Laborationer med MC68 – ML4
Grundläggande assemblerprogrammering

Du ska redovisa dina laborationsresultat vid kontrollstationer. Då du nått en sådan, ska du därför tillkalla handledare som kontrollerar och fyller i följande tabell.

Kontroll station	Uppvisat för:	Godkänt (Datum)	Godkänt (Signatur)
1			
2			
3			
4			
5			
6			
7			

Namn och grupp (skriv ditt namn och linje/klass/grupp tillhörighet här)

Innan du börjar...

Utmatning från bildskärm indikeras i detta häfte med *grå* bakgrund, exempelvis:

```
dbg32:
dbg32: go 4000<Enter>
```

Dessutom används COURIER typsnitt för att ange text som skrivits ut av något program *och* text som *du* ska skriva in. Du skiljer på dessa genom att den text du skriver är understruken.

Även programexempel som visas markeras speciellt på detta sätt exempelvis:

```
      ORG         $4000

loop:
      MOVE.B      ($FFFFF011).L,D0
      MOVE.B      D0,($FFFFF019).L
      BRA         LOOP
```

Text i dubbel ram innebär att du ska *göra* någonting, exempelvis:

Skriv

```
      dbg32: tr<Enter>
```

För att exekvera en instruktion i taget

Kompletterande beskrivningar (handböcker) för laborationskortet *ML4* och laborationsdatorn *MC68* hittar du på Internetadressen: **http://www.gbgmv.se**.

Användning av terminalen

Under detta moment kommer du att lära dig att:

⇒ hur enkel in- och utmatning kan utföras
⇒ hur du använder enkla kommandon hos monitorprogrammet *dbg32*

Du bör ha arbetat dig igenom avsnitt 1 i arbetsboken innan du påbörjar denna laboration

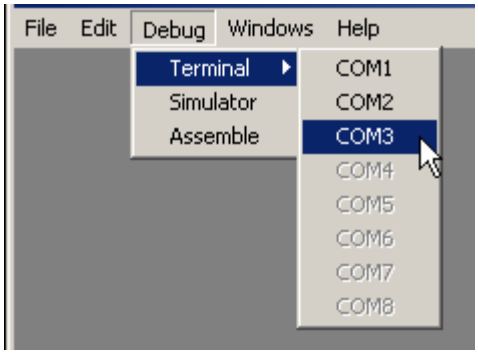
Inledning

ETERM är ett utvecklingssystem för programutveckling i assembler. *ETERM* har utvecklats för undervisningsändamål och fungerar under *Microsoft Windows*. Det finns flera varianter av *ETERM* beroende på vilken måldator (laborationsdator) som används. För detta häfte ska du använda *ETERM 6 för MC68*.

ETERM omfattar funktioner för:

1. **Textredigering**, källtexten skrivs/redigeras med hjälp av en *Editor*, filnamnet ska sluta med *.S68* (source MC68000).
2. **Assemblering**, källtexten översätts till en laddfil som innehåller maskinkoden, med tillägget *.S19* av den inbyggda *assemblatorn*.
3. **Test**, laddfilen överförs till den inbyggda *simulatorn* eller till laborationsdatorn via ETERM's *terminal* med hjälp av respektive programdels laddningsprogram.

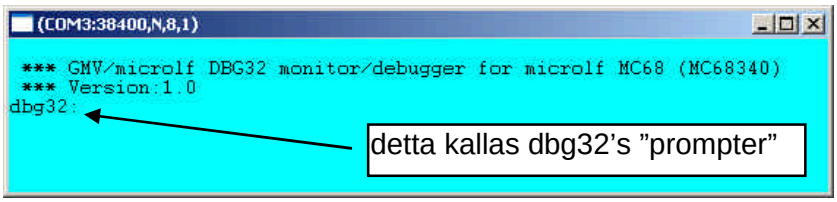
I detta häfte behandlas inledande laborationer med *MC68/ML4*. Du bör ha arbetat dig igenom avsnitt 1 i arbetsboken **innan** du påbörjar denna laboration.



Starta *ETERM* via "Start-Menyn".

Välj **Debug | Terminal** och därefter den serieport som laborationsdatorn är ansluten till för att starta en *terminal* i *ETERM*.

Kontrollera att laborationsdatorn fungerar genom att trycka RESET på laborationsdatorn, du ska då se ungefär följande utskrift i terminalfönstret:



OBS! Du kan alltid trycka på RESET på *laborationsdatorn*. På så sätt *återstartas* denna och prompter'n skrivs ut som en indikation på att laborationsdatorn fungerar korrekt och är redo att ta emot kommandon från dig.

Du skall i allmänhet INTE trycka på person-datorns RESET-knapp.

MC68's inbyggda program identifierar sig. Monitorprogrammet, eller monitor/debugger *dbg32*, är ett hjälpprogram som finns i laborationsdatorns *PROM-minne*. Oftast används endast beteckningen *monitor* för detta program.

PROM-minne är ett *icke-flyktigt* minne, dvs behåller sitt innehåll vid spänningsfrånslag

Monitorn *dbg32* innehåller ett antal programrutiner för att du bland annat ska kunna:

- undersöka och ändra minnesinnehåll (*modify memory*)
- visa eller ändra processorns registerinnehåll (*register modify*)
- starta program (*execute program*)
- dissassamblera minnesinnehåll (*dissassemble*)
- utföra en instruktion åt gången (*trace instruction*)

debug, *testa och felsöka*

Monitorkommandon

Vi skall nu undersöka några av *dbg32*'s olika kommandon, dvs prova några olika programrutiner i monitorn.

Skriv:
`dbg32: help<Enter>`
och studera utskriften

Du ska nu få en hjälptext som visar de olika kommandon *dbg32* accepterar. Läs på nytt igenom punkterna som beskriver programrutinerna i monitorn och försök hitta dom i hjälptexten. Du får ytterligare hjälp om ett speciellt kommando genom att skriva

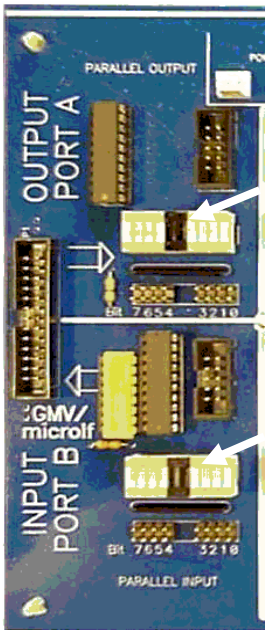
`dbg32: help` ***`kommando-namn`*** `<Enter>`

Ta några minuter och studera:

- "Att visa minnesinnehåll"
- "Att ändra minnesinnehåll"

du kommer strax att ha nytta av det.

Enkel in- och ut- matning



Ljusdioder som visar utdata till *ML4*

Ljusdioder som visar indata från *ML4*

Låt oss nu mata ut några olika bitmönster till *ML4*. Studera laborationskortet *ML4* och lokalisera lysdioderna som visar utdata från laborationsdatorn. Du använder "mm"-kommandot *(modify memory*) för att skriva ut olika bitmönster.

