



Dagens tema

- Skifting og rotasjoner (Irvine-boken 7.2)
- Datatyper
 - Heltall (4 til 64 bit) (Irvine-boken 7.5-6)
 - Vektorer (Irvine-boken 4.4)
 - Mengder (Irvine-boken 6.3.5)
 - Tekster (Irvine-boken 9.2)
- Mer om funksjoner
 - Instansblokker (Irvine-boken 8.4)
 - Rekursive funksjoner (Irvine-boken 8.5)
- Tidtaking
 - To måter å gjøre det på

INF1070

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».

INF1070

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})

INF1070

Hva gjør vi så hvis det er fortegnstegn? Ved skift mot venstre spiller det ingen rolle, men for skift mot høyre er løsningen å kopiere inn fortegnstegn-bit-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!)

INF1070

Rotasjoner

En variasjon av shifting er at bit-ene som «dette 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

INF1070

Heltall av ulik størrelse

Vi kjenner hitil til de tre vanligste størrelsene:

Bit	Med fortegn	Uten fortegn
8	-128 til 127	0-255
16	-32768 til 32767	0-65536
32	-2147483648 til 2147483647	0-4294967295

BCD-tall

Noen ganger trenger man «Binary Coded Decimals» (BCD) der hvert desimale siffer lagres som 4 bit (en «nibble»).

14 byte:

$$29 = 001010101011$$
$$12\ 345\ 678 = \begin{array}{|c|c|c|c|} \hline 12_{16} & 34_{16} & 56_{16} & 78_{16} \\ \hline \end{array}$$

INF1070

Regning med BCD-tall

La oss først prøve en vanlig add:

```
movb $0x28,%al # AL = 0x28
addb $0x66,%al # AL = 0x8e
```

Dette er galt, fordi 0xe ikke er et lovlig desimalt siffer.

Heldigvis finnes instruksjonen daa («decimal adjust after addition») som retter opp resultatet:

```
movb $0x28,%al # AL = 0x28
addb $0x66,%al # AL = 0x8e
daa # AL = 0x94
```

Konklusjon: BCD-tall er

- Fleksible med hensyn på størrelsen.
- Rimelig raske å regne med.
- Rimelig kompakte å lagre.
- Raske å skrive ut og lese inn.

De brukes i COBOL-programmer og lommekalkulatorer.

INF1070

Tall lagret som tegn

Det er også mulig å regne med tall lagret som ASCII-tegn. Hvis det er snakk om enkle operasjoner, er dette raskere enn å konvertere til binær form, regne og så konvertere tilbake.

Instruksjonen aaa («ASCII adjust after addition») ordner opp etter en vanlig addisjon; eventuell mente lagres i %AH.

Anta at vi vi addere to 3-sifrede tall på tekstform:

```
movw $0,%ax # Null ut AH.
movb 2(%ecx),%al # Legg sammen tredje
addb 2(%edx),%al # siffer av begge tallene.
aaa # Rett opp svaret.
orb $0x30,%al # Omform til ASCII og
movb %al,2(%ecx) # legg tilbake.

shrw $8,%ax # Flytt menten til AL.
addb 1(%ecx),%al # Legg till andre siffer
addb 1(%edx),%al # fra de to tallene.
aaa # Rett opp svaret.
orb $0x30,%al # Omform til ASCII og
movb %al,1(%ecx) # legg tilbake.

shrw $8,%ax # Flytt menten til AL.
addb 0(%ecx),%al # Legg til første siffer
addb 0(%edx),%al # fra de to tallene.
aaa # Rett opp svaret.
orb $0x30,%al # Omform til ASCII og
movb %al,0(%ecx) # legg tilbake.
```

INF1070

Store tall

Av og til trenger vi ekstra store heltall, for eksempel 64 eller 128 bit. Disse lagres i så mange byte som trengs.

