EDA216

Digital- och datorteknik

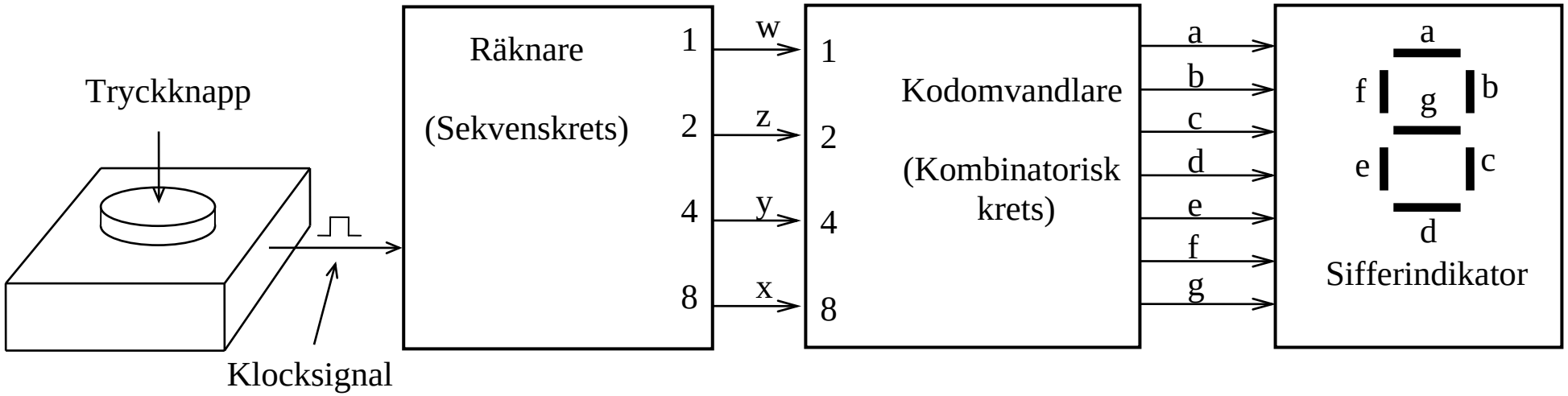
Diverse kompletterande material

Diverse kompletterande material (pdf):

- Introbild med sifferindikator
- Positionssystem.
- Samband mellan binära och hexadecimal tal.
- Exempel på excess 8-kod.
- Grafiska symboler för de grundläggande logikoperationerna. (Tabell 3.11)
- Viktiga satser i boolesk algebra.
- Booleska räkneregler och motsvarande grindnät.
- Övningsexempel på minimering med karnaughdiagram.
- NAND-, NOR- och XOR-grindar.
- Uppgift 4.14 (Don't care ex).
- Kodomvandlare för en digital vinkelgivare.
- Uppgifter för Demo1.
- Två olika kodomvandlare.
- Väljare och fördelare.
- Princip för realisering av funktion med väljare.
- Realisering av funktion med väljare, exempel.
- Representation av heltal med ett begränsat antal sifferpositioner.
- Decimal addition för heltal utan tecken.
- Decimal subtraktion för heltal utan tecken (och 10-komplementaritmetik).
- Binär addition för heltal utan tecken.
- Binär subtraktion för heltal utan tecken (och 2-komplementaritmetik).
- 10-komplementaritmetik och 10-komplementrepresentation för heltal med tecken.
- 2-komplementaritmetik och 2-komplementrepresentation för heltal med tecken.
- Flaggor som används vid aritmetiska operationer.
- Logikkrets för addition av binära tal (Heladderare).
- Uppg 4.26. Lösning.
- Uppg 6.5a,c,f; 6.8c,d,i,j.
- Krets för subtraktion.
- Aritmetik- logikenhet (ALU) för FLEX.
- Latchar. SR-latchen. Klockad (grindad) SR-latch. D-latch.
- Vippor. SR-, D-, JK- och T-vippa.
- Exempel på D-vippa med grindad laddingång.
- Uppg 5.1; 5.3; 5.7; 5.8.
- 5.3 Register.
- Möjligheter att koppla ihop logikkretsars utgångar.
- Register i dataväg.
- Systemexempel.
- Uppg 7.3a,e och 7.4a.
- Läs- och skrivbart minne.

- 5.4 Räknare.
- Uppg 5.13; 5.14.
- Uppg X.1.
- Excitationstabeller för SR-, D-, JK- och T-vippan.
- Uppg X.5.
- Tentauppg syntes.
- Användning av T-vippor vid konstruktion av räknare.
- Konstruktion av sekvensierare för FLEX- eller FLIS-processorn.
- Datorn enligt von Neumann.
- FLEX-datorn, detaljbild.
- FLEX- eller FLIS-datorn, utifrån sett.
- FLEX-datorn, programmerarens bild.
- FLEX-datorn, detaljbild.
- FLIS-datorn, programmerarens bild.
- Maskinprogram och assemblerprogram.
- Uppgifter Ext-8.
- FLIS-dator med I/O-portar.
- Beräkning av exekveringstid, uppgift F.14a-d.
- FLEX. Fast kopplad styrenhet, kopplingsblankett.
- FLIS-datorn, detaljbild.
- FLIS-processorns ALU.
- Den automatiska styrenheten (FLISP).
- Implementering av branch-instruktioner.
- Assemblatordirektiv (pseudoinstruktioner).
- Simulatorns I/O-enheter.
- Enkel styrning av borrmaskin.
- Uppgift X.17.
- Ext-10 (listfil, pdf) (Subrutinexempel.)
- Styrstrukturer i C.
- Hur FLIS-processorn läser och skriver i minnet.
- Krets för detektering av läsning på en viss minnesadress.
- Yttre signaler som påverkar exekveringen hos FLIS-processorn.
- Något om hur processorn CPU12 läser och skriver i minnet.
- Adressrum 16 bitar.
- Principen för anslutning av minnesmoduler och portar till CPU12.
- Exempel på fullständig och ofullständig adressavkodning.
- Uppg X.11.
- Flyttal komplettering.
- Principen för multiplikation och division av binära heltal utan tecken.
- Multiplikation genom upprepad addition eller genom addition och skift.
- Division genom upprepad subtraktion eller genom subtraktion och skift.

Introbild med sekvensnät + kombinatoriskt nät



Talsystem

Positionssystem

Vårt vanliga decimala talsystem är ett positionssystem. Siffrornas vikt bestäms av deras position i talet.

Ex. Talet 5768,29

Position: 3 2 1 0 -1 -2

Talvärde: $5\ 7\ 6\ 8,\ 2\ \ 9 = 5\cdot10^{3} + 7\cdot10^{2} + 6\cdot10^{1} + 8\cdot10^{0} + 2\cdot10^{-1} + 9\cdot10^{-2}$

Ett positionssystem har en bas. Basen är förhållandet (kvoten) mellan vikten hos den vänstra och den högra av två närliggande sifferpositioner. Basen är också lika med antalet olika siffersymboler (siffror) i talsystemet.

Siffersymbolerna är normalt de som används i det decimala talsystemet dvs. 0, 1, 2, …, basen-1. Om de decimala siffrorna inte räcker till kompletterar man normalt med stora bokstäver från alfabetets början.

Decimala talsystemet har t ex basen 10, som också är antalet olika siffersymboler.

Man kan göra positionssystem för alla heltalsbaser som är större än eller lika med 2.

Vanliga talsystem:

Bas	Talsystem	Siffror
2	Binära	0, 1
(8	Oktala	0, 1, 2, …, 7)
10	Decimala	0, 1, 2, …, 9
16	Hexadecimala	0, 1, 2, …, 9, A, B, C, D, E, F

Positionssystemen fungerar som vägmätaren i en gammal bil. Basen för det aktuella positionssystemet är antalet siffror på varje sifferhjul.

Då bilen körs så vrids sifferhjulet längst till höger fram en siffra för varje kilometer. När siffran 0 kommer fram en gång per varv på ett sifferhjul så vrids sifferhjulet till vänster fram en siffra. Samma princip gäller för alla sifferhjul i vägmätaren.

Ex.

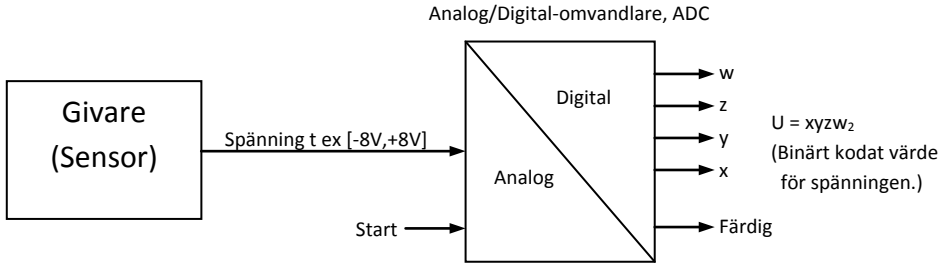
För en decimal vägmätare med 5 siffror är lägsta värdet givetvis 00000 och det högsta värdet 99999 = 10^5-1.
Talintervallet som kan visas är alltså $[0, 99999_{10}] = [0, 10^5\text{-}1]$.
Efter det högsta värdet varvar vägmätaren, dvs. den börjar om på 0. Antalet olika tal som vägmätaren kan visa är 10^5 = 100000. Man säger att den visar vägsträckan modulo 10^5.

För en binär vägmätare med 8 siffror är lägsta värdet 00000000 och det högsta värdet $11111111_2 = 2^8\text{-}1 = 255_{10}$.
Talintervallet som kan visas är alltså $[0, 255_{10}] = [0, 2^8\text{–}1]$.
Efter det högsta värdet varvar vägmätaren, dvs. den börjar om på 0. Antalet olika tal som vägmätaren kan visa är 2^8 = 256. Vägsträckan visas modulo 2^8.

Illustration av sambandet mellan binära och hexadecimala tal

Bin	Hex	Dec
00 0000	0 0	0
00 0001	0 1	1
00 0010	0 2	2
00 0011	0 3	3
00 0100	0 4	4
00 0101	0 5	5
00 0110	0 6	6
00 0111	0 7	7
00 1000	0 8	8
00 1001	0 9	9
00 1010	0 A	10
00 1011	0 B	11
00 1100	0 C	12
00 1101	0 D	13
00 1110	0 E	14
00 1111	0 F	15
01 0000	1 0	16
01 0001	1 1	17
01 0010	1 2	18
01 0011	1 3	19
01 0100	1 4	20
01 0101	1 5	21
01 0110	1 6	22
01 0111	1 7	23
01 1000	1 8	24
01 1001	1 9	25
01 1010	1 A	26
01 1011	1 B	27
01 1100	1 C	28
01 1101	1 D	29
01 1110	1 E	30
01 1111	1 F	31
10 0000	2 0	32
10 0001	2 1	33
10 0010	2 2	34
10 0011	2 3	35
10 0100	2 4	36
10 0101	2 5	37
10 0110	2 6	38
10 0111	2 7	39
10 1000	2 8	40
10 1001	2 9	41
10 1010	2 A	42
10 1011	2 B	43
10 1100	2 C	44
10 1101	2 D	45
10 1110	2 E	46
10 1111	2 F	47
11 0000	3 0	48

Exempel på excess 8-kod (excess 2^{n-1}-kod, med n = 4)



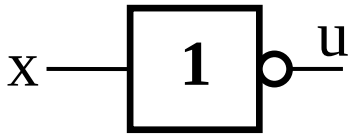
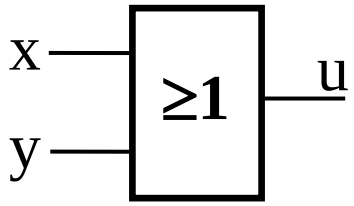
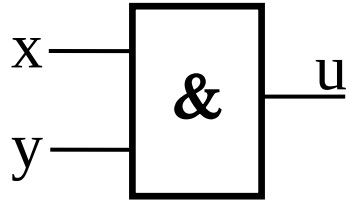
Koden decimalt:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Koden binärt:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Spänningsvärde:	-8V	-7V	-6V	-5V	-4V	-3V	-2V	-1V	0V	+1V	+2V	+3V	+4V	+5V	+6V	+7V

U

+8V

Man får här spänningen i volt genom att subtrahera 8 från kodordets värde (excess 8).

Lägg märke till att spänningsvärdet +8 V inte får något kodord för denna kod.

Grind (Gate)	Symbol	Funktions-tabell	Booleskt uttryck
INVERTERARE (ICKE, NOT)		x | u 0 | 1 1 | 0	u = x’
ELLER (OR)		x | y | u 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1	u = x + y
OCH (AND)		x | y | u 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1	u = x · y

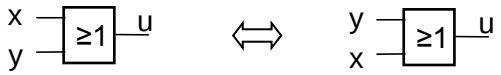
Tabell 3.11 kompletterad (Grafiska symboler för de grundläggande logikoperationerna.)

Några viktiga satser inom Boolesk algebra.

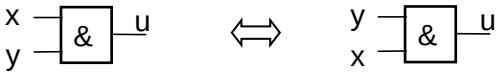
1.	$x + y = y + x$ $x \cdot y = y \cdot x$	**Kommutativa lagarna**
2.	$x \cdot (y + z) = x \cdot y + x \cdot z$ $x + (y \cdot z) = (x + y) \cdot (x + z)$	**Distributiva lagarna**
3.	$x + 0 = x$ $x \cdot 1 = x$	
4.	$x + x' = 1$ $x \cdot x' = 0$	
5.	$x + 1 = 1$ $x \cdot 0 = 0$	
6.	$x + x = x$ $x \cdot x = x$	
7.	$x + (y + z) = (x + y) + z$ $x \cdot (y \cdot z) = (x \cdot y) \cdot z$	**Associativa lagarna**
8.	$(x + y)' = x' \cdot y'$ $(x \cdot y)' = x' + y'$	**De Morgans lagar**
9.	$(x')' = x$	

Viktiga satser i switchnätalgebra och motsvarande grindnät

1. $x + y = y + x$

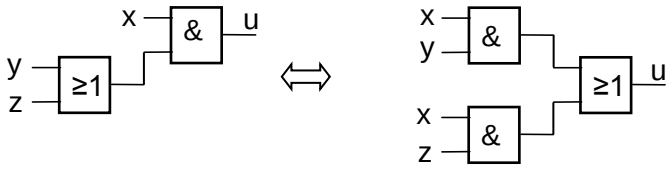


$x \cdot y = y \cdot x$

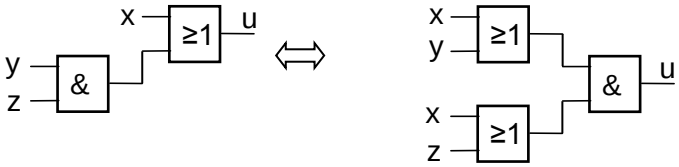


---

2. $x \cdot (y + z) = x \cdot y + x \cdot z$

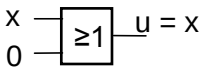


$x + (y \cdot z) = (x + y) \cdot (x + z)$

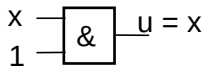


---

3. $x + 0 = x$

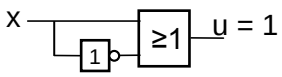


$x \cdot 1 = x$

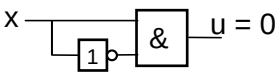


---

4. $x + x' = 1$

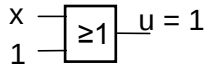


$x \cdot x' = 0$

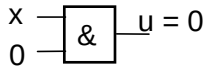


---

5. $x + 1 = 1$

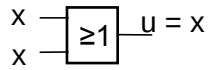


$x \cdot 0 = 0$

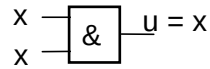


---

6. $x + x = x$

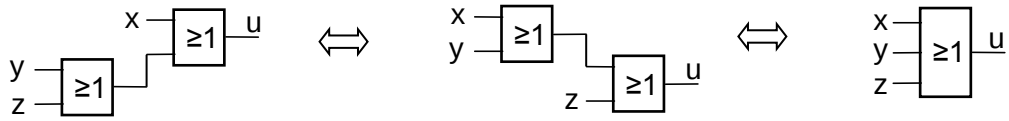


$x \cdot x = x$

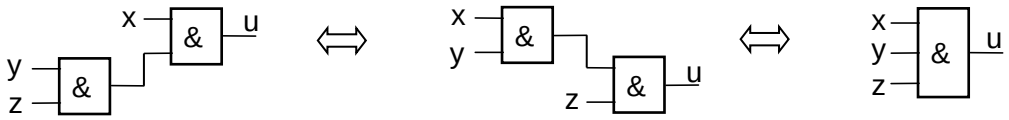


---

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

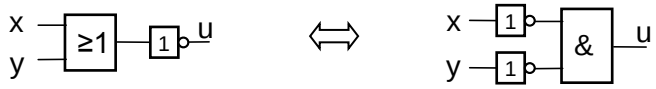


$x \cdot (y \cdot z)$ = $(x \cdot y) \cdot z$ = $x \cdot y \cdot z$

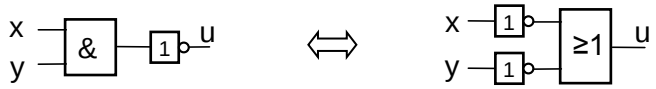


---

8. $(x + y)'$ = $x' \cdot y'$

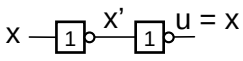


$(x \cdot y)'$ = $x' + y'$



---

9. $(x')' = x$



Övningsexempel på karnaughdiagram

Tag fram minimala SP-uttryck ur karnaughdiagrammen nedan! (Ettorna!)

f_1	zw 00	01	11	10
xy 00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

$f_1 =$

f_2	zw 00	01	11	10
xy 00	1	1	1	1
01	0	1	1	0
11	0	0	0	0
10	0	0	0	0

$f_2 =$

f_3	zw 00	01	11	10
xy 00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

$f_3 =$

f_4	zw 00	01	11	10
xy 00	0	1	1	0
01	0	1	1	1
11	0	1	1	1
10	0	0	0	0

$f_4 =$

f_5	zw 00	01	11	10
xy 00	1	1	1	1
01	1	1	1	1
11	0	0	0	0
10	0	1	1	0

$f_5 =$

f_6	zw 00	01	11	10
xy 00	1	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

$f_6 =$

f_7	zw 00	01	11	10
xy 00	1	0	0	1
01	1	1	1	1
11	0	1	1	0
10	1	0	0	1

$f_7 =$

f_8	zw 00	01	11	10
xy 00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

$f_8 =$

f_9	zw 00	01	11	10
xy 00	1	0	0	1
01	0	1	0	0
11	0	0	1	0
10	1	0	0	1

$f_9 =$

Svar:

$f_1 = y'w' + yw$ $f_2 = x'y' + x'w$ $f_3 = y + w'$

$f_4 = x'w + yw + yz$ $f_5 = x' + y'w$ $f_6 = x' + y + w'$

$f_7 = y'w' + yw + x'y$ $f_8 = y'w' + yw + xw$ $f_9 = y'w' + x'yz'w + xyzw$

alternativt

$f_7 = y'w' + yw + x'w'$ $f_8 = y'w' + yw + xy'$

Tag fram minimala PS-uttryck ur karnaughdiagrammen nedan! (Nollorna!)

f_{10}	zw 00	01	11	10
xy 00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

f_{10} =

f_{11}	zw 00	01	11	10
xy 00	1	1	1	1
01	0	1	1	0
11	0	0	0	0
10	0	0	0	0

f_{11} =

f_{12}	zw 00	01	11	10
xy 00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

f_{12} =

f_{13}	zw 00	01	11	10
xy 00	0	1	1	0
01	0	1	1	1
11	0	1	1	1
10	0	0	0	0

f_{13} =

f_{14}	zw 00	01	11	10
xy 00	1	1	1	1
01	1	1	1	1
11	0	0	0	0
10	0	1	1	0

f_{14} =

f_{15}	zw 00	01	11	10
xy 00	1	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

f_{15} =

f_{16}	zw 00	01	11	10
xy 00	1	0	0	1
01	1	1	1	1
11	0	1	1	0
10	1	0	0	1

f_{16} =

f_{17}	zw 00	01	11	10
xy 00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

f_{17} =

f_{18}	zw 00	01	11	10
xy 00	1	0	0	1
01	0	1	0	0
11	0	0	1	0
10	1	0	0	1

f_{18} =

Svar:

$f_{10} = (y' + w)(y + w')$ $f_{11} = x'(y' + w)$ $f_{12} = y + w'$

$f_{13} = (y + w)(z + w)(x' + y)$ $f_{14} = (x' + y')(x' + w)$ $f_{15} = x' + y + w'$

$f_{16} = (y + w')(x' + y' + w)$ $f_{17} = (y' + w)(x + y + w')$

$f_{18} = (y' + w)(y + w')(x' + y' + z)(x + y' + z')$ eller

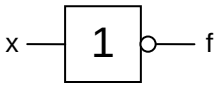
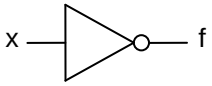
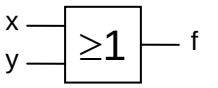
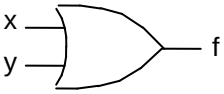
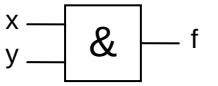
$f_{18} = (y' + w)(y + w')(x' + y' + z)(x + z' + w')$ eller

$f_{18} = (y' + w)(y + w')(x' + z + w')(x + y' + z')$ eller

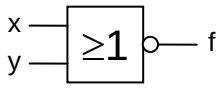
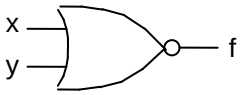
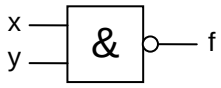
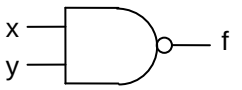
$f_{18} = (y' + w)(y + w')(x' + z + w')(x + z' + w')$

Grindsymboler och logikfunktioner

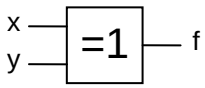
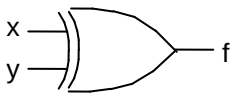
Tabell 3.11 Grafiska symboler för de grundläggande logikoperationerna.

Funktion	*Grind*	*IEC-symbol*	*(Amerikansk symbol)*
f = x'	INVERTERARE (NOT)		
f = x+y	ELLER (OR)		
f = x·y	OCH (AND)		

Tabell 3.12 Grafiska symboler för logiksambanden NOR och NAND.

Funktion	*Grind*	*IEC-symbol*	*(Amerikansk symbol)*
f = (x+y)'	NOR		
f = (x·y)'	NAND		

Tabell 3.14 Grafiska symboler för XOR-grinden.

Funktion	*Grind*	*IEC-symbol*	*(Amerikansk symbol)*
f = x ⊕ y	XOR		

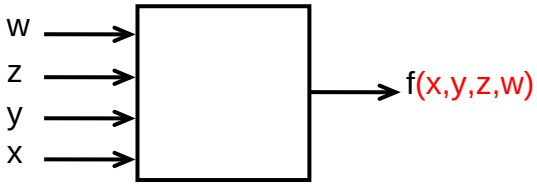
Tabell 3.13

x	*y*	*ELLER f = x+y*	*EXKLUDERANDE ELLER (XOR) f = x'y+xy'*
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

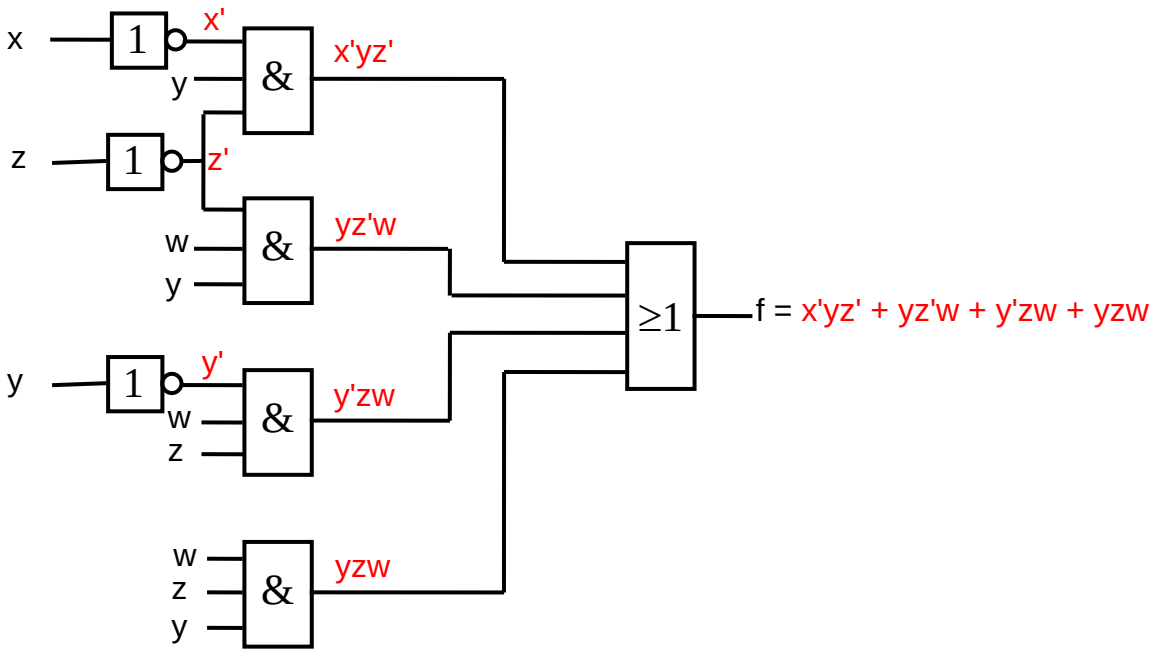
Ext-4 (Ver 2013-09-09)

Praktikfall, minimering av grindnät

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



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



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

Ext-4 (Ver 2013-09-09)

f = x'yz' + yz'w + y'zw + yzw

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

Realisering:

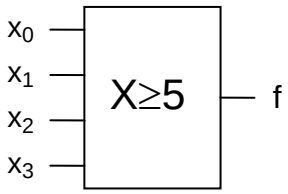
Minimering:

f	zw			
	00	01	11	10
00	0	0	1	0
01	1	1	1	0
xy 11	0	1	1	0
10	0	0	1	0

f =

Introduktionsexempel med "don't care"-termer

Uppg 4.14 (Exempel på don't care.)

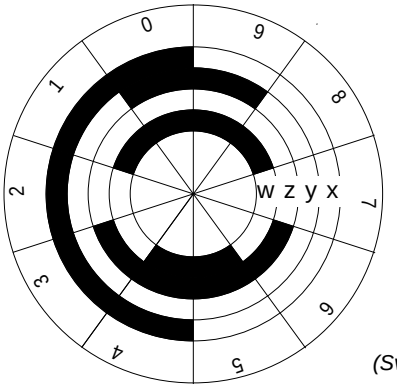


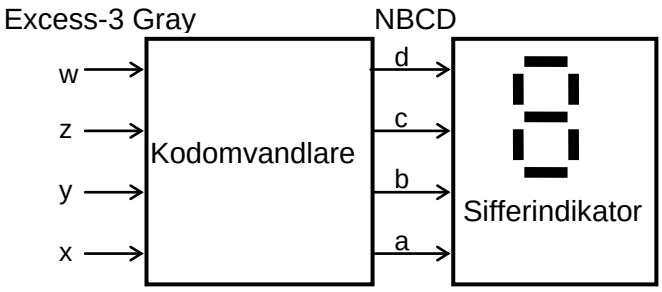
Figuren gäller för ett kombinatoriskt nät som har utsignalen f = 1 om den NBCD-kodade siffran $X = (x_3x_2x_1x_0)_{NBCD}$ är större än eller lika med fem ($X \geq 5$).

För alla andra NBCD-kodade siffror skall gälla att f = 0.
Tal som inte tillhör NBCD-koden skall betraktas som "don't care".

Konstruera ett minimalt kombinatoriskt nät med NAND-grindar och INVERTERARE.

Komplettering av uppgift 4.2 i arbetsboken.





(Svart fält på kodskivan motsvarar nolla.)

Decimal siffra Z	***NBCD abcd***	***Excess-3-Graykod xyzw***
0	0000	0010
1	0001	0110
2	0010	0111
3	0011	0101
4	0100	0100
5	0101	1100
6	0110	1101
7	0111	1111
8	1000	1110
9	1001	1010

Excess-3-Gray				NBCD			
x	y	z	w	a	b	c	d
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				

Övningsuppgifter (ur KMP) (Demo)

2.1 Omvandla till decimal form

a) 101101_2 **f)** 101.101_2

2.3 Omvandla till decimal form

a) $2C3_{16}$

2.5 Omvandla till hexadecimal form

a) 10101010_2

2.6 Omvandla till binär form (naturlig binär kod).

a) 23_{10}

2.13 Skriv $563{,}782_{10}$ på NBCD-kod.

3.4 I denna uppgift betecknar x, y och z booleska variabler. Bevisa genom fullständig binär evaluering, att

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

3.5 Åskådliggör med hjälp av grindsymboler

a) $z\cdot(x'+y)$

3.13 Skriv nedanstående booleska funktion på disjunktiv normal form.

d) $f = x\cdot(y'+ z)$

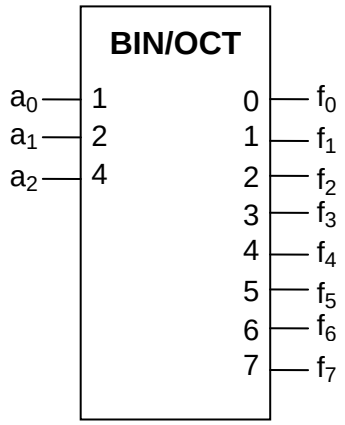
3.14 Skriv nedanstående booleska funktion på konjunktiv normal form.

c) $f = x\cdot(y'+ z)$

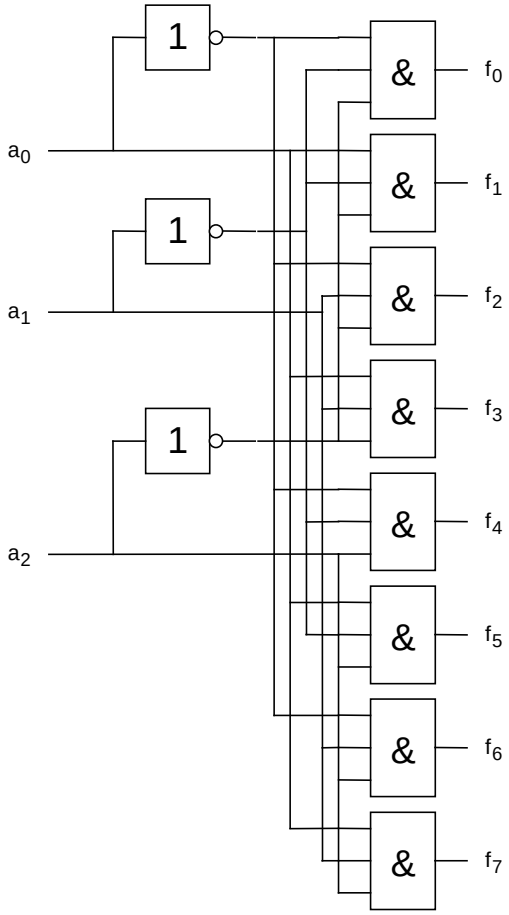
4.15 Konstruera ett minimalt kombinatoriskt nät som omvandlar den NBCD-kodade siffran $X = (x_3x_2x_1x_0)_{NBCD}$ till sitt 9-komplement $Y = 9\text{-}X = (y_3y_2y_1y_0)_{NBCD}$. Nätet skall således ha fyra in- och fyra utsignaler. Tal som inte tillhör NBCD-koden kommer inte att uppträda. Vid realiseringen får INVERTERARE, samt NAND- och XOR-grindar användas. Realiseringen skall ha så få grindar som möjligt.

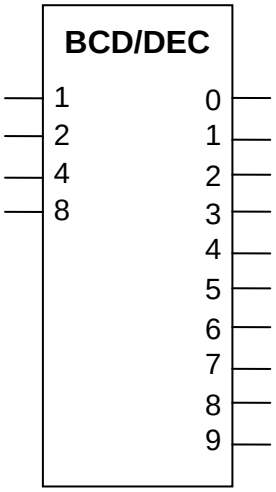
4.25 Realisera funktionen f = b'd' + bd + acd med hjälp av en ”1 av 8”-väljare. INVERTERARE får också användas.

Kodomvandlare



Figur 4.10 Prosamsymbol för "NBC till en av åtta" ("Binary/Octal") kodomvandlare.

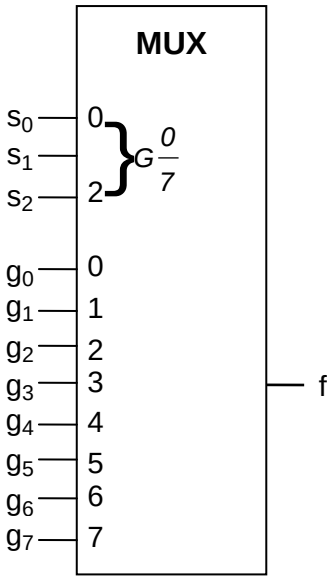




Figur 4.12 Prosamsymbol för "NBCD till en av tio" (NBCD/Decimal) kodomvandlare.

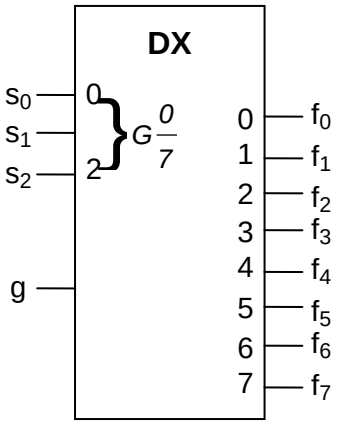
a_3	a_2	a_1	a_0	Dec	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
0	0	1	0	2	0	0	1	0	0	0	0	0	0	0
0	0	1	1	3	0	0	0	1	0	0	0	0	0	0
0	1	0	0	4	0	0	0	0	1	0	0	0	0	0
0	1	0	1	5	0	0	0	0	0	1	0	0	0	0
0	1	1	0	6	0	0	0	0	0	0	1	0	0	0
0	1	1	1	7	0	0	0	0	0	0	0	1	0	0
1	0	0	0	8	0	0	0	0	0	0	0	0	1	0
1	0	0	1	9	0	0	0	0	0	0	0	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	-	?	?	?	?	?	?	?	?	?	?

Väljare och fördelare



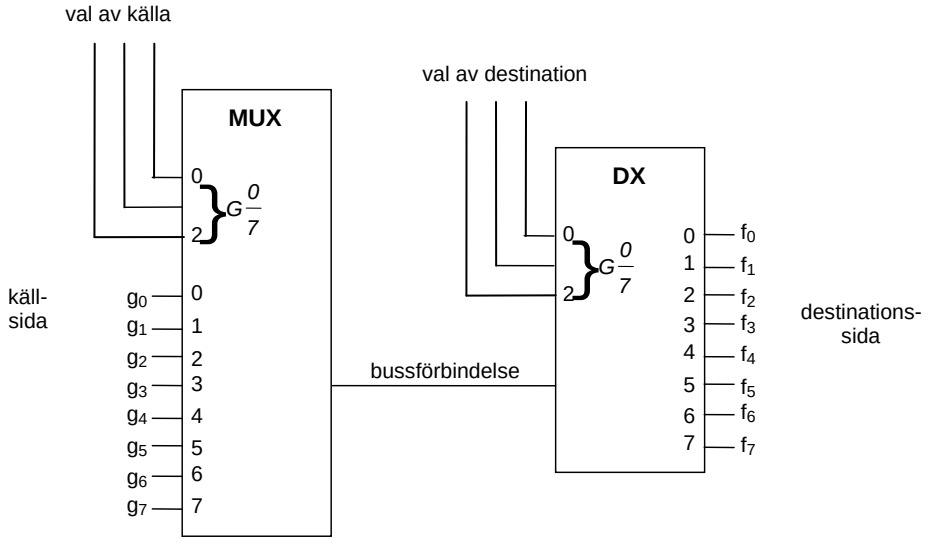
s_2	s_1	s_0	f
0	0	0	g_0
0	0	1	g_1
0	1	0	g_2
0	1	1	g_3
1	0	0	g_4
1	0	1	g_5
1	1	0	g_6
1	1	1	g_7

Figur 4.1 Symbol och funktionstabell för 1 av 8 väljare.



s_2	s_1	s_0	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
0	0	0	g	0	0	0	0	0	0	0
0	0	1	0	g	0	0	0	0	0	0
0	1	0	0	0	g	0	0	0	0	0
0	1	1	0	0	0	g	0	0	0	0
1	0	0	0	0	0	0	g	0	0	0
1	0	1	0	0	0	0	0	g	0	0
1	1	0	0	0	0	0	0	0	g	0
1	1	1	0	0	0	0	0	0	0	g

Figur 4.2 Symbol och funktionstabell för en "1 till 8 fördelare".



Figur 4.3 Princip för tidsmultiplex med hjälp av väljare och fördelare

Realisering av booleska funktioner med hjälp av väljare.

Nedan visas funktionstabellen för en godtycklig boolesk funktion ***f*** av fyra variabler .

Vi observerar att värdena för x, y och z är lika parvis uppifrån och ner i hela tabellen. Radparet med ***x = 0***, ***y = 1*** och ***z = 0*** är markerat. Till höger om detta radpar visas samtliga fyra möjliga kombinationer av ***f***-värden för de två raderna.

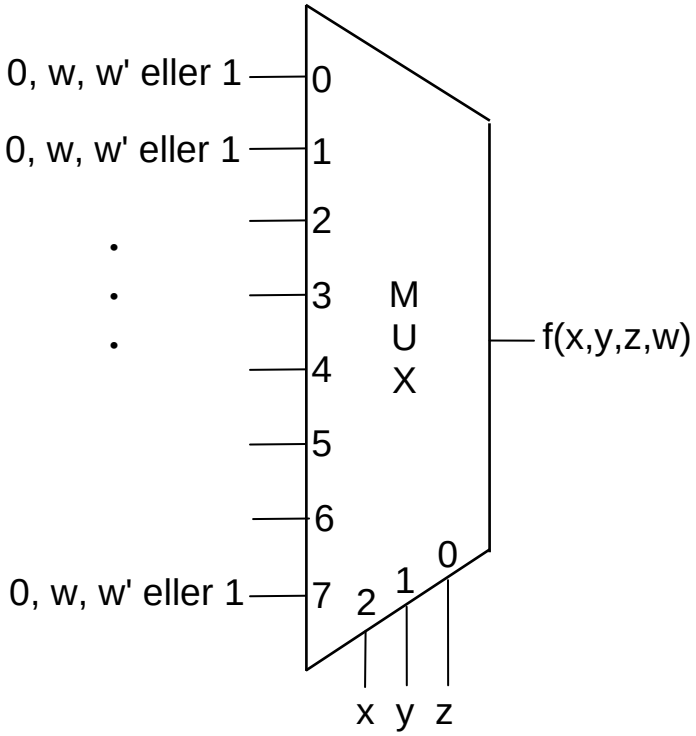
Vi konstaterar att ***f***-värdet kan vara ***0***, ***w***, ***w'*** eller ***1*** för radparet och att samma sak gäller samtliga radpar i tabellen.

