



## Dagens tema

- Datamaskinenes historie
  - Når, hvor og hvorfor ble de første datamaskiner laget?
  - Hvordan har utviklingen gått?
  - Hva inneholder en datamaskin?
- x86-prosessoren
- Enkel assemblerprogrammering

INF1070

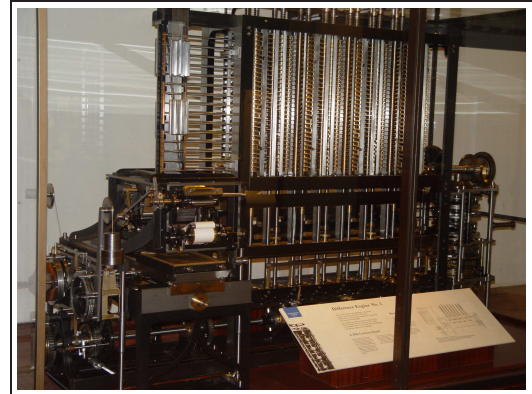
## Forhistorien

Menneskene har alltid prøvd å lage maskiner for å løse sine problemer.

### Charles Babbage

Midt på 1800-tallet var problemet *tabeller* med feil.

Charles Babbage konstruerte sin *Difference Engine* som kunne lage tabeller automatisk ved å regne ut polynomer:



INF1070

Maskinen ble først ferdig i 1991 og finnes hos *Science Museum* i London.

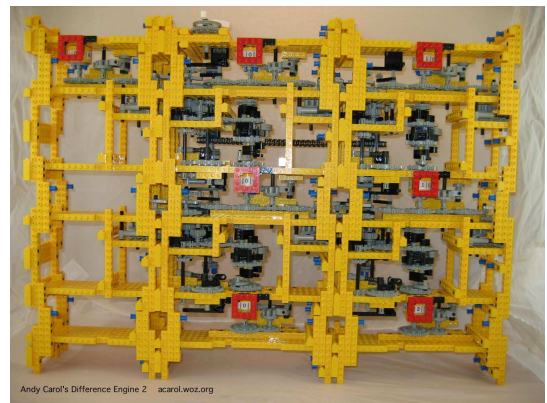
Dette var ingen ekte datamaskin:

- Binær
- Elektronisk
- Generell
- Komplett
- Program i minnet

Han arbeidet også med en *Analytical Engine* som skulle bli en generell beregningsmaskin.

INF1070

Andre har latt seg inspirere av maskinen til Babbage:

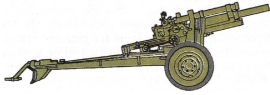


Om noen har lyst til å programmere en *Analytical Engine*, finnes det en emulator på [//www.fourmilab.ch/babbage/applet.html](http://www.fourmilab.ch/babbage/applet.html).

INF1070

## De første moderne datamaskiner

Problemet rundt 1930–40 var kanoner.



Det er mulig å beregne en prosjektilbane, men det er mye arbeid for en matematiker. *U.S. Army Ordnance Department Ballistic Research Laboratory* trengte data for dusinvis av nye kanoner i 1930-årene.

### Løsning

Lag en arbeidsbeskrivelse, og la egne «beregner» gjøre jobben (etter en kort opplæring).

Fra en eldre utgave av *Webster's Dictionary*:

**computer** n, one that computes; *specif*: an automatic electronic machine for performing calculations

INF1070

## En typisk arbeidsbeskrivelse

- ③ Tast 2,78 inn i regnemaskinen.
- ④ Multipliser tallet med 3,1415926535. Skriv svaret i rubrikk 28.
- ⑤ Hvis tallet i rubrikk 71 er  $< 0$ , gå til punkt 88.
- ⑥ Tast tallet i rubrikk 29 inn i regnemaskinen.
- ⑦ Legg til 1.
- ⑧ Gå til punkt 4.

