Dagens tema

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

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:

2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
↓	↓		↓	↓	↓	↓	↓	↓	↓
1	1	,	0	0	1	0	0	1	0

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

$$1 \leq M < 2$$

Binæren foran binær-kommaet vil altså alltid være **1** (med mindre hele tallet er 0).

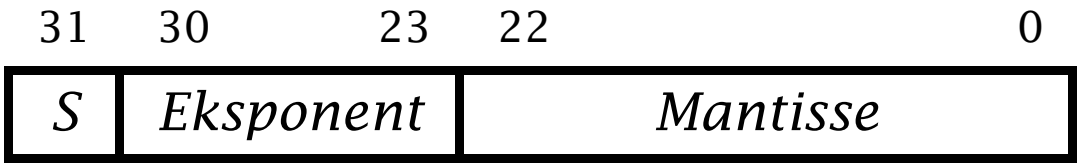
Eksponenten

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

Grunntallet

Grunntallet er nesten alltid 2. Blir ikke lagret.

Standarden IEEE 754 for 32-bits flyt-tall



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

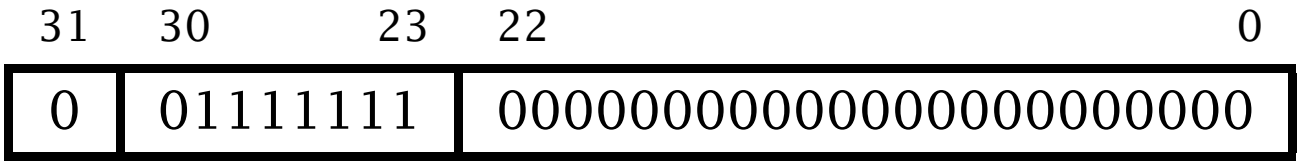
Grunntallet er 2.

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.

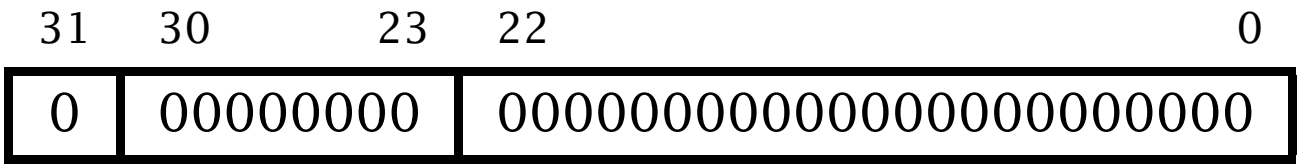
Hvorledes lagres 1,0?

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



Hvordan lagres 0?

Som spesialkonvensjon er 0 representert av kun 0-bit:

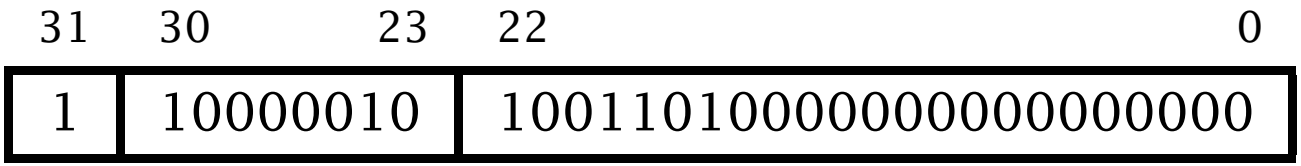


Hvorledes lagres -12,8125?

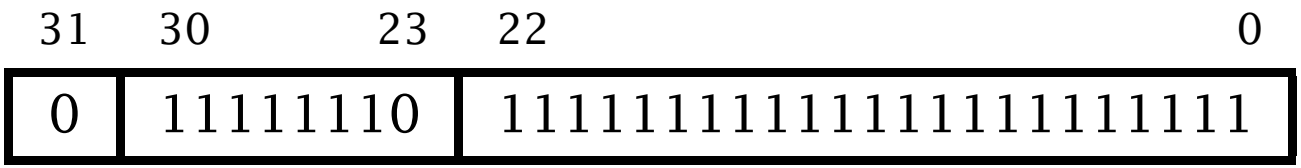
$12{,}8125_{\text{dec}} = 1100{,}1101_{\text{bin}} = 1{,}1001101_{\text{bin}} \times 2^3$

Eksponent er 3+127=130=10000010_{bin}.

