Programmering i maskinspråk (Maskinassemblering)

Programutveckling i assemblerspråk

Begreppet assemblerspråk introduceras i arbetsboken (ARB) kapitlen 14-16. En del korta programavsnitt skrivs med assemblerspråk i övningsuppgifterna för FLISP-processorn i samma bok. Assemblerspråket gör det möjligt för oss att skriva program där vi har fullständig kontroll över den **maskinkod** som hamnar i minnet. Maskinkoden utgör det verkliga programmet som körs av processorn. Varje processor har sitt eget assemblerspråk och man säger att det är ett **maskinorienterat språk**.

Man kan fråga sig om det finns någon anledning att programmera i ett maskinorienterat språk när det finns så många effektiva högnivåspråk som idag?

Svaret på frågan är ja i vissa sammanhang. Situationer när det finns anledning att skriva program i processorns assemblerspråk är när

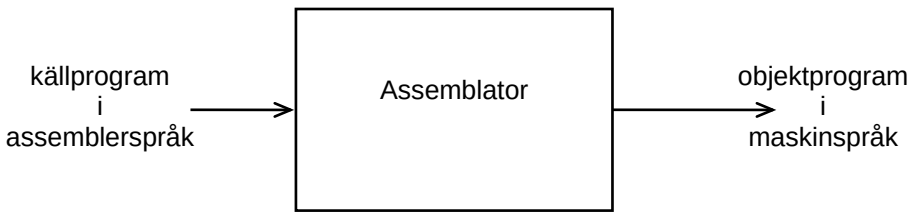
- det inte finns någon kompilator, som genererar kod, för den aktuella processorn.
- det finns uppgifter som inte går att beskriva i ett högnivåspråk.
- någon uppgift är så tidskritisk att en kompilator inte klarar av att göra koden så att den klarar tidskraven.
- man vill lära sig "hands-on" hur den aktuella processorn och dess instruktioner fungerar. Detta kan även gälla processorns interaktion yttre enheter.

Det finns således goda skäl för att lära sig programmera i assemblerspråk.

Övning i att programmera i assemblerspråk ger också förståelse för hur program i högnivåspråk verkligen realiseras som maskinprogram, d v s i maskinspråket. Man blir därmed medveten om begränsningar som finns och om fel som kan uppstå.

Kunskaper om programmeringsspråk och programmering är viktiga för kompilator-konstruktörer. Behovet av att konstruera kompilatorer ökar i samma takt som nya processorer eller processorvarianter konstrueras. En kompilator är ju egentligen ett program som fungerar som assemblerprogrammerare.

Precis som det finns **kompilatorer** för att översätta program från **högnivåspråk** till **maskinspråk** (ofta med assemblerspråket som mellansteg) så finns det **assemblatorer** (eng **assemblers**) som översätter program från assemblerspråk till maskinspråk, se figur 1.



Figur 1

För en given processor har såväl assemblerspråk som assemblator vanligen konstruerats tillsammans. Oftast är de konstruerade av processortillverkaren själv. Ett program i assemblerspråk är således inte bara avsett för en processor utan också för en assemblator.

Assemblerspråket utgörs huvudsakligen av satser, som endera svarar mot en maskininstruktion eller mot en instruktionsliknande anvisning (upplysning) till assemblatorn, ett s k **assemblatordirektiv** (eng **assembler directive**), ibland också kallad pseudoinstruktion.

Innan det är dags att gå in mer i detalj på assemblatorn och dess funktionssätt är det lämpligt att först behandla programskrivningen (kodningen) något.

Programskrivningen underlättas av att assemblatorn ger möjlighet för programmeraren att använda symboliska beteckningar inte enbart för maskininstruktionerna utan även för variabler, konstanter och adresser.

Assemblatorn har två viktiga uppgifter.

1) Den skall översätta de mnemoniska (= till stöd för minnet) instruktionsbeteckningarna till maskinkod. Detta för att programmeraren inte direkt skall behöva arbeta med instruktionernas OP-koder.

