# Laboration A
# TSEA57 Datorteknik I

Anders Nilsson

2015 version 1.1

# Innehåll

# 1 Introduktion

## 1.1 Syfte

Avsikten med den här laborationen är att du ska bekanta dig med Tutor, både hårdvaran och mjukvaran (själva Tutor-systemet), samt även att få en viss kopplingsvana med labutrustningen. Tutor är ett enkelt mikrodatorsystem uppbyggt kring processorn M68008, och är det system som används i kursens alla laborationer.

## 1.2 Förkunskaper

För att på ett bra sätt tillgodogöra sig laborationen behöver man ha grunderna från tidigare digitalteknikkurs. Man behöver känna till digitala elektriska egenskaper, såsom hög och låg, och även förstå enkla booleska operationer. Däremot kommer vi inte att syssla med Karnaughdiagram, räkna med booleska ekvationer eller dylikt.

## 1.3 Bakgrund

Processorn 68008 från Motorola introducerades 1982 och slutade tillverkas 1996. Den är med andra ord ganska gammal (för att vara en processor i alla fall), men också seglivad då den används här och var, t ex i den här kursen. Skälet till det är helt enkelt att det är en lagom enkel processor att lära sig grunderna på jämfört med dagens betydligt mer moderna flerkärniga och komplexa diton. Ha det i åtanke om ni tycker att 68008 verkar krånglig.

## 1.4 Material

Allt material som behövs till laborationen finns i labsalen, även labinstruktionen (denna skrift), fast bara då i elektronisk form som PDF-fil från kursen websida. Vill man ha en tryckt kopia så får man skriva ut den själv. Detta är dock inte nödvändigt för att kunna utföra själva laborationen.

## 1.5 Förberedelser

Som förberedelse behöver du ha läst igenom denna laboration. Det är också bra att bekanta sig med Tutor-systemets manual, och orientera sig med vad som finns där. Manualen finns som en separat PDF-fil, tillgänglig via kursens websida under kursmaterial.

# 2 Kom igång

## 2.1 Förbered miljön

**Logga in** på en av labsalens Linux-datorer med ditt LiU-ID. OBS, det du sparar på ditt konto är endast åtkomligt från ISY's datorer, inte hela LiU's datorsystem.

**Ladda labbmodulen** genom att skriva ett kommando i ett teminalfönster. Ett terminalfönster öppnas lättast genom att högerklicka i bakgrunden och välja "**Open Terminal**" ur menyn. Skriv sedan kommandot "**module add BUSSENLAB**" i terminalfönstret. Labbmodulen ger tillgång till nödvändiga kommandon för access till Tutor och för att kunna assemblera programkod.

```
bussen> module add BUSSENLAB
bussen>
```

OBS, om du öppnar flera terminalfönster måste du även ladda labbmodulen i dessa för att få tillgång till nödvändiga kommandon där också.

**Skapa en katalog** i vilken du sedan sparar dom filer som har med just den här labben att göra. Om vi antar att katalognamnet ska vara "**labA**" så kan du i terminalfönstret ge kommandot "**mkdir labA**" för att skapa den. Gå sedan in i katalogen med kommandot "**cd labA**".

```
bussen> mkdir labA
bussen> cd labA
bussen\labA>
```

## 2.2 Starta Tutor

**Slå på spänningsaggregatet** till labbplattan via switchen på spänningsaggregatets front. Aggregatets röda lysdiod ska då lysa och +5 volt likspänning produceras. Om lysdioden inte lyser är aggregatet inte inkopplad till vägguttaget, eller så är 5-volts-utgången på aggregatet kortsluten.

**Starta Tutor** genom att skriva kommandot **tutor.sh** i terminalfönstret. Det räcker faktiskt med att skriva **tut** och sedan trycka på TAB-tangenten så fylls resten i automatiskt.

```
bussen> tutor.sh

WHAT

TUTOR  1.3 >
```

Då ska Tutor starta, och Tutor-prompten visas (enligt ovan). Om detta inte sker, prova att trycka på ENTER-tangenten ett par gånger, kontrollera att Tutor-kortet är anslutet till matningsspänningen och att spänningsaggregatet är påslaget. Fungerar det fortarande inte så kan det vara dags att be om hjälp.

Nu är du inne i Tutor och kan ge ett antal olika kommandon för att undersöka och ändra register i processorn M68008, ändra och titta på minnesinnehållet, skapa och köra program (stegvis eller i full fart) med och utan brytpunkter m m. Mer om detta i nästa avsnitt.

Tutor-miljön är dock ganska primitiv, med tanke på hur moderna utvecklingsverktyg kan se ut. Här finns t ex ingen kommandohistorik (dvs det går inte att trycka *pil-upp* för att återta ett tidigare kommando) och skriver man fel (särskilt med avseende på CTRL-sekvenser) så är det inte säkert att det hjälper att radera med BACKSPACE-tangenten, utan man får då helt enkelt skriva kommandot på nytt.

# 3 Kommandon i Tutor

Syfte: Detta avsnitt visar några enkla kommandon för att titta och ändra på Tutor-systemet minnes- och registerinnehåll. Dessa kommandon blir senare nyttiga för att kunna köra, testa och felsöka program på ett effektivt sätt.

De kommandon som visas i detta avsnitt tillhör dom vanligaste och nyttigaste i Tutor. En komplett redovisning av Tutor-systemets alla kommandon och deras användning finns i systemets manual, tillgänglig på kursens websida.

