Assemblerprogrammering – del 2

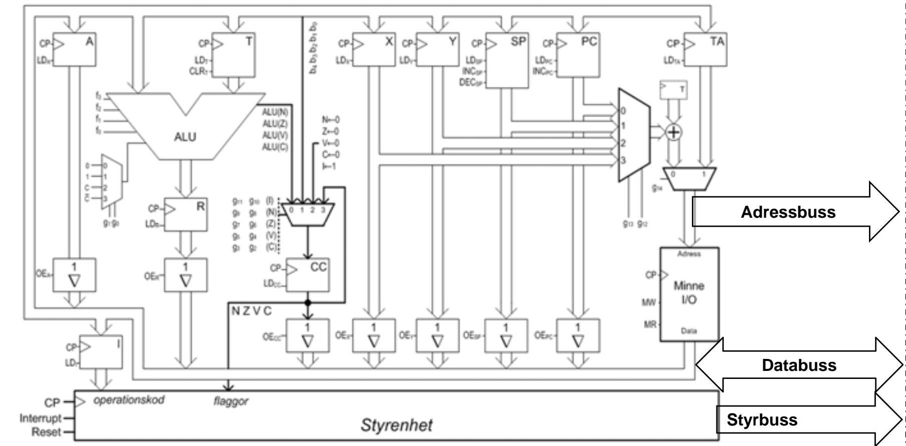
Dagens föreläsning behandlar:

Kompendiet kapitel 9
Arbetsboken kapitel 16

Ur innehållet:

- In- och ut-enheter
- Tilldelningar och uttrycksevaluering
- Programflödeskontroll

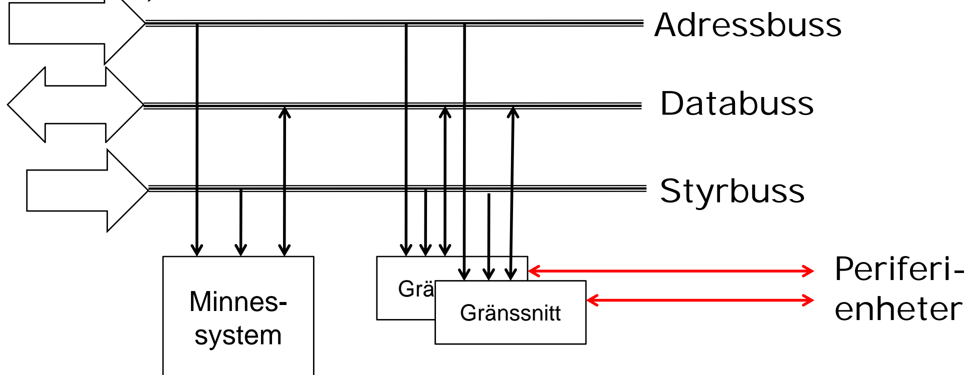
FLISP och omvärlden



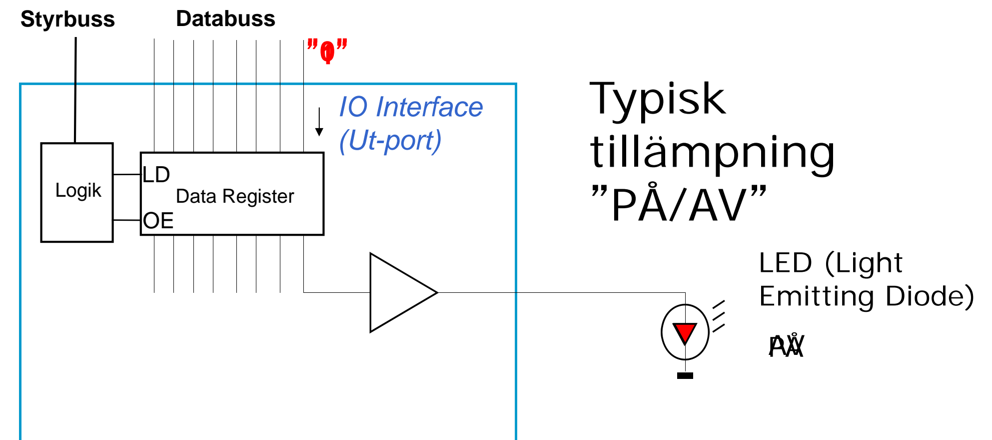
Periferienheter

Periferienheter

Enhet som ansluts till centralenhetens buss-system kallas ”periferienhet”. För varje periferienhet finns ett gränssnitt för in- och ut-matning (IO-interface)