Exempel 1

I FLISP assemblerspråk kan man som bekant skriva

```
LDA    $20          (Vi antar att programavsnittet börjar på adress 40₁₆)
ADDA   $21
ANDA   #%00001111
STA    $30
JMP    $40          Hopp till början av programavsnittet
```

istället för

```
F1  20              (F1  20 svarar mot    LDA    $20)
A6  21              (A6  21 svarar mot    ADDA   $21)
99  0F              (99  0F svarar mot    ANDA   #$0F)
E1  30              (E1  30 svarar mot    STA    $30)
33  40              (33  40 svarar mot    JMP    $40)
```

2) Assemblatorn skall ge programmeraren möjlighet att arbeta med **symboliska beteckningar** (namn) för adresser och datavärden. Detta för att avlasta programmeraren från att (vid programmeringen) hålla reda på det exakta siffervärdet på en adress där en variabel finns lagrad eller dit ett hopp skall göras o s v. Assemblatorn skall därför översätta även dessa symboliska beteckningar till sina rätta värden.

Exempel 2

Assemblatorn ger oss möjlighet att skriva programavsnittet i exempel 1 som:

```
LOC1  LDA    ALFA1
      ADDA   ALFA2
      ANDA   # CONST
      STA    SUM
      JMP    LOC1
```

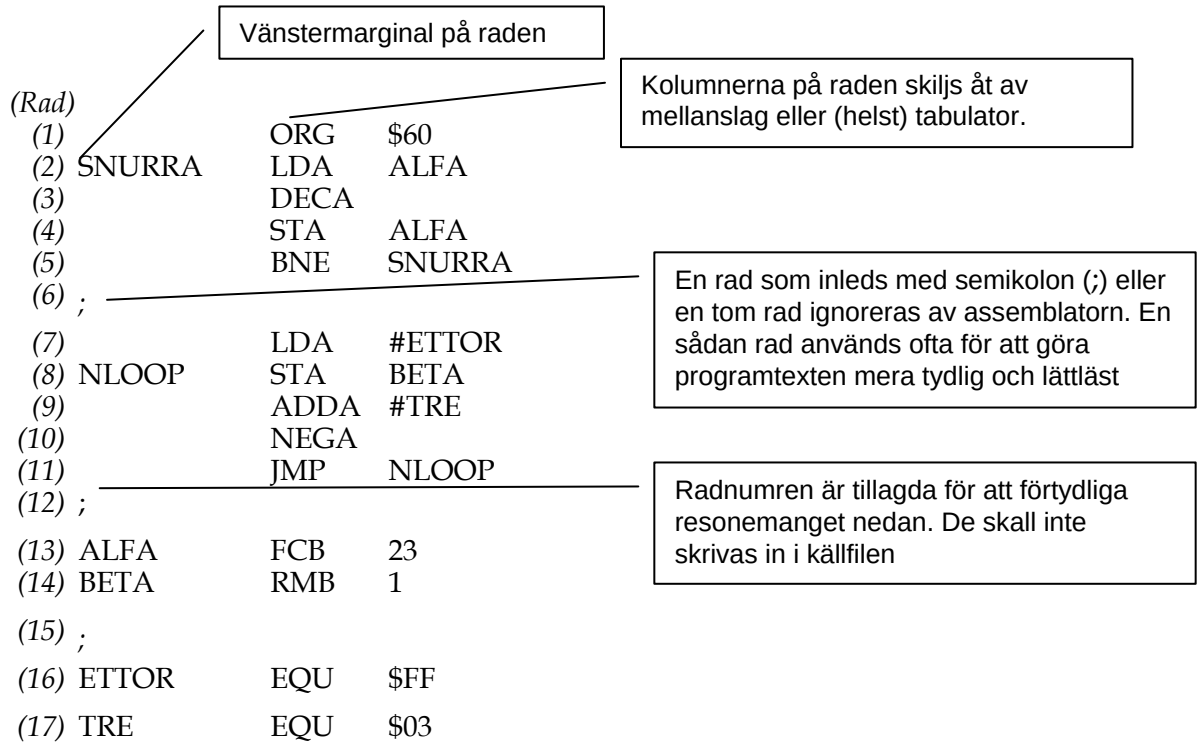
där vi har ersatt adressen 40_{16} med den symboliska beteckningen LOC1 och adresserna 20_{16}, 21_{16} och 30_{16} med variabelnamnen ALFA1, ALFA2 och SUM, samt konstanten $0F_{16}$ med namnet CONST.

