Maskinorienterad Programmering 2010/11

Maskinnära programmering – en introduktion

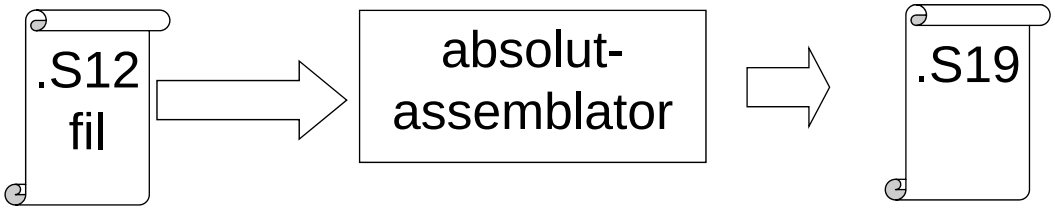
Ur innehållet:

- Assemblatorn, assemblerspråk
- Datatyper
- Tilldelningar, unära och binära operationer
- Permanenta/tillfälliga variabler
- Programkonstruktion i assemblerspråk, programstrukturer
- Programflödeskontroll

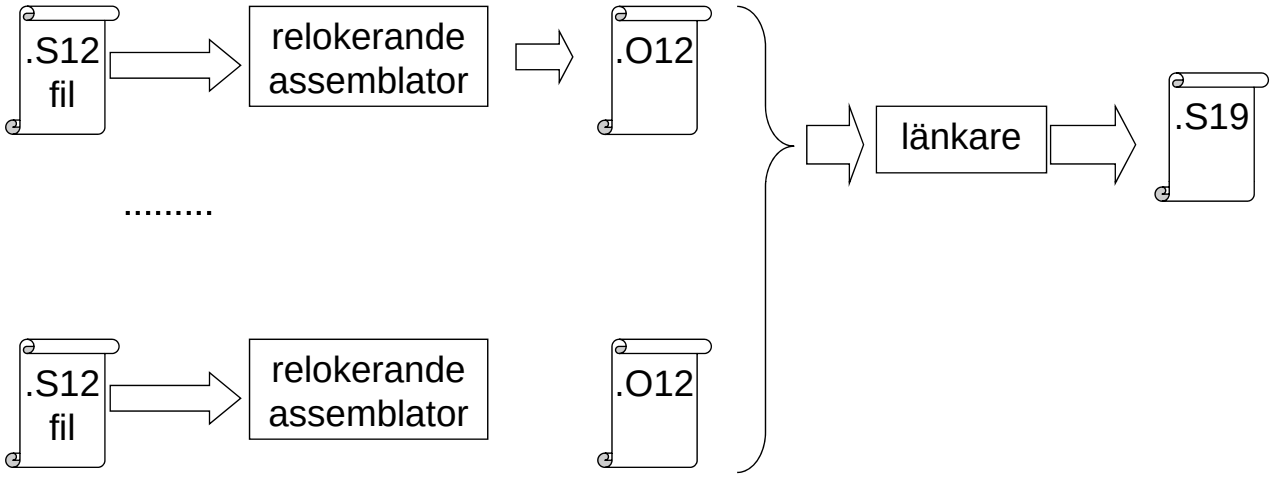
Absolut assemblering

All källtext assembleras samtidigt och alla referenser löses upp omedelbart.

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

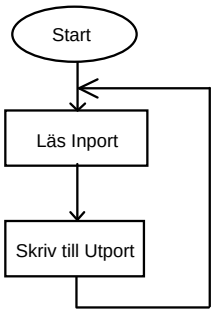


Relokerande assemblator



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

Assemblerprogrammets struktur; exempel




```
; 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
```

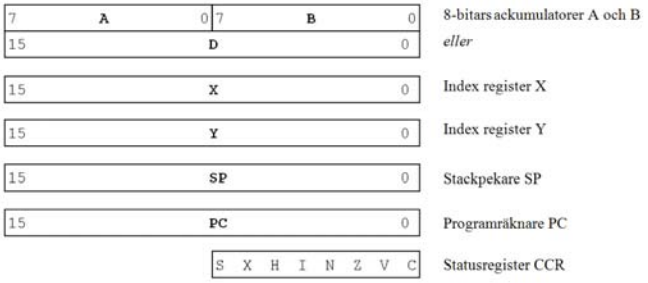
Syntaktiskt korrekt men extremt svårläst på grund utav illa valda symbolnamn...

