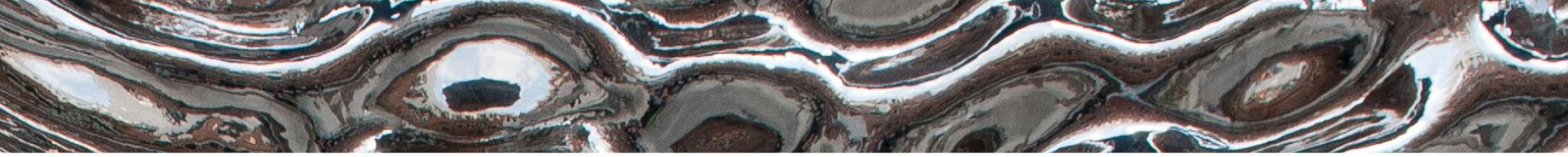


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

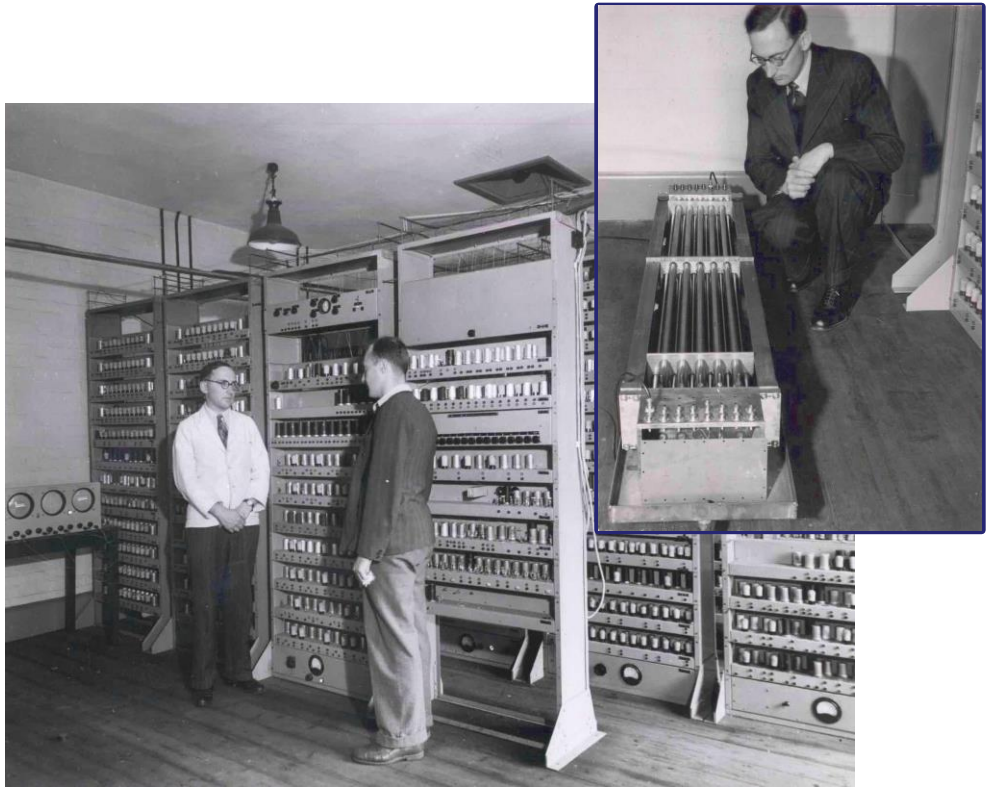
Oppsummering, Programmering

Gjennomgått materiale

- Datahistorie
  - Hva definerer en moderne maskin?
- Digital representasjon
  - Hvordan representerer vi tall, tekst, bilde og lyd med 0ere og 1ere?
- Moderne prosessorarkitektur og programmering
  - LMC og x86-64

Kriterier for Moderne Maskiner

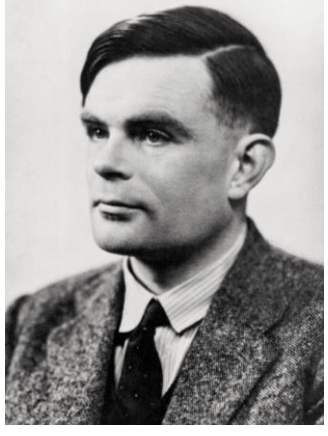
- I bruk
- Generell
- Elektronisk
- Binær
- Programmeres i RAM

