Assemblerprogrammering för HCS12

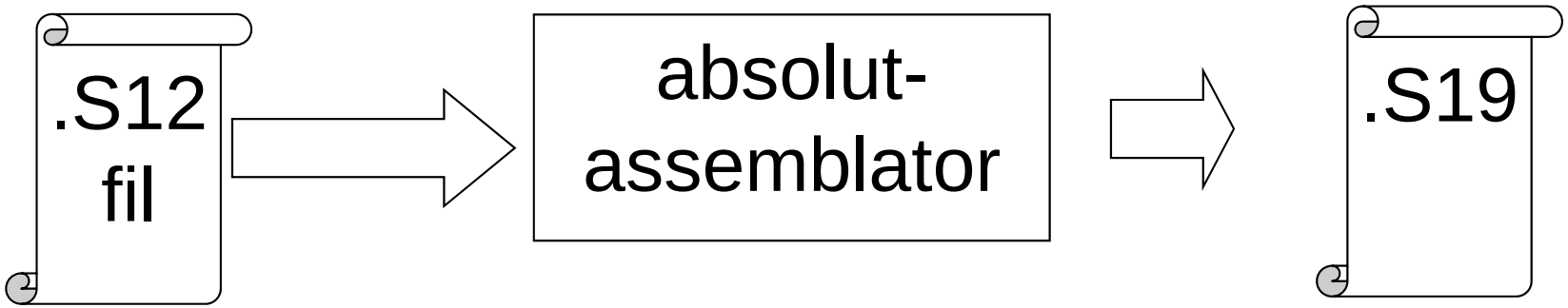
Ur innehållet:

- Assemblatorn, assemblerspråk
- Ordlängder och datatyper
- Tilldelningar, binära operationer
- Registerspill, permanenta och tillfälliga variabler
- Programkonstruktioner i assemblerspråk
- Subrutiners parametrar och returvärden
- Kodningsexempel och exekveringstidsanalys

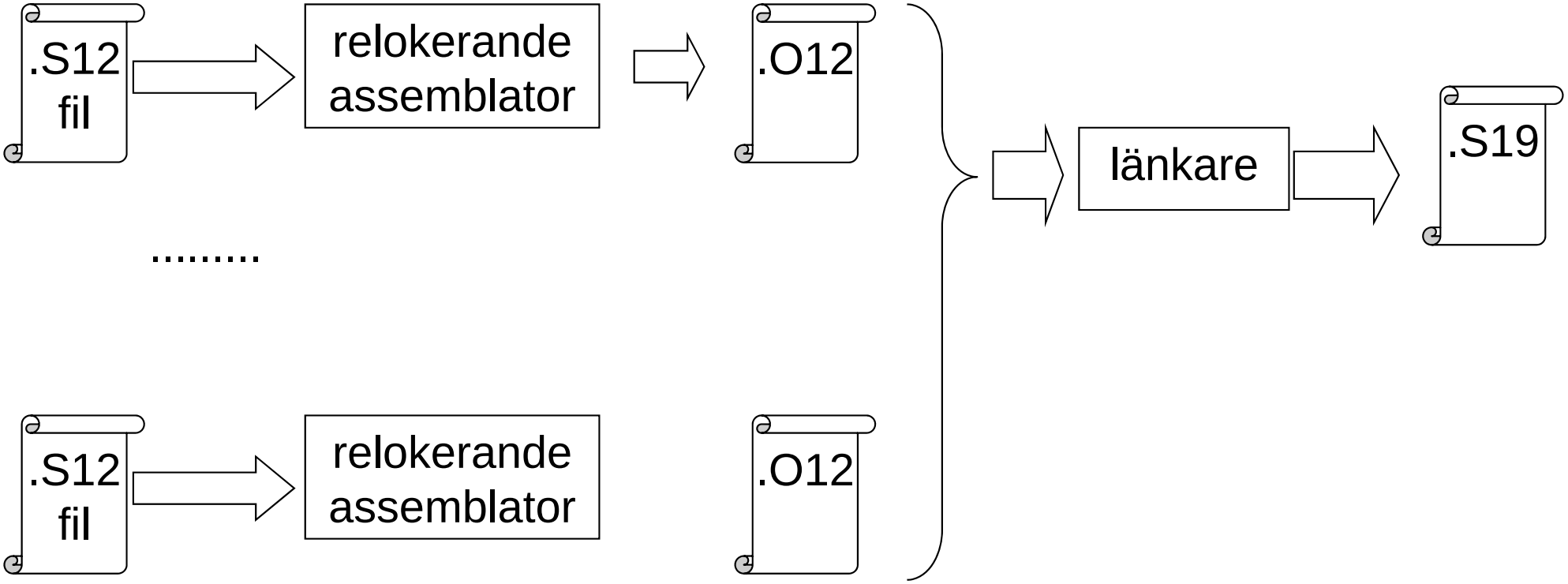
Absolut assemblering

All källtext assembleras samtidigt och alla referenser löses upp omedelbart.

Resultatet är en ”bild” av program/minne färdig att överföras till måldatorn.

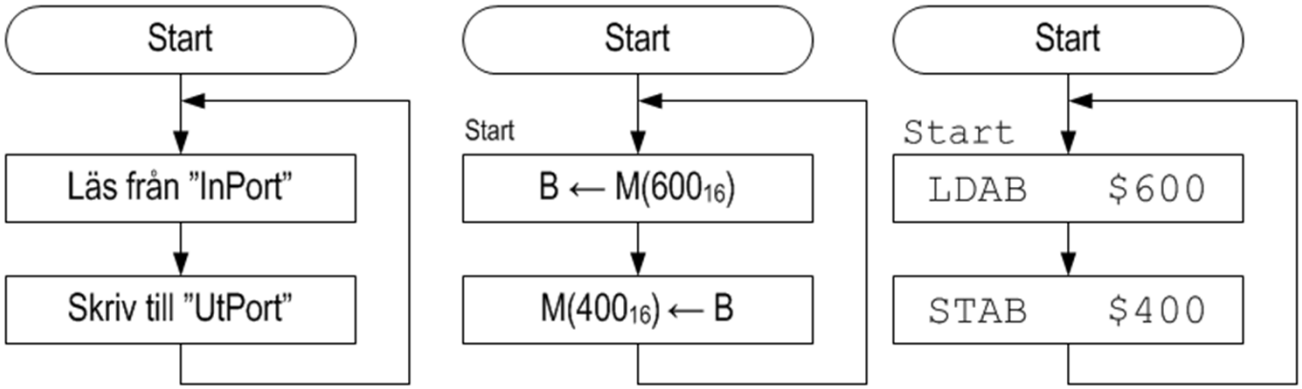


Relokerande assemblator



Källtexter assembleras till ett ”objektfilsformat” med symbolisk representation av adresser.
Vid ”länkningen” ersätts den symboliska informationen med absoluta adresser

Assembler-programmets struktur; exempel



Specifikation *Registerallokering* *Implementering*

```
; Programmet läser från en inport och kopierar till en utport
InPort      EQU         $600
OutPort     EQU         $400
            ORG         $1000
Start:
            LDAB        InPort          ; Läs från inporten…
            STAB        OutPort         ; Skriv till utporten
            BRA         Start           ; Börja om...
```

Symbolfält, blankt eller kommentar	Instruktion (mnemonic) eller assembler-direktiv	Operand(er) till instruktion eller argument till direktiv	Eventuell kommentarstext

Fälten separeras med blanktecken, dvs ”tabulatur” eller ”mellanslag”.

Assemblerspråkets element

ALLA textsträngar är ”context”-beroende

”**Mnemonic**”, ett ord som om det förekommer i instruktionsfältet tolkas som en assemblerinstruktion ur processorns instruktionsuppsättning. Mot varje sådan mnemonic svarar som regel EN maskininstruktion.

”**Assemblerdirektiv**” (”Pseudoinstruktion”), ett direktiv till assemblatorn.

Symboler, textsträng som börjar med bokstav eller _. Ska bara förekomma i symbol- eller operand- fälten

Direktiv och mnemonics är inte ”reserverade” ord i vanlig bemärkelse utan kan till exempel också användas som symbolnamn

Ett (dåligt) exempel...

```
        ORG     $1000
BRA     LDAA    ADDA
        ADDB    LDAA
        BRA     BRA
RMB     EQU     1
EQU     EQU     2
ADDA    EQU     EQU
LDAA    RMB     RMB
```

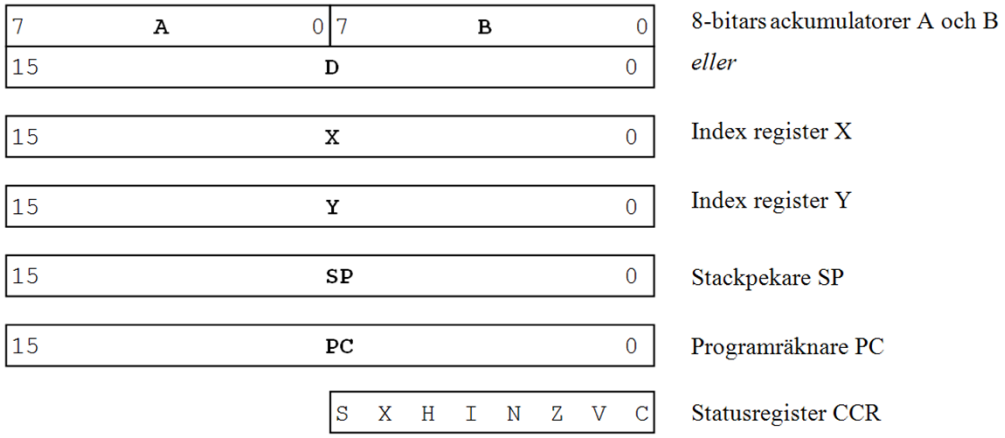
Syntaktiskt korrekt men extremt svårläst på grund utav illa valda symbolnamn...