## 3.1 MD - Memory Display

Syntax : `MD [adress] [antal] [;DI]`

Kommandot **MD** visar minnesinnehållet i Tutor för en angiven adress. Prova kommandot **MD 1000**:

```
TUTOR  1.3 > MD 1000
001000    42 00 55 6F 31 32 20 00  00 1A 00 29 08 41 42 00  B.Uo12 ....).AB.

TUTOR  1.3 >
```

Resultatet visas som en rad av siffror bokstäver och andra tecken. OBS, att det *du* får som resultat kan se annorlunda ut. Vad betyder då allt detta? Jo, till att börja med visas allt i hexadecimal form, och dom numeriska argument som ges till kommandon förutsätts också vara hexadecimala. Dvs, adressen 1000 ovan är alltså hexadecimal, m a o inte adress *ett tusen* utan något annat (nämligen 4096 decimalt, ty $1000_{16} = 16^3 = 4096_{10}$).

Dom följande 16 hexadecimala tvåsiffriga talen är alltså minnesinnehållet på adress 1000 och framåt. Dvs adress 1000 innehåller 42, adress 1001 innehåller 00, adress 1002 innehåller 55 o s v till adress 100E som innehåller 42 och adress 100F som innehåller 00.

Sist på raden kommer ett antal (16 st) punkter och andra tecken. Dessa visar minnesinnehållet på nytt, fast nu såsom ASCII-tecken. T ex så innehåller adresserna 100D och 100E dom hexadecimala värdena 41 och 42, vilka motsvaras av ASCII-tecknen A respektive B. Alla kombinationer av tvåsiffriga hexadecimala tal har dock inte något motsvarande skrivbart ASCII-tecken, så i dessa fall visas en punkt istället.

Syntaxen för kommandot MD visar att det kan ta flera argument, enskilt och i kombination med varandra. T ex kan man skriva på följande sätt:

```
TUTOR  1.3 > MD 1000 20
001000    42 00 55 6F 31 32 20 00  00 1A 00 29 08 41 42 00  B.Uo12 ....).AB.
001010    00 68 02 32 00 43 02 00  00 85 00 1C 90 2E 00 00  .h.2.C..........

TUTOR  1.3 >
```

Det andra argumentet till MD (`20`) anger antal adresser som ska visas. I det här fallet $20_{16} = 32_{10}$ adresser.

Kommandot MD kan också ta ett tredje argument, `;DI` (*disassemble*), som då tolkar all hexkod som assemblerinstruktioner och visar dom som sådana.

```
TUTOR  1.3 > MD 1000 ;DI
001000    4200                CLR.B   D0

TUTOR  1.3 >
```

Tydligen motsvarar dom hexadecimala koderna `42 00` instruktionen `CLR.B D0` (dvs nollställ byten i register D0). OBS, igen, att du kan ha ett annat resultat.

Om du nu bara trycker på ENTER-tangent så upprepas kommandot för dom efterföljande adresserna på så sätt att 16 programrader skrivs ut.

```
TUTOR  1.3 >
001002    556F31B2            SUBQ.W  #2,12722(A7)
001006    2000                MOVE.L  D0,D0
001008    001A0029            OR.B    #41,(A2)+
00100C    0841                DC.W    $0841
00100E    4200                CLR.B   D0
001010    006802A2005C        OR.W    #674,92(A0)
001016    02000085            AND.B   #133,D0
00101A    001C                DC.W    $001C
00101C    902E0000            SUB.B   0(A6),D0
001020    0076                DC.W    $0076
001022    00B8018C000000AB    OR.L    #25952256,$000000AB
00102A    00EF                DC.W    $00EF
00102C    4000                NEGX.B  D0
00102E    000000A3            OR.B    #163,D0
001032    00400020            OR.W    #32,D0
001036    00000013            OR.B    #19,D0

TUTOR  1.3 >
```

Här kan flera saker observeras. T ex att *instruktionerna är olika långa*. Instruktionen på adress 1006 är 2 byte lång medan den på adress 1022 är 8 byte lång. Argumenten till instruktionerna anges ibland som hexadecimala tal (då med $-tecken framför talet) och ibland som decimala tal (då utan $-tecken). Vissa adresser innehåller inte *riktiga* instruktioner, utan tolkas bara som konstanter. Dessa skrivs då som `DC.W` (Data Constant av Word-storlek, 16-bitar) följt av ett tal, på adresserna 100C, 101A, 1020 och 102A ovan.

OBS, som vanligt, att resultatet för dig kan se helt annorlunda ut, beroende på att minnesinnehållet i ditt Tutor-system kan vara annorlunda.

## 3.2 MM - Memory Modify

Syntax : `MM adress [;DI]`

Med kommandot **MM** kan man ändra minnesinnehållet i Tutor på en angiven adress.

```
TUTOR  1.3 > MM 1000
001000    42 ?C2
001001    00 ?
001002    55 ?.

TUTOR  1.3 >
```

I exemplet ovan visas först innehållet på adress 1000 (vilket är 42) följt av ett ?-tecken. Här kan man nu ange ett nytt värde, t ex C2 som ovan och trycka på Enter. Då visas nästa adress, 1001, och dess innehåll, 00. Genom att bara trycka på Enter utan att ange ett nytt värde så behålls det nuvarande värdet (00 i exemplet ovan). För att avsluta inmatningen av nya värden anger man en punkt (.) och trycker på Enter.

