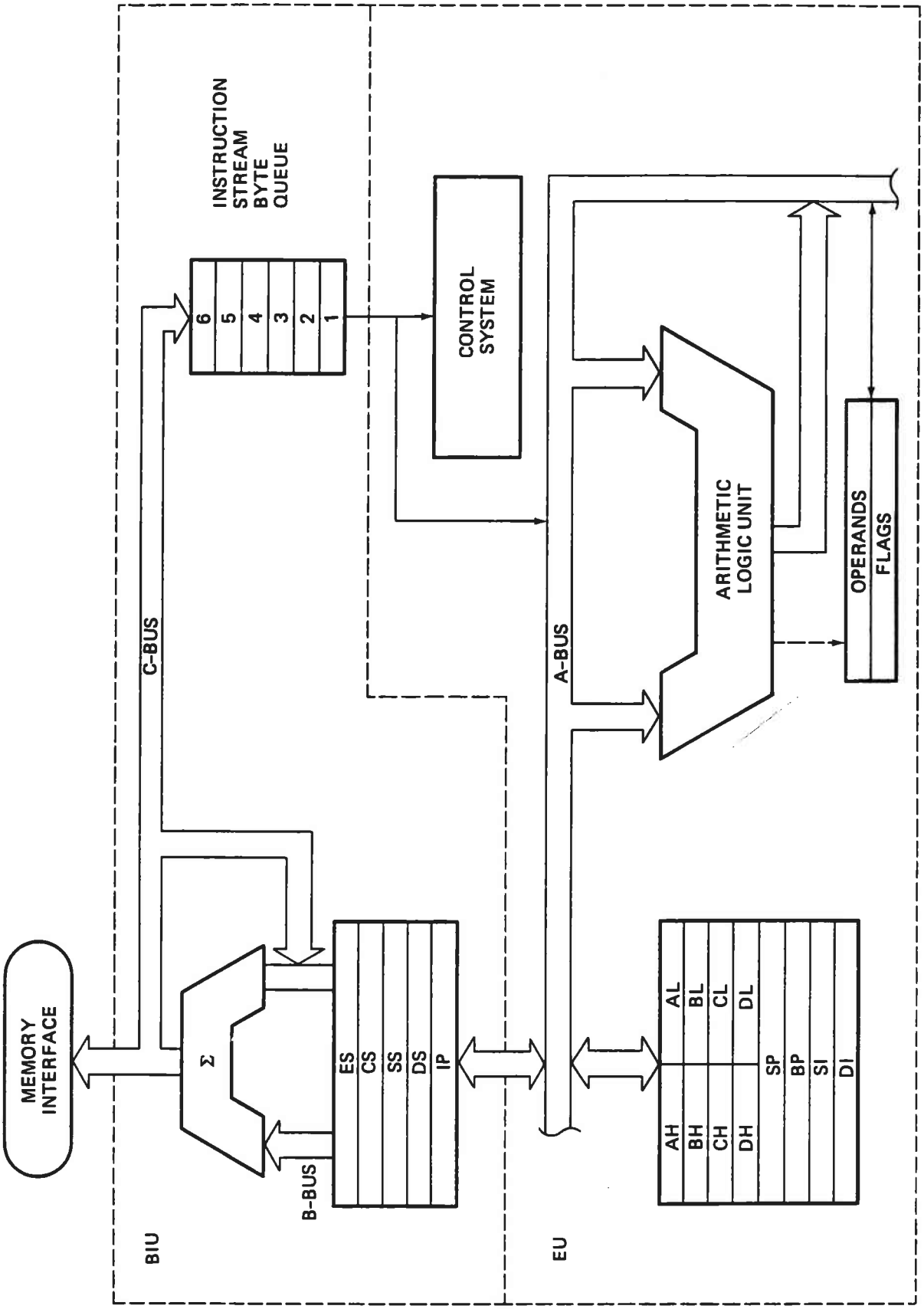
Når man har foretaget de ønskede ændringer af parametrene, kan man genoptage den afbrudte udførelse ved at trykke på F-tasten.

A - Afbryd efter udførelse af igangværende ordre

Den funktion kan anvendes i situationer, hvor man under programudførelsen opdager fejl, som bevirker, at resten af programudførelsen mister sin værdi. I stedet for at skulle sidde og vente på den bitre ende, kan man ved kald af denne funktion få maskinen til at stoppe programudførelsen, når den igangværende instruktion er færdig. Man er altså nødt til at trykke på F-tasten for at genoptage udførelsen, før programmet vil blive afbrudt.

4.3.a

8086 CPU arkitektur



8086 internal block diagram. (*Intel Corp.*)

Instruction Set for the Intel 8088

The Intel 8088 Instruction Set is explained in detail in this section. The instruction set for the Intel 8088 is the same as the instruction set for the Intel 8086.

Instruction Set Groupings

The 8086/8088 instructions can be grouped according to the purpose of the instruction. The six general classifications of instructions are

Data transfer
Arithmetic
Bit manipulation
String manipulation
Control transfer
Flag and processor control

Table 8088.1 shows how the individual instructions within these categories are grouped.

Table 8088.1
Instruction Set Groupings

Instruction	*Meaning*
Data Transfer	
IN	Input from port
LAHF	Load AH register with flags
LDS	Load DS register
LEA	Load effective address
LES	Load ES register
MOV	Move
OUT	Output to port
POP	Remove data from stack
POPF	Remove flags from stack
PUSH	Place data on stack
PUSHF	Place flags on stack
SAHF	Store AH into flag register
XCHG	Exchange
XLAT	Translate

Table 8088.1—cont.

Instruction	*Meaning*
Arithmetic	
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC	Add with carry
ADD	Add
CBW	Convert byte to word
CMP	Compare
CWD	Convert word to doubleword
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC	Decrement
DIV	Divide
IDIV	Integer divide
IMUL	Integer multiply
INC	Increment
MUL	Multiply
NEG	Negate
SBB	Subtract with carry
SUB	Subtract
Bit Manipulation	
AND	Logical AND on bits
NOT	Logical NOT on bits
OR	Logical OR on bits
RCL	Rotate left through carry
RCR	Rotate right through carry
ROL	Rotate left
ROR	Rotate right
SAL	Arithmetic shift left
SAR	Arithmetic shift right
SHL	Shift left
SHR	Shift right
TEST	Test bits
XOR	Logical exclusive-or on bits

Table 8088.1—cont.

Instruction	*Meaning*
String Manipulation	
CMPSB	Compare strings byte for byte
CMPSW	Compare strings word for word
LODSB	Load a byte from string into AL
LODSW	Load a word from string into AX
MOVSB	Move string byte-by-byte
MOVSW	Move string word-by-word
REP	Repeat
REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
SCASB	Scan string for byte
SCASW	Scan string for word
STOSB	Store byte in AL at string
STOSW	Store word in AX at string
Control Transfer	
CALL	Perform subroutine
INT	Software interrupt
INTO	Interrupt on overflow
IRET	Return from interrupt
JA	Jump if above
JAE	Jump if above or equal
JB	Jump if below
JBE	Jump if below or equal
JC	Jump on carry
JCXZ	Jump if CX=0
JE	Jump if equal
JG	Jump if greater
JGE	Jump if greater or equal
JL	Jump if less than
JLE	Jump if less than or equal
JMP	Jump
JNA	Jump if not above
JNAE	Jump if not above or equal
JNB	Jump if not below
JNBE	Jump if not below or equal
JNC	Jump on no carry

Table 8088.1—cont.

Instruction	*Meaning*
JNE	Jump if not equal
JNG	Jump if not greater than
JNGE	Jump if not greater than or equal
JNL	Jump if not less than
JNLE	Jump if not less than or equal
JNO	Jump on no overflow
JNP	Jump on no parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LOOP	Loop
LOOPE	Loop while equal
LOOPNE	Loop while not equal
LOOPNZ	Loop while not zero
LOOPZ	Loop while zero
RET	Return from subroutine
Flag and Processor Control	
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
ESC	Escape
HLT	Halt
LOCK	Lock bus
NOP	No operation
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
WAIT	Wait

Dekodningstabel for 8086-maskininstruktioner

Hi \ Lo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD b,f,r/m	ADD w,f,r/m	ADD b,t,r/m	ADD w,t,r/m	ADD b,ia	ADD w,ia	PUSH ES	POP ES	OR b,f,r/m	OR w,f,r/m	OR b,t,r/m	OR w,t,r/m	OR b,i	OR w,i	PUSH CS	
1	ADC b,f,r/m	ADC w,f,r/m	ADC b,t,r/m	ADC w,t,r/m	ADC b,i	ADC w,i	PUSH SS	POP SS	SBB b,f,r/m	SBB w,f,r/m	SBB b,t,r/m	SBB w,t,r/m	SBB b,i	SBB w,i	PUSH DS	POP DS
2	AND b,f,r/m	AND w,f,r/m	AND b,t,r/m	AND w,t,r/m	AND b,i	AND w,i	SEG =ES	DAA	SUB b,f,r/m	SUB w,f,r/m	SUB b,t,r/m	SUB w,t,r/m	SUB b,i	SUB w,i	SEG =CS	DAS
3	XOR b,f,r/m	XOR w,f,r/m	XOR b,t,r/m	XOR w,t,r/m	XOR b,i	XOR w,i	SEG =SS	AAA	CMP b,f,r/m	CMP w,f,r/m	CMP b,t,r/m	CMP w,t,r/m	CMP b,i	CMP w,i	SEG =DS	AAS
4	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6	PUSHA	POPA	BOUND w,f,r/m						PUSH w,i	IMUL w,i	PUSH b,i	IMUL b,i	INS b	INS w	OUTS b	OUTS w
7	JO	JNO	JB/ JNAE	JNB/ JAE	JE/ JZ	JNE/ JNZ	JBE/ JNA	JNBE/ JA	JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG
8	Immed b,r/m	Immed w,r/m	Immed b,r/m	Immed is,r/m	TEST b,r/m	TEST w,r/m	XCHG b,r/m	XCHG w,r/m	MOV b,f,r/m	MOV w,f,r/m	MOV b,t,r/m	MOV w,t,r/m	MOV sr,f,r/m	LEA	MOV sr,t,r/m	POP r/m
9	XCHG AX	XCHG CX	XCHG DX	XCHG BX	XCHG SP	XCHG BP	XCHG SI	XCHG DI	CBW	CWD	CALL l,d	WAIT	PUSHF	POPF	SAHF	LAHF
A	MOV m→AL	MOV m→AX	MOV AL→m	MOV AX→m	MOVS	MOVS	CMPS	CMPS	TEST b,i,a	TEST w,i,a	STOS	STOS	LODS	LODS	SCAS	SCAS
B	MOV i→AL	MOV i→CL	MOV i→DL	MOV i→BL	MOV i→AH	MOV i→CH	MOV i→DH	MOV i→BH	MOV i→AX	MOV i→CX	MOV i→DX	MOV i→BX	MOV i→SP	MOV i→BP	MOV i→SI	MOV i→DL
C	Shift b,i	Shift w,i	RET. (i+SP)	RET	LES	LDS	MOV b,i,r/m	MOV w,i,r/m	ENTER	LEAVE	RET. l.(i+SP)	RET l	INT Type 3	INT (Any)	INTO	IRET
D	Shift b	Shift w	Shift b,v	Shift w,v	AAM	AAD		XLAT	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
E	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCXZ	IN b	IN w	OUT b	OUT w	CALL d	JMP d	JMP l,d	JMP si,d	IN v,b	IN v,w	OUT v,b	OUT v,w
F	LOCK		REP	REP Z	HLT	CMC	Grp 1 b,r/m	Grp 1 w,r/m	CLC	STC	CLI	STI	CLD	STD	Grp 2 b,r/m	Grp 2 w,r/m

where:

mod□r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR	—	SAR
Grp 1	TEST	—	NOT	NEG	MUL	IMUL	DIV	IDIV
Grp 2	INC	DEC	CALL id	CALL l,id	JMP id	JMP l,id	PUSH	—

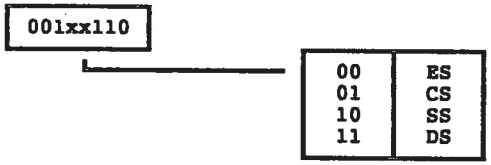
b = byte operation
d = direct
f = from CPU reg
i = immediate
ia = immed. to accum.
id = indirect
is = immed. byte, sign ext.
l = long ie. intersegment
m = memory
r/m = EA is second byte
si = short intrasegment
sr = segment register
t = to CPU reg
v = variable
w = word operation
z = zero

OBJECT KODE FORMAT.

8088/8086 Object koderne er sammensat af forskellige dele, alt efter instruktions typen. De instruktioner, der ikke kræver operander, fylder som regel 1 byte, hvorimod instruktioner med operander, kan fylde fra 2 til 7 bytes.

Nedenstående er generelt for alle instruktionerne, men der findes instruktioner, der har sit eget specielle format.

Segment override byte.
Dette er den først byte, der kan være i en instruktion. Den fortæller at man ønsker at benytte et andet segment register end hvad default angiver. Den kan kun forekomme sammen med instruktioner, med memory adressering.



Instruktioners object kode.
Denne byte fortæller, hvilken instruktion, der er tale om. I denne byte kan der forekomme nogle bits, der har betydning for instruktionens format. Disse kan bla. være:

D. (er placeret som bit nummer 1).
Denne bit angiver, hvad der er source og hvad der er destination (højre og venstre operand).

D = 1:
Så angives destinationen af REG-feltet og source af MOD og R/M feltet i den efterfølgende MODRM byte.

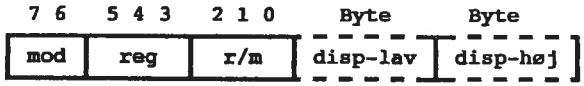
D = 0:
Så angives destinationen af MOD og R/M felterne og source angives af REG feltet, i den efterfølgende MODRM byte.

W. (er placeret som bit nummer 0).
Denne bit angiver om operanderne er bytes eller words.

W = 0: Er operanderne BYTES.
W = 1: Er operanderne WORDS.

MORDM byten/bytes.
Denne (fylder 1,2 eller 3 bytes) benyttes til at specificere operanden/operanderne i instruktionen. Hvordan denne opfattes er lidt forskellig fra instruktion til instruktion.

Den først byte (MODRM), ser generelt sådan ud:



MOD feltet angiver om MODRM efterfølges af et displacement til instruktionen (samt om det er et 8 eller 16 bits displacement), eller om r/m feltet skal opfattes som et "REG" felt.

Mod	Displacement.
00	Displacement findes ikke *
01	Der findes et 8 bits displacement
10	Der findes et 16 bits displacement
11	R/M feltet opfattes som et "REG" felt.

NB: Hvis MOD = 00 og R/M feltet er 110, findes et 16 bit displacement, der opfattes som EA (addr-lav.addr-høj).

REG feltet angiver hvilket register, der skal indgå som en operand. Bemærk at R/M feltet også kan opfattes som et REG felt (hvis MOD = 11).

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

NB: I visse instruktioner, er REG feltet i selve object kode byten, f.eks POP og PUSH instruktionerne. Betydningen er dog den samme som i skemaet.

R/M feltet angiver sammen med MOD feltet, hvilken memory adresseringsform, der eventuelt skal benyttes (MOD feltet <> 11). Nedenstående skema viser hvilke kombinationer af MOD og R/M feltet, der kan forekomme.

	MOD=00	MOD = 01	MOD = 10	MOD=11	
R/M	Offset	Offset	Offset	W=1	W=0
000	BX + SI	BX + SI + D8	BX + SI + D16	AX	AL
001	BX + DI	BX + DI + D8	BX + DI + D16	CX	CL
010	BP + SI	BP + SI + D8	BP + SI + D16	DX	DL
011	BP + DI	BP + DI + D8	BP + DI + D16	BX	BL
100	SI	SI + D8	SI + D16	SP	AH
101	DI	DI + D8	DI + D16	BP	CH
110	Note 1.	BP + D8	BP + D16	SI	DH
111	BX	BX + D8	BX + D16	DI	BH

Note 1: Hvis Mod er 00 og R/M er 110, efterfølges MODRM byten af to bytes, der angiver et offset.

Eksempel 1:
Object kode for MOV DX,[BX+1]

Formatet er MOV reg,mem, og object koden for dette format er:

100010dw	mod reg r/m	disp8/disp16

For ovenstående instruktion, vil object kode være:

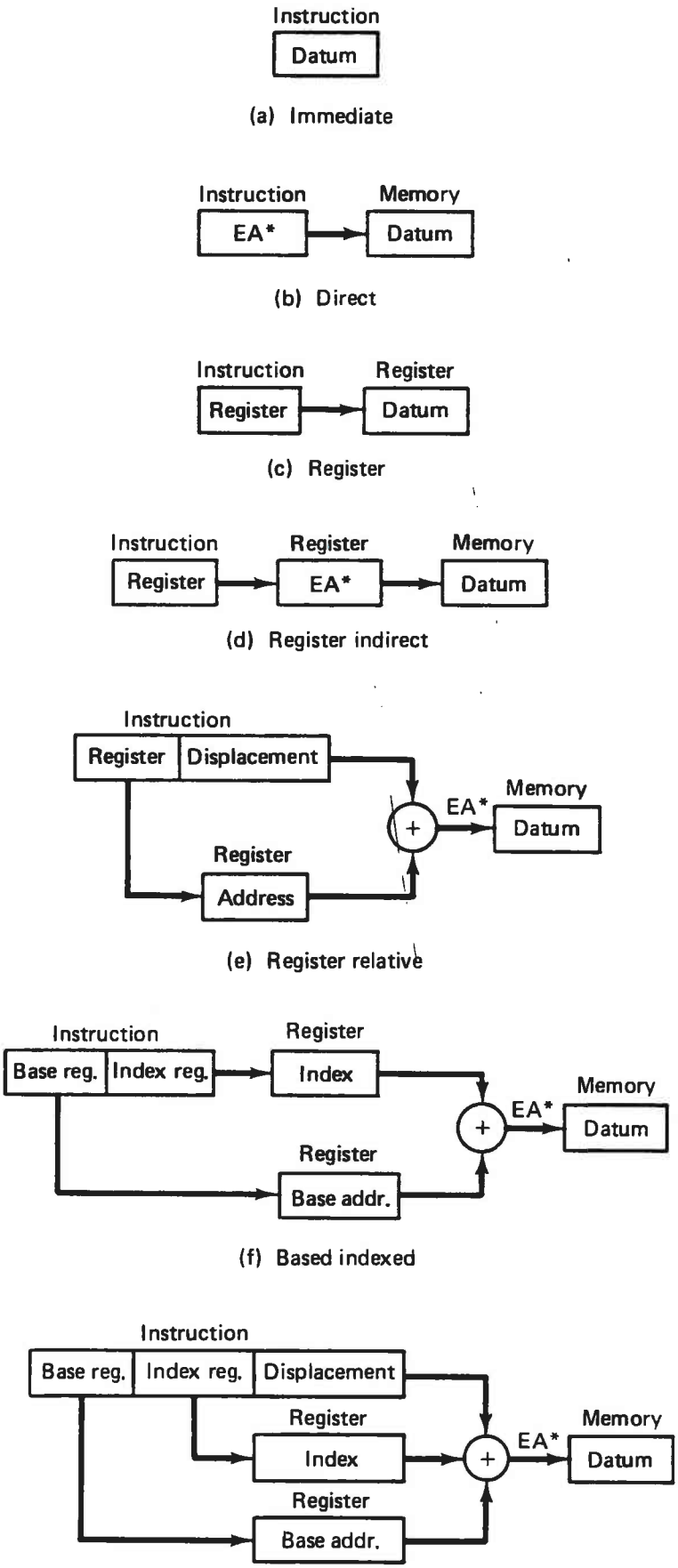
10001011	01 010 111	00000001

= 8BH 57H 01H.

- 8 bits displacement.
- Når MOD = 01, betyder 111 i R/M, BX+disp8.
- 010 i REG angiver DX.
- MOD = 01 angiver disp8.
- W bit = 1, operanderne er words.
- D bit = 1 angiver at REG angiver destinationen og MOD og R/M angiver sourcen.

8086 Architecture

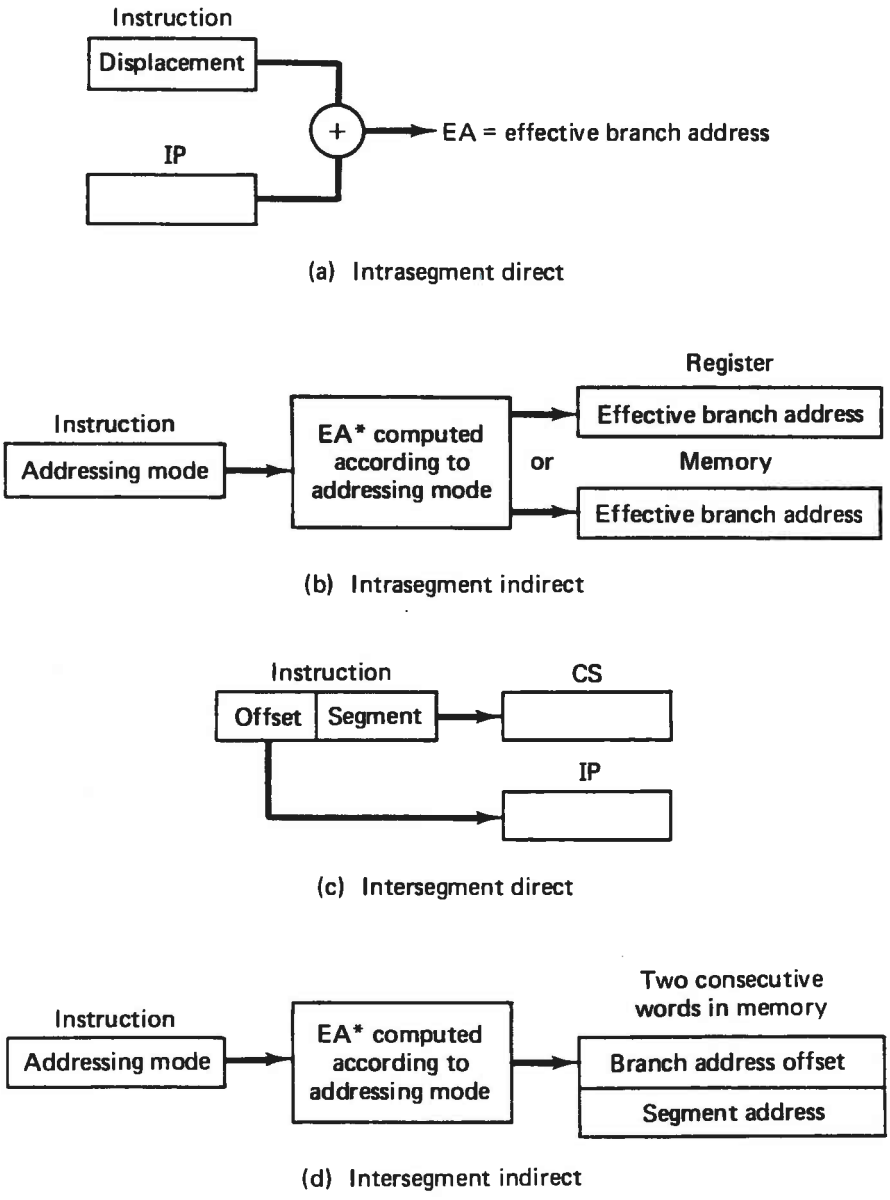
Data-related addressing modes.



