Dagens tema

- Feilsøking
  - gdb
  - ddd
  - Egne feilutskrifter
- Funksjoner
  - Lokale variable
- Overflyt
- Egne instruksjoner for byte-sekvenser, f eks tekster

Debuggere

En «debugger» er et meget nyttig feilsøkingsverktøy. Det kan

- analysere en program-dump,
- vise innholdet av variable,
- vise hvilke funksjoner som er kalt,
- kjøre programmet én og én linje, og
- kjøre til angitt stoppunkter.

Debuggeren gdb er laget for å brukes sammen med gcc. Den har et vindusgrensesnitt som heter ddd som kan brukes på Unix-maskiner.

For å bruke gdb/ddd må vi gjøre to ting:

- kompilere våre programmer med opsjonen -g, og
- angi at vi ønsker programdumper:

```
ulimit -c unlimited
```

  hvis vi bruker bash. (Da må vi huske å fjerne programdumpfilene selv; de er noen ganger *store!*)

Et program med feil

Hovedprogrammet; se også (B&O'H-boken 3.11):

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

extern char *mystrcpy (char *til, char *fra);

char *s;

int main (void)
{
  mystrcpy(s, "Abc");
  printf("Teksten \"%s\" har %d tegn.", s, strlen(s));
  exit(0);
}
```

Assemblerrutinen:

```
        .globl mystrcpy
# Navn:         mystrcpy.
# Synopsis:     Kopierer en tekst.
# C-signatur:   char *mystrcpy (char *til, char *fra)
# Registre:     AL  - tegn som flyttes
#               ECX - til (som økes)
#               EDX - fra (som økes)

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

        movl    8(%ebp),%ecx    # Hent til
        movl    12(%ebp),%edx   # og fra.
                                # do {
mys_l:  movb    (%edx),%al      #   AL = *fra
        incl    %edx            #              ++ .
        movb    %al,(%ecx)      #   til    = AL.
        incl    %ecx            #       ++
        cmpb    $0,%al          #          AL != 0
        jne     mys_l           # } while (       )

mys_x:  movl    8(%ebp),%eax    #        til.
        popl    %ebp            #
        ret                     # return
```

INF1070

Under kjøring går dette galt:

```
> gcc -g -o feil-test-strcpy feil-test-strcpy.c strcpy.s
> ./feil-test-strcpy
Segmentation fault (core dumped)
```

De viktigste spørsmålene da er:

❶ Hvor skjer feilen?

❷ Hva vet vi om situasjonen når feilen inntreffer?

Svarene finner vi ved å analysere programdumpene.

Programdumper

Når et program dør på grunn av en feil («aborterer»), prøver det ofte å skrive innholdet av hele prosessen† på en fil slik at det kan analyseres siden.

```
> ls -l core*
-rw-------  1 dag 61440 2006-03-27 09:07 core.22577
```

† Dette kalles ofte en «core-dump» siden datamaskinene for 30–50 år siden hadde hurtiglager bygget opp av ringer med kjerne av feritt. I Unix heter denne filen derfor core.*.

Debuggeren gdb (B&O'H-boken 3.12)

Den enkleste debuggeren er gdb som finnes overalt.

```
> gdb feil-test-strcpy core.22577
GNU gdb Red Hat Linux (6.3.0.0-1.90rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, ...

