

# Datorteknik TSEA82 + TSEA57

## Fö9

A/D-omvandling

# Datorteknik Fö9 : Agenda

- Repetition Preprocessor & macro
- A/D-omvandling
- Lab 4
- LAX
- Tid för frågor

# Repetition Preprocessor & macro

# Preprocessor och kompilering

Preprocessorn förstår ett antal direktiv, som alltså inte är egentlig assemblerkod utan just direktiv för att definiera, strukturera och tolka den övriga programkoden.

Preprocessordirektiven finns för att man enklare och tydligare ska kunna skriva och strukturera sin programkod.

Dom är egentligen inte nödvändiga för att åstadkomma det som programmet ska göra, men underlättar kodandet och gör det tydligare och lättare att läsa det resulterande programmet.

Direktiv	Namn	Betydelse
<code>.org</code>	<i>origin</i>	Skriv här (adress)
<code>.byte</code>	<i>byte</i>	Reservera byte i SRAM
<code>.dseg</code>	<i>data segment</i>	Följande gäller SRAM
<code>.cseg</code>	<i>code segment</i>	Följande gäller programminnet
<code>.eseg</code>	<i>extra segment</i>	Följande gäller EEPROM
<code>.def</code>	<i>define</i>	Döp register till namn
<code>.equ</code>	<i>equate</i>	Döp konstant
<code>.db</code>	<i>define byte</i>	Skriv följande <i>byte</i> (8-bit) i minnet
<code>.dw</code>	<i>define word</i>	Skriv följande <i>word</i> (16-bit) i minnet
<code>.macro</code>	<i>macro</i>	"copy-paste" av följande
<code>.endmacro</code>	<i>endmacro</i>	...avsluta ett macro
<code>&lt;&lt; n</code>	<i>shift left</i>	vänsterskift <i>n</i> bitar
<code>&amp;,  , ^</code>	<i>logical AND, OR, XOR</i>	bitvis OCH, ELLER, XOR
<code>+, -, *, /</code>	<i>arithmetic</i>	som förväntat
<code>HIGH, LOW</code>	<i>high low</i>	ger höga resp låga delen av följande uttryck

# Preprocessordirektiv

**.org** sätter kompilatorn (dvs kodgenereringen) till en specifik adress i SRAM- eller Flash-minnet. Den adressen räknas automatiskt upp vid den fortsatta kodgenereringen.

**.cseg** anger att efterföljande kod ska hamna i programminnet.

**.db** definierar tabellvärden i programminnet (Flash)

```
.cseg                ; default
.org $0000
jmp START
;
;   avbrottsvektorer
;
.org INT_VECTORS_SIZE ; 52
TAB: .db 1, 2, 3, 4 ←————— Inte körbar kod, dvs exekveringen får inte komma hit.
START:
;   programstart
```

# Preprocessordirektiv

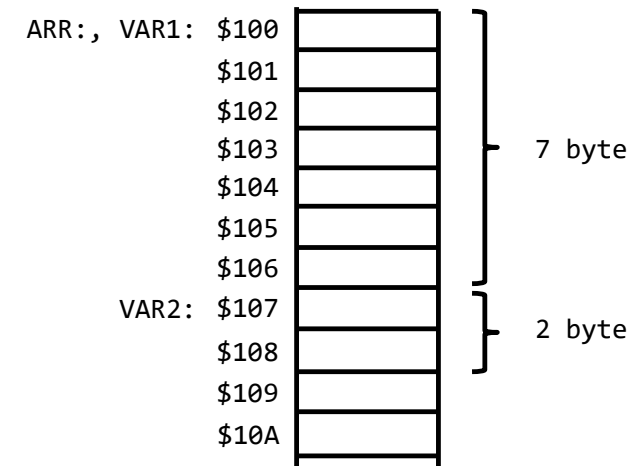
**.org** sätter kompilatorn (dvs kodgenereringen) till en specifik adress i SRAM- eller Flash-minnet. Den adressen räknas automatiskt upp vid den fortsatta kodgenereringen.

