TSEA28 Datorteknik Y (och U)

Föreläsning 2

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

- Kort repetition
- Större programmeringsexempel
  - Hur strukturera kod
- Subrutiner
- Stack
- Ordlängder, SI-enheter
- Adresseringsmoder

Praktiska kommentarer

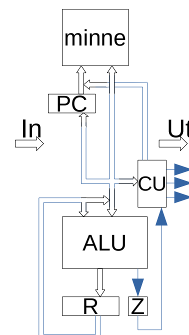
- Labanmälan öppnar måndag 23/1 kl 12.30
  - Anmälningssystemet via lisam
  - Varje labb uppdelad i två delar (2h + 2h), välj lämpligen två tillfällen med tid emellan
  - Jobba så långt ni hinner under de första 2 timmarna
  - Väl förberedd kan bli klar med lab redan efter första 2 timmarna
- Jobba i 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

Enkel exempeldator

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

Maskininstruktionsformat

Typ	Argument
xxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxx
4	28

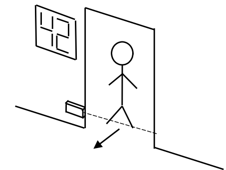


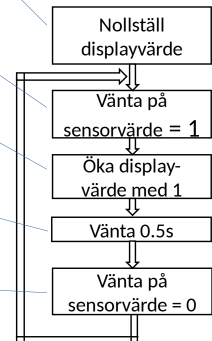
TypKod	Assemblerinstruktion	Föklaring
0	mov R,#värde	Ladda in argument i
1	add R,#värde	Addera argument till R
2	cmp R,#värde	Jämför argument med R, Z=1 om R=argument, Z=0 annars
3	bne adress	Sätt PC till argument om Z=0
4	beq adress	Sätt PC till argument om Z=1
5	b adress	Sätt PC till argument
6	bl 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

Räkna studenter

- Assemblerkod
  - Ingen adress given
  - Översätts till maskinkod av assemblator
- I minnet ligger bara maskinkod
  - Binära värden

```
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
```





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!

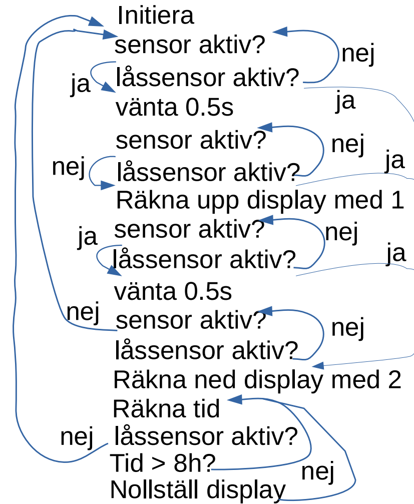
Ersätt med riktig assemblerkod

- 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
```

Subrutiner, stack

Återanvändning av kod

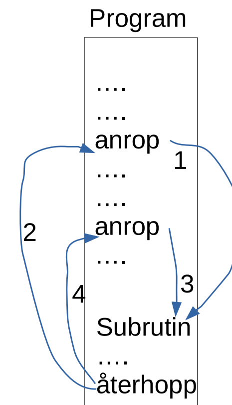
- Många likadana sekvenser i koden
  - Fördröjning på 0.5s ser likadan ut på två ställen

```
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

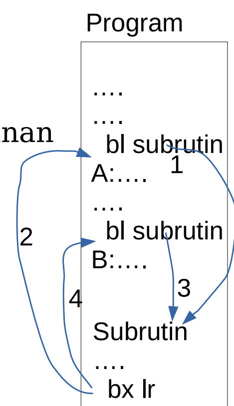
Subrutiner

- 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.

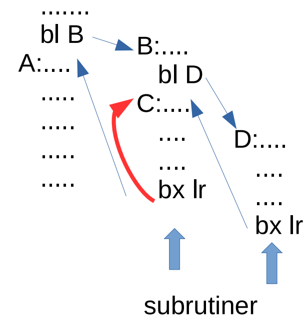
- Vid anrop till subrutin
  - bl subrutinaddr ; hopp till subrutin som startar på adressen subrutinaddr i minnet
  - Sparar adressen till instruktionen efter bl (finns i PC-registret innan hoppet utförs, A respektive B i exemplet) i ett register (extra register kallat lr)
- När subrutinen är slut
  - 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….

