Programmering i maskinkode på *AMIGA*

A.Forness & N.A.Holten

Hæfte 3

Indhold

Binær aritmetik

Logiske operatorer

Status–registret

Branching

Bitplanes og Copper

DATASKOLEN
Postboks 62
Nordengen 18
2980 Kokkedal

Telefon 49 18 00 77

Postgiro 7 24 23 44

LOGISKE OPERATIONER

I dette kapitel skal vi se på BINÆR aritmetik (først og frem-mest plus og minus), samt såkaldte logiske operationer (AND, OR, NOT og XOR). Det er noget du vil få meget brug for. Det kan virke lidt indviklet når du ser det første gang - men det er IKKE vanskeligt.

Når vi i decimale talsystemer tager tallet 9 og lægger 1 til, så nulstiller vi talpositionen for enere og forhøjer talpositionen for tiere med 1.

Sådan: 9 + 1 = 10

I ti-tals systemet har vi altså 10 tal (fra 0 til 9) til vor rådighed når vi regner. I det BINÆRE talsystem har vi kun 2 tal: 0 og 1. Men det fungerer på akkurat samme måde.

Lad os se på følgende opsætning:

1 + 1 = 10

I det DECIMALE talsystem strider det som står ovenfor imod al fornuft, men det er helt korrekt i det BINÆRE talsystem. Det har vi været gennem tidligere (se BREV I), men nu skal vi gå lidt mere i dybden.

Følgende regnestykke (BINÆRT) vil blive brugt som forklarende eksempel i dette afsnit.

Eksempel nr. 1:

```
 % 1000 1001        (CARRY = 0)
+% 1010 1010
 ----------
=% 0011 0011        (CARRY = 1)
 ==========
```

Nu skal vi gennemgå dette regnestykke bogstavelig talt BIT for BIT. Den BIT som kaldes LSB (den længst til højre) i det øverste tal (%1000 1001) skal lægges sammen med LSB i tallet nedenunder (%1010 1010). Altså skal vi lægge de to BITs 1 og 0 sammen. Det bliver 1.

Det fungerer nøjagtigt som i det DECIMALE system: 1 + 0 = 1

Så tager vi næste BIT umiddelbart til venstre for LSB i de to tal: 0 og 1. Også her fungerer det akkurat som i det DECIMALE system: 0 + 1 = 1

Næste par BITs er to nuller. Fungerer også som i DECIMALE talsystem: 0 + 0 = 0

Næste to BITs er to enere. Her kommer den funktion ind i bil-ledet som vi nævnte før, når man går fra 9 til 10 i det deci-male talsystem. Man har jo kun to tal som kan bruges

når man regner BINÆRT – 1 og 0. Så når to enere skal adderes får vi et 0 og så 1 i mente. Altså opstår det som vi indledte dette kapitel med: 1 + 1 = 10.

Mente havner i næste position til venstre, således at det bliver 0 i den position vi er i gang med, og så tager vi men-ten med til næste operation.

Næste to BITs er to nuller. Som vist ovenfor er 0 + 0 = 0 i alle talsystemer. Nu er der også en mente, Så det totale reg-nestykke for denne talposition bliver:

0 + 0 + 1 = 1

Næste BIT-par er 0 og 1. Som tidligere bliver resultatet 1 (ingen mente).

I næste position er der to nuller. Facit bliver her 0 (ingen mente)

I BIT-parret længst til venstre MSB, står der to enere. Resultatet bliver altså: 1 + 1 = 10. Her noterer vi derfor nul i denne talposition i svaret og sætter CARRYen til 1 for at markere, at der er en mente. Vi behøver altså 9 BITs for at kunne få angivet svaret korrekt.

Opgave 0301: Læg de BINÆRE tal %10100111 og %10001101 sammen.

Nu burde du være klar til at prøve SUBTRAKTION af BINÆRE tal. Det foregår på samme måde som ved subtraktion af DECIMALE tal. Reglerne er her:

BINÆRT regel 1: 0 – 0 = 0
regel 2: 1 – 0 = 1
regel 3: 1 – 1 = 0
regel 4: 0 – 1 = ? (Tallet bliver NEGATIVT)

Reglerne 1, 2 og 3 burde være indlysende. Det er akkurat som i DECIMALT, som du ser. Vi kan ikke forestille os at du har problemer med det. Vi vender tilbage til regel 4 i et senere brev under et kapitel, som vil behandle SIGNEREDE TAL (negative tal).

AND - OR - NOT - XOR

Nu skal vi til at se på DATA-LOGIK. En matematiker konstrue-rede denne logik før der fandtes datamaskiner, men hans logik blev ikke fuldt ud værdsat før datamaskinerne gjorde deres indtog. Nu kan vi ikke forestille os at undvære denne logik.

Denne logik - som vi starter på om et øjeblik - bruges blandt andet når man skal sætte en BIT i et register uden at forstyrre de andre BITs - eller hvis man vil nulstille en BIT i et register uden at ændre på de andre BITs, som findes i registeret. Vi skal nu forklare dette lidt nærmere.

Vi begynder med **AND.**

Ordet AND betyder OG. Hvis man har to BYTEs og skal "ANDe" dem, så udfører man en bestemt operation for hver BIT i de to BINÆRE tal.

Eksempel nr 2:

```
   % 1011 0101
AND % 1001 1110
   ------------
 = % 1001 0100
   ============
```

Vi sagde at AND betød OG, og nu skal du få at se hvordan det hele fungerer. BITsene længst til venstre i de to BINÆRE tal i EKSEMPEL 2, er to enere. Logikken siger at hvis du ANDer to enere (1 OG 1) så bliver resultatet ALTID 1. Hvis du har noget andet - som 1 og 0 eller 0 og 1, så bliver svaret ALTID 0. Hvis du da sammenligner BIT for BIT i de to BINÆRE tal, så ser du at hver gang man ANDer to enere, så bliver svaret 1 - i alle andre tilfælde bliver svaret nul.

Man kan sætte en tabel op over dette. En sådan tabel for logiske operationer kaldes en sandhedstabel.

