Datorteknik TSEA82 + TSEA57
Fö5

Adresseringsmoder

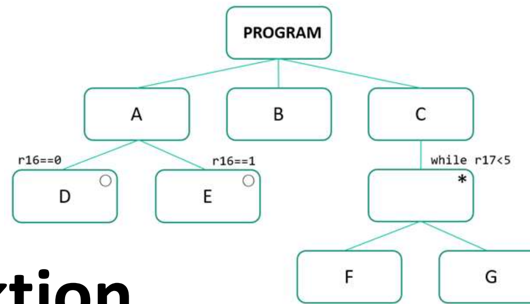
LINKÖPINGS UNIVERSITET

Datorteknik Fö5 : Agenda

- Repetition
- Adresseringmoder
- Labb 2
- Labb 1 : Tips
- Tid för frågor

Repetition

Från JSP till assembler



Sekvens

```
PROGRAM:
    call A
    call B
    call C
LOOP:
    ...
    jmp  LOOP


B:
    ...
    ret
```

Sekvens

Selektion

```
A:
    cpi  r16,0
    brne A_1
    call D
    jmp  A_EXIT
A_1:
    cpi  r16,1
    brne A_DEFAULT
    call E
    jmp  A_EXIT
A_DEFAULT:
    ...
A_EXIT:
    ret
```

Selektion ○

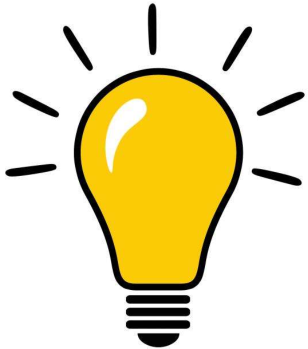
Iteration

```
C:
    clr  r17
C_LOOP:
    cpi  r17,5
    breq C_EXIT
    call F
    call G
    inc  r17
    jmp  C_LOOP
C_EXIT:
    ret
```

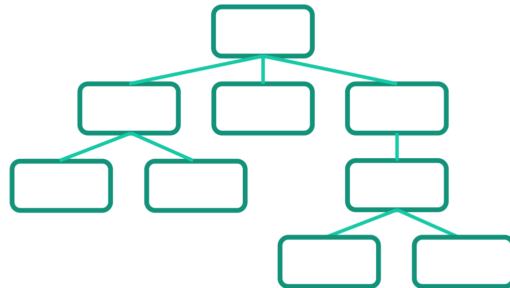
Iteration *

Från idé till program

Idé



Strukturerad lösning
t ex via JSP



Programkod
.asm

```
      .org 0
      jmp  MAIN

      .dseg $0100
ARR:
      .byte $0C

      .cseg
MAIN:
      ldi  r16,HIGH
      out  SPH,r16
      ldi  r16,LOW
      out  SPL,r16
      ...
```

Maskinkod
.hex

```
:1001000049726F6E
:100110006D616964
:10012000656E7275
:100130006C657A20
:
:
:
:
:
:
```

Övergripande princip

Definiera först, använd sen

Vektortabell : I huvudsak, hopp till huvudprogram men avbrottsvektorer

Minnesanvändning : I huvudsak, deklaration av var minnesanvändningen börjar, men även deklaration av variabler i minnet.

Subrutiner : Definition av subrutiner, gärna grupperade i samhörande moduler.
Även definition av avbrottsrutiner.

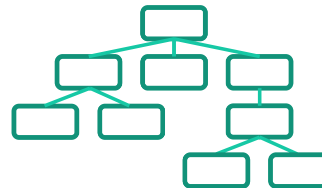
Huvudprogram : Initiering av variabler/data, följt av huvudloop med anrop av subrutiner och villkorsstyrda förgreningar.