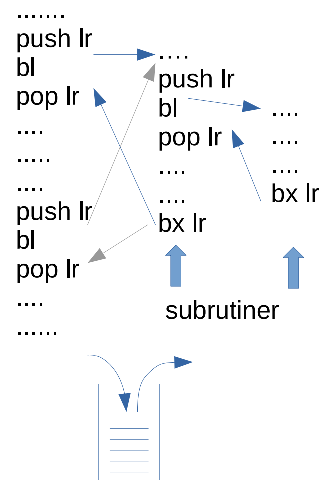
- Rekursion kräver möjlighet att lagra flera anrop i rätt ordning
  - Om bara ett register kan användas tappas första återhoppsadressen bort vid andra anropet
- Exempel
  - Första bl => återhoppsregister LR = A
  - Andra bl => återhoppsregister LR = C (ersätter värdet A).
  - Återhopp i subrutin D =>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

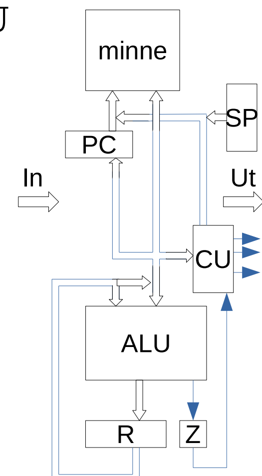


Implementering av en stack

- Ett speciellt (dedikerat) minne inuti processorn för stack
  - Ineffektivt
  - 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

Adr	Data
:	:
95	xxx
96	xxx
97	xxx
98	xxx
99	xxx
:	:

Program växer nedåt (mot högre adresser)

Adr	Data
:	:
99	xxx
98	xxx
97	xxx
96	xxx
95	xxx
:	:

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):
  - Stack växer mot lägre adresser, pekar på senast placerade data

	Adr	Data
	:	:
	95	xxx
	96	xxx
	97	xxx
SP →	98	456
	99	123
	:	:

Växer nedåt (mot lägre adresser)

	Adr	Data
	:	:
	95	123
	96	456
	97	789
SP →	98	abc
	99	xxx
	:	:

Växer uppåt (mot högre adresser)

Exempel på sekvens av stackoperationer

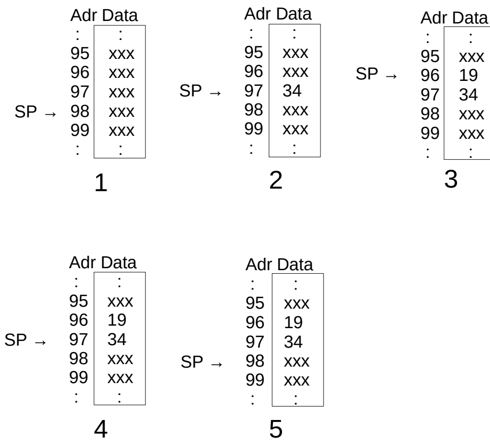
1: SP = 98
- Okänt innehåll (xxx) i minnet

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).

Ordlängd, SI-prefix, Minnesaccess, minnesadressering

En dators ordlängd (antal bitar i data)

- Register och minnen lagrar data bestående av flera bitar
  - Ofta längder som är ”jämna”, t ex 4, 8, 12, 16, 24, 32, 64, 80
- Några standardiserade namn på ordlängder
  - Nibble: 4 bitar (talområde 0-15)
  - Byte: 8 bitar (talområde 0-255)
  - Word/long: varierar i olika datorfamiljer vad som menas
    - Byte ⩽ word ⩽ long
    - Exempel: word = 16 bitar, long = 32 bitar
    - Vissa (t ex ARM) användre också notationer som halfword etc.
      - Halfword = 16 bit, word = 32 bit, doubleword = 64 bit

Antal bitar i register hos en processor

- Notationen ”X-bitars dator” indikerar maximalt antal bitar i dataord som hantera på en gång
  - Ofta är dataregister (motsvarande R-registret i exempeldatorn X bitar stort
    - 8-bitars dator arbetar med 8-bitars data internt i processorn
    - Matchar ofta storleken på data in i ALU
  - Undantag finns
    - Extra långa register för adresser, speciella funktioner etc.
- Adresslängd ofta dubbelt så lång som dataordlängden för små datorer
  - 8-bitars datorer har 16 bitars adressrymd (64 kilobyte adressrymd)
  - 16-bitars datorer har 32-bitars adressrymd (4 gigabyte adressrymd)
  - 32-bitars datorer ofta 32 bitars adressrymd
  - Inte så intressant längre i och med 64-bitars datorer (2^64 = drygt 1.8E19, dvs 16 exabyte)