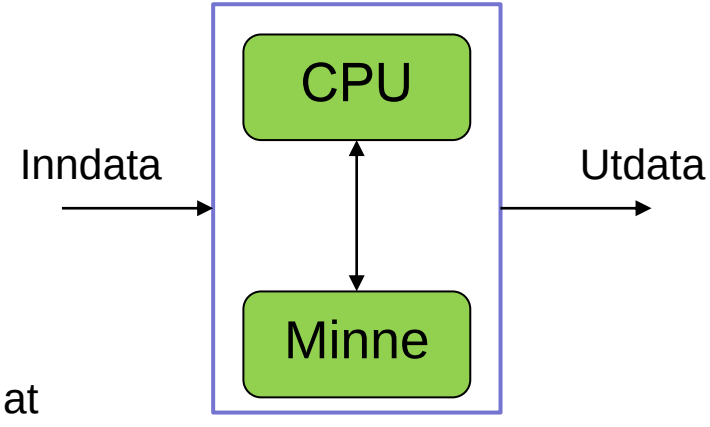


Electronic Delay Storage Automatic Calculator – EDSAC, 1949

https://en.wikipedia.org/wiki/EDSAC

Von Neuman Arkitekturen

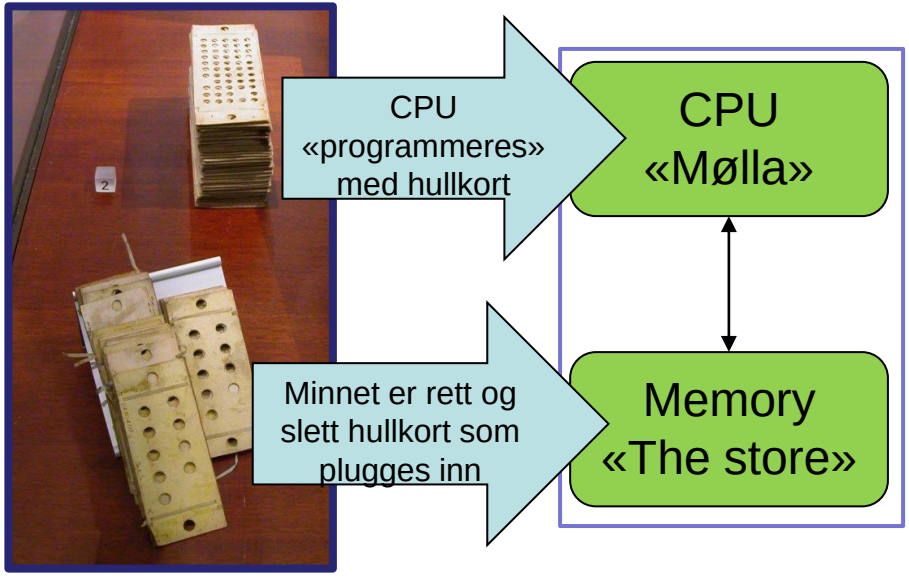




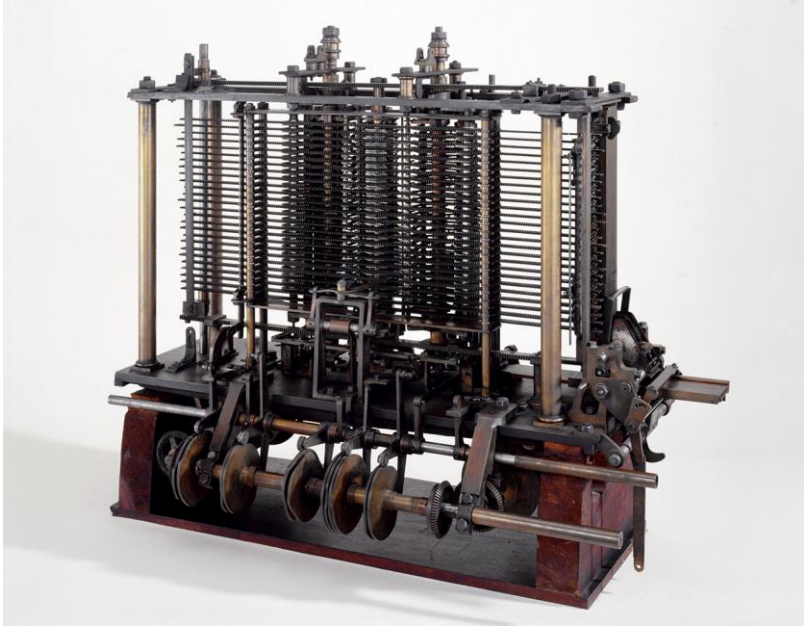
Innovasjonen er at **minnet** i maskinen inneholder **både:**

- Program
- Data

Hva betyr det at program og data ligger i minnet?

