

Dagens tema

1. Funksjonskall

- Stakken
- Lokale variable

2. Minnet

- Fast minne
 - Store og små indianere
 - «align»-ing

- Noen nyttige instruksjoner
 - Vektorer
 - Hva er adressen?
 - Bit-operasjoner

3. Feilsøking

- gdb
- ddd
- Valgrind
- Egne testutskriften

Hva skjer ved et kall?

Rutiner [REB&DRO'H 3.7]

Vi har tre typer variabler:

- 1 Globale variabler (ligger på fast plass i minnet)
- 2 Parametre (legges på stakken av kalleren)
- 3 Lokale variabler (enten i registre eller legges på stakken av funksjonen)

Hva skjer ved et kall?

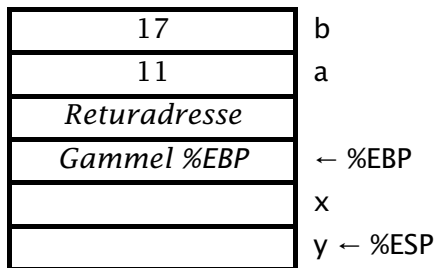
Anta at vi har
C-funksjonen

```

int f (int a, int b) 00001000
{
    int x, y;         0000fffc
    :                 0000ff8
}                    0000ff4

int main (void)      0000ff0
{
    f(11, 17);       0000fec
    :
}

```



```

a   8(%ebp)   x  -4(%ebp)
b  12(%ebp)  y  -8(%ebp)

```

Hva skjer ved et kall?

Hvis vi trenger lokale variabler på stakken, må funksjonen se slik ut:

```
f:      .globl  f
        pushl  %ebp
        movl   %esp,%ebp
        subl  $8,%esp

        # ----

        movl   %ebp,%esp
        popl   %ebp
        ret
```

Men ofte klarer vi oss med registrene!

Faste variabler

Faste variabler lever så lenge programmet kjører. De kan gis en initialverdi. Det vanlige er å legge slike variabler i .data-segmentet.

I C:

```
int a, b;  
static char c;  
long d = 5;  
  
void f (void) {}
```

I assemblerkode:

```
.globl a, b, d, f  
.text  
f:  
ret  
  
.data  
a: .long 0  
b: .long 0  
c: .byte 0  
.align 2  
d: .long 5
```

«Alignment»

Hva om vi ber CPUen utføre

```
movl    var,%eax
```

der adressen til var er 0x-----3?

Noen prosessorer klarer ikke slikt, men x86 gjør det selv om det tar mer tid.

Enda verre er det ved skrijving til minnet. På en multiprosessormaskin kan vi til og med få galt svar!

Brukeren kan angi at variabler skal være *alignet*, dvs ikke krysse ordgrenser:

```
.align n
```

Denne spesifikasjonen får assembleren til å legge inn 0 eller flere byte med ett eller annet inntil adressen er har *n* 0-bit sist.

Byte-rekkefølgen

De fleste datamaskiner i dag er byte-maskiner der man adresserer hver enkelt byte. short, int og long trenger da 2-4 byte.

Anta at register %EAX inneholder 0x01234567. Om resultatet av

```
movl    %eax,0x100
```

blir

0x100	0x101	0x102	0x103
01	23	45	67

kalles maskinen **big-endian**.

0x100	0x101	0x102	0x103
67	45	23	01

kalles maskinen **little-endian**.

Vektorer

En vektor er et sammenhengende område i minnet der man kan *regne* seg frem til hvert elements adresse.

```
int a[4];
```

ligger slik i minnet:

	0x104	0x108	0x10C	0x110	
...	a[0]	a[1]	a[2]	a[3]	...

Vektorer i x86-kode

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

$$k(\%EAX, \%EBX, n)$$

som gir adressen

$$\%EAX + n \times \%EBX + k$$

n må være 1, 2, 4 eller 8.

Vektorer [REB&DRO'H 3.8]

```

        .globl arrayadd
# Navn:      arrayadd.
# Synopsis:  Summerer verdiene i en vektor.
# C-signatur: int arrayadd (int a[], int n).
# Register:  %eax:   summen så langt
#           %ecx:   indeks til a (teller ned)
#           %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                       #

```

En egen løkkeinstruksjon

Det finnes en egen instruksjon for å gå i løkke et gitt antall ganger: `loop`.

