



Dagens tema

- Flyt-tall
 - Oppbygning
 - IEEE 754
 - Programmering med flyt-tall
- Selvmodifiserende kode
 - Core War

Flyt-tall

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

$$1 \leq M < 2$$

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

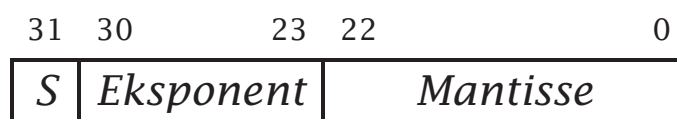
Eksponenten

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.

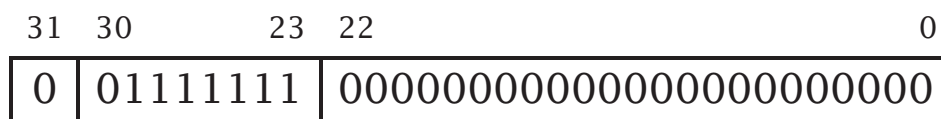
Mantissen er helst normalisert og på 24 bit, men kun de 23 etter binærkommaet lagres.

Hvorledes lagres 1,0?

$1,0_{10} = 1,0_2$ som er normalisert.

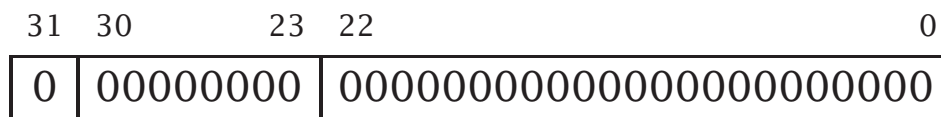
Eksponent er $0+127=127=1111111_2$.

Fortegnet er 0.



Hvordan lagres 0?

Som spesialkonvensjon er 0 representert av kun 0-bit:



Hvorledes lagres -12,8125?

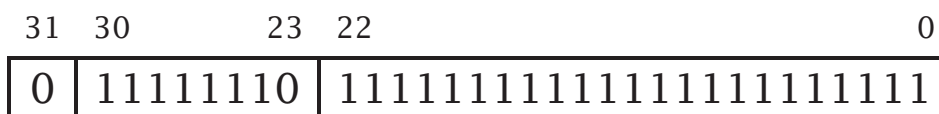
$12,8125_{10} = 1100,1101_2 = 1,1001101_2 \times 2^3$

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

Fortegnet er 1.

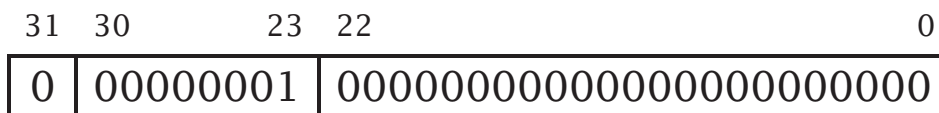


Største tall



omtrent $2^{254-127} \times 2 \approx 3,4 \cdot 10^{38}$.
(Eksponenten 0 er reservert for tallet 0, eksponenten 255 for **NAN**, «*not a number*».)

Minste normaliserte positive tall

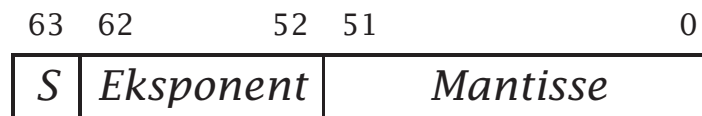


omtrent $2^{1-127} \times 1 \approx 1,2 \cdot 10^{-38}$.

