

Grundlæggende assembler-programmering

Lars K. Schunk
schunk@gmail.com

2001

Åben dokumentlicens

Dette dokument er frigivet under Åben dokumentlicens (ÅDL). Det betyder i bund og grund at du frit må videregive og/eller ændre dokumentet eller dele af det under forudsætning af at ÅDL overholdes. Læs ÅDL for detaljerne.¹

1 Indledning

Det at programmere i assembler er noget af det tætteste du kan komme på din computers CPU. Assembler-koder er næsten en direkte oversættelse af CPU'ens instruktioner som består af 0'er og 1-taller. Derfor er det muligt at lave meget små og meget hurtige programmer i assembler. Til gengæld kræver det også mere af programmøren.

Nu om dage er hastighed og størrelse ikke en særlig væsentlig fordel ved assembler. Nutidens computere er hurtige og har masser af hukommelse. Derfor fører assembler måske nok lidt af en skyggetilværelse, for man kan lave komplicerede programmer, meget hurtigere og lettere, i f.eks. C++, og har du tænkt dig at lave Windows-programmer, så vil det tage *meget* lang tid at lave noget fornuftigt i assembler som du ikke kunne lave på en enkelt dag i Visual Basic eller Delphi.

Men hvorfor så lære assembler? Svaret er simpelt: for at lære lidt mere om hvordan din computer fungerer—inde bag det hele. En Windows-haj ved måske nok meget om computere—og hans eller hendes viden kan også bruges til noget. Men ved du hvordan assembler fungerer, så ved du også en del mere om hvordan din CPU fungerer. Og du vil endda forstå en lille smule mere når du klikker på knappen `Detaljer` efter en ulovlig handling i Windows. Hvad man så end vil bruge det til ...

Inden vi går i gang vil jeg lige understrege at den type assembler som du lærer her, ikke er alt det nyeste. Det jeg beskriver her, er assembler som det fungerer på en 80386'er, men eftersom alle nye computere er bagudkompatible, så fungerer det også på f.eks. en Pentium. Når du først lige har fået fat i idéen, så er grundlæggende assembler faktisk rimelig simpelt, men skal du videre, og lære Pentium-specifikke instruktioner, så bliver det straks sværere og mere uoverskueligt, for man erstatter ikke

¹Læs ÅDL på <http://www.linuxbog.dk/licens.html>.

den gamle teknologi—man bygger ovenpå, og det medfører både fordele (bagudkompatibilitet) og ulemper (uoverskuelighed). Desuden får du nok ikke brug for de teknologiske nyskabelser når det gælder CPU'ens registre og instruktionsæt lige med det samme.

Hvis du efter at have læst denne artikel, har lyst til at læse en mere detaljeret gennemgang, så kan jeg varmt anbefale den fremragende, underholdende og meget begyndervenlige bog *Assembly Language Step-by-Step* af Jeff Duntemann². Mere eller mindre alt hvad jeg selv ved om assembler-programmering har jeg fra denne bog. Hvad kan min artikel så tilbyde? Først og fremmest kan den give et hurtigt og relativt kort overblik over emnet. Herudover afslutter jeg artiklen med et større eksempel, nemlig et fire på stribe-spil, som jeg synes er noget mere interessant end de små eksempler der findes i førnævnte bog. Men alting til sin tid. Lad os starte ved begyndelsen.

2 De nødvendige værktøjer

Inden man overhovedet kan komme i gang med at programmere i assembler, skal man have de rigtige værktøjer. Heldigvis findes der gratis programmer som du kan bruge. Det vigtigste program er assembleren NASM³. Det er det program som skal oversætte din assembler-kode (.asm-filer) til såkaldt object code (.obj-filer).

Når NASM har lavet en .obj-fil, skal programmet ALink⁴ lave en .exe-fil ud af .obj-filen. ALink er det man kalder en linker. Ved hjælp af ALink kan du sammenkæde flere .obj-filer til én .exe-fil. Du kan f.eks. lave en .asm-fil med en masse rutiner som du skal bruge i dine programmer. Den laves til en .obj-fil med NASM. Så kan du lave selve programmet i en ny .asm-fil som du også laver til en .obj-fil, og sammenkæder dem så til sidst med ALink. Bemærk at selv om du kun har én .asm-fil, skal du stadig bruge linkeren til at lave din .obj-fil om til en .exe-fil.

Et tredje program er ikke strengt nødvendigt, men det er meget smart, i hvert fald mens man er i gang med at lære assembler. Det er programmet NASM-IDE⁵. Det kan du bruge til at skrive dine assembler-programmer i. NASM-IDE fremhæver de forskellige instruktioner så du får et godt overblik over dit program. Hvis du ikke vil bruge NASM-IDE, så kan du bare bruge DOS Edit eller Windows Notepad. NASM, ALink og NASM-IDE er alle gratis at bruge.

Jeg anbefaler at du laver et bibliotek som hedder C:\Asm. Så kan du lægge NASM i C:\Asm\Nasm, og ALink i C:\Asm\ALink osv. Du bør også kalde et bibliotek C:\Asm\Projekt eller lignende som du kan bruge til at gemme dine assembler-programmer i.

Det er en god idé at lægge de to biblioteker C:\Asm\Nasm og C:\Asm\ALink ind i din sti (path). Hvis du har lagt NASM-IDE ind, er det også en god idé at lave en nasmide.bat-fil der ser således ud:

```
@echo off
```

²Læs mere om bogen på Jeff Duntemanns hjemmeside: <http://www.duntemann.com>

³<http://sourceforge.net/projects/nasm>

⁴<http://alink.sourceforge.net/download.html>

⁵http://uk.geocities.com/rob_anderton

```
cd c:\asm\projekt
c:\asm\nasmide\nasmide.exe
```

Gem filen i et bibliotek som ligger i din sti. Så kan du starte NASM-IDE hvor som helst ved at skrive `nasmide`, og så kommer du automatisk ind i `C:\Asm\Projekt` hvor du bør gemme dine assembler-filer.

Til sidst skal du sørge for at NASM-IDE kan finde NASM. Det gør du ved at starte NASM-IDE op og klikke på `Assembler` i menuen `Options`. I det vindue som kommer frem, skal du i feltet `NASM Location`, skrive sti og filnavn på NASM. Hvis du har lagt programmerne hvor jeg anbefalede dig at lægge dem, skal der stå `C:\Asm\Nasm\nasm.exe`.

3 Computerens hukommelse (RAM)

Når man skal programmere i assembler, er det vigtigt at vide noget om hvordan computerens hukommelse er organiseret og hvordan den fungerer. Først og fremmest skal du vide hvad man kalder de forskellige mængder af informationer som kan lagres i hukommelsen. Der er mange, men de vigtigste er bit, nibble, byte, word, double word, kilobyte og megabyte.

Bit En bit er det mindste stykke information som computeren kan arbejde med. En bit er enten 0 eller 1.

Nibble En nibble er en halv byte eller 4 bit og en nibble kan derfor have en værdi mellem 0 og 15. I hexadecimal mellem 0 og F. En nibble kan altså altid betegnes med et enkelt hexadecimalt ciffer.

Byte En byte er "måleenheden" i computerens hukommelse ligesom meter er det i metersystemet. En byte består af 8 bit (2 nibbles) og kan derfor have en værdi mellem 0 og 255 (00 og FF i hex). Så en byte kan altså altid beskrives med et to-cifret hexadecimalt tal.

Word Et word består af 2 byte efter hinanden. Den første byte kaldes *high byte*, og den sidste byte kaldes *low byte*. Et word kan have en værdi mellem 0 og 65535 (hex: 0000-FFFF). Det binære talsystem og det hexadecimale talsystem passer så godt sammen at man kan tage hver byte for sig, f.eks. FF og AA (dec: 255 og 170) og bare sætte dem sammen til FFAA som i decimal er 65450, og i det binære talsystem er det 111111110101010. Hvis du deler dette binære tal op i to stykker med 8 bit (en byte) i hver, bliver det til 11111111 og 10101010. Hver for sig kan de oversættes til hex, og vi kommer tilbage til FF og AA. Det er altså meget let at omregne fra hex til binær (og omvendt) og det er hovedårsagen til at hex bruges så meget i computerverdenen.

Double word Et double word (dword) består af to words efter hinanden og har en værdi mellem 0 og 4.294.967.295 (hex: 0-FFFFFFFF).

Bit, byte og word er dem du kommer til at bruge mest. Der er mange flere, men de bruges sjældent. Det næste du skal vide er hvordan man adresserer computerens hukommelse. Det gøres ved hjælp af segmenter og offsets. Når du laver almindelige DOS-programmer, har du den første megabyte hukommelse til rådighed. Det er 1.048.576 byte som så har adresserne 0 til FFFFF. Hver byte har en adresse, så forestil dig hukommelsen som en meget lang gade hvor hver byte bor i et hus som har et nummer. Det højeste nummer er her FFFFF. Bemærk at det består af 5 F'er og derfor af 20 bit. Du er altså nødt til at bruge 2 words for at få nummeret på en adresse (selv om 2 words i alt består af 32 bit). Det ene word indeholder segmentet, mens det andet indeholder offset-værdien.

Hver 16'ne byte i hukommelsen er begyndelsen på et segment, dvs. segment 0000 starter på adresse 00000. Segment 0001 starter på adresse 00010 (dec 16), segment 0002 starter i adresse 00020 (dec 32), osv. Det sidste segment FFFF starter på adresse FFFF0 (dec 1.048.560). Når du har et segment, skal du have en offset-værdi. Offset-værdien er også et word, dvs. et tal mellem 0000 og FFFF. Offset-værdien viser hvor langt fra segment-adressen, din ønskede adresse ligger. I små programmer er det meget almindeligt at man angiver segmentet én gang for alle, og så flytter rundt med offset-værdien. Det giver 64 kilobyte (0-65535; 0-FFFF) at bevæge sig rundt på hvilket sikkert er fuldt tilstrækkeligt for dine første mange assembler-programmer. Du angiver en adresse på denne måde:

segment:offset

For eksempel C800:0AC3. Her er du i segment C800, og offset-værdien er 0AC3. Hvis du vil omregne dette til den egentlige adresse kan du starte med at omregne C800 til decimal. Det er 51.200. Det ganger du med 16 og får 819.200 (husk på at der er 16 byte mellem hvert segments start-adresse). Til sidst omregner du offset-værdien 0AC3 til decimal og lægger den til. 0AC3 er 2755 i decimal og $819.200 + 2755$ er lig med 821.955 hvilket er den endelige adresse. Den kan du omregne til hex og så får du C8AC3. Som du kan se er der en vis lighed mellem segment:offset-adressen og den egentlige adresse, men eftersom den egentlige adresse bruger 5 hexadecimale cifre, er du nødt til at bruge 2 words til at beskrive adressen.

Du kan betragte det hele som om computeren er lidt nærsynet, og derfor kun kan "se" 64 kilobyte frem for sig. Derfor placerer du den på et segment. Vær i øvrigt opmærksom på at eftersom der kun er 16 byte mellem de punkter hvor segmenter starter, og at man fra hvert segment kan se 64 kilobyte frem, så er der mange andre segment:offset-kombinationer med hvilke du kan finde frem til adressen C8AC3.

Hvis du har lyst til at eksperimentere lidt med hex, binær og decimal, så kan du bruge f.eks. Windows' lommeregner til at regne med disse talsystemer.

4 CPU'ens registre

Når du programmerer i assembler, har du CPU'ens interne registre til rådighed hvori du kan lægge resultater, mellemresultater og mange andre ting. Du har først og fremmest nogle segmentregistre. Disse registre er 16 bit store og kan (og bør kun) bruges til at gemme segment-adresser. De er som følger:

CS *Code Segment*. Dit program består af nogle instruktioner som befinder sig et sted i computerens hukommelse. CS-registret indeholder adressen for det segment hvori dit program ligger.

DS *Data Segment*. Dine data (variable o.lign.) ligger også et sted i hukommelsen. DS-registret indeholder adressen for det segment hvori dine data ligger.

SS *Stack Segment*. Stakken er et område i hukommelsen hvor computeren og du kan gemme midlertidige data. Jeg vender tilbage til stakken i afsnit 7. Nu skal du bare vide at stakken har en segment-adresse som skal ligge i SS.

ES *Extra Segment*. Dette er et ekstra segment som ikke skal bruges til noget specielt. Det kan du bare bruge hvis du får behov for at pege på et eller andet segment. Der er to registre mere af denne type som hedder **FS** og **GS**. Deres navne står ikke for noget specielt.

Husk på at hver gang du ser et register der ender på S, så er det et segmentregister.

De næste fire registre er registrene **AX**, **BX**, **CX** og **DX**. Disse registre er også 16 bit store, og de bruges til ting som mellemregninger og resultater. Du kan få adgang til hver af disse registres "halvdel" ved at bruge et H eller et L i stedet for X'et. H'et står for *high byte* og L'et står for *Low byte*. AX består altså af AH og AL, BX består af BH og BL, CX af CH og CL, og DX af DH og DL.

Hvis du lægger værdien A43C ind i register AX, så vil AH indeholde værdien A4, og AL vil indeholde værdien 3C. Hvis du så ændrer værdien af AL til f.eks. B3, så har AX nu værdien A4B3. En smart ting ved dette er at du kan ændre en værdi i AL uden at røre ved værdien i AH. Disse fire registre er de eneste registre som har den egenskab at de kan deles op i 8-bit-halvdele, og ud over denne egenskab har hvert af disse fire registre en speciel egenskab hvoraf BX's kommer i næste afsnit. Grunden til at man ikke bare har givet alle fire registre alle de specielle egenskaber er at CPU'en har nogle fysiske begrænsninger. Det er altså begrænset hvor mange transistorer der er i CPU'en, og derfor har man måttet gå på kompromis mange steder. Du vil støde på flere sådanne begrænsninger senere hen, for som nævnt i indledningen, så er det at programmere i assembler mere eller mindre så tæt på CPU'en som du kommer.