Fortegnet er 1.



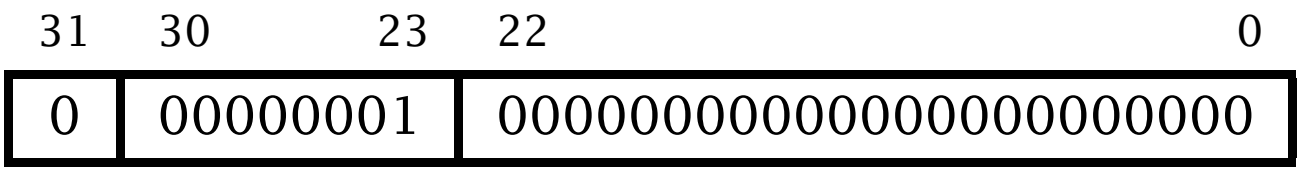
Største tall



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



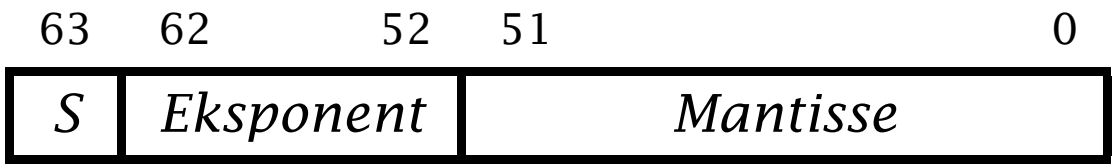
er omtrent $2^{1-127} \times 1 \approx 1{,}2 \cdot 10^{-38}$.

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.

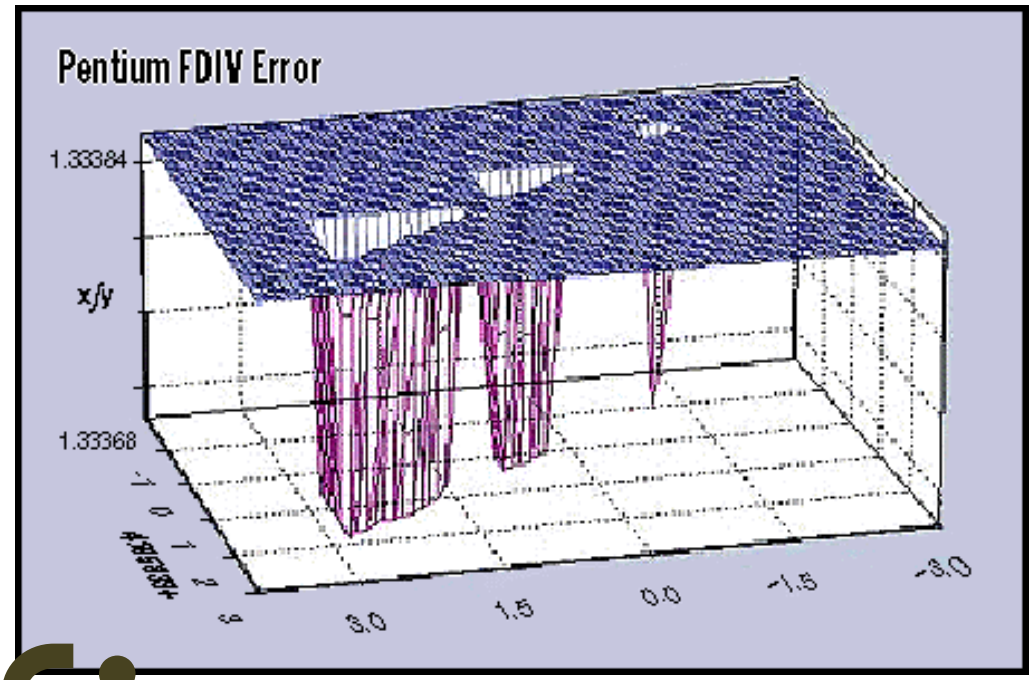
```
#include <stdio.h>

int main (void)
{
  float v1 = 1.1, vd, v2, vx, fmul = 10.0;
  int i;

  for (i = 1;  i <= 8;  ++i) {
    vd = 1.0/fmul;  v2 = v1+vd;  vx = v2-v1;
    printf("%f %f %f\n", v1, v2, vx*fmul);
    fmul = fmul*10.0;
  }
  return 0;
}
```