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.

```
#include <stdio.h>

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

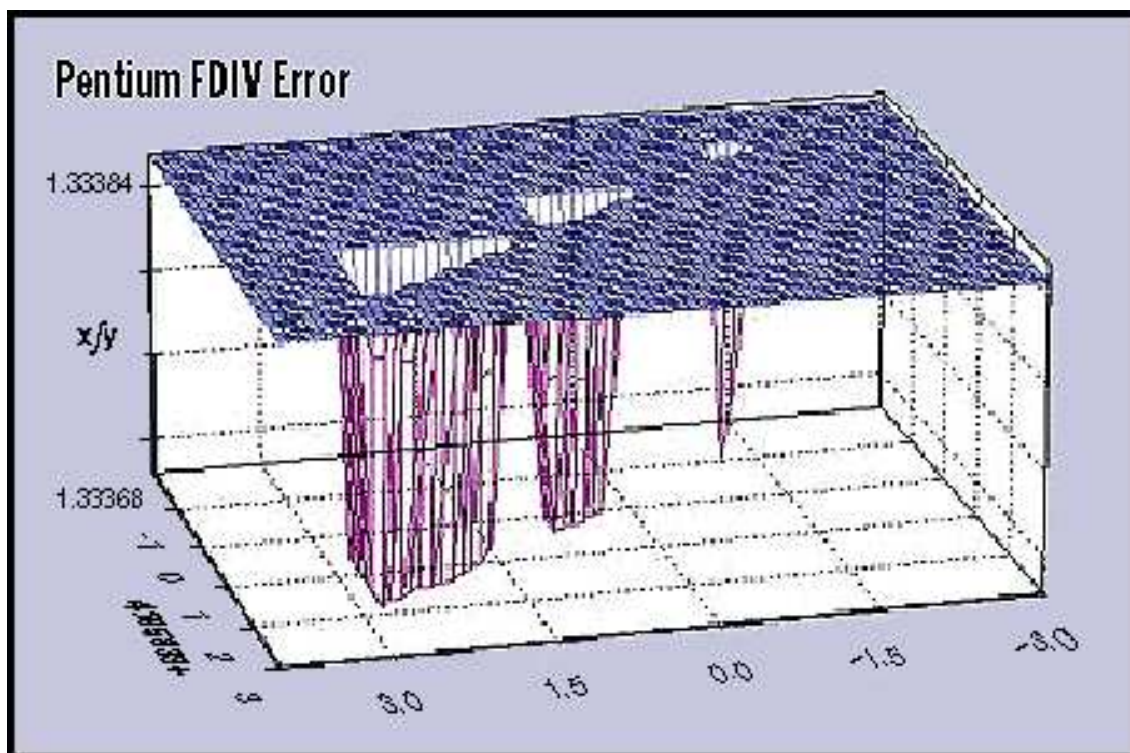
    for (i = 1; i <= 8; ++i) {
        vd = 1.0/fmul; v2 = v1+vd; vx = (v2-v1);
        printf("%f %f %f\n", v1, v2, vx*fmul);
        fmul = fmul*10.0;
    }
    return 0;
}
```

gir følgende resultat:

```
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:



En designsvakhet i Intel Pentium?

Dette lille programmet kjører på 1,59 s på en Intel Pentium 4 med 2,60GHz:

```
#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;
    }
}
```

mens dette bruker 58,22 s:

```
#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;
    }
}
```

Jeg vil kalle dette en designsvakhet i Intel Pentium.

Å regne med flyt-tall

X86 har en egen flyt-tallsprosessor x87:

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

ST(0) (ofte bare kalt **ST**) er toppen.

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

[†] 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 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   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
fmu1s  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:

$C(3) = 1$ om $ST(0) = op$

$C(0) = 1$ 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 C(3) i Z-flagget og C(0) 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
    fldl                    #   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
```

Bit-mønstre

Husk:

*Alt som finnes i datamaskinen er
bit-mønstre!*

Det er opp til programmereren å la datamaskinen tolke dem på riktig måte.

Eksempel

En byte med innholdet 195 = 0xCE kan være

- Verdien 195
- Verdien -61
- En del av et 16-bits, 32-bits eller 64-bit heltall (med eller uten fortegns-bit)
- En del av et 32-bits eller 64-bits flyt-tall
- Tegnet Å
- En del av en tekst
- Instruksjonen ret
- En del av en fler-bytes instruksjon

for ikke å snakke om alle mulige typer data håndtert av et program.