Der er også to indeksregistre **SI** og **DI** i CPU'en (du kan også kalde dem for offset-registre). Disse bruges til at gemme offset-værdier, og de skal bruges sammen med de førnævnte segmentregistre når du skal pege på et sted i hukommelsen. SI står for *Source Index* (kilde-indeks), og DI står for *Destination Index*. Hvis du får brug for et ekstra indeksregister, kan du også bruge BX som indeksregister (det er BX's specielle egenskab). Hvis du f.eks. vil pege på adressen C8AC3 som på segment:offset-formen kan skrives som C800:0AC3, så gemmer du C800 (segment-adressen) i et af segmentregistrene (som regel DS eller ES) og 0AC3 (offset-værdien) gemmer du i SI, DI eller BX. Hvis du har gemt segment og offset i DS og SI, kan du beskrive adressen ved at sige DS:SI. Hvis intet andet er angivet, så går man ud fra at du mener DS (Data Segment-registret). Dette vil give mere mening når du kommer i gang.

Der er tre registre mere som du nok ikke får så meget brug for. Det er **BP**, **SP** og **IP** som også indeholder offset-værdier. BP og SP står for henholdsvis *Base Pointer* og *Stack Pointer*. Disse bruges sammen med det midlertidige lager *stakken* (som jeg

forklarer mere om i afsnit 7). Hvis intet andet er angivet, går man ud fra at det tilhørende segment-register er SS (Stack Pointer-registeret). SP peger på det næste frie sted på stakken. *Du gør klogt i ikke at røre SP*. BP kan du bruge til at pege på andre steder i stakken. IP står for *Instruction Pointer* og bruges sammen med CS (Code Segment-registeret) til at pege på den næste instruktion som skal udføres. CPU'en sørger selv for at ændre IP efterhånden som den udfører dine instruktioner. IP kan du *slet ikke ændre direkte*—og det er måske også kun godt nok ...

Flag-registeret er det sidste register. Det er 16 bit stort og bliver kaldt **FLAGS**. Flag-registerets bit er de forskellige flag. Der er 9 flag som hver har en forkortelse på to bogstaver. Flagene er som følger: Overflow Flag (OF), Direction Flag (DF), Interrupt Enable Flag (IF), Trap Flag (TF), Sign Flag (SF), Zero Flag (ZF), Auxiliary Flag (AF), Parity Flag (PF) og Carry Flag (CF). Det er heldigvis ikke nødvendigt at huske alle flagene da du, som programmør, ikke har brug for særlig mange. De to flag som du vil bruge mest er Zero Flag (ZF) og Carry Flag (CF). Ét flag er repræsenteret af én bit i Flag-registeret, og et flag kan således være enten 1 (flaget er hejst) eller 0 (flaget er ikke hejst).

Flagene bruges til at vise nogle forskellige tilstande i CPU'en. Dine programmer kan så teste flagene og reagere alt efter om et flag er *set* (hejst) eller *cleared* (ikke hejst). Zero Flag bliver som regel sat når en instruktion medfører at et eller andet giver nul. Hvis en operation medfører et nul, så bliver ZF's bit sat lig med 1, og hvis en operation ikke bliver nul, så bliver ZF's bit sat lig med 0. Det virker måske lidt underligt, men husk at 1 betyder "vift med flaget". Og hvis en operation bliver nul, så skal ZF vifte med sit flag (ZF=1). Det andet flag som du kan få brug for, er Carry Flag (CF). Carry betyder mente, og hvis en operation medfører at en bit kommer i mente, så bliver CF sat lig med 1. Flagene får du først brug for når du skal lave noget lidt mere avanceret.

5 Det første program

Nu skal vi til at programmere assembler! Du kan jo passende starte med at taste følgende ind i NASM-IDE eller Notepad eller hvad du nu bruger (du behøver ikke at taste de linjer ind som starter med et semikolon):

```
;-----  
;           Start på vores fil  
;-----  
  
[BITS 16]  
  
;-----  
;           Vores programkode  
;-----  
  
SEGMENT kode  
  
..start:
```

```

mov     ax,data
mov     ds,ax
mov     ax,minStak
mov     ss,ax
mov     sp,stacktop

mov     dx,besked
mov     ah,9
int     21h

mov     ax,4C00h
int     21h

;-----
;           Vores data
;-----

SEGMENT data

        besked     db     "Assembler-programmering!", 13, 10, "$"

;-----
;           Vores stak
;-----

SEGMENT minStak stack

        resb 64

stacktop:

;-----
;           Slut på vores fil
;-----

```

Programmet er skrevet i det der hedder Real Mode Segmented Model. Det er faktisk en forældet måde at skrive programmer på, men når jeg har valgt at beskrive denne model, er det fordi det en god idé at lære noget om CPU'en og computerens hukommelse. Senere kan du skrive programmer i Real Mode Flat Model som forenkler en hel del ting hvad angår segmenter og offsets.

Alle linjer der starter med et semikolon, er kommentarer til os selv, og ignoreres af assembleren.

Så den første egentlige linje [BITS 16] fortæller assembleren at den skal generere 16-bit-kode. Husk på at den fil du lige har skrevet er kildekoden. Kildekoden skal oversættes til maskinkode. Hvis du vil køre programmet, skal du først gemme filen som

eks1.asm. Derefter oversætter du den med NASM:

```
C:\ASM\PROJEKT>nasm eks1.asm -f obj -o eks1.obj
```

Så har du en fil som hedder eks1.obj. Den laver du til en .exe-fil med ALink:

```
C:\ASM\PROJEKT>alink eks1.obj
```

Og så kører du blot den nye fil som hedder eks1.exe. Bemærk at .exe-filen ikke fylder mere end 139 byte!

Nå, videre med gennemgangen af vores kildekode. Resten af koden er delt op i tre dele: Code Segment, Data Segment og Stack Segment. Code-delen indeholder selve programmet. Data-delen indeholder de data vi skal bruge i vores program, og Stack-delen har med stakken at gøre.

Vi markerer vores Code Segment med direktivet `SEGMENT kode`. Herefter markerer vi hvor vores programkode starter med en *etiket* (eng. *label*). Den skal hedde `..start:` (det skal skrives med små bogstaver, med to punktummer foran, og et kolon bagefter!) Det virker måske lidt dumt at man skal vise hvor programmet starter, men et program behøver ikke nødvendigvis at starte fra begyndelsen af filen. Så kommer programmets første fem instruktioner:

```
mov     ax,data
mov     ds,ax
mov     ax,minStak
mov     ss,ax
mov     sp,stacktop
```

De bruges til at initialisere vores segmentregistre. Dette skal faktisk gøres i ethvert program. Som du kan se, bruges **mov**-instruktionen meget. Det er også den mest brugte instruktion i assembler. Det eneste **mov** gør, er at flytte rundt med data. Du vil opdage at du vil komme til at bruge meget tid på bare at flytte rundt med data. Hvis du ser på vores program, vil du se at det næsten ikke bruger andet end **mov**-instruktioner!

Det første vi skal have gjort er at flytte vores datasegment-adresse ind i CPU-registret DS så vi kan bruge vores data. Her støder vi på CPU'ens første begrænsning: Man kan *ikke* lægge en værdi direkte ind i DS. Derfor er vi nødt til først at lægge vores adresse ind i AX, og så derefter flytte indholdet af AX ind i DS. Det er det som de to første **mov**-instruktioner gør. Instruktionen `mov ax,data` betyder: flyt datasegment-adressen ind i register AX. Nu er vi så heldige at vi ikke behøver at kende den præcise adresse på vores datasegment. Det har vi fået assembleren til at holde styr på ved at kalde vores datasegment for data med direktivet `SEGMENT data` længere nede i kildekodefilen. Hvis vi i stedet havde kaldt vores datasegment for f.eks. `papir`, så skulle vi have brugt instruktionen `mov ax,papir` i stedet.

Så nu indeholder register AX altså vores datasegment-adresse, men den skal jo ligge i register DS (husk: **D**ata **S**egment). Det gør vi med instruktionen `mov ds,ax` som betyder: flyt indholdet af AX ind i DS. Måske synes du at det hele virker omvendt, og at instruktionen burde have været `mov ax,ds`. Det virker meget mere naturligt, men sådan er det ikke! Overalt i Intel assembler kommer *destinationen først og kilden*

kommer bagefter. I øvrigt er navnet **mov** lidt vildledende, for når du flytter noget fra AX til DS, så bliver værdien ikke slettet fra AX, så det du egentlig foretager dig, er at du kopierer værdien fra register AX til register DS.

De næste to instruktioner gør det samme, bare med staksegmentet (Stack Segment). Den sidste instruktion `mov sp, stacktop` sætter SP (Stack Pointer) til at pege på den første plads i stakken. Du skal ikke vide så meget om stakken endnu, for vi skal ikke selv bruge den, men den skal være med i programmet alligevel.

De næste tre instruktioner er vores egentlige program:

```
mov     dx, besked
mov     ah, 9
int     21h
```

Som du kan se, bruger vi igen nogle **mov**-instruktioner, og til sidst en **int**-instruktion. Nede i vores datadel har vi indsat en sætning, en streng af tegn. Den har vi kaldt `besked`, og vi er nu igen så heldige at assembleren tager sig af at holde styr på *offset-adressen* for os. `besked` er nemlig lig med offset-adressen på starten af vores tekststreng (som vi har defineret længere nede i kildekode-filen).

int-instruktionen bruges til at kalde et såkaldt interrupt. I dette tilfælde er det interrupt 21h. Bemærk h'et efter 21. Det fortæller assembleren at det er en hexadecimal værdi du her bruger, og du skal altid bruge et h til at markere at et tal er i hex. I appendikset kan du se at interrupt 21h er de såkaldte DOS-funktioner. Det er altså et bibliotek af rutiner som du kan bruge i dine programmer. Nummeret på den ønskede funktion skal du gemme i register AH *inden* du kalder interrupt 21h. Det er det vi gør med `mov ah, 9`. Vi flytter 9 ind i register AH (AH er den øvre halvdel af register AX). I oversigten kan du se at funktion 9 hedder Print String, dvs. at den kan udskrive en streng på skærmen. Funktion 9 skal på en eller anden måde vide hvilken streng den skal udskrive for dig, så det skal også angives før brug af `int 21h`. Du skal gemme strengens placering i hukommelsen i DS:DX, dvs. at segmentet skal ligge i DS, og offset-værdien skal ligge i DX. Segmentet har vi allerede gemt. Det var faktisk det første vi gjorde i programmet (de to første instruktioner). Eftersom "variablen" `besked` er lig med offset-værdien til starten af vores streng, bruger vi instruktionen `mov dx, besked` til at flytte offset-værdien ind i DX. Så sørger assembleren for at indsætte den rigtige offset-værdi på `besked`'s plads. Så tager interrupt 21h over og udskriver vores streng på skærmen tegn for tegn indtil den støder på tegnet \$. Hvis du vil have den til at skrive et dollartegn, kan du altså ikke bruge interrupt 21h funktion 9!

De to sidste instruktioner er:

```
mov     ax, 4C00h
int     21h
```

Her bruger vi også interrupt 21h. I modsætning til før, så bruger vi nu funktion 4Ch i stedet for funktion 9. Funktion 4Ch kaldes også for Terminate Process hvilket betyder at vores program slutter. 4Ch skal gemmes i AH (ligesom 9 skulle gemmes i AH for at vælge funktion 9), så hvorfor gemmer vi 4C00h i AX? Det er faktisk bare en hurtigere måde at gemme to værdier på. Vi kunne i stedet have brugt to **mov**-instruktioner: Først gemmer vi 4Ch i AH, dernæst gemmer vi 00 i AL. I AL skal vi nemlig lægge en

errorlevel-værdi når vi kalder funktion 4Ch. Det er ikke strengt nødvendigt at gemme noget i AL, men det er god skik at gemme et 00 hvis errorlevels alligevel ikke bruges. Errorlevels kan bruges hvis du har et stort program med mange forskellige måder at afslutte på. Så kan du gemme en forskellig errorlevel-værdi for hvert sted hvor programmet bliver afsluttet, så du kan holde styr på hvor dit program bliver afsluttet vha. en .bat-fil. Dette bruges især til at finde fejl med.

Den næste del er vores datadel:

```
SEGMENT data
```

```
        besked      db      "Assembler-programmering!", 13, 10, "$"
```

Her definerer vi vores data. I dette program har vi ikke så meget data. Vi indsætter i dette tilfælde en masse tegn efter hinanden, omringet af anførselstegn, herefter indsætter vi 13 og 10, og til sidst et dollartegn som markerer afslutningen på strengen. 13 og 10 markerer en vognretur og et linjeskift. Du kan se disse koder i en ASCII-tabel⁶. Grunden til at man skal bruge to koder for at skifte linje er at man af tekniske årsager gjorde det på den måde i den fjerne fortid. Foran står der `besked` som betyder at vi sætter `besked` til at være lig med den første byte i vores streng (her er det et A). Så kan vi bruge ordet `besked` i stedet for den rigtige offset-værdi. Midt imellem de to ting står der `db`. Det står for **D**efine **B**yte. Man kan også definere words og double words med `dw` og `dd`. Det har vi dog ikke brug for her. Ét tegn fylder netop én byte. I stedet for at skrive et dollartegn i anførselstegn kunne du også bare have skrevet ASCII-værdien for det. Så ville sidste del af linjen have set sådan ud: `..., 13, 10, 36` da et dollartegn har ASCII-værdien 36.

