# Laboration 1 i Datorteknik

## Utvecklingssystemet IAR Embedded Workbench
## Grundläggande assemblerprogrammering för ARM

Målet med laborationen är att få begrepp om

- Assemblerprogrammering med IAR Embedded Workbench
- Grundläggande assembler
- Enkel inmatning

## Uppgift1: IAR Embedded Workbench

### Syfte:

Lära sig använda grunderna i utvecklingsverktyget IAR Embedded Workbench.

### Beskrivning:

När man använder IAR Embedded Workbench arbetar man alltid med projekt. Ett projekt är en samling filer som tillsammans definierar all programkod och information om hur den ska översättas till maskinkod.

Bilaga1: Hårdvarubeskrivning – SAM3U-EK Evaluation Kit

Bilaga2: SAM3U Memory Mapping – AT91SAM ARM-based Flash MCU, SAM3U Series Summary – kapitel 8, sid 30

Bilaga3: Parallel Input/Output Controller User Interface - AT91SAM ARM-based Flash MCU, SAM3U Series, kapitel 30., sid 509-548

### Steg1: Skapa nytt projekt

- Skapa en katalog för projektet.
- Starta IAR Embedded Workbench

  Börja med Project->Create New Project

  Ställ in ARM

  asm

  asm [OK]

  Döp projektet till LAB1_1
- Gå in under Project-Options

  1) General Options

  Välj Atmel/ Atmel AT91SAM3U4 under Device

  2) Linker

  Välj Override default och välj där (browse) sam3u4-sram.icf

  3) Debugger

  Välj J-Link/J-Trace under Driver

Den tomma filen asm.s dyker upp. Skriv in din kod (enbart själva programmet) i stället för ursprungskoden.

Ursprungskoden:

```
main  NOP
      B       main
```

Byt ut direktivet CODE32 mot THUMB (två ställen). Under `SECTION .intvec` lägg till följande kod:

```
      SECTION .intvec : CODE (2)
      THUMB
      DATA
__vector_table
      DCD   0x20008000;
      DCD   __iar_program_start
```

**Programkoden** läggs in under SECTION .text

```
      SECTION .text : CODE (2)
      THUMB
__iar_program_start
      B     main
main  MOV   R0, #0      ; (R0) <-- 0
      MOV   R1, #0      ; (R1) <-- 0
LOOP  CMP   R0, #9      ; Kolla om (R0)=9
      BNE   ADD0        ; Nej --> forts till ADD0
      MOV   R0, #0      ; Ja --> Starta om
      ADD   R1, R1, #1  ; ... och uppdatera R1 först
      B     ADD1        ; Hoppa
ADD0  ADD   R0, R0, #1  ; Add #1 to R0, (R0)<--(R0)+1
ADD1  CMP   R1, #10     ; Jämför R1 med #10
      BNE   LOOP        ; Upprepa!
STOP  B     STOP
      END
```

## Steg2: Bekanta sig med assemblerkoden

### Teori: Assemblykod

För att kunna förstå fortsättningen, behövs vissa kunskaper om *assemblerspråket*, som är det mest maskinnära språket där varje rad motsvarar en datorinstruktion. Programexemplet illustrerar några vanliga *instruktioner* och *assemblerdirektiv* (direktiv till *assemblatorn*, programmet som översätter till maskinkod). Den text som utgör programmet kallas källkod. Källkoden är ”källan” eller ursprunget till det som ska bli ett program som mikrodatorn kan förstå. En rad består av ett antal fält:

- Symbolfält (labels)

- Instruktionsfält
- Kommentarsfält

### 1. Symboler

Nästan alla rader kan inledas med en symbol. Dessa skrivs alltid *längst till vänster*. I programmet är **MAIN, LOOP, ADD0, ADD1 och STOP** *symboler*. De definierar rader i programkoden, och dessa brukar kallas *programlägen* eller *etiketter* (eng. *labels*). I programkoden använder man ofta symboler före en instruktion för att kunna hoppa till instruktionen. Symboler bör ha ett förklarande namn. I programmet skulle symbolerna ha kunnat heta sym1 och sym2. Detta är inte speciellt klokt. Man bör välja sina symbolnamn med omsorg, så att de beskriver vad de symboliserar.

