

KOMPLEMENT TILL

"ARBETSBOK FÖR DIGIFLISP" (BLÅ BOK)

FÖR ANVÄNDNING MED NY PROGRAMVARA DIGIFLISP 9.

1 Transistorn som strömställare

I detta första kapitel behandlar vi transistorn som en grundläggande byggsten. Vi kommer inte att beröra transistorens fysikaliska egenskaper dvs. ämnet *halvledarteknik*, utan enbart transistorens förmåga att uppträda som en strömställare. Genom att studera några enkla transistorkopplingar kommer du samtidigt att bekanta dig med logikfunktioner som är centrala inom digitaltekniken.

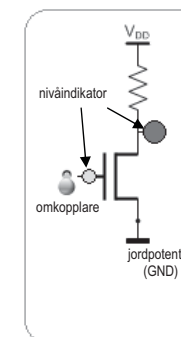
1.1 Spänningsnivåer och logikvärden

Inom digitaltekniken arbetar man med två olika värden, 1 och 0. Dessa så kallade *logikvärden* (sanningsvärden) motsvaras i digitala elektronikkretsar av två olika spänningsnivåer: vi kallar dem här V_{DD} respektive GND. Symbolen V_{DD} används för att indikera en *hög* spänningspotential, och motsvarande logikvärde är 1. Symbolen GND, eller *jordpotential*, används för logikvärde 0.

Omkopplarna anger en punkt som kan kopplas antingen direkt till V_{DD} , eller direkt till GND i signalvägen. Genom att klicka på en omkopplare kan du ändra dess läge och därmed logiknivån i en punkt i signalvägen.

-  Uppåt, internt kopplad till V_{DD} , logiknivå 1
-  Nedåt, internt kopplad till GND, logiknivå 0

Signalnivåer kan avläsas på de små nivåindikatorerna som är placerade omedelbart i anslutning till signalvägen. På vissa ställen, vanligtvis utsignaler, används något större indikatorer. I DigiFlisp använder vi grön eller röd färg för att indikera V_{DD} , logikvärdet 1, medan vi konsekvent använder ljusgrå färg för att indikera logikvärdet 0.



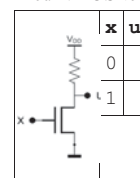
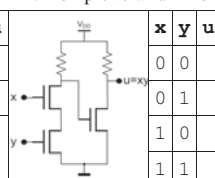
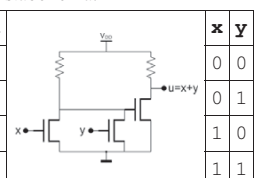
Uppgift 1.1

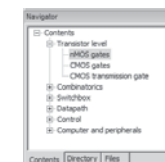
I Navigator|Contents, märk ditt val,

Transistor level | nMOS gates.

genom att högerklicka, välj Open.

Använd simulatormen och analysera följande kopplingar uppbyggda med NMOS-teknik. Komplettera funktionstabellerna.

	<table><tr><th>x</th><th>u</th></tr><tr><td>0</td><td></td></tr><tr><td>1</td><td></td></tr></table>	x	u	0		1										
x	u															
0																
1																
	<table><tr><th>x</th><th>y</th><th>u</th></tr><tr><td>0</td><td>0</td><td></td></tr><tr><td>0</td><td>1</td><td></td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td></td></tr></table>	x	y	u	0	0		0	1		1	0		1	1	
x	y	u														
0	0															
0	1															
1	0															
1	1															
	<table><tr><th>x</th><th>y</th><th>u</th></tr><tr><td>0</td><td>0</td><td></td></tr><tr><td>0</td><td>1</td><td></td></tr><tr><td>1</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td></td></tr></table>	x	y	u	0	0		0	1		1	0		1	1	
x	y	u														
0	0															
0	1															
1	0															
1	1															



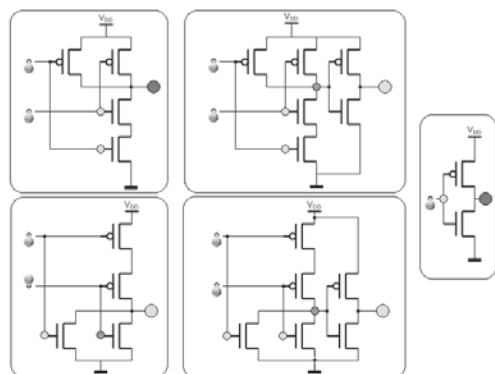
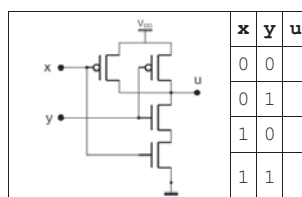
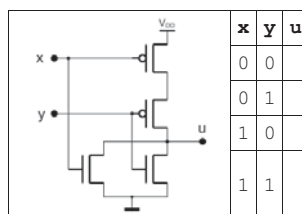
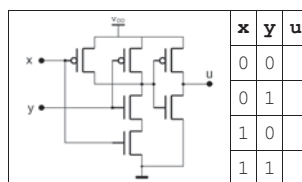
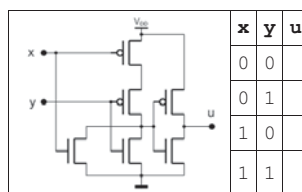
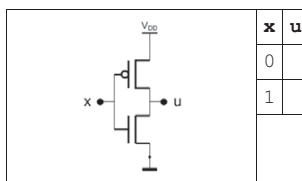
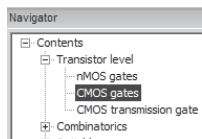
Uppgift 1.2

Vi studerar kopplingar i CMOS-teknik, välj denna gång

Navigator|Contents:

Transistor level | CMOS gates.

Identifiera nu följande kopplingar i simulatorm, analysera kopplingarna genom att variera logiknivåerna på ingångarna och iaktta logiknivåerna på utgångarna. Komplettera funktionstabellerna med utsignalernas logiknivåer (1 eller 0) för varje koppling.



2 Grindar

Grindar realiserar logikfunktioner och är de byggstenar vi använder för att illustrera större och större logikblock som så småningom resulterar i en komplett dator.

2.1 Enkla grindar

I simulatorns Navigator|Contents:

Combinatorics | Elementary logic functions

finns grindar som realiserar de grundläggande logikfunktionerna.

Omkopplarna kan du klicka på för att ändra nivåerna hos grindarnas ingångar. Ingångarnas nivåer kan avläsas på de små indikatorerna omedelbart till vänster om varje grind.

Utgångarnas nivåer läser du av på de större indikatorerna till höger om respektive grind.

Uppgift 2.1

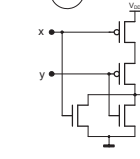
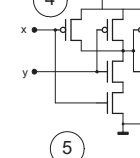
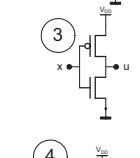
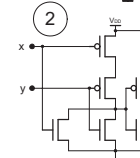
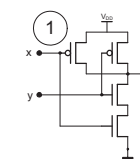
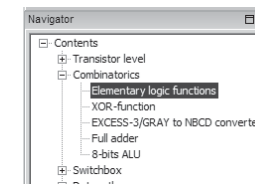
Välj: Navigator|Contents:

Combinatorics | Elementary logic functions

Verifiera, med hjälp av simulatorm, logikfunktionerna som visas i Symbol-kolumnen i tabellen.

- Fyll i funktionstabellen till höger.
- Jämför funktionstabellerna med dina resultat från Uppgift 1.2 och identifiera motsvarande CMOS-koppling (figur 1–5 i marginalen)
- Ange slutligen grindens engelska namn (logiknamn).

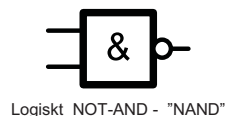
Symbol	CMOS-koppling (1-5)	Grind (logiknamn)	Funktionstabell		
			x	u	
			0		
			1		
			x	y	u
			0	0	
			0	1	
			1	0	
			1	1	
			x	y	u
			0	0	
			0	1	
			1	0	
			1	1	



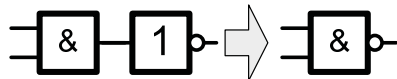
2.2 Grindar med sammansatta logikfunktioner

Grindar som kombinerar olika logikfunktioner är vanliga. Speciellt viktiga är de grindar som realiserar "NOT-AND" (NAND) respektive "NOT-OR" (NOR).

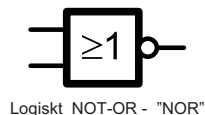
Funktionen NAND bildas genom att en INVERTERARE ansluts direkt till utgången på AND-grinden. Denna sammansatta funktion indikeras i symbolen för NAND genom att utgången försetts med en liten ring:



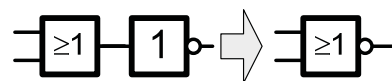
Logiskt NOT-AND - "NAND"



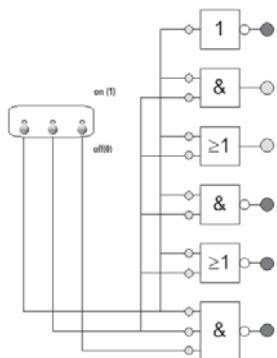
På motsvarande sätt bildas funktionen NOR:



Logiskt NOT-OR - "NOR"



Förutom kombinerade grindar kan vi också skapa grindar med flera ingångar som exempelvis en 3-ingångars NAND-grind.



Uppgift 2.2

Verifiera, med hjälp av simulatorm, (Combinatorics | Elementary logic functions) de sammansatta logikfunktionerna NAND och NOR.

Fyll i funktionstabellen till höger, i kolumnen Grind logiknamn anger du symbolens engelska logiknamn och i kolumnen Boolesk funktion ger du det booleska uttrycket för utsignalen u.

Symbol	Grind logiknamn	Boolesk funktion	Funktionstabell			
x y			x	y	u	
			0	0		
			0	1		
			1	0		
x y			x	y	u	
			0	0		
			0	1		
			1	0		
x y z			x	y	z	u
			0	0	0	
			0	0	1	
			0	1	0	
			0	1	1	
			1	0	0	
			1	0	1	
			1	1	0	
			1	1	1	

"EXKLUSIVT ELLER" (XOR) är ett annat viktigt exempel på en grundläggande sammansatt logikfunktion.

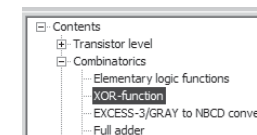
Uppgift 2.3

Välj (Navigator|Contents):

Combinatorics | XOR-function

Analysera symbolerna och använd simulatorm för att ge de svar som krävs för att komplettera följande tabell.

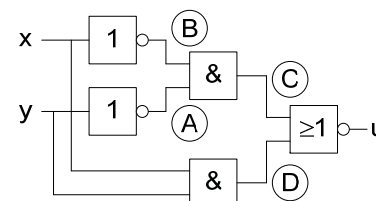
Symbol	Grind logiknamn	Boolesk Funktion	Funktionstabell		
			x	y	u
x y			0	0	
			0	1	
			1	0	
			1	1	
x y			x	y	u
			0	0	
			0	1	
			1	0	
			1	1	



XOR-funktionen kan realiseras på olika sätt, med hjälp av enkla grindfunktioner. Simulatorm vägleder dig i följande uppgifter.

Uppgift 2.4

Analysera följande koppling.



Uttryck nu funktionerna A, B, C, D och u som booleska funktioner av insignalerna x och y.

A=f(y)= $\overline{y}$	B=f(x)= $\overline{x}$	C=f(x,y)=
D=f(x,y)=	u=f(x,y)=	

x	y	A	B	C	D	u
0	0	1				
0	1					
1	0					
1	1					

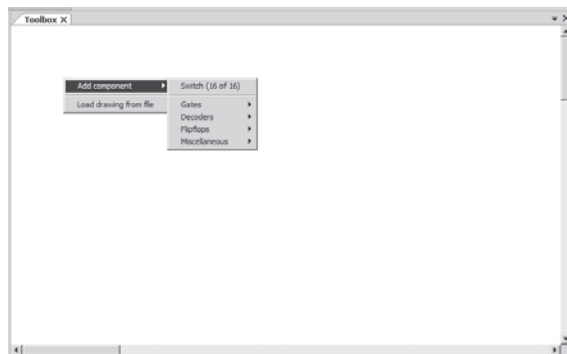
Använd simulatorm och studera nätets signaler (A,B,C,D och u) för olika insignaler x och y. Komplettera tabellen i marginalen och jämför med dina funktioner ovan.

3 Kopplingsboxen

I detta kapitel ges en introduktion till *kopplingsboxen*, och efter att ha arbetat igenom kapitlet kommer du att självständigt kunna koppla och analysera grindnät med varierande komplexitet.

Välj Switchbox|New switchbox.

Kopplingsboxen består av en arbetsyta som inledningsvis är tom. Genom att placera markören i det vita arbetsfältet och högerklicka, kan du välja någon av kopplingsboxens komponenter och därefter placera ut denna någonstans på arbetsytan.



3.1 Inledande demonstration

Uppgift 3.1

Du ska börja med att verifiera de viktiga *kommutativa* lagarna inom boolesk algebra med hjälp av kopplingsboxen, se punkt 1 i figuren i marginalen.

Det första sambandet säger att:

$$x \times y = y \times x$$

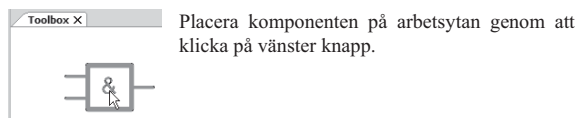
dvs. att:

$$x \text{ AND } y = y \text{ AND } x$$

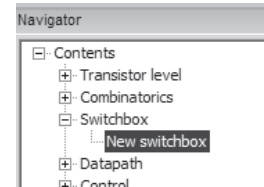
Det krävs alltså nu två AND-grindar, en för varje led. Välj

Add component | Gates | 2-input AND

Flytta nu ut markören i kopplingsboxens arbetsyta; du ska se komponentens siluett.



Placera komponenten på arbetsytan genom att klicka på vänster knapp.



Satser från boolesk algebra

1. Kommutativa lagar

$$x \times y = y \times x$$

$$x + y = y + x$$

2. Distributiva lagar

$$x \times (y + z) = x \times y + x \times z$$

$$x + (y \times z) = (x + y) \times (x + z)$$

3. $x + 0 = x$

$$x \times 1 = x$$

4. $x + \overline{x} = 1$

$$x \times \overline{x} = 0$$

5. $x + 1 = 1$

$$x \times 0 = 0$$

6. $x + x = x$

$$x \times x = x$$

7. Associativa lagar

$$x + (y + z) = (x + y) + z$$

$$x \times (y \times z) = (x \times y) \times z$$

8. De Morgans lagar

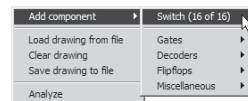
$$\overline{(x + y)} = (\overline{x} \times \overline{y})$$

$$\overline{(x \times y)} = (\overline{x} + \overline{y})$$

9. $\overline{(\overline{x})} = x$

Multiplikationstecknet ($\times$) utelämnas oftast där det inte kan missförstås och vi skriver exempelvis enklare:

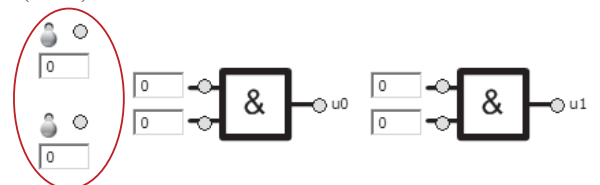
$$xy \text{ i stället för } x \times y$$



Komponenten ritas nu om och får dessutom små indikatorer som visar signalnivåer (0 eller 1) på ingångar och utgång. Varje ingång har ett litet "fönster" där du kan skriva in såväl konstanta värden, 0 eller 1, som något godtyckligt variabelnamn. För nya komponenter är detta värde alltid 0. Komponentens utgång har ett namn som bestäms av kopplingsboxen, i detta fall "u0". Du kan använda denna utsignal om du vill koppla utgången till en annan komponents ingång.

Placera ut ytterligare en AND-grind.

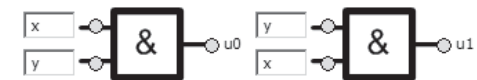
Du kan skapa oberoende variabler med hjälp av *strömställare* (switch). Placera ut två strömställare.



Definiera nu två variabelnamn x och y genom att ange dessa i strömställarnas kontrollfönster (se marginalen).

Variabelnamn som du har deklarerat på detta sätt kan nu användas som insignaler till komponenter i kopplingsboxen.

Ge nu insignaler till de båda AND-grindarna genom att placera markören i det fönster som hör till ingången och skriva in namnet på den variabel som ska kopplas till ingången.

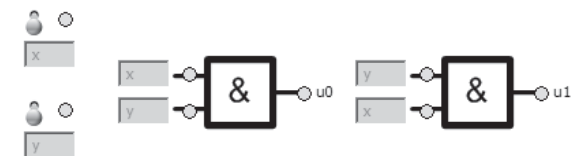


Notera att den första grinden realiserar $x \text{ AND } y$, medan den andra grinden realiserar $y \text{ AND } x$.

Det är nu dags att *simulera* kopplingen.

Högerklicka någonstans i kopplingsboxens arbetsyta och välj Analyze (se marginalen).

Nu händer flera saker: Fönster som tidigare kunde redigeras, blir grå och kan inte längre ändras. Du kan inte heller placera ut nya komponenter i detta läge.



Menyvalet Analyze byter namn till Design. Du gör detta val för att återgå till det läge där du kan redigera din koppling. Dessutom har du menyvalet Clock, som vi strax återkommer till (se marginalen).



Du kan nu ändra de oberoende variabelnas värden genom att klicka på strömbrytarena. För att verifiera den första kommutativa lagen ($x \cdot y = y \cdot x$) måste du undersöka de båda utsignalerna u0 och u1 för varje insignalsskombination. Gör detta och fyll i följande tabell.

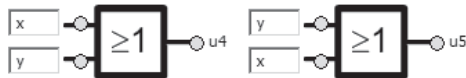
x	y	u0	u1
0	0		
0	1		
1	0		
1	1		

Den andra satsen av de kommutativa lagarna säger att $x + y = y + x$, vilket innebär att två OR-grindar ska användas. Du ska nu också verifiera sambandet $(\overline{\overline{x}}) = x$ genom att koppla samman två inverterare.

Uppgift 3.2

För att ändra eller bygga vidare på din koppling klickar du på Design. Du får då tillbaka de vita ändringsbara fälten.

Placera ut två OR-grindar på följande sätt:



Om du har kvar dina AND-grindar och strömställare kommer OR-grindarnas utgångar här att tilldelats signalnamnen u4 och u5 av kopplingsboxen.

För varje kombination av insignaler, fyll i utsignalerna u4 och u5 i följande tabell:

x	y	u4	u5
0	0		
0	1		
1	0		
1	1		

Verifiera nu sambandet $(\overline{\overline{x}}) = x$ genom att koppla samman två inverterare enligt följande figur. Dvs. du kopplar samman utgången från den första inverteraren med ingången till den andra genom att skriva utsignalens namn i den andra inverterarens insignalfönster. Komplettera avslutningsvis följande tabell:

x	$(\overline{\overline{x}})$	$(\overline{\overline{\overline{x}}})$
0		
1		

uX, namn på utgångssignaler som tilldelas av kopplingsboxen i den ordning komponenter sätts ut. De kallas också *beroende variabler*.
 Oberoende variabler är de (godtyckliga) variabelnamn du själv inför med hjälp av strömställare.

3.2 En översikt av kopplingsboxen

Med kopplingsboxen kan du simulera enkla uppkopplingar av digitala nät. Du har 26 olika komponenter att välja mellan (se några exempel på symboler i marginalen). Du kan också använda upp till 16 *oberoende invariabler* i form av strömställare.

Kopplingsboxen har två olika lägen, för redigering eller simulering.

Redigering – Här kan du placera ut komponenter på arbetsytan. En komponent kan ha en eller flera insignaler som måste definieras, insignalerna ska vara någon av:

- Konstanten 1
- Konstanten 0
- uX (eller qX) utsignalen från någon komponent (X) i nätet
- någon *oberoende variabel*, som du själv definierar genom att placera ut en strömställare och skriva in variabelns symboliska *namn* i ett inmatningsfönster. Namnet får bestå av högst 4 ASCII-tecken som får vara A–Z och/eller 0–9. Första tecknet i variabelnamnet måste vara en bokstav.



I redigeringsläget har alla inmatningsfönster vit bakgrund och innehållet kan ändras. Ändring av komponenters insignaler reflekteras dock först då kopplingsboxen försätts i simuleringsläge.

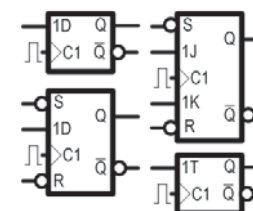
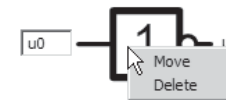
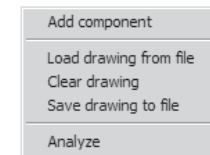
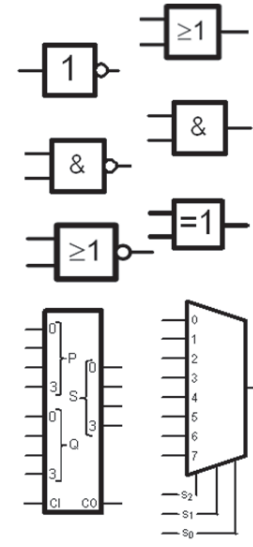
Redigeringsläget tillåter också att du

- Sparar kopplingen (Save drawing to file)
- Laddar en tidigare sparad koppling (Load drawing from file)
- Rensar arbetsytan (Clear drawing)

Simulering – Då du är färdig med nätet, dvs. har placerat ut dina komponenter, redigerat insignalerna till varje komponent och dessutom definierat de oberoende variablerna du använder, väljer du på Analyze. Kopplingen kontrolleras då och alla utsignaler bestäms. Du kan nu ändra oberoende variabels värden, genom att klicka på strömställare, och studera hur nätets tillstånd påverkas av dessa. Du kan inte ändra innehållet i ett inmatningsfönster (dessa är grå). I simuleringsläget kan du inte heller lägga till eller ta bort komponenter, du måste återgå till redigeringsläget för att göra detta.

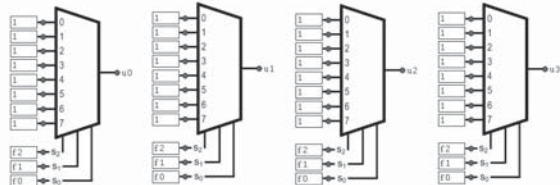
Vill du flytta eller ta bort en komponent, högerklickar du med markören över komponenten. Detta gäller både redigerings- och simuleringsläge.

Klocksignal – Några komponenter har en *klockingång* C. I simuleringsläge kan du generera en (gemensam) klocksignal för dessa komponenter genom att klicka på Clock. Klocksignalen har ingen inverkan på komponenter utan klockingång.



Uppgift 7.3

Starta kopplingsboxen, placera ut 4 st. 8-ingångars väljare så att deras respektive utgångar får namnen, u0, u1 u2 och u3. Definiera funktionssignalerna f2, f1 och f0, samt för in dessa hos respektive väljare:



Spara kopplingen under filnamnet alu4.tb. Spara kopplingen fortsättningsvis allt eftersom du bygger vidare på ALU:n.

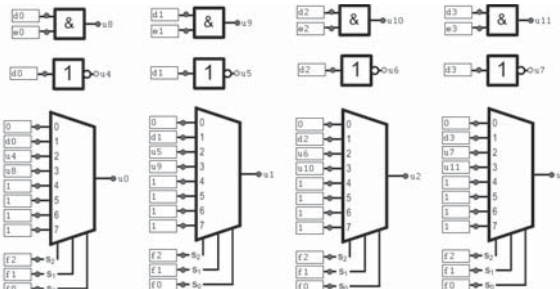
Vi börjar implementera ALU:ns funktioner, dvs. bilda insignalerna för väljarnas ingångar. För dessa funktioner väntar vi tills vidare med att bilda flaggorna N,Z,V och C.