Om man kopplar in variablerna x, y och z till selektoringångarna på en väljare med 8 (2^3) dataingångar så kommer varje kombination av x, y och z att motsvara ett radpar och därmed en egen dataingång på väljaren.

Utsignalen från väljaren blir därför samma som värdet på den dataingång som motsvarar det inställda värdet på x, y och z. Genom att koppla in något av värdena 0, w, w' eller 1till varje dataingång kan man därför realisera varje funktion ***f*** av fyra variabler med hjälp av en väljare med tre selektoringångar.

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

0	***1***	***2***	***3***
0	0	1	1
0	1	0	1



Ett exempel på användning av väljare

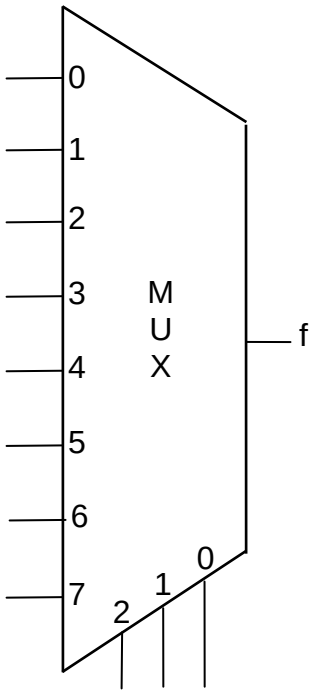
Man kan realisera vilken som helst av samtliga möjliga booleska funktioner av n variabler genom att använda en "1 av 2^{n-1} väljare" och eventuellt en NOT-grind.

Alltså kan en "1 av 8 väljare" tillsammans med en NOT-grind användas för att realisera vilken som helst av de 2^{16} = 65536 olika booleska funktionerna av fyra variabler.

Uppgift: Använd en "1 av 8 väljare" för att realisera den booleska funktionen f(x,y,z,w) som definieras av funktionstabellen nedan.

Skriv in nedan till höger hur variablerna kopplas till väljarens styringångar. Skriv också in vilka värden som kopplas in på väljarens dataingångar.

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



Representation av heltal med ett begränsat antal sifferpositioner

Med ett begränsat antal sifferpositioner kan man bara representera heltal korrekt inom ett begränsat talområde. Man kan dock representera större heltal genom att helt enkelt stryka de siffror till vänster som inte ”får plats”, men man får då en osäkerhet (ett fel) som beror på värdet av de strukna siffrorna.

Ex.
Med fyra decimala siffror kan man representera heltal i intervallet [0, 9999] = [0, 10^4-1] korrekt. Med fyra decimala sifferpositioner säger man att talen representeras modulo 10^4. Det innebär att alla tal som är större än $10^4 – 1 = 9999$ kommer att representeras med en osäkerhet (fel) i form av termen $n \cdot 10^4$, där n är ett heltal (n = 1, 2, 3, ...).

Talet $37680_{10} = 3 \cdot 10^4 + 7680$ representeras med fyra siffror av talet 7680 och felet är $3 \cdot 10^4$.

Ex.
Med åtta binära siffror (bitar) kan man representera heltal i intervallet [0, 11111111_2] = [0, 2^8-1] = = [0, 255_{10}] korrekt. Med åtta bitpositioner säger man att talen representeras modulo 2^8. Det innebär att alla tal som är större än $2^8 – 1 = 255_{10}$ kommer att representeras med en osäkerhet (fel) i form av termen $n \cdot 2^8$, där n är ett heltal (n = 1, 2, ...).

Talet $943_{10} = 1110101111_2 = 11_2 \cdot 2^8 + 10101111_2 = 3 \cdot 2^8 + 175_{10}$ representeras alltså med åtta bitar av talet $10101111_2 = 175_{10}$ och felet är $3 \cdot 2^8 = 3 \cdot 256_{10} = 768_{10}$.

Decimal addition för heltal utan tecken, S = X + Y

Exempel med två heltal X och Y med fyra sifferpositioner:

X = 7396

Y = 5842

```
  4  3  2  1  0     sifferposition
  1  1  1  0  0     minnessiffra (carry)
     7  3  9  6      X
   +5  8  4  2    +Y
  1  3  2  3  8  = S = 13238 = 10000 + 3238
```

Additionen ger en summa som kräver 5 siffror. ”***Overflow***” (spill) för tal utan tecken har inträffat eftersom den femte siffran $c_4 = 1$.

”***Overflow***” vid addition av heltal utan tecken innebär att talområdets övre gräns har passerats.

Den femte siffran c_4 har vikten $10^4 = 10000$, så värdet med endast 4 sifferpositioner blir 3238.

Summan av två heltal med vardera 4 siffror ryms alltid i 5 sifferpositioner.

Talområdet för 4-siffriga heltal utan tecken är $[0, 10^4 - 1] = [0, 9999]$.

Decimal subtraktion för heltal utan tecken, D = X – Y

Exempel med två heltal X och Y med fyra sifferpositioner:

X = 7396

Y = 5842

```
 4   3   2   1   0      sifferposition
 0   0  10   0   0      lån (borrow)
     7   3   9   6    X
   - 5   8   4   2    - Y
     1   5   5   4  = D
```

”***Overflow***” vid subtraktion av heltal utan tecken inträffar om det krävs ett lån vid subtraktionen av siffrorna längst till vänster. Det innebär att resultatet skulle ha blivit negativt, dvs. utanför talområdet. Negativa tal finns ju inte för tal utan tecken.

Man kan också utföra subtraktionen med ***10-komplementaritmetik*** som visas nedan.

Byt ut X – Y mot $10^4 + X - Y$.

$10^4 + X - Y = 10000 + X - Y = 9999 + 1 + X - Y = X + (9999 - Y) + 1 = X + Y_{9k} + 1$

$9999 - Y = Y_{9k}$ som kallas (siffervis) 9-komplement av Y.
Man bildar Y_{9k} genom att byta ut alla siffror i Y mot $9 - y_i$ (9 – siffran).

Med samma värden på X och Y som ovan får vi:
$Y_{9k} = 4157$

```
 4  3  2  1  0        sifferposition
 1  0  1  1  1     1  minnessiffra (carry)
    7  3  9  6     X
  + 4  1  5  7   + Y9k
 ~~1~~ 1  5  5  4  = D = ~~10000~~ + 1554
```

Lägg märke till att vi har satt minnessiffran längst till höger till 1 ($c_0 = 1$) och att minnessiffran ut $c_4 = 1$. c_4 har ju vikten $10^4 = 10000$, som vi lade till från början. Vi får alltså den korrekta skillnaden genom att stryka c_4.

Overflow vid subtraktion med 10-komplementaritmetik upptäcks genom att minnessiffran som bildas (c_4 i exemplet ovan) vid additionen av siffrorna längst till vänster blir 0. Eftersom $c_4 = 1$ i exemplet blir det inte *overflow* i detta fall.

Binär addition för heltal utan tecken, S = X + Y

Exempel med två 8-bitars heltal X och Y:

$X = 10101101_2 = 173_{10}$

$Y = 01100110_2 = 102_{10}$

```
  8 7 6 5 4 3 2 1 0  bitposition
  1 1 1 0 1 1 0 0    minnessiffra (carry)
    1 0 1 0 1 1 0 1    X
   +0 1 1 0 0 1 1 0   +Y
  ---------------------
  1 0 0 0 1 0 0 1 1 = S = 275(10)= 256(10) + 19(10)
```

Additionen ger en summa som kräver 9 bitar. ”***Overflow***” (spill) för 8-bitars heltal utan tecken har inträffat eftersom den nionde biten $c_8 = 1$.

”***Overflow***” vid addition av heltal utan tecken innebär att talområdets övre gräns har passerats.

Den nionde biten c_8 har vikten $2^8 = 256_{10}$, så värdet med endast 8 bitar blir 19_{10}.

Summan av två 8-bitars heltal ryms alltid i 9 bitar.

(Summan av två n-bitars heltal ryms alltid i n + 1 bitar.)

Talområdet för 8-bitars heltal utan tecken är $[0, 2^8 – 1] = [0, 256_{10} – 1] = [0, 255_{10}]$.

(Talområdet för n-bitars heltal utan tecken är $[0, 2^n – 1]$.)

Binär subtraktion för heltal utan tecken, D = X – Y

Exempel med två 8-bitars heltal X och Y:

$X = 10101101_2 = 173_{10}$

$Y = 01100110_2 = 102_{10}$

```
 8 7 6 5 4 3 2 1 0     bitposition
 0 0 2 0 0 0 2 2 0     lån (borrow)
   1 0 1 0 1 1 0 1     X
  -0 1 1 0 0 1 1 0    -Y
   0 1 0 0 0 1 1 1 = D = 71(10)
```

"***Overflow***" vid subtraktion av heltal utan tecken inträffar om det krävs ett lån vid subtraktionen av siffrorna längst till vänster. Det innebär att resultatet skulle ha blivit negativt, dvs. utanför talområdet. Negativa tal finns ju inte för tal utan tecken.

Addition och subtraktion utförs olika och kräver därmed olika grindnät (= kretsar).

Vi använder istället ***2-komplementaritmetik***, på samma sätt som vi tidigare använde ***10-komplementaritmetik*** och kan då utföra subtraktionen som en addition.

Byt ut X – Y mot $2^n + X – Y$, där n är antalet bitar vi arbetar med.

Exempel med n = 8:

$2^8 + X – Y = 256 + X – Y = 255 + 1 + X – Y = X + (11111111_2 – Y) + 1 = X + Y_{1k} + 1$

$11111111_2 – Y = Y_{1k}$ som kallas (siffervis) 1-komplement av Y.
Man bildar Y_{1k} genom att invertera alla bitar i Y.

Med samma värden på X och Y som ovan får vi:

```
 8 7 6 5 4 3 2 1 0       bitposition
 1 0 1 1 1 0 0 1 1   1   minnessiffra (carry)
   1 0 1 0 1 1 0 1   X
  +1 0 0 1 1 0 0 1  +Y1k
 1 0 1 0 0 0 1 1 1 = D = 71(10)
```

Lägg märke till att vi har satt minnessiffran längst till höger till 1 ($c_0 = 1$) och att minnessiffran ut $c_8 = 1$. c_8 har ju vikten $2^8 = 256$, som vi lade till från början.
Vi får den korrekta skillnaden genom att stryka c_8.

Overflow vid subtraktion med 2-komplementaritmetik upptäcks genom att minnes-siffran (c_8 i exemplet ovan) vid additionen av siffrorna längst till vänster blir 0. Eftersom $c_8 = 1$ i exemplet blir det inte *overflow* i detta fall.

Decimal aritmetik för heltal med tecken

10-komplementrepresentation av decimala heltal med tecken

Först två definitioner:
De tal som används för att representera verkliga tal i en dators register eller minne kallas ofta ***maskintal***.

Operationen $10^n - X = (99\ldots9 - X) + 1 = X_{9k} + 1$ kallas ***10-komplementering av X***
och resultatet blir "***10-komplementet av X***", som skrivs $\boldsymbol{X_{10k}}$.

Ex.
Antag att vi har tre positiva decimala heltal A, B och C med vardera 4 siffror.
A = 7652, B = 2153 och C = 4789

Vi bildar *maskintalen* för A, B och C genom att lägga till en teckensiffra 0 längst till vänster i talen för att tala om att de är positiva: A = 07652, B = 02153 och C = 04789

Som *maskintal* för de negativa talen –A, –B och –C används 10-komplementen av *maskintalen* de positiva talen A, B och C.

$A_{10k} = 10^5 - A = (99999 - A) + 1 = A_{9k} + 1$ (Detta tal är alltså maskintalet för –A.)
$B_{10k} = 10^5 - B = (99999 - B) + 1 = B_{9k} + 1$ (Detta tal är alltså maskintalet för –B.)
$C_{10k} = 10^5 - C = (99999 - C) + 1 = C_{9k} + 1$ (Detta tal är alltså maskintalet för –C.)

10-komplementering av *maskintalet* innebär byte av tecken från positivt till negativt eller tvärt om.

Talet – A = – 7652 motsvaras av *maskintalet* A_{10k} = 99999 – 07652 + 1 = 92347 + 1 = 92348
Talet – B = – 2153 motsvaras av *maskintalet* B_{10k} = 99999 – 02153 + 1 = 97846 + 1 = 97847
Talet – C = – 4789 motsvaras av *maskintalet* C_{10k} = 99999 – 04789 + 1 = 95210 + 1 = 95211

Maskintal för positiva tal inleds med 0 och negativa med 9.
(Man kan även tänka sig att m*askintal* för positiva tal inleds med någon av siffrorna 0-4 och *maskintal* för negativa tal med någon av siffrorna 5-9, men detta skulle komplicera identifieringen av talets tecknet.)
Med 0 och 9 som teckensiffror i 5-siffriga decimala *maskintal* blir talområdet [-10000, +9999]

10-komplementaritmetik för decimala heltal

Vi använder talen A, B och C ovan som exempel.

Addition (Alla beräkningar nedan utförs med endast 5 siffror.)
(+A) + (+B) = 07652 + 02153 = 09805 (Korrekt eftersom teckensiffrorna 0 + 0 = 0.)
(+A) + (+C) = 07652 + 04789 = 12441 (Overflow eftersom teckensiffrorna 0 + 0 = 1.)
(+A) + (–B) vilket motsvarar A + B_{10k} = 07652 + 97847 = 05499 (Korrekt, olika tecken.)
(+A) + (–C) vilket motsvarar A + C_{10k} = 07652 + 95211 = 02863 (Korrekt, olika tecken.)
(–A) + (+B) vilket motsvarar A_{10k} + B = 92348 + 02153 = 94501, dvs. – 5499 (Korrekt, olika tecken.)
(–A) + (+C) vilket motsvarar A_{10k} + C = 92348 + 04789 = 97137, dvs. – 2863 (Korrekt, olika tecken.)
(–A) + (–B) = vilket motsvarar A_{10k} + B_{10k} = 92348 + 97847 = 90195 (Korrekt, 9 + 9 = 9.)
(–A) + (–C) = vilket motsvarar A_{10k} + C_{10k} = 92348 + 95211 = 87559 (Overflow, 9 + 9 = 8.)

Subtraktion (Alla beräkningar nedan utförs med endast 5 siffror.)
(+A) – (+B) vilket motsvarar A + B_{9k} + 1 = 07652 + 97846 +1 = 05499 (Korrekt, olika tecken.)
(+A) – (+C) vilket motsvarar A + C_{9k} + 1 = 07652 + 95210 +1 = 02863 (Korrekt, olika tecken.)
(+A) – (–B) vilket motsvarar A + $(B_{10k})_{9k}$ + 1 = 07652 + $(97847)_{9k}$ + 1 = 07652 + 02152 +1 = 09805 (Korrekt, 0 + 0 = 0)
(+A) – (–C) vilket motsvarar A + $(C_{10k})_{9k}$ + 1 = 07652 + $(95211)_{9k}$ + 1 = 07652 + 04788 +1 = 12441 (Overflow, 0 + 0 = 1)
(–A) – (+B) vilket motsvarar A_{10k} + B_{9k} + 1 = = 92348 + 97846 + 1 = 90195, dvs. – 9805 (Korrekt, 9 + 9 = 9)
(–A) – (+C) vilket motsvarar A_{10k} + C_{9k} + 1 = = 92348 + 95210 + 1 = 87559 (Overflow, 9 + 9 = 8)
(–A) – (–B) vilket motsvarar A_{10k} + $(B_{10k})_{9k}$ + 1 = 92348 + $(97847)_{9k}$ + 1 = 92348 + 02152 + 1 = 94501, dvs – 5499 (Korrekt)
(–A) – (–C) vilket motsvarar A_{10k} + $(C_{10k})_{9k}$ + 1 = 92348 + $(95211)_{9k}$ + 1 = 92348 + 04788 + 1 = 97137, dvs – 2863 (Korrekt)

Binär aritmetik för heltal med tecken

2-komplementrepresentation av binära heltal med tecken

Först två definitioner:
De tal som används för att representera verkliga tal i en dators register eller minne kallas ofta ***maskintal***.

Operationen $2^n - X = (11\ldots1_2 - X) + 1 = X_{2k} + 1$ kallas ***2-komplementering av X***
och resultatet blir "***2-komplementet av X***", som skrivs $\boldsymbol{X_{2k}}$.

Ex.
Antag att vi har tre positiva binära heltal A, B och C med vardera 7 bitar.

$A = 1011001_2$ $(= 89_{10})$ $B = 0100110_2$ $(= 38_{10})$ $C = 0111001_2$ $(= 57_{10})$

Vi bildar *maskintalen* för A, B och C genom att lägga till en teckensiffra 0 längst till vänster i talen för att tala om att de är positiva: $A = 01011001_2$ $B = 00100110_2$ $C = 00111001_2$

Som *maskintal* för de negativa talen –A, –B och –C används 2-komplementen av *maskintalen* för de positiva talen A, B och C.
$A_{2k} = 2^8 - A = (11111111_2 - A) + 1 = A_{1k} + 1$ (Detta tal är alltså maskintalet för –A)
$B_{2k} = 2^8 - B = (11111111_2 - B) + 1 = B_{1k} + 1$ (Detta tal är alltså maskintalet för –B)
$C_{2k} = 2^8 - C = (11111111_2 - C) + 1 = C_{1k} + 1$ (Detta tal är alltså maskintalet för –C)

2-komplementering av *maskintalet* innebär byte av tecken från positivt till negativt eller tvärt om.

Talet $-A = -1011001_2$ motsvaras av *maskintalet* $A_{2k} = (01011001_2)_{1k} + 1 = 10100110_2 + 1 = 10100111_2$ (motsvarar -89_{10})
Talet $-B = -0100110_2$ – ” – $B_{2k} = (00100110_2)_{1k} + 1 = 11011001_2 + 1 = 11011010_2$ (” -38_{10})
Talet $-C = -0111001_2$ – ” – $C_{2k} = (00111001_2)_{1k} + 1 = 11000110_2 + 1 = 11000111_2$ (” -57_{10})

Maskintal för positiva tal inleds med 0 och negativa med 1.

Talområdet med 8-bitars maskintal: $[-10000000_2, +01111111_2] = [-128_{10}, +127_{10}]$.
Man ser att talet -128_{10} inte har någon positiv motsvarighet.

2-komplementaritmetik

Vi använder talen A, B och C ovan som exempel.

Addition (Alla beräkningar nedan utförs med 8 bitar.)
(+A) + (+B) = $\underline{0}1011001_2 + \underline{0}0100110_2 = \underline{0}1111111_2 = +127_{10}$ (Korrekt eftersom teckensiffrorna 0 + 0 = 0.)
(+A) + (+C) = $\underline{0}1011001_2 + \underline{0}0111001_2 = \underline{1}0010010_2$, motsv. -110_{10} (Overflow eftersom teckensiffrorna 0 + 0 = 1.)
(+A) + (–B) motsvaras av $A + B_{2k} = \underline{0}1011001_2 + \underline{1}1011010_2 = \underline{0}0110011_2 = +51_{10}$ (Korrekt, olika tecken.)
(+A) + (–C) – ” – $A + C_{2k} = \underline{0}1011001_2 + \underline{1}1000111_2 = \underline{0}0100000_2 = +32_{10}$ (Korrekt, olika tecken.)
(–A) + (+B) – ” – $A_{2k} + B = \underline{1}0100111_2 + \underline{0}0100110_2 = \underline{1}1001101_2$, motsv. -51_{10} (Korrekt, olika tecken.)
(–A) + (+C) – ” – $A_{2k} + C = \underline{1}0100111_2 + \underline{0}0111001_2 = \underline{1}1100000_2$, motsv. -32_{10} (Korrekt, olika tecken.)
(–A) + (–B) – ” – $A_{2k} + B_{2k} = \underline{1}0100111_2 + \underline{1}1011010_2 = \underline{1}0000001_2$ motsv. -127_{10} (Korrekt, 1 + 1 = 1.)
(–A) + (–C) – ” – $A_{2k} + C_{2k} = \underline{1}0100111_2 + \underline{1}1000111_2 = \underline{0}1101110_2 = +110_{10}$ (Overflow, 1 + 1 = 0.)

Subtraktion (Alla beräkningar nedan utförs med 8 bitar.)
(+A) – (+B) motsvaras av $A + B_{1k} + 1 = \underline{0}1011001_2 + \underline{1}1011001_2 + 1 = \underline{0}0110011_2 = +51_{10}$ (Korrekt, olika tecken.)
(+A) – (+C) – ” – $A + C_{1k} + 1 = \underline{0}1011001_2 + \underline{1}1000110_2 + 1 = \underline{0}0100000_2 = +32_{10}$ (Korrekt, olika tecken.)
(+A) – (–B) – ” – $A + (B_{2k})_{1k} + 1 = \underline{0}1011001_2 + \underline{0}0100101_2 + 1 = \underline{0}1111111_2 = +127_{10}$ (Korrekt, 0 + 0 = 0)
(+A) – (–C) – ” – $A + (C_{2k})_{1k} + 1 = \underline{0}1011001_2 + \underline{0}0111000_2 + 1 = \underline{1}0010010_2$ motsv. -110_{10} (Overflow, 0 + 0 = 1)
(–A) – (+B) – ” – $A_{2k} + B_{1k} + 1 = = \underline{1}0100111_2 + \underline{1}1011001_2 + 1 = \underline{1}0000001_2$ motsv. -127_{10} (Korrekt, 1 + 1 = 1)
(–A) – (+C) – ” – $A_{2k} + C_{1k} + 1 = = \underline{1}0100111_2 + \underline{1}1000110_2 + 1 = \underline{0}1101110_2 = +110_{10}$ (Overflow, 1 + 1= 0)
(–A) – (–B) – ” – $A_{2k} + (B_{2k})_{1k} + 1 = \underline{1}0100111_2 + \underline{0}0100101_2 + 1 = \underline{1}1001101_2$, motsv. -51_{10} (Korrekt, olika tecken.)
(–A) – (–C) – ” – $A_{2k} + (C_{2k})_{1k} + 1 = \underline{1}0100111_2 + \underline{0}0111000_2 + 1 = \underline{1}1100000_2$, motsv. -32_{10} (Korrekt, olika tecken.)

Flaggorna N, Z, V och C som sätts vid aritmetiska operationer.

När man utför de aritmetiska operationerna addition eller subtraktion mellan två maskintal i en dator är det oftast resultatet, dvs summan eller skillnaden man söker. Man vill också veta om resultatet är korrekt med det antal sifferpositioner som används. I vissa fall vill man veta om resultatet är noll eller om det är positivt eller negativt om det är ett tal med tecken. Av dessa skäl bildar man förutom resultatet också ett antal ”flaggor” som innehåller information om resultatets giltighet, värde och tecken.

N-flaggan (Negative) sätts till resultatets mest signifikanta bit (msb), dvs ”teckenbiten” för tal med tecken.

Z-flaggan (Zero) sätts till värdet 1 om resultatet är noll.

V-flaggan (oVerflow) sätts till värdet 1 om ”overflow” för tal med tecken har inträffat, dvs resultatet av en addition eller subtraktion är felaktigt eftersom det korrekta resultatet ligger utanför det tillgängliga talområdet (= inte ryms med det antal sifferpositioner som används).

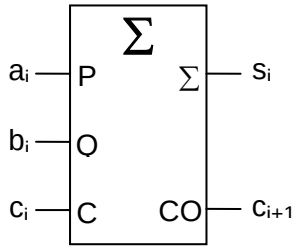
C-flaggan (Carry/Borrow):
C-flaggan sätts till värdet 1 om ”overflow” för tal utan tecken har inträffat vid addition, dvs resultatet av additionen ligger utanför talområdet (= inte ryms med det antal sifferpositioner som används).

C-flaggan har en annan funktion för subtraktion av ”tal utan tecken”. Den sätts till värdet 1 om resultatet av en subtraktion skulle bli negativt. (Det finns ju inga negativa tal för ”tal utan tecken”.)

Principen och grindnät för addition av två n-bitars tal

$$
\begin{array}{llllllll}
c_n & c_{n-1} & c_{n-2} & .. & c_{i+1} & c_i & ..c_1 & 0 \quad \text{minnessiffror} \\
 & a_{n-1} & a_{n-2} & ..... & a_i & & ..a_1 & a_0 \quad \text{augendsiffror} \\
+ & b_{n-1} & b_{n-2} & ..... & b_i & & ..b_1 & b_0 \quad \text{addendsiffror} \\
\hline
s_n & s_{n-1} & s_{n-2} & ..... & s_i & & ..s_1 & s_0 \quad \text{summasiffror}
\end{array}
$$

För att addera två n-bitars tal krävs det enligt uppställningen ovan n st grindnät som bildar utsignalerna s_i och c_{i+1} av insignalerna a_i, b_i och c_i, dvs n st heladderare med symbol och funktionstabell nedan.



Heladderare

a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ur funktionstabellen får man fram s_i och c_{i+1} på SP normal form:

$$s_i = a_i'b_i'c_i + a_i'b_ic_i' + a_ib_i'c_i' + a_ib_ic_i \qquad \text{(4.2 i KMP)}$$

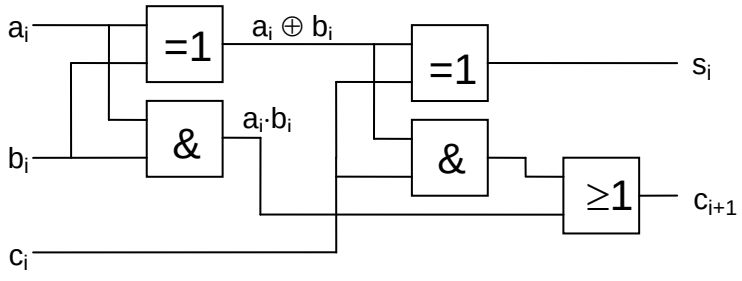
$$c_{i+1} = a_i'b_ic_i + a_ib_i'c_i + a_ib_ic_i' + a_ib_ic_i \qquad \text{(4.3 i KMP)}$$

Förenkling ger:

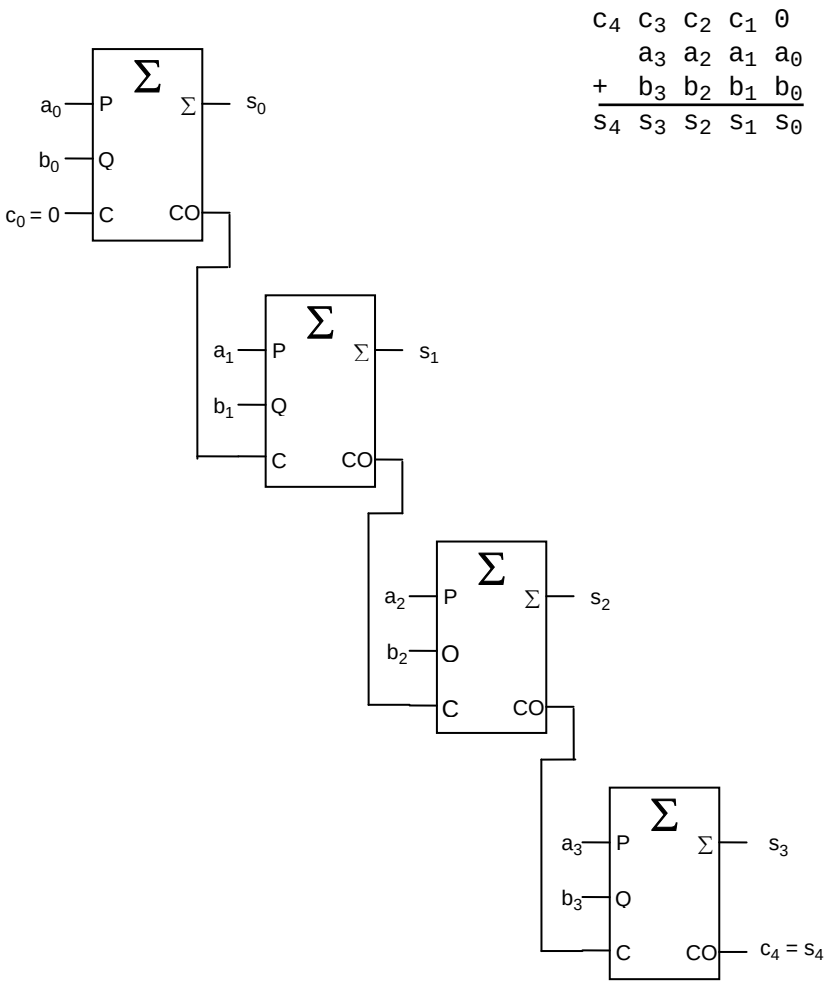
$$s_i = (a_i \oplus b_i) \oplus c_i \qquad \text{(4.4 i KMP)}$$

$$c_{i+1} = a_ib_i + (a_i \oplus b_i)\cdot c_i \qquad \text{(4.5 i KMP)}$$

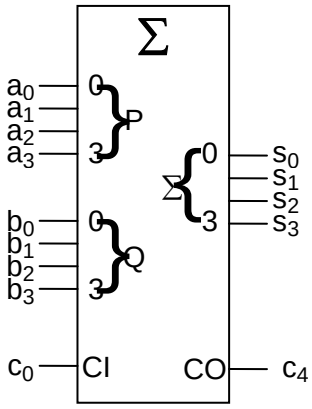
Motsvarande grindnät:



Genom att koppla ihop fyra heladderare enligt figuren nedan får man en ”fyrabitars heladderare” som adderar två st fyrabitars tal enligt uppställningen till höger.

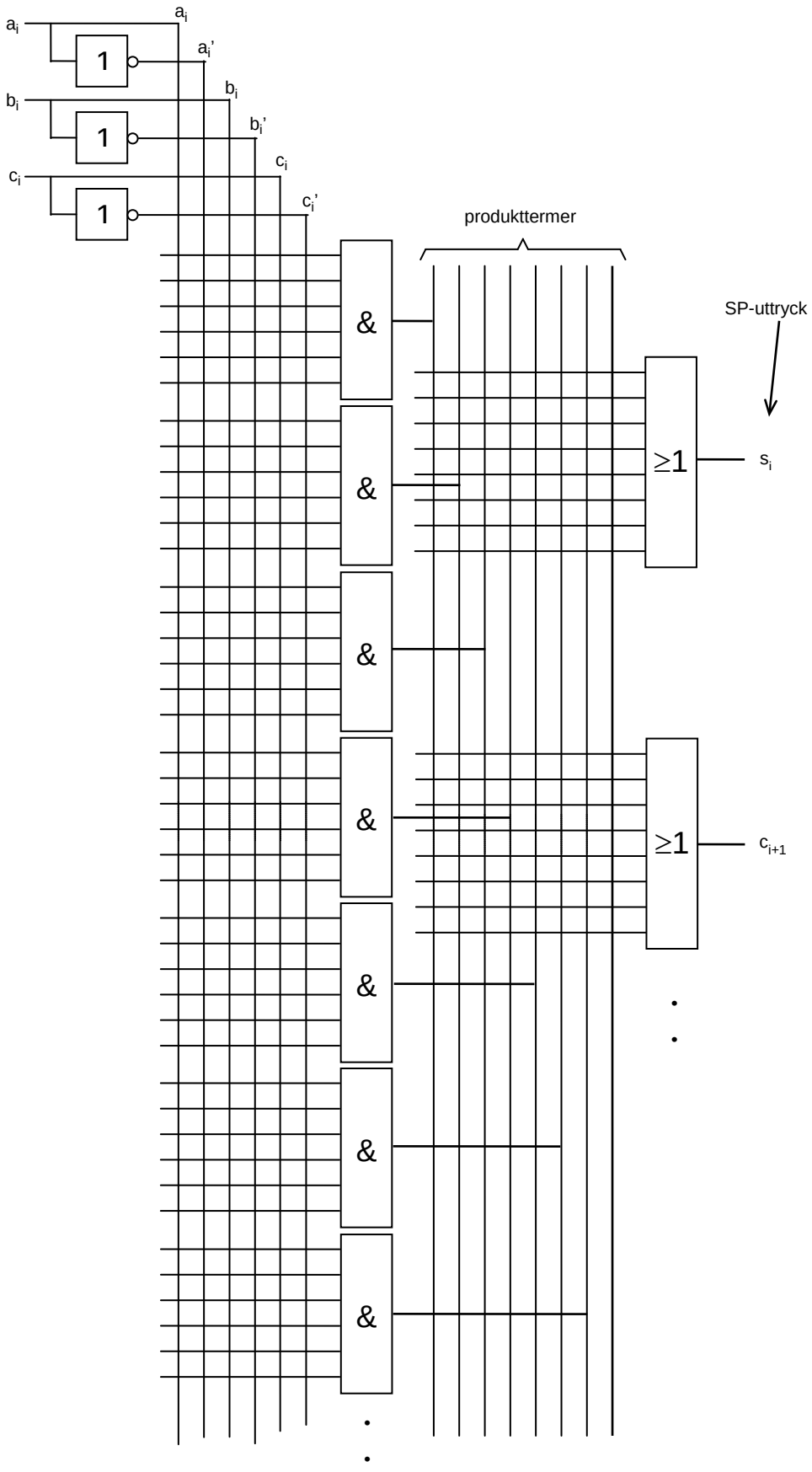


Symbol för en fyrabitars heladderare:



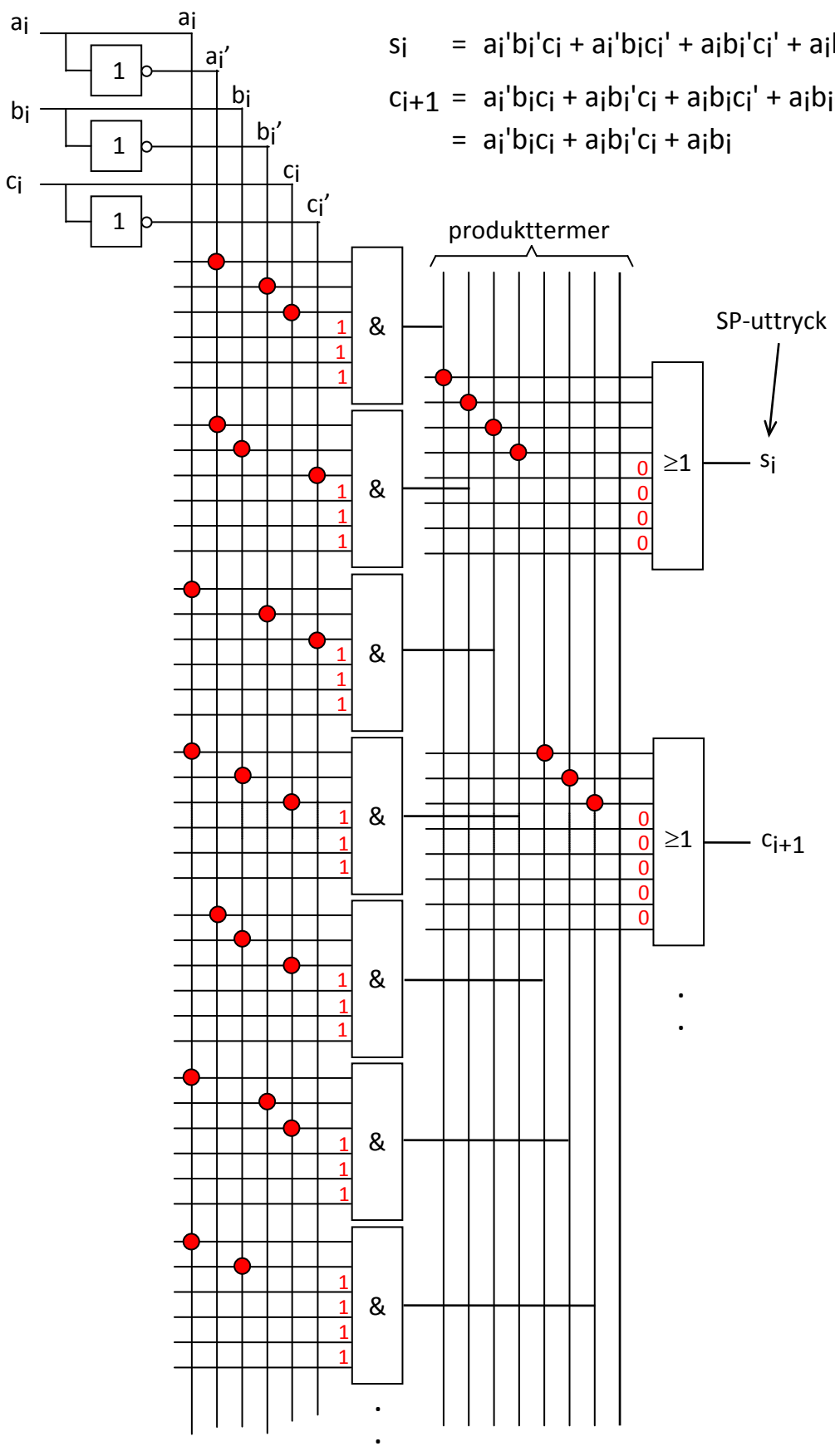
Realisering av heladderare med AND/OR-nät

4.26



Realisering av heladderare med AND/OR-nät (Lösning)

4.26



$s_i = a_i'b_i'c_i + a_i'b_ic_i' + a_ib_i'c_i' + a_ib_ic_i$

$c_{i+1} = a_i'b_ic_i + a_ib_i'c_i + a_ib_ic_i' + a_ib_ic_i =$

$= a_i'b_ic_i + a_ib_i'c_i + a_ib_i$

Aritmetikuppgifter

6.5 Beräkna 2-komplementet till följande tal

a) 0101_2

c) 1011_2

f) 001100_2

6.8 I en dator representeras heltal av kodord $x_7x_6x_5x_4\ x_3x_2x_1x_0$.

För positiva tal är $x_7 = 0$ medan $x_6x_5x_4\ x_3x_2x_1x_0$ anger talets storlek enligt naturlig binärkod.

