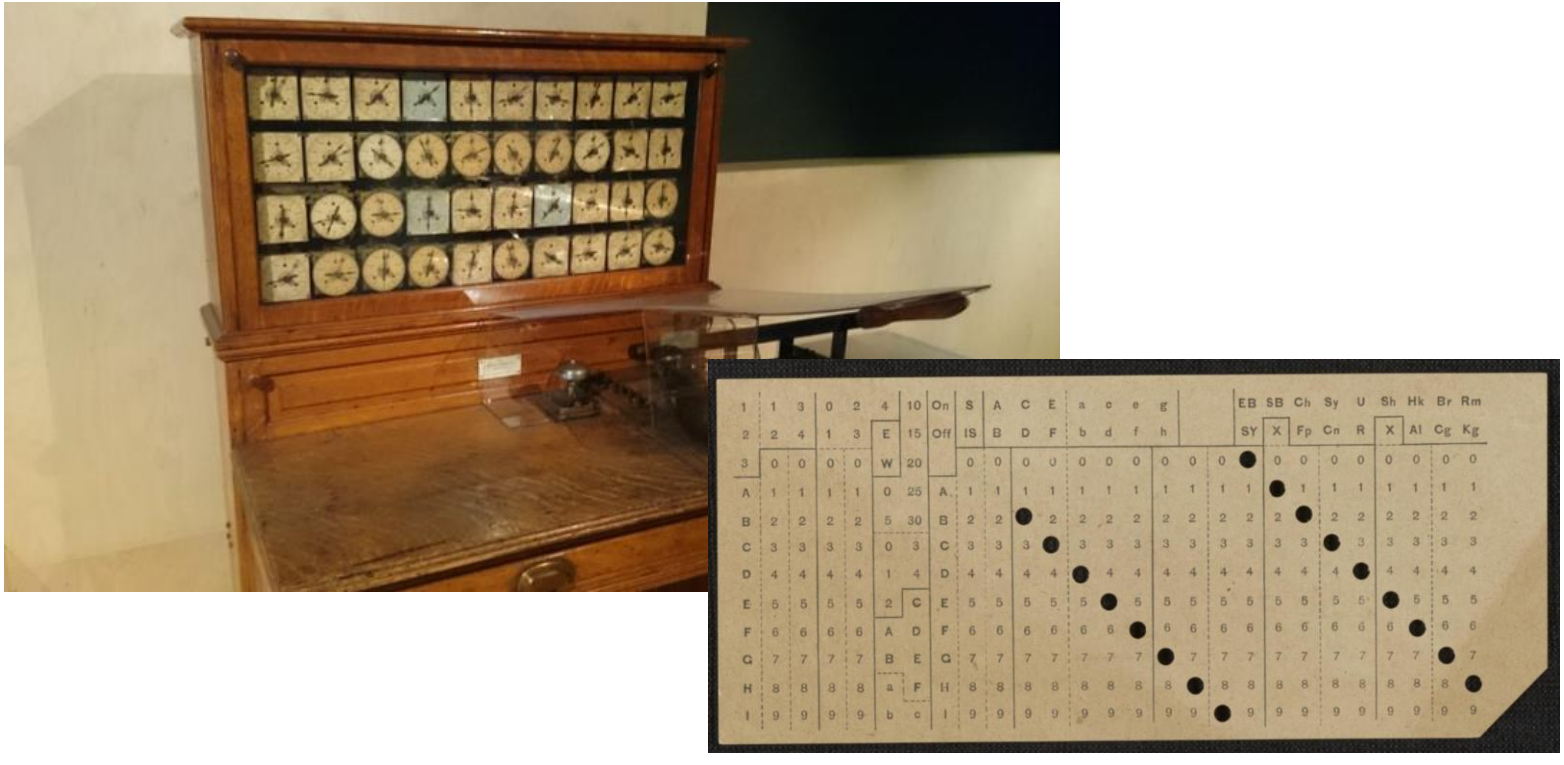


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

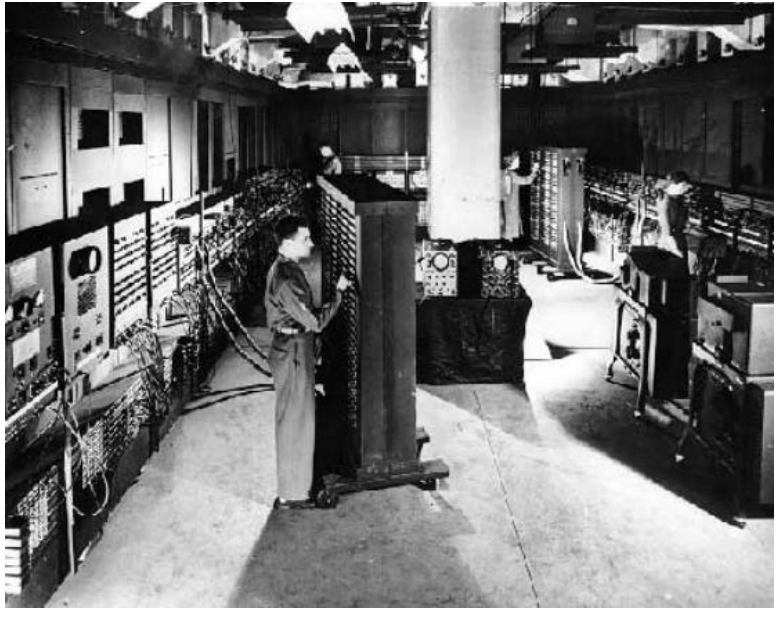
Avanserte Prosessorer og Moderne Programmering



Den første binære maskinen?



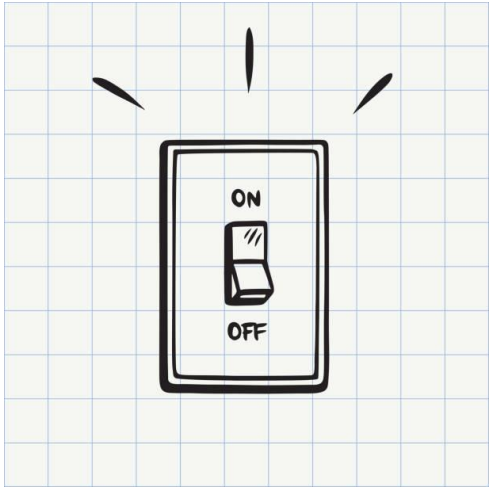
Eller denne?





Et radiorør er som en bryter.





To tilstander, på eller av, 1 eller 0, altså binært.

1. generasjon: Radiorør

År	Teknologi	Størrelse	Instr/sek	Pris (2019kr)
1945-60	Radiorør	$10m^3$	2000	60 mill





Maskinene var gigantiske og radiorørene måtte byttes ofte.