Sandhedstabel for AND:

```
0 AND 1 = 0
0 AND 0 = 0
1 AND 0 = 0
1 AND 1 = 1
```

Som du ser får du kun 1 som svar hvis du ANDer to BITs og begge er sat: 1 AND 1 (dansk: 1 OG 1). I alle andre tilfælde får du som sagt nul. Denne funktion er vældig god at kunne når du skal nulstille BITs i et register, hvor forskellige BITs har forskellig be-tydning.

Hvis du for eksempel i et eller andet register skal nulstille BIT nummer 3 (fjerde BIT regnet fra højre - 3,2,1,0), så kan du ANDe registret med følgende BINÆRE tal: %111 0111 - hvis registret på forhånd så således ud: % 0110 0011 - og BIT nummer 3 bliver nulstillet. Det bliver lidt mere overskueligt hvis vi sætter tallene op under hinanden:

```
REGISTRET indeholder: % 0110 1011
Vi AND'er med:        % 1111 0111
                      ----------
Resultat:             % 0110 0011
                      ==========
```

Her ser man tydeligere at fjerde BIT er nulstillet. Man behøver ikke vide hvilken værdi de øvrige BITs i registret har. Det er nok at du kender den BIT, der skal ændres når du ANDer - alle andre BITs lader du være urørt. Funktionen AND kommer du til at benytte ofte i programmeringen på AMIGA.

Opgave 0302: Udfør en AND på %10110110 og % 10001111.

Næste logiske funktion kaldes **OR** - dansk: ELLER. Hermed menes, at når to BINÆRE tal ORes og blot en af to BITs har værdien 1 (egentlig en eller begge to), så bliver svaret 1.

Sandhedstabel for OR (ELLER):

```
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
```

Af sandhedstabellen for OR (ELLER) ser du at, hvis en af to BITs som ORes - eller begge to - er 1, så bliver resultatet af de to BITs: 1.

Eksemplet som kommer her OR'er to BINÆRE tal. Studer det nøje, og sammenlign med sandhedstabellen for OR.

```
    % 1010 1010
OR  % 1100 0101
    ----------
=   % 1110 1111
    ==========
```

I programmering i MC har denne logiske operation den modsatte virkning af AND. Med OR sætter du BITs i et register til 1. Lad os tage det samme eksempel som under forklaringen af AND-funktionen. Registret forudsættes at indeholde %0110 1011. Efter ANDingen indeholdt samme register %0110 0011. Nu vil vi bruge OR-operationen til igen at give BIT 3 (den fjerde BIT fra LSB) værdien 1.

```
REGISTRET indeholder:   % 0110 0011
Vi udfører en OR med:   % 0000 1000
                        ----------
Resultat:               % 0110 1011
                        ==========
```

Den fjerde BIT er atter 1, og vi er tilbage, hvor vi begynd-te. Dette er også en meget anvendelig operation som du vil komme til at bruge mange gange.

Opgave 0303: Udfør en OR på % 01010101 og % 10011001.

Nu er vi kommet til den logiske operator **NOT** (dansk: IKKE). Vi sætter en sandhedstabel op med det samme:

Sandhedstabel for NOT (IKKE):

NOT 0 = 1
NOT 1 = 0

NOT 0 er det samme som at sige: IKKE 0 - og det må jo blive 1. På samme måde bliver NOT 1 (IKKE 1) lig med nul. Med denne logiske operation udfører man en såkaldt konvertering (ven-ding) af BITs. Hvis man skriver: NOT %1010 1010, bliver re-sultatet: %0101 0101. Mere logisk kan det vel ikke blive ?Operationen NOT vil vi blandt andet komme tilbage til i for-bindelse med aflæsninger af tastaturet på AMIGA.

Og så er der den sidste logiske operation: **XOR** (udtales: eks-år). Den kan oversættes med EKSKLUSIV ELLER. Vi sætter en sandhedstabel op først:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

Af tabellen kan man se, at det er **kun** når den **ene** BIT er 1, at resultatet bliver 1, når man udfører en XOR. Vi viser her et eksempel på en udførsel af XOR på to BYTEs:

```
REGISTRET indeholder: % 1010 1010
Vi udfører en XOR:    % 1101 1011
                      ----------
Resultat:             % 0111 0001
                      ==========
```

En gang til: Når to BITer XORes, så bliver resultatet 1 (BITen bliver sat) hvis den ene BIT er sat - og **kun** da. Resultatet af en XOR bliver nul, hvis begge BITs er 0 eller begge BITs er 1. Denne logiske operation bliver sjældent brugt, men vi vil dog få brug for den i et senere brev.

Egentlig er det hele meget logisk, men man skal nok øve sig på det således, at man kan bruge dem uden at måtte tænke over sagen først. Så øv dig grundigt på dette!

Opgave 0304: Udfør en XOR med % 10010010 og % 10100111.

MASKINKODEPROGRAMMERING

Så kaster vi os ud i MC igen. Vi laver et lille program, som indlæser de 16 første hukommelses-celler – adresserne – i AMIGA. Derefter placerer vi dem et andet sted i maskinens hukommelse i en BUFFER, som vi selv har opsat.

Vi viser to eksempler af samme program. Det første er lidt "tungt" for tydelighedens skyld (men fuldt brugbart), og det andet er lidt mere avanceret: kortere, hurtigere og smartere.

Her er første version:

```
1   move.l   #16,d0
2   move.l   #$00,a0
3   lea.l    buffer,a1
4   loop:
5   move.b   (a0),d1
6   add.l    #1,a0
7   move.b   d1,(a1)
8   add.l    #1,a1
9   sub.l    #1,d0
10  cmp.l    #0,d0
11  bne      loop
12  rts
13
14  buffer:
15  blk.b    16,0
```

Linie 1: Flytter det decimale tal 16 ind i d0. Dette register bruges som en tæller – altså en slags regnskab – når vi kræver, at en loop skal gå et vist antal gange (her: 16 gange). Tegnet "#" kan du oversætte med "Tag følgende tal". Altså: Tag følgende tal (16) og placer det i dataregister d0. I linie 6: Adder følgende tal (1); i linie 9: Subtraher følgende tal (1), og i linie 10: Sammenlign følgende tal (0) med det som findes i dataregister d0.

