

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Mer om prosessoren, og sikker programmering

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



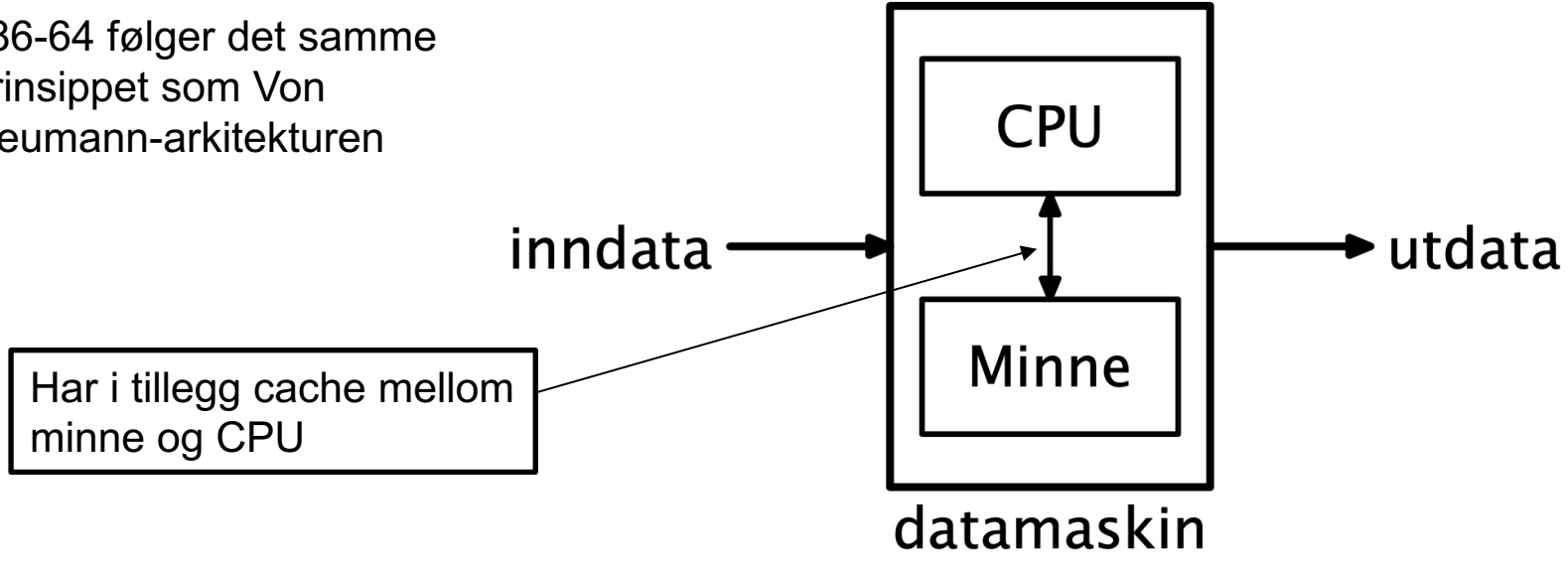
I dag

- En moderne prosessor: x86-64
  - Likheter og forskjeller med LMC
- Sikker programmering

Prosessoren x86-64

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

x86-64 følger det samme prinsippet som Von Neumann-arkitekturen



Eksekveringsløkken

Både LMC og x86-64 har samme eksekveringsløkke:

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

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

Basic five-stage pipeline

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

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

					%EAX				
							%AX		
%RAX	1 byte						%AH	%AL	Accumulator
%RBX							%BH	%BL	Base
%RCX							%CH	%CL	Counter
%RDX							%DH	%DL	Data

