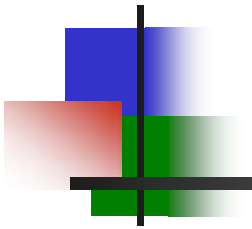


Kodegenerering, del 2:
Resten av Kap. 8
pluss tilleggsnotat (fra kap. 9 i ”ASU”)
INF5110 – V2007

Stein Krogdahl,
Ifi UiO

NB: Innfører noen begreper som først og fremst har mening om man skal gå videre med optimaliseing:

- Generell ”liveness”
- ”Løkker” i flyt-grafer
- Samt noen typiske adresserings-måter i maskin-språk

Rekursiv prosedyre for P-kodegenerering (hører til kap. 8, Louden)

Hit skal en "break" i kildeprogr. gå

```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
      if (t->val==0) emitCode("ldc false");
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);   Rek. kall
      lab1 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab1);
      emitCode(codestr);
      genCode(t->child[1],label);   Rek. kall
      if (t->child[2] != NULL)
      { lab2 = genLabel();
        sprintf(codestr,"%s %s","ujp",lab2
        emitCode(codestr);}
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      if (t->child[2] != NULL)
      { genCode(t->child[2],label);   Rek. kall
        sprintf(codestr,"%s %s","lab",lab2
        emitCode(codestr);}
      break;
```

```
    case WhileKind:
      lab1 = genLabel();
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      genCode(t->child[0],label);   Rek. kall
      lab2 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab2);
      emitCode(codestr);
      genCode(t->child[1],lab2);   Kode for S  Rek. kall
      sprintf(codestr,"%s %s","ujp",lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);
      break;
    case BreakKind:
      sprintf(codestr,"%s %s","ujp",label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
  }
```

Pensumoversikt - kodegenerering

8.1 Bruk av mellomkode

8.2 Basale teknikker for kode-generering

8.3 Kode for referanser til datastrukturer (ikke 8.3.2)

8.4 Kode for generereing for kontroll-setninger og logiske uttrykk

(8.5 Kode-generering for prosedyrer og kall)

--------- (resten ikke pensum) ------

8.6 Kode produsert av to kommersielle kompilatorer

8.7 TM: En enkel maskin

8.8 Kode-gen for Tiny på TM

8.9 og 8.10 Optimalisering

Også pensum:

En del fra utdelt kap 9 fra Aho, Sethi og Ullmann ' s kompilatorbok

(ASU: "Drage-boka") :

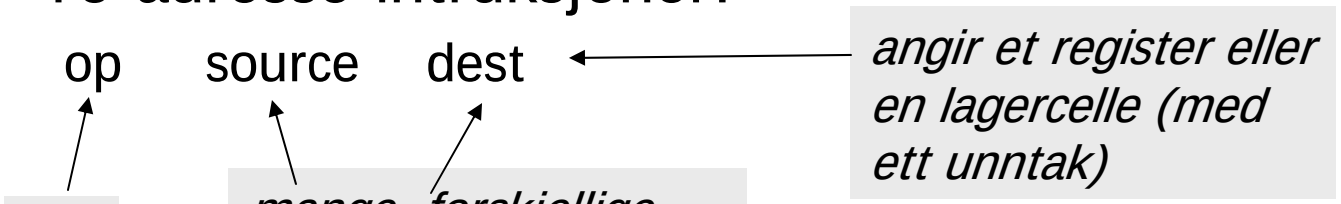
9.4, 9.5 og 9.6

Pluss kanskje litt om

Javas Byte-kode

Maskinen det oversettes til i kap. 9, ASU

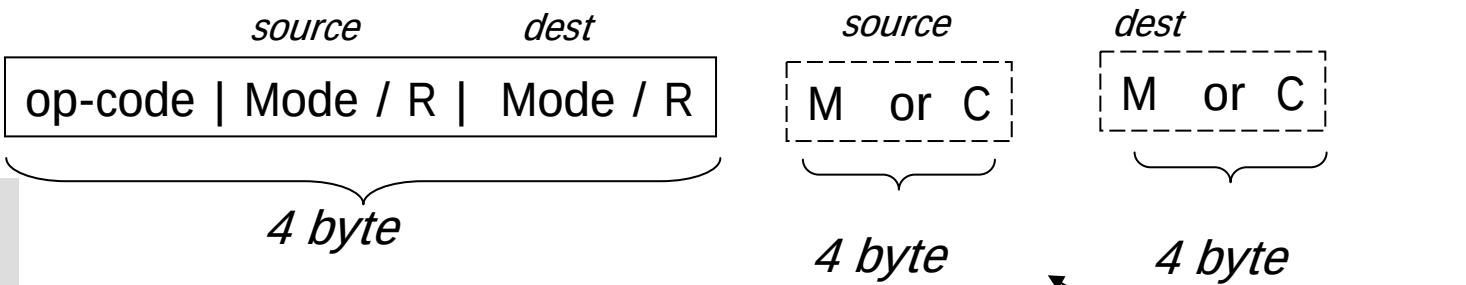
- To-adresse-intruksjoner:



ADD a b
SUB a b ← Legger *b-a* i *b*
MUL a b
……
GOTO l
+ Betingete hopp
+ Prosedyrekall
++

Mer er en CISC-maskin enn en RISC-maskin

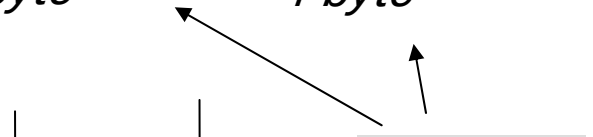
Instruksjonsformat og adressemodi – del 1



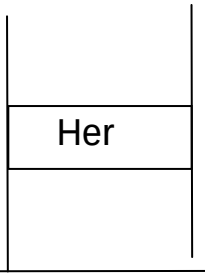
Boka bruker total størrelse av instruksjonene som tidsmål for eksekveringen. Grunnkostnad for én instruksjon er 1

Tilleggskostnad for adressering

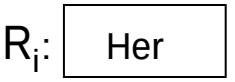
Adresseringsmodi:



1 Absolutt: M

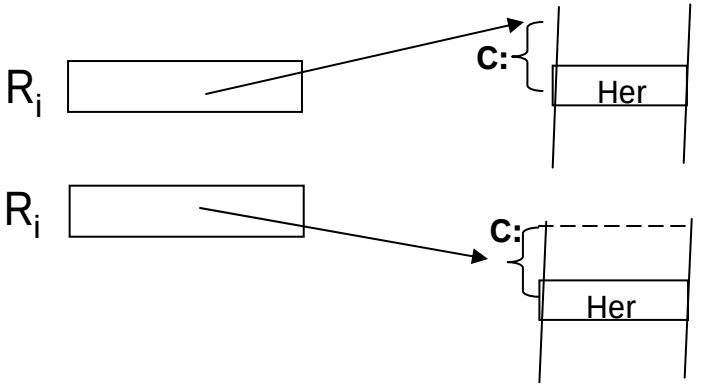


0 Register: Ri

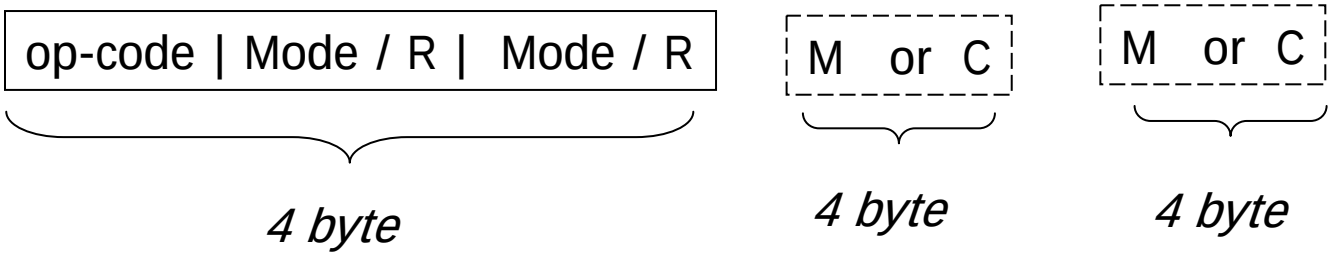


1 Indeksert : C(RI)

To måter å bruke dette på:

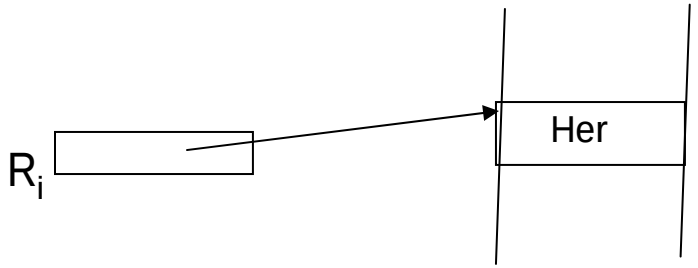


Instruksjonsformat og adressemodi – del 2

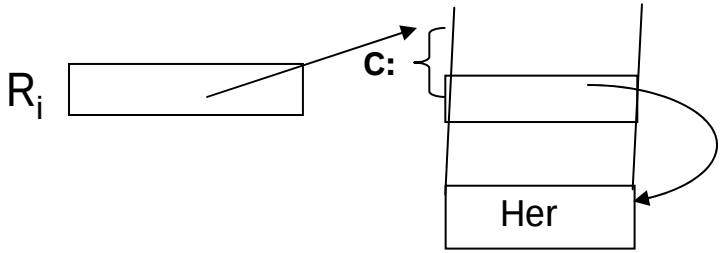


Adresseringsmodi:

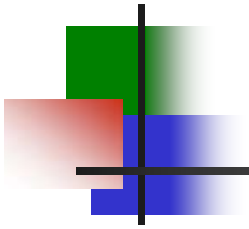
0 Indirekte Register *R



1 Indirekte *C(Ri)



1 Literal #M (NB: Bare for source)

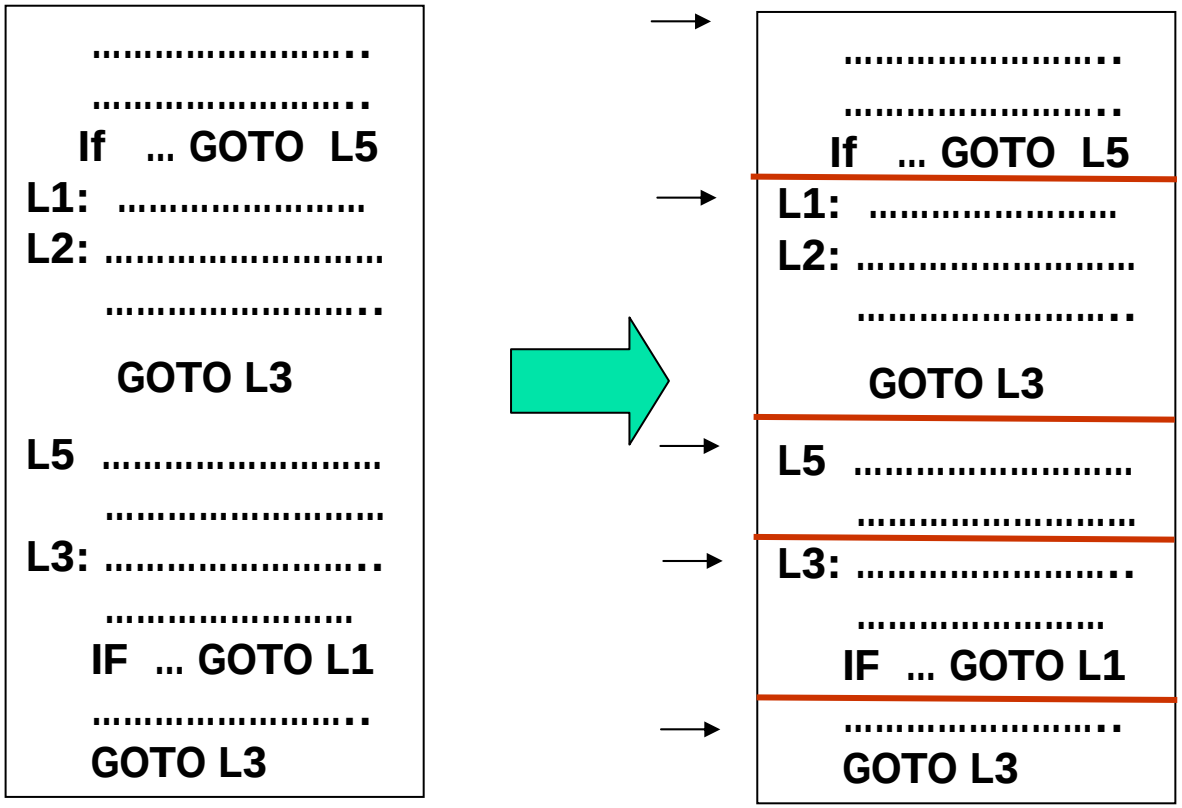


9.4 Basale blokker og Flyt-grafer

- En Flyt-graf er en graf der nodene er sekvenser av treadresse-setninger som alltid ”utføres sammen”
- Nodene i grafen kalles Basale Blokker
  - En basal blokk representer en sekvens av beregenings-instruksjoner
  - Programkontrollen kommer alltid inn i første setning og går sekvensielt gjennom hver setning uten stopp eller sidesprang. Eneste unntak er at siste setning kan være et hopp – også betinget.
- Kantene i grafen representerer de mulige veier programflyten-kontrollen kan ta
- Innen en Basal Blokk er det lett å holde oversikt over hvor ting er etc,. og lett å gjøre ”abstrakt interpretasjon”.