*EA is added to 16_{10} times the contents of the appropriate segment register.

8086 Architecture

Branch-related addressing modes.



*EA is added to 16_{10} times the contents of the appropriate segment register.

DOS-KASSEN

Hvordan virker interrupts?

Interrupts er uundværlige - men irriterende.

Af Klaus Møllgaard

Med den hastigt stigende udbredelse af "add-on boards" (blandt andet multimedia-hardware) til opgradering af PC'er vil et stigende antal brugere løbe ind i både interrupt-kanal, DMA-kanal og adresse-konflikter.

Interrupts er væsentlige for, at en PC overhovedet kan fungere, og det følgende er ment som en kort indføring i interrupt-funktionen, de problemer interrupt-konflikter kan give for PC-brugeren og nogle retningslinier for, hvordan man undgår interrupt-konflikter.

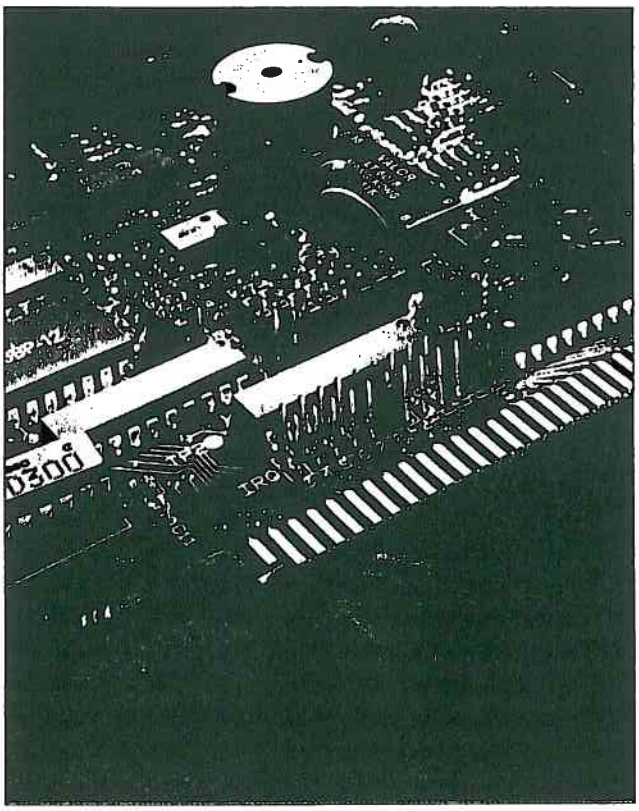


Foto: Morten Holtum Nielsen

Valg af IRQ-kanal kan foretages ved at jongle-re med jumper-indstillingen på kortet. I visse tilfælde kan IRQ-kanalen ændres med et program.

Hvordan håndterer vi afbrydelser?

Interrupt betyder afbrydelse og for at forstå, hvad et interrupt er, er det vigtigt at forstå, hvordan PC'en håndterer afbrydelser. Hverdagen er fyldt med uregelmæssige, ikke forudsigelige afbrydelser, og hvordan håndterer vi selv dem? Søndag morgen - koncentreret om avislæsning - telefonen ringer, samtidig med at det banker på døren, og den mindste skriger på et bleskift. Tre samtidige afbrydelser i det, der var hovedopgaven: at læse avisen. Afbrydelserne skal først prioriteres efter betydning og dernæst klares én for én, før avislæsningen kan genoptages.

PC'en håndterer uregelmæssige eller asynkrone afbrydelser på samme måde - blot som oftest noget hurtigere og med en lidt mere rigid prioritering. En asynkron afbrydelse kan være et tryk på tastaturet, en bevægelse af musen, søgning på disken, opdatering af skærmen og så videre. Hvis PC'en regelmæssigt skulle ud og checke samtlige I/O-enheder for, om de muligvis havde noget, de skulle have hjælp til, ville CPU-regnekraften nedsættes betragteligt.

Interrupt request-kanaler

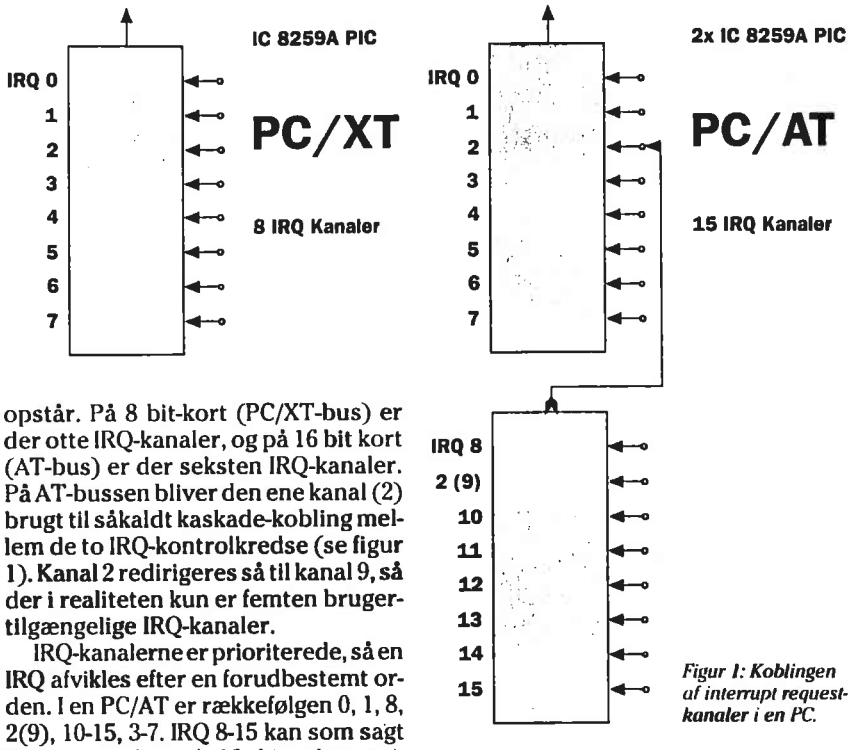
Computeren har derfor indbygget en række interrupt-kanaler, der kan tildeles de forskellige perifere enheder til at håndtere sådanne asynkrone afbrydelser. Her kan de så rejse en interrupt request (IRQ) - en anmodning om afbrydelse af og hjælp fra CPU'en. Computerens CPU udfører sine tildelte opgaver og instruktioner på en skemalagt og sekventiel måde indtil en IRQ dukker op. CPU'en stopper, hvad den er i gang med, behandler (og udfører) interrupt'en og genoptager herefter sin sekventielle afvikling af de tidligere instruktioner.

Interrupt-problemerne på dagens computere udspringer (endnu engang) af IBM's manglende fremsyn med hensyn til udviklingen inden for den personlige computer (det er altid nemt at være bagklog). IBM "glemte" nemlig at lære PC'en at dele IRQ-kanaler på ISA-bussen. Utallige er de timer, PC-ejere har brugt på at løse problemer, der i sidste ende har vist sig at ligge i en interrupt-konflikt; for eksempel mellem de serielle porte, modem og en serielt tilsluttet mus på IRQ 3 eller 4. Eller VGA/netkort-konflikt på IRQ 2, der fungerer nogenlunde på samme IRQ, indtil man kalder Windows. EISA- og MCA-bus systemkort tillader til en vis grad deling af IRQ mellem flere funktioner og overkommer dermed en af ISA-buskonstruktionens store bagdele.

Soft- og hardware IRQs

Der er i alt 256 IRQ-kanaler i en PC. Alle IRQ-kanaler kan adresseres via software, men de første 8/16 (IRQ 0-15) er også addressérbare direkte via ISA-bussen via tilsluttet hardware, og det er som regel her, problemerne ▷

DOS-KASSEN



Figur 1: Koblingen af interrupt request-kanaler i en PC.

opstår. På 8 bit-kort (PC/XT-bus) er der otte IRQ-kanaler, og på 16 bit kort (AT-bus) er der seksten IRQ-kanaler. På AT-bussen bliver den ene kanal (2) brugt til såkaldt kaskade-kobling mellem de to IRQ-kontrolkredse (se figur 1). Kanal 2 redirigeres så til kanal 9, så der i realiteten kun er femten brugertilgængelige IRQ-kanaler.

IRQ-kanalerne er prioriterede, så en IRQ afvikles efter en forudbestemt orden. I en PC/AT er rækkefølgen 0, 1, 8, 2(9), 10-15, 3-7. IRQ 8-15 kan som sagt kun anvendes af 16 bit ekspansionsslot/kort, hvilket giver en stor belastning af de første otte kanaler. Her ligger både COM- og LPT-porte, floppydisk-controller, tastatur og systemtimer, mens harddisk-controller og FPU er flyttet højere op (se tabel 1). For at opnå fuld kompatibilitet med gamle maskiner, har mange producenter indtil nu valgt at lave kort i 8 bit-udgaver med deraf følgende potentielle konflikter kort imellem og mellem kort og kommunikationsporte. Det er ved at ændre sig i takt med, at 8 bit-busmaskiner er på vej ud.

Indstilling af en IRQ

På et kort kan IRQ-kanalvalg enten foregå via en fysisk justering med en jumper, der flyttes til den ønskede indstilling eller via et program, der læser den valgte kanal ind i en lille hukommelse på kortet. Jumperindstillingen er måske den mest overskuelige og den "ældste" måde at gøre det på. Den har den ulempe, at man skal splitte computeren ad for at ændre (eller kontrollere) indstillingen. Den nyere software-kontrollerede indstilling (af hardware interrupt) er meget nemmere og vinder mere og mere indpas på markedet. Her kalder man et lille program, der som regel er i stand til at fortælle brugeren, hvilke IRQ-kanaler, der er ledige til den nye enhed inden valget foretages.

Hvis man vælger en IRQ, der allerede er besat af en anden enhed, kan der opstå flere forskellige problemer, afhængig af hvilke to enheder, der konflikter med hinanden. Som regel giver konflikter sig udtryk i, at computeren låser uhjælpeligt fast og må boot'es om. Hvis computeren låser allerede under opstarten, kan man prøve at starte fra diskette og så finde problemet - ellers kan det være nødvendigt at fjerne de installerede kort et efter et; mindst betydende kort fjernes først - selvfølgelig. Andre årsager til problemer kan være BIOS I/O adresse-konflikt, memory adresse-konflikt eller sjældnere DMA kanal-konflikt (Direct Memory Access).

Husk at ...

Periferi-kort kommer med en standard indstilling af jumpere og DIP-switches, så de umiddelbart kan anvendes i de fleste computere uden problemer. Visse programmer, der understøtter fx lydkort, kan selv detektere de adresser og kanaler, som kortene anvender, mens andre programmer skal have informationen ved installation eller programstart. Det er derfor en god ide ved installation af ekstra kort at notere de forskellige indstillinger, så du ikke skal splitte maskinen ad hver gang, du bliver spurgt.

Med det stigende antal "add-on boards" (lyd, video, scannere, CD-ROM, osv.) vil man før eller siden løbe ind i problemer eller overvejelser med hensyn til fordelingen af IRQ-kanaler - især med for mange 8 bit-boards, hvor 16 bit-boards giver adgang til mange flere ikke reserverede kanaler. Betal hellere lidt mere for 16 bit-udgaver af kort; den investering betaler sig af flere grunde i længden. For alle 16 bit-kort bør du vælge IRQ i den høje ende, så 8 bit-kortene kan bruge de andre.

Du kan godt tildele et kort en interrupt fra en kommunikationsport, du ikke anvender (for eksempel LPT2 eller COM2), men hvis du har tre eller fire serielle eller parallelle porte eller modem, så husk at tage højde for, at port 1 og 3 samt port 2 og 4 som regel er sat op til at dele IRQ-kanal. Brug ikke IRQ 2 til noget, hvis du har VGA-kort eller netkort, det *kan* give problemer. □

Tabel 1

IRQ kanaler på PC/AT

	Fast anvendelse	Normal anvendelse
0	System-timer	
1	Tastatur	
2 (9)		Generel (VGA/netkort)
3		Generel (COM2/COM4)
4		Generel (COM1/COM3)
5		Generel (LPT2/LPT4)
6		Floppy controller
7		Generel (LPT1/LPT3)
8	Real time clock	
10		Generel
11		Generel
12		Generel
13	Coprocessor (FPU)	
14		Harddisk-controller
15		Generel

4.3.g

Anvendelse af 8086-registrene

Arbejdsregistrene:

AX **(Accumulator-register)**
Anvendes som allround arbejdsregister.
Kan opdelses i to "halve" registre (Ah og Al), der kan benyttes som selvstændige 8-bits registre.
Multiplikation og division kan kun udføres med dette register som "1. operant", idet DX-registret dog medtages ved 16-bit beregninger, således at DX-AX tilsammen fungerer som et 32-bits register.
IO-instruktioner knytter sig også til dette register.
Instruktionen CVB anvendes til at udvide værdi (med fortegn) i AL til 16-bit i AX. CWD anvendes til at udvide værdi (med fortegn) i AX til 32-bit i registrer-paret DXAX.

BX **(Base-register)**
Registret kan anvendes til memory-adressering (evt. sammen med SI- eller DI-registret).
Registret kan ligesom AX benyttes som to 8-bits registre.
Registret kan anvendes som ekstra arbejdsregister (register-variabel).

CX **(Counter-register)**
Registret anvendes som counter-register i forbindelse med LOOP-instruktionerne samt nogle af streng-instruktionerne. I disse instruktioner kan man ikke selv angive hvilke registre, der skal anvendes.
I forbindelse med shift-instruktionerne anvendes CL-registret til, at angive hvor mange "pladser" der skal shiftes. Det er heller ikke her muligt, at angive et andet register.
Registret kan ligesom AX benyttes som to 8-bits registre.
Registret kan anvendes som ekstra arbejdsregister (register-variabel).

DX **(Destination-register)**
Registret anvendes sammen med AX i forbindelse med multiplikation og division.
I forbindelse med en række funktionskald (Software-interrupt) anvendes registret sammen med DS (Datasegment) som startadresse (pointer) til memory-buffer.
Registret kan ligesom AX benyttes som to 8-bits registre.
Registret kan anvendes som ekstra arbejdsregister (register-variabel).

Adresse-/pointer-registre.

BP **(Base-pointer)**
Registret anvendes primært til memory-adressering på stakken (evt. sammen med SI- eller DI-registret). **Når BP-registret anvendes vil SS registret blive anvendt som segmentregister** (også samnen med SI/DI), hvis der ikke bliver angivet et segment-register, imodsætning til BX-, SI- , DI-registrene der anvender DS registret som segmentregister.
Registret kan **kun** benyttes som 16-bit register.
Registret kan anvendes som et ekstra arbejdsregister (register-variabel).

SI **(Source-index)** **(source = kilde/afsender)**
Registret anvendes primært til memory-adressering.
Registret anvendes i forbindelse med streng-instruktionerne sammen med DS-segmentregistret. I disse instruktioner kan man ikke selv angive hvilke registre, der skal anvendes.
Registret kan **kun** benyttes som 16-bit register.
Registret kan anvendes som et ekstra arbejdsregister (register-variabel).

DI **(Destination-index)** **(destination = modtager)**
Registret anvendes primært til memory-adressering.
Registret anvendes i forbindelse med streng-instruktionerne sammen med ES-segmentregistret. I disse instruktioner kan man ikke selv angive hvilke registre, der skal anvendes.
Registret kan **kun** benyttes som 16-bit register.
Registret kan anvendes som et ekstra arbejdsregister (register-variabel).

Specielle adresse-/pointer-registre

SP **(Stak-pointer)**
Registret anvendes sammen med SS-segmentregistret som pointer til staktoppen.
Registret rettes "automatisk" ved anvendes af instriktionerne PUSH, POP, CALL, RET, INT og IRET.
Registeret må ikke benyttes som arbejdsregister.

IP **(Instruktions-pointer)**
Registret anvendes til udpegning af instruktioner sammen med CS-segmentregistret.
Registret "flyttes" automatisk frem til næste instruktion, idet der kan foretages justereing med jump-instruktioner, call, ret m.fl.
Registret må ikke benyttes som arbejdsregister.

Segmentadresse-registre

CS **(Code-segment)**
Registret anvendes til udpegning af instruktioner sammen med IP-segmentregistret.

DS **(Data-segment)**
Registret anvendes til udpegning af data.

SS **(Stak-segment)**
Registret anvendes til udpegning af staktop sammen med SP-registret.

ES **(Ekstra-segment)**
Registret kan anvendes anvendes til udpegning af data, idet der så normalt anvendes segment-overwrite.
Registret anvendes sammen med DI-registeret i forbindelse med nogle af streng-instruktionerne.

Flag-register

F **(Flag)**
Registret inderholder dels resultats-flag og dels kontrol-flag.
Resultatsflagene sættes i forbindelse med udførslen af aritmetiske og logiske instruktioner. Flagene kan efterfølgende aftestes med betingede jumpinstruktioner. Carry-flaget kan evt. sættes on/off med STC og CLS.
Kontrolflagene sættes med specielle instruktioner og anvendes til styring af cpu-ens arbejdsmåde.
Der er f.eks. kontrolflag til at lukke og åbne for interruptsystemet.
I forbindelse med streng-instruktionerne findes et kontrolflag, som styrer den "retning" der arbejdes med (om adresser tælles op eller de tælles ned).

5.

DOS-maskinen - operativsystemkernen

Indhold:

DOS servicefunktioner

I DOS manualerne findes en beskrivelse af servicefunktionerne ordnet efter deres nummer; her skal funktionskaldene tabelleres efter deres anvendelsesområder.

De allerfleste ikke-reserverede funktionskald er underfunktioner til interrupt 21H, men INT 20H, 25H, 26H og 27H kan også være nyttige. INT 20H og 27H afslutter en programudførelse, men INT 27H på en sådan måde, at programmet forbliver i RAM lageret efter afbrydelsen. INT 25H og 26H hhv. læser og skriver sektorer på diskette ved hjælp af absolut adressering.

Tastatur I/O. INT 21H.

AH=01H	Vent til næste tegn og læs dette med ekko til skærm. Resultat i AL. Hvis Ctrl-Break, udføres INT 23H.
AH=06H	DL=FFH. Hvis tegn er parat, hentes dette ind i AL og ZF=0. Hvis ikke, sættes ZF=1. Ctrl-Break ignoreres.
AH=07H	Som AH=01H, men uden ekko og uden Ctrl-Break detektering.
AH=08H	Som AH=01H, men uden ekko.
AH=0AH	Læs til buffer ved DS:DX+2. DS:DX peger på bufferlængden.
AH=0BH	AH sættes til FFH, hvis tegn er parat; ellers 00H. Hvis Ctrl-Break, udføres INT 23H.
AH=0CH	Tøm tastaturbuffer og udfør en af funktionerne 01H, 06H, 07H, 08H eller 0AH ovf. afhængigt af AL's indhold.
AH=33H	Undersøg eller aktiver Ctrl-Break. Kun DOS 2.

Monitor I/O. INT 21H.

AH=02H	Vis tegnet i DL. Ctrl-Break medfører INT 23H.
AH=06H	DL(<FFH) vises. Ctrl-Break ignoreres.
AH=09H	Vis tekst i buffer DS:DX frem til $ tegn. Iøvr. som AH=02H

Skriver I/O. INT 21H.

AH=05H	Skriv tegnet i DL. Kun 1 skriver understøttes.

Seriel port. INT 21H.

AH=03H	Vent til næste tegn og anbring det i AL. Ingen status information; jfr. ROM BIOS INT 14H i tabel 3.5.3.
AH=04H	Send tegn i AL til seriel port. Iøvr. som AH=03H.

RAM lager. INT 21H. Kun DOS 2.00 og følgende versioner.

AH=48H	Skaf om muligt BX paragraffer. Hvis muligt, returneres 1. paragrafnummer i AX; ellers returneres største frie blok i BX.
AH=4AH	Giv ES segmentet den størrelse, som BX angiver. Iøvr. som for AH=48H.
AH=49H	Aflever lager fra ES og frem til den frie pulje.

Program håndtering. INT 21H.

AH=4BH	Eksekver og/eller load program, hvis filnavn er i DS:DX og hvis parameter blok (se DOS manual) er i ES:BX. Hvis AL=0, konstrueres et PSP og programmet loades og eksekveres; hvis AL=3, udføres kun load. Kun under DOS 2.
AH=4DH	Hent returkode i AL. Bør altid udføres efter 4BH. Kun DOS 2.
AH=26H	Opret nyt programsegment.
AH=00H	Afslut programudførelsen uden returkode. Samme som INT 20H.
AH=4CH	Afslut programudførelsen med returkode. Kun under DOS 2.
AH=31H	Afslut programudførelsen og forbliv resident. Kun under DOS 2. Samme som INT 27H, dog kan returkode sættes (i AL).

DOS SERVICEFUNKTIONER FORTSAT

Filhåndtering. INT 21H. DOS 1.10 og følgende versioner

AH=0EH	Gør disk DL (0=A,1=B osv.) til den aktuelle. AL sættes til antal drev ialt.
AH=19H	Hent nummer på aktuelle drev i AL.
AH=0DH	Initialiser disk og nedlæg alle buffere.
AH=16H	Opret ny fil med FCB i DS:DX. AL=0, hvis det lykkedes; ellers AL=FFH.
AH=0FH	Åbning af fil. FCB fyldes ud. Iøvr. som AH=16H.
AH=10H	Lukning af fil. Iøvr. som ovf.
AH=13H	Nedlæg en lukket fil. Iøvr. som AH=16H.
AH=17H	Giv fil et nyt navn. Iøvr. som AH=16H.
AH=29H	Undersøg filnavn for bl.a. filtype, ?- og *-tegn og opret en FCB på adresse ES:DI for filen uden at åbne denne.
AH=11H	Opsøg 1. fil i en uåbnet FCB med "?"-tegn i filnavn.
AH=12H	Opsøg næste fil i en uåbnet FCB med "?"-tegn i filnavn.
AH=1BH	Hent FAT information for det aktuelle drev.
AH=1CH	Hent FAT information for et udvalgt drev.
AH=1AH	Opret buffer med adresse (DTA) angivet i DS:DX.
AH=14H	Sekventiel læsning af post. DS:DX peger på åben FCB. Unormal afslutning: AL=01-ingen data; AL=02-ikke plads nok i DTA; AL=03-kun del af post fundet.
INT 25	Læs CX sektorer ind i DS:BX. (Absolut adressering).
AH=15H	Sekventiel skrivning af post. Iøvr. som AH=14H; dog betyder AL=01: fuld diskette og AL=02: ikke plads i DTA.
INT 26	Skriv CX sektorer fra DS:BX. (Absolut adressering).
AH=24H	Sæt pegepind til næste ikke-sekventielle post.
AH=21H	Ikke-sekventiel læsning af en post. Iøvr. som AH=14H.
AH=27H	Ikke-sekventiel læsning af en blok af poster. Iøvr. som ovf.
AH=22H	Ikke-sekventiel skrivning af en post. Iøvr. som AH=14H.
AH=28H	Ikke-sekventiel skrivning af blok m. CX poster.Iøvr.som ovf.
AH=23H	Hent antal poster i filen.
AH=28H	CX=0. Giv filen en ny størrelse.
AH=2EH	Slå "læs-efter-skriv verifikation" til(AL=1) eller fra(AL=0)

Fil håndtering. 21H. Kun DOS 2.00 og følgende versioner.