Laborationsdatorns utport mot *ML4* är placerad på adress $FFFFF019, du ska därför ge följande kommando.

```
dbg32: mm -b FFFFF019<Enter>
```

Observera att du utelämnar '$'-tecknet på kommandoraden. *dbg32* tolkar automatiskt detta som ett hexadecimalt tal.

kontrollera att utskriften blir

```
FFFFF019 00:
```

Utskriften innebär att på adress $FFFFF019 finns data $00. Notera också att ljusdioderna för utporten på *ML4* är släckta:

För att tända den högra lysdioden matar vi ut en etta, skriv:

```
FFFFF019 00:1<Enter>
```

Kontrollera att *dbg32* skriver ut data ($01) på denna adressen. Observera att en ljusdiod tänds.

```
FFFFF019 01 :
```

$-tecket framför ett tal anger att talet ges på hexadecimal form

%-tecknet framför ett tal anger att talet ges på binär form.

$ och % kallas i detta sammanhang *prefix*

Hexa-decimala tal	Binära motsvarigheter
$23	%
$70	%
$0A	%
$C5	%
$BD	%

Eftersom *dbg32* accepterar hexadecimala tal kan vi skriva dessa direkt. I marginalen visas en tabell där vissa hexadecimala tal är givna. Använd monitorn för att hitta den binära motsvarigheten till de hexadecimala talen genom att skriva ut dessa till utporten och läsa av ljusdioderna.

För att avsluta inmatningen av data och återgå till *dbg32*'s prompter trycker du bara "." (punkt).

Vi kan även med monitorns hjälp läsa data från *ML4*. Denna port är placerad på adress $FFFFF011.

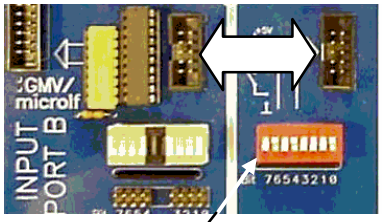
Skriv
`dbg32: mm -b FFFFF011<Enter>`

och *dbg32* svarar:

```
FFFFF011 00 :
```

Orsaken till att laborationsdatorn läser $00 från denna adress är att vi inte ännu ställt *indata*. För att ställa data (ge indata) till inporten måste sektionen *DIP-SWITCH INPUT* anslutas till *INPUT PORT B* på *ML4*.

En 10-polig kabel ansluts mellan sektionerna DIP-SWITCH INPUT och PARALLEL INPUT



Strömbrytare

Anslut en 10-polig flatkabel mellan *DIP-SWITCH INPUT* och *PARALLEL INPUT* på *ML4*

Ställ nu indata genom att ändra strömbrytarna på *DIP-SWITCH INPUT* och tryck <Enter>

```
FFFFF011 00: <Enter>
FFFFF011 xx:
```

Du kommer att läsa den hexadecimala motsvarigheten till det binära värdet du ställde in *DIP-SWITCH* på. Detta är indikerat som "xx"ovan. För att läsa ett nytt värde du ställt in trycker du bara <Enter> på nytt.

Binära tal	Hexadecimala motsvarigheter
%00010010	$
%11000000	$
%10101010	$
%01010101	$
%10111111	$

Fyll i tabellen i marginalen

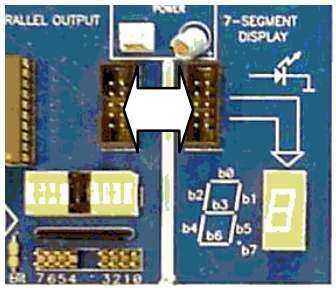
Avsluta mm-kommandot genom att trycka "punkt" på tangentbordet eller tryck RESET på *MC68* och kontrollera att prompter skrivs ut.

7-sifferindikatorn

Här får du lära dig:
⇒ hur olika siffror kan visas på ML4's 7-sifferindikator

Vi skall under detta laborationsmoment studera en 7-*segmentsdisplay* eller kort och gott *sifferindikator*, som kan användas till exempelvis tid- eller temperaturvisning. Här använder vi en enkel mindre sifferindikator som finns på laborationskortet *ML4*.

Här ansluts sifferindikatorn till ML4's inport



Koppla upp *ML4* med laborationsdatorn som ansluts till utvecklingssystemet. Växla till *Terminal* och tryck sedan RESET så att du ser laborationsdatorns prompter.

Anslut en 10-polig flatkabel mellan *OUTPUT PORT A* och *7-SEGMENTS DISPLAY* som figuren i marginalen visar

Du skall nu skriva ut olika bitmönster från laborationsdatorns utport till sifferindikatorn. Använd "mm"-kommandot för att skriva ut olika bitmönster.

Skriv:

```
dbg32: mm -b FFFFF019<Enter>
```

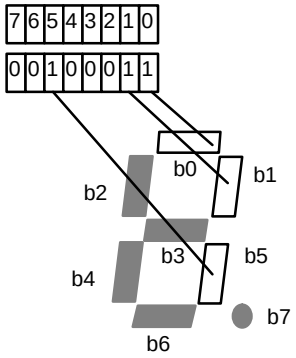
för att "öppna" utporten och sedan

```
FFFFF019 00 : 23<Enter>
```

för att skriva $23 till utporten

Vad ser du på sifferindikatorn?

Bestäm nu vilka bitmönster du måste skriva på utporten för att visa de decimala siffrorna 0-9 och fyll i följande tabell. Testa varje sjusegmentskod du kommit fram till genom att använda "mm-kommandot" och skriva ut siffrorna till sifferindikatorn.
OBS:
Har du arbetat med 'Arbetsboken' har du redan gjort denna uppgift, då kan du föra över resultaten direkt därifrån.



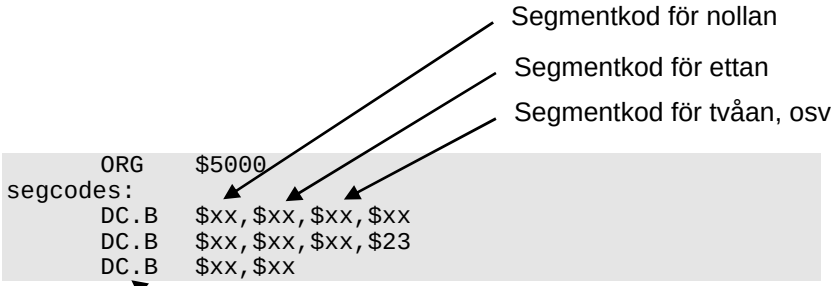
En *etta* tänder ett segment, en *nolla* släcker segmentet.
I vårt exempel skrev vi ut 23 vilket medför att detta bitmönster formar en "sjua" på sifferindikatorn

Decimalsiffra		Sjusegmentskod	
	Binärkod	Binärkod	Hexkod
0	0000		
1	0001		
2	0010		
3	0011		
4	0100		
5	0101		
6	0110		
7	0111	0010 0011	23
8	1000		
9	1001		

