TSEA28 Datorteknik Y (och U)

Föreläsning 2

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

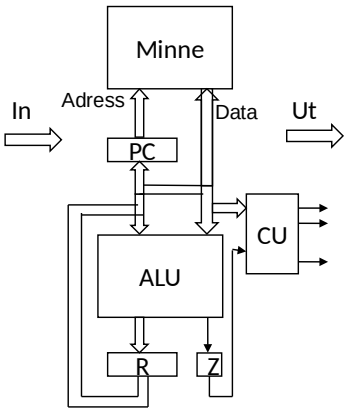
- Kort repetition
-
-
- Större programmeringsexempel
  - Hur strukturera kod
- Subrutiner
- Stack
- Adresseringsmoder

Praktiska kommentarer

- Labanmälan öppnar måndag 22/1 kl 12.45
  - Anmälningssystemet via lisam
  - Varje labb uppdelad i två delar (2h + 2h), välj lämpligen två tillfällen med tid emellan
  - Jobba så långt ni hinner under de första 2 timmarna
  - Väl förberedd kan bli klar med lab redan efter första 2 timmarna
- Jobba i par
  - Ok att jobba ensam, kan dock bli lite svårt med plats om allt för många väljer detta
- Glöm inte möjligheten att testköra allt på distans

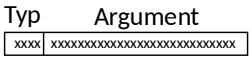
Exekvering av program kräver minst

- Programräknare (PC)
  - Håller reda på position för aktuell instruktion i minnet
- Aritmetikenhet (ALU)
  - Beräknar addition etc.
  - Z: flagga = 1 om resultat = 0
- Tillfälliga register (R)
  - Kan vara flera stycken
- Styrenhet (CU)



Programmet lagras i minnet (exempel)

- Kallas en maskininstruktion
  - Binärt datamönster
  - Tolkas av styrenheten i processorn
- Lagras i en minnescell
  - I detta exempel: 32 bitar i en minnescell
- Två delar
  - Vilken typ av operation
    - Exempel: valt 4 bitar => 16 olika typer
  - Argument till operationen
    - Resten av bitarna: 32-4 = 28 bitar argument



4 bitar 28 bitar

Typkod	Förklaring
0000 (0)	Ladda in argument i R
0001 (1)	Addera argument till R
0010 (2)	Jämför argument med R, Z=1 om R=argument,Z=0 annars
0011 (3)	Sätt PC till argument om Z=0 (villkorligt hopp)
0100 (4)	Sätt PC till argument om Z=1
0101 (5)	Sätt PC till argument (hopp i koden)
0110 (6)	om argument = 2000 skicka R till display
0110 (6)	om argument = 2100 hämta aktuellt displayvärde till R
0110 (6)	om argument = 2200 hämta sensorvärde till R, R=1 om någon står i dörren, R=0 annars
0111 (7)	läs värde till R från minne på adress i argumentet
1000 (8)	skriv värde I R till minne på adress i argumentet

Assemblerinstruktioner

- Svårt komma ihåg och läsa binärmönster
  - Använd så kallade mnemonics istället
  - Oftast förkortningar

Typkod	Assemblerinstruktion	Föklaring
0000 (0)	mov R,#värde	Ladda in argument i R (MOVe)
0001 (1)	add R,#värde	Addera argument till R
0010 (2)	cmp R,#värde	Jämför R med argument, Z=1 om R=argument, Z=0 annars (CoMPare)
0011 (3)	bne adress	Sätt PC till argument om Z=0 (Branch Not Equal)
0100 (4)	beq adress	Sätt PC till argument om Z=1 (Branch EQual)
0101 (5)	b adress	Sätt PC till argument (Branch)
0110 (6)	bl 2000	om argument = 2000 skicka R till display (Branch and Link)
0110 (6)	bl 2100	om argument = 2100 hämta aktuellt displayvärde till R
0110 (6)	bl 2200	om argument = 2200 hämta sensorvärde till R, R=1 om någon står i dörren, R=0 annars

Assemblerinstruktioner, forts

- Olika processorfamiljer har olika mnemonics
  - Olika namn för samma funktion
  - Samma namn för olika funktioner
- Finns även pseudo-instruktioner i assembler
  - Styr assemblerns generering av maskininstruktioner
  - Kan ibland översätta en instruktion till en/flera andra

Z80	ARM	68000	x86_64	Betydelse
ld	ldr	move	mov	Läs data från minne
ld	str	move	mov	Skriv data till minne
jr	b	bra	jmp	Hoppa i programmet
cp	cmp	cmp	cmp	Jämför
add	add	add	add	Addera

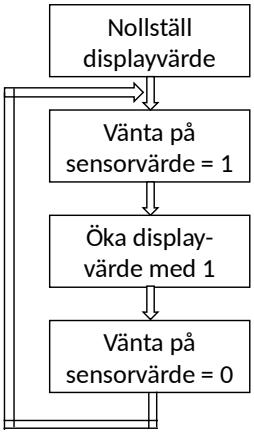
Pseudoinstruktion: Clr r0
Ska nollställa R0

Assembler översätter kanske till
xor r0,r0 ; r0 = r0 xor r0

Implementering i exempeldatorn

- Beskriv varje blocks funktion med hjälp av befintliga instruktioner

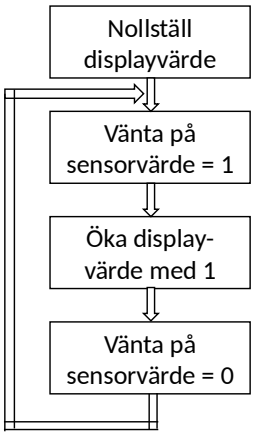
Adress	Maskinkod	Assembler-instruktion	Förklaring
0		mov R,#0	; Nollställ displayvärde
1		bl 2000	; sätt display till 00
2		bl 2200	; Hämta sensorvärde
3		cmp R,#1	; Står någon i dörren?
4		bne 2	; nej, gör om
5		bl 2100	; hämta displayvärdet
6		add R,#1	; öka displayvärdet
7		bl 2000	; visa det nya värdet
8		bl 2200	; Hämta sensorvärde
9		cmp R,#0	; Är dörren tom?
10		bne 8	; nej, kontrollera igen
11		b 2	; börja om



Implementering i exempeldatorn, forts.

