

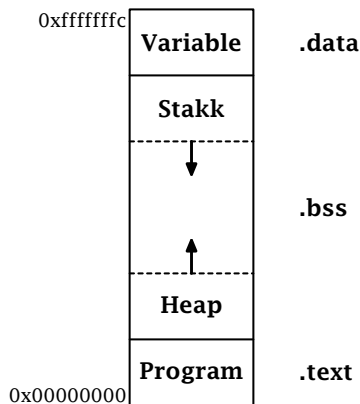
Programmering av x86 [REB&DRO'H 3.2-3.7]

- Minnestrukturen i en prosess
- Flytting av data
 - Endring av størrelse
- Aritmeriske regneoperasjoner
 - Flagg
- Maskeoperasjoner
- Skifting og rotasjoner
- Hopp
 - Tester
- Stakken
- Rutinekall
 - Kall og retur
 - Frie og opptatte registre
 - Dokumentasjon

Hvordan ser minnet ut?

Minnestrukturen

Grovt sett ser minnet for hver process slik ut:



(**bss** = «block started by symbol» fra IBM 704 ca 1950.)

Hva er de viktigste registrene?

Registre

En x86-prosessor har følgende registre:

Mer eller mindre generelle 32-bits

%AX			
%EAX		%AH	%AL
%EBX		%BH	%BL
%ECX		%CH	%CL
%EDX		%DH	%DL

%EBP
%ESP
%ESI
%EDI

Flytting av data

Instruksjonen mov kan flytte data til/fra

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

Men ...

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

Hvordan flytte data til og fra minnet?

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

        .data
var:    .long 17    # En long med verdi 17
arr:    .fill 8     # 8 byte uten initialverdi
```

Variable

Man kan sette av plass til variable med spesifikasjonen `.long` eller `.fill`. De bør legges i `.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.

Hvilke operasjoner ble nevnt forrige gang?

Aritmetiske operasjoner

Hittil kjenner vi

Addisjon: addb addw addl

Økning: incb incw incl

Subtraksjon: subb subw subl

Senkning: decb decw decl

Siden negative verdier lagres som *2-erkomplement*, fungerer disse både for tall med og uten fortegns-bit.

I tillegg har vi

Negasjon: negb negw negl

Multiplikasjon: — imulw imull

neg- fungerer på registre og variable, mens imul- er for konstanter, registre og inntil én minnelokasjon.

NB! imul- fungerer *bare* på tall *med* fortegn-bit!

Multiplikasjon

I tillegg til den vanlige utgaven nevnt på forrige ark, finnes en versjon som jobber med faste registre:

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

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

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

(Denne versjonen er fra tidlige versjoner av x86 og demonstrerer at x86 ikke alltid er systematisk bygget opp.)

NB! Denne siste versjonen har bare én parameter:

```
imull 4(%esp)
```

Fordelen med denne utgaven er at den finnes både for verdier *med* fortegn (imul- og versjonen på forrige ark) og *uten* fortegn (mul-).

Ulempen er at parameteren kan være register eller minnelokasjon, men ikke konstant.

Divisjon

Divisjon gir to svar (kvotient og rest). Den er også litt rar når det gjelder registerbruk:

divl og idivl	$\%edx:\%eax \div op \rightarrow \%eax$	$\%edx$
divw og idivw	$\%dx:\%ax \div op \rightarrow \%ax$	$\%dx$
divb og idivb	$\%ax \div op \rightarrow \%al$	$\%ah$

idiv- regner *med* fortegn og div- *uten* fortegn.

Disse instruksjonen kan ikke dele på konstanter, kun på variable og registerverdier.

Eksempel

Denne funksjonen deler et tall med 10 og returnerer svaret.

```
.globl div10

# C-signatur: unsigned int div10(unsigned int v).

div10:
    movl    4(%esp),%eax    # %eax = v.
    movl    $0,%edx        # %edx:
    movl    $10,%ecx       # %ecx = 10.
    divl   %ecx            # (%eax,%edx) =
                          # (%edx:%eax/10,%edx:%eax%10).
    ret                    # Retur.
```

Testprogram

```
#include <stdio.h>

extern unsigned int div10 (unsigned int v);

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

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

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

Kjøring

```
$ gcc -m32 test-div10.c div10.s -o test-div10
$ ./test-div10
0/10 = 0
19/10 = 1
226/10 = 22
```

Advarsel!

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

Floating point exception

Hvilke flagg har vi?

Flagg

De fleste operasjonene har en bieffekt: visse egenskaper ved resultatet blir lagret i **flaggene** («condition codes»).

