Digitalteknik

Michael Josefsson

Version 0.41, 2021-08-25

Innehåll

I. Kombinatoriska nät 5

1. Logik 7
1.1. Logisk ekvation 8
1.2. Boolesk algebra 10
1.3. NAND- och NOR-grindar 15
1.4. Booleska axiom och räknelagar 16
1.4.1. Axiom 17
1.4.2. Associativa lagen 17
1.4.3. Kommutativa lagen 18
1.4.4. Absorptionslagen 18
1.4.5. Distributiva lagen 18
1.4.6. Consensuslagen 18
1.4.7. De Morgans lag 18

2. Binära tal och kodningar 21
2.1. Positionssystem 21
2.2. Andra talkodningar 23
2.2.1. BCD, Binary Coded Decimal 23
2.2.2. Andra BCD-kodningar 24
2.2.3. Gray-kodning 25

3. Kombinatorik, normalform och minimering 27
3.1. SP-normalform 27
3.2. Minimering 28
3.2.1. Grafisk metod för minimering, ”karnaughminimering” . . . 29
3.2.2. Hur ringar man? 31
3.2.3. Sammanfattning av SP-form 33
3.2.4. *Don’t cares*, DC 33
3.3. Binär addition. Tillämpning av kombinatorik. 35
3.3.1. Realisering med enbart NAND-grindar 38
3.3.2. Realisering av trebitars adderare 38
3.3.3. PS-normalform 38

II. Sekvensiella nät 43

4. Sekvensnät 45
4.1. Vippan 45
4.1.1. Vippkonstruktioner 46
4.2. Tillståndsgraf 47
4.3. Sekvensnät 48
4.4. Mealy- och Moorenät 52
4.4.1. Mealy-typ 52
4.4.2. Moore-typ 53
4.5. Synkronism 54
4.6. Sekvensnät med insignal 55
4.7. Typexempel Moore 58
4.8. Typexempel Mealy 60
4.9. Ett mått på komplexitet 61
4.10. Grinddelning 62
4.11. Skillnad mellan Moore- och Mealy-nät 62

5. Större sekvensnät 65
5.1. Tillståndsminne 65
5.2. Automat 66
5.3. Strukturerade lösningar 67

Del I.

Kombinatoriska nät

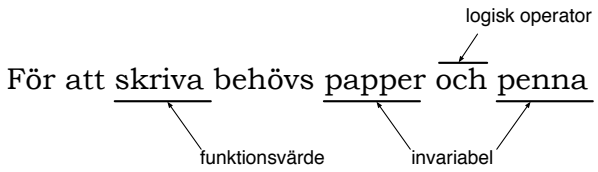
1. Logik

Vi använder logik hela tiden och det är så naturligt att vi inte ens tänker på det: När vi säger saker som ”i rummet finns en stol och en byrå”, ”höger och vänster helljus fungerar” gör vi logiska utsagor. Dessa är inte bara exempel på logiska utsagor utan kan dessutom beskrivas med en logik som har blott *två tillstånd*: Ja/nej, av/på, sant/falskt osv.

Påståendet ”i rummet finns en stol och en byrå” betyder att det finns just *en* stol och *en* byrå i rummet. Inte ”*en* stol och *ingen* byrå” eller ”*ingen* stol och *en* byrå” eller ”*ingen* stol och *ingen* byrå”.

Exempel 1 Satslogik i vardagligt språk

Påståendet ”För att skriva behövs papper och penna” innehåller både en *invariabel* (papper, penna), *funktionsvärde* (skriva) och en *logisk operator* OCH. Påståendet betyder så klart att det behövs både papper OCH penna för att kunna skriva. Enbart det ena *eller* det andra räcker inte.

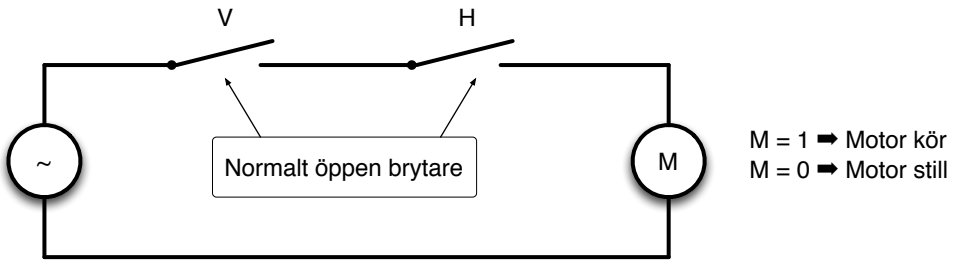


■

Exempel 2 Död mans grepp

För att inte klippa av sig händerna i en motordriven elektrisk vedklyv används ”död mans grepp”, dvs maskinen går inte att starta om man inte har vänster OCH höger hand på startknapparna, vilka är väl skiljda från vedklyvens rörliga delar.

Vi kan rita ett elektriskt schema över situationen:



I figuren förekommer signalerna M för motorn, V för vänster startknapp och H för höger startknapp:

M	Betydelse	V	Betydelse	H	Betydelse
0	Motor stopp	0	Vänster öppen	0	Höger öppen
1	Motor start	1	Vänster sluten	1	Höger sluten

1.1. Logisk ekvation

Med signalerna ovan kan vi skriva en ekvation för utsignalen M:

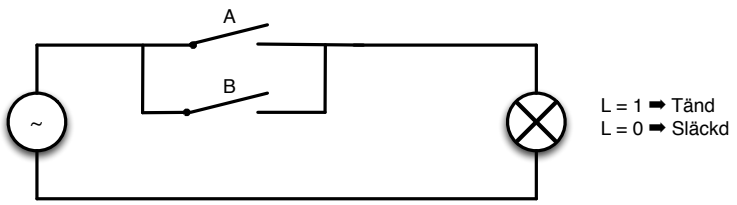
$$M = V \text{ OCH } H = V \cdot H = VH$$

Här har vi dessutom skrivit "·" istället för OCH. Att göra detta är naturligt om vi tänker oss att de ingående variabler är antingen 1 eller 0. Då kommer ju en multiplikation ge samma resultat som vårt tidigare OCH. Slutligen har vi, precis som i vanlig algebra, underförstått multiplikationen och kan helt sonika skriva ihop variablerna till produkten "VH".

Exempel 3 ■

> Man vill kunna tända en lampa med två knappar, knapp A eller knapp B. Rita ett schema över situationen och beskriv med algebra.

Här kan man alltså tända lampan L från knapp A *eller* knapp B.



Som vanligt har en *aktiv/påverkad* brytare värdet 1 medan en *oaktiverad/opåverkad* brytare har värdet 0. För att tända lampan L krävs att L=1.

En ekvation för detta schema kan skrivas

$$L = A \text{ ELLER } B = A + B$$

det vill säga L=1 om A=1 ELLER B=1 . Dessutom är lampan tänd om *båda* knapparna nedtryckta. **I digitaltekniken är inte 1+1=2 utan 1+1=1!**

Det sista kan upplevas något förvirrande till en början med med lite övning kommer man snart in i det rätta tänket.

■

Exempel 4 Invertering

För att ge en busschaufför klartecken, K, att köra krävs

1. Dörrarna, D, är stängda
2. Ingen trycker på STOP-knappen, S

Med signalerna D och S definierade enligt

D	Betydelse
0	Dörr öppen
1	Dörr stängd

S	Betydelse
0	Ingen har tryckt på STOP
1	Någon har tryckt på STOP

■

kan vi skriva en ekvation för signalen K:

$$\mathtt{K} = \mathtt{D} \cdot \overline{\mathtt{S}}$$

Här har S försetts med ett *inverteringsstreck* som byter betydelsen hos signalen. $\overline{\mathtt{S}}$ är 1 om S=0.[1]

Ekvationen kan tolkas som att klarsignalen är 1 då dörren stängd (D=1) och *ingen* S är för handen. Ingen S betyder ju S=0 och alltså blir $\overline{\mathtt{S}} = 1$. Vilket ger K = 1·1 = 1 och chauffören kan få klartecken.

■

En buss har ju i allmänhet flera dörrar. Om vi kallar varje dörr för d_i kan ett samlat uttryck för D skrivas

$$\mathtt{D} = d_1 \cdot d_2 \cdot d_3 \cdot d_4 \cdot \ldots$$

som innebär att alla dörrar måste vara stängda för att D skall vara 1.

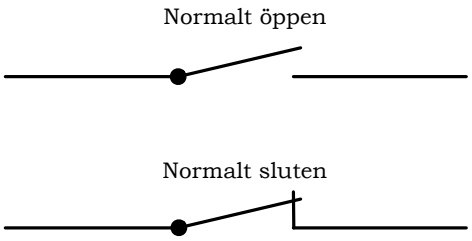
En buss har dessutom ofta flera STOP-knappar, s_i. Ett korrekt uttryck för S kan då skrivas

$$\mathtt{S} = s_1 + s_2 + s_3 + s_4 + \ldots$$

som kan tolkas som att det räcker att *någon* tryckt på STOP för att hela uttrycket skall bli sant. Så länge ingen tryckt på STOP är S noll, dvs $\overline{\mathtt{S}} = 1$. Så fort någon tryckt STOP blir S=1 och $\overline{\mathtt{S}} = 0$. Detta är precis vad vi önskar.

En vanlig brytare, till exempel S, är *normalt öppen*. För att rita $\overline{\mathtt{S}}$-brytaren använder vi istället en som är *normalt sluten*. De ritas i scheman som s_i respektive $\overline{s_i}$ nedan

[1] Vi har alltså $\overline{0} = 1$ och $\overline{1} = 0$. En del författare skriver dessutom inte inverteringsstreck utan ett *prim*-tecken, $\overline{0} = 0' = 1$. Här använder vi uteslutande inverteringsstreck.



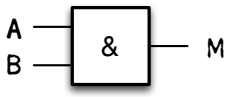
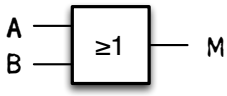
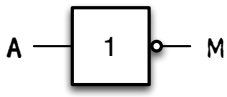
Övning: Rita ett schema som visar hur klarsignalen skall kopplas om det finns flera olika dörrar och stoppknappar. Börja med ekvationen $K = D \cdot \overline{S}$ och komplettera sedan med signalerna d_i och s_i. Denna övning kan vara lite knepig men gå inte vidare innan du gjort ett ordentligt försök.

1.2. Boolesk algebra

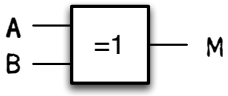
De ekvationer vi tecknade ovan är exempel på *booleska ekvationer*. Vi kommer använda booleska uttryck hela tiden i digitaltekniken och dessutom ta till oss en *boolesk algebra* för att kunna manipulera dessa booleska uttryck.

Den booleska algebran innehåller operatorerna OCH, ELLER, ICKE och EXKLUSIVT ELLER. De första tre första har vi redan använt (ICKE är inverteringsstrecket, som i $\overline{S}$).

Dessa utgör samtliga vanliga *grindfunktioner* och är sammanställda nedan i var sin *sanningstabell*. En sanningstabell beskriver förhållandet mellan samtliga in- och utsignaler för ett booleskt uttryck/funktion.

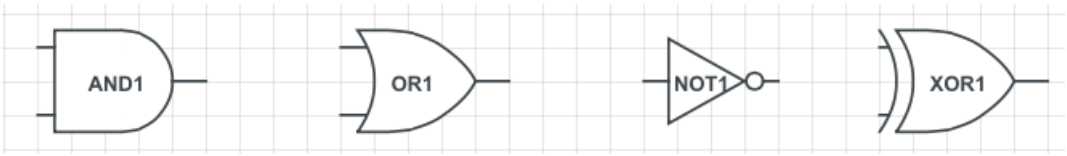
Logisk funktion	Uttryck	Schemasymbol	Sanningstabell
OCH / AND	$M = A \cdot B$		AB | M 00 | 0 01 | 0 10 | 0 11 | 1
ELLER / OR	$M = A + B$		AB | M 00 | 0 01 | 1 10 | 1 11 | 1
ICKE / NOT	$M = \overline{A}$		A | M 0 | 1 1 | 0

EXKLUSIVT ELLER är en mycket viktig variant av ELLER med den skillnaden att uttrycket är 1 om *enbart* ***en*** insignal är 1:

Logisk funktion	Uttryck	Schemasymbol	Sanningstabell
EXKLUSIVT ELLER/XOR	$M = A \bigoplus B$		AB | M 00 | 0 01 | 1 10 | 1 11 | 0

NOTera hur grindsymbolen ser ut och att den faktiskt betyder exakt det grinden gör. "≥ 1" innebär att utsignalen är 1 när summan av insignalerna är ett eller mer, "$= 1$" att *exakt* en insignal skall vara 1. I kursen ritar vi grindsymbolerna på detta sätt enligt IEC-standarden.

Även symboler enligt amerikanska ANSI-standarden förkommer. Det ingår i kursen att känna till dessa också:



- AND, OR och NOT förekommer ofta tillsammans. Det finns goda anledningar till att kombinera ihop NOT-AND, NOT-OR osv till NAND- och NOR-grindar.
- NOT behövs, ty inte med bästa vilja i världen är det möjligt att utföra en invertering med enbart AND och OR.
- XOR är en bastard i sammanhanget då den inte går att bevisa med hjälp av de övriga. Den är dock inte mindre sann för det och i digitalteknik-sammanhang mycket flitigt använd som vi kommer att se.

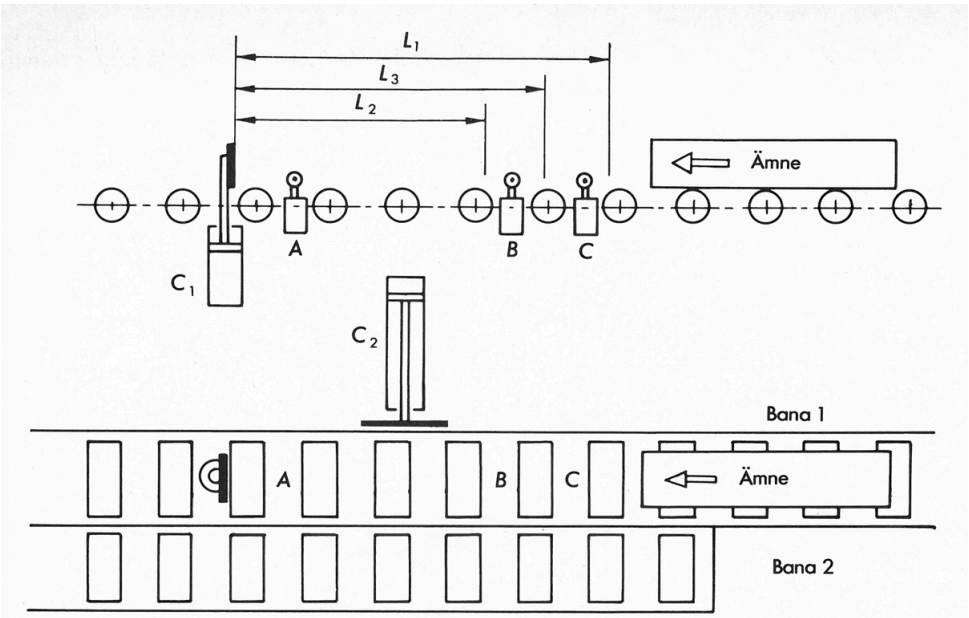
Nu är vi redo för en övning!