Vi skall nu konstruera ett enkelt program som fortlöpande skriver ut de decimala siffrorna till sifferindikatorn. Problemet går att lösa på ett antal olika sätt, exempelvis genom att i tur och ordning använda instruktionsföljden

```
      MOVE.B      #segmentkod_noll,($FFFFF019).L
      MOVE.B      #segmentkod_ett,($FFFFF019).L
      MOVE.B      #segmentkod_två,($FFFFF019).L
      MOVE.B      #segmentkod_tre,($FFFFF019).L
      osv.
```

Det är dock lämpligare att skapa en *programslinga* som skriver ut segmentkoderna en efter en. Vi måste då ange dessa segmentkoder (som konstanter) i någon data-area. Vi deklarerar därför en *tabell* med segmentkoder enligt följande:



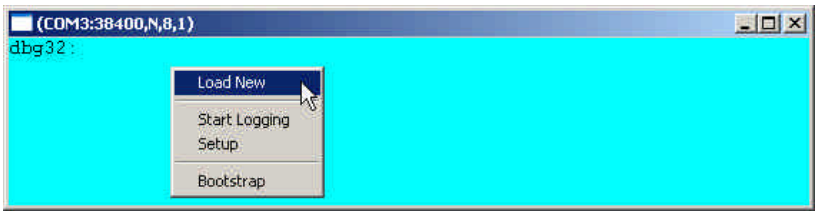

```
      ORG    $5000
segcodes:
      DC.B   $xx,$xx,$xx,$xx
      DC.B   $xx,$xx,$xx,$23
      DC.B   $xx,$xx
```

Assemblerdirektiv *define constant byte*
`DC.B`
används här för att skapa en tabell i minnet

Skapa en källtextfil, `LABMOM1.S68`, med segmentkodtabellen enligt ovanstående exempel. Assemblera och ladda till laborationsdatorn.

Terminalfunktioner

Högerklicka då markören är inom ett terminalfönster. En POPUP-meny visas då...



Här väljer du **Load New** då du vill ladda en fil till målsystemet.

Funktionen **Start Logging**, öppnar en fil och skriver därefter all terminalkommunikation till filen. Du stänger denna genom att högerklicka igen och välja **Stop Logging.**

Funktionerna **Setup** respektive **Bootstrap** kommer du inte att behöva under dessa laborationer.

För att nu studera tabellen i laborationsdatorns minne använder vi monitorn *dbg32*.

Skriv

```
dbg32: dm -b 5000 A <Enter>
```

för att visa tio bytes på skärmen med start på adress $5000. Observera att *dbg32* vid detta kommando tolkar argumentet 'A' som den hexadecimala siffran för 10 (decimalt).

Du bör nu kunna hitta dina 10 segmentkoder i tur och ordning. Skärmutskriften tolkar du som så att på adress $5000 finner du segmentkoden för nollan, på adress $5001 hittar du segmentkoden för ettan, på $5002 segmentkoden för trean osv.

Skriv

```
dbg32: dm -b 4000<Enter>
```

(samma kommando som ovan fast 'A' är utelämnat). Nu visas 256 bytes på skärmen med start på adress $5000.

Kontrollstation 1

Anpassning till realtid

Vi fortsätter nu att arbeta med sifferindikatorn från förra momentet.

Redigera en ny källtext, `LABMOM2.S68` med ett program som matar ut segmentkoderna 0 t.o.m 9 till sju-siffer indikatorn. Placera programmet med start på adress $4000, placera tabellen med start på adress $5000. Assemblera och ladda till laborationsdatorn.

Använd monitorn *dbg32* för att undersöka programmet i minnet.

Skriv

```
dbg32: dasm 4000<Enter>
```

för att dissassemblera minnesinnehållet i laborationsdatorn från adress $4000 och tio instruktioner framåt.

Ser du ett förväntat resultat?

”Dissassemblering”:
I laborationsdatorns monitor/debugger finns funktioner för att översätta maskinkod (nollor och ettor) till assemblerkod (mnemonics).

Funktionerna är användbara för att inspektera och lokalisera programavsnitt.

Skriv nu

```
dbg32: dasm 5000<Enter>
```

för att dissassemblera minnesinnehållet i laborationsdatorn från adress $5000 och tio instruktioner framåt.

Ser du ett förväntat resultat?

”Dissassemblering”:
I laborationsdatorns monitor/debugger finns funktioner för att översätta maskinkod (nollor och ettor) till assemblerkod (mnemonics).

Funktionerna är användbara för att inspektera och lokalisera programavsnitt.

Prova i stället

```
dbg32: dm -b 5000 A <Enter>
```

för att visa tio bytes på skärmen med start på $5000

Studera denna dataarea, du har sett den tidigare. Vad är det?

Dissassembleringsfunktionen kan inte skilja på programkod och data, allt tolkas som om det vore programkod.

Nu är det dags att låta laborationsdatorn utföra programmet. Du tartar programmet på följande sätt:

Skriv
`dbg32: go 4000<Enter>`
för att starta programmet.

Förmodligen visas nu bara en *åtta*. Programmet som ska skriva ut siffrorna 0-9 i tur och ordning visar endast en siffra. Vad kan det vara för fel?.

trace = "spåra"

Låt oss undersöka programmet med hjälp av *dbg32*'s "trace"-funktion som låter laborationsdatorn utföra *en* instruktion i taget.

Tryck RESET och skriv sedan:
`dbg32: tr 4000<Enter>`
för att *exekvera* den första instruktionen

exekvera = utföra

Nu skrivs ett antal rader till skärmen. Dessa visar processorns registerinehåll och sist den instruktion som står i tur att exekveras.

Studera utskriften, betrakta speciellt innehållen i de register du använt för programmet.

Du ska nu exekvera en instruktion i taget och ger därför följande kommando *för varje instruktions* exekvering:

Skriv
`dbg32: tr<Enter>`
För att exekvera en instruktion i taget

Studera speciellt, för varje "`tr`"-kommando, vilken instruktion som står i tur att exekveras. Observera också om sifferindikatoren så småningom tänds med siffran noll som är den första siffran som ska skrivas ut.

inkrementera = öka med 1

Efter ett antal "`tr`"-kommandon kan du slå fast om ditt program trots allt fungerar riktigt, om inte rätta programmet. Då det fungerar korrekt (med "trace"), starta normal exekvering på nytt med kommandot:

Skriv
`dbg32: go 4000<Enter>`

Fortfarande lyser endast en åtta trots att programmet är korrekt. Förklaringen till detta är enkel. Processorn utför helt enkelt instruktionerna så snabbt att den mänskliga hjärnan inte hinner uppfatta skillnaderna mellan 0,1,2,3 osv.

Vi måste därför använda någon form av *fördröjning* mellan varje gång vi matar ut en siffra så att ögat och hjärnan hinner med att uppfatta att en nolla visas, en etta visas, en tvåa visas, osv.

Komplettera programmet med en fördröjningsslinga. Försök att anpassa fördröjningskonstanten så att varje siffra lyser i cirka en sekund.

Redigera, assemblera, ladda ner och testa det kompletta programmet. Kontrollera att programmet fungerar som avsett.