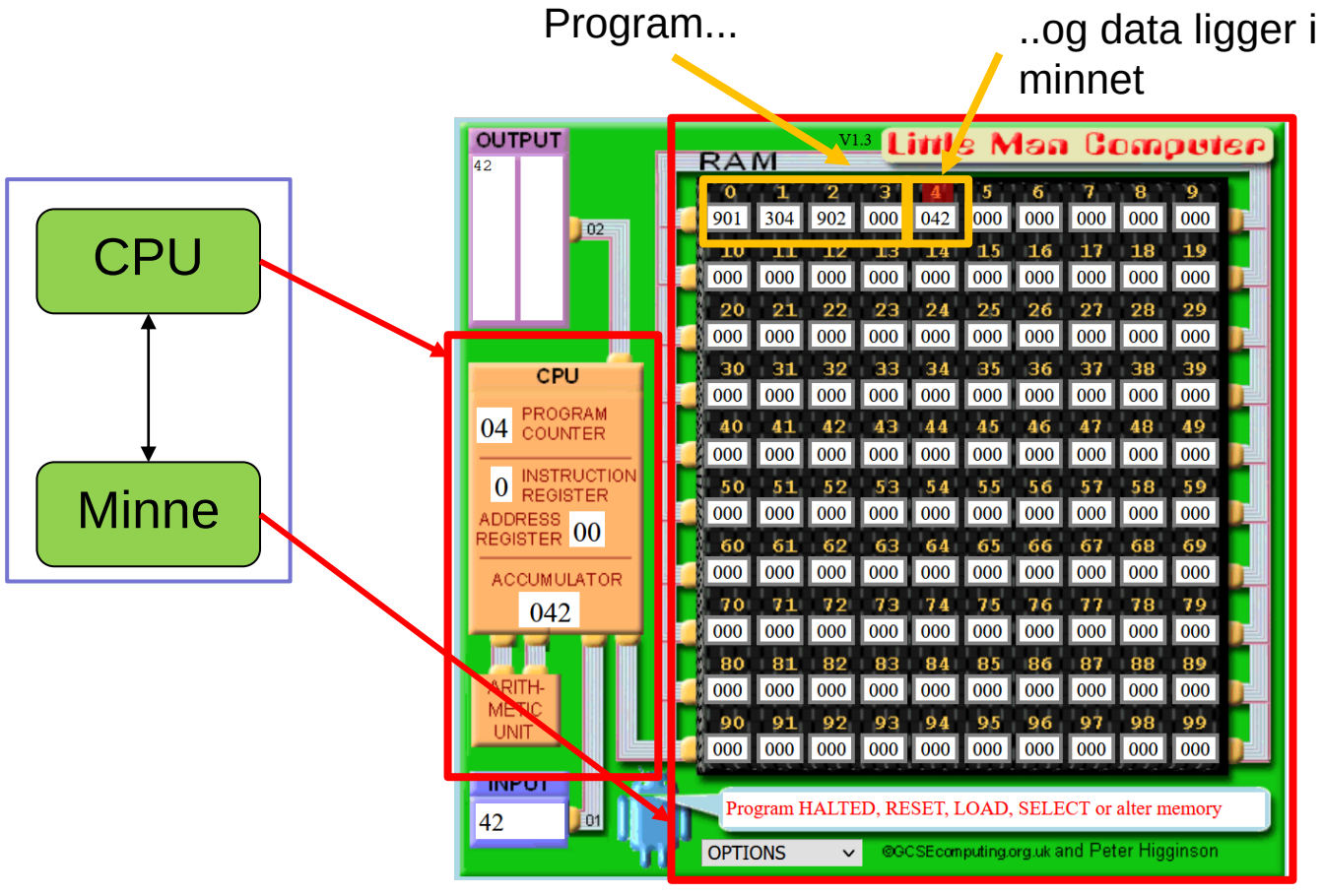


Programmet til CPU ligger **ikke** minnet – det er hullkort som man putter inn i «mølla».



Charles Babbage's «Analytiske maskin»

https://en.wikipedia.org/wiki/Analytical_Engine


Program...
..og data ligger i minnet
CPU
Minne
OUTPUT
42
02
CPU
04 PROGRAM COUNTER
0 INSTRUCTION REGISTER
ADDRESS REGISTER 00
ACCUMULATOR
042
ARITH-METIC UNIT
INPUT
42
01
V1.3
Little Man Computer
RAM
0 1 2 3 4 5 6 7 8 9
901 304 902 000 042 000 000 000 000 000
10 11 12 13 14 15 16 17 18 19
000 000 000 000 000 000 000 000 000 000
20 21 22 23 24 25 26 27 28 29
000 000 000 000 000 000 000 000 000 000
30 31 32 33 34 35 36 37 38 39
000 000 000 000 000 000 000 000 000 000
40 41 42 43 44 45 46 47 48 49
000 000 000 000 000 000 000 000 000 000
50 51 52 53 54 55 56 57 58 59
000 000 000 000 000 000 000 000 000 000
60 61 62 63 64 65 66 67 68 69
000 000 000 000 000 000 000 000 000 000
70 71 72 73 74 75 76 77 78 79
000 000 000 000 000 000 000 000 000 000
80 81 82 83 84 85 86 87 88 89
000 000 000 000 000 000 000 000 000 000
90 91 92 93 94 95 96 97 98 99
000 000 000 000 000 000 000 000 000 000
Program HALTED, RESET, LOAD, SELECT or alter memory
OPTIONS
©GCSEcomputing.org.uk and Peter Higginson

Det viktigste å få med seg om digital representasjon

- Enten det er tall...
- Maskininstruksjoner...
- Tekst...
- Bilder...
- Lyd...

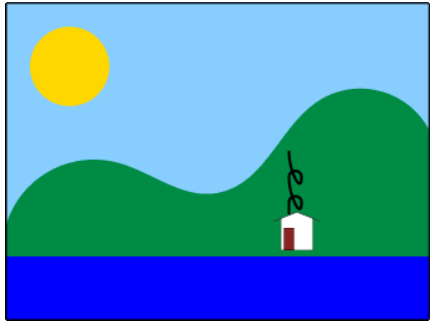
- **<u>ALT ER BIT!</u>**

256

3.141567

0xDEAD

```
00 INP
01 STA 04
02 OUT
03 HLT
04 DAT 00
```



Forskjellige tallsystemer

Desimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binær	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Tallsystemer

- Har en **base**

Tallsystem	Base
Binært	2
Oktal	8
Desimalt	10
Hexadesimalt	16

- Vi angir hvilket tallsystem ved å skrive basen som subskript
  - 1001_2 og 1001_{10}
- Med ett unntak: hexadesimal
  - 0xDEADC0FFEE

Tallsystemer er vektbaserte

- **Vekten** til et siffer er **basen** vi er i opphøyd i **posisjonen**
  - $\boldsymbol{vekt = base^{posisjon}}$
- Eksempel
  - 1001_2

1	0	0	1
$\mathbf{2^3}$	$\mathbf{2^2}$	$\mathbf{2^1}$	$\mathbf{2^0}$

Konvertering til desimal (fra hva som helst)

- Gang hvert siffer i tallsystemet du er i..
- ..med vekten til det sifferet..
- ..og legg sammen tallene.

- $1001_2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 8 + 0 + 0 + 1 = 9$
- $-1001_2 = -(1*2^3 + 0*2^2 + 0*2^1 + 1*2^0) = -(8 + 0 + 0 + 1) = -9$
- $1001_4 = 1*4^3 + 0*4^2 + 0*4^1 + 1*4^0 = 64 + 0 + 0 + 1 = 65$

Konvertering fra desimal (til hva som helst)

- Heltallsdivider tallet på basen og legg unna resten.
  - Gjenta til vi har igjen 0.
- Eksempel: 9_{10} til binært

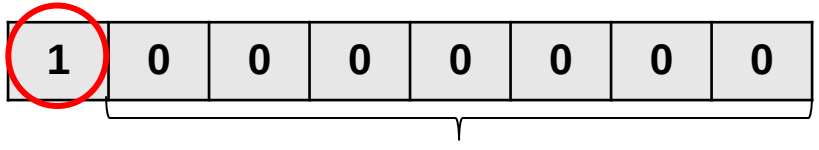
	Verdi	Rest
9/2 = 4, rest på 1.	9	1
4/2 = 2, rest på 0.	4	0
2/2 = 1, rest på 0.	2	0
1/2 = 0, rest på 1.	1	1
	0	

Svaret leser vi nedenfra: $9_{10} = 1001_2$

2'er komplement – representasjon av negative tall

- Øverste bit (fortegnsbit) tolkes som det **negative** av den (desimale) verdien det bit'et ellers ville hatt.
  - Legg så på den (positive) verdien av de resterende bitene

- Eksempel: Hvilken verdi har **byte'n** 10000000_2 i 2'er komplement?