Exempel 5: Stålavskiljare

Stålämnen som matas fram på en rulltransportör skall sorteras i tre längder[2]

- L_1 För lång
- L_2 För kort
- L_3 Rätt längd

Se figuren nedan. I rullbana 1 är givarna A, B och C placerade med vars hjälp de olika längderna kan avkännas. För långa och för korta ämnen skall skjutas åt sidan till bana 2 med hjälp av tryckluftcylinder C_2. Cylinder C_1 stoppar ämnena och släpper fram sådana med rätt längd.



För ämneslängderna L_1 är A, B och C påverkade. För ämneslängderna L_2 är A påverkad och B och C opåverkade. För ämneslängderna L_3 är A och B påverkade men C opåverkad.

I figuren tänker vi oss tryckluftscylindrarna i läget som motsvarar styrsignalen 0. Med styrsignal 1 till C_1 går C_1 tillbaka och styrsignal 1 till C_2 trycker C_2 framåt.

Hur ska styrsignalerna till C_1 och C_2 konstrueras? Fundera själv på detta innan du läser vidare!

[2]Ur Jan Ståhl, *Reglerteknik, B Styrteknik*

Exempel 5 Stålavskiljare forts.

Möjliga insignaler till konstruktionen är A, B och C, utsignaler är C_1 och C_2. Vi ställer upp alla insignalkombinationer i en sanningstabell:

ABC	C_1	C_2
000		
001		
010		
011		
100		
101		
110		
111		

Tre insignaler ger totalt $2^3 = 8$ insignalkombinationer och de är uppräknade i vänsterkolumnen. Själva konstruerandet handlar nu om att fylla i de två högra kolumnerna. Vi gör det systematiskt uppifrån och ner och bestämmer vad varje insignalkombination betyder i stålavskiljaren.

Resonemanget går så här:

- ”Om $ABC = 000$ har inget ämne kommit fram än. Då ska vi inte aktivera varken C_1 eller C_2 dvs sätt deras respektive värden i tabellens första rad till $0\,0$.”
- ”Samma gäller för insignalerna $ABC = 001, 010, 011$. C_1 eller C_2 utförs som $0\,0$ även för dessa rader.”
- Vi kommer sakta fram till att $ABC = 100, 110, 111$ betyder att ämnet är för kort, har rätt längd respektive är för långt och fyller i C_1- och C_2-kolumnerna därefter.
- Vad betyder egentligen $ABC = 101$?

Slutlig tabell blir (med booleska algebraiska uttryck för respektive rad längst till höger)

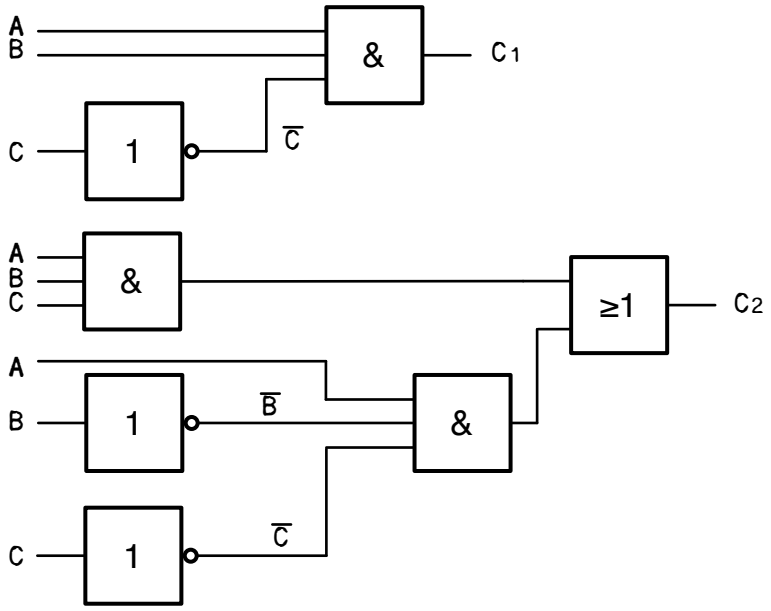
ABC	C_1	C_2	algebra
000	0	0	$\overline{A}\,\overline{B}\,\overline{C}$
001	0	0	$\overline{A}\,\overline{B}\,C$
010	0	0	$\overline{A}\,B\,\overline{C}$
011	0	0	$\overline{A}\,B\,C$
100	0	1	$A\,\overline{B}\,\overline{C}$
101	0	0	$A\,\overline{B}\,C$
110	1	0	$A\,B\,\overline{C}$
111	0	1	$A\,B\,C$

Ur tabellen kan vi direkt se vilka insignalkombinationer som ger de önskade `1`-orna. Vi behöver bara leta efter de som ger utsignalen `1`, övriga ger ju alltid `0` ut:

$$C_1 = A\,B\,\overline{C}$$
$$C_2 = A\,B\,C + A\,\overline{B}\,\overline{C}$$

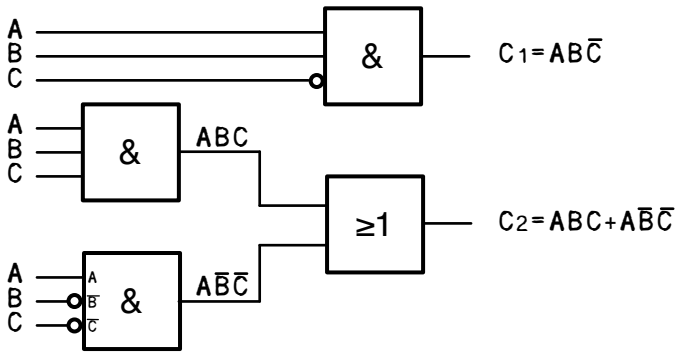
Observera att $\overline{B}\,\overline{C} \neq \overline{B\,C}$. Detta är ett vanligt nybörjarfel.

Med ekvationerna skrivna kan vi direkt realisera konstruktionen med schemasymboler:



Här kan man följa signalerna ABC från vänster till höger till utresultatet C_i.

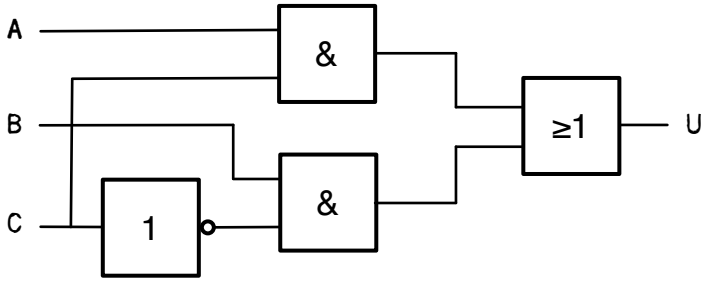
För att förenkla både schemaritandet och läsandet brukar man förenkla inverteraren (NOT-grinden) till enbart *inverteringsringen*:



Detta förenklade ritsätt är önskvärt. Lägg också märke till att signalnamnen mycket väl kan förekomma flera gånger till vänster om så förenklar ledningsdragningen. Allt för att få ett så tydligt schema som möjligt.

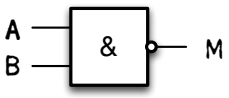
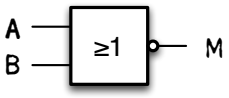
Övning: Schema till booleskt uttryck.

Bestäm ekvationerna och avslutande sanningstabell för nätet nedan. Beskriv dess funktion i ord?



1.3. NAND- och NOR-grindar

Vi nämnde att kombinationer av de grundläggande grindarna är vanliga. De mest förekommande är NAND- och NOR-grindar med schemasymbol enligt nedan:

Logisk funktion	Uttryck	Schemasymbol	Sanningstabell
ICKE-OCH / NAND	$M = \overline{A \cdot B}$		AB | M 00 | 1 01 | 1 10 | 1 11 | 0
ICKE-ELLER / NOR	$M = \overline{A + B}$		AB | M 00 | 1 01 | 0 10 | 0 11 | 0

Man kan visa att **all** logik kan konstrueras med enbart NAND- och/eller NOR-grindar.

Övning: Tillverka en inverterare med hjälp av NAND- respektive NOR-grind. Hur gör du en XOR-funktion med dessa NAND respektive NOR?

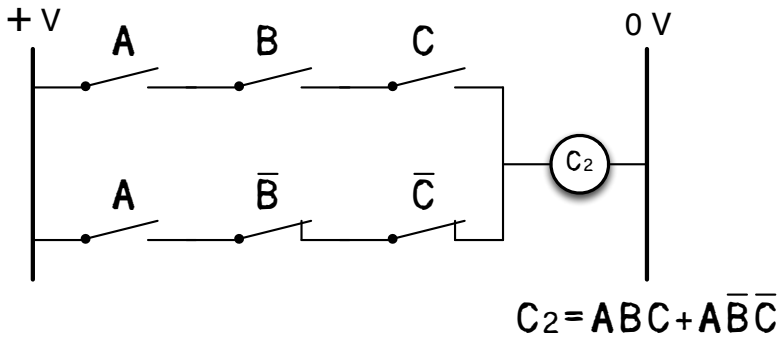
1.4. Booleska axiom och räknelagar

Vi måste hela tiden hantera digitala uttryck. Här visas inledningsvis ekvationer och räknelagar med hjälp av elektriska scheman men man måste lära sig dem utantill rent algebraiskt. Med hjälp av räknelagar kan man utföra effektiva algebraiska manipuleringar av logiska uttryck.

Exempel 6

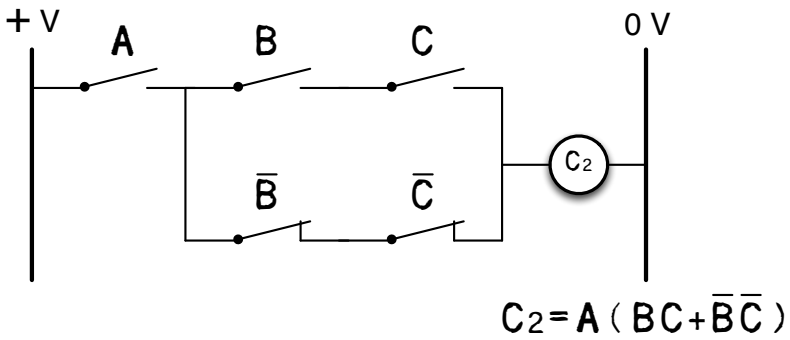
Cylinderekvationen C_2 ur exempel 5 blev $C_2 = A\,B\,C + A\,\overline{B}\,\overline{C}$.

Vi ritar ett schema över denna ekvation:



Detta är ett så kallat *reläschema* där utsignalen är en reläspole som kan påverka ytterligare reläer. Vi kan se denna utsignal som ett algebraiskt deluttryck.

Man ser att schemat, då de båda A-brytarna alltid går i parallell, kan förenklas något om det ritas som



Där vi sparade in kostnaden för en brytare. Ur schemat kan vi skriva ekvationen för C_2 på ett annat sätt, med parenteser.

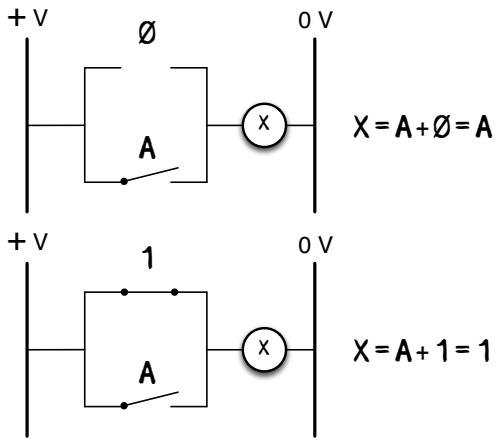
■

Man kan nu ana att det finns räknelagar för booleska uttryck. Det borde gå att manipulera uttryck utan att behöva rita upp ett elektriskt schema varje gång?

Svaret är ja och på samma sätt som i den vanliga algebran utgår boolesk algebra från några axiom varefter resten följer.

1.4.1. Axiom

Axiomen exemplifieras utgående från några scheman:



Där vi enkelt kan se att en parallellkopplad öppen brytare (”0”) inte påverkar uttrycket och att en parallellkopplad kortslutning (”1”) över en brytare alltid trumfar resten av uttrycket och dominerar.

På liknande sätt kan man övertyga sig om att även följande regler gäller:

$$\begin{array}{cc} 0 + A = A & 0 \cdot A = 0 \\ 1 + A = 1 & 1 \cdot A = A \\ A + A = A & A \cdot A = A \\ A + \overline{A} = 1 & A \cdot \overline{A} = 0 \end{array}$$

Vi fortsätter nu på en inslagna vägen och presenterar ett antal räknelagar för boolesk algebra. Lägg märke till att lagarna förekommer parvis, en med ELLER-operatorer (till vänster nedan) och en annan med OCH-operatorer, ett *dualt* uttryck (till höger).

1.4.2. Associativa lagen

Parenteser fungerar som vanligt:

$(x + y) + z = x + (y + z)$ $\qquad$ $(xy)z = x(yz)$

1.4.3. Kommutativa lagen

Termernas (eller faktorernas) ordning är oväsentlig:

$x + y = y + x$ $\qquad$ $xy = yx$

1.4.4. Absorptionslagen

Man kan multiplicera in och bryta ut som vanligt:

$x + xy = x\underbrace{(1 + y)}_{=1\forall y} = x \cdot 1 = x$ $\qquad$ $x(x + y) = \underbrace{xx}_{=x} + xy = x + xy$

1.4.5. Distributiva lagen

Ur absorptionslagen får man också:

$x(y + z) = xy + xz$ $\qquad$ $x + yz = (x + y)(x + z) = \underbrace{xx}_{=x} + \underbrace{xz + xy}_{=0\ om\ x=0} + yz = x + yz$

1.4.6. Consensuslagen

Överraskande kan man i vissa fall lägga till termer utan att uttrycket förändras:

$xy + \overline{x}z = xy + \overline{x}z + \underbrace{yz}_{!!}$ $\qquad$ $(x + y)(\overline{x} + z) = (x + y)(\overline{x} + z)\underbrace{(y + z)}_{!!}$

Där den så kallade *consensustermen* är markerad med "!!". Consensuslagen är ofta användbar då man manipulerar uttryck och kört fast och behöver fler variabler för att kunna fortsätta.

Sammantaget tillåter den booleska algebran de vanliga manipulationerna men speciell uppmärksamhet måste ges på att $1 + 1 = 1$ och consensuslagen.

Övning: Rita reläschema för lagarna ovan! Övertyga dig om att de stämmer.

1.4.7. De Morgans lag

De ovanstående lagarna går att bevisa utifrån axiomen. De Morgans lag kan inte härledas från axiomen men kan konstateras utifrån sanningstabell. Även denna lag finns i två varianter:

$$\overline{x + y} = \overline{y} \cdot \overline{x} \qquad \overline{x \cdot y} = \overline{y} + \overline{x}$$

Övning: Den självklara övningen här är att fylla i sanningstabellen för dessa två uttryck för Morgans lag.

Räknelagarna **måste** nötas in så de sitter och kan hanteras lika smärtfritt som vanlig algebra.

Exempel 7 Förenkling av uttryck

Man vill förenkla uttrycket

$$x = A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}C + AB\overline{C}$$

Det går att använda räknelagarna ovan. Försök gärna det.

Ett alternativ är att fylla i en sanningstabell för $x = x(A, B, C)$. Vi noterar att uttrycket för x har fyra termer. Dessa kan antingen vara `1` eller `0` och det räcker att **en** term är `1` för att **hela** uttrycket skall vara `1`.

Det räcker således att hitta dessa fyra ettor, fylla i tabellen med dem och utföra resten med nollor. Det är en mycket arbetsbesparande strategi!