- 1 `%ECX = %ECX - 1`
- 2 Hvis `%ECX ≠ 0`, hopp.

Eksempel

```
loop1:  ...
        loop   loop1
```

tar 15 ns.

```
loop1:  ...
        decl  %ecx
        jnz  loop1
```

tar 6 ns.

Instruksjonen lea

Instruksjonen lea («load effective address») fungerer som en mov men henter adressen i stedet for verdien.

```
eks1:  leal    var,%eax

eks2:  movl    index,%edx
       leal    array,%eax
       leal    (%eax,%edx,4),%ecx

       .data
var:    .long  12
array:  .fill  100
index:  .long  8
```

Hva kan et bit brukes til?

Bit-mønstre [REB&DRO'H 2.1]

Husk!

Alt som finnes i datamaskinen er bit-mønstre!

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 Å i kodingen ISO LATIN-1
- Starten av et Unicode-tegn
- Instruksjonen ret
- En del av en fler-bytes instruksjon
- Brukdefinerte data

Hva kan et bit brukes til?

Bit-fikling

Når alt er bit, gir det oss nye muligheter.

```

        .globl bigendian
# Navn:      bigendian.
# Synopsis:  Er maskinen «big-endian»?
# C-signatur: int bigendian (void)

bigendian:
    pushl    %ebp                # Standard
    movl    %esp,%ebp          # f-start.

    movb    v+3,%al             # Hent byte
    andl    $0x000000ff,%eax    # og null ut.

    popl    %ebp                # Standard
    ret                     # retur.

        .data
v:      .fill    1                # 0,0,0,1 e1
                                             # 1,0,0,0

#include <stdio.h>
extern int bigendian (void);

int main (void)
{
    printf("Denne maskinen er %s-endian.\n",
        (bigendian() ? "big" : "little"));
}

gir
$ ./test-endian
Denne maskinen er little-endian.

```



Hva kan et bit brukes til?

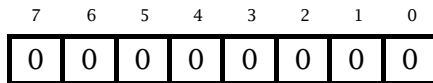
Pakking av bit

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

- for å spare plass
- for å programmere nettverk
- for å håndtere ulike tegnsett
- ...

Nummerering av bit

Det vanlige i dag er å gi minst signifikante bit (det «høyre») nr 0.



Hva kan et bit brukes til?

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

```
.globl bit2til4
# Navn: bit2til4
# Synopsis: Henter bit 2-4.
# C-signatur: int bit2til4 (int v)
# Register: EAX - arbeidsregister
```

bit2til4:

```
    pushl   %ebp                # Standard
    movl   %esp,%ebp           # funksjonstart.

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

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

Hva kan et bit brukes til?

Vi kan også sette inn bit:

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

set2til4:

```
    pushl   %ebp                # Standard
    movl   %esp,%ebp           # funksjonsstart.

    movl   8(%ebp),%eax         # Hent opprinnelig verdi
    andl   $0xffffffe3,%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    4                    # retur.
```



Hva kan et bit brukes til?

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.
```

Debuggere [REB&DRO'H 3.11]

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

- analysere en program-dump,
- vise innholdet av variabler,
- 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. Det finnes flere vindusgrensesnitt som kan brukes på Unix-maskiner: ddd, insight,

Programdumper

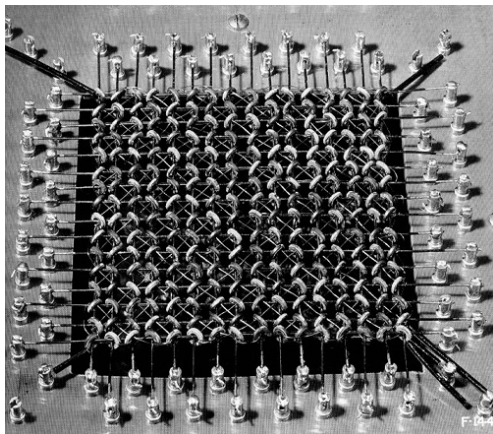
Når et program dør på grunn av en feil («aborterer»), prøver det ofte å skrive innholdet av hele prosessen på en fil slik at det kan analyseres siden.

```
$ ls -l core*
```

```
-rw----- 1 dag ifi-a 143360 2012-03-14 10:29 core.17608
```

Hva er 'core'?

