

# Dagens tema

## Digital eksamen

### Raskere kode [REB&DRO'H kap 5]

- Når er det viktig?
- Hvordan måle eksekveringshastighet?
- Hvordan oppnår man raskere kode?
- Blanding av C og assemblerkode

## Digital eksamen

Eksamen 11. juni kl 14.30 vil bli gjennomført på datamaskin:

- Oppgavene vises én og én i en sikker nettleser:
  - flervalgsoppgaver besvares direkte
  - tekstopp-gaver besvares i en enkel editor
  - programmeringsoppgaver besvares i en annen editor
- Det er ikke tilgang på nett eller annen programvare enn nettleseren.
- Det er anledning til å besvare oppgaver på papir.

Hva skjer ved eksamen på datamaskin?

## Prøveeksamen

Det vil bli arrangert en frivillig prøveeksamen tirsdag 20. mai kl 10.15–12.

## Tilbakemelding

Vi vil veldig gjerne høre etterpå hva dere synes om digital eksamen. Dette er viktig med tanke på å ta det i bru i flere emner.

Er hastighet så viktig?

## Når er eksekveringshastighet viktig?

Ofte har noen millisekunder fra eller til lite å si, men andre ganger er det avgjørende for suksess:

- Når prosessering må skje i sanntid
- Når det er en skikkelig stor oppgave som skal løses
- Når man lager kode som mange skal bruke ofte
- Når det er en konkurransesituasjon

## Vårt eksempel

Vi trenger en rask **strlen** fordi vi skal jobbe mye med *lange* tekster i *Latin-1* på *norsk*.

## Data

Som eksempel skal vi bruke den norsk oversettelsen fra 1930 av Bibelen.

```
% wc bibelen.text  
31167 819253 4061543 bibelen.text
```

Vi bruker 100 eksemplarer etter hverandre.

## Forsøk 1

### En naiv implementasjon

```
int strlen_array(char s[])  
{  
    int ix = 0;  
  
    while (s[ix] != 0) ++ix;  
    return ix;  
}
```

Hvor rask er denne?

# Hvordan måle ekskveringstiden?

## Målemetode 1

Tell antall eksekveringssykler

Noen prosessorer teller sykler; på en x86-maskin kan vi bruke

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

(På våre maskiner er C-typen `long long` på 64 bit og funksjonsverdier av denne typen returneres i `%EDX:%EAX` når vi er i 32-bits modus.)

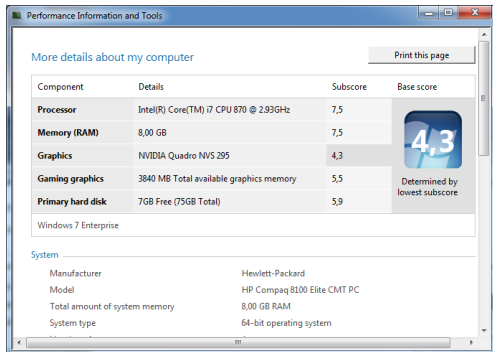
Hvordan måle hastighet?

## Hvor lang tid tar hver sykel?

### Linux

```
% more /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
model name : Intel(R) Core(TM) i7 CPU
            870@2.93GHz
cpu MHz : 1199.000
cache size : 8192 KB
fdiv_bug : no
fpu : yes
bogomips : 5852.19
```

### Windows 7



The screenshot shows the 'Performance Information and Tools' window in Windows 7. It displays a table of system components with their details, subcores, and base scores. A large '4.3' score is shown on the right, determined by the lowest subcore score. Below the table, system information is listed, including the manufacturer (Hewlett-Packard), model (HP Compaq 8100 Elite CMT PC), total system memory (8,00 GB RAM), and system type (64-bit operating system).