- Översätt sedan till binär form
  - Dessa binära värden är vad som lagras i minnet

Adress	Maskinkod Typ	Arg.	Assembler-instruktion	Förklaring
0	0	0	mov R,#0	; Nollställ displayvärde
1	6	2000	bl 2000	; sätt display till 00
2	6	2200	bl 2200	; Hämta sensorvärde
3	2	1	cmp R,#1	; Står någon i dörren?
4	3	2	bne 2	; nej, gör om
5	6	2100	bl 2100	; hämta displayvärdet
6	1	1	add R,#1	; öka displayvärdet
7	6	2000	bl 2000	; visa det nya värdet
8	6	2200	bl 2200	; Hämta sensorvärde
9	2	0	cmp R,#0	; Är dörren tom?
10	3	8	bne 8	; nej, kontrollera igen
11	5	2	b 2	; börja om



Assembler (hjälpprogram)

- Översätter assemblerinstruktioner (text) till maskinkod (binärdata)
  - Skriv en textfil med assemblerinstruktioner
- Håller även reda på adresser
  - Ange bara symboliska namn på platser i programmet (kallas label)

```
wait0:  bl 2200      ; Hämta sensorvärde
        cmp R,#0     ; Är dörren tom?
        bne wait0    ; nej, kontrollera igen
```

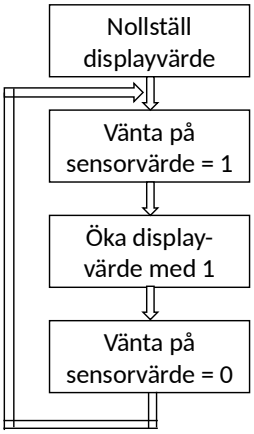
label

Adress	Maskinkod Typ	Arg.
8	6	2200
9	2	0
10	3	8

Assemblerversion av programmet

- Textbeskrivning, inklusive kommentarer
  - Label måste starta längst till vänster
  - Instruktioner måste ha ett mellanslag innan

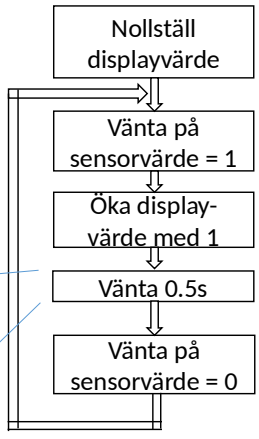
```
start:  mov R,#0      ; Nollställ R
        bl 2000       ; sätt display till 00
wait1:  bl 2200       ; Hämta sensorvärde
        cmp R,#1      ; Står någon i dörren?
        bne wait1     ; nej, gör om
        bl 2100       ; hämta displayvärdet
        add R,#1      ; öka displayvärdet
        bl 2000       ; visa det nya värdet
wait0:  bl 2200       ; Hämta sensorvärde
        comp R,#0     ; Är dörren tom?
        bne wait0     ; nej, kontrollera igen
        b wait1       ; börja om
```



Ytterligare funktion: fördröjning

- Varje besökare kan ge flera pulser
  - Två ben, väska, kläder
- Lösning: vänta en stund efter display räknat upp
  - En loop i programmet som bara tar tid att utföra

Label	Assembler-instruktion	Förklaring
start:	mov R,#0	; Nollställ R
	bl 2000	; sätt display till 00
wait1:	bl 2200	; Hämta sensorvärde
	cmp R,#1	; Står någon i dörren?
	bne wait1	; nej, gör om
	bl 2100	; hämta displayvärdet
	add R,#1	; öka displayvärdet
	bl 2000	; visa det nya värdet
	mov R,#0	; starta timer på 0
delay:	add R,#1	; öka timer med 1
	cmp R,#10000	; lämpligt antal klockcykler?
	bne delay	; inte tillräckligt många varv
wait0:	bl 2200	; Hämta sensorvärde
	cmp R,#0	; Är dörren tom?
	bne wait0	; nej, kontrollera igen
	b wait1	; börja om



Hur skriva ett ”större” program

- Öka kraven i exemplet
  - Lägg till en sensor för dörrlås (räkna inte när jag går och hämtar kaffe…)
  - Kräver ny instruktion för att läsa dörrlåssensor
    - Bl 2300 Läs dörrlås, R=1 om dörr stängd, R=0 annars
  - Räkna upp siffran på display varannan gång (studenter går in och ut)
  - Automatisk nollställning varje natt (om dörr stängd > 8h så nollställ)
- Behöver lite mer strukturerad metod
  - Tänk igenom förväntad funktion, rita flödesschema eller pseudokod
  - Detaljerad beskrivning
  - Minnesceller
  - Riktig assemblerkod, namnge platser, rensa upp

