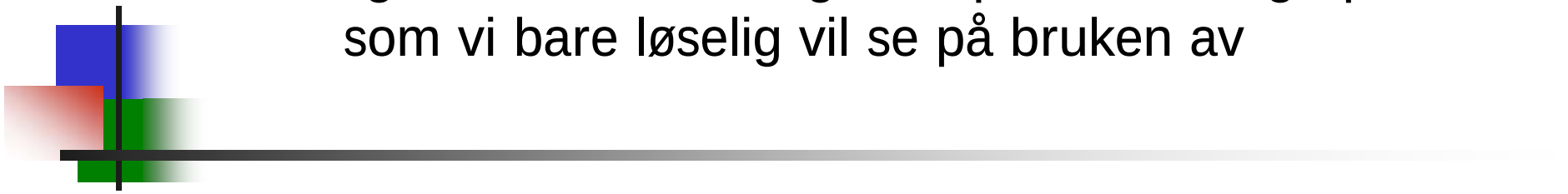
INF5110 – 7. mai 2014

Stein Krogdahl, Ifi, UiO

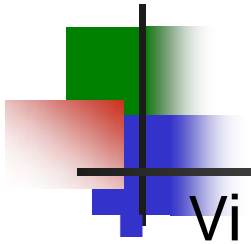
NB: I dagens stoff skal vi også se på en del begreper som vi bare løselig vil se på bruken av



Dette er foiler til:

Tilleggsnotat fra bok av Aho, Sethi og Ullman (ASU)
Notatet ligger på undervisnings-planen.
Beklager dårlig kopi!

Som bakgrunn er det lurt å lese kap. 8.9 i boka



Ting å tenke på når man skal oversette til maskininstruksjoner for en gitt maskin

- Vi tenker her at vi skal oversette :
  - Fra den type TA-kode (3A-kode) vi har sett på til et maskinspråk der instruksjonene har to adresser (2A-kode)
  - Og at maskinen har et (ikke veldig stort) antall “registre”, som er mye raskere å aksessere enn data i hovedlageret
  - Dette siste er nå litt “tilkludret “ pga. bruk av diverse cacher

- Viktige ting og spørsmål:
  - Man må sette seg godt inn i maskinkoden og dens finesser
  - Hvordan takle problemet med å gå fra 3A-kode til 2A-kode?
  - Hvor store biter av programmet bør man se på av gangen?
  - Bestemme løpende hvor data skal ligge under utførelsen.
  - Hvordan avgjøre hva som er beste maskinkodesekvens om man har flere alternativer?
  - Få oversikt over hvilke data som vil bli brukt videre i programmet, og eventuelt hvor snart de skal brukes.

Maskinen det oversettes til i notatet

Detaljer ikke viktige, bare eksempel på typisk maskin

- To-adresse-instruksjoner:

op source dest

hva

mange forskjellige ulike former, se neste foil

Hver av *source* og *dest* angir

- Enten et register
- Eller en lagercelle
- *Source* kan også angi en konstant

ADD a b
SUB a b

Merk: Her beregnes b – a og svaret legges i b.

NB: I maskinen skissert på s. 12 i Louden er det *omvendt* !

MUL a b
.....
GOTO l
+ Betingete hopp
+ Prosedyrekall
++

Complex Instruction Set Computer

Reduced Instruction Set Computer

Er mer en CISC-maskin enn en RISC-maskin

Hvordan måle eksekv.tiden for en instruksjon?

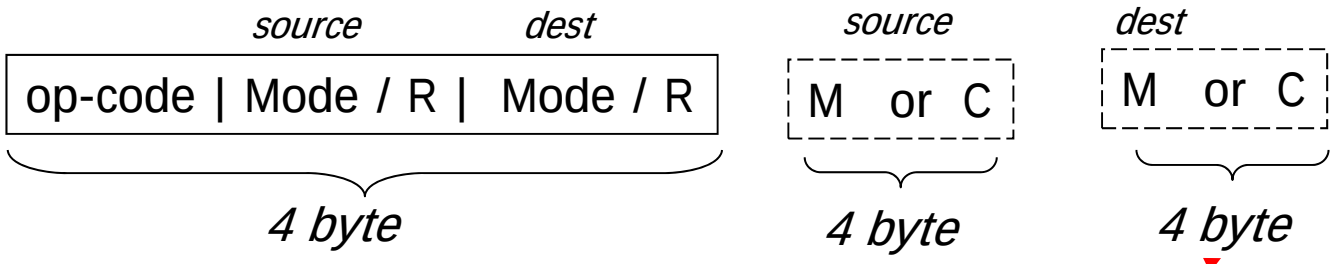
- Det som er brukt som eksekveringstid i notatet er rett og slett *lengden av instruksjonen*.
  - Dette kan virke litt merkelig siden det jo er utføringen, etter at instruksjonen er hentet opp, vi vanligvis tenker på som "tiden instruksjonen tar".
- Men ofte er det slik at det tar minst like lang tid å *hente* instruksjonen som å *utføre* den.
  - Og det er også vanlig at en lang instruksjon også tar lenger tid å utføre.
  - Men man kan selvfølgelig også bruke andre "godhetsmål" for kodesekvenser, f.eks.:
    - Hvor mange ganger man må gå til hovedlageret
    - Kanskje tar noen instruksjoner spesielt lang tid
  - Hvor lang tid ting faktisk tar er nå uberegnelig, pga. caching m.m.
- I notatet vil en instruksjon ta tid (eller ha "kost"):