Negativa heltal representeras som 2-komplementet till motsvarande positiva tal.

Datorn arbetar med binär aritmetik. Subtraktion utförs som addition av subtrahendens 1-komplement med additionskretsens minnessiffra in (CI) ettställd.

c) Visa hur additionen $(+73)_{10} + (+45)_{10}$ utförs i datorn.

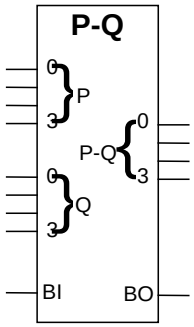
d) Visa hur subtraktionen $(+73)_{10} - (+45)_{10}$ utförs i datorn.

i) Visa hur additionen $(+123)_{10} + (+45)_{10}$ utförs i datorn.
Vilket decimalt tal motsvaras summan av?
Observera, att alla tal representeras av 8-bitars kodord i datorn. Diskutera resultatet.

j) Visa hur subtraktionen $(-45)_{10} - (+123)_{10}$ utförs i datorn.
Vilket decimalt tal motsvaras skillnaden av?
Observera, att alla tal representeras av 8-bitars kodord i datorn. Diskutera resultatet.

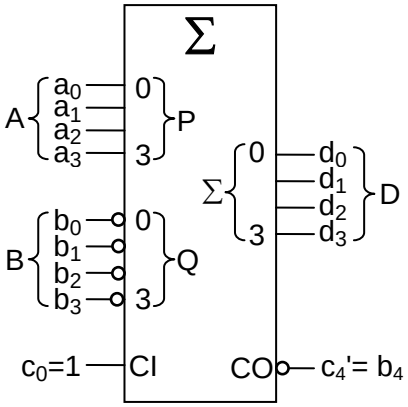
Subtraktionskrets

b_n	$b_{n-1}b_{n-2}...b_i...b_10$	lånesiffror
	$x_{n-1}x_{n-2}...x_i...x_1x_0$	minuendsiffror
-	$y_{n-1}y_{n-2}...y_i...y_1y_0$	subtrahendsiffror
	$s_{n-1}s_{n-2}...s_i...s_1s_0$	skillnadsiffror



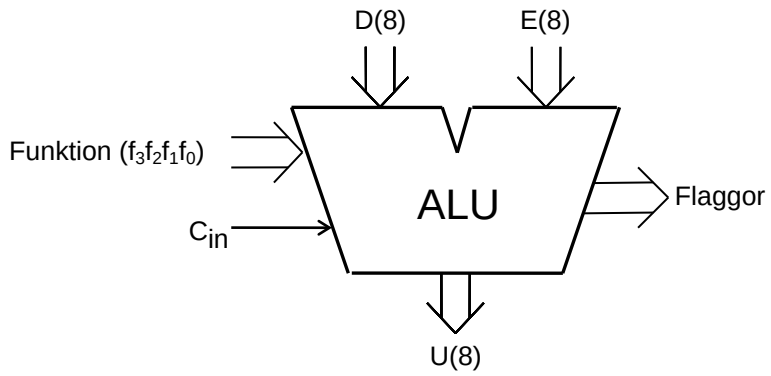
Figur 6.5

Istället för kretsen ovan utför man subtraktion med additionskretsen nedan enligt: $D = A + B_{1k} + 1$



Inverteringen av c_4 förklaras av att man vill att $b_4 = 1$ om $A < B$, dvs om A är mindre än B.
Man utför ju operationen $D = A + B_{1k} + 1 = 2^4 + A - B$, som är mindre än 2^4 om $A < B$.
Det innebär att $c_4 = 0$ om $A < B$. Genom att invertera c_4 får vi därför rätt värde på b_4.

Aritmetik- logikenhet för FLEX (FLEX-ALU)



ALU:ns **logik-** och **aritmetikoperationer** på indata **D** och **E** definieras av ingångarna **Funktion (F)** och $\mathbf{C_{in}}$ enligt tabellen nedan. **F = (f_3, f_2, f_1, f_0).**

I kolumnen Operation förklaras (där det behövs) hur operationen utförs. Tecknen **"+"** och **"–"** avser **aritmetiska operationer**. Med $\mathbf{D_{1k}}$ menas att samtliga bitar i **D** inverteras.

f_3 f_2 f_1 f_0	U = f(D,E,F,C_{in})	
	Operation	**Resultat**
0 0 0 0	Bitvis nollställning	0
0 0 0 1		D
0 0 1 0		E
0 0 1 1	Bitvis invertering	D_{1k}
0 1 0 0	Bitvis invertering	E_{1k}
0 1 0 1	Bitvis OR	D OR E
0 1 1 0	Bitvis AND	D AND E
0 1 1 1	Bitvis XOR	D XOR E
1 0 0 0	D + 0 + C_{in}	D + C_{in}
1 0 0 1	D + FFH + C_{in}	D – 1 + C_{in}
1 0 1 0	D + E + C_{in}	D + E + C_{in}
1 0 1 1	D + D + C_{in}	2D + C_{in}
1 1 0 0	D +E_{1k} + C_{in}	D – E – 1 + C_{in}
1 1 0 1	Bitvis nollställning	0
1 1 1 0	Bitvis nollställning	0
1 1 1 1	Bitvis ettställning	FFH

Carryflaggan (C) innehåller minnessiffran ut (carry-out) från den mest signifikanta bitpositionen (längst till vänster) om en aritmetisk operation utförs av ALU:n.

Vid **subtraktion** gäller för denna ALU att
C = 1 om lånesiffra (borrow) uppstår och
C = 0 om lånesiffra inte uppstår.

Carryflaggans värde är 0 vid andra operationer än aritmetiska.

Overflowflaggan (V) visar om en aritmetisk operation ger "overflow" enligt reglerna för 2-komplementaritmetik.

V-flaggans värde är 0 vid andra operationer än aritmetiska.

Zeroflaggan (Z) visar om en ALU-operation ger värdet noll som resultat på U-utgången.

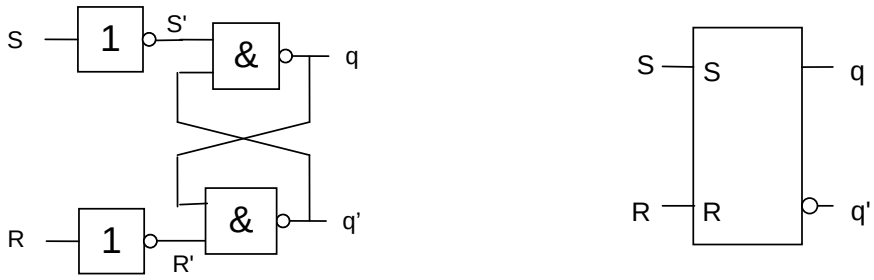
Signflaggan (N) är identisk med den mest signifikanta biten (teckenbiten) av utsignalen U från ALU:n.

Kapitel 5 Sekvensnät

5.1 Grundläggande minneselement (Latchar)

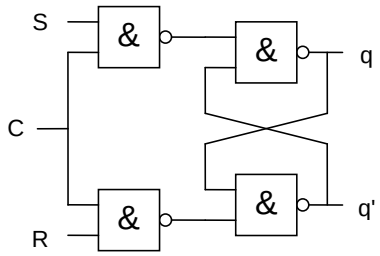
SR-latch:

Figur 5.10 SR(RS)-latch. Koppling och symbol.

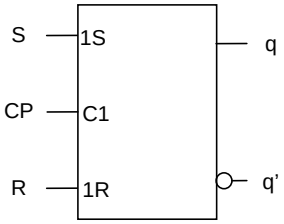


Figur 5.11 Annan SR-latch. Koppling och symbol.

Grindad (Klockad) SR-latch:

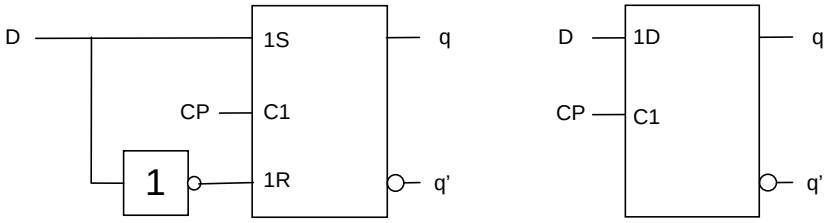


Figur 5.12 Grindad SR-latch. Koppling.



Figur 5.15 Grindad SR-latch. Symbol.

Klockad D-latch:

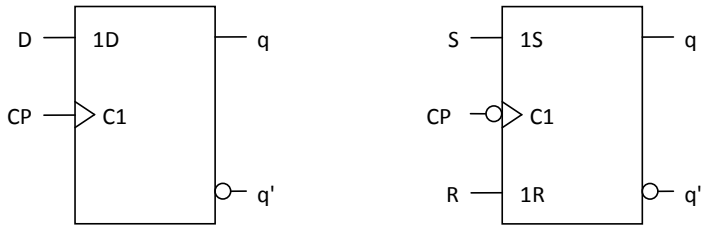


Figur 5.17 Klockad D-latch. Koppling och symbol.

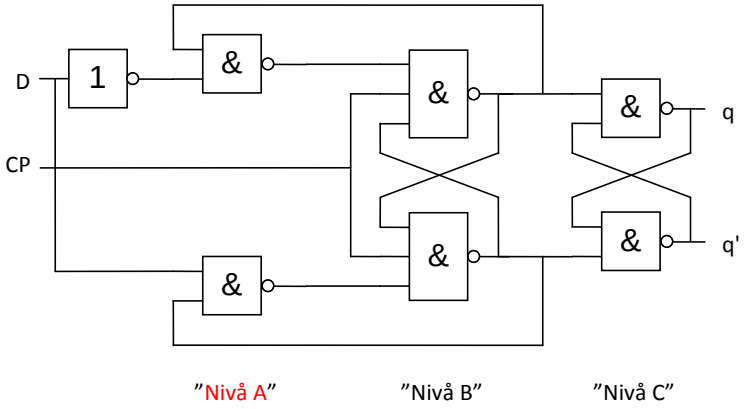
Kapitel 5 Sekvensnät

5.2 Vippor

D- och SR-vippan:

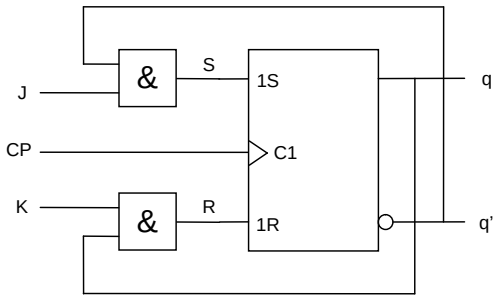


Figur 5.24 Symboler för D-vippa med triggning på positiv flank och SR-vippa med triggning på negativ flank (högra figuren).

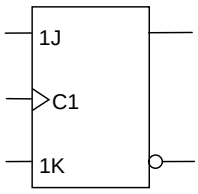


Figur 5.25 D-vippa med triggning på positiv flank.

JK-vippan:



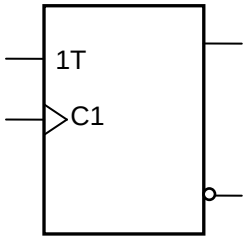
Figur 5.26 Principen för JK-vippan.



Figur 5.27 Symbol för JK-vippan.

T-vippan:

T-vippan är en JK-vippa med insignalen T kopplad till både J- och K-ingången.

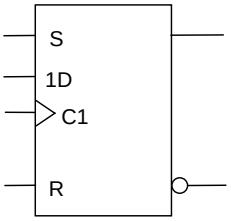


Symbol för T-vippan.

Asynkrona ingångar för ettställning och nollställning:

När en vippa ansluts till matningsspänningen vet man inte om den startar med Q = 0 eller 1.

Man kan dock förse vipporna med extra ingångar som tvingar dem att starta med ett känt Q-värde. Figuren nedan visar symbolen för en D-vippa med asynkrona Set- och Resetingångar. Att ingångarna är asynkrona innebär att de inte är beroende av klocksignalen. Det innebär att när t ex S är aktiv, så kommer Q att anta värdet 1 (S = Set, ettställ) oberoende av övriga insignaler. Om R är aktiv får Q värdet 0 (R = Reset, nollställ) på motsvarande sätt. Endast en av S och R skall vara aktiv samtidigt.



D-vippa med asynkrona Set- och Resetingångar.

Funktionstabeller för de olika vipptyperna:

D	q^+
0	0
1	1

S	R	q^+
0	0	q
0	1	0
1	0	1
1	1	?

J	K	q^+
0	0	q
0	1	0
1	0	1
1	1	q'

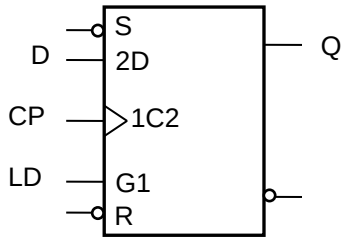
T	q^+
0	q
1	q'

D-vippa med "load enable"-ingång

En vanlig D-vippa laddas med nytt innehåll från D-ingången vid varje klockpuls. I ett system där en gemensam klocksignal är ansluten till alla vippor är detta inte lämpligt. Där vill man ha möjligheten att välja vilka vippor, som skall ladda ett nytt värde. I sådana fall krävs en extra ingång som bestämmer om vippan skall laddas med ett nytt värde.

Man kan komplettera en vanlig D-vippa så att den kan förberedas för att ladda eller inte ladda ett nytt värde vid positiv klockpulsflank.

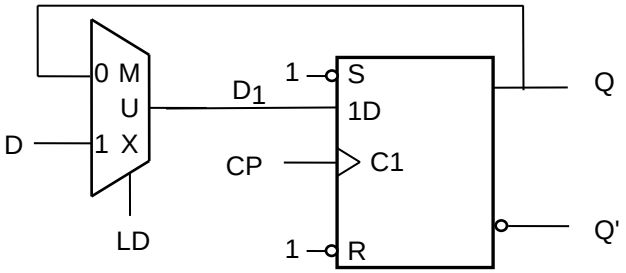
För att möjliggöra selektiv laddning skall vippan alltså kompletteras så att den också får en **"grindande"** ingång G1, som är en s k "load enable"-ingång. Ett exempel på en sådan D-vippa ges av Prosamsymbolen i figuren nedan, där **LD**, står för **"load enable"**. Denna D-vippa har som synes också asynkrona Set- och Reset-ingångar.



SEK Prosam-symbol för en D-vippa med "load enable"-ingång

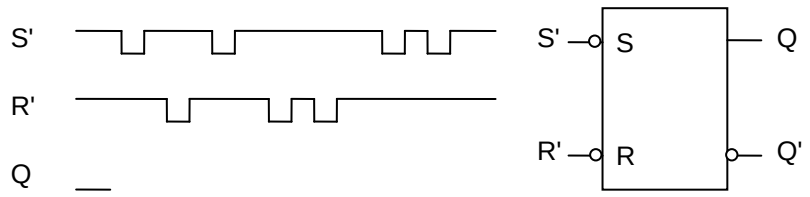
Funktionsbeskrivning

Genom att komplettera en vanlig D-vippa med en ”1 av 2 väljare” enligt figuren nedan kan man välja med insignalen LD, om vippan skall laddas med ett nytt värde Q = D vid aktiv klockpulsflank eller behålla det gamla värdet Q. När LD=1 laddas den med det nya värdet D och när LD = 0 återladdas den med sitt gamla värde Q. LD är således en **styrsignal,** som styr hur vippan skall laddas vid klockpulsens aktiva flank, i detta fall den positiva flanken.

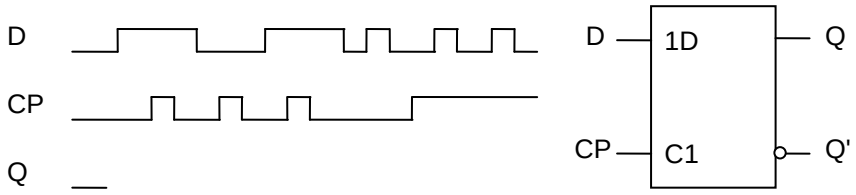


Studium av latchar och vippor

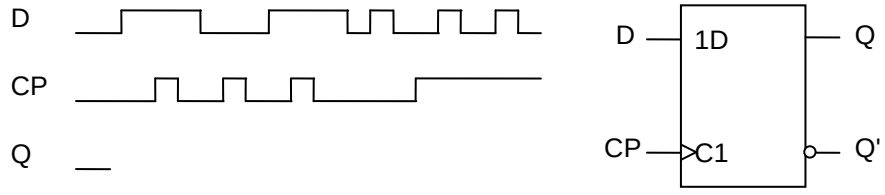
5.1 I figuren visas en RS-latch jämte tidsdiagram för S'- och R'-signalerna. Rita upp tidsdiagrammet för Q om Q = 0 från början.



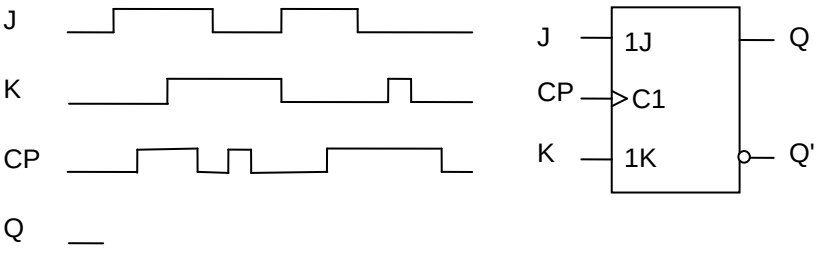
5.3 I figuren visas en klockad D-latch jämte tidsdiagram för D- och CP-signalerna. Rita upp tidsdiagrammet för Q om Q = 0 från början.



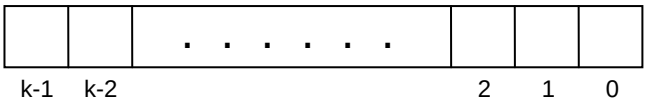
5.7 I figuren visas en flanktriggad D-vippa jämte tidsdiagram för CP- och D-signalerna. Rita upp tidsdiagrammet för Q om Q = 0 från början.



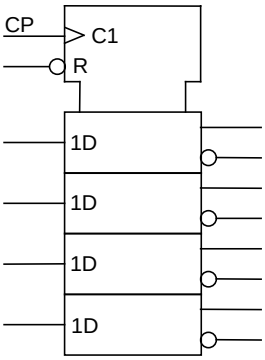
5.8 I figuren visas en flanktriggad JK-vippa jämte tidsdiagram för CP-, J- och K-signalerna. Rita upp tidsdiagrammet för Q om Q = 0 från början.



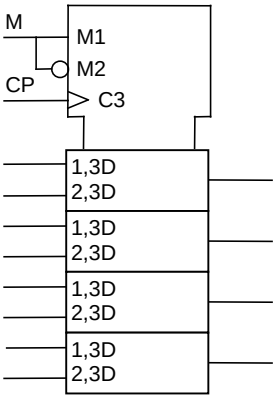
Kapitel 5.3 Register



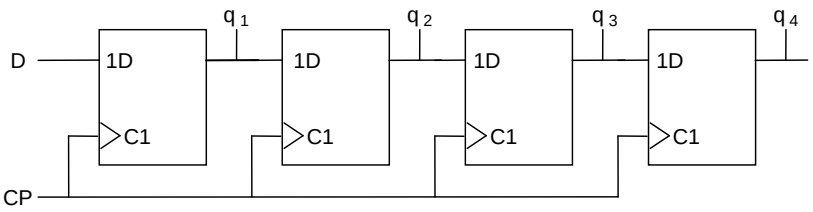
Figur 5.32 Vanligen använd symbol för k bitars register.



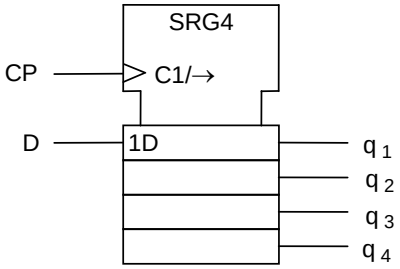
Figur 5.33 Fyrabitars register med asynkron nollställning



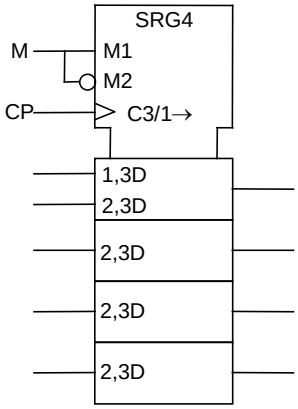
Figur 5.34 Fyrabitars 2-ingångs register



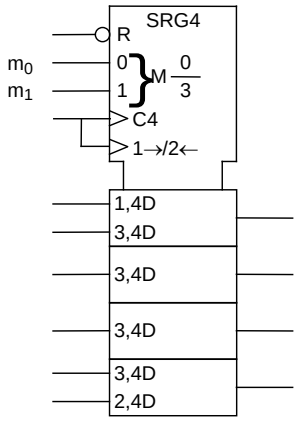
Figur 5.35 4-bitars skiftregister uppbyggt med 4 stycken D-vippor.



Figur 5.1 Symbol för ett 4-bitars skiftregister.



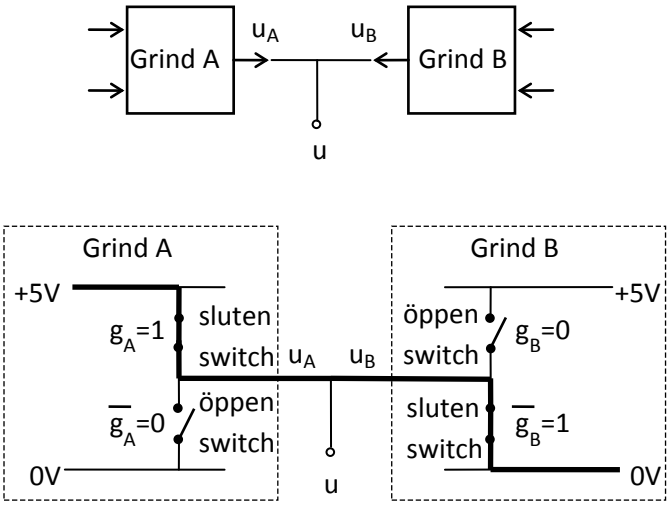
Figur 5.38 Symbol för 4-bitars skiftregister med parallellinskrivning.



Figur 5.39 Prosamsymbol för skiftregistret 74194.

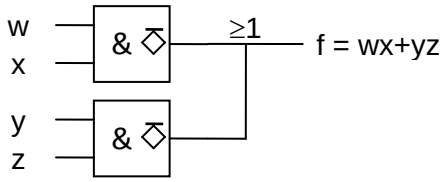
4.1 Möjligheter att koppla ihop logikkretsars utgångar

(Grundregeln är att logikkretsutgångar inte får kopplas ihop.)



Figur 4.19 Två standardgrindar med ihopkopplade utgångar.

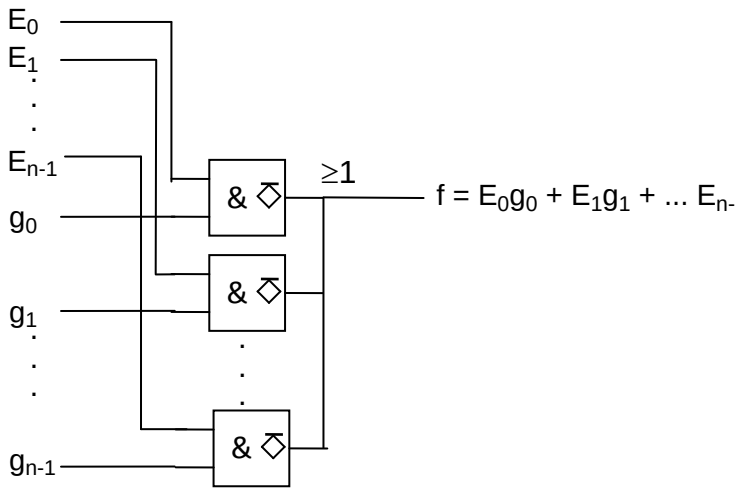
Trådningsbara grindar



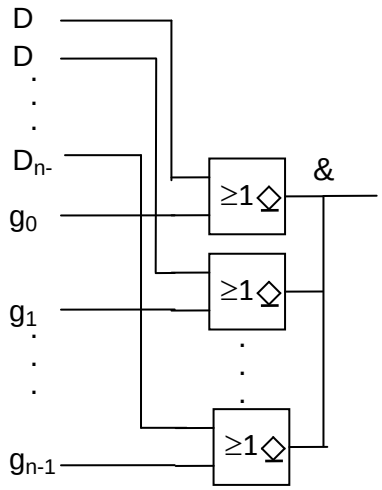
Figur 4.20 Wired-OR.

Tabell 4.8

Operation	*Svensk standard*
ELLER-bildning i föreningspunkt (wired-Or)	≥1
OCH-bildning i föreningspunkt (wired-AND)	&

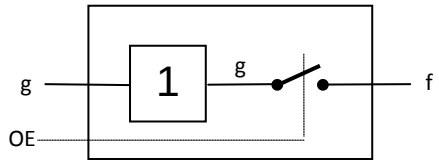


Figur 4.21 Väljarfunktion genom ELLER-bildande trådning.



Figur 4.22 Väljarfunktion genom OCH-bildande trådning.

Kap 7.2 Ihopkoppling av utgångar via "three-state"-buffert



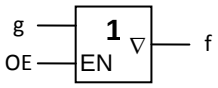
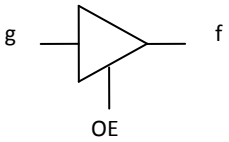
Tabell 7.1

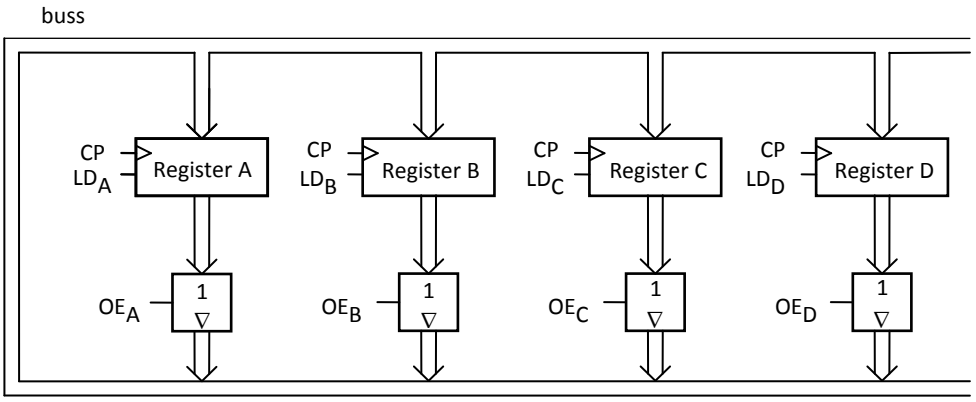
OE	g	f
0	0	Z
0	1	Z
1	0	0
1	1	1

Figur 7.20 Three-state buffert

Utöver de två tillstånden 0 och 1 kan utgången f också inta ett tredje tillstånd Z, där utgången är bortkopplad från ingången, dvs bortkopplad från g-signalen.
Signalen som styr switchen i figur 7.20 kallas för ***OE (Output Enable)***. När OE = 1 så är switchen sluten, vilket innebär att f = g.
Standardsymbolerna för "three-state" bufferten ges i tabell 7.2.

Tabell 7.1 (EN = Enable)

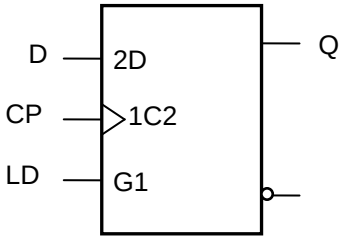
Operation	Svensk standard	Amerikansk symbol
"three-state"		



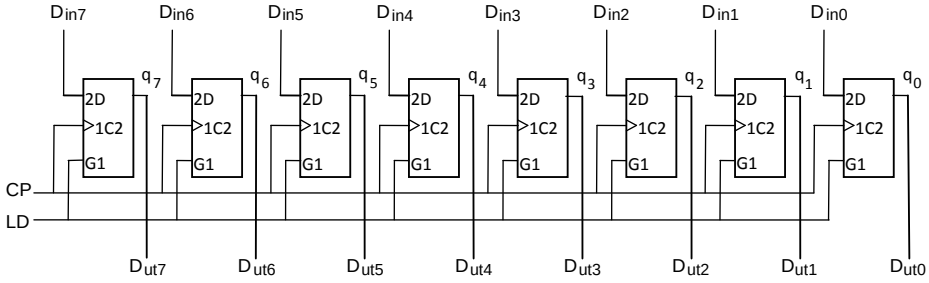
Figur 7.23 Registerutgångars anslutning till buss via "three-state" buffertar.

Kapitel 7 Systemexempel

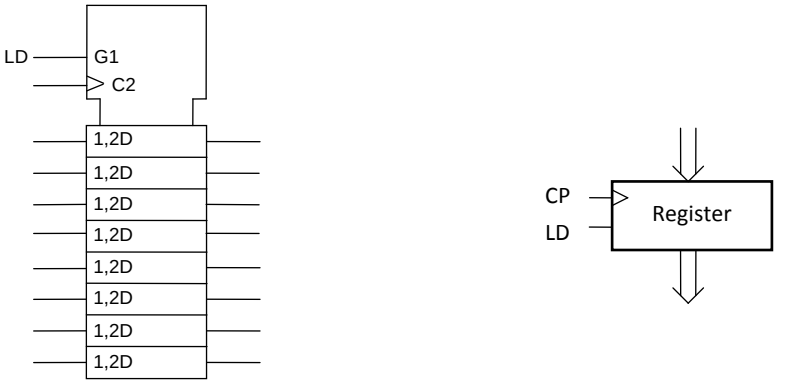
Grindad D-vippa:



Register med laddingång:

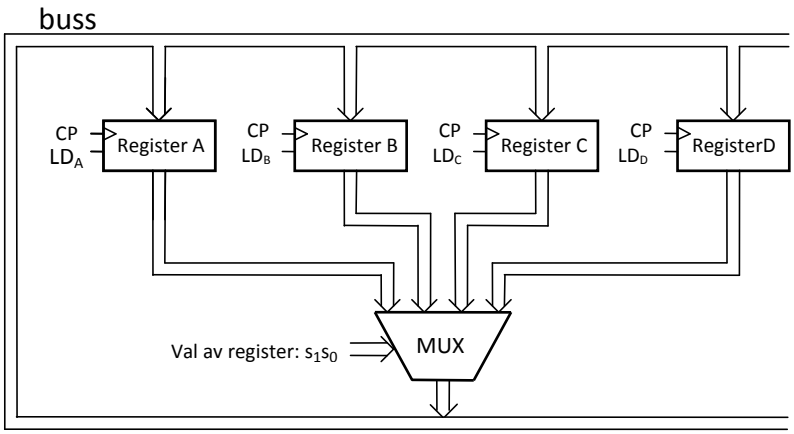


Figur 7.11 Principen för ett 8-bitars laddbart register.

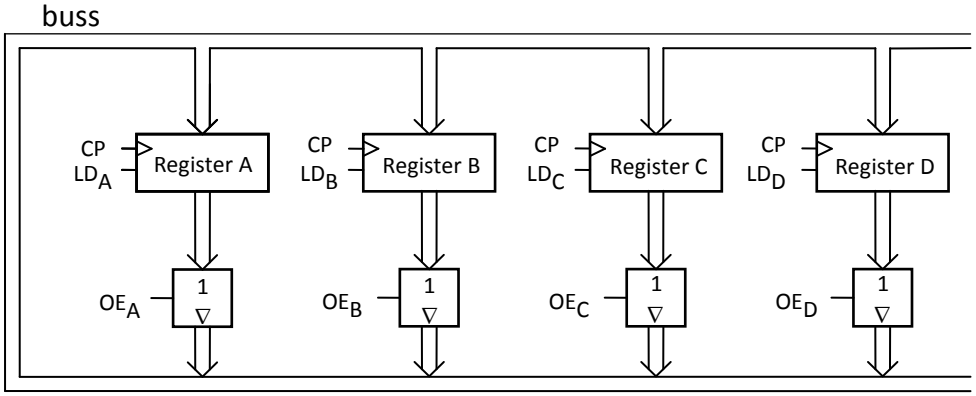


a) standardsymbol b) förenklad symbol

Figur 7.12 Symboler för ett 8-bitars laddbart register.



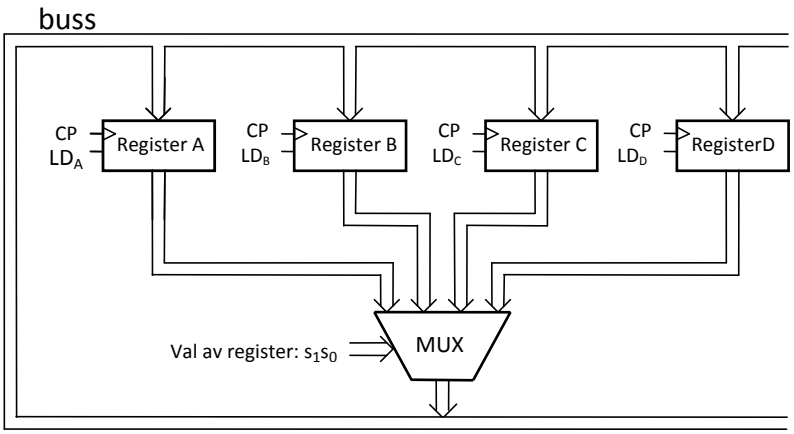
Figur 7.17 Registerutgångars anslutning till buss via väljare.



Figur 7.23 Registerutgångars anslutning till buss via three-state buffertar.

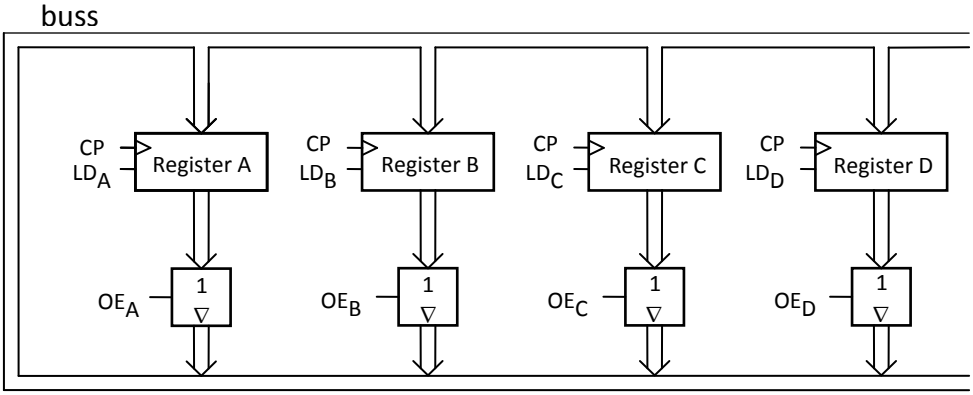
Kapitel 7 Systemexempel (forts.)

Hur man kan koppla ett av registerinnehållen till bussen.



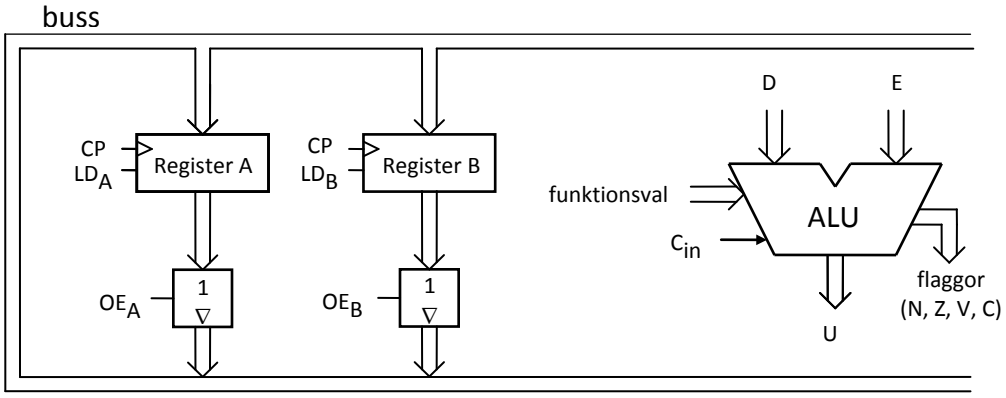
Figur 7.17 Registerutgångars anslutning till buss via väljare.

Alternativ utan väljare (multiplexer):



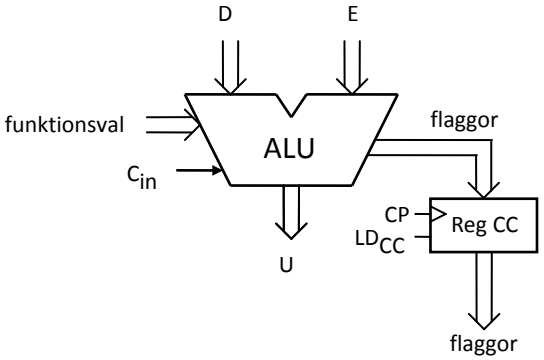
Figur 7.23 Registerutgångars anslutning till buss via three-state buffertar.

Nu vill vi även koppla in en ALU till bussen.



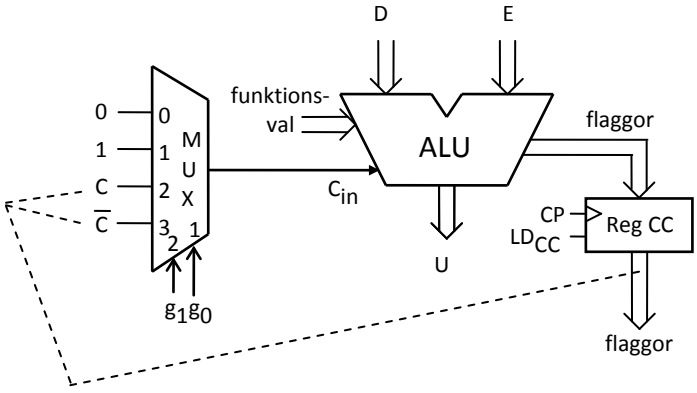
Figur 7.25 Dataväg med två register och en aritmetik-logik-enhet (ALU).

Vi gör det i flera steg.



Figur 7.26 ALU där flaggvärdena lagras i ett flaggregister CC.
(CC står för Condition Code.)