warning: core file may not match specified executable file.
Core was generated by `./feil-test-strcpy'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  mys_l () at strcpy.s:18
18              movb    %al,(%ecx)      #   til    = AL.
(gdb)quit
```

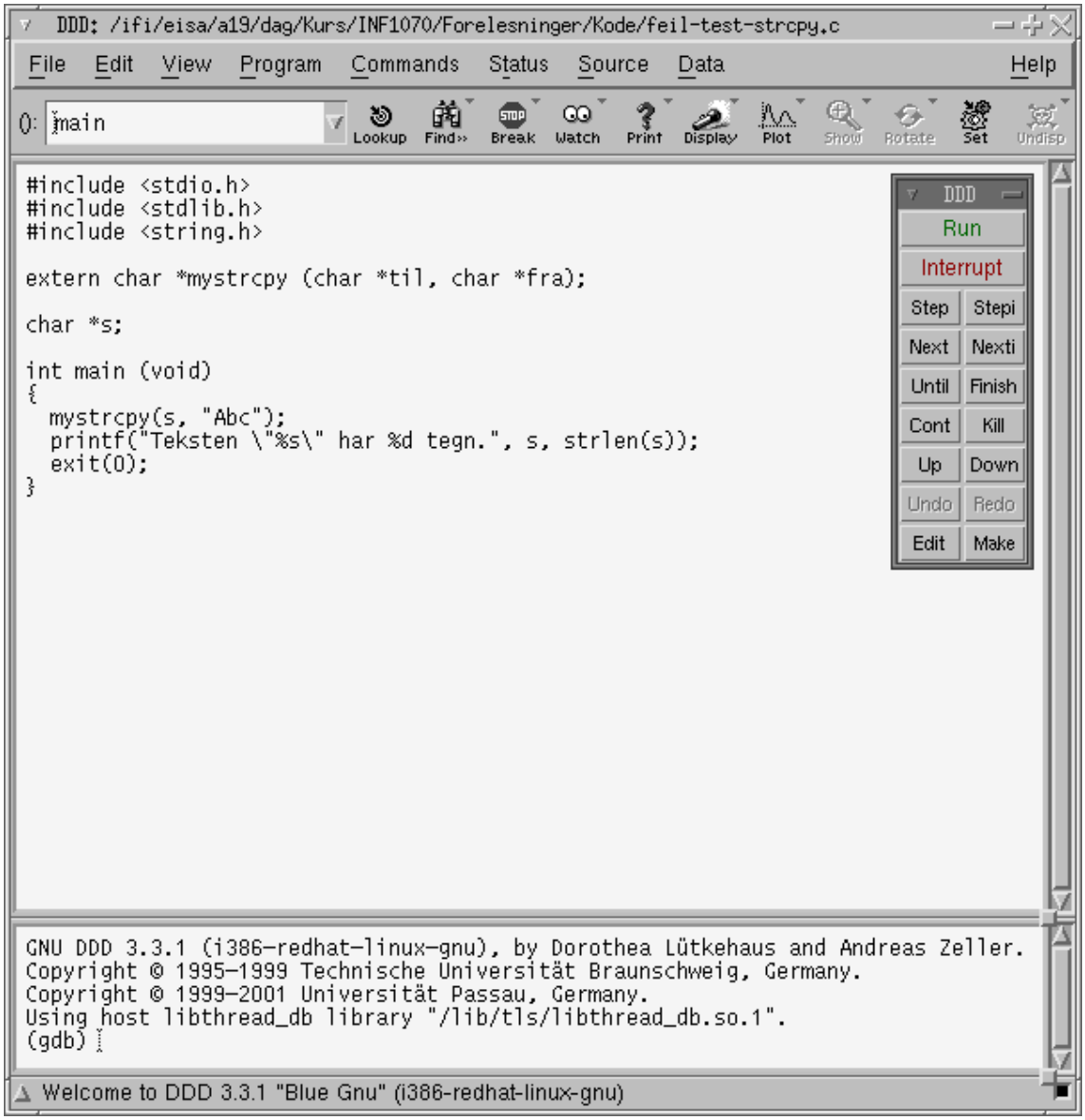
Da vet vi *hvor* feilen oppsto.

Debuggeren ddd

Denne debuggeren (som egentlig bare er et grafisk grensesnitt mot gdb) finnes dessverre ikke på alle maskiner.

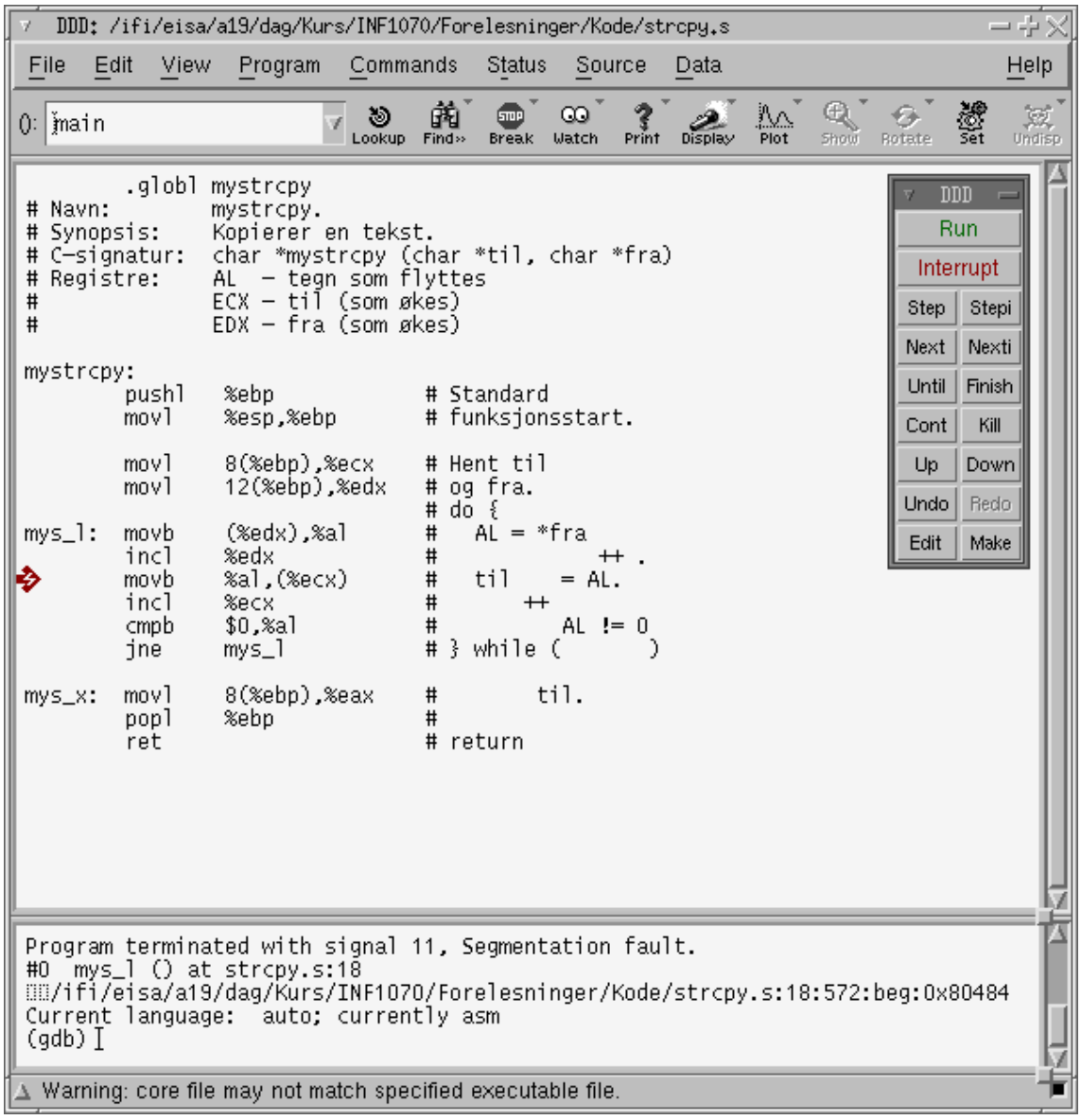
Programmet startes slik:

```
> ddd feil-test-strcpy &
```



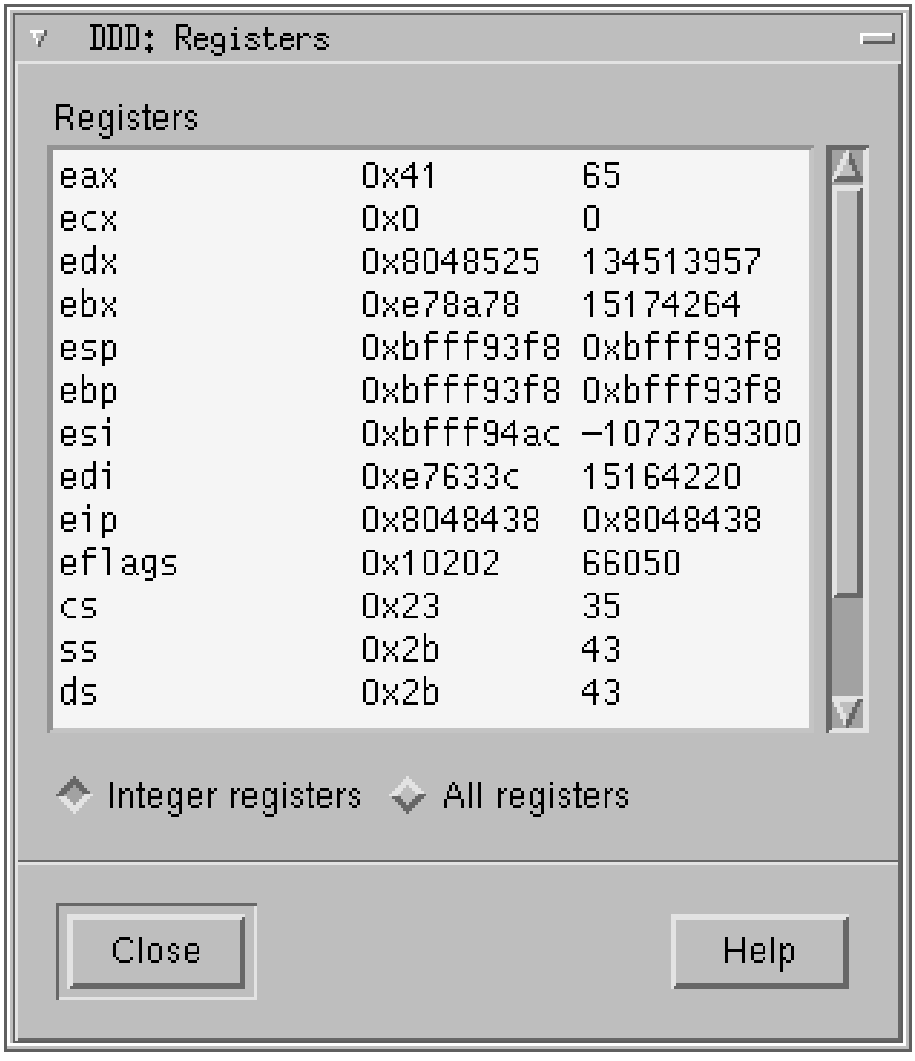
Sjekke programdumpen

I File-menyen finner vi «Open core dump» og da ser vi *hvor* feilen oppsto:



Sjekke registrene

I Status-menyen finner vi «Registers»:



Et eksempel til
Hovedprogrammet:

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

extern void swap (int *a, int *b);

int *pa, *pb;

int main (void)
{
  pa = malloc(sizeof(int));  pa = malloc(sizeof(int));
  *pa = 3;  *pb = 17;

  printf("*pa = %d, *pb = %d\n", *pa, *pb);
  swap (pa, pb);
  printf("*pa = %d, *pb = %d\n", *pa, *pb);
  return 0;
}
```