Linie 2: Lægger tallet $00 ind i adresseregister a0. I og med at vi bruger LONGWORD, nulstiller vi dermed alle BITs (32 stykker). Dette adresseregister bruger vi som en pointer (den pointer altså i dette tilfælde på adresse $000000): den adresse vi vil læse **fra**.

Linie 3: Lægger adressen på vores egen BUFFER (se linie 14) ind i a1. Eller med andre ord: Vi har sat en del af AMIGAens hukommelse af til lagringsplads for det vi indlæser. Denne lagringsplads begynder på den adresse i maskinen, hvor vores LABEL befinder sig (LABEL-navnet assembles jo ikke til enere og nuller, som du ved. Det bliver derfor den **første** BYTEs adresse, som repræsenterer begyndelsen på BUFFEREN).

Adresseregisteret a1 bruges altså som en pointer til det sted (den adresse), hvor vi ønsker at vore data skal lagres. Læg mærke til at vi ikke forandrer noget i de adresser vi læser. Dataene bliver bare **kopieret** ind i vores BUFFER.

Linie 4: En LABEL

Linie 5: Flytter tallet som befinder sig i den BYTE (eller bedre: adresse), som adresseregistret a0 peger på, ind i d1. Parantesen rundt om a0 betyder, at man ikke skal bruge de data, som adresseregistret a0 måtte indeholde fra før, men at man holder sig til den adresse, som a0 indeholder (peger på) og henter de data som ligger **der**. Denne måde at adressere på kaldes for: REGISTER INDIRECT (oversat: indirekte register-adressering) og benyttes meget ofte i MC på AMIGA.

Linie 6: Adder 1 til det tal som findes i a0. Det resulterer i at a0 vil pege på den **næste adresse** vi vil læse fra.

Linie 7: Flytter tallet, som er sig i d1, ind på den adresse, som adresseregister a1 peger på. Se kommentarer til LINIE 5.

Linie 8: Adderer 1 til det tal, som findes i a1 (forhøjer altså a1 med en). Dette resulterer i at a1 nu vil pege på den adresse i hukommelsen (i vores buffer), hvor vi skal lagre næste BYTE.

Linie 9: Træk tallet 1 fra d0. Dette er vores tæller, som bliver formindsket med **en**, hver gang vi har læst en BYTE - ialt 16 gange.

Linie 10: Check om d0 er blevet 0. CMP (compare) betyder "sammenlign" og bruges her for at se om vi **har** læst 16 BYTEs (for så er dataregister d0 blevet 0).

Linie 11: Hvis d0 **ikke er lig med** 0, hopper programmet tilbage til vores LABEL "loop:" (linie 4). Hvis d0 **er lig med 0**, er LOOPen udført 16 gange, og programmet vil ikke hoppe tilbage. I stedet vil programmet starte på næste instruktion.

Linie 12: RETURNER FRA SUBRUTINE (eller afslut), og giv kontrollen tilbage til K-SEKA.

Linie 14: LABELen som peger på begyndelsen af vores egen BUFFER.

Linie 15: Her har vi deklareret (sat plads af til) 16 BYTEs til vores buffer. Vi kunne også have skrevet:

blk.l 4,0 eller
blk.w 8,0.

det havde givet samme resultat. BLK står for "blok", og 4,0 (og 8,0) viser antal LONGWORDs (eller antal WORDs), som vi ønsker at reservere som BUFFER. Tallet "0" bliver lagt ind i alle BYTEs i BUFFERen, således at vi samtidig får BUFFERen "renset" (eller tømt – i tilfælde af at der skulle ligge gamle data fra tidligere). Se også kommende afsnit.

Lad os se på nogen nye instruktioner:

CMP	(sammenlign)
BNE,BEQ,BHL,BLO	(forskellige "BRANCHes" dvs afgreninger)
BLK,DC	(BLOK– og DECLARE–kommandoer)

Vi vil først se på STATUS–registret før vi går igennem instruktionerne.

STATUSREGISTRET

	MSB				LSB
INDHOLD:	X	N	Z	V	C

Dette er en del af det såkaldte STATUSREGISTER i MC–68000. Vi kommer ikke til at læse eller skrive direkte i dette regis–ter. Du behøver heller ikke at vide, hvilke BIT–numre de forskellige FLAG ligger på (ordet FLAG forklares nedenfor). Alt hvad du behøver at vide er hvilke FLAG, som findes, og hvad de bruges til.

FLAG er et nyt udtryk, som vi skal se lidt nærmere på. Her er først deres navne:

C	–	**C**arry flag	(mente)
V	–	o**V**erflow flag	(advarsel, tallet er blevet for stort)
Z	–	**Z**ero flag	(noget er blevet nul)
N	–	**N**egative flag	(tallet er negativt)
X	–	e**X**tend flag	(speciel mente)

At bruge en BIT til at vise, hvis en bestemt tilstand er opnået, kaldes at benytte sig af BITen som flag. Når FLAGET er 1, siger man at det er SAT.

Forestil dig en længdesprings–konkurrence. Ved stregen, hvor springeren sætter af, står der en og vifter med et rødt flag, hver gang der er overtrådt – og hvidt hvis springet er

gyldigt. Samme princip gælder for FLAGENE i STATUSREGISTERET. De fortæller den, som programmerer, for eksempel når et tal er negativt (**N**-flaget), når et resultat er blevet nul (**Z**-flaget), når et tal er blevet for stort (**C**-flaget og **V**-flaget) og så videre. Vi forklarer hvert flag mere indgående nedenfor.

I næste eksempel bliver CARRY-flaget 1 fordi summen af de to tal (200 og 100) overstiger det tal, en BYTE kan indeholde (255).

```
move.b#200,d0
add.b #100,d0
rts
```

Som du ved kan et 8-BITs tal, eller en BYTE kun indeholde tal fra 0 til og med 255. I eksemplet ovenfor lægger vi 200 ind i d0, og tillægger så 100. Der burde altså ligge 300 i vores BYTE. Da et så stort tal ikke kan få plads der, bliver CARRY-flaget sat (til 1) for at fortælle os at tallet er blevet for stort.

```
    11001000   (200 deciamlt)
+   01100100   (100 decimalt)
    --------
= 1 00101100   (300 decimalt)
  ==========
```

Som du ser må vi tage en niende BIT i brug for at få rigtigt svar. Det er denne BIT (længst til venstre i svaret), som havner i CARRY-flaget (CARRY-BITen). Hvis resultatet bliver 255 eller mindre nulstilles CARRY-flaget i stedet.