Tabellen blir:

ABC	x	term
0 0 0	0	
0 0 1	0	
0 1 0	1	$\overline{A}\,B\,\overline{C}$
0 1 1	0	
*1 0 0	1	$A\,\overline{B}\,\overline{C}$
*1 0 1	1	$A\,\overline{B}\,C$
1 1 0	1	$A\,B\,\overline{C}$
1 1 1	0	

Betrakta nu de med "*" markerade raderna $ABC = 100$ och 101. Båda dessa har utsignalen 1 oavsett vad C har för värde! Uppenbarligen spelar C:s värde ingen roll och de resterade två variablerna kan skrivas som $A\overline{B}$. Med algebra kan detta skrivas

$$A\overline{B}\,\overline{C} + A\overline{B}C = A\overline{B}(\overline{C} + C) = A\overline{B} \cdot 1 = A\overline{B}$$

På samma sätt kan de resterande ettorna slås samman till $B\overline{C}$ då A är irrelevant för dem.

Efter förenkling får vi till slut

$$x = A\overline{B} + B\overline{C}$$

som inte kan förenklas ytterligare.

■

Övning: Rita det till ekvationen för x hörande schemat med grindlogik (och reläer om andan faller på).

Exempel 8 NAND- och NOR-ifiering

Tidigare har nämnts att all möjlig grindlogik går att realisera med enbart NAND- eller NOR-grindar. Så här går det till med uttrycket från exemplet ovan

$$x = A\overline{B} + B\overline{C}$$

som inte innehåller någon uppenbar NAND- eller NOR-grind. Men med de Morgans lag kan uttrycket skrivas

$$x = \overline{\overline{x}} = \overline{\overline{\overline{A\overline{B} + B\overline{C}}}} = \overline{\overline{A\overline{B}} \cdot \overline{B\overline{C}}}$$

som enbart består av NAND-logik:

$$\overbrace{\overline{\underbrace{\overline{A\overline{B}}}_{\text{NAND}} \cdot \underbrace{\overline{B\overline{C}}}_{\text{NAND}}}}^{\text{NAND}}$$

Fortsätter man med de Morgan får man till slut

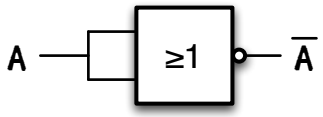
$$x = \overline{\overline{A\overline{B}} \cdot \overline{B\overline{C}}} = \overline{(\overline{A} + B) \cdot (\overline{B} + C)} = \overline{(\overline{A} + B)} + \overline{(\overline{B} + C)}$$

som på motsvarande sätt är samma logiska funktion realiserad med NOR-grindar:

$$\overbrace{\underbrace{\overline{(\overline{A} + B)}}_{\text{NOR}} + \underbrace{\overline{(\overline{B} + C)}}_{\text{NOR}}}^{\text{OR}}$$

som då också kräver en avslutande invertering eftersom sista steget i grindlogiken blir en OR-grind.

En OR-funktion realiseras enkelt med en NOR-grind följt av en inverterare. En inverterare fås ur en NOR-grind kopplad enligt



En NAND-grind kan användas på motsvarande sätt. Övertyga dig om detta med hjälp av sanningstabellen.

2. Binära tal och kodningar

All uppräkning av bitar i sanningstabellerna hittills har skett enligt det binära talsystemet. Det är det normala sättet och vi måste bekanta oss med detta talsystem tills det känns naturligt för oss.

Vi kommer också här presentera det ofta använda *hexadecimala* talsystemet med basen 16.

Här kommer vi bara behandla positiva heltal.

2.1. Positionssystem

Glädjande nog är det inget speciellt konstigt med det binära talsystemet, det fungerar precis som vårt vanliga decimalsystem — fast med binära tal då. Båda är positionsssytem och för att se likheterna betraktar vi först ett tal i det decimala talsystemet och efter det ett tal i det binära. Observera att likheterna är större än skillnaderna!

Exempel 9 Decimala talsystemet.

I det decimala talsystemet har vi som bekant tio symboler (siffror), $0 - 9$. Sedan tar det slut och vi måste lägga till en tiotalssiffra innan vi börjar om igen med entalssiffran för att få $10 - 19$ osv.

Med basen tio skrivs talet 394 som

$$394_{10} = 3 \cdot \underline{10}^2 + 9 \cdot \underline{10}^1 + 4 \cdot \underline{10}^0 = 3 \cdot 100 + 9 \cdot 10 + 4 \cdot 1$$

Den faktiska talbasen har här för tydlighets skull markerats med en understrykning. Exponenterna är *positionsvikter* och räknas från höger 0, 1, 2,. . .

Exempel 10 Binära talsystemet

Det binära talsystemet utförs på samma sätt men det finns bara två symboler `0` och `1`, dvs basen är 2.

Talet `101` med basen 2 motsvarar i analogi med det decimala fallet

$$101_2 = 1 \cdot \underline{2}^2 + 0 \cdot \underline{2}^1 + 1 \cdot \underline{2}^0 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$$

Notera att positionsvikterna är desamma (0, 1, 2) som tidigare men basen är ändrad. Summerar vi högerledet får vi summan 5 och kan konstatera att

$$101_2 = 5_{10}$$

Där den nedsänkta siffran anger talets bas.[1]

Ett godtyckligt binärt tal N av längden $p+1$ kan enligt ovan skrivas

$$N = \sum_{0}^{p} x_i 2^i \qquad\qquad x_i = \{0, 1\}$$

Allmänt kan vi konstruera ett tal i vilken bas, b, som helst på samma sätt:

$$N = \sum_{0}^{p} x_i b^i \qquad\qquad x_i = \{0, 1, \ldots, b-1\}$$

Övning: Hur stort blir N maximalt för $p = 1, 2, 3, \ldots$ osv? Antag binär talbas.

Fyra bitar ihop i en grupp brukar kallas en *nibble*. Vi kommer använda just grupper om fyra ofta, både i digital- och datorteknikkurserna. I en tabell kan kan de binära bitarna räknas upp tillsammans med sitt decimala värde:

Decimal	Binär 8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)	BCD	Hexa-decimal
0	0	0	0	0	0	0
1	0	0	0	1	1	1
2	0	0	1	0	2	2
3	0	0	1	1	3	3
4	0	1	0	0	4	4
5	0	1	0	1	5	5
6	0	1	1	0	6	6
7	0	1	1	1	7	7
8	1	0	0	0	8	8
9	1	0	0	1	9	9
10	1	0	1	0	–	A
11	1	0	1	1	–	B
12	1	1	0	0	–	C
13	1	1	0	1	–	D
14	1	1	1	0	–	E
15	1	1	1	1	–	F

Förutom den binära motsvarigheten har här också lagts till hexadecimal tolkning såväl som BCD *binary coded decimal*.

[1] Basen anges alltid i decimal form. Varför?

Exempel 11

Översätt det hexadecimala talet B3 till decimalt och binärt tal.

Med kännedom om tabellen ovan blir den första omvandlingen enkel. Hexadecimala tal har basen 16, för övrigt gör vi precis som tidigare:

$$\texttt{B3} = \texttt{B} \cdot 16^1 + 3 \cdot 16^0 = 11 \cdot 16 + 3 \cdot 1 = 176 + 3 = 179_{10}$$

Det hexadecimala talet B3 är alltså 179 decimalt.

Ett alternativ är att skriva B3 binärt och sedan bedöma det utifrån basen 2:

$$\texttt{B3} = \texttt{10110011} = 1 \cdot 2^7 + 0 \cdot 2^6 + \ldots + 1 \cdot 2^1 + 1 \cdot 2^0 = 128 + 32 + 16 + 2 + 1 = 179_{10}.$$

Svaret blir naturligtvis 179 även denna gång och metoden ger en förklaring till att basen 16 (och basen 8 förr) är synnerligen lämplig i digitala sammanhang:

$$\texttt{B3} = \underbrace{\overbrace{\texttt{1011}}^{\texttt{B}\cdot 16^1}\overbrace{\texttt{0011}}^{3\cdot 16^0}}_{128\,64\,32\,16\,8\,4\,2\,1}$$

I sin position är B värd exakt 16 gånger mer än om den skulle stått längst till höger. En multiplikation med 16 är samma som multiplikation med 2^4 dvs talet vänster skiftat fyra bitar. Och det är ju precis det som skett ovan!

■

Övning: Omvandla $\texttt{B3}_{16}$ till basen 8 med metoden ovan.

Övning: Vad händer om basen är till exempel 5? Varför ställer det till problem?

2.2. Andra talkodningar

Förutom rena ”binära” binärkodningar används ofta binärkodade decimaltal och gray-kod.

2.2.1. BCD, Binary Coded Decimal

Med BCD-kodade tal menas att man enbart använder siffrorna 0–9 ovan och på det vanliga, decimala sättet.

Exempel 12 Decimal -> BCD-tal

Hur skrivs talet 179_{10} som BCD-tal?

Ur tabellen läser vi av siffra för siffra:

$$179 = \overbrace{\underbrace{0001}_{1}}^{hundratal}\ \overbrace{\underbrace{0111}_{7}}^{tiotal}\overbrace{\underbrace{1001}_{9}}^{ental}$$

Det finns inget i själva talet som avslöjar att det är BCD-kodat, det får man hålla ordning på själv.

Exempel 13 BCD -> decimaltal

Tolka BCD-talet `100101000` som ett decimaltal

Gruppera talet i grupper om fyra bitar från höger och fyll på med nollor till vänster om det skulle behövas till en jämn fyrgrupp

$$\underbrace{000}_{extra}100101000$$

Ur tabellen får man sedan som förut

$$\underbrace{0001}_{1}\underbrace{0010}_{2}\underbrace{100}_{8} = 128_{10}$$

2.2.2. Andra BCD-kodningar

Med varje BCD-siffra uppbyggd av positionsvikterna 8421 kallas kodningen egentligen NBCD, ***Naturally*** *Binary Coded Digit*. Andra kodningar finns, till exempel 5421 eller 2421 men kallas då inte naturlig utan just ”5421-kodning” respektive ”2421-kodning”. I dagligt tal avses dock NBCD när man nämner BCD.

	NBCD		
Decimal	8421	5421	2421
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	1100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111

2.2.3. Gray-kodning

Med graykodning menas en kodning där de olika talen, vid uppräkning, *skiljer sig åt med endast en bit* i varje steg. Detta är emellanåt en önskvärd egenskap.

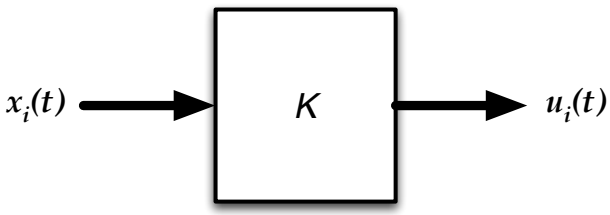
Graykodningen tillhör en grupp av kodningar som kallas *speglade*. En egenskap som kan skönjas i tabellen nedan där koden i tur och ordning är speglad i de horisontella strecken under tillägg av en mer signifikant bit.

Tabellen kan med fortsatt spegling sätt utökas till godtycklig bit-bredd.

Decimal	Gray
0	000
1	001
2	011
3	010
4	110
5	111
6	100
7	101

3. Kombinatorik, normalform och minimering

Utan att vi tänkt speciellt på det har vi hittills betraktat nät och logikfunktioner som system, K, med en eller fler insignaler $x_i(t)$ och en eller flera utsignaler $u_i(t)$. Det har varit underförstått att nätet har varit *momentant*, omedelbart. Det har reagerat utan fördröjning, som en uppslagning i en tabell. Nät med denna egenskap kallar vi *kombinatoriska*.



3.1. SP-normalform

Med SP-normalform menar man ett booleskt funktionsuttryck som innehåller *alla* variabler och *alla* termer uttryckta som ”summa av produkt”. Termer som alltid blir 0 utesluts dock.

Exempel 14

> Ett kombinatoriskt nät med insignalerna $\mathcal{X} = \{x_3x_2x_1x_0\}$ har en utsignal $lo3 = 1$ då $\mathcal{X}$ är mindre än 3, dvs för insignalerna 0, 1 och 2. $lo3 = 0$ för övrigt.

I *SP-normalform* kan det booleska uttrycket för $lo3$ skrivas som en uppräkning av de tre önskade ettorna:

$$lo3 = \overline{x_3}\,\overline{x_2}\,\overline{x_1}\,\overline{x_0} + \overline{x_3}\,\overline{x_2}\,\overline{x_1}x_0 + \overline{x_3}\,\overline{x_2}x_1\overline{x_0}$$

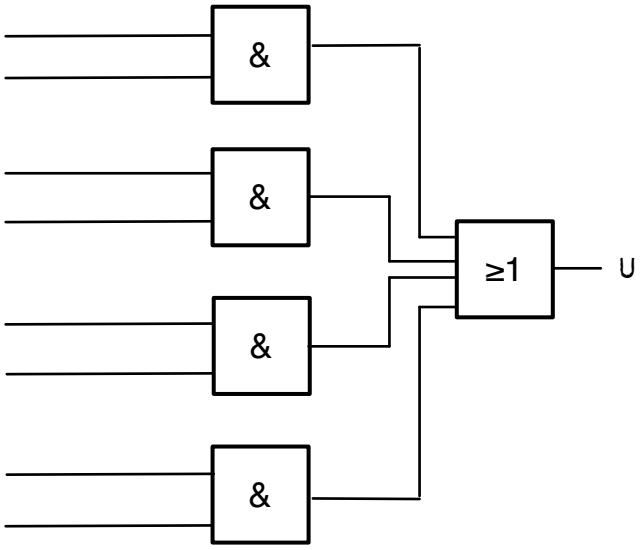
Varje *term* i uttrycket ovan kallas en *min-term* och motsvarar bitarna

$$\mathcal{X} = \{0000, 0001, 0010\}$$

Konstatera själv att

- uttrycket innehåller *alla* variabler och *alla* termer (som blir något), och
- uttrycket verkligen är en *summa av produkter*.

Ritat i grindlogikschema får ett uttryck i denna form typ-utseendet



Det vill säga insignalerna går först genom ett lager av OCH-grindar och sedan en gemensam ELLER-grind. En alternativ benämning är ”OCH-ELLER-form”.

3.2. Minimering

Normalformen ovan är generöst utförd och inte minimal. Vi kan få exakt samma utsignal med färre grindar genom en process vi kallar *minimering*. Principen för minimering är i sig rätt enkel och naturlig:

1. Gruppera ihop termer som skiljer sig i **en** variabel.
2. Bryt ut den gemensamma variabeln.
3. Eliminiera den variabel som skiljer sig åt

Med metoden kan vi få fram en annan SP-form, som inte är normalformen utan enklare:

$$
\begin{aligned}
lo3 &= \overbrace{\overline{x_3}\,\overline{x_2}\,\overline{x_1}\,\overline{x_0}}^{*} + \overline{x_3}\,\overline{x_2}\,\overline{x_1}x_0 + \overline{x_3}\,\overline{x_2}x_1\overline{x_0} = \\
&= \overline{x_3}\,\overline{x_2}\,\overline{x_1}\underbrace{(\overline{x_0} + x_0)}_{=1} + \overline{x_3}\,\overline{x_2}x_1\overline{x_0} = \\
&= \overline{x_3}\,\overline{x_2}\,\overline{x_1} + \overline{x_3}\,\overline{x_2}x_1\overline{x_0} + \overbrace{\overline{x_3}\,\overline{x_2}\,\overline{x_1}\,\overline{x_0}}^{*} = \\
&= \overline{x_3}\,\overline{x_2}\,\overline{x_1} + \overline{x_3}\,\overline{x_2}\,\overline{x_0}\underbrace{(\overline{x_1} + x_1)}_{=1} = \\
&= \overline{x_3}\,\overline{x_2}\,\overline{x_1} + \overline{x_3}\,\overline{x_2}\,\overline{x_0}.
\end{aligned}
$$

Termen $\overline{x_3}\,\overline{x_2}\,\overline{x_1}\,\overline{x_0}$ har markerats med $*$ och lagts till igen för att kunna använda x_1 som gemensam variabel.[1]

Ytterligare gemensamma variabler att bryta ut finns inte, uttrycket är minimalt.