AH=3CH	Opret ny fil som specificeret i DS:DX og med attribut i CX. Fil "handle" returneres i AX.
AH=3DH	Åbning af fil. AL=0,1,2: read only, write only, read/write.
AH=3EH	Lukning af fil med handle i BX.
AH=41H	Nedlæg en lukket fil specificeret i DS:DX.
AH=56H	Giv fil (DS:DX) et nyt navn (ES:DI).
AH=4EH	Opsøg første fil med "?"-tegn i filnavn.
AH=4FH	Opsøg næste fil med "?"-tegn i filnavn.
AH=45H	Giv fil en ny handle.
AH=46H	Lad handle pege på en anden fil.
AH=2FH	Hent buffer adresse (DTA) i ES:BX.
AH=3FH	Læs CX byte ind i buffer ved DS:DX fra fil med handle i BX.
AH=40H	Som AH=3F, men skrivning.
AH=42H	Sæt pegepind til næste datum, der læses eller skrives.
AH=44H	Hent eller sæt status.
AH=39H	Opret underkatalog (MKDIR) med navn ved DS:DX.
AH=3AH	Nedlæg underkatalog (RMDIR) med navn ved DS:DX.
AH=3BH	Skift til nyt katalog (CHDIR) med navn ved DS:DX.
AH=47H	Hent aktuelle katalog ind i DS:SI for drev angivet i DL.
AH=36H	Hent oplysninger om ledig plads på disketten i drev DL.
AH=43H	Hent (AL=0) eller sæt (AL=1) filattributter i CX.
AH=57H	Hent (AL=0) eller sæt (AL=1) dato:klokkeslet i DX:CX.
AH=54H	Hent "læs-efter-skriv verifikation" status i AL.

Specielle funktioner. INT 21H.

AH=2AH	Hent dato i CX:DX.
AH=2CH	Hent klokkeslet i CX:DX.
AH=2BH	Sæt dato i CX:DX.
AH=2DH	Sæt klokkeslet i CX:DX.
AH=35H	Hent interrupt vektor nr. AL i ES:BX. Kun DOS 2.
AH=25H	Sæt interrupt vektor nr. AL til værdien i DS:DX.
AH=38H	Hent nationale oplysninger (mønt, dato format mv.) anbragt på adresse DS:DX. Kun DOS 2.
AH=30H	Hent DOS versionens nummer i AL:AH. Kun DOS 2.

BIOS INTERRUPTS.

BIOS interrupts går fra 00H til 1FH og DOS's fra 20H.
I denne oversigt er = anvendt som inddata til BIOS og
:= anvendt som uddata fra BIOS.

INT nr. Hex.	Dec.	Kommentarer og parametre.
0		Genereres af CPU'en ved division med 0.
1		Single step - anvendes af DEBUG el. SYMDEB.
2 NMI		NON MASKABLE INT. Normalt memory paritetsfejl.
3		Breakpoints. Anvendes af debuggere.
4		Overflow - (anvendes normalt ikke - retur).
5		Print Screen.
6		Reserveret.
7		- - " - - .
8 (IRQ 0)		Timer of Day. Brug 1CH i stedet. 18,2 * pr sek
9 (IRQ 1)		Tastatur interrupt.
A til F er		maskinafhængig - pointer til BIOS rutiner.
A (IRQ 2)		Vertikal retrace for EGA- og VGA- kort.
B (IRQ 3)		Seriel COM2 printer controller.
C (IRQ 4)		- " - COM1 - - - - " - - - - .
D (IRQ 5)		Parallel LPT2 - - - - " - - - - .
E (IRQ 6)		Hard disk controller.
F (IRQ 7)		Parallel LPT1 printer controller.

Skærmen.	AH.	
10H	0	Sæt skærm mode. (Se side 228). AL skal indeholde en af følgende værdier : 0 = 40 * 25 S/H. 1 = 40 * 25 farve (8/16). 2 = 80 * 25 S/H. 3 = 80 * 25 farve (8/16). 4 = 320 * 200 farve (4). 5 = 320 * 200 S/H. 6 = 640 * 200 S/H. 7 = 80 * 25 for monokrom skærm. 0DH = 320 * 200 farve (16). 0EH = 640 * 200 farve (16). 0FH = 640 * 350 S/H. 10H = 640 * 350 farve (16). 11H = 640 * 480 farve (2). 12H = 640 * 480 farve (16). 13H = 320 * 200 farve (256).
10H	1	Sæt cursor mode/type. (Se side 229) Cursorens blink kan ikke standses, da det er sat hardware mæssigt, men kan redefineres da den på en monokrom skærm er 14 linier høj og 8 linier høj på en grafisk skærm. F.eks. CH og CL begge = 0DH vil kun sidste linie blive anvendt. CH = 0DH og CL = 00H vil kun første og sidste linie blive anvendt. CH = Cursor start linie. CL = Cursor stop linie.
10H	2	Sæt cursor til bestemt sted på skærmen. BH = Sidenummer. (Se side 230) DH = Rækkenr. DL = Kolonnenr.

BIOS interrupts.

10H	3	Læs cursorens aktuelle position på skærmen. BH = Sidenr. (Se side 231) CH/CL := Nuværende cursor mode (som funk.1). DH := Rækkenr. DL := Kolonnenr.
10H	4	Læs lyspens aktuelle position på skærmen.
10H	5	Vælg aktive skærmside. AL = 0-3 for mode 2 og 3 se funk. 0.(4 sider) AL = 0-7 - - - - 0 og 1 - - - - - .(8 sider)
10H	6	Aktiv skærmside scrolles op. (Se side 233) AL = 0 så blankes det hele. AL = X så blankes X linier i bunden af siden. BH = Attributen til blankning. CH/CL = Øvre venstre række, kolonne til scroll DH/DL = Nedre højre - - - - - " - - - - - - -
10H	7	Aktiv skærmside scrolles ned. (Se side 233) Som for funktion 6.
10H	8	Læs tegnet/attribut fra aktuel cursorposition. BH = Sidenr. (Se side 234) AL := ASCII tegn. AH := Attributen.
10H	9	Skriver tegn/attribut til - - - - " - - - - -. BH = Sidenr. (Se side 235) BL = Attribute. CX = Antal gange tegnet skal gentages. AL = ASCII tegnet.
10H	10	Skriv tegn til aktuel cursorposition. AL = ASCII tegn. BH = Sidenr. CX = Antal gange tegnet skal gentages.
10H	11	Sæt farve palette (kun i grafik mode). BH = Farve palette nr. (Se side 237) BL = 0 Vælges baggrundsfarven ifølge BH. BL = 1 Vælges palette ifølge BH.
10H	12	Skriv til pixel. (Se side 238) AL = Farve. DX = Rækkenr. CX = Kolonnenr.
10H	13	Læs pixel værdi (som for funktion 12). (Se side 239)
10H	14	Skriv tegn som TTY (teletype). (Se side 240) AL = Tegnet. BL = Forgrundsfarven (kun i grafik mode). BL = Sidenr.
10H	15	Returner nuværende skærmmode. (Se side 241) AH := Antal skærmkolonner (40 el. 80). AL := Nuværende mode. (Se INT 10H funk. 0). BH := Aktiv sidenr.

BIOS interrupts.

RS 232C (Seriel printer port - normalt 2 - COM1 og COM2)

```
14H        0   Initiering af porten.
               AL = Bit 0,1 Ordlængde (01=7 bits,11=8 bits).
                         2  Stopbits. ( 0=1 bit , 1=2 bits).
                       3,4 Paritet.  (00=ingen , 01 = ulige
                                                 10 = lige).
                     5,6,7 Baud Rate  :
                                        000 = 110
                                        001 = 150
                                        010 = 300
                                        011 = 600
                                        100 = 1200
                                        101 = 2400
                                        110 = 4800
                                        111 = 9600

14H        1   Send et tegn.
               AL  = Tegnet.
               AH := bit 7 = 1 så fejl, ved bit 7 = 0 så ok.
                     bit 6...0 se under funktion 3.

14H        2   Modtag et tegn.
               AL := Tegnet.
               AH := 0 så ok, ellers fejl se funktion 3.

14H        3   Hent portens status.
               AH := Bit 7 = Time out.
                         6 = Shift register tom.
                         5 = Holding register tom.
                         4 = Break detected.
                         3 = Framing fejl.
                         2 = Paritets fejl.
                         1 = Overflow fejl.
                         0 = Data Ready.

               AL := Bit 7 = Received Line Signal Detect.
                         6 = Ring indicator.
                         5 = Data Set Ready.
                         4 = Clear to Send.
                         3 = Delta Received Line Signal Det.
                         2 = Trailing Edge Ring Detector.
                         1 = Delta Data Set Ready.
                         0 = Delta Clear to Send.
```

Tastatur. AH

```
16H        0   Læs næste ASCII tegn til AL og Scan kode i AH.
               (Se side 261)
16H        1   Afgør om der er tegn i bufferen. (Se side 262)
               Z bit = 1 hvis bufferen er tom.
               Z bit = 0 AH = Scan kode.   (Anvend funk. 0 da
                         AL = ASCII tegn.   det forbliver i
                                            bufferen.)
```

BIOS interrupts.

```
16H        2   AL := Tastaturets statusbyte. (Se side 263)
               bit 0 = Højre shift taste tryk.
                   1 = Venstre - - - " - - - .
                   2 = Ctrl/shift  - " - - - .
                   3 = Alt shift   - " - - - .
                   4 = Scroll lock.
                   5 = Numerisk lock.
                   6 = Caps lock.
                   7 = Ins aktiv.
```

Printer.(Parallel printer port - normalt 3 - LPT1,LPT2,LPT3)

```
17H        0   Send et tegn til printer.
               AL = Tegn.
               DX = Printernr. (0,1 el.2)
               AH := Statusbits se funktion 1.

17H        1   Initier printer porten.
               DX = Printernr.
               AH := Printerens status.
                     Bit 7 = Printer klar.
                         6 = ACK.
                         5 = Intet papir i printer.
                         4 = Printer selected.
                         3 = I/O fejl.
                         2 = Ubrugt.
                         1 = Ubrugt.
                         0 = Time out. (Valg af ej existeren-
                                        de printer).

17H        2   Hent printerportens status.
               DX = Printernr. (0,1 el. 2).
               AH := Printerens status - se under funktion 1.
```

Timer. AH

```
1AH        0   Læs nuværende klokværdi. (Time of Day).
               AL := 0 hvis midnat er passeret siden sidst.
               CX+DX := Timerens værdi.
               Timeren tæller 65.536 gange i timen eller om-
               kring 18,204 gange pr. sekund.

1AH        1   Sæt klokværdier.
               CX+DS = Time of Day med CX som high byte.
```

Diverse.

```
1BH            Tastaturets BREAK adresse d.v.s. hvortil der
               fortsættes ved BREAK.

1CH            Timer Tick Interrupt d.v.s. man får adressen
               på IRET ordren i interruptrutinen, som kal-
               des ca. 18,2 gange pr. sekund. Se INT 8.
1DH            Pointer til skærm parameter tabel.
1EH            Pointer til diskette parameter tabel.
1FH            Pointer til grafiktabel. Se side 226.
               Hvert tegn er 8 bytes lang.
```

Int 33h Function 00h V2
Initialize the Mouse

Determines whether a mouse is installed, resets the driver, and returns the number of buttons on the mouse

Calling Registers: AX 0000h

Return Registers: AX 0, mouse not installed
-1, mouse installed
BX Number of buttons (2 for Microsoft, 3 for some other brands)

Comments: When a program in which you want to use the mouse starts, that program must verify the mouse's presence. The usual way to do this is to call Function 00h. This function resets the mouse to the center of the screen, makes sure that the mouse is off, and sets the default mouse cursor and default movement ratios.

In DOS V2.x, it is best to verify that the vector for Interrupt 33h points to code before using this function. If the four bytes of the vector are not all 00, it should be safe to use Function 00h to determine whether the driver is present.

The initial conditions established for the mouse driver by this function are

Display page:	Page 0
Cursor range:	Entire screen (x=0 to 639, y=0 to 199)
Exclusion area:	None
Cursor position:	At screen center (x=320, y=100)
Cursor state:	Hidden
Cursor shape:	Arrow for graphics modes Reverse block for text modes
User Interrupts:	Disabled
Light pen emulator:	Enabled
Mickey/Pixel ratio:	Horizontal = 8 to 8 Vertical = 16 to 8
Speed threshold:	64 mickeys per second (Mickey is the unit of mouse motion. One mickey is approximately 1/200 inch.)

Int 33h Function 01h V2
Show Mouse Cursor

Causes the mouse cursor to appear on the display

Calling Registers: AX 0001h

Return Registers: None

Comments: Turns on the mouse cursor, allowing display on the screen. This is not an absolute function; rather, it increments an internal mouse-cursor flag. Initially, this flag is set to a −1. Whenever the flag is zero, the mouse cursor will be displayed. Function 02h decrements the cursor flag, causing the cursor to disappear if the original value was zero. Thus, multiple calls to Function 02h require multiple calls to this function. The software prevents the flag's value from becoming greater than zero, however.

Int 33h Function 02h V2
Hide Mouse Cursor

Turns off display of the mouse cursor

Calling Registers: AX 0002h

Return Registers: None

Comments: This function turns off the display function but does not disable the driver. As noted in the comment for Function 01h, Function 02h decrements a cursor flag. If the value is not zero, the cursor is turned off. Because the flag can never be greater than zero, a single call to this function is guaranteed to hide the cursor.

Int 33h Function 03h V2
Get Mouse Position

Returns current mouse position and button status

Calling Registers: AX 0003h

Return Registers: BX Button status
CX X-coordinate (horizontal)
DX Y-coordinate (vertical)

Comments: This function tells you where the mouse is located. No matter what mode the screen is in, Function 03h always returns an x-coordinate (column) between 0 and 639 and a y-coordinate (row) between 0 and 199. Table M.1 shows the mouse cursor's allowable positions for each display mode, in terms of screen (pixel) coordinates.

Table M.1. *Mouse Cursor's Position*

Screen Mode	*Mouse Coordinates*
00h, 01h	x = 16 × column y = 8 × row
02h, 03h	x = 8 × column y = 8 × row
04h, 05h	x = 2 × screen X y = Screen Y
06h	x = Screen X y = Screen Y
07h	x = 8 × screen column y = 8 × screen row
0Eh–10h	x = Screen X y = Screen Y

The status of the mouse buttons is returned in BX; only the low-order bits are significant. Table M.2 illustrates the meaning of the bits. Because each button acts independently, the value can be anything from 0 to 3 for a two-button mouse or 0 to 7 for the three-button version.

Table M.2. *Mouse Button Status Bits*

Bit 76543210	*Meaning*
.....0	Left button up
.....1	Left button down
.....0.	Right button up
.....1.	Right button down
.....0..	Center button (if present) up
.....1..	Center button (if present) down
xxxxx...	Undefined

Int 33h — Function 04h — V2

Set Mouse Position

Sets the mouse's position on the screen

Calling Registers:
- AX — 0004h
- CX — New x-coordinate (horizontal)
- DX — New y-coordinate (vertical)

Return Registers: None

Comments: You can use this function to place the mouse cursor anywhere on the screen. (The mouse driver will resume operating from that location.) This function is useful, for example, when you want to start the mouse at the first item on a menu that is brought up on the screen.

If either coordinate has a value inappropriate for the current screen mode as defined for Function 03h, it will be adjusted to the nearest appropriate value. If the specified position lies outside the display range established by calls to Functions 07h and 08h, the cursor will be placed as close to the specified position as possible while remaining within the range limits. If the position lies within an exclusion area defined by Function 10h, it will be hidden.

Int 33h — Function 05h — V2

Get Button-Press Information

Returns information about button presses

Calling Registers:
- AX — 0005h
- BX — Button
  - 0, left
  - 1, right
  - 2, center (if present)

Return Registers:
- AX — Button status (see table M.2)
- BX — Count of button presses (reset to zero at each call)
- CX — Cursor's horizontal position at last button press
- DX — Cursor's vertical position at last button press

5.1.d

Bjørk Busch

DOS-interrupt funktioner for skrivning på skærm/std.output

Funktion 02: Tegn output med Ctrl/break check

```
     Mov        Ah,02h
     Mov        Dl,Tegn          ; Tegn der skal udskrives
     Int        21h
```

Funktion 05: Tegn print - printerudgang

```
     Mov        Ah,06h
     Mov        Dl,Tegn          ; Tegn der skal udskrives
     Int        21h
```

Funktion 06: Tegn - direct Console I/O.

```
     Mov        Ah,06h
     Mov        Dl,Tegn          ; Tegn som udskr. (ikke
                FFh)
     Int        21h
```

Funktion 09: Streng udskrift

```
     Mov        Ah,06h
     Lea        Dx,Tekststr      ; streng afsluttes med $
     Int        21h
```

eks. på tekststr

```
     Tekststr  DB   'Tekstlinie for udskrift',13,10,'$'
```

13 er ascii-tegn carrige-return
10 er ascii-tegn linefeed

Funktion 40: Write blok (ikke linier - også til alm. filer)

```
     Mov        Ah,40h
     Mov        Bx,1             ; handle 1 = std. out.
                                 ;        2 = std. error
                                 ;        4 = print (prn:)
     Lea        Dx,Outputdata    ; adr. på buffer der
                                 ; skal skrives
     Mov        Cx,Outlen        ; recordlængde
     Int        21h
     JC         error            ; C-flag sat hvis fejl
     Cmp        Ax,Cx            ; Ax - faktisk skrevet
     JNE        error2           ; ikke alt skrevet
```

DOS-interrupt funktioner for læsning fra tastatur / std.input

Funktion 01: Tegn input med ECHO og med Ctrl/Break check

```
     Mov        Ah,01h
     Int        21h
     Mov        Tegn,Al          ; Hvis 00h så extended;
```

Funktion 07: Tegn input **UDEN** ECHO og **UDEN** Ctrl/Break check

Som funktion 01.

Funktion 08: Tegn input **UDEN** ECHO og **MED** Ctrl/Break check

Som funktion 01.

Funktion 06: Tegn - direct Console I/O.

```
     Mov        Ah,06h
     Mov        Dl,0FFh          ; for læsning
     Int        21h
     JZ         ingeninp         ; Z-flag sat hvis tom
                                 buffer
     Mov        Tegn,Al          ; Hvis 00h så extended
```

Funktion 0A: Buffered Input (linie - til carrige return):

```
     Mov        Ah,0Ah
     Lea        Dx,Inpbuf
     Int        21h
```

Før kald: 1.byte i Inpbuf skal indeholde længde af buffer excl. 2 byte (1 byte med max. længde og 2.byte med indlæst længde)

Efter kald: 2.byte indeholder faktiske antal indlæste tegn excl. carrige-return (ascii-tegn 13). 3-?? byte indeholder de indlæste tegn samt en afsluttende carrige-return.

Inpbuf kan defineres på følgende måde:

```
     Inpbuf     DB   20              ; max. længde incl. CR
     Inplen     DB   ?               ; til faktisk længde
     Inpdata    DB   20 DUP (?)      ; egentlig input
```

eller med brug af ekstra adresse-symbol og label

```
     Inpbuf     LABEL BYTE           ; startadr.
     Inpmax     DB   20              ; max. længde incl. CR
     Inplen     DB   ?               ; til faktisk længde
     Inpdata    DB   20 DUP (?)      ; egentlig input
```

når LABEL anvendes som her afsættes ikke plads, men der knyttes en adresse og type til symbol. Adresse bliver her den samme som for "Inpmax".

Funktion 3F: Read blok (ikke linier - også til alm. filer)

```
     Mov        Ah,3Fh
     Mov        Bx,0             ; handle 0 = std. inp.
     Lea        Dx,Inputdata     ; adr. på buffer til input
     Mov        Cx,Maxinp        ; recordlængde
     Int        21h
     JC         error            ; C-flag sat hvis fejl
     Cmp        Ax,0             ; Ax - faktisk indlæst
     Jz         EndOfFile        ;    der var EOF
```

5.1.e

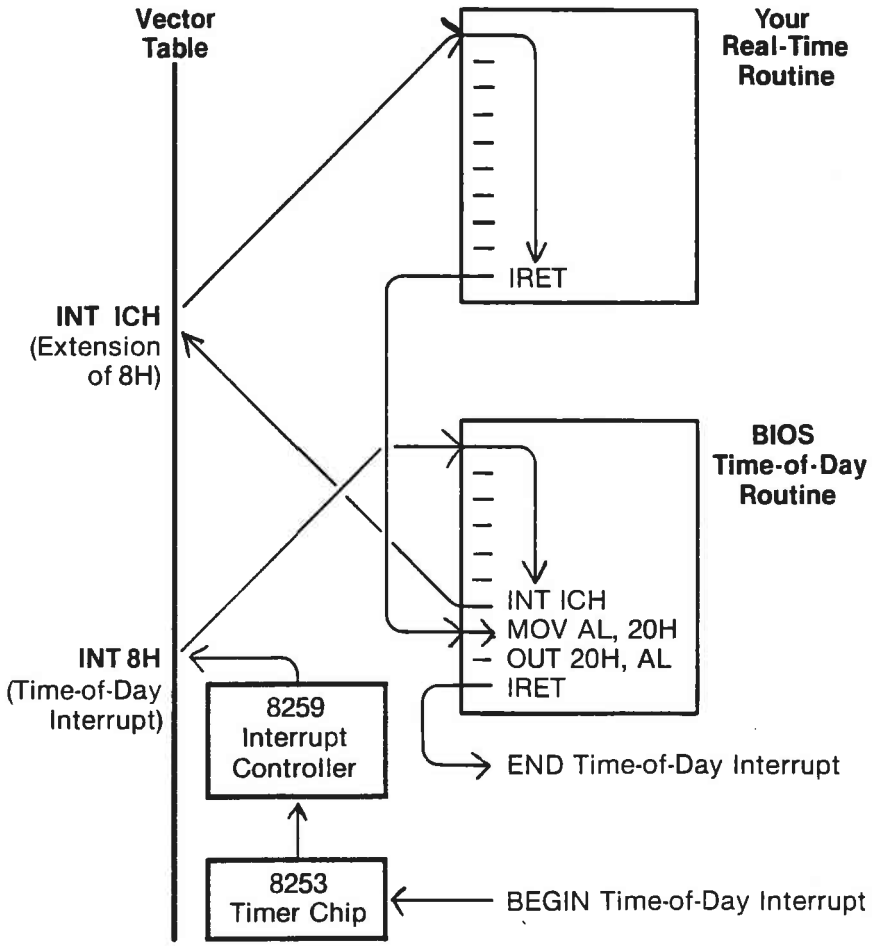
Selvdefinerede interruptrutiner

Bjørk Busch

Der er muligt for programmøren, selv at skrive nye interruptrutiner, såvel hardware- som software.

