

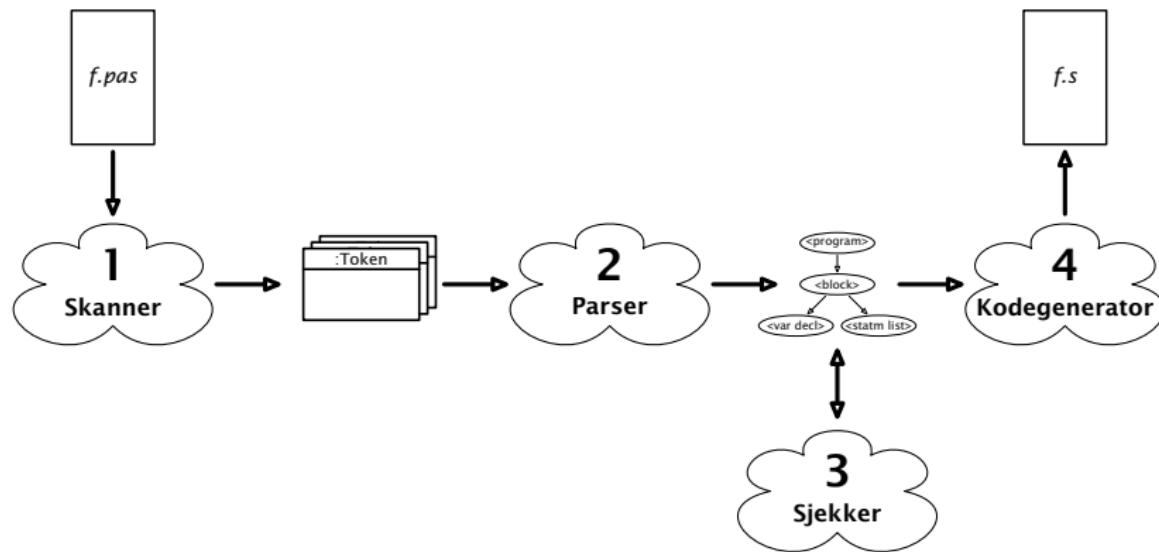
Dagens tema

## Dagens tema:

### Maskinkode del 1

- Litt datamaskinhistorie
- Hva er maskin- og assemblerkode?
- x86-prosessoren
- Programkode for setninger
- Konstanter og uttrykk

## Prosjektoversikt



Charles Babbage

## Datamaskinenes historie

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

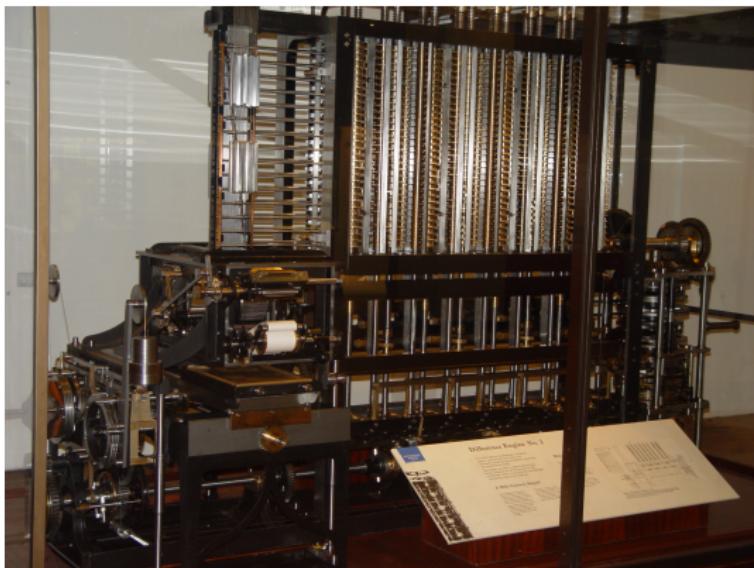
Midt på 1800-tallet var det store problemet *tabeller* med feil.