Med kommandot `MD` kan vi nu kontrollera innehållet på adress 1000, och med tilägget `;DI` får vi också se vilket instruktion det nya värdet motsvarar.

```
TUTOR  1.3 > MD 1000 ;DI
001000    C200                AND.B   D0,D1

TUTOR  1.3 >
```

Det tidigare innehållet på adresserna 1000-1001 (42 00) som motsvarade instruktionen `CLR.B D0`, har nu med hexkoden `C2 00` blivit `AND.B D0,D1` istället. På det här sättet skulle det förstås vara möjligt att via hexkoder mata in hela program, men också ganska omständligt och dessutom svårt att lära sig alla hexkoder. Därför finns i Tutor, förstås, möjligheten att direkt skriva in instruktionen i textform. Det sker med tillägget `;DI` till kommandot `MM`:

```
TUTOR  1.3 > MM 1000 ;DI
001000    C200                 AND.B   D0,D1 ? MOVE.W #$3D,D1
```

Här visas nu på angiven adress (1000) den nuvarande instruktionen följt av ett ?-tecken, varefter man kan mata in en ny instruktion, t ex `MOVE.W #$3D,D1` och trycka på Enter. OBSERVERA, att innan den nya instruktionen, direkt efter ?-tecknet, måste man först skriva ett mellanslag (space). I annat fall kommer Tutor protestera, skriva `X?`, och man måste skriva instruktionen på nytt. Gör man på rätt sätt så kommer Tutor i terminalfönstret att skriva över den gamla instruktionen med den nya, och därefter ge möjlighet att ändra på instruktionen på efterföljande adress:

```
TUTOR  1.3 > MD 1000 ;DI
001000    323C003D                      MOVE.W   #$3D,D1
001004    31B22000001A                  MOVE.W   0(A2,D2.W),26(A0,D0.W) ?.

TUTOR  1.3 >
```

Inmatning avslutas med en punkt (.). Märk väl, att den nya instruktionen vi matade in `MOVE.W #$3D,D1` behövde 4 byte i minnet (nämligen `32 3C 00 3D`), medan den gamla instruktionen `CLR.B D0` bara behövde 2 byte (nämligen `42 00`). Med den nya instruktionen har vi alltså inte bara skrivit över innehållet i adresserna 1000-1001 där den gamla låg, utan även dom efterföljande adresserna 1002-1003.

Dvs, det som fanns där, är nu borta. Helst hade vi ju sett att Tutor hade gjort plats för den nya instruktionen och flyttat dom efterföljande framåt i minnet, men Tutor är simpel, och så fungerar det alltså inte. Däremot finns det andra metoder att mata in program där det här löser sig. Vi kommer till det senare.

## 3.3 MS - Memory Set

Syntax: `MS adress data [data ...]`

Kommandot `MS` är ytterligare en instruktion för att ändra minnesinnehållet. Den fungerar dock lite annorlunda jämfört med `MM` och har andra möjligheter. Med `MS` kan man efter angiven adress skriva in flera byte på en gång, även ange textsträngar som data, och när man trycker på Enter så lagras allt på konsekutiva adresser:

```
TUTOR  1.3 > MS 2000 41 4E 44 45 52 53 20 'NILSSON' 00

TUTOR  1.3 > MD 2000
002000    41 4E 44 45 52 53 20 4E  49 4C 53 53 4F 4E 00 40  ANDERS NILSSON.@

TUTOR  1.3 >
```

Med ovanstående kommandon borde du nu t ex kunna ta reda på vilka hexadecimala ASCII-koder som ingår i ditt eget namn. Tyvärr fungerar det inte att använda svenska tecken, såsom Å, Ä och Ö.

## 3.4 DF - Display Formatted registers

Syntax: `DF`

Kommandot `DF visar innehållet i processorns register`.

```
TUTOR  1.3 > DF
PC=00000000 SR=2700=.S7..... US=FFFFFFFF SS=00000786
D0=FFFFFFFF D1=FFFFFFFF D2=FFFFFFFF D3=FFFFFFFF
D4=FFFFFFFF D5=FFFFFFFF D6=FFFFFFFF D7=FFFFFFFF
A0=FFFFFFFF A1=FFFFFFFF A2=FFFFFFFF A3=FFFFFFFF
A4=FFFFFFFF A5=FFFFFFFF A6=FFFFFFFF A7=00000786
-------------000000    600003E6            BRA.L    $0003E8

TUTOR  1.3 >
```

Förutom innehållet i åtta dataregister (D0..D7) och åtta adressregister (A0..A6) visas även bl a programräknaren (PC) och statusregistret (SR).

Sist visas också den instruktion som programräknaren (PC) för tillfället pekar ut, (`BRA.L $0003E8`).

## 3.5 .*RR* - Individual register display/change

Syntax: .*RR* `[data]`

*RR* ska här ersättas med ett registernamn, dvs något av D0..D7, A0..A7, PC, SR, US eller SS.

```
TUTOR  1.3 > .A7
.A7=00000786

TUTOR  1.3 > .A7 7000

TUTOR  1.3 > .A7
.A7=00007000

TUTOR  1.3 >
```

Observera den inledande punkten (.) innan registernamnet. Utan efterföljande data visas bara registrets innehåll. Med efterföljande data sätts registrets innehåll till det datat. På

detta sätt kan man alltså påverka innehållet i processorns olika register. Det är ett effektivt sätt att t ex ge olika indata till ett program utan att behöva skriva om programmet, eller för att påverka programflödet då det beror på innehållet i vissa register.