Beskriv funktionen som en sekvens av enklare funktioner

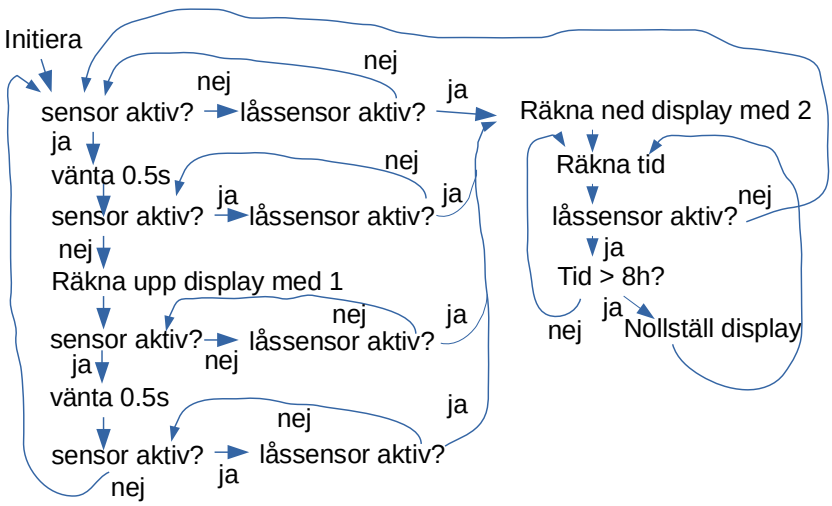
- Tänk igenom förväntad sekvens (scenario)
  - Ingen ide räkna upp medan dörren är stängd
- Rita lite fritt först, med stora komplexa instruktioner
  - Nollställ när strömmen slås på
  - Om dörr stängd räkna ned antal med 2 (måste passera dörren för att kunna stänga den), vänta tills dörren öppnas igen
  - Räkna upp om dörrsensor aktiveras

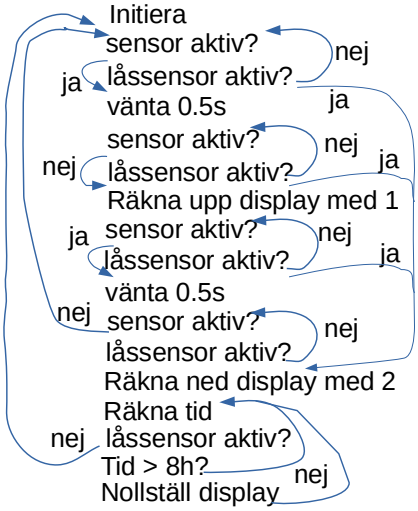
```
Initiera antalsräknare = 0
vänta på aktivitet på dörr eller sensor
   om dörr öppen
      om sensor aktiv räkna upp antalsräknare,
      vänta tills passerat
      vänta på nästa aktivering (personen går ut)
   om dörr stängd
      minska antalsräknare med 2 (ska både ut
och
                                   in i rummet)
      räkna tid tills dörr öppnas
         om tid > 8h nollställ antalsräknaren
```

Dela upp större funktioner i mindre sekvenser

- Försök dela upp i små steg
  - Varje litet steg bör kunna direkt översättas til maskinkod
- T ex: vänta på nästa aktivering (person går ut)
  - Läs sensor, gå vidare om sensor aktiv, annars läs om igen
  - Vänta 0.5s
  - Läs sensor, gå vidare om sensor inte aktiv, annars läs igen

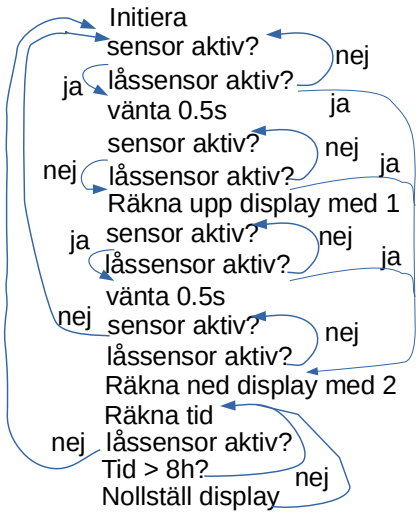
Initial beskrivning följt av ordning in i sekvens





Minnesutrymme

- R används till mycket
  - Sensorvärde
  - Räknarvärden
- Placera räknarvärde i minnet
  - Bestäm adresser för detta
  - 3000: displayvärde
  - 3001: tidräknare
  - Kom ihåg att minnesceller kan ha slumpmässigt startvärde!
    - Måste initiera!



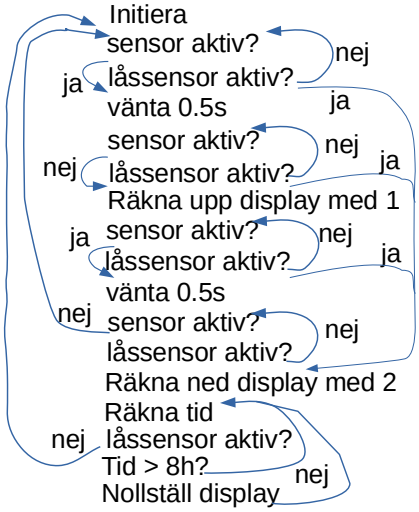
Ersätt med riktig assemblerkod

- Namnge alla platser som ska hoppas till

```
init:   mov R,#0
        str R,3000
        bl 2000

sens1:  bl 2200
        cmp R,#1
        beq ja1

        bl 2300
        cmp R,#0
        beq sens1
        b decrement
```