  1 (1x4 byte), 2 (2x4 byte) eller 3 (3x4 byte)

Instruksjonsformat og adressemodi – del 1

Kan se dette som eksempler på typiske adresserings-former

Grunnkost for en fire-bytes instruksjon er 1

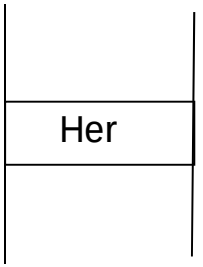


Bare om M eller C er med i adresseringen

Tilleggskost for denne typen adressering

Adresseringsmodi:

1 Absolutt: M



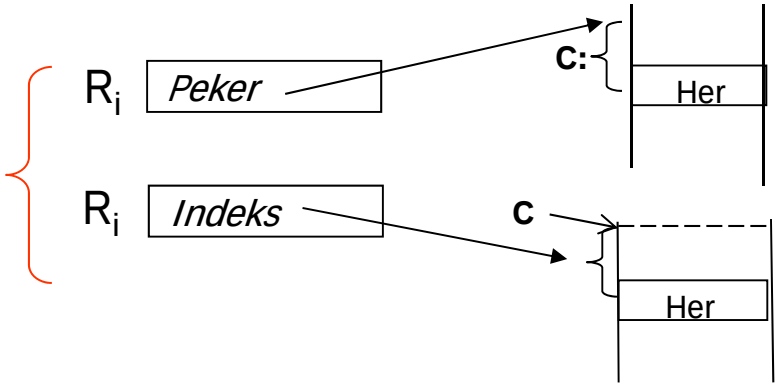
0 Register: R_i R_i Her

1 Indeksert : $C(R_i)$

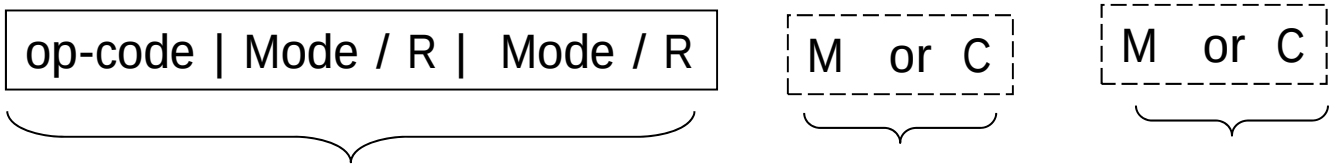
Grei på to måter:

(1) Til å la R inneholde en objekt-peker og la C være en kompilatorkjent relativadresse i objektet

(2) Til å la C være peker til en fastliggende array, og la R ha en indeks inn i denne

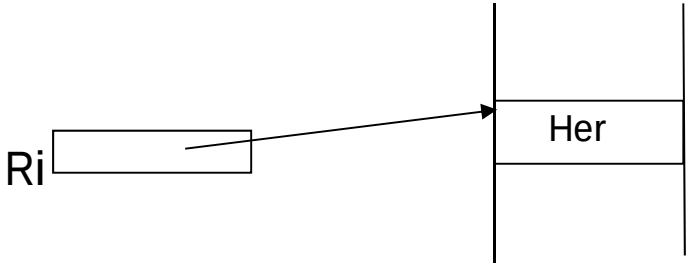


Instruksjonsformat og adressemodi – del 2



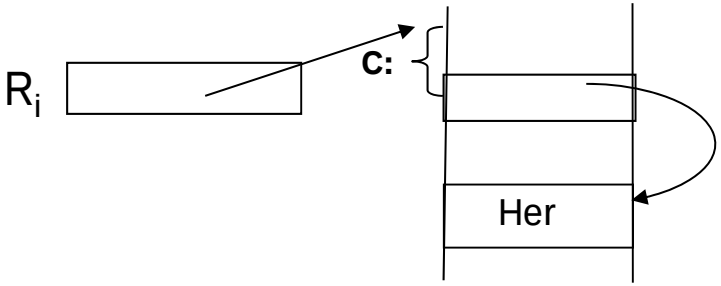
Tilleggskost for denne typen adressering

0 Indirekte Register: *R



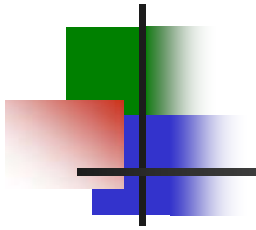
1 Indirekte: *C(Ri)

