



## Dagens tema

### Programmering av x86

- Flytting av data
  - Endring av størrelse
- Aritmeriske operasjoner
  - Flagg
- Maskeoperasjoner
- Hopp
  - Tester
- Stakken
- Rutinekall
  - Kall og retur
  - Frie og opptatte registre
  - Dokumentasjon

**Husk!**

*Alt er bare bit-mønstre!*

## Flytting av data (Irvine-boken 4.1)

Instruksjonen `mov` kan flytte data til/fra

konstanter	\$10
registre	%eax
navngitte variable	navn
lagerlokasjoner pekt på	0(%esp)

```
move:      .text
           movl   $3,%eax
           movl   4(%esp),%eax
           movl   %eax,var
           ret

var:       .data
           .long 17
```

### Men ...

- Man kan ikke flytte *til* en konstant.
- Maksimalt én lagerlokasjon.

### Variable

Man kan sette av plass til variable med spesifikasjonen `.long`. De bør legges i *segmentet* `.data`.

## Byte, ord og langord

mov- finnes for -b («byte»), -w («word» = 2 byte) og -l («long» = 4 byte).

```
movb    $0x12,%al
movw    $0x1234,%ax
movl    $0x12345678,%eax
```

Kun de aktuelle delene av registrene endres.

## Konvertering mellom størrelser

Fra større til mindre størrelser dropper man bare de bit-ene man ikke trenger.<sup>†</sup>

00000000	00000000	00000000	00000001
----------	----------	----------	----------

Fra mindre til større *unsigned* verdier er det bare å sette inn 0-er foran.

Fra mindre til større *signed* verdier finnes disse:

```
cbw    Utvider %al til %ax.
cwd    Utvider %ax til %dx:%ax.
cwde   Utvider %ax til %eax.
cdq    Utvider %eax til %edx:%eax.
```

<sup>†</sup> Hva om tallet er for stort? *Overflow* vil vi ta for oss senere i kurset.

# Aritmetiske operasjoner

(Irvine-boken 4.2 + 7.4)

Hittil kjenner vi

Addisjon:      addb   addw   addl

Økning:        incb   incw   incl

Subtraksjon:   subb   subw   subl

Senkning:      decb   decw   decl

I tillegg har vi

Negasjon:      negb   negw   negl

Alle fungerer på registre og inntil én minnelokasjon.

## Multiplikasjon

Multiplikasjon er litt sær siden den kun jobber med faste registre:

mulb og imulb  $\%al \times op \rightarrow \%ax$

mulw og imulw  $\%ax \times op \rightarrow \%dx:\%ax$

mull og imull  $\%eax \times op \rightarrow \%edx:\%eax$

mul- er for verdier *uten* fortegn mens imul- er for de med.

Operand 2 kan være register eller minnelokasjon, men ikke konstant.

### Eksempel

```
.globl mul10
mul10:  movl    $10,%eax
        mull   4(%esp)
        ret
```

## Divisjon

Divisjon er akkurat like rar:

`divl og idivl    %edx:%eax ÷ op → %eax    %edx`  
`divw og idivw    %dx:%ax   ÷ op → %ax    %dx`  
`divb og idivb    %ax     ÷ op → %al    %ah`

## Eksempel

Denne funksjonen deler et tall med 10 og returnerer svaret og resten der de to adressene i parameter 2 og 3 angir.

```
.globl div10
# C-signatur: void div10(int v, int *q, int *r).
div10:
    movl    4(%esp),%eax    # %eax = v.
    cdq
    movl    $10,%ecx       # %ecx = 10.
    idivl   %ecx           # (%eax,%edx) = (%edx:%eax/10, %edx:%eax%10).
    movl    8(%esp),%ecx   # *q
    movl    %eax,%ecx     # = %eax.
    movl    12(%esp),%ecx  # *r
    movl    %edx,%ecx     # = %edx.
    ret
```

## Testprogram

```
#include <stdio.h>

extern void div10 (int v, int *q, int *r);

int data[] = { 0, 19, 226, -17 };

int main (void)
{
    int data_len = sizeof(data)/sizeof(int), a1, a2, ix;

    for (ix = 0; ix < data_len; ++ix) {
        div10(data[ix], &a1, &a2);
        printf("%d/10 = %d, %d%%10 = %d\n",
            data[ix], a1, data[ix], a2);
    }
    return 0;
}
```

## Kjøring

```
> gcc test-div10.c div10.s -o test-div10
> ./test-div10
0/10 = 0, 0%10 = 0
19/10 = 1, 19%10 = 9
226/10 = 22, 226%10 = 6
-17/10 = -1, -17%10 = -7
```

## Advarsel!

Overflyt ved divisjon eller divisjon med 0 er ekstra farlig; hvis det skjer, får vi se følgende:

Floating point exception



## **Flagg** (Irvine-boken 4.2.6)

De fleste operasjonene har en bieffekt: visse egenskaper ved resultatet blir lagret i *flaggene*.

**Z** («Zero») settes til 1 når svaret er 0 (og 0 ellers).

**S** («Sign») settes lik øverste bit i svaret. (Om vi regner med *signed* tall, er dette et tegn på at tallet er negativt.)