Uppgift 7.4

Implementera funktionerna 0 t.o.m 3, dvs följande:

funktion			operation	utsignaler			
f ₂	f ₁	f ₀		u ₃	u ₂	u ₁	u ₀
0	0	0	RTN	u ₃	u ₂	u ₁	u ₀
0	0	1	$U=0 \rightarrow U$	d ₃	d ₂	d ₁	d ₀
0	1	0	$U=D_{1k} \rightarrow U$	$\overline{d_3}$	$\overline{d_2}$	$\overline{d_1}$	$\overline{d_0}$
0	1	1	$U=D \wedge E$	$d_3 \wedge e_3$	$d_2 \wedge e_2$	$d_1 \wedge e_1$	$d_0 \wedge e_0$

Placera komponenterna så tätt som möjligt...



Fortsätt nu med att även definiera insignalerna d3, d2, d1 och d0, respektive e3, e2, e1 och e0 och slutligen cin.

9 Transmissionsgrinden

Vi inför här två nya logiknivåer (*högimpedanstillstånd* och *odefinierat tillstånd*) som kan uppträda i nät där vi vill koppla samman utgångar från två eller flera olika kretsar.

De olika logiknivåerna illustreras i simulatorm enligt följande:

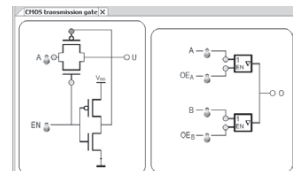
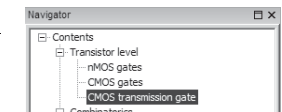
- Röd, kopplad till V_{DD} , logiknivå '1'
- Grå, kopplad till GND, logiknivå '0'
- Vit, högimpedanstillstånd, inte kopplad till vare sig V_{DD} eller GND, logiknivå 'Z'
- Svart, odefinierat tillstånd. Punkten utgör en kortslutning mellan V_{DD} och GND, logiknivå 'X'.

Uppgift 9.1

Välj Transistor level | CMOS transmission gate.

Figuren till höger visar hur en signal A kopplas via en transmissionsgrind, styrd av signalen EN, till punkten U. Undersök logiknivån i punkten U och fyll i följande funktionstabell:

A	EN	U
0	0	
1	0	
0	1	
1	1	



I figuren till höger visas hur två oberoende signaler A och B kopplas samman i punkten O, via var sin transmissionsgrind. Undersök kopplingen och fyll i följande tabell med de resulterande logiknivåerna.

A	B	OE _A	OE _B	O
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	

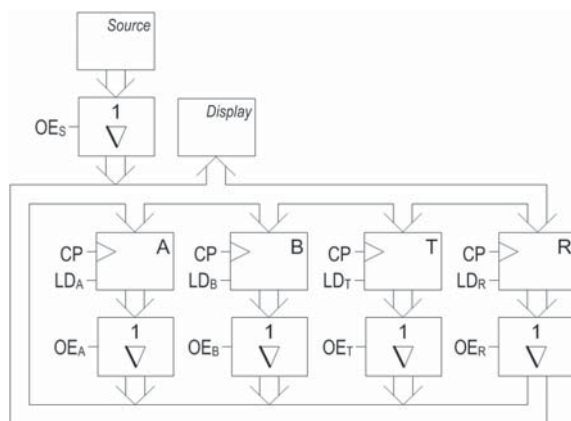
10 Registeröverföring

Vi har tidigare sett hur en ALU fungerar, hur data kan lagras i minneselement som vi med ett gemensamt ord kan kalla *register*, hur vi kan skriva nya data till ett sådant register och hur vi kan läsa från registret. Vi ska nu också se hur flera register kan kopplas samman via *bussar* och därmed påbörja konstruktion av en *dataväg*.

Med hjälp av datavägen och ALU:n ska registerinnehåll kunna bearbetas på olika sätt. Data måste därför kunna överföras *till* och *från* ALU:n där den egentliga bearbetningen utförs. Därför studerar vi först hur data kan flyttas runt i datavägen *från* ett register *till* ett annat register *via* en databuss. För ändamålet krävs ett antal styrsignaler; vi kallar detta *registeröverföring*.

Styrsignalerna talar om *varifrån* data hämtas och *vart* data ska placeras. Något oegentligt kallas detta ofta att "flytta" data. Det är inte det som händer, utan egentligen *kopierar* vi data från ett ställe (källan) till ett annat ställe (destinationen). Styrsignaler kan genereras på olika sätt, och vi återkommer till detta längre fram. Tills vidare anger vi styrsignaler i tabellform med penna och papper eller klickar på olika symboler i simulatorm.

En koppling för överföring av data mellan olika register visas i Figur 10.1 nedan. Observera hur ingångar och utgångar förbinds med en enda buss för att vi ska kunna flytta data från ett register till ett annat.



Figur 10.1 Dataöverföring mellan register via en buss

Vi ska nu studera hur kopiering av data från ett register till ett annat går till. När man vill flytta (kopiera) data från ett register måste registrets OE-signal aktiveras. För registret, vars innehåll ska modifieras, måste LD-signalen aktiveras. Om vi, som exempel, vill kopiera data som finns i register R till register A måste följande styrsignaler ges:

- **OE_R = 1** Innehållet i register R kopplas till bussen via three-state-bufferten. Därmed finns det på ingångarna till samtliga register.
- **LD_A = 1** Vid nästa positiva klockpulsflank laddas register A med det som finns på bussen och får samma innehåll som register R.

RTN för operationen är:

$R \rightarrow A$ (= innehållet i register R kopieras till register A)

Då signalerna OE_R och LD_A är aktiva, verkställs dataöverföringen vid nästa positiva klockpulsflank. Lägg märke till att alla enheter i systemet är anslutna till samma klocksignal CP; alla register klockas följaktligen samtidigt. Operationen beskrivs också i form av styrsignaler för datavägen i följande tabell. Samtliga styrsignaler med värdet 1 i raden aktiveras under en klockcykel. En rad i tabellen avser alltså en klockcykel.

OE _S	OE _A	OE _B	OE _T	OE _R	LD _A	LD _B	LD _T	LD _R	RTN-beskrivning
0	0	0	0	1	1	0	0	0	$R \rightarrow A$

Uppgift 10.1

Styrsignaler för enkel dataväg

Fyll i styrsignalvärdena för överföringen $A \rightarrow R$ i följande tabell.

OE _S	OE _A	OE _B	OE _T	OE _R	LD _A	LD _B	LD _T	LD _R	RTN-beskrivning
									$R \rightarrow A$

Uppgift 10.2

Styrsignaler för enkel dataväg

Fyll i styrsignalvärdena för överföringarna:

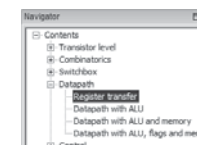
$A \rightarrow T, B \rightarrow A, T \rightarrow B$ ($A \leftrightarrow B$)

i följande tabell.

OE _S	OE _A	OE _B	OE _T	OE _R	LD _A	LD _B	LD _T	LD _R	RTN-beskrivning
									$A \rightarrow T$
									$B \rightarrow A$
									$T \rightarrow B$

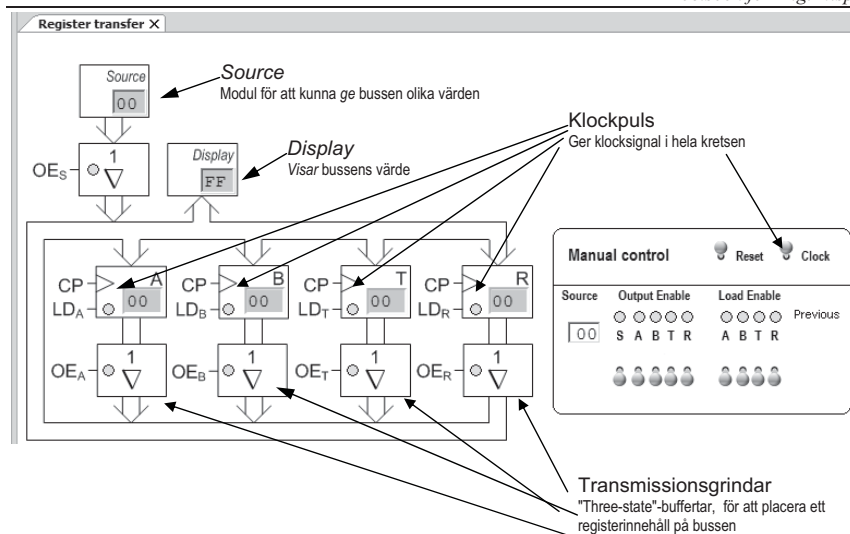
Uppgift 10.3

Välj alternativet Datapath | Register transfer.



Exempel på RTN-symboler, fullständig tabell finns i appendix

N _r	Konstanten N uttryckt i talbasen r.
M(N _r)	Minnesinnehåll på adressen N _r .
→	Kopiering
Följande symboler är beteckningar som reserverats för register	
A	Register A
T	Register T
R	Register R



Figur 10.2 Dataöverföring mellan register

Simulatorns Register transfer, Figur 10.2 visar en dataväg, ett antal register (A, B, T och R), en Data source modul (Source) och en Bus display modul (Display). Respektive moduls styr signaler (LD, OE) kan sättas till 0 eller 1 med strömställarna i den manuella styrenheten.

Genom att klicka på brytaren Clock ger du en signal på samtliga klockingångar (CP) vilket då försätter kretsen i ett "nästa tillstånd".

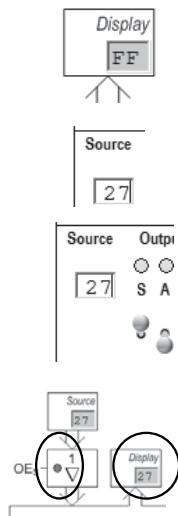
Raden Previous visar indikatorer för att hjälpa dig minnas hur du ställde styrsignalerna innan du klickade på Clock. En klickning på klockpuls innebär att nätets aktiverade register klockar in det som för tillfället finns på bussen.

1. Observera att bussens värde FF₁₆ visas i Display-modulen. Detta kan tolkas som att bussen är i högimpedanstillstånd och att ingen enhet för tillfället lägger ut något värde på bussen.

2. Skriv in 27₁₆ i Source-modulen. Enklart är att märka upp siffrorna i fönstret med musen och sedan skriva in ett nytt värde.

3. Aktivera nu styrsignalen OE_S (klicka på symbolen för strömställaren) och....

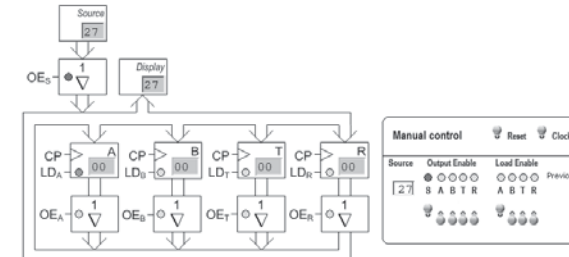
4. ...notera hur transmissionsgrinden för Source-modulen aktiveras och att bussens värde 27₁₆ nu också visas på Display-modulen.



5. Klicka på Clock

... notera hur OE_S-signalens indikator "föregående" styrsignal nu tänds. Detta har ingen annan praktisk betydelse än att hjälpa dig komma ihåg vilka signaler som var aktiva vid den senaste klockpuls.

6. Aktivera nu även LD_A och observera hur de aktiva styrsignalerna märks ut.



7. Ge slutligen ytterligare en klockpuls: Vad innehåller register A?

För att flytta runt data mellan de olika registren krävs alltså att man först ställer in lämpliga styrsignalerna, och därefter verkställer dataflyttningen med en klockpuls. Använd nu simulatorns Register transfer och lös följande uppgifter.

Uppgift 10.4

- Sätt signalen OE_S till 1
- Ändra innehållet i Source-modulen
- Observera vad som händer på bussen via Display-modulen.

Ställ nu in värdet 38₁₆ i Source-modulen, aktivera signalen LD_R. Händer det något med innehållet i register R?

Ge en klockpuls: Händer det nu något med innehållet i register R?

Ge en RTN-beskrivning av operationen.

Uppgift 10.5

Värdet 71_{16} kan placeras i samtliga register A, T och R under en klockcykel. Ge en RTN-beskrivning av operationen

Ange det nya värdet i Source-modulen samt de styrsignaler som måste aktiveras för operationen.

Source	OE _S	OE _A	OE _B	OE _T	OE _R	LD _A	LD _B	LD _T	LD _R
--------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

Kontrollera din lösning med hjälp avsimulatorn.

Uppgift 10.6

Du ska nu undersöka vad som händer om du aktiverar två (eller flera) OE-signaler samtidigt.

Placera värdet $5C_{16}$ i register A och värdet 21_{16} i register T. Lägg ut båda dessa registerinnehåll till bussen genom att aktivera signalerna OE_A och OE_T, och studera bussens värde i displaymodulen.

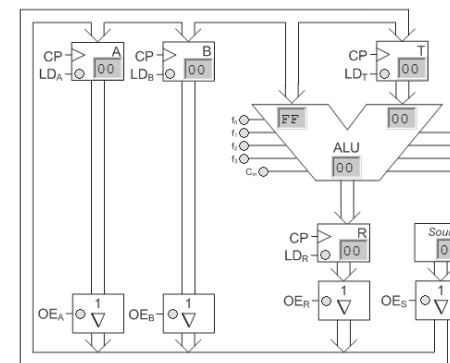
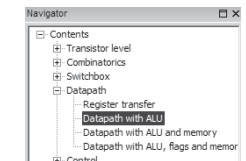
Vilket resultat får du?

Vad kan dra för slutsatser om *bussens* värde då flera moduler samtidigt *driver bussen* (skriver ut på bussen)?

$53_{16} \rightarrow A, 21_{16} \rightarrow T$
OE_A; OE_T

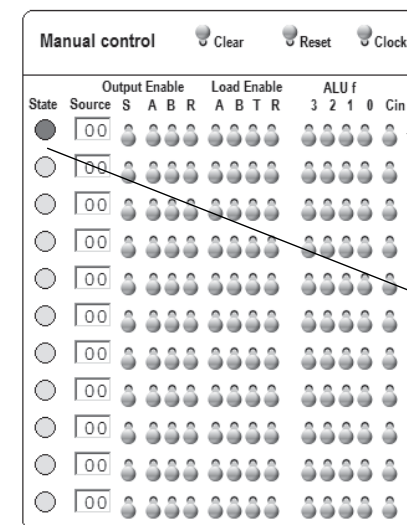
11 Dataväg med ALU

I detta kapitel kompletteras den enkla datavägen vi hittills använt oss av med en ALU. Med vår modifierade arkitektur, se Figur 11.1, och den enkla manuella styrenheten i Figur 11.2, ska vi nu utföra enkla databearbetningar som en sekvens operationer kontrollerade av styrsignaler från styrenheten. I detta kapitel studerar vi alltså speciellt ALU:ns användning i datavägen.



f_2	f_1	f_0	funktion	operation
0	0	0	RTN	$U=0$
0	0	1		$U=FD_{16}$
0	1	0		$U=FE_{16}$
0	1	1		$U=FF_{16}$
0	1	0		$U=E$
0	1	1		$U=D_{16} + C_{in}$
0	1	1		$U=D \vee E$
0	1	1		$U=D \wedge E$
1	0	0		$U=D \oplus E$
1	0	1		$U=D + C_{in}$
1	0	1		$U=D + FF_{16}$
1	0	1		$U=D + E + C_{in}$
1	1	0		$U=D + E_{16} + C_{in}$
1	1	0		$U=D < 1 (C_{in})$
1	1	1		$U=(C_{in}) D >> 1$
1	1	1		$U=(d_7) D >> 1$

Figur 11.1 Dataväg med ALU och dess funktionstabell



"Fjädrande" strömbrytare för nollställning, återställning och klockpuls

11 uppsättningar tvåvägs omkopplare för styrsignaler.
Varje uppsättning (steg) motsvarar en klockpuls.
Den manuella styrenheten kan därför utföra "program" om maximalt 11 klockpulser.

Längst ut till vänster finns en indikator som är aktiv (röd) för det steg som står i tur att utföras.
För varje steg finns också en Source-modul som kan användas för att ge indata i steget.

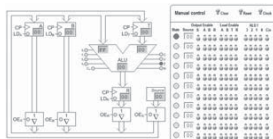
Figur 11.2 Manuell styrenhet för dataväg med ALU

Exempel på RTN-symboler, fullständig tabell finns i appendix	
N _r	Konstanten N uttryckt i talbasen r.
M(N _r)	Minnesinnehåll på adressen N _r .
→	Kopiering
Följande symboler är beteckningar som reserverats för register	
A	Register A
B	Register B
T	Register T
R	Register R
Operatörer	
+	Addition
-	Subtraktion
^	Logiskt "OCH" (AND)
∨	Logiskt "ELLER" (OR)
⊕	Logiskt "EXKLUSIVT ELLER" (XOR)
Opr<<d	"Opr" skiftas vänster. Biten d skiftas in i den minst signifikanta positionen.
d>>Opr	"Opr" skiftas höger. Biten d skiftas in i den mest signifikanta positionen.
Opr'	Bitvis komplementering av operanden "Opr"

Fält där vi utelämnar styrsignal 0 eller 1, ska betraktas som signalen 0.

För att vara så tydliga som möjligt kan vi alltså välja om vi vill skriva ut nollan eller ej.

xx betyder "don't care".



Ett *program* för den enkla datavägen är helt enkelt en beskrivning av hur styrsignalerna ska aktiveras i någon bestämd sekvens. Styrsignalsekvensen måste följaktligen utformas speciellt för varje enskild operation man vill att processorn ska kunna utföra. Följande generella "blankett" kan användas för att "programmera" den manuella styrenheten i en följd av steg, dvs. en sekvens, för en operation.

RTN	steg	Source	OE _S	OE _A	OE _B	OE _R	LD _A	LD _B	LD _T	LD _R	C _{in}	f ₃	f ₂	f ₁	f ₀
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	10														
	11														

För varje steg anges styrsignaler för registeröverföringar (OE och LD), det finns möjlighet att ange konstanter i operationer och koppla in dessa i datavägen via *Source*-fältet. Det har också tillkommit styrsignaler samt en "carry in"-signal för ALU:n.

Vi illustrerar hur blanketten fylls i med att konstruera två olika styrsignalsekvenser för operationen:

0 → A

dvs. nollställ innehållet i register A.

Metod 1: använd *Source*-fältet, vars register initieras till 0:

RTN	steg	Source	OE _S	OE _A	OE _B	OE _R	LD _A	LD _B	LD _T	LD _R	C _{in}	f ₃	f ₂	f ₁	f ₀
0 → A	1	00	1	0	0	0	1	0	0	0	0	0	0	0	0

Metod 2: använd ALU'ns funktion för bitvis nollställning:

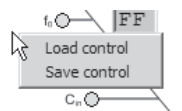
RTN	steg	Source	OE _S	OE _A	OE _B	OE _R	LD _A	LD _B	LD _T	LD _R	C _{in}	f ₃	f ₂	f ₁	f ₀
0 → R	1	xx								1	0	0	0	0	0
R → A	2	xx					1	1							

Av skäl som vi ska återkomma till bör vi *alltid* välja lösningar som utnyttjar ALU:n snarare än *Source*-fältet där sådana lösningar är möjliga.

Under resten av detta kapitel kan du nu självständigt konstruera styrsignalsekvenser för en rad olika operationer som kan utföras på dataväg/ALU med denna enkla manuella styrenhet. Välj Datapath | Datapath with ALU.

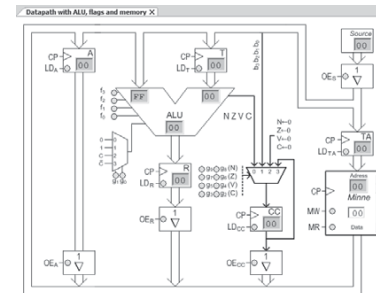
Du kan spara en styrsignalsekvens genom att högerklicka, välj Save control, och ange ett filnamn.

Återställ en tidigare sparad styrsignalsekvens genom att högerklicka, välja Load control, och ange dess filnamn.

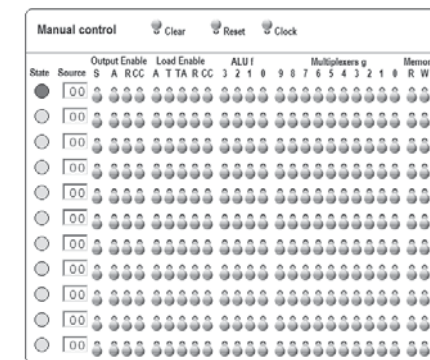


12 Dataväg med flaggregister och minne

Vi har hittills bara kunnat utföra operationer på ett fåtal variabler lagrade i datavägens arbetsregister A eller B. Nu utökar vi datavägen med en minnesmodul som utökar kapaciteten till operationer med en mängd olika variabler som då är lagrade i minnet, det är nu tillräckligt med ett arbetsregister och vi tar därför bort register B.

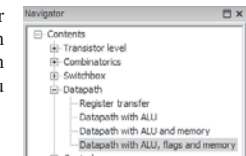


För att kunna adressera minnet har vi infört det speciella registret TA (*temporary address*). Vi har också lagt till logik för att kunna välja C_{in}-funktionen till ALU:n med hjälp av styrenheten. Styrsignalerna g₀ och g₁ tillkommer för detta ändamål. Ett nytt register CC (*condition codes*) tillsammans med en väljare och ytterligare styrsignaler g₂ t.o.m. g₉ används för att samla ihop ALU:ns flaggor.



Den manuella styrenheten har kompletterats med strömställare för de nya styrsignalerna. Vi använder också en utökad "blankett", med följande kolumner, vid programmering av den nya styrenheten.

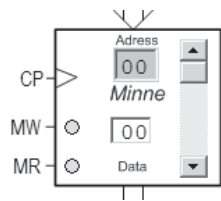
RTN	steg	Source	OE _S	OE _A	OE _R	OE _{CC}	LD _A	LD _T	LD _R	LD _{CC}	f ₃	f ₂	f ₁	f ₀	g ₉	g ₈	g ₇	g ₆	g ₅	g ₄	g ₃	g ₂	g ₁	g ₀	MR	MW



Exempel på RTN-symboler, fullständig tabell finns i appendix	
N _r	Konstanten N uttryckt i talbasen r.
M(N _r)	Minnesinnehåll på adress N _r .
M[N _r]	Indirektion: = M(M(N _r))
→	Kopiering
Följande symboler är beteckningar som reserverats för register	
A	Register A
T	Register T
R	Register R
TA	Register TA
CC	Register CC
Operatörer	
+	Addition
-	Subtraktion
^	Logiskt "OCH" (AND)
∨	Logiskt "ELLER" (OR)
⊕	Logiskt "EXKLUSIVT ELLER" (XOR)
Opr<<d	"Opr" skiftas vänster. Biten d skiftas in i den minst signifikanta positionen.
d>>Opr	"Opr" skiftas höger. Biten d skiftas in i den mest signifikanta positionen.
Opr'	Bitvis komplementering av "Opr"

Tabellen med RTN-koder har kompletterats med symboler för registren TA och CC.

12.1 Simulators minnesmodul



Simulatorns minnesmodul har plats för 256 st. 8-bitars dataord. Styrsignalerna MW (*Memory Write*) och MR (*Memory Read*) används för att skriva till, respektive läsa från minnet. MW-signalen är synkron, dvs. skrivning till minnet sker vid klockpuls om MW är aktiv. MR-signalen är asynkron, dvs. då MR aktiveras kopplas en minnescell omedelbart direkt till bussen. Detta kan jämföras med registrens OE-signaler.