**.dseg** anger att efterföljande definitioner ska använda SRAM (dataminnet).

**.byte** reserverar ett antal byte för variabler, bara det. Det går inte att tilldela värden till dessa variabler här.

```
.dseg
.org $100      ; adress $100 i SRAM
ARR:          ; ARR=$100, handtag till struct nedan
VAR1: .byte 7 ; VAR1, adressen till 0-te byten av dessa 7
VAR2: .byte 2 ; VAR2, adressen till första lediga efter VAR1

.cseg
; till programminnet igen
lds r16,VAR1
lds r17,VAR2
...
```



# Preprocessordirektiv

Följande visar på vilka möjligheter som preprocessorn ger oss när vi ska skriva kod.

Alla kodrader ger samtliga exakt samma resultat efter att preprocessorn gjort sitt.

```
ldi r16,65
ldi r16,$41
ldi r16,0x41
ldi r16,0b01000001
ldi r16,(1<<6)|(1<<0)
ldi r16,0xF1&0x4F
ldi r16,'A'
ldi r16,8*8+1
ldi r16,HIGH($4122)
ldi r16,LOW($2241)
ldi r16,HIGH(16674)
ldi r16,LOW(8769)
```

```
ldi r16,0x41
```

Även följande ger samma resultat.

```
.equ N = $40
.def tmp = r16

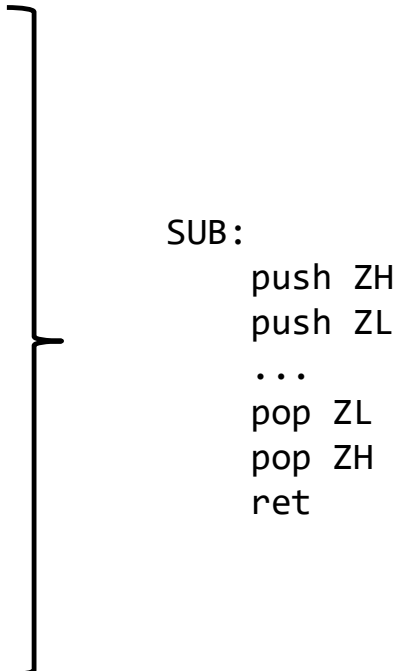
ldi tmp,N+1
ldi tmp,N|(1<<0)
ldi tmp,HIGH((N+1)<<8)
ldi tmp,LOW(N|1)
```

# Macron

Macron definierar ett kodstycke som ska kopieras in i koden. Det är inte detsamma som en subrutin, utan fungerar snarare som en ”stämpel” som preprocessor använder när den genererar kod.

```
.macro  PUSHZ
    push ZH
    push ZL
.endmacro

.macro  POPZ
    pop  ZL
    pop  ZH
.endmacro
...
SUB:
    PUSHZ
    ...
    POPZ
    ret
```



```
    SUB:
        push ZH
        push ZL
        ...
        pop  ZL
        pop  ZH
        ret
```



# Macro

Macro kan även definieras med argument, vilket gör det lite mer användbart och dynamiskt. Man skulle t ex kunna definiera den saknade instruktionen ADDI:

```
.macro  ADDI          ; macro med argumenten @0 och @1
      subi @0,-@1
.endmacro

...

ADDI   r20,3          ; r20 = r20 + 3
...
subi   r20,-3         ; r20 = r20 - (-3)
```

```
subi   r20,-3
...
subi   r20,-3
```

# Macron

Macron ska dock användas sparsamt och ha väl valda namn. Annars blir koden snabbt obegriplig och svårläst, eftersom man egentligen skapar nya ord i grammatiken som inte tillhör språket från början. Dvs, den oinvigde måste själv göra översättningar av alla macron för att förstå koden.

Man får dessutom inte se vad macrot gör vid simulering, utan det bara utförs.