**Charles Babbage** konstruerte på 1830-tallet sin *Difference Engine* som kunne lage tabeller automatisk ved å løse differensligninger. (Den ble først ferdig i 1991.)

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

Charles Babbage

*The difference engine  
på Science Museum i  
London.*



En demo: <https://www.youtube.com/watch?v=jiRgdaknJCg>

Tidlige datamaskiner

## De første moderne datamaskiner

Problemet i 1930-årene 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.

### Tradisjonell løsning

Lag *arbeidsbeskrivelse*, og la egne «beregnere» gjøre jobben.

## Tidlige datamaskiner

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

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



## Tidlige datamaskiner

### Problem

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

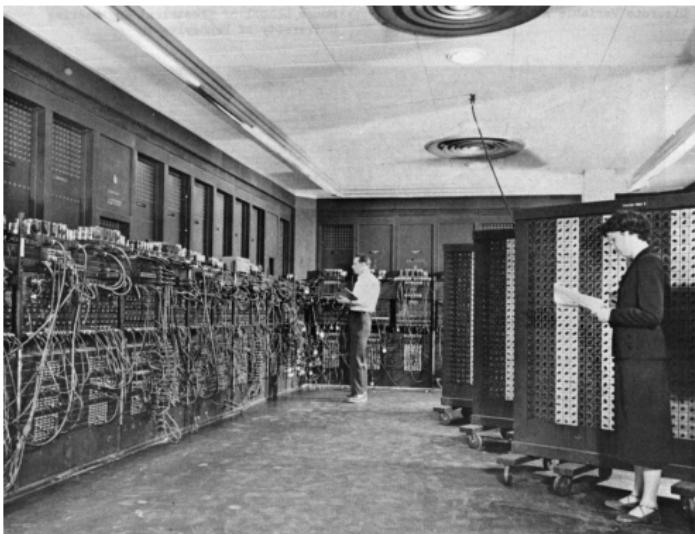
### Løsning

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 1945. Den kunne beregne en kulebane på drøyt 10 s.

Eniac målte  
 $2\frac{1}{2} \times 1 \times 30$  m, veide  
30 tonn og inneholdt  
18 000 radiorør.

Den var i drift til 1955.



# Oppbygningen av Eniac

Tanken var å kopiere en menneskelig beregner. Den har **Aritmetisk enhet** («ALU») tilsvarte regnemaskinen med de fire regneartene:

+ - × :

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

**Minnet** tilsvarte et ark med mellomresultater.

Datamaskinen kunne overføre innholdet av registeret til eller fra en celle i minnet.

**Programmet** tilsvarte beregnerens arbeidsbeskrivelse.

Det skulle følges helt slavisk.

# 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.

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

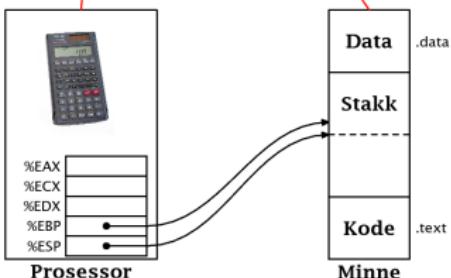
**Tester** i forbindelse med hopp var typisk på om registeret var  $< 0$ ,  $= 0$  eller  $> 0$ .

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

Ennå i dag

## En moderne datamaskin

har grovt sett samme oppbygning den dag i dag.



## Prosessoren

## x86-prosessoren

Denne prosessoren er den mest suksessrike gjennom tidene.

- Introdusert i 1978 med **8086**.
- Produseres fremdeles (som x86-64).
- Brukes i de fleste PC-er.



## Prosessoren

Minnet inneholder tre former for data:

- ① **Data** inneholder globale variabler (men vi skal ikke bruke det).
- ② **Stakken** inneholder lokale variabler og parametre.
- ③ **Koden** er programmet (dvs instruksjonene i numerisk form).