Selvmodifiserende kode

Når programkode lagres som bit-mønstre, kan man da la programmet endre på seg selv?

```
1          .globl teller
2          .data
3 0000 55          teller: pushl   %ebp
4 0001 89E5        movl    %esp,%ebp
5
6 0003 B8010000    movl    $1,%eax
6          00
7 0008 83050400    addl   $1,teller+4
7          000001
8
9 000f 5D          popl   %ebp
10 0010 C3          ret
```

Denne funksjonen returnerer 1 første gang den kalles. Samtidig endres instruksjonen slik at den vil gi 2 neste gang den utføres.

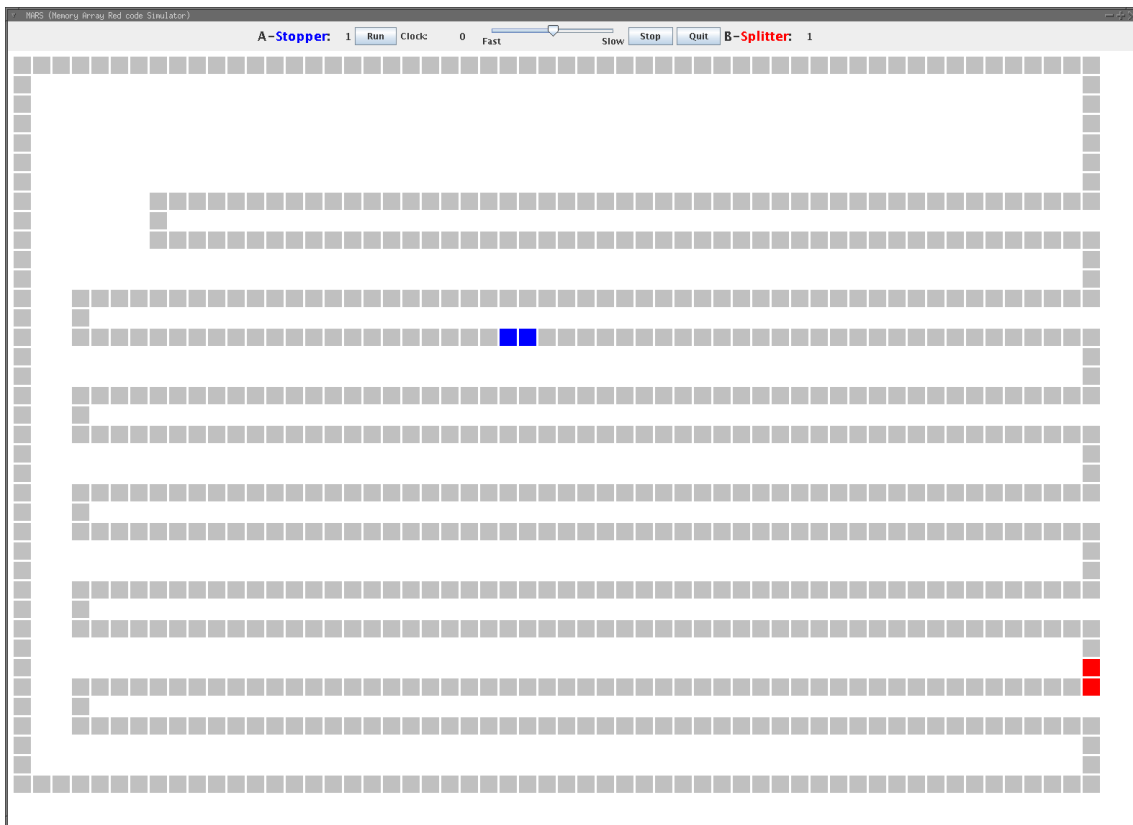
- Koden er plassert i `.data` for å kunne endres.
- På noen maskiner vil det kunne bli rot med data- og instruksjons-cache.

Core War

For å illustrere slevmodifiserende kode, skal vi bruke simulatoren/spillet *CoreWar*.

- Opprinnelig laget D G Jones og A K Dewdney i 1984.
- Kjent gjennom tre artikler av sistnevnte i *Scientific American*
- Voldsomt populært i de neste 10 årene med artikler og VM; i dag nesten glemt.
- Tre standarder: 86, 88 og 94. (Vi skal holde oss til 88.)