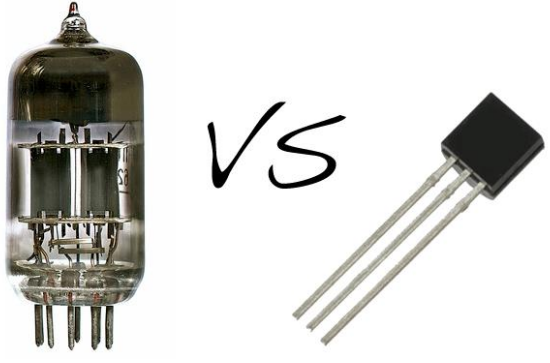

東京ラジオデパート
TOKYO RADIO DEPARTMENT STORE



2. generasjon: Transistorer

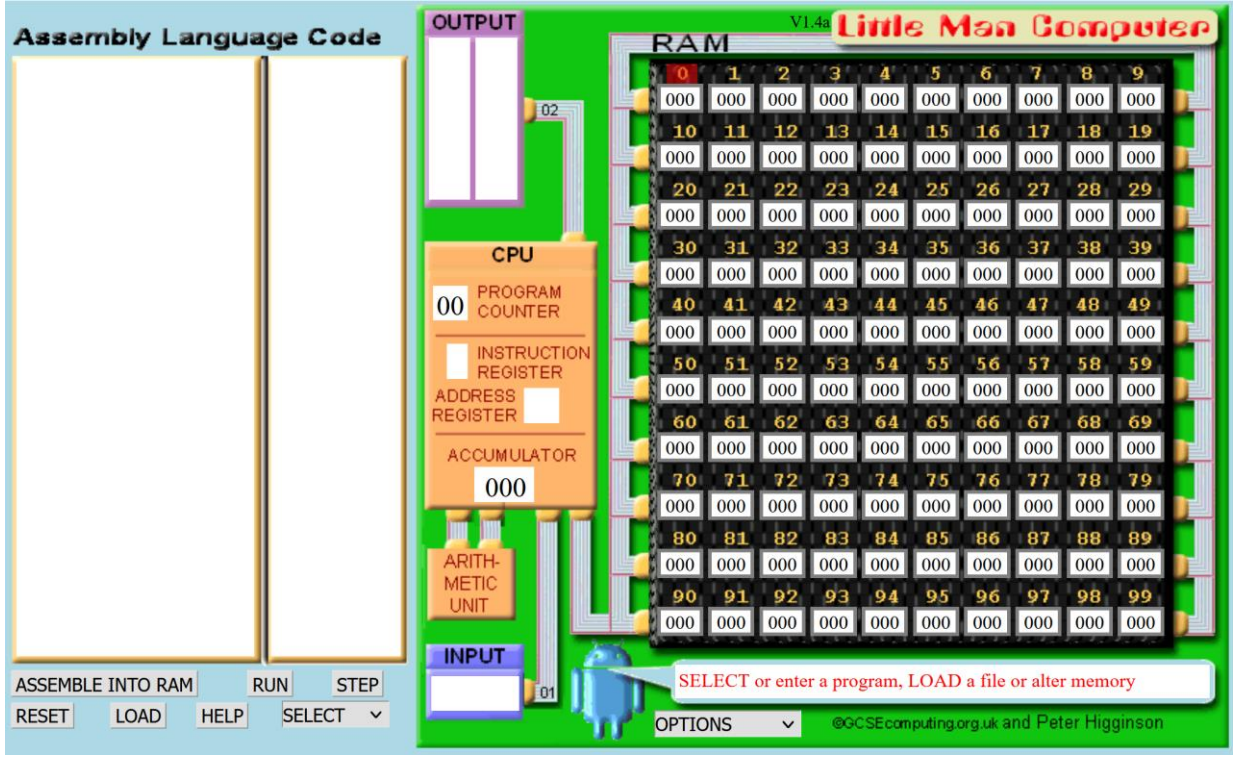
År	Teknologi	Størrelse	Instr/sek	Pris (2019kr)
1945-60	Radiorør	$10m^3$	2000	60 mill
1960-68	Transistorer	$500dm^3$	500 000	50 mill





Maskinene ble mindre, raskere og mye mer pålitelige

Little Man Computer blir født (1965).

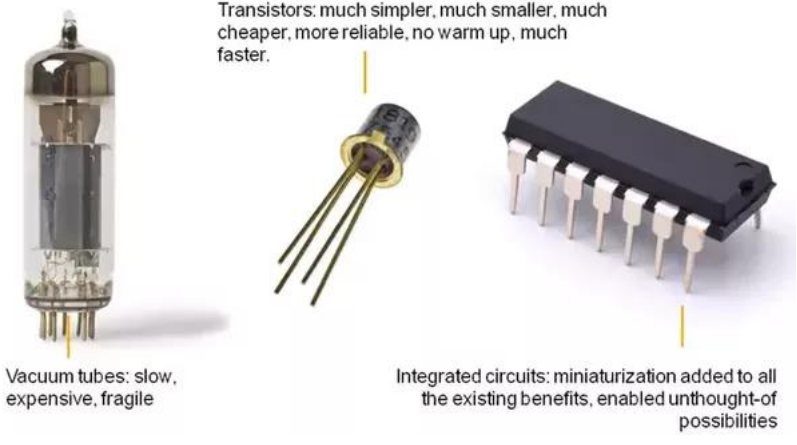


(Dette er ikke den originale)

3. generasjon: Integrerte kretser

År	Teknologi	Størrelse	Instr/sek	Pris (2019kr)
1945-60	Radiorør	$10m^3$	2000	60 mill
1960-68	Transistorer	$500dm^3$	500 000	50 mill
1970-80	Integrerte kretser	$80dm^3$	300 000	200 000





Maskinene ble mindre, og mye billigere

The Voyager Probes (1977)