Hvilken verdi har det øverste bit'et (fortegnsbit'et)?

Hvilken verdi har de resterende bit'ene?

Noen 2’er komplementer til på en byte

1	0	0	0	0	0	0	0

- 128 + 0 = - 128

1	0	0	0	0	0	0	1

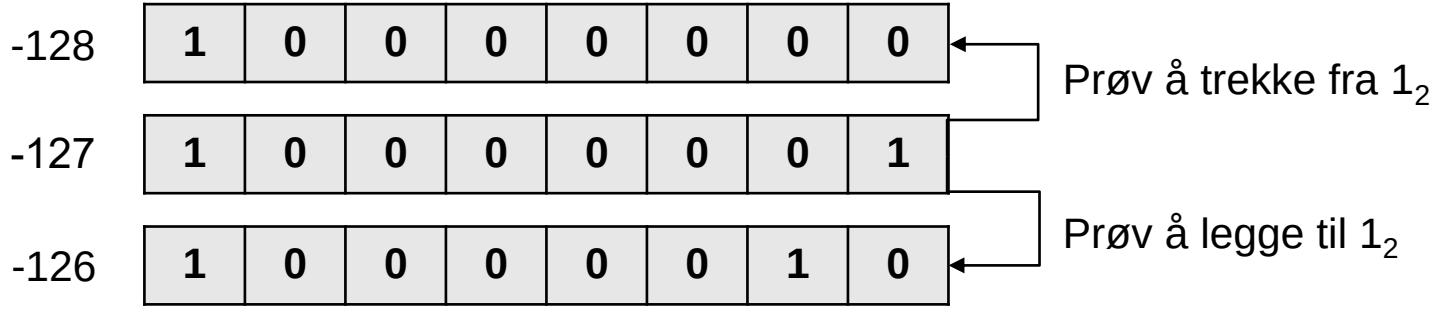
1	0	0	0	0	1	0	0

0	1	1	1	1	1	1	1

Hvordan vet maskinen at den jobber med fortegnsbit?

- Svar: Det vet den ikke!

- Det fine her er at maskinen kan legge til og trekke fra et tall på 2'er komplement, og fremdeles få et fornuftig svar.

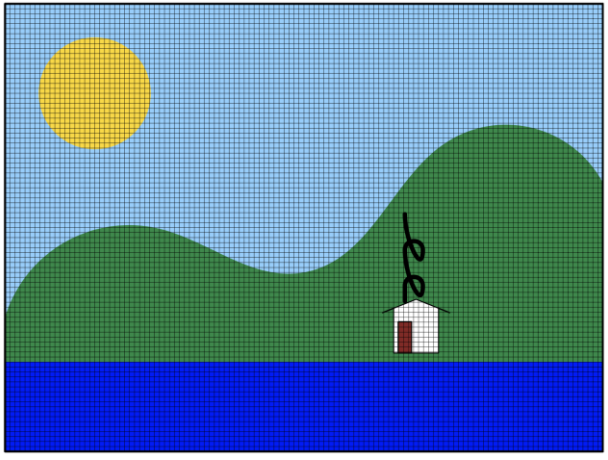


Representasjon av tekst (ASCII)

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	{		

Representasjon av bilder

- Rasterbilder og vektorbilder
- Formater for forskjellige formål
  - GIF, PNG, JPEG
- Komprimering
  - Palett-tabell, variabel lengde koding, det menneskelige øyet

