# Programmering i maskinkode på *AMIGA*

**A.Forness & N.A.Holten**


## Hæfte 4

**Indhold**

Maskinkode III

Bitmap

Farveregistre

Bitplane DMA tidsforbrug

Overscan

**DATASKOLEN**
Postboks 62
Nordengen 18
2980 Kokkedal

Telefon 49 18 00 77

Postgiro 7 24 23 44

## MASKINKODE III

Så skal vi til at programmere lidt i MC igen. Vi starter med et let program-eksempel, der illustrerer, hvordan forskellige ting udføres i MC. Først selve programopstillingen:

Programeksempel 0401:

```
1  start:
2  move.l     #$00,d0
3  move.l     #$04,d1
3  lea.l      table,a0
5
6  loop01:
7  add.w      (a0)+,d0
8  dbra       d1,loop01
9
10 lea.l      result,a0
11 move.l     d0,(a0)
12
13 rts
14
15 result:
16 blk.l      1,0
17
18 table:
19 dc.w       2,4,6,8,10
```

Dette program tager tallene i linie 19 og summerer dem. Resultatet havner i et LONGWORD i DATAREGISTER d0. Programmet udfører altså ikke særligt meget, men det giver nyttig viden til senere brug.

Linie 1: Her begynder programmet, og startpunktet har fået LABELen "start:". K-SEKA danner nu en variabel med navnet "start", og lægger adressen på programmets første BYTE ind i denne variabel.

Linie 2: Vi skal bruge et DATAREGISTER til at summere tallene (2,4,6,8,10) i, vi vælger d0. Vi nulstiller registret ved at lægge $00 ind i det. Vi kunne have brugt kommandoen clr.l d0 (=CLEAR LONGWORD DATAREGISTER d0) i stedet. Det ville have givet en hurtigere programkørsel end move.l #$00,d0

Linie 3: Vi skal summere 5 tal, så vi lægger en tæller ind i et DATAREGISTER (i dette tilfælde i d1). Vi vil lade en LOOP køre fem gange, og hver gang indlæse et nyt tal, som vi adderer til det tal, som er i d0. Da LOOPen skal "rotere" fem gange lægger vi 4 ind i det DATAREGISTER (d1), hvori vi tæller antal LOOPs. Vi tæller ned på denne måde: 4,3,2,1,0 ialt 5.

Linie 4: Lægger adressen på de fem tal fra linie 19 ind i ADRESSEREGISTER a0. I linie 19 beder vi assembleren om at reservere fem WORDS, og lagre tallene 2,4,6,8 og 10 i hvert sit WORD. Assembleren laver en variabel, som den kalder for "table" (i linie 18), og lægger adressen på det første af vore fem tal tallet 2) ind i den; så ved den, hvor den kan genfinde tallene. Egentlig indlægger den et tal (en OFFSET), som viser afstanden (målt i antal BYTEs) mellem programmets første BYTE og vore tal. På den måde spiller det ingen rolle, hvor i maskinen vort program havner. Der vil jo altid være samme afstand mellem program-start og vore fem tal.

Linie 6: Er er en LABEL, vi har kaldt den "loop01:". Læg mærke til kolonnet (:), der viser, at dette er en LABEL, og at ordet "loop01" ikke assembles til enere og nuller. Se iøvrigt forklaringen til linie 1. Hvis du har mange LOOPs i dine programmer, og det er meget sandsynligt, kan det være fornuftigt enten at nummerere dem fortløbende: loop01, loop02, loop03 osv. eller at give dem beskrivende navne som f. eks test, copper, buffer, mustest osv. I K-SEKA kan LABELS bestå af lige så mange bogstaver (og næsten alle tegn) du vil have. Du er altså ikke bundet af en maximum længde på dine LABEL-navne. Der skelnes ikke mellem store og små bogstaver, så hvis du har to LABELs, der hedder henholdsvis "dataskolen:" og "DATASKOLEN:" så giver K-SEKA en fejlmelding, fordi den betragter de to LABELs som ens.

Linje 7: Her hentes, første gang LOOP'en udføres, tallet 2 fra BUFFERen - a0 indeholder jo adressen på det første tal. Det lægges derefter i d0, og adressen i a0 forhøjes med to (husk vi arbejder med WORDs = 2 BYTEs) for at få det næste tals adresse i BUFFERen.

Linie 8: Instruktionen "dbra" gør tallet i d1 1 mindre. Derefter testes d1 for, om indholdet er mindre end nul (egentlig -1, det forklares i andet brev). Er det ikke det, hopper programmet tilbage til linie 7, og LOOPen udføres en gang til

Linie 10: Når alle tal er adderet lægger vi adressen på BUFFERen ("result") ind i a0. Se linie 15.

Linie 11: Resultatet af sammenlægningen flyttes (som et LONG-WORD) fra d0 til den adresse som ADRESSEREGISTER a0 indeholder, nemlig adressen på BUFFERen "result". Se linie 10.

Linie 13: Programmet slutter her og returnerer til K-SEKA, hvis det var derfra, du startede programmet.

Linie 15: LABELen på BUFFERen "result".

Linie 16: Her reserveres et LONGWORD, som derefter nulstilles. Det er i dette LONGWORD vores resultat lægges.

Linie 18: Læs forklaringen til linie 4 en gang til! Når du har skrevet dette program ind, assembler du det. Det du skal gøre er at skrive "a", og når K-SEKA spørger om OPTIONS> skriver du vh og trykker igen på RETURN. Den vil nu assemble programmet og stoppe for hver side. Tryk på SPACE (mellemrums- tasten) for at fortsætte. Hvis du ikke skriver noget, når K-SEKA beder om OPTIONS, så kommer der ikke noget frem på skærmen; nøjes du med at skrive "v", så vil assemblingen rulle non-stop over skærmen.

Når assemblingen er udført skriver K-SEKA alle LABELs ud med tilhørende OFFSET, regnet fra LABELen "start:".

Du starter programmet ved at skrive "jstart" (= JUMP til LABEL "start:").

I den liste over registrene, som K-SEKA viser efter programkørslen, kan du se at d0 indeholder $1E (=30), - og det er lige netop summen af 2+4+6+8+10.

Men vi lagde jo resultatet i en BUFFER som et WORD (linie 11). Kan vi få K-SEKA til at vise os den også? Hvis du skriver "qresult" ("q" står for QUERY = spørgsmål), viser K-SEKA dig den del af hukommelsen, der indeholder vores LONGWORD (med summen $1E). Den adresse findes i LABELen "result:". Ved siden af resultatet vil du se tallene $0002, $0004, $0006, 0008, $000A, som er vore tals hexadecimale form.

**OPGAVE 0401: Vi håber du forstod alt dette, og at du går ind i programmet og forandrer her og der. Du kan f.eks. ændre antallet af tal, som skal summeres.**

## BITPLANES I

Et BITPLANE er et hukommelsesområde i AMIGA, som indeholder skærmens grafik-information (data). Jo flere farver du vil benytte, desto flere BITPLANES reserveres til skærm-data.