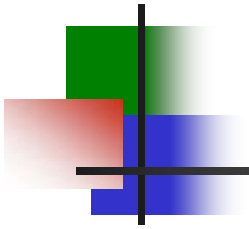
Eksempel på oppdeling i basale blokker

- Basale blokker: Fra og med en ”leder” fram til neste
- Algoritme:
  - Første setning er leder
  - en ”goto i” (betinget eller ikke) gjør setning ”i” til en leder
  - setninger etter ”goto ..” (betinget eller ikke) er ledere



Det er tydeligvis ingen som går til L2.

Metodekall tenker vi ikke på. Kan behandles litt forskj. avh. av formål.



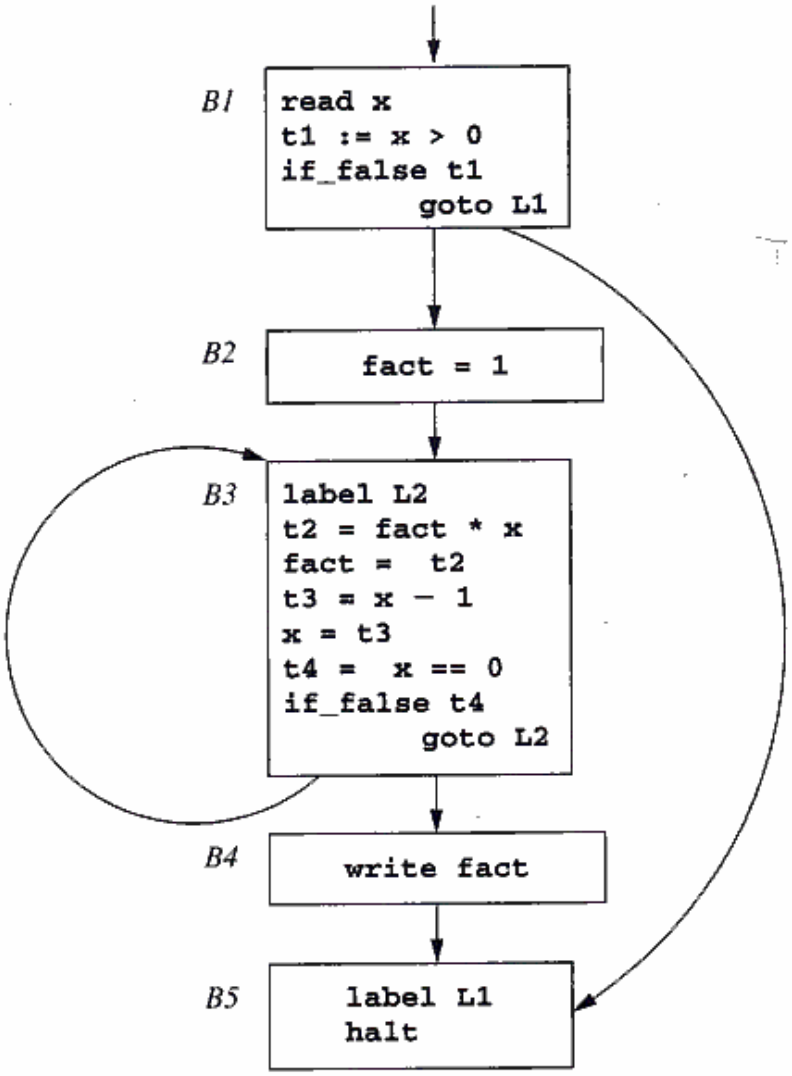
Flyt-graf

- Nodene i flyt-grafen er de basale blokkene med en angitt start-blokk (den første basale blokken i programmet)

- Det er en rettet kant fra blokk B_1 til B_2 hvis B_2 kan følge direkte etter B_1 i en eller annen utførelse. Det vil si at det enten er:
  - En betinget eller ubetinget goto fra siste setning i B_1til første setning i B_2 eller
  - B_2 følger direkte etter B_1 i programmet, og B_1 ender ikke i en ubetinget goto.