Kan brukes til å hoppe to steg av gangen langs en linket liste, f.eks ved følging av "lang" access link



1 Konstant: #M bare for source (spesiell, men veldig brukbar!)

Legger verdien M inn på sted angitt av *dest*



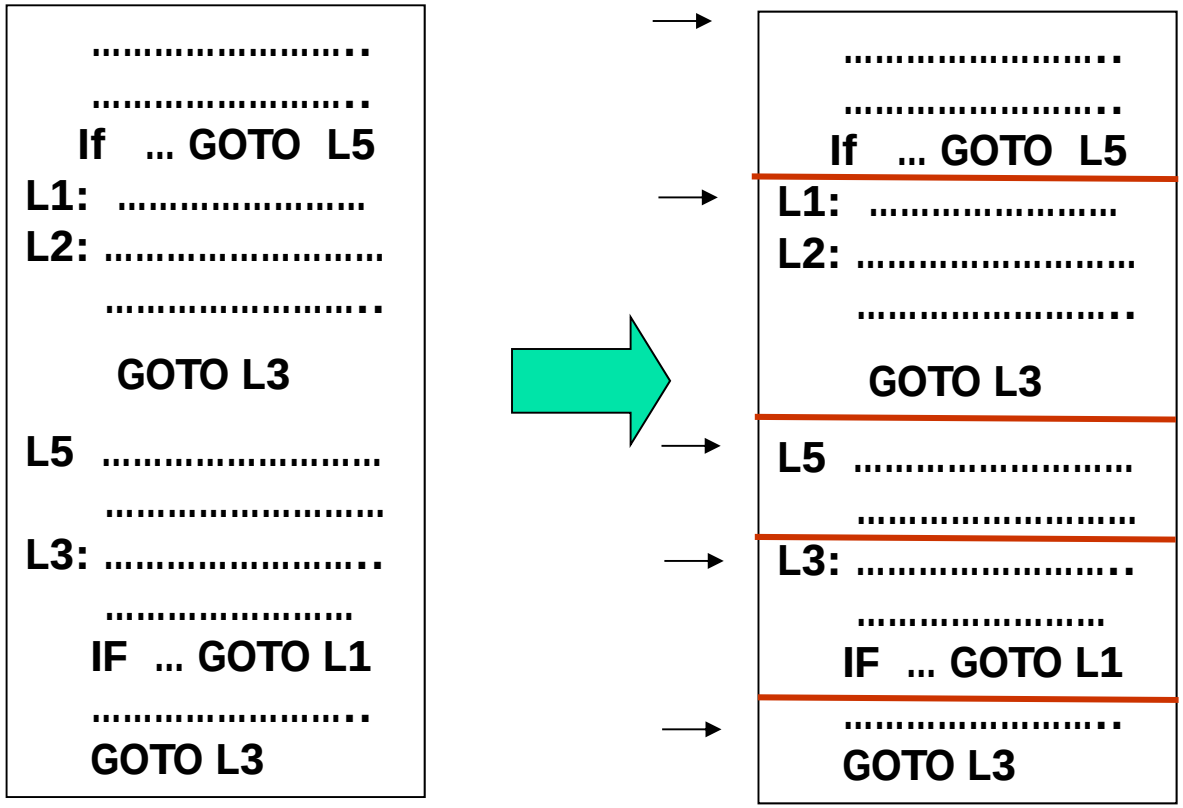
9.4 Basale blokker og Flyt-grafer

- En Basal Blokk er en del av programmet der alle instruksjonene vil bli utført etter hverandre, uten uthopp eller innhopp.
  - Innen en slik blokk står kompilatoren fritt til å lagre verdier der den finner det fornuftig (bare de settes tilbake etterpå)
  - Den kan lett skaffe oversikt over hvilke verdier og variable som inngår i blokken, og hvordan disse er avhengig av hverandre
  - Man kan da utnytte dette til kodegen. vha. ”abstrakt interpretasjon” eller ”statisk simulering” (noe vi så på da vi oversdatte fra P-kode til TA-kode).