%EFLAGS
%EIP

- Z («Zero») settes til 1 når svaret er 0 (og til 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 1-bit.

Inneholder flaggene nyttig informasjon? Av og til.

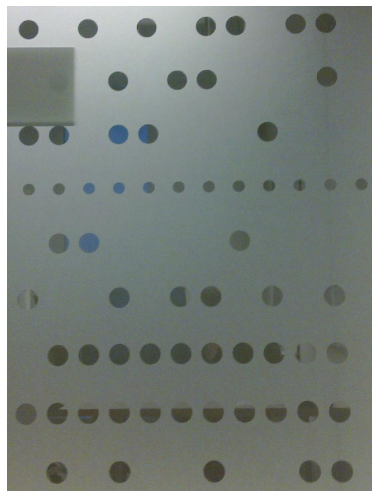
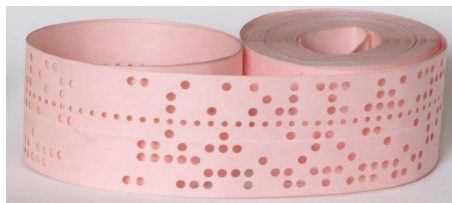


Hva er paritet?

Paritet

Paritet ble innført for å oppdage feil.

OJDs hus er dekorert med *hullbånd* der det står *Universitas* med **even parity**.



Maskeoperasjoner

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

Maske-AND

AND *nuller ut* de bit som ikke er markert i masken.

	0	1	0	1	0	1	0	1
andb	0	0	0	0	1	1	1	1
=	0	0	0	0	0	1	0	1

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`

Maske-OR

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

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

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

Maske-NOT

Denne operasjonen snur *alle* bit-ene.

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

Den finnes også i C og heter der \sim .

Maske-XOR

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

	0	1	0	1	0	1	0	1
xorb	0	0	0	0	1	1	1	1
=	0	1	0	1	1	0	1	0

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

Skift-operasjoner

Dette flytter alle bit-ene i et ord.

Logisk skift

Her settes det inn 0-er fra enden:

	0	1	0	1	0	1	1	1
salb \$1,%al	1	0	1	0	1	1	1	0
salb \$2,%al	1	0	1	1	1	0	0	0

shrb \$1,%al	0	1	0	1	1	1	0	0
shrb \$4,%al	0	0	0	0	0	1	0	1

C-flagget settes til det siste bit-et som «faller utenfor».

Aritmetisk skift

I vårt desimale tallsystem kan man gange med 10 ved å sette inn en 0, og dele med 10 ved å fjerne siste siffer:

$$42 \times 10 = 420$$

$$217/10 = 21$$

Det samme gjelder i det binære tallsystemet, men her er effekten å gange med 2 eller dele på 2:

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 (=42_{dec})

0	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

 (=84_{dec})

1	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---

 (=217_{dec})

0	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---

 (=108_{dec})

Aritmetisk skift

Hva gjør vi så hvis det er fortegnsbitt? Ved skift mot venstre spiller det ingen rolle, men for skift mot høyre er løsningen å kopiere inn fortegnsbitt-et.

	0	1	0	1	0	1	1	1	=	87 _{dec}
sarb \$1,%al	0	0	1	0	1	0	1	1	=	43 _{dec}
sarb \$2,%al	0	0	0	0	1	0	1	0	=	10 _{dec}

	1	1	0	1	0	1	1	1	=	-41 _{dec}
sarb \$1,%al	1	1	1	0	1	0	1	1	=	-21 _{dec}
sarb \$2,%al	1	1	1	1	1	0	1	0	=	-6 _{dec}

(Legg merke til at negative tall rundes av mot $-\infty$ og ikke mot 0!)

Rotasjoner

En variasjon av skifting er at bit-ene som «detter utenfor» kommer tilbake fra den andre siden:

rolb \$1,%al

0	1	0	1	0	1	1	1
1	0	1	0	1	1	1	0
1	0	1	1	1	0	1	0

rolb \$2,%al

rorb \$1,%al

0	1	0	1	1	1	0	1
1	1	0	1	0	1	0	1

rorb \$4,%al

Enda en variant er å ta med C-flagget i rotasjonen:

								C
	1	1	0	1	0	1	1	1
rclb \$1,%al	1	0	1	0	1	1	1	1
rclb \$2,%al	1	0	1	1	1	1	1	0
rcrib \$1,%al	0	1	0	1	1	1	1	1
rcrib \$4,%al	1	1	1	1	0	1	0	1

Hopp

Instruksjonen for å hoppe heter jmp.