### 2. Instruktioner och direktiv

Kärnan av programmet utgörs av instruktioner och assemblerdirektiv. Om man inte använder programlägen framför dessa, ska de alltid inledas med *blanktecken* eller *TAB*. Direktiven motsvarar ingen speciell programkod, utan är som namnet anger, direktiv till assemblern. I vår källkod förekommer följande direktiv:

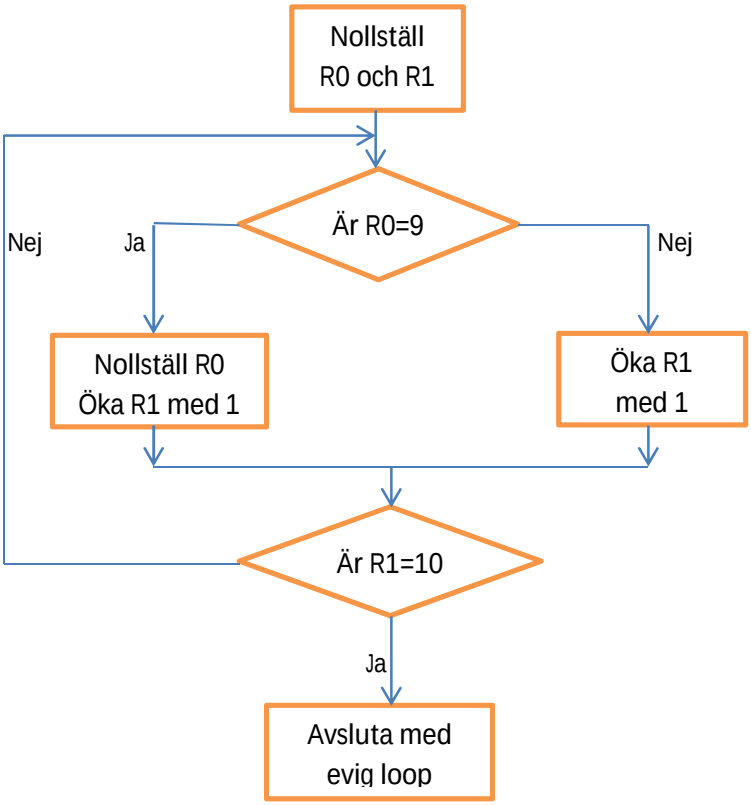
| | |
|---|---|
| **EQU** | Ger ett värde ett namn, det som står framför EQU |
| **THUMB** | indikerar kodtyp innan kodstart |
| **END** | markerar slutet på programmet |

### 3. Kommentarer

De rader, som *inleds med* semikolon, är kommentarer till programmet. När man skriver program i assemblerspråk, är det mycket viktigt att man då och då gör en kommentar som förklarar det som inte är uppenbart i programmet. Vad som är uppenbart, är naturligtvis olika från fall till fall. En erfaren programmerare kanske bara har några få inledande kommentarer, som beskriver vad ett programavsnitt gör. Den oerfarne däremot, skriver gärna alltför många kommentarer så att programmet ändå blir svårt att förstå. I programexemplet ovan är de flesta rader kommenterade, men det beror på att det är det första exemplet i denna kurs. En regel är, att så fort man funderat över en eller flera instruktioner, bör man skriva en liten förklaring i form av en kommentar.

Första delen av koden lägger upp konstanter. Dessa tar ingen minnesplats utan är endast till för assemblatorn (översättningsprogrammet). Då översättningen görs ersätts de definierade namnen i koden med dess motsvarande värden.

Detta följs av den kod som utgör själva programmet. Den beskrivs enklast med ett flödesschema:




### Kodens funktion:

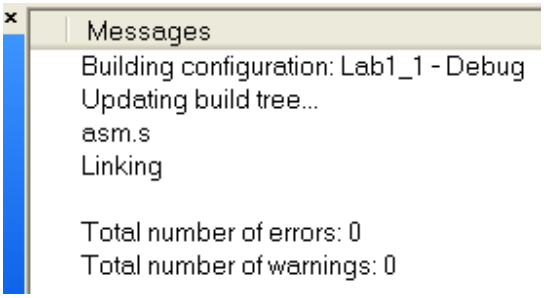
Koden räknar upp **R0** tills den blir 9. Så fort **R0** ska bli 10 slår i stället **R1** om. **R0** nollställs och allt fortsätter tills **R1** blir 10. Då stannar programmet. Man skulle kunna säga att **R0** räknar ental och **R1** tiotal.