Til sidst er der stakken:

```
SEGMENT minStak stack
```

```
        resb 64
```

```
stacktop:
```

Her kalder vi stakken for `minStak` og den er af typen `stack`. Vi reserverer 64 byte til stakken med direktivet `resb 64` (**R**eserve **B**ytes). Til sidst har vi en etiket der peger på toppen af stakken, så derfor kalder vi den for **stacktop**: Denne etiket bruger vi i instruktion nr. 5 i vores program når vi skal sætte SP (**S**tack **P**ointeren) til at pege på starten af stakken, dvs. toppen af stakken. Husk på at selv om `stacktop:` er i bunden af kildekodefilen, så arbejder assembleren nedefra og op i hukommelsen, mens den læser kildekodefilen oppefra og ned, så etiketten `stacktop:` er altså det sidste som assembleren bearbejder, og `stacktop:` bliver så lig med den offset-værdi der ligger højest i hukommelsen (inden for vores program)!

⁶Se f.eks. <http://www.asciitable.com>

6 DEBUG-programmet

Når du har fået lavet din `eks1.exe`-fil, kan du kigge nærmere på den med programmet Debug. Du starter Debug ved at skrive:

```
C:\ASM\PROJEKT>debug eks1.exe
```

Debug svarer med et - (minustegn). Prøv at skrive et d (**d**ump) efterfulgt af Enter. Her ser du den rå kode for vores program. Til venstre kan du se hvor i hukommelsen programmet ligger, angivet med segment:offset-notationen. I midten er vores kode i hexadecimal tal. Dette kunne lige så godt have stået i binære tal, men det ville fylde en hel del mere og ville ikke give et særlig godt overblik. Hvert tocifret hex-tal repræsenterer en byte (8 bit) i hukommelsen. Til højre på skærmen står vores kode oversat til ASCII-tekst. Hvert tegn har en ASCII-værdi; det er dog ikke alle ASCII-tegn som kan skrives med et symbol (f.eks. Enter) og disse tegn skrives som et . (punktum). Til højre kan du genkende vores tekststreng som vi skrev. Bemærk de to punktummer mellem ordet *Assembler-programmering!* og dollar-tegnet. Ovre i hex-tallene kan du se at de er tallene 0D og 0A (dec 13 og 10) som jo var vores linjeskift og som ikke kan skrives med symboler. Derfor er de skrevet som to punktummer.

Prøv nu at skrive et u (**u**nassemble) efterfulgt af Enter. Her har Debug oversat hex-tallene tilbage til assembler. Du kan f.eks. se i adresse 0000:0010 at B4 09 er instruktionen `mov ah, 09`. (Grunden til de fire spørgsmålstegn er at segment-adressen ikke vil være den samme på din computer som på min. Det kommer an på hvor i hukommelsen der var plads til programmet da det blev indlæst i Debug.) Du kan også se at vores etiketter (`data`, `minStak`, `besked` osv.) er blevet erstattet af de rigtige segment-adresser og offset-værdier. Efter instruktionen `CD 21` (`int 21`) kommer der nogle instruktioner som vi ikke har noget med at gøre. De kan være hvad som helst, og de er højst sandsynligt ikke engang "rigtige" instruktioner. Der ligger bare nogle tal i hukommelsen efter vores program er slut som Debug oversætter til instruktioner.

Hvis du taster t (trace), kører du programmet én instruktion ad gangen hvorefter du ser en oversigt over registrenes indhold. Efter den første instruktion er udført, kan du se at AX er blevet lig med adressen på vores datasegment. På tredje linje af oversigten kan du se hvilken instruktion som CS:IP (Code Segment:Instruction Pointer) peger på. Det er den næste instruktion som vil blive udført når du taster t. Den næste instruktion er her `mov ds, ax`. Tast t, og i den nye oversigt kan du se at DS er blevet lig med AX, og IP er blevet ændret til at pege på den næste instruktion. Hvis du fortsætter sådan et stykke tid, kommer du til instruktionen `int 21` (Debug arbejder kun i hex, og bruger derfor ikke noget h til at markere at tallene er i hex). Hvis du trace'r denne instruktion, vil du opdage at du kommer et helt andet sted hen i hukommelsen. Det sker netop fordi du med `int 21` kalder en af de mange DOS-funktioner, og disse DOS-funktioner ligger jo også et sted i hukommelsen og består af maskinkode-instruktioner; Debug følger bare efter. Tryk q (quit), for at afslutte Debug da DOS-funktionerne er ret lange, og du har sikkert ikke lyst til at trace igennem det hele.

7 Stakken

Hvordan finder computeren egentlig tilbage til vores program når den er færdig med at udføre en DOS-funktion? CS:IP kan jo kun pege på én instruktion ad gangen. Det er her stakken kommer i brug. Stakken er et midlertidigt lager hvor du kan gemme data og hente dem igen. Når computeren udfører en **int**-instruktion, så bliver adressen på den næste instruktion i programmet gemt på stakken. Når DOS-funktionen så bliver afsluttet med en **iret**-instruktion (**I**nterrupt **R**eturn), så henter den adressen fra stakken igen. Stakken fungerer efter LIFO-princippet (**L**ast **I**n, **F**irst **O**ut) hvilket betyder at det sidste der bliver lagt på stakken, også er det første der bliver hentet igen. Du kan naturligvis også bruge stakken manuelt. Her skal vi bruge instruktionerne **push** og **pop**.

push	ax	push	ax
push	bx	push	bx
... andre koder andre koder ...	
pop	bx	pop	ax
pop	ax	pop	bx

Rigtigt! Forkert!!

Lad os se to eksempler på brug af **push** og **pop**. Vi forestiller os at du skal bruge registrene AX og BX til noget, men du vil ikke miste de værdier som de allerede har! Derfor skubber du dem ind på stakken med **push**. Efter du udført en masse andre ting, giver du AX og BX deres gamle værdier tilbage med **pop**-instruktionen. Til venstre ovenfor er det gjort på den rigtige måde. BX kom sidst ind, så derfor kommer BX først ud. Til højre er det gjort forkert. Her vil AX og BX have byttet værdier efter **pop**'erne. Det kan selvfølgelig være at det var meningen, men så kunne du have brugt instruktionen `xchg ax, bx` i stedet. Ligesom CS:IP peger på den næste instruktion der skal udføres, så peger SS:SP på den næste plads på stakken hvor der er plads til noget data. Når du bruger **push** og **pop**, så bliver SP ændret så den peger på det rigtige sted. Husk at der altid kun er én stak i brug ad gangen. Det vil sige at din stak bliver brugt af hele computeren. I vores program reserverede vi kun 64 byte til stakken, men det er sikrere at reservere f.eks. 512 byte til den. Hvis der opstår et såkaldt Stack Overflow (stakken "flyder over" sine grænser), så begynder der at ske underlige ting og sager.

8 Real Mode Flat Model

I afsnit 5 skrev vi vores program i Real Mode Segmented Model. I dette afsnit vil vi lave nøjagtig samme program, men denne gang i det der hedder Real Mode Flat Model. Denne model er mere enkel, men begge modeller er lidt forældede. Den nye model hedder Protected Mode Flat Model der bruges til 32-bit-programmering, men den model vil jeg ikke komme ind på her.

Da vi brugte Real Mode Segmented Model havde vi hele den første megabyte af computerens hukommelse til rådighed. Vores program var dog et meget lille program, og det vil de fleste af dine fremtidige programmer nok også være hvis du skriver dem i

assembler. Så kan vi jo nøjes med en .com-fil som fylder under 64 kilobyte. 64 kilobyte lyder ikke af meget, men det er en pæn størrelse når vi snakker assembler!

Som du husker(?) brugte vi i Real Mode Segmented Model segmenter og offsets til at udpege områder i hukommelsen. Vi placerede et segment i et segmentregister, og så placerede vi en offset-værdi i et indeksregister. Offset-værdien kunne have en værdi mellem 0 og 65.535 (0-FFFF) og vi kunne således "se" 64 kilobyte frem for os fra det aktuelle segment.

I Real Mode Flat Model behøver vi ikke bekymre os om segmenter eftersom hele vores program, vores data og stakken ligger inden for 64 kilobyte. Computeren sætter derfor automatisk alle segmentregistre for dig. Lad os se på vores program i Real Mode Flat Model:

```
[BITS 16]
[ORG 100h]

[SECTION .text]

    mov     dx, besked
    mov     ah, 9
    int     21h

    mov     ax, 4c00h
    int     21h

[SECTION .data]

    besked  db      "Assembler-programmering!", 13, 10, "$"

;-----
;           Slut på vores fil
;-----
```

Som du kan se er programmet blevet væsentlig mindre. Du kan compilere programmet med NASM på følgende måde hvis du har gemt filen som eks2.asm:

```
C:\>nasm eks2.asm -f bin -o eks2.com
```

Bemærk at når du laver en simpel .com-fil behøver du ikke bruge linkerens (ALink) bagefter. NASM ordner selv tingene for dig ved brug af -f bin. Programmet fylder nu kun 39 byte!

Men lad os nu se på de nye ting. Vi starter endnu engang med [BITS 16] for at fortælle NASM at vi vil generere 16-bit-kode. Altså et almindeligt DOS-program. Herefter har vi direktivet [ORG 100h]. Det betyder at vi springer de første 256 byte (hex 100h) over, og computeren skal udføre vores instruktioner herfra. Grunden til at

dette er nødvendigt er at .com-programmer bruger de første 256 byte til forskellige ting og sager som vi ikke har lyst til at pille ved—derfor springer vi dem over, og markerer med [ORG 100h] at vores program starter 256 byte længere fremme.

Næste linje er [SECTION .text]. I Real Mode Flat Model arbejder vi ikke med segmenter, men med sektioner inden for det enkelte segment som vi har til rådighed. Den førnævnte linje fortæller hvor vores programkode befinder sig. Bemærk her at vi nu ikke skal initialisere segmentregistre som vi gjorde i Real Mode Segmented Model (de første fem instruktioner). På den måde bliver programmet endnu simplere. Vi har heller ikke brug for en start-etiket da vi skal kalde vores kodesektion for .text.

Vores data-sektion markerer vi med [SECTION .data], og så kan vi initialisere vores data på samme måde som vi gjorde i Real Mode Segmented Model.

Vi har ikke brug for at reservere plads til stakken. Den ligger nemlig i toppen af de 64 kilobyte (vores segment), og hver gang vi **push**'er noget på stakken “bevæger” stakken sig tættere på vores programkode. Det vil sige at hvis vores program er meget stort, og vi lagrer en masse data på stakken, så kan de to ting kollidere, og resultatet af dét er sikkert ikke særlig pænt ...

9 Adressering af data

Der er tre måder at adressere data på: *Immediate*, *register* og *memory*. Immediate er den nemmeste at forstå, og vi har også allerede brugt den. Her er et eksempel:

```
mov     ah, 9
```

Her flytter vi tallet 9 ind i register AH. Der er ingen tvivl om at tallet er 9. Det vil altid være tallet 9 der bliver lagt ind i AH når denne instruktion bliver udført. Immediate data er altså en fast værdi som er “indbygget” i instruktionen. Her er nogle andre eksempler:

```
int     21h
mov     ax, 4c00h
mov     dx, besked
```

Den sidste kræver måske lidt forklaring. Her skal du huske at vi i data-sektionen satte besked til at være lig med offset-værdien til starten af det område i hukommelsen som indeholder vores streng. Vi sætter altså kort sagt DX til at være lig med adressen på starten af strengen.

Her er et eksempel på brug af register-data:

```
mov     ax, bx
```

Her sætter vi AX til at være lig med BX. Vi kopierer altså indholdet af BX over i AX. Bemærk her at man ikke kan kopiere indholdet af et 2-byte-register over i et 1-byte-register og omvendt som f.eks AH over i BX. Her er nogle flere eksempler:

```
mov     ax, di
mov     si, cx
```

Til sidst er der memory-data. Det er en lille smule mere kompliceret. Her er et eksempel:

```
mov     ax, [bx]
```

Her fortæller de kantede parenteser at vi snakker om memory-data. Det antages så at register BX indeholder en *adresse*. Den værdi som ligger på denne adresse i hukommelsen, bliver kopieret over i register AX. BX indeholder her offset-værdien. Segmentet antages i de fleste tilfælde at ligge i register DS (**D**ata **S**egment), men det skal du jo ikke bekymre dig om når du arbejder i Real Mode Flat Model. Her er et eksempel der måske illustrerer tingene lidt bedre:

```
mov     ax, [besked]
```

besked repræsenterer en offset-adresse som peger på vores streng. Ved at sætte kantede parenteser omkring, fortæller vi at det er indholdet af den adresse *besked* peger på som vi vil have fat i. Hvis vi holder os til vores program fra før, vil AX nu indeholde ASCII-værdien for tegnet A som jo er første tegn i vores streng.

Vi havde ikke brug for memory-metoden i vores program. Vi skulle bruge *offset*-værdien til starten af vores streng inden vi kaldte interrupt 21h, funktion 9, som forventer at DX indeholder offset-værdien til starten af den streng der skal udskrives.

Der er en smart ting ved memory-data. Betragt følgende eksempel:

```
mov     [bx+di+2], al
```

Her flytter vi indholdet af register AL ind på den adresse, som BX+DI+2 peger på. Det vil sige at computeren først tager indholdet af BX. Derefter lægger den indholdet af DI til, og til sidst lægger den 2 til resultatet. Det er denne adresse hvis indhold bliver sat lig med indholdet af AL. På den måde kan vi ændre register DI, og instruktionen vil pege på en helt anden adresse. Der er nogle krav til brugen af denne metode. Det første register *skal* være enten BX eller BP. Det andet register *skal* være DI eller SI, og den sidste værdi *skal* være en konstant, dvs. et tal som er fast defineret i koden. Du kan dog godt udelade det sidste tal, men du kan ikke tilføje flere registre som f.eks. [BX+DI+SI+3]. Måske har du svært ved lige at se det smarte i det her, men i det store, afsluttende eksempel i afsnit 15 kommer vi til at bruge det i stor stil.

