

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

CPU-en, den sentrale prosesseringsenheten, del 2

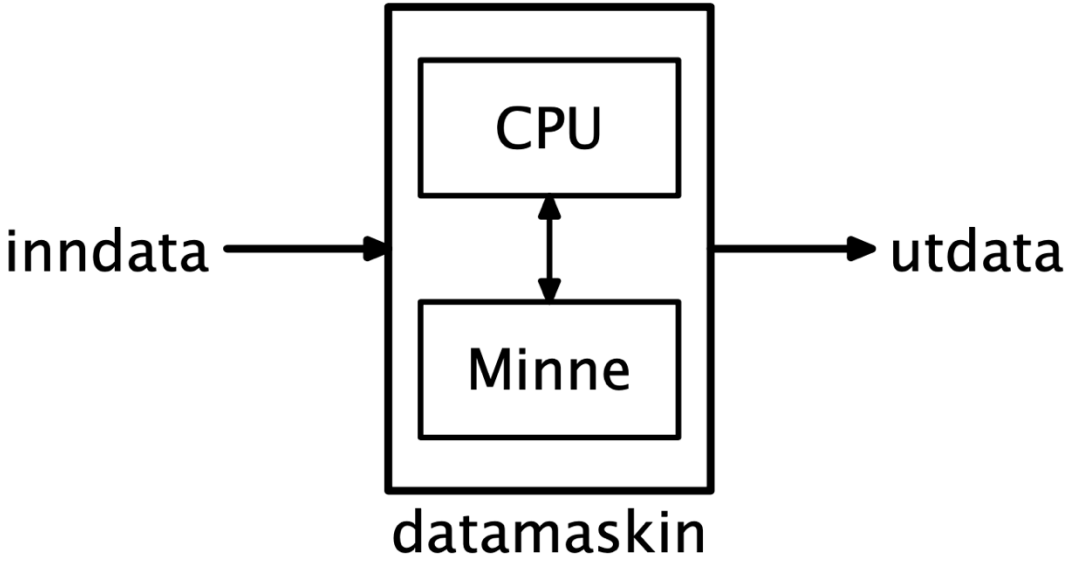
Eyvind W. Axelsen (eyvinda@ifi.uio.no)
Foiler er basert på tidligere materiale laget av Dag Langmyhr



I dag

- Rask oppsummering/repetisjon fra sist
- Mer om assemblerkode
- Hvordan programmere med dette?
  - Instruksjoner og data
  - Løkker og if-tester
  - Array-er

Von Neumann-arkitekturen



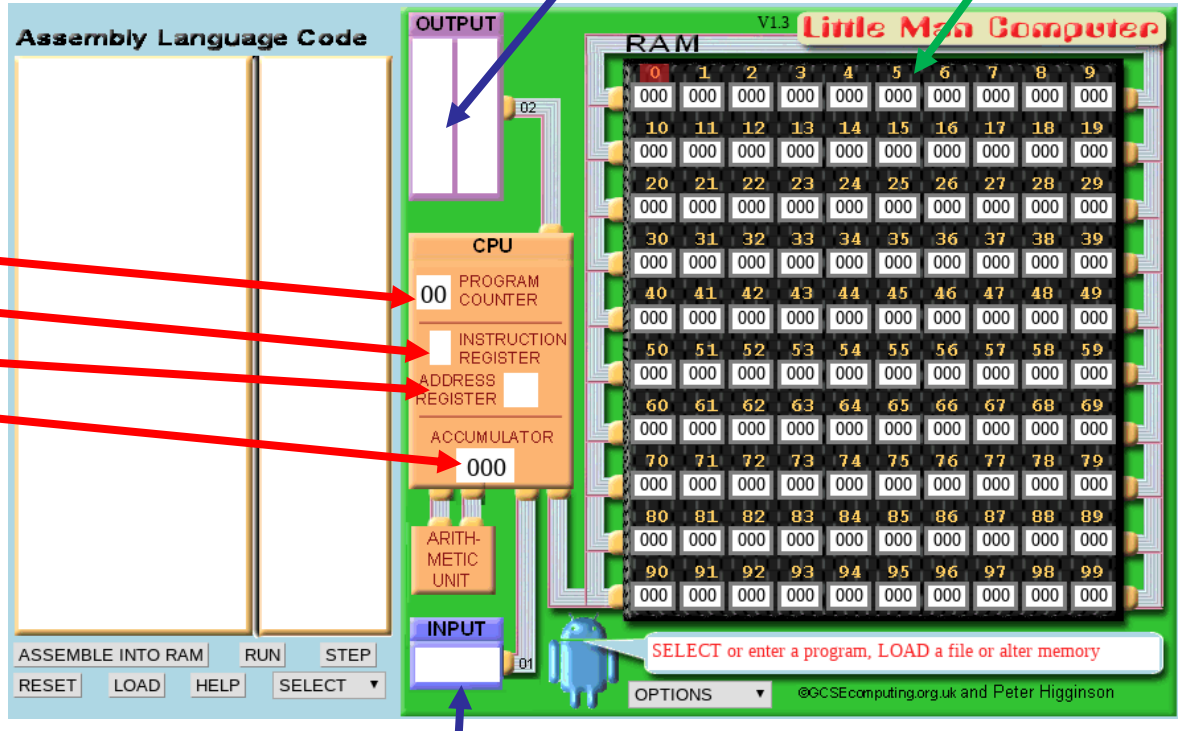
Den store erkjennelsen er at minnet holder på både *data* og *instruksjoner* til CPU-en (altså *programmet*)