# 4 Skriva in och köra program

Syfte: Vi har tidigare lärt oss hur man via kommandon kan se och förändra innehåll i Tutors minne och register. Naturligtvis vill vi också kunna göra det via ett program. För att vi ska ha nåt att laborera med kommer vi att skriva ett program som summerar talen från 1 till och med 10 och visar resultatet i ett register.

## 4.1 Programinmatning

Nu skulle man kunna använda kommandot `MM` för att mata in ett program. Det har dock några nackdelar. Det går inte att spara ett inskrivet program, det är omständligt att göra förändringar och det går inte att använda symboliska namn på adresser.

Istället ska vi skriva in programmet som en ren textfil, sedan assemblera denna och slutligen ladda in det assemblerade resultatet i Tutor, där vi sedan kan köra programmet. Enda nackdelen med det sättet är att vi inte kan bestämma var i programminnet som programmet ska hamna, då det alltid kommer att börja på adress $1000_{16}$. Det är dock i praktiken inget problem.

På labsalens datorer finns olika texteditorer installerade, t ex Emacs. Välj någon editor och skriv in följande program, och spara det sedan som `summa1.s`:

```
        CLR.B   D2      ; clear result in D2
        MOVE.B  #1,D0   ; set loop value in D0
LOOP    ADD.B   D0,D2   ; add D0 to result
        ADD.B   #1,D0   ; increase loop value in D0 by 1
        CMP.B   D0,D1   ; compare loop value with end value in D1
        BGE     LOOP    ; if D1 greater or equal, jump to LOOP
        NOP             ; no operation
```

Observera hur programkoden är uppdelad i fyra kolumner. Första kolumnen längst till vänster är reserverad för symboliska namn på hoppadresser, i det här faller endast namnet `LOOP`. I nästa kolumn finns programmets instruktioner, och i kolumnen därefter instruktionernas argument (om sådana finns). Sista kolumen är bara kommentarer, och de är inte nödvändiga för programmets funktion (allt efter ett semikolon fram till slutet på raden tolkas som en kommentar), men kommentarer kan vara nog så viktiga för att man ska förstå ett program.

Kolumnernas bredd har ingen betydelse, bredden kan t o m variera mellan raderna utan problem. Det enda viktiga är att *instruktioner aldrig* står längst till vänster, i absoluta början på en rad och att kolumnerna inte innehåller mellanslag/space (förutom kommentarskolumnen), eftersom mellanslag används för att separera kolumner.

Just det, även om det går bra att skriva in svenska tecken (å, ä och ö) i en texteditor så tycker inte assemblatorn, som vi snart ska använda, om det. Därför är kommentarerna ovan på engelska.

Spara filen med ett lämpligt namn, t ex `summa1.s`, i samma katalog som du startade Tutor.

## 4.2 Assemblera program

Starta ett nytt terminalfönster och ladda labbmodulen med kommandot **module add BUSSENLAB**. Gå till den katalog där du sparat ditt program och använd sedan kommandot **assemble.sh summa1.s** för att assemblera programmet:

```
bussen\labA> module add BUSSENLAB
bussen\labA> assemble.sh summa1.s
bussen\labA>
```

Om allt går bra med assembleringen så får man inga meddelanden. Finns det däremot problem så kommer ett felmeddelande att tala om vad som är fel, vilket kan bero på olika orsaker. Du kan ha skrivit in programmet inkorrekt, du befinner dig i fel katalog, du har inte laddat labbmodulen, eller nåt annat. Hittar du inte felet så be om hjälp.

När allt fungerar som det ska har du fått en resulterande körbar programfil, med namnet `a.out`, och en listfil, med namnet `summa1.s.lst` (förutsatt att filen som assemblerades heter `summa1.s`).

## 4.3 Ladda program till Tutor

För att köra ett program i tutor måste det först laddas in. Förutsättningen för att det ska gå att göra är att du lyckosamt har assemblerat ditt program till en körbar fil (med namnet `a.out`) och att den filen finns i samma katalog som Tutor startades ifrån. I Tutor trycker man sedan tangentkombinationen **CTRL-F L**, dvs först tangenterna **CTRL** och **F** samtidigt och därefter enbart tangenten **L**. Då ska programmet laddas in enligt följande:

```
TUTOR  1.3 >
Nerladdning pågår....
Nerladdningen klar
TUTOR  1.3 >
```

Programmet har nu laddats in med början på adress $1000_{16}$. Kontrollera detta genom att titta på minnesinnehållet från adress $1000_{16}$. Hur gör man det? Om du inte kommer ihåg så backa tillbaka och ta reda på det ;-)

## 4.4 Köra program

Att köra ett nyskrivet program, särskilt om det är för första gången, bör göras med viss försiktighet. Risken är annars att ett litet misstag får det hela att löpa amok. I bästa fall blir då bara resultatet felaktigt, i sämsta fall går nåt sönder. Det gäller framför allt när programmet ska påverka eventuell hårdvara som man kanske också precis har kopplat in för första gången.

Därför börjar vi med att köra programmet under kontrollerade former, stegvis.

