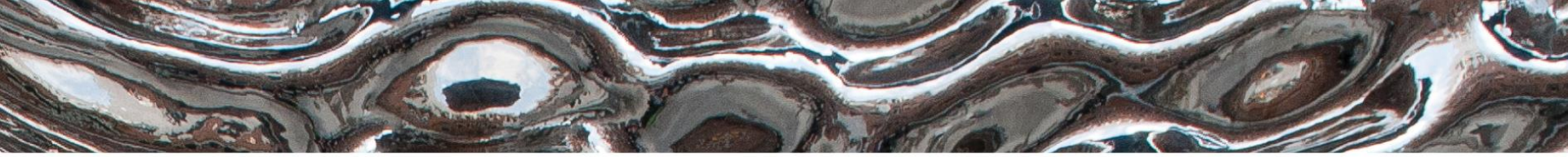


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Del 2

CPU-en, den sentrale prosesseringsenheten

I dag

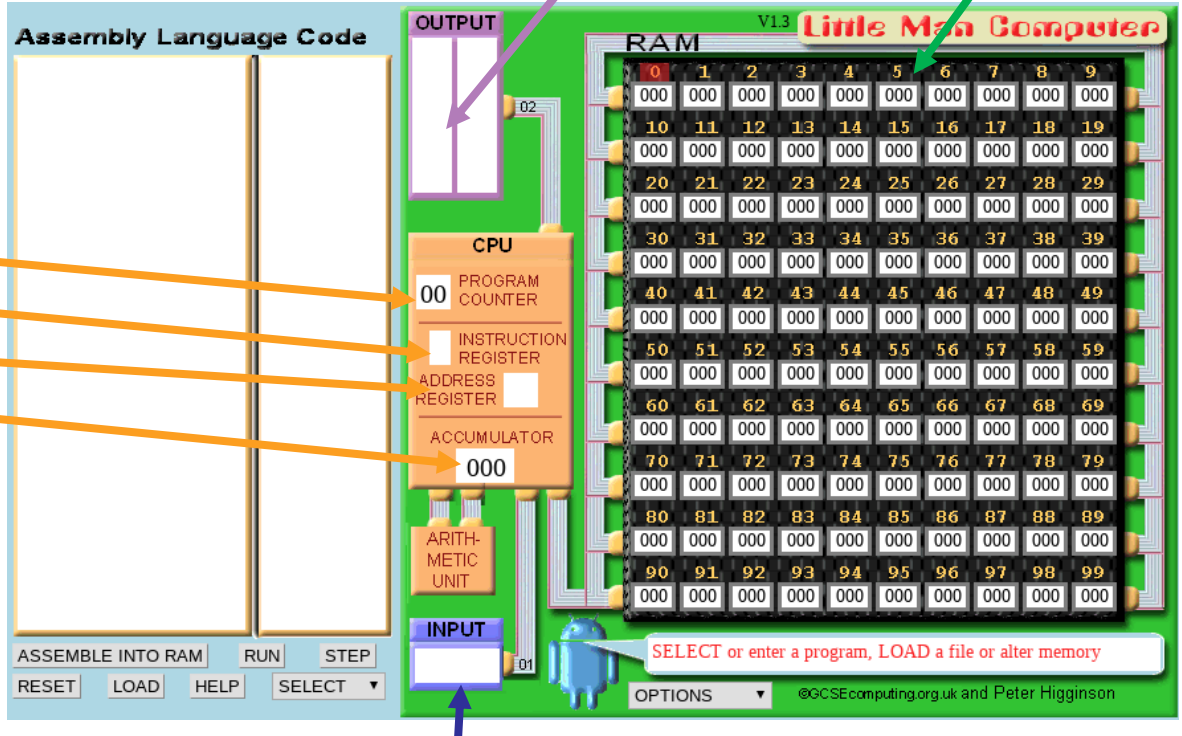
- Innføring i assemblernotasjon
  - Repetisjon fra torsdag
- Resterende instruksjoner for LMC
  - Printing av ASCII
  - Løkker og hopp
- Flere LMC eksempler

LMC-overblikk

Resultater skrives ut her

Minne med 100 celler for program og data

Fire registre
- Program counter
- Instruction register
- Address register
- Accumulator



LMC kan lese tall som brukeren (vi) skriver inn

Husker dere hva dette programmet gjorde?