En fil med lagerinnholdet 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.*.



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

- 1 kompilere våre programmer med opsjonen -g, og
- 2 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

```

#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("\n%s\n" har %d tegn.",
        s, strlen(s));
    exit(0);
}

        .globl mystrcpy
# Navn:      mystrcpy.
# Synopsis:  Kopierer en tekst.
# C-signatur: char *mystrcpy (char *til, char *fra)
# Register:  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 {
        movb   (%edx),%al          # AL = *fra
        incl   %edx                #
        movb   %al,(%ecx)         # til = AL.
        incl   %ecx                #
        cmpb   $0,%al             #
        jne   mys_1                # } while ( )

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

```



Under kjøring går dette galt:

```
$ gcc -m32 -g -o feil-strcpy feil-strcpy.c strcpy.s  
$ ulimit -c unlimited  
$ ./feil-strcpy  
Segmentation fault (core dumped)
```

De viktigste spørsmålene da er:

- 1 Hvor skjer feilen?
- 2 Hva vet vi om situasjonen når feilen inntreffer?

Svarene finner vi ved å analysere programdumpene.

Programmet 'gdb'

Debuggeren gdb

Den enkleste debuggeren er gdb som finnes overalt.

```
$ gdb feil-strcpy core.17608
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-45.el5)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /hom/dag/Kurs/INF2270/2012/Forelesninger/uke-12/feil-strcpy...done.
[New Thread 25248]
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by './feil-strcpy'.
Program terminated with signal 11, Segmentation fault.
#0  mys_l () at strcpy.s:18
18      movb  %al,(%ecx)      #   til   = AL.
(gdb)
```

Da vet vi *hvor* feilen oppsto.



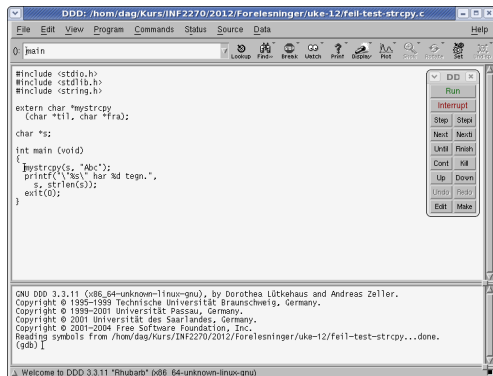
Programmet 'ddd'

Debuggeren ddd

Denne debuggeren
(som er et grafisk
grensesnitt mot gdb)
finnes på Ifi.

Programmet startes
slik:

```
$ ddd feil-strcpy &
```



Programmet 'ddd'

Sjette programdumpen

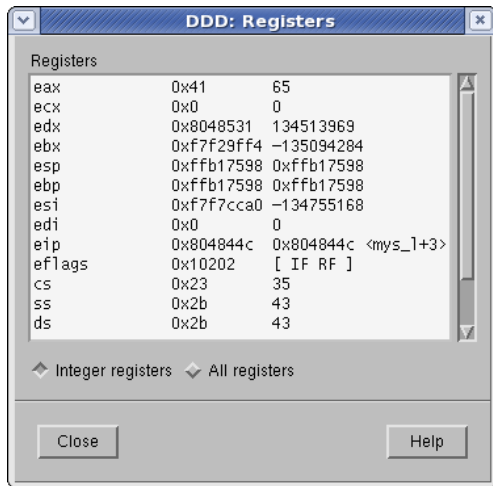
I File-menyen finner vi «Open core dump» og da ser vi *hvor* feilen oppsto:

```
0: main
  .globl mysrcpy
  # Navn: mysrcpy
  # Synopsis: Kopierer en tekst.
  # C-signatur: char *mysrcpy (char *til, char *fra)
  # Register: AL - tegn som flyttes
  #          ECI - til (som økes)
  #          EDI - fra (som økes)
  #
mysrcpy:
  pushl %ebp          # Standard
  movl   %esp,%ebp    # Funksjonsstart.
  movl   0(%ebp),%ecx  # Hent til
  movl   12(%ebp),%edx # og fra.
  mys_1:
  movb  (%edx),%al     # do {
  incl  %eax           # AL = *fra
  movb  %al,(%ecx)    # til ++ = AL.
  incl  %ecx           # ++
  cmpb  $0,%al        # AL != 0
  jne   mys_1         # } while (
  mys_x:
  movl  0(%ebp),%eax   # til.
  popl  %ebp
```