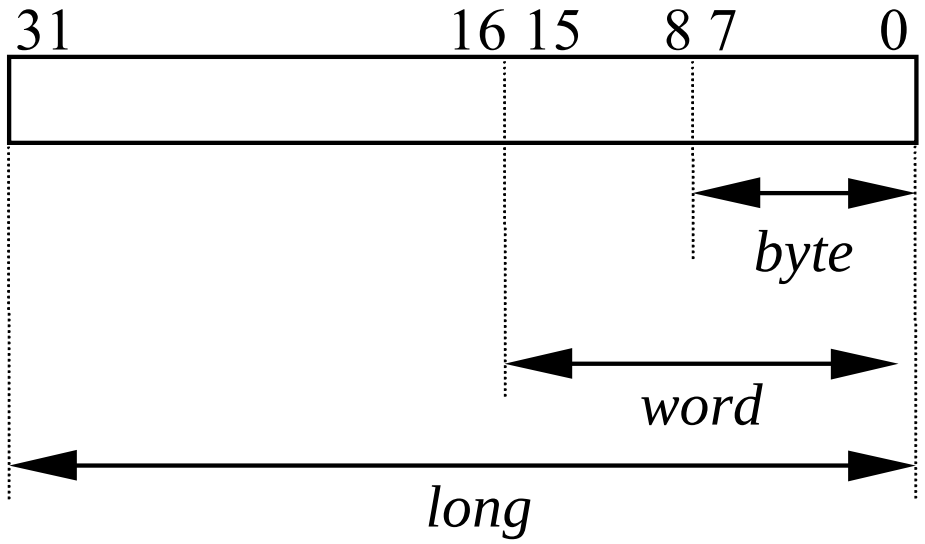
Ett bra exempel...

```
              ORG    $1000
main:         JSR    init
main_loop:    JSR    read
              JSR    ...
              ---
              BRA    main_loop

init:         ---
init_0:       RTS

read:         ---
read_loop:
read_exit:    RTS
```

Symbolnamnen väljs så att sammanblandning undviks. Undvik också generella symbolnamn som exempelvis LOOP

CPU12, ordlängder och datatyper

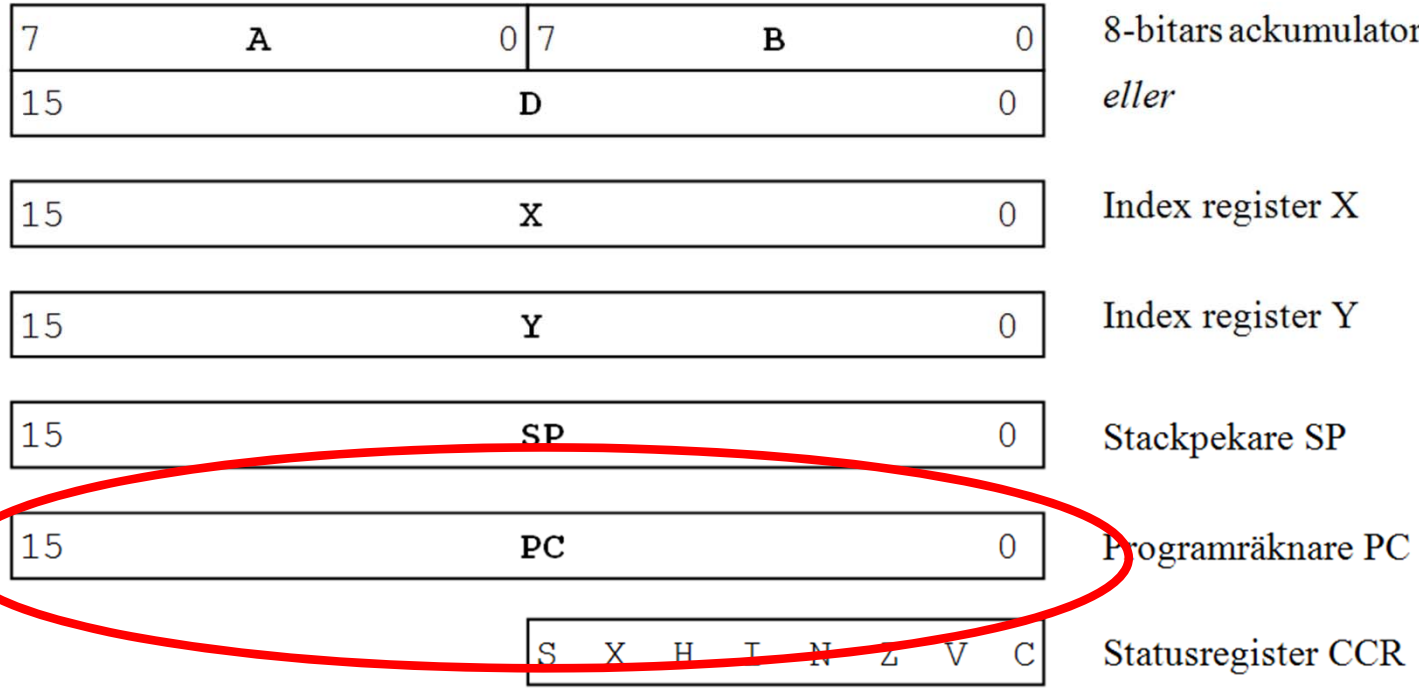





```
char     c;    /* 8-bitars datatyp, storlek byte */
short    s;    /* 16-bitars datatyp, storlek word */
long     l;    /* 32-bitars datatyp, storlek long */
int      i;    /* 16-bitars datatyp, storlek word */
```

Lämpliga arbetsregister för **`short och int`** är **`D`** och för **`char B`**
32 bitars datatyper ryms ej i något enstaka CPU12-register.

```
char  *cptr;      /* pekar på 8-bitars datatyp */
short *sptr;      /* pekar på 16-bitars datatyp */
int   *iptr;      /* pekar på 32-bitars datatyp */
```



8-bitars ackumulatorer A och B

eller

Index register X

Index register Y

Stackpekare SP

Programräknare PC

Statusregister CCR

ALLA pekartyper är 16 bitar
Lämpliga arbetsregister är **X** eller **Y**

Tilldelningar

Pseudo språk:

```
char  variable;

variable = 1;


.....


short variable

variable = 1;
```

Assemblerspråk:
kan kodas på flera olika sätt, exempelvis:

```
variable    RMB   1
1) MOVB     #1,variable

2) LDAB     #1
   STAB     variable

3) LDAA     #1
   STAA     variable
       .....

variable    RMB   2
1) MOVW     #1,variable

2) LDD      #1
   STD      variable
```

Addition av 8-bitars tal

Pseudo språk:

```
char  ca,cb,cc;
    ...
ca = cb + cc;
```

Assemblerspråk:

```
ca      RMB   1
cb      RMB   1
cc      RMB   1
        ...
        LDAB  cb     ; operand 1
        ADDB  cc     ; adderas
        STAB  ca     ; skriv i minnet
```

Addition av 16-bitars tal

Pseudo språk:

```
short sa,sb,sc;
    ...
sa = sb + sc;
```

Assemblerspråk:

```
sa      RMB   2
sb      RMB   2
sc      RMB   2
        ...
        LDD   sb      ; operand 1
        ADDD  sc      ; adderas
        STD   sa      ; skriv i minnet
```

Addition av 32-bitars tal

Pseudo språk:

```
long la,lb,lc;
    ...
la = lb + lc;
```

Assemblerspråk:

```
la      RMB   4
lb      RMB   4
lc      RMB   4
        ...
        LDD   lb+2  ; minst signifikanta “word” av b
        ADDD  lc+2  ; adderas till minst signifikanta “word” av c
        STD   la+2  ; tilldela, minst signifikanta “word”
        LDD   lb    ; mest signifikanta “word” av b
        ADCB  lc+1  ; adderas till låg byte av mest signifikanta
                    ; “word” av c
        ADCA  lc    ; adderas till hög byte av mest signifikanta
                    ; “word” av c
        STD   la    ; tilldela, mest signifikanta “word”
```

Kodförbättringar, framför allt för *byte*-operationer

Pseudo språk:

```
char  ca,cb;
     ...
ca = ca + 1;




cb = cb - 1;
```

Assemblerspråk:

```
ca      RMB   1
cb      RMB   1
        ...
        LDAB  ca
        ADDB  #1
        STAB  ca
eller
        INC   ca

        ...

        DEC   cb
```

Registerspill

Delresultat kan sparas på stacken vid evaluering av uttryck där processorns register inte räcker till...

EXEMPEL

```
unsigned short int _a,_b,_c,_d;
```

Evaluera: `(_a*_b)+(_c*_d);`

Lösning: För 16 bitars multiplikation använder vi EMUL-instruktionen.
Denna förutsätter att operanderna finns i D respektive Y-registren.

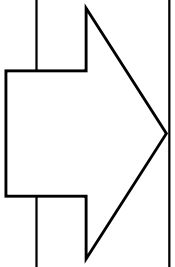
```
LDD     _a
LDY     _b
EMUL            ; första parentesen evaluerad
PSHD            ; placera delresultat på stacken
LDD     _c
LDY     _d
EMUL            ; andra parentesen evaluerad
ADDD    0,SP    ; addera med första delresultatet
LEAS    2,SP    ; återställ stackpekaren
```

Efter instruktionssekvensen finns hela uttryckets värde i register D,
stackpekaren har återställts till det värde den hade före instruktionssekvensen.