Utan att ha alla tidigare macro-definitioner i huvudet blir det svårt att veta vad följande kod gör:

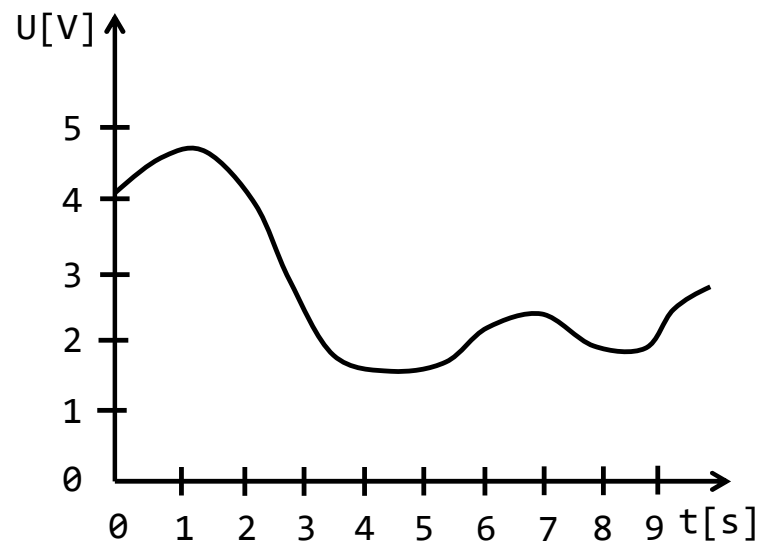
```
.macro ...  
.endmacro  
...  
LAST      r22  
MIX       r17,r22  
PUT       r17,4  
BOX  
LAST      Z+  
...
```

**Grundregel:** Undvik macron om det inte är en *jättebra* idé och har en entydig och lättolkad funktion.  
Använd macron sparsamt.

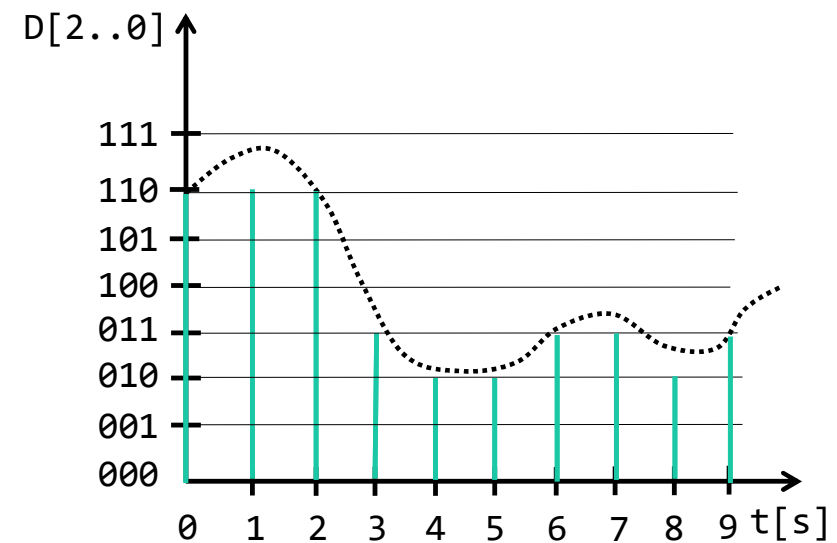
# A/D-omvandling

# A/D-omvandling

Analog.  
Amplitudkontinuerlig och tidskontinuerlig



Digital.  
Amplituddiskret och tidsdiskret



Kvantiseringseffekter med förlorad precision.

