UNIVERSITETET I OSLO
DET MATEMATISK-NATURVITENSKAPELIGE FAKULTET

Dagens tema

- Flyt-tall (B&O'H-boken 2.4, 3.14)
  - Hvordan lagres de?
  - Hvordan regner man med dem?
- Bit-fikling (B&O'H-boken 2.1.7)
  - Skifting (B&O'H-boken 3.5.3–4)
  - Pakking
  - Instruksjoner for enkelt-bit

INF2270

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.

INF2270

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

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

INF2270

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

$$1 \leqslant M < G$$

For binær representasjon innebærer dette at

$$1 \leqslant M < 2$$

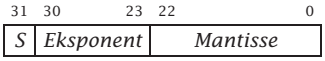
Binæren foran binær-kommaet vil altså alltid være **1** (med mindre hele tallet er 0).

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.

INF2270

Standarden IEEE 754 for 32-bits flyt-tall



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

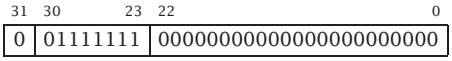
Grunntallet er 2.

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

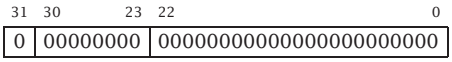
Mantissen er helst normalisert og på 24 bit, men kun de 23 etter binærkommaet lagres.

INF2270

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



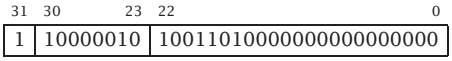
Hvordan lagres 0?
Som spesialkonvensjon er 0 representert av kun 0-bit:



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

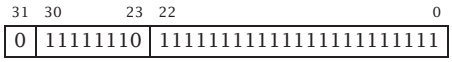
Eksponent er 3+127=130=10000010_2.

Fortegnet er 1.



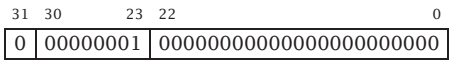
INF2270

Største tall



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

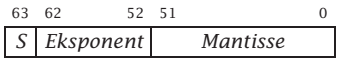


omtrent $2^{1-127} \times 1 \approx 1{,}2 \cdot 10^{-38}$.

Nøyaktighet
Mantissen er på 24 bit, og $2^{24} \approx 1{,}7 \cdot 10^7$.
Dette gir 7 desimale sifre.

INF2270

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.

INF2270

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;
}
```

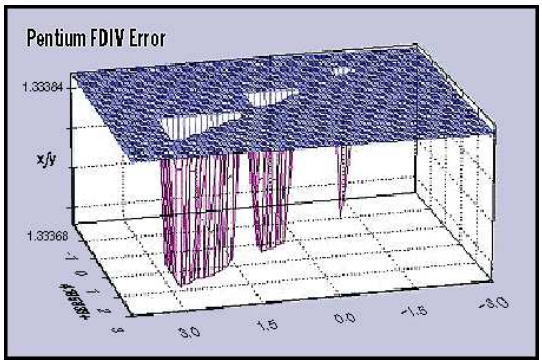
gir følgende resultat:

```
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
```

INF2270

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:



INF2270

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.

(Det er nesten tilsvarende på en AMD Athlon 3500+ prosessor på 2,2 GHz: 0,97 s og 10,40 s.)

INF2270

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.

INF2270

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
```

INF2270

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     #
```

INF2270

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*

INF2270

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)
```

INF2270

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
        fld1                    #   dytt 1.0 på x87-stakken.

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

INF2270

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
```

INF2270

Bit-fikling

Når alt er bit, gir det oss som programmerere nye muligheter.

Er maskinen big-endian?

Denne funksjonen tester det:

```
        .globl bigendian
# Navn:          bigendian
# Synopsis:      Er denne maskinen big-endian?
# C-signatur:    int bigendian (void)
# Registre:      EAX - test-byte or resultat

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

        movb    endian+3,%al    # Hent «siste» byte av 1
        andl    $1,%eax         # og test det.
                                # (Og null ut resten av EAX.)

        popl    %ebp            # Standard
        ret                     # retur.