Permanenta och tillfälliga variabler

Pseudo språk:

```
char   gc;

Sub_0
{
       char   lc;

       lc = 5;
}
```



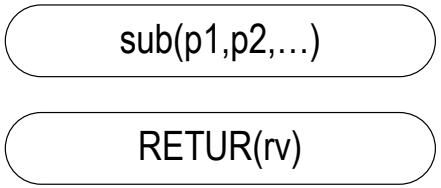
Assemblerspråk:

```
_gc:    RMB    1

Sub_0:
        LEAS   -1,SP

        LDAB   #5
        STAB   0,SP

        LEAS   1,SP
        RTS
```

I subrutinen refereras variabeln `lc` *som* `0,SP`.

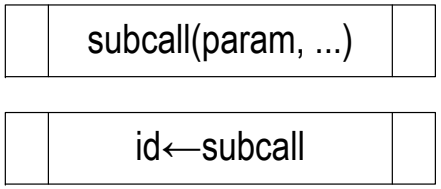
Som en direkt följd är variabeln **`gc`** ”synlig” hela tiden, i hela programmet medan variabeln **`lc`** endast är synlig (existerar) i subrutinen ”Sub_0”.

Flödesdiagram för programstrukturer



Inträde i och utträde ur subrutiner.

Inträde, typiskt med subrutinens namn och symbolisk representation av eventuella parametrar då sådana finns.
Utträde, (”RETUR”) typiskt med angivande av returnvärde (rv) om sådant finns.

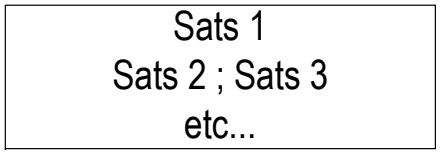


Subrutinanrop.

Med parametrar, symbolisk representation av eventuella aktuella parametrar som skickas med subrutinanropet.
Med returvärde, tilldelningsoperator placeras framför den anropade subrutinen.

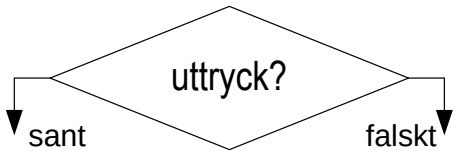
Engångsinitieringar.

Placeras typiskt i omedelbar anslutning till en inträdessymbol



Exekveringsblock.

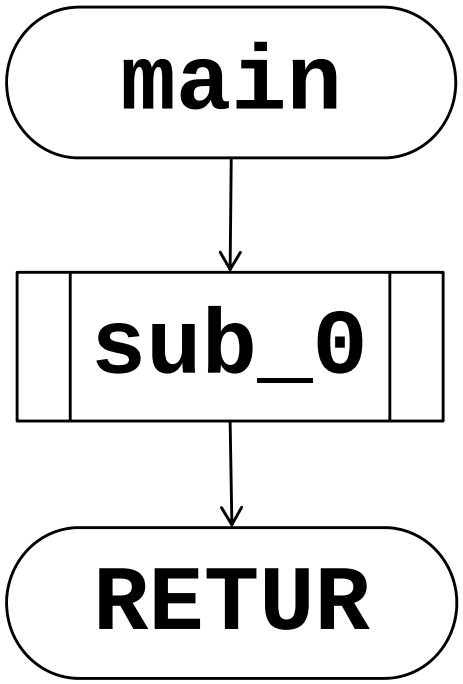
En eller flera satser som ordnats och exekveras sekvensiellt.



Villkorsblock.

Ett uttryck testas, utfallet kan vara sant eller falskt och exekveringsväg väljs därefter.

Programmering i assemblerspråk, programstrukturer



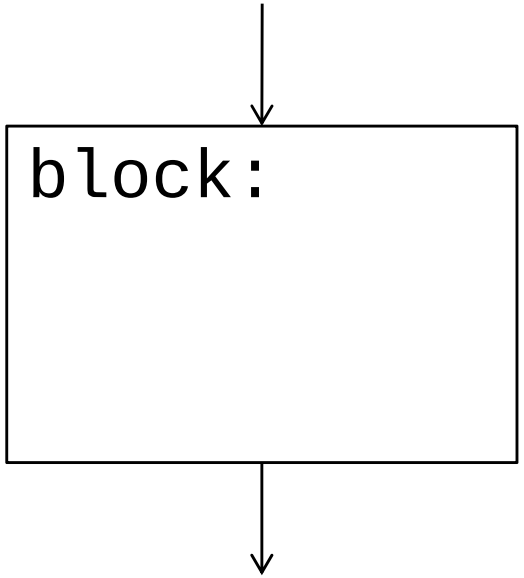
Pseudospråk:

```
main
{


sub_0();


}
```

Assemblerspråk:

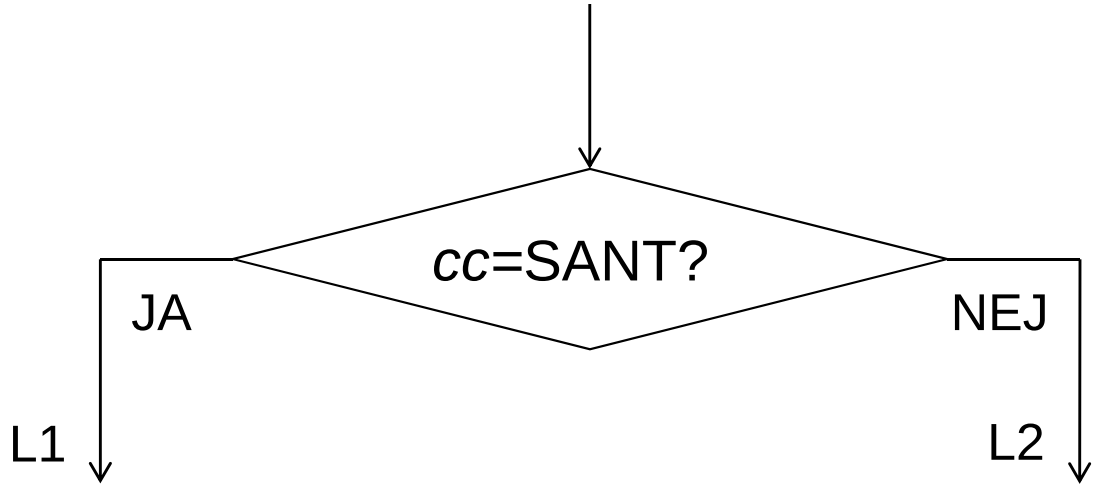
```
main:


        JSR    sub_0


        RTS
```

Sekvensiellt/villkorligt programflöde



Assemblerspråk:

```
        ...
block:instr 1
      instr 2
      instr 3
      ...
```

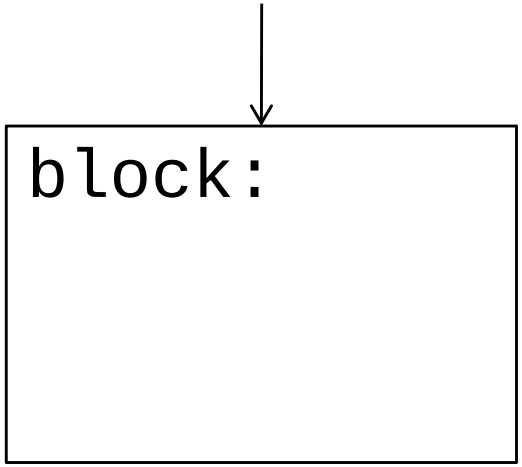


Assemblerspråk:

```
        ...
        Bcc    L1
L2:
        ...
        ...
L1:
```

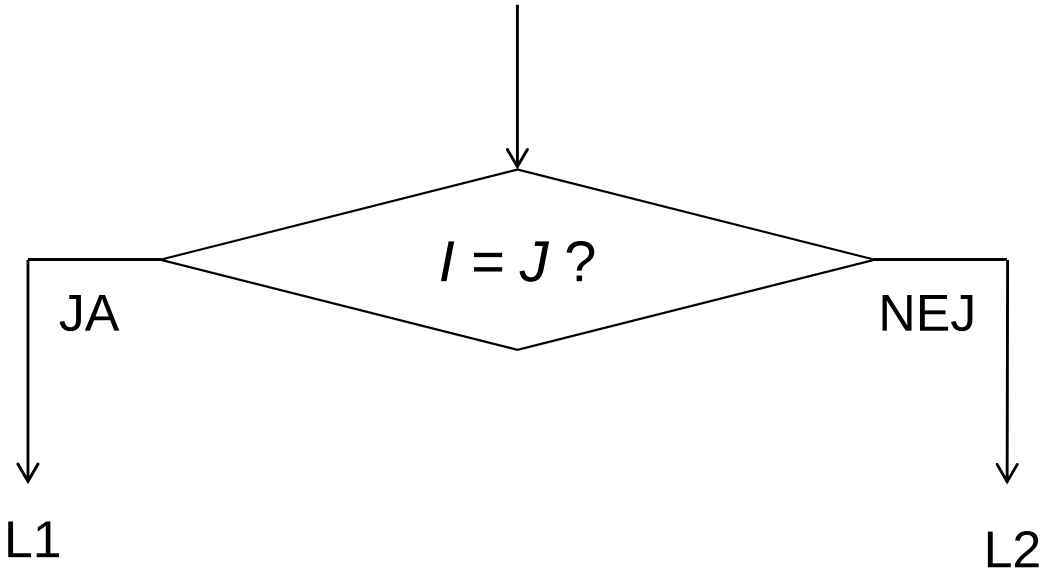
Programflödeskontroll

Ovillkorlig och villkorlig programflödesändring



Assemblerspråk:

```
        JMP    block
eller
        BRA    block
```

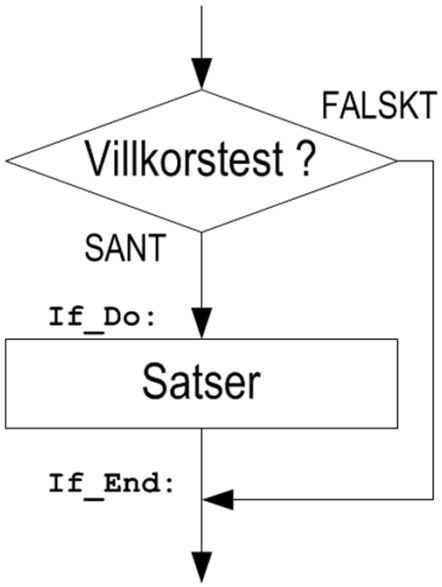


Assemblerspråk:

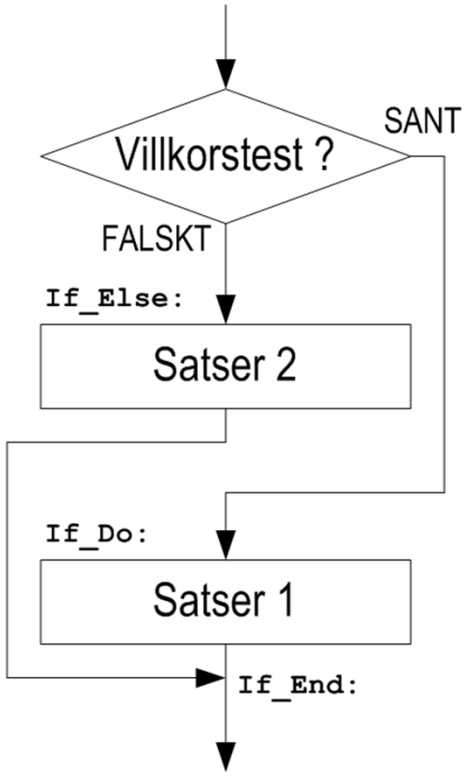
```
        LDD    I
        CMPD   J
        BEQ    L1
L2:     ...
```

Kontrollstrukturer

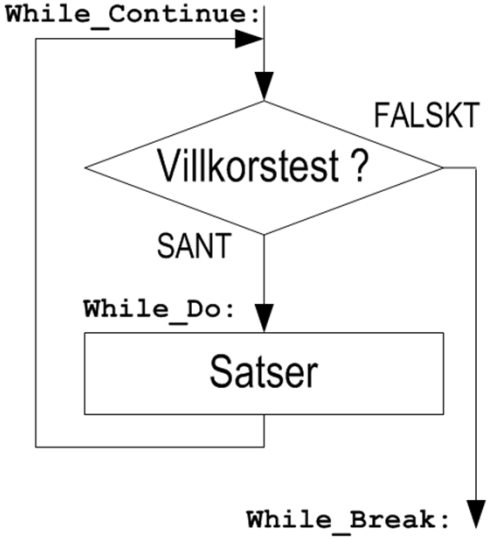
```
if( villkor )
{
  Satser
}
```



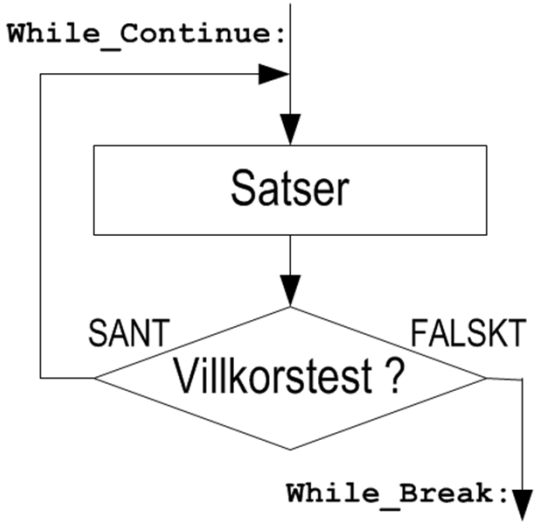

```
if( villkor ){
   Satser1
}else{
   Satser2
}
```



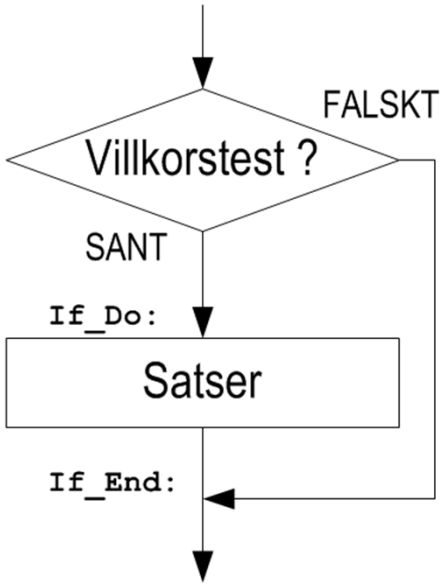

```
while( villkor )
{
  Satser
}
```




```
do{
  Satser
} while( villkor );
```



If (...) {...}



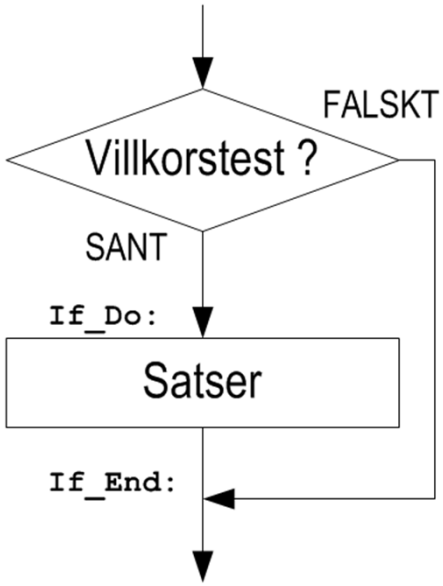

```
if (DipSwitch != 0)
   HexDisp = Dipswitch;
```

”Rättfram” kodning:

```
DipSwitch   EQU   $600
HexDisp     EQU   $400

            ...
            LDAB  DipSwitch
            CMPB  #0
            BNE   If_Do
            BRA   If_End

If_Do:      LDAB  DipSwitch
            STAB  HexDisp
If_End:
```

BNE	“Hopp” om ICKE *zero*	Z=0
BEQ	“Hopp” om *zero*	Z=1

If (...) {...}



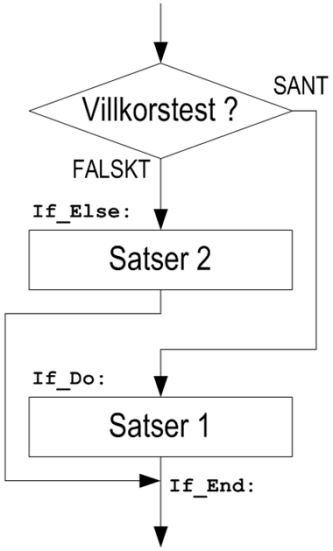

```
if (DipSwitch != 0)
   HexDisp = Dipswitch;
```

Användning av komplementärt villkor leder till bättre kodning:

```
DipSwitch    EQU    $600
HexDisp      EQU    $400

             ...
             LDAB   DipSwitch
             CMPB   #0
             BEQ    If_End


If_Do:       LDAB   DipSwitch
             STAB   HexDisp
If_End:
```

BNE	"Hopp" om ICKE *zero*	Z=0
BEQ	"Hopp" om *zero*	Z=1

If (...) {...} else { ...}

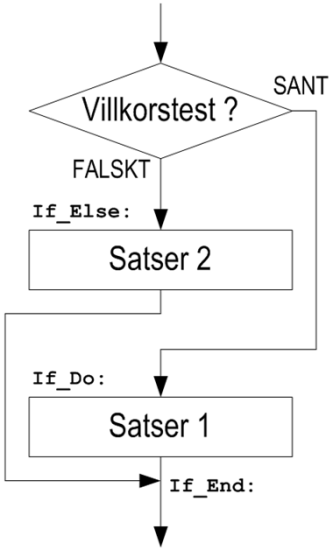



```
if (DipSwitch >= 5)
        HexDisp = 1;
else
        HexDisp = 0;
```

```
DipSwitch    EQU   $600
HexDisp      EQU   $400

             ...
             LDAB  DipSwitch
             ...
             CMPB  #5
             BHS   If_Do
If_Else:     LDAB  #0
             STAB  HexDisp
             BRA   If_End

If_Do:       LDAB  #1
             STAB  HexDisp
If_End:      ...
```

Jämförelser av tal utan tecken		
BHS (BCC)	Villkor: R≥M	C=0
BLO (BCS)	Villkor: R<M	C=1

If (...) {...} else { ...}

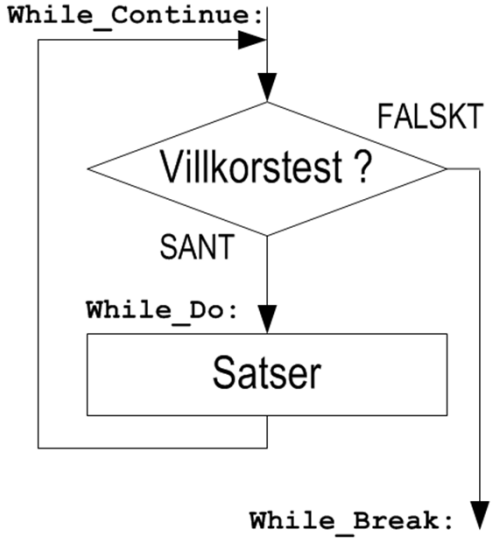



```
if (DipSwitch >= 5)
        HexDisp = 1;
else
        HexDisp = 0;
```

```
DipSwitch    EQU   $600
HexDisp      EQU   $400

             ...
             LDAB  DipSwitch
             ...
             CMPB  #5
             BLO   If_Else
If_Do:       LDAB  #1
             STAB  HexDisp
             BRA   If_End

If_Else:     LDAB  #0
             STAB  HexDisp
If_End:      ...
```

