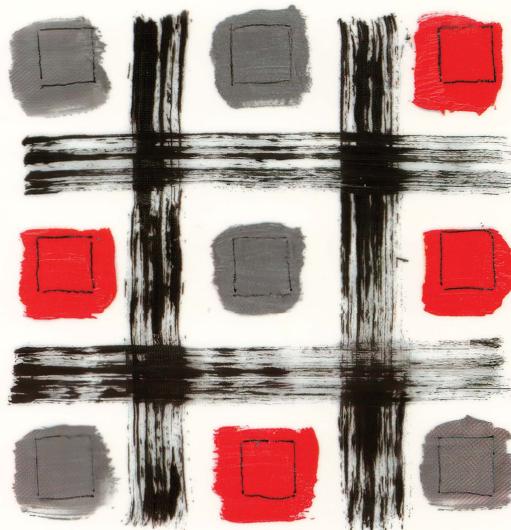


www.

Lars-Hugo Hemert

# Digitala kretsar





Lars-Hugo Hemert

# Digitala kretsar



## KOPIERINGSFÖRBUD

Detta verk är skyddat av lagen om upphovsrätt. Kopiering, utöver lärares rätt att kopiera för undervisningsbruk enligt BONUS-Presskopias avtal, är förbjuden.

Sådant avtal tecknas mellan upphovsrättsorganisationer och huvudman för utbildningsanordnare t.ex. kommuner/universitet. För information om avtalet hänvisas till utbildningsanordnarens huvudman eller BONUS-Presskopia.

Den som bryter mot lagen om upphovsrätt kan åtalas av allmän åklagare och dömas till böter eller fängelse i upp till två år samt bli skyldig att erlägga ersättning till upphovsman/rättsinnehavare.

Denna trycksak är miljöanpassad, både när det gäller papper och tryckprocess.

Art.nr 3454

ISBN 978-91-44-01918-5

© Lars Hugo Hemert och Studentlitteratur 1992, 2001

Omslagsbild: Agneta Hemert

Omslagslayout: LithoMontage AB

Tredje upplagan

Printed in Sweden

Studentlitteratur, Lund

[www.studentlitteratur.se](http://www.studentlitteratur.se)

# Innehåll

## Förord 9

### 1 Inledning 11

- 1.1 Digital – analog 11
  - Analogt telefonsystem 12
  - Digitalt telefonsystem 13
- 1.2 Klassificering av integrerade kretsar 20
  - Kundspecifierade kretsar 21
  - Standardkretsar 23
- 1.3 Talsystem och koder 25
  - Representation av tal i talsystem med olika baser 25
  - Omvandling mellan olika baser 28
  - Elementär aritmetik för binära, hexadecimala och oktala heltal utan tecken 32
  - Numeriska och alfanumeriska koder 34
- 1.4 Realisering av en liten krets i PLD – en introduktion till VHDL 41
  - Specifikation 42
  - Beskrivning 43
  - Syntes 50
  - VHDL – ett måste vid modern kretskonstruktion 52
- 1.5 Övningsuppgifter 53

### 2 Grindar, vippor, kombinations- och sekvenskretsar 56

- 2.1 Grindar och kombinationskretsar 56
  - OCH-grind 56
  - ELLER-grind 58
  - Inverterare 58
  - Några kombinationskretsar 59
  - NAND-grind 72
  - NOR-grind 74

	Exklusivt-ELLER-grind, XOR-grind	74
	Några fler kombinationskretsar	75
	Three-state-utgång	77
	Fördräjning	78
2.2	Vippor och sekvenskretsar	80
	D-vippan	80
	Register och skiftregister	82
	Räknare	84
	Några generella sekvenskretsar	91
2.3	Övningsuppgifter	96
<b>3</b>	<b>Boolesk algebra</b>	102
3.1	Boolesk algebra – definition	103
3.2	Räknelagar	104
	Räknelagar för en variabel	104
	Räknelagar för flera <u>variabler</u>	105
3.3	Operationen XOR (Exklusivt-ELLER)	111
	Räknelagar	111
3.4	Övningsuppgifter	115
<b>4</b>	<b>Kombinationskretsar</b>	118
4.1	Definition av kombinationskrets	119
4.2	Booleska funktioner	120
4.3	Förenkling och realisering av booleska funktioner i grindnät	126
	Karnaughdiagram	127
	NAND- och NOR-nät	139
	Ofullständigt specificerade funktioner	142
	Grinddelning	145
	Standardgrindnät i PLD för realisering av booleska funktioner	147
	Faktorisering	152
4.4	Quine-McCluskeys förenklingsmetod	155
4.5	Kombinationskretsar i tidsplanet	167
	Hasard	167
	Kapplöpning	169
4.6	Adderare	171
4.7	Aritmetisk Logisk Enhet (ALU)	175
	Aritmetik	175
	Aritmetisk enhet	185
	Logisk enhet	188
	Aritmetisk Logisk Enhet (ALU)	190

- 4.8 Paritetskrets 196
  - Felupptäckande och felnärtande kod 196
  - Paritetskrets 200
- 4.9 Komparator för likhet 201
- 4.10 Övningsuppgifter 202

## 5 Sekvenskretsar 213

- 5.1 Generella sekvenskretsar 213
  - Realisering av en sekvenskrets typ Moore 213
  - Realisering av en sekvenskrets typ Mealy 227
- 5.2 Räknare 250
  - Binärräknare modulo- $2^n$  med  $n$  bitar 251
  - Dekadräknare 269
- 5.3 Register och skiftregister 274
  - Serie in/parallel in – serie ut/parallel ut 274
  - Serie in/parallel in – serie ut/parallel ut – skift höger/skift vänster 275
- 5.4 Latchar 277
  - SR-latchen 277
  - D-latchen 281
- 5.5 Asynkrona sekvenskretsar 284
  - Realisering av en asynkron sekvenskrets 286
- 5.6 Synkronisering av asynkrona signaler och metastabilitet 299
- 5.7 Övningsuppgifter 302

## 6 MOS-transistorn

### Grindar i CMOS och nMOS 312

- 6.1 MOS-transistorn 313
  - Uppbyggnad 313
  - Funktion 317
  - Kapacitanser 322
  - Liten historik 324
- 6.2 CMOS-inverteraren 324
  - Struktur 324
  - Logisk funktion 326
  - Statiska egenskaper 328
  - Dynamiska egenskaper 334
  - Effektförbrukning 337
- 6.3 nMOS-inverteraren 338
  - nMOS-inverterare med en resistor som passiv pull-up 338

	nMOS-inverterare med en transistor som aktiv pull-up	339
6.4	Grindar i CMOS och nMOS	341
	NAND-grind	341
	NOR-grind	343
	OCH-grind och ELLER-grind	344
	Exklusivt-ELLER-grind, XOR-grind	345
	Three-state-utgång	346
6.5	Övningsuppgifter	349

## 7 Halvledarminnen 352

7.1	Inledning – minnesmodell och en översikt av halvledarminnen	352
	Minnesmodell	353
	Klassificering av halvledarminnen	357
7.2	Läsminnen	361
	ROM	361
	Expansion till ett minne med flera kapslar	364
	PROM	367
	Modell för ett minneschip	369
	EPROM	370
	EEPROM	374
	Flash-minnen	376
7.3	Läs/skriv-minnen	378
	Statiska RWM (SRAM)	378
	Dynamiska RWM (DRAM)	383
7.4	Övningsuppgifter	389

## 8 Programmerbara logiska kretsar 391

## 9 VHDL – en introduktion 398

9.1	VHDL – bakgrund	398
9.2	Beskrivning av kombinationskretsar	399
	Parallella (concurrent) signaltilldelningar	399
	Process	404
9.3	Beskrivning av generella sekvenskretsar	407
9.4	Beskrivning av räknare	415
9.5	Beskrivning av skiftregister	419
9.6	Strukturbeskrivning	420
9.7	Beskrivning av en D-vippa och en D-latch	429
	Beskrivning av en D-vippa	429

Beskrivning av en D-latch	430
<b>9.8 Övningsuppgifter</b>	<b>431</b>
<b>10 D/A- och A/D-omvandlare</b>	<b>434</b>
10.1 D/A-omvandlare	435
D/A-omvandlare med spänningsdelare	436
D/A-omvandlare med viktade resistorer	438
D/A-omvandlare med R-2R-resistorstege	440
Fel i överföringskarakteristiken hos D/A-omvandlare	442
Unipolär och bipolär utsignal	444
D/A-omvandling genom pulsbreddsmodulering (PWM)	446
10.2 A/D-omvandlare	447
A/D-omvandlare – typ parallell ("flash")	448
A/D-omvandlare – typ nivåramp	450
A/D-omvandlare – typ successiv approximation	452

**Appendix 1 2-potenser** 455

**Appendix 2 ASCII-koden** 456

**Appendix 3 PSpice simuleringsfiler** 457

**Appendix 4 Halvledarminnen – datablad** 459

**Appendix 5 PLD – datablad** 472

**Svar till övningsuppgifter** 500

1 Inledning	500
1.3 Talsystem och koder	500
1.4 Realisering av en liten krets i VHDL - en introduktion till VHDL	502
2 Grindar, vippor, kombinations- och sekvenskretsar	504
2.1 Grindar och kombinationskretsar	504
2.2 506	
Vippor och sekvenskretsar	506
3 Boolesk algebra	509
4 Kombinationskretsar	511
4.2 Booleska funktioner	511

4.3 Förenkling och realisering av booleska funktioner i grindnät	511
4.5 Kombinationskretsar i tidsplanet	517
4.6 Adderare	517
4.7 Aritmetisk logisk enhet (ALU)	518
<b>5 Sekvenskretsar</b>	<b>519</b>
5.1 Genrella sekvenskretsar	519
<b>6 MOS-transistorn</b>	
Grindar i CMOS och nMOS	530
6.4 Grindar i CMOS och nMOS	530
<b>7 Halvledarminnen</b>	<b>532</b>
<b>9 VHDL – en introduktion</b>	<b>535</b>
<b>Sakregister</b>	<b>572</b>

# Förord

Denna bok är avsedd som kurslitteratur för en grundkurs i digitalteknik på högskolenivå. Innehållet är till största delen av fundamental och tidlös karaktär, nödvändigt för att förstå digitala kretsars funktion och för att kunna konstruera digitala kretsar.

Huvuddelen av boken ägnas åt kombinationskretsar och sekvenskretsar, de fundamentala klasserna av digitala kretsar. Kombinationskretsar är kretsar utan minne, medan sekvenskretsar är kretsar med minne. Efter en inledning om begreppen analog och digital samt binära talsystem *analyseras* enkla kombinations- och sekvenskretsar uppbyggda med digitalteknikens minsta byggstenar, grindarna och vipporna. Tanken är att denna analys skall belysa begreppen kombinationskrets och sekvenskrets. Förutom analys av generella kretsar visas där också exempel på fundamentala kretsar som används som byggblock i digitala kretsar, såsom multiplexer, demultiplexer, avkodare, räknare och register. Efter analysen av kretsarna följer en genomgång av boolesk algebra, ett ovärderligt hjälpmittel för analys och syntes av digitala kretsar, som varje digitaltekniker måste vara förtrogen med. Därefter följer sedan en ingående behandling av *syntes* av kombinations- och sekvenskretsar.

Konstruktion, syntes, av digitala kretsar sker med mycket avancerade datorbaserade syntesverktyg. Vid konstruktionsprocessens början beskrivs kretsen som skall konstrueras, utgående från kretsspecifikationen, i något formellt språk för inmatning i datorn i syntesverktyget. Det finns ett standardiserat språk för beskrivning av kretsar, språket **VHDL** (*Very High Speed Integrated Circuit Hardware Description Language*), skapat under första hälften 1980-talet och med en första standard 1987. Redan i bokens inledningskapitel ges en försmak av språket VHDL med ett litet exempel där en kombinationskrets beskrivs i VHDL. I kapitel 9 ges en fylligare introduktion till språket VHDL för beskrivning av kombinations- och sekvenskretsar. Många av kombinations- och sekvenskretsarna som behandlas i de tidigare kapitlen i den löpande texten och i övningsuppgifterna, beskrivs i där

**VHDL.** Det är lämpligt att samtidigt som kombinations- och sekvenskretsar studeras i kapitel 4 och 5, parallellt studera beskrivning av kretsarna i VHDL i kapitel 9, och om det finns tillgång till syntesverktyg, realisera dem i programmerbara kretsar. En kurs i digitalteknik kräver praktiskt arbete med syntesverktyg och kretsar för fullgott resultat.

Filerna till de i den löpande texten i kapitel 9 i VHDL beskrivna kretsarna samt filerna till kretsarna som skall beskrivas i övningsuppgifterna finns att hämta på <http://www.studentlitteratur.se/3454>. Beskrivningarna är gjorda i VHDL-93, nuvarande standard som följer på första standarden VHDL-87.

Introduktionen av VHDL i kapitel 9 ger hjälp att ta första steget in i VHDL. Även om det är ett ganska litet steg, VHDL är ett stort språk och här behandlas bara en delmängd av VHDL, så ger det möjlighet att att beskriva och realisera relativt komplicerade kretsar.

I boken behandlas även halvledarminnen och Digital/Analog- och Analog/Digital-omvandlare. Vidare visas hur digitalteknikens minsta byggstenar, grindarna, realiseras med MOS-transistorer, den komponent som varit och är förutsättningen för den fantastiska utvecklingen av digitala kretsar.

Det är min förhoppning att boken skall ge en gedigen grund i digitalteknik för yrkesliv och vidare studier. Lycka till!

Lund i augusti 2001

Lars-Hugo Hemert

# 1 Inledning

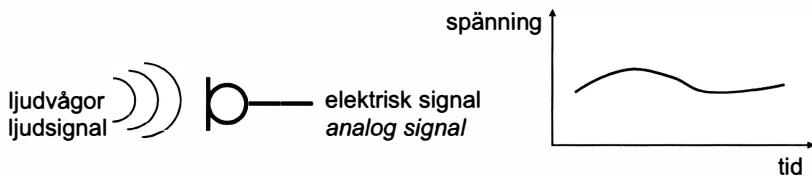
Kapitlet inleds med en analys av begreppen *digital* och *analog*, skillnaden mellan digital och analog signal samt fördelar med digital representation. Därefter följer en liten klassificering av integrerade kretsar. Bl. a. inordnas där de viktiga klasserna *programmerbara logiska kretsar (PLD)* och *tillämpningsspecifika kretsar* (eng. *Application Specific Integrated Circuits, ASIC*). Informationen i digitaltekniken kodas normalt binärt som binära ord eller tal. Framställningen i boken förutsätter att man är väl förtrogen med binära talsystemet. I detta inledningskapitel behandlas därför representation i viktiga talsystem i digitaltekniken, såsom binära, hexadecimala och oktala talsystemen samt några olika koder för de tio decimala siffrorna 0 till 9. Kapitlet avslutas med ett litet exempel som belyser arbetsgången vid realisering av en digital krets i en PLD och ger en introduktion till VHDL.

## 1.1 Digital – analog

*Digital*: ”som avser siffror”, (SAOL, 1998). Ordet kommer från det engelska ordet *digit* (sv. *siffra*), som i sin tur kommer från det latinska ordet *digitus* (sv. *finger, tå*). Vi träffar också på ordet i *digitalis*, det latinska namnet på fingerborgsblomma. Ordet *digit* härstammar synbarligen från det primitiva sättet att räkna på fingrarna.

I digitala system representeras informationen med siffror och informationsöverföringen sker med *digitala signaler*.

En *signal* är en informationsbärande storhet för kommunikation mellan människor, djur, maskiner etc. Ordet *signal* kommer från det latinska ordet *signum*, ”tecken”. Exempel på signaler i tekniska sammanhang är: *elektrisk spänning* och *elektrisk ström*. Motsatsen till en digital signal är en *analog signal*. Exempel på en *analog signal* är den elektriska spänningen från en mikrofon.



Figur 1.1 Exempel på en analog signal.

Ordet *analog* är grekiska och betyder *liknande, överensstämmande*.

Ljudsignalens variationer representeras här med en fysikalisk storhet, elektrisk spänning, vars amplitud är direkt proportionell mot ljudsignalen. Den elektriska spänningen *liknar* ljudsignalen, den är en *analog signal*.

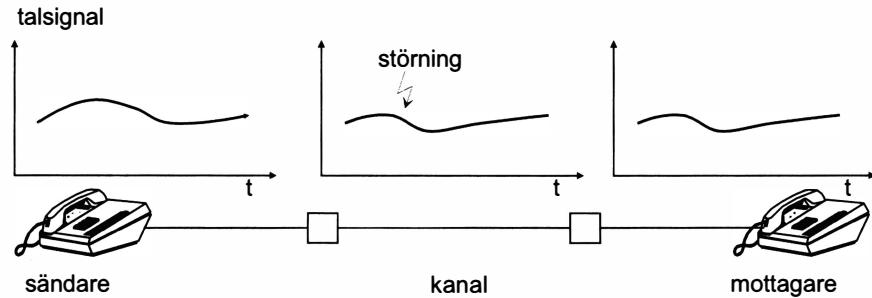
Den analoga signalen är en *kontinuerligt* varierande signal och ordet *analog* används synonymt med ordet *kontinuerlig* om signaler.

En *digital signal* är en *diskontinuerlig signal*.

Ljudsignaler representeras idag i stor utsträckning som digitala signaler, exempelvis i digitala telefonsystem, CD-skivor och digital radio.

Låt oss studera skillnaden mellan analog och digital representation av ljudsignalen i telefonsystemet och fördelar med digital representation.

## Analogt telefonsystem

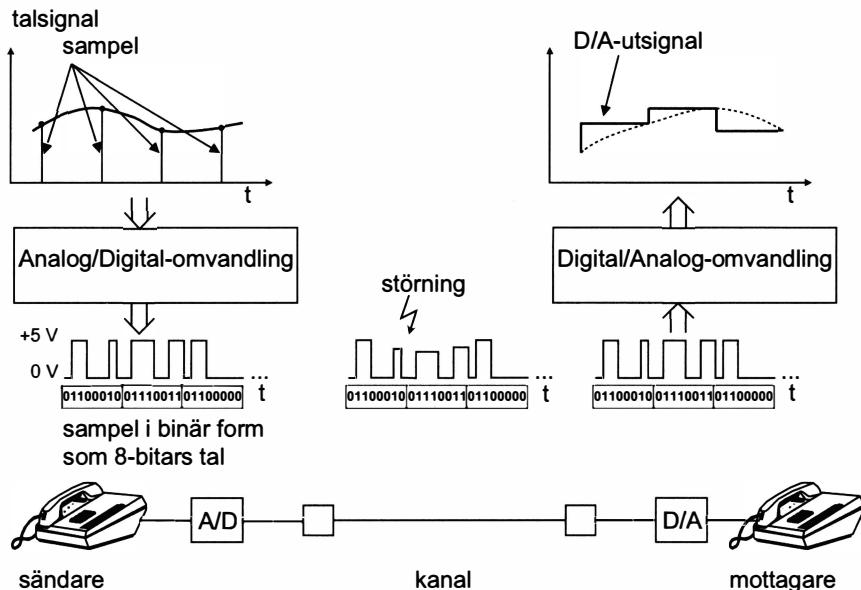


Figur 1.2 Princip för överföring av talsignalen i ett analogt telefonsystem.

I ett analogt telefonsystem överförs talsignalen från sändare till mottagare som en analog signal i princip i samma form som den har efter mikrofonen i telefonen. Störningar i överföringskanalen som påverkar talsignalens amplitud, enligt figuren ovan, är kritiska och försämrar kvaliteten.

## Digitalt telefonsystem

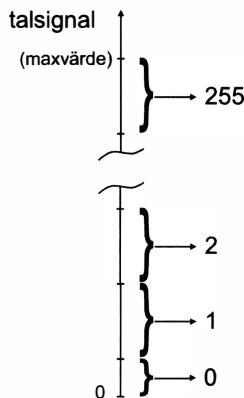
I ett digitalt telefonsystem överförs talsignalens amplitud i form av siffervärden. Den analoga talsignalen från mikrofonen *samples*, mäts, (eng. *sample* = sv. *prov, mätvärde*) med jämna mellanrum och sampelns siffervärdet överförs genom kanalen till mottagaren, där sifervärdena återskapas till en analog talsignal.



Figur 1.3 Princip för överföring av talsignal i ett digitalt telefonsystem.

Omvandlingen av sampelen vid sändaren från analog till digital form görs i en *Analog/Digital-omvandlare* (*A/D*) och omvänt vid mottagaren i en *Digital/Analog-omvandlare* (*D/A*).

Varje sampel tilldelas av A/D-omvandlaren ett sifervärde. Man har för telefonsystemet bestämt att talsignalen skall kunna anta 256 olika sifervärden, värdena 0, 1, 2, ..., 255. Talsignalen *kvantiseras* i 256 olika intervall, innebärande att alla nivåer hos talsignalen inom ett visst intervall tilldelas samma sifervärde, enligt principen i figuren nedan.



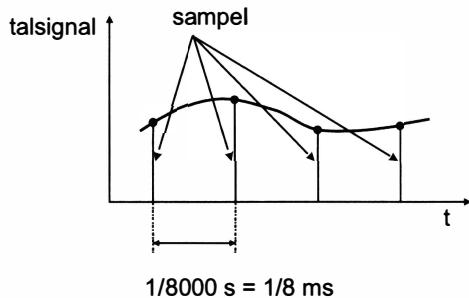
Figur 1.4 Princip för kvantisering av en analog signal i 256 olika nivåer.

De minsta byggstenarna i digitala system, *switcharna*, är *tvåvärdiga, binära*, har två lägen, 'till' och 'från', 'leder ström' respektive 'leder inte ström'. Digitala signaler har därför bara två värden, normalt +5V och 0V vid representation med elektrisk spänning. Informationen i digitala system kodas binärt med kodord innehållande bara siffrorna 0 och 1. Avbildningen av signalnivåerna +5V och 0V på siffrorna 0 och 1 görs vanligen så att +5V avbildas på siffran 1 och 0V avbildas på siffran 0, s.k. *positiv logik*, men även den omvänta avbildningen kan förekomma, s.k. *negativ logik*.

I det digitala telefonsystemet kan talsignalens 256 nivåer kodas med *kodord*, i detta fall binära tal, som har 8 binära siffror, *bitar*, (eng. *binary digits*),  $2^8 = 256$ . Kodorden är 00000000, 00000001, ..., 11111111.

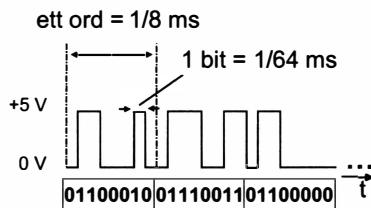
*Samplingsteoremet* (amerikanen *Claude Shannon*, 1949) säger att: "All information i ursprungssignalen finns med i den med sampel beskrivna signalen, om samplingsfrekvensen är större än dubbla högsta frekvensen i ursprungssignalen".

Redan för det analoga telefonsystemet bestämdes att frekvensområdet skall vara 300–3400Hz. Högsta frekvensen i talsignalen som överförs är alltså 3400Hz och enligt samplingsteoremet skall då samplingsfrekvensen vara minst  $2 \cdot 3400 = 6800$  Hz. Samplingsfrekvensen bestämdes till 8000Hz.



Figur 1.5 Sampling av talsignal med samplingsfrekvensen 8000 Hz.

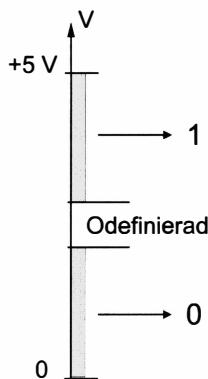
Med samplingsintervallets längd  $1/8$  ms och kodordets samtliga 8 bitar fördelade över hela intervallet, så blir *bitlängden*  $1/64$  ms och bithastigheten 64000 bit/s. Den binära kodningen av samplen kallas *Pulskodmodulering, PCM* (eng. *Pulse Code Modulation*).



Figur 1.6 Sampel av talsignalen i PCM med 8 bitar.

I figur 1.3 ovan visas påverkan av en störning på de digitala ordenen i överföringskanalen. Amplitudstörningar är inte så kritiska i digitala system. Så länge nivåerna för ettorna och nollorna håller sig i 1-intervallet respektive 0-intervallet som är definierat för de digitala komponenterna och ej hamnar i det odefinierade området, så påverkas ej informationen i kodordet. Den

fundamentala principen för binära signaler i digitaltekniken visas i figur 1.7 nedan. Den binära signalen här exemplifierad med en spänning i området 0 till +5 V, är kvantiserad i två *diskreta* (åtskilda) intervall, ett 1-intervall och ett 0-intervall, åtskilda av ett område där signalen är odefinierad. Digitala signaler får statiskt bara ligga i 1-intervallet och 0-intervallet och endast vid omslag befina sig i det odefinierade området.

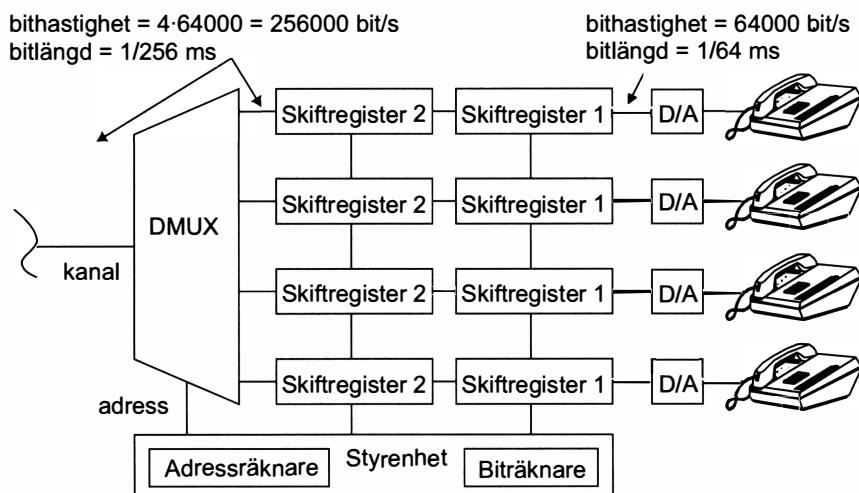
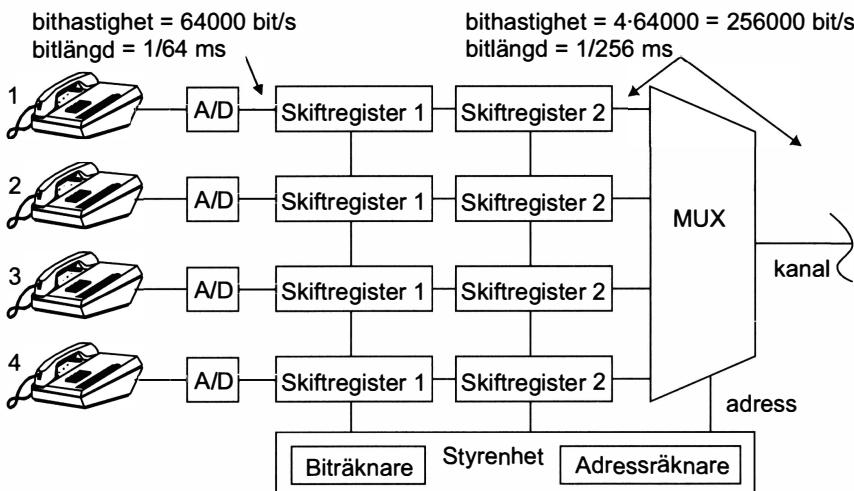


Figur 1.7 Principen för en binär signal.

En stor fördel med digital representation jämfört med analog är att det inte är några större noggrannhetskrav på komponenterna, det spelar ingen roll var i det tillåtna intervallet signalen ligger, bara den ligger i intervallet och ej i det odefinierade området. Med relativt onoggranna komponenter kan man med digital representation uppnå godtyckligt hög precision genom att bara ta till tillräckligt många binära siffror. Med analog representation representeras värden med en kontinuerlig signal, en fysikalisk storhet, t.ex. elektrisk spänning, vars storlek är direkt proportionell mot värdet, och precisionen blir helt avhängig komponenternas noggrannhet. I digitala system har man noggrannhetskrav i gränssnittet till den analoga världen, i det digitala telefonsystemet ovan i A/D- och D/A-omvandlaren.

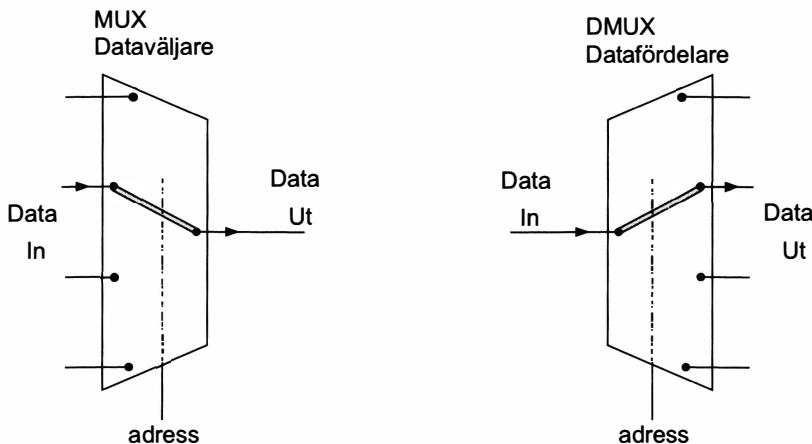
En annan fördel med digitala system är att de digitala komponenterna kan återbilda ”dåliga” ettor och nollor till ”ordentliga” ettor och nollor, som antyts i figur 1.3 ovan, där trots störningar i kanalen ”ordentliga” ettor och nollor når mottagaren.

Låt oss återvända till det digitala telefonsystemet. Samplingen och den digitala representationen möjliggör att överföra flera telefonsamtal på samma ledning (kanal). Binära ord från flera telefonsamtal kan på sändarsidan sammansättas i en *Multiplexer (MUX)* och på mottagarsidan fördelats ut igen i en *Demultiplexer (DMUX)*. Principen visas i blockschemat nedan för fyra telefonsamtal.



Figur 1.8 Princip för tidsmultiplex av fyra telefonsamtal i ett digitalt telefon-system.

En MUX är en *dataväljare* (eng. *data selector*) och en DMUX är en *datafördelare* (eng. *data distributor*) vars princip visas nedan i en mekanisk analogi (MUX och DMUX i det digitala telefonsystemet ovan är naturligtvis elektroniska komponenter som kan göra omkopplingar mellan kanalerna på någon nanosekund; i äldre telefonsystem var det faktiskt mekaniska komponenter enligt nedan). I MUX:en väljs med adressen en av de fyra inkommande kanalerna och en dataväg upprättas till den utgående kanalen. I DMUX:en väljs med adressen en av de fyra utgående kanalerna och en dataväg upprättas från den inkommande kanalen.

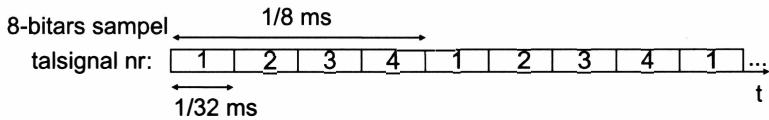


Figur 1.9 Mekanisk analogi för en MUX och en DMUX.

Funktionen hos det digitala telefonsystemet ovan med de fyra telefonerna är i stora drag följande. – De fyra talsignalernas 8-bitars sampel inmatas bit för bit, i serieform, i skiftregister 1 (en typ av minne för lagring av ett ord) med 64000 bit/s. När hela ordet inkommit överförs det omedelbart till skiftregister 2, samtliga bitar på en gång, i parallellform. I skiftregister 1 kan då nästa 8-bitars sample inmatas. Orden i skiftregister 2 sänds sedan ett i taget via multiplexern ut på kanalen, med en hastighet som är 4 gånger så hög som den tidigare hastigheten, eftersom 4 ord skall hinna sändas på samma tid som det tar för ett ord att inkomma i skiftregister 1.

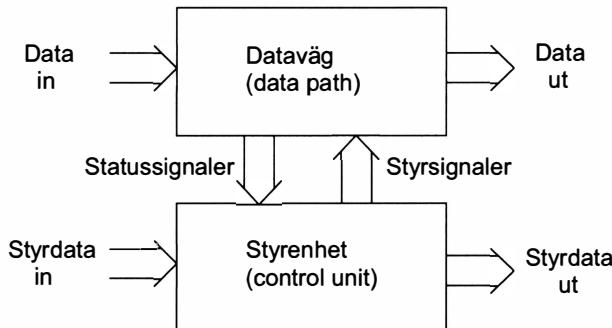
På mottagarsidan inkommer orden ett i taget via demultiplexern till skiftregister 2. När hela ordet inkommit i skiftregister 2 överförs det omedelbart till skiftregister 1, varifrån bitarna skiftas ut. Kanalen utnyttjas alltså av de fyra talsignalerna i tur och ordning under  $(1/8)/4 = 1/32$  ms för

varje 8-bitars sampel enligt figur 1.10 nedan. Denna princip för tidsuppdelning av kanalen benämnes *tidsmultiplex*.



Figur 1.10 Överföring av de fyra talsignalerna i tidsmultiplex på kanalen.

Det digitala telefonsystemet ovan med 4 telefoner har en uppbyggnadsprincip som är karakteristisk för digitala system, en uppdelning i en *dataväg* (eng. *data path*) och en *styrenhet* (eng. *control unit*) enligt figur 1.11 nedan.



Figur 1.11 Uppbyggnadsprincip för digitala system.

Styrenheten kan innehålla mikrodatorer, mikrostyrkretsar, specialkretsar av standardtyp, kundanpassade kretsar etc. I figur 1.8 ovan har antypts att det i styrenheten bör finnas räknare, i detta fall en biträknare som håller reda på antalet bitar och en adressräknare som adresserar in- och utgångarna i MUX:en respektive DMUX:en. Datavägen innehåller normalt MUX:ar och DMUX:ar samt skiftregister och register.

Räknare, skiftregister, multiplexer och demultiplexer är fundamentala kretsar vars uppbyggnad och funktion vi kommer att studera längre fram i boken.

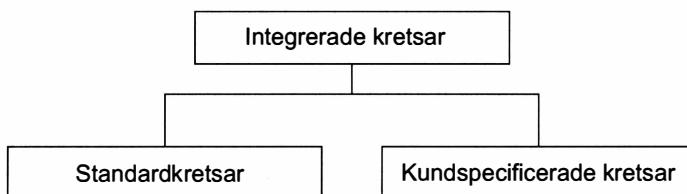
Ovan nämntes bl.a. kundanpassade kretsar, en klass av integrerade kretsar, och i avsnittet som följer görs en liten klassificering av integrerade kretsar.

## 1.2 Klassificering av integrerade kretsar

Först ett par begrepp. – *Halvledarfabrikanten* (eng. *semiconductor manufacturer*) tillverkar de integrerade kretsarna. *Kunden* (eng. *customer*) köper dessa kretsar för att använda dem i sina produkter. Tillverkning av integrerade kretsar kräver mycket hög kompetens och mycket stora investeringar och det finns därför bara ett relativt litet antal halvledarfabrikanter i världen.

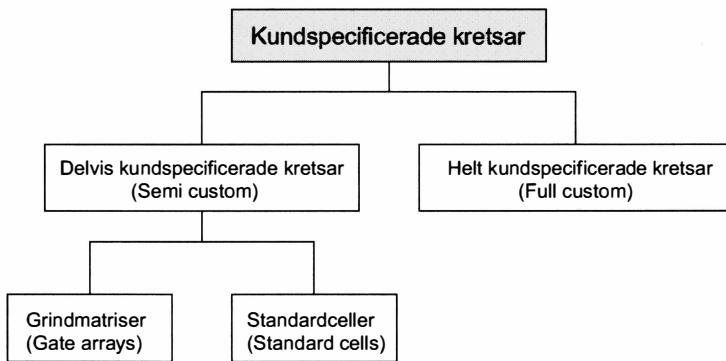
En första indelning av integrerade kretsar kan göras i *standardkretsar* och *kundspecifierade* (eng. *custom specified*) *kretsar*. Kundspecifierade kretsar specificeras, som namnet säger, av kunden och tillverkas sedan av halvledarfabrikanten. Specificationen kan vara på olika nivå, från beteende till struktur. Ena ytterligheten är en fullständig specifikation av kretsens struktur på kiselnivå med alla masker. Specifikationen är då alltså en fullständig konstruktion av kunden (eng. *full custom design*) och förutsätter då mycket hög kompetens och avancerade konstruktionsverktyg hos kunden. Andra ytterligheten är bara en specifikation av kretsens beteende och konstruktionen överläts sedan helt till halvledarfabrikanten.

Kundspecifierade kretsar konstrueras och tillverkas för tillämpning i en speciell produkt, de är s.k. *tillämpningsspecifika kretsar* (eng. *Application Specific Integrated Circuits, ASIC*). 1990-talet har inneburit det stora genombrottet för ASIC, vilket till viss del får tillskrivas framsteg inom halvledarteknologin, men till största del framsteg för avancerade datorbaserade utvecklingsverktyg som kan köras på persondatorer och arbetsstationer. Exempel på användning av ASIC är i mobiltelefoner.



Figur 1.12 Klassificering av integrerade kretsar.

## Kundspecifierade kretsar



Figur 1.13 Klassificering av kundspecifierade kretsar.

### Delvis kundspecifierade kretsar (semi custom)

#### Grindmatriser

Halvledarfabrikanten tillverkar en *grindmatris* (eng. *gate array*) med flera hundratusen *grindar* (eng. *gate*) placerade på chippet. Grindar är fundamentala byggstenar i alla digitala kretsar, uppbyggda av de tidigare nämnda tvåvärda switcharna och ligger i komplexitetsnivå just ovanför dessa. I grindmatrisen ligger grindarna i ett matrismönster, ett s.k. "hav av grindar" (eng. *sea of gates*) eller i block med mellan liggande kanaler för förbindningar. Grindarna är inte förbundna och kretsen har sålunda i detta stadium ingen definierad funktion. Grindmatrisen är ett halvfabrikat som halvledarfabrikanten håller som lagervara. Utgående från halvledarfabrikantens specifikation av grindmatrisen konstruerar kunden sin krets och bestämmer grindarnas förbindningsmönster, grindar är ju de minsta byggstenarna i alla digitala kretsar. Kunden sänder sedan in beskrivningen av förbindningsmönstret för sin krets till halvledarfabrikanten, som tar grindmatrisen, halvfabrikatet, från lagret och färdigtillverkar kretsen genom att lägga på det aktuella förbindningsmönstret, normalt i form av tre masker, två för metalliseringsslager av förbindningar och en för förbindningar mellan lagren.

Vid konstruktion med grindmatris kan ej grindarna utnyttjas till hundra procent. Normalt kan bara ca 50–75 % av grindarna i matrisen utnyttjas, beroende på restriktioner vid förbindning av grindarna. Om exempelvis en krets kräver ca 500.000 grindar, så måste man alltså realisera den i en grindmatris med ca 1000.000 grindar. Kiselytan på chippet utnyttjas sålunda inte optimalt vid grindmatriser, medförande högre kiselkostnad jämfört med konstruktion med standardceller. Tiden från beställning av en grindmatris till leverans rör sig om ett antal veckor. Grindmatriskretsar lämpar sig för medelstora volymer.

### *Standardceller*

Kunden konstruerar sin krets med standardceller ur en katalog som tillhandahålls av halvledarfabrikanten som skall tillverka kretsen. Standardcellerna kan vara typ enkla räknare, register e.d., men kan också vara komplicerade standardkretsar typ mikroprocessorer, signalprocessorer e.d. Kunden placerar ut standardcellerna på chippet och drar förbindningarna. Kretsschemat översänds till halvledarfabrikanten som gör maskerna och tillverkar kretsen. Till skillnad från tillverkning med grindmatriser så utgår nu halvledarfabrikanten inte från ett halvfabrikat utan chippet tillverkas från början i alla processteg. Dock har halvledarfabrikanten färdiga masker för sina standardceller som kan integreras med det av kunden genererade förbindningsmönstret. Jämfört med grindmatriser så utnyttjas kiselytan här mer optimalt, det finns i detta fall inga outnyttjade komponenter på kiselytan. Tiden från beställning till leverans är längre än för grindmatriser och rör sig om några månader. Standardceller lämpar sig för stora volymer av kretsar.

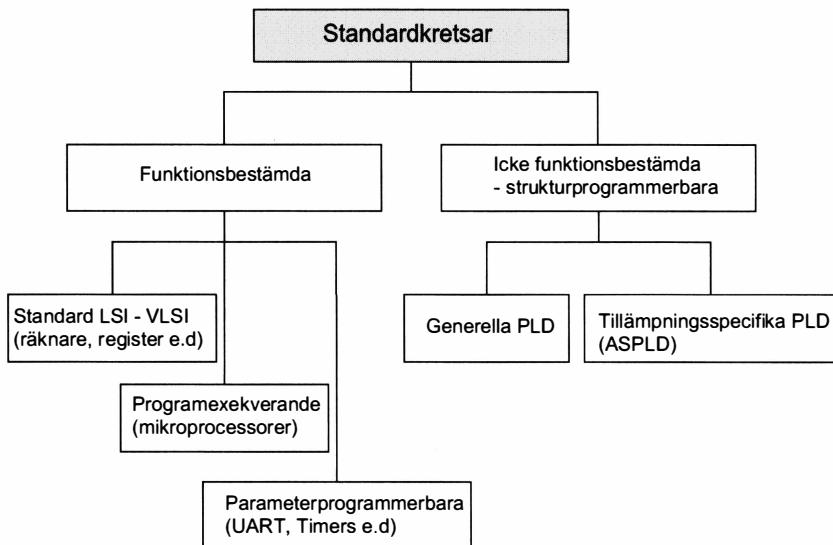
### *Helt kundspecifierade kretsar (full custom)*

Halvledarfabrikanten tillverkar samtliga masker från början. Det krävs mycket stor kompetens av kunden för att utföra konstruktion på denna nivå och det sker då normalt i nära samarbete med halvledarfabrikanten eller görs helt av halvledarfabrikanten. Denna typ av konstruktion är lämplig bara för kretsar i mycket stora volymer, typ fickräknare, klockor etc. Ibland kan en kundspecifierad krets så småningom bli en standardkrets. Exempel på en sådan krets är mikroprocessorn, som 1969 beställdes hos det amerikanska företaget Intel, då bara ett år gammalt, som en kundspecifierad krets för en fickräknare av en japansk kalkylatortillverkare. Hur det gick

sedan vet vi, kretsen blev 1971 den första mikroprocessorn Intel 4004, en 4-bitars mikroprocessor, och starten på mikroprocessorepoken.

## Standardkretsar

Standardkretsar är exempelvis mikroprocessorer, signalprocessorer, minneskretsar, kretsar ur logikfamiljer såsom grindar, räknare, register osv. Dessa kretsar är bestämda till sin funktion. Till standardkretsarna bör man också räkna de programmerbara kretsarna (PLD), som saluförs på samma sätt som de förstnämnda ”vanliga” standardkretsarna, samma krets ofta av flera olika fabrikanter, s.k. alternativfabrikanter (eng. *second source*). De programmerbara kretsarna skiljer sig dock från de ”vanliga” standardkretsarna, genom att de inte är bestämda till sin funktion, vilken bestäms först vid programmeringen. En programmerbar krets kan realisera funktionen hos många olika ”vanliga” standardkretsar. En möjlig klassificering av standardkretsar visas nedan.



Figur 1.14 Klassificering av standardkretsar.

## Funktionsbestämda kretsar

Funktionen hos dessa kretsar är helt bestämd. Viss mindre förändring av funktionen genom inställning av parametrar kan dock ibland vara möjlig, som t.ex i en UART (*Universal Asynchronous Receiver and Transmitter*), en kommunikationskrets, där parametrar som baudrate (bit/s), ordlängd, antal stoppbitar o.s.v. är programmerbara. Sådana kretsar kan kallas *parameterprogrammerbara*. En mikroprocessor kan exekvera olika program och kan därför kanske inte tyckas vara funktionsbestämd, men det är den uppenbart, dess funktion är att vara mikroprocessor och exekvera program.

## Icke funktionsbestämda kretsar – strukturprogrammerbara

Programmerbara logiska kretsar (eng. *Programmable Logic Device, PLD*) är icke funktionsbestämda, strukturen i kretsen måste först programmeras innan funktionen blir bestämd och kretsen kan realisera det önskade beteendet. Generella PLD är inte konstruerade för en viss tillämpning, utan blir en tillämpningsspecifik krets först efter programmeringen, då strukturen blir bestämd, kretsen blir då en ASIC. Vissa PLD är konstruerade för speciella tillämpningar, såsom t.ex. för bussanslutning till mikroprocessorer, och dessa PLD brukar då benämñas *tillämpningsspecifika PLD* (eng. *Application Specific PLD*), *ASPLD*. PLD är en viktig klass av kretsar. De används som tillämpningsspecifika kretsar i produkter och för prototypstestning vid utveckling av ASIC. Dagens PLD är så komplexa att de medger möjlighet att lägga in mikroprocessorer och signalprocessorer i kapseln. I exempelvis en XILINX Virtex-II FPGA kan man programmera in en mikroprocessor, typ IBM PowerPC 405 32-bit RISC CPU, en DSP (Digital Signal Processor) eller andra dylika kretsar som man kan köpa av fabrikanten som färdiga s.k. IP-block (IP = Intellectual Property) och använda dessa kretsar till-sammans med de kretsar som man själv konstruerar. Ofta används ordet PLD endast för enklare typer av programmerbara kretsar med den struktur de hade i begynnelsen. För mer komplexa PLD bestående av ett antal av de ursprungliga PLD på samma chip med en programmerbar förbindningsmatris mellan dem, används benämningen *CPLD* (eng. *Complex PLD*). För en komplex PLD med en matrisstruktur av konfigurerbara logikblock med mellanliggande förbindningsledningar, används benämningen *FPGA* (eng. *Field Programmable Gate Array*).

# 1.3 Talsystem och koder

## Representation av tal i talsystem med olika baser

Det decimala talsystemet har *basen* (eng. *base, radix*) 10 och har tio siffer-symboler (siffror) 0, 1, 2, ..., 9, alltså lika många siffer-symboler som basen. Ett decimaltal representeras med en sekvens av dessa siffer-symboler. Det decimala talsystemet är ett positionssystem, innebärande att siffrans *vikt* (eng. *weight*) i talet bestäms av siffrans position i talet. Sålunda är det decimala talet 370,81 ett symboliskt skrivsätt för

$$3 \cdot 10^2 + 7 \cdot 10^1 + 0 \cdot 10^0 + 8 \cdot 10^{-1} + 1 \cdot 10^{-2}$$

En siffras vikt i ett decimaltal är som bekant en potens av basen 10.

Principerna ovan gäller för ett godtyckligt talsystem och för ett talsystem med basen  $b$  gäller alltså att det har  $b$  siffer-symboler 0, 1, 2, ...,  $(b-1)$  och att siffrornas vikt är en potenser av basen  $b$ . Notera att siffran omedelbart till vänster om kommat (punkten) alltid har vikten 1, är en entalssiffra, oavsett basen  $b$ , eftersom  $b^0 = 1$ .

Det *binära talsystemet* (eng. *binary number system*) har basen 2 och två siffer-symboler 0 och 1. Siffrornas vikt är potenser av 2. Exempelvis är det binära talet 1011.101 ett symboliskt skrivsätt för

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

Räknar vi ut denna summa finner vi att den är lika med 11,625. Vi konstat-erar således att det binära talet 1011.101 är lika med decimala talet 11,625, vilket kan uttryckas som

$$1011.101_2 = 11,625_{10}$$

där index anger talets bas (vi använder ”punkt” i stället för ”komma” vid andra baser än 10).

I det binära talsystemet har alltså siffrorna till vänster om binärpunkten (heltalssiffrorna) vikterna 1, 2, 4, 8, 16, 32, 64, ..., och siffrorna till höger om binärpunkten (bråktalssiffrorna) vikterna  $1/2, 1/4, 1/8, 1/16, \dots$ .

I ett tal benämns siffran som har den största vikten, *mest signifikant siffra* (eng. *Most Significant Digit, MSD*) och siffran som har den minsta vikten,

minst signifikant siffra (eng. *Least Significant Digit, LSD*). För 4-bitarsalet 1011 (*bit = binary digit*) är alltså

MSD LSD

↓ ↓

1 0 1 1

När det gäller binära siffror, som i detta fall, används också benämningen *MSB (Most Significant Bit)* och *LSB (Least Significant Bit)*.

Binära talvariabler brukar namnges så att det framgår vilken bit som är mest respektive minst signifikant. Normalt används längsta index för den minst signifikanta siffran, t.ex. kan en 4-bitars talvariabel betecknas  $d=(d_3, d_2, d_1, d_0)$ . Ibland används vikterna som index,  $d=(d_8, d_4, d_2, d_1)$ . I vissa fall, exempelvis i databöcker för digitala kretsar, låter man bokstaven A beteckna den minst signifikanta siffran, bokstaven B den näst minst signifikanta siffran o.s.v. Man kan se en 4-bitars insignalvariabel betecknad  $I=(D, C, B, A)$  och en tillståndsvariabel betecknad  $Q=(Q_D, Q_C, Q_B, Q_A)$ .

Det finns två andra talsystem som används i digitaltekniken, det *hexadecimala talsystemet* och det *oktala talsystemet*, och då speciellt det förstnämnda. Den huvudsakliga användningen av dessa talsystem är för att bekvämt och kompakt representera långa binärtal, som är svåröverskådliga och svårhanterliga. Orsaken till att just dessa två talsystem används för representation av binärtal är, som vi skall se längre fram, att det är lätt att omvandla ett binärtal till ett hexadecimalt eller ett oktalt tal och tvärtom.

Det *hexadecimala talsystemet* har *basen 16* och 16 sifversymboler och sifferornas vikter är potenser av 16. En komplikation här är att det behövs sex ytterligare sifversymboler utöver symbolerna 0, 1, 2, ..., 9. Om sifversymbolerna 10, 11, 12 o.s.v. skulle användas, kunde missförstånd lätt uppstå vid tolkningen av talen. T.ex. skulle talet 12 kunna tolkas som symbolen 12 eller som symbolerna 1 och 2. För att slippa markera med parenteser e.d. vad som avses, betecknas sifversymbolerna 10, 11, 12, ..., 15 med bokstäverna A, B, C, ..., F. Exempel på ett hexadecimalt tal är B9.F för vilket alltså gäller

$$B9.F_{16} = 11 \cdot 16^1 + 9 \cdot 16^0 + 15 \cdot 16^{-1} = 185,9375_{10}$$

Det *oktala talsystemet* har basen 8 och 8 sifversymboler 0, 1, 2, ..., 7 och siffrornas vikter är potenser av 8.

Exempel på ett oktalt tal är 125.4 för vilket gäller

$$125.4_8 = 1 \cdot 8^2 + 2 \cdot 8^1 + 5 \cdot 8^0 + 4 \cdot 8^{-1} = 85,5_{10}$$

Vi kan nu sammanfatta och konstatera att för ett tal  $N=x_p x_{p-1} \dots x_1 x_0 x_{-1} \dots x_{-k}$  i ett talsystem med basen  $b$  gäller att

$$\begin{aligned} N = & x_p \cdot b^p + x_{p-1} \cdot b^{p-1} + \dots + x_1 \cdot b^1 + x_0 \cdot b^0 \\ & + x_{-1} \cdot b^{-1} + x_{-2} \cdot b^{-2} + \dots + x_{-k} \cdot b^{-k} \end{aligned} \quad (1.1)$$

där talets siffror  $x_i$  kan anta  $b$  olika värden 0, 1, 2, ..., ( $b-1$ ).

Det råder ett enkelt samband mellan ett talsystems bas och antalet erforderliga positioner (siffror) för att representera ett visst talområde. Antag att de  $M$  heltalen 0, 1, 2, ..., ( $M-1$ ) skall representeras i ett talsystem med basen  $b$ . Med  $n$  siffror och basen  $b$  kan bildas  $b \cdot b \cdot \dots \cdot b$  ( $n$  st) =  $b^n$  olika tal, ty första siffran kan väljas på  $b$  olika sätt, andra siffran på  $b$  olika sätt osv. För att de  $M$  heltalen skall kunna representeras med  $n$  siffror måste alltså gälla att  $b^n \geq M$ . Med exempelvis fyra siffror, dvs  $n = 4$ , kan i det binära talsystemet bildas  $2 \cdot 2 \cdot 2 \cdot 2 = 2^4 = 16$  olika tal, 0000, 0001, 0010, ..., 1111, motsvarande heltalen 0, 1, 2, ..., 15. Uppenbart ju mindre basen är desto fler siffror åtgår det för att representera ett visst talområde.

Tabell 1.1: Heltalen 0 t.o.m. 15 i några olika baser.

bas	2	3	4	5	8	10	16
	0000	000	00	00	00	00	0
	0001	001	01	01	01	01	1
	0010	002	02	02	02	02	2
	0011	010	03	03	03	03	3
	0100	011	10	04	04	04	4
	0101	012	11	10	05	05	5
	0110	020	12	11	06	06	6
	0111	021	13	12	07	07	7
	1000	022	20	13	10	08	8
	1001	100	21	14	11	09	9
	1010	101	22	20	12	10	A
	1011	102	23	21	13	11	B
	1100	110	30	22	14	12	C
	1101	111	31	23	15	13	D
	1110	112	32	24	16	14	E
	1111	120	33	30	17	15	F

## Omvandling mellan olika baser

### Omvandling mellan binära och hexadecimala talsystemen

Denna omvandling är speciellt enkel eftersom det hexadecimala talsystems bas 16 är en potens av det binära talsystemets bas,  $2^4=16$ . Som framgår av tabell 1.1 ovan så motsvaras de 16 hexadecimala symbolerna 0-F av samtliga 4-bitars binärtal 0000 - 1111. Omvandling av ett binärtal till ett hexadecimalt tal utförs helt enkelt genom att det binära talet delas upp i grupper om fyra bitar med början från binärpunkten, varvid eventuell komplettering med nollor tänkes utförd i början och slutet av talet, så att även de yttersta grupperna får fyra bitar. Exempelvis omvandlas binärtal 1110110110.001101 enligt

binärtal:                11      1011    0110 . 0011    01

hexadecimaltal:        3        B        6     .    3        4

Sålunda är  $1110110110.001101_2 = 3B6.34_{16}$

Omvandlingen av ett hexadecimaltal till ett binärtal sker självklart genom att varje hexadecimal siffra skrivs som en grupp om fyra bitar.

### Omvandling mellan binära och oktala talsystemen

Omvandling mellan binära och oktala talsystemen sker på samma sätt som mellan binära och hexadecimala talsystemen, bara med skillnaden att grupperna skall innehålla tre bitar i stället för fyra, beroende på att det oktala talsystems bas 8 också är en potens av det binära talsystemets bas,  $2^3=8$  och att de 8 oktala sifversymbolerna 0-7 motsvaras av samtliga 3-bitars tal 000 - 111. Exempelvis omvandlas binärtal 1010111101.11010 enligt

binärtal:                1    010    111    101.    110    10

oktaltal:                1    2      7        5.    6      4

Sålunda är  $1010111101.11010_2 = 1275.64_8$

## Omvandling mellan decimala och binära talsystemen

### *Omvandling av ett binärtal till ett decimalt tal*

Omvandling av ett binärtal till ett decimaltal kan ske genom **summering** av positionsvikterna för de positioner i binärtal som har värdet 1. Exempelvis omvandlas binärtal 110.101 enligt

$$\begin{aligned} 110.101_2 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 4 + 2 + 0 + 0,5 + 0 + 0,125 \\ &= 6,625_{10} \end{aligned}$$

För längre binära heltal så kan det vara lämpligare att omvandla via det hexadecimala talet i stället för som ovan genom att summa 2-potenserna (t.ex. om man för hand eller genom huvudräkning vill ta fram det decimala talet). Orsaken till att det är enklare via hexadecimala talet är att man då ”greppar” över fyra bitar i taget. I ett hexadecimalt hettal är ju positionsvikterna från LSD till MSD: 1, 16, 256, 4096, 65536 o.s.v. Exempelvis kan 14-bitars talet 10000110111100 omvandlas enligt

$$\begin{aligned} 10000110111100_2 &= 21BC_{16} \\ &= 2 \cdot 4096 + 1 \cdot 256 + 11 \cdot 16 + 12 \cdot 1 \\ &= 8636_{10} \end{aligned}$$

Liksom i det decimala talsystemet kan det i det binära talsystemet vara bekvämt att använda kilo (k), Mega (M), Giga (G) o.s.v. Speciellt ser man dessa prefix i datorsammanhang vid angivande av minnesstorlekar. I binära talsystemet gäller att  $2^{10} = 1024 = 1\text{kilo} = 1\text{k}$  och  $2^{20} = 2^{10} \cdot 2^{10} = 1\text{k} \cdot 1\text{k} = 1\text{Mega} = 1\text{M}$  samt  $2^{30} = 2^{10} \cdot 2^{20} = 1\text{k} \cdot 1\text{M} = 1\text{Giga} = 1\text{G}$ . Dessa prefix gör det bl.a. lättare att få en uppfattning om storleken av stora 2-potenser. Exempelvis kan  $2^{16}$  skrivas  $2^6 \cdot 2^{10} = 64\text{k}$  och  $2^{24}$  skrivas som  $2^4 \cdot 2^{20} = 16\text{M}$ . Man bör definitivt kunna 2-potenserna upp till  $2^{10}$  utantill och i appendix 1 finns en tabell med 2-potenser.

### *Omvandling av ett decimaltal till ett binärtal*

Omvandling av ett decimaltal till ett binärtal sker genom omvandling av heltalsdelen och bråkdelen var för sig. Ett sätt är att man bestämmer vilka positionsvikter som ska ingå i binärtal, t.ex. för heltalet med hjälp av en tabell med 2-potenser enligt appendix 1. Man börjar med den mest signifi-

kanta positionen och bestämmer först vilken största positionsvikt som är mindre eller lika med decimaltalet som skall omvandlas. Denna positionsvikt subtraheras från decimaltalet och om resten blir större än noll upprepas proceduren. Omvandlingen av heltalsdelen leder till ett slut efter ett ändligt antal steg, vilket ändå inte omvandlingen av bråkdelen alltid gör (bråkdelen kan inte alltid skrivas som en summa av 2-potenser,  $2^{-n}$ ), utan får avslutas när önskad noggrannhet uppnåtts.

Ett annat sätt att göra omvandlingen är att använda en *generell metod*, som inte bara gäller för omvandling av ett decimaltal till ett binärtal, utan för omvandling mellan vilka talsystem som helst där talen skrivs som en summa positionsvikter. Metoden innebär bestämning av koefficienterna  $x_p, x_{p-1}, \dots, x_0$  i heltalsdelen och koefficienterna  $x_{-1}, x_{-2}, \dots, x_{-k}$  i bråkdelens ekvation (1.1) ovan. Låt oss studera metoden och börjar då med att dela upp  $N$  i en heltalsdel, betecknad  $N_0$  och en bråkdel, betecknad  $N_{-1}$ .

$$N = N_0 + N_{-1} \quad (1.2)$$

$$\begin{aligned} N_0 &= x_p \cdot b^p + x_{p-1} \cdot b^{p-1} + \dots + x_1 \cdot b^1 + x_0 \cdot b^0 \\ &= (x_p \cdot b^{p-1} + x_{p-1} \cdot b^{p-2} + \dots + x_1) \cdot b + x_0 \\ &= N_1 \cdot b + x_0 \end{aligned} \quad (1.3)$$

$$\begin{aligned} N_{-1} &= x_{-1} \cdot b^{-1} + x_{-2} \cdot b^{-2} + \dots + x_{-k} \cdot b^{-k} + \dots \\ &= b^{-1} \cdot (x_{-1} + x_{-2} \cdot b^{-1} + \dots + x_{-k} \cdot b^{-k+1} + \dots) \\ &= b^{-1} \cdot (x_{-1} + N_{-2}) \end{aligned} \quad (1.4)$$

Av ekvation (1.3) för heltaldelen  $N_0$  ovan framgår att division av  $N_0$  med basen  $b$  ger kvoten  $N_1$  och resten  $x_0$ , dvs resten blir lika med den minst signifikanta siffran. Kvoten  $N_1$  kan sedan skrivas enligt samma princip som ekvation (1.3) och division av  $N_1$  med basen  $b$  ger kvoten  $N_2$  och resten  $x_1$ . Siffrorna  $x_p, x_{p-1}, \dots, x_0$  i heltalsdelen erhålls alltså genom successiva divisioner med basen  $b$  enligt uppställningen nedan.

Av ekvation (1.4) för bråkdelen  $N_{-1}$  ovan framgår att multiplikation av bråkdelen med basen  $b$  ger en heltalsdel  $x_{-1}$  och en bråkdel  $N_{-2}$ , dvs heltalsdelen

blir lika med den mest signifikanta siffran. Bråkdelen  $N_{-2}$  kan sedan skrivas enligt samma princip som ekvation (1.4) och multiplikation av  $N_{-2}$  med basen  $b$  ger heltalsdelen  $x_{-2}$  och bråkdelen  $N_{-3}$ . Siffrorna  $x_{-1}, x_{-2}, \dots, x_{-k}$  i bråkdelen erhålls alltså genom successiva multiplikationer med basen  $b$  enligt uppställningen nedan.

Tabell 1.2: Schema för omvandling av heltalsdel och bråkdel.

Heltalsdel	Bråkdel
kvot      rest	heltal      bråkdel
$N_0/b = N_1 + x_0$	$N_{-1} \cdot b = x_{-1} + N_{-2}$
$N_1/b = N_2 + x_1$	$N_{-2} \cdot b = x_{-2} + N_{-3}$
.	.
.	.
.	.
$N_p/b = 0 + x_p$	$N_{-k} \cdot b = x_{-k} + N_{-k-1}$
	.
	.

Exempelvis kan decimaltalet 201,732 omvandlas till binärtal som följer.

Division av heltalsdelen 201 med 2 och multiplikation av bråkdelen 0,732 med 2 enligt schemat i tabell 1.2 ovan ger

Heltalsdel: $N_0=201_{10}$	Bråkdel: $N_{-1}=0,732_{10}$
kvot      rest	heltal      bråkdel
$201/2 = 100 + 1$	$0,732 \cdot 2 = 1 + 0,464$
$100/2 = 50 + 0$	$0,464 \cdot 2 = 0 + 0,928$
$50/2 = 25 + 0$	$0,928 \cdot 2 = 1 + 0,856$
$25/2 = 12 + 1$	$0,856 \cdot 2 = 1 + 0,712$
$12/2 = 6 + 0$	$0,712 \cdot 2 = 1 + 0,424$
$6/2 = 3 + 0$	$0,424 \cdot 2 = 0 + 0,848$
$3/2 = 1 + 1$	$0,848 \cdot 2 = 1 + 0,696$
$1/2 = 0 + 1$ (MSD)	.
	.

Resultat:  $201,732_{10} = 11001001.1011101.._2$

## Elementär aritmetik för binära, hexadecimala och oktala heltal utan tecken

### Addition

Addition utförs på samma sätt som i det decimala talsystemet. I en dator adderas två operander i taget och nedan visas som exempel addition av de två decimala talen 154 och 94 i de binära, hexadecimala och oktala talsystemen.

Decimaltal	Binärtal	Hexadecimaltal	Oktaltal
$\begin{array}{r} 1 \\ 154 \\ + 94 \\ \hline 248 \end{array}$	$\begin{array}{r} 1 \\ 0011010 \\ + 01011110 \\ \hline 11111000 \end{array}$	$\begin{array}{r} 1 \\ 9A \\ + 5E \\ \hline F8 \end{array}$	$\begin{array}{r} 1 \\ 232 \\ + 136 \\ \hline 370 \end{array}$

I det binära talsystemet inträffar *minnessiffra* (eng. *carry*) för  $1+1=10_2$  och för  $1+1+1=11_2$  och behandlas på samma sätt som i det decimala talsystemet. I det hexadecimala och oktala talsystemet inträffar minnessiffra då summan blir  $\geq 16_{10}$  respektive  $\geq 8_{10}$ . I additionen av hexadecimala talen ovan så är  $A_{16}+E_{16} = 10_{10}+14_{10} = 24_{10} = 18_{16}$ . I additionen av de oktala talen så är  $2_8+6_8 = 8_{10} = 10_8$ .

### Subtraktion

Decimaltal	Binärtal	Hexadecimaltal	Oktaltal
$\begin{array}{r} 10 \\ 248 \\ - 154 \\ \hline 94 \end{array}$	$\begin{array}{r} 2_{10} \\ \downarrow \\ 10101010 \\ - 10011010 \\ \hline 01011110 \end{array}$	$\begin{array}{r} 16_{10} \\ \downarrow \\ F8 \\ - 9A \\ \hline 5E \end{array}$	$\begin{array}{r} 8_{10} \\ \downarrow \\ 370 \\ - 232 \\ \hline 136 \end{array}$

Subtraktion kan också utföras på samma sätt som i det decimala talsystemet. *Lånesiffran* (eng. *borrow*) behandlas som vid subtraktion av decimala tal. I det decimala talsystemet med basen 10 lånar man från positionen till

vänster om den aktuella positionen en etta, som då blir värd 10 gånger så mycket då den flyttas ett steg åt höger, dvs  $10_{10}$ . I det binära talsystemet med basen 2 lånar man också från positionen till vänster en etta, som då blir värd 2 gånger så mycket då den flyttas ett steg åt höger, dvs  $2_{10}$  eller  $10_2$ . Oavsett talets bas  $b$ , så lånar man alltså alltid  $10_b$ . Uttryckt i det decimala talsystemets bas så lånar man  $2_{10}$ ,  $16_{10}$  och  $8_{10}$  i det binära, hexadecimala respektive oktala talsystemet.

### Multiplikation av binära tal

Nedan visas multiplikation av binärtalen  $1101 (= 13_{10})$  och  $1001 (= 9_{10})$ , som ger produkten  $1110101 (= 117_{10})$ .

$$\begin{array}{r}
 1101 & \text{multiplikand} \\
 \cdot 1001 & \text{multiplikator} \\
 \hline
 1101 \\
 0000 \\
 0000 \\
 + 1101 \\
 \hline
 1110101 & \text{produkt}
 \end{array}$$

Liksom i det decimala talsystemet utförs multiplikation som ett antal additioner av multiplikanden skiftad åt vänster. I det binära talsystemet är det speciellt enkelt, eftersom siffrorna i multiplikatorn bara har värdet 0 eller 1. Sålunda om multiplikatorsiffran = 1 skall multiplikanden adderas, om multiplikatorsiffran = 0 skall multiplikanden inte adderas.