Assemblerfunksjonen:

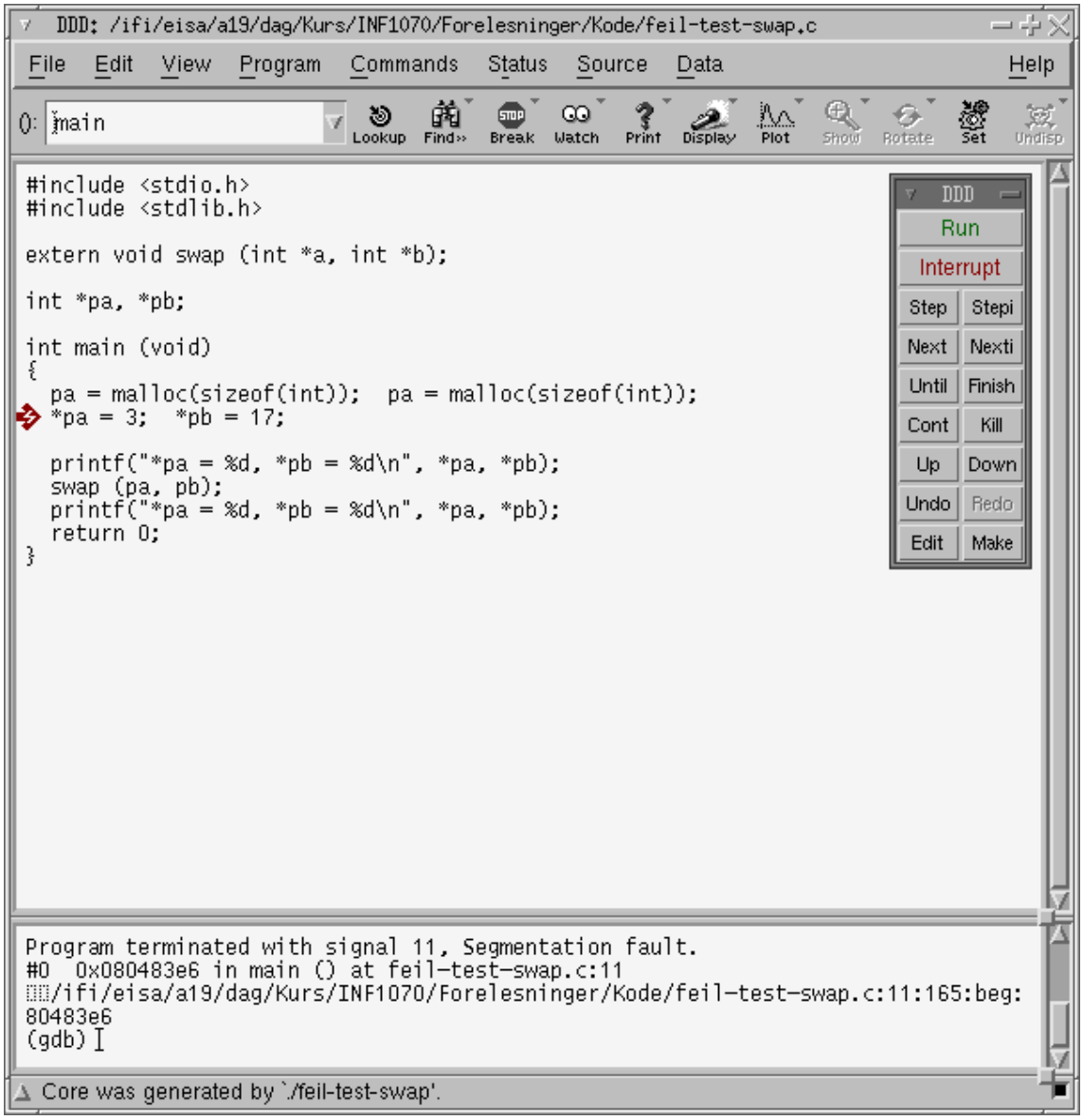
```
        .globl swap
# Navn:          swap.
# Synopsis:      Bytter om to variable.
# C-signatur:    void swap (int *a, int *b).

swap:   push    %ebp              # Standard
        movl    %esp,%ebp         # funksjonsstart

        movl    8(%ebp),%eax      # %eax = a.
        movl    12(%ebp),%ecx     # %ecx = b.

        push    (%eax)            # push *a.
        push    (%ecx)            # push *b.
        pop     (%eax)            # pop *a.
        pop     (%ecx)            # pop *b.

        pop     %ebp              # Standard retur
        ret
```

INF1070

Kjøringen:

```
> gcc -g -o feil-test-swap feil-test-swap.c swap.s
> ./feil-test-swap
Segmentation fault (core dumped)
> ddd feil-test-swap &
```

Etter «Open core dump» ser vi:



Ved å peke på navnene pa og pb ser vi at pa=0x9c06018 og pb=0x0. Dette bør fortelle oss hva som gikk galt.