3.2.1. Grafisk metod för minimering, ”karnaughminimering”

Minimeringen ovan kan enklare utföras rent grafiskt med ett så kallat *karnaughdiagram*, KD: Genom att skriva uttryckets sanningstabell på en form där varje kolumn och rad skiljer sig med enbart *en* variabel kan man sedan lätt (jo!) se lämpliga sammanslagningar av typen $\overline{x_0} + x_0$.

Exempel 15

Använd karnaughdiagram för att minimera funktionen i sanningstabellen för $u = u(x_2x_1x_0)$.

$x_2x_1x_0$	u
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

dvs $u(x_2x_1x_0) = \overline{x_2}\,\overline{x_1}x_0 + x_2\overline{x_1}x_0 + x_2x_1x_0$

[1] Då $x = x + x$ kan vi lägga till ytterligare x utan att uttryckets sanningstabell förändras.

1. Konstruera ett karnaugdiagram med de tre invariablerna $x_2x_1x_0$. Fyll därefter i utsignalen u. Här är det lättast är att markera ettorna först, de är ju bara tre stycken[2] och sedan fylla ut med nollor i övriga rutor. Det som inte är ettor är ju nollor, eller hur?

x_2 \ u \ x_1x_0	00	01	11	10
0		1		
1		1	1	

$\rightarrow$

x_2 \ u \ x_1x_0	00	01	11	10
0	Ø	1	Ø	Ø
1	Ø	1	1	Ø

2. Själva minimeringen handlar sedan om att ringa in grupper av närliggande ettor. Som diagrammet är upplagt skiljer dessa sig åt i bara en variabel.

x_2 \ u \ x_1x_0	00	01	11	10
0	Ø	1	Ø	Ø
1	Ø	1	1	Ø

Den heldragna ringen motsvarar utsignalen "1" då $\overline{x_1}x_0$, det spelar ingen roll vad x_2 är! Den streckade ringen motsvarar på liknande sätt utsignalen "1" för x_2x_0, det spelar ingen roll vad x_1 är i detta fall. Vi har alltså uttryck för alla ettor och det är allt som behövs.

Totala utsignalen är summan av dessa dvs uttrycket

$$u = \overline{x_1}x_0 + x_2x_0.$$

Ytterligare minimering omöjlig. Vi har ett minimalt uttryck på SP-form (dock inte på SP-normalform).

■

Övning: Genomför algebraisk minimering av ovanstående exempel. Se till att förstå varför metoden fungerar.

[2] Är det färre nollor fyller man i dem först osv.

Exempel 16

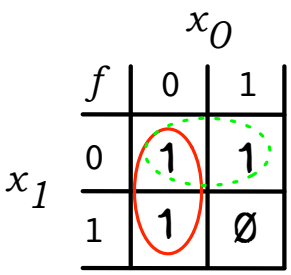
Minimera uttrycket

$$f = \overline{x_1}\,\overline{x_0} + x_1\overline{x_0} + \overline{x_1}x_0.$$

Genom successiva tillämningar av metoden kan vi algebraiskt skriva

$$\begin{aligned} f &= \overline{x_1}\,\overline{x_0} + x_1\overline{x_0} + \overline{x_1}x_0 = \\ &= \overline{x_1}(\overline{x_0} + x_0) + \overline{x_0}(\overline{x_1} + x_1) = \\ &= \overline{x_1} + \overline{x_0}. \end{aligned}$$

Med KD kan vi direkt få fram samma sak:



■

3.2.2. Hur ringar man?

Efter en stunds funderande (och möjligen lite övning) kan man se att man måste forma ringar inneslutande 1, 2, 4, 8 rutor osv. Det finns ingen möjlighet att ringa till exempel en tre-grupp. Inte heller är det möjligt att ringa diagonalt.

Vid ringningen tänk: *vilken kolumn/rad är gemensam?*, *vilken kolumn/rad spelar ingen roll?*. [3]

[3] Man kan jämföra med koordinater eller geografiska positioner: Vad är gemensamt i höjd respektive sida?

Exempel 17 KD med fyra variabler

KD kan utan problem användas vid minimering upp till fyra variabler. Diagrammet

x_3x_2 \ x_1x_0	00	01	11	10
00	1	1	Ø	1
01	Ø	Ø	1	1
11	Ø	Ø	1	1
10	Ø	1	Ø	Ø

kan ringas på vanligt sätt under iakttagande att man får så få ringar som möjligt. Det kan finnas olika möjliga ringningar och den mest minimala anses vara den med minst antal ingångar på den resulterande schemarealiseringen.

Så få och så små termer som möjligt eftersträvas och man får prova sig fram. En möjlig kandidat är

x_3x_2 \ x_1x_0	00	01	11	10
00	1	1	Ø	1
01	Ø	Ø	1	1
11	Ø	Ø	1	1
10	Ø	1	Ø	Ø

Vilket motsvarar

$$f = x_2x_1 + \overline{x_3}\,\overline{x_2}\,\overline{x_0} + \overline{x_2}\,\overline{x_1}x_0.$$

Antalet ingångar blir i detta fall $2 + 3 + 3 = 8$ i OCH-nivån och 3 i ELLER-nivån alltså sammanlagt 11 ingångar.

Övning: Prova själv med andra ringningar och undersök om det finns någon med färre antal ingångar.

3.2.3. Sammanfattning av SP-form

En SP-form är en *summa av produkter* dvs uttryck av typen

$$f(x, y, z) = x\overline{y}\,\overline{z} + xyz + \ldots$$

Ett skrivsätt är att skriva ut de insignalkombinationer som ger ettor, till exempel:

$$f(x, y, z) = \sum(4, 7)$$

Där $f = 1$ vid insignalerna $xyz = 4, 7 = \{100, 111\}$. Man tjänar på att lära sig numrera de olika rutorna i KD. I fallet med 4 insignaler i ordning $x_3x_2x_1x_0$ numreras KD sålunda:

x_3x_2 \ x_1x_0 (f)	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Minimering sker genom att ringa de insignalkombinationer som ger ettor ut!

3.2.4. *Don't cares*, DC

Om vissa insignalkombinationer inte är definierade och alltså inte behöver ge någon speciell utsignal benämnes dessa *don't cares*, DC, och markeras dessa rutor med streck "—" i KD.[4]

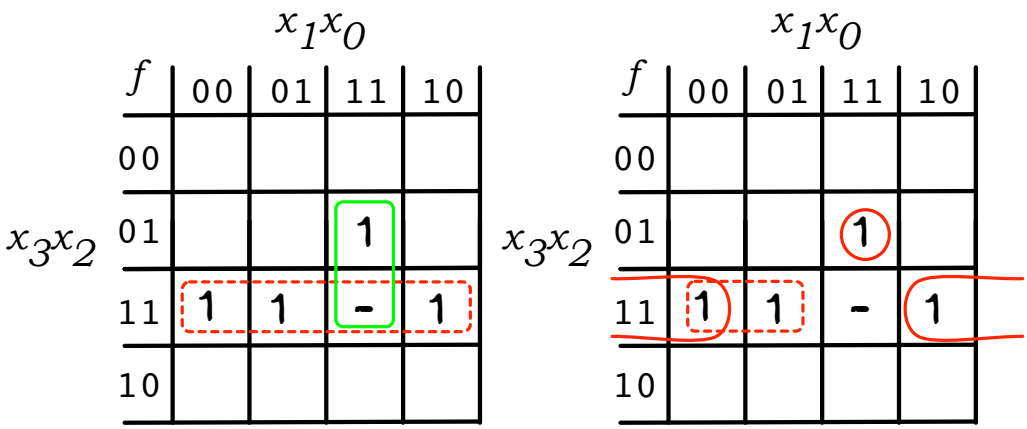
Vid minimeringen *kan* "—" sedan ringas som nollor eller ettor allt efter vad som behövs för så stora ringningar som möjligt.

Det finns inget krav att dessa DC *skall* ringas. DC ger extra frihet att ringa på ett minimalt sätt i och med att man inte behöver vara övertydlig i sina ringningar utan kan tillåta ett visst "slack".

[4] I $\sum$-uttrycket markeras detta efter parentesen med "+d(1,3, . . .)" osv för respektive utsignaler.

Exempel 18 KD med Don't cares

Diagrammen nedan[5] visar olika möjliga ringningar för samma utfunktion, f.



DC *kan* användas, som till vänster, men behöver inte ringas, som till höger. Större ringningar är vanligtvis bättre än mindre varför den vänstra är minimal. Beräkna själv vilket antal ingångar som hör till respektive KD.

[5] 0:or utelämnade för tydlighets skull.

3.3. Binär addition. Tillämpning av kombinatorik.

Vid addition med binära tal kan fyra olika fall uppkomma

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} \underline{1}\,1 \\ +\ 1 \\ \hline 1\,0 \end{array}$$

Lägg här märke till att "+" betyder den vanliga additionen och inte det binära ELLER.

Exempel 19

Med dessa fyra additioner kan vi beräkna summan av två binära tal $\mathcal{X} = 101_2$ och $\mathcal{X} = 101_2$:

$$\begin{array}{r} \underline{1}\ \ \underline{1}\ \ \ \\ 1\ \ 0\ \ 1 \\ +\ \ 1\ \ 0\ \ 1 \\ \hline 1\ \ 0\ \ 1\ \ 0 \end{array}$$

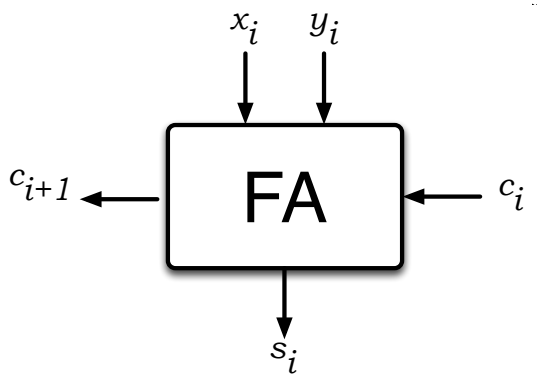
Notera ovan hur en *minnessiffra* (pilarna) uppstått vid addition av två ettor. Lägg också märke till hur resultatet är fyra bitar medan termerna bara har tre bitar vardera, den sista (inringade) biten är s.k. *utgående minnessiffra, carry*: Med tre bitar kan talen $0-7$ representeras medan resultatet är $5+5 = 10_{10}$ och överstiger detta talområde. För att representera talet tio behövs fyra bitar som i resultatet ovan.

Med vår kännedom om kombinatoriska nät skall vi nu ge oss i kast med att konstruera en adderare som kan utföra beräkningen ovan. Lösningen innehåller flera tänkvärda knep och procedurer som är centrala i digitaltekniken.

Exempel 20 Heladderare

> Konstruera ett kombinatoriskt nät som utför addition av två tre-bitars binära tal.

Metoden blir som vanligt att utföra uppgiften i mindre steg: Med en fulladderare, *Full Adder*, menas en enhet som kan addera två bitar x_i och y_i samt en inkommande carry c_i från höger. Som resultat lämnas en summabit s_i och en utgående carry c_{i+1}. Det vill säga vi söker innehållet i denna låda:



Det här känns egendomligt bekant eller hur? En låda med tre insignaler och två utsignaler! Det är ju ett K-nät! Och K-nät kan vi sedan tidigare. Vi är på hemmaplan!

En sanningstabell för resultatet kan vi formulera genom att bara räkna på och betrakta $\{c_{i+1}\, s_i\}$ som ett två bits binärt tal (där s_i är minst signifikant bit):

$x_i\, y_i\, c_i$	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

För att hitta ett logiskt uttryck mellan de tre insignalerna och *vardera* resultatkolumn använder vi KD:

c_{i+1} \ c_i	$x_i\, y_i$ 00	01	11	10
0	Ø	Ø	1	Ø
1	Ø	1	1	1

s_i \ c_i	$x_i\, y_i$ 00	01	11	10
0	Ø	1	Ø	1
1	1	Ø	1	Ø

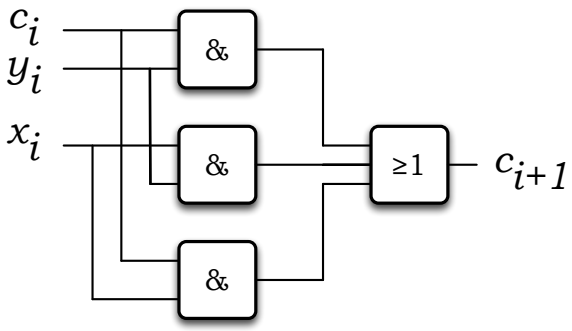
Vilket ger ekvationerna (gör ringningarna själv)

$$
\begin{aligned}
c_{i+1} &= c_i y_i + x_i y_y + c_i x_i \\
s_i &= c_i \overline{x_i}\,\overline{y_i} + \overline{c_i}\,\overline{x_i} y_i + c_i x_i y_i + \overline{c_i} x_i \overline{y_i}
\end{aligned}
$$

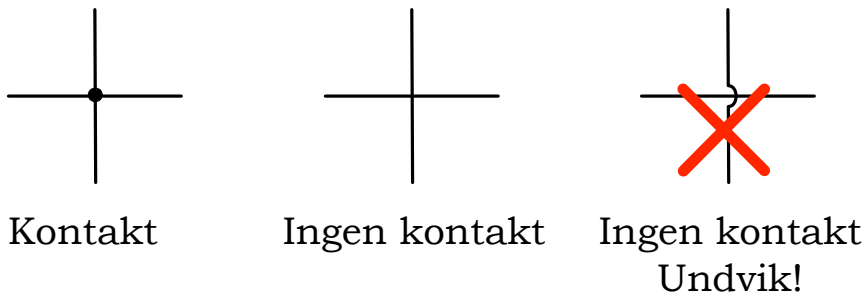
Den första ekvationen ser rimlig ut men med den andra hade vi maximal otur; enbart isolerade ettor.

Övning: Fundera på hur XOR-funktionen, $\oplus$, kan underlätta i detta fall.

Med momentet *realisering* menas att översätta ekvationerna till grindlogik. Här för c_{i+1}:



Vid schemaritning korsar ledningar varann utan kontakt. För att markera kontakt används en liten punkt i korspunkten:



Övning: Rita själv realiseringen av s_i.