OVERFLOW-flaget ligner CARRY-flaget meget. Forskellen er at det bruges som den "niende BIT" i 2-KOMPLIMENT-TAL eller SIGNEREDE tal. Mere om dette i et senere brev.

ZERO-flaget bruges tit og i mange forskellige sammenhænge. Hver gang en operation resulterer i et nul-resultat, bliver dette flag sat. I det første eksempel i dette kapitel står der i linie 10: cmp.l #0,d0. Denne linie kunne godt fjernes fordi, så snart linie 9 resulterer i nul i dataregister d0, sættes ZERO-flaget automatisk. F.eks. betyder instruktionen:

```
cmp.l  #100,d0
```

...tag følgende tal (100) og sammenlign det med det, som er i dataregister d0. Hvis d0 indeholder 100, resulterer sammen-ligningen i en lighed, og ZERO-flaget sættes.

NEGATIVE-flaget bliver sat for at vise, at resultatet er negativt (minus). Vi vil beskæftige os med dette senere.

EXTEND-flaget bliver brugt ved rotering eller skiftning af BITs. Flaget bliver også brugt i komplicerede adderings- eller multiplikations-rutiner og fungerer da som et CARRY-flag. Vi kommer tilbage til dette flag senere.

Instruktionen CMP for "COMPARE" (sammenlign) bruges til at sammenligne tal. For at den skal have nogen nytte må den kombineres med en BRANCH. Vi tager nogen eksempler:

```
move.l#100,d0
cmp.l #50,d0   (næste linie er en af følgende BRANCHer):
```

BEQ	hop når tallet er lig med	(BRANCH ON EQUAL)
BNE	hop når tallet er ulig med	(BRANCH ON NOT EQUAL)
BHI	hop når tallet er højere	(BRANCH ON HIGHER)
BLO	hop når tallet er lavere	(BRANCH ON LOWER)

BEQ-instruktionen (**B**ranch on **EQ**ual) vil ikke udføre et hop fordi d0 i vores lille eksempel (som indeholder 100) ikke er lig med 50.

BNE-instruktionen (**B**ranch on **N**ot **E**qual) udfører et hop, fordi d0 er ulig med 50.

BHI-instruktionen (**B**ranch on **HI**gher) udfører et hop fordi d0 er "højere" end 50. Her er tallet 50 brugt som reference. Du skal altså tyde instruktionen "bagvendt" for, at det bliver rigtigt, altså: "Er d0 større end 50

BLO-instruktionen (**B**ranch on **LO**wer) udfører ikke et hop, fordi d0 ikke er "lavere" end 50.

Det er meget vigtig at du forstå dette godt. Se også i programeksemplet på linie 11. Vi kunne der have skrevet BHI i stedet for BNE.

Nu skal vi se lidt på BLK- og DC-instruktionen. BLK-instruktionen (**BL**oc**K**) bruges til at reservere et område i hukommelsen hvori vi kan behandle eller lagre data. BLK er ingen instruktion, og assembles ikke til enere og nuller.
Lad os se lidt på, hvordan man bruger den.

```
1 blk.b  10.0
2 blk.b  $10,0
3 blk.l  100,0
4 blk.b  $2800,255
5 blk.w  $2800,255
```

Linie 1: Reserverer 10 BYTEs og nulstiller dem alle.
Sådan: $00 00 00 00 00 00 00 00 00 00

Linie 2: Reserverer $10 ($10 = 16) BYTEs og nulstiller dem.
Sådan: $00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Linie 3: Reserverer 100 LONGWORDs (ialt 400 BYTEs) og lægger 0 (nulstiller) i alle LONGWORDene.
Sådan: $00000000 00000000 00000000 00000000....(100 gange)

Linie 4: Reserverer 2800 BYTEs (decimalt=10240) og lægger 255 i alle BYTEs. (255 DECIMALT er lig med $FF HEXADECIMALT)
Sådan: $FF FF FF FF FF FF FF FF FF FF FF FF...(10240 gange)

Linie 5: Reserverer $2800 WORDs og lægge 255 i alle WORDene
Sådan: $00FF 00FF 00FF 00FF 00FF 00FF 00FF... (10240 gange)

DC-instruktionen (DECLARE, oversat deklarere) har samme funktion som BLK, men reserverer kun **en** BYTE af, **et** WORD
ller **et** LONGWORD.

```
1 dc.b  0
2 dc.b  255,0,10
3 dc.b  "AMIGA"
4 dc.w  $100
5 dc.l  123456
```

Linie 1: Reserverer en byte og sætter den til 0.
Sådan: $00

Linie 2: Det er muligt at have flere tal på samme linie. Så reserveres der lige så mange BYTEs (vi bruger kommandoen DC.B), som der er tal (her 3). Her sættes de til 255 (=$FF), 0 og 10 (=$0A).
Sådan: $FF 00 0A

Linie 3: Denne linie er lidt speciel. Du ved sikkert at **et** tegn bruger **en** byte. Når tegn lagres i datamaskiner, bliver de lagret som tal. Vi har tidligere nævnt at bogstavet A lagres som tallet 65 ($41). Bagest i din Amiga's manual er en tabel, som viser hvilke tal, der repræsenterer hvilke bogstaver. Denne tabel kaldes for en ASCII-tabel (udtales: ASKI). Forkortelsen står for: American Standard Code for Information Interchange, oversat: Amerikansk kode for udveksling af information. Denne data-standard for at lagre information bruges i alle datamaskiner idag. Via ASCII-tabellen kan du læse ordet AMIGA som HEXADECIMALE tal.
Sådan: $41 4D 49 47 41

Linie 4: Reserverer et WORD, og sætter det til $100 (=256)
Sådan: $0100

Linie 5: Reserverer et LONGWORD, og sætter det til 123456 (=$1E240).
Sådan: $0001E240

Forstod du det hele? hvis du er det mindste i tvivl så læs det en gang til. Det er uhyre vigtigt at du forstår det. Det skal være en af grundstenene i din viden om programmering.