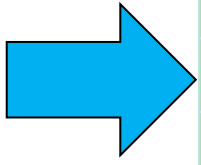
```
901 306 901 106 902 000
```

Alle programmer er lagret som tall når de kjøres, men:

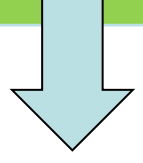
- Ikke lett å huske alle tallkodene, vanskelig å forstå ut fra kontekst
- Hvilke(n) minnelokasjoner (celler) inneholder data?
- Det er fort gjort å skrive en adresse feil
- Om vi endrer programmet, må vi også endre flere instruksjoner

Fra Maskinkode til Assemblerkode

Adresse i minnet	Instruksjon
0	901
1	306
2	901
3	106
4	902
5	000



Instruksjon	Adressedel	Kommentar
INP		// Les inn tall 1
STA	06	// Lagre i minnet
INP		// Les inn tall 2
ADD	06	// Legg til fra minnet
OUT		// Skriv ut svaret
HLT		// Stopp!



Hva med vårt mellomlagrede tall på adresse 6?

Pseudoinstruksjonen DAT

- Hvordan sier vi:
  - «Adresse 6 brukes til DATA»
- **DAT <verdi>**
  - Deklarerer en variabel i minnet
  - Initialiseres med **verdi**

Addresse	Instruksjon	Adresse del
0	INP	
1	STA	6
2	INP	
3	ADD	6
4	OUT	
5	HLT	
6	DAT	0

Bruk av etiketter («labels»)

- Dette er fremdeles rotete
  - Hva hvis vi legger til flere instruksjoner?

Alle disse instruksjonene sin adressedel må nå endres til «7»

Addresse	Instruksjon	Adresse del
0	INP	
1	STA	7
2	INP	
3	ADD	7
4	ADD	7
5	OUT	
6	HLT	
7	DAT	0

En ekstra ADD

Data i adresse 6 flytter seg til adresse 7

Bruk av etiketter «labels»

- En **etikett** «holder» adressen til en assembler instruksjon
  - Feks. **tall** til høyre ☺
- Bruk etiketten i adressedelen
- Assembleren finner ut hvor i minnet «ting» havner

Etikett ☺ ☺	Instruksjon	Adresse del
	INP	
	STA	tall
	INP	
	ADD	tall
	ADD	tall
	OUT	
	HLT	
tall	DAT	0

Oppsummering av assembler

- Assembler forenkler programmering!
  - Kanskje litt lenger vei å gå for å gjøre det *virkelig* enkelt (?)
  - Kan bli uhyre effektivt (og ganske morsomt)

- Håndtering av data i minnet
  - **DAT** brukes til å si "her ligger det et tall til mellomregning"
  - **Etiketter** brukes til å slippe å huske *hvor* alt ligger
  - (Og **kommentarer** gjør det klart for mennesker hvordan programmet virker)

- Sjekk "innføring til LMC" for å øve litt på dette!

Alle instruksjonene i LMC

Kode	Assembler notasjon	Beskrivelse	LMC-1	LMC-2
0xx	HLT	Stopper eksekveringen	☑	
1xx	ADD	Adderer verdien i angitt minnelokasjon med akkumulatoren	☑	
2xx	SUB	Subtraherer verdien i angitt minnelokasjon med akkumulatoren	☑	
3xx	STA	Lagrer akkumulatoren i angitt minnelokasjon	☑	
5xx	LDA	Henter verdi fra minnet til akkumulatoren	☑	
6xx	BRA	Hopper til angitt adresse		☑
7xx	BRZ	opper til angitt adresse hvis akkumulatoren er 0		☑
8xx	BRP	kkumulatoren er ≥ 0		☑
901	INP	svaret i akkumulatoren	☑	
902	OUT	Skriver ut verdien i akkumulatoren	☑	
922	OTC	Skriver ut ASCII-tegn (ikke i boka)		☑

Dette kalles «mnemonics»

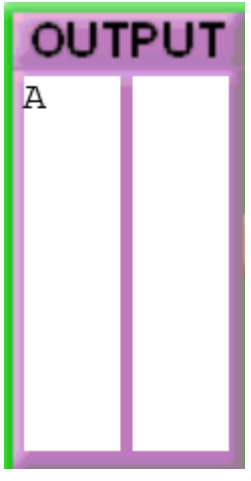
Veien til «Hello world»

```
>>> print("Hello World!")
Hello World!
>>>
```

Printing av tekst – instruksjonen OTC

- **OTC** virker litt som OUT..
  - Men i stedet for å skrive ut akkumulatoren..
  - ..så skriver den ut akkumulatoren som et ASCII tegn!