- Vi sier at B_1 er en forgjenger til B_2 og B_2 er en etterfølger til B_1

Flytgraf fra Louden 8.9
oftest: En flytgraf for hver metode for hver metode

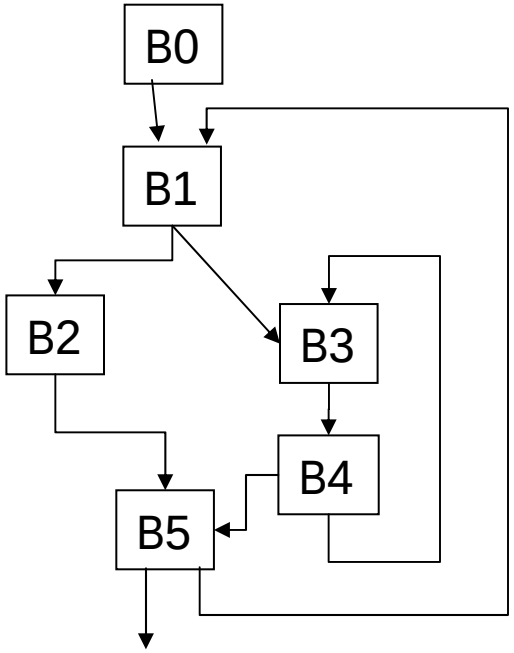


En goto-setning eller if-goto-setning vil alltid være siste setning setning i sin basale blokk (men ikke alle basale blokker slutter slik!)

- Metodekall kan plasseres litt forskjellig avh. av grafens bruk
  - Danne egne basale blokker
  - Kan ligge først i en basal blokk?

Løkker i flyt-grafer

- Bruk, for eksempel :
  - Kan vi flytte beregninger ut av løkka?
    while (i<n) { i++; A[i] = 2*k; }
  - Kan kanskje holde mye brukte variable i registre mens vi er i løkka



En løkke er et utplukk L av noder slik at:

1) Dersom Bx $\in$L og By $\in$L, så går det en rettet vei fra Bx til By av lengde $\geq$ 1 *(også om Bx og By er samme node!)*

2) L har bare én "inngang": Det finnes bare én B$\in$L slik at Bn $\rightarrow$ B og Bn$\notin$L.

Begrunnelse rent praktisk: Ett sted å initialisere løkka & Ett sted om vi skal flytte noe "ut av løkka"

Eksempler:

{B3,B4} og {B1,B2,B3,B4,B5} er løkker

{B1,B2,B5} er ikke løkke (!?)

Hva er ”liveness” (”i live”) ?

- Begrepet er uavhengig av basale blokker
- Defineres i 9.4 og brukes i 9.5
  - Terminologi:

```
a := x + y;
if (x<a) goto L;
```

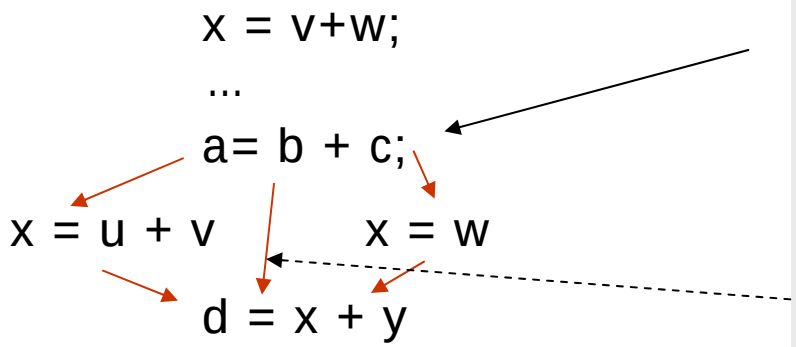
Her ”defineres” a, og ”brukes” x og y

Her ”brukes” x og a.

Intuitiv definisjon:

En variabel x er ”levende” (eller ”i live”) på et gitt sted i programmet dersom den verdien den der har kan bli brukt senere i en eller annen utførelse.

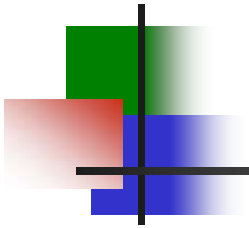
Definisjon som kan avgjøre om x er levende



Stedet ”i”
Er ”x” i live her ?

Svaret er ”ja”, fordi det finnes en TA-setning ”j” slik at det er minst én eksekveringsvei fra ”i” til ”j” uten noen tilordning til (eller definisjon av) ”x”.

Definisjon: En variabel som ikke er ”i live” på et gitt punkt, sies å være ”død” på dette punktet (og verdien kan da ”kastes”)



Andre def. som kan være interessant for optimalisering

Kalles global dataflyt-analyse. Eksempler:

- Gitt en TA-instruksjon der x brukes:
  - Finn alle de tilordninger (definisjoner) der denne verdien på x kan være satt
- Gitt en tilordning der x blir satt:
  - Finn alle de steder der denne verdien av x kan bli brukt

Disse og liknende sammenhenger kan ”lett” bergenes ved en iterasjons-algoritme på de basale blokkene.