EDA216

Digital- och datorteknik

Ext-1 – Ext-17

(Ext-16 saknas)

VT14

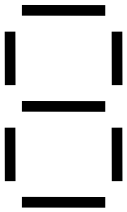
Kompletterande material Ext-1 - Ext-17 (pdf):

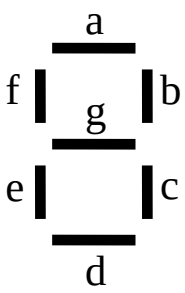
- Ext-1 Introduktionsexempel.
- Ext-2 Omvandling mellan talsystem.
- Ext-3 Mintermer. Disjunktiv (SP) normal form. Maxtermer. Konjunktiv (PS) normal form.
- Ext-4 Praktikfall, minimering av grindnät.
- Ext-5 NAND- och NOR-logik. Exempel.
- Ext-6 Förenklad förklaring i anslutning till kapitel 6, avsnitt 6.3 och 6.4.
- Ext-7 Synkrona sekvensnät. Analys och syntes.
- Ext-8 Datorn enligt von Neumann.
- Ext-9 Instruktioner för villkorliga hopp med PC-relativ adressering.
- Ext-10 Exempel på stackanvändning med FLIS-processorn.
- Ext-11 Extra programmeringsuppgifter för FLIS-processorn.
- Ext-12 Flödesplan. Program för multiplikation genom upprepad addition.
- Ext-13 RTN-beskrivning av FLEX-instruktioner.
- Ext-14 FLEX-processorns styrenhet med fast kopplad logik.
- Ext-15 Assemblator för FLIS-processorn.
- Ext-17 Kapitel 13 ur "Grundläggande digital- och datorteknik, del 2".

Vad är digitalteknik för något?

Digitalteknik = **Siffer**teknik

Sifferindikator

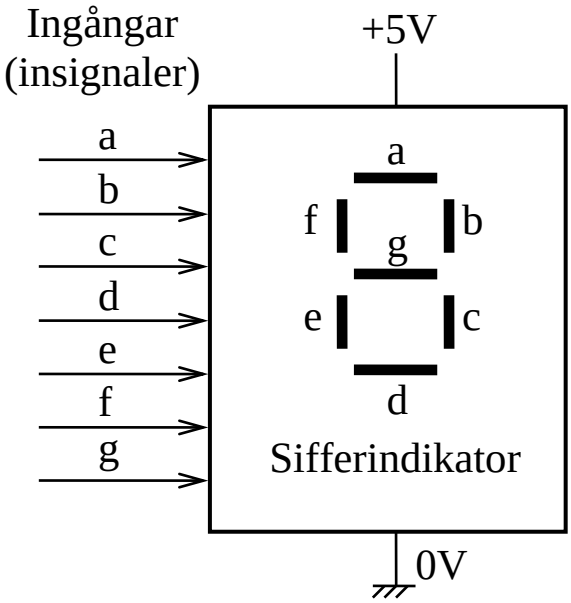


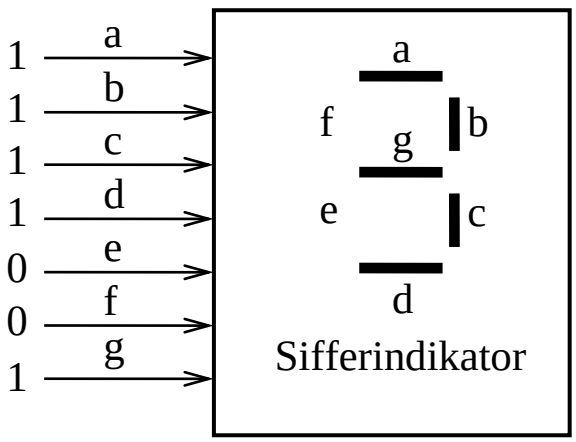


Sifferindikatorn har sju segment. Man kan utifrån välja vilka segment som skall synas och vilka som skall vara osynliga.

För denna sifferindikator antas gälla att när en "etta" läggs på en av dess ingångar t ex a, så syns a-segmentet i sifferfönstret. En "nolla" på en ingång medför att motsvarande segment är osynligt.

Sifferindikatorn som visas här har matningsspänningen 5V. Det normala är då att en "etta" motsvaras av spänningen +5V relativt jord och en nolla av 0V.



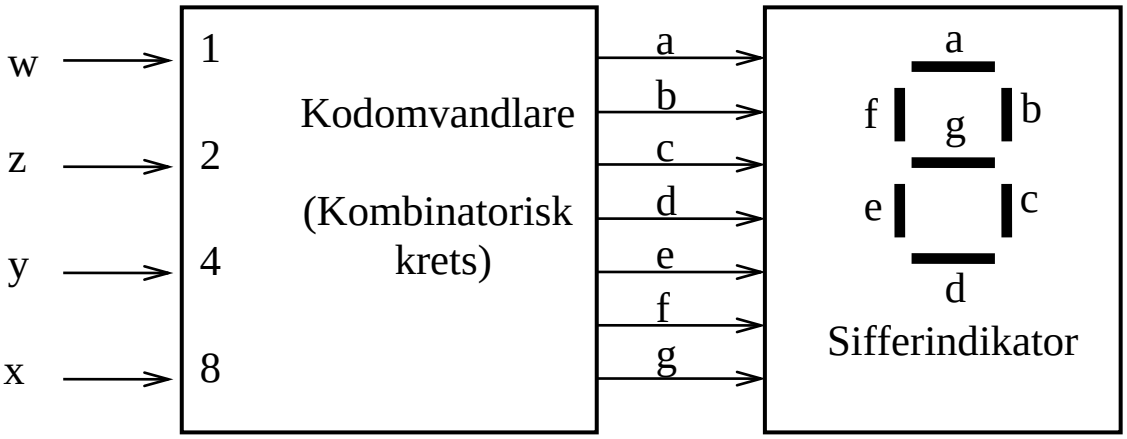


Ett **digitalt** system arbetar med **siffror**.

I princip kan man arbeta med siffror i vilket talsystem som helst.

Eftersom det är enklast att konstruera komponenter som kan växla mellan två tillstånd (som man kan beteckna "0" och "1") har det binära talsystemet, som har siffersymbolerna 0 och 1, blivit helt dominerande inom digitaltekniken.

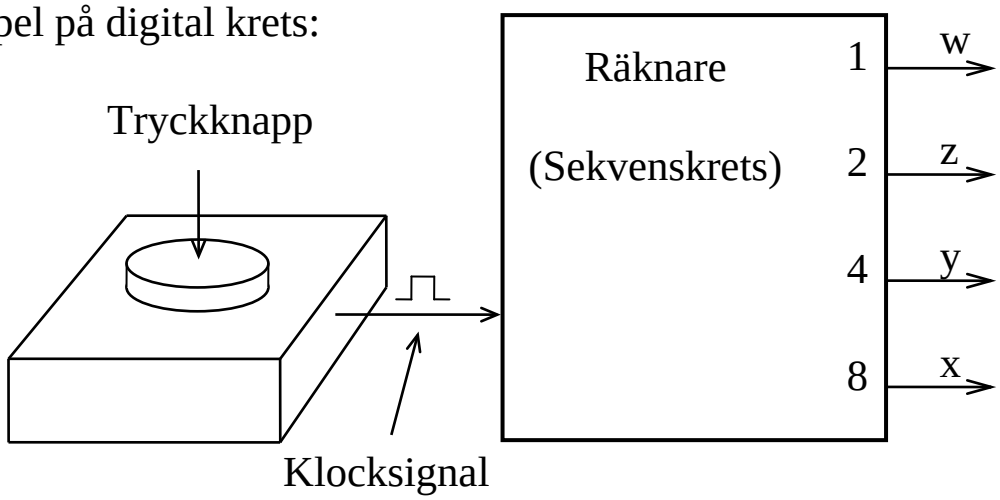
Exempel på digital krets:



Kodomvandlaren till vänster i figuren ovan är en digital krets. Den omvandlar ett binärt tal med siffrorna xyzw (ingångarna) till ett mönster av nollor och ettor på utgångarna abcdefg, som kopplas till motsvarande ingångar på sifferindikatorn, så att den visar det binära talet $(xyzw)_2$ på decimal form.

I kursen visas metoder för att studera (analys) och konstruera (syntes) digitala kretsar liknande kodomvandlaren.

Annat exempel på digital krets:



Varje gång man trycker ned tryckknappen kommer det en puls (klocksignal) på ledningen som är kopplad till räknaren.

För varje tryckning (klockpuls) ökar räknaren värdet med ett (1) på det fyrsiffriga binära talet $(xyzw)_2$ som finns på utgångarna x, y, z och w.

Om räknaren startar på värdet noll $(0000)_2$ kommer den alltså att visa värdet 5 $(0101)_2$ efter 5 klockpulser.

Räknaren håller alltså reda på hur många klockpulser den tagit emot. Detta är liktydigt med att räknaren har "minne".

Denna typ av digital krets är alltså beroende av vilken sekvens av insignaler som har påverkat kretsen tidigare och kallas därför **sekvenskrets**.

I kursen studeras också metoder för att studera (analys) och konstruera (syntes) digitala kretsar liknande räknaren ovan, dvs sekvenskretsar.

Utsignalvärdet för kodomvandlaren som visades först var bara beroende av det aktuella insignalvärdet. Kodomvandlaren kallas **kombinatorisk krets**.

Digitaltekniken breder ut sig i alla riktningar inom tekniken och är av central betydelse för modern mätning och styrning.

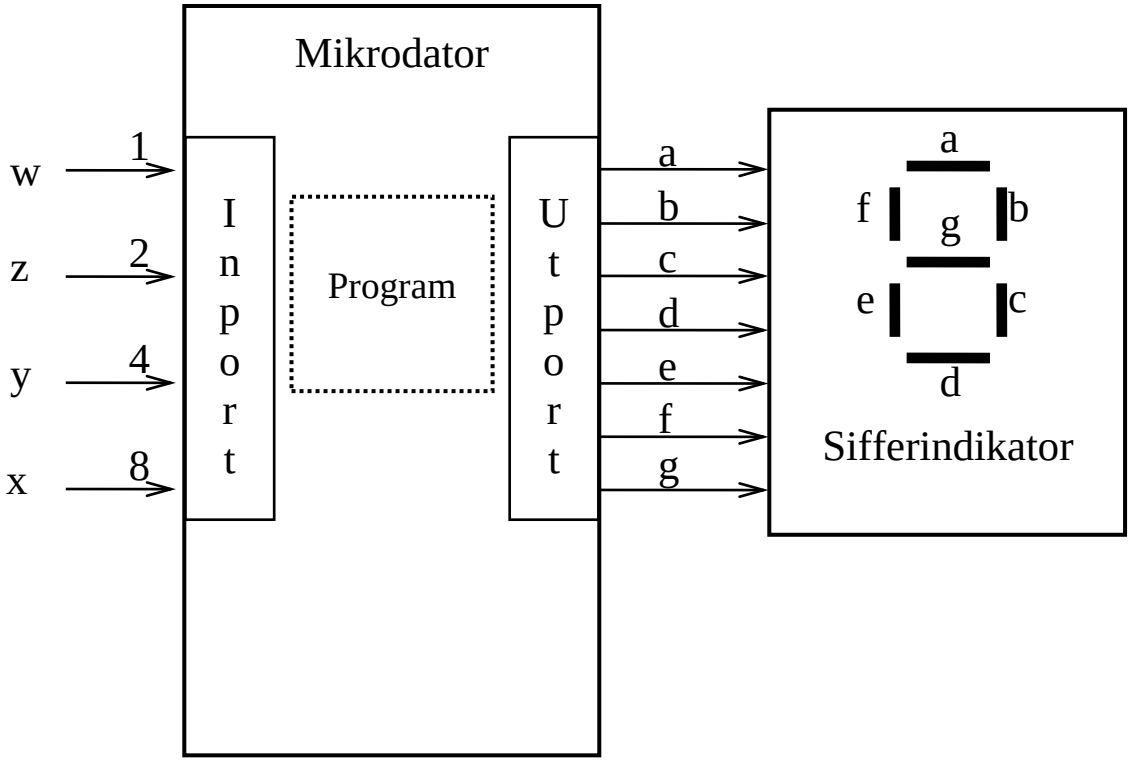
Kursen ger en god uppfattning om grundläggande digitalteknik och hur den används.

DATORTEKNIK

Datorn används idag som arbetsredskap och generellt hjälpmedel av de flesta människor. Idag och under överskådlig framtid är datorn ingenjörens viktigaste hjälpmedel.

De flesta datorer, i form av mikrodatorer, är idag inbyggda i olika konsumentprodukter och medför att dessa kan utföra avancerade uppgifter till en mycket låg kostnad.

Ett mycket enkelt exempel på hur en mikrodator kan användas, istället för kodomvandlaren för sifferindikatorn som beskrevs tidigare, visas nedan.



Kursen förklarar datorns konstruktion på ett enkelt sätt utgående från digitaltekniska komponenter.

Under föreläsningar, simulatorövningar och laborationer tar vi fram och sätter samman datorns byggstenar till en enkel fungerande dator.

Vi skriver sedan program för vår dator i maskinspråk och assemblerspråk.

Omvandling av ett tal N med basen 10 till basen 2.

$$N_{10} \rightarrow N_2$$

Talet N_{10} delas upp i heltalsdel $N_{10\,H}$ och bråktalsdel $N_{10\,B}$.

$$N_{10} = (N_H, N_B)_{10}$$

Heltalsdelen och bråktalsdelen behandlas sedan var för sig.

Heltalsdelen:

Heltalsdelen skall skrivas med siffror i basen 2 som

$$N_{2\,H} = d_{n-1}\, d_{n-2} \ldots d_0$$

Man tänker sig heltalsdelen av talet $N_{10\,H}$ skrivet med nya basen 2.

$$N_{10\,H} = d_{n-1}\cdot 2^{n-1} + d_{n-2}\cdot 2^{n-2} + \ldots + d_1\cdot 2^1 + d_0\cdot 2^0$$

Dividera talet med basen 2.

$$N_{10\,H} / 2 = \underbrace{d_{n-1}\cdot 2^{n-2} + d_{n-2}\cdot 2^{n-3} + \ldots + d_1\cdot 2^0}_{\text{heltal}} + \underbrace{d_0\cdot 2^{-1}}_{\text{bråktal}}$$

Efter divisionen "dyker" d_0 upp som bråktalsdel (rest).

Fortsatt division av det nya heltalet med basen 2 ger siffrorna d_1, d_2, d_3, ... i talet, fast nu i basen 2.

Bråktalsdelen:

Bråktalsdelen skall skrivas med siffror i basen 2 som

$$N_{2\,B} = .d_{-1}\, d_{-2} \ldots d_{-m}$$

Man tänker sig bråktalsdelen av talet $N_{10\,B}$ skrivet i den nya basen 2.

$$N_{10\,B} = d_{-1}\cdot 2^{-1} + d_{-2}\cdot 2^{-2} + \ldots + d_{-m}\cdot 2^{-m}$$

Multiplicera talet med basen 2.

$$2\,N_{10\,B} = \underbrace{d_{-1}\cdot 2^{0}}_{\text{heltal}} + \underbrace{d_{-2}\cdot 2^{-1} + \ldots + d_{-m}\cdot 2^{-m+1}}_{\text{bråktal}}$$

Efter multiplikationen "dyker" d_{-1} upp som heltalsdel.

Fortsatt multiplikation av den nya bråktalsdelen med basen 2 ger siffrorna d_{-2}, d_{-3}, ... i talet, fast nu i basen 2.

Till sist sätts talet N_2 samman av heltalsdelen och bråktalsdelen.

$$N_2 = (N_H.N_B)_2 = (d_{n-1}\, d_{n-2} \ldots d_0.d_{-1}\, d_{-2} \ldots d_{-m})_2$$

3.5 Mintermer och maxtermer

3.6 Disjunktiv (SP) normal form och konjunktiv (PS) normal form

Vi börjar med ett enkelt exempel:

f(x,y,z) = x y + y z + x'z

Rita funktionstabell (3 variabler => 2^3 = 8 rader)

Princip för tabellen:

x	y	z	f
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Funktionsvärden ifyllda:

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Man konstaterar att invariabelkombinationen (x, y, z) i varje rad i funktionstabellen är unik.

För varje rad i funktionstabellen kan man därför ta fram en unik produkt av invariabler som ger 1 · 1 · 1 = 1. En sådan produkt har alltså värdet 1 endast för den kombination av insignalvärden som motsvarar den aktuella raden. För alla andra insignalkombinationer har produkten värdet 0.

Produkten för andra raden blir t ex: x'· y'· z

Om man väljer att ta fram produkter för samtliga rader där funktionen har värdet 1 och adderar dessa produkter blir summan ett uttryck för den booleska funktionen f. En summa av produkter kallas också SP-form.

f = x'y'z + x'y z + x y z' + x y z

Detta uttryck är speciellt på så sätt att varje term är unik och innehåller samtliga invariabler. Termerna kallas mintermer och funktionen sägs vara skriven på disjunktiv normal form eller SP normal form.

Eftersom det finns en minterm för varje rad i funktionstabellen finns det sammanlagt 2^3 = 8 mintermer för en funktion av tre variabler. Se tabell 3.8 i kompendiet!

För varje rad i funktionstabellen kan man på samma sätt ta fram en unik summa av invariabler som ger 0 + 0 + 0 = 0. En sådan summa har alltså värdet 0 endast för den kombination av insignalvärden som motsvarar den aktuella raden. För alla andra insignalkombinationer har summan värdet 1.

Summan för första raden blir tex (x + y + z)

Om vi väljer att ta fram sådana summor för samtliga rader där funktionen har värdet 0 och multiplicerar dessa summor blir produkten ett uttryck för den Booleska funktionen f. En produkt av summor kallas också PS-form.

f = (x + y + z)(x + y'+ z)(x'+ y + z)(x'+ y +z')

Detta uttryck är speciellt på så sätt att varje faktor är unik och innehåller samtliga invariabler. Faktorerna kallas maxtermer och funktionen sägs vara skriven på konjunktiv normal form eller PS normal form.

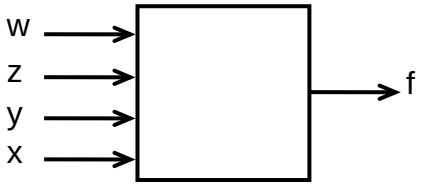
Eftersom det finns en maxterm för varje rad i funktionstabellen finns det sammanlagt $2^3 = 8$ maxtermer för en funktion av tre variabler. Se tabell 3.9 i kompendiet!

Man ser ibland uttrycket kanonisk form i stället för normal form.

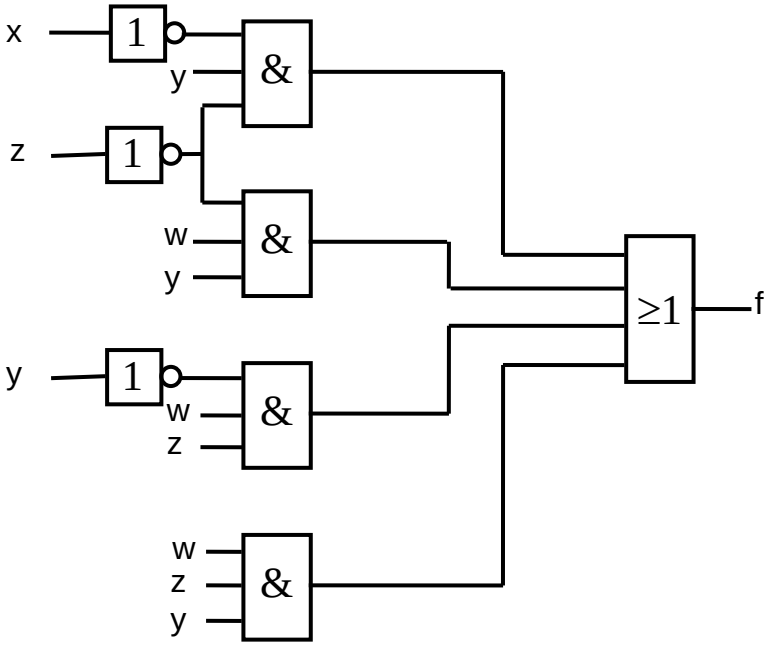
Ext-4 (Ver 2012-10-31)

Praktikfall, minimering av grindnät

Ett grindnät med utsignalen f och de fyra insignalerna x, y, z och w är givet.



Grindnätet är uppbyggt med OCH-, ELLER- och INVERTERAR-grindar enligt figuren nedan.



Kan man konstruera ett "mindre" nät med samma typer av grindar, som realiserar funktionen f?
Med ett "mindre" nät menar vi här att antalet grindar eller ingångar är mindre.

Ext-4 (Ver 2012-10-31)

x	y	z	w	f
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Realisering:

Minimering:

f	zw			
xy	00	01	11	10
00				
01				
11				
10				

f =

Ext-5 (Ver 2012-10-31)

NAND- och NOR-logik

En tank är försedd med en givare för vätskenivån. Nivågivaren är ansluten till en A/D-omvandlare som omvandlar givarvärdet till ett fyrabitars tal X i intervallet 0000_2 - 1111_2 = 0 - 15.

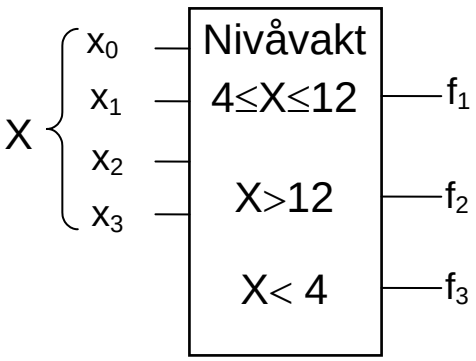
För att man skall kunna styra tillflödet till tanken på lämpligt sätt behöver man en nivåvakt med tre utsignaler f_1, f_2 och f_3.

f_1 skall lämna ut värdet 1 ifall $4 \leq X \leq 12$. För alla andra värden på X skall den lämna ut värdet 0.

f_2 skall lämna ut värdet 1 ifall $X > 12$. För alla andra värden på X skall den lämna ut värdet 0.

f_3 skall lämna ut värdet 1 ifall $X \leq 3$. För alla andra värden på X skall den lämna ut värdet 0.

x_0 är minst signifikant och x_3 är mest signifikant siffra.



a) Konstruera ett minimalt grindnät för nivåvakten med hjälp av NAND-grindar och INVERTERARE.

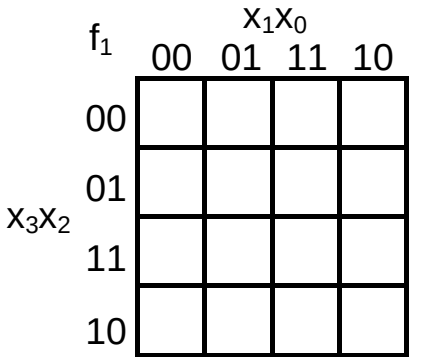
b) Konstruera ett minimalt grindnät för nivåvakten med hjälp av NOR-grindar och INVERTERARE.

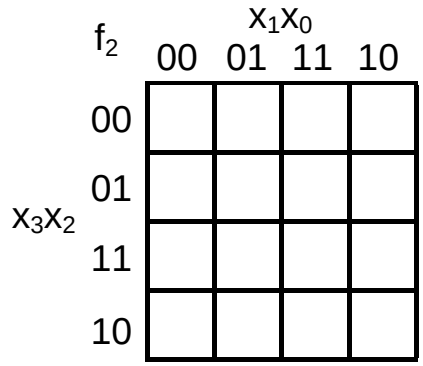
Ext-5 (Ver 2012-10-31)

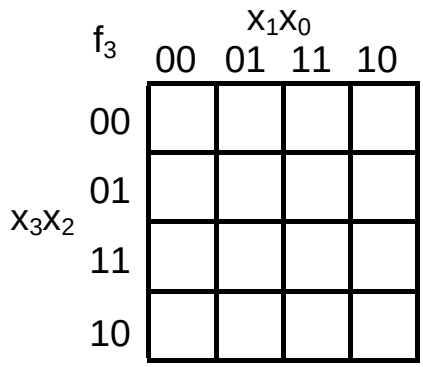
Funktionstabell:

x_3	x_2	x_1	x_0	f_3	f_2	f_1
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

Minimering:







Realisering:

Förenklad förklaring i anslutning till kompedieavsnitten 6.3 och 6.4

Tecken-beloppsrepresentation av heltal

Hur skall man kunna räkna med negativa tal i ett digitalt system , t ex en dator.

Det enda man kan arbeta med är ju binära siffror, dvs 0 och 1.

En enkel metod vore ju att avdela en siffra för tecknet, t ex 0 för + (plus)
och 1 för - (minus) och använda resten av siffrorna för talets storlek (absolutbelopp).

Med 8 bitars ordlängd kan man då skriva t ex talet $+14 = +1110_2$ som

Teckenbit ⟶ 0 0001110 ⟵ Belopp

och talet -19 som

Teckenbit ⟶ 1 0010011 ⟵ Belopp

För 8-bitars tal blir talområdet $[-(2^{8-1}-1),+(2^{8-1}-1)] = [-127,+127]$.

Teckenbyte görs genom invertering av teckenbiten.

Tecken-beloppsrepresentation blir dock mycket besvärlig att använda i praktiken när man skall addera eller subtrahera. Se följande exempel.

Addition ↓

Exempel: (+7) + (+5) = + (7+5) = +12

(+7) + (-5) = + (7-5) = +2

(-7) + (+5) = - (7-5) = -2 ⟵ Subtraktion

(-7) + (-5) = - (7+5) = -12

Det som från början såg ut som en addition kan senare visa sig bli en subtraktion.
Dessutom måste man hålla reda på och bilda rätt tecken.

2-komplementrepresentation av heltal

En enkel lösning på problemet tal med tecken får man med 2-komplementrepresentation.

Denna innebär att man behåller positiva tal som de är. Negativa tal byts ut mot 2-komplementet (egentligen 2^n-komplementet, där n är antalet bitar i talet) av motsvarande positiva tal.

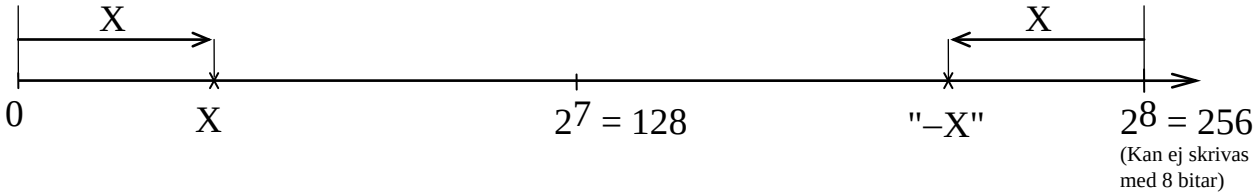
Exempel.
För 8-bitars tal gäller:

+5 används som det är dvs $(00000101)_2$

–5 byts mot $2^8 - 5 = 256 - 5 = (100000000)_2 - (00000101)_2 = (11111011)_2$

–38 byts mot $2^8 - 38 = 256 - 38 = (100000000)_2 - (00100110)_2 = (11011010)_2$

Princip: Om $X \geq 0$ används X självt för att representera talet.
Om $X < 0$ används $2^8 - |X|$ istället för X för att representera talet.



Ett åttabitars heltal kan anta ett värde i intervallet $[0,2^8-1] = [0,255]$. I figuren ovan har detta intervall visats på tallinjen.

I figuren är också talen X och motsvarande "negativa" tal "–X" utsatta.

Man ser att så länge $X < 256/2 = 128$, så kan man skilja X och "–X" åt genom att X finns i den vänstra halvan av det visade intervallet, medan "–X" finns till höger. Man begränsar därför talområdet för positiva tal till den vänstra halvan av intervallet. På detta sätt får varje positivt tal på den vänstra halvan en "negativ" motsvarighet på den högra halvan.

Samtliga talvärden X på den vänstra halvan (dvs "positiva tal") har den mest signifikanta biten $x_7 = 0$. På motsvarande sätt har samtliga talvärden X på den högra halvan (dvs "negativa tal") den mest signifikanta biten $x_7 = 1$. Biten längst till vänster kallas därför **teckenbit**.

För 8-bitars tal blir talområdet $[-2^{8-1},+(2^{8-1}-1)] = [-128,+127]$.
Talet 0 betraktas som positivt. Talet –128 har ingen positiv motsvarighet.

Teckenbyte görs genom 2-komplementering, dvs $X_{2k} = 2^8 - X$
(Fungerar ej för X = –128!)

$(X_{2k})_{2k} = 2^8 - (2^8 - X) = X$ (Teckenbyte två gånger ger det ursprungliga talet.)

Den stora fördelen med 2-komplementrepresentation är att man kan använda vanlig binär addition och subtraktion.

Exempel på addition med 8-bitars ordlängd:

Istället för –5 används 256 – 5 = 251 = 11111011_2

Istället för –7 används 256 – 7 = 249 = 11111001_2

(+7) + (+5) Utförs som: 7 + 5 = 12

```
           00001110 carry
Binärt:    00000111
         + 00000101
           00001100 = 12
```

(+7) + (–5) Utförs som: 7 + (256 – 5) = 256 + 2

```
           11111110 carry
Binärt:    00000111
         + 11111011
          100000010 = 256+2 (motsvarar 2 om minnes-
                             siffran ut stryks)
```

(–7) + (+5) Utförs som: (256 – 7) + 5 = 256 – 2

```
           000000010 carry
Binärt:    11111001
         + 00000101
           11111110 = 256-2 (motsvarar -2)
```

(–7) + (–5) Utförs som: (256 – 7) + (256 – 5) = 256 + 256 – 12

```
           11111110 carry
Binärt:    11111001
         + 11111011
          111110100 = 256 + 256-12 (motsvarar -12 om minnes-
                                    siffran ut stryks)
```

Subtraktion behandlas på motsvarande sätt. Operationen X – Y ersätts standardmässigt med operationen X + 256 – Y = X + (255 – Y) + 1 = X + Y_{1k} + 1.

Exempel på subtraktion med 8-bitars ordlängd:

Istället för –5 används 256 – 5 = 251 = 11111011_2

Istället för –7 används 256 – 7 = 249 = 11111001_2

5_{1k} = 255 – 5 = 250 = 11111010_2

251_{1k} = 255 – 251 = 4 = 00000100_2

(+7) – (+5) Utförs som: 7 + 5_{1k} +1 = 7 + 250 +1 = 256 + 2

```
          11111111 carry
Binärt:    00000111
         + 11111010
          100000010 = 256 + 2 (motsvarar 2 om minnes-
                               siffran ut stryks)
```

(+7) – (–5) Utförs som: 7 + $(256 – 5)_{1k}$ + 1 = 7 + 251_{1k} + 1 = 7 + 4 + 1 = 12

```
          000001111 carry
Binärt:    00000111
         + 00000100
           00001100 = 12
```

(–7) – (+5) Utförs som: (256 – 7) + 5_{1k} + 1 = 256 – 7 + 250 + 1 = 256 + 256 – 12

```
          11111110 carry
Binärt:    11111001
         + 11111010
          111110100 = 256 + 256-12 (motsvarar -12 om minnes-
                                    siffran ut stryks)
```

(–7) – (–5) Utförs som: (256 – 7) + $(256 – 5)_{1k}$ + 1 = 256 – 7 + 251_{1k} + 1 =
= 256 – 7 + 4 +1 = 256 – 2

```
          000000011 carry
Binärt:    11111001
         + 00000100
           11111110 = 256 - 2 (motsvarar -2)
```

Sammanfattning av addition och subtraktion med 2-komplementaritmetik

Vi konstaterar att vanlig binär addition och subtraktion kan användas även för tal med 2-komplementrepresentation. Det problem som kan inträffa kallas **overflow** och innebär att gränsen för det tillgängliga talområdet har överskridits.

Vid addition av två tal kan overflow inträffa om bägge talen är positiva eller om bägge talen är negativa, eftersom summans absolutbelopp då blir större än absolutbeloppet av det största av talen i additionen. Om summans absolutbelopp överskrider talområdets gräns, på den positiva eller negativa sidan, så inträffar overflow. Man upptäcker enkelt overflow genom att betrakta teckenbitarna hos de bägge talen och hos summan. Om man adderar två tal med lika tecken och summan får motsatt tecken, så har overflow inträffat. (Pos + Pos = Neg eller Neg + Neg = Pos)

Vid subtraktion av ett tal från ett annat tal kan overflow inträffa om det första talet i subtraktionen är positivt och det andra negativt eller tvärt om. Skillnadens absolutbelopp blir ju då större än det största talets absolutbelopp. Overflow har inträffat om subtraktion av ett negativt tal från ett positivt tal ger en negativ summa (Pos – Neg = Neg) eller om subtraktion av ett positivt tal från ett negativt ger en positiv summa (Neg – Pos = Pos)

Dessa iakttagelser leder oss till följande booleska uttryck för overflowfunktionen V för additionen S = X +Y eller subtraktionen S = X – Y, där X, Y och S är 8-bitars tal med 2-komplementrepresentation och med teckenbitarna x_7, y_7 och s_7:

$V(x_7,y_7,s_7) = (SUB)'(x_7'y_7's_7 + x_7y_7s_7') + SUB(x_7'y_7s_7 + x_7y_7's_7')$

Variablen SUB får värdet 1 vid subtraktion.

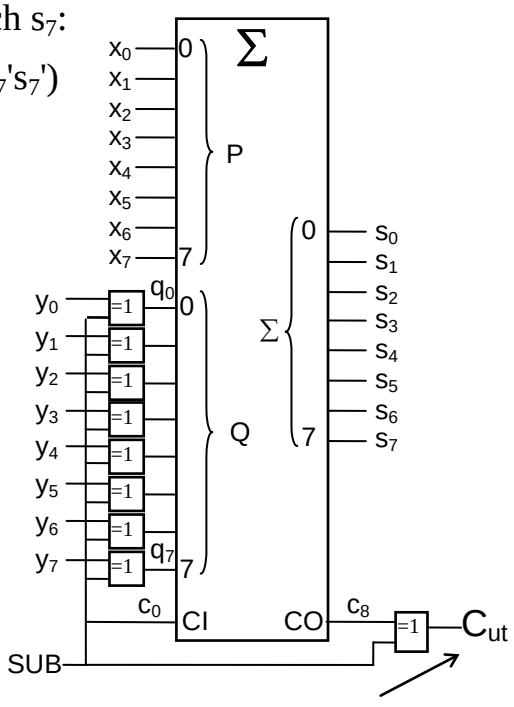
Subtraktionen X – Y utförs normalt med en adderare enligt standardmetoden $X + (2^8 – Y) = X + Y_{1k} + 1$, enligt figuren till höger. Detta innebär att talet Y finns på adderarens Q-ingång vid addition och talet Y_{1k} finns på Q-ingången vid subtraktion.

Vid addition är därför $q_7 = y_7$ och vid subtraktion är $q_7 = y_7'$.

Genom att uttrycka V som en funktion av x_7, q_7 och s_7 kan man få ett enklare uttryck.

$V = (SUB)'(x_7'q_7's_7 + x_7q_7s_7') + SUB(x_7'q_7's_7 + x_7q_7s_7') =$

$= [(SUB)' + SUB](x_7'q_7's_7 + x_7q_7s_7') = x_7'q_7's_7 + x_7q_7s_7'$



Dvs addition och subtraktion får samma overflowvillkor uttryckt i adderarens teckenbitar.

Digital- och datorteknik

Synkrona sekvensnät

Analys

Syntes av räknare

Syntes av små sekvensnät

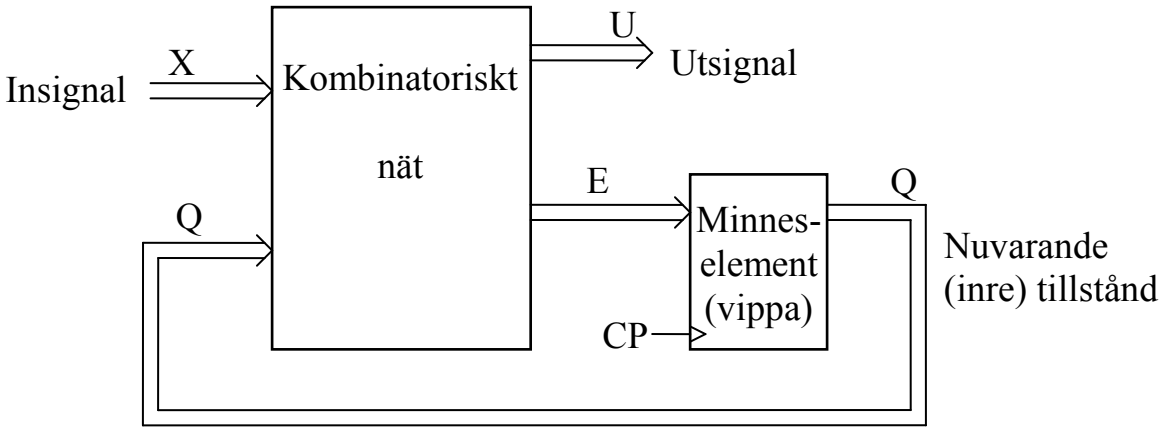
Extrauppgifter

X.1 - X.14

på analys och syntes

Analys av synkrona sekvensnät

Allmän modell av ett synkront sekvensnät



$E = f(Q,X)$
$Q^+ = g(E) = \delta(Q,X)$
$U = \lambda(Q,X)$

(Q^+ är Q-värdet efter nästa klockpuls. Q^+ är alltså nästa tillstånd.)

Analys innebär att man bestämmer ***f-, g-, δ-*** och λ-**funktionerna** som **booleska uttryck** och som **tabeller**.

Från **δ(λ)-tabellen** ritar man sedan **tillståndsgrafen**, som grafiskt visar hur övergångar mellan inre tillstånd sker. Eventuellt kompletteras tillståndsgrafen med pulsdiagram, som visar klockpulser, tillståndsvariabler och utsignaler.

Arbetsgång:

1. Tag fram $E = f(Q,X)$, $Q^+ = \delta(Q,X)$, $U = \lambda(Q,X)$ i form av booleska uttryck.

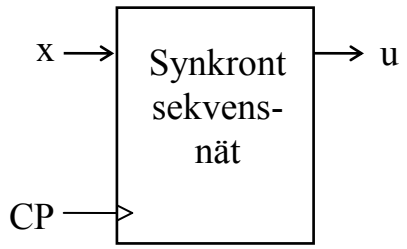
2. Sammanställ $E = f(Q,X)$, $Q^+ = \delta(Q,X)$, $U = \lambda(Q,X)$ i form av en δ(λ)-tabell.

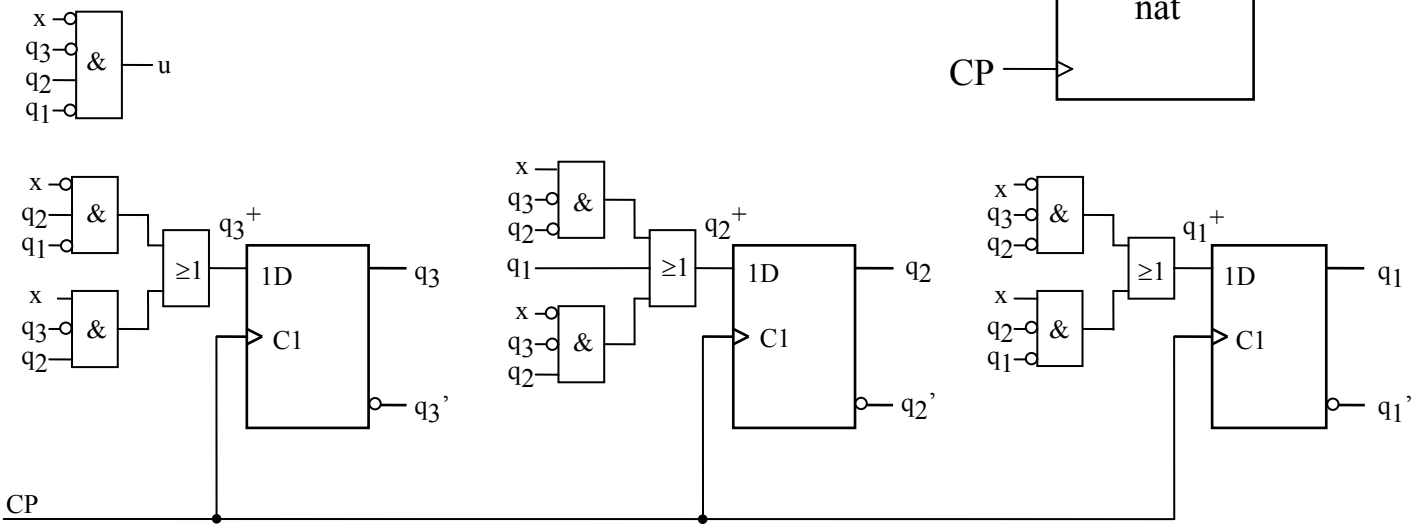
3. Rita en tillståndsgraf utgående från tabellen.

4. Rita eventuellt pulsdiagram för CP, Q och U.

Exempel på analys:

Tillämpa analysstegen 1 - 3 på nedanstående nät.





1.

$q_1^+ =$

$q_2^+ =$

$q_3^+ =$

$u =$

2.

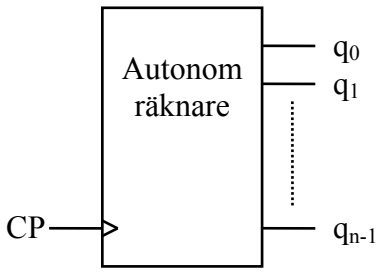
x	q_3	q_2	q_1	q_3^+	q_2^+	q_1^+	u
0	0	0	0				
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0				
1	0	1	1				
1	1	0	0				
1	1	0	1				
1	1	1	0				
1	1	1	1				

3.

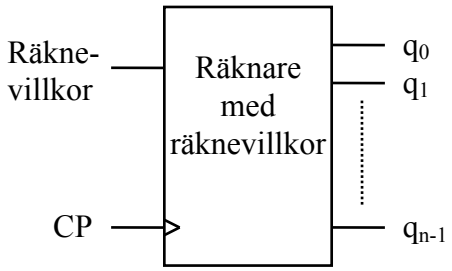
Syntes (konstruktion) av synkrona räknare

En (synkron) **räknare** registrerar antalet klockpulser som uppträder på klockingången. Räknaren är **synkron** om samtliga vippor är kopplade till samma klocksignal och triggar på samma flank. Räknarens utsignaler utgörs av Q-värdena hos räknarens samtliga vippor. En räknare sägs räkna **modulo-N** om den antar N olika tillstånd då M > N klockpulser uppträder på klockingången. En modulo-N räknare antar alltså samma tillstånd för klockpuls nr i och nr N + i.

En **autonom** räknare saknar insignaler som styr hur räknaren räknar och följer därför alltid en och samma tillståndssekvens. Se figuren nedan.



En räknare med **räknevillkor** har en eller flera insignaler som styr vilket nästa tillstånd räknaren antar. Med k styrsignaler kan man få en räknare att räkna med maximalt 2^k olika tillståndssekvenser.



Övningsuppgifter:

1. Konstruera en autonom synkron modulo-6 räknare med räknesekvens och vipptyp enligt nedan. Samtliga vipptyper nedan triggar på positiv flank. Dessutom får AND-, OR- och INVERTERAR-grindar användas.

   a) Räknesekvens: $q_1q_2q_3$ = 000, 001, 101, 111, 011, 010, 000. Vipptyp: D.
   b) Räknesekvens: - " - Vipptyp: SR.
   c) Räknesekvens: - " - Vipptyp: JK.
   d) Räknesekvens: - " - Vipptyp: T.

2. Konstruera en synkron räknare med en styrsignal x, som bestämmer räknarens funktion på följande sätt

   x = 0 Räknaren räknar enligt sekvensen q_1q_2 = 00, 01, 11, 10, 00
   x = 1 Räknaren räknar enligt sekvensen q_1q_2 = 00, 10, 11, 01, 00

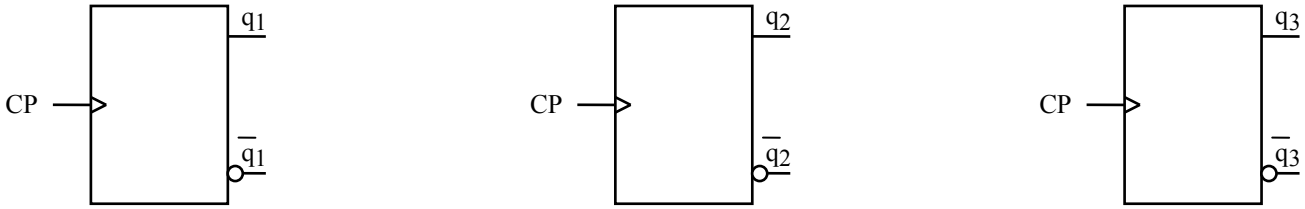
   Av de två sekvenserna ovan framgår att räknaren är **reversibel** dvs den kan räkna en bestämd sekvens framåt eller bakåt.

   JK-vippor som triggar på positiv flank, NAND-grindar och INVERTERARE får användas.

Lösningar på övningsuppgifter, syntes av synkrona räknare.

1. Antag att tillstånden skrivs som $q_1q_2q_3$.
Lösningen visas här för samtliga typer av vippor vi har gått igenom!

Räknaren har tre vippor med utsignalerna q_1, q_2 och q_3 enligt figuren nedan.



Det gäller nu att bestämma insignalerna till vipporna så att räknesekvensen blir den rätta. Vi ställer därför upp en tabell som visar nuvarande (q_i) och nästa tillstånd (q_i^+) samt de insignalvärden som ger "rätt" nästa tillstånd för de fyra olika typerna av vippor. Dessa insignalvärden får man ur excitations-tabellerna nedan.

Excitationstabeller:

D-vippan

q	q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

SR-vippan

q	q^+	S	R
0	0	0	–
0	1	1	0
1	0	0	1
1	1	–	0

JK-vippan

q	q^+	J	K
0	0	0	–
0	1	1	–
1	0	–	1
1	1	–	0

T-vippan

q	q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

q_1	q_2	q_3	q_1^+	q_2^+	q_3^+	D_1	D_2	D_3	S_1	R_1	S_2	R_2	S_3	R_3	J_1	K_1	J_2	K_2	J_3	K_3	T_1	T_2	T_3
0	0	0	0	0	1	0	0	1	0	–	0	–	1	0	0	–	0	–	1	–	0	0	1
0	0	1	1	0	1	1	0	1	1	0	0	–	–	0	1	–	0	–	–	0	1	0	0
0	1	0	0	0	0	0	0	0	0	–	0	1	0	–	0	–	–	1	0	–	0	1	0
0	1	1	0	1	0	0	1	0	0	–	–	0	0	1	0	–	–	0	–	1	0	0	1
1	0	0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	0	1	1	1	1	1	1	1	–	0	1	0	–	0	–	0	1	–	–	0	0	1	0
1	1	0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	1	1	0	1	1	0	1	1	0	1	–	0	–	0	–	1	–	0	–	0	1	0	0

För D-vippor gäller att nästa tillstånd (q^+) är lika med D-värdet före klockningen. Därför skall D-värdet för varje D-vippa väljas till motsvarande q^+-värde, vilket framgår av tabellen ovan.

Karnaughdiagram för vippornas insignaler visas på nästa sida.

a) D-vippor

D_1	00	01	11	10
0	0	1	0	0
1	–	1	0	–

$D_1 = \overline{q_2} q_3$

D_2	00	01	11	10
0	0	0	1	0
1	–	1	1	–

$D_2 = q_1 + q_2 q_3$

D_3	00	01	11	10
0	1	1	0	0
1	–	1	1	–

$D_3 = q_1 + \overline{q_2}$

b) SR-vippor

S_1	00	01	11	10
0	0	1	0	0
1	–	–	0	–

$S_1 = \overline{q_2} q_3$

S_2	00	01	11	10
0	0	0	–	0
1	–	1	–	–

$S_2 = q_1$

S_3	00	01	11	10
0	1	–	0	0
1	–	–	–	–

$S_3 = \overline{q_2}$

R_1	00	01	11	10
0	–	0	–	–
1	–	0	1	–

$R_1 = q_2$

R_2	00	01	11	10
0	–	–	0	1
1	–	0	0	–

$R_2 = \overline{q_3}$

R_3	00	01	11	10
0	0	0	1	–
1	–	0	0	–

$R_3 = \overline{q_1} q_2$

c) JK-vippor

J_1	00	01	11	10
0	0	1	0	0
1	–	–	–	–

$J_1 = \overline{q_2} q_3$

J_2	00	01	11	10
0	0	0	–	–
1	–	1	–	–

$J_2 = q_1$

J_3	00	01	11	10
0	1	–	–	0
1	–	–	–	–

$J_3 = \overline{q_2}$

K_1	00	01	11	10
0	–	–	–	–
1	–	0	1	–

$K_1 = q_2$

K_2	00	01	11	10
0	–	–	0	1
1	–	–	0	–

$K_2 = \overline{q_3}$

K_3	00	01	11	10
0	–	0	1	–
1	–	0	0	–

$K_3 = \overline{q_1} q_2$

d) T-vippor

T_1	00	01	11	10
0	0	1	0	0
1	–	0	1	–

$T_1 = \overline{q_1}\,\overline{q_2} q_3 + q_1 q_2$

T_2	00	01	11	10
0	0	0	0	1
1	–	1	0	–

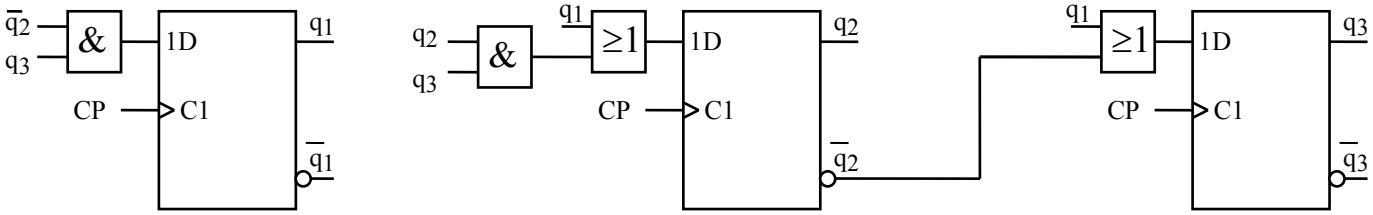
$T_2 = q_1 \overline{q_2} + q_2 \overline{q_3}$

T_3	00	01	11	10
0	1	0	1	0
1	–	0	0	–

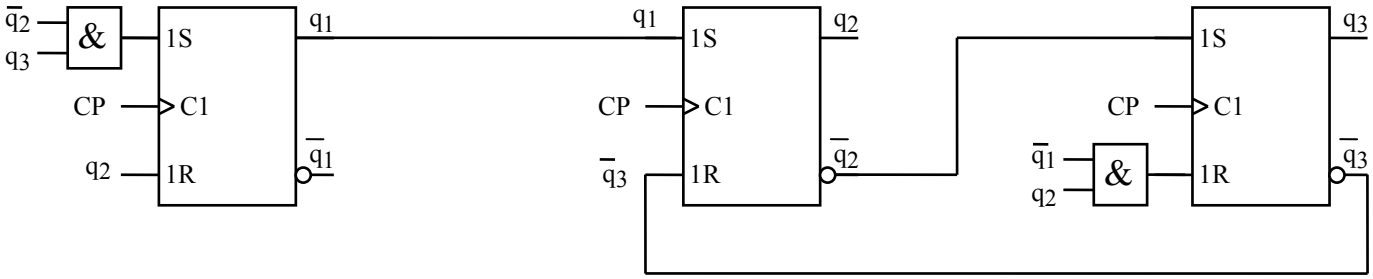
$T_3 = \overline{q_2}\,\overline{q_3} + \overline{q_1} q_2 q_3$

Nedan visas den kompletta räknaren med de olika vipptyperna.