Vektortabell

Definition av **minnesanvändning**

Definition av **subrutiner**/ underfunktioner

Huvudprogram med huvudloop



```
        .org 0
        jmp  MAIN

        .dseg $0100
ARR:
        .byte $0C

        .cseg
SUB1:
        push r16
        ...
        pop  r16
        ret

        ...

MAIN:
        ldi  r16,HIGH
        out  SPH,r16
        ldi  r16,LOW
        out  SPL,r16
MAIN_LOOP:
        ...
        call SUB1
        ...
        jmp  MAIN_LOOP
```

Prioritetsordning

1. **Funktion**: Programmets funktion är överordnat, men nästan lika viktigt är nästa punkt, struktur.
2. **Struktur**: För att ett program ska fungera bra, vara smidigt att utveckla och vidareutveckla, vara läsbart och senare kunna optimeras är dess struktur avgörande.
3. **Optimering**: När funktion och struktur finns på plats blir optimering ofta enkelt. Det handlar då typiskt om att stegvis reducera ”randvillkor” mellan olika subrutiner eller moduler, om nu detta är nödvändigt. En skicklig och erfaren programmerare kan säkert uppnå extremt optimerad kod genom gå direkt på optimeringssteget och hoppa över struktureringen, men sådan kod blir ofta obskyr och mer eller mindre omöjlig att vidareutveckla.

Adresseringsmoder

CISC vs RISC

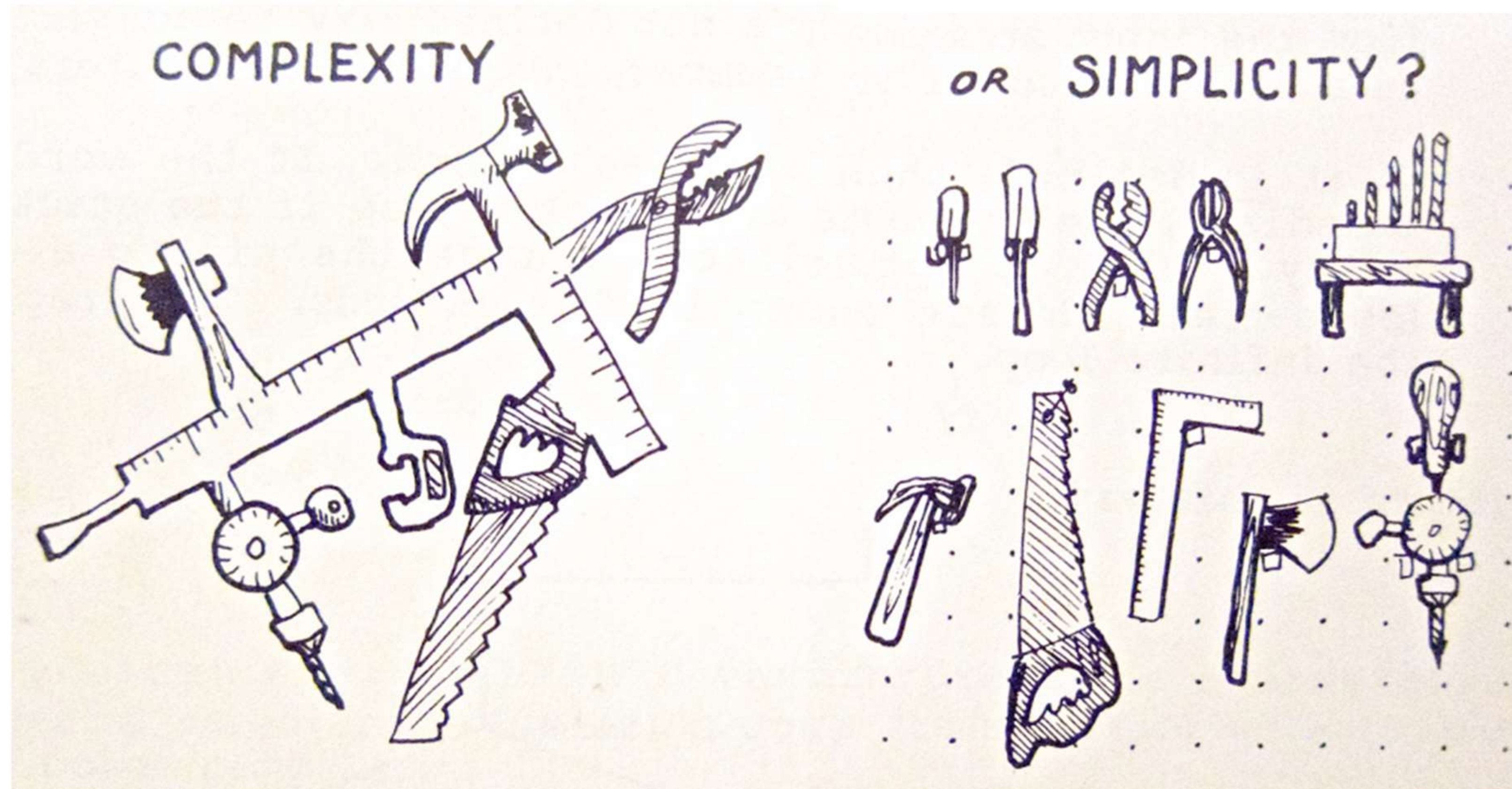
CISC (Complex Instruction Set Computer)