- En Flyt-graf er en rettet graf der:
  - Nodene er de basale blokkene
  - Kantene representerer de mulige veier programflyten kan ta mellom de basale blokkene

Oppdeling i basale blokker

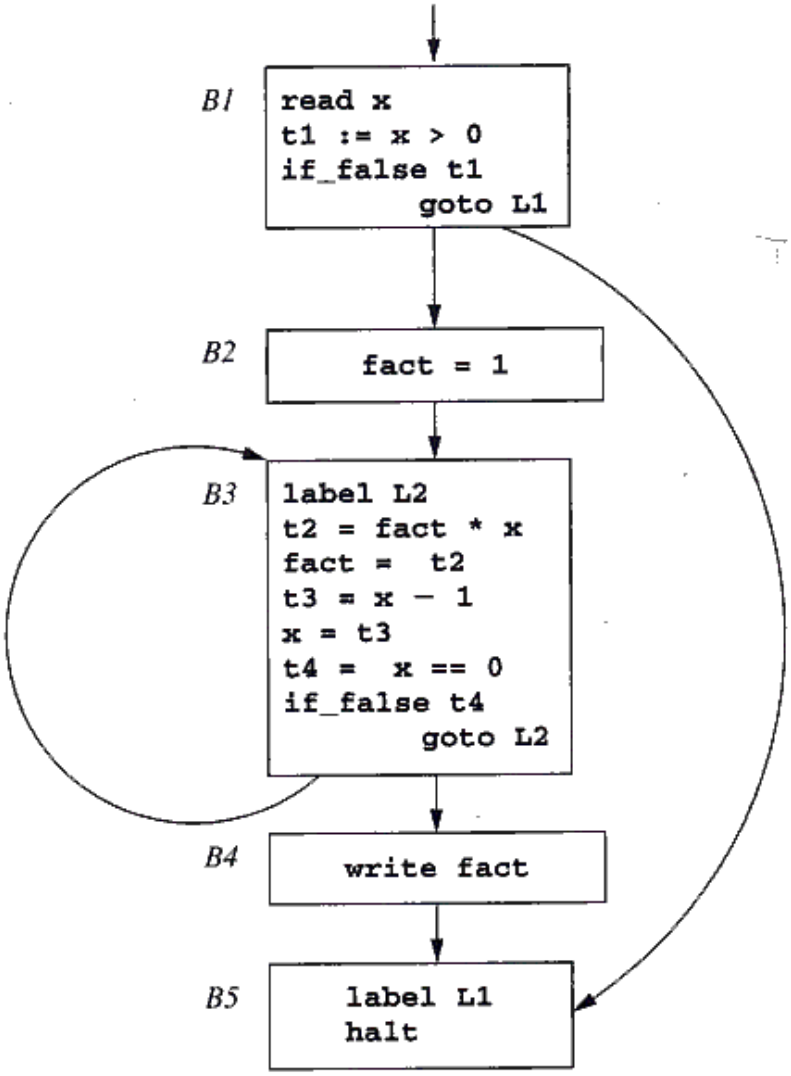
- Algoritme for å finne alle «ledere»:
  - Første setning er leder
  - en ”goto i”, gjør setning ”i” til en leder
  - setninger etter ”goto ..” er ledere
- Basale blokk: Fra og med en leder fram til neste, eller til slutten



Det er tydligvis ingen som går til L2!

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

Flytgraf fra Louden 8.9
Oftest: En flytgraf for hver metode



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

Metodekall kan passes inn i dette på litt forskjellig måter, avh. av flyt-grafens bruk.

Vi ser ikke på det i pensum

Kode kan analyseres og arbeides med (f.eks. til optimalisering) på tre nivåer:

- Inne i én basal blokk
- Hele flytgrafen for én metode ("globalt nivå")
- Alle flytgrafene for hele programmet

Løkker i flyt-grafer

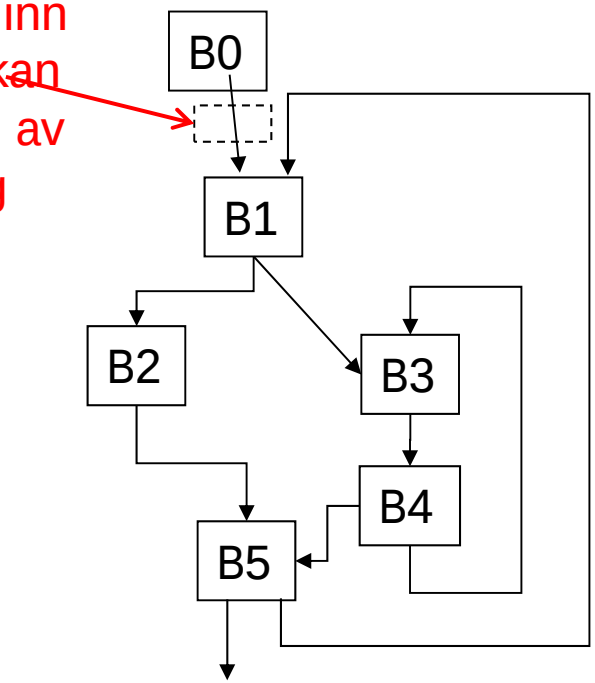
- Typisk bruk, for eksempel :

```
while (i<n) { i++; A[i] = 3*k; }
```

  - Kan vi flytte beregninger ut av løkka? Ja, beregning av ”3*k”
  - Kan kanskje holde mye brukte variable i registre mens vi er i løkka? F.eks. variablen ”i”

Er {B1,B2,B5} en lovlig løkke?
Nei, ikke ut fra definisjonen. Og den er slik fordi: Anta at vi vil sjekke om en gitt variabel *k* blir forandret et sted i løkka. Da kunne vi få feil svar om vi bare sjekket i B1, B2 og B5. Blokkene B3 og B4 må også sjekkes.