        .data
endian: .fill   1               # 0,0,0,1 eller 1,0,0,0
```

INF2270

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 Ã
- 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.

INF2270

Hvordan lagres flyt-tall?
Vi kan bruke assemblerkode til å flytte innholde av en float til en byte-vektor og dermed unngå typereglene i høynivåspråk.

```
        .globl float2byte
        float2byte
# Navn:
# Synopsis:     Viser hvordan en float lagres i 4 byte
# C-signatur:   void float2byte (float f, unsigned char b[])
# Registre:     EAX - f
#               EDX - b (dvs adressen)
#
float2byte:     pushl   %ebp            # Standard
                movl    %esp,%ebp       # funksjonsstart.

                movl    8(%ebp),%eax    #       f
                movl    12(%ebp),%edx   # *b =   /* uten konvertering */
                movl    %eax,(%edx)

                popl    %ebp            # Standard
                ret                     # retur.
```



```
#include <stdio.h>

typedef unsigned char byte;

extern int bigendian (void);
extern void float2byte(float f, byte b[]);

void test (float f)
{
  byte b[4];

  float2byte(f, b);
  if (bigendian())
    printf("%10.3f lagres som %02x %02x %02x %02x\n",
           f, b[0], b[1], b[2], b[3]);
  else
    printf("%10.3f lagres som %02x %02x %02x %02x\n",
           f, b[3], b[2], b[1], b[0]);
}

int main (void)
{
  test(0.0);  test(1.0);  test(-12.8125);
  return 0;
}
```

gir resultatet

```
     0.000 lagres som 00 00 00 00
     1.000 lagres som 3f 80 00 00
   -12.812 lagres som c1 4d 00 00
```


Dette kan vi også gjøre i C ved hjelp av en spesiell konstruksjon:

En **union** plasserer sine elementer *oppå* hverandre.

```
int bigendian (void)
{
  union endian {
    int v;
    unsigned char b[4];
  } e;

  e.v = 1;
  return e.b[3];
}
```

```
void float2byte (float f, unsigned char b[])
{
  union f2b {
    float f;
    unsigned char b[4];
  } u;
  int i;

  u.f = f;
  for (i = 0;  i < 4;  ++i) b[i] = u.b[i];
}
```


I C har vi også mulighet til å omgå typereglene ved å bruke pekere:

```
int bigendian (void)
{
  int v = 1;

  return *((unsigned char*)&v) == 0;
}
```

```
void float2byte (float f, unsigned char b[])
{
  int i;

  for (i = 0;  i < 4;  ++i)
    b[i] = ((unsigned char *)&f)[i];
}
```

Skift-operasjoner

Dette er operasjoner som flytter alle bit-ene i et ord mot høyre eller venstre.

Logisk skift

Her settes det inn 0-er fra enden:

	0	1	0	1	0	1	1	1
salb $1,%al	1	0	1	0	1	1	1	0
salb $2,%al	1	0	1	1	1	0	0	0
shrb $1,%al	0	1	0	1	1	1	0	0
shrb $4,%al	0	0	0	0	0	1	0	1

C-flagget settes til det siste bit-et som «faller utenfor».

INF2270

Aritmetisk skift

I vårt desimale tallsystem kan man gange med 10 ved å sette inn en 0, og dele med 10 ved å fjerne siste siffer:

$$42 \times 10 = 420$$
$$217/10 = 21$$

Det samme gjelder i det binære tallsystemet, men her er effekten å gange med 2 eller dele på 2:

0	0	1	0	1	0	1	0	$(=42_{10})$
0	1	0	1	0	1	0	0	$(=84_{10})$
1	1	0	1	1	0	0	1	$(=217_{10})$
0	1	1	0	1	1	0	0	$(=108_{10})$

INF2270

Hva gjør vi så hvis det er fortegnsbit? Ved skift mot venstre spiller det ingen rolle, men for skift mot høyre er løsningen å kopiere inn fortegnsbit-et.

	0	1	0	1	0	1	1	1	$= 87_{10}$
sarb $1,%al	0	0	1	0	1	0	1	1	$= 43_{10}$
sarb $2,%al	0	0	0	0	1	0	1	0	$= 10_{10}$
	1	1	0	1	0	1	1	1	$= -41_{10}$
sarb $1,%al	1	1	1	0	1	0	1	1	$= -21_{10}$
sarb $2,%al	1	1	1	1	1	0	1	0	$= -6_{10}$

(Legg merke til at negative tall rundes av mot $-\infty$ og ikke mot 0!)

INF2270

Rotasjoner

En variasjon av skifting er at bit-ene som «detter utenfor» kommer tilbake fra den andre siden:

	0	1	0	1	0	1	1	1
rolb $1,%al	1	0	1	0	1	1	1	0
rolb $2,%al	1	0	1	1	1	0	1	0
rorb $1,%al	0	1	0	1	1	1	0	1
rorb $4,%al	1	1	0	1	0	1	0	1

Enda en variant er å ta med **C**-flagget i rotasjonen:

	1	1	0	1	0	1	1	1	1
rclb $1,%al	1	0	1	0	1	1	1	1	1
rclb $2,%al	1	0	1	1	1	1	1	1	0
rcrb $1,%al	0	1	0	1	1	1	1	1	1
rcrb $4,%al	1	1	1	1	0	1	0	1	1

INF2270

Pakking av bit

Noen ganger ønsker vi å pakke flere datafelt inn i ett ord

- for å spare plass
- for å programmere nettverk

Ved hjelp av skifting og masking kan vi hente frem bit-felt:

```
# Navn:         .globl  bit2til4
# Synopsis:             bit2til4
# C-signatur:           Henter bit 2-4.
# Registre:             int bit2til4 (int v)
                        EAX - arbeidsregister
bit2til4:
                pushl   %ebp            # Standard
                movl    %esp,%ebp       # funksjonstart.

                movl    8(%ebp),%eax    # Hent v og
                sarl    $2,%eax         # skift 2 mot høyre.
                andl    $0x7,%eax       # Fjern alt uten
                                        # 3 nederste bit.

                popl    %ebp            # Standard
                ret                     # retur.
```


Dette kan vi også gjøre i C:

```
#include <stdio.h>

struct data {
  unsigned int a: 2;
  unsigned int b: 3;
  unsigned int c:27; } pakket;

extern float2byte (struct data d, unsigned char byte[]);
/* Egentlig er denne funksjonen for float->byte,
   men det vet ikke C-kompilatoren. */

int main (void)
{
  unsigned char b[4];

  pakket.a = 1;  pakket.b = 7;  pakket.c = 0x123456;
  float2byte(pakket, b);
  printf("struct {0x%x; 0x%x; 0x%x;} lagres som %02x %02x %02x %02x\n",
        pakket.a, pakket.b, pakket.c, b[0], b[1], b[2], b[3]);
  return 0;
}
```


Resultatet blir

```
struct {0x1; 0x7; 0x123456;} lagres som dd 8a 46 02
```

som kan tolkes slik:

02	46	8a	dd
00000010	01000110	10001010	11011101

00000010010001101000101011	0	111	01

Vi kan også sette inn bit, men da må vi bruke en **maske**:

```
        .globl  set2til4
# Navn:         set2til4
# Synopsis:     Bytter ut bit 2-4 med gitt verdi.
# C-signatur:   int set2til4 (int orig, int v2til4)
# Registre:     EAX - arbeidsregister

set2til4:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.

        movl    8(%ebp),%eax    # Hent opprinnelig verdi
        andl    $0xfffffe3,%eax # og null ut bit-feltet.
        movl    12(%ebp),%ecx   # Hent ny verdi og sørg
        andl    $0x7,%ecx       # for at den ikke er for stor.
        sall    $2,%ecx         # Skift på plass
        orl     %ecx,%eax       # og sett inn.

        popl    %ebp            # Standard
        ret                     # retur.
```

Enkelt-bit
Det finnes fire operasjoner for å jobbe med enkelt-bit:

- btl gjør ingenting
- btcl snur bit-et
- btrl nuller bit-et
- btsl setter bit-et

Alle kopierer dessuten det opprinnelige bit-et til **C**-flagget.

```
    btl    $2,%eax  # Sjekker bit 2 i EAX.
```

Hele byte
Når vi jobber med hele byte, har vi direkte tilgang til dem.

INF2270