					%EBP			
							%BP	
%RBP								%BPL
%RSP								%SPL
%RDI								%DIL
%RSI								%SIL

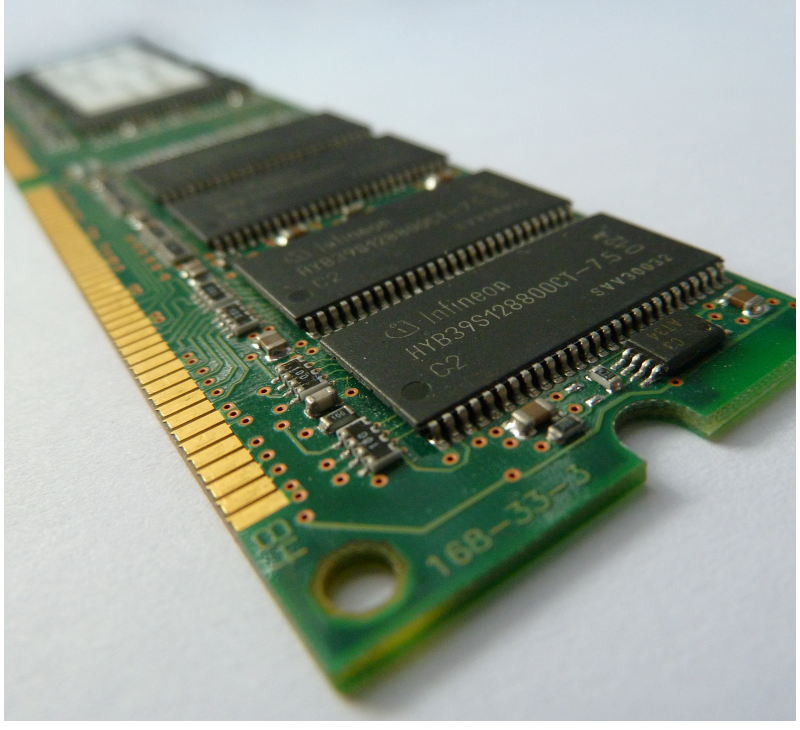
					%R8D			
							%R8W	
%R8								%R8B
%R9								%R9B
%R10								%R10B
%R11								%R11B
%R12								%R12B
%R13								%R13B
%R14								%R14B
%R15								%R15B

Minne (RAM)

LMC har 100 celler i minnet, og hver celle kan lagre verdier –999–+999.

x86-64 kan ha inntil $2^{64} \approx 1,8 \cdot 10^{19}$ celler; dvs ca 17 000 000 TB («terabyte») eller 16 EB («exabyte»), men ingen maskiner i dag har så mye installert. Hver celle inneholder én byte (= 8 bit).

I tillegg har x86-64 støtte for virtuelt minne.



CPU-arkitektur

LMC er en RISC-maskin mens x86-64 er en CISC-maskin.

CISC («Complex instruction set computer») har noen enkle og noen ganske kompliserte instruksjoner, f.eks.

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 RICS, 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
```

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

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.

Eksempel på feil

```
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.



```
000024
000025      PROCEDURE DIVISION.
000026      0001-MAIN.
000027          INSPECT FUNCTION REVERSE(STR-1)
000028                  TALLYING WS-LEN1 FOR LEADING SPACES.
000029          COMPUTE  WS-LEN = LENGTH OF STR-1 - WS-LEN1.
000030          DISPLAY WS-LEN.
000031          MOVE 1 TO I.
000032          MOVE WS-LEN TO J.
000033          PERFORM REV-PARA WS-LEN TIMES.
000034          DISPLAY STR-1.
000035          DISPLAY STR-2.
000036          GOBACK.
000037       REV-PARA.
000038          MOVE STR-1(J:1) TO STR-2(I:1).
000039          SUBTRACT 1 FROM J.
000040          ADD 1 TO I.
000041          EXIT.
****** ********************************* Bottom of Data ********************************
```

👍 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

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

👎 De fleste nye språkene lager tregere kode

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, . .

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

Hvorfor lager man ikke 100 % trygge og sikre språk?

- Språk som aldri gir
  - Overflytfeil
  - NullPointerException
  - Buffer overrun
  - Divide by zero-feil
  - Ingen ArrayIndexOutOfBoundsException
  - Ingen TypeError
  - Ingen MethodNotFoundException