**C** («Carry» = mente) settes lik den menteoverføringen som skjedde øverst i resultatet.

**O** («Overflow») settes om svaret var for stort.

**P** («Parity») settes om *laveste byte* har et partall antall 1-bit.

Les instruksjonsoversikten i Appendix B for å finne ut når flaggene får verdi.

**Inneholder flaggene nyttig informasjon?**

Av og til, men ikke alltid.

# Maskeoperasjoner

## (Irvine-boken 6.2)

Maskeoperasjonene brukes til å sette eller nulle ut bit i henhold til et gitt mønster (en såkalt *maske*).

## Maske-AND

Denne operasjonen *nuller ut* de bit som ikke er markert i masken.<sup>†</sup>

$$\begin{array}{r} \text{andb} \\ = \end{array} \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline \end{array}$$

Denne operasjonen er tilgjengelig i C og heter der `&`.

**NB!** Det er stor forskjell på `&` (maske-AND eller bit-AND) og `&&` (logisk AND) i C:

$$1 \ \& \ 4 == 0$$

$$1 \ \&\& \ 4 == 1$$

<sup>†</sup> Siden operasjonen er symmetrisk, er det vilkårlig hvilken operand som betraktes som maske og hvilken som er data.

## Maske-OR

Denne operasjonen *setter* de bit som er markert i masken.

$$\begin{array}{r} \text{orb} \\ = \end{array} \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

Denne operasjonen er tilgjengelig i C og heter der `|`.

## Maske-NOT

Denne operasjonen snur alle bit-ene.

$$\begin{array}{r} \text{notb} \\ = \end{array} \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

Den finnes også i C og heter der  $\sim$ .

## Maske-XOR

Denne operasjonen *snur* bare de bit som er markert i masken.

$$\begin{array}{r} \text{xorb} \\ = \end{array} \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline \end{array} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline \end{array} \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Denne operasjonen også ofte «logisk addisjon». Den er tilgjengelig i C og heter der  $\wedge$ .

## Hopp (Irvine-boken 4.5)

Instruksjonen for å hoppe heter jmp.

```
jmp dit
```

```
dit:
```

## Betinget hopp (Irvine-boken 6.3)

Man kan angi at flaggene skal avgjøre om man skal hoppe.

```
jz      dit      # Hopp om Z(ero)
jnz     dit      # Hopp om ikke Z
jc      dit      # Hopp om C(arry)
jnc     dit      # Hopp om ikke C
js      dit      # Hopp om S(ign)
jns     dit      # Hopp om ikke S
jo      dit      # Hopp om O(verflow)
jno     dit      # Hopp om ikke O
jp      dit      # Hopp om P(arity)
jnp     dit      # Hopp om ikke P
```

## Testing

Flaggene kan settes som følge av vanlige instruksjoner:

```
.globl abs2
abs2:  movl    4(%esp),%eax
        addl    8(%esp),%eax
        jns    ret2
        negl    %eax
ret2:  ret
```

Alternativt kan vi eksplisitt sjekke to verdier mot hverandre med instruksjonen `cmp`:-

```
.globl abs1
abs1:  movl    4(%esp),%eax
        cmpl    $0,%eax
        jns    ret1
        negl    %eax
ret1:  ret
```

Hva er riktige flagg å sjekke på ved for eksempel `%eax ≤ -17`? Heldigvis finnes spesielle varianter som er enklere å bruke:

### Verdier *med* fortegn

<code>je</code>	<code>dit</code>	<code># Hopp ved =</code>	<code>(= Z)</code>
<code>jne</code>	<code>dit</code>	<code># Hopp ved !=</code>	<code>(= ~Z)</code>
<code>jl</code>	<code>dit</code>	<code># Hopp ved &lt;</code>	<code>(= S!=0)</code>
<code>jle</code>	<code>dit</code>	<code># Hopp ved &lt;=</code>	<code>(= Z    S!=0)</code>
<code>jg</code>	<code>dit</code>	<code># Hopp ved &gt;</code>	<code>(= ~Z &amp;&amp; S==0)</code>
<code>jge</code>	<code>dit</code>	<code># Hopp ved &gt;=</code>	<code>(= S==0)</code>

### Verdier *uten* fortegn

<code>je</code>	<code>dit</code>	<code># Hopp ved =</code>	<code>(= Z)</code>
<code>jne</code>	<code>dit</code>	<code># Hopp ved !=</code>	<code>(= ~Z)</code>
<code>jb</code>	<code>dit</code>	<code># Hopp ved &lt;</code>	<code>(= C)</code>
<code>jbe</code>	<code>dit</code>	<code># Hopp ved &lt;=</code>	<code>(= Z    C)</code>
<code>ja</code>	<code>dit</code>	<code># Hopp ved &gt;</code>	<code>(= ~C &amp;&amp; ~Z)</code>
<code>jae</code>	<code>dit</code>	<code># Hopp ved &gt;=</code>	<code>(= ~C)</code>

