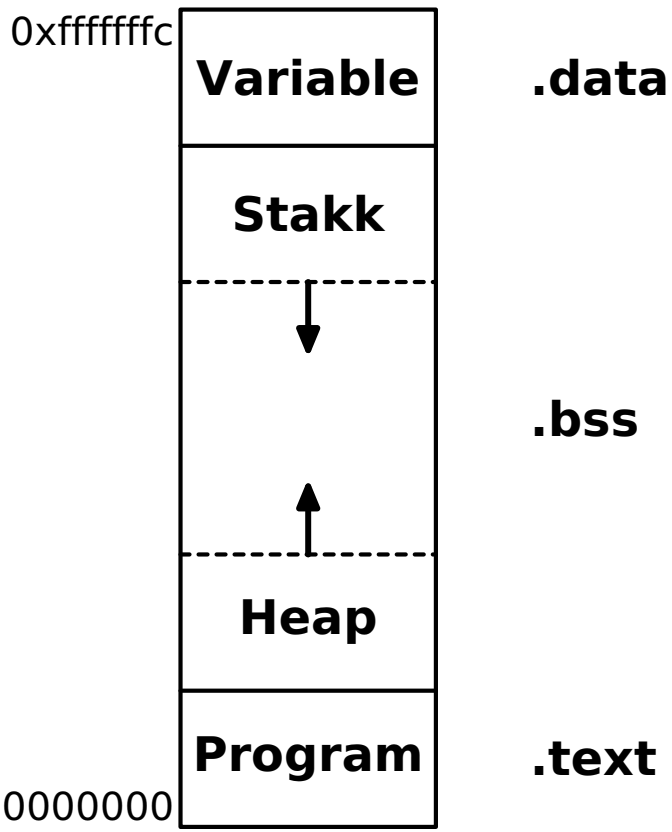
Programmering av x86 [REB&DRO'H 3.2–3.7]

- 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

Minnestrukturen

Grovt sett ser minnet for hver prosess slik ut:



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

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 variabler	`navn`
lagerlokasjoner pekt på	`0(%esp)`

Men . . .

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

```
        .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
```

Variabler

Man kan sette av plass til variabler 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.

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 variabler, mens **imul–** er for konstanter, registre og inntil én minnelokasjon.

NB! Disse instruksjonene er for data *med* fortegns-bit!

Multiplikasjon

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

mulb og **imulb**	%AL	× *op* →	%AX
mulw og **imulw**	%AX	× *op* →	%DX:%AX
mull og **imull**	%EAX	× *op* →	%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 et register eller en minnelokasjon, men ikke en konstant.

Divisjon

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

		Svar	**Rest**
divl og **idivl**	%EDX:%EAX ÷ *op* →	%EAX	%EDX
divw og **idivw**	%DX:%AX ÷ *op* →	%AX	%DX
divb og **idivb**	%AX ÷ *op* →	%AL	%AH

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

Disse instruksjonen kan ikke dele på konstanter, kun på variabler 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);