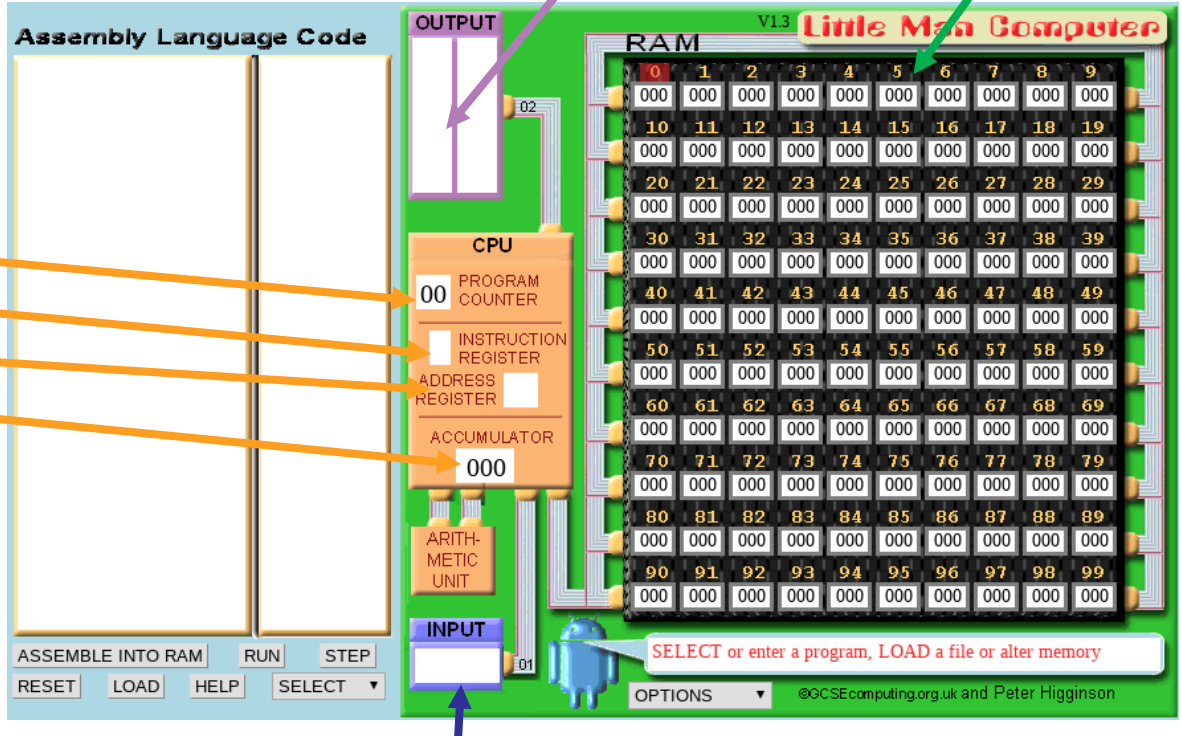


LMC-overblikk

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

LMC vs. x86-64

Hva	LMC	x86-64
Minne	100 celler, -999 til +999	Milliarder av celler, hver celle en binær byte
Instruksjoner	RISC, < 10 instruksjoner	CISC, tusenvis av instruksjoner
Ytelse	Noen titalls instruksjoner per sekund, avhengig av maskin	10^9 instruksjoner per sekund
Eksekveringsløkke	Sekvensiell	Parallell («pipelining»)
CPU	4 registre hvorav én akkumulator, -999 til +999	16 registre med opp til 8 byte per register
IO	Input felt og tekst-output felt	Mus, tastatur, skjerm, høytaler, nettverk, printer...

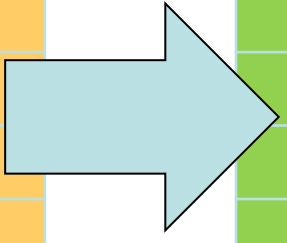
I tillegg har x86-64 ting som virtuelt minne og cache mellom CPU og minnet (og mer)..

Alle instruksjonene i LMC

Maskinkode	Assemblernotasjon	
0xx	HLT	Stopp!
1xx	ADD	Addisjon og subtraksjon
2xx	SUB	
3xx	STO	Lagre til minnet Hente fra minnet
4xx	-	
5xx	LDA	
6xx	BRA	Hopping
7xx	BRZ	
8xx	BRP	
901	INP	Input/Output (IO)
902	OUT	
922	OTC	

Maskinkode vs. assembler

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

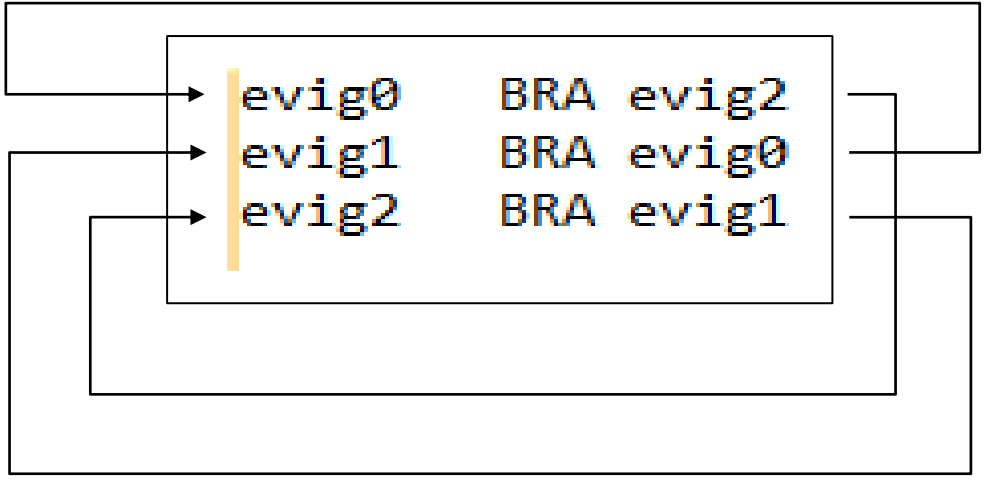


Etikett	Instruksjon	Adressedel
	INP	
	STA	tall
	INP	
	ADD	tall
	OUT	
	HLT	
tall	DAT	0