Og så den anden version:

```
1  move.l   #15,d0
2  move.l   #$00,a0
3  lea.l    buffer,a1
4  loop:
5  move.b   (a0)+,(a1)+
6  dbra     d0,loop
7  rts
8
9  buffer:
10 blk.b    16,0
```

Linie 1: Lægger tallet 15 ind i d0. Her bruges d0 igen som tæller.

Linie 2: Lægger tallet $00 ind i a0.

Linie 3: Lægger adressen på vores buffer ind i a1.

Linie 4: En LABEL.

Linie 5: Erstatter linierne 5,6,7 og 8 i det andet eksempel. Vi forklarer instruktionen nærmere nedenfor.

Linie 6: Erstatter linierne 9, 10 og 11 i det andet eksempel; også denne instuktion forklares nedenfor.

Linie 7: Afslutter programmet.

Linie 9: LABELen til vores BUFFER.

Linie 10: Reserverer 16 BYTEs og nulstiller dem. Den udgave af MOVE–instruktionen, du ser i linie 5 bruges meget ofte, så lad os tage fat på den først:

Du har lært, at hvis du sætter parenteser om et adresse–register, som i linie 5 i det første eksempel, så peger adresseregistret på den adresse, der skal bruges. Denne nye version virker på samme måde, men har en ekstra finesse. Se på dette eksempel:

```
move.b   (a0)+,d0
```

Den udfører det samme som:

```
move.b   (a0),d0
add.l    #1,a0
```

Altså: Tallet, som ligger i a0, bruges som adresse. Derefter går den til den adresse og henter det tal, der ligger der, og lægger det ind i dataregister d0.

Derefter adderer den automatisk 1 til a0, fordi der står et plustegn efter parantesen. Havde der stået "move.w" i stedet for "move.b" ville den have adderet 2 til a0 (fordi et WORD består af to BYTEs). Havde der stået "move.l" i stedet for "move.b" ville den have adderet 4 til a0 (fordi et LONGWORD består af fire BYTEs).

Instruktionen MOVE.B (a0)+,(a1)+ vil:

1: Hente tallet på adressen a0 peger på, og lagre det på den adresse som a1 peger på.

2: Adderer 1 til a0 (Vi arbejder med størrelsen BYTE).

3: Adderer 1 til a1 (Vi arbejder med størrelsen BYTE).

Du kommer til at bruge denne adresseringsmåde ofte. Specielt når du behandler data-tabeller, og når du flytter rundt på data i AMIGAs hukommelse. Det er derfor **meget** vigtigt at du lærer disse ting.

Den næste instruktion vi skal se på er DBRA.:

```
move.l   #9,d0
loop:
dbra     d0,loop
rts
```

Eksemplet udfører det samme som:

```
move.l   #10,d0
loop:
sub.l    #1,d0
cmp.l    #0,d0
bne      loop
rts
```

Det DBRA gør er at:

1: Trække 1 fra i registret (I dette tilfælde d0).

2: Sammenligne med -1.

3: Er det -1? Hvis ikke, hop tilbage.

(Hvordan et tal kan være negativt i en BYTE, et WORD eller et LONGWORD kommer vi tilbage til. I øjeblikket behøver du ikke spekulere på det problem).

Når d0 er blevet -1 vil den fortsætte på næste instruktion. Husk at DBRA hele tiden sammenligner med -1, så vi skal angive et tal i tælleren (d0), som er 1 mindre en det antal gange LOOPen skal udføres.

COPPER OG BITPLANES

Nu kaster vi os ud på det store ocean, som kaldes BITPLANEs (udtales bittpleins og kan oversættes med "BIT-flader"). Vi forklarer mere om COPPERen samtidig og det vil medføre en del nødvendig hoppen frem og tilbage i emnet.

Det kan betale sig først at gennemlæse hele kapitlet en gang for at få overblik over stoffet, og derefter læse det hele forfra igen.

Vi begynder med at forklare lidt om et HARDWARE-register som i datalitteraturen om AMIGA har fået navnet "DMACON". Navnet er en forkortelse af "DMA-CONTROL".

DMACON er navnet på et specielt WORD (en 2 BYTEs adresse) som betegnes REGISTER. HARDWARE-registre ikke har noget med d0, a0 eller andre af registrene i MC-68000 at gøre. HARDWARE-registrene ligger i en helt anden del af maskinen, nemlig i AGNUS, PAULA og DENISE. Registrene har derfor en adresse, og adressen til DMACON er $DFF096. Alle adresser på HARDWARE-registre begynder med $DFF??? Vi kalder $DFF000 for HARDWARE-registrenes BASE-ADRESSE.

NB! Alle HARDWARE-registre er 16 BITs, altså et WORD.

Det er vigtigt at lægge mærke til, at HARDWARE-registrene er opbygget således at de enten kun kan **skrives til**, eller kun **læses fra**. F.eks kan der kun skrives til farveregistrene. Hvis du prøver at læse dem, vil du få returneret enten nul eller et andet vilkårligt tal.

JOYSTICK-registrene er den anden type, den der kun kan læses fra. At skrive til disse har ingen betydning. (Du kan altså ikke bede din JOYSTICK ryge og rejse!) Nogle registre har **to** adresser - en skrive-adresse og en læse-adresse. Det er sådan et register (DMACON) vi nu skal gennemgå. Hvis du skal skrive til DMACON skal du bruge adresse $DFF096, men hvis du skal læse fra det, skal du bruge adresse $DFF002.

DMACON er opsat således (Husk! 1 WORD = 16 BITs):

$DFF096 WRITE - $DFF002 READ

BIT NR.	FUNKTION	ACCES
15	SET/CLR	W
14	BLITTER BUSY	R
13	BLITTER ZERO	R
12	-	
11	-	
10	BLITTER NASTY	R/W
9	DMA ENABLE	R/W
8	BITPLANE DMA	R/W
7	COPPER DMA	R/W
6	BLITTER DMA	R/W
5	SPRITE DMA	R/W
4	DISK DMA	R/W
3	AUDIO CHANNEL 3 DMA	R/W
2	AUDIO CHANNEL 2 DMA	R/W
1	AUDIO CHANNEL 1 DMA	R/W
0	AUDIO CHANNEL 0 DMA	R/W

Vi har, som du ser, valgt at bruge de engelske betegnelser. Det er der to grunde til: For det første så kan du lettere referere til (og forstå) special-litteratur på engelsk, når du kender de engelske udtryk, og for det andet så bliver oversættelserne meget dårlige, da der ofte ikke findes et tilsvarende ord i dansk-dataterminologi.