10 Type Specifiers

Hvis du vil være på den sikre side (og nogle gange kræver assembleren at du er), så kan du bruge såkaldte *type specifiers* til at angive om en offset-værdi peger på en byte eller et word eller noget helt tredje. Nogle gange kan man selv regne ud, hvilken type der er tale om, men det er ikke altid assembleren kan. Det kan du specificere sådan her:

```
mov     al, BYTE [BX]
```

BX indeholder en offset-værdi. Men denne offset-værdi peger jo både på en byte og et word osv. Ved at angive typen som det er gjort ovenfor, kan der ikke være tvivl om at det er en byte vi er ude efter. Det kan være svært at indse hvornår assembleren kræver en type specificer. Men du kan bare starte med at lade være med at bruge dem. Hvis assembleren så, under kompileringen, skriver "operation size not specified" og et linjenummer, så ved du hvor den kræver én, og så er det jo blot at indsætte den.

11 Procedurer

Når du laver større programmer, kan det være smart at opdele dit program i flere blokke. Så kan du lave et hovedprogram som kalder de forskellige blokke. Lad os se på en version af vores sædvanlige program:

```
[BITS 16]
[ORG 100h]

[SECTION .text]

    call  write
    jmp   end

write:
    mov  dx, besked
    mov  ah, 9
    int  21h
    ret

end:
    mov  ax, 4c00h
    int  21h

[SECTION .data]

    besked  db      "Assembler-programmering!", 13, 10, "$"

;-----
;           Slut på vores fil
;-----
```

Vi har nu delt vores program op i to underrutiner som vi har markeret med etiketter, her `write:` og `end:`. Vores "egentlige" program består nu kun af en **call**-instruktion og en **jmp**-instruktion. Første instruktion er `call write`, så computeren springer til

etiketten `w r i t e :` og fortsætter derfra indtil den møder en **ret**-instruktion (**return**) der fortæller computeren at den skal gå tilbage til det oprindelige program. Derefter kalder den end-proceduren som afslutter programmet. end-proceduren kaldes med **jmp**-instruktionen. Computeren glemmer så alt om hvor den var og følger blindt efter. Det gør dog ikke noget, for denne procedure skal jo afslutte programmet, og vi har ikke brug for at vende tilbage til hovedprogrammet. end-proceduren har derfor heller ikke nogen **ret**-instruktion. Husk: **call** og **ret** hører sammen!

Der er ikke de store fordele ved at dele et så lille program op i procedurer, men det hjælper meget på store programmer som vi også skal se i afsnit 15.

12 Instruktionerne **inc** og **dec**

Du kan bruge instruktionen **inc** til at forhøje en værdi med 1, og instruktionen **dec** til at formindske med 1. Eksempel:

```
mov     ax, 5
inc     ax
dec     ax
```

Her lægger vi først 5 ind i AX. Så bruger vi **inc** på AX. Nu vil AX have værdien 6. Herefter bruger vi **dec** på AX, og AX har værdien 5 igen. Hvis AX havde været 255 (hex FF) og man brugte **inc**, så ville AX få værdien 0. Omvendt hvis du bruger **dec** og AX har værdien 0, så vil den få værdien 255.

13 Instruktionerne **add** og **sub**

Instruktionerne **add** og **sub** kan du bruge hvis du vil lægge en værdi til eller trække en værdi fra en anden værdi. Eksempel på **add**:

```
mov     ax, 5
mov     cx, 4
add     ax, cx
```

Her lægger vi 4 til værdien 5, og resultatet 9 bliver lagt i AX som dermed erstatter den ene operand, her 5. **sub** fungerer på næsten samme måde:

```
mov     ax, 10
sub     ax, 3
```

AX får værdien 10 hvorefter **sub** trækker 3 fra værdien i AX, og resultatet bliver lagt i AX som dermed mister den første værdi, her 10. Hvis resultatet er negativt, så bliver Carry Flag (CF) sat lig med 1. Du kan så teste Carry Flag, og lade programmet træffe en beslutning. Mere om disse ting i næste afsnit.

14 Lad computeren træffe beslutninger

Overskriften til dette afsnit er måske ukorrekt. Det du skal lære nu, er hvordan du kan teste flagenes tilstand, og så gøre noget forskelligt alt efter resultatet af en sådan test.

Vi starter med at se på instruktionen **cmp** (**compare** = sammenlign):

```
cmp    cx, 5
```

cmp-instruktionen er faktisk en **sub**-instruktion. Forskellen er bare at resultatet af subtraktionen ikke gemmes. Men hvis resultatet bliver nul (dvs. de to værdier var ens), så sættes Zero Flag (ZF) til 1. Hvis ikke, så sættes ZF til 0. Vi kan altså bruge ovenstående eksempel til at teste om CX har værdien 5. Hvis ja, så bliver ZF lig med 1; hvis nej, så bliver ZF lig med 0. Vi kan teste med **je**-instruktionen (**jump if equal**):

```
cmp    cx, 5
je     videre
```

Hvis CX er lig med 5, så sender **je**-instruktionen os til det sted i programmet som er markeret med etiketten *videre*:. Hvis CX er forskellig fra 5, så fortsætter programmet med at udføre instruktionen efter **je**. Vi kunne også have brugt den “omvendte” funktion **jne** (**jump if not equal**). Så ville computeren springe til *videre*: hvis CX var forskellig fra 5.

Der er mange **j??**-instruktioner. Her en liste over de mest anvendelige:

ja	jump if above
jae	jump if above or equal
jb	jump if below
jbe	jump if below or equal
je/jz	jump if equal/Jump if zero
jne/jnz	jump if not equal/Jump if not zero
jl	jump if less
jle	jump if less or equal
jc	jump if Carry Flag set
jnc	jump if not Carry Flag set

Som du kan se, så er den eneste beslutning en computer kan træffe, en beslutning om enten at hoppe—eller at lade være med at hoppe. Men et hop kan selvfølgelig have stor betydning. Computeren “hopper” ved at ændre register IP (**I**nstruction **P**ointer) som jo indeholder adressen på den næste instruktion som skal udføres. Så IP sættes simpelthen bare lig med den adresse som står efter **j??**-instruktionen. Denne adresse er så typisk repræsenteret vha. af en etiket.

Den førnævnte **call**-instruktion fungerer på samme måde som **jmp**-instruktionen. Men umiddelbart inden IP ændres, bliver den gamle værdi (som indeholder adressen på instruktionen lige efter **call**-instruktionen) gemt på stakken. **ret**-instruktionen gør så ikke andet end at hente værdien tilbage fra stakken igen og lægge den ind i register IP!

Sådan nogle detaljer behøver du naturligvis slet ikke at vide noget om, men det er alligevel kendskabet til sådanne ting som kan gøre assembler-programmering sjovt.

15 Det afsluttende eksempel

I dette afsnit skal vi se på et afsluttende eksempel: et fire på stribe-spil. Det ser måske lidt overvældende ud—og det er det måske egentlig også. Men her kommer programkoden, og så vil jeg forklare det bagefter. Det vil være en god idé at have appendikset ved hånden under læsning og gennemgang af programmet.

```
; *** Fire på stribe ***

[BITS 16]

;-----
;-----Code Section-----
;-----

SEGMENT code

; *** Main Program ***

..start:
    mov  ax,data
    mov  ds,ax           ; Initialisér data-segmentet.
    mov  ax,minStak
    mov  ss,ax          ; Initialisér stakken.
    mov  sp,stacktop

    call clrBoard       ; Tømmer pladen for brikker.

turn:
    mov  BYTE [spiller], '1' ; Spiller 1's tur.
    call drawScreen     ; Tegner pladen.
    call getMove        ; Henter spiller 1's træk.
    call makeMove       ; Udfører spiller 1's træk.
    call checkStatus    ; Er spillet vundet eller uafgjort?

    mov  BYTE [spiller], '2' ; Spiller 2's tur.
    call drawScreen     ; Tegner pladen.
    call getMove        ; Henter spiller 2's træk.
    call makeMove       ; Udfører spiller 2's træk.
    call checkStatus    ; Er spillet vundet eller uafgjort?

    jmp  turn
```

```

; *** Generelle procedurer ***

gotoXY:                ; !!! Gem X i DL og Y i DH !!!
    mov  bh,0           ; Display Page 0
    mov  ah,02h        ; VIDEO Service 02h: Position Cursor
    int  10h
    ret

clrScreen:
    mov  cx,0           ; Vælg hele skærmen.
    mov  dx,184Fh      ; (184Fh <=> Y=24 og X=79)
    mov  al,0          ; 0=Slet i stedet for scroll.
    mov  bh,07h        ; Display attribute = 7 (Normal)
    mov  ah,06h        ; VIDEO Service 06h: Initialize/Scroll
    int  10h
    ret

write:                 ; !!! Gem strengens adresse i DS:DX !!!
    mov  ah,09h        ; DOS-funktion 09h: Print String
    int  21h
    ret

end:
    mov  ax,4C00h      ; Afslut programmet.
    int  21h

; *** Clear Board ***

clrBoard:
    mov  ax,ds         ; Gem Data-segment i ES.
    mov  es,ax

    mov  di,plade      ; Gem pladens start-offset i DI.
    mov  cx,6*7        ; Alle pladens felter (6*7)
    mov  al,'+'        ; skal sættes lig med '+'.

    cld                ; Vi skal skyde +’er nede fra og op.
    rep stosb         ; Her skyder vi vores +’er!

    ret

```

```

; *** Draw Screen ***

drawScreen:
    call clrScreen        ; Rydder skærmen.

    mov  dl,60            ; X=60
    mov  dh,0             ; Y=0
    call gotoXY           ; Placér markør.
    mov  dx,streng1       ; Skriver "9: Afslutter spillet" i
    call write            ;   skærmens øverste højre hjørne.

    mov  dx,streng2       ; Skriver "   1234567"
    call write

    mov  bx,1             ; File Handle=1 (Standard Output)
    mov  cx,7             ; Skriv 7 byte.
    mov  si,0             ; Start med pladens første række.

printRow:
    mov  dx,plade         ; Gem pladens start-offset i DX.
    add  dx,si            ; Gem rækkenr. i DX.

    mov  ah,40h          ; DOS-funktion 40h: Write to File
    int  21h

    mov  dx,streng3       ; Linjeskift + "   "
    call write

    add  si,7             ; Gå til næste række
    cmp  si,42           ;   hvis der er flere...
    jne  printRow

    mov  dx,nylinje       ; Linjeskift.
    call write

    ret

; *** Get Move ***

getMove:
    mov  dx,streng4       ; Skriver "Spiller "
    call write

    mov  dl,[spiller]     ; Skriver spillerens nr.

```

```

    mov ah,02h          ; DOS-funktion 02h: Print Character
    int 21h

    mov dx,streng5      ; Skriver "-----".
    call write

prompt:
    mov dx,streng6      ; Skriver "Indtast træk: "
    call write

    mov ah,0           ; Keyboard Service 00h: Get Key From Buffer
    int 16h

    cmp al,'9'         ; 9 slutter spillet.
    je end

    cmp al,'1'         ; Tester for mindre end 1.
    jl badKey

    cmp al,'7'         ; Tester for større end 7.
    jg badKey

    sub al,49          ; Konverterer ASCII-koden
                        ; til en værdi mellem 0 og 6.
    mov bx,plade       ; Gem pladens start-offset i BX.
    mov ah,00          ; Sørger for at high byte ikke ændrer værdien.
    mov di,ax          ; Gem trækket i DI.
    cmp BYTE [bx+di], '+' ; Hvis det _ikke_ er et +, så er
    jne illegalMove   ; trækket ugyldigt.

    ret

badKey:
    mov dx,streng7      ; Skriver "Du skal indtaste..."
    call write
    jmp prompt

illegalMove:
    mov dx,streng8      ; Skriver "Ugyldigt træk!"
    call write
    jmp prompt

; *** Make Move ***

```

```

makeMove:
    mov  bx,plade           ; Gem pladens start-offset i BX.
    add  bx,35              ; Hop til pladens nederste række.
    mov  di,ax              ; Gem træk (kolonne) i DI.
                                ; AL blev lig med træk i getMove.

itIsNot:
    cmp  BYTE [bx+di], '+'  ; Tester om feltet er tomt (et +).
    je   itIs
    sub  bx,7               ; Hop op til næste række.
    jmp  itIsNot

itIs:
    cmp  BYTE [spiller], '1' ; Er det spiller 1's træk?
    jne  player2
    mov  BYTE [bx+di], 'X'   ; Læg spiller 1's brik (X) på pladen.

    ret

player2:
    mov  BYTE [bx+di], 'O'   ; Læg spiller 2's brik (O) på pladen.

    ret

; *** Check Status ***

checkStatus:
    cmp  BYTE [spiller], '1' ; Er det spiller 1's tur?
    je   yes
    mov  BYTE [spiller], 'O' ; Nej, læg spiller 2's brik i spiller.
    jmp  horizInit

yes:
    mov  BYTE [spiller], 'X' ; Ja, læg spiller 1's brik i spiller.

; --- Tjekker for 4 på stribe vandret ---

horizInit:
    mov  bx,plade           ; Gem pladens start-offset i BX.
    mov  al,[spiller]       ; Flyt spillerens brik ind i AL til testning.

    mov  di,0               ; Start i pladens øverste venstre hjørne.
    mov  cx,0

```

```

horiz:
    cmp [bx+di],al
    jne not1
    cmp [bx+di+1],al
    jne not1
    cmp [bx+di+2],al
    jne not1
    cmp [bx+di+3],al
    jne not1
    jmp gameWon ; Der er 4 på stribe vandret!

not1:
    inc di ; Næste felt.
    inc cx

    cmp di,39 ; Er vi færdige med at tjekke vandret?
    je vertInit ; Ja, fortsæt med lodret.

    cmp cx,4 ; Bevæger vi os over i 3. zone?
    je newRow1 ; Ja, næste række.

    jmp horiz ; Nej, fortsæt med næste felt.

newRow1:
    add di,3 ; Næste række.
    mov cx,0

    jmp horiz ; Gennemløb løkken igen.

; --- Tjekker for 4 på stribe lodret ---

vertInit:
    mov di,0 ; Start i pladens øverste venstre hjørne.

vert:
    cmp [bx+di],al
    jne not2
    cmp [bx+di+7],al
    jne not2
    cmp [bx+di+14],al
    jne not2
    cmp [bx+di+21],al
    jne not2
    jmp gameWon ; Der er 4 på stribe lodret!