Spara nu filen under ett nytt namn `LABMOM2EVEN.S68`. Ändra i den nya filen så att programmet endast visar *jämna* siffror på sifferindikatorn (siffrorna 0, 2, 4, 6 och 8 skall alltså visas). Testa programmet och rätta eventuella fel så att det fungerar korrekt.

Spara nu filen under ett nytt namn `LABMOM2ODD.S68`. Ändra i den nya filen så att programmet endast visar *udda* siffror på sifferindikatorn (siffrorna 0, 2, 4, 6 och 8 skall alltså visas). Testa programmet och rätta eventuella fel så att det fungerar korrekt.

Kontrollstation 2

Logiska instruktioner

Under detta moment behandlas:

⇒ hur du kombinerar in- och utmatning i ett program
⇒ hur du använder monitorn *dbg32* för att felsöka i program

Du ska under detta moment skriva en programsekvens som läser en inport, bearbetar data från inporten och därefter skriver resultatet till en utport. Bearbetningen innebär här att ett binärtal översätts till en motsvarande segmentkod.

Följande "pseudokod" beskriver programmets funktion

```
do{
      byte=read_inport();       läs inporten
      byte=convert(byte);       översätt till segmentkod
      delay(1s);                vänta en sekund
      write_outport(byte);      skriv till utporten
}forever
```

Anslut en 10-polig flatkabel mellan *DIP_SWITCH* och *INPUT PORT*.
Anslut en 10-polig flatkabel mellan *OUTPUT PORT A* och *7-SEGMENTS DISPLAY*.
Koppla upp laborationsdator, *ML4* och starta *ETERM*. Tryck därefter RESET så att du ser laborationsdatorns prompter.

För att testa uppkopplingen mot inporten och utporten gör du på samma sätt som tidigare. Läs inporten med “mm”-kommandot för att se det inställda bitmönstret på strömbrytarna.

Skriv:

```
dbg32: mm -b FFFFF011<Enter>
```

för att “öppna” inporten och sedan trycker du <Enter> ett antal gånger medan du ändrar strömbrytarna

```
FFFFF011 xx :<Enter>
FFFFF011 yy :<Enter>
FFFFF011 zz :<Enter>
```

För att testa utporten används också “mm-kommandot”. Tänk först på att avsluta “mm-kommandot” med "." (punkt) innan du anger en ny adress.

Tryck

```
FFFFF011 zz :.
```

tryck endast en punkt för att återfå *dbg32*’s prompter och ge därefter kommandot:

```
dbg32: mm -b FFFFF019<Enter>
```

för att “öppna” utporten och skriv ut en sjua på sifferindikatorn med:

```
FFFFF019 xx: 23<Enter>
```

När du testat kommunikationen med in- och utporten och på så sätt verifierat att uppkopplingen är korrekt kan du återgå till programmet.

Decimalt	Binärt
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Som pseudokoden visar skall vi läsa en byte (ett binärtal) från inporten och visa detta som ett decimaltal på sifferindikatorn. Funderar vi lite på hur stora binärtal vi kan läsa från inporten så inser vi direkt att dessa kan vara större än nio vilket är det största decimaltal vi kan visa med en enstaka sifferindikator.

En byte innehåller 8 bitar vilket innebär att binärtalet ligger i området

00000000_2 motsvarar decimalt 0

11111111_2 motsvarar decimalt $2^8-1 = 255$

Vi kan *enbart* använda oss av de fyra minst signifikanta bitarna när vi läser inporten för att kunna översätta det inlästa binärtalet till ett decimaltal mellan 0 och 9.

Studera hur strömbrytarna ser ut i figuren nedan. Vi skall alltså här använda oss av endast stömbrytarna 0, 1, 2 och 3 (eller bit 0-3).

Vi vill nu gardera oss mot att någon (ofrivilligt?) använder strömbrytarna för bit 4 till 7, dessa måste ju alltid vara frånslagna för att resultatet av vår översättning inte ska bli fel. En möjlighet är att helt enkelt "maskera" de fyra övre strömbrytarna (bit 4-7) med maskeringstejp för att ingen skall kunna ställa in för stora tal för oss. I stället för att *fysiskt* maskera dessa på kortet kan vi göra det med hjälp av processorn. Vi använder en bitvis logisk AND-instruktion.

Redigera nu filen `LABMOM3.S68`, skriv ett program enligt beskrivningen ovan, assemblera och ladda ner programmet.

Vi startar programmet i vanlig ordning.

Ge kommandot:
`dbg32 : go 4000<Enter>`

Programmet som löper i en oändlig slinga läser hela tiden strömbrytarna, omvandlar det inlästa binärtalet till en segmentkod och skriver ut denna kod till sifferindikatorn.

Testa nu alla bitmönster *för 0 t.o.m 9* genom att ställa in olika värden på strömbrytarna. Testa också att ändra *de övre fyra bitarna* (bit 4-7). Kontrollera att programmet fungerar som avsett.

Testa nu alla bitmönster *för 10 t.o.m 15* genom att ställa in olika värden på strömbrytarna. Fyll i tabellen i marginalen.

decimalt	binärt	avläst på indikator
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

Avlägsna matningsspänningen till laborationsdatorn under minst 30 sekunder. Koppla på nytt in spänningen, ladda återigen ned programmet och testa på nytt alla bitmönster *för 10 t.o.m 15* genom att ställa in olika värden på strömbrytarna. Fyll i tabellen i marginalen.

decimalt	binärt	avläst på indikator
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

Redogör kortfattat för skillnader/likheter och försök förklara vad detta beror på.

Du ska nu rätta programmet så att hänsyn tas till att alla möjliga bitmönster kan ställas in. Om ett bitmönster som inte kan visas korrekt ställs in ska programmet skriva segmentkoden för bokstaven 'E' (Error) till sifferindikatorn.

Inför rättelser i `LABMOM3.S68`, redigera, assemblera, ladda ner. Testa nu alla bitmönster mellan 1001 ($9) och 1111 ($F) genom att ställa in olika värden på strömbrytarna.

Kontrollstation 3

Villkorligt programflöde

Under detta moment studerar vi:
- ⇒ flaggsättning
- ⇒ hur *test* och *villkorliga* instruktioner kan användas för att styra programflödet.

I detta moment ska du modifiera programexemplet från föregående moment. I programmet ska du lägga till en instruktion som jämför om indata är högre än nio och i så fall utför hopp till en programsekvens som skriver ut tecknet 'E', på sifferindikatorn. Vi kan uttrycka det modifierade programmet med följande pseudokod:

```
do{
      byte=read_inport();       Läs inporten
      if(byte < 10){            Om värdet är mindre än 10
        byte=convert(byte);     så omvandla till segmentkod
        display(byte);          och visa data
      }else                     annars
        display('E');           visa ett 'E'
}forever
```

Det nya i programkonstruktionen är 'if'-satsen. För just detta ändamål finns, i instruktionsuppsättningen, ett antal villkorliga hoppinstruktioner, (Bcc=*branch on condition*). Gemensamt för dessa instruktioner är att flaggorna i statusregistret testas för att avgöra om ett speciellt villkor är sant eller inte. Hoppet utförs om villkoret är sant annars fortsätter exekveringen med instruktionen omedelbart efter branch-instruktionen.
För att använda dig av dessa villkorliga instruktioner måste du också övertyga dig om att flaggorna satts korrekt av en tidigare instruktion. Åtskilliga instruktioner påverkar flaggorna och det är därför viktigt att den instruktion som omedelbart föregår branch-instruktionen väljs med omsorg.