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

```
1.100000 1.200000 1.000000
1.100000 1.110000 0.999999
1.100000 1.101000 1.000047
1.100000 1.100100 1.000166
1.100000 1.100010 1.001358
1.100000 1.100001 0.953674
1.100000 1.100000 1.192093
1.100000 1.100000 0.000000
```

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: Why didn't Intel call the Pentium the 586?

A: Because they computed 486+100 on the first Pentium and got 585.999983605.

En designsvakhet i Intel Pentium?

```
#include <stdio.h>

int main (void)
{
  long ia;
  float xa = 1.0E-30, xb;

  printf("Test %g / 10.0\n", xa);
  for (ia = 0;  ia < 100000000;  ++ia) {
    xb = xa / 10.0;
  }

}
```

0,73 s på en Intel Xeon

```
#include <stdio.h>

int main (void)
{
  long ia;
  float xa = 1.0E-38, xb;

  printf("Test %g / 10.0\n", xa);
  for (ia = 0;  ia < 100000000;  ++ia) {
    xb = xa / 10.0;
  }

}
```

9,99 s på samme maskin

Jeg vil kalle dette en designsvakhet.

(Det er tilsvarende på AMD-prosessorene.)

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

```
flds    var     # Dytter float var på stakken
fldl    var     # Dytter double var på stakken
fld     st1     # Dytter kopi av ST(x) på stakken
```

Skrive til minnet

```
fsts    var     # Skriver ST(0) til var som float
fstl    var     # Skriver ST(0) til var som double
fst     st4     # Kopierer ST(0) til ST(x)

fstps   var     # Som instruksjonene over,
fstpl   var     # men popper stakken etterpå.
fstp    st5     #
```

Konvertering

X87 kan konvertere mellom heltall og flyt-tall:

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

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    st4     # ST(0) += ST(x)
faddp           # ST(1) += ST(0) ; 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    st7     # Sammenlign ST(0) med ST(x)
fcom            # Sammenlign ST(0) med ST(1)
fcomps  ivar    # Som de over,
fcompl  ivar    # men popper etterpå.
fcomp   st7     #
fcomp           #
fcompp          # Som fcom men popper to ganger
```

Resultatet havner i flaggene:

$C3 = 1$ om ST(0) = op

$C0 = 1$ om 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).
```