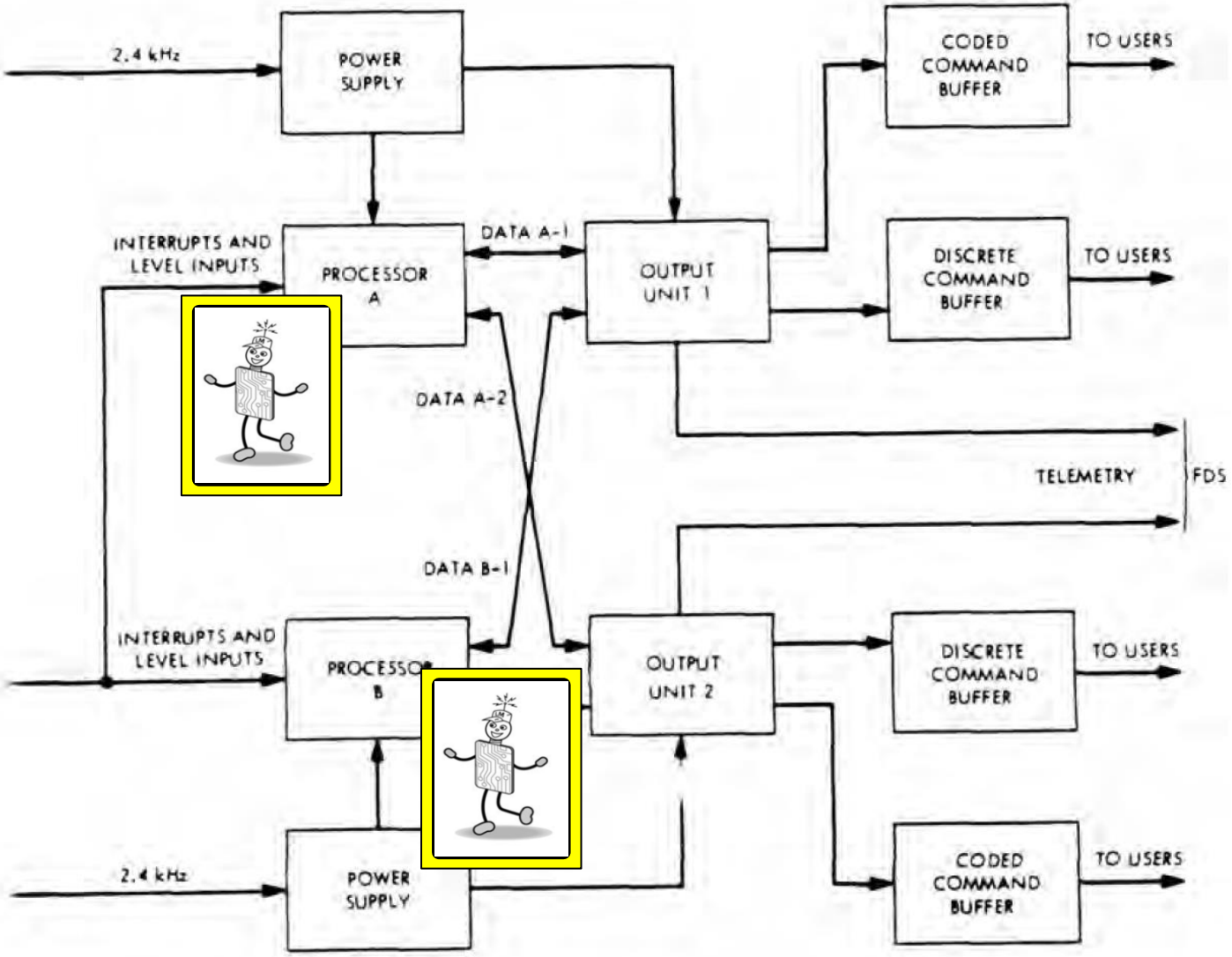


Courtesy of NASA
https://science.nasa.gov/gallery/images-of-voyager/

From Voyager CPU Reference Manual

The Command Computer's central processor contained the registers, data path control and instruction interpreter[76]. The machine was serial in operation, thus reducing complexity, weight, and power requirements. It had 18-bit words and used the least significant 6 bits for operation codes and the most significant 12 for addresses, as numbered from right to left. This permitted 64 instructions and 4K of direct addressing, both of which were fully utilized. Data were stored in signed two's complement form, yielding an integer range from -131,072 to +131,071. Average instruction cycle time came to 88 microseconds. Thirteen registers were in the Command Computer, mostly obvious types such as an 18-bit accumulator, 12-bit program counter, 12-bit link register that pointed to the next address to be read, and a 4-bit condition code register that stored the overflow, minus, odd parity, and nonzero flags[77].

«Computers in Spaceflight: The NASA Experience»
https://ntrs.nasa.gov/citations/19880069935


2.4 kHz
POWER SUPPLY
CODED COMMAND BUFFER
TO USERS
INTERRUPTS AND LEVEL INPUTS
PROCESSOR A
DATA A-1
OUTPUT UNIT 1
DISCRETE COMMAND BUFFER
TO USERS
DATA A-2
TELEMETRY
FDS
DATA B-1
INTERRUPTS AND LEVEL INPUTS
PROCESSOR B
OUTPUT UNIT 2
DISCRETE COMMAND BUFFER
TO USERS
2.4 kHz
POWER SUPPLY
CODED COMMAND BUFFER
TO USERS

Pale Blue Dot

14. Feb 1990



4. generasjon: Mikroprosessorer

År	Teknologi	Størrelse	Instr/sek	Pris (2019kr)
1945-60	Radiorør	$10m^3$	2000	60 mill
1960-68	Transistorer	$500dm^3$	500 000	50 mill
1970-80	Integrerte kretser	$80dm^3$	300 000	200 000
1980-??	Mikroprosessorer	$0,1\text{-}20dm^3$	10^9	1-10 000



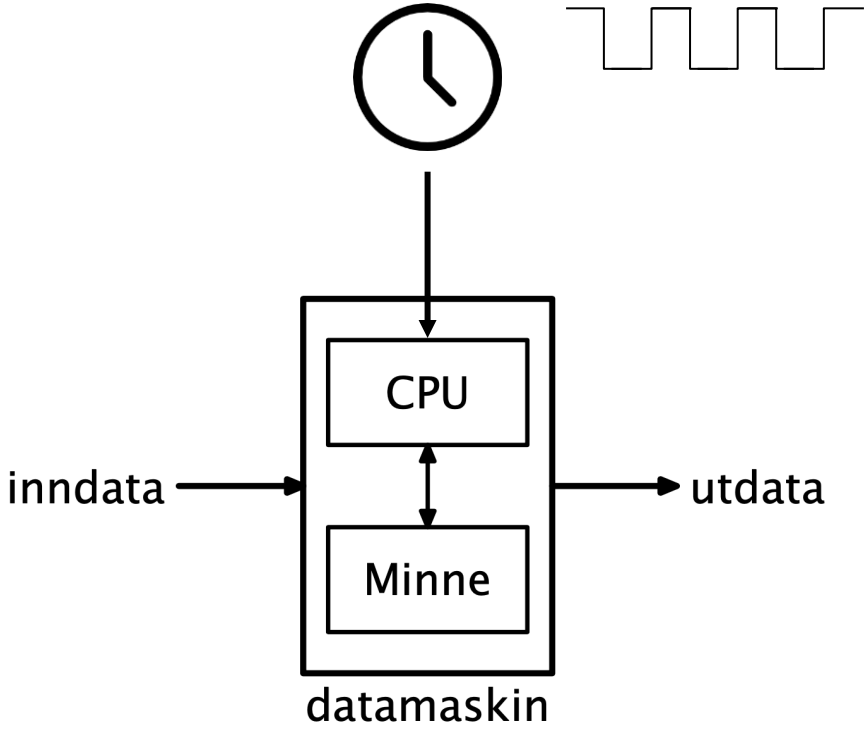


Klokkefrekvens

Sier noe om hvor fort en CPU arbeider.

Desto høyere klokkefrekvens, jo raskere CPU.

Eks. 10 Hz :
Klokka slår 10 ganger i sekundet.