# A/D-omvandling

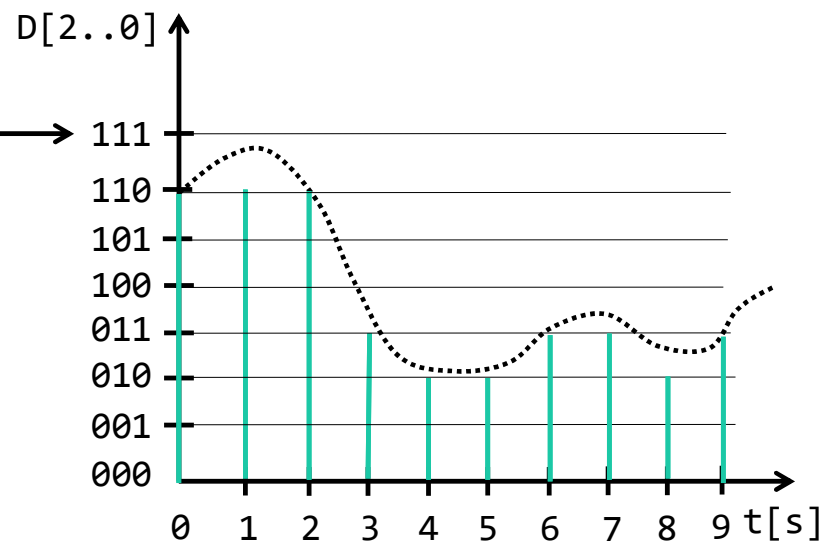
Vid A/D-omvandling jämförs den analoga insignalen med en referensspänning, AREF, som motsvarar taket, dvs den högsta tänkbara spänning som insignalen får ha.

Om insignalen uppnår AREF, så resulterar det i ett digitalt resultat med alla bitar 1-ställda.

En princip för A/D-omvandling är successiv approximation. Då jämförs den analoga insignalen stegvis med respektive bits motsvarande spänning, med början på mest signifikant bit. Om jämförelsen är mindre, behålls biten (som 1:a), annars tas den bort, och sedan lägger man till spänningen för nästa bit.

För en 3-bitars omvandling skulle det då ta 3 cykler att komma fram till rätt digitalt värde.

Digital.  
Amplituddiskret och tidsdiskret



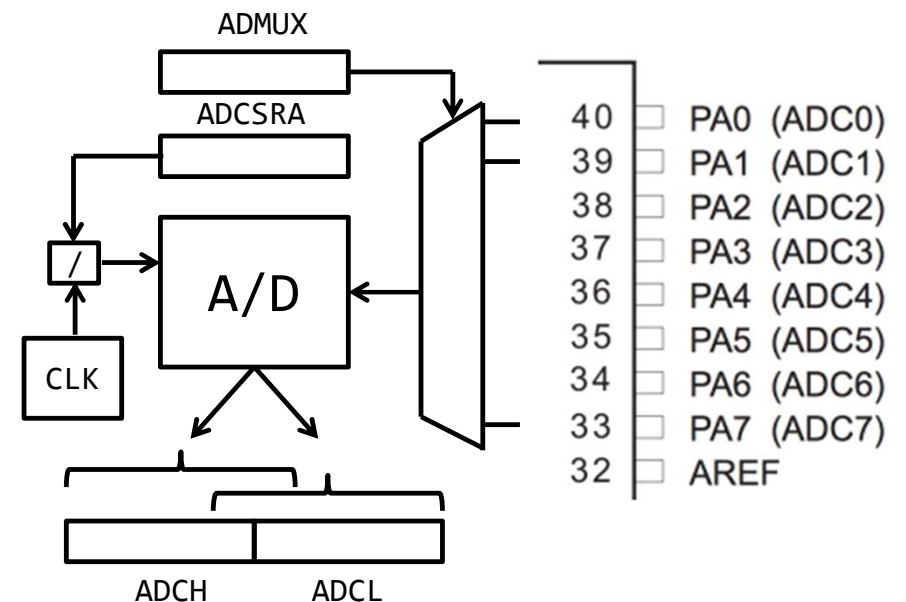
Kvantiseringseffekter med förlorad precision.

# A/D-omvandling

ATMega16 har en 10-bitars A/D-omvandlare, vilket innebär att det digitala värdet har  $2^{10} = 1024$  nivåer, dvs 0-1023 ( $0000000000_2 - 1111111111_2$ ). Resultatet sparas i registerparet ADCH:ADCL, skiftat till vänster eller höger.

