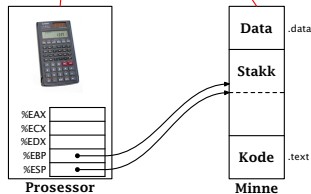
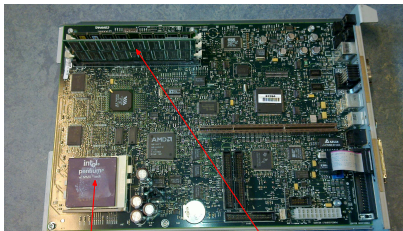


Det viktigste i en moderne datamaskin er *hovedkortet* («motherboard»):

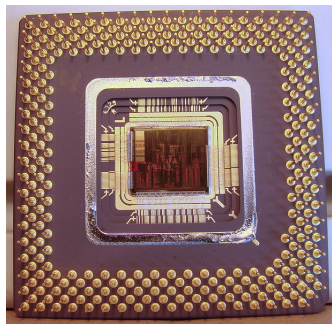


Grovt sett inneholder et hovedkort

- En prosessor
- Minne (for både program og data)
- Klokke
- Kontrollere for periferutstyr.

Prosessoren

Prosessoren er «selve datamaskinen».



Det finnes mange ulike prosessorer fra ulike produsenter. I dette kurset skal vi se på **x86** (= **IA-32**) fra *Intel*, en 32-bits prosessor.

Det finnes en hel familie med x86-prosessorer fra Intel og andre, fra Intels 8086 (fra 1987) til dagens Intel Haswell. Vi skal konsentrere oss om **Pentium 4** fra 2000.

Prosessoren

En prosessor inneholder:

- En ALU («Arithmetic and Logic Unit») eller flere
- Register
- Kontrolllogikk

(I dag ligger det også coprosessorer, *cache* og mye annet på prosessorbrikken, men dette er logisk sett ikke del av prosessoren.)

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 |

Spesielle 16-bits

| |
|-----|
| %CS |
| %DS |
| %ES |
| %FS |
| %GS |
| %SS |

Spesielle 32-bits

| |
|---------|
| %EFLAGS |
| %EIP |

Assembler-programmering

Denne lille C-funksjonen returnerer verdien 19:

```
int nineteen (void)
{
    return 19;
}
```

nineteen.c

Denne assembler-funksjonen gjør det samme:

```
    .globl  nineteen
nineteen:
    movl   $19, %eax
    ret
```

nineteen.s

Den legger verdien 19 i register %EAX der alle funksjoner legger resultatverdien, og returnerer.



Et testprogram

Dette programmet kan teste funksjonen:

```
_____ test-nineteen.c _____  
#include <stdio.h>  
  
extern int nineteen (void);  
  
int main (void)  
{  
    printf("nineteen() = %d\n", nineteen());  
    return 0;  
}
```


Programmet gcc kan håndtere både C-kompilering og assemblering:

```
$ gcc -m32 test-nineteen.c nineteen.s -o test-nineteen
```

NB!

Ikke glem opsjonen **-m32!**

Det ferdige programmet kan nå kjøres:

```
$ ./test-nineteen  
nineteen() = 19
```

Maskinkode er nesten det samme som assemblerkode

Maskinkode kontra assemblerkode

Programmet ligger lagret som *maskinkode*, dvs bit-mønstre.¹

nineteen.list

GAS LISTING nineteen.s

page 1

```
1  
2          .globl  nineteen  
3  nineteen:  
3  0000 B8130000    movl   $19, %eax  
3  0000 00000000  
4  0005 C3         ret
```

DEFINED SYMBOLS

```
    nineteen.s:2      .text:0000000000000000 nineteen
```

NO UNDEFINED SYMBOLS

Siden numeriske koder er vanskelige å huske, brukes *assemblerkode* som bare er huskekoder.

¹Ved å skrive `gcc -m32 -Wa,-a -c minfil.s >minfil.list` kan vi se maskinkoden.

Assemblerkode kontra høynivåprogrammering

Det er stor forskjell på assemblerkode og høynivåspråk:

Kompilering av høynivåspråk

- Vi vet ikke hvilke maskininstruksjoner som genereres (men er ikke interessert).
- Programmet kan (stort sett) flyttes uendret til en annen datamaskin uansett fabrikat eller operativsystem.