## Prosessoren

Prosessoren inneholder:

- ➊ En regneenhet (kalt ALU = «Arithmetic Logic Unit») som kan
  - ➊ utføre de fire regneartene (+, -, × og :)
  - ➋ sammenligne to tall
  - ➌ avgjøre om programmet skal hoppe
  - ➍ flytte tall
- ➋ Registre
  - ➊ %EAX, %ECX og %EDX benyttes til beregninger og sammenligninger. (%AL er en del av %EAX.)
  - ➋ %EBP («Extended Base Pointer») og %ESP («Extended Stack Pointer») peker på data i minnet.
- ➌ En flyttallsenhet (som vi ikke skal bruke nå)

# Instruksjonene vi skal bruke

<code>movl</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Flytt <code>&lt;v<sub>1</sub>&gt;</code> til <code>&lt;v<sub>2</sub>&gt;</code> .
<code>cdq</code>		Omform 32-bits %EAX til 64-bits %EDX:%EAX.
<code>leal</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Flytt <code>&lt;v<sub>1</sub>&gt;</code> s <i>adresse</i> til <code>&lt;v<sub>2</sub>&gt;</code> .
<code>pushl</code>	<code>&lt;v&gt;</code>	Legg <code>&lt;v&gt;</code> på stakken.
<code>popl</code>	<code>&lt;v&gt;</code>	Fjern toppen av stakken og legg verdien i <code>&lt;v&gt;</code> .
<code>negl</code>	<code>&lt;v&gt;</code>	Skift fortegn på <code>&lt;v&gt;</code> .
<code>addl</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Adder <code>&lt;v<sub>1</sub>&gt;</code> til <code>&lt;v<sub>2</sub>&gt;</code> .
<code>subl</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Subtraher <code>&lt;v<sub>1</sub>&gt;</code> fra <code>&lt;v<sub>2</sub>&gt;</code> .
<code>imull</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Multipliser <code>&lt;v<sub>1</sub>&gt;</code> med <code>&lt;v<sub>2</sub>&gt;</code> .
<code>idivl</code>	<code>&lt;v&gt;</code>	Del %EDX:%EAX med <code>&lt;v&gt;</code> ; svar i %EAX; rest i %EDX.
<code>andl</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Logisk AND.
<code>orl</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Logisk OR.
<code>xorl</code>	<code>&lt;v<sub>1</sub>&gt;, &lt;v<sub>2</sub>&gt;</code>	Logisk XOR.

## Prosessoren

<b>call</b> <lab> <b>enter</b> \$<n <sub>1</sub> >, \$<n <sub>2</sub> >  <b>leave</b> <b>ret</b>	Kall funksjon/prosedyre i <lab>. Start en funksjon/prosedyre på blokknivå <n <sub>2</sub> > med <n <sub>1</sub> > byte lokale variabler. Rydd opp når funksjonen/prosedyren er ferdig. Returner fra funksjon/prosedyre.
<b>cmp</b> l    <v <sub>1</sub> >, <v <sub>2</sub> > <b>jmp</b> <lab> <b>je</b> <lab> <b>sete</b> <v> <b>setne</b> <v> <b>setl</b> <v> <b>setle</b> <v> <b>setg</b> <v> <b>setge</b> <v>	Sammenligning <v <sub>1</sub> > og <v <sub>2</sub> >. Hopp til <lab>. Hopp til <lab> hvis =. Sett <v>=1 om =, ellers <v>=0. Sett <v>=1 om ≠, ellers <v>=0. Sett <v>=1 om <, ellers <v>=0. Sett <v>=1 om ≤, ellers <v>=0. Sett <v>=1 om >, ellers <v>=0. Sett <v>=1 om ≥, ellers <v>=0.