Egne utskrifter
De beste feilmeldingene får vi ved å lage dem selv.

- Regn med at programmet ditt vil inneholde feil!
- Programmér feilutskrifter du kan slå av og på.
- Husk at du kan kalle C-funksjoner (dine egne og standardfunksjoner som printf) fra assemblerkode.

  (Husk bare at disse kan ødelegge **%EAX**, **%ECX** og **%EDX.**)

~inf1070/programmer/dumpreg.s anbefales:

```
        .globl  dumpreg
# Navn:         dumpreg
# Synopsis:     Skriver ut alle registrene.
# C-signatur:   void dumpreg (void).
# Registre:     Ødelegger _ingen_ registre!

dumpreg:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart
        pushl   %eax            # Gjem også EAX,
        pushl   %ecx            # ECX og EDX (siden
        pushl   %edx            # ‘printf´ kan
                                # ødelegge dem)
        pushl   %edx            # Legg EDX,
        pushl   %ecx            # ECX,
        pushl   %ebx            # EBX og
        pushl   %eax            # EAX på stakken.
        movl    4(%ebp),%eax    # Legg PC (returadr)
        pushl   %eax            # på stakken.
        leal    form1,%eax      # Legg adr til form1
        pushl   %eax            # på stakken.
        call    printf          # Kall ‘printf´.
        popl    %eax            #
        popl    %eax            # Rydd
        popl    %eax            # opp
        popl    %eax            # på
        popl    %eax            # stakken.
        popl    %eax            #

        pushl   %edi            # Legg EDI
        pushl   %esi            # og ESI på stakken.
        movl    0(%ebp),%eax    # Hent riktig EBP
        pushl   %eax            # og legg på stakken.
        movl    %ebp,%eax       # Riktig ESP er
        subl    $4,%eax         # EBP-4;  legg
        pushl   %eax            # den på stakken.
        lea     form2,%eax      # Legg adr til form2
        pushl   %eax            # på stakken.
        call    printf          # Kall ‘printf´.
        popl    %eax            #
        popl    %eax            # Rydd
        popl    %eax            # opp
        popl    %eax            # på
        popl    %eax            # stakken.

        popl    %edx            # Hent tilbake EDX,
        popl    %ecx            # ECX og
        popl    %eax            # EAX.
        popl    %ebp            #
        ret                     # Retur.
```

INF1070

```
        .data

form1:  .asciz  "Dump: PC=%08x EAX=%08x EBX=%08x ECX=%08x EDX=%08x\n"
form2:  .asciz  "                    ESP=%08x EBP=%08x ESI=%08x EDI=%08x\n"
```

Eksempel på bruk:

```
#include <stdio.h>

extern void dumpreg (void);

void f (void)
{
  dumpreg();
}

int main (void)
{
  dumpreg();
  f();
  dumpreg();
  return 0;
}
```