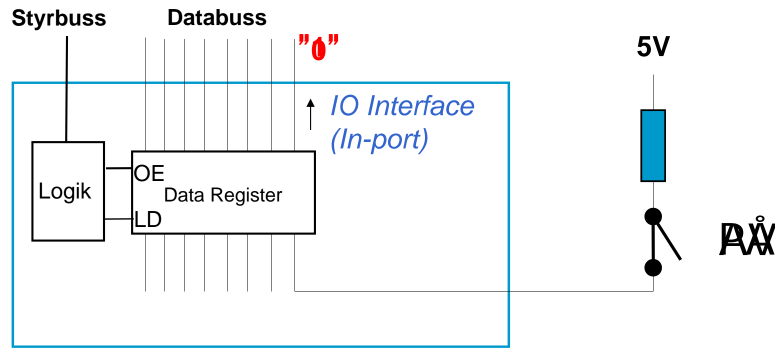


Parallell utmatning

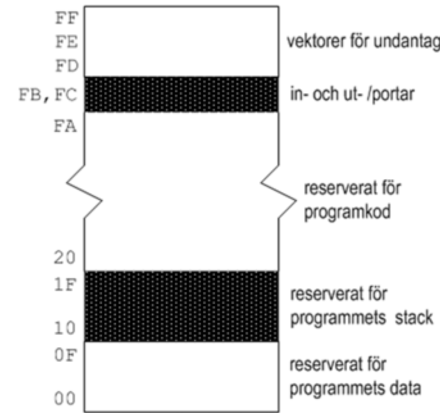


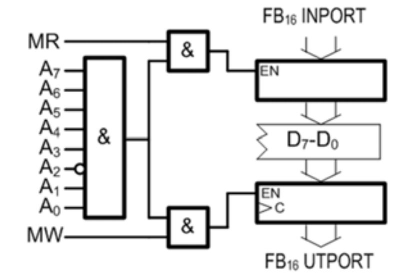
Parallell inmatning

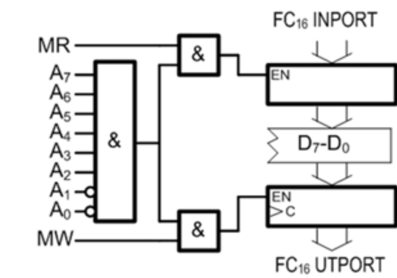
Typisk tillämpning:
Avläs ”PÅ/AV”