- Eksempel: Bokstaven A kodes som 65_{10}
  - **LDA** <adresse med tallet 65>
  - **OTC**
  - Gir da ----------→



ASCII

H: $48_{16} = 4*16 + 8 = 72_{10}$

e: $65_{16} = 6*16 + 5 = 101_{10}$

i: $69_{16} = 6*16 + 9 = 105_{10}$

Første vellykkede forsøk på standardisering var ASCII (American Standard Code for Information Interchange) i 1963; den har 128 tegn:

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hei, verden!

- Bruk DAT for å «forhåndslagre» ascii-koder for tall
  - Ikke glem at DAT <verdi> initialiserer minnet til <verdi> !
- Bruk etiketter for å laste inn ascii-kodene i akkumulator

Adresse	Mnemonic	Adressedel
0	LDA	7
1	OTC	
2	LDA	8
3	OTC	
4	LDA	9
5	OTC	
6	HLT	
7	DAT	72
8	DAT	101
9	DAT	105

Enklere ASCII-tabell?

10	LF	39	'	47	/	62	>	93	]	124	|
32		40	(	48	0	63	?	94	^	125	}
33	!	41	)	:		64	@	95	_	126	~
34	"	42	*	57	9	65	A	96	‘		
35	#	43	+	58	:	:		97	a		
36	$	44	,	59	;	90	Z	:			
37	%	45	-	60	<	91	[	122	z		
38	&	46	.	61	=	92	\	123	{		

De mest nødvendige tegnene, i base 10.

Løkker og Hopping

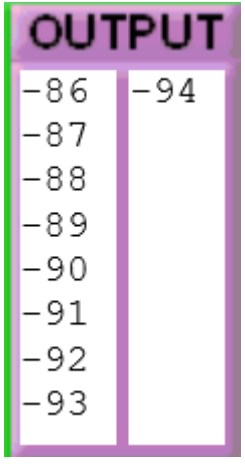
```
a = 10
while( True ):
      print(a)
      a = a - 1
```

Hopp med BRA

Instruksjonen **BRA** («Branch») med kode **6xx** gjør at programmet hopper til et annet sted i minnet. Dette lar oss blant annet lage løkker

```
        LDA     v10     // Start med 10
TELL    OUT             // Skriv ut
        SUB     v1      // og tell ned.
        BRA     TELL    // Gjenta
        HLT             // Stopp

v1      DAT     1       // Konstanten 1
v10     DAT     10      // Konstanten 10
```



La oss kjøre det i LMC →

Betinget hopp – instruksjonen BRP

Ofte ønsker vi å hoppe noen ganger, men ikke *alltid.*

Instruksjonen **BRP** («Branch if positive») har kode **8xx**.
Den hopper til adressen **xx** hvis akkumulatoren inneholder en verdi ≥ 0; ellers gjør den ingen ting.

```
        LDA     v10   // Start med 10
TELL    OUT           // Skriv ut
        SUB     v1    // og tell ned.
        BRP     TELL  // Gjenta om akkumula
        HLT           // Stopp

v1      DAT     1   // Konstanten 1
v10     DAT     10  // Konstanten 10
```

```
a = 10
while( a >= 0 ):
        print(a)
        a = a - 1
```

La oss kjøre det i LMC →

Instruksjonen BRZ

Instruksjonen **BRZ** («Branch if zero») har kode **7xx**. Den utfører et hopp til adressen **xx** dersom akkumulatoren inneholder verdien 0; ellers gjør den ingenting.

Programmeringseksempler i LMC

Eksempel: Multiplikasjon

Vi ønsker å multiplisere to tall.

1. Har vi en instruksjon som gjør det?
2. Kan vi oppnå samme resultat med én eller flere andre instruksjoner?

Vi vet at

```
a * b = b + b + ... + b,
```

a ganger.