En PIXEL (udtales piksell og er en forkortelse af PICTURE CELL, oversat: billed-element) er det mindste punkt du kan se på skærmen. En PIXEL kan være ON eller OFF (tændt eller slukket). Informationen om en PIXELS tilstand (om den er ON eller OFF) lagres i en BIT i den del af hukommelsen, der er afsat til dine skærm-data.

Lad os sige at et BITPLANE begynder på adresse $010000. BYTEene ligger så ordnet på denne måde:

```
PIXEL:      * * * * * * * * * * * * * * * * * * * * * * ......
BIT nr:     7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 ......
ADRESSE:        $010000           $010001           $010002
```

Den første skærmlinie henter så sine PIXEL-data fra adresse 010000, $010001, $010002 og så videre. Når første linie er tegnet, fortsætter elektronkanonen på næste linie ved at hente flere data fra skærmhukommelsen.

Hvis du har en skærm, der er 320 PIXELs bred, så svarer det til (320/8=) 40 BYTEs pr. linie. Det betyder, at i vort eksempel findes nummer 2 skærmlinies data fra og med adresse 010028.

**OPGAVE 0402:** Hvor begynder data for skærmlinie 4 i eksemplet ovenfor?

Amigaen har mange forskellige måder at vise grafik på, de såkaldte GRAFIK-MODES (udtales:-mouds). Nedenfor beskrives nu de forskellige GRAFIK-MODES:

| TYPE | OPLØSNING(X*Y) | ANTAL BITPLANES | ANTAL FARVER |
|---|---|---|---|
| LORES | 320 * 256 | 1 til 6 | 2 til 64 |
| HIRES | 640 * 256 | 1 til 4 | 2 til 16 |
| LORES-LACE | 320 * 512 | 1 til 6 | 2 til 64 |
| HIRES-LACE | 640 * 512 | 1 til 4 | 2 til 16 |
| HAM | 320 * 256 | 6 | 4096 |
| HAM-LACE | 320 * 512 | 6 | 4096 |

Forkortelserne i den venstre kolonne betyder:

| | | |
|---|---|---|
| LORES | LOW RESOLUTION | (lav opløsning) |
| HIRES | HIGH RESOLUTION | (høj opløsning) |
| LORES-LACE | LOW RES.INTERLACE | (lav opløsning interlace) |
| HIRES-LACE | HIGH RES.INTERLACE | (høj opløsning interlace) |
| HAM | HOLD AND MODIFY | (hold og ændre) |
| HAM-LACE | HAM INTERLACE | (hold og ændre interlace) |

Alle disse begreb vil blive behandlet efterhånden som vi kommer til dem.

Når der benyttes flere BITPLANEs, kan man forestille sig dem lagt "oveni hinanden" som kort i en kortstak. Prøv at forestille dig 2 plader, hvori der er stanset små huller tæt ved siden af hinanden over det hele. Når disse to plader lægges sammen, så er hullerne nøjagtigt lige over hinanden.

Forestil dig, at hver plade repræsenterer en BITPLANE og et hul repræsenterer en BIT (som kan være 1 eller 0). Pladerne ligger nu nøjagtigt over hinanden kant mod kant og forestil dig så, at en knappenål bliver stukket igennem et af hullerne. Disse to huller (læs BITs), som knappenålen er stukket igennem, bruger AMIGA til at finde ud af, hvilket farveregister, der skal benyttes for at farvelægge det punkt på skærmen, som BITsene repræsenterer. Hvis det f.eks var hullerne længst oppe i venstre hjørne nålen blev stukket igennem, så repræsenterer de det øverste venstre punkt (PIXEL) på din skærm.

Lad os bruge WORKBENCH-skærmen som eksempel.

WORKBENCH-skærmen vises i HIRES med 640 * 256 PIXEL's opløsning og benytter sig af 2 BITPLANES. Der kan derfor højst være 4 farver på skærmen samtidigt. Studer FIGUR 3 bagest i dette brev og læs forklaringen nedenfor:

Lad os sige, at BITPLANE 1 begynder på adresse $010000, og BITPLANE 2 begynder på adresse $020000. Derefter forestiller vi os BITPLANE 1 lagt ovenpå BITPLANE 2. Hver PIXEL repræsenteres på den måder af 2 BITs, en fra hvert BITPLANE. Som du ved, kan et 2-BITs-tal indeholde værdier fra 0 til 3.

%00 = 0
%01 = 1
%10 = 2
%11 = 3

Hvis første BIT i BITPLANE 1 er sat til "1", og første BIT i BITPLANE 2 er sat til "0" vil dette blive: %01 (eller tallet "1" decimalt"). Observer at den øverste BITPLANE indeholder LSB-BITsene (de mindst værdifulde - det er derfor man får $01 og ikke $10, når man lægger dem sammen).

Lad os studere de såkaldte FARVE-REGISTRE i AMIGA'en. AMIGA har 32 farveregistre, som hver består at et WORD (16 BITs). Disse BITs er inddelt således:

```
BIT nr:    15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
Indhold:    -  -  -  - R3 R2 R1 R0 G3 G2 G1 G0 B3 B2 B1 B0
```

Som du måske har gættet, så betyder R,G og B: RØDT, GRØNT og BLÅT. Disse farver kaldes for grundfarver. Ved at blande rødt, grønt og blåt i forskellige mængder,kan man lave præcis den farve, som ønskes på skærmen.

Der er dog en lille begrænsning: AMIGA kan kun blande dem således, at den får 4096 forskellige farver at "lege" med (16 nuancer af hver grundfarver: 16*16*16 = 4096). Et TV bruger de samme grundfarver, men der kan de blandes til flere millioner forskellige farver.

R0 til R3 er en 4 BITs-gruppe (NIBBLE), som kan indeholde en værdi fra og med 0 til og med 15 (Se BREV 1 hvis du ikke kan huske hvorfor). Det giver 16 forskellige værdier, eller med andre ord: 16 forskellige nuancer. Dette gælder også G0 til G3, og B0 til B3. BITsene med nummer 12 til 15 bruges ikke, så dem kan du glemme. (COMMODORE advarer stærkt imod at bruge sådanne "tiloversblevne" BITs til at lægge data ind i).