### 4.4.1 Stega igenom program

Ett program kan köras stegvis, instruktion för instruktion, genom att initialt sätta programpekaren (`PC`) till önskad startadress och därefter starta stegningen med kommandot **TR** (TRace):

```
TUTOR  1.3 > .PC 1000

TUTOR  1.3 > TR
PHYSICAL ADDRESS=00001000
PC=00001002 SR=2714=.S7X.Z.. US=FFFFFFFF SS=00000786
D0=FFFFFFFF D1=FFFFFFFF D2=FFFFFF00 D3=FFFFFFFF
D4=FFFFFFFF D5=FFFFFFFF D6=FFFFFFFF D7=FFFFFFFF
A0=FFFFFFFF A1=FFFFFFFF A2=FFFFFFFF A3=FFFFFFFF
A4=FFFFFFFF A5=FFFFFFFF A6=FFFFFFFF A7=00000786
--------------------001002    103C0001             MOVE.B   #1,D0

TUTOR  1.3 :>
```

Trace-kommandot har då kört den instruktion som började på adress $1000_{16}$ (`CLR.B D2`) som nollställde byten (de 8 minst signifikanta bitarna) i register D2. Därefter räknas programpekaren (`PC`) upp till adressen för nästa instruktion (i det här fallet adress $1002_{16}$), och statusregistret uppdateras med avseende på vad som hänt.

Hela registerkartan visas med resultatet efter att instruktionen körts, och sist kommer en rad med information om nästa instruktion som är på tur att köras.

Observera också att kommandoprompten har fått ett kolon (:) i sig, vilket indikerar att Tutor är i trace-mode. Dvs, nu räcker det med att enbart trycka på Enter-tangenten för att köra nästa instruktion. Gör det, och observera hur byten i register `D0` sätts till 1.

Om vi nu betraktar vad summa-programmet faktiskt gör så ser vi att det använder tre olika data-register, `D0`, `D1` och `D2`. `D0` är loopräknaren som räknar från 1 och uppåt för dom tal vi ska summera (`D0` sätts till initialvärdet 1 i programmets andra rad). `D2` innehåller själva summan, och `D2` nollställs i programmets första rad. `D1` ska innehålla ett slutvärde (10 i vårat fall), som kan jämföras med loopvärdet i `D0` så att vi vet när vi summerat alla tal från 1 till och med 10.

Men `D1` har inte initierats till något i programmet. Skälet till detta är att vi då själva enkelt kan sätta den gränsen till olika tal utan att behöva skriva om programmet. Kommer du ihåg hur man sätter ett register till ett värde? Om inte. backa och ta reda på det. Sätt sedan register `D1` till 10, *decimalt*. Så vad ska du skriva när du sätter registret?

När du sätter `D1` till sitt värde så går Tutor ur trace-mode. Men det är bara att ge **TR**-kommandot på nytt så fortsätter programmet där det var och kör nästa instruktion.

Fortsätt nu att med Enter-tangenten stega dig igenom programmet instruktion för instruktion (i kanske ett par loopvarv) och **tänk efter** vad som borde hända **innan** du trycker. Annars är det inte stor mening med den här övningen.

—o-O-o—

När du stegat dig igenom programmet ett par loopvarv så börjar du nog snart få en känsla av att det faktiskt fungerar, och det blir tråkigt och tidsödande att bara gå ett steg i taget. Kan man inte åtminstone köra ett helt loopvarv vid varje steg? Jo, det kan man, genom att använda brytpunkter.

### 4.4.2 Köra program till temporär brytpunkt

Man kan köra ett program i full fart och sedan få det att stanna vid en viss adress. Vi ska göra det med kommandot **GT** (Go To) som kör från nuvarande adress fram till den adress man ger som argument till kommandot och sedan stannar där. Så vilken adress bör vi lämpligen köra till? Varför inte gå fram till slutet av loopen, till instruktionen `BGE LOOP`? Så vilken adress finns den på då? Kommer du ihåg hur man tar reda på det? Om du skrivit programmet korrekt ska den finnas på adress $100C_{16}$. Kör sedan kommandot **GT** enligt följande:

```
TUTOR  1.3 > GT 100C
PHYSICAL ADDRESS=0000100C
PHYSICAL ADDRESS=0000100C

AT BREAKPOINT
PC=0000100C SR=2700=.S7..... US=FFFFFFFF SS=00000786
D0=FFFFFF03 D1=0000000A D2=FFFFFF03 D3=FFFFFFFF
D4=FFFFFFFF D5=FFFFFFFF D6=FFFFFFFF D7=FFFFFFFF
A0=FFFFFFFF A1=FFFFFFFF A2=FFFFFFFF A3=FFFFFFFF
A4=FFFFFFFF A5=FFFFFFFF A6=FFFFFFFF A7=00000786
--------------------00100C    6CF8                BGE.S   $001006

TUTOR  1.3 >
```

Genom att upprepade gånger nu skriva kommandot **GT 100C**, så kör programmet ett helt loopvarv i taget, stannar och visar registerkartan. Om du gör det, observera hur loopvärdet i `D0` ökar med 1 var gång. **Men**, det går inte att göra hur många gånger som

helst, i vårat fall. Eftersom programmet bara kommer att gå runt i loopen så många varv som värdet i register `D1` anger. Och vad händer när programmet uppnått det värdet?

Jo, programmet fortsätter. Dvs med nästa instruktion, som i vårat fall är programmets sista instruktion `NOP` (som gör just ingenting). Men vad händer sen?

Processorn stannar inte bara för att vi inte har skrivit in fler instruktioner, utan processorn räknar upp programräknaren (`PC`) till nästa adress, tolkar innehållet där till en instruktion (om det nu går) och kör den (om det nu går). Skulle vi bara låta processorn köra på, okontrollerat, så är det mest troliga att den till slut hamnar på en adress som inte kan tolkas som en instruktion och processorn kraschar med något felmeddelande. Och vad som har hänt fram till dess har vi ju heller ingen kontroll över.

Bäst är alltså att själv bestämma var processorn ska stanna. Förutom att använda en temporär brytpunkt så kan man även göra det med fasta brytpunkter.