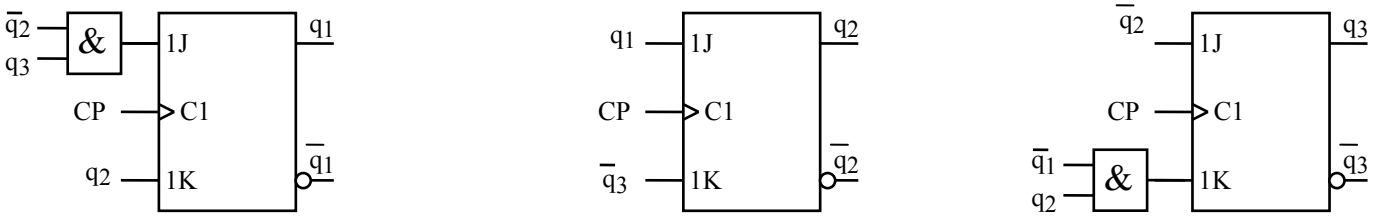
a)



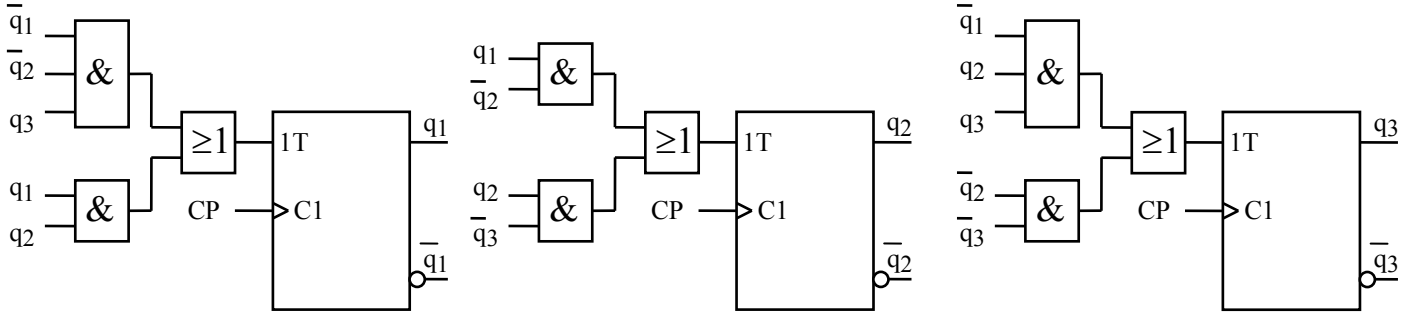
b)



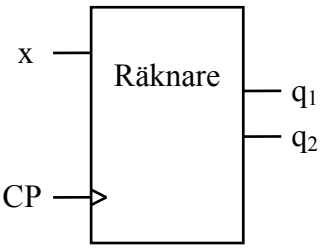
c)



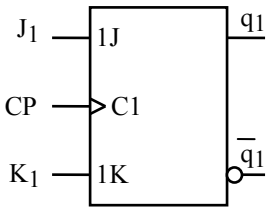
d)

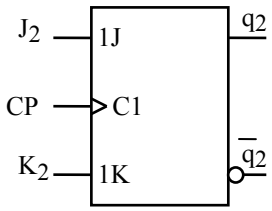


2.



Räknaren har en insignal x och två utsignaler q_1 och q_2. Den innehåller alltså två JK-vippor.





Konstruktion av räknaren innebär att man skall bilda vippornas insignaler J_1, K_1, J_2 och K_2 av signalerna x, q_1 och q_2 så att rätt räknesekvenser bildas, dvs så att alla (q_1,q_2) följs av rätt (q_1^+,q_2^+).

x = 0 Räknaren räknar enligt sekvensen q_1q_2 = 00, 01, 11, 10, 00
x = 1 Räknaren räknar enligt sekvensen q_1q_2 = 00, 10, 11, 01, 00

Tillståndstabell:

q_1	q_2	x	q_1^+	q_2^+	J_1	K_1	J_2	K_2
0	0	0	0	1	0	–	1	–
0	0	1	1	0	1	–	0	–
0	1	0	1	1	1	–	–	0
0	1	1	0	0	0	–	–	1
1	0	0	0	0	–	1	0	–
1	0	1	1	1	–	0	1	–
1	1	0	1	0	–	0	–	1
1	1	1	0	1	–	1	–	0

Excitationstabell:

q	q^+	J	K
0	0	0	–
0	1	1	–
1	0	–	1
1	1	–	0

Tillståndstabellen består av $2^3 = 8$ rader eftersom vi har tre variabler.

Man fyller först i kolumnerna q_1^+och q_2^+ med hjälp av de givna insignalsekvenserna.

Därefter använder man excitationstabellen och fyller i J_i- och K_i-värden för varje par (q_i,q_i^+) för de två vipporna.

Med hjälp av tillståndstabellen ritas Karnaughdiagram för J_1, K_1, J_2 och K_2. Se nästa sida!

J_1:

q_1 \ q_2x	00	01	11	10
0	0	1	0	1
1	–	–	–	–

$J_1 = \overline{q}_2x + q_2\overline{x} = q_2 \oplus x$

J_2:

q_1 \ q_2x	00	01	11	10
0	1	0	–	–
1	0	1	–	–

$J_2 = q_1x + \overline{q}_1\overline{x} = \overline{(q_1 \oplus x)}$

K_1:

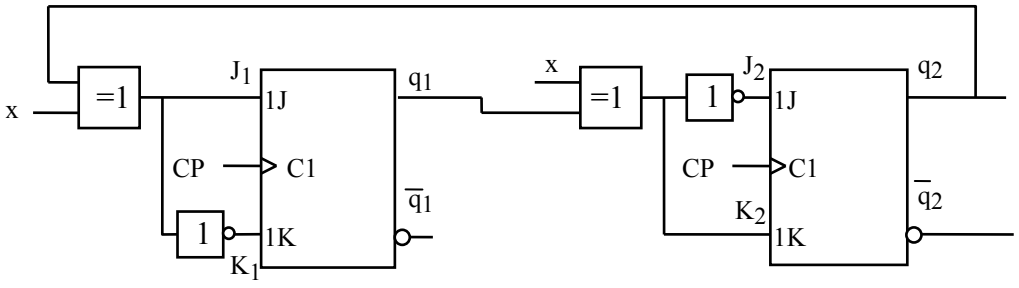
q_1 \ q_2x	00	01	11	10
0	–	–	–	–
1	1	0	1	0

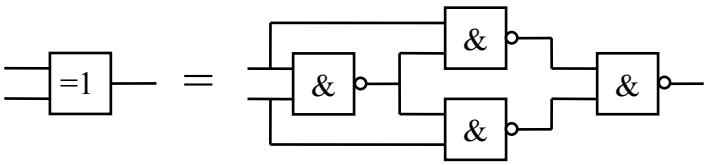
$K_1 = \overline{q}_2\overline{x} + q_2x = \overline{(q_2 \oplus x)}$

K_2:

q_1 \ q_2x	00	01	11	10
0	–	–	1	0
1	–	–	0	1

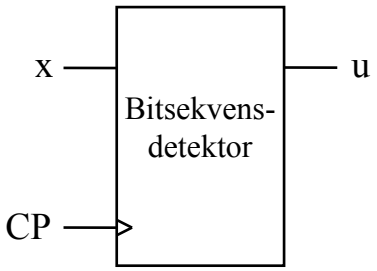
$K_2 = \overline{q}_1x + q_1\overline{x} = q_1 \oplus x$





Konstruktion av enkla synkrona sekvensnät

Metodiken vid konstruktion av enkla synkrona sekvensnät visas här genom att ett sekvensnät för detektering av en speciell bitsekvens, en bitsekvensdetektor, tas fram.



Bitsekvensdetektorn i figuren ovan skall uppfylla följande specifikation:

1. Bitsekvensdetektorn skall vara ett synkront sekvensnät med en insignal x och en utsignal u.

2. Utsignalen u skall anta värdet 1 om och endast om insignalen x haft värdena 11000 i de fem senaste klockpulsintervallen.

3. Utsignalen u=1 skall uppträda i samma klockpulsintervall som den sista nollan i en korrekt insignalsekvens.

4. Insignalen x är synkroniserad med klocksignalen CP så att om x ändrar värde så sker det direkt efter positiv flank på klocksignalen.

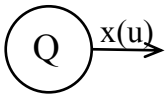
5. Sekvensnätet skall konstrueras med D-vippor (högst 3 st) med positiv flanktriggning samt ett minimalt antal standardgrindar.

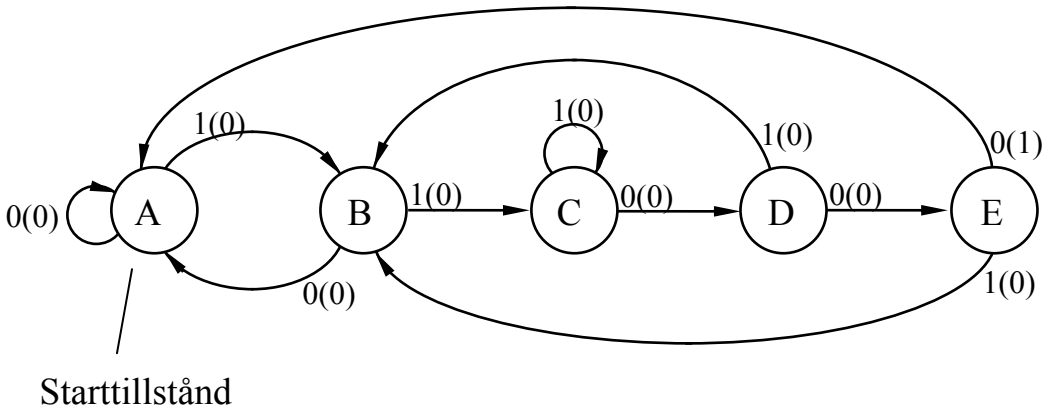
6. Det får förutsättas att alla vippor i sekvensnätet har Q=0 från starten.

Konstruktionsarbetet

Steg 1:

Rita en tillståndsgraf som uppfyller specifikationen.

Tillstånden ritas här som: 



Denna tillståndsgraf uppfyller specifikationen och vi nöjer oss med detta i vår kurs.

Det finns metoder för att avgöra om en given tillståndsgraf innehåller onödiga tillstånd. Man kan därför först ta fram en icke minimal tillståndsgraf, som uppfyller en given specifikation. Därefter minimerar man grafen och får då en ny mindre tillståndsgraf, som också uppfyller specifikationen. Sådana metoder berörs inte i denna kurs.

Steg 2:

Motsvarande δ(λ)-tabell fylls i.

δ(λ)		x	
		0	1
Q	A	A(0)	B(0)
	B	A(0)	C(0)
	C	D(0)	C(0)
	D	E(0)	B(0)
	E	A(1)	B(0)

Steg 3:

δ(λ)-tabellen kodas binärt, dvs tillstånden A, B, C, D och E binärkodas.

Eftersom antalet tillstånd i detta fall är 5 krävs det minst 3 tillståndsvariabler ($2^2 < 5 \leq 2^3 = 8$) för att varje tillstånd skall få ett unikt 3-bitars kodord.

Vilken binär kod som är bäst (optimal) i någon mening är ett svårt problem att knäcka.

Med våra begränsade kunskaper nöjer vi oss med en godtycklig binär kod, som ger en unik kombination av nollor och ettor för varje tillstånd. Vi hugger till med en kod enligt tabellen nedan, där kodordet för starttillståndet A har valts till 000 enligt punkt 6 i specifikationen!

Q	q_2	q_1	q_0
A	0	0	0
B	0	1	0
C	0	1	1
D	0	0	1
E	1	0	0

Bokstäverna A, B, C, D och E i δ(λ)-tabellen byts nu ut mot de binära kodorden från tabellen ovan. Den resulterande δ(λ)-tabellen visas nedan.

δ(λ)		x	
		0	1
$q_2q_1q_0$	000	000(0)	010(0)
	010	000(0)	011(0)
	011	001(0)	011(0)
	001	100(0)	010(0)
	100	000(1)	010(0)

δ(λ)-tabellen visar alltså nu q_2^+, q_1^+, och q_0^+ och u som funktion av x, q_2, q_1, och q_0 eller annorlunda uttryckt

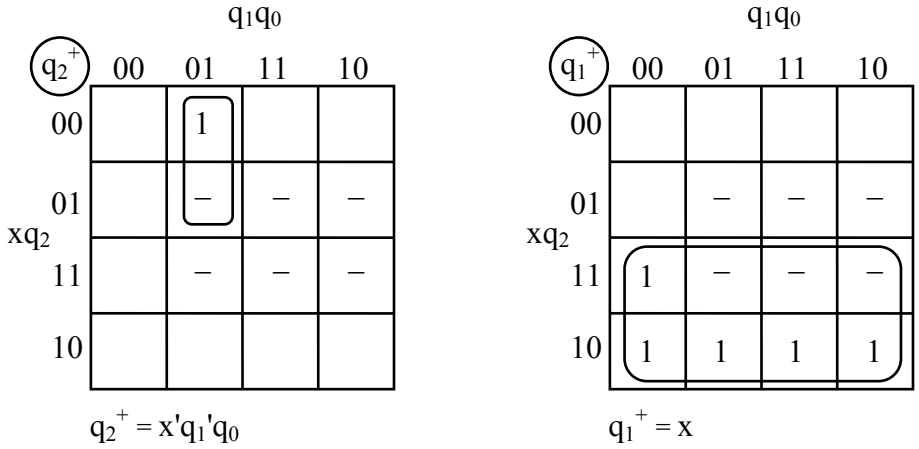
$$Q^+ = \delta(Q,X)$$
$$U = \lambda(Q,X)$$

(Jämför med analys av synkrona sekvensnät på sidan 2.)

Steg 4:

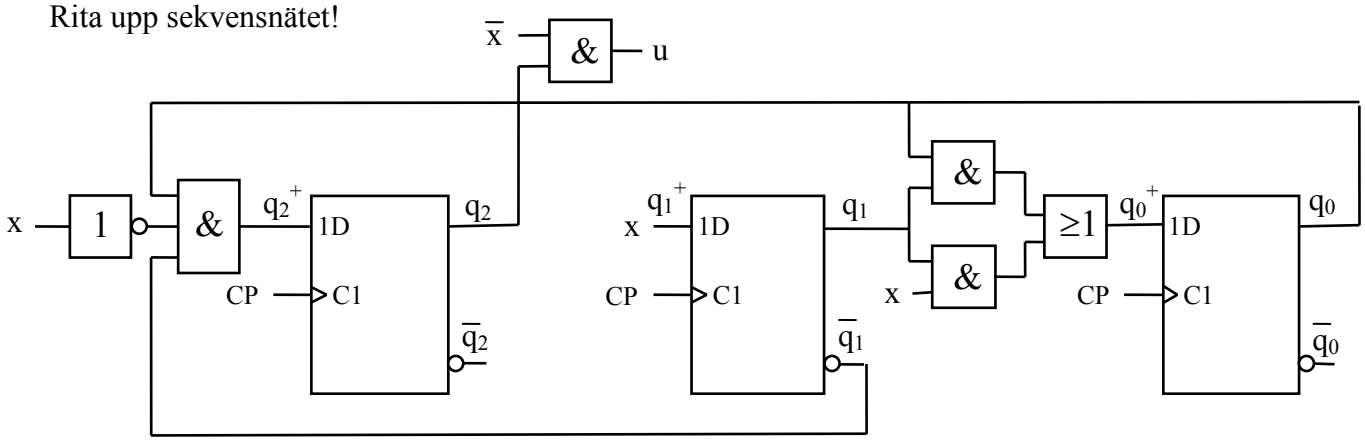
Rita nu Karnaughdiagram för q_2^+, q_1^+, och q_0^+ och u med hjälp av δ(λ)-tabellen.

Observera att endast 5 av 8 möjliga tillstånd används i δ(λ)-tabellen. De 3 oanvända tillstånden ger upphov till 6 st "don't care"-termer i Karnaughdiagrammen.



Steg 5:

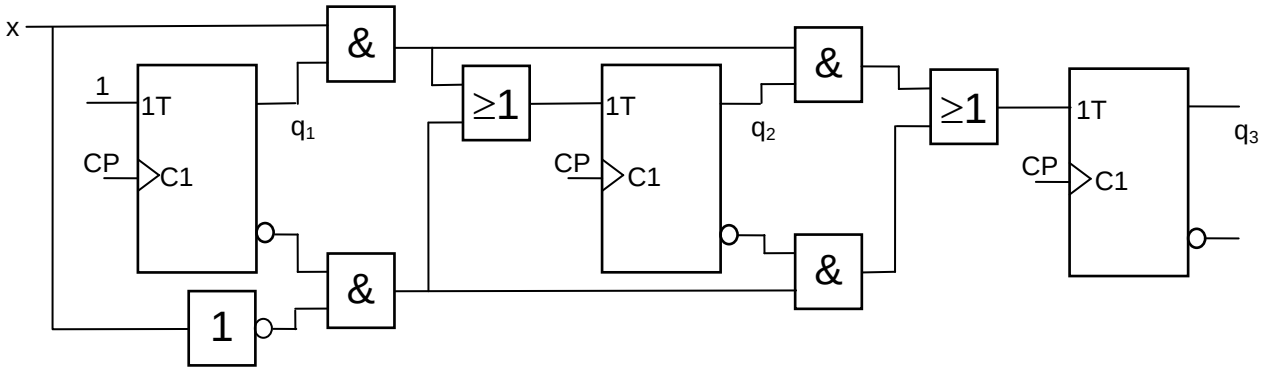
Rita upp sekvensnätet!



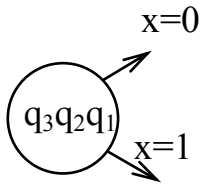
Extrauppgifter, synkrona sekvensnät.

Synkrona räknare. Analys.

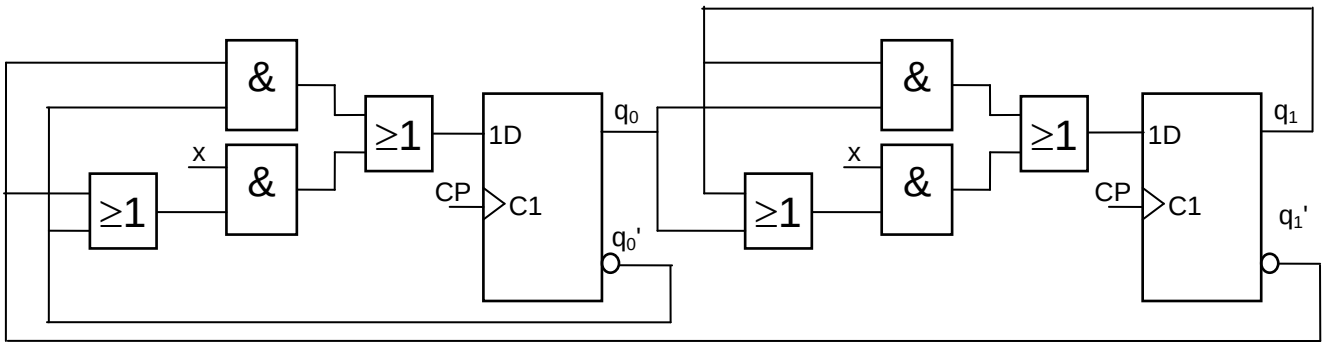
X.1 En räknare med räknevillkoret x visas nedan. Rita en tillståndsgraf för räknaren.



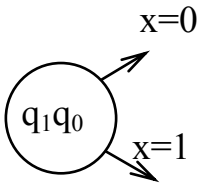
Rita tillstånden enligt:



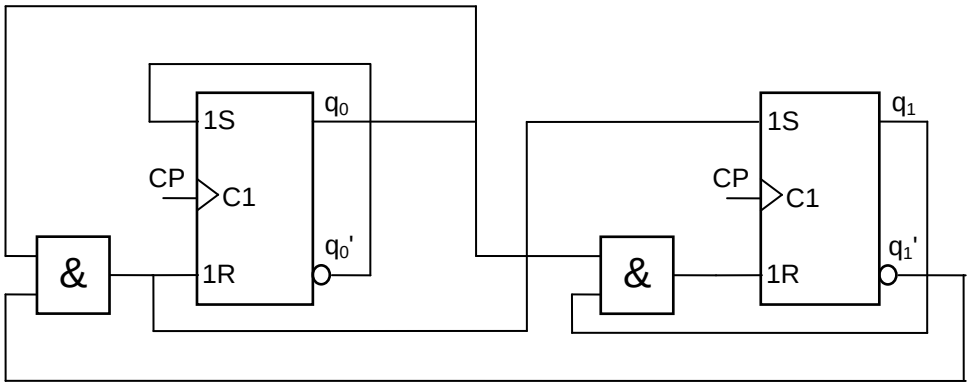
X.2 En räknare med räknevillkoret x visas nedan. Rita en tillståndsgraf för räknaren.



Rita tillstånden enligt:



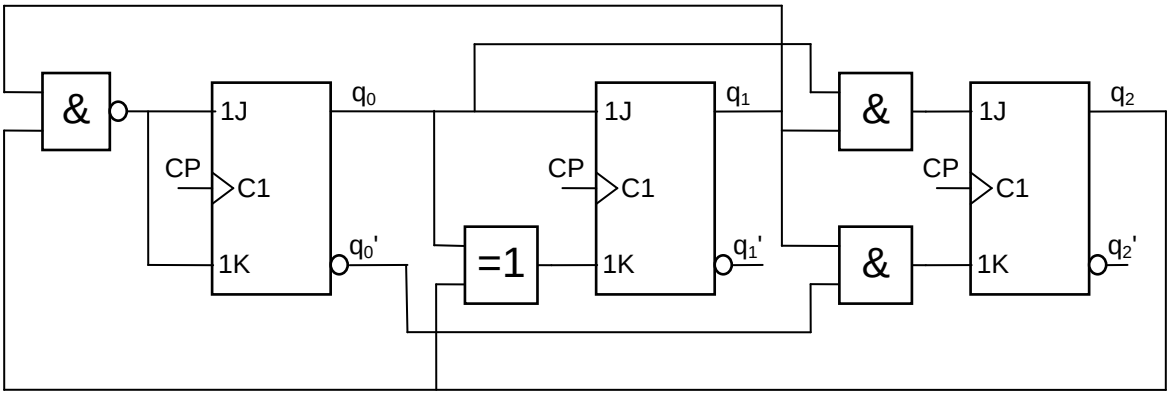
X.3 Rita en tillståndsgraf och rita pulsdiagram för den autonoma räknaren nedan. Pulsdiagrammet skall utgå från tillståndet Q = $(q_1q_0)_2$ = 0

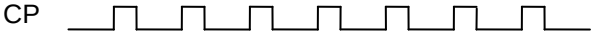


q_0 ___

q_1 ___

X.4 Rita en tillståndsgraf och rita pulsdiagram för den autonoma räknaren nedan. Pulsdiagrammet skall utgå från tillståndet Q = $(q_2q_1q_0)_2$ = 0





q_0 ___

q_1 ___

q_2 ___

Synkrona räknare. Syntes.

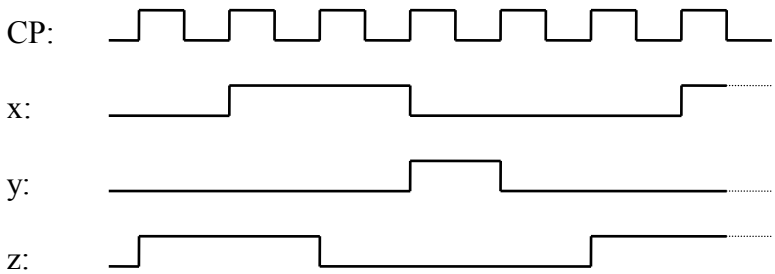
Sekvensnät av denna typ måste starta från ett känt starttillstånd. Inga åtgärder för att försätta sekvensnäten i detta starttillstånd behöver vidtagas i uppgifterna nedan.

X.5 En autonom synkron räknare med räknesekvens enligt figuren nedan skall konstrueras. JK-vippor med positiv flanktriggning, NAND-grindar med valfritt antal ingångar samt INVERTERARE får användas. x, y och z i pulsdiagrammen nedan är utsignaler (q) från var sin vippa.

a) Rita en tillståndsgraf enligt pulsdiagrammen nedan som beskriver räknaren !

b) Realisera räknaren!

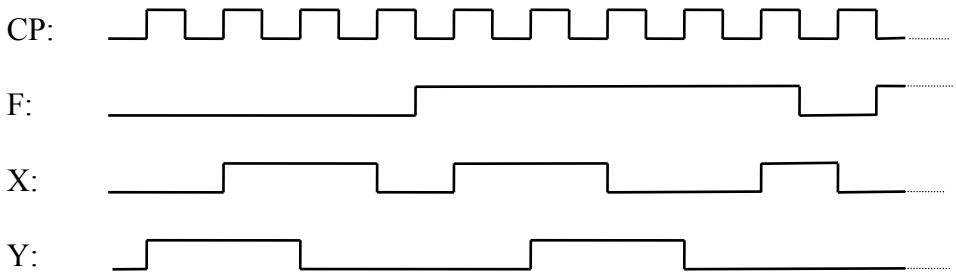
c) Komplettera grafen från a) så att räknarens samtliga tillstånd kommer med!



X.6 En sk stegmotor används för att styra en funktion på en maskin. Styrkretsen för stegmotorn är ett synkront sekvensnät, som förutom klocksignalen CP, har insignalen Fram/Back (F) samt utsignalerna X och Y.

Insignalen F är synkroniserad med klocksignalen CP.

Utsignalerna X och Y är kopplade till stegmotorns drivelektronik. Hur utsignalerna beror av klocksignalen CP och insignalen F framgår av figuren nedan.



X och Y ovan är utsignalen (q) från var sin vippa.

Konstruera sekvensnätet!

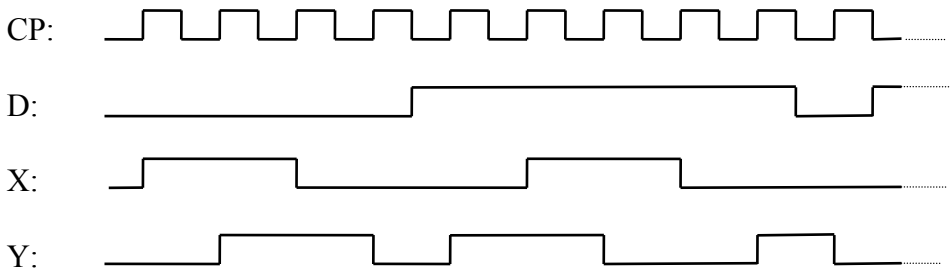
Två JK-vippor med positiv flanktriggning och NAND-grindar med valfritt antal ingångar samt INVERTERARE får användas.

X.7 Konstruera en 2-bitars synkron räknare med insignalen x, som är synkroniserad med klocksignalen.

För x = 0 skall räknesekvensen (q_1q_0) vara 00, 01, 11, 00, ..
För x = 1 skall räknesekvensen (q_1q_0) vara 00, 11, 01, 00, ..

JK-vippor med triggning på positiv flank, NAND-grindar och INVERTERARE får användas.

X.8 En styrkrets för en stegmotor skall konstrueras. Styrkretsen skall utgöras av ett synkront sekvensnät, som förutom klocksignalen CP, har insignalen Direction (D) samt utsignalerna X och Y. Utsignalerna är kopplade till stegmotorns drivelektronik. Hur utsignalerna X och Y beror av klocksignalen CP och insignalen D framgår av figuren nedan.



X och Y ovan är utsignalen (q) från var sin vippa.

Konstruera sekvensnätet!

Två SR-vippor med positiv flanktriggning, NAND-grindar med valfritt antal ingångar, EXCLUSIVE-OR-grindar och INVERTERARE får användas.

X.9 En synkron tvåbitars (q_1q_0) uppräknare med ett räknevillkor x skall konstrueras. Då x = 1 skall räknaren räkna uppåt med vanlig binärkod. Då x = 0 skall räknaren stanna kvar i det tillstånd den befinner sig.

D-vippor med positiv flanktriggning, NOR-grindar med valfritt antal ingångar och INVERTERARE får användas.

a) Rita en tillståndsgraf som beskriver räknaren !

b) Realisera räknaren!

X.10 En synkron tvåbitars (q_1q_0) nedräknare med ett räknevillkor x skall konstrueras.

Då x = 1 skall räknaren räkna nedåt med vanlig binärkod, dvs sekvensen ..., 00, 11, 10, 01, 00, ...
Då x = 0 skall räknaren stanna kvar i det tillstånd den befinner sig.

T-vippor med positiv flanktriggning, NAND-grindar med valfritt antal ingångar och INVERTERARE får användas.

a) Rita en tillståndsgraf som beskriver räknaren !

b) Realisera räknaren!

Enkla synkrona sekvensnät. Syntes.

Sekvensnät av denna typ måste starta från ett känt starttillstånd. Inga åtgärder för att försätta sekvensnäten i detta starttillstånd behöver vidtagas i uppgifterna nedan.

X.11 Ett synkront sekvensnät skall ha en insignal x och en utsignal u. Utsignalen u skall få värdet 1 för varje insignalsekvens som består av exakt en nolla följd av en etta. Med exakt en nolla menas att flera nollor följda av en etta inte skall ge utsignalen ett. Se exemplet nedan!

Exempel: σ_x = 0100101100001 01... (Insignalsekvens)
σ_u = 0100001000000 01... (Utsignalsekvens)

Utsignalen skall ges värdet ett när den sista biten i en korrekt insignalsekvens anländer på x-ingången och behålla värdet ett så länge x har kvar sitt värde under detta klockpulsintervall.

a) Rita en tillståndsgraf för sekvensnätet.

b) Konstruera sekvensnätet.

JK-vippor som triggar på positv flank, NAND-grindar med valfritt antal ingångar och INVERTERARE får användas.

X.12 Ett synkront sekvensnät med en insignal x, och en utsignal u skall konstrueras. Insignalen är synkroniserad med klocksignalen.

Utsignalen u skall få värdet ett varje gång två x-värden i rad har varit olika. Ettan på utgången skall uppträda i samma klockpulsintervall som det andra x-värdet på ingången. Se exemplet nedan!

Exempel: σ_x = 0011000110111001111000101 10000...
σ_u = 0010100101100101000100111 01000...

a) Rita en tillståndsgraf som beskriver sekvensnätet!

b) Realisera sekvensnätet!

JK-vippor som triggar på positiv flank, NAND-grindar med valfritt antal ingångar och INVERTERARE får användas.

X.13 Ett synkront sekvensnät med en insignal x (synkroniserad med klocksignalen) och en utsignal u skall konstrueras.

Utsignalen u skall få värdet ett varje gång exakt två x-värden i rad har varit lika. Ettan på utgången skall uppträda i det klockpulsintervall som följer efter intervallen med de två lika x-värdena på ingången. Se exemplet nedan!

Exempel: σ_x = 0011000110111001111111000010110000...
σ_u = 0010100001000001000000000000001000...

a) Rita en tillståndsgraf som beskriver sekvensnätet!

b) Realisera sekvensnätet!

JK-vippor som triggar på positiv flank, NAND-grindar med valfritt antal ingångar och INVERTERARE får användas.

X.14 Ett synkront sekvensnät med en insignal x (synkroniserad med klocksignalen) och en utsignal u skall konstrueras.

Utsignalen u skall få värdet ett varje gång exakt tre x-värden i rad har varit lika. Ettan på utgången skall uppträda när den sista biten i en korrekt sekvens anländer på ingången. Se exemplet nedan!

Rita en tillståndsgraf som beskriver sekvensnätet! Nätet behöver ej konstrueras! Det räcker med tillståndsgrafen.

Exempel: σ_x = 0010000001101111001111110000101000 0...
σ_u = 0000010000000010000010000010000001 0...

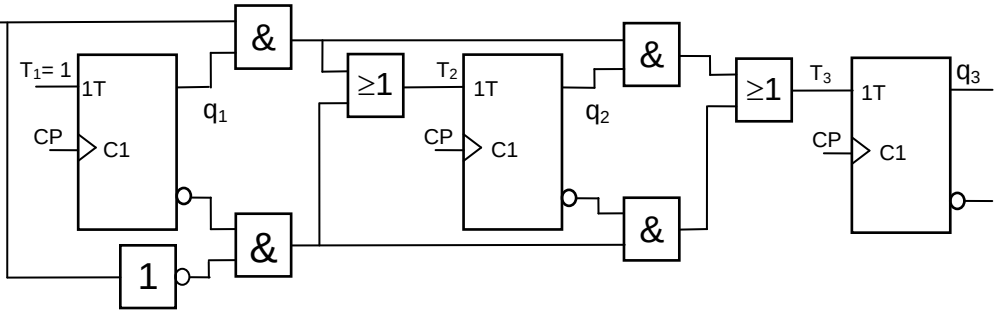
Lösningar till X-uppgifter.

Synkrona sekvensnät

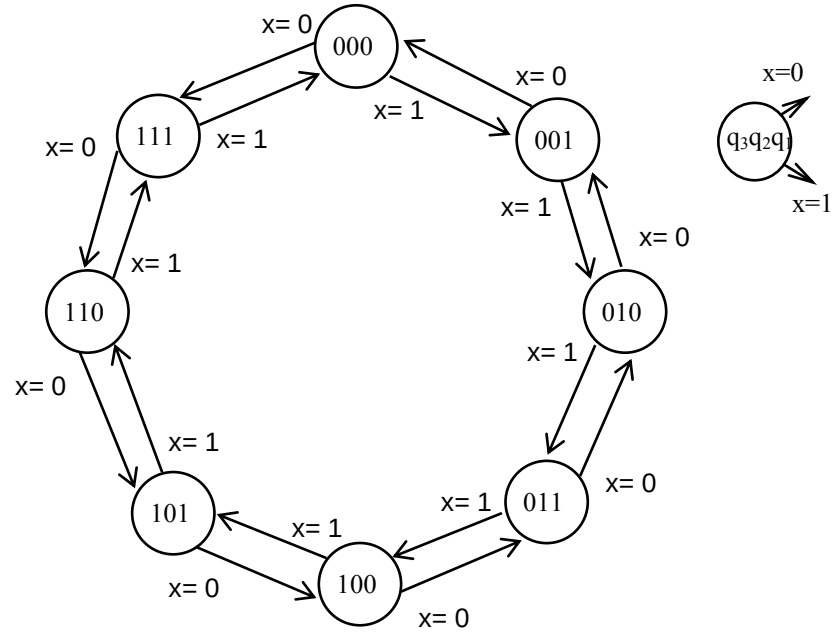
X.1 $T_3 = xq_1q_2 + x'q_1'q_2'$

$T_2 = xq_1 + x'q_1'$

$T_1 = 1$

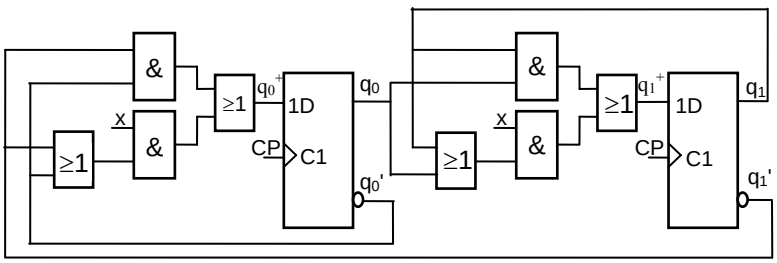


x	q_3	q_2	q_1	T_3	T_2	T_1	q_3^+	q_2^+	q_1^+
0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	1	1	0	0	1
0	0	1	1	0	0	1	0	1	0
0	1	0	0	1	1	1	0	1	1
0	1	0	1	0	0	1	1	0	0
0	1	1	0	0	1	1	1	0	1
0	1	1	1	0	0	1	1	1	0
1	0	0	0	0	0	1	0	0	1
1	0	0	1	0	1	1	0	1	0
1	0	1	0	0	0	1	0	1	1
1	0	1	1	1	1	1	1	0	0
1	1	0	0	0	0	1	1	0	1
1	1	0	1	0	1	1	1	1	0
1	1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	1	0	0	0



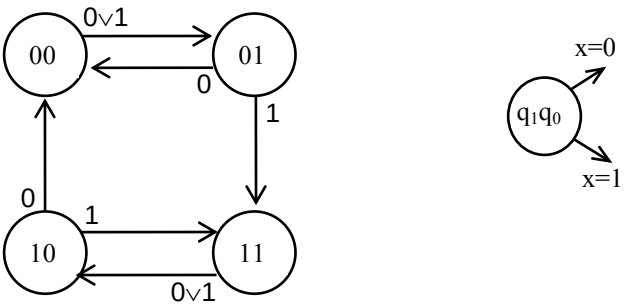
X.2 $q_0^+ = q_1'q_0' + xq_1' + xq_0'$

$q_1^+ = q_1q_0 + xq_1 + xq_0$



x	q_1	q_0	q_1^+	q_0^+
0	0	0	0	1
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

(0∨1 = 0 eller 1)



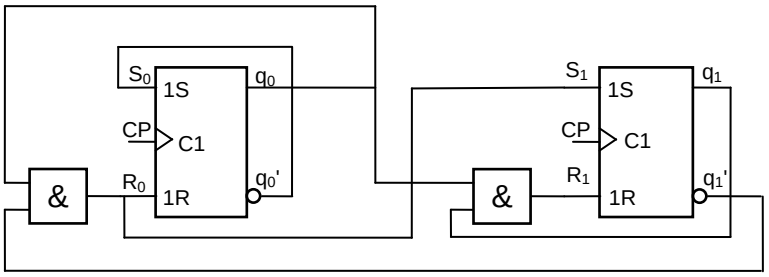
X.3 $S_1 = q_1'q_0$

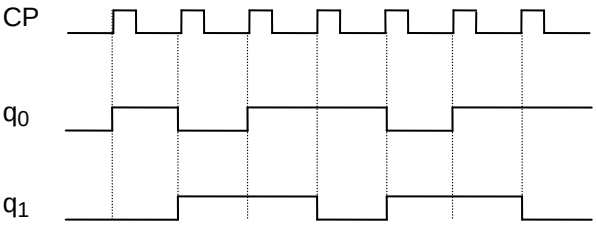
$R_1 = q_1q_0$

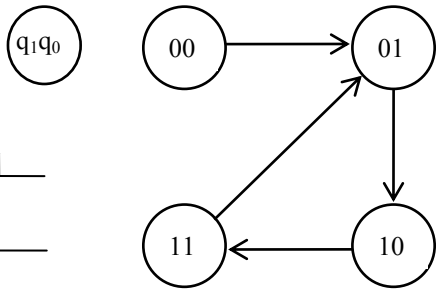
$S_0 = q_0'$

$R_0 = q_1'q_0$ ($= S_1$)

q_1	q_0	S_1	R_1	S_0	R_0	q_1^+	q_0^+
0	0	0	0	1	0	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	1	1
1	1	0	1	0	0	0	1







X.4 $J_2 = q_1q_0$

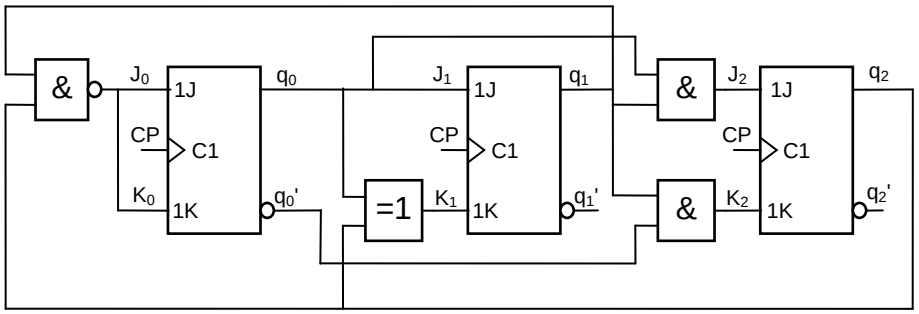
$K_2 = q_1q_0'$

$J_1 = q_0$

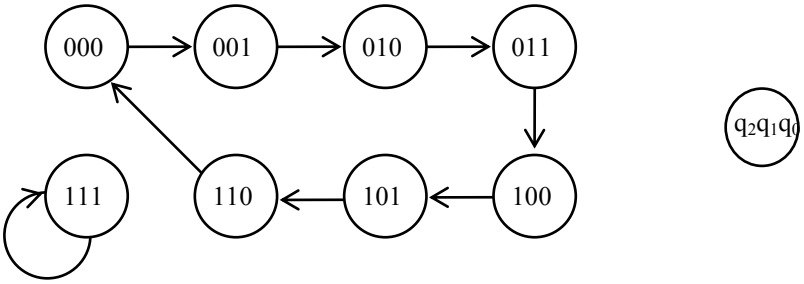
$K_1 = q_2 \oplus q_0$

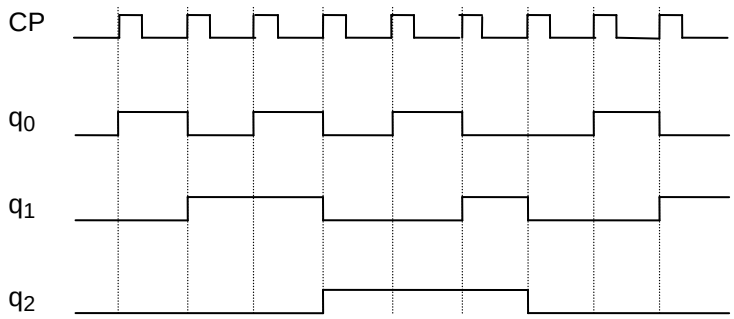
$J_0 = q_2' + q_1'$

$K_0 = J_0$

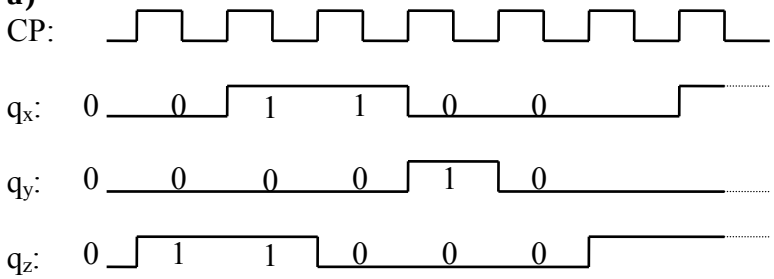


q_2	q_1	q_0	J_2	K_2	J_1	K_1	J_0	K_0	q_2^+	q_1^+	q_0^+
0	0	0	0	0	0	0	1	1	0	0	1
0	0	1	0	0	1	1	1	1	0	1	0
0	1	0	0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	1	1	1	1	0	0
1	0	0	0	0	0	1	1	1	1	0	1
1	0	1	0	0	1	0	1	1	1	1	0
1	1	0	0	1	0	1	0	0	0	0	0
1	1	1	1	0	1	0	0	0	1	1	1

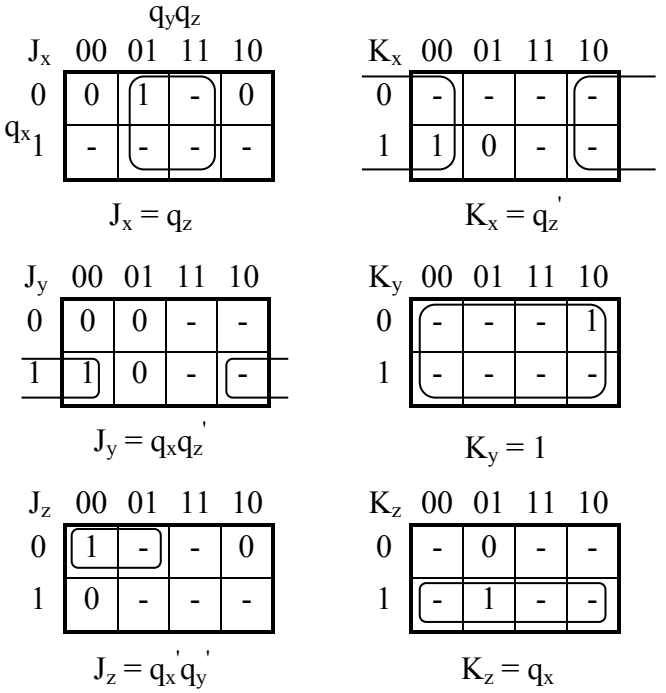


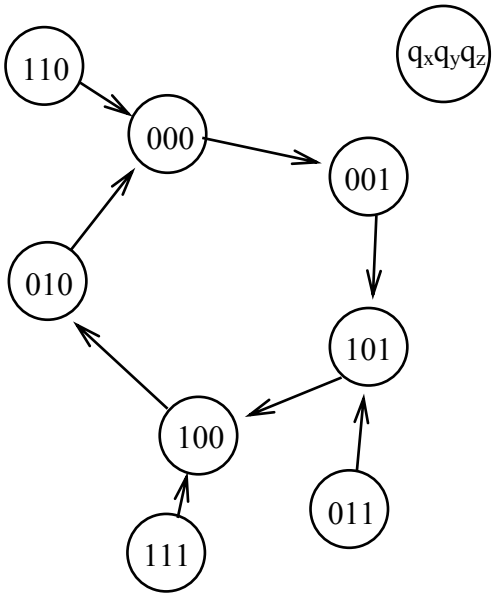


X.5 **a)**

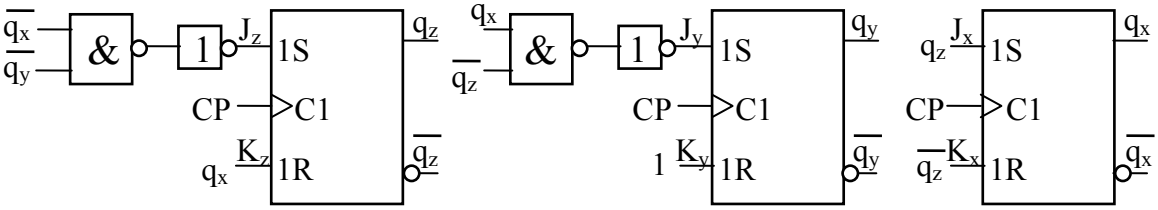


q_x	q_y	q_z	q_x^+	q_y^+	q_z^+	J_x	K_x	J_y	K_y	J_z	K_z
0	0	0	0	0	1	0	-	0	-	1	-
0	0	1	1	0	1	1	-	0	-	-	0
0	1	0	0	0	0	0	-	-	1	0	-
0	1	1	-	-	-	-	-	-	-	-	-
1	0	0	0	1	0	-	1	1	-	0	-
1	0	1	1	0	0	-	0	0	-	-	1
1	1	0	-	-	-	-	-	-	-	-	-
1	1	1	-	-	-	-	-	-	-	-	-





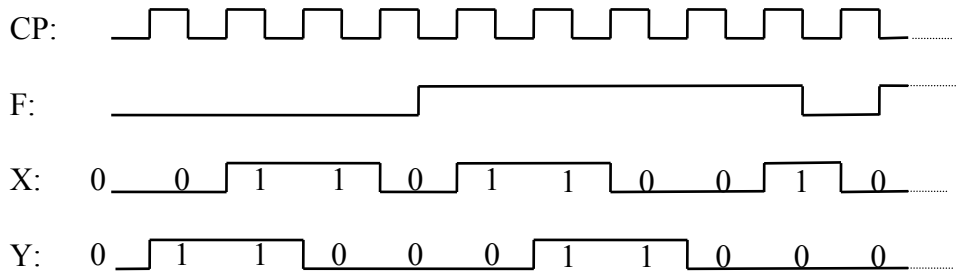
b)



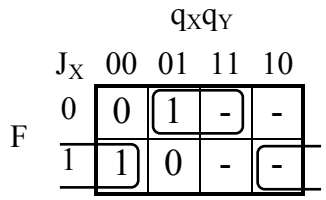
c)

q_x	q_y	q_z	q_x^+	q_y^+	q_z^+	J_x	K_x	J_y	K_y	J_z	K_z
0	0	0	0	0	1	0	~~1~~	0	~~1~~	1	~~0~~
0	0	1	1	0	1	1	~~0~~	0	~~1~~	~~1~~	0
0	1	0	0	0	0	0	~~1~~	~~0~~	1	0	~~0~~
0	1	1	~~1~~	~~0~~	~~1~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~
1	0	0	0	1	0	0	1	1	~~1~~	0	~~1~~
1	0	1	1	0	0	~~1~~	0	0	~~1~~	0	1
1	1	0	~~0~~	~~0~~	~~0~~	~~0~~	~~1~~	~~1~~	~~1~~	~~0~~	~~1~~
1	1	1	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~1~~

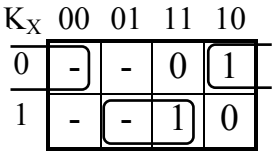
Inritade i graf ovan



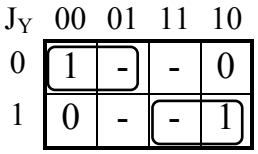
F	q_X	q_Y	q_X^+	q_Y^+	J_X	K_X	J_Y	K_Y
0	0	0	0	1	0	-	1	-
0	0	1	1	1	1	-	-	0
0	1	0	0	0	-	1	0	-
0	1	1	1	0	-	0	-	1
1	0	0	1	0	1	-	0	-
1	0	1	0	0	0	-	-	1
1	1	0	1	1	-	0	1	-
1	1	1	0	1	-	1	-	0



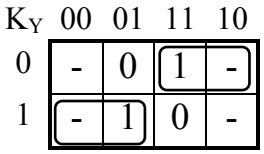
$J_X = F'q_Y + F\,q_Y' = F \oplus q_Y$



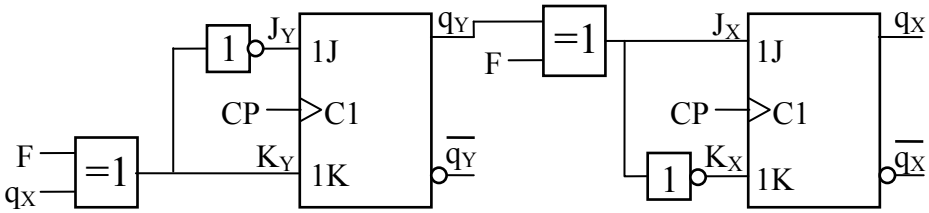
$K_X = F'q_Y' + F\,q_Y = (F \oplus q_Y)'$