Bekanta dig väl med koden innan du går vidare!

## Steg3: Kompilering

- Tryck på knappen **Make** (F7) för att kompilera din kod.

Gick allt bra ska följande visas i **Message**-rutan



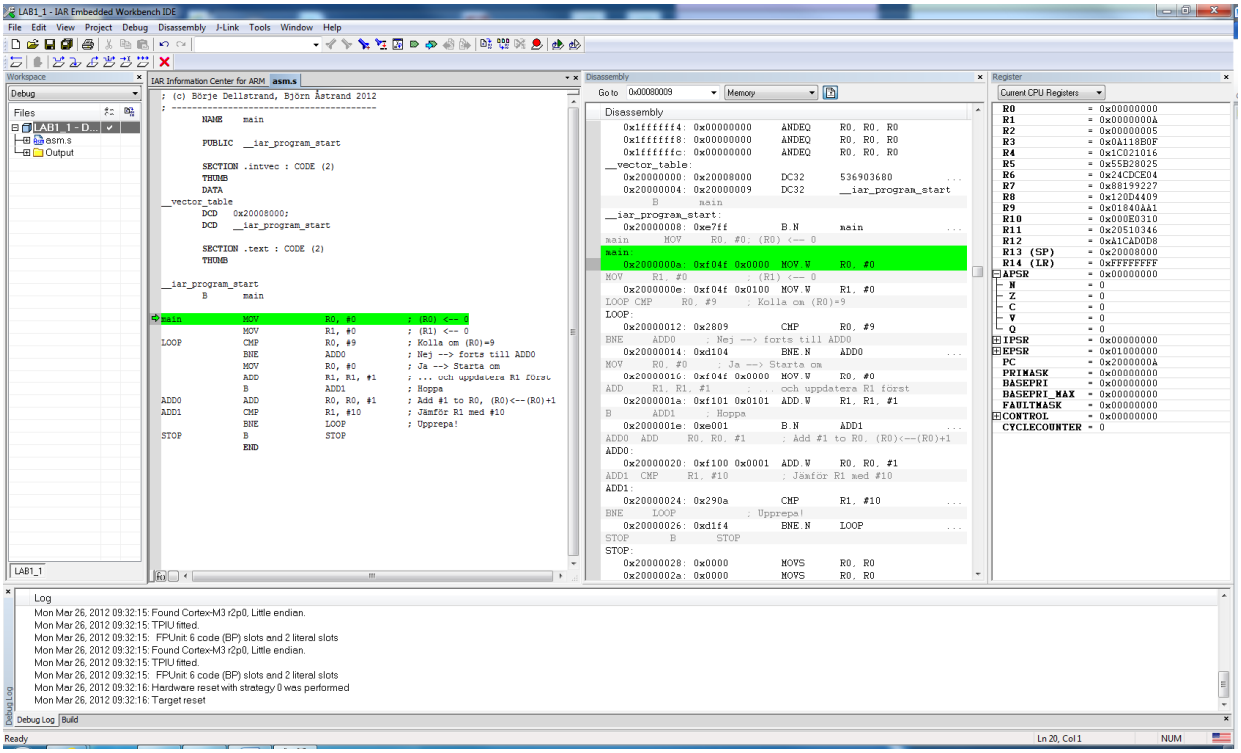

- Tryck sedan på **debug**-knappen (längst t. h.). Då ska skärmen växla utseende och flera nya fönster komma upp. Det bör se ut ungefär som nedanstående:

## Steg4: Debug-läge

Detta läge används för att köra själva koden. Här finns funktioner för att stega i koden samt övervaka minne och register. Det finns flera finesser inbyggt i programmet, men de vanligaste och viktigaste är de som syns.