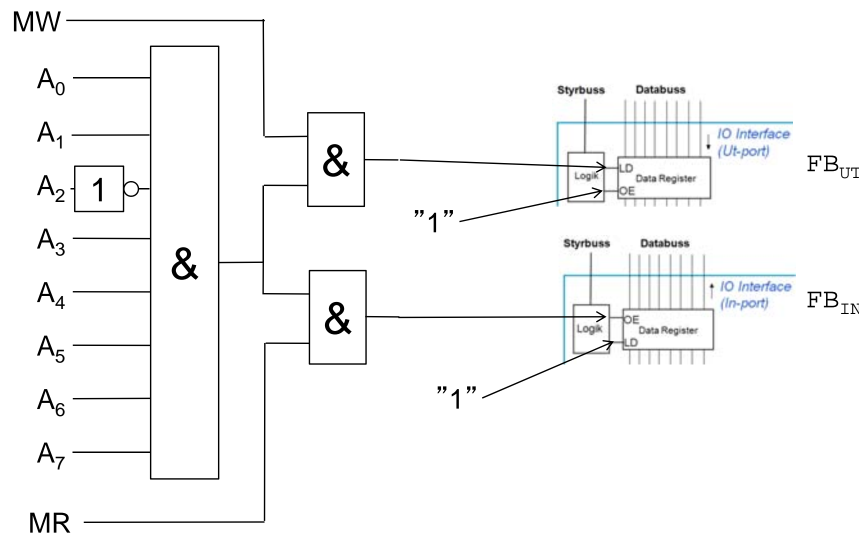
FLISP - Minnesdisposition



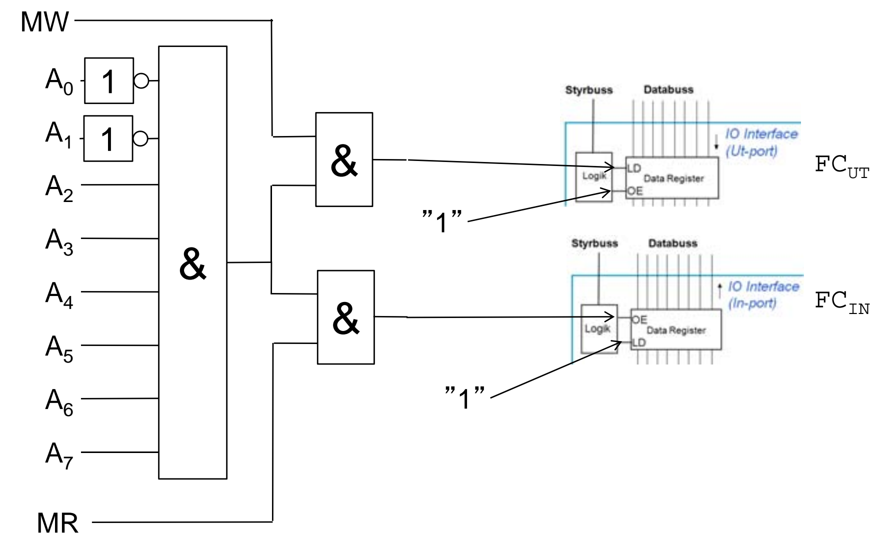




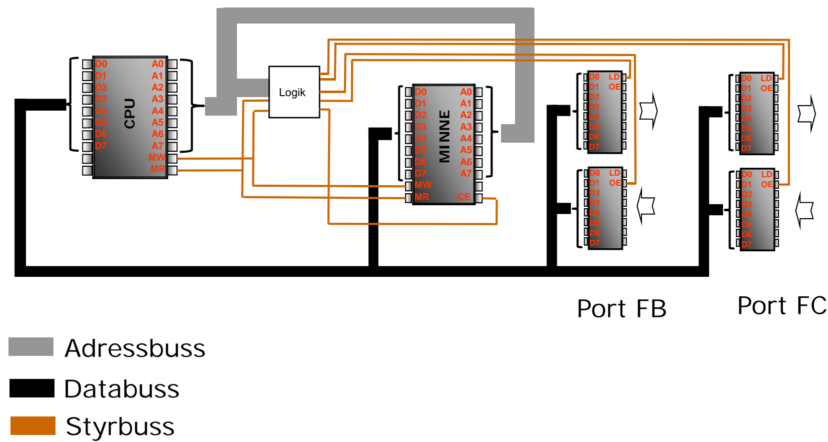
Port FB - Avkodningslogik



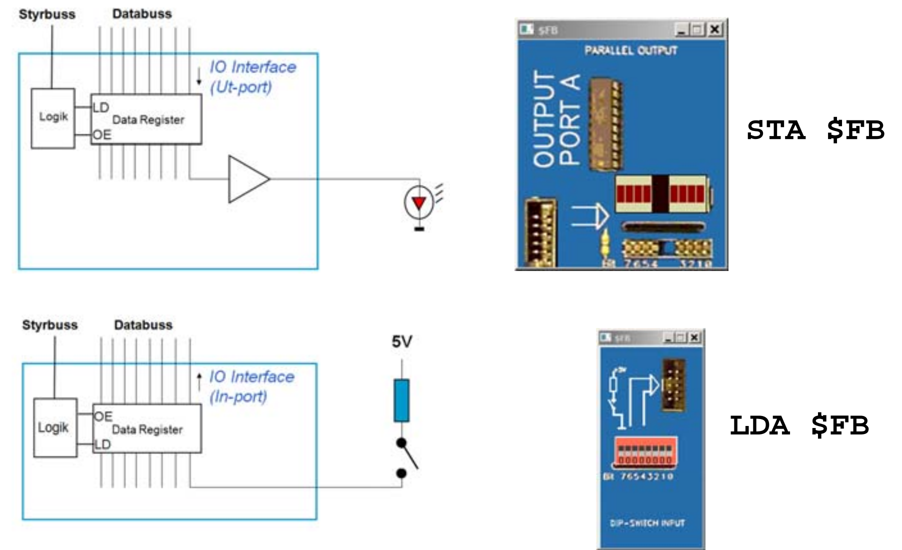
Port FC - Avkodningslogik



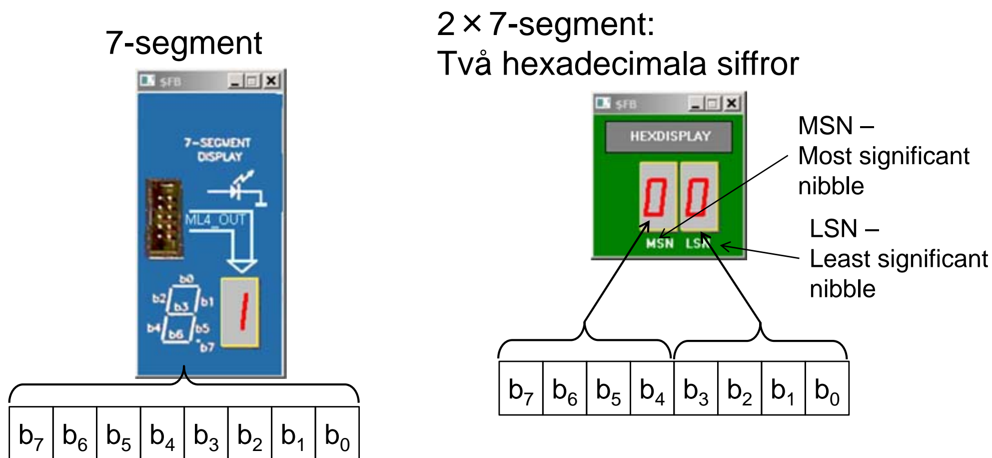
FLISP - Realisering



Periferikretsar



Utenheter - sifferindikatorer



EXEMPEL – ”Rinnande ljus”

Källtext

RunDiode.sflisp

```
;
;        RunDiode.sflisp
;
LED      EQU     $FB
         ORG     $20
RunDiode:
         ANDCC   #$FE            ; Nollställ Carry
         LDA     #1              ; Initialt mönster
RunDiode_1:
         STA     LED             ; Skriv till indikator
         ROLA                    ; Skifta mönster
         JMP     RunDiode_1      ; Repetera...
```

Listfil - skapas vid assembleringen

```
...r, Version RC:7

File: RunDiode.lst
                 1. ;
                 2. ;    RunDiode.sflisp
                 3. ;
FB               4. LED  EQU     $FB
20               5.      ORG     $20
20               6. RunDiode:
20 01 FE         7.      ANDCC   #$FE          ; Nollställ...
22 F0 01         8.      LDA     #1            ; Initialt ...
24               9. RunDiode_1:
24 E1 FB        10.      STA     LED           ; Skriv till ...
26 0D           11.      ROLA                  ; Skifta mönster
27 33 24        12.      JMP     RunDiode_1    ; Repetera...
                13.
29              14.
29              15.
```

Laddfil för simulator och laborationssystem – skapas vid assemblering

RunDiode.hwflisp

```
QAflisp - FLISP Absolute crossassemb
(c) GMV 1989-2013
File: RunDiode.hwflisp

 # ClearAllMemory
 # ClearAllRegisters
 #setMemory  20=01
 #setMemory  21=FE
 #setMemory  22=F0
 #setMemory  23=01
 #setMemory  24=E1
 #setMemory  25=FB
 #setMemory  26=0D
 #setMemory  27=33
 #setMemory  28=24
```

Datatyper – globala variabler

- En variabeldeklaration används för att reservera minnesutrymme.
- Deklarationen anger ett unikt symbolnamn med en distinkt ”datatyp”.
- **char** är en beteckning på en datatyp i programspråket ”C”.
- *Unsigned* respektive *signed* är så kallade *typspecifierare* som anger typens talområde.