```

```

not2:
    inc di                ; Næste felt.

    cmp di,21            ; Er vi færdige med at tjekke lodret?
    je diag1Init        ; Ja, fortsæt med diagonalt.

    jmp vert            ; Nej, fortsæt med næste felt.

; --- Tjekker 1. zone for 4 på stribe diagonalt ---

diag1Init:
    mov di,0            ; Start i pladens øverste venstre hjørne.
    mov cx,0

diag1:
    cmp [bx+di],al
    jne not3
    cmp [bx+di+8],al
    jne not3
    cmp [bx+di+16],al
    jne not3
    cmp [bx+di+24],al
    jne not3
    jmp gameWon        ; Der er 4 på stribe diagonalt!

not3:
    inc di                ; Næste felt.
    inc cx

    cmp di,18            ; Er vi færdige med at tjekke 1. zone?
    je diag2Init        ; Ja, fortsæt med 2. zone.

    cmp cx,4            ; Bevæger vi os over i 3. zone?
    je newRow3          ; Ja, næste række.

    jmp diag1            ; Nej, fortsæt med næste felt.

newRow3:
    add di,3            ; Næste række.
    mov cx,0

    jmp diag1            ; Gennemløb løkken igen.

```

```

; --- Tjekker 2. zone for 4 på stribe diagonalt ---

diag2Init:
    mov di,21                ; Start i række 3 (3*7=21).
    mov cx,0

diag2:
    cmp [bx+di],al
    jne not4
    cmp [bx+di-6],al
    jne not4
    cmp [bx+di-12],al
    jne not4
    cmp [bx+di-18],al
    jne not4
    jmp gameWon              ; Der er 4 på stribe diagonalt!

not4:
    inc di
    inc cx

    cmp di,39                ; Er vi færdige med at tjekke 2. zone?
    je tieInit               ; Ja, fortsæt med test for uafgjort.

    cmp cx,4                 ; Bevæger vi os over i 3. zone?
    je newRow4              ; Ja, næste række.

    jmp diag2                ; Nej, fortsæt med næste felt.

newRow4:
    add di,3                 ; Næste række.
    mov cx,0

    jmp diag2                ; Gennemløb løkken igen.

; --- Tjekker om spillet er uafgjort (ingen tomme felter) ---

tieInit:
    mov di,0

tie:
    cmp BYTE [bx+di], '+'    ; Er der flere tomme felter?

```

```

        je    notTie                ; Ja, fortsæt spillet.

        inc  di
        cmp  di,7                   ; Er sidste felt nået?
        je   gameTie                ; Ja, spillet er uafgjort.
        jmp  tie                    ; Nej, gennemløb løkken igen.

notTie:
        ret

```

```

; --- Spillet er uafgjort ---

```

```

gameTie:
        call drawScreen
        mov  dx,strengB
        call write
        call end

```

```

; --- Spillet er vundet ---

```

```

gameWon:
        cmp  al,'X'                 ; Har spiller 1 vundet?
        jne  player2won

        call drawScreen
        mov  dx,streng9             ; Ja.
        call write
        call end

```

```

player2won:
        call drawScreen
        mov  dx,strengA             ; Nej, spiller 2 vandt.
        call write
        call end

```

```

;-----
;-----Data Section-----
;-----

```

SEGMENT data

plade resb 6*7 ; Pladens højde=6, bredde=7

streng1 db "9: Afslutter spillet",13,10,13,10,'\$'

streng2 db " 1234567",13,10," \$"

streng3 db 13,10," \$"

streng4 db "Spiller \$"

streng5 db 13,10,"-----\$"

streng6 db 13,10,"Indtast Træk: \$"

streng7 db 13,10,13,10,"Du skal indtaste et træk mellem 1 og 7!",13,10,'\$'

streng8 db 13,10,13,10,"Ugyldigt træk!",13,10,'\$'

streng9 db 13,10,"Spiller 1 har vundet!",13,10,'\$'

strengA db 13,10,"Spiller 2 har vundet!",13,10,'\$'

strengB db 13,10,"Spillet er uafgjort!",13,10,'\$'

nylinje db 13,10,'\$'

spiller db '*'

;------

;------Stack Section-----

;------

SEGMENT minStak stack

resb 512

stacktop:

;------End Of File-----

Inden vi går i gang med at gennemgå programmet, vil jeg anbefale dig at prøve det først så du kan se hvordan det skal køre. Programmet er skrevet i Real Mode Segmented Model. Eftersom det jo er et lille program (alt er relativt), så havde det sikkert været bedre at skrive det i Real Mode Flat Model, men det vil jeg overlade til dig som en øvelse. Når jeg har gennemgået programmet, skal jeg nok give nogle tip til hvordan det gøres. Det er faktisk rimelig simpelt.

I starten af programmet fortæller vi assembleren at vi ønsker at generere 16-bit-kode med [BITS 16]. Vores fire på stribe-program er delt op i de tre segmenter

som jo er traditionelle for Real Mode Segmented Model: `SEGMENT code`, `SEGMENT data` og `SEGMENT minStak stack`. Lad os starte med at se på data-segmentet.

I data-sektionen reserverer vi $6*7$ (altså 42) byte i hukommelsen vha. direktivet `resb`. Den første af disse 42 byte er så repræsenteret af navnet `plade`. Disse 42 byte skal bruges til at indeholde oplysninger om spillepladen (eller "spillestativet").

Her er et kort over de 42 byte der illustrerer brugen lidt bedre:

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41

`plade` er lig med adressen på byte 0, dvs. pladens øverste venstre hjørne. Hvis dette felt er tomt, så gemmer vi ASCII-værdien for et + i adressen `plade`. Har spiller 1 en brik i feltet, så gemmer vi ASCII-værdien for et X, og hvis spiller 2 har en brik i feltet, så gemmer vi ASCII-værdien for et O. Hvis vi i stedet skal bruge f.eks. felt nr. 3, skal vi bruge adressen `plade+3` osv. De to optrukne streger på kortet skal du ikke tænke på nu. De gør det mere tydeligt hvad der sker når vi skal til at tjekke om der er fire på striben.

Der er ikke nogen tekniske årsager til at skrive `resb 6*7` i stedet for `resb 42`. Det er for at tydeliggøre at vores plade har 6 rækker og 7 kolonner.

Herefter initialiserer vi en masse strenge. Det er beskeder og kommentarer som vi under programmet skal have vist på skærmen.

Til sidst initialiserer vi en byte `spiller` til at have ASCII-værdien for en *. Denne byte skal bruges til at holde styr på hvis tur det er. Er det spiller 1's tur, gemmer vi ASCII-værdien for 1. Er det spiller 2's tur, så gemmer vi ASCII-værdien for 2. Stjernen er bare en pladsholder under initialiseringen, og har ikke nogen praktisk betydning. Den vil blive erstattet når den første spiller begynder sin tur. Egentlig kunne vi også have reserveret en byte vha. `resb` og på den måde gjort vores .exe-fil én byte mindre, men nu kan vi lettere finde vores byte i et program som DEBUG. Så skal vi lede efter en stjerne (selv om der jo godt kan dukke en stjerne op i .exe-filen på andre måder).

I stak-segmentet reserverer vi 512 byte til stakken `minStak`. Stakken blev gennemgået i afsnit 7, og vi kommer ikke til at bruge den direkte i dette program.

15.1 Main Program

Lad os så komme i gang med selve programmet. De første fem instruktioner efter `..start`: -etiketten initialiserer segmenterne som det blev forklaret i afsnit 5. Herefter kalder vi under rutinen `clrBoard` som sørger for at sætte alle de 42 byte, som repræsenterer pladen, til at være lig med ASCII-værdien for + (tomt felt).

Herefter kommer vi til vores hovedrutine, som vi har markeret som `turn`:

`turn`:

```
mov BYTE [spiller], '1' ; Spiller 1's tur.
call drawScreen        ; Tegner pladen.
```

```

call  getMove           ; Henter spiller 1's træk.
call  makeMove          ; Udfører spiller 1's træk.
call  checkStatus       ; Er spillet vundet eller uafgjort?

mov   BYTE [spiller], '2' ; Spiller 2's tur.
call  drawScreen        ; Tegner pladen.
call  getMove           ; Henter spiller 2's træk.
call  makeMove          ; Udfører spiller 2's træk.
call  checkStatus       ; Er spillet vundet eller uafgjort?

jmp   turn

```

Her har vi delt vores hovedrutine op i to dele, én for spiller 1 og én for spiller 2. Når spiller 2 er færdig med sin tur, tager vi turen endnu en gang med `jmp turn` osv. En tur starter med at sætte den adresse som `spiller` peger på til at være lig med den ASCII-værdi som repræsenterer vores aktuelle spiller (her ASCII-værdien for tegnene 1 og 2). Husk at vi bruger kantede parenteser om `spiller` fordi vi skal ændre indholdet af den adresse som `spiller` peger på, og ikke ændre indholdet af `spiller` selv. Vi sætter apostroffer omkring 1 og 2 for at vise at det ikke er tallet 1, men ASCII-værdien for tegnet 1 som vi vil gemme. Vi kunne selvfølgelig have slået op i en ASCII-tabel og så have skrevet ASCII-værdien direkte, men det er lettere at læse programmet på den måde vi bruger her. Bemærk at vi her skal bruge en type specificer så assembleren ved at `spiller` peger på en *byte*.

Herefter kalder vi de fire store underrutiner som får hele programmet til at fungere: `drawScreen`, `getMove`, `makeMove` og `checkStatus`. Jeg bruger altid engelske navne til mine underrutiner så jeg slipper for at bruge æ, ø og å.

`drawScreen` opdaterer det vi ser på skærmen, dvs. pladen og diverse beskeder. Hver gang en spiller har indtastet sit træk skal pladen opdateres så man kan se den nye brik på pladen. `getMove` er den underrutine som får spillerens træk. Den udskriver en besked, og spilleren skal så indtaste sit træk. `makeMove` udfører trækket, dvs. opdaterer pladen i hukommelsen (de 42 byte som repræsenterer vores plade). Til sidst kalder vi `checkStatus`. Det er den største og mest komplicerede rutine. Det er den som efter hvert træk skal teste om der er fire på stribe, enten vandret, lodret eller diagonalt. Hvis det er tilfældet, så er spillet slut. Hvis ikke, så vender vi tilbage til hovedrutinen, og spillet fortsætter.

15.2 Generelle procedurer

Efter hovedprogrammet har vi lige nogle generelle procedurer. Dem laver vi så vi i resten af programmet blot kan skrive `call clrScreen` for at rydde skærmen. Det gør programmet lettere at læse.

`gotoXY`-rutinen bruges til at placere markøren et bestemt sted på skærmen:

```

gotoXY:                ; !!! Gem X i DL og Y i DH !!!
    mov  bh,0           ; Display Page 0
    mov  ah,02h        ; VIDEO Service 02h: Position Cursor

```

```
int 10h
ret
```

Som du kan se af `int 10h`-instruktionen bruger vi interrupt 10h: Video Display Services. Jeg har samlet nogle få af disse "VIDEO services" i appendikset. Når man kalder interrupt 10h, forventes det at nummeret på den ønskede service er gemt i register AH. Vi gemmer derfor 02h i AH lige inden vi kalder interruptet. I appendikset kan vi se at service 2 hedder Position Cursor, og den kan bruges til at placere markøren et bestemt sted på skærmen. I beskrivelsen af service 2 står der at vi skal gemme et 0 i BH, så det gør vi også inden kaldet. 0'et angiver Display Page 0. Display Pages er noget halvkompliceret noget, og derfor har jeg bare skrevet at man skal gemme et 0 i BH. Der står også i beskrivelsen at vi skal gemme X-koordinaten for markørens ønskede position i DL og Y-koordinaten i DH. Det er der dog ikke meget mening i at gøre i selve rutinen. Det skal gøres inden man kalder `gotoXY`. Vi husker naturligvis at afslutte rutinen med `ret` for return så vi vender tilbage til det kaldende program.

`clrScreen`-rutinen bruges til at rydde skærmen:

```
clrScreen:
    mov cx,0           ; Vælg hele skærmen.
    mov dx,184Fh      ; (184Fh <=> Y=24 og X=79)
    mov al,0          ; 0=Slet i stedet for scroll.
    mov bh,07h        ; Display attribute = 7 (Normal)
    mov ah,06h        ; VIDEO Service 06h: Initialize/Scroll
    int 10h
    ret
```

Her bruger vi VIDEO service 6: Initialize/Scroll. Derfor gemmer vi et 6 i AH. I CL og CH skal vi gemme X- og Y-koordinaterne for det øverste venstre hjørne af det skærmareal som vi vil påvirke, og i DL og DH skal vi gemme koordinaterne for det nederste højre hjørne. Husk på at skærmens øverste venstre hjørne har koordinaterne (0,0). Skærmens nederste højre hjørne har koordinaterne (79,24) så længe du bruger en skærm med 25 linjer a 80 tegn. Her skal vi bruge hele skærmens areal. Vi sætter CX til 0 så både CL og CH bliver 0. DX sætter vi til 184Fh. 18h er 24 i decimal og 4F er 79 i decimal. I AL kan man angive hvor mange linjer området som man har markeret, skal scrolles (rulles som når man trykker Enter i bunden af skærmen). Hvis man angiver et 0, slettes området. Vi har gemt et 0 så området slettes. I BH gemmes en display attribute. Her gemmer vi 7 som er det normale. Efter alt dette, kan vi nu kalde interrupt 10h, og derefter vende tilbage til det kaldende program med `ret`.