3.3.1. Realisering med enbart NAND-grindar

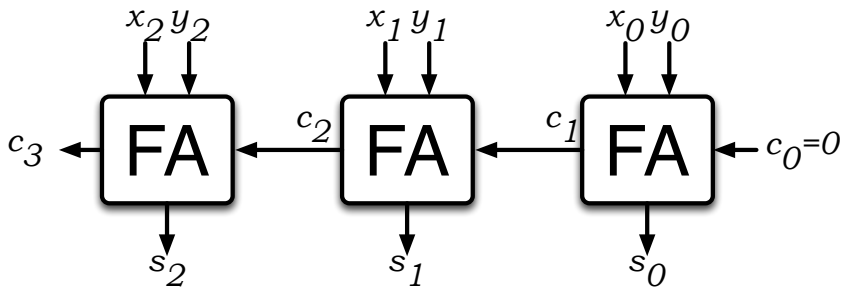
Ekvationen $c_{i+1} = c_i y_i + x_i y_y + c_i x_i$ realiserades ovan med OCH- och ELLER-grindar. Med enbart NAND-grindar till hands kan en lösning realiseras efter manipulation av ekvationen:

$$c_{i+1} = \overline{\overline{c_{i+1}}} = \overline{\overline{c_i y_i + x_i y_y + c_i x_i}} = \overline{\overline{c_i y_i} \cdot \overline{x_i y_y} \cdot \overline{c_i x_i}}$$

Knepet är alltså att ”dubbelinvertera” och sedan använda de Morgan på resultatet.

3.3.2. Realisering av trebitars adderare

Med en en-bits fulladderare realiserad enligt ovan kan vi bygga på den framgången för att realisera en tre-bitars adderare bit för bit. Notera hur ingående carry c_0 sätts till noll och c_3 är utgående carry. Summan är $s_2 s_1 s_0$.



Övning: Notera hur lösningen är strukturerad och bygger på redan tidigare löst delproblem. Blir det svårare/lättare att realisera ett enda K-nät för trebitars adderaren?

3.3.3. PS-normalform

Läs detta avsnitt först vid en en andra genomgång av materialet. Det är lätt att blanda ihop begreppen om man försöker ta in dem vid samma tidpunkt.

Förutom den vanliga och redan nämnda Summa-av-produkt-formen, SP-formen, förekommer en dual även till den: *Produkt-av-summa-*, *PS-formen*, med typutseendet:

$$\overline{f} = (x + y + z)(x + y + \overline{z})(\ldots)$$

och erhålls genom att man i KD ringar nollor istället för ettor. Det man då får fram är naturligtvis inte $f(x, y, z)$ utan $\overline{f}(x, y, z)$. Med de Morgans formler kan man sedan beräkna $f(x, y, z)$.

På liknande sätt som SP-formen har ett skrivsätt med $\sum$ har PS-formen ett skrivsätt med Π:

$$f(x, y, z) = \Pi(0, 1, 2, 3, 5, 6)$$

Och det är nu de insignalkombinationer som ger *nollor* som anges[6] medan själva PS-formen tas fram med inversen enligt ovan.

Exempel 21

Ta fram SP- och PS-formerna för $f(x, y, z, t)$ i karnaughdiagrammet nedan:

$x\,y$ \ $z\,t$	00	01	11	10
00	1	1	Ø	Ø
01	Ø	Ø	1	Ø
11	Ø	Ø	1	Ø
10	1	1	1	Ø

För SP-formen ringar vi 1-or och för PS-formen tar vi fram inversen $\overline{f}$ genom att ringa 0-or:

$x\,y$ \ $z\,t$	00	01	11	10
00	1	1	Ø	Ø
01	Ø	Ø	1	Ø
11	Ø	Ø	1	Ø
10	1	1	1	Ø

$x\,y$ \ $z\,t$	00	01	11	10
00	1	1	Ø	Ø
01	Ø	Ø	1	Ø
11	Ø	Ø	1	Ø
10	1	1	1	Ø

[6] Alltså $f(x, y, z) = \Pi(0, 1, 2, 3, 5, 6) = \sum(4, 7)$.

Uttrycken kan vi läsa av som

$$
\begin{aligned}
f &= \overline{y}\,\overline{z} + x\overline{y}t + yzt \\
\overline{f} &= y\overline{z} + \overline{x}\,\overline{y}z + z\overline{t}
\end{aligned}
$$

PS-formen erhålls efter de Morgan i två steg på uttrycket för $\overline{f}$ — men först en inledande invertering eftersom vi önskar f och inte $\overline{f}$ som slutresultat:

$$
\begin{aligned}
f = \overline{(\overline{f})} &= \overline{y\overline{z} + \overline{x}\,\overline{y}z + z\overline{t}} = \\
&= \overline{y\overline{z}} \cdot \overline{\overline{x}\,\overline{y}z} \cdot \overline{z\overline{t}} = \\
&= (\overline{y} + z)(x + y + \overline{z})(\overline{z} + t).
\end{aligned}
$$

Det följer direkt ur uttrycken att realisering av SP-formen motsvaras av ett OCH/ELLER-nät och realisering av PS-formen av ett ELLER/OCH-nät. Den ena formen kan vara mer minimal än den andra. Ibland måste man prova sig fram.

Vi såg tidigare att en realisering med NAND-grindar var enkel utgående från SP-formen. Med liknande resonemang kan en PS-form enkelt utföras med NOR-grindar:

$$
\begin{aligned}
f &= (\overline{y} + z)(x + y + \overline{z})(\overline{z} + t) = \\
&= \overline{\overline{(\overline{y} + z)(x + y + \overline{z})(\overline{z} + t)}} = \\
&= \overline{\overline{(\overline{y} + z)} + \overline{(x + y + \overline{z})} + \overline{(\overline{z} + t)}}.
\end{aligned}
$$

Där uttrycket nu i sin helhet består av NOR-grindar.

Övning: Den uppenbara övningen är nu att rita tillhörande schema med *enbart* NOR-grindar.

■

Mer om SP- och PS-form Vi vet nu att SP-formen fås genom att *identifiera alla de insignalkombinationer som ger logisk 1:a och OR:a ihop dessa.* För två insignaler, $x_1x_0 = \{00, 01, 10, 11\}$, motsvarande siffrorna $0 \ldots 3$, blir det största möjliga uttrycket:

$$\overline{x_1}\,\overline{x_0} + \overline{x_1}x_0 + x_1\overline{x_0} + x_1x_0$$

Där respektive term blir 1 för en viss insignal $0 \ldots 3$ och man tar med enbart de termer som man vill ska ge utsignalen 1. Det räcker ju att en av termerna är 1 så är hela uttrycket 1.

Funktionen f på SP-form

$$f = \overline{x_1}\,\overline{x_0} + \overline{x_1}x_0$$

blir till exempel 1 för $x_1x_0 = \{00, 01\}$.

Ett alternativ till SP-formen är dess *dual*, PS-formen, som fås genom att *identifiera alla de insignalkombinationer som ger logisk 0:a och AND:a ihop dessa.*

PS-uttrycket

$$(x_1 + x_0)(x_1 + \overline{x_0})(\overline{x_1} + x_0)(\overline{x_1} + \overline{x_0})$$

består av faktorer där det räcker att *en enda blir 0* för att hela uttrycket skall bli 0.[7]

För att få samma resultat som SP-funktionen ovan, alltså 1 för $x_1x_0 = \{00, 01\}$, måste vi i PS-formen bevara de faktorer som ger 0, det vill säga $x_1x_0 = \{10, 11\}$ och ta fram inversen till f:

$$\overline{f} = (\overline{x_1} + x_0)(\overline{x_1} + \overline{x_0})$$

Med lite boolesk algebra och de Morgans lagar kan man se att de två uttrycken är identiska. Gör det!

[7] Jämfört med SP-formen där vi letar 1:or och det räcker med *en enda 1:a* för att hela uttrycket skall bli 1.

Del II.

Sekvensiella nät

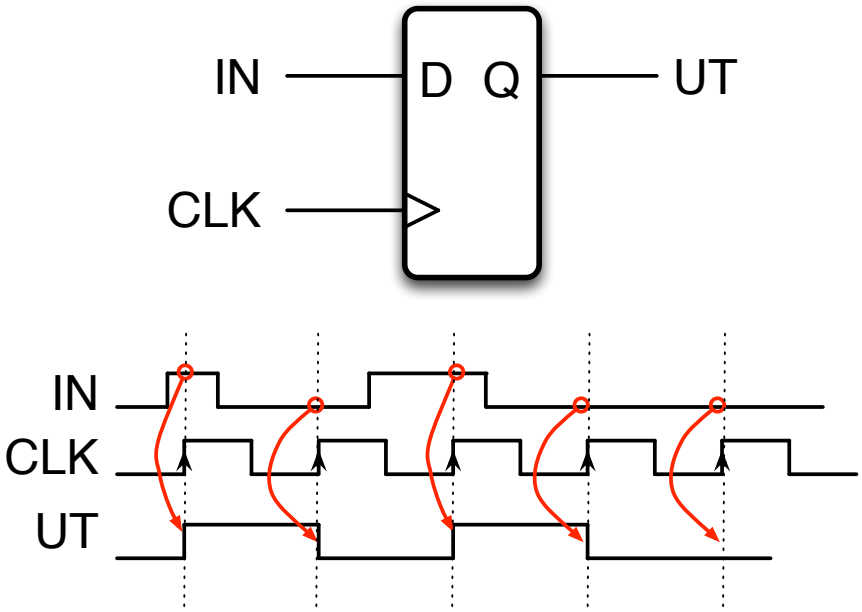
4. Sekvensnät

Vi har sett att grindar och kombinatoriska nät ger momentana resultat: Lägg på insignaler och utsignalen ändras omedelbart. I praktiska kretsar rasslar det möjligen till i några nanosekunder innan en stabil utsignal erhållits men poängen är att det är en form av tabelluppslagning. Tabellen ifråga är sanningstabellen och det vi ägnat all tid åt hittills är att realisera minimala nät som imiterar sanningstabellen.

4.1. Vippan

Genom att införa ett *minneselement* kan vi konstruera en helt annan typ av beteende.

En vippa är ett sådant minneselement där en insignal inte påverkar utsignalen förrän på *kommando* från en *klocksignal*:



I figuren ses överst en D-vippas (av engelskans *delay*) schemasymbol. Därunder visas funktionen i ett signalschema. Klocksignalen, CLK, styr när insignalen skall kännas av och slussas till utsignalen.

D-vippan styrs av klocksignalens *aktiva flank*. I figuren är det när signalen går från noll till ett, en s.k. *positiv* flank. Den aktiva flanken kan också vara på fallande, *negativ* flank. Den aktiva flanken är här, för tydlighets skull, markerad med pilar.

Det värde som ligger och väntar på vippans D-ingång flyttas till vippans Q-utgång just i det ögonblick den aktiva flanken sker. Mellan de aktiva flankerna hålls Q-signalen konstant även om D-signalen ändras.

Vi ser att D-vippan håller kvar det senaste värdet åtminstone tills en ny aktiv flank kommer. Vippan har på detta sätt två möjliga utsignaler ”1” eller ”0”. Vi kallar detta värde för vippans *tillstånd*. Tillståndet betecknas med q. Nästa ”väntande” tillstånd betecknas q^+. Vippan fungerar som ett 1-bits minne.

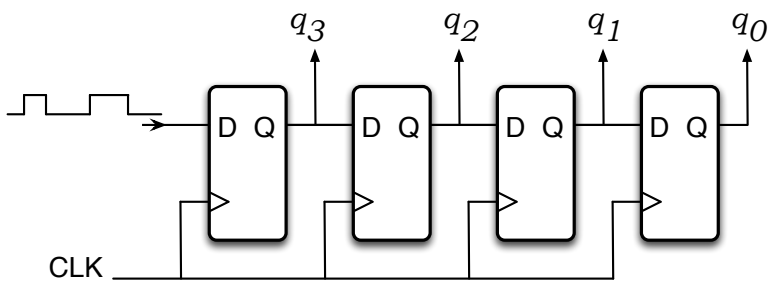
4.1.1. Vippkonstruktioner

Flera D-vippor kan kopplas ihop på några olika sätt. Här visar vi exempel på hur de kan kopplas ihop till två typer av *register*.

Exempel 22. Seriellt 4-bits skiftregister

> Ett seriellt skiftregister kan användas som en *serie-parallellomvandlare*. Det seriella datat skickas in i takt med klockan, bit för bit med minst signifikant bit först. Efter fyra klockor kan det parallella resultatet läsas ut som vippornas tillstånd $Q = \{q_3 q_2 q_1 q_0\}$.

Det är brukligt att använda versaler för tillstånd, Q, och gemener med index för de enskilda bitarna, q_i.



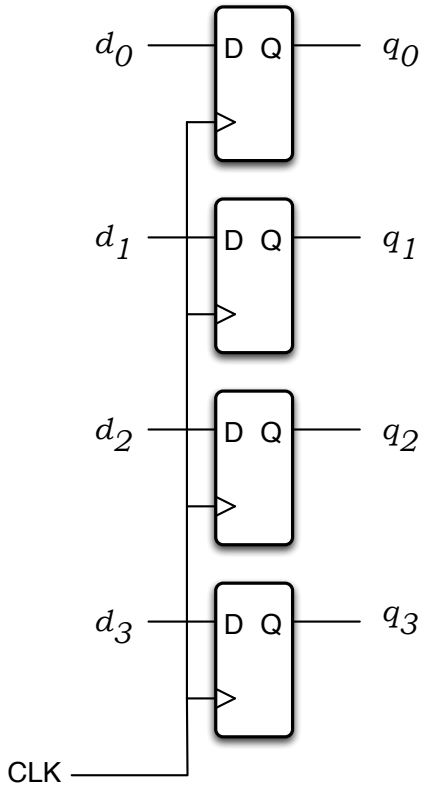
Man kan notera speciellt att insignalen känns av exakt vid klockans aktiva flank och att avläsningen av nästa tillstånd inte får utföras förrän den aktiva flanken sker.

■

Övning: Konstruera själv omvändningen, dvs en parallell-till-serieomvandlare.
Övning: Teckna ekvationerna ”$q_i^+ = \ldots$” för skiftregistret.

Exempel 23. 4-bits regster

Med ett 4-bits register kan man lagra en hel BCD-siffra.



Indatat läggs här som värdet $D = \{d_3 d_2 d_1 d_0\}$ och utvärdet är $Q = \{q_3 q_2 q_1 q_0\}$.

Den ekvation som motsvarar nästa tillstånd kan med vår tidigare införda beteckning skrivas komponentvis $q_i^+ = d_i$ eller sammantaget

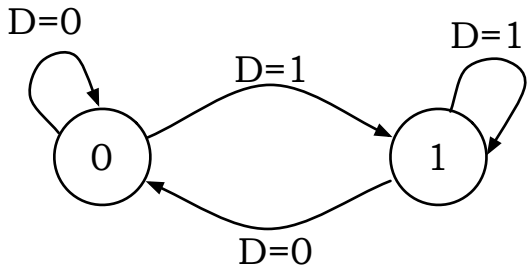
$$Q^+ = D$$

4.2. Tillståndsgraf

Med minneselement kan vi alltså konstruera helt nya beteenden som inte var möjliga med rent kombinatoriska kretsar. För detta duger inte sanningstabellen längre, då den inte kunde innehålla minne. Nu måste vi beskriva funktionen i en *tillståndsgraf*[1].

[1] En del kallar det för *tillståndsdiagram* men rent matematiskt är det en *graf* varför vi behåller detta begrepp.

En enkel tillståndsgraf är den som beskriver D-vippans funktion:



D-vippans två tillstånd, ”0” eller ”1”, är de två ringarna. Beroende på insignalen till vippan, D, kan vippan antingen byta eller hålla sitt tillstånd: Om vippan är i tillstånd ”0” kommer den hålla sitt tillstånd så länge insignalen $D = 0$. Om däremot $D = 1$ kommer nästa tillstånd vara ”1”. Övergången till det nya tillståndet sker som vi vet exakt på klocksignalens aktiva flank. På liknande sätt kan resten av tilltåndsgrafen förklaras.