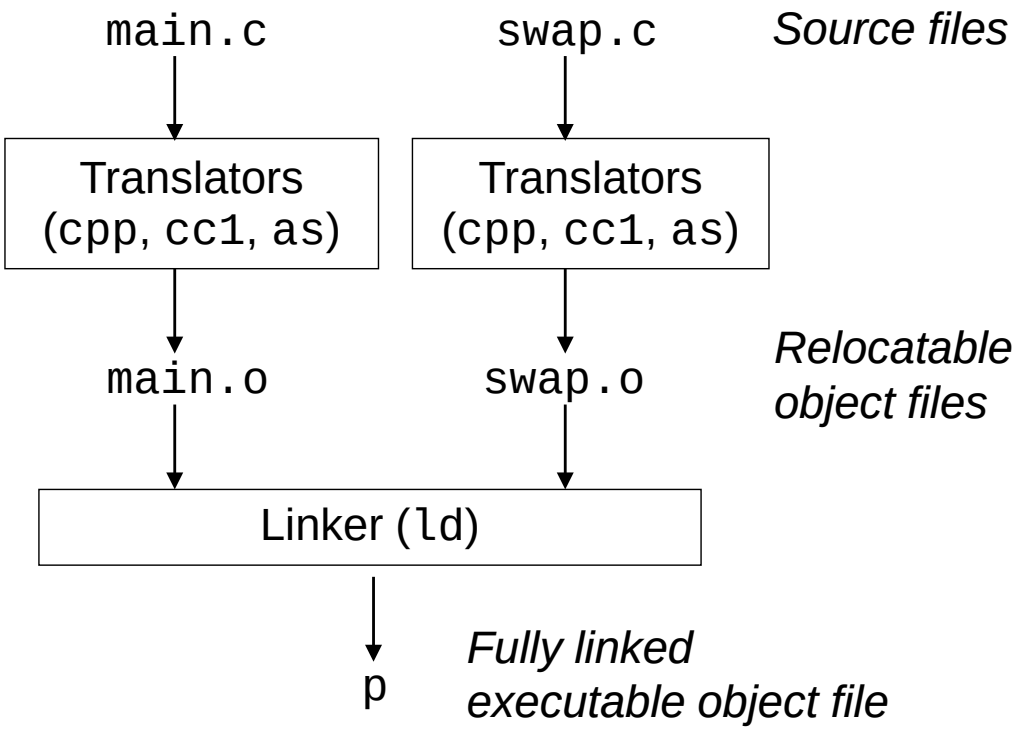
Andre

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]

Vi skal fra C- og assemblerkode til kjørbar kode i maskinen.



Et eksempel

```
        .globl  add, res
        .extern n
add:    pushl   %ebp
        movl    %esp,%ebp

        movl    8(%ebp),%eax
        addl    n,%eax
        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 0000 55            add:    pushl   %ebp
   4 0001 89E5                  movl    %esp,%ebp
   5
   6 0003 8B4508                movl    8(%ebp),%eax
   7 0006 03050000              addl    n,%eax
   7      0000
   8 000c A3000000              movl    %eax,res
   8      00
   9
  10 0011 5D                    popl    %ebp
  11 0012 C3                    ret
  12
  13                            .data
  14 0000 00000000      res:    .long   0

DEFINED SYMBOLS
               add.s:3      .text:0000000000000000 add
               add.s:14     .data:0000000000000000 res

UNDEFINED SYMBOLS
n
```

Hver x86-konstruksjon lagres i 1–15 byte.

pushl %ebp = 55

pushl = 50+*reg*

%ebp = 5

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

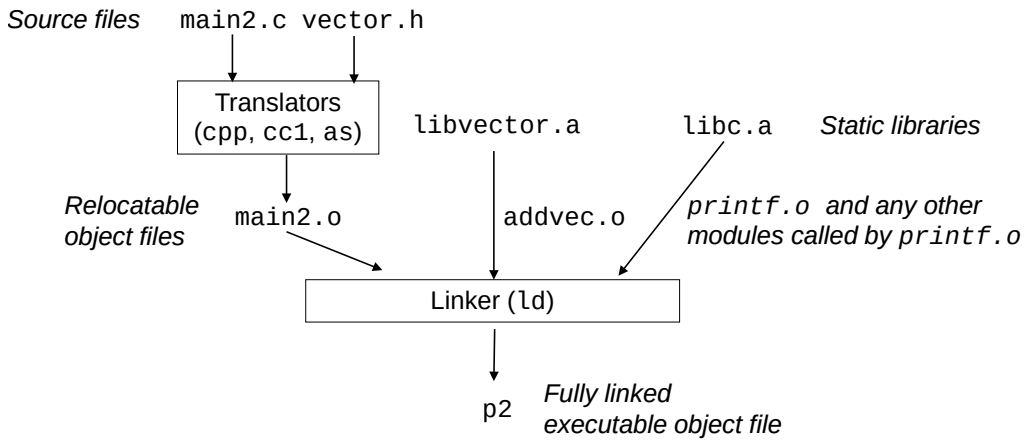
addl = 3

%eax / *minne* = $00\text{-}000\text{-}101_{\text{bin}} = 5$

00000000 = addr(n)