Som nævnt kan hver grundfarve justeres i 16 trin. Lad os tage et eksempel.

| FARVE | RØDT | GRØNT | BLÅT | HEXADECIMALT |
|---|---|---|---|---|
| Sort | 0 | 0 | 0 | $0000 |
| Hvid | 15 | 15 | 15 | $0FFF |
| Gult | 15 | 15 | 0 | $0FF0 |
| Rødt | 15 | 0 | 0 | $0F00 |
| Lyseblå | 0 | 15 | 15 | $00FF |
| Mørkegrå | 7 | 7 | 7 | $0777 |
| Lysegrå | 12 | 12 | 12 | $0CCC |
| Rosa | 15 | 0 | 15 | $0F0F |

Som du ser bliver farven mere intens jo højere tal, der bruges. Derfor bliver $0000 sort (ingen farve benyttes), og $0FFF bliver hvid (fuld intensitet af de tre grundfarver rød, grøn og blå). I og med at værdierne består af 4 BITs (1 NIBBLE) er det meget enkelt at skrive farverne HEXADECIMALT:

```
HEXADECIMALT:      0       F       F       F

HVID:              -      RØD     GRØN     BLÅ
```

Bemærk at grundfarverne ikke er de samme som i den almindelige farvelære, hvor det er rød, blå og gul der er grundfarver.

Tilbage til eksemplet om BITPLANEs. Altså: når man bruger 2 BITPLANES således som f.eks WORKBENCH-skærmen gør, vil man få en værdi fra 0 til 3 for hver PIXEL (Husk to BITPLANEs er to BITs, to BITs er fire farver). Det fungerer på den måde, at hvis værdien på en PIXEL er 2 - vil PIXELen blive farvet med den farve, som FARVEREGISTER 2 indeholder.

Det første farveregister (nummer 0) ligger på adressen $DFF180 og er et Word langt. Det næste farveregister (nummer 1) ligger derfor på adressen $DFF182 (et WORD = 2 BYTEs). Det tredie farveregister (nummer 2) ligger på adressen $DFF184. Hvis du lægger værdien $0F00 ind i det tredie register (nummer 2), så vil den PIXEL, som vi nævnte i afsnittet ovenfor blive farvet rød. Klart som blæk, ikke sandt?

Farveregistrene nummereres fra 0 til 31. Kanten på skærmen, som på engelsk kaldes BORDER (udtales med hårdt d), henter sin farve fra det første farveregister (nummer 0 - $DFF180), Altså: Hvis værdien af de to BITs på PIXELen er 0, vil den få samme farve som kanten af skærmen (BORDERen).

Lad os nu forestille os en LORES-skærm med 3 BITPLANEs. Den kan vise 8 forskellige farver (tre BITPLANEs er tre BITs, tre BITs er otte farver). Lad os regne ud, hvor meget hukommelse der skal til:

Vi ved at skærmen er 320 PIXELs bred. Vi ved også at en BYTE har otte BITs. Altså: 320/8 = 40 BYTEs pr linie. 40 * 256 linier = 10240 BYTEs pr BITPLANE. Til sidst ganger vi med antal BITPLANES: 10240 * 3 = 30720 BYTEs. Så svaret bliver, at denne skærm - for at kunne vise 8 farver - skal bruge 30720 BYTEs af AMIGA's hukommelse.

Dette svarer til 30 kB. Hvad er så 30 kB? Jo det lille "k" betyder kilo. I dagligdags tale betyder lille "k" 1000 enheder af et eller andet, f.eks kilometer eller kilogram. I data-verdenen derimod, betyder lille "k" 1024 enheder. Her taler vi om BYTEs, så vi mener altså 30 kiloBYTEs (30 kB). Dette tal får vi ved at dividere antallet af BYTEs (30720) med 1024 (antallet BYTEs i et "kilo"). Resultatet bliver 30 kB. Bemærk skrivemåden af "kB": lille "k" og stort "B". Det er ikke alle data-forfattere som ved dette, så du kan finde mange for-skellige kombinationer!

Nu kan du regne ud, at hvert BITPLANE, du bruger, optager 10 kB af AMIGA's hukommelse. (10240/1024 = 10)

<u>Vi demonstrerer nu vor nye viden med et programeksempel:</u>

## Programeksempel 0402:

```
1  move.w   #$01a0,$dff096
2
3  move.w   #$1200,$dff100
4  move.w   #$0000,$dff102
5  move.w   #$0000,$dff104
6  move.w   #0,$dff108
7  move.w   #0,$dff10a
8  move.w   #$2c81,$dff08e
9  move.w   #$f4c1,$dff090
10 move.w   #$38c1,$dff090
11 move.w   #$0038,$dff092
12 move.w   #$00d0,$dff094
13
14 lea.l    screen,a1
15 lea.l    bplcop,a2
16 move.l   a1,d1
17 move.w   d1,6(a2)
18 swap     d1
19 move.w   d1,2(a2)
20
21 lea.l    copper,a1
22 move.l   a1,$dff080
23
24 move.w   #$8180,$dff096
25
26 wait:
27 btst     #6,$bfe001
28 bne      wait
29
30 move.w   #$0080,$dff096
31 move.l   $04,a6
32 move.l   156(a6),a1
33 move.l   38(a1),$dff080
34 move.w   #$80a0,$dff096
35 rts
36
37 copper:
38 dc.w     $2c01,$fffe
39 dc.w     $0100,$1200
40
41 bplcop:
42 dc.w     $00e0,$0000
43 dc.w     $00e2,$0000
44
45 dc.w     $0180,$0000
46 dc.w     $0182,$0ff0
47
48 dc.w     $ffdf,$fffe
49 dc.w     $2c01,$fffe
50 dc.w     $0100,$0200
51 dc.w     $ffff,$fffe
52
53 screen:
54 blk.b    10240,0
```

## BITPLANES II

I dette afsnit om BITPLANES, indleder vi med at forklare, hvad der foregår i COPPER-listen til det program i BREV III, som tegnede røde, hvide og blå linier:

Linie 18 venter på monitor-linie $90, position $01.

Linie 19 lægger helt rød ind i FARVEREGISTER 0.

Linie 20 venter på monitor-linie $A0, position $01.

Linie 21 lægger helt hvid ind i FARVEREGISTER 0.

Linie 22 venter på monitor-linie $A4, position $01.

Linie 23 lægger helt blå ind i FARVEREGISTER 0.

Linie 24 venter på monitor-linie $AA, position $01.

Linie 25 lægger helt hvid ind i FARVEREGISTER 0.

Linie 26 venter på monitor-linie $AE, position $01.

Linie 27 lægger helt rød ind i FARVEREGISTER 0.

Linie 28 venter på monitor-linie $BE, position $01.

Linie 29 lægger helt sort ind i FARVEREGISTER 0.

COPPER-listen startes af hovedprogrammet og køres uafhængigt af alt andet, om og om igen. Dette er også et eksempel på, hvordan man ved hjælp af COPPERen og kun et FARVEREGISTER kan få flere farver på skærmen.

**Opgave 0403:** Studer COPPER-listen indgående. Gå derefter ind den og forsøg at lave forskellige ændringer. Se om du kan ændre den, således at linierne bliver blå og gule.