I detta fall är tydligen *utsignalen lika med tillståndet*.

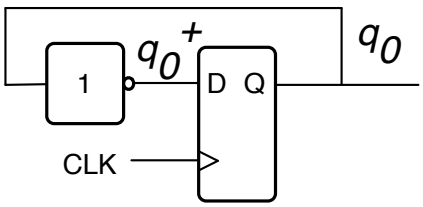
Övning: Försäkra dig om att du känner igen D-vippans funktion i tillståndsgrafen.

4.3. Sekvensnät

Tillståndsgrafens funktion kan realiseras med vippor och kombinatoriska nät. Vi börjar med analys av ett exempel.

Exempel 24. Frekvensdelare

Förklara funktionen hos följande konstruktion. Rita dess tillståndsgraf.



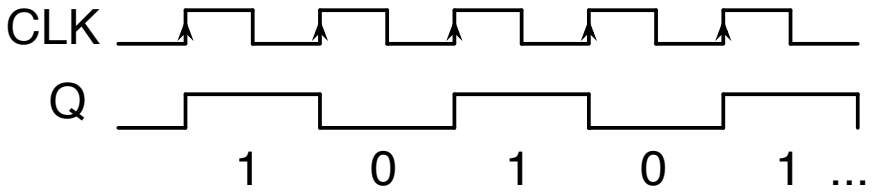
Konstruktionen beskrivs av dess *tillståndsekvation*, som man får genom att studera schemat, i detta fall:

$$q_0^+ = \overline{q_0}.$$

Konstruktionens hela beteende ligger i denna ekvation: Antag till exempel att $q_0 = 1$, då blir nästa tillstånd $q_0^+ = \overline{1} = 0$. Men om $q_0 = 0$ då blir nästa tillstånd $q_0^+ = \overline{0} = 1$. Utsekvensen, dvs q_0, är en omväxlande serie ettor och nollor:

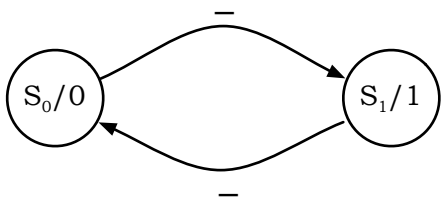
$$1 \to 0 \to 1 \to 0 \to 1 \to 0 \to 1 \to 0 \to 1 \to 0 \to \ldots$$

I ett signalschema kan vi visualisera detta som



Vippan antar vid varje aktiv flank sitt motsatta värde. Och det var ju precis det tillståndsekvationen visade!

En tillståndsgraf för detta repetetiva beteende blir



där utsignalen markerats höra till respektive tillstånd S_x med notationen $S_0/0$ och $S_1/1$. Tillståndet kan på detta sätt symboliseras med S_x och frikopplas från den egentliga utsignalen.

Notera också frånvaron av insignal till konstruktionen, denna konstruktion är *autonom*, dvs den bara håller på och håller på och hoppar mellan de två tillstånden. Tillståndsövergång oberoende av insignal anges med ett streck ”–” på bågarna.

Med sekvensnät sålunda presenterade kommer här ett centralt genomarbetat exempel:

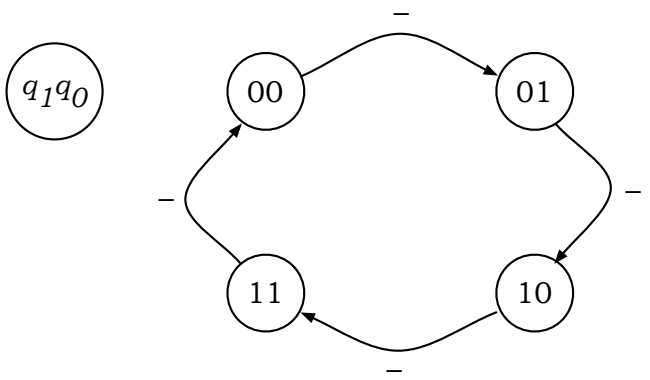
Exempel 25. 2-bits räknare

Konstruera en 2-bits autonom[2] binärräknare, dvs en räknare som genomlöper tillstånden {00, 01, 10, 11, 00, . . . }.

[2]Självgående, dvs utan insignal.

Steg 1: Tillståndsgraf

För att representera fyra tillstånd behövs två bitar (ty $4 = 2^2$). En tillståndsgraf för denna autonoma räknare med tillstånden $q_1 q_0$ blir:



Steg 2: Tillståndstabell

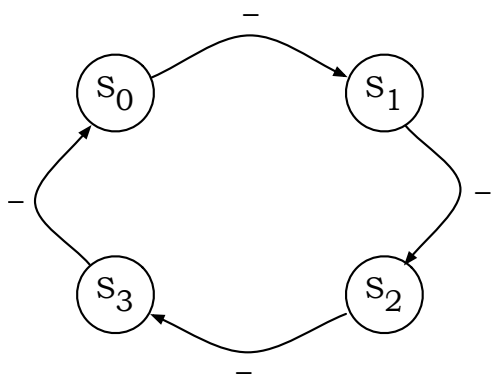
För den vidare konstruktionen beskrivs sedan tillståndsgrafen ovan i en *tillståndstabell*:

$q_1 q_0$	$q_1^+ q_0^+$
0 0	0 1
0 1	1 0
1 0	1 1
1 1	0 0

där man, rad för rad, kan följa nuvarande tillstånd $q_1 q_0$ till nästa tillstånd $q_1^+ q_0^+$. Då det finns fyra tillstånd blir det fyra rader i tillståndstabellen.

Steg 3: Tillståndskodning

I detta exempel är det naturligt att tillstånden också är utsignal. Om det inte skulle vara fallet kan man symboliskt benämna de olika tillstånden $S_0, \ldots, S_3$ (eller något annat) och först i detta steg bestämma vad de olika tillstånden skall kodas som. I detta fall till exempel:



Här överensstämmer dock tillstånden med kodningen (det var ju viktigt för utsignalen)

Till-stånd	Kod-ning
S_0	0 0
S_1	0 1
S_2	1 0
S_3	1 1

Steg 4: Tillståndsekvationer

Med tillståndsekvationer avses de ekvationer som motsvarar tillståndsgrafen. Ekvationerna binder samman nuvarande tillstånd med nästa tillstånd

$$Q^+ = f(Q)$$

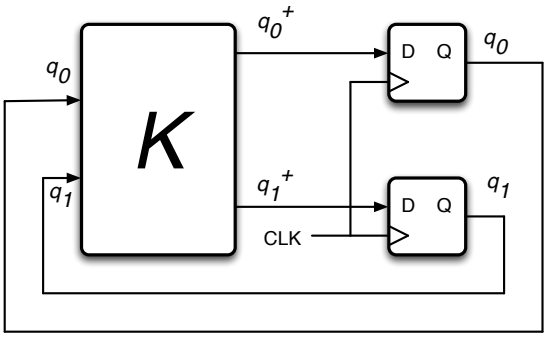
dvs

$$q_1^+ q_0^+ = f(q_1 q_0)$$

eller tecknat bitvis

$$q_1^+ = f_1(q_1 q_0)$$
$$q_0^+ = f_0(q_1 q_0)$$

Konstruktionsidén är nu att låta ett kombinatoriskt nät K utföra funktionen f som nedan:



Återstår att konstruera K. Direkt ur tillståndstabellen kan dessa erhållas som

$$q_1^+ = \overline{q_1} q_0 + q_1 \overline{q_0}$$
$$q_0^+ = \overline{q_1}\,\overline{q_0} + q_1 \overline{q_0}$$

eller minimerade efter lite algebraiska manipulationer (eller med karnaugh-diagrammets fulla kraft) som:

$$q_1^+ = \overline{q_1}q_0 + q_1\overline{q_0}$$
$$q_0^+ = \overline{q_0}\overbrace{(\overline{q_1} + q_1)}^{=1} = \overline{q_0}$$

Med dessa ekvationer realiserade i K kommer korrekt nästa tillstånd alltid att beräknas utgående från nuvarande tillstånd.

Notera hur q_0 faktiskt växlar $\ldots 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \ldots$ på grund av ekvationen $q_0^+ = \overline{q_0}$.

■

4.4. Mealy- och Moorenät

Det finns två vanliga huvudgrupper av sekvensnät: Nät av *Mealy-* eller *Moore-*typ.

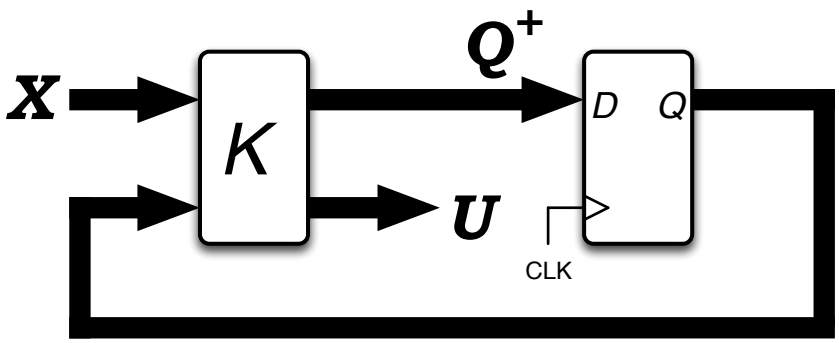
Lägg märke till hur *tillståndsövergångarna* bestäms på *exakt* samma sätt oavsett sekvensnätstyp nedan. *Utsignalen* tillverkas dock på olika sätt.

4.4.1. Mealy-typ

Ett sekvensnät av Mealy-typ karakteriseras av ekvationerna

$$\begin{cases} Q^+ & = f(Q, X) \\ U & = g(Q, X) \end{cases}$$

Ekvationerna skall tolkas som att nästa tillstånd Q^+är en funktion av *både* nuvarande tillstånd Q *och* insignalen X. På liknande sätt är utsignalen också en funktion av *både* nuvarande tillstånd *och* insignalen:



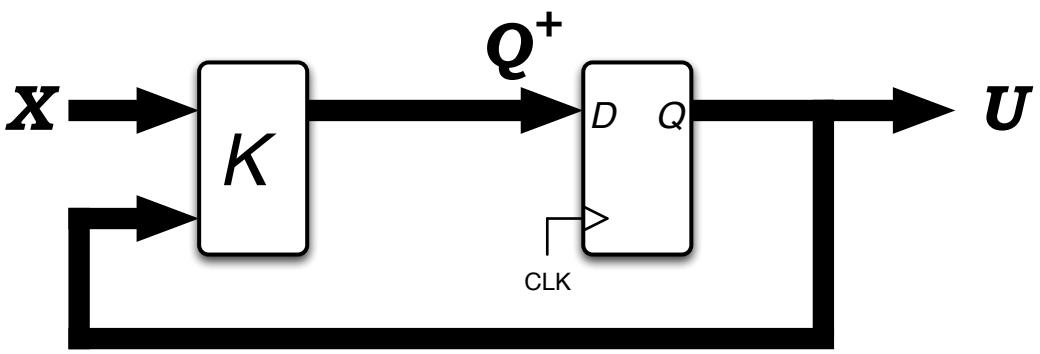
Speciellt ser man i figuren att utsignalen kan ändras inte bara vid klockflank utan också *direkt* när insignalen $\mathcal{X}$ ändras. Om sekvensnätet står still i samma tillstånd Q kan alltså utsignalen ändå ändras då U beror av X [ekvationen $U = g(Q, X)$] . En signal som på detta sätt kan ändra värde utan klockflank kallas *asynkron*.

4.4.2. Moore-typ

Ett sekvensnät av Moore-typ karakteriseras av ekvationerna

$$\begin{cases} Q^+ & = f(Q, X) \\ U & = g(Q) \end{cases}$$

Ekvationerna skall tolkas som att nästa tillstånd även här är en funktion av *både* nuvarande tillstånd och insignalen, precis som i mealyfallet. I Moorenätet är däremot utsignalen enbart en funktion av nuvarande tillstånd. Utsignalen är *synkron* med systemets klockflanker.



I figuren har antagits att utsignalen är identiskt lika med tillståndens kodning, $U = Q$. Så behöver inte vara fallet, den faktiska utsignalen kan också utgöra en kombinatorisk funktion av tillståndet enligt

$$U = g(Q).$$

Utsignalen är fortfarande synkron med klockflankerna ehuru möjligen något fördröjd[3].

Not: För båda nättyperna kan det vara enklare att först hantera tillståndsövergångarna för sig och *därefter* utsignalen.

Övning: Övertyga dig att den tidigare konstruerade 2-bits binärräknaren är av Moore-typ? Varför?

[3] Den kombinatoriska fördröjningen antas för övrigt alltid vara så liten att nästa tillstånd hinner bildas.

4.5. Synkronism

De sekvensnät som hittills behandlats tillhör en större klass av *synkrona system*. Med synkrona system avses sådana konstruktioner där alla tillståndsövergångar sker i takt med en gemensam global klocka, *systemklockan*. Synkronism är en viktig egenskap som förenklar design av digitala system.

Generellt gäller i synkrona system att alla tillståndsövergångar skall ske i takt med den aktiva klockflanken.

För att detta skall vara möjligt ställs krav på tiden mellan två på varandra följande klockflanker, det vill säga systemets klockfrekvens.

Klockfrekvensen får inte drivas upp så högt att de ingående kombinatoriska näten inte hinner beräkna klart nästa tillstånd. Ty om nästa tillstånd inte skulle vara helt färdigberäknat när den aktiva flanken kommer spårar sekvensnätet ur!

Fenomenet illustreras i figuren nedan:

I figuren ses hur nästa tillstånd vid de första klockflankerna hinner bli färdigberäknad innan nästa aktiva flank kommer. Direkt vid den den aktiva flanken börjar sekvensnätets kombinatoriska delar beräkna nästa tillstånd. Under denna tid, i figuren angivet med "??????", är inte nästa tillstånd färdigberäknat. När nästa tillstånd är beräknat och stabilt, "Klar" i figuren, får klockflanken komma. Detta säkerställer sekvensnätets korrekta funktion.

Vid figurens sista aktiva klockflank kan man dock se att nästa tillstånd ännu inte är färdigberäknat varför sekvensnätet kan anta vilket nytt tillstånd som helst och har ur alla praktiska synvinklar kraschat!

Setup-tid Ur datablad kan man läsa en vippas *setup*-tid. Med setuptid menas den minsta tid en signal måste vara stabil på vippans ingång innan klockan kommer. Den är i storleksordningen nanosekunder och anger alltså en minsta tid för "Klar" i figuren.

Den maximala klockfrekvensen för ett sekvensnät styrs således till stor del av den ingående kombinatorikens tidsfördröjningar. Lägg märke till att dessa tidsfördröjningar kan vara olika beroende på vilka omslag som sker i kombinatoriken. Man pratar om *kritisk väg, critical path,* för den signalväg som tar absolut längst tid, den dimensionerar systemets maximala klockfrekvens f_{max}.

4.6. Sekvensnät med insignal

Grundtanken bakom sekvensnät är att tillståndet, Q, beskriver var man *är* och nästa tillstånd, Q^+, vart man *skall gå*:

- Vi lagrar nuvarande tillstånd i sekvensnätets minne (hittills D-vipporna).
- Vi använder dessutom ett kombinatoriskt nät för att tolka nuvarande tillstånd och peka ut nästa tillstånd. Det måste vara fullt möjligt att använda andra, utifrån kommande signaler, för att ytterligare styra nästa tillstånd. Detta visas i exempel 26 nedan.

