Dagens tema

- Lengselen etter fart
  - Når er fart viktig?
  - Hvordan måle fart?
  - Hvordan oppnå fart?
- Spesielle instruksjoner
- Makroer
- Blanding av C og assemblerkode
- Selvmodifiserende kode

Tidtagning

Det er tre fundamentalt ulike måter å måle tiden på.

Å lese manualene

Utifra dokumentasjonen kan man finne ut hva som er sykeltiden og hvor mange sykler hver instruksjon tar.

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.
```

Operativsystemet vet slikt som sykeltid:

```
% more /proc/cpuinfo
processor  : 0
vendor_id  : AuthenticAMD
model name : AMD Athlon(tm) 64 Processor 3500+
cpu MHz    : 2194.835
cache size : 512 KB
fdiv_bug   : no
fpu        : yes
bogomips   : 4391.86
```


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);
}
```


Hvor lang tid tar en mull?

Her er to kall med og uten instruksjonen:

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

        movl    $16,%eax

        pop     %ebp
        ret

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

        movl    $16,%eax
        mull    %eax

        pop     %ebp
        ret

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

        movl    $16,%eax
        sall    $4,%eax

        pop     %ebp
        ret
```

. . . og her er måleprogrammet:

```
#include <stdio.h>
#include "tid.h"

#define N 1000

extern void tom(void), mult(void), skift(void);

int main (void)
{
  double tid_intet, tid_tom, tid_mul, tid_skift;
  long long i, nMill = (long long)N*1000000;

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

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

  start_tid();
  for (i = 1;  i <= nMill;  ++i) mult();
  tid_mul = slutt_tid();
  printf("Multiplikasjon: %fs\n", tid_mul);
  printf("En mull tar %gs\n", (tid_mul-tid_tom)/nMill);

  start_tid();
  for (i = 1;  i <= nMill;  ++i) skift();
  tid_skift = slutt_tid();
  printf("Skifting:       %fs\n", tid_skift);
  printf("En skift tar %gs\n", (tid_skift-tid_tom)/nMill);
  return 0;
}
```

```
Tom løkke:      7.480000s
Tom funksjon:   9.040000s
Multiplikasjon: 9.910000s
En mull tar 8.7e-10s
Skifting:       10.030000s
En skift tar 9.9e-10s
```

INF2270

Måling av ulike kodeversjoner

Vi vil måle ekskveringstiden på forskjellige versjoner av strlen. Her er måleprogrammet:

```
#include <stdio.h>
#include "tid.h"

extern int mylen (char *s);

#define N 10000
#define SIZE 1000000

char a[SIZE+1];

int main (void)
{
  int i, res;

  for (i = 0;  i < SIZE;  ++i) a[i] = 'x';
  a[SIZE] = 0;

  start_tid();
  for (i = 1;  i <= N;  ++i) res = mylen(a);
  double t = slutt_tid();
  printf("res = %d   søketid = %fs\n", res, t/N);
  return 0;
}
```

INF2270

Bruk av vektorer («array-er»)

Denne koden:

```
int mylen (char s[])
{
  int i;

  for (i = 0;  ;  ++i)
    if (s[i] == 0) return i;
}
```

Resultat: 2,48 ms

INF2270

Bruk av pekere

Her er en versjon som bruker pekere i stedet:

```
int mylen (char *s)
{
  char *p = s;

  while (*p) ++p;
  return p-s;
}
```

Resultat: 1,65 ms (dvs en forbedring på 33%)

INF2270

Assemblerkode

Hva med å skrive funksjonen i assemblerkode i stedet?

```
        .globl  mylen
mylen:  movl    4(%esp),%eax
myl_l:  cmpb    $0,(%eax)
        jz      myl_x
        incl    %eax
        jmp     myl_l

myl_x:  subl    4(%esp),%eax
        ret
```

Resultat: 1,26 ms (dvs 24% raskere enn pekere)

INF2270

Blokkoperasjoner (B&O'H-boken 3.4.2–3)

X86 har noen spesielle operasjoner som er til hjelp ved tekstoperasjoner og ved flytting av store mengder data («sb» = «string of bytes») som tekst:

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

INF2270

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

rep	**%ecx** ganger
repz	**%ecx** ganger og **Z**=1
repnz	**%ecx** ganger og **Z**=0

Eksempel

Denne funksjonen vil nulle ut et område i minnet:

```
        .globl 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.
```

INF2270

Bruk av spesialinstruksjoner

Hvor fort går det med spesialinstruksjonene?

```
        .globl  mylen
mylen:  pushl   %edi
        movl    8(%esp),%edi
        movl    $0xffffffff,%ecx
        cld
        movb    $0,%al
        repnz scasb

        movl    $-2,%eax
        subl    %ecx,%eax
        popl    %edi
        ret
```

Resultat: 1,64 ms (dvs 30% mer enn vanlig assemblerkode!)

INF2270

Standardfunksjonen

Men hva med Cs standardfunksjon?