Addisjon (og subtrasjon) er enkelt takket være **C**-flagget som inneholder en *mente* eller et *lån*.

```
movl $0xa0001000,%eax # a0001000
movl $0x12340088,%edx # 12340088
addl $0x60002000,%eax # + 60002000 C=1
adcl $0x11110000,%edx # 11110000
# =2345008900003000
```

INF1070

Vektorer

Det finnes en egen adresseringmåte for å slå opp i en vektor:

$20(\%eax, \%ebx, n)$

som gir adressen

$\%eax + n \times \%ebx + 20$

n må være 1, 2, 4 eller 8.

```
.globl arrayadd
# Navn: arrayadd.
# Synopsis: Summerer verdiene i en vektor.
# C-signatur: int arrayadd (int a[], int n).
# Registre: %eax: summen så langt
#           %ecx: indeks til a (teller nedover)
#           %edx: adressen til a
arrayadd:
    pushl %ebp # Standard
    movl %esp,%ebp # funksjonsstart.

    movl $0,%eax # sum = 0.
    movl 12(%ebp),%ecx # ix = n.
    movl 8(%ebp),%edx # a.

a_loop: decl %ecx # while (--ix
    js a_exit # >=0) {
    addl (%edx,%ecx,4),%eax # sum += a[ix].
    jmp a_loop # }

a_exit: popl %ebp # return sum.
    ret #
```

INF1070

Mengder

En mengde er en en datastruktur hvor verdiene kan være med eller ikke helt uavhengig av hverandre.

Mulige tips i tipping: H U B

Garderinger: {H,U} {H,B} {U,B} {H,U,B}

Det er i alt 8 mulige mengder over H,U,B:

{ } {H} {U} {B} {H,U} {H,B} {U,B} {H,U,B}

Implementasjon

Bestem bit-posisjon: 0=H, 1=U, 2=B

{H} 0 0 0 0 0 0 0 1

{B} 0 0 0 0 0 0 1 0 0

{H,U} 0 0 0 0 0 0 0 1 1

INF1070

Bit-operasjoner

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.
```

INF1070

Snitt mellom mengder

$$\{H,B\} \cap \{H,U\}$$

```

0 0 0 0 0 1 0 1
AND 0 0 0 0 0 0 1 1
= 0 0 0 0 0 0 0 1
= {H}

```

Union mellom mengder

$$\{H,B\} \cup \{H,U\}$$

```

0 0 0 0 0 1 0 1
OR 0 0 0 0 0 0 1 1
= 0 0 0 0 0 1 1 1
= {H,U,B}

```

Mengde-differanse

$$\{H,B\} \setminus \{H,U\}$$

```

0 0 0 0 0 1 0 1
AND NOT 1 1 1 1 1 1 0 0
= 0 0 0 0 0 1 0 0
= {B}

```

INF1070

Tekster

X86 har noen spesielle operasjoner som er til hjelp ved tekstoperasjoner og ved flytting av store mengder data:

movsb flytter en byte fra (%esi) til (%edi)
 cmpsb sammenligner (%esi) og (%edi)
 scasb sammenligner (%edi) med %al
 stosb lagrer %al i (%edi)

Alle vil dessuten øke (%esi) og (%edi). Det vil si:

D = 0 økning
 D = 1 senkning

D-flagget gis riktig verdi med

cld D-flagget nulles
 std D-flagget settes

INF1070

Tekstinstruksjonene kan gis et *prefiks* som forteller hvor lenge de skal jobbe:

```

rep  gjenta så lenge %ecx>0
repz gjenta så lenge %ecx>0 og Z=1
repnz gjenta så lenge %ecx>0 og Z=0

```

Eksempel

Denne funksjonen vil nulle ut et område i minnet:

```

.global erase
# Navn:      erase.
# Synopsis:  Nuller ut et område i minnet.
# C-signatur: void erase (char *a, int n).
erase:      pushl   %ebp          # Standard
            movl   %esp,%ebp     # funksjonsstart.
            pushl  %edi          # Gjem unna EDI.

            movl   8(%ebp),%edi   # Initiér EDI
            movl   12(%ebp),%ecx  # og ECX.
            cld                   # Økende adresser.
            movl   $0,%eax        # Fyllverdien er 0.
            rep   stosb          # Og sett i gang!