Redigera en källtextfil `LABMOM4.S68` med ett program enligt specifikationen ovan.

Testa programmet på samma sätt som tidigare.

Kontrollstation 4

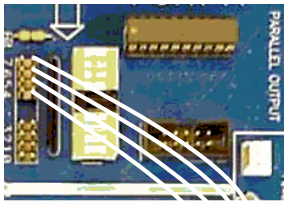
Subrutiner: Styrning av stegmotor

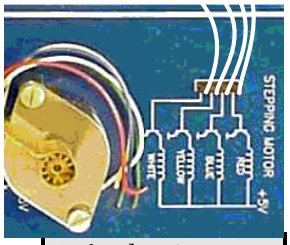
ML4 är utrustad med en stegmotor. Stegmotorn som är avsedd för unipolär drivning kan anslutas med fyra stycken enpoliga kopplingskablar till PORT A's stiftlist SL1. Stegmotorns axel fås att rotera genom att de olika faserna (RED, BLUE, YELLOW och WHITE) styrs ut. Betrakta figuren i marginalen. Faserna ansluts via stiftlist SL4.

Här får du lära dig:
⇒ hur du använder subrutiner
⇒ att styra en stegmotor

Observera att dessa faser är anslutna till +5V. För att få stegmotorn att rotera skall 0V kopplas till två av faserna medan de två andra faserna skall kopplas till +5V. Riktningen som stegmotorn roterar ges av följande tabell.

Stegmotorns rotationsriktning				
Fas	MEDURS ® ¬ MOTURS			
BLÅ	GND	GND	+5V	+5V
GUL	+5V	+5V	GND	GND
RÖD	GND	+5V	+5V	GND
VIT	+5V	GND	GND	+5V





Anslutning av faser:	
RÖD	bit 7
BLÅ	bit 6
GUL	bit 5
VIT	bit 4

Som tabellen visar har vi fyra tillstånd. Vi ansluter först låg nivå (0V) till blå och röd fas samtidigt som faserna gul och vit får en hög nivå (+5V). I nästa tillstånd ändras endast två av faserna, nämligen röd och vit som inverteras. Detta mönstret upprepas hela tiden för att få stegmotorn att kontinuerligt rotera.

Vi ansluter här de fyra faserna till PORT A's höga bitar, b_7-b_4.

Anslut faserna från *SL4* på *STEPPING MOTOR* till *SL1* på *OUTPUT PORT A* med enpoliga kopplingskablar enligt tabellen i marginalen .

Fyll i följande tabell som anger stegmotorns faser för att rotera medurs.

	b7	b6	b5	b4	HEX
Tillstånd 1					
Tillstånd 2					
Tillstånd 3					
Tillstånd 4					

Använd monitorn för att skriva ut några av tillstånden till stegmotorn och kontrollera att den roterar ett steg i taget, skriv:

```
dbg32: mm -b FFFFF019<Enter>
FFFFF019 xx :zz<Enter>
```

och observera att stegmotorn byter tillstånd. (zz är det/de tillstånd du vill ange)

Du ska nu skriva *ett program* som får stegmotorn att rotera medurs. Se följande exempel.

Du kan ange tillstånden i programmet som hexadecimala tal eller som binärtal. Om det är ett hexadecimalt tal så inleds detta med $-tecknet. Använder du binärtal skall du använda %-tecknet som prefix.

```
            ORG         .....       startadress
*Definiera symbol för portadress
ML4_OUT     EQU         ......

loop:
            MOVE.B      #Tillst_1,(ML4_OUT).L
            MOVE.B      #Tillst_2,(ML4_OUT).L
            MOVE.B      #Tillst_3,(ML4_OUT).L
            MOVE.B      #Tillst_4,(ML4_OUT).L
            BRA         loop
```

Här skriver du in de olika tillstånd som du fyllt i tabellen tidigare

Redigera programmet, använd filnamnet `LAB5_1.S68`. Assemblera och ladda ner.

Nu skall vi testa detta program. Använd *först* "trace"-funktionen hos monitorn *dbg32*.

Skriv:

```
dbg32: tr 4000<Enter>
```

och observera att stegmotorn ändrar sig. Ge sedan ett antal tr-kommandon enligt

```
dbg32: tr<Enter>
dbg32: tr<Enter>
```

Fungerar nu detta? Vrider stegmotorn sig ett steg i taget? Om inte så bör du nu kontrollera vad lysdioderna för b_7, b_6, b_5 och b_4 på *OUTPUT PORT A* visar.

Rätta eventuella fel så att ditt program blir korrekt. Stegmotorn skall vrida sig ett steg för varje "tr"-kommanodo du ger.

Därefter provar du "go"-kommandot.

Skriv:

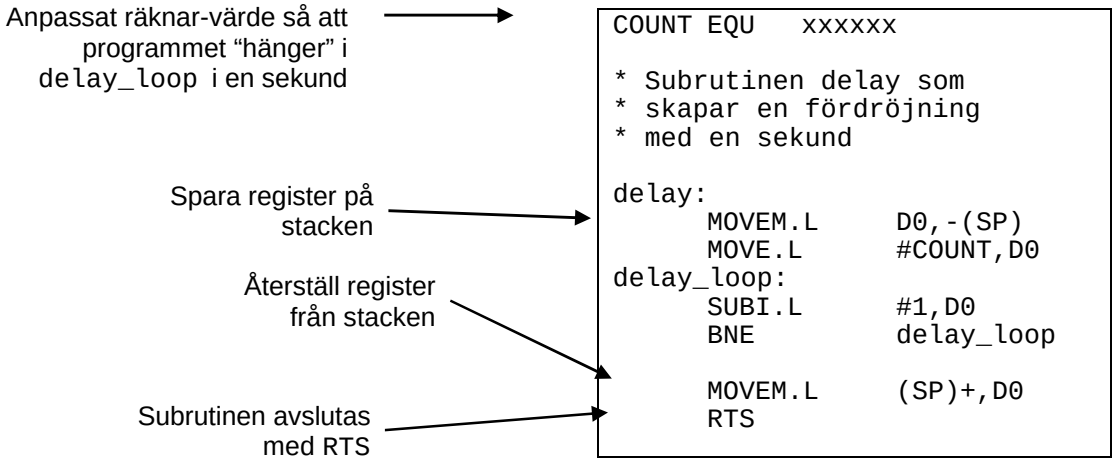
```
dbg32: go 4000<Enter>
```

för att starta programmet .