Da kan vi lage en *løkke* med addisjoner, side LMC har instruksjoner for dette

Multiplikasjon i LMC; en skisse

Det kan ofte være lurt å skissere programmet med «pseudo-kode» først:

1. Les inn *a* og *b*.
2. Start med resultatverdi 0.
3. Så lenge a != 0:
   - Øk resultatverdien med *b*.
   - Sett a = a – 1 (gjør en subtraksjon med 1)
4. Skriv ut resultatverdien.

```
START INP      // a =
      STA a    //   int(input(""))
      INP      // b =
      STA b    //   int(input(""))

LOOP  LDA a    //
      BRZ EXIT // while a != 0:
      SUB v1   //   a = a - 1
      STA a    //

      LDA Res  //   Res = Res + b
      ADD b    //
      STA Res  //

      BRA LOOP //

EXIT  LDA Res  //
      OUT      // print( Res )
      HLT      //

a     DAT
b     DAT
Res   DAT 0

v1    DAT 1
```

Nå har vi vært innom alle instruksjonene
Hva kan vi nå programmere?

- Variabler (med DAT) og minneadresser (etiketter)
- Tilordning (med STA og LDA)
- While-løkker (med BRA, BRZ og BRP)
- If-tester (med BRZ og BRP)
- Regning
  - Legge sammen (med ADD)
  - Trekke fra (med SUB)
  - Gange
  - Dele (med kode fra ukeoppgave)

Instruksjoner og data

Hva er problemet med dette programmet?

```
        INP             // Les inn
        STA     x       // verdien x.
        ADD     x       // Doble den.
        OUT             // Skriv den ut.

x       DAT     0       // Variabel x
```

La oss kjøre det i LMC →

Det mangler en HLT-instruksjon. Hva skjer da når vi leser inn verdien **600**? Hva med **101**?

Hvordan vet LMC om en celle inneholder en instruksjon eller en dataverdi?

Det vet den ikke!

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv    LDA txt
         BRZ Ferdig
         OTC
         LDA Skriv
         ADD v1
         STA Skriv
         BRA Skriv
Ferdig   HLT
```

```
v1       DAT 1 // Konstant 1
txt      DAT 86 // 'V'
         DAT 101 // 'e'
         DAT 108 // 'l'
         DAT 107 // 'k'
         DAT 111 // 'o'
         DAT 109 // 'm'
         DAT 109 // 'm'
         DAT 101 // 'e'
         DAT 110 // 'n'
         DAT 33 // '!'
         DAT 0 // Slutt
```

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv    LDA txt
         BRZ Ferdig
         OTC
         LDA Skriv
         ADD v1
         STA Skriv
         BRA Skriv
Ferdig   HLT

v1       DAT 1 // Konstant 1
txt      DAT 86 // 'V'
         DAT 101 // 'e'
         DAT 108 // 'l'
         DAT 107 // 'k'
         DAT 111 // 'o'
         DAT 109 // 'm'
         DAT 109 // 'm'
         DAT 101 // 'e'
         DAT 110 // 'n'
         DAT 33 // '!'
         DAT 0 // Slutt
```

Les inn verdien i posisjonen txt (altså 'V')

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv    LDA txt
         BRZ Ferdig
         OTC
         LDA Skriv
         ADD v1
         STA Skriv
         BRA Skriv
Ferdig   HLT

v1       DAT 1 // Konstant 1
txt      DAT 86 // 'V'
         DAT 101 // 'e'
         DAT 108 // 'l'
         DAT 107 // 'k'
         DAT 111 // 'o'
         DAT 109 // 'm'
         DAT 109 // 'm'
         DAT 101 // 'e'
         DAT 110 // 'n'
         DAT 33 // '!'
         DAT 0 // Slutt
```

Hopp om verdien er 0

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv   LDA txt
        BRZ Ferdig
        OTC
        LDA Skriv
        ADD v1
        STA Skriv
        BRA Skriv
Ferdig  HLT

v1      DAT 1 // Konstant 1
txt     DAT 86 // 'V'
        DAT 101 // 'e'
        DAT 108 // 'l'
        DAT 107 // 'k'
        DAT 111 // 'o'
        DAT 109 // 'm'
        DAT 109 // 'm'
        DAT 101 // 'e'
        DAT 110 // 'n'
        DAT 33 // '!'
        DAT 0 // Slutt
```