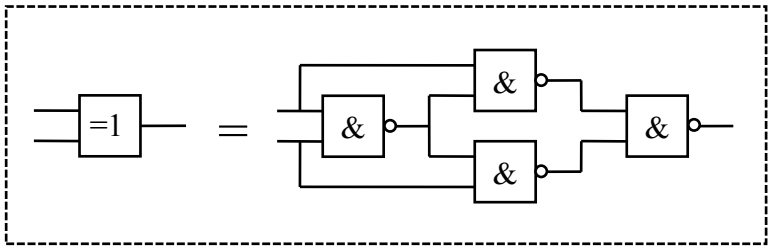


$J_Y = F'q_X' + F\,q_X = (F \oplus q_X)'$



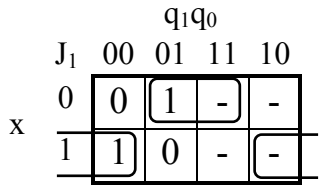
$K_Y = F'q_X + F\,q_X' = F \oplus q_X$



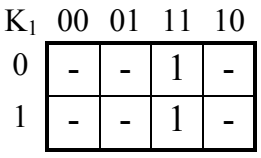


X.7

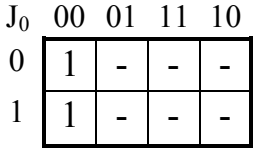
x	q_1	q_0	q_1^+	q_0^+	J_1	K_1	J_0	K_0
0	0	0	0	1	0	-	1	-
0	0	1	1	1	1	-	-	0
0	1	0	-	-	-	-	-	-
0	1	1	0	0	-	1	-	1
1	0	0	1	1	1	-	1	-
1	0	1	0	0	0	-	-	1
1	1	0	-	-	-	-	-	-
1	1	1	0	1	-	1	-	0



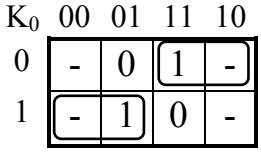
$J_1 = x'q_0 + x\,q_0' = x \oplus q_0$



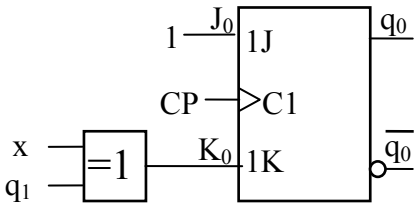
$K_1 = 1$

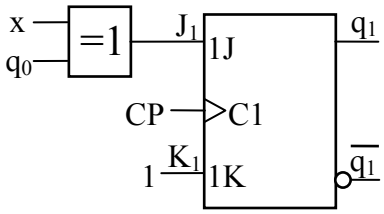


$J_0 = 1$



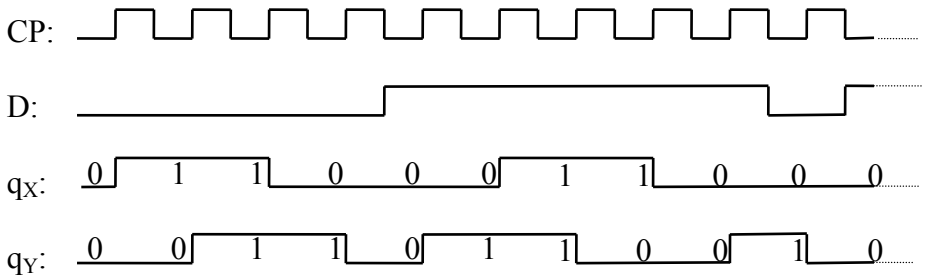
$K_0 = x'q_1 + x\,q_1' = x \oplus q_1$



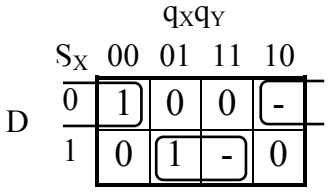


(Realisering av XOR-funktionen med NAND-grindar enligt uppgift X.6)

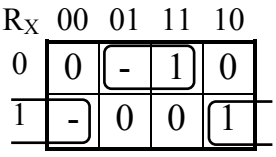
X.8



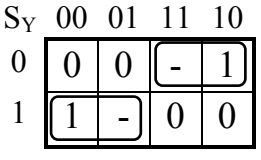
D	q_X	q_Y	q_X^+	q_Y^+	S_X	R_X	S_Y	R_Y
0	0	0	1	0	1	0	0	-
0	0	1	0	0	0	-	0	1
0	1	0	1	1	-	0	1	0
0	1	1	0	1	0	1	-	0
1	0	0	0	1	0	-	1	0
1	0	1	1	1	1	0	-	0
1	1	0	0	0	0	1	0	-
1	1	1	1	0	-	0	0	1



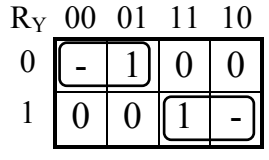
$S_X = D'q_Y' + D\,q_Y = (D \oplus q_Y)'$



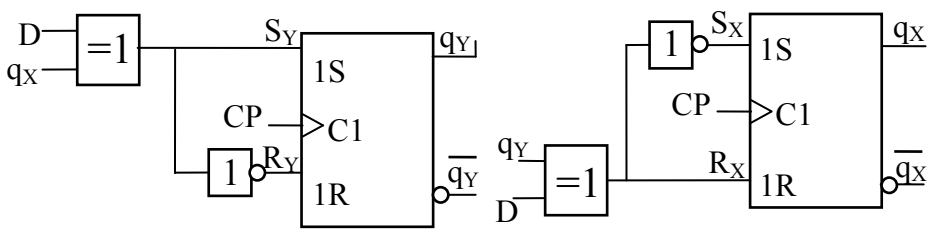
$R_X = D'q_Y + D\,q_Y' = D \oplus q_Y$



$S_Y = D'q_X + D\,q_X' = D \oplus q_X$

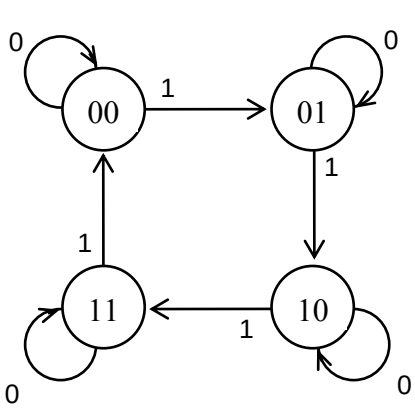


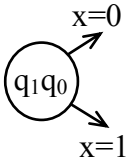
$R_Y = D'q_X' + D\,q_X = (D \oplus q_X)'$

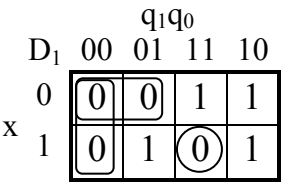


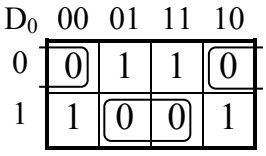
X.9

x	q_1	q_0	$D_1=q_1^+$	$D_0=q_0^+$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

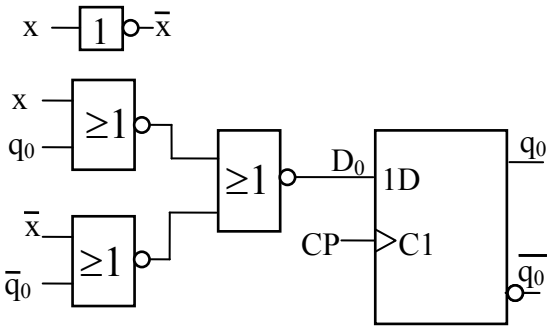


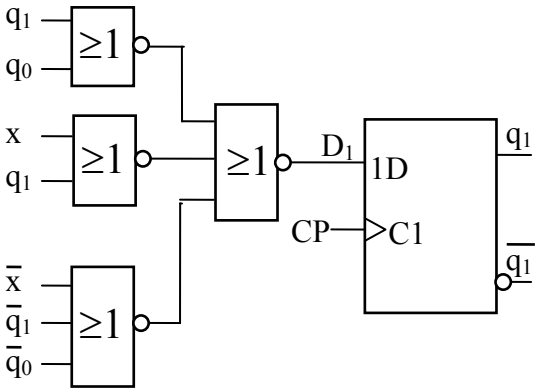






$D_1 = (q_1+q_0)(x+q_1)(x'+q_1'+q_0')$ $D_0 = (x+q_0)(x'+q_0')$

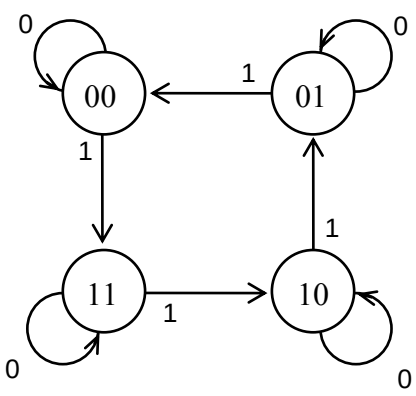


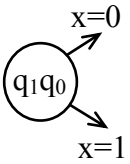


X.10

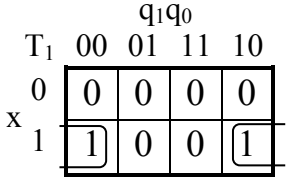
x	q_1	q_0	q_1^+	q_0^+	T_1	T_0
0	0	0	0	0	0	0
0	0	1	0	1	0	0
0	1	0	1	0	0	0
0	1	1	1	1	0	0
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	1	0	0	1	1	1
1	1	1	1	0	0	1

a)

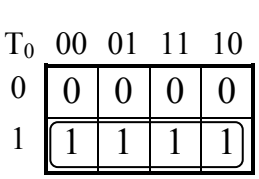




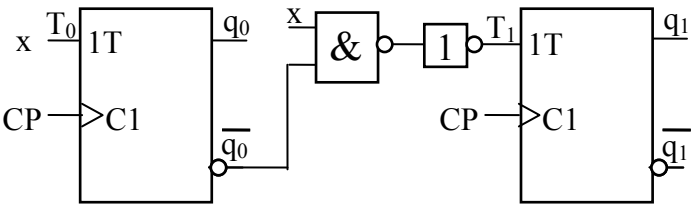
b)



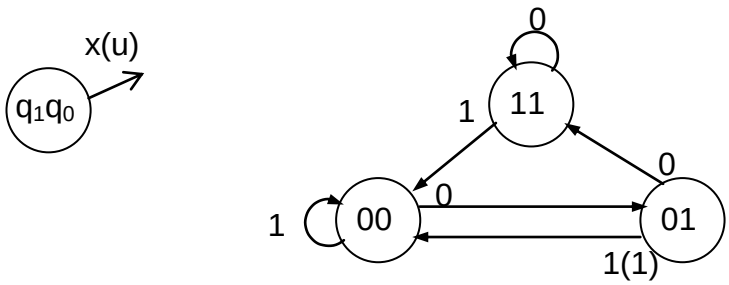
$T_1 = x\ q_0'$



$T_0 = x$

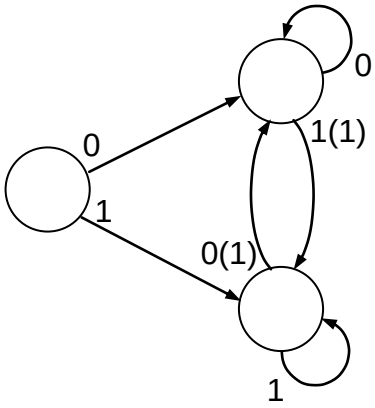


X.11

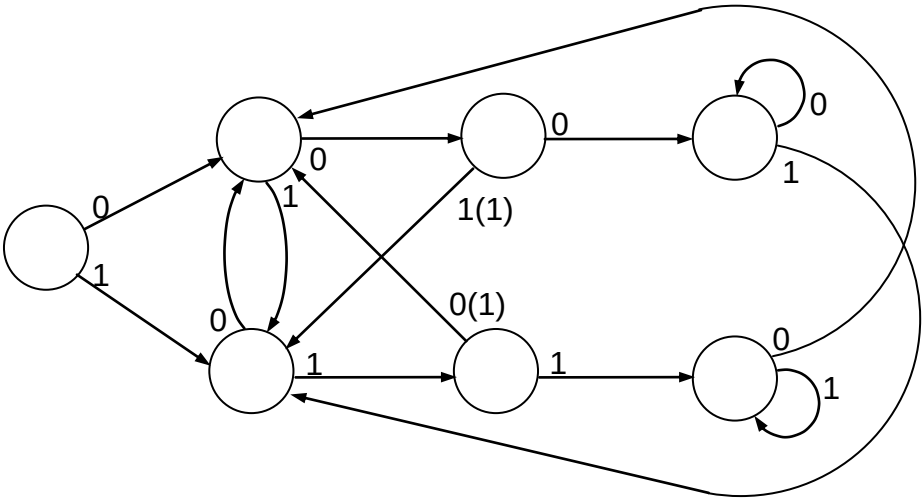


(Ej utsatta utsignaler = 0)

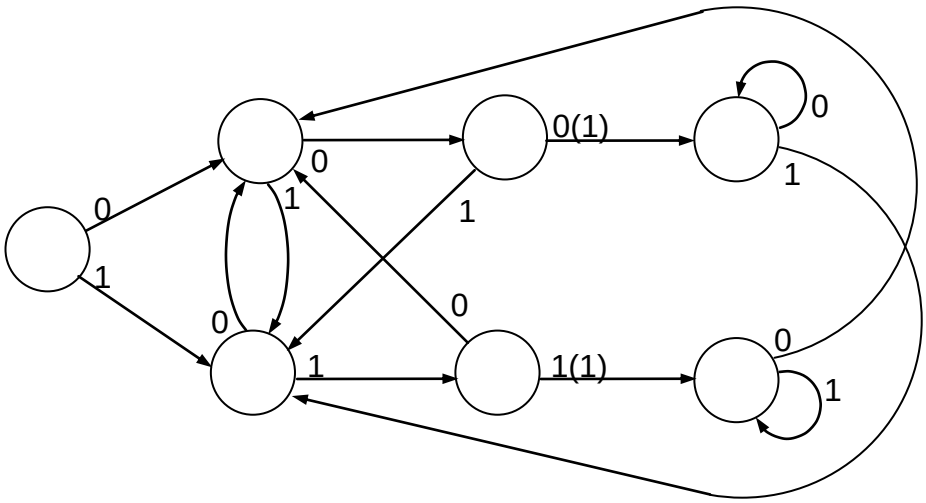
X.12



X.13



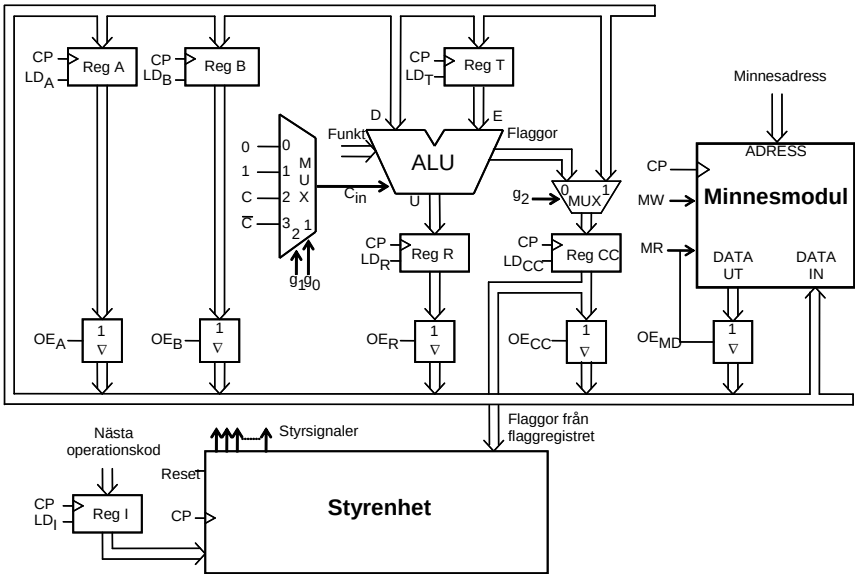
X.14



FLX Datorn enligt von Neumann (≈ kap 10)

Observera att FLIS-processorns instruktionsuppsättning och maskinspråk används i alla programexempel i detta häfte!

Den databehandlande enheten med minne, från kapitel 7, återges nedan i figur F.1. System liknande det i figur F.1 var kända strax efter andra världskrigets slut. Man kunde vid denna tid konstruera processorer vars styrenheter var utformade för att lösa vissa speciella problem, som tex beräkningar av projektilbanor. Det fanns även processorer med modifierbara ("programmerbara") styrenheter. "Programmeringen" gick till så att man ställde in styrenhetens beteende genom att koppla om ledningar och ställa in switchar.



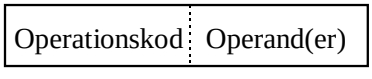
Figur F.1 Databehandlande enhet med minne.

F.1 von Neumanns idé

Den amerikanske matematikern John von Neumann och hans medarbetare bidrog på detta stadium med en genial och mycket enkel idé:

> "*Man kan koda de olika operationerna som binära tal på samma sätt som data och lagra både operationer och data i minnet.*" = "***Det lagrade programmets princip.***"

Data kan behandlas genom att en följd av instruktioner och data växelvis hämtas från minnet. Instruktionerna utför de önskade operationerna på data och skriver vid behov tillbaka data i minnet.



En instruktion är alltså kodad som ett binärt tal i vilket en del av bitarna representerar koden för en operation medan övriga bitar representerar en eller flera operander (data). En typisk instruktion består alltså av två delar enligt figuren till höger.

Operanden är antingen *data* (datavärdet) eller någon form av *adress till data*. Vissa instruktioner saknar dock operand och därmed också operanddel.

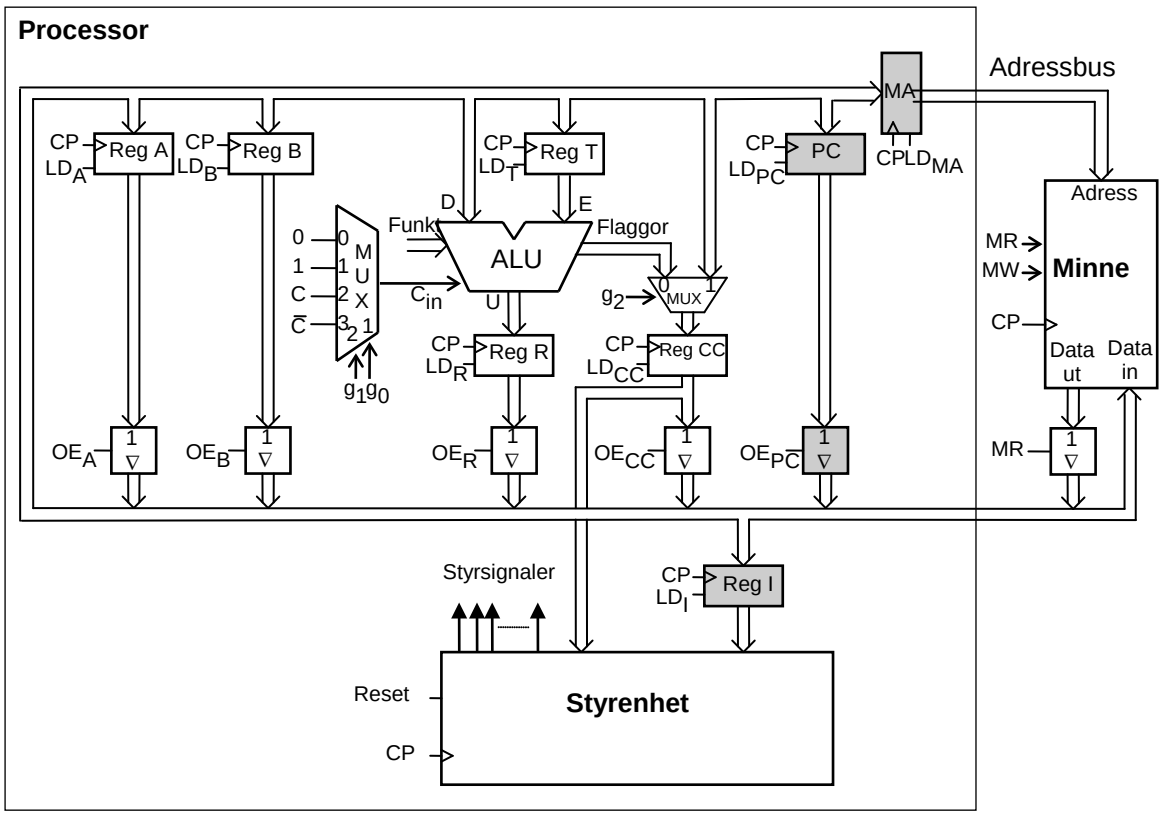
F.2 von Neumannprocessorn

En komplett instruktionssekvens som utför en specifik databehandlingsuppgift kallas ett program. När ett program skall placeras i minnet är det praktiskt om instruktionerna hamnar på konsekutiva (på varandra följande) minnesadresser. Då programmet körs (exekveras) kommer därför processorn att hämta instruktionerna på konsekutiva adresser i minnet.

Det måste finnas en mekanism i processorn som skiljer mellan instruktioner och data eftersom man inte kan se någon skillnad på dem i minnet. Minnesadressen till den instruktion i programmet som står i tur att utföras lagras därför i ett nyinfört register i processorn. Registret kallas programräknaren (eng. program counter, PC) eftersom innehållet normalt ökas med ett för varje ny läsning av en instruktion i minnet. Se figur F.2.

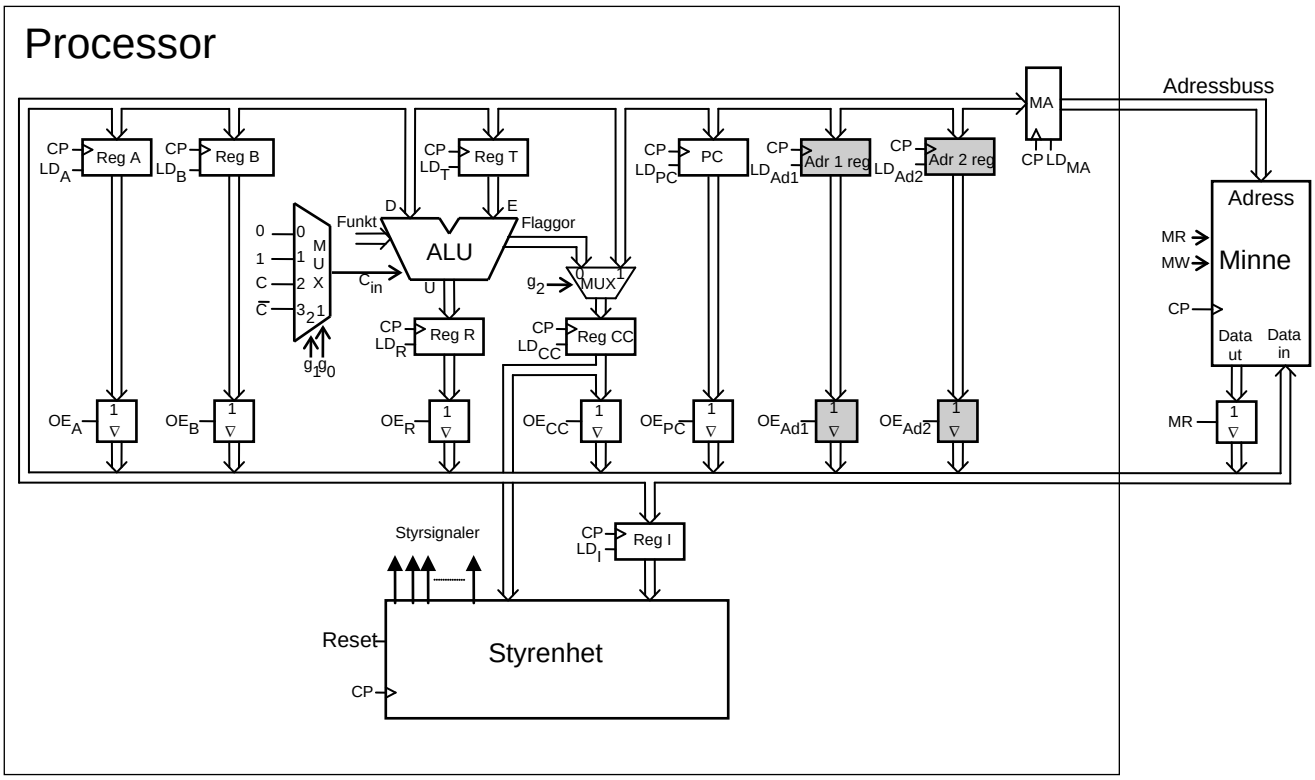
Då processorn skall läsa en instruktion i minnet måste först adressen som finns i PC placeras på minnets adressingång. Av detta skäl har ett minnesadressregister MA (eng **memory address register**) kopplats in mellan databussen och minnets adressingång.

Instruktionsregistrets (I) ingång har dessutom anslutits till databussen så att instruktionens operationskod kan laddas i I-registret då processorn läser instruktionen i minnet.



Figur F.2 Kombinationen von Neumannprocessor-minne.

Utöver läsning av programinstruktioner måste processorn kunna läsa och skriva data i minnet. Därför är det lämpligt att den också innehåller ett eller flera register för adresser till data. Figur F.3 nedan visar en von Neumannprocessor och minne, där von Neumannprocessorn från figur F.2 är kompletterad med två register för adresser till data.



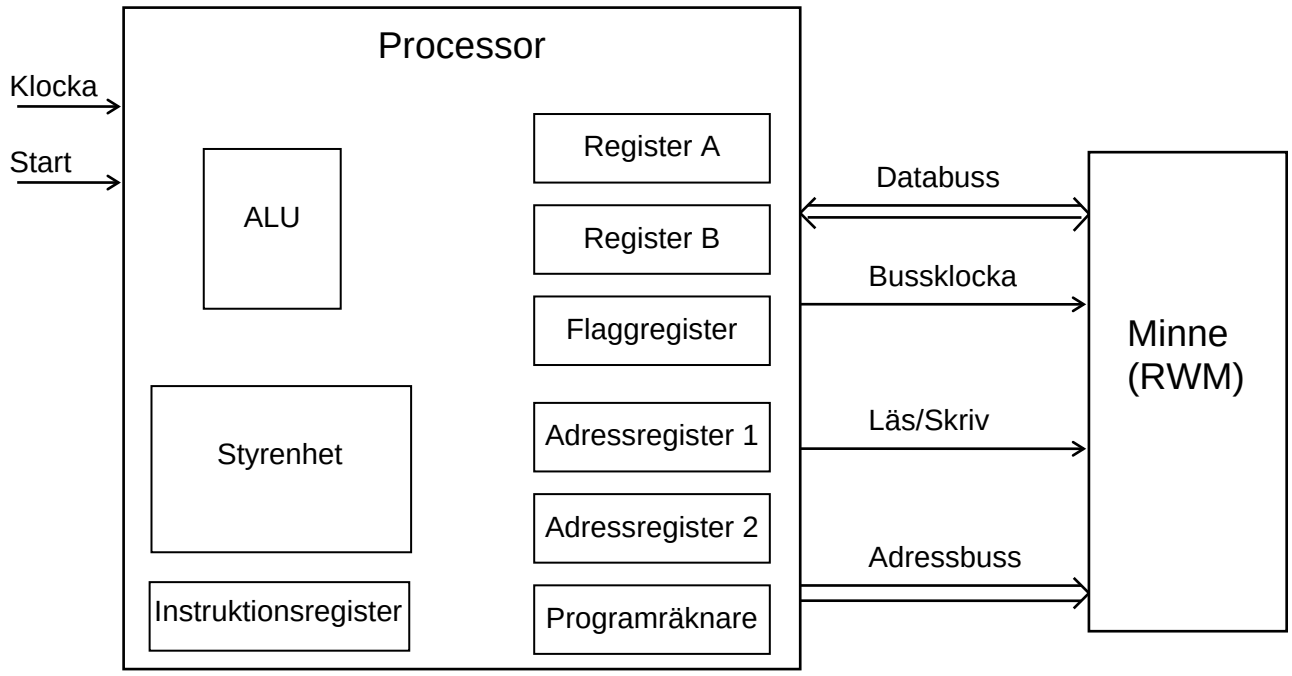
Figur F.3 Kombination av en utökad von Neumannprocessor och minne.

Kombinationen processor-minne i figurerna F.2 och F.3 kallas ofta von Neumanndator. Med ordet dator menar man egentligen ett system som består av processor, primärminne och in-/utenheter (I/O-enheter). Eftersom in-/utenheter saknas i figurerna visar de alltså inte kompletta datorer.

I vissa tillämpningar är det vanligt att man placerar program och konstanter i en minnestyp där man endast kan läsa data men inte skriva ny data. Denna minnestyp kallas läsminne (eng. read only memory, ROM). Variabler måste dock placeras i läs- och skrivbart minne (eng. read write memory, RWM), ofta oegentligt kallat "RAM", random access memory.

F.3 Programmeringsmodell för von Neumanndator

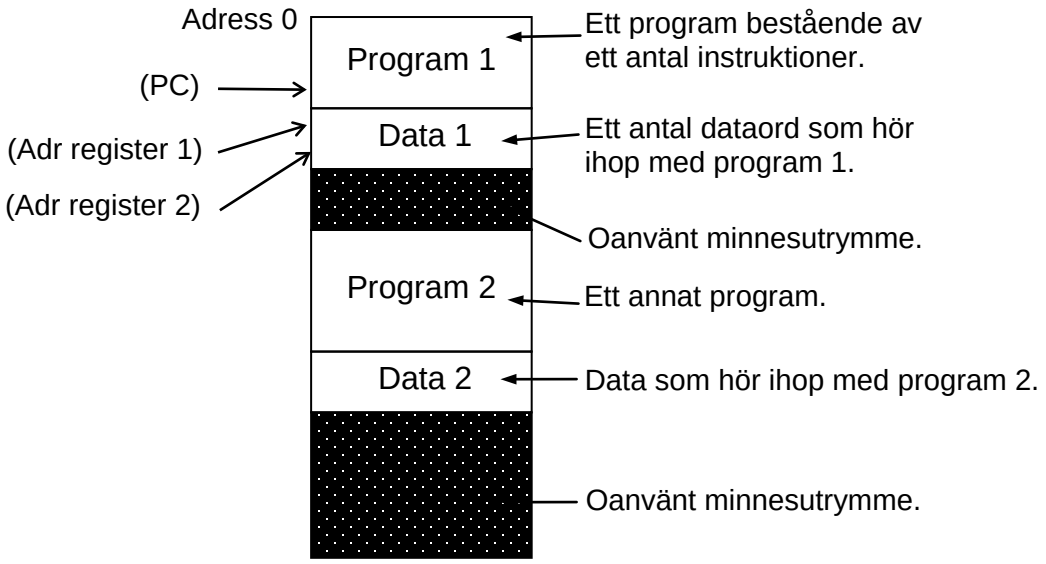
Bilden av von Neumanndatorn i figur F.3 innehåller en stor mängd detaljer som är onödiga att känna till för en person som endast skriver program för datorn. Man har därför infört en beskrivningsnivå som bara innehåller de detaljer som är av intresse för programmeraren. Figur F.4 nedan visar hur datorn i figur F.3 är uppbyggd sett med en programmerares ögon, **programmeringsmodellen**.



Figur F.4 Programmeringsmodell för enkel von Neumanndator.

I figur F.4 finns ett register med flaggbitar. Många instruktioner laddar nya värden på flaggbitarna i flaggregistret. Normalt är det flaggvärdena från ALU:n som laddas i flaggregistret vid någon ALU-operation . De innehåller alltså information om resultatet av den senaste ALU-operationen som påverkade flaggregistret. Flaggbitarna används till exempel när villkorliga instruktioner skall utföras av processorn. Villkorliga instruktioner utförs endast om det aktuella villkoret, t ex carry-flaggan = 1, är uppfyllt.

För att förenkla användandet av datorn och få program med överskådlig uppbyggnad försöker man strukturera användningen av minnet, ofta i en hierarkisk struktur. Figur F.5 visar exempel på en enkel sk minnesdisposition.



Figur F.5 Minnesdisposition för von Neumanndator.

Programräknaren PC innehåller adressen till den instruktion som står i tur att utföras. Detta kan tolkas så att PC *pekar på* ett visst ställe i minnet och betecknas som i figur F.5. Parentesen runt PC, "(PC)", anger att man menar *innehållet i* PC. Eftersom PC pekar på ett ställe i Program 1 kan man dra slutsatsen att processorn håller på att köra Program 1.

På samma sätt pekar "Adressregister 1" på en viss position i minnet där ett dataord för program 1 finns lagrat, dvs "Adressregister 1" innehåller adressen till denna minnesposition..

F.4 Ordlängd, adressbredd- och databussens bredd

Man brukar ange en processors **ordlängd** som det antal databitar ALU:n tar emot på ena dataingången eller lämnar på datautgången. Vanliga ordlängder är idag (2004) 8, 16, 32 och 64 bitar.

Databussens bredd (antal bitar) bestämmer hur många bitar man kan hämta från minnet vid varje läsning eller lagra i minnet vid varje skrivning.

I de fall instruktioner eller dataord består av fler bitar än databussens bredd måste flera läsningar göras för varje instruktion och flera läsningar eller skrivningar göras för varje dataord.

Omvänt gäller att databussen kan vara bredare än processorns ordlängd. I så fall kan processorn nå flera dataord vid varje läsning eller skrivning i minnet. Motsvarande gäller vid läsning av instruktioner i minnet. Det lästa minnesordet kan i detta fall innehålla två eller flera instruktioner.

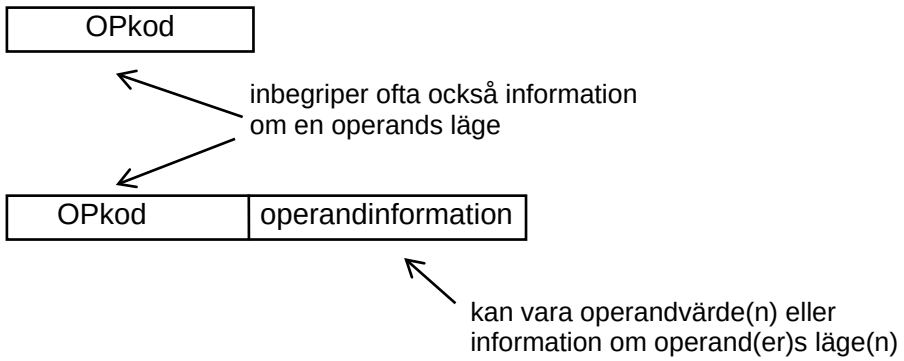
Vanliga bredder på databussen är idag 8, 16, 32 och 64 bitar.

Adressbussens bredd brukar överensstämma med programräknarens bredd (antal bitar) och bestämmer hur många olika adresser som kan användas av processorn. Mängden av samtliga adresser en processor kan adressera kallas **processorns adressrum**.

En adressbuss med 16 bitar ger t. ex. 2^{16} = 65536 olika adresskombinationer. Processorn kan då adressera 2^{16} = 65536 = 64k olika minnesord. (Vi använder konventionen att *1k = 2^{10} = 1024*.) Vanliga bredder på adressbussen är idag 16, 24 och 32 bitar.

F.5 Instruktionsformat

En instruktion innehåller den information (kodat med nollor och ettor) som behövs för att processorn skall kunna exekvera dess operation. En instruktion skall innehålla information om både operation och operand(er). Den del av instruktionen som specificerar operationen kallas operationskod, **OPkod**. Vanligen inkluderar denna kod också viss information om en operand. När så behövs ges ytterligare **operandinformation i en tilläggsdel**. Operations- och operandinformationen följer ett s k **instruktionsformat,** se figur F.6.



Figur F.6 Instruktionernas längd och innehåll följer ett sk instruktionsformat.

Sättet att koda operations- och operandinformation varierar starkt mellan processorer från olika tillverkare och ibland även mellan processorer från samma tillverkare.

F.6 Instruktionsuppsättning

En processors användbarhet bestäms av de instruktioner som den är konstruerad att exekvera. Mängden instruktioner kallas för processorns **instruktions-uppsättning** (eng. **instruction set**). Varje processortillverkare ger en detaljerad beskrivning av instruktioner och de operationer de utför i en **instruktionslista** (eng. **instruction manual**). Trots att det finns stora skillnader mellan processorer från olika tillverkare så finns det också stora likheter dem emellan vad operationer och instruktionstyper beträffar.

Varje instruktion kännetecknas av

1. **en OPkod** = ett binärt ord som i instruktionslistan ges i hexadecimal form
2. **en längd** = ett antal bytes
3. **en exekveringstid** = ett antal klockcykler
4. **en operation med åtföljande flaggpåverkan** = det instruktionen utför
5. **en adresseringsmod**, som anger hur operandläget identifieras
6. **en mnemonisk beteckning** = en av tillverkaren rekommenderad symbolisk beteckning som har anknytning till operation och adresseringsmod

Instruktionerna grupperas efter operation i kategorier. Engelska namn anges inom parentes.

1. **Dataöverföring** (**Data transfer**) — Flyttar data mellan processor och minne eller mellan interna register i processorn

   Flyttning (kopiering) av data från minnet till ett register i processorn kallas att **ladda** från minnet. (eng **LOAD**).

   Flyttning (kopiering) av data från ett register i processorn till minnet kallas att **lagra** i minnet (eng. **STORE**).

2. **Aritmetik** (**Arithmetic**) — Utför aritmetik, normalt med 2-komplementrepresentation Byter tecken på tal. Ökar eller minskar tal med 1.

3. **Logik** (**Logic**) — Utför logikoperationer som bitvis AND, OR eller XOR

4. **Test** (**Test**) — Testar och jämför dataord.

5. **Hopp** (**Jump, Branch**) — Utför ovillkorliga och villkorliga hopp i program.

6. **Andra** — Utför speciella processorberoende funktioner.

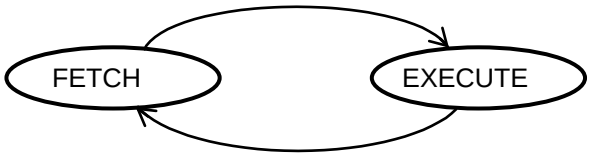
F.7 Instruktioners exekvering

En instruktion utförs i två faser.

I den första fasen, som kallas **hämtfasen** (eng. **FETCH**), hämtas (läses) operationskoden för instruktionen i minnet och lagras i instruktionsregistret I.

Hämtfasen börjar med att innehållet i programräknaren skickas ut på adressbussen och en läsning görs i minnet. Det aktuella minnesordet (operationskoden) läggs då ut på databussen av minnet, läses av processorn och placeras (laddas) i instruktionsregistret I. Under tiden ökas innehållet i programräknaren med ett eftersom man normalt kommer att använda nästa adress i minnet nästa gång programräknaren används. Därmed är hämtfasen (FETCH) slut och processorn övergår till utföradefasen.

I den andra fasen, som kallas **utförandefasen** (eng. **EXECUTE**), känner styrenheten av innehållet i instruktionsregistret I, dvs operationskoden, och alstrar en styrsignalsekvens som beror på vilken instruktion som skall utföras. När utförandefasen är slut startas nästa instruktions hämtfas enligt tillståndsgrafen i figur F.7 nedan.



Figur F.7 Tillståndsgraf för instruktionernas två faser.

F.8 FLEX-processorn

För att illustrera hur en processor är uppbyggd och arbetar har en enkel von Neumann-processor byggts vid institutionen. Den kallas FLEX för att den är flexibel i den mening att den vid behov kan omkonfigureras på flera olika sätt. FLEX-processorn är en vidareutveckling av den utökade von Neumann-processorn i figur F.3.

Programräknaren, PC, har ersatts av en laddningsbar räknare med räknevillkoret **IncPC** för ökning med 1 (inkrementering).

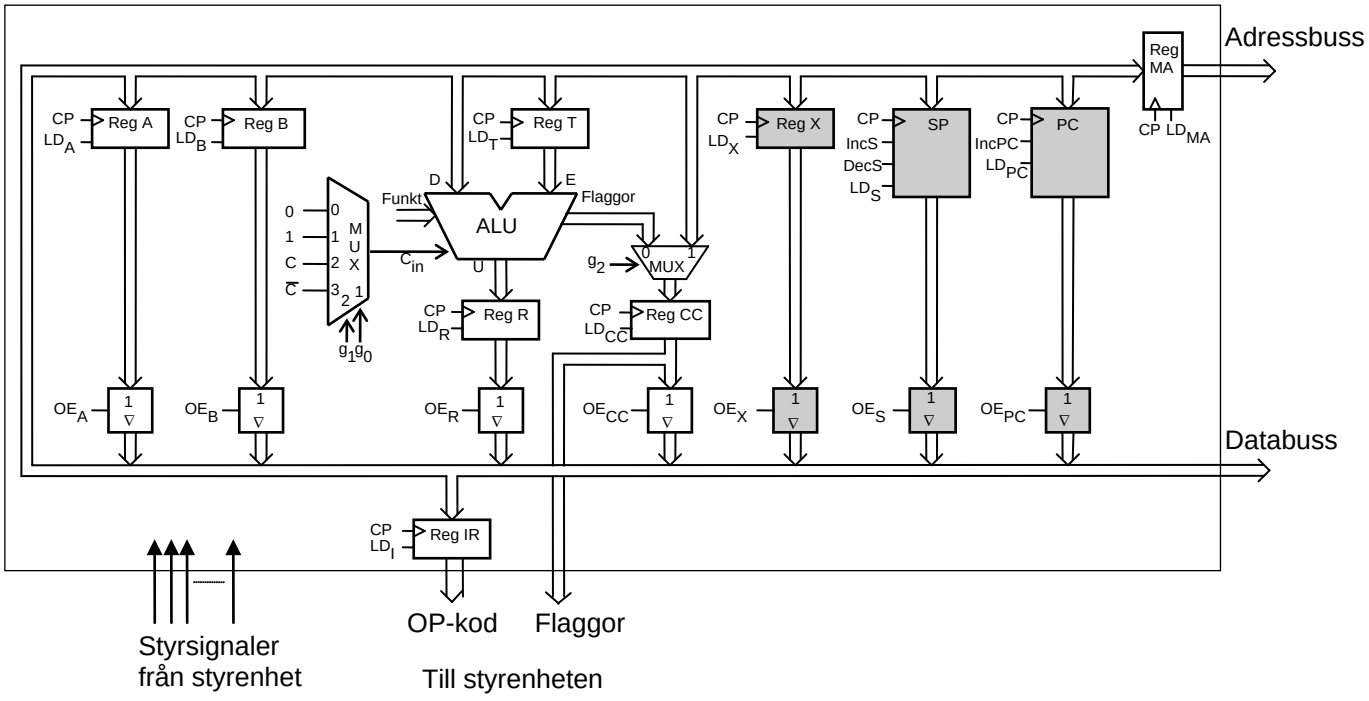
Det ena adressregistret har ersatts av en laddningsbar räknare med namnet **SP**. Denna räknare har räknevillkoret **IncSP** för ökning med 1 (inkrementering) och **DecSP** för minskning med 1 (dekrementering).

SP står här för **stackpekare**. Vad som menas med det förklaras senare.

Det andra adressregistret har fått namnet **X**.

Datavägen för FLEX-processorn ges i figur F.8.

FLEX arbetar genomgående med åttabitars ordlängd, vilket innebär att den har en åttabitars ALU, åttabitars register samt åttabitars adress- och databuss. Med åttabitars adressbuss begränsas primärminnets storlek till $2^8 = 256$ adresser.



Figur F.8 FLEX-processorns dataväg.

Ett blockschema för **styrenheten** i FLEX-processorn ges i figur F.9. Styrenheten genererar de styrsignaler som behövs i datavägen. Den genererar också styrsignalerna för läsning och skrivning i primärminnet.

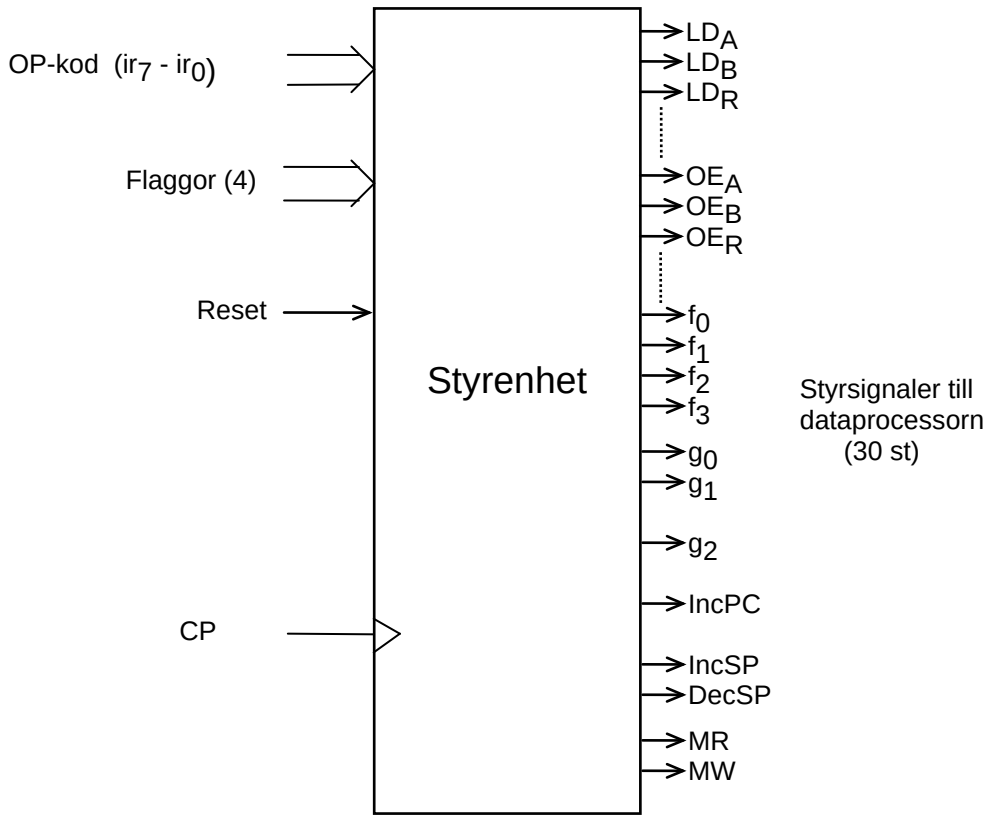
I kapitel 7 såg vi hur den databehandlade enheten kunde styras klockcykel för klockcykel genom att styrsignalerna först gavs önskade värden varpå en klockpuls CP (positiv flank) verkställer laddning av ett eller flera utvalda register. Klockpulssignalen CP bestämmer således arbetstakten.

Styrenheten utgöres av ett synkront sekvensnät med CP som klocksignal och processorns samtliga styrsignaler som utsignaler.

Styrenheten för FLEX-processorn skall kunna generera samtliga styrsignaler som behövs för att instruktionerna i dess instruktionslista skall utföras korrekt. Eftersom detta bl a medför läsningar och skrivningar i minnet måste även dessa styrsignaler genereras. Styrenheten måste alltså generera ett stort antal olika styrsignalsekvenser av varierande längd och komplexitet.

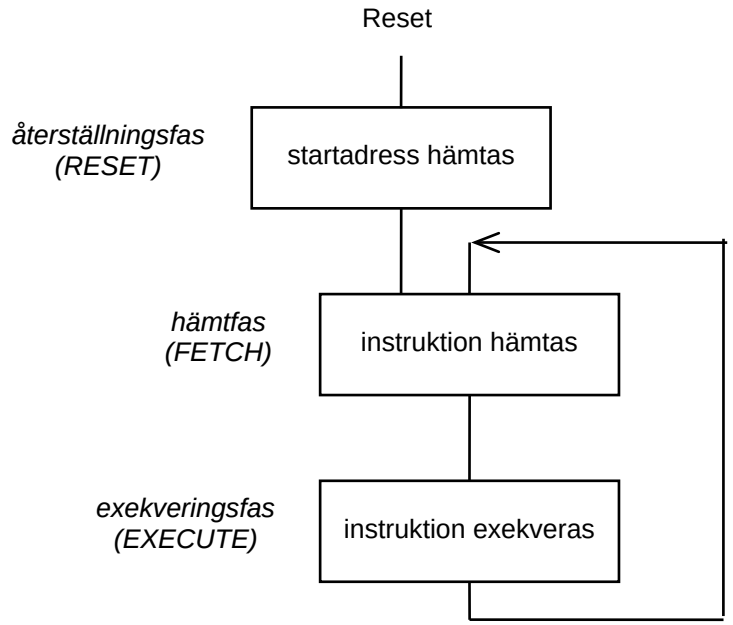
Som insignaler används dels innehållet i instruktionsregistret (OP-koden), dels flaggornas värden. OP-koden bestämmer utförandefasen och flaggornas värden kan användas för ett villkorligt vägval i exekveringen.

För att processorn skall kunna startas på ett väldefinierat sätt behövs även en startsignal Reset.



Figur F.9 FLEX-processorns styrenhet

Styrenhetens arbetssätt beskrivs bäst utifrån den tidigare omtalade instruktionscykeln. Den måste dock kompletteras med en fas för start/återstart, såsom visas i figur F.10. Den senare kallas vanligen för **återställningsfas** för att programräknaren **återställs** (eng **Reset**) till ett värde som definierar begynnelseadressen i det program som skall köras vid start/återstart.



Figur F.10 Instruktionscykeln kompletterad med s k återställningsfas.

Vi skall här inte gå in på hur styrenheten är uppbyggd men konstaterar att den är ett synkront sekvensnät och kan beskrivas med en tillståndsgraf såsom tidigare beskrivits i kapitel 5. Tillståndsgrafen visar hur växling sker mellan interna tillstånd, klockcykel för klockcykel, när instruktioner exekveras, dvs när instruktionscyklerna genomlöps.

Enligt figur F.10 sker uppstart när en startsignal (Reset) anländer till styrenheten. En återställningsfas (RESET) (ett antal tillstånd) kommer då att genomlöpas. Därefter övergår processorn till hämtfasen (FETCH). Under normalt arbete växlar den sedan ständigt mellan hämtfasen och exekveringsfasen (EXECUTE) enligt figur F.10.

För att ge en bild av hur processorn verkligen arbetar skall återställningsfasen och hämtfasen beskrivas.

Återställningsfasen (RESET)

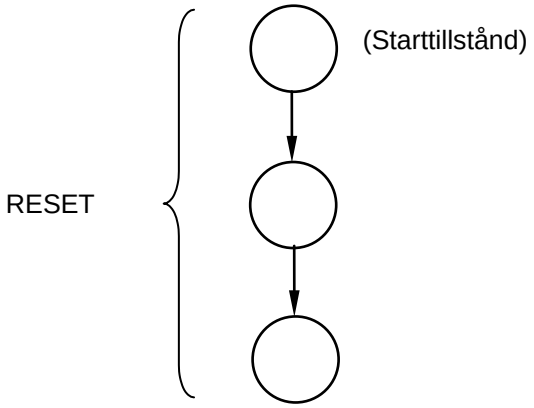
Återställning förbereds (initieras) genom att FLEX-processorns insignal Reset aktiveras. Den därpå följande klockpulsen startar återställningen som därefter sker under ett antal klockcykler. Processorn genomlöper en sekvens av tillstånd, den s k återställningssekvensen (RESET-sekvensen).