Vad händer nu? Troligen ingenting. Du har stött på detta problem tidigare. Utöka nu programmet med en *delay-rutin* för att skapa en nödvändig fördröjning. Detta rättar till problemet....

```
do{
      write_outport(Tillst_1);
      delay();
      write_outport(Tillst_2);
      delay();
      write_outport(Tillst_3);
      delay();
      write_outport(Tillst_4);
      delay
}forever
```

Skapa en subrutin för rördröjningen enligt följande:



Lägg till i `LABMOM5_1.S68` enligt ovan och testa.

Ändra konstanten `COUNT` i delayrutinen så att du får ungefär en sekund mellan varje nytt tillstånd för stegmotorn.

Spara därefter filen under det nya namnet `LABMOM5_2.S68` och fortsätt arbeta med denna fil.

Vi fortsätter nu att lägga till subrutiner till programmet. Följande lilla rutin tar en parameter (D0.byte) och skriver parameterns värde till *ML4_OUT*. Lägg till subrutinen i filen `LABMOM5_2.S68`.

```
* subrutin write_outport(value)
* skriver värdet i D0.b till ML4 utport
* inga register ändras
write_outport:
      MOVE.B      D0,(ML4_OUT).L
      RTS
```

Skriv en subrutin `sleep(value)` som tar en parameter (D0.L) multiplicerar denna med 10ms och åstadkommer denna fördröjning.

Exempelvis ska följande anrop:

```
      MOVE.L      #300,D0
      BSR         sleep
```

resultera i en 3 sekunder lång fördröjning
Anmärkning: För att åstadkomma en fördröjning om 10 ms kan du använda konstanten COUNT som du bestämt tidigare. Definiera helt enkelt en ny konstant enligt:

```
CD10ms      EQU   COUNT/100
```

Skriv ytterligare en subrutin `read_inport()` som läser av inporten och returnerar detta värde i D0.b.

Vi ska nu också konstruera en subrutin `next_state()` som bestämmer nästa tillstånd för att vrida stegmotorn ett steg medsols. Tillståndet ska returneras i D0.b. Eftersom stegmotorn kräver att dessa tillstånd ges i en speciell ordning måste vi alltså åstadkomma en subrutin med någon form av "minne", dvs det tillstånd som rutinen ska returnera beror av det föregående tillståndet. Ett sätt att göra detta är att använda en global variabel, vi kallar den `state_index`, och denna variabel kan anta värdena 0,1,2,3 (ett värde för varje tillstånd).

Vi definierar subrutinen i pseudospråk:

```
initialt värde indexvariabel
state_index = 0;
konstant vektor
state_vector={Tillst_1,Tillst_2,Tillst_3, Tillst_4};

next_state:
      next_state = state_vector[state_index];
      state_index = state_index+1;
      state_index = state_index & 3;    modulo 3
```

Vi allokerar (upplåter) utrymme för variabeln `state_index` med hjälp av direktivet *define storage*.

```
state_index:
      DS.B  1     allokera en byte till variabel
state_vector:
      DC.B  Tillst_1,Tillst_2,Tillst_3,Tillst_4
      ALIGN
* Notera: Assemblerdirektivet ALIGN måste användas
* här eftersom vi annars riskerar att subrutinens
* första instruktion hamnar på udda adress (Detta
* klarar inte MC68xxx).

next_state:
      MOVEM.L     D1/A0,-(SP)
      MOVE.B      (state_index).L,D0
      MOVEA.L     #state_vector,A0
      MOVE.B      D0,D1
      ADDI.B      #1,D0
      ANDI.B      #3,D0       modulo 3 add...
      MOVE.B      D0,(state_index).L
      MOVE.B      (A0,D1),D0
      MOVEM.L     (SP)+,D1/A0
      RTS
```

Fortsätt redigera filen LABMOM5_2.S68, lägg till rutinen next_state (och variabeldeklarationer). Assemblera och rätta ev. fel tills programmet är syntaktiskt riktigt.

Test av subrutin

För att testa subrutinerna används nu följande huvudprogram

```
**********************************
* MOM8B.S68
* Test av subrutinpaket...
**********************************
      ...               Definitioner
      ORG   $4000
main:
      NOP               "no operation"
      BSR   (subrutin under test...)
      BRA   main
      ....
**********************************
```

Använd *dbg32's trace*-funktion. För att utföra en hel subrutin (exempelvis "sleep") kan du använda kommandot `go -n` (go next). Detta kommando ska du då ge när "nästa instruktion" är BSR *sleep*.

dvs ge först kommandot

`dbg32: tr 4000<Enter>`

instruktionen `NOP` utförs nu. Ge därefter kommandot

`dbg32: go -n<Enter>`

för att utföra hela den subrutin som testas

Varvtalsreglering

Vi ska nu avslutningsvis använda de konstruerade subrutinerna. Följande algoritm ska implementeras i ett huvudprogram:

- Läs ML4's inport
- Utför fördröjning (inportens värde*0,01 sek)
- Utför ett steg hos stegmotorn
- Repetera

För små värden på inporten blir alltså fördröjningen liten och därmed varvtalet högt. För stora värden på inporten får vi det omvända.

Huvudprogrammet kan också beskrivas med följande pseudokod

```
state_index = 0;
do{
      byte = read_inport();
      sleep(byte);
      byte = next_state();
      write_outport(byte);
}forever;
```

Redigera nu ett komplett program för varvtalsreglering i en ny källtextfil `LABMOM5_3.S68`, assemblera och testa programmet.

Kontrollstation 5

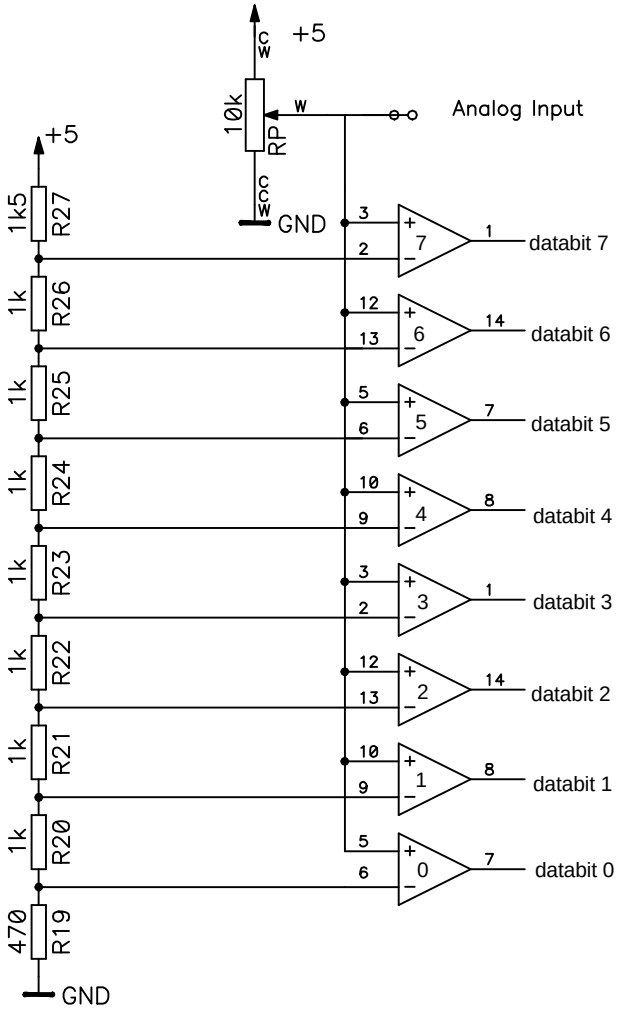
Analog/Digital omvandling

ML4-kortets A/D-omvandlare är av typen FLASH-omvandlare. Ett schema över omvandlaren visas nedan. Du ställer in den analoga inspänningen genom att ändra potentiometern *RP*.