- Instruktioner som klarar många kombinationer av argument och adresseringsmoder
- Varje instruktion tar ett flertal klockcykler att utföra
- Många instruktioner kan ofta jobba direkt mot minnet
- En stor del av chip-ytan går åt för att implementera och avkoda instruktioner
- Numera är ”bara” Intel x86 CISC
- Moderna Intel-processorer kan sägas vara hybrider av CISC och RISC

RISC (Reduced Instruction Set Computer)

- Adresseringsmoden är vanligen inbyggd i instruktionen, dvs det krävs typiskt fler instruktioner än för CISC
- Varje instruktion tar ett fåtal (ofta bara 1) klockcykler att utföra
- Endast load-store-instruktioner kan jobba direkt mot minnet
- Chip-ytan kan användas för minnen och register istf instruktionsavkodning
- ARM är en typisk RISC-processor, även AVR (RISC-mikrokontroller)

CISC or RISC



Effektiv Adress, EA

I samband med adresseringsmoder brukar man prata om det som kallas effektiv adress, EA.

EA är den slutliga adress som argumentet till en instruktion läses ifrån alternativt skrivs till.

Via olika adresseringsmoder ’beräknas’ den effektiva adressen först, innan sedan instruktionen gör det den ska.

Ju mer komplex en adresseringsmod är, ju mer beräkning (hårdvara) går det åt för att få fram den effektiva adressen.

Adresseringsmoder

AVR-mikrokontrollern kan hantera följande adresseringsmoder

Mod	Exempelkod			
1.Omedelbar	`subi`	`r20,$12`		
2.Register direkt	`com`	`r16`	`add`	`r20,r21`
3.Data direkt (Absolut)	`lds`	`r20,$A3`	`sts`	`$A4,r20`
4.Data indirekt (Indirekt)	`ld`	`r20,X`	`st`	`Y,r16`
- med förskjutning (offset)	`ldd`	`r20,Y+$05`	`std`	`Z+3,r20`
- med post-inkrement	`ld`	`r20,X+`	`st`	`Y+,r16`
- med pre-dekrement	`ld`	`r20,-X`	`st`	`-Y,r16`

Mod 1 - Omedelbar

Omedelbar (immediate) adresseringsmod innebär att argumentet till instruktionen finns i instruktionens omedelbara närhet, i själva instruktionsordet.

En omedelbar instruktion har bokstaven i (för immediate) inbakad i instruktionen, för att indikera att det är en omedelbar instruktion.

T ex:

```
ldi   r16,$A7       ; ladda r16 med värdet $A7
subi  r22,35        ; ladda r22 med värdet 35
```

Argumenten `$A7` och `35` är konstanter som är omedelbart tillgängliga, dvs det behöver inte göras någon adressberäkning för att få fram den effektiva adressen EA.

Mod 2 – Register direkt

Register direkt innebär att argumentet finns i ett generellt register, dvs att utpekat generellt register utgör källan till argumentet.

Beroende på instruktion, och hur den är skriven, så kan källregistret även utgöra destinationsregister.

