En oppsummering (og litt som står igjen)

- Pensumoversikt
- Hovedtanker i kurset
- Selvmodifiserende kode
- Overflyt
- Eksamen

Oppsummering

- **Pensum**
  - Læreboken til og med kapittel 7
  - kompendiet
  - forelesningene
  - de obligatoriske oppgavene
  - øvingsoppgavene

(Forelesningene, de obligatoriske oppgavene og ukeoppgavene demonstrerer hva vi legger vekt på.)

Hvorfor har dere lært dette?

Ikke mange programmerer assemblerkode i dag, men

- noen gjør det
- noen må jobbe med kompilatorer som lager slik kode
- kjennskap til slik kode er en forutsetning for å forstå hvordan datamaskinen er bygget opp og fungerer
- kjennskap til maskinkode gjør oss til bedre programmere i høynivåspråk

Vi har lært mest om Intel x86-kode, men *prinsippene* er de samme for alle prosessorer.

Hovedtanke

Glem aldri!

Alt er *bit*!

Selvmodifiserende kode

Når programkode lagres som bit-mønstre, kan man da la programmet endre på seg selv?

```
 1                                      .globl  teller
 2                                      .data
 3 0000 55                      teller: pushl   %ebp
 4 0001 89E5                            movl    %esp,%ebp
 5
 6 0003 B8010000                        movl    $1,%eax
 6      00
 7 0008 83050400                        addl    $1,teller+4
 7      000001
 8
 9 000f 5D                              popl    %ebp
10 0010 C3                              ret
```

Denne funksjonen returnerer 1 første gang den kalles. Samtidig endres instruksjonen slik at den vil gi 2 neste gang den utføres osv.

- Koden er plassert i `.data` for å kunne endres.
- På noen maskiner vil det kunne bli rot med data- og instruksjons-cache.

Konklusjon

Det er morsomt at det går an, men slik kode kan neppe kalles hverken lettlest eller trygg.

Overflyt

```
#include <stdio.h>

int main (void)
{
  signed char v = 100;
  v = v + v;
  printf("v = %d\n", v);
  return 0;
}
```

$$100 = 01100100_2$$
$$200 = 11001000_2 = -56$$

gir galt svar:

```
v = -56
```

Feilen skyldes overflyt. Hva kan man gjøre med slikt?

- Strutseteknikken
- Sjekke data før operasjonen
- Sjekke om operasjonen gikk bra

Heltall *uten* fortegns-bit

- For addisjon og subtraksjon settes $\mathcal{C}$-flagget ved overflyt.
- Ved multiplikasjon kommer svaret med dobbelt så mange bit. Da kan vi sjekke øverste 32 bit av svaret.
- Ved divisjon kan det bli et avbrudd!

```
        .globl  ovfl
ovfl:   movl    mill,%eax
        imull   mill
        idivl   ti
        ret

        .data
mill:   .long   1000000
ti:     .long   10
```

gir

```
Floating point exception
```

Heltall *med* fortegns-bit

- Ved addisjon kan man bruke $\mathcal{O}$-flagget som settes ved overflyt. Nærmere bestemt settes det når
  1. begge operandene har likt fortegns-bit *og*
  2. resultatet har motsatt fortegns-bit.
- Tilsvarende skjer ved subtraksjon om
  1. operandene har ulikt fortegn *og*
  2. resultatregisteret skifter fortegn.
- Multiplikasjon og divisjon er som for tall uten fortegns-bit.

Konklusjon

Ved behov kan man bruke assemblerprogrammering til å sjekke på overflyt.

Forventet kunnskap til min del av eksamen

- oppbygningen av datamaskiner (registre, flagg, minne, stakk, . . .)
- bit-lagring av instruksjoner, heltall (av ulike størrelser, ± fortegns-bit), flyt-tall, C-tekster, . . .
- notasjon for assemblerkode (registre, konstanter, adresser, . . .)
- instruksjoner for dataflytting og -testing, regning, masking, hopp (± testing), skifting/rotasjoner, . . .
- implementasjon av datastrukturer (vektorer, lister, C-tekster, `struct`-er, `union`-er, . . .)
- funksjoner og kall på disse
- overflyt, selvmodifiserende kode

Hva slags oppgaver kan forventes fra meg?

- Programmering i C og assembler, for eksempel oversettelse den ene eller andre veien.
- Finne ut hva et program gjør.

Hva er viktig?

- Overbevise sensor om at man har skjønt *ideen* bak stoffet.
- Gode *korte* kommentarer.
- Enkel kode.

Hva er ikke så viktig?

- Syntaksdetaljer
- Rask kode