FLEX-processorns RESET-sekvens (vi kallar den ofta så) börjar med att processorn läser minnesinnehållet på den högsta adressen (FF_{16}) i adressrummet. Det lästa minnesinnehållet är den adress (8 bitar) där processorn skall hämta den första instruktionen i programmet som skall köras. Det lästa dataordet, som alltså är startadressen för ett program, placeras därför i programräknaren (PC) och processorn kan övergå till hämtfasen. Det som sker i processorn under återställningsfasen beskrivs i tabell F.1.

Tabell F.1

Klockcykel (State nr)	RTN-beskrivning	Styrsignaler	Kommentar
0	$FF_{16} \rightarrow R$	ALU-funktion = F_{16}, LD_R=1.	ALU-funktionen väljs så att talet FF_{16} finns på ALU:ns utgång. Laddingången på R-registret ettställs så att utvärdet från ALU'n (FF_{16}) laddas i R-registret vid nästa klockpuls.
1	$R \rightarrow MA$	OE_R=1 LD_{MA}=1.	Talet FF_{16} i R-registret kopplas ut på bussen. Talet FF_{16} på bussen laddas i minnesadress-registret vid nästa klockpuls.
2	$M \rightarrow PC$	MR=1, LD_{PC}=1.	Minnesinnehållet på adressen FF_{16} läses genom att minnet aktiveras för läsning. Det dataord som läses placeras i PC vid nästa klockpuls. Nästa klockcykel skall vara den första i fetchfasen.

RESET-sekvensen kan alltså utföras med tre tillstånd enligt tillståndsgrafen i figur F.11. De tre ringarna representerar var sitt (unika) tillstånd, precis som i tillståndsgraferna som beskrevs i kapitel 5. Övergång mellan ringarna verkställs av klockpulser till styrenheten. I varje ring skall vissa styrsignaler aktiveras (ettställas), vilka framgår av kolumnen styrsignaler i tabell F.1.



Figur F.11

Hämtfasen

I FLEX-processorn har alla instruktioner samma krav på hämtfasen, dvs arbetet som utförs under hämtfasen (FETCH) är detsamma för varje instruktion.

En instruktion utförs genom att OPkoden först hämtas från minnet med en läsoperation och placeras i instruktionsregistret IR under hämtfasen. Därefter inleds exekveringsfasen vars arbete styrs av den aktuella OPkoden i IR-registret.

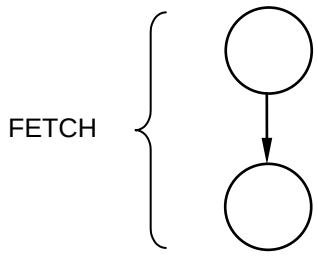
Under hämtfasen genomlöps en tillståndssekvens som inleds med en klockcykel där innehållet i programräknaren PC laddas i minnesadressregistret MA för att användas som adress vid en läsning i minnet. I samma klockcykel ökas dessutom PC-värdet med ett.

Det lästa dataordet, som är en OPkod för en instruktion, laddas av nästa klockpuls i instruktionsregistret IR och exekveringsfasen kan därefter inledas i den därpå följande klockcykeln. Skeendet under hämtfasen beskrivs i tabell F.2.

Tabell F.2

Klockcykel (State nr)	RTN-beskrivning	Styrsignaler	Kommentar
0	PC→MA, PC+1→PC	OE_{PC}=1, LD_{MA}=1, IncPC=1.	Adressen för nästa instruktions operationskod kopieras från PC till minnesadressregistret MA. Adressen som finns i PC ökas med ett.
1	M→IR	MR=1, LD_I=1.	Läs operationskoden från minnet. Placera den i instruktionsregistret IR. Nästa klockcykel skall vara den första i executefasen.

Hämtfasen kan alltså utföras med två tillstånd enligt figur F.12. När styrenheten "befinner sig" i den nedre av de två ringarna kommer den att övergå från FETCH till EXECUTE då nästa klockpuls kommer.

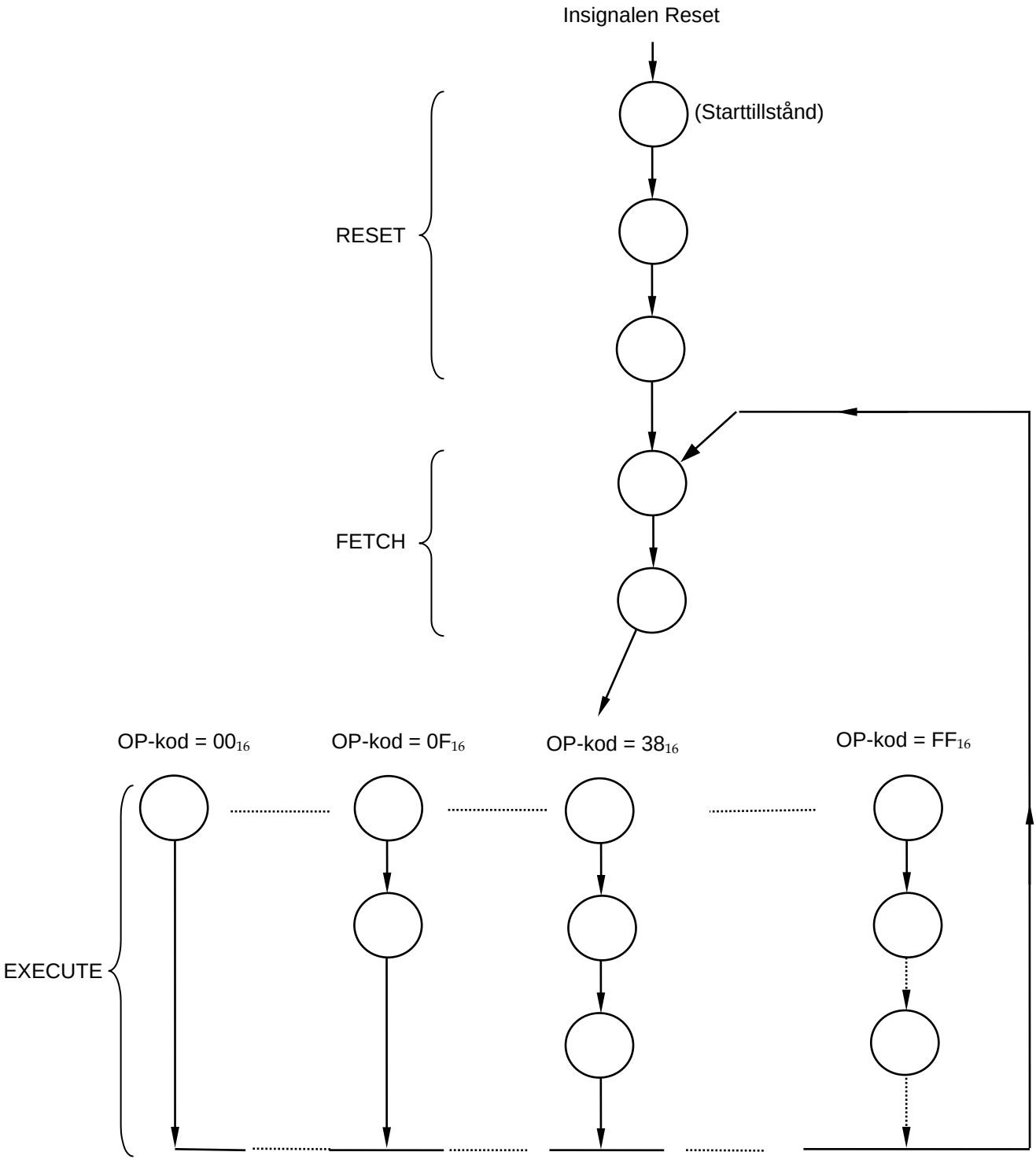


Figur F.12

Tillståndsgraf för styrenheten

Under EXECUTE bestäms styrenhetens beteende av OP-koden i instruktionsregistret IR och eventuellt av flaggornas värden. FLEX-processorns OP-kod består av åtta bitar och kan därför ha 2^8= 256 olika värden. Den representerar alltså 256 möjliga EXECUTE-faser.

Övergången från FETCH-fasen till någon av EXECUTE-faserna illustreras i tillståndsgrafen i figur F.13. Där visas RESET-sekvensen, FETCH-sekvensen och EXECUTE-sekvenserna för OP-koderna 00_{16} t o m FF_{16}.



Figur F.13 Tillståndsgraf för styrenheten till FLEX-processorn.

I tillståndsgrafen skall egentligen också finnas övergångar (pilar) från varje ring till den översta ringen, dvs starttillståndet. Denna typ av övergång skall ske när styrenhetens insignal Reset = 1.

Processorns tillståndsgraf kommer att studeras mer i detalj senare.

F.9 Instruktionsuppsättning för FLIS-processorn

Observera att vi här använder FLIS-processorn istället för FLEX-processorn!

En processors instruktionsuppsättning är sammansatt av instruktioner som behövs för att utföra uppgifter av generell typ. Ett maskinprogram skall kunna konstrueras utgående från en algoritmisk beskrivning av hur en uppgift utförs. Maskinprogrammet i sig är också en algoritmisk beskrivning. Processorn skall därför ha instruktioner som kan uttrycka konstruktionerna **sekvens**, **selektion** och **iteration** vilka behövs i algoritmer.

Instruktionsuppsättningen är ofta ganska primitiv. I enkla processorer saknas ex vis ofta instruktioner för operationer som multiplikation och division. Lite mer komplexa operationer implementeras då istället som följder av processorns primitiva operationer. Alla **operationer** som erbjuds framgår av instruktionsuppsättningen som visas i processorns instruktionslista. Merparten av instruktionerna tillhör någon av följande kategorier

* flyttning av data
* bearbetning av data, aritmetik och logik
* tester
* hopp

De två första används för att forma sekvenser, dvs göra tilldelningar och beräkna uttryck och de två sista används huvudsakligen för att uttrycka selektioner och iterationer.

Alla kategorierna är på något sätt beroende av adresser. De tre första kategorierna hanterar operander och måste på något sätt specificera lägen för operander och resultat. Dessa lägen kan vara interna processorregister eller externa adresser i primärminnet. I den sista operationskategorin måste den adress specificeras på vilken programexekveringen efter ett hopp skall fortsätta.

Programmeraren måste ges en god bild av hur ett operandläge eller en hoppadress kan specificeras via instruktioner. Specificeringen kan ske på ett flertal olika sätt. Processorn sägs ha olika **adresseringsmoder** (eng **addressing modes**). Dessa framgår också av instruktionslistan. I texten definieras och diskuteras adresseringsmoderna när det är lämpligt.

I detta avsnitt beskrivs både operationer och adresseringsmoder som är typiska för en enkel processors instruktionsuppsättning, samt hur de kan användas. Operationerna beskrivs med RTN och namnges med gängse engelska benämningar. Detta görs huvudsakligen utifrån ovanstående kategoriindelning.

De instruktioner (maskininstruktionerna) som processorn hämtar från minnet (primärminnet) består av enbart nollor och ettor. För att vi människor lättare skall kunna läsa och förstå maskinprogram har man infört sk **assemblerspråk** (eng **assembly language**), där varje instruktion representeras av en bokstavsförkortning av dess funktion, en sk **mnemonisk beteckning**.

Processortillverkaren Motorola (numera Freescale) har definierat assemblerspråk för sina kommersiella mikroprocessorer, som kommer att användas i en senare kurs. Vi har därför använt samma skrivsätt för FLIS-processorns instruktioner.

Då tal skall anges i assemblerinstruktionerna kan man välja att använda det hexadecimala, binära eller decimala talsystemet. Valet av talsystem framgår av ett prefix till talsiffrorna enligt uppställningen nedan:

$hexadecimala talsiffror = hexadecimalt
%binära talsiffror = binärt
decimala talsiffror = decimalt (inget prefix)

Instruktioner för flyttning av data

Såvida inget annat anges är flyttning av data detsamma som kopiering av data. I instruktionsuppsättningen finns instruktioner för att flytta data **mellan interna register**

$r_{src} \rightarrow r_{dst}$	transfer
$r_1 \leftrightarrow r_2$	exchange (inbördes byte)

och **mellan interna register och register i minnet (M, primärminnet)**

$r_{src} \rightarrow M$	store
$M \rightarrow r_{dst}$	load

Med en "transfer"-instruktion flyttas ett ord ifrån ett av processorns register till ett annat. Det register varifrån data hämtas kallas allmänt för **källregister, r_{src}** (eng. **source register**) och det register data flyttas till kallas **destinationsregister, r_{dst}** (eng. **destination register**). De register som instruktionen skall använda måste specificeras i maskininstruktionen.

Exempel F.1

Av instruktionslistan framgår att FLISP har flera "transfer"- och "exchange"-instruktioner.

Instruktionen TFR X,Y kopierar värdet i register X och placerar det i register Y. Instruktionen utgörs av en byte, som är Opkoden.

maskininstruktion	operation
1A (OPkod)	$X \rightarrow Y$

Med "store"- och "load"-instruktionerna lagras resp. hämtas ett operandvärde på en adress i minnet. Dessa instruktioner finns i olika varianter beroende på hur operandadressen specificeras.

En variant är att explicit (direkt) ge operandadressen i instruktionen. I maskinspråket är i dessa fall den verkliga (absoluta) adressen till destinationen resp. källan placerad i instruktionen, efter OPkoden. Genom en sådan instruktion tvingas programmet att alltid använda den givna adressen. Man kan säga att programmet använder sig av ett variabelvärde som finns på en bestämd adress. Adresseringsmoden kallas ibland **absolut** (eng **absolute**) och ibland **direkt** (eng. **direct**).

Exempel F.2

En FLISP-instruktion med absolut (absolute) adressering är STA $AB som i maskinspråket har utseendet

maskininstruktion	operation
E1 (OPkod)	
AB (operandadress)	$A \rightarrow M(AB_{16})$

Här är det OPkoden $E1_{16}$ som anger att A-registret är källregister och att den följande byten, AB_{16}, utgör adressen till operanden. Det interna registret som instruktionen använder sig av specificeras således i OPkoden.

$-tecknet används för att ange att adressen är på hexadecimal form.

I en variant av "load"-instruktionen kan operandvärdet ingå i instruktionen. Det adresseras då av programräknaren och hämtas omedelbart in till processorn i samband med att instruktionen hämtas. Adresseringsmoden kallas **omedelbar** (eng **immediate**). Denna instruktion används när operandvärdet är en konstant. Konstanter används ju ofta vid beräkning av uttryck.

Exempel F.3

Hos FLISP anger #-tecknet omedelbar (immediate) adressering, dvs att operanden följer omedelbart i instruktionen. Instruktionen LDA #$35 har i maskinspråket utseendet

maskininstruktion	operation
F0 (OPkod)	
35 (operand)	$35_{16} \rightarrow A$

Här är det OPkoden $F0_{16}$ som anger att A-registret är destinationsregister och att den efterföljande byten 35_{16} är en operand.

För FLISP finns det ingen "store"-instruktion med omedelbar adressering.

En processor har vanligen också "store"- och "load"-instruktioner med andra adresseringsmoder, dvs där operander specificeras på andra sätt. Dessa beskrivs senare.

Bearbetning av data, aritmetik och logik

I detta delavsnitt beskrivs de instruktioner som utför aritmetik- resp. logikoperationer. De delas in efter om operationen är unär eller binär. Tillsammans med instruktioner för dataflyttning används de i hög grad för beräkning av uttryck. De kan använda sig av operander i form av konstanter eller variabelvärden. Vid användning av konstanter kan adresseringen vara **omedelbar** (eng. **immediate**). Vid användning av variabelvärden är adressering ordnad enligt någon annan adresseringsmod.

Unära operationer

Unära operationer arbetar endast på en operand. Operationen består i en bearbetning (manipulation) av operanden. Hos varje processor finns ett antal operationer av denna natur. Läget för operanden kan antingen vara ackumulator A eller en adress i minnet. Läget indikeras tillsammans med den mnemoniska beteckningen, här generellt betecknad MNE, på följande sätt

```
f(A) → r        operation,          MNEA
f(M) → M            ”               MNE   operandinfo
```

där A är beteckningen på ackumulator A och M är beteckningen på en adress i minnet.

I det första fallet, där operationen arbetar på innehållet i ett register, specificeras registret **inne i** instruktionens OPkod. På engelska sägs då specificeringen vara "inherent" i OPkoden och följaktligen kallas denna adresseringsmod **"inherent".**

I det senare fallet kan adresseringsmoden vara omedelbar, direkt (absolut) eller någon av de minnesrelaterande moder som beskrivs senare i kapitlet. Adressen bildas med ledning av den operandinfo som följer med instruktionen. Man kan i dessa fall få intrycket att processorn kan bearbeta ord utanför processorn, direkt i minnet. Detta är förstås ej möjligt utan operanden hämtas in till processorn, bearbetas och återplaceras i minnet med en och samma instruktion. Här följer en beskrivning av de vanligaste unära operationerna.

Inkrementering och dekrementering av operand

```
A+1 → A       increment,        INCA
M+1 → M                         INC   operandinfo

A-1 → A       decrement         DECA
M-1 → M                         DEC   operandinfo
```

Ofta används innehållet i ackumulator A eller på en adress som ett ”räknarvärde”, som skall användas på något sätt under ett programs gång. Ett sådant värde kan med dessa instruktioner ökas med 1, inkrementeras, eller minskas med 1, dekrementeras, när så önskas.

Exempel F.4

I FLISP inkrementeras innehållet i ackumulator A med instruktionen

	maskininstruktion	operation
INCA	07 (OPkod)	A+1 → A

Exempel F.5

Innehållet på adress 12_{16} inkrementeras med FLISP-instruktionen

		maskininstruktion	operation
INC	$12	37 (OPkod) 12 (operand adress)	$M(12_{16})+1 \rightarrow M(12_{16})$

Nollställning av operand

0 → A	clear,	CLRA
0 → M		CLR operandinfo

Invertering av operands bitar

$\overline{A} \rightarrow A$	complement,	COMA
$\overline{M} \rightarrow M$		COM operandinfo

Den senare operationen används dels som en ren logikoperation för att bitvis invertera (komplementera) en operands bitar, dels tillsammans med "clear"-operationen för att ettställa alla bitarna i en operand.

Exempel F.6

Ackumulator A:s bitar inverteras med FLISP-instruktionen

	maskininstruktion	operation
COMA	0A (OPkod)	$\overline{A} \rightarrow A$

vilket innebär att om innehållet i A är 01000011 så ändrar instruktionen ackumulatorns innehåll till 10111100.

Exempel F.7

Hos FLISP kan alla bitarna i A-ackumulatorn ettställas med instruktionssekvensen

```
  .
CLRA
COMA
  .
```

Negering (tvåkomplementering) av en operand

$0 - A = 0 + \overline{A} + 1 \rightarrow A$	negate,	NEGA
$0 - M = 0 + \overline{M} + 1$		NEG operandinfo

Operationen används i aritmetik med tal på tvåkomplementform för att byta tecken på en operand.

Exempel F.8

Talet på adress 45_{16} tvåkomplementeras med instruktionen

	maskininstruktion	operation
NEG $45	36 (OPkod) 45 (operandadress)	$0 - M(45_{16}) \rightarrow M(45_{16})$

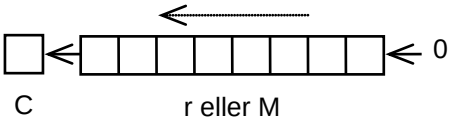
Om $M(45_{16})$ = 01110100 så ändrar instruktionen detta till 10001100.

Exempel F.9

Konstanten $(-32)_{10}$ på 2-komplementrepresentation placeras på adress $D1_{16}$ med instruktionssekvensen

```
  .
LDA  #32
NEGA
STA  $D1
  .
```

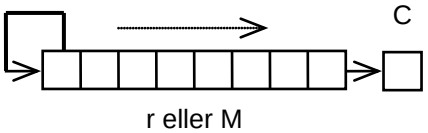
Aritmetiskt skift åt vänster



arithmetic shift left, ASLA
ASL opinfo

Denna instruktion skiftar operandens bitar ett steg åt vänster på det sätt som visas i operationsfiguren. Den används i aritmetik med tal på tvåkomplementform för att multiplicera operanden med 2. Detta illustreras i exemplet på nästa sida.

Aritmetiskt skift åt höger



arithmetic shift right, ASRA
ASR opinfo

Denna instruktion skiftar operandens bitar ett steg åt höger på det sätt som visas i operationsfiguren. Den används i aritmetik med tal på tvåkomplementform för att dividera operanden med 2.

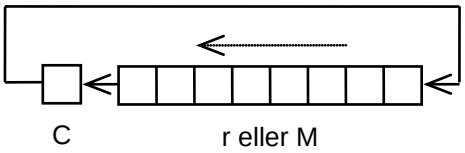
Exempel F.10

Utgå ifrån talet W = $(+25)_{10}$ = $(00011001)_2$ och iakttag hur W och -W ändras när de multipliceras med 2. Binärpunkten antas vara placerad till höger om bit 0.

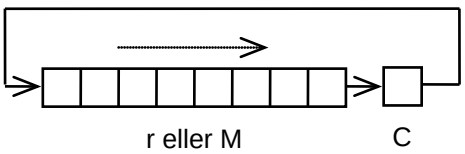
```
                dec         bin
positivt tal
     W =        25       00011001

     2W =       50       000110010
                                  ↖ skiftas in i LSB
                ↙
              hamnar i C-flaggan
negativt tal
     -W =       -25      11100111

     -2W =      -50      111001110
                                  ↖ skiftas in i LSB
                ↙
              hamnar i C-flaggan
```

Man löper alltid en risk att tappa signifikanta siffror vid dubblering.

Skift vid rotation



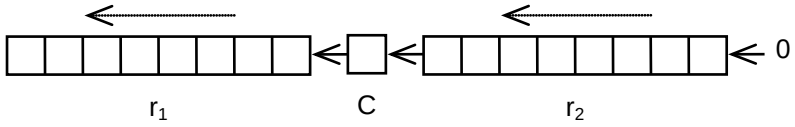

```
rotate left,    ROLA
                ROL opinfo
```




```
rotate right,   RORA
                ROR opinfo
```

Vid rotationsskift åt vänster skiftas operandens bitar ett steg åt vänster. Operandens två ändar sammanlänkas med C-flaggan. Detta innebär att C-flaggans värde skiftas in i en operandända och att den bit som skiftas ut i den andra ändan placeras i C-flaggan. Genom 9 likadana rotationsinstruktioner kan således bitmönstret i C-flaggan + operanden roteras (cirkuleras) ett varv.

Rotationsinstruktionerna kan med fördel användas tillsammans med andra skiftinstruktioner för att ta hand om de bitar som dessa skiftar ut ur en operand. Man kan på så vis göra en aritmetisk operandutvidgning från 8 till 16 bitar enligt den idé som ges i figur F.14. (Minst ett register antas finnas i minnet.) De kan också användas för att separera en operand i flera delar och placera dem på olika ställen.



$ASLr_2 + ROLr_1$

Figur F.14 Operandutvidgning åt vänster.

Binära operationer

Eftersom binära operationer arbetar på två operander borde tre lägen (två källor och en destination) vara förknippade med dessa. Så är emellertid ej fallet hos de vanliga processorerna, ty destinationen för resultatet är vanligen densamma som källan för en av operanderna. Hos FLISP är detta läge normalt ackmulatorn, A, men kan också vara flaggregistret. De binära operationerna betecknas med RTN och mnemonik på följande sätt

```
r∗M          operation,          MNEr   operandinfo
```

där som förut r är beteckningen på det interna registret och M är beteckningen på adressen i minnet. Den andra operanden finns således alltid i processorns minne. Dess adresseringsmod kan vara omedelbar, direkt (absolut) eller någon av de som beskrivs senare.

Aritmetikinstruktioner

```
A+M → A     add,            ADDA  operandinfo
```

Dessa additionsinstruktioner utför additionen P=D + E på följande sätt

$$
\begin{array}{ll}
 & c_8c_7c_6c_5c_4c_3c_2c_1c_0 \\
 & \quad d_7d_6d_5d_4d_3d_2d_1d_0 \\
 & +\ e_7e_6e_5e_4e_3e_2e_1e_0 \\
\hline
\text{carry} = c_8 & \quad p_7p_6p_5p_4p_3p_2p_1p_0
\end{array}
$$

vilket lämpar sig väl för addition av två 8 bitars tal med eller utan tecken. För tal med tecken skall 2-komplementrepresentation användas.

```
A-M → A     subtract,       SUBA  operandinfo
```

Det är känt att subtraktion utförs genom addition. Subtraktionsinstruktioner utför subtraktionen P=D - E på följande sätt

$$
\begin{array}{l}
c_8\,c_7\ c_6\ c_5\ c_4\ c_3\ c_2\ c_1\ 1 \\
\quad d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0 \\
+\ e_7'e_6'e_5'e_4'e_3'e_2'e_1'e_0' \\
\hline
\quad p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0
\end{array}
\qquad \text{svarar mot} \qquad
\begin{array}{l}
b_8\,b_7b_6b_5b_4b_3b_2b_1b_0 \\
\quad d_7d_6d_5d_4d_3d_2d_1d_0 \\
-\ e_7e_6e_5e_4e_3e_2e_1e_0 \\
\hline
\quad p_7p_6p_5p_4p_3p_2p_1p_0
\end{array}
$$

Det skall vid subtraktioner observeras att det är l**ånesiffran** (eng. **borrow**) b_8 som placeras i C-flaggan. Mellan b_8 och c_8 i uppställningen gäller att

$$b_8 = c_8'$$

Tänk igenom varför!

Exempel F.11

Med FLISP adderas talet på adress 20_{16} till talet i ackumulator A med instruktionen

		maskininstruktion	operation
ADDA	$20	A6 (OPkod) 20 (operandadress)	A + M(20_{16}) → A

Exempel F.12

Talet på adress 24_{16} subtraheras ifrån talet i ackumulator A med FLISP-instruktionen

		maskininstruktion	operation
SUBA	$24	A4 (OPkod) 24 (operandadress)	A - M(24_{16}) → A

I många fall måste operander uttryckas med fler än 8 bitar. Vid användning av 8 bitars processorer uttrycks de då med 16, 24 eller någon annan multipel av 8 bitar. Man säger att operanden har **dubbelprecision, trippelprecision** eller **multipelprecision**. Vid aritmetik för sådana operander måste då bearbetningen delas upp i en följd av 8 bitars operationer. För att klara av detta har processorn de speciella additions- och subtraktionsinstruktionerna

```
A+M+C → A    add with carry,         ADCA  operandinfo
A-M-C → A    subtract with borrow,   SBCA  operandinfo
```

Dessa används på följande sätt:

Additionen, P = D + E, där D och E är 16 bitars ord med eller utan tecken, delas då upp i två delar enligt

$$\begin{array}{rl} c_{16} & c_{15}c_{14}c_{13}c_{12}c_{11}c_{10}c_9c_8 \\ & d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8 \\ + & e_{15}e_{14}e_{13}e_{12}e_{11}e_{10}e_9e_8 \\ \hline & p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8 \end{array} \qquad \begin{array}{rl} c_8 & c_7c_6c_5c_4c_3c_2c_1c_0 \\ & d_7d_6d_5d_4d_3d_2d_1d_0 \\ + & e_7e_6e_5e_4e_3e_2e_1e_0 \\ \hline & p_7p_6p_5p_4p_3p_2p_1p_0 \end{array}$$

I denna adderas först operandernas minst signifikanta bytes,

PL = DL + EL

och därefter deras mest signifikanta bytes tillsammans med den minnessiffra, c_8, som den första additionen producerat

PH = DH + EH + c_8

Denna senare addition kan således utföras med en "add with carry"-instruktion, eftersom c_8 efter den första additionen hamnar i C-flaggan.

På liknande sätt delas subtraktionen, P = D - E, upp i två delar. Även om det är känt att den utförs som addition av tvåkomplementet av E, så illustreras här den subtraktion som additionen utför

$$\begin{array}{rl} b_{16} & b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8 \\ & d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8 \\ - & e_{15}e_{14}e_{13}e_{12}e_{11}e_{10}e_9e_8 \\ \hline & p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8 \end{array} \qquad \begin{array}{rl} b_8 & b_7b_6b_5b_4b_3b_2b_1b_0 \\ & d_7d_6d_5d_4d_3d_2d_1d_0 \\ - & e_7e_6e_5e_4e_3e_2e_1e_0 \\ \hline & p_7p_6p_5p_4p_3p_2p_1p_0 \end{array}$$

där operandernas minst signifikanta bytes behandlas först

PL = DL - EL

och deras mest signifikanta bytes därefter

PH = DH - EH - b_8

Av detta inses att den senare subtraktionen kan utföras med en "subtract with borrow"-instruktion, eftersom lånesiffran efter den första subtraktionen b_8 finns i C-flaggan.

Exempel F.13

Med FLISP kan 16-bitars talet på adresserna 11_{16} (MSB) och 12_{16} (LSB) subtraheras från 16-bitars talet på adresserna 13_{16} (MSB) och 14_{16} (LSB) med instruktionssekvensen

```
        .
LDA  $14    Hämta låg byte
SUBA $12    Här hamnar lånesiffran i C
STA  $14    Spara låg byte. C påverkas ej
LDA  $13    Hämta hög byte.    – ” -
SBCA $11    Här finns lånesiffran kvar i C
STA  $13    Spara hög byte
        .
```

Logikinstruktioner

Processorn har instruktioner för att bitvis bilda AND, OR och EXCLUSIVE-OR mellan operandernas bitar. Hur detta görs skall beskrivs i följande uppställning

$$\begin{array}{rll} & d_7d_6d_5d_4d_3d_2d_1d_0 & \text{operand 1} \\ \text{operation} & *e_7e_6e_5e_4e_3e_2e_1e_0 & \text{operand 2} \\ & p_7p_6p_5p_4p_3p_2p_1p_0 & \text{producerat resultat} \end{array}$$

där * svarar mot operationen. När en operation utförs bitvis finns ingen koppling i sidled mellan operandernas bitar utan varje resultatbit är enbart en funktion av motsvarande bitar i de två operanderna

$p_i = d_i * e_i$

Logikinstruktionerna spelar en stor roll i mikrodatortekniken, ty man kan med dem förändra enstaka bitar i en operand. Fortsättningsvis skall logikinstruktionernas huvudsakliga användning beskrivas.

A∧M → A and, ANDA operandinfo

Med OCH-operationen kan man selektivt nollställa enstaka bitar i en operand såsom framgår av uppställningen

	$d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0$	operand
$\wedge$	0 0 1 1 1 1 0 0	mask = nollställande bitmönster
	0 0 $d_5\ d_4\ d_3\ d_2$ 0 0	resultat

Man säger att operationen maskerar de bitar som nollställs och frigör de övriga för fortsatt användning.

$A \vee M \rightarrow A$ or, ORA operandinfo

På samma sätt kan man med ELLER-operationen selektivt ettställa enstaka bitar i en operand och på så sätt maskera deras gamla värden

	$d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0$	operand
$\vee$	0 0 0 0 1 1 1 1	mask = ettställande bitmönster
	$d_7\ d_6\ d_5\ d_4$ 1 1 1 1	resultat

Exempel F.14

Maskering med AND- och OR-instruktioner illustreras med en FLISP-instruktionssekvens, som undersöker bit 7 (teckenbiten) på adress 31_{16}.
Om den är 1 så ettställs bit 4 på adress 32_{16} och om den är noll så nollställs bit 4 på adress 32_{16}.

```
Adr
 .      .
 11     LDA  $31          Hämta innehållet på adress $31
 13     ANDA #%10000000   Här maskas bit7 fram i ackumulator A
 15     BMI  $1D          Om bit7=1 fortsätter man på adress $1D
 17     LDA  #%11101111   Man ser här att bit4 = 0, övriga = 1
 19     ANDA $32          Här nollställs bit4 i ackumulator A
 1B     JMP  $21          Här fortsätter man på adress $21
 1D     LDA  #%00010000   Man ser här att bit4 = 1, övriga = 0
 1F     ORA  $32          Här ettställs bit4 i ackumulator A
 21     STA  $32          Spara ackumulator A på adress $32
 23      .
  .      .
  .      .
```

$A \oplus M \rightarrow A$ exclusive or, EORA operandinfo

Med denna operation kan man selektivt invertera enstaka bitar i en operand

	$d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0$	operand
$\oplus$	1 1 1 1 0 0 0 0	bitmönster för selektiv invertering
	$d_7'd_6'd_5'd_4'\ d_3\ d_2\ d_1\ d_0$	resultat

Exempel F.15

Följande instruktionssekvens för FLISP inverterar bit 2 av adress $F1_{16}$

```
        .
LDA   $F1
EORA  #$04   Här är bit2 = 1
STA   $F1
        .
```

Tester

Selektions- och iterationskonstruktioner är ofta baserade på villkor. På processornivå bildas dessa typiskt genom undersökning av en operand eller som resultatet av ett beräknat uttryck. För att undersöka en operand har processorn en eller flera testinstruktioner. Dessa operationer har vanligtvis två operander, en som skall undersökas och en annan som den testas mot för att undersökningen skall ge resultat. Tre typer av testinstruktioner är vanliga. Dessa kännetecknas av att de endast påverkar flaggregistret, dvs ger flaggpåverkan.

Jämförelse med operand

`A-M ⇒ flaggpåverkan  compare,  CMPA  operandinfo`

är en operation som använder subtraktion för att avgöra om en operand är mindre än, lika med eller större än en annan operand. Svaret finns i flaggregistret. Skillnaden bevaras ej.

Exempel F.16

Om ackumulator A innehåller det uppmätta temperaturvärdet 12_{16} (18 °C) så utför instruktionen CMPB #20 operationen

```
  00010010
- 00010100
  11111110   producerat resultat
```

med flaggvärdestilldelningen

C=1 V=0 Z=0 N=1

Den jämför således det uppmätta värdet med ett känt referensvärde 20 °C. Detta kan göras för att t ex starta uppvärmning när N-flaggan blir "1".

Jämförelse med noll

```
A-0 ⇒ flaggpåverkan   test,        TSTA
M-0 ⇒ flaggpåverkan                TST   operandinfo
```

är en operation som subtraherar noll ifrån operanden för att avgöra om den är noll, positiv eller negativ. Svaret finns i flaggregistret.

Exempel F.17

Om adress 98_{16} innehåller värdet 45_{16} så utför instruktionen TST $98 operationen

```
  01000101
- 00000000
  01000101   producerat resultat
```

och ger Z- och N-flaggorna värdena

Z:=0 N:=0

utan att påverka något annat registers innehåll.

I båda dessa jämförelseoperationer betraktas operanden som ett numeriskt värde som jämförs med ett testobjekt. Relationerna är beroende av om operand och testobjekt ses som tal med eller utan tecken. Helt allmänt gäller att relationen mellan operand och jämförelseobjekt kan uttryckas med flaggor enligt tabell F.3. Observera att när tal utan tecken jämförs med 0 så behövs bara relationerna = och ≠ .

Tabell F.3

relation	tal utan tecken	tal med tecken
>	C'·Z'	(N⊕V)'·Z'
≥	C'	(N⊕V)'
=	Z	Z
≠	Z'	Z'
≤	(C'·Z')' = C+Z	((N⊕V)'·Z')' = (N⊕V)+Z
<	C	(N⊕V)

Observera att flaggvärdena antas vara resultat av en subtraktion och att låne-biten b_8 finns i C-flaggan. Vid tal utan tecken avgörs relationerna av C- och Z-flaggorna.

Tal med tecken har 2-komplementrepresentation och tillhör intervallet [-128, +127]. Eftersom talen nu har teckenbit måste flaggorna N, V och Z användas för att bestämma storleksrelationerna.

Vi vet att när "overflow" inträffar vid subtraktion upptäcks detta genom att teckenbiten N får fel värde. Trots detta kan man skapa en korrekt teckenbit, som gäller vare sig "overflow" inträffar eller ej, genom att bilda N⊕V. Om overflow inträffar vet man ju att teckenbiten N får fel värde. Eftersom V samtidigt får värdet 1 inverterar man N genom att bilda N⊕V (N⊕1 = N'). Om overflow inte inträffar ger N ⊕ V värdet N eftersom N ⊕ 0 = N.

Tänk igenom hur uttrycken verkligen representerar respektive relation!

I den tredje testoperationen betraktas operanden ej som ett värde utan enbart som ett bitmönster.

Test av operandbitar

```
A∧M ⇒ flaggpåverkan  bit test,     BITA  operandinfo
```

är en operation som utför OCH-operationen mellan bitarna i en operand och bitarna i ett testmönster, en mask. Testet resulterar enbart i påverkan av N- och Z-flaggorna. Detta är huvudsakligen ett sätt att med en mask ta reda på om alla bitarna i den icke maskerade bitgruppen av operanden är noll eller ej. Vanligen används denna operation för att undersöka en av operandens bitar genom att maskera de övriga bitarna. Denna operation är således ett enklare och bättre sätt att undersöka individuella bitar än att göra det med skiftoperationer och C-flaggan.

Exempel F.18

Bit 3 av ackumulator A undersöks med BITA #$08

```
          d7 d6 d5 d4 d3 d2 d1 d0      operand i A
      ∧   0  0  0  0  1  0  0  0       mask
          0  0  0  0  x3 0  0  0       resultat hamnar ej i A
```

N-flaggan får värdet

N = 0

Z- flaggan får värdet

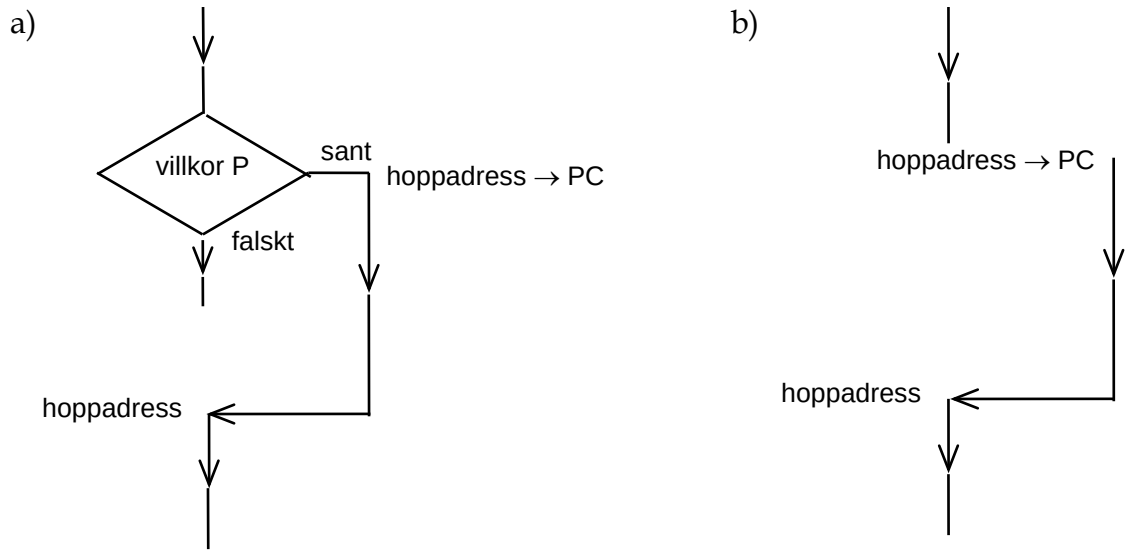
Z = 1 om $x_3 = 0$

eller

Z = 0 om $x_3 = 1$

Hopp i program

Denna kategori av instruktioner är också oerhört viktig för selektions- och iterationskonstruktioner. I båda dessa konstruktionstyper måste **den raka exekveringsföljden** i en sekvensen kunna brytas. I ett maskinprogram görs detta med **hopp**. Hopp sker genom att programräknaren inte ökas till nästa adress i en sekvens utan laddas med en ny adress. Ett hopp kan vara **villkorligt** (eng. **conditional**) eller **ovillkorligt** (eng. **unconditional**), se figur F.15.



Figur F.15 Exekveringsföljd vid a) villkorliga resp. b) ovillkorliga hopp.

Av denna ser man att den raka exekveringsföljden alltid bryts vid ett ovillkorligt hopp, då programräknaren laddas med en hoppadress och exekveringen fortsätter med början på denna. Instruktioner för ovillkorliga hopp benämns vanligen **"jump"-instruktioner**.

Vid villkorliga hopp bryts den raka exekveringsföljden endast om ett specificerat **hoppvillkor P** är uppfyllt (sant) och då laddas programräknaren med en hoppadress. Om villkoret inte är sant (dvs falskt) så fortsätts den raka exekveringsföljden. Man kan säga att exekveringsföljden har en **förgrening** (eng. **branch**) via en villkorlig hoppinstruktion. Därför benämns instruktioner för villkorliga hopp **"branch"-instruktioner**

Hoppen behöver inte alltid gå framåt i programmet såsom visas i figur F.15 utan kan också gå bakåt.

Instruktioner för villkorliga hopp uttrycks ofta på följande sätt

```
if P then              "branch" om P är sann
  PC:= HOPPADRESS      (med symbolen := menas att PC
                        tilldelas värdet efter symbolen)
```

Om hoppvillkoret P är sant eller falskt kan avgöras med hjälp av flaggorna i flaggregistret, t ex m h a relationerna i tabell F.3. Hoppvillkoret används för att testa ett resultat som har erhållits i instruktionsexekveringen. De olika hopp-villkoren framgår av instruktionslistan, där man lägger märke till att de förekommer i par. Om det finns en "branch"-instruktion för ett hoppvillkor så finns också en annan "branch"-instruktion för det motsatta hoppvillkoret.

Exempel F.19

FLISP-instruktionen BVS $63 tyds på följande sätt:

```
      if V=1 then      "branch if V is set"              BVS $63
        PC:=63₁₆       (hoppa om "overflow"-flaggan är 1)
```

I instruktionslistan finns då också en "branch"-instruktion BVC för hopp om "overflow"-flaggan ej är 1, dvs 0. Den kan med BVC $AB beskrivas på följande sätt:

```
      if V=0 then      "branch if V is cleared"          BVC  $AB
        PC:=AB₁₆       (hoppa om "overflow"-flaggan är 0)
```

Exempel F.20

Antag för FLISP att en operand med tecken (med 2-komplementrepresentation) i ackumulator A jämförs med 0 genom instruktionen TSTA. För att basera ett villkorligt hopp på denna jämförelse finns i instruktionslistan "branch"-instruktionen BLE för hopp om operanden är mindre eller lika med noll.
Den beskrivs med BLE $56 på följande sätt:

```
  if (N⊕V)+Z=1 then    "branch if less or equal"         BLE  $56
    PC:=56₁₆           (hoppa om (N⊕V) + Z=1)
```

De två vanliga instruktionerna för ovillkorliga hopp är

```
hoppadress → PC      jump
                     branch always
```

som i grund och botten båda är "jump"-instruktioner. Vid dessa sker hopp till den hoppadress som operandinformationen specificerar.

"Branch"-instruktionerna har ett speciellt sätt att specificera hoppadressen, dvs en speciell adresseringsmod. I maskininstruktionens andra byte, **ges avståndet till hoppadressen från den adress som finns i programräknaren**, som ett tal med tvåkomplementrepresentation. Med 8 bitar kan då hoppadressen finnas i området

$$PC-128 \leq \text{hoppadressen} \leq PC+127$$

Det skall då observeras att innehållet i PC är adressen till den instruktion som i minnet ligger omedelbart efter "branch"-instruktionen. I maskininstruktionen ges således hoppadressen relativt programräknarens innehåll och adresseringsmoden kallas därför för **PC-relativ** (eng. **PC-relative**).

Exempel F.21

Med FLISP instruktionen BVS $63 på adress 50_{16} blir maskinprogrammet:

adress hex	innehåll hex	kommentar
.	.	
50	26	OPkod för BVS
51	11	hoppavstånd = 63_{16}-PC = 63_{16}-52_{16} = 11_{16}
52	.	början på nästa instruktion
.	.	
.	.	
63	.	hoppadress = PC+11_{16} = 52_{16}+11_{16} = 63_{16}
.	.	

Detta sätt att ange hoppadressen gäller även instruktionen "branch always". Därför benämns den "branch" trots att den ej är en verklig "branch" (förgrening).

Exempel F.22

Med FLISP-instruktionen BRA $09 på adress 00_{16} blir maskinprogrammet:

adress hex	innehåll hex	kommentar
00	21	OPkod för BRA
01	07	hoppavstånd = 09_{16}-02_{16} = 07_{16}
02	.	början på nästa instruktion
.	.	
.	.	
09	.	hoppadress = PC+07_{16} = 02_{16}+07_{16} = 09_{16}
.	.	

Exempel F.23

Om hoppet i ex F.22 istället görs med "jump"-instruktionen blir maskinprogrammet:

```
adress    innehåll    kommentar
 hex        hex

 ------------------
 00          33       OPkod för JMP
 01          09       hoppadress

 ------------------
 02           .       början på nästa instruktion
  .           .
  .           .
  .           .
 --------------------
 09           .       hoppadress
  .           .
```

Genom att specificera hoppet med avståndet till hoppadressen i maskininstruktionen görs hoppet **positionsoberoende**, (eng **position independent**) dvs hoppet sker till rätt adress inom programmet oberoende av var programmet placeras i minnet.

Om maskininstruktionen för ett hopp innehåller den explicita hoppadressen är hoppet positionsberoende. I vissa fall är det viktigt med positionsberoende hopp, ex vis när det är nödvändigt att hoppa till ett program som börjar på en bestämd adress, ett s k **minnesresident** program.

Positionsoberoende hopp är väsentliga för att ett program skall kunna vara **relokerbart**, dvs kunna placeras var som helst i minnet.

Mer om adressering av data

Tidigare i detta avsnitt har adresseringsmoderna

- inherent (eng inherent)
- omedelbar (eng immediate)
- absolut, direkt (eng absolute, direct)
- PC-relativ (eng PC-relative)

definierats i samband med att instruktionerna beskrevs.

Vid absolutadressering adresseras en operand i minnet genom att operandadressen, den s k **effektivadressen**, **EA** (eng **effective address**) explicit ingår i instruktionen. Absolutadressering är endast lämplig när data uppträder på fasta adresser.

Vi skall nu se på några vanliga alternativ till absolutadressering. Det är adresseringsmoder egenskaper som

- att programmet skall kunna arbeta med operandvärden oberoende av deras placering i primärminnet, dvs en instruktions operand skall kunna ha olika adress från körning till körning
- att programmet skall kunna arbeta med och inom datastrukturer av varierande slag, som tabeller, stackar etc
- att dataarean tillhörande ett positionsoberoende program också skall vara positionsoberoende

Indirekt adressering

Ett program kan från körning till körning, använda sig av operander med varierande lägen i M genom instruktioner med indirekt adressering. I dessa fall innehåller instruktionen ej operandens effektivadress utan anger istället det läge, minnesadress eller processorregister, i vilket effektivadressen finns placerad. Man skall därvid se effektivadressen som en parameter till programmet.

När effektivadressen finns lagrad i primärminnet på en adress som explicit finns med i instruktionen, säger man helt generellt att adresseringsmoden är **indirekt** (eng **indirect**). I instruktionen ges en minnesadress och på den finns adressen till operanden.

Den vanligaste formen för indirekt adressering är att låta ett processorregister innehålla effektivadressen och tjäna som **pekare** (eng **pointer**) till operanden. Processorn har då maskininstruktioner använder innehållet i detta register som effektivadress. Pekarregistret anges explicit i instruktionen. Denna adresseringsmod kallas **register-indirekt** (eng **register indirect**).

Register-indirekt adressering med konstant "offset" (indexerad adressering)

Indexering har sitt ursprung i matematikens sätt att numrera element tillhörande en mängd. I datorer används indexering på liknande sätt för att numrera operander i en datastruktur lagrad på en följd av konsekutiva adresser i adressrummet (jfr tabell). En sådan datastruktur kännetecknas av tre parametrar:

- **basadress** (startadress)
- **operandavstånd** (eng. **offset**), som är operandens adressavstånd från basadressen
- **längd**, som är den sista operandens avstånd från basadressen.

Adressen till en operand i strukturen skapas genom att addera operandens avstånd (offset) och basadressen. Man ser tydligt en likhet mellan avstånd och index.

Det finns tre vanliga varianter av indexerad adressering. De skiljer sig åt i sättet att specificera basadress och index och kan sammanfattas enligt

a) basadress i instruktionen och index i processorregister

b) basadress i processorregister och index i instruktionen

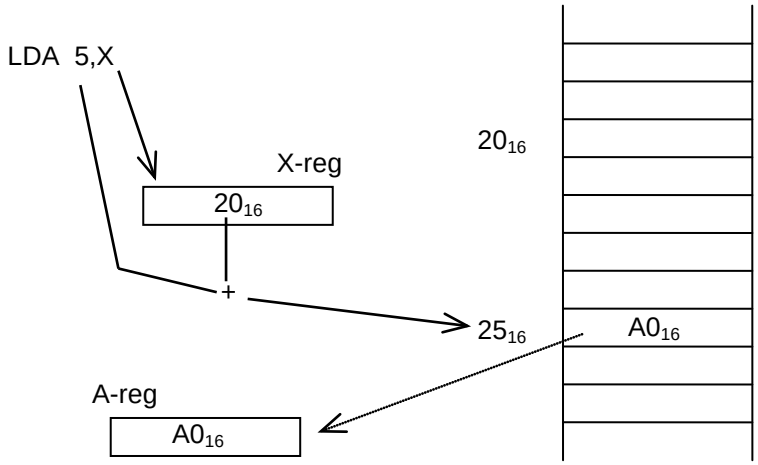
c) basadress i ett processorregister och index i ett annat processorregister

Exempel F.24

Hos FLISP har de två första varianterna formen MNE n,r_{bas} där n är ett positivt eller negativt heltal, i området $-128 \leq n \leq 127$ och r_{bas} något av registren X, Y eller SP. Adresseringsmoden anges av att kommatecknet föregås av n.

Exempel F.25

Indexerad adressering enligt variant a) eller b) sker ex vis med instruktionen **LDA 5,X** . Den läser data på adressen X + 5 och placerar det i ackumulator A, såsom visas i figur F.16.



Figur F.16 Principen för indexerad adressering enligt variant b).

Den har i maskinspråket utseendet

maskininstruktion	operation
F3 (OPkod)	
05 (n)	$M(X + n) \rightarrow A$

OPkoden $F3_{16}$ anger:

- att det är en "load"-instruktion
- att A-registret är destinationsregister
- att X-registret används för basadress (eller index)
- att byten efter OPkoden innehåller index (eller basadressen)

Av figur F.16 ser man att X-registret här innehåller datastrukturens basadress och att index finns i instruktionen, dvs variant b).

Även en tolkning enligt variant a) är möjlig i detta fall. I så fall innehåller X-registret index som adderas till basadressen 5. Den effektiva adressen (adressen till data) blir i båda fallen densamma.