T ex:

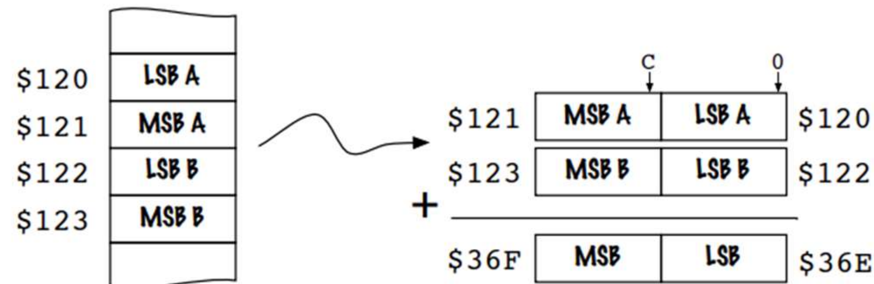
```
com   r16           ; invertera värdet i r16
add   r20,r21       ; addera värdet i register r21 till r20
out   PORTB,r16     ; skriv värdet i r16 till PORTB
eor   r16,r16       ; gör xor med värdet i r16 med r16
```

Här behövs (görs) ingen adressberäkning för att få fram den effektiva adressen EA.

Mod 3 – Data direkt (absolut adress)

För data direkt (absolut adress) anges den effektiva adressen EA som en konstant direkt i instruktionen. EA är den minnesadress där argumentet finns.
T ex: Addera 16-bitarstalen på adress `$120` och `$122`, spara på adress `$36E`

```
lds   r16,$120      ; lds och sts påverkar inte flaggor
lds   r17,$122
add   r16,r17
sts   $36E,r16
lds   r16,$121
lds   r17,$123
adc   r16,r17
sts   $36F,r16
```

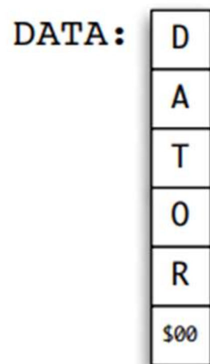


Bokstaven `s` i `lds` och `sts` indikerar att adressen är statisk, dvs för att använda några andra adresser i programmet ovan så måste det skrivas om.

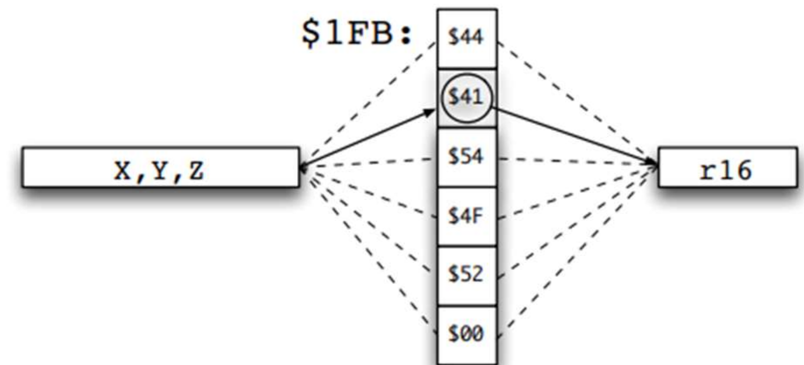
Mod 4 – Data indirekt

Den indirekta adresseringsmoden medför användning av pekare. Pekaren innehåller den effektiva adressen EA, och i den adressen finns argumentet.

Exempel:
Läs en NUL-terminerad sträng, ASCII-kodad, ur dataminnet SRAM med början på adress `$1FB`.