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

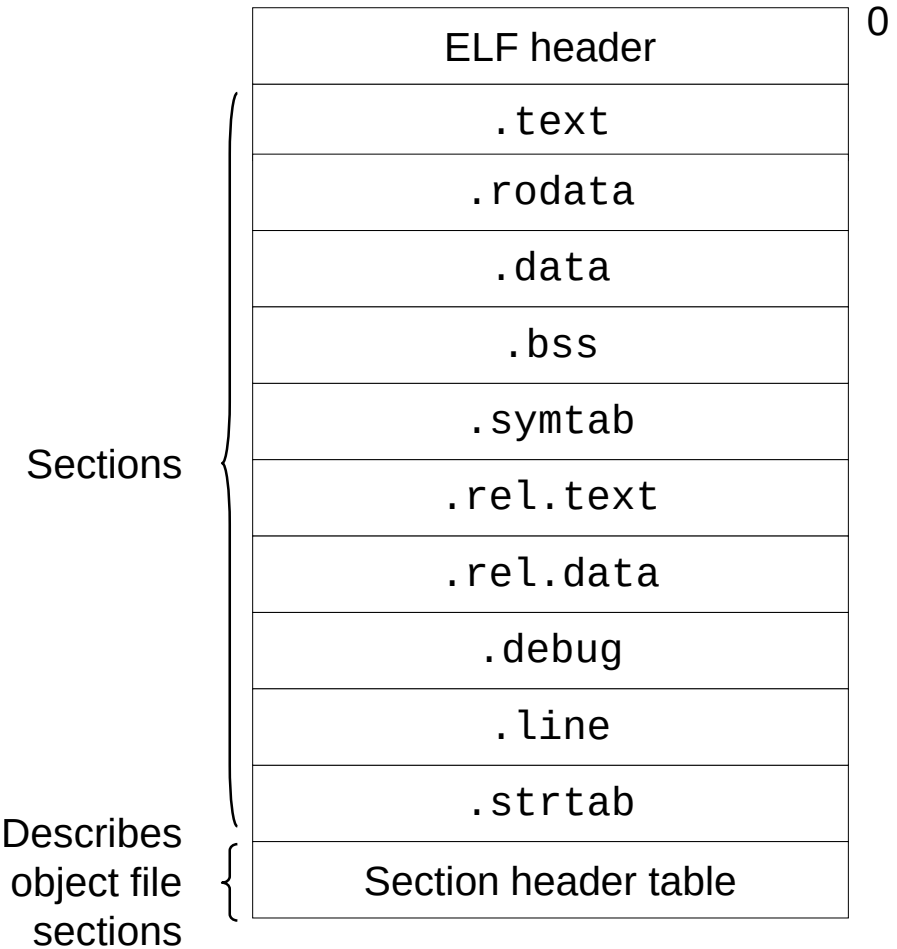
På Unix-maskiner lagres programkoden i formatet ELF («Executable and Linkable Format»).

```
$ 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:

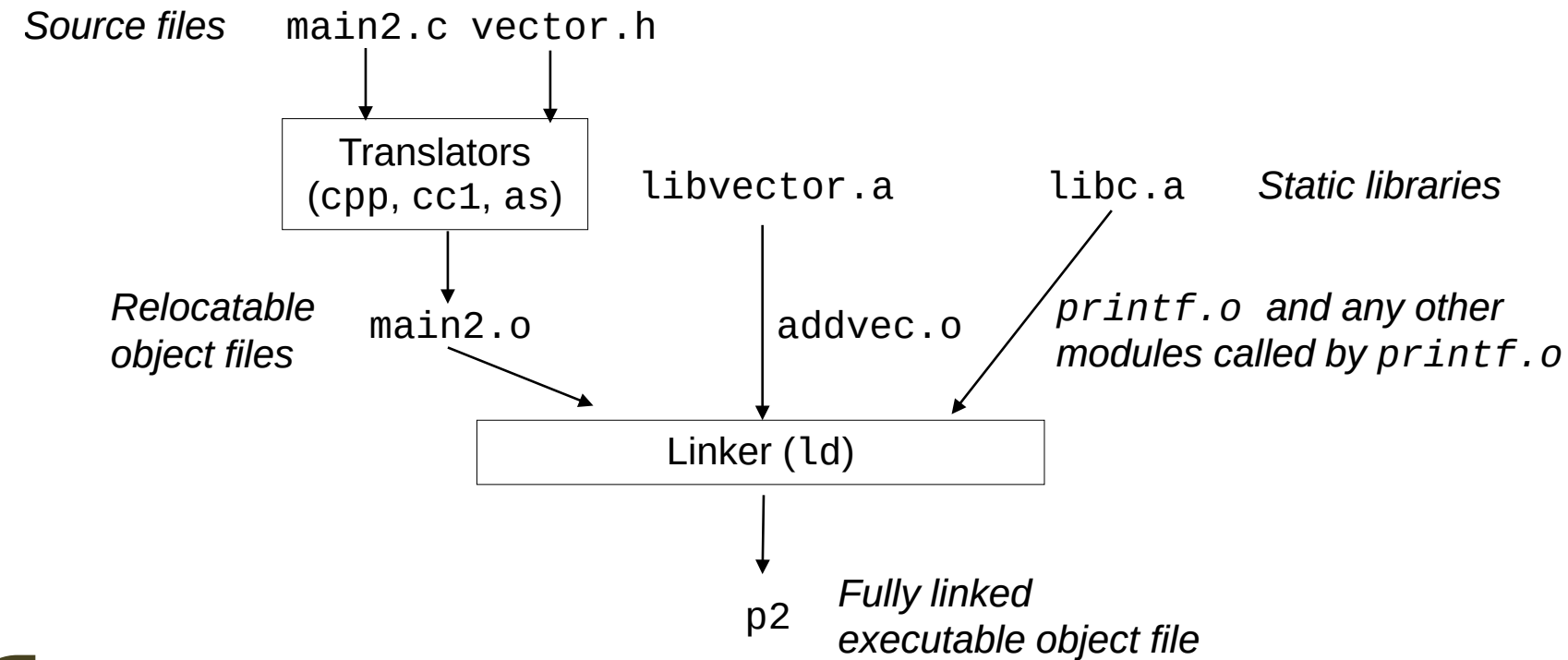
```
$ objdump -d add.o