SI-prefix och datorer

- SI-prefix är alltid bas 1000 (K = 1000, M=1000 000)
  - Närmaste "jämna" 2-potens är 1024 (2^10)
- Tidigare antogs K i datasammanhang motsvara 1024
  - 64 Kbyte minne = 64*1024 = 65536 byte (= antal kombinationer för 16 bitars adress)
- Numera: Jämna tvåpotenser mindre viktigt
  - Vanliga SI-prefix (bas 1000) används numera (t ex hårddiskar, datahastigheter etc)
  - Kvarvarande 1024: Primärminne, cache etc. räknas i KB etc. där K = 1024, M=1024*K, G=1024*M
- Nya enheter finns definierade av IEC, men inte alltid använda
  - Kibi (kilobinary) = 1024, Mebi (megabinary) = 1024 Kibi, Gibi(gigabinary) etc.
  - Jag använder "slarvigt" fortfarande K=1024, M=1048576 etc.

Minnesadressering

- Minne beskrivs oftast som en vektor av byte
  - Adress väljer vilken byte som ska läsas/skrivas
  - Även datorer som hanterar längre ordlängd 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 processor
- Även 16, 32 och 64-bitars processorer beskriver oftast minnet som en uppsättning byte
  - Kan vara begränsad till att alltid ha adresser som är jämnt delbara med 2 eller 4.

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)

Adress
00 03
04
08
12
16
20
32 bitar

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

0011001000010000

00010000 00110010
adress n n+1

00110010 00010000
adress n n+1

Närmare beskrivning av olika minnesläsningar

- Många olika instruktioner för att flytta värden så här långt (samma mnemonic i vissa fall, olika typ, olika kodning)

  mov R,#värde

  ldr R,adress

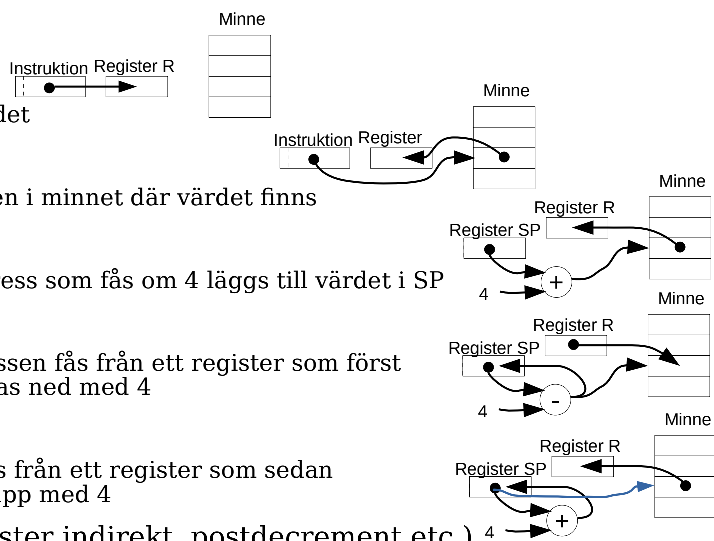
  mov SP,#värde

- Finns också instruktioner som liknar load

  Push R => str R,[-1,SP]! ; minska SP med 1, flytta R till adress ; som nya värdet på SP registret pekar på

  Pop R => ldr R,[SP],#1 ; läs värde från minne på adress i SP, öka ; SP med 1.

Adresseringsmoder



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

In och utmatning

Användning av bl i exempeldator

- bl är subrutinanrop som gömmer koden för hur sensorer och display hantera
  - Kallas ibland för drivrutiner (byte hårdvara kräver bara byte av drivrutin)
  - Någon måste fortfarande skriva rutinerna och bygga in gränssnittet i datorn
- Dessa subrutiner är exempel på olika I/O-funktioner
  - Skicka ut data till display
  - Läsa av sensor och låssensor
- Hur?
  - Måste finnas en del i modelldatorn där display och sensorer kan anslutas
  - Värde i minne eller register måste kunna skickas ut, värde från sensor kunna läsas av