Exempel 26. Sekvensnät med insignal

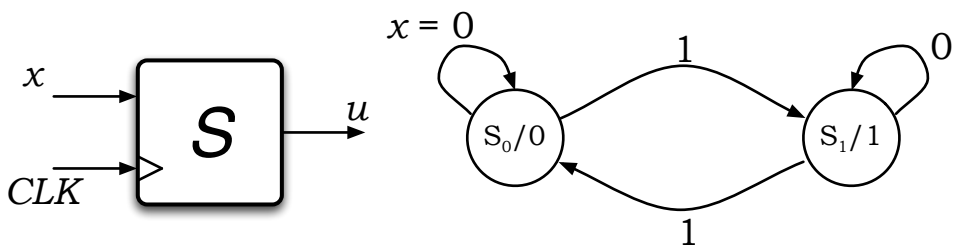
> Konstruera ett sekvensnät av Moore-typ som ständigt byter tillstånd $\{0, 1\}$ när insignalen, x, är hög men står kvar ock stampar i senaste tillstånd om x är låg.

Detta är ett exempel på ett sekvensnät med en insignal. Samma metod kan användas till alla sekvensnät av samma typ oavsett komplexitetsgrad.

Lägg märke till att man i de olika stegen nedan hela tiden beskriver exakt samma sak och som har sitt ursprung i och överensstämmer med textbeskrivningen ovan. De olika stegen är delmoment för att på ett strukturerat sätt översätta textbeskrivningen till tillståndsekvationer och slutligen hårdvara.

Steg 1: Tillståndsgraf

Vi börjar med att fastställa en tillståndgraf för den önskade funktionen: Ett sekvensnät med insignalen x och utsignalen u. Systemet måste vara klockat.



I tillståndsgrafen till höger ser man hur utsignalen hör till tillståndet, dvs redan här kan man se en mooremaskin i vardande. På bågarna anges det värde på x som gör att respektive tillståndsövergång sker.

Steg 2: Tillståndstabell

Tillståndsgrafen kan ekvivalent skrivas i en tillståndstabell om så önskas:

Q	x	Q^+
S0	0	S0
S0	1	S1
S1	0	S1
S1	1	S0

Steg 3: Tillståndskodning

Låt oss för fullständighets skull även införa tillståndskodning i tabellform. Detta steg kan elimineras om det inte är nödvändigt för tydligheten.

Q	Tillståndskodning
S0	0
S1	1

Steg 4: Tillståndstabell med kodning

Ofta kan man hoppa på detta steg direkt från tillståndstabellen i steg 2. Insignalerna är q_0 och x och ur dessa skall nästa tillstånd Q^+ och utsignal u beräknas.

q_0	x	Q^+	u
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

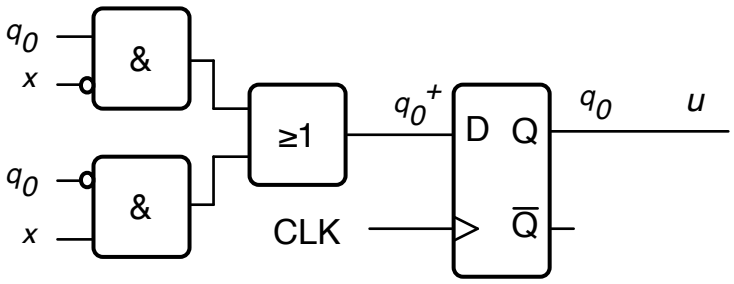
Steg 5: Tillståndsekvationer

Med systemets önskade beteende enligt steg 4 kan vi använda tidigare kunskaper om kombinatoriska nät och dessas minimering för att slutligen erhålla:

$$\begin{cases} q_0^+ &= \overline{q_0}x + q_0\overline{x} \\ u &= q_0 \end{cases}$$

Steg 6: Realisering

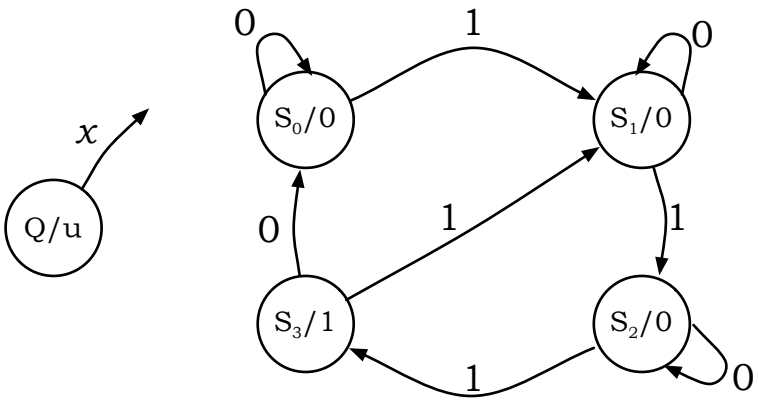
Om uppgiften även kräver *realisering* skall denna redovisas med ett schema:



4.7. Typexempel Moore

Exempel 29. Typexempel Moore

Konstruera ett sekvensnät av typen Moore som uppfyller beskrivningen nedan.



Vi ser direkt att det handlar om ett moorenät då utsignalen hör till tillståndet.[4]

Vi kan också kontrollera att det är komplett definierat då det finns en båge ut från varje tillstånd för $x = 0$ och $x = 1$. Slutligen noterar vi att grafen har fyra tillstånd varför det kommer krävas två vippor för dessa $Q = \{q_1 q_0\} = \{00, 01, 10, 11\}$, dock inte nödvändigtvis i denna ordning.

De olika konstruktionsstegen vi hittills använt följer nu direkt på varann:

Q	x	Q^+	$u = f(Q)$
S0	0	S0	0
S0	1	S1	0
S1	0	S1	0
S1	1	S2	0
S2	0	S2	0
S2	1	S3	0
S3	0	S0	1
S3	1	S1	1

$\rightarrow$

$q_1 q_0 x$	$q_1^+ q_0^+$	$u = f(q_1 q_0)$
000	0 0	0
001	0 1	0
010	0 1	0
011	1 1	0
100	0 0	1
101	0 1	1
110	1 1	0
111	1 0	0

Där graykodade tillstånd använts. Med denna kodning, dvs kodning enligt $S_0 S_1 S_2 S_3 = 00, 01, 11, 10$ får man ofta enklare logik, men det är ingen naturlag. Lägg märke till hur den högra tabellen numrerats i naturlig binär ordning. Detta är praktiskt för att snabbt kunna fylla i KD, vilket är nästa steg:

[4] Denna tillståndsgraf beskriver ett visst beteende beroende på insignalen x. Vilket beteende är det?

q_1^+ \ $q_0 x$	00	01	11	10
q_1 0	Ø	Ø	1	Ø
q_1 1	Ø	Ø	1	1

q_0^+ \ $q_0 x$	00	01	11	10
q_1 0	Ø	1	1	1
q_1 1	Ø	1	Ø	1

Varur man, beroende på vilka ringningar man gör, drar de booleska tillståndsekvationerna

$$\begin{cases} q_1^+ & = q_1 q_0 + q_0 x \\ q_0^+ & = \overline{q_0}\, x + \overline{q_1} x + q_0\, \overline{x} \end{cases}$$

eller

$$\begin{cases} q_1^+ & = q_1 q_0 + q_0 x \\ q_0^+ & = \overline{q_0}\, x + \overline{q_1} q_0 + q_0\, \overline{x} \end{cases}$$

Utsignalen u kan fås fram med ett motsvarande KD men här lätt läsas direkt ur tillståndstabellen

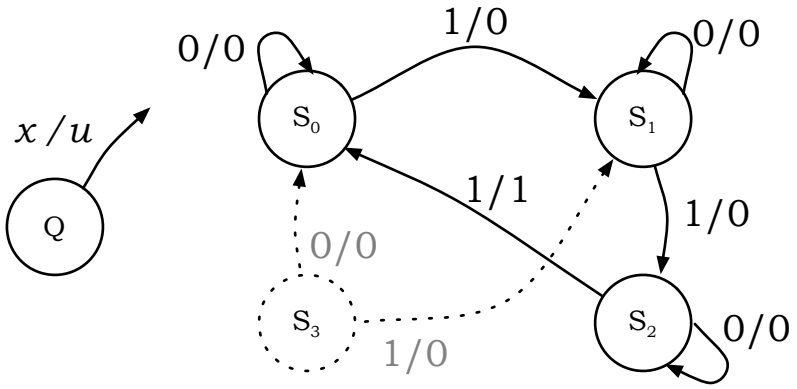
$$u = q_1 \overline{q_0}$$

Med ekvationerna sålunda bestämda återstår realiseringssteget, dvs att rita schemat för apparaten.

4.8. Typexempel Mealy

Exempel 30. Typexempel Mealy

En tillståndsgraf för samma funktion som ovan men utförd som ett mealynät är (bortse från S_3 för tillfället):



I grafen ovan anges in- och utsignal på bågarna enligt x/u: Vid tillståndet S_1 kommer $x = 0$ ge utsignalen $u = 0$ och man kommer stanna kvar i S_1. Insignalen $x = 1$ ger däremot *omedelbart* $u = 1$ och vid nästa klockflank antages tillståndet S_2.

Det visar sig att tre tillstånd är tillräckligt! Givet att vi vid start hamnar i någon av $S_0S_1S_2$ kommer maskinen aldrig kunna nå S_3 varför vi inte behöver ta hänsyn till detta tillstånd. Om vi däremot kan starta i vilket tillstånd som helst måste vi hantera även uppstart i S_3, vilket kan göras enligt de prickade delarna i grafen ovan.[5]

För just denna konstruktion antar vi att starten alltid sker i S_0 och bortser därmed från S_3 (sätter *don't cares* i tillståndstabellens högra kolumn).

Tillståndstabellen anges nu med utsignalen $u = f(Q, x)$ tydligt knuten till *vägen till nästa tillstånd.*

Q	x	Q^+/u		q_1q_0x	$q_1^+q_0^+/u$
S0	0	S0/0		000	00 / 0
S0	1	S1/0		001	01 / 0
S1	0	S1/0		010	01 / 0
S1	1	S2/0	→	011	11 / 0
S2	0	S2/0		100	– – / –
S2	1	S0/1		101	– – / –
S3	0	—/–		110	11 / 0
S3	1	—/–		111	00 / 1

[5]Vad händer om vi råkar starta i S_1 eller S_2 då? Slutsatsen är att vi alltid vill kunna starta från ett känt initialtillstånd!

Graykodade tillstånd har använts även denna gång.

Karnaughdiagram konstrueras nu för $q_1^+ q_0^+ / u$ och ekvationerna extraheras

q_1^+ \ $q_0 x$	00	01	11	10
q_1 0	Ø	Ø	1	Ø
q_1 1	-	-	Ø	1

q_0^+ \ $q_0 x$	00	01	11	10
q_1 0	Ø	1	1	1
q_1 1	-	-	Ø	1

Varur

$$\begin{cases} q_1^+ & = \overline{q_1} q_0 x + q_1 \overline{q_0} \\ q_0^+ & = \overline{q_0}\, x + \overline{q_1} q_0 + q_0\, \overline{x} \end{cases}$$

där tyvärr ingen fördel har kunnat dras av *don't cares* i diagrammet för q_0^+.

Utsignalen u kan återigen fås fram med ett motsvarande KD men läses här direkt ur tillståndstabellen

$$u = q_1 q_0 x$$

Övning: Med en annan kodning kan enklare logik erhållas. Vilken?

Not: Vad hände med S_3? Sätt in $q_1 q_0 x = \{100, 101\}$ i ekvationerna. Man ser att $Q^+(100) = 10 = S_3$ respektive $Q^+(101) = 11 = S_2$ så de slutliga ekvationerna kommer likväl ha ett beteende för S_3. Har man tur är det ett acceptabelt beteende, har man otur kan man ibland göra andra ringningar i KD för att till slut ändå få acceptabelt beteende för S_3.

Observera att S_3:s beteende bara är intressant om man skulle råka hamna i det tillståndet. I denna uppgift antog vi att start/RESET alltid sker till S_0 och då kan aldrig S_3 nås om inget gått väldigt fel.

Om realisering krävs utförs den direkt, eventuellt via omskrivning för tillgängliga komponenter, i ett schema som förut.

4.9. Ett mått på komplexitet

En enkelt mått på komplexitet är antalet ingångar i den resulterande logiken. Den lösning som har färst[6] ingångar anses vara den minimala, åtminstone vid konstruktion med TTL-grindfunktioner.

[6] Yepp! Det är ett ord!

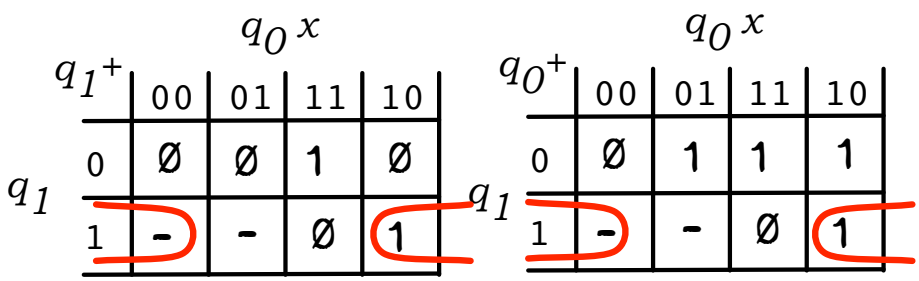
För de två lösningarna till exemplen ovan (med hänsyn till att vi behöver inverterare för x men inte för q_i vilka finns gratis som utgång på varje vippa) kan vi notera

Signal	Moore	Mealy
q_1^+	5	8
q_0^+	11	11
u	2	3
Summa:	18	22

Här vann mooremaskinen med 18 ingångar. Med annan kodning av tillstånden skulle situationen kunna vara den omvända.

4.10. Grinddelning

En möjlighet att minimera antalet ingångar kan öppna sig vid minimeringen. Betrakta KD ur sista exemplet:



Med lite eftertanke ser man att det finns möjlighet till gemensamma ringningar i de båda diagrammen då $q_1\overline{x}$ är en gemensam term i uttrycken både för q_1^+ och q_0^+. Denna term behöver då bara beräknas på ett ställe och resultatet kan användas i båda uttrycken. Metoden kallas *grinddelning* och kan emellanåt användas för att spara ytterligare några ingångar.

4.11. Skillnad mellan Moore- och Mealy-nät

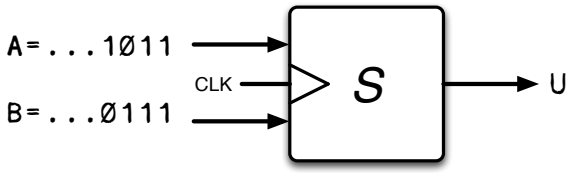
För att belysa en fundamental skillnad i de båda sekvensnätstyperna betraktar vi ett exempel med speciellt fokus på *när* utsignalen ändras.

