Dagens tema

- Bit-fikling
- Makroer
- Blanding av C og assemblerkode
- Mer om Core War

Bit-fikling

Når alt er bit, gir det oss som programmerere nye muligheter.

Er maskinen big-endian?

Denne funksjonen tester det:

```
        .globl bigendian
# Navn:         bigendian
# Synopsis:     Er denne maskinen big-endian?
# C-signatur:   int bigendian (void)
# Registre:     EAX - test-byte or resultat

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

        movb    endian+3,%al    # Hent «siste» byte av 1
        andl    $1,%eax         # og test det.
                                # (Og null ut resten av EAX.)

        popl    %ebp            # Standard
        ret                     # retur.

        .data
endian: .fill   1               # 0,0,0,1 eller 1,0,0,0
```

INF1070

Hvordan lagres flyt-tall?

Vi kan bruke assemblerkode til å flytte innholde av en float til en byte-vektor og dermed unngå typereglene i høynivåspråk.

```
        .globl float2byte
# Navn:          float2byte
# Synopsis:      Viser hvordan en float lagres i 4 byte
# C-signatur:    void float2byte (float f, unsigned char b[])
# Registre:      EAX - f
#                EDX - b (dvs adressen)

float2byte:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.

        movl    8(%ebp),%eax    #      f
        movl    12(%ebp),%edx   #
        movl    %eax,(%edx)     # *b =   /* uten konvertering */

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

INF1070

```c
#include <stdio.h>

typedef unsigned char byte;

extern int bigendian (void);
extern void float2byte(float f, byte b[]);

void test (float f)
{
  byte b[4];

  float2byte(f, b);
  if (bigendian())
    printf("%10.3f lagres som %02x %02x %02x %02x\n",
          f, b[0], b[1], b[2], b[3]);
  else
    printf("%10.3f lagres som %02x %02x %02x %02x\n",
          f, b[3], b[2], b[1], b[0]);
}

int main (void)
{
  test(0.0);  test(1.0);  test(-12.8125);
  return 0;
}
```

gir resultatet

```
     0.000 lagres som 00 00 00 00
     1.000 lagres som 3f 80 00 00
   -12.812 lagres som c1 4d 00 00
```

Dette kan vi også gjøre i C ved hjelp av en spesiell konstruksjon:

En **union** plasserer sine elementer *oppå* hverandre.

```
int bigendian (void)
{
  union endian {
    int v;
    unsigned char b[4];
  } e;

  e.v = 1;
  return e.b[3];
}
```

```
void float2byte (float f, unsigned char b[])
{
  union f2b {
    float f;
    unsigned char b[4];
  } u;
  int i;

  u.f = f;
  for (i = 0;  i < 4;  ++i) b[i] = u.b[i];
}
```

I C har vi også mulighet til å omgå typereglene ved å bruke pekere:

```
int bigendian (void)
{
  int v = 1;

  return *((unsigned char*)&v) == 0;
}
```

```
void float2byte (float f, unsigned char b[])
{
  int i;

  for (i = 0;  i < 4;  ++i)
    b[i] = ((unsigned char *)&f)[i];
}
```

Pakking av bit

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

- for å spare plass
- for å programmere nettverk

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)
# Registre:     EAX - arbeidsregister

bit2til4:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonstart.

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

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

INF1070

Vi kan også sette inn bit, men da må vi bruke en **maske**:

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

set2til4:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.

        movl    8(%ebp),%eax    # Hent opprinnelig verdi
        andl    $0xfffffff3,%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                     # retur.
```

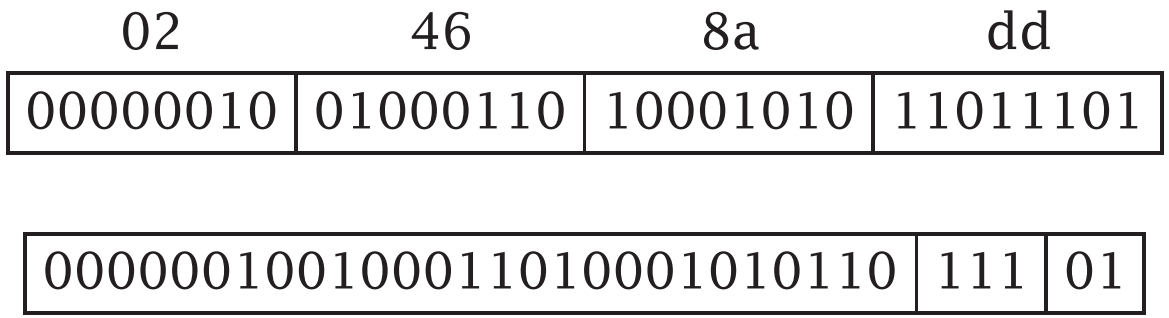
Dette kan vi også gjøre i C:

```
#include <stdio.h>

struct data {
  unsigned int a: 2;
  unsigned int b: 3;
  unsigned int c:27; } pakket;

extern float2byte (struct data d, unsigned char byte[]);
/* Egentlig er denne funksjonen for float->byte,
   men det vet ikke C-kompilatoren. */

int main (void)
{
  unsigned char b[4];

  pakket.a = 1;  pakket.b = 7;  pakket.c = 0x123456;
  float2byte(pakket, b);
  printf("struct {0x%x; 0x%x; 0x%x;} lagres som %02x %02x %02x %02x\n",
         pakket.a, pakket.b, pakket.c, b[0], b[1], b[2], b[3]);
  return 0;
}
```

Resultatet blir

struct {0x1; 0x7; 0x123456;} lagres som dd 8a 46 02

som kan tolkes slik:



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.
```

Hele byte

Når vi jobber med hele byte, har vi direkte tilgang til dem.

Makroer

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

```
        .macro  heap    size
        .long   \size-4
        .fill   \size-8
        .long   1
        .endm

.globl  myheap
myheap: heap    4096
var:    .long   0x12345678
```

INF1070

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

```
> > gcc -c -Wa,-al gd.s
   1                            .macro  heap    size
   2                            .long   \size-4
   3                            .fill   \size-8
   4                            .long   1
   5                            .endm
   6
   7                    .globl  myheap
   8 0000 FC0F0000      myheap: heap    4096
   8      00000000
   8      00000000
   8      00000000
   8      00000000
   9 1000 78563412      var:    .long   0x12345678
```

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.

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
```

INF1070

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 addere 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**
```

INF1070

«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;
}
```

INF1070

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.

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;
}
```

Dette testprogrammet

```
int main (void)
{
  unint val = 0xffffffc0;
  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 **
```

INF1070

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.

Litt mer om Core War

I *Core War* kan et program lage en kopi av seg selv et annet sted i minnet, og så kjøre de to i parallell:

```
; The 'turtle´
; (Creates concurrent copies of itself.)

Turtle
        mov     #10,Counter
        mov     #3,From
        add     #123,To
Loop    mov     <From, <To
        djz     Loop, Counter
        spl     Turtle
        jmp     @To

From    dat     $0
To      dat     $0
Counter dat     $0
        end     Turtle
```

INF1070