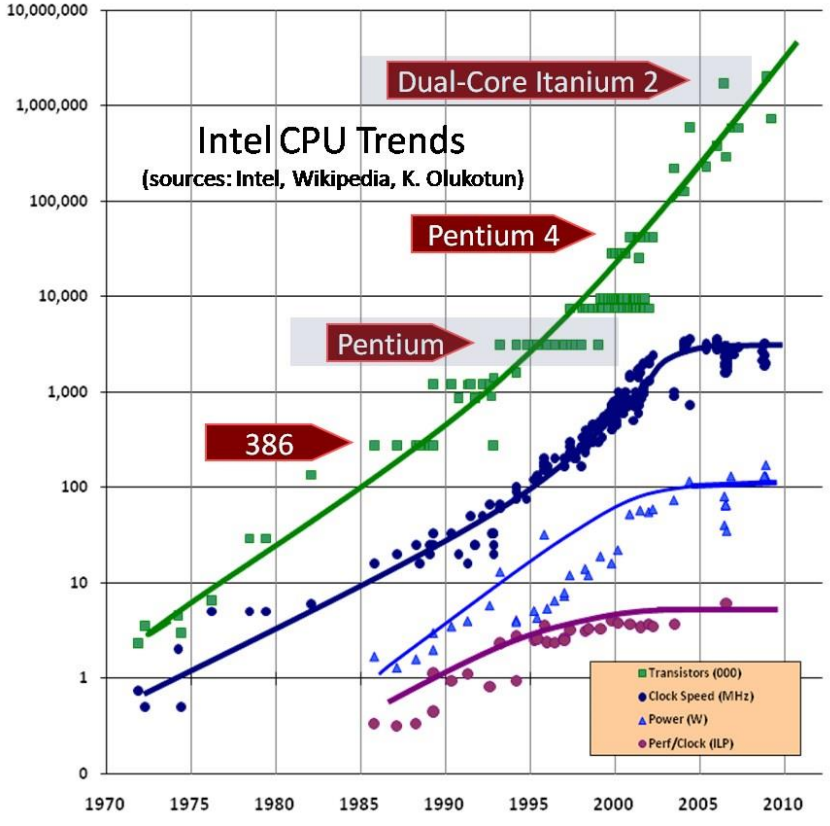
Prosessoren x86

Intel's **x86** har gått fra å være én av mange prosessorer til å bli den dominerende

Noen årsaker:

- Tidlig ute med nyvinninger
- Bakoverkompatibel

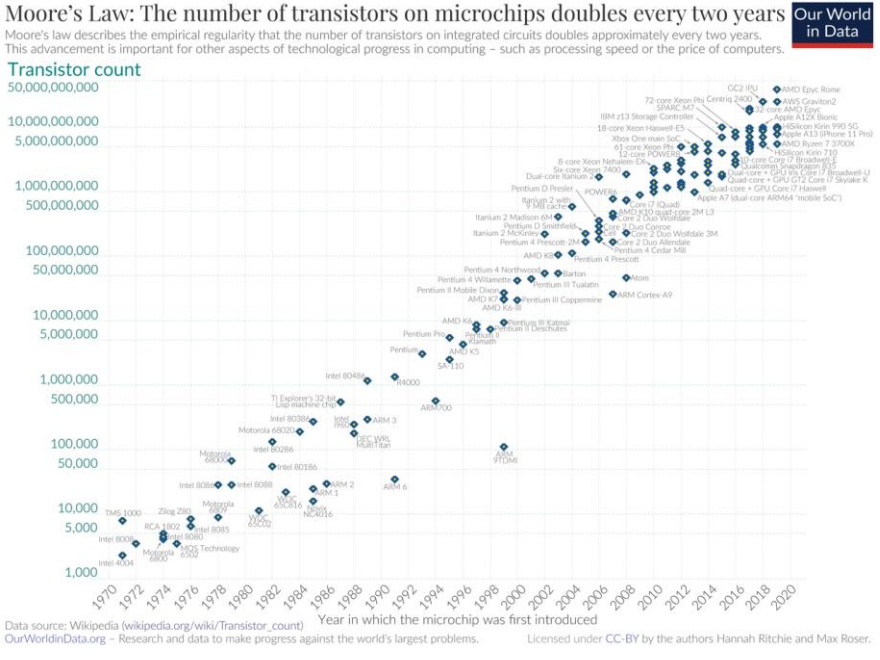
Utfordres av ARM.



Moore's «lov»

Antall transistorer i en krets dobler seg annethvert år.

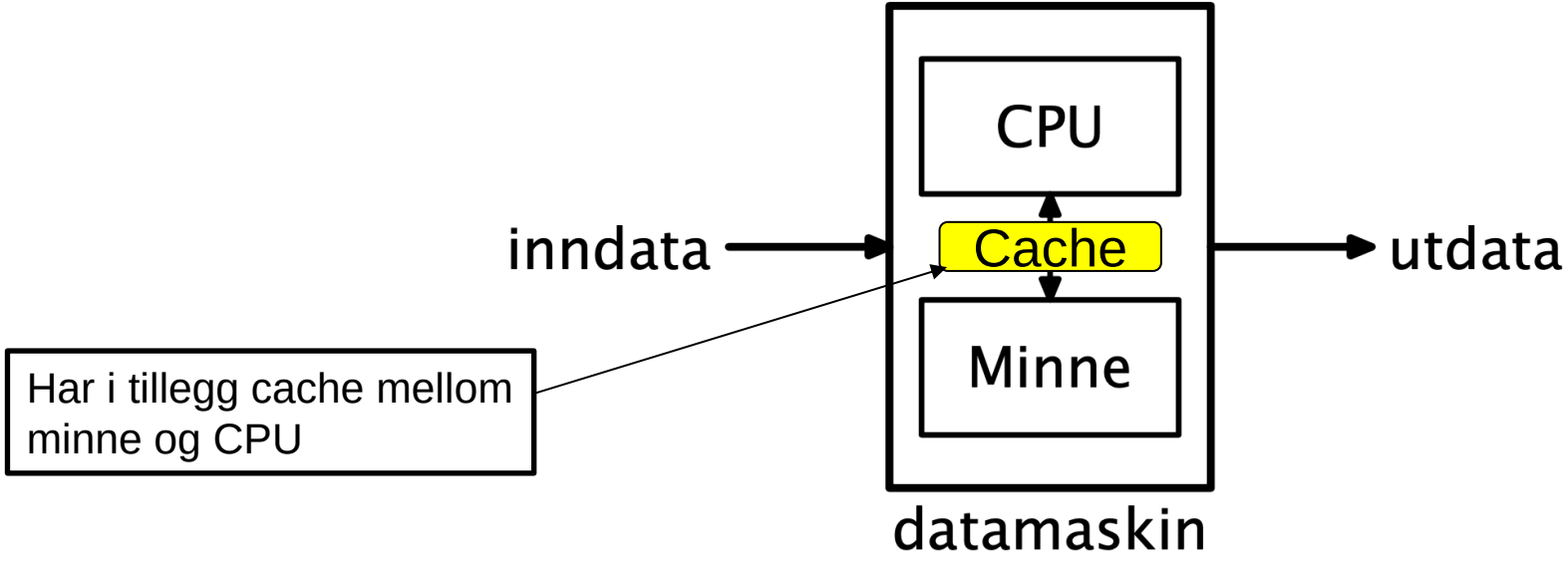
Ble observert av Gordon Moore ved Intel allerede i 1965.



Kilde: https://en.wikipedia.org/wiki/Moore's_law

Prosessoren x86-64

x86-64 fra Intel og AMD finnes i de aller fleste «vanlige» datamaskiner i dag.



Cache

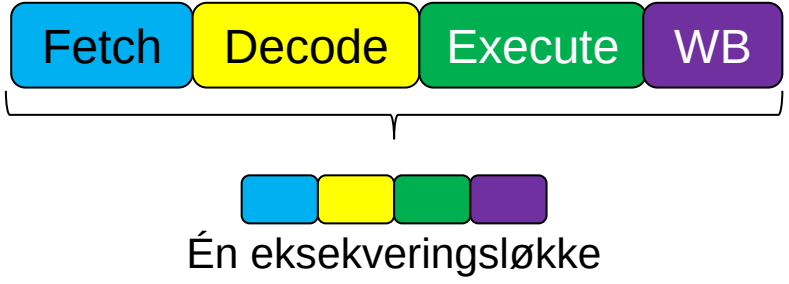
- Gjør minneoppslag i RAM raskere.
- Er relativt liten.
  - I moderne prosessorer, noen MB.
  - Vs. Flere titalls GB i RAM.
- Finnes både data- og instruksjons-cacher.