Ny basal blokk der vi kan putte inn ting som kan «flyttes ut av løkka», og gjøres på forhånd (f.eks. k’=k*3)



En løkke er et utplukk L av noder slik at:

1. Alle-til-alle-vei: Dersom Bx $\in$L og By $\in$L, så går det en rettet vei fra Bx til By av lengde ≥ 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 der Bn$\notin$L.

Begrunnelse for punkt 2. er rent praktisk: Ett sted å initialisere løkka og ett sted om vi skal flytte noe ”ut av løkka” (stiplet boks).

Eksempler: {B3,B4} og {B1,B2,B3,B4,B5} er løkker (men ikke {B1,B2,B5}, se øverst)

Hva er ”liveness” (”i live”) ?

- Begrepet er *uavhengig* av basale blokker (*ikke* klart i notatet)
- Begrepet forklares i 9.4 og brukes i 9.5
  - Terminologi:

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

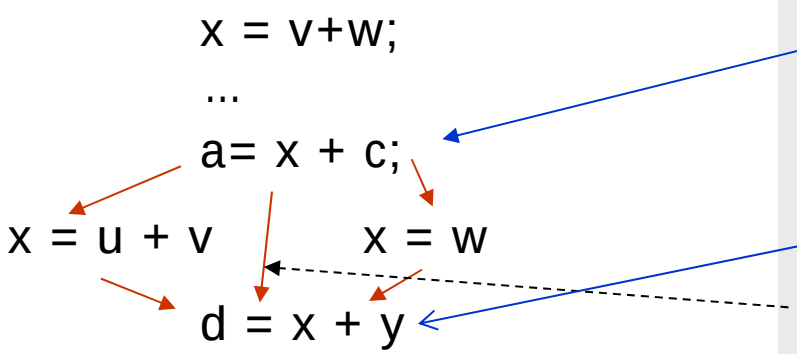
Her ”defineres” **a**, og her ”brukes” **x** og **y**

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.*

Teknisk definisjon av: Er ”x” levende etter instruksjonen ”i”:

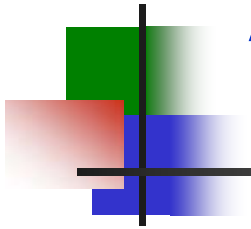


Stedet ”i” er denne TA-instruksjonen

Er ”x” i live etter denne instruksjonen ?

Svaret er ”ja”, fordi det finnes en TA-setning ”j” som *bruker* ”x”, og *det er minst én eksekveringsvei fra ”i” til ”j” uten noen tilordning til* ”x”.

Definisjon: En variabel som ikke er ”levende” på et gitt punkt, sies å være ”død” på dette punktet (og dens verdien behøver da ikke lagres)



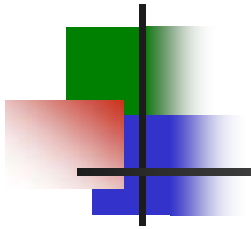
Andre ting vi kan være interessante med tanke på optimalisering

Global dataflyt-analyse. Eksempler:

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

Disse og liknende sammenhenger kan ”lett” bergenes ved en kompletterings-algoritme på de basale blokkene (omtrent som når vi finner First og Follow)

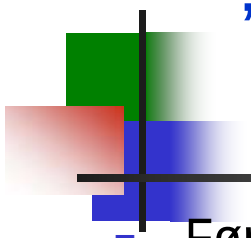
Vi skal, som et eksempel, se på en slik algoritme



Notatet, kap 9.5. Her ser man på:
Å generere ”lur” kode for én og én basal blokk

- Innen en basal blokk er det lett å holde orden på hvilke verdier som er i hvilke registre etc. ned gjennom blokken
- Filosofien for metoden vi ser på:
  - Mellom hver basal blokk sørger vi for:
    - Alle verdier av program-variable ligger i variablens lokasjon ”ute i hovedlageret” (også kalt ”hjemme-posisjonen”)
    - Vi antar også et TA-koden er laget slik at temporære variable ikke skal bære verdier fra én basal blokk til en annen. Temporære variable er altså døde ved begynnelsen og slutten av hver basal blokk
  - Det kan jo også hende at *programvariable* er døde ved slutten av en basal blokk.
    - Om vi skal få oversikt over dette må vi gjøre *global dataflytanalyse*
    - Men det gjør vi ikke foreløpig. Vi må derfor anta at alle *program-variable* er i live ved slutten av en basal blokk.