### Division av binära tal

$$\begin{array}{r}
 & 1100 & \text{kvot} \\
 & \overline{101} & \\
 \text{divisor} & \underline{101} & \\
 & \overline{1010} & \\
 & - 101 & \\
 & \overline{0010} & \\
 & - 101 & \\
 & \overline{00100} & \text{rest}
 \end{array}$$

Ovan visas division av binärtalen 1000000 (=  $64_{10}$ ) och 101 (=  $5_{10}$ ). 1000000:101, som resulterar i kvoten  $1100 (= 12_{10})$  och resten  $100 (= 4_{10})$ . Division görs som i det decimala talsystemet som ett antal subtraktioner av divisorn från dividenden. I det binära talsystemet är det enklare, vid varje subtraktion *kan* divisorn antingen subtraheras högst en gång (ger kvotsiffran = 1) eller *kan inte* divisorn subtraheras (ger kvotsiffran = 0). Algoritmen för subtraktion är betydligt mer komplicerad än algoritmen för multiplikation. Problemet ligger i att avgöra om divisorn kan eller inte kan subtraheras. Ett sätt är att alltid utföra subtraktionerna och om subtraktionen resulterar i ett lån i positionen till vänster om divisorn, så kan inte subtraktionen utföras, kvotsiffran sätts till 0 och dividenden återställs genom att divisorn adderas. Om i stället subtraktionen inte resulterar i ett lån, kan subtraktionen utföras och kvotsiffran sätts till 1. Denna algoritm kallas *division med återställning* (eng. *restoring division*)

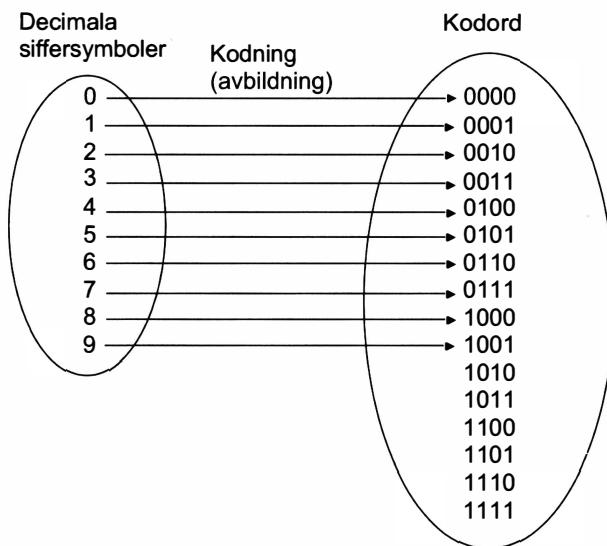
Återvänder vi till subtraktion av binära tal så *kan* den alltså utföras enligt den konventionella metoden ovan, men i digitala kretsar, datorer, brukar subtraktion inte göras på detta sätt, utan faktiskt i stället som en addition!, addition av det s.k. *2-komplementet*, som behandlas i ett senare kapitel i samband med genomgång av en aritmetisk enhet. Det är onekligen en fördel att samma digitala krets, en additionsenhet, kan användas för både addition och subtraktion, och inte nog med det, multiplikation görs som ett antal additioner och division som ett antal subtraktioner och sålunda kan alla fyra räknesätten utföras med en additionsenhet.

## Numeriska och alfanumeriska koder

### Koder för de decimala siffrorna 0–9

Decimala siffror som skall inmatas till displayrar, typ 7-segment e.d. för visning, eller som skall utmatas från ett numeriskt tangentbord, måste *kodas* binärt. Eftersom 10 decimala siffror 0, 1, 2, ..., 9 skall kodas binärt så måste det binära *ordet* (eng. *word*) ha en *ordlängd* (eng. *word length*)  $n$ , sådan att  $2^n \geq 10$ . Här används begreppet *ord*, beroende på att det inte nödvändigtvis behöver vara ett tal, utan kan vara en godtycklig sekvens av binära symboler. I anslutning till begreppet *ord* kan det nämnas att ett binärt ord med längden 8 bitar, på engelska brukar benämñas *byte*. För att koda de 10 decimala siffrorna krävs en ordlängd av minst 4, eftersom  $2^4 > 10 > 2^3$ . Det finns  $2^4 = 16$  binära ord med ordlängden 4 och en kod för de 10 decimala

siffrorna kan fås genom ett val av 10 av dessa 16 ord. Detta kan faktiskt göras på  $29059430400 (= 16!/6!)$  olika sätt.



Figur 1.15 Kodningen av de decimala sifversymbolerna 0–9 i BCD-kod.

Det naturligaste sättet att koda de 10 decimala sifversymbolerna är självklart att i mängden av 4-bitars ord välja de till siffrorna motsvarande binära talen enligt figuren ovan. Denna kod benämnes *BCD-kod* (eng. *Binary Coded Decimal*). (Ibland kan man se denna kod benämnd *NBCD-kod*, *Natural BCD-kod* för markering att det är det naturligaste sättet att koda de binära siffrorna).

Nedan visas som exempel det decimala talet 146 kodat, dels i BCD med 3 siffror, dels som ett 8-bitars binärtal.

Decimaltal:      1          4          6

BCD:                0001    0100    0110

8-bitars binärtal: 10010010

I BCD-koden har ju de binära siffrorna, från vänster räknat, vikterna 8, 4, 2, 1. BCD-koden kan benämnes 8421-kod efter positonsvikterna. I tabell 1.3 nedan visas några andra koder med viktade positioner.

Tabell 1.3: Några koder med viktade positioner för de decimala siffrorna 0–9.

Decimal siffra <b>d</b>	<b>Kodord <math>y = (y_3, y_2, y_1, y_0)</math></b>			
	Positionsvikter: $v_4, v_3, v_2, v_1$			
	BCD 8421	5421	5211	7421
0	0000	0000	0000	0000
1	0001	0001	0001	0001
2	0010	0010	0100	0010
3	0011	0011	0110	0011
4	0100	0100	0111	0100
5	0101	1000	1000	0101
6	0110	1001	1001	0110
7	0111	1010	1011	1000
8	1000	1011	1110	1001
9	1001	1100	1111	1010

Sambandet mellan den decimala siffran  $d$  och kodordet  $y = (y_3, y_2, y_1, y_0)$  med positionsvikterna  $v_3, v_2, v_1, v_0$ , i koderna ovan är alltså

$$d = y_3 \cdot v_3 + y_2 \cdot v_2 + y_1 \cdot v_1 + y_0 \cdot v_0$$

För exempelvis kodordet 1011 i 5421-koden erhålls den decimala siffran  $d$  som

$$d = 1 \cdot 5 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 8$$

## Alfanumeriska koder

Koder som kodar både de decimala sifversymbolerna och alfabetets bokstäver benämnes *alfanumeriska koder* (eng. *alphanumeric code*). Den vanligaste alfanumeriska koden för överföring av data mellan digitala system är *ASCII-koden* (*American Standard Code for Information Interchange*). Denna kod visas i appendix 2. ASCII-koden innehåller även kodord för diverse specialsymboler och styrsymboler, samtliga symboler som kodas brukar gemensamt benämnes *tecken* (eng. *character*).

Vissa fundamentala kodord i ASCII-koden bör man lära sig utantill. Siffrorna och alfabetets bokstäver är kodade i grupper på ett logiskt sätt, så att om man kan koden för första siffran respektive bokstaven så får man lätt fram koderna för de andra siffrorna och bokstäverna, eftersom de är kodade i ordning.

<b>Tecken</b>	<b>ASCII-kod (decimal)</b>
siffran '0'	48
stora bokstaven 'A'	65
lilla bokstaven 'a'	97
'blanksteg' (SP, space)	32
'ny rad' (LF, Line Feed)	10
'vagnretur' (CR, Carriage return)	13      (återgång till radbörjan)

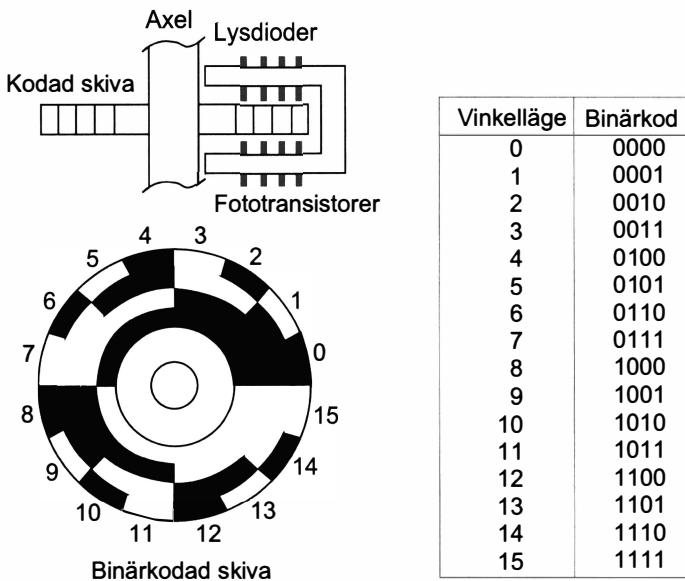
ASCII-koden till siffran '0' är 48. Om man kan denna kod så får man lätt koden för övriga siffror genom att addera siffran till 48, eftersom siffrorna är kodade efter varandra i ordning. Bokstäverna är kodade på samma sätt. ASCII-koden för 'A', den första stora bokstaven i alfabetet, är 65. Kodorden för övriga stora bokstäver följer sedan i ordning. ASCII-koden för 'a', den första lilla bokstaven, är 97, dvs avståndet mellan bokstäverna 'a' och 'A' är 32. Exempelvis kan alla små bokstäver göras om till stora bokstäver genom subtraktion av 32 från ASCII-koden och vice versa genom addition av 32.

ASCII-koden är en 7-bitarskod, innebärande att det är möjligt att koda totalt  $2^7=128$  tecken. En åtonde bit kan användas för paritetskontroll, se kapitel 4, Paritetskrets, eller kan användas för utökning av koden att även inkludera kodord för grafiska tecken.

## Gray-kod

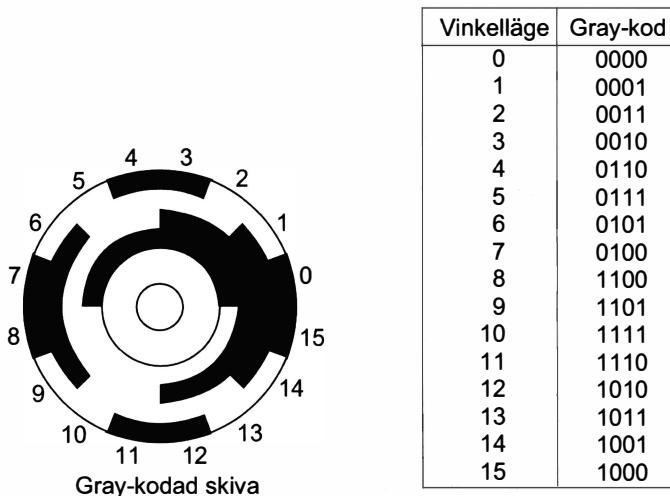
Antag att vinkelläget hos en axel skall indikeras i digital form. Axeln kan då förses med en kodad skiva med genomskinliga (vita) och icke-genomskinliga fält (svarta) enligt figur nedan. Skivan roterar med axeln. I radiell led sitter en rad lysdioder och fototransistorer, fast monterad i förhållande till skivan. Lysdioderna lyser på skivan och fototransistorerna detekterar ljus eller icke-ljus (mörker). Antar vi att fototransistorn ger utsignalen 1 för ljus och utsignalen 0 för icke-ljus, så får alltså vinkelläget i en binärkod. I detta

fall med 4 lysdioder/fototransistorer, en 4-bitars binärkod, som medger indikering av  $2^4 = 16$  olika vinkellägen, markerade 0, 1, ..., 15 längs skivans periferi. Kodningen av skivan är gjord i vanlig binärkod (MSB närmast axeln).



Figur 1.16 Digital vinkelgivare kodad i vanlig binärkod.

Antag att skivan har ett sådant läge att gränsen mellan sektorerna 7 (0111) och 8 (1000) ligger mitt för fototransistorerna. Dessa kan då i bästa fall indikera 0111 eller 1000, dvs. något av de korrekta lägena. I värsta fall kan de lika gärna detektera 1111 eller 0000, båda grovt felaktiga vinkelangivelser. Orsaken till att detta kan inträffa är att de två *intilliggande* vinkellägena 7 och 8 är kodade så att kodorden skiljer sig i mer än en position. I detta fall skiljer sig kodorden 0111 och 1000 i samtliga positioner! Problemet med de felaktiga vinkelangivelserna kan lösas genom att vinkellägena i stället kodas så att *kodorden till intilliggande vinkellägen endast skiljer sig i en position*, vilket fallet är med skivan i figuren nedan som är kodad i *Gray-kod*. Vi ser för denna skiva att om nu gränsläget mellan sektorerna 7 (0100) och 8 (1100) hamnar mitt för fototransistorerna, så kan de endast indikera 0100 eller 1100, dvs. något av de korrekta vinkellägena.



Figur 1.17 Digital vinkelgivare kodad i Gray-kod.

Gray-koden kallas också *reflekterad binärkod*, en benämning som kommer av sättet att konstruera koden. Gray-koden ovan har fyra bitar, men den kan konstrueras med ett godtyckligt antal bitar. Konstruktionsprincipen framgår av tabellen nedan, där Gray-koder med 1, 2, 3 och 4 bitar visas. Längst till vänster är den triviala Gray-koden med 1 bit. Gray-koden med 2 bitar konstrueras ur 1-bitsskoden genom att denna reflekteras (speglas) i den horisontella linjen och en andra bit tillfogas till vänster som 0 ovanför linjen och 1 nedan för linjen. Gray-koden med 3 bitar konstrueras på samma sätt ur 2-bitarskoden osv.

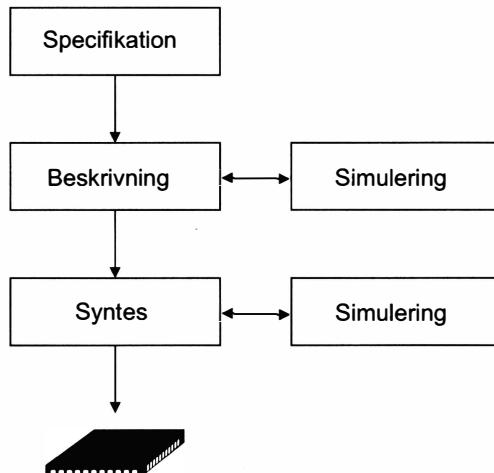
Tabell 1.4: Gray-kod med 1, 2, 3 och 4 bitar.

0	00	000	0000
1	<u>01</u>	001	0001
	11	011	0011
	10	<u>010</u>	0010
		110	0110
		111	0111
		101	0101
		100	<u>0100</u>
			1100
			1101
			1111
			1110
			1010
			1011
			1001
			1000

## 1.4 Realisering av en liten krets i PLD – en introduktion till VHDL

I denna sektion skall arbetsgången vid realisering av en krets i PLD belysas med ett litet exempel, som också blir en introduktion till hårdvarubeskrivande språket VHDL, som närmare behandlas i kapitel 9.

En kretsrealisering innehåller fyra huvudmoment – *specifikation, beskrivning, simulering och syntes* av kretsen.



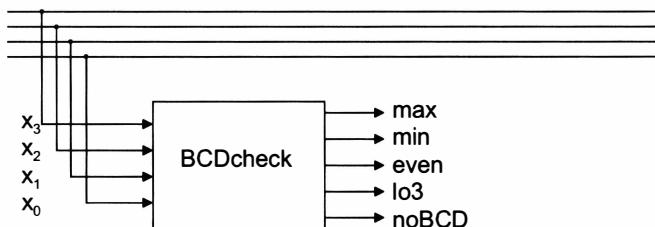
Figur 1.18 Huvudmoment vid realisering av en krets.

Efter specifikation av kretsen, beskriver man den formellt, simulerar, modifierar beskrivningen, simulerar, modifierar kanske specifikationen o.s.v., och när man är nöjd med simuleringen, börjar syntesen, som så småningom resulterar i en krets. Efter test av kretsen måste man kanske börja om från början igen med modifiering av specifikationen, beskrivningen o.s.v.

Låt oss nu gå över till realiseringen av kretsen.

## Specifikation

Decimala siffror i BCD-kod (se tabell 1.3) överförs på fyra dataledningar. En krets, benämnd *BCDcheck*, som skall indikera några olika egenskaper hos de decimala siffrorna, skall konstrueras. Kretsen skall ha fyra ingångar  $x_3, x_2, x_1, x_0$  på vilka den decimala siffran, betecknad  $x = (x_3, x_2, x_1, x_0)$ , inmatas. Kretsen skall ha fem utgångar *max*, *min*, *even*, *lo3*, *noBCD* som skall indikera siffrans egenskaper.



Figur 1.19 Blockschema för kretsen BCDcheck.

För utsignalerna skall gälla:

$\text{max} = 1$	om och endast om	$x = 9$ (största siffran)
$\text{min} = 1$	-"-	$x = 0$ (minsta siffran)
$\text{even} = 1$	-"-	$x$ är en jämn siffra
$\text{lo3} = 1$	-"-	$x < 3$ ( <i>lower than 3</i> ))
$\text{noBCD} = 1$	-"-	$x > 9$ (ej BCD)

## Formalisering av specifikationen

En specifikation av en krets kan vara mer eller mindre formell och kräva olika grader av formalisering, t.ex. strukturering i delblock, uppritning av funktionstabell, tillståndsdiagram, tillståndstabell, flödesplan, logiska uttryck. I detta fall med bara 16 insignal kombinationer kan det vara motiverat att rita en funktionstabell.

Tabell 1.5: Funktionstabell för kretsen BCDcheck.

Insignaler	Utsignaler				
	max	min	even	lo3	noBCD
x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> x <sub>0</sub>					
0 0 0 0	0	1	1	1	0
0 0 0 1	0	0	0	1	0
0 0 1 0	0	0	1	1	0
0 0 1 1	0	0	0	0	0
0 1 0 0	0	0	1	0	0
0 1 0 1	0	0	0	0	0
0 1 1 0	0	0	1	0	0
0 1 1 1	0	0	0	0	0
1 0 0 0	0	0	1	0	0
1 0 0 1	1	0	0	0	0
1 0 1 0	0	0	1	0	1
1 0 1 1	0	0	0	0	1
1 1 0 0	0	0	1	0	1
1 1 0 1	0	0	0	0	1
1 1 1 0	0	0	1	0	1
1 1 1 1	0	0	0	0	1

## Beskrivning

Efter att kretsspecifikationen fått en någorlunda formell utformning vidtar beskrivning av kretsen i det aktuella datorbaserade syntesverktyget. En del syntesverktyg har ett eget beskrivningsspråk med en speciell syntax. Syntesverktygens egna språk är normalt mycket effektiva med möjlighet till kompakta beskrivningar, men tyvärr inte standardiserade. Nackdelen med att beskriva sina kretsar i ett icke-standardiserat språk är att man blir bunden till syntesverktyget, eftersom ett byte av syntesverktyg är förenat med en hel del arbete, dels att lära sig ett nytt språk, dels att överföra sina tidigare kretsbeskrivningar till ett nytt språk, vilket kan behövas om man vill utnyttja redan beskrivna kretsmoduler i nya konstruktioner. Liksom när det gäller ”vanliga” programmeringsspråk där det utvecklats standarder, som möjliggör för en konstruktör som exempelvis skall konstruera ett mikrodatortbaserat system med någon enchipsdator att beskriva sitt program i ett standardiserat språk t.ex. språket C, så måste det självklart när det gäller beskrivningsspråk för kretsar vara önskvärt för en konstruktör att kunna

beskriva sina kretsar i ett standardiserat språk. Det finns ett språk för beskrivning av kretsar, som blev standard 1986. Språket heter *VHDL*, som står för **Very High Speed Integrated Circuit Hardware Description Language**. Det påminner om ”vanliga” programmeringsspråk, men är som namnet antyder avsett för *beskrivning* av kretsar, ett ”hårdvarubeskrivande språk”. Ytterligare bakgrund till VHDL ges i kapitel 9.

Låt oss direkt kasta oss över en beskrivning i VHDL av kretsen **BCDcheck**.

### Beskrivning i VHDL av kretsen **BCDcheck**

```
--BCDcheck
--Indikerar egenskaper hos siffror i BCD-kod
--2001-07-04
--Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity BCDcheck is
    port(x3, x2, x1, x0: in std_logic;
          max, min, even, lo3, noBCD: out std_logic);
end entity BCDcheck;

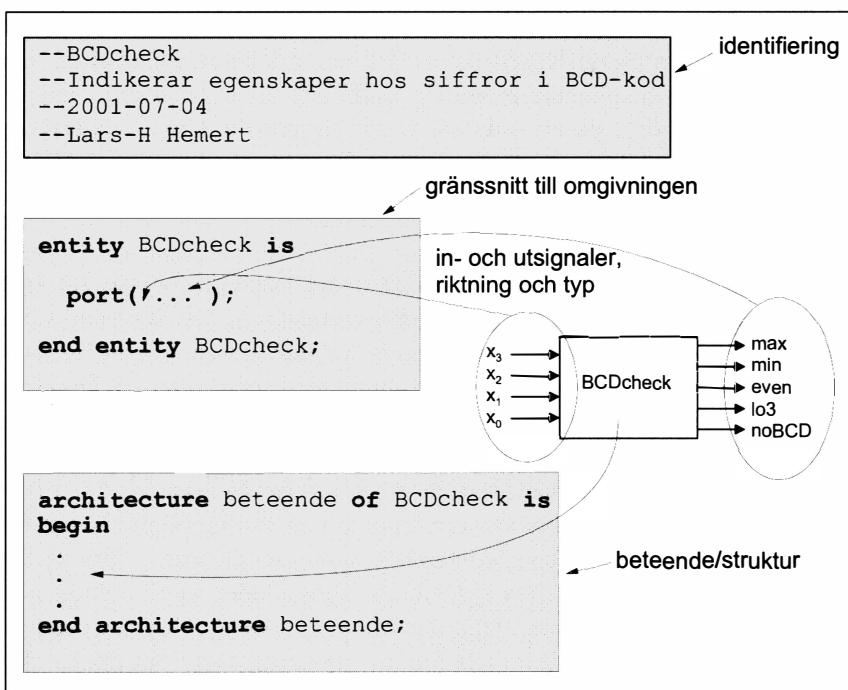
architecture beteende of BCDcheck is
begin
    max    <= x3 and not x2 and not x1 and x0;
    min    <= not x3 and not x2 and not x1 and not x0;
    even   <= not x0;
    lo3    <= (not x3 and not x2 and not x1 and not x0) or
              (not x3 and not x2 and not x1 and x0) or
              (not x3 and not x2 and x1 and not x0);
    noBCD <= (x3 and x2) or (x3 and not x2 and x1);
end architecture beteende;
```

### Kommentarer till beskrivningen

#### entity och architecture

En kretsbeskrivning i VHDL består av två delar, **entity** och **architecture**, som beskriver hur kretsen ”ser ut utifrån” respektive ”ser ut

inuti". – Det engelska ordet *entity* betyder på svenska *väsen, verklighet, existens* och i *entity* beskrivs kretsens *gränssnitt* (eng. *interface*) till omgivningen, omfattande *portar* (in- och utsignaler), *riktning* hos signalerna, samt *typ* av signaler. Beskrivningen i *entity* för kretsen BCDcheck är i princip en *formell beskrivning av blockschemat för kretsen* i figur 1.19 med de fyra insignalerna  $x_3, x_2, x_1$  och  $x_0$  och de fem utsignalerna *max*, *min*, *even*, *lo3* och *noBCD*, samtliga binära. En *entity* inleds med "**entity namn is**" och avslutas med "**end entity namn;**". Signalerna deklarerar i *entity* inom **port()**, där riktningen anges som **in** för insignal och **out** för utsignal och **inout** för dubbelriktad signal. Typen kan man själv definiera eller välja att använda en i VHDL fördefinierad standardiserad typ, som i detta fall typen **std\_logic**, definierad som ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'), där bl.a. 'Z' = *högohmig* och '-' = *don't care*. En annan fördefinierad typ är **bit**, definierad som ('0', '1').



Figur 1.20 Huvuddelar i en VHDL-beskrivning.

En **architecture** är alltid relaterad till en **entity** och beskriver hur kretsen ”ser ut inuti”. Denna beskrivning kan göras på många olika sätt, det kan vara en beskrivning av kretsens *struktur* (eng. *structure*) på låg abstraktionsnivå, det kan vara en beskrivning av kretsens *beteende* (eng. *behaviour*) på hög abstraktionsnivå, utan anknytning till någon speciell struktur. En **entity** kan ha flera olika **architecture**, beroende på vilken abstraktionsnivå man vill använda för att beskriva kretsen. I **architecture** för BCDcheck ovan har vi beskrivit beteendet i form av kretsens logiska funktion, utan att säga något om kretsens struktur, t.ex. realiseringen med grindar. En **architecture** inleds med ”**architecture namn of entity-namnet is**” och själva beskrivningen står mellan ”**begin**” och ”**end architecture namn;**”. Före begin kanstå några *deklarationer* (eng. *declaration*), som vi skall se längre fram. Namnet i **architecture** väljs lämpligt och i detta fall har valts namnet *beteende*, eftersom den beskriver kretsens beteende.

Reserverade ord i VHDL, dvs ord med speciell betydelse och som bara får användas på bestämda platser, kommer att skrivas med fet stil. *Namn, identifierare* (eng. *identifier*) som man själv definierar och som alltså ej får vara reserverade ord, skrivas med normal stil. De får innehålla bokstäver och sifferor, men måste börja på en bokstav. Inuti namnet får finnas understryckningar (\_), för uppdelning av namnet i flera delar för ökad tydlighet. VHDL skiljer inte på små och stora bokstäver. *Kommentarer* i VHDL inleds med två streck (- -) och varar till radens slut.

Beskrivningen i **architecture** mellan **begin** och **end** består för kretsen BCDcheck endast av fem *tilldelningssatser* (eng. *assignment statement*). Symbolen ( $<=$ ) är en *tilldelningssymbol*, och satsen ” $a <= b$ ” utläses ” $a$  tilldelas värdet av  $b$ ”. En sats avslutas med semikolon (:). Den som har lite erfarenhet av programmering i högnivåspråk, ser redan på detta stadium stora likheter mellan VHDL och högnivåprogrammering. *En skillnad* skall dock direkt påpekas, nämligen att de fem satserna i **architecture** ovan *inte genomlöps sekventiellt*, som fallet är i ett vanligt högnivåprogram, utan är *samtidiga satser* (eng. *concurrent statements*), som utförs så fort någon förändring inträffar hos signalerna i högerleddet av tilldelningssatserna. Detta är naturligt då ju VHDL beskriver hårdvara, och i vårt exempel påverkar insignalerna samtidigt alla fem utsignalerna. Det finns dock också satser i VHDL som genomlöps sekventiellt, nämligen satserna i en *process*, som vi kommer att studera längre fram i kapitel 9 och som speciellt används vid beskrivning av beteendet för sekvenskretsar.

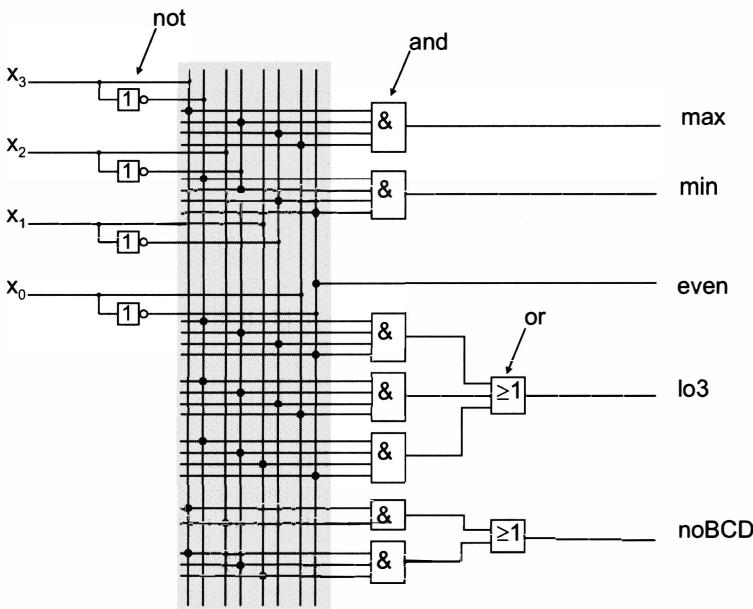
Uttrycken för utsignalerna i beteendebeskrivningen för BCDcheck är angivna med logiska operationer. I VHDL ingår de logiska operationerna **not**, **and**, **or**, **nand**, **nor**, **xor** och **xnor**. Logiska operationen **not** har högst prioritet, medan det bör observeras att de övriga logiska operationerna har samma prioritet, innebärande att parenteser måste användas i ett logiskt uttryck för att ange operationernas ordningsföljd.

Beskrivningen av de logiska uttrycken för utsignalerna har gjorts direkt utifrån funktionstabellen i tabell 1.5. I tabellen tolkas '1' som logiska konstanter 'Sann' (eng. *True*) och '0' som logiska konstanter 'Falsk' (eng. *False*). Exempelvis är utsignalen  $max = 'Sann'$  om och endast om insignalen ( $x_3 = 'Sann'$ ) OCH (ICKE ( $x_2 = 'Sann'$ )) OCH (ICKE ( $x_1 = 'Sann'$ )) OCH ( $x_0 = 'Sann'$ ). Logiska uttrycket för  $max$  omfattar alltså bara en s.k. *OCH-produkt*, eftersom kolumnen i funktionstabellen bara innehåller en etta. Logiska uttrycket för  $lo3$  innehåller tre OCH-produkter som sammanfogats med ELLER, eftersom funktionen innehåller tre ettor. Uttrycket för  $lo3$  är en s.k. *ELLER-summa av OCH-produkter*, som är det naturligaste sättet att skriva ett logiskt uttryck. Man skriver då alltså en OCH-produkt för varje etta och sammanfogar dem med ELLER. Det är också möjligt att i stället beskriva när utsignalen är 'Falsk', vilket är lämpligt då utsignalen innehåller färre nollor än ettor.

Förenkling av de logiska uttrycken är värdefull att kunna göra. Den utförs av syntesverktygen för att kunna minska antalet komponenter (grindar) som krävs för att realisera de logiska uttrycken i kretsen. I vårt fall har vi redan vid beskrivningen av utsignalen  $noBCD$  gjort en liten förenkling. Exempelvis ser vi i funktionstabellen att  $noBCD = 1$  om ( $x_3 = 1$ ) OCH ( $x_2 = 1$ ) oberoende av värdet hos  $x_1$  och  $x_0$ , som resulterar i den första OCH-produkten i uttrycket.

Beskrivningen i VHDL av kretsen görs med en vanlig texteditor till en fil *BCDcheck.vhd*. Fil tillägget (extensionen) *.vhd* används för att markera en VHDL-beskrivning.

En krets kan beskrivas på många olika abstraktionsnivåer. Beskrivningen ovan är på en låg nivå, mycket nära en tänkbar fysisk realisering. Logiska operationer realiseras i digitala kretsar med komponenter benämnda *grindar* (eng. *gate*), som vi kommer att studera i kapitel 2. I figuren nedan har ritats ett schema för kretsen uppbyggd med grindar direkt utgående från utsignalernas logiska uttryck i tilldelningssatserna i VHDL-beskrivningen. Grindarna har ritats med standardiserade IEC-simboler (IEC är akronym för International Electrotechnical Commission).



Figur 1.21 Kretsen BCDcheck realiseras med grindar.

Schemat ovan har ritats så att det skall antyda principen för hur strukturen hos en programmerbar krets, PLD, kan programmeras för att realisera en krets. I en PLD finns normalt ett antal OCH-ELLER-nät med struktur enligt ovan (normalt mycket större nät). Korsningspunkterna i den skuggade matrisen mellan OCH-grindarnas horisontella ingångsledningar och kretssens vertikala ingångsledningar är programmerbara som 'förbindelse' respektive 'icke-förbindelse', så att olika logiska uttryck kan realiseras. Förbindelse har ovan markerats med kraftiga punkter.

Ett annat sätt att beskriva kretsen i VHDL visas nedan. Denna beskrivning är på en lite högre abstraktionsnivå, nära beskrivningen av kretssens beteende i specifikationen.

### Kommentarer till beskrivningen

`std_logic_vector(3 downto 0)`

Insignalerna har deklarerats som en *vektor*. Detta kan vara lämpligt för en homogen mängd av signaler, t.ex. bussar. Syntaxen framgår av deklaratio-

nen nedan. I detta fall har vektorn deklarerats med högsta index längst till vänster, vilket är vanligast i digitaltekniken. Önskar man högsta index längst till höger används ”**0 to 3**”. En vektor kan tilldelas värdet. T.ex. kan en 8-bitars vektor tilldelas en bitsträng enligt  $v <= "01101011"$ , där bitsträngen skall omges med dubbla citattecken. Ett värde kan anges hexadecimalt med prefix ’X’ eller ’x’, oktalt med ’O’ eller ’o’ eller binärt med prefix ’B’ eller ’b’ (samma som inget prefix). Exempel x”7B”, o”153”. En bit av en vektor betecknas med bitnumret inom parentes t.ex.  $v(3)$  och kan tilldelas ett värde med  $v(3) <= '1'$ , där värdet skall omges med enkelt citattecken.

#### *when else*

Får bara användas i parallella tilldelningar. Syntaxen framgår av beskrivningen nedan, dock skall tilläggas att det är tillåtet att använda flera *when else* efter varandra, där sista *else* kan utelämnas (VHDL-93).

### Alternativ beskrivning av kretsen BCDcheck

```
-- BCDcheck2
-- Indikerar egenskaper hos sifferor i BCD-kod
-- 2001-07-04
-- Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity BCDcheck2 is
port(x : in std_logic_vector(3 downto 0);
      max, min, even, lo3, noBCD : out std_logic);
end entity BCDcheck2;

architecture beteende of BCDcheck2 is
begin
max    <= '1' when x = X"9" else '0';
min    <= '1' when x = X"0" else '0';
even   <= not x(0);
lo3    <= '1' when (x >= X"0") and (x < X"3") else '0';
noBCD <= '1' when (x > X"9") and (x <= X"F") else '0';
end architecture beteende;
```

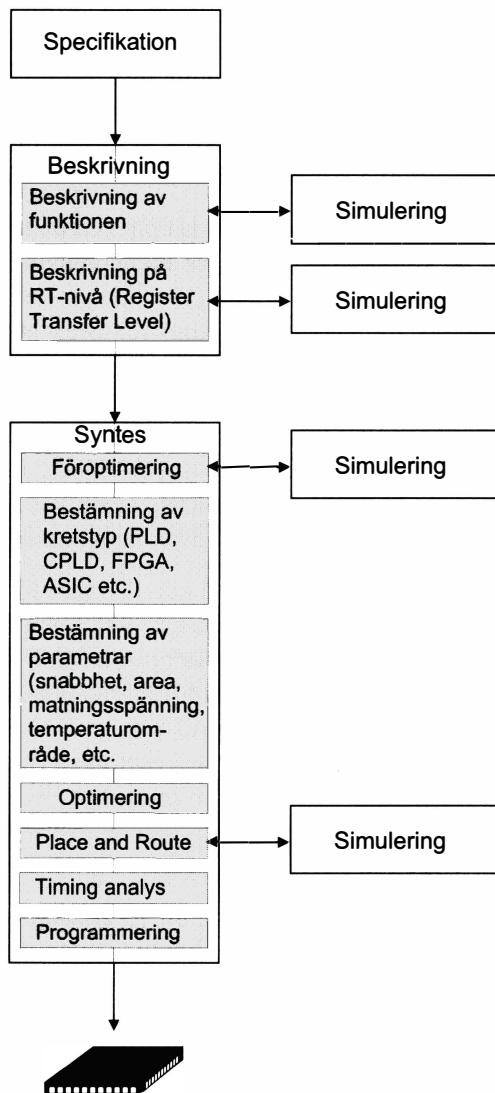
## Syntes

Vi har ovan sett exempel på hur vår lilla krets BCDcheck kan beskrivas på olika abstraktionsnivåer. Konstruktörer önskar naturligtvis få beskriva sin krets på så hög abstraktionsnivå som möjligt och sedan överläta åt syntesverktyget att göra resten. Fullt så enkelt är det inte.

VHDL skapades för att *beskriva* kretsar på ett standardiserat sätt. Simulatorer utvecklades tidigt och idag finns det mycket avancerade simulatorer som stöder hela VHDL. Simulering kan göras vid olika stadier i konstruktionsprocessen. Första simuleringen efter VHDL-beskrivningen på beteendenivå är till för att verifiera funktionen, att den är i enlighet med specifikationen. Simuleringen görs då med en modell av kretsen beskriven i VHDL, som kan vara på en mycket hög abstraktionsnivå, utan anknytning till den fysiska verkligheten. Ett problem vid simulering är tiden. Trots mycket snabba datorer kan simuleringar ta lång tid.

Syntes av kretsar beskrivna i VHDL är mycket mer komplicerat än simulering och det är mycket svårare att utveckla syntesverktyg än att utveckla simulatorer. Under 1990-talet har det utvecklats syntesverktyg för kretsar beskrivna i VHDL, en del är knutna till kretsfabrikanter, en del är oberoende och stöder olika kretsfabrikat. Syntesverktyg kommer inte att beskrivas här, det får man sätta sig in i med hjälp av manualer och praktiskt arbete.

Syntesverktygen stöder bara en delmängd av VHDL och kan skilja sig beträffande vilken delmängd de stöder. Det finns också begränsningar på hur hög abstraktionsnivå som får vara. Lämplig nivå är normalt *RT-nivå*, *RTL* (eng. *Register Transfer Level*). På denna nivå är kretsen uppdelad i register, multiplexrar, demultiplexrar, adderare, tillståndsmaskiner etc. I figuren nedan visas moment i kretsrealiseringen. Syntesen, översättningen av kretsbeskrivningen i VHDL till primitiva komponenter, grindar, vippor etc., på chippet, motsvarar översättningen av ett program från högnivåspråk till maskinspråk, kompileringen, vid programutveckling och i syntesverktyg används ofta begreppen *Compiler* och *Compilation*. Optimeringarna innebär avlägsnade av redundanta komponenter, förenkling och anpassning av logiska uttryck. *Place and Route* innehåller utplacering och förbindning av komponenterna på chippet. I *Timing analysis* analyserar man fördräjningar mellan olika noder i realiseringen och ser om de uppfyller kravspecifikationen eller om det behövs omplacering av några komponenter på chippet. Man går alltså inte framåt genom flödesplanen hela tiden, utan måste ibland gå tillbaka och föreslå omkonstruktioner.



Figur 1.22 Moment vid realisering av en krets.

## VHDL – ett måste vid modern kretskonstruktion

Det är mycket begärt att beskrivningen av den lilla kretsen BCDcheck i VHDL skall belysa fördelar med beskrivning av kretsar i VHDL. Beskrivningen gjordes på relativt låg nivå. Trots att vi i den första beskrivningen av kretsen använde ordet "beteende" så beskrev vi nästan med de logiska uttrycken i tilldelningssatserna *strukturen* av realiseringen på grindnivå.

Större kretsar realiseras idag ofta på ett enda chip i en enda kapsel, t.ex. i en PLD. Kretsen kan på chippet bestå av flera komponenter. Detaljstrukturen för varje komponent bestäms kanske helt eller till stor del av syntesverktyget. Det är naturligtvis fördelaktigt för konstruktören att kunna beskriva beteendet på hög nivå för varje komponent för sig utan att behöva bekymra sig om detaljstrukturen. Strukturbeskrivningen görs i stället på hög nivå och kanske med grafisk editor eller med en beskrivning i VHDL hur komponenterna skall sammankopplas, ett blockschema i VHDL. Man kan arbeta i olika hierarkier med sin konstruktion. En krets beskriven i en nivå i hierarkin kan användas som en komponent i en ovanför liggande nivå.

Även om kretsen BCDcheck var enkel, så kan det vara fördelaktigt att den är beskriven i VHDL. Den skall kanske ingå som komponent i en större krets och kan då läggas in i en strukturbeskrivning i VHDL som en komponent. Filen BCDcheck.vhd kompileras då tillsammans med övriga komponenter och simulering kan göras på hög nivå och syntes kan sedan göras av den totala kretsen utgående från strukturbeskrivningen (blockschemat) och de ingående komponenternas VHDL-beskrivningar.

Beskrivning i VHDL har många fördelar, bl.a:

- Standardiserat språk
- Möjliggör beskrivning av kretsens struktur på blockschemanivå
- Möjliggör beskrivning av de ingående komponenternas beteende
- Möjliggör simulering av komplexa kretsar
- Möjliggör återanvändning av komponenter
- Ger god dokumentation av kretsen på olika nivåer
- Ger med hjälp av bra syntesverktyg snabb framtagning av kretsar
- Frammanar och uppmunstrar ett strukturerat arbetsätt hos konstruktören
- Ökar kvaliteten i konstruktionsprocessen och kretsen

# 1.5 Övningsuppgifter

## 1.3 Talsystem och koder

1.1 Omvandla till decimaltal

- a)  $10100_2$
- b)  $1100101_2$
- c)  $100010001_2$
- d)  $0.101_2$
- e)  $0.0111_2$
- f)  $0.00101_2$

1.2 Omvandla till decimaltal

- a)  $1A_{16}$
- b)  $A0C_{16}$
- c)  $20BF_{16}$
- d)  $37_8$
- e)  $174_8$
- f)  $2061_8$

1.3 Omvandla till binärtal, genom att bestämma vilka positionsvikter som skall ingå i talet, de decimala talen

- a) 81
- b) 150
- c) 401
- d) 645
- e) 1854
- f) 5176

1.4 Omvandla till binärtal, med den generella metoden med successiva divisioner med 2, talen i uppgift 1.3.

1.5 Omvandla de decimala talen nedan till binärtal, genom att bestämma vilka positionsvikter som skall ingå i talet. Noggrannhet 4 bitar (avrundat) till höger om binärpunkten.

- a)  $0,63_{10}$
- b)  $0,29_{10}$
- c)  $0,22_{10}$

1.6 Omvandla till binärtal, med den generella metoden med successiva multiplikationer med 2, talen i uppgift 1.5 med samma noggrannhet.

1.7 Omvandla till hexadecimala tal, talen i uppgift 1.1.

1.8 Omvandla till oktala tal, talen i uppgift 1.1.

1.9 Omvandla till binärtal

- a)  $D9A.4_{16}$
- b)  $B15A.7F2_{16}$
- c)  $174.6_8$
- d)  $467.013_8$

1.10 Omvandla 8-bitars binärtalen nedan till decimaltal, genom att gå via hexadecimala tal

- a) 10101010
- b) 01010101
- c) 00110011
- d) 10001111
- e) 01001100
- f) 10010111
- g) 01101001
- h) 01111110

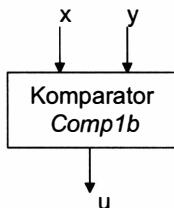
1.11 Omvandla till 8-bitars binärtal, genom att gå via hexadecimala tal, de decimala talen

- a) 31
- b) 43
- c) 100
- d) 141
- e) 225
- f) 255

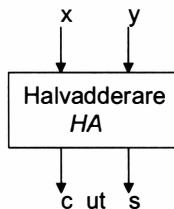
- 1.12 Omvandla till 4-siffrors hexadecimala tal, de decimala talen  
a) 705      b) 4336      c) 32767      d) 32768
- 1.13 Omvandla följande decimala tal till 4-siffrors hexadecimala tal  
a)  $2^8$       b)  $2^8 - 1$       c)  $2^{13}$       d)  $2^{13} - 1$       e)  $2^{16} - 1$
- 1.14 Skriv som 4-siffrors hexadecimala tal  
a) 1k      b) 2k      c) 4k      d) 8k      e) 16k      f) 32k  
g) 1/2k      h) 3k      i) 7k      j) 15k      k) 31k      l) 63k
- 1.15 Skriv som 6-siffrors hexadecimala tal  
a) 256k      b) 1M      c) 4M      d) 8M      e) 15M
- 1.16 Skriv som 8-siffrors hexadecimala tal  
a) 64M      b) 128M      c) 256M      d) 1G      e) 3G
- 1.17 Utför additionerna av de hexadecimala talen  
a)  $12 + 34$       b)  $76 + 54$       c)  $2A + DC$       d)  $BF + 2F$   
e)  $0FFF + 0001$       f)  $7FFF + 7FFF$       g)  $E0FF + 1001$
- 1.18 Utför subtraktionerna av de hexadecimala talen  
a)  $CD - AB$       b)  $98 - 89$       c)  $7A - 0F$       d)  $F0 - 0F$   
e)  $F000 - 0001$       f)  $F000 - 7FFF$       g)  $AF00 - AEFF$
- 1.19 Skriv de decimala talen i uppgift 1.3 med siffrorna i BCD-kod.
- 1.20 Konstruera en 84(-2)(-1)-kod för de decimala sifversymbolerna.
- 1.21 a) 84(-2)(-1)-koden och 5211-koden i tabell 1.3 har en gemensam egenskap. Vilken?  
b) Om man räknar 0, 1, 2, ..., 9, 0, 1, ... i 5421-koden i tabell 1.3, hur varierar då mest signifikanta biten?  
c) Ange en speciell egenskap hos 7421-koden i tabell 1.3.

## 1.4 Realisering av en liten krets i PLD – en introduktion till VHDL

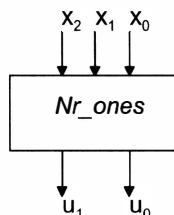
- 1.22 Beskriv i VHDL en komparator *Comp1b* (eng. *comparator*), som jämför två 1-bitars ord *x* och *y* med avseende på likhet. För komparatorn skall gälla att  $u = 1$  om och endast  $x = y$ . Börja med att rita en funktionstabell. Beskriv sedan kretsen med en tilldelningssats för *u* och använd då operationerna *and*, *or* och *not*. Ge kretsen namnet *Comp1b*.



- 1.23 Beskriv i VHDL en halvadderare HA (eng. *Half Adder*) som adderar två bitar *x* och *y* och bildar summan som en summasiffra *s* och en minnessiffra *c\_ut*. Börja med att rita en funktionstabell. Beskriv sedan halvadderaren med två tilldelningssatser för *s* och *c\_ut* och använd då operationerna *and*, *or* och *not*. Ge kretsen namnet HA.



- 1.24 Beskriv i VHDL en krets *Nr\_ones* med tre insignalér  $x_2, x_1, x_0$  och två utsignalér  $u_1, u_0$ , som indikerar antalet ettor hos insignalérna. Exempel: Om  $x_2, x_1, x_0 = 101$ , dvs innehåller två ettor, skall kretsen ge  $u_1, u_0 = 10$ , dvs talet två. Börja med att rita en funktionstabell. Deklarera  $x = (x_2, x_1, x_0)$  som en vektor. Beskriv sedan  $u_1$  med en *when-else*-sats och  $u_0$  med en tilldelningssats med operationerna *and*, *or* och *not*. Ge kretsen namnet *Nr\_ones*.



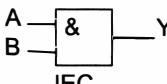
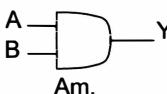
# 2 Grindar, vippor, kombinations- och sekvenskretsar

I detta kapitel ges en översikt över de fundamentala grindarna, som utgör de minsta byggstenarna i alla digitala kretsar. Grindarna visas som logiska byggblock. Hur grindarna ”ser ut inuti”, realiseras med MOS-transistorer, visas i ett senare kapitel. Digitala kretsar uppbyggda med grindar, s.k. kombinationskretsar, bl.a. fundamentala kretsar som multiplexer och demultiplexer visas och hur beteendet kan beskrivas med sanningstabell och logiskt uttryck. Kretsarnas funktion i tidsplanet och begreppet fördöjning berörs.

D-vippan, en annan fundamental byggsten, en krets med minne, presenteras också. Den används tillsammans med grindar för att realisera digitala kretsar med minnesfunktion, s.k. sekvenskretsar. Sådana kretsar såsom register, skiftregister och räknare och även generella sekvenskretsar visas och hur beteendet kan beskrivas med tillståndsdiagram och tillståndstabell.

## 2.1 Grindar och kombinationskretsar

### OCH-grind

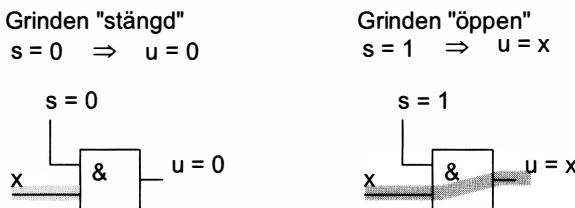
Symbol	Sanningstabell	Logisk operation															
 IEC	<table border="1"><thead><tr><th>A</th><th>B</th><th>Y</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></tbody></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	$Y = A \cdot B = AB$
A	B	Y															
0	0	0															
0	1	0															
1	0	0															
1	1	1															
 Am.		alt. $Y = A \text{ and } B$ $Y = A \wedge B$ $Y = A \& B$ $Y = A^*B$															

Figur 2.1 OCH-grind (eng. AND gate).

I sanningstabellen (eng. *truth table*) till OCH-grinden är de logiska konstanterna 'Falsk' (eng. *False*) och 'Sann' (eng. *True*) representerade med binära konstanterna '0' respektive '1'. Att sanningstabellen beskriver logiska operationen OCH får vi fram genom att säga att "Y = 1 om och endast om (A = 1) OCH (B = 1)".

Under IEC-symbolen visas den amerikanska symbolen, som har fördelen att den visar signalriktningen, men nackdelen att den är svårare att rita för hand. I fortsättningen används IEC-symboler. När det gäller symbol för logiska operationen OCH så förekommer som synes flera olika alternativ, sammanhanget får avgöra vilken symbol som är lämpligast att använda.

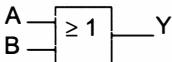
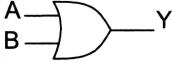
Bakgrunden till benämningen "grind" kan förklaras med figuren nedan. – Insignalen  $s$  kan betraktas som en styrsignal med vilken grinden kan öppnas och stängas, och insignalen  $x$  som en godtycklig signal som kan passera eller inte passera grinden. För styrsignalen  $s = 0$  (se A = 0 i sanningstabellen) är  $u = 0$  (se Y i sanningstabellen) oavsett värdet hos insignalen  $x$ , grinden är "stängd", och för  $s = 1$  (se A = 1 i sanningstabellen) följer  $u$  värdet hos  $x$ , dvs  $u = x$ , och  $x$  kan passera genom grinden, grinden är "öppen".



Figur 2.2 OCH-grinden som en "grind".

OCH-grinden i figur 2.1 är visad med två ingångar, men kan ha godtyckligt många ingångar. Exempelvis för en OCH-grind med tre ingångar A, B och C har sanningstabellen åtta rader ( $2^3 = 8$  kombinationer av insignalerna A, B och C) och Y = 1 om och endast om A = B = C = 1).

## ELLER-grind

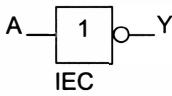
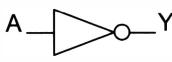
Symbol	Sanningstabell	Logisk operation															
 IEC	<table border="1"> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1	$Y = A + B$ alt. $Y = A \text{ or } B$ $Y = A \vee B$ $Y = A   B$ $Y = A \# B$
A	B	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
 Am.																	

Figur 2.3 ELLER-grind (eng. OR gate).

Ur sanningstabellen för ELLER-grinden utläser vi att ” $Y = 1$  om och endast om ( $A = 1$ ) ELLER ( $B = 1$ )”. Notera alltså att för den logiska operationen ELLER gäller att  $Y = 1$  också för insignal kombinationen  $AB = 11$ . Längre fram visas en annan variant av logiska operationen ELLER, benämnd Exklusivt-ELLER, för vilken gäller att  $Y = 0$  för insignal kombinationen  $AB = 11$ .

I grindsymbolen för ELLER ingår ” $\geq 1$ ”, vilket motiveras av att ELLER-grindens funktion också kan beskrivas som att ” $Y = 1$  om och endast om *en eller flera insignaler har värdet 1*”. Liksom OCH-grinden kan ELLER-grinden ha godtyckligt många ingångar.

## Inverterare

Symbol	Sanningstabell	Logisk operation						
 IEC	<table border="1"> <tr> <th>A</th> <th>Y</th> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	A	Y	0	1	1	0	$Y = A'$ alt. $Y = \text{not } A$ $Y = \sim A$ $Y = !A$ $Y = /A$ $Y = \neg A$ $Y = \overline{A}$
A	Y							
0	1							
1	0							
 Am.								

Figur 2.4 Inverterare (eng. inverter).

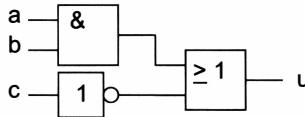
Inverteraren realiseras den logiska funktionen ICKE (NOT), ty ”Y = 1 om och endast om ICKE (A = 1)”. Observera att när vi verbalt beskriver logiska funktionerna för ICKE, OCH och ELLER, så gör vi det utgående från ’= 1’ (Sann). – I grindsymbolen är det ringen på utgången som representerar ICKE.

Liksom för OCH och ELLER så finns det för ICKE många olika symboler för logiska operationen. Den nedersta symbolen i figuren ovan, är ett streck (—) över variabeln. Denna symbol förekommer i databöcker för digitala kretsar. Den är olämplig att använda i komplexa logiska uttryck. Symbolen används också för att i datablad beteckna att en insignal är *aktiv låg*, t.ex. beteckningen write för en insignal till en minneskapsel innebär att skrivning av ett ord i minnet sker med signalen *läg*, *write* = 0.

## Några kombinationskretsar

### *Exempel 2.1*

Bestäm logiskt uttryck och sanningstabell till grindnätet nedan.



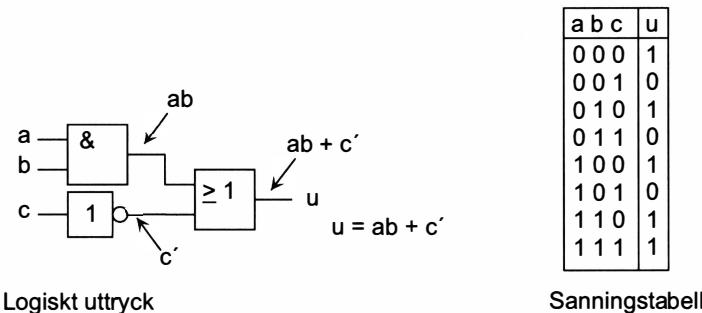
Figur 2.5 Grindnät exempel 2.1.

### *Lösning*

Logiska uttrycket tar vi fram genom att vandra genom grindnätet med början från nätets ingångar och för varje grind skriva upp logiska uttrycket vid grindens utgång, se figuren nedan där vi vandrat från vänster till höger genom nätet.

Sanningstabellen får vi här lättast fram genom att vi börjar vid nätets utgång. För ELLER-grinden gäller att utsignalen är 1 när någon insignal är 1. Den översta insignalen till ELLER-grinden är 1 när utsignalen från OCH-grinden är 1, dvs när insignalerna a = 1 och b = 1. I sanningstabellen skriver vi in u = 1 för dessa insignalkombinationer, dvs för de två sista raderna i tabellen. Den understa insignalen till ELLER-grinden är 1 när utsignalen från inverteraren är 1, dvs när insignalen c = 0. I sanningstabellen skriver vi in u = 1 för alla insignalkombinationer där c = 0. – Sanningstabellen kan

också bestämmas direkt utgående från det logiska uttrycket, vilket vi skall se är enkelt efter att vi studerat boolesk algebra i nästa kapitel.



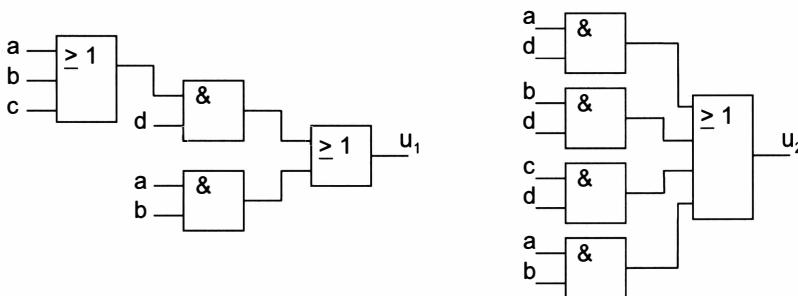
Figur 2.6 Logiskt uttryck och sanningstabell till grindnätet i exempel 2.1.

I fortsättningen kommer i logiska uttryck normalt att användas symbolerna  $(\cdot)$ ,  $(+)$  och  $(')$  för OCH, ELLER och ICKE, där punkten ej skrivs ut, som i logiska uttrycket i figur 2.6 ovan.

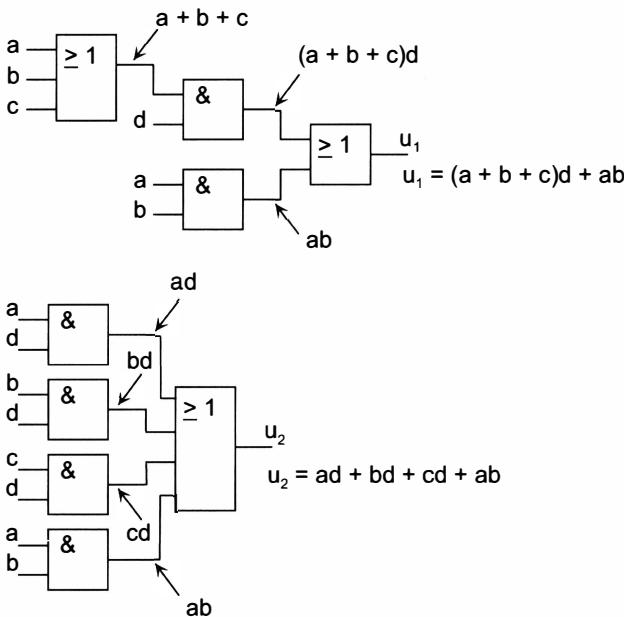
□

### Exempel 2.2

Bestäm logiskt uttryck och sanningstabell till grindnäten nedan.



Figur 2.7 Grindnät till exempel 2.2.

*Lösning*

a	b	c	d	$u_1$	$u_2$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Sanningstabell

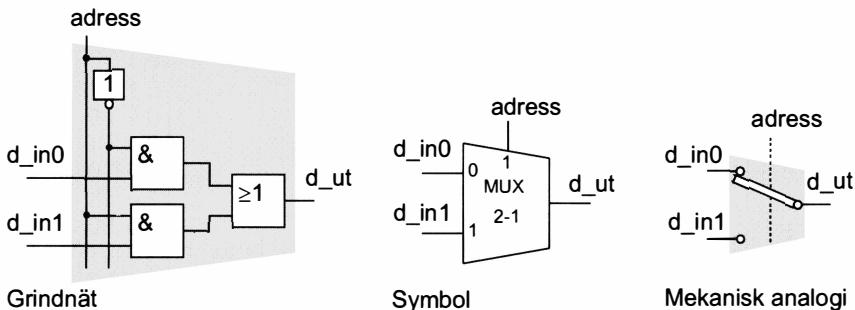
Figur 2.8 Logiskt uttryck och sanningstabell till grindnäten i exempel 2.2.

□

Vi ser i exempel 2.2 ovan hur de två grindnäten har samma sanningstabell. Allmänt gäller att en sanningstabell kan realiseras med oändligt många olika grindnät. Jämför vi de två grindnäten så kan vi konstatera att grindnätet till  $u_1$  har färre grindar än grindnätet till  $u_2$ . I gengäld har grindnätet till  $u_2$  bara maximalt två *grindnivåer* (eng. *gate levels*) räknat från ingång till utgång jämfört med grindnätet till  $u_1$  som har tre grindnivåer, dvs det tar kortare tid för en insignal att komma till utgången i grindnätet till  $u_2$  än i grindnätet till  $u_1$ . Exemplet belyser två aspekter som ständigt måste beaktas vid konstruktion av digitala kretsar, nämligen *snabbhet* och *komplexitet*. Man vill ofta konstruera snabba kretsar, men tvingas kompromissa mellan snabbhet och komplexitet, för att få acceptabel komplexitet får man ge avkall på snabbheten, ”tid är pengar”. Vi skall i ett kommande kapitel Kombinationskretsar närmare behandla syntes av grindnät, i detta kapitel ägnar vi oss bara åt analys av enkla grindnät.

## Multiplexer

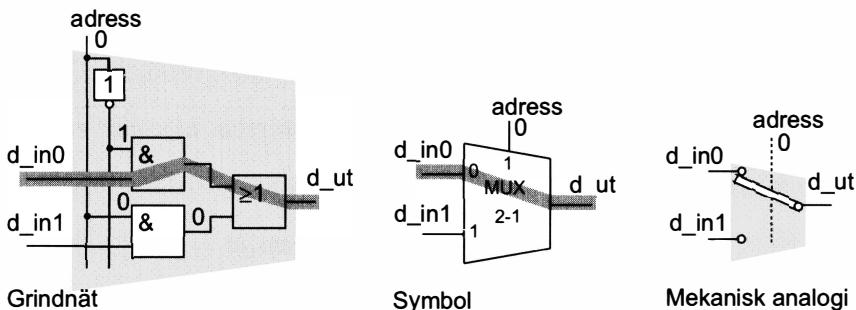
MUX 2-1. – I inledningskapitlet berördes funktionen för en *multiplexer* (MUX) (eng. *multiplexer*). Ett alternativt namn är *datavälvjare* (eng. *data selector*), som väl avspeglar funktionen. Den enklaste multiplexern är en *multiplexer 2-1*, som kan realiseras med en inverterare, två OCH-grindar och en ELLER-grind enligt figuren nedan.



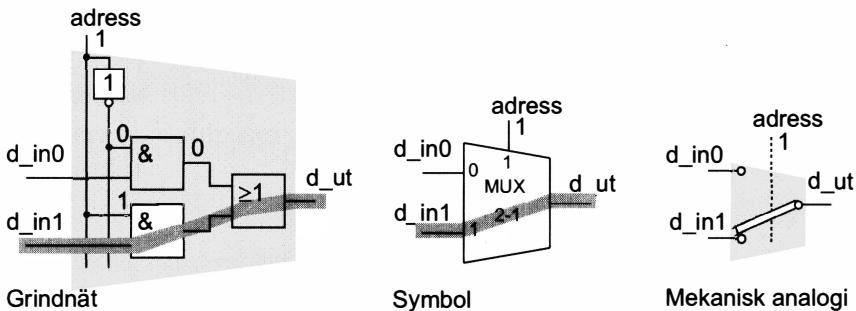
Figur 2.9 Multiplexer 2-1, MUX 2-1.

Multiplexern är en ”elektronisk” omkopplare, vars funktion bra illustreras med den mekaniska omkopplaren i figuren ovan. Omkopplingen styrs med insignalen *adress*, som väljer ut den ingång som skall förbindas med utgången (insignalen *adress* benämns ibland på engelska *select* (sv. *utvälja*)).

I figurerna nedan visas hur ingång  $d_{in0}$  väljs ut med  $adress = 0$  respektive hur ingång  $d_{in1}$  väljs ut med  $adress = 1$ , så att data kan sändas från den utvalda ingången till utgången. *Adress* = 0 medför att övre OCH-grinden är ”öppen” så att data på ingången  $d_{in0}$  sänds genom OCH-grinden och vidare genom ELLER-grinden till  $d_{ut}$ , medan nedre OCH-grinden är ”stängd” och dess utgångsvärde blir 0 oavsett värdet på ingången  $d_{in1}$ . *Adress* = 1 medför att nedre OCH-grinden är ”öppen” så att data på ingången  $d_{in1}$  sänds genom OCH-grinden och vidare genom ELLER-grinden till  $d_{ut}$ , medan övre OCH-grinden är ”stängd” och dess utgångsvärde blir 0 oavsett värdet på ingången  $d_{in0}$ .



Figur 2.10 MUX 2-1, där adress = 0 valt ut ingång d\_in0.



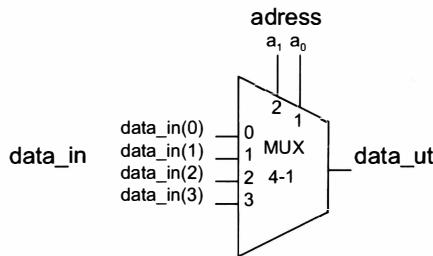
Figur 2.11 MUX 2-1, där adress = 1 valt ut ingång d\_in1.

Jämfört med den mekaniska omkopplaren, som kan sända ström i båda riktningarna, så kan signalerna i den digitala multiplexern endast gå från ingångarna till utgången.

I symbolen för multiplexern betecknar siffran 1 vid ingången *adress*, ingångens vikt och siffrorna vid dataingångarna ingångens adress. Symbolen är inte standard, men antyder med sin form och beteckningar väl multiplexerns funktion, den avsmalnande formen att antalet utgångar är färre än antalet ingångar.

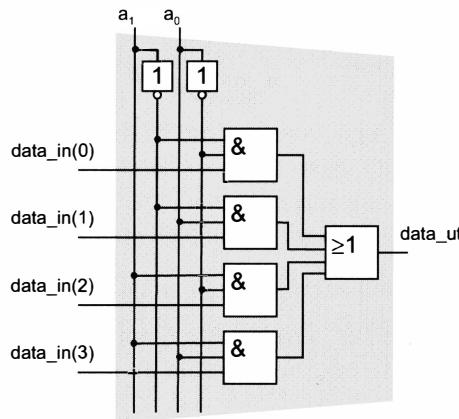
### Exempel 2.3

Realisera en multiplexer 4-1, MUX 4-1, med inverterare, OCH- och ELLER-grindar enligt samma princip som MUX 2-1 ovan.



Figur 2.12 Multiplexer 4-1, MUX 4-1, symbol.

### Lösning



Figur 2.13 Multiplexer 4-1, MUX 4-1, grindnät.

I en MUX 4-1 skall med adressen kunna väljas ut 1 av 4 ingångar. Adressen har därför 2 bitar, adress =  $(a_1, a_0)$ , vilket ger 4 ( $2^2 = 4$ ) kombinationer 00, 01, 10 och 11. Den MUX 2-1 som tidigare konstruerades hade två OCH-grindar, en för varje ingång, här i en MUX 4-1 med fyra ingångar skall vi uppenbart ha fyra OCH-grindar. Till varje OCH-grind ansluts både adresssignalerna inverterad eller icke-inverterad, så att OCH-grinden öppnas, dvs

dess båda adressingångar har värdet 1 för den adress som skall öppna OCH-grinden. Anslutningen görs enligt

Tabell 2.1: Adresssignaler till OCH-grindarna i MUX 4-1 i figuren ovan.

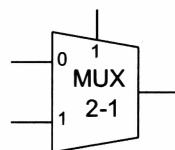
adress	data_in	adresssignaler till OCH-grind
00	0	$a_1' a_0'$
01	1	$a_1' a_0$
10	2	$a_1 a_0'$
11	3	$a_1 a_0$

Vi ser i tabellen ovan att OCH-produkterna för adresssignaler är lika med 1 för tillhörande adress och lika med 0 för de andra adresserna, innebärande att OCH-grinden för tillhörande adress är öppen och utgångsvärdet bestäms av data\_in, medan övriga OCH-grindar är stängda och har utgångsvärdet 0.

□

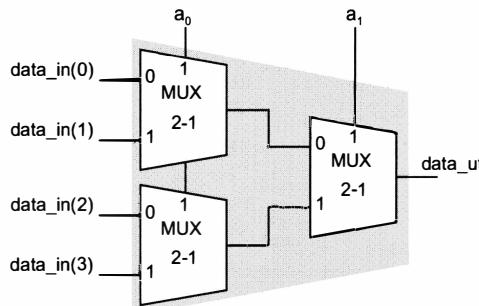
### Exempel 2.4

Realisera en multiplexer 4-1, MUX 4-1, enligt figur 2.9 med tre MUX 2-1 enligt nedan.



Figur 2.14 MUX 2-1, symbol.

### Lösning

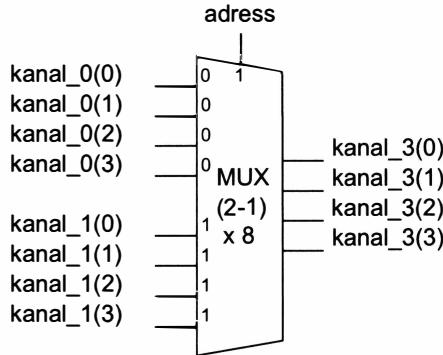


Figur 2.15 Multiplexer 4-1.