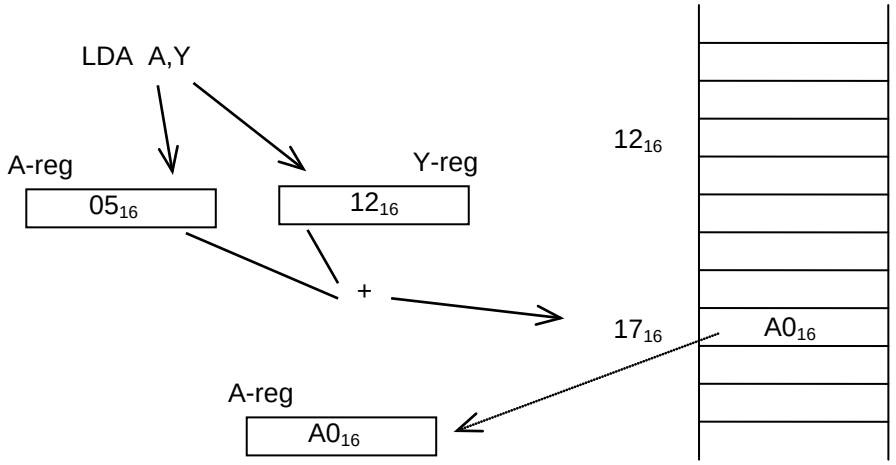
Register-indirekt adressering med ackumulator-"offset"

Exempel F.26

Den tredje varianten har hos FLISP formen MNE A,r_{bas} där r_{bas} är X- eller Y-registret, som innehåller basadressen. Ackumulator A innehåller index, som är ett positivt eller negativt heltal i området [-128, +127]. Adresseringsmoden anges av att kommatecknet föregås av A.

Exempel F.27

Indexerad adressering enligt variant c) sker ex vis med FLISP-instruktionen LDA A,Y . Den läser data på adressen Y + A och placerar det i ackumulator A, såsom visas i figur F.17.



Figur F.17 Principen för indexerad adressering enligt variant c).

Den har i maskinspråket utseendet

maskininstruktion	operation
FA (OPkod)	M(Y + A) $\rightarrow$ A

OPkoden FA_{16} anger:

- att det är en "load"-instruktion
- att A-registret är destinationsregister
- att Y-registret används för basadress (eller index)
- att A-registret används för index (eller basadress)

Av figur F.17 ser man att Y-registret här innehåller datastrukturens basadress och att index finns i A-registret. Även den omvända tolkningen är möjlig.

Register-indirekt adressering med "autoinkrement" och "autodekrement"

Register-indirekt adressering har en naturlig utvidgning som är lämplig att använda när samhörande operander lagrats på konsekutiva (på varandra följande) adresser och man i programmet vill flytta sig från operand till operand. Detta är ex vis vanligt när man arbetar med tabelliknande datastrukturer.

Den naturliga utvidgningen är att låta pekarregistrets innehåll ökas med ett eller minskas med ett varje gång instruktionen utförs. På detta sätt kan man i program röra sig framåt eller bakåt i datastrukturen.

Instruktioner med denna typ av automatisk modifiering av pekarregistrets innehåll har adresseringsmoden **register-indirekt med "autoinkrement" eller "autodekrement".**

Pre- resp. post-inkrementering innebär att instruktionen ökar innehållet i pekarregistret med 1 före resp. efter det att adresserad operand använts för att förbereda användning av nästa element i strukturen.

Pre- resp. post-dekrementering innebär att instruktionen minskar innehållet i pekarregistret med 1 före resp. efter det att adresserad operand använts.

Exempel F.28

Hos FLISP finns register-indirekt adressering med pre- och post-inkrementering

```
MNE  ,+X
MNE  ,X+
```

och med pre- och post-dekrementering

```
MNE  ,-X
MNE  ,X-
```

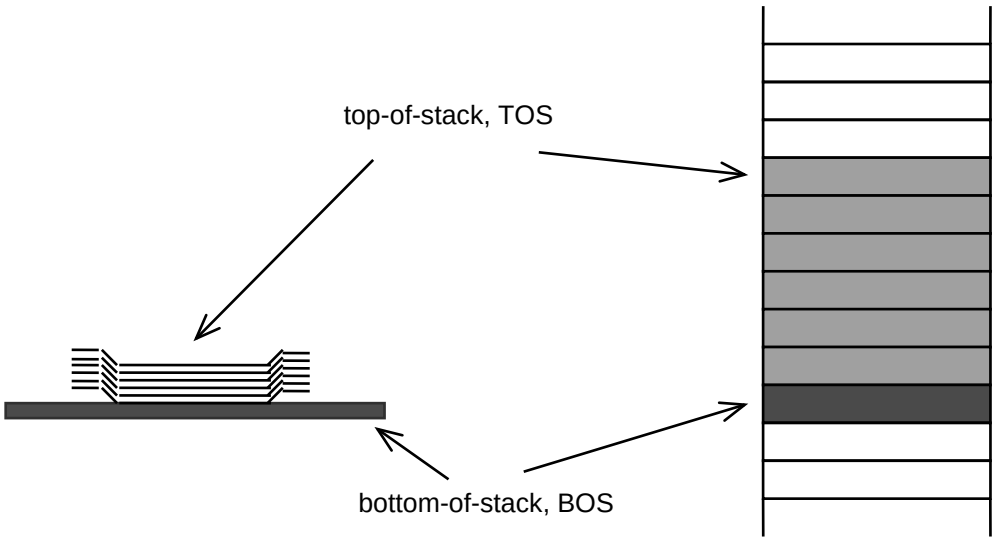
För laddning av A finns då instruktionerna

LDA ,+X	X+1 → X M(X) → A	LDA ,-X	X-1 → X M(X) → A
LDA 1,X+	M(X) → A X+1 → X	LDA ,X-	M(X) → A X-1 → X

Stackbyggnad och användning

Med "store"- och "load"-instruktioner som har register-indirekt adressering med autoinkrement och autodekrement kan man skapa och använda en datastruktur som kallas **stack**.

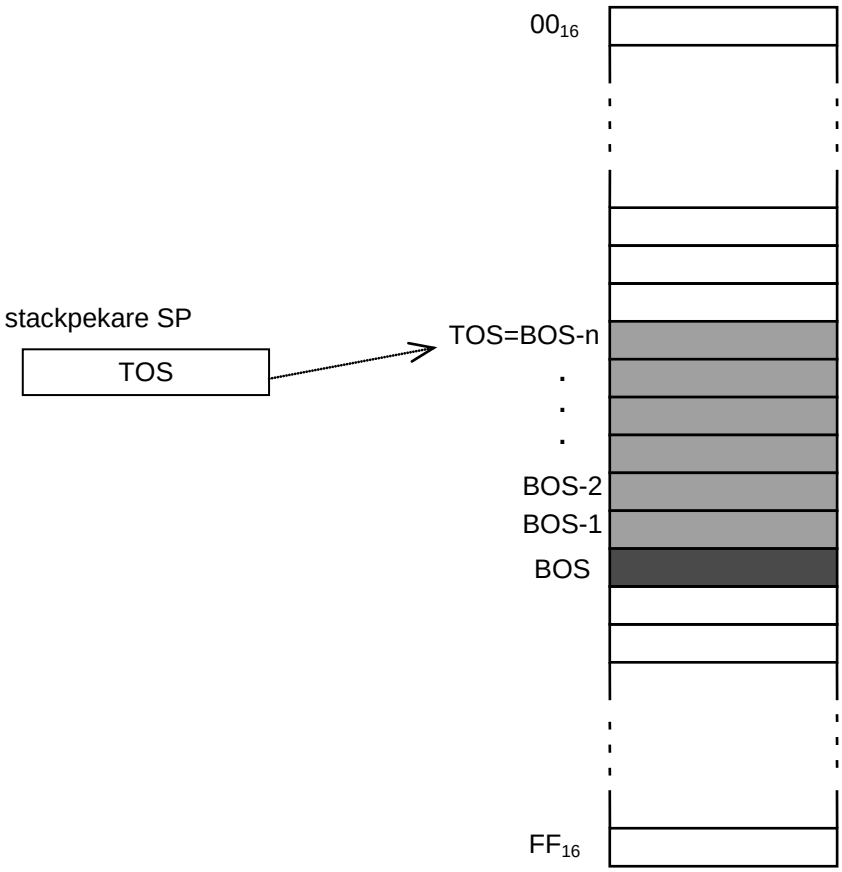
På en stack lagras binära ord på precis samma sätt som man lägger hö på en höstack eller staplar tallrikar på en hylla, dvs bildligt talat ovanpå varandra, se figur F.18. Basläget kallas **"bottom-of-stack", BOS** och läget för det sist pålagda elementet kallas **"top-of-stack", TOS**.



Figur F.18 En stackstruktur.

Nya element tillförs ovanifrån, dvs läggs på toppen. Principen för att komma åt de binära orden, höet och tallrikarna är "sist in - först ut" (eng. last in - first out, LIFO). Processen att lägga nya element på stacken benämns **"push"** och processen att hämta element ifrån stacken benämns **"pull"** (ibland **"pop"**).

Stacken är en **LIFO-struktur**, som främst används för att temporärt lagra data på ett enkelt sätt. Ett smidigt sätt att göra detta är via "store"- resp. "load"-instruktioner med register-indirekt adressering och med automatisk dekrementering resp. inkrementering. Ett pekarregister, kallat **stackpekare SP** (eng **stack pointer**) får hålla reda på TOS-läget enligt figur F.19. Stacken initierades när stackpekaren en gång laddades med BOS-adressen.



Figur F.19 Stackens tillväxt och adressering via stackpekaren.

I en processor finns i regel minst ett stackpekarregister, SP. Det finns då också speciella "push"- och "pull"-instruktioner som lagrar data från processorns interna register på stacken och hämtar data från stacken till interna register

SP-1 → SP	push r_{src}
r_{src} → M(SP)	
M(SP) → r_{dst}	pull r_{dst}
SP+1 → SP	

Exempel F.29

FLISP har en stackpekare SP och "push"-instruktioner samt "pull"-instruktioner.

"PUSH"	"PULL"
PSHA	PULA
PSHX	PULX
PSHY	PULY
PSHC	PULC

Exempel F.30

FLISP-instruktionen PSHA placerar en kopia av det ord som finns i ackumulator A på stacken, som definieras av stackpekare SP.

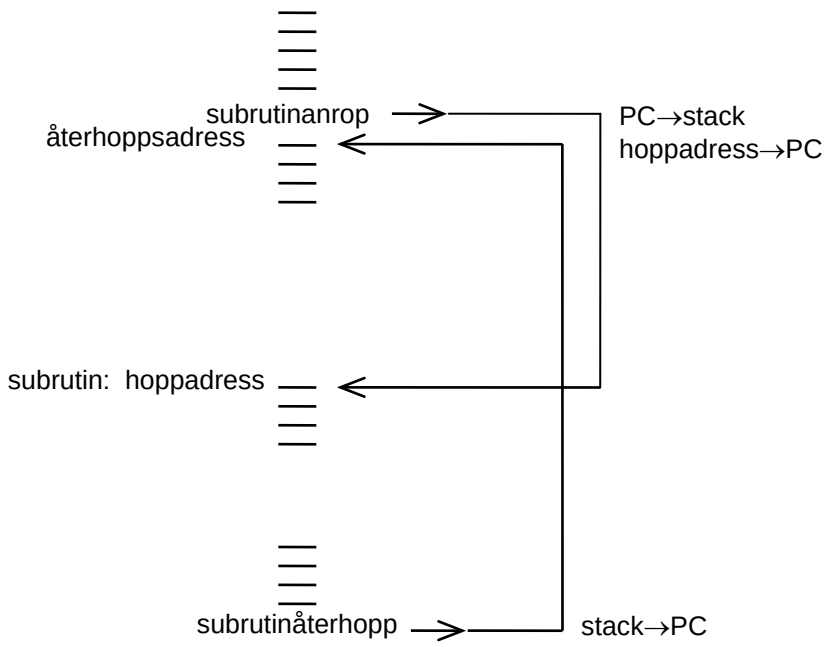
maskininstruktion	operation
10 (OPkod)	SP-1 → SP A → M(SP)

Hopp till subrutin

Utöver de hoppinstruktioner som tidigare beskrivits har processorn instruktioner för att göra hopp till och från s k **subrutiner** (eng **subroutines**). Med dessa kan programmerare modularisera sina program och placera modulerna fritt i minnet. Hopp till subrutin är vanligen ovillkorliga och kallas subrutinanrop. Följande instruktioner är typiska

```
återhoppsadress → stack
hoppadress → PC                              jump to subroutine
                                             branch to subroutine

återhoppsadress (från stack) → PC            return from subroutine
```

Vid de två första instruktionerna sker hopp till den hoppadress som operandinformationen specificerar. Innan de laddar programräknaren med hoppadressen, lagrar de först programräknarens innehåll på stacken (att stackpekaren justeras automatiskt är underförstått). På stacken lagras således adressen till den instruktion i det anropande programmet som ligger omedelbart efter hoppinstruktionen. Detta görs för att man senare skall kunna hoppa tillbaka till denna adress. Återhopp sker via en "pull"-instruktion, som har den mnemoniska beteckningen RTS. Den återladdar programräknaren med adress från stacken. Denna utför exakt samma sak som en "pull"-instruktion PULPC skulle göra. I realiteten utför den ett ovillkorligt hopp till subrutinens återhoppsadress. Exekveringsföljden vid hopp till en subrutin och återhopp visas i figur F.20.



Figur F.20 Exekveringsföljd vid subrutinanrop.

Med instruktioner för anrop av och återhopp från subrutin kan programmeraren på ett smidigt sätt använda programavsnitt som placerats på annat ställe i minnet genom att anropa dem som subrutiner.

Exempel F.31

Hos FLISP finns följande instruktioner för subrutinanrop

JSR adr	SP-1 → SP; PC → M(SP) adr → PC
BSR adr	SP-1 → SP; PC → M(SP) adr = PC + offset → PC

och följande instruktion för återhopp

RTS	M(SP) → PC; SP+1 → SP

BSR är som synes en "branch"-instruktion vilket innebär att hoppadressen ingår som ett avstånd (offset) i maskininstruktionen. När instruktionen utförs så adderas avståndet till PC's värde och bildar hoppadressen.

Övningsuppgifter

Övningsuppgifterna i kapitel F avser FLIS-processorn, vars instruktioner och motsvarande koder definieras i ”INSTRUKTIONSLISTA FÖR FLISP”.

F.1 Ge i hexadecimal form maskininstruktionen för

a) LDA $3A
b) LDA 58
c) STA 235
d) STA $EB
e) LDA #$56
f) LDA #%01010110
g) LDA #86

F.2 Ett antal på varandra följande minnesord har följande hexadecimala innehåll:

a) F1, 20, E1, 21, 96, 0F, A9, 25, 06, 97, 10.
b) F0, 80, A1, 50, FA, 94, 4A, 99, F0, 1B.
c) 90, 40, F1, A0, 0B, F4, 54, 00.

Den första byten ($F1_{16}$, $F0_{16}$ resp 90_{16}) är en operationskod för FLIS-processorn. Översätt sekvensen till assemblerspråk, dvs disassemblera sekvensen.

F.3 Ge en sekvens av LDA- och STA-instruktioner (dvs ett programavsnitt) som flyttar innehållet på adresserna $1A_{16}$-$1C_{16}$ till adresserna $2A_{16}$-$2C_{16}$.

F.4 Ge en sekvens av LDA- och STA-instruktioner som placerar telefonnumret 4631678900 i NBCD-form på adresserna 21_{16}-25_{16}.

F.5 Manuell översättning från assemblerspråk till maskinkod kallas ofta "**handassemblering**".

a) Handassemblera följande instruktionssekvens. Första instruktionen skall placeras på adress 80_{16}. Hur många minnespositioner upptar instruktionssekvensen?

```
LDA     $10
ANDA    #$F
ORA     $11
EORA    #$44
ANDA    #$EE
COMA
STA     $10
```

b) Antag att talet 10110110_2 finns på adress 10_{16}. Ange i hexadecimal form det tal som kommer att finnas i denna minnesposition efter exekvering av instruktionssekvensen ovan, om adress 11_{16} innehåller 01_{16} före exekveringen.

F.6 a) Skriv en instruktionssekvens som inverterar bitarna b_0 och b_7 i A-registret och ettställer bitarna b_2 och b_5 samt nollställer bit b_4.

b) Handassemblera instruktionssekvensen. Första instruktionen skall placeras på adress 20_{16}.

F.7 I minnet finns data enligt figuren till höger.

Skriv en instruktionssekvens som ökar innehållet på adressen 80_{16} med 5 och minskar innehållet på adressen 81_{16} med 7.

Vidare skall instruktionssekvensen addera innehållen på minnesadresserna 82_{16} och 83_{16} samt placera summan på adressen 84_{16}.

Översätt instruktionssekvensen till maskinkod (hexadecimal form) och placera den i minnet med början på adressen 20_{16}.

Adr	
80_{16}	56_{16}
81_{16}	23_{16}
82_{16}	05_{16}
83_{16}	16_{16}
84_{16}	$B7_{16}$

F.8 Två åttabitars tal finns lagrade i minnet på adresserna 50_{16} och 51_{16}. Skriv en instruktionssekvens som adderar talen och placerar summan på adressen 52_{16} i minnet. Summan förutsätts rymmas i åtta bitar (<256). Översätt instruktionssekvensen till maskinkod. Startadressen skall vara 10_{16}.

F.9 Skriv en instruktionssekvens som adderar 9 till talet som finns på adressen 60_{16} i minnet och placerar summan (åtta bitar) på adressen 54_{16}. Översätt instruktionssekvensen till maskinkod. Startadress skall vara 18_{16}.

F.10 På adress 65_{16} finns ett tal med inbyggt tecken (tvåkomplementrepresentation).

a) Skriv en instruktionssekvens som multiplicerar talet med 4.

Vad hamnar på adress 65_{16} om innehållet där från början är

b) 18_{10}

c) -23_{10}

d) 38_{10}

F.11 "Branch"-instruktionen BRA $50 är placerad med början på adress 37_{16}. Vad är dess maskinkod?

F.12 "Branch"-instruktionen BRA $05 är placerad med början på adress 20_{16}. Vad är dess maskinkod?

F.13 Instruktionssekvensen nedan multiplicerar innehållen (talen x och y) på minnesadresserna $5C_{16}$ (x) och $5D_{16}$ (y) genom att addera det ena talet till register A så många gånger som det andra talet anger. Varje gång det uppstår en minnessiffra ut från denna addition ökas den mest signifikanta byten av produkten med ett. Produkten (16 bitar) lagras på adresserna $5E_{16}$ och $5F_{16}$ med mest signifikant del på $5E_{16}$. Minnesadressen $5B_{16}$ är ledig och kan användas.

Översätt instruktionssekvensen till maskinkod. Startadress skall vara 30_{16}.
Analysera instruktionssekvensen genom att rita en maskinberoende flödesplan som beskriver multiplikationsförloppet.

Läge	Operation	Operand	Kommentar
START	CLR	$5E	Nollställ variabeln för produkten.
	CLR	$5F	-"-
	LDA	$5C	Hämta talet x till reg A.
	TSTA		Undersök talet x.
	BEQ	SLUT	Talet x = 0. Hoppa ut.
	STA	$5B	Använd talet x som räknare i minnet.
	TST	$5D	Undersök talet y.
	BEQ	SLUT	Talet y = 0. Hoppa ut.
	CLRA		Nollställ reg A.
ADDLOOP	ADDA	$5D	Addera talet y till reg A.
	BCC	NOCARRY	Kontrollera om summan i reg A > 255.
	INC	$5E	Ja, öka innehållet i produktens höga byte.
NOCARRY	DEC	$5B	Minska talet x med ett.
	BNE	ADDLOOP	Upprepa tills talet x = 0.
	STA	$5F	Nu har vi adderat talet y till reg A x ggr.
			Skriv därför produktens låga byte i minnet.
SLUT	.		
	.		
	.		

F.14 I följande instruktionssekvens genereras en puls i position 0 på utport FB_{16}.

```
Läge      Operation   Operand        Kommentar
START     CLR         $FB
          LDA         #25
          STA         $10
SCOUNT    DEC         $10
          BNE         SCOUNT
          LDA         #%00000001
          STA         $FB
          LDAB        #100
          STA         $10
PCOUNT    DEC         $10
          BNE         PCOUNT
          COMA
          STA         $FB
```

a) Rita en maskinberoende flödesplan för sekvensen.

b) Antag att FLIS-processorn har klockfrekvensen 1 MHz och att sekvensen startas vid tidpunkten t = 0. När i tiden genereras pulsen? Hur lång är den?

c) Kommentera instruktionerna på lämpligt sätt. Kommentarerna skall helst vara maskinoberoende.

d) Översätt instruktionssekvensen till maskinkod. Antag att START skall läggas på adressen 20_{16}. Måste den maskinkod du nu producerat alltid ligga på dessa adresser?

F.15 Vad blir den *effektiva adressen* i följande instruktioner om innehållet i X-registret = 20_{16}. För instruktioner som arbetar med data är *effektiva adressen* adressen till data. Översätt instruktionerna till maskinkod.

a) LDA 0,X
b) LDA 16,X
c) LDA -16,X
d) LDA ,X+
e) LDA ,-X

F.16 Skriv en instruktionssekvens som omvandlar ett 4-bitars binärt tal till Graykod. Det binära talet skall kontinuerligt läsas från bit b_0-b_3 på inport FB_{16}. Bit b_4-b_7:s värden är ej kända och kan variera mellan läsningarna. Graykoden skall matas ut på bit b_0-b_3 på utport FB_{16} med bit b_4-b_7 nollställda. Graykoden för siffrorna 0_{16}-F_{16} finns lagrade i bit b_0-b_3 i adress 30_{16} - $3F_{16}$, med bit b_4-b_7 nollställda.

F.17 I den första fjärdedelen av minnet innehåller vid ett visst tillfälle varje position värdet av adressen.

Ex:

adress (hex)	innehåll (hex)
00	00
.	.
.	.
1C	1C
.	.
.	.
32	32
.	.
.	.
3F	3F

Ange innehållen i registren A och X efter varje instruktion i instruktionssekvensen nedan.

```
                          A        X
LDX   #$12
LDA   $3E
LDA   0,X
LDX   18,X
LDA   $E0,X
LDX   4,X
LDA   ,X+
LDA   0,X
LDA   ,-X
```

F.18 Skriv en subrutin MASK som nollställer bit 7 i varje byte inom adressområdet 40_{16}-60_{16}.

F.19 En tabell med 5 olika 8-bitars tal utan tecken lagras med början på adress 18_{16} i minnet. Skriv en instruktionssekvens som letar upp det största talet.

Talets värde och läge skall anges i minnespositionerna VALUE resp PLACE, vilka anses fördefinierade.

F.20 Ge en sekvens av instruktioner som definierar en stack med början på adress $F0_{16}$ och placerar operanderna 06_{16}, 35_{16} och 28_{16} på stacken. Vad är stackpekarens innehåll när sekvensen genomlöpts av processorn?

F.21 Utgå ifrån att stacken redan är definierad (vilket är den vanligaste situationen vid skrivning av programavsnitt) och ge en sekvens av instruktioner som använder stacken för att flytta innehållet i ackumulator A till register X.

F.22 Antag att vi har ett huvudprogram och två subrutiner enligt nedan:

Huvudprogram

Adr	Mem.	Opr
40	LDA	$10
42	JSR	$50 (Tvåkomp)
44	JMP	$44

Subrutin Tvåkomp

Adr	Mem	Opr
50	PSHA	
51	NEGA	
52	STA	$11
54	JSR	$60 (Plusett)
56	PULA	
57	RTS	

Subrutin Plusett

Adr	Mem	Opr
60	PSHA	
61	ADDA	#$01
63	STA	$12
65	PULA	
66	RTS	

Antag vidare att huvudprogrammet körs.

Fyll i minnesadress, instruktion, registervärden och stackens innehåll så att de motsvarar situationen efter det att motsvarande instruktion har exekverats. Markera även med en pil det minnesord som stackpekaren (SP) pekar på. Utgå från situationen efter första instruktionen, vilken finns given nedan.

Adr Instruktion 1
40 LDA $10

Register:

$PC = 42_{16}$

$A = FF_{16}$

$SP = D0_{16}$

Stacken:

CA	
CB	
CC	
CD	
CE	
CF	
D0	00 ← (S)

Adr Instruktion 2
42 JSR $50

Register:

PC =

A =

SP =

Stacken:

CA	
CB	
CC	
CD	
CE	
CF	
C0	00

Adr Instruktion 3

Register:

PC =

A =

SP =

Stacken:

CA	
CB	
CC	
CD	
CE	
CF	
D0	00

Adr Instruktion 4

Register:

PC =

A =

SP =

Stacken:

CA	
CB	
CC	
CD	
CE	
CF	
D0	00

Adr Instruktion 5

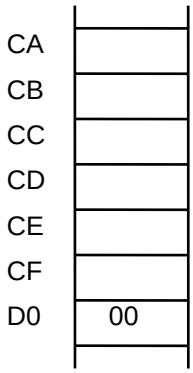
Register:

PC =

A =

SP =

Stacken:



Adr Instruktion 6

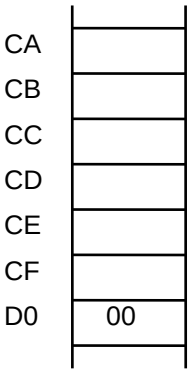
Register:

PC =

A =

SP =

Stacken:



Adr Instruktion 7

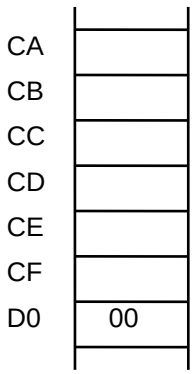
Register:

PC =

A =

SP =

Stacken:



Adr Instruktion 8

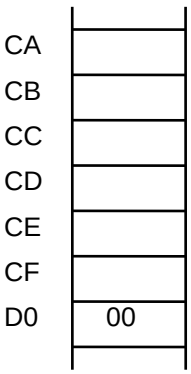
Register:

PC =

A =

SP =

Stacken:



Adr Instruktion 9

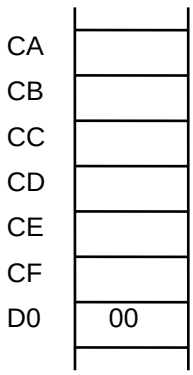
Register:

PC =

A =

SP =

Stacken:



Adr Instruktion 10

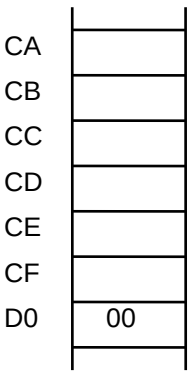
Register:

PC =

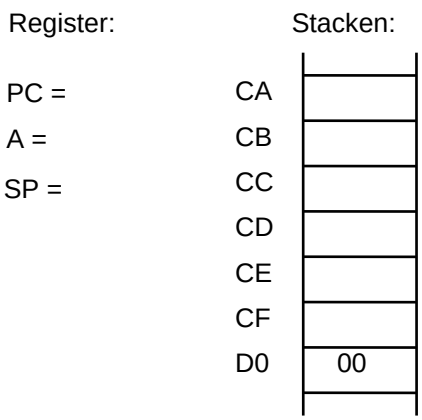
A =

SP =

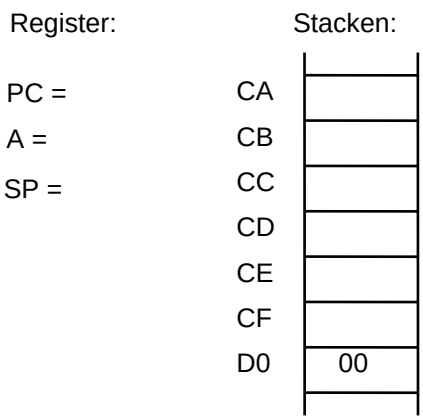
Stacken:



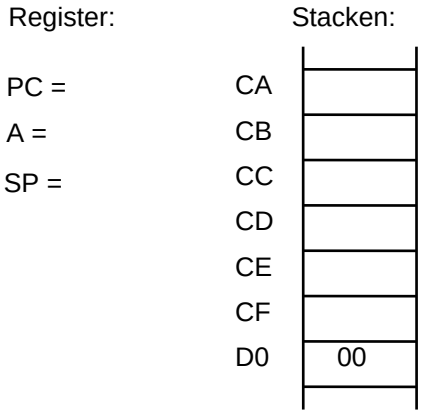
Adr Instruktion 11



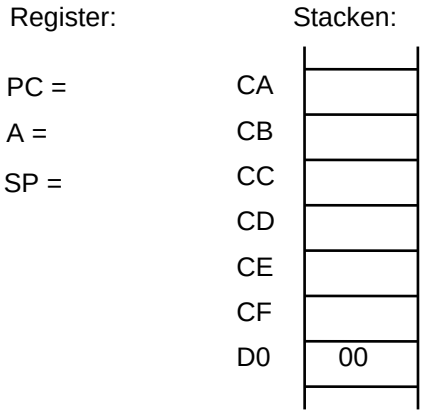
Adr Instruktion 12



Adr Instruktion 13



Adr Instruktion 14



F.23 Skriv ett programavsnitt som läser ett tal från inport FB_{16}. Om talet är 00_{16} skall en subrutin FALL1 anropas. Talet skall då ligga i A-registret för att subrutinen skall kunna använda det för bearbetning. Om talet är 18_{16} skall på motsvarande sätt en subrutin FALL2 anropas. Om talet inte är lika med något dessa värden skall en subrutin FALL3 anropas.
Rita också en flödesplan.

F.24 Skriv en subrutin som adderar två 8-bitars naturligt binärkodade tal. De två talen skall finnas i Y- resp. A-registret vid anrop av subrutinen. Mest signifikanta byten skall returneras i Y-reg. och minst signifikanta byten i A-reg.

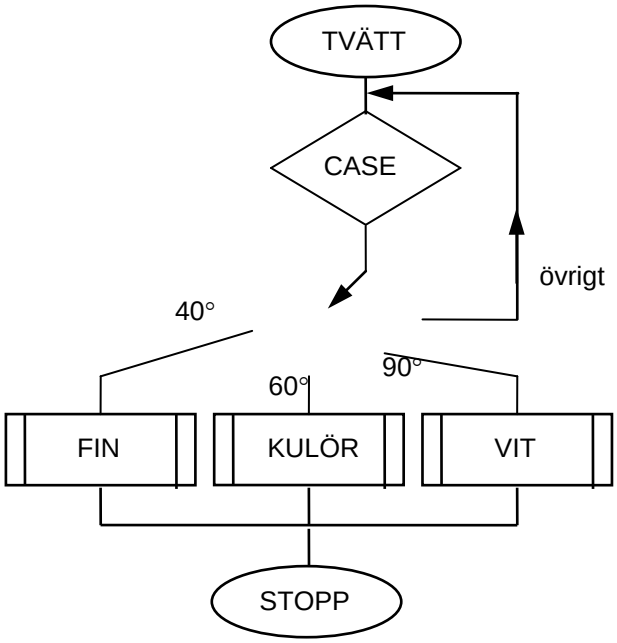
Av de interna registren får subrutinen endast påverka Y- och A-registren samt flaggregistret. Z-flaggan skall sättas till 1 om summan är lika med noll. De övriga flaggornas värden saknar betydelse.

Rita även en flödesplan.

F.25 En enkel datorstyrd tvättmaskin har tre olika tvättprogram

- 40°C Fintvätt / 60°C Kulörtvätt / 90°C Vittvätt.

När tvättmaskinen startats agerar den enligt flödesplanen nedan.



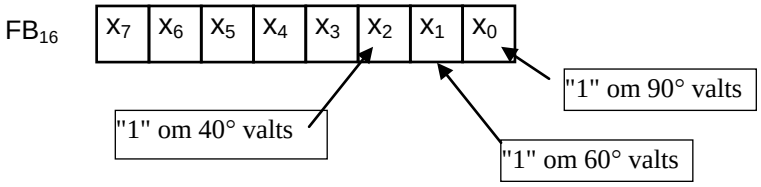
a) Rita om flödesplanen så att fyrvalssituationen realiseras som en följd av tvåval (**if**-**then**-**else**)!

b) Användaren kan trycka på tre olika knappar som svarar mot de olika tvättprogrammen. Den valda tvättemperaturen är åtkomlig för processorn via inporten FB_{16} enligt figuren nedan.

Observera att vissa värden på $x_2x_1x_0$ representerar ogiltiga val (t.ex. om både x_1 och x_0 är ett). Dessa fall motsvaras av alternativet "övrigt" i flödesplanen.

De olika tvättprogrammen är subrutiner med början på de symboliska adresserna "Fin", "Kulör" respektive "Vit". Omsätt den nya flödesplanen i ett program skrivet med FLEX instruktioner.

Ledning:
Varje tvättemperatur motsvaras av ett unikt bitmönster i de tre minst signifikanta positionerna i data från inport FB_{16}, 60°C Kulörtvätt motsvaras t.ex. av $x_2x_1x_0$ = 010.



F.26 Skriv en subrutin som skriver in talen 00,01,02,...,$7F_{16}$ i adresserna 00, 01, ..., $7F_{16}$.

F.27 Vad är det för fel på följande subrutin?

```
SUBRTN   PSHA
         LDA    TAL_1
         ADDA   TAL_2
         NEGA
         RTS
```

Svar till vissa av övningsuppgifterna

F.1
```
a) F1 3A        b) F1 3A
c) E1 EB        d) E1 EB
e) F0 56        f) F0 56
g) F0 56
```

F.2

a)
```
LDA  $20
STA  $21
ADDA #$0F
ANDA $25
NEGA
CMPA #$10
```

b)
```
LDA  #$80
LDY  $50
LDA  A,Y
SUBA #$4A
ANDA #$F0
TFR  Y,X
```

c)
```
LDX  #$40
LDA  $A0
ASLA
LDA  A,X
JSR  0,X
```

F.3
```
LDA $1A
STA $2A
LDA $1B
STA $2B
LDA $1C
STA $2C
```

F.4
```
LDA #$46
STA $21
LDA #$31
STA $22
LDA #$67
STA $23
LDA #$89
STA $24
LDA #$00
STA $25
```

F.5 a)
```
Adress:   80 81 82 83 84 85 86 87 88 89 8A 8B 8C
Innehåll: F1 10 99 0F AA 11 9B 44 99 EE 0A E1 10
```
Instruktionssekvensen upptar 13 minnespositioner.

b) BD_{16}

F.6 a)
```
EORA   #%10000001
ORA    #%00100100
ANDA   #%11101111
```

b)
```
Adress: 20, 21, 22, 23, 24, 25
Innehåll:  9B, 81, 9A, 24, 99, EF
```

F.7
```
LDA  $80
ADDA #5
STA  $80
LDA  $81
SUBA #7
STA  $81
LDA  $82
ADDA $83
STA  $84
```

```
Adress:  20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31
Sekvens: F1 80 96 05 E1 80 F1 81 94 07 E1 81 F1 82 A6 83 E1 84
```

F.8
```
LDA  $50
ADDA $51
STA  $52
```
```
Adress:  10 11 12 13 14 15
Sekvens: F1 50 A6 51 E1 52
```

F.9
```
LDA  $60
ADDA #9
STA  $54
```
```
Adress:  18 19 1A 1B 1C 1D
Sekvens: F1 60 96 09 E1 54
```

F.10 a) `ASL $65` `ASL $65` b) 72_{10} c) -92_{10} d) -104_{10}

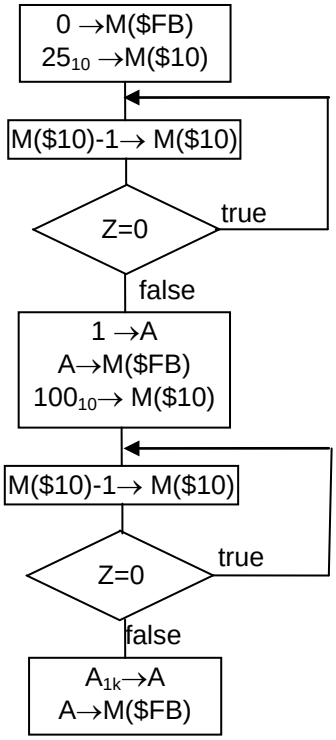
F.11 21, 17

F.12 21, E3

F.13

	Instr.(hex)		Adress	Innehåll	
START	CLR	$5E	30	35	
			31	5E	
	CLR	$5F	32	35	
			33	5F	
	LDA	$5C	34	F1	
			35	5C	
	TSTA		36	09	
	BEQ	SLUT	37	24	
			38	15	($4C_{16}$-39_{16} = 13_{16})
	STA	$5B	39	E1	
			3A	5B	
	TST	$5D	3B	39	
			3C	5D	
	BEQ	SLUT	3D	24	
			3E	0D	($4C_{16}$-$3F_{16}$ = $0D_{16}$)
	CLRA		3F	05	
ADDLOOP	ADDA	$5D	40	A6	ADDLOOP = 40_{16}
			41	5D	
	BCC	NOCARRY	42	29	
			43	02	(46_{16}-44_{16} = 02_{16})
	INC	$5E	44	37	
			45	5E	
NOCARRY	DEC	$5B	46	38	NOCARRY = 46_{16}
			47	5E	
	BNE	ADDLOOP	48	25	
			49	F6	(40_{16}-$4A_{16}$= $F6_{16}$)
	STA	$5F	4A	E1	
			4B	5F	
SLUT	...		4C		(SLUT = $4C_{16}$)

F.14 a)



	Instruktion	Adr.	Inneh.	~	Kommentar
START	CLR $FB	20	35	3	Nollställ utport M($FB)
		21	FB		
	LDA #25	22	F0	2	Sätt för-
		23	19		dröjning
	STA $10	24	E1	3	
		25	10		
SCOUNT	DEC $10	26	38	4	Vänta
		27	10		
	BNE SCOUNT	28	25	4	
		29	FC		26_{16}-$2A_{16}$=FC_{16}
	LDA #%00000001	2A	F0	2	
		2B	01		
	STA $FB	2C	E1	3	Mata ut. Pulsstart.
		2D	FB		
	LDA #100	2E	F0	2	Sätt ny
		2F	64		väntetid
	STA $10	30	E1	3	
		31	10		
PCOUNT	DEC $10	32	38	4	Vänta
		33	10		
	BNE PCOUNT	34	25	4	
		35	FC		32_{16}-36_{16}=FC_{16}
	COMA	36	0A	3	
	STA $FB	37	E1	3	Mata ut Pulsen
		38	FB		slut

b) Pulsen genereras efter 3+2+3+25*(4+4)+2+3 = 213 klockcykler, d v s vid t = 0,213 ms.

Den är 2+3+100*(4+4)+3+3 = 811 klockcykler, d v s 0,811 ms lång.

d) Nej, koden är inte beroende av sin placering i minnet, förutom att den inte får överlappa adressen 10_{16}.

F.15

Instruktion	Maskinkod	Effektiv adress
LDA 0,X	F3,00	20_{16}
LDA 16,X	F3,10	30_{16}
LDA -16,X	F3,F0	10_{16}
LDA ,X+	F5	20_{16}
LDA ,-X	F8	$1F_{16}$

F.16

```
        LDX   #$30
LOOP    LDA   $FB
        ANDA  #$0F
        LDA   A,X
        STA   $FB
        BRA   LOOP
```

F.17

Instruktion	A	X (efter instruktionen)
LDX #$12	-	12
LDA $3E	3E	12
LDA 0,X	12	12
LDX 18,X	12	24
LDA $E0,X	04	24
LDX 4,X	04	28
LDA ,X+	28	29
LDA 0,X	29	29
LDA ,-X	28	28

F.18

```
MASK    PSHA
        PSHC
        PSHX
        LDX    #$40        startadress
LOOP    LDA    0,X         hämta byten
        ANDA   #$7F        nollställ ms bit
        STA    ,X+         skriv tillbaks byte
        CMPX   #$61
        BNE    LOOP        om inte: fortsätt
MASKBIT PULX
        PULC
        PULA
        RTS
```

F.19

```
        LDX    #$18        startadress
        LDA    #4          antal - 1
        STA    COUNT       räknare för antal bytes
        LDA    0,X         första värdet
        STX    PLACE       adress+1 till första värdet
        STA    VALUE       första värdet är nu max
  LOOP  LDA    ,+X         hämta nästa värde
        CMPA   VALUE       nytt maxvärde?
        BLS    LESS        nej
        STA    VALUE       ja, lagra nytt värde
        STX    PLACE       adress+1 till första värdet
  LESS  DEC    COUNT       antal bytes kvar - 1
        BNE    LOOP        färdigt? nej
```

F.20
```
        LDSP   #$F0
        LDA    #$06
        PSHA
        LDA    #$35
        PSHA
        LDA    #$28
        PSHA                (SP efter = ED16)
```

F.21
```
        PSHA
        PULX
```

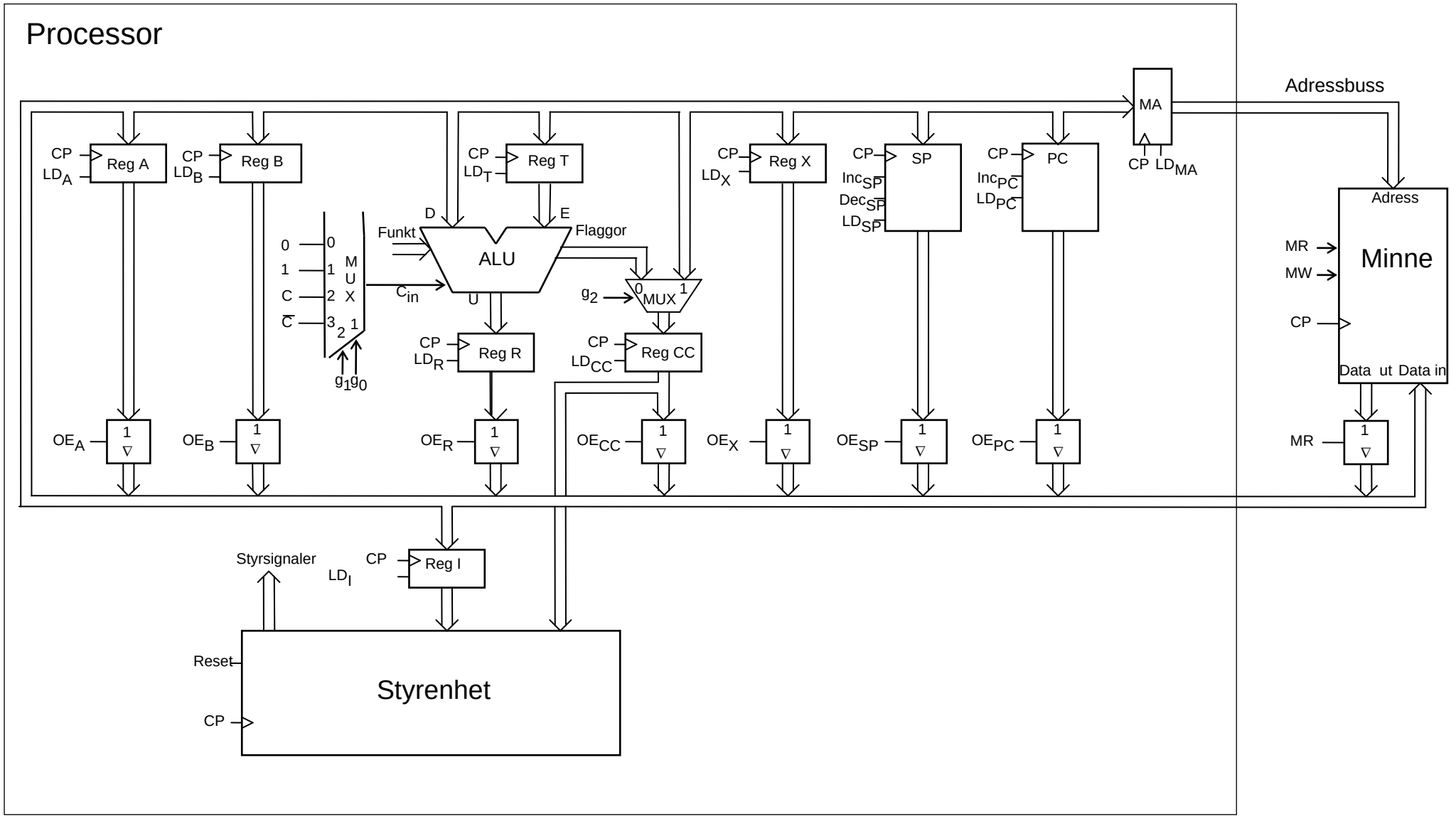
F.23
```
              LDA  $FB
              TSTA
              BEQ  LEV1
              CMPA #$18
              BEQ  LEV2
              JSR  FALL3
              BRA  END
        LEV1  JSR  FALL1
              BRA  END
        LEV2  JSR  FALL2
        END
```

F.24
```
        ADDAY  PSHY
               ADDA   0,SP
               BCC    TST0
               LDY    #1
        TST0   CMPY   #$00
               BNE    EXIT
               TSTA
        EXIT   LEASP  1,SP
               RTS
```

F.26
```
        WRIMEM PSHA
               PSHC
               PSHX
               LDX    #$00
               CLRA
        LOOP   STA    ,X+
               INCA
               CMPX   #$80
               BNE    LOOP
               PULX
               PULC
               PULA
               RTS
```

F.27 Obalanserad stack, fel återhoppsadress

Ext-8



Datorn FLEX sammansatt av dataväg, styrenhet och minnesenhet.

Ext-9 (Ver 2014-02-26)

Villkorliga hopp

Instruktionsuppsättningen för FLEX- och FLIS-processorn har ett antal villkorliga hoppinstruktioner. De kan indelas i följande tre grupper:

1. **Enkla hoppvillkor.**
2. **Hoppvillkor för tal utan inbyggt tecken.**
3. **Hoppvillkor för tal med inbyggt tecken. (2-komplementrepresentation)**

1. Enkla hoppvillkor.
Vid de enkla villkorliga hoppen testas innehållet i en av flaggvipporna N, Z, V eller C och hoppet utförs om villkoret är uppfyllt, dvs den aktuella flaggvippans värde, är 0 resp 1.

2. Hoppvillkor för tal utan inbyggt tecken.
Vid villkorliga hopp, där hoppvillkoret avser tal utan inbyggt tecken, förutsätts att flaggorna har påverkats av en subtraktion X - Y före hopptestet. X och Y är 8-bitars tal som tillhör intervallet [0, 255]. Instruktionssekvensen nedan visar hur flaggorna är tänkta att påverkas före ett villkorligt hopp.

```
LDAA      XVALUE     Hämta talet X från minnet till ackumulator A
CMPA      #Y         Låt skillnaden X - Y påverka flaggorna
B(Villkor) Hoppadress Utför hoppet om villkoret är uppfyllt
```

Eftersom talen X och Y nu betraktas som 8-bitars tal utan inbyggt tecken tillhör de intervallet [0, 255]. När skillnaden X - Y bildas av "compare"-instruktionen ovan påverkas flaggorna och man kan av deras värden få information om storleksrelationen mellan X och Y. Det finns sex olika sådana storleksrelationer:

$X > Y$, $X \geq Y$, $X = Y$, $X \neq Y$, $X \leq Y$ och $X < Y$.

För tal utan inbyggt tecken kan dessa relationer bestämmas med hjälp av flaggvipporna C och Z enligt nedan.

Relation	C	Z	Sant om
$X > Y$	0	0	$C' \cdot Z' = 1$
$X = Y$	0	1	$Z = 1$
$X < Y$	1	0	$C = 1$

Observera att fallet C = Z = 1 inte inträffar och därför kan betraktas som "don't care".

Nedan formuleras hoppvillkoren för de sex villkorliga hopp som undersöker de sex storleksrelationerna mellan X och Y, om X och Y tolkas som tal utan inbyggt tecken.

Relation	Villkorlig hoppinstruktion	Hoppvillkor
$X > Y$	BHI (Branch if X is higher than Y)	$C' \cdot Z'$
$X \geq Y$	BHS (Branch if X is higher or same as Y)	C'
$X = Y$	BEQ (Branch if X is equal to Y)	Z
$X \neq Y$	BNE (Branch if X is not equal to Y)	Z'
$X \leq Y$	BLS (Branch if X is lower or same as Y)	$(C' \cdot Z')' = C + Z$
$X < Y$	BLO (Branch if X is lower than Y)	C

Ext-9 (Ver 2014-02-26)

3. Hoppvillkor för tal med inbyggt tecken. (2-komplementrepresentation)
Vid villkorliga hopp, där hoppvillkoret avser tal med inbyggt tecken, påverkas flaggorna av en subtraktion före hopptestet på exakt samma sätt som i föregående avsnitt. Enda skillnaden är att talen X och Y nu tolkas som 8-bitars tal med inbyggt tecken. Talen tillhör därför intervallet [-128, +127]. Eftersom talen nu har tecken måste flaggorna N, V och Z användas för att bestämma storleksrelationerna mellan X och Y enligt nedan. Först krävs dock ett litet resonemang kring flaggorna N och V.

När (2-komplement-) overflow inträffar vid subtraktionen X - Y upptäcks detta genom att teckenbiten N får fel värde. Man kan dock skapa en korrekt teckenbit, som gäller vare sig overflow inträffar eller ej, genom att bilda N ⊕ V. Om overflow inträffar vet man ju att teckenbiten N får fel värde. Eftersom V samtidigt får värdet 1 inverterar man N genom att bilda N ⊕ V (N ⊕ 1 = N'). Om overflow inte inträffar ger N ⊕ V värdet N eftersom N ⊕ 0 = N.

Relation	N⊕V	Z	Sant om (Korrekt teckenbit)
X > Y	0	0	(N ⊕ V)'· Z' = 1
X = Y	0	1	Z = 1
X < Y	1	0	N ⊕ V = 1

Observera att fallet N ⊕ V = Z = 1 inte inträffar och därför kan betraktas som "don't care".

Nedan formuleras hoppvillkoren för de sex villkorliga hopp som undersöker de sex storleksrelationerna mellan X och Y, om X och Y tolkas som tal med inbyggt tecken.