En kort beskrivning av fönstren:

| | |
|---|---|
| Register | De interna processorregistren (View – Register) |
| Disassembly | Den översatta koden. Det som egentligen ligger i minnet (View – Disassembly) |
| asm.s | Själva koden |



## Steg5: Köra koden

Betrakta följande rader i koden:

```
__iar_program_start
      B     main
main  MOV   R0, #0
```

Lite längre upp i koden står följande:

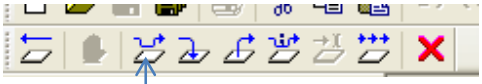
```
SECTION .text : CODE (2)
```

Det betyder att här läggs programkoden in i minnet, som i detta fall är det interna SRAM:et. SRAM:ets startadress är 0x20000000. På SRAM:et läggs också sektionen `.intvec` och här läggs interupt-vektorerna (mer om detta senare) på SRAM:et. Om man titta i Disassembly fönstret ser man var i minnet programmet startar. Labeln `__iar_program_start` ligger på adressen 0x20000008 och programmet börjar med att man hoppar till `main`. Labeln `main` ligger på adressen 0x2000000a.

Efter detta ska utseendet i kodfönstret ändras och man ser en grönmarkerad rad där program-räknaren PC för närvarande står.

- Börja stega i koden (kör rad för rad) genom att klicka på **step-over**-knappen (eller tryck

- på **F10**).



Notera hur registren ändrar sig. Man kan se vilken senaste registerskrivning är då det registret blir rödmarkerat.

Notera också hur programräknaren i kodrutan flyttas framåt. Stega dig fram till första kodraden i själva huvudprogrammet.

- Stega vidare i programmet och notera hur **R0** ändrar sig. Stega fram tills **R0** når värdet 8 och du står på första raden efter `LOOP`.

OBS! Starta om koden:
Om du skulle behöva börja om från början (t ex om du stegar för långt) väljer du enklast **Debug->Reset** då **PC**-registret återställs till 0x2000000a (`main`). (Kommer du inte ihåg hur, se ovan)

- Vad är följande flaggor satta till i statusregistret **APSR** nu? (Syns enklast genom att klicka på pluset bredvid register APSR).

| | |
|---|---|
| N | |
| Z | |

- Stega vidare tills **R0** är 9 och du står på raden efter `LOOP`.

- Stega förbi den raden och notera hur statusregistret **APSR** ändras. Det datorn utför vid en jämförelseinstruktion (`CMP`) är att minska *operand1* med *operand2*. I vårt fall blir det **R0** - 9.

- Vad blir flaggorna satta till nu?

| | |
|---|---|
| N | |
| Z | |

- Teorifråga: Instruktionen `CMP R0,#9` utför operationen R0 - 9 och sätter sedan flaggorna utefter resultatet. Vad bör flaggorna bli om R0 har följande värden?

| | R0<9 | R0=9 | R0>9 |
|---|---|---|---|
| N | | | |
| Z | | | |

Notera hur **R0** nollställs och **R1** ökas då man fortsätter stega.

Ett annat sätt att stega är att sätta en brytpunkt i koden och köra full fart mellan brytpunkterna. Lämpligt kan vara att sätta brytpunkten på den rad som uppdaterar **R1** så behöver man inte stega genom uppräkningen av **R0**, nu när man vet att det fungerar.

- Sätt en brytpunkt på raden `ADD R1,R1,#1` genom att sätta markören på raden och tryck på **Toggle breakpoint** (röd punkt). En röd punkt kommer fram på aktuell rad i koden.

- Kör nu vidare i koden fast med **go**-knappen. Den kommer att stanna på brytpunkten. Prova detta tills **R1** blivit 5.

Ett annat mycket smidigt sätt att stega i koden är funktionen **Run to cursor**.

- Börja med att ta bort brytpunkten. Sätt markören på brytpunktsraden och välj ännu en gång **Toggle breakpoint**. Sätt markören på raden genom att endast klicka i kodfönstret.

- Välj knappen **Run to cursor** denna gång för att stega.

Notera hur enkelt man kan sätta snabba brytpunkter i koden. Efter du har stegat något steg på det här sättet kan du köra programmet i full fart.

- Tryck på **go**. Programmet kör nu i full fart!

Notera att du inte kan se hur någonting på skärmen ändrar sig då du kör i full fart. Detta är inte så konstigt, då registren i processorn uppdateras många gånger snabbare än vad man skulle hinna uppfatta. Vill man kunna se något får man stega eller sätta brytpunkter.