28	8,7336276	:
29	3	:
30	-11	:
31		:
32	1,25	:
33		:
34		:

INF1070

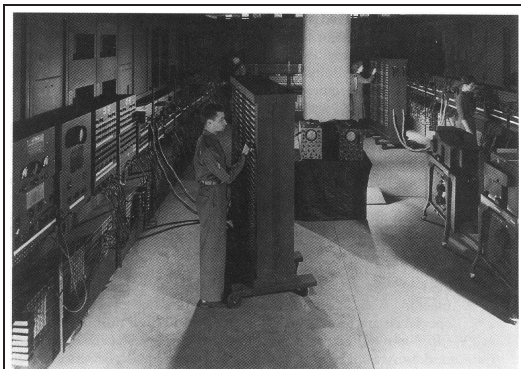
### Problem

Hver bane tok opptil 20 timer å beregne (selv med elektrisk bordregnemaskin), og man trengte 2–4000 baner for hver kanon.

### Løsning

Lag en maskin som gjør dette automatisk.

*Moore School of Electrical Engineering* ved universitetet i Pennsylvania gjorde det med penger fra *Ballistic Research Laboratory*. Resultatet ble Eniac som ble ferdig i 1946. Den målte  $2\frac{1}{2} \times 1 \times 30$  m, veide 30 tonn og inneholdt 19 000 radiorør. Den kunne beregne en kulebane på rundt 30 s. Den brukte nesten 200 Kwatt.



INF1070

## Oppbyggingen av Eniac

Tanken bak Eniac var å kopiere en menneskelig beregner. Derfor fikk man

**Aritmetisk enhet** («ALU») tilsvarte regnemaskinen. Den kunne de fire regneartene:

+ - × ÷

Regnemaskinen har et tall for videre beregning; datamaskinen har et **register** for dette.

**Minnet** tilsvarte arket med

mellomresultater. Datamaskinen kunne skrive innholdet av registeret til en celle i minnet, og hente innholdet av en celle tilbake til registeret.

**Programmet** tilsvarte beregnerens arbeidsbeskrivelse. Det skulle følges helt slavisk.

INF1070

## Programmet

Et program for datamaskinen inneholdt de samme elementene som beregnerens arbeidsbeskrivelse:

**Aritmetiske operasjoner** var mulig i de fire regneartene; svaret kom i registeret.

**Mellomlagring av data** skjedde ved at registeret ble kopiert til en angitt celle i minnet. Derfra kunne det hentes tilbake ved behov.

**Hopp** til en angitt instruksjon var nødvendig for å kunne gå i løkker.

**Tester** i forbindelse med hopp var typisk på om registeret var  $< 0$ ,  $= 0$  eller  $> 0$ .

Programmene ble etter hvert kodet som tall (men Eniac ble kodet med kabler).

INF1070

## Hva er beholdt i dag?

Alle disse instruksjonene finnes i dagens datamaskiner. I tillegg har vi fått

- flere registre (ofte 8-32),
- binær representasjon av tallene (i stedet for desimal),
- operasjoner på bit (skifting, masking) og
- flyt-tall som kan lagre svært store og svært små tall (som  $6,02 \times 10^{23}$ ).

## Programmerer noen i maskinkode i dag?

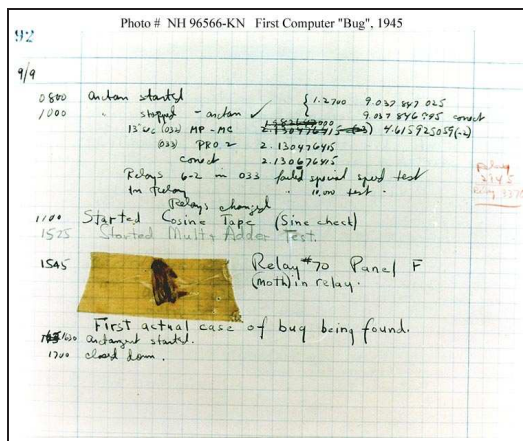
Maskinkode brukes når man har behov for ekstremt raske beregninger, for eksempel grafiske applikasjoner. (Og når man lager virus!)

Kjennskap til det som skjer i maskinen vil imidlertid gjøre oss til mye bedre programmere og brukere!