□

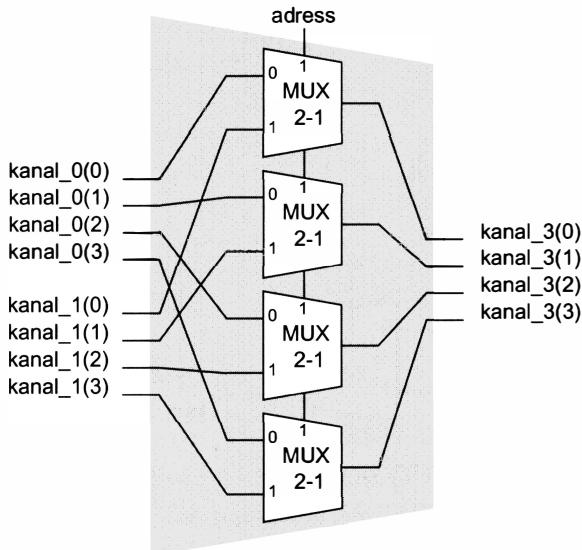
**Exempel 2.5**

Realisera en multiplexer  $(2-1) \times 4$ , MUX  $(2-1) \times 4$ , som skall användas som kanalväljare för två 4-bitars kanaler, kanal 0 och kanal 1. För adress = 0 skall data på kanal 0 överföras till kanal 3, medan för adress = 1 skall data på kanal 1 överföras till kanal 3. Realisera multiplexern med ett lämpligt antal multiplexrar 2-1 enligt figur 2.11 ovan.



Figur 2.16 Multiplexer  $(2-1) \times 4$ .

*Lösning*

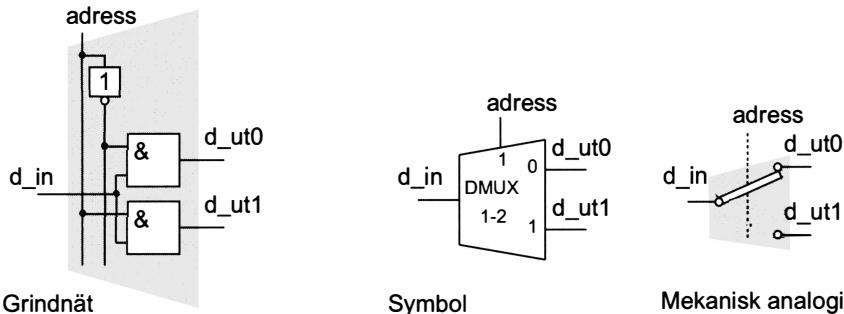


Figur 2.17 Multiplexer  $(2-1) \times 4$  med fyra multiplexer  $(2-1)$

□

## Demultiplexer

DMUX 1-2. – I inledningskapitlet berördes också funktionen för en *demultiplexer* (DMUX) (eng. *demultiplexer*). Ett alternativt namn är *datafördelare* (eng. *data distributor*), som väl avspeglar funktionen. Den enklaste demultiplexern är en *demultiplexer 1-2* enligt figuren nedan.

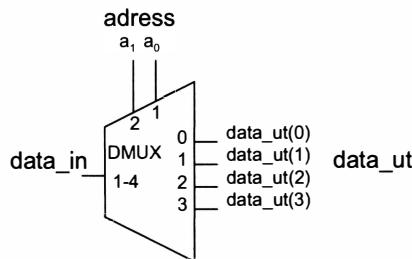


Figur 2.18 Demultiplexer 1-2.

Demultiplexern är liksom multiplexern en ”elektronisk” omkopplare, men funktionen är ”tvärtom” mot multiplexerns. Med insignalen *adress* väljs här ut en utgång som skall förbindas med ingången. *Adress* = 1 medför att nedre OCH-grinden är ”öppen” så att data på ingången d\_in sänds genom OCH-grinden till d\_ut1, medan övre OCH-grinden är ”stängd” och dess utgångsvärde blir 0 oavsett värdet på ingången d\_in.

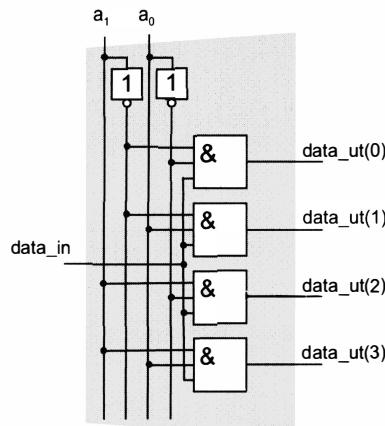
### Exempel 2.6

Realisera en demultiplexer 1-4, DMUX 1-4, med inverterare, OCH- och ELLER-grindar enligt samma princip som DMUX 1-2 ovan.



Figur 2.19 Demultiplexer 1-4, DMUX 1-4, symbol.

### Lösning

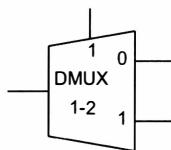


Figur 2.20 Demultiplexer 1-4, DMUX 1-4, grindnät.

□

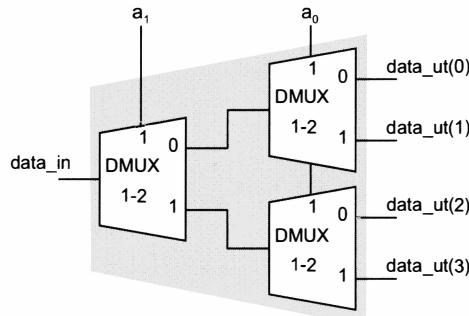
### Exempel 2.7

Realisera en demultiplexer 1-4, DMUX 1-4, med tre DMUX 1-2 enligt nedan.



Figur 2.21 DMUX 1-2, symbol.

### Lösning

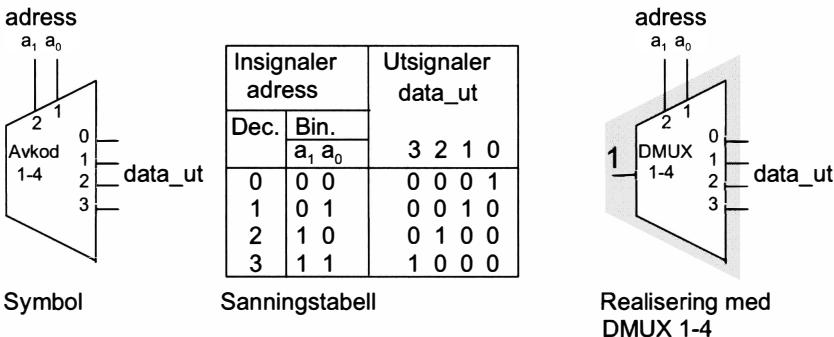


Figur 2.22 Demultiplexer 1-4.

□

## Avkodare

En *avkodare* (eng. *decoder*) är en variant av en demultiplexer. Funktionsprincipen illustreras i figuren nedan med en *avkodare 1-4* som är uppbyggd med en demultiplexer 1-4, där dataingången data\_in getts konstant värde 1.



Figur 2.23 Avkodare 1-4 realiserad med en DMUX 1-4.

I sanningstabellen ovan ser vi hur för varje kombination av adressbitarna en och endast en av utsignalerna blir 1, den utsignal vars nummer överensstämmer med decimaltalet för adressen, medan övriga utsignaler är 0. Adressen *avkodas* här till en *1-av-4-kod* (en och endast en av bitarna i koden är 1 medan övriga är 0) på avkodarens utsignaler.

Generellt gäller för en avkodare som har en adress på  $n$  bitar att adressen avkodas till en  $1\text{-av-}2^n$ -kod på avkodarens utsignaler. I vissa fall behövs inte alla  $2^n$  utgångar, som t.ex. för en BCD-avkodare, som skall avkoda de decimala siffrorna 0 till 9 till en 1-av-10-kod. Den vanliga BCD-koden har 4 bitar, vilket vid en fullständig avkodning ger  $2^4 = 16$  utgångar, men för BCD-avkodaren behövs sålunda inte utsignalerna och OCH-grindarna som realiseras dem för kombinationerna 10 till 15.

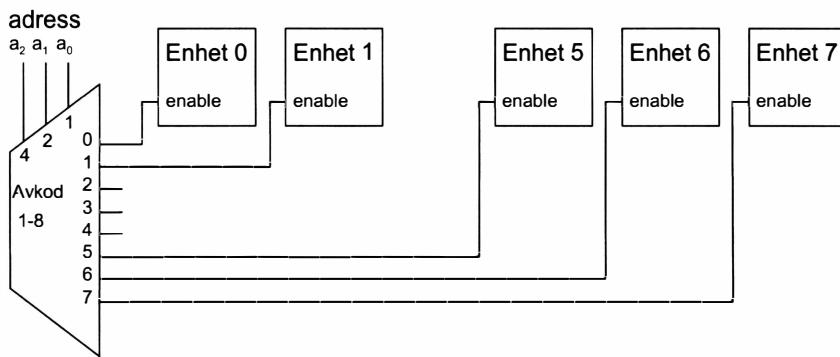
En avkodare kan också realiseras så att den av adressen "utpekade", aktiva utsignalen blir 0 medan övriga bitar är 1, dvs koden är inverterad jämfört med den tidigare koden ovan. En sådan avkodare realiseras uppenbart genom att man förser avkodaren ovan med inverterare på utgångarna.

I avkodaren adresseras, utpekas, alltså en av utgångarna och den vanligaste användningen är för att adressera kretsar eller enheter så att en och endast en krets, enhet, i taget är aktiv som framgår av exemplet nedan.

### Exempel 2.8

Fem enheter är vardera försedda med en insignal *enable* (eng. *enable* = sv. *möjliggöra för* ngn) som styr om enheten skall vara aktiv (till) eller inaktiv (från); *enable* = 1 medför aktiv, medan *enable* = 0 medför inaktiv. En och endast en enhet i taget skall vara aktiv och detta skall styras med en avkodare 1-8, vars insignaler är en 3-bitars adress ( $a_2, a_1, a_0$ ). Enheterna är betecknade 0, 1, 5, 6 och 7 och en enhet skall aktiveras med en adress till avkodaren motsvarande enhetens nummer. För t.ex.  $(a_2, a_1, a_0) = 110$  blir utgång 6 hos avkodaren lika med 1, medan övriga utgångar är lika med 0 och sålunda enhet 6 är aktiv, medan övriga enheter är inaktiva.

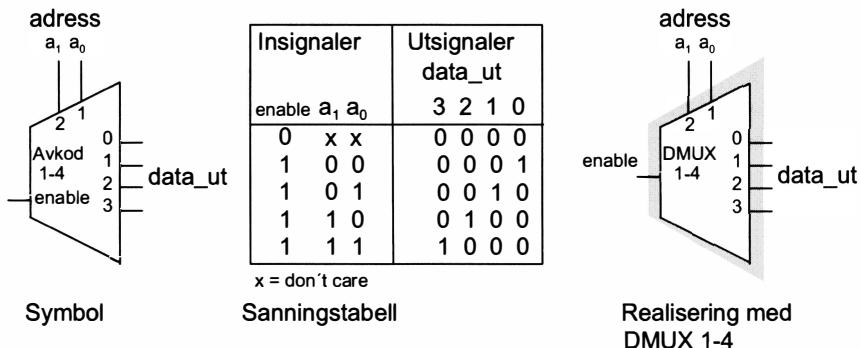
### Lösning



Figur 2.24 Adressering av enheter, en i taget, med en avkodare.

□

Digitala kretsar förses ofta med en eller flera insignaler *enable*, som styr om kretsen skall vara aktiv (till) eller inaktiv (från). Avkodaren i figur 2.20, som är realiseras med en DMUX 1-4 kan förses med en insignal *enable* genom att DMUX:ens insignal i stället för att ges ett konstant värde 1 får bli en insignal *enable* enligt figuren nedan. För *enable* = 0 blir samtliga utgångar hos DMUX:en lika med 0, oavsett värdet hos adressen. För *enable* = 1 blir funktionen densamma som för realiseringen i figur 2.20, dvs DMUX:en fungerar som avkodare och adresserad utsignal får värdet 1, medan övriga utsignaler är 0.



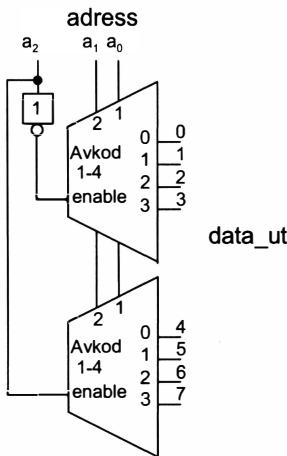
Figur 2.25 Avkodare 1-4 försedd med enable-signal.

Insignalen *enable* hos avkodaren kan användas för sammankoppling av flera avkodare till en större avkodare.

### Exempel 2.9

Realisera en avkodare 1-8 med två avkodare 1-4 försedda med enable.

### Lösning



Figur 2.26 Avkodare 1-8 realiserad med två avkodare 1-4.

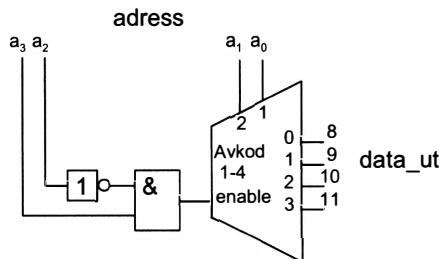
För  $a_2 = 0$  är den övre avkodaren *till* och den undre *från*, medan för  $a_2 = 1$  är den övre avkodaren *från* och den undre *till*.

□

### Exempel 2.10

En 4-bitars adress ( $a_3, a_2, a_1, a_0$ ) används för att adressera enheter så att en och endast en enhet i taget är aktiv. Enheterna 8-11 skall adresseras med en avkodare 1-4 som är försedd med en insignal *enable*. Anslut adresssignaler till avkodaren så att den kan adressera i det angivna adressområdet. Använd lämpliga grindar.

### Lösning

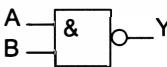
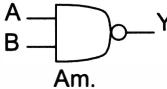


Figur 2.27 Avkodare som adresserar i adressområdet 8-11 med en 4-bitars adress.

$enable = 1$  om och endast om  $a_3 = 1$  och  $a_2 = 0$ . Avkodaren är alltså aktiv enbart för adresserna 1000, 1001, 1010 och 1011, dvs 8, 9, 10 och 11.

□

## NAND-grind

Symbol	Sanningstabell	Logisk operation															
 IEC	<table border="1" data-bbox="498 1128 595 1268"> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0	$Y = (AB)'$
A	B	Y															
0	0	1															
0	1	1															
1	0	1															
1	1	0															
 Am.																	

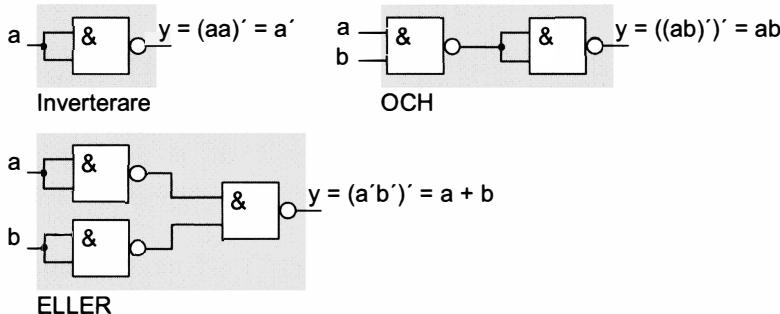
Figur 2.28 NAND-grind.

I sanningstabellen för NAND-grinden läser vi att ” $Y = 1$  om och endast om  $ICKE((A = 1) \text{ OCH } (B = 1))$ ”. Kretsen realiseringen alltså den logiska operationen ICKE-OCH (eng. NOT-AND, NAND).

**Exempel 2.11**

Realisera en Inverterare, en OCH-grind och en ELLER-grind med enbart 2-ingångars NAND-grindar.

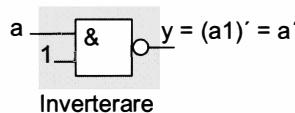
*Lösning*



Figur 2.29 Inverterare, OCH och ELLER realiserad med NAND-grindar.

Realiseringarna fås ur sanningstabellen för NAND-grinden. Inverteraren för insignalerna  $AB = 00$  och  $AB = 11$ . OCH-grinden genom att byta 0 mot 1 och tvärtom i Y-kolumnen. ELLER-grinden genom att i sanningstabellen invertera A och B (byta 0 mot 1 och tvärtom) och sedan skriva om raderna i ordning, vilket ger sanningstabellen för ELLER. Med hjälp av boolesk algebra, som vi skall studera i nästa kapitel, kan man lätt visa realiseringarna rent algebraiskt.

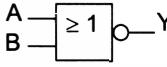
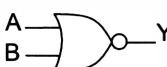
Som alternativ till att lägga samma signal till båda ingångarna vid realiseringen av inverteraren kan i stället den ena ingången ges insignalen 1 enligt figuren nedan. Observera *inte* insignalen 0, ty då blir utsignalen hos NAND-grinden alltid lika med 1 oavsett värdet hos den andra insignalen. Oanvända ingångar hos OCH- och NAND-grindar skall alltså ges insignalen 1.



Figur 2.30 Alternativ realisering av inverterare med 2-ingångars NAND-grind.

□

## NOR-grind

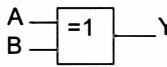
Symbol	Sanningstabell	Logisk operation															
 IEC	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	$Y = (A + B)'$
A	B	Y															
0	0	1															
0	1	0															
1	0	0															
1	1	0															
 Am.																	

Figur 2.31 NOR-grind.

I sanningstabellen för NOR-grinden läser vi att ” $Y = 1$  om och endast om  $ICKE((A = 1) ELLER (B = 1))$ ”. Kretsen realiseras den logiska operationen ICKE-ELLER (eng. NOT-OR, NOR).

NAND- och NOR-grindar är enklare att realisera med transistorer än OCH- och ELLER-grindar beroende på transistorns naturliga förmåga att invertera, som utnyttjas vid realisering av NAND och NOR, men som måste kompenseras med en extra inverterare vid realisering av OCH och ELLER. Även om fabrikanterna visar funktionsschema för en krets med OCH- och ELLER-grindar, så är det vanligen bara en beskrivning av den logiska funktionen och inte den verkliga realiseringen med grindar, som sannolikt är med NAND- och NOR-grindar.

## Exklusivt-ELLER-grind, XOR-grind

Symbol	Sanningstabell	Logisk operation															
 IEC	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0	$Y = A \oplus B$ alt. $Y = A \text{ xor } B$ $Y = A \wedge B$
A	B	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	0															
 Am.																	

Figur 2.32 Exklusivt-ELLER-grind, XOR-grind (eng. Exclusive-Or gate).

I sanningstabellen för Exklusivt-ELLER-grinden utläser vi att ” $Y = 1$  om och endast om *antingen* ( $A = 1$ ) *ELLER* ( $B = 1$ )”. Exklusivt-ELLER är således rent språkligt ”*antingen-eller*”, en skärpning av det ”vanliga” *eller* genom att  $A = 1$  och  $B = 1$  utesluter, exkluderar, varandra, därav benämningen Exklusivt-ELLER. I grindsymbolen för Exklusivt-ELLER står ”=1” motiverat av att ” $Y = 1$  om och endast om *exakt en* insignal har värdet 1”.

Det finns andra tolkningar av Exklusivt-ELLER som ger viktiga användningar av Exklusivt-ELLER-grinden. – En tolkning är att ” $Y = 1$  om och endast om insignalerna är *olika*” eller alternativt att ” $Y = 0$  om och endast om insignalerna är *lika*”, som gör det möjligt att använda grinden som *jämförare, komparator*.

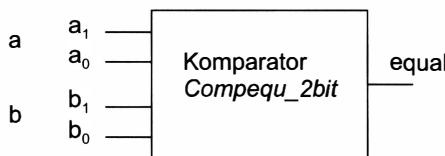
## Några fler kombinationskretsar

### Komparator

Ofta behöver man i digitala system jämföra (eng. *compare*) två binära ord med avseende på relationen ”likhet” (=), dvs överensstämelse i alla bitar.

En komparator *Compequ\_2bit* som jämför två 2-bitars ord  $a = (a_1, a_0)$  och  $b = (b_1, b_0)$  med avseende på likhet visas nedan. Komparatorn har fyra ingångar  $a_1, a_0, b_1$  och  $b_0$  och en utgång *equal*. Beteendet skall vara

$$\text{equal} = 1 \text{ om och endast om } a = b$$

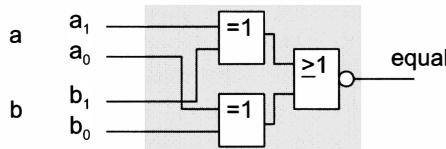


Figur 2.33 Komparator som jämför två 2-bitars ord med avseende på likhet.

För orden  $a$  och  $b$  gäller att

$$a = b \text{ om och endast om } (a_1 = b_1) \text{ och } (a_0 = b_0)$$

Jämförelsen av  $a_1$  med  $b_1$  respektive  $a_0$  med  $b_0$  kan göras i var sin XOR-grind. Ut signalerna från de två XOR-gindarna ansluts sedan till en grind för vilken gäller att utsignalen är 1 om och endast om båda insignalerna är 0, dvs en NOR-grind.

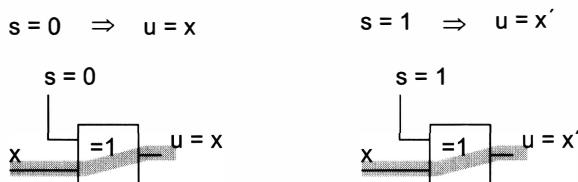


Figur 2.34 Realisering av komparatorn Compequ\_2bit.

Jämförelsen av orden görs alltså bitvis (eng. *bitwise*)

En annan tolkning av Exklusivt-ELLER är att ” $Y = 1$  om och endast om ett *udda* antal insignaler har värdet 1”, som gör det möjligt att använda grinden för s.k. *paritetskontroll*, en metod för felupptäckt i binära ord, som berörs i ett senare kapitel.

XOR-grinden kan också användas som styrbar inverterare, enligt figuren nedan.



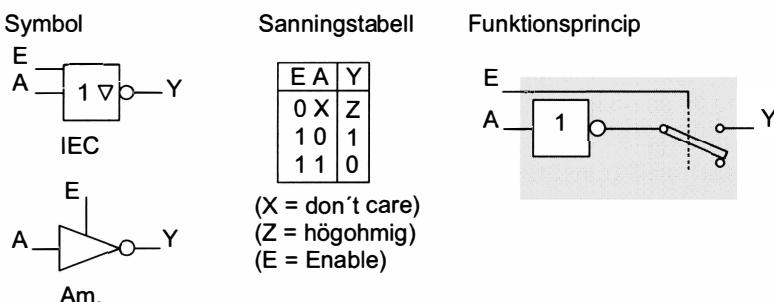
Figur 2.35 XOR-grinden som styrbar inverterare.

Med insignalen  $s$  i figuren ovan betraktad som en styrsignal, så gäller att för  $s = 0$  (se A = 0 i sanningstabellen ovan för XOR-grinden) så går insignalen  $x$  opåverkad genom XOR-grinden (se B och Y i sanningstabellen), medan för  $s = 1$  (se A = 1 i sanningstabellen) så inverteras insignalen  $x$  (se B och Y i sanningstabellen).

## Three-state-utgång

De hittills studerade grindarna har haft två utgångsvärden (utgångstillstånd), Låg (0) eller Hög (1). Många digitala kretsar är försedda med utgångar som förutom dessa två utgångstillstånd även kan anta ett tredje tillstånd, då utgången varken är Låg eller Hög, utan är *frisvävande*, *högohmig* (eng. *high impedance*). Kretsen har alltså en utgång med *tre* utgångstillstånd (eng. *three-state output*, av. eng. *tri-state*). I figuren nedan visas symbol, sanningstabell och funktionsprincip för en inverterare som är försedd med three-state-utgång.

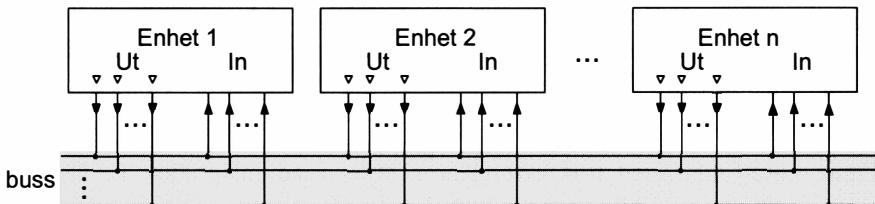
Three-state-funktionen hos inverteraren styrs med insignalen *E*, *Enable* (eng. *enable* = sv. *möjliggöra för* ngn). För  $E = 0$  är utgången *Y* frisvävande, medan för  $E = 1$  är utgången *Y* ansluten till inverteraren och kretsen fungerar som en inverterare. – Insignaler benämnda *Enable* förekommer ofta hos digitala kretsar och används för att *aktivera* respektive *inaktivera* någon funktion hos kretsen.



Figur 2.36 Inverterare med three-state-utgång

Orsaken till att digitala kretsar ofta har utgångar av typ three-state är att det skall gå lätt att koppla samman kretsar. Dataöverföring mellan olika enheter i digitala system sker normalt via s.k. *bussar* (av latinets *omnibus*: "till för alla"), så är fallet t.ex. i en dator. En buss i detta sammanhang är ett antal parallella ledningar, se figuren nedan, som bildar en väg på vilken binära ord, med lika många bitar som antalet ledningar, kan överföras. Ett godtyckligt antal enheter skall kunna anslutas till bussen, som uppenbarligen är "till för alla". Förutsättningen för att en enhet skall kunna anslutas till bus-

sen är att dess utgångar är av typ three-state. Principen för kommunikation på bussen är att endast en enhet i taget, sändande enheten, får ha sina utgångar aktiva, medan samtliga andra enheter har sina utgångar inaktiva, frivävande, så att de inte påverkar bussledningarnas tillstånd.



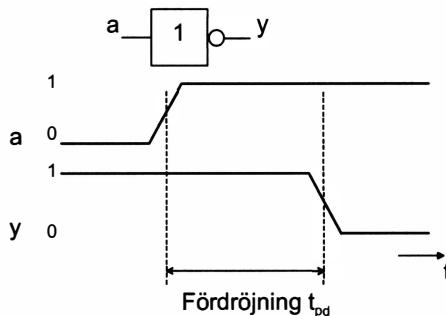
Figur 2.37 Princip för anslutning av enheter till en buss.

## Fördräjning

I digitala kretsar tar det alltid en viss tid innan en ändring av en insignal ger en ändring av en utsignal, det finns en *fördräjning* (eng. *propagation delay*),  $t_{pd}$ , mellan insignal och utsignal. Fördräjningen beror på att det i kretsens uppbyggnad med transistorer på kiselbrickan, naturligt och oundvikligen finns diverse kondensatorer, som skall laddas upp eller laddas ur vid signaländringar. Fördräjningens storlek beror på kondensatorernas storlek och upp- och urladdningsströmmarnas storlek. Ju mindre kondensator och ju större ström desto snabbare upp- och urladdning och mindre fördräjning.

Digitala kretsar blir snabbare och snabbare för varje år, dvs fördräjningarna blir mindre och mindre. Detta beror på att man lyckas göra kretsarna mindre och mindre och därmed blir också kondensatorerna mindre och mindre och sålunda även fördräjningarna. Amerikanen Gordon Moore, företaget Intels grundare, påstod 1965 att man kommer att kunna fördubbla antalet transistorer per kvadrattum på ett chip var 18:e månad. Utvecklingen har hittills med förvånansvärd noggrannhet följt denna s.k. *Moore's lag*.

Fördräjningen i en digital krets illustreras nedan med fördräjningen i en inverterare.



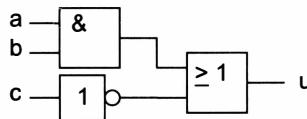
Figur 2.38 Fördräjning i en inverterare.

Signalerna ändrar sig inte momentant från 0 (L) till 1 (H) och tvärtom, utan har alltid en viss *stigtid* (eng. *rise time*) och *falltid* (eng. *fall time*), som också beror på upp- och urladdning av kondensatorer, såsom schematiskt åskådliggjorts i figuren ovan. Fördräjningen mäts vid signalernas 50 %-nivå.

### Exempel 2.12

Bestäm maximala fördräjningen för grindnätet nedan (samma grindnät som i figur 2.5). Antag följande fördräjningar

$$t_{pd\_INV} = 0,2 \text{ ns} \quad t_{pd\_OCH} = 0,4 \text{ ns} \quad t_{pd\_ELLER} = 0,5 \text{ ns}$$



### Lösning

Fördräjningarna från ingång till utgång är

$$a, b \rightarrow u: t_{pd\_OCH} + t_{pd\_ELLER} = 0,4 + 0,5 = 0,9 \text{ ns}$$

$$c \rightarrow u: t_{pd\_INV} + t_{pd\_ELLER} = 0,2 + 0,5 = 0,7 \text{ ns}$$

Maximala fördräjningen i grindnätet är 0,9 ns.

□

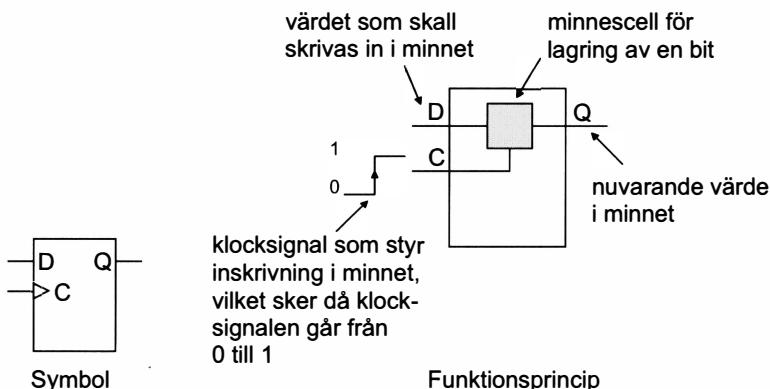
Grindarna och grindnäten vi sett exempel på i detta kapitel tillhör en klass av kretsar som kallas *kombinationskretsar* (eng. *combinational circuits*). Det är kretsar utan minne. Utsignalerna hos en kombinationskrets beror endast av det aktuella insignalvärdet och ej av tidigare insignalvärdet, eftersom kretsen ej kommer ihåg vad som skett tidigare. Tiden är intressant för en kombinationskrets bara vad gäller fördräjningen mellan insignal och utsignal.

Vi skall nu studera D-vippan, en minneskrets, som används för att realisera kretsar med minne, tillhörande klassen *sekvenskretsar* (eng. *sequential circuits*).

## 2.2 Vippor och sekvenskretsar

### D-vippan

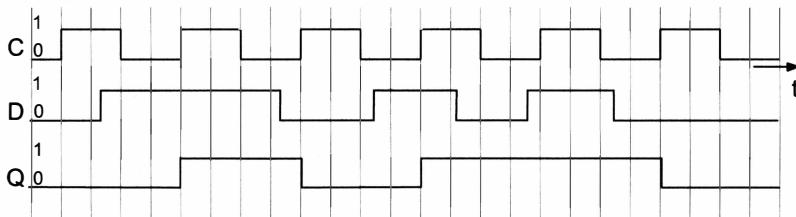
D-vippan (eng. *D flip flop*) är en minneskrets som kan lagra en bit. Den kan realiseras med grindar, vilket vi skall se i ett senare kapitel. Här skall vi bara betrakta den som ett byggblock.



Figur 2.39 D-vippa, symbol och funktionsprincip.

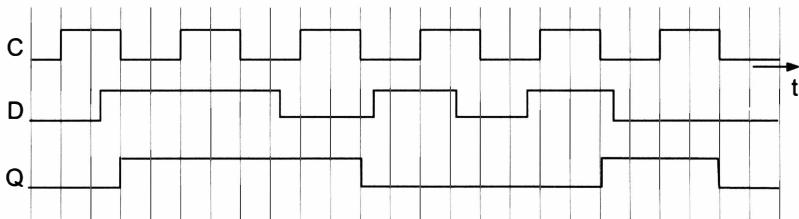
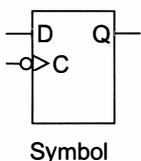
Som framgår av figuren ovan så har D-vippan två ingångar, en ingång D (Data) för biten som skrivas in och en ingång C (Clock) som styr inskrivning i minnescellen, och en utgång Q som visar värdet hos biten som är lagrad minnescellen i D-vippan. Klocksignalen är normalt en fyrkantvåg

(eng. *square wave*) enligt exemplet i figuren nedan, som visar D-vippans funktion i tidsplanet. Inskrivning av värdet på D-ingången i minnescellen sker då klocksignalen går från 0 (L) till 1 (H), på klocksignalens *positiva flank* (eng. *positive edge*), man säger att vippan är *positivt flanktriggad* (eng. *positive edge triggered*). Värdet på Q-utgången kan bara förändras på denna flank, med en viss födröjning som ej är utritad i figuren nedan.



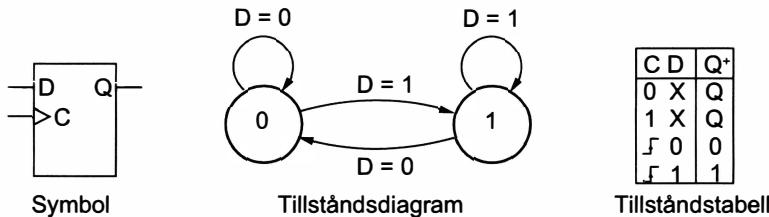
Figur 2.40 D-vippans funktion i tidsplanet.

D-vippan kan också konstrueras så att inskrivning av värdet på D-ingången i stället sker då klocksignalen går från 1 (H) till 0 (L), på klocksignalens *negativa flank* (eng. *negative edge*), och är då *negativt flanktriggad* (eng. *negative edge triggered*). I symbolen för D-vippa markeras detta med en inverterarrangemang på klocksignalingången C, enligt figuren nedan.



Figur 2.41 D-vippa, negativt flanktriggad.

D-vippans beteende kan beskrivas grafiskt i ett *tillståndsdiagram* (eng. *state diagram*) och i en *tillståndstabell* (eng. *state table*) enligt figuren nedan.



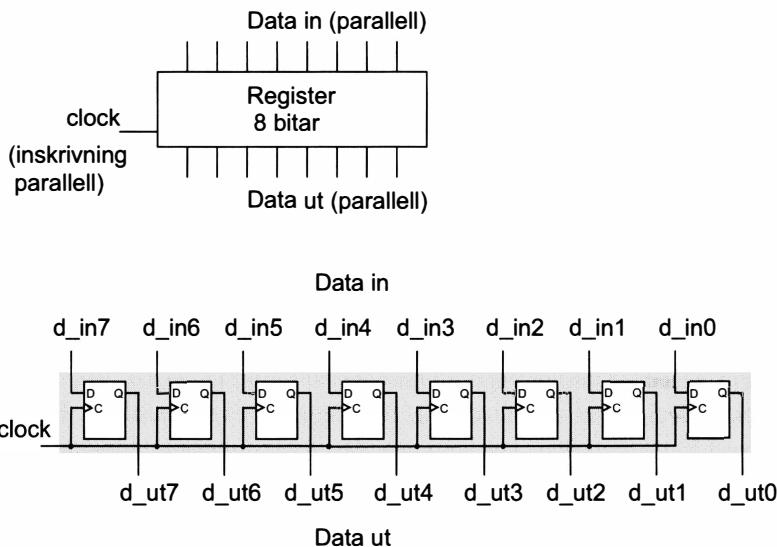
Figur 2.42 D-vippan beskriven i ett tillståndsdiagram och en tillståndstabell.

D-vippan kan befina sig i två olika *tillstånd* (eng. *state*)  $Q = 0$  och  $Q = 1$ . I tillståndsdiagrammet representeras tillstånden med cirklar och tillståndsövergångarna med pilar. Tillståndsövergångar sker för D-vippan ovan på klocksignalens positiva flank. På pilarna står ingångsvärdeet som ger tillståndsövergången. I tillståndstabellen betecknar  $Q^+$  nästa tillstånd (eng. *next state*), tillståndet som intas efter klocksignalens positiva flank. Vidare står X i tabellen för *don't care*, *irrelevant*. Sålunda då klocksignalen är konstant 0 eller 1 så är sker ingen tillståndsändring, dvs  $Q^+ = Q$ , nästa tillstånd är lika med nuvarande tillstånd (eng. *present state* äv. *current state*).

D-vippan används inte för uppbyggnad av stora minnen, till det används enklare minnesceller, som visas längre fram i kapitlet Halvledarminnen. D-vippans främsta användning är dels för konstruktion av speciella sekvenskretsar såsom *register*, *skifregister* och *räknare*, vilka används som byggblock vid realisering av digitala kretsar, dels för konstruktion av generella sekvenskretsar.

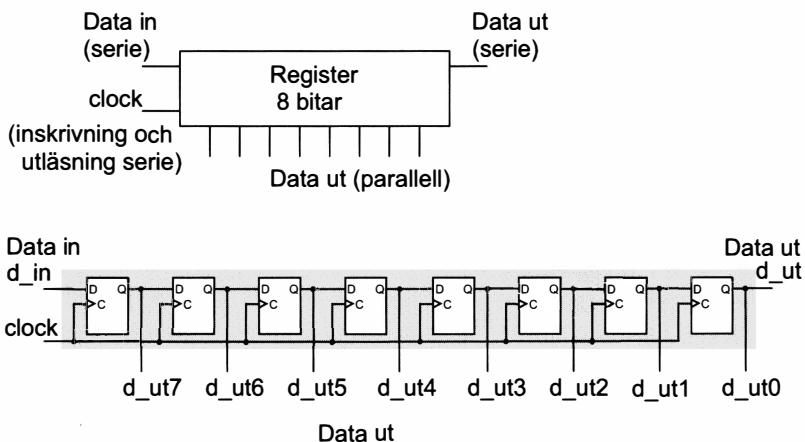
## Register och skifregister

Ett *register* (eng. *register*) är ett minne för lagring av ett ord. I figuren nedan visas exempel på ett register för lagring av ett 8-bitars ord. Inskrivning av ordet i registret nedan sker med klocksignalen.



Figur 2.43 Register för 8-bitars ord, parallel in - parallel ut.

Registret ovan är av typ *parallel in - parallel ut*, dvs inskrivning av ett ord och utläsning av ett ord sker med alla bitar på en gång. Det finns också register där inskrivning och utläsning av ett ord sker med en bit i taget, s.k. *skiftregister* (eng. *shift register*). Exempel på ett sådant register visas i figuren nedan.

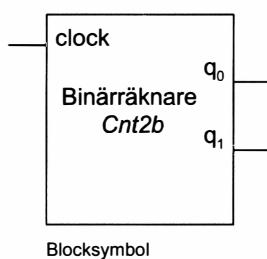
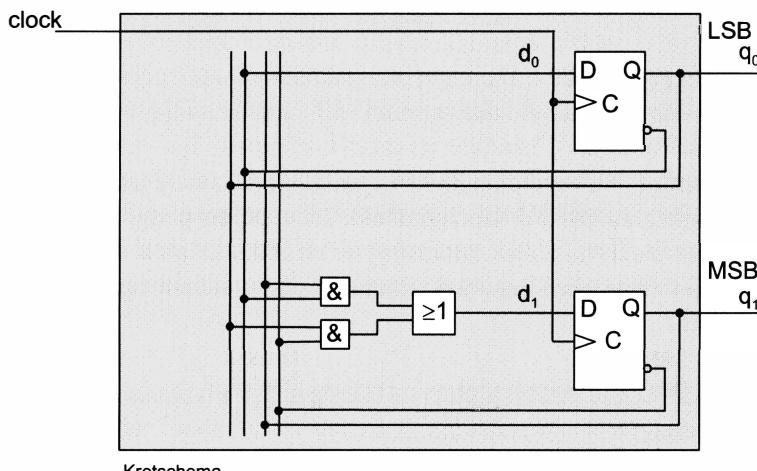


Figur 2.44 Skiftregister för 8-bitars ord, serie in - parallel/serie ut.

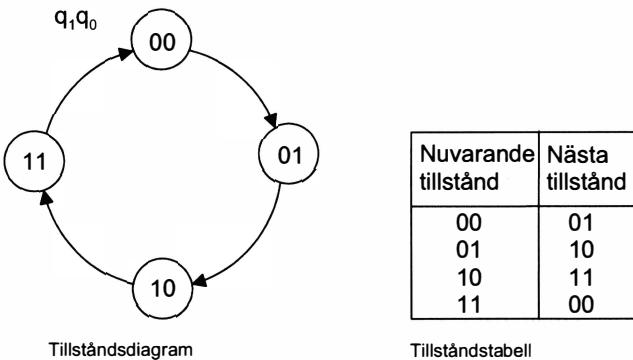
Bitarna läggs en i taget på ingången *Data in* och skrivs in, skiftas in åt höger, med klocksignalen (*clock*). Utläsning kan här ske med alla bitarna på en gång eller genom utläsning av en bit i taget på utgången *Data ut* och skift höger av bitarna i registret med klocksignalen. Registreringen är av typ *serie in - parallell/serie ut*. Fler varianter av skiftregister förekommer och visas i ett senare kapitel.

## Räknare

En annaniktig användning av D-vipporna är för realisering av *räknare* (eng. *counter*). I figuren nedan visas en räknare som räknar 00, 01, 10, 11, 00, .... Uttryckt i decimaltal räknar den 0, 1, 2, 3, 0, ..., dvs modulo-4. Räkningen sker med klocksignalen. I schemat nedan är D-vipporna försedda med en inversutgång *Q'*, vilket ofta är fallet.



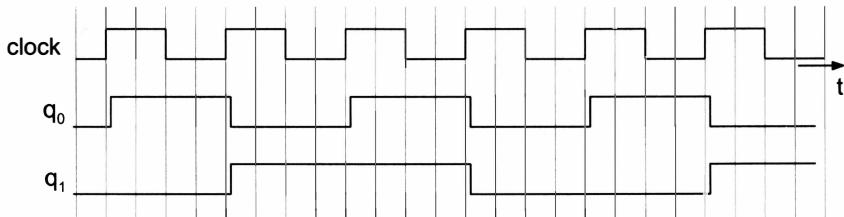
Figur 2.45 2-bitars binärräknare *Cnt2b*, som räknar modulo-4.



Figur 2.46 2-bitars binärräknare *Cnt2b*, tillståndsdiagram och tillståndstabell.

Räknaren har två D-vippor, som vardera kan anta två olika tillstånd 0 och 1. Alltså har räknaren  $2 \cdot 2 = 2^2 = 4$  tillstånd 00, 01, 10 och 11, som representeras med 4 cirklar i tillståndsdiagrammet ovan. Tillståndsövergångarna fås fram ur binärräknarens schema i figuren ovan genom att man för varje värdekomination hos D-vipporna, nuvarande tillstånd, bestämmer  $d_1$  och  $d_0$ , dvs nästa tillstånd för D-vipporna. För exempelvis  $q_1 q_0 = 00$  blir  $d_1 d_0 = 01$ . Tillståndsdiagrammet ovan har ett för räknare karakteristiskt utseende, tillstånden ligger i en ring. I blocksymbolen har räknaren getts namnet *Cnt2b*, där *Cnt* står för Counter, 2 för antalet bitar och *b* för att räknaren räknar binärt.

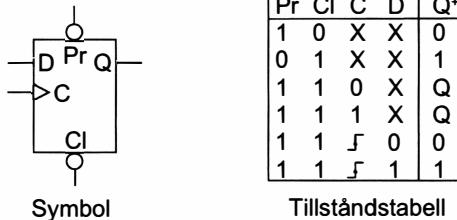
I figuren nedan visas binärräknarens utsignaler i tidsplanet. Vi ser här att frekvensen hos utsignalen  $q_0$  är hälften av klocksignalens frekvens och att frekvensen hos utsignalen  $q_1$  är en fjärdedel av klocksignalens frekvens. *Frekvensdelare* (eng. *frequency divider*) är också ett användningsområde för räknare.



Figur 2.47 Tidsdiagram för 2-bitars binärräknaren *Cnt2b* ovan.

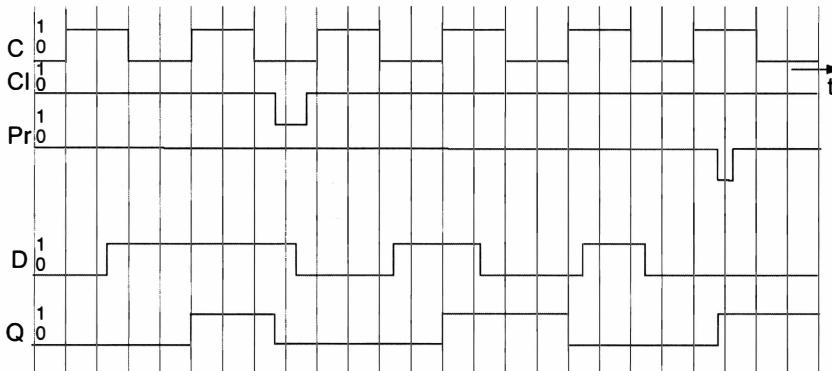
## Nollställning

D-vippan är normalt försedd med *asynkrona* insignalen för nollställning (eng. *Clear*, *Cl*) och ettställning (eng. *Preset*, *Pr*) enligt figuren nedan. Normalt är signalerna aktivt Låga, som i detta fall.



Figur 2.48 D-vippa med signaler för ettställning (*Pr*) och nollställning (*Cl*).

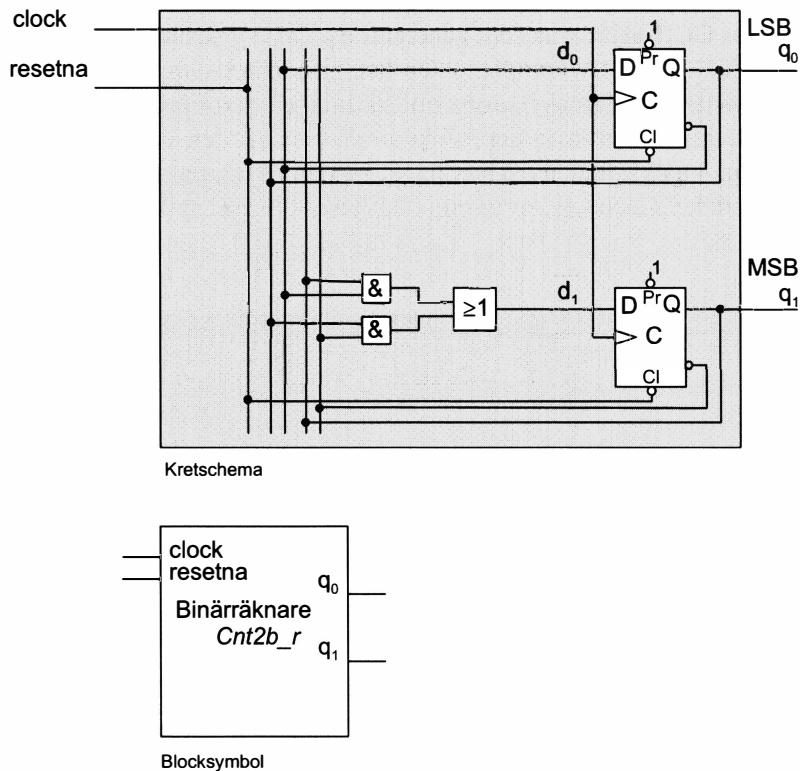
Insignalerna *Preset* och *Clear* är *asynkrona*, vilket innebär att de påverkar D-vippan utan medverkan av klocksignalen. Insignalen D är ju en synkron insignal, klocksignalens aktiva flank krävs för att verkställa inmatning av D-signallens värde. I figuren nedan visas funktionen i tidsplanet.



Figur 2.49 D-vippans funktion i tidsplanet med signalerna Preset och Clear.

### Asynkron nollställning av räknaren

Räknaren i figur 2.45 kan enkelt förses med asynkron signal *resetna* för nollställning enligt figuren nedan.

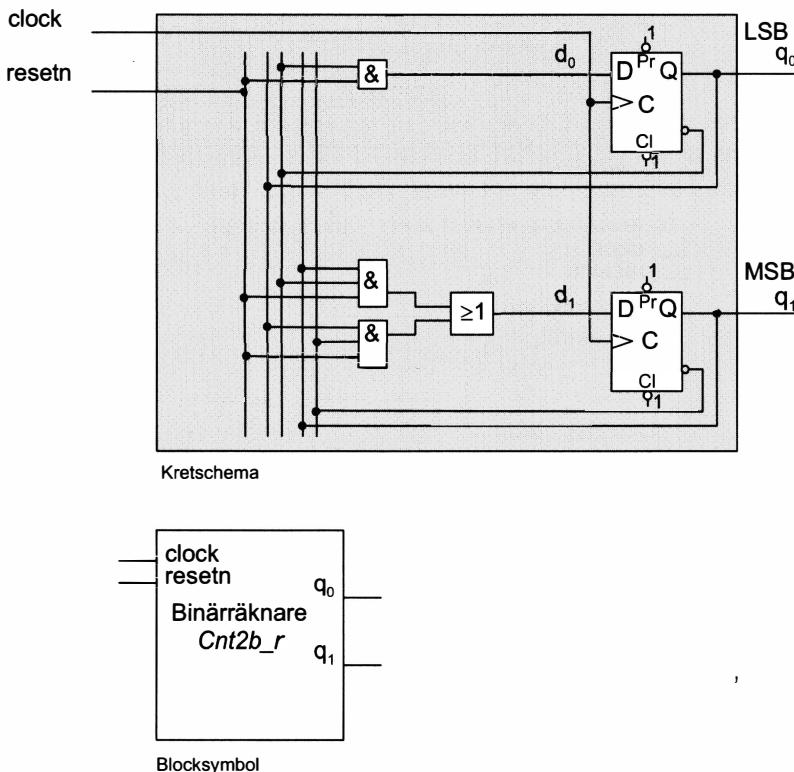


Figur 2.50 2-bitars binärräknare, som räknar modulo-4 med asynkron resetna.

Nollställning av räknaren sker sålunda med  $\text{resetna} = 0$ , utan medverkan av klocksignalen. I signalnamnet *resetna* har speciella egenskaper hos signalen markerats, 'n' ("not") anger att signalen är aktiv låg och 'a' att den är asynkron. I schemat har till de icke använda ingångarna Pr anslutits 1 (Hög) för att poängterta att man ej skall lämna dem öppna.

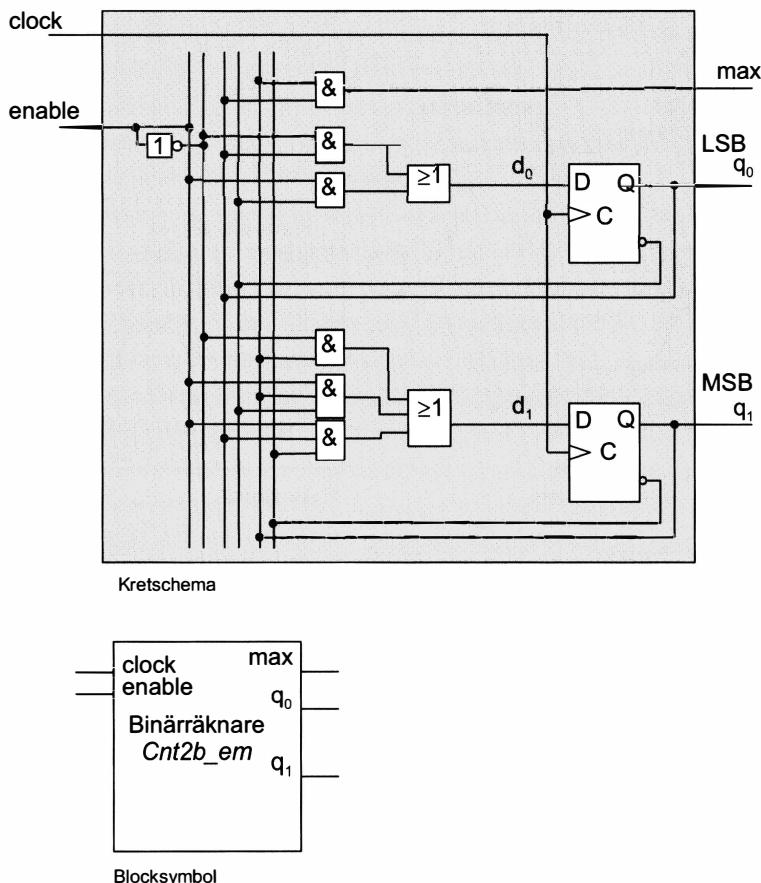
### Synkron nollställning av räknaren

En synkron signal *resetn* (vi markerar inte i signalnamnet att den är synkron) kan införas i räknaren enligt figuren nedan. Eftersom signalen är synkron måste den påverka insignalerna d till D-vipporna. För *resetn* = 0 skall sålunda båda d-signalerna bli 0, medan för *resetn* = 1 skall d-signalerna inte påverkas. Problemet löses med en OCH-grind för signalen  $d_0$  och extra ingångar hos OCH-grindarna som genererar  $d_1$ . Den synkrona nollställningen ger alltså en ökning av komplexiteten hos grindnätet som genererar d-signalerna, vilket inte den asynkrona nollställningen ger. Fördelen med synkron nollställning är att man har full kontroll över när den skall ske, den kan bara ske på klocksignalens aktiva flank, medan asynkron nollställning kan ske var som helst och t.ex. av någon oönskad störning ”spik”.



Figur 2.51 2-bitars binärräknare, som räknar modulo-4 med synkron *resetn*.

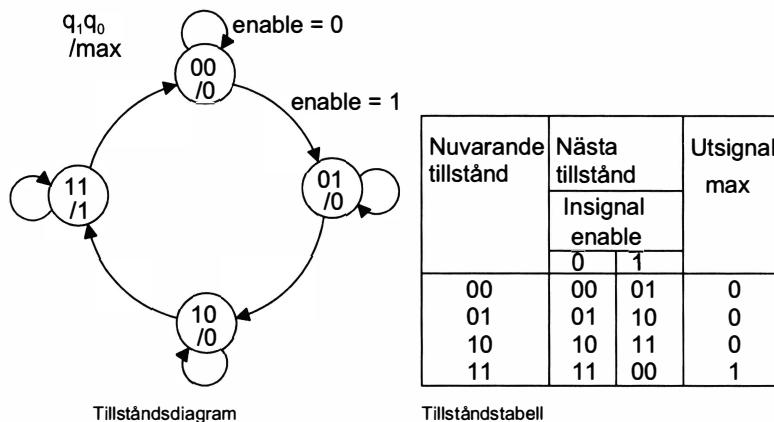
Räknare kan förses med diverse andra in- och utsignaler. En vanlig insignal är *enable*, som styr om räknaren skall ”räkna” eller ”inte räkna”. Räknaren i figuren nedan är den ursprungliga räknaren *Cnt2b* som försetts med *enable* sådan att  $\text{enable} = 0$  medför att räknaren inte räknar och  $\text{enable} = 1$  att den räknar. Räknaren har också försetts med utsignal *max*, som indikerar att räknaren är i sitt maxläge, dvs 11.



Figur 2.52 Binärräknaren *Cnt2b*, försedd med insignal *enable* och utsignal *max*.

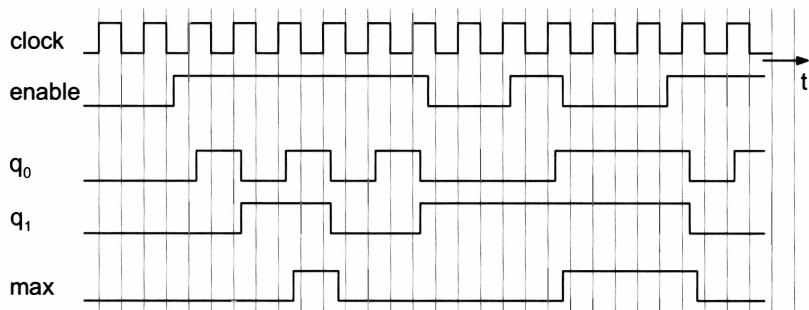
I schemat för räknaren *Cnt2b\_em* ovan ser vi att  $\text{max} = q_1 q_0$ , dvs att  $\text{max} = 1$  om och endast om  $q_1 q_0 = 11$ . Vidare ser vi att för  $\text{enable} = 0$  är för  $d_0$  den övre OCH-grinden öppen och den nedre stängd, innebärande att  $d_0 = q_0$ , och för  $d_1$  den övre OCH-grinden öppen och de två nedre stängda,

innebärande att  $d_1 = q_1$  och sålunda sammantaget att  $d_1 d_0 = q_1 q_0$ , dvs att nästa tillstånd = nuvarande tillstånd, dvs räknaren räknar inte. För enable = 1 är de övre OCH-grindarna hos  $d_0$  och  $d_1$  stängda, medan övriga OCH-grindar öppna och  $d_0$  och  $d_1$  får samma värden som i den tidigare räknaren, dvs räknaren räknar. Tillståndsdiagram och tillståndstabell blir enligt nedan. Utsignalen max, som endast beror av tillståndet, har skrivits inne i tillståndscirklarna som /max och tillfogats som en kolumn i tillståndstabellen.



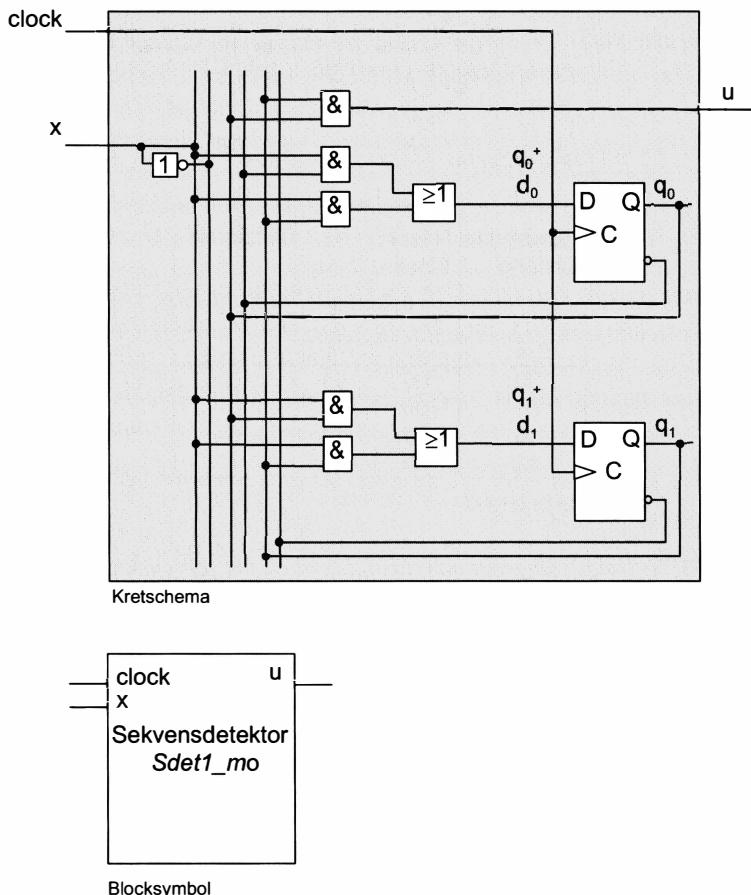
Figur 2.53 Tillståndsdiagram och tillståndstabell för räknaren Cnt2b\_em.

Nedan visas exempel på ett tidsdiagram för räknaren Cnt2b\_em.



Figur 2.54 Tidsdiagram för räknaren Cnt2b\_em.

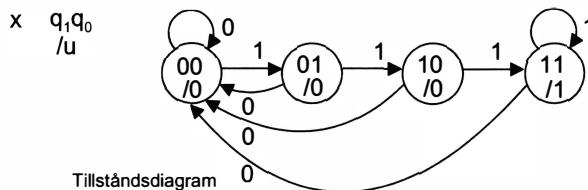
## Några generella sekvenskretsar



Figur 2.55 Sekvenskretsen *Sdet1\_mo*, en sekvenskrets av typ Moore.

Tillståndsdiagram och tillståndstabell för sekvenskretsen *Sdet1\_mo* visas nedan. Sekvenskretsen är benämnd *sekvensdetektor* och har konstruerats för att i en godtyckligt lång insekvens i  $x$ , detektera förekomster av en delsekvens bestående av minst tre på varandra följande ettor. Utsignalen  $u$  blir 1 när delsekvensen detekteras, annars är  $u = 0$ . Observera att när  $x$  varit

1 i tre på varandra följande klocksignalperioder, blir u lika med 1 i nästa klocksignalperiod. Starttillstånd är 00.

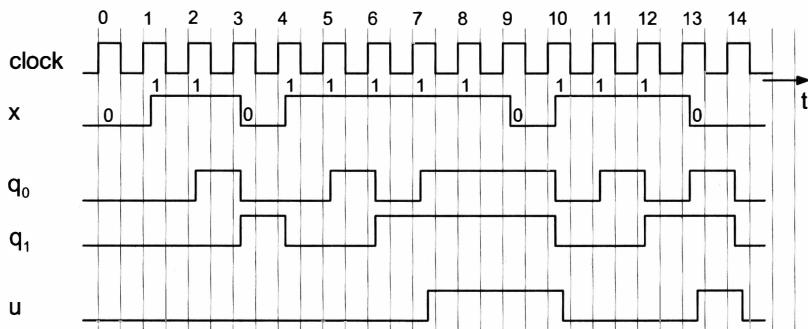


Nuvarande tillstånd $q_1q_0$	Nästa tillstånd $q_1^+q_0^+$		Utsignal u	
	Insignal x			
	0	1		
00	00	01	0	
01	00	10	0	
10	00	11	0	
11	00	11	1	

Tillståndstabell

Figur 2.56 Tillståndsdiagram och tillståndstabell för sekvenskretsen Sdet1\_mo.

Som framgår av kretsschemat och tillståndsdiagrammet så beror utsignalen  $u$  endast av nuvarande tillstånd och ej av insignalen  $x$ . Sekvenskretsar för vilka detta gäller benämnes typ *Moore* [Moore, E. F., "Gedanken Experiments on Sequential Machines," i C. E. Shannon och J. McCarthy (Ed.), *Automata Studies*, Princeton Univ. Press, Princeton, N.J., 1956]. Nedan visas tidsdiagram för en insekvens  $x$  lika med 0110111101110.



Figur 2.57 Tidsdiagram för sekvenskretsen Sdet1\_mo.

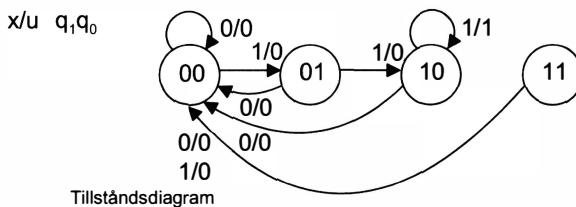
Det förtjänar att påpekas att beteendet för en kombinationskrets kan beskrivas i en sanningstabell, där utgångsvärdet anges för varje insignal kombination, men att detta inte är möjligt för en sekvenskrets, eftersom *i en sekvenskrets samma insignalvärde vid olika tidpunkter kan ge olika utsignalvärdet*. Vi ser ju för sekvenskretsen *Sdet1\_mo* ovan hur  $x = 1$  ibland ger  $u = 0$  och ibland  $u = 1$  och hur  $x = 0$  ibland ger  $u = 0$  och ibland  $u = 1$ .

#### Klocksignal-

intervall	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x	0	1	1	0	1	1	1	1	1	0	1	1	1	0	0
u	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0

För att kunna bestämma utsignalen till en sekvenskrets vid en godtycklig tidpunkt räcker det sålunda inte med att veta insignalen utan man måste också veta tillståndet. I tillståndstabellen används beteckningen  $q^+$  för nästa tillstånd, som ju är detsamma som insignalen d till D-vippan som också visas i kretsschemat.

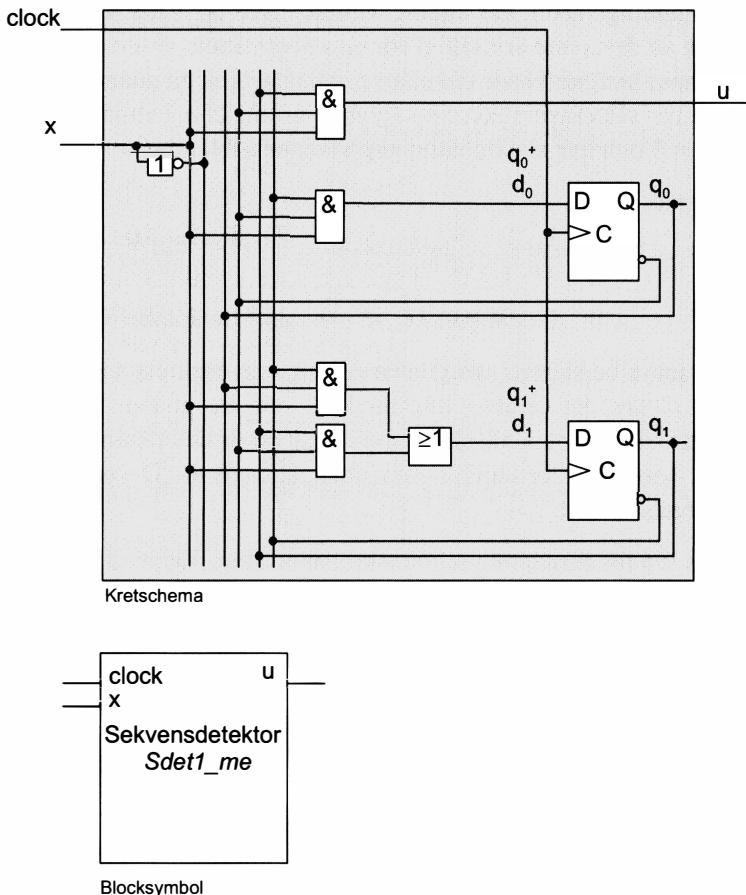
Det finns två fundamentala sekvenskretsmodeller, *Moore* som vi sett exempel på ovan, och *Mealy* [Mealy, G. H., "A Method for Synthesizing Sequential Circuits," *Bell System Tech. J.*, 34, No 5, 1045-1080 (September, 1955).], som skall visas nedan.



$q_1 q_0$	Nuvarande tillstånd		$q_1 q_0^+ / u$	
	Nästa tillstånd / Utsignal u			
	X			
	0	1		
00	00/0	01/0		
01	00/0	10/0		
10	00/0	10/1		
11	00/0	00/0		

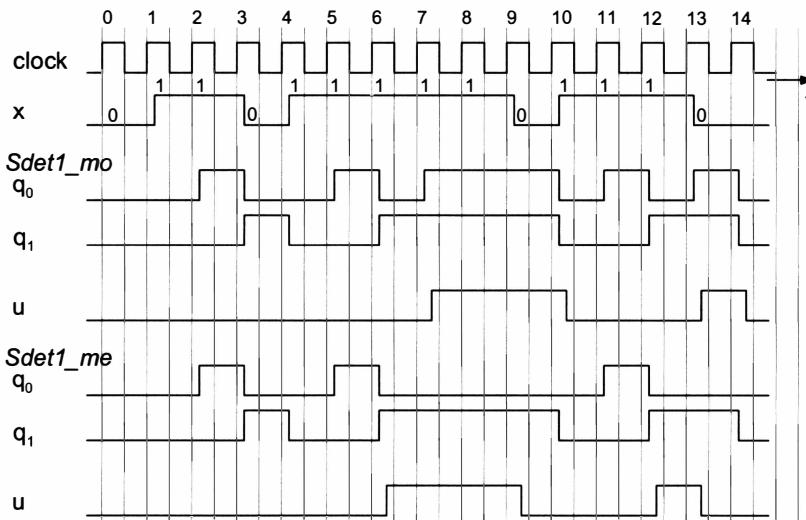
Tillståndstabell

Figur 2.58 Tillståndsdiagram och tillståndstabell för sekvenskretsen *Sdet1\_me*.


 Figur 2.59 Sekvenskretsen  $Sdet1\_me$ , en sekvenskrets av typ Mealy.