```
#include <string.h>

int mylen (char *s)
{
  return strlen(s);
}
```

Resultat: 0,54 ms (dvs 57% fortere enn vår assemblerkode!)

INF2270

Konklusjon

En velkjent tese er:

> 90% av eksekveringstiden brukes i 10% av koden.

Av dette følger:

> Om man fordobler hastigheten i den mest «aktive» delen av koden, vil man tjene 45%.
>
> Om man fordobler hastigheten i den minst aktive delen, vil man tjene 5%.

Derfor

Optimalisér koden der det virkelig teller — og legg mest vekt på lesbarhet ellers.

INF2270

Automatisk optimalisering

Hva klarer så gcc når den blir bedt om å optimalisere koden (med opsjonen «-O3»)?

	Uten -O3	Med -O3
Vektorer	2,48 ms	1,25 ms
Pekere	1,65 ms	0,86 ms
Assembler	1,26 ms	1,25 ms
repnz-operator	1,64 ms	1,63 ms
Standard strlen	0,54 ms	0,54 ms

Ofte er dette den beste løsningen.

INF2270

Beregning av π

I INF1000 er en obligatorisk oppgave å beregne π med 15 000 desimaler. Som eksempel skal jeg gjøre det samme med 500 000 desimaler.

Rett frem-løsning i Java	2.47.25	1 t
Forbedret algoritme	2.05.00	¼ t
C-versjon	1.34.31	2 t
C-kode med -O3	1.26.45	0 t
Assemblerkode	0.18.15	5 t

Konklusjon (nesten sann)

Det er ingenting som som slår en god assemblerprogrammerer!

INF2270

Alle prosessorer har en variant av add-instruksjonen som tar med seg mente:

adcb $1,**%al**

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0

Den er ypperlig egnet til å addere multipressisjons heltall.

INF2270

Makroer

Ofte gjentar man kodelinjer når man skriver assemblerkode. Da kan det lønne seg å definere en *makro*:

```
        .macro  funcstart
        pushl   %ebp
        movl    %esp,%ebp
        .endm

        .macro  funcend
        popl    %ebp
        ret
        .endm

        .globl  evenize
# Navn:         evenize.
# Synopsis:     Runder av verdien til et partall.
# C-signatur:   int evenize (int x)
evenize:
        funcstart
        movl    8(%ebp),%eax
        andl    $0xfffffffe,%eax
        funcend
```

INF2270

En makro er *tekst* som settes inn under assembleringen.

```
> > gcc -c -Wa,-al gd.s
  1                             .macro  funcstart
  2                             pushl   %ebp
  3                             movl    %esp,%ebp
  4                             .endm
  5
  6                             .macro  funcend
  7                             popl    %ebp
  8                             ret
  9                             .endm
 10
 11                             .globl  evenize
 12                             evenize.
 13                     # Navn:         Runder av verdien til et partall.
 14                     # Synopsis:     int evenize (int x)
 15                     # C-signatur:
 16 0000                evenize:
 17 0000 5589E5                 funcstart
 18 0003 8B4508                 movl    8(%ebp),%eax
 19 0006 83E0FE                 andl    $0xfffffffe,%eax
 20 0009 5DC3                   funcend
```


Nye instruksjoner

Man kan også bruke makroer til å definere instruksjoner man savner:

```
        .macro  clrb    reg
        xorb    \reg,\reg
        .endm

        .macro  clrw    reg
        xorw    \reg,\reg
        .endm

        .macro  clrl    reg
        xorl    \reg,\reg
        .endm

.globl f
f:      clrl    %eax
        ret
```

Tese

God bruk av makroer gjør programkoden bedre, men dårlig bruk av makroer gjør den mye verre.

INF2270

Avansert bruk av makroer

Kombinert med tester har man nesten et eget programmeringsspråk:

```
1                       .macro  ints   from, to
2                       .long   \from
3                       .if     \to-\from
4                       ints    (\from+1),\to
5                       .endif
6                       .endm
7
8                       .globl  table
9 0000 05000000         table:  ints   5,8
9      06000000
9      07000000
9      08000000
```


Å blande C og assemblerkode

```
#include <stdio.h>

typedef unsigned int uint;

uint mult (uint a, uint b)
{
  return a*b;
}

int main (void)
{
  uint res = 1;
  int i;

  for (i = 1;  i <= 12;  ++i) {
    res = mult(res,10);
    printf("%2d:%14u\n", i, res);
  }
  return 0;
}
```

Dette programmet gir galt svar:

```
 1:             10
 2:            100
 3:           1000
 4:          10000
 5:         100000
 6:        1000000
 7:       10000000
 8:      100000000
 9:     1000000000
10:     1410065408
11:     1215752192
12:     3567587328
```


La oss bruke assemblerkode til å multiplsere i stedet:

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned int unint;

unint mult (unint a, unint b)
{
  unint res, top;
  asm("mull %%edx" :
      "=a" (res), "=d" (top) : "a" (a), "d" (b));
  if (top) {
    fprintf(stderr, "\n**Overflow**\n");  exit(1);
  }
  return res;
}

int main (void)
{
  unint t = 1;
  int i;

  for (i = 1;  i <= 12;  ++i) {
    t = mult(t,10);
    printf("%2d:%14u\n", i, t);
  }
}
```

```
 1:             10
 2:            100
 3:           1000
 4:          10000
 5:         100000
 6:        1000000
 7:       10000000
 8:      100000000
 9:     1000000000

**Overflow**
```


«Funksjonen» asm

Man skriver «inline assembly» med en asm-konstruksjon. Den har inntil fire deler adskilt med kolon(!):

❶ selve koden

❷ utparametre

❸ innparametre

❹ ekstra registre

Assemblerkoden

Koden skrives som vanlig assemblerkode, men

- registre angis som **%%eax**
- **%0**, **%1**, . . . angir parametre
- flere instruksjoner skilles med \n

Kompilatoren gcc genererte denne koden

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    8(%ebp), %eax
        movl    12(%ebp), %edx
#APP
        mull %edx
#NO_APP
        movl    %eax, -4(%ebp)
        movl    %edx, %eax
        movl    %eax, -8(%ebp)
        cmpl    $0, -8(%ebp)
        je      .L5
        movl    $.LC0, 4(%esp)
        movl    stderr, %eax
        movl    %eax, (%esp)
        call    fprintf
        movl    $1, (%esp)
        call    exit
.L5:
        movl    -4(%ebp), %eax
        leave
        ret
```

fra funksjonen

```
unint mult (unint a, unint b)
{
  unint res, top;
  asm("mull %%edx" :
      "=a" (res), "=d" (top) : "a" (a), "d" (b));
  if (top) {
    fprintf(stderr, "\n**Overflow**\n");  exit(1);
  }
  return res;
}
```

INF2270

Parametrene
Ut- og innparametre bruker en spesiell notasjon

"*xxx*" (*var*)

som tolkes slik:

- Variabelen forteller hvilken C-variabel det dreier seg om.
- Spesifikasjonen *xxx* legger restriksjoner på hvorledes variabelen kommer til assemblerkoden:
  - **a** register **%EAX**
  - **b** register **%EBX**
  - **r** et vilkårlig register
  - **m** minnet
  - **g** ingen restriksjoner
  - ***n*** samme som parameter nr *n*
  - **=** variabelen blir endret

Ekstra registre
Her angis om man bruker (dvs ødelegger) andre registre enn parametrene.

INF2270

Et eksempel til
Denne funksjonen sjekker en addisjon ved å se om **C**-flagget blir satt:

```
unint add (unint a, unint b)
{
  unint res, carry;
  asm("xorl %%edx,%%edx\n addl %3,%0\n adcl %%edx,%%edx\n movl %%edx,%1"
      : "=r" (res), "=g" (carry)
      : "0" (a), "g" (b)
      : "edx");
  if (carry) {
    fprintf(stderr, "\n** Overflow **\n");
    exit(1);
  }
  return res;
}
```

INF2270

Dette testprogrammet

```
int main (void)
{
  unint val = 0xfffffffc0;
  int i;

  for (i = 1;  i <= 12;  ++i) {
    val = add(val,10);
    printf("%2d:%14u\n", i, val);
  }
  return 0;
}
```

gir dette resultatet:

```
 1:   4294967242
 2:   4294967252
 3:   4294967262
 4:   4294967272
 5:   4294967282
 6:   4294967292

** Overflow **
```

INF2270

Assemblerkoden tolkes slik:

- Koden inneholder fire instruksjoner:

```
xorl   %edx,%edx        # Nuller ut EDX
addl   b,res            # Addisjonen
adcl   %edx,%edx        # Flytt C-flagg til EDX
movl   %edx,carry       # og så til carry.
```

- Det er to utparametre som begge endres:
  - **res** må være i et register
  - **carry** kan være i hva som helst
- Det er to innparametre:
  - **a** er i samme register som res
  - **b** kan være hvor som helst
- Registeret **%EDX** blir ødelagt.

Oppsummering

(«Språket» for blandingskode er ganske mye rikere enn det som er nevnt hittil.)

+ Blandingskode kan gi tilgang til maskinressurser som ikke kan nås fra høynivåspråket.
+ Det er en liten hastighetsgevinst i å slippe call+ret.
+ Man reduserer antall filer.
– Programmene er like lite portable som assemblerfiler.
– Man må lære et nytt «språk» for å programmere blandingskode.
– Koden blir mindre oversiktlig.
– Man er aldri sikker på om kompilatoren genererer riktig kode.

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.

Konklusjon

Det er morsomt at det går an, men slik kode kan neppe kalles hverken lettlest eller trygg.