- Tryck på **break**-knappen!

Programmet stannar nu på den raden den körde för tillfället. I vårt fall raden:

```
STOP        B       STOP
```

Denna eviga loop är ett effektivt sätt att stanna en kod så att den inte skenar iväg i minnet. Hade man inte låst upp koden på detta sätt hade processorn fortsatt hur långt som helst i minnet. Detta är ju inte så konstigt då det enda en processor gör är att utföra instruktionscykeln.

## Uppgift2: Modifiering av kod. BCD-räknare

### Syfte:

Lära sig skapa egen kod.

### Beskrivning:

Målet är att skapa ett projekt och modifiera en egen kod.

### Steg1: Skapa ett nytt projekt

- Se Uppgift1 Steg1
  Lämpligt katalognamn är **Lab1_2_bcd**.

Kopiera s-filen från Uppgift1 Steg1.

### Steg2: Lägg till projektfil och provkompilera

- Nu lägger du till den nya filen asm.s genom att välja **Project->Add Files**. Lägg till filen asm.s genom att markera filen och tryck **Open**-knappen.

Efter detta bör du välja och kunna se filen i projektfönstret under Common sources.

- Markera filen asm.s och välj **Project->Compile (**alt **Make)**.

Nu bör Messages-fönstret dyka upp. Inga fel ska synas.

### Steg3: Förändring av källkod

- Efter detta går du över i debug-läge (med **Debug**-knappen) och arrangerar upp fönstren i debugfönstret (register och asm.s).

- Gå tillbaka till editeringsmode genom att trycka på knappen **Stop Debugging**

Du ska nu ändra programmet enligt följande:

1. Hitta raden som ökar upp **R1** med 1. Ändra registret från **R1** till **R0**, dvs. så att det blir **R0** som ökar med ett. Ta även bort raden ovanför som nollställer **R0**.
2. Kompilera och gå över i debugläge och prova programmet. Det ska nu räkna upp **R0** hela tiden.

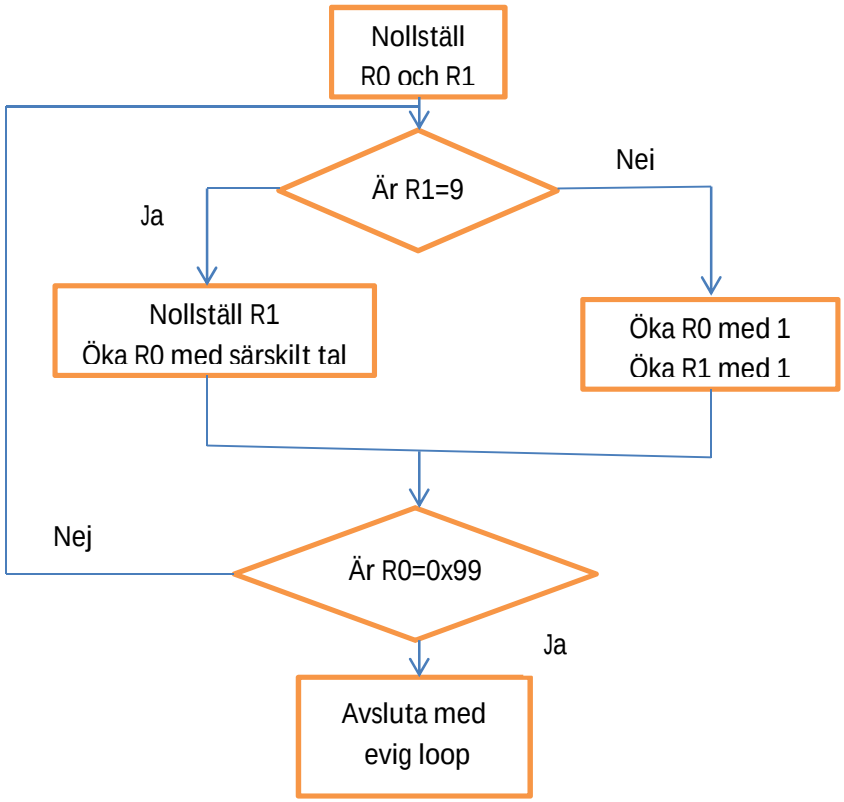
Testa och ändra tills ovanstående stämmer! Notera att **R0** visas hexadecimalt.