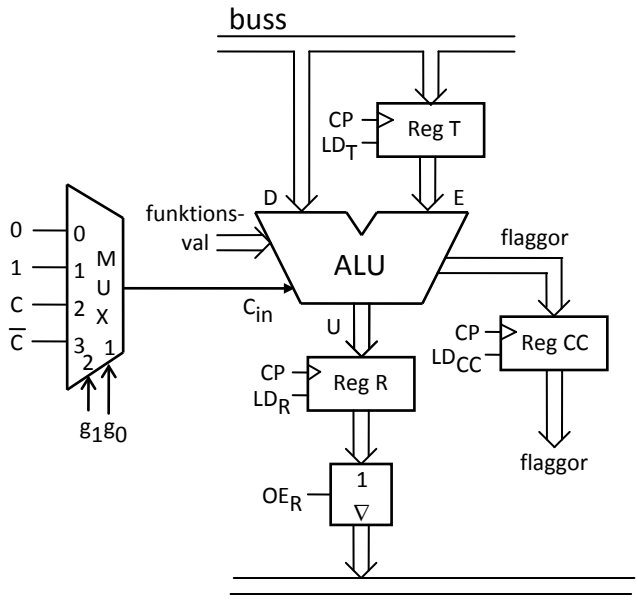
Se till att C_{in} till ALU'n kan väljas på alla nödvändiga sätt.
Då passar det bra med en multiplexer. (C tas från flaggregistret.)



Figur 7.27 Inkoppling av väljare (multiplexer) för val av C_{in}.

Eftersom ALU'n är ett kombinatoriskt nät så måste alla insignaler finnas samtidigt på ingångarna.

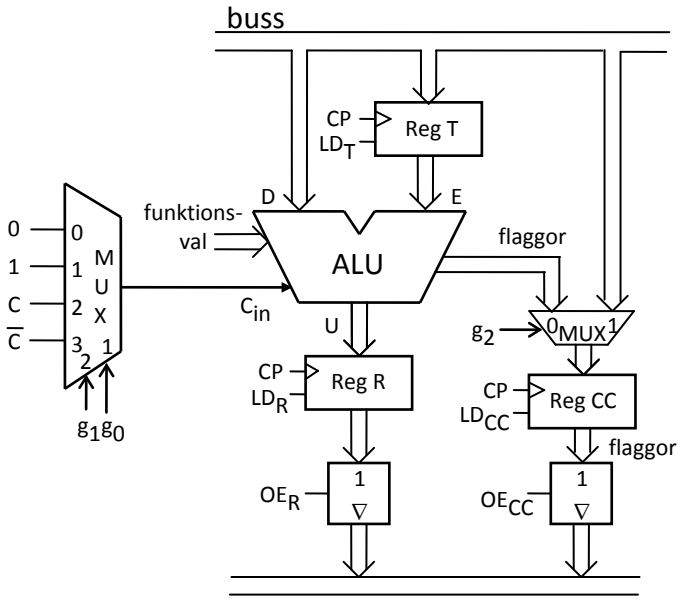
Därför kopplar vi in ett register (T) till E-ingången för den ena invariabeln till ALU'n. Den andra invariabeln kopplar vi direkt till D-ingången från bussen.



Figur 7.28 Användning av temporärregister, T och R, för E respektive U.

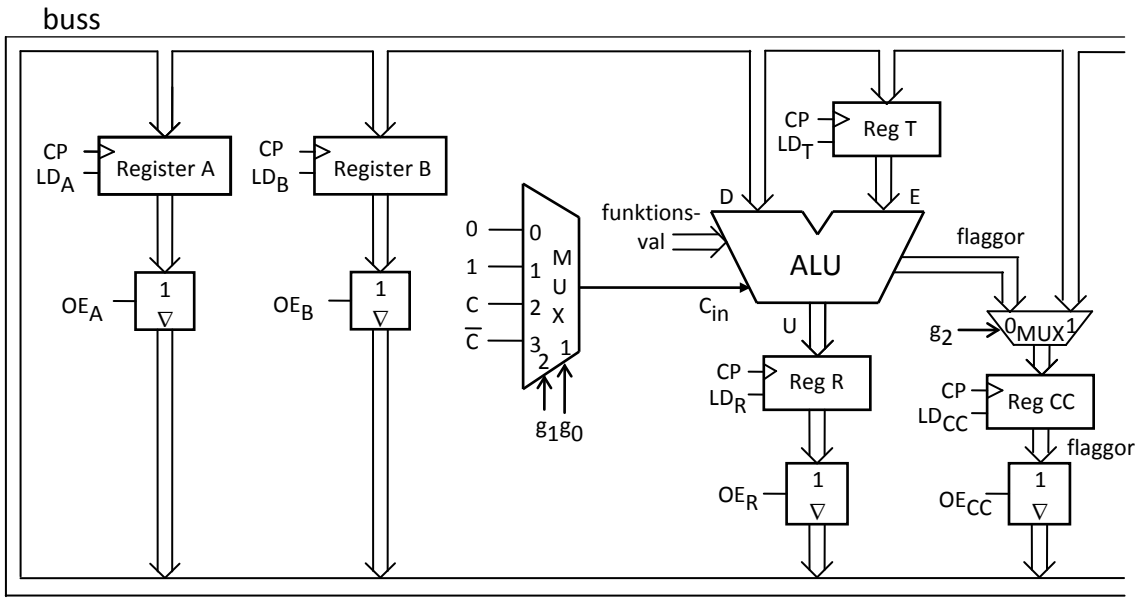
Bussen är upptagen av invariabeln till ALU'ns D-ingång. Vi kopplar därför ALU'ns utsignal U till ingången på ett register (R), som vi senare kan koppla ut på bussen via en three-state buffert, när bussen är ledig.

För att kunna välja om flaggorna skall laddas från ALU'n eller från bussen ansluter vi en väljare till CC-registrets ingång. Vi ansluter också CC-registrets utgång till bussen via en "three-state" buffert.



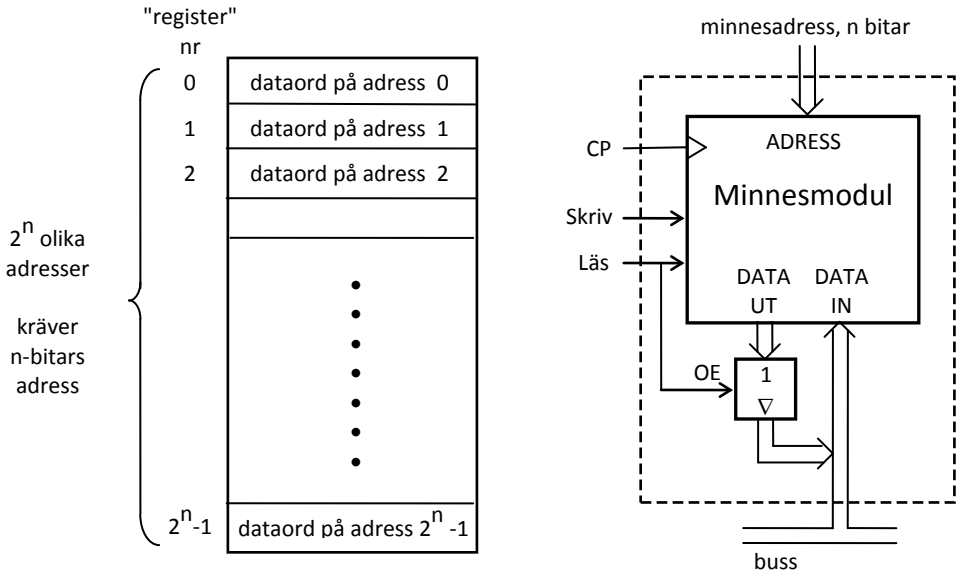
Figur 7.29 Inkoppling av väljare (multiplexer) för val av indata till CC-registret.

Den fullständiga kopplingen visas nedan:



Figur 7.30 Dataväg med register och ALU.

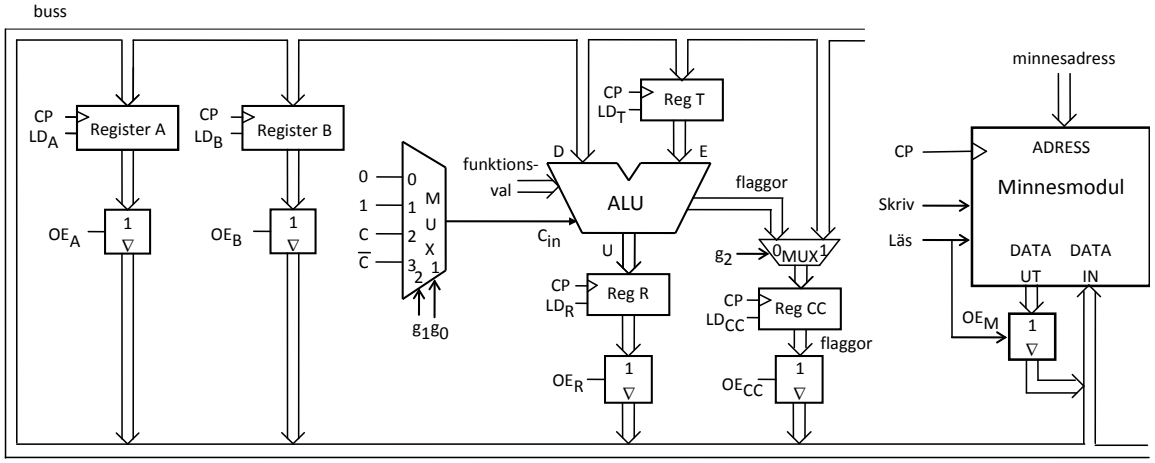
Principen för en minnesmodul med läs- och skrivbart minne (RWM).



a) adresseringsprincip b) förenklad symbol

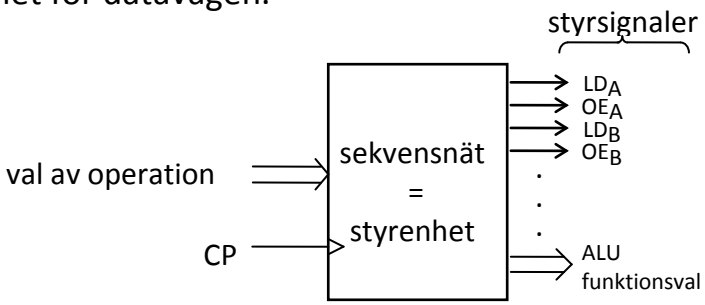
Figur 7.31 RWM, minnesmodul för läsning och skrivning.

Datavägen kompletterad med läs- och skrivbart minne (RWM).



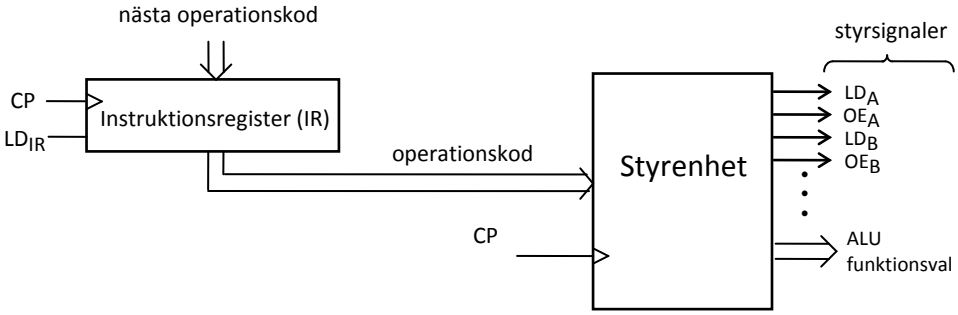
Figur 7.32 Dataväg med RWM.

En styrenhet för datavägen.



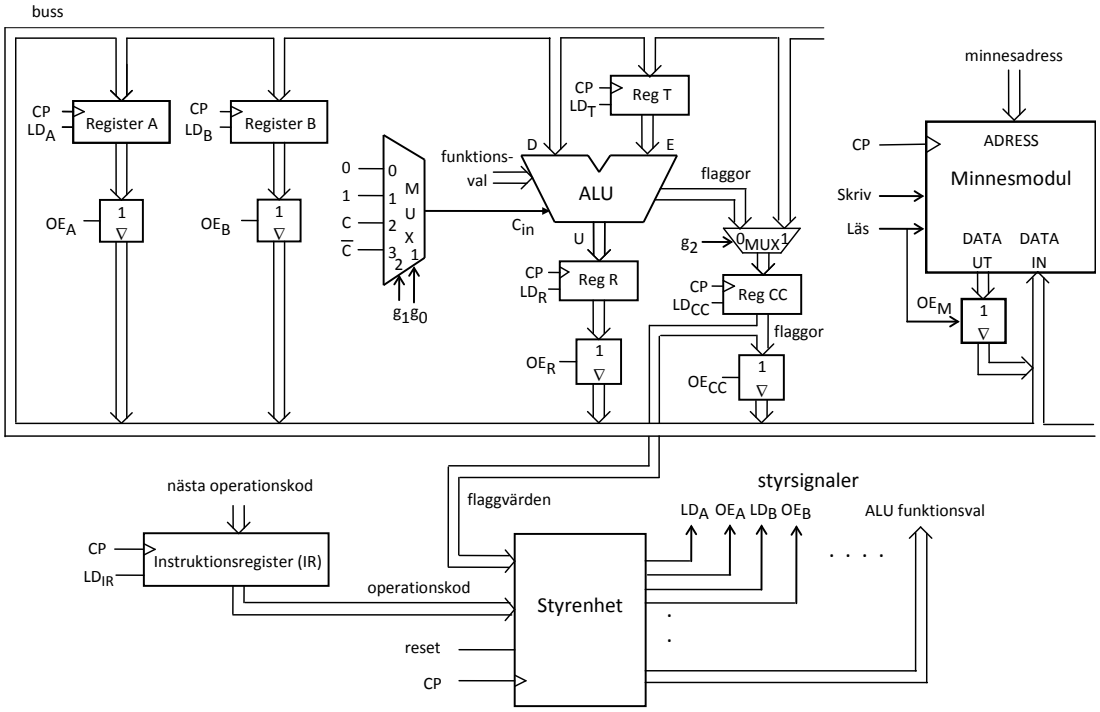
Figur 7.33 Generering av styrsignaler till datavägen.

Komplettera med ett register för operationsnummer.



Figur 7.34 Operationskoden finns i instruktionsregistret.

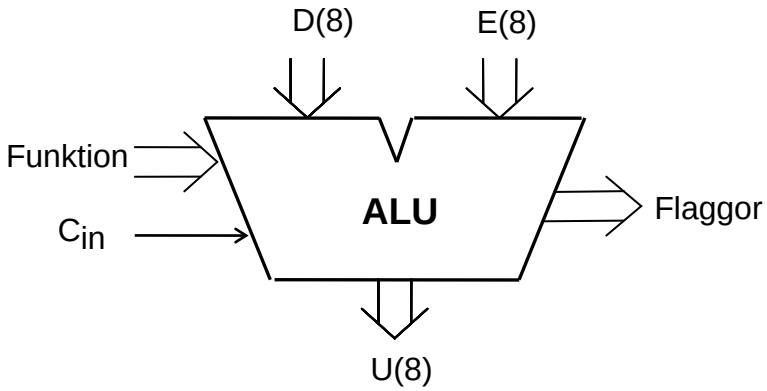
Dataväg med styrenhet.



Figur 7.35 Dataväg med styrenhet.

RTN-beskrivning och styrsignaler för operationer i en enkel dataväg

För övningsuppgifterna 7.3 och 7.4 gäller systemet i figur 7.30 eller 7.32. För ALU:n gäller:

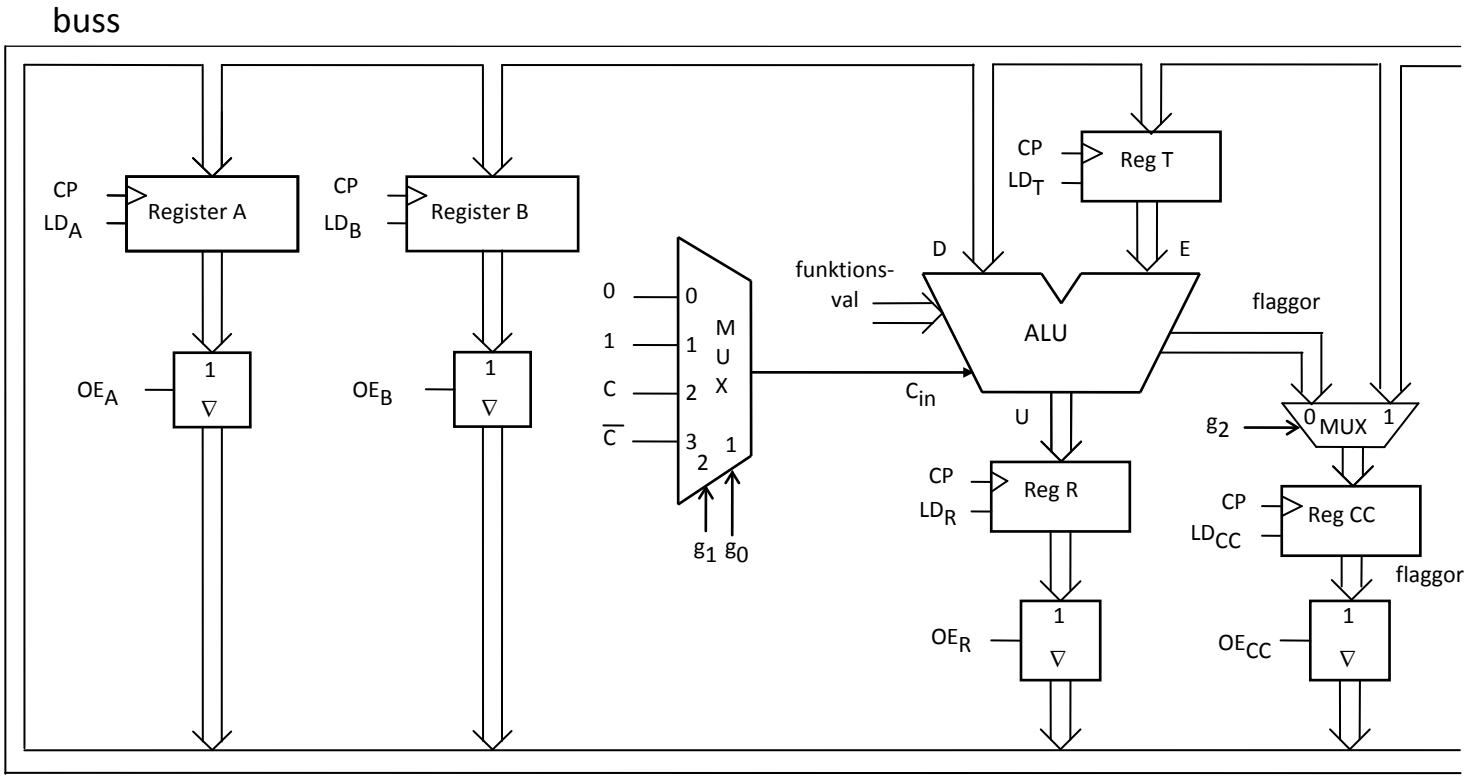


- ALU:ns **logik-** och **aritmetikoperationer** på indata **D** och **E** definieras av ingångarna **Funktion (F)** och $\mathbf{C_{in}}$ enligt tabellen.
  F = $\mathbf{(f_3, f_2, f_1, f_0)}$

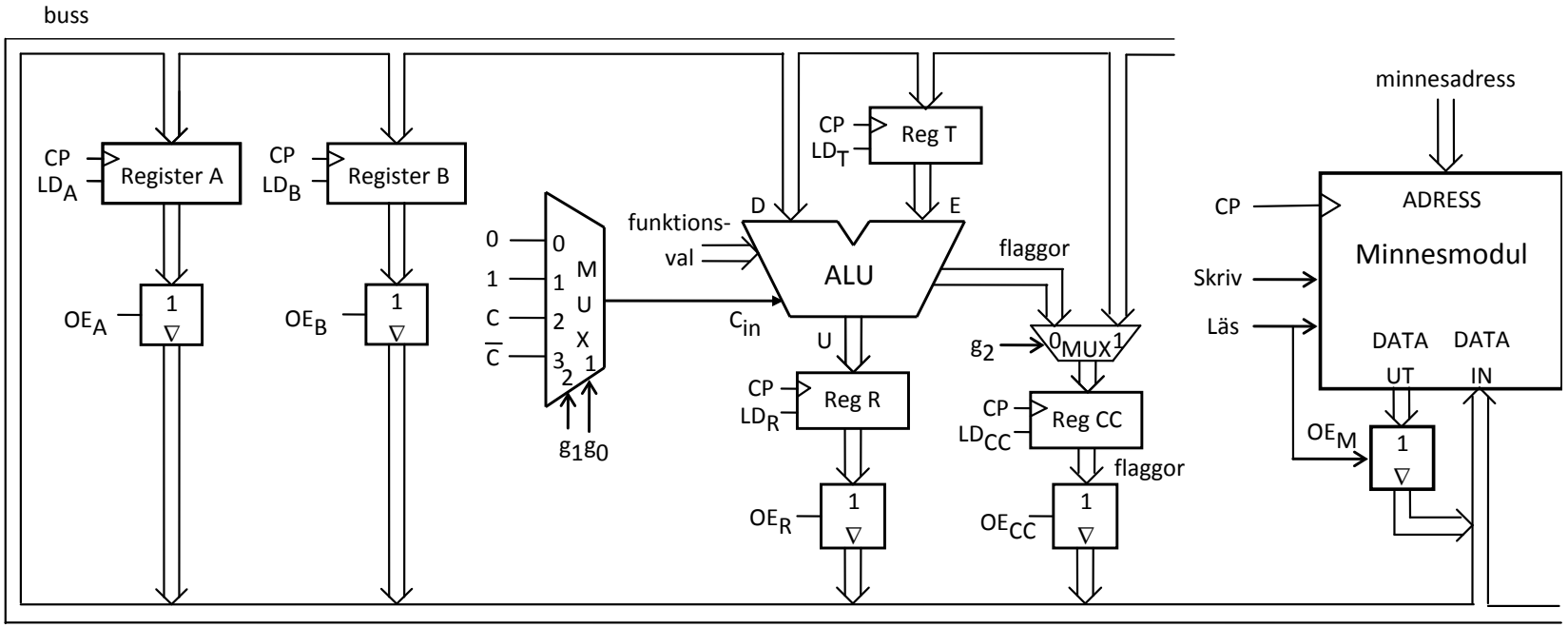
- "**+**" och "**–**" i tabellen avser **aritmetiska operationer**

- med $\mathbf{D_{1k}}$ menas att samtliga bitar i **D** inverteras

$f_3f_2f_1f_0$	U = f(D,E,C_{in})
0 0 0 0	0
0 0 0 1	D
0 0 1 0	E
0 0 1 1	D_{1k}
0 1 0 0	E_{1k}
0 1 0 1	D OR E
0 1 1 0	D AND E
0 1 1 1	D XOR E
1 0 0 0	D + C_{in}
1 0 0 1	D – 1 + C_{in}
1 0 1 0	D + E + C_{in}
1 0 1 1	D + D + C_{in}
1 1 0 0	D – E – 1 + C_{in}
1 1 0 1	0
1 1 1 0	0
1 1 1 1	FFH



Figur 7.30 Dataväg med register och ALU.



Figur 7.32 Dataväg med RWM.

7.3 Gör för respektive operation en tabell med tabellhuvudet:

Klock-cykel	**OE_A**	**OE_B**	**OE_R**	**LD_A**	**LD_B**	**LD_T**	**LD_R**	**LD_{CC}**	**g2**	**g1**	**g0**	**ALU-funktion**	**RTN**

och fyll sedan i styrsignalernas värden för de klockcykler som behövs för att verkställa operationen.

Tag inte hänsyn till eventuellt overflow.

Kolumnen **Klockcykel** längst till vänster skall ange klockcykelns nummer.

ALU-funktion skall ange funktionen som ALU:n skall utföra.

Kolumnen **RTN** skall ge RTN beskrivningen för deloperationerna på aktuell rad.

a) Innehållet i register A skall användas för att göra bitvis-OCH-operation med innehållet i register B. Resultat skall placeras i register B.

RTN: A AND B $\rightarrow$ B, Innehållet i register A får inte ändras.

e) RTN: $5 \cdot B - 3 \cdot A \rightarrow A$, flaggor $\rightarrow$ CC, register B får inte ändras.
Diskutera flaggbildningen.

7.4 Gör för respektive operation en tabell med tabellhuvudet:

Klock-cykel	OE_A	OE_B	OE_R	LD_A	LD_B	LD_T	LD_R	LD_{CC}	g2	g1	g0	ALU-funktion	Läs	Skriv	Minnes-adress	RTN

Kolumnen Klockcykel längst till vänster skall ange klockcykelns nummer.

Kolumnerna Läs och Skriv skall ange när läsning eller skrivning skall utföras i minnet.

Vid läsning i minnet (Läs=1) läggs minnesinnehållet ut på bussen på samma sätt som ett registerinnehåll.
Skrivning i minnet (Skriv = 1) verkställs vid positiv klockpulsflank.

I kolumnen ALU-funktion skrivs vilken operation som avses t ex D+E+Cin.

Val av C_{in} som 0, 1 eller C-värdet från register CC anges i kolumnerna g_1 och g_0.

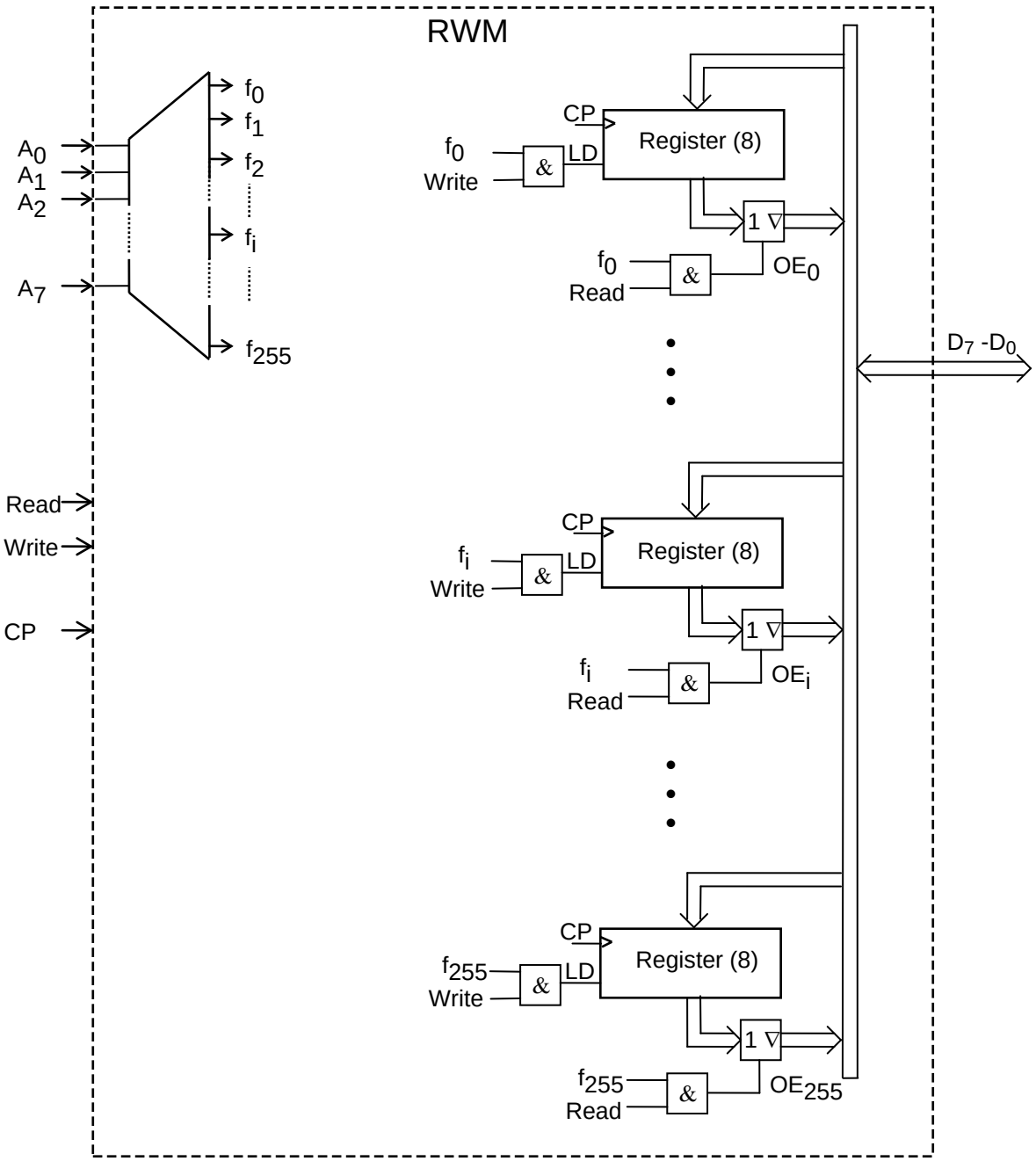
Det får förutsättas att summan ryms i 16 bitar. Innehållen i register A och B får inte ändras. 16-bitars tal lagras i minnet med mest signifikant del på den lägsta av två närliggande adresser.

Tecknet ":" i RTN-beskrivningen A:B betyder att innehållen i register A och B betraktas som ett 16-bitars tal, dvs skrives ihop till ett 16-bitars tal.

a) Innehållen i register A och B betraktas tillsammans som ett 16-bitars tal med den mest signifikanta delen i register A. Detta 16-bitars tal skall adderas till ett annat 16-bitars tal som finns i minnet på adresserna 20H och 21H. Resultatet (16 bitar) skall läggas på adresserna 22H och 23H.

RTN: A:B + M(20H):M(21H) $\rightarrow$ M(22H):M(23H), flaggor$\rightarrow$ CC

Läs- och skrivbart minne (RWM) 256∗8



5.1 Räknare

En räknare är ett antal sammankopplade vippor som registrerar antalet inkommande klockpulser.

Med n st vippor kan maximalt 2^n st klockpulser räknas.

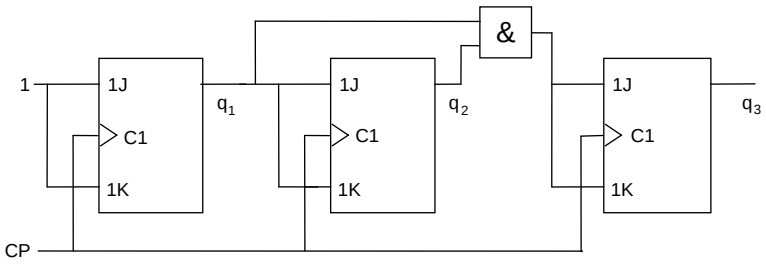
Antalet klockpulser registreras i form av n-bitars kodord.

En räknare som registrerar klockpulser från 0 till m-1 kallas för en modulo-m räknare.

Om antalet inkommande klockpulser överstiger m-1 så kommer klockpulserna med numren m, m+1, ... att registreras som pulser med numren 0, 1, ... (jämför med vägmätaren i en bil).

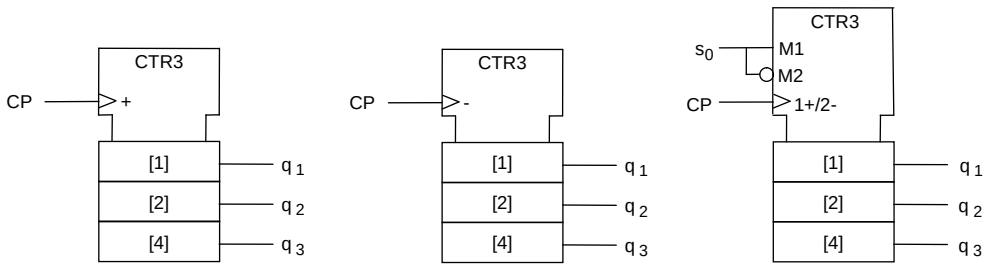
I detta avsnitt skall ett antal vanliga räknare visas.

I figur 5.40 visas kopplingen för en 3-bitars (upp)räknare som räknar med naturlig binärkod (NBC), dvs. sekvensen
$q_3q_2q_1$: 000, 001, 010, 011, 100, 101, 110, 111, 000, …



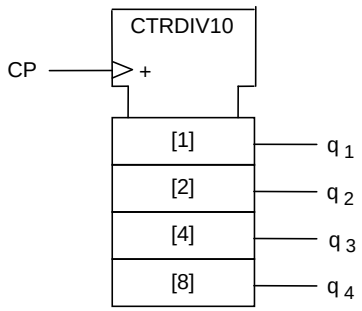
Figur 5.40 3-bitars binärräknare

Symbolen för räknaren visas till vänster i figur 5.46, tillsammans med två andra räknare för nedräkning, respektive reversibel räkning.



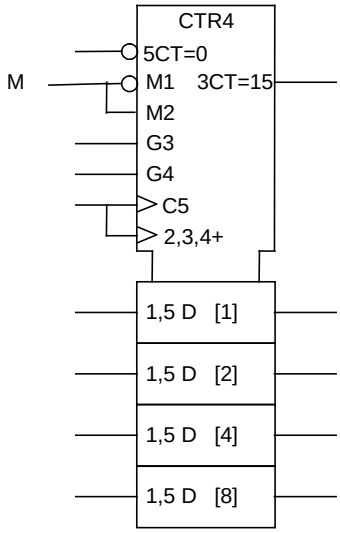
Figur 5.46 Symboler för 3-bitars binärräknare som räknar a) uppåt (+), b) nedåt (-) och c) uppåt eller nedåt (1+/2-).

Figur 5.47 visar symbolen för en dekadräknare med sekvensen $q_4q_3q_2q_1$: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 0000, …, som alltså räknar i NBCD-kod (0 – 9) och börjar om på 0 igen efter 9.



Figur 5.47 Symbol för dekadräknare.

Figur 5.50 visar symbolen för en 4-bitars räknare som räknar i naturlig binärkod $q_4q_3q_2q_1$: $0000_2 – 1111_2$, dvs. (0 – 15). Räknaren är konstruerad så att man enkelt kan utöka antalet bitar i steg om 4 bitar genom att ansluta nya likadana räknare. Denna räknare kommer vi att använda i styrenheten till den processor som vi har börjat konstruera.

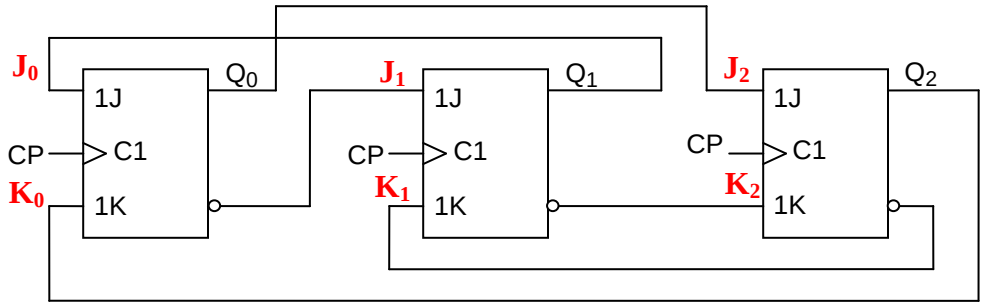


Figur 5.50 PROSAM-symbol för en ofta använd binärräknare.

Analys av synkrona räknare med JK- och SR-vippor

5.13

I figuren visas kopplingen för en räknare. Rita en tillståndstabell och en tillståndsgraf med tillstånden numrerade ($Q_2Q_1Q_0$). Rita också tidsdiagram för Q_2 - Q_0 om alla vipporna har Q = 0 från början.

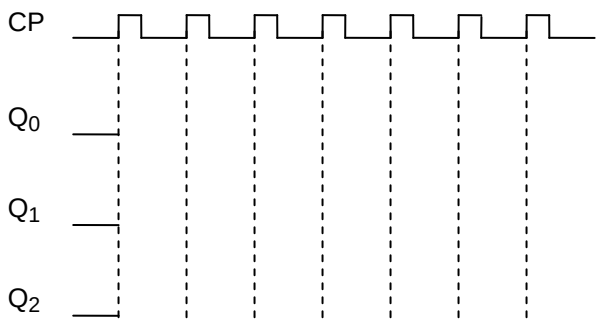


Lösning: J_2 = K_2 = J_1 = K_1 = J_0 = K_0 =

Q_2	Q_1	Q_0	J_2	K_2	J_1	K_1	J_0	K_0	Q_2^+	Q_1^+	Q_0^+
0	0	0									
0	0	1									
0	1	0									
0	1	1									
1	0	0									
1	0	1									
1	1	0									
1	1	1									

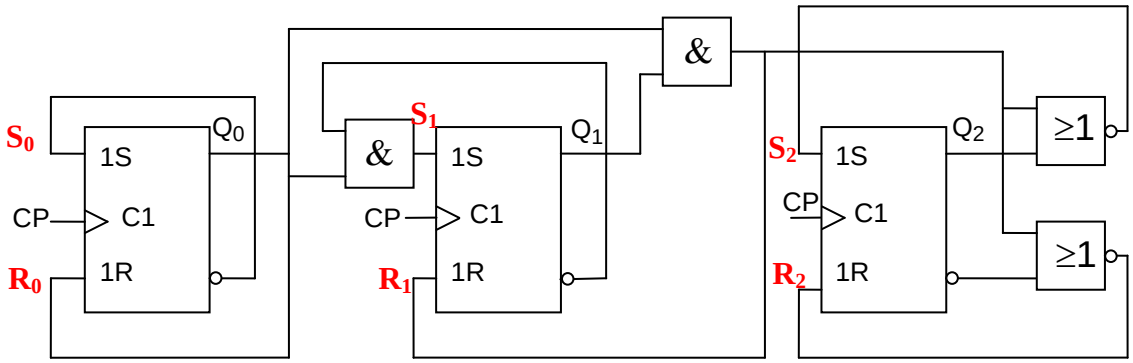
J	K	Q^+
0	0	
0	1	
1	0	
1	1	

$Q_2Q_1Q_0$



5.14

I figuren visas kopplingen för en räknare. Rita en tillståndstabell och en tillståndsgraf med tillstånden numrerade ($Q_2Q_1Q_0$). Rita också tidsdiagram för Q_2 - Q_0 om alla vipporna har Q = 0 från början.