Hela programmet, del 1

```
init:   mov R,#0        ; initiera displayräknare till 0
        str R,3000      ; sätt sparat displayvärde till 0
        bl 2000         ; och sätt display till 0

sens1:  bl 2200         ; hämta sensorvärde
        cmp R,#1        ; är sensor aktiv?
        beq ja1         ; ja, fortsätt nedan

        bl 2300         ; hämta låssensor
        cmp R,#0        ; är dörren låst
        beq sens1       ; nej, testa sensor och lås igen
        b decrement     ; Ja, hantera i decrement-rutinen

ja1:    mov R,#0        ; vänta 0.5s
wait0:  add R,#1        ; räkna upp
        cmp R,#1000         ; väntat färdigt?
        bne wait0       ; nej, vänta mer

sens2:  bl 2200         ; hämta sensorvärde
        cmp R,#0        ; är sensor inaktiv?
        beq nej1        ; ja, fortsätt nedan

        bl 2300         ; hämta låssensor
        cmp R,#0        ; är dörren låst
        beq sens2       ; nej, testa sensor och lås igen
        b decrement     ; Ja, hantera i decrement-rutinen

nej1:   ldr R,3000      ; räkna upp display
        add R,#1        ; räkna upp
        str R,3000      ; spara det nya värdet
        bl 2000         ; visa på display
```


Hela programmet, del 2

```
sens3:  bl 2200         ; hämta sensorvärde
        cmp R,#1        ; är sensor aktiv?
        beq ja2         ; ja, fortsätt nedan

        bl 2300         ; hämta låssensor
        cmp R,#0        ; är dörren låst
        beq sens3       ; nej, testa sensor och lås igen
        b decrement     ; Ja, hantera i decrement-rutinen

ja2:    mov R,#0        ; vänta 0.5s
wait1:  add R,#1        ; räkna upp
        cmp R,#1000         ; väntat färdigt?
        bne wait1       ; nej, vänta mer

sens4:  bl 2200         ; hämta sensorvärde
        cmp R,#0        ; är sensor inaktiv?
        beq sens1       ; ja, börja om

        bl 2300         ; hämta låssensor
        cmp R,#0        ; är dörren låst
        beq sens4       ; nej, testa sensor och lås igen

decrement:
        ldr R,3000      ; hämta aktuellt displayvärde
        add R,#-2       ; minska med två
        str R,3000      ; spara det nya värdet
        bl 2000         ; visa på displayen
        mov R,#0        ; nollställ tidräknare
tidloop: add R,#1       ; öka tid
        str R,3001      ; spara tidvärde i minnet
```

Hela programmet, del 3

```
bl 2300          ; hämta låssensor
cmp R,#0         ; är dörren låst
beq sens1        ; nej, börja om från början
ldr R,3001       ; har 8h passerat?
cmp R,#80000
beq tidloop      ; fortsätt vänta
mov R,#0         ; nollställ displayvärde
str R,3000       ; spara värdet I minnet
bl 2000          ; visa på displayen
b tidloop        ; nollställ tidräknare
```


Läsa och skriva värden till/från minnet

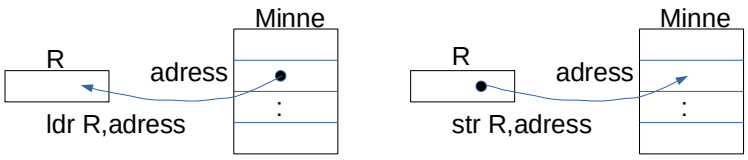
Billigare exempelsystem: Förenkla display

- Enklare display => billigare system
  - Ta bort möjligheten att läsa av värdet
  - Kan bara skriva värde => värde behöver lagras i datorn också
  - R-registrets värde förstörs när sensor läses, kan inte användas
- Två alternativ:Fler register eller lagring i minnet
  - Fler register => fler operationstyper
    - En uppsättning operationer för varje register
  - Lagra i minnet => Två nya instruktioner för att läsa och skriva i minnet
    - Välj adress som inte används till annat (i detta exempel: 100)

Instruktioner för att läsa och skriva i minnet

- Lägg till instruktioner för att spara och hämta ett värde i minnet

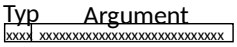
Kod	Assemblerinstr.	Förklaring
7	ldr R,adress	Läs värdet på angiven adress i minnet och placera det i R
8	str R,adress	Skriv värdet i R på angiven adress i minnet

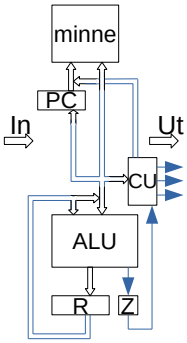


Enkel exempeldator

- 2 register (PC och R), 1 ALU, styrenhet (CU)
- I/O: display och sensor

Maskininstruktionsformat





TypKod	Assemblerinstruktion	Föklaring
0	mov R,#värde	Ladda in argument i
1	add R,#värde	Addera argument till R
2	cmp R,#värde	Jämför argument med R, Z=1 om R=argument, Z=0 annars
3	bne adress	Sätt PC till argument om Z=0 (dvs villkorligt hopp i kodsekvens)
4	beq adress	Sätt PC till argument om Z=1 (dvs villkorligt hopp i kodsekvens)
5	b adress	Sätt PC till argument (dvs hoppa i kodsekvensen)
6	bl 2000	om argument = 2000 skicka R till display
6	bl 2100	om argument = 2100 hämta aktuellt displayvärde till R
6	bl 2200	om argument = 2200 hämta sensorvärde till R, R=1 om någon I dörren
7	ldr R,adress	Läs värde från minnet på minnesadress adress och placera i R
8	str R,adress	Spara värdet i R i minnet på minnesadressen adress

Skillnad mellan instruktionstyp 0 och 7

- Olika funktion!

0	mov R,#värde	t ex mov R,#1	Placera värdet 1 i R
7	ldr R,adress	t ex ldr R,1	Läs minnesadress 1 och placera dess värde i R.