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

Alle instruksjonene i LMC

Kode	Navn	Beskrivelse
0xx	HLT	Stopper eksekveringen
1xx	ADD	Adderer verdien i angitt minnelokasjon med akkumulatoren
2xx	SUB	Subtraherer verdien i angitt minnelokasjon med akkumulatoren
3xx	STO	Lagrer akkumulatoren i angitt minnelokasjon
4xx	-	Ikke i bruk
5xx	LDA	Henter verdi fra minnet til akkumulatoren
6xx	BRA	Hopper til angitt adresse
7xx	BRZ	Hopper til angitt adresse hvis akkumulatoren er 0
8xx	BRP	Hopper til angitt adresse hvis akkumulatoren er ≥ 0
901	INP	Leser verdi fra input, og legger svaret i akkumulatoren
902	OUT	Skriver ut verdien i akkumulatoren
922	OTC	Skriver ut ASCII-tegn (ikke i boka)

Eksempel: å skrive ut en streng

Dette programmet skriver **Hei**. Hvert av de tre tegnene hentes inn i akkumulatoren og skrives ut som tegn:

```
507 922 508 922 509 922 000 072 101 105
```

For å kunne angi tegnene i «Hei» må vi benytte ASCII-tabellen

Kode	Navn	Beskrivelse
0xx	HLT	Stopper eksekveringen
1xx	ADD	Adderer verdien i angitt minnelokasjon med akkumulatoren
2xx	SUB	Subtraherer verdien i angitt minnelokasjon med akkumulatoren
3xx	STO	Lagrer akkumulatoren i angitt minnelokasjon
4xx	-	Ikke i bruk
5xx	LDA	Henter verdi fra minnet til akkumulatoren
6xx	BRA	Hopper til angitt adresse
7xx	BRZ	Hopper til angitt adresse hvis akkumulatoren er 0
8xx	BRP	Hopper til angitt adresse hvis akkumulatoren er ≥ 0
901	INP	Leser verdi fra input, og legger svaret i akkumulatoren
902	OUT	Skriver ut verdien i akkumulatoren
922	OTC	Skriver ut ASCII-tegn (ikke i boka)

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

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.

Numerisk kode

507 922 508 922 509 922 000 072 101 105

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

- Det er ikke så lett å huske alle tallkodene
- Vanskelig både å skrive og lese programmer
- Det er problematisk å holde oversikt hva som ligger i hvilke celler
- Det er fort gjort å skrive et cellenummer feil
- Om vi endrer programmet, må vi oftest endre veldig mange instruksjoner

Assemblerkode

Direkte oversettelse numerisk ←→ tekstlig representasjon

- Oversettes til numerisk kode før kjøring
- Mye enklere å lese og skrive for mennesker
- Kan også sette navn på celler i minnet, slik at det blir lettere å holde styr på data i minnet

Assemblerkode

Direkte oversettelse numerisk ←→ tekstlig representasjon

```
START           LDA     H
                OTC
                LDA     e
                OTC
                LDA     i
                OTC
END             HLT

H               DAT      72  // = 'H'
e               DAT     101  // = 'e'
i               DAT     105  // = 'i'
```

Spesifikasjonen **DAT** (= «data») er ingen instruksjon, men en angivelse av at det skal settes av en celle med en gitt startverdi.

La oss teste programmet i LMC →

Instruksjonen 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 akkumulator >= 0
        HLT          // Stopp

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

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.

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. Gå i løkke *a* ganger:
   – Øk resultatverdien med *b*.
4. Skriv ut resultatverdien.

```
START INP      //     int(input(""))
      STA a    // a =
      INP      //     int(input(""))
      STA b    // b =
      LDA v0   //       0
      STA Res  // Res =

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

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

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

a     DAT
b     DAT
Res   DAT

v0    DAT 0
v1    DAT 1
```

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

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

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

- 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 (litt) avanserte ting

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)