Exempel 9.12 Enkla datadeklarationer, reservering av minnesutrymme

En variabeldeklaration:

```
unsigned char counter;
```

anger att en *byte* ska reserveras för symbolen counter, då den används i jämförelseoperationer ska den betraktas som ett 8-bitars tal utan inbyggt tecken. I assemblerspråk motsvarar deklarationen:

```
counter:  RMB  1
```

Deklarationen:

```
signed char summa;
```

anger också att en byte ska reserveras, denna gång för symbolen summa, i jämförelseoperationer ska symbolen betraktas som ett 8-bitars tal med inbyggt tecken och i assemblerspråk motsvarar deklarationen:

```
summa:    RMB  1
```

Observera att direktivet RMB inte innehåller någon typinformation om minnesinnehållet utan bara anger storleken på det reserverade utrymmet.

Tilldelningar – ”load/store”-instruktioner

Exempel 9.15 Tilldelningar av konstant till variabel

Antag att vi har följande globala variabel:

```
char index;
```

I assemblerspråk motsvaras deklarationen av:

```
index:    RMB   1
```

Vi vill göra följande tilldelning till denna variabel:

```
index = 1;
```

för detta exempel finns det fler alternativ:

```
LDA  #1
STA  index
```

eller:

```
LDX  #1
STX  index
```

eller:

```
LDY  #1
STY  index
```

hade fungerat lika bra.

Exempel 9.16 Tilldelning av variabel från variabel

Antag att vi har följande globala variabler:

```
char index;
char counter;
```

I assemblerspråk motsvaras deklarationerna av:

```
index:    RMB  1
counter:  RMB  1
```

Exempelvis kan man nu koda tilldelningen:

```
counter = index;
```

som:

```
LDA  index
STA  counter
```


Vektorer

```
char  char_array[4];
char  init_array[4] = {1,2,3,4};
```

Adress	Innehåll	Assemblerkod (disassemblering)		
			ORG	$20
20		char_array:	RMB	4
21				
22				
23				
24	1	init_array	FCB	1
25	2		FCB	2
26	3		FCB	3
27	4		FCB	4
28				

EXEMPEL: Referens av vektorer, konstant index

Antag variabeldeklarationerna:

```
char char_array[4];
char item;
```

Vi vill göra tilldelningen:

```
item = char_array[2];
```

dvs. kopiera element med ordningsnummer 3 i vektorn till variabeln item.

```
            ORG   0
char_array  RMB   4
item        RMB   1
            ORG   $20
            LDX   #char_array
            LDA   2,X
            STA   item
```

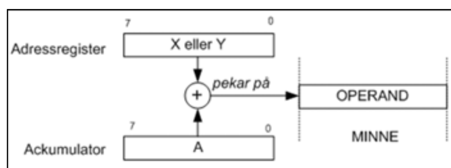
Den omvända tilldelningen är lika enkel, om vi vill göra tilldelningen:

```
char_array[2] = item;
```

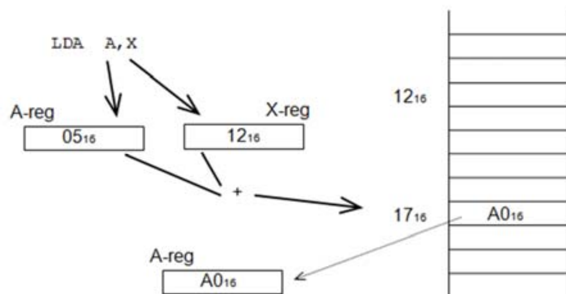
dvs. kopiera variabeln item till element 3 i vektorn, blir detta.

```
            ORG   $20
            LDX   #char_array
            LDA   item
            STA   2,X
```

Referens av vektorer, evaluerat index



Ackumulatorindexerad adressering sker ex vis med FLISP-instruktionen `LDA A,X`. Den läser data på adressen X + A och placerar det i ackumulator A.