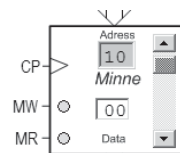
För att underlätta användningen av simulatorns minnesmodul kan man skriva in data i minnet på ett förenklat sätt. Minnet har två små "fönster". "Adress"-fönstret anger adressen till den minnescell som för tillfället adresseras, minnescellens innehåll visas i "Data"-fönstret.

Uppgift 12.1

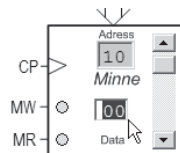
Placera värdet 15_{16} i minnescell på adress 10_{16} . Med RTN skriver vi denna operation som:

$$15_{16} \rightarrow M(10_{16})$$

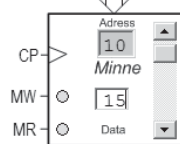
Använd "rullningslisten" hos minnesmodulen för att bläddra fram adressen.



"Dubbelklicka" i "Data"-fönstret.



Skriv in det nya värdet.



Metoden är lämplig att använda då man snabbt vill modifiera minnesinnehållet för att kunna testa någon programmerad minnesoperation.

12.2 Läscopykeln

Minnet kan adresseras via datavägens TA-register.

En läscopykel går till på följande sätt:

1. Minnesadress placeras i TA
2. MR-signalen aktiveras.

Om den aktuella adressen inte redan finns i TA-registret så kräver läscopykeln alltså ett extra steg än om vi läser data från ett register i datavägen.

13.6 Konstruktion av sekvensierare för FLISP

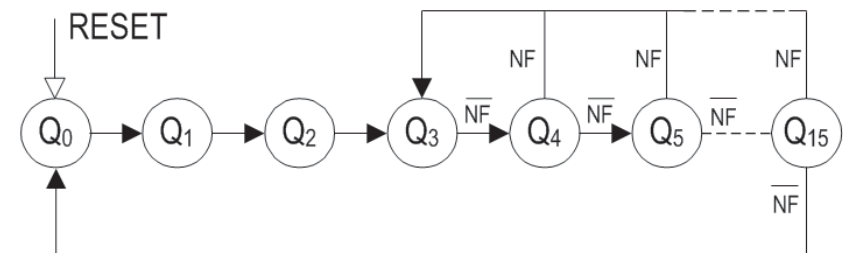
I detta avsnitt ska en räknare med två styrsignaler konstrueras. Maskinen konstrueras med D-vippor och enkla grindar. Vi kommer att använda maskinen som *sekvensierare i styrenheten* i nästa kapitel.

Uppgift 13.7

I denna uppgift ska du konstruera en något mer komplex tillståndsmaskin. Maskinen har 16 olika tillstånd.

- Räknesekvensen är 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0...
- Den asynkrona signalen RESET = 1 sätter maskinen i tillstånd 0 oavsett vilket tillstånd maskinen befinner sig i.
- Den synkrona signalen NF=1 sätter maskinen i tillstånd 3 om maskinen befinner sig i något av tillstånden Q_4 t.o.m Q_{15} .

Följande tillståndsgraf beskriver då maskinen:



Eftersom vi här har fem oberoende variabler men våra Karnaughdiagram bara låter oss hantera fyra variabler åt gången delar vi upp konstruktionen i två steg.

Under det första steget bestämmer vi konstruktionen för NF=0, dvs en autonom räknare med den angivna räknesekvensen. Detta ger oss fyra Karnaughdiagram för att bilda vippornas insignaler.

I nästa steg bestämmer vi konstruktionen för NF=1, vilket ger oss fyra nya Karnaughdiagram med ytterligare insignaler som ska adderas till de tidigare.

Därmed har vi bestämt maskinens synkrona beteende.

Börja med att bestämma d-funktionerna i följande följande tabell.

Nuvarande tillstånd									Nästa tillstånd			
NF	q ₃	q ₂	q ₁	q ₀	d ₃	d ₂	d ₁	d ₀	q ₃ ⁺	q ₂ ⁺	q ₁ ⁺	q ₀ ⁺
0	0	0	0	0					0	0	0	1
0	0	0	0	1					0	0	1	0
0	0	0	1	0					0	0	1	1
0	0	0	1	1					0	1	0	0
0	0	1	0	0					0	1	0	1
0	0	1	0	1					0	1	1	0
0	0	1	1	0					0	1	1	1
0	0	1	1	1					1	0	0	0
0	1	0	0	0					1	0	0	1
0	1	0	0	1					1	0	1	0
0	1	0	1	0					1	1	0	0
0	1	1	0	0					1	1	0	1
0	1	1	0	1					1	1	1	0
0	1	1	1	0					1	1	1	1
0	1	1	1	1					0	0	0	0
1	0	0	0	0					0	0	0	1
1	0	0	0	1					0	0	1	0
1	0	0	1	0					0	0	1	1
1	0	0	1	1					0	1	0	0
1	0	1	0	0					0	0	1	1
1	0	1	0	1					0	0	1	1
1	0	1	1	0					0	0	1	1
1	0	1	1	1					0	0	1	1
1	1	0	0	0					0	0	1	1
1	1	0	0	1					0	0	1	1
1	1	0	1	0					0	0	1	1
1	1	0	1	1					0	0	1	1
1	1	1	0	0					0	0	1	1
1	1	1	0	1					0	0	1	1
1	1	1	1	0					0	0	1	1
1	1	1	1	1					0	0	1	1

Fyll därefter i Karnaughdiagrammen på nästa sida.

d₃ (NF=0) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₂ (NF=0) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₁ (NF=0) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₀ (NF=0) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₃ (NF=1) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₂ (NF=1) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₁ (NF=1) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

d₀ (NF=1) q₁ q₀

	00	01	11	10
q ₃ q ₂ 00				
01				
11				
10				

Ange de minimerade funktionerna på boolesk form och, observera:

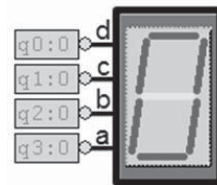
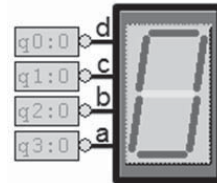
NF=0		NF=1
$d_3 =$	+	
$d_2 =$	+	
$d_1 =$	+	
$d_0 =$	+	

förenkla tills endast AND- och XOR-grindar används i uttrycken

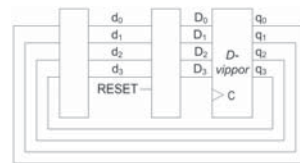
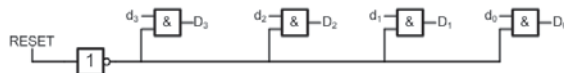
NF=0		NF=1
$d_3 =$	+	
$d_2 =$	+	
$d_1 =$	+	
$d_0 =$	+	

Koppla upp räknaren i kopplingsboxen.

- Använd komponenten Hex-sifferindikator för att kontrollera räknesekvensen.
- Spara kopplingen, den ska strax utökas.



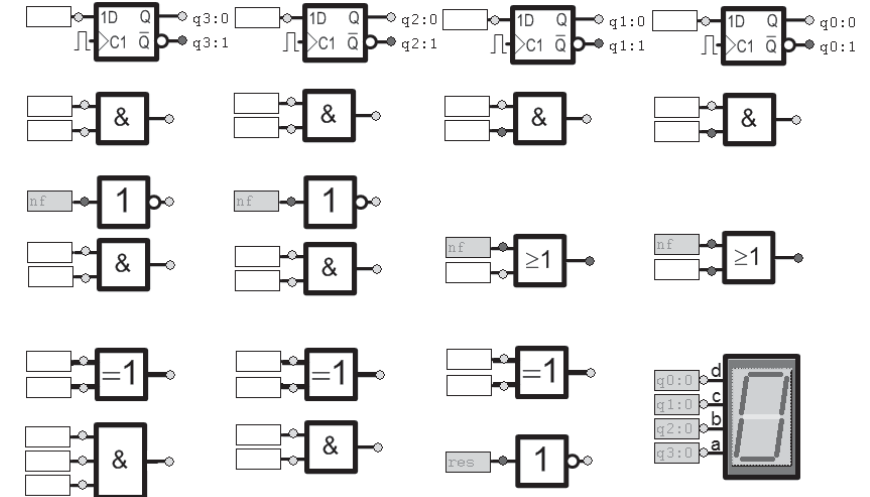
Vi tar nu hand om den asynkrona signalen RESET genom att "skjuta in" ett kombinatoriskt nät där såväl våra d-funktioner som RESET-signalen ingår. Utsignalerna D_3 , D_2 , D_1 och D_0 från detta nät kopplas nu till vippornas ingångar i stället för d_3 , d_2 , d_1 och d_0 från det första konstruktionssteget.



Följande skiss visar en koppling för hela tillståndsmaskinen. Observera att uttrycken för d-signalerna här har förenklats för användning av XOR-grindar.

Jämför med dina egna uttryck från den inledande uppgiften.

Färdigställ skissen genom att skriv in- och utsignalsnamn i anslutning till de komponenter som saknar dessa.



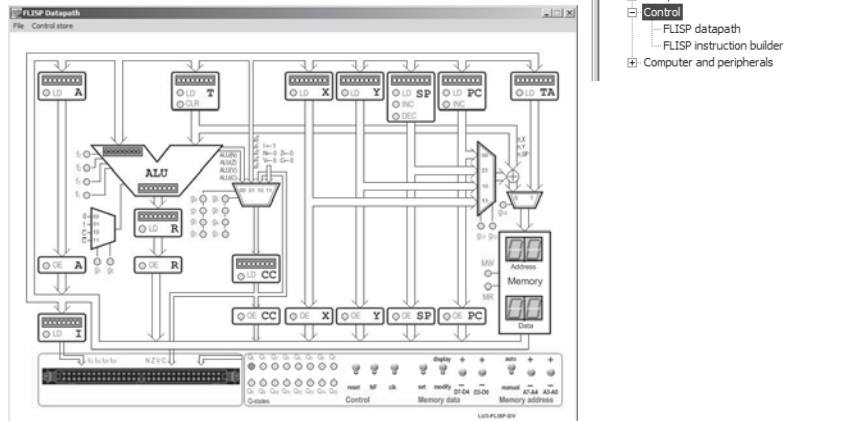
Koppla slutligen upp maskinen i kopplingsboxen och kontrollera funktionen med avseende på räknesekvens och insignalerna.

Spara filen under namnet `FlispStateMachine.tb`.

14 Den automatiska styrenheten

I detta kapitel arbetar vi med FLIS-processorn och dess automatiska styrenhet. Simulatoren omfattar två fönster, FLISP:s dataväg och ett fönster för att skapa instruktioner för FLISP.

Välj: Control | FLISP datapath.



Strömställarnas funktioner:

Control:

- reset – asynkron, återställ sekvensieraren till Q_0
- NF – utför upprepade klocksignaler tills nästa tillstånd är Q_3 , uttryckt på ett annat sätt, utför en hel instruktion
- clk – ge en klocksignal till datavägen.

Memory data:

- display/modify – i läge display, visas innehållet på den adress som anges av indikator Address, på indikator Data. I läge modify kan innehållet på adressen ändras med hjälp av vippströmställarna D7-D4 och D3-D0
- D7-D4, D3-D0 används för att ställa in Data i modify-läge
- set – innehållet som visas på Data skrivs till adressen som anges av Address.

Memory address:

- auto/manual, i läge auto visar Address den adress som bildats från multiplexer med styrsignal g_{14} , i läge manual kan adressen ändras med hjälp av vippströmställarna A7-A4 och A3-A0
- A7-A4, A3-A0 används för att ställa in Address i manual-läge.

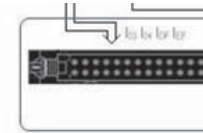
Menyer:

- File | Load, används för att ladda innehåll till primärminnet
- Control store | FLISP control enabled, aktivera styrenhetens FLISP-instruktioner.

Datavägens menyval:

File | Load – ladda en fil med *fmem* (*flisp memory*) format till datavägen. Formatet är textbaserat och en fil kan enkelt skapas med en textredigerare. Formatet utgörs av direktiv till datavägen. Varje direktiv inleds på ny rad med tecknet '#', rader som inleds med annat tecken tolkas som kommentarer. Följande direktiv kan användas:

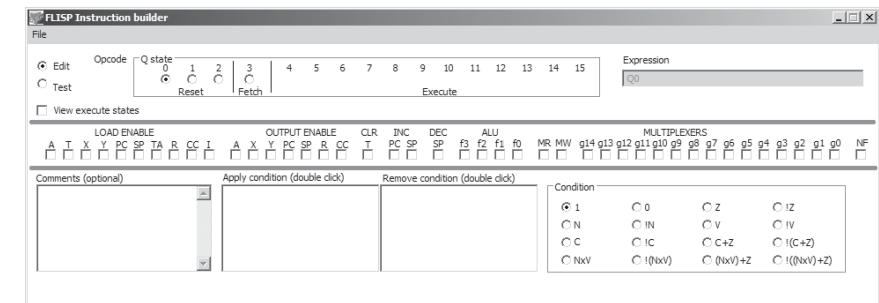
#ClearAllMemory	Nollställ primärminne
#ClearAllRegisters	Nollställ alla register
#SetMemory ADR=VÄRDE	Initiera minnescell, ADR och VÄRDE anges på hexadecimal form
#SetRegister REG=VÄRDE	REG kan vara något av: A,T,X,Y,PC,SP,TA,R,CC eller I. VÄRDE anges på hexadecimal form



Control store | FLISP control enabled – Datavägen kan konfigureras att använda en komplett styrenhet för FLISP, nya instruktioner måste då ha någon av de odefinierade operationskoderna 03,04,DF eller EF. I normalfallet är denna funktion deaktiverad och vi måste då tillhandahålla alla styrsignalsekvenser.

Instruction builder används för att skapa styrsignalsekvenser för instruktioner som kan utföras av FLIS-processorns styrenhet.

Välj: Control | FLISP instruction builder.



Instruction builders menyval

Clear – all styrsignalinformation raderas

Load – styrsignalinformation laddas från fil med *fcs*-format.

Save – styrsignalinformation sparas som *fcs*-format.

Save as – spara styrsignalinformation till en fil som *fcs*-format.

Export – spara styrsignalsinformation i form av C-kod.

Exit – stäng Instruction Builder.

Filer för datavägen har ändelsen *fmem* (*FLISP memory*).

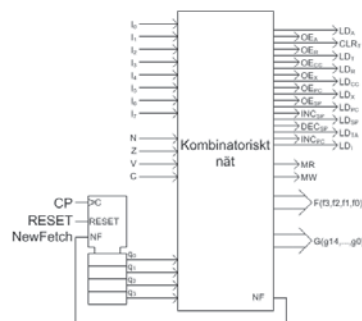
Filer för Instruction builder har filnamnsändelsen *fcs* (*FLISP control state*).

14.1 Styrenheten i FLISP

Instruction builder används för att redigera och testa instruktioner. För instruktionsexekvering bildas styrsignalerna som en kombination av tillstånd (betecknas Q) och operationskod (betecknas Opcode). För att implementera villkorliga instruktioner krävs också att statusflaggorna från CC (N,Z,V,C) finns tillgängliga i styrenheten.

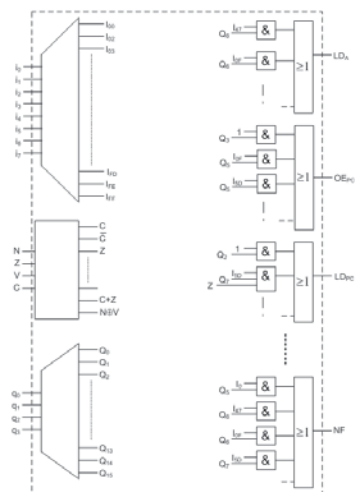
I styrenheten har de Booleska funktioner som krävs för de villkorliga instruktionerna implementerats med kombinatoriska nät. Utsignaler från dessa nät finns tillgängliga i sektionen Condition.

Vi känner igen styrsignalerna som nu ska genereras automatiskt. Det har dock tillkommit ytterligare en signal, NF (New Fetch) som anger att en ny instruktion ska hämtas i minnet.



Figur 14.1 Översikt av automatisk styrenhet

Det kombinatoriska nätet utformas för att generera de *summerter* som används för att aktivera de olika styrsignalerna för olika kombinationer av tillstånd och operationskod. En summerterm är alltså en produkt av en operationskod hämtad från instruktionsregistret och ett specifikt tillstånd hämtad från räknaren.



Figur 14.2 Illustration av styrenhetens kombinatoriska nät

Observera

Förväxla inte styrsignalen NF med datavägens funktion för att utföra en hel instruktion.

Styrenhetens funktioner kan indelas i tre olika faser:

Återställningsfas (RESET): FLISP återställs genom att en startadress läses från RESET-vektorn (adress FF₁₆) i minnet och placeras i programräknaren PC.

Hämtfas (FETCH): Innehållet på adress PC läses och placeras i instruktionsregistret I, PC ökas med 1.

Exekveringsfas (EXECUTE): varje instruktion har en unik styrsignalsekvens som ska genereras under respektive exekveringsfas. Då exekveringsfasen utförts ska PC ha uppdaterats så att den innehåller adressen till nästa instruktion i minnet.

Observera att styrsignalsekvenserna för RESET och FETCH är oberoende av operationskoden.

Observera också att exekveringsfasen alltid måste avslutas med att generera NF för att starta nästa instruktionshämtning. Om tillståndsmaskinen klockas ur tillståndet Q₁₅ utan att NF-signalen genererats kommer FLISP att återstartas (RESET).



Figur 14.3 Tillståndsgraf för styrenheten

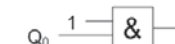
14.1.1 Styrsignalsekvens för RESET-fasen

Återställningsfasen utförs under tre steg och representeras av de tre tillstånden Q₀, Q₁ och Q₂.

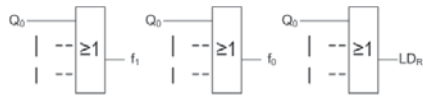
Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler	Kommentarer
Q ₀	(Q ₀ •1)	(FF) ₁₆ →R	f ₁ =1; f ₀ =1; LD _R =1	ALU-funktionen väljs så att talet FF ₁₆ finns på ALU:ns utgång, dvs. funktionskod 3, ALU-funktion = (0011) ₂ . Laddningängen på R-registret ettsätts så att utvärdet från ALU:n FF ₁₆ laddas i R-registret vid nästa klockpuls.
Q ₁	(Q ₁ •1)	R→TA	OE _R =1; LD _{TA} =1;	Talet FF ₁₆ i R-registret kopplas ut på bussen. Talet FF ₁₆ på bussen laddas i temporäradressregistret vid nästa klockpuls.
Q ₂	(Q ₂ •1)	M→PC	MR=1; g ₁₄ =1; LD _{PC} =1;	Minnesinnehållet på adress FF ₁₆ läses genom att minnet aktiveras för läsning. Temporäradressregistret adresserar minnet. Det dataord som läses placeras i PC vid nästa klockpuls.

Vi kan nu börja skapa summerter för styrsignalerna genom att först identifiera de signaler som ska aktiveras vid tillståndssignal Q₀. Av tabellen ovan framgår att dessa är f₁, f₀ och LD_R.

Dessa signaler ska aktiveras oavsett vad som finns i instruktionsregistret. AND-villkoret blir därför här:

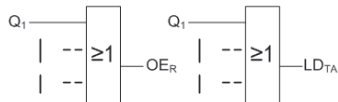


Denna summaterm skrivs alltså $Q_0 \bullet 1$, vilket är samma sak som Q_0 . Vi påför därför Q_0 -signalen på ELLER-grindarna för de tre styrsignalerna:



Figur 14.4 Bidrag till f_1 , f_0 och LD_R från RESET-fasen

På samma sätt ska styrsignalerna OE_R och LD_{TA} aktiveras i tillstånd Q_1 :



Figur 14.5 Bidrag till OE_R och LD_{TA} från RESET-fasen

och slutligen styrsignalerna MR , g_{14} och LD_{PC} för tillstånd Q_2 .



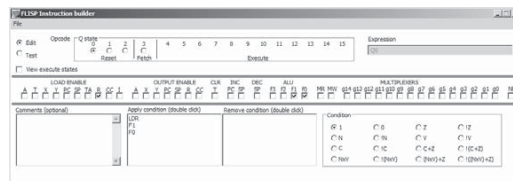
Figur 14.6 Bidrag till MR , g_{14} och LD_{PC} från RESET-fasen

Vi övergår nu till simulatören.

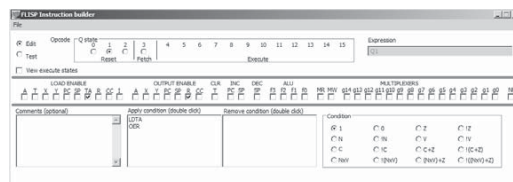
Uppgift 14.1

Skapa styrsignaler för RESET-fasen i den automatiska styrenheten.

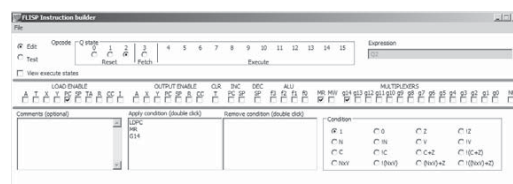
1. Aktivera styrsignalerna LD_R , f_0 och f_1 för tillståndsterm Q_0



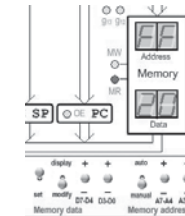
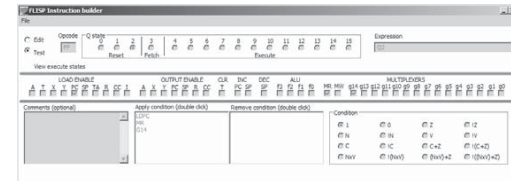
2. Aktivera styrsignalerna OE_R och LD_{TA} för tillståndsterm Q_1



3. Aktivera styrsignalerna MR , g_{14} och LD_{PC} för tillståndsterm Q_2



Det är nu dags att prova återställningssekvensen. Växla *Instruction builder* till *Test*-funktion.



Lägg in värdet 20_{16} i RESET-vektorn (adress FF_{16}).



Återställ FLISP genom att klicka på datavägens reset-omkopplare. Tillståndindikatorn (Q-states) indikerar nu tillståndet Q_0 . Observera datavägen, speciellt de signaler som ska vara aktiva i tillstånd Q_0 .

Ge styrenheten en klockpuls genom att klicka på omkopplaren CP. Kontrollera de aktiva signalerna i datavägen, nu för tillståndet Q_1 .

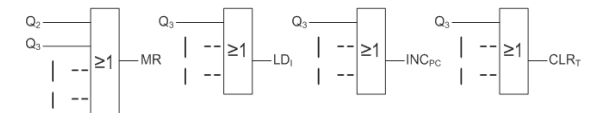
Ge ytterligare två klockpulser så att RESET-fasen slutförs och FETCH-fasen inleds. Tillståndindikatorn visar då tillstånd Q_3 . Kontrollera att adressen 20_{16} nu finns i PC.

14.1.2 Styrsignalsekvens för FETCH-fasen

Instruktionshämtningen sker i tillståndet Q_3 . Här förutsätts att PC innehåller adressen till den instruktion som ska hämtas från minnet.

Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler	Kommentarer
Q_3	$(Q_3 \bullet 1)$	$M(PC) \rightarrow I$; $0 \rightarrow T$;	$MR=1$; $LD=1$; $INC_{PC}=1$; $CLR_T=1$;	Adressen för nästa instruktions operationskod, dvs. PC, kopplas till minnets adressbuss. Läs operationskoden från minnet och placera i instruktionsregistret I. Adressen som finns i PC ökas med ett. Register för index vid adressberäkningar nollställs.