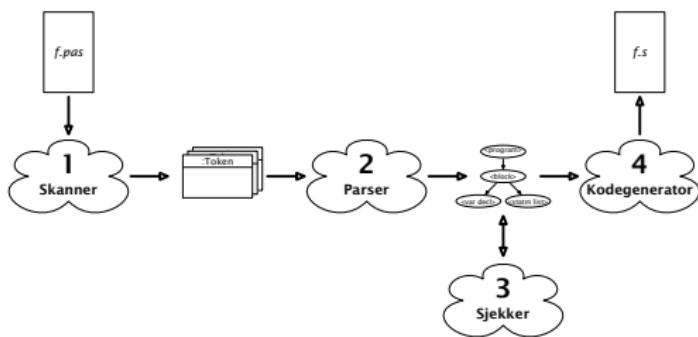
## Prosessoren

## Formålet

Vårt oppdrag er å lage en kompilator:

**Inndata** er tre-representasjonen av Pascal2016-programmet laget i del 2 og 3.

**Utdata** er en fil med x86 assemblerkode.



## Maskinkode

Dette er den *numeriske representasjonen* av instruksjonene; vi skal ikke benytte den i dette kurset.

## Assemblerkode

Dette er den *tekstlige representasjonen* av instruksjonene.

func:  
Navnelapp

movl  
Instruksjon

\$0,%eax  
Parametre

# Initier til 0.  
Kommentar

## Maskin- og assemblerkode

## Noen eksempler

```
Start:  movl    $17,%eax   # Legg verdien 17 i %EAX
        movl    $-2,%edx   # Legg verdien -2 i %EDX
        addl    %edx,%eax   # Addér %EDX til %EAX
        jmp     Start      # Hopp til Start
```

## Et eksempel

```
# Code file created by Pascal2016 compiler 2016-11-07 15:47:07
.globl main
main:
        call    prog$mini_1          # Start program
        movl    $0,%eax             # Set status 0 and
        ret                 # terminate the program
prog$mini_1:
        enter   $32,$1              # Start of mini
        movl    $120,%eax            #     'x'
        pushl   %eax                # Push next param.
        call    write_char           #     'x'
        addl    $4,%esp              # Pop param.
        leave
        ret                 # End of mini
```

program Mini;  
begin  
 write('x');  
end.

## Klassen CodeFile

# Klassen Main.CodeFile

```
package main;

import java.io.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public class CodeFile {
    private String codeFileName;
    private PrintWriter code;
    private int numLabels = 0;

    CodeFile(String fName) {
        codeFileName = fName;
        try {
            code = new PrintWriter(fName);
        } catch (FileNotFoundException e) {
            Main.error("Cannot create code file " + fName + "!");
        }
        code.println("# Code file created by Pascal2016 compiler " +
            new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date()));
    }
}
```

## Klassen CodeFile

```
public void genInstr(String lab, String instr, String arg, String comment) {  
    if (lab.length() > 0)  
        code.println(lab + ":";  
    if ((instr+arg+comment).length() > 0) {  
        code.printf("      %-7s %-23s ", instr, arg);  
        if (comment.length() > 0) {  
            code.print("# " + comment);  
        }  
        code.println();  
    }  
}
```

## Et eksempel

Anta at vi har Pascal2016-koden `v := 1 + 2;`

Disse x86-instruksjonene gjør dette:

```
movl    $1,%eax      # %EAX=1  %ECX=? %EDX=? stack=...
pushl    %eax          # %EAX=1  %ECX=? %EDX=? stack=1 ...
movl    $2,%eax      # %EAX=2  %ECX=? %EDX=? stack=1 ...
movl    %eax,%ecx      # %EAX=2  %ECX=2  %EDX=? stack=1 ...
popl    %eax          # %EAX=1  %ECX=2  %EDX=? stack=...
addl    %ecx,%eax      # %EAX=3  %ECX=2  %EDX=? stack=...
movl    -4(%ebp),%edx  # %EAX=3  %ECX=2  %EDX=ba stack=...
movl    %eax,-36(%edx) # v = 3
```

Lag riktig kode!