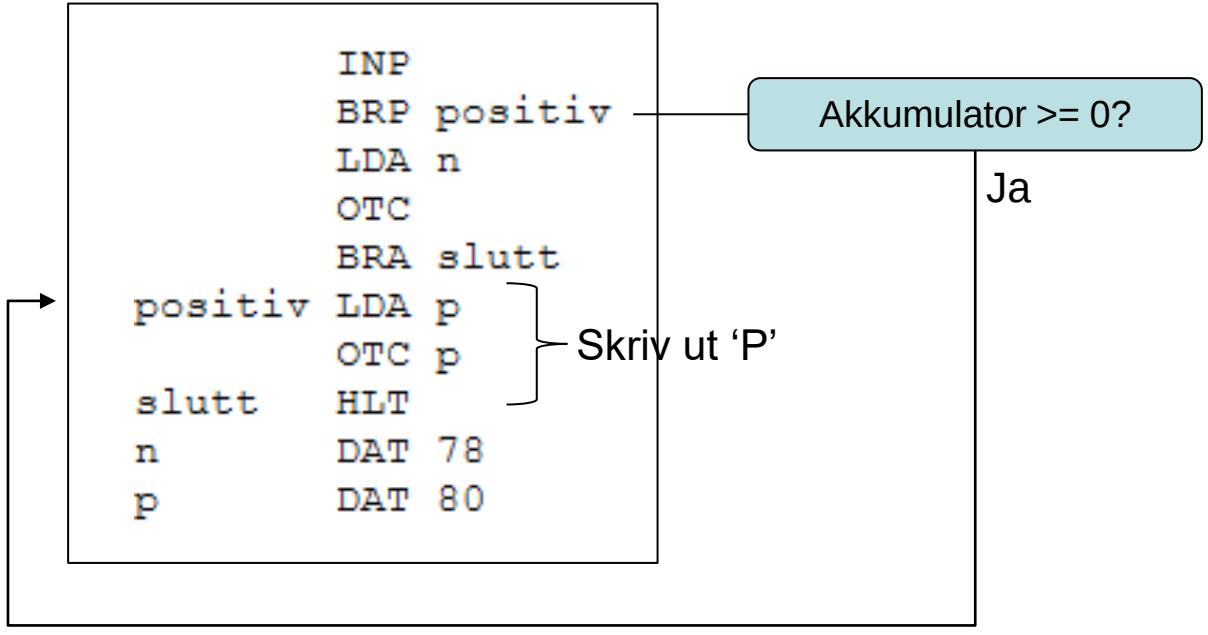
Hopp og Løkker

Type	Beskrivelse	Betingelse
BRA etsted0	Hopp til instruksjonen på etikett etsted0	Ingen betingelse
BRP etsted1	Hopp til instruksjonen på etikett etsted1	Akkumulator er positiv (>=0)
BRZ etsted2	Hopp til instruksjonen på etikett etsted1	Akkumulator er null