INF1070

## Den aller første «bug»

Også de første datamaskinene hadde feil. Den 9. september 1945 kl 15.45 fant man feilen i relé nr 70 i panel F i en Mark II Aiken elektromekanisk datamaskin og foretok den første «debugging»:



INF1070

**Generasjoner**  
Det er vanlig å dele datamaskinene inn i generasjoner:

Gen	År	Teknologi	Størrelse	Instr./sek	Pris (2005-kr)
1	1945-60	Radiorør	10m <sup>3</sup>	2000	50 mill
2	1960-68	Transistorer	650dm <sup>3</sup>	500 000	40 mill
3	1968-78	Integrerte kretser	80dm <sup>3</sup>	300 000	1 500 000
4	1978-??	Lsi og vlsi	0,1-20dm <sup>3</sup>	10 <sup>9</sup>	10 000

## Størrelser

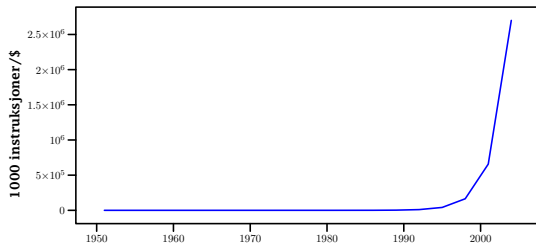
Det er også vanlig å dele inn datamaskiner etter størrelse:

stormaskin («mainframe»), minimaskin, mikromaskin, bærbart maskin, PDA

INF1070

### Moore's lov

Tegner vi et diagram over utviklingen av ytelsen i forhold til prisen, ser vi at grovt sett blir ytelsen fordoblet hvert 1½ år.

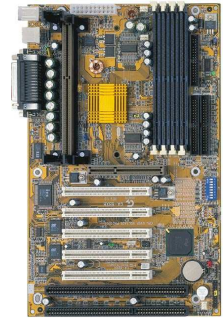


Ingen vet hvorfor det er slik, men dette kalles **Moore's lov** etter mannen som formulerte den.

INF1070

## Oppbyggingen av en datamaskin

Det viktigste i en moderne datamaskin er *hovedkortet* («motherboard»):



Grovt sett inneholder det

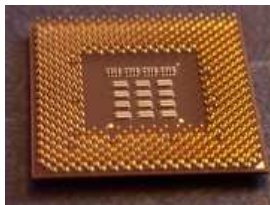
- En prosessor
- Minne (for både program og data)
- Klokke
- Kontrollere for periferutstyr.

Dette er den såkalte *Von Neuman-arkitekturen* som omtrent alle datamaskiner siden Eniac har fulgt.

INF1070

### Prosessoren (Bryant-boken 3.1)

Prosessoren er «selve datamaskinen».



Det finnes mange ulike prosessorer fra ulike produsenter. I dette kurset skal vi se på **IA-32 (= x86)** fra *Intel*.

Det finnes en hel familie med x86-prosessorer fra Intel, fra 8086 (introdusert i 1987) til dagens Pentium 4.

INF1070

	År	#trans	
8086/-88	1978	0,029M	16(8)-bits; IBM PC med MS-DOS; 640K adresserbart
80286	1982	0,134M	MS Windows; 1M adresserbart
i386	1985	0,275M	32-bits; Unix; flatt minne
i486	1989	1,9M	Integrert 8087 (flyt-tall)
Pentium	1993	3,1M	
PentiumPro	1995	6,5M	Nytt design (P6)
Pentium/MMX	1997	4,5M	Op for heltallvektorer
Pentium II	1997	7M	= PentiumPro + Pentium/MMX
Pentium III	1999	8,2M	Op for heltalls- og flyt-tallsvektorer
Pentium 4	2001	42M	128-bits data

INF1070

## Prosessoren

En prosessor inneholder:

- En ALU («Arithmetic and Logic Unit») eller flere
- Register
- Kontrollogikk

(I dag ligger det også coprosessorer, *cache* og mye annet på prosessorbrikken, men dette er logisk sett ikke del av prosessoren.)

INF1070

## Register (Bryant-boken 3.4)

En x86-prosessor har følgende registre:

### Mer eller mindre generelle 32-bits

%AX

%EAX		%AH	%AL
%EBX		%BH	%BL
%ECX		%CH	%CL
%EDX		%DH	%DL

%EBP
%ESP
%ESI
%EDI

### Spesielle 16-bits

%CS	%ES
%SS	%FS
%DS	%GS

### Spesielle 32-bits

%EFLAGS
%EIP

INF1070

## Assembler-programmering

### Et meget enkelt eksempel

Denne lille C-funksjonen returnerer verdien 19:

```
int nineteen (void)
{
    return 19;
}
```

Denne assembler-funksjonen gjør det samme:

```
.globl nineteen
nineteen:
    movl    $19, %eax
    ret
```

Den legger verdien 19 i register **%EAX** der alle funksjoner legger resultatverdien, og returnerer.