Jämförelser av tal utan tecken		
BHS (BCC)	Villkor: R≥M	C=0
BLO (BCS)	Villkor: R<M	C=1

while (...) {...}



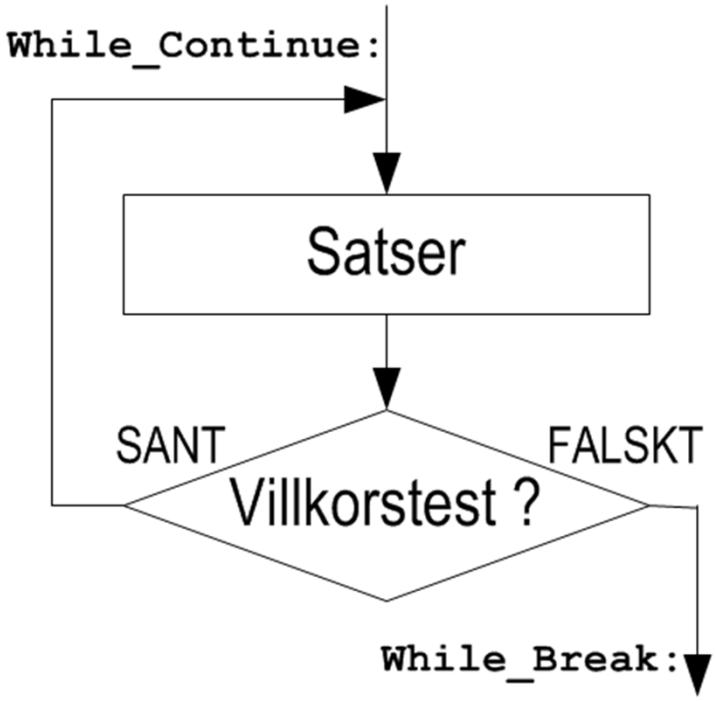

```
while (DipSwitch != 0)
       HexDisp = 1;
HexDisp = 0;
```

Vid kodning av ”while”-iteration används det komplementära villkoret

```
DipSwitch    EQU   $600
HexDisp      EQU   $400

             ...

While_Continue:
             LDAB  DipSwitch
             CMPB  #0
             BEQ   While_Break
             LDAB  #1
             STAB  HexDisp
             BRA   While_Continue
While_Break:
             LDAB  #0
             STAB  HexDisp
```

do {...} while (...)

Vid kodning av ”do-while”-iteration används uttryckets villkor




```
DipSwitch   EQU   $600
HexDisp     EQU   $400
                  ...
While_Continue:
            CLR   HexDisp
                  ...
            LDAB  DipSwitch
            CMPB  #0
            BNE    While_Continue
While_Break:
                  ...
```

```
do
{
      HexDisp = 0;
}while (DipSwitch != 0);
```

Sammanfattning, villkorlig programflödeskontroll

C-operator	Betydelse	Datatyp	Instruktion
==	Lika med	signed/unsigned	BEQ
!=	Skild från	signed/unsigned	BNE
<	Mindre än	signed	BLT
		unsigned	BCS
<=	Mindre än eller lika	signed	BLE
		unsigned	BLS
>	Större än	signed	BGT
		unsigned	BHI
>=	Större än eller lika	signed	BGE
		unsigned	BCC

Parameteröverföring till, och returvärden från subrutiner

Parametrar

- Via register

  enkla datatyper, snabbt, effektivt och enkelt

- Via stacken

  generellt

Returvärden

- Via register

  för enkla datatyper som ryms i processorns register

- Via stacken

  sammansatta datatyper (poster och fält)

Parameteröverföring via register

Antag att vi alltid använder register D, X (i denna ordning) för parametrar som skickas till subrutinen ”Sub_0”.
Då kan funktionsanropet

```
Sub_0(la,lb);
```

översättas till:

```
LDD   la
LDX   lb
BSR   Sub_0
```

Då vi kodar subrutinen ”Sub_0” vet vi (på grund av våra regler) att den första parametern finns i D, och den andra i X.

Metoden är enkel och ger bra prestanda men är begränsad i antal parametrar som kan överföras.

Tänkbar komplikation:
”Registerspill” i den anropade subrutinen?
Skapa ”lokala variabler” – använd stacken för temporär lagring

```
; Sub_0(la,lb);

Sub_0:
; parametrar finns i register,
; spara dessa på stacken (behöver registren)
        STD     2,-SP   ;(Push D)
        STX     2,-SP   ;(Push X)
        ----    här används registren
        ----    för andra syften
; återställ parametrar från stacken
        LDD     2,SP
        LDX     0,SP
        ---
        ---
        LEAS    4,SP    ; återställ stackpekare
        RTS
```

Låt oss anta att SP har värdet 3000 vid inträdet i subrutinen

Adress	Innehåll	SP före	SP efter
3000		◀	
2FFF	D.lsb		
2FFE	D.msb		
2FFD	X.lsb		
2FFC	X.msb		◀
2FFB			

Parameteröverföring via stacken

Antag att listan av parametrar som skickas till en subrutin behandlas från höger till vänster. Då kan funktionsanropet:

```
Sub_0(la,lb);
```

översättas till:

```
LDD     lb
PSHD    ; (STD  2,-SP)
LDD     la
PSHD
BSR     Sub_0
LEAS    4,SP
```

Stackens utseende		
Innehåll	Kommentar	Adressering via SP i subrutinen
lb.lsb	Parameter lb	
lb.msb		4,SP
la.lsb	Parameter la	
la.msb		2,SP
PC.lsb	Återhoppsadress,	
PC.msb	placeras här vid BSR	0,SP

```
Sub_0:
          . .
          LDD      2,SP
; parameter la till register D
          . .
          LDD      4,SP
; parameter lb till register D
          . .
```

Returvärden via register

Register väljs, beroende på returvärdets typ (storlek).
En regel (konvention) bestäms och följs därefter vid kodning av alla subrutiner

EXEMPEL:

Storlek	Benämning	C-typ	Register
`8 bitar`	`byte`	`char`	`B`
`16 bitar`	`word`	`short int`	`D`
`32 bitar`	`long`	`long int`	`Y/D`

Returvärden via stack

Krävs typiskt då en subrutin ska returnera en komplett post, eller ett helt fält. Detta är avsevärt mer komplicerat och det finns flera olika metoder.

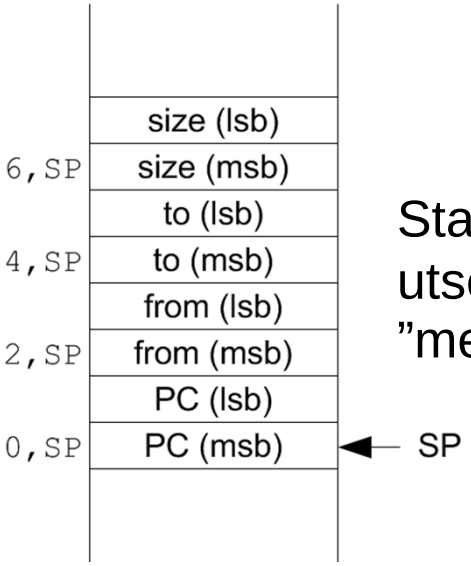
Exempelvis kan den anropande subrutinen reservera utrymme på stacken och skicka en pekare till detta utrymme som en *icke synlig* parameter.
gcc och xcc gör på detta vis men man måste alltid konsultera dokumentationen för den använda kompilatorn.

Kodningsexempel: subrutinen ”memcpy(from , to, size)”

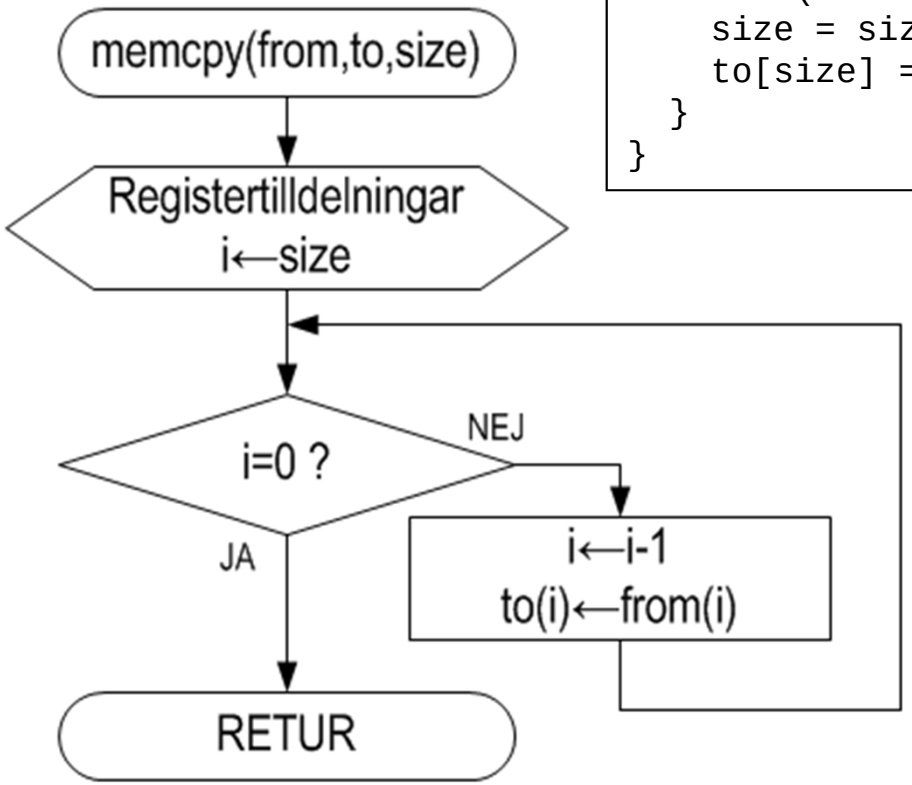
Exempel på anrop (formellt):

```
        PUSH    size
        PUSH    to
        PUSH    from
        JSR     memcpy
        LEAS    6,SP
```