  - Motsvarande maskinkod (binärt format)

    mov R,#1 => 00000000000000000000000000000001

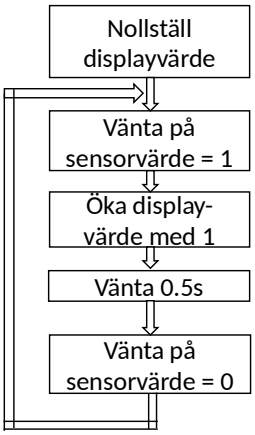
    ldr R,1 => 01110000000000000000000000000001

  - Dvs, samma argument i maskinkod, men olika innebörd pga olika operationstyp
    - Skillnaden indikeras i assemblerkoden i detta fall med tecknet # framför argumentet (samt att här är mnemonic olika)

Exempel med displayvärdet i minnet

- Operation
  bl 2100
- ersatt med
  ldr R,100
  och
  str R,100

```
start:  mov R,#0        ; Nollställ R
        str R,100       ; spara displayvärde adress 100
        bl 2000         ; sätt display till 00
wait1:  bl 2200         ; Hämta sensorvärde
        cmp R,#1        ; Står någon i dörren?
        bne wait1       ; nej, gör om
        ldr R,100       ; hämta displayvärdet
        add R,#1        ; öka displayvärdet
        bl 2000         ; visa det nya värdet
        str R,100       ; spara nya värdet i minnet
        mov R,#0        ; starta timer på 0
delay:  add R,#1        ; öka timer med 1
        cmp R,#10000    ; lämpligt antal klockcykler?
        bne delay       ; inte tillräckligt många varv
wait0:  bl 2200         ; Hämta sensorvärde
        cmp R,#0        ; Är dörren tom?
        bne wait0       ; nej, kontrollera igen
        b wait1         ; börja om
```



Fler register behövs (ett R-register är för lite)

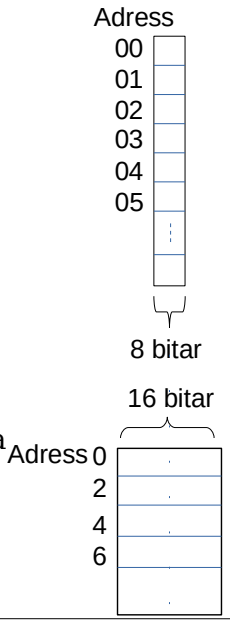
- Svårt t ex skriva värden i en tabell
  - Adressen till plats i tabellen (index) behöver beräknas
    - Lagras i R
  - Värdet som ska placeras på denna plats i tabellen behöver också lagras i R
    - Behöver fler register
- Fler register brukar finnas
  - Labbdatorn ARM har 16 (några med specialfunktion)
  - Gamla labdatorn (68000) har 16 (8 vanliga plus 8 speciella för adresser)
  - 6502 (Apple II) har ett register A samt två indexregister för adresser.
    - Kombinerat med speciell adresseringsmode med minnesadress 0-255
  - 80x86 (laptop/stationär dator): 15 register (32 bitars register för 80386 och uppåt)

Labdatorns registeruppsättning (DARMA)

- 13 generella 32-bitars register (istället för ett ensamt register R)
  - Kallade R0 - R12
  - Flera möjliga instruktioner behövs därför
    - De binära typkoden i maskininstruktionen använder fler än 4 bitar
- PC ingår också (kallat R15 ibland)
- Kan behöva kopiera värden mellan register
  - MOV R0,R1 ; flyttar en kopia av värdet i R1 och placerar i R0
  - Flyttar alltid alla 32 bitarna

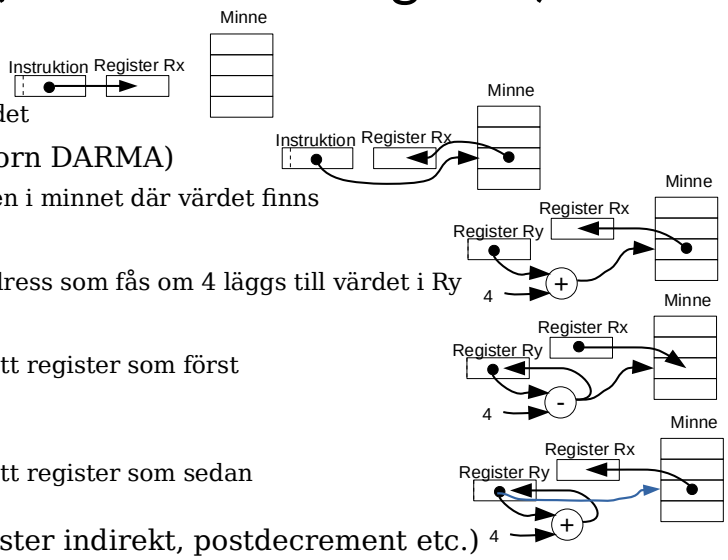
Minnesadressering

- Minne beskrivs oftast som en vektor av byte
  - Adress väljer vilken byte som ska läsas/skrivas
  - Även datorer som hanterar längre ordlängd (t ex 32 bitars register) har minnet beskrivet som array av byte
- Fysiskt kan minnet beskrivas som en annan ordlängd
  - T ex alltid skickar 64-bitar i en äldre PC som har en 32-bitars register i processorn
  - Upp till processorn att automatiskt plocka ut den del som ska användas



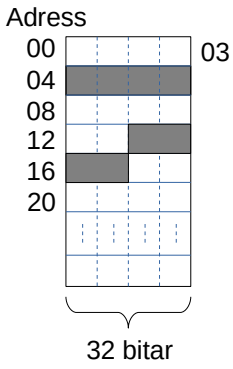
Adresseringsmoder (behöver fler register)

- Omedelbar (immediate, literal)
  - mov Rx,#värde ; argumentet är värdet
- Direkt (absolut) (Finns EJ i labdatorn DARMA)
  - ldr Rx,adress ; argumentet är adressen i minnet där värdet finns
- Offset
  - ldr [Ry,#4],Rx ; Läs minnet på den adress som fås om 4 läggs till värdet i Ry
- Indirekt med predecrement
  - str Rx,[Ry,#-4]! ; adressen fås från ett register som först ; räknas ned med 4
- Indirekt med post increment
  - ldr Rx,[Ry],#4 ; adressen fås från ett register som sedan ; räknas upp med 4
- Många fler moder är möjliga (register indirekt, postdecrement etc.)