Man kan desuden overtage de oprindelige, idet man evt. inddrage det "gamle" ved, at flytte det til et andet interrutnr og kalde det fra sin egen interruprutine.

Timerinterruptet er på forhånd etableret, så det er let for programmøren, at udvide rutinen, idet man "blot" instalerer sig på interruptnr 1Ch.

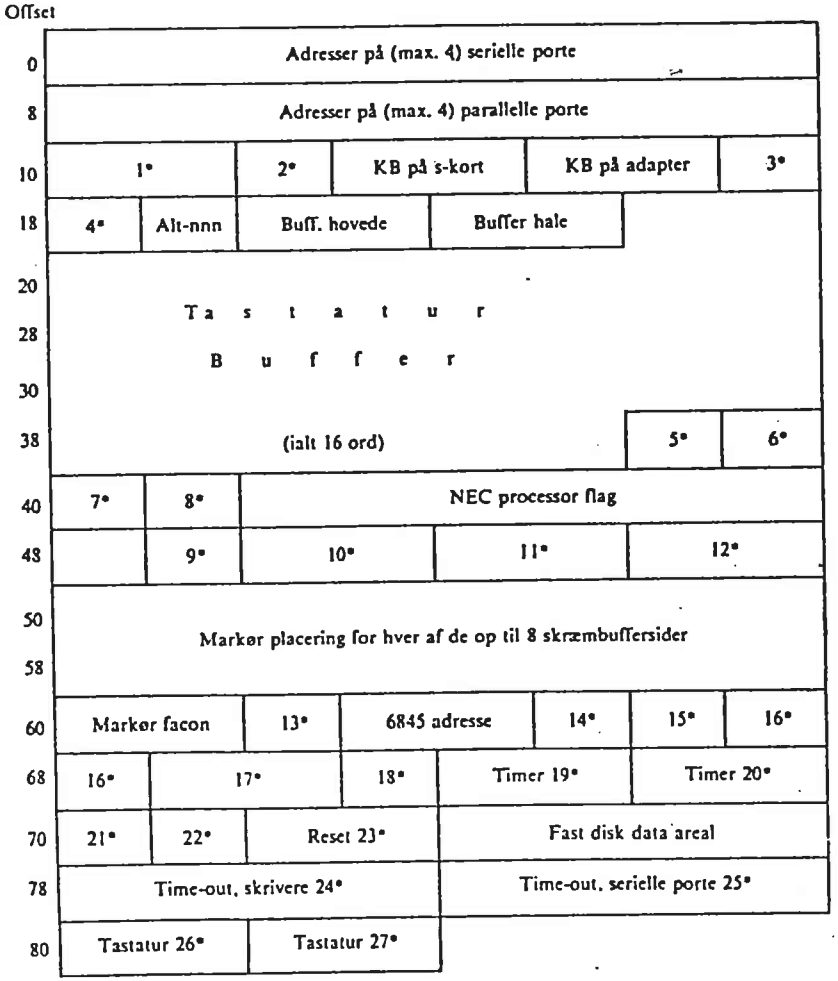


Extending the timer interrupt.

ROM BIOS data arealer

ROM BIOS data befinder sig i segment 40H fra offset 0 til 83H, altså på adresse 0000:0400 - 0000:0483.

Neden for er vist data områdets lay-out. Dernæst følger detailler for felter markeret i lay-outet med et tal og en stjerne.



1* Bit i "equipment" flag
Offset: 10H. Længde: 2 byte.

....1	Mindst et diskette drev
.... nn..	(nn+1)x16 KB RAM. nn=3 betyder mindst 64 KE
..nn	nn=01: 40x25 farve monitor nn=10: 80x25 farve monitor. nn=11: 80x25 monokrom monitor
nn..	nn+1 diskettedrev
.... nnn.	nnn serielle porte
....1	spil adapter installeret
.... nn..	nn parallelle porte (skrivere)

2* Offset adresse på data areal til fejlsøgning under fabrikation
Offset: 12H. Længde: 1 byte.

3* 1. tastatur flag
Offset: 17H. Længde: 1 byte.

....1	Højre skifte taste holdes nede
.... ..1.	Venstre skifte taste holdes nede
.... .1..	Ctrl taste holdes nede
.... 1...	Alt taste holdes nede
...1	Scroll Lock er aktiv
..1.	Num Lock er aktiv
.1..	Caps Lock er aktiv
1...	Ins er aktiv

4* 2. tastatur flag
Offset: 18H. Længde: 1 byte.

.... 1...	Ctrl-NumLock (pause) aktiv
...1	Scroll Lock taste holdes nede
..1.	Num Lock taste holdes nede
.1..	Caps Lock taste holdes nede
1...	Ins taste holdes nede

5* Diskette søge status
Offset: 3EH. Længde: 1 byte.

....0	Drev A behøver rekalibrering før næste søgning
.... ..0.	Drev B behøver rekalibrering før næste søgning
.... .0..	Drev C behøver rekalibrering før næste søgning
.... 0...	Drev D behøver rekalibrering før næste søgning

6* Diskette motor status
Offset: 3FH. Længde: 1 byte.

....	...1	Drev A's motor kører
....	..1.	Drev B's motor kører
....	.1..	Drev C's motor kører
....	1...	Drev D's motor kører
1...	....	Igangværende operation er skrivning

7* Time-out værdi til slukning af diskette motor
Offset: 40H. Længde: 1 byte.

8* Diskette status byte
Offset: 41H. Længde: 1 byte.

....	...1	Ugyldig kommando sendt til diskette adapter
....	..1.	Formateringsfejl
....	..11	Skrivning forsøgt på skrivebeskyttet diskette
....	.1..	Sektor ikke fundet
....	11..	DMA overløb
...1	....	64 KB grænse overskredet ved DMA
..1.	....	Bitfejl ved læsning
..1.	....	Fejl i NEC processor
.1..	....	Forgæves søgning
1...	....	Time-out

9* Aktuel video status
Offset: 49H. Længde: 1 byte.
0 = 40x25 sort/hvid tekst
1 = 40x25 farve tekst
2 = 80x25 sort/hvid tekst
3 = 80x25 farve tekst
4 = 320x200 farve grafik
5 = 320x200 sort/hvid grafik
6 = 640x200 sort/hvid grafik
7 = monokrom tekst

10* Antal søjler på skærm
Offset: 4AH. Længde: 2 byte. Værdi: 40 eller 80.

11* Skærmbuffer længde i bytes
Offset: 4CH. Længde: 2 byte.

12* Skærmbuffers relative start adresse
Offset: 4EH. Længde: 2 byte.

13* Sidenummer for aktuel side
Offset: 62H. Længde: 1 byte. Værdi: 0-7.

14* Register status
Offset: 65H. Længde: 1 byte.

15* Aktuel palet
Offset: 66H. Længde: 1 byte.

16* Kassetteports synkroniserings ord
Offset: 67H. Længde: 2 byte.

17* Bitfejl kontrol (CRC) for kassetteport
Offset: 69H. Længde: 2 byte.

18* Sidst læste/skrevne data byte fra kassetteport
Offset: 6BH. Længde: 1 byte.

19* Tælles 1 op hver 5/91 sekund
Offset: 6CH. Længde: 2 byte.

20* Tælles 1 op hver time. Nulstilles efter 1 døgn.
Offset: 6EH. Længde: 2 byte.

21* Overløb af tæller (ur) siden sidste aflæsning
Offset: 70H. Længde: 1 byte.

22* Bit 7 = 1, hvis Ctrl-Break har været aktiveret; ellers 0.
Offset: 71H. Længde: 1 byte.

23* 1234H, hvis Ctrl-Alt-Del genstart er igang; ellers FF00H.
Offset: 72H. Længde: 2 byte.

24* Hver byte angiver en skrivers time-out
Offset: 78H. Længde: 4x1 byte. Værdi 14H = 20 sek.

25* Hver byte angiver en seriel ports time-out
Offset: 7CH. Længde: 4x1 byte. Værdi: 1 sek.

26* Offset adresse for tastatur buffers start
Offset: 80H. Længde: 2 byte. Værdi: 001E.

27* Offset adresse for tastatur buffers slutning
Offset: 80H. Længde: 2 byte. Værdi: 003E.

Program Segment Prefix.

PSP'et er et område på 256 bytes (100 hex), som MS-DOS initierer ved opstart af programmet. Området er fælles for alle programmer startet under MS-DOS, og findes altså også i programmer i EXE-format.

De sidste 128 bytes af området indeholder eventuelle parametre, som i forbindelse med opstarten blev indtastet i kommandolinien efter programnavnet. Disse oplysninger kan nu blive læst af programmet, og danne grundlag for variationer i programmets eksakte virkemåde.

Den følgende skitse viser opbygningen af PSP'et:

Offset (hex)	Længde bytes	Beskrivelse
00	2	INT 20 maskininstruktion (stop program)
02	2	Segmentadressen for den sidste del af det lager, som er allokeret (tildelt) til programmet. Ved hjælp af denne information kan programmet beregne, hvor meget lager, der aktuelt er tilknyttet til det.
04	1	Reserveret.
05	5	Maskininstruktion, som kan anvendes til kald af MS-DOS funktionerne.
0A	4	INT 22 adresse.
0E	4	INT 23 adresse (Ctrl-C afbrydelse).
12	4	INT 24 adresse (afbrydelse på grund af kritisk fejl).
16	22	Reserveret.
2C	2	Environment segment adresse. I den såkaldte environment blok kan man indsætte forskellige former for information, som kan benyttes af operativsystemet og programmet.
2E	34	Reserveret.
50	3	INT 21 og RETF maskininstruktioner.
53	9	Reserveret.
5C	16	FCB#1
6C	16	FCB#2. De to FCB'er er et levn fra en tid, hvor man brugte en anden form for filbehandling i MS-DOS. Siden er denne erstattet af en langt mere moderne teknik.
80	1	Længden på kommandoliniens parametre.
81	127	Kommandoliniens parametre. Parametrene er indeholdt i en tekststreng, som begynder med tegnet umiddelbart efter kommandoens navn (et blanktegn). Herefter følger kommandoliniens parametre i uformatteret form, som de blev indtastet, med blanktegn, kommaer og andre skilletegn.

5.2.c

Environment-blok

Bjørk Busch

Environment-blokken anvendes som bindeled mellem et program og styresystemet. Ved opstart af et program vil der blive skabt en kopi af den aktuelle environment-blok til programmet. Programmet kan direkte få adgang til denne kopi gennem PSP'et, der ligeledes skabes i forbindelse med opstart af et program. Når programmet afslutter frigives hukommelsen til såvel den tilknyttede environment-blok som PSP og programmet selv. Dette gælder dog ikke for residente programmer.

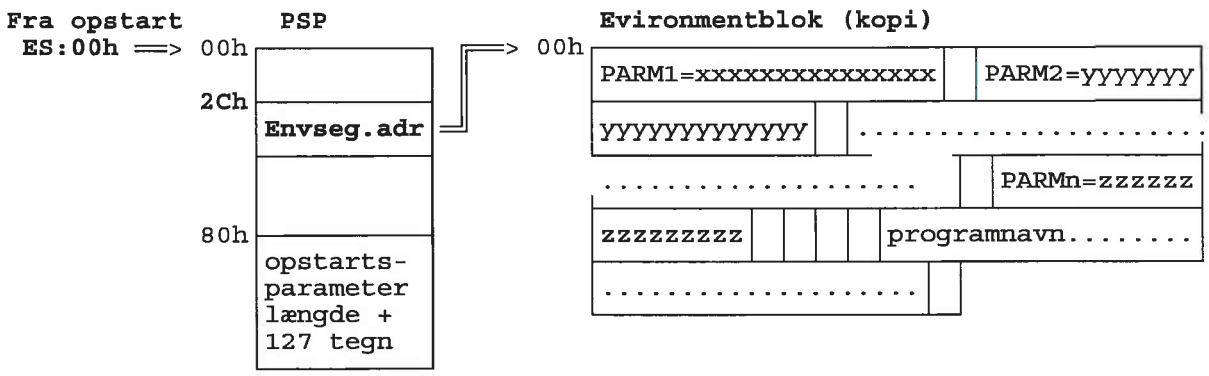
Systemvarable.

Environment-blokken består af en tabel med NUL-terminerede tekststrenge, der indeholder systemvariable.
Strengene indeholder ingen længdebyte, men de afsluttes hver især med en byte med binært-nul. Der er således ikke faste størelser på elementerne og tabellen må søges igennem sekventielt.
Tabellen med systemvariable afsluttes med en "ekstra" byte med binært nul, idet der således er 2 bytes med binært nul efter sidste systemvariabel.
Hver af element i tabellen indeholder først navnet på systemvariablen (med store bogstaver), herefter følger et lighedstegn og efter dette indholdet af systemvariablen.

Programnavn.

Efter tabellen med systemvariable følger en byte med 01h, herefter en byte med 00h og herefter en NUL-termineret streng med navnet på det opstartede program med fuld stinavn.
Den sidste beskrivelse er ikke i de officielle beskrivelser fra Microsoft, men kan være nyttig, idet man her kan skaffe sig navnet på det katalog, hvor programmet befinder sig.

Sammenkædning til program.



5.2.d

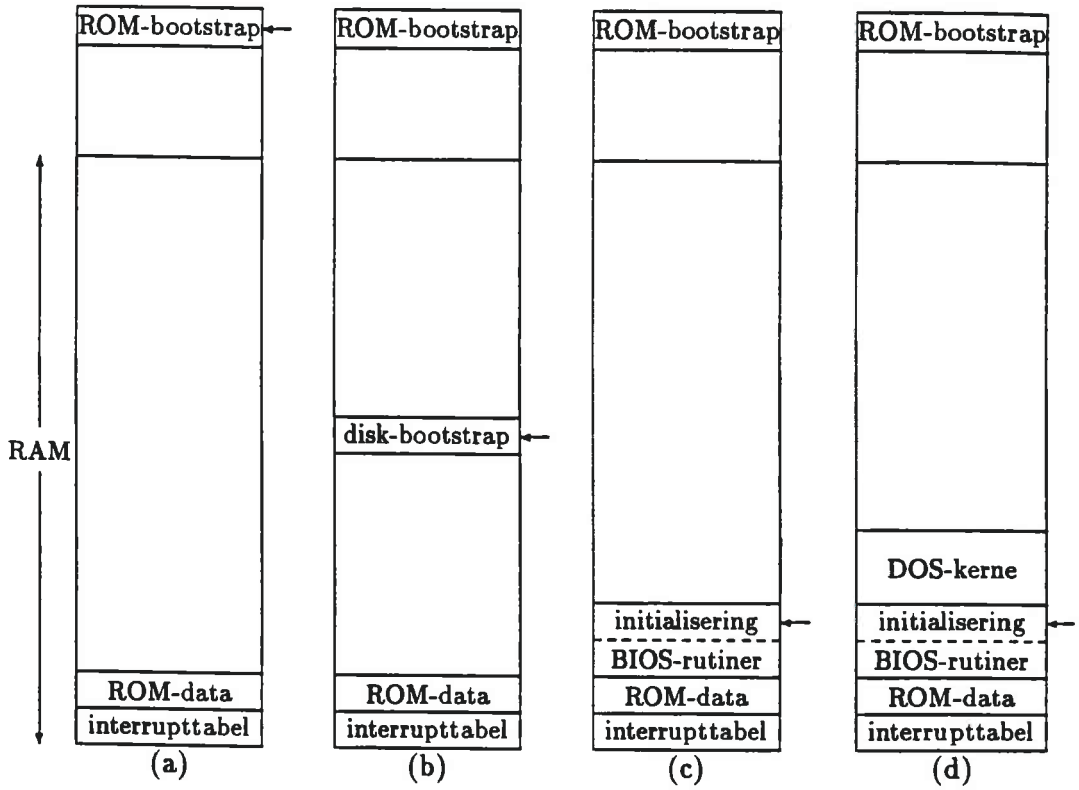
Adresserummet under DOS

Således anvendes memory under DOS

	XMS (extended memory) XMS kan ikke anvendes direkte i DOS programmer, men bruges til RAMDISK, CACHE m.m. Ved brug af EMM386 kan XMS anvendes til simulering af EMS	?? MB
110000h ---->	HMA (evt. flyttes DOS hertil)	64 KB
100000h ---->	ROM BIOS (incl. bootstrap)	16 KB
F0000h ---->	UPPER MEMORY (residente prog./dosdrivere)	64 KB
	ROM BIOS for video, harrddisk m.m.	
C0000h ---->	Video-memory COLOR TEKST/grafik	32 KB
B8000h ---->	Video-memory MONO TEKST/grafik	32 KB
B0000h ---->	Video-memory EGA/VGA grafik/ramfonte	64 KB
A0000h ---->	Transient program areal (brugerprogrammer)	640 KB
	Residente del af DOS	
	ROM BIOS og BASIC dataareal	
00500h ---->	ROM BIOS dataareal	
00400h ---->	Interrupt adresse tabel	
00000h ---->		

5.3.a

Opstartsfaser for DOS

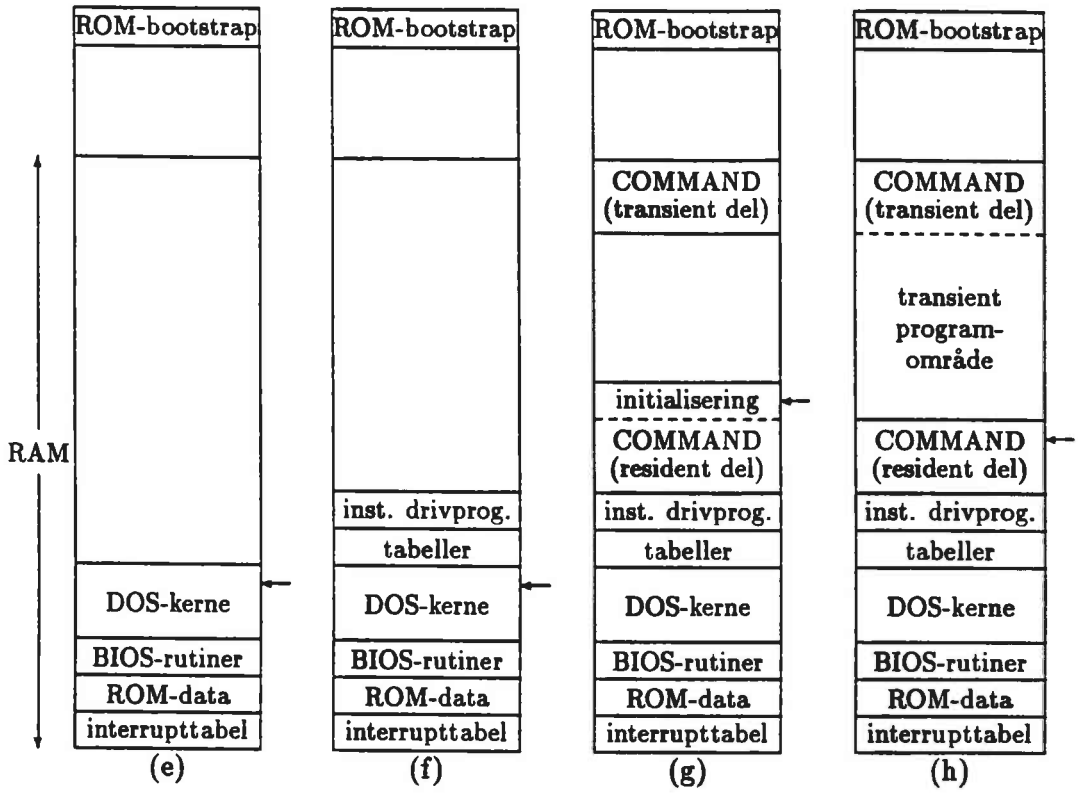


Figur 17.1: DOS-"boot"-proces

Maskinen starter, når der sættes strøm til systemet eller efter systemreset (aktiveret af den velkendte <Ctrl><Alt><Del>-tastekombination), udførelse af et program på adresse 0FFFF0H (i ROM). Dette program foretager en grundlæggende systemtest, initialiserer interruptvektortabellen og visse ROM BIOS-dataområder, og afslutter med udførelse af ROM-bootstrap programmet, som forsøger at indlæse en "diskboot-record" (se Figur 17.1 (a)). På figurerne er udførselspunktet antydet med en pil.

Hvis en sådan "boot-record" findes (på disketten i A:, eller på harddisken C:), indlæses diskbootstrapprogrammet til lageret, og udførelsen af det begyndes (se Figur 17.1 (b)), ellers startes den ROM residente BASIC fortolker.

Disk boot-record'en indeholder et lille program, der har til formål at kontrollere om DOS-systemfilerne findes i rodkataloget på boot-disken. Hvis disse filer er til stede (i korrekt rækkefølge, med IBMBIO.COM som den første fil på disken), indlæses IBMBIO.COM-filen til lageret. IBMBIO.COM-filen består egentlig af to komponenter, dels *BIOS-rutinerne* med forskellige hardwarespecifikke drivprogrammer, samt en *initialiseringsdel.* Initialiseringsdelen startes (Figur 17.1 (c)),



Figur 17.2: DOS-"boot"-proces (fortsat)

og blandt andet foretages her indlæsning af selve *DOS-kernen*, (Figur 17.1 (d)), fra filen IBMDOS.COM, og ved en lille tryllekunst relokeres denne til sin endelige position umiddelbart efter BIOS-rutinerne (Figur 17.2 (e)).

Herefter gives kontrol til DOS-kernen (Figur 17.2 (e)), som opbygger sine interne arbejdstabeller for diskfiladministration og lageradministration. Det er på dette tidspunkt, at en eventuel CONFIG.SYS-fil indlæses og behandles, og installering af eventuelle *installerbare drivprogrammer* foretages (Figur 17.2 (f)).