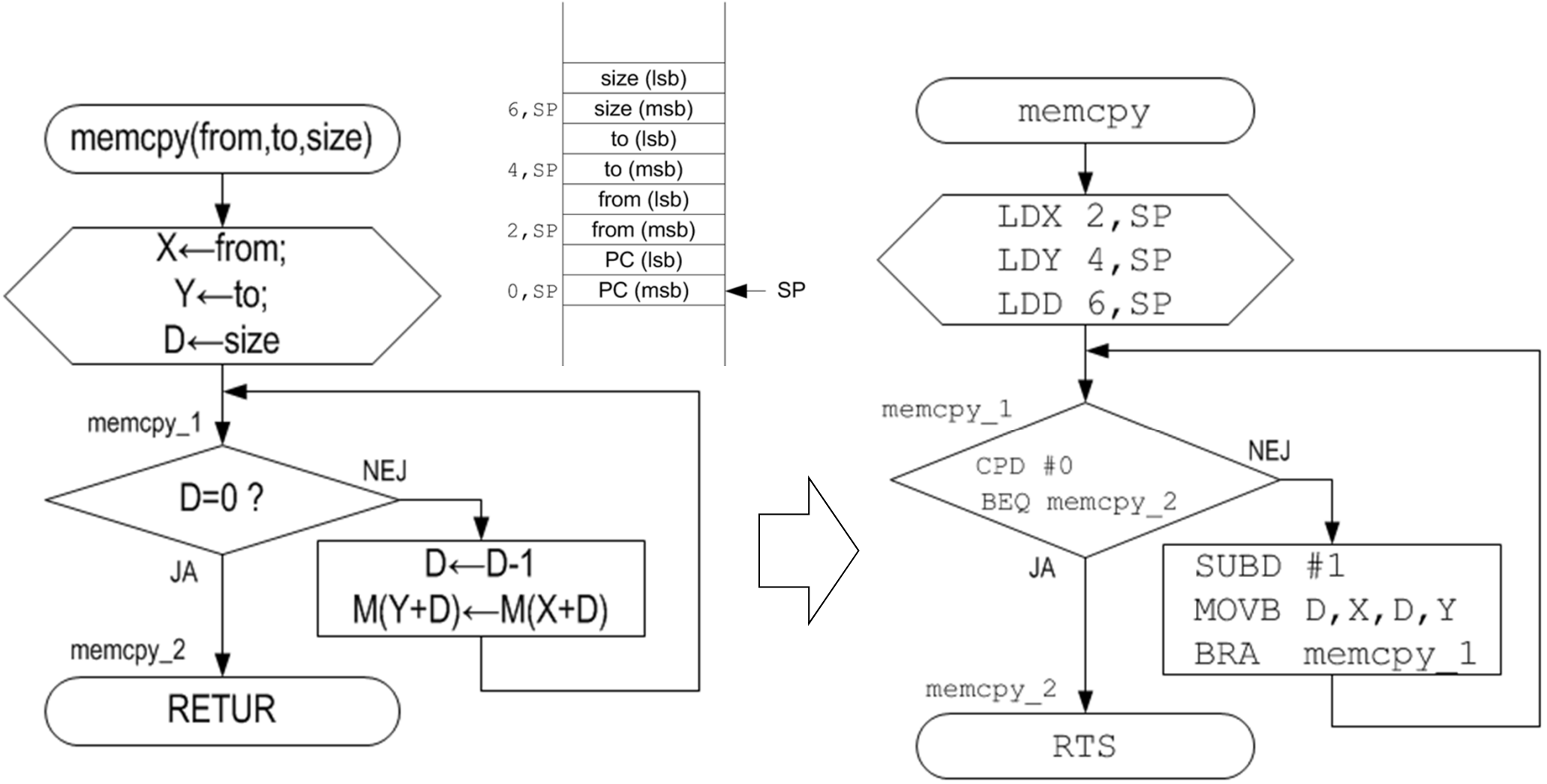
```
void memcpy(  unsigned char from[],
              unsigned char to[],
              unsigned int size )
{
  while (size > 0){
    size = size - 1;
    to[size] = from[size];
  }
}
```



Stackens utseende i ”memcpy”



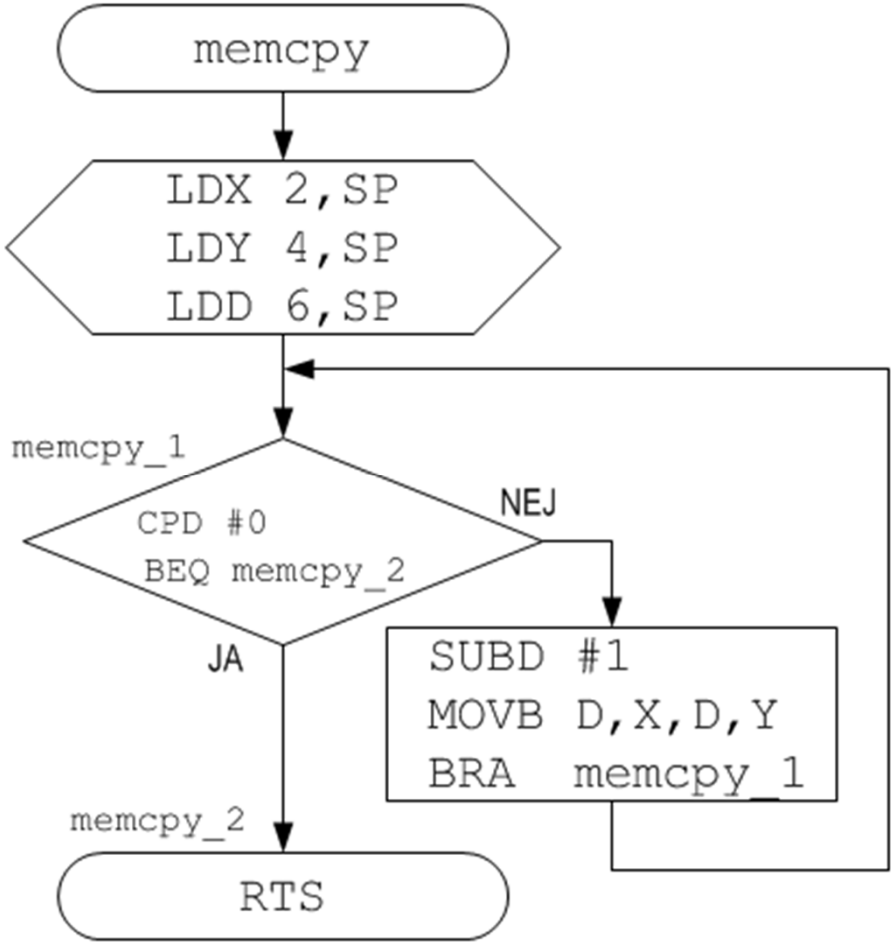
forts.”memcpy(from , to, size)”

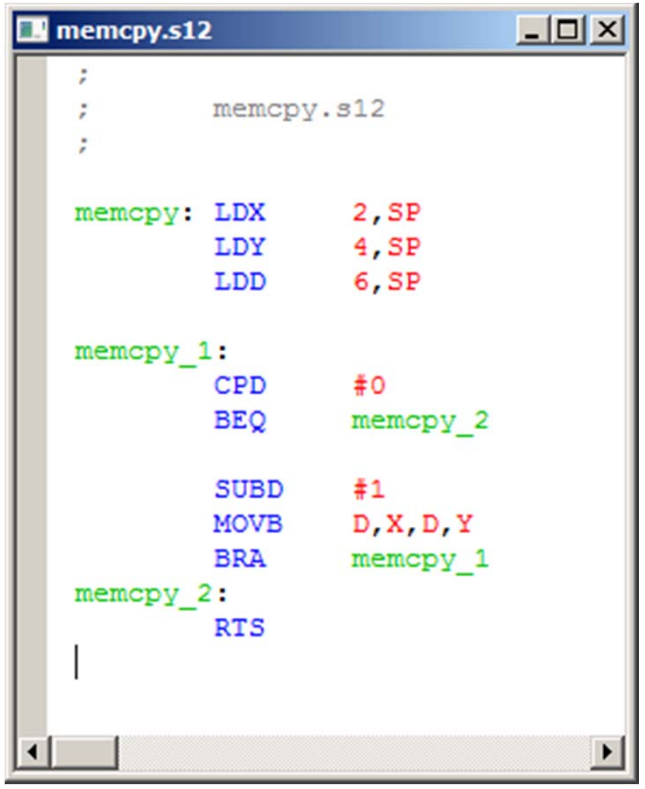


Efter registerallokering...