Eksekveringsløkken

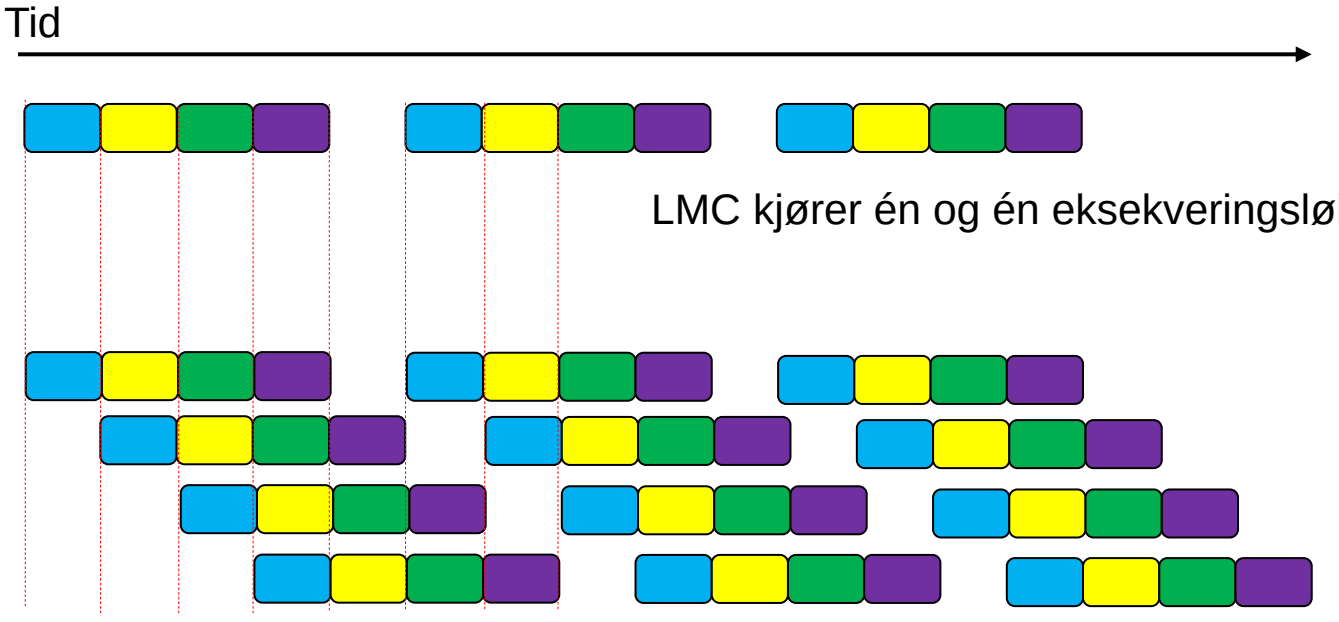
Både LMC og x86-64 har samme eksekveringsløkke:

1. Hent instruksjonen i minnet; programtelleren angir hvor.
2. Dekod instruksjonen i feks. operasjon og adresse.
3. Utfør instruksjonen.
4. Om aktuelt, skriv svaret i minnet/register.

x86-64 har i tillegg pipeline for å kunne jobbe raskere.



Pipelining



x86-64 kjører flere eksekveringsløkker i parallell «pipelining»

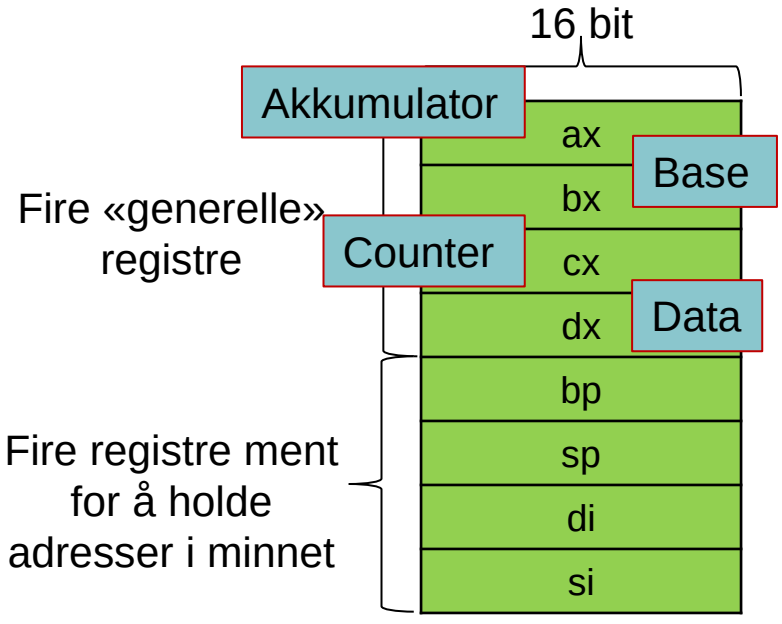
Registre

LMC har ett register som programmereren kan bruke (akkumulator-registeret)

Det kan lagre verdier –999 til +999.

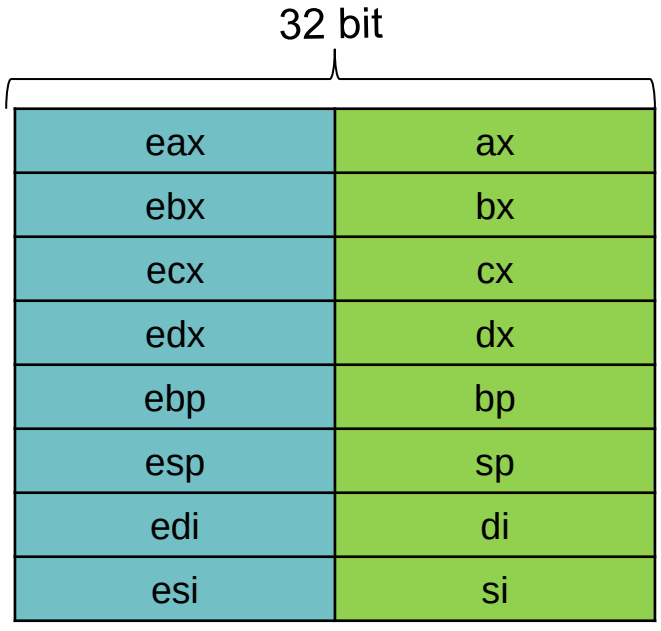
x86-64 har 16 registre. De kan brukes som 1-, 2-, 4- og 8-bytes registre.