`write`-rutinen gør ikke andet end at kalde DOS-funktionen Print String som så udskriver den streng på skærmen hvis start-adresse er gemt i DS:DX.

`end`-rutinen slutter programmet.

15.3 Clear Board

```
clrBoard:
    mov ax,ds
```

```

mov  es,ax          ; Gem Data-segment i ES.

mov  di,plade      ; Gem pladens start-offset i DI.
mov  cx,6*7        ; Alle pladens felter (6*7)
mov  al,'+'        ; skal sættes lig med '+'.

    cld            ; Vi skal skyde +’er nede fra og op.
rep  stosb         ; Her skyder vi vores +’er!

ret

```

Denne rutine kaldes inden spillets start. Den skal sørge for at sætte alle pladens felter lig med ASCII-værdien for tegnet + som er vores symbol for at feltet er tomt. Her bruger vi en af CPU’ens mere avancerede instruktioner, nemlig `stosb` (**St**ore **St**ring by **B**yte). Den har jeg ikke omtalt før, men her kommer lidt forklaring. Den bruges til at kopiere indholdet af register AL til adressen som fremkommer vha. ES:DI. Vi vil gerne kopiere ASCII-værdien for + til den adresse som `plade` peger på. Derfor kopierer vi vores datasegment-adresse over i ES (husk at man ikke kan kopiere direkte fra DS til ES, så vi bruger AX som mellemmand). I DI gemmer vi pladens start-offset, dvs. adressen på pladens første byte. Nu er vi egentlig klar til at udføre `stosb`. Men det hjælper jo ikke meget kun at tømme det første felt på pladen. Vi skal gerne have tømt alle 42 felter! Her kan vi bruge `rep`-præfikset til `stosb`. `rep` står for **re**peat (gentag) og udfører altså `stosb`-instruktionen mere end én gang. Hvor mange gange den skal gentages angiver vi i CX, så vi flytter 42 (6*7) ind i CX så vi får tømt alle felterne. Nu kan man undre sig over om man på denne måde ikke bare skyder en masse +’er ind på den samme plads. ES:DI peger jo stadig på pladens første felt, ikke? Men sådan foregår det ikke. `rep` sørger nemlig for at forhøje DI med 1 for hver gentagelse. Inden vi begynder at “skyde” +’er, udfører vi kommandoen `cld` (**C**lear **D**irection **F**lag). Det sætter flaget DF til 0. Når DF er 0, så vil `rep` forhøje DI med 1 for hver gentagelse. Er DF lig med 1, så vil `rep` formindske DI med 1 for hver gentagelse, og på den måde gå “tilbage” i hukommelsen.

Når alle pladens felter er sat lig med ASCII-værdien for +, så er `clrBoard` færdig med arbejdet, og kontrollen kan vende tilbage til det kaldende program.

15.4 Draw Screen

Denne rutine kaldes hver gang skærbilledet skal opdateres. Vi starter med at kalde rutinen `clrScreen` som rydder skærmen. De næste fem linjer er her:

```

mov  dl,60          ; X=60
mov  dh,0           ; Y=0
call gotoXY        ; Placér markør.
mov  dx,streng1    ; Skriver "9: Afslutter spillet" i
call write         ; skærmens øverste højre hjørne.

```

Vi kalder vores rutine `gotoXY` som kræver at vi gemmer koodinaterne for vores ønskede position i DL og DH. Vi skal have skrevet en besked om hvordan man afslutter

spillet i højre hjørne. Den besked har vi defineret i datasegmentet og markeret starten af den med navnet `streng1`. For at kunne udskrive strengen, skal start-adressen ligge i `DX`, så vi kopierer den derover. Bemærk her at vi ikke bruger kantede parenteser da vi netop skal bruge *adressen* på strengen inden vi kalder vores `write`-rutine.

```
mov dx,streng2      ; Skriver " 1234567"  
call write
```

Her skriver vi en række numre således at hver kolonne har et nummer mellem 1 og 7. Vi rykker også lige et par mellemrum ind på skærmen så det kommer til at se lidt pænt ud.

Nu skal vi have "tegnet" vores plade. Da vi gemmer ASCII-værdierne +, X og O direkte i hukommelsen, kan vi skrive dem direkte på skærmen. Men vi kan ikke bruge DOS-funktion 9: Print String da den forventer at strengen afsluttes med et \$. Vi vælger i stedet at bruge DOS-funktion 40h: Write To File. Den forventer at vi gemmer et *file handle* i `BX`. Et file handle er et nummer som repræsenterer en åben fil. Nummer 1 er dog Standard Output, og hvis vi skriver til "filen" med file handle 1, så skriver vi faktisk til skærmen! Så vi sætter `BX=1`:

```
mov bx,1            ; File Handle=1 (Standard Output)  
mov cx,7            ; Skriv 7 byte.
```

DOS-funktion 40h skal selvfølgelig også vide hvornår den skal slutte med at skrive tegn. Den forventer at antallet af byte som skal skrives er gemt i `CX`. Vi skal udskrive en række ad gangen, og vores plade har 7 kolonner, så vi gemmer 7 i `CX`.

```
mov si,0            ; Start med pladens første række.
```

`printRow:`

```
mov dx,plade        ; Gem pladens start-offset i DX.  
add dx,si           ; Gem rækkenr. i DX.
```

```
mov ah,40h          ; DOS-funktion 40h: Write to File  
int 21h
```

Vi sætter `SI` til at være 0. `SI` skal holde styr på hvilken række vi skal til at udskrive. Vi starter med den første række. DOS-funktion 40h forventer at start-adressen på de data der skal udskrives, ligger i `DS:DX`. Så vi sætter `DX` til at være lig med den adresse som `plade` peger på. Herefter lægger vi indholdet af `SI` til `DX` og gemmer resultatet i `DX` (`add`-instruktionen). Når `SI` er 0, så er `DX` stadig lig med den adresse som `plade` peger på som jo er første felt (og første række) på pladen. Vi kalder nu funktion 40h som udskriver 7 byte (pladens første 7 felter, dvs. den første række) på skærmen.

```
mov dx,streng3      ; Linjeskift + " "  
call write
```

Når det er gjort, skifter vi linje og skriver 4 mellemrum igen.

```

add  si,7          ; Gå til næste række
cmp   si,42        ; hvis der er flere...
jne  printRow

```

Vi lægger 7 til SI. Hvis SI var 0 i forvejen bliver SI nu 7. Når vi nu kører rutinen igen fra `printRow:`, så bliver DX lig med `plade+7`, og peger således på anden række (se kortet over pladen). Inden vi haster videre med at udskrive næste række, skal vi dog lige sørge for at teste om SI er blevet lig med 42 hvilket betyder at vi er klar til at udskrive række nr. 7. Da vi kun har 6 rækker, er det en dårlig en idé at udskrive række 7. Så vi tester om SI er lig med 42. Hvis det ikke er tilfældet (**J**ump **I**f **N**ot **E**qual), kan vi roligt springe til `printRow:`. Hvis SI er lig med 42, går vi videre til næste instruktion:

```

mov  dx,nylinje    ; Linjeskift.
call write

ret

```

Vi udskriver et linjeskift så der bliver lidt afstand mellem pladen og den efterfølgende tekst.

15.5 Get Move

```

getMove:
mov  dx,streng4    ; Skriver "Spiller "
call write

```

`getMove`-rutinen bruger vi til at hente spillerens træk. Spilleren skal udføre sit træk ved at indtaste et tal mellem 1 og 7 for at angive hvilken kolonne (eller søjle i stativet) som han eller hun vil lade sin brik falde ned i. De første to linjer udskriver den streng som `streng4` peger på, dvs. "Spiller ". Bagefter skal vi have skrevet spillerens nummer. ASCII-værdien for nummeret står i `[spiller]`, dvs. at det står i den adresse som spiller peger på. Da vi bruger ASCII-værdier til at holde styr på hvis tur det er, kan vi udskrive indholdet direkte på skærmen. Vi bruger her DOS-funktion 2: Print Character:

```

mov  dl,[spiller] ; Skriver spillerens nr.
mov  ah,02h       ; DOS-funktion 02h: Print Character
int  21h

```

DOS-funktion 2 udskriver et enkelt tegn på skærmen. ASCII-koden for tegnet skal ligge i DL, så vi lægger indholdet af den adresse som spiller peger på (`[spiller]`) ind i DL.

Herefter laver vi en understregning så det kommer til at se pænt ud.

```

mov  dx,streng5    ; Skriver "-----".
call write

```

Vi skriver en prompt så spilleren kan se at der nu skal indtastes et træk:

```
prompt:
    mov dx,streng6      ; Skriver "Indtast træk: "
    call write
```

Vi skal nu have læst et tegn fra tastaturet. Det gør vi vha. DOS-interrupt 16h som giver os adgang til nogle keyboard services. Når vi kalder DOS-interrupt 16h med int 16h, skal vi i AH angive hvilken service vi ønsker at benytte os af. I vores tilfælde skal vi have læst et tegn fra tastaturet. Det er Keyboard Service 0: Get Key From Keyboard Buffer:

```
    mov ah,0           ; Keyboard Service 00h: Get Key From Buffer
    int 16h
```

Keyboard Service 0 returnerer ASCII-koden for det indtastede tegn i AL. Vi skal nu sikre os at det indtastede er lovligt i vores spil. Vi har allerede skrevet en besked på skærmen om at et tryk på 9-tasten afslutter spillet. Så det tester vi for her:

```
    cmp al,'9'        ; 9 slutter spillet.
    je end
```

Vi husker at vi har med ASCII-koder at gøre, og derfor sætter vi apostroffer omkring 9-tallet for at angive ASCII-værdien for tegnet 9. AL indeholder fra Keyboard Service 0 ASCII-værdien for det som spilleren indtastede. Det sammenligner vi så med ASCII-værdien for 9 (cmp al,'9'). Hvis disse to værdier er ens (Jump If Equal), så har spilleren indtastet 9, og ønsker at afslutte spillet; vi hopper til end-rutinen. Hvis de to værdier er forskellige fra hinanden, går computeren videre til næste instruktion. Det er en ny test:

```
    cmp al,'1'        ; Tester for mindre end 1.
    jl badKey
```

Denne gang tester vi med ASCII-værdien for tegnet 1. Hvis vi slår op i en ASCII-tabel, kan vi se at tegnet 1 har ASCII-værdien 49. Alle de tegn hvis ASCII-værdi er *mindre end* 49 er derfor *ikke* et af vores lovlige tal. Vi kan derfor bruge jl (Jump If Less). Hvis spilleren har indtastet et tegn hvis ASCII-værdi er mindre end 49, hopper vi til badKey som ligger længere nede i getMove-rutinen. Vi skal også have afskaffet alle de tegn, som har en ASCII-værdi på over 55 som er ASCII-værdien for tegnet 7. Det gør vi med jg (Jump If Greater):

```
    cmp al,'7'        ; Tester for større end 7.
    jg badKey
```

Hvis vi er kommet så langt som hertil, er vi nu sikre på at spilleren har tastet et tal mellem 1 og 7, og AL indeholder en værdi mellem 49 og 55 (ASCII-værdierne for tegnene 1 til 7). Vi skal nu finde ud af om trækket er gyldigt. Hvis spilleren f.eks. vælger at smide en brik i søjle 3, og søjle 3 allerede er fyldt op, så er trækket ugyldigt. Vi skal altså have testet pladens øverste række (følg med på skemaet over pladen):

```

sub  al,49                ; Konverterer ASCII-koden
                           ; til en værdi mellem 0 og 6.
mov  bx,plade            ; Gem pladens start-offset i BX.
mov  ah,00               ; Sørger for at high byte ikke ændrer værdien.
mov  di,ax               ; Gem trækket i DI.
cmp  BYTE [bx+di], '+'   ; Hvis det ikke er et +, så er
jne  illegalMove        ; trækket ugyldigt.

ret

```

Vi starter med at trække 49 fra det tal som ligger i AL. Vi husker at AL indeholder et tal mellem 49 og 55. Ved at trække 49 fra tallet, får vi et tal mellem 0 og 6 som er noget mere hensigtsmæssigt i de efterfølgende instruktioner. Vi gemmer `plade` i BX. BX indeholder nu adressen for felt 0, dvs. pladens øverste venstre hjørne. Vi vil gerne have gemt AL i DI (forklaringen på dette kommer om lidt). Vi husker her at DI er et 16-bit-register, mens AL er et 8-bit-register, eller rettere: den lave halvdel af 16-bit-registret AX. Vi kan derfor ikke lægge AL over i DI. I stedet gemmer vi et 0 i AH (AX's øvre halvdel) og så indeholder AX jo værdien AL (vores oprindelige værdi i AL har bare fået foranstillet 8 nuller). Nu kan vi flytte AX over i DI da de begge er 16-bit-registre. Grunden til denne "register-akrobatik" er at vi med instruktionen `cmp BYTE [bx+di], '+'` nu kan teste lige præcis det rigtige felt. BX indeholder pladens første felt, og når vi lægger DI (som har en værdi mellem 0 og 6) til, får vi det rigtige felt. Vi tester så om det indeholder ASCII-værdien for tegnet + som i vores spil repræsenterer et tomt felt. Gør det *ikke* det, er feltet allerede optaget, og hvis det øverste felt i en søjle er optaget, så er hele søjlen fyldt op, og trækket er ugyldigt. Så vi hopper til rutinen `illegalMove` som ligger længere nede i `getMove`-rutinen.

```

badKey:
    mov dx,streng7        ; Skriver "Du skal indtaste..."
    call write
    jmp  prompt

```

Her kommer vi til hvis spilleren har indtastet noget som ikke var et tal mellem 1 og 7 (vi har allerede testet for 9 som afslutter spillet), og vi skriver her en meddelelse til spilleren om at indtaste et tal mellem 1 og 7. Så hopper vi tilbage til `prompt:`, og venter på en ny indtastning.

```

illegalMove:
    mov dx,streng8        ; Skriver "Ugyldigt træk!"
    call write
    jmp  prompt

```

Her ender vi hvis spillerens træk er ugyldigt, dvs. at der ikke er plads til flere brikker i den ønskede søjle. Spilleren får besked og vi hopper tilbage til `prompt:`.