Under detta moment får du:

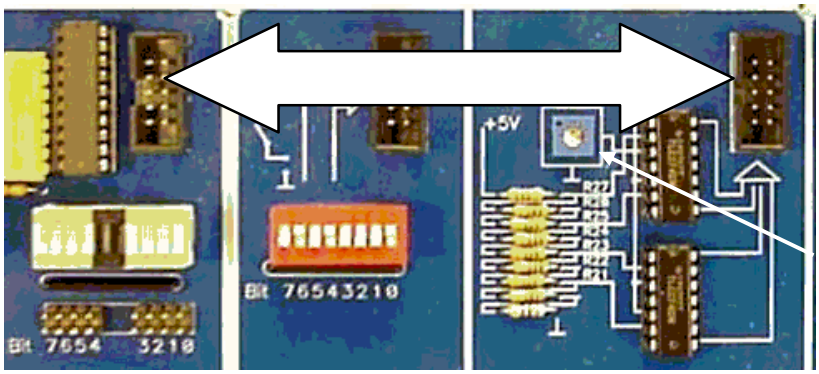
⇒ lära dig att testa program med användning av så kallade *brytpunkter*.

⇒ se exempel på hur snabb A/D-omvandling kan utföras.



Studera operationsförstärkare 0. Om du ställer in den analoga insignalen till 0V kommer denna (och de övriga operationsförstärkarna) att leverera 0V ut då alla referensspänningarna har en högre nivå och då dessa är anslutna till minusingångarna på operationsförstärkarna. På så sätt kommer alla databitarna att vara noll från A/D-omvandlaren.

När du höjer den analoga insignalen till en nivå som är högre än exempelvis referensspänning för operationsförstärkare 4 kommer denna att leverera +5V ut. Detta gäller även operationsförstärkare 0, 1, 2 och 3 då insignalen nu är mycket högre än deras referensspänningar. Detta medför att databitarna 0 - 4 är ettställda och databitarna 5, 6 och 7 är nollställda.



Anslut en 10-polig flatkabel mellan *INPUT PORT B* och *A/D-CONVERTER*.

Använd en liten skruvmejsel och skruva försiktigt på potentiometern för att ändra spänning...

Vi kan verifiera vårt föregående resonemang genom att studera utdata från A/D-omvandlaren som nu visas på ljusdioderna.

Testa A/D-omvandlaren genom att skruva på potentiometern och studera ljusdioderna.

Använd även monitorn för att läsa A/D omvandlaren.

Skriv:

```
dbg32: mm -b FFFFF011<Enter>
```

Ändra potentiometern och tryck `<Enter>` på nytt.

Utdata från AD-omvandlaren	motsvarighet
00000000	0
00000001	1
00000011	2
00000111	3
00001111	4
00011111	5
00111111	6
01111111	7
11111111	8

Du bör nu fått en uppfattning om hur A/D-omvandlarens utdata ser ut. Med hjälp av en tabell hittar vi snabbt den *binära* motsvarigheten till de *bitmönster* vi avläser från A/D-omvandlaren (se marginalen).

Hur skall vi nu gå tillväga för att omvandla bitmönstret `00011111` till talet 5 med datorns hjälp? Omvandlingen från bitmönstret som läses från A/D-omvandlaren till den decimala motsvarigheten kan göras på flera olika sätt. Här väljer vi en metod som innebär att programmet räknar antalet ettställda bitar i bitmönstret. Följande algoritm utför just detta, observera dock att resultatet av algoritmen blir *antal 1-ställda bitar* + 1.

```
number = 0;
byte=read_inport();
do{
      number = number +1;
      BIT = shift_right(byte);
}until BIT == 0
```

Här utnyttjar vi en *skift-instruktion* hos processorn, LSR (logical shift right). Betrakta följande exempel på hur instruktionen fungerar:

Bitmönstret	`0 0 0 1 1 1 1 1`	skiftas till höger
så att resultatet blir	`0 0 0 0 1 1 1 1` ⇒ `1.`	

Ettan till höger om pilen indikerar utskiftad bit *och denna placeras i Carry-flaggan*.

Vi kan därför använda en skiftinstruktion tillsammans med en villkorlig hoppinstruktion (branch) som utför ett hopp beroende på om carryflaggan är nollställd eller inte.

Betrakta följande förslag till subrutin:

```
read_AD:
      CLR.L       D0                  Antal ettor = 0
      MOVE.B      (ML4_IN).L,D1       Läs A/D-omvandlaren
read_AD_loop:
      ADDI.B      #1,D0               Antal ettor +1
      LSR.B       #1,D1               Skifta bitmönstert 1 steg höger
      BCS         read_AD_loop        Hoppa om utskiftad bit = 1
      RTS
```

Observera hur den givna rutinen lämnar värdena 1 till och med 9 i stället för värdena 0 till och med 8.

Ett enkelt huvudprogram som använder subrutinen kan se ut enligt:

```
ML4_IN      EQU   ...
START       EQU   ...

            ORG   START
main:       BSR   read_AD
            BRA   main


read_AD:    ...   subrutinen följer här
```

Redigera en ny källtext `LABMOM6.S68`, med huvudprogram och subrutin enligt ovanstående. Assemblera och rätta ev. fel, Ladda ned och testa programmet genom att använda "trace"-kommandot.

Som du säkert märkt så är det förhållandevis tidskrävande att testa ett program med "trace"-kommandot. Vi kan i stället utnyttja brytpunkter (*breakpoints*). Vi kan med denna funktion få processorn att exekvera till den kommer till en bestämd adress i programmet.

I vårt exempel är det *slutresultatet* efter att programslingan är klar som är intressant dvs. variabeln "antalet ettor" som lagras i register D0. Denna variabel kan undersökas när vi kommit till instruktionen `RTS.`

Skriv:

`dbg32: dasm 4000 <Enter>`

och anteckna adressen för `RTS`-instruktionen i subrutinen.

Adress:_____

För att "sätta" en brytpunkt skriver du:

```
dbg32: bp 0 set Adress<Enter>
```

Detta innebär att du sätter brytpunkt nummer 0 på adress *Adress*, som nu anger adressen för `RTS`-instruktionen. Nu kan du på nytt starta programmet och exekvera detta fram till `RTS`-instruktionen direkt.