Tvåpass-assemblatorns funktionssätt

För att förstå vad assemblatorn kan göra och vad den inte kan göra är det viktigt att känna till principen för hur den fungerar. Här visas med hjälp av ett exempel hur en sk tvåpass-assemblator går tillväga vid översättning av ett program till maskinkod.

Exempel 3

Programmet nedan skall användas för visa hur en typisk assemblator arbetar.




```
(Rad)
 (1)              ORG    $60
 (2)  SNURRA      LDA    ALFA
 (3)              DECA
 (4)              STA    ALFA
 (5)              BNE    SNURRA
 (6)  ;
 (7)              LDA    #ETTOR
 (8)  NLOOP       STA    BETA
 (9)              ADDA   #TRE
(10)              NEGA
(11)              JMP    NLOOP
(12)  ;
(13)  ALFA        FCB    23
(14)  BETA        RMB    1
(15)  ;
(16)  ETTOR       EQU    $FF
(17)  TRE         EQU    $03
```

Assembatorn går igenom assemblerprogrammet från början till slut två gånger. Assembleringen sägs ske i två **pass** och assemblatorn kallas därför **tvåpassassemblator**. Arbetssättet beror bl a på att assemblatorn först måste bestämma värdet hos samtliga symboler (**pass 1**) innan den genererar maskinkoden (**pass 2**).

Pass 1

Assemblatorns viktigaste uppgift under första passet är att definiera värdena hos de symboler som ingår i programmet ("SNURRA", "NLOOP", "ALFA", "BETA", "ETTOR " och "TRE" i exempel 3). De vanligaste symbolerna är **lägesnamn** , d v s symbolens värde skall vara adressen till den instruktion som (normalt) står på samma rad. Symbolerna "SNURRA", "NLOOP ", "ALFA" och "BETA" i exempel 3 är lägesnamn.

För att kunna bestämma värdena för lägesnamnen måste assemblatorn veta på vilken adress första instruktionen skall ligga samt längden av varje instruktion. Internt i assemblatorn finns därför en variabel **LC** (eng **Location Counter**)**,** som är en adressräknare. Den anger hela tiden vilken adress den instruktion, som assemblatorn just då behandlar, skall ligga på.

Den första raden i exempel 3

```
        ORG     $60
```

är ingen maskininstruktion utan något som kallas för **assemblatordirektiv** eller kortare bara **direktiv** (eng directive). Ett direktiv är en anvisning till assemblatorn att göra något, i detta fall att sätta LC till 60_{16} ($-tecknet står som vanligt för ett hexadecimalt tal). Innebörden blir att programavsnittet kommer att börja på adressen 60_{16}. I appendix E i arbetsboken (ARB) redovisas samtliga direktiv som kan användas till FLISP-datorns assemblator.

På rad 2 ser assemblatorn ordet "SNURRA" i första kolumnen och efter det en assembler-instruktion i de två följande kolumnerna. Det innebär att assemblatorn skall definiera en symbol med namnet "SNURRA" och värdet hos LC dvs 60_{16}. Det är den adress som maskinkoden för instruktionen kommer att börja på. Denna information införs i en tabell, som kallas **(användarens) symboltabell**.