EXEMPEL: ”Tabelluppslagning”

```
char   ascii_table[]={'0','1','2','3','4','5','6','7','8','9'};

ascii_table:  FCB    '0','1','2','3','4','5','6','7','8','9'
```

- Låt oss nu anta register A innehåller ett binärt värde, 0 t.o.m 9.
- Vi vill ersätta värdet i A med dess motsvarande ASCII-värde.
- Det kan göras med följande sekvens:

```
LDX    #ascii_table        ; startadressen till tabellen laddas till X
LDA    A,X                 ; värdet i A adderas till X,
                           ; bildar ny adress till ett tecken i tabellen
                           ; detta tecken laddas till A
```


Uttrycksevaluering

- Ett uttryck sätts samman av konstanter, variabler och operatorer.
- Resultatet av ett uttryck är alltid ett värde (evaluerat uttryck) som antingen kan användas för att tilldelas någon variabel eller användas som test för någon villkorlig operation.
- Exempel på FLISP-instruktioner för uttrycksevaluering:
  `ADDA, SUBA, ANDA, ORA, EORA, COMA, NEGA`

Exempel:

```
char  sum,op1,op2,op3;
```

motsvarande assemblerdeklarationer blir:

```
      ORG   0
sum   RMB   1
op1   RMB   1
op2   RMB   1
op3   RMB   1
```

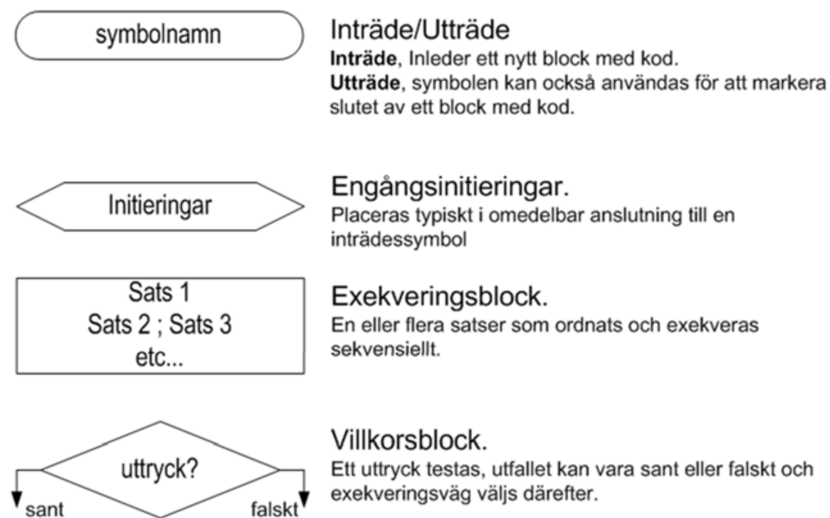
Tilldelningen:

```
      sum = op1+op2-op3;
```

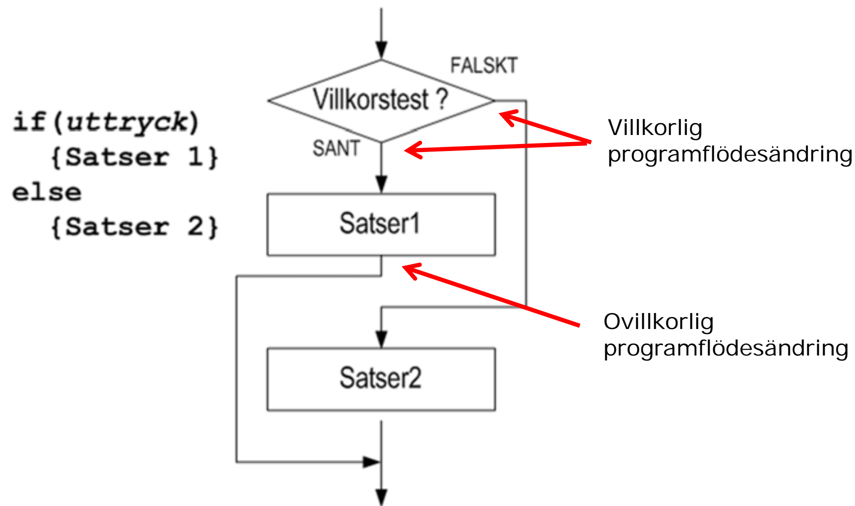
kan då kodas:

```
      LDA   op1
      ADDA  op2
      SUBA  op3
      STA   sum
```