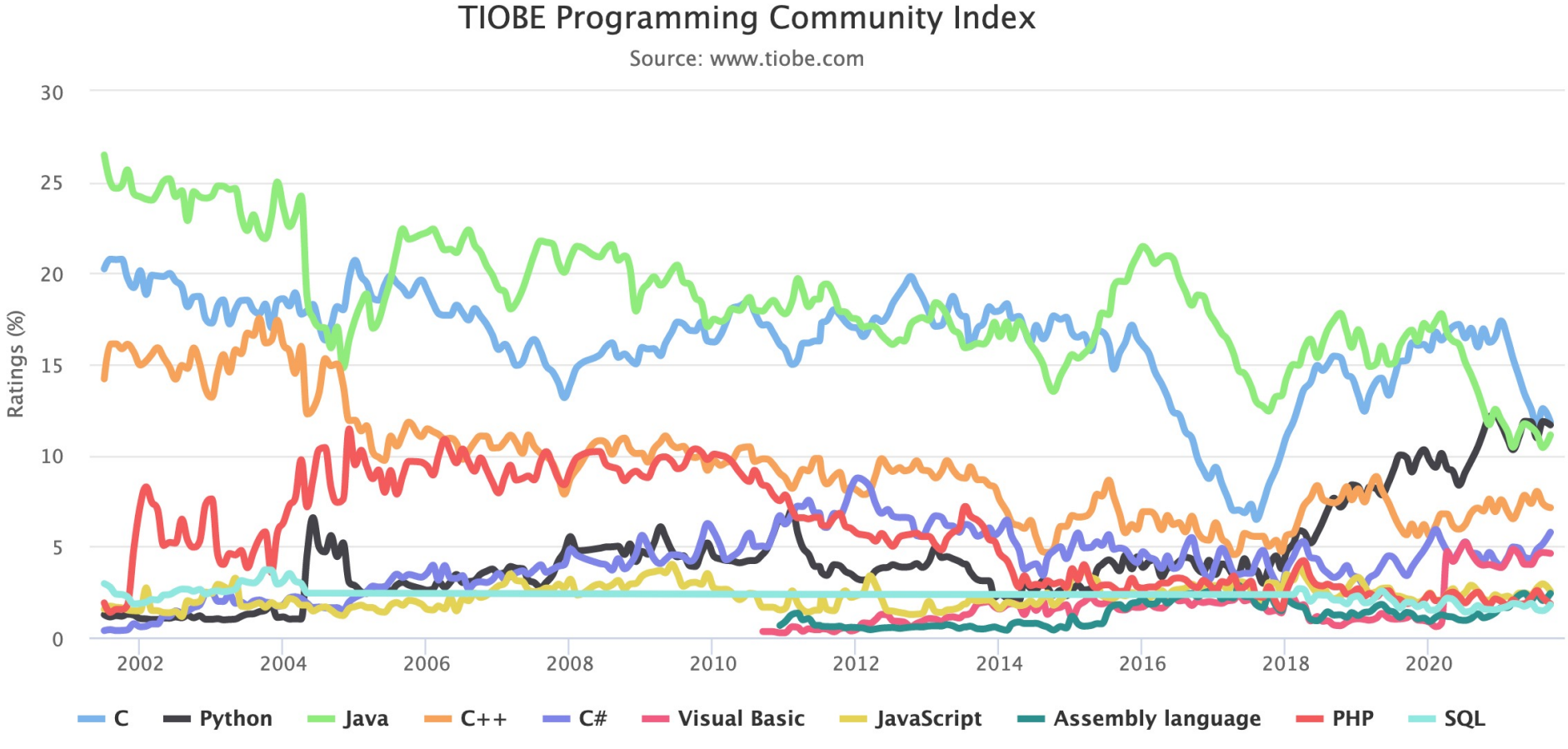
Feil og «Feil»

- Det er forskjell på at programmeringsspråket kaster et unntak, og at hele maskinen booter, eller overskriver et tilfeldig minneområde, f.eks.

  Førstnevnte er **trygt**, sistnevnte er ikke. Og det er heller ikke **sikkert.**

Hvilke trender ser vi innen programmeringsspråk i dag?

- Multiparadigme-språk blir populært
  - Java, C#, TypeScript, etc tar inn trender fra både OO og FP
- Streng typesjekking blir mer oppnåelig
  - Både sikkerhet og trygghet
    (programmet er trygt for angrep, og gjør ikke «gale» ting)
  - Rust, F#, Scala, TypeScript, Elm