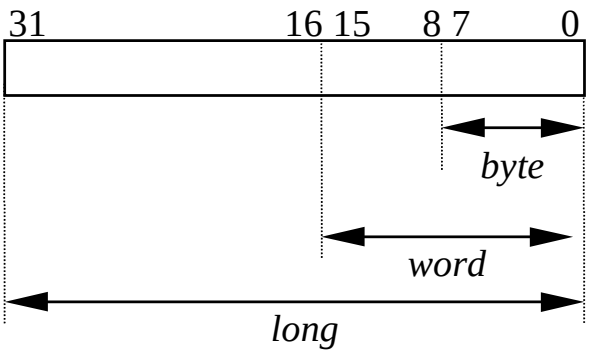
Ett bra exempel...

```
            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

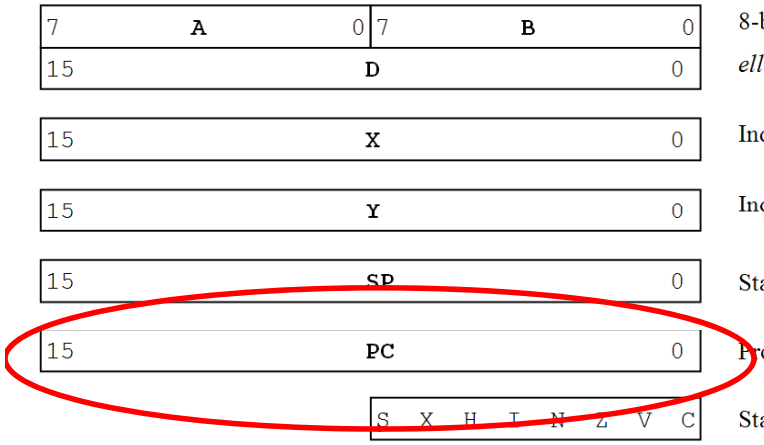





```
char      c;    /* 8-bitars datatyp, storlek byte */
short     s;    /* 16-bitars datatyp, storlek word */
long      l;    /* 32-bitars datatyp, storlek long */
int       i;    /* storlek implementationsberoende */
```

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

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



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 förutsäller 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   ca;

Main
{
       char   la;

       la = 5;
}
```

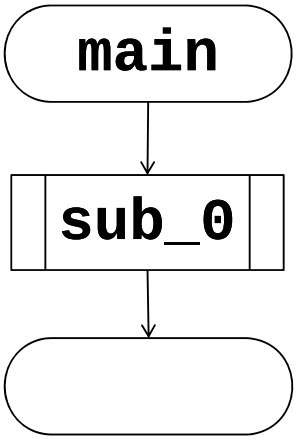
Assemblerspråk:

```
ca      RMB    1

Main:
        LEAS   -1,SP

        LDAB   #5
        STAB   0,SP

        LEAS   1,SP
        RTS
```

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

Som en direkt följd är variabeln **`ca`** ”synlig” hela tiden, i hela programmet medan variabeln **`la`** endast är synlig (existerar) i subrutinen Main.

Programmering i assemblerspråk, programstrukturer



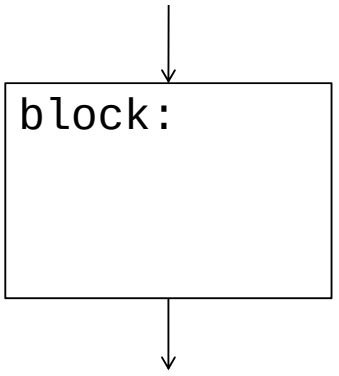
Pseudospråk:

```
main
{

sub_0();

}
```

Assemblerspråk:

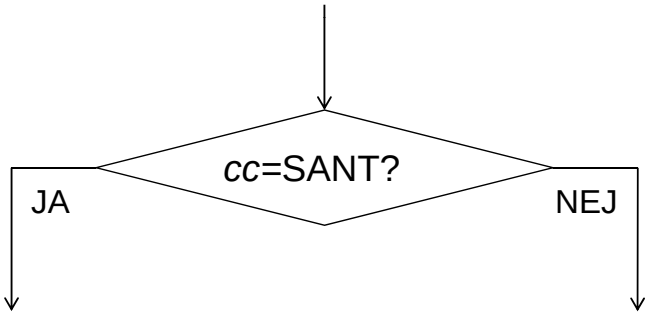
```
main:

        JSR    sub_0

        RTS
```


Sekvensiellt/villkorligt programflöde



Assemblerspråk:

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

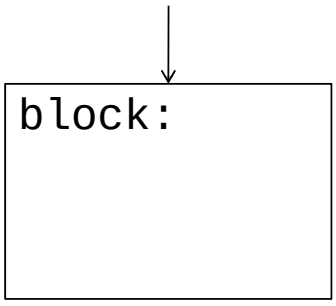


Assemblerspråk:

```
        ...
        Bcc    L2
        ...
```

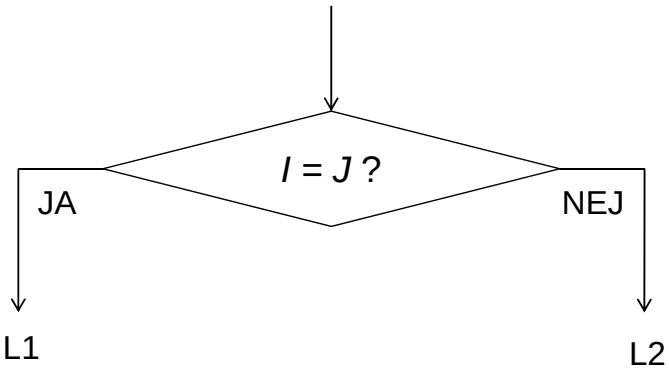
Programflödeskontroll

Ovillkorlig och villkorlig programflödesändring



Assemblerspråk:

```
        JMP    block
eller
        BRA    block
```

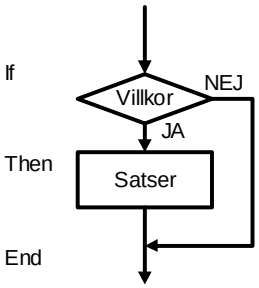


Assemblerspråk:

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

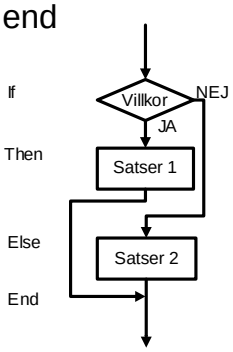

Kontrollstrukturer

If(Villkor) then ...



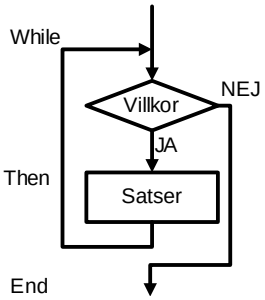

```
if(Villkor)
{
  Satser;
}
```

if(Villkor) then ...
else ...
end



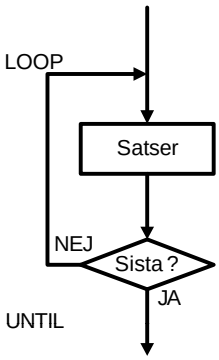

```
if(Villkor)
{
  Satser1;
}else{
  Satser2;
}
```

while(Villkor)
loop



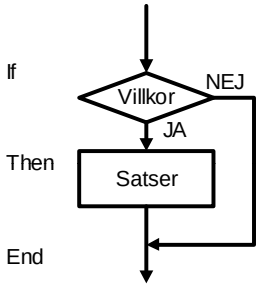

```
while(Villkor)
{
  Satser;
}
```

loop
...
until(Villkor)