Symboler i flödesgrafen



Programflödeskontroll



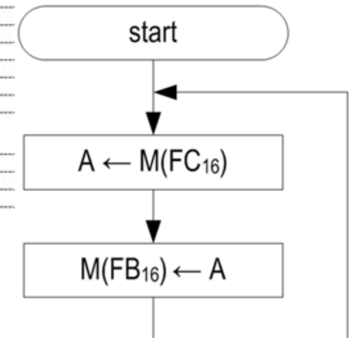
EXEMPEL: Ovillkorlig programflödesändring

Följande programexempel som vi såg inledningsvis använder den ovillkorliga programflödesinstruktionen JMP.

Symbolfält	Mnemonic eller direktiv	Operand	Kommentarsfält
	ORG	$20	; Program börjar
InPort	EQU	$FC	
UtPort	EQU	$FB	
start:	LDA	InPort	; Läs data
	STA	UtPort	; Skriv data
	JMP	**start**	; Repetera...

Vi hade också kunnat använda BRA-instruktionen:

start:	LDA	InPort	; Läs data
	STA	UtPort	; Skriv data
	BRA	**start**	; Repetera...



Instruktioner för villkorlig programflödesändring

Mnemonic	Funktion	Villkor
	Enkla flaggtest	
BCS	"Hopp" om *carry*	C=1
BCC	"Hopp" om ICKE *carry*	C=0
BEQ	"Hopp" om *zero*	Z=1
BNE	"Hopp" om ICKE *zero*	Z=0
BMI	"Hopp" om *negative*	N=1
BPL	"Hopp" om ICKE *negative*	N=0
BVS	"Hopp" om *overflow*	V=1
BVC	"Hopp" om ICKE *overflow*	V=0
	Test av tal utan tecken	
BHI	Villkor: R>M	C + Z = 0
BHS	Villkor: R≥M	C=0
BLO	Villkor: R<M	C=1
BLS	Villkor: R≤M	C + Z = 1
	Test av tal med tecken	
BGT	Villkor: R>M	Z + (N ⊕ V) = 0
BGE	Villkor: R≥M	N ⊕ V = 0
BLT	Villkor: R<M	N ⊕ V = 1
BLE	Villkor: R≤M	Z + (N ⊕ V) = 1

EXEMPEL: Villkorlig programflödesändring

Mnemonic	Funktion
	Enkla flaggtest
BCS	"Hopp" om *carry*
BCC	"Hopp" om ICKE *carry*
BEQ	"Hopp" om *zero*
BNE	"Hopp" om ICKE *zero*
BMI	"Hopp" om *negative*
BPL	"Hopp" om ICKE *negative*
BVS	"Hopp" om *overflow*
BVC	"Hopp" om ICKE *overflow*
	Test av tal utan tecken
BHI	Villkor: R>M
BHS	Villkor: R≥M
BLO	Villkor: R<M
BLS	Villkor: R≤M
	Test av tal med tecken
BGT	Villkor: R>M
BGE	Villkor: R≥M
BLT	Villkor: R<M
BLE	Villkor: R≤M