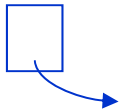
”Neste bruk” innen en basal blokk, og ”i live”



- Før man gjør kodegenerering for en basal blokk er det lurt å skaffe seg oversikt over bruk av variable (temporære og andre) i blokka. Vi ser på en gitt variabel-forkomst i en gitt TA-instruksjon i blokka:
  - En variabel-forekomst kan ha en ”neste-bruk” i blokka (derved ”i live”):
    - Def: Den verdien den her har blir brukt senere i samme basale blokka.
    - Da kan det f.eks. være lurt å la den bli værende i et register, om mulig
  - En variabel-forekomst som ikke har noen ”neste-bruk”, kan fremdeles være ”i live”:
    - Da: Verdien i variabelen blir ikke brukt senere i blokka
    - Men denne verdien kan bli brukt i andre blokker senere.
    - Dette gjelder i vår setting bare program-variable (ikke temporære)
  - Noen variabel-forekomster som helt sikkert er døde:
    - Alle temporære variable som ikke blir referert mer i blokka
    - Alle variable som blir gitt ny verdi lenger ned i blokka, og som ikke brukes før det.

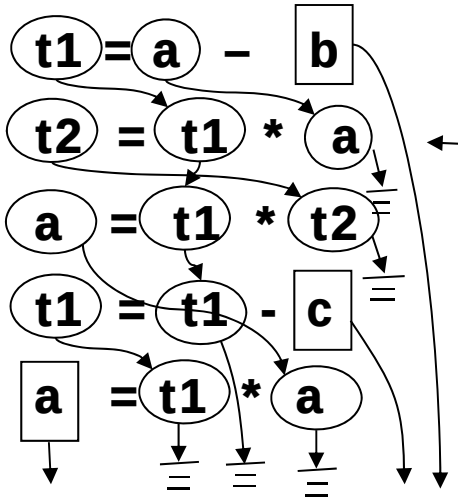
Eksempel på informasjon om ”neste bruk” og ”i live” innen en basal blokk

: Angir at den har ”neste bruk” i blokka (og hvor, men dette brukes ikke av vår algoritme). Slike er ”i live”



: Angir at den er ”i live”, men uten noen ”neste bruk” i blokka

: Angir at den er ”død” etter denne inst.



a er ”død” her siden den får ny verdi innen denne blokka og ikke blir brukt før det

t1 er død her, siden vi antar at temporære bare brukes innen samme blokka

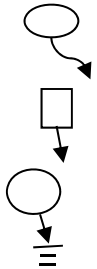
Vi antar altså: Programvariablene **a, b, c** er i live etter blokka.

Kommentarer:

1. Global ”dataflyt-analyse” finner de variable som faktisk er i live etter en basal blokk.
2. Temporære brukes også ofte til å bære verdier fra én basal blokk til en annen

Algoritme for å finne informasjon om variablers ”neste bruk” i blokka og om de er ”i live”

Vi har en tabell T over alle variable i blokka, der hver variabel kan merkes som:



1) ”i live”, og har en angitt ”neste bruk” (i blokka)
2) ”i live”, men uten ”neste bruk” (i blokka)
3) ”død” (og dermed ingen ”neste bruk”)

Initialisering av tabellen T:

De variablene som er ”i live” ved slutten av blokka (her: program-variablene) merkes med (2), resten (de temporære) merkes (3)

Steget (gjentas for hver TA-instr. ”x = y op z”, fra siste til første):