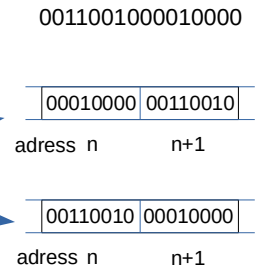
Hur många bitar innehåller en adress?

- Ofta kan flera byte data hämtas vid en läsning
  - Minnet läses eller skrivs en rad i taget
    - Varje rad kan vara längre än 1 byte, t ex 2 eller 4 byte
  - 4 stycken adresser med 8 bitar var läses på en gång
  - Adressen som anges pekar på 1:a byten som ska läsas
- I många fall kan man inte läsa 16 och 32-bitars värden från udda adresser
  - LSB (Least Significant bit/bits) i adress alltid = 0
  - Ex: Läs 32-bitars värde på adress 4 går bra (läser adresser 4, 5, 6 och 7)
  - Ex: läs 32-bitars värde på adress 14 går inte bra (adress 14 och 15 ligger på annan rad än adress 16 och 17)
  - Gäller inte labdatorn (DARMA)



Big endian vs little endian

- Inte helt klart hur ett 16-bitars ord lagras i minnet
  - Två adresser används (adress n och n+1)
  - Ligger mest signifikant byte först eller sist?
    - Beror på processortillverkare
    - Little endian: minst signifikant byte först
    - Big endian: mest signifikant byte först
- Finns ett publicerat papper om bakgrunden till skillnad och namn
  - Danny Cohen, ”On Holy Wars and a Plea for Peace”
  - Notera publiceringsdatum: 1980-04-01….
  - Referenser till Gullivers resor av Jonathan Swift



Subrutiner, stack

Återanvändning av kod

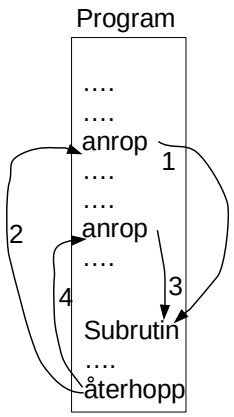
- Många likadana sekvenser i koden
  - Fördröjning på 0.5s ser likadan ut på två ställen

```
ja2:    mov R,#0        ; vänta 0.5s
wait1:  add R,#1        ; räkna upp
        cmp R,#1000     ; väntat färdigt?
        bne wait1       ; nej, vänta mer
```

  - Kan vara många fler rader i koden
- Lagra endast en kopia av sekvensen
  - Återanvänd genom att hoppa till sekvensen från flera olika platser
  - Kallas subrutin
  - Samma idé som funktioner och procedurer i vanliga programspråk

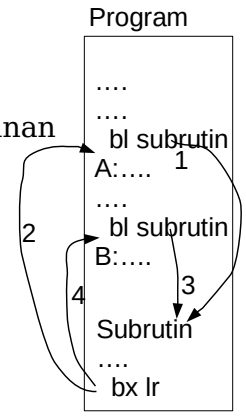
Subrutiner

- Måste komma ihåg var programmet ska fortsätta efter subrutinen
- Återhopp 4 har en annan adress än återhopp 2!
  - Vanligt hopp fungerar inte (som måste ha fast adress) för återhoppet
- Spara adress till nästa instruktion när anrop till subrutin görs
  - Speciell subrutinsinstruktion istället för vanligt hopp
  - Den sparade adressen kan användas vid återhoppet (ytterligare en speciell instruktion)



Subrutiner, forts.

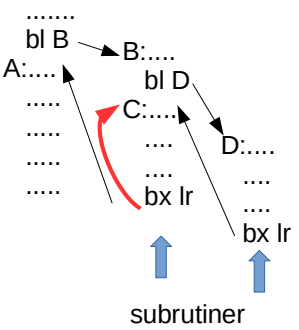
- Vid anrop till subrutin
  - bl subrutinaddr ; hopp till subrutin som startar på adressen subrutinaddr i minnet
  - Sparar adressen till instruktionen efter bl (finns i PC-registret innan hoppet utförs, A respektive B i exemplet) i ett register (extra register kallat lr)
- När subrutinen är slut
  - bx lr; retur från subrutin
  - Använd den sparade adressen från bl-instruktionen (liggande i registret lr) och placera den i PC-registret (dvs gör ett hopp)
- Placering av den sparade återhoppsadressen?
  - Ett speciellt register (LR) alternativt en speciell adress i minnet



Subrutiner: rekursiva anrop, ett LR-register....

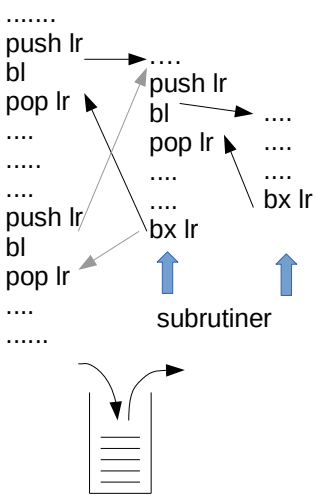
- Rekursion kräver möjlighet att lagra flera anrop i rätt ordning
  - Om bara ett register kan användas tappas första återhoppsadressen bort vid andra anropet
- Exempel
  - Första bl => återhoppsregister LR = A
  - Andra bl => återhoppsregister LR = C (ersätter värdet A).
  - Återhopp i subrutin D => återhopp till C
  - Återhopp i subrutin B => återhopp till C!