Component	Details	Subscore	Base score
Processor	Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz	7,5	4,3 Determined by lowest subcore
Memory (RAM)	8,00 GB	7,5	
Graphics	NVIDIA Quadro NVS 295	4,3	
Gaming graphics	3840 MB Total available graphics memory	5,5	
Primary hard disk	7GB Free (75GB Total)	5,9	
Windows 7 Enterprise			

System

Manufacturer	Hewlett-Packard
Model	HP Compaq 8100 Elite CMT PC
Total amount of system memory	8,00 GB RAM
System type	64-bit operating system

(altså 2,93 GHz)



## Konklusjon

- + Nøyaktig (i utgangspunktet)
- men moderne CPU-er kan redusere hastigheten for å spare strøm
- Vil bli forstyrret av OS-er og andre prosesser

Fungerer bare på maskiner uten OS eller for veldig korte instruksjonssekvenser.

### Løsning

Kjør testkoden mange ganger og velg det *raskeste* resultatet.

## Målemetode 2

Ta tiden med stoppeklokke

```
$ time ./prog  
real    0m17.571s  
user    0m13.749s  
sys     0m3.778s
```

(men nøyaktigheten er sjelden 1 ms så kjør testen flere ganger!)

## Konklusjon

- + Realistisk
- Ikke helt nøyaktig
- Tar med initieringen
- Forutsetter at det går få andre prosesser på maskinen

## Målemetode 3

### Bruk OS-mekanismer

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

double cpu_time(void)
{
    struct tms t;

    times(&t);
    return (t.tms_utime+t.tms_cutime)/
        (double)sysconf(_SC_CLK_TCK);
}
```

(På min Linux-maskin er tikk 10 ms.)

## Resultatet

Enkel C-versjon med array 865,6 ms

Testen ble kjørt i løkke 50 ganger for å få et rimelig nøyaktig resultat.

### Tommelfingerregel

La testen kjøre minst 10 s.

Kan dette gjøres bedre?

## Forsøk 2

### Bruk pekere i stedet for array.

```
long strlen_pekere(char *s)
{
    char *p = s;

    while (*p != 0) ++p;
    return p-s;
}
```

## Resultatet

Enkel C-versjon med array      865,6 ms

Enkel C-versjon med pekere      853,0 ms

Hvorfor det?

## Med array:

```
.L3:    jmp     .L2
        addl  $1, -4(%ebp)
.L2:    movl   -4(%ebp), %eax
        addl  8(%ebp), %eax
        movzbl (%eax), %eax
        testb %al, %al
        jne  .L3
```

## Med pekere:

```
.L8:    jmp     .L7
        addl  $1, -4(%ebp)
.L7:    movl   -4(%ebp), %eax
        movzbl (%eax), %eax
        testb %al, %al
        jne  .L8
```

## Forsøk 3

### Er ikke assemblerkode raskere?

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

l_as:   movl   8(%ebp),%eax
        cmpb   $0, (%eax)
        jz     x_as
        incl   %eax
        jmp    l_as

x_as:   subl   8(%ebp),%eax
        popl   %ebp
        ret
```



## Resultatet

Enkel C-versjon med array            865,6 ms

Enkel C-versjon med pekere           853,0 ms

Enkel versjon i assemblerkode       254,7 ms

Ikke verst, men kan den bli enda bedre?

# Spesialinstruksjoner

## Forsøk 4

Har ikke en CISC-prosessor som x86 noen egnede instruksjoner?

**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 %EDI og (for **movsb** og **cmplib**)

%ESI. Det vil si:

$D = 0$  økning

$D = 1$  senkning

$D$ -flagget gis riktig verdi med

**cld**  $D$ -flagget nulles

**std**  $D$ -flagget settes

Byte-instruksjonene gis et *prefiks* som forteller hvor lenge de skal jobbe:

**rep**      %ECX ganger

**repz**     %ECX ganger og  $Z=1$

**repnz**    %ECX ganger og  $Z=0$

## Koden blir da:

```
.globl  strlen_ass_rep
strlen_ass_rep:
    pushl   %ebp
    movl   %esp,%ebp
    pushl   %edi

    movl   8(%ebp),%edi
    movl   $0xffffffff,%ecx
    cld
    movb   $0,%al
    repnz  scasb

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

## Resultatet

Enkel C-versjon med array	865,6 ms
Enkel C-versjon med pekere	853,0 ms
Enkel versjon i assemblerkode	254,7 ms
Assemblerkode med <b>repnz</b>	252,0 ms

## Forsøk 5

### Hva med optimalisering?

De fleste kompilatorer kan lage optimalisert kode.

-03

Enkel C-versjon med array	865,6 ms	249,2 ms
Enkel C-versjon med pekere	853,00 ms	249,6 ms
Enkel versjon i assemblerkode	254,7 ms	
Assemblerkode med <b>repnz</b>	252,0 ms	

## Forsøk 6

### Kan vi forbedre algoritmen?

Vi kan kanskje få en raskere funksjon om vi sjekker 4 og 4 tegn.

- Vi får færre tester.
- Vi reduserer antall hopp. (Og hopp tar lang tid pga pipeline-en.)
- Vi utnytter maskinens båndbredde og får færre aksesser til cache-en.

(Punkt 1 og 2 kalles «utrulling av løkker».)



## Problem 1: Hvordan sjekke raskt om minst én av fire byte er 0.

### Løsning

$$(((x - 0x01010101) \& \sim x) \& 0x80808080) == 0$$

- 1 Ved  $-1$  vil 0 skifte fortegn til 1, andre verdier ikke.
- 2 Ved  $\&\sim x$  vil de bit som skiftet  $0 \rightarrow 1$ , bli 1, de øvrige 0.
- 3 Ved siste  $\&$  nuller vi alle bit unntatt fortegnsbite-ene.

Kompilatoren kan bedre!

$$((x - 0x01010101) \& \sim x) \& 0x80808080 == 0$$
$$00000000 - 00000001 \rightarrow 11111111 \& 11111111 \rightarrow$$
$$11111111 \& 10000000 \rightarrow 1000000 \neq 0$$
$$0xxxxxxx - 00000001 \rightarrow 0yyyyyyy \& 1\overline{xxxxxxx} \rightarrow$$
$$0zzzzzzz \& 10000000 \rightarrow 00000000 = 0$$
$$10000000 - 00000001 \rightarrow 01111111 \& 01111111 \rightarrow$$
$$01111111 \& 10000000 \rightarrow 00000000 = 0$$
$$1xxxxxxx - 00000001 \rightarrow 1yyyyyyy \& 0\overline{xxxxxxx} \rightarrow$$
$$0zzzzzzz \& 10000000 \rightarrow 00000000 = 0$$

Kompilatoren kan bedre!

Testløkken blir da:

```
        movl    8(%ebp),%eax
        movl    $0x01010101,%ebx
        :
l_c4b:  movl    (%eax),%edx
        addl    $4,%eax
        movl    %edx,%ecx
        subl    %ebx,%edx
        notl   %ecx
        andl   %ecx,%edx
        testl  $0x80808080,%edx
        jz     l_c4b
```

## Et ekstra problem

Når testen slår til, må vi sjekke byte-ene én for én for å finne 0-byte-en.

```
subl    $4,%eax
cmpb   $0,(%eax)
je     x_c4b
incl   %eax
cmpb   $0,(%eax)
je     x_c4b
incl   %eax
cmpb   $0,(%eax)
je     x_c4b
incl   %eax
```

## En forbedring

Vi bør unngå å bryte ord-grenser. Derfor må vi ta 0–3 byte først.