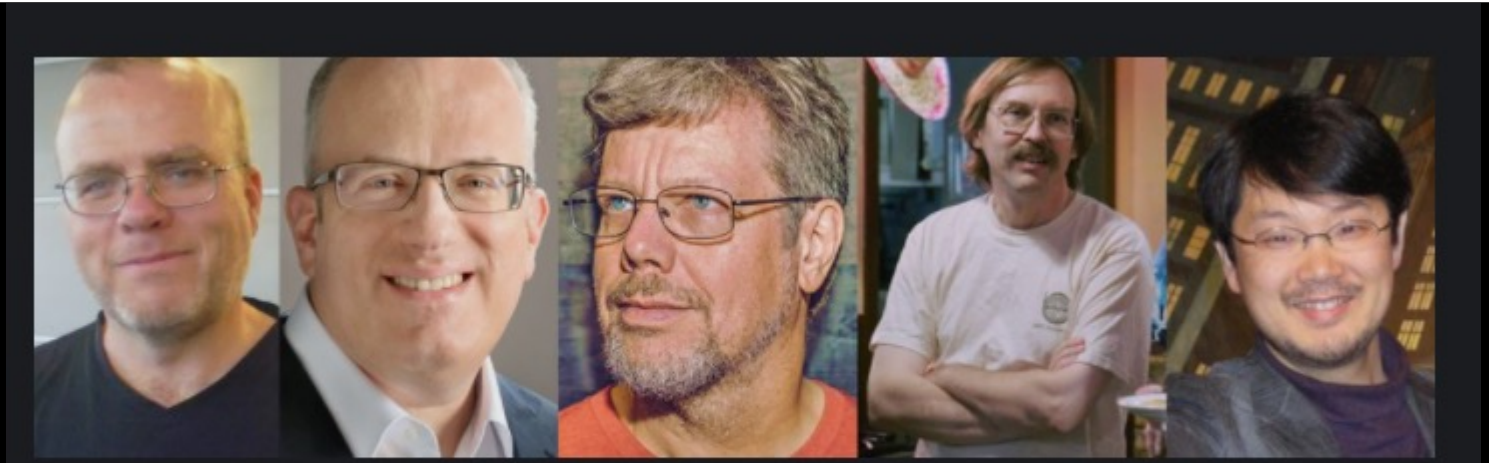

TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
C
Python
Java
C++
C#
Visual Basic
JavaScript
Assembly language
PHP
SQL

En oppfordring til slutt!

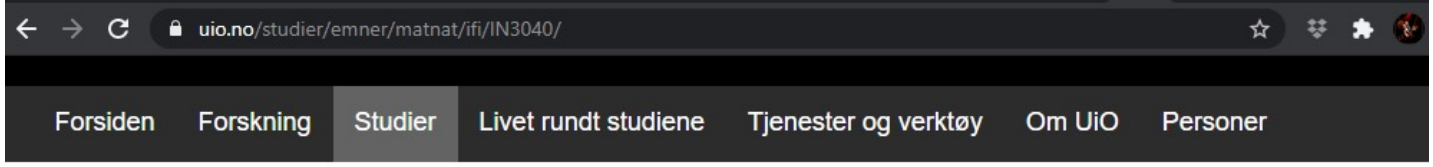
Det å lage programmeringsspråk er fremdeles et felt der enkeltpersoner kan gjøre en innsats.

Etter et masterstudium ved Ifi (med fornuftig valg av frie emner) vil dere ha nok teknisk kunnskap til å kunne lage deres eget programmeringsspråk.

Flere av dagens mest populære språk er laget av *ivrige enkeltpersoner*: **Perl, Python, Ruby, JavaScript, C++, PHP!**



From the left: Rasmus Lerdorf (creator of PHP), Brendan Eich (creator of JavaScript), Guido van Rossum (creator of Python), Larry Wall (creator of Perl) and Yukihiro Matsumoto (creator of Ruby)



IN3040 – Programmeringsspråk

Hva lærer du?

Etter å ha tatt IN3040 kan du:

- Enklere lære nye programmeringsspråk, kjenne til en større bredde av eksisterende språk, og forstå konstruksjonene i disse bedre
- Delta i utvikling av, og diskusjon rundt, nye programmeringsspråk basert på en forståelse av designrommet for slik utvikling, og hvilke avveininger som ligger bak de språkene vi har i dag, og deres særegenheter
- Forklare hvordan sentrale språkbegreper er implementert på et høyt nivå, og diskutere hva som ligger bak designvalg som er gjort i så måte
- Forstå forskjellene på forskjellige paradigmer innen programmering, slik som objektorientert, funksjonell og logisk programmering
- Vurdere fordeler og ulemper ved de forskjellige måter å programmere på

Fakta om emnet

Studiepoeng
10

Nivå
Bachelor

Undervisning
Høst

Eksamen
Høst

Undervisningsspråk
Norsk

Kontakt

Institutt for informatikk