INF1070

## Test-program

Dette programmet kan teste funksjonen:

```
#include <stdio.h>

extern int nineteen (void);

int main (void)
{
    printf("nineteen() = %d\n", nineteen());
    return 0;
}
```

### Kompilering, assemblering og kjøring

Programmet gcc kan håndtere både C-kompilering og assemblering:

```
gcc test-nineteen.c nineteen.s -o test-nineteen
```

Det ferdige programmet kan kjøres:

```
> ./test-nineteen
nineteen() = 19
```

INF1070

**Maskinkode kontra assemblerkode** (Bryant-boken 4.1)  
 Programmet ligger lagret som *maskinkode*, dvs bit-mønstre i minnet.  
 Funksjonen *nineteen* ligger *egentlig* lagret som<sup>†</sup>

```

CAS LISTING nineteen.s           page 1
1                               .globl nineteen
2                               nineteen:
3 0000 B8130000                 movl   $19, %eax
3 0000 00                       ret
4 0005 C3
  
```

Siden slike numeriske koder er vanskelige huske, brukes i stedet *assemblerkode* som bare er huskekoder for maskinstruksjonene.  
<sup>†</sup> Ved å skrive gcc -Wa,-a-c minfil.s -minfil.list kan vi få se hvordan maskinkoden ser ut.

INF1070

## Assemblerkode kontra høynivåprogrammering

Det er stor forskjell på kompilering og programmering i et høynivåspråk:

### Kompilering

- Vi vet ikke hvilke maskininstruksjoner som genereres (men er ikke interessert).
- Programmet kan (stort sett) flyttes uendret til en annen datamaskin uansett fabrikk eller operativsystem.

### Assembling

- Vi vet nøyaktig hvilke maskininstruksjoner som genereres.
- Programkoden kan ikke flyttes til datamaskiner med annet instruksjonssett eller operativsystem.

Med andre ord: vi har full kontroll over den genererte koden.

INF1070

## Et eksempel med én parameter

Parametre ligger på den *stakken* så man får tak i dem med «4(%esp)», «8(%esp)», ...

```

.globl incr
incr:
    movl   4(%esp),%eax
    incl   %eax
    ret
  
```

### Et testprogram:

```

#include <stdio.h>

extern int incr (int n);

int main (void)
{
    int i;

    for (i = 10; i <= 14; ++i)
        printf("incr(%d) = %d\n", i, incr(i));
    return 0;
}
  
```

### Resultatet av kjøringen:

```

incr(10) = 11
incr(11) = 12
incr(12) = 13
incr(13) = 14
incr(14) = 15
  
```

INF1070

## Et eksempel til

Denne funksjonen har to parametre:

```

.globl add
add:
    movl   4(%esp),%eax
    addl   8(%esp),%eax
    ret
  
```

```

#include <stdio.h>

extern int add (int a, int b);

int tab[] = {1, 17, -3};

int main (void)
{
    int tab_length = sizeof(tab)/sizeof(int);
    int i1, i2;

    for (i1 = 0; i1 < tab_length; ++i1) {
        for (i2 = 0; i2 < tab_length; ++i2) {
            int a1 = tab[i1], a2 = tab[i2];

            printf("add(%d,%d) = %d\n", a1, a2, add(a1,a2));
        }
    }
    return 0;
}
  
```

```

add(1,1) = 2
add(1,17) = 18
add(1,-3) = -2
add(17,1) = 18
add(17,17) = 34
add(17,-3) = 14
add(-3,1) = -2
add(-3,17) = 14
add(-3,-3) = -6
  
```

INF1070

## Oppsummering

Vi kjenner nå til følgende instruksjoner:

movl Flytt en verdi

addl Addér en verdi til en annen

subl Subtraher en verdi fra en annen

incl Øk en verdi med 1

decl Senk en verdi med 1

Operander kan være:

\$17 Konstanter

%eax Registeret %EAX

4(%esp) Parametre (på stakken)

INF1070