```
a_c4b:  testl    $0x3,%eax
         jz      l_c4b
         cmpb   $0, (%eax)
         je     x_c4b
         incl  %eax
         jmp   a_c4b
```

Kompilatoren kan bedre!

Det endelige resultatet er da:

**-03**

Enkel C-versjon med array	865,6 ms	249,2 ms
Enkel C-versjon med pekere	853,0 ms	249,6 ms
Enkel versjon i assemblerkode	254,7 ms	
Assemblerkode med <b>repnz</b>	252,0 ms	
Sjekk 4 og 4 tegn	98,6 ms	

Men vi kjører nå på en 64-bits maskin! Hva om vi dropper -m32 og sjekker 8 og 8 byte?

**-03**

Enkel C-versjon med array	865,6 ms	249,2 ms
Enkel C-versjon med pekere	853,0 ms	249,6 ms
Enkel versjon i assemblerkode	254,7 ms	
Assemblerkode med <b>repnz</b>	252,0 ms	
Sjekk 4 og 4 tegn	98,6 ms	
Sjekk 8 og 8 tegn	53,7 ms	

## Hva gjør Linux `clib` og `gcc`?

- 1 Kall på `strlen("xxx...")` beregnes av `gcc`.
- 2 Andre direkte kall på `strlen` oversettes til kode med **repz** (altså ingen kall og retur).
- 3 Indirekte kall på `strlen` (f eks som parameter) tester 4 og 4 (eller 8 og 8) byte med følgende test basert på at det nesten bare brukes engelske bokstaver:

```
(x - 0x01010101) & 0x80808080
```

(Falske positive må oppdages under byte-for-byte-testen.)



## Oppsummering

**-03**

Enkel C-versjon med array	865,6 ms	249,2 ms
Enkel C-versjon med pekere	853,0 ms	249,6 ms
Enkel versjon i assemblerkode	254,7 ms	
Assemblerkode med <b>repnz</b>	252,0 ms	
Sjekk 4 og 4 tegn	98,6 ms	
Sjekk 8 og 8 tegn	53,7 ms	
Linux inline-kode	250,4 ms	
strlen i clib 32 bit	92,0 ms	
strlen i clib 64 bit	53,8 ms	

Hvordan unngår man kall og retur?

## 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
```



Hvordan unngår man kall og retur?

La oss bruke assemblerkode til å multiplisere i stedet:

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

typedef unsigned int uint;

uint mult (uint a, uint b)
{
    uint res, top;
    asm("mull %%edx" :
        "=a" (res), "=d" (top) :
        "a" (a), "d" (b));
    if (top) {
        fprintf(stderr,
            "\n**0verflow**\n");
        exit(1);
    }
    return res;
}
```

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

**\*\*0verflow\*\***

## «Funksjonen» asm

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

- 1 selve koden
- 2 utparametre
- 3 innparametre
- 4 ekstra registre

## Assemblerkoden

Koden skrives som vanlig assemblerkode, men

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

Hvordan unngår man kall og retur?

```

uint mult (uint a, uint b)
{
  uint res, top;
  asm("mull %%edx" :
      "=a" (res), "=d" (top) :
      "a" (a), "d" (b));
  if (top) {
    fprintf(stderr,
            "\n**0verflow**\n");
    exit(1);
  }
  return res;
}

```

```

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

```

## Parametrene

Ut- og innparametre bruker en spesiell notasjon

"xxx" (*var*)

som tolkes slik:

- Variabelen *var* angir en C-variabel.
- 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 param *n*
  - =** variabelen blir endret

Hvordan unngår man kall og retur?

## Ekstra registre

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



## Oppsummering om inline-kode

(«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** og **pushl/popl** av parametrene.
- + 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.

Hastighet er viktig, men ikke alltid!

## Nyttig å tenke på

**Amdahls lov** sier at hvis vi forbedrer en  $\alpha$ -del av eksekveringen med en faktor  $k$ , er forbedringen

$$S = \frac{1}{(1 - \alpha) + \alpha/k}$$

En velkjent tese er:

*90% av kjøretiden brukes i 10% av koden.*

- Fordobler man hastigheten i den mest «aktive» delen av koden, oppnår man  $S = \frac{1}{(1-0,9)+0,9/2} = 1,82$ .
- Om man dobler hastigheten i den minst aktive delen, får man  $S = \frac{1}{(1-0,1)+0,1/2} = 1,05$ .

Hastighet er viktig, men ikke alltid!

## Konklusjon

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