Assemblatorn innehåller en instruktionslista med uppgifter om OP-koden och längden (antal bytes) för varje instruktion. Med hjälp av listan konstaterar assemblatorn att maskin-instruktionen ”LDA Adr” har längden 2 bytes och ökar därför LC med 2. LC innehåller därefter adressen 62_{16} till nästa instruktions OP-kod.

På rad 3 i programmet finns assemblerinstruktionen, DECA, vars maskinkod är 1 byte lång, vilket gör att LC ökas med 1 till 63_{16}.

Proceduren upprepas på samma sätt tills assemblatorn stöter på rad 8 som börjar med ordet "NLOOP". Assemblatorn för då in symbolen "NLOOP" i symboltabellen och ger den LC’s värde.

Proceduren fortsätter tills assemblatorn stöter på rad 13 som börjar med ordet "ALFA" och därefter har ett nytt direktiv till assemblatorn; "FCB 23" (Form Constant Byte 23).

Assemblatorn för då in symbolen "ALFA" med LC's värde i symboltabellen. Direktivet "FCB 23" talar om för assemblatorn att en byte (med innehållet 23) skall hamna i minnet på den adress som finns i LC. Eftersom en byte upptar ett minnesord ökar assemblatorn LC med 1. Senare (i pass 2) skall talet 23 placeras på adressen "ALFA".

Rad 14 börjar med ordet "BETA" och följs av assemblatordirektivet "RMB 1" " (Reserve Memory Byte 1), som skall reservera 1 byte i minnet för en variabel.

Assemblatorn för därför in symbolen "BETA" med LC's värde i symboltabellen och ökar sedan LC med 1 så att en ”lucka” på 1 byte skapas i minnet.

Raderna 16 och 17 börjar med orden "ETTOR" och "TRE" följda av direktivet EQU (Equate). Detta medför att symbolen till vänster tilldelas värdet till höger om "EQU". "EQU"-direktivet påverkar alltså inte LC.

Resultatet av pass 1 blir symboltabellen till höger.

Namn	Värde
SNURRA	$60
NLOOP	$69
ALFA	$70
BETA	$71
ETTOR	$FF
TRE	$03

Exempel 3 forts.

Under pass 1 skapar alltså assemblatorn symboltabellen till höger m h a programavsnittet.

Symboltabellen behövs i pass 2, då assemblatorn skall generera maskinkoden för de olika instruktionerna i programmet.

Pass 2

Assemblatorn börjar om från rad 1 och skall nu generera programkoden i form av maskinkod i en sk laddfil. Samtidigt genereras också en listfil som innehåller både programtexten (källkoden) och maskinkoden.

Pass 2 börjar på samma sätt som pass 1 med att LC sätts till programmets startadress (60_{16} i exempel 3) och fortsätter sedan radvis.

På rad 2 ser assemblatorn återigen maskininstruktionen "LDA ALFA" och konstaterar då (via en intern tabell) att OP-koden för denna instruktion är $F1_{16}$. OP-koden och adressen 60_{16}, som ges av LC, skrivs in i respektive filer. På raden följer sedan adressen med namnet "ALFA" efter "LDA". Värdet på "ALFA" (70_{16}) hämtas ur symboltabellen och skrivs in i ladd- och listfilen på adressen efter OP-koden. Därefter ökas LC med 2 (instruktionslängden för ”LDA Adress”) till 62_{16} och assemblatorn går till nästa rad.

På rad 3 behöver inte symboltabellen användas utan assemblatorn fyller i OP-koden (08_{16}) för ”DECA” i resp fil på adressen i LC (62_{16}). LC ökas med 1 (instruktionslängden för ”DECA”) till 63_{16}, och proceduren upprepas.

När assemblatorn stöter på instruktionen "BNE SNURRA" på rad 5 vet den att operanddelen skall innehålla hoppavståndet (positivt eller negativt) från instruktionen efter "BNE SNURRA" till den instruktion dit hoppet skall ske.