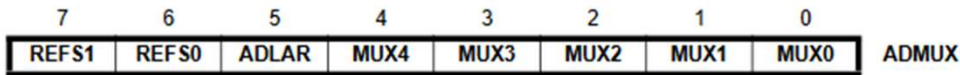
A/D-omvandlaren kan styras till att omvandla en av åtta analoga ingångar (ADC7-ADC0), via en multiplexer. ADMUX-registret väljer analog ingång, men även referensspänning, AREF, och resultatets skift.

ADCSRA är ett kontrollregister som aktiverar A/D-omv. Sätter omvandlingstakt m m.



# A/D-omvandling, ADMUX

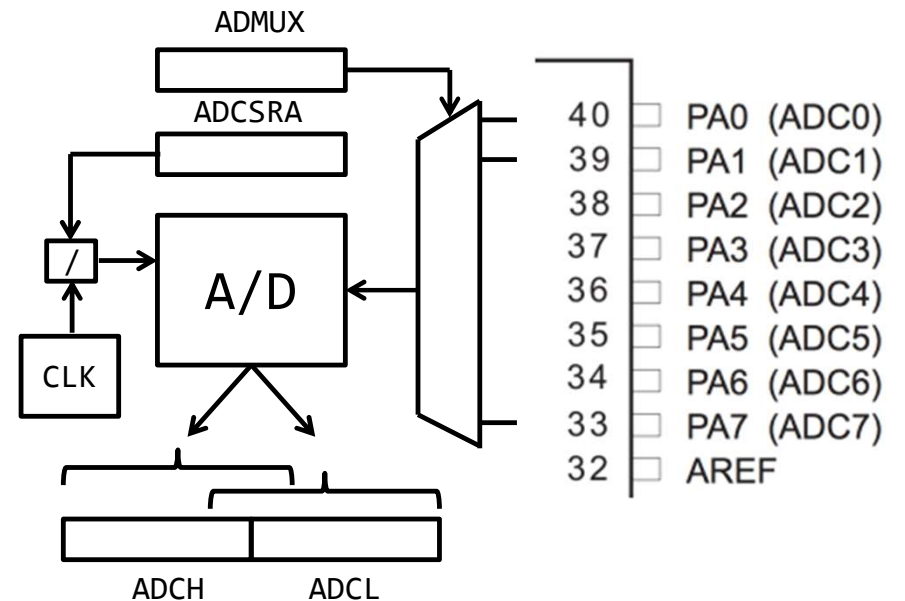
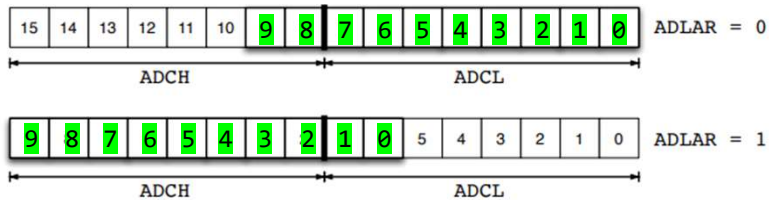
ADMUX väljer analog ingång, var referensspänningen kommer ifrån, samt justering av resultatet.



Referensspänningen kan komma från extern ingång AREF, AV<sub>CC</sub> eller från en intern spänning på 2.56V.

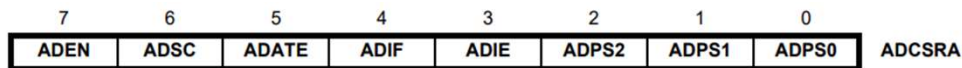
REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AV <sub>CC</sub> with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Resultatet på 10 bitar ryms inte i ett register utan placeras till vänster eller höger inom registerparet ADCH:ADCL



# A/D-omvandling, ADCSRA

ADCSRA aktiverar A/D-omvandlaren (ADEN), styr omvandlingstakten (ADPSx), startar en omvandling (ADSC) m m.