### 4.4.3 Köra program till fast brytpunkt

Om vi nu tänker att vi tidigare provat att köra programmet steg för steg, och sett att det troligen fungerar korrekt, så vill vi förstås kunna köra hela programmet i full fart. Men, att det samtidigt stannar när på rätt ställe när det är klart. Det skulle förstås gå att göra med det tidigare kommandot `GT` genom att ange rätt adress, dvs adressen till instruktionen `NOP`. Men finessen med fasta brytpunkter är dels att man kan ha flera på en gång, dels att man inte var gång man skriver ett kör-kommando behöver hålla reda på exakt vilken adress man ska köra till.

Så hur sätter man en brytpunkt? Jo med kommandot **BR**. Antag att vi vill ha en brytpunkt i början av programmet, precis innan loopen, och en i slutet, precis efter loopen. Brytpunkterna skulle då hamna på adresserna $1002_{16}$ respektive $100E_{16}$.

Prova alltså att göra följande:

```
TUTOR  1.3 > BR 1002

BREAKPOINTS
001002    001002

TUTOR  1.3 > BR 100E

BREAKPOINTS
001002    001002
00100E    00100E

TUTOR  1.3 >
```

Med dessa brytpunkter på plats kan vi nu köra programmet i full fart från början med kommandot **GO 1000** (GO from), eftersom programmet börjar på den adressen. Gör det:

```
TUTOR  1.3 > GO 1000
PHYSICAL ADDRESS=00001000

AT BREAKPOINT
PC=00001002 SR=2704=.S7..Z.. US=FFFFFFFF SS=00000786
D0=FFFFFFFF D1=FFFFFFFF D2=FFFFFF00 D3=FFFFFFFF
D4=FFFFFFFF D5=FFFFFFFF D6=FFFFFFFF D7=FFFFFFFF
A0=FFFFFFFF A1=FFFFFFFF A2=FFFFFFFF A3=FFFFFFFF
A4=FFFFFFFF A5=FFFFFFFF A6=FFFFFFFF A7=00000786
--------------------001002    103C0001             MOVE.B  #1,D0

TUTOR  1.3 >
```

Programmet kör fram till första brytpunkten och visar registerkartan. Nu kan vi fortsätta programflödet därifrån vi är genom att bara skriva **GO** utan argument:

```
TUTOR  1.3 > GO
PHYSICAL ADDRESS=00001002

AT BREAKPOINT
PC=0000100E SR=2709=.S7.N..C US=FFFFFFFF SS=00000786
D0=FFFFFF0B D1=FFFFFF0A D2=FFFFFF37 D3=FFFFFFFF
D4=FFFFFFFF D5=FFFFFFFF D6=FFFFFFFF D7=FFFFFFFF
A0=FFFFFFFF A1=FFFFFFFF A2=FFFFFFFF A3=FFFFFFFF
A4=FFFFFFFF A5=FFFFFFFF A6=FFFFFFFF A7=00000786
--------------------00100E    4E71                 NOP

TUTOR  1.3 >
```

Programmet har nu kört alla 10 loopvarv och stannat vid den andra brytpunkten. Nu har vi också fått ett resultat av summeringen, nämligen $37_{16}$ i register D2. Stämmer det? Vad borde det bli? Låt oss räkna ut summan av talen från 1 till och med 10. Det kan enkelt göras med en generell formel på följande sätt:

$$\sum_{i=1}^{k} i = \frac{k(1+k)}{2}$$

Räkna ut summan då k=10. Stämmer resultatet i register `D2`?

—o-O-o—

Nu när vi har ett fungerande program (har du inte det så gå tillbaka och se till att det fungerar) så kan vi experimentera lite med att summera större serier. Börja med att ta

bort alla brytpunkter med kommandot **NOBR** i Tutor. Sätt därefter in en brytpunkt på programmets sista rad, adress $100E_{16}$.

Prova nu att summera talen från 1 till och med 20. Kom ihåg att den övre gränsen för talserien ska skrivas i på hexadecimal form i register `D1`. Vad blev resultatet? Stämmer det?

Prova sedan att summera talen från 1 till och med 30. Stämmer det? Varför inte? Vad kan man göra åt det?

Vid det här laget kanske du börjar tycka att det är krångligt att omvandla tal mellan decimal och hexadecimal form. Du kan då använda labdatorns kalkylator (Applications/Accessories/Calculator) i *programming mode*, eller gå till en websida, t ex http://www.binaryhexconverter.com/ där omvandling mellan olika talbaser kan göras.

# 5 Styra hårdvara

Syfte: När man ska ansluta hårdvara för styrning från ett datorsystem är det bästa att gå stegvis fram och prova delarna var för sig innan man sätter samman allt. Det ska vi göra här, och slutligen koppla in lysdioder och tryckknappar som kontrolleras av ett mindre program.

## 5.1 Konfigurera PIA:n

Till Tutor-systemet kan man koppla in diverse olika labmoduler via de två parallellportarna PA och PB. Titta på labplattan framför dig så ser du dom säkert (eller se figur 5.2). Varje port är 8 bitar bred och varje bit kan fungera som en separat ingång eller utgång.