## Eksempel

Denne funksjonen finner det minste av to tall:

```
min2:  movl    4(%esp),%eax
        cmpl   8(%esp),%eax
        jle   ret
ret:    movl   8(%esp),%eax
        ret
```

## NB!

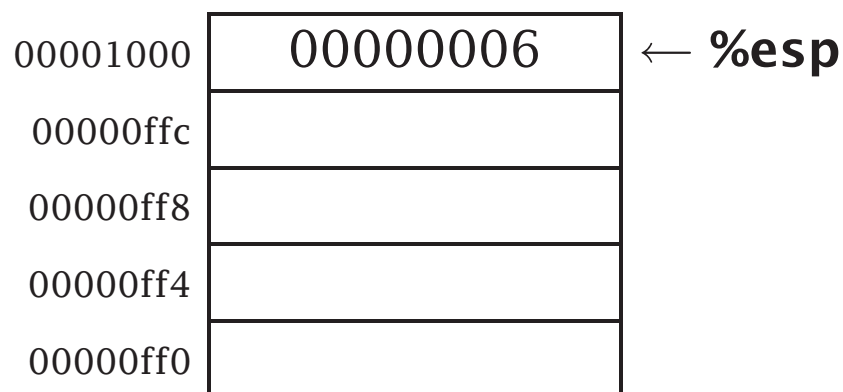
Testen blir *omvendt* i Linux siden operandene kommer i en annen rekkefølge!



## Stakken (Irvine-boken 5.4)

*Stakken* er veldig sentral i x86-arkitekturen. Den benyttes til

- rutinekall
- parameteroverføring
- lagring av mellomresultater
- plass til lokale variable



Av historiske grunner vokser stakken mot *lavere* adresser.

**Å legge elementer på stakken**  
Instruksjonene pushw og pushl legger verdier på stakken:

```
pushl    $0x000000a5
```

00001000	00000006	
00000ffc	000000a5	← %esp
00000ff8		
00000ff4		
00000ff0		

Legg merke til at vi kan få tak i alle elementene på stakken:

```
movl    0(%esp),%eax    # Toppen  
movl    4(%esp),%eax    # Nest øverst
```

## Å fjerne elementer fra stakken

Til dette brukes `popw` og `popl`:

```
popl    %eax
```

00001000	00000006	← %esp
00000ffc	000000a5	
00000ff8		
00000ff4		
00000ff0		

Verdiene blir ikke fysisk fjernet.

## Rutiner (Irvine-boken 5.5)

Ved et rutinekall skjer følgende:

- ❶ Parametrene beregnes og legges på stakken *bakfra!*
- ❷ Instruksjonen call fungerer som en jmp men legger adressen til neste instruksjon på stakken.

Kallet

```
f(4, 17, 11);
```

vil gi denne stakken:

00001000	11	
00000ffc	17	
00000ff8	4	
00000ff4	<i>Returadresse</i>	← %esp
00000ff0		

Ved retur vil ret fjerne returadressen fra stakken og hoppe dit.

(Det er opp til kalleren å fjerne parametrene fra stakken.)

## Registerbruk

Hvilke registre kan vi endre i en funksjon uten å ødelegge for kalleren?

### Frie registre

Konvensjonen er at

`%eax, %ecx og %edx`

er *frie registre* («caller save»).

### Bundne registre

De andre registrene er *bundne registre* («callee save»). Om de endres, må man ta vare på den opprinnelige verdien og sette denne tilbake før retur.

## En forbedring

Hittil har vi hentet parametrene som  
4(%esp), 8(%esp), ...

Men hva om vi ønsker å lagre  
mellomresultater på stakken? Da må  
adresseringen endres!

Løsningen er å bruke et eget register **%ebp**  
til å peke på parametrene:

```
pushl   %ebp
movl   %esp,%ebp
```

00001000	11
00000ffc	17
00000ff8	4
00000ff4	<i>Returadresse</i>
00000ff0	<i>Gammel %ebp</i> ← %esp ← %ebp

Nå er parametrene tilgjengelige som 8(%ebp),  
12(%ebp), ...

Retur må nå gjøres slik:

```
popl   %ebp
ret
```

# Dokumentasjon

Målet med dokumentasjon er man skal kunne få vite alt man trenger for å bruke en funksjon ved å lese dokumentasjonen. Dette inkluderer:

- ❶ funksjonens navn
- ❷ hva den gjør (kort fortalt)
- ❸ parametrene

I tillegg kan det være nyttig å vite hva de ulike registrene brukes til når man skal lese koden.

```
.globl mystrlen

# Name:  mystrlen.
# Synopsis:  Beregner antall tegn i en tekst.
# C-signatur:  int mystrlen (char *s)
# Register:  EAX:  len
#           ECX:  s

mystrlen:
    movl    4(%esp),%ecx    # %ecx = s.
    movl    $0,%eax        # %eax = 0.
loop:    cmpb    $0, (%ecx)    # while (%ecx
        je      exit        #           !=0) {
        incl    %eax        #     ++len.
        incl    %ecx        #     ++s.
        jmp     loop        # }
exit:    ret                # return len.
```