Exempel 31. Seriell binär jämförare

Skriv tillståndsgrafen för det sekvensnät som jämför två seriella tal A och B,

$$\begin{cases} A &= \{a_n a_{n-1} \ldots a_1 a_0\} \\ B &= \{b_n\, b_{n-1} \ldots\, b_1 b_0\} \end{cases}$$

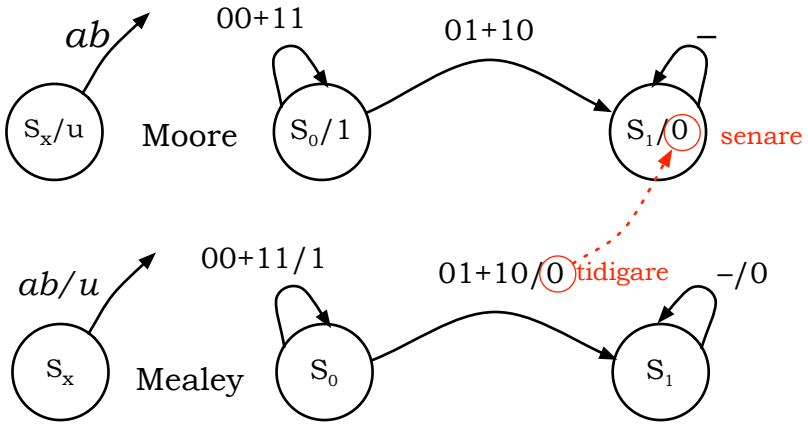
och ger utsignalen 1 om $A = B$.



För att jämföra två tal seriellt jämför man först minst signifikanta biten hos de båda talen och om den biten är lika är talen lika. I nästa steg jämförs den mer signifikanta biten hos de båda talen och om även den är lika är talen fortfarande lika. Så fort likheten upphör är talen olika och kan inte bli lika igen. Utsignalen skall alltså vara 1 så länge $a_i = b_i$.

Talen i figuren är lika de två första klockcyklerna men sedan olika.

Uppgiften kan lösas med sekvensnät enligt Moore och Mealy. Valet av nät ger dock utsignaler vid olika tidpunkt som visas i de två tillståndsgraferna nedan.



Vi övertygar oss först om att tillståndsgraferna faktiskt löser uppgiften:

- För moorefallet: Med start i S_0 antar vi att talen är lika till en början. För detta tillstånd gäller sedan att utsignalen är 1 så länge $a_i = b_i$, bågen "`00+11`". Om $a_i \neq b_i$ sker tillståndsövergång till S_1 och utsignalen blir 0 när detta tillstånd klockats in. Om flödet kommit till detta tillstånd, ett *terminaltillstånd*, kan det inte senare lämnas.

- För mealyfallet: Vi utgår från samma antagande som ovan med utsignalen 1 från början och fortsatt så då $a_i = b_i$. Om insignalerna i något klockintervall skiljer sig, äntrar vi bågen "`01+10/0`" från S_0, ger utsignalen 0 *direkt* och väntar på en klocka för att hamna i terminaltillståndet S_1.

Observera *när* utsignalen ändrar sig. I moorefallet krävs en klockflank för att det beräknade resultatet skall påverka tillståndet och därmed utsignalen. I mealyfallet påverkas utsignalen så fort insignalerna ändras.

Detta bör inte komma som en överraskning utan är en direkt följd av hur de olika sekvensnätstyperna definierats.

Speciellt ska man lägga på minnet att utsignalen i mealyfallet är *rent kombinatorisk*, det finns en direkt väg mellan insignal och utsignal. I moorefallet finns ingen sådan direkt väg och utsignalen är isolerad (med en klockflank) från insignalen via en nivå med D-vippor.

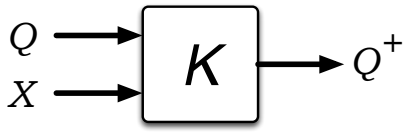
Övning: En utsignal från ett moorenät kan återkopplas till insignalen utan problem medan en återkopplad signal från ett mealynät kan orsaka rundgång (oscillation) då den är oberoende av systemklockan. Rita upp dessa båda scenarion och övertyga dig om att så är fallet.

5. Större sekvensnät

Vid större sekvensnät blir snart vår hittillsvarande realiseringsmetod med grindar otymplig. I detta avsnitt presenteras ett alternativt sätt att realisera sekvensnät som tillåter praktiskt taget godtyckligt stora sekvensnät.

5.1. Tillståndsminne

Hittills har vi använt kombinatorik för att beräkna nästa tillstånd Q^+:



K-nätet fungerar som en ren uppslagning och en *tabell* skulle fungera lika bra. Vi betraktar då Q och X som en sammansatt *adress*

$$q_n q_{n-1} \ldots q_1 q_0 x_m x_{m-1} \ldots x_1 x_0$$

som pekar ut en rad i denna tabell. På den raden är nästa tillstånd beläget.

Exempel 32 Tillståndsminne

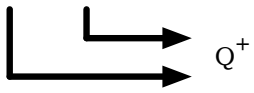
Som exempel på minnesinnehåll i ett tillståndsminne använder vi nu mealy-exemplet tidigare (exempel 30).

Tillståndsekvationerna var

$$\begin{cases} q_1^+ & = \overline{q_1} q_0 x + q_1 \overline{q_0} \\ q_0^+ & = \overline{q_0}\, x + \overline{q_1} q_0 + q_0\, \overline{x} \end{cases}$$

vilka överfört till ett minne med åtta rader blir

	q_1q_0x	q_1^+	q_0^+
0	000	0	0
1	001	0	1
2	010	0	1
3	011	1	1
4	100	0	0
5	101	0	1
6	110	0	1
7	111	1	0

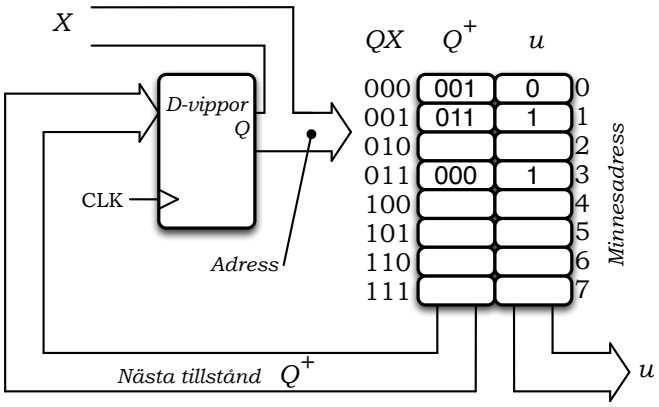


Funktionen $Q^+(q_1q_0x)$ är identisk med den tidigare presenterade kombinatoriska lösningen, men realiseringen sker med en tabell.

Man förstår att tabellösningen förenklar då vi slipper hantera grindlogik med olika minimering beroende på valda tillståndskodningar, tillgängliga komponenter och uppmärksamhet på eventuell grinddelning med mera.

5.2. Automat

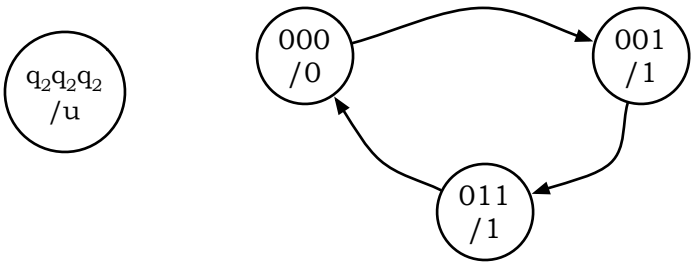
Med en *automat* menar vi ett alternativt sätt att konstruera sekvensnät genom att ta hjälp av minne och D-vippor. Den övergripande strukturen är



där ett antal D-vippor håller nuvarande tillstånd, Q. Detta nuvarande tillstånd tillsammans med eventuella insignaler X utgör sedan radadressen $QX = \{q_nq_{n-1}\ldots q_1q_0x_px_{p-1}\ldots x_1x_0\ldots\}$ till minnet.

Ut från minnet kommer utsignalen som hör till tillstånd/insignal-kombinationen och det nästa tillstånd som D-vipporna skall anta vid nästa aktiva klockflank.

Med minnesinnehållet enligt figuren åstadkoms tillståndsdiagrammet



Där utsignalen programmerats i minnets högra del. Genom att låta insignalen X ingå i adressen kan insignalen påverka hoppsekvensen.

En dylik automat kan användas för att realisera sekvensnät av både mealy- och mooretyp beroende på hur minnet programmeras.

Övning: Hur kan man se om en tillståndstabellslösning motsvarar en en mealy- eller mooremaskin?

5.3. Strukturerade lösningar

Det är lätt att föreställa sig att vi löser alla problem med tillräckligt stora sekvensnät. Konstruktion av stora K-nät via KD är krångligt och tidsödande. Inte minst är en omkonstruktion ofta omfattande att genomföra. Mycket kan förenklas om vi strukturerar lösningen och tänker igenom vilka byggblock som behövs, tillverkar dessa med metoderna hittills och använder dem som KOMPONENTER i konstruktionen.

Så har man traditionellt gjort och i TTL-serien finns åtskilliga ”bra” byggblock från enkla grindar till mer komplicerade funktioner.

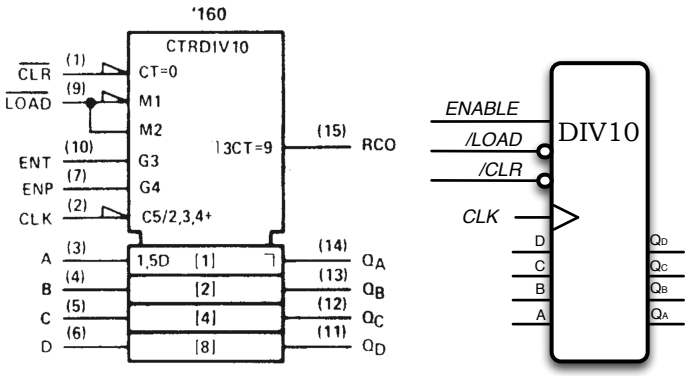
I en större konstruktion, till exempel en mikroprocessor, känner man snart igen sådana byggblock (adderare, skiftregister, räknare, addressavkodare med flera).

Även om digitalteknik numera realiseras via hårdvarubeskrivande språk i programmerbara kretsar, i stället för TTL-kapslar, är ett strukturerat blocktänk användbart och skall inte bortses ifrån.

I ett hårdvarubeskrivande språk kan vi tillverka en 0-till-7-räknare (oktalräknare). Med TTL-byggblock i form av en räknare kan vi modifiera beteendet med lite kringkomponenter vilket exempel 33 visar.

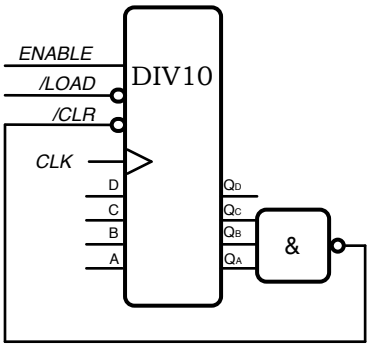
Exempel 33 Oktalräknare

En dekadräknare (till exempel LS160) har räknarsekvensen $0 \rightarrow 1 \rightarrow 2 \ldots \rightarrow 9 \rightarrow 0 \ldots$. Den är försedd med räknevillkor (ENABLET/P), laddning av indata ($\overline{\text{LOAD}}$) samt nollställning ($\overline{\text{CLR}}$). Med $\overline{\text{CLR}} = 0$ blir $Q_{D\ldots A} = 0000$ vid nästa aktiva klockflank.



Använd denna räknare för att realisera en oktalräknare, dvs en räknare med sekvensen $0 \to 1 \to 2 \ldots \to 7 \to 0 \ldots$

Lösningen är att identifiera $Q_{D\ldots A} = 111$ med ett lämpligt K-nät som sedan får nollställa räknaren:

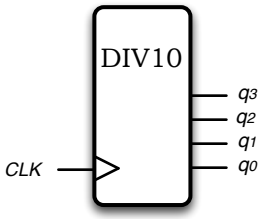


Med kunskap om några *standardfunktioner*: räknare, komparatorer, multiplexer med flera, kan funktionella nät byggas upp på ett effektivt sätt utan att behöva hänge sig åt enstaka grindar mer än på några få ställen.

Exempel 34. Utökad funktionalitet

> Vi ska här visa hur man kan resonera när det gäller att förse en befintlig dekadräknare med signaler för RUN och RESET.

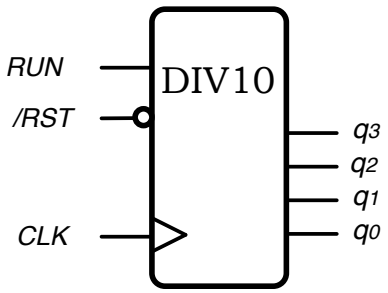
En dekadräknare kan vi konstruera som en mooremaskin:



Vi kan teckna tillhörande tillståndsekvationer $q_i^+ = f_i$ utan att just nu bry oss om de faktiska uttrycken:

$$Q^+ = f_i(Q) \text{ eller bitvis } \begin{cases} q_3^+ & = f_3(q_3q_2q_1q_0) \\ q_2^+ & = f_2(q_3q_2q_1q_0) \\ q_1^+ & = f_1(q_3q_2q_1q_0) \\ q_0^+ & = f_0(q_3q_2q_1q_0) \end{cases}$$

Istället för att omkonstruera fungerande och testade f_i (med påföljande minimeringar, komponentval osv) kan vi lägga till signalerna RUN och nollställning, RST. RST utförs aktiv låg[1], $\overline{\text{RST}}$. Den ytterligare funktionaliteten blir då



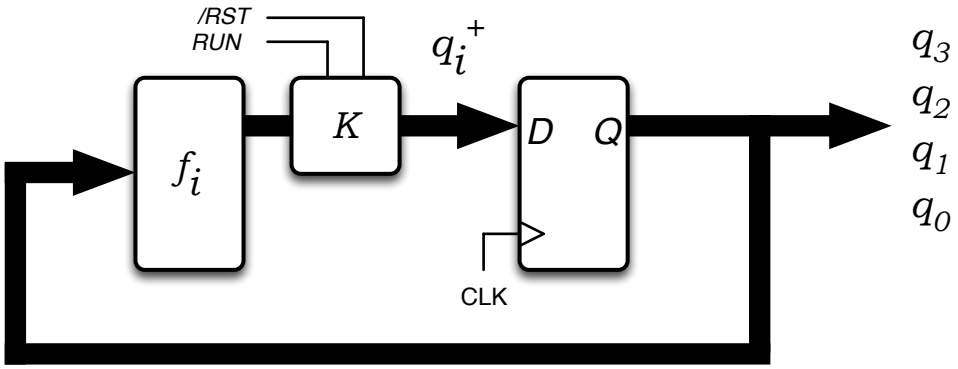
De nya signalerna skall påverka nästa tillstånd enligt

$$\begin{cases} \text{RUN} = 1 & \Rightarrow Q^+ = Q + 1 \\ \text{RUN} = 0 & \Rightarrow Q^+ = Q \\ \overline{\text{RST}} = 0 & \Rightarrow Q^+ = 0 \end{cases}$$

Med dessa nya signaler kan ekvationerna skrivas

$$\begin{cases} q_3^+ & = \overline{\text{RST}} \cdot (\text{RUN} \cdot f_3(q_3q_2q_1q_0) + \overline{\text{RUN}} \cdot q_3) \\ q_2^+ & = \overline{\text{RST}} \cdot (\text{RUN} \cdot f_2(q_3q_2q_1q_0) + \overline{\text{RUN}} \cdot q_2) \\ q_1^+ & = \overline{\text{RST}} \cdot (\text{RUN} \cdot f_1(q_3q_2q_1q_0) + \overline{\text{RUN}} \cdot q_1) \\ q_0^+ & = \overline{\text{RST}} \cdot (\text{RUN} \cdot f_0(q_3q_2q_1q_0) + \overline{\text{RUN}} \cdot q_0) \end{cases}$$

I ett blockschema kan man se att modifieringen rör sig runt, men inte i, funktionsuttrycken f_i:



[1]För det brukar vara så.