Adressen till instruktionen efter "BNE SNURRA" ges av LC (65_{16}) plus 2 ("BNE"-instruktionens längd), dvs $65_{16} + 2 = 67_{16}$.

Adressen dit man skall hoppa ges av värdet hos symbolen "SNURRA" (60_{16}).

Hoppavståndet blir $60_{16} – 67_{16} = -7_{16}$, eller $F9_{16}$ med 2-komplementrepresentation.

Assemblatorn fortsätter sedan och kommer så småningom till rad 13 med direktivet "FCB 23". Den fyller då i koden för talet 23 (17_{16}) i ladd- och listfilen på den adress som anges av LC (70_{16}) och ökar LC med 1.

På rad 14 med direktivet "RMB 1" ökar assemblatorn LC med 1 utan att fylla i något ladd- eller listfilen, vilket innebär att en lucka på 1 byte skapas i minnet.

Resultatet av pass 2 blir alltså två filer. Den ena med det assemblerade programmets maskinkod och motsvarande minnesadresser (dvs laddfilen). Den andra med en kombination av programtexten och maskinkoden (dvs listfilen).

Exempel 3 forts.

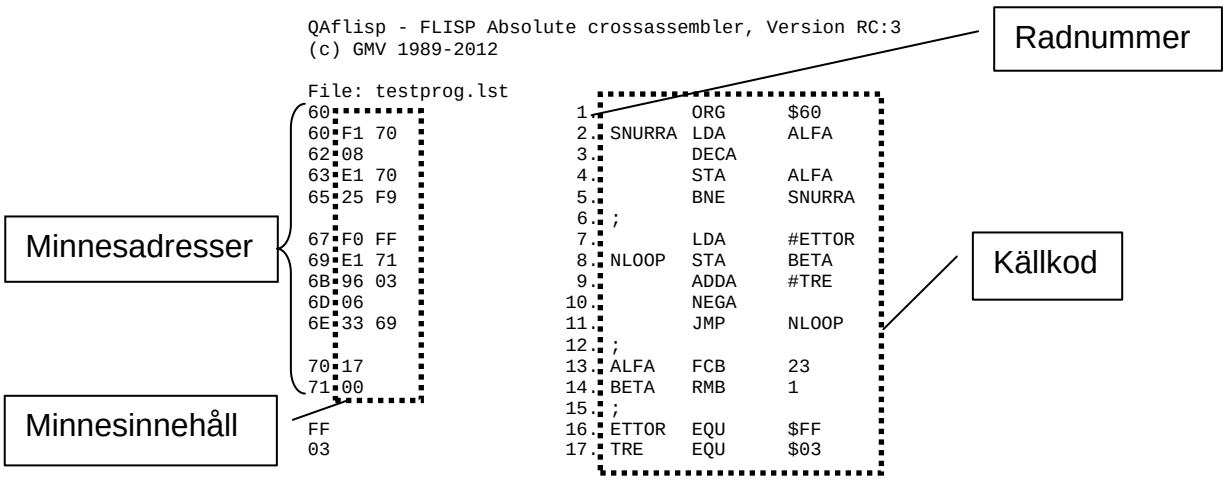
Under pass 2 går assemblatorn igenom programavsnittet igen, skapar programkod och placerar in den i en laddfil och en listfil enligt principen nedan.

Adress hex	**Värde** hex			
60	F1	SNURRA	LDA	ALFA
61	70			
62	08		DECA	
63	E1		STA	ALFA
64	70			
65	25		BNE	SNURRA
66	F9			
67	F0		LDA	#ETTOR
68	FF			
69	E1	NLOOP	STA	BETA
6A	71			
6B	96		ADDA	#TRE
6C	03			
6D	06		NEGA	
6E	33		JMP	NLOOP
6F	69			
70	17	ALFA	FCB	23
71	-	BETA	RMB	1

Vid assembleringen genereras två olika filer, en *objektfil* med filnamnstillägget *.s19* och en *listfil* med filnamnstillägget *.lst*. Objektfilen innehåller maskinkoden och adressinformation. Listfilen är en kombination av innehållen i källfilen och objektfilen. De nya filerna placeras i det bibliotek där källfilen finns. Man kan studera filerna genom att öppna dem med textredigeringsverktyget.