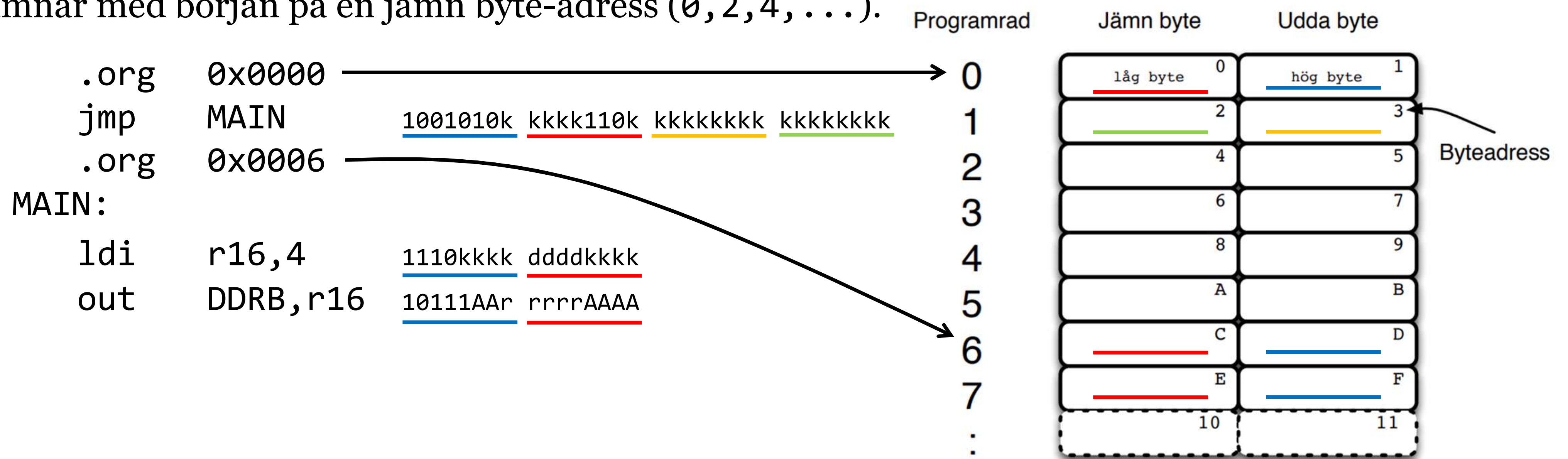



```
    ldi   XH,HIGH(DATA)
    ldi   XL,LOW(DATA)
NEXT:
    ld    r16,X ; indirekt
    cpi   r16,0
    breq  KLAR
    call  PROCESS
    adiw  XL,1
    jmp   NEXT
KLAR:
```



Flash-minnet (programminnet)

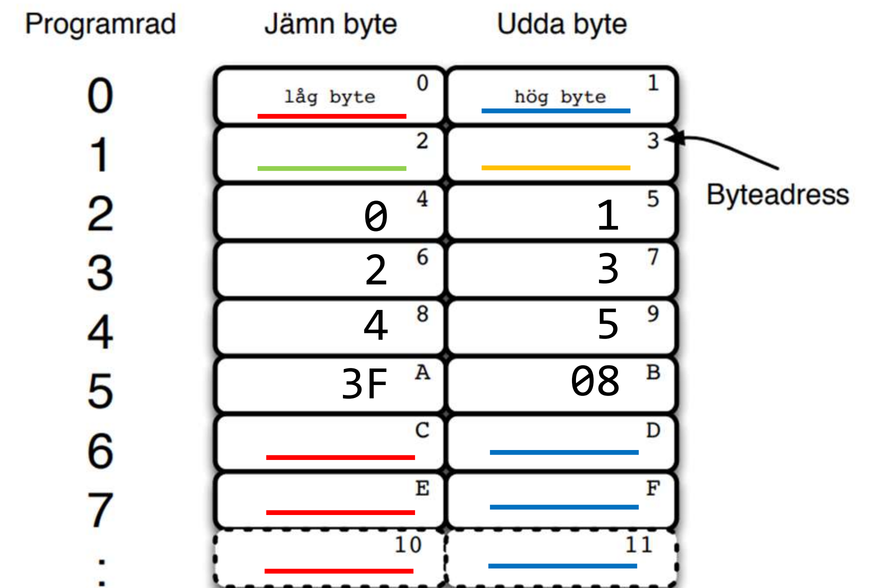
Flash-minnet (programminnet) är 16 bitar brett och används primärt för att lagra instruktioner, dvs maskinkoden för själva programmet. Eftersom varje programrad (`0,1,2,...`) alltså upptar 16 bitar (två byte, dock för vissa instruktioner fyra byte) så medför det att varje maskinkodsinstruktion hamnar med början på en jämn byte-adress (`0,2,4,...`).