Skriv:

```
dbg32: go 4000<Enter>
```

för att starta programmet

Programmet stannar direkt på den önskade brytpunkten och visar registerinnehållen. Du kan nu studera vad register D0 innehåller och därmed kontrollera vilket värde variabeln "Antal ettor" har.

För att ta bort en brytpunkt skriver du:

```
dbg32: bp 0 rem<Enter>
```

Sätt denna gång en brytpunkt på `BRA`-instruktionen.
Starta programmet på nytt med "go"-kommandot efter det att du ändrat potentiometern på A/D-omvandlaren och testa några olika värden.

Kontrollstation 6

Digital/Analog omvandling

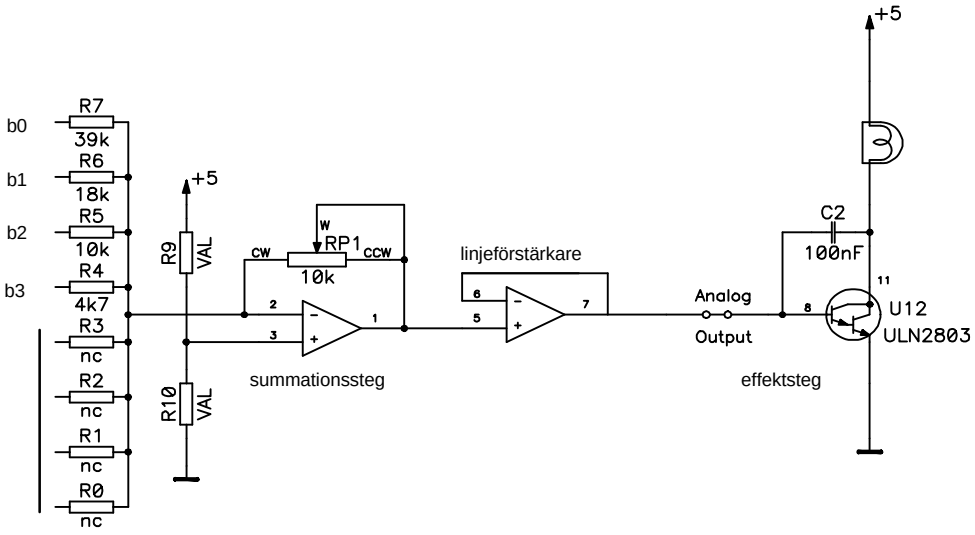
I detta avslutande moment visas hur digital representation kan omvandlas till en kontinuerlig utspänning. För att indikera utspänningen används glödlampan på *ML4* och som insignal använder vi A/D-omvandlaren från föregående moment. Detta innebär att vi konstruerar en “steglöst variabel belysning”.

D/A-omvandlaren består av tre block:

Summationsförstärkaren används för att summera bidragen från den digitala utsignalen. Endast de fyra *minst signifikanta bitarna* används. Logiken är *negativ*, dvs ett logiskt värde 0 motsvaras av bitmönstret %1111, ett logiskt värde 1 motsvaras av bitmönstret %1110, osv

Linjeförstärkaren isolerar summationsteg från effektsteg.

Effektsteget driver glödlampan.



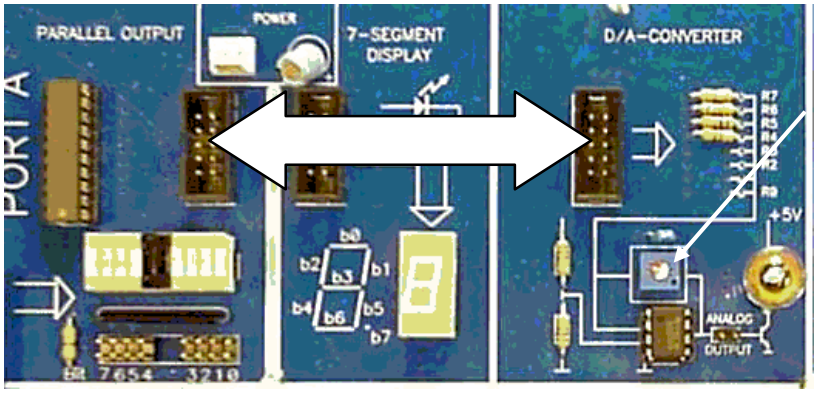
Som figuren ovan visar är endast 4 bitar anslutna till D/A-omvandlaren. Detta innebär att vi kan ge de digitala värdena 0000 till 1111 som insignal. Studera nu kopplingen och fundera ut när lampan lyser.

Lyser lampan vid insignalen 0000 eller 1111?

Koppla nu upp ML4-kortet.

Anslut sektionen
PARALLEL OUTPUT till *D/A-CONVERTER*

Anslut sektionen
PARALLEL INPUT till *A/D-CONVERTER*



Här kan D/A omvandlarens förstärkning justeras

Då glödlampan blir varm ändras resistansen hos denna en aning. Dessutom finns *spridning* hos komponenterna i kopplingen och därför måste *förstärkningen* justeras. Det kan då också bli aktuellt att upprepa justeringen efter en stund.

Justering av D/A-omvandlaren.
Använd monitorprogrammet för att skriva ut bitmönster till utporten.

Skriv bitmönstret %11111110 till utporten:

```
dbg32: mm -b FFFFF019 FE<Enter>
```

Justera RP1 så att glödtråden lyser mycket, mycket svagt.

Skriv bitmönstret %11111111 till utporten:

```
dbg32: mm -b FFFFF019 FF<Enter>
```

Glödtråden ska nu slockna helt, om inte upprepa hela förfarandet

Med detta är D/A-omvandlaren justerad. Prova att skriva ut andra bitmönster till D/A-omvandlaren. (Kom ihåg att endast de fyra minst signifikanta bitarna används.)

Redigera en ny källtextfil `LABMOM7.S68` och skriv ett huvudprogram som kontinuerligt läser av A/D-omvandlaren och reglerar ljusintensiteten hos glödlampan via D/A-omvandlaren. Ljusintensiteten ska vara störst för det maximala värde som A/D-omvandlaren levererar.

```
do{
      byte=read_AD();
      byte = ! byte;          invertera
      write_outport(byte);
}forever
```

Redigera, assemblera och ladda programmet. När du nu testar programmet så ändra potentiometern på A/D-omvandlaren för att ändra glödlampans ljusintensitet.

Undersök programmet och eller läs av på utportens lysdioder för att undersöka vilka värden som skrivs till D/A-omvandlaren.

Vilket MAX-värde skrivs till D/A-omvandlaren?

Vilket MIN-värde skrivs till D/A-omvandlaren?

Lampans ljus som drivs av D/A-omvandlaren kommer att stegvis ändra styrka när du ändrar potentiometern. På detta (stegvisa) sätt ändras utsignalen från alla D/A-omvandlare.

Om *ML4*-kortet vore utrustad med en D/A-omvandlare med högre *upplösning* dvs flera bitar som påverkar D/A-omvandlarens ingång, skulle den stegvisa ändringen i lampans sken inte vara så märkbar.

Kontrollstation 7