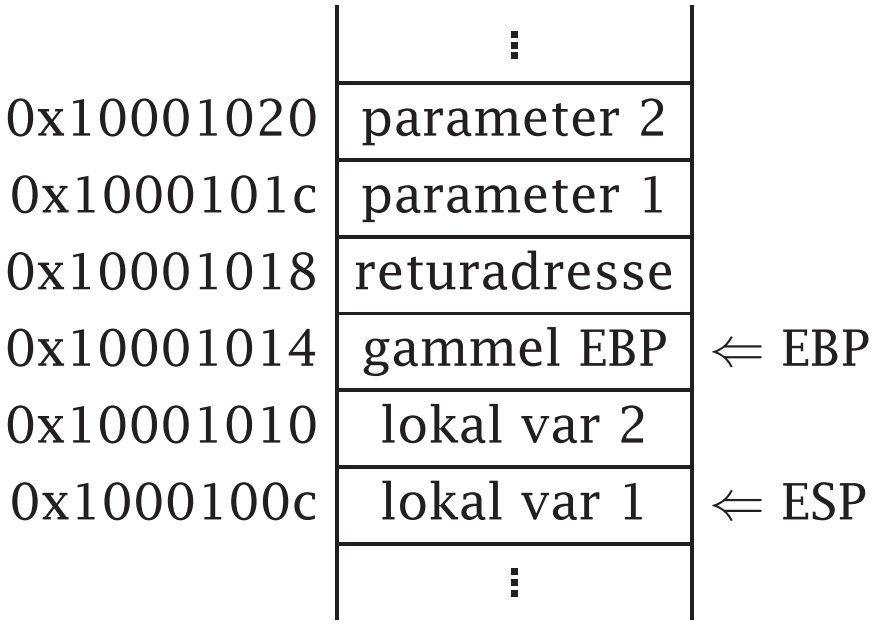
Utskriften:

```
Dump: PC=08048396 EAX=00000000 EBX=00dbda78 ECX=bfffb910 EDX=bfffb984
                  ESP=bfffb8e4 EBP=bfffb8f8 ESI=bfffb98c EDI=00dbb33c
Dump: PC=0804837f EAX=00000000 EBX=00dbda78 ECX=bfffb910 EDX=bfffb984
                  ESP=bfffb8d4 EBP=bfffb8e8 ESI=bfffb98c EDI=00dbb33c
Dump: PC=080483a0 EAX=00000000 EBX=00dbda78 ECX=bfffb910 EDX=bfffb984
                  ESP=bfffb8e4 EBP=bfffb8f8 ESI=bfffb98c EDI=00dbb33c
```

Funksjonskall (B&O'H-boken 3.7)

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.
        subl    $8,%esp         # Sett av kallblokk.
```



INF1070

Eksempel

Standardeksemplet på en rekursiv funksjon er Fibonnacchi-funksjonen:

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

```
.globl fib
# Navn:          fib.
# Synopsis:      Beregner et gitt Fibonnacci-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       # Fiks stakken.
        popl    %ebp            #
        ret                     # return f.
```

Testprogram:

```
#include <stdio.h>

extern int fib (int n);

int main (void)
{
  int i;

  for (i = 0;  i <= 10;  ++i)
    printf("fib(%2d) = %6d\n", i, fib(i));
  return 0;
}
```

Resultatet:

```
fib( 0) =      1
fib( 1) =      1
fib( 2) =      2
fib( 3) =      3
fib( 4) =      5
fib( 5) =      8
fib( 6) =     13
fib( 7) =     21
fib( 8) =     34
fib( 9) =     55
fib(10) =     89
```

Overflyt (B&O'H-boken 2.3)

Dette programmet

```
#include <stdio.h>

int main (void)
{
  signed char v = 100;
  v = v << 1;
  printf("v = %d\n", v);
  return 0;
}
```

gir galt svar:

```
v = -56
```

Feilen skyldes overflyt. Hva kan man gjøre med slikt?

- Strutse-teknikken
- Sjekke data før operasjonen
- Sjekke etter operasjonen

Heltall *uten* fortegns-bit

For addisjon og subtraksjon vil **C**-flagget bli satt ved overflyt.

Ved multiplikasjon blir det aldri problemer siden svaret kommer med dobbelt så mange bit.

Ved divisjon kan det bli et avbrudd!

```
.globl ovfl
ovfl:   movl    mill,%eax
        imull   mill
        idivl   ti
        ret

        .data
mill:   .long   1000000
ti:     .long   10
```

gir

```
Floating point exception
```

Heltall *med* fortegns-bit
Ved addisjon og subtraksjon kan man bruke **O**-flagget som settes ved overflyt. Nærmere bestemt settes det når

❶ begge operandene har likt fortegns-bit *og*

❷ resultatet har motsatt fortegns-bit.

Multiplikasjon og divisjon er som for tall uten fortegns-bit.

Konklusjon
Ved behov kan man sjekke på overflyt i assemblerprogrammering.

Tekster (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

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.
```