Dessa styrsignalers bidrag till AND/OR-nätet i styrenheten visas i följande figur.

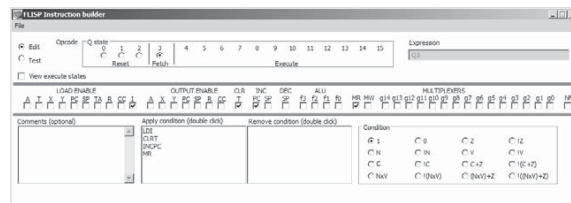


Figur 14.7 Bidrag till MR , LD , INC_{PC} och CLR_T från FETCH-fasen

En anmärkning kan vara på sin plats angående CLR_T -signalen. Eftersom T-registret används för offset vid adressberäkningar för vissa register (X, Y och SP) är det tillräckligt att nollställa detta i varje FETCH-fas. Detta gynnar de instruktioner som gör sådana adressberäkningar redan i exekveringsfasens första tillstånd.

Uppgift 14.2

Skapa styrsignaler för FETCH-fasen i den automatiska styrenheten genom att lägga in styrsignalerna MR, LD_i, INC_{PC} och CLR_T för summatermen Q₃.



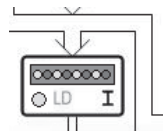
Du ska nu prova den sammanhängande RESET/FETCH-sekvensen, dvs. återstart och den första instruktionshämtningen.

Lägg in värdet 20₁₆ i RESET-vektorn (adress FF₁₆).

Lägg in värdet 55₁₆ på adress 20₁₆, detta blir "operationskoden" för den instruktion som ska hämtas.

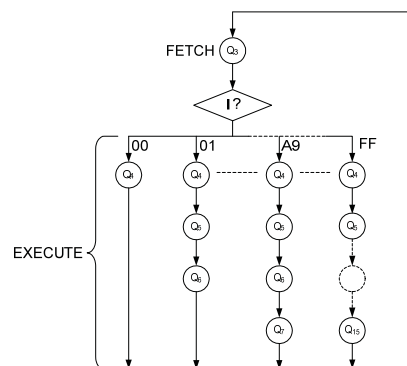
Återställ den automatiska styrenheten (klicka på RESET) och klocka fram tillståndet, då du når tillstånd Q₄, dvs. FETCH-fasen har utförts, ska värdet 55₁₆ finnas i instruktionsregistret (register I).

Välj File | Save as i *Instruction Builder* och spara styrsignalerna i filen "flisp_reset.fcs".



14.1.3 Exekveringsfasen

Med övergången till tillstånd Q₄ inleds exekveringsfasen. Eftersom varje instruktion (operationskod) har en unik styrsignalsekvens kommer nu varje summaterm att utgöras av ett AND-villkor (tillstånd och operationskod). Exekveringsfaserna har också olika längd (antal tillstånd) beroende på komplexiteten hos instruktionerna. I Figur 14.8 illustreras antalet tillstånd hos styrsignalsekvenserna för såväl den kortaste instruktionen (NOP, operationskod 00) som den längsta möjliga styrsignalsekvensen (ogiltig operationskod, FF).



Figur 14.8 Lagrade programets princip (FETCH/EXECUTE)

Detaljerna hos varje FLISP-instruktion finns i instruktionslistan. Instruktionen *No Operation* specificeras exempelvis på följande sätt:

NOP No operation

RTN	
Flaggor	Påverkas ej
Beskrivning	Instruktionen utför ingenting

Detaljer:

Instruktion	Adressering	Operation	Flaggor
NOP	metod OP # ~		N Z V C
NOP	Inherent 00 1 2	No operation	- - - -

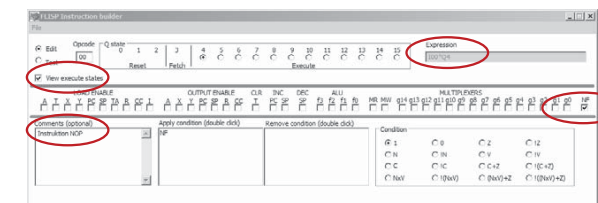
Instruktionen är den enklast tänkbara, "utför ingenting". Ur kolumnen Adressering läser vi ut att operationskoden (OP) är 00, att instruktionen upptar 1 byte i minnet (#) och att den tar två cykler(~) att utföra. I exekveringstiden ingår hämtfasen, varför antalet tillstånd i exekveringsfasen alltid är ett mindre än vad som anges här.

Uppgift 14.3

Implementera instruktionen NOP. Avkodningen av instruktionsregistret ger att signalen I_{00} är aktiv endast för denna operationskod, samtidigt som exekveringsfasens tillstånd är Q₄. Detta ger oss summatermen för den signal (de signaler) som är aktiva under respektive tillstånd. De aktiva styrsignalerna anges i sin tur som *styrsignal*=1.

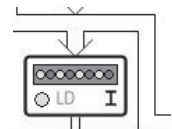
Tillstånd	Summa-term	RTN-beskrivning	Styrsignaler	Kommentarer
Q ₄	(Q ₄ • I ₀₀)		NF=1;	Instruktionen utför ingenting.

1. Aktivera View execute states, skriv in operationskoden 00 (avsluta med Enter), och kontrollera att rätt summaterm (I00*Q4) visas i fönstret Expression.
2. Aktivera instruktionens styrsignal (NF) för summaterm.
3. Skriv eventuellt en kommentar i avsett fält.



Testa nu instruktionen NOP:

1. Lägg in värdet 20₁₆ i RESET-vektorn (adress FF₁₆).
2. Lägg in värdet 00₁₆ på adress 20₁₆, dvs. operationskoden för instruktionen NOP.
3. Återställ den automatiska styrenheten (klicka på RESET) och klocka fram tillståndet, då du når tillstånd Q₄, dvs. FETCH-fasen har utförts, ska värdet 00₁₆ finnas i instruktionsregistret (register I).
4. Ge nu ytterligare en klockpuls för att utföra NOP-instruktionens exekveringsfas. Kontrollera att FLISP då återgår till FETCH-fasen. Instruktionen är därefter implementerad.
5. Välj File | Save as i *Instruction Builder* och spara styrsignalerna i filen "flisp_nop.fcs".



14.2 Implementering av instruktioner

Resten av detta kapitel ägnas åt en översikt av flertalet av de instruktioner som definierats för FLISP. Instruktionerna delas in i grupper med avseende på operationer. Observera att vi här nöjer oss med att exemplifiera med några utvalda adresseringssätt och instruktionsbeskrivningarna är därför inte fullständiga. För detaljerade beskrivningar hänvisas till FLISP-handboken.

14.2.1 Läs från minne

Data kan läsas från minnet med en LD-instruktion ("load"). Alternativt kan andra registerinnehåll, ev. med någon offset, läsas med en LEA-instruktion ("load effective address"). LD-instruktionen finns för samtliga register, dvs.

LDA, LDX, LDY, LDSP

LEA-instruktion finns bara för adressregistren:

LEAX, LEAY, LEASP

Följande utdrag ur FLISP-handboken ger detaljer om instruktionen LDA för tre olika adresseringssätt:

LD Load register

RTN	M(EA)→R
Flaggor	N: Ettställs om resultatets teckenbit (bit 7) får värdet 1. Z: Ettställs om samtliga åtta bitar i resultatet blir noll. V: Nollställs. C: Påverkas ej.
Beskrivning	Laddar dataord från minnet till angivet register R (A,X,Y eller SP)

Detaljer:

Instruktion	Adressering	Operation	Flaggor
LD	metod	OP # ~	N Z V C
LDA #Data	Immediate	F0 2 2 Data → A	Δ Δ 0 -
LDA Adr	Absolute	F1 2 3 M(Adr) → A	
LDA n, SP	Indexed	F2 2 3 M(n+SP) → A	

LDA #Data

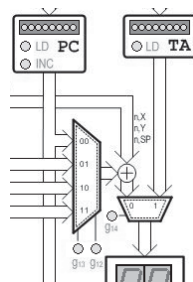
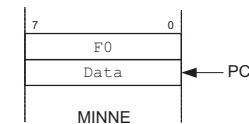
Av detaljinformationen ser vi att instruktionen, som upptar två bytes i minnet (#), tar totalt två klockcykler (~) att exekveras. Eftersom denna siffra även omfattar hämtfasen innebär detta att exekveringsfasen ska utföras under en klockcykel.

Då exekveringsfasen inleds innehåller PC adressen till ordet efter operationskoden, i detta fall data som ska läsas in till register A. RTN-beskrivningen för att läsa in data blir då:

$M(PC) \rightarrow A$

Vi kopplar därför PC till MA, aktiverar MR och LD_A. Se figuren i marginalen: eftersom g_{13} och g_{12} båda är 0, väljs PC av "1-av-4"-väljaren. Därefter passerar PC adderarsteget, dock utan att innehållet i T adderas, detta utförs bara om någon av g_{13} eller g_{12} är 1, dvs för register X,Y och SP. Slutligen väljs PC från utgången från adderarsteget eftersom g_{14} , som styr "1 av 2"-väljaren, är 0.

Samtidigt uppdateras PC för att peka på nästa instruktion:



$PC+1 \rightarrow PC$

För flaggsättningen observerar vi att data som ska påverka CC också finns på ALU:ns D-ingång och genom att använda operationen:

$D+C_{in} \rightarrow U; C_{in}=0$

kan vi utnyttja ALU:ns flaggsättning för flaggorna N och Z. Vi använder sedan styrsignaler (g-) för att nollställa V och låta C vara opåverkad. Därmed är styrsignalsekvensen för denna variant klar.

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q4	(Q4•I=0)	M(PC)→A; PC+1→PC; D+0→U; 0→V; CC(C)→C;	LD _A ; MR; INC _{PC} ; f ₅ ; f ₆ ; g ₅ ; g ₃ ; g ₂ ; LD _{CC} ; NF;	Data från minnet till register A Uppdatera PC ALU-funktion 9 för flaggsättning N och Z nollställ V ingen påverkan C uppdatera CC ny instruktion

LDA Adr

Av detaljinformationen ser vi att exekveringsfasen av denna variant av instruktionen, tar två klockcykler (~) för att exekveras. Den extra cykeln kommer av att vi här måste göra två läsningar i minnet. Först ska adressen (Adr) läsas från instruktionen, därefter ska data läsas från denna adress.

Adressen från minnet läses till adressregister TA och PC uppdateras:

$M(PC) \rightarrow TA; PC+1 \rightarrow PC$

under nästa cykel kopplas TA till MA genom att g_{14} sätts till 1, varvid MR och LD_A-signalerna aktiveras.

$M(TA) \rightarrow A$

I följande tabell anges styrsignalerna för att läsa data från en adress till register A. Observera att tabellen inte beskriver den fullständiga instruktionsvarianten, jämför med LDA #Data.

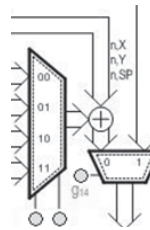
Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q4	(Q4•I=1)	M(PC)→TA; PC+1→PC;	LD _{TA} ; MR; INC _{PC} ;	Adress från minnet till temporär adress Uppdatera PC
Q5	(Q5•I=1)	M(TA)→A; (etc.)	LD _A ; g ₁₄ ; MR; (etc.)	Data från "Adr" till A

LDA n, SP

Även indexerade adresseringssätt kräver en extra läsning av operanden (Data). Här måste vi dock först genomföra en beräkning av den effektiva adressen, i detta fall $n+SP$.

Datavägen i FLISP har förberetts för en sådan adressberäkning genom att innehållet i temporärregister T adderas, som offset, till innehållet i något adressregister. Observera att offset adderas endast i 3 av de 8 möjliga sätten att utföra adressberäkningarna. Följande tabell beskriver funktionen för väljarsignalerna g_{14} , g_{13} och g_{12} .

g ₁₄	g ₁₃	g ₁₂	Register till adressbuss:	RTN
0	0	0	Register PC, ingen offset	M(PC)
0	0	1	Register SP ("bas") och register T ("offset")	M(T+SP)
0	1	0	Register Y ("bas") och register T ("offset")	M(T+Y)
0	1	1	Register X ("bas") och register T ("offset")	M(T+X)
1	0	0	Adressberäkningsregister TA, ingen offset	M(TA)
1	0	1	Adressberäkningsregister TA, ingen offset	M(TA)
1	1	0	Adressberäkningsregister TA, ingen offset	M(TA)
1	1	1	Adressberäkningsregister TA, ingen offset	M(TA)



Offseten (n) läses från minnet till register T och PC uppdateras:

$M(PC) \rightarrow T$; $PC+1 \rightarrow PC$

under nästa cykel kopplas adressen $n+SP$ till MA genom att g_{12} sätts till 1, MR och LD_A-signalerna aktiveras.

$M(T+SP) \rightarrow A$

Styrsignaler för inläsning av data från adress "n+SP" till register A visas i följande tabell:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •I _{F2})	$M(PC) \rightarrow T$; $PC+1 \rightarrow PC$;	LD _T ; MR; INC _{PC} ;	Offset från minnet till T Uppdatera PC
Q ₅	(Q ₅ •I _{F2})	$M(T+SP) \rightarrow A$; (etc.)	LD _A ; g ₁₂ ; MR; (etc.)	Data från "n+SP" till A

Vi ser här att styrsignalsekvenser för instruktioner med de indexerade adresseringssätten n_R ($R=X, Y$ eller SP) skiljs åt endast genom användningen av g_{13} och g_{12} .

Uppgift 14.4

Implementering och test av LDA-instruktioner.

1. Radera först all styrsignalinformation (File|Clear) och utgå från din sparade fil `flisp_nop.fcs`, dvs. ladda filen (File|Load) till *Instruction Builder*. Nu finns enbart styrsignaler för RESET och FETCH faserna och instruktionen NOP i styrenheten.
2. Implementera nu instruktionerna LDA #Data, LDA Adr och LDA n, SP enligt tidigare anvisningar. Spara styrsignalinformationen (File|Save as) med namnet `flisp_ins.fcs`.
3. Skapa en fil `test_load.fmem`, för test av de implementerade instruktionerna (se marginalen). Vi har placerat en instruktionssekvens med start på adress 20_{16} i minnet; detaljerna framgår av följande:

Adress	Maskinkod	Assemblerkod	RTN
20		LDA #7	$7 \rightarrow A$
21			
22		LDA 10_{16}	$M(10_{16}) \rightarrow A$
23			
24		LDA $1, SP$	$M(1+SP) \rightarrow A$
25			
26			

Efterhand som du implementerar och testar nya FLISP-instruktioner lägger du till dessa i filen:

flisp_ins.fcs.

Det är däremot lämpligt att skapa separata testfiler:

test_XXX.fmem
för de olika instruktionerna.

test_load.fmem

```
#ClearAllMemory
#ClearAllRegisters
Operationskoder och
operandinformation läggs i
minnet:
#SetMemory 20=F0
#SetMemory 21=07
#SetMemory 22=F1
#SetMemory 23=10
#SetMemory 24=F2
#SetMemory 25=01
Data läggs på plats och register
SP ges initialvärde
#SetMemory 10=00
#SetRegister SP=0F
#SetMemory 10=81
RESET-vektor
#SetMemory FF=20
```

Testa implementeringen av instruktionerna genom att läsa in testfilen, gör RESET och ge klockpulser tills instruktionssekvensen utförts. Kontrollera mellan varje instruktion att denna utförs korrekt och ange testresultaten i följande tabell; kontrollera även flaggsättningen.

	A	N	Z	V	C
initialt					
LDA #Data					
LDA Adr					
LDA n, SP					

Då vi jämför LD-instruktioner med samma adresseringssätt ser vi att de är mycket lika: jämför exempelvis LDA, LDSP och LDX.

Instruktion LD	Adressering				Operation				Flaggor			
	metod	OP	#	~					N	Z	V	C
LDA #Data	Immediate	F0	2	2	Data → A				Δ	Δ	0	-
LDSP #Data	Immediate	92	2	2	Data → SP							
LDX #Data	Immediate	90	2	2	Data → X							

För styrsignalsekvensen innebär detta att det är enbart operationskoden och en enstaka styrsignal som skiljer dem åt:

LDSP #Data

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •I ₃₂)	M(PC)→SP; PC+1→PC; D+0→U; 0→V; CC(C)→C;	LD _{SP} ; MR; INC _{PC} ; f ₃ ; f ₀ ; g ₅ ; g ₃ ; g ₂ ; LD _{CC} ; NF;	Data från minnet till register SP Uppdatera PC ALU-funktion 9 för flaggsättning N och Z nollställ V ingen påverkan C uppdatera CC ny instruktion

Uppgift 14.5

Implementera först instruktionen:

LDSP #Data

Lägg till instruktionen i filen flisp_ins.fcs.

Jämför styrsignaltabellen för LDSP #Data, med följande styrsignaltabell för LDX #Data, komplettera tabellen med operationskod och aktiva styrsignaler.

LDX #Data:

Tillstånd	Summaterm	RTN-beskrivning	Aktiva (=1) Styrsignaler
Q ₄	(Q ₄ •)	M(PC)→X; PC+1→PC; Flags→CC;	f ₃ ; f ₀ ; g ₅ ; g ₃ ; g ₂ ; LD _{CC} ; NF;

Lägg även till instruktionen LDX #Data i filen flisp_ins.fcs. Skapa en lämplig fil för test av instruktionerna och kontrollera att de fungerar som de ska.

LEA-instruktionerna är i första hand avsedda för adressberäkningar. De tillåter att adresser kopieras mellan adressregistren X,Y och SP, ev. med addition/subtraktion av någon offset.

LEA Load effective address

RTN	EA → R ; R kan vara X,Y eller SP
Flaggor	Påverkas ej
Beskrivning	Laddar effektiva adressen i R. Används för att addera/subtrahera registerinnehåll.

Detaljer:

Instruktion LEA	Adressering				Operation	Flaggor			
	metod	OP	#	~		N	Z	V	C
LEAX n, X	Indexed	CC	2	4	X + n → X	-	-	-	-
LEAX n, SP	Indexed	DC	2	4	SP + n → X				
LEAY n, Y	Indexed	CD	2	4	Y + n → Y				
LEAY n, SP	Indexed	DD	2	4	SP + n → Y				
LEASP n, SP	Indexed	BE	2	4	SP + n → SP				
LEASP n, X	Indexed	CE	2	4	X + n → SP				
LEASP n, Y	Indexed	DE	2	4	Y + n → SP				

Låt *K* beteckna det register (X,Y eller SP) som ingår i operanden, *källregistret*, och låt *D* beteckna det register som är en del av instruktionsnamnet, *destinationsregistret*. Det är den *effektiva adressen* n+K som ska placeras i register *D* och vi kan därför *inte*, som förut, använda metoden (T+K), vilket ger en adress som alltid adresserar minnet. Vi måste därför göra själva adressberäkningen n+K med hjälp av ALU:n. Utförandefasen kräver tre cykler:

Offseten (n) från minnet läses till register T och PC uppdateras:

M(PC)→T; PC+1→PC

under nästa cykel kopplas register *K* till bussen och ALU:n utför addition:

T+K→R

Resultatet återförs till destinationsregistret:

R→D

Styrsignaler för instruktionen med de generella beteckningarna visas i följande tabell:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	M(PC)→T; PC+1→PC	LD _T ; MR; INC _{PC} ;	Offset från minnet till T Uppdatera PC
Q ₅	(Q ₅ •)	T+K→R	f ₃ ; f ₁ ; f ₀ ; OE _K ; LD _R	Bestäm effektiv adress "n+K"
Q ₆	(Q ₆ •)	R→D	OE _R ; LD _D ; NF	Återför resultat

Uppgift 14.6

Implementera instruktionerna:

LEASP n, SP
 LEAX n, X
 LEAX n, Y

Börja med att föra in styrsignalerna i följande tabeller:

LEASP n, SP

Tillstånd	Summatern	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	M(PC)→T; PC+1→PC		Offset från minnet till T Uppdatera PC
Q ₅	(Q ₅ •)	T+SP →R		Bestäm effektiv adress "n+SP"
Q ₆	(Q ₆ •)	R →SP		Återför resultat till SP

LEAX n, X

Tillstånd	Summatern	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	M(PC)→T; PC+1→PC		Offset från minnet till T Uppdatera PC
Q ₅	(Q ₅ •)	T+X →R		Bestäm effektiv adress "n+X"
Q ₆	(Q ₆ •)	R →X		Återför resultat till X

LEAX n, SP

Tillstånd	Summatern	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	M(PC)→T; PC+1→PC		Offset från minnet till T Uppdatera PC
Q ₅	(Q ₅ •)	T+SP →R		Bestäm effektiv adress "n+SP"
Q ₆	(Q ₆ •)	R →X		Återför resultat till X

Lägg till instruktionerna i filen flisp_ins.fcs och använd följande testsekvens för att kontrollera styrsignalsekvensernas funktion.

Adress	Maskin-kod	Assemblerkod	RTN
20	BE	LEASP 5, SP	SP+5→SP
21	05		
22	BE	LEASP -1, SP	SP+(-1)→SP
23	FF		
24	CC	LEAX 2, X	X+2→X
25	02		
26	DC	LEAX 2, SP	SP+2→X
27	02		

Kontrollera slutligen styrsignalsekvensernas funktion och rätta eventuella fel.

14.2.2 Skriv till minne

Data kan skrivas till minnet med ST-instruktionen ("store"). Instruktionen finns för samtliga register, dvs.:

STA, STX, STY, STSP

Följande utdrag ur FLISP-handboken ger detaljer om instruktionen STA för två olika adresseringssätt:

ST Store register

RTN	R → M (EA)
Flaggor	Påverkas ej
Beskrivning	Lagrar angivet registerinnehåll (A,X,Y,SP) i minnet på den effektiva adressen

Detaljer:

Instruktion	Adressering	Operation	Flaggor
Variant	metod	OP # ~	N Z V C
STA Adr	Absolute	E1 2 3 A → M(Adr)	- - - -
STA n, SP	Indexed	E2 2 3 A → M(n+SP)	- - - -

Uppgift 14.7

Implementera instruktionerna

STA Adr
 STA n, SP

STA Adr

Exekveringsfasen delas upp i två steg:

1: Adr→TA
 2: A→M(TA)

I följande tabell har vi detaljerat RTN-beskrivningen ytterligare. Komplettera tabellen med operationskod och aktiva styrsignaler:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	M(PC)→TA; PC+1→PC;		Effektiv adress från minnet till TA Uppdatera PC
Q ₅	(Q ₅ •)	A → M(TA);		Data från A till minne Ny hämtfas

STA n, SP

Även i detta fall delas exekveringsfasen upp i två steg:

1: n→T
 2: A→M(T+SP)

Komplettera även följande tabell med operationskod och aktiva styrsignaler:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	M(PC)→T; PC+1→PC;		Offset från minnet till T Uppdatera PC
Q ₅	(Q ₅ •)	A → M(T+SP);		Data från A till minne Ny hämtfas

```
test_store.fmem
#ClearAllMemory
#ClearAllRegisters
Operationskoder och
operandinformation läggs i minnet:
#SetMemory 20=E1
#SetMemory 21=10
#SetMemory 22=E2
#SetMemory 23=05
Registren ges initialvärden
#SetRegister A=33
#SetRegister SP=0C
RESET-vektor
#SetMemory FF=20
```

Lägg till dessa instruktioner till instruktionsuppsättningen flisp_ins.fcs och använd sedan följande instruktionssekvens för att testa instruktionerna (se även marginalen).