Vi fortsætter med registre, deres funktioner og opsætning:

**$DFF100 WRITE - BPLCON0 (BITPLANE CONTROL 0):**

| BIT NR. | FUNKTION |
|---|---|
| 15 | HIRES |
| 14 | BPU2 |
| 13 | BPU1 |
| 12 | BPU0 |
| 11 | HAM MODE |
| 10 | DUAL PLAYFIELD |
| 9 | COLOR ENABLE |
| 8 | GAUD |
| 7 | - |
| 6 | - |
| 5 | - |
| 4 | - |
| 3 | LPEN ENABLE |
| 2 | INTERLACE |
| 1 | EXT.RESYNC |
| 0 | - |

BIT 15: Denne BIT skal være 1, hvis du vil bruge HIRES (640 PIXELs horisontalt)

BIT 12 - 14: Dette er en 3-BIT-gruppe, som bruges til at angive, hvor mange BITPLANES du vil bruge.

BIT 11: Denne BIT skal sættes til 1 hvis du vil benytte HAM (**H**OLD **A**ND **M**ODIFY).

BIT 10: Denne BIT skal være 1 for at aktivere DUAL PLAYFIELD MODE.

BIT 9: Denne BIT skal være 1 for at få farvesignal på videoudgangen (ingen indflydelse på A-500).

BIT 8: Denne BIT sættes til 1, for at få lyd på enkelte GENLOCK. (**G**ENLOCK **AUD**IO ENABLE).

BIT 4 - 7: Ikke benyttet.

BIT 3: Denne BIT skal sættes til 1, hvis du vil bruge en LYSPEN.

BIT 2: Denne BIT sættes til 1 for at aktivere INTERLACE MODE (512 PIXELs-linier, vertikalt).

BIT 1: Denne BIT sættes til 1 i enkelte tilfælde, når man bruger GENLOCK.

BIT 0: Ikke benyttet.

**$DFF08E WRITE - DIWSTRT (DISPLAY WINDOW START)** er sat op således:

| BIT NR | FUNKTION |
|---|---|
| 15 | V7 |
| 14 | V6 |
| 13 | V5 |
| 12 | V4 |
| 11 | V3 |
| 10 | V2 |
| 9 | V1 |
| 8 | V0 |
| 7 | H7 |
| 6 | H6 |
| 5 | H5 |
| 4 | H4 |
| 3 | H3 |
| 2 | H2 |
| 1 | H1 |
| 0 | H0 |

**$DFF090 WRITE - DIWSTOP (DISPLAY WINDOW STOP)** er sat op således:

| BIT NR | FUNKTION |
|---|---|
| 15 | V7 |
| 14 | V6 |
| 13 | V5 |
| 12 | V4 |
| 11 | V3 |
| 10 | V2 |
| 9 | V1 |
| 8 | V0 |
| 7 | H7 |
| 6 | H6 |
| 5 | H5 |
| 4 | H4 |
| 3 | H3 |
| 2 | H2 |
| 1 | H1 |
| 0 | H0 |

**$DFF092 WRITE - DDFSTRT (DISPLAY DATAFETCH START)** er sat op således:

| BIT NR | FUNKTION |
|---|---|
| 15 | - |
| 14 | - |
| 13 | - |
| 12 | - |
| 11 | - |
| 10 | - |
| 9 | - |
| 8 | - |
| 7 | H8 |
| 6 | H7 |
| 5 | H6 |
| 4 | H5 |
| 3 | H4 |
| 2 | H3 |
| 1 | - |
| 0 | - |

**$DFF094 WRITE - DDFSTOP (DISPLAY DATAFETCH STOP)** er sat op således:

| BIT NR | FUNKTION |
|---|---|
| 15 | - |
| 14 | - |
| 13 | - |
| 12 | - |
| 11 | - |
| 10 | - |
| 9 | - |
| 8 | - |
| 7 | H8 |
| 6 | H7 |
| 5 | H6 |
| 4 | H5 |
| 3 | H4 |
| 2 | H3 |
| 1 | - |
| 0 | - |

**BITPLANE POINTERs:**

| BITPLANE | ADRESSE | HI/LO |
|---|---|---|
| 1 | $DFF0E0 | HI(BIT 16-31) |
| 1 | $DFF0E2 | LO(BIT 0-15) |
| 2 | $DFF0E4 | HI(BIT 16-31) |
| 2 | $DFF0E6 | LO(BIT 0-15) |
| 3 | $DFF0E8 | HI(BIT 16-31) |
| 3 | $DFF0EA | LO(BIT 0-15) |
| 4 | $DFF0EC | HI(BIT 16-31) |
| 4 | $DFF0EE | LO(BIT 0-15) |
| 5 | $DFF0F0 | HI(BIT 16-31) |
| 5 | $DFF0F2 | LO(BIT 0-15) |
| 6 | $DFF0F4 | HI(BIT 16-31) |
| 6 | $DFF0F6 | LO(BIT 0-15) |

**BPLCON0 ($DFF100):** Dette register indeholder mange nye udtryk. Vi starter med at forklare dem vi får brug for lige nu. Resten kommer i BREV XII.

HIRES sættes til "0" for at sætte bredden til 320 PIXELs (LORES), eller til "1" for at sætte bredden til 640 PIXELs (HIRES).

BPU0-2 er en 3-BITs-gruppe. Disse BITs bliver brugt til at sætte hvor mange BITPLANEs, som skal bruges (0-6).
COLOR sættes til "1" for at aktivere farvesignal på videoudgangen. Denne BIT sættes næsten altid til "1".

BPLCON1-registret ($DFF102 bruges i forbindelse med SCROLL (flytning) af skærmbilledet. Vi vender tilbage til dette register i et senere brev.

BPLCON2-registret ($DFF104) benyttes til SPRITEs og DUAL PLAYFIELDS. Vi vender tilbage til dette register i BREV V og BREV XII.

MODULO-registrene ($DFF108 og $DFF10A) bliver forklaret i BREV VI i forbindelse med BLITTERen.

DIWSTRT ($DFF08E) benyttes til at bestemme, hvor billedets øverste venstre hjørne skal være på skærmen. Øverste linie er som regel linie $2C (44 DECIMALT). Denne position kan sættes mellem $15 og $FF (21 og 255 DECIMALT). En anden "standard"-position er $1C (28 DECIMALT). Den medfører, at øverste linie havner ovenover "plastik-kanten" på monitoren. Dette kaldes OVERSCAN. Venstre position er som regel $81 (LORES) eller $80 (HIRES). Hvis man ønsker at billedet skal gå ud over monitorens kant (OVERSCAN) bruges positionen $71 (LORES) eller

$70 (HIRES), altså 16 PIXELs bredere.

Eksempel: $2C81,$1C71,$2C80

Vi har nu gennemgået, hvordan man fastsætter billedets øverste venstre hjørne. Næste register hedder DIWSTOP ($DFF090).

Dette register bruges til at bestemme, hvor nederste højre hjørne skal være. Det er mere kompliceret. Lad os begynde med højre position. Denne værdi er, som i det andet register, opgivet i 8 BITs, altså en BYTE, som kan indeholde værdier fra 0 ($00) til 255 ($FF).