Ur kretsschemat får att  $u = xq_1q_0'$ , dvs utsignalen  $u$  beror av insignalen  $x$ . Detta är karakteristiskt för en sekvenskrets av typ *Mealy*. Under hela klock-pulsperioden kan insignalerna påverka utsignalerna i Mealy-modellen, vilket inte är möjligt i Moore-modellen där utsignalerna endast beror av tillståndet. I tillståndsdiagrammet till en sekvenskrets av typ Mealy skriver vi utsignalerna tillsammans med insignalerna på pilarna åtskilda med ett snedstreck (/). Tolka beteckningen på pilen rätt! Exempelvis för tillståndet 10 gäller att ”då nuvarande tillstånd är 10 så är  $u = 1$  om  $x = 1$  och  $u = 0$  om

$x = 0''$ , vilket gäller under hela klocksignalperioden tills nästa positiva flank, då tillståndsändring sker. Utsignalen som anges på pilen erhålls alltså inte på tillståndsövergången, utan under hela tiden i det aktuella tillståndet för den tillhörande insignalen. I tillståndstabellen för en sekvenskrets av typ Mealy skrivs utsignalen tillsammans med nästa tillstånd åtskild med ett snedstreck (/) inne i tillståndstabellen. Nedan visas tidsdiagram för en insekvens  $x$  lika med 0110111101110 till sekvenskretsen  $Sdet1\_me$ , samma insekvens som till sekvenskretsen av typ Moore,  $Sdet1\_mo$  vars signaler också visas igen i diagrammet nedan för jämförelse.



Figur 2.60 Tidsdiagram för sekvenskretsarna  $Sdet1\_me$  och  $Sdet1\_mo$ .

#### Klocksignal-

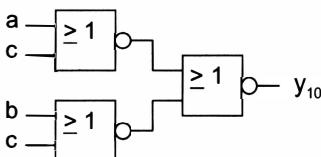
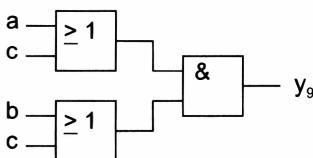
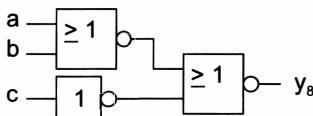
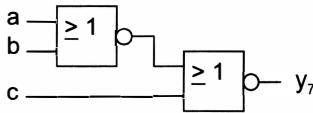
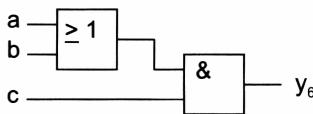
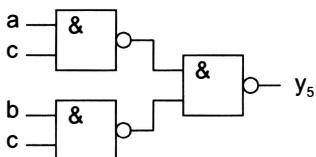
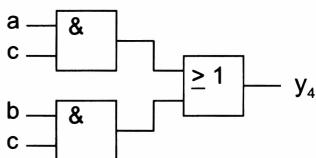
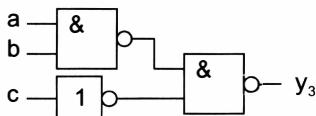
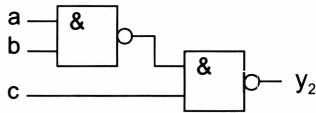
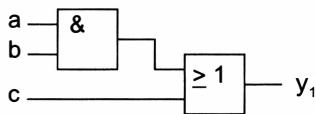
intervall	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x	0	1	1	0	1	1	1	1	1	0	1	1	1	0	0
$u \ Sdet1\_mo$	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0
$u \ Sdet1\_me$	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0

I sekvenskretsen  $Sdet\_me$  används endast tre tillstånd för att realisera beteendet, men sekvenskretsen har fyra tillstånd, eftersom totala antalet tillstånd är en 2-potens av antalet D-vippor. Sekvenskretsen har konstruerats så att det icke använda tillståndet 11 har starttillståndet 00 som nästa tillstånd.

## 2.3 Övningsuppgifter

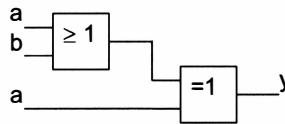
### 2.1 Grindar och kombinationskretsar

- 2.1 a)** Skriv upp logiska uttrycken för grindnäten nedan. Använd här och i fortsättningen symbolerna ( $\cdot$ ), (+) och ( $'$ ) för OCH, ELLER respektive ICKE, där punkten för OCH ej skrivs ut.



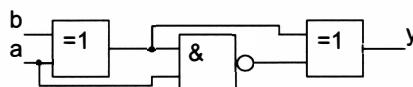
- b)** Bestäm sanningstabellerna till grindnäten ovan.

**2.2 a)** Skriv upp logiska uttrycket för grindnätet nedan.



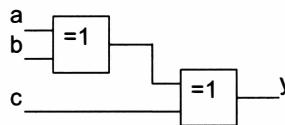
**b)** Bestäm sanningstabellen för grindnätet.

**2.3 a)** Skriv upp logiska uttrycket för grindnätet nedan.



**b)** Bestäm sanningstabellen för grindnätet.

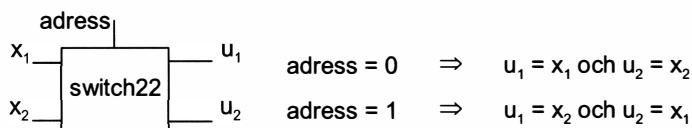
**2.4 a)** Skriv upp logiska uttrycket för grindnätet nedan.



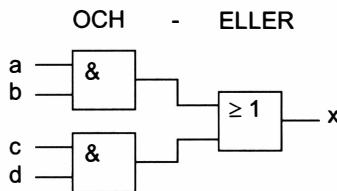
**b)** Bestäm sanningstabellen för grindnätet.

**c)** Utgående från sanningstabellen, ge en beskrivning av grindnätets funktion.

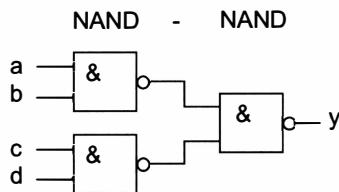
**2.5** Konstruera med OCH-, ELLER-grindar och Inverterare ett grindnät, *switch22*, med två ingångar  $x_1$  och  $x_2$ , två utgångar  $u_1$  och  $u_2$  och beteende enligt nedan.



**2.6 a)** Skriv upp logiska uttrycket för OCH-ELLER-nätet nedan.



**b)** Skriv upp logiska uttrycket för NAND-NAND-nätet nedan.



**c)** Bestäm sanningstabellerna för grindnäten i a) och b).

**d)** Vad gäller för sanningstabellerna till OCH-ELLER-nätet och NAND-NAND-nätet? Formulera något lämpligt samband mellan OCH-ELLER-nät och NAND-NAND-nät. Formulera samband mellan nättens logiska uttryck?

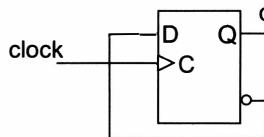
**2.7 a)** Skriv upp logiska uttrycket till data\_ut i MUX 4-1 i figur 2.13.  
**b)** Skriv upp logiska uttrycken till data\_ut(0), data\_ut(1), data\_ut(2) och data\_ut(3) i DMUX 1-4 i figur 2.20.

**2.8 a)** Om man byter plats på  $a_0$  och  $a_1$  i MUX 4-1 i figur 2.15, hur skall då insignalerna numreras uppifrån och ner?  
**b)** Om man byter plats på  $a_0$  och  $a_1$  i DMUX 1-4 i figur 2.22, hur skall då utsignalerna numreras uppifrån och ner?

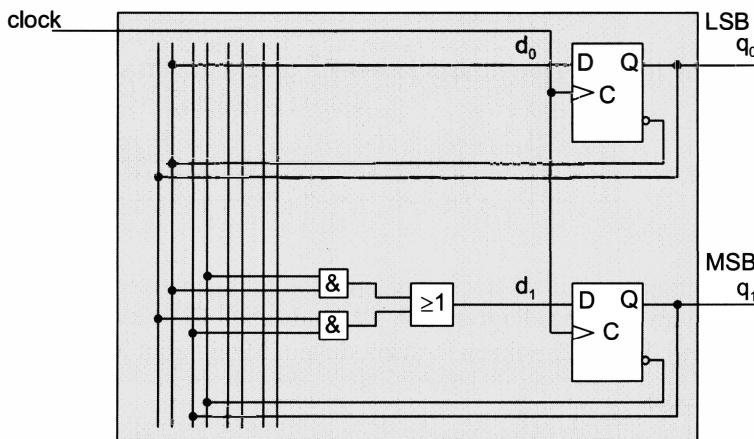
**2.9** Bestäm maximala fördröjningen från ingång till utgång för grindnäten i övningsuppgifterna 2.1–2.4.  
Antag följande fördröjningar för grindarna.  
OCH: 1,4ns, ELLER: 1,3 ns, NAND: 0,8 ns, NOR: 0,9 ns,  
XOR: 1,9 ns, inverterare: 0,6 ns.

## 2.2 Vippor och sekvenskretsar

**2.10** Rita klocksignal och utsignal q i tidsplanet för D-vippan nedan.

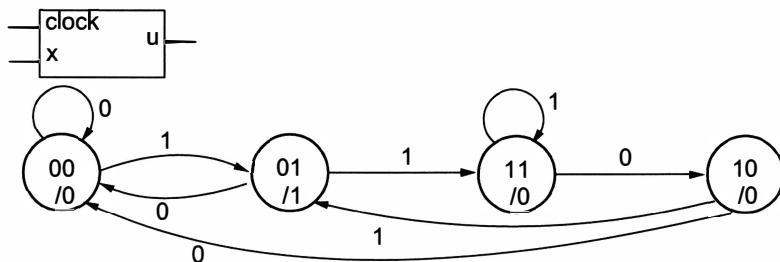


**2.11 a)** Rrita tillståndsdiagram för sekvenskretsen nedan.

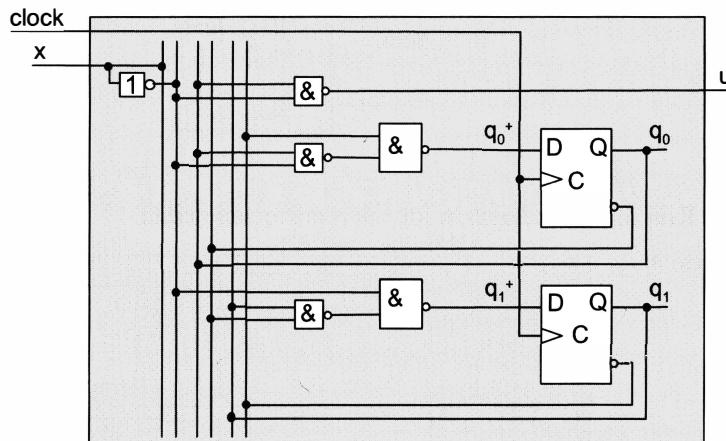


**b)** Utgående från tillståndsdiagrammet, ge en sammanfattande beskrivning av beteendet.

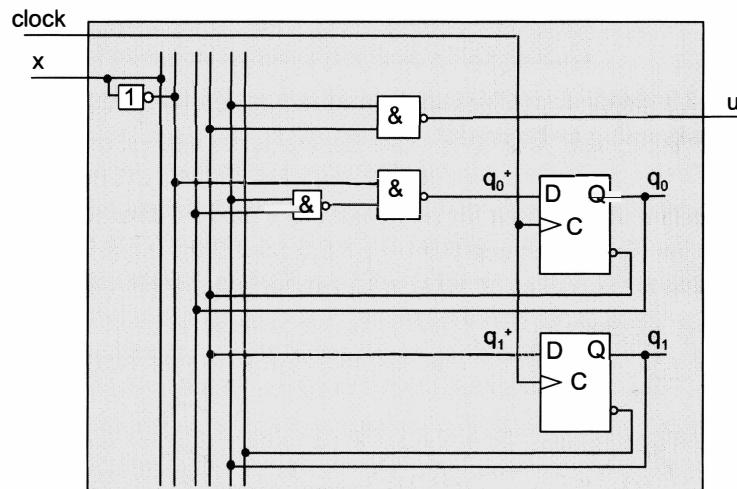
**2.12** Bestäm utsekvensen för sekvenskretsen med tillståndsdiagrammet nedan för insekvensen  $0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0$ . Starttillstånd är 00. Vilken är sekvenskretsmodellen, Mealy eller Moore?



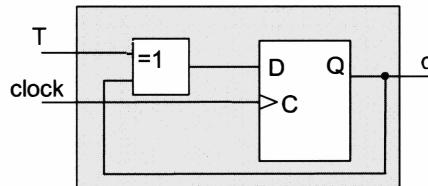
**2.13** Bestäm tillståndsdiagram och tillståndstabell för sekvenskretsen nedan. Vilken är sekvenskretsmodellen, Mealy eller Moore?



**2.14** Bestäm tillståndsdiagram och tillståndstabell för sekvenskretsen nedan. Vilken är sekvenskretsmodellen, Mealy eller Moore?

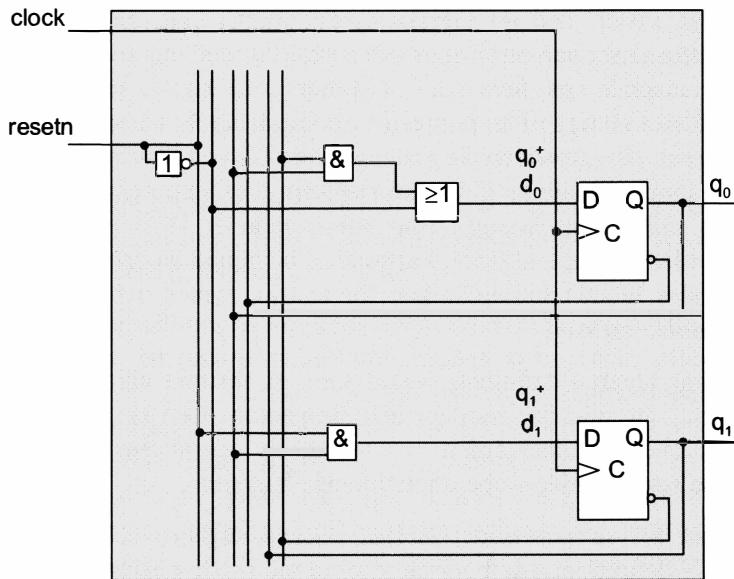


**2.15 a)** Bestäm tillståndsdiagram för sekvenskretsen nedan.



- b)** Vilken är sekvenskretsmodellen, Mealy eller Moore?
- c)** Formulera utgående från tillståndsdiagrammet en verbal beskrivning av sekvenskretsens beteende.
- d)** Antag sekvenskretsen startas i tillståndet 0. Ange utsekvensen för insekvensen 0 0 0 1 1 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1 .

**2.16** Bestäm tillståndsdiagram och tillståndstabell för sekvenskretsen nedan.



# 3 Boolesk algebra

I detta kapitel presenteras boolesk algebra och exemplifieras användning av axiom och räknelagar för omformning av logiska uttryck. Boolesk algebra är ett oumbärligt hjälpmittel för analys och syntes av digitala kretsar.

Boolesk algebra har fått sitt namn efter George Boole (1815–64), engelsk matematiker och logiker, som lade grunden till den moderna symboliska logiken. I uppsatserna *Mathematical analysis of logic* (1847) och *Laws of thought* (1854) utvecklade han en logisk algebra för matematisk behandling av logiska satser och relationer. I vårt dagliga språk kan vi formulera satser, påståenden, som kan vara sanna eller falska. Satserna kan sättas samman till nya satser med de logiska operationerna *och*, *eller* och *icke*. Sammansatta satser kan omformas och förenklas med lagarna i logisk algebra. I programspråk, t.ex. Java och C, använder man logiska satser och relationer. Den första viktiga tillämpningen av boolesk algebra inom digitalteknik kan tillskrivas den amerikanske matematikern Claude Shannon (f 1916), främst känd som grundare av informationsteorin (i vilken ingår det i kapitel 1 citeraade *samplingsteoremet*). Han publicerade år 1938 en uppsats där han visade hur boolesk algebra, i uppsatsen benämnd *switching algebra*, kunde användas inom telefoniken för analys, syntes och förenkling av nät med reläkontakter.

Boolesk algebra definieras nedan som ett abstrakt algebraiskt system, på samma sätt som den vanliga heltalsalgebran, med en mängd av element (konstanter, variabelvärden), en mängd av operationer och en mängd av axiom som definierar operationerna på elementen.

För att belysa sambandet mellan boolesk algebra och symbolisk logik, anges i definitionen av boolesk algebra vid sidan av konstanterna och operationerna, motsvarande inom symbolisk logik, Falsk och Sann respektive ELLER, OCH och ICKE. Sätt in dessa logiska konstanter och operationer i axiomen och verifiera att axiomen överensstämmer med vanligt språkbruk.

## 3.1 Boolesk algebra – definition

*Konstanter (variabelvärden)*

0 (Falsk)

1 (Sann)

*Operationer*

+ (ELLER)

· (OCH)

‘ (ICKE)

*Axiom*

$0 + 0 = 0$  (A1)

$1 \cdot 1 = 1$  (A2)

$1 + 1 = 1$  (A3)

$0 \cdot 0 = 0$  (A4)

$0 + 1 = 1 + 0 = 1$  (A5)

$1 \cdot 0 = 0 \cdot 1 = 0$  (A6)

$0' = 1$  (A7)

$1' = 0$  (A8)

En trivial illustration till sambandet mellan boolesk algebra och symbolisk logik kan fås genom att i axiom (A8) sätta in logiska konstanter och operationer, varvid erhålls: ICKE(Sann) = Falsk.

I den fortsatta framställningen kommer operationerna ICKE, ELLER och OCH, normalt att betecknas enligt definitionen ovan, dvs. med apostrofstecken, prim, ( ‘ ), plustecken ( + ) respektive multiplikationstecken ( · ). Som nämntes i kapitel 2 förekommer andra beteckningar. För operationen ICKE används bl.a. snedstreck ( / ), utropstecken ( ! ) och ordet NOT, samt speciellt inom symbolisk logik, tecknet (  $\neg$  ). För operationen ELLER används inom symbolisk logik vanligen tecknet (  $\vee$  ), som är första bokstaven i det latinska ordet ”vel” (sv. eller). Detta tecken kommer att användas när förväxling med vanlig addition kan befaras. För operationen OCH kommer multiplikationstecknet ( · ) normalt att utlämnas, som brukligt är vid multiplikation. När tecknet för operationen OCH måste sättas ut,

används ofta något av tecknen ( \* ), ( & ), ordet *and* eller tecknet (  $\wedge$  ), det sistnämnda tecknet speciellt inom symbolisk logik, vilket inte har något samband med ordet ”och”, utan bara är tecknet (  $\vee$  ) för ELLER vänt upp och ned.

Operationerna ( + ) och ( · ) i boolesk algebra har alltså inget att göra med vanlig addition och multiplikation, men det hindrar inte att vi kan konstatera likheten mellan axiomen (A1) - (A6), bortsett från (A3), och motsvarande axiom i den vanliga heltalsalgebran. Studerar vi vidare axiomen ser vi att de grupperats parvis. Detta beror på att det råder ett enkelt samband mellan axiomen inom ett par, det ena axiomet kan erhållas ur det andra genom att man byter 0 och 1 samt operationerna ( + ) och ( · ). Man säger att axiomen inom ett par är *duala*, och denna dualitetsprincip gäller generellt inom boolesk algebra, *gäller en likhet, så gäller också den duala likheten*.

## 3.2 Räknelagar

### Räknelagar för en variabel

$$x + x = x \quad (\text{L1})$$

$$x \cdot x = x \quad (\text{L2})$$

$$x + x' = 1 \quad (\text{L3})$$

$$x \cdot x' = 0 \quad (\text{L4})$$

$$x + 1 = 1 \quad (\text{L5})$$

$$x \cdot 0 = 0 \quad (\text{L6})$$

$$x + 0 = x \quad (\text{L7})$$

$$x \cdot 1 = x \quad (\text{L8})$$

$$(x')' = x \quad (\text{L9})$$

Liksom i vanlig heltalsalgebra kan i boolesk algebra införas variabler. Dessa kan bara anta värdena 0 och 1, eftersom booleska algebran bara innehåller konstanterna 0 och 1. Ur axiomen kan satser (teorem) för variabler visas. Dessa satser kan användas som ”räknelagar” för boolesk algebra.

Räknelagarna (L1) - (L9) kan enkelt visas med s.k. perfekt induktion, dvs genom att man låter variablerna genomlöpa samtliga värden och verifierar varje fall med ett axiom. Låt oss t.ex. visa (L3).

### Bevis av (L3)

$$\begin{array}{lll} x = 0 \text{ i (L3) ger} & 0 + 0' = 0 + 1 & (\text{enl. A7}) \\ & = 1 & (\text{enl. A5}) \\ x = 1 \text{ i (L3) ger} & 1 + 1' = 1 + 0 & (\text{enl. A8}) \\ & = 1 & (\text{enl. A5}) \end{array}$$

□

## Räknelagar för flera variabler

$$x + (y + z) = (x + y) + z \quad (\text{L10}) \text{ (associativa lagarna)}$$

$$x(yz) = (xy)z \quad (\text{L11})$$

$$x + y = y + x \quad (\text{L12}) \text{ (kommutativa lagarna)}$$

$$xy = yx \quad (\text{L13})$$

$$x(y + z) = xy + xz \quad (\text{L14}) \text{ (distributiva lagarna)}$$

$$x + yz = (x + y)(x + z) \quad (\text{L15})$$

$$x + xy = x \quad (\text{L16}) \text{ (absorptionslagarna)}$$

$$x(x + y) = x \quad (\text{L17})$$

$$xy + x'z = xy + x'z + yz \quad (\text{L18}) \text{ (consensuslagarna)}$$

$$(x + y)(x' + z) = (x + y)(x' + z)(y + z) \quad (\text{L19})$$

$$(x + y)' = x'y' \quad (\text{L20}) \text{ (De Morgans lagar)}$$

$$(xy)' = x' + y' \quad (\text{L21})$$

Räknelagarna (L6) – (L8) och (L10) – (L14) överensstämmer med motsvarande lagar i heltalsalgebran. Distributiva lagen (L14) läst från vänster till höger känner vi igen som ”multiplicera in” och läst från höger till vänster som ”bryta ut”. Den distributiva lagen (L15) gäller dock inte i vanlig heltalsalgebra (verifiera med ett exempel med heltal!). Räknelagarna är inte alla oberoende, utan vissa räknelagar kan visas med hjälp av de övriga räknelagarna. Sådana kan räknelag (L15) visas med hjälp av (L14), (L2), (L16) och (L13) enligt nedan.

**Bevis av (L15)**

$$\begin{aligned}(x + y)(x + z) &= (x + y)x + (x + y)z && (\text{enl. L14}) \\&= x(x + y) + z(x + y) && (\text{enl. L13}) \\&= xx + xy + zx + zy && (\text{enl. L14}) \\&= x + xy + zx + zy && (\text{enl. L2}) \\&= x + zx + zy && (\text{enl. L16}) \\&= x + xz + yz && (\text{enl. L13}) \\&= x + yz && (\text{enl. L16})\end{aligned}$$

□

Låt oss illustrera användningen av räknelagarna med ett litet exempel. För att inte lösningarna skall bli onödigt omfattande utelämnas här och i fortsättningen triviala mellanled som t.ex. användning av kommutativa lagarna.

**Exempel 3.1**

Förenkla så långt möjligt uttrycket  $z'(1 + zx)$ .

**Lösning**

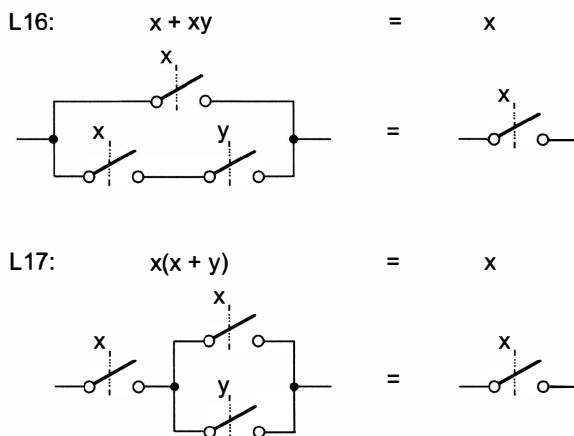
$$\begin{aligned}z'(1 + zx) &= z' \cdot 1 + z' \cdot zx && (\text{enl. L14}) \\&= z' + 0 \cdot x && (\text{enl. L8,L4}) \\&= z' + 0 && (\text{enl. L6}) \\&= z' && (\text{enl. L7})\end{aligned}$$

□

Absorptionslagarna, consensuslagarna och De Morgans lagar är viktiga räknelagar som förtjänar speciell uppmärksamhet. En verbal tolkning av dessa lagar underlättar förståelse och memorering.

Absorptionslagen (L16) kan tolkas som ”om i en summa av två produkter, (i detta fall  $x$  och  $xy$ ;  $x$  är en trivial produkt av bara en variabel), den ena produkten är en del av den andra produkten, (i detta fall är  $x$  del av  $xy$ ), så får den längsta produkten strykas”, ty summan av produkterna är ju enligt (L16) lika med  $x$ . Orden ”summa” och ”produkt” här avser självklart ELLER-summa och OCH-produkt. Räknelagarna är angivna för enskilda variabler, men gäller också med variablerna ersatta med godtyckligt stora logiska uttryck. Så kan t.ex. absorptionslagen tillämpas på logiska uttrycket  $yz' + xyz'w$ , där alltså produkten  $yz'$  är en del av produkten  $xyz'w$  och sålunda kan den längre produkten strykas och uttrycket reduceras till  $yz'$ .

Absorptionslagarna (L16) och (L17) kan åskådliggöras med switchar enligt figuren nedan. I det vänstra nätet till (L16) så är förbindelsen i nedre grenen sluten bara om switchen  $x$  är sluten, men då är ju förbindelsen i övre grenen också sluten. Den undre grenen är alltså onödig och näset till vänster kan ersättas med näset till höger, som absorptionslagen (L16) säger. I det vänstra näset till (L17) så är förbindelsen sluten bara om switchen  $x$  till vänster är sluten, men då är ju switchen  $x$  i den övre parallella grenen också sluten. De parallella grenarna är alltså onödiga och näset till vänster kan ersättas med näset till höger, som absorptionslagen (L17) säger.



Figur 3.1 Absorptionslagen illustrerad med switchar.

Consensuslagen (L18) kan tolkas som ”om i en summa av två produkter, (i detta fall  $xy$  och  $x'z$ ), finns en variabel som är icke-inverterad i den ena produkten och inverterad i den andra produkten, (i detta fall  $x$  och  $x'$ ), så får till uttrycket adderas produkten av de i de två produktorna övriga ingående variablerna (i detta fall  $yz$ )”. Det kan tyckas konstigt att man vill göra ett logiskt uttryck längre, men det är faktiskt så man normalt använder consensuslagen, dvs från vänster till höger, och inte från höger till vänster, där uttrycket blir kortare. Ofta används consensuslagen och absorptionslagen tillsammans omväxlande vid förenkling av uttryck, som visas i exempel 3.2 nedan, där consensuslagen först ger ett längre uttryck, men sedan absorptionslagen ger ett kortare och enklare uttryck.

**Exempel 3.2**

Visa likheten  $x + x'y = x + y$ .

Lösning

$$\begin{aligned}x + x'y &= x \cdot 1 + x'y && (\text{enl. L8}) \\&= x \cdot 1 + x'y + 1 \cdot y && (\text{enl. L18}) \\&= x + x'y + y && (\text{enl. L8}) \\&= x + y && (\text{enl. L16})\end{aligned}$$

□

**Exempel 3.3**

Förenkla uttrycket  $yz' + xyz + x'z$ .

Lösning

$$\begin{aligned}yz' + xyz + x'z &= yz' + xyz + x'z + yz && (\text{enl. L18}) \\&= yz' + x'z + yz && (\text{enl. L16}) \\&= y(z' + z) + x'z && (\text{enl. L14}) \\&= y \cdot 1 + x'z && (\text{enl. L3}) \\&= y + x'z && (\text{enl. L8})\end{aligned}$$

□

Lägg speciellt märke till i exemplet ovan förenklingen  $yz' + yz = y(z' + z) = y \cdot 1 = y$ , där man alltså bryter ut och erhåller en parentes av typen  $(z' + z)$ , som är lika med 1. Denna förenklingsprincip är viktig och vi kommer att möta den i samband med förenkling av logiska uttryck med Karnaughdiagram i kapitel 4.

**Exempel 3.4**

Förenkla uttrycket  $xy' + x'y + xy$ .

Lösning

$$\begin{aligned}xy' + x'y + xy &= xy' + x'y + xy + xy && (\text{enl. L1}) \\&= x(y' + y) + y(x' + x) && (\text{enl. L14}) \\&= x \cdot 1 + y \cdot 1 && (\text{enl. L14}) \\&= x + y && (\text{enl. L8})\end{aligned}$$

□

Notera i exemplet ovan hur termen  $xy$  i uttrycket adderas till uttrycket en gång till. Det är tillåtet att till ett uttryck addera termer i uttrycket hur många gånger som helst, p.g.a. räknelagen (L1):  $x + x = x$ .

De Morgans lagar (L20) och (L21) från vänster till höger kan tolkas som ”inverstecknet utanför en parentes får flyttas in på de enskilda variablerna (uttrycken) om samtidigt operationen mellan variablerna (uttrycken) bytes, ( $+$ ) mot ( $\cdot$ ) och tvärtom”.

De Morgans lagar kan utvidgas till ett godtyckligt antal variabler (uttryck) enligt:

$$(x_1 + x_2 + \dots + x_n)' = x_1' x_2' \dots x_n' \quad (\text{L20a})$$

$$(x_1 x_2 \dots x_n)' = x_1' + x_2' + \dots + x_n' \quad (\text{L21a})$$

### Exempel 3.5

Omforma uttrycket  $(x'(yz)' + xy)'$  till ett uttryck där inverstecknen verkar direkt på de enskilda variablerna.

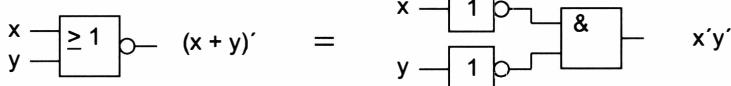
Lösning

$$\begin{aligned} (x'(yz)' + xy)' &= (x'(yz))' (xy)' && (\text{enl. L20}) \\ &= (x + (yz))(x' + y') && (\text{enl. L21, L9}) \\ &= xx' + x'y'z + xy' + y'yz && (\text{enl. L14}) \\ &= x'y'z + xy' && (\text{enl L4, L6, L7}) \end{aligned}$$

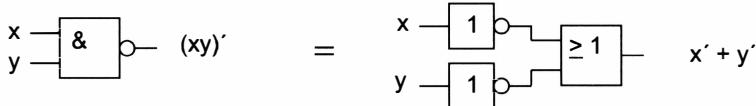
□

De Morgans lagar kan illustreras med grindar enligt figuren nedan.

L20:

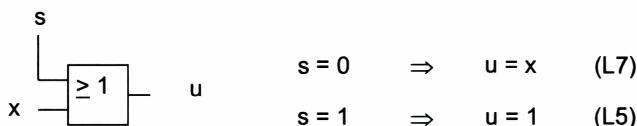
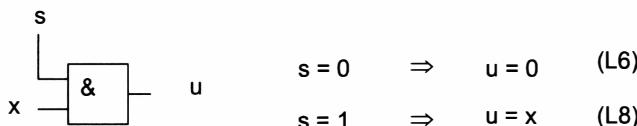
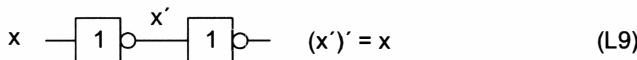


L21:



Figur 3.2 De Morgans lagar illustrerade med grindar.

I figuren nedan illustreras några andra räknelagrar med grindar. – Om insignalen  $s$  i figuren tolkas som en styrsignal, så kan OCH-grinden betraktas som stängd för  $s = 0$ , då utsignalen  $u = 0$  oavsett värdet hos insignalen  $x$ , och betraktas som öppen för  $s = 1$ , då utsignalen  $u = x$  och alltså  $x$  kan passera genom grinden. Namnet grind är här som synes relevant. För ELLER-grinden ser vi att den är stängd för  $s = 1$ , då utsignalen  $u = 1$  oavsett värdet hos insignalen  $x$ , medan den är öppen för  $s = 0$ .



Figur 3.3 Några räknelagrar illustrerade med grindar.

### 3.3 Operationen XOR (Exklusivt-ELLER)

I kapitel 2 visades en XOR-grind och gavs olika tolkningar av operationen XOR. Denna operation har stor betydelse i digitaltekniken och vi skall nedan kort beröra några räknelagar. Operationen XOR ingår inte i booleska algebran, men vi skall se hur den kan beskrivas med operationer i booleska algebran och tvärtom hur operationen OR i booleska algebran kan beskrivas med operationen XOR.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Figur 3.4 Sanningstabell för operationen XOR.

### Räknelagar

Ur sanningstabellen ovan fås direkt räknelagarna (E1) – (E3) för konstanterna 0 och 1.

#### Räknelagar för konstanterna 0 och 1

$0 \oplus 0 = 0$	(E1)
$0 \oplus 1 = 1 \oplus 0 = 1$	(E2)
$1 \oplus 1 = 0$	(E3)

Ur räknelagarna för konstanterna 0 och 1 ovan fås räknelagarna nedan för en och flera variabler. Lagarnas riktighet kan man verifiera på samma sätt som i booleska algebran genom att sätta in samtliga värden på variablerna och tillämpa räknelagarna för konstanterna 0 och 1 ovan.

#### Räknelagar för en variabel

$x \oplus 0 = x$	(E4)
$x \oplus x = 0$	(E5)

## Räknelagar för flera variabler

$x \oplus (y \oplus z) = (x \oplus y) \oplus z$	(E6) (associativa lagen)
$x \oplus y = y \oplus x$	(E7) (kommutativa lagen)
$x(y \oplus z) = xy \oplus xz$	(E8) (distributiva lagen)
$(x \oplus y = x \oplus z) \Leftrightarrow (y = z)$	(E9)

Räknelag (E9) kan jämföras med motsvarande lag för addition i heltalsalgebran. (E9) från höger till vänster innebär att det är tillåtet att till båda sidor av en likhet addera xor samma storhet, medan (E9) från vänster till höger innebär att det är tillåtet att på båda sidor om en likhet stryka samma storhet.

Med hjälp av räknelagarna (E9) och (E5) kan ekvationer lösas på samma sätt som i heltalsalgebran enligt principen i exemplet nedan.

### Exempel 3.6

Lös ekvationen  $x \oplus b = a$ .

*Lösning*

$$\begin{aligned} x \oplus b &= a \\ \Rightarrow x \oplus b \oplus b &= a \oplus b \quad (\text{enl. E9}) \\ \Rightarrow x \oplus 0 &= a \oplus b \quad (\text{enl. E5}) \\ \Rightarrow x &= a \oplus b \quad (\text{enl. E4}) \end{aligned}$$

□

### Omvandling från XOR till operationer i Boolesk algbra och tvärtom

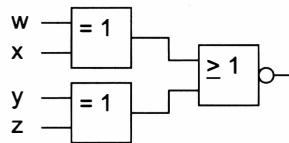
$x \oplus y = x'y + xy'$	(E10)
$(x \oplus y)' = x'y' + xy$	(E11)
$x + y = x \oplus y \oplus xy$	(E12)
$x \oplus 1 = x'$	(E13)

**Exempel 3.7**

Omforma uttrycket  $w'x'y'z' + w'x'yz + wxy'z' + wxyz$  så att det kan realiseras i ett grindnät innehållande två XOR-grindar och en NOR-grind.

**Lösning**

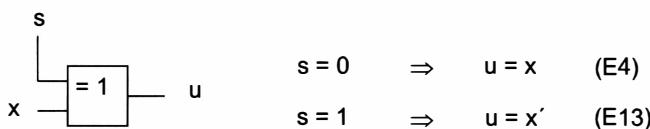
$$\begin{aligned}
 w'x'y'z' + w'x'yz + wxy'z' + wxyz &= w'x'(y'z' + yz) + wx(y'z' + yz) \\
 &= (w'x' + wx)(y'z' + yz) \\
 &= (w \oplus x)'(y \oplus z)' \\
 &= ((w \oplus x) + (y \oplus z))'
 \end{aligned}$$



Figur 3.5 Grindnät till exempel 3.7.

□

Räknelagarna (E4) och (E13) ger en intressant tillämpning av XOR-grinden som styrbar inverterare, visad i figuren nedan (tidigare visad i kapitel 2).



Figur 3.6 XOR-grinden som styrbar inverterare.

Med insignalen  $s$  i figuren ovan betraktad som en styrsignal, så gäller att för  $s = 0$  så går insignalen  $x$  opåverkad genom XOR-grinden, medan för  $s = 1$  så inverteras insignalen  $x$ . Denna tillämpning av XOR-grinden används, som vi kommer att se längre fram, bl. a. i programmerbara grindnät för att ge icke-inverterad alternativt inverterad utsignal.

**Exempel 3.8**

Omforma logiska uttrycket

$$1 \oplus c \oplus abc,$$

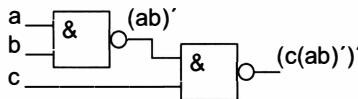
med hjälp av lagarna i booleska algebran och lagar för XOR, så att det kan realiseras i ett grindnät med två 2-ingångars NAND-grindar. Insignaler till grindnätet är a, b och c.

*Lösning*

$$1 \oplus c \oplus abc = (c \oplus abc)' \quad (\text{enl. E13})$$

$$= (c(1 \oplus ab))' \quad (\text{enl. E8})$$

$$= (c(ab)')' \quad (\text{enl. E13})$$



Figur 3.7 Grindnät till exempel 3.8

□

**Exempel 3.9**

Omforma logiska uttrycket

$$((a \oplus b)' + (a + b')' + a'b)'$$

med hjälp av lagarna i booleska algebran och lagar för XOR, så att det kan realiseras i ett grindnät med en 2-ingångars NOR-grind och en 3-ingångars NOR-grind. Insignaler till grindnätet är a och b.

*Lösning*

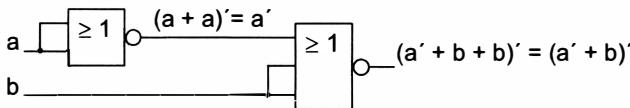
$$((a \oplus b)' + (a + b')' + a'b)' = (a \oplus b)(a + b')(a'b)' \quad (\text{enl. L20})$$

$$= (a'b + ab')(a + b')(a + b') \quad (\text{enl. E10, L21})$$

$$= (a'b + ab')(a + b') \quad (\text{enl. L2})$$

$$= ab' \quad (\text{enl. L4, L7})$$

$$= (a' + b)' \quad (\text{L20, L9})$$



Figur 3.8 Grindnät till exempel 3.9.

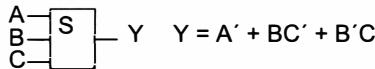
□

## 3.4 Övningsuppgifter

- 3.1** Förenkla logiska uttrycken med hjälp av räknelagarna i booleska algebran.
- $ac'd + ad$
  - $ac'd + acd$
  - $a(b' + a'c + ab)$
  - $ab' + abc$
  - $a + b' + a'b + c'$
  - $a + abc + ad + a'b + ad' + a'bc$
  - $(a + bc')(a'b' + c)$
  - $a'bc' + a'd + bc'd'$
  - $(a + b')(a' + b)(a + b)$
  - $a'b'c' + abc' + a'bc' + ab'c'$
  - $a'b'c + abc + a'bc$
  - $ae + abc'd + bc'e'$
  - $a'b'c' + a'bd' + cd$
  - $a(b + c)'$
  - $a + (a'b)'$
  - $(ab)'(ab)'$
  - $(a' + (a'b + c)')'$
  - $((ab')(a + b))'$
- 3.2** Omforma logiska uttrycken till grindnäten i övningsuppgift 2.1 till summa av produkter.
- 3.3** Visa följande likheter genom tillämpning av booleska algebrans räkne lagar på vänstra ledet så att högra ledet erhålls.
- $(vw + x + y)(x' + y)(x' + y + z) = vwx' + y$
  - $(x + y' + xy')(xy + x'z + yz) = xy + x'y'z$
  - $(x + y' + xy)(x + y')x'y = 0$
  - $wx + w'y + xyz = wx + w'y$
  - $xy' + yz' + x'z = x'y + y'z + xz'$
- (*Ledning:* Använd consensuslagen L18, först tre gånger för att lägga till produkter och därefter tre gånger för att ta bort produkter.)
- $((x + y') + x'z)' = x'yz'$
  - $((xz + (yz)')' + y(xz)')' = y' + xz$

- 3.4** Realisera med ett minimalt antal 2-ingångars NAND- och NOR-grindar (motivera kopplingen med logiskt uttryck).
- a) En 4-ingångars OCH-grind    b) En 4-ingångars ELLER-grind

- 3.5** Visa att med enbart S-grindar (fiktiv grind) är det möjligt att realisera operationerna ICKE, OCH, ELLER samt XOR. Konstanterna 0 och 1 får anslutas som insignal.

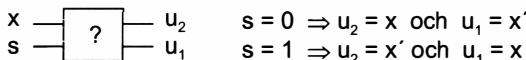


- 3.6** Visa att
- a)  $x \oplus xy = xy'$     b)  $x \oplus (x + y) = x'y$     c)  $(xy + (x + y)')' = x \oplus y$   
 d)  $x' \oplus y = x \oplus y'$     e)  $x \oplus y = x' \oplus y'$     f) not A xor B = A xnor B

- 3.7** Visa att med enbart G-grindar (fiktiv grind) är det möjligt att realisera operationerna ICKE, OCH och ELLER. Konstanterna 0 och 1 är tillgängliga som insignal.



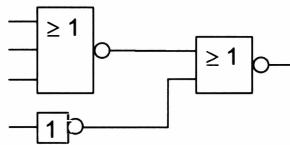
- 3.8** Konstruera med exakt två XOR-grindar och inga andra grindar, ett grindnät med två ingångar x och s och två utgångar  $u_1$  och  $u_2$  enligt specifikationen nedan. Konstanterna 0 och 1 är tillgängliga som insignal.



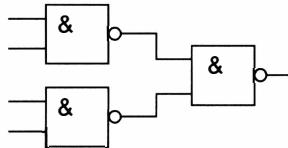
- 3.9** Omforma logiska uttrycket  $(b \oplus c) + bc + a$ , med hjälp av lagarna i booleska algebran och lagar för XOR, så att det kan realiseras i en 4-ingångars ELLER-grind. Insignaler till grindnätet är a, b och c.

- 3.10** Omforma logiska uttrycket  $((a \oplus b)c + b')'$ , med hjälp av lagarna i booleska algebran och lagar för XOR, så att det kan realiseras i ett grindnät med fyra 2-ingångars NAND-grindar. Insignaler till grindnätet är a, b och c.

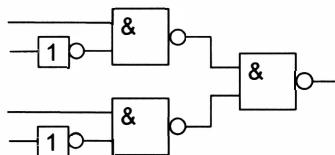
**3.11** Omforma logiska uttrycket  $(ab' + a'b)c + cd(a'b' + ab) + abcd'$ , med hjälp av lagarna i booleska algebran, så att det kan realiseras i grindnätet nedan. Insignaler till grindnätet är a, b, c och d.



**3.12** Omforma logiska uttrycket  $(ac)'(bd(ac))' + ac(bd(ac))'$ , med hjälp av lagarna i booleska algebran, så att det kan realiseras i grindnätet nedan. Insignaler till grindnätet är a, b, c och d.



**3.13** Omforma logiska uttrycket till grändatet i övningsuppgift 2.2 med hjälp av lagarna i booleska algebran och lagar för XOR, så att det kan realiseras i grändatet nedan. Insignaler till grändatet är a och b. Konstanterna 0 och 1 är tillgängliga som insignal.



**3.14** Omforma logiska uttrycket till grändatet i övningsuppgift 2.3 med hjälp av lagarna i booleska algebran och lagar för XOR, så att det kan realiseras med endast två av grändarna (flera varianter är möjliga!). Insignaler till grändatet är a och b. Konstanterna 0 och 1 är tillgängliga som insignal.

# 4 Kombinationskretsar

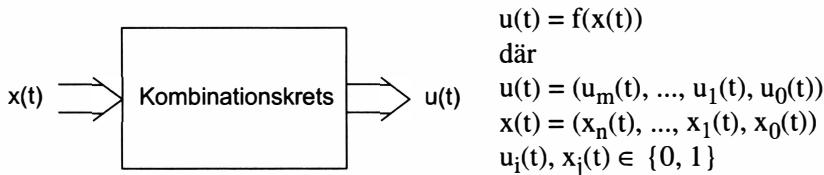
Kombinationskretsar är en klass av kretsar utan minnesfunktion, vilka karakteriseras av att samma insignalvärde vid olika tidpunkter alltid ger samma utsignalvärde. I kapitel 2 studerade vi några kombinationsskretsar uppbyggda med grindar, såväl generella som speciella fundamentala kombinationskretsar såsom multiplexer, demultiplexer och avkodare. Vi ägnade oss huvudsakligen åt *analys*, i detta kapitel skall vi studera *syntes* av kombinationskretsar, hur de utgående från en given specifikation realiseras med ett minimalt antal grindar.

Booleska funktioner är intimt förknippade med kombinationskretsar och vi skall närmare studera normalformer och minimering av booleska funktioner. Minimeringen innebär att man strävar efter att realisera en boolesk funktion i en kombinationskrets med ett minimalt antal grindar. Minimering är naturligtvis önskvärd vid de flesta realiseringar, speciellt viktig är den vid realisering av kretsar i programmerbara logiska kretsar, där den programmerbara kretsens komplexitet är given och definierad av halvledarfabrikanten och man alltså måste se till att få plats med kretsen som skall realiseras i den programmerbara kretsen i kapseln.

Kretskonstruktion görs med datorbaserade syntesverktyg och naturligtvis finns det datorprogram som utför minimering av booleska funktioner. Det har utvecklats nya minimeringsmetoder av heuristisk typ, som kanske inte alltid ger det absolut minimala nätet, men som är minnessnåla och snabba. I utvecklingssystem för programmerbara kretsar ingår alltid program för minimering av booleska funktioner. Icke desto mindre skall vi minimera booleska funktioner för hand med s.k. Karnaughdiagram, dels därför att det ökar förståelsen av booleska funktioner och minimering, dels därför att Karnaughdiagrammet är ett användbart verktyg för minimering av booleska funktioner av upp till sex variabler som varje digitaltekniker måste kunna hantera och som här ger oss möjlighet till enkel minimering av booleska funktioner vid realisering av kombinations- och sekvenskretsar i den fortsatta framställningen.

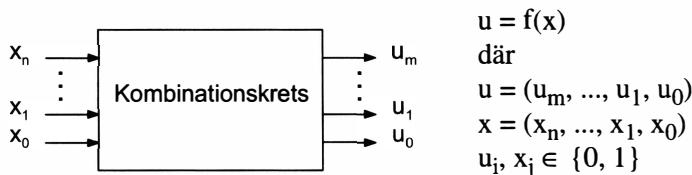
## 4.1 Definition av kombinationskrets

Formellt definieras en *kombinationskrets* (eng. *combinational circuit*) som en krets där utsignalvärdet  $u(t)$  vid en godtycklig tidpunkt endast beror av insignalvärdet  $x(t)$  vid samma tidpunkt..



Figur 4.1 Kombinationskrets, formell definition.

Tidsvariabeln  $t$  behövs ej för att beskriva beteendet hos en kombinationskrets och användes ju heller inte då vi studerade kombinationskretsar i kapitel 2. Vi utelämnar den i fortsättningen och använder modellen i figur 4.2 nedan för en kombinationskrets. Tiden är bara intressant för en kombinationskrets vad gäller fördräjningen mellan insignal och utsignal.



Figur 4.2 Kombinationskrets, modell.

Kombinationskretsens insignaler  $x_0, x_1, \dots, x_n$  och utsignaler  $u_0, u_1, \dots, u_m$  är ovan definierade binära på mängden  $\{0, 1\}$ . Ibland kan de vara definierade på en större mängd, t.ex. vara *ternära* (trevärda) och definierade på mängden  $\{0, 1, Z\}$ , där  $Z$  är det i kapitel 2 tidigare omnämnda signalvärdet *högochmig*. Vi betecknar normalt i fortsättningen insignalen  $x$  såsom  $x = (x_n, \dots, x_1, x_0)$  och utsignalen  $u$  såsom  $u = (u_m, \dots, u_1, u_0)$ , så att mest signifikant bit (MSB) är placerad till vänster och har högst index.

## 4.2 Booleska funktioner

En *boolesk funktion* (eng. *boolean function*)  $f(x_{n-1}, \dots, x_1, x_0)$  av  $n$  variabler är en funktion som bara kan anta värdena 0 och 1 och vars variabler också bara kan anta värdena 0 och 1.

$$f: \{0, 1\}^n \rightarrow \{0, 1\}$$

Exempel på en boolesk funktion är funktionen *lo3* i kretsen BCDcheck, vars funktionstabell visas i tabell 4.1 nedan. Funktionen *lo3* har fyra variabler  $x_0, x_1, x_2, x_3$  och eftersom varje variabel bara kan anta två värden, 0 och 1, så blir antalet variabelkombinationer lika med  $2 \cdot 2 \cdot 2 \cdot 2 = 2^4 = 16$ . För var och en av de 16 variabelkombinationerna kan en godtycklig boolesk funktion av fyra variabler också bara anta två värden, 0 och 1. Totala antalet booleska funktioner av fyra variabler blir därför  $2 \cdot 2 \cdots 2$  ( $2^4$  tvåor)  $= 2^{2^4} = 2^{16} = 65536$ . Funktionen *lo3* är bara en av dessa möjliga booleska funktioner av fyra variabler. Enligt samma resonemang kan vi lätt räkna ut antalet booleska funktioner av  $n$  variabler. Antalet variabelkombinationer blir lika med  $2^n$  och sålunda antalet booleska funktioner av  $n$  variabler lika med  $2^{2^n}$ .

Tabell 4.1: Booleska funktionen *lo3* i kretsen BCDcheck.

<b><math>x_3</math></b>	<b><math>x_2</math></b>	<b><math>x_1</math></b>	<b><math>x_0</math></b>	<b><math>lo3</math></b>
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

I tabellen nedan visas antalet booleska funktioner för några olika värden på antalet variabler  $n$ . Vi kan där se hur antalet funktioner växer mycket snabbt med antalet variabler.

Tabell 4.2: Antalet booleska funktioner för olika antal variabler.

Antal variabler $n$	Antalet booleska funktioner ( $2^{2^n}$ )
0	2
1	4
2	16
3	256
4	65 536
5	4 294 967 296

Låt oss nu se hur man kan skriva upp ett booleskt uttryck till en boolesk funktion specificerad i en funktionstabell. I kapitel 3, Boolesk algebra, har vi sett hur booleska uttryck kan omformas och förenklas och man inser därför att varje boolesk funktion kan beskrivas med många (i princip hur många som helst) olika booleska uttryck av helt olika utseende. Det är dock bara ett fåtal uttryck som är intressanta vid realisering av en boolesk funktion i ett grindnät, speciellt intressanta är booleska uttrycken på s.k. *normalform*.

Varje boolesk funktion kan skrivas på två olika normalformer och låt oss skriva upp dessa till booleska funktionen  $lo3$  i tabell 4.1 ovan. Vi börjar då med att komplettera funktionstabellen med ytterligare tre kolumner enligt tabell 4.3 nedan.

Studerar vi *mintermerna*, ser vi att om den till mintermen  $m_i$  hörande variabelkombinationen  $i$  sätts in i mintermen, så blir den lika med 1, medan den för alla övriga variabelkombinationer blir lika med 0. *Mintermen* är alltså bara 1 för en enda variabelkombination, därav namnet minterm. För *maxtermerna* gäller det omvänta förhållandet. Om den till maxtermen  $M_i$  hörande variabelkombinationen  $i$  sätts in i maxtermen, så blir den lika med 0, medan den för alla övriga variabelkombinationer blir lika med 1. *Maxtermen* är alltså 0 för bara en enda variabelkombination, därav namnet maxterm. Vidare kan konstateras att det råder ett samband mellan samhörande minterm och maxterm, nämligen att  $M_i = m_i'$ . Exempelvis gäller att  $m_3' = (x_3' x_2' x_1 x_0)' = x_3 + x_2 + x_1' + x_0' = M_3$ .

Tabell 4.3: Booleska funktionen lo3 med mintermer och maxtermer.

Variabelkomination nr i	x <sub>3</sub>	x <sub>2</sub>	x <sub>1</sub>	x <sub>0</sub>	lo3	Minterm m <sub>i</sub>	Maxterm M <sub>i</sub>
0	0	0	0	0	1	x <sub>3</sub> 'x <sub>2</sub> 'x <sub>1</sub> 'x <sub>0</sub> '	x <sub>3</sub> +x <sub>2</sub> +x <sub>1</sub> +x <sub>0</sub>
1	0	0	0	1	1	x <sub>3</sub> 'x <sub>2</sub> 'x <sub>1</sub> x <sub>0</sub> '	x <sub>3</sub> +x <sub>2</sub> +x <sub>1</sub> +x <sub>0</sub> '
2	0	0	1	0	1	x <sub>3</sub> 'x <sub>2</sub> 'x <sub>1</sub> x <sub>0</sub> '	x <sub>3</sub> +x <sub>2</sub> +x <sub>1</sub> '+x <sub>0</sub>
3	0	0	1	1	0	x <sub>3</sub> 'x <sub>2</sub> 'x <sub>1</sub> x <sub>0</sub>	x <sub>3</sub> +x <sub>2</sub> +x <sub>1</sub> '+x <sub>0</sub> '
4	0	1	0	0	0	x <sub>3</sub> 'x <sub>2</sub> x <sub>1</sub> 'x <sub>0</sub> '	x <sub>3</sub> +x <sub>2</sub> '+x <sub>1</sub> +x <sub>0</sub>
5	0	1	0	1	0	x <sub>3</sub> 'x <sub>2</sub> x <sub>1</sub> 'x <sub>0</sub>	x <sub>3</sub> +x <sub>2</sub> '+x <sub>1</sub> +x <sub>0</sub>
6	0	1	1	0	0	x <sub>3</sub> 'x <sub>2</sub> x <sub>1</sub> x <sub>0</sub> '	x <sub>3</sub> +x <sub>2</sub> '+x <sub>1</sub> '+x <sub>0</sub>
7	0	1	1	1	0	x <sub>3</sub> 'x <sub>2</sub> x <sub>1</sub> x <sub>0</sub>	x <sub>3</sub> +x <sub>2</sub> '+x <sub>1</sub> '+x <sub>0</sub>
8	1	0	0	0	0	x <sub>3</sub> x <sub>2</sub> 'x <sub>1</sub> 'x <sub>0</sub> '	x <sub>3</sub> '+x <sub>2</sub> +x <sub>1</sub> +x <sub>0</sub>
9	1	0	0	1	0	x <sub>3</sub> x <sub>2</sub> 'x <sub>1</sub> 'x <sub>0</sub>	x <sub>3</sub> '+x <sub>2</sub> +x <sub>1</sub> +x <sub>0</sub>
10	1	0	1	0	0	x <sub>3</sub> x <sub>2</sub> 'x <sub>1</sub> x <sub>0</sub> '	x <sub>3</sub> '+x <sub>2</sub> +x <sub>1</sub> '+x <sub>0</sub>
11	1	0	1	1	0	x <sub>3</sub> x <sub>2</sub> 'x <sub>1</sub> x <sub>0</sub>	x <sub>3</sub> '+x <sub>2</sub> +x <sub>1</sub> '+x <sub>0</sub>
12	1	1	0	0	0	x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> 'x <sub>0</sub> '	x <sub>3</sub> '+x <sub>2</sub> '+x <sub>1</sub> +x <sub>0</sub>
13	1	1	0	1	0	x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> 'x <sub>0</sub>	x <sub>3</sub> '+x <sub>2</sub> '+x <sub>1</sub> +x <sub>0</sub>
14	1	1	1	0	0	x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> x <sub>0</sub> '	x <sub>3</sub> '+x <sub>2</sub> +x <sub>1</sub> '+x <sub>0</sub>
15	1	1	1	1	0	x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> x <sub>0</sub>	x <sub>3</sub> '+x <sub>2</sub> '+x <sub>1</sub> '+x <sub>0</sub>

En **minterm** (**maxterm**) till en boolesk funktion  $f(x_{n-1}, \dots, x_2, x_1)$  är alltså en **boolesk produkt (summa)** av *samtliga* variabler, där variabeln antingen är icke-inverterad eller inverterad. Mintermen (maxtermen) är 1 (0) för en och endast en variabelkombination.

Med hjälp av **mintermerna** och **maxtermerna** kan man nu lätt skriva upp booleska uttryck till en boolesk funktion. Uppenbart kan funktionen lo3 i tabell 4.3 ovan skrivas som

$$\begin{aligned} \text{lo3} &= f(x_3, x_2, x_1, x_0) = m_0 + m_1 + m_2 \\ &= x_3'x_2'x_1'x_0' + x_3'x_2'x_1'x_0 + x_3'x_2'x_1x_0' \end{aligned} \quad (4.1)$$

Funktionen lo3 har skrivits som en boolesk *summa* av de **mintermer** för vilka lo3 = 1, vilket innebär att lo3 blir lika med 1 för de och endast de variabelkombinationer, vars mintermer finns med i booleska uttrycket. Booleska uttrycket till lo3 är en *summa av produkter, SP-form*, och efter-

som varje produkt är en minterm benämnes uttrycket *SP-normalform*. Således gäller att *SP-normalformen till en boolesk funktion f är booleska summan av de mintermer för vilka f = 1*.

SP-normalformen till en boolesk funktion f(x<sub>n-1</sub>, ..., x<sub>1</sub>, x<sub>0</sub>) av n variabler kan komprimerat skrivas som

$$f(x_{n-1}, \dots, x_1, x_0) = \sum_{i=0}^{2^n - 1} f_i \cdot m_i \quad (4.2)$$

Uttrycket (4.2) är en boolesk summa av samtliga mintermer vardera multiplicerad (boolesk produkt) med tillhörande funktionsvärde, innebärande att SP-normalformen blir summan av de mintermer för vilka funktionsvärdet f<sub>i</sub> är lika med 1.

I stället för att beskriva en boolesk funktion i en funktionstabell kan det vara bekvämt att bara ange numret för de variabelkombinationer för vilka funktionen har värdet 1. Exempelvis kan funktionen lo3 beskrivas som

$$\text{lo3} = f(x_3, x_2, x_1, x_0) = \sum(0, 1, 2)$$

Vi har nu sett hur en boolesk funktion kan skrivas på SP-normalform med mintermerna och p.g.a. booleska algebrans dualitetsprincip förväntar vi oss naturligtvis en dual form som är precis tvärtom, dvs. skriven med maxtermerna. Funktionen lo3 skriven på normalform med maxtermerna blir

$$\text{lo3} = f(x_3, x_2, x_1, x_0)$$

$$= M_3 \cdot M_4 \cdot M_5 \cdot M_6 \cdot M_7 \cdot M_8 \cdot M_9 \cdot M_{10} \cdot M_{11} \cdot M_{12} \cdot M_{13} \cdot M_{14} \cdot M_{15}$$

Denna normalform är booleska produkten av de maxtermer för vilka funktionen har värdet 0. Uppenbart blir funktionen lo3 skriven med maxtermerna ovan lika med noll då en maxterm i produkten är lika med 0, vilket inträffar för den till maxtermen hörande variabelkombinationen. Normalformen med maxtermer är en produkt av summor, *PS-form*, och eftersom summorna är maxtermer benämnes den *PS-normalform*.

PS-normalformen till en boolesk funktion f(x<sub>n-1</sub>, ..., x<sub>1</sub>, x<sub>0</sub>) av n variabler kan komprimerat skrivas som

$$f(x_{n-1}, \dots, x_1, x_0) = \prod_{i=0}^{2^n - 1} (f_i + M_i) \quad (4.3)$$

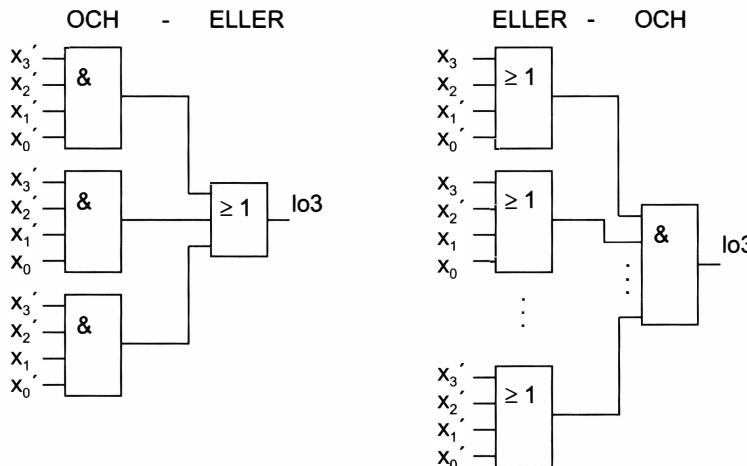
I booleska produkten (4.3) gäller att om  $f_i = 0$ , så blir  $f_i + M_i = 0 + M_i = M_i$ , dvs. maxtermen kommer att ingå i produkten, medan om  $f_i = 1$  så blir  $f_i + M_i = 1 + M_i = 1$ , dvs. maxtermen kommer inte att ingå i produkten.

PS-normalformen kan också användas till att kompakt beskriva en boolesk funktion med numren för de variabelkombinationer för vilka funktionen är lika med 0. Funktionen lo3 kan beskrivas som

$$lo3 = f(x_3, x_2, x_1, x_0) = \prod(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)$$

Självklart är de nummer som ingår i PS-normalformen de som saknas i SP-normalformen.

SP-normalformen som är en summa av produkter, realiseras naturligt i ett grindnät av typ OCH-ELLER, och PS-normalformen som är en produkt av summor, realiseras lika naturligt i ett grindnät av typ ELLER-OCH. I figur 4.3 nedan visas realiseringen av SP-normalformen och PS-normalformen till funktionen lo3. Bortsett från en eventuell första nivå av inverterare som krävs för att invertera insignalerna, så har grindnäten *två nivåer* grindar, en nivå av OCH-grindar och en nivå av ELLER-grindar. Ur födröjningssynpunkt är två-nivå-näten fördelaktiga och normalt realiseras booleska funktioner i två-nivå-nät. I detta fall där funktionen lo3 har avsevärt fler nollor än ettor så blir ELLER-OCH-nätet självklart mer komplext och innehåller 13 ELLER-grindar i första nivån och en OCH-grind med lika många ingångar i andra nivån.



Figur 4.3 SP- och PS-normalformerna till funktionen lo3 realiserade i grindnät.

Vi avslutar nu denna sektion med att studera samtliga booleska funktioner  $f(x, y)$  av två variabler. Det finns enligt tabell 4.2 ovan totalt 16 sådana funktioner, dvs. inte fler än att vi kan skriva upp dem i en tabell enligt nedan.

*Tabell 4.4: Samtliga booleska funktioner av två variabler.*

x	y	$f_0$	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$f_6$	$f_7$	$f_8$	$f_9$	$f_{10}$	$f_{11}$	$f_{12}$	$f_{13}$	$f_{14}$	$f_{15}$
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

De 16 booleska funktionerna  $f_0, f_1, \dots, f_{15}$  kan skrivas som booleska uttryck enligt nedan (prova själv att ta fram dem genom att skriva upp SP- eller PS-normalformerna och sedan tillämpa räknelagarna i booleska algebran).

$$f_0 = 0$$

$$f_1 = xy \text{ (OCH)}$$

$$f_2 = xy'$$

$$f_3 = xy' + xy = x$$

$$f_4 = x'y$$

$$f_5 = x'y + xy = y$$

$$f_6 = x'y + xy' \text{ (XOR)}$$

$$f_7 = x + y \text{ (ELLER)}$$

$$f_8 = x'y' = (x + y)' \text{ (NOR)}$$

$$f_9 = x'y' + xy \text{ (XNOR)}$$

$$f_{10} = x'y' + xy' = y'$$

$$f_{11} = x + y'$$

$$f_{12} = x'y' + x'y = x'$$

$$f_{13} = x' + y'$$

$$f_{14} = x' + y' = (xy)'$$

$$f_{15} = 1$$

## 4.3 Förenkling och realisering av booleska funktioner i grindnät

Låt oss börja med att algebraiskt förenkla funktionen  $lo3$ , som i föregående sektion skrevs på SP-normalform i uttryck (4.1).

$$lo3 = \begin{matrix} 0 & 1 & 2 \\ x_3'x_2'x_1'x_0' + x_3'x_2'x_1'x_0 + x_3'x_2'x_1x_0' \end{matrix} \quad (4.4)$$

$$\begin{aligned} &= \begin{matrix} 0+1 & 2 \\ x_3'x_2'x_1'(x_0' + x_0) + x_3'x_2'x_1x_0' \end{matrix} \\ &= \begin{matrix} 0+1 & 2 \\ x_3'x_2'x_1' + x_3'x_2'x_1x_0' \end{matrix} \end{aligned} \quad (4.5)$$

Principen för förenklingen ovan är: *ur två produkttermer som skiljer sig i bara en variabel, som är icke-inverterad i den ena termen och inverterad i den andra, bryter man ut den gemensamma delen.* Termerna 0 och 1 ovan skiljer sig i bara  $x_0$ -variabeln och den gemensamma delen  $x_3'x_2'x_1'$  kan brytas ut. Eftersom  $x_0' + x_0 = 1$ , så kan alltså de två termerna 0 och 1 slås samman till termen  $x_3'x_2'x_1'$ .

Det förenklade uttrycket (4.5) ovan för  $lo3$  har bara två produkttermer jämfört med SP-normalformen (4.4) som har tre, innehårande att  $lo3$  kan realiseras i ett OCH-ELLER-nät med bara två OCH-grindar i första nivån.

Om man låter ett syntesverktyg generera en minimal SP-form till funktionen  $lo3$  så får man

$$lo3 = x_3'x_2'x_1' + x_3'x_2'x_0' \quad (4.6)$$

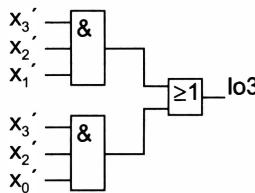
Jämför vi uttrycken (4.5) och (4.6) ser vi att båda uttrycken innehåller lika många produkttermer, men att uttrycket (4.5) har en variabel mer i andra produkttermen, motsvarande en ingång mer hos OCH-grinden som realiseras denna produktterm. Det går således att förenkla uttrycket (4.5) ytterligare och låt oss göra det.

I normalformen (4.4) har produkttermerna 0 och 1 kombinerats, men vi ser att det också skulle vara möjligt att kombinera produkttermerna 0 och 2 som skiljer sig bara i variabeln  $x_1$ . Produkttermen 0 skulle sålunda behöva kombineras med två produkttermer, vilket också är möjligt. Booleska algebrans räknelag (L1):  $x + x = x$ , säger nämligen att en term kan upprepas hur många gånger som helst utan att det ursprungliga uttrycket förändras. En

produktterm får alltså upprepas godtyckligt många gånger i en boolesk summa och förenklingen av lo3 kan därför göras enligt nedan.

$$\text{lo3} = x_3'x_2'x_1'x_0' + x_3'x_2'x_1'x_0 + x_3'x_2'x_1x_0 \quad (4.7)$$

$$\begin{aligned} &= x_3'x_2'x_1'(x_0' + x_0) + x_3'x_2'x_0'(x_1' + x_1) \\ &= x_3'x_2'x_1' + x_3'x_2'x_0' \end{aligned} \quad (4.8)$$



Figur 4.4 Det förenklade uttrycket (4.8) till lo3 realiseras i ett OCH-ELLER-nät.