Assemblering

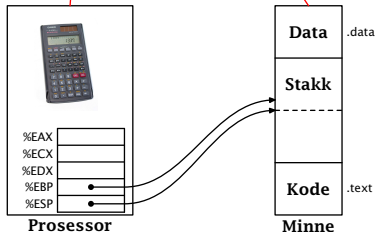
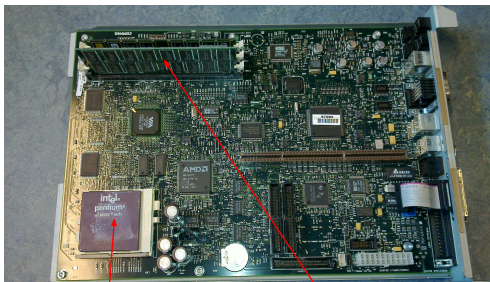
- Vi vet nøyaktig hvilke maskininstruksjoner som genereres.
- Programkoden kan ikke flyttes til datamaskiner med annet instruksjonssett eller operativsystem.



Parametre

Parametre ligger på *stakken* så man får tak i dem med

«**4(%esp)**»,
«**8(%esp)**», ...



Parametre

incr.s

```
incr:      .globl incr
           movl      4(%esp),%eax
           incl     %eax
           ret
```

test-incr.c

```
#include <stdio.h>
extern int incr (int n);

int main (void)
{
    int i;

    for (i = 10; i <= 14; ++i)
        printf("incr(%d) = %d\n", i, incr(i));
    return 0;
}
```

Resultatet av
kjøringen:

```
incr(10) = 11
incr(11) = 12
incr(12) = 13
incr(13) = 14
incr(14) = 15
```

Et eksempel til

Denne funksjonen har to parametre:

```
_____ add.s _____  
      .globl add  
add:  movl    4(%esp),%eax  
      addl    8(%esp),%eax  
      ret
```

Parametre

test-add.c

```
#include <stdio.h>

extern int add (int a, int b);

int tab[] = {1, 17, -3};

int main (void)
{
    int tab_length = sizeof(tab)/sizeof(int);
    int i1, i2;

    for (i1 = 0; i1 < tab_length; ++i1) {
        for (i2 = 0; i2 < tab_length; ++i2) {
            int a1 = tab[i1], a2 = tab[i2];

            printf("add(%d,%d) = %d\n", a1, a2, add(a1,a2));
        }
    }
    return 0;
}
```

```
add(1,1) = 2
add(1,17) = 18
add(1,-3) = -2
add(17,1) = 18
add(17,17) = 34
add(17,-3) = 14
add(-3,1) = -2
add(-3,17) = 14
add(-3,-3) = -6
```



Oppsummering

Vi kjenner nå til følgende instruksjoner:

| | |
|-------------------|-----------------------------|
| <code>movl</code> | Flytt en verdi |
| <code>addl</code> | Adder en verdi til en annen |
| <code>subl</code> | Subtraher |
| <code>incl</code> | Øk en verdi med 1 |
| <code>decl</code> | Senk en verdi med 1 |
| <code>ret</code> | Returner fra funksjon |

Operander kan være:

| | |
|----------------------|------------------------|
| <code>\$17</code> | Konstanter |
| <code>%eax</code> | Registeret %EAX |
| <code>4(%esp)</code> | Parametre (på stakken) |

Ikke alle vil gjøre det på samme måte

Et par andre småting

Kjøring på Windows- og Mac-maskiner

Her brukes en litt annen notasjon for globale navn:

```
        .globl _nineteen
_nineteen:
        movl    $19, %eax
        ret
```

Heldigvis kan man bruke begge notasjonene i samme fil:

```
        .globl nineteen
        .globl _nineteen
nineteen:
_nineteen:
        movl    $19, %eax
        ret
```



Ikke alle vil gjøre det på samme måte

Ulik notasjon

Dessverre benytter ulike assemblere forskjellig notasjon.

AT&T brukes i Linux-miljøet og *i dette kurset*.

Intel brukes av Intel og Microsoft.

| | Intel | AT&T |
|-----------------------|-------------------------|-------------------------|
| Konstanter (desimale) | 4 | <u>\$</u> 4 |
| Konstanter (hex) | 123 <u>h</u> | <u>\$0x</u> 123 |
| Registre | eax | <u>%</u> eax |
| Rekkefølge | <i>res, op, op, ...</i> | <i>op, op, ..., res</i> |
| Størrelse | mov | mov <u>l</u> |
| Indeksering | <u>[eax+1]</u> | <u>1(%eax)</u> |