Mod 4 – Data indirekt

Med instruktionen lpm och Z-pekaren kan man peka ut enstaka byte i programminnet.
Innehållet i Z utgör den effektiva adressen EA.
T ex: Läs data ur tabell i programminnet (flash-minnet)

```
    .org    0x0000
    jmp     LOOKUP
TAB:
    .db     0,1,2,3,4,5      ; jämnt antal byte
    .dw     0x083F
LOOKUP:
    ldi     ZH,HIGH(TAB*2)   ; ladda tabellstart
    ldi     ZL,LOW(TAB*2)
    lpm     r16,Z            ; hämta 0
    call    PROCESS          ; gör nåt med datat
    adiw    ZH:ZL,1          ; peka ut nästa
    lpm     r16,Z            ; hämta 1
    call    PROCESS          ; gör nåt med datat
    ...                      ; osv
```

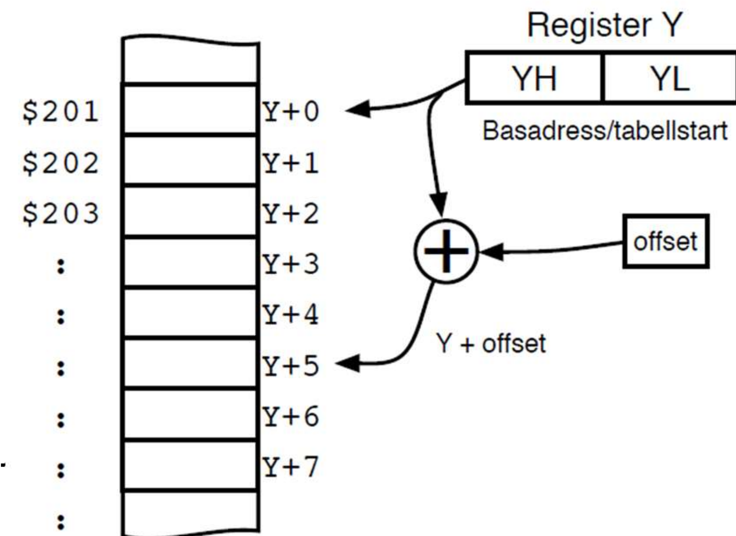


Mod 4 – Data indirekt (indexerad) med förskjutning

Indirekt adressering med förskjutning kallas även indexerad med förskjutning. Den använder dels ett indexregister/pekarregister (Y eller Z) som basadress + förskjutning (offset) med en konstant. Summan av pekare+offset utgör den effektiva adressen EA.

Exempel:
Kopiera en byte från Y+6 till Y+1 med basadress Y= $201

```
ldi    YH,HIGH($201)    ; set base adress
ldi    YL,LOW($201)
ldd    r16,Y+6          ; load with displacement
std    Y+1,r16          ; store with displacement
```



Mod 4 – Data indirekt med post-inkrement

Med indexerad adressering vill man ofta ta sig fram till nästa element. Det kan göras manuellt med adiw, men också med hjälp av post-inkrement. Observera plus-tecknets placering efter Z.
T ex: Läs tecken för tecken ur tabell i programminnet, med Z-pekaren och post-inkrement

```
STRING:   .db    ”DATORTEKNIK”,0
SEND:     ldi    ZH,HIGH(STRING*2)   ; set Z to start of STRING
          ldi    ZL,LOW(STRING*2)
SEND_NEXT:
          lpm    r16,Z+              ; load with post increment
          cpi    r16,0               ; NUL?
          breq   SEND_DONE
          call   PROCESS
          jmp    SEND_NEXT
SEND_DONE:
```

Mod 4 – Data indirekt med pre-dekrement

Man kan även minska indexet, då med pre-dekrement. Observera minus-tecknets placering före Z. Det går inte att göra post-dekrement eller pre-inkrement.

Exempel
Sök baklänges i Minnet (SRAM) med början på adress $3FF, tills talet 7 hittas.

```
FIND:       ldi     ZH,HIGH($400)     ; set Z to $3FF+1
            ldi     ZL,LOW($400)      ;
FIND_PREV:
            ld      r16,-Z            ; decrement Z first then load
            cpi     r16,7             ; 7?
            breq    FIND_DONE
            jmp     FIND_PREV
FIND_DONE:
```

Angående load/store, pekare och minnen

Load och store-instruktionerna har beroende på variant begränsade möjligheter för att ibland bara använda vissa pekare, och kan bara arbeta mot antingen SRAM (dataminne) eller Flash (programmine) beroende på instruktion.