Adress	Maskin-kod	Assemblerkod	RTN
20	E1	STA 10 ₁₆	A→M(10 ₁₆)
21	10		
22	E2	STA 5, SP	A→M(5+SP)
23	05		

Kontrollera minnesadress 10₁₆ och 11₁₆ som båda ska innehålla värdet 33₁₆ efter instruktionssekvensen.

14.2.3 Registeröverföringar

Instruktionerna TFR ("transfer") och EXG ("exchange") utgör en liten grupp instruktioner för dataöverföring mellan olika register.

TFR Transfer register to register

RTN	R1 → R2
Flaggor	Påverkas ej, såvida man inte kopierar ett registerinnehåll till CC-registret
Beskrivning	Data kopieras mellan angivna register

Detaljer:

Instruktion	Adressering				Operation	Flaggor			
TFR	Variant	metod	OP	#	~	N	Z	V	C
TFR A, CC	Inherent	18	1	2	A → CC	Δ	Δ	Δ	Δ
TFR X, Y	Inherent	1A	1	2	X → Y	-	-	-	-

Instruktioner som kopierar data mellan register är speciellt enkla. De kräver bara att källans OE-signal och destinationens LD-signal aktiveras. Om CC är destinationsregister måste man dock också aktivera rätt styrsignaler för väljaren på CC-registrets ingång.

Uppgift 14.8

I denna uppgift ska du implementera styrsignalsekvenserna för
TFR X, Y
TFR A, CC

I båda fallen räcker det med en klockcykel för exekveringsfasen. Fyll i operationskoder, RTN-beskrivningar och aktiva styrsignaler i följande tabeller:

TFR X, Y

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	X→Y		Data från X till Y Nästa...

TFR A, CC

Till-stånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	A→CC		Data från A till bussen buss kopplas till CC data från buss till CC

Lägg till dessa instruktioner till instruktionsuppsättningen flisp_ins.fcs. Skapa också en fil för test av de implementerade instruktionerna. Placera instruktionssekvensen med start på adress 20₁₆ i minnet. Detaljerna framgår av följande:

Adress	Maskin-kod	Assemblerkod	RTN
20	18	TFR A, CC	A→CC
21	1A	TFR X, Y	X→Y
22			

```
test_tfr.fmem
#ClearAllMemory
#ClearAllRegisters
Operationskoder och
operandinformation läggs i minnet:
#SetMemory 20=18
#SetMemory 21=1A
Registren A och X ges initialvärden
#SetRegister A=FF
#SetRegister X=FF
RESET-vektor
#SetMemory FF=20
```

Utför instruktionerna och kontrollera funktionen.

EXG-instruktionen utbyter två registerinnehåll. För att temporärt lagra det ena registret överför vi dess innehåll opåverkat via ALU:n till register R. Betrakta exempelvis $K \leftrightarrow D$. Den generella RTN-sekvensen för instruktionen kan då skrivas:

$K \rightarrow R, D \rightarrow K, R \rightarrow D$

EXG Exchange register contents

RTN	R1 ↔ R2
Flaggor	Påverkas endast om CC-registret är det ena registret som används
Beskrivning	Data växlas mellan angivna register

Detaljer:

Instruktion	Adressering				Operation	Flaggor			
EXG	Variant	metod	OP	#	~	N	Z	V	C
EXG A, CC	Inherent	9F	1	4	A ↔ CC	Δ	Δ	Δ	Δ
EXG X, Y	Inherent	AF	1	4	X ↔ Y	-	-	-	-

Uppgift 14.9

Implementera instruktionerna:

EXG X, Y
EXG A, CC

Registerutbytet kräver tre cykler, komplettera tabellerna.

EXG X, Y

Till- stånd	Summa- term	RTN- beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)			Data från X till R
Q ₅	(Q ₅ •)			Data från Y till X
Q ₆	(Q ₆ •)			Data från R till X; Nästa...

EXG A, CC

Till- stånd	Summa- term	RTN- beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)			Data från A till R
Q ₅	(Q ₅ •)			Data från CC till A
Q ₆	(Q ₆ •)			Data från R till CC; Nästa...

Lägg till de nya instruktionerna till filen `misc.hwflisp`.

Skapa nu en konfigurationsfil för test av de implementerade instruktionerna. Placera instruktionssekvensen med start på adress 20₁₆ i minnet. Detaljerna framgår av följande:

Adress	Maskin- kod	Assemblerkod	RTN
20	9F	EXG A, CC	A ↔ CC
21	AF	EXG X, Y	X ↔ Y

Placera initialvärden i A och X; kontrollera funktionen.

14.2.4 Unära aritmetiska operationer

Vanligt förekommande operationer kan ges separata instruktioner av prestandaskäl, trots att de kan utföras på andra sätt. I FLISP har vi exempelvis CLR ("clear"), NEG ("negate"), DEC ("decrement") och INC ("increment"). I detta avsnitt arbetar vi speciellt med *decrement*-instruktionen. Implementering av styrsignalsekvenser för de övriga är likartad. Vi börjar med att titta närmre på instruktionens beskrivning:

DEC Decrement register or memory

RTN	A-1 → A eller M(EA)-1 → M(EA)
Flaggor	N: Ettställs om resultatets teckenbit (bit 7) får värdet 1 Z: Ettställs om samtliga åtta bitar i resultatet blir noll V: Ettställs om 2-komplementoverflow uppstår C: Påverkas ej
Beskrivning	Subtraherar 1 från operanden

Detaljer:

Instruktion	Adressering	Operation	Flaggor
Variant	metod	OP # ~	N Z V C
DECA	Inherent	08 1 3 A-1 → A	Δ Δ Δ -
DEC Adr	Absolute	38 2 4 M(Adr) -1 → M(Adr)	
DEC n, SP	Indexed	48 2 4 M(n+SP) -1 → M(n+SP)	
DEC n, X	Indexed	58 2 4 M(n+X) -1 → M(n+X)	
DEC A, X	Indexed	68 1 4 M(A+X) -1 → M(A+X)	
DEC n, Y	Indexed	78 2 4 M(n+Y) -1 → M(n+Y)	
DEC A, Y	Indexed	88 1 4 M(A+Y) -1 → M(A+Y)	

Instruktionen är av typ "Read/Modify/Write", dvs. operanden måste först läsas, för att bli tillgänglig i ALU:n, därefter utförs själva operationen och slutligen ska resultatet skrivas tillbaka till samma plats som där det hämtades.

I de nästföljande uppgifterna ska styrsignalsekvenser för DEC-instruktionen implementeras. Vi börjar med varianten DECA, speciellt för att belysa själva operationen och dess flaggsättning. Vi fortsätter därefter med tre ytterligare adresseringssätt Adr, n, SP och A, X, som väsentligt utökar användbarheten av instruktionen.

Uppgift 14.10

Implementera styrsignalsekvens för instruktionen DECA.

Börja med att skapa ytterligare en konfigurationsfil `test_dec.fmem`, för test av den nya instruktionen. Fortsätt komplettera `flisp_ins.fcs` så att de nya instruktionerna läggs till instruktionsuppsättningen.

Instruktionen beskrivs av följande RTN:

A-1 → R; ALU(N, Z, V) → CC

R → A

För operationen kan lämpligen följande ALU-operation användas:

1	0	0	1	U = D + C _{in}	u ₇	(1)	(2)	(3)
1	0	1	0	U = D + FF ₁₆	u ₇	(1)	(2)	(3)
1	0	1	1	U = D - 1	u ₇	(1)	(2)	(3)

Flaggorna, utom C, ska sättas av ALU-operationen.

```
test_dec.fmem
#ClearAllMemory
#ClearAllRegisters

Operationskod och operandinformation
läggs i minnet:
#SetMemory      20=08

Register A ges initialvärde
#SetRegister     A=FF

RESET-vektor
#SetMemory      FF=20
```

DECA

Till- stånd	Summa- term	RTN- beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ ●)			Operationens resultat till R Flaggsättning
Q ₅	(Q ₅ ●)			Resultatet återförs till A

Skapa också en konfigurationsfil för test av instruktionen.

Adress	Maskin- kod	Assemblerkod	RTN
20	08	DECA	A←A-1
21			

Uppgift 14.11

Implementera styrsignalsekvenser för instruktionsvarianterna:

DEC Adr
DEC n, SP
DEC A, X

Gemensamt för varianterna är att instruktionen måste delas upp i tre delar:

- Adressberäkning, dvs. läsning från minnet
- Operand läses från minnet, operation utförs
- Skrivcykel, återför resultat till minnet.

DEC Adr

Adressberäkningarna utförs under första cykeln samtidigt ökas PC för att peka på nästa instruktion. Under andra cykeln utförs själva operationen så att resultatet kan återföras till minnet under den sista cykeln.

Adr→TA; PC+1→PC

R→M(TA)

Operandens adress hålls här i TA-registret under instruktionen. Komplettera följande tabell:

Till- stånd	Summa- term	RTN- beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ ●)	Adr→TA; PC+1→PC		Adressberäkning
Q ₅	(Q ₅ ●)			Operandhämtning, Operation, resultat till R Flaggsättning
Q ₆	(Q ₆ ●)	R→M(TA)		Resultatet återförs till minnet

DEC n, SP

Denna variant skiljer sig under adressberäkningen genom att vi nu i stället läser den konstanta förskjutningen given av n, till register T. Därefter bildar vi styrsignaler för att adressera minnet med n, SP.

n→T; PC+1→PC

Operandens adress utgörs nu av n+SP, operanden betecknas M(n+SP), komplettera tabellen:

Till- stånd	Summa- term	RTN- beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ ●)	n→T; PC+1→PC		Adressberäkning
Q ₅	(Q ₅ ●)			Operandhämtning, Operation, resultat till R Flaggsättning
Q ₆	(Q ₆ ●)	R→M(n+SP)		Resultatet återförs till minnet

DEC A, X

I denna sista variant används innehållet i register A som förskjutning. Denna variant skiljer sig därför under adressberäkningen från n, SP genom att vi nu i stället läser den innehållet i A till register T. Därefter bildar vi styrsignaler för att adressera minnet med A, X.

A→T

Operandens adress utgörs nu av A+X, operanden betecknas M(A+X), komplettera tabellen:

Till- stånd	Summa- term	RTN- beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ ●)	A→T		Adressberäkning
Q ₅	(Q ₅ ●)			Operandhämtning, Operation, resultat till R Flaggsättning
Q ₆	(Q ₆ ●)	R→M(A+X)		Resultatet återförs till minnet

Då du konstruerat styrsignalsekvenserna fortsätter du med att föra in dem i flisp_ins.fcs.

Modifera nu också konfigurationsfilen för test av DEC (se exemplet i marginalen) så att den innehåller följande testprogram:

Adress	Maskin- kod	Assemblerkod	RTN
20	08	DECA	A←A+1
21	38	DEC B ₁₆	M(B ₁₆)←M(B ₁₆)+1
22	0B		
23	48	DEC 3, SP	M(3+SP)←M(3+SP)-1
24	03		
25	68	DEC A, X	M(A+X)←M(A+X)-1
26			

```
test_dec.fmem
#ClearAllMemory
#ClearAllRegisters
```

Operationskoder och operandinformation
läggs i minnet:

```
#SetMemory 20=08
#SetMemory 21=38
#SetMemory 22=0B
#SetMemory 23=48
#SetMemory 24=03
#SetMemory 25=68
```

Initialvärden för minne och register:

```
#SetMemory 0B=45
#SetMemory 09=55
#SetMemory 08=65
#SetRegister A=06
#SetRegister SP=06
#SetRegister X=03
```

RESET-vektor:

```
#SetMemory FF=20
```

I konfigurationsfilen används också direktiv för att placera följande värden i minne och register för att utföra test:

M(0B₁₆)=45₁₆
 M(09₁₆)=55₁₆
 M(08₁₆)=65₁₆
 A=6
 SP=6
 X=3

Utför nu programsekvensen i simulatören, rätta eventuella fel i styrsignalsekvenserna så att alla instruktioner fungerar som de ska. Då sekvensen har utförts ska du kunna avläsa följande minnesinnehåll:

M(0B₁₆)=44₁₆
 M(09₁₆)=54₁₆
 M(08₁₆)=64₁₆

14.2.5 O villkorlig programflödeskontroll

Vi har hittills enbart sett exempel på instruktioner som uppdaterar PC på ett sätt som gör att den lämnas pekande på nästa sekventiella instruktion i programflödet. För att avbryta ett sådant sekventiellt flöde används någon instruktion för "programflödeskontroll". Instruktionen kan vara villkorlig, och är då kopplad till någon test av flaggorna i CC-registret. Den kan annars vara ovillkorlig vilket betyder att en programflödesändring sker, oavsett flaggornas tillstånd. För ovillkorlig programflödeskontroll har vi bland andra instruktionerna JMP ("jump") och BRA ("branch").

JMP Jump

RTN	EA → PC
Flaggor	Påverkas ej
Beskrivning	Ovillkorlig programflödesändring; nästa instruktion hämtas från effektiva adressen EA.

Detaljer:

Instruktion	Adressering	Operation	Flaggor
variant	metod	OP # ~	N Z V C
JMP Adr	Absolute	33 2 2 Adr → PC	- - - -

BRA Branch always

RTN	PC+Offset → PC
Flaggor	Påverkas ej
Beskrivning	Ett hopp utförs till adressen ADDRESS = PC+Offset. Offset räknas från adressen efter branchinstruktionen, dvs. vid uträkningen av hoppadressen pekar PC på operationskoden som (eventuellt) finns direkt efter branchinstruktionen i minnet

Detaljer:

Instruktion	Adressering	Operation	Flaggor
BRA	metod	OP # ~	N Z V C
BRA Adr	Relativ	21 2 4 PC+Offset → PC	- - - -

De visade instruktionsformerna åstadkommer samma sak, dvs. placerar en ny adress i PC. Skillnaden är hur denna adressinformation kodas in i instruktionen. I det första fallet, JMP, anges den absoluta adressen, medan den andra varianten BRA, kodar adressen som en offset till aktuell PC, dvs. instruktionens förhållande till destinationsadressen är positionsoberoende.

Ytterligare en skillnad mellan instruktionerna är att BRA endast finns med PC-relativ adressering, medan JMP kan användas med flera adresseringssätt.

I detta och de kommande avsnitten ska vi gå igenom exempel på styrsignalsekvenser för instruktioner för programflödeskontroll.

Uppgift 14.12

Då du konstruerat styrsignalsekvenserna fortsätter du med att föra in dem i flisp_ins.fcs.

JMP Adr

Instruktionen placerar effektiva adressen i PC, observera att detta är det värde som följer direkt efter operationskoden i minnet och det räcker alltså med en läscykel. PC ska nu inte, som tidigare inkrementeras för att peka på nästa instruktion. Ingen flaggsättning ska heller utföras, varför implementeringen blir enkel:

M(PC)→PC

Komplettera följande tabell med de nödvändiga styrsignalerna:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ ●)	M(PC)→PC		Data från minne till PC; Nästa...

BRA Adr

Den PC-relativa adressberäkningen för BRA-instruktionen är något mer komplicerad. Instruktionen består av operationskod och "offset", relativ PC, som beräknats då PC innehåller adressen till nästa instruktion. Sambandet mellan BRA-instruktionens adress, destinationens adress och offseten skrivs därför:

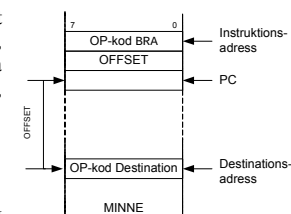
Instruktionsadress + 2 + OFFSET = Destinationsadress

Då exekveringsfasen inleds innehåller PC adressen till OFFSET, RTN för hela instruktionen kan därför skrivas:

PC+1+(PC)→PC

Vi delar upp detta i en styrsignalsekvens genom att först placera PC i T-registret för den kommande beräkningen och samtidigt placera PC i register TA för inläsning av OFFSET från minnet. Observera att värdet för PC i register T nu är 1 mindre än det värde som ska användas vid beräkningen av destinationsadressen:

PC→T; PC→TA



1	0	1	1	U + FF ₁₆ → U	U ₇ (1)(2)(3)
1	0	1	1	D + E + C ₆ → U	U ₇ (1)(2)(3)
1	1	1	1	R → PC	U ₇ (1)(2)(3)

Därefter utförs själva adressberäkningen. Vi kompenserar nu värdet för PC med att addera även konstanten 1 till den resulterande destinationsadressen och resultatet placeras i register R:

$$M(TA)+T+1 \rightarrow R$$

Slutligen återförs resultatet till PC, varefter instruktionen är utförd:

$$R \rightarrow PC$$

Kompletera följande tabell med de saknade styrsignalerna:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •I ₂₁)	PC→T; PC→TA		Operandhämtning
Q ₅	(Q ₅ •I ₂₁)	M(TA)+T+1→R		Adressberäkning
Q ₆	(Q ₆ •I ₂₁)	R→PC		Resultatet återförs till PC

Efter att ha lagt till styrsignalsekvenser för instruktionerna JMP och BRA i kan vi använda följande enkla testprogram för kontroll (se även exempel i marginalen):

Adress	Maskin-kod	Assemblerkod	RTN
20	33	JMP 24 ₁₆	24 ₁₆ →PC
21	24		
22			
23			
24	21	BRA 20 ₁₆	20 ₁₆ →PC
25	FA		
26			

Observera hur offset och destinationsadress för BRA-instruktionen bestäms i programexemplet: det ska gälla att:

$$\text{Instruktionsadress} + 2 + \text{OFFSET} = \text{Destinationsadress}$$

dvs.

$$(24_{16} + 2 + FA_{16}) \pmod{256} = 20_{16}$$

14.2.6 Villkorlig programflödeskontroll

Villkorlig programflödesändring använder flaggorna i CC-registret för att bestämma om en programflödesändring ska utföras, eller inte. Vi kan välja att testa enskilda flaggor, men även olika kombinationer av flaggbitar som realiserar önskade testvillkor.

Följande tabell listar de enkla flaggvillkoren:

Instruktion (Mnemonic)	Funktion	Villkorsindikation
"Branch if carry set" (BCS)	"Hopp" om <i>carry</i>	C=1
"Branch if carry clear" (BCC)	"Hopp" om <i>ICKE carry</i>	C=0
"Branch if equal" (BEQ)	"Hopp" om <i>zero</i>	Z=1
"Branch if not equal" (BNE)	"Hopp" om <i>ICKE zero</i>	Z=0
"Branch if minus" (BMI)	"Hopp" om <i>negative</i>	N=1
"Branch if plus" (BPL)	"Hopp" om <i>ICKE negative</i>	N=0
"Branch if overflow set" (BVS)	"Hopp" om <i>overflow</i>	V=1
"Branch if overflow clear" (BVC)	"Hopp" om <i>ICKE overflow</i>	V=0

Betrakta nu som exempel instruktionen BCS (Branch if carry set).

BCS Adr

Instruktion	Adressering	Operation	Flaggor
BCS	metod OP # ~		N Z V C
BCS Adr	Relativ 28 2 4	if(C=1) PC+Offset → PC	- - - -

Om C-flaggan är 1 ska programflödesändring utföras, annars ska instruktionen omedelbart efter den villkorliga instruktionen utföras. Detta kan kortare skrivas:

```
if (C=1)
    Destinationsadress→PC
else
    PC+2→PC
```

Om vi utgår från utförandefasen av BRA i föregående uppgift ersätts nu den ovillkorliga överföringen i tillstånd Q₆ (R→PC) av den villkorliga överföringen:

```
if (C=1)
    R→PC
else
    PC+2→PC
```

Styrsignalsekvensen för en villkorlig instruktion ska därför dels beräkna destinationsadressen vid uppfyllt villkor och placera denna adress i register R, dels ska adressen till nästa instruktion placeras i PC, för den händelse villkoret inte är uppfyllt. Avslutningsvis aktiveras LD_{PC} endast om villkoret är uppfyllt, RTN cykelvis blir:

```
PC→T; PC→TA,
M(TA)+T+1→R; PC+1→PC,
if(C=1) R→PC
```

Därefter är instruktionen utförd. Vi sätter samman och får den färdiga styrsignalsekvensen enligt följande tabell:

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •I ₂₈)	PC→T; PC→TA	OEP _C ; LD _T ; LD _{TA}	Operandhämtning
Q ₅	(Q ₅ •I ₂₈)	M(TA)+T+1→R; PC+1→PC	MR _f ; f ₁ ; f ₀ ; g ₀ ; g ₁₄ ; LD _R ; INC _{PC}	Adressberäkning
Q ₆	(Q ₆ •I ₂₈)	if(C=1) R→PC	OER _R ; LD _{PC} =C; NF	Om C=1 återförs resultatet till PC

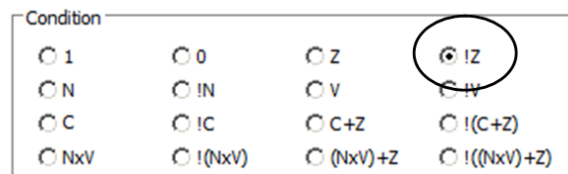
Styrsignalsekvenser för övriga villkorliga instruktioner är likartad; det är bara bildandet av villkoret (LD_{PC}=?) som skiljer dem åt. Följande tabell ger en översikt av villkorsindikatorer med motsvarande uttryck (syntax).

Instruktion (Mnemonic)	Villkors-indikation	Syntax
"Branch if carry set" (BCS)	C=1	C
"Branch if carry clear" (BCC)	C=0	! C
"Branch if equal" (BEQ)	Z=1	Z
"Branch if not equal" (BNE)	Z=0	! Z
"Branch if minus" (BMI)	N=1	N

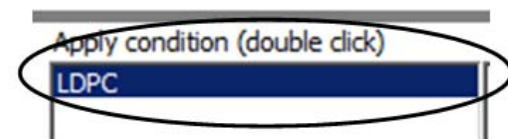
"Branch if plus" (BPL)	N=0	! N
"Branch if overflow set" (BVS)	V=1	V
"Branch if overflow clear" (BVC)	V=0	! V
"Branch if higher" (BHI)	C+Z = 0	! (C+Z)
"Branch if lower or same" (BLS)	C+Z = 1	C+Z
"Branch if greater than" (BGT)	(N⊕V)+Z= 0	! ((NXV)+Z)
"Branch if greater or equal" (BGE)	N⊕V=0	! (NXV)
"Branch if less than" (BLT)	N⊕V=1	NXV
"Branch if less or equal" (BLE)	(N⊕V)+Z=1	(NXV)+Z

Du kan använda Instuction Builder:s villkorsfält för att bilda AND-villkor mellan enskilda styrsignaler och evaluerade villkor.

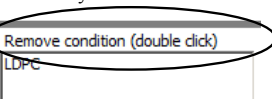
För att som exempel ladda PC under villkoret Z=0, väljer du först detta villkor i Condition-fältet.



Därefter flyttar du styrsignalen från fältet Apply condition, till fältet Remove Condition, genom att dubbelklicka på signalens namn.



Namnet flyttas nu:



och detta betyder att villkoret läggs till just denna styrsignal. Observera att signaler som inte flyttas till detta fält inte heller kommer att ingå i villkorsuttrycket. Vill du återställa, dvs. ta bort villkoret från styrsignalen dubbelklickar du på dess namn i Remove condition-fältet.

Uppgift 14.13

Implementera styrsignalsekvensen för instruktionen BNE.

Instruktion	Adressering	Operation	Flaggor
BNE	metod OP # ~		N Z V C
BNE Adr	Relativ 25 2 4	if(Z=0) PC+Offset → PC	- - - -

Komplettera följande tabell:

BNE Adr

Tillstånd	Summa-term	RTN-beskrivning	Aktiva (=1) Styrsignaler	Kommentarer
Q ₄	(Q ₄ •)	PC→T; PC→TA		Operandhämtning
Q ₅	(Q ₅ •)	M(TA)+T+1→R; PC+1→PC		Adressberäkning
Q ₆	(Q ₆ •)	if(Z=0) R→PC		Om Z=0 återförs resultatet till PC

Lägg till instruktionen i flisp_ins.fcs.