jmp dit

dit:

Betinget hopp

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:

```
abs1:    .globl  abs1
         movl   4(%esp),%eax
         addl   $0,%eax
         jns   ret2
         negl   %eax
ret2:    ret
```

Alternativt kan vi eksplisitt sjekke to verdier mot hverandre med instruksjonen `cmp-`:¹

```
abs2:    .globl  abs2
         movl   4(%esp),%eax
         cmpl  $0,%eax
         jns   ret1
         negl  %eax
ret1:    ret
```

En tredje mulighet er å teste om visse bit er satt med
test-:²

```
s3:      .globl  s3
        testl  $0x80000000,4(%esp)
        jz    a3_pos
        movl  $1,%eax
        jmp  a3_x
a3_pos:  movl  $0,%eax
a3_x:    ret
```

Hvilket flagg skal jeg sjekke?

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

Verdier *med fortegn*

je	dit	# Hopp ved =	(= Z)
jne	dit	# Hopp ved !=	(= ~Z)
jl	dit	# Hopp ved <	(= S)
jle	dit	# Hopp ved <=	(= Z S)
jg	dit	# Hopp ved >	(= ~Z && ~S)
jge	dit	# Hopp ved >=	(= ~S)

Hvilket flagg skal jeg sjekke?

Verdier uten fortegn

je	dit	# Hopp ved =	(= Z)
jne	dit	# Hopp ved !=	(= ~Z)
jb	dit	# Hopp ved <	(= C)
jbe	dit	# Hopp ved <=	(= Z C)
ja	dit	# Hopp ved >	(= ~C && ~Z)
jae	dit	# Hopp ved >=	(= ~C)

Hvilket flagg skal jeg sjekke?

Eksempel

Denne funksjonen finner det minste av to tall:

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

NB!

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

Stakken

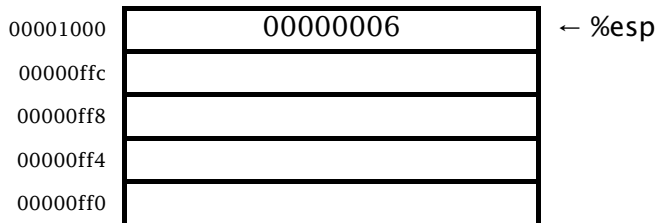
Fra *Bokmålsordboka*:

stakk stor, såteformet haug av høy, løv el med en stake i midten

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

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

Hva er en stakk

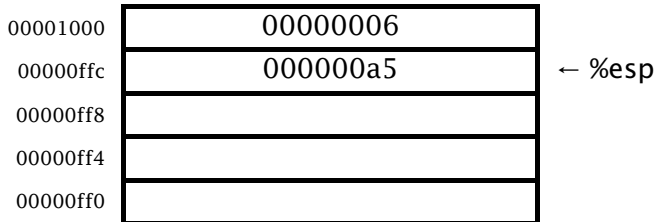


Av historiske grunner vokser stakken mot *lavere* adresser.

Å legge elementer på stakken

Instruksjonene `pushw` og `pushl` legger verdier på stakken:

```
pushl    $0x000000a5
```



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`:

<code>popl</code>	<code>%eax</code>	
00001000	00000006	← <code>%esp</code>
00000ffc	000000a5	
00000ff8		
00000ff4		
00000ff0		

Verdiene blir ikke fysisk fjernet.

Rutiner

Ved et rutinekall skjer følgende:

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

Hva skjer ved et kall?

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.)

Hvilke registre kan jeg bruke?

Registerbruk

Hvilke registre kan vi endre i en funksjon?

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

Returadresse

00000ff0

Gammel %ebp

← %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:

- 1 funksjonens navn
- 2 hva den gjør (kort fortalt)
- 3 parametrene

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

Dokumentasjon av funksjoner

```
                .globl mystradd

# Name:          mystradd.
# Synopsis:      Legger et tegn til en C-tekst.
# C-signatur:    void mystradd (char *s, char c)
# Register:      EAX:    c
#                ECX:    gjennomløper s

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

    movl    8(%ebp),%ecx        # %ecx = s

# Finn slutten av teksten:
s_loop:  cmpb    $0,(%ecx)        # while (* %ecx)
        jz     s_add             # {
        incl  %ecx              # ++ %ecx
        jmp   s_loop            # }

# Sett inn c og 0-byte:
s_add:   movl    12(%ebp),%eax    #          c
        movb   %al,(%ecx)        # * %ecx =
        movb   $0,1(%ecx)        # *(%ecx+1) = 0

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