Lösning: S_2 =

R_2 =

S_1 =

R_1 =

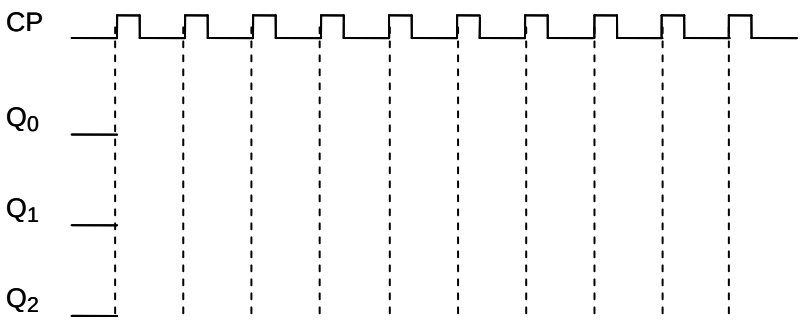
S_0 =

R_0 =

Q_2	Q_1	Q_0	S_2	R_2	S_1	R_1	S_0	R_0	Q_2^+	Q_1^+	Q_0^+
0	0	0									
0	0	1									
0	1	0									
0	1	1									
1	0	0									
1	0	1									
1	1	0									
1	1	1									

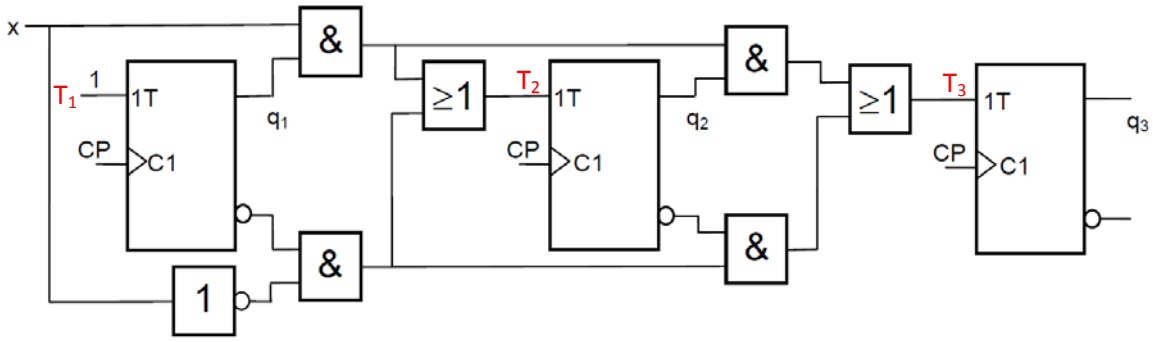
S	R	Q^+
0	0	
0	1	
1	0	
1	1	

$Q_2Q_1Q_0$



Analys av synkron räknare medT-vippor

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

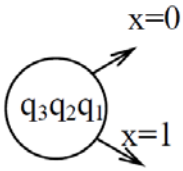


T_3 =

T_2 =

T_1 =

Rita tillstånden enligt:



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

T	Q^+
0	Q
1	Q'

Funktionstabeller för de olika vipptyperna:

D	q^+
0	0
1	1

S	R	q^+
0	0	q
0	1	0
1	0	1
1	1	?

J	K	q^+
0	0	q
0	1	0
1	0	1
1	1	q'

T	q^+
0	q
1	q'

Excitationstabeller för de olika vipporna:

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

q	q^+	S	R
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0

q	q^+	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

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

Konstruktion (syntes) av autonom synkron räknare

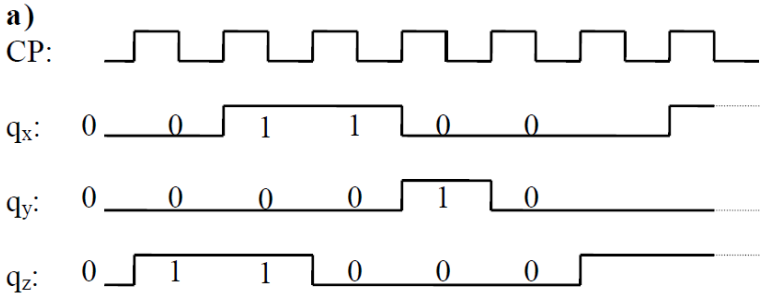
X.5 En autonom synkron räknare med räknesekvens enligt figuren nedan skall konstrueras. JK-vippor med positiv flanktriggning, NAND-grindar med valfritt antal ingångar samt INVERTERARE får användas. x, y och z i pulsdiagrammen nedan är utsignaler (q) från var sin vippa.

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

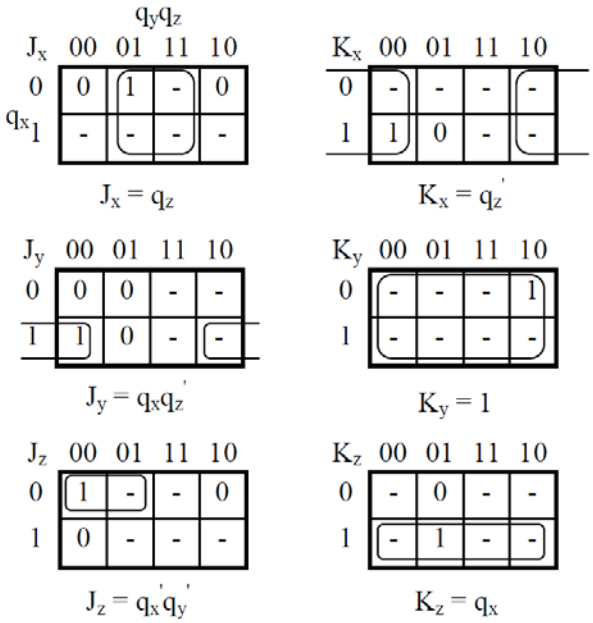
b) Realisera räknaren!

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

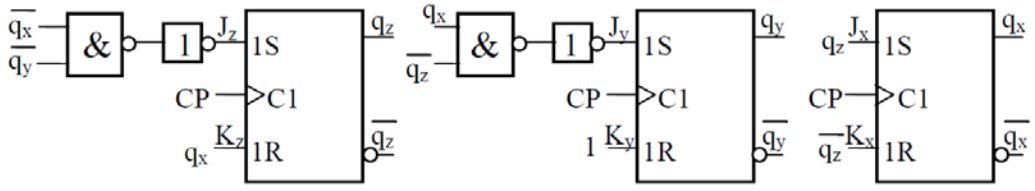
a)



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



b)



Tentaexempel på syntes av räknare

Tenta 2011-10-17 uppgift nr 3

b)

Realisera en autonom räknare med räknesekvensen $q_2q_1q_0$:

000, 010, 100, 110, 101, 011, 001, 000.

T-vippor, NAND-grindar med valfritt antal ingångar och NOT-grindar får användas.

Tenta 2010-12-13 uppgift nr 3

b)

Realisera en räknare med räknevillkoret x och räknesekvensen q_1q_0:

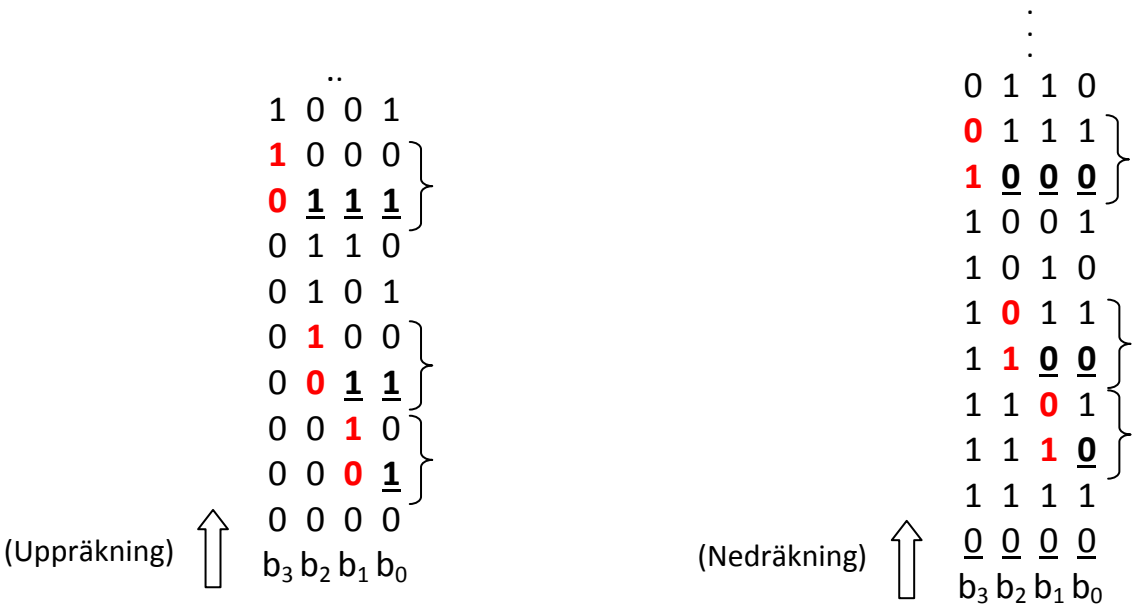
x = 0: 00, 10, 01, 11, 00, …
x = 1: 00, 11, 01, 10, 00, …

T-vippor, NAND-grindar med valfritt antal ingångar, XOR-grindar och NOT-grindar får användas.

Användning av T-vippan vid konstruktion av binärräknare

T-vippor passar utmärkt när man skall konstruera räknare för naturlig binärkod eftersom man kan ta fram enkla villkor för när en bit skall ändra värde, dvs när T = 1.

Exempel: 4-bitars räknare



Vid binär uppräkning skall alla bitar till höger i talet ha värdet 1 för att en given bit skall ändras, dvs bit b_i ändras om bitarna $b_{i-1} = b_{i-2} = \ldots = b_0 = 1$. Se tabellen till ovan vänster.

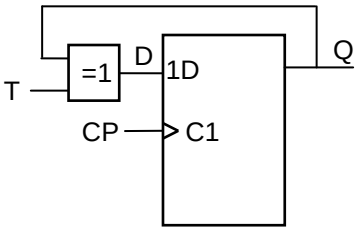
Vid binär nedräkning skall alla bitar till höger i talet ha värdet 0 för att en given bit skall ändras, dvs bit b_i ändras om bitarna $b_{i-1} = b_{i-2} = \ldots = b_0 = 0$. Se tabellen till ovan höger.

Om man använder D-vippor kan man "konvertera" dem till T-vippor enligt beskrivningen nedan.

T	Q	Q^+	D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

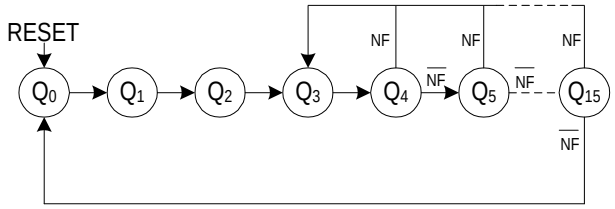
T \ D Q	0	1
0	0	1
1	1	0

$D = T'Q + TQ' = T \oplus Q$



Konstruktion av en 4-bitars räknare för styrenheten i FLEX-eller FLIS-processorn

Räknaren som bestämmer tillståndet hos styrenheten skall ha tillståndsgrafen nedan:



RESET är en yttre styrsignal som skall nollställa räknaren.

NF är en styrsignal som skall sätta räknarvärdet till $0011_2 = 3$.

Vi börjar med att konstruera en 4-bitars autonom räknare. (Räknesekvens: 0, 1, 2, …, 14, 15, 0, 1, 2, …) Tillstånden numreras $q_3q_2q_1q_0$. T-vippor är det naturliga valet enligt tidigare resonemang.

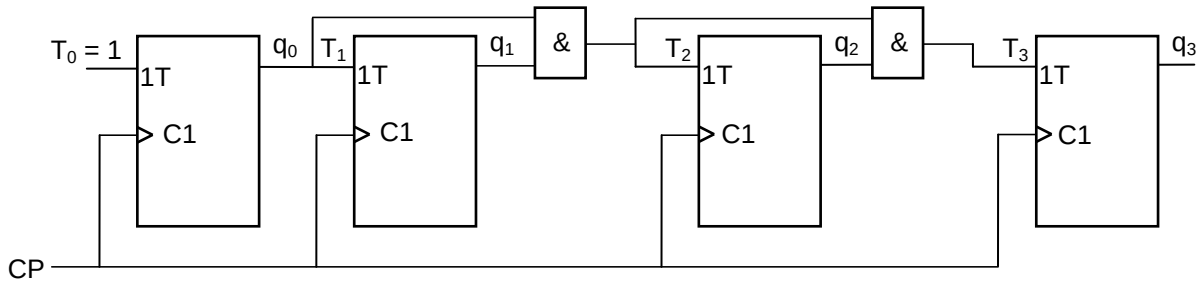
För T-vippor gäller att $q^+ = Q$ för $T = 0$ och $q^+ = Q'$ för $T = 1$. Q ändras alltså om $T = 1$.

Räknarens utsignaler : $q_3q_2q_1q_0$

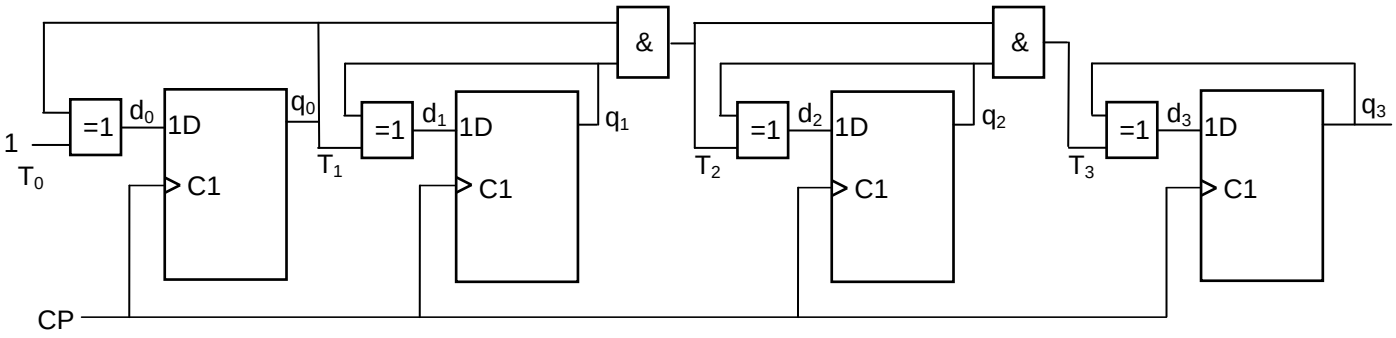
När man betraktar de binära värdena vid uppräkning ser man att bit q_0 alltid ändras efter klockpuls och att övriga bitar bara ändrar sig om alla bitar med lägre nummer är ettor.

Det innebär att följande uttryck fås för T: $T_0 = 1$; $T_1 = q_0$; $T_2 = q_1q_0$; $T_3 = q_2q_1q_0$

Omskrivning ger: $T_0 = 1$; $T_1 = q_0$; $T_2 = q_1T_1$; $T_3 = q_2T_2$



Vi byter ut T-vipporna mot D-vippor (D-vippa + XOR-grind, enligt tidigare) och får då kopplingen:

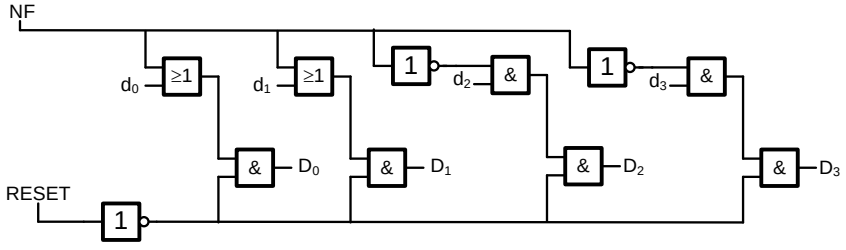


XOR-grinden längst till vänster, som bildar d_0, ersätts av en inverterare.

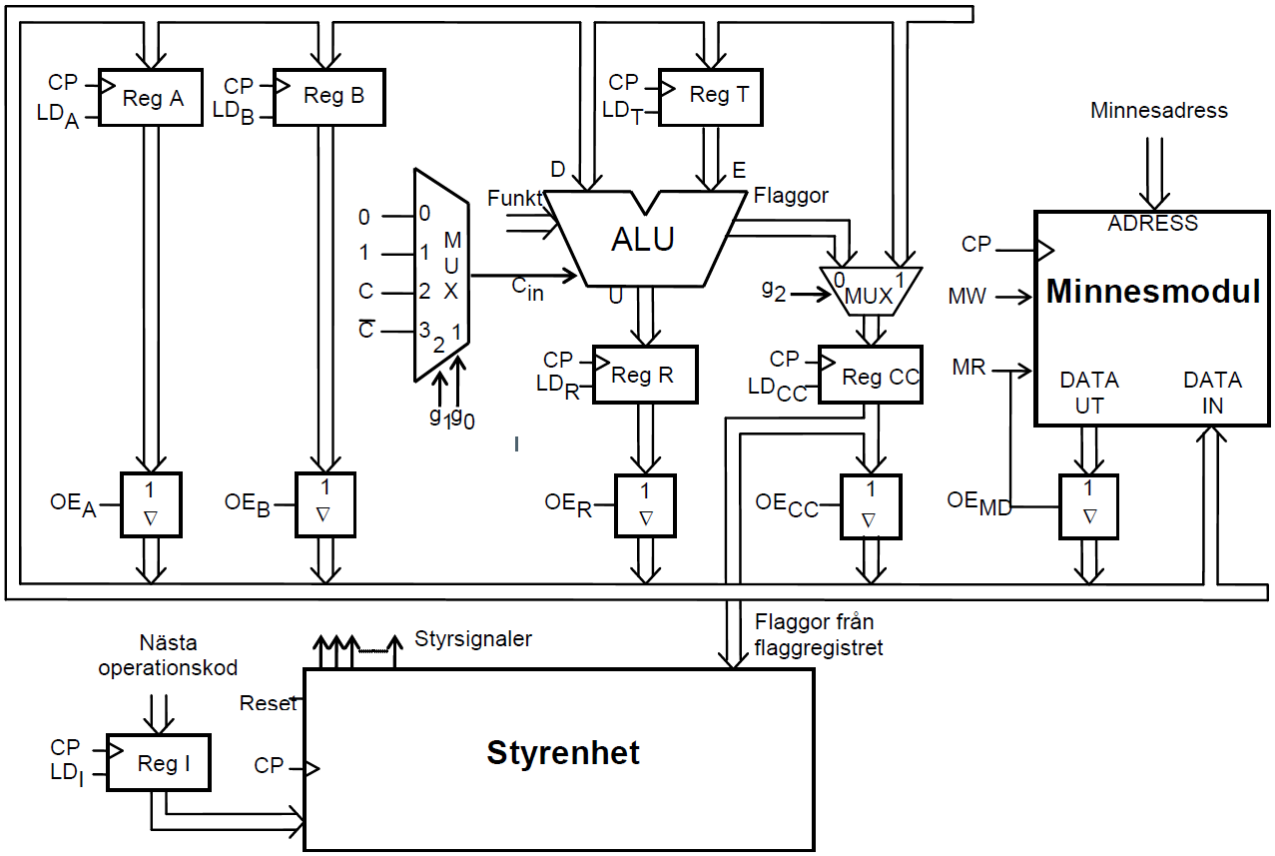
Vi kompletterar till sist räknaren med ett grindnät för synkron nollställning via insignalen RESET och med möjligheten att ladda värdet $q_3q_2q_1q_0 = 0011$ via insignalen NF, som står för New Fetch.

Utsignalerna d_3, d_2, d_1 och d_0 från XOR-grindarna ovan kopplas inte direkt till D-ingångarna på vipporna.

Vi kopplar istället in dem i kretsen nedan och kopplar sedan signalerna D_3, D_2, D_1 och D_0 till vippornas D-ingångar.



von Neumanndatorn (Ext-8)



Figur F.1 Databehandlande enhet med minne.

von Neumanns idé

"Man kan koda de olika operationerna som binära tal på samma sätt som data och lagra både operationer och data i minnet."

=

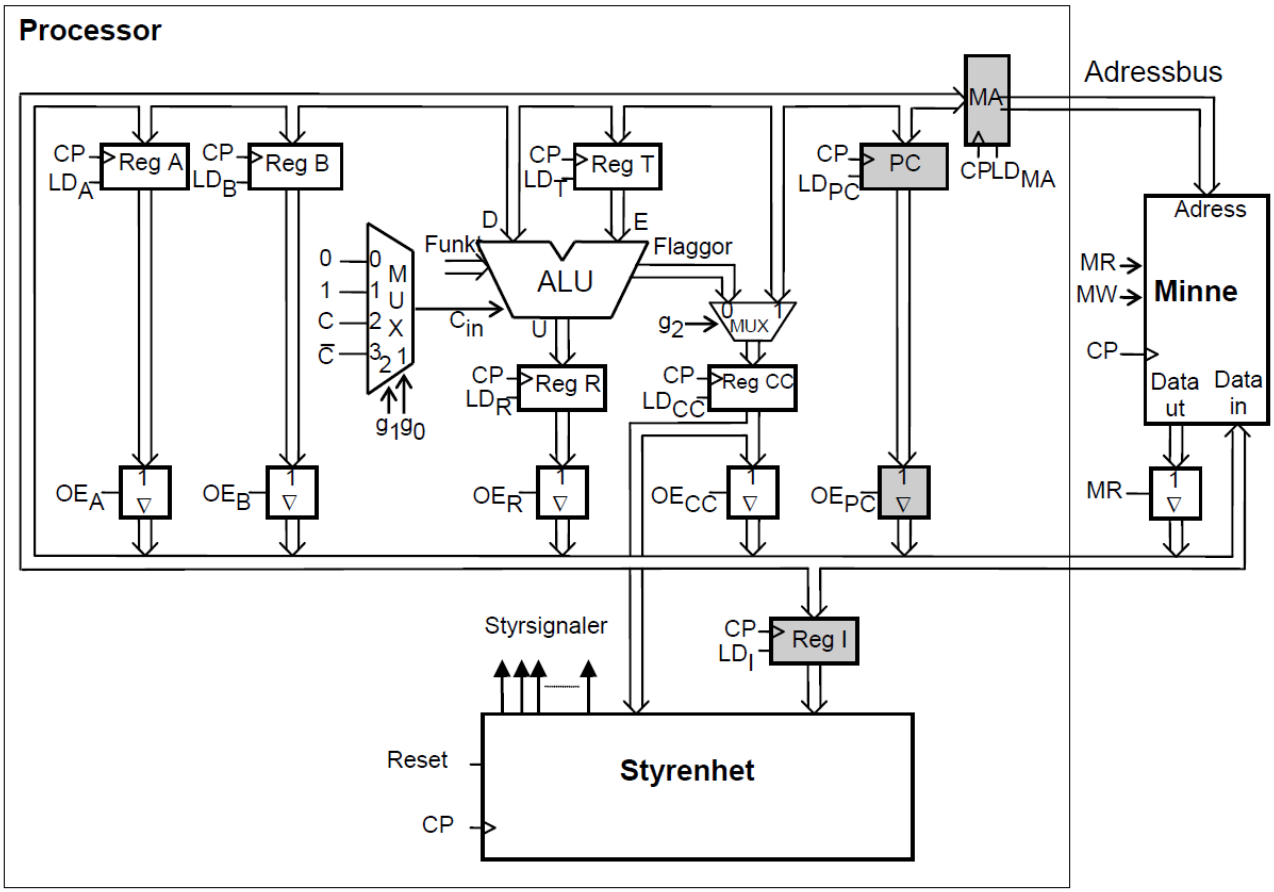
"Det lagrade programmets princip."

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

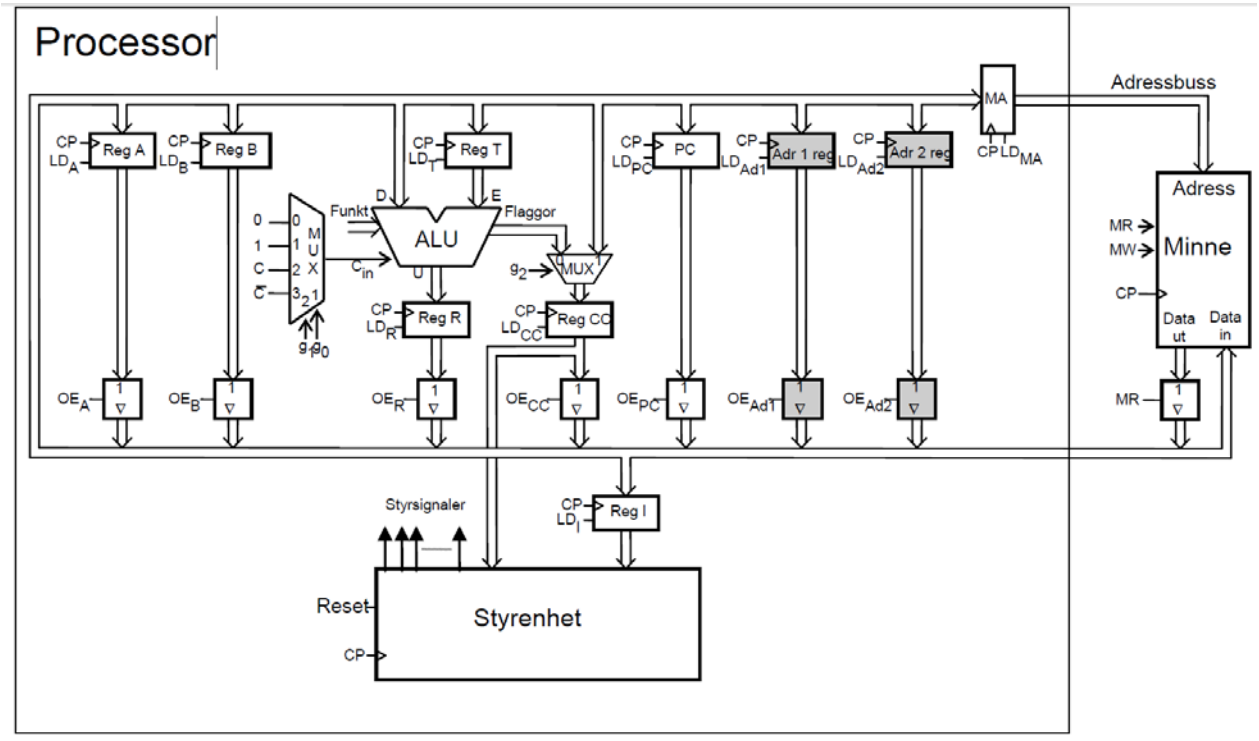
En instruktion är alltså kodad som ett binärt tal. En del av bitarna i detta tal är koden för en operation medan övriga bitar är en eller flera operander (data). En typisk instruktion består alltså av två delar enligt figuren nedan.

Operationskod	Operand(er)

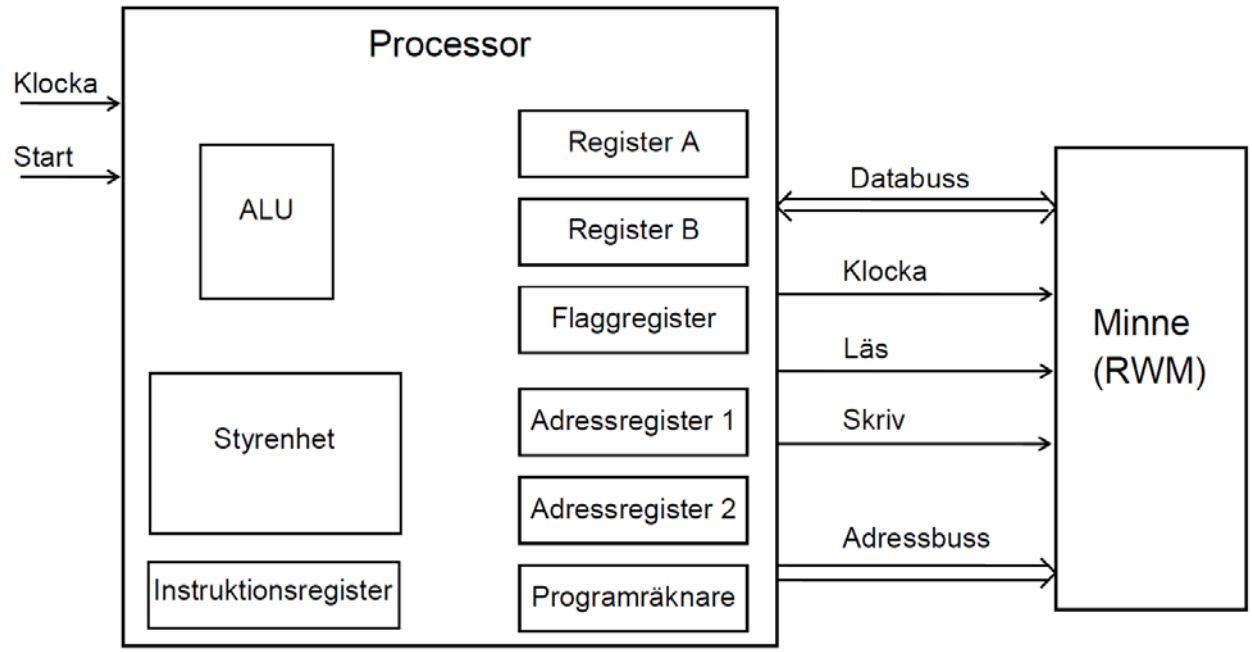
Operanden är antingen *data* (datavärdet) eller någon form av *adress till data*. Vissa instruktioner saknar dock operand och därmed också operanddel.



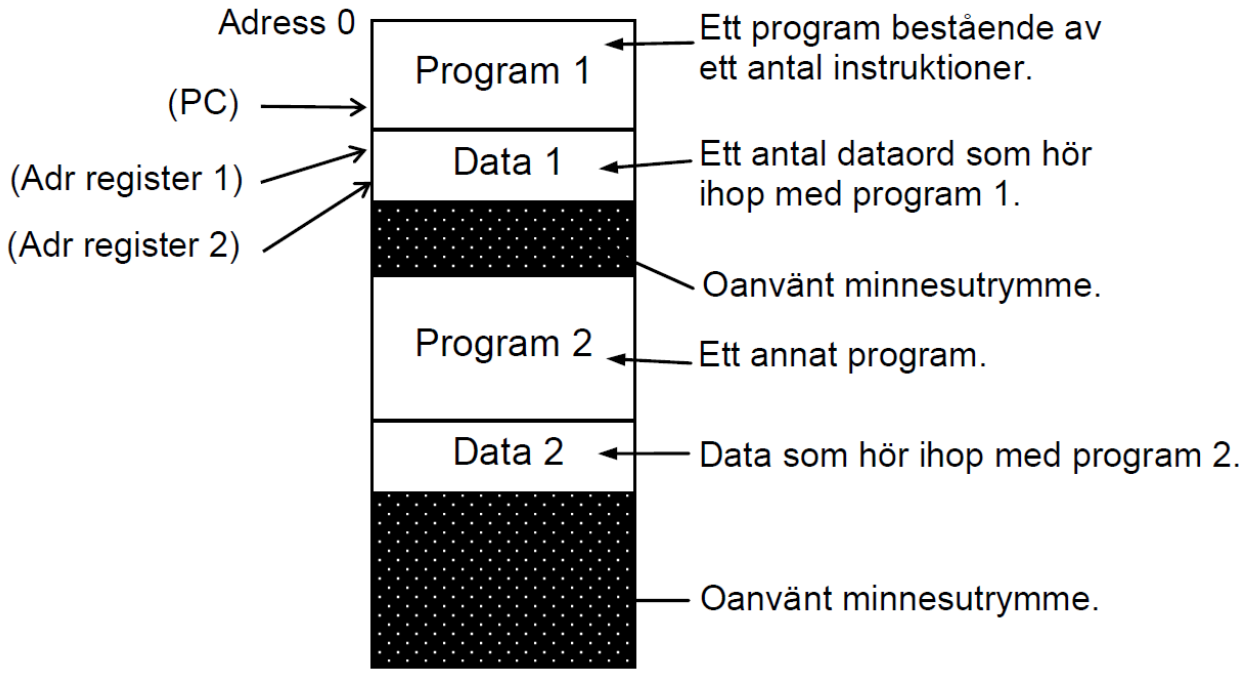
Figur F.2 Kombinationen von Neumannprocessor-minne.



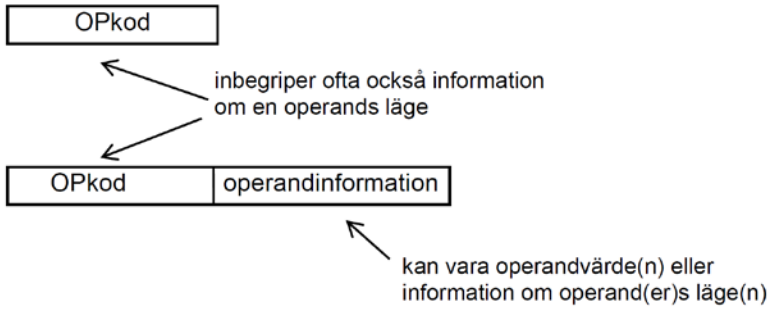
Figur F.3 Kombination av en utökad von Neumannprocessor och minne.



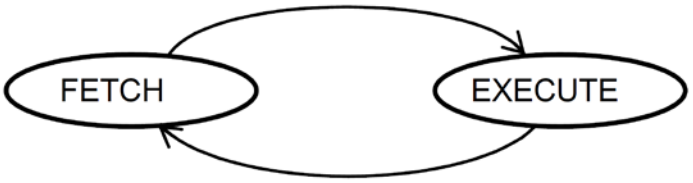
Figur F.4 Programmeringsmodell för enkel von Neumanndator.



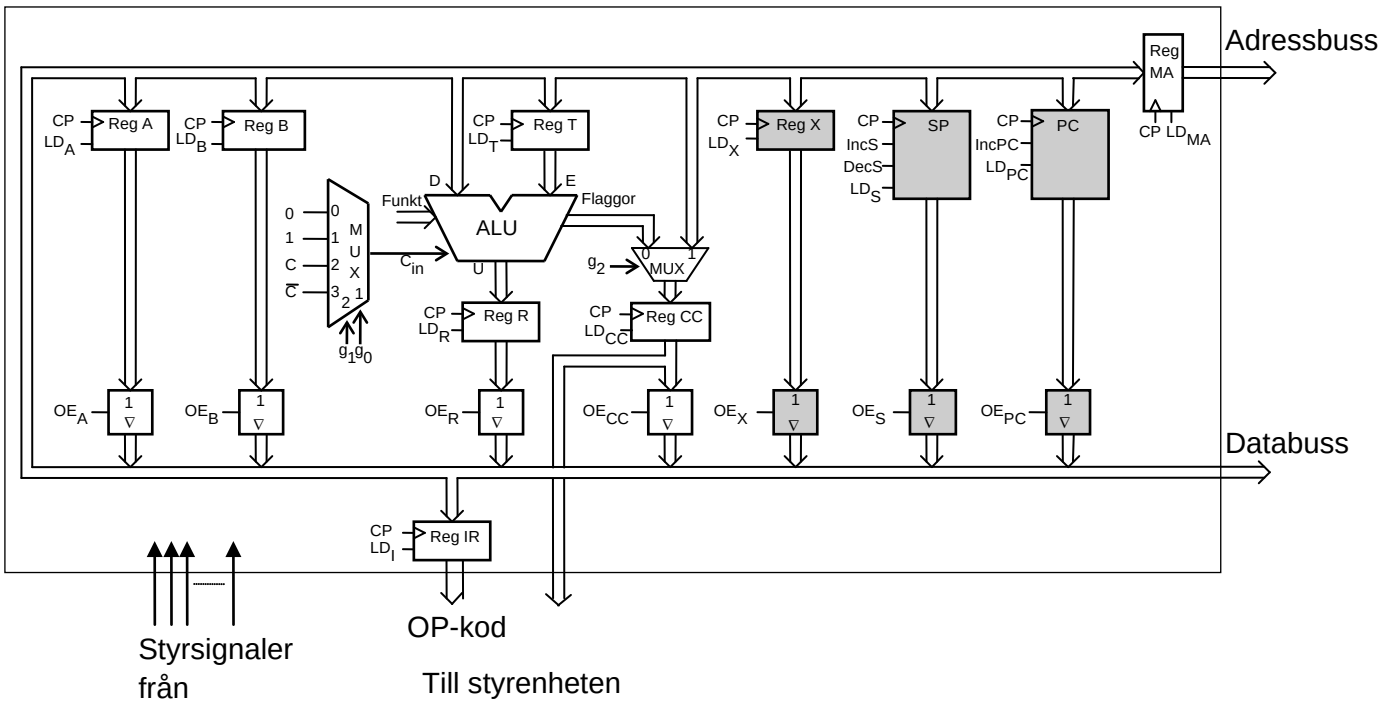
Figur F.5 Minnesdisposition för von Neumanndator.



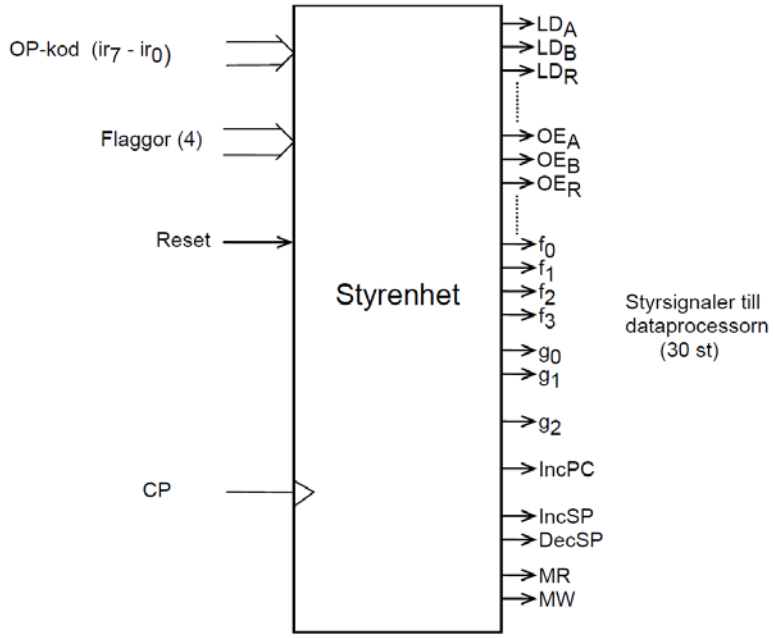
Figur F.6 Instruktionernas längd och innehåll följer ett sk instruktionsformat.