Nu er DOS som sådan på plads, og det sidste trin i bootprocessen er indlæsning af en *user-shell*, som i de fleste tilfælde blot er den sædvanlige kommandofortolker (fra COMMAND.COM-filen). DOS-kernen indlæser COMMAND.COM og påbegynder udførelsen af initialiseringsdelen, som blandt andet foretager behandling af en eventuel AUTOEXEC.BAT-fil.

Tilsidst får den egentlige kommandofortolker kontrol, tilkendegivet ved, at den viser sin "prompt", hvor den afventer indtastning af brugerkommandoer (Figur 17.2 (g) og (h)). Lagerområdet benyttet til initialiseringsdelen bliver frigivet, og det udgør sammen med resten af RAM'en og området hvortil den transiente del af kommandofortolkeren er indlæst, det såkaldte *transiente programområde*, hvori DOS kan bringe andre programmer til udførelse.

6.

Assemblermaskinen

Indhold:

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

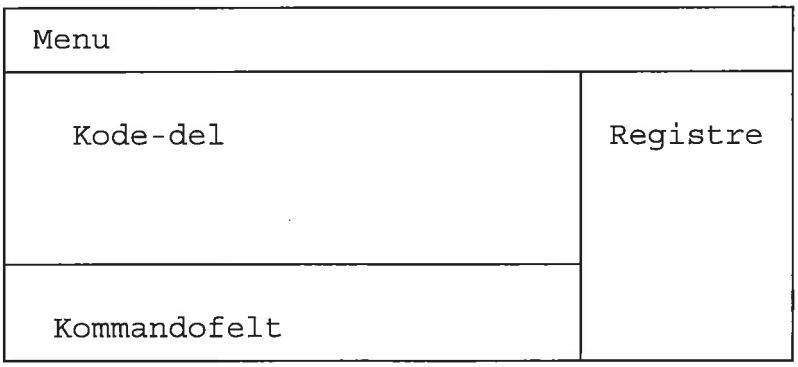
Prog er navnet på det færdige programmet (EXE-fil).

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.**

7.

Pascalmaskinen

Indhold:

7.1.a

Implementering af konstanter og variable datafelter

Bjørk Busch

Initiering på oversættelsestidspunktet:

PASCAL

```
CONST
     KONSTANT = 15;
     STARTTAL: INTEGER = 10;
```

ASSEMBLER

```
KONSTANT  EQU  15
STARTTAL  DW   10
```

Værditilskrivning på kørselstidspunktet:

PASCAL

```
VAR  {main}
     ITAL       :INTEGER;
     ITAL2      :INTEGER;
     BTAL       :BYTE;
     BTAL2      :BYTE;
     TEGN       :CHAR
     TEGN2      :CHAR
     TEKST      :STRING[5]
     ITABEL     :ARRAY[1..5]
                 OF INTEGER;
```

ASSEMBLER

```
{datasegment}
ITAL      DW   ?
ITAL2     DW   ?
BTAL      DB   ?
BTAL2     DB   ?
TEGN      DB   ?
TEGN2     DB   ?
TEKST     DB   6     DUP (?)
ITABEL    DW   5     DUP (?)
```

Tilskrivning med konstant:

```
ITAL := 5;                MOV  ITAL,5

TEKST := 'ABC';           MOV  TEKST+0,3        ;længde
                          MOV  TEKST+1,'A'
                          MOV  TEKST+2,'B'
                          MOV  TEKST+3,'C'

ITABEL[1] := 10;          MOV  ITABEL+0,10
ITABEL[2] := 20;          MOV  ITABEL+2,20
ITABEL[3] := 30;          MOV  ITABEL+4,30
ITABEL[4] := 40;          MOV  ITABEL+6,40
ITABEL[5] := 50;          MOV  ITABEL+8,50
```

Tilskrivning med variabel:

```
ITAL := ITAL2;                  MOV  AX,ITAL2
                                MOV  ITAL,AX

BTAL := BTAL2;                  MOV  Al,BTAL2
                                MOV  BTAL,Al

TEGN := TEGN2;                  MOV  Al,TEGN2
                                MOV  TEGN,Al

ITABEL[3] := ITAL;              MOV  AX,ITAL
                                MOV  ITABEL+6,AX
```

Tilskrivning med simpel typekonvertering:

```
TALB := ORD(TEGN);              MOV  Al,TEGN
                                MOV  TALB,Al


TEGN := CHR(BTAL);              MOV  Al,BTAL
                                MOV  TEGN,Al

ITAL := BTAL;                   MOV  AL,BTAL
                                CBW              ;AX:=AL
                                MOV  ITAL,AX

BTAL := ITAL;                   MOV  AX,ITAL
                                MOV  BTAL,AL     ;-kontrol
```

Tilskrivning med simpel 16-bits beregning:

```
ITAL := ITAL + ITAL2;           MOV  AX,ITAL
                                ADD  AX,ITAL2
                                MOV  ITAL,AX

ITAL := ITAL - ITAL2;           MOV  AX,ITAL
                                SUB  AX,ITAL2
                                MOV  ITAL,AX

ITAL := ITAL * ITAL2;           MOV  AX,ITAL
                                MOV  BX,ITAL2
                                IMUL BX
                                MOV  ITAL,AX

ITAL := ITAL DIV ITAL2;         MOV  AX,ITAL
                                CWD
                                MOV  BX,ITAL2
                                IDIV BX
                                MOV  ITAL,AX

ITAL := ITAL MOD ITAL2;         MOV  AX,ITAL
                                CWD
                                MOV  BX,ITAL2
                                IDIV BX
                                MOV  ITAL,DX
```

7.1.b

Implementering af selektion og iteration

Bjørk Busch

Simpel IF sætning:

PASCAL.

A og B er 2 integer variable:

```
IF (A < B) THEN BEGIN
.....
END
ELSE BEGIN
...
END;
```

Omskrevet til assembler:

```
IFTST:    MOV       AX,A
          CMP       AX,B
          JNL       ELSEBEG   eller     JL    IFBEG
                               og       JMP   ELSEBEG
IFBEG:    .....
          .....
          JMP       IFEND
ELSEBEG:  .....
          .....
IFEND:
```

PASCAL.

A, B og C er 3 integer variable:

```
IF ((A + B) = C) THEN BEGIN
.....
END
ELSE BEGIN
...
END;
```

Omskrevet til assembler:

```
IFCALC:   MOV       AX,A
          ADD       AX,B
IFTST:    CMP       AX,C
          JNE       ELSEBEG   eller     JE    IFBEG
                               og       JMP   ELSEBEG
IFBEG:    .....
          .....
          JMP       IFEND
ELSEBEG:  .....
          .....
IFEND:
```

Sammensat IF sætning med AND:

PASCAL.

A,B og C er 3 integer variable:

```
IF (A < B) AND (B > C) THEN BEGIN
.....
END
ELSE BEGIN
...
END;
```

Omskrevet til assembler:

```
IFTST:     MOV      AX,A
           CMP      AX,B
           JNL      ELSEBEG   eller    JL    TEST2
                              og      JMP   ELSEBEG
IFTST2:    MOV      AX,B
           CMP      AX,C
           JNG      ELSEBEG   eller    JG    IFBEG
                              og      JMP   ELSEBEG
IFBEG:     .....
           .....
           JMP      IFEND
ELSEBEG:   .....
           .....
IFEND:
```

Sammensat IF sætning med OR:

PASCAL.

A,B og C er 3 integer variable:

```
IF (A < B) OR (B > C) THEN BEGIN
.....
END
ELSE BEGIN
...
END;
```

Omskrevet til assembler:

```
IFTST:     MOV      AX,A
           CMP      AX,B
           JL       IFBEG
           MOV      AX,B
           CMP      AX,C
           JNG      ELSEBEG   eller    JG    IFBEG
                              og      JMP   ELSEBEG
IFBEG:     .....
           .....
           JMP      IFEND
ELSEBEG:   .....
           .....
IFEND:
```

Simpel WHILE sætning:

PASCAL.

```
A og B er 2 integer variable:

WHILE (A < B) DO BEGIN
.....
END;
```

Omskrevet til assembler:

```
WHILE:     MOV       AX,A
           CMP       AX,B
           JNL       ENDWHILE
WHILEBEG:  ......
           ......
           JMP       WHILE
ENDWHILE:
```

Simpel REPEAT sætning:

PASCAL.

```
A og B er 2 integer variable:

REPEAT
......
UNTIL (A < B);
```

Omskrevet til assembler:

```
REPEAT:    .......
           .......
UNTIL:     MOV       AX,A
           CMP       AX,B
           JNL       REPEAT
```

Simpel FOR sætning hvor tæller ikke anvendes:

PASCAL.

```
FOR I := 1 TO 10 DO BEGIN
......
END;
```

Omskrevet til assembler:

```
FORINIT:   MOV       CX,10
FORBEG:    ......
           ......
FORNEXT:   LOOP      FORBEG     eller     DEC CX + JNZ
                                 og        JNZ  FORBEG
```

Gennerel model for FOR-konstruktion der tæller op og hvor tæller kan anvendes:

PASCAL.

```
I er en integer.

FOR I := START TO STOP DO BEGIN
......
END;
```

Omskrevet til assembler:

```
FORINIT:  MOV     AX,START
          MOV     I,AX
FORTEST:  MOV     AX,I
          CMP     AX,STOP
          JG      FOREXIT
FORBEG:   PUSH    I          ; SAVE af I - kan undværes
          .....
          .....
FOREND:   POP     I          ;  --- "" ---
          INC     I
          JMP     FORTEST
FOREXIT:
```

Gennerel model for FOR-konstruktion der tæller ned og hvor tæller kan anvendes:

PASCAL.

```
I, Start og Stop er integer.

FOR I := Start DOWNTO  Stop DO BEGIN
......
END;
```

Omskrevet til assembler:

```
FORINIT:  MOV     AX,START
          MOV     I,AX
FORTEST:  MOV     AX,I
          CMP     AX,STOP
          JL      FOREXIT
FORBEG:   PUSH    I          ; SAVE af I - kan undværes
          .....
          .....
FOREND:   POP     I          ;  --- "" ---
          DEC     I
          JMP     FORTEST
FOREXIT:
```

Sammensat IF sætning med boolske variable og OR:

PASCAL.

A,B og C er 3 byte's med der kun kan indeholde 0 og 1 (svarende til pascal boolean):

```
IF (A = B) OR (B > C) THEN BEGIN
.....
END
ELSE BEGIN
...
END;
```

Omskrevet til assembler:

BOOLSK - 2 test løsning uden CMP, SUB, ADD m.m.:

```
IFTST:    MOV     AX,A
          XOR     AX,B
          JZ      IFBEG          B C  | B > C
          MOV     AX,C           -----|-------
          NOT     AX             0 0  |   0
          AND     AX,B           0 1  |   0
          JZ      ELSEBEG        1 0  |   1
IFBEG:    .....                  1 1  |   0
          .....                  -----|-------
          JMP     IFEND               |B AND (NOT C)
ELSEBEG:  .....
          .....
IFEND:
```

REN BOOLSK løsning:

```
IFTST:    MOV     AX,A
          XOR     AX,B
          NOT     AX             ;AX := (A = B)

          MOV     BX,AX          ;Gem

          MOV     AX,C
          NOT     AX
          AND     AX,B           ;AX := (B > C)

          OR      AX,BX          ;Saml. med OR

          JZ      ELSEBEG
IFBEG:    ......
          ......
          JMP     IFEND
ELSEBEG:  ......
          ......
IFEND:
```

7.1.c

Implementering af tabeller med index-adressering

Bjørk Busch

På assemblerniveau arbejdes ikke med elementnumre men med celleadresser, og index skal derfor omregnes til celleantal.

1.dimensional tabel

PASCAL

```
VAR  {main}

     ITABEL     :ARRAY[1..5]
                 OF INTEGER;
     INDEX      :INTEGER;
     ITAL       :INTEGER;

     ITABEL[INDEX] := ITAL;

     ITAL := ITABEL[INDEX];

     ITAL := ITABEL[INDEX+2];
```

ASSEMBLER

```
{datasegment}
ELMLEN    EQU  2
STARTIDX  EQU  1
ITABEL    DW   5    DUP (?)

INDEX     DW   ?
ITAL      DW   ?

MOV  AX,ITAL
PUSH AX
MOV  AX,INDEX
SUB  AX,STARTIDX
MOV  SI,ELMLEN
MUL  SI
MOV  SI,AX
POP  AX
*    MOV  ITABEL[SI],AX

*    kan erstattes af:
          LEA  BX,ITABEL
          MOV  [BX+SI],AX

MOV  AX,INDEX
SUB  AX,STARTIDX
MOV  SI,ELMLEN
MUL  SI
MOV  SI,AX
*    MOV  AX,ITABEL[SI]
     MOV  ITAL,AX

*    kan erstattes af:
          LEA  BX,ITABEL
          MOV  AX,[BX+SI]

MOV  AX,INDEX
SUB  AX,STARTIDX
MOV  SI,ELMLEN
MUL  SI
MOV  SI,AX
*    MOV  AX,ITABEL+2[SI]
     MOV  ITAL,AX

*    kan erstattes af:
          LEA  BX,ITABEL
          MOV  AX,[BX+SI]
```

2.dimensional tabel

PASCAL

```
TYPE
     DIM1        :ARRAY[1..3]
                  OF INTEGER;

VAR  {main}
     ITABEL2     :ARRAY[1..5]
                  OF DIM1;
     IDX1        :INTEGER;
     IDX2        :INTEGER;
     ITAL        :INTEGER;

     ITABEL2[IDX2,IDX1] := ITAL;

     ITAL := ITABEL2[IDX2,IDX1];
```

ASSEMBLER

```
{datasegment}
ELMLEN     EQU  2
STARTIDX1  EQU  1
STARTIDX2  EQU  1
ANTDIM1    EQU  3
ITABEL2    DW   5*3  DUP (?)

IDX1       DW   ?
IDX2       DW   ?
ITAL       DW   ?

MOV  AX,ITAL
PUSH AX
MOV  AX,IDX2
SUB  AX,STARTIDX2
MOV  SI,ANTDIM1
MUL  SI
ADD  AX,IDX1
SUB  AX,STARTIDX1
MOV  SI,ELMLEN
MUL  SI
MOV  SI,AX
POP  AX
*    MOV  ITABEL[SI],AX

*    kan erstattes af:
          LEA  BX,ITABEL
          MOV  [BX+SI],AX

MOV  AX,IDX2
SUB  AX,STARTIDX2
MOV  SI,ANTDIM1
MUL  SI
ADD  AX,IDX1
SUB  AX,STARTIDX1
MOV  SI,ELMLEN
MUL  SI
MOV  SI,AX
*    MOV  AX,ITABEL[SI]
     MOV  ITAL,AX

*    kan erstattes af:
          LEA  BX,ITABEL
          MOV  AX,[BX+SI]
```

7.1.d

Implementering af sekventiel tabelgennemløb

Bjørk Busch

Når man skal gennemløbe en tabel sekventielt, kan man vælge at beregne indexadresse for hver tilgang (se foregående afsnit), man kan imidlertid også vælge at optælle adresse istedet for, hvilket giver hurtigere gennemløb.

PASCAL.

```
VAR
     TABEL:     ARRAY[1..10] of INTEGER;
     TAL:       INTEGER;

     FOR I := 1 TO 10 DO BEGIN
          TABEL[I] := 4;
          ........
          TAL := TABEL[I];
     END;
```

Omskrevet til assembler:

```
ELMLENGTH EQU  2
TABEL     DW   10 DUP(?)
TAL       DW   ?


     FORINIT:  MOV       CX,10
               MOV       SI,0            ; indexadresse
     FORBEG:
               MOV       AX,4
               MOV       TABEL[SI],AX
               ......
               MOV       AX,TABEL[SI]
               MOV       TAL,AX
     FORNEXT:  ADD       SI,ELMLENGTH    ; SI -> næste element
               LOOP
```

Udgave uden brug af symbolsk adresse

```
               LEA       SI,TABEL  ; DS:SI->TABEL[0] før start
               ......
               ......
     FORINIT:  MOV       CX,10
     FORBEG:
               MOV       AX,4
               MOV       [SI],AX
               ......
               MOV       AX,[SI]
               MOV       TAL,AX
     FORNEXT:  ADD       SI,ELMLENGTH    ; SI -> næste element
               LOOP
```

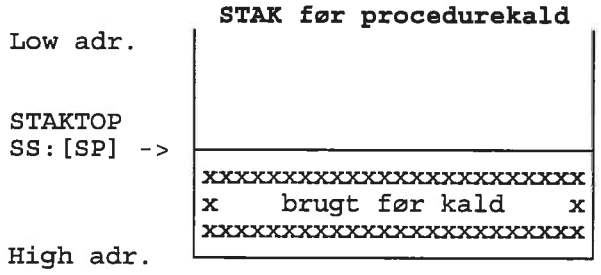
7.2.a

Implementering af procedurer og funktioner

Bjørk Busch

I assembler er der i forbindelse med procedurer og funktioner flere måder at overføre parametre og returværdier på.
Den ene måde er ved at anvende registrene til indhold eller som pointere, en anden måde er ved at anvende stakken.
Når man anvender stakken til parameter overførsel kan dette igen gøres på flere måder.
PASCAL anvender stakken på en måde og sproget C gør det en anden. Hovedforskellen er dog primært om det er det kaldende program eller underrutinen, der skal ryde op på stakken.
I PASCAL er det underrutinens opgave at ryde op på stakken og i C er det kaldende program, der selv ryder op.
I det efterfølgende vises hvordan stakken kan anvendes efter den metode, som PASCAL anvender.

Parameteroverførsel ved brug af stakken.



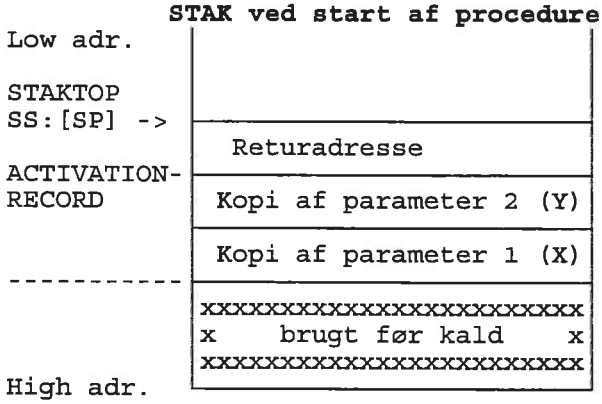
Procedurekald med value parametre.

Proceduren kan være defineret som:

PROCEDURE *minproc* (x,y:integer)

Opstart af proceduren:

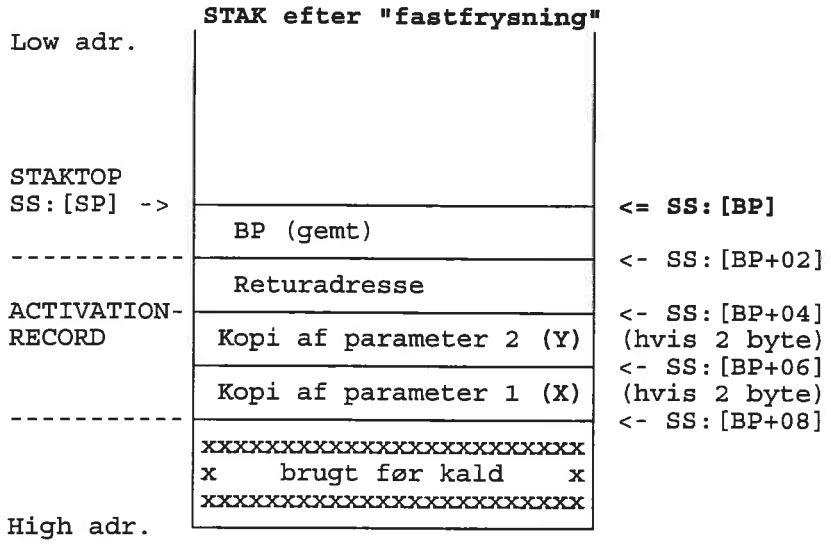
```
PUSH    xmain
PUSH    ymain
CALL    minproc
```



Indledning i procedure (PROLOG):

"FASFRYS" udganspunkt returadr. og parametre

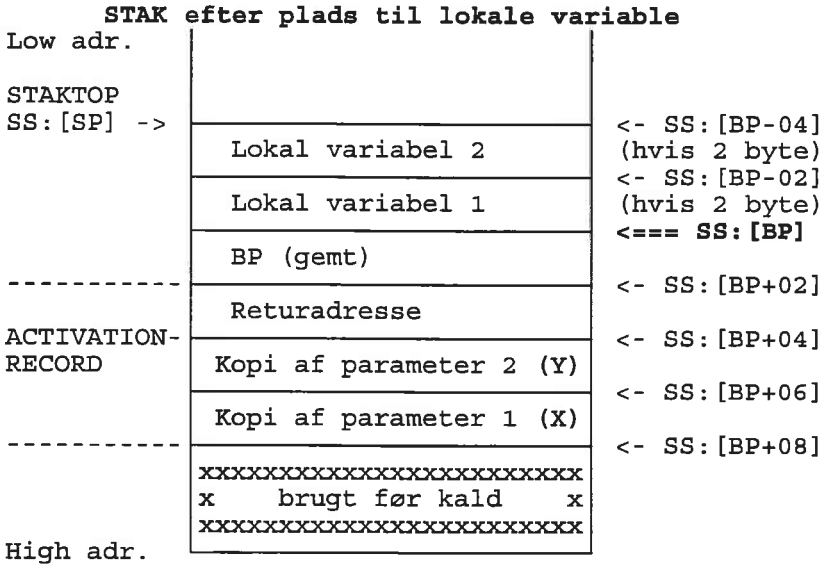
```
PUSH    BP
MOV     BP,SP
```



Indledning i procedure fortsat - hvis lokale variable:

Der skaffes hukommelse til lokale variable (her 2 integer)