## Steg4: Slutgiltigt program. BCD-räknare

Som slutkläm ska ett program som räknar ”decimalt” i R0 konstrueras. Man vill helt enkelt ha följande räknesekvens i R0 (i hexadecimal form): 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15 ända till 0x99

En lösning (flera finns) är att låta ett extra arbetsregister, t ex R1, räkna upp 0-9. Varje gång det registret blir 9 ska man öka ett visst tal i R0, annars bara talet 1. På så vis kan man få R0 att räkna upp till synes decimalt. Man brukar säga att R0 då räknar BCD-kod.

Ett program som gör detta får följande flödesdiagram:




Konstruera ett program som gör detta!

# Uppgift3: Porthantering

## Syfte:

Lära sig konstruera enkla program som kommunicerar med hårdvaran på PortA och PortB.

## Steg1: Skapa projekt

- Se Uppgift1 Steg1
  Lämpligt katalognamn är **Lab1_3_PAoB**.
- Kopiera kod från assemblerfilen ”Lab1_3_student.s”

## Steg2: Konfigurering av PortA och PortB

### Teori:

Alla portar på ARM nås genom att läsa alt. skriva till en speciell adress. Principen att kunna skriva till resp. läsa från en port precis som om det vore en del av själva minnet kallas *minnesmappad I/O*. Huvudskälet till detta är för att spara instruktioner. Annars skulle man behöva specialinstruktioner för att nå varje port på chipet. Alltså nås den fysiska porten precis som om den vore vilken minnesplats som helst!

Alla portar har ett Enable Register kopplat till sig för att kunna styra motsvarande pinnar. Dessutom finns Output Enable/Disable Register som bestämmer portens elektriska attribut. PortA och B går att konfigurera som både ut- resp. insignal (se bilaga3, Kap 30). Alltså måste outputregistret ställas beroende på den hårdvara som är kopplad till systemet.
Se bilaga 1 (Kap 4 och Tabell 4-5) för hur hårdvaran är kopplad till PortA (`BP_LEFT` och `BP_RIGHT` kopplade till bit18 resp. 19) och PortB (LED1 och LED2 kopplade till bit0 resp. 1). Bit0 och 1 i PortB:s PIO_OER måste alltså aktiveras (”output enable”) liksom bit18 och 19 i PortA:s PIO_ODR (”output disable”).
För utgångarna behövs inga pull-up resistorer varför PIO_PUDR aktiveras (”disablas”), för ingångarna gäller att PIO_PUER aktiveras (’enablas’). För ingångarna gäller också att interrupt ”disablas” med PIO_IDR (vi använder här polling).
Läsning sker med registret PIO_PDSR och skrivning sker med registren PIO_SODR (set) resp. PIO_CODR (clear).

Till de flesta av dessa ”styrregister” (enable, output data, pull-upp etc.) finns två register (minnespositioner) som sätter respektive släcker bitar (vid skrivning till dessa register så anger en 1:a att motsvarande bit ska sättas eller släckas, medan en 0:a har ingen effekt). Dessa ”styrregister” kan via ett tredje register (minnesposition) läsas av. Så till exempel för port A:s styrregister Enable Register så finns minnespositionerna PIOA_PER och PIOA_PDR att slå på respektive av en viss bit; medan man på PIOA_PSR kan läsa av hur Enable Register är inställd.

## Steg3: Läsning av data från PortA

Initiering av PortA görs med registren PIOA_PER, PIOA_ODR, PIOA_IDR och PIOA_PUER (se ovan). Adresserna för respektive register fås från Bilaga3.
Att läsa från PortA görs genom att läsa in data från register PIOA_PDSR. Detta görs genom att först läsa in adressen till PIOA_PDSR i ett register. Sedan kan man läsa från den adressen, t ex enligt nedan:

```
LDR       R1,=PIOA_PDSR  ; Adress till PortA:s dataregister
LDR       R0,[R1]        ; Läser från adress i R1 till R0
```

**R0** kommer att få PortA:s värde. Notera att man INTE kan läsa från PortA direkt utan måste gå omvägen att först lägga adressen i **R1**, sen läsa från adressen som anges av **R1**. Detta gäller alltid vid assemblerprogrammering!