Efter implementering...

forts.”memcpy(from , to, size)”





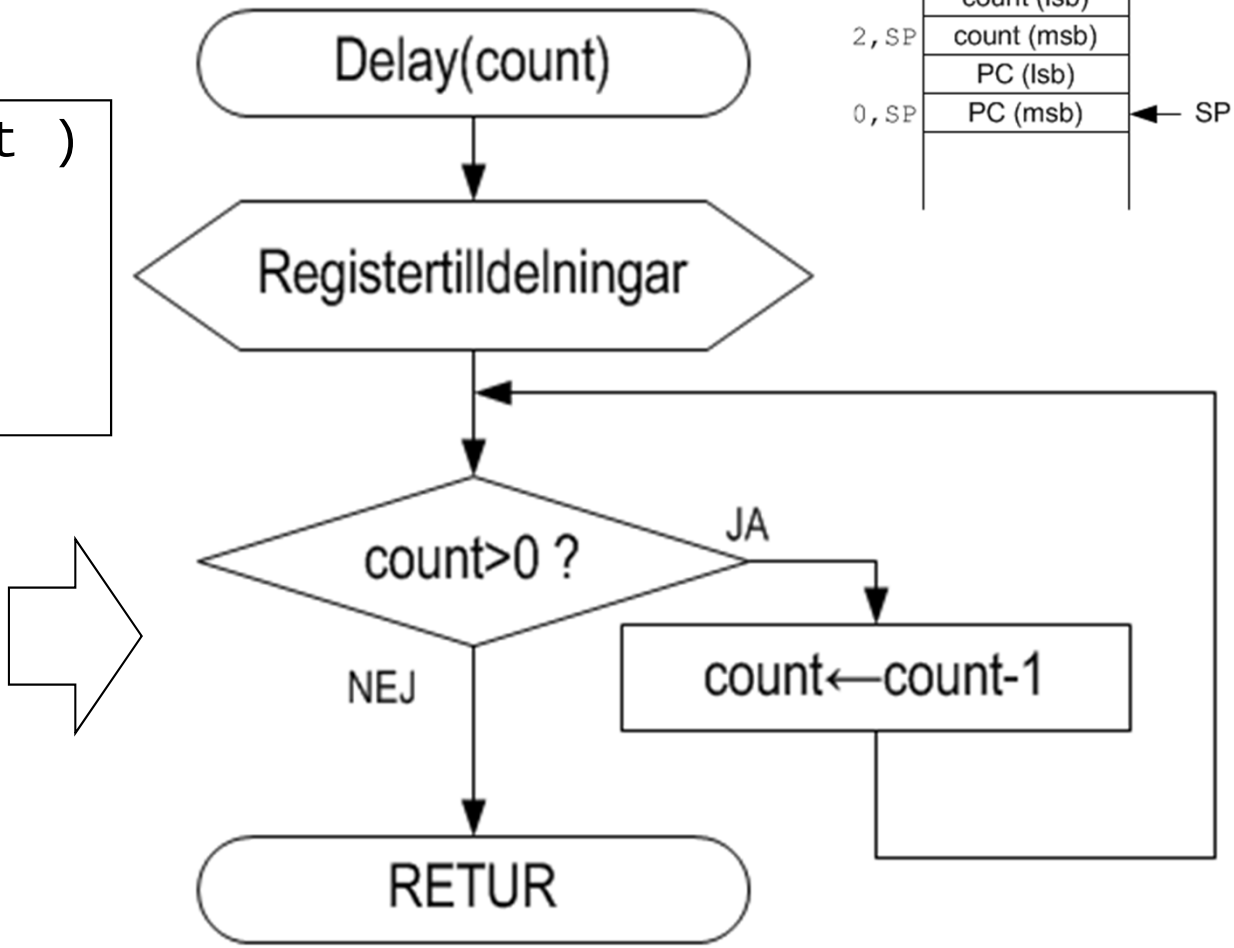
Efter linearisering...

Kodningsexempel, fördröjning: subrutinen ”Delay(count)”

```
void Delay( unsigned int count )
{
  while (count > 0)
    count = count - 1;
}
```

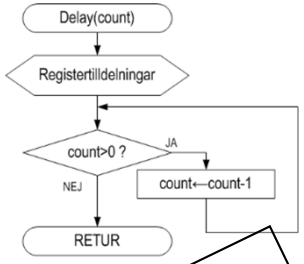
Exempel på anrop :

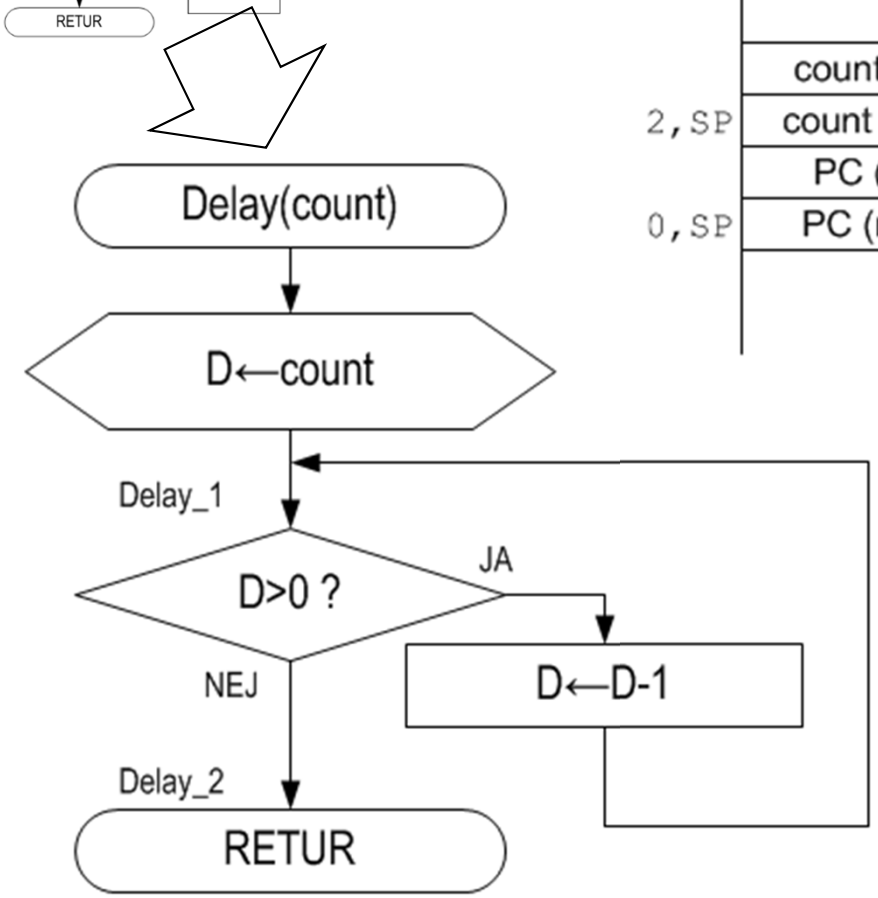
```
        LDD     #8000
        PSHD
        JSR     Delay
        LEAS    2,SP
```

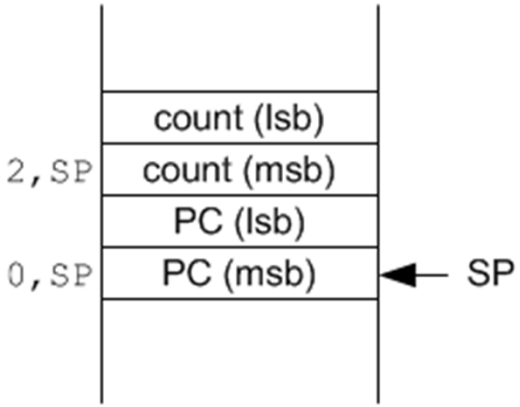


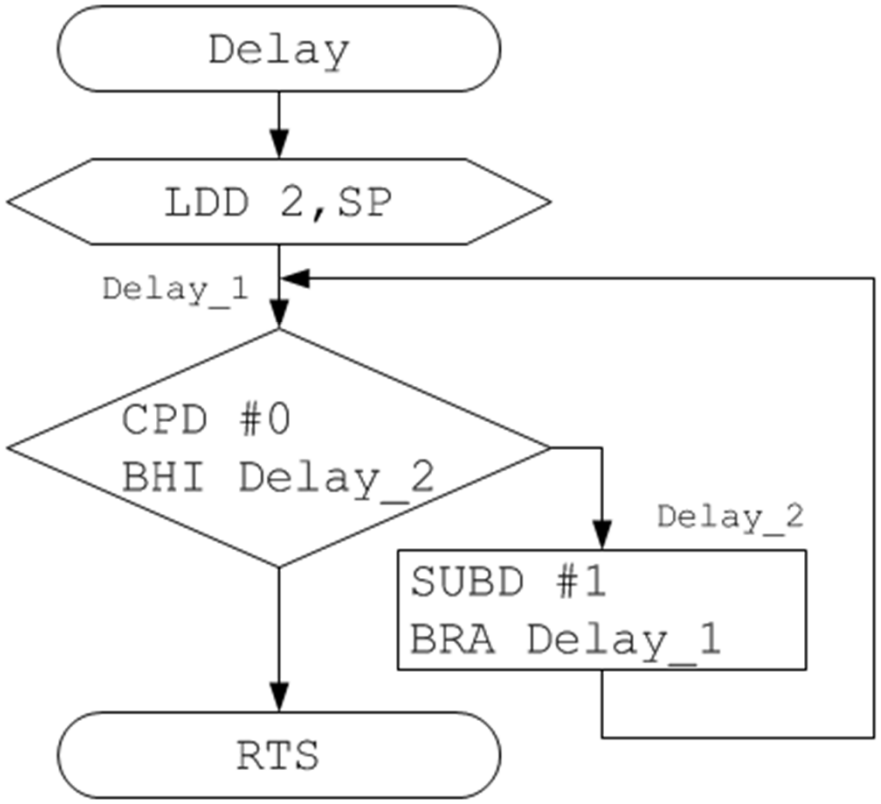
forts.”Delay(count)”