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

int main (void)
{
  int data_len = sizeof(data)/sizeof(unsigned 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
```

Flagg

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

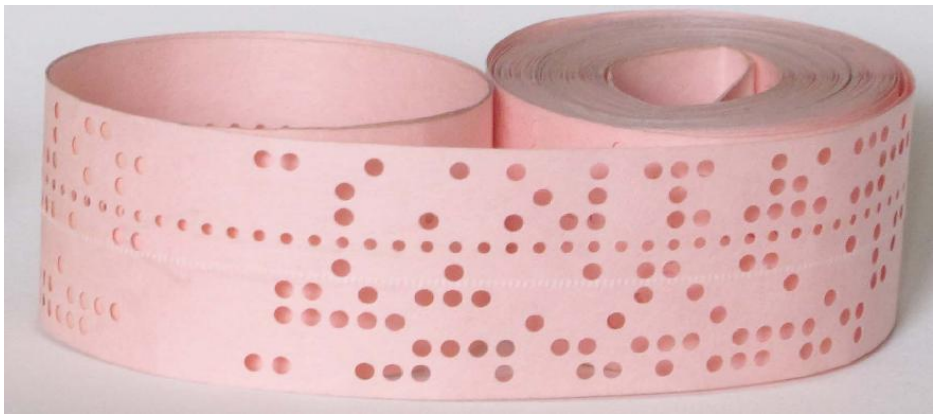
- $\mathcal{Z}$ («Zero») settes til 1 når svaret er 0 (og til 0 ellers).
- $\mathcal{S}$ («Sign») settes lik øverste bit i svaret. (Om vi regner med *signed* tall, er dette et tegn på at tallet er negativt.)
- $\mathcal{C}$ («Carry» = mente) settes lik den menteoverføringen som skjedde øverst i resultatet.
- $\mathcal{O}$ («Overflow») settes om svaret var for stort.
- $\mathcal{P}$ («Parity») settes om *laveste byte* har et partall 1-bit.

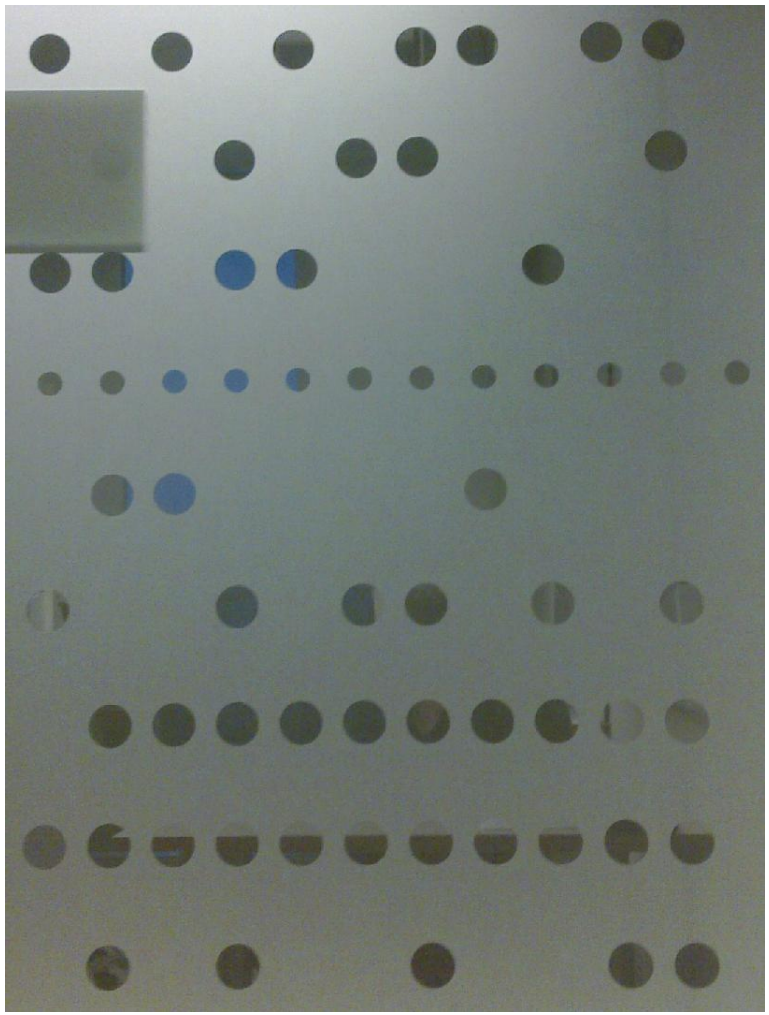
Inneholder flaggene nyttig informasjon? Av og til.

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.

	0	1	0	1	0	1	0	1
orb	0	0	0	0	1	1	1	1
=	0	1	0	1	1	1	1	1

Denne operasjonen er tilgjengelig i C og heter der |.

Maske-NOT

Denne operasjonen snur *alle* bit-ene.

notb	0	1	0	1	1	1	1	1
=	1	0	1	0	0	0	0	0

Den finnes også i C og heter der ~.

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 ^.

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:

$$42 \times 10 = 420 \qquad 217/10 = 21$$

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	$(=42_{10})$
0	1	0	1	0	1	0	0	$(=84_{10})$

1	1	0	1	1	0	0	1	$(=217_{10})$
0	1	1	0	1	1	0	0	$(=108_{10})$

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

	0	1	0	1	0	1	1	1	$= 87_{10}$
`sarb $1,%AL`	0	0	1	0	1	0	1	1	$= 43_{10}$
`sarb $2,%AL`	0	0	0	0	1	0	1	0	$= 10_{10}$

	1	1	0	1	0	1	1	1	$= -41_{10}$
`sarb $1,%AL`	1	1	1	0	1	0	1	1	$= -21_{10}$
`sarb $2,%AL`	1	1	1	1	1	0	1	0	$= -6_{10}$

(NB! 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:

	0	1	0	1	0	1	1	1
`rolb $1,%AL`	1	0	1	0	1	1	1	0
`rolb $2,%AL`	1	0	1	1	1	0	1	0

`rorb $1,%AL`	0	1	0	1	1	1	0	1
`rorb $4,%AL`	1	1	0	1	0	1	0	1

Enda en variant er å ta med $\mathcal{C}$-flagget i rotasjonen:

									$\mathcal{C}$
	1	1	0	1	0	1	1	1	1
`rclb $1,%AL`	1	0	1	0	1	1	1	1	1
`rclb $2,%AL`	1	0	1	1	1	1	1	1	0
`rcrb $1,%AL`	0	1	0	1	1	1	1	1	1
`rcrb $4,%AL`	1	1	1	1	0	1	0	1	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:

```
        .globl  abs1
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]

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

[1] Instruksjonen **cmp–** er egentlig en **sub–** der resultatet kastes.

En tredje mulighet er å teste om visse bit er satt med **test–**:[2]

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

[2] Instruksjonen **test–** er egentlig en **and–** der resultatet kastes.

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

Verdier *med* fortegn

```
je      dit     # Hopp ved =
jne     dit     # Hopp ved !=
jl      dit     # Hopp ved <
jle     dit     # Hopp ved <=
jg      dit     # Hopp ved >
jge     dit     # Hopp ved >=
```

Verdier *uten* fortegn

```
je      dit     # Hopp ved =
jne     dit     # Hopp ved !=
jb      dit     # Hopp ved <
jbe     dit     # Hopp ved <=
ja      dit     # Hopp ved >
jae     dit     # Hopp ved >=
```

Eksempel

Denne funksjonen finner det minste av to tall:

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

Stakken

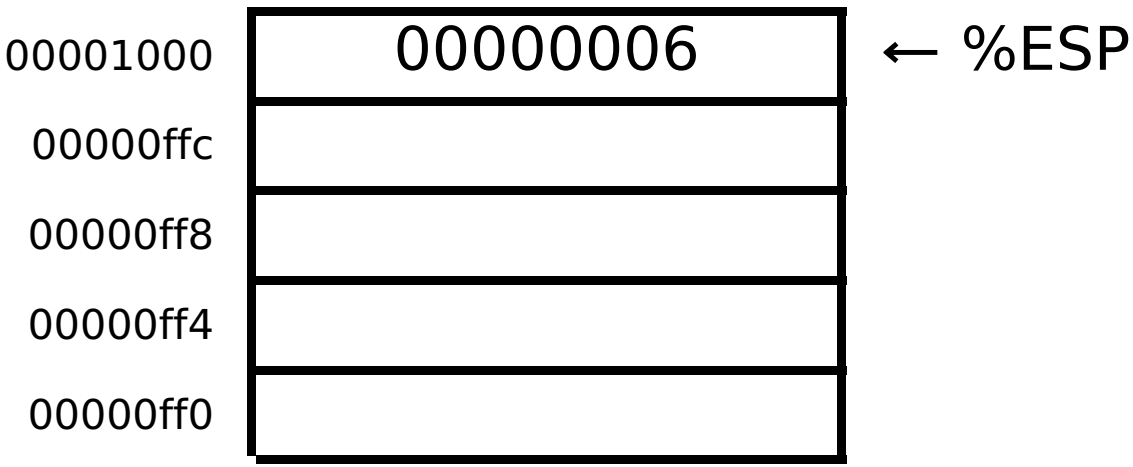
Fra *Bokmålsordboka*:

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

Stakken er veldig sentral i x86-arkitekturen til

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

Hva er en stakk

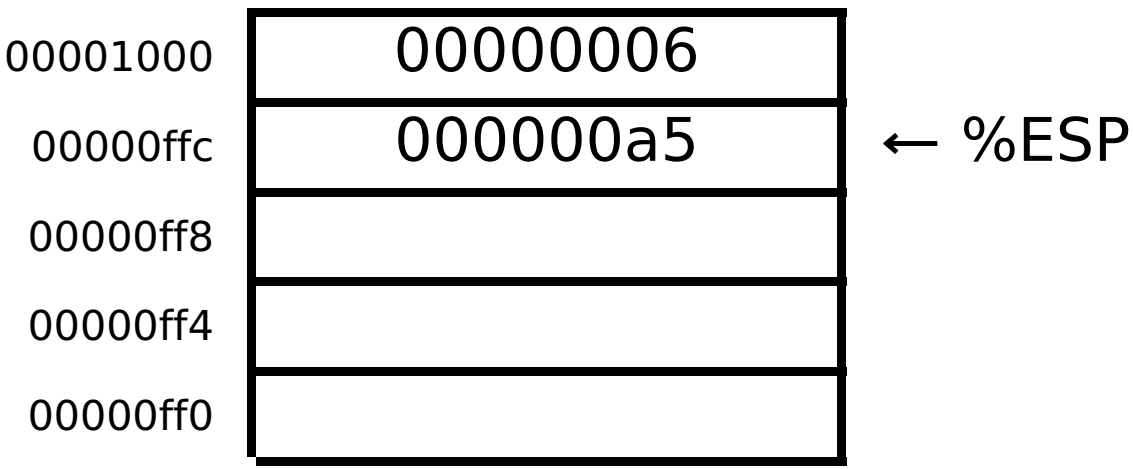


Av historiske grunner vokser stakken mot *lavere* adresser.

Å legge elementer på stakken

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

```
    pushl   $0x000000a5
```



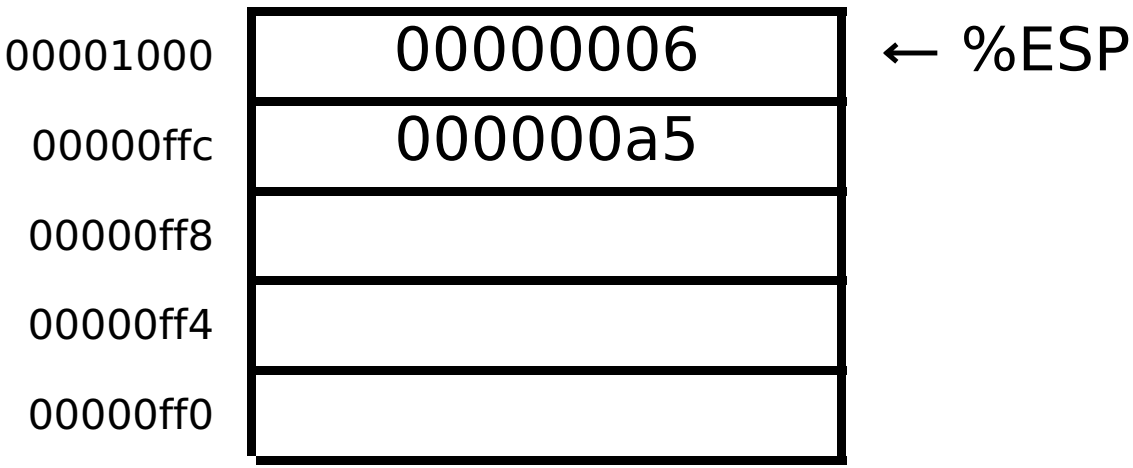
Legg merke til at vi kan få tak i alle elementene på stakken:

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

Å fjerne elementer fra stakken

Til dette brukes **popw** og **popl**:

```
popl    %eax
```



Verdiene blir ikke fysisk fjernet.