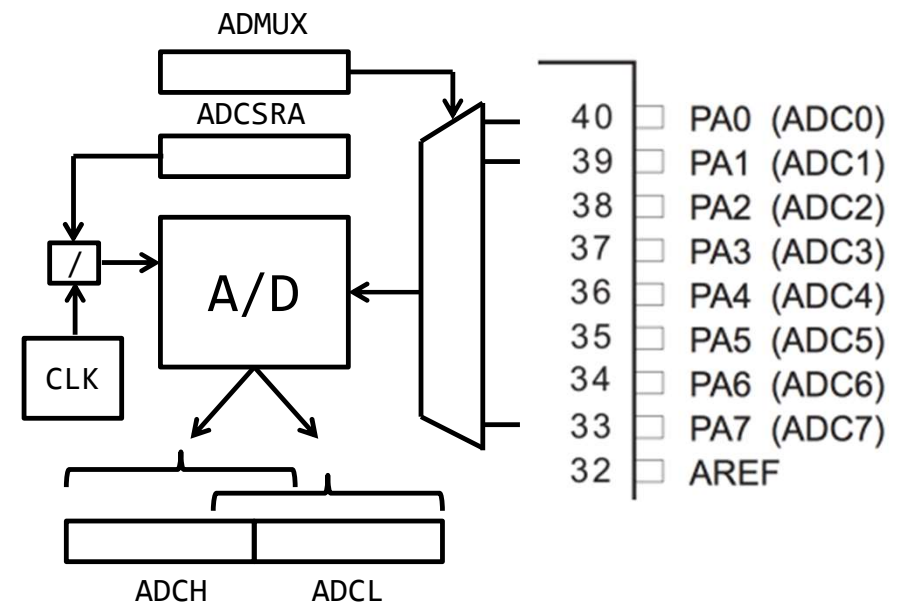


Omvandlingstakten är max 15kSPS (enl. datablad), och en omvandling tar 13 cykler. Det medför att en processorklocka på t ex 16 MHz måste skalas ned innan den används i A/D-omv.<sup>1</sup>

Table 21-5. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

En omvandling startas genom att 1-ställa ADSC-biten, och ADSC-biten 0-ställs automatisk när omvandlingen är klar.