            popl   %edi          # Hent tilbake EDI
            popl   %ebp          # og EBP.
            ret                   # return.

```

INF1070

Funksjonskall

Hittil har vi ikke trengt lokale variable i en funksjon; det gjør vi i rekursive funksjoner. Det enkleste er å sette av en *kallblokk* på stakken:

```

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

```

0x10001020	:	
	:	
0x1000101c	parameter 2	
0x10001018	parameter 1	
0x10001014	returadresse	
0x10001010	gammel EBP	← EBP
0x1000100c	lokal var 2	
	lokal var 1	← ESP
	:	
	:	

INF1070

Eksempel

Standardeksemplet på en rekursiv funksjon er Fibonacci-funksjonen:

$$F_0 = F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

```
.globl fib
# Navn: fib.
# Synopsis: Beregner et gitt Fibonacci-tall.
# C-signatur: int fib (int n).
# Teknikk: Tallet beregnes med vanlig formel:
#         fib(0)=fib(1)=1,
#         fib(n)=fib(n-1)+fib(n-2).
# Lokale var: -4(%ebp) nx: n
#           -8(%ebp) f1: fib(n-1)

fib:   pushl   %ebp           # Standard
      movl   %esp,%ebp    # funksjonsstart.
      subl   $8,%esp     # Sett av kallblokk.

      movl   $1,%eax     # Hvis n<=1,
      cmpl  $1,8(%ebp)  # er svaret
      jle   fib_x        # 1.

      movl   8(%ebp),%edx # n
      decl  %edx         # -1.
      movl   %edx,-4(%ebp) # nx =
      pushl %edx         # nx).
      call  fib          # fib(
      movl  %eax,-8(%ebp) # f1 =
      popl  %edx         # /* Rydd opp */

      movl  -4(%ebp),%edx # nx
      decl  %edx         # -1
      pushl %edx         #
      call  fib          # fib(
      addl  -8(%ebp),%eax # f = fib1+
      popl  %edx         # /* Rydd opp */

fib_x: movl   %ebp,%esp   # Gjenopprett stakker.
      popl  %ebp
      ret
```

INF1070

Tidtagning

Det er to fundamentalt ulike måter å måle tiden på.

Bruke OS-mekanismer

Dette er en enkel pakke med tid.h og tid.c:

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/times.h>
```

```
void start_tid (void);
double slutt_tid (void);
```

```
#include "tid.h"
```

```
static clock_t st_time;
```

```
static clock_t read_time (void)
```

```
{
    return times(NULL);
}
```

```
void start_tid (void)
```

```
{
    st_time = read_time();
}
```

```
double slutt_tid (void)
```

```
{
    return (read_time()-st_time)/
           (double)sysconf(_SC_CLK_TCK);
}
```

INF1070

Hvor lang tid tar en mull?

Her er to kall med og uten instruksjonen:

```
.globl tom
tom:   pushl   %ebp
      movl   %esp,%ebp

      movl   $17,%eax

      popl   %ebp
      ret

.globl mult
mult:  pushl   %ebp
      movl   %esp,%ebp

      movl   $17,%eax
      mull  %eax

      popl   %ebp
      ret
```

INF1070

... og her er måleprogrammet:

```
#include "tid.h"

#define N 1000000000

extern void tom (void);
extern void mult (void);

int main (void)
{
    double tid_tom, tid_mul;
    int i;

    start_tid();
    for (i = 1; i <= N; ++i) tom();
    tid_tom = slutt_tid();
    printf("Tom løkke: %fs\n", tid_tom);

    start_tid();
    for (i = 1; i <= N; ++i) mult();
    tid_mul = slutt_tid();
    printf("Multiplikasjon: %fs\n", tid_mul);

    printf("En mull tar %gs\n", (tid_mul-tid_tom)/N);
    return 0;
}
```

```
Tom løkke: 4.650000s
Multiplikasjon: 8.520000s
En mull tar 3.87e-09s
```

INF1070

Å telle sykler

En annen mulighet er å bruke prosessorens innebygde teller som gir antall utførte sykler siden den ble slått på.

```
rdtsc      # Legger antall sykler i %edx:%eax.
```

Mer om dette siden.

INF1070