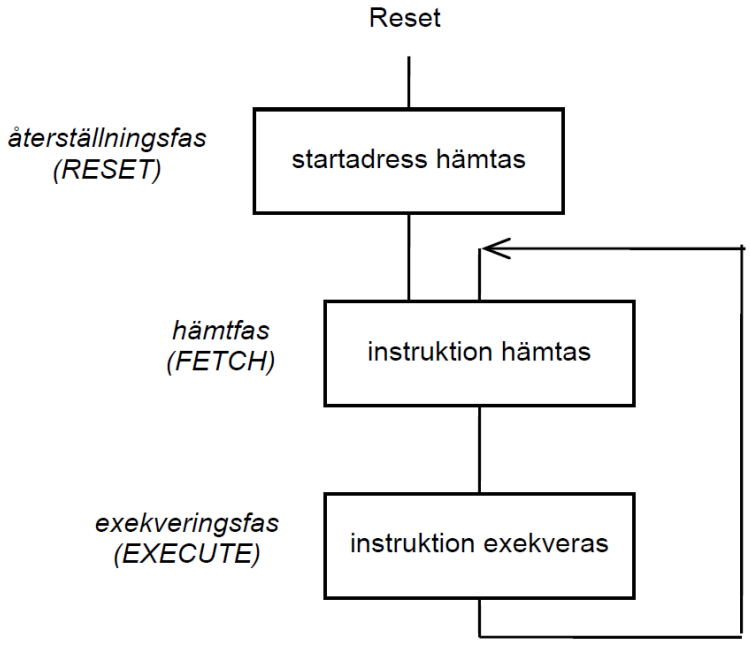
Figur F.7 Tillståndsgraf för instruktionernas två faser.



Figur F.8 FLEX-processorns dataväg.



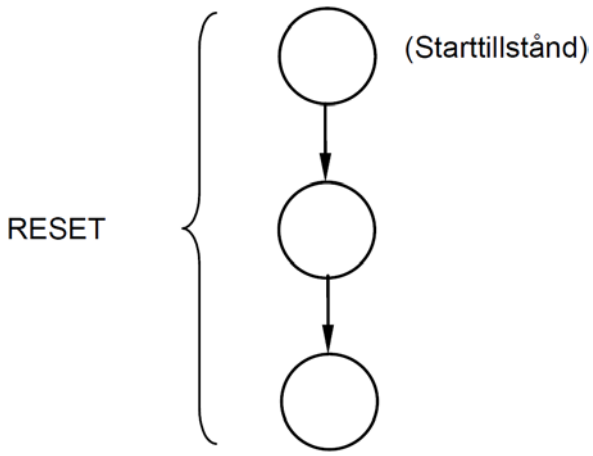
Figur F.9 FLEX-processorns styrenhet



Figur F.10 Instruktionscykeln kompletterad med s k återställningsfas.

Tabell F.1 Detaljerad beskrivning av RESET-fasen för FLEX-processorn

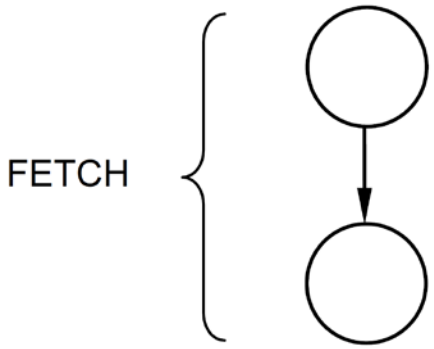
Klockcykel (State nr)	RTN-beskrivning	Styrsignaler	Kommentar
0	$FF_{16} \rightarrow R$	ALU-funktion = F_{16}, LD_R=1.	ALU-funktionen väljs så att talet FF_{16} finns på ALU:ns utgång. Laddingången på R-registret ettställs så att utvärdet från ALU'n (FF_{16}) laddas i R-registret vid nästa klockpuls.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



Figur F.11

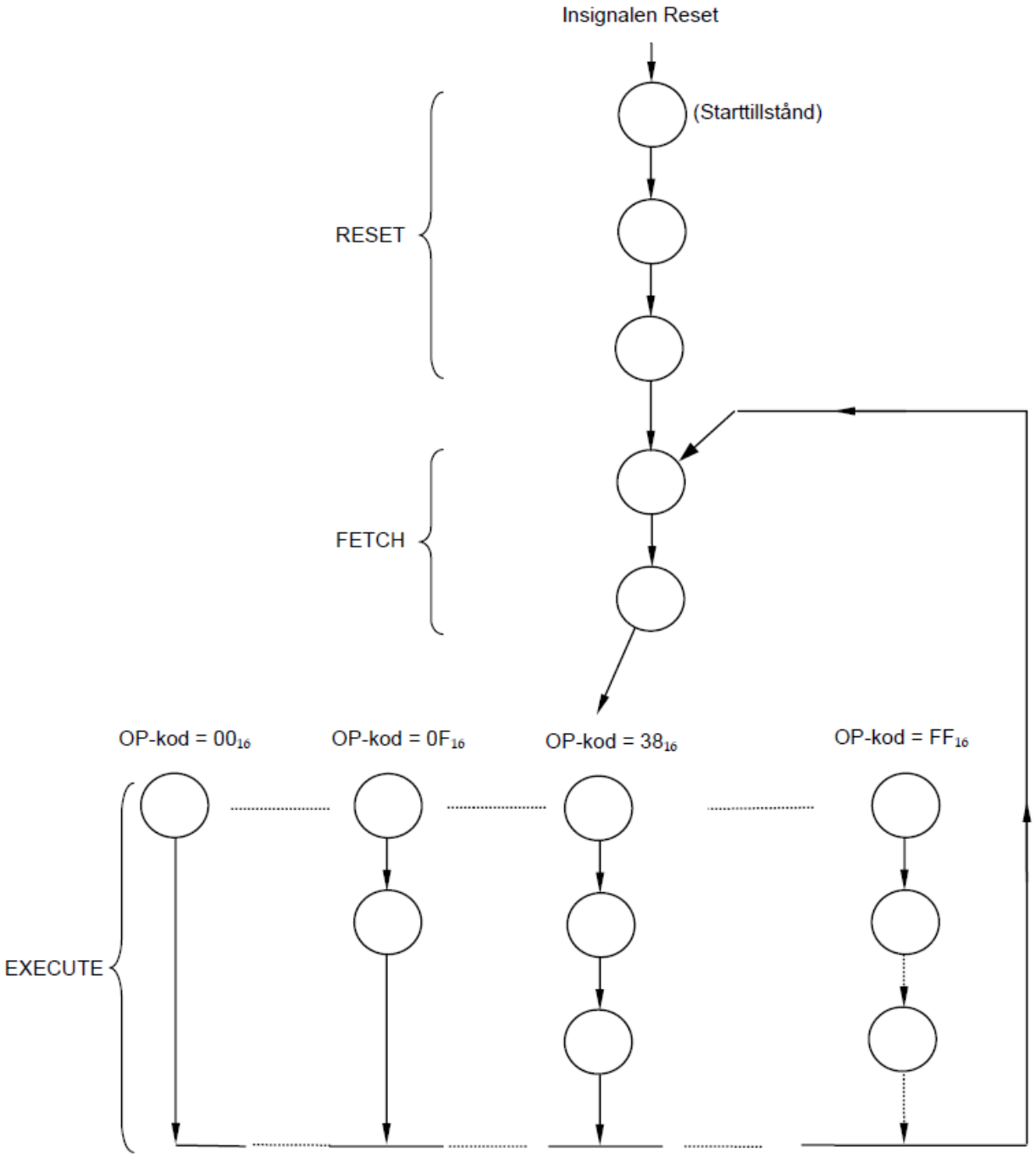
Tabell F.2 Detaljerad beskrivning av FETCH-fasen för FLEX-processorn

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

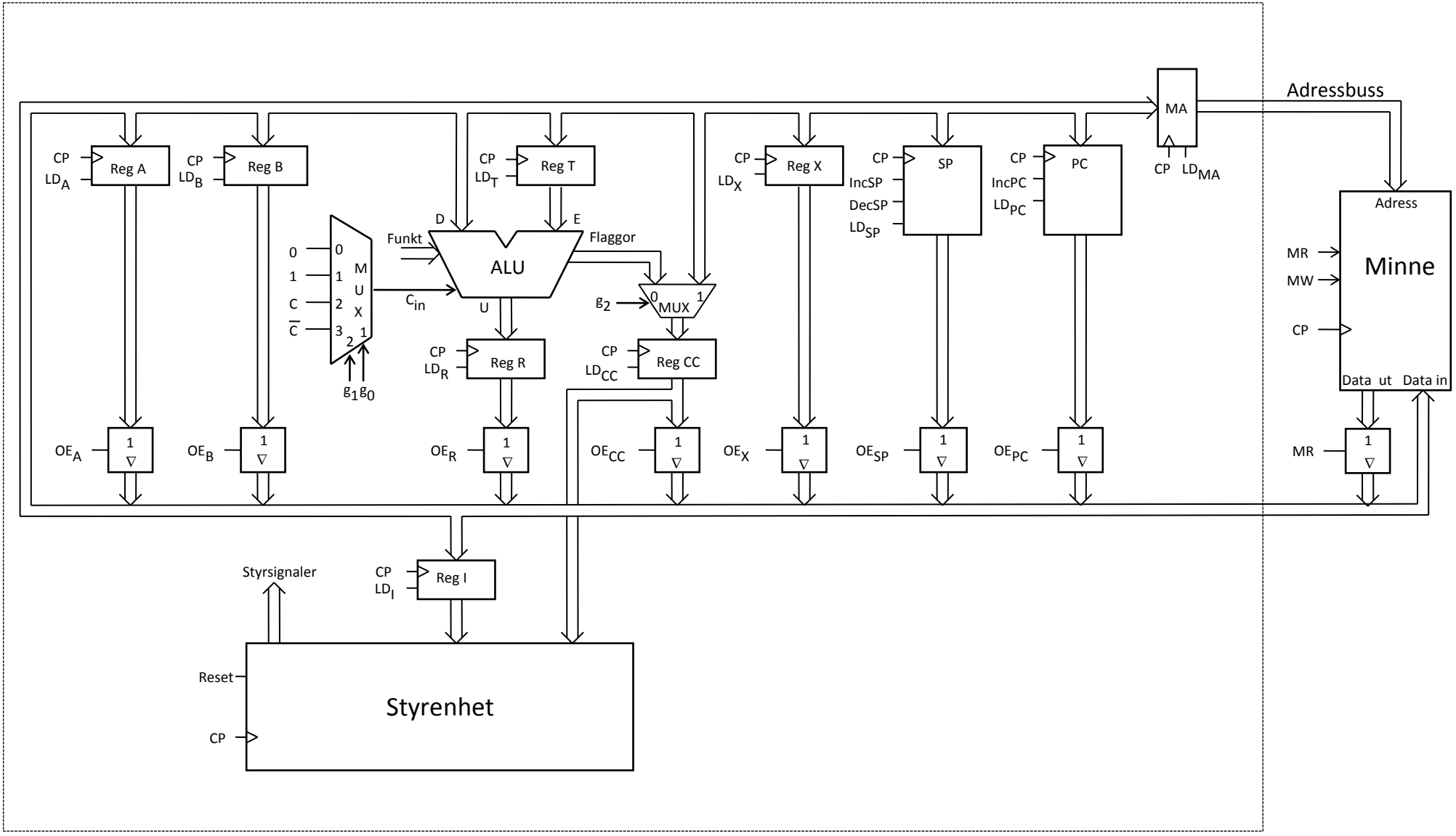


Figur F.12

Tillståndsgraf för FLEX-processorns arbetsfaser

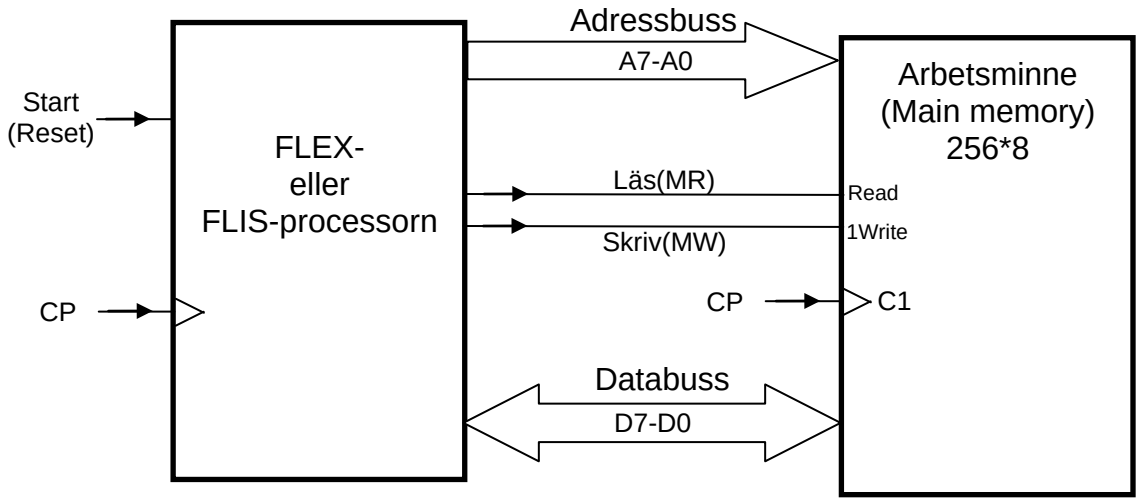


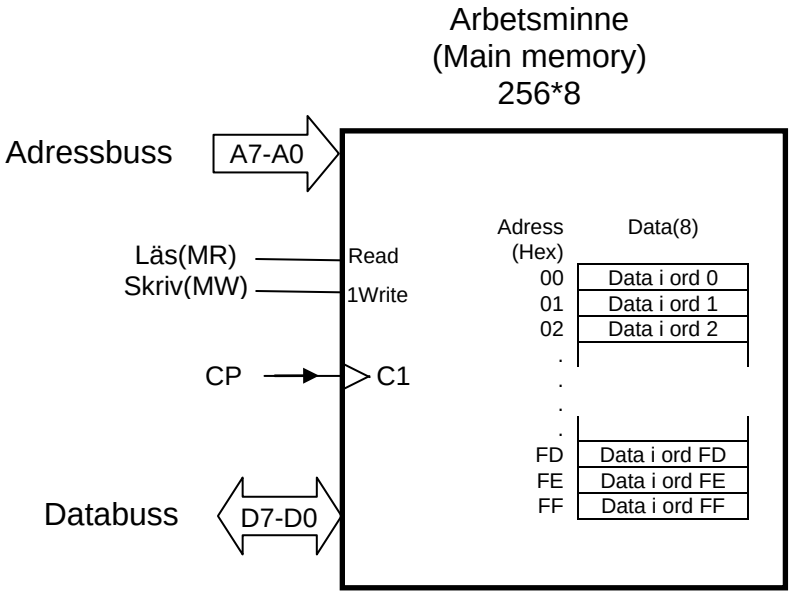
Figur F.13 Tillståndsgraf för styrenheten till FLEX-processorn.



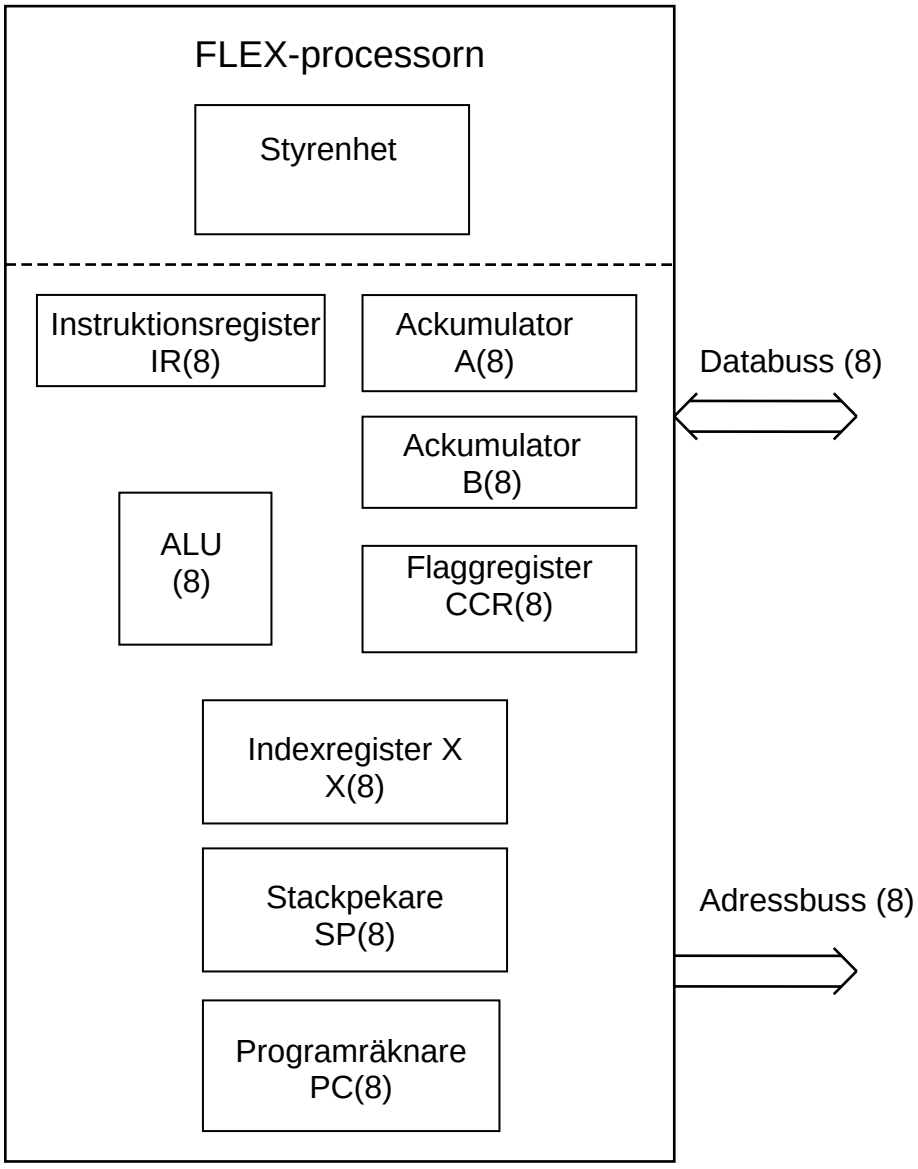
FLEX-datorn i detalj

FLEX- eller FLIS-datorn utifrån sett





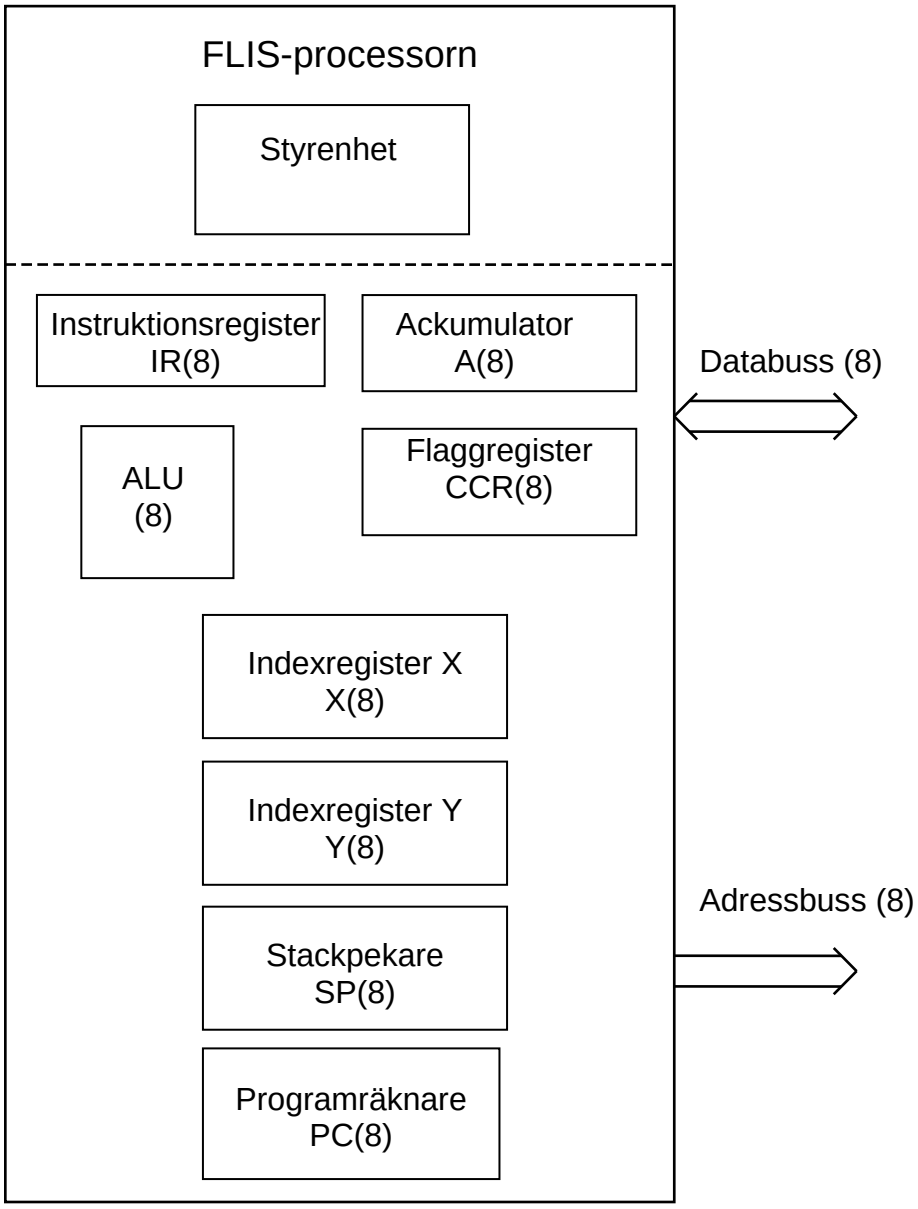
Programmerarens bild av FLEX-processorn



Flaggregister CCR

7	6	5	4	3	2	1	0
-	-	-	-	N	Z	V	C

Programmerarens bild av FLIS-processorn



Flaggregister CCR

7	6	5	4	3	2	1	0
-	-	-	I	N	Z	V	C

Maskinprogram i datorns minne

En del av ett datorprogram (ett maskinprogram på binär form) kan se ut som i figuren nedan när det hamnar i datorns minne. För en människa är väl detta närmast obegripligt.

Adress	Data
01111111	?
10000000	11110000
10000001	00000101
10000010	10100110
10000011	00110000
10000100	11100001
10000101	00001111
10000110	?

Man kan komprimera de binära värdena för adress och data genom att skriva dem på hexadecimal form enligt nedan, men de är fortfarande obegripliga. Vi kan dessutom inte avgöra om minnesinnehållet bara är datavärden som är lagrade i minnet eller om de är en del av ett program.

Adress	Data
$7F_{16}$	?
80_{16}	$F0_{16}$
81_{16}	05_{16}
82_{16}	$A6_{16}$
83_{16}	30_{16}
84_{16}	$E1_{16}$
85_{16}	$0F_{16}$
86_{16}	?

Man använder därför ett speciellt språk, assemblerspråk, för att göra maskinprogrammet begripligt för oss människor. I detta språk används mnemoniska (till stöd för vårt minne) beteckningar och symboler som förklarar instruktionernas och ibland även de använda adressernas och datavärdenas funktion.

Nedan visas programavsnittet ovan översatt till FLIS-processorns assemblerspråk.

```
 ?
LDA    #5    Ladda register A med talet 5
ADDA   48    Addera innehållet på minnesadress 48₁₀ till register A
STA    15    Lagra innehållet i register A på adress 15₁₀ i minnet
 ?
```

Vilka adresser instruktionerna finns på i minnet framgår dock inte av assemblerprogrammet i detta fall.

Övningsuppgifter

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

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

a) F1, 20, E1, 21, 96, 0F, A9, 25, 06, 97, 10.

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

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

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

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

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

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

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

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

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

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

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

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

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

F.11 a) "Jump"-instruktionen JMP $50 är placerad med början på adress 37_{16}.
Vad är dess maskinkod?

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

F.12 a) "Jump"-instruktionen JMP $05 är placerad med början på adress 20_{16}.
Vad är dess maskinkod?

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

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

```
Läge     Operation Operand        Kommentar
START    CLR       $FB
         LDA       #25
SCOUNT   DECA
         BNE       SCOUNT
```

a) Rita en maskinberoende flödesplan för sekvensen.

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

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

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

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

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

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

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

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

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

F.23 Skriv ett programavsnitt som läser ett tal från inport FB_{16}.

Om talet är 00_{16} skall en subrutin FALL1 anropas. Talet skall då ligga i A-registret för att subrutinen skall kunna använda det för bearbetning.

Om talet är 18_{16} skall på motsvarande sätt en subrutin FALL2 anropas.

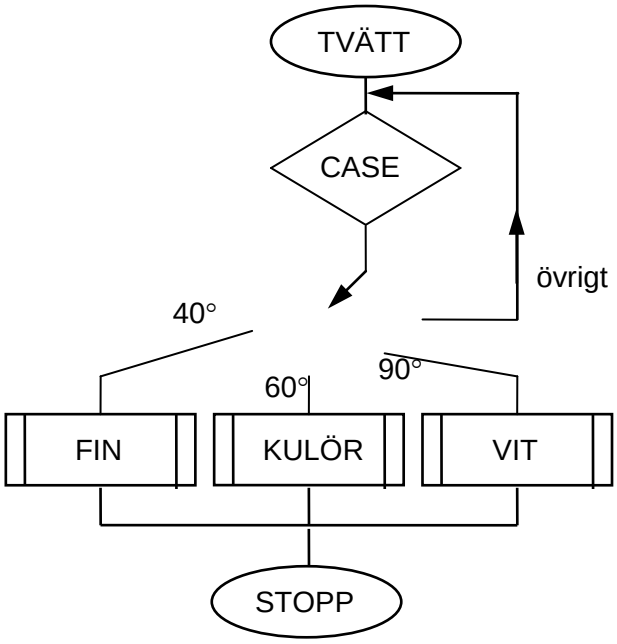
Om talet inte är lika med något dessa värden skall en subrutin FALL3 anropas.

Rita också en flödesplan.

F.25 En enkel datorstyrd tvättmaskin har tre olika tvättprogram

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

När tvättmaskinen startats agerar den enligt flödesplanen nedan.



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

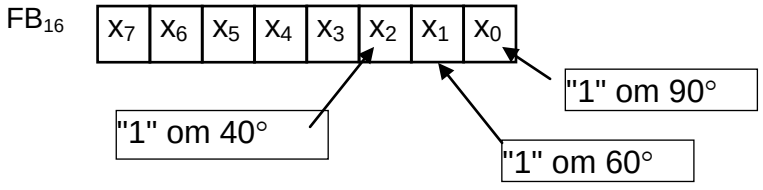
b) Användaren kan trycka på tre olika knappar som svarar mot de olika tvättprogrammen. Den valda tvättemperaturen är åtkomlig för processorn via inporten FB_{16} enligt figuren nedan.

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

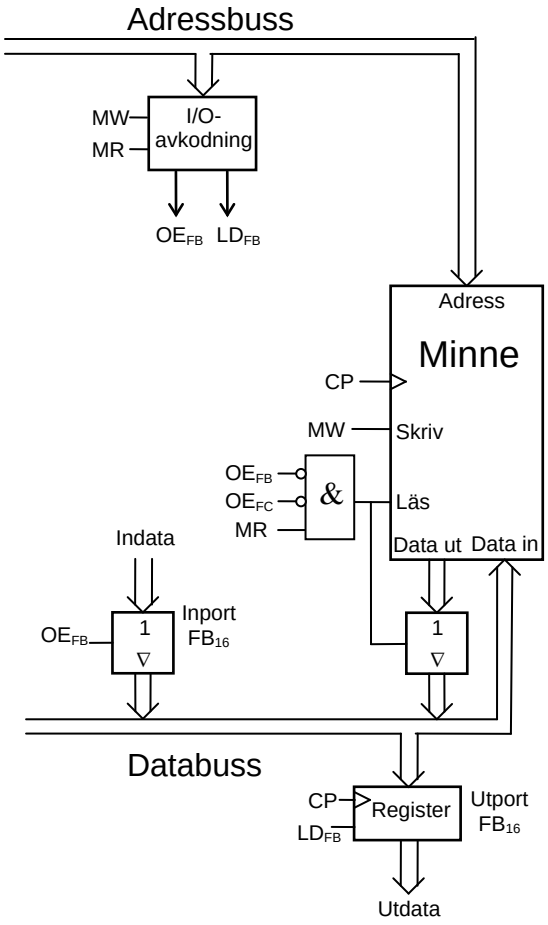
De olika tvättprogrammen är subrutiner med början på de symboliska adresserna "Fin", "Kulör" respektive "Vit". Omsätt den nya flödesplanen i ett program skrivet med FLISP instruktioner.

Ledning:

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



FLIS-dator med in- och utport inkopplad på adress- och databussen.



På adressen FC_{16} finns också en inport och en utport inkopplade enligt samma princip som portarna på adressen FB_{16}.

Beräkning av exekveringstid (körtid) för programsekvens med slingor

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

```
Läge      Operation  Operand        Kommentar
START     CLR        $FB
          LDA        #25
SCOUNT    DECA
          BNE        SCOUNT
          LDA        #%00000001
          STA        $FB
          LDA        #100
PCOUNT    DECA
          BNE        PCOUNT
          CLR        $FB
```

a) Rita en maskinberoende flödesplan för sekvensen.

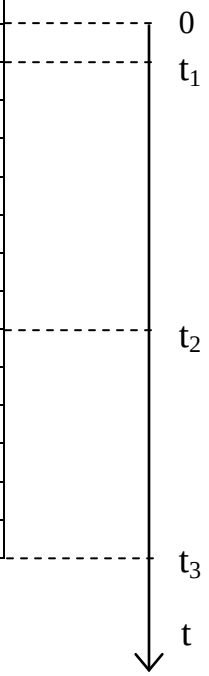
b) Antag att FLIS-processorn har klockfrekvensen 1 MHz och att sekvensen startas vid tidpunkten t = 0. När i tiden genereras pulsen? Hur lång är den?

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

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

Uppgift F.14 b) och d)

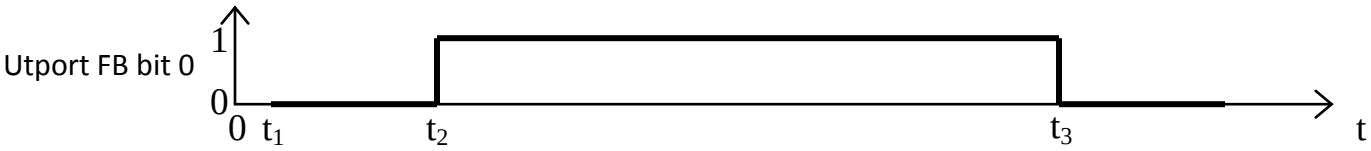
Läge	Operation	Operand	Maskinkod			Klock-cykler
			Adress (Hex)	Minnes-Innehåll (Hex)		
START	CLR	$FB	20	35	FB	3
;						
	LDA	#25	22	F0	19	2
SCOUNT	DECA		24	08		3
	BNE	SCOUNT	25	25	***FD***	4
;						
	LDA	#%00000001	27	F0	01	2
	STA	$FB	29	E1	FB	3
;						
	LDA	#100	2B	F0	64	2
PCOUNT	DECA		2D	08		3
	BNE	PCOUNT	2E	25	***FD***	4
;						
	CLR	$FB	30	35	FB	3



t_1 = 3

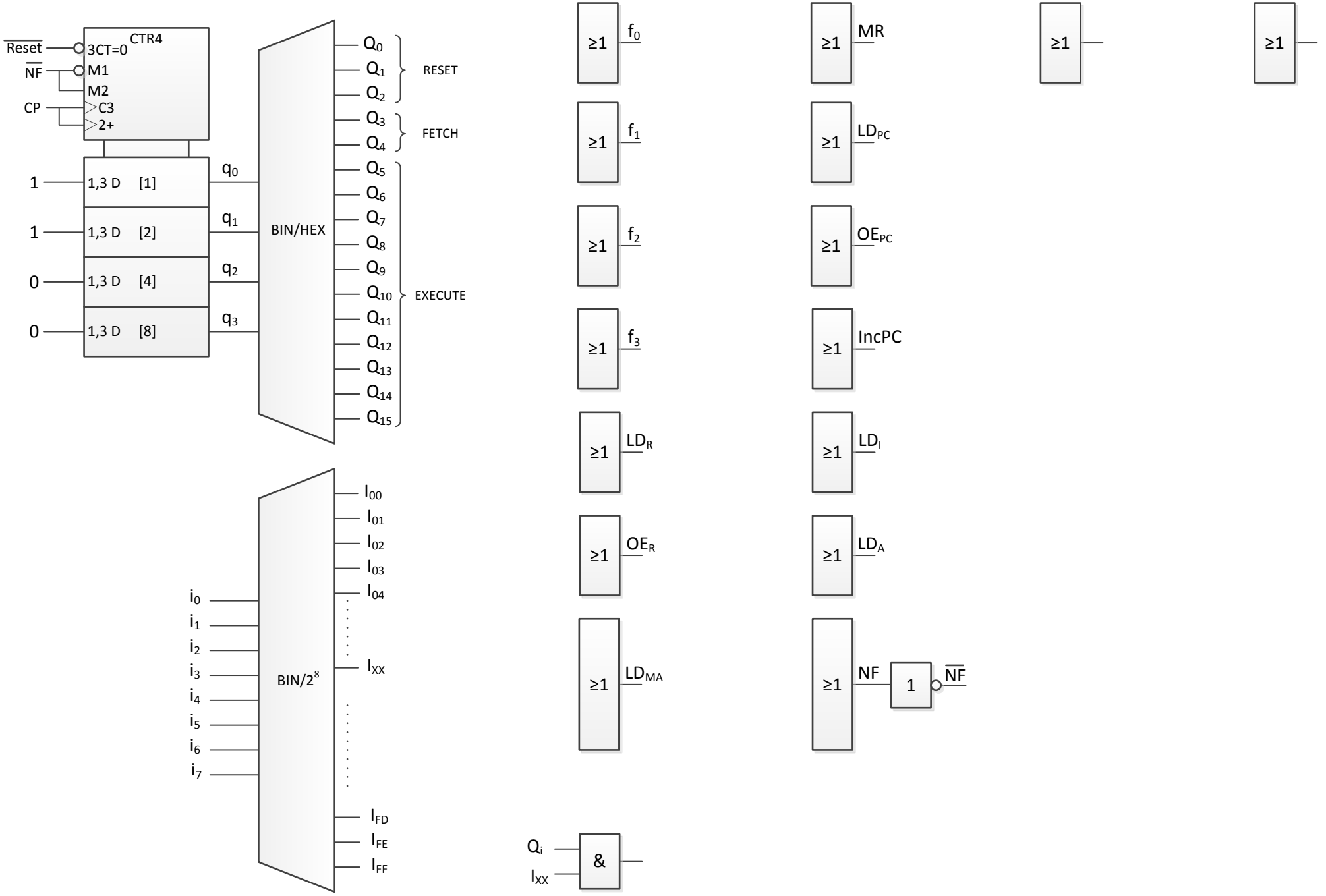
$t_2 = t_1 + 2 + (3 + 4)\cdot 25 + 2 + 3 = 3 + 182 = 185$

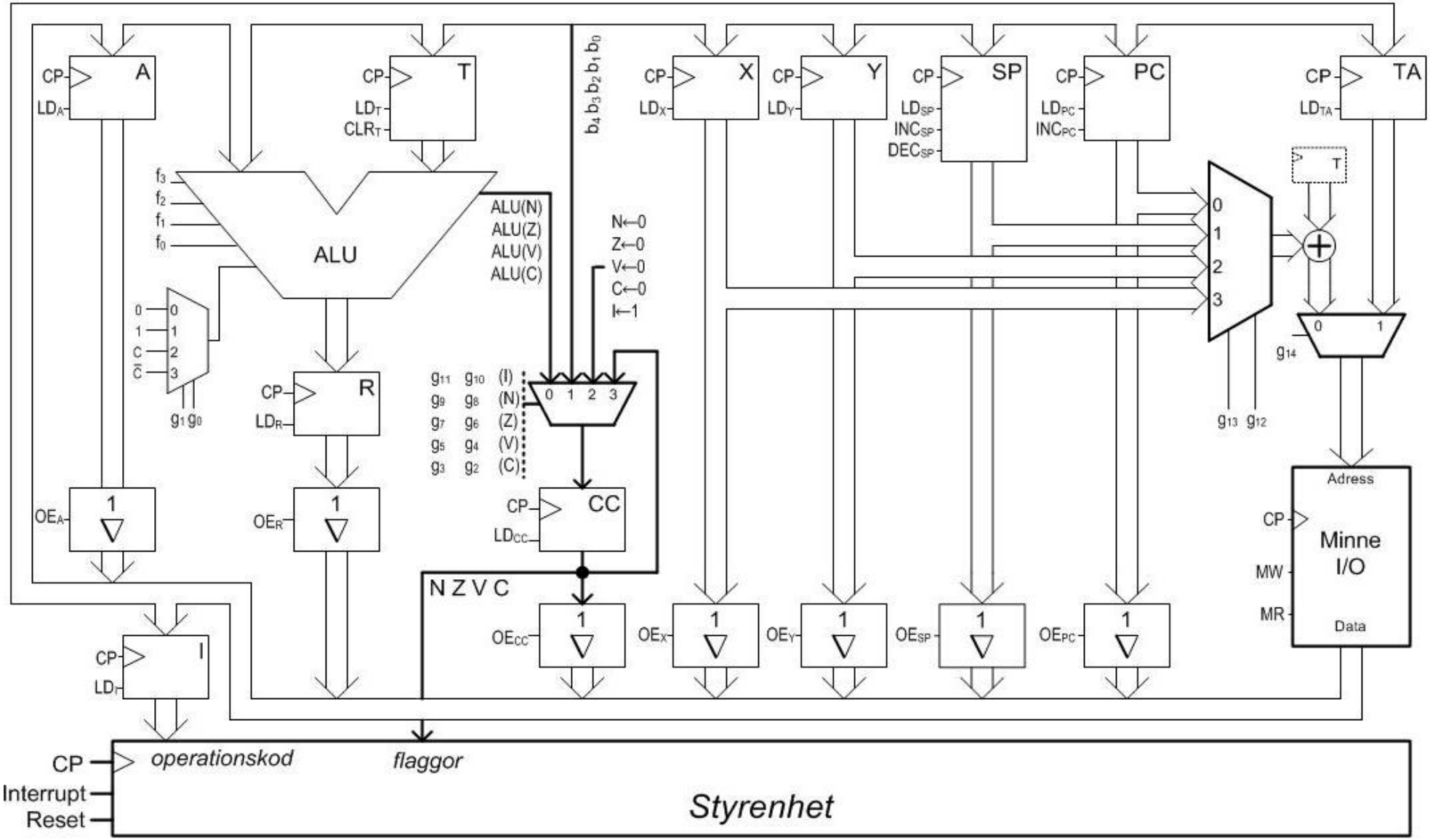
$t_3 = t_2 + 2 + (3 + 4)\ 100 + 3 = 185 + 705 = 890$



c)

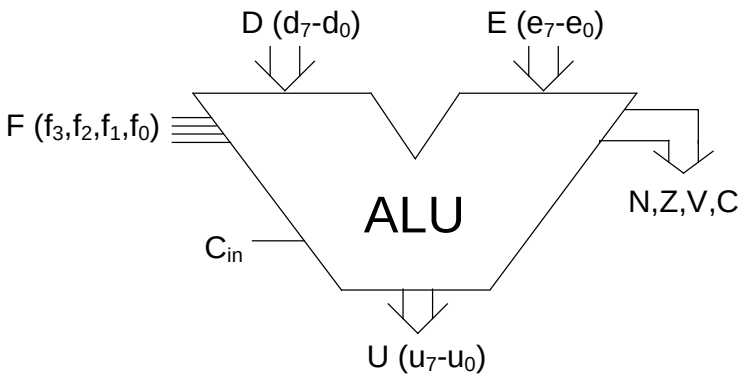
```
Läge        Operation   Operand      Kommentar
                                     Programavsnitt för generering av 1-puls

START       CLR         $FB          Noll till utport (bit 0)
;
            LDA         #25          Räknarinitiering för loop1
SCOUNT      DECA                     Minska loopräknare
            BNE         SCOUNT       Färdigt? Nej
;                                    Ja
            LDA         #%00000001   Bit noll skall ettställas
            STA         $FB          Ett till utport (bit 0)
;
            LDA         #100         Räknarinitiering för loop2
PCOUNT      DECA                     Minska loopräknare
            BNE         PCOUNT       Färdigt? Nej
;                                    Ja
            CLR         $FB          Noll till utport (bit 0)
```


Reset
NF
CP
CTR4
3CT=0
M1
M2
C3
2+
1
1
0
0
1,3 D [1]
1,3 D [2]
1,3 D [4]
1,3 D [8]
q0
q1
q2
q3
BIN/HEX
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9
Q10
Q11
Q12
Q13
Q14
Q15
RESET
FETCH
EXECUTE
i0
i1
i2
i3
i4
i5
i6
i7
BIN/2^8
I00
I01
I02
I03
I04
IXX
IFD
IFE
IFF
≥1
f0
f1
f2
f3
LDR
OER
LDMA
MR
LDPC
OEPC
IncPC
LDI
LDA
NF
1
NF
Qi
IXX
&



FLIS-datorn i detalj

FLIS-processorns ALU



funktion				*operation*	*resultat (U)*	*flaggor*			
f_3	f_2	f_1	f_0	*RTN*		N	Z	V	C
0	0	0	0	U = 0	00_{16}	0	1	0	0
0	0	0	1	U = FD_{16}	FD_{16}	1	0	0	0
0	0	1	0	U = FE_{16}	FE_{16}	1	0	0	0
0	0	1	1	U = FF_{16}	FF_{16}	1	0	0	0
0	1	0	0	U = E	E	u_7	(1)	0	0
0	1	0	1	U = D_{1k} + C_{in}	–D – 1 + C_{in}	u_7	(1)	(9)	(8)
0	1	1	0	U = D∨E	bitvis D *OR* E	u_7	(1)	0	0
0	1	1	1	U = D∧E	bitvis D *AND* E	u_7	(1)	0	0
1	0	0	0	U = D⊕E	bitvis D *XOR* E	u_7	(1)	0	0
1	0	0	1	U = D + C_{in}	D + C_{in}	u_7	(1)	(2)	(3)
1	0	1	0	U = D + FF_{16}	D – 1	u_7	(1)	(2)	(3)
1	0	1	1	U = D + E + C_{in}	D + E + C_{in}	u_7	(1)	(2)	(3)
1	1	0	0	U = D + E_{1k} + C_{in}	D – E – 1 + C_{in}	u_7	(1)	(2)	(10)
1	1	0	1	U = D(C_{in})<< 1	2D + C_{in}	d_6	(1)	(6)	(4)
1	1	1	0	U = (C_{in}) D >> 1	$2^7 \cdot C_{in}$ + D/2	C_{in}	(1)	(7)	(5)
1	1	1	1	U = (d_7) D >> 1	D/2 (med tecken)	d_7	(1)	0	(5)

Anm:

(1) Z = 1 då samtliga bitar i värdet U är 0. Z = 0 annars.