    FEL!!! (ganska vanligt fel i labblösningar)



Subrutiner: stöd för rekursiva anrop

- Rekursion kräver möjlighet att lagra flera återhopps-adresser i rätt ordning
  - Push instruktionen sparar värdet i LR på en hög med tidigare värden innan bl skriver över LR med ett nytt värde
  - Pop återställer LR med tidigare återhoppsadress från toppen av högen innan återhopp
- Denna datastruktur (hög) kallas för stack (last in first out)
  - Jfr en låda med papper
  - Värden läggs till högst upp, värden tas bort från toppen

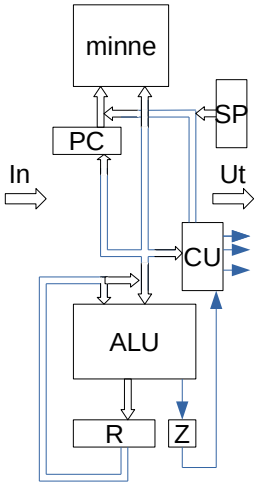


Implementering av en stack

- Ett speciellt (dedikerat) minne inuti processorn för stack
  - Ineffektivt
    - Begränsad storlek
    - Kan inte användas till annat
  - Kan möjligen ge lite bättre prestanda i vissa fall
- Stacken använder en del av minnet
  - Måste hålla reda på aktuell position (toppen på stacken)
    - Speciellt register i processorn
    - Kallas stackpekare (SP)
  - Stacken kan växa uppåt eller nedåt i minnet
    - Kan ibland väljas för processorn

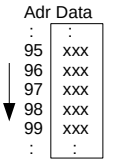
Utökning av exempeldatorn med stack

- Adress till minne måste styras med PC, SP och CU
  - PC för nästa instruktion
  - CU för adress från argument i instruktion
  - SP för läsa/skriva till stacken
- Måste initiera SP innan den används!
  - Måste finnas skrivbart minne på adresserna
  - Alternativ 1: Sätt SP vid reset (spänningspåslag) till ett förbestämt värde
  - Alternativ 2: Sätt SP i programmet
    - Kräver ny instruktion: mov SP, #adress
- Eventuellt extra push R och pop R instruktioner

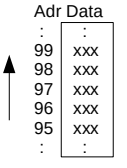


Grafisk beskrivning av minnesinnehåll

- Program ökar PC med 1 för varje instruktion
  - Naturligt skriva lägsta adressen högst upp (näst instruktion under föregående instruktion i listan)
  - Låga adresser skrivs längst upp, höga adresser skrivs längst ned
  - Kan ibland bli lite motsägelsefullt
    - Om något växer, ökar eller minskar adressen?
  - Används i kursen
- Ibland ritas minneskartor med omvänd adressordning
  - Låga adresser längst ned, höga adresser längst upp



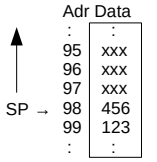
Program växer nedåt (mot högre adresser)



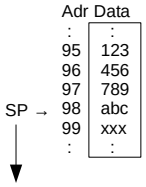
Program växer uppåt (mot högre adresser)

Alternativ för stackimplementering

- Stack kan växa uppåt eller nedåt
  - Mer vanligt att stacken växer mot lägre adresser (nedåt)
  - Program växer i storlek mot högre adresser => stack växer från andra hållet
    - Förhoppningsvis möts de aldrig!
- SP kan peka på första lediga plats eller senaste placerade data
  - Valet påverkar om SP ändras före eller efter att data lagts på stacken
- ARM (labbutrustningen DARMA):
  - Stack växer mot lägre adresser, pekar på senast placerade data



Växer nedåt (mot lägre adresser)



Växer uppåt (mot högre adresser)

Exempel på sekvens av stackoperationer

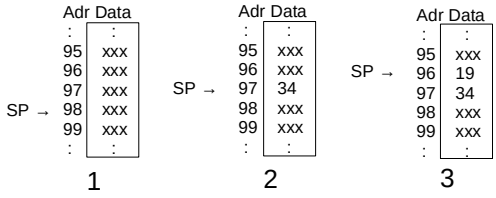
1: SP = 98
- Okänt innehåll (xxx) i minnet

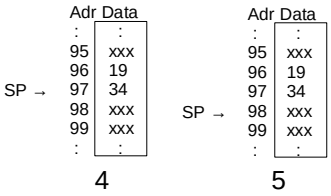
2: mov R,#34 Push(R)
- SP = 97
- Minne[97] = 34

3: mov R,#19 Push(R)
- SP = 96
- Minne[96] = 19

4: pop(R)
- R = Minne[96] = 19
- SP = 97

5: pop(R)
- R = Minne[97] = 34
- SP = 98





Subrutinanrop alt. 1: utan automatisk stack

- Extra register enbart avsett att lagra PC vid subrutinanrop
  - LR : Link Register
- bl subrutinadress ; hopp till subrutin
  - LR = adress till instruktion efter bl (dvs PC-registrets värde)
  - PC = subrutinadress
- Bx lr ; Återhopp vid slut på subrutin
  - PC = LR
- Subrutinen måste själv spara undan LR:s nuvarande värde innan nästa bl (t ex på en stack)
  - Push (LR) innan nästa nivå av subrutinanrop, Pop(LR) innan återhopp