- Gå in under *File|Open* och välj *List Files (.lst)* i rutan *Files of type:* längst ner. Markera filnamnet *testprog.lst* och klicka på *Open*.

Textredigeringsfönstret öppnas med följande innehåll:




```
QAflisp - FLISP Absolute crossassembler, Version RC:3
(c) GMV 1989-2012

File: testprog.lst
60                      1           ORG     $60
60 F1 70                2   SNURRA  LDA     ALFA
62 08                   3           DECA
63 E1 70                4           STA     ALFA
65 25 F9                5           BNE     SNURRA
                        6   ;
67 F0 FF                7           LDA     #ETTOR
69 E1 71                8   NLOOP   STA     BETA
6B 96 03                9           ADDA    #TRE
6D 06                  10           NEGA
6E 33 69               11           JMP     NLOOP
                       12   ;
70 17                  13   ALFA    FCB     23
71 00                  14   BETA    RMB     1
                       15   ;
FF                     16   ETTOR   EQU     $FF
03                     17   TRE     EQU     $03
```

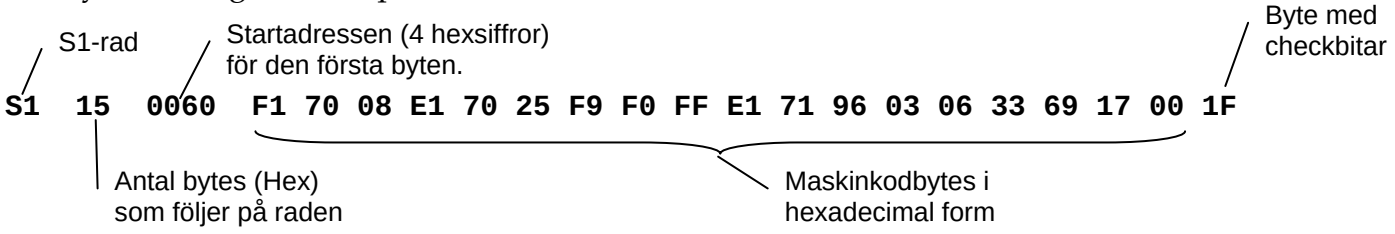
Man känner igen källkoden till höger i fönstret. Närmast till vänster om källkoden har varje rad försetts med radnummer 1-17. Längst till vänster finns minnesadressen och maskinkoden för varje instruktion. Man noterar också att raderna med EQU-definitioner inleds med den definierade symbolens värde längst till vänster. Listfilen är mycket användbar när man skall felsöka program.

- Öppna objektfilen under fliken *File|Open*. Välj där *Any File(*.*)* i rutan *Files of type:* och klicka på filnamnet *testprog.s19.* Ett fönster med följande innehåll öppnas då:

```
S1150060F17008E17025F9F0FFE171960306336917001F
S90300609C
```

Fönstret visar innehållet i objektfilen, som består av minnesinnehåll, adressinformation och paritetsbitar, allt i form av ASCII-tecken. Objektfilen används för laddning av programkoden till minnet, antingen i en simulator eller i den verkliga datorn.

Maskinkoden finns på rader som inleds med S1. I figuren nedan visas hur S1-raden i fönstret ovan är uppbyggd. Alla datavärden och adresser är givna på hexadecimal form, dvs som ASCII-tecknen för de hexadecimala siffrorna. Checkbitarna räknas ut så att den aritmetiska summan av alla bytes till höger om S1 på en rad blir FF_{16} modulo 2^8.



Slutraden i en objektfil inleds med S9, som talar om för laddverktyget att maskinkoden är slut.