## Karnaughdiagram

Förenklingen av SP-normalformen var i detta fall enkel, normalt blir det omfattande algebraiska beräkningar och framförallt är det svårt att avgöra vilka termer som skall kombineras för att det enklaste uttrycket skall erhållas. I stället för att göra algebraisk förenkling av SP-normalformen kan man emellertid använda s.k. *Karnaughdiagram*, där den mänskliga hjärnans fina förmåga att känna igen olika mönster kan utnyttjas för att bestämma vilka termer som skall kombineras till det enklaste booleska uttrycket. Karnaugh-diagram är utmärkta hjälpmedel vid förenkling för hand av booleska funktioner upp till ungefär sex variabler, vid större antal variabler blir det svårt även för vår hjärna att urskilja optimala mönster. Förutom som rent förenklingsverktyg är Karnaughdiagram också av stort pedagogiskt värde för förståelsen av booleska funktioner och minmering.

Vi har tidigare sett hur en boolesk funktion kan beskrivas i en funktionstabell. Karnaghdiagrammet är i princip bara funktionstabellen i form av ett rutmönster. Rutmönstret skall ha lika många rutor som antalet variabelkombinationer, dvs. som antalet rader i funktionstabellen, och i varje ruta skall funktionsvärdet anges. Låt oss rita ett Karnaughdiagram till den tidigare funktionen  $lo_3$  med funktionstabellen 4.1.

Funktionen  $lo_3$  är en funktion av fyra variabler och sålunda  $2^4 = 16$  variabelkombinationer, varför Karnaughdiagrammet skall ha 16 rutor. Karnaughdiagrammet blir enligt figur 4.6 nedan.

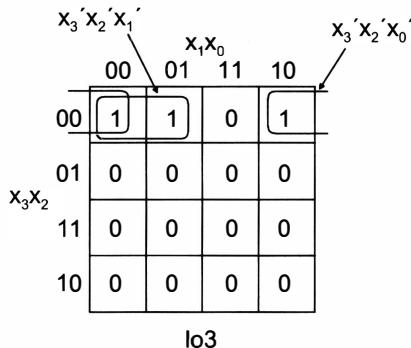
		$x_1x_0$				
		00	01	11	10	
		00	1	1	0	1
		01	0	0	0	0
		11	0	0	0	0
		10	0	0	0	0

$lo_3$

Figur 4.5 Karnaughdiagram till funktionen  $lo_3$ .

Vid den tidigare algebraiska förenklingen av SP-normalformen till funktionen  $lo_3$ , kombinerades termer som skilde sig i bara en variabel och den gemensamma delen bröts ut. Finessen med Karnaughdiagrammet är att det är konstruerat så att *rutor som ligger intill varandra horisontellt eller vertikalt har variabelkombinationer som skiljer sig i bara en variabel*. Vi ser i diagrammet ovan hur rutorna i övre raden från vänster representerar variabelkombinationerna 0000, 0001, 0011 och 0010, dvs. binärtalen skiljer sig ruta för ruta i bara en variabel (observera att 0011 placeras före 0010 för att detta skall gälla). Vertikalt skiljer sig också binärtalen för rutorna bara i en variabel. Vidare skiljer sig radernas respektive kolumnernas ytterrutor också i bara en variabel. För att det direkt skulle framgå att ytterrutorna skiljer sig i bara en variabel så borde diagrammet ritas på en toroid.

Karnaughdiagrammet ger oss alltså möjlighet att lätt se om två mintermer kan slås samman, då skall tillhörande 1:or ligga intill varandra horisontellt eller vertikalt. För funktionen  $lo_3$  i Karnaughdiagrammet i figur 4.5 ovan, ser vi direkt att produkttermerna hörande till rutorna 0000 och 0001 kan slås samman och så även produkttermerna hörande till rutorna 0000 och 0010 eftersom ytterrutorna också ligger intill varandra. Sammanslagningen markeras genom att de två 1:orna inringas enligt figuren nedan.



Figur 4.6 Inringningar i Karnaghdiagrammet till funktionen  $lo_3$ .

Vid den algebraiska förenklingen av  $lo_3$  såg vi vid sammanslagningen av produkttermerna att variabeln som skiljer sig i de båda termerna försvinner p.g.a. att  $x + x' = 1$ . Sammanslagningen i Karnaughdiagrammet ovan av ettorna i rutorna 0000 och 0001 resulterar i termen  $x_3'x_2'x_1'$ , dvs. variabeln  $x_0$  försvinner eftersom den skiljer sig för de båda 1:orna i sammanslagningen, den är 0 för ruta 0000 och 1 för ruta 0001.

*Regeln* för att skriva upp produkttermen till en inringning av 1:or blir således, att endast en variabel som har samma värde för hela inringningen skall ingå i produkten och ingå som inverterad om den har värdet 0 för hela inringningen och icke-inverterad om den har värdet 1 för hela inringningen.

**Exempel 4.1**

	$x_0$	0	1
$x_1$	0	1	0
1	1	1	1

$$f(x_1, x_0) = x_1 + x_0'$$

□

**Exempel 4.2**

	$x_1 x_0$	00	01	11	10
$x_2$	0	1	1	1	0
1	0	0	0	0	1

$$f(x_2, x_1, x_0) = x_2' x_1' + x_2' x_0 + x_2 x_1 x_0'$$

□

**Exempel 4.3**

	$x_1 x_0$	00	01	11	10
$x_3 x_2$	00	0	0	0	1
01	1	1	0	0	0
11	1	1	0	0	0
10	0	0	0	0	1

$$f(x_3, x_2, x_1, x_0) = x_2 x_1' + x_2' x_1 x_0'$$

□

I exempel 4.3 ovan har fyra ettor inringats. Inringning av de två ettorna i översta raden av denna inringning ger termen  $x_3' x_2 x_1'$  och inringning av de två ettorna i understa raden ger termen  $x_3 x_2 x_1'$ . Dessa två termer ger sedan  $x_3' x_2 x_1' + x_3 x_2 x_1' = x_2 x_1'(x_3' + x_3) = x_2 x_1'$ , vilket är termen till inringningen av de fyra 1:orna.

Vi har i exemplen ovan sett hur inringning av en, två och fyra ettor kan göras och allmänt gäller att *antalet ettor som kan förekomma i en inringning måste vara en potens av 2*, dvs 1, 2, 4, 8, 16, osv. I figur 4.7 nedan visas de olika typer av inringningar som kan förekomma i ett Karnaugh-diagram med fyra variabler.

Ju större inringning desto färre antal variabler i produkttermen och sålunda färre antal ingångar i OCH-grinden i första nivån. Ju färre antal inringningar desto färre antal produkttermer i funktionsuttrycket och färre antal OCH-grindar i första nivån och ingångar i ELLER-grinden i andra nivån. Normal målsättning vid realisering av en boolesk funktion i ett booleskt uttryck är därför *så få och så korta termer som möjligt* i funktionsuttrycket och alltså i Karnaughdiagrammet, *så få och så stora inringningar som möjligt*.

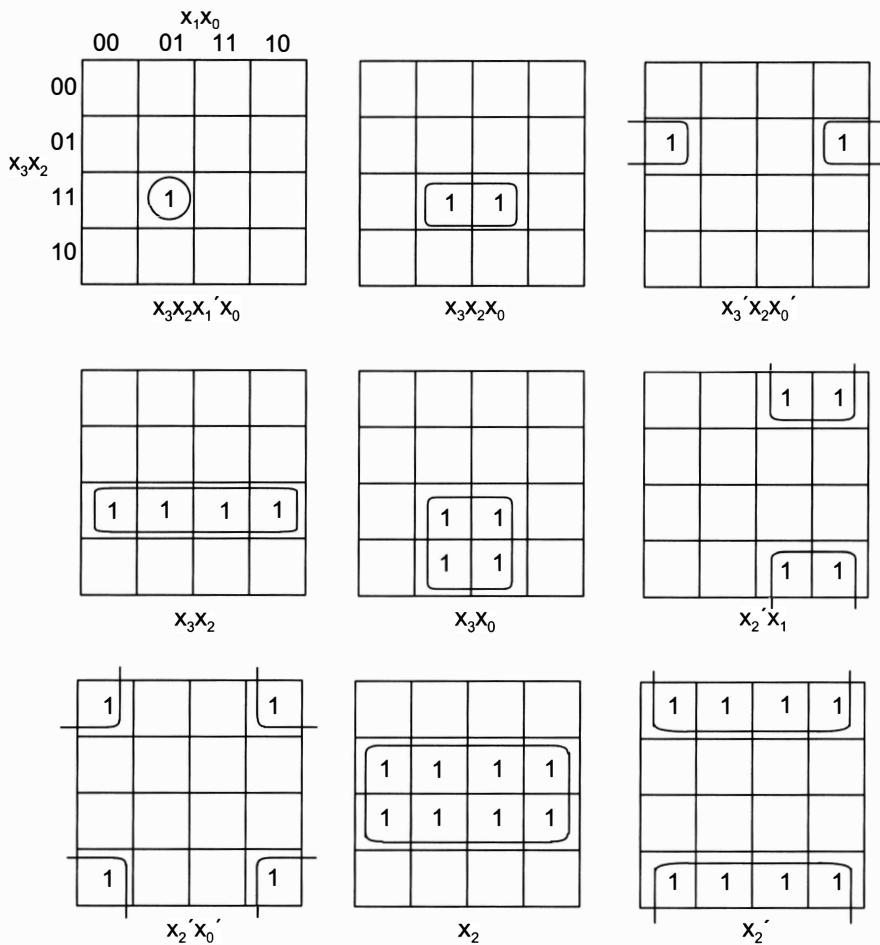
#### *Definitioner:*

En produktterm hörande till en maximal inringning benämnes **primimplikator**.

En primimplikator som täcker en minterm (etta i Karnaughdiagrammet) som inte täcks av någon annan primimplikator, benämnes **väsentlig primimplikator**.

En minimal SP-form till en boolesk funktion med så få och så korta termer som möjligt är uppenbart liktydigt med en summa av så få primimplikatorer som möjligt. Samtliga väsentliga primimplikatorer måste alltid ingå i det minimala uttrycket, eftersom de vardera täcker en minterm som inte täcks av någon annan primimplikator.

Inringningarna till funktionen  $lo_3$  i Karnaughdiagrammet i figur 4.6 är båda väsentliga primimplikatorer. Dels är de maximala inringningarna, dvs. primimplikatorer, dels täcker de ettor i rutorna 0001 respektive 0010 som inte täcks av någon annan primimplikator.



Figur 4.7 Inringningar i ett Karnaughdiagram med fyra variabler.

Bestämning av en minimal SP-form till en boolesk funktion (utan att man är intresserad av att bestämma samtliga primimplikatorer) kan ske i följande tre steg.

*Steg 1.*

Bestäm samtliga väsentliga primimplikatorer genom att leta upp ettor i Karnaughdiagrammet som bara täcks av en primimplikator, dvs. en väsentlig primimplikator. Markera sådana ettor med en asterisk (\*), så att man ser vilken detta som motiverar att det är en väsentlig primimplikator (det kan finnas flera sådana ettor för en väsentlig primimplikator, men det räcker att markera en).

*Steg 2.*

Bestäm återstående primimplikatorer, som krävs för att täcka ettorna som ej är täckta av väsentliga primimplikatorer.

*Steg 3.*

Skriv upp det minimala uttrycket till funktionen som summan av primimplikatorerna som erhållits i steg 1 och 2.

Steg 1 är relativt enkelt. Observera att det inte alltid behöver finnas väsentliga primimplikatorer. Steg 2 kan ibland innehåra svårigheter att bestämma vilka primimplikatorer som skall medtagas i det minimala uttrycket. – Låt oss nu ta några exempel.

**Exempel 4.4**

Bestäm en minimal SP-form till booleska funktionen  $f(x_3, x_2, x_1, x_0) = \Sigma(3, 4, 5, 7, 9, 13, 14, 15)$ .

*Lösning*

		$x_1x_0$				
		00	01	11	10	
		00	0	0	1*	0
		01	1*	1	1	0
		11	0	1	1	*1
		10	0	1*	0	0

Minimala SP-formen blir

$$f = x_3'x_2x_1' + x_3x_2x_1 + x_3'x_1x_0 + x_3x_1'x_0$$

Figur 4.8 Karnaughdiagram till exempel 4.4.

□

I exempel 4.4 ovan, om man inte följer steg 1–3 ovan, frestas man kanske att börja med att göra en så stor inringning som möjligt och ringa in de fyra ettorna i mitten av Karnaughdiagrammet. Detta är onekligen en primimplikator, men inte en väsentlig primimplikator och skall inte ingå i det minimala uttrycket eftersom samtliga ettor täcks av väsentliga primimplikatorer.

### Exempel 4.5

Bestäm en minimal SP-form till booleska funktionen  $f(x_3, x_2, x_1, x_0) = \Sigma(2, 3, 7, 8, 10, 12, 15)$ .

*Lösning*

		$x_1x_0$				
		00	01	11	10	
		00	0	0	1	1
		01	0	0	1	0
$x_3x_2$		11	*1	0	*1	0
		10	1	0	0	1

*En minimal SP-form blir*

$$f = x_3x_1'x_0' + x_2x_1x_0 + x_3x_2'x_0' + x_3'x_2'x_1$$

Figur 4.9 Karnaughdiagram till exempel 4.5.

□

I exempel 4.5 ovan finns det tre ettor som inte täcks av väsentliga primimplikatorer. En minimal SP-form visas ovan. Det finns emellertid två andra minimala SP-former. Vilka?

Vi har nu sett hur Karnaughdiagrammet kan användas för att skriva en boolesk funktion som en minimal summa av produkter, minimal SP-form, som kan realiseras i ett minimalt OCH-ELLER-nät. Karnaughdiagrammet kan lika bra användas för att skriva funktionen som en minimal produkt av summor, minimal PS-form, som kan realiseras i ett minimalt ELLER-OCH-nät. Metoden är mycket enkel. Steg 1–3 ovan appliceras på *funktionens invers*. Den erhållna minimala SP-formen till funktionens invers inverteras sedan och ger som resultat den minimala PS-formen.

**Exempel 4.6**

Bestäm en minimal PS-form till booleska funktionen  $f(x_3, x_2, x_1, x_0) = \Sigma(2, 3, 7, 8, 10, 12, 15)$  (samma funktion som i exempel 4.5 ovan).

**Lösning**

I stället för att rita ett speciellt Karnaughdiagram för funktionens invers  $f'$  och där göra inringningar av ettorna går det självklart lika bra att inringa nollorna i Karnaughdiagrammet till funktionen  $f$ .

		$x_1x_0$		
		00	01	11
00		0*	0	1
01	0	0	1	*0
	1	0*	1	0
10		1	0	*0

$$f' = x_3'x_1' + x_1'x_0 + x_2x_1x_0' + x_3x_2'x_0$$

De Morgans lagar ger

$$f = (f')' = (x_3 + x_1)(x_1 + x_0')(x_2' + x_1' + x_0) \cdot (x_3' + x_2 + x_0')$$

Figur 4.10 Karnaughdiagram till exempel 4.6.

□

Vi fortsätter med ytterligare exempel på Karnaughdiagram. Som tidigare nämnts är Karnaughdiagram användbara åtminstone upp till 6 variabler och exemplen som följer gäller minimering av funktioner av 5 respektive 6 variabler.

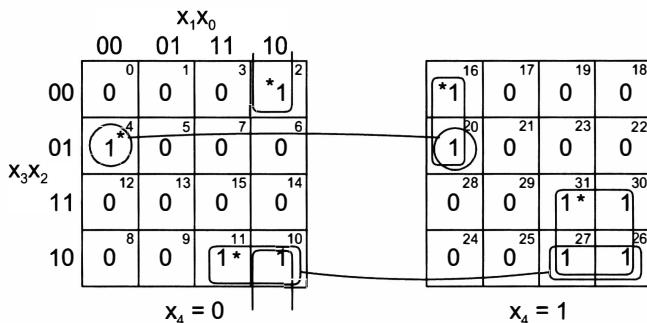
**Exempel 4.7**

Bestäm en minimal SP-form till booleska funktionen  $f(x_4, x_3, x_2, x_1, x_0) = \Sigma(2, 4, 10, 11, 16, 20, 26, 27, 30, 31)$ .

**Lösning**

Fem variabler innebär ett Karnaughdiagram med  $2^5 = 32$  rutor. Vi väljer att rita två stycken Karnaughdiagram med vardera 16 rutor, ett för  $x_4 = 0$  och ett för  $x_4 = 1$ . I rutorna har angetts den decimala siffran till rutans binära variabelkombination, vilket kanske kan underlättा inplacering av ettorna på rätt plats.

Inringning i de två Karnaughdiagrammen görs på samma sätt som tidigare, men en inringning kan nu även gå över båda diagrammen på så sätt att om en inringning förekommer på samma ställe i båda diagrammen (markerade sammanbundna i diagrammen nedan), så försvinner  $x_4$ -variabeln för den till inringningen hörande produkttermen.



Figur 4.11 Karnaughdiagram till exempel 4.7.

Den minimala SP-formen blir

$$f = x_4 x_3 x_1 + x_4' x_2' x_1 x_0' + x_4 x_3' x_1' x_0' + x_3 x_2' x_1 + x_3' x_2 x_1' x_0'$$

□

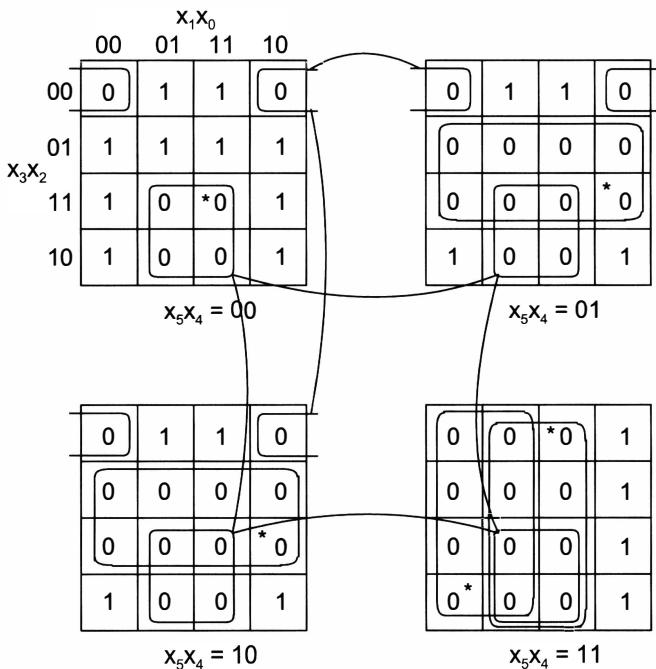
### Exempel 4.8

Bestäm en minimal PS-form till booleska funktionen

$$\begin{aligned} f(x_5, x_4, x_3, x_2, x_1, x_0) \\ = \Sigma(1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 17, 19, 24, 26, 33, 35, 40, 42, 50, 54, 58, 62). \end{aligned}$$

### Lösning

Sex variabler innebär ett Karnaughdiagram med  $2^6 = 64$  rutor. Vi väljer att rita fyra stycken Karnaughdiagram med vardera 16 rutor, för  $x_5 x_4 = 00, 01, 11$  respektive 10. Observera att de fyra diagrammen ritas i en sådan ordning att  $x_5 x_4$ , horisontellt och vertikalt, skiljer sig i bara en variabel på samma sätt som rutorna inne i själva diagrammet. Samma inringning kan nu kombineras i två diagram, horisontellt eller vertikalt, varvid den variabel av  $x_5$  och  $x_4$  som skiljer sig i de båda diagrammen, försvinner i termen som hör till inringningen. Om samma inringning kan kombineras i alla fyra diagrammen försvinner både  $x_5$  och  $x_4$  i termen.



Figur 4.12 Karnaghdiagram till exempel 4.8.

Den minimala SP-formen till  $f'$  blir

$$f' = x_3 x_0 + x_5' x_4 x_2 + x_5 x_4' x_2 + x_5 x_4 x_1' + x_5 x_4 x_0 + \\ x_5' x_3' x_2' x_0' + x_4' x_3' x_2' x_0'$$

Den minimala PS-formen till  $f$  blir

$$f = (x_3' + x_0')(x_5 + x_4' + x_2')(x_5' + x_4 + x_2')(x_5' + x_4' + x_1) \cdot \\ (x_5' + x_4' + x_0')(x_5 + x_3 + x_2 + x_0)(x_4 + x_3 + x_2 + x_0)$$

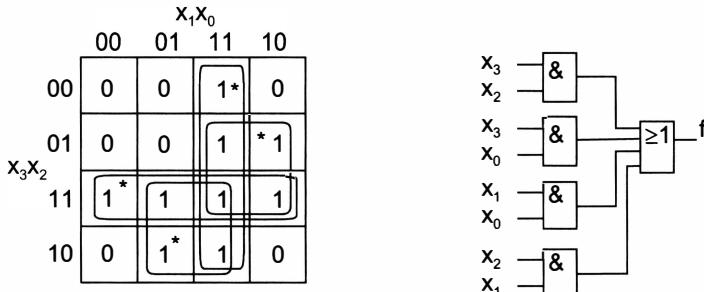
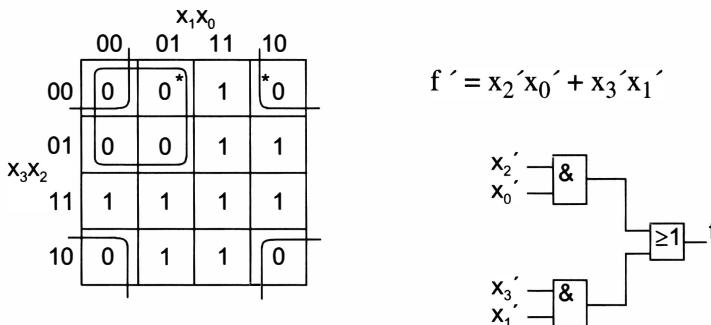
□

**Exempel 4.9**

Realisera booleska funktionen  $f(x_3, x_2, x_1, x_0) = \prod(0, 1, 2, 4, 5, 8, 10)$  och dess invers i var sitt minimalt OCH-ELLER-nät.:

*Lösning*Realisering av funktionen  $f$ :

$$f = x_3x_2 + x_3x_0 + x_1x_0 + x_2x_1$$

Figur 4.13 Karnaughdiagram och grindnät till funktionen  $f$ .Realisering av funktionens invers  $f'$ :Figur 4.14 Karnaughdiagram och grindnät till funktionen  $f'$ .

□

Notera i exempel 4.9 ovan att funktionens invers går att realisera med färre produkttermer än funktionen. Detta inträffar ibland och i vissa fall, som t.ex. i PLD som vi skall se längre fram, kan det vara lämpligt att realisera funktionens invers följd av en inverterare som bildar funktionen.

Låt oss nu avsluta denna sektion om Karnaughdiagram med att återvända till kretsen BCDcheck. Vi har förenklat funktionen lo3 i Karnaughdiagram

och det återstår att förenkla funktionen noBCD, vilken som det visar sig vi inte skrivit på minimal SP-form i den första VHDL-beskrivningen.

		$x_1x_0$				
		00	01	11	10	
		00	0	0	0	0
		01	0	0	0	0
$x_3x_2$		11	1 *	1	1	1
		10	0	0	1 *	1
noBCD						

Figur 4.15 Minimering av funktionen noBCD i kretsen BCDcheck.

Den minimala SP-formen till noBCD blir

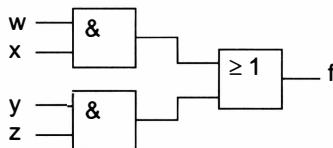
$$\text{noBCD} = x_3x_2 + x_3x_1$$

som har lika många produkter som i VHDL-beskrivningen, men där den andra produkten bara har två variabler jämfört med tre i VHDL-beskrivningen.

## NAND- och NOR-nät

I kapitel 2 nämnde vi att funktionerna OCH och ELLER är mer komplexa att realisera med transistorer än NAND och NOR, beroende på transistorns naturliga förmåga att invertera, som direkt kan utnyttjas i NAND och NOR, men som måste kompenseras med en extra inverterare i OCH och ELLER. Vi har i det föregående vid förenkling av booleska funktioner uteslutande diskuterat realisering i OCH-ELLER-nät och ELLER-OCH-nät. Vi skall nu se hur ett OCH-ELLER-nät är ekvivalent med ett NAND-NAND-nät och ett ELLER-OCH-nät är ekvivalent med ett NOR-NOR-nät. När vi i fortsättningen exempelvis visar realisering av en boolesk funktion i ett OCH-ELLER-nät så är det bara för att detta nät direkt ger en bild av realisering av en summa av produkter, SP-form, och inte hur funktionen i verkligheten är realiserad, vilket normalt är i ett NAND-NAND-nät.

Betrakta OCH-ELLER-nätet nedan.



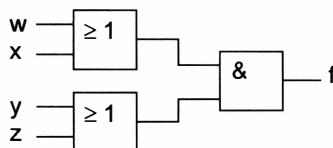
Figur 4.16 OCH-ELLER-nät.

Booleska funktionen  $f = wx + yz$  på SP-form, som är realiserad i OCH-ELLER-nätet kan omformas enligt

$$f = wx + yz = ((wx + yz)')' = ((wx)'(yz)')'$$

Vi ser i det omformade uttrycket att de inre parenteserna liksom den yttre parentesen är NAND-funktioner. Samtliga grindar i OCH-ELLER-nätet kan alltså ersättas med NAND-grindar!

Betrakta på samma sätt ELLER-OCH-nätet nedan.



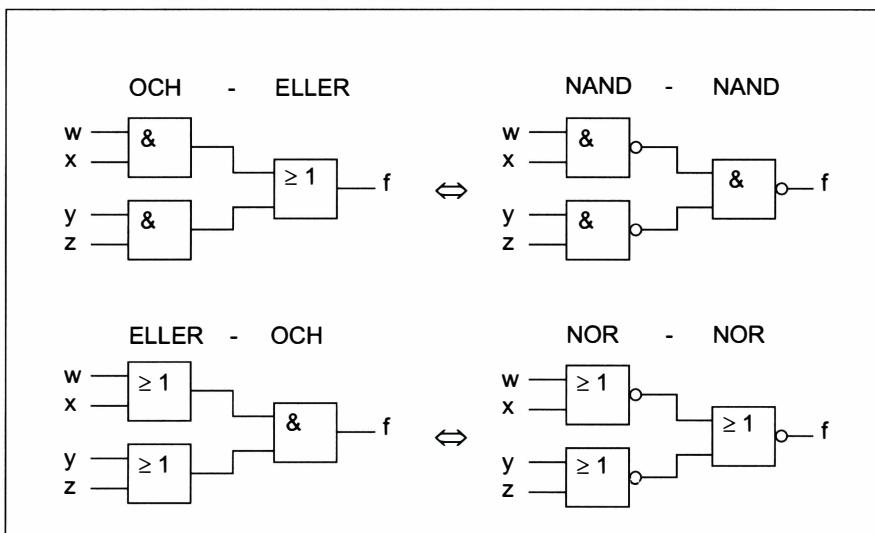
Figur 4.17 ELLER-OCH-nät.

Booleska funktionen  $f = (w + x)(y + z)$  på PS-form, som är realiserad i ELLER-OCH-nätet kan omformas enligt

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

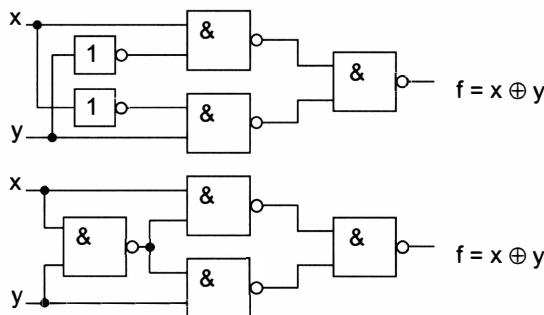
Vi ser i det omformade uttrycket att de inre parenteserna liksom den yttre parentesen är NOR-funktioner. Samtliga grindar i ELLER-OCH-nätet kan alltså ersättas med NOR-grindar!

Sammanfattning:



Figur 4.18 OCH-ELLER  $\Leftrightarrow$  NAND-NAND och ELLER-OCH  $\Leftrightarrow$  NOR-NOR.

I OCH-ELLER-, ELLER-OCH-, NAND-NAND-, och NOR-NOR-näten måste insignalernas inverser, om de ej redan finns tillgängliga, genereras i en första grindnivå av inverterare (eller NAND- och NOR-grindar). I vissa fall, vid användning av NAND-grindar, kan invertering av två eller flera signaler utföras i en NAND-grind. Principen illustreras i figur 4.19 nedan vid realisering av Exklusivt-ELLER-funktionen  $f = x \oplus y = x'y + xy'$ .



Figur 4.19 Realisering av XOR på två olika sätt.

I det övre grindnätet är Exklusivt-ELLER-funktionen realiseras på konventionellt sätt med en första nivå inverterare, medan i det undre grindnätet inverterarna ersatts med en NAND-grind. Principen bygger på sambandet

$$ab' = a(ab)' \quad (4.9)$$

Sambandet (4.9), som uppenbart gäller, ty  $a(ab)' = a(a' + b') = ab'$ , kan utläsas som att ”i en produkt av variabler, i detta fall  $ab'$ , kan en variabel flyttas in under en invertering, om variabeln fortfarande också behålls utanför inverteringen”. Principen gäller inte bara en variabel, utan ett godtyckligt antal variabler, t.ex. gäller  $abc' = ab(abc)'$ . Enligt denna princip kan Exklusivt-ELLER-funktionen skrivas om enligt

$$f = x'y + xy' = (xy)'y + x(xy)'$$

I stället för två inverterare för generering av  $x'$  och  $y'$  kan således en NAND-grind för generering av  $(xy)'$  användas.

## Ofullständigt specificerade funktioner

För en kombinationskrets kan specifikationen vara sådan att vissa insignalcombinationer aldrig kan förekomma. Detta innebär att utgångsvärdet fritt kan väljas till 0 eller 1 för dessa insignalcombinationer (självklart ger kombinationskretsen utgångsvärden även för de insignalcombinationer som aldrig kan inträffa). Specifikationen kan också vara sådan att utgångsvärdet är ointressant för vissa insignalcombinationer och därför fritt kan väljas till 0 eller 1 för dessa.

### Exempel 4.10

En kombinationskrets *C5421toBCD* som översätter 5421-kod (se tabell 1.3 i kapitel 1) till BCD-kod skall konstrueras. Kretsen skall realiseras i ett minimalt OCH-ELLER-nät.



Figur 4.20 Blockschema för kodomvandlare 5421-kod till BCD-kod.

**Lösning**

Utgående från tabell 1.3 ritar vi upp Karnaughdiagrammen för utsignalerna  $y_8$ ,  $y_4$ ,  $y_2$  och  $y_1$ .

		$x_2x_1$			
		00	01	11	10
$x_5x_4$	00	0	0	0	0
	01	0	-	-	-
	11	$1^*$	-	-	-
	10	0	0	$1^*$	0
$y_8$					

		$x_2x_1$			
		00	01	11	10
$x_5x_4$	00	0	0	0	0
	01	$1^*$	-	-	-
	11	0	-	-	-
	10	$1^*$	$1^*$	0	$1^*$
$y_4$					

		$x_2x_1$			
		00	01	11	10
$x_5x_4$	00	0	$1^*$	$1^*$	
	01	-	-	-	-
	11	-	-	-	-
	10	$1^*$	0	$1^*$	
$y_2$					

		$x_2x_1$			
		00	01	11	10
$x_5x_4$	00	0	$1^*$	1	0
	01	-	-	-	-
	11	-	-	-	-
	10	$1^*$	0	0	1
$y_1$					

Figur 4.21 Karnaughdiagram till kombinationskretsen C5421toBCD.

I Karnaughdiagrammen har funktionvärdena för de sex insignalkombinationer som ej kan förekomma i 5421-koden markerats med streck (-). Sådana värden som ”inte har någon betydelse”, benämnes ju *don't care*. I den komprimerade funktionsspecifikationen  $\Sigma()$  kan ospecifierade värden tillfogas som d(). Exempelvis kan funktionen  $y_8$  komprimerat skrivas som:  $y_8 = \Sigma(11, 12) + d(5, 6, 7, 13, 14, 15)$ .

Inringningar får nu även innehålla (-), ospecifierade funktionsvärden. Om en ospecifierad ruta ingår i en inringning innebär det att funktionsvärdet i rutan blir 1, medan om en ospecifierad ruta inte ingår i någon inringning blir funktionvärdet 0.

Väsentliga primimplikatorer har på vanligt sätt markerats med en asterisk (\*). Samtliga funktioner med undantag för  $y_4$  har realiseras med enbart väsentliga primimplikatorer. Funktionen  $y_4$  kan realiseras på flera olika sätt med användning av också de streckade inringningarna.

Med inringningarna i Karnaughdiagrammen ovan blir de minimala SP-formerna:

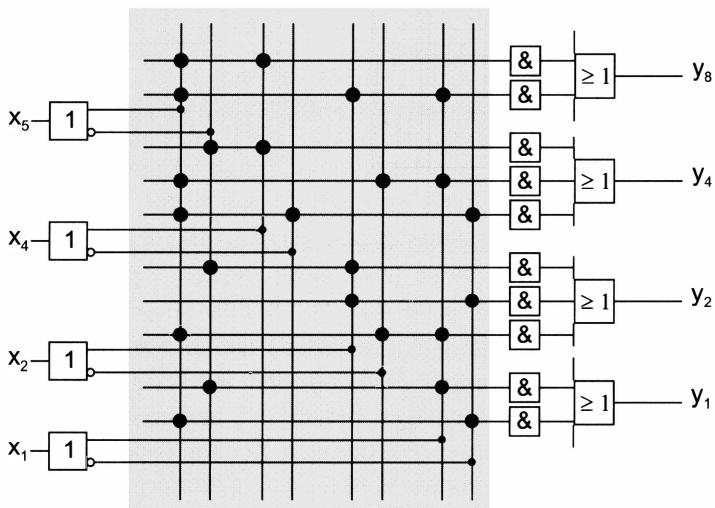
$$y_8 = x_5 x_4 + x_5 x_2 x_1$$

$$y_4 = x_5' x_4 + x_5 x_4' x_1' + x_5 x_2' x_1$$

$$y_2 = x_5' x_2 + x_2 x_1' + x_5 x_2' x_1$$

$$y_1 = x_5' x_1 + x_5 x_1'$$

Ett alternativ till minimeringen ovan med ospecifierade värden vore att från början sätta alla don't care till 0 eller till 1. De minimala funktionerna blir då normalt mer komplexa beroende på att större restriktioner lagts på funktionerna. Det är alltså viktigt att minimeringen får bestämma de ospecificerade funktionsvärdena.



Figur 4.22 Grindnät till kombinationskretsen C542ItoBCD.

Grindnätet ovan har ritats så att det antyder principen för programmering av förbindningar i en PLD. De fyllda stora ringarna i den skuggade matrisen markerar programmerade förbindningar mellan kretsens ingångar och OCH-grindarnas ingångar. Ett komprimerat ritsätt har använts för de 8-ingångars OCH-grindarna. Ingångarna symboliseras med en enda linje på vilken anslutning till en ingång markeras med en tjock fyllt ring.

## Grinddelning

Vid realiseringen av kombinationskretsen C5421toBCD minimerades de fyra booleska funktionerna var för sig och realiseras i skilda grindnät. Det kan finnas möjlighet att minska totala antalet grindar i kretsen genom att utnyttja samma grind i mer än ett grindnät, använda *grinddelning*. Studerar vi de minimala uttrycken till booleska funktionerna för C5421toBCD, så ser vi att funktionerna  $y_4$  och  $y_2$  båda innehåller produkttermen  $x_5x_2'x_1$ . Denna produktterm behöver bara realiseras i en OCH-grind vars utgång ansluts dels till ELLER-grinden i  $y_4$  dels till ELLER-grinden i  $y_2$ . Vi sparar alltså en grind genom detta förfarande. I detta fall finns inga fler gemensamma produkttermer i funktionerna som ger mer besparing av grindar. Dock går det som vi skall se, att ytterligare reducera totala antalet grindar i kretsen C5421toBCD. Detta kan ske genom att redan vid minimeringen av de olika funktionerna beakta möjlighet till grinddelning, motsvarande en optimering av sammanlagda antalet grindar i hela kretsen i stället för optimering av antalet grindar i var och en av kretsarna till de olika funktionerna. Låt oss se hur detta kan göras.

		$x_2x_1$							
		00	01	11	10				
00		0	0	0	0				
01		0	-	-	-				
11		1 *	-	-	-				
10		0	0	1 *	0				
		$y_8$							

		$x_5x_4$							
		00	01	11	10				
00		0	0	0	0				
01		0	-	-	-				
11		1 *	-	-	-				
10		0	0	1 *	0				
		$y_4$							

		$x_2$							
		00	01	11	10				
00		0	0	1 *	1				
01		0	-	-	-				
11		0	-	-	-				
10		0	1 *	0	1 *				
		$y_2$							

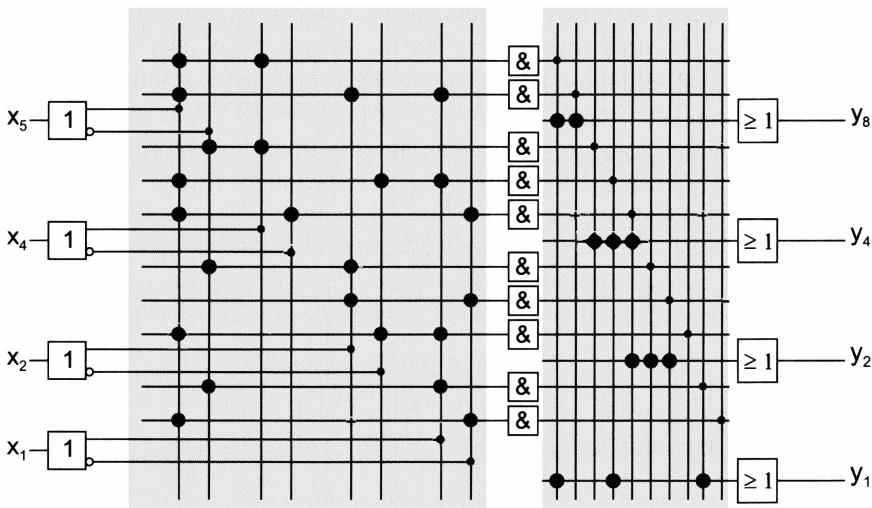
		$x_1$							
		00	01	11	10				
00		0	1 *	1	0				
01		0	-	-	-				
11		1	-	-	-				
10		1	0	0	1				
		$y_1$							

Figur 4.23 Minimering av  $y_8$ ,  $y_4$ ,  $y_2$  och  $y_1$  under beaktande av grinddelning.

Inringning i Karnaughdiagrammen under beaktande av grinddelning startar lämpligen med att man för varje funktion gör alla inringningar för vilka grinddelning ej är möjlig. Detta är liktydigt med att man börjar med att leta efter ettor som bara finns i ett Karnaughdiagram på den aktuella platsen och ringar in dem på bästa sätt. Sådana ettor i Karnaughdiagrammen i figur 4.23 ovan är, för funktionen  $y_8$  ettan i ruta 1011, för funktionen  $y_4$  ettan i ruta 0100, och för funktionen  $y_1$  ettan i ruta 0001. Därefter görs inringningar för grinddelning, innebärande att man måste analysera flera Karnaughdiagram samtidigt, vilket inte alltid är så lätt. I exemplet ovan så har jämfört med inringning utan grinddelning, endast funktionen  $y_1$  blivit annorlunda. I denna har inringningen  $x_5x_1'$  ersatts med de två inringningarna  $x_5x_4$  och  $x_5x_4'x_1'$  som delas med funktionerna  $y_8$  respektive  $y_4$ . Nedan visas booleska uttrycken för funktionerna med grinddelning. Gemensamma produkttermer i funktionerna har markerats med samma nummer.

$$\begin{aligned}y_8 &= x_5^1 x_4 + x_5 x_2 x_1 \\y_4 &= x_5' x_4^2 + x_5 x_4' x_1^3 + x_5 x_2' x_1 \\y_2 &= x_5' x_2 + x_2 x_1' + x_5 x_2' x_1^3 \\y_1 &= x_5' x_1^1 + x_5 x_4^2 + x_5 x_4' x_1'\end{aligned}$$

I grindnätet i figuren nedan har grinddelningen i den skuggade matrisen också ritats för att illustrera principen i en PLD. ELLER-grindarna har åtta ingångar till vilka kan anslutas utgångar från åtta av de tio OCH-grindarna. Liksom för OCH-grindarna har för ELLER-grindarna använts ett förenklat ritsätt med bara en ingångsledning. Som framgår av figuren så används totalt åtta OCH-grindar vid realiseringen med grinddelning (observera två av OCH-grindarna används ej), jämfört med tio OCH-grindar vid realiseringen utan grinddelning i figur 4.22 ovan.



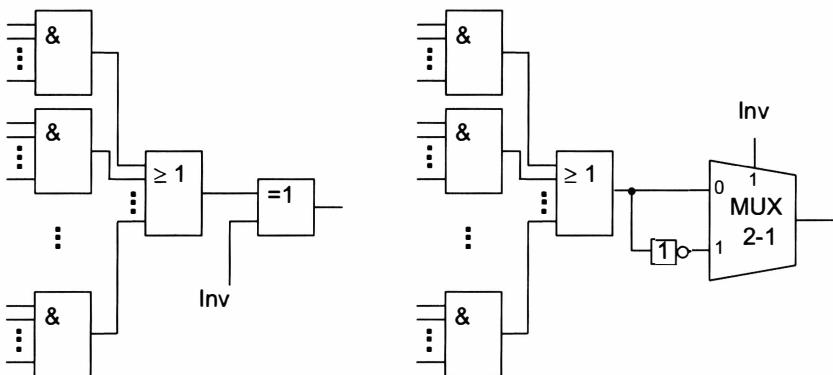
Figur 4.24 Grindnät med grinddelning till kombinationskretsen C5421toBCD.

## Standardgrindnät i PLD för realisering av booleska funktioner

I programmerbara logiska kretsar PLD, finns ett antal standardgrindnät av typ OCH-ELLER av en viss komplexitet. Normalt kan bara anslutningarna till OCH-grindarnas ingångar programmeras, men det finns också PLD där även OCH-grindarnas anslutning till ELLER-grindarna kan programmeras och alltså grinddelning utnyttjas. Vid realisering av en krets i en PLD så bestämmer syntesverktyget hur programmeringen av anslutningarna skall göras.

I en PLD där bara anslutningarna till OCH-grindarnas ingångar kan programmeras är antalet OCH-grindar ingående i ett OCH-ELLER-nät bestämt av tillverkaren av PLDn. En SP-form som skall realiseras i ett enda sådant OCH-ELLER-nät får då ha högst så många OCH-produkter som det finns OCH-grindar i OCH-ELLER-nätet. Vi har tidigare (exempel 4.9) sett hur ibland funktionens invers har färre produkttermer än funktionen. Det kan alltså hända att det inte är möjligt att realisera funktionen i ett givet OCH-

ELLER-nät i PLDn p.g.a. för få OCH-grindar, medan det är möjligt att realisera funktionens invers. Om funktionens invers  $f'$  realiseras i OCH-ELLER-nätet så måste utsignalen från ELLER-grinden inverteras för att funktionen  $f$  skall erhållas. Tillverkare av PLD gör därför standardgrindnät enligt principen i figuren nedan som OCH-ELLER-nät följdta av en styrbar invertering, vilken kan programmeras att inte invertera för realisering av funktionen eller invertera för realisering av funktionens invers.

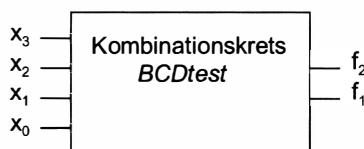


Figur 4.25 Standardgrindnät av typ OCH-ELLER med styrbar invertering.

Vid realisering av en funktion  $f$  i standardgrindnätet skall  $\text{Inv} = 0$  om den minimala SP-formen till funktionen  $f$  skall realiseras i OCH-ELLER-nätet, ty då går utsignalen  $f$  från ELLER-grinden genom XOR-grinden respektive MUX:en opåverkad (icke-inverterad). Om i stället den minimala SP-formen till funktionens invers  $f'$  skall realiseras i OCH-ELLER-nätet så skall  $\text{Inv} = 1$ , ty utgången från ELLER-grinden ger då funktionens invers  $f'$ , vilken sedan inverteras i XOR-grinden respektive via inverteraren och MUX:en så att  $(f')' = f$  erhålls. Låt oss studera ett exempel.

#### **Exempel 4.11**

En kombinationskrets BCDtest skall konstrueras.

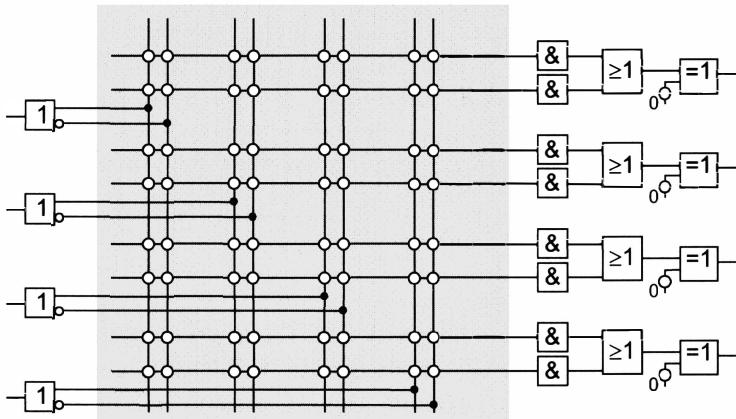


Figur 4.26 Blockschema för kombinationskretsen BCDtest.

Kretsen skall ha fyra ingångar  $x_3, x_2, x_1, x_0$  på vilka inmatas decimala sifferor i BCD-kod, betecknade  $x = (x_3, x_2, x_1, x_0)$ . Kretsen skall ha två utgångar  $f_1$  och  $f_2$  för vilka gäller:

$$\begin{array}{lll} f_1 = 1 & \text{om och endast om} & 5 \leq x \leq 9 \\ f_2 = 1 & \text{--"-} & x > 0 \text{ och jämnt delbar med 4} \end{array}$$

Kretsen skall realiseras i en PLD enligt nedan i vilken OCH-matrisen är programmerbar. De ofyllda ringarna i korsningspunkterna i matrisen är programmerbara. Ofylld ring markerar ingen förbindelse mellan x- och y-ledning, medan fylld ring som i tidigare figur 4.24 markerar förbindelse.

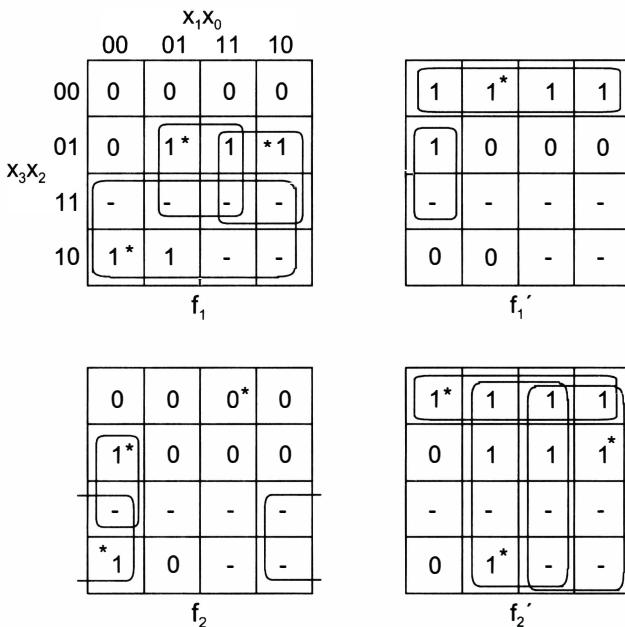


Figur 4.27 Programmerbart grindnät i en PLD.

### Lösning

<b>x Dec.</b>	<b>x i BCD x<sub>3</sub> x<sub>2</sub> x<sub>1</sub> x<sub>0</sub></b>	<b>f<sub>2</sub></b>	<b>f<sub>1</sub></b>
0	0 0 0 0	0	0
1	0 0 0 1	0	0
2	0 0 1 0	0	0
3	0 0 1 1	0	0
4	0 1 0 0	1	0
5	0 1 0 1	0	1
6	0 1 1 0	0	1
7	0 1 1 1	0	1
8	1 0 0 0	1	1
9	1 0 0 1	0	1

Figur 4.28 Funktionstabell för kombinationskretsen BCDtest.



Figur 4.29 Karnaughdiagram för kombinationskretsen BCtest.

### Minimala booleska funktioner

Ur Karnaughdiagrammen erhålls

$$f_1 = x_3 + x_2x_0 + x_2x_1$$

$$f_1' = x_3'x_2' + x_2x_1'x_0'$$

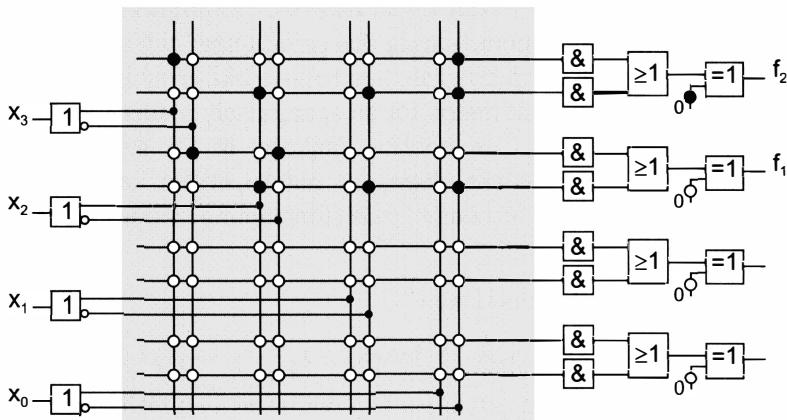
$$f_2 = x_3x_0' + x_2x_1'x_0'$$

$$f_2' = x_0 + x_1 + x_3'x_2'$$

### Realisering av booleska funktionerna i PLDn

Funktionerna  $f_1$  och  $f_2'$  har båda tre produkttermer och är därför inte möjliga att realisera i PLDn i vardera ett OCH-ELLER-nät, som bara har

två OCH-grindar.  $f_1'$  och  $f_2$  som båda har två produkttermer kan däremot realiseras enligt nedan.



Figur 4.30 Programmering av kombinationskretsen BCDtest i PLDn.

I en oprogrammerad (raderad) PLD är det förbindelse mellan x- och y-ledningarna. Detta innebär att samtliga OCH-grindar har utgångsvärdet 0, eftersom det i OCH-produkten ingår produkter av variabeln och variabelns invers, dvs.  $x \wedge x' = 0$ . Programmeringen innebär för varje OCH-grind borttagning av alla förbindelser mellan x- och y-ledningar som ej skall finnas. Ingen förbindelse mellan x- och y-ledning innebär insignalen 1 till OCH-grinden. I grindnätet ovan finns OCH-grindar som ej används och som alltså egentligen skall ha fylda ringar mellan alla x- och y-ledningar. Av bekvämlighet har dessa ej ritats ut.

För de styrbara inverterarna, XOR-grindarna, gäller på samma sätt att ofylld ring markerar ingångsvärdet 1, vilket medför att den inverterar insignalen ( $1 \oplus x = x'$ ), medan fyld ring markerar förbindelse, i detta fall med signalen 0 som medför ingen invertering ( $0 \oplus x = x$ ). I det första OCH-ELLER-nätet realiseras  $f_2$  som går opåververkad genom XOR-grinden. I det andra OCH-ELLER-nätet realiseras  $f_1'$  som inverteras i den efterföljande XOR-grinden.

□

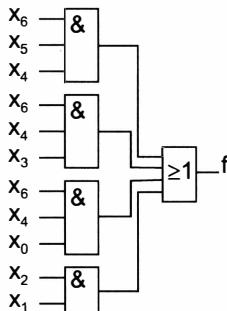
## Faktorisering

Vi har hittills uteslutande behandlat grindnät med två nivåer, typ OCH-ELLER etc. och poängterat att booleska funktioner bör realiseras i denna typ av nät för att fördröjningen skall bli så liten som möjligt. I vissa fall kan man dock av någon anledning tvingas öka grinddjupet och därmed tyvärr också fördröjningen. Exempelvis kan man behöva öka grinddjupet vid realisering av en krets i en grindmatris för att spara kiselyta eller i en PLD för att antalet produkttermer i en boolesk funktion är för stort för OCH-ELLER-näten i PLDn. Genom ökning av antalet nivåer kan grindnätets komplexitet minskas, ”tid är pengar”, tid (fördröjning) växlas mot pengar (komplexitet).

Låt oss studera booleska funktionen

$$f(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = x_6x_5x_4 + x_6x_4x_3 + x_6x_4x_0 + x_2x_1$$

som är skriven på SP-form och alltså kan realiseras i ett OCH-ELLER-nät enligt figuren nedan.



Figur 4.31 Realisering av funktionen  $f$  i ett grindnät med två nivåer.

Om vi definierar *komplexiteten* för grindnätet som summan av samtliga grindsars in- och utgångar, så blir komplexiteten K för grindnätet ovan

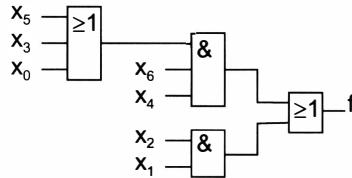
$$K = (3 + 1) + (3 + 1) + (3 + 1) + (2 + 1) + (4 + 1) = 20$$

De fyra första parenteserna är komplexiteten hos OCH-grindarna och den sista parentesen är komplexiteten hos ELLER-grinden.

En boolesk funktion kan ju i princip skrivas med hur många olika uttryck som helst och funktionen  $f$  kan faktoriseras genom att  $x_6x_4$  brytes ut ur de tre första termerna enligt

$$f(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = x_6x_4(x_5 + x_3 + x_0) + x_2x_1$$

och realiseras i ett grindnät med tre nivåer enligt figur 4.24 nedan.

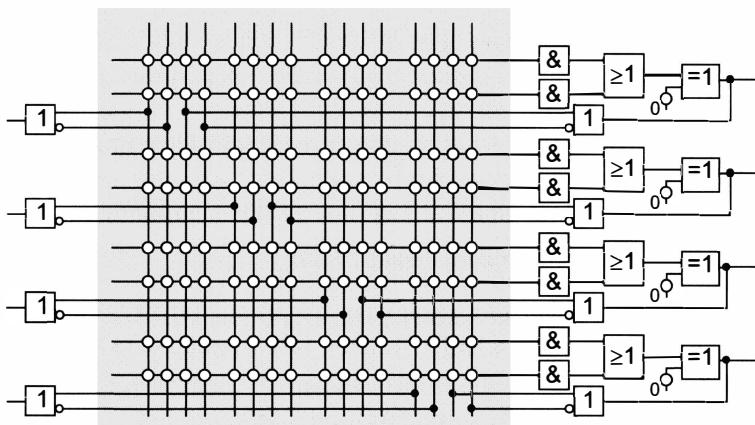


Figur 4.32 Realisering av funktionen  $f$  i ett grindnät med tre nivåer.

Komplexiteten  $K_f$  för grindnätet i figur 4.32 till det faktoriserade uttrycket blir

$$K_f = (3 + 1) + (3 + 1) + (2 + 1) + (2 + 1) = 14$$

Vi konstaterar alltså att komplexiteten minskat till priset av ökad födröjning.



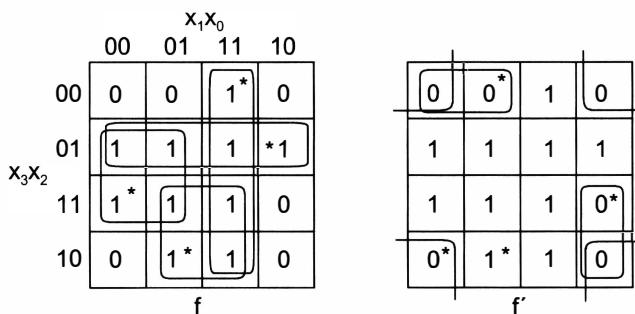
Figur 4.33 PLD med återkoppling av utgångarna till OCH-matrisen.

I PLD kan grinddjupet ökas genom att flera OCH-ELLER-nät kan kopplas efter varandra. Detta möjliggörs genom att OCH-ELLER-nätens utgångar har återkoppling till OCH-matrisen för eventuell anslutning till OCH-grindarna enligt principen i figuren ovan. Låt oss se hur detta kan göras i ett exempel.

### Exempel 4.12

Realisera booleska funktionen  $f(x_3, x_2, x_1, x_0) = \Pi(0, 1, 2, 8, 10, 14)$  i PLDn i figur 4.33 ovan.

Lösning



Figur 4.34 Karnaughdiagram till booleska funktionen i exempel 4.12.

### Minimala booleska funktioner

Ur Karnaughdiagrammen erhålls

$$f = x_3'x_2 + x_2x_1' + x_3x_0 + x_1x_0$$

$$f' = x_2'x_0' + x_3'x_2'x_1' + x_3x_1x_0'$$

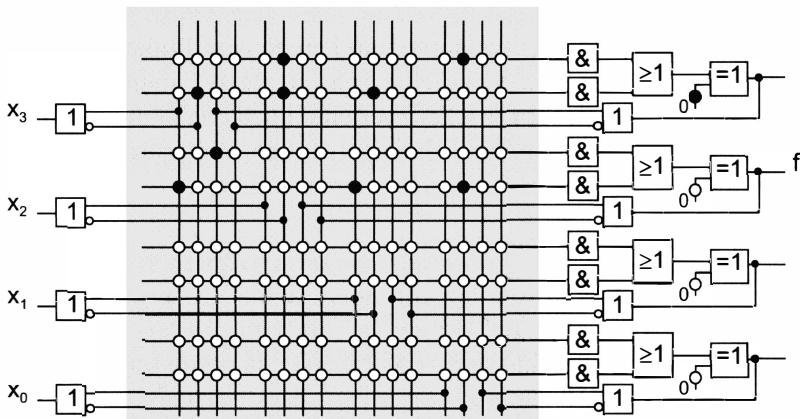
### Realisering av booleska funktionerna i PLDn

Funktionerna  $f$  och  $f'$  har fyra respektive tre produkttermer och kan därför inte realiseras direkt i OCH-ELLER-näten i PLDn som bara innehåller två OCH-grindar.

Funktionen  $f'$  kan realiseras i två OCH-ELLER-nät som

$$f' = (x_2'x_0' + x_3'x_2'x_1') + x_3x_1x_0'$$

där uttrycket i parentesen realiseras i ett första OCH-ELLER-nät, vars utgång sedan tillsammans med sista termen i uttrycket till  $f'$  realiseras i ett andra OCH-ELLER-nät och vars utgång sedan slutligen inverteras i den styrbara inverteraren så att  $f$  erhålls.



Figur 4.35 Programmering av booleska funktionen i exempel 4.12 i PLDn.

□

## 4.4 Quine-McCluskeys förenklingsmetod

Karnaughdiagram är ett ypperligt hjälpmedel för minimering av booleska funktioner och för själva förståelsen av minimeringsmetodiken med begreppen primimplikator, väsentlig primimplikator etc. De är användbara för minimering av booleska funktioner upp till maximalt ca sex variabler, sedan blir Karnaughdiagrammen för stora att hantera både för hand och hjärna. Vid minimering av booleska funktioner med mer än sex variabler erfordras minimeringsmetoder som kan köras på dator. Den första och klassiska algoritmen utvecklades först av Quine (1952) och förbättrades sedan av McCluskey (1956) och algoritmen brukar benämnes *Quine-McCluskey*. Under 1980-talet har minimeringsalgoritmer fått ökad betydelse och en del nya algoritmer, Espresso, FACT m.fl., har kommit fram i samband med expansionen av området *Programmerbara logiska kretsar (PLD, Programmable Logic Devices)*, där syntesverktygen för PLD innehåller minimeringsalgoritmer, som utför minimering av booleska funktioner som skall realiseras i PLD. Om så önskas kan detta avsnitt överhoppas utan att sam-

manhanget går förlorat, men den intresserade kan se det nära sambandet med Karnaughdiagrammet och kanske inspireras till ett närmare studium av moderna avancerade minimeringsmetoder. Quine-McCluskeys minimeringsmetod har nämligen nackdelen att den är mycket minneskrävande i datorn om kretsen har många insignaler, vilket ofta är fallet.

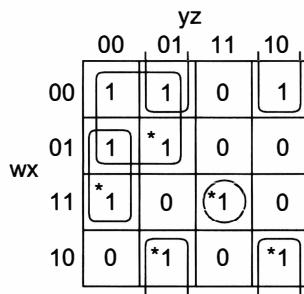
Möjligheten att slå samman två mintermer i en boolesk funktion till en, kortare term, förutsätter som vi tidigare sett, att de två mintermerna bara skiljer sig i en variabel, som är icke-inverterad i den ena mintermen och inverterad i den andra mintermen. Exempelvis kan mintermerna  $x'y'z$  och  $x'yz$ , som skiljer sig i variabeln  $y$ , slås samman enligt

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

Variabelkombinationerna som hör till mintermerna  $x'y'z$  och  $x'yz$  är 001 respektive 011. Dessa binärtal skiljer sig bara i en position,  $y$ -positionen, och om två mintermer skall kunna slås samman så måste uppenbart de till mintermerna hörande binärtalen skilja sig i bara en position. Quine-McCluskey-algoritmen bygger på en systematisk behandling av binärtalen till den booleska funktionens mintermer. Algoritmen illustreras enklast med ett exempel. Låt oss minimera booleska funktionen

$$f(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 9, 10, 12, 15)$$

Vi börjar med att minimera funktionen med Karnaughdiagram.



Figur 4.36 Karnaughdiagram för funktionen  $f$ .

Booleska funktionen  $f$  på minimal SP-form blir

$$f = w'y' + xy'z' + x'y'z + x'yz' + wxyz$$

Väsentliga primimplikatorer har på vanligt sätt markerats med asterisk (\*) i Karnaughdiagrammet. Notera att funktionen  $f$  har ytterligare en primimp-

likator  $w'x'z'$ , som inte ritats in i Karnaughdiagrammet, men som kommer att genereras med algoritmen Quine-McCluskey som genererar samtliga primimplikatorer.

Låt oss nu minimera funktionen  $f$  med Quine-McCluskey. Algoritmen omfattar två huvudmoment:

1. Generering av samtliga primimplikatorer.
2. Bestämning av en eller flera minimala mängder av primimplikatorer för funktionen.

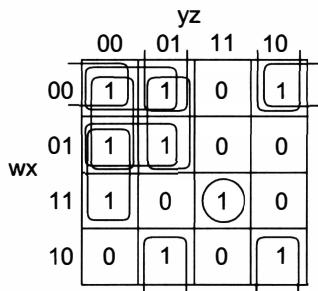
### *1. Generering av samtliga primimplikatorer.*

Första steget i algoritmen omfattar gruppering av binärtalen till funktionens mintermer efter antalet ettor i binärtalen, se kolumnen "Mintermer" i tabell 4.5 nedan. Decimaltalen till binärtalen anges bredvid. Samtliga möjliga sammanslagningar av mintermer för eliminering av en variabel enligt principen ovan, skall nu göras. Sammanslagning av två mintermer är möjlig om mintermerna skiljer sig bara i en variabel och kan därför göras bara om mintermerna ligger i två angränsande grupper. Binärtalen jämförs nu systematiskt på så sätt att varje binärtal i grupp 0 jämförs med samtliga binärtal i grupp 1, varje binärtal i grupp 1 jämförs med samtliga binärtal i grupp 2, osv. Om två binärtal skiljer sig i bara en position, bildas ett nytt binärtal som har samma siffror i de positioner där binärtalen är lika och ett streck (-) i den position där binärtalen skiljer sig, markerande att denna variabel elimineras. Dessa nya binärtal skrivs upp i en ny kolumn bredvid den första kolumnen av mintermer, se kolumnen "Första sammanslagningen" i tabell 4.5 nedan, och grupperas så att binärtalen som erhålls vid jämförelse av samma två grupper sammanförs till en grupp. Vid sidan av varje binärtal anges också decimaltalen (i stigande följd) för mintermerna som sammanslagits, och inom parentes positionsvikten för den variabel som elimineras.

När samtliga jämförelser gjorts och kolumnen är färdig, prickas de mintermer som ingår i en sammanslagning av med en "bock" (✓). Om någon minterm inte blir avprickad är den en primimplikator som måste ingå i det minimala funktionsuttrycket. I vårt exempel är minterm 15 en primimplikator, vilket också framgår av Karnaughdiagrammet i figur 4.36. Sammanslagningarna i kolumnen "Första sammanslagningen" visas som jämförelse också i ett Karnaughdiagram i figur 4.37 nedan.

Tabell 4.5: Generering av samtliga primimplikatorer till funktionen  $f$ .

Mintermer				Första sammanslagningen				Andra sammanslagningen			
Antal 1:or	bin.	dec.		000-	0,1	(1)	✓	0-0-	0,1,4,5	(1,4)	)
0	0000	0	✓	000-	0,1	(1)	✓	0-0-	0,1,4,5	(1,4)	)
1	0001	1	✓	00-0	0,2	(2)		0-0-	0,4,1,5	(1,4)	)
	0010	2	✓	0-00	0,4	(4)	✓				
	0100	4	✓	0-01	1,5	(4)	✓				
2	0101	5	✓	-001	1,9	(8)					
	1001	9	✓	-010	2,10	(8)					
	1010	1	✓	010-	4,5	(1)	✓				
		0									
	1100	1	✓	-100	4,12	(8)					
		2									
4	1111	1									
		5									

Figur 4.37 Karnaughdiagram för första sammanslagningen till funktionen  $f$ .

Nästa steg är sammanslagningen av binärtal i kolumnen "Första sammanslagningen". Detta motsvarar i Karnaughdiagrammet ovan sammanslagning av inringning omfattande två ettor med "intilliggande" inringning med två ettor. På samma sätt som vid sammanslagning av binärtalen till mintermerna behöver också här bara jämföras binärtal för sammanslagningar i två angränsande grupper. För att två binärtal skall kunna slås samman måste de ha streck på samma ställe, dvs samma variabel

måste vara elimineras, och så får de liksom tidigare bara skilja sig i en position. I vårt exempel kan 0,1 och 4,5 slås samman till 0,1,4,5, motsvarande i Karnaughdiagrammet ovan sammanslagning av de båda vågräta inringningarna i övre vänstra hörnet. Vidare kan 0,4 och 1,5 slås samman till 0,4,1,5, motsvarande sammanslagning av de båda lodräta inringningarna i övre vänstra hörnet. Den resulterande inringningen med fyra ettor blir för båda dessa fall densamma. Den sistnämnda sammanslagningen behöver inte göras och allmänt gäller regeln att *sammanslagningar skall bara göras då decimaltalen bildar en stigande talföljd*. I exemplet bildar 0,1 och 4,5 en stigande talföljd, medan 0,4 och 1,5 gör det inte och skall därför inte slås samman.

I detta fall kan alltså bara bildas en sammanslagning av binärtalen i kolumnen ”Första sammanslagningen”, nämligen 0,1,4,5, som bildar kolumnen ”Andra sammanslagningen”. Precis som tidigare prickar vi av med en bock binärtal i kolumnen ”Första sammanslagningen” vars decimaltal ingår i sammanslagningar i kolumnen ”Andra sammanslagningen”. I detta fall prickar vi alltså av 0,1 och 4,5. Icke avprickade binärtal i ”Första sammanslagningen” är primimplikatorer.