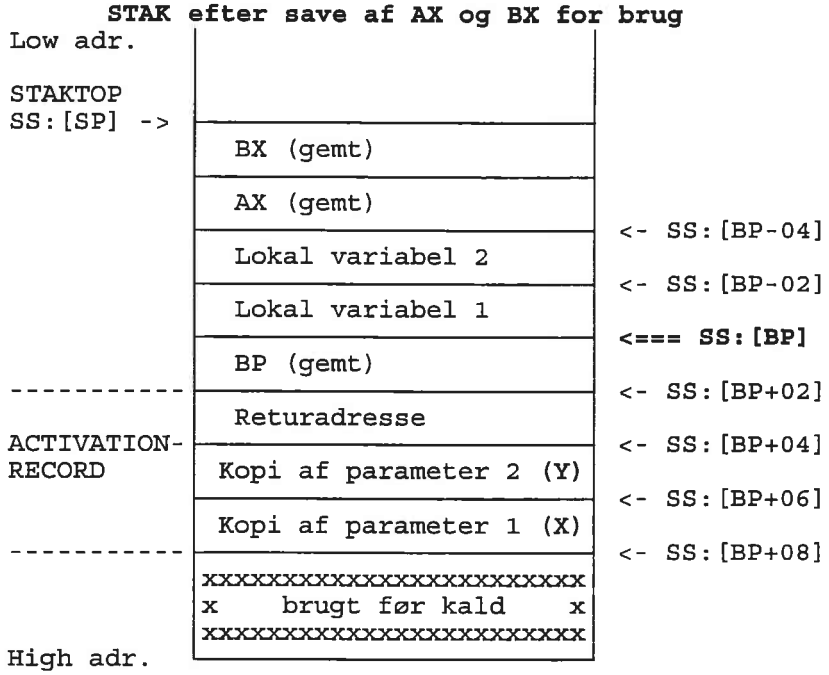
```
SUB     Sp,4
```



Senere i procedure hvor Arbejdsregistrene AX og BX gemmes:

```
PUSH    AX
PUSH    BX
```

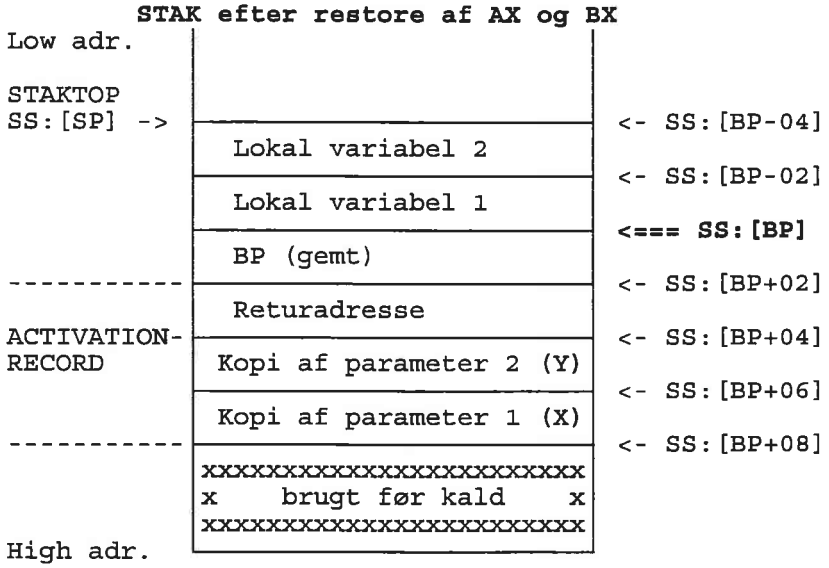
STAK efter save af AX og BX for brug



Før afslutning af procedure hvor Arbejdsregistrene AX og BX bliver reetableret:

```
POP     BX
POP     AX
```

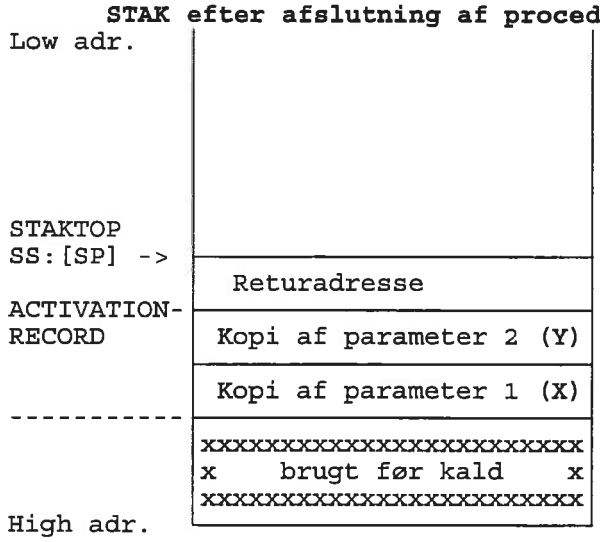
STAK efter restore af AX og BX



Afslutning af procedure (EPILOG)

```
MOV     SP,BP
POP     BP
```

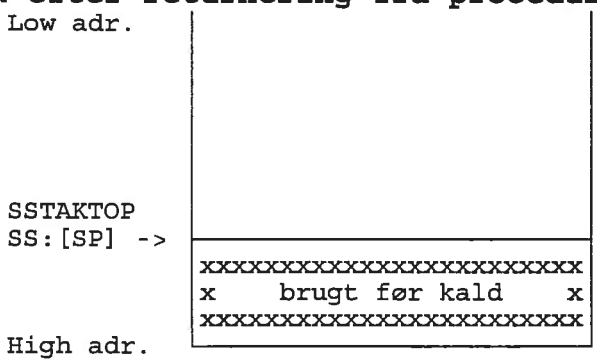
STAK efter afslutning af procedure



Returnering fra procedure (parametre fyldte 4 bytes (2 word))

```
RET     4           ;returner og ryd op på stak
```

STAK efter returnering fra procedure



Procedurer med returparametre.

I forbindelse med procedurer med returparametre overføres **adressen** på parametrene på stakken istedet for indholdet. Procedureindledningen er som tidligere beskrevet, og stakken ser efter denne således ud for en procedure med to VAR-parametre og een lokal variabel af typen integer/word.

Proceduren kan være defineret som:

```
PROCEDURE PROC2 (VAR X,Y: INTEGER)
```

Opstart af proceduren:

```
LEA     Ax,xmain
PUSH    Ds
PUSH    Ax
LEA     Ax,ymain
PUSH    Ds
PUSH    Ax
CALL    PROC2
```

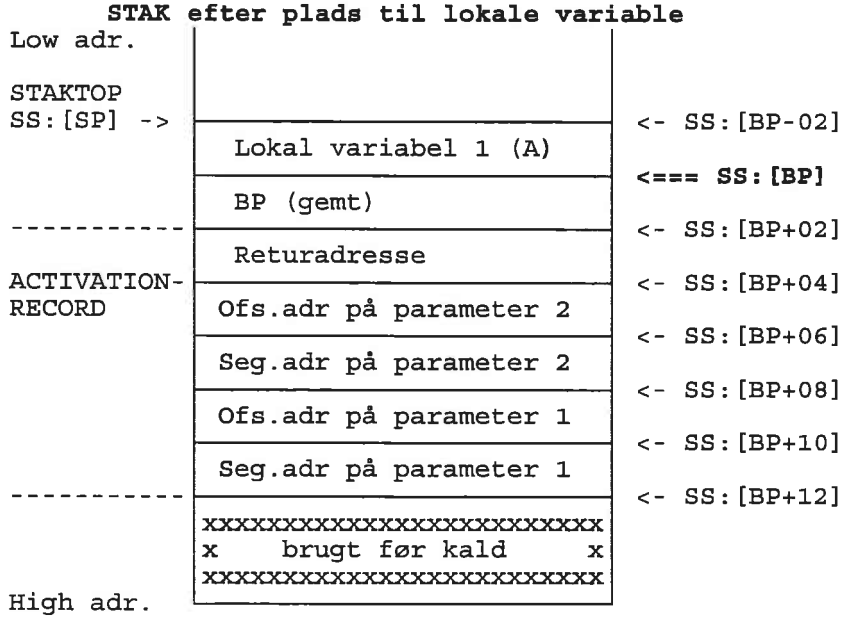
Indledning i procedure (PROLOG):

"FASFRYS" udganspunkt, returadr., parametre og afsæt plads til lokalvariabel.

```
PUSH    BP
MOV     BP,SP

SUB     Sp,2
```

STAK efter plads til lokale variable



Hvis proceduren skal ombytte X og Y på følgende måde:

```
A := X;
X := Y;
Y := A;
```

Det centrale i proceduren kan nu ske på følgende måde:

```
LES     Bx,Ss:[Bp+8]        ; Bx:=ofs. og Es:=seg. på X
MOV     Ax,Es:[Bx]          ; Ax := X
MOV     Ss:[Bp-2],Ax        ; A  := Ax

LES     Bx,Ss:[Bp+4]        ; Es:Bx := adr. på Y
MOV     Ax,Es:[Bx]          ; Ax := Y
LES     Bx,Ss:[Bp+8]        ; Es:Bx := adr. på X
MOV     Es:[Bx],Ax          ; X := Ax

MOV     Ax,[Bp-2]           ; Ax := A
LES     Bx,Ss:[Bp+4]        ; Es:Bx := adr. på Y
MOV     Es:[Bx],Ax          ; Y := Ax
```

Bemærk at LES-instruktionen kan erstatte to MOV-instruktioner
Som det sikker fremgår, er det muligt at spare nogle instruktioner, hvis man bruger flere arbejdsregistre.

Afslutning af procedure (EPILOG)

```
MOV     SP,BP
POP     BP
```

Returnering fra procedure (**adresse-parametre fyldte 8 bytes** (hver adresse var på 2*2 word))

```
RET     8           ;returner og ryd op på stak
```

Eksempel på hvordan procedure GETDATE kan se ud.

Proceduren er i pascal defineret på følgende måde:

Procedure GetDate (var aa:integer; var mm, dd, ugedg: byte);

```
GetDate Proc    Near
        PUSH    Bp              ;gem basispointer
        MOV     Bp,Sp           ;Bp -> staktop
                                ;   sidste parameter er 4 højere

        PUSH    Ax              ;gem arbejdsregistre
        PUSH    Bx
        PUSH    Cx
        PUSH    Dx
        PUSH    Es

        MOV     Ah,2Ah          ;funktion GET DATE
        INT     21h             ;Dos service funktion

        LES     Bx,Ss:[Bp+4]    ;Es:Bx := adr. på sidste parm
        MOV     Es:[Bx],Al      ;returner ugedag

        LES     Bx,Ss:[Bp+8]    ;Es:Bx := adr. på 2. sidste parm
        MOV     Es:[Bx],Dl      ;returner dag

        LES     Bx,Ss:[Bp+12]   ;Es:Bx := adr. på 3. sidste parm
        MOV     Es:[Bx],Dh      ;returner måned

        LES     Bx,Ss:[Bp+16]   ;Es:Bx := adr. på 4. sidste parm
        MOV     Es:[Bx],Cx      ;returner årstal

        POP     Es              ;restore arbejdsregistre
        POP     Dx
        POP     Cx
        POP     Bx
        POP     Ax

        POP     Bp              ; restore basispointer
        RET     16              ;returner og juster stak (parm 4*(2 ord))
GetDate EndP
```

Eksempel på brug af proceduren:

Variable i datasegmentet, som antages at være sat op i DS.

```
.......................................
AARSTAL DW      ?
MAANED  DB      ?
DAG     DB      ?
UGEDAG  DB      ?
.......................................
        LEA     AX,AARSTAL
        PUSH    DS              ; 1. parameters adresse
        PUSH    AX              ;     lægges på stak
        LEA     AX,MAANED
        PUSH    DS              ; 2. parameters adresse
        PUSH    AX              ;     lægges på stak
        LEA     AX,DAG
        PUSH    DS              ; 3. parameters adresse
        PUSH    AX              ;     lægges på stak
        LEA     AX,UGEDAG
        PUSH    DS              ; 4. parameters adresse
        PUSH    AX              ;     lægges på stak
        CALL    GetDate
.......................................
```

Funktioner:

I forbindelse med **Funktioner** returneres **funktionsværdien** på stakken.

Det kaldende program afsætter plads til funktionsværdien.

Bliver funktionen kaldt med parametre, overføres disse på samme måde som der gælder for procedurer, idet der først sættes plads til funktionsværdien og herefter udlægges parametre.

Funktionen kan være defineret som:

FUNCTION *SUM* (X, Y: INTEGER) : INTEGER

STAK efter plads til lokale variable

```
Low adr.
STAKTOP
SS:[SP] ->  |                          |   <- SS:[BP-02]
            |  Lokal variabel 1 (A)    |
            |                          |   <=== SS:[BP]
            |  BP (gemt)               |
-----------  |                          |   <- SS:[BP+02]
            |  Returadresse            |
ACTIVATION-  |                          |   <- SS:[BP+04]
RECORD      |  Kopi af parameter 2 (Y) |
            |                          |   <- SS:[BP+06]
            |  Kopi af parameter 1 (X) |
            |                          |   <- SS:[BP+08]
            |  Funktionsværdi (indhold)|
-----------  |                          |   <- SS:[BP+10]
            |xxxxxxxxxxxxxxxxxxxxxxxxxx|
            |x    brugt før kald     x|
            |xxxxxxxxxxxxxxxxxxxxxxxxxx|
High adr.
```

Hvis funktionen skal returnere summen af de 2 parametre kan den se ud på følgende måde:

```
SUM     Proc    Near
        PUSH    Bp              ; gem basispointer
        MOV     Bp,Sp           ; Bp -> staktop
                                ;    sidste parameter er 4 højere
        PUSH    Ax              ; gem arbejdsregistre
        MOV     Ax,SS:[Bp+6]    ; Ax := X
        ADD     Ax,SS:[Bp+4]    ; Ax := Ax + Y
        MOV     SS:[Bp+8],Ax    ; funktionsværdi := Ax
        POP     Bp              ; restore basispointer
        RET     4               ; returner og juster stak, idet
                                ;     funktionsværdi efterlades på stak-top
SUM     EndP
```

Eksempel på brug af funktionen:

Variable i datasegmentet, som antages at være sat op i DS.

```
.......................................
TAL1    DW      3
TAL2    DW      4
SUMTAL  DW      ?
.......................................
        SUB     SP,2            ; Afsæt plads til funktionsværdi
        PUSH    TAL1            ; overfør parameter 1 til stak (kopi)
        PUSH    TAL2            ; overfør parameter 2 til stak (kopi)
        CALL    SUM
        POP     SUMTAL          ; SUMTAL := funktionsværdi
.......................................
```

Bemærk at PUSH og POP kun kan anvendes ved word-parametre, ellers må man selv justere SP-register og bruge MOV-instruktionen til at flytte data med.

Flere eksempler på procedurer og funktioner.

PROCEDURE READCHAR (VAR TEGN: CHAR);

Hent tegn fra tastaturbuffer.
Procedure kodet i assembler.

```
ReadChar Proc    Near
         PUSH    Bp              ;gem basispointer
         MOV     Bp,Sp           ;Bp -> staktop
                                 ;   sidste parameter er 4 højere
         PUSH    Ax              ;gem arbejdsregistre
         PUSH    Bx
         PUSH    Es
         MOV     Ah,08h          ;funktion INPUT - ECHO + CTRL C
         INT     21h             ;Dos service funktion
         LES     Bx,Ss:[Bp+4]    ;Es:Bx := adr. på sidste parm
         MOV     Es:[Bx],Al      ;returner tegn
         POP     Es              ;restore arbejdsregistre
         POP     Bx
         POP     Ax

         POP     Bp              ; restore basispointer
         RET     4               ;returner og juster stak
ReadChar EndP
```

Anvendelse fra assembler.

```
TEGN     DB      ?
.......................................
        LEA     AX,TEGN
        PUSH    DS              ; 1. parameters adresse
        PUSH    AX              ;     lægges på stak
        CALL    ReadChar
```

Samme rutine men som funktion:

FUNCTION READKEY : CHAR;

Procedure kodet i assembler.

```
ReadKey  Proc    Near
         PUSH    Bp              ;gem basispointer
         MOV     Bp,Sp           ;Bp -> staktop
                                 ;   sidste parameter er 4 højere
         PUSH    Ax              ;gem arbejdsregistre
         MOV     Ah,08h          ;funktion INPUT - ECHO + CTRL C
         INT     21h             ;Dos service funktion
         MOV     Ss:[Bp+4],Al    ;Returner funktionsværdi
         POP     Ax

         POP     Bp              ; restore basispointer
         RET     0               ; returner og juster stak
ReadKey EndP
```

Anvendelse fra assembler.

```
TEGN     DB      ?
.......................................
        SUB     Sp,1            ; Sæt plads til funktionsværdi
        CALL    ReadKey
        MOV     Bp,Sp           ; Hent funktionsværdi
        MOV     Al,Ss:[Bp]      ;   fra stakken
        ADD     Sp,1            ; Fjern fra stak
        MOV     TEGN,Al         ; Overfør funktionsværdi
```

Som det kan ses er det noget besværlig når der arbejdes med ulige antal bytes på stak. Der kan anvendes word istedet, hvor den ene byte så blot ignoreres.

PROCEDURE WRITECHR (TEGN: CHAR);

Skriv tegn på skærm.
procedure kodet i assembler.

```
WriteChr Proc    Near
         PUSH    Bp                ;gem basispointer
         MOV     Bp,Sp             ;Bp -> staktop
                                   ;   sidste parameter er 4 højere
         PUSH    Ax                ;gem arbejdsregistre
         PUSH    Dx
         MOV     Dl,Ss:[Bp+4]      ;Dl := parameterværdi
         MOV     Ah,02h            ;funktion OUTPUT CHARACTER
         INT     21h               ;Dos service funktion
         POP     Dx                ;restore arbejdsregistre
         POP     Ax

         POP     Bp                ;restore basispointer
         RET     1                 ;returner og juster stak
WriteChr EndP
```

Opstart fra assembler.

```
TEGN     DB      ?
.......................................
        SUB     Sp,1               ; Plads til parameter
        MOV     Bp,Sp              ; Overfør parameter
        MOV     Al,Tegn            ;   til stak
        MOV     Ss:[Bp],Al         ;(ulige antal byte er besværlig)
        CALL    WriteChr
```

Som det kan ses er det noget besværlig når der arbejdes med ulige antal bytes på stak. Der kan anvendes word istedet, hvor den ene byte så blot ignoreres.

Hvis der anvedes word istedet ville procedure og kald se således ud:

```
WriteChr Proc    Near
         PUSH    Bp                ;gem basispointer
         MOV     Bp,Sp             ;Bp -> staktop
                                   ;   sidste parameter er 4 højere
         PUSH    Ax                ;gem arbejdsregistre
         PUSH    Dx
         MOV     Dl,SS:[Bp+4]      ;Dl := parameterværdi
         MOV     Ah,02h            ;funktion OUTPUT CHARACTER
         INT     21h               ;Dos service funktion
         POP     Dx                ;restore arbejdsregistre
         POP     Ax

         POP     Bp                ;restore basispointer
         RET     2                 ;returner og juster stak
WriteChr EndP
```

Opstart fra assembler.

```
TEGN     DB      ?
.......................................
        PUSH    TEGN               ;Overfør parameter til stak
        CALL    WriteChr
```

Det blev det noget enklere af ikke !!!

Bemærk at proceduren skal tage 1 mere af stakken.

I praksis anvendes normalt 2 byte, da det er enklere.

FUNCTION GETCHARXY (X, Y: Integer);

Hent tegn fra skærmbuffer position X,Y (hjørne = 1,1).
Procedure kodet i assembler.

```
SKRMSEG     EQU     0B800h          ;Kunne også findes via. skærmmode

GetCharXY   Proc    Near

            PUSH    Bp              ;gem basispointer
            MOV     Bp,Sp           ;Bp -> staktop
                                    ;   sidste parameter er 4 højere

            PUSH    Ax              ;gem arbejdsregistre
            PUSH    Si
            PUSH    Es

;                                           ;indexadresse=(((Y-1)*80)+X-1)*2
            MOV     Ax,Ss:[Bp+4]    ;Ax := Y
            DEC     Ax              ;ryk start til 0
            MOV     Si,80           ;tegn pr. linie
            MUL     Si              ;Si := start tegn for linie
            ADD     Ax,Ss:[Bp+6]    ;adder kolonne = X
            DEC     Ax              ;ryk X-start til 0
            SHL     Ax,1            ;Ax := tegnoffset 2 byte pr. tegn
            MOV     Si,Ax           ;Si := udvalgt tegnadresse

            MOV     Ax,SKRMSEG
            MOV     Es,Ax           ;opsæt skærmbuffer segmentadr.
            MOV     Al,Es:[Si]      ;hent ascii fra skærmbuffer

            XOR     Ah,Ah           ;udvid til word - nulstil Ah
            MOV     Ss:[Bp+8],Ax    ;overfør funktionsværdi til stak

            POP     Es
            POP     Si
            POP     Ax

            POP     Bp              ;restore basispointer
            RET     4               ;returner og juster stak
                                    ;funktionsværdi bliver på stak
GetCharXY   EndP
```

Opstart fra assembler.

```
TEGN        DB      ?
X           DW      4
Y           DW      12
.......................................
        SUB     Sp,2            ;plads til funktionsværdi
        PUSH    X               ;1. parameters indhold på stak
        PUSH    Y               ;2. parameters indhold på stak
        CALL    GetCharXY
        POP     Ax              ;Hent funktionsværdi og
        MOV     Tegn,Al         ;overfør den til Tegn.
```

Det skal bemærkes, at der her er anvendt word parametre i procedure og "afkortning" først sker efter kaldet ved overførselen af funktionsværdien til TEGN.

Det er muligt at gøre procedurerne mere "læsevenlige" ved brug af EQU sætninger. Dette vises på næste side med samme procedure.

FUNCTION GETCHARXY (X, Y: Integer);

Denne gang mere "læsevenlig" ved brug af EQU-sætninger.

```
SKRMSEG     EQU     0B800h
Y           EQU     WORD PTR Ss:[Bp+4]      ; 2. parameter
X           EQU     WORD PTR Ss:[Bp+6]      ; 1. parameter
FUNK        EQU     WORD PTR Ss:[Bp+8]      ; funktionsværdi - retur

GetCharXY   Proc    Near
            PUSH    Bp                  ;gem basispointer
            MOV     Bp,Sp               ;Bp -> staktop
                                        ;   sidste parameter er 4 højere

            PUSH    Ax                  ;gem arbejdsregistre
            PUSH    Di
            PUSH    Es

            MOV     Ax,Y                ;Ax := Y
            DEC     Ax                  ;ryk start til 0
            MOV     Di,80               ;tegn pr. linie
            MUL     Di                  ;Di := start tegn for linie
            ADD     Ax,X                ;adder X
            DEC     Ax                  ;ryk start til 0
            SHL     Ax,1                ;Ax := tegnoffset 2 byte pr. tegn
            MOV     Di,Ax               ;Di := udvalgt tegnadresse
            MOV     Ax,SKRMSEG
            MOV     Es,Ax               ;opsæt skærmbuffer segmentadr.
            MOV     Al,Es:[Di]          ;hent ascii fra skærmbuffer
            XOR     Ah,Ah
            MOV     FUNK,Ax             ;overfør funktionsværdi til stak

            POP     Es
            POP     Di
            POP     Ax

            POP     Bp                  ;restore basispointer
            RET     4                   ;returner og juster stak
                                        ;funktionsværdi bliver på stak
GetCharXY   EndP
```

Under oversættelsen erstattes symbolnavne fra EQU med det de er defineret som og der er derfor ingen forskel på den oversatte procedure i forhold til foregående udgave.