Læg mærke til at vi i sidste kolonne (ACCES, oversat: tilgang til) angiver, hvilke BITs, som kan læses ("R" for READ), og hvilke BITs, der kan skrives ("W" for WRITE) til.

NB! hvis en DMA skal være OFF (eller nulstillet), sættes den til 0. Hvis en DMA derimod skal være ON (eller aktiveres), sættes den til 1.

BIT 0-3: DMA til lydkanal 0 til 3 on/off

Bit 4: DISK-DMA on/off

BIT 5: SPRITE-DMA on/off

BIT 6: BLITTER-DMA on/off

BIT 7: COPPER-DMA on/off

BIT 8: BITPLANE-DMA on/off

BIT 9: Denne BIT kaldes for DMA ENABLE (oversat: aktiver DMA) og skal være sat (1) for at DMA'er skal kunne benyttes (kan sammenlignes med en hoved-afbryder). Hvis du af en eller anden grund vil slukke **alle** DMA'er, så nulstiller du bare denne BIT. Du bør også være opmærksom på, hvis DISK-DMAen

og SPRITE-DMAen er on, når du slukker for DMAen, så vil det kun være disse to DMAer, som aktiveres, når du tænder dem igen.

BIT 10: BLITTER NASTY. Mere om denne BIT i brev 6 og 7.

BIT 11-12: Benyttes ikke.

BIT 13: BLITTER ZERO. Mere om denne BIT i brev 6 og 7.

BIT 14: BLITTER BUSY. Mere om denne BIT i brev 6 og 7

BIT 15: SET/CLEAR. Dette vil blive forklaret nedenfor.

Registret DMACON er lidt specielt når det gælder skrivning. F.eks. hvis du skal aktivere COPPER-DMA, skal du lægge disse data ind i registret på adresse $DFF096:

BIT 7 og 15 skal være "1", resten skal være "0".
HEXADECIMALT: $8080,
BINÆRT: %1000 0000 1000 0000.
(Hvis du har glemt omregning mellem HEXADECIMALT
og BINÆRT, så læs brev I igen).

Når COPPEREN skal slukkes skal BIT 7 være "1", resten skal være "0".
HEXADECIMALT: $0080,
BINÆRT: %0000 0000 1000 0000.

Når du vil slukke for f.eks. BITPLANE-DMAen og SPRITE-DMAen, skal BIT 15 være 0, BIT 5 og 8 skal være 1 og resten være 0. HEXADECIMALT bliver det $0120, BINÆRT %0000 0001 0010 0000, eller som en instruktion:

```
move.w  #$0120,$DFF096
```

Hvis du nu vil aktivisere de samme DMAer igen, ændrer du bare BIT 15 til 1. Altså:

```
move.w  #$8120,$DFF096
```

(eller move.w #%1000000100100000,$DFF096)

Vi vil gerne forklare det en gang til. Det er nemlig sådan, at der findes 4 registre (8 hvis du regner med både READ og WRITE registrene), som fungerer på denne måde, og det er derfor vigtigt, du forstå dette helt:

Hvis du sætter BIT 15 til "1" og viser hvilke BITs, der skal påvirkes, så vil AMIGA selv sørge for, at de BITs du har valgt sættes til "1", også. Hvis du derimod sætter BIT 15 til "0", så vil den sætte de af dig valgte BITs til "0".

En gang til – men med andre ord: Alle de BITs, som du sætter til "1" får den værdi, som BIT 15 har. Alle andre BITs (som er "0") forbliver uforandrede.

Enkelt og let, ikke sandt?

NB NB NB NB NB !!!!!!

Hvis du skal åbne for en DMAer og slukke for en anden DMA samtidig, skal du skrive til registret to gange – én gang for at lukke for DISK DMA'en, og én gang for at åbne for BLITTER–DMAen på (eksempelvis).

Forklaring til COPPER-instruktionerne:

Studer følgende opsætninger af de to WORDs, som en komplet COPPER-instruktion består af:

Første WORD:

BIT NR:	MOVE	WAIT
15	A15	V7
14	A14	V6
13	A13	V5
12	A12	V4
11	A11	V3
10	A10	V2
9	A9	V1
8	A8	V0
7	A7	H8
6	A6	H7
5	A5	H6
4	A4	H5
3	A3	H4
2	A2	H3
1	A1	H2
0	0	1

Andet WORD:

BIT NR:	MOVE	WAIT
15	D15	BFD
14	D14	MV6
13	D13	MV5
12	D12	MV4
11	D11	MV3
10	D10	MV2
9	D9	MV1
8	D8	MV0
7	D7	MH8
6	D6	MH7
5	D5	MH6
4	D4	MH5
3	D3	MH4
2	D2	MH3
1	D1	MH2
0	D0	0

En Copper-instruktion er **altid 4** BYTEs lang (2 WORDs). Vi plejer at dele den i to dele, som vist her:

dc.w $5001,$FFFE (dc.w = DECLARE WORD)

Lad os gå igang med WAIT-instruktionen:

I vores forklaring vil vi bruge "VV" for at beskrive vertikal venteposition. Med det mener vi, at COPPERen får besked på, at elektronstrålen skal komme til den linie du angiver, før den (COPPERen) henter næste instruktion.

"HH" bruges for at beskrive horisontal venteposition for COPPERen. Med det mener vi, at COPPERen får besked på, at elektronstrålen skal komme til et bestemt punkt på den linie du angiver, før den (COPPERen) henter næste instruktion.

VVHH,$FFFE

VV=$00 til $FF
HH=$01 til DF (kun ulige tal kan benyttes)

Vi kommer til at bruge værdien $01 som horisontal position i alle WAIT-instruktioner. (Det er punktet længst til venstre på linien.) Venten på horisontale positioner større end $01 vil blive forklaret i brev 12.