>	Större än	signed	BGT
		unsigned	BHI

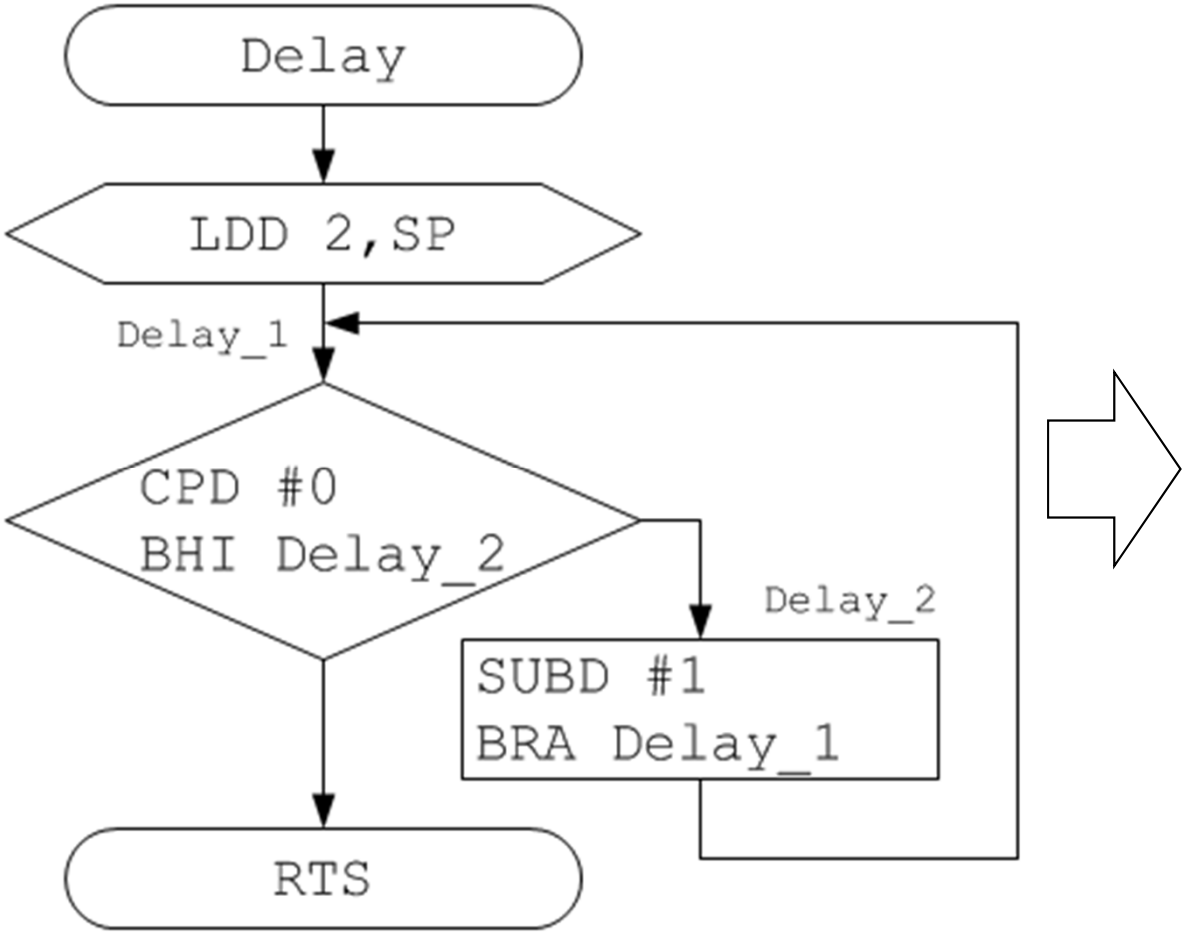








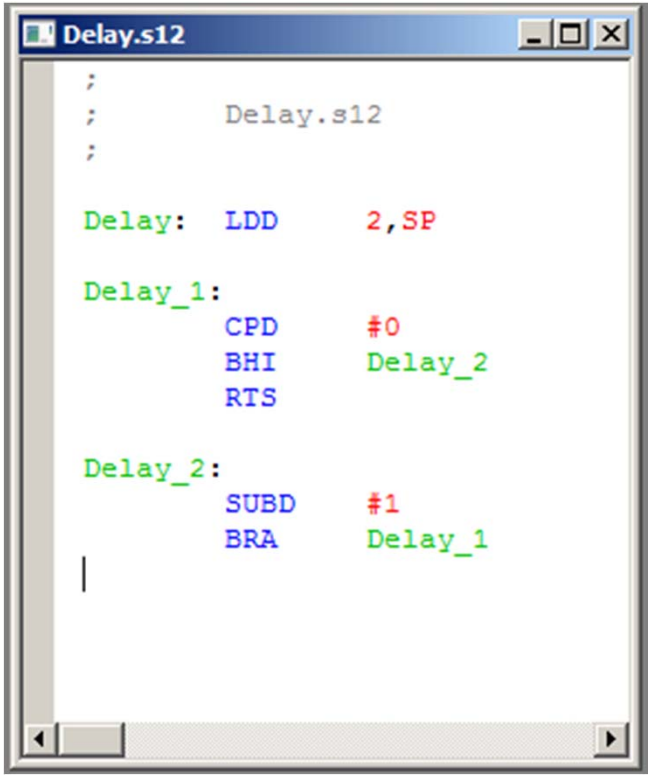
forts.”Delay(count)”



Delay.s12

```
;
;         Delay.s12
;

Delay:    LDD       2,SP

Delay_1:
          CPD       #0
          BHI       Delay_2
          RTS

Delay_2:
          SUBD      #1
          BRA       Delay_1
```

ET (”execution time”) statisk analys: Bestäm subrutinens fördröjning i realtid



instruktion	antal ggr.
LDD	1
CPD	count+1
BHI	count (”taken”)
BHI	1 (”not taken”)
SUBD	count
BRA	count
RTS	1

= LDD (1)
+ CPD (count+1)
+ BHI_T (count)
+ BHI_{NT} (1)
+ SUBD (count)
+ BRA (count)
+ RTS (1)
= ?

Antal cykler

```
= LDD (1)
+ CPD (count+1)
+ BHI_T (count)
+ BHI_NT (1)
+ SUBD (count)
+ BRA (count)
+ RTS (1)
=  ?
```

Antalet cykler för respektive instruktion fås ur handboken

instruktion	# cykler
LDD n,SP	3
CPD #	2
SUBD #	2
BHI	3/1
RTS	5
BRA	3

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
SUBD *#opr16i*	IMM	83 jj kk	PO	OP
SUBD *opr8a*	DIR	93 dd	RPf	RfP
SUBD *opr16a*	EXT	B3 hh ll	RPO	ROP
SUBD *oprx0_xysp*	IDX	A3 xb	RPf	RfP
SUBD *oprx9,xyssp*	IDX1	A3 xb ff	RPO	RPO
SUBD *oprx16,xysp*	IDX2	A3 xb ee ff	fRPP	fRPP
SUBD [D,*xysp*]	[D,IDX]	A3 xb	fIfRPf	fIfRfP
SUBD [*oprx16,xysp*]	[IDX2]	A3 xb ee ff	fIPRPf	fIPRfP

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BHI *rel8*	REL	22 rr	PPP/P(1)	PPP/P(1)

1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
RTS	INH	3D	UfPPP	UfPPP

```
= (LDD)3(1)
+ (CPD)2(count+1)
+ (BHItaken)3(count)
+ (BHInottaken)1(1)
+ (SUBD)2(count)
+ (BRA)3(count)
+ (RTS)5(1)
= 10×count+11
```

Subrutinens exekveringstid:

$ET(\text{"Delay"}) = (11 + 10 \times count) \times cykeltid$

```
Delay( unsigned int count )
{
  while (count > 0)
    count = count - 1;
}
```

Vi kan nu bestämma minimala och maximala fördröjningar vid olika klockfrekvenser (cykeltider)

	Frekvens/ cykeltid	Min. ('count' = 0) 11 cykler	Max. ('count' = $FFFF) 65535+11 cykler
	4 MHz/250 ns.	2,75 μs	164 ms
MC12	8 MHz/125 ns.	1,375 μs	82 ms
	16 MHz/62,5 ns.	687,5 ns	41 ms
	25 MHz/40 ns.	440 ns	26 ms

Exempel: Bestäm ’count’ för 10 ms fördröjning i ett MC12-system

	Frekvens/ cykeltid	Min. (’count’ = 1) 11 cykler	Max. (’count’ = $FFFF) 65535+11 cykler
MC12	8 MHz/125 ns.	1,375 μs	82 ms

Lösning:

$10\ ms = (11 + 10 \times count) \times 125\ ns$

$10 = (11 + 10 \times count) \times 125\ 10^{-6}$

$(10 \times 10^{6}) / 125 = (11 + 10 \times count)$

$((10^{7} / 125)-11)/10 = count$

$count = 7998,9 \approx 8000$

SI-enheter:

$ms = s \times 10^{-3}$

$ns = s \times 10^{-9}$

Uppskatta motsvarande fördröjning i ETERM/XCC simulatorn

... Tar c:a 10 sekunder