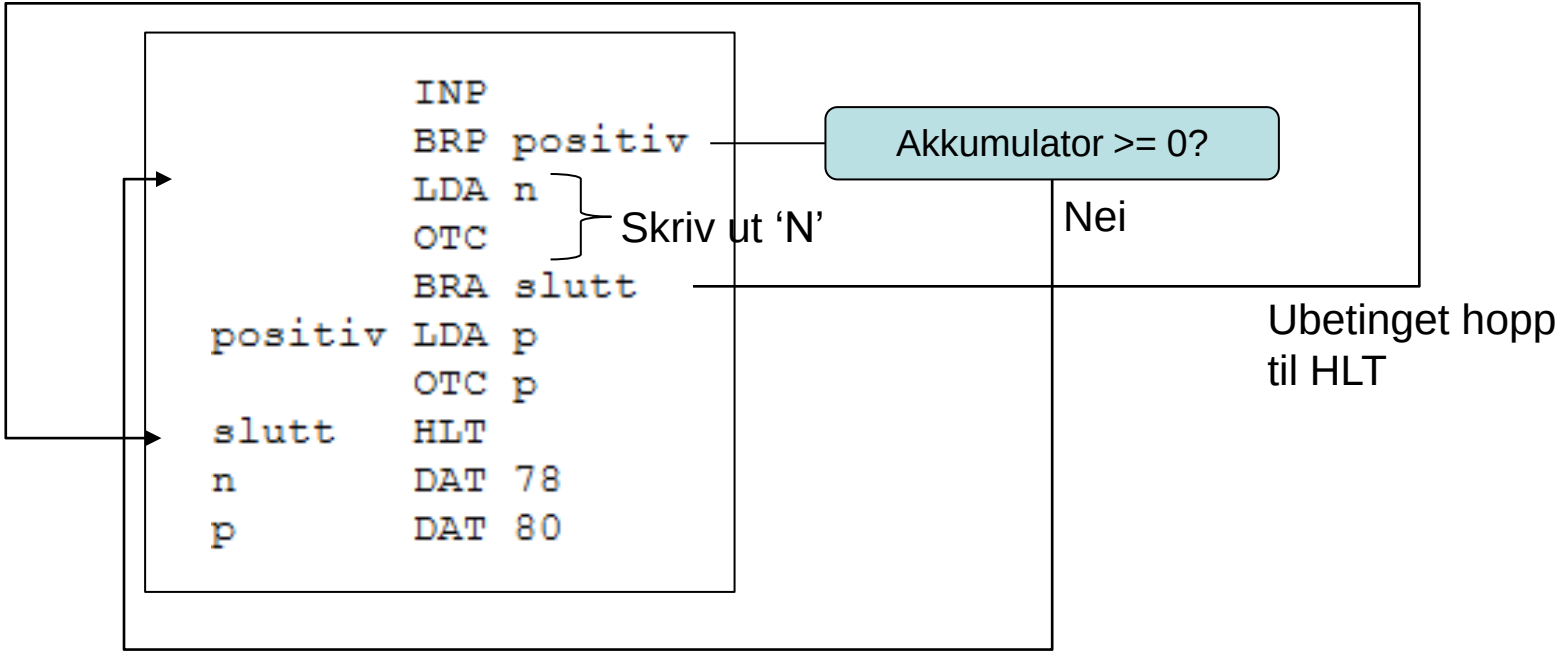
Hopp til evig tid med BRA



Hopp dersom akkumulator er positiv

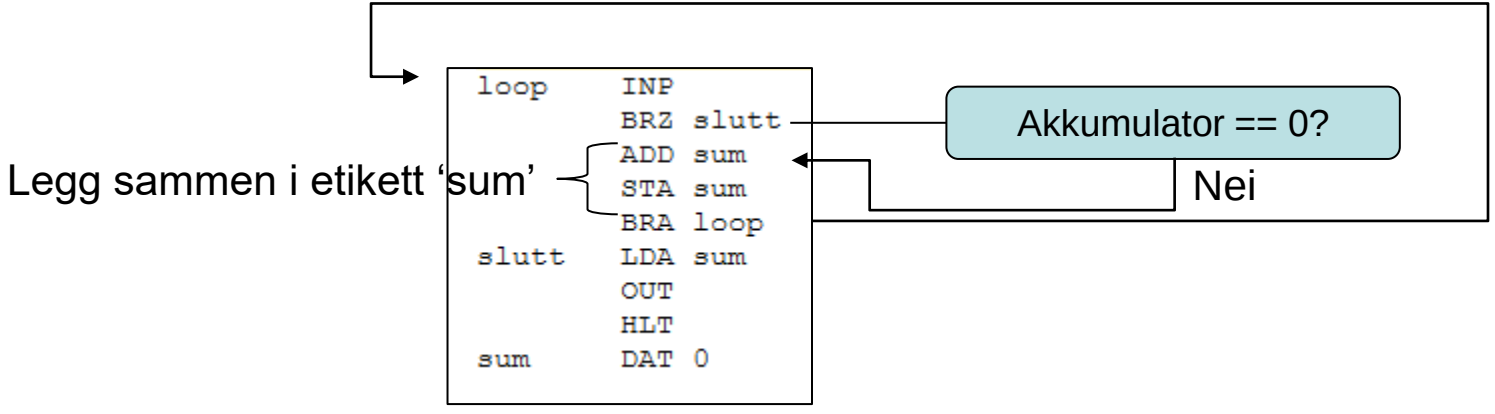


Hopp dersom akkumulator er positiv

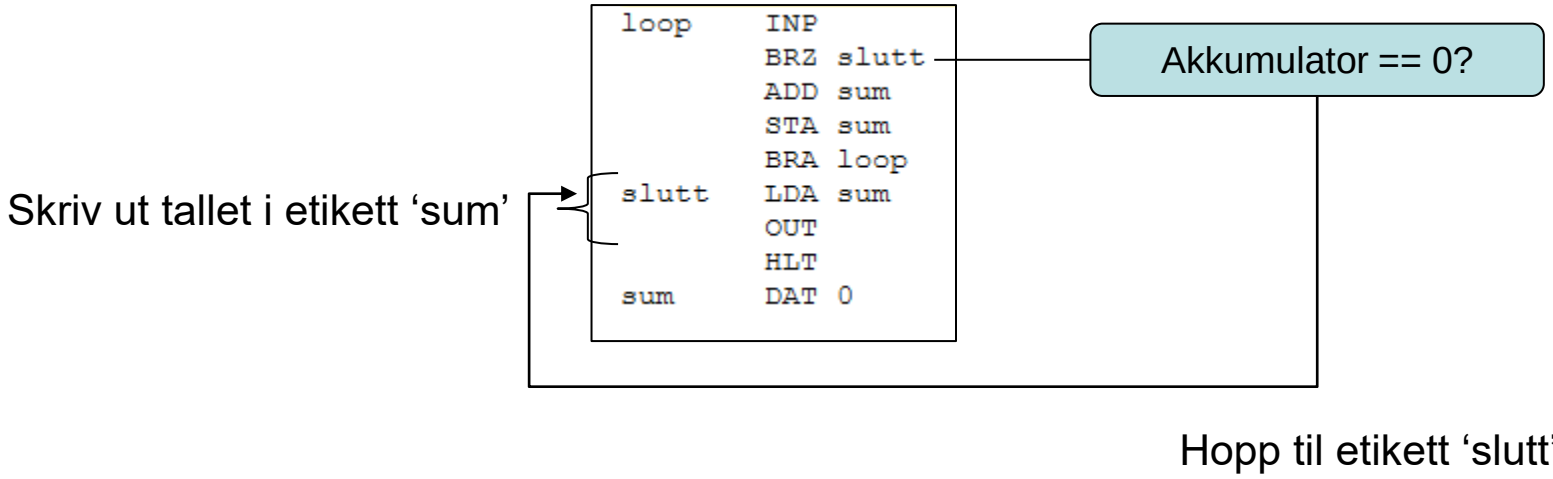


Les inn tall og summér til bruker gir null med BRZ

Ubetinget hopp til etikett ‘loop’



Les inn tall og summér til bruker gir null med BRZ



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 med løkke i python

```
a = 1
b = 2
res = 0

while a > 0:
        res = res + b
        a   = a   - 1

print(res)
```

Gå inn i while-løkken

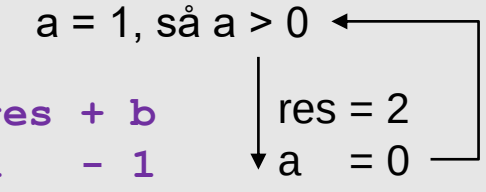
a = 1, så a > 0

res = 0 + 2

a = 1 - 1

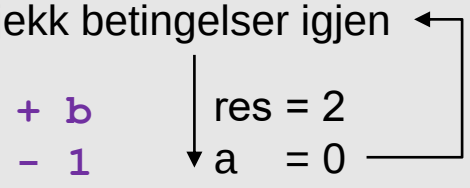
Multiplikasjon med løkke i python

```
a = 1
b = 2
res = 0

while a > 0:
        res = res + b
        a   = a   - 1

print(res)
```



Multiplikasjon med løkke i python

```
a = 1
b = 2
res = 0

while a > 0:
        res = res + b
        a   = a   - 1

print(res)
```



Multiplikasjon med løkke i python

```
a = 1
b = 2
res = 0

while a > 0:
        res = res + b
        a   = a   - 1

print(res)
```

a = 0, så a er ikke > 0

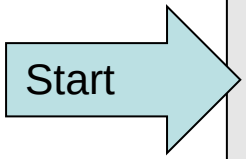
Løkka «brytes» og vi fortsetter med å skrive ut resultatet.

a * b = 1 * 2 = 2,
eller «2 plusset på seg selv 1 gang(er)»

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.