Genereringen av primimplikatorer fortsätter sedan på samma sätt med att sammanslagningsmöjligheterna undersöks i kolumnen ”Andra sammanslagningen”. I detta fall finns inga möjligheter till sammanslagningar och samtliga primimplikatorer har nu alltså genererats.

Genereringen av primimplikatorer i exemplet ovan har skett genom studium av mintermernas binärtal, vilket är lämpligt i ett inledande exempel där principerna för sammanslagningar skall förklaras. Med fördel används i fortsättningen endast mintermernas decimaltal vid generering av primimplikatorerna. Detta sker i stort enligt samma princip som ovan och går till på följande sätt.

Mintermerna grupperas på samma sätt som tidigare efter antalet ettor, men endast decimaltalen till mintermerna skrivs upp i kolumnen ”Mintermer”. Liksom tidigare jämförs också mintermerna i två angränsande grupper. Binärtalen får bara skilja sig i en position, dvs decimaltalen skall skilja sig med en 2-potens. *Observera* dock att decimaltalet i gruppen som har en etta måste vara störst. Detta inses lätt då binärtalen skall vara lika i alla positioner utom i den position, där det ena binärtalatet har en etta mer, dvs är en 2-potens större. Exempelvis kan två binärtal 1000 och 1100, som ligger i angränsande grupper och har *en* etta respektive *två* ettor, slås samman, och tillhörande decimaltal 8 och 12 skiljer sig med en 2-potens,  $2^2 = 4$ , och talet

12 som har en etta mer, är större än 8. Däremot kan inte två binärtal 1000 och 0110, som också ligger i angränsande grupper och har en etta respektive två ettor, slås samman trots att talen skiljer sig med en 2-potens,  $2^1 = 2$ , ty talet 6 i gruppen med en etta mer, är mindre än talet 8. Decimaltalen för mintermerna som kan slås samman, skrivs upp på samma sätt som i tabell 4.5 i kolumnen ”Första sammanslagningen” tillsammans med, inom parentes, positionsvikten för den variabel som elimineras, som ju är decimaltalens skillnad, 2-potensen.

”Andra sammanslagningen” och eventuellt efterföljande sammanslagningar, bildas genom jämförelse av decimaltalföljderna i angränsande grupper som har samma talföljd inom parentesen, dvs med samma variabler elimineras. Liksom tidigare måste decimaltalföljderna bilda en stigande talföljd. För att decimaltalföljderna skall kunna slås samman måste decimaltalen i de två talföljderna positionsvis skilja sig med en och samma 2-potens. I vårt exempel ovan har talföljderna 0,1 och 4,5 samma parentes (1), och  $4,5 - 0,1 = 4,4$ , dvs de skiljer sig med en och samma 2-potens,  $2^2 = 4$ , och alltså kan decimaltalföljderna slås samman till 0,1,4,5 (1,4). Parentesen (1) till vardera 0,1 och 4,5 utökas med skillnaden 4 mellan talföljderna till (1,4) och markerar positionsvikten för de två variabler som elimineras i sammanslagningen 0,1,4,5.

När samtliga primimplikatorer genereras skrivs primimplikatorerna lätt upp med hjälp av decimaltalföljderna. Mintermen till decimaltalföljdens första decimaltal skrivs upp och därefter utesluts i denna minterm de variabler med positionsvikter enligt decimaltalföljden i parentesen. Exempelvis för vårt exempel i tabell 4.5 ovan erhålls primimplikatorn  $P_1$  till sammanslagningen 0,1,4,5 (1,4) genom att i minterm 0,  $w'x'y'z'$ , utesluta variablerna med positionsvikterna 1 och 4, dvs. variablerna  $z'$  och  $x'$ . Således blir  $P_1 = w'y'$ .

Samtliga primimplikatorer till funktionen f blir

$$P_1 = w'y'$$

$$P_2 = w'x'z'$$

$$P_3 = x'y'z$$

$$P_4 = x'yz'$$

$$P_5 = xy'z'$$

$$P_6 = wxyz (= m_{15})$$

**2. Bestämning av en eller flera minimala mängder av primimplikatorer för funktionen f.**

Vi skall nu bestämma en eller flera minimala mängder av primimplikatorer som realiseras funktionen f, dvs täcker funktionens samtliga mintermer. För att göra detta ritar vi en *primimplikatortabell* enligt nedan, med en rad för varje primimplikator och en kolumn för varje minterm i funktionen (primimplikatorer som utgörs av mintermer, liksom tillhörande mintermer, ritas dock inte in i tabellen som rader respektive kolumner, eftersom dessa mintermer alltid skall ingå i den minimala funktionen). Primimplikatorerna grupperas efter stigande komplexitet. I detta fall har P<sub>1</sub> minst komplexitet och bildar en grupp, medan övriga primimplikatorer har samma komplexitet och bildar en grupp. Inne i tabellen markeras sedan med kryss (x) att en primimplikator täcker en minterm, dvs att mintermens nummer ingår i primiplikatorn decimaltalföljd, vilket i Karnaughdiagrammet motsvarar att en detta ingår i en primimplikatorinringning.

*Tabell 4.6: Primimplikatortabell för funktionen f.*

	0 ✓	1 ✓	2 ✓	4 ✓	5 ✓	9 ✓	10 ✓	12 ✓
* P <sub>1</sub> : 0,1,4,5 (1,4)	x	x		x	⊗			
P <sub>2</sub> : 0,2 (2)	x		x					
* P <sub>3</sub> : 1,9 (8)		x				⊗		
* P <sub>4</sub> : 2,10 (8)			x		x		⊗	
* P <sub>5</sub> : 4,12 (8)				x				⊗

Processen att välja ut en eller flera minimala mängder av primimplikatorer som realiseras funktionen f startar med bestämning av väsentliga primimplikatorer. I Karnaughdiagrammet letar vi då efter ettor som bara ingår i en enda inringning, innebärande att denna inringning är en väsentlig primimplikator. I primimplikatortabellen motsvarar detta att vi skall leta efter kolumner med bara ett enda kryss. I tabell 4.6 ovan har sådana kryss inringats och tillhörande väsentliga primimplikator markerats med en asterisk (\*).

Mintermerna som täcks av de väsentliga primimplikatorerna bockas sedan av, inte bara mintermerna i kolumner med inringade kryss, utan även de

andra mintermerna som täcks av de väsentliga primimplikatorerna. Som synes täcks i detta exempel samtliga mintermer av väsentliga primimplikatorer. Så är normalt inte fallet och då blir bestämningen av en minimal mängd primimplikatorer inte lika enkel och man måste rita fler primimplikatortabeller.

Primimplikatorn  $P_2$  skall alltså i exemplet ovan ej ingå i det minimala funktionsuttrycket, vilket också har framgått av Karnaughdiagrammet i figur 4.36 ovan. Funktionen  $f$  på minimal SP-form blir alltså

$$f = P_1 + P_3 + P_4 + P_5 + P_6 = w'y' + x'y'z + x'yz' + xy'z' + wxyz$$

En minimal SP-form till funktionen i exemplet ovan bestämdes relativt enkelt. Den omfattade enbart väsentliga primimplikatorer. Vi skall nu ta ett annat exempel där så inte är fallet som belyser några problem vid bestämning av en minimal mängd av primimplikatorer för en funktion.

Låt oss minimera funktionen

$$f_q(x_3, x_2, x_1, x_0) = \Sigma(2, 3, 7, 8, 10, 12, 15)$$

Funktionen har tidigare minimerats i exempel 4.5 med Karnaughdiagram, som visas nedan.

		$x_1x_0$				
		00	01	11	10	
		00	0	0	1	1
		01	0	0	1	0
$x_3x_2$		11	*1	0	*1	0
		10	1	0	0	1

*En minimal SP-form blir*

$$f_q = x_3x_1'x_0' + x_2x_1x_0 + x_3x_2'x_0' + x_3'x_2'x_1$$

Figur 4.38 Karnaughdiagram för booleska funktionen  $f_q$

Tabell 4.7: Generering av samtliga primimplikatorer till funktionen  $f_q$ 

Mintermer			Första sammanslagningen	
Antal 1:or dec.				
1	2	✓	2,3	(1)
	8	✓	2,10	(8)
2	3	✓	8,10	(2)
	1	✓	8,12	(4)
	0			
1	✓		3,7	(4)
2				
3	7	✓	3,7	(4)
4	1	✓	7,15	(8)
5				

2. Bestämning av en eller flera minimala mängder av primimplikatorer för funktionen  $f_q$ .

Tabell 4.8: Primimplikatortabell för funktionen  $f_q$ 

	2	3	7 ✓	8 ✓	10	12 ✓	15 ✓
P <sub>1</sub> : 2,3 (1)	✗	✗					
P <sub>2</sub> : 2,10 (8)	✗				✗		
P <sub>3</sub> : 8,10 (2)				✗	✗		
* P <sub>4</sub> : 8,12 (4)				✗		⊗	
P <sub>5</sub> : 3,7 (4)		✗	✗				
* P <sub>6</sub> : 7,15 (8)			✗				⊗

Funktion  $f_q$  har endast två väsentliga primimplikatorer,  $P_4$  och  $P_6$ . Vi reducerar nu primimplikatortabellen genom att ta bort raderna med de väsentliga primimplikatorerna och kolumnerna med de mintermer som täcks av de väsentliga primimplikatorerna. Då erhålls följande reducerad primimplikatortabell.

Tabell 4.9: Reducerad primimplikatortabell för funktionen  $f_q$ 

	2	3	10
P <sub>1</sub> : 2,3 (1)	×	×	
P <sub>2</sub> : 2,10 (8)	×		×
P <sub>2</sub> ⊃ P <sub>3</sub> : 8,10 (2)			×
P <sub>1</sub> ⊃ P <sub>5</sub> : 3,7 (4)		×	

Samtliga primimplikatorer är i detta fall lika komplexa, omfattar lika många variabler, och har därför samma prioritet vid val av minimal mängd. I tabell 4.9 ovan framgår att alla mintermer som täcks av primimplikator P<sub>3</sub>, täcks också av primimplikator P<sub>2</sub>, man säger att P<sub>2</sub> domineras av P<sub>3</sub>, som kan betecknas P<sub>2</sub> ⊃ P<sub>3</sub>. Likaså gäller att P<sub>1</sub> domineras av P<sub>5</sub>. Om vi bara söker en minimal mängd av primimplikatorer kan alltså P<sub>3</sub> och P<sub>5</sub> föras bort från resonemanget och tabellen reduceras ytterligare till tabellen nedan.

Tabell 4.10: Reducerad primimplikatortabell för funktionen  $f_q$ 

	2	3	10
P <sub>1</sub> : 2,3 (1)	×	×	
P <sub>2</sub> : 2,10 (8)	×		×

En minimal SP-form till funktionen  $f_q$  blir alltså

$$\begin{aligned} f_q(x_3, x_2, x_1, x_0) &= P_4 + P_6 + P_1 + P_2 \\ &= x_3x_1'x_0' + x_2x_1x_0 + x_3'x_2'x_1 + x_2'x_1x_0 \end{aligned}$$

Observera att dominerande primimplikatorer kan bara uppträda i en reducerad tabell. I den ursprungliga tabellen kan självfallet ingen primimplikator täcka en annan primimplikator, ty då vore enligt definitionen på primimplikator den täckta primimplikatorn ingen primimplikator.

I detta exempel har vi nöjt oss med *en* minimal form. Om samtliga minimala former önskas, kan efter bestämningen av väsentliga primimplikatorer, valet av övriga primimplikatorer reduceras till ett logiskt problem och utföras med en metod av Petrick (1956). Låt oss applicera denna metod på exemplet ovan.

Vi utgår från den reducerade primimplikatortabellen i tabell 4.9, som erhållits efter eliminering av väsentliga primimplikatorer. – Med varje primimplikator  $P_i$  i denna tabell associerar vi en logisk variabel  $p_i$ , sådan att  $p_i = 1$  (sann) innebär att primimplikatorn  $P_i$  skall ingå i en minimal SP-form till  $f_q$ , medan  $p_i = 0$  (falsk) innebär att  $P_i$  inte skall ingå i en minimal SP-form till  $f_q$ . Vi kan då sätta upp ett logiskt uttryck (påstående), som säger vilka primimplikatorer som måste medtagas i en minimal SP-form för att täcka samtliga mintermer i tabell 4.9 ovan. Det logiska uttrycket blir

$$P = (p_1 + p_2)(p_1 + p_5)(p_2 + p_3)$$

Uttrycket  $P$  betyder, att för att täcka minterm 2 skall primimplikator ( $P_1$  eller  $P_2$ ) medtagas, och för att täcka minterm 3 skall ( $P_1$  eller  $P_5$ ) medtagas, och för att täcka minterm 10 skall ( $P_2$  eller  $P_3$ ) medtagas.

För att ur uttrycket  $P$  få fram de olika alternativen, omvandlar vi uttrycket  $P$  med lagarna i Booleska algebran, t ex distributiva lagarna, absorptionslagen etc, till SP-form enligt

$$\begin{aligned} P &= (p_1 + p_2)(p_1 + p_5)(p_2 + p_3) \\ &= (p_1 + p_2p_5)(p_2 + p_3) \\ &= p_1p_2 + p_1p_3 + p_2p_5 + p_2p_3p_5 \\ &= p_1p_2 + p_1p_3 + p_2p_5 \end{aligned}$$

Vi ser här att det förutom den tidigare framtagna minimala formen med  $P_1$  och  $P_2$  (första termen i uttrycket  $P$  ovan), även finns två andra minimala former som innehåller  $P_1$  och  $P_3$  respektive  $P_2$  och  $P_5$ .

Vid bestämning av en minimal form enbart med hjälp av primimplikatortabellen konstaterades att dominerade rader kunde strykas. Nedan domineras rad  $P_r$  av rad  $P_s$ , varför rad  $P_s$  alltså kan strykas. Med Petricks metod erhålls naturligtvis samma resultat med uttrycket  $(p_r + p_s)p_r$  som enligt absorptionslagen är lika med  $p_r$

$P_r$	$\times$	$\times$	dominerande rad
$P_s$	$\times$		dominerad rad

I reducerade primimplikatortabeller kan också uppträda dominerande och dominerade kolumner enligt nedan, där kolumn  $K_u$  domineras av kolumn  $K_v$ .

	$K_u$	$K_v$
$P_r$	×	×
$P_s$		×
$P_t$	×	×
	dominerad kolumn	dominerande kolumn

I fallet med dominerade och dominerande kolumner får den dominerande (obs!) kolumnen strykas. Detta är uppenbart möjligt då alla primimplikatorer som täcker mintermen till den dominerade kolumnen alltid också täcker mintermen till den dominerande kolumnen. Ovan kan alltså den dominerande kolumnen  $K_v$  strykas, vilket också erhålls med Petricks metod och absorptionslagen enligt  $(p_r + p_t)(p_r + p_s + p_t) = p_r + p_t$

### Ofullständigt specificerade funktioner

Minimering av ofullständigt specificerade funktioner med Quine-McCluskey skiljer sig inte nämnvärt från minimering av fullständigt specificerade funktioner. Vid generering av samtliga primimplikatorer behandlas de ospecificerade positionerna som ettor, medförande alltså att en maximal mängd primimplikatorer genereras. Vid val av en minimal mängd primimplikatorer i primimplikatortabellen medtages däremot inte mintermerna för de ospecificerade positionerna, endast mintermerna där funktionen är specificerad skall ju täckas av primimplikatorer.

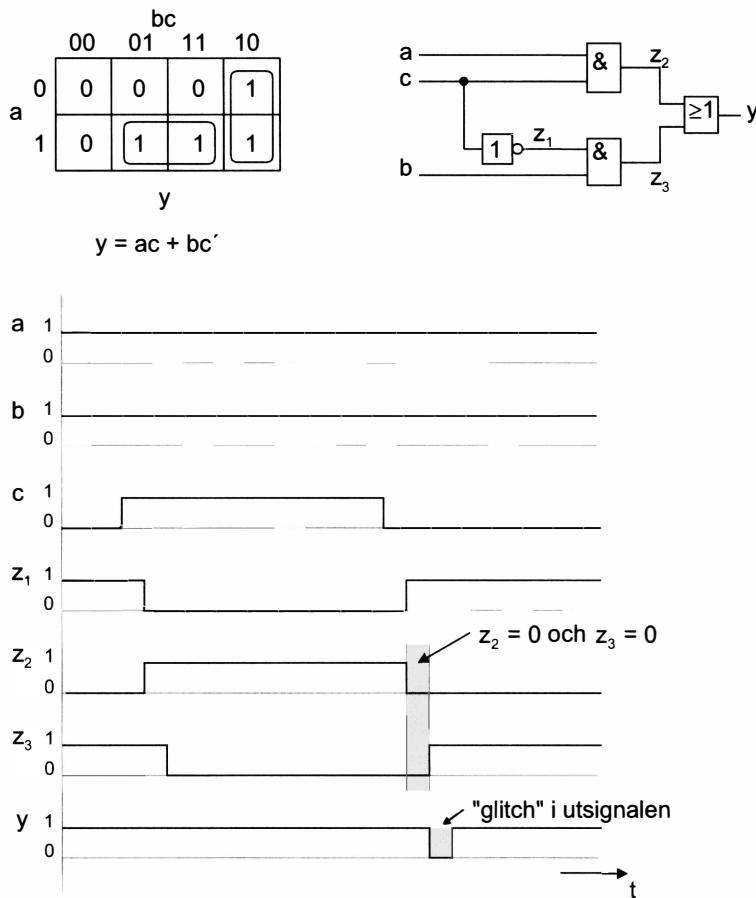
### Minimering av kombinationsnät med flera utgångar

Quine-McCluskey kan även användas för minimering av flera funktioner samtidigt med beaktande av grinddelning. Man genererar då primimplikatorer inte bara för funktionerna var för sig, utan även för funktionernas "snittmängder". Exempelvis om man har tre funktioner  $f_1$ ,  $f_2$  och  $f_3$ , bildar man på vanligt sätt primimplikatorerna för funktionerna var för sig, men också för funktionerna  $f_1f_2$ ,  $f_1f_3$ ,  $f_2f_3$  och  $f_1f_2f_3$ . Vi går inte närmare in på metoden här.

## 4.5 Kombinationskretsar i tidsplanet

### Hasard

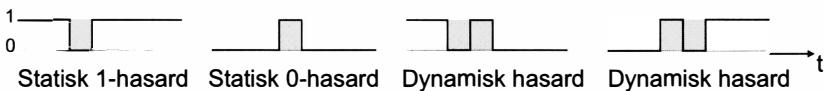
Låt oss studera kombinationskretsen nedan i tidsplanet.



Figur 4.39 Kombinationskrets med hasard.

Som framgår av funktionens  $y$  Karnaughdiagram och booleska uttryck, så är  $y = 1$  om  $a = b = 1$  oavsett värdet hos  $c$ . Antag nu att  $a = b = 1$  och att  $c$  ändras enligt  $0 \rightarrow 1 \rightarrow 0$ . Om vi antar att grindarna har samma födröjning, då blir tidsdiagrammet enligt figuren ovan. Vi ser i tidsdiagrammet att signalerna  $z_2$  och  $z_3$  kortvarigt blir lika med 0 samtidigt, under en tid som bestäms av grindarnas födröjning, medförande att utsignalen efter födröjningen i ELLER-grinden kortvarigt antar värdet 0. Det blir en oönskad ”spik”, på engelska ofta benämnd *glitch*, i utsignalen. Fenomenet brukar benämñas *hasard* (eng. *hazard*) och är alltså formellt kortvariga icke önskade värden i utsignalen hos en kombinationskrets, då insignalen ändras från ett värde till ett annat.

Man brukar skilja på olika typer av hasard, *statisk hasard* och *dynamisk hasard*, och för statisk hasard, *statisk 1-hasard* och *statisk 0-hasard*. Hasarden i grindnätet ovan är en statisk 1-hasard, innebörande att en signal som skall vara 1, kortvarigt antar värdet 0. En statisk 0-hasard innebär att en signal som skall vara 0, kortvarigt antar värdet 1. Dynamisk hasard innebär att en signal som skall ändra värde från 0 till 1 eller från 1 till 0 ändrar värde tre gånger eller fler.

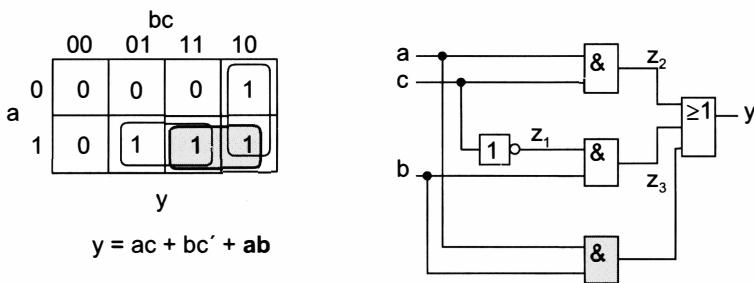


Figur 4.40 Hasard.

Hasard kan ställa till problem i vissa sammanhang och orsaka felaktigt betende hos en krets och måste beaktas vid konstruktion av digitala kretsar, t.ex. vid konstruktion av asynkrona sekvenskretsar som behandlas i kapitel 5. Som namnet hasard antyder så medför den en osäker funktion hos en krets. Under vissa födröjningar hos de ingående komponenterna så kan kretsen fungera korrekt, medan samma krets uppbyggd med andra komponenter ej fungerar korrekt.

*Hasard* i en kombinationskrets kan elimineras genom att den booleska funktionen realiseras med samtliga primimplikatorer.

Funktionen  $y$  ovan kan realiseras hasardfri, enligt figur 4.41 nedan, genom att även primimplikatorn ab medtages i funktionens booleska uttryck.

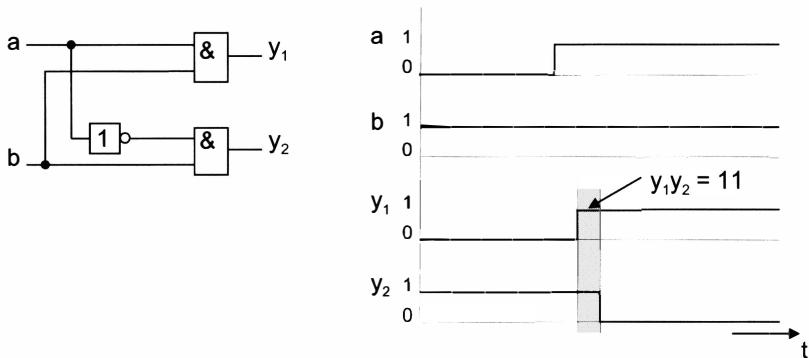


Figur 4.41 Hasardfrei realisering.

Vi ser i det nya booleska uttrycket för funktionen  $y$  ovan, att termen  $ab$  som lagts till det ursprungliga uttrycket är konsensus till de två första termerna. Om nu i de nya funktionsuttrycket  $a = b = 1$ , så är alltså  $ab = 1$  och således  $y = 1$  oavsett värdet och förändringar hos  $c$ .

## Kapplöpning

Låt oss studera kombinationskretsen nedan.



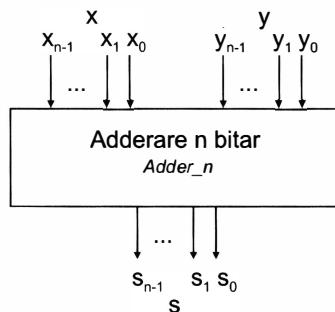
Figur 4.42 Kombinationskrets med kapplöpning.

Antag nu i kombinationskretsen ovan att  $a = 0$  och  $b = 1$ , varvid alltså  $y_1 = 0$  och  $y_2 = 1$ . Om nu  $a$  ändras till 1, så skall utsignalerna bli  $y_1 = 1$  och  $y_2 = 0$ , och ändringen sker enligt signaldiagrammet i figur 4.42 ovan. Utsignalerna  $y_1y_2$  ändras från 01 till 10, vilket sker enligt  $01 \rightarrow 11 \rightarrow 10$ . Under ändringen av utsignalerna uppkommer alltså mellanliggande utgångsvärden. Fenomenet brukar benämnas *kapplöpning* (eng. *race*) och kan inte undvikas när mer än en utsignal skall ändras. I detta fall skall två utsignaler ändra värde och en av utsignalerna kommer alltid att ändra sig före den andra, vinna kapplöpningen, i detta fall utsignal  $y_1$ . Med en annan konstruktion av kretsen så skulle ändringen 01 till 10 lika väl kunnat ske enligt  $01 \rightarrow 00 \rightarrow 10$ . Kapplöpning kan alltså inte undvikas när mer än en utsignal skall ändras och om det ställer till problem i en krets, så får det lösas genom att kretsen konstrueras om så att en ändring av en insignal alltid resulterar i ändring av högst en utsignal.

## 4.6 Adderare

Adderaren är en mycket viktig kombinationskrets, som förekommer i bl.a. fickräknare, datorer, mikrostyrkretsar och signalprocessorer. Den är ”kärnan” i *Aritmetiska enheten* (eng. *Arithmetic Unit, AU*), i vilken utförs aritmetiska operationer addition, subtraktion, multiplikation och division.

En adderare som kan bilda summan  $x + y = s = (s_{n-1}, \dots, s_1, s_0)$  av två binära tal  $x = (x_{n-1}, \dots, x_1, x_0)$  och  $y = (y_{n-1}, \dots, y_1, y_0)$  är en kombinationskrets med  $2n$  insignalter och  $n$  utsignalter enligt figuren nedan. Den skulle kunna konstrueras enligt tidigare som ett två-nivå OCH-ELLER-nät, NAND-NAND-nät e.dyl, men redan för relativt få bitar hos de binära talen som skall adderas får detta nät en sådan komplexitet att det inte är möjligt att realisera kretsen med två nivåer. Ett två-nivå-nät karakteriseras av att det är mycket snabbt. Gör vi avkall på snabbheten och ökar djupet, så kan komplexiteten fås ned till en acceptabel nivå, ”tid är pengar”. Additionsprocessens *iterativa* natur, dvs att samma förfarande upprepas för varje position i talet, bildande av en summasiffra och en minnessiffra enligt additionsexemplet nedan, kan utnyttjas för uppbyggnad av en adderare.



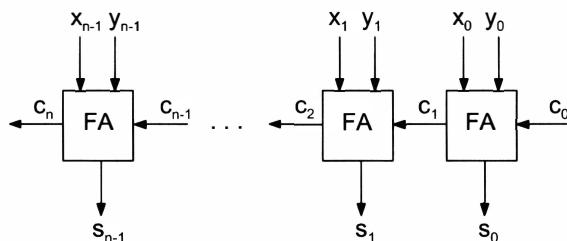
Figur 4.43 Adderare – kombinationskrets.

Additionen av de två 8-bitars talen  $10011010 + 01011011$  kan utföras enligt

$$\begin{array}{r} x \\ y \\ \hline s \end{array} \quad \begin{array}{r} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ + 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{array} \quad \begin{array}{r} 1 \\ 1 \\ + \\ \hline 5 & 4 \\ 9 & 1 \\ \hline 2 & 4 & 5 \end{array}$$

*Minnessiffra* (eng. *carry*) inträffar för  $1+1 = 10_2$  och för  $1+1+1 = 11_2$ , och behandlas på samma sätt som i det decimala talsystemet.

Adderaren kan således byggas upp av ett antal likadana delnät, s.k. *heladderare* (eng. *full adder*, FA). Delnäten benämnes heladderare p.g.a. att näten även adderar föregående minnessiffra, ett nät som bara adderar x och y brukar benämns halvadderare (HA). Adderaren är uppbyggd för att bara kunna addera två tal och så konstrueras adderare normalt. Addition av flera tal utförs genom upprepade additioner och lagring av mellanresultat.



Figur 4.44 Adderare uppbyggd med heladderare (FA).

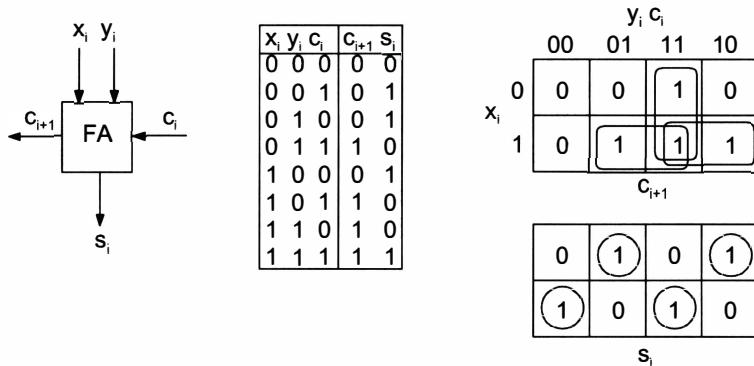
En heladderare kan konstrueras på vanligt sätt som en kombinationskrets med två nivåer, enligt figuren nedan.

Ur Karnaughdiagrammen erhålls

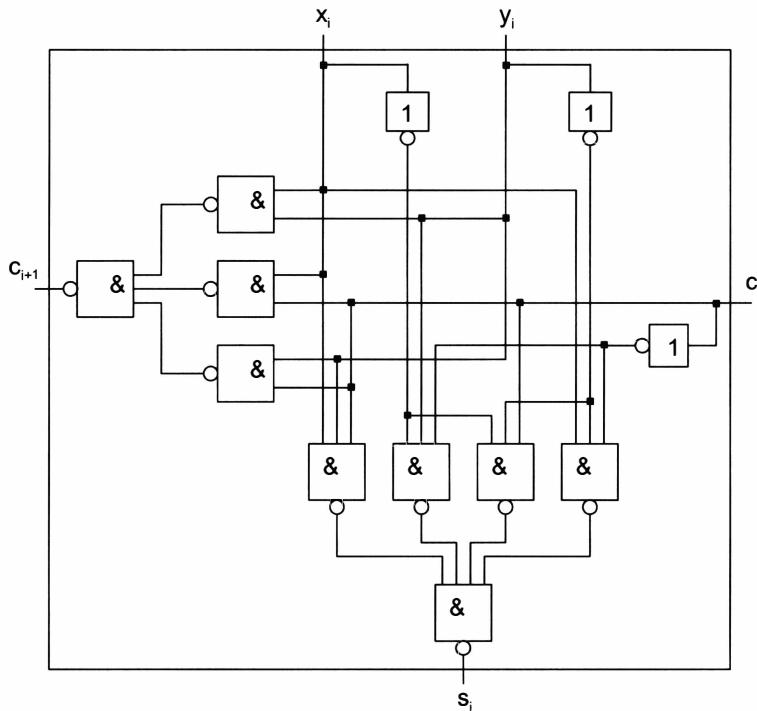
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

$$s_i = x_i' y_i' c_i + x_i' y_i c_i' + x_i y_i' c_i' + x_i y_i c_i$$

Heladderaren uppbyggd med NAND-grindar visas i figur 4.46 nedan.

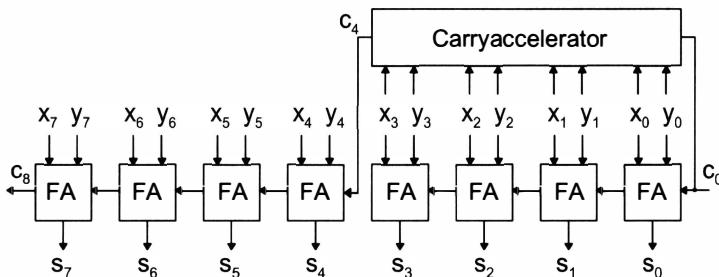


Figur 4.45 Funktionstabell och Karnaghdiagram för heladderaren (FA).



Figur 4.46 Grindnät NAND-NAND till heladderaren (FA).

För en adderare uppbyggd med kaskadkopplade heladderare enligt principen i figur 4.44 är antalet grindnivåer är proportionellt mot antalet bitar. Om adderaren realiseras med heladderare uppbyggda med grindar enligt figur 4.46 ovan, blir längsta signalvägen  $c_0 \rightarrow c_1 \rightarrow \dots \rightarrow c_{n-1} \rightarrow s_{n-1}$ , där signalen har att passera inte mindre än  $(n - 1) \cdot 2 + 3 = 2n + 1$  grindnivåer. Om fördräjningen per grind är exempelvis 2 ns, så tar det för en adderare med 64 bitar, 129 ns från det att insignalerna x och y appliceras tills summan s bildats. Den kritiska signalen i adderaren är alltså minnessiffran som måste transporteras hela vägen från LSB till MSB. Med speciella nät, s.k. *carry-acceleratorer* (eng. *look-ahead carry generator*), kan sista minnessiffran för en grupp av heladderare bildas betydligt snabbare än i den vanliga adderaren. Principen illustreras i figuren nedan.



Figur 4.47 Adderare för 8 bitar försedd med carryaccelerator.

Antag att carryacceleratoren i figuren ovan är realiseras i ett grindnät med två nivåer. Maximala antalet nivåer i adderaren blir då 11, ty signalvägen  $c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow s_3$  har 9 nivåer och  $c_0 \rightarrow c_4 \rightarrow c_5 \rightarrow c_6 \rightarrow c_7 \rightarrow s_7$  har 11 nivåer. Utan carryacceleratoren blir maximala antalet nivåer 17. Vid större antal bitar kan flera carryacceleratorer användas.

Adderaren är en kritisk komponent i en dator. I den utförs alla aritmetiska operationer, inte bara addition utan även subtraktion, multiplikation och division. Subtraktion utförs nämligen, som vi skall se i nästa sektion, som addition. Multiplikation utförs som ett antal additioner och division som ett antal subtraktioner, dvs. ett antal additioner. För att aritmetiska operationer skall kunna utföras snabbt i en dator är det sålunda viktigt att adderaren är snabb.

## 4.7 Aritmetisk Logisk Enhet (ALU)

I denna sektion skall vi studera aritmetik med n-komplementet och speciellt aritmetik med 2-komplementet, representation av negativa tal och BCD-aritmetik. Endast addition och subtraktion av heltal kommer att behandlas. Vi skall studera uppbyggnaden av en Aritmetisk Enhet och en Logisk Enhet och avslutningsvis skisseras en Aritmetisk Logisk Enhet (ALU).

### Aritmetik

#### Aritmetik med n-komplementet

Addition och subtraktion av binära heltal berördes kort i kapitel 1. Låt oss börja med att visa en subtraktion ”på vanligt sätt” av två 8-bitars tal. Talen i den fortsatta framställningen är normalt binärtal, dock förekommer ibland i texten decimaltal, men av sammanhanget bör det framgå vilken typ av tal som avses. I vissa fall anges för tydlighets skull talets bas som index, exempelvis  $2_{10} = 10_2$  (decimaltalet 2 är lika med binärtal 10).

#### Subtraktion

Subtraktionen av de två 8-bitars talen  $10001001 - 01010100$  kan utföras enligt nedan

$$\begin{array}{r}
 & \begin{array}{c} 10 \\ \uparrow \\ 0 \end{array} \quad 1 \\
 x & \begin{array}{r} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{array} & & & & & & \\
 y & \begin{array}{r} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{array} & & & & & & \\
 \hline
 s & \begin{array}{r} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{array} & & & & & &
 \end{array}
 \qquad
 \begin{array}{r}
 & \begin{array}{c} 10 \\ \uparrow \\ 3 \end{array} \quad 7 \\
 & \begin{array}{r} 8 \\ 4 \end{array} & & \\
 \hline
 & \begin{array}{r} 5 \\ 3 \end{array} & &
 \end{array}$$

*Lånesiffra* (eng. *borrow*) behandlas på samma sätt som vid subtraktion av decimala tal. I det decimala talsystemet med *basen* 10 lånan man en enhet från positionen till vänster om den aktuella positionen, som då blir värd 10 gånger så mycket då den flyttas ett steg åt höger. I det binära talsystemet med basen 2 lånan man också en enhet från positionen till vänster, som då blir värd 2 gånger så mycket då den flyttas ett steg åt höger. Oavsett talets bas b, så lånan man alltså alltid  $10_b$ .

Liksom för addition utförs subtraktion av bara två tal i taget. Subtraktion av flera tal utförs genom upprepade subtraktioner och lagring av mellanresultat.

Subtraktionsenheter konstrueras normalt inte för att utföra subtraktion enligt principen ovan. *Subtraktion utförs istället såsom en addition!*, såsom addition av det s.k. *2-komplementet* till talet som skall subtraheras. Om det ur kretssynpunkt är enkelt att realisera 2-komplementet, vilket vi nedan skall se är fallet, är det uppenbart fördelaktigt att utföra subtraktion på detta sätt, eftersom då en additionsenhet kan användas för både addition och subtraktion, och inte nog med det, utan eftersom multiplikation är ett antal additioner och division är ett antal subtraktioner, så kan alltså alla fyra räknesätten utföras i en additionsenhet.

**2-komplementet** till ett binärtal bildas enligt regeln:

1. Invertera samtliga bitar i talet. (4.10)
2. Addera 1 till det inverterade talet.

Av definitionen (4.10) ovan inses att det ur kretssynpunkt bör vara relativt enkelt att realisera 2-komplementet. För realisering av steg 1 krävs bara lika många inverterare som bitar i talet, och för realisering av steg 2 kan ettan adderas i den minst signifikanta positionen i additionsenheten i t.ex. ingången för minnessiffran  $c_0$ .

### Exempel 4.13

Bilda 2-komplementet till binärtal 01010100.

*Lösning*

$$\begin{array}{r}
 & \underline{1} \quad 1 \\
 & 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 & \text{steg 1} \\
 + & \underline{\quad \quad \quad \quad \quad \quad \quad \quad \quad} & \underline{1} & \text{steg 2} \\
 \hline
 & 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0
 \end{array}$$

□

*Alternativ regel för att bilda 2-komplementet :*

1. Leta upp den minst signifikanta ettan i binärtal. (4.11)
2. Invertera alla bitar till vänster om denna.

Att denna regel ger samma resultat som regel 4.10 inses lätt. Nollorna till höger om den minst signifikanta ettan kommer i steg 1 att inverteras och bli ettor, men kommer sedan åter att bli nollar vid additionen av ettan i steg 2. Minnessiffran vid additionen av ettan i steg 2 kommer att ”dö ut” vid den minst signifikanta nollan i det inverterade talet, dvs vid den minst signifikanta ettan i det ursprungliga talet, som alltså åter blir en etta. Siffrorna till vänster om denna etta förblir inverterade. Denna princip att bilda 2-komplementet används inte vid realisering av en additions-/subtraktionsenhet, utan då används principen i regel (4.10), men den alternativa regeln (4.11) är lämplig att använda då man snabbt i huvudet skall bilda 2-komplementet.

### **Exempel 4.14**

Utför subtraktionen 10001001 - 01010100 som utfördes på ”vanligt” sätt ovan, som addition av 2-komplementet.

#### *Lösning*

$10001001 + {}^201010100$  (vi betecknar 2-komplementet till ett binärtal  $N$  med  ${}^2N$ ).

$$\begin{array}{r}
 & & 1 & \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 + & \underline{1} & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
 & \underline{+} & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1
 \end{array}
 \quad {}^201010100$$

Minnessiffran som erhålls vid additionen av de två mest signifikanta siffrorna ingår ej i resultatet. Som synes blir resultatet detsamma som vid den ”vanliga subtraktionen”.

□

Subtraktion genom addition av 2-komplementet verkar kanske så här långt vara trolleri med siffror, men bakgrunden är ganska enkel, som vi skall se i det följande.

För att formellt bevisa subtraktion genom addition av 2-komplementet, ger vi nu en rent *aritmetisk definition av 2-komplementet*.

**2-komplementet**  ${}^2N$  till ett binärtal  $N$  med  $n$  heltalssiffror definieras som:

$${}^2N = 2^n - N \quad (4.12)$$

Subtraktionen  $M - N$  av två n-bitars binärtal M och N kan utföras som additionen av 2-komplementet, ty

$$\begin{aligned}
 M - N &= M - N + 0 \\
 &= M - N + 2^n - 2^n \\
 &= M + (2^n - N) - 2^n \\
 &\quad \uparrow \quad \uparrow \\
 2\text{-komplementet till } N &\qquad \text{minnessiffran som stryks i resultatet}
 \end{aligned}$$

2-komplementet till 8-bitars binärtal 10101011 i exemplet 4.14 ovan bildat enligt definition 4.12 blir:

$$\begin{aligned}
 {}^210101011 &= 2^8 - 10101011 \\
 &= 100000000 - 10101011 \\
 &= (11111111 + 1) - 10101011 \\
 &= (11111111 - 10101011) + 1 \\
 &\quad \uparrow \quad \uparrow \\
 \text{steg 1 i def (4.10)} &\quad \text{steg 2 i def (4.10)}
 \end{aligned}$$

Inverteringen motsvaras här alltså av  $(11111111 - 10101011)$ , dvs för varje siffra i talet bildas ”1 minus siffran”, vilket benämnes *1-komplementet*. (Observera ovan att  $2^8$  är ett 9-bitars tal!). Vi ser alltså att definitionerna 4.10 och 4.12 är ekvivalenta.

Definitionen 4.12 gäller även för bråktalsdelen till ett binärtal.

### ***Exempel 4.15***

Bilda 2-komplementet till talet 0.10110110 (obs! talet har en heltalssiffra).

Lösning

$$\begin{aligned}
 {}^20.10110110 &= 2^1 - 0.10110110 \\
 &= 10.00000000 - 0.10110110 \\
 &= (1.11111111 + 0.00000001) - 0.10110110 \\
 &= 1.11111111 - 0.10110110 + 0.00000001 \\
 &= 1.01001001 + 0.00000001 \\
 &= 1.01001010
 \end{aligned}$$

□

**Exempel 4.16**

Utför subtraktionen  $0.11101011 - 0.10110110 = 0.00110101$  som addition av 2-komplementet.

*Lösning*

$$\begin{array}{r} 0.1\ 1\ 1\ 0\ 1\ 0\ 1\ 1 \\ + 1.0\ 1\ 0\ 0\ 1\ 0\ 1\ 0 \\ \hline 1\ 0.0\ 0\ 1\ 1\ 0\ 1\ 0\ 1 \end{array} \quad ^20.10110110$$

□

2-komplementet till 2-komplementet till ett binärtal ger tillbaka det ursprungliga talet, dvs

$$^2(^2N) = N \quad (4.13)$$

$$\text{ty } ^2(^2N) = 2^n - (2^n - N) = N$$

Komplementaritmetik är inte förbehållet det binära talsystemet, utan kan användas för vilken bas som helst. Sålunda kan i det decimala talsystemet aritmetik utföras med *10-komplementet* enligt följande.

*10-komplementet*  ${}^{10}N$  till ett decimaltal  $N$  med  $n$  heltalssiffror definieras:

$${}^{10}N = 10^n - N \quad (4.14)$$

Subtraktionen  $M - N$  av två  $n$ -siffrors decimaltal  $M$  och  $N$  kan utföras som addition av 10-komplementet, ty

$$\begin{aligned} M - N &= M - N + 0 \\ &= M - N + 10^n - 10^n \\ &= M + (10^n - N) - 10^n \\ &\quad \uparrow \qquad \uparrow \\ &\quad \text{10-komplementet till talet } N \qquad \text{minnessiffran som stryks i resultatet} \end{aligned}$$

*10-komplementet* till ett decimaltal bildas enligt regeln:

1. Ersätt varje siffra i talet med 9-komplementet till siffran, dvs "9 minus siffran".
  2. Addera 1 till det så erhållna talet.
- (4.15)

**Exempel 4.17**

Bilda 10-komplementet till 5-siffrors decimaltalet 17396 enligt definition 4.14.

*Lösning*

$$\begin{aligned}
 {}^{10}17396 &= 10^5 - 17396 \\
 &= 100000 - 17396 \\
 &= (99999 + 1) - 17396 \\
 &= (99999 - 17396) + 1 \\
 &= 82603 + 1 \\
 &= 82604
 \end{aligned}$$

□

Vi ser av exemplet ovan att definitionerna 4.14 och 4.15 är ekvivalenta.

**Exempel 4.18**

Utför subtraktionen  $7492 - 3879 = 3613$  som addition av 10-komplementet.

*Lösning*

$$\begin{array}{r}
 7\ 4\ 9\ 2 \\
 + 6\ 1\ 2\ 1 \\
 \hline
 4\ 3\ 6\ 1\ 3
 \end{array} \quad {}^{10}3879$$

□

**Representation av negativa tal**

Positiva och negativa tal brukar ju markeras med plustecken (+) respektive minustecken (-), oftast sätter man inte ut plustecknet utan använder konventionen att inget tecken markerar positivt tal. I digitaltekniken där tal representeras med de binära siffrorna 0 och 1, måste också talets tecken representeras med dessa siffror, med en *teckenbit* (eng. *sign bit*), som naturligt placeras längst till vänster i talet. Normalt används representationen teckenbit 0 för positiva tal och teckenbit 1 för negativa tal. Undantag är s.k. "binär-offset-representation", där det är tvärtom.

Det vore nu naturligt att låta bitarna till höger om teckenbiten utgöra talets belopp, s.k. "tecken-belopp-representation". Normalt används emellertid inte denna representation i aritmetiska sammanhang p.g.a. att den leder till en relativt komplicerad realisering av en additions-/subtraktionsenhets. Efter vad som tidigare sagts om fördelen med att utföra subtraktion som addition av 2-komplementet, inser man kanske att det är lämpligare att låta *negativa*

*tal representeras som 2-komplementet till motsvarande positiva tal* (inklusive teckenbiten).

Låt oss som exempel anta att vi har 8 bitar till vårt förfogande för att skapa ett talområde omfattande positiva och negativa heltal. Med 8 bitar är det möjligt att representera  $2^8 = 256$  olika tal och skapa ett heltalsområde från -128 till +127. I tabellen nedan visas några heltal i detta talområde där de negativa talen representeras som 2-komplementet till motsvarande positiva tal.

Tabell 4.11: Talområde -128 till +127 med 8-bitars tal.

Decimaltal	Binärtal
+127	01111111
+126	01111110
+125	01111101
:	:
+4	00000100
+3	00000011
+2	00000010
+1	00000001
0	00000000
-1	11111111
-2	11111110
-3	11111101
-4	11111100
:	:
-125	10000011
-126	10000010
-127	10000001
-128	10000000

Talområdet ovan med heltalen -128 till +127 förekommer exempelvis i programspråket C, med datatypen benämnd *signed char* och i programspråket Java, med datatypen benämnd *byte*.

Med de negativa talen representerade som 2-komplementet till motsvarande positiva tal kan de behandlas på exakt samma sätt som de positiva talen, de kan adderas på vanligt sätt och subtraheras genom addition av 2-komplementet. Nedan visas några exempel på operationerna addition och subtrakt-

tion av positiva och negativa tal i talområdet -128 till +127 hämtade ur tabellen ovan. (Håll isär operationen och talets tecken!).

Tabell 4.12: Exempel på addition och subtraktion i talområdet -128 till +127

Operation addition		Operation subtraktion	
127 + -2 125	01111111 + 11111110 101111101	2 - 1 3	00000010 + 00000001 ( <sup>2</sup> 11111111) 00000011
125 + -128 -3	01111101 + 10000000 11111101	-127 - 2 -125	10000001 + 00000010 ( <sup>2</sup> 11111110) 10000011
3 + -3 0	00000011 + 11111101 100000000	2 - 127 -125	00000010 + 10000001 ( <sup>2</sup> 01111111) 10000011
-1 + -2 -3	11111111 + 11111110 111111101	-100 - 100 -200 (< -128!)	10011100 + 10011100 ( <sup>2</sup> 01100100) 100111000

Låt oss närmare analysera uppkomsten av minnessiffran och vad som indikerar att talområdet överskrids vid additioner och subtraktioner. – Ett negativt tal M betecknas som  $-M$ , vilket alltså är 2-komplementet till M, dvs.  $-M = {}^2M = 2^n - M$ . (Håll isär operationen och talets tecken i analysen nedan!).

### Addition

#### Resultatet inom talområdet

1.  $M + N \geq 0$ : ingen minnessiffran.
2.  $M + -N \geq 0$ :  $M + -N = M + (2^n - N) = 2^n + (M - N) = (M - N)$ : minnessiffran.
3.  $M + -N < 0$ :  $M + -N = M + (2^n - N) = 2^n - (N - M) = -(N - M)$ : ingen minnessiffran.
4.  $-M + -N < 0$ :  $-M + -N = (2^n - M) + (2^n - N) = 2^n + (2^n - (M + N)) = -(M + N)$ : minnessiffran

#### Resultatet utanför talområdet

För att detta skall kunna inträffa måste båda talen ha samma tecken, antingen båda talen positiva eller båda talen negativa.

1.  $M + N$ : ingen minnessiffra, men minnessiffra till teckenbiten, så att resultatet utanför talområdet även får felaktigt, negativt tecken.

2.  $-M + -N$ : minnessiffra, men ingen minnessiffra till teckenbiten, så att resultatet utanför talområdet även får felaktigt, positivt tecken.

### Subtraktion

*Resultatet inom talområdet*

1.  $M - N \geq 0$ :  $M - N = M + (2^n - N) = 2^n + (M - N) = (M - N)$ : minnessiffra.

2.  $M - N < 0$ :  $M - N = M + (2^n - N) = 2^n - (N - M) = -(N - M)$ : ingen minnessiffra.

3.  $M - -N \geq 0$ :  $M - -N = M + (2^n - (2^n - N)) = (M - N)$ : ingen minnessiffra.

4.  $-M - -N < 0$ :  $-M - -N = (2^n - M) + (2^n - (2^n - N)) = 2^n - (M - N) = -(M - N)$ : ingen minnessiffra.

*Resultatet utanför talområdet*

För att detta skall inträffa måste talen ha olika tecken.

1.  $M - -N$ : ingen minnessiffra, men minnessiffra till teckenbiten, så att resultatet även får felaktigt tecken.

2.  $-M - N$ : minnessiffra, men ingen minnessiffra till teckenbiten, så att resultatet utanför talområdet även får felaktigt tecken.

Identifiera de tidigare additions- och subtraktionsexemplen bland fallen ovan och gör även själv exempel till de fall som inte förekommer bland exemplen.

Programspråket Java innehåller ett antal olika datatyper för heltal. De negativa heltalen representeras som 2-komplementet till motsvarande positiva heltal. Datatypernas antal bytes och talområde visas nedan.

Datotyp	Antal bytes	Talområde
byte	1	-128 till +127
short	2	-32768 till +32767
int	4	-2147483648 till +2147483647
long	8	$\approx -9 \cdot 10^{18}$ till $\approx +9 \cdot 10^{18}$

### BCD-aritmetik

Aritmetiska operationer i en dator utförs normalt i binärarytmetik enligt ovan, med talen representerade som rena binärtal. Vid in- och utmatning av tal i digitala system representeras ofta talets siffror i BCD-kod (*Binary Coded Decimal*), se tabell 1.3 i kapitel 1. Efter inmatning av tal i BCD-kod utförs omvandling till ren binärkod, varefter aritmetik enligt ovan kan ske. Före utmatningen av talen utförs omvandling till BCD-kod, varefter ut-

matning och presentation kan ske. I vissa enstaka fall förekommer dock att man gör aritmetik med talen i BCD-kod, dvs man gör decimalaritmetik. Låt oss studera vilka problem man då måste beakta.

Antag att decimaltalen 27 och 54, representerade i BCD-kod som 0010 0111 respektive 0101 0100, skall adderas som två 8-bitars tal med summan bibehållen i BCD-kod. Additionen av de två 8-bitartalen utförd på vanligt sätt, blir då

$$\begin{array}{r} & \frac{1}{0\ 0\ 1\ 0\ 0\ 1\ 1\ 1} \\ + & \frac{0\ 1\ 0\ 1\ 0\ 1\ 0\ 0}{0\ 1\ 1\ 1\ 1\ 0\ 1\ 1} \end{array} \qquad \begin{array}{r} & \frac{1}{2\ 7} \\ + & \frac{5\ 4}{8\ 1} \end{array}$$

Additionen av de två BCD-talen ovan ger som synes ej det önskade resultatet 1000 0001, talet 81, utan istället 0111 1011, talet "sjuttioelva". Uppenbart uppstår ett felaktigt resultat då additionen av två BCD-siffror ger en summa som är större än 9, dvs ej är en BCD-siffra. Det felaktiga resultatet kan emellertid justeras enligt nedan.

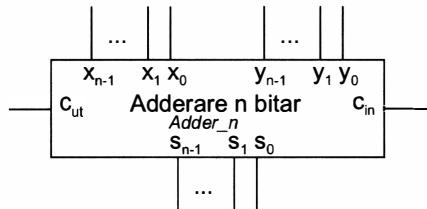
Antag att summan av två BCD-siffror blir 1010, dvs talet tio. I BCD-kod skall summan egentligen vara 0001 0000, decimaltalet 10. Det korrekta resultatet kan erhållas om till det felaktiga resultatet adderas talet 6, ty  $1010 + 0110 = 1\ 0000$ . Uppenbart gäller alltså att talet 6 skall adderas till resultsiffran om binärtalat för denna är större än 9 och mindre eller lika med 15. Emellertid skall talet 6 även adderas om binärtalat för summan av de två BCD-siffrorna är större än 15, dvs då det uppstår en minnessiffra vid additionen. Om exempelvis BCD-siffrorna 8 och 9 adderas blir resultatet 17 i ren binärkod, dvs 1 0001, och även i detta fall skall alltså talet 6 adderas, vilket ger  $1\ 0001 + 0110 = 1\ 0111$ , dvs det korrekta resultatet 17 i BCD-kod.

Sammanfattningsvis skall alltså justering av resultatet vid addition av två tal i BCD-kod göras, då binärtalat för resultsiffran är större än nio eller då en minnessiffra uppstår vid additionen av de två siffrorna. I datorer finns normalt en instruktion för att justera resultatet av en BCD-addition enligt principen ovan.

Subtraktion av två tal i BCD-kod kan utföras som en addition av 10-komplementet, som tidigare behandlats, varvid då 9-komplementet till varje siffra bildas med hjälp av en tabell. Justering av resultatet av additionen görs sedan enligt regeln ovan.

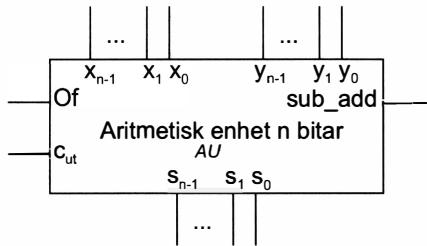
## Aritmetisk enhet

Adderaren behandlades i sektion 4.6. Låt oss införa symbolen nedan för en n-bitars adderare.



Figur 4.48 Symbol för en n-bitars adderare.

En *Aritmetisk enhet* (eng. *Arithmetic Unit*, AU) som kan utföra addition och subtraktion av två n-bitars tal, skall nu konstrueras. Vi använder symbolen nedan.



Figur 4.49 Symbol för en Aritmetisk enhet (AU).

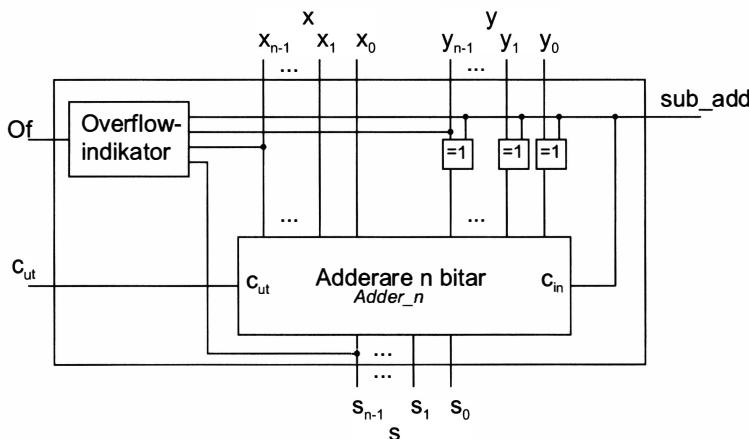
Insignalen *sub\_add* bestämmer om operationen skall vara addition eller subtraktion enligt

$$\begin{aligned} \text{sub\_add} = 0 &\Rightarrow \text{addition, } s = x + y \\ \text{sub\_add} = 1 &\Rightarrow \text{subtraktion, } s = x - y \end{aligned}$$

Utsignalen  $c_{ut}$  är samma minnessiffra som i adderaren i figur 4.47. Utsignalen  $Of$  (eng. *overflow*), indikerar om talområdet överskrider, ( $Of = 1$ ), eller inte överskrider, ( $Of = 0$ ), vid addition och subtraktion av tal i ett talområde med både positiva och negativa tal. Vi kommer längre fram att studera hur dessa signaler skall realiseras.

I avsnittet Aritmetik såg vi hur subtraktion kan utföras som addition av 2-komplementet. I Aritmetiska enheten kan subtraktionen  $s = x - y$  utföras som addition av 2-komplementet, dvs  $s = x - y = x + ^2y$ . Sålunda bör en en Aritmetisk enhet innehålla en adderare för att realisera addition och subtraktion. Vidare måste finnas en enhet som kan bilda 2-komplementet till  $y$  vid subtraktion.

Vid addition,  $s = x + y$ , skall  $y$  gå direkt in till adderaren, medan vid subtraktion,  $s = x - y = x + ^2y$ , skall 2-komplementet till  $y$  gå in till adderaren. Enligt tidigare bildas ju 2-komplementet genom invertering av samtliga bitar och efterföljande addition av talet 1. Additionen av talet 1 kan göras i adderaren, ingång  $c_{in}$ , som ej används. Återstår alltså vid bildande av 2-komplementet problemet att kunna invertera  $y$ . Detta kan göras med XOR-grindar, som ju kan användas som styrbara inverterare. I figuren nedan visas uppbyggnaden av Aritmetiska enheten. Där framgår att  $sub\_add = 0$  gör att  $y$  går opåverkad genom XOR-grindarna och att  $c_{in} = 0$  medförande  $s = x + y$ , medan  $sub\_add = 1$  gör att  $y$  inverteras i XOR-grindarna och att  $c_{in} = 1$  medförande  $s = x + ^2y = x - y$ .



Figur 4.50 Aritmetisk enhet (AU).

Vi skall nu studera uppbyggnaden av overflow-indikatorn, som genererar signalen *Of*. – I avsnittet Aritmetik utredes i detalj förutsättningarna för att talområdet skall överskridas och vad som då händer med minnessiffran  $c_n$ . Där konstaterades att talområdet kan överskridas vid addition bara om båda talen har samma tecken, antingen båda talen positiva eller båda talen negativa, och vid subtraktion bara om talen har olika tecken. Om dessa förutsättningar är uppfyllda inträffar overflow, om teckenbiten  $s_{n-1}$  i resultatet  $s$  inte har samma värde som teckenbiten  $x_{n-1}$  i talet  $x$ . Funktionstabellen för signalen *Of* blir enligt nedan. Analysera noga värdet hos *Of* för de olika insignal-kombinationerna till overflow-indikatorn.

sub_add	$s_{n-1}$	$x_{n-1}$	$y_{n-1}$	Of
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	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

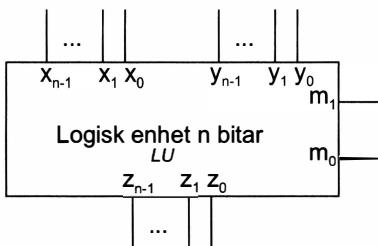
sub_add	$s_{n-1}$	$x_{n-1}y_{n-1}$			
		00	01	11	10
00	0	0	1	0	
01	1	0	0	0	
11	0	1	0	0	
10	0	0	0	1	
					Of

Figur 4.51 Funktionstabell och Karnaughdiagram för funktionen *Of* (Overflow).

Ur Karnaughdiagrammet fås booleska uttrycket för *Of* till

$$\begin{aligned} \text{Of} = & \quad \text{sub\_add}' s_{n-1}' x_{n-1} y_{n-1} + \text{sub\_add}' s_{n-1} x_{n-1}' y_{n-1}' + \\ & \quad \text{sub\_add} s_{n-1} x_{n-1}' y_{n-1} + \text{sub\_add} s_{n-1}' x_{n-1} y_{n-1}' \end{aligned}$$

## Logisk enhet



Figur 4.52 Symbol för Logisk enhet (LU).

En Logisk enhet (eng. *Logic Unit*, *LU*), symbol enligt figuren ovan, som kan utföra de logiska operationerna Invers, OCH, ELLER och XOR på två n-bitars ord ska konstrueras. Logiska operationen skall väljas med två mod-signaler  $m_0$  och  $m_1$  enligt tabellen nedan.

Tabell 4.13: Logiska operationer bestämda med signaleerna  $m_1$  och  $m_0$ .

$m_1 m_0$	Logisk operation
0 0	$z = x'$ (Invers)
0 1	$z = x \wedge y$ (OCH)
1 0	$z = x \oplus y$ (XOR)
1 1	$z = x \vee y$ (ELLER)

Logiska operationerna skall utföras *bitvis* (eng. *bitwise*) på n-bitars operänderna x och y enligt

$$\text{Invers: } z_i = x_i'$$

$$\text{OCH: } z_i = x_i \wedge y_i$$

$$\text{XOR: } z_i = x_i \oplus y_i$$

$$\text{ELLER: } z_i = x_i \vee y_i$$

**Exempel 4.19**

Utför bitvisa logiska operationer OCH, ELLER och XOR på de två 8-bitars orden  $x = 10110101$  och  $y = 00101110$ .

*Lösning*

$$\begin{array}{rccccc} x & 10110101 & & 10110101 & & 10110101 \\ y & \wedge \quad 00101110 & \vee \quad 00101110 & \oplus \quad 00101110 \\ z & 00100100 & 10111111 & 10011011 \end{array}$$

□

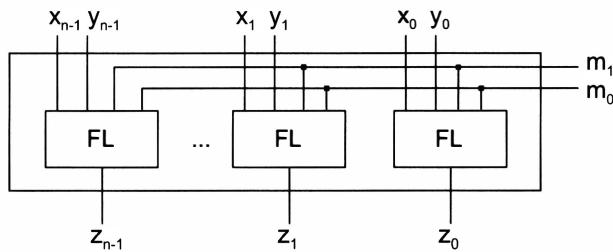
Med de bitvisa logiska operationerna kan bitar i ett ord nollställas, ettställas eller inverteras.

*Nollställning* av bitar i  $x$  sker med operationen OCH, där nollar placeras i operanden  $y$  på de platser där  $x$  skall nollställas och ettor på övriga platser i operanden  $y$  där  $x$  inte skall påverkas.

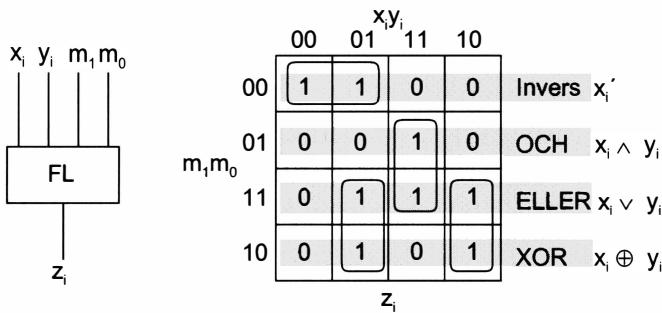
*Ettställning* av bitar i  $x$  sker med operationen ELLER, där ettor placeras i operanden  $y$  på de platser där  $x$  skall ettställas och nollar på övriga platser i operanden  $y$  där  $x$  inte skall påverkas.

*Invertering* av bitar i  $x$  sker med operationen XOR, där ettor placeras i operanden  $y$  på de platser där  $x$  skall inverteras och nollar på övriga platser i operanden  $y$  där  $x$  inte skall påverkas.

Eftersom de logiska operationerna utförs bitvis kan Logiska enheten byggas upp av  $n$  likadana delnät, enligt figuren nedan, betecknade *FL* (*Full Logic*) i analogi med heladderarna FA.



Figur 4.53 Logisk enhet (LU).



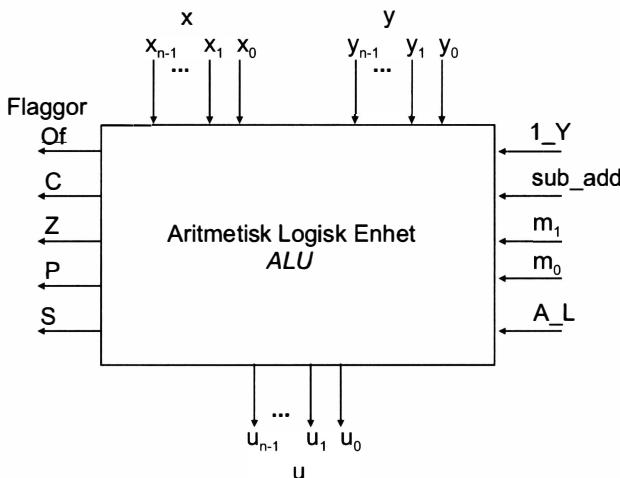
Figur 4.54 Symbol och Karnaughdiagram för en FL.

Ur Karnaughdiagrammet ovan erhålls booleska uttrycket för  $z_i$  till

$$z_i = m_1' m_0' x_i' + m_0 x_i y_i + m_1 x_i' y_i + m_1 x_i y_i'$$

I Logiska enheten (LU) i figur 4.53 ovan finns inget problem med transport av någon carry som i Aritmetiska enheten, utan signalvägen är endast vertikalt i schemat från  $x_i$ ,  $y_i$ ,  $m_1$ ,  $m_0$  till  $z_i$  och en FL kan realiseras i ett två-nivå-nät med inverterare till insignalerna.

## Aritmetisk Logisk Enhet (ALU)



Figur 4.55 Symbol för Aritmetisk Logisk Enhet (ALU)

En *Aritmetisk Logisk Enhet* (eng. *Arithmetic Logic Unit*, ALU) är en viktig del i varje dator. Den ingår i *Centralenheten* (eng. *Central Processing Unit*, CPU), mikroprocessorn, ”hjärnan” i datorn, som exekverar programmet. Nedan skall skisseras uppbyggnaden av en ALU, som bör påpekas inte visar en verlig ALU utan bara en möjlig realisering.

Funktionen hos ALUn beskrivs i tabellen nedan ( $x = \text{don't care}$ ). De vanliga operationsmoderna för logiska och aritmetiska operationer har utökats med två moder, *ökning* (eng. *increment*) *med 1* och *minskning* (eng. *decrement*) *med 1*. Orsaken är att dessa två speciella operationer förekommer så ofta i datorprogram (t.ex. i loopräknare) att det är motiverat att ha dem i en ALU. I tabellen längst till höger antyds med s.k. *mnemonics*, tänkbara namn för de olika operationerna i CPUns instruktionsrepertoar. Mnemonics, förkortningar lätt att minnas (grek. *mneme* = sv. minne), bildar *assemblyspråket*, det maskinnära programspråket.

Tabell 4.14: Operationsmoder för ALU

<b>A_L</b>	<b>sub_add</b>	<b>1_Y</b>	<b>m<sub>1</sub> m<sub>0</sub></b>	<b>Operationsmod</b>	<b>mnemonic</b>
0	x	x	0 0	$u = x'$	Invers
0	x	x	0 1	$u = x \wedge y$	ANDXY
0	x	x	1 0	$u = x \oplus y$	XORXY
0	x	x	1 1	$u = x \vee y$	ORXY
1	0	0	x x	$u = x + y$	Addition
1	1	0	x x	$u = x - y$	Subtraktion
1	0	1	x x	$u = x + 1$	Ökning med 1
1	1	1	x x	$u = x - 1$	Minskning med 1

I figuren nedan visas blockschemat för ALU:n. Där ingår naturligt en Arimetisk Enhet (AU) och en Logisk Enhet (LU) samt vidare också två multiplexrar som används för att bilda vägar för operanderna till och från AU:n och LU:n. Låt oss se lite närmare hur ALU:n fungerar.