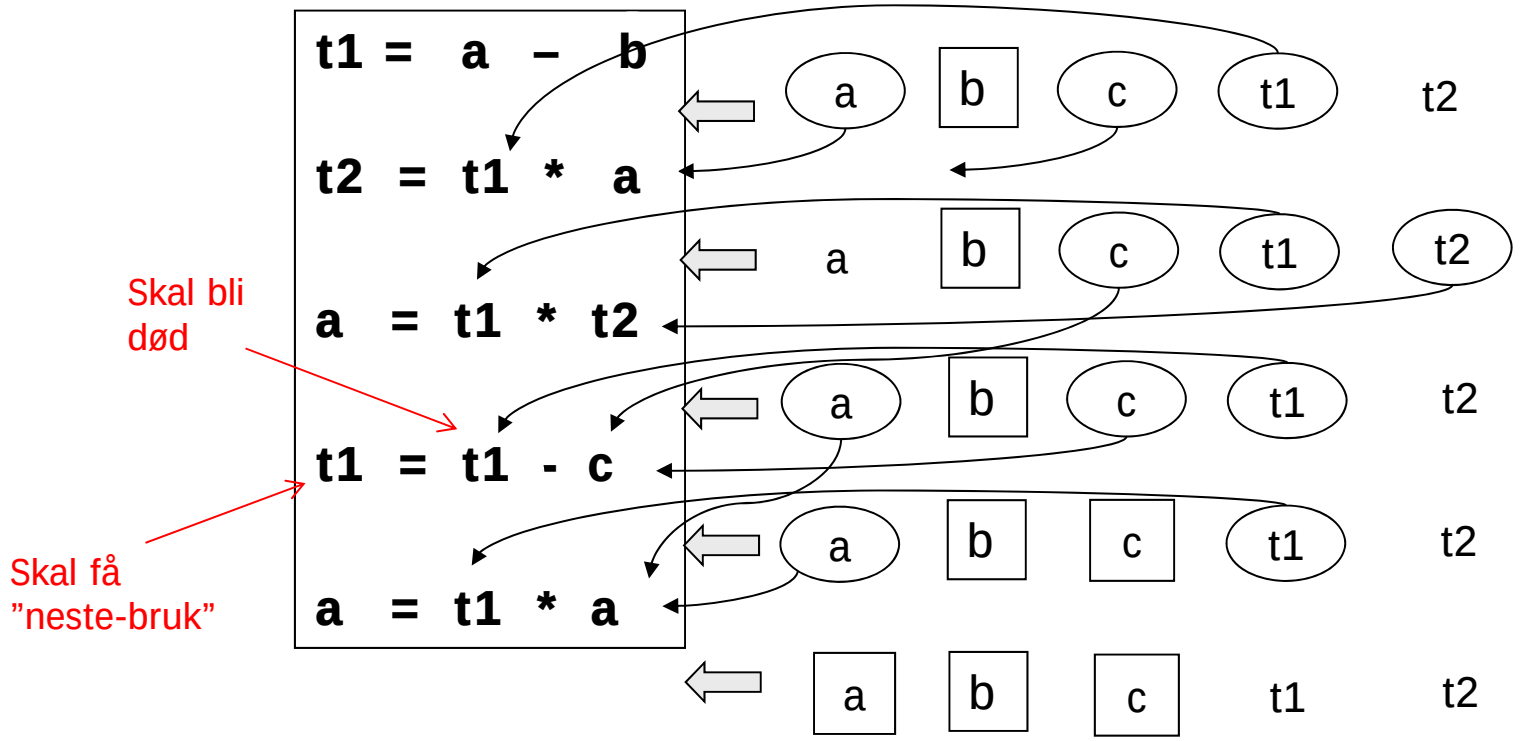
1. Merk x i TA-instr. slik x er merket i T
2. Forandre x sitt merke i T til 3 (altså død)
3. Merk y og z i TA-instr. slik de er merket i T
4. Forandre i T merkene for y og z til (1), med ”neste bruk” satt til h.h.v. y og z i TA-instruksjonen.

Må skille mellom 1. og 3. for at `a = a + b;` skal bli riktig behandlet.
(Trykkfeil som er rettet i det utdelte notat)

Eksempel:
Finn ”neste bruk” og ”i live” innen en basal blokk

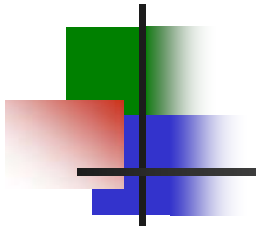
- Gå bakfra og hold greie på status til alle variable:
  (N.B. Trykkfeil i notatets algoritme – er rettet i den utlagte kopien)

Tabellen T på de forskjelliige stadier:



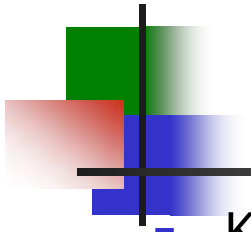
Døde variable er tegnet uten ring eller firkant rundt.

I nederste linje (initialiseringen) antar at vi at bare progr. variable overlever fra en blokk til en annen.



Kodegenererings-algoritme

- Altså, en enkel algoritme: Lager maskinkode for én og én Basal Blokk
- Algoritmen lager kode som, innefor hver Basal Blokk, holder beregnede verdier i registre så langt det er mulig og ønskelig (spesielt viktig om de har ”neste bruk” i denne basale blokken)
- Når programkontrollen går mellom de basale blokkene så skal samtlige variabel-verdier ligge i sine respektive hukommelses-plasser (mens temporære variable ikke ”er i live”, og dermed ikke behøver lagres)
- Kodegenerering for hver basal blokk blir da:
  - Utfør algoritmen for å finne ”neste bruk” og ”i live” (går altså baklengs)
  - Det genereres kode for én og én treadresse-setning av gangen, i tur og orden fra første til siste setning
  - OG husk: Etter siste setning genereres kode for å legge verdier fra registre tilbake til sine respektive hukommelses-plasser der det er nødvendig.
- Noen mangler ved algoritmen (men disse kan i stor grad lett kan rettes opp)
  - Variable som kun blir *lest* innenfor en Basal Blokk blir *aldri* lagt i registre, selv ikke om det er gjentatte referanser til variabelen i blokken
  - I enkleste utgave utnytter den ikke kommutativitet for + og *