Instruktion	Pekare och/eller konstant	Minne
ldi	konstant värde	---
lds, sts	konstant adress	SRAM
ld, st	X,X+,-X, Y,Y+,-Y, Z,Z+,-Z	SRAM
ldd, std	Y+konstant, Z+konstant	SRAM
lpm	Z,Z+	Flash

SPM – Store Program Memory

Likväl som att man kan läsa från programminnet (Flash) så kan man skriva till programminnet.
Det är det som händer när en Arduino programmeras med att nytt program.
Instruktionen `SPM` skriver hela block med data åt gången. `SPM` används inte i kursens labbar.

Arduinon har en bootloader, som efter reset väntar på maskinkod (hex-data) och programmerar sedan sig själv, via `SPM`, med det man 'häller in' i den.

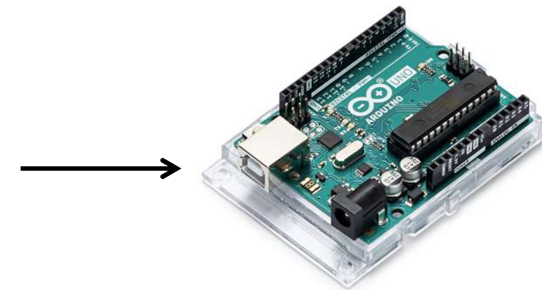
Om man inte skickar in någon maskinkod så körs istället det program som redan finns.

Maskinkod

.hex

```
:1001000049726F6E
:100110006D616964
:10012000656E7275
:100130006C657A20
```

Bootloader med `SPM`



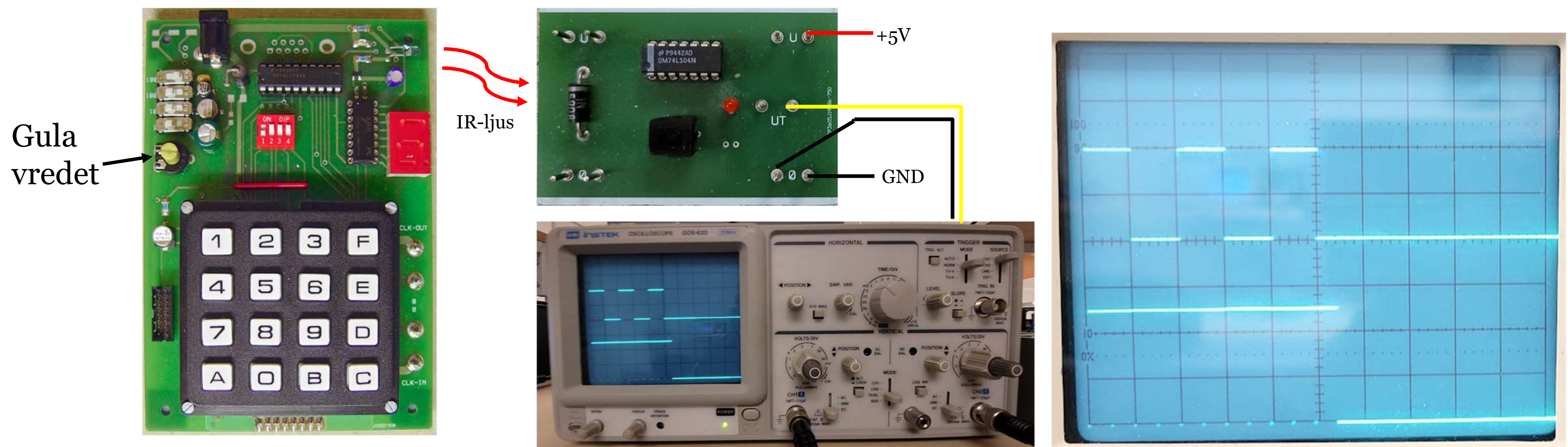
Labb 2

...

Tips Labb 1

Tips Labb 1

Börja med att koppla in IR-mottagaren till oscilloskopet (t ex `CH1`) med inställningar enligt oscilloskopet nedan (t ex `2 ms`). Tryck på (håll ner) A och vrid sedan på gula vredet tills pulståget liknar oscilloskopbilden till höger.



Tid för Frågor

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET