Dagens tema

- Prøveeksamen
- Flyt-tall
  - Hvordan lagres de?
  - Hvordan regner man med dem?
- Koding
  - Koding av instruksjoner
  - Kodefiler
  - Linking
- 64-bits kode

Prøveeksamen

Det vil arrangert en frivillig prøveeksamen **mandag 27. mai** kl 10.15.

- Eksamen vil være en test på **digital eksamen** på datamaskin med programvare fra *Inspera*.
- Den vil ta to timer; etterpå blir det gjennomgang av
  - oppgaven og løsningsforslag
  - hvordan dere opplevde en digital eksamen
- Alle hjelpemidler, inkludert programvaren på Ifis Linux-maskiner, er tillatt.

Flyt-tall [REB&DRO'H 2.4]

Tall med desimalkomma kan skrives på mange måter:

$$8\,388\,708{,}0$$
$$8{,}388708 \cdot 10^6$$
$$8{,}39 \cdot 10^6$$

De to siste $(\pm M \cdot G^E)$ er såkalte **flyt-tall** og består av

- Mantisse («significand») (M).
- Grunntall («radix») (G).
- Eksponent (E).
- Fortegn.

Her lagrer man *selve tallet* og *størrelsen* hver for seg.

Fordelen er at man alltid har like mange tellende sifre.

Representasjon av mantissen

En desimalbrøk: 3,14159265 har **desimaler**.

En binærbrøk: 11,0010010 har **binærer**.

Brøken tolkes slik:

$$
\begin{array}{cccccccccc}
2 & 1 & & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} & \frac{1}{32} & \frac{1}{64} & \frac{1}{128} \\
\downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
1 & 1 & , & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

Resultatet er

$$2^1 + 2^0 + 2^{-3} + 2^{-6} = 2 + 1 + \frac{1}{8} + \frac{1}{64} \approx 3{,}1406$$

En **normalisert** mantisse er en binærbrøk med følgende egenskap:

$$1 \leq M < G$$

For binær representasjon innebærer dette at

$$1 \leq M < 2$$

Det er altså alltid ett binært siffer foran binær-kommaet og det vil alltid være **1** (med mindre hele tallet er 0).

Eksponenten

Eksponenten lagres normalt med et fast tillegg slik at vi alltid får et positivt tall.

(Da kan vi bruke heltallssammenligning av positive flyt-tall.)

Grunntallet

Grunntallet er nesten alltid 2. Blir ikke lagret.

Standarden IEEE 754 for 32-bits flyt-tall

31	30 — 23	22 — 0
S	*Eksponent*	*Mantisse*

S er fortegnet; 0 for positivt, 1 for negativt.

Grunntallet er 2.

Eksponenten er på 8 bit og lagres med fast tillegg 127.

Mantissen er helst normalisert og på 24 bit, men kun de 23 etter binærkommaet lagres.

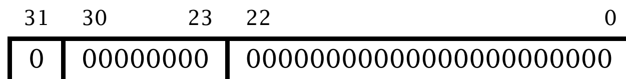
Hvorledes lagres 1,0?

$1{,}0_{10} = 1{,}0_2$ som er normalisert.
Eksponent er $0+127=127=1111111_2$.
Fortegnet er 0.

31	30 ... 23	22 ... 0
0	01111111	00000000000000000000000

Hvordan lagres 0?

Som spesialkonvensjon er 0 representert av kun 0-bit:

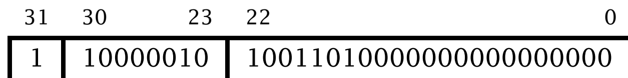


Hvorledes lagres –12,8125?

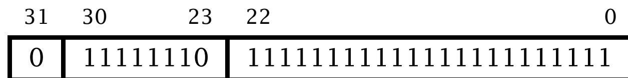
$12{,}8125_{10} = 1100{,}1101_2 = 1{,}1001101_2 \times 2^3$

Eksponent er $3+127=130=10000010_2$.

Fortegnet er 1.



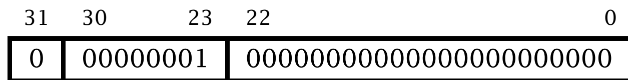
Største tall



er omtrent $2^{254-127} \times 2 \approx 3{,}4 \cdot 10^{38}$.

(Eksponenten 0 er reservert for unormaliserte tall og tallet 0, eksponenten 255 for ∞ og **NAN** «*not a number*».)

Minste normaliserte positive tall



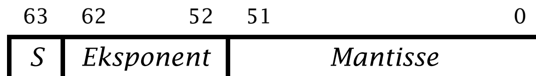
er omtrent $2^{1-127} \times 1 \approx 1{,}2 \cdot 10^{-38}$.

Nøyaktighet

Mantissen er på 24 bit, og $2^{24} \approx 1{,}7 \cdot 10^7$.

Dette gir 7 desimale sifre.

Standarden IEEE 754 for 64-bits flyt-tall



Endringer:

- Eksponenten er økt fra 8 til 11 bit. Lagres med fast tillegg 1023.
- Mantissen er økt fra 24 til 53 bit. Øverste bit lagres stadig ikke.

Største tall

Det største tallet som kan lagres, finner vi utfra formelen

$$2^{(2^{11}-2)-1023} \times 2 = 2^{1023} \times 2 \approx 1{,}8 \cdot 10^{308}$$

Minste positive normaliserte tall

$$2^{1-1023} \times 1 = 2^{-1022} \times 1 \approx 2{,}2 \cdot 10^{-308}$$

Nøyaktighet

Mantissen er på 53 bit, og $2^{53} \approx 9{,}0 \cdot 10^{15}$.

Dette gir nesten 16 desimale sifre.

Flyt-tall er vanskelige

Flyt-tall er oftest bare en tilnærmet verdi; dette kan lett gi uventede feil, spesielt ved subtraksjon.

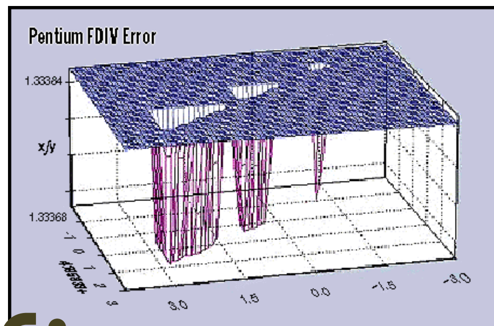
$$(1.1 + \frac{1}{i} - 1.1) \times i$$

```
#include <stdio.h>

int main (void)
{
  int i;

  for (i = 1;  i <= 1000000000;  i *= 10) {
    float v1 = 1.1 + (1.0/i);
    printf("%f\n", (v1-1.1)*i);
  }
  return 0;
}
```

```
1.000000
1.000000
1.000001
0.999951
1.000404
1.003742
0.977516
1.430511
2.384186
23.841858
```

Et annet eksempel

I 1994 kom Intel Pentium. Den hadde en ny algoritme med tabelloppslag som skulle forbedre ytelsen til det 3-dobbelte for flyt-tallsdivisjon. Dessverre ble 5 av 1066 verdier i tabellen uteglemt, og dette ga av og til en feil i 6. desimal:



Q: How many Pentium designers does it take to screw in a light bulb?

A: 1.99904274017

Å regne med flyt-tall

X86 har en egen flyt-tallsprosessor x87:

- Den har egne registre **ST(0)–ST(7)** som brukes som en stakk; de inneholder double-verdier.[1]

  ST(0) (ofte bare kalt **ST**) er toppen.
- Den har egne instruksjoner.
- Den har egne flagg *C0–C5*.
- Parametre overføres på stakken (som vanlig).
- Returverdi fra funksjon legges i **ST(0)**.

[1] Egentlig lagrer de 80-bits flyt-tall på et eget format.

Konstanter

```
fldz                # Dytter 0.0 på stakken.
fld1                # Dytter 1.0 på stakken.
```

Lese fra minnet/stakken

```
flds     var        # Dytter float var på stakken
fldl     var        # Dytter double var på stakken
fld      %st(1)     # Dytter kopi av ST(x) på stakken
```

Skrive til minnet/stakken

```
fsts     var        # Skriver ST(0) til var som float
fstl     var        # Skriver ST(0) til var som double
fst      %st(4)     # Kopierer ST(0) til ST(x)

fstps    var        # Som instruksjonene over,
fstpl    var        # men popper stakken etterpå.
fstp     %st(5)     #
```

Konvertering

X87 kan konvertere mellom heltall og flyt-tall:

```
fildsivar     # Dytter short var på stakken.
fildl   ivar     # Dytter long var på stakken.
fildq   ivar     # Dytter long long var på stakken.

fists   ivar     # Skriver ST(0) til var som short
fistl   ivar     # Skriver ST(0) til var som long
fistps  ivar     # Popper stakken til var som short
fistpl  ivar     # Popper stakken til var som long
fistpq  ivar     # Popper stakken til var som long long
```

Fortegnsoperasjoner

```
fabs             # Gjør ST(0) positivt
fchs             # Snu fortegnet på ST(0)
```

Aritmetiske operasjoner

```
fadds   var     # ST(0) += float var
faddl   var     # ST(0) += double var
fadd    %st(4)  # ST(0) += ST(x)
faddp           # ST(1) = ST(0)+ST(1); popp
fiadds  ivar    # ST(0) += short ivar
fiaddl  ivar    # ST(0) += long ivar
```

Tilsvarende operasjoner finnes for subtraksjon, multiplikasjon og divisjon:

```
fsubs   var     # ST(0) -= float var
fmuls   var     # ST(0) *= float var
fdivs   var     # ST(0) /= float var
```

Sammenligninger

```
ftst                # Sammenlign ST(0) med 0.0
fcoms    ivar       # Sammenlign ST(0) med short var
fcoml    ivar       # Sammenlign ST(0) med long var
fcom     %st(7)     # Sammenlign ST(0) med ST(x)
fcom                # Sammenlign ST(0) med ST(1)
fcomps   ivar       # Som de over,
fcompl   ivar       # men popper etterpå.
fcomp    %st(7)     #
fcomp               #
fcompp              # Som fcom men popper to ganger
```

Resultatet havner i flaggene:

$C3 = 1$ om $\mathrm{ST}(0) = op$

$C0 = 1$ om $\mathrm{ST}(0) < op$

Dessverre finnes ingen hopp som sjekker disse flaggene, men vi kan flytte dem over til x86 og teste der. Da havner *C3* i *Z*-flagget og *C0* i *C*-flagget.

```
        .globl  fnotzero
# Navn:         fnotzero.
# Synopsis:     Returnerer x, eller 1.0 om x er null.
# C-signatur:   float fnotzero (float x).

fnotzero:
        pushl   %ebp                # Standard
        movl    %esp,%ebp           # funksjonsstart.

        flds    8(%ebp)             # Dytt x på x87-stakken.
        ftst                        # Test mot 0.0.
        fstsw   %eax                # Overfør x87-flaggene til EAX
        sahf                        # og derfra til x86-flaggene.
        jnz     fn_xit              # Om x er null,
        fstp    %st                 #   popp x og
        fld1                        #   dytt 1.0 på x87-stakken.

fn_xit: popl    %ebp                #
        ret                         # return SP(0).
```

Andre

Det finnes dusinvis av andre instruksjoner, som

```
fsqrt           # ST(0) = sqrt(ST(0))
fyl2xp1         # ST(1) = ST(1)*log2(ST(0)+1.0) ;  popp
```

Koding [REB&DRO'H 3.1–2]

```
        .globl  add, res
        .extern n

add:    pushl   %ebp
        movl    %esp,%ebp

        movl    8(%ebp),%eax
a_x:    addl    n,%eax
        jz      a_x
        movl    %eax,res

        popl    %ebp
        ret

        .data
res:    .long   0
```

```
#include <stdio.h>

extern int add (int v);
extern int res;

int n = 17;

int main (void)
{
  printf("add(1) = %d", add(1));
  printf("  res = %d\n", res);
  return 0;
}
```

Vi lager kode av add.s:

```
$ gcc -m32 -Wa,-a -c add.s >add.list
```

```
GAS LISTING add.s                                page 1


   1                                    .globl  add, res
   2                                    .extern n
   3
   4 0000 55                add:        pushl   %ebp
   5 0001 89E5                          movl    %esp,%ebp
   6
   7 0003 8B4508                        movl    8(%ebp),%eax
   8 0006 03050000          a_x:        addl    n,%eax
   8      0000
   9 000c 74F8                          jz      a_x
  10 000e A3000000                      movl    %eax,res
  10      00
  11
  12 0013 5D                            popl    %ebp
  13 0014 C3                            ret
  14
  15                                    .data
  16 0000 00000000          res:        .long   0

DEFINED SYMBOLS
               add.s:4      .text:0000000000000000 add
               add.s:16     .data:0000000000000000 res
               add.s:8      .text:0000000000000006 a_x

UNDEFINED SYMBOLS
               n
```

Hver x86-instruksjon lagres i 1–15 byte.

pushl %ebp = 55

pushl = 50+*reg*
%ebp = 5

addl n,%eax = 03 05 00 00 00 00

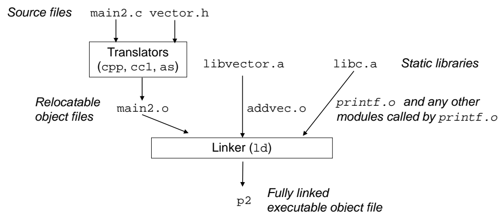
addl = 3
%eax / *minne* = $00\text{-}000\text{-}101_2$ = 5
00000000 = addr(n)

Ellers kan vi legge merke til:

- Et navn angir en adresse som kan være enten kode eller data.
- Navnet a_x er *lokalt*.
- Navnene add og res er *lokalt definert* men *globalt kjent*.
- Navnet n antas å være globalt kjent og definert i en annen kodefil. Adressen er foreløbig ukjent.
- Siden avstanden fra jz til a_x er kjent, er den angitt med *relativ adresse* ($F8_{16} = -8$).

Kompilering/assemblering [REB&DRO'H 7]

- En *oversetter* lager maskinkode i en .o-fil.
- En *linker* lager ferdig eksekverbar kode.



Kodefilen

Kan .o-filen rett og slett være de genererte kode-bytene?

Nei, av (minst) to grunner:

1. Eksterne referanser må kobles sammen.
    1. `add.o` må fortelle at den trenger en `n` fra en annen .o-fil.
    2. Den må også fortelle at den tilbyr `add` og `res` (om noen skulle være interessert).
2. Adressene (til både kode og data) må endres («relokeres»). Derfor må .o-filen opplyse om
    1. Hvilke instruksjoner inneholder adresser som skal endres.

ELF

På Unix-maskiner lagres programkoden i formatet ELF («Executable and Linkable Format»).

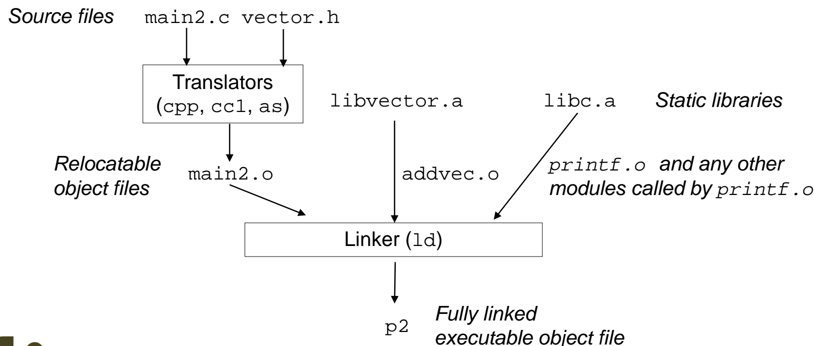
```
$ file add.s
add.s: ASCII text
$ file add.o
add.o: ELF 32-bit LSB relocatable, Intel 80386,
       version 1 (SYSV), not stripped
```

objdump forteller oss om hva filen inneholder:

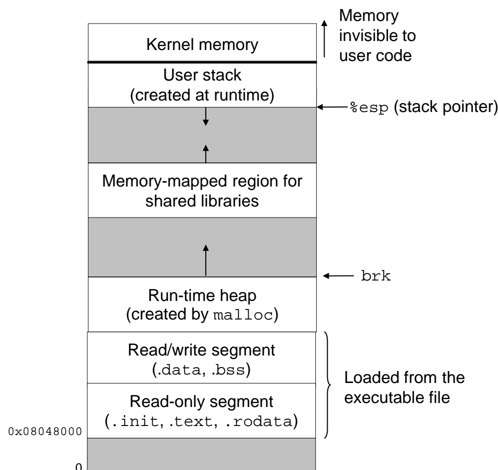
```
$ objdump -d add.o

add.o:     file format elf32-i386

Disassembly of section .text:

00000000 <add>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 45 08                mov    0x8(%ebp),%eax

00000006 <a_x>:
   6:   03 05 00 00 00 00       add    0x0,%eax
   c:   74 f8                   je     6 <a_x>
   e:   a3 00 00 00 00          mov    %eax,0x0
  13:   5d                      pop    %ebp
  14:   c3                      ret
$ objdump -r add.o
add.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_386_32          n
0000000f R_386_32          res
```

Kjørbar kode

Til sist setter linkeren sammen alle kodefilene (*.o) og bibliotekene (*.a) til kjørbar kode. Eksterne referanser kobles sammen og adresser oppdateres.



Det komplette bildet av minnet i Linux-prosesser ser slik ut:



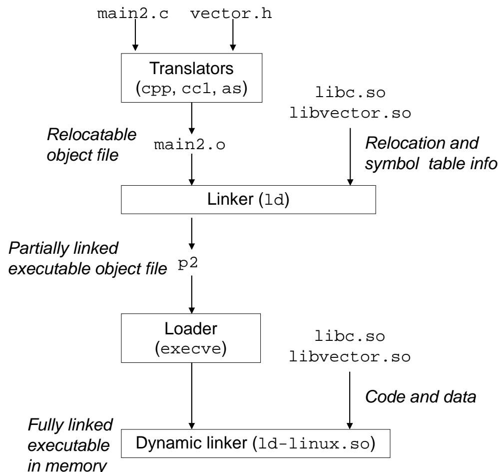
Dynamiske biblioteker

Mye kode er felles i mange prosesser:

- standardbiblioteker
- grafiske biblioteker

Er det mulig å spare plass ved å la prosessene dele kode?

Om man bruker dynamiske biblioteker (.so-filer), vil det bare eksistere én kopi av hver enhet i minnet.



For å få dette til å fungere, må koden i dynamiske biblioteker være

reentrant slik at flere prosesser kan kjøre koden samtidig (dvs ingen variable i faste adresser)

posisjonsuavhengig (= «PIC») slik at den kan plasseres hvor som helst i minnet (dvs kun relative adresser i hopp).

64-bits kode [REB&DRO'H 3.13]

Omtrent alle pc-er i dag kjører en 64-bits kode **x86-64** utviklet av AMD.

- Adresser er 64 bit.
- Flere registrene %R8–%R15, og alle registrene i 64-bits utgave.
- Operasjoner har også 64-bits varianter (som addq).
- Funksjonskall har en annen protokoll (AMD64 ABI for Linux og Mac, Microsoft x64 for Windows).

Hva er nytt?

					%EAX			
							%AX	
%RAX							%AH	%AL
%RBX							%BH	%BL
%RCX							%CH	%CL
%RDX							%DH	%DL

					%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

Funksjonskall

- De seks første parametrene ligger i

  %RDI, %RSI, %RDX, %RCX, %R8, %R9

  Øvrige parametre ligger på stakken.
- Parameterregistrene samt %R10 og %R11 er frie registre, de øvrige er bundne.
- Returverdien skal ligge i %RAX.

Et eksempel

```
        .globl  sum64
# Navn:         sum64.
# Synopsis:     Summerer en array.
# C-signatur:   long sum64 (long a[], long n).
# Registre:     %RDI    a (1. parameter)
#               %RSI    n (2. parameter)

sum64:  movq    $0,%rax         # Initier summen
        movq    %rsi,%rcx       # og telleren.
s64_l:                          # Gå i løkke og
        addq    -8(%rdi,%rcx,8),%rax
        loop    s64_l           # summer.
        ret                     # Ferdig.
```

```
#include <stdio.h>

extern long sum64 (long a[], long n);

long data[] = { 123, 10000, 1000000, 100000000,
                10000000000, 1000000000000 };

int main (void)
{
  int size = sizeof(data)/sizeof(long);

  printf("Svaret er %ld.\n", sum64(data,size));
}
```

```
$ gcc -o test-sum64 test-sum64.c sum64.s
$ ./test-sum64
Svaret er 1010101010123.
```