Skriv ut ASCII-tegn

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv   LDA txt
        BRZ Ferdig
        OTC
        LDA Skriv
        ADD v1
        STA Skriv
        BRA Skriv
Ferdig  HLT

v1      DAT 1 // Konstant 1
txt     DAT 86 // 'V'
        DAT 101 // 'e'
        DAT 108 // 'l'
        DAT 107 // 'k'
        DAT 111 // 'o'
        DAT 109 // 'm'
        DAT 109 // 'm'
        DAT 101 // 'e'
        DAT 110 // 'n'
        DAT 33 // '!'
        DAT 0 // Slutt
```

NB: last instruksjonen som står i posisjon Skriv (5xx) inn i akkumulatoren

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv   LDA txt
        BRZ Ferdig
        OTC
        LDA Skriv
        ADD v1
        STA Skriv
        BRA Skriv
Ferdig  HLT

v1      DAT 1 // Konstant 1
txt     DAT 86 // 'V'
        DAT 101 // 'e'
        DAT 108 // 'l'
        DAT 107 // 'k'
        DAT 111 // 'o'
        DAT 109 // 'm'
        DAT 109 // 'm'
        DAT 101 // 'e'
        DAT 110 // 'n'
        DAT 33 // '!'
        DAT 0 // Slutt
```

Legg til 1 til denne instruksjonen (5xx = 5xx + 1)

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv   LDA txt
        BRZ Ferdig
        OTC
        LDA Skriv
        ADD v1
        STA Skriv
        BRA Skriv
Ferdig  HLT

v1      DAT 1 // Konstant 1
txt     DAT 86 // 'V'
        DAT 101 // 'e'
        DAT 108 // 'l'
        DAT 107 // 'k'
        DAT 111 // 'o'
        DAT 109 // 'm'
        DAT 109 // 'm'
        DAT 101 // 'e'
        DAT 110 // 'n'
        DAT 33 // '!'
        DAT 0 // Slutt
```

Lagre instruksjonen tilbake samme sted – vi har endret programmet!

Selvmodifiserende kode

Når instruksjoner kodes som tall, kan vi endre programmet vårt mens det kjøres.

```
Skriv   LDA txt
        BRZ Ferdig
        OTC
        LDA Skriv
        ADD v1
        STA Skriv
        BRA Skriv
Ferdig  HLT

v1      DAT 1 // Konstant 1
txt     DAT 86 // 'V'
        DAT 101 // 'e'
        DAT 108 // 'l'
        DAT 107 // 'k'
        DAT 111 // 'o'
        DAT 109 // 'm'
        DAT 109 // 'm'
        DAT 101 // 'e'
        DAT 110 // 'n'
        DAT 33 // '!'
        DAT 0 // Slutt
```

Hopp tilbake til Skriv

Array-er

En **array** (som i stor grad tilsvarer en *liste* i Python) er et sammenhengende område av celler i minnet. Vi bruker et heltall (en *indeks*) til å hente riktig element.

Eksempel: Dette lille Python-programmet forteller hvor mange dager en gitt måned har (forutsatt at det ikke er skuddår).

```
1   antallDager = [ 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]
2   m = int(input(""))
3   print(antallDager[m])
```

```
1   antallDager = [ 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]
2   m = int(input(""))
3   print(antallDager[m])
```

```
// Les m som input fra bruker:
        INP
// Legg sammen innlest verdi og LDA_i:
        ADD     LDA_i
// Lagre ny instruksjon i posisjonen hent:
        STA     hent
hent    DAT     0       // Plass til LDA.
        OUT             // Skriv svaret.
        HLT             // Ferdig.

LDA_i   LDA     antDager
```

```
antDager  DAT     0
          DAT    31  // Jan
          DAT    28  // Feb
          DAT    31  // Mar
          DAT    30  // Apr
          DAT    31  // Mai
          DAT    30  // Jun
          DAT    31  // Jul
          DAT    31  // Aug
          DAT    30  // Sep
          DAT    31  // Okt
          DAT    30  // Nov
          DAT    31  // Des
```

Oppsummering

- I moderne maskinarkitekturer er det typisk en beskyttelse som forhindrer at man modifiserer kode som data
  - Dette er av sikkerhetshensyn
  - Moderne prosessorer har i det store og hele fjernet dette behovet
- Men:
  - Operativsystemer og kompilatorer må betrakte brukernes programmer som data
  - Om man lager virus så er dette en veldig aktuell teknikk
  - Enkelte programmeringsspråk støtter også å se på egen kode som data (f.eks. C#, Maude, Common Lisp, etc)

Oppsummering

- Vi kan nå programmere maskinkode!
  - Lese og skrive verdier/input og output
  - Enkle regneoperasjoner
  - If-tester og løkker
  - Array-lignende datastrukturer
- Assembler er en enkel mapping fra maskinkode til tekst
  - Det er vanskelig å lese og forstå en masse tall for mennesker
  - ...men det er det maskinen forstår!
- Vi kan lage selvmodifiserende kode for å få til avanserte ting