Prøv selv at lave følgende procedurer/funktioner med tilhørende eksempel på kald:

Du kan også prøve at afteste under CODEVIEW.

Til løsning kan anvendes BIOS-kald 10h.

1) PROCEDURE GOTOXY (X, Y: Integer)
   Placer cursor

2) FUNCTION GETX : Integer; og FUNCTION GETY : Integer;
   Returner cursor position, henholdsvis X og Y koordinat.

3) PROCEDURE GETXY (VAR X, Y: Integer);
   Returner både X og Y koordinat

7.5.a

Programeksempel i pascal

Bjørk Busch

Dette programeksempel er udgangspunkt for senere implementering med inline assembler og implementering i "ren" assembler.

```
{$R-} {$S-} {$I-}      {compiler-direktiver  -range -stak -io check}
{$V-}                  {ingen strengtypecheck}
Program SumSnit;
Type
   StalType = String[5];
   ItabType = Array[1..100] of Integer;

Procedure Bin2Asc(Ital: Integer; Var Stal: StalType);
Var
   I:    Integer;
   Wtal: Integer;
begin
   Stal[0] := Chr(5);
   For I := 5 downto 1 do begin
      Wtal := Ital MOD 10;
      Stal[I] := Chr(Wtal+Ord('0'));
      Ital := Ital DIV 10;
   end;
end;

Procedure WriteStr(Var Streng: String);
Var
   I    : Integer;
   Antal: Integer;
begin
   Antal := Length(Streng);
   I := 1;
   While Antal > 0 do begin
      Write(Streng[I]);
      I := I + 1;
      Antal := Antal - 1;
   end;
end;

Procedure WriteInt(Ital: Integer);
Var
   Stal: StalType;

begin
   Bin2Asc(Ital,Stal);
   WriteStr(Stal);
   WriteLN;
end;

Procedure WriteIntTab(Antal: Integer; Var Itab: ItabType);
Var
   I:    Integer;
   Ital: Integer;
begin
   I := 1;
   While Antal > 0 do begin
      Ital := Itab[I];
      WriteInt(Ital);
      I := I + 1;
      Antal := Antal - 1;
   end;
end;
```

```
Procedure Asc2Bin(Var Stal: StalType; Var Ital: Integer);
Var
   I:     Integer;
   L:     Integer;
   Wtal1: Integer;
   Wtal2: Integer;
begin
   L := Ord(Stal[0]);
   Wtal1 := 0;
   For I := 1 to L do begin
      Wtal1 := Wtal1 * 10;
      Wtal2 := Ord(Stal[I]) - Ord('0');
      Wtal1 := Wtal1 + Wtal2;
   end;
   Ital := Wtal1;
end;

Procedure ReadInt(Var Tekst: String; Var Ital: Integer);
Var
   Stal:  StalType;
   Wtal:  Integer;
begin
   WriteStr(Tekst);
   ReadLN(Stal);
   Asc2Bin(Stal,Wtal);
   Ital := Wtal;
end;

Procedure ReadIntTab(Var Tekst: String; Antal: Integer;
                     Var Itab: ItabType);
Var
   I:    Integer;
   Ital: Integer;
begin
   I := 1;
   While Antal > 0 do begin
      ReadInt(Tekst,Ital);
      Itab[I] := Ital;
      I := I + 1;
      Antal := Antal - 1;
   end;
end;

Procedure IntTabSum(Antal: Integer; Var Itab: ItabType;
                    Var Sum: Integer);
Var
   I:    Integer;
   Wsum: Integer;
begin
   I := 1;
   Wsum := 0;
   While Antal > 0 do begin
      Wsum := Wsum +  Itab[I];
      I := I + 1;
      Antal := Antal - 1;
   end;
   Sum := Wsum;
end;
Procedure WriteIntTabSum(Antal: Integer; Var Itab: ItabType);
Var
   Sum: Integer;
begin
   IntTabSum(Antal,Itab,Sum);
   WriteInt(Sum);
end;
Procedure IntTabGsnit(Antal: Integer; Var Itab: ItabType;
                      Var Gsnit: Integer);
Var
   Wsum:   Integer;
begin
   IntTabSum(Antal,Itab,Wsum);
   Gsnit := Wsum DIV Antal;
end;
```

```
Procedure WriteIntTabGsnit(Antal: Integer; Var Itab: ItabType);
Var
   Gsnit:    Integer;
begin
   IntTabGsnit(Antal,Itab,Gsnit);
   WriteInt(Gsnit);
end;




{Main}


Var
  Tekst:     String;
  Itab:      ItabType;
  Antal:     Integer;
  Sum:       Integer;
  Gsnit:     Integer;

begin

   Tekst := 'Beregning af sum og gennemsnit'#13#10;
   WriteStr(Tekst);

   Tekst := 'Intast positivt heltal: ';
   Antal := 5;
   ReadIntTab(Tekst,Antal,Itab);

   Tekst := 'Oversigt over tallene:'#13#10;
   WriteStr(Tekst);

   WriteIntTab(Antal,Itab);

   Tekst := 'Summen er: ';
   WriteStr(Tekst);

   WriteIntTabSum(Antal,Itab);

   Tekst := 'Gennemsnittet er: ';
   WriteStr(Tekst);

   WriteIntTabGsnit(Antal,Itab);

   ReadLN;

end.
```

7.5.b

Programeksempel i pascal med inline assembler

Bjørk Busch

Dette programeksempel bygger på det foregående pascalprogram, men implementeringen er i nedenstående eksempel foretaget med inline assembler.

Der er i denne oversættelse ikke foretaget nogen form for optimering, idet hver pascal-sætning er oversat for sig.

Turbo-Pascal forventer kun at registrene Ds, Ss, Sp og Bp intakte, og i eksemplet er det derfor kun disse registre, der reetableres i forbindelse med procedurerne.

I forbindelse med indlæsning med READLN har det været nødvendigt, at definere en indlæsningsbuffer, der er anvendet istedet for den oprindelige streng "Stal", og der er ikke foretaget kopiering af data til denne oprindelige streng.

I main-rutinen er tilskrivninger af faste tekster til streng-variable ikke oversat. Disse tilskrivninger vil kræve en flytning af hvert eneste karakter for sig.

```
{$R-} {$S-} {$I-}     {compiler-direktiver  -range -stak -io check }
{$V-}                 {ingen strengtypecheck}
Program SumSnit;

Type

   StalType = String[5];
   ItabType = Array[1..100] of Integer;

Procedure Bin2Asc(Ital: Integer; Var Stal: StalType);
Var
   I:    Integer;
   Wtal: Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Wtal       Equ       [Bp-4]
   I          Equ       [Bp-2]
   Stal       Equ       [Bp+4]              adresse
   Ital       Equ       [Bp+8]              value
-----------------------------------------------------------------}
   ASM
        LEs       Bx,Ss:[Stal]       {Stal[0] := Chr(5);          }
        Mov       Al,5
        Mov       Es:[Bx],Al
{ForInit}
        Mov       Ax,5
        Mov       Ss:[I],Ax          {For I := 5 downto 1 do begin}
@ForTst:
        Mov       Ax,Ss:[I]
        Cmp       Ax,1
        Jl        @EndFor
{ForSave}
        Push      Ss:[I]
{ForBody}
        Mov       Bx,10              {   Wtal := Ital MOD 10;     }
        Mov       Ax,Ss:[Ital]
        Cwd
        Div       Bx                 {  ; Ax=Kvotient, Dx=Rest    }
        Mov       Ss:[Wtal],Dx

        Mov       Ax,Ss:[Wtal]       {   Stal[I] := Chr(Wtal+     }
        Add       Ax,'0'             {        Ord('0'));          }
        Mov       Di,Ss:[I]          {   ; udregn adr. på Stal[I] }
        LEs       Bx,Ss:[Stal]       {   ; NB!  elm.længde=1 byte }
        Mov       Es:[Bx+Di],Al      {   ; NB!  start=0           }

        Mov       Bx,10              {   Ital := Ital DIV 10;     }
        Mov       Ax,Ss:[Ital]       {   Ital := Ital DIV 10;     }
        Cwd
        Div       Bx                 {   ; Ax=Kvotient, Dx=Rest   }
        Mov       Ss:[Ital],Ax

{ForRestore}                         {End                         }
        Pop       Ss:[I]
{ForNext}
        Dec       Word PTR Ss:[I]
        Jmp       @ForTst
@EndFor:
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp              retabler stak
              Pop       Bp
              Ret       6          2+4     fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure WriteStr(Var Streng: String);
Var
   I     : Integer;
   Antal: Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Antal      Equ       [Bp-4]
   I          Equ       [Bp-2]
   Streng     Equ       [Bp+4]              adresse
-----------------------------------------------------------------}
   ASM
        LEs       Bx,Ss:[Streng]          { Antal:=Length(Streng);}
        Mov       Al,Es:[Bx]
        Cbw                               {    ; Ax := Al;        }
        Mov       Ss:[Antal],Ax

        Mov       Word PTR Ss:[I],1       { I := 1;               }

@WhileTst:                                { While Antal > 0       }
        Cmp       Word PTR Ss:[Antal],0
        Jle       @EndWhile
@WhileBody:                               { do begin              }
                                          {    Write(Streng[I]);  }
        Mov       Si,Ss:[I]               ; udregn adr.på Stal[I]
        LEs       Bx,Ss:[Streng]          ; NB! elm.længde=1 byte
        Mov       Dl,Es:[Bx+Si]           ; NB! start=0           }
        Mov       Ah,02
        Int       21h

        Inc       Word PTR Ss:[I]         {    I := I + 1;        }
        Dec       Word PTR Ss:[Antal]     {    Antal := Antal - 1;}
        Jmp       @WhileTst               { end;                  }

@EndWhile:
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       4                   fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure WriteInt(Ital: Integer);
Var
   Stal: StalType;

begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,6                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Stal       Equ       [Bp-6]
   Ital       Equ       [Bp+4]              value
-----------------------------------------------------------------}
   ASM
        Mov       Ax,[Ital]             { Bin2Asc(Ital,Stal);  }
        Push      Ax
        Lea       Bx,[Stal]
        Push      Ss                    { adr. på Stal }
        Push      Bx
        Call      Bin2Asc

        Lea       Bx,[Stal]             { WriteStr(Stal); }
        Push      Ss                    { adr. på Stal }
        Push      Bx
        Call      WriteStr

        Mov       Ah,2                  { WriteLN }
        Mov       Dl,13
        Int       21h
        Mov       Dl,10
        Int       21h
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       2                   fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure WriteIntTab(Antal: Integer; Var Itab: ItabType);
Var
   I:    Integer;
   Ital: Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Ital       Equ       [Bp-4]
   I          Equ       [Bp-2]
   Itab       Equ       [Bp+4]              adresse
   Antal      Equ       [Bp+8]              value
-----------------------------------------------------------------}
   ASM
        Mov       Word PTR Ss:[I],1       { I := 1;                }
@WhileTst:                                { While Antal > 0        }
        Cmp       Word PTR Ss:[Antal],0
        Jle       @EndWhile
@WhileBody:                               { do begin               }
        Mov       Ax,Ss:[I]               {    Ital := Itab[I];    }
        Sub       Ax,1                { juster til startindex 0 }
        Mov       Bx,2                { elementlængde = 2       }
        Mul       Bx                  { beregn indexadr.        }
        Mov       Si,Ax
        LEs       Bx,Ss:[Itab]
        Mov       Ax,Es:[Bx+Si]
        Mov       Ss:[Ital],Ax

        Push      Ss:[Ital]               {    WriteInt(Ital);     }
        Call      WriteInt

        Inc       Ss:[I]                  {    I := I + 1;         }

        Dec       Ss:[Antal]              {    Antal := Antal - 1;}

        Jmp       @WhileTst               { end;                   }
@EndWhile:
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       6                   fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure Asc2Bin(Var Stal: StalType; Var Ital: Integer);
Var
   I:      Integer;
   L:      Integer;
   Wtal1:  Integer;
   Wtal2:  Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,8                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Wtal2      Equ       [Bp-8]
   Wtal1      Equ       [Bp-6]
   L          Equ       [Bp-4]
   I          Equ       [Bp-2]
   Ital       Equ       [Bp+4]              adresse
   Stal       Equ       [Bp+8]              adresse
-----------------------------------------------------------------}
   ASM
        LEs       Bx,Ss:[Stal]        {L := Ord(Stal[0]);}
        Mov       Al,Es:[Bx]
        Cbw
        Mov       Ss:[L],Ax

        Mov       Word PTR Ss:[Wtal1],0    { Wtal1 := 0;}

{ForInit}
        Mov       Ax,1
        Mov       Ss:[I],Ax           { For I := 1 to L do begin  }
@ForTst:
        Mov       Ax,Ss:[I]
        Cmp       Ax,L
        Jg        @EndFor
{ForSave}
        Push      Ss:[I]
{ForBody}
        Mov       Ax,Ss:[Wtal1]       {    Wtal1 := Wtal1 * 10; }
        Mov       Bx,10
        Mul       Bx
        Mov       Ss:[Wtal1],Ax

        Mov       Si,Ss:[I]           {    Wtal2 := Ord(Stal[I])- }
        LEs       Bx,Ss:[Stal]
        Mov       Al,Es:[Bx+Si]       {     NB!  startadr. = 0    }
        Sub       Al,'0'              {              Ord('0');    }
        Cbw
        Mov       Ss:[Wtal2],Ax

        Mov       Ax,Ss:[Wtal2]       {    Wtal1 := Wtal1 + Wtal2;}
        Add       Ax,Ss:[Wtal1]
        Mov       Ss:[Wtal1],Ax
{ForRestore}                          { End;  }
        Pop       Ss:[I]
{ForNext}
        Inc       Word PTR Ss:[I]
        Jmp       @ForTst
@EndFor:
        Mov       Ax,Ss:[Wtal1]         { Ital := Wtal1; }
        LEs       Di,Ss:[Ital]
        Mov       Es:[Di],Ax
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       8          4+4      fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure ReadInt(Var Tekst: String; Var Ital: Integer);
Type
   BufferType = RECORD
      BufLen: Byte;
      BufArr: String[6];
   END;
Var
   Stal:  StalType;
   Wtal:  Integer;
   Buf:   BufferType;          { bruges istedet for orpindelig Stal}
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,16                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Buf        Equ       [Bp-16]
   Buf.BufLen Equ       [Bp-16]
   Buf.BufArr Equ       [Bp-15]
   Wtal       Equ       [Bp-8]
   Stal       Equ       [Bp-6]
   Ital       Equ       [Bp+4]               adresse
   Tekst      Equ       [Bp+8]               adresse
-----------------------------------------------------------------}
   ASM
        LEs       Bx,[Tekst]            { WriteStr(Tekst);  }
        Push      Es                    { adr. på Tekst }
        Push      Bx
        Call      WriteStr

        Push      Ds                    { ReadLN(Stal) }
        Mov       Ax,Ss
        Mov       Ds,Ax
        Mov       Al,6
        Mov       Ss:[Buf.Buflen],Al
        Lea       Dx,[Buf]
        Mov       Ah,0Ah
        Int       21h
        Mov       Dl,10
        Mov       Ah,2
        Int       21h
        Pop       Ds

        Lea       Bx,[Buf.Bufarr]       { Asc2Bin(Buffarr,Wtal);  }
        Push      Ss                    { adr. på lokalvar. BufArr}
        Push      Bx
        Lea       Bx,[Wtal]             { adr. på lokalvar Wtal }
        Push      Ss
        Push      Bx
        Call      Asc2Bin

        Mov       Ax,Ss:[Wtal]          { Ital := Wtal }
        LEs       Bx,[Ital]
        Mov       Es:[Bx],Ax
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp                retabler stak
              Pop       Bp
              Ret       8          4+4       fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure ReadIntTab(Var Tekst: String; Antal: Integer;
                     Var Itab: ItabType);
Var
   I:    Integer;
   Ital: Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Ital       Equ       [Bp-4]
   I          Equ       [Bp-2]
   Itab       Equ       [Bp+4]              adresse
   Antal      Equ       [Bp+8]              value
   Tekst      Equ       [Bp+10]             adresse
-----------------------------------------------------------------}
   ASM
        Mov       Word PTR Ss:[I],1      { I := 1;                  }
@WhileTst:                              { While Antal > 0 do begin}
        Cmp       Word PTR Ss:[Antal],0
        Jle       @EndWhile
{While-body}
        LEs       Bx,[Tekst]             {    ReadInt(Tekst,Ital); }
        Push      Es                     { adr. på Tekst }
        Push      Bx
        Lea       Bx,[Ital]
        Push      Ss                     { adr. på lokalvar. Ital }
        Push      Bx
        Call      ReadInt

        Mov       Ax,Ss:[Ital]           {    Itab[I] := Ital;      }
        Push      Ax                     {    ;gem ital             }
        Mov       Ax,Ss:[I]              {    ;beregn indexadr.     }
        Sub       Ax,1                 { juster til startindex 0 }
        Mov       Bx,2                 { elementlængde = 2       }
        Mul       Bx                   { beregn indexadr.        }
        Mov       Di,Ax
        LEs       Bx,Ss:[Itab]
        Pop       Ax                     {    ;hent ital            }
        Mov       Es:[Bx+Di],Ax

        Inc       Word PTR Ss:[I]        {    I := I + 1;           }
        Dec       Word PTR Ss:[Antal]    {    Antal := Antal - 1;   }

        Jmp       @WhileTst              { end;                     }
@EndWhile:
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       10         4+2+4    fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure IntTabSum(Antal: Integer; Var Itab: ItabType;
                    Var Sum: Integer);
Var
   I:    Integer;
   Wsum: Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Wsum       Equ       [Bp-4]
   I          Equ       [Bp-2]
   Sum        Equ       [Bp+4]              adresse
   Itab       Equ       [Bp+8]              adresse
   Antal      Equ       [Bp+12]             Value
-----------------------------------------------------------------}
   ASM
        Mov       Word PTR Ss:[I],1      { I := 1;                 }

        Mov       Word PTR Ss:[Wsum],0   { Wsum := 0;              }

@WhileTst:                               { While Antal > 0 do begin}
        Cmp       Word PTR Ss:[Antal],0
        Jle       @EndWhile
{While-body}
        Mov       Ax,Ss:[I]              {    Wsum := Wsum+Itab[I];}
        Sub       Ax,1                 { juster til startindex 0 }
        Mov       Bx,2                 { elementlængde = 2       }
        Mul       Bx                   { beregn indexadr.        }
        Mov       Si,Ax
        LEs       Bx,Ss:[Itab]
        Mov       Ax,Es:[Bx+Si]
        Add       Ax,Ss:[Wsum]
        Mov       Ss:[Wsum],Ax

        Inc       Word PTR Ss:[I]        {    I := I + 1;          }

        Dec       Word PTR Ss:[Antal]    {    Antal := Antal - 1;  }

        Jmp       @WhileTst              { end;                    }
@EndWhile:
        Mov       Ax,Ss:[Wsum]           { Sum := Wsum;            }
        LEs       Bx,Ss:[Sum]
        Mov       Es:[Bx],Ax
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp                retabler stak
              Pop       Bp
              Ret       10         2+4+4     fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure WriteIntTabSum(Antal: Integer; Var Itab: ItabType);
Var
   Sum: Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,2                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Sum        Equ       [Bp-2]
   Itab       Equ       [Bp+4]              adresse
   Antal      Equ       [Bp+8]              Value
-----------------------------------------------------------------}
   ASM
        Mov       Ax,Ss:[Antal]       { IntTabSum(Antal,Itab,Sum); }
        Push      Ax                  { value Antal }
        LEs       Bx,[Itab]           { adr. på Itab }
        Push      Es
        Push      Bx
        Lea       Bx,[Sum]            { adr. på lokalvar. Sum }
        Push      Ss
        Push      Bx
        Call      IntTabSum

        Mov       Ax,Ss:[Sum]         { WriteInt(Sum);            }
        Push      Ax                  { value Sum }
        Call      WriteInt
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       6          2+4      fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure IntTabGsnit(Antal: Integer; Var Itab: ItabType;
                      Var Gsnit: Integer);
Var
   Wsum:   Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,2                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Wsum       Equ       [Bp-2]
   Gsnit      Equ       [Bp+4]              adresse
   Itab       Equ       [Bp+8]              adresse
   Antal      Equ       [Bp+12]             Value
-----------------------------------------------------------------}
   ASM
        Mov       Ax,Ss:[Antal]       { IntTabSum(Antal,Itab,Sum); }
        Push      Ax                  { value Antal }
        LEs       Bx,[Itab]           { adr. på Itab }
        Push      Es
        Push      Bx
        Lea       Bx,[WSum]           { adr. på lokalvar. WSum }
        Push      Ss
        Push      Bx
        Call      IntTabSum

        Mov       Ax,Ss:[WSum]        { Gsnit := Wsum DIV Antal; }
        Cwd
        Mov       Bx,Ss:[Antal]
        DIV       Bx
        LEs       Bx,[Gsnit]           { adr. på Gsnit }
        Mov       Es:[Bx],Ax
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       10         2+4+4    fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Procedure WriteIntTabGsnit(Antal: Integer; Var Itab: ItabType);
Var
   Gsnit:   Integer;
begin
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,2                plads til lokal-var

   Der er desuden oprettet følgende symboler:
   Gsnit      Equ       [Bp-2]
   Itab       Equ       [Bp+4]              adresse
   Antal      Equ       [Bp+8]              Value
-----------------------------------------------------------------}
   ASM
        Mov       Ax,Ss:[Antal]      { IntTabGsnit(Antal,Itab,Sum);}
        Push      Ax                 { value Antal }
        LEs       Bx,[Itab]          { adr. på Itab }
        Push      Es
        Push      Bx
        Lea       Bx,[Gsnit]         { adr. på lokalvar. Gsnit }
        Push      Ss
        Push      Bx
        Call      IntTabGsnit

        Mov       Ax,Ss:[Gsnit]      { WriteInt(Gsnit);            }
        Push      Ax                 { value Gsnit }
        Call      WriteInt
   END;
{-----------------------------------------------------------------
   pascal har her selv indskudt følgende epilog
              Mov       Sp,Bp               retabler stak
              Pop       Bp
              Ret       6          2+4      fjern activationrecord
-----------------------------------------------------------------}
end;
```

```
Var
  Tekst:     String;
  Itab:      ItabType;
  Antal:     Integer;
  Sum:       Integer;
  Gsnit:     Integer;

begin

   Tekst := 'Beregning af sum og gennemsnit'#13#10; {ikke oversat}
   ASM
        Lea       Bx,[Tekst]            { WriteStr(Tekst); }
        Push      Ds                    { adr. på tekst }
        Push      Bx
        Call      WriteStr
   END;

   Tekst := 'Intast positivt heltal: ';            {ikke oversat}
   ASM
        Mov       Antal,2          { Antal := 2;  }
   END;
   ASM
        Lea       Bx,[Tekst]       { ReadIntTab(Tekst,Antal,Itab);}
        Push      Ds               { adr. på tekst }
        Push      Bx
        Mov       Ax,Antal         { value antal  }
        Push      Ax
        Lea       Bx,[Itab]        { adr. på Itab }
        Push      Ds
        Push      Bx
        Call      ReadIntTab
   END;

   Tekst := 'Oversigt over tallene:'#13#10;        {ikke oversat}
   ASM
        Lea       Bx,[Tekst]       { WriteStr(Tekst); }
        Push      Ds               { adr. på tekst }
        Push      Bx
        Call      WriteStr
   END;

   ASM
        Mov       Ax,Antal         { WriteIntTab(Antal,Itab); }
        Push      Ax               { value antal              }
        Lea       Bx,[Itab]        { adr. på Itab }
        Push      Ds
        Push      Bx
        Call      WriteIntTab
   END;

   Tekst := 'Summen er: ';                         {ikke oversat}
   ASM
        Lea       Bx,[Tekst]       { WriteStr(Tekst); }
        Push      Ds               { adr. på tekst }
        Push      Bx
        Call      WriteStr
   END;

   ASM
        Mov       Ax,Antal         { WriteIntTabSum(Antal,Itab); }
        Push      Ax               { value antal                 }
        Lea       Bx,[Itab]        { adr. på Itab }
        Push      Ds
        Push      Bx
        Call      WriteIntTabSum
   END;
```

```
  Tekst := 'Gennemsnittet er: ';                    {ikke oversat}
  ASM
       Lea       Bx,[Tekst]        { WriteStr(Tekst); }
       Push      Ds                { adr. på tekst }
       Push      Bx
       Call      WriteStr
  END;

  ASM
       Mov       Ax,Antal          { WriteIntTabGsnit(Antal,Itab); }
       Push      Ax                { value antal                   }
       Lea       Bx,[Itab]         { adr. på Itab }
       Push      Ds
       Push      Bx
       Call      WriteIntTabGsnit
  END;

  ReadLN;                                           {ikke oversat}

end.
```

7.5.c

Programeksempel i assembler

Bjørk Busch

Dette programeksempel bygger på det foregående pascalprogram, men implementeringen er istedet foretaget i assembler.