Intel 16-bit


16 bit
Akkumulator
ax
Base
bx
Fire «generelle» registre
Counter
cx
Data
dx
bp
Fire registre ment for å holde adresser i minnet
sp
di
si

Intel 16-bit

Intel 32-bit



Intel 16-bit

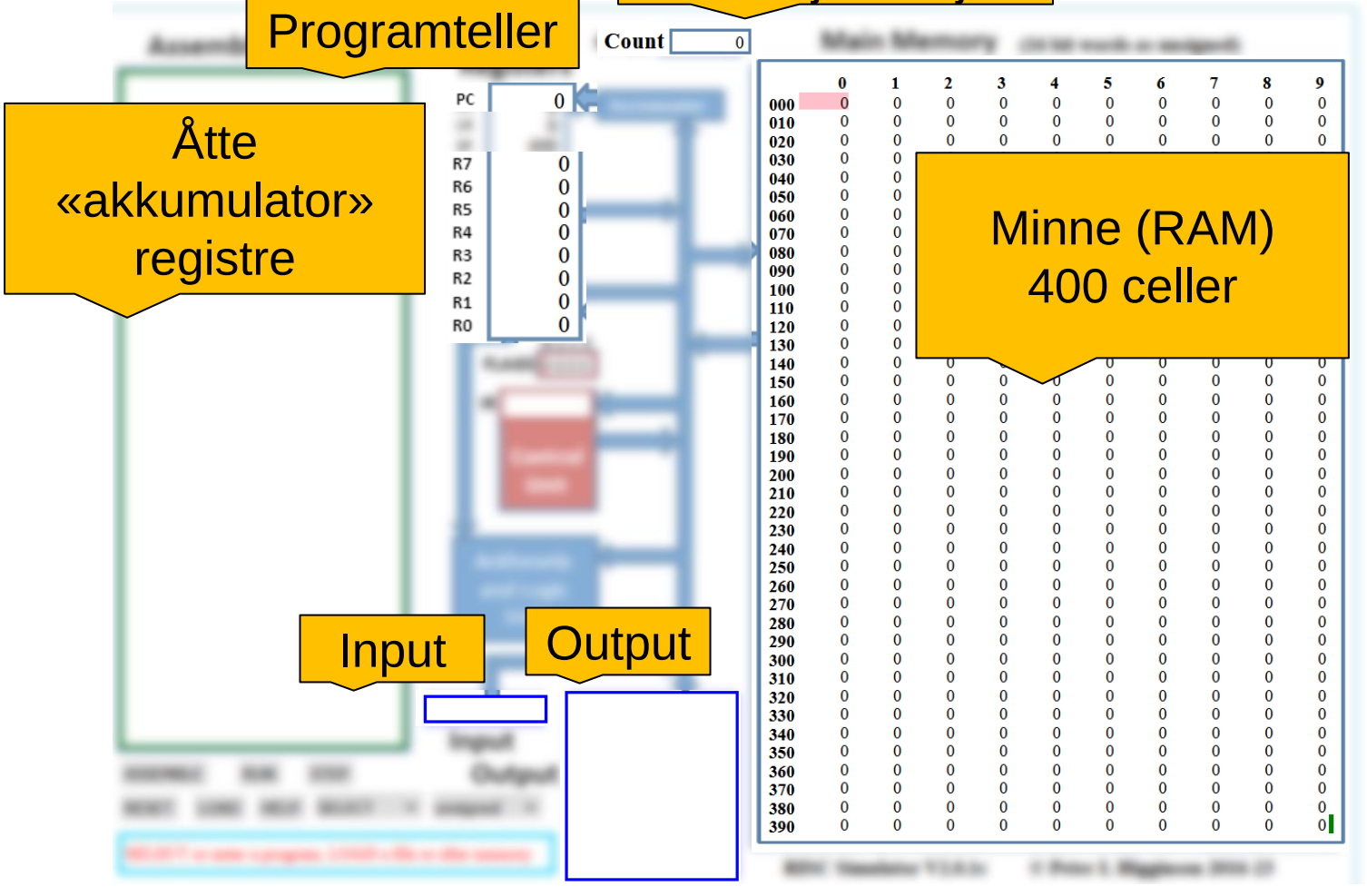
Intel 32-bit

Intel 64-bit

64 bit

rax	eax	ax
rbx	ebx	bx
rcx	ecx	cx
rdx	edx	dx
rbp	ebp	bp
rsp	esp	sp
rdi	edi	di
rsi	esi	si
r8		
r9		
r10		
r11		
r12		
r13		
r14		
r15		

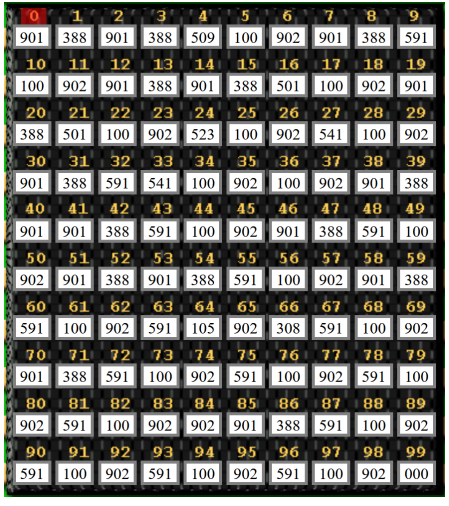
oderne LMC



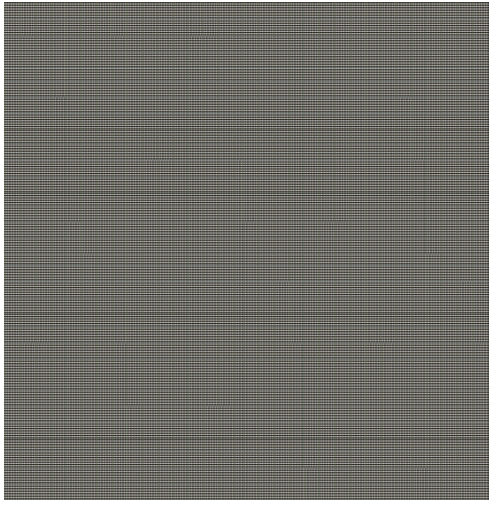
https://peterhigginson.co.uk/RISC/V2.php

Minne (RAM)

Det finnes ingen maskin med så mye minne installert (typisk 8-64 GB).



LMC: 100 celler.
Ca. 138 byte (11 bit per celle).



x86_64: 2^{64} celler.
Hver celle er 1 byte, altså:
17 000 000 000 GB

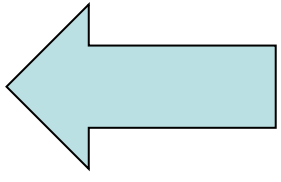
I tillegg har x86_64 noe som kalles **virtuelt minne**.

Hva hvis vi går tom for minne?

Med **virtuelt minne (x86_64)** kan vi flytte data fra RAM til en harddisk.
- Og motsatt!

Celle 80 til 99 blir i dette tilfellet gjort tilgjengelig for nye data.





0	1	2	3	4	5	6	7	8	9
901	388	901	388	509	100	902	901	388	591
10	11	12	13	14	15	16	17	18	19
100	902	901	388	901	388	501	100	902	901
20	21	22	23	24	25	26	27	28	29
388	501	100	902	523	100	902	541	100	902
30	31	32	33	34	35	36	37	38	39
901	388	591	541	100	902	100	902	901	388
40	41	42	43	44	45	46	47	48	49
901	901	388	591	100	902	901	388	591	100
50	51	52	53	54	55	56	57	58	59
902	901	388	901	388	591	100	902	901	388
60	61	62	63	64	65	66	67	68	69
591	100	902	591	105	902	308	591	100	902
70	71	72	73	74	75	76	77	78	79
901	388	591	100	902	591	100	902	591	100
80	81	82	83	84	85	86	87	88	89
902	591	100	902	902	901	388	591	100	902
90	91	92	93	94	95	96	97	98	99
591	100	902	591	100	902	591	100	902	000

Hva med sensitiv informasjon?