Portarnas funktion och databitarnas riktining (in eller ut) kontrolleras av en särskild krets i Tutorsystemet, en s k PIA (Periferal Interface Adapter). PIA:n måste konfigureras för att portarna ska få önskad funktion, och detta kan göras med ett litet program. Skapa en ny fil, skriv in följande program och spara det som t ex `led1.s`:

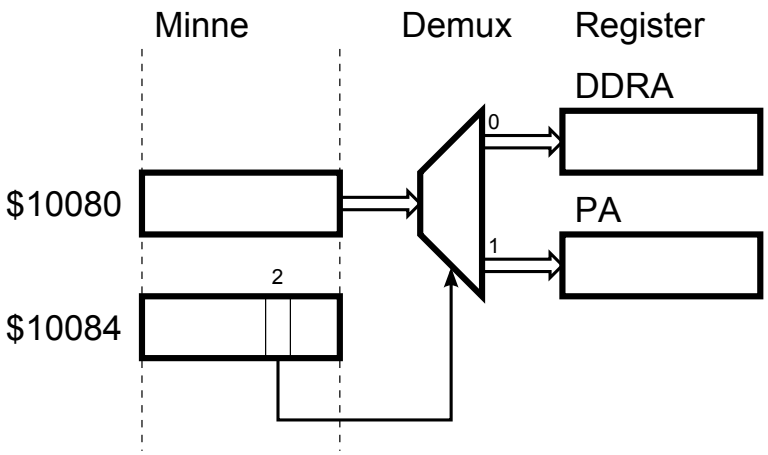
```
CLR.B   $10084          ; select DDRA
MOVE.B  #$FF,$10080     ; set DDRA as output
MOVE.B  #$04,$10084     ; select PIAA

CLR.B   $10086          ; select DDRB
MOVE.B  #$00,$10082     ; Set DDRB as input
MOVE.B  #$04,$10086     ; Select PIAB
```

Utan att gå in på för mycket detaljer så kommer just det här programmet att konfigurera PIA:n så att alla bitar i port A (PA) blir utgångar, och alla bitar i port B (PB) ingångar. Detta kan förstås varieras på många olika sätt via andra värden i programmet, men vi nöjer oss med att det fungerar just så den här gången.

Assemblera, ladda in och *stega* igenom de sex programraderna med trace-kommandot. Eller kör i full fart med brytpunkt på lämplig adress.

Innan vi går vidare behöver vi veta lite om hur PIA:n fungerar. Den har två portar, PA och PB, som båda är *minnesmappade*. Dvs, man kan komma åt dom via vissa adresser i Tutors minnesrymd. PIA-A har dels ett datariktningsregister (DDRA) och dels själva porten (PA).

I figur 5.1 visas hur PIA:n fungerar för porten PA (det fungerar p s s för PB, men då med andra adresser). Bit 2 i adress $10084_{16}$ styr demuxen på så sätt att det som skrivs vid adress $10080_{16}$ hamnar i registret DDRA om bit 2 är en 0:a, eller i registret PA (dvs porten PA) om bit 2 är en 1:a.



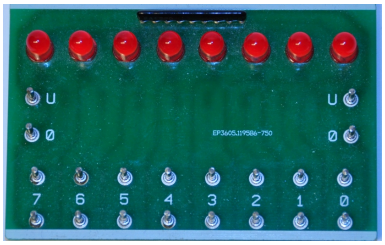

Figur 5.1: PIA-A

Varje bit i DDRA bestämmer om respektive bit i porten PA är en ingång eller utgång, 0:a för ingång och 1:a för utgång.

## 5.2 Koppla in och tända lysdioder

Nu vill vi ju se att det går att använda portarna, så låt oss koppla in några lysdioder till port A. Det finns en modul med 8 lysdioder, se figur 5.3. Egentligen ska vi aldrig koppla när matningsspänningen är påslagen, så börja med att slå av den.


Figur 5.2: Labplatta




Figur 5.3: LED-modul

Vi ska nu koppla samman respektive bit i port A med var sin lysdiod. Varje bit i port A har två anslutningar, så även varje lysdiod, men det räcker med att använda en av dessa för porten respektive lysdioden. Koppla alltså port A bit 0, även kallad PA0, till lysdiod 0, PA1 till lysdiod 1 och så vidare för alla lysdioder.

LED-modulen behöver även matningsspänning. Till detta finns två anslutningar, U och 0 (även dessa i dubbel uppsättning) längs modulens kortsida. Koppla U till 5V på matningsskenan (se figur fig:labplatta) och 0 till 0 på matningsskenan.

Slå på matningsspänningen igen, ladda in och kör programmet på samma sätt som förut.

Nu kan du alltså direkt påverka portens värde genom att ändra minnesinnehållet för adress $10080_{16}$. Prova att göra följande:

```
TUTOR  1.3 > MM 10080
010080    00 ?AA=
010080    AA ?55=
010080    55 ?.

TUTOR  1.3 >
```

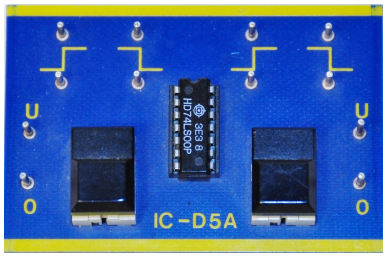
Efter det första värdet `AA` kommer varannan lysdiod att lysa, och efter det andra värdet `55` så blir det tvärtom. Observera =-tecknet efter varje inmatat värde. När man använder =-tecknet under kommandot `MM` så räknas *inte* adressen upp, utan stannar kvar där man är, dvs 10080 i vårat fall. Det är mycket användbart när man vill prova att skriva olika värden till samma port utan att behöva starta om kommandot `MM` varje gång.