Lad os sige, at vi skal sætte en LORES-skærm op, som er 320 PIXELs bred og 256 PIXELs høj. Hvis vi siger, at venstre position er $81 (standard), vil det sige at højre position er $81 + $140 (320) = $1C1. Som du ser, er dette tal for stort til at kunne passe ind i en BYTE. Derfor er maskinen konstrueret således, at vi bare lægger $C1 ind i registret. Maskinen vil selv lægge 256 ($100) til. OBS! dette sker i selve CHIPens logik, altså ikke via SOFTWARE. Det vil sige, at hvis du lægger værdien $00 ind i registeret, vil den egentlige værdi blive $100.

Den sidste position vi skal angive er "nederste linie" på skærmen. Hvis man prøver at regne dette ud, bliver tallene også i dette tilfælde for store. Hvis "øverste linie" har position $2C og vi ønsker en skærm, som er 256 linier høj, får vi tallet $2C + $100 (256) = $12C (Du tror måske at det virker på samme måde som ovenfor? - Desværre....). I USA har de kun 200 linier vertikalt på AMIGA. og de har derfor ingen problemer med dette register, fordi $2C + $C8 (200) = $F4. I Europa derimod, hvor vi har 256 linier, må vi gøre sådan: først lægges $F4 ind i registeret og derefter lægges $38 (56) ind i registeret. Disse to tal vil blive adderet inden i CHIPen og vi får værdien $F4 + $38 = $12C. Det gøres sådan:

```
move.w  #$F4C1,$DFF090
move.w  #$38C1,$DFF090
```

Disse to operationer skal udføres lige efter hinanden, ellers vil #$38C1 blive betragtet som en ny værdi, og ikke blive adderet til registeret. Husk at "højre position" ($C1) er uforandret.

En anden ting vi vil fremhæve er, at hvis du skal sætte en HIRES-skærm op (640 PIXELs i bredden), så skal du bruge LORES-tal (320 PIXELs), når du regner værdierne ud.

Altså: Hvis du som "venstre-position" i HIRES bruger $80 (som er standard i HIRES) og lægger 640 til, vil du få et forkert tal. Du skal lægge 640/2 = 320, til. Vi siger at DIWSTRT ($DFF08E) og DIWSTOP ($DFF090) angives i "LORES-tal".

DDFSTRT ($DFF092) og DDFSTOP ($DFF094) er to andre registre,

som bruges til opsætning af en skærm. Du behøver egentlig ikke at vide, hvad som sker med disse registre, fordi de afhænger af værdierne DIWSTRT og DIWSTOP. Lad os starte med LORES først. Hvis vi siger, at "venstre position" i DIWSTRT er $81 og vi benytter LORES, skal DDFSTRT være $0038. Lad os vise dette ved hjælp af en formel:

LORES:

DDFSTRT: ($81/$2_{10}$) - $8,5_{10}$ = $0038

DDFSTOP: (($320_{10}/16_{10}$) - $1_{10}$) * $8_{10}$) + $0038 = $00D0

Husk at hvis DDFSTRT-værdierne ikke ender på $0 eller $8, så skal de rundes af nedefter (eks. $0030, $0038, $0040, osv). DDFSTOP værdierne skal også ende på $0 eller $8, men rundes af opefter (eks: $00E8, $00D0, $00D8, osv).

HIRES:

DDFSTRT: ($80/$2_{10}$) - $4_{10}$ = $003C

DDFSTOP: (($640_{10}/16_{10}$) - $2_{10}$) * $4_{10}$) + $003C = $00D4

Værdierne i DDFSTRT og DDFSTOP i HIRES-mode skal altid ende på $4 eller $C, og rundes af på samme måde som i LORES, altså DDFSTRT rundes ned og DDFSTOP rundes op.

Lad os lige gå tilbage til program-eksempel 0402.

Linie 14: Hent adressen på vores BUFFER og læg den i A1.

Linie 15: Hent adressen på punktet "bplcop:" (nede i vores copperliste) og læg den ind i A2.

Linie 16: Læg indholdet af A1 ind i D1.

Linie 17: Læg de første 16 BITs fra D1 ind på adressen som (A2 + 6) peger på. Tallet 6 i denne instruktion kaldes en OFFSET. Dette betyder at dataene bliver lagt 6 BYTEs **foran** den adresse som A2 peger på.

Linie 18: Lad de 16 første og de 16 sidste BITs i D1 bytte plads.

Linie 19: Læg de første 16 BITs fra D1 ind i adressen som (A2 + 2) peger på.

Linie 53: LABELen til vores skærm-data.

Linie 54: Her deklareres vores skærm-data. 10240 bytes er nok til en 320 * 256 skærm med **en** BITPLANE.(320/8 = 40. 40 * 256 = 10240. 10240 * 1 = 10240).

Som du ser, har vi i linie 54 sat hele hukommelsen til 0 (blk.b 10240,**0**). Prøv nu at lægge en linie ind sådan:

```
53  screen:
54  dc.b   $80
55  blk.b  10240,0
```

Du vil nu se en pixel øverst i skærmens venstre hjørne.

**Opgave 0404:** Prøv selv at lægge andre tal ind, EKSPERIMENTER!

I "BREV4"-kataloget på KURSUSDISKETTE nr 1 ligger der en fil, som hedder "SCREEN". Denne fil indeholder et billede, som du kan hente ind i din skærmbuffer. Dette gøres i K-SEKA ved hjælp af kommandoen "ri" (READ IMAGE). Gør sådan:

```
SEKA>ri
FILENAME>brev4/screen
BEGIN>screen
END>
```

På spørgsmålet "BEGIN" angiver du, hvor du vil indlæse filen. I dette tilfælde har vores skærm-buffer LABELen: "screen" - og filen havner der. På spørgsmålet "END" trykker du bare RETURN for at indlæse hele filen. Husk at du skal assemble programmet først, og at skærmbufferen slettes mellem hver assembling, således at du skal læse den ind en gang til.