Kontrollera funktionen med hjälp av följande testprogram, se även exemplet i marginalen.

Adress	Maskin-kod	Assemblerkod	RTN
20	F0	LDA #3	3→A
21	03		
22	08	DECA	A-1→A
23	25	BNE 22 ₁₆	if(Z=0)22 ₁₆ →PC
24	FD		
25	21	BRA 20 ₁₆	20 ₁₆ →PC
26	F9		

Konfigurationsfil för test av BNE
#ClearAllMemory
#ClearAllRegisters

Operationskoder och operandinformation läggs i minnet:

#SetMemory	20=F0
#SetMemory	21=03
#SetMemory	22=08
#SetMemory	23=25
#SetMemory	24=FD
#SetMemory	25=21
#SetMemory	26=F9

RESET-vektor:
#SetMemory FF=20

Lägg slutligen till instruktionerna:

BEQ, BCS och BCC

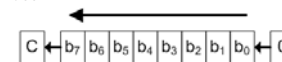
i flisp_ins.fcs.

14.2.7 Skiftoperationer

Instruktioner för skiftoperationer, dvs. aritmetiskt skift, logiskt skift och rotation skiljer sig åt endast i hur bitar skiftas in och ut i ändpositionerna. Vi ger här exempel på logiskt skift.

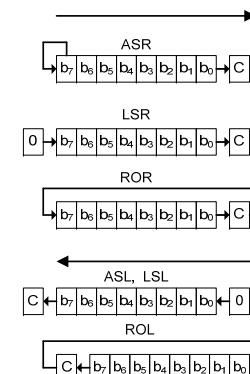
LSL Logical shift left

RTN	A <<1 → A eller M(EA) <<1 → M(EA)
Flaggor	N: Kopiering av bit 7 efter skiftet. Z: Ettställs om samtliga åtta bitar i resultatet blir noll. V: Ettställs om C och bit 7 är olika efter operationen, dvs. overflow vid 2-komplements-representation inträffar. C: bit 7 före skiftet blir ny carrybit efter skiftet.
Beskrivning	Skiftar operanden ett steg till vänster, dvs. multiplicerar ett tal med eller utan inbyggt tecken med 2



Detaljer:

Instruktion	Adressering	Operation	Flaggor
LSL	metod OP # ~		N Z V C
variant	metod OP # ~	A <<1 → A	Δ Δ Δ Δ
LSLA	Inherent 0B 1 3	A <<1 → A	Δ Δ Δ Δ



1 1 0 1	D <<1 (C _{in}) → U
1 1 1 0	(C _{in}) D >> 1 → U
1 1 1 1	(d ₇) D >> 1 → U

Skiftoperationer

15 Maskinprogrammering

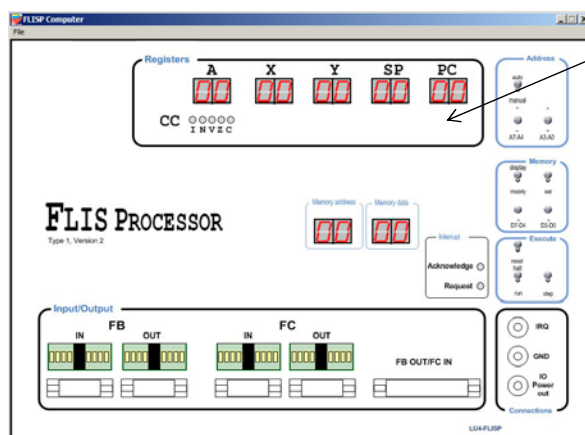
I detta kapitel fortsätter vi att studera hur maskininstruktioner sätts samman i ett maskinprogram som vi placerar i minnet och därefter låta FLISP utföra. Kapitlet omfattar:

- en översikt där du får tillfälle att bekanta dig med hur simulatoren fungerar, samt
- maskinprogrammering, dvs. inmatning och test av operationskoder och operander, för enklare instruktionssekvenser.

15.1 Översikt av simulatoren

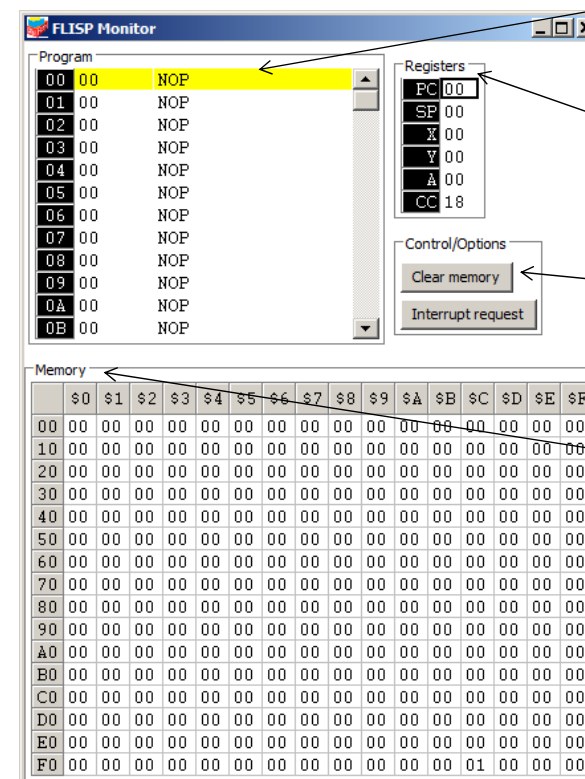
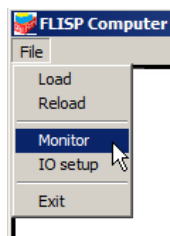
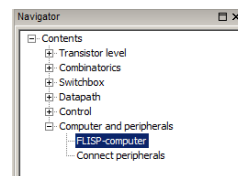
Vi skall börja med att använda FLISP-simulatoren för att mata in ett maskinprogram (ett antal maskininstruktioner) i minnet. Vi fortsätter med att studera utförandet av programmet, dels genom att *stega* oss genom programmet, dels genom att exekvera programmet automatiskt.

Starta FLISP-simulatoren, välj Computer and peripherals|FLISP-computer.



FLISP-simulatoren innehåller en rad olika funktioner men vi koncentrerar oss i detta kapitel på en övervakningsfunktion (monitor) med vilken vi kan undersöka såväl minnesinnehåll som programutförande i FLISP.

Välj nu därför menyalternativet File|Monitor.



Program
Visar minnesinnehåll tolkat som instruktioner (disassemblering).

Registers
Visar registerinnehåll, dessa kan ändras genom att markören placeras i registrets fönster och det nya värdet skrivs in, avsluta med <Enter>.

Nollställ hela minnet

Memory:
Visar minnesinnehåll, minnets innehåll kan ändras genom att markören placeras i minnesadressens fönster och det nya värdet skrivs in, avsluta med <Enter>.

Uppgift 15.1

Placera nu följande programsekvens i minnet genom att mata in maskinkoden för sekvensen med start på adress 20₁₆.

Adress	Maskin-kod	Assemblerkod	RTN
20	F0	LDA	#\$44 44 ₁₆ → A
21	44		
22	94	SUBA	#1 A-1 → A
23	01		
24	E1	STA	\$05 A → M(5)
25	05		
26	21	BRA	\$22 22 ₁₆ → PC
27	FA		

Ställ markören på adress 20₁₆ i minnet och märk upp innehållet. Skriv in den första operationskoden (F0₁₆). Ställ därefter markören på adress 21₁₆. Skriv nu in operanden 44₁₆.

Du kan använda programsektionen för att kontrollera att du verkligen matar in rätt instruktion. Placera programsekvansens startadress 20₁₆ i register PC och tryck <Enter>.

	\$0	\$1	\$2
00	00	00	00
10	00	00	00
20	F0	44	00
30	00	00	00



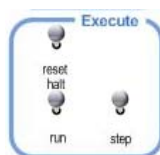
Observera hur innehållet i programsektionen nu uppdateras med disassembleringen av den inmatade instruktionen.

Program				Registers	
20	F0 44	LDA	#\$44	PC	20
22	00	NOP		SP	00
24	00	NOP			

Fortsätt mata in resten av maskinkoderna t.o.m adress 27₁₆ i minnet, observera ändringar i programsektionen allt eftersom minnesinnehållen ändras. Då hela sekvensen matats in ska det se ut på följande sätt:

Program			
20	F0 44	LDA	#\$44
22	94 01	SUBA	#\$01
24	E1 05	STA	\$05
26	21 FA	BRA	\$22
28	00	NOP	

Programexekvering styrs från FLISP-simulatorn. Med hjälp av simulatorn ska du nu titta närmre på instruktionsexekveringen i FLISP. Simulatorn har flera funktioner för detta men i detta kapitel nöjer vi oss med funktionen **step**, utför nästa instruktion. Den instruktion som står i tur att utföras, utpekad av register PC, märks upp med gul bakgrund i simulatorns programsektion.



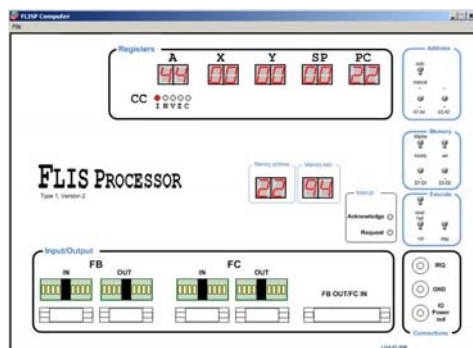
Uppgift 15.2

- Utför nästa instruktion genom att klicka på omkopplaren **step** ("stega").

PC ökas och pekar nu på nästa instruktion. Programsektionen uppdateras och återspeglar detta varefter registersektionen uppdateras med nytt innehåll i register A.

Program				Registers	
20	F0 44	LDA	#\$44	PC	22
22	94 01	SUBA	#\$01	SP	00
24	E1 05	STA	\$05	X	00
26	21 FA	BRA	\$22	Y	00

Observera att såväl monitorfönstret som FLISP-simulatorns fönster uppdateras.



- Stega SUBA-instruktionen, observera innehållet i register A.
- Stega STA-instruktionen, observera innehållet i minnesadress 5.

16 Assemblerprogrammering

I detta avslutande kapitel behandlar vi assemblerprogrammering, en ”ett-till-ett”-översättning från maskinspråket. Kapitlet behandlar huvudsakligen:

- Programutveckling för FLISP och ytterligare funktioner hos simulatören.
- Grundläggande assemblerprogrammering:
 - enkla programstrukturer som flödesval och subrutiner, samt
 - reservering av minne för program och data.
- Beskrivning av yttre enheter anslutna till FLISP-datorn:
 - programstyrd in- och utmatning från/till yttre enheter, samt
 - avbrottsstyrd in- och utmatning från/till yttre enheter.

16.1 Programutveckling i assemblerspråk

DigiFlisp kan användas för programutveckling i assemblerspråk och innehåller funktioner för:

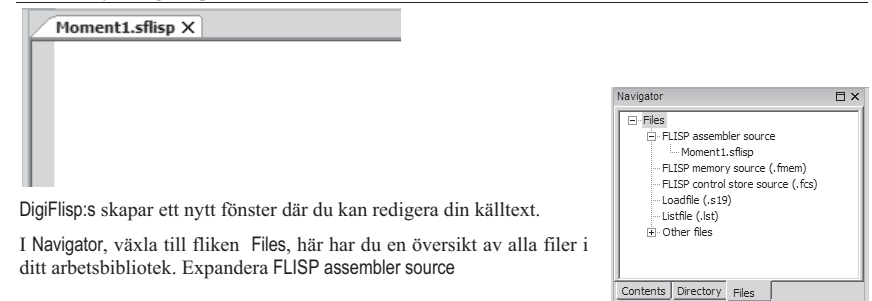
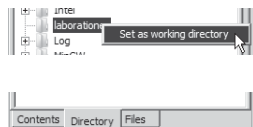
- **Textredigering** – programmet skrivs i form av källtext, dvs. en textfil som innehåller instruktioner och direktiv till assemblern.
- **Assemblering** – då programmet är färdigt måste det översättas till maskinkod innan det kan testas i en måldator eller simulator. Översättningen av programmet, assembleringen, utförs automatiskt av assemblern. Vid assembleringen skapas laddfiler och en listfil.
- **Test** – i en laddfil finns programmet representerat på en form som kan överföras till simulatorer och laborationsutrustning där det tolkas som instruktioner och data. Då programmet har överförts till simulator eller laborationsutrustning kan det utföras (*exekveras*). Man kan då kontrollera programmets funktion.

16.1.1 Skapa ett assemblerprogram

Källtexten skrivs/redigeras med hjälp av en *Editor*, filnamnet ska sluta med *.sflisp* (*source FLISP*) för att kännas igen som en källtextfil för FLISP. Färgad syntax används för att hjälpa dig upptäcka enklare stavningsfel.

Börja med att skapa ett lämpligt arbetsbibliotek för dina filer, (här har vi valt *C:\laborationer* men du vill säkert placera dina filer på något annat ställe). Använd Navigator, fliken Directory och välj arbetsbiblioteket, se även figur i marginalen.

Du skapar en ny källtextfil genom att välja *File | New* från menyn. Därefter skriver du in namnet på den fil du vill skapa, skriv nu *Moment1* och klicka på *Save*. Om du inte anger något filnamstillägg lägger DigiFlisp automatiskt till *.sflisp*. Nu skapas ett nytt fönster:



DigiFlisp:s skapar ett nytt fönster där du kan redigera din källtext.

I Navigator, växla till fliken *Files*, här har du en översikt av alla filer i ditt arbetsbibliotek. Expandera *FLISP assembler source*

Uppgift 16.1

Skriv nu in en källtext med följande instruktionssekvens:

```

;
;      Moment1.sflisp
;
      ORG      $20
start:  LDA     $FC
        STAA    $FB
        JMP     start
|

```

Observera hur texteditorn färglägger din text:

- Ett giltigt symbolfält färgas grönt
- En giltig instruktion (eller ett assemblerdirektiv) färgas blå
- En giltig operand (eller argument till direktiv) färgas röd
- Kommentarer färgas grå.

Notera speciellt hur instruktionen:

```
STAA    $FB
```

färgas grön, dvs. tolkas som en symbol. Detta beror på att vi (avsiktligt) stavat instruktionen fel: rätt stavning är *STA*. Låt felet vara kvar, vi ska strax rätta till det.

- Rader som inleds med *;* tolkas som kommentarer.
- Med direktivet *ORG* (origin) anger du programmets startadress.
- Dollartecknet anger att påföljande talvärde ska tolkas på hexadecimal form (*\$FB = FB₁₆*).

För att spara filen använder du nu *File | Save*.

Om du snabbt vill göra en kopia av denna källtext gör du *File | Save As* och väljer ett nytt namn.

16.1.2 Assemblering

Vid assembleringen översätts källtexten till en laddfil som innehåller maskinkoden, med tillägget .s19 av den inbyggda assemblern. Dessutom skapas ytterligare en laddfil, med annorlunda format och med tillägget .fmem, som även kan användas med den fasta styrenheten i DigiFlisp. Slutligen skapas också en listfil (med tillägget .lst) som kortfattat kan sägas innehålla information från såväl källtexten som laddfilen.

Du kan välja på två olika sätt att assemblera källtexten.

1. Välj från menyn File/Assemble från menyn, då öppnas en dialogruta där du får ange namnet på den fil du vill assemblera.
2. I Navigator/Files, välj den fil du vill assemblera, högerklicka och välj Assemble FLISP source.

Uppgift 16.2

- Assemblera filen Moment1.sflisp

Assembleratorn kommer att klaga på den felstavade instruktionen:

```
Messages
Assembling: 'Moment1.sflisp'
C:/laborationer/Moment1.sflisp:6:Error: Illegal mnemonic
1 error(s).
```

Felutskrift från assemblerator

Efter filnamnet, med fullständig sökväg (som kan se annorlunda ut i ditt exempel) skrivs radnummer inom parentes, därefter typ av fel.

'Illegal mnemonic' betyder att det inte finns någon sådan instruktion (STAA).

- Dubbelklicka nu (vänster knapp) på felutskriften. Markören i marginalen pekar ut raden i källtextfilen som genererat felet.

```
Moment1.sflisp X
;
;      Moment1.sflisp
;
      ORG      $20
start: LDA      $FC
      STAA     $FB
      JMP      start
```

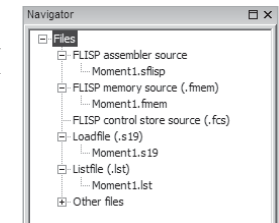
- Det är meningslöst att försöka testa ett program som gett felutskrift vid assembleringen. Rätta därför felet och assemblera på nytt.

Tänk på att korrekt "färgad syntax" inte nödvändigtvis innebär att ditt assemblerprogram är korrekt: det är snarare till för att göra dig uppmärksam på enklare stavfel, syntaxfel etc. Därför kan det i bland hända att du får felmeddelanden även om du stavat såväl instruktioner som operander riktigt.

16.1.3 Laddfil, konfigurationsfil och listfil

Laddfilen används för att överföra programmet (som maskinkod) till en måldator eller en simulator. Det är knappast nödvändigt att känna till laddfilsformatet i detalj men vi visar ändå hur laddfilen (Moment1.s19) för vårt programexempel Moment1.sflisp ser ut.

```
Moment1.sflisp  Moment1.s19 X
S1090020F1FCE1FB3320BA
S9030020DC
```



Exempel på en laddfil.

Vid assembleringen skapas också konfigurationsfilen Moment1.fmem, med det format vi använde i kapitel 14. Detta ger dig möjlighet generera testprogram för FLISP:s fasta styrenhet.

Listfilen (Moment1.lst) kan ofta vara användbar då man testar sitt program. Listfilen innehåller dels det ursprungliga assemblerprogrammet, men dessutom information om den maskinkod som skapats vid assembleringen och på vilka adresser maskinkoden hamnat. En del av listfilen för vårt programexempel ser ut på följande sätt:

```
File: Moment1.lst
1. ;
2. ;      Moment1.sflisp
3. ;
4.          ORG      $20
20 F1 FC    5. start: LDA      $FC
22 E1 FB    6.          STA      $FB
24 33 20    7.          JMP      start
26          8.
```

Listfilen innehåller förutom källtexten information om absoluta adresser och den maskinkod som genereras vid assembleringen.

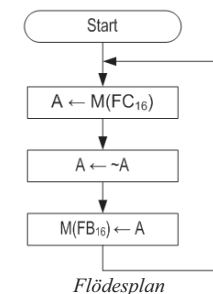
Listfilen används vanligtvis för att identifiera absoluta adresser som man angett med hjälp av symboler. Exempelvis vill man kunna kontrollera vissa variablers värden (minnesinnehåll på någon adress) eller sätta så kallade *brytpunkter* för programexekvering.

I nästa avsnitt ska vi studera ytterligare funktioner hos FLISP-simulatorn. Vi använder då en enkel programsekvens som du nu skapar i följande uppgift.

Uppgift 16.3

Instruktionen COMA används för att invertera (bitkomplementera) bitmönstret i register A. Utgå ifrån Moment1.sflisp, skapa en ny källtext Complement.sflisp och skriv ett assemblerprogram som utför operationer enligt flödesplanen i marginalen.

Spara programmet: du ska få tillfälle att undersöka det alldeles strax.



Flödesplan

16.2 Simulators grundläggande funktioner

Med FLISP-simulators hjälp kan du få en god förståelse för hur assemblerinstruktioner fungerar. Du kan utföra ett assemblerprogram instruktionsvis och i lugn och ro studera effekterna.

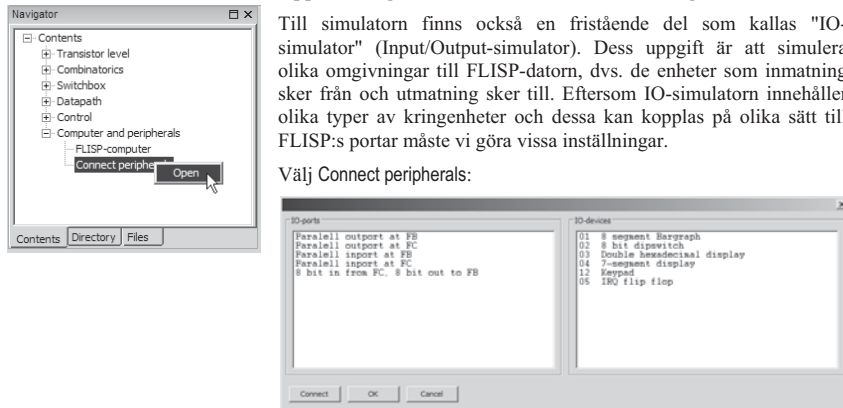
16.2.1 In- och utmatning

Vi ska nu fortsätta arbeta med programmet i `Complement.sflisp` från föregående avsnitt. Eftersom programmet läser från och skriver till portar (utför in- och utmatning) ska vi göra vissa förberedelser innan vi provar programmet i simulatorn.

Öppna dialogfönstret för simulatorns anslutningar

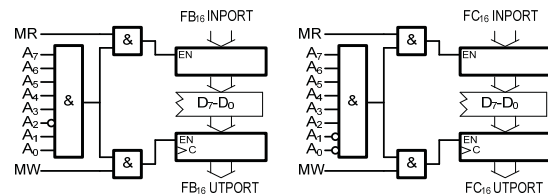
Till simulatorn finns också en fristående del som kallas "IO-simulator" (Input/Output-simulator). Dess uppgift är att simulera olika omgivningar till FLISP-datorn, dvs. de enheter som inmatning sker från och utmatning sker till. Eftersom IO-simulatorn innehåller olika typer av kringenheter och dessa kan kopplas på olika sätt till FLISP:s portar måste vi göra vissa inställningar.

Välj Connect peripherals:



Dialogfönster för IO-simulatorns möjliga anslutningar till in- och utportar hos FLISP

De två minnesadresserna som upplåts för in- och utmatning möjliggör totalt 4 portar eftersom vi använder MR och MW signalerna för att avkoda riktningen.



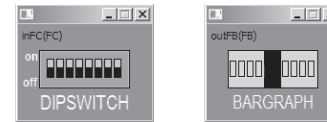
Figur 16.1 FLISP avkodningslogik för IO

De två adresserna FB₁₆ och FC₁₆ är alltså var för sig försedda med en inport och en utport, vilket gör att vi kan ansluta maximalt 4 kringenheter samtidigt. Genom att välja mellan de olika alternativen i dialogfönstret kan du kombinera flera olika anslutningar.

- Välj nu först IO-porten Parallell inport at FC och därefter IO-enheten 8 bit dipswitch. Klicka på Connect.
- Välj nu Parallell outport at FB och därefter 8 segment Bargraph, klicka på Connect.
- Klicka slutligen på OK för att återgå.

Två nya fönster skapas nu: DIPSWITCH i IO-simulatorn används för att simulera en 8-bitars omkopplare. Omkopplaren används för att ge indata till vårt program.

Dessutom skapas den simulerade utenheten BARGRAPH, en ljusdiodramp, som används som indikator för utdata från vårt program.



De olika brytarnas lägen på omkopplaren kan nu läsas som ett 8-bitars dataord från adress FC₁₆:

```
LDA $FC
```

Du kan ändra omkopplarnas lägen genom att klicka i de grå/svarta fälten.



De 8 omkopplarna representeras med fält av grå/svarta ytor. Genom att klicka i ett sådant fält ändrar du omkopplarens utsignal mellan 0 (off) och 1 (on).

Ställ in värdet 1 på omkopplaren genom att klicka på fältet för bit 0.



Din 'klickning' motsvarar en omställning av denna knapp. Du kan ändra varje knapp på samma sätt.