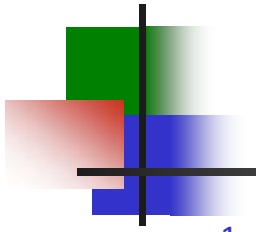


Register- og adresse-deskriptorer

- Kodegenerator-algoritmen bruker deskriptorer for å holde greie på hva som er i registre og i program-variablene:
  - En register-deskriptor for hvert register holder greie på hva som for tiden er i registerene. Ved starten skal alle register-deskriptorer angi at registeret er ledig. Generelt angir register-deskriptoren enten at registeret er ledig eller at det inneholder verdien til en eller flere angitte variable.
  - En adresse-deskriptor holder greie på hvor verdien av en variabel finnes i øyeblikket. Den kan være i ett eller flere registre, og/eller i variabelens lager-lokasjon (”hjemme-posisjon”)
    - Disse desriptorene opprettes etter hvert som det blir ”snakk om” variablene. At det ikke er noen adresse-deskriptor for ’x’ betyr:
      - x er programvariabel: Verdien ligger (bare) i variablens lager-lokasjon
      - x er temporær variabel: Variabelen er ikke i bruk nå
  - Informasjonen er redundant – dvs. vi har begge deskriptor-typene (adresse og register) ”bare” for å få raske oppslag. Kunne greid oss med én av dem.

Typisk bruk av deskriptorene

Setninger	Generert kode	Reg. deskriptorer	Adr. deskriptorer
		Alle Reg er ubrukte	
t = a - b	MOV a, R0 SUB b, R0	R0 inneholder t	t i R0
u = a – c	MOV a, R1 SUB c, R1	R0 inneholder t R1 inneholder u	t i R0 u i R1
v = t + u	ADD R1, R0	R0 inneholder v R1 inneholder u	v i R0 u i R1
d = v + u	ADD R1, R0	R0 inneholder d	d i R0 og ikke i hukommelsen
Avslutning av basal blokk	MOV R0, d	Alle Reg er ubrukte	(Alle prog.variable i home posistion)



Kodegenerering for: **X = Y op Z**

(Rettelser som er angitt i notatet er gjort her)

1. Finn et register for å holde resultatet:
   - L = getreg(”X = Y op Z”) // Helst et sted Y allerede er
2. Sørg for at verdien av Y faktisk er i L:
   - Hvis Y er i L, oppdater adressediskr. til Y: Y ikke lenger i L **else**
   - Y’ := ”beste lokasjon” der verdien av Y finnes
   - OG: generer: **MOV Y’ L**
3. Sjekk adresse-deskriptoren for Z:

   Z’ := ”beste” lokasjon der verdien til Z ligger // Helst et register
   - Generer så ”hovedinstruksjonen”: **OP Z’ L**
4. For hver av Y og Z: Om den er død og er i et register

   Oppdater i så fall register-deskriptoren:

   Registrene inneholder nå ikke lenger Y og/eller Z
5. Oppdaterer deskriptorer i forhold X:
   - X sin adr.deskr. := {L}, og X er ingen andre steder.
6. Hvis L er et register så oppdater register-deskr. for L:
   - L sin reg.deskr. := {X}

Getreg (”X = Y op Z”)

Instruksjonen som utfører operasjonen vil få Y som target-adresse

1. Hvis Y ikke er ”i live” etter ”X = Y op Z”, og Y er alene i R_i:
   - return(R_i) (punkt 1 kan lett forfines en god del) **else**
2. Hvis det finnes et tomt register R_i : return (R_i) **else**
3. Hvis X har en ”neste bruk” eller X er lik Z eller operatoren ellers krever et register:
   - Velg et (opptatt) register R
   - Hvis verdien i R ikke også ligger ”hjemme” i hukommelsen:
     - Generer **MOV R mem** // mem er hjemmeposisjon for R-verdien
     - Oppdater adresse-deskriptor for **mem**
   - return (R) **else**
4. return (X), altså lever hukommelses-plassen til X (må kanskje opprettes om X er en temp-variabel)

Opprinnelig verdi av X ødelegges

NB: For at X = Y + X skal funke, måtte pnk. 3 modifiseres, ellers ville vi fått:

```
MOV  Y  X
ADD  X  X
```

Eksempel på kode-generering

(samme som en tidligere foil)

Setninger	Generert kode	Reg. deskriptorer	Adr. deskriptorer
		Alle Reg er ubrukte	
t = a - b	MOV a, R0 SUB b, R0	R0 inneholder t	t i R0
u = a – c	MOV a, R1 SUB c, R1	R0 inneholder t R1 inneholder u	t i R0 u i R1
v = t + u	ADD R1, R0	R0 inneholder v R1 inneholder u	v i R0 u i R1
d = v + u	ADD R1, R0	R0 inneholder d	d i R0 og ikke i hjemmeposisjon
Avslutning av basal blokk	MOV R0, d	Alle Reg er ubrukte	(Alle prog.variable i hjemmepos.)