## BITPLANE III

```
1   move.w   #$01a0,$dff096
2
3   move.w   #$5200,$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   #$1c71,$dff08e
10  move.w   #$f4d1,$dff090
11  move.w   #$40d1,$dff090
12
13  move.w   #$0030,$dff092
14  move.w   #$00d8,$dff094
15
```

```
16  lea.l    screen,a1
17  move.l   #$dff180,a2
18  moveq    #31,d0
19  colorloop:
20  move.w   (a1)+,(a2)+
21  dbra     d0,colorloop
22
23  move.l   a1,d1
24  lea.l    bplcop,a2
25  addq.l   #2,a2
26  moveq    #4,d0
27
28  bplloop:
29  swap     d1
30  move.w   d1,(a2)
31  addq.l   #4,a2
32  swap     d1
33  move.w   d1,(a2)
34  addq.l   #4,a2
35  add.l    #12320,d1
36  dbra     d0,bplloop
37
38  lea.l    copper,a1
39  move.l   a1,$dff080
40
41  move.w   #$8180,$dff096
42
43  wait:
44  btst     #6,$bfe001
45  bne      wait
46
47  move.w   #$0080,$dff096
48
49  move.l   $4,a6
50  move.l   156(a6),a1
51  move.l   38(a1),$dff080
52
53  move.w   #$80a0,$dff096
54
55  rts
56
57  copper:
58  dc.w     $1c01,$fffe
59  dc.w     $0100,$5200
60
61  bplcop:
62  dc.w     $00e0,$0000
63  dc.w     $00e2,$0000
64  dc.w     $00e4,$0000
65  dc.w     $00e6,$0000
66  dc.w     $00e8,$0000
67  dc.w     $00ea,$0000
68  dc.w     $00ec,$0000
69  dc.w     $00ee,$0000
70  dc.w     $00f0,$0000
71  dc.w     $00f2,$0000
```

```
72
73  dc.w      $ffdf,$fffe
74
75  dc.w      $3401,$fffe
76
77  dc.w      $0100,$0200
78
79  dc.w      $ffff,$fffe
80
81  screen:
82  blk.l     $3c38,0
```

Linie 1: Sluk BITPLANE, COPPER og SPRITE DMA'erne.

Linie 3: Opsætter 5 BITPLANEs.

Linie 4: SCROLLværdi til 0.

Linie 5: BITPLANE PRIORITY til 0.

Linie 6-7: EVEN og ODD MODULO til 0.

Linie 9: Sætter venstre skærmposition til $71 og øverste linie til $1C. Dette resulterer i at vi får både venstre og øvre position 16 PIXELs længere ud end normalt - OVERSCAN.

Linie 10: Sætter højre skærmposition til $D1 og nederste linie til $F4. Dette vil resultere i at vi får højre position 16 PIXELs længere ud. Den nederste linie er nu 244 - med andre ord **ikke** OVERSCAN. Det kommer i næste linie.

Linie 11: Sætter højre skærmposition til $D1 og adderer $48 til nederste linieposition. Den nederste linie er nu også blevet sat til $F4 + $40, altså OVERSCAN. Linie 9 til 11 har nu resulteret i en skærm, som er 352 * 280.

Linie 13: Sætter DISPLAY DATAFETCH START til $0030. (Se tidligere forklaringer).

Linie 14: Sætter DISPLAY DATAFETCH STOP til $00D8. (Se tidligere forklaringer).

Linie 16: Lægger adressen på "screen:" ind i A1.

Linie 17: Lægger #$DFF180 ind i A2. $DFF180 er som bekendt farveregister 0. Du skal være opmærksom på at det er selve tallet $DFF180, som lægges ind i A2 og ikke en værdi, som adresse $DFF180 peger på.

Linie 18: Lægger tallet 31 (DECIMALT) ind i D0. Dette register bliver brugt som tæller.

Linie 19: En LABEL.

Linie 20: Lægger værdien af adressen som A1 peger på, ind i adressen, som A2 peger på (et WORD af gangen - 2 BYTEs). Derefter adderes 2 til både A1 og A2, Denne instruktion vil læse de farve-data, som ligger i begyndelsen af vores billede (som vi skal indlæse senere) ind i FARVEREGISTRENE.

Linie 21: Trækker **en** fra i D0 og tester om D0 er -1 hvis ikke så hop tilbage til "colorloop:". Den vil køre loopen 32 gange, således at vi får indlæst alle FARVEREGISTRENE.

Linie 23: Læg indholdet af A1 ind i D1. A1 peger nu på billedets første skærmdata-BYTE.

Linie 24: Læg adressen på "blpcop:" ind i A2.

Linie 25: Læg (adder) 2 til A2.

Linie 26: Læg tallet 4 ind i D0. D0 bliver også her brugt som tæller. Denne gang tæller den BITPLANEs (5 stk)

Linie 28: En LABEL.

Linie 29: SWAP (ombyt) er en ny instruktion, som bruges til at bytte om på BIT 0-15 og BIT 16-31 i et data-register. Altså bytte om på WORDene i data-registret.

Linie 30: Læg WORDets indhold (BIT 0-15) ind på adressen som A2 peger på. Som du ser vil den lægge det øverste WORD fra D1 (i og med at vi udførte en SWAP på forrige linie) ind iCOPPER-listens MOVE-instruktion. Det er i dette tilfælde en MOVE til register $DFF000 + $00E0 = $DFF0E0 (se forklaring om COPPERen i BREV III), som er BITPLANE-POINTEREN til BITPLANE 1. BITPLANE-POINTEREN i adresse $DFF0E0 skal indeholde øverste WORD (BIT 16-31) af adressen til vores BITPLANE-DATA (skærm-data). Adresse $DFF0E2, som også tilhører BITPLANE 1 skal indeholde nederste WORD (BIT 0-15) af adressen på vores BITPLANE-DATA. Dette WORD bliver lagt ind i COPPER-listen i linie 33. Se tabellen over BITPLANE-POINTERs i forrige kapitel.

Linie 31: Læg 4 til i A2. Det får A2 til at pege på næste COPPER-instruktion.

Linie 32: Byt om på WORDene i D1 igen.

Linie 33: Læg WORDet (BIT 0-15) i D1 ind i adressen, som

A2 peger på.

Linie 34: Adder tallet 4 til A2.

Linie 35: Adder tallet 12320 (DECIMALT) i D1. Dette medfører at D1 peger på næste BITPLANE hvis adresse vi skal lægge ind i COPPER-listen. Tallet udregnes således:

Vores skærm er 352 PIXELs bred (fordi vi har 16 PIXELs ekstra på hver side) og 352/8 = 44 BYTEs. Højden på skærmen er 280 idet vi også definerede 16 PIXELs ekstra i top og bund, og 44 * 280 = 12320.

Linie 36: Træk **en** fra i D0. Test om D0 er blevet -1. Hvis ikke, hop tilbage til "bplloop:". Denne instruktion vil udføre loopen 5 gange, således at vi får lagt alle BITPLANE (skærmdata) adresserne ind i COPPER-listen.

Linie 38: Læg adressen på begyndelsen af COPPER-listen ind i A1.

Linie 39: Læg indholdet af A1 ind på adresse $DFF080 - som er COPPER-POINTEREN.

Linie 41: Tænd BITPLANE og COPPER DMA'erne igen.

Linie 43: En LABEL.

Linie 44: Check bit 7 i adresse $BFE001. BIT 7 i denne adresse indeholder: "1", hvis venstre musknap ikke er trykket ned - og "0" hvis knappen **er** trykket ned Denne instruktion vil sætte ZERO-flaget til "1", hvis BITen den testede var "1" - og "0" hvis ikke.

Linie 45: Denne instruktion laver et hop til "wait:" hvis ZERO-flaget er "1". Hvis ZERO-flaget er "0" vil den fortsætte til næste instruktion.