Relation	Villkorlig hoppinstruktion	Hoppvillkor HV
X > Y	BGT (Branch if X is greater than Y)	(N ⊕ V)'· Z'
X ≥ Y	BGE (Branch if X is greater than or equal to Y)	(N ⊕ V)'
X = Y	BEQ (Branch if X is equal to Y)	Z
X ≠ Y	BNE (Branch if X is not equal to Y)	Z'
X ≤ Y	BLE (Branch if X is less than or equal to Y)	((N ⊕ V)'· Z')' = (N ⊕ V) + Z
X < Y	BLT (Branch if X is less than Y)	N ⊕ V

Exempel på stackanvändning med FLIS-processorn

Minnesadress (Hex)

```
10:       LDSP   #$F0
12:
 .          .
 .          .
17:       LDX    #$50     }  Huvudprogram
19:       CLRA
1A:       JSR    $40
1C:       STA    $E0
1E:
 .          .
 .          .
40:       PSHA
41:       PSHX
42:       PSHC
43:       LDX    #$E0
45:
 .          .
 .          .
4D:       LDA    #$03     }  Subrutin 1
4F:       JSR    $60
51:
 .          .
 .          .
          PULC
          PULX
          PULA
          RTS
             .
             .
60:       PSHA
61:       PSHC
62:       PSHX
 .        LDX    #$15
 .        INCA
 .           .             }  Subrutin 2
 .           .
          PULX
          PULC
          PULA
          RTS
 .
 .
```

Extra programmeringsuppgifter för FLIS-processorn

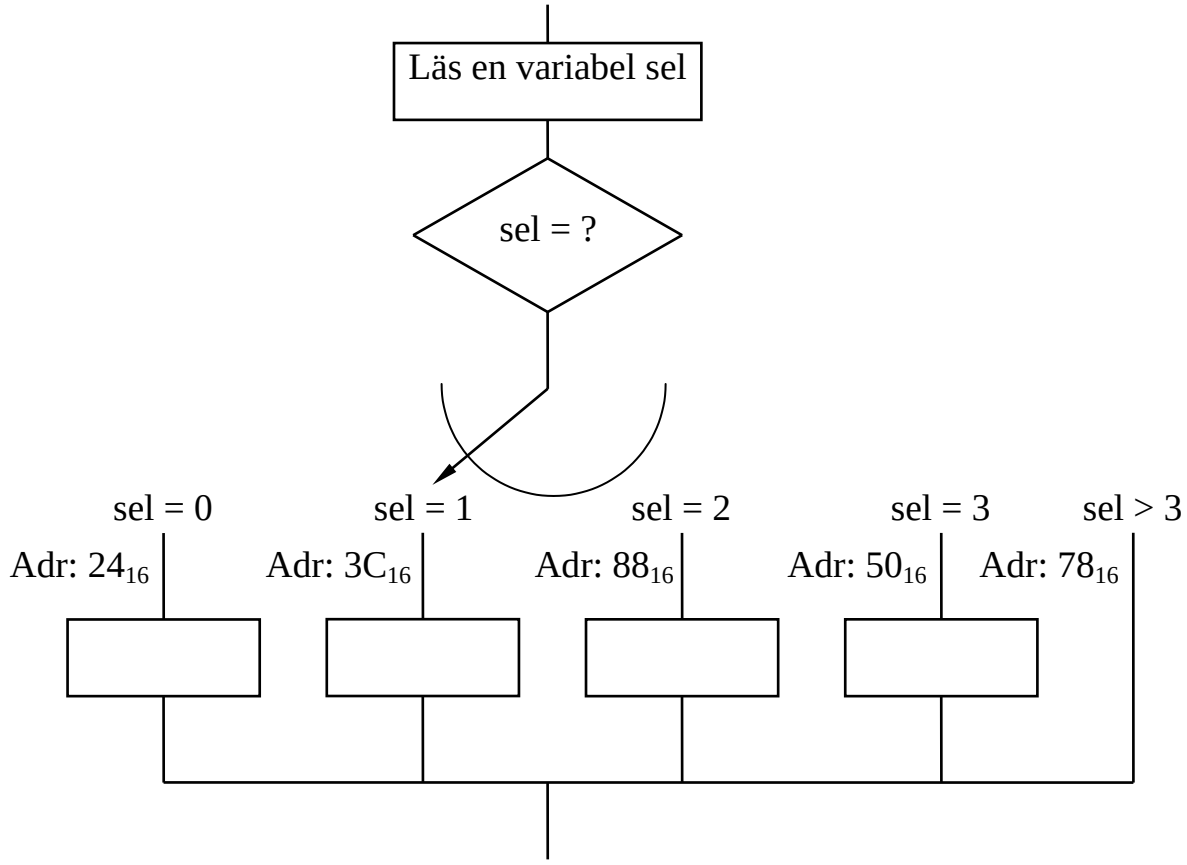
X.15

a) Skriv en instruktionssekvens för FLIS-processorn som nollställer bit 3-0 i alla minnesord i adressintervallet [35_{16}, 49_{16}]. Använd X-registret för adressering.

b) Skriv en subrutin för FLIS-processorn som maskerar (nollställer) bit 7 i varje byte inom adressområdet 40_{16}-$5F_{16}$.

X.16 Skriv en instruktionssekvens för FLIS-processorn som adderar det fjärde och sjunde elementet i en tabell med åttabitars ord och placerar summan i tabellen som det femte elementet. Det gamla element fem skall skrivas över. Använd X-registret för adressering.

X.17 Skriv en instruktionssekvens för FLIS-processorn som implementerar ett flerval enligt flödesplanen nedan.



X.18 För vilka värden på W utförs hoppet nedan? Betrakta W som ett tal [0,255].

```
LDA      #$85
CMPA     #W
B(Villkor)  Hopp
```

om det villkorliga hoppet är

a) BHI
b) BHS
c) BLS
d) BLO
e) BGT
f) BGE
g) BLE
h) BLT

X.19 Skriv en subrutin som har ett binärt tal i A-registret som indata.

Om talet tillhör talområdet $[0000,1111]_2$ skall subrutinen returnera ASCII-tecknet för motsvarande hexadecimala siffra som utdata i A-registret.

Om talet är större än $(1111)_2$ skall subrutinen returnera talet FF_{16} i A-registret.

De värden som finns i alla register, utom A och PC, vid anrop av subrutinen måste vara oförändrade vid återhopp från subrutinen.

X.20 Skriv en subrutin som nollställer en specificerad bit i dataordet på en adress i minnet.

Vid anrop av subrutinen finns dataordets minnesadress i X-registret och bitnumret på den bit som skall nollställas i A-registret. Om bitnumret är större än 7 skall dataordet ej påverkas.

De värden som finns i alla register, utom PC, vid anrop av subrutinen måste vara oförändrade vid återhopp från subrutinen.

Exempel på anrop:

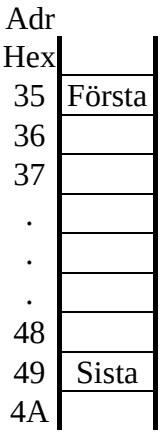
```
LDA     #5
LDX     #$20
JSR     BITZERO     Bit nr 5 i dataordet på adress 20₁₆ i minnet nollställs
```

Lösningar extrauppgifter X.15 - X.20

X.15a)

Lösning 1:

```
        .
        .
        LDX   #$35          Sätt pekare till tabellen
        LDA   #$15          Sätt varvräknare till antal dataord
        STA   COUNT         Startvärde för räknare i minnet
LOOP    LDA   0,X           Hämta data från tabellen
        ANDA  #%11110000    Nollställ bit 3-0
        STA   ,X+           Skriv tillbaka till tabell. Öka pekare
        DEC   COUNT         Minska varvräknare i minnet med ett
        BNE   LOOP          Fortsätt med nästa värde
        .                   Färdigt, fortsätt med något annat
        .
```



Lösning 2:

```
        .
        .
        LDX   #$35          Sätt pekare till tabellen
        LDA   #$15          Sätt varvräknare till antal dataord
        PSHA                Startvärde till stacken
LOOP    LDA   0,X           Hämta data från tabellen
        ANDA  #%11110000    Nollställ bit 3-0
        STA   ,X+           Skriv tillbaka till tabell. Öka pekare
        DEC   0,SP          Minska varvräknare på stacken med ett
        BNE   LOOP          Fortsätt med nästa värde
        PULA                (eller LEASP 1,SP) Återställ stackpekare
        .                   Färdigt, fortsätt med något annat
        .
```

Lösning 3:

```
        .
        .
        LDX   #$35          Sätt pekare till tabellen
LOOP    LDA   0,X           Hämta data från tabellen
        ANDA  #%11110000    Nollställ bit 3-0
        STA   ,X+           Skriv tillbaka till tabell. Öka pekare
        CMPX  #$4A          Kontrollera om pekaren har passerat hela tabellen
        BNE   LOOP          Nej, fortsätt med nästa värde i tabellen
        .                   Färdigt, fortsätt med något annat
        .
```

X.15b)
Lösning

```
MASKBIT  PSHA
         PSHC
         PSHX
         LDX   #$40      Startadress
LOOP     LDA   0,X       Hämta byten
         ANDA  #$7F      Nollställ ms bit
         STA   ,X+       Skriv tillbaks byte
         CMPX  #$60      Klart?
         BNE   LOOP      Om inte: fortsätt
MASKBIT  PULX
         PULC
         PULA
         RTS
```

X.16
Lösning

Numrera elementen från noll och uppåt, dvs första elementet har nummer 0, andra har nummer 1 osv.

Adress	-
Tabell+0	Första
Tabell+1	Andra
Tabell+2	Tredje
Tabell+3	Fjärde
Tabell+4	Femte
Tabell+5	Sjätte
Tabell+6	Sjunde
Tabell+7	Åttonde

```
         .
         .
LDX      #Tabell    Sätt pekare till tabellen
LDA      3,X        Hämta fjärde elementet från tabellen till A-reg
ADDA     5,X        Addera sjätte elementet från tabellen till fjärde elementet i A-reg
STA      4,X        Placera summan som femte element i tabellen
         .
         .
```

X.17
Lösning 1:

```
        .
LDA     SEL     Läs variabeln SEL från minnet
CMPA    #$03    Kontrollera om SEL > 3 eller SEL = 3
BHI     $78     SEL är > 3. Hoppa till adressen 78(16) enligt graf
BEQ     $50     SEL är = 3. Hoppa till adressen 50(16) enligt graf
TSTA            Kontrollera om SEL = 0
BEQ     $24     SEL är = 0. Hoppa till adressen 24(16) enligt graf
CMPA    #$01    Kontrollera om SEL = 1
BEQ     $3C     SEL är = 1. Hoppa till adressen 3C(16)
BRA     $88     SEL måste vara = 2. Hoppa till adressen 88(16)
        .
        .
```

I lösning 2 antas att hoppadresserna finns lagrade i en tabell i minnet på adressen JUMPTAB.

Adress	Data (Hex)
	-
JUMPTAB	24
JUMPTAB+1	3C
JUMPTAB+2	88
JUMPTAB+3	50

Lösning 2:

```
        .
LDA     SEL     Läs variabeln SEL från minnet
CMPA    #$03    Kontrollera om SEL > 3
BHI     $78     SEL är > 3. Hoppa till adressen 78(16) enligt graf

LDX     #JUMPTAB  SEL är ≤ 3. Sätt adress till tabell med hoppadresser
LDA     A,X       Hämta hoppadress till A från JUMPTAB
PSHA              Skriv hoppadressen i A på stack
PULX              Hämta hoppadressen på stacken till X
JMP     0,X       Hoppa till den adress som hämtades från JUMPTAB
        .
        .
        .
```

Istället för att använda stacken för att kopiera hoppadressen med "push" och "pull" i lösning 2 kan man mellanlagra den på en adress i minnet.

X.18

Lösning

```
LDA       #$85      Talet 85₁₆ till A
CMPA      #W        Bilda 85₁₆ - W
B(Villkor) Hopp
```

Storleksjämförelse görs alltså mellan 85_{16} = 133 och W.

Alla hoppvillkoren i a - d avser tal utan inbyggt tecken.
För 8-bitars tal utan inbyggt tecken gäller: $0 \leq W \leq 255$

Om det villkorliga hoppet är:

a) BHI (Higher) utförs hoppet om $133 > W$, dvs $W < 133$
Svar: $0 \leq W < 133$

b) BHS (Higher or Same) utförs hoppet om $133 \geq W$, dvs $W \leq 133$
Svar: $0 \leq W \leq 133$

c) BLS (Lower or Same) utförs hoppet om $133 \leq W$, dvs $W \geq 133$
Svar: $133 \leq W \leq 255$

d) BLO (Lower) utförs hoppet om $133 < Y$, dvs $W > 133$
Svar: $133 < W \leq 255$

Alla hoppvillkoren i e - h avser tal med inbyggt tecken (2-komplementrepresentation).
För 8-bitars tal med inbyggt tecken (2-komplementrepresentation) gäller: $-128 \leq W \leq +127$
Talet 85_{16} tolkas som det negativa talet $-(100_{16} - 85_{16}) = -7B_{16} = -123$

Om det villkorliga hoppet är:

e) BGT (Greater Than) utförs hoppet om $-123 > W$, dvs $W < -123$

W skall alltså tillhöra intervallet [-128, -123) Svar: $128 \leq W < 133$

f) BGE (Greater or Equal) utförs hoppet om $-123 \geq W$, dvs $W \leq -123$

W skall alltså tillhöra intervallet [-128, -123] Svar: $128 \leq W \leq 133$

g) BLE (Less or Equal) utförs hoppet om $-123 \leq W$, dvs $W \geq -123$

W skall alltså tillhöra intervallet [-123, +127] Svar: $0 \leq W \leq 127$ eller $133 \leq W \leq 255$

h) BLT (Less Than) utförs hoppet om $-123 < W$, dvs $W > -123$

W skall alltså tillhöra intervallet (-123, +127] Svar: $0 \leq W \leq 127$ eller $133 < W \leq 255$

Lösning: X.19

Alt 1:

```
HEXASC  PSHC
        CMPA  #$0F      Kontrollera övre gräns
        BHI   FEL       Talet större än övre gräns
        ADDA  #$30      Bilda ASCII-tecken för decimal siffra
        CMPA  #$39      Kontrollera övre gräns för decimal siffra
        BLS   EXIT      Siffra 0-9
        ADDA  #$07      Bilda ASCII-tecken för bokstav A-F
EXIT    PULC
        RTS
FEL     LDA   #$FF      Signalera att talet för stort med A=FF16
        BRA   EXIT
```

Alt 2:

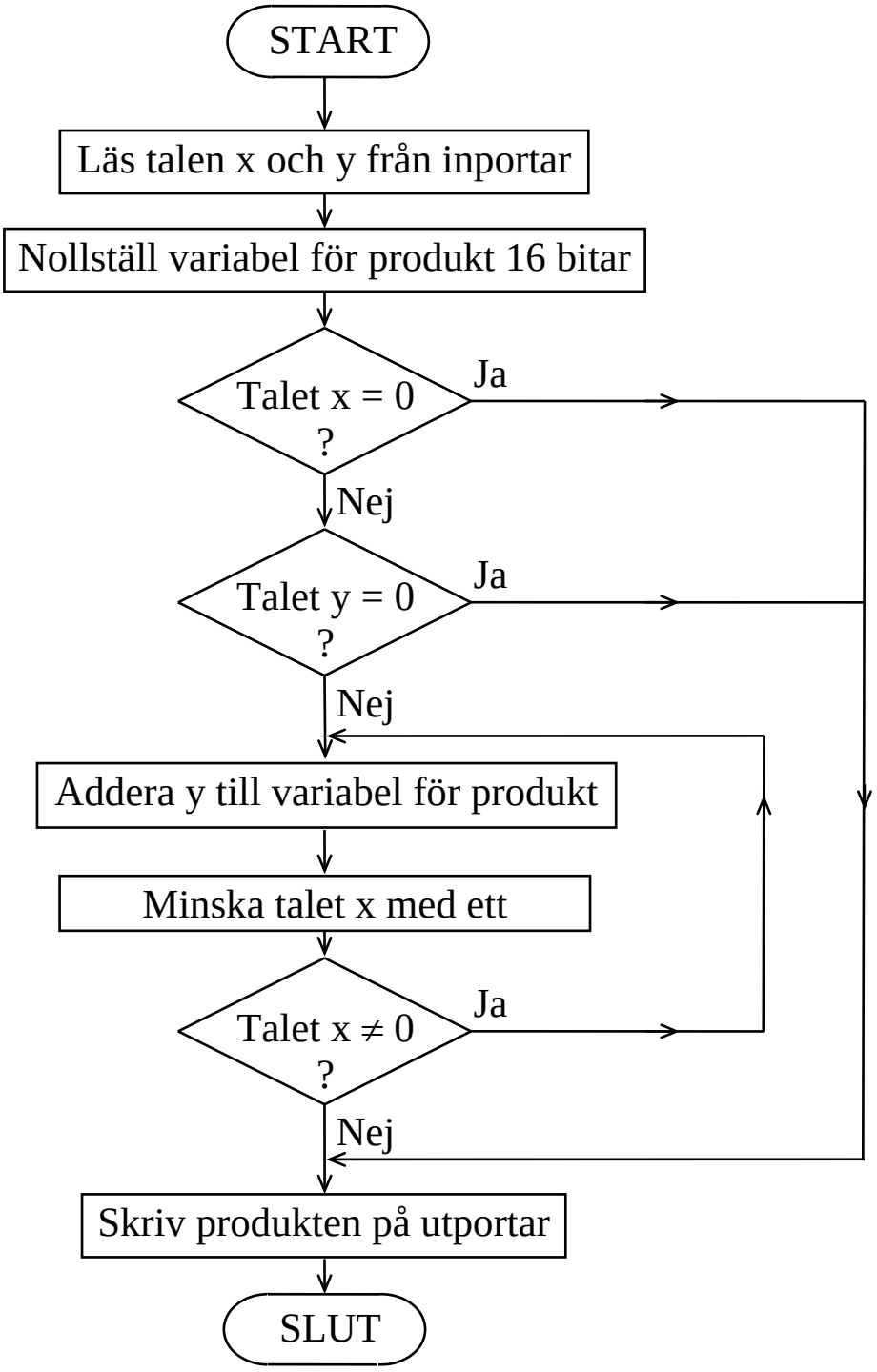
```
HEXASC  PSHX
        PSHC
        CMPA  #$0F      Kontrollera övre gräns
        BHI   FEL       Talet större än övre gräns
        LDX   #ASCTAB   Sätt pekare till tabell med samtliga 16 ASCII-tecken 0-F
        LDA   A,X       Hämta ASCII-tecken från tabell
EXIT    PULC
        PULX
        RTS
FEL     LDA   #$FF      Signalera att talet för stort med A=FF16
        BRA   EXIT

ASCTAB  FCB   $30       ASCII-tecken för 0
        FCB   $31       -"-          1
         .
         .
        FCB   $39       -"-          9
        FCB   $41       -"-          A
         .
         .
        FCB   $46       -"-          F
```

Lösning: X.20

```
BITZERO  PSHA                Spara register på stack
         PSHC
         PSHY

         CMPA  #$07          Kontrollera övre gräns för bitnummer
         BHI   EXIT          Bitnumret är större än 7

         LDY   #MSKTAB       Sätt pekare till tabell med masker
         LDA   A,Y           Hämta bitmask från tabell till A-reg
         ANDA  0,X           Nollställ bit i dataordet
         STA   0,X           Skriv tillbaka dataordet till minnet

EXIT     PULY                Hämta sparade register från stack
         PULC
         PULA
         RTS

MSKTAB   FCB   %11111110     Mönster för nollställning av bit nummer  0
         FCB   %11111101                    -"-                        1
         FCB   %11111011                    -"-                        2
         FCB   %11110111                    -"-                        3
         FCB   %11101111                    -"-                        4
         FCB   %11011111                    -"-                        5
         FCB   %10111111                    -"-                        6
         FCB   %01111111                    -"-                        7
```

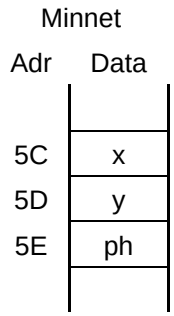
Ext-12 (Ver 2013-08-25)

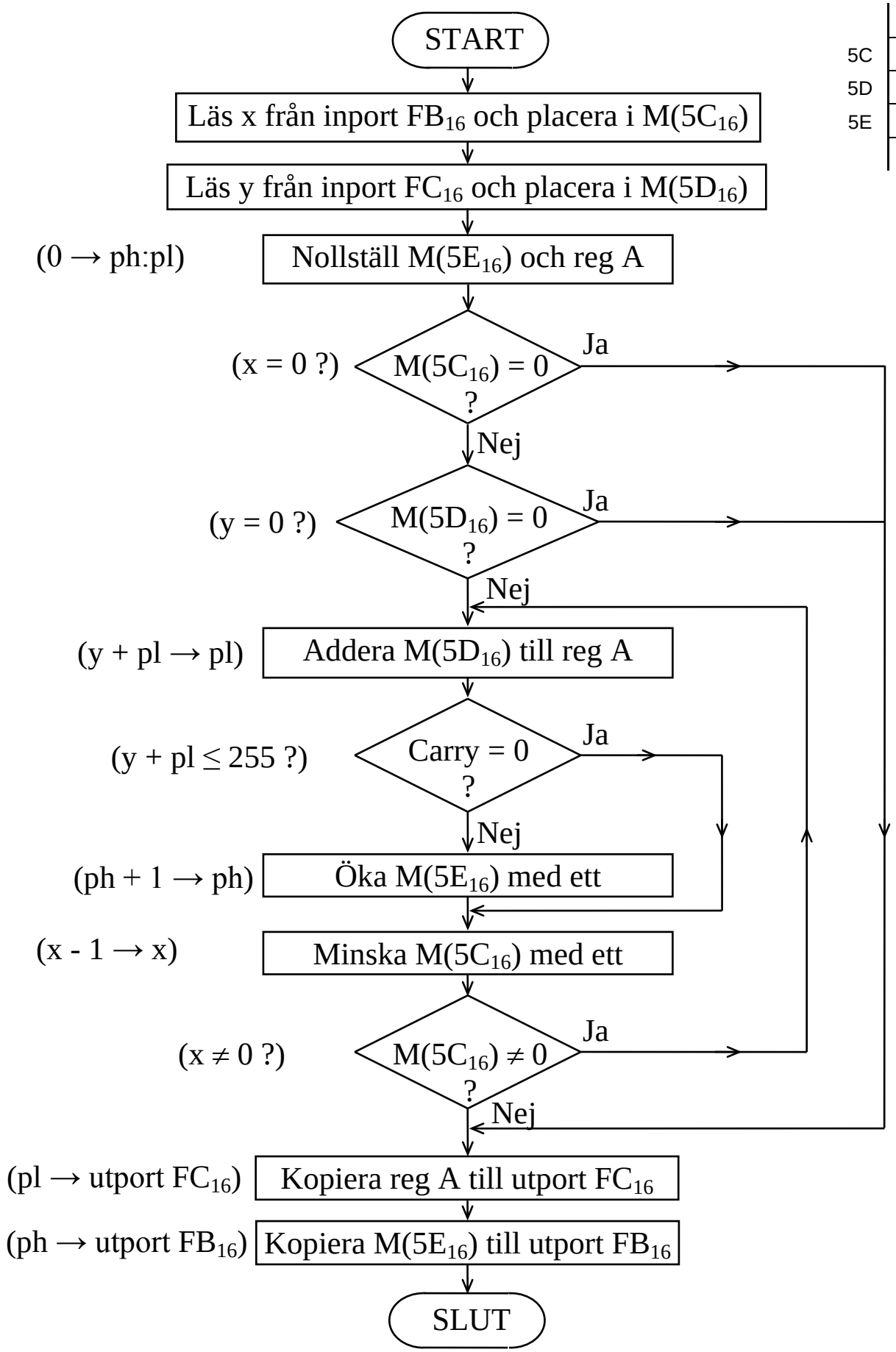
Normal flödesplan
(Multiplikation av 8-bitars tal genom upprepad addition)
x(8) · y(8) = p(16)



Maskinberoende flödesplan:

$x(8) \cdot y(8) = p(16) = ph(8):pl(8)$





Ext-12 (Ver 2013-08-25)

Läge	Operation	Operand	Kommentar
	LDA	$FB	Läs x-värdet till reg A
	STA	$5C	x-värdet till adr $5C_{16}$
	LDA	$FC	Läs y-värdet till reg A
	STA	$5D	y-värdet till adr $5D_{16}$
	CLR	$5E	Nollställ variabeln för produkten
	CLRA		-"-
	TST	$5C	Undersök talet x
	BEQ	OUTPUT	Talet x = 0. Hoppa ut
	TST	$5D	Undersök talet y
	BEQ	OUTPUT	Talet y = 0, Hoppa ut
ALOOP	ADDA	$5D	Addera talet y till reg A
	BCC	NOCY	Kontrollera om summan i reg A > 255
	INC	$5E	Ja, öka innehållet i produktens höga byte
NOCY	DEC	$5C	Minska talet x med ett
	BNE	ALOOP	Upprepa tills talet x = 0
OUTPUT	STA	$FC	Produktens låga byte till utport FC_{16}
	LDA	$5E	Läs produktens höga byte till reg A
	STA	$FB	Produktens höga byte till utport FB_{16}
SLUT	JMP	SLUT	

			Adr	Program	
	LDA	$FB	30	F1	FB
	STA	$5C	32	E1	5C
	LDA	$FC	34	F1	FC
	STA	$5D	36	E1	5D
	CLR	$5E	38	35	5E
	CLRA		3A	05	
	TST	$5C	3B	39	5C
	BEQ	OUTPUT	3D	24	0E
	TST	$5D	3F	39	5D
	BEQ	OUTPUT	41	24	0A
ALOOP	ADDA	$5D	43	A6	5D
	BCC	NOCY	45	29	02
	INC	$5E	47	37	5E
NOCY	DEC	$5C	49	38	5C
	BNE	ALOOP	4B	25	F6
OUTPUT	STA	$FC	4D	E1	FC
	LDA	$5E	4F	F1	5E
	STA	$FB	51	E1	FB
SLUT	JMP	SLUT	53	33	53

Ext-13 (Ver 2013-04-21)

Exempel på RTN-beskrivning av FLEX-instruktioner

1. Figur 1 på sidan 12 i detta häfte visar hur datorn FLEX är uppbyggd. På sidan 11 visas dessutom hur ALU:ns funktion väljs med styrsignalerna f_3 - f_0 och hur C_{in} väljs med styrsignalerna g_1 och g_0.

a) I tabellen nedan visas utförandefasen för en av FLEX-processorns instruktioner. Vilken instruktion har denna EXECUTE-sekvens? Skriv instruktionen med assemblerspråk.

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M+1→R, Flags→CC	MR, f_3, g_0, LD_R, LD_{CC}.
Q_8	R→M	OE_R, MW, (NF).

b) Rita en tabell motsvarande den i a) ovan, som visar utförandefasen för maskininstruktionen ANDA Adr för FLEX-processorn. I instruktionslistan för FLEX-processorn beskrivs instruktionen enligt följade tabell.

Instruktion		Adressering			Operationsbeskrivning	Flaggpåverkan			
Operation	Beteckning	Absolute				3	2	1	0
		OP	#	~		N	Z	V	C
AND	ANDA Adr	17	2	7	A AND M(Adr) → A	a	a	0	0

I tabellen anges antalet states till 7, varav 2 tillhör hämtfasen.

2. Figur 1 på sidan 12 i detta häfte visar hur datorn FLEX är uppbyggd. På sidan 11 visas dessutom hur ALU:ns funktion väljs med styrsignalerna f_3 - f_0 och hur C_{in} väljs med styrsignalerna g_1 och g_0.

I tabellen nedan visas styrsignalerna för utförandefasen för en av FLEX-processorns instruktioner. Dessutom visas RTN-beskrivningen för tillstånd 0 i EXECUTE-sekvensen.

a) Ge RTN-beskrivningen för de övriga tillstånden. Använd samma skrivsätt som i RTN-beskrivningen för den första raden.

b) Vilken instruktion har denna EXECUTE-sekvens? Skriv instruktionen med assemblerspråk.

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6		MR, LD_{MA}, LD_R.
Q_7		MR, LD_T.
Q_8		OE_R, f_3, f_2, g_0, LD_R, LD_{CC}.
Q_9		OE_R, MW, (NF).

c) Rita en tabell motsvarande den ovan, som visar utförandefasen för maskininstruktionen JMP Adr för FLEX-processorn. I instruktionslistan för FLEX-processorn beskrivs instruktionen enligt följande tabell.

Instruktion		Adressering			Operationsbeskrivning	Flaggpåverkan			
Operation	Beteckning	Absolute				3	2	1	0
		OP	#	~		N	Z	V	C
Unconditional jump	JMP Adr	59	2	4	Adr → PC	•	•	•	•

I tabellen anges antalet tillstånd till 4, varav 2 tillhör FETCH-sekvensen.

3. Figur 1 på sidan 12 i detta häfte visar hur datorn FLEX är uppbyggd. På sidan 11 visas dessutom hur ALU:ns funktion väljs med styrsignalerna f_3 - f_0 och hur C_{in} väljs med styrsignalerna g_1 och g_0.

I tabellen nedan visas styrsignalerna för utförandefasen för en av FLEX-processorns instruktioner. Dessutom visas RTN-beskrivningen för tillstånd 0 i EXECUTE-sekvensen.

a) Ge RTN-beskrivningen för de övriga tillstånden. Använd samma skrivsätt som i RTN-beskrivningen för den första raden.

b) Vilken instruktion har denna "execute"-fas? Skriv instruktionen med assemblerspråk.

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6		MR, LD_{MA}.
Q_7		MR, LD_T.
Q_8		OE_A, f_3, f_1, LD_R, LD_{CC}.
Q_9		OE_R, LD_A, (NF).

c) Rita en tabell motsvarande den ovan, som visar utförandefasen för maskininstruktionen BRA Adr för FLEX-processorn. I instruktionslistan för FLEX-processorn beskrivs instruktionen enligt följade tabell.

Instruktion		Adressering			Operations-beskrivning	Flagg-påverkan			
Operation	Beteckning	Relative				3	2	1	0
		OP	#	~		N	Z	V	C
Unconditional branch	BRA Adr	5A	2	5	PC+Offs → PC	•	•	•	•

I tabellen anges antalet states till 5, varav 2 tillhör hämtfasen.

Lösningar

1. a) I state Q_5 kopieras innehållet i PC till MA-registret. Detta innebär att adressen till minnesordet efter OP-koden hamnar i MA-registret. PC ökas med ett så att den pekar på nästa operationskod i minnet.

I state Q_6 läses minnesordet på adressen efter OP-koden och placeras i MA-registret. Detta innebär att instruktionens adressdel placeras i MA.

I state Q_7 läses minnesinnehållet som ökas med ett och placeras i R. Flaggorna uppdateras.

I state Q_8 skrivs det ökade minnesinnehållet tillbaka i minnet på den adress som finns i MA, dvs instruktionens adressdel och executesekvensen avslutas.

Detta innebär att instruktionen är en **INC Adr**

b) ANDA Adr `Ord1: 17H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	A AND T→R, Flags →CC	OE_A, f_2, f_1, LD_R, LD_{CC}.
Q_9	R→A	OE_R, LD_A, (NF).

(Efter det sista tillståndet i EXECUTE följer det första tillståndet i FETCH.)

Ext-13

2. a)

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA, 00H→R	MR, LD_{MA}, LD_R.
Q_7	M→T	MR, LD_T.
Q_8	R – T→R, Flags→CC	OE_R, f_3, f_2, g_0, LD_R, LD_{CC}.
Q_9	R→M	OE_R, MW, (NF).

(Efter det sista tillståndet i EXECUTE följer det första tillståndet i FETCH.)

b) I state Q_5 kopieras innehållet i PC till MA-registret. Detta innebär att adressen till minnesordet efter OP-koden hamnar i MA-registret. PC ökas med ett så att den pekar på nästa operationskod i minnet.

I state Q_6 läses minnesordet på adressen efter OP-koden och placeras i MA-registret. Detta innebär att instruktionens adressdel placeras i MA. Dessutom placeras värdet 00H i R-registret.

I state Q_7 läses minnesinnehållet som pekas ut av instruktionens adressdel och placeras i T-registret.

I state Q_8 bildas skillnaden 0 – M(Adr) = –M(Adr) som placeras i R-registret. Flaggorna uppdateras.

I state Q_9 skrivs –M(Adr) tillbaka i minnet, dvs – M(Adr) →M(Adr), och executesekvensen avslutas.

Detta innebär att instruktionen är en **NEG Adr**

c)

JMP Adr `Ord1: 59H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA	OE_{PC}, LD_{MA}.
Q_6	M→PC	MR, LD_{PC}, (NF).

(Efter det sista tillståndet i EXECUTE följer det första tillståndet i FETCH.)

3. **a)**

State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	A + T→R, Flags →CC	OE_A, f_3, f_1, LD_R, LD_{CC}.
Q_9	R→A	OE_R, LD_A, (NF).

(Efter det sista tillståndet i EXECUTE följer det första tillståndet i FETCH.)

b)

I state Q_5 kopieras innehållet i PC till MA-registret. Detta innebär att adressen till minnesordet efter OP-koden hamnar i MA-registret. PC ökas med ett så att den pekar på nästa operationskod i minnet.

I state Q_6 läses minnesordet på adressen efter OP-koden och placeras i MA-registret. Detta innebär att instruktionens adressdel placeras i MA.

I state Q_7 läses minnesinnehållet som pekas ut av instruktionens adressdel och placeras i T-registret.

I state Q_8 bildas summan A + M(Adr) som placeras i R-registret. Flaggorna uppdateras.

I state Q_9 kopieras summan från R-registret till A-registret och executesekvensen avslutas.

Detta innebär att instruktionen är en **ADDA Adr**

c) **BRA Adr** `Ord1: 5AH(Opkod) Ord2: Offset`

CP	State	RTN-beskrivning	Aktiva styrsignaler (=1)
Q_5	0	PC→MA, PC→T	OE_{PC}, LD_{MA}, LD_T.
Q_6	1	M+T+1→R	MR, f_3, f_1, g_0, LD_R.
Q_7	2	R→PC	OE_R, LD_{PC}, (NF).

(Efter det sista tillståndet i EXECUTE följer det första tillståndet i FETCH.)

Beskrivning av EXECUTE-fasen för vissa FLEX-instruktioner

(Efter det sista tillståndet i EXECUTE följer det första tillståndet i FETCH.)
Observera att dessa RTN-beskrivningar inte gäller för FLISP!

NOP `Ord1: 00H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5		(NF).

TFR A,B `Ord1: 01H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	A→B	OE_A, LD_B, (NF).

TFR A,CC `Ord1: 03H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	A→CC	OE_A, g_2, LD_{CC}, (NF).

EXG A,B `Ord1: 07H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	A→R	OE_A, f_0, LD_R.
Q_6	B→A	OE_B, LD_A.
Q_7	R→B	OE_R, LD_B, (NF).

LDAA #Data `Ord1: 0FH(Opkod) Ord2: Data`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→A	MR, LD_A, (NF).

STAA Adr `Ord1: 13H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	A→M	OE_A, MW, (NF).

ANDA #Data `Ord1: 19H(Opkod) Ord2: Data`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→T	MR, LD_T.
Q_7	A AND T→R, Flaggor→CC	OE_A, f_2, f_1, LD_R, LD_{CC}.
Q_8	R→A	OE_R, LD_A, (NF).

ORAB Adr `Ord1: 1CH(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	B OR T→R, Flaggor→CC	OE_B, f_2, f_0, LD_R, LD_{CC}.
Q_9	R→B	OE_R, LD_B, (NF).

COMA `Ord1: 23H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	A'→R, Flaggor→CC	OE_A, f_1, f_0, LD_R, LD_{CC}.
Q_6	R→A	OE_R, LD_A, (NF).

ANDCC #Data `Ord1: 26H(Opkod) Ord2: Data`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→T	MR, LD_T.
Q_7	CC AND T→R	OE_{CC}, f_2, f_1, LD_R.
Q_8	R→CC	OE_R, LD_{CC}, g_2, (NF).

ADDB Adr `Ord1: 29H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	B+T→R, Flaggor→CC	OE_B, f_3, f_1, LD_R, LD_{CC}.
Q_9	R→B	OE_R, LD_B, (NF).

ADCA #Data `Ord1: 2EH(Opkod) Ord2: Data`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→T	MR, LD_T.
Q_7	A+T+C→R, Flaggor→CC	OE_A, f_3, f_1, g_1, LD_R, LD_{CC}.
Q_8	R→A	OE_R, LD_A, (NF).

SUBA Adr `Ord1: 30H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	A-T→R, Flaggor→CC	OE_A, f_3, f_2, g_0, LD_R, LD_{CC}.
Q_9	R→A	OE_R, LD_A, (NF).

SBCA Adr `Ord1: 34H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	A-T-C→R, Flaggor→CC	OE_A, f_3, f_2, g_1, g_0, LD_R, LD_{CC}.
Q_9	R→A	OE_R, LD_A, (NF).

NEG Adr `Ord1: 3AH(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA, 00H→R	MR, LD_{MA}, LD_R.
Q_7	M→T	MR, LD_T.
Q_8	R-T→R, Flaggor→CC	OE_R, f_3, f_2, g_0, LD_R, LD_{CC}.
Q_9	R→M	OE_R, MW, (NF).

ASLA `Ord1: 3BH(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	2A→R, Flaggor→CC	OE_A, f_3, f_1, f_0, LD_R, LD_{CC}.
Q_6	R→A	OE_R, LD_A, (NF).

ASL Adr `Ord1: 3DH(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	2M→R, Flaggor→CC	MR, f_3, f_1, f_0, LD_R, LD_{CC}.
Q_8	R→M	OE_R, MW, (NF).

ROLA `Ord1: 3EH(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	2A+C→R, Flaggor→CC	OE_A, f_3, f_1, f_0, g_1, LD_R, LD_{CC}.
Q_6	R→A	OE_R, LD_A, (NF).

ROL Adr `Ord1: 40H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	2M+C→R, Flaggor→CC	MR, f_3, f_1, f_0, g_1, LD_R, LD_{CC}.
Q_8	R→M	OE_R, MW, (NF).

INCA `Ord1: 41H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	A+1→R, Flaggor→CC	OE_A, f_3, g_0, LD_R, LD_{CC}.
Q_6	R→A	OE_R, LD_A, (NF).

CLRA `Ord1: 47H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	00H→R, Flaggor→CC	LD_R, LD_{CC}.
Q_6	R→A	OE_R, LD_A, (NF).

CMPA #Data `Ord1: 4EH(Opkod) Ord2: Data`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→T	MR, LD_T.
Q_7	A-T, Flaggor→CC	OE_A, f_3, f_2, g_0, LD_{CC}, (NF).

TSTA `Ord1: 52H(Opkod)`

State	RTN-beskrivning	Styrsignaler
Q_5	A→R, Flaggor→CC	OE_A, LD_R, f_0, LD_{CC}, (NF).

BITA Adr `Ord1: 55H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC.
Q_6	M→MA	MR, LD_{MA}.
Q_7	M→T	MR, LD_T.
Q_8	A AND T, Flaggor→CC	OE_A, f_2, f_1, LD_{CC}, (NF).

JMP Adr `Ord1: 59H(Opkod) Ord2: Adr`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA	OE_{PC}, LD_{MA}.
Q_6	M→PC	MR, LD_{PC}, (NF).

BRA Adr `Ord1: 5AH(Opkod) Ord2: Offset`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC→T	OE_{PC}, LD_{MA}, LD_T.
Q_6	M+T+1→R	MR, f_3, f_1, g_0, LD_R.
Q_7	R→PC	OE_R, LD_{PC}, (NF).

BMI Adr `Ord1: 5BH(Opkod) Ord2: Offset`

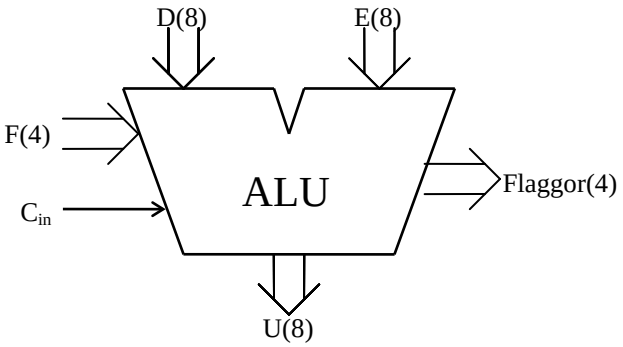
State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC→T	OE_{PC}, LD_{MA}, LD_T.
Q_6	M+T+1→R, PC+1→PC	MR, f_3, f_1, g_0, LD_R, IncPC.
Q_7	If N=1: R→PC; else: (Ingenting)	OE_R, LD_{PC}=N, (NF).

BPL Adr `Ord1: 5CH(Opkod) Ord2: Offset`

State	RTN-beskrivning	Styrsignaler
Q_5	PC→MA, PC→T	OE_{PC}, LD_{MA}, LD_T.
Q_6	M+T+1→R, PC+1→PC,	MR, f_3, f_1, g_0, LD_R, IncPC.
Q_7	If N=0: R→PC, (NF); else: (Ingenting)	OE_R, LD_{PC}=N', (NF).

Bilaga 1

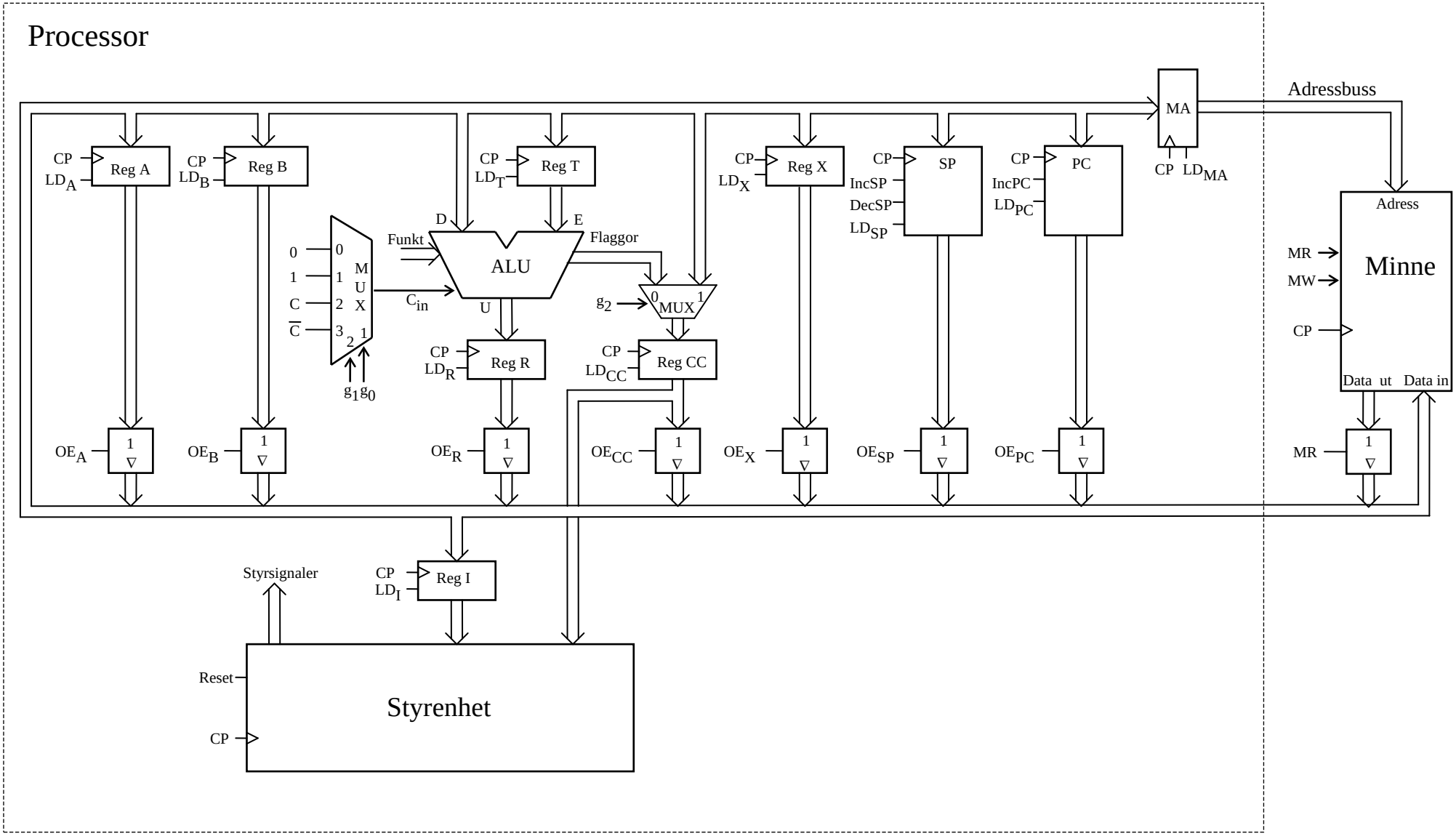
ALU:ns funktion



ALU:ns logiska och aritmetiska operationer på indata D och E definieras av ingångarna $F = (f_3, f_2, f_1, f_0)$ och C_{in} enligt tabellen nedan.

f_3 f_2 f_1 f_0	$U = f(D,E,C_{in})$
0 0 0 0	0
0 0 0 1	D
0 0 1 0	E
0 0 1 1	D'
0 1 0 0	E'
0 1 0 1	D OR E
0 1 1 0	D AND E
0 1 1 1	D XOR E
1 0 0 0	$D + C_{in}$
1 0 0 1	$D - 1 + C_{in}$
1 0 1 0	$D + E + C_{in}$
1 0 1 1	$D + D + C_{in}$
1 1 0 0	$D - E - 1 + C_{in}$
1 1 0 1	0
1 1 1 0	0
1 1 1 1	FFH

I tabellen ovan avser "+" och "-" aritmetiska operationer.
Med t ex D' menas att samtliga bitar i D inverteras.



Figur 1. Datorn FLEX.

Ext-13

Ext-14

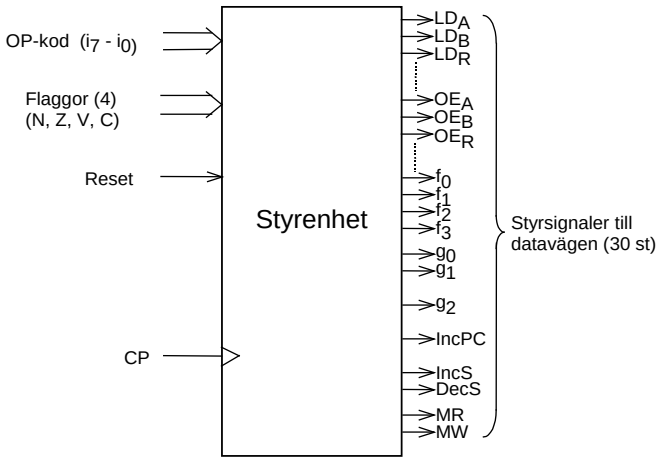
FLEX-processorns styrenhet med fast logik

FLEX-processorns styrenhet med fast logik

En styrenhet för FLEX-processorn skall kunna generera alla styrsignaler till datavägen och minnet för samtliga instruktioner i instruktionslistan. Under körning av ett program måste därför ett stort antal olika styrsignalsekvenser av varierande längd och komplexitet genereras.

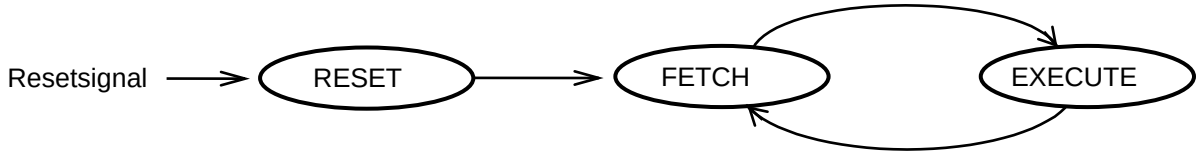
Vi har tidigare sett hur datavägen och minnet kan styras genom att styrsignalerna först ges rätta värden varpå en klockpuls CP (positiv flank) verkställer laddning av ett eller flera utvalda register. Efter klockpulsflanken byter man till nya styrsignalvärden och verkställer nästa laddning med nästa klockpulsflank osv. Byte av styrsignaler och laddning av register sker alltså i samma takt. Det är därför lämpligt att använda samma klockpulssignal CP till datavägen, minnet och styrenheten. Styrenheten kommer därför att utgöras av ett synkront sekvensnät med CP som klocksignal och samtliga styrsignaler till datavägen och minnet som utsignaler.

Som insignaler till styrenheten används innehållet i instruktionsregistret (dvs OP-koden) och innehållet i flaggregistret (dvs flaggorna). Styrenheten behöver OP-koden eftersom den bestämmer utförandefasen (EXECUTE) av instruktionen. Flaggorna behövs när en villkorlig instruktion skall utföras. För att processorn skall kunna startas eller återstartas behövs även en insignal (startsignal) Reset. Figur 1 visar styrenheten utifrån sett.



Figur 1. Styrenhet till FLEX-processorn.

En lämplig utgångspunkt för att bestämma styrenhetens interna uppbyggnad är den grova flödesplanen nedan. Den visar hur växling sker mellan interna tillstånd när instruktioner exekveras. Först konstateras att processorn måste starta från ett starttillstånd och utföra en startsekvens (RESET). Därefter skall den övergå till hämtfasen (FETCH), som följs av utförandefasen (EXECUTE) för den instruktion vars OP-kod finns i instruktionsregistret. Under normalt arbete skall den sedan ständigt växla mellan hämtfasen (FETCH) och utförandefasen (EXECUTE) enligt grafen.



Tillståndsgraf för styrenhet

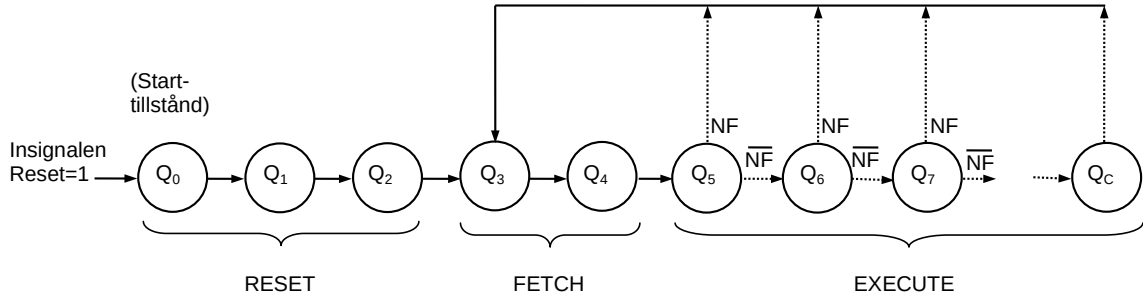
I häftet Ext-8 visas att RESET-sekvensen kräver tre tillstånd och FETCH-sekvensen två tillstånd.

Varje instruktion har ett unikt beteende i sin EXECUTE-sekvens, som därmed blir olika lång. Sekvensens längd bestäms ju av det nyttiga arbete den aktuella instruktionen skall utföra.

Vi antar till en början att det räcker med åtta tillstånd för att utföra allt som behöver göras i EXECUTE-fasen för den längsta instruktionen.

Maximala antalet tillstånd som styrenheten behöver genomlöpa är i så fall 3 + 2 + 8 = 13.

Figur 2 visar RESET-, FETCH- och EXECUTE-sekvensen. De 13 tillstånden har döpts Q_0 – Q_C.



Figur 2. Tillståndsgraf för FLEX-processorns styrenhet.

Av figur 2 framgår också att en ny styrsignal NF (Next FETCH) har införts, detta för att förenkla övergången från det sista tillståndet i en EXECUTE-sekvens till det första tillståndet Q_3 i FETCH-sekvensen. NF används enbart internt inom styrenheten. I det sista tillståndet i en EXECUTE-sekvens skall styrsignalen NF ges värdet 1.

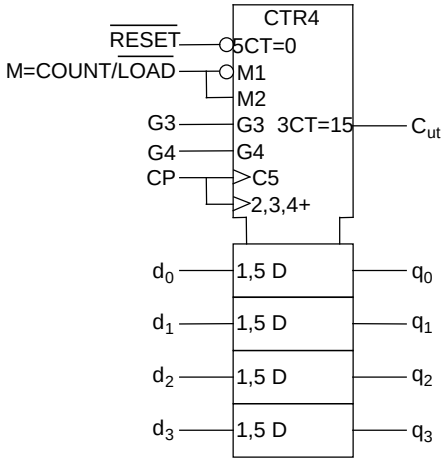
I varje tillstånd aktiveras ett antal av de sammanlagt (30 + 1) st styrsignalerna. Varje styrsignal är en Boolesk funktion av tillståndsvariablerna som definierar tillstånden (ringarna) och OP-koden. Eftersom det finns villkorliga instruktioner ingår även flaggorna i de Booleska uttrycken för vissa av styrsignalerna.

