Hvordan programmere assemblerspråk

Å skrive programmer på assembler-nivå kan virke tungvint og vanskelig for mange som er vant til høynivåspråk som C og Java.
Ofte kan det være lurt å først skrive ned algoritmen man skal implementere i (java liknende) pseudokode for deretter å «kompilere» denne ned til assemblerspråk for hånd.
Dette rommet tar for seg en del vanlige kontrollstrukturer i høynivåspråk og viser hvordan disse kan oversettes til assemblerspråk for load/store-arkitekturen til Dark.

if, else if, else

Dette er kanskje den viktigste kontrollstrukturen. Man vil garantert få bruk for noe som ligner på dette når man programmerer.
Først viser vi et eksempel på bruk av if-else i pseudokode:

```
if (a < b) {
  d = a;
} else if (b > c) {
  d = b;
} else {
  d = c;
}
```

Dette kan oversettes til assemblerspråk slik:

```
; $1 : a
; $2 : b
; $3 : c
; $4 : d

        jge $1, $2, ifa         ; if( a < b ) {
        add $4, $1, zero        ;       d = a;
        jmp ifend               ; }
                                ;
ifa:    jle $2, $3, ifb         ; else if( b > c ) {
        add $4, $2, zero        ;       d = b;
        jmp ifend               ; }

ifb:                            ; else {
        add $4, $3, zero        ;       d = c;
                                ; }
ifend:
```

Vi ser at en if-setning består av et betinget hopp som hopper til neste else dersom if-betingelsen *ikke* er oppfyllt (betingelsen til hoppinstruksjonen er derfor den motsatte av if-betingelsen i pseudokoden). Dersom betingelsen *er* oppfyllt utføres det som står inne i if-blokken (d = a) og det gjøres et hopp til slutten av if-setningen (ifend).
Else-if implementeres på akkurat samme måte.
Else-setningen har ingen betingelse og vil derfor alltid utføre dersom vi kommer til dette punktet i programmet.

while

While er en vanlig måte å lage løkker på. Vi har en betingelse som sjekkes for hver iterasjon i løkken. Dersom den *ikke* er oppfyllt hopper vi ut av løkken.

```
while (a < b) {
      a++;
}
```

Dette kan implementeres på følgende måte:

```
; $1 : a
; $2 : b

wstart: jge $1, $2, wend        ; while( a < b ) {
        inc $1                  ;       a++;
        jmp wstart              ; }
wend:
```

Vi har et betinget hopp som hopper ut av løkken dersom betingelsen ikke er oppfyllt. Hvis ikke utføres løkken og vi har på bunnen en hoppinstruksjon som hopper tilbake til toppen av løkken.

do while

Do-while er nesten lik while bortsett fra at betingelsen sjekkes på slutten av hver iterasjon i stedet for på begynnelsen. Denne benyttes mye sjeldnere enn while i språk som C og java.

```
do {
      a++;
} while (a < b);
```

Dette kan implementeres slik:

```
; $1 : a
; $2 : b

loop:                           ; do {
        inc $1                  ;    a++;
                                ;
        jle $1, $2, loop        ; } while (a < b);
```

Vi har her ikke noe betinget hopp på toppen av løkken, vi har i stedet satt betingelsen på hoppinstruksjonen på bunnen. Dette er egentlig mer intuitivt enn vanlig while i assemblerspråk. Vi får en hoppinstruksjon mindre noe som både er mer effektivt og oversiktlig.

for

For-løkker er veldig vanlig og nyttig. For-løkker i C og java er veldig generelle, men vi har vanligvis en teller som begynner på en startverdi og teller opp til vi når en sluttverdi. Og for hver gang vi teller opp telleren utføres løkka en gang.

```
for (i = 0; i < a; i++) {
        b++;
}
```

Dette kan implementeres slik:

```
; $1 : i
; $2 : a
; $3 : b

        mov $1, 0               ; for(i = 0; i < a; i++) {
floop:  jge $1, $2, fend        ;
        inc $3                  ;       b++;
        inc $1                  ;
        jmp floop               ; }
fend:
```

Vi må først sette tellerregisteret vårt til 0. Deretter starter løkken. På toppen av løkken finnes et betinget hopp som hopper ut av løkken hvis betingelsen ikke er oppfyllt. Innholdet i løkken utføres, tellerregisteret inkrementeres og vi hopper tilbake til toppen av løkken igjen.

prosedyrer

Prosedyrekall er veldig nyttig når vi ønsker å utføre den samme kodebiten flere steder i programkoden. I C kalles prosedyrene for *funksjoner*, i java kalles de ofte for *metoder*.
Det finnes svært mange måter å implementere prosedyrer på i assemblerspråk. Hovedsaklig består de av en kodebit som ender med instruksjonen ret. Prosedyren kalles med instruksjonen call. Problemet består i hvordan man overbringer argumenter og returverdier.
Her er et eksempel:

```
int main() {

  int a = 10;
  int b = 2;
  int c = 5;

  c = addisjon(a,b) + subtraksjon(b,c);

}

int addisjon(int a, b) {
  return a+b;
}

int subtraksjon(int a, b) {
  return a-b;
}
```

Dette kan vi på en enkel måte oversette slik:

```
; $1 : a
; $2 : b
; $3 : c
; $4 : arg a til add
; $5 : arg b til add
; $6 : returverdi fra add
; $7 : arg a til sub
; $8 : arg b til sub
; $9 : returverdi fra sub
```

```
        load $1, 10
        load $2, 2
        load $3, 5

        add $4, $1, 0
        add $5, $2, 0
        call add

        add $7, $2, 0
        add $8, $3, 0
        call sub

        add $3, $6, $9

;--------------------


add:    add $6, $4, $5
        ret

sub:    sub $9, $7, $8
        ret
```

Vi ser at hver prosedyre har fått allokert et sett med registre som benyttes til argumenter og returverdi. Dette er enkelt og raskt, men har mange problemer. Hver gang vi kalles en prosedyre vil noen av registrene få slettet sin verdi. Dessuten vil vi få problemer med nøstede prosedyrekall og rekursjon.
Løsningen er ofte å bruke en stakk. Før prosedyren kalles kan man lagre alle registre på stakken. Da kan man etter at prosedyren er ferdig gjenopprette disse. Argumenter og returverdier lagres også på stakken. Det samme gjør vi med eventuelle lokale variable som ikke får plass i registre. Dette gjør at vi ikke får noen problemer med rekursjon og nøstede kall.