Linie 47: Slukker COPPER DMA'en.

Linie 49-51: Lægger adressen på den gamle COPPER-liste tilbage i COPPER-POINTERen. Når den gamle COPPER-liste startes igen, vil den vise SEKA-skærmen (hvis det var derfra du startede programmet).

Linie 53: Tænder COPPER- og SPRITE DMA'erne.
Linie 55: Afslut.

Linie 57: LABELen til COPPER-listen.

Linie 58: WAIT ($01,$1C) Denne COPPER-instruktion får COPPERen til at vente til elektronstrålen har nået første punkt ($01) på linie $1C (28 DECIMALT). Husk at COPPERen er en selvstændig processor med egne instruktioner, som ikke må forveksles med hoved-

processoren (MC-68000) og dens instruktioner. Mens COPPERen udfører sit program (COPPER-listen), vil MC-68000 i dette program-eksempel det meste af tiden bare gå i en loop og checke om musknappen er blevet trykket ned. En anden meget vigtig ting er, at position $01 på en linie (i dette tilfælde linie $1C) ikke er position 1 på skærmen. Første position på billedet (i dette tilfælde) er $71, altså 112 PIXELs længere mod højre end position 1. Du skal forestille dig, at en skærmlinie starter langt udenfor det "synlige" skærmbillede (ca. 15 cm til venstre ud i luften fra skærmkanten på AMIGA-monitoren).

Linie 59: COPPERen vil lægge tallet $5200 ind på adresse $DFF100, som er BITPLANE CONTROL 0 registret. Se forrige eksempel.

Linie 61: En LABEL.

Linie 62: COPPERen vil lægge tallet, som blev defineret af hovedprocessoren i linie 23-36 ind på adresse $DFF0E0. Som tidligere omtalt er dette BITPLANE-POINTERen (BIT 16-31) til BITPLANE 1.

Linie 63: Denne linie udfører det samme som ovenfor, men lægger tallet ind på adresse $DFF0E2, som er BITPLANE-POINTERen (BIT 0-15) for BITPLANE 1.

Linie 64-71: Udfører det samme som linierne 62 og 63, men lægger sine værdier ind i POINTERne til BITPLANE 2,3,4 og 5.
Forstået ? Det håber vi. Det er nemlig **meget vigtigt** at du forstår dette helt.

Linie 73: COPPERen venter til elektronstrålen har nået position $DF på linie $FF. Husk at man angiver PIXELs i WAIT-instruktionens horisontalposition to og to, så den egentlige position i dette tilfælde er $FD*2 = $1FA eller 506 DECIMALT. Denne COPPER-instruktion gør det muligt at vente på en position, som er større end $FF vertikalt. Linie $100 vil nu blive linie $00. Altså, hvis vi vil vente på linie $120, udfører vi først denne instruktion, derefter afventer vi på linie $20.

Linie 75: Vent på linie $3C - som egentlig er linie $13C.

Linie 77: COPPERen lægger tallet $0200 ind på adresse $DFF100. Som du ser har vi i toppen af COPPER-listen en MOVE, som lægger $5200 ind på samme adresse. Du har måske opdaget, at værdien $5200 sætter antal BITPLANEs til 5 (se forrige kapitel). $0200 vil sætte antal BITPLANES til 0, som giver en blank skærm. Hvis man ikke gør dette, risikerer man at maskinen fortsætter med at tegne skærmen op nedenfor den definerede skærmbund. Men

i satte jo skærmstørrelsen med DIWSTRT og DIWSTOP længere oppe i programmet siger du? Jo, men nogle maskiner har en gammel version af PAL-CHIPen, og de får problemer, når der sættes en PAL-skærm op (en skærm, som er højere end 200 linier). Det er derfor bedst at bruge denne metode, således at det fungerer på alle AMIGA'er.

Linie 79: Denne COPPER-instruktion kaldes ofte for STOP. Den stopper COPPERen, hopper til begyndelsen af COPPERLISTen og starter forfra, På denne måde går COPPERen i en LOOP indtil vi slukker for COPPER DMA'en.

Linie 81: LABELen til vores skærmbuffer.

Linie 82: Her deklareres vores skærmbuffer. Læg mærke til, at vi har angivet den i LONGWORDS (blk.l). Det kan være smart, hvis man har stor hukommelse, fordi SEKA så i enkelte tilfælde assembler dit program dobbelt så hurtigt. (Af en eller anden grund...).

For at få dette program til at vise andet end en sort skærm, skal du lægge noget ind i skærmbufferen. På kursusdisketten findes i "BREV4"-kataloget en fil, der hedder "SCREEN2". Du har allerede lært at læse sådan en ind, så prøv at læse denne fil ind på "screen", og kør programmet. HAVE FUN!

## BITPLANE DMA TIDSFORBRUG

Til sidst i dette brev skal vi se lidt på "TIME SLOT ALLOCATION PER HORIZONTAL LINE", som vi havde i BREV II.

Lad os først forklare hvad en CYKLUS er. En CYKLUS er et tidsrum, som er lige så langt som den tid det tager for elektronstrålen at tegne 2 PIXELs op på skærmen (4 HIRES PIXELs). I dette tidsrum kan man få **en** Acces (udtales akses og betyder adgang, dvs læsning eller skrivning) til hukommelsen.

Studer først BITPLANE DMA-delen af figuren i BREV II. Figur 2 i BREV IV er et udsnit af hvordan BITPLANE DMA'en opfører sig, når man benytter en LORES, 2 BITPLANE (4-farvers) skærm. De nummerede felter er CYKLUS, som BITPLANE DMA'en bruger for at hente skærmdata fra hukommelsen. Nummeret angiver hvilken BITPLANE, der henter dataene.

De hvide (eller unummerede) felter er ledige CYKLUS, som kan benyttes af hovedprocessoren (MC-68000), COPPER DMA'en eller BLITTER DMA'en. Afstanden fra punktet $38 på figuren til punktet $40 svarer til 16 PIXELs på skærmen. De data, som skal bruges for at vise 16 PIXELs med 2 BITPLANES, er 4 BYTEs-data (en BYTE har som bekendt 8 BITs eller 8 PIXELs),

På et felt henter BITPLANE DMA'en et WORD (2 BYTEs). Som du ser i figuren er der to nummerede felter pr. 16 PIXELs. Det viser, at den vil hente 2 WORDs pr. 16 PIXELs, og det er nok data til at vise 2 BITPLANES.

På denne måde kan du ved hjælp af figuren i BREV II se hvad der sker, når du benytter f.eks. 3 BITPLANEs (8 farver). Som du ser, er linien delt i to på langs af figuren i BREV II. Den øverste del af linien gælder, hvis du er i LORES (320), mens den nederste del gælder, hvis du er i HIRES (640). Som du ser har den nederste del to 1'ere pr. 16 PIXELs bredde, mens den øverste kun har en 1'er. Det er fordi maskinen i HIRES viser dobbelt så mange PIXELs på samme bredde som i LORES.