I tillståndsgrafen skall egentligen också finnas övergångar (pilar) från varje ring till ringen längst till vänster, dvs starttillståndet. Övergång till starttillståndet skall ske så fort styrenhetens insignal Reset = 1 vid positiv klockpulsflank. Detta är den enda påverkan insignalen Reset har på styrenheten.

Realisering av styrenhet med fast logik

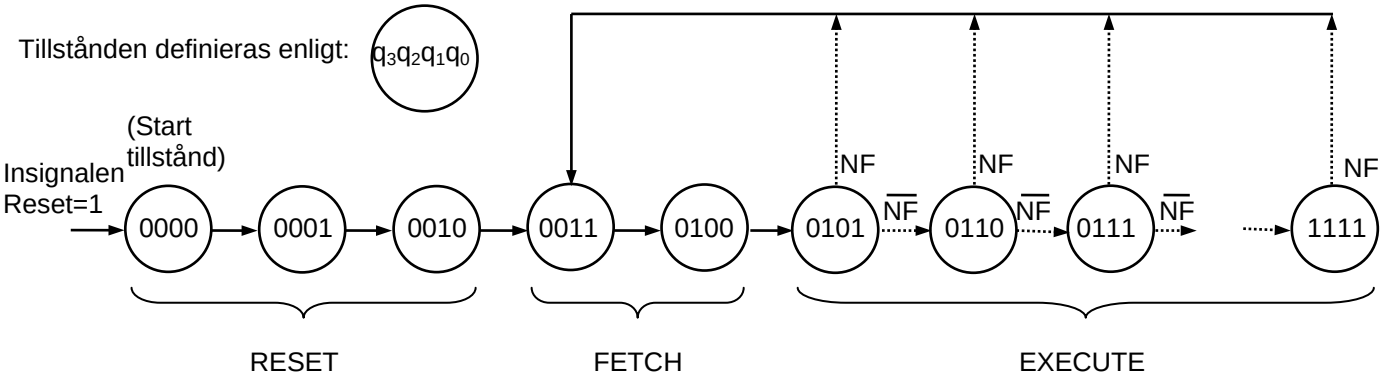
Det framgår av tillståndsgrafen i figur 2 att styrenheten liknar en räknare. Efter återställning, med insignalen Reset = 1, startar den från tillstånd Q_0 och räknar sedan uppåt genom RESET, FETCH- och EXECUTE-sekvensen. När det sista tillståndet i EXECUTE-sekvensen för den aktuella instruktionen har nåtts startar den om från första tillståndet i FETCH (Q_3).

Den laddningsbara 4-bitars räknaren 74HC163, som användes i arbetshäfte nr 2, passar utmärkt för realisering av styrenheten. Räknarens symbol visas i figur 3.



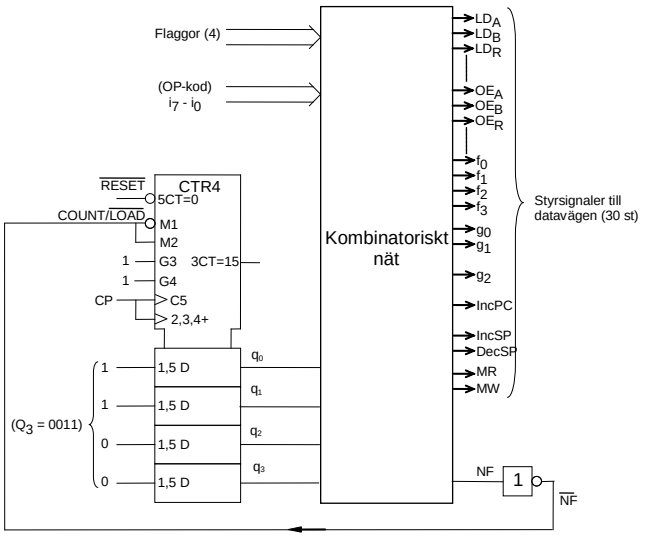
Figur 3. 4-bitars räknaren 74HC163.

Om man kodar tillstånden Q_i i tillståndsgrafen i figur 2 som $q_3q_2q_1q_0$ och väljer Q_0 – Q_C som 0000 -1100, får man samma tillståndssekvens som i räknaren. Tillståndsgrafen visas i figur 4, där EXECUTE-sekvensen dessutom har kompletterats med de tre extra tillstånden Q_D – Q_F, som man får på köpet genom att använda räknarens samtliga tillstånd.



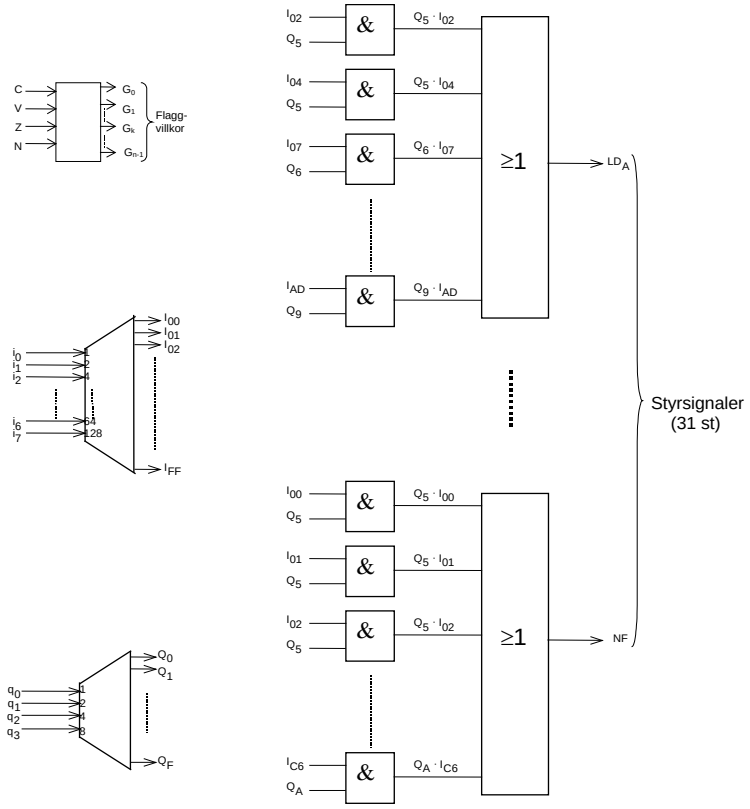
Figur 4. Exempel på tillståndskodning för FLEX-processorns automatiska styrenhet.

Kretsrealiseringen för denna tillståndsgraf blir mycket enkel och visas i figur 5.



Figur 5. Enkel realisering av FLEX-processorns automatiska styrenhet.

Det kombinatoriska nätet i figur 5 är uppbyggt enligt principen i figur 6. Förutom avkodning av tillståndsräknare och instruktionernas OP-koder innehåller det en s k AND-OR area där styrsignalerna skapas som (Booleska) summor av produkttermer. Produkttermerna består av tillståndsvariablerna q_0 - q_3, OP-kodbitarna i_0 - i_7 och eventuellt flaggbitarna om en villkorlig instruktion skall utföras.



Figur 6 Princip för kombinatorisk del av styrenhet med fast kopplad logik.

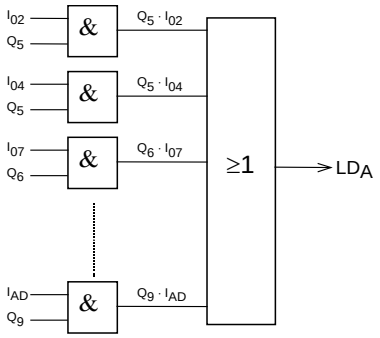
Av figur 6 framgår att räknarens utsignaler q_3, q_2, q_1 och q_0 är kopplade till en 4/16-avkodare med de sexton utsignalerna Q_0 – Q_F. Detta innebär t ex att när räknarens tillstånd $q_3q_2q_1q_0$ = $(0011)_2$ = 3, är Q_3 = 1, medan övriga Q-värden från avkodaren är 0. Det framgår också att

instruktionsregistrets åtta bitar i_0 - i_7 är kopplade till en 8/256-avkodare med de 256 utsignalerna I_{00} -I_{FF}.

Genom att studera hur styrsignalen LD_A bildas i det kombinatoriska nätet i figur 6 får man en god uppfattning om hur hela nätet är konstruerat.

Styrsignalen LD_A, är utsignal från en OR-grind med ett stort antal ingångar enligt figur 7 nedan. Den är alltså A-registrets "load signal" och måste vara aktiv (=1) när A-registret skall laddas med ett nytt datavärde från bussen. Vid alla andra tillfällen skall den vara passiv (=0).

Samtliga instruktioner som laddar register A måste därför generera en etta på OR-grindens utgång under det tillstånd då laddningen skall ske.



Motsvarande Booleska uttryck:

$LD_A = Q_5 \cdot I_{02} + Q_5 \cdot I_{04} + Q_6 \cdot I_{07} + \ldots + Q_9 \cdot I_{AD}$

Figur 7 Logiknät och motsvarande Booleska uttryck för styrsignalen LD_A.

Av figur 7 kan man se att den översta AND-grinden (och därmed OR-grinden som bildar styrsignalen LD_A) ger utsignalen 1 om $Q_5 \cdot I_{02} = 1$.

Register A laddas alltså när styrenheten befinner sig i tillstånd Q_5 och OP-koden 02H samtidigt finns i instruktionsregistret. Eftersom Q_5 är det första tillståndet i EXECUTE laddas register A i det första tillståndet i EXECUTE för instruktionen med OP-koden 02H.

Nästa AND-grind ger utsignalen 1 om $Q_5 \cdot I_{04} = 1$.
Register A laddas när styrenheten befinner sig i tillstånd Q_5 och OP-koden 04H samtidigt finns i instruktionsregistret. Här laddas register A i det första tillståndet i EXECUTE för instruktionen med OP-kod 04H.

Följande AND-grind ger utsignalen 1 om $Q_6 \cdot I_{07} = 1$.
Register A laddas när styrenheten befinner sig i tillstånd Q_6 och OP-koden 07H samtidigt finns i instruktionsregistret. Här laddas register A i det andra tillståndet i EXECUTE för instruktionen med OP-kod 07H.

Den nedersta AND-grinden (och OR-grinden) ger utsignalen 1 om $Q_9 \cdot I_{AD} = 1$.
Register A laddas när styrenheten befinner sig i tillstånd Q_9, d v s det femte tillståndet i EXECUTE för instruktionen med OP-kod ADH.

Eftersom A-registret endast laddas i EXECUTE-fasen (av de instruktioner som påverkar A), består samtliga termer i summan som bildar LD_A av en produkt $Q_i \cdot I_j$, där Q_i (i = 5, …, FH), är en av utsignalerna från tillståndsavkodaren i figur 6 och I_j, (j = 0, …, FFH), är en av utsignalerna från instruktionsavkodaren i samma figur.

Implementering av styrenhet för fyra instruktioner

För att illustrera principen för styrenheten i föregående avsnitt skall vi nu konstruera en styrenhet som klarar de fyra instruktioner som ingår i programmet nedan

```
        LDAA    #$90
LOOP    INCA
        STAA    $05
        JMP     LOOP
```

Styrenheten skall alltså klara av RESET-sekvensen (för start av programmet), FETCH-sekvensen och EXECUTE-sekvenserna för de fyra instruktionerna i programmet.

En lämplig utgångspunkt vid konstruktionen är att beskriva samtliga sex sekvenser som behövs med RTN och styrsignaler.

RESET- och FETCH-sekvensen har beskrivits tidigare och återges i tabellform nedan.

RESET:

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler (=1)
Q_0	Q_0	FFH→R	f_3, f_2, f_1, f_0, LD_R
Q_1	Q_1	R→MA	OE_R, LD_{MA}
Q_2	Q_2	M→PC	MR, LD_{PC}

FETCH:

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler (=1)
Q_3	Q_3	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC
Q_4	Q_4	M→I	MR, LD_I

EXECUTE-sekvensen för LDAA #Data visas nedan.

EXECUTE för LDAA #Data: (OP-kod = 0FH)

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler (=1)
Q_5	$Q_5 \cdot I_{0F}$	PC→MA, PC+1→PC	OE_{PC}, LD_{MA}, IncPC
Q_6	$Q_6 \cdot I_{0F}$	M→A, (Next Fetch)	MR, LD_A, NF

I kolumnen *Tillstånd* visas styrenhetens tillstånd. I tabellerna ingår den nya kolumnen *Summaterm*, som visar summaterm $Q_i \cdot I_j$ enligt principen i figur 7.

Övningsuppgift 1a: Fyll i RTN-beskrivning och styrsignaler för EXECUTE-sekvensen för instruktionen INCA i tabellen nedan. Observera att det första tillståndet i EXECUTE alltid är Q_5 och att styrsignalen NF alltid skall vara aktiv i det sista tillståndet i EXECUTE. Tänk också på att INCA skall påverka flaggorna!

EXECUTE för INCA: (OP-kod = 41H)

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler (=1)
Q_5	$Q_5 \cdot I_{41}$		
Q_6	$Q_6 \cdot I_{41}$		

¤¤¤

Övningsuppgift 1b: Fyll i OP-kod, Tillstånd, Summaterm, RTN-beskrivning och Styrsignaler i tabellerna nedan. Observera att det första tillståndet i EXECUTE alltid är Q_5.

Styrenheten skall alltid återvända till första tillståndet i FETCH efter det sista tillståndet i EXECUTE. Därför måste man sätta NF=1 i det sista tillståndet i EXECUTE.

Antalet klockcykler (states) för EXCUTE-sekvenserna får man fram genom att slå upp antalet states för respektive instruktion i instruktionslistan för FLEX-processorn och sedan dra bort antalet states i FETCH-sekvensen. På detta sätt får man 5 - 2 = 3 för STAA Adr och 4 - 2 = 2 för JMP Adr

EXECUTE för STAA Adr: (OP-kod = H)

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler (=1)

EXECUTE för JMP Adr: (OP-kod = H)

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler (=1)

¤¤¤

Övningsuppgift 1c:
Översätt instruktionssekvensen nedan till maskinkod med hjälp av instruktionslistan för FLEX-processorn. Instruktionssekvensens startadress i minnet skall vara 20H.

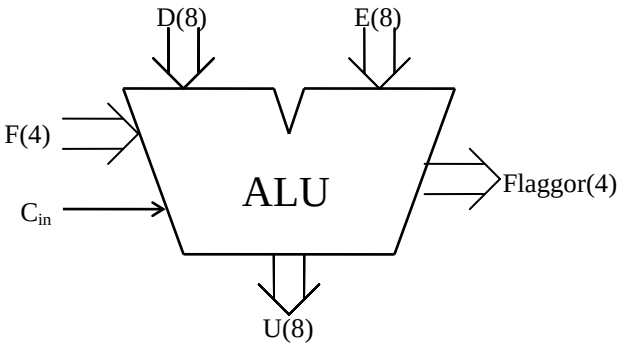
Adress (Hex)	
1F	-
20	
21	
22	
23	
24	
25	
26	
27	

```
        LDAA    #$90
LOOP    INCA
        STAA    $05
        JMP     LOOP
```

¤¤¤

Bilaga 1

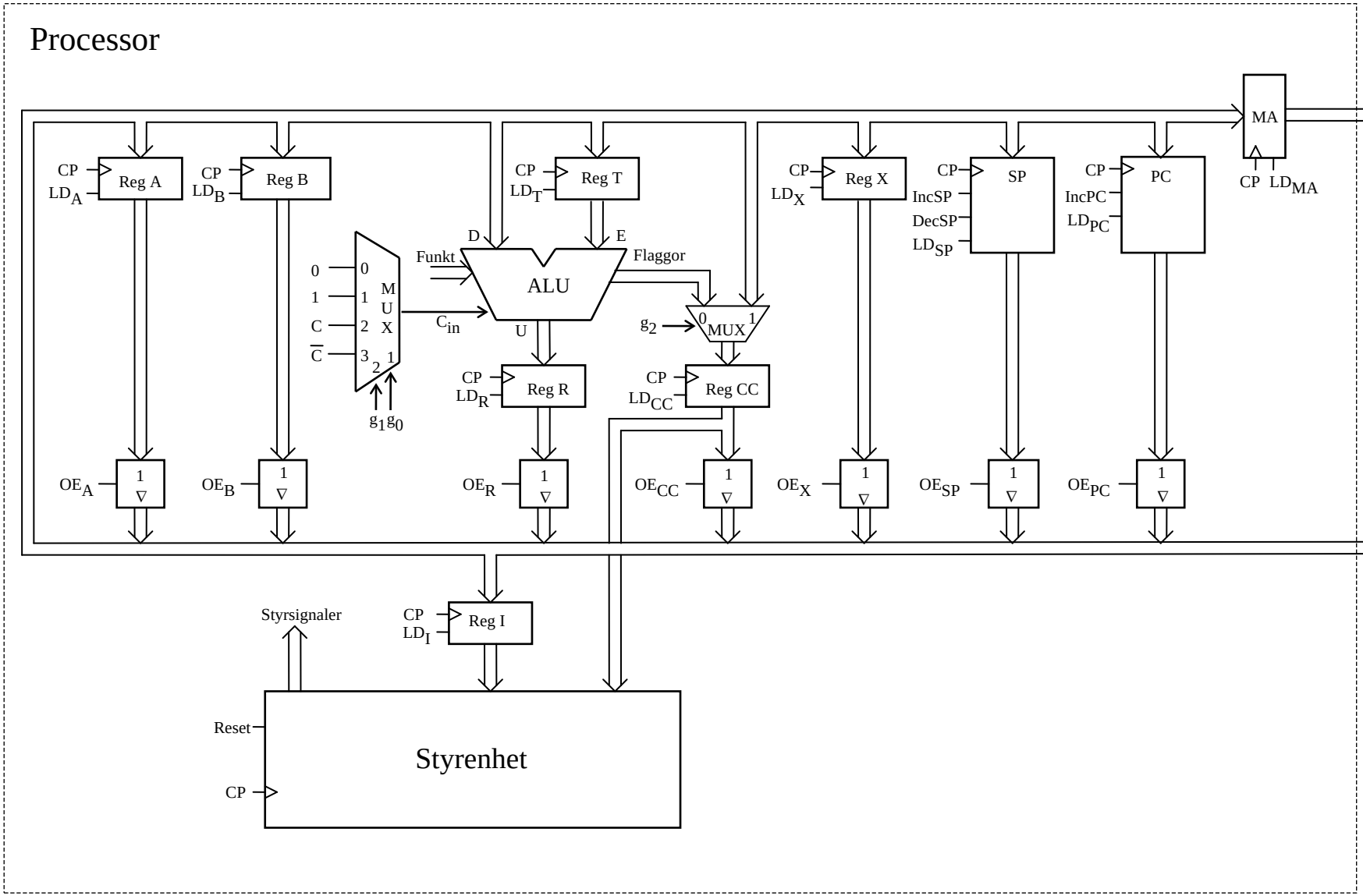
ALU:ns funktion



ALU:ns logiska och aritmetiska operationer på indata D och E definieras av ingångarna F = (f_3, f_2, f_1, f_0) och C_{in} enligt tabellen nedan.

f_3 f_2 f_1 f_0	U = f(D,E,C_{in})
0 0 0 0	0
0 0 0 1	D
0 0 1 0	E
0 0 1 1	D'
0 1 0 0	E'
0 1 0 1	D OR E
0 1 1 0	D AND E
0 1 1 1	D XOR E
1 0 0 0	D + C_{in}
1 0 0 1	D – 1 + C_{in}
1 0 1 0	D + E + C_{in}
1 0 1 1	D + D + C_{in}
1 1 0 0	D – E – 1 + C_{in}
1 1 0 1	0
1 1 1 0	0
1 1 1 1	FFH

I tabellen ovan avser "+" och "-" aritmetiska operationer. Med t ex D' menas att samtliga bitar i D inverteras.



Figur 1. Datorn FLEX.

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

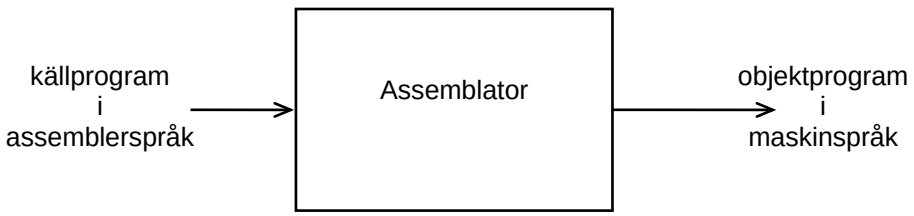
- det inte finns någon kompilator, som genererar kod, för den aktuella processorn.
- det finns uppgifter som inte går att beskriva i ett högnivåspråk.
- någon uppgift är så tidskritisk att en kompilator inte klarar av att göra koden så att den klarar tidskraven.
- man vill lära sig "hands-on" hur den 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
```

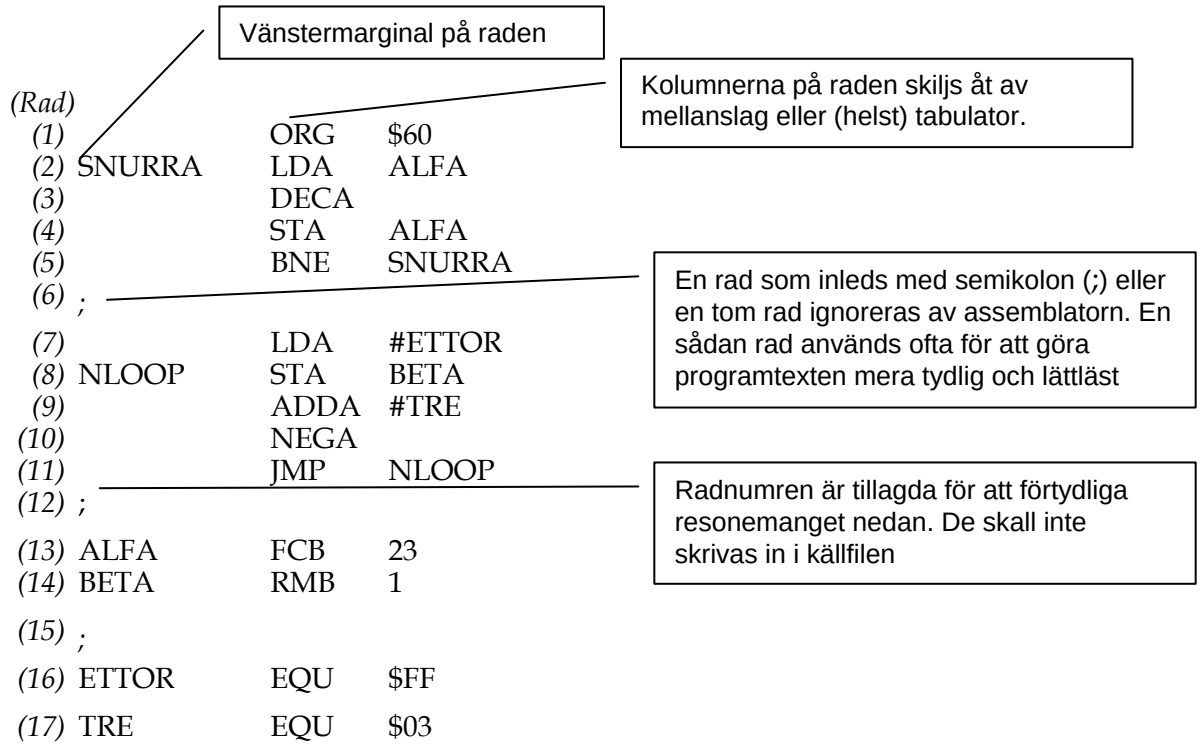
där vi har ersatt adressen 40_{16} med den symboliska beteckningen LOC1 och adresserna 20_{16}, 21_{16} och 30_{16} med variabelnamnen ALFA1, ALFA2 och SUM, samt konstanten $0F_{16}$ med namnet CONST.

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.

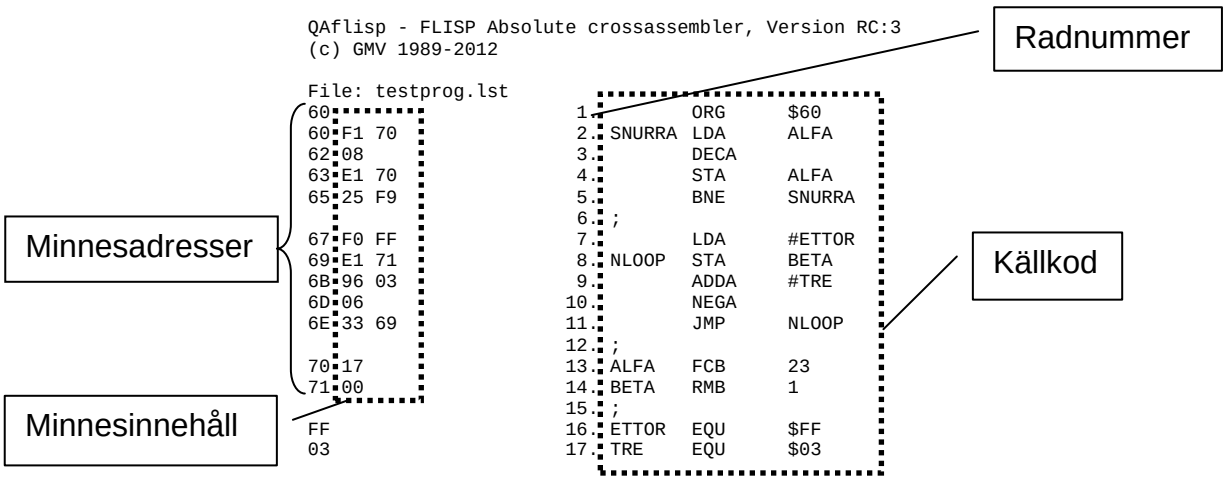
Under pass 2 går assemblatorn igenom programavsnittet igen, skapar programkod och placerar in den i en laddfil och en listfil enligt principen nedan.

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
```

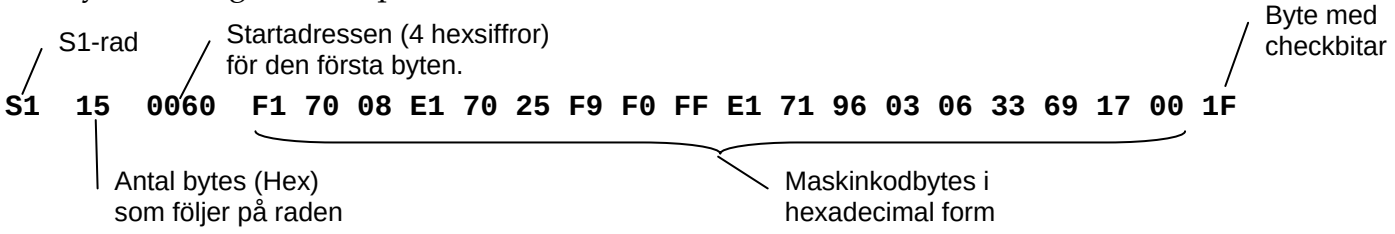
Man känner igen källkoden till höger i fönstret. Närmast till vänster om källkoden har varje rad försetts med radnummer 1-17. Längst till vänster finns minnesadressen och maskinkoden för varje instruktion. Man noterar också att raderna med EQU-definitioner inleds med den definierade symbolens värde längst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

Utdrag ur:

Grundläggande digital- och datorteknik

Del 2 – Datorteknik

Kapitel 13:

- Anslutning av minnes- och I/O-moduler till CPU12-buss

13 Anslutning av minnes- och I/O-moduler till CPU12-buss

Varje processor har minst ett ***adressrum***. Detta definieras som den mängd möjliga lagringsställen som kan adresseras via adressbussen och det kallas i fortsättningen för adressrum M (minne, memory). I kapitel 11 sågs att en stor del av maskininstruktionerna arbetar mot detta adressrum, som huvudsakligen rymmer primärminnet. Via programräknaren eller instruktioner kan processorn hämta eller placera binära ord på någon av adressrummets adresser under förutsättning att det finns ett läs- eller skrivbart register på adressen. Att hämta eller placera ett ord på en adress är liktydigt med att ***läsa*** (eng. ***read***) respektive ***skriva*** (eng. ***write***) på adressen.

Hur adressrummet används (är fyllt) beror mycket på den tillämpning i vilken processorn skall användas. Den största delen används för primärminnet. Vad som menas med primärminne beskrevs i avsnitt 9.5. Man kan se primärminnet som en stor mängd numrerade register.

Två typer av minnesmoduler kan användas. I den ena är registren både läs- och skrivbara. Sådana minnen kallas för ***läs/skrivminnen***, **RWM** (eng. ***read/write memories***I den andra är registren enbart läsbara och de kallas för ***läsminnen***, **ROM** (eng. ***read only memories***).

En mindre del av adressrummet brukar användas för ***separata register***, som inte tillhör primärminnet utan används för andra ändamål. Några av dessa är register som processorn använder för att kommunicera med omvärlden. Denna kommunikation kallas för ***in- och utmatning av data*** (eng. ***input and output*** eller **I/O**). Register som används för in- och utmatning av data kallas ofta ***I/O-portar***. För processorn är en I/O-port enbart läsbar (en inport) eller enbart skrivbar (en utport) i normalfallet.

På varje adress i adressrummet kan det således finnas antingen ett register som är både läs- och skrivbart (som i RWM) eller två register av vilka det ena enbart är läsbart från processorn (en inport) och det andra enbart är skrivbart från processorn (en utport).

Detta kapitel behandlar hur adressrummet kan utnyttjas dels för primärminne, dels för in- och utmatning av data. För att göra detta behöver man god kännedom om hur processorn arbetar mot sitt adressrum M. Kapitlet inleds med en beskrivning av dels begreppen "random access" och "random access"-minnen, dels hur CPU12 arbetar mot sitt adressrum. Detta för att CPU12 används flitigt i kursen för exemplifiering.

Mål: Efter att ha studerat kapitlet skall man kunna:

- konstruera ett CPU12 baserat system med önskvärd kombination av RWM- och ROM-moduler, samt I/O-portar
- konstruera adressavkodare för ett givet innehåll i adressrummet M
- konstruera system med endera fullständig eller ofullständig adressavkodning

13.1 Något om "random access"-minnen

Det skall vara lika lätt att komma åt innehållet på en adress i adressrummet oavsett var adressen är. Med detta menas att åtkomsttiden (= accesstiden) skall vara densamma för alla adresser. Principen kallas för "random access". Minnen som svarar upp mot detta kallas för ***"random access"-minnen,*** **RAM** (eng ***random access memories***) (access = åtkomst).

I dagens datorer utgöres minnesmodulerna av ***halvledarminnen*** (eng. ***semiconductor memories***). Detta är integrerade kretsar som innehåller en stor mängd minnesregister av något slag. Som tidigare sagts skiljer man på läs/skrivminnen, RWM och läsminnen, ROM. Båda är "random access"-minnen.

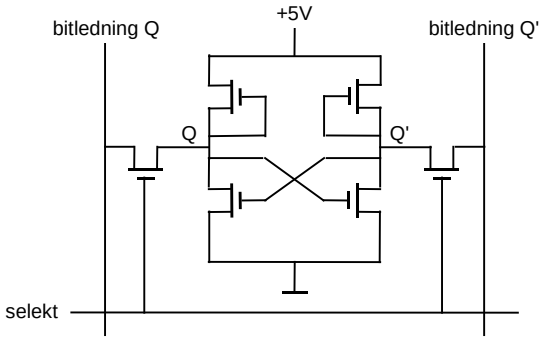
RWM ger användaren möjlighet att byta ut program och data eller att göra ändringar i dataareor. Halvledar-RWM förlorar sitt informationsinnehåll när matningsspänningen slås av och sägs därför vara ***flyktigt*** (eng. ***volatile***).

Exempel 13.1

Här exemplifieras två typiska bitceller för läs/skrivminnen, RWM. De är realiserade i en halvledarteknologi som kallas nMOS (n-channel Metal-Oxide-Semiconductor).

6-transistorcell

Kopplingen i Figur 13.1 innehåller sex nMOS-transistorer. Den mittre delen är en modern variant av den triggerkoppling (latch) som Eccles och Jordan presenterade 1919 (jfr figur 5.9 i kapitel 5. Det är en latch som både går att ettställa och nollställa. De två omgivande transistorerna är switchar som möjliggör att man kan komma åt latchen.



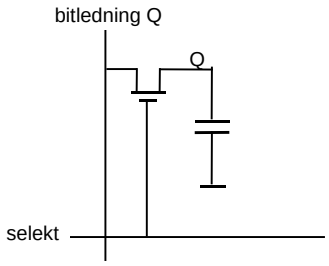
Figur 13.1

Kopplingen används som bitcell i statiska läs/skrivminnen (de kallas på engelska static RAMs).

Cellen är ***statisk*** för att den behåller sitt värde så länge den har matningsspänning (+5V). Innehållet i cellen kan läsas eller ändras genom att aktivera selektledningen och därefter på lämpligt sätt använda bitledningarna.

1-transistorcell

Kopplingen i Figur 13.2 har inga likheter med latchen. Istället lagras bitvärdet som laddning i en kondensator. Laddning i kondensatorn svarar mot "1" och ingen laddning svarar mot "0".



Figur 13.2

nMOS-transistorn fungerar som switch och möjliggör att man kan komma åt kondensatorn. Via selekt kan då kondensatorn kopplas till eller från bitledningen. När kondensatorn är bortkopplad från bitledningen så "läcker" den långsamt, dvs laddningen försvinner långsamt. Man säger att en kondensator är en ***dynamisk*** bitcell just för att den läcker. Funktionen som bitcell bygger därför på att kondensatorn används ofta, dvs laddas upp (med "1") eller ur (med "0") ofta. Innehållet i cellen kan läsas eller ändras genom att aktivera selektledningen och bitledningen.

Vid läsning laddas kondensatorn ur. Efter läsningen har den således ingen laddning. Det lästa värdet måste således återskrivas i cellen, dvs kondensatorn måste återladdas, annars har ju cellen "tappat minnet".

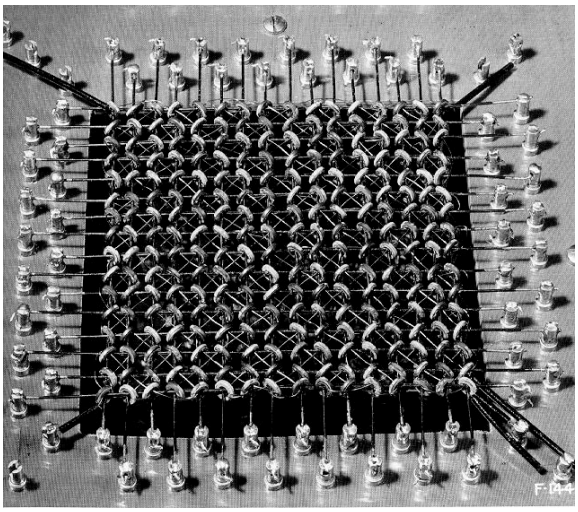
Kopplingen används som bitcell i dynamiska läs/skrivminnen (dynamic RAMs). Den är trots sina egenheter mycket attraktiv för att den är enkel och tar liten plats på ett chip. Den tar ju bara upp en sjättedel så stor yta som den statiska bitcellen.

Kärnminnet - gårdagens RWM

I kapitel 9 gavs en del viktiga milstolpar i datorernas utveckling. En av dem löd:

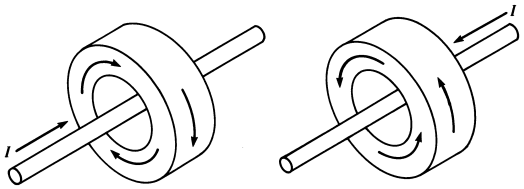
1953 installeras ringformade magnetiska kärnor att användas för lagring av bitvärden, s k kärnminnen, för första gången i en dator

I mer än 20 år fram till mitten av 1970-talet när halvledarminnen fick sitt stora genombrott utgjordes primärminnet av ***kärnminnesmatriser***. En sådan visas i Figur 13.3. I dessa utgjordes bitcellen av en liten ringformad kärna av magnetiserbart material (ferrit). Kärnans diameter var typiskt 0,5 mm.



Figur 13.3

Kärnan kan magnetiseras endera medsols eller motsols med hjälp av ström I, såsom visas i Figur 13.4.



ettställning nollställning

Figur 13.4

Magnetiseringen blir kvar när strömmen slås av och kärnan minns därför vilken riktning strömmen hade senast. Man kan således med strömriktningen lagra värdet "0" eller "1" i kärnan.

För att läsa vad som finns i kärnan måste magnetiseringsriktningen detekteras och detta gjorde man genom att slå på strömmen I i en av riktningarna, säg nollriktningen, och mäta om magnetiseringen ändrades eller inte. Ändrades den inte så innehåller cellen "0". Ändrades magnetiseringen så innehöll cellen "1", men läsningen har nu ändrat innehållet till "0". När cellen innehöll "1" så måste värdet "1" återskrivas efter läsningen. Vid läsning har således kärnan samma egenskaper som kondensatorn i den dynamiska cellen.

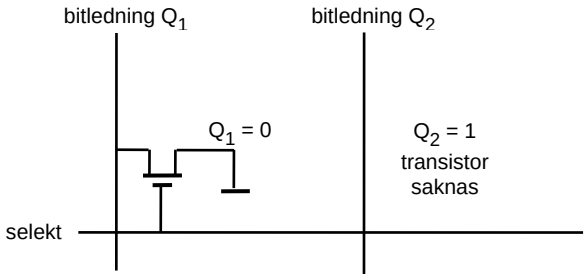
ROM är förprogrammerade och används för program och/eller data, som alltid skall vara tillgängliga i systemet, och för vilka inget behov av utbytbarhet föreligger. Läsminnen behåller sitt informationsinnehåll även när matningsspänningen är frånslagen. Man säger att ett läsminne är ***icke-flyktigt*** (eng. ***non volatile***).

Exempel 13.2

Läsminnen (ROM) kännetecknas av att de behåller sitt innehåll även om matningsspänningen slås av. De finns olika varianter och de klassificeras efter det sätt som minnesmodulen ges sitt innehåll. Följande varianter är vanliga.

Mask-programmerade ROM får sitt innehåll när modulen tillverkas. Innehållet kan inte ändras. Det är dyrt att skapa den mask som definierar innehållet, så denna typ av minne lämpar sig bäst vid tillverkning av stora serier med ett och samma innehåll.

I Figur 13.5 visas de två typiska fallen på en bitcells utseende. Den är som synes en 1-transistorcell och tar därför upp liten yta på ett chips. Detta kännetecknar även bitcellerna för varianterna som följer.



Figur 13.5

Fält-programmerbara ROM ges sitt innehåll ute på "fältet" (där de skall användas) innan de används. När de väl fått ett innehåll så går det inte att ändra. Modulerna kallas PROM (programmable ROM). Programmeringen sker i en speciell utrustning kallad PROM-brännare.

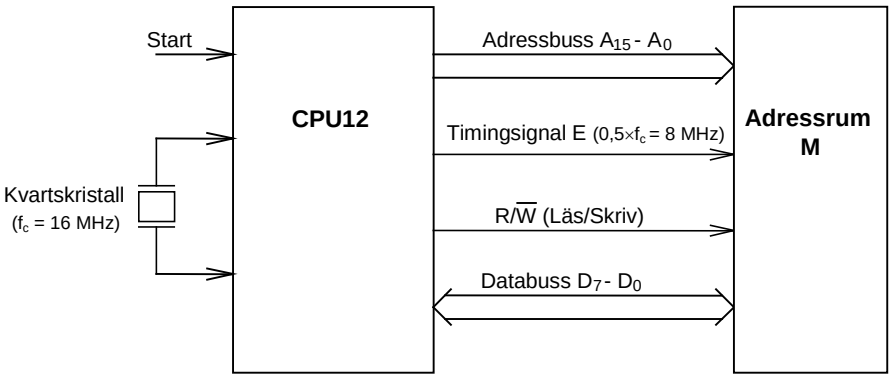
Raderbara ROM ges sitt innehåll ute på fältet innan de används. De kan raderas med ultraviolet ljus och omprogrammeras på elektrisk väg. För att göra detta måste modulen plockas ut ur det system i vilket det används. Radering och omprogrammering kan ske många gånger. Dessa moduler kallas EPROM (Erasable Programmable ROM).

Elektriskt raderbara ROM ges sitt innehåll ute på fältet innan de används. De kan raderas och omprogrammeras på elektrisk väg utan att ta ut modulen ur systemet i vilket de används. Dock behövs ett elektriskt specialarrangemang för att göra ändringarna. Ändringarna görs ord för ord, ett i sänder. Dessa moduler kallas också EEPROM (Electrically Erasable PROM)

"Flash"-minnen är en form av elektriskt raderbara ROM. I dessa raderas block av ord istället för ett ord isänder.

13.2 Processorn CPU12

I Figur 13.6 visas de signaler CPU12 har för arbete mot adressrum M.

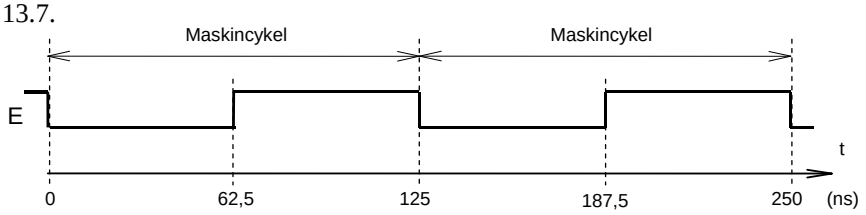


Figur 13.6 CPU12 kopplad till adressrum M.

Läsning respektive skrivning kallas för minnesreferenser och sker vanligen under en klockcykel. Namnet minnesreferens kommer av att adressrum M används för såväl primärminne som andra minnesregister.

CPU12-klockning (timing)

CPU12 genererar "timing"-signalen E under hela tiden den arbetar. Den bildas ur signalen från en kristalloscillator med frekvensen 16 MHz i normalfallet. Frekvensen hos E är halva kristallfrekvensen, dvs 8 MHz i normalfallet, se Figur 13.7.



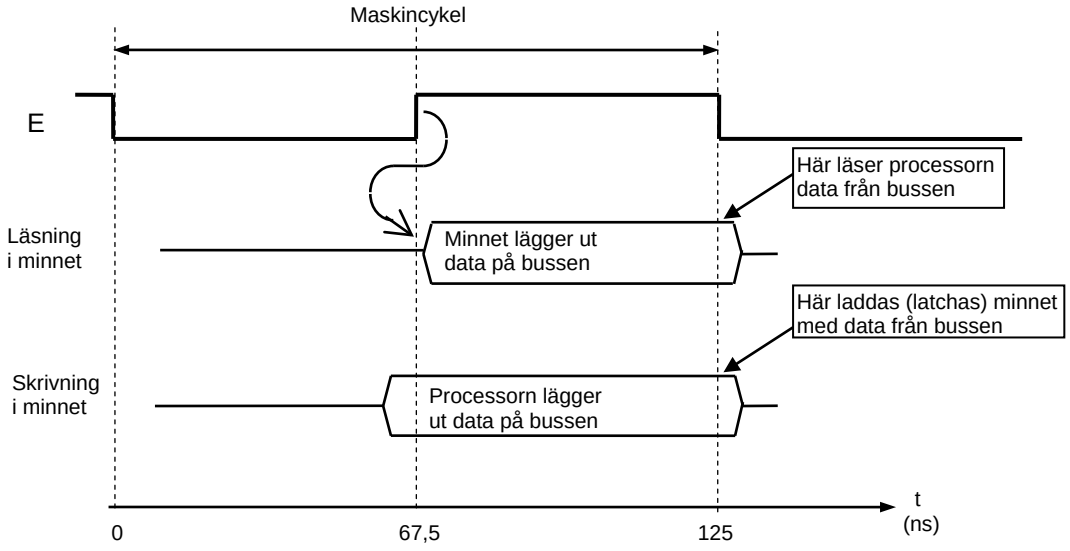
Figur 13.7 Timingsignalen E hos CPU12.

När CPU12 läser eller skriver i minnet utför den en ***maskincykel***. Den inleds när ***signalen* E** går från hög till låg nivå (negativ flank) och avslutas med nästa negativa flank på E-signalen, såsom visas i Figur 13.7.

En läsning i minnet (en läscykel) inleds med att processorn lägger ut adressen, där läsningen skall ske, på adressbussen samtidigt som signalen ***Läs*** / ***Skriv*** (***R*** / ***W***) ges värdet "1". Det sker vid negativ flank på E-signalen.

På grund av att det finns interna fördröjningar i processorn så kommer varken adressbitar eller R / W -signalen att få rätt värde förrän ***efter*** E-signalens negativa flank. Dessutom kommer de inte att få rätt värde samtidigt på grund av att fördröjningarna är olika för de olika signalerna. Vid tidpunkten för E-signalens positiva flank är dock adressen och R / W -signalen garanterat korrekta (stabila) och först då bör minnet börja "plocka" fram data från den aktuella adressen och placera det på databussen, vilket visas i Figur 13.8. Processorn läser sedan av data från databussen vid slutet av maskincykeln dvs vid E-signalens nästa negativa flank.

Ett liknande resonemang kan föras för adresser och R / W -signalen vid skrivning. Skrivningen inleds med att processorn lägger ut adressen, där skrivningen skall ske, på adressbussen samtidigt som signalen ***Läs*** / ***Skriv*** (***R*** / ***W***) ges värdet "0". Därefter lägger processorn ut data på databussen på det sätt som visas i Figur 13.8. Minnet laddas sedan med data från databussen vid slutet av maskincykeln dvs vid E-signalens nästa negativa flank.



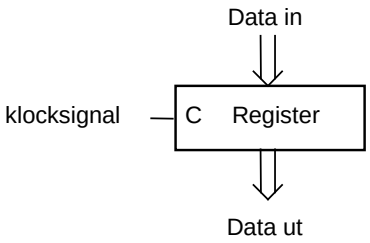
Figur 13.8 Läsning och skrivning med CPU12-processorn

Vid läsning och skrivning är således adressen och R / W -signalen, som läggs ut från processorn vid E-klockans negativa flank, inte garanterat korrekta (stabila) förrän vid E-klockans positiva flank. Under resterande halvan av maskincykeln är dock adressen stabil. Adressignalerna och R / W signalen har således stabila värden under tiden som E är "1". E- signalen är därför en **VMA-signal** (***Valid Memory Address***). För CPU12 används signalen VMA = E ofta som grindsignal vid arbete mot adressrum M.

När minnet är anslutet till processorns bussar enligt principen i figurerna 13.6 och 13.8 är det processorn som bestämmer timing vid läsning och skrivning i minnet. Det förutsätts då att minnet hinner med att hantera data vid läsning och skrivning. Denna princip kallas **synkron bussanslutning**. Vid **asynkron bussanslutning** måste processorn vänta på klarsignal från minnet vid läsning och skrivning.

13.3 Minnesmoduler

Primärminnet byggs upp av moduler. I detta avsnitt beskrivs den principiella uppbyggnaden av en minnesmodul som är både läsbar och skrivbar, dvs en RWM-modul.

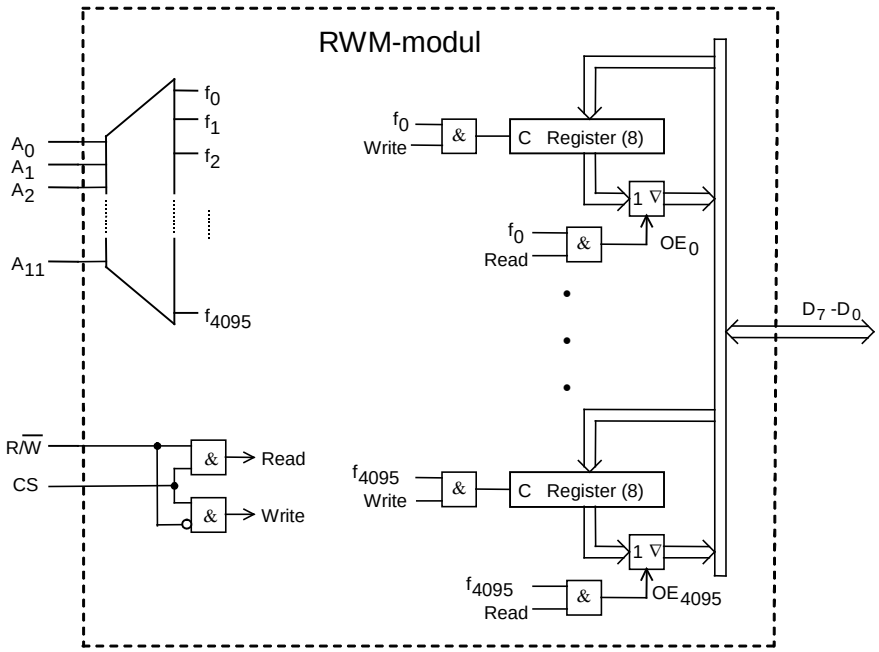


Figur 13.9 Blocksymbol för register uppbyggt med klockade D-latchar.

RWM-moduler innehåller register av latchtyp. Registren är därför uppbyggda med enkla klockade D-latchar, sådana som tidigare visats i figur 5.13. För ett sådant register används här blocksymbolen i Figur 13.9. Kom ihåg att ett sådant register inte är flanktriggat utan att innehållet kan ändras via "Data in"-ingångarna under hela den tid som klocksignalen är "1".

I Figur 13.10 visas schematiskt hur man kan tänka sig att en minnesmodul är uppbyggd med denna registertyp.

Ett register i modulen adresseras via adressledningarna. Eftersom var och en av dem kan anta värdena "0" och "1", så är antalet ord i modulen en heltalspotens av 2. Modulen i figuren har 12 st adressledningar och innehåller således 2^{12} = 4096 st ord.



Figur 13.10 Principiellt utseende av en 4k RWM-modul.

Orden har vanligen åtta bitar, dvs ordlängden är en byte. Vanliga modulstorlekar ligger i området från 1kbyte till 256 kbyte (1kbyte = 1024 bytes = 2^{10} bytes). Modulen i figur har således 4 kbytes (= 4·1024 bytes).

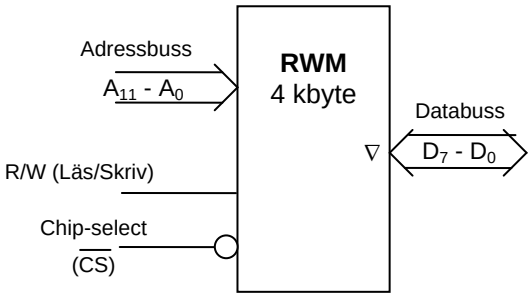
Det är nödvändigt att minnesmoduler har "three-state"-buffertar på registerutgångarna, som i Figur 13.10, ty varje modul skall kunna kopplas till en gemensam databuss för att lägga ut data på den. Detta sker internt via respektive registers OE-signal.

I en modul som också skall ta emot data från databussen måste registren kunna laddas via en signal. I Figur 13.10 sker detta internt via en signal till respektive registers grindingång C.