Utdata som skrivs till adress FB₁₆ kan avläsas på ljusdiodrampen:

```
STA $FB
```

Värdet i ackumulator A skrivs till ljusdiodrampen.



Röda indikatorer tolkar du som '1', medan de gula tolkas som '0'.

Uppgift 16.4

1. Öppna filen `Complement.sflisp`

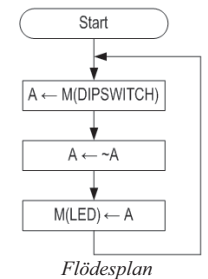
Assemblerdirektiv är EQU (*equate*) kan användas för att definiera konstanter och fasta adresser. Om exempelvis en DIPSWITCH är ansluten till adressen FC₁₆ i måldatorns minne kan vi definiera:

```
DIPSWITCH EQU $FC
```

På motsvarande sätt kan en ljusdiodramp på adress FB₁₆ definieras:

```
LED EQU $FB
```

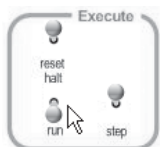
1. Ändra i källtexten `Complement.sflisp` så att symbolerna DIPSWITCH och LED används som portadresser.
2. Assemblera filen, rätta eventuella fel.
3. Du kan starta FLISP-simulatorn via Navigator|Files, expandera Loadfile och välj filen `Complement.s19`, högerklicka:
4. För att kunna sätta programräknaren PC till programmets startadress, 20₁₆, kan vi använda simulatorns monitorfunktion på samma sätt som i förra kapitlet. Välj File | Monitor i FLISP-simulatorn.



Flödesplan



5. Sätt programmets startadress 20_{16} i PC och tryck <Enter>.



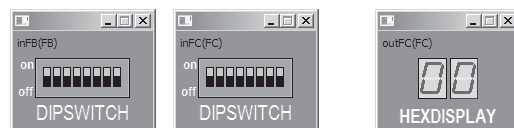
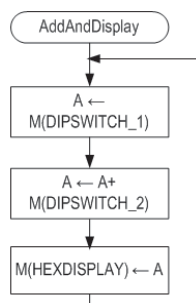
Disassembly				Registers
	contents	decode		
20	F1 FC	LDA	%FC	PC %0
22	0A	COMA		SP %0
23	E1 FB	STA	%FB	X %0
25	33 20	JMP	%20	Y %0
27	00	NOP		A %0
28	00	NOP		CC %0

6. Ställ in följande olika värden på omkopplaren – utför programmet med run, dvs. klicka på strömställaren halv/run, du kan ändra hastigheten genom att nu klicka på step, prova detta, upprepa gånger. Under simulatorns programexekvering ändrar du inställningarna på omkopplaren och observerar ändringarna hos ljusdiodrampen. Fyll i följande tabell:

Inställt värde (binärt)	Avläst värde (binärt)
1111 0000	
1010 1010	
1100 0011	

Uppgift 16.5

I denna uppgift används två omkopplare och en visningsindikator för hexadecimala siffror (HEXDISPLAY).



Skapa en källtextfil `AddAndDisplay.sflisp`. Skriv en programsekvens som läser av och adderar värden från två omkopplare anslutna till adress FB_{16} (`DIPSWITCH_1`) och FC_{16} (`DIPSWITCH_2`) och därefter skriver ut summan på visningsindikatorn ansluten till adress FC_{16} , vi bortser här ifrån spill. Skriv färdigt programsekvensen:

```

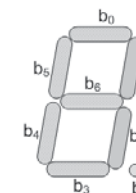
; Symbolfält      Mnemonic/      Operand
;                  direktiv
DIPSWITCH_1:
DIPSWITCH_2:
HEXDISPLAY:
ORG                $20
AddAndDisplay:

```

Assemblera och testa programsekvensen, rätta eventuella fel.

16.2.2 Sju-sifferindikator

Med en sju-sifferindikator kan man på ett enkelt sätt presentera tecken som kan hänföras till de välbekanta siffrorna 0-9. Namnet kommer av att det faktiskt går att representera dessa siffror, om än något kantigt, med endast sju olika streck, vilka också kallas *segment*. Det finns också ett åttonde segment vars uppgift är att tända en decimalpunkt.



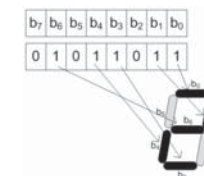
Under detta moment ska du konstruera ett assemblerprogram som utför översättning och utmatning av de binära siffrorna 0 t.o.m. 9 till motsvarande representationer på sju-sifferindikatorn.

Sju-sifferindikatorn 7-SEG DISPLAY fungerar enligt följande:

- Varje bit, i det dataord (8 bitar) som skrivs till utporten, motsvarar ett segment på sju-sifferindikatorn.
- En *etta* tänds ett segment, en *nolla* släcker segmentet.

Översättningen från en siffra (0-9, A-F) till motsvarande sju-segmentskod beror naturligtvis helt och hållet på vilken typ av sju-sifferindikator man använder.

Se exemplet på översättningen av den decimala siffran '2' till dess motsvarande sju-sifferkod i marginalen. För att representera siffran 2 måste vi tända de segment som (tillsammans) ger det mest "2-lika" utseendet, i detta fallet segmenten 0,1,3 och 6. Detta motsvarar hexadecimala talet $5B_{16}$ som därför formar siffran två på sifferindikatorn



Uppgift 16.6

Följande tabell illustrerar förhållandet mellan binära koder och sju-segmentskod. Studera speciellt föregående exempel och komplettera i tabellen med de saknade sju-segmentskoderna.

Decimal siffror		Sju-segmentskod	
	Binär kod	Binär form	Hexadecimal form
0	0000	0111 0111	3F
1	0001		
2	0010	0101 1011	5B
3	0011		
4	0100		
5	0101		
6	0110		
7	0111		
8	1000		
9	1001		

16.2.3 Statisk minnesinitiering

Med "statisk minnesinitiering" menar man att ett bestämt värde placerats på en given adress innan programmet startas.

Detta görs med assemblerdirektivet `FCB` (*Form Constant Byte*) som instruerar assemblern att placera ett värde i måldatorns primärminne.

FCB värde

Flera argument (värden) kan ges med direktivet. Dessa måste då skiljas åt med kommatecken.

FCB värde1, värde2, värde3 etc.

Observera att inga blanka tecken får finnas mellan värden och kommatecken.

Uppgift 16.7

Följande exempel illustrerar en tabell, med start på adress 70₁₆. Tabellen innehåller de decimala värdena 0–9.

ORG \$70
FCB 0,1,2,3,4,5,6,7,8,9

Om tabellen är stor kan man dela upp den i flera rader, assemblerdirektivet ska då upprepas. Följande konstruktion är exempelvis ekvivalent med ovanstående:

FCB 0, 1, 2, 3, 4

FCB 5, 6, 7, 8, 9

Skapa en ny källtextfil `DisplaySeg.sflisp` och lägg här in en liknande tabell som i stället för de decimala värdena innehåller de segmentkoder du bestämde i föregående uppgift.

Symbolfält	Mnemonic/ direktiv	Operand
	ORG	\$70
Segmentkod:	FCB	

I ett flödesdiagram kan vi symboliskt skriva:

$$A \leftarrow M(X+A)$$

Dvs.

- Bestäm en minnesadress genom att addera X och A.
- Placera innehållet på denna adress i A.

Motsvarande operation utförs av assemblerinstruktionen:

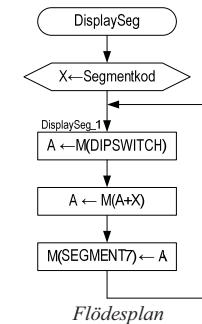
LDA A,X

Uppgift 16.8

I denna uppgift skapas en programsekvens där vi läser ett värde från omkopplaren ansluten till adress FC_{16} , använder detta värde för att indexera i en tabell med start på adress 70_{16} och slutligen skriver ut det indexerade tabellvärdet till sjsuifferindikatorn som är ansluten till adress FB_{16} .

- Fortsätt nu med `DisplaySeg.sflisp`, dvs. tabellen med start på adress 70_{16} som innehåller segmentskoder för siffrorna 0..9 i tur och ordning. Skapa programtexten enligt flödesplanen i marginalen och färdigställ följande:

Symbolfält	Mnemonic/ direktiv	Operand
DIPSWITCH:		
SEGMENT7:		
	ORG	\$20
DisplaySeg:		
DisplaySeg_1:		
	ORG	\$70
Segmentkod:		



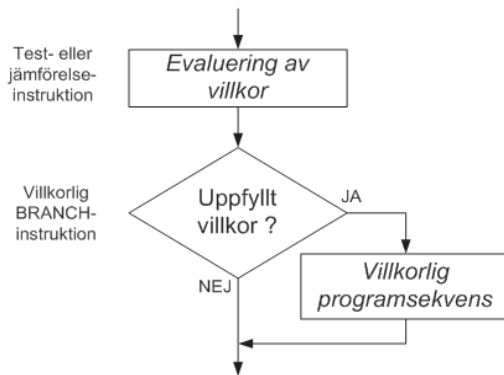
- Assemblera och rätta eventuella fel, koppla 7-SEG DISPLAY till adress FB₁₆ och DIPSWITCH till adress FC₁₆.
- Använd simulatorm och övertyga dig om att programmet fungerar som det ska, dvs. ställ in värdena 0 t.o.m. 9 (0000₂ – 1001₂) på omkopplaren och läs av sifferindikatorn.
- Om du gjort allting rätt ska programmet kunna visa siffrorna 0–9 på sifferindikatorn. Om inte, felsök och rätta i programmet och tabellen.
- Prova slutligen med att ställa in värden som är större än 9 på omkopplaren.

Eftersom segmentkoddatabellen bara innehåller segmentkoder för de 10 första fallen kommer värdena 10–15 att resultera i "odefinierade" segmentkoder utanför tabellen. Det finns olika sätt att lösa det uppkomna problemet.

- Komplettera tabellen med någon speciell segmentkod för "fel", exempelvis 'E' för alla otillåtna värden hos indata.
- Gör en kontroll (jämförelse) av indata och skriv direkt ut felkoden om det är ett otillåtet värde.

16.2.4 Villkorlig programflödesändring

Instruktionstypen B_{cc} ("Branch on condition") används också för att ange så kallade "villkorliga programflödesändringar", dvs. beroende på hur någon test har utfallit så utförs antingen den ena "grenen" eller den andra. Vi ska nu titta närmare på hur detta är tänkt att användas.



Evaluering av villkor

Villkorsevaluering kan göras explicit med speciella test- eller jämförelse-instruktioner. För jämförelse av två operander kan någon av följande instruktioner (*compare*) användas:

Mnemonic	Funktion	Operation
CMPA	Jämför A med minne	(A)-(M)
CMPSP	Jämför SP med minne	(SP)-(M)
CMPX	Jämför X med minne	(X)-(M)
CMPY	Jämför Y med minne	(Y)-(M)

Observera att även andra instruktioner sätter flaggor på samma sätt som dessa. Det kan exempelvis vara överflödigt att använda en jämförelseinstruktion direkt efter en aritmetikinstruktion.

För test av en operand kan någon av följande instruktioner användas:

Mnemonic	Funktion	Operation
TST	Testa minnesinnehåll	(M)-\$00
TSTA	Testa register A	(A)-\$00

Testinstruktionerna påverkar endast Z- och N-flaggorna. Även dessa kan i vissa fall utelämnas, då många andra instruktioner påverkar flaggorna på samma sätt.

Villkorstest

14 olika villkor (*condition codes*) kan anges,

Assemblersyntaxen är:

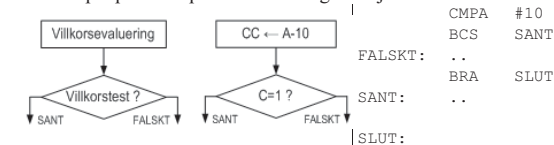
Bcc <symbol>

Där cc står för något flaggvillkor givet i tabellen nedan och <symbol> är någon lägesangivelse i programmet. Flaggorna N,Z,V och C som används för att bilda de olika villkoren finns samlade i CC-registret i simulatorns registersektion.

CC ○○○○○
I N V Z C

Mnemonic	Funktion	Flaggvillkor
Enkla test		
BEQ	"Hopp" om zero	Z=1
BNE	"Hopp" om ICKE zero	Z=0
BMI	"Hopp" om negative	N=1
BPL	"Hopp" om ICKE negative	N=0
BVS	"Hopp" om overflow	V=1
BVC	"Hopp" om ICKE overflow	V=0
Jämförelse av tal utan tecken		
BHI	Villkor: $R > M$	$C + Z = 0$
BCC	Villkor: $R \geq M$	$C = 0$
BCS	Villkor: $R < M$	$C = 1$
BLS	Villkor: $R \leq M$	$C + Z = 1$
Jämförelse av tal med tecken		
BGT	Villkor: $R > M$	$Z + (N \oplus V) = 0$
BGE	Villkor: $R \geq M$	$N \oplus V = 0$
BLT	Villkor: $R < M$	$N \oplus V = 1$
BLE	Villkor: $R \leq M$	$Z + (N \oplus V) = 1$

Ett exempel på flödesplan och kodning är följande:



dvs. "SANT"-grenen utförs om värdet i register A är i intervallet 0..9.

Man kan också koda genom att välja komplementärvillkoret. I vårt fall är exempelvis följande instruktionssekvenser likvärdiga:

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

Uppgift 16.9

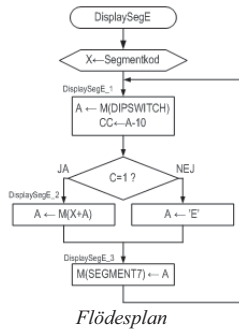
I denna uppgift förbättrar vi funktionen hos programsekvensen vi skapade under föregående uppgift.

- Felkoden 'E' kan visas på sju sifferindikatorn enligt figuren till höger. Definiera segmentkoden som en konstant enligt:

SEG_ERROR: EQU



- Skapa en ny källtext `DisplaySegE.sflisp`, som en kopia av källtexten från föregående uppgift.
- Modifiera programsekvensen så att hänsyn tas till icke-befintliga segmentkoder, se flödesplanen i marginalen. Färdigställ följande assemblerprogram.



Symbolfalt	Mnemonic/ direktiv	Operand
DIPSWITCH:	EQU	\$FC
SEGMENT7:	EQU	\$FB
SEG_ERROR	EQU	
	ORG	\$20
DisplaySegE:		
DisplaySegE_1:		
DisplaySegE_2:		
DisplaySegE_3:		
	JMP	DisplaySegE_1
Segmentkod:	FCB	
	FCB	

- Assemblera, testa och verifiera att programsekvensen fungerar korrekt.

16.2.5 Rinnande ljus, fördröjning

I detta avsnitt ska vi skapa ett "rinnande ljus", och samtidigt se exempel på hur exekveringshastigheten påverkar funktion av en programsekvens.

Uppgift 16.10

Flödesdiagrammet i marginalen visar en programsekvens som åstadkommer ett "rinnande ljus" på en ljusdisdiodramp.

- Komplettera följande ofullständiga programsekvens som en direkt implementering av flödesdiagrammet i marginalen. Spara programsekvensen i filen `RunDiode.sflisp`.

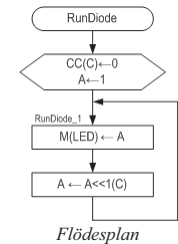
```
; Symbolfalt      Mnemonic/   Operand
LED:              EQU          $FB

                  ORG           $20

RunDiode:

RunDiode_1:

                JMP            RunDiode_1
```



- Assemblera filen, rätta eventuell fel.
- Anslut ljusdiodrampen till adress FB₁₆.
- Kontrollera programsekvensen med simulatorns funktion run, ljusdiодerna ska nu tändas en efter en från höger till vänster och ge illusionen av ett "rinnande ljus". Kontrollera programmets funktion och rätta eventuella fel.
- Prova nu även programexekvering i högre hastighet (klicka på step för att ändra hastigheten). Det "rinnande ljuset" ersätts då av ett flimmer på ljusdiодerna.

Eftersom snabb exekvering ger intrycket av att samtliga dioder flimrar, beroende på att dioderna tänds och släcks i allt för högt tempo, behöver vi fördröja programexekveringen mellan det att en diod tänds och släcks. En sådan fördröjning är lämplig att utföra i form av en *subrutin*.

16.2.7 Subrutiner

Det är vanligt att man försöker organisera ett program i olika *funktioner (procedurer)* eller som vi vanligtvis, då det gäller assemblerspråk, kallar det *subrutiner*. FLIS-processorn tillhandahåller instruktionerna JSR, BSR och RTS för att underlätta modularisering av program genom användande av subrutiner.

En subrutin karakteriseras av att den har ett *inträde* och ett *utträde*. Inträdet anges oftast genom att man placerar en symbol i symbolfältet. Utträdet, dvs. "åter från subrutin" specificeras av en speciell maskininstruktion, RTS.

Operanden för JSR är en subrutins inträde. Denna kan anges med flera olika adresseringssätt, men det vanligaste är en symbolisk adress.

JSR-instruktionen utför två väsentliga operationer:

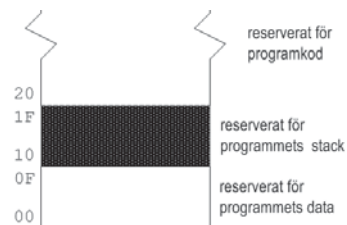
- adressen till nästa instruktion sparas på stacken
- programräknaren initieras med adressen till subrutinen.

RTS-instruktionen utför operationen:

- adressen till nästa instruktion återställs från stacken.

Stackpekaren, register SP, måste alltså ha initierats med en lämplig adress innan JSR används. För detta måste man dessutom först ha avdelat en lämplig del av minnet för stackanvändningen.

I följande figur, som delvis beskriver adressrummet i FLISP (en fullständig beskrivning finns i *appendix E*), visar hur 16 bytes har reserverats för stackanvändning (adresser 10₁₆–1F₁₆), ytterligare 16 bytes reserverats för data (adresser 0–F₁₆) och utrymme för programkod börjar på adress 20₁₆.



Figur 16.2 Minnesanvändning i FLISP

För att initiera stackpekaren används då lämpligen instruktionen:

LDSP # \$20

Vi utformar nu ett nytt huvudprogram RunDiodeDelay för användning tillsammans med en subrutin som vi kallar Delay. Flödesplanerna för den föregående och den nya lösningen visas i marginalen.

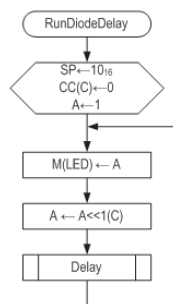
Då vi utformar subrutinen Delay, måste vi också ta hänsyn till själva huvudprogrammet. Av flödesplanen framgår tydligt att såväl register A som C-flaggan i register CC används i huvudprogrammet. Deras innehåll måste därför bevaras över anropet till subrutinen Delay.

Följaktligen måste subrutinen Delay spara dessa registerinnehåll innan de används, för att avslutningsvis återställs dem.

Stacken används också för att tillfälligt spara registerinnehåll i sådana här fall. Instruktionerna PSH (*PuSH register*) respektive PUL (*PULl register*) används för detta ändamål.

Subrutins namn

Flödesplanssymbol för subrutin



Uppgift 16.11

Flödesdiagrammet i marginalen visar en subrutin Delay.

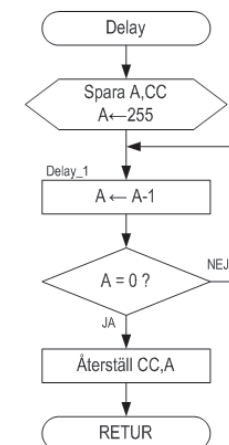
Komplettera följande ofullständiga programsekvens av det modifierade huvudprogrammet tillsammans med subrutinen. Spara programmet i filen RunDiodeDelay.sflisp.

```

; Symbolfält      Mnemonic/   Operand
; direktiv
LED:              EQU         $FB

;
;
; ORG             $20
;
RunDiodeDelay:
;
;
RunDiodeDelay_1:
;
; JSR             Delay
; JMP             RunDiodeDelay_1
;
Delay:            PSHA
;                PSHC
;
Delay_1:
;
;                PULC
;                PULA
;                RTS

```



Flödesplan för subrutinen 'Delay'

- Assemblera filen, rätta eventuella fel.
- Anslut ljusdiodrampen till adress FB₁₆.
- Kontrollera programsekvensen med simulatorns största exekveringshastighet. Ljusdioderna ska nu tändas en efter en från höger till vänster och ge illusionen av ett "rinnande ljus". Kontrollera programmet funktion och rätta eventuella fel.

16.2.8 Terminal, parametrar och returvärdet

En *terminal* är en kombinerad in- utmatningsenhet för inmatning av ASCII-tecken från ett tangentbord och utmatning av ASCII-tecken till en bildskärm. I detta avsnitt använder vi FLISP-simulatorns termin al för att dels illustrera enkel parameteröverföring till och från subrutiner, dels för att visa hur du kan använda brytpunkter då du testat ett program.



FLISP-simulatorns terminal

Terminalen ansluts till två portar samtidigt, en utport och en inport. Ett tecken som matas in i terminalens fönster (svart) blir tillgängligt på terminalens inport, adress FC_{16} . Tecknet finns kvar tills det kvitterats genom att 0 skrivs till terminalens utport, adress FB_{16} .

Tecken matas ut till terminalens fönster genom att ett 7-bitars ASCII-tecken ($1-7F_{16}$) skrivs till utporten. För en fullständig beskrivning av ASCII-koderna, se *appendix G*.

Parametrar och returvärden i register

För att ge en så enkel användning av terminalen som möjligt definierar vi två olika subrutiner:

- `TerminalOut`, skriver ett tecken till terminalens bildskärm.
- `TerminalIn`, läser ett tecken från terminalens tangentbord.

Register A kan användas för att rymma ett enstaka 7-bitars ASCII-tecken. En lämplig konvention är då att överföra tecknen till TerminalOut respektive från TerminalIn via register A. Utmatningsrutinen kan då beskrivas på följande sätt (se även flödesplanen i marginalen).

```
; subrutin TerminalOut  
; Skriver ett ASCII-tecken till terminalens bildskärm  
; Parametrar: Register A, tecken som ska skrivas
```

Uppgift 16.12

Vi kan använda subrutinen för en programsekvens som matar ut en textsträng till terminalen. Ett nytt assemblerdirektiv kommer då till användning:

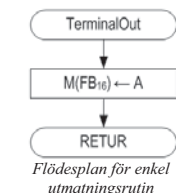
```
FCS "ascii text" ; Form Constant String
```

Med direktivets hjälp skapar vi alltså en sträng med ASCII-tecken i minnet. För att markera strängens slut är det brukligt att placera 0 efter ascii-tecknen. Skapa en ny källtextfil `HelloWorld.sflisp`, implementera programsekvensen enligt flödesdiagrammet.

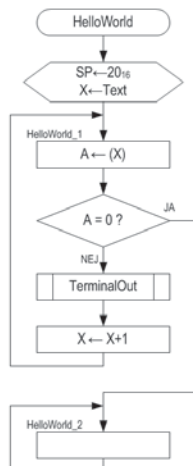
```

; Symbolfalt      Mnemonic/      Operand
; direktiv
ORG              $20
HelloWorld:      LDSP             #$20
                  LDX              #Text
HelloWorld_1:
HelloWorld_2:
TerminalOut:
Text:            FCS              "Hello World"
                  FCB              0

```



Flödesplan för enkel
utmatningsrutin



*Flödesplan för
programsekvens som matar
ut en textsträng*

Då terminalfönstret är aktivt skickas tecken från tangentbordet till terminalen. Tecknet placeras i en intern buffert och blir tillgängligt för ett program.