Når det drejer sig om den vertikale position gælder følgende:

Den øverste, hvide linie, du ser på f.eks din WORKBENCH-skærm er **ikke** linie 0. Det er som regel linie $2C (44 DECIMALT). Det vil sige at elektronstrålen reelt begynder 44 linier længere oppe end den øverste linie, du kan se. Det vil igen sige at den nederste linie på WORKBENCH-skærmen bliver nummer 256 + 44 = 300 (WORKBENCH-skærmen er 256 linier høj). Under disse 300 linier ligger yderligere 13 linier - som du delvis ikke kan se - så den totale størrelse på en fuld AMIGA-skærm er 313 linier. COPPERen har derfor et totalt venteområde fra linie 0 til og med linie 312.

Det andet WORD i WAIT-instruktionen har man sjældent behov for at ændre. Denne del vil blive forklaret senere. Vi kommer til at bruge samme værdi hele tiden, nemlig: $FFFE

FORKLARING AF MOVE-INSTRUKTIONEN

MOVE-instruktionen kan kun skrive data til HARDWARE-registrene i området $DFF000 til DFF200.

Eksempel: dc.w $180,$0FFF

Det første WORD i MOVE-instruktionen skal indeholde **adressen**. Det andet WORD i MOVE skal indeholde **de data** du lægger ind på adressen.

Adressen regnes således ud:

$0180 + $DFF000 = $DFF180.

Denne adresse skal være et lige tal – næste adresse bliver derfor $DFF182 (en afstand på et WORD). Hvis du derimod skriver sådan (med ulige-tals-adresse):

dc.w $0181,$0FFF

vil COPPERen tolke den som en WAIT-instruktion i stedet. COPPERen benytter BIT 0 i det første WORD til at skille en WAIT- og en MOVE-instruktion fra hinanden.

Den tredie COPPER-instruktion hedder SKIP. Denne instruktion har du praktisk taget aldrig brug for, så vi vil ikke spilde tid på den nu. Den vil blive forklaret i et senere brev.

Forklaring til eksemplet i BREV II (side 17):

Linie 1: Sluk BITPLANE-, COPPER- og SPRITE-DMAene.

Linie 2: Lægger adressen til COPPER-listen ind i a1.

Linie 3: Lægger indholdet af a1 ind i $DFF080. Vi må lægge adressen på vores egen COPPER-liste ind i dette register, for at COPPERen skal vide, hvor COPPER-listen befinder sig. Som du ser bruger vi i denne linie LONGWORD, når vi kopierer adressen til a1, selv om vi har sagt at du kun kan bruge WORD, når du adresserer HARDWARE-registrene. Dette er en undtagelse. Egentlig er COPPER-pointeren to adresser:

$DFF080 (HI) og
$DFF082 (LO)

Betegnelserne HI og LO står for HØJ (engelsk HIGH) adresse, og LAV (engelsk: LOW) adresse. Som du ved er et LONGWORD det samme som to WORDs. BIT 31 til 16 havner således i $DFF080 og repræsenterer de 16 højeste (mest "værdifulde") BITs i den komplette adresse, og BIT 15 til 0 havner i $DFF082 og repræsenterer de 16 laveste (mindst "værdifulde") BITs i den komplette adresse.

Linie 4: Aktiviser COPPERen igen. Denne instruktion starter vores egen COPPER-liste.

Linie 6: En LABEL.

Linie 7: Denne instruktion har vi ikke gennemgået endnu. Det den gør, er at checke BIT 6 (btst = **BitTeST**) i adressen $BFE001. Hvis BITen er 0 vil den sætte ZERO-flaget til 1. Er BITen 1, sættes ZERO-flaget til 0. Når du trykker på venstre mus-tast, bliver denne BIT i $BFE001 sat til 0.

Linie 8: Denne BRANCH undersøger om ZERO-flaget er 0, og hvis den er det, så har du ikke trykket på venstre mus-tast, og programmet hopper tilbage til reference-punktet "wait:".
Programlinierne 7 og 8 er en vente-funktion, til du har trykket på musen.

Linie 10: Sluk COPPER-DMAen.

Linie 11: Lægger indholdet af adresse $000004 ind i a6. (linierne 11 til 13 bliver nærmere forklaret i BREV X. Man kan kort sige at de henter WORKBENCH-skærmen tilbage).

Linie 12: Lægger det, som findes på adressen a6 + 156 ind i a1.

Linie 13: Lægger det, som findes på adressen a1 + 38 ind i COPPER-POINTERen.

Linie 14: Åbner COPPER-DMAen igen.

Linie 15: Afslutter programmet.

Linie 17: LABEL til COPPER-listen.

Linie 18: WAIT ($01,$90)

Linie 19: MOVE $0F00 -> $DFF180

Linie 20: WAIT ($01,$A0)

Linie 21: MOVE $0FFF-> $DFF180

Linie 22: WAIT ($01,$A4)

Linie 23: MOVE $000F -> $DFF180

Linie 24: WAIT ($01,$AA)

Linie 25: MOVE $0FFF -> $DFF180

Linie 26: WAIT ($01,AE)

Linie 27: MOVE $0F00 -> $DFF180

Linie 28: WAIT ($01,$BE)

Linie 29: MOVE $0000 -> $DFF180

Meget mere og endnu mere udførligt - vil blive forklaret i BREV IV.