Insignalen  $A_L$  bestämmer om ALU:n skall utföra en logisk operation ( $A_L = 0$ ) eller en aritmetisk operation ( $A_L = 1$ ). Som framgår av block-schemat så används signalen  $A_L$  som adress till en multiplexer vilken bildar en dataväg antingen från LU:n eller från AU:n till ALUs utgångar  $u$ . Sålunda görs på operanderna  $x$  och  $y$  både en logisk och en aritmetisk operation, men resultatet från en av operationerna överförs till utgångarna  $u$ .

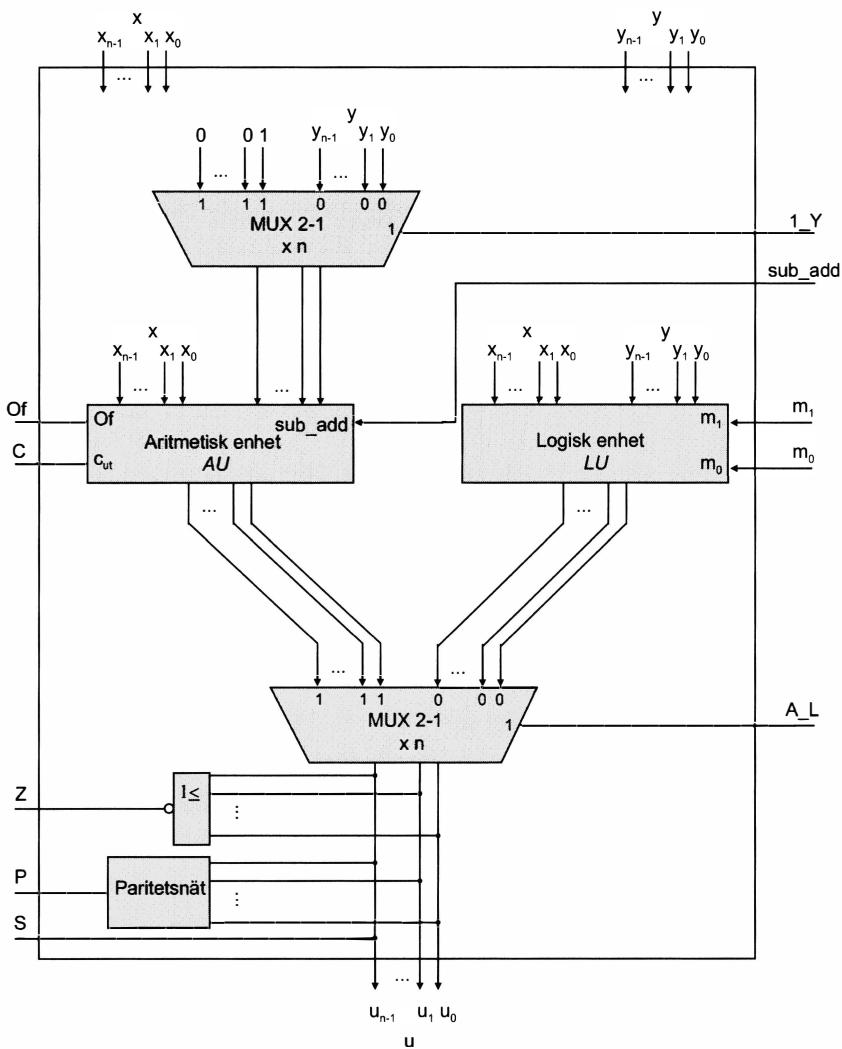
De vanliga aritmetiska och logiska operationerna behöver ingen närmare förklaring då de behandlats tidigare i samband med genomgången av AU:n och LU:n. Operationerna *ökning med 1* och *minskning med 1* utförs i ALU:n som  $x + y$  respektive  $x - y$  med operanden  $y = 1$ , vilket fås med insignalen  $1\_Y = 1$ . I den övre multiplexern är signalen  $1\_Y$  adress som väljer att till aritmetiska enheten antingen överföra operanden  $y$  ( $1\_Y = 0$ ) eller talet 1 ( $1\_Y = 1$ ).

Utsignalerna Of, C, Z, P och S från ALU:n är s.k. *flaggor*, signaler som indikerar värdefulla uppgifter hos ALU:ns resultat  $u$ . Signalerna Of och C har redan berörts för AU:n. För de övriga statussignalerna gäller att  $Z$  (*Zero*) indikerar resultatet  $u = 0$  ( $Z = 1$ ) eller  $u \neq 0$  ( $Z = 0$ ). Signalen  $P$  (*Parity*) indikerar jämn paritet, dvs. jämnt antal ettor i resultatet  $u$  ( $P = 1$ ) eller udda paritet, dvs. udda antal ettor i resultatet  $u$  ( $P = 0$ ). Signalen  $S$  (*Sign*) indikerar positivt resultat  $u$  ( $S = 0$ ) eller negativt resultat  $u$  ( $S = 1$ ). Samtliga dessa statussignaler realiseras enkelt. Signalen  $Z$  realiseras med en NOR-grind med insignalerna  $u_0, u_1, \dots, u_{n-1}$ , signalen  $P$  realiseras med XOR-grindar, se sektion 4.9 och signalen  $S$  realiseras som  $S = u_{n-1}$  (teckenbit om talområde med positiva och negativa tal används).

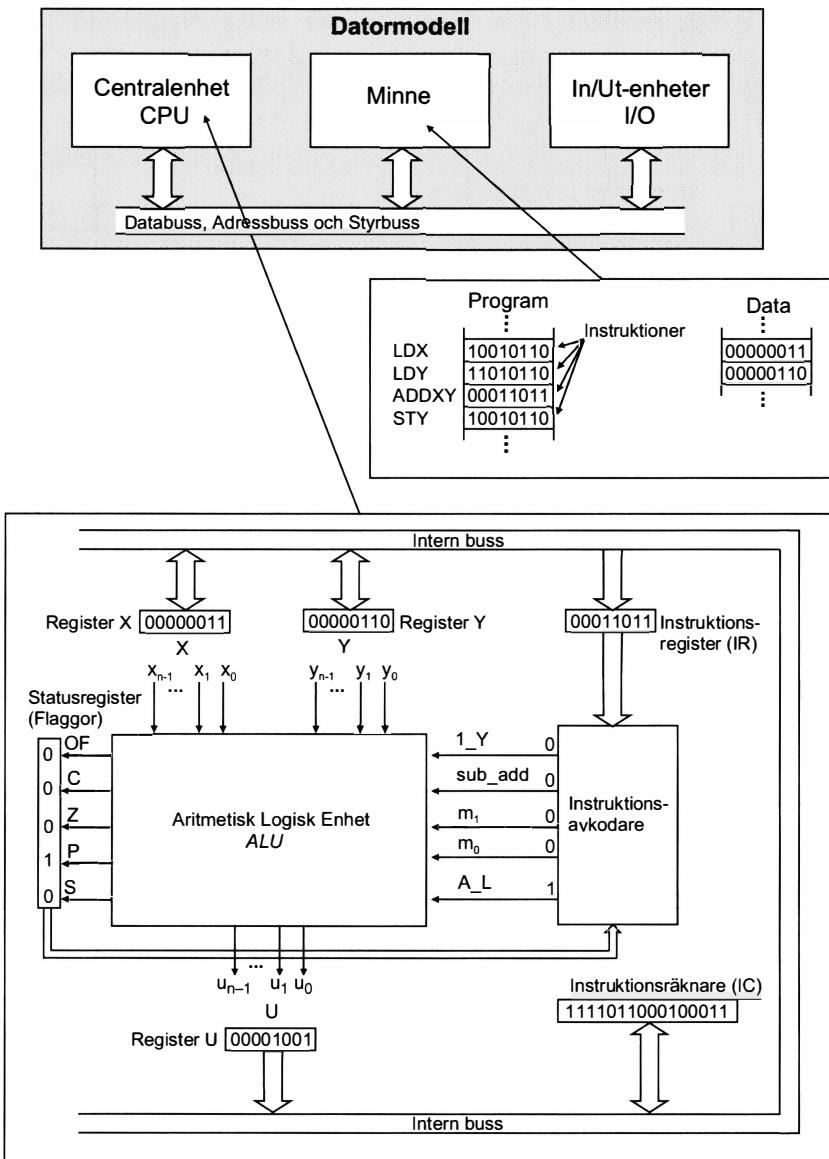
Efter figuren som visar ALU:n följer en figur som visar en enkel datormodell, där ALU:n är placerad i CPU:n. Vi avslutar denna sektion med att kort beröra datorns arbetssätt.

Program och data ligger i minnet i form av binära ord. Programmet består av *instruktioner* som normalt utförs i tur och ordning. Instruktionen innehåller uppgift om vilken *operation* som skall utföras och på vilken *operand* operationen skall utföras. En CPU:s instruktionsrepertoar, som bestäms av tillverkaren av CPU:n, omfattar runt hundra olika instruktioner. Varje instruktion representerar normalt bara en mycket enkel operation, typ addera två operander, och varje problem som skall lösas i en dator måste slutligen brytas ned i ett program bestående av sådana enkla instruktioner av binära ord som konstituerar maskinspråket, det primitivaste programspråket och det enda språk som CPU:n begriper.

I minnet i datormoden i figuren ovan visas som exempel operationskoden för några fiktiva instruktioner och bredvid tillhörande mnemonic ur assemblyspråket. LDX och LDY står för LoaD (Ladda) register X respektive Y, ADDXY för ADDera register X till register Y. Datorn exekverar programmet genom att CPUn hämtar (eng. *fetch*) en instruktion och utför (eng. *execute*) den, hämtar nästa instruktion och utför den osv.



Figur 4.56 Skiss till en Aritmetisk Logisk Enhet (ALU).



Figur 4.57 Datormodell.

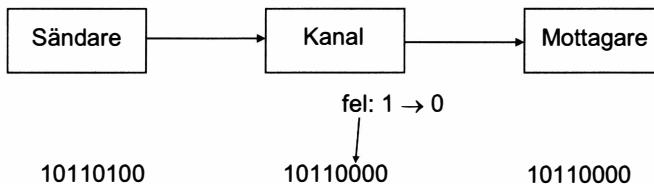
En CPU innehåller alltid vissa fundamentala enheter. Dessa är, ett *Instruktionsregister*, *IR* i vilket CPUn placerar instruktionens operationskod efter uthämtning från minnet, en *Instruktionsräknare* (eng. *Instruction Counter*, *IC*), även benämnd *Programräknare* (eng. *Program Counter*, *PC*) som håller reda på vilken instruktion i programmet som är i tur att exekveras, en *Instruktionsavkodare* (eng. *Instruction Decoder*) som avkodar operationskoden och sänder ut diverse styrsignaler till enheter i eller utanför CPUn så att den aktuella operationen utförs, en *Aritmetisk Logisk Enhet* (*ALU*) och några register för lagring av operander (i CPU:n ovan registren X, Y och U).

I programexemplet ovan har instruktionerna LDX och LDY utförts, innebärande att registren X och Y laddats med operanderna i dataarean i minnet 00000011 respektive 00000110. Instruktionens ADDXY operationskod 00011011 har sedan hämtats och lagts i instruktionsregistret i CPU:n. Utförande av denna instruktion går till så att instruktionsavkodaren sänder styrsignaler, se figuren, till ALU:n så att en addition utförs. Summan 00001001 hamnar i register U och överförs sedan till Y, eftersom X skulle adderas till Y. Statussignalerna vid operationen i ALU:n överförs till ett *Statusregister* (*Flaggor*), vars utsignaler överförs till instruktionsavkodaren, där de används vid villkorliga instruktioner som t.ex villkorliga hopp som JNZ (Jump if Not Zero), ”hoppa om resultatet inte var lika med noll”.

## 4.8 Paritetskrets

### Felupptäckande och felnödande kod

Informationsöverföring kan beskrivas med modellen *sändare-kanal-mottagare* nedan. Man tänker kanske närmast på tekniska sammanhang såsom telekommunikation el.dyl. när man ser modellen, men den kan appliceras på all form av informationsöverföring. Det kan exempelvis vara en person som säger något till en annan person, vilka är sändaren respektive mottagaren och kanalen kan vara luften mellan mun och öra. Det kan också vara överföring av information mellan datorer, längre sträckor via telenätet eller kortare sträckor i lokala datanät. Det kan också vara överföringar av ord inne i själva datorn mellan olika enheter.

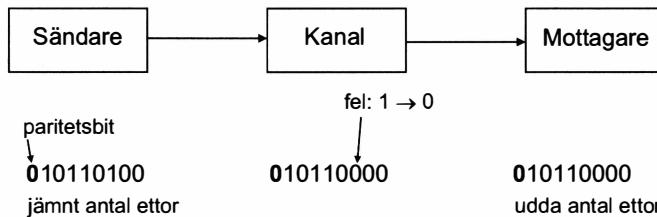


Figur 4.58 Modell för informationsöverföring.

Vid informationsöverföringen kan fel inträffa. En störning i kanalen kan medföra att mottagaren får felaktig information. Det är då naturligtvis önskvärt att mottagaren kan *upptäcka* att fel inträffat. En vanlig felmodell som används vid överföring av binära ord är att antaga att bara två typer av fel kan inträffa, att en nolla felaktigt blir en etta eller att en etta felaktigt blir en nolla. I figuren ovan visas som exempel överföring av en byte där fel inträffar i en bit så att en etta blir en nolla. Om vilken byte som helst kan förväntas från sändaren, så kan mottagaren uppenbart inte upptäcka att ett fel inträffat. För att *felupptäckt* (eng. *error detection*) skall vara möjlig så måste de informationsbärande orden hämtas ur en större mängd ord, så att ett fel alltid resulterar i ett icke informationsbärande ord, som då kan upptäckas.

En enkel metod med vilken fel av typen ovan kan upptäckas är att använda s.k. *paritetskontroll* (eng. *parity check*). Denna metod innebär att de binära orden utökas med en *överflödig, redundant* (eng. *redundant*) bit, *paritetsbit*

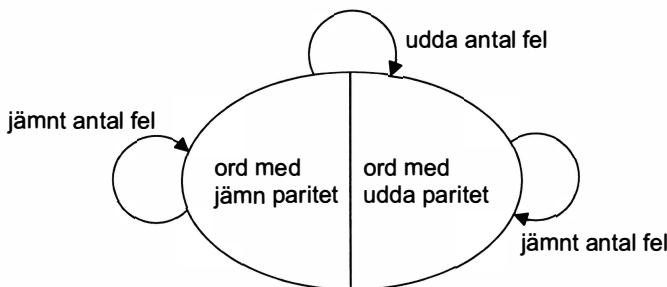
(eng *parity bit*), enligt regeln att antalet ettor i det nya ordet (inklusive paritetsbiten) skall bli jämnt, s.k. *jämn paritet* (eng. *even parity*). Om jämn paritet används i exemplet ovan så blir paritetsbiten en nolla eftersom antalet ettor i det ursprungliga ordet är jämnt. Informationsöverföringen sker då enligt figuren nedan, med paritetsbiten tillfogad längst till vänster.



Figur 4.59 Informationsöverföring med jämn paritet.

Felet i överföringen ovan gör att antalet ettor i ordet blir udda och mottagaren, kan nu om han alltid räknar antalet ettor i de mottagna orden, upptäcka felet. Fel i *en* bit kan alltid upptäckas, medan fel i två bitar inte kan upptäckas. Sannolikheten för två fel är avsevärt mindre än sannolikheten för ett fel och det är normalt tillräckligt att kunna upptäcka enkelfel för att tillförlitligheten i överföringen skall bli god (egentligen kan alla fel av udda antal upptäckas). Jämn paritet har använts ovan, men sändaren och mottagaren kan lika bra komma överens om att alla ord alltid skall ha ett udda antal ettor, benämnt *udda paritet* (eng. *odd parity*).

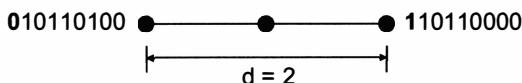
Tillägg av en paritetsbit innebär att totala antalet ord blir dubbelt så stort. I fallet ovan med 8-bitars ord har ordmängden totalt  $2^8 = 256$  ord, medan ordmängden efter tillägg av en extra bit totalt har  $2^9 = 512$  ord. De ursprungliga 256 orden försedda med en paritetsbit för jämn paritet är en delmängd av mängden av samtliga 512 stycken 9-bitars ord. Denna delmängd utgör hälften av orden, den andra hälften utgör orden med udda paritet. När det inträffar ett enkelfel i ett ord i den ena delmängden, så blir det felaktiga ordet ett ord i den andra delmängden och felet således möjligt att upptäckas, se figuren nedan.



Figur 4.60 Felbeteende för en ordmängd med jämn och udda paritet.

Paritetskontroll medger alltså upptäckt av enkelfel, men inte rättningsförmåga för fel. Detta beror på att enkelfel i två olika ord kan resultera i samma felaktiga ord. I exemplet i figur 4.58 ovan resulterade ett enkelfel i ordet **010110100** i det felaktiga ordet **010110000**, vilket också erhålls t.ex. vid sändning av ordet **110110000** och ett enkelfel i biten längst till vänster, paritetsbiten. För att *felrättning* (eng. *error correction*) skall vara möjlig, så måste *distansen* mellan orden vara större. *Distansen*  $d(w_1, w_2)$  mellan två binärord  $w_1$  och  $w_2$  med lika många bitar, definieras som antalet positioner i vilka orden skiljer sig.

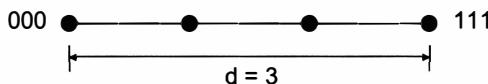
Sålunda gäller för orden ovan **010110100** och **110110000** med paritetsbit, att distansen  $d(010110000, 110110000) = 2$ , se figuren nedan. För hela mängden av ord med jämn paritet (liksom för hela mängden av ord med udda paritet) gäller att distansen mellan två ord är minst lika med 2 (varför?).



Figur 4.61 Distansen  $d$  mellan orden 010110100 och 110110000.

Eftersom distansen mellan två ord i mängden av ord med jämn paritet är minst lika med 2, så ger ett enkelfel i ett sådant ord inte ett ord i denna mängd utan ett ord med udda paritet, se figur 4.59 ovan, och felet således möjligt att upptäcka.

För att ett enkelfel skall kunna *rättas* för en mängd av ord, så måste uppenbart distansen mellan två godtyckliga ord i mängden vara minst 3, ty då ger enkelfel i två olika ord aldrig samma felaktiga ord. Antag för enkelhets skull att vi har de två 1-bitars orden 0 och 1, som skall förses med redundanta positioner så att det blir möjligt att rätta enkelfel. Detta kan enkelt ske genom att ordet 0 utökas med 00 till 000 och ordet 1 utökas med 11 till 111, varvid således distansen  $d(000, 111) = 3$ , se figuren nedan.



Figur 4.62 Distansen  $d$  mellan orden 000 och 111.

För den enkla koden omfattande bara de två orden 000 och 111 kan *enkelfel rättas och dubbelfel upptäckas*. De ursprungliga orden var alltså 0 och 1, som hade distansen 1, och för att kunna rätta enkelfel fick vi avbilda orden 0 och 1 på två ord i en större mängd, i detta fall på orden 000 respektive 111 med distansen 3 i mängden av 3-bitars ord, som innehåller 8 ord.

Möjlighet till felupptäckt och felrättning kostar pengar – vid överföring och behandling av bitar i parallell form kostar det fler kretsar, större bussar etc., och vid överföring och behandling av bitar i serieform kostar det tid, som ju också är pengar. Normalt får man av kostnadsskäl nöja sig med *felupptäckande koder* (eng. *error detecting codes*) och ev. möjlighet till begäran om sändning av ordet en gång till, denna gång kanske med lyckat resultat. Endast i avancerade sammanhang där man inte har möjlighet att upprepa sändningen kan man ha råd med *felrättande koder* (eng. *error correcting codes*).

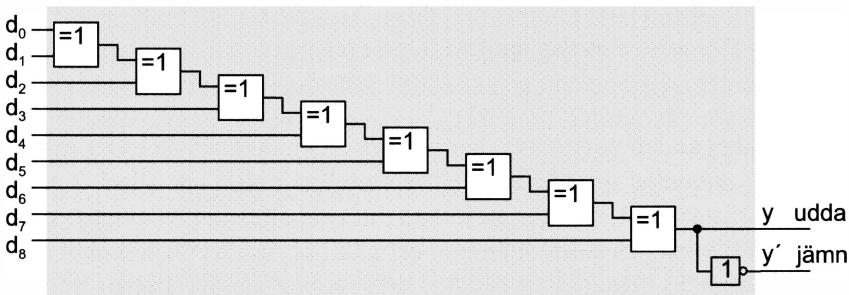
Låt oss nu avsluta sektionen med vad som egentligen är rubriken för sektionen, en *paritetskrets*.

## Paritetskrets

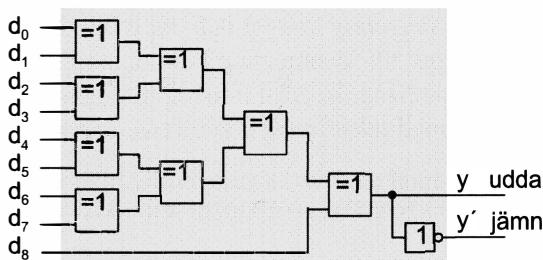
Vid behandlingen av XOR-grinden i kapitel 2 nämdes flera tolkningar av grindens funktion, bl. a. att ”utsignalen är 1 om och endast om ett *udda* antal av insignalerna är 1”. För XOR-uttrycket nedan gäller att ” $y = 1$  om och endast om ett udda antal av variablerna  $x_0, x_1, \dots, x_{n-1}$  är lika med 1”.

$$y = x_0 \oplus x_1 \oplus \dots \oplus x_{n-1}$$

Nedan visas en krets som realiseringen för  $n = 9$  bitar. Kretsen kan *kontrollera pariteten* för ett 9-bitars ord för udda paritet ( $y = 1$ ) eller jämn paritet ( $\bar{y} = 1$ ). Kretsen kan också *generera paritetsbit* för ett 9-bitars ord varvid då funktionen  $y$  blir paritetsbit för jämn paritet och  $\bar{y}$  paritetsbit för udda paritet. Kretsen har djupet åtta XOR-grindar plus en inverterare. En alternativ realisering av samma funktion med djupet fyra XOR-grindar och en inverterare visas nedan.



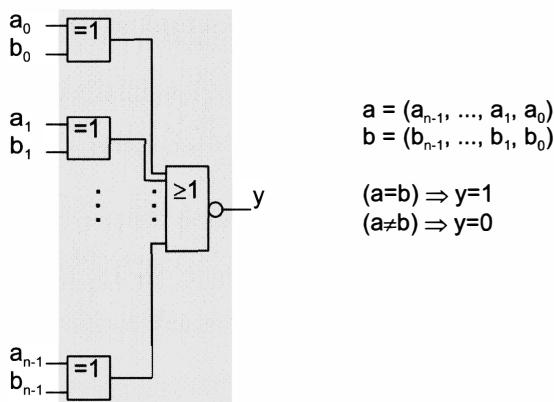
Figur 4.63 Krets för 9-bitars *udda/jämn* paritetskontroll/paritetsgenerering.



Figur 4.64 Krets för 9-bitars *udda/jämn* paritetskontroll/paritetsgenerering – alternativ realisering med färre antal grindnivåer.

## 4.9 Komparator för likhet

Ofta behöver man i digitala system jämföra (eng. *compare*) två binära ord  $a = (a_{n-1}, \dots, a_1, a_0)$  och  $b = (b_{n-1}, \dots, b_1, b_0)$  med avseende på ”likhet” ( $=$ ), dvs överensstämelse i alla bitar, och om  $a$  och  $b$  är binärtal jämföra med avseende på ”mindre än” ( $<$ ). I en mikroprocessor görs detta med operationen subtraktion i ALUn, se sektion 4.7, varvid Z-flaggan indikerar ”likhet” och C-flaggan (ev. tillsammans med OF, beroende på talområdet) indikerar ”mindre än”. Speciella kretsar för jämförelser, *jämförare* (eng. *comparator*) kan också ibland behöva realiseras. En jämförare med avseende på ”likhet” realiseras enkelt enligt figuren nedan, med XOR-grindar som gör bitvis jämförelse och en avslutande NOR-grind, som ger utsignalen 1 om och endast om utsignalen från samtliga XOR-grindar är 0, dvs om bitvis likhet råder i samtliga positioner i orden.



Figur 4.65 Jämförare, komparator, för likhet ( $=$ ).

Jämförare för ”mindre än” ( $<$ ) blir redan för ett fåtal bitar för komplex att realisera som ett två-nivå-nät och måste liksom en adderare ges en iterativ uppbyggnad. Detta är kanske inte speciellt förvånande när vi redan nämnt att jämförelse för ”mindre än” i en mikroprocessor görs med operationen subtraktion.

## 4.10 Övningsuppgifter

### 4.2 Booleska funktioner

- 4.1** Omvandla algebraiskt booleska funktionerna nedan till SP- och PS-normalform. (Ledning: Multiplisera produkterna i SP-formen med lämpliga uttryck av typen  $(a + a')$ , som ju är = 1).

a)  $f(x, y, z) = xy' + yz' + x'z$

b)  $f(w, x, y, z) = xy + x'z + wyz$

c)  $f(x, y, z) = (x + y')(xyz + y'(x + z)) + xyz'(x + x'y)$

- 4.2** Visa att det finns  $2^{2^n-1}$  booleska funktioner av n variabler sådana att

$$f(x_{n-1}, \dots, x_2, x_1, x_0) = f(x_{n-1}', \dots, x_2', x_1', x_0')$$

### 4.3 Förenkling och realisering av booleska funktioner i grindnät

- 4.3** Bestäm med Karnaughdiagram samtliga minimala SP-former till funktionerna nedan samt även till funktionernas invers.

a)  $f(x_3, x_2, x_1, x_0) = \Sigma(0, 2, 4, 8, 10, 12)$

b)  $f(x_3, x_2, x_1, x_0) = \Pi(0, 1, 4, 5, 10, 11, 14, 15)$

c)  $f(x_3, x_2, x_1, x_0) = \Sigma(0, 2, 3, 4, 6, 7, 8, 9, 10, 12, 13, 14)$

d)  $f(x_3, x_2, x_1, x_0) = \Sigma(1, 4, 6, 7, 13)$

e)  $f(x_3, x_2, x_1, x_0) = \Sigma(1, 3, 7, 13, 15)$

f)  $f(x_4, x_3, x_2, x_1, x_0) = \Sigma(9, 11, 12, 13, 14, 15, 16, 18, 24, 25, 26, 27)$

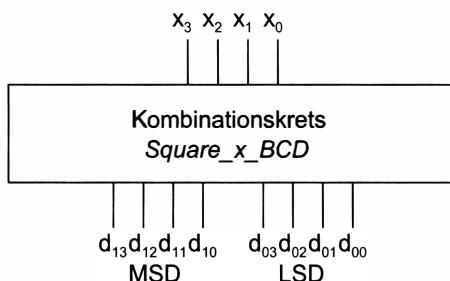
g)  $f(x_4, x_3, x_2, x_1, x_0) = \Sigma(8, 9, 13, 14, 15, 16, 18, 24, 25, 26, 27)$

h)  $f(x_5, x_4, x_3, x_2, x_1, x_0) = \Sigma(8, 9, 10, 11, 17, 19, 21, 23, 25, 27, 41, 43, 44, 45, 46, 47, 56, 57, 58, 59)$

- 4.4** Bestäm med Karnaughdiagram minimala SP-former till de ofullständigt specificerade funktionerna nedan samt även till funktionernas invers.

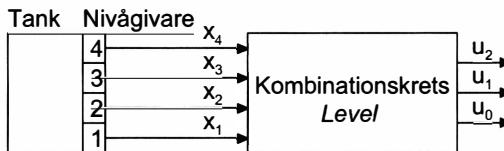
a)  $f(x_3, x_2, x_1, x_0) = \Sigma(3, 5, 7, 11) + d(6, 15)$

- 4.4** b)  $f(x_3, x_2, x_1, x_0) = \Sigma(1, 4, 5) + d(2, 3, 6, 7, 8, 9, 12, 13)$   
 c)  $f(x_4, x_3, x_2, x_1, x_0) = \Sigma(3, 5, 7, 11, 12, 29, 31) + d(1, 2, 6, 10, 28)$   
 d)  $f(x_3, x_2, x_1, x_0) = \Sigma(1, 3, 5, 8, 9, 11, 15) + d(2, 13)$   
 e)  $f(x_3, x_2, x_1, x_0) = \Sigma(4, 5, 7, 12, 14, 15) + d(3, 8, 10)$   
 f)  $f(x_4, x_3, x_2, x_1, x_0) = \Pi(6, 7, 8, 9, 10, 13, 14, 16, 17, 22, 24, 25, 29)$   
 $\quad \quad \quad + d(0, 12, 15, 30)$
- 4.5**  $x = (x_2, x_1, x_0)$  är binära positiva heltal i talområdet 0 - 7. Funktionen  $u = x^2$ , där  $u = (u_5, u_4, u_3, u_2, u_1, u_0)$ , skall realiseras i en kombinationskrets. Insignaleraternas inverser är tillgängliga. Exempel: Om  $x = 101$ , dvs talet 5, så skall kretsen ge  $u = 011001$ , dvs talet 25, eftersom  $5^2 = 25$ . Gör en funktionstabell för kombinationskretsen och bestäm med Karnaughdiagram minimala SP-former till  $u$ -funktionerna och deras invers.
- 4.6**  $x = (x_3, x_2, x_1, x_0)$  är decimala siffror 0–9 i BCD-kod. Realisera i en kombinationskrets funktionen  $z = 3 \cdot x$  där  $z = (z_4, z_3, z_2, z_1, z_0)$  är heltal i vanlig binärkod. Exempel: Om  $x = 0111$ , dvs siffran 7, så skall kretsen ge  $z = 10101$ , dvs talet 21, eftersom  $z = 3 \cdot 7 = 21$ . Realisera  $z$ -funktionerna i minimala NAND-NAND-nät. Insignaleraternas inverser är tillgängliga.
- 4.7**  $x = (x_3, x_2, x_1, x_0)$  är decimala siffror 0–9 i BCD-kod. En kombinationskrets *Square\_x\_BCD* med fyra insignaler  $x_3, x_2, x_1, x_0$  samt åtta utsignaler  $d_{13}, d_{12}, d_{11}, d_{10}, d_{03}, d_{02}, d_{01}, d_{00}$  som skall bilda kvadraten av den decimala siffran, skall konstrueras.

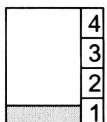


Kvadraten skall ges i 2-siffrors BCD-kod. Exempel: Om  $x = 0110$ , dvs siffran 6, så skall utsignalerna bli  $(d_{13}, d_{12}, d_{11}, d_{10}) = 0011$ , dvs siffran 3, och  $(d_{03}, d_{02}, d_{01}, d_{00}) = 0110$ , dvs siffran 6,  $6^2 = 36$ . Realisera kombinationskretsen i ett minimalt NAND-NAND-nät. Insignaleraternas inverser är tillgängliga.

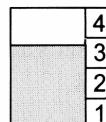
- 4.8**  $x_4, x_3, x_2, x_1$  är utsignaler från en nivågivare till en tank och insignalen till en kombinationskrets *Level*.



När ingen vätska finns i tanken är  $x_4 = x_3 = x_2 = x_1 = 0$ . När vätska fylls på och nivån är i område 1 blir  $x_1 = 1$  och när nivån är i område 2 blir  $x_2 = x_1 = 1$ , osv. Kombinationskretsen skall ge binärtalatet  $u_2, u_1, u_0$  för vätskans nivå. Exempel:



$$\Rightarrow u_2 u_1 u_0 = 001$$



$$\Rightarrow u_2 u_1 u_0 = 011$$

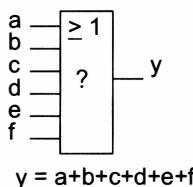
Realisera kombinationskretsen i ett minmalt NAND-NAND-nät. Insignalerna inverser är tillgängliga.

- 4.9** En kombinationskrets *Maxnumber* med fem insignalen  $x_1, x_2, \dots, x_5$  och tre utsignaler  $u_0, u_1$  och  $u_2$  skall realiseras i ett minimalt OCH-ELLER-nät. Kretsen skall ge binärtalatet  $u = (u_2, u_1, u_0)$  för högsta numret på den insignal som har värdet 0. Om ingen insignal är 0, skall kretsen ge  $u = 000$ . Exempel: Om  $x = (x_5, x_4, x_3, x_2, x_1) = 10100$ , så skall utsignalen bli  $u = 100$ , dvs decimaltalet 4, eftersom högsta numret på insignalen som har värdet 0 är fyra,  $x_4 = 0$ . Insignalernas inverser är tillgängliga.
- 4.10**  $x = (x_3, x_2, x_1, x_0)$  är decimala siffror 0 - 9 i BCD-kod. Realisera i en kombinationskrets funktionen  $z = 5 \cdot x$ , där  $z = (z_5, z_4, z_3, z_2, z_1, z_0)$  är heltalet i vanlig binärkod. Exempel: Om  $x = 0111$ , dvs siffran 7, så skall kretsen ge  $z = 100011$ , dvs talet 35, eftersom  $z = 5 \cdot 7 = 35$ . – Realisera kombinationskretsen som ett minmalt OCH-ELLER-nät, under beaktande av grinddelning. Insignalernas inverser är tillgängliga.
- 4.11** Booleska funktionen  $f(x, y, z) = z(xy)' + z'xy$  skall realiseras med NAND-grindar i tre nivåer. Insignaler till grindnätet skall vara  $x, y$  och  $z$ . Det är lätt att konstruera nätet med fem NAND-grindar. Visa att det går att konstruera nätet med endast fyra NAND-grindar.

**4.12** Visa att varje boolesk funktion av tre variabler kan realiseras i en SP-form med maximalt fyra produkttermer av tre variabler (Ledning: Analysera Karnaughdiagrammet!).

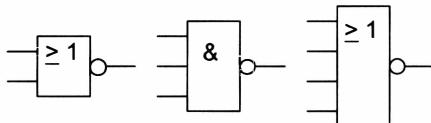
**4.13** Realisera en 6-ingångars ELLER-grind med en 2-ingångars NOR-grind, en 3-ingångars NAND-grind och en 4-ingångars NOR-grind. Visa med logiska uttryck att den föreslagna kopplingen realiseras den önskade ELLER-operationen.

Grind som skall  
realiseras:



$$y = a+b+c+d+e+f$$

Tillgängliga grindar:



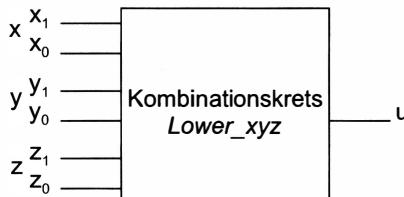
**4.14 a)** Realisera med två XOR-grindar och en NAND-grind, booleska funktionen

$$f(w, x, y, z) = \Pi(5, 6, 9, 10)$$

**b)** Realisera med två XOR-grindar och en 2-ingångars NOR-grind, booleska funktionen nedan. Grindnätet skall ha två nivåer. Till grindarna får anslutas variablerna och konstanterna 0 och 1 (variablernas inverser är ej tillgängliga).

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

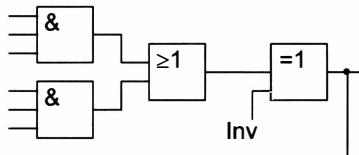
**4.15** En kombinationskrets *Lower\_xy* som jämför tre tal  $x = (x_1, x_0)$ ,  $y = (y_1, y_0)$  och  $z = (z_1, z_0)$  med avseende på relationen "mindre än" ( $<$ ) skall konstrueras. För utsignalen  $u$  skall gälla att  $u = 1$  om och endast om  $x < y < z$ . Bestäm en minimal SP-form till utsignalen  $u$ .



**4.16 a) Realisera booleska funktionen**

$$f = bc'd' + a'd' + ac' + a'c$$

i ett grindnät enligt nedan som kan tänkas ingå i en programmerbar logisk krets (PLD). Till en ingång på OCH-grindarna kan anslutas en variabel eller dess invers eller konstanten 0 eller 1. Till ingången Inv på XOR-grinden kan bara anslutas konstanten 0 eller 1.

**b) Realisera booleska funktionen**

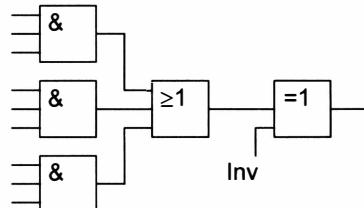
$$f = a'b'c' + abcde$$

i en PLD som innehåller flera grindnät enligt uppgift a), där alltså utsignalen kan anslutas som insignal till grindnäten. Använd så få nät som möjligt.

**4.17** En jämförare, komparator, för två binära 2-bitars tal  $x = (x_1, x_0)$  och  $y = (y_1, y_0)$  skall realiseras i en PLD. Komparatorn skall ha två utsignaler  $u_1$  och  $u_0$  för vilka skall gälla att

$$\begin{aligned} u_1u_0 &= 11 \quad \text{om} \quad x = y \\ u_1u_0 &= 01 \quad \text{om} \quad x < y \\ u_1u_0 &= 10 \quad \text{om} \quad x > y \end{aligned}$$

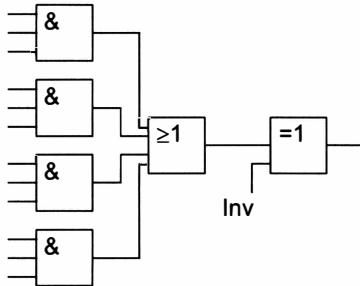
I PLDn finns två OCH-ELLER-nät enligt nedan. Till en ingång på OCH-grindarna kan anslutas variabeln eller variabelns invers eller konstanten 0 eller 1. Till ingången Inv på XOR-grinden kan bara anslutas 0 eller 1.



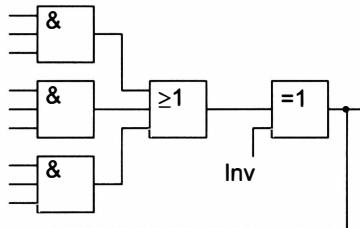
**4.18 a) Realisera funktionen**

$$f = 1 \oplus x_0 \oplus x_0 x_1 \oplus x_2 x_3 \oplus x_3$$

i ett grindnät enligt nedan i en PLD. Till en ingång på OCH-grindarna kan anslutas en variabel eller dess invers eller konstanten 0 eller 1. Till ingången Inv på XOR-grinden kan bara anslutas konstanten 0 eller 1.



**b)** Realisera samma funktion som i uppgift a), i en PLD i vilken finns flera grindnät enligt nedan. Jämfört med grindnätet ovan har alltså dessa grindnät bara tre OCH-grindar, men grindnätens utsignal kan återkopplas och anslutas som insignal till grindnäten.

**4.19 Realisera booleska funktionen**

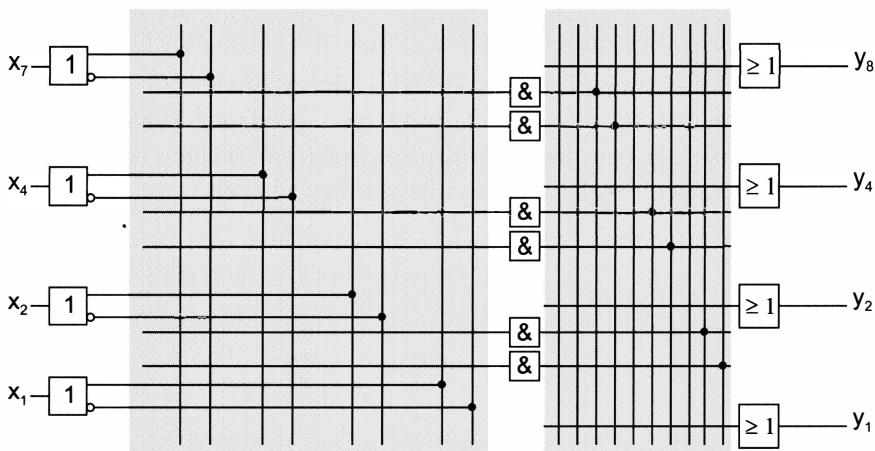
$$f(x_4, x_3, x_2, x_1, x_0) = (((x_4' x_2)' x_1)' x_3 + x_4 x_2 (x_1' x_0)' + x_4 x_3 x_2' x_1' x_0)''$$

i ett grindnät enligt uppgift 4.18 a.

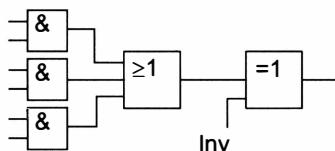
- 4.20** De decimala siffrorna 0 till 9 är kodade i 7421-kod (viktad kod med positionsvikterna 7, 4, 2 och 1). En kombinationskrets som översätter från denna kod till BCD-kod skall konstrueras.



Realisera kombinationskretsen i ett minimalt OCH-ELLER-nät under beaktande av grinddelning. Insignalernas inverser är tillgängliga. Visa realiseringen i PLDn nedan genom markering av de programmerade förbindelserna i OCH- och ELLER-matrisen.



- 4.21**  $x = (x_3, x_2, x_1, x_0)$  är decimala siffror 0–9 i BCD-kod. En kombinationskrets med ingångarna  $x_3, x_2, x_1$  och  $x_0$  och utgångarna  $y_3, y_2, y_1$  och  $y_0$  skall konstrueras. Kombinationskretsen skall bilda *9-komplementet* till siffran  $x$ , definierat som  $y = (y_3, y_2, y_1, y_0) = 9 - x$ . Realisera kretsen i en PLD som innehåller fyra grindnät enligt nedan.

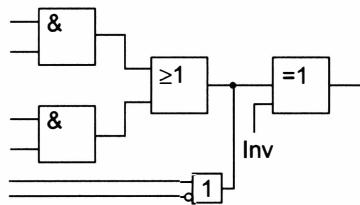


**4.22 Realisera booleska funktionerna**

$$f_1(x_3, x_2, x_1, x_0) = \Sigma(0, 1, 2, 3, 8, 9, 10, 11, 12, 14)$$

$$f_2(x_3, x_2, x_1, x_0) = \Sigma(1, 3, 4, 5, 6, 7, 13, 15)$$

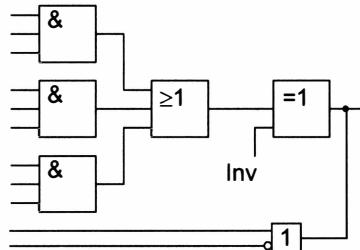
i två grindnät enligt nedan som ingår i en PLD. Till en ingång på OCH-grindarna kan anslutas en variabel eller dess invers eller konstanten 0 eller 1. Till ingången Inv på XOR-grinden kan bara anslutas konstanten 0 eller 1. Grindnätnets utsignal och dess invers kan återkopplas och anslutas som insignal till OCH-grindarna.

**4.23 Realisera booleska funktionerna**

$$f_1(x_4, x_3, x_2, x_1, x_0) = \Sigma(4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31)$$

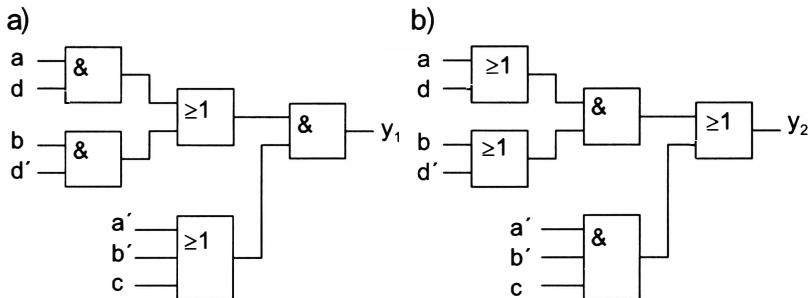
$$f_2(x_4, x_3, x_2, x_1, x_0) = \Sigma(0, 1, 2, 3, 5, 8, 10, 12, 14, 16, 17, 18, 19, 21, 22, 23, 30, 31)$$

i två grindnät enligt nedan som ingår i en PLD. Till en ingång på OCH-grindarna kan anslutas en variabel eller dess invers eller konstanten 0 eller 1. Till ingången Inv på XOR-grinden kan bara anslutas konstanten 0 eller 1. Grindnätnets utsignal och dess invers kan återkopplas och anslutas som insignal till OCH-grindarna.

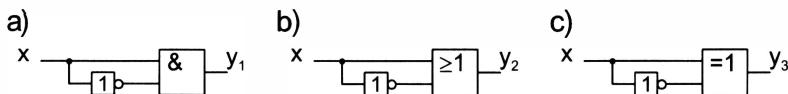
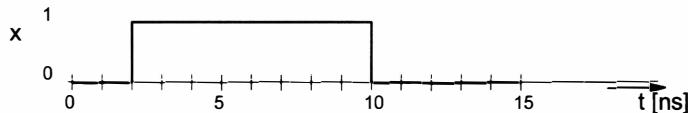


## 4.5 Kombinationskretsar i tidsplanet

**4.24** Bestäm statisk 0- och 1-hasard för grindnäten nedan. Konstruera om näten som hasardfria två-nivå-nät av typ OCH-ELLER.



**4.25** Rita tidsdiagram för utsignalerna till grindnäten nedan för den angivna insignalen. Födröjningen för inverteraren är 1 ns och för de andra grindarna 2 ns.



## 4.6 Adderare

**4.26** Skriv summasignalen s<sub>i</sub> hos heladderaren FA i figur 4.45 med enbart operationer XOR.

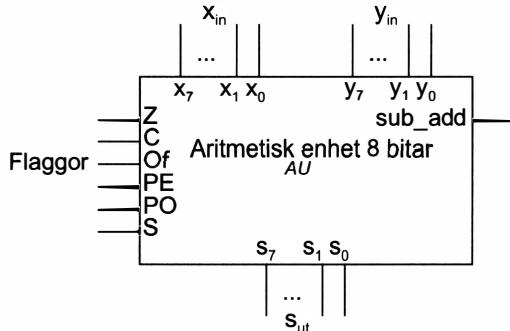
**4.27** a) I figur 4.47 visas en adderare för 8 bitar försedd med en carryaccelerator för 4 bitar. Rita om schemat så att adderaren i stället blir försedd med tre carryacceleratorer för 2 bitar.

b) Realisera carryacceleratorn för två bitar med insignalerna  $x = (x_{i+1}, x_i)$ ,  $y = (y_{i+1}, y_i)$  och minnessiffran  $c_i$  samt utsignalen  $c_{i+2}$  i ett minmalt NAND-NAND-nät.

c) Utan carryacceleratorer blir maximala antalet nivåer i adderaren 17. Beräkna maximala antalet nivåer i adderaren försedd med de tre carryaccelatorena ovan.

## 4.7 Aritmetisk logisk enhet (ALU)

**4.28** En aritmetisk enhet för 8 bitar är uppbyggd enligt principen i figur 4.50 och utför subtraktion som addition av 2-komplementet. Den är försedd med utsignaler för flaggor Z (Zero), C (Carry), Of (Overflow), PE (Parity Even), PO (Parity Odd) och S (Sign).



Nedan visas exempel på en addition och en subtraktion.

### Addition

$$\begin{array}{r}
 \text{Dec} \quad \text{Hex} \quad \text{Bin} \\
 \hline
 & & 0 \quad 1 \\
 x_{in} & 49 & 31 \\
 + y_{in} & 65 & 41 \\
 \hline
 s_{ut} & 114 & 72 \\
 \end{array}
 \quad
 \begin{array}{l}
 00110001 \\
 +01000001 \\
 \hline
 01110010
 \end{array}$$

Z C Of PE PO S  
0 0 0 1 0 0

### Subtraktion

$$\begin{array}{r}
 \text{Dec} \quad \text{Hex} \quad \text{Bin} \\
 \hline
 & & 0 \quad 111111 \\
 x_{in} & 49 & 31 \\
 - y_{in} & 65 & 41 \\
 \hline
 s_{ut} & -16 & F0 \\
 \end{array}
 \quad
 \begin{array}{l}
 00110001 \\
 10111110 \\
 +00000001 \\
 \hline
 11110000
 \end{array}$$

Z C Of PE PO S  
0 0 0 1 0 1

Utför följande additioner och subtraktioner enligt principen ovan och bestäm  $s_{ut}$  och flaggor.

### Addition

a)  $-49 + -65$       b)  $-49 + 65$       c)  $49 + -65$

d)  $49 + -49$       e)  $49 + 85$       f)  $-49 + -85$

### Subtraktion

g)  $-49 - -65$       h)  $-49 - 65$       i)  $49 - -65$

j)  $49 - 49$       k)  $49 - -85$       l)  $-49 - 85$

- 4.29 a)** I ett 8-bitars ord skall bitarna 0 och 6 nollställas i en Logisk Enhet (LU), utan att övriga bitar påverkas. Ange logisk operation och 8-bitars operand för att utföra nollställningen.
- b)** I ett 8-bitars ord skall bitarna 1, 3 och 7 ettställas i en Logisk Enhet (LU), utan att övriga bitar påverkas. Ange logisk operation och 8-bitars operand för att utföra ettställningen.
- c)** I ett 8-bitars ord skall bitarna 1, 2, 4, 5 och 7 inverteras i en Logisk Enhet (LU), utan att övriga bitar påverkas. Ange logisk operation och 8-bitars operand för att utföra inverteringen.
- 4.30**  $x = (x_4, x_3, x_2, x_1, x_0)$  är binära heltal i talområdet  $-16$  till  $+15$  med de negativa talen representerade som 2-komplementet till de positiva talen.
- a)** En kombinationskrets med insignalerna  $x_4, x_3, x_2, x_1, x_0$  och utsignalerna  $y_4, y_3, y_2, y_1, y_0$  som bildar beloppet av talet  $x$ , dvs.  $y = |x|$  skall konstrueras. Bestäm minimala SP-former till utsignalerna  $y$ .
- b)** Talen i talområdet  $-16$  till  $+15$  kan enkelt omvandlas till tal i talområdet 0 till  $+31$ . Hur?
- c)** Talen i talområdet 0 till  $+31$  kan enkelt omvandlas till tal i talområdet  $-16$  till  $+15$ . Hur?

# 5 Sekvenskretsar

Sekvenskretsar är en klass av kretsar med minnesfunktion. För dessa kretsar är tiden en väsentlig parameter – samma insignalvärde kan vid olika tidpunkter ge olika utsignalvärdet. I kapitel 2 studerade vi några enkla sekvenskretsar, såväl generella sekvenskretsar som speciella sekvenskretsar såsom räknare och register. Vi *analyserade* då beteendet för givna sekvenskretsar och diskuterade bl.a. tillståndsbegreppet, sättet att beskriva beteendet med tillståndsdiagram och tillståndstabell, klocksignal och hur tillståndsregistret kan realiseras med D-vippor. Vidare berördes de två sekvenskretsmodellerna av typ Moore och Mealy. I detta kapitel skall vi nu studera *syntes* av sekvenskretsar, dvs realisering av sekvenskretsar utgående från en given specifikation av beteendet och se hur syntesen sker i väldefinierade steg från specifikation till färdig krets. Huvudsakligen kommer vi att studera synkrona sekvenskretsar, men även att kort beröra asynkrona sekvenskretsar. Vi kommer att återvända till D-vippan och diskutera tidsaspekter och maximal klockfrekvens för en sekvenskrets. En annan vippa, T-vippan, som är fördelaktig att använda vid realisering av räknare, kommer att studeras. Latchar, enklare minneskretsar jämfört med vippor, kommer också att studeras. – Sekvenskretsar benämnes ibland på engelska *Finite State Machines (FSM)*, ”tillståndsmaskiner”.

## 5.1 Generella sekvenskretsar

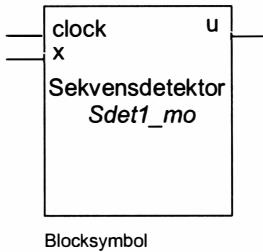
### Realisering av en sekvenskrets typ Moore

I kapitel 2 *analyserade* vi beteendet för en given sekvenskrets av typ Moore, en sekvensdetektor benämnd *Sdet1\_mo*, vars kretsschema visades i figur 2.55. Det önskade beteendet som legat till grund för konstruktion av

sekvenskresten, beskrevs. Tillståndsdiagram, tillståndstabell, tidsdiagram för en insekvens visades. Vi skall nu återvända till denna sekvenskrets och göra *syntesen* av sekvenskretsen och börjar då med den tidigare givna beskrivningen av beteendet.

### Specifikation av beteendet för sekvensdetektorn *Sdet1\_mo*

En sekvensdetektor *Sdet1\_mo*, med en insignal *x*, en utsignal *u* samt en klocksignal *clock*, skall konstrueras. Sekvensdetektorn skall i en godtyckligt lång insekvens i *x*, detektera förekomster av delsekvenser bestående av minst tre på varandra följande ettor. Utsignalen *u* skall bli 1 när delsekvensen detekteras, annars skall *u* vara 0. Delsekvenserna får vara överlappande. Sekvenskretsen skall vara av typ Moore. Sekvenskretsen förutsätts starta i ett speciellt starttillstånd då inmatning av en ny insekvens börjar (hur övergång till starttillståndet skall ske, behöver inte beaktas här, det behandlas längre fram).



Blocksymbol

Figur 5.1 Blocksymbol för sekvensdetektorn, *Sdet1\_mo*.

### Tillståndsdiagram

Realiseringen av en sekvenskrets börjar med det svåraste steget i hela processen, att formellt beskriva det specificerade beteendet med tillstånd. Det finns ingen väldefinierad metod för detta, man får huvudsakligen lita till intuition och erfarenhet. Beskrivningen kan ske på olika sätt, grafiskt i ett tillståndsdiagram, i tabellform i en tillståndstabell, eller i en textfil i t.ex. hårdvarubeskrivande språket VHDL. Vilken beskrivning som är lämpligast i ett aktuellt fall bestäms närmast av regelbundenheten i sekvenskretsens

beteende. Om sekvenskretsen som skall realiseras är en räknare eller ett skiftregister så är beteendet mycket regelbundet och antalet tillstånd ofta mycket stort och man beskriver då beteendet i en textfil i VHDL. Om i stället sekvenskretsen har ett oregelbundet beteende och ett mindre antal tillstånd så kan det vara lämpligt att börja med att grafiskt beskriva beteendet i ett tillståndsdiagram som sedan beskrivs i en textfil i VHDL.

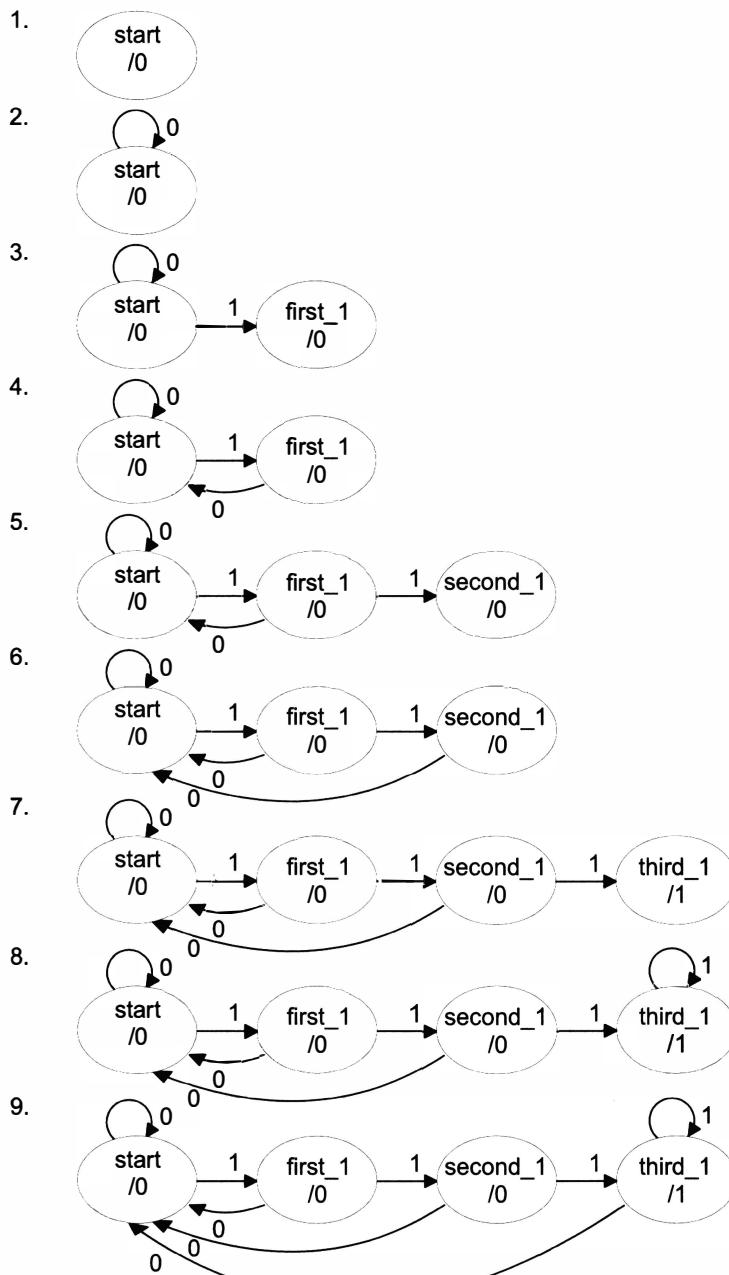
Tillstånden i tillståndsdiagrammet representeras med cirklar, ellipser el.dyl. och tillståndsovergångarna med pilar, som vi redan sett i kapitel 2. Man vet från början inte hur många tillstånd som krävs för att beskriva det aktuella beteendet, utan man får börja med ett tillstånd och bygga på med fler tillstånd efter hand. Det kan hända att man råkar använda fler tillstånd än vad som egentligen behövs. Det har ingen större betydelse, kan innebära att det krävs någon extra D-vippa i tillståndsregistret, men antalet D-vippor är normalt inte något problem. Orsaken till att man har fått fler tillstånd än vad som krävs är att det då finns tillstånd som är s.k. ekvivalenta och som kan slås samman till ett tillstånd.

Konstruktionen av tillståndsdiagrammet börjar med ett starttillstånd, som lämpligen benämnes *start*. Principen är sedan att när sekvenskretsen behöver ”komma ihåg” något, så går man till ett nytt tillstånd. I ett aktuellt tillstånd för en viss insignal har man att ta ställning till om man skall

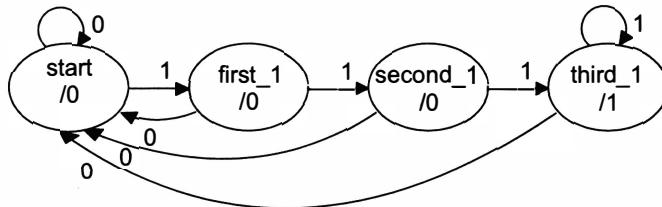
- stanna kvar i tillståndet
- gå till ett annat redan befintligt tillstånd
- skapa ett nytt tillstånd och gå till detta

I figur 5.2 visas steg för steg, som kommenteras nedan, konstruktionen av tillståndsgrafen.

1. Starttillstånd *start*;  $u = 0$ .
2. Inledande 0:or behöver ej kommas ihåg, stanna kvar;  $u = 0$ .
3. En 1:a är inledning på en delsekvens och måste kommas ihåg, skapa ett nytt tillstånd *first\_1* och gå till detta;  $u = 0$ .
4. En 0:a efter inledande 1:an innebär avbruten delsekvens, gå tillbaks till starttillståndet;  $u = 0$ .
5. En 1:a efter inledande 1:an innebär andra 1:an i delsekvensen som måste kommas ihåg, skapa ett nytt tillstånd *second\_1* och gå till detta;  $u = 0$ .
6. En 0:a efter två 1:or innebär avbruten delsekvens, gå tillbaks till starttillståndet;  $u = 0$ .
7. En 1:a efter två 1:or innebär korrekt delsekvens som måste kommas ihåg, skapa ett nytt tillstånd *third\_1* och gå till detta;  $u = 1$ .
8. En 1:a efter den korrekta delsekvensen innebär ytterligare en korrekt delsekvens (överlappande);  $u = 1$ .
9. En 0:a innebär avbruten delsekvens, gå tillbaks till starttillståndet;  $u = 0$ .



Figur 5.2 Konstruktionen av tillståndsdiagrammet för Sdet1\_mo.



Figur 5.3 Slutliga tillståndsdiagrammet för sekvensdetektorn *Sdet1\_mo*.

Tillstånden bör döpas så att namnen antyder vad tillstånden innebär, såsom gjorts i tillståndsdiagrammet ovan, där namnen anger hur långt man kommit i den korrekta delsekvensen.

### Tillståndstabell

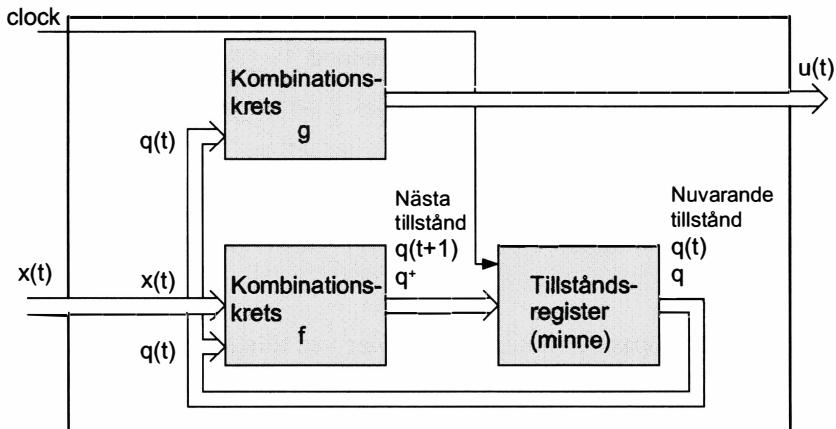
Tillståndstabellen innehåller ingen ny information jämfört med tillståndsdiagrammet.

Nuvarande tillstånd	Nästa tillstånd		Utsignal u	
	Insignal x			
	0	1		
start	start	first_1	0	
first_1	start	second_1	0	
second_1	start	third_1	0	
third_1	start	third_1	1	

Figur 5.4 Tillståndstabell för sekvensdetektorn *Sdet1\_mo*.

Sekvenskretsen skall enligt specifikationen realiseras som typ Moore. Nedan visas den generella modellen för en sekvenskrets av typ Moore.

## Modell för en sekvenskrets av typ Moore



Figur 5.5 Modell för en sekvenskrets av typ Moore.

Kretsschemat för sekvensdetektorn *Sdet1\_mo* som visades i figur 2.55 i kapitel 2 har ritats så att det överensstämmer med modellen ovan. Tillståndsregistret utgörs där av D-vipporna och kombinationskretsarna  $f$  och  $g$  av grindnäten.

För Moore-modellen ovan gäller

$$q(t+1) = f(q(t), x(t))$$

$$u(t) = g(q(t))$$

Ofta utelämnas av bekvämlighet parametern  $t$  i uttrycken ovan. Nästa tillstånd betecknar vi med  $q^+$ , som vi redan sett i kapitel 2. Uttrycken blir då

$$q^+ = f(q, x)$$

$$u = g(q)$$

Karakteristiskt för Moore-modellen är att utsignalen  $u$  endast beror av nuvarande tillstånd  $q$  och ej av insignalen  $x$ .

Vi fortsätter nu realiseringen av sekvensdetektorn *Sdet1\_mo*.

## Tillståndskodning – ”binary”

Tillståndsregistret som håller nuvarande tillstånd, realiseras normalt med D-vippor. Tillstånden i tillståndsdiagrammet/tillståndstabellen måste därför kodas som binära ord för att kunna lagras i tillståndsregistret. Tillstånden kan kodas på många olika sätt, i princip hur många som helst. Komplexiteten hos kombinationskretsarna f och g beror av kodningen. Det finns ingen metod som anger optimal kod. Vi skall i det följande visa exempel på några olika typer av *tillståndskodningar* (eng. *state assignment*), som brukar erbjudas i syntesverktyg. Låt oss börja med, som rubriken antyder, med ”vanlig” binärkod, här benämnd ”binary”.

Sekvenskretsen *Sdet1\_mo* har fyra tillstånd. Det krävs minst två bitar för att koda tillstånden, ty  $2 = 2^1 < 4 < 2^2 = 4$ . Låt oss välja minimalt antal bitar, dvs två bitar. Tillståndet q betecknar vi i fortsättningen  $q = (q_n, q_{n-1}, \dots, q_0)$ , sålunda index 0 för LSB.

Tillståndskodning ”binary” innebär kodning av den definierade uppräknade tillståndsmängden med binärtal i ordning med början på binärtalat 0. Det krävs sålunda en definierad *uppräknebar* (eng. *enumerated*) tillståndsmängd för att kodningen skall vara relevant. En sådan tillståndsmängd definierar man t.ex. i VHDL-beskrivningen. För sekvenskretsen *Sdet1\_mo* är det naturligt att defiera tillståndsmängden som {start, first\_1, second\_1, third\_1}. Tillståndskodningen ”binär” med två bitar blir då

tillstånd	kodning ”binary”
	$q_1 q_0$
start	0 0
first_1	0 1
second_1	1 0
third_1	1 1

Nuvarande tillstånd $q_1 q_0$	Nästa tillstånd $q_1 q_0^+$		Utsignal u	
	Insignal x			
	0	1		
00	00	01	0	
01	00	10	0	
10	00	11	0	
11	00	11	1	

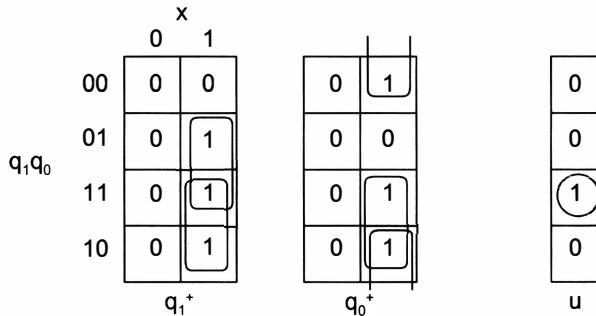
Figur 5.6 Tillståndstabell med tillståndskodning ”binary” för *Sdet1\_mo*.

Vi fortsätter realiseringen med denna tillståndskodning och återkommer sedan till några andra typer av kodningar. Nästa steg i realiseringssprocessen blir att bestämma booleska uttryck för booleska funktionerna i kombinationskretsarna f och g i sekvenskretsmodellen i figur 5.5.

### Booleska uttryck för nästa tillstånd $q^+$ och utsignal u

Booleska funktioner för nästa tillstånd  $q_1^+$  och  $q_0^+$  samt utsignal u är specificerade i tillståndstabellen i figur 5.6 ovan. Vi använder på vanligt sätt Karnaughdiagram för att bestämma minimala booleska uttryck på SP-form.

*Karnaughdiagram*



Figur 5.7 Karnaughdiagram för nästa tillstånd  $q^+$  och utsignal u.

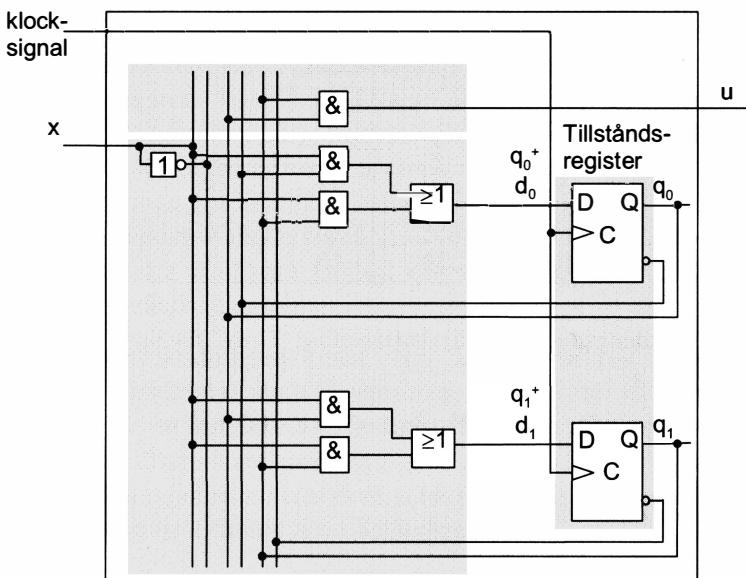
### Booleska uttryck

$$q_1^+ = q_0x + q_1x$$

$$q_0^+ = q_0'x + q_1x$$

$$u = q_1q_0$$

## Kretsschema



Figur 5.8 Kretsschema för Sdet1\_mo.