CoreWar spilles på en idealisert datamaskin med 800 celler:



To spillere skriver hvert sitt lille assemblerprogram som plasseres tilfeldig i maskinen MARS.

- Programmene bytter på å utføres.
- Det første som dør, har tapt. Et program dør når det prøver å gjøre noe ulovlig:
 - dividere med 0
 - prøve å utføre data som instruksjon.
- Spillet går ut på å vinne ved å
 - unngå å tabbe seg ut,
 - ødelegge motstanderens program og
 - forsvare seg mot motstanderens angrep.

Ekte datamaskiner?

CoreWar-maskinene og -koden ligner på ekte datamaskiner, men

- minnet er mye mindre.
- instruksjonssettet er mye mindre.
- vanlige datamaskiner skiller ikke på instruksjoner og data.

Og så er det ikke forskjell på store og små bokstaver.

To små programmer

Ett minimalt program er **Blind** som ligger på filen Blind.red:

```
; The 'blind' program:  
; (It just steps onto an illegal instruction!)  
  
Blind    nop          Blind  
         end
```

Det inneholder

- ① to kommentarlinjer
- ② en instruksjon NOP som ikke gjør noe
- ③ et direktiv END som markerer slutten

Programmet **Static** blir bare stående på samme sted:

```
; The 'static' program:  
; (It never moves!)  
  
Static: jmp          Static  
         end         Static
```

Kjøring

Siden hele lageret (unntatt de to programmene) er fylt med data:

DAT #0

er Blind dømt til å tape.

Det mest kjente CoreWar-programmet

Programmet Imp flytter seg selv til neste instruksjon:

```
; The famous 'imp' program:  
Imp      mov      Imp, Next  
Next  
          end      Imp
```

Hvordan er det mulig? Blir ikke adressen i instruksjon da gal?

I CoreWar skjer all adressering *relativt*, dvs at adresse 0 alltid er den instruksjonen som utføres. Imp kan derfor også skrives

```
; The famous 'imp' program:  
          mov      0, 1
```

Hva skjer om en Imp tar igjen en Static?

Den vil skrive over koden til Static slik at Static bli en Imp.

Hittil har vi ikke gjort noe for å vinne.

Programmet **Stopper** prøver å drepe en Imp ved å kaste DAT-er bak seg.

```
; Program 'stopper'  
; (Tries to stop an Imp.)  
  
Stopper mov    Next,Stopper-1  
          jmp    Stopper  
Next      end    Stopper
```

(Instruksjonen MOV kopierer en celle til et annet sted.)

Dette kan gå bra, men ikke alltid.

Instruksjonene i CoreWar

Alle instruksjonene (og DAT) består av fem felt:

| | | | | |
|-------------|-------|-----------|-------|-----------|
| Instruksjon | M_A | A-adresse | M_B | B-adresse |
|-------------|-------|-----------|-------|-----------|

Modifikatoren M_x kan være

tom vanlig adresse

konstant

@ *indirekte adressering* (tilsvarer * i C)

< indirekte adressering med *dekrementering*
(tilsvarer *--p i C)

Instruksjonene i CoreWar

Her er *alle* instruksjonene i CoreWar88:

| | |
|-----|---|
| dat | data; ulovlig instruksjon |
| nop | gjør ingenting |
| mov | flytt data |
| add | + |
| sub | - |
| mul | × |
| div | ÷ |
| mod | % (finn rest ved divisjon) |
| jmp | hopp |
| jmz | hopp om = 0 |
| jmn | hopp om \neq 0 |
| djz | --; hopp så om \neq 0 |
| seq | hopp over neste instr om $p_1 = p_2$ |
| sne | hopp over neste instr om $p_1 \neq p_2$ |
| slt | hopp over neste instr om $p_1 < p_2$ |
| spl | splitt (tas neste forelesning) |

En bombekaster

Programmet **Dwarf** kaster DAT-er utover minnet:

```
; The 'dwarf'  
; (Throws bombs around.)  
  
Dwarf   add    #4, Data  
        mov    Data, @Data  
        jmp    Dwarf  
Data    dat    #0, #0
```

Den vil ofte slå en Imp.