Programmet er optimeret med hensyn til tabeladressering ved sekventiel genemløb. Der anvendes desuden registre istedet for lokal-variable, hvor det er muligt.

```
; Program SumSnit

DATA    Segment Para Public 'DATA'
;==========================================================
Tekst   Db      128 DUP (?)             ;1+127
Itab    Dw      100 DUP (?)
Antal   Dw      ?
Sum     Dw      ?
Gsnit   Dw      ?
Tekst1  Db      'Beregning af sum og gennemsnit',13,10,0
Tekst2  Db      'Intast positivt heltal: ',0
Tekst3  Db      'Oversigt over tallene:',13,10,0
Tekst4  Db      'Summen er: ',0
Tekst5  Db      'Gennemsnittet er: ',0
;==========================================================
DATA    EndS                            ; slut på data-segment

CODE    Segment Para Public 'CODE'
;==========================================================
        Assume Cs:CODE,Ds:DATA,Es:nothing,Ss:STACK

; Proceduren InitStr anvendes til initiering af streng fra
; strengkonstant
;  Ds:Si -> nyt indhold termineret med binært nul
;  Es:Di -> modtager streng
InitStr Proc    Near
        Push    Ax
        Push    Di
        Push    Si
        Mov     Ah,0                    ; Længde
StrLoop:
        Inc     Di                      ; næste til tegn
        Mov     Al,Ds:[Si]
        Cmp     Al,0
        Je      StrLoopEnd
        Mov     Es:[Di],Al
        Inc     Ah                      ; antal tegn + 1
        Inc     Si                      ; Næste fra tegn
        Jmp     StrLoop
StrLoopEnd:
        Pop     Si
        Pop     Di
        Mov     Es:[Di],Ah
        Pop     Ax
        Ret
InitStr  EndP
```

```
;-----------------------------------------------------------------
Bin2Asc  Proc   Near     ; (Ital: Integer; Var Stal: StalType);
  aStal        Equ       [Bp+4]                ;adresse
  aItal        Equ       [Bp+8]                ;value
;-----------------------------------------------------------------
; prolog
               Push      Bp
               Mov       Bp,Sp
               Sub       Sp,0                  ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Cx
        Push      Dx
        Push      Di
        Push      Es

        LEs       Di,Ss:[aStal]    ;Stal[0] := Chr(5);
        Mov       Al,5
        Mov       Es:[Di],Al

        Mov       Cx,5             ;For I := 5 downto 1 do begin
        Add       Di,5                  ; Es:Di -> Stal[I]
                                        ;counter og index adskilles
        Mov       Ax,Ss:[aItal]
aForBegin:
        Mov       Bx,10            ;   Wtal := Ital MOD 10;

        Cwd                   ; DxAx := Ax  forbered 16bit division
        Div       Bx          ; Ax:=Ital DIV 10, Dx:=Ital MOD 10

        Add       Dl,'0'           ;   Stal[I]:= Chr(Wtal+Ord('0'));
        Mov       Es:[Di],Dl
                                   ;   Ital := Ital DIV 10;  se DIV

        Dec       Di               ;end;
        Loop      aForBegin        ;counter auto, index manuel

        Pop       Es
        Pop       Di
        Pop       Dx
        Pop       Cx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
               Mov       Sp,Bp    ;          retabler stak
               Pop       Bp
               Ret       6        ;  2+4     fjern activationrecord
;-----------------------------------------------------------------
Bin2Asc  EndP
```

```
WriteStr Proc   Near    ;(Var Streng: String);
;-----------------------------------------------------------------
  bStreng     Equ       [Bp+4]                ;adresse
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,0                  ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Cx
        Push      Dx
        Push      Si
        Push      Es

        LEs       Si,Ss:[bStreng]         ; Antal := Length(Streng);
        Mov       Cl,Byte PTR Es:[Si]
        Mov       Ch,0                    ; Cx = antal

        Inc       Si                      ; Es:Si => Streng[I]

        Cmp       Cx,0                    ; While Antal > 0 do begin
bWhileTst:
        Jle       bEndWhile
        Mov       Dl,Es:[Si]              ;    Write(Streng[I]);
        Mov       Ah,02
        Int       21h
        Inc       Si                      ;    I := I + 1;
        Dec       Cx                      ;    Antal := Antal - 1;
        Jmp       bWhileTst               ; end;

bEndWhile:
        Pop       Es
        Pop       Si
        Pop       Dx
        Pop       Cx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp                ;retabler stak
              Pop       Bp
              Ret       4                    ;fjern activationrecord
;-----------------------------------------------------------------
WriteStr EndP
```

```
WriteInt  Proc  Near    ; (Ital: Integer);
;-----------------------------------------------------------------
  cStal       Equ       [Bp-6]
  cItal       Equ       [Bp+4]              ;value
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,6                 ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Dx
        Push      Es


        Mov       Ax,[cItal]              ; Bin2Asc(Ital,Stal);
        Push      Ax
        Lea       Bx,[cStal]
        Push      Ss                     ; adr. på Stal
        Push      Bx
        Call      Bin2Asc

        Lea       Bx,[cStal]              ; WriteStr(Stal);
        Push      Ss                     ; adr. på Stal
        Push      Bx
        Call      WriteStr

        Mov       Ah,2                   ; WriteLN
        Mov       Dl,13
        Int       21h
        Mov       Dl,10
        Int       21h

        Pop       Es
        Pop       Dx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp              ; retabler stak
              Pop       Bp
              Ret       2                  ; fjern activationrecord
;-----------------------------------------------------------------
WriteInt  EndP
```

```
WriteIntTab   Proc   Near    ;(Antal: Integer; Var Itab: ItabType);
;-----------------------------------------------------------------
  dItab       Equ       [Bp+4]              ;adresse
  dAntal      Equ       [Bp+8]              ;value
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
;-----------------------------------------------------------------
        Push      Ax
        Push      Cx
        Push      Si
        Push      Es

        LEs       Si,Ss:[dItab]         ; I := 1; Es:Si=>Itab[1]

        Mov       Cx,Ss:[dAntal]        ; While Antal > 0 do begin
        Cmp       Cx,0
dWhileTst:
        Jle       dEndWhile
        Mov       Ax,Es:[Si]            ;    Ital := Itab[I];

        Push      Ax                    ;    WriteInt(Ital);
        Call      WriteInt

        Add       Si,2                  ;    I := I + 1;

        Dec       Cx                    ;    Antal := Antal - 1;

        Jmp       dWhileTst             ; end;
dEndWhile:
        Pop       Es
        Pop       Si
        Pop       Cx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp               ;retabler stak
              Pop       Bp
              Ret       6                   ;fjern activationrecord
;-----------------------------------------------------------------
WriteIntTab   EndP
```

```
Asc2Bin   Proc   Near   ;(Var Stal: StalType; Var Ital: Integer);
;-----------------------------------------------------------------
  eItal       Equ       [Bp+4]                ;adresse
  eStal       Equ       [Bp+8]                ;adresse
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,8                  ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Cx
        Push      Dx
        Push      Di
        Push      Si
        Push      Es

        LEs       Si,Ss:[eStal]          ; Es:Si=>Stal[0]
        Mov       Cl,Byte PTR Es:[Si]    ; L := Ord(Stal[0]);
        Mov       Ch,0

        Inc       Si                     ; Es:Si=>Stal[1]

        Mov       Ax,0                   ; Wtal1 := 0;

        Mov       Bx,10
        Mov       Dh,0;
eForTst:
        Cmp       Cx,0                   ; For I := 1 to L do begin
        Jle       eEndFor
        Mul       Bx                     ;    Wtal1 := Wtal1 * 10;

        Mov       Dl,Es:[Si]             ;    Wtal2 := Ord(Stal[I])
        Sub       Dl,'0'                 ;                - Ord('0');
        Add       Ax,Dx                  ;    Wtal1 := Wtal1+Wtal2;

        Inc       Si
        Dec       Cx                     ; end;

        Jmp       eForTst
eEndFor:
        LEs       Bx,Ss:[eItal]          ; Ital := Wtal1;
        Mov       Es:[Bx],Ax

        Pop       Es
        Pop       Si
        Pop       Di
        Pop       Dx
        Pop       Cx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp     ;          retabler stak
              Pop       Bp
              Ret       8         ;  4+4     fjern activationrecord
;-----------------------------------------------------------------
Asc2Bin   EndP
```

```
ReadInt   Proc   Near    ;(Var Tekst: String; Var Ital: Integer);
;-----------------------------------------------------------------
  fBuf         Equ       [Bp-10]
  fBufxBufLen  Equ       [Bp-10]
  fBufxBufArr  Equ       [Bp-09]
  fWtal        Equ       [Bp-2]
  fItal        Equ       [Bp+4]              ;adresse
  fTekst       Equ       [Bp+8]              ;adresse
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,10               ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Dx
        Push      Es

        LEs       Bx,[fTekst]           ; WriteStr(Tekst);
        Push      Es                    ; adr. på Tekst
        Push      Bx
        Call      WriteStr

        Push      Ds                    ; ReadLN(Stal)
        Mov       Ax,Ss
        Mov       Ds,Ax
        Mov       Al,6
        Mov       Ss:[fBufxBuflen],Al
        Lea       Dx,[fBuf]
        Mov       Ah,0Ah
        Int       21h
        Mov       Dl,10
        Mov       Ah,2
        Int       21h
        Pop       Ds

        Lea       Bx,[fBufxBufarr]      ; Asc2Bin(Buffarr,Wtal);
        Push      Ss                    ; adr. på lokalvar. BufArr
        Push      Bx
        Lea       Bx,[fWtal]            ; adr. på lokalvar Wtal
        Push      Ss
        Push      Bx
        Call      Asc2Bin

        Mov       Ax,Ss:[fWtal]         ; Ital := Wtal
        LEs       Bx,[fItal]
        Mov       Es:[Bx],Ax

        Pop       Es
        Pop       Dx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp     ;          retabler stak
              Pop       Bp
              Ret       8         ; 4+4      fjern activationrecord
;-----------------------------------------------------------------
ReadInt   EndP
```

```
ReadIntTab  Proc  Near  ;(Var Tekst: String; Antal: Integer;
                        ;  Var Itab: ItabType);
;-----------------------------------------------------------------
  gItal       Equ       [Bp-4]
  gI          Equ       [Bp-2]
  gItab       Equ       [Bp+4]              ;adresse
  gAntal      Equ       [Bp+8]              ;value
  gTekst      Equ       [Bp+10]             ;adresse
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4                ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Cx
        Push      Di
        Push      Es

        LEs       Di,Ss:[gItab]         ; I := 1;
                                              ; Es:Di -> Itab[I]
        Mov       Cx,Ss:[gAntal]
        Cmp       Cx,0                  ; While Antal > 0 do begin
gWhileTst:
        Jle       gEndWhile

        Push      Cx                    ; save før kald
        Push      Di                    ;   kan undværes da
        Push      Es                    ;   underprog. reetabler

        LEs       Bx,[gTekst]           ;    ReadInt(Tekst,Ital);
        Push      Es                    ; adr. på Tekst
        Push      Bx
        Lea       Bx,[gItal]
        Push      Ss                    ; adr. på lokalvar. Ital
        Push      Bx
        Call      ReadInt

        Pop       Es                    ; restore efter kald
        Pop       Di                    ;   kan undværes da
        Pop       Cx                    ;   underprog. reetabler

        Mov       Ax,Ss:[gItal]         ;    Itab[I] := Ital;
        Mov       Es:[Di],Ax

        Add       Di,2                  ;    I := I + 1;

        Dec       Cx                    ;    Antal := Antal - 1;

        Jmp       gWhileTst             ; end;
gEndWhile:

        Pop       Es
        Pop       Di
        Pop       Cx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp     ;          retabler stak
              Pop       Bp
              Ret       10        ; 4+2+4    fjern activationrecord
;-----------------------------------------------------------------
ReadIntTab  EndP
```

```
IntTabSum  Proc   Near  ;(Antal: Integer; Var Itab: ItabType;
;                         Var Sum: Integer);
;-----------------------------------------------------------------
  hWsum       Equ       [Bp-4]
  hI          Equ       [Bp-2]
  hSum        Equ       [Bp+4]              ;adresse
  hItab       Equ       [Bp+8]              ;adresse
  hAntal      Equ       [Bp+12]             ;Value
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,4               ; plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Cx
        Push      Di
        Push      Si
        Push      Es

        LEs       Si,Ss:[hItab]          ; I := 1;
                                                 ; Es:Si -> Itab[I]
        Mov       Ax,0                   ; Wsum := 0;
        Mov       Cx,Ss:[hAntal]
        Cmp       Cx,0                   ; While Antal > 0 do begin
hWhileTst:
        Jle       hEndWhile

        Add       Ax,Es:[Si]             ;    Wsum:= Wsum + Itab[I];

        Add       Si,2                   ;    I := I + 1;

        Dec       Cx                     ;    Antal := Antal - 1;

        Jmp       hWhileTst              ; end;
hEndWhile:
        LEs       Bx,Ss:[hSum]           ; Sum := Wsum;
        Mov       Es:[Bx],Ax

        Pop       Es
        Pop       Si
        Pop       Di
        Pop       Cx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp      ;         retabler stak
              Pop       Bp
              Ret       10         ;2+4+4    fjern activationrecord
;-----------------------------------------------------------------
IntTabSum  EndP
```

```
WriteIntTabSum Proc NEAR  ;(Antal: Integer; Var Itab: ItabType);
;-----------------------------------------------------------------
  iSum        Equ       [Bp-2]
  iItab       Equ       [Bp+4]              ;adresse
  iAntal      Equ       [Bp+8]              ;Value
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,2                ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Es

        Mov       Ax,Ss:[iAntal]        ; IntTabSum(Antal,Itab,Sum);
        Push      Ax                    ; value Antal
        LEs       Bx,[iItab]            ; adr. på Itab
        Push      Es
        Push      Bx
        Lea       Bx,[iSum]             ; adr. på lokalvar. Sum
        Push      Ss
        Push      Bx
        Call      IntTabSum

        Mov       Ax,Ss:[iSum]          ; WriteInt(Sum);
        Push      Ax                    ; value Sum
        Call      WriteInt

        Pop       Es
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp     ;         retabler stak
              Pop       Bp
              Ret       6         ; 2+4     fjern activationrecord
;-----------------------------------------------------------------
WriteIntTabSum EndP
```

```
IntTabGsnit Proc Near  ;(Antal: Integer; Var Itab: ItabType;
;                        Var Gsnit: Integer);
;-----------------------------------------------------------------
  jWsum       Equ       [Bp-2]
  jGsnit      Equ       [Bp+4]              ; adresse
  jItab       Equ       [Bp+8]              ; adresse
  jAntal      Equ       [Bp+12]             ; Value
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,2                ; plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Dx
        Push      Es

        Mov       Ax,Ss:[jAntal]        ; IntTabSum(Antal,Itab,Sum);
        Push      Ax                    ; value Antal
        LEs       Bx,[jItab]            ; adr. på Itab
        Push      Es
        Push      Bx
        Lea       Bx,[jWSum]            ; adr. på lokalvar. WSum
        Push      Ss
        Push      Bx
        Call      IntTabSum

        Mov       Ax,Ss:[jWSum]         ; Gsnit := Wsum DIV Antal;
        Cwd
        Mov       Bx,Ss:[jAntal]
        DIV       Bx
        LEs       Bx,[jGsnit]            ; adr. på Gsnit
        Mov       Es:[Bx],Ax

        Pop       Es
        Pop       Dx
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp       ;         retabler stak
              Pop       Bp
              Ret       10          ;2+4+4    fjern activationrecord
;-----------------------------------------------------------------
IntTabGsnit EndP
```

```
WriteIntTabGsnit Proc Near  ;(Antal: Integer; Var Itab: ItabType);
;-----------------------------------------------------------------
  kGsnit      Equ       [Bp-2]
  kItab       Equ       [Bp+4]              ;adresse
  kAntal      Equ       [Bp+8]              ;Value
;-----------------------------------------------------------------
; prolog
              Push      Bp
              Mov       Bp,Sp
              Sub       Sp,2                ;plads til lokal-var
;-----------------------------------------------------------------
        Push      Ax
        Push      Bx
        Push      Es

        Mov       Ax,Ss:[kAntal]      ; IntTabGsnit(Antal,Itab,Sum);
        Push      Ax                  ; value Antal
        LEs       Bx,[kItab]          ; adr. på Itab
        Push      Es
        Push      Bx
        Lea       Bx,[kGsnit]         ; adr. på lokalvar. Gsnit
        Push      Ss
        Push      Bx
        Call      IntTabGsnit

        Mov       Ax,Ss:[kGsnit]      ; WriteInt(Gsnit);
        Push      Ax                  ; value Gsnit
        Call      WriteInt

        Pop       Es
        Pop       Bx
        Pop       Ax
;-----------------------------------------------------------------
; epilog
              Mov       Sp,Bp       ;         retabler stak
              Pop       Bp
              Ret       6           ;2+4      fjern activationrecord
;-----------------------------------------------------------------
WriteIntTabGsnit EndP


; Proceduren InitDs opsætter adresse på datasegmentet
InitDS  Proc    Near
        Mov     Ax,Seg DATA             ; etabler
        Mov     Ds,Ax                   ;     datasegment
        Ret                             ; retur fra procedure
InitDS  EndP

; Proceduren StopPG overgiver kontrollen til DOS igen
StopPG  Proc    Near
        Mov     Al,0                    ; returkode = 0
        Mov     Ah,4Ch                  ; opsæt funktion for
        Int     21h                     ; terminate og udfør
;  Da denne funktion ikke giver kontrollen tilbage
;  er der ingen Ret-instruktion
StopPG  EndP
```

```
; Proceduren Main er den rutine der får kontrollen ved start
Main    Proc    Near
        Call    InitDS              ; udfør InitDS

        Mov     Ax,Ds
        Mov     Es,Ax
        Lea     Di,Tekst
        Lea     Si,Tekst1
        Call    InitStr

        Lea       Bx,[Tekst]        ; WriteStr(Tekst);
        Push      Ds                ; adr. på tekst
        Push      Bx
        Call      WriteStr

        Mov     Ax,Ds
        Mov     Es,Ax
        Lea     Di,Tekst
        Lea     Si,Tekst2
        Call    InitStr

        Mov     Antal,2

        Lea       Bx,[Tekst]        ; ReadIntTab(Tekst,Antal,Itab);
        Push      Ds                ; adr. på tekst
        Push      Bx
        Mov       Ax,Antal          ; value antal
        Push      Ax
        Lea       Bx,[Itab]         ; adr. på Itab
        Push      Ds
        Push      Bx
        Call      ReadIntTab

        Mov     Ax,Ds
        Mov     Es,Ax
        Lea     Di,Tekst
        Lea     Si,Tekst3
        Call    InitStr

        Lea       Bx,[Tekst]        ; WriteStr(Tekst);
        Push      Ds                ; adr. på tekst
        Push      Bx
        Call      WriteStr

        Mov       Ax,Antal          ; WriteIntTab(Antal,Itab);
        Push      Ax                ; value antal
        Lea       Bx,[Itab]         ; adr. på Itab
        Push      Ds
        Push      Bx
        Call      WriteIntTab

        Mov     Ax,Ds
        Mov     Es,Ax
        Lea     Di,Tekst
        Lea     Si,Tekst4
        Call    InitStr

        Lea       Bx,[Tekst]        ; WriteStr(Tekst);
        Push      Ds                ; adr. på tekst
        Push      Bx
        Call      WriteStr
```

```
        Mov     Ax,Antal            ; WriteIntTabSum(Antal,Itab);
        Push    Ax                  ; value antal
        Lea     Bx,[Itab]           ; adr. på Itab
        Push    Ds
        Push    Bx
        Call    WriteIntTabSum

        Mov     Ax,Ds
        Mov     Es,Ax
        Lea     Di,Tekst
        Lea     Si,Tekst5
        Call    InitStr

        Lea     Bx,[Tekst]          ; WriteStr(Tekst);
        Push    Ds                  ; adr. på tekst
        Push    Bx
        Call    WriteStr

        Mov     Ax,Antal            ; WriteIntTabGsnit(Antal,Itab);
        Push    Ax                  ; value antal
        Lea     Di,[Itab]           ; adr. på Itab
        Push    Ds
        Push    Di
        Call    WriteIntTabGsnit

        Mov     Ah,01
        Int     21h                 ; vent på tast

        Call    StopPG              ; udfør StopPG
Main    EndP

;==========================================================
CODE    EndS                        ; slut på code-segment

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

        End     Main                ; Slut og opsæt startadr.
```