Nei; a = 1; fortsett

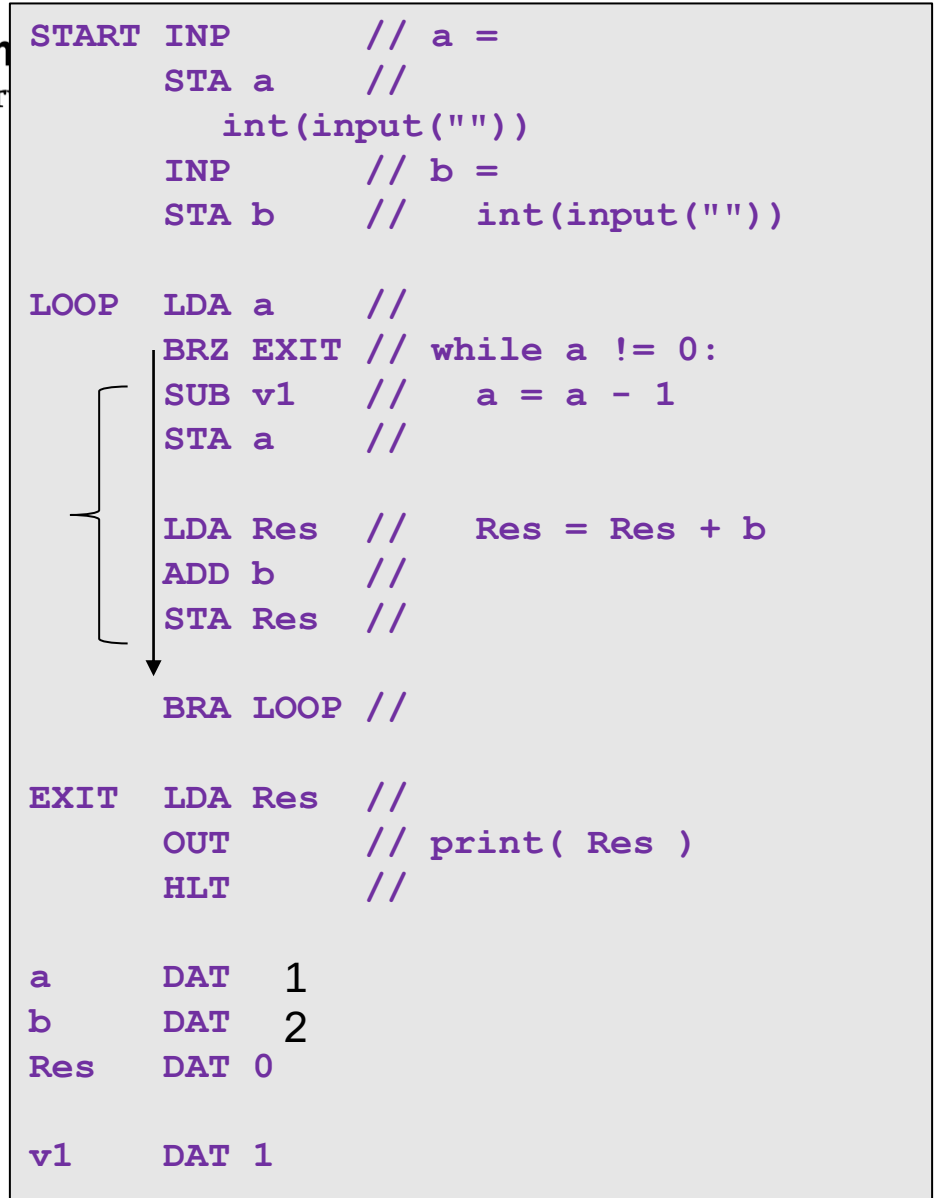
a = a – 1
res = res + b

```
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  1
b     DAT  2
Res   DAT 0

v1    DAT 1
```

Er **a** lik 0 (null)?

Nei; a = 1; fortsett

a = 1 – 1
res = 0 + 2

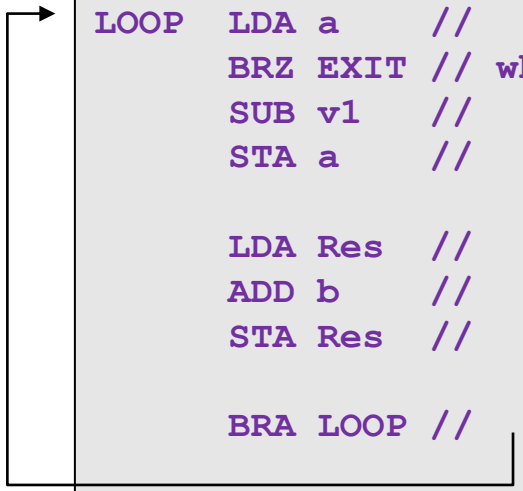


Nei; a = 1; fortsett

a = 1 – 1
res = 0 + 2

```
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  0
b     DAT  2
Res   DAT  2

v1    DAT 1
```

Ubetinget hopp tilbake til etikett 'loop'



```
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  0
b     DAT  2
Res   DAT  2

v1    DAT 1
```

```
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  0
b     DAT  2
Res   DAT  2

v1    DAT 1
```

Er **a** lik 0 (null)?

Ja; a = 0!

Hopp til etikett ‘exit’

Skriv ut svaret
a*b = 1*2 = 2

- Hvis a er større enn 1, vil programmet gjenta «BRZ løkken» og fortsette å legge **b** til **res**, **a** ganger.

Maskinkode vs. assembler

- 👍 Man slipper å huske numeriske koder.
- 👍 Adresser håndteres stort sett automatisk med etiketter.

- 👎 Stadig ingen feilsjekking og ingen begrensninger på hva man får lov til.
- 👎 Vanskelig å feilsøke

Veien mot høynivå språk

- 3.generasjonsspråk (FORTRAN, COBOL)
  - Fikk med seg mye fra assembler verdenen (feks GOTO) og var ikke helt trygge å programmere i
  - ..men introduserte også en del konsepter som arrayer
  - Språk som C, C++, Simula ble til – særspråk for systemprogrammering

- 4.generasjonsspråk
  - Interpreterte språk som python
  - Trygge, men også trege

Lykke til! Spørsmål?