(2) V = 1 vid overflow enligt reglerna för 2k-aritmetik. V = 0 annars.

(3) C = 1 om summan är större än 255. C = 0 annars.

(4) C = utskiftad bit, dvs. bit d_7.

(5) C = utskiftad bit, dvs. bit d_0.

(6) V = $d_7 \oplus d_6$ dvs. sätts till 1 om skiftet ger teckenbyte.

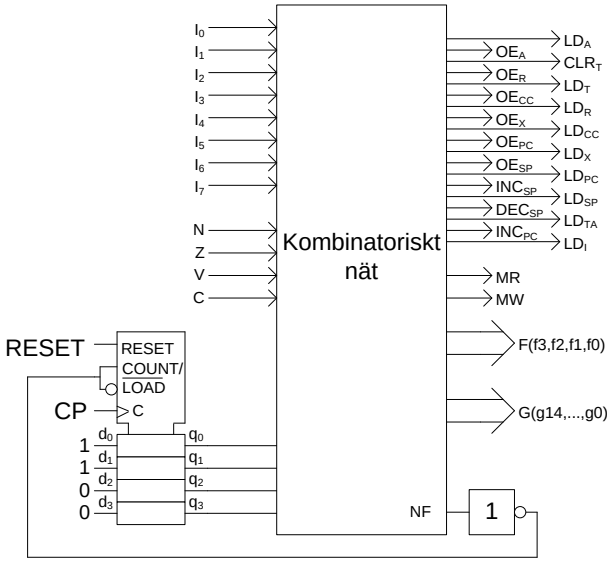
(7) V = $C_{in} \oplus d_7$ dvs. sätts till 1 om skiftet ger teckenbyte.

(8) C = 0 om D = 00000000_2. C = 1 annars.

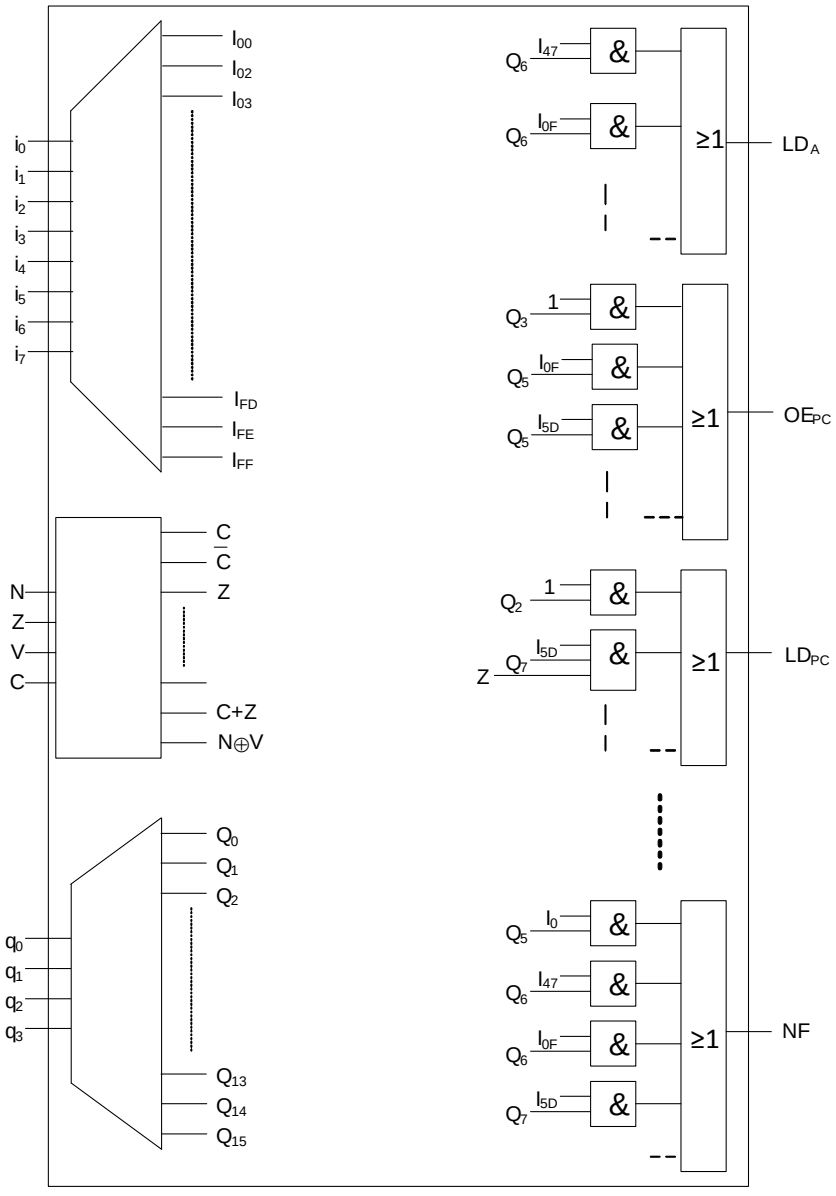
(9) V = 1 om D=10000000_2. V = 0 annars.

(10) C = 1 om summan är mindre än 256. C = 0 annars.

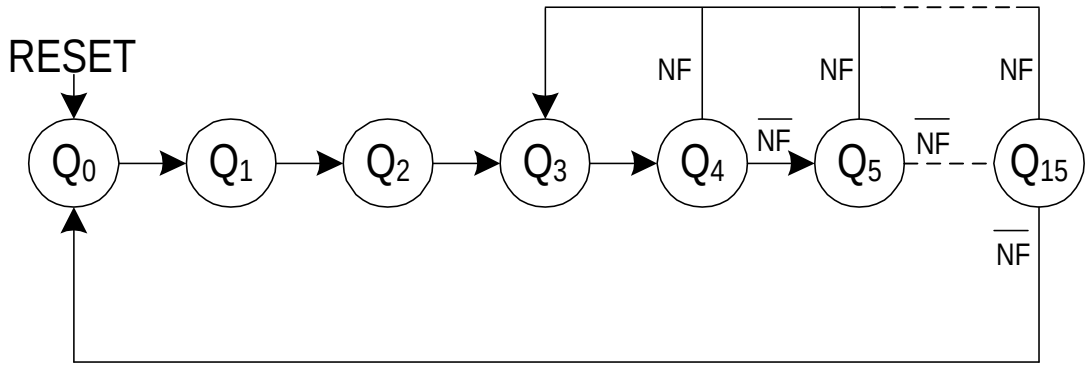
FLIS-processorns styrenhet



Det kombinatoriska nätet



Styrenhetens tillståndsgraf



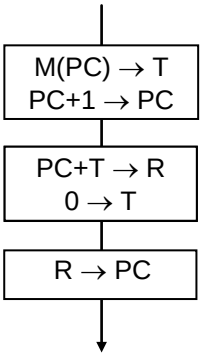
Implementering av ovillkorlig branch-instruktion i den fasta styrenheten för FLISP

BRA Adr Operationsbeskrivning: Flödesplan:

PC+Offs → PC

Instruktion: BRA Adr
Format: Ord1: 21_{16}(Opkod) Ord2: Offset

Tillstånd	Summaterm	RTN-beskrivning	Styrsignaler
Q_4	$Q_4 \cdot I_{21}$	M(PC)→T, PC+1→PC	MR, LD_T, INC_{PC}
Q_5	$Q_5 \cdot I_{21}$	PC+T→R, 0→T	OE_{PC}, f_3, f_1, f_0, LD_R, CLR_T
Q_6	$Q_6 \cdot I_{21}$	R→PC, (New Fetch)	OE_R, LD_{PC}, NF



Implementering av villkorliga branch-instruktioner med fast styrenhet för FLISP

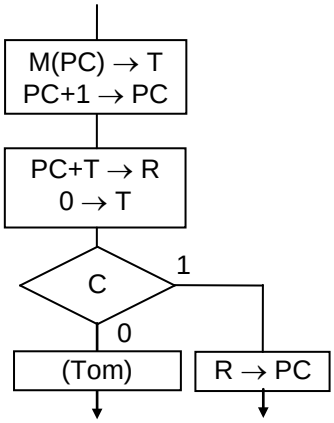
Metodiken för implementering av villkorliga instruktioner belyses av följande exempel:

Ex) BCS Adr Operationsbeskrivning: Flödesplan:

If C=1: PC+Offs → PC

Instruktion: BCS Adr
Format: Ord1: 28_{16}(Opkod) Ord2: Offset

State	Summaterm	RTN-beskrivning	Styrsignaler
Q_4	$Q_4 \cdot I_{28}$	M(PC)→T, PC+1→PC	MR, LD_T, INC_{PC}
Q_5	$Q_5 \cdot I_{28}$	PC+T→R, 0→T	OE_{PC}, f_3, f_1, f_0, LD_R, CLR_T
Q_6	$Q_6 \cdot I_{28} \cdot C$	If C=1: R→PC	OE_R, LD_{PC}
	$Q_6 \cdot I_{28}$	New Fetch	NF



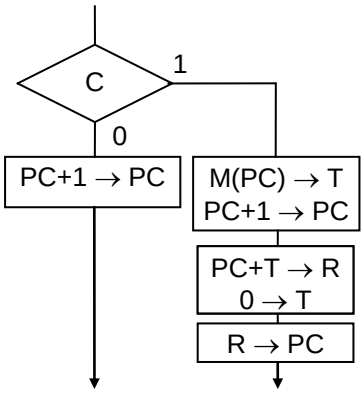
Man kan modifiera den villkorliga branch-instruktionen BCS så att den exekveras enligt flödesplanen nedan till höger. Beskriv EXECUTE-fasen i RTN-tabellen nedan. Använd opkod 03_{16}.

BCS Adr Operationsbeskrivning: Flödesplan:

If C=1: PC+Offs → PC

Instruktion: BCS Adr
Format: Ord1: 03_{16} (Opkod) Ord2: xx_{16} (offset)

State	Summaterm	RTN-beskrivning	Styrsignaler
Q_4			
Q_5			
Q_6			



Assemblerspråket för FLIS-processorn.

Assemblerspråket använder sig av mnemoniska beteckningar liknande dem som processorkonstruktören MOTOROLA (FREESCALE) specificerat för maskininstruktioner för mikroprocessorerna 68XX och instruktioner till assemblatorn, s k pseudoinstruktioner eller assemblatordirektiv. Pseudoinstruktionerna listas i tabell 1.

Tabell 1

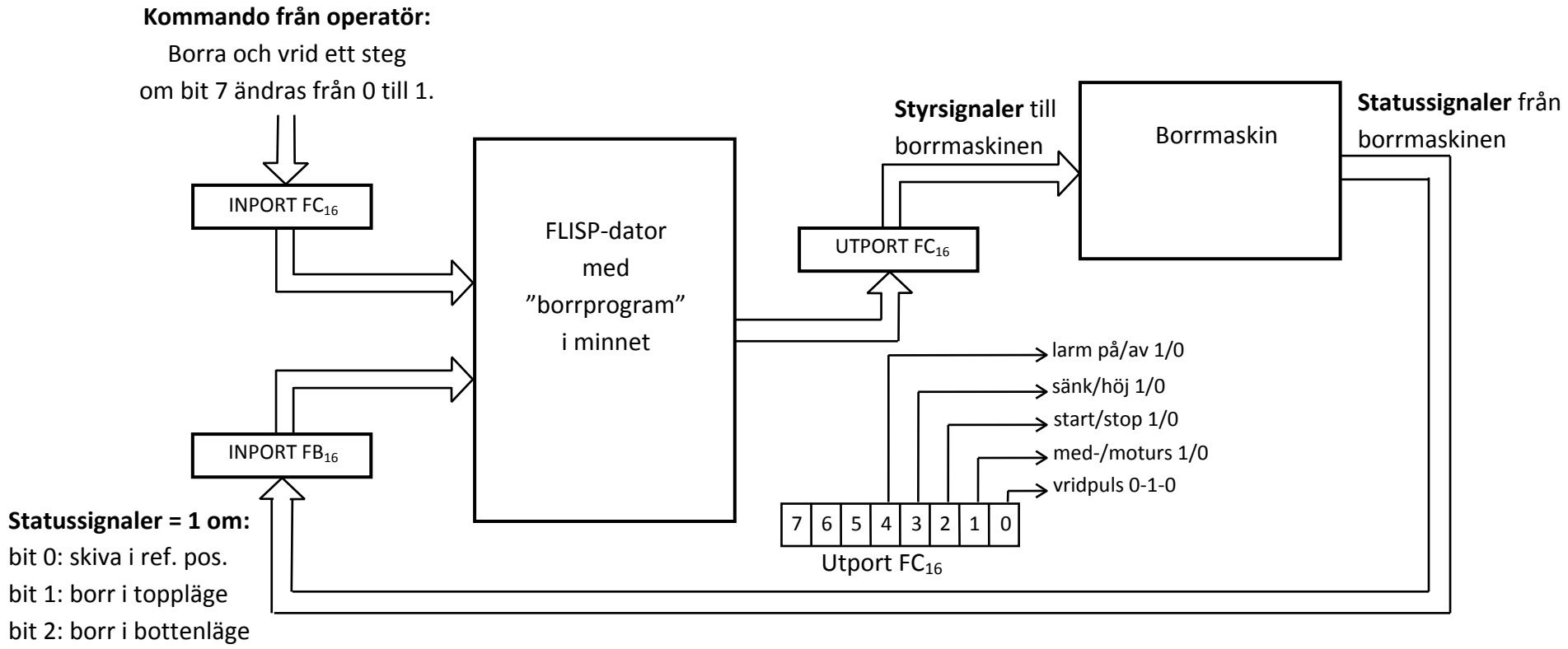
Direktiv			Förklaring
	ORG	N	Placerar den efterföljande koden med början på adress N. (ORG för ORiGin = ursprung)
L	RMB	N	Avsätter N bytes i följd i minnet (utan att ge dem värden), så att programmet kan använda dem. Följden placeras med början på adressen L. (RMB för Reseve Memory Bytes)
L	EQU	N	Ger symbolen L konstantvärdet N. (EQU för EQUates)
L	FCB	N1, N2	Avsätter en byte för varje argument i följd i minnet. Respektive byte ges konstantvärdet N1, N2 etc. Följden placeras med början på adressen L. (FCB för Form Constant Byte)
L	FCS	”ABC”	Avsätter en byte för varje tecken i teckensträngen ”ABC” i följd i minnet. Respektive byte ges ASCII-värdet för A B C, etc. Följden placeras med början på adressen L. (FCS för Form Character String)

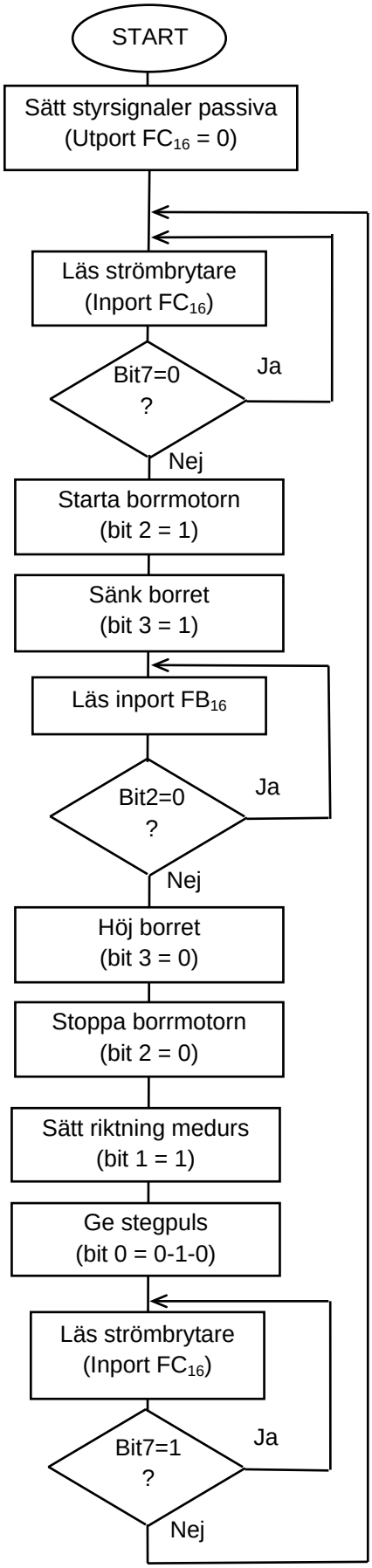
```
; Demoprogram för yttre enheter som kan anslutas till FLIS-processorn
;
; I/O-enheter
;
;*******************************************************************************
; Det finns två inportar och två utportar i FLISP-datorn på adresserna $FB OCH $FC
;
; På adressen $FB finns både en inport och en utport. Adressen är det enda som dessa
; har gemensamt, så data som skrivs på t ex utport $FB har inget att göra med data som
; läses på inport $FB.
;
; Det finns ett antal inenheter och utenheter som man kan ansluta till portarna.
;
; Inenheter som kan anslutas till inportar:
; "DIPSWITCH" (FB,FC), med åtta strömbrytare (switchar), som kan ställas in på 0/1.
; "Keyboard" (FB,FC), där koden för en nedtryckt tangent kan avläsas.
; "Drill" (FB), där tre givare från en borrmaskin kan avläsas.
; "Console" (FC), där man kan läsa ASCII-tecken från din dators tangentbord.
;
; Utenheter som kan anslutas till utportar:
; "LED" (FB,FC), med åtta lysdioder (lampor), som kan tändas (1) eller släckas (0).
; "HEXDISPLAY" (FB,FC), där man kan visa ett 8-bitars dataord som två hexsiffror.
; "7-SEGMENT" (FB,FC), där man kan styra varje segment i ett 7-segments sifferfönster.
; "Console" (FB), där man kan skriva ut tecken på en bildskärm.
; "Drill" (FC), där man kan styra olika funktioner hos en borrmaskin.
;*******************************************************************************
;
; Assemblera programmet ("Debug|Assemble") och ladda det i minnet ("Load New").
;
; Anslut inenheten "DIPSWITCH" till inport $FB och utenheten "LED" till utport $FB.
;
; Kör sedan programmet,först instruktion för instruktion med "Step",
; sedan med "Run Slow" och till sist med "Run Fast".
; Ändra på strömbrytarna på "DIPSWITCH" medan du kör och iakttag utenheten.
;
; Stoppa programmet och byt i tur och ordning utenhet till "HEXDISPLAY", "7-SEGMENT"
; och "CONSOLE" och kör programmet på samma sätt med dessa.
;
; Stoppa programmet och byt både inport och utport till adressen $FC, assemblera
; och ladda det.
; Anslut sedan "DIPSWITCH" till inport $FC och "DRILL" till utport $FC och kör
; programmet.
;
;*******************************************************************************
        ORG     0       Programmets startadress
START   LDA     $FB     Läs av de åtta databitarna från ansluten inenhet
        STA     $FB     Mata ut de åtta databitarna till ansluten utenhet
        JMP     START   Upprepa förloppet
STOP                    ;Här är första adressen efter programmet
```

Enkel borrmaskinsstyrning (uppgift 16.16 från arbetsboken, modifierad)

FLIS-datorn skall användas för att styra en komplett borrning av ett hål med borrmaskinen. En flödesplan för borrningen ges på nästa sida. Borrningen skall starta när man med hjälp av en switch ändrar värdet på bit 7 på inport FC_{16} från "0" till "1". När ett hål är färdigborrat skall arbetsstycket vridas ett steg medurs.

Skriv ett program enligt flödesplanen och översätt till maskinspråk. Ange även vilka adresser som de olika instruktionerna placeras på. Startadressen för programmet skall vara $\mathbf{00_{16}}$.



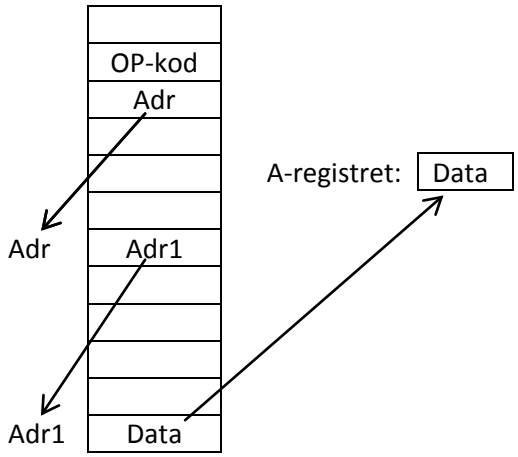

START
Sätt styrsignaler passiva
(Utport FC16 = 0)
Läs strömbrytare
(Inport FC16)
Bit7=0
?
Ja
Nej
Starta borrmotorn
(bit 2 = 1)
Sänk borret
(bit 3 = 1)
Läs inport FB16
Bit2=0
?
Ja
Nej
Höj borret
(bit 3 = 0)
Stoppa borrmotorn
(bit 2 = 0)
Sätt riktning medurs
(bit 1 = 1)
Ge stegpuls
(bit 0 = 0-1-0)
Läs strömbrytare
(Inport FC16)
Bit7=1
?
Ja
Nej

Programförslag:

```
;       Program för borrning av ett hål. Borrning startas när en positiv
;       flank har detekterats på bit 7 på inport $FC.
;       Borrmaskinens styrsignaler är anslutna till utport $FC och
;       statussignaler till inport $FB.

;       Ingen hänsyn har tagits till att borrmaskinen är ett mekaniskt
;       system som reagerar långsamt på givna kommandon. Efter varje
;       kommando borde därför en tillräckligt lång fördröjning läggas in.

;       Efter borrningen vrids arbetsstycket ett steg medurs.
;*************************************************************************
;
;       Anslut "DIP-SWITCH" till inport $FC och borrmaskinen till utport $FC
;       och inport $FB!
;
;*************************************************************************
        ORG     0
        CLR     $FC             Passiva styrsignaler till borrmaskinen

LOOP    TST     $FC             Läs strömbrytare
        BPL     LOOP            Kolla bit7. Vänta om bit7= 0

        LDA     #%00000100      Bit2=1 => Starta motor
        STA     $FC             Starta borrmotorn!

        ORA     #%00001000      Bit3=1 => Sänk borr
        STA     $FC             Sänk borret!

WAIT1   LDA     $FB             Läs borrmaskinens givare
        ANDA    #%00000100      Maska fram "borr i bottenläge"
        BEQ     WAIT1           Vänta tills borr i bottenläge

        LDA     #%00000100      Höj borr med motor på
        STA     $FC             Höj borret!

        ANDA    #%11111011      Bit2=0 => Stäng av motor
        STA     $FC             Stoppa borrmotorn!

;                               Vrid ett steg
        ORA     #%00000010      Bit2=1 => Medurs vridning vid puls på bit0
        STA     $FC             Mata ut vridriktning till borrmaskinen!

        ORA     #%00000001      Bit0=1 => positiv flank på bit0
        STA     $FC             Mata ut puls på bit0 till borrmaskinen 0-1-0
        ANDA    #%11111110
        STA     $FC

WAIT2   TST     $FC             Läs operatörens strömbrytare
        BMI     WAIT2           Vänta tills bit7=0 (borrkommando borta)

        BRA     LOOP
```

Indirekt adressering **(Finns ej i FLISP)**

LDA [Adr] M(M(Adr)) = M (Adr1) ⟶ A Instruktionens adressdel är adressen till adressen till data.



Demoprogram som visar stacken vid anrop av en subrutin som anropar en ny subrutin.

```
Adr  Data
                        ORG     $10     Huvudprogram
10   92 F0      START   LDSP    #$F0    Sätt stackpekare till BOS
12   00                 NOP
13   00                 NOP
14   00                 NOP
15   00                 NOP
16   00                 NOP
17   90 50              LDX     #$50    X och A får värden i huvudprogrammet
19   05                 CLRA
1A   34 40              JSR     $40     Anrop av subrutin 1
1C   E1 E0              STA     $E0
1E   21 FE      READY   BRA     READY


                        ORG     $40
40   10         SUBR1   PSHA            Här sparas A-värdet från huvudprogrammet på stacken
41   11                 PSHX            Här sparas X-värdet från huvudprogrammet på stacken
42   13                 PSHC            Här sparas flaggorna från huvudprogrammet på stacken
43   90 E0              LDX     #$E0    Här ändras X-värdet i subrutin 1
45   00                 NOP
46   00                 NOP
47   00                 NOP
48   00                 NOP
49   00                 NOP
4A   00                 NOP
4B   00                 NOP
4C   00                 NOP
4D   F0 03              LDA     #$03    Här ändras A-värdet i subrutin 1
4F   34 60              JSR     $60     Anrop av subrutin 2

51   17                 PULC            Här hämtas huvudprogrammets flaggvärden från stacken
52   15                 PULX            Här hämtas huvudprogrammets X-värde från stacken
53   14                 PULA            Här hämtas huvudprogrammets A-värde från stacken
54   43                 RTS             Slut på subrutin 1. Återhopp till huvudprogram
                ;       Vid återhoppet är registervärdena från huvudprogrammet oförändrade


                        ORG     $60
60   10         SUBR2   PSHA            Här sparas A-värdet från subrutin 1 på stacken
61   13                 PSHC            Här sparas flaggorna från subrutin 1 på stacken
62   11                 PSHX            Här sparas X-värdet från subrutin 1 på stacken
63   90 15              LDX     #$15    Här ändras X-värdet i subrutin 2

65   07                 INCA            Här ändras A-värdet i subrutin 2

66   15                 PULX            Här hämtas X-värdet från subrutin 1 från stacken
67   17                 PULC            Här hämtas flaggvärden från subrutin 1 från stacken
68   14                 PULA            Här hämtas A-värdet från subrutin 1 från stacken
69   43                 RTS             Slut på subrutin 2. Återhopp till subrutin 1
                ;       Vid återhoppet är registervärdena från subrutin 1 oförändrade
```

Heltalsvariabler och styrstrukturer i C

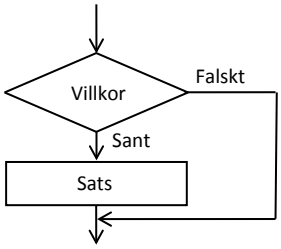
Heltalsvariabler i C.

Heltalsvariabler med tecken deklareras som: character, short integer, integer eller long.
För heltalsvariabler utan tecken föregås variabeldeklarationen av unsigned, t ex unsigned integer.

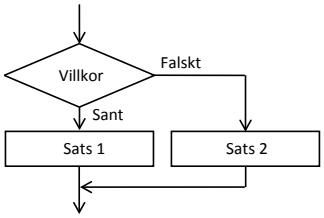
Om variablerna har tecken eller saknar tecken bestämmer vilken uppsättning av villkorliga hopp, "signed" eller "unsigned", som används vid "Test" eller bestämning av "Villkor" nedan.

Styrstrukturer i C

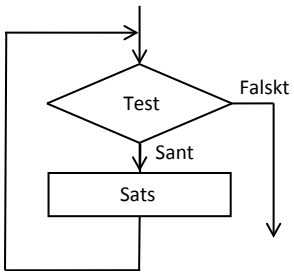
Ex.
```
if ( i > 0 )
  sats
```



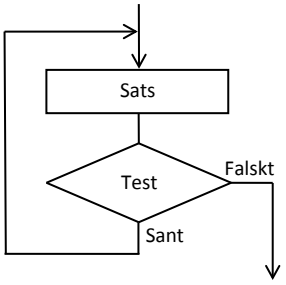
Ex.
```
if ( i > 0 )
  sats1
else
  sats2
```



Ex.
```
while ( i > 0 )
  sats
```

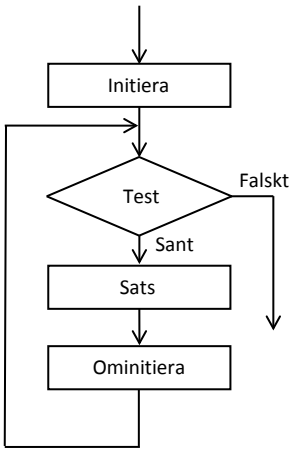


Ex.
```
do
   sats
  while ( i > 0 );
```



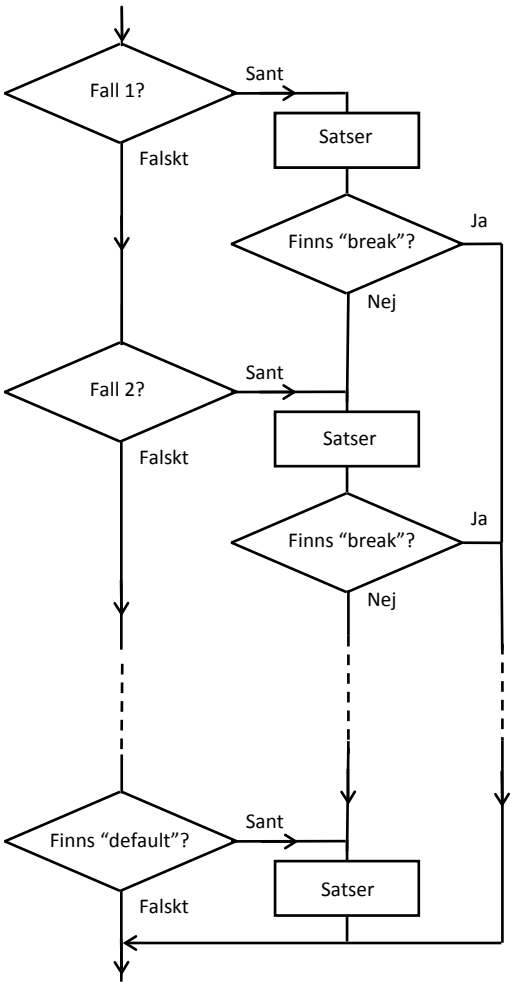
Ex.

```
for ( i = 0; i < n; i++)
  sats
```



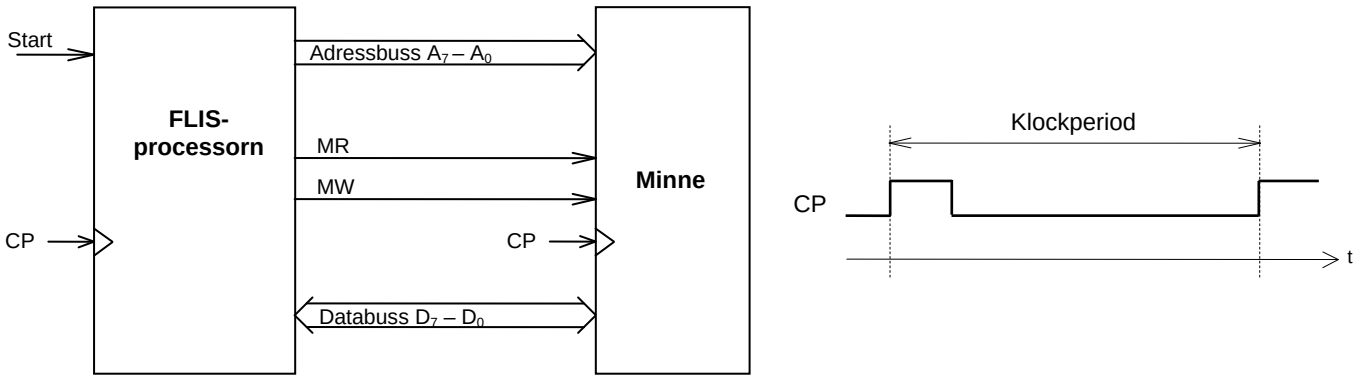
Ex.

```
switch ( i ) {
   case   Fall1:
      satser
      break;
   case   Fall2:
      satser
      break;
   .
   .
   .
   default:
      satser
      break;
}
```



Hur FLIS-processorn läser och skriver i minnet

I figur 1 visas de signaler FLIS-processorn använder för läsning och skrivning i minnet.



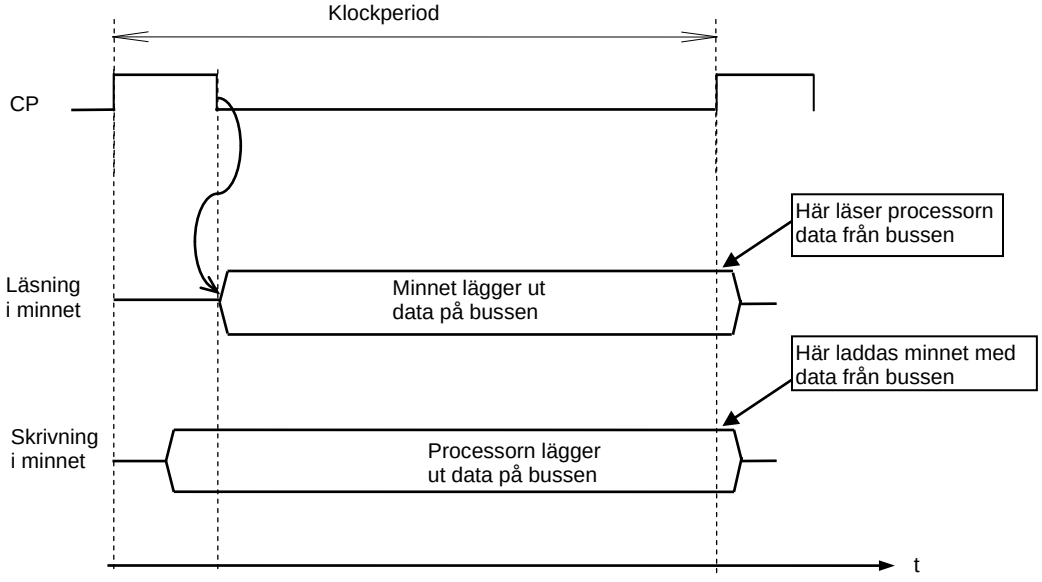
Figur 1 FLIS-processorn kopplad till minnet.

FLISP-processorn läser eller skriver i minnet under en klockperiod. Den inleds när CP går från låg till hög nivå (positiv flank) och avslutas med nästa positiva flank på CP-signalen, såsom visas i figur 1.

En läsning i minnet (en läscykel) inleds med att processorn lägger ut adressen, där läsningen skall ske, på adressbussen samtidigt som den lägger ut hög nivå på signalen MR.