Subrutinanrop alt. 2: automatisk stack

- Kombinera push och bl i en enda instruktion, respektive bx lr och pop i en instruktion
  - Behöver inget LR-register i detta fall (skriver PC direkt till stack)
- Jsr subrutinadress
  - Push (PC), dvs adress till instruktionen efter jsr
  - PC = subrutinadress
- Rts ; återhopp vid slut på subrutin
  - PC = pop()
- En subrutin kan anropa sig själv utan extra hantering
  - För många anrop gör att stacken blir för stor (SP -> 0 eller SP pekar på annan viktig data, t ex programmet)

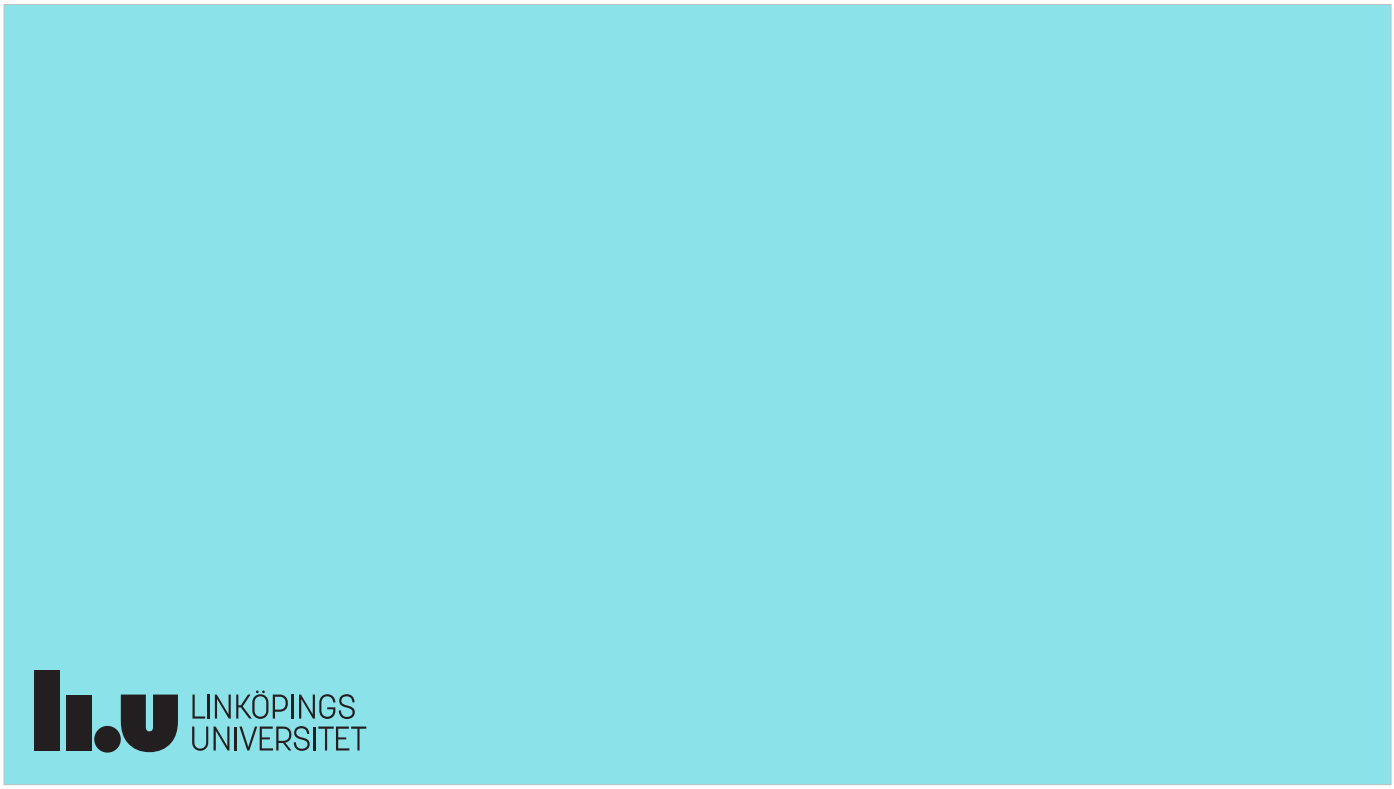
```
nfak(N):
If N=1 then
  Return 1;
Else
  Return nfak(N-1)*N
end
```

Jämförelse subrutinanrop med/utan automatisk stack

- Fördel automatisk stack (68000, x86 mfl)
  - Enklare att programmera
  - Färre instruktioner i subrutinerna
    - Behöver inte extra instruktioner för att spara undan och hämta LR på stacken
  - Behöver inte extra instruktioner för hantering av LR registret
- Fördel ingen automatisk stack (ARM mfl)
  - Snabbare instruktion
    - Behöver inte spara onödiga adresser på stacken
  - bx lr behöver inte läsa från minnet vid återhopp

Skicka information till/från subrutiner

- Data till subrutin
  - Sätt värde i register som subrutinen förväntar sig, anropa sedan
  - Klarar subrutiner där få värden behövs som inparametrar
- Resultat från subrutin
  - Subrutinen sätter värde i register innan återhopp
  - Klarar subrutiner som behöver returnera få värden
- För stora datamängder behöver minnet användas
  - Alt. 1: Fix adress för data som är globalt (tillgängligt/delat för alla rutiner)
  - Alt. 2: Lagra extra data på stacken innan subrutinanrop => Läses av subrutin genom att läsa adresser relativt LR-registret (visst avstånd från aktuellt LR-värde).


LINKÖPINGS
UNIVERSITET