För CPU12 kan dessa interna signaler (till C och till "three-state" buffertar) skapas m h a signalen $R/\overline{W}$, som är "1" vid läsning och "0" vid skrivning.

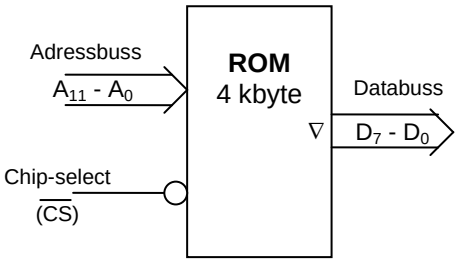
En modul som skall aktiveras (för läsning eller skrivning) skall kunna pekas ut (väljas) och det görs på en ingång som normalt kallas för ***"Chip Select"*** **(CS)** eller "Chip Enable" (CE) och den är oftast aktiv låg ($\overline{CS}$ eller $\overline{CE}$).

Minnesmodulens adressledningar (12 st), A_{11}-A_0, är inuti modulen kopplade till en stor avkodare (12/4096), såsom visas i Figur 13.10, för att välja ut rätt minnesregister för läsning eller skrivning. I fortsättningen kommer blocksymbolen i Figur 13.11 att användas för en 4 kbyte RWM-modul.



Figur 13.11 Blocksymbol för läs-/skrivminnesmodul (RWM) om 4 kbyte.

Man kan på ett liknande sätt beskriva den principiella uppbyggnaden av en ROM-modul, dvs en modul som enbart är läsbar. Det görs dock inte här. Dock ges blocksymbolen för en 4 kbyte ROM-modul i Figur 13.12. Den har som synes ingen insignal för att avgöra om läsning eller skrivning skall ske. Detta för att den enbart är läsbar.



Figur 13.12 Blocksymbol för läsminnesmodul (ROM) om 4 kbyte.

13.4 Anslutning av minnesmoduler till CPU12

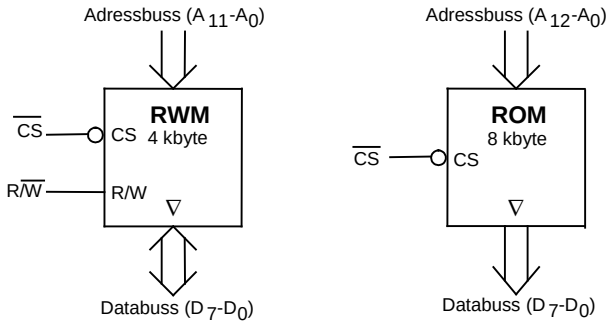
När minnesmoduler, lika de i Figur 13.11 och Figur 13.12, placeras i processorns adressrum M ansluts adressingångarna till motsvarande ledningar i adressbussen. De övriga adressledningarna i bussen, A_{15} -A_{12}, används för att välja rätt minnesmodul genom att man låter "Chip Select"-signalen bildas med hjälp av dem. Därmed aktiverar "Chip Select"-signalen minnesmodulen i det 4k-ords adressintervall, inom vilket processorn skall adressera minnesmodulen.

Eftersom flera moduler är kopplade till samma databuss måste man se till att inte två eller fler moduler lägger ut data på bussen samtidigt. Det är sedan tidigare känt att adressvärdet som processorn släpper ut inte är korrekt (stabilt) omedelbart efter E-signalens negativa flank. För att undvika "busskollisioner" på grund av "falska" adressvärden i början av maskincyklerna måste man förutom de högsta adressbitarna även använda VMA = E som grindsignal till det logiknät som bildar "Chip Select"-signalerna.

En konstruktör, som skall göra adressavkodningen för en dator, utgår ifrån hur stort minne och vilken typ av minne som behövs. Därefter avgörs hur stora minnesmoduler som skall användas. Valet bestäms ofta av sådana faktorer som snabbhet, pris, leveranstid m m. Nästa steg är att placera in de olika modulerna i processorns adressrum. Hur det görs beskrivs i ett exempel.

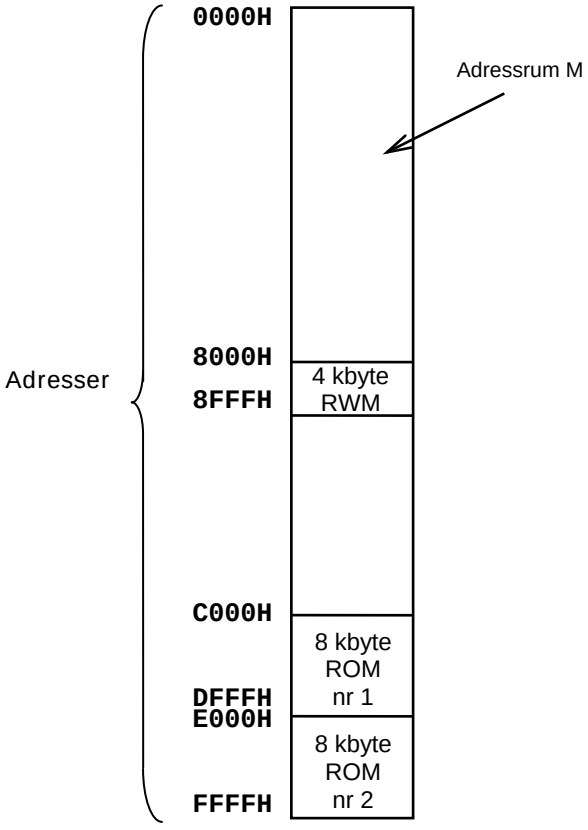
Exempel 13.3

För en viss CPU12-tillämpning behövs ett primärminne om 4 kbyte RWM och 16 kbyte ROM. Man har bara tillgång till RWM- och ROM-moduler av den typ som visas i Figur 13.13.



Figur 13.13 Tillgängliga RWM- och ROM-moduler.

Inplaceringen i adressrummet skall vara enligt Figur 13.14. Lägg märke till att modulerna placerats in på ett systematiskt sätt så att deras adresser börjar på "bra" adresser ex vis 8000H, C000H, E000H etc.



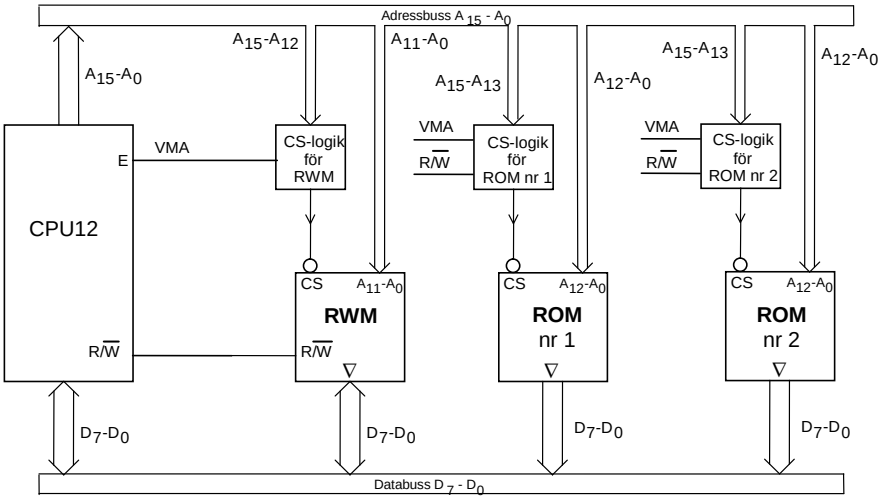
Figur 13.14

Inplaceringen kan också sammanfattas i tabellform:

Modul	*Adress*	*Adressignal nr*															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RWM	8000H	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	8FFFH	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
ROM nr 1	C000H	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	DFFFH	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
ROM nr 2	E000H	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	FFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

De olika modulerna skall adresseras inom unika adressintervall (områden). Inom respektive minnesmoduls adressområde är ett litet antal av de mest signifikanta adressbitarna konstanta. De är markerade med en ram. De har samma (unika) värden för alla adresser inom området och kan sägas bilda en ID-kod för respektive område. ID-koden kan därför användas för att "peka ut" respektive minnesmodul, dvs för att bilda CS-signalen, som är aktiv när processorn skall läsa eller skriva i just denna modul.

Modulerna ansluts till adress- och databussen enligt Figur 13.15.

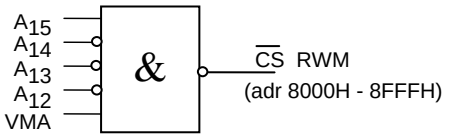


Figur 13.15 Anslutning av minnesmoduler till processorn CPU12.

Adressavkodningen (CS-logiken) sker i logikblock med de mest signifikanta adressbitarna och VMA, samt när så behövs signalen R / $\overline{W}$, som insignaler. Utsignalerna består av "chip select"-signaler till de olika modulerna.

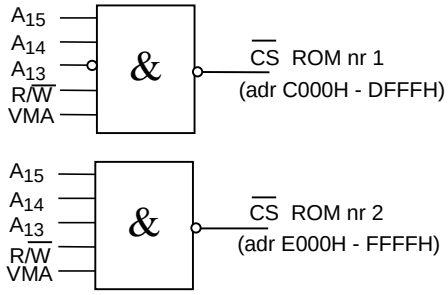
Börja med att betrakta adressavkodning för RWM-modulen. När modulstorleken är 4k = 2^{12} används 12 st adressbitar för intern adressavkodning inom modulen. Detta framgår av Figur 13.15. De övriga fyra adressbitarna, A_{15} - A_{12}, används för att bestämma var någonstans i processorns adressrum modulen placeras.

Enligt disponeringen av adressrummet i Figur 13.14 skall RWM-modulen placeras i adressintervallet 8000H - 8FFFH. I detta har adressbitarna A_{15} - A_{12} bitvärdena 1000 (se tabellen). Modulen kan då pekas ut med CS-signalen, som bildas enligt Figur 13.16. 1000 kan således ses som ID-koden för modulen



Figur 13.16 CS-logik för 4kbyte RWM-modul.

De två ROM-modulerna om vardera 8 kbyte skall ligga i adressområdena C000H-DFFFH (ROM nr 1) respektive E000H-FFFFH (ROM nr 2). I Figur 13.17 visas hur CS-signalerna, för att peka ut respektive ROM, bildas.



Figur 13.17 CS-logik för två 8 kbyte ROM.

Här används även R / $\overline{W}$ -signalen vid bildandet av CS eftersom man då kan undvika problem vid ett eventuellt försök till skrivning (av processorn) på en ROM-adress. Om R / $\overline{W}$ -signalen inte används på detta sätt så skulle en eventuell skrivning i en ROM-modul innebära att både processorn och ROM-modulen släpper ut data på databussen och då fås en "busskollision". Försök till skrivning i ROM kan ex vis inträffa på grund av ett programmeringsfel.

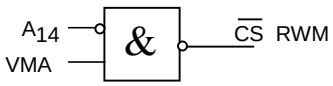
I Exempel 13.3 används adressbussens alla bitar i minnesavkodningen. Detta kallas för ***fullständig adressavkodning***. För att förenkla adressavkodningsblocken kan man ibland låta bli att använda alla adressbitarna vid avkodningen. Detta får till följd att adressavkodningen inte blir entydig dvs samma CS-signal kan bli aktiv i fler än ett adressintervall. Man kallar detta förfarande för ***ofullständig adressavkodning***

Har man valt att göra adressavkodningen ofullständig så blir det också svårt att vid en senare tidpunkt bygga ut minneskapaciteten i datorn. En annan nackdel med ofullständig adressavkodning är att samma modul kan adresseras på olika ställen i adressrummet. Detta kan ställa till problem om fel gjorts vid programskrivningen.

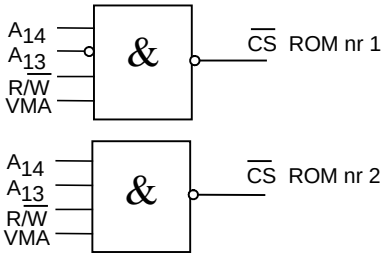
Exempel 13.4

Om avkodningen i Exempel 13.3 istället gjorts med CS-avkodare enligt Figur 13.18 a och b så är den ofullständig.

a) CS-logik för 4kbyte RWM-modulen



b) CS-logik för de två 8 kbyte ROM-modulerna



Figur 13.18

Enligt Figur 13.18 så finns det RWM överallt där adressbiten A_{14} är 0 och ROM överallt där A_{14} är 1. Detta har markerats i följande tabell, som avbildar processorns hela adressrum M.

Modul	***Adress***	***Adressignal nr***															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RWM	0000H	**0**	0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	0FFFH	**0**	0	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
RWM	1000H	**0**	0	**0**	**1**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	1FFFH	**0**	0	**0**	**1**	1	1	1	1	1	1	1	1	1	1	1	1
RWM	2000H	**0**	0	**1**	**0**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	2FFFH	**0**	0	**1**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
RWM	3000H	**0**	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	3FFFH	**0**	0	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1
ROM nr 1	4000H	**0**	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	5FFFH	**0**	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
ROM nr 2	6000H	**0**	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	7FFFH	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RWM	8000H	**1**	0	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	8FFFH	**1**	0	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
RWM	9000H	**1**	0	**0**	**1**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	9FFFH	**1**	0	**0**	**1**	1	1	1	1	1	1	1	1	1	1	1	1
RWM	A000H	**1**	0	**1**	**0**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	AFFFH	**1**	0	**1**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
RWM	B000H	**1**	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	BFFFH	**1**	0	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1
ROM nr 1	C000H	**1**	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	DFFFH	**1**	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
ROM nr 2	E000H	**1**	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	:									:							
	FFFFH	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Det ofullständiga framgår av att modulerna, 4kbyte och 2×8 kbyte, nu "tar upp" (blockerar) mycket mer av utrymmet i adressrummet M, närmare bestämt hela utrymmet M, vilket ses i tabellen. Utöver de områden som är önskvärda så tar modulerna vid denna ofullständiga adressavkodning även upp de områden som gråmarkerats i tabellen.

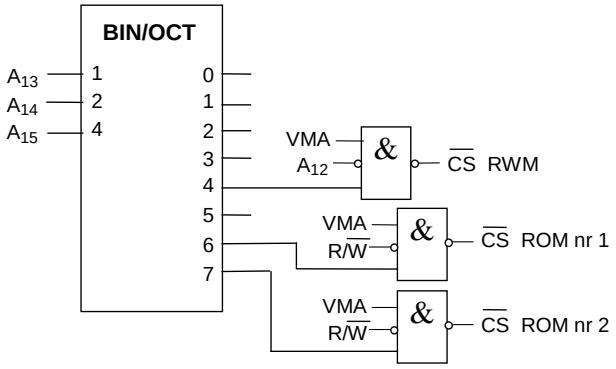
Anledningen till detta är att de adressbitar som markerats med fetstil i tabellen inte ingår i adressavkodningen. De skall därmed betraktas som "don't care"-värden.

I stället för att bygga upp adressavkodningen med diskreta grindar som i de visade exemplen använder man i praktiken ofta färdiga avkodare, fördelare, snabba ROM, PLA- eller PAL-kretsar för att bilda CS-signalerna. PLA- och PAL-kretsar är exempel på kretsar med ***programmerbar logik*, PLD (*Programmable Logic Device*).**

Exempel 13.5

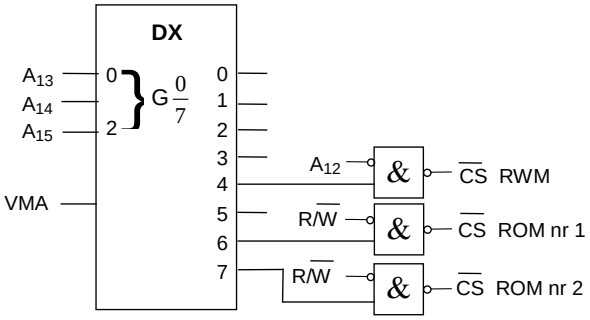
Avkodningen för de tre modulerna i Exempel 13.3 kan anordnas med en färdig funktionsmodul och kompletterande grindelement.

I Figur 13.19 är den ordnad m h a "NBC till en av åtta" avkodare (3/8 avkodare)



Figur 13.19

och i Figur 13.20 m h a en fördelare. Lägg märke till hur VMA-signalen ansluts.



Figur 13.20

I båda fallen ser man att såväl avkodaren som fördelaren också har outnyttjade utgångar som kan användas om systemet behöver utvidgas med ytterligare moduler.

Detta avsnitt skall nu avslutas med att betrakta ett ibland förekommande avkodningsproblem. Det uppstår när en modul behöver överlappa en annan i adressrummet M. Därvid täcks en del av adressrummet av två moduler. I avkodningen måste då en av modulerna prioriteras i det gemensamma adressområdet. CS-logiken för den prioriterade modulen skall då dölja (skymma) det gemensamma området i den oprioriterade modulen. Tekniken för detta kan man kalla för ***prioriterad avkodning***. Tillvägagångssättet studeras via ett exempel.

Exempel 13.6

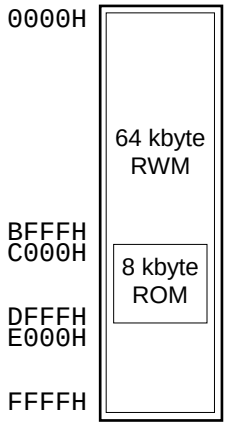
Exemplet är ett fall där ett CPU12 system innehåller dels en 64kbyte RWM-modul, dels en 8kbyte ROM-modul.

Avsikten är att adressrummet skall innehålla 8kbyte ROM och 56kbyte RWM (64k – 8k) med inplacering enligt Figur 13.21.

Anledningen till att man valt en 64kbyte RWM-modul kan vara att det är billigare än att välja flera mindre moduler.

När det gäller CS-bildningen för modulerna ser man av figuren att CS-logiken för ROM-modulen måste dölja motsvarande adressdel av RWM-modulen.

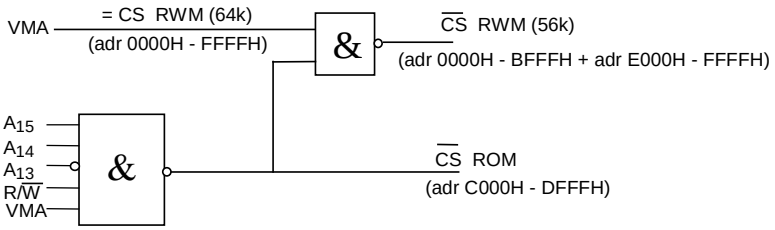
I Figur 13.22 visas hur detta görs. Avkodningen för ROM-modulen blir densamma som tidigare visats för ROM nr 1 i Figur 13.17.



Figur 13.21

Avkodningen för 64kbyte RWM (hela modulen) blir enkel, CS = VMA ty modulen tar upp hela adressområdet.

Tekniken för att dölja adressområdet C000H – DFFFH i RWM är enkel. Eftersom CS-signalen för ROM är 0 endast när detta område adresseras och 1 annars, så används den som grindsignal för att förhindra att CS-signalen för RWM samtidigt är 0. På så sätt används CS-signalen för ROM för att dölja RWM i det gemensamma området.



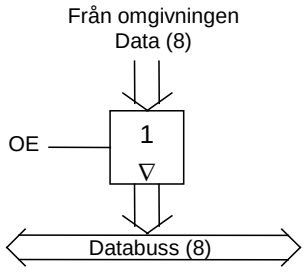
Figur 13.22

13.5 In- och utmatning av data

En dators arbete är utan värde om datorn inte kan ta emot indata från och släppa ut utdata till omgivningen. Varje dator behöver därför anordningar för att sköta kontakten med omvärlden.

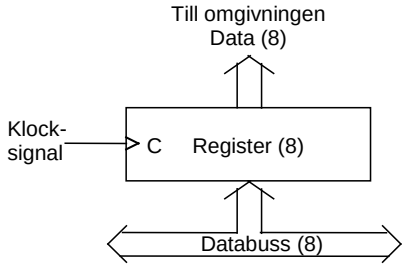
Principiellt påminner in- och utmatning av data mycket om läsning och skrivning i ett primärminne med skillnaden att minnets register bytts ut mot separata register. Man kan helt enkelt bestämma att vissa adresser används för in- och utportar istället för minnesregister.

En ***inport*** behöver dock inte vara ett register utan består ibland bara av en "three-state"-buffert som är kopplad till databussen, såsom visas i Figur 13.23. När processorn läser på "inportadressen" aktiveras buffertens OE-ingång så att data från omgivningen läggs in på databussen och läses av processorn. "Three-state"-bufferten beskrevs i kapitel 7.



Figur 13.23 Principen för en inport.

En ***utport*** är ett register vars D-ingångar är kopplade till databussen och vars vippor laddas parallellt vid den aktiva klockflanken (beskrevs i kapitel 5), se Figur 13.24. Klocksignalen skall aktiveras när processorn skriver på "utportadressen". Databussens innehåll kommer då att placeras i utportregistret. Innehållet i registret förblir sedan oförändrat tills nästa skrivning görs på samma adress.



Figur 13.24 Principen för en utport.

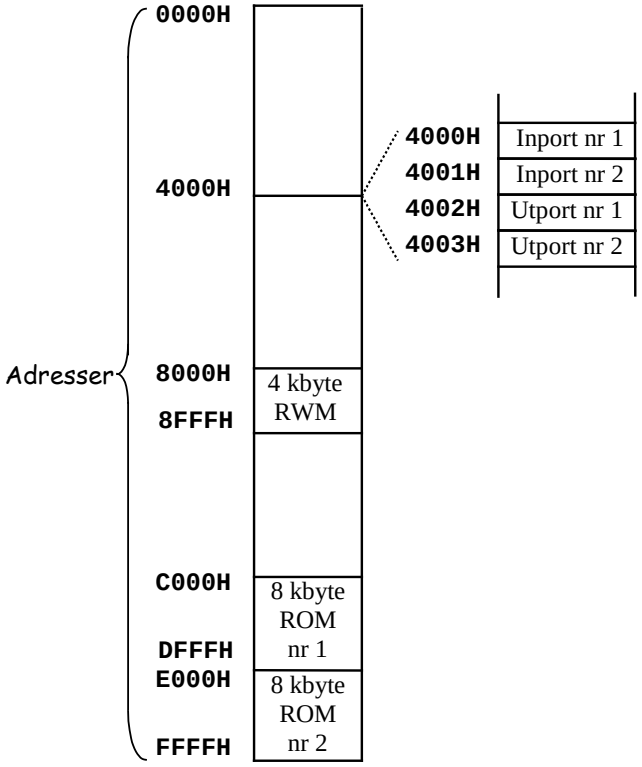
När en processor i likhet med CPU12 behandlar in- och utmatning på samma sätt som läsning och skrivning i minnet kallas detta ***minnesadresserad in- och utmatning*** (eng. ***memory mapped I/O***). Vissa andra processorer har ***speciella I/O-instruktioner*** och därmed också speciella styrsignaler för in- och utmatning. Detta gäller t ex mikroprocessorerna 80X86, som i många år använts i IBM-kompatibla persondatorer. Principen kallas ***separatadresserad in- och utmatning*** (eng. ***isolated I/O***). Fördelen med separatadresserad in- och utmatning är att man inte behöver inkräkta på minnets adressutrymme samt att adressavkodningen för in- och utmatning blir enklare. I sådana fall har man alltså två adressrum.

13.6 Anslutning av in- och utmoduler till CPU12

Beskrivningen görs här genom en utvidgning av systemet i Exempel 13.3 med in- och utmoduler.

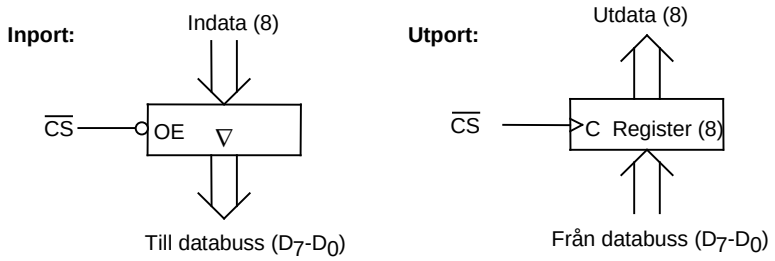
Exempel 13.7

Systemet i Exempel 13.3 skall nu utökas med två in- och två utportar, som skall placeras i adressrummet på adresserna 4000H - 4003H enligt Figur 13.25.



Figur 13.25

Portarna skall vara av den typ som ges i Figur 13.26.

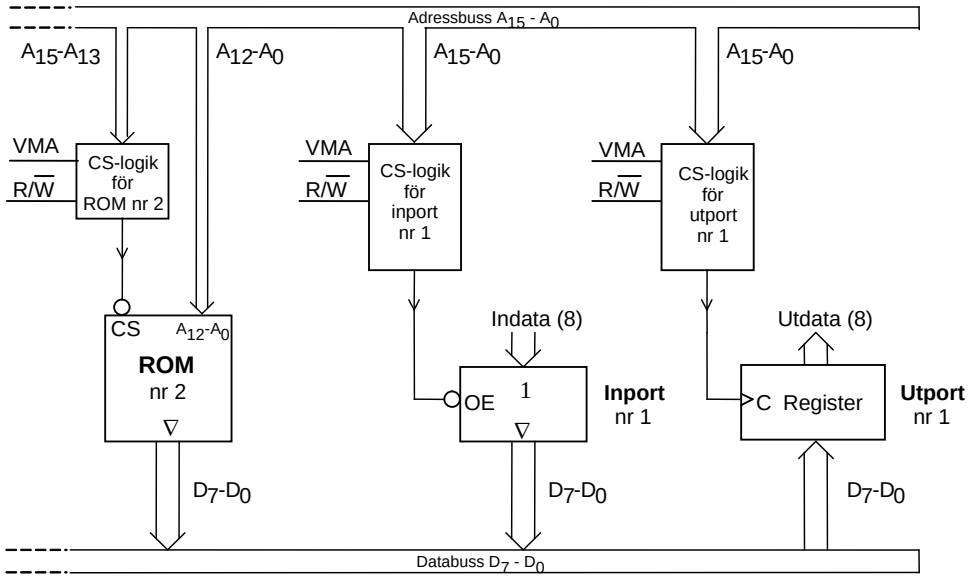


Figur 13.26 Blocksymboler för inports- och utportsmoduler.

Inplaceringen kan sammanfattas i tabellform:

Modul	***Adress***	***Adressignal nr***															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Inport 1	4000H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inport 2	4001H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Utport 1	4002H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Utport 2	4003H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1

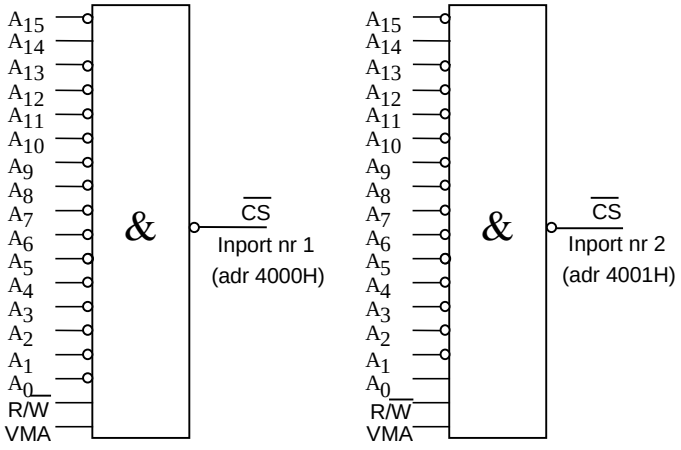
Av Figur 13.26 och tabellen framgår att varje port har en egen (unik) adress. För in- och utportarna krävs alla 16 bitarna i adressen för att "peka ut" rätt enhet, vilket visas i Figur 13.27.



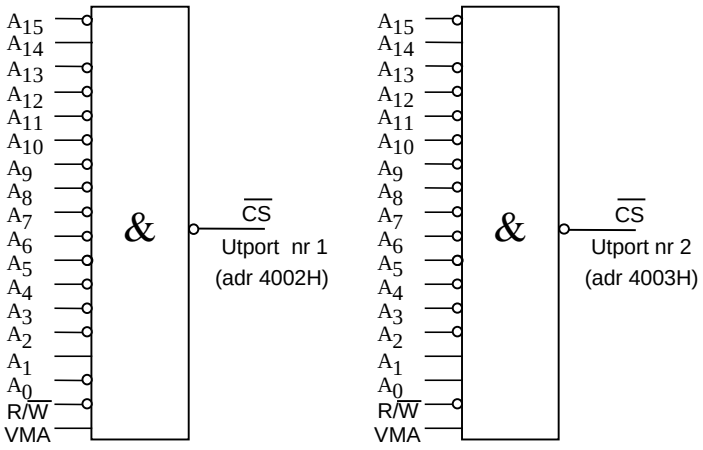
Figur 13.27 Anslutning av minnes- och in-/utmoduler till CPU12.

Som syns i Figur 13.27 sker adressavkodningen (i CS-logiken) för in- och utportarna i logikblock med signalerna A_{15}-A_0, R / $\overline{W}$ och VMA som insignaler. Utsignalerna är "chip-select"-signaler till de olika modulerna.

Figur 13.28 och Figur 13.29 visar schematiskt logiken för adressavkodningen för in- och utportarna enligt den användning av adressrummet som ges i Figur 13.25.



Figur 13.28 "Chip-select"-bildning för inportarna.



Figur 13.29 "Chip-select"-bildning för utportarna.

En jämförelse mellan de fyra "chip-select"-blocken i Figur 13.28 och Figur 13.29 ger att de är mycket lika.

Skillnaden mellan blocken i Figur 13.28 respektive Figur 13.29 är att adressbit 0 är inverterad i det vänstra blocket.

Den principiella skillnaden mellan Figur 13.28 (inportar) och Figur 13.29 (utportar) är att R / $\overline{W}$ -signalen är inverterad i Figur 13.29. Detta medför att blocken i Figur 13.28 är avsedda för läsning i adressrummet (inportar: R / $\overline{W}$ = 1) medan blocken i Figur 13.29 är avsedda för skrivning (utportar: R / $\overline{W}$ = 0).

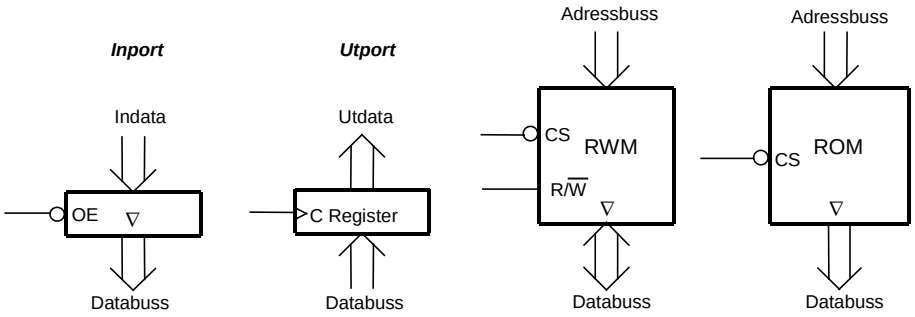
13.7 Resumé

Kapitlet behandlar hur minnes- och I/O-moduler kan placeras i ett adressrum M som definieras av processor-egenskaper som läs/skrivprincip och adressbussens utseende. Hur adressrummet används (är fyllt) beror mycket på den tillämpning i vilken processorn skall användas. Läsaren har därmed fått en inblick i hur en processors adressrum M kan användas dels för primärminne, dels för I/O-portar.

Speciellt har ***läs/skrivminnesmoduler***, **RWM** och ***läsminnesmoduler***, **ROM** beskrivits och använts i exempel på hur modulerna avkodas för att kunna anpassas till processorns egenskaper. Detsamma gäller enkla I/O-portar.

Övningsuppgifter

Om inget annat anges så skall moduler av denna typ användas i uppgifterna.



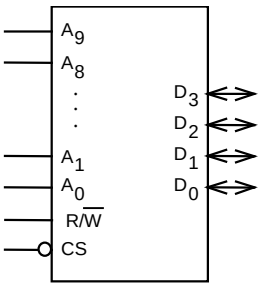
13.1 Figuren visar in- och utgångarna på en typisk minneskapsel.

a) Hur många minnesord finns det i kapseln? Hur breda är minnesorden?

b) Till vilka bussar är de visade in- och utgångarna vanligen anslutna när de används i ett mikrodatorsystem med mikroprocessorn CPU12?

c) Hur många kapslar av denna typ krävs för att bilda fullständiga dataord?

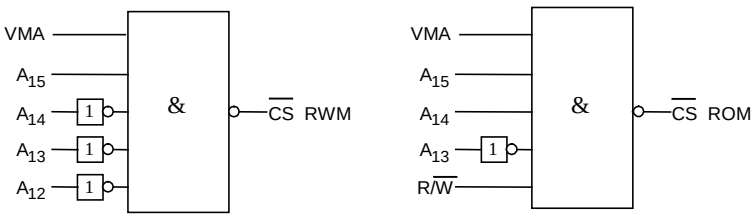
d) Beskriv kortfattat hur CS-ingången används.



13.2 Här ges logiknäten för adressavkodningen för två fullständigt avkodade minneskapslar.

a) Ange i vilket adressområde respektive kapsel är placerad. De två minneskapslarna aktiveras när CS-ingångarna har låg logiknivå (0).

b) Ange minneskapaciteten (antal ord) för respektive minneskapsel.



13.3 En ROM-kapsel om 16 kbyte och en RWM-kapsel om 8kbyte skall anslutas till CPU12.

a) Konstruera CS-logiken för ROM-kapseln. Den skall ha begynnelseadressen C000H.
Visa hur samtliga signaler skall kopplas mellan processorn och minneskapseln.

b) Konstruera CS-logiken för RWM-kapseln. Den skall ha begynnelseadressen 8000H.
Visa hur samtliga signaler skall kopplas mellan processorn och minneskapseln.

Inverterare och NAND-grindar med valfritt antal ingångar får användas. Fullständig adressavkodning skall användas.

13.4 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 32 kbyte ROM-kapsel, 1 st 2 kbyte ROM-kapsel, 1 st 4 kbyte RWM-kapsel, 1 st 8-bitars "three-state"-buffert som inport och ett 8-bitars register som utport. Modulerna har valts för att passa CPU12's timing.

32 kbyte ROM-kapseln skall placeras med början på adress 0000H.
4 kbyte RWM-kapseln skall placeras med början på adress 8000H.
2 kbyte ROM-kapseln skall placeras så att den slutar på adress FFFFH.

In- och utporten skall båda placeras på adressen C000H.

Uppgift: Konstruera den logik som krävs för "chip-select"-avkodningen. Endast grundläggande logikgrindar med valfritt antal ingångar får användas. Adressintervallet för varje modul skall anges i hexadecimal form. Använd ofullständig adressavkodning.

13.5 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 8 kbyte ROM-kapsel, 2 st 4 kbyte RWM-kapslar, 1 st 8-bitars "three-state"-buffert som inport och ett 8-bitars register som utport.

Placera ROM-kapseln så att sista adressen hamnar på adress FFFFH. Placera själv övriga komponenter i adressrummet på lämpligt sätt.

Rita adressavkodningslogik för samtliga komponenter. Det valda adressintervallet för varje komponent skall anges i hexadecimal form.

13.6 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 32 kbyte ROM-kapsel med slutadressen FFFFH, 1 st 16 kbyte RWM-kapsel med placering direkt före ROM-kapseln, 1 st 8-bitars "three-state"-buffert på adressen 0000H som inport och 1 st 8-bitars register på adressen 0001H som utport.

Modulerna har valts för att passa CPU12's timing.

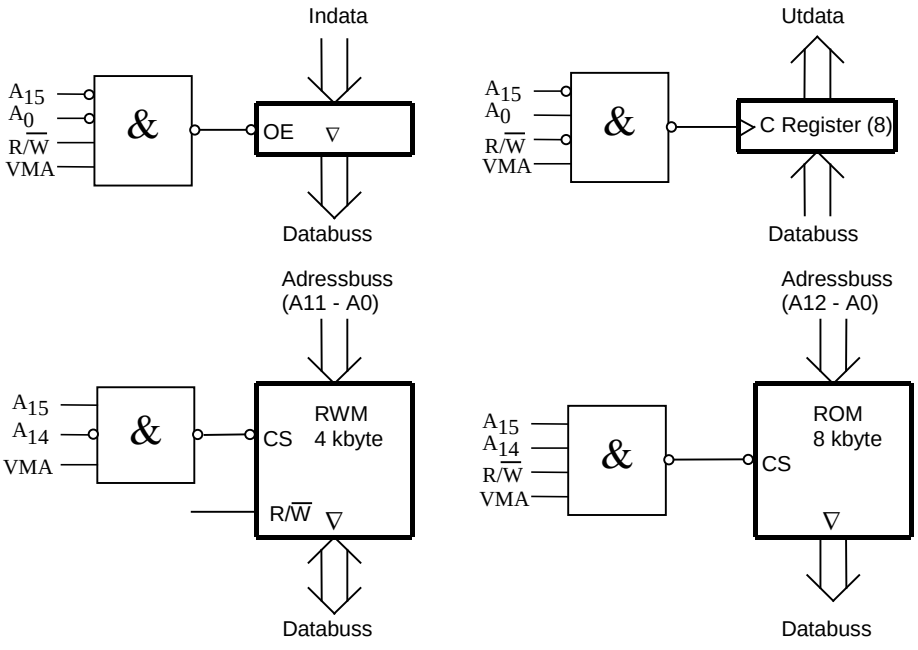
Uppgift: Konstruera den logik som krävs för "chip-select"-avkodningen. Använd fullständig adressavkodning. Endast grundläggande logikgrindar med valfritt antal ingångar får användas. Adressintervallet för varje modul skall anges i hexadecimal form.

13.7 Antag att ett mikrodatorsystem med CPU12 innehåller en ROM-kapsel och en RWM-kapsel för att lagra program och variabler, samt en 8-bitars "three-state"-buffert som inport och ett 8-bitars register som utport.

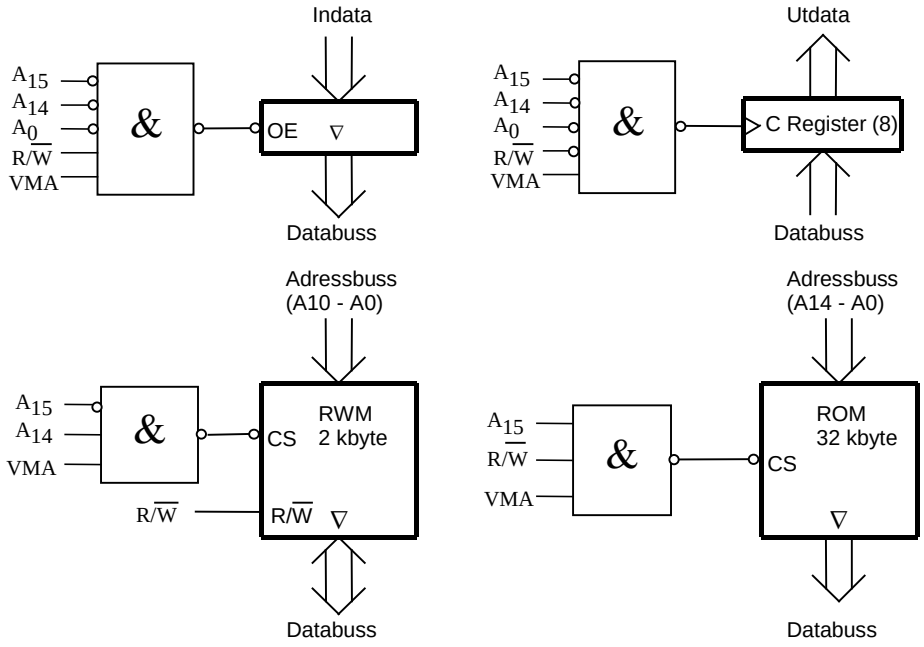
I figurerna a), b) och c) visas för tre fall hur dessa moduler är anslutna till processorbussen, samt hur “chip-select”-avkodningen utförs.

Redogör för var i adressrummet processorn kan läsa eller skriva i de olika enheterna. Alla addresser, där en enhet kan nås, skall redovisas.

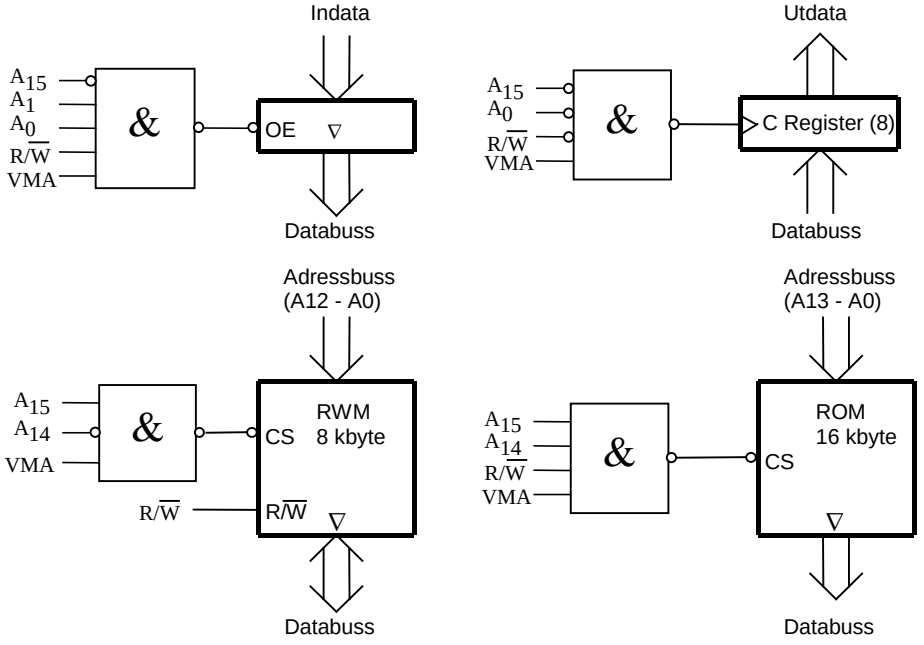
a)



b)


Indata
A15
A14
A0
R/W
VMA
&
OE
Databuss
Utdata
C Register (8)
Adressbuss
(A10 - A0)
RWM
2 kbyte
CS
Adressbuss
(A14 - A0)
ROM
32 kbyte

c)


Indata
A15
A1
A0
R/W
VMA
&
OE
Databuss
Utdata
C Register (8)
Adressbuss
(A12 - A0)
RWM
8 kbyte
CS
Adressbuss
(A13 - A0)
ROM
16 kbyte

13.8 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 16 kbyte ROM-kapsel, 1 st 16 kbyte RWM-kapsel, 1 st 8-bitars "three-state"-buffert som inport och ett 8-bitars register som utport.

ROM-kapseln skall placeras på de högsta adresserna i adressrummet.

Mikrodatorsystemet skall tillverkas i två olika versioner, en dyrare utbyggbar och en billigare som inte skall byggas ut. I den utbyggbara versionen kan i extremfallet samtliga adresser i adressrummet fyllas med minnesmoduler eller moduler för in- och utmatning.

Konstruera minimala logiknät för CS'-signalerna för:

a) Den utbyggbara, dyrare versionen.

b) Den billigare versionen, som inte skall byggas ut.

NAND-grindar med valfritt antal ingångar samt INVERTERARE får användas. Adressintervallet för varje modul skall anges i hexadecimal form. Modulerna har valts för att passa CPU12's timing.

Modulerna, förutom ROM-modulen, får placeras på lämpligt sätt i adressrummet. Alla program som är körbara i den dyrare versionen skall också kunna köras i den billigare.

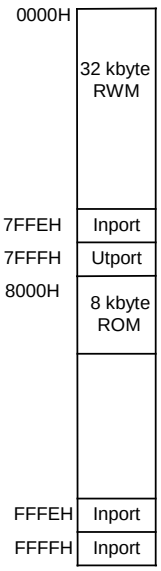
13.9 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 8 kbyte ROM-kapsel, 1 st 32 kbyte RWM-kapsel, 3 st 8-bitars "three-state"-buffertar som inportar och 1 st 8-bitars register som utport.

I figuren visas var i adressrummet modulerna skall placeras.
Modulerna har valts för att passa CPU12's timing.

Av figuren framgår det att inporten med adressen 7FFEH och utporten med adressen 7FFFH överlappar de två sista adresserna i RWM-modulen. För att avkodningen skall fungera måste därför inporten prioriteras så att en läsning på adressen 7FFEH inte aktiverar RWM-modulen.

Uppgift: Konstruera den logik som krävs för "chip-select"-avkodningen. Använd fullständig adressavkodning. Endast grundläggande logikgrindar med valfritt antal ingångar får användas. Adressintervallet för varje modul skall anges i hexadecimal form.

Inportarna på adresserna FFFEH och FFFFH skall användas på ett speciellt sätt.
Förklara hur du tror att de skall användas!

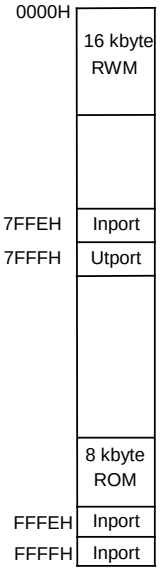


13.10 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 8 kbyte ROM-kapsel, 1 st 16 kbyte RWM-kapsel, 3 st 8-bitars "three-state"-buffertar som inport och 1 st 8-bitars register som utport.

I figuren visas var i adressrummet modulerna skall placeras. ROM-modulen skall placeras så att dess sista adress blir FFFFH. Modulerna har valts för att passa CPU12's timing.

Av texten och figuren framgår att inportarna på adresserna FFFEH och FFFFH överlappar de två sista adresserna i ROM-modulen. För att avkodningen skall fungera måste därför inportarna prioriteras s läsningar på adresserna FFFEH och FFFFH inte aktiverar ROM-modulen.

Uppgift: Konstruera den logik som krävs för "chip-select"-avkodningen. Använd fullständig adressavkodning. Endast grundläggande logikgrindar med valfritt antal ingångar får användas. Adressintervallet för varje modul skall anges i hexadecimal form.

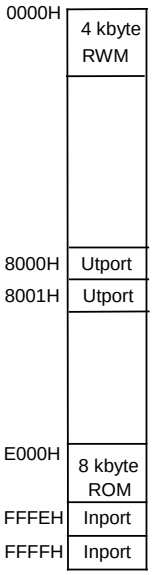


13.11 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 8 kbyte ROM-kapsel, 1 st 4 kbyte RWM-kapsel, 2 st 8-bitars "three-state"-buffertar som inportar och 2 st 8-bitars register som utportar.

I figuren visas var i adressrummet modulerna skall placeras.
Modulerna har valts för att passa CPU12's timing.

Av figuren framgår att inportarna på adresserna FFFEH och FFFFH överlappar de två sista adresserna i ROM-modulen. För att avkodningen skall fungera måste därför inportarna prioriteras så att läsningar på adresserna FFFEH och FFFFH inte aktiverar ROM-modulen.

Uppgift: Konstruera den logik som krävs för "chip-select"-avkodningen. Använd fullständig adressavkodning. Endast grundläggande logikgrindar med valfritt antal ingångar får användas. Adressintervallet för varje modul skall anges i hexadecimal form.



13.12 Ett mikrodatorsystem skall konstrueras med CPU12, 1 st 32 kbyte ROM-kapsel och 1 st 32 kbyte RWM-kapsel!

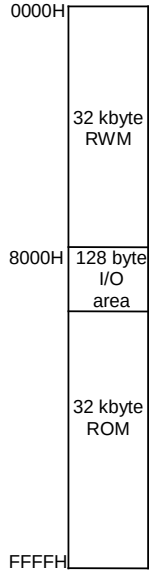
Inom ROM-kapselns adressområde skall dessutom i början finnas ett adressintervall för in- och utmatn

I figuren visas var i adressrummet modulerna skall placeras. Vissa moduler har valts för att passa CPU1 timing. Utportarna har negativ flanktriggning.

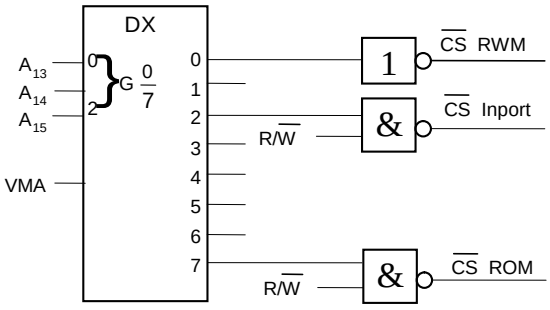
För att undvika busskollisioner måste inportarna prioriteras så att läsningar i de adressintervall, som innehåller eller kan innehålla inportar, inte aktiverar ROM-modulen.

Uppgift: Konstruera CS-logiken för RWM- och ROM-modulen. Använd fullständig adressavkodning. Endast grundläggande logikgrindar med valfritt antal ingångar får användas. Det användbara adressint vallen för de två minnesmodulerna skall anges i hexadecimal form.

CS-signaler för inportar och utportar behöver inte konstrueras!



13.13 För ett CPU12-system med 1 st 16 kbyte RWM och 1 st 4 kbyte ROM har CS-logiken i figuren konstruerats. Analysera den!



a) Ange hur adressrummet utnyttjas.

b) Förklara vad det är för fördel med att använda en fördelare (demultiplexer) för adressavkodning.

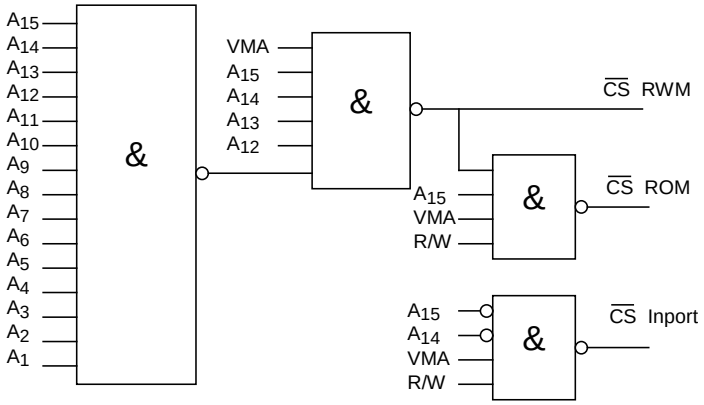
c) Vad är mindre lyckat med kopplingen i figuren?

d) Visa hur man kan åtgärda "felet" i **c)**.

e) Anslut en utport till systemet. Använd ofullständig adressavkodning så att porten ser ut att finnas på var och en av adresserna 4000H - 5FFFH.

f) Anslut ytterligare port, en inport, till adress 60F0H. För den skall adressavkodningen vara fullständig och realiseras den m h a fördelaren.

13.14 I samband med ett CPU12-system har man funnit CS-logiken i figuren. I systemet finns 1 st 2 kbyte RWM och 1 st 4 kbyte ROM. Analysera CS-logiken!



a) Visa hur adressrummet utnyttjas.

b) Vad finns det för skäl till denna disposition av adressrummet?

c) Visa hur man kan koppla in en utport till detta system. Välj lämplig(a) adress(er).

d) Vilka värden får N, Z, V och C- flaggan i CC-registret efter följande sekvens av instruktioner?

```
LDAA   $8000      (Fungerar på samma sätt som i FLISP, med undantaget att adresserna har 16 bitar.)
CMPA   $C000
```

Gör lämpliga antaganden, som stämmer överens med datorsystemet ovan.