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

If (...) {...}



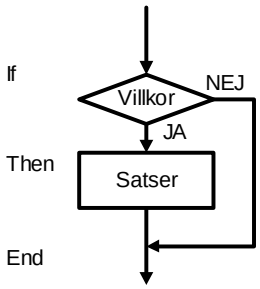

```
if (DipSwitch != 0)
   HexDisp = Dipswitch;
```

”Rättfram” kodning...

```
DipSwitch    EQU   $600
HexDisp      EQU   $400

             ...
             TST   DipSwitch
             BNE   assign
             BRA   end

assign       LDAB  DipSwitch
             STAB  HexDisp
end:
```

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

If (...) {...}



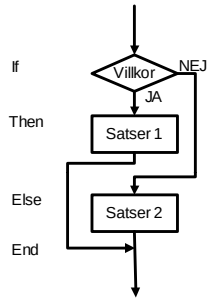

```
if (DipSwitch != 0)
   HexDisp = Dipswitch;
```

Bättre kodning...

```
DipSwitch    EQU   $600
HexDisp      EQU   $400

             ...
             TST   DipSwitch
             BEQ   end

             LDAB  DipSwitch
             STAB  HexDisp
end:
```

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

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

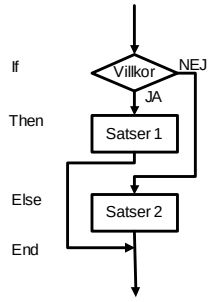



```
if (DipSwitch == 0)
      HexDisp = 1;
else
      HexDisp = 0;
```

```
DipSwitch   EQU   $600
HexDisp     EQU   $400

            ...
            LDAB  DipSwitch
            ...
            TSTB
            BEQ   not_else
            LDAB  #0
            STAB  HexDisp
            BRA   end
not_else:   LDAB  #1
            STAB  HexDisp

end:
```

BEQ	"Hopp" om *zero*	Z=1

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

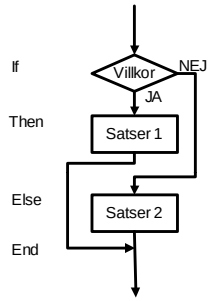



```
if (DipSwitch == 0)
      HexDisp = 1;
else
      HexDisp = 0;
```

```
DipSwitch   EQU   $600
HexDisp     EQU   $400

            ...
            LDAB  DipSwitch
            ...
            TSTB
            BNE   else
            LDAB  #1
            STAB  HexDisp
            BRA   end
else:       LDAB  #0
            STAB  HexDisp

end:
```

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

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



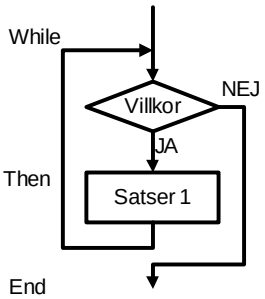

```
if (DipSwitch >= 5)
      HexDisp = 1;
else
      HexDisp = 0;
```

```
DipSwitch   EQU   $600
HexDisp     EQU   $400

            ...
            LDAB  DipSwitch
            ...
            CMPB  #5
            BHS   then
            LDAB  #0
            STAB  HexDisp
            BRA   end
then:       LDAB  #1
            STAB  HexDisp

end:
```

Test av tal utan tecken		
BHS	Villkor: R≥M	C=0

Test av tal med tecken		
BGE	Villkor: R≥M	N ⊕ V = 0

while (...) {...}

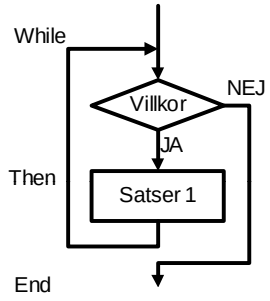



```
while (DipSwitch != 0)
      HexDisp = 1;
HexDisp = 0;
```

```
DipSwitch   EQU   $600
HexDisp     EQU   $400

            ...

while:
            LDAB  DipSwitch
            ...
            TSTB
            BEQ   end_while
            LDAB  #1
            STAB  HexDisp
            BRA   while
end_while:
            LDAB  #0
            STAB  HexDisp
```