Rutiner

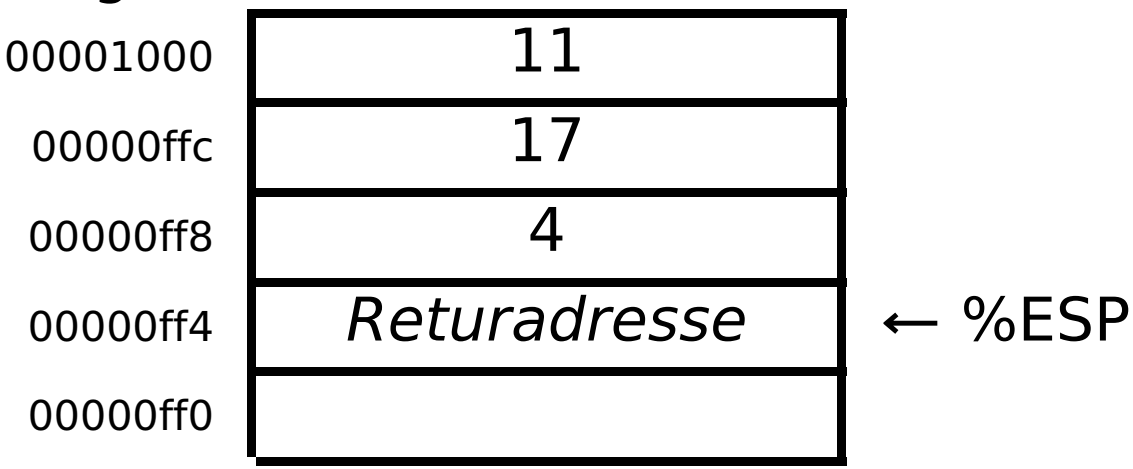
Ved et rutinekall skjer følgende:

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

Kallet

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

vil gi denne stakken:



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?

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.

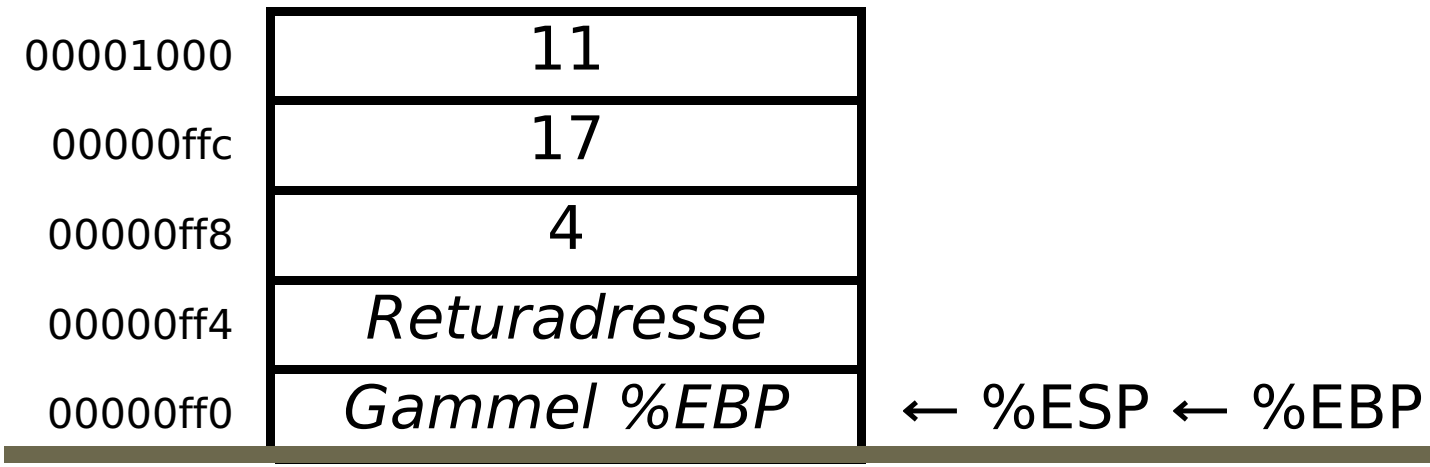
En forbedring

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 %EBP til å peke på parametrene:

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



Nå er paramtrene 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.

```
                .globl  mystradd

# Name:         mystradd.
# Synopsis:     Legger et tegn til en C-tekst.
# C-signatur:   void mystradd (char *s, char c)
# Registre:     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.
```