På grund av att det finns interna fördröjningar i processorn så kommer varken adressbitar eller MR-signalen att ha rätt värde förrän ***efter*** CP-signalens positiva flank. Dessutom kommer de inte att få rätt värde samtidigt på grund av att fördröjningarna är olika för de olika signalerna. Vid tidpunkten när CP-signalen går låg igen är dock adressen och MR-signalen garanterat korrekta (stabila) och först då skall minnet börja "plocka" fram data från den aktuella adressen och placera det på databussen, vilket visas i figur 2. Processorn läser sedan av data från databussen vid slutet av klockcykeln dvs. vid CP-signalens nästa positiva flank.

Ett liknande resonemang kan föras för adresser och MW-signalen vid skrivning. Skrivningen inleds med att processorn lägger ut adressen, där skrivningen skall ske, på adressbussen samtidigt som signalen MW ges värdet "1". Därefter lägger processorn ut data på databussen på det sätt som visas i figur 2. Minnet laddas sedan med data från databussen vid slutet av klockcykeln dvs vid CP-signalens nästa positiva flank.



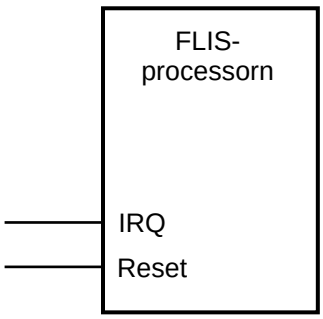
Figur 2 Läsning och skrivning i minnet med FLIS-processorn

Vid läsning och skrivning är således adressen samt MR- resp. MW-signalen, som läggs ut från processorn vid CP-signalens positiva flank, inte garanterat korrekta (stabila) förrän vid CP-signalens negativa flank. Adressen och MR- eller MW-signalen har alltså stabila värden under tiden som CP är "0".

Yttre styrning (Ver 2013-05-07)

Yttre signaler som påverkar exekveringen hos FLIS-processorn

Nedan beskrivs två yttre styrsignaler som styr exekveringen hos FLIS-processorn.



1. Reset
När en etta utifrån läggs på Reset-ingången upphör processorns arbete.

När sedan Reset-ingången nollställs händer följande:

I-biten (I-flaggan) i CC-registret ettställs och övriga flaggor nollställs. Att I-biten ettställs innebär att FLIS-processorns avbrottssystem är avstängt. (Se punkt 2 nedan!)

Processorn läser sedan innehållet i den s k "resetvektorn" på minnesadressen FF_{16} och placerar det lästa 8-bitars dataordet i programräknaren (PC). Sedan hämtar processorn nästa instruktion från denna adress.

Genom att aktivera "Reset"-signalen kan man alltså få processorn att starta om program-exekveringen från den adress som finns lagrad i "resetvektorn" på adressen FF_{16} i minnet. Reset-signalen aktiveras alltid när matningsspänningen till processorn slås på från början.

2. IRQ (Interrupt request, Avbrottsbegäran)
En yttre enhet kan få processorn att (oftast tillfälligt) byta program genom att lägga en etta på insignalen IRQ. Processorn exekverar då färdigt den instruktion den håller på med. Om avbrottssystemet är aktiverat, dvs. om I-biten i flaggregistret (CC-registret) är nollställd, kommer processorn sedan att acceptera avbrottet. I annat fall kommer programexekveringen att fortsätta som vanligt.

Om avbrottsbegäran accepteras sparar processorn innehållen i de interna registren i ordningen PC, Y, X, A och CC på stacken. (Innehållet i PC är här adressen till nästa instruktion i det avbrutna programmet.) Därefter ettställs I-flaggan i CC-registret för att nya avbrott skall utestängas.

Processorn läser sedan innehållet i den sk "IRQ-vektorn" på minnesadressen FD_{16}. Detta innehåll, som är adressen till en avbrottsrutin, placeras i programräknaren. Därmed har processorn utfört ett hopp från det vanliga programmet till avbrottsrutinen.

Yttre styrning (Ver 2013-05-07)

När avbrottsrutinen börjar exekveras vet man att den händelse som orsakar avbrott har inträffat. Avbrottsrutinen skall därför utföra arbetet som är förknippat med avbrottshändelsen och sedan återvända till det vanliga (avbrutna) programmet med återställda registerinnehåll. Det finns en speciell instruktion, RTI (Return from interrupt), som återställer innehållen i samtliga processorregister genom att läsa de gamla registerinnehållen från stacken. Eftersom RTI även återställer innehållet i PC kommer exekveringen efter RTI att fortsätta i det program som tidigare blev avbrutet. En avbrottsrutin avslutas därför med instruktionen RTI.

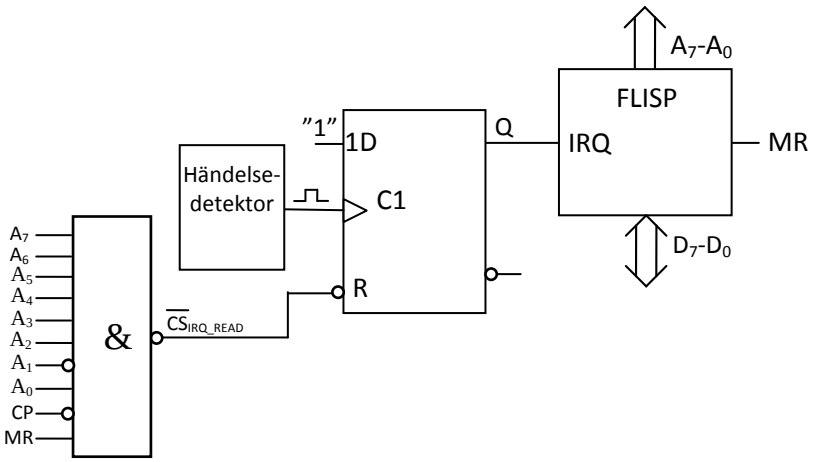
Vid återhoppet från avbrottsrutinen är det viktigt att den aktiva signal som orsakade det pågående avbrottet har försvunnit, annars kommer ju ett nytt avbrott direkt att genereras av den "gamla" signalen efter återhoppet.

Lägg märke till att avbrottssystemet automatiskt aktiveras då de gamla registerinnehållen i processorn återställs vid återhopp från avbrottsrutinen eftersom det gamla CC-innehållet, som hämtas från stacken, innehåller en nolla i I-biten.

Nedan visas hur några olika vektorer och in-/utportar är placerade i processorns adressrum.

FB_{16}: I/O-portar (Inport och utport)
FC_{16}: I/O-portar (Inport och utport)
FD_{16}: IRQ-vektor (För adress till avbrottsrutin.)
FE_{16}: TRAP-vektor (Adress för hantering av otillåten operationskod.)
FF_{16}: RESET-vektor (För startadress)

Standardkoppling för avbrottsgenerering till FLIS-processorn:



Exempel på avbrott med FLISP

Ett datorsystem med FLIS-processorn skall användas för att köra ett huvudprogram.

Samtidigt som huvudprogrammet körs skall en variabel KNAPP (8 bitar) på minnesadressen 60_{16} ökas med ett varje gång en tryckknapp aktiveras.

Ökningen av variabeln KNAPP skall ske med hjälp av IRQ-avbrott genom att huvudprogrammet avbryts vid varje knapptryckning och en avbrottsrutin KNPINC då körs. Se figuren på nästa sida.

Avbrottsrutinen KNPINC skall ha startadressen 30_{16} i minnet.

Avbrottsvektorn för IRQ-avbrott finns i RWM (läs- och skrivbart minne) på adressen FD_{16}.

Avbrottsrutinen på adressen KNPINC (30_{16}) blir:

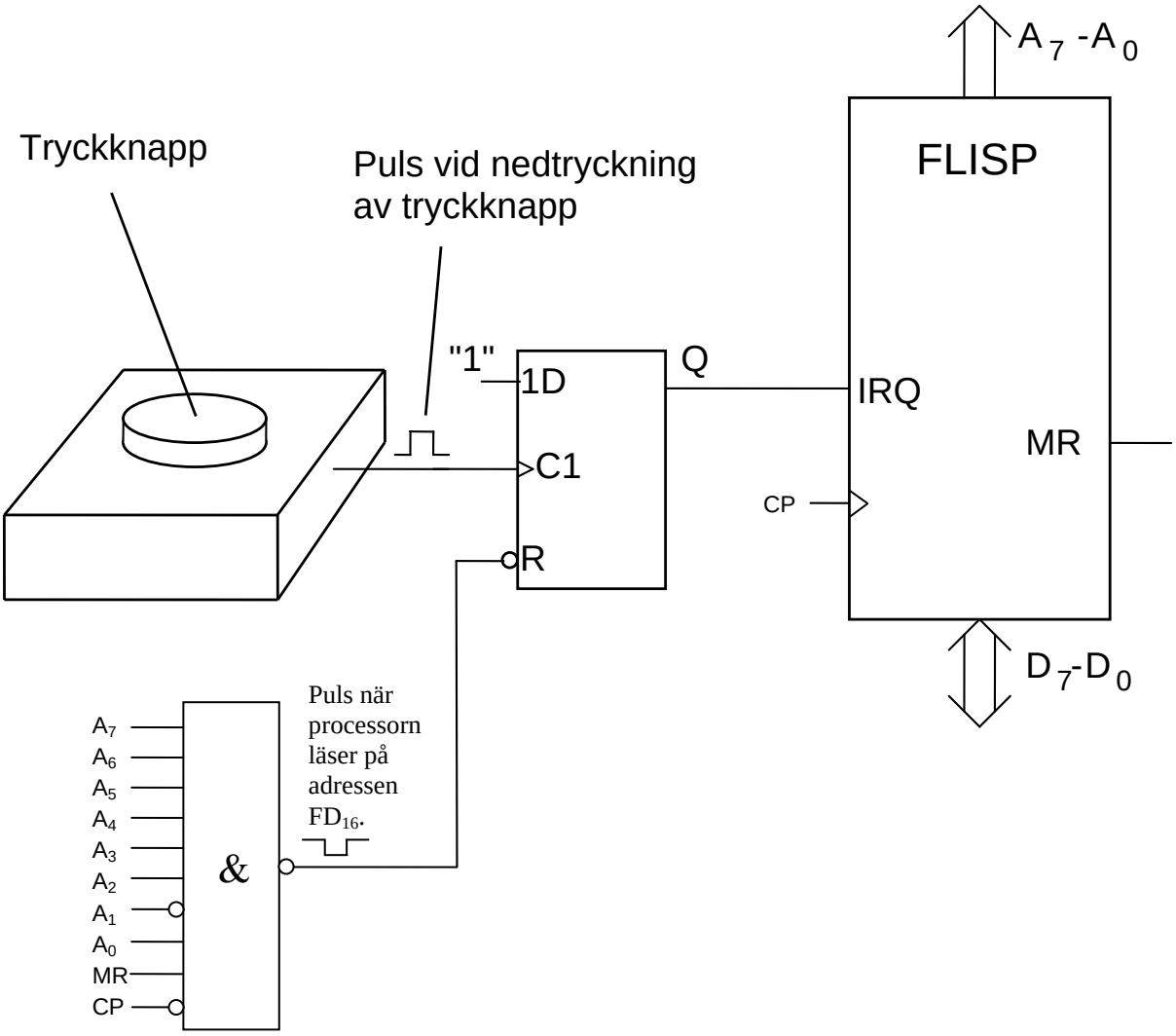
```
KNPINC  INC     $60     Variabel KNAPP ökas med ett.
        TST     $FD     Nollställ avbrottsvippan.
        RTI             Återvänd till huvudprogrammet.
```

Nedan visas nödvändiga initieringar i huvudprogrammet för att avbrott på IRQ-ingången skall accepteras. Dessutom visas hur IRQ-vektorn ges sitt rätta värde KNPINC (60_{16}).

Huvudprogram:

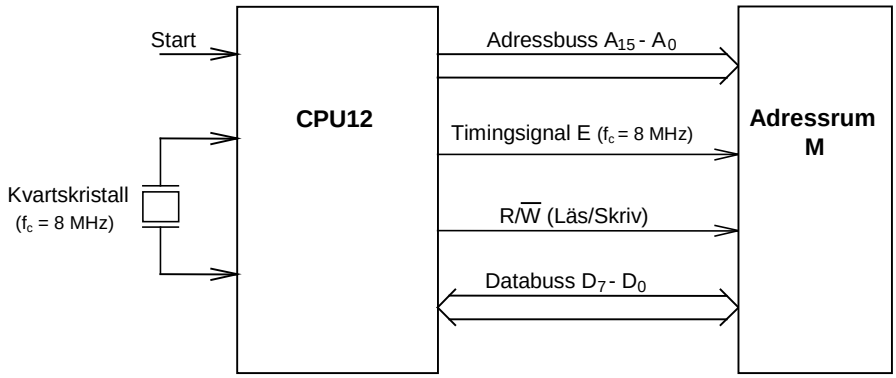
```
MAINPG  LDSP    #STACKADR   BOS definieras först.

        LDX     #KNPINC     Adr KNPINC = 30(16)
        STX     $FD         Ge IRQ-vektorn värdet KNPINC

        TST     $FD         Nolla avbrottsvippan

        CLR     $60         Nolla variabeln KNAPP (60(16))

        ANDCC   #%11101111  Aktivera avbrottssystemet
;         .                 (Nolla I-biten i CC-registret)
          .
```

Yttre styrning (Ver 2013-05-07)



Hur processorn CPU12 läser och skriver i minnet (från kap 13 i KMP)

I Figur 13.6 visas de signaler CPU12 använder för läsning och skrivning i minnet (adressrum M).

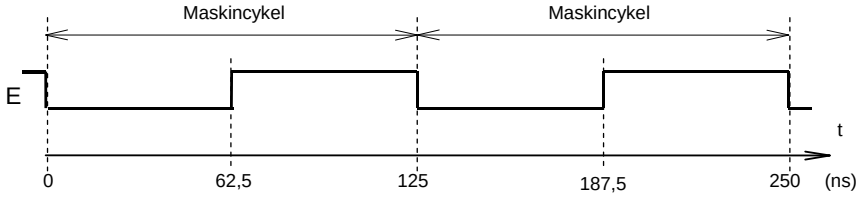


Figur 13.6 CPU12 kopplad till minnet (adressrum M).

Läsning och skrivning i minnet kallas minnesreferenser och sker under en E-klockcykel.

CPU12-klockning (timing)

CPU12 genererar den symmetriska "timing"-signalen E under hela tiden den arbetar. Den bildas ur signalen från en kristalloscillator med frekvensen 8 MHz i vårt fall. Se Figur 13.7.



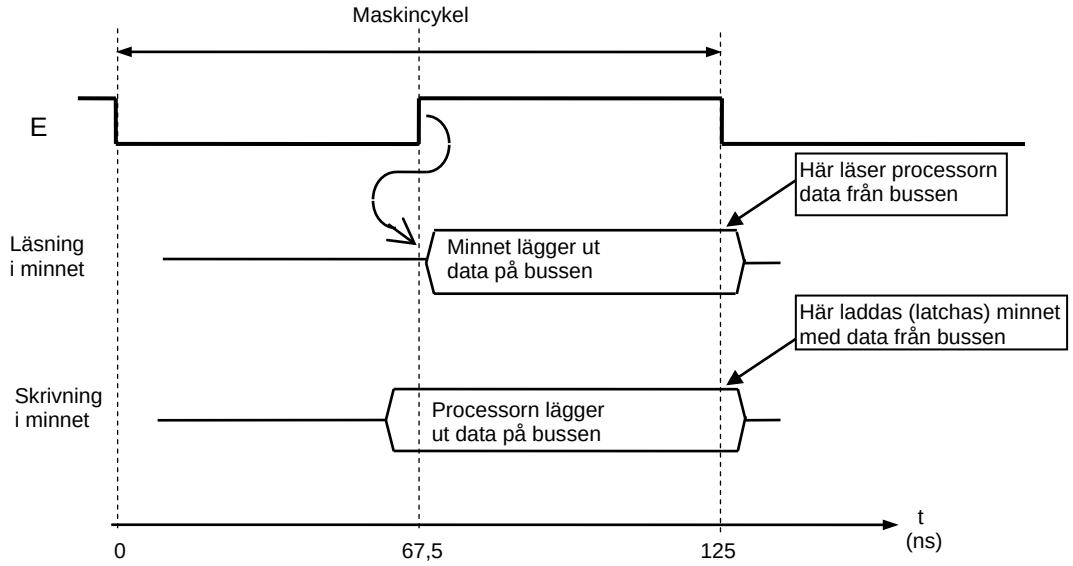
Figur 13.7 Timingsignalen E hos CPU12.

När CPU12 läser eller skriver i minnet utför den en ***maskincykel***. Den inleds när ***signalen* E** går från hög till låg nivå (negativ flank) och avslutas med nästa negativa flank på E-signalen, såsom visas i Figur 13.7.

En läsning i minnet (en läscykel) inleds med att processorn lägger ut adressen, där läsningen skall ske, på adressbussen samtidigt som signalen ***Läs*/*Skriv*** (***R*/*W***) ges värdet "1". Det sker vid negativ flank på E-signalen.

På grund av att det finns interna fördröjningar i processorn så kommer varken adressbitar eller R/$\overline{W}$-signalen att ha rätt värde förrän ***efter*** E-signalens negativa flank. Dessutom kommer de inte att få rätt värde samtidigt på grund av att fördröjningarna är olika för de olika signalerna. Vid tidpunkten för E-signalens positiva flank är dock adressen och R/$\overline{W}$-signalen garanterat korrekta (stabila) och först då bör minnet börja "plocka" fram data från den aktuella adressen och placera det på databussen, vilket visas i Figur 13.. Processorn läser sedan av data från databussen vid slutet av maskincykeln dvs vid E-signalens nästa negativa flank.

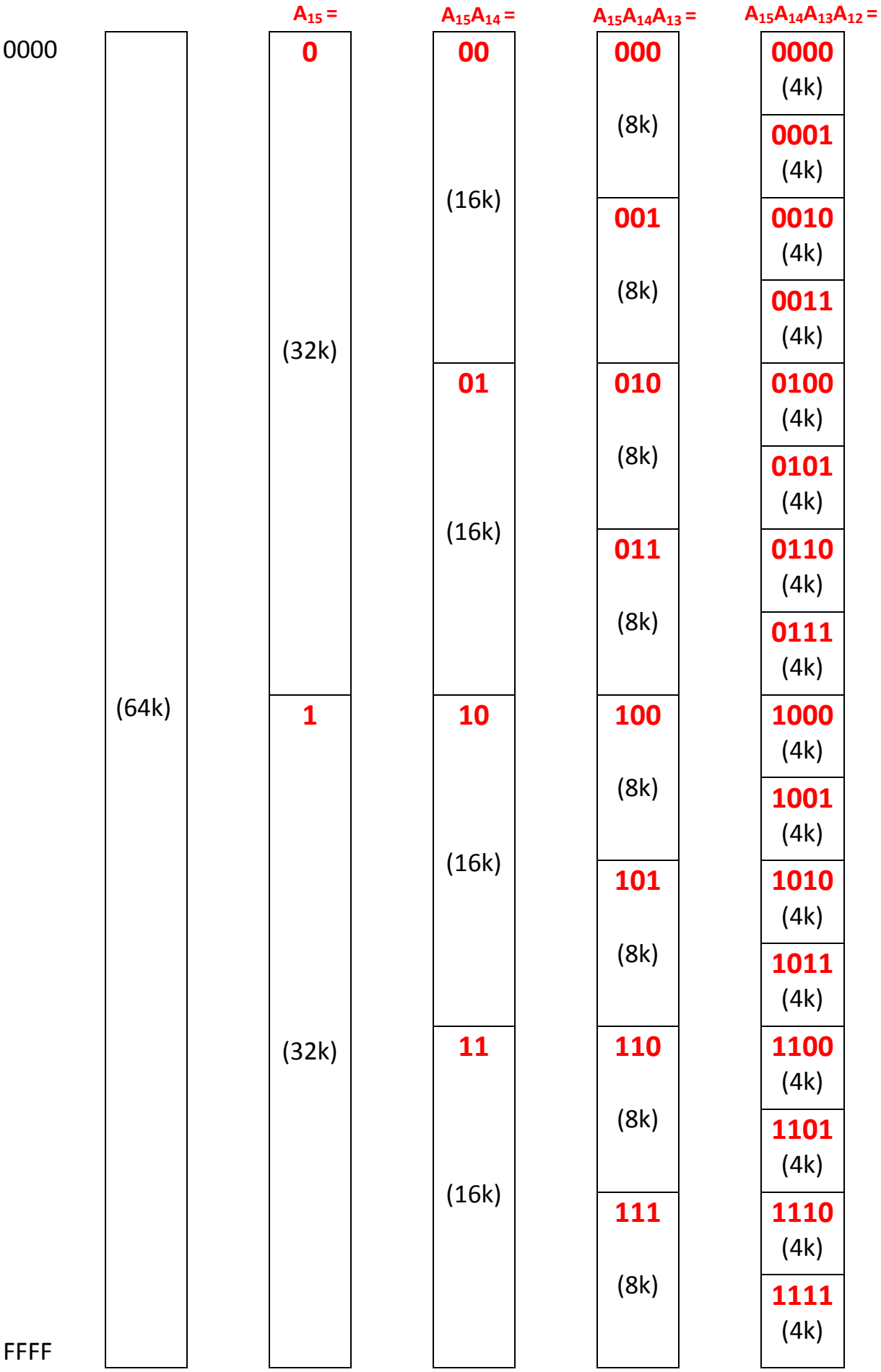
Ett liknande resonemang kan föras för adresser och $R/\overline{W}$ -signalen vid skrivning. Skrivningen inleds med att processorn lägger ut adressen, där skrivningen skall ske, på adressbussen samtidigt som signalen ***Läs*** / ***Skriv*** (***R*** /***W***) ges värdet "0". Därefter lägger processorn ut data på databussen på det sätt som visas i Figur 13.. Minnet laddas sedan med data från databussen vid slutet av maskincykeln dvs vid E-signalens nästa negativa flank.



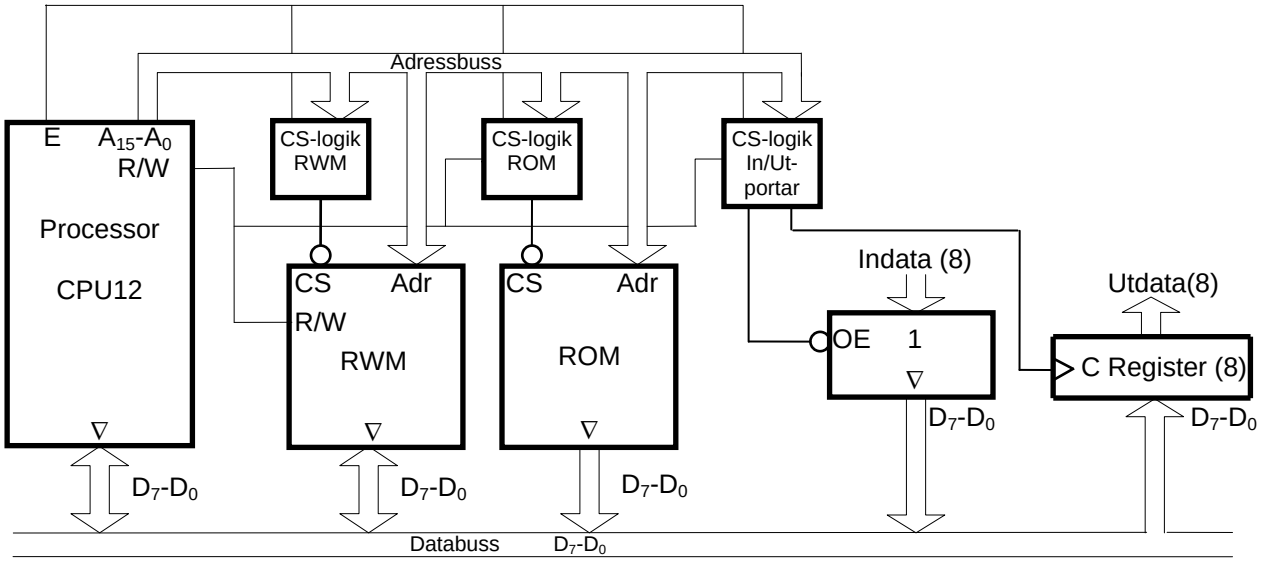
Figur 13.8 Läsning och skrivning med CPU12-processorn

Vid läsning och skrivning är således adressen och $R/\overline{W}$ -signalen, som läggs ut från processorn vid E-klockans negativa flank, inte garanterat korrekta (stabila) förrän vid E-klockans positiva flank. Under resterande halvan av maskincykeln är dock adressen stabil. Adressignalerna och $R/\overline{W}$ signalen har således stabila värden under tiden som E är "1". E- signalen är därför en **VMA-signal** (***Valid Memory Address***). För CPU12 används signalen VMA = E ofta som grindsignal för timing vid arbete mot olika minnesmoduler och in/ut-portar i adressrum M.

Adressrummet med 16 bitars adress (64k):

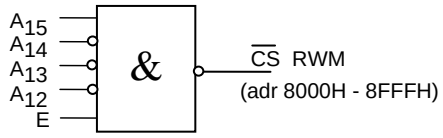


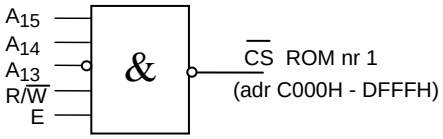
Principen för anslutning av minnesmoduler, inportar och utportar till processorn CPU12

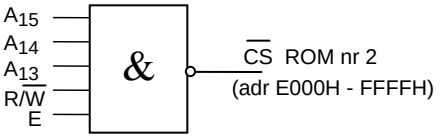


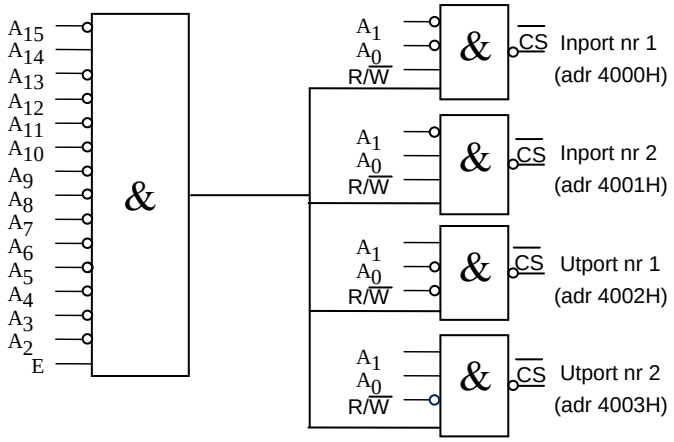
Fullständig adressavkodning

```
Inport 1:          4000H = 0100 0000 0000 0000
Inport 2:          4001H = 0100 0000 0000 0001
Utport 1:          4002H = 0100 0000 0000 0010
Utport 2:          4003H = 0100 0000 0000 0011

4k RWM     Start:  8000H = 1000 0000 0000 0000

           Slut:   8FFFH = 1000 1111 1111 1111

8k ROM #1  Start:  C000H = 1100 0000 0000 0000

           Slut:   DFFFH = 1101 1111 1111 1111

8k ROM #2  Start:  E000H = 1110 0000 0000 0000

           Slut:   FFFFH = 1111 1111 1111 1111
```









Ofullständig adressavkodning

```
Inport 1:          4000H = 0100 0000 0000 0000
Inport 2:          4001H = 0100 0000 0000 0001
Utport 1:          4002H = 0100 0000 0000 0010
Utport 2:          4003H = 0100 0000 0000 0011

4k RWM     Start:  8000H = 1000 0000 0000 0000

           Slut:   8FFFH = 1000 1111 1111 1111

8k ROM #1  Start:  C000H = 1100 0000 0000 0000

           Slut:   DFFFH = 1101 1111 1111 1111

8k ROM #2  Start:  E000H = 1110 0000 0000 0000

           Slut:   FFFFH = 1111 1111 1111 1111
```

Var hittar man modulerna i adressrummet i detta fall?

```
Inport 1: Alla adresser 4*n,   där n = 0-1FFFH
Inport 2: Alla adresser 4*n+1, där n = 0-1FFFH
Utport 1: Alla adresser 4*n+2, där n = 0-1FFFH
Utport 2: Alla adresser 4*n+3, där n = 0-1FFFH
```

```
4k RWM: 8000-8FFFH
        9000-9FFFH
        A000-AFFFH
        B000-BFFFH
```

Med denna adressavkodning ser vi att modulerna inte får några överlappande adressområden, men I/O-modulerna och RWM kommer att ”synas” i flera adressområden.

```
8k ROM #1: C000-DFFFH

8k ROM #2: E000-FFFFH
```

Konstruktion av en bitsekvensdetektor

X.11 Ett synkront sekvensnät skall ha en insignal x och en utsignal u.

Utsignalen u skall få värdet 1 för varje insignalsekvens som består av exakt en nolla följd av en etta. Med exakt en nolla menas att flera nollor följda av en etta inte skall ge utsignalen ett. Se exemplet nedan!

Exempel: σ_x = `010010110000101...` (Insignalsekvens)
σ_u = `0100001000000001...` (Utsignalsekvens)

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

a) Rita en tillståndsgraf för sekvensnätet.

b) Konstruera sekvensnätet.

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

Version 2013-05-07

Flyttal komplettering (32-bitars)

Format: s / c / f
Antal bitar: 1 8 23

s är teckenbit: 0 motsvarar + och 1 motsvarar –.

c är karakteristika: c = exp + 127.

f kallas fraction och är normaliserade mantissan utan inledande 1.

Högsta värde (255) och lägsta värde (0) på karakteristikan är reserverade för ∞ resp. 0.
± ∞: s / 255 / 0
± 0: s / 0 / 0 (Obs! i detta fall sätts inledande mantissabiten = 0.)

Talområde för exponenten: $-126 \leq exp \leq 127$

Största belopp blir: $1.111\ 1111\ 1111\ 1111\ 1111\ 1111 \cdot 2^{+127} \approx 2^{+128}$

Minsta belopp (förutom 0) blir: $1.000\ 0000\ 0000\ \ldots 0 \cdot 2^{-126} = 2^{-126}$

För att kunna representera tal med ännu mindre belopp än minsta beloppet ovan har man infört undantaget nedan, som dock ger sämre upplösning än normalfallet.

Undantag:

Man kan i uttrycket för ”0” ovan välja $f \neq 0$.

Största belopp skulle i detta fall bli:

$0.111\ldots1 \cdot 2^{-127} = (1 - 2^{-23}) \cdot 2^{-127}$

Det uppstår då ett ”glapp” till minsta beloppet (2^{-126}) i "normalfallet" ovan. För att undvika ”glappet” bestämmer man att exponenten skall sättas till –126 istället för –127 då c = 0.

Man får då största beloppet = $(1 - 2^{-23}) \cdot 2^{-126}$ i detta fall.

Heltalsmultiplikation för tal utan tecken: P = X · Y

Exempel

Y(4): Multiplikand $Y = 1011_2$

X(4): Multiplikator $X = 1101_2$ $\quad X = (x_3x_2x_1x_0)_2$

P(8): Produkt

Princip:

$$P = X \cdot Y = (x_3 \cdot 2^3 + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0) \cdot Y =$$

$$= x_3 \cdot 2^3 \cdot Y + x_2 \cdot 2^2 \cdot Y + x_1 \cdot 2^1 \cdot Y + x_0 \cdot 2^0 \cdot Y =$$

$$= x_3 \cdot Y \cdot 2^3 + x_2 \cdot Y \cdot 2^2 + x_1 \cdot Y \cdot 2^1 + x_0 \cdot Y \cdot 2^0$$

				1	0	1	1		Y
				1	1	0	1		X
	1	1	1	1	0	1	1		
			0	0	0	0			
		1	0	1	1				
+	1	0	1	1					
1	0	0	0	1	1	1	1		P

	Add $x_i \cdot Y$				Multiplikator			
	0	0	0	0	1	1	0	1
+ 1·Y	1	0	1	1				
→	1	0	1	1	1	1	0	1
	0	1	0	1	1	1	1	0
+ 0·Y	0	0	0	0				
→	0	1	0	1	1	1	1	0
	0	0	1	0	1	1	1	1
+ 1·Y	1	0	1	1				
→	1	1	0	1	1	1	1	1
	0	1	1	0	1	1	1	1
+ 1·Y	1	0	1	1				
→ 1	0	0	0	1	1	1	1	1
	1	0	0	0	1	1	1	1

Heltalsdivision för tal utan tecken: Y/X = Q + R/X

Exempel

Y(8): Dividend Y = 1001 1000_2 (= 152_{10})

X(4): Divisor X = 1010_2 (= 10_{10})

Q(8): Kvot

R(4): Rest

Princip:

$Y/X = Q + R/X \Leftrightarrow Y = Q \cdot X + R \Leftrightarrow R = Y - Q \cdot X$

$Q = (q_{n-1}q_{n-2}\ldots q_2q_1q_0)_2 = (q_7q_6q_5q_4q_3q_2q_1q_0)_2$

$$R = Y - Q \cdot X = Y - (q_{n-1} \cdot 2^{n-1} + q_{n-2} \cdot 2^{n-2} + \ldots + q_2 \cdot 2^2 + q_1 \cdot 2^1 + q_0 \cdot 2^0) \cdot X =$$
$$= Y - (q_7 \cdot 2^7 + q_6 \cdot 2^6 + q_5 \cdot 2^5 + q_4 \cdot 2^4 + q_3 \cdot 2^3 + q_2 \cdot 2^2 + q_1 \cdot 2^1 + q_0 \cdot 2^0) \cdot X$$

Divisionen går till så att man väljer q_i, för i = n-1, n-2, …, 2, 1, 0, så att R blir så litet som möjligt, men ≥ 0. Om R = 0 på slutet är Y jämnt delbart med X. I vårt exempel gäller att n = 8.

	q_7	q_6	q_5	q_4	q_3	q_2	q_1	q_0					
Q	0	0	0	0	1	1	1	1					
Y	1	0	0	1	1	0	0	0	1	0	1	0	X
-		1	0	1	0								
	0	1	0	0	1	0							
-			1	0	1	0							
		0	1	0	0	0	0						
-				1	0	1	0						
			0	0	1	1	0	0					
-					1	0	1	0					
				0	0	0	1	0					R

Heltalsmultiplikation genom upprepad addition P(16) = X(8)*Y(8)

Multiplikatorn X(8) finns från början i M(XVAR)
Multiplikanden Y(8) finns i M(YVAR)

Produkten P(16) finns efter multiplikationen i M(PHI):A (Hög del på adress PHI och låg del i register A)

Register A har från början värdet 0

```
LOOP    ADDA    YVAR    Addera multiplikanden till partialprodukt låg byte
        BCC     NOCY    Kontrollera om summan > 255
        INC     PHI     Ja, öka innehållet i partialprodukt hög byte
NOCY    DEC     XVAR    Minska multiplikatorn med ett (nu en räknare)
        BNE     LOOP    Upprepa tills räknaren = 0
        …               Färdigt
```

Heltalsmultiplikation genom addition och skift P(16) = X(8)*Y(8)

Multiplikatorn X(8) finns från början i M(XVAR)
Multiplikanden Y(8) finns i M(YVAR)

Produkten P(16) finns efter multiplikationen i A:M(PLO)   (Hög del i register A och låg del på adress PLO)

Register A och M(PLO) har från början värdet 0
M(COUNT) är en skifträknare, som från början har värdet 8

```
LOOP    LSR     XVAR    Minst signifikant bit i X (lsb) till carry
        BCC     NOADD   Carry = 0. Ej addition
        ADDA    YVAR    Carry = 1. Addera Y till produkt hög byte
NOADD   RORA            Skifta produkten (16 bitar) ett steg åt höger
        ROR     PLO                     -  "  -
        DEC     COUNT   Ett skift mindre
        BNE     LOOP    8 skift? Nej.
        …               Ja. Färdigt
```

Heltalsdivision genom upprepad subtraktion Y(16)/X(8) = Q(16) + R(8)/X(8)

Dividenden Y(16) finns från början i M(YHI):M(YLO) (Hög del på adress YHI och låg del på adress YLO)
Divisorn X(8) finns i M(Xval)

Kvoten Q(16) finns efter divisionen i M(QHI):M(QLO) (Hög del på adress QHI och låg del på adress QLO)
Resten R(8) finns efter divisionen i M(REST)

```
LOOP    LDA     YLO     Hämta låg byte av dividend
        SUBA    Xval    Subtrahera divisor
        STA     YLO     Uppdatera YLO

        LDA     YHI     Hämta hög byte av dividend
        SBCA    #0      Drag bort ev borrow från YHI
        STA     YHI     Uppdatera YHI

        BLO     READY   Division färdig

        INC     QLO     Ej klart, öka låg byte av kvoten med 1
        BNE     LOOP    Fortsätt om ej carry till hög byte av kvoten

        INC     QHI     Carry till hög byte
        BRA     LOOP

READY   LDA     YL      Hämta rest (nu negativ)
        ADDA    Xval    Justera rest (addera divisor)
        STA     REST    Uppdatera rest
          …             Färdigt
```

Heltalsdivision genom subtraktion och skift Y(16)/X(8) = Q(16) + R(8)/X(8)

Dividenden Y(16) finns från början i M(YHI):M(YLO) (Hög del på adress YHI och låg del på adress YLO)
Divisorn X(8) finns i M(Xval)

Kvoten Q(16) finns efter divisionen i M(QHI):M(QLO) (Hög del på adress QHI och låg del på adress QLO)

Resten R(8) finns efter divisionen i register A och M(REST)

Register A har från början värdet 0
M(COUNT) är en skifträknare, som från början har värdet 16

```
LOOP    LSL     YLO     2*dividend (msb till C-flaggan)
        ROL     YHI          -" -

        ROLA            MS-bit till A (C-flaggan till A-registret)
        BCS     SUBTR   Overflow i A-reg! Subtrahera divisor från dividendens ms-bitar

        CMPA    Xval    Skall vi subtrahera divisorn från ms-bitar från dividenden?
        BHS     SUBTR   Ja!

        LSL     QLO     Nej, skifta kvoten vänster med nolla in från höger
        ROL     QHI     Hög byte
        BRA     NEXT

SUBTR   SUBA    Xval    Utför subtraktionen. Blir korrekt även vid overflow!
        ORCC    #1      Sätt carry (ger etta i kvoten)
        ROL     QLO     Skifta kvoten vänster med etta in från höger
        ROL     QHI                  -" -

NEXT    DEC     COUNT   Skifträknare
        BNE     LOOP    Färdigt? Nej

        STA     REST    Ja. Uppdatera rest
          …             Färdigt
```