(gdb) core-file /hom/dag/Kurs/INF2270/2012/Forelesninger/uke-12/core.25248
warning: core file may not match specified executable file.
[New Thread 25248]
Core was generated by './feil-test-srcpy'.
Program terminated with signal 11, Segmentation fault.
#0 mys_1 () at srcpy.s:18
[~/hom/dag/Kurs/INF2270/2012/Forelesninger/uke-12/srcpy.s:18:572: beg: 0x80484
(gdb)]
Warning: core file may not match specified executable file.

Sjette registrene

I Status-menyen finner vi «Registers» og da bør vi se feilen.



Programmet 'ddd'

Et eksempel til

```

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

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

int *pa, *pb;

int main (void)
{
    pa = malloc(sizeof(int));
    pb = 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;
}

```

```

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

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

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

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

        popl   %ebp           # Standard retur
        ret

```

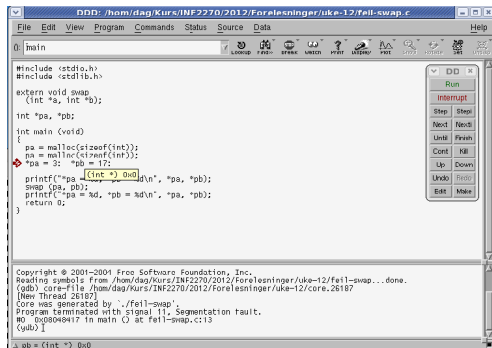


Programmet 'ddd'

Kjøringen:

```
$ gcc -m32 -g -o feil-swap  
feil-swap.c swap.s  
$ ./feil-swap  
Segmentation fault (core dumped)  
$ ddd feil-swap &
```

Etter «Open core dump» og så å peke på pa og pb ser vi at pa=0x9c06018 og pb=0x0. Dette bør fortelle oss hva som gikk galt.



```
DDD: /hom/dag/Kurs/INF2270/2012/Forelesninger/uke-12/feil-swap.c  
File Edit View Program Commands Status Source Data Help  
0: main  
#include <stdio.h>  
#include <stdlib.h>  
  
extern void swap  
(int *a, int *b);  
  
int *pa, *pb;  
  
int main (void)  
{  
    pa = malloc(sizeof(int));  
    pb = malloc(sizeof(int));  
    *pa = 3; *pb = 17;  
  
    printf("pa = %d\n", *pa, *pb);  
    swap (pa, pb);  
    printf("pa = %d, *pb = %d\n", *pa, *pb);  
    return 0;  
}  
  
Copyright © 2001-2004 Free Software Foundation, Inc.  
Reading symbols from /hom/dag/Kurs/INF2270/2012/Forelesninger/uke-12/feil-swap...done.  
(gdb) core-file /hom/dag/Kurs/INF2270/2012/Forelesninger/uke-12/core.26187  
[New Thread 26187]  
Core was generated by './feil-swap'.  
Program terminated with signal 11, Segmentation fault.  
#0  0x0804e417 in main () at feil-swap.c:13  
(gdb) !  
pa = (int *) 0x0
```

Minnelekkasje

Valgrind (<http://valgrind.org/>) er et ypperlig feilfinningsverktøy, spesielt for å finne minnelekkasjer.

Eksempel

Her er et program som leser en fil, byger opp et binært søketre av ordene og skriver dem ut i sortert rekkefølge. Vi tester dette på en tekst fra Ifis hjemmeside:

Den digitale tidsalder har festet grepet. Overalt finnes små og store datamaskiner. Det moderne samfunnet bryter sammen uten en velfungerende digital infrastruktur og hverdagen vår består av stadig flere digitale operasjoner. Informatikk er læren om alt dette og mer til.