Nu bör du kunna ta reda på hur man gör för att tända bara dom två yttre lysdioderna. Vilket värde ska man skriva till porten PA då? Fortsätt sedan med dom två näst innersta, osv ända in till dom två innersta (i mitten). Du bör alltså få fyra olika värden. Följande hexadecimala bitvikter kanske kan hjälpa dig:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Vikt | 80 | 40 | 20 | 10 | 08 | 04 | 02 | 01 |

## 5.3 Koppla in och läsa av tryckknappar

Vi har ju en port till, PB. Den porten har, enligt programmet, alla bitar konfigurerade som ingångar. Så låt oss koppla in ett par insignaler till den. Vi kan använda följande tryckknappsmodul:




Figur 5.4: Tryckknappsmodul

Tryckknappsmodulen har för varje knapp två utgångar (och dessutom två anslutningar per utgång), där den ena utgången går från låg till hög (s k positiv flank) när knappen trycks ned, och den andra utgången då samtidigt går från hög till låg (s k negativ flank).

Anslut utgångarna för dom positiva flankerna till var sin ingång på på porten PB, lämpligen PB0 och PB1. Anslut också matningsspänning till tryckknappsmodulen, +5V till U och GND till 0.

Nu kan vi prova tryckknapparna och se vilket värde dom ger på porten PB, som finns vid adress $10082_{16}$. Prova nu följande med kommandot `MM`. Håll nere någon av tryckknapparna, mata in ett =-tecken (och tryck Enter), släpp upp tryckknappen, mata in ett =-tecken (och tryck Enter). Prova även att göra så med den andra tyckknappen, och båda knapparna samtidigt:

```
TUTOR  1.3 > MM 10082
010082    F8 ?=
010082    F9 ?=
010082    F8 ?=
010082    FA ?=
010082    FB ?
```

Var noga med att alltid ange ett =-tecken, så att vi var gång läser av värdet på adress $10082_{16}$. I annat fall räknas adressen upp.

Tycker du att resultatet verkar vettigt? Med tryckknapparna ska vi ju egentligen bara kunna påverka dom två lägsta bitarna på adress $10082_{16}$, men det kan hända att även övriga bitar varierar, och dessutom verkar dom flesta av dom bitarna ligga höga. Varför då?

Låt bli tryckknapparna för ett ögonblick, lägg ett finger över dom oanslutna ingångarna på PB och läs av ett värde. Ta bort fingret och läs av ett värde igen. Beroende på konduktiviteten (ledningsförmågan) i ditt finger (och kanske din grad av upphetsning just nu) så bör värdet kunna variera ganska mycket. Oanslutna ingångar är alltså mycket lättpåverkade och kan variera okontrollerat.

Så kan vi inte ha det. Ett sätt att bli av med problemet är förstås att koppla alla oanslutna ingångar till jord (GND), och på så vis bestämma att dom ska ha värdet noll. Om du gör det så ska föregående prov med att läsa av tryckknapparna ge följande resultat:

```
TUTOR  1.3 > MM 10082
010082    00 ?=
010082    01 ?=
010082    00 ?=
010082    02 ?=
010082    03 ?
```

Ett annat sätt att bli av med problemet (och därmed slippa att koppla in alla ingångar) är att låta ett program ta hand om värdet och *maska bort* dom bitar som vi vet är opålitliga.

## 5.4 In- och utmatning via ett program

Låt oss göra ett litet program som initierar PIA:n (enligt tidigare), läser av tryckknapparna, och tänder dom fyra vänstra lysdioderna om vänster tryckknapp är nedtryckt, samt tänder dom fyra högra lysdioderna om höger tryckknapp är neddtryckt. Programmet kan då skrivas som följer:

```
        CLR.B   $10084          ; select DDRA
        MOVE.B  #$FF,$10080     ; set DDRA as output
        MOVE.B  #$04,$10084     ; select PIAA

        CLR.B   $10086          ; select DDRB
        MOVE.B  #$00,$10082     ; set DDRB as input
        MOVE.B  #$04,$10086     ; select PIAB

LOOP    MOVE.B  $10082,D0       ; read PB to D0
        AND.B   #$03,D0         ; mask out unwanted bits

        CMP.B   #$01,D0         ; compare for left button
        BEQ     LEFT            ; if equal, jump to LEFT

        CMP.B   #$02,D0         ; compare for right button
        BEQ     RIGHT           ; if equal, jump to RIGHT

        MOVE.B  #$00,$10080     ; clear port value
        BRA     LOOP            ; jump to LOOP

LEFT    MOVE.B  #$F0,$10080     ; light 4 left LEDs
        BRA     LOOP            ; jump to LOOP

RIGHT   MOVE.B  #$0F,$10080     ; light 4 right LEDs
        BRA     LOOP            ; jump to LOOP
```

När du skrivit in, sparat, assemblerat, laddat in och startat programmet ska du kunna tända dom fyra vänstra eller fyra högra lysdioderna beroende på vilken knapp som är nedtryckt. Men vad händer om du håller nere båda knapparna? Tänk först, prova sen.

—o-O-o—

Som avslutning, märk att programmet nu kör i en oändlig loop och vi har inga brytpunkter. Så hur gör vi för att avsluta programmet och komma tillbaka till Tutor-prompten? Jo, Tutor-kortet har två tryckknappar, en röd och en svart. Den röda är en reset-knapp som startar om själva Tutor-kortet. Den svarta aktiverar ett avbrott som helt enkelt avbryter det som körs just nu, dvs programmet. Alltså, tryck på den svarta knappen. Skulle det, av något skäl, inte fungera, prova då att trycka på den röda.

—o-O-o—