Vi begynder med at sætte en skærm op:

```
1  move.w   #$01a0,$dff096
2
3  move.w   #$1200,$dff100
4  move.w   #0,$dff102
5  move.w   #0,$dff104
6  move.w   #0,$dff108
7  move.w   #0,$dff10a
8
9  move.w   #$2c81,$dff08e
10 move.w   #$f4c1,$dff090
11 move.w   #$38c1,$dff090
12
13 move.w   #$0038,$dff092
14 move.w   #$00d0,$dff094
15
16 lea.l    copper,a1
17 move.l   a1,$dff080
18
19 move.w   #$8180,$dff096
20
21 wait:
22 btst     #6,$bfe001
23 bne      wait
24
25 move.w   #$0080,$dff096
26
27 move.l   $4,a6
28 move.l   156(a6),a1
29 move.l   38(a1),$dff080
30
31 move.w   #$80a0,$dff096
32
33 rts
34
35 copper:
36 dc.w     $2c01,$fffe
37 dc.w     $0100,$1200
38
39 dc.w     $00e0,$0000
40 dc.w     $00e2,$0000
41
42 dc.w     $0180,$0000
43 dc.w     $0182,$0ff0
44
45 dc.w     $ffdf,$fffe
46
47 dc.w     $2c01,$fffe
48
49 dc.w     $0100,$0200
50
51 dc.w     $ffff,$fffe
```

Her følger en kort forklaring til hver instruktion. I BREV IV vil alt blive mere udførligt forklaret. Tag det derfor ikke så tungt, hvis du ikke forstår alt i første forsøg.

Linie 1: Luk for BITPLANE-, COPER- og SPRITE-DMAerne.

Linie 3: Sætter LORES - lav opløsning - (320 * 256), og et BITPLANE, altså 2 farver.

Linie 4: Sætter SCROLL-værdi til 0.

Linie 5: Sætter BITPLANE PRIORITY til 0

Linie 6: Sætter MODULO for ODD BITPLANES til 0.

Linie 7: Sætter MODULO for EVEN BITPLANES til 0.

Linie 9: Sætter øverste, venstre hjørne på skærmen til position: Y (vertikalt) = $2C og X (horisontalt) = $81.

Linie 10: Sætter nederste højre hjørne på skærmen til position: Y (horisontalt) = $F4 og X (vertikalt) = $C1.

Linie 11: Lægger $38 til Y-positionen, altså $fe + $38 = $12C. X-positionen bliver den samme.

Linie 13: Sætter DATAFETCH START til $0038.

Linie 14: Sætter DATAFETCH STOP til $00d0.

Linie 16: Lægger copper-listen's adresse ind i a1.

Linie 17: Lægger værdien, som er i a1 ind i COPPER-POINTERen.

Linie 19: Aktiviserer BITPLANE- og COPPER-DMAerne igen (ikke SPRITE-DMAen)

Linie 21: EN LABEL.

Linie 22: Kontrollerer om venstre mustast trykkes ned.

Linie 23: Hvis der ikke er trykket på mustasten, så hop tilbage til "wait:".

Linie 25: Slukker COPPER-DMAen.

Linie 27: Henter (Husk egentlig er det: kopierer) tallet, som ligger i adresse $000004 ind i a6.

Linie 28: Henter tallet, som ligger på den adresse som a6 indeholder + 156, ind i a1.

Linie 29: Henter tallet, som ligger på den adresse som a1 indeholder + 38 ind i COPPER-POINTERen ($DFF080).

Linie 31: Åbner COPPER- og SPRITE-DMAen igen.

Linie 33: Afslutter.

Linie 35: LABELen til COPPER-listen.

Linie 36: WAIT ($01,$2C)

Linie 37: MOVE $1200 -> $DFF100

Linie 39: MOVE $0000 -> $DFF0E0

Linie 40: MOVE $0000 -> $DFF0E2

Linie 42: MOVE $0000 -> $DFF180 (sort baggrund)

Linie 43: MOVE $0FF0 -> $DFF182 (gul forgrund)

Linie 45: WAIT ($DF,$FF). Dette medfører at maskinen slår om til PAL AMIGA med 256 linier. I USA, hvor man har et andet system (NTSC), har AMIGA kun 200 linier).

Linie 47: WAIT ($01,$2C). Den vertikale position ($2C) vil her svare til $251252C, fordi vi udførte PAL-WAIT på linien ovenfor.

Linie 49: MOVE $0200 -> $DFF100

Linie 51: Stopper COPPERen og hopper tilbage til begyndelsen af COPPER-listen (udfører den endnu engang).

LØSNINGER TIL BREV II

Opgave 0201: Fordi der findes 8 SPRITE DMAer, 4 AUDIO DMAer, 4 BLITTER DMA, 1 DISK DMA, 6 BITPLANE DMAer og 1 COPPER DMA.

Opgave 0202: En SUBRUTINE er en delrutine eller et underprogram, som udfører en speciel opgave.

Opgave 0203: I et WORD er der to BYTEs lig med 16 BITs.

Opgave 0204: I et LONGWORD er der 4 BYTEs, som er lig med 32 BITs.

Opgave 0205: I MC-68000 findes der 8 ADRESSEREGISTRE, og de hedder A0, A1, A2, A3, A4, A5, A6 og A7.

Opgave 0206: I MC-68000 findes der 8 DATAREGISTRE, og de hedder D0, D1, D2, D3, D4, D5, D6 og D7.

Opgave 0207: Instruktionen lægger det, som findes i adresse $10 ind i ADRESSE-REGISTER 3.

Opgave 0208: MSB betyder MOST SIGNIFICANT BIT, og angiver den højeste værdi-BIT i en BIT-gruppe. LSB betyder LEAST SIGNIFICANT BIT, og angiver den laveste værdi-BIT i en BIT-gruppe.

Opgave 0209: COPPERen har 3 instruktioner og de hedder MOVE, WAIT og SKIP.

Opgave 0210: En elektronkanon er den stråle, som tegner billedet på skærmen.

KOMMENTARER TIL BREV III

Dette er et langt brev! Her er meget at gennemgå. Det er heller ikke alt som er lige let, men med lidt tålmodighed kommer du langt.

Når man skriver et brevkursus i et så omfattende emne som AMIGA, så sker det at vi må hoppe lidt frem og tilbage i forklaringerne. Skulle der være noget du ikke forstår, eller som du synes er noget rodet, så vent til du modtager BREV IV, det forklarer meget af det, som måske er lidt "tåget" i øjeblikket.

Det er meget vigtigt at du bruger K-SEKA (eller en anden assembler) og øver dig. Skriv alle eksempler ind og prøv at ændre på dem. Hvis programmet låser sig fast, så prøv igen. Det er utrolig så meget man lærer af sine fejl.

Hvad du end finder på - GIV IKKE OP !

Med venlig hilsen
DATASKOLEN

Carsten Nordenhof