BEQ	"Hopp" om *zero*	Z=1

while (...) {...}



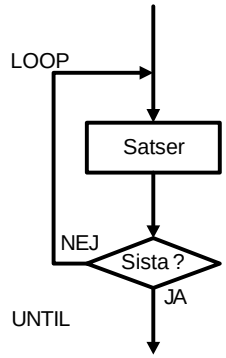

```
Delay( unsigned int count )
{
  while (count > 0)
    count = count - 1;
}
```

```
Delay:      LDD   ”count”
Delay_loop:
            NOP
            ...
            NOP
            SUBD  #1
            BHI   Delay_loop
Delay_end:  RTS
```

Test av tal utan tecken		
BHI	Villkor: R>M	C + Z = 0

Test av tal med tecken		
BGT	Villkor: R>M	Z + (N ⊕ V) = 0

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




```
do
{
      HexDisp = 0;
}while (DipSwitch >= 10);
```

```
DipSwitch   EQU   $600
HexDisp     EQU   $400
            ...
do:
            MOVB  #0,HexDisp

            LDAB  DipSwitch
            CMPB  #10
            BHS   do
            ...
```

Test av tal utan tecken		
BHS	Villkor: R≥M	C=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

Fördröjning

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

Parameter 'count' finns i register D vid anrop. Anm. count=0 är EJ TILLÅTET.

```
; Subrutin 'Delay'
Delay:          NOP
Delay_loop:     NOP
                NOP
                SUBD    #1
                BHI     Delay_loop
                RTS
```

instruktion	antal ggr.
NOP	1
NOP	count
NOP	count
SUBD #1	count
BHI	count ("taken")
BHI	1 (not taken)
RTS	1

= NOP (1 + 2 count)
+ SUBD#1 (count)
+ BHI_T (count-1)
+ BHI_{NT} (1)
+ RTS (1)
= ?

```
= NOP ( 1 + 2 count)
+ SUBD#1 ( count )
+ BHI_T   ( count )
+ BHI_NT ( 1 )
+ RTS ( 1 )
=  ?
```

(exekveringstider, dvs antal cykler, fås ur handboken...)

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
NOP	INH	A7	O	O

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	ROP
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

instruktion	# cykler
NOP	1
SUBD #1	2
BHI	3/1
RTS	5

```
= 1 ( 1 + 2 count)
+ 2 ( count )
+ 3 ( count-1 )
+ 1 ( 1 )
+ 5 ( 1 )
= 7 count + 4
```


Minimal/maximal fördröjning vid olika klockfrekvenser

Frekvens/ cykeltid	Min. ('count' = 1) 11 cykler	Max. ('count' = $FFFF) 458749 cykler
4 MHz/ 250 ns.	2,75 µs	115 ms
8 MHz/ 125 ns.	1,375 µs	57,34 ms
16 MHz/ 62,5 ns.	687,5 ns	28,67 ms
25 MHz/ 40 ns.	440 ns	18,35 ms

MC12

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

	Frekvens/ cykeltid	Min. ('count' = 1) 11 cykler	Max. ('count' = $FFFF) 458749 cykler
MC12	8 MHz/ 125 ns.	1,375 µs	57,34 ms

Lösning:

$$(7count + 4)125ns = 10ms$$

$$(7count + 4)125ns = 10000000ns$$

$$count = \frac{\frac{10000000}{125} - 4}{7} = 11428$$

Uppskatta motsvarande fördröjning i simulatorn

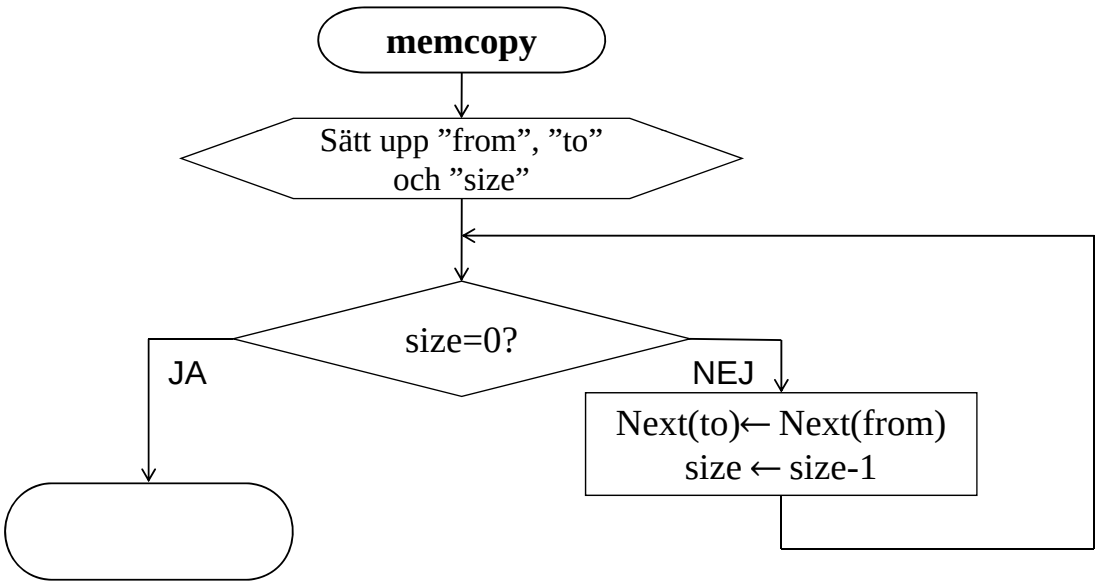
... Tar c:a 14 sekunder

Om längre fördröjningar krävs måste 'Delay'-funktionen utföra fler instruktioner i varje slinga, exempelvis genom anrop av följande rutin:

```
;
; subrutin 'ADelay'
ADelay:  BSR ADelay1
ADelay1: BSR ADelay2
ADelay2: RTS
```

Hur många bytes kod motsvarar rutinen?
Hur många klockcyklers fördröjning ger den?

EXEMPEL, subrutin ”memcopy(from , to, size)”



EXEMPEL – ”memcopy0(from , to, size)”

Kan (informellt) kodas...

```
memcopy0:           LDAB   ”size”
                    LDX    ”from”
                    LDY    ”to”

memcopy0_loop:      TSTB
                    BEQ    memcopy0_end
                    LDAA   1,X+
                    STAA   1,Y+
                    DECB
                    BRA    memcopy0_loop

memcopy0_end:       RTS
```

EXEMPEL – ”memcopy1(from , to, size)”

Kan (informellt) kodas...

```
memcopy1:           LDAB   ”size”
                    LDX    ”from”
                    LDY    ”to”

memcopy1_loop:      TSTB
                    BEQ    memcopy1_end
                    MOVB   1,X+,1,Y+
                    DECB
                    BRA    memcopy1_loop

memcopy1_end:       RTS
```


EXEMPEL – ”memcopy2(from , to, size)”

Kan (informellt) kodas...

```
memcopy2:           LDAB   ”size”
                    LDX    ”from”
                    LDY    ”to”

memcopy2_loop:      SUBB   #1
                    BMI    memcopy2_end
                    MOVB   B,X,B,Y
                    BRA    memcopy2_loop

memcopy2_end:       RTS
```

EXEMPEL – ”memcopy3(from , to, size)”

Kan (informellt) kodas...

```
memcopy3:           LDAB   ”size”
                    LDX    ”from”
                    LDY    ”to”

memcopy3_loop:      MOVB   1,X+,1,Y+
                    DBNE   B,memcopy3_loop

memcopy3_end:       RTS
```

vad händer om ”size” är 0, vid anrop???

EXEMPEL – ”memcopy(from , to, size)”

En effektiv implementering....

```
memcopy:            LDAB   ”size”
                    BEQ    memcopy_end
                    LDX    ”from”
                    LDY    ”to”

memcopy_loop:       MOVB   1,X+,1,Y+
                    DBNE   B,memcopy_loop

memcopy_end:        RTS
```