Passordet til Omid ligger i programmet.



Håkon sitt program leser ut passordet!

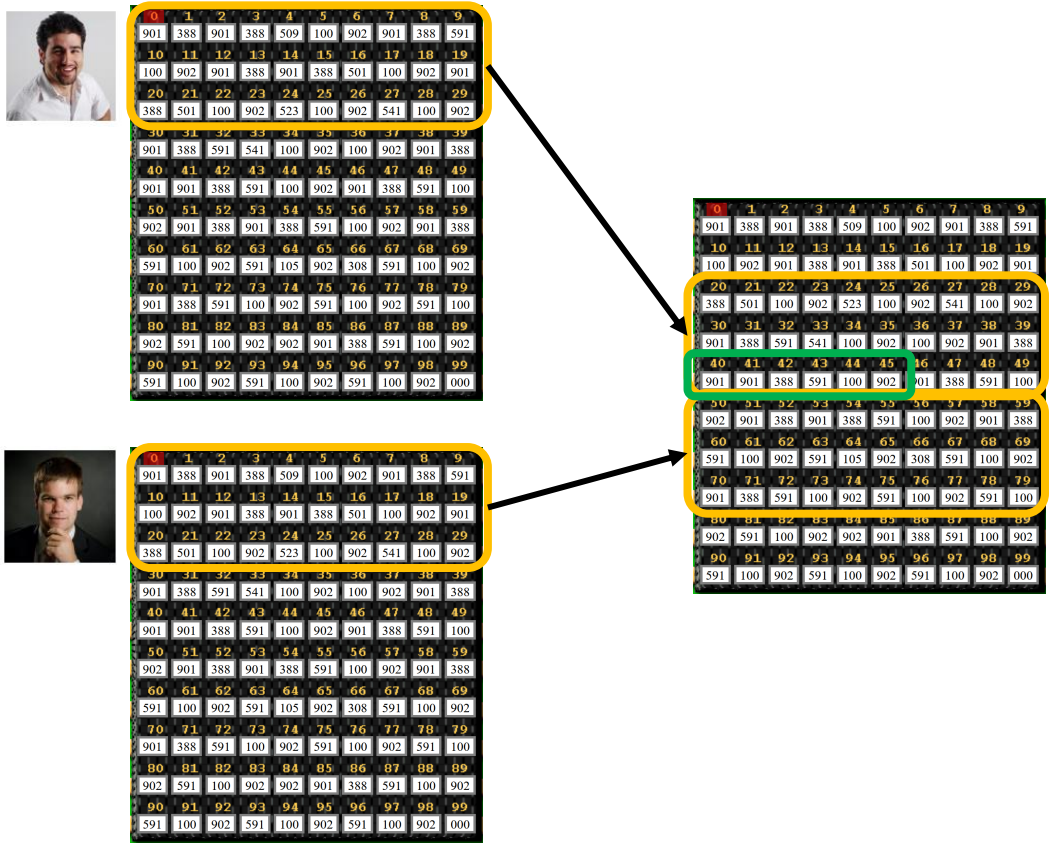
0	1	2	3	4	5	6	7	8	9
901	388	901	388	509	100	902	901	388	591
10	11	12	13	14	15	16	17	18	19
100	902	901	388	901	388	501	100	902	901
20	21	22	23	24	25	26	27	28	29
388	501	100	902	523	100	902	541	100	902
30	31	32	33	34	35	36	37	38	39
901	388	591	541	100	902	100	902	901	388
40	41	42	43	44	45	46	47	48	49
901	901	388	591	100	902	901	388	591	100
50	51	52	53	54	55	56	57	58	59
902	901	388	901	388	591	100	902	901	388
60	61	62	63	64	65	66	67	68	69
591	100	902	591	105	902	308	591	100	902
70	71	72	73	74	75	76	77	78	79
901	388	591	100	902	591	100	902	591	100
80	81	82	83	84	85	86	87	88	89
902	591	100	902	902	901	388	591	100	902
90	91	92	93	94	95	96	97	98	99
591	100	902	591	100	902	591	100	902	000

Hva med sensitiv informasjon?

Med **virtuelt minne (x86_64)** bruker hvert program «sitt eget *virtuelle* minne» med 100 celler.

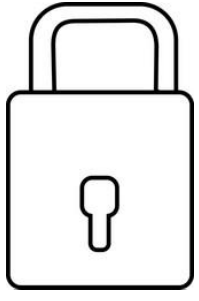
Disse ligger *egentlig* på helt andre, fysisk isolerte celler.

Håkon kan ikke lenger hente passordet til Omid.



Virtuelt Minne (x86_64)

- Vi kan bruke mer minne (RAM) enn vi egentlig har
- Et program kan ikke stjele data fra et annet program







CPU-arkitektur

LMC (og ARM) er RISC-maskiner mens x86-64 er en CISC-maskin.

CISC («Complex instruction set computer») har noen enkle og noen ganske kompliserte instruksjoner, som:

REP MOVSB kan flytte et vilkårlig antall byte fra ett sted i minnet til et annet.

FYL2XP1 beregner $x \cdot (\log_2 y + 1)$.

RISC («Reduced instruction set computer») har et begrenset antall enkle instruksjoner, men kan utføre disse ekstremt raskt og effektivt.

CISC vs RISC, et eksempel

Anta at vi skal lage kode tilsvarende Python-setningen

```
a[i] += 1
```

x86-64:

```
leaq    a, %r8
movq    i, %r9
incq    (%r8, %r9, 8)
```

LMC:

```
          LDA     LDA_A_i
          ADD     i
          STA     INCR_i
          LDA     STA_A_i
          ADD     i
          STA     INCR_i2
INCR_i    DAT     0
          ADD     v1
INCR_i2   DAT     0
          :
LDA_A_i   LDA     A
STA_A_i   STA     A
v1        DAT     1
i         DAT     0
```

	IN1020 LMC	Ny LMC	Game Boy Prosessor LR35902	Intel 8086	Intel 80386	Intel 64-bit
Bits	11	16	16	16	32	64
Generelle Registre	1	4	4	4	4	12
Instruksjoner	9	54	81*	81	140-150	>1000
Minne (maks)	100	400	16 kB	1 MB	4 GB	2^{64}
Klokkefrekvens			4 MHz	5-10 MHz	12-40 MHz	800 MHz**

*8086 + enhancements
**Itanium

Sikker programmering

Som programmerere er det viktig å skrive programmer som er

trygge («safe»), dvs vil alltid oppføre seg fornuftig

sikre («secure»), dvs tåler bevisste angrep

De første datamaskinene (1945 – 1955)

De første datamaskinene ble programmert med numeriske koder (som LMC); dette kalles gjerne **1. generasjonsspråk**.