15.6 Make Move

Vi har nu fået spillerens træk fra `getMove`-rutinen og sikret os at det er gyldigt. Trækket er et tal mellem 0 og 6 og blev gemt i AX hvor det stadig ligger når vi kommer til `makeMove`-rutinen. Det der skal ske her, er at vi skal have opdateret pladen i hukommelsen med det nye træk. Det nye træk bliver senere udskrevet på skærmen når den bliver opdateret i `drawScreen`-rutinen.

```
makeMove:
    mov  bx,plade           ; Gem pladens start-offset i BX.
    add  bx,35             ; Hop til pladens nederste række.
    mov  di,ax             ; Gem træk (kolonne) i DI.
                                ; AL blev lig med træk i getMove.
```

Vi gemmer pladens start-offset (felt 0) i BX. Herefter lægger vi 35 til værdien så vi kommer ned på pladens nederste række (se igen skemaet over pladen). Så lægger vi trækket fra AX ind i DI. Adressen som BX+DI peger på, er nu det nederste felt i den ønskede søjle. Vi skal nu teste om dette felt er tomt så vi kan lægge en brik i det:

```
itIsNot:
    cmp  BYTE [bx+di], '+'   ; Tester om feltet er tomt (et +).
    je   itIs
    sub  bx,7                ; Hop op til næste række.
    jmp  itIsNot
```

Her tester vi så for vores +. Hvis der er et + i det pågældende felt, så er feltet tomt, og vi hopper til `itIs`-rutinen længere nede i `makeMove`-rutinen. Hvis feltet ikke er tomt, så trækker vi 7 fra BX. På den måde kommer vi til rækken lige oven over den nuværende. Hvis det nuværende felt f.eks. er felt 37, så trækker vi 7 fra og befinder os så på felt 30 som er feltet lige ovenover (se skemaet igen). Vi tager så testen endnu engang (`jmp itIsNot`) for at se om dette felt så er tomt, og fortsætter så indtil vi finder søjlens nederste tomme felt. Husk på at vi allerede har testet om der overhovedet er et tomt felt i søjlen. Det gjorde vi i `getMove`-rutinen. Vi ved altså at der er et tomt felt, spørgsmålet er bare hvor det er. Når vi finder det, skal vi have placeret en brik i feltet:

```
itIs:
    cmp  BYTE [spiller], '1' ; Er det spiller 1's træk?
    jne  player2
    mov  BYTE [bx+di], 'X'   ; Læg spiller 1's brik (X) på pladen.

    ret

player2:
    mov  BYTE [bx+di], 'O'   ; Læg spiller 2's brik (O) på pladen.

    ret
```

Vi tjekker om det er spiller 1 der har turen. I så fald skal spiller 1's brik lægges på det tomme felt som vi lige har fundet. Det gør vi ved at lægge ASCII-værdien for tegnet X ind på den adresse som BX+DI peger på. Er det spiller 2's tur, springer vi ned til `player2:` og lægger ASCII-værdien for tegnet O ind på den rette plads.

15.7 Check Status

Vi kommer nu til den sidste af de store underrutiner som får spillet til at fungere. Det er den rutine som efter hvert træk skal teste om der er 4 på stribe eller om spillet er uafgjort (pladen er fyldt op uden nogen vinder). Det er den største og mest komplicerede rutine, men lad os se på den stykke for stykke:

```
checkStatus:
    cmp  BYTE [spiller], '1' ; Er det spiller 1's tur?
    je   yes
    mov  BYTE [spiller], 'O' ; Nej, læg spiller 2's brik i spiller.
    jmp  horizInit

yes:
    mov  BYTE [spiller], 'X' ; Ja, læg spiller 1's brik i spiller.
```

Vi skal have testet vores plade på kryds og tværs efter de rigtige kombinationer. Vores plade består af +'er, X'er og O'er. `[spiller]` indeholder enten tegnet 1 eller 2 alt efter hvis tur det er. Her er det hensigtsmæssigt at erstatte 1 eller 2 med X eller O så vi kan sammenligne pladens felter med `[spiller]`. Det gør vi ved at teste om `[spiller]` har ASCII-værdien for tegnet 1. Hvis det er tilfældet, hopper vi til `yes:` som erstatter '1' med 'X'. Hvis `[spiller]` ikke indeholder '1', så indeholder det ASCII-værdien for tegnet 2, og vi kan så gemme 'O' i `[spiller]` og hoppe over `yes:` vha. `jmp horizInit`.

```
horizInit:
    mov  bx,plade           ; Gem pladens start-offset i BX.
    mov  al,[spiller]      ; Flyt spillernes brik ind i AL til testning.

    mov  di,0              ; Start i pladens øverste venstre hjørne.
    mov  cx,0
```

Vi skal nu til at teste om der er fire på stribe vandret—et eller andet sted på pladen. Vi gemmer `plade`'s start-offset (felt 0) i BX. Vi skal til at teste det område i hukommelsen som indeholder vores plade, og da det ikke er muligt at sammenligne memory-data med memory-data, flytter vi `[spiller]` ind i AL. På den måde kommer vi til at sammenligne memory-data med register-data.

Vi tager et kig på skemaet over pladen, og finder ud af at hvis spilleren har en brik i felt 0, så skal spilleren også have brikker i felt 0+1, felt 0+2 og felt 0+3 for at have fire på stribe vandret. Hvis det er tilfældet, er spillet vundet. Hvis ikke, tager vi felt 1 som udgangspunkt. Så skal spilleren have brikker i felt 1, felt 1+1, felt 1+2 og felt 1+3 for at have 4 på stribe vandret. Sådan fortsættes indtil vi kommer til felt 4. Her ser vi at

felt 4+3 er lig med felt 7, som jo ligger på næste række. Vi skal altså ikke bruge felt 4, 5 og 6 som udgangspunkt for testning. Jeg har trukket en lodret linje på kortet på side 9 mellem søjle 3 og søjle 4 (den første søjle er søjle 0). Vi skal altså have gennemløbet en løkke hvor alle felter på venstre side af den optrukne linje, tur efter tur, bliver brugt som udgangspunkt for en test af om der fire på stribe vandret.

Da BX indeholder adressen på felt 0, bruger vi DI til at gemme nummeret for feltet som skal være udgangspunkt. CX bruger vi til at sørge for at vi holder os på venstre side af den optrukne streg.

```

horiz:
    cmp [bx+di],al
    jne not1
    cmp [bx+di+1],al
    jne not1
    cmp [bx+di+2],al
    jne not1
    cmp [bx+di+3],al
    jne not1
    jmp gameWon ; Der er 4 på stribe vandret!

```

Her går testen løs. Vi sammenligner felterne med AL som indeholder den aktuelle spillers brik. Hvis bare ét felt ikke indeholder samme værdi som AL, så har spilleren ikke en brik i dette felt, og spilleren har ikke fire på stribe i dette tilfælde. Så springer vi til not1:. Hvis vi kommer helskindet igennem testen, så er der fire på stribe vandret, og vi springer til gameWon: som ligger længere nede i checkStatus-rutinen.

```

not1:
    inc di ; Næste felt.
    inc cx

    cmp di,39 ; Er vi færdige med at tjekke vandret?
    je vertInit ; Ja, fortsæt med lodret.

    cmp cx,4 ; Bevæger vi os over i 3. zone?
    je newRow1 ; Ja, næste række.

    jmp horiz ; Nej, fortsæt med næste felt.

newRow1:
    add di,3 ; Næste række.
    mov cx,0

    jmp horiz ; Gennemløb løkken igen.

```

Hvis der ikke var fire på stribe, er vi klar til at tage næste felt som udgangspunkt for en test. Vi forhøjer DI med 1 (inc di), og vi forhøjer CX med 1. Hvis DI nu er lig med 39, så er vi kommet ned til bunden af pladen og står lige til højre for den optrukne linje

(se skemaet), og vi kan hoppe videre til `vertInit`: hvor vi tester for fire på stribe lodret. Ellers går vi videre og tester om CX er lig med 4. Hvis det er tilfældet, så er vi kommet over på den højre side af stregen på kortet, og vi skal hoppe til næste række. Det gør vi i `newRow1`: hvor vi lægger 3 til DI som så vil starte på en ny række, og vi sætter CX tilbage til 0 så den kan holde øje med stregen på kortet igen. Vi hopper så tilbage og tester for fire på stribe vandret igen. Denne gang med DI's nye værdi som udgangspunkt.

Hvis der ikke var fire på stribe vandret, går programmet videre og tester for fire på stribe lodret. Vi nulstiller DI så vi igen starter med pladens øverste venstre hjørne som udgangspunkt:

```
vertInit:
    mov di,0                ; Start i pladens øverste venstre hjørne.
```

Hvis vi igen kigger på kortet og siger at felt 0 er udgangspunkt, så skal den aktuelle spiller have en brik i felt 0, felt 0+7, felt 0+14 og felt 0+21 for at have 4 på stribe lodret. Jeg har trukket en vandret linje på kortet, og når vi kommer under denne linje, er der ikke flere muligheder for at have fire på stribe lodret. Det fungerer altså i princippet på samme måde som testen af fire på stribe vandret. Dog er der her den fordel at vi bare forhøjer DI med 1 for hver gennemkørsel indtil DI er lig med 21, og vi slipper for at skulle rode med CX:

```
vert:
    cmp [bx+di],al
    jne not2
    cmp [bx+di+7],al
    jne not2
    cmp [bx+di+14],al
    jne not2
    cmp [bx+di+21],al
    jne not2
    jmp gameWon             ; Der er 4 på stribe lodret!

not2:
    inc di                  ; Næste felt.

    cmp di,21              ; Er vi færdige med at tjekke lodret?
    je diag1Init          ; Ja, fortsæt med diagonalt.

    jmp vert               ; Nej, fortsæt med næste felt.
```

Hvis DI er lig med 21, har vi testet for alle muligheder for fire på stribe lodret, og vi kan gå videre til `diag1Init`: , som tester for fire på stribe diagonalt.

Når vi skal teste for fire på stribe diagonalt, deler vi pladen op i tre "zoner". Zonerne afgrænses af de to optrukne streger på skemaet. Zone 1 er det areal der er øverst til venstre. Zone 2 er arealet der er nederst til venstre. Zone 3 er hele området til højre for den lodrette streg.

Vi starter med at teste zone 1 for fire på stribe diagonalt, dvs. at alle felterne i zone 1 skal bruges som udgangspunkt i test. Når vi starter med felt 0, så skal den aktuelle spiller have brikker i felt 0, felt 0+8, felt 0+16 og felt 0+24 for at have fire på stribe diagonalt. Vi sætter DI til 0 (felt 0) og CX til 0 da vi skal holde øje med at vi ikke bevæger os over i zone 3:

```
diag1Init:
    mov  di,0                ; Start i pladens øverste venstre hjørne.
    mov  cx,0

diag1:
    cmp  [bx+di],al
    jne  not3
    cmp  [bx+di+8],al
    jne  not3
    cmp  [bx+di+16],al
    jne  not3
    cmp  [bx+di+24],al
    jne  not3
    jmp  gameWon            ; Der er 4 på stribe diagonalt!
```

Hvis testen ikke resulterer i at der er fire på stribe, så går vi videre med næste felt:

```
not3:
    inc  di                ; Næste felt.
    inc  cx

    cmp  di,18            ; Er vi færdige med at tjekke 1. zone?
    je   diag2Init       ; Ja, fortsæt med 2. zone.

    cmp  cx,4            ; Bevæger vi os over i 3. zone?
    je   newRow3        ; Ja, næste række.

    jmp  diag1           ; Nej, fortsæt med næste felt.

newRow3:
    add  di,3            ; Næste række.
    mov  cx,0

    jmp  diag1           ; Gennemløb løkken igen.
```

Det her skulle gerne minde dig om testen for fire på stribe vandret. Vi forhøjer DI og CX med 1. Hvis DI herefter er lig med 18, så er vi færdige med at teste zone 1 for fire på stribe diagonalt, og hopper videre til `diag2Init`: som tester zone 2. Hvis CX er lig med 4, så er vi kommet over i zone 3, og vi hopper ned til `newRow3`: som lægger 3 til DI så vi kommer ned til starten af næste række, og CX sættes tilbage til 0 så den er klar til at holde øje med zonegrænsen igen.

Hvis der ikke var fire på stribe diagonalt i zone 1, går vi til zone 2. Vi sætter her DI til at være lig med 21 som er første felt i zone 2. Vi nulstiller også CX.

```
diag2Init:
    mov di,21                ; Start i række 3 (3*7=21).
    mov cx,0
```

Her i zone 2 skal vi teste på en lidt anden måde end i zone 1 hvor vi testede for diagonalt “nedad mod højre”. I zone 2 skal vi teste for diagonalt “opad mod højre”. Hvis vi f.eks. står i felt 21, så skal den aktuelle spiller have brikker i felt 21, felt 21-6=15, felt 21-12=9 og felt 21-18=3 for at have fire på stribe diagonalt:

```
diag2:
    cmp [bx+di],al
    jne not4
    cmp [bx+di-6],al
    jne not4
    cmp [bx+di-12],al
    jne not4
    cmp [bx+di-18],al
    jne not4
    jmp gameWon             ; Der er 4 på stribe diagonalt!
```

Hvis der ikke er fire på stribe, fortsættes med næste felt:

```
not4:
    inc di
    inc cx

    cmp di,39                ; Er vi færdige med at tjekke 2. zone?
    je tieInit              ; Ja, fortsæt med test for uafgjort.

    cmp cx,4                 ; Bevæger vi os over i 3. zone?
    je newRow4              ; Ja, næste række.

    jmp diag2                ; Nej, fortsæt med næste felt.

newRow4:
    add di,3                 ; Næste række.
    mov cx,0

    jmp diag2                ; Gennemløb løkken igen.
```

Vi forøger DI og CX med 1. Hvis DI herefter er blevet 39, er vi kommet over i zone 3 og kan springe videre til `tieInit`: som tester om pladen er fyldt op og spillet dermed er uafgjort. Vi tester om CX er 4 og hopper i så fald til `newRow4`: som lægger 3 til DI og dermed peger på næste rækkes første felt på pladen.