## En presisering

Det finnes mange mulige kodebiter som gjør det samme. I kompendiet står angitt ganske nøyaktig hvilke som skal brukes.

**NB!**

Det er viktigere at koden er riktig enn at den er rask!

Metoden 'genCode'

## Hvordan implementere kodegenerering

Det beste er å følge samme opplegg som for å sjekke programkoden:

### Kodegenerering

Legg en metode `genCode` inn i alle klasser som representerer en del av Pascal2016-programmet (dvs er subklasse av `PascalSyntax`).

## Metoden 'genCode'

```
class WhileStatm extends Statement {  
    Expression test;  
    Statement body;  
  
    @Override void check(Block curScope, Library lib) {  
        ;  
    }  
  
    @Override void genCode(CodeFile f) {  
        ;  
    }  
  
    static WhileStatm parse(Scanner s) {  
        ;  
    }  
  
    @Override void prettyPrint() {  
        ;  
    }  
}
```

## Konvensjoner

# Konvensjoner

Kodegenerering blir mye enklere om vi setter opp noen fornuftige konvensjoner:

- Alle beregninger skal ende opp i %EAX.
- %ECX og %EDX er hjelpperegistre.
- Hovedstakken (aksessert via %ESP) er til
  - variabler ( neste uke)
  - mellomresultater
  - funksjons- og prosedyrekall ( neste uke)

## While-setningen

I kompendiet finnes kodeskjemaer for det meste i Pascal2016. Skjemaet for while-setningen ser slik ut:

while <e> do <S>

⇒

```
(lab1):
  (Beregn <e> med svar i %EAX)
  cmpl    $0,%eax
  je     (lab2)
  <S>
  jmp    (lab1)
(lab2):
```

## Navnelapper

## Lokale navnelapper

Når vi skal lage slike hopp, trenger vi stadig nye navnelapper. Dette kan vi få fra `CodeFile`-objektet:

```
public class CodeFile {  
  
    private int numLabels = 0;  
  
    public String getLocalLabel() {  
        return String.format(".L%04d", ++numLabels);  
    }  
}
```

Komplett kode

## Hele koden for WhileStatm

```
class WhileStatm extends Statement {  
    Expression expr;  
    Statement body;  
  
    @Override void genCode(CodeFile f) {  
        String testLabel = f.getLocalLabel(),  
              endLabel = f.getLocalLabel();  
  
        f.genInstr(testLabel, "", "", "Start while-statement");  
        expr.genCode(f);  
        f.genInstr("", "cmpl", "$0,%eax", "");  
        f.genInstr("", "je", endLabel, "");  
        body.genCode(f);  
        f.genInstr("", "jmp", testLabel, "");  
        f.genInstr(endLabel, "", "", "End while-statement");  
    }  
}
```

Konstanter

## Uttrykk

Uttrykk (og alle deluttrykk) skal resultere i en verdi i %EAX-registeret.

### Konstanter



Dette fungerer også for char-literaler; da bruker vi ASCII-verdien.

Tilsvarende forutsetter vi False=0 og True=1.

## Konstanter

## Operatorer

 $\langle e_1 \rangle + \langle e_2 \rangle$ 

```
(Beregn <e1> med svar i %EAX)
pushl %eax
(Beregn <e2> med svar i %EAX)
movl %eax,%ecx
popl %eax
addl %ecx,%eax
```

 $\langle e_1 \rangle \text{ div } \langle e_2 \rangle$ 

```
(Beregn <e1> med svar i %EAX)
pushl %eax
(Beregn <e2> med svar i %EAX)
movl %eax,%ecx
popl %eax
cdq
idivl %ecx
```

Konstanter

## Neste uke

Til neste uke tar vi det som står igjen:

- sammenligninger
- blokker
- variabler
- funksjons- og prosedyrekall
- biblioteket
- hovedprogrammet