Programmet 'Valgrind'

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct node {
6      char *navn;
7      struct node *v, *h;
8  };
9
10 struct node topp = { "", NULL, NULL };;
11
12 void sett_inn (struct node *p,
13               struct node *ny)
14 {
15     if (strcmp(ny->navn,p->navn) < 0) {
16         if (p->v) sett_inn(p->v,ny);
17         else p->v = ny;
18     } else {
19         if (p->h) sett_inn(p->h,ny);
20         else p->h = ny;
21     }
22 }
23
24 void skriv_ut (struct node *p)
25 {
26     if (p->v) skriv_ut(p->v);
27     printf("%s\n", p->navn);
28     if (p->h) skriv_ut(p->h);
29 }
31 void rydd_opp (struct node *p)
32 {
33     if (p->v) rydd_opp(p->v);
34     if (p->h) rydd_opp(p->h);
35     if (p != &topp) free(p);
36 }
37
38 int main (int argc, char *argv[])
39 {
40     FILE *f = fopen(argv[1], "r");
41     char n[200];
42
43     while (fscanf(f,"%s",n) != EOF) {
44         struct node *nx =
45             malloc(sizeof(struct node));
46         nx->navn = strdup(n);
47         nx->v = nx->h = NULL;
48         sett_inn(&topp, nx);
49     }
50     fclose(f);
51     skriv_ut(&topp); rydd_opp(&topp);
52
53     return 0;
54 }

```



Programmet ser ut til å fungere fint:

""	"har"
"Den"	"hverdagen"
"Det"	"infrastruktur"
"Informatikk"	"læren"
"Overalt"	"mer"
"alt"	"moderne"
"av"	"og"
"består"	"og"
"bryter"	"og"
"datamaskiner."	"om"
"dette"	"operasjoner."
"digital"	"samfunnet"
"digitale"	"sammen"
"digitale"	"små"
"en"	"stadig"
"er"	"store"
"festet"	"tidsalder"
"finnes"	"til."
"flere"	"uten"
"grepet."	"velfungerende"
	"vår"

Programmet 'Valgrind'

Mer er alt bra? Vi spør Valgrind:

```

$ gcc -g -m32 -O0 -o navn navn.c && valgrind --leak-check=yes navn tekst.txt
==26386== Memcheck, a memory error detector
[...]
==26386==
==26386== HEAP SUMMARY:
==26386==   in use at exit: 272 bytes in 40 blocks
==26386==   total heap usage: 81 allocs, 41 frees, 1,104 bytes allocated
==26386==
==26386== 272 bytes in 40 blocks are definitely lost in loss record 1 of 1
==26386==   at 0x6DF5B83: malloc (vg_replace_malloc.c:195)
==26386==   by 0x6EA019F: strdup (in /lib/libc-2.5.so)
==26386==   by 0x8048672: main (navn.c:46)
==26386==
==26386== LEAK SUMMARY:
==26386==   definitely lost: 272 bytes in 40 blocks
==26386==   indirectly lost: 0 bytes in 0 blocks
==26386==   possibly lost: 0 bytes in 0 blocks
==26386==   still reachable: 0 bytes in 0 blocks
==26386==   suppressed: 0 bytes in 0 blocks
==26386==
==26386== For counts of detected and suppressed errors, rerun with: -v
==26386== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 13 from 8)

```

Gjør det selv!

Egne utskrifter

De beste feilmeldingene får vi ved å lage dem selv.

- Regn med at programmet ditt vil inneholde feil!
- Programmer 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 samt flaggene.)

Gjør det selv!

~inf2270/programmer/dumpreg.s anbefales:

```
#include <stdio.h>

extern void dumpreg (void);

void f (void)
{
    dumpreg();
}

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

```
Dump: PC=080483c7 EAX=ffb79544 EBX=f7eeeff4 ECX=ffb794c0 EDX=00000001
      ESP=ffb79494 EBP=ffb794a8 ESI=f7f40ca0 EDI=00000000
Dump: PC=080483af EAX=ffb79544 EBX=f7eeeff4 ECX=ffb794c0 EDX=00000001
      ESP=ffb79484 EBP=ffb79498 ESI=f7f40ca0 EDI=00000000
Dump: PC=080483d1 EAX=ffb79544 EBX=f7eeeff4 ECX=ffb794c0 EDX=00000001
      ESP=ffb79494 EBP=ffb794a8 ESI=f7f40ca0 EDI=00000000
```