Hvis der ikke har været fire på stribe overhovedet, skal vi teste om pladen er fyldt, og spillet dermed er uafgjort. Det kræver kun at vi tester pladens øverste række:

```

tieInit:
    mov  di,0

tie:
    cmp  BYTE [bx+di], '+'      ; Er der flere tomme felter?
    je   notTie                ; Ja, fortsæt spillet.

    inc  di
    cmp  di,7                  ; Er sidste felt nået?
    je   gameTie               ; Ja, spillet er uafgjort.
    jmp  tie                    ; Nej, gennemløb løkken igen.

notTie:
    ret

```

Vi sætter DI til 0 (pladens øverste venstre hjørne). Så tester vi om feltet er tomt (så vil der være et '+'). Hvis det er tilfældet, så er pladen endnu ikke fyldt og vi hopper til `notTie`: som vender tilbage til hovedrutinen og fortsætter spillet. Hvis feltet ikke var tomt, forhøjer vi DI med 1 så det peger på næste felt. Hvis DI er lig med 7, så er vi kommet ned på række 2 uden at finde et tomt felt i øverste række og spillet er dermed uafgjort. Vi hopper derfor til `gameTie`:

Som du kan se, så skal computeren igennem rigtig mange ting hver eneste gang en spiller har udført et træk, men det hele sker på langt under et sekund (medmindre du har en ufattelig langsom computer).

```

gameTie:
    call drawScreen
    mov  dx,strengB
    call write
    call end

```

Denne rutine kommer vi til hvis spillet er uafgjort. Vi kalder `drawScreen`-rutinen så spillerne kan se at det sidste træk blev udført. Derefter udskriver vi en besked om resultatet på skærmen, og til sidst kalder vi `end`-rutinen som afslutter programmet.

```

gameWon:
    cmp  al,'X'                ; Har spiller 1 vundet?
    jne  player2won

    call drawScreen
    mov  dx,streng9            ; Ja.
    call write
    call end

```

Hvis en spiller har fået fire på stribe, kommer vi hertil. Vi tester om det var spiller 1 der vandt ved at sammeligne AL med 'X'. Vi husker at AL har indeholdt den aktuelle spillers brik gennem `checkStatus`-rutinen. Hvis det ikke er tilfældet, så var det

spiller 2 der vandt, og der hoppes til `player2won`: . Ellers opdateres skærbilledet så vi kan se det træk der afgjorde spillet. Der skrives en besked om resultatet på skærmen og end-rutinen kaldes.

```
player2won:
    call drawScreen
    mov dx,strengA          ; Nej, spiller 2 vandt.
    call write
    call end
```

Her ender vi hvis det var spiller 2 der vandt spillet. Skærbilledet opdateres, der skrives en besked på skærmen og programmet afsluttes.

15.8 Optimering af programmet

Hvis vi kompilerer og linker programmet, får vi en `.exe`-fil ud af det som fylder 918 byte. Det er jo i sig selv en meget lille fil nu om dage, men den kan blive endnu mindre! Vi kan jo starte med at konvertere det til Real Mode Flat Model og dermed lave en `.com`-fil i stedet. Jeg vil give lidt grundlæggende tip om dette, og overlade resten af det til dig som en øvelse.

Start med at tage et kig på det lille program i starten af afsnit 8. Som du kan se, så kan du starte med at slette hele stak-segmentet i vores program.

Du skal også huske at en `.com`-fil bruger de første 256 byte i det segment som du får stillet til rådighed. Derfor indsætter du direktivet `[ORG 100h]` i starten af programmet.

Et program i Real Mode Flat Model eksisterer kun inden for ét segment, så derfor skal du ikke opdele programdelene i segmenter, men i sektioner. Kode-delen skal markeres som `[SECTION .text]`. Datadelen skal markeres som `[SECTION .data]`. I den forbindelse kommer her noget som jeg ikke har været inde på før. I sektionen `.data` skal du definere dine initialiserede data, dvs. de data som du giver en værdi med det samme (f.eks. fast definerede strenge). Dine ikke-initialiserede data skal være i en sektion som du markerer med `[SECTION .bss]`. Her skal du definere de områder i hukommelsen som du reserverer med f.eks. `resb`. I vores tilfælde skal plade altså defineres i denne sektion.

Vi skal slette de instruktioner som har med segmentregistre at gøre idet hele vores program ligger i samme segment. Vi kan altså fjerne de første fem instruktioner i programmet. Det er dem som initialiserer segmenterne i Real Mode Segmented Model. I rutinen `clrBoard` har vi også to instruktioner som behandler DS og ES. Disse instruktioner slettes også.

Til sidst skal vi slette etiketten `..start:`.

Hvis vi nu kompilerer til en `.com`-fil, kommer vores program til fylde 780 byte, altså 138 byte mindre end før. Vores lille program kan allerede ligge på én almindelig 1.44MB-diskette langt over tusind gange, så måske er der ikke nogen grund til at anstrenge sig for at gøre det mindre. Men du kan godt gøre det, for programkoden er ikke rigtigt optimeret. Hvis du ser den grundigt efter, kan du måske finde nogle instruktioner som egentlig er overflødige. For eksempel hvis man initialiserer et register

som allerede har den rigtige værdi fra en anden rutine, eller `jmp`-instruktioner der kunne være placeret mere hensigtsmæssigt.

Med det afsluttende eksempel slutter også denne artikel, men der er gode muligheder for at lære mere på internettet. Du kan evt. starte din tur med at lægge vejen forbi siden www.programmersheaven.com hvor der er en hel sektion om assembler-programmering med artikler, eksempler og diskussionsfora.

A Appendiks

I dette appendiks kan du finde en oversigt over CPU'ens registre, flagene og hvad man kalder de forskellige mængder af hukommelse. Jeg har også samlet de mest interessante Video Services, Keyboard Services og DOS-funktioner. Du kan sikkert finde langt mere udførlige oversigter på internettet, men de er muligvis også mere uoverskuelige. En ting du lige skal bide mærke i, er at en ASCIIZ-streng er en række ASCII-værdier som ender med værdien 0—*ikke* ASCII-værdien for tegnet 0.

A.1 CPU'ens registre

Segmentregistre

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment

Andre registre

AX	(AH,AL)
BX	(BH,BL) (DS:BX)
CX	(CH,CL)
DX	(DH,DL)
SI	Source Index (DS:SI)
DI	Destination Index (DS:DI)
BP	Base Pointer (SS:BP)
SP	Stack Pointer (SS:SP)
IP	Instruction Pointer (CS:IP)

FLAG-registret:

Flag	Navn	Set-symbol i DEBUG	Clear-symbol i DEBUG
OF	Overflow Flag	OV	NV
DF	Direction Flag	DN	UP
IE	Interrupt Enable Flag	EI	DI
TF	Trap Flag	-	-
SF	Sign Flag	NG	PL
ZF	Zero Flag	ZR	NZ
AF	Auxiliary Flag	AC	NA
PF	Parity Flag	PE	PO
CF	Carry Flag	CY	NC

A.2 Udtryk for mængder af hukommelse

Navn	Antal byte		Værdi-mængde	
	Dec	Hex	Dec	Hex
Byte	1	01h	0-255	00h-FFh
Word	2	02h	0-65.535	00h-FFFFh
Double Word	4	04h	0-4.294.967.295	00h-FFFFFFFFh
Quad Word	8	08h	-	-
Paragraph	16	10h	-	-
Page	256	100h	-	-
Segment	65536	10000h	-	-

A.3 BIOS-interrupt 10h: Video Display Services

Gem nummeret på den ønskede service i AH før brug af INT 10h.

VIDEO Service 02h: Position Cursor

Gem et 0 i BH.
Gem X-koordinat i DL.
Gem Y-koordinat i DH.

VIDEO Service 06h: Initialize/Scroll

Gem X-koordinat for øverste venstre hjørne i CL.
Gem Y-koordinat for øverste venstre hjørne i CH.
Gem X-koordinat for nederste højre hjørne i DL.
Gem Y-koordinat for nederste højre hjørne i DH.
Gem antallet af linjer der skal scrolles i AL. Gem 0 for at slette.
Gem display attribute i BH (07h er det normale).

VIDEO Service 1Ah: PS/2 Identify Adapter

Gem et 0 i AL.

Returnerer: AL=1Ah hvis en PS/2 BIOS er til stede (VGA eller MCGA).
Hvis ikke – så er der *ikke* en VGA- eller MCGA-skærm.
BL=Display Adapter Code hvis PS/2 BIOS er til stede.

A.4 DOS-interrupt 16h: Keyboard Services

Gem nummeret på den ønskede service i AH før brug af INT 16h.

Keyboard Service 00h: Get Key From Keyboard Buffer

Hvis der ikke er et tegn i bufferen, venter computeren til der er.
Returnerer SCAN-kode i AH.
Returnerer ASCII-kode i AL.

Keyboard Service 01h: Checks to see if a key is ready to grab

Sætter Zero Flag hvis en taste er klar til grab.
Grab den med Keyboard Service 00h.
Returnerer SCAN-kode i AH.
Returnerer ASCII-kode i AL.

Keyboard Service 02h: Returns shift flags in AL

Bit 7: Insert er aktiv
Bit 6: Caps Lock er aktiv
Bit 5: Num Lock er aktiv
Bit 4: Scroll Lock er aktiv
Bit 3: Alt er nedtrykket
Bit 2: Ctrl er nedtrykket
Bit 1: Venstre Shift er nedtrykket
Bit 0: Højre Shift er nedtrykket

Keyboard Service 03h: Typematic Rate and Delay

Gem 5 i AL.
Gem Delay Value i BH (0-3: 250, 500, 750, 1000 millisekunder).
Gem Typematic Rate i BL (0-1Fh). 1Fh = langsomst (2 tegn/sek), 0 = hurtigst (30 tegn/sek)

Keyboard Service 05h: Stuff Keyboard

Gem SCAN-kode i CH.
Gem ASCII-kode i CL.

Returnerer: 0 = Ingen fejl
 1 = Keyboard-buffer er fuld

Keyboard Service 12h: Returns shift flags in AL and AH

Fungerer ligesom Keyboard Service 02h, men AH har flere informationer:
Bit 7: SysRq er nedtrykket

Bit 6: Caps Lock er aktiv
Bit 5: Num Lock er aktiv
Bit 4: Scroll Lock er aktiv
Bit 3: Højre Alt er aktiv
Bit 2: Højre Ctrl er aktiv
Bit 1: Venstre Alt er aktiv
Bit 0: Venstre Ctrl er aktiv

A.5 DOS-interrupt 21h: DOS-funktioner

Gem den ønskede DOS-funktion i AH før brug af INT 21h.

Funktion 02h: Print Character

Gem tegnets ASCII-kode i DL.

Funktion 09h: Print String

Gem strengens adresse i DS:DX. Strengen skal slutte med et \$ (ASCII 36/24h).

Funktion 3Ch: Create File

Gem fil-attributter i CL:

Bit 0: Read-Only
Bit 1: Hidden
Bit 2: System
Bit 3: Volume Label
Bit 4: Subdirectory
Bit 5: Archive
Bit 6: Reserveret
Bit 7: Reserveret

I DS:DX gemmes en pointer til en ASCIIZ-streng der indeholder filnavnet.

Returnerer: CF=1: Der opstod en fejl. Error Code gemmes i AX (stort set unødvendig).
CF=0: Ingen fejl. File Handle gemmes i AX (skal bruges til at referere til filen).

Funktion 3Dh: Open File

Gem Open Mode i AL:

Bits 2-0: Access Code

000: Read Only Access
001: Write Only Access
010: Read and Write Access

I DS:DX gemmes en pointer til en ASCIIZ-streng der indeholder filnavnet.

Returnerer: CF=1: Der opstod en fejl. Error Code gemmes i AX (stort set unødvendig).
CF=0: Ingen fejl. File Handle gemmes i AX (skal bruges til at referere til filen).

Funktion 3Eh: Close File

Gem File Handle i BX.

Returnerer: CF=1: Der opstod en fejl.

Funktion 3Fh: Read From File

Gem File Handle i BX.

Gem antallet af byte der skal læses i CX.

DS:DX skal indeholde adressen hvor de læste data skal gemmes.

Returnerer: AX = Antallet af byte som blev læst.
Hvis det er 0, så prøvede du at læse fra slutningen af filen.

Funktion 40h: Write To File

Gem File Handle i BX.

Gem antallet af byte der skal skrives i CX.

DS:DX skal indeholde adressen på de data som skal gemmes i filen.

Returnerer: AX = Antallet af byte som blev læst.
Hvis tallet ikke er lig med det antal byte du ville skrive, så er der en fejl.

Funktion 41h: Delete File

I DS:DX gemmes en pointer til en ASCIIZ-streng der indeholder filnavnet.

Returnerer: CF=1: Der opstod en fejl. Error Code gemmes i AX:
AX=2: File Not Found
AX=3: Path Not Found
AX=5: Access Denied
CF=0: Ingen fejl.

Funktion 4Ch: Terminate Process

Gem evt. en ERRORLEVEL-værdi i AL

(det er god skik at gemme et 00h hvis ERRORLEVELS ikke bruges).