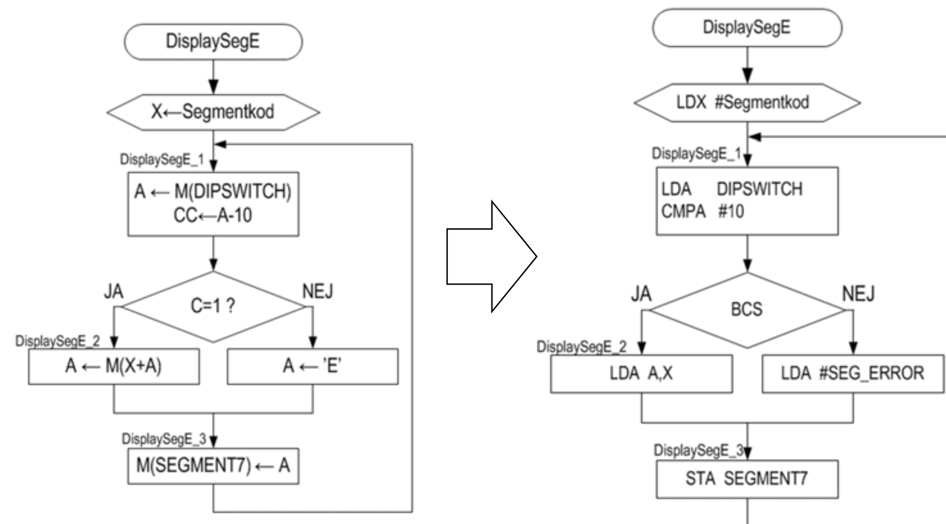
```
signed    char sindex;
unsigned  char uindex;
...
   if( uindex < 10 )
      { satser }
kodas:
   LDA  uindex
   CMPA #10
   BLO  satser
medan
...
   if( sindex < 10 )
      { satser }
ska kodas:
   LDA  sindex
   CMPA #10
   BLT  satser
```

Komplementära villkor

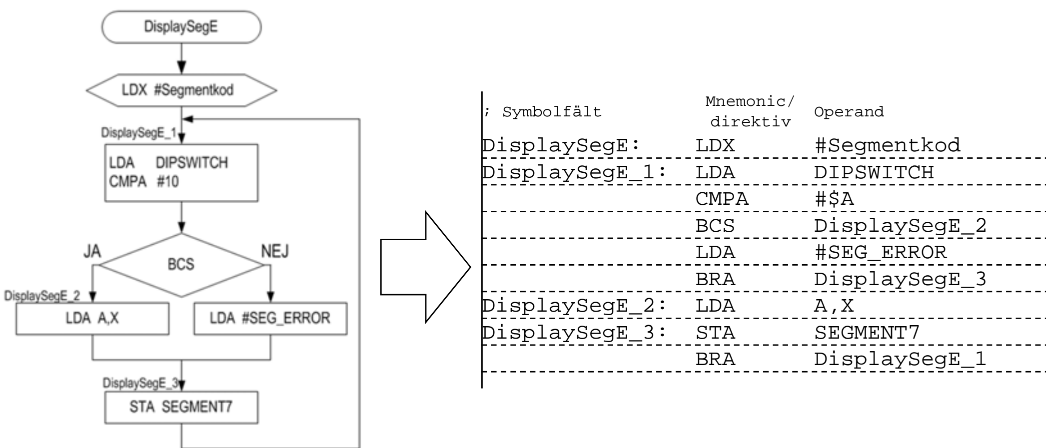
```
         CMPA  #10                           CMPA  #10
         BCC   FALSKT                        BCS   SANT
SANT:    ..                        FALSKT:   ..
         BRA   SLUT                          BRA   SLUT
FALSKT:  ..                        SANT:     ..

SLUT:                              SLUT:
```


EXEMPEL: arbetsbokens uppgift 16.9



”Linjärisera” koden…



; Symbolfält	Mnemonic/ direktiv	Operand
DisplaySegE:	LDX	#Segmentkod
DisplaySegE_1:	LDA	DIPSWITCH
	CMPA	#$A
	BCS	DisplaySegE_2
	LDA	#SEG_ERROR
	BRA	DisplaySegE_3
DisplaySegE_2:	LDA	A,X
DisplaySegE_3:	STA	SEGMENT7
	BRA	DisplaySegE_1