In och utdata, speciella instruktioner

- Vissa datorer använder speciella instruktioner för att skicka data in och ut från I/O
  - Kräver extra instruktioner
    - in R,ioadress ; läs av inport nr ioadress och sparar i R
    - out R,ioadress ; skicka värde i R till utport nr ioadress
  - Kräver extra signaler ut från processorn (styrsignaler)
    - Anger om data ska till/från I/O eller minne
  - In och utdata påverkar inte tillgänglig mängd minne
  - Eget adressutrymme för I/O separat från vanligt minne
    - out R,100 ; R:s värde ut på I/O enhet 100, INTE i minne adress 100
    - str R,100 ; skriver till minne adress 100, påverkar inte I/O-enheter

In och utdata, minnesmappad I/O

- Vissa datorer (dom flesta numera) använder speciella minnesadresser för att ta emot och skicka data från/till I/O
  - Kräver inga extra instruktioner eller extra styrsignaler
  - Enkelt att hantera i högnivåspråk (t ex C/C++)
  - Kräver att vissa minnesadresser inte kan användas som vanligt minne
    - Läsning kan ge annat värde än det som skrivs till adressen
- Används av labutrustningen

```
; display finns på adress
; 65532, sensor på 65534

putdisplay:        ; adress 2000
        str R,65532
        bx lr

readsensor:        ; adress 2200
        ldr R,65534
        bx lr
```

Avkodning av minnesmappad I/O

- Exempel: vill ha en I/O-port på adress 0001 0010 0011 0100.
  - Alternativ 1: fullständig avkodning
    - Alla 16 bitarna måste undersökas
    - Kräver stor logisk nät (hårdvara) för att detektera rätt adress (långsamt, dyrt)
    - Numera ganska vanligt (hårdvara billig och snabb)
  - Alternativ 2: ofullständig avkodning
    - Avkoda bara en del av bitarna (t ex bara de första 12)
    - Går snabbare, tar mindre hårdvara
    - Porten hamnar då på alla adresser som börjar med 0001 0010 0011, dvs 16 olika
      - Skrivning till 0001 0010 0011 0100 samma som skrivning till 0001 0010 0011 1100

Kuriosa: Bitbanding (alternativ användning av adressavkodning, finns i labutrustningen…)

- Vissa I/O-register placerade på många adresser
- Del av adressen används för att nollställa vissa bitar när adressen läses
  - Exempel: I/O-port på adress 0x12300-0x123FF (alla adresser pekar på samma port).
  - Bit 9-2 i adressen anger om biten som läses från porten ska nollställas. Dvs bit 9 i adress måste vara 1 för att bit 7 i inläst värde ska vara portens värde. Om bit 9 i adress = 0 så nollställs alltid bit 7 i inläst värde från porten.
  - Exempel: Läsning på adress 0x123c6 kommer nollställa bitarna 7, 6, 5 och 0 i värdet som läses.
  - Orsak till att labbutrustningen läser adress 0x4002507c för port F eftersom bit 7-5 inte kan användas. Alla bitar läses om adress 0x4002503fc används istället.
-

Hopp

LINKÖPINGS UNIVERSITET

Olika typer av hopp

- Branch (ARM: b)
  - Ovillkorligt (utförs alltid)
- BNE (Branch Not Equal)
  - Villkorligt, hoppa om icke lika (antag jämförelse gjorts innan)
- BEQ (Branch EQual)
  - Villkorligt, hoppa om lika (antag jämförelse gjorts innan)
- BL (Branch and Link)
  - Branch and link, subrutinanrop som sparar var nästa instruktion efter subrutinanropet är
- BX (Branch eXchange)
  - Återhopp från subrutin, återställer PC till värde efter BL-anrop

LINKÖPINGS UNIVERSITET

Relativa och absoluta hopp

- Skillnad mellan absoluta och relativa hopp
  - Brukar finnas både jump (absolut) och branch (relativt)
  - Även möjligt med BSR och JSR (branch respektive jump subroutine)
- Absolut hopp: Adressen i argumentet är adress
  - Exakt den adress som PC ska få när hoppet är utfört
  - Fördel: behöver inte beräknas, kan peka på fast adress även om kod flyttas
- Relativt hopp: argumentet är ett avstånd relativt aktuell PC
  - Argumentet anger hur många steg framåt eller bakåt hoppet ska göras relativt aktuellt adress i PC
  - Fördel: hopp inom rutinen behöver inte ändras även om rutinen placeras på annan plats i minnet

LINKÖPINGS UNIVERSITET