👍 Effektivt (for maskinen)

👎 Veldig lett å gjøre ulike feil:

- bruke gal instruksjonskode
- skrive gal adresse
- ødelegge instruksjoner ved å overskrive med data
- prøve å utføre data som instruksjoner

👎 Veldig vanskelig å feilsøke

Assemblerkode

Programmeringen ble enklere med assemblerkode, ofte kalt **2. generasjonsspråk**:

- 👍 Man slipper å huske numeriske koder.
- 👍 Adresser håndteres stort sett automatisk.
- 👎 Stadig ingen feilsjekking og ingen begrensninger på hva man får lov til.
- 👎 Vanskelig å feilsøke

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

x DAT 0         // Variabel x
```

De første programmeringsspråkene (1955 – 1965)

Man ønsket å

- programmere raskere
- gjøre programmene mer lesbare
- gjøre færre feil

Resultatet ble **3. generasjonsspråk**, som f.eks. FORTRAN, COBOL, BASIC.

👍 Vi fikk array-er, setninger og funksjoner; mer strukturert programmering
👎 Stadig ingen sjekk av grenser, antall parametre etc
👎 Ingen krav om å deklarere variabler
👎 Noen i ettertid «dumme» eller «rare» ideer

Dårlige Idéer

- Sikkerhet
  - Bedre enn for assembler, men stadig en lang vei å gå

- FORTRAN

```
   22 IF (E) 17, 24, 5

   24 DO 5 K = 1,3
```

- COBOL

```
        ALTER stmt1 TO PROCEED TO stmt2.

   stmt1.
        GO TO stmt3
```

https://en.wikibooks.org/wiki/Fortran/Fortran_examples

Nye programmeringsparadigmer (1965–1980)

Nå er datamaskinene blitt større og raskere, og tiden er moden for mer avanserte programmeringsspråk.

👍 Vi får nye paradigmer som objektorientert programmering, funksjonell programmering, logisk programmering etc

👍 Vi får spesialiserte språk for systemprogrammering, databasespørring etc

👍 Man finner ut at goto ikke er så lurt.

👍 De fleste språkene får i det minste noe sjekking av grenser

Eksempler: C, C++, Scheme, Pascal, Simula, SQL

Interpreterte programmeringsspråk (1980–1990)

Mikromaskinene kommer, og det blir laget mange typisk interpreterte språk: **4. generasjonsspråk**.

Interpreterte språk som Eiffel, Perl, Python, Ruby, . .

Interpreterte programmeringsspråk (1980–1990)

Mikromaskinene kommer, og det blir laget mange typisk interpreterte språk: **4. generasjonsspråk**.

Interpreterte språk som Eiffel, Perl, Python, Ruby, . .

- 👍 Mer fleksible, krever mindre av programmereren
- 👍 Kjapt å lage småprogrammer
- 👍 Veldig god sjekking av ulovligheter (under kjøring 👎)
- 👎 Enda tregere eksekvering

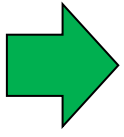
Hang on.. Interpreterte Språk?

Ikke-interpreterte språk, som C

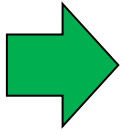
```
#include <stdio.h>

int main(){
  fprintf(stdout, «Hello, world!»);
  return 0;
}
```

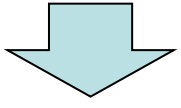
C Kildekode for «hello world»



1. Kompilering
2. Assembler
3. Linker



Kjørbar kode





Programmet «python» genereres på samme måte.

Python Source Code

1. Kompilering
2. Assembler
3. Linker

Kjørbar kode «python»



Python kildekode i C

Et script skrevet i python, interpreteres (leses og kjøres) av programmet «python», som i seg selv er et program.

```
#! /usr/bin/python

Print(«Hello, world!»)
```

Python Source Code

Python kildekode i C

1. Kompilering
2. Assembler
3. Linker

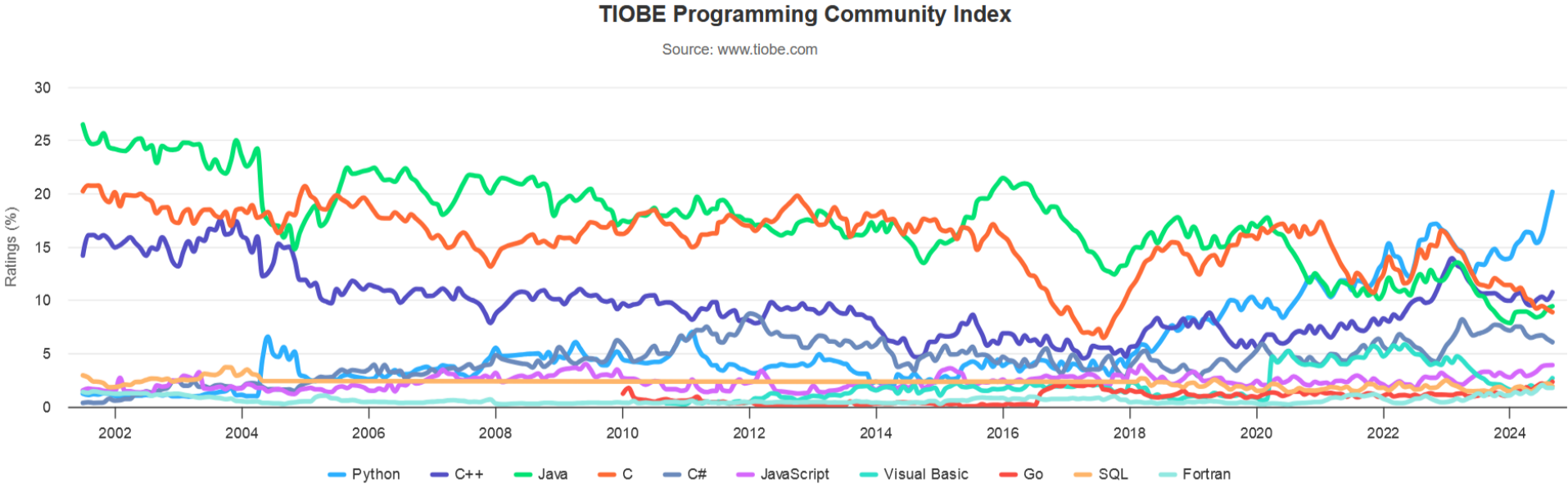
Kjørbar kode «python»



Programmeringsspråk for Internettet (1990-)

👍 Noen har gode mekanismer for distribuert eksekvering
👍 Noen er laget for integrering i nettlesere

Emerald, Java, Javascript, PHP, ...


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
2024
Python
C++
Java
C
C#
JavaScript
Visual Basic
Go
SQL
Fortran

IN1020 - Programmeringsdelen

Mandag	Onsdag	Fredag	Ukesoppgaver	Oblig
	Datahistorie	~~Digital Representasjon~~		
Digital Representasjon	Prosessoren – en introduksjon	Prosessoren - Assemblerkode	Digital representasjon	Presenteres i dag
	Hvordan håndtere kompleksitet? Abstraksjonsnivå betydning	Hvordan oppnå høyere hastighet?	Intro til LMC LMC – 1	
	Avanserte prosessorer		LMC – 2	
	Operativsystemer	Datasikkerhet		Oblig 1 deadline