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
   6:   03 05 00 00 00 00       add    0x0,%eax
   c:   a3 00 00 00 00          mov    %eax,0x0
  11:   5d                      pop    %ebp
  12:   c3                      ret

$ objdump -r add.o
add.o:     file format elf32-i386

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

ELF-filer inneholder all nødvendig informasjon om koden.

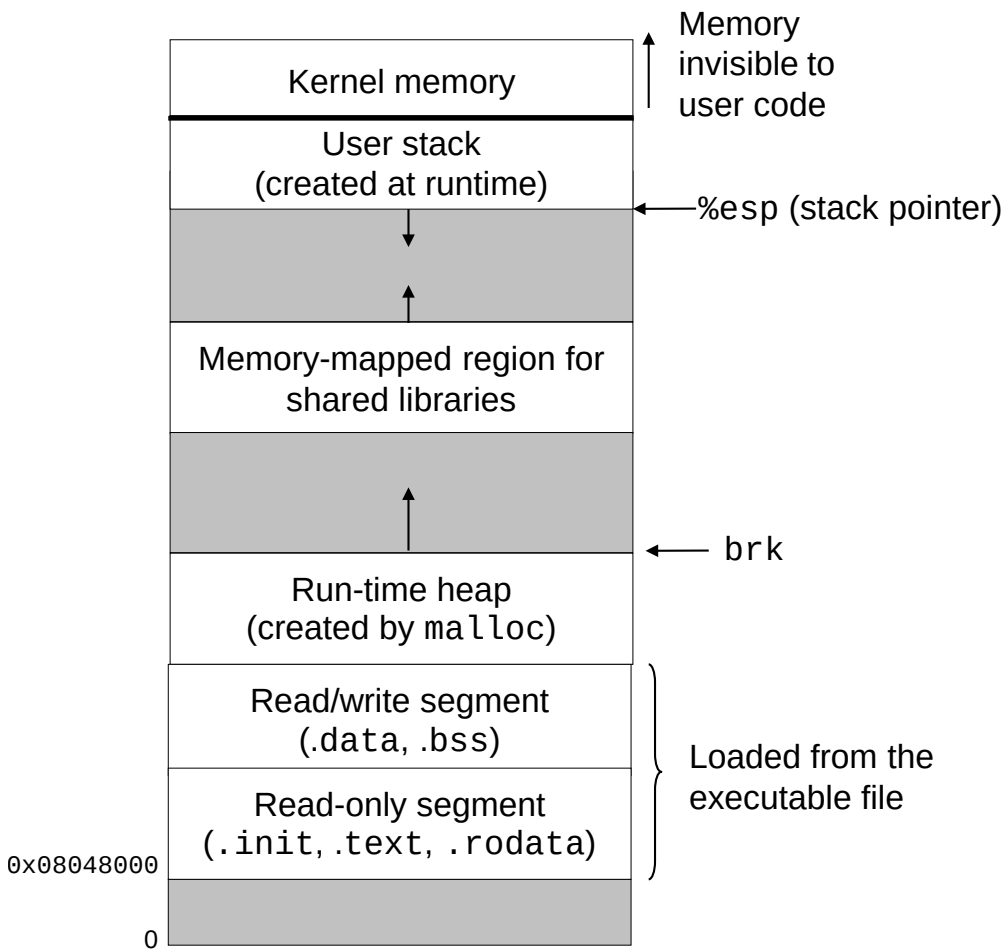


Kjørbar kode

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:



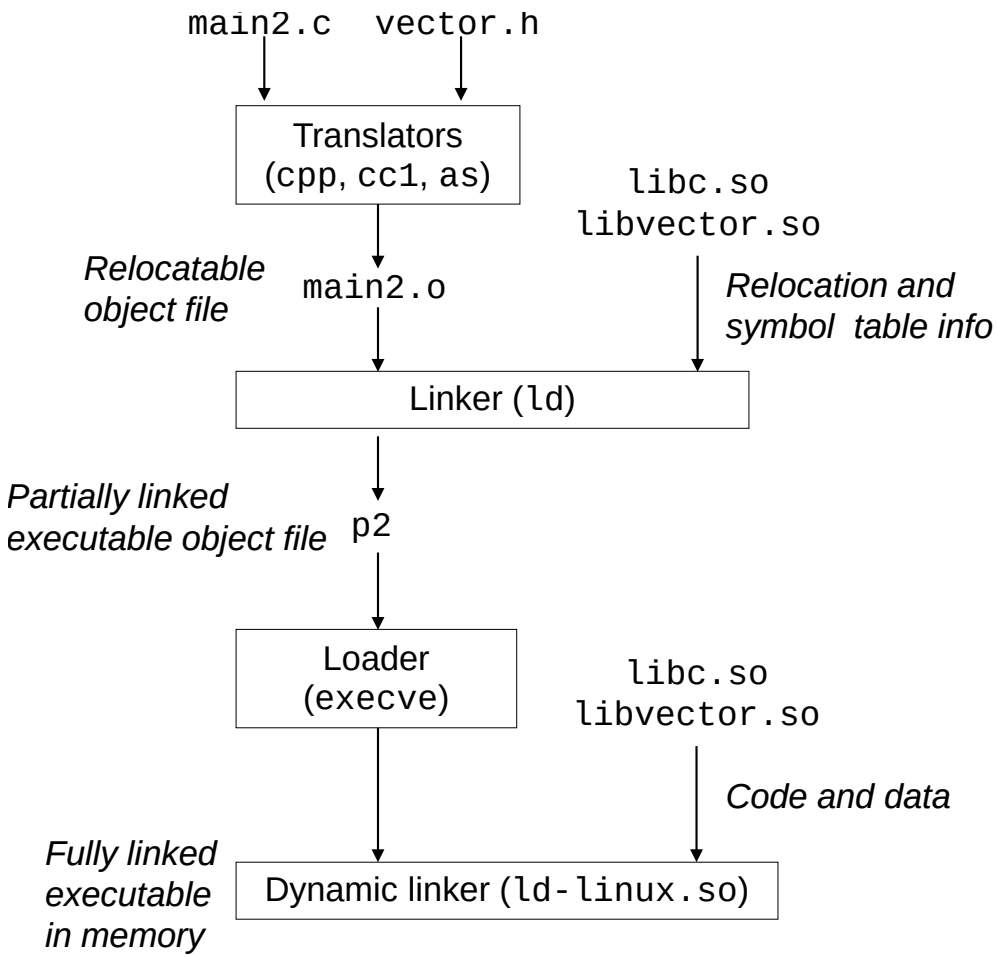
Dynamiske biblioteker

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 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).