ALtså, jo flere BITPLANES man opererer med, jo mindre CYKLUS bliver der tilovers til f.eks hovedprocessoren. Maskinen bliver altså langsommere og langsommere for hvert BITPLANE, som aktiveres.

## KOMMENTARER TIL BREV IV

Når man gennemgår et så omfattende emne som AMIGA, så sker det ofte at vi hopper lidt frem og tilbage i vores forklaringer. Du skal derfor af og til tage de tidligere breve frem og opfriske gammel lærdom. Nogle gange må du måske endda vente på næste brev, for helt at forstå det brev du er igang med. At lære at programmere i MC på AMIGA er ikke gjort i en håndevending - det kræver en indsats.

Læs vanskelige afsnit flere gange! Al den information du har brug for, findes der, så hvis noget virker tåget, så læs afsnittet igen..og igen...og...og.

Det er meget vigtigt, at du bruger K-SEKA (eller en anden assembler) og øver dig. Skriv alle eksempler ind og prøv at ændre i dem. Hvis programmet chrasher, så prøv igen. Det er utroligt, hvor meget man lærer af sine egne fejl. Vi har selv lært MC på AMIGA ved at bruge denne "trial and error"-metode. Den fungerede for os, og den vil også fungere for dig.

Hvad du end finder på - GIV IKKE OP!

Hvis du ikke allerede har anskaffet dig KURSUSDISKETTE #1, så er det på tide at gøre det nu. I næste brev er der et meget stort eksempel, som kun findes listet på disketten. Foruden program-eksemplerne indeholder disketten også en IFF-converter, en HUNK-converter og en WAVE-generator; så den er under alle omstændigheder god at have.

Med venlig hilsen
DATASKOLEN

Carsten Nordenhof

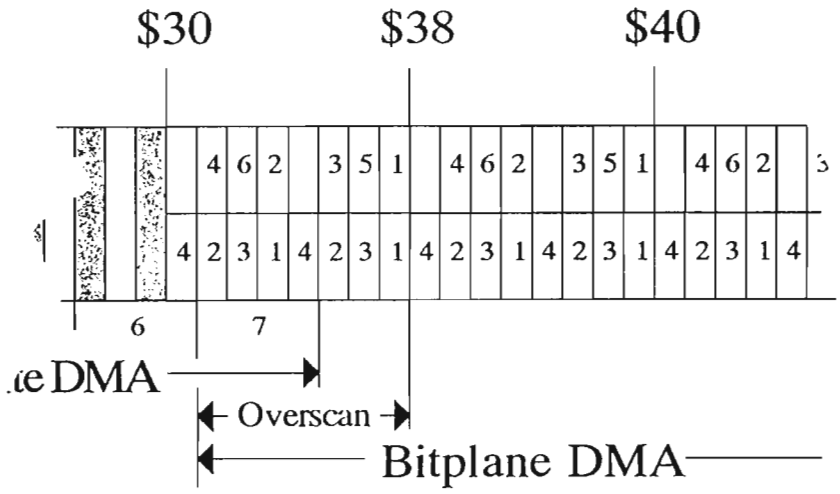
## LØSNINGER TIL BREV III

Opgave 0301: Summen bliver: % 1 0011 0100 (CARRY sat)

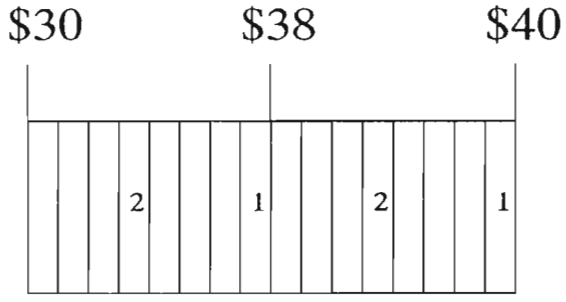
Opgave 0302: Efter en AND: %1000 0110

Opgave 0303: Efter en OR: %1101 1101

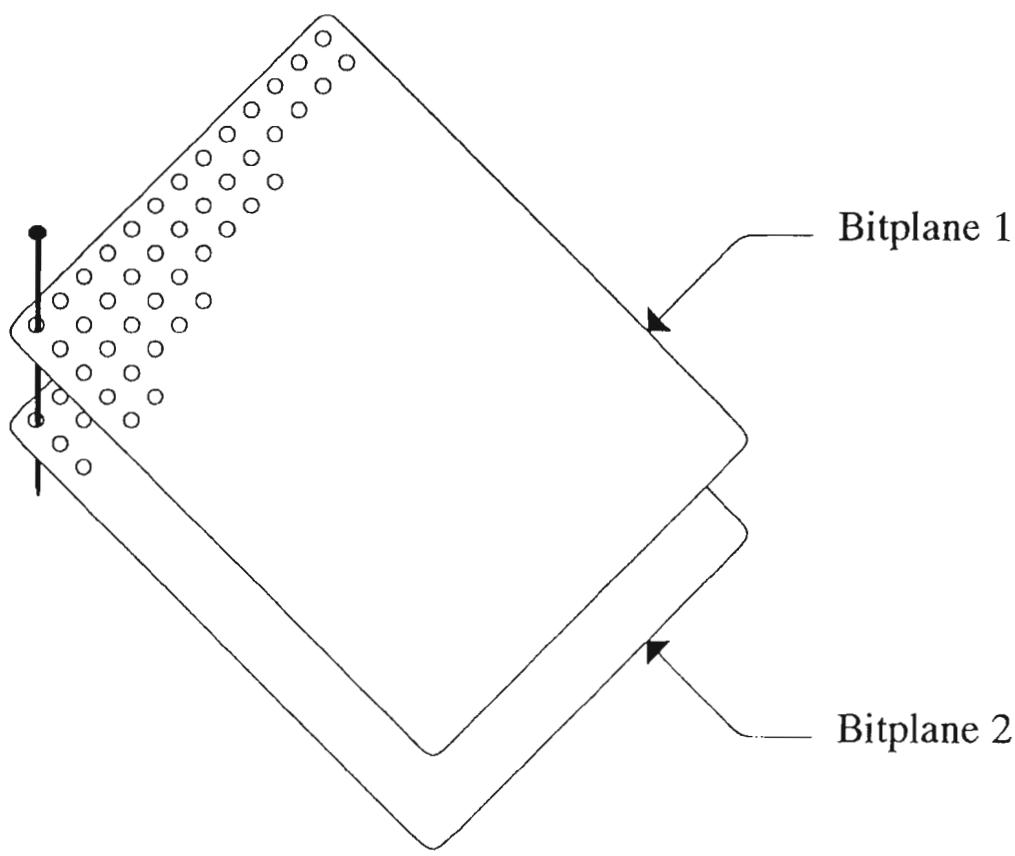
Opgave 0304: Efter en XOR: %0011 0101




Figur 1.




Figur 2.




Figur 3.