- Lägg till ovanstående kod i en evig loop som huvudprogram. En evig loop är när man lägger en label i början av ett kodparti och en branch (ovillkorligt hopp) under koden som hoppar tillbaks till början. Detta är endast för att kunna testa inläsningsrutinen på ett smidigt sätt.

```
LOOP
        <egen kod>
        B         LOOP          ; En evig loop
```

## Steg4: Avläsning av port. Kontroll i minnet

- Leta i rätt på PortA:s basadress (bilaga2, Figur 8-1) och de olika registrens offsetadresser (bilaga3, Kap 30 speciellt Tabell 30-2).

  PIOA_PER: ..……

  PIOA_ODR: ……

  PIOA_IDR: ……..

  PIOA_PUER: ……

  PIOA_PDSR: ……..

- Öppna minnesfönster för att kunna se minnet genom **View->Memory**. Ställ in basadressen till PortA i rutan **Go to**. Stega ditt program över portinläsningen och håll inne resp. släpp *den vänstra* av de två knapparna på kretskortet. Notera vilken bit som ändrar sig. Tänk på att registret är 32 bitar!

OBS! Du måste uppdatera minnesfönstret för att kunna se förändringar. Uppdatering sker då man stegar i koden.

Vilken bit är det som ändras? .....

Gör samma sak med den högra knappen.

Är biten 1 eller 0 då knappen är nedtryckt? .....

Vilken programrad är det som gör själva inläsningen till minnet?.....

## Steg5: Maskning av PortA

Nu när du vet vilken bit det är som ändras gäller det att få datorn att reagera på just den biten. En teknik för detta kallas maskning. Genom att utföra operationen **AND** mellan inläst portvärde och en s.k. bitmask kan man plocka ut information om en enskild bit. Alla andra bitar än den intressanta sätts då till noll. Den/de intressanta bitarna kommer då att behållas.

### Exempel

Anta att data från PortA är inläst i R0 och bit4 är den viktiga. Man skapar då en bitmask med en etta på den positionen, dvs. binära talet 00010000 = 0x10

```
            AND       R1, R0, #0X10       ; R0 AND:as med 00010000
            CMP       R1, #0              ; Jämför R1 med 0
            BEQ       NOLLA               ; Är R1 noll var bit4=0
ETTA        …
```

- Modifiera ditt program så att man läser in och jämför i en loop. Om knappen inte trycks ner ska man ligga och snurra i loopen. Trycker man ner den vänstra knappen (`USR_LEFT`) ska programmet hoppa ur loopen och fastna i en evig loop. Trycker man på den högra knappen (`USR_RIGHT`) ska ingen hända. Dispositionen av din kod bör nu se ut som följer:

  Tänk på att maska ut knappen enligt principen beskriven ovan! Du vet ju sen steg4 vilken bit som ändrar sig.

```
            <initiering av PortA>
LOOP        <inläsning>
            <jämförelse>
            <villkorligt hopp tillbaka till LOOP>
STOP        B         STOP                ; Slutar med en evig
loop
```

Testa programmet genom att sätta en brytpunkt på den eviga loopen. Kör programmet i full fart. Om allt fungerar ska programmet hamna på brytpunkten då knappen trycks in.

## Steg6: Tändning/släckning av LED

Komplettera programmet så att LED1 tänds då knappen `USR_LEFT` är nedtryckt, släckt f.ö. Samma sak ska gälla LED2 och knappen `USR_RIGHT`. Om båda knapparna är intryckta ska båda lysdioderna vara tända.

Visa upp resultatet för labhandledaren!

## Summering

Efter utförd laboration har studenten praktisk kunskap i följande:

Färdighet i IAR Embedded Workbench

- Skapa och kompilera projekt
- Stega och debugga kod. Breakpoints, Dissassembler, Memory Window, Register
- Lägga till och modifiera filer

ARM-assembler

- Förståelse för enkla instruktioner och direktiv
- Initierat PortA och PortB och gjort enkel läsning och skrivning
- Maskning av bitar vid inläsning