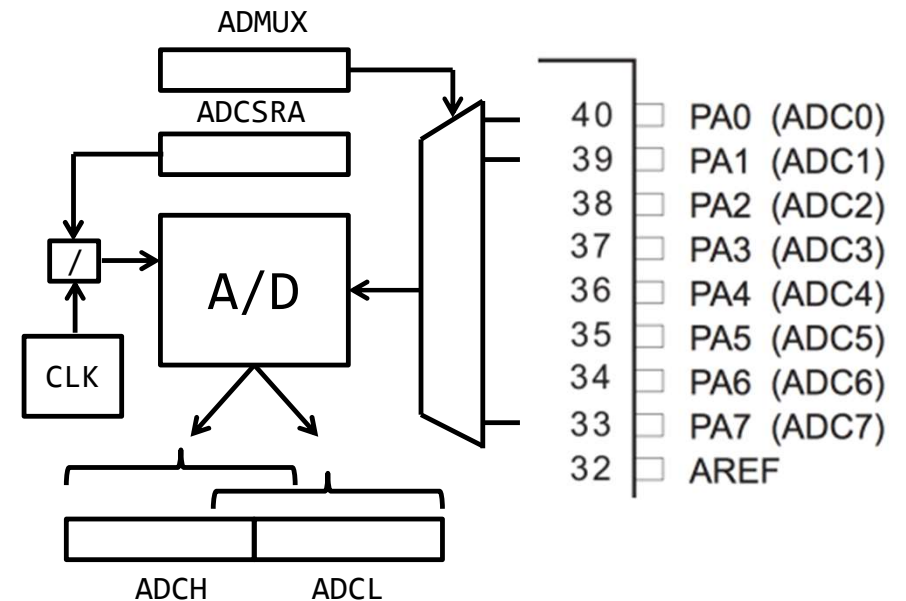


# A/D-omvandling, 10-bitars

## Exempel: 10-bitars omvandling

```

ADC10:
    ldi    r16,(1<<REFS0)    ; kanal 0, AVCC ref, ADLAR=0
    out   ADMUX,r16
    ldi    r16,(1<<ADEN)     ; A/D enable, ADPSx=111
    ori    r16,(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
    out   ADCSRA,r16
ADC10_CONVERT:
    in     r16,ADCSRA
    ori    r16,(1<<ADSC)
    out   ADCSRA,r16        ; starta omvandling
ADC10_WAIT:
    in     r16,ADCSRA
    sbrc  r16,ADSC          ; om 0-ställd, klar
    rjmp  ADC10_WAIT       ; annars vänta
    in     r16,ADCL         ; obs, läs låg byte först
    in     r17,ADCH         ; hög byte sedan
  
```



Resultatet återfinns i registerparet r17:r16

(Pågående omvandling)



Föregående omvandling



# A/D-omvandling, 8-bitars

## Exempel: 8-bitars omvandling

```

ADC8:
    ldi    r16,(1<<REFS0)|(1<<ADLAR)    ; kanal 0, AVCC ref
                                           ; left adjust

    out   ADMUX,r16

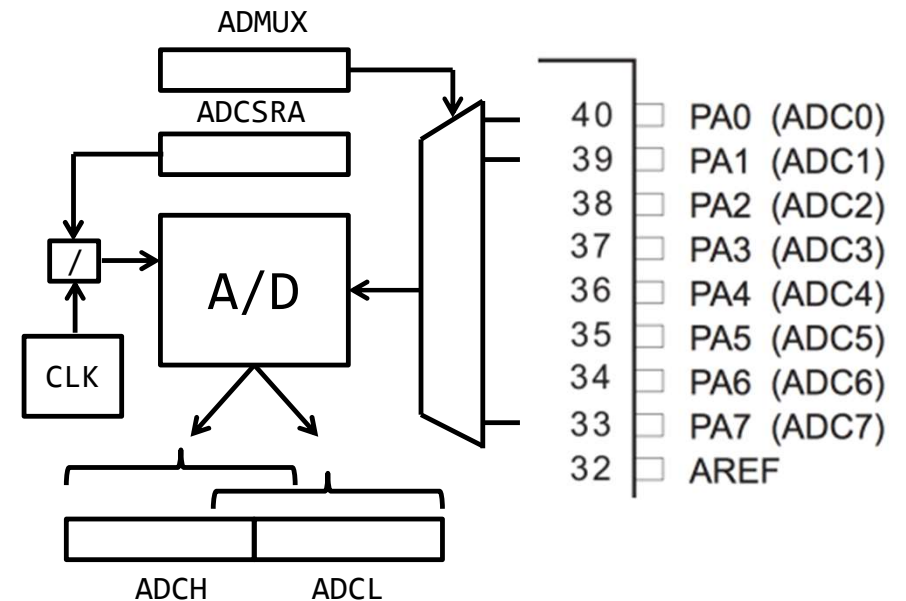
    ldi    r16,(1<<ADEN) ; A/D enable, ADPSx=111
    ori    r16,(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
    out   ADCSRA,r16

ADC8_CONVERT:
    in    r16,ADCSRA
    ori    r16,(1<<ADSC)
    out   ADCSRA,r16           ; starta omvandling

ADC8_WAIT:
    in    r16,ADCSRA
    sbrc  r16,ADSC           ; om 0-ställd, klar
    rjmp  ADC8_WAIT         ; annars vänta
    in    r16,ADCH           ; en läsning av hög byte
  
```

Resultatet återfinns i register r16.

Egentligen görs en 10-bitars omvandling, men endast de 8 höga bitarna används. Dvs, något sämre precision.



# Labb 4

...

LAX

## LAX

- LAX:en går 23-25/5
- LAX:en är 90 minuter
- Anmälan i Lisam
- LAX:en görs enskilt, dvs inget samarbete
- Hjälpmedel: Datablad (delas ut vid LAX:en)
- Det finns övnings-LAX:ar på kurshemsidan
- Labbar + LAX = godkänd kurs

# Tid för Frågor

Anders Nilsson

[www.liu.se](http://www.liu.se)