Subrutinen `Terminaln`, undersöker om det finns något tecken i bufferten. Om bufferten är tom innehåller den tecknet 0; om en tangent tryckts ned innehåller bufferten ASCII-tecknet för tangenten. För att indikera för terminalen att tecknet tagits om hand måste det kvitteras genom att 0 skrivs till terminalens utport.

```
; subrutin TerminalIn
; Läser ett ASCII-tecken från terminalens tangentbord
; Returvärde: Register A, tecken som lästs från
; tangentbordet
```

Subrutinen `Terminalln "väntar"` alltså på att en tangent trycks ned, kvitterar ASCII-tecknet och returnerar sedan koden för den nedtryckta tangenten.

Uppgift 16.13

För att kontrollera funktionen skapar vi ett enkelt testprogram, `Echo`, som kontinuerligt läser ett tecken från terminalens tangentbord för att därefter skriva samma tecken till terminalens bildskärm. Färdigställ följande programsekvens och implementera den i filen `Echo.sflisp`, vi testar den i nästa uppgift.

```

; Symbolfalt      Mnemonic/      Operand
                   direktiv
                   ORG             $20

Echo:
Echo_1:

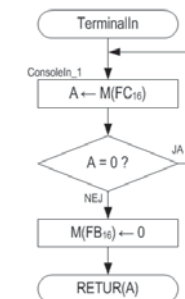
TerminalIn:

TerminalOut:

```

Brytpunkter

En *brytpunkt* kan sättas på på en adress i programmet som innehåller en operationskod. Då simulatorn ska utföra instruktionen och upptäcker att dess adress överensstämmer med någon brytpunkt stoppas i stället exekveringen. Detta ger dig ett bekvämt sätt att exekvera programmet tills något villkor är uppfyllt eller helt enkelt till någon speciell subrutin, där du sedan kan undersöka programexekveringen i detalj.



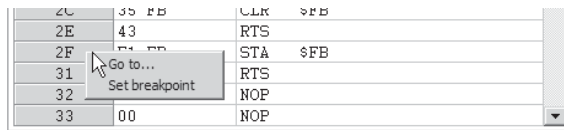
Flödesplan för enkel
inmatningsrutin



Flödesplan för test av inmatningsrutin

Brytpunkter kan sättas på någon adress med användning av brytpunktstabellen. För att bestämma vilken brytpunkten ska placeras på är det lämpligt att använda den listfil som skapas vid assembleringen.

Lägg till en brytpunkt genom att högerklicka på en rad i programfönstret.



Två alternativ ges nu, antingen en tillfällig brytpunkt, Go to..., eller en permanent brytpunkt (Set breakpoint). Den tillfälliga brytpunkten använder du då du vill stoppa programmet *en gång* vid denna punkt. Permanent brytpunkt används då programmet ska stoppas *varje gång* programmet når denna adress.

En brytpunkt illustreras genom att raden får en röd bakgrund, för en tillfällig brytpunkt är texten svart, för en permanent brytpunkt är texten vit.

För att ta bort en permanent brytpunkt högerklickar du på den uppmärksade raden och väljer Remove breakpoint.

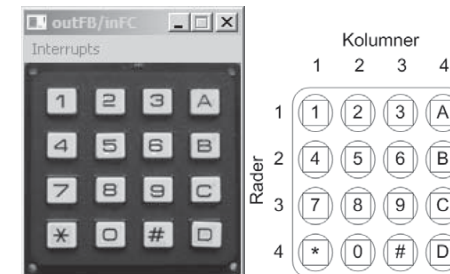
Bekanta dig snabbt med brytpunktsfunktionerna innan du utför nästa uppgift.

Uppgift 16.14

- Assemblera filen `Echo.sflisp`.
- Ladda programmet till simulatören.
- Anslut terminalen till portarna.
- Öppna listfilen `Echo.lst` och lokalisera subrutinen `TerminalOut`, använd monitorns programsektion och sätt en brytpunkt på adressen till subrutinens första instruktion.
- Starta programmet, aktivera (klicka i) terminalfönstret och skriv in tecknet 'b'. Programmet ska nu stanna på den första instruktionen i `TerminalOut`. Fortsätt med att stega instruktionsvis och kontrollera att tecknet 'b' nu också skrivs till terminalens bildskärm.
- Ta bort brytpunkten och starta programmet igen, skriv några godtyckliga tecken och kontrollera att de skrivs ut korrekt.
- Prova också med att sätta ut en tillfällig brytpunkt, dvs. Go to....

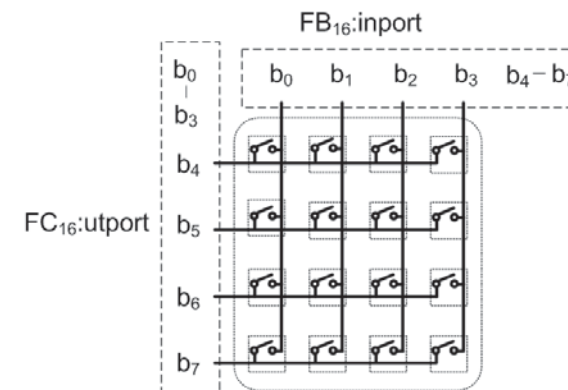
16.2.10 Tangentbord

KEYPAD är ett enkelt tangentbord med 16 tangenter och lämpar sig för inmatning av numeriska värden till ett program. Tangentbordet måste avkodas programvarumässigt, till skillnad från terminalens tangentbord som genererar ASCII-tecken.



Tangentbordet kan också konfigureras för att generera *avbrott*, men detta återkommer vi till i nästa avsnitt.

Tangentbordet är organiserat i rader och kolumner. Raderna har anslutits till bit 4 t.o.m bit 7 i utporten på adress FC_{16} . Kolumnerna är anslutna till bit 0 t.o.m bit 3 hos inporten på adress FB_{16} .



För att känna av en tangentnedtryckning måste någon rad i utporten aktiveras. Genom att sätta någon bit till '1' aktiveras motsvarande rad.

Efter att en rad aktiverats kan kolumnerna läsas av från inporten. En tangents omkopplarfunktion är sluten då tangenten trycks ned och öppen då tangenten släpps upp. Om någon rad aktiverats och någon av kolumnerna har logikvärdet '1' betyder detta därför att tangenten i motsvarande kolumn är nedtryckt (det kan vara fler än en tangent). Om ingen tangent är nedtryckt så läses endast ettur från kolumnerna.

Med vetskap om vilken rad som aktiverats och vilken kolumn som ger indikation ('0') vet vi den nedtryckta tangentens position och kan därför också bestämma dess tangentkod.

Uppgift 16.15

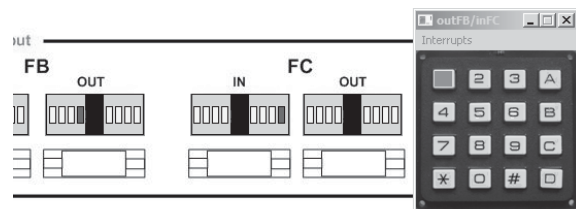
Undersök tangentbordets funktion.

1. Anslut enheten KEYPAD.

2. Ställ in adressen FB_{16} och värdet 10_{16} hos FLISP-simulaton.

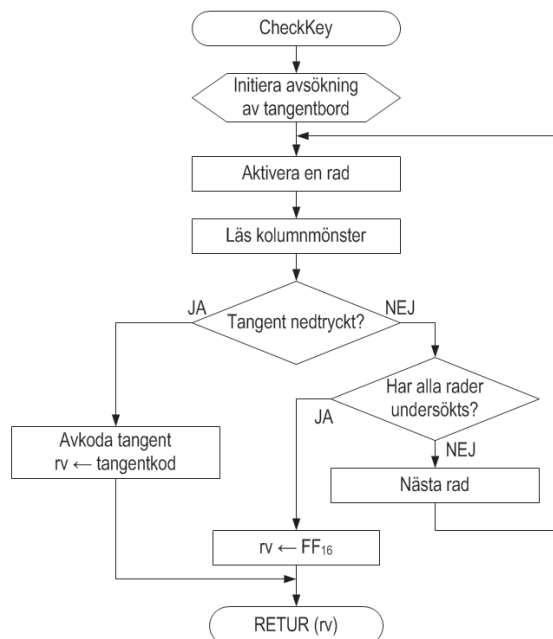


3. "Tryck ned" (klicka en gång) på tangentbordets första tangent och observera hur värdet på inporten ändras. Klicka ytterligare en gång för att "släppa upp" tangenten.



4. Upprepa förfarandet, aktivera andra rader och tryck ned tangenter i andra kolumner.

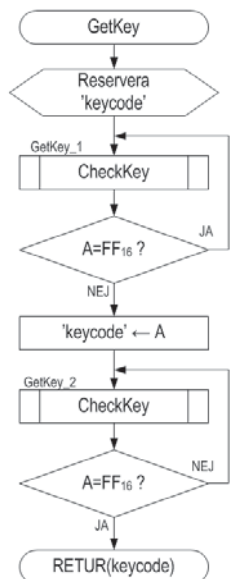
För att avgöra om en tangent är nedtryckt avsöks tangentbordet rad för rad, dvs. en rad aktiveras och kolumnerna läses av. Om en nedtryckt tangent upptäcks, ska den avkodas och dess tangentkod bestämmas. Om ingen tangent är nedtryckt ska en felkod, i detta fall FF_{16} , ange just detta. Följande algoritm kan användas för en sådan funktion:



Uppgift 16.16

Skapa en ny källtext, `CheckKey.sflisp` och implementera `CheckKey` enligt flödesplanen. Testa med ett enkelt program enligt följande:

Symbolfält	Mnemonic/ direktiv	Operand
	ORG	$\$20$
TestCheckKey:	LDSP	$\$20$
TestCheckKey_1:	JSR	CheckKey
	JMP	TestCheckKey_1
CheckKey:	LDA	$\$10$;bitmönster rad 1
CheckKey_1:	STA	$\$FB$;aktivera (nästa) rad
	LDY	$\$FC$;läs kolumner
		;om nedtryckt,avkoda
		;nästa rad
		;om fler, nästa rad
		;alla genomsökta
		;ingen nedtryckt
CheckKey_2:	LSRA	;tangent nedtryckt...
	LSRA	;skifta radmönster till
	LSRA	;låg nybble
	LSRA	
	LSRA	;.. omvandla radmönster..
	CMPA	$\#4$;.. till radoffset...
	BNE	CheckKey_3 ;.. 0,1,2,3
	SUBA	$\#1$
CheckKey_3:		; multiplicera radoffset..
		; med 4 och spara ..
	PSHA	; .. på stacken
		; kopiera Y till A
		; via stacken
		; översätt kolumnmönster
		; till kolumnoffset
CheckKey_4:		; bestäm tangentoffset
		; som (radoffset*4)+
		; kolumnoffset
		; balansera stacken
	RTS	
KeyCode:	FCB	1,2,3,\$A,4,5,6,\$B,7,8,9,\$C,\$F,0,\$E,\$D



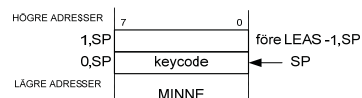
Flödesplan för tangentbordsrutin 'GetKey'

Flödesdiagrammet i marginalen visar funktionen GetKey som utformats så att den väntar tills en tangent tryckts ned, därefter väntar till tangenten släpps upp och slutligen returnerar den nedtryckta tangentens värde.

Lokala variabler

Register A måste här användas både för att undersöka nedtryckt och uppsläppt tangent eftersom registret används för returvärdet från CheckKey. Själva tangentkoden, som ju läses samtidigt som programsekvensen detekterar en nedtryckt tangent måste därför sparas på något annat sätt. En möjlighet är att först reservera utrymme på stacken och sedan använda denna plats för tillfällig lagring av den nedtryckta tangentens kod. Då tangenten släppts upp igen kan tangentkoden återställas från stacken och returneras i register A.

Instruktionen LEASP -1, SP kan användas för att minska stackpekaren med 1, dvs. reservera 1 byte minnesutrymme. Adressen till detta minnesutrymme, dvs. 0, SP, kallar vi symboliskt för keycode.



Flödesplanen i marginalen beskriver en tangentbordsrutin GetKey där följande instruktioner är lämpliga för hantering av den lokala variabeln keycode:

"Reservera keycode"

```
LEASP -1, SP
```

"keycode ← A"

```
STA 0, SP
```

"RETUR(keycode)"

```
LDA 0, SP ; återställ 'keycode' från stack till register A
LEASP 1, SP ; återställ stackpekare
RTS ; återgå från subrutin
```

Vi konstruerar nu subrutinen GetKey.

Uppgift 16.17

Skapa en ny källtext, TestGetKey.sflisp och implementera GetKey enligt flödesplanen. Testa med ett enkelt program på samma sätt som tidigare.:

Uppgift 16.18

I denna uppgift ska du konstruera ett program DisplayKbd som visar den sist nedtryckta tangenten som en hexadecimal siffra på en sju-segmentsindikator.

Programmet ska delas upp i

- huvudprogram DisplayKbd.
- subrutin för tangentbordsinmatning, GetKey enligt föregående uppgift.
- subrutin för utmatning av hexadecimal siffra (0..F₁₆) på en sju-segmentsindikator DispSeg7; jämför med Uppgift 16.9.

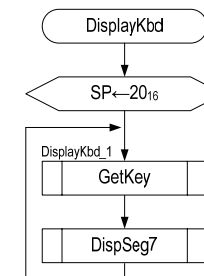
För DispSeg7 gäller att rutinen anropas med det hexadecimala värdet (tangentkoden) i register A. Alla möjliga tangentkoder ska kunna visas och det behövs därför ingen felkontroll.

- Skapa en ny källtextfil DisplayKbd.sflisp. Återanvänd kod från tidigare uppgifter. Börja sedan med att komplettera segmentskodb Tabellen med giltiga sju-segmentskoder för de hexadecimala värdena A₁₆ till och med F₁₆.
- Utforma huvudprogrammet enligt flödesdiagrammet i marginalen och färdigställ enligt följande.

```
; Symbolfält      Mnemonic/ direktiv      Operand
=====
SEGMENT7:         EQU          $FC
=====
ORG               $20
=====
DisplayKbd:
DisplayKbd_1:
JMP               DisplayKbd_1
=====
GetKey:            ...          enligt tidigare...
=====
DispSeg7:
=====
RTS
Segmentkod:       FCB
FCB
FCB
```

- Kontrollera att programmet fungerar som avsett.

Decimal siffra	Sju-segmentskod		
	Binär kod	Binär form	Hex form
0	0000	0111 0111	3F
1	0001		
2	0010	0101 1011	5B
3	0011		
4	0100		
5	0101		
6	0110		
7	0111		
8	1000		
9	1001		
A	1010		
B	1011		
C	1100		
D	1101		
E	1110		
F	1111		



Flödesplan för 'DisplayKbd'

16.2.11 Avbrott

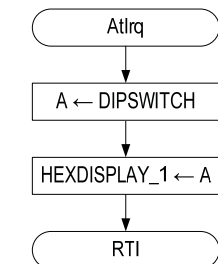
FLISP initieras för avbrott genom att:

- Stackpekare initieras.
- En avbrottsrutin, speciellt konstruerad för den avbrottsgenererande enheten, konstrueras.
- Avbrottsvektorn, dvs. adress FD_{16} , initieras med adressen till denna avbrottsrutin.
- Processorns avbrottsmask, dvs. I-flaggan i CC-registret, nollställs så att processorn accepterar ett avbrott då detta aktiveras.

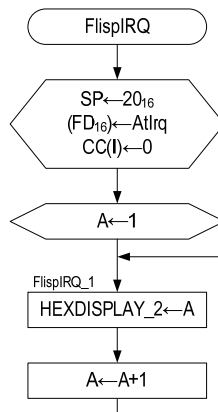
Uppgift 16.19

I denna uppgift ska du konstruera en enkel applikation för användning av avbrott.

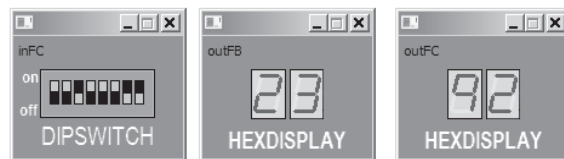
- En DIPSWITCH ska anslutas till adress FC₁₆ och två indikatorer ska anslutas, HEXDISPLAY_1 till adress FB₁₆ och HEXDISPLAY_2 till adress FC₁₆.
- Huvudprogrammet ska konstrueras som en ”räknare” som hela tiden räknar upp värdet hos HEXDISPLAY_2.
- En avbrottsrutin ska läsa data från omkopplaren och skriva värdet till HEXDISPLAY_1.



Flödesplan för avbrottsrutinen



Flödesplan för huvudprogrammet



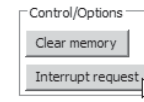
- Skapa en källtextfil `FlispIRQ.sflisp`, implementera applikationen.

```

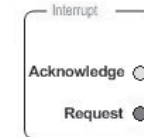
; Symbolfalt      Mnemonic/      Operand
; direktiv
DIPSWITCH:        EQU             $FC
HEXDISPLAY_1:     EQU             $FB
HEXDISPLAY_2:     EQU             $FC
                  ORG              $20
FlispIRQ:
FlispIRQ_1:
AtIrq:
RTI

```

För att testa funktionen använder vi en inbyggd simulatorfunktion för att generera avbrottsignalen IRQ.



Avbrottet aktiveras genom att du klickar på Interrupt request i monitorfönstret.



Då FLISP detekterer avbrottet lyser dioden Request.



Då FLISP startar avbrottshanteringen tänds dioden Acknowledge, samtidigt släcks dioden Request.

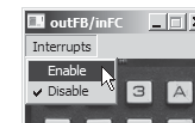


Avbrottet deaktiveras då simulatorn utför instruktionen `RTI`.

Testa programmet på följande sätt:

- Stega ett antal instruktioner så att initieringar i huvudprogrammet har utförts några värden skrivits till `HEXDISPLAY_2`.
- Generera ett avbrott genom att klicka på interrupt request. Observera att dioden interrupt pending ännu inte påverkas; det sker först då nästa instruktion ska utföras (flanktriggad IRQ).
- Stega nästa instruktion, observera hur avbrottshantering inleds och indikatordioden tänds. Stega igenom avbrottsrutinen och notera hur indikatordioden släcks då `RTI` utförs.

Det är vanligt att yttre enheter kan generera avbrott. Exempelvis kan tangentbordet fås att generera avbrott (IRQ) genom att menyvalet Interrupts aktiveras.



Aktivering av tangentbordets avbrottsmekanism

Då avbrottsmekanismen hos tangentbordet är aktiverad kommer en godtycklig tangentnedtryckning att signaleras till FLIS-processorns avbrottsystem. Om även processorns avbrottsmekanism är aktiverad, dvs. l-flagan i CC-registret är 0, kommer avbrottet att betjänas. Observera att avbrottsignalen är aktiv så länge någon tangent är nedtryckt (nivåtriggad IRQ).

Globala variabler

Överföring av data mellan avbrottsrutin och huvudprogram måste ske via ett delat ("globalt") minnesutrymme. Utrymme för globala variabler skapas med assemblerdirektivet

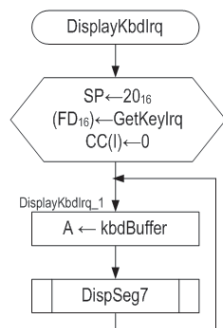
RMB <antal> ; Reserve memory bytes

där <antal> anger hur många bytes som ska reserveras.

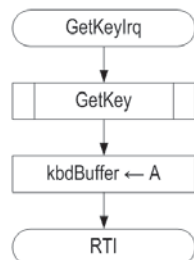
För att exempelvis definiera och reservera utrymme för en global variabel `kbdBuffer` som upptar 1 byte, skriver vi:

```
kbdBuffer: RMB    1
```

Vi kan sedan använda denna för att kommunicera data mellan den avbrottsdrivna tangenbrordsrutinen och huvudprogrammet.



Flödesplan för huvudprogrammet



Flödesplan för den avbrottsdrivna subrutinen

Uppgift 16.20

I denna uppgift ska du konvertera programmet från

Uppgift 16.18 för användning med avbrott. Båda subrutinerna GetKey och DispSeg7 ska återanvändas i oförändrat skick.

- Skapa en ny kalltextfil `DisplayKbdIrq.sflisp`.
- Kopiera subrutinerna `GetKey` och `DispSeg7`.
- Implementera det nya huvudprogrammet `DisplayKbdIrq` och avbrottsrutinen `GetKeyIrq` enligt flödesdiagrammen i marginalen.

Symbolfalt	Mnemonic/ direktiv	Operand
	ORG	\$20
DisplayKbdIrq:		
GetKeyIrq:		
GetKey:	...	som tidiqare...
DispSeg7:	...	som tidiqare...
	ORG	0
kdbBuffer	RMB	1

Glöm inte att aktivera tangentbordets avbrottsmekanism då du testar programmet. Kontrollera funktionen och rätta eventuella fel

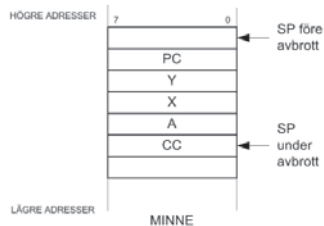
16.2.12 Otillåten operationskod

Instruktionslistan innehåller ett fåtal otillåtna operationskoder. Då FLISP försöker avkoda en sådan vidtar i stället undantagshantering och i följande uppgift ser vi exempel på hur sådan kan utföras.

Uppgift 16.21

I denna uppgift ska du konstruera en hanteringsrutin för "otillåtna operationskod", som bestämmer och skriver ut den otillåtna operationskoden till en HEXDISPLAY och därefter återupptar exekveringen efter den otillåtna operationskoden.

- Skapa en källtext `IllegalOp.sflisp`.
- Färdigställ följande program där hanteringsrutinen `IllOpService` är ofullständig. I rutinen har stacken följande utseende:



Vid detta undantag pekar PC på instruktionen omedelbart efter den otillåtna operationskoden.

```
; Symbolfält      Mnemonic/  Operand
; direktiv
HEXDISPLAY:      EOU          $FB
;
;                ORG          $20
IllegalOp:       LDSP         #$20
;                LDX          #IllOpService
;                STX          $FE
;                NOP
;                FCB          $DF
;                NOP
;                FCB          $EF
;                NOP
;                FCB          $FF
;                NOP
;                JMP          IllegalOp
;
IllOpService:    LDX
;
;                LDA
;
;                STA          HEXDISPLAY
;                RTI
```

- Assemblera, rätta eventuella fel och ladda därefter till simulatören. Koppla en HEXDISPLAY till adress `FB16`.
- Testa programmet genom att stega instruktionsvis. Utskrifterna ska i tur och ordning vara `DF`, `EF` och `FF`.

16.2.13 RESET

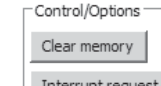
Som sista övningsmoment ska du undersöka FLIS-processorns återstartsförlopp.

Uppgift 16.22

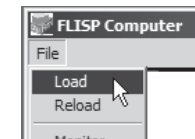
Fortsätt arbeta med `IllegalOp.sflisp`. Placera startadressen till programmet (symbolisk adress `IllegalOp`) på RESET-vektorns plats, adress `FF16` med hjälp av assemblerdirektiv enligt följande exempel.

```
; Symbolfält      Mnemonic/  Operand
; direktiv
HEXDISPLAY:      EOU          $FB
;
;                ORG          $20
;                ...          som tidigare
;
IllOpService:    ...          som tidigare
;
;                ORG          $FF
;                FCB          IllegalOp
```

- Assemblera programmet och rätta eventuella fel.
- Nollställ simulatörens minnesinnehåll (Clear memory).



- Ladda programmet till simulatören.



- Återställ FLISP ("RESET")



- Kontrollera att programräknaren PC nu innehåller programmets startadress.