I kretsschemat ovan identifierar vi i de skuggade områdena tillståndsregistret samt kombinationskretsarna som beräknar nästa tillstånd och utsignal. I kapitel 2 visades tidsdiagram för en insekvens.

Låt oss nu gå tillbaka i realiseringssprocessen och studera några andra tillståndskodningar.

### Tillståndskodning – ”Gray”

tillstånd	kodning ”Gray”
	$q_1q_0$
start	0 0
first_1	0 1
second_1	1 1
third_1	1 0

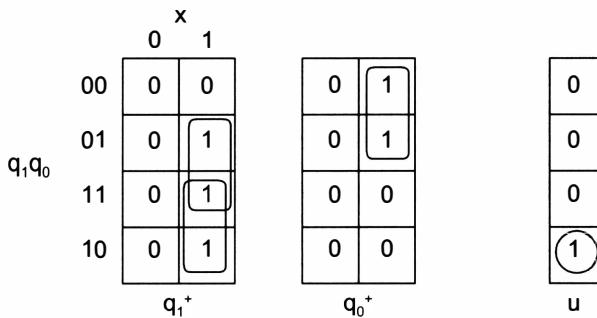
Tillståndskodningen ”Gray” innebär kodning av den uppräknebara tillståndsmängden i Gray-kod.

Nuvarande tillstånd $q_1 q_0$	Nästa tillstånd $q_1^+ q_0^+$		Utsignal u	
	Insignal x			
	0	1		
00	00	01	0	
01	00	11	0	
11	00	10	0	
10	00	10	1	

Figur 5.9 Tillståndstabell med tillståndskodning "Gray" för Sdet1\_mo.

**Booleska uttryck för nästa tillstånd  $q^+$  och utsignal u**

*Karnaughdiagram*

Figur 5.10 Karnaughdiagram för nästa tillstånd  $q^+$  och utsignal u.

**Booleska uttryck**

$$q_1^+ = q_0 x + q_1 x$$

$$q_0^+ = q_1' x$$

$$u = q_1 q_0'$$

Jämför vi dessa booleska uttryck med motsvarande för tillståndskodningen "binär" så kan vi konstatera att  $q_1^+$  har samma komplexitet, två produktter-

mer med vardera två variabler. Även  $u$  har samma komplexitet, en produktterm med två variabler. Däremot är  $q_0^+$  något enklare, bara en produktterm. Generellt behöver inte tillståndskodningen "Gray" ge minst komplexitet, utan det är strukturen hos tillståndsdiagrammet som bestämmer vilken av tillståndskodningarna "binär" och "Gray" som ger minst komplexitet, för ett annat tillståndsdiagram så ger kanske tillståndskodningen "binär" minst komplexitet. Orsaken till att man använder tillståndskodningen "Gray" är normalt inte att man vill få mindre komplexitet, utan i stället att man vill att endast en tillståndsvariabel i taget skall ändra sig vid tillståndsvändningarna. För att detta skall vara möjligt krävs naturligtvis också en speciell struktur hos tillståndsdiagrammet.

För både tillståndskodningen "binär" och "Gray" ovan har vi använt minimalt antal tillståndsvariabler, två variabler. Sålunda har tillståndsregistret samma komplexitet för de två tillståndskodningarna, det innehåller två D-vippor. I nästa tillståndskodning skall vi se hur man kan minska komplexiteten hos kombinationskretsarna som genererar nästa tillstånd och utsignal, genom att öka komplexiteten hos tillståndsregistret.

### Tillståndskodning – "one-hot"

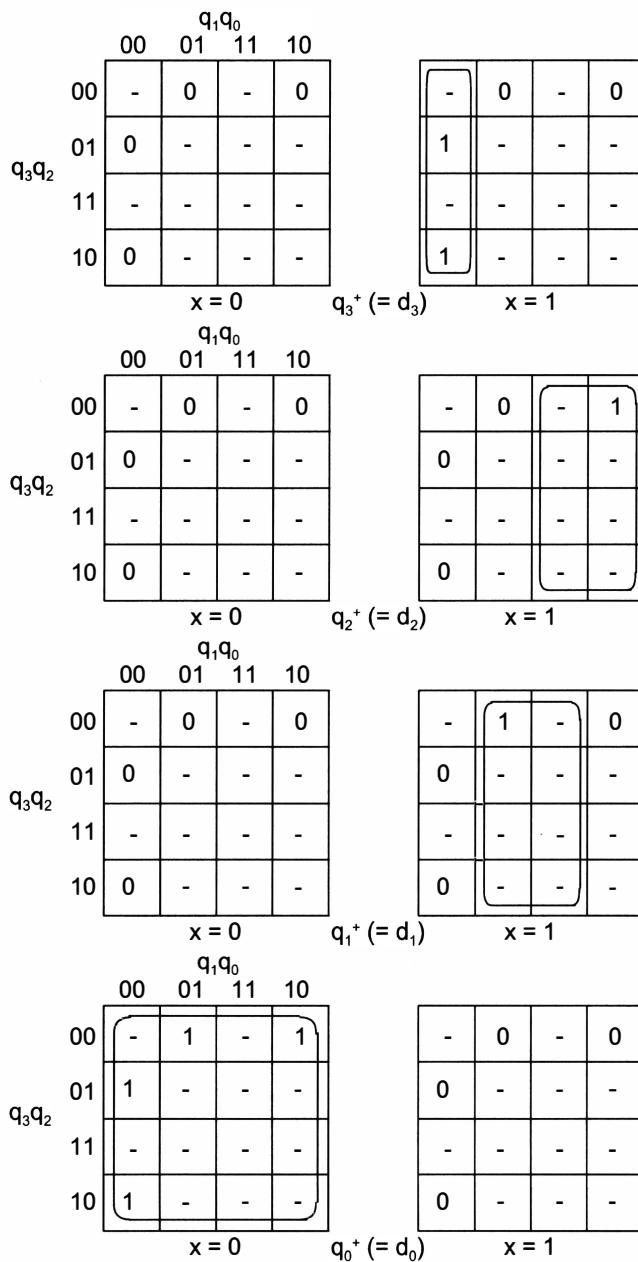
tillstånd	kodning "one-hot"
	$q_3q_2q_1q_0$
start	0 0 0 1
first_1	0 0 1 0
second_1	0 1 0 0
third_1	1 0 0 0

Tillståndskodningen "one-hot" innebär kodning med kodord som vardera innehåller endast en etta, en "het etta". Antalet bitar i kodorden blir lika med antalet tillstånd, i tillståndsdiagrammet ovan sålunda fyra bitar.

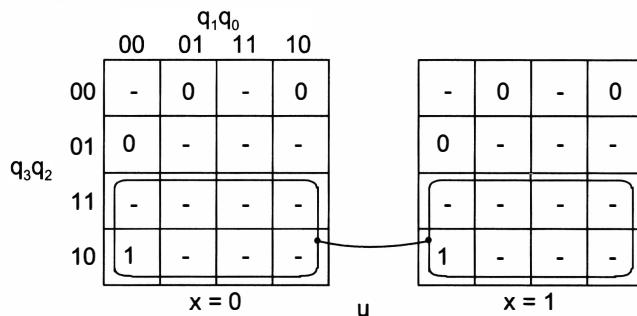
Nuvarande tillstånd $q_3q_2q_1q_0$	Nästa tillstånd $q_3^+q_2^+q_1^+q_0^+$		Utsignal $u$	
	Insignal $x$			
	0	1		
0001	0001	0010	0	
0010	0001	0100	0	
0100	0001	1000	0	
1000	0001	1000	1	

Figur 5.11 Tillståndstabell med tillståndskodning "one-hot" för Sdet1\_mo.

**Booleska uttryck för nästa tillstånd  $q^+$  och utsignal  $u$**   
**Karnaughdiagram**



Figur 5.12 Karnaughdiagram för nästa tillstånd  $q^+$ .

Figur 5.13 Karnaughdiagram för utsignalen  $u$ 

Booleska uttryck

$$q_3^+ = q_1' q_0' x$$

$$q_2^+ = q_1 x$$

$$q_1^+ = q_0 x$$

$$q_0^+ = x'$$

$$u = q_3$$

Kodningen "one-hot" ger normalt mindre komplexitet i form av färre produkttermer i booleska uttrycken till nästa tillstånd och utsignaler p.g.a. att kodorden bara innehåller en etta, men till priset av fler bitar (antal D-vippor) i tillståndsregistret. Kodningen "one-hot" ökar antalet tillstånd. Tillståndsdiagrammet till *Sdet1\_mo* konstruerades med fyra tillstånd, men tillståndskodningen "one-hot" med fyra bitar ger  $2^4 = 16$  tillstånd. Kodningen "one-hot" kan vara lämplig att använda i PLD av typ FPGA, som innehåller många D-vippor, men inte har så komplexa OCH-ELLER-nät för generering av nästa tillstånd och utsignaler.

Realiseringen av *Sdet1\_mo* med tillståndskodningen "one-hot" resulterade i en sekvenskrets med totalt 16 tillstånd, vilket är 12 "extra" tillstånd utöver de fyra tillstånd som finns i tillståndsdiagrammet. Låt oss studera hur tillståndsovergångar och utsignal blivit för dessa extra tillstånd. I figuren

nedan visas den fullständiga tillståndstabellen med samtliga 16 tillstånd och de fyra ursprungliga tillstånden i tillståndsdiagrammet skuggade.

Nuvarande tillstånd $q_3 q_2 q_1 q_0$	Nästa tillstånd $q_3^+ q_2^+ q_1^+ q_0^+$		Utsignal u	
	Insignal x			
	0	1		
0001	0001	0010	0	
0010	0001	0010	0	
0100	0001	0100	0	
1000	0001	1000	1	
0000	0001	1000	0	
0011	0001	0110	0	
0101	0001	0010	0	
0110	0001	0100	0	
0111	0001	0110	0	
1001	0001	0010	1	
1010	0001	0100	1	
1011	0001	0110	1	
1100	0001	1000	1	
1101	0001	0010	1	
1110	0001	0100	1	
1111	0001	0100	1	

Figur 5.14 Fullständiga tillståndstabellen för Sdet1\_mo med tillståndskodningen "one-hot".

Vi har nu gått igenom de fundamentala stegen i realiseringen av en sekvenskrets, som kan sammanfattas enligt nedan.

### Realisering av en sekvenskrets – fundamentala steg

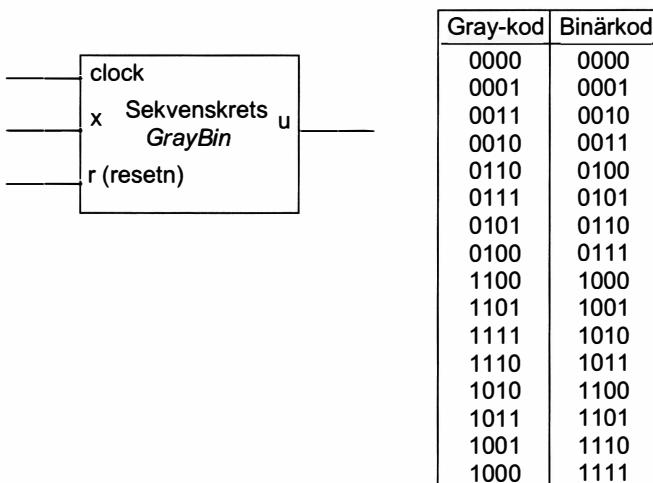
1. Specifikation
2. Tillståndsdiagram/tillståndstabell
3. Tillståndskodning
4. Booleska uttryck för nästa tillstånd och utsignaler
5. Kretsschema

## Realisering av en sekvenskrets typ Mealy

I kapitel 2 *analyserade* vi beteendet för en given sekvenskrets av typ Mealy, en sekvensdetektor benämnd *Sdet1\_me* och jämförde beteendet med sekvensdetektorn *Sdet1\_mo*, realiserad som typ Moore. Vi skall nu göra *syntes* av en sekvenskrets typ Mealy och börjar med beskrivningen av beteendet.

### Specifikation av beteendet för sekvensdetektorn GrayBin

En sekvenskrets *GrayBin* som skall omvandla 4-bitars Gray-kod (se kapitel 1, figur 1.17 och tabell 1.4) till 4-bitars vanlig binärkod enligt tabellen nedan, skall konstrueras. Sekvenskretsen skall ha en insignal x, en insignal r (resetn), en utsignal u samt en klocksignal clock.

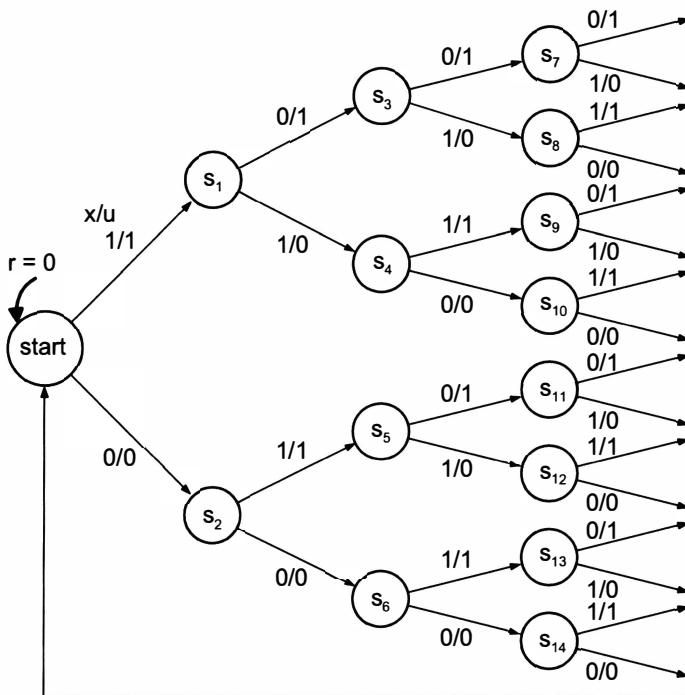


Figur 5.15 Blocksymbol för sekvenskretsen GrayBin och översättningstabell.

Orden i Gray-kod matas in i serieform, med MSB först, på ingången x synkront med klocksignalen clock. Orden i binärkod skall matas ut på utgången u bit för bit samtidigt som bitarna i Gray-koden inkommer, endast med den fördräjning som finns i grindnätet i sekvenskretsen. Sekvenskretsen antages befina sig i ett starttillstånd då första ordet inkommer. Övergång till starttillståndet sker med en synkron resetsignal r, aktiv Låg.

## Tillståndsdiagram

Kravet i specifikationen att bitarna i binärkoden skall matas ut på utgången i samma klockpulsintervall som motsvarande bit i Gray-koden matas in på ingången  $x$ , leder fram till en sekvenskrets av typ Mealy. Tillståndsdiagrammets utseende för en sekvenskrets av typ Mealy berördes i kapitel 2. Specifikationen av beteendet med en översättningstabell gör det lämpligt att konstruera tillståndsdiagrammet som ett träd direkt utgående från tabellen. Starttillståndet betecknas *start*, övriga tillstånd  $s_1, s_2, \dots$ .

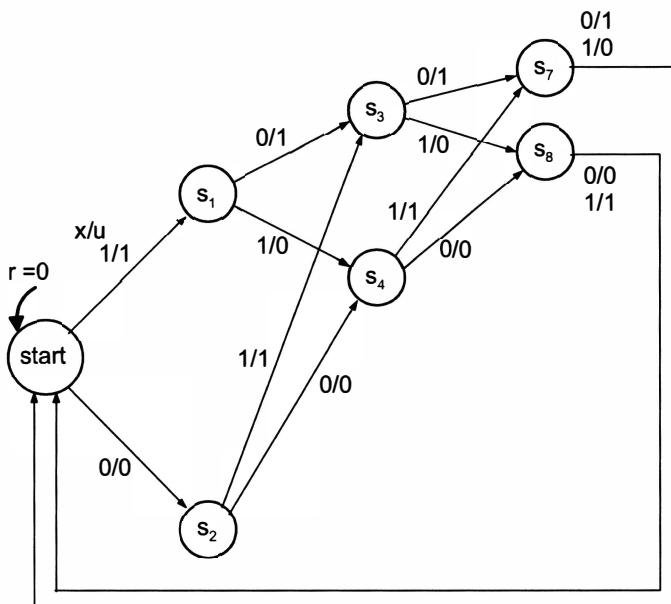


Figur 5.16 Tillståndsdiagram för sekvenskretsen *GrayBin*.

## Tillståndsminimering

Vi ser i tillståndsdiagrammet ovan att tillstånden  $s_4$  och  $s_6$  har exakt samma utgående grenar. Tillstånden  $s_6$ ,  $s_{13}$ , och  $s_{14}$  kan därför tas bort och tillståndsövergången  $0/0$  från  $s_2$  till  $s_6$  flyttas till  $s_4$ . Tillstånden  $s_3$  och  $s_5$  har också exakt samma grenar, varför tillståndet  $s_5$  kan tas bort och tillstånd-

övergången 1/1 från  $s_2$  till  $s_5$  flyttas till  $s_3$ . Tillstånden  $s_7$  och  $s_9$  har samma grenar, varför  $s_9$  kan tas bort och tillståndsövergången 0/1 från  $s_4$  till  $s_9$  flyttas till  $s_7$ . Slutligen ser vi att  $s_8$  och  $s_{10}$  har samma grenar, varför  $s_{10}$  kan tas bort och tillståndsövergången från  $s_5$  till  $s_{10}$  flyttas till  $s_8$ . Det minimerade tillståndsdiagrammet visas i figuren nedan.



Figur 5.17 Minimerat tillståndsdiagram för sekvenskretsen GrayBin.

Antalet tillstånd har genom tillståndsminimeringen minskats från 15 till 7, innebärande att antalet bitar i tillståndsregistret kan minskas från 4 till 3 om man väljer att koda tillstånden med minimalt antal bitar. Tillståndsminimeringen ovan var enkel att utföra, normalt är det inte lika enkelt. Det finns algoritmer för tillståndsminimering, men de har ingen större praktisk betydelse, vilket belyses av det faktum att syntesverktyg för programmerbara kretsar alltid har en eller flera algoritmer för minimering av booleska funktioner, men sällan någon algoritm för tillståndsminimering. Orsaken är att normalt är inte problemet att få plats med tillståndsregistret p.g.a. för många bitar, utan i stället att få plats med booleska funktionerna i OCH-ELLER-näten. Vi har ju t.o.m. sett hur man ibland medvetet ökar antalet

bitar i tillståndsregistret med kodningen ”one-hot” för att minska komplexiteten hos booleska funktionerna. Tillståndsminimeringen ovan har gjorts för att belysa att det tillståndsdigram som man konstruerat utgående från specifikationen sålunda kan innehålla fler tillstånd än vad som är nödvändigt för att realisera det önskade beteendet.

### Tillståndstabell

Nuvarande tillstånd	Nästa / Utsignal u tillstånd	
	x	
	0	1
start	s <sub>2</sub> /0	s <sub>1</sub> /1
s <sub>1</sub>	s <sub>3</sub> /1	s <sub>4</sub> /0
s <sub>2</sub>	s <sub>4</sub> /0	s <sub>3</sub> /1
s <sub>3</sub>	s <sub>7</sub> /1	s <sub>8</sub> /0
s <sub>4</sub>	s <sub>8</sub> /0	s <sub>7</sub> /1
s <sub>7</sub>	start/1	start/0
s <sub>8</sub>	start/0	start/1

Figur 5.18 Tillståndstabell för sekvenskretsen GrayBin.

### Tillståndskodning

Vi väljer tillståndskodning ”binary”, dvs kodning av den uppräknebara tillståndsmängden {start, s<sub>1</sub>, s<sub>2</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>7</sub>, s<sub>8</sub>} med 3-bitars binärtal i ordning med början på binärtalat 000.

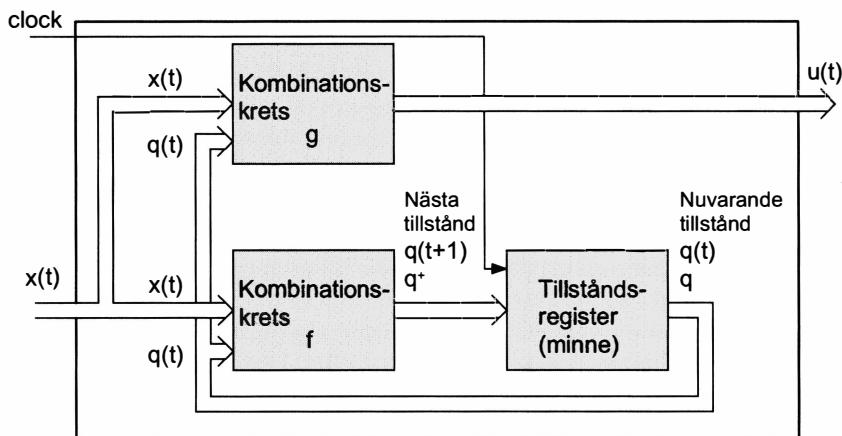
tillstånd	kodning ”binary”
	q <sub>2</sub> q <sub>1</sub> q <sub>0</sub>
start	0 0 0
s <sub>1</sub>	0 0 1
s <sub>2</sub>	0 1 0
s <sub>3</sub>	0 1 1
s <sub>4</sub>	1 0 0
s <sub>7</sub>	1 0 1
s <sub>8</sub>	1 1 0

Nuvarande tillstånd $q_2q_1q_0$	Nästa tillstånd / Utsignal u	
	$q_2^+q_1^+q_0^+/ u$	
	x	
0 0 0	0 1 0/0	0 0 1/1
0 0 1	0 1 1/1	1 0 0/0
0 1 0	1 0 0/0	0 1 1/1
0 1 1	1 0 1/1	1 1 0/0
1 0 0	1 1 0/0	1 0 1/1
1 0 1	0 0 0/1	0 0 0/0
1 1 0	0 0 0/0	0 0 0/1

Figur 5.19 Tillståndstabell med tillståndskodning "binary" för GrayBin.

Sekvenskretsen skall realiseras som typ Mealy. Nedan visas den generella modellen för en sekvenskrets av typ Mealy.

### Modell för en sekvenskrets av typ Mealy



Figur 5.20 Modell för en sekvenskrets av typ Mealy.

För Mealy-modellen ovan gäller

$$q(t+1) = f(q(t), x(t))$$

$$u(t) = g(q(t), x(t))$$

eller med parametern  $t$  i uttrycken ovan utelämnad och nästa tillstånd betecknad med  $q^+$

$$q^+ = f(q, x)$$

$$u = g(q, x)$$

Karakteristiskt för Mealy-modellen är att utsignalen  $u$  beror, förutom av nuvarande tillstånd  $q$ , även av insignalen  $x$ .

Vi fortsätter nu realiseringen av sekvenskretsen GrayBin.

### Booleska uttryck för nästa tillstånd $q^+$ och utsignal $u$

		$q_0x$								
		00	01	11	10					
		00	0	0	1	0				
$q_2q_1$	00	1	0	1	1	1	0	0	1	
	01	0	1	1	1	0	1	1	0	
	11	0	0	-	-	0	0	-	-	
	10	1	1	0	0	1	0	0	0	
		$q_2^+$				$q_1^+$				
		00	01	11	10	00	01	11	10	
		00	0	1	1	0	1	0	1	
		01	1	0	0	1	1	0	1	
		11	0	1	-	-	0	0	-	
		10	1	1	0	0	1	0	0	
		$q_0^+$				$u$				
		00	01	11	10	00	01	11	10	
		00	0	1	0	1	0	1	0	
		01	1	0	0	1	1	0	1	
		11	0	1	-	-	0	0	-	
		10	1	1	0	0	1	0	0	

Figur 5.21 Karnaughdiagram för nästa tillstånd  $q^+$  och utsignal  $u$ .

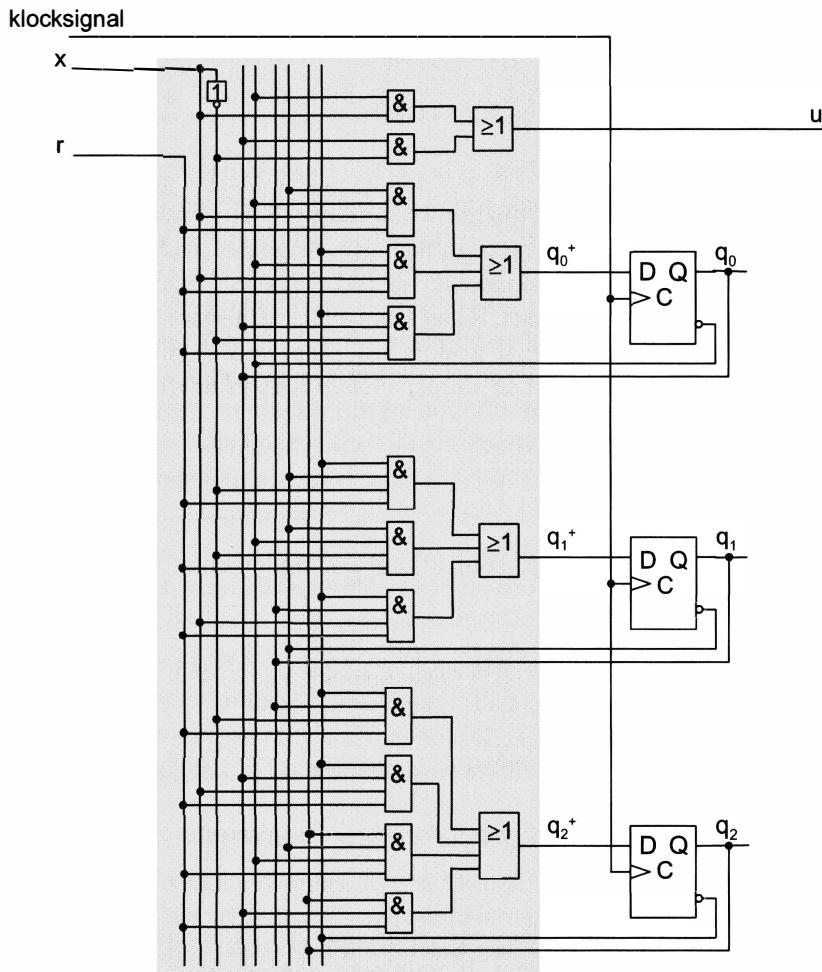
$$q_2^+ = q_2' q_0 x + q_2' q_1 x' + q_2 q_1' q_0' + q_1 q_0$$

$$q_1^+ = q_2' q_1' x' + q_2' q_1 x + q_1' q_0' x'$$

$$q_0^+ = q_1' q_0' x + q_2' q_0' x + q_2' q_0 x'$$

$$u^- = q_0' x + q_0 x'$$

## Kretsschema



Figur 5.22 Kretsschema för sekvenskretsen GrayBin.

I kretsschemat har inkopplats resetsignalen r. Det är speciellt enkelt att tillfoga en resetsignal när starttillståndet har kodats med ett binärord med samtliga bitar nollor. Vid reset skall då samtliga  $q^+$ -signaler, insignaler till D-vipporna bli noll, dvs samtliga produkter i  $q^+$ -funktionerna bli noll. Detta kan åstadkommas genom att samtliga produkter multipliceras (OCH) med resetsignalen r om reset skall vara aktiv Låg, som i detta fall, eller  $r'$  om reset skall vara aktiv Hög.

$q^+$ -funktionerna med reset r blir sålunda

$$q_2^+ = rq_2' q_0 x + rq_2' q_1 x' + rq_2 q_1' q_0' + rq_1 q_0$$

$$q_1^+ = rq_2' q_1' x' + rq_2' q_1 x + rq_1' q_0' x'$$

$$q_0^+ = rq_1' q_0' x + rq_2' q_0' x + rq_2 q_0 x'$$

Infogande av en resetsignal medför att samtliga OCH-grindar måste utökas med en extra ingång för resetsignalen. Resetsignalen ovan har tillfogats som en *synkron* reset, innebärande att resetsignalen anger reset, medan klocksignalen verkställer reset. Reset kan i stället tillfogas som en *asynkron* reset genom att resetsignalen ansluts till vippornas asynkrona nollställningsingång Clear, varvid man slipper utöka OCH-grindarna med en extra ingång.

### Exempel 5.1

En sekvenskrets TwoComp, som bildar 2-komplementet till ett binärtal, skall konstrueras.



Figur 5.23 Sekvenskrets TwoComp som bildar 2-komplementet till ett binärtal.

Binärtal, som kan innehålla godtyckligt många bitar, matas in i serieform på ingången x med LSB först. Inmatningen sker synkront med klocksignalen clock. Inmatningen av ett nytt tal föregås alltid av en synkron reset r, aktiv Låg,  $r = 0$ , som överför sekvenskretsen i ett starttillstånd. På utgången

u skall 2-komplementet matas ut i serieform, bit för bit, i samma klocksignalperiod som motsvarande bit i talet x matas in.

### Lösning

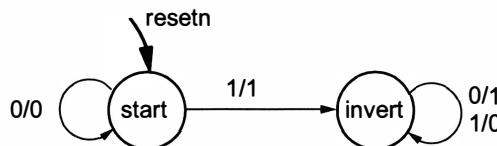
#### Tillståndsdiagram

2-komplementet skall uppenbart bildas enligt den alternativa regeln 4.11 som tidigare redovisats i kapitel 4.

*Alternativ regel för att bilda 2-komplementet:*

1. Leta upp den minst signifikanta ettan i binärtalatet.
2. Invertera alla siffror till vänster om denna.

Kravet att utsignalen u skall matas ut i samma klocksignalperiod som tillhörande x inkommer leder till en sekvenskrets av typ Mealy. Vi börjar med ett starttillstånd *start*. För insignalen  $x = 0$  kan vi stanna kvar i starttillståndet, eftersom inget behöver kommas ihåg för de först inkommande nollorna. Det är först när den minst signifikanta ettan kommer som det behöver registreras. Utsignalen för  $x = 0$  i starttillståndet skall enligt regeln för 2-komplementet ovan vara  $u = 0$ . För  $x = 1$  i starttillståndet måste vi gå till ett nytt tillstånd, som vi döper till *invert*, eftersom därefter invertering av bitarna skall ske. Utsignalen för  $x = 1$  i starttillståndet skall enligt regeln ovan för 2-komplementet vara  $u = 1$ .



Figur 5.24 Tillståndsdiagram för sekvenskretsen *TwoComp* av typ Mealy.

I tillståndet *invert* kan vi sedan stanna kvar för både  $x = 0$  och  $x = 1$  eftersom inget behöver registreras, endast invertering skall ske. Utsignalen i detta tillstånd skall alltså vara  $u = 1$  för  $x = 0$  och  $u = 0$  för  $x = 1$ . Det fullständiga tillståndsdiagrammet innehåller sålunda bara två tillstånd.

## Tillståndstabell

Nuvarande tillstånd $q$	Nästa tillstånd $q^+$ / Utsignal $u$	
	Insignal $x$	
	0	1
start	start/0	invert/1
invert	invert/1	invert/0

Figur 5.25 Tillståndstabell för sekvenskretsen TwoComp av typ Mealy.

## Tillståndskodning

Sekvenskretsen TwoComp har alltså två tillstånd, *start* och *invert*, som måste *kodas* med binära ord för att kunna lagras i sekvenskretsens tillståndsregister. Det räcker med en tillståndsvariabel för att koda två tillstånd. *Tillståndskodningen* (eng. *state assignment*) kan göras på två sätt enligt

Kodning 1	Kodning 2
$q$	$q$
start $\leftrightarrow 0$	start $\leftrightarrow 1$
invert $\leftrightarrow 1$	invert $\leftrightarrow 0$

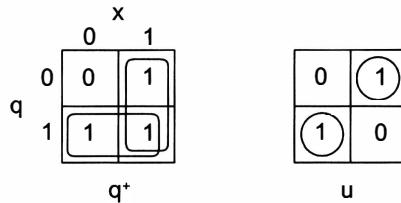
Val av kod påverkar komplexiteten hos realiseringen av sekvenskretsen. Reset blir enkel att implementera om starttillståndet kodats med enbart nollor. Vi väljer därför kodning 1.

## Booleska funktioner för nästa tillstånd och utsignal

Nuvarande tillstånd $q$	Nästa tillstånd $q^+$ / Utsignal $u$	
	Insignal $x$	
	0	1
0	0/0	1/1
1	1/1	1/0

Figur 5.26 Tillståndstabell för sekvenskretsen TwoComp med kodning 1.

Ur tillståndstabellen erhålls direkt Karnaughdiagrammen nedan för funktionerna  $q^+$  och  $u$



Figur 5.27 Karnaughdiagram för sekvenskretsen TwoComp typ Mealy.

Ur Karnaughdiagrammen erhålls

$$q^+ = x + q$$

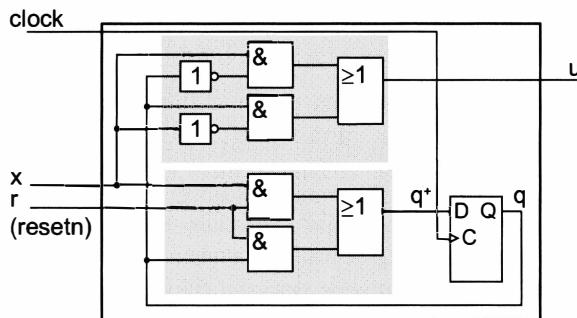
$$u = q'x + qx'$$

### Reset

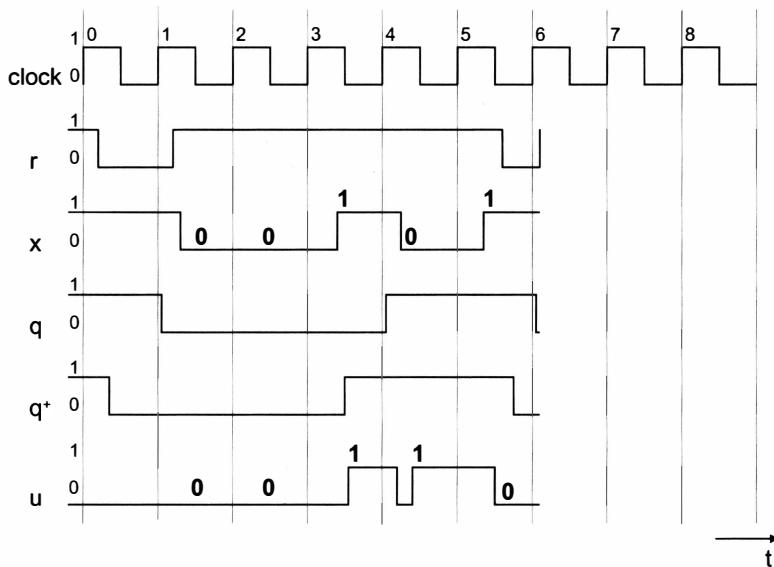
Reset  $r = 0$  skall ge övergång till starttillståndet. Eftersom starttillståndet är kodat som 0, innebär detta att  $r = 0$  skall ge  $q^+ = 0$ . Detta villkor kan uppfyllas genom införande av signalen  $r$  i  $q^+$  enligt

$$q^+ = r(x + q) = rx + rq$$

### Schema för sekvenskretsen



Figur 5.28 Schema för sekvenskretsen TwoComp av typ Mealy.

**Tidsdiagram vid inmatning av  $x = 10100$** 

*Figur 5.29 Tidsdiagram för sekvenskretsen TwoComp vid inmatning av  $x=10100$ .*

I figuren ovan visas ett tidsdiagram för sekvenskretsen TwoComp vid bildande 2-komplementet till talet  $x = 10100$ . Låt oss analysera beteendet i de olika klockpulsintervallen. Inmatningen av talet  $x$  sker i intervallet 1–5 med LSB först, vilket framgår av diagrammet där bitarna är angivna på signalen  $x$ . I diagrammet har hänsyn tagits till fördräjningen i grindarna.

*Klockpulsintervall 0.* Reset förbereds med  $r = 0$ , som efter fördräjning i OCH-, ELLER-grindar ger  $q^+ = 0$ . Insignalen  $x$ , har ett värde från en tidigare inmatning, liksom  $q$  och  $u$ .

*Klockpulsintervall 1.* Klockpulsens positiva flank verkställer reset och  $q$  tilldelas värdet av  $q^+$  och får värdet 0 efter omslagsfördräjningen i tillståndsregistret, markerad liten i diagrammet. Inmatningen av  $x = 0$  görs inne i intervallet, som ger  $u = 0$  (ingen förändring).  $q^+$  förändras inte och bibehåller värdet 0.

*Klockpulsintervall 2.* Klockpulsens positiva flank verkställer att  $q$  tilldelas värdet av  $q^+$ , som inte ger någon förändring av  $q$ . Inmatningen av  $x = 0$  görs

(värdet från förra intervallet bibehålls), som ger  $u = 0$  (ingen förändring).  $q^+$  förändras inte och bibehåller värdet 0.

*Klockpulsintervall 3.* Klockpulsens positiva flank verkställer att  $q$  tilldelas värdet av  $q^+$ , som inte ger någon förändring av  $q$ . Inmatningen av  $x = 1$  görs inne i intervallet, som ger  $u = 1$  och  $q^+ = 1$  efter födröjning i inverterare, OCH-, ELLER-grindar.

*Klockpulsintervall 4.* Klockpulsens positiva flank verkställer att  $q$  tilldelas värdet av  $q^+$  och får värdet 1 (kommer att bibehållas tills Reset). Inmatningen av  $x = 0$  görs inne i intervallet, som ger  $u = 1$  (resten av bitarna i  $x$  skall ju inverteras enligt regeln för 2-komplementbildningen). Observera att  $u$  blir 0 en tid innan  $u$  blir 1 p.g.a. att  $x$  har kvar värdet 1 en tid in i intervallet efter att  $q$  blivit 1.

*Klockpulsintervall 5.* Klockpulsens positiva flank verkställer att  $q$  tilldelas värdet av  $q^+$  och får värdet 1 (oförändrat värde). Inmatningen av  $x = 1$ , sista biten i talet  $x$ , görs inne i intervallet, som ger  $u = 0$ . Reset förbereds med  $r = 0$ , som efter födröjning i OCH-, ELLER-grindar ger  $q^+ = 0$ .

*Klockpulsintervall 6.* Klockpulsens positiva flank verkställer reset och inmatning av nytt tal  $x$  kan börja.

□

### Exempel 5.2

Konstruera sekvenskretsen TwoComp som typ Moore, innebärande modifiering av specifikationen så att utsignalen  $u$  hörande till en insignal  $x$  i ett klocksignalintervall, matas ut i nästföljande klocksignalintervall.

### Lösning

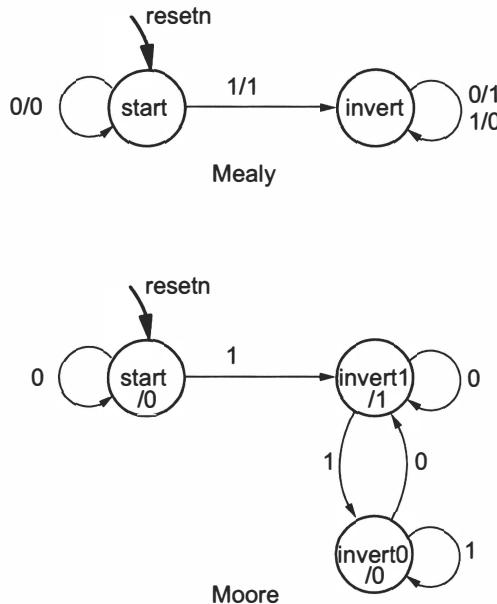
#### Tillståndsdiagram

I stället för att konstruera tillståndsdiagrammet utgående från specifikationen, väljer vi att transformera det tidigare konstruerade tillståndsdiagrammet av typ Mealy till typ Moore.

#### Regel för transformering av ett tillståndsdiagram av typ Mealy till Moore

Varje tillstånd *till* vilket leder till tillståndsövergångar med olika utsignalvärde, delas upp i lika många tillstånd som antalet olika utsignalvärden på tillståndsövergångarna till tillståndet.

Transformeringen för sekvenskretsen TwoComp av Mealy till Moore blir enligt figuren nedan, där även tillståndsdiagrammet av typ Mealy visas för jämförelse.



Figur 5.30 Transformering Mealy till Moore för sekvenskretsen TwoComp.

Tillståndet *start* skall inte delas upp eftersom tillståndsövergången till detta tillstånd bara har ett utsignalvärdet, i detta fall 0. Tillståndet *invert* skall däremot delas upp i två tillstånd eftersom tillståndsövergångar med utsignalvärdena 0 och 1 leder till detta tillstånd. Vi döper de två tillstånden efter utsignalvärdena till *invert0* och *invert1*.

### Tillståndstabell

Tillståndstabellen nedan har tre tillstånd, innehärande att kodningen kräver minst två tillståndsvariabler, eftersom  $2 = 2^1 < 3 < 2^2 = 4$ . Två tillståndsvariabler ger de fyra kodorden 00, 01, 10 och 11, ur vilka tre skall väljas. Valet kan ske på 24 olika sätt, som ger olika komplexitet hos kombinationskretsarna i sekvenskretsen. Som tidigare påpekats är det enkelt att realisera

reset om starttillståndet kodas med ett ord med enbart nollar. Om vi väljer att koda start som 00, så är det sex möjligheter att koda de två andra tillstånden.

Nuvarande tillstånd $q$	Nästa tillstånd $q^+$		Utsignal $u$	
	Ingångsvärde $x$			
	0	1		
start	start	invert1	0	
invert1	invert1	invert0	1	
invert0	invert1	invert0	0	

Figur 5.31 Tillståndstabell för sekvenskretsen TwoComp av typ Moore.

Eftersom utsignalen för en sekvenskrets av typ Moore är en funktion enbart av tillståndet, så är det naturligt att försöka välja tillståndskoder så att det råder ett så enkelt samband som möjligt mellan tillståndskod och utsignal. Enklast i detta fall är att låta utsignalen vara lika med värdet av en tillståndsvariabel eller inversen av en tillståndsvariabel.

Låt oss prova några olika tillståndskodningar.

$$\begin{array}{ll} \text{Kodning 1} & q_1 q_0 \\ \text{start} & \leftrightarrow 00 \\ \text{invert1} & \leftrightarrow 01 \\ \text{invert0} & \leftrightarrow 10 \end{array}$$

$$\begin{array}{ll} \text{Kodning 2} & q_1 q_0 \\ \text{start} & \leftrightarrow 00 \\ \text{invert1} & \leftrightarrow 11 \\ \text{invert0} & \leftrightarrow 10 \end{array}$$

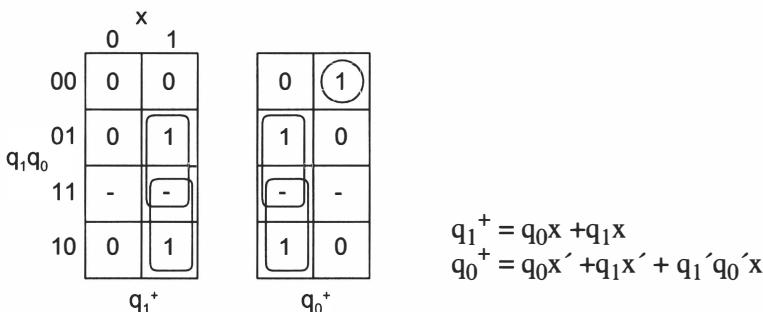
För båda kodningarna gäller att  $u = q_0$ . Vi skall nu undersöka komplexiteten för  $q_1^+$  och  $q_0^+$ .

## Booleska funktioner för nästa tillstånd och utsignal

### Kodning 1

Nuvarande tillstånd $q_1 q_0$	Nästa tillstånd $q_1^+ q_0^+$		Utsignal u	
	Insignal x			
	0	1		
00	00	01	0	
01	01	10	1	
10	01	10	0	

Figur 5.32 Tillståndstabell för TwoComp typ Moore kodning 1.

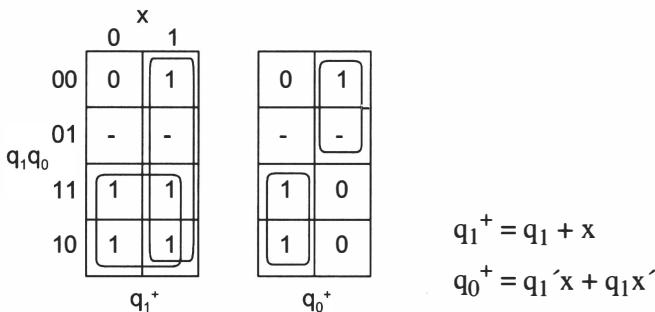


Figur 5.33 Karnaughdiagram för TwoComp typ Moore kodning 1.

### Kodning 2

Nuvarande tillstånd $q_1 q_0$	Nästa tillstånd $q_1^+ q_0^+$		Utsignal u	
	Insignal x			
	0	1		
00	00	11	0	
11	11	10	1	
10	11	10	0	

Figur 5.34 Tillståndstabell för TwoComp typ Moore kodning 2.



Figur 5.35 Karnaughdiagram för TwoComp typ Moore kodning 2.

Vi konstaterar att kodning 2 ger enklast  $q^+$ -funktioner. Komplettering med reset r, aktiv låg, ger

$$q_1^+ = rq_1 + rx$$

$$q_0^+ = rq_1'x + rq_1x'$$

### Outnyttjade tillstånd

Kodning av tillstånden i sekvenskretsen TwoComp med två tillståndsvariabler ger en krets innehållande totalt  $2^2 = 4$  tillstånd. Beteendet för sekvenskretsen har beskrivits med bara tre tillstånd. Det finns alltså ett outnyttjat tillstånd. För detta tillstånd har nästa tillstånd markerats som don't care i Karnaughdiagrammen. Nästa tillstånd bestäms av inringningarna för  $q_1^+$  och  $q_0^+$ , som gjorts med tanke på minimala booleska uttryck. För de två kodningarna ovan blir nästa tillstånd för det outnyttjade tillståndet enligt nedan (om don't care ingår i en inringning blir  $q^+ = 1$ , medan om den inte ingår blir  $q^+ = 0$ ).

#### Kodning 1

Outnyttjat tillstånd                    11

Nästa tillstånd för  $x = 0$ :    01

– ” –                                     $x = 1$ :    10

#### Kodning 2

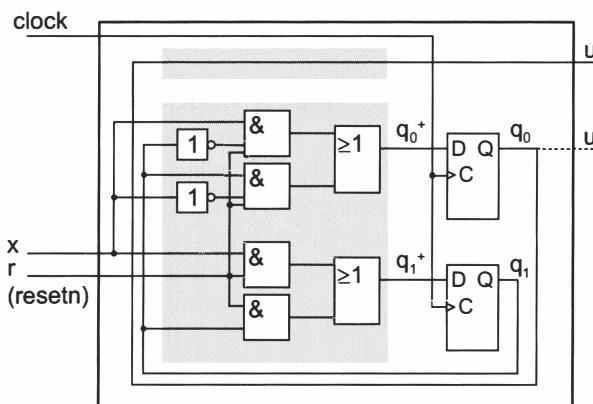
01

00

11

Outnyttjade tillstånd kan behandlas på olika sätt. Don't care som ovan, ger minst komplexitet hos  $q^+$ -funktionerna. Om detta inte är acceptabelt får tillståndsövergångarna för de outnyttjade tillstånden specificeras, t.ex. att de skall leda till starttillståndet, men det leder till ökad komplexitet p.g.a. fler restriktioner. Vilket alternativ som bör användas får avgöras från fall till fall av konstruktören. Don't care ger som sagt minst komplexitet, men även om man specificerat de outnyttjade tillstånden är det inte säkert att det räcker för att återställa korrekt beteende vid en störning, utan det kan kräva ytterligare åtgärder.

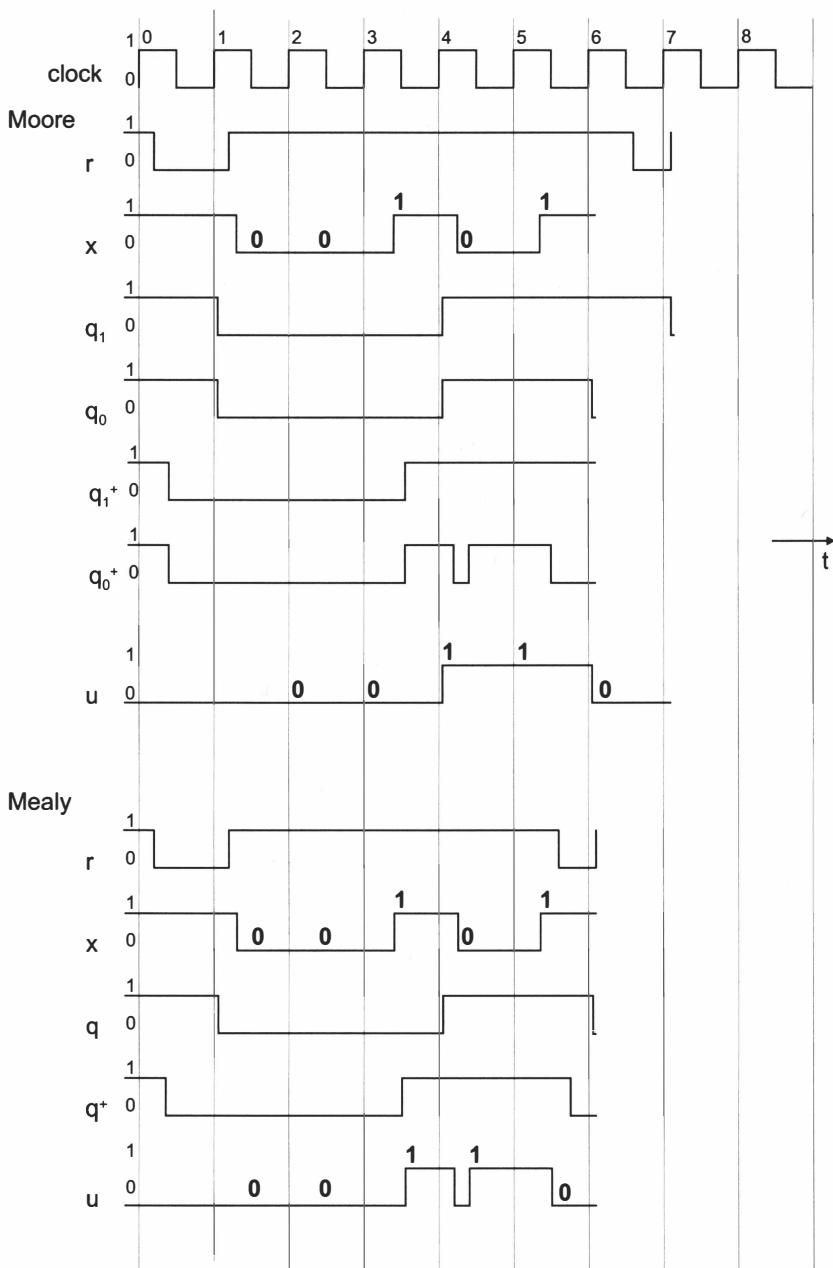
### Schema för sekvenskretsen



Figur 5.36 Schema för sekvenskretsen TwoComp av typ Moore med kodning 2.

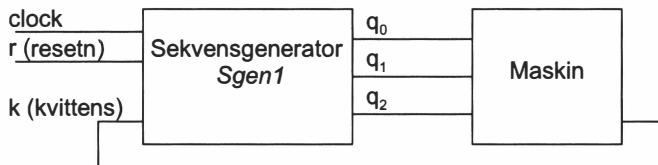
### Tidsdiagram vid inmatning av $x = 10100$

I nedre delen av tidsdiagrammet har som jämförelse ritats in signalerna för Mealy-modellen som tidigare visats. Vi ser i signaldiagrammet för Moore-modellen att utsignalen bara ändrar värde på klocksignalens aktiva flank och är sedan stabil under hela klockpulsintervallet. Detta är karakteristiskt för Moore-modellen liksom att utgångsvärdena ligger ett klockpulsintervall efter Mealy-modellen. I övre delen av diagrammet har för Moore-modellen förberedelse för reset flyttats ett intervall senare, ty annars skulle sista utgångsvärdet missats.

Figur 5.37 Tidsdiagram för TwoComp typ Moore vid inmatning av  $x=10100$ .

**Exempel 5.3**

En sekvensgenerator *Sgen1*, som skall generera ett startförlöpp till en maskin, skall konstrueras.



*Figur 5.38 Sekvensgenerator Sgen1.*

Startförlöppet skall föregås av en aktiv låg reset,  $r = 0$ , som ger övergång till starttillståndet  $q_2q_1q_0 = 000$ . Efter reset skall genereras sekvensen

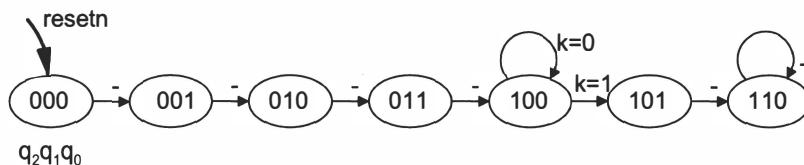
$q_2q_1q_0 : 000, 001, 010, 011, 100, 101, 110, 110, \dots$ .

Under generering av sekvensen får tillståndet 100 inte lämnas förrän maskinen ger kvittens  $k = 1$ .

*Lösning*

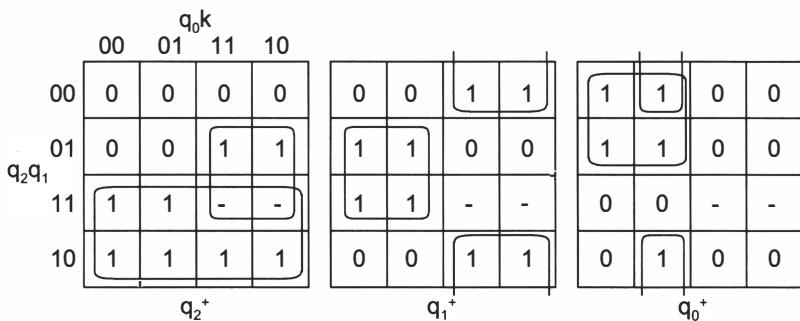
**Tillståndsdiagram**

Tillståndsvariablerna används som utsignaler.



*Figur 5.39 Tillståndsdiagram för sekvensgeneratorn Sgen1.*

## Booleska funktioner för nästa tillstånd och utsignal



Figur 5.40 Karnaughdiagram för  $q^+$  för Sgen1.

$$q_2^+ = q_2 + q_1 q_0$$

$$q_1^+ = q_1 q_0' + q_1' q_0$$

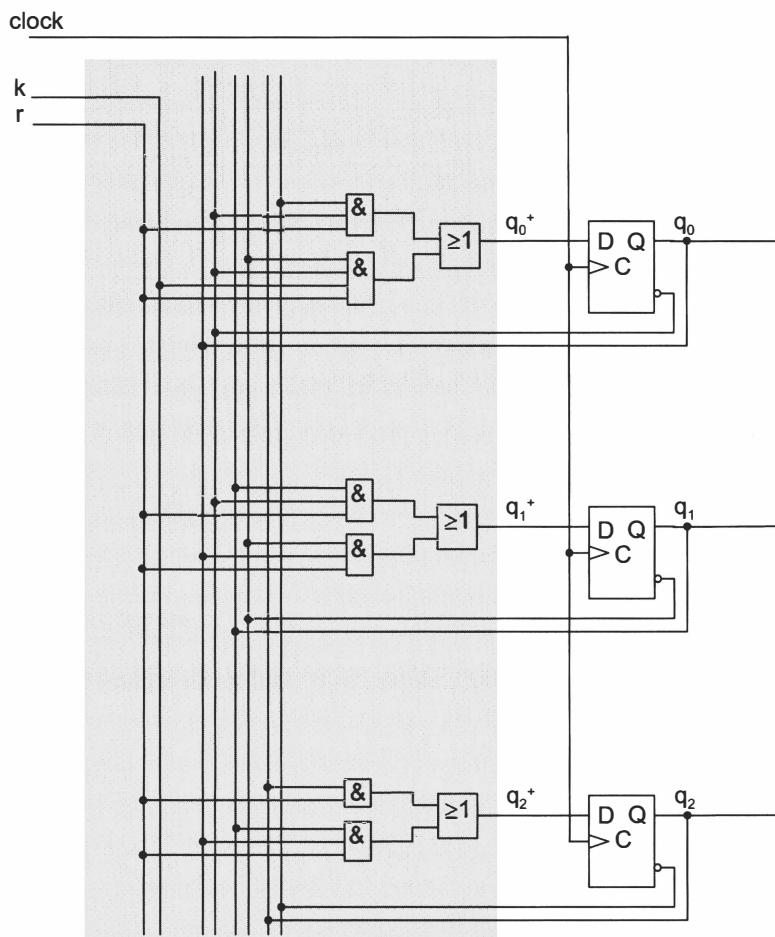
$$q_0^+ = q_2' q_0' + q_1' q_0' k$$

Starttillståndet är kodat som 000. Reset  $r = 0$  skall ge övergång till starttillståndet 000. Med reset  $r$  blir  $q^+$

$$q_2^+ = rq_2 + rq_1 q_0$$

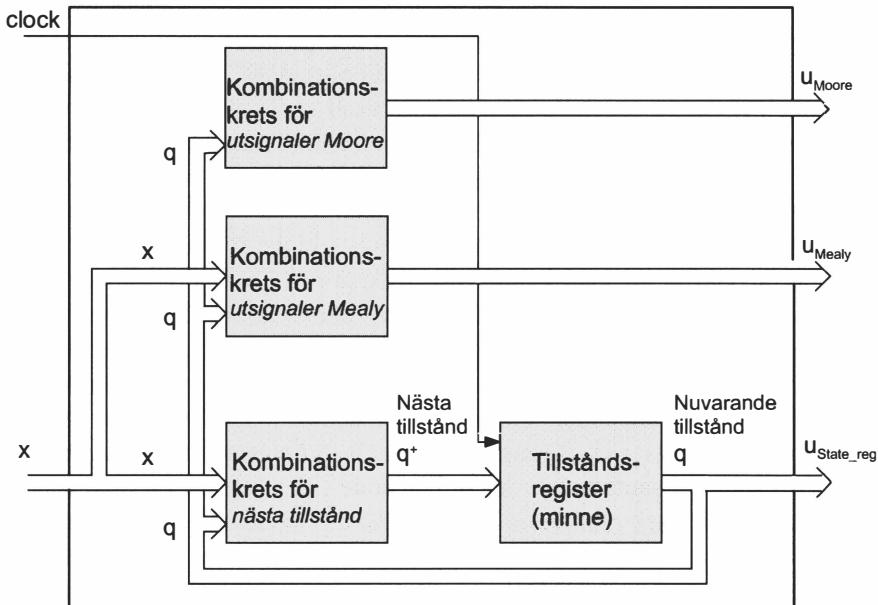
$$q_1^+ = rq_1 q_0' + rq_1' q_0$$

$$q_0^+ = rq_2' q_0' + rq_1' q_0' k$$

**Schema för Sgen1**

Figur 5.41 Schema för Sgen1.

Vi har nu sett exempel på sekvenskretsar av typ Moore och Mealy. Ofta är sekvenskretsar inte ”rena” tillämpningar av sekvenskretsmodellerna, utan innehåller delar från båda modellerna. En sådan blandad sekvenskretsmodell visas nedan. Där finns utsignaler av typ Moore och Mealy och även utsignaler direkt från tillståndsregistret, som vi såg i sista exemplet med sekvenskretsgeneratorn Sgen1.



Figur 5.42 Generell sekvenskretsmodell.

I sekvenskretsarna i detta kapitel har vi av bekvämlighet betecknat reset med  $r$ . Av beteckningen framgår det inte om den är aktiv låg eller hög eller om den är synkron eller asykon. I kapitel 2 betecknade vi en reset, som var aktiv låg och asynkron, som *resetna*, där  $n$  ”not” betecknar aktiv låg och  $a$  asynkron. Det är lämpligt att försöka använda något sådant system, speciellt i VHDL-beskrivningar.

## 5.2 Räknare

Vi har i kapitel 2 analyserat en räknare, en 2-bitars binärräknare. En *räknare* (eng. *counter*) är i sin ursprungliga form ”en enhet som räknar antalet inkommande pulser”. Räknaren är en sekvenskrets som registrerar de inkommande pulserna genom att för varje puls gå till ett nytt tillstånd som representerar antalet pulser. En räknares utsignaler är normalt tillståndsvariablerna, men det kan även finnas andra utsignaler.

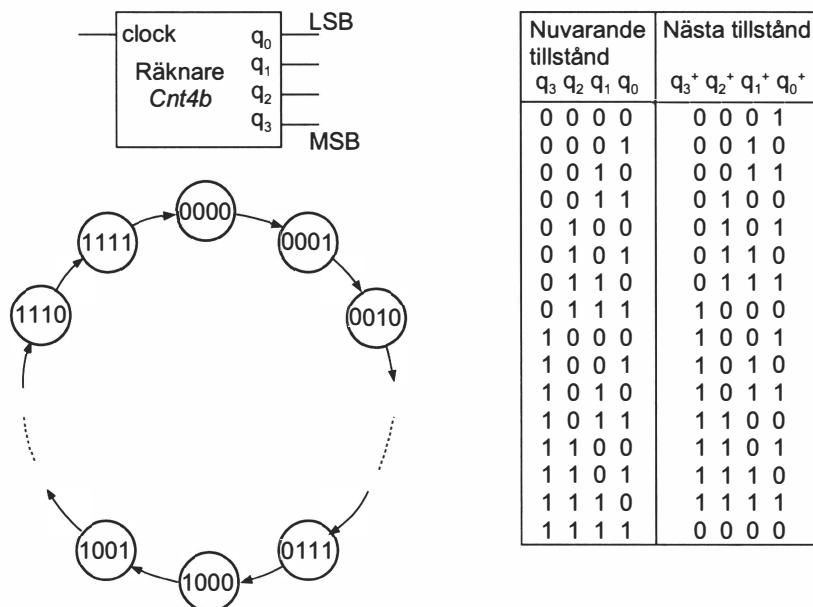
I databöcker för standardkretsar finner man många olika typer av räknare. I dag när digitala kretsar ofta realiseras i PLD eller ASIC kan man själv relativt enkelt realisera speciella räknare som skall ingå i den aktuella kretsen. Man kan i det hårdvarubeskrivande språket VHDL, som vi skall se i kapitel 9, med några få textrader beskriva beteendet för en speciell räknare och sedan realisera den i en krets. Beskrivningen av en speciell räknare görs dock säkert utgående från det fundamentala beteendet för generella räknare, som det är viktigt att man känner till. Räknare har också en fundamental struktur som man måste känna till, liksom sambandet mellan beteende och struktur, för att kunna göra en optimal realisering. När det gäller strukturen så bestäms den vid realisering till stor del av syntesverktyget utgående från det specificerade beteendet. Ibland händer det att syntesverktyget inte får plats med exempelvis en räknare i en aktuell PLD. Det kan då vara värdefullt att känna till sambandet mellan beteende och struktur. En liten acceptabel modifiering av räknarens beteende kan göra att räknaren får plats i PLD:n. En orsak till att en räknare inte får plats i en PLD kan vara att antalet produkttermer är för stort. Det kan då vara värdefullt att veta att antalet produkttermer normalt blir färre om räknaren realiseras med T-vippor jämfört med om den realiseras med D-vippor. Vid realisering med T-vippor får den kanske plats i PLD:n.

Räknare brukar indelas i *synkrona räknare* (eng. *synchronous counters*) och *asynkrona räknare* (eng. *asynchronous counters* eller *ripple counters*). En *synkron räknare* karakteriseras av att samtliga vippor klockas av en gemensam klocksignal. I en sådan räknare slår vipporna om samtidigt och räknarens utgångar är därför ostabila bara under vippans omslagstid. I en *asynkron räknare* klockas vippor av andra vippors utgångar, innebärande att omslaget hos en vippa orsakar omslag av en annan vippa, vars omslag i sin tur orsakar omslag av en annan vippa osv. I en asynkron räknare kan det därför dröja relativt sett lång tid från det omslagsförloppet startat tills omslaget är fullbordat och utgångarna stabiliserat sig. I dag konstrueras huvudsakligen synkrona räknare och vi kommer enbart att behandla sådana

räknare i det följande. Orsaken till att man förr konstruerade asynkrona räknare och att de första räknarna konstruerades asynkrona, var att de är enklare att realisera än synkrona räknare. Kombinationskretsen  $f$  som genererar nästa tillstånd i sekvenskretsmodellen blir nämligen enklare för en asynkron räknare än för en synkron räknare. För exempelvis en vanlig binärräknare realiseras utan grindar oavsett antal bitar hos räknaren, medan realisera som en synkron räknare blir kombinationskretsen  $f$  ett grindnät vars komplexitet ökar med antalet bitar.

## Binärräknare modulo- $2^n$ med n bitar

En binärräknare modulo- $2^n$  räknar  $0, 1, 2, \dots, (2^n-1), 0, 1, \dots$ . Principen illustreras i figuren nedan med blockschema, tillståndsdiagram och tillståndstabell för en binärräknare modulo- $2^4$ , med 4 bitar som räknar modulo-16, dvs.  $0, 1, 2, \dots, 15, 0, 1, \dots$ .

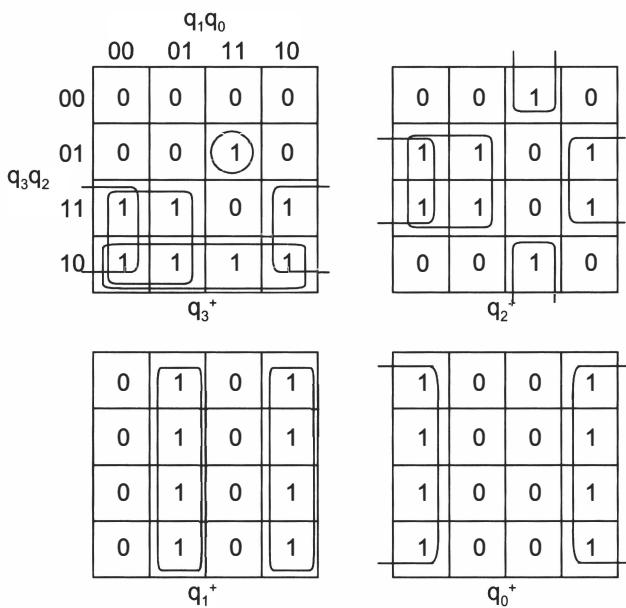


Figur 5.43 Tillståndsdiagram och tillståndstabell för en 4-bitars binärräknare.

Räknaren har fyra bitar, fyra tillståndsvariabler,  $2^4 = 16$  tillstånd och tillståndsdiagrammet i figuren ovan har ett för en räknare karakteristiskt utseende – tillstånden ligger i en ring. Denna räknare har ingen insignal förutom klocksignalen. För varje klockpuls går den till ett nytt tillstånd. Vi använder denna räknare för att i det följande belysa några olika problem.

### Realisering av räknaren med D-vippor

Booleska uttryck för nästa tillstånd  $q^+$



Figur 5.44 Karnaughdiagram för  $q^+$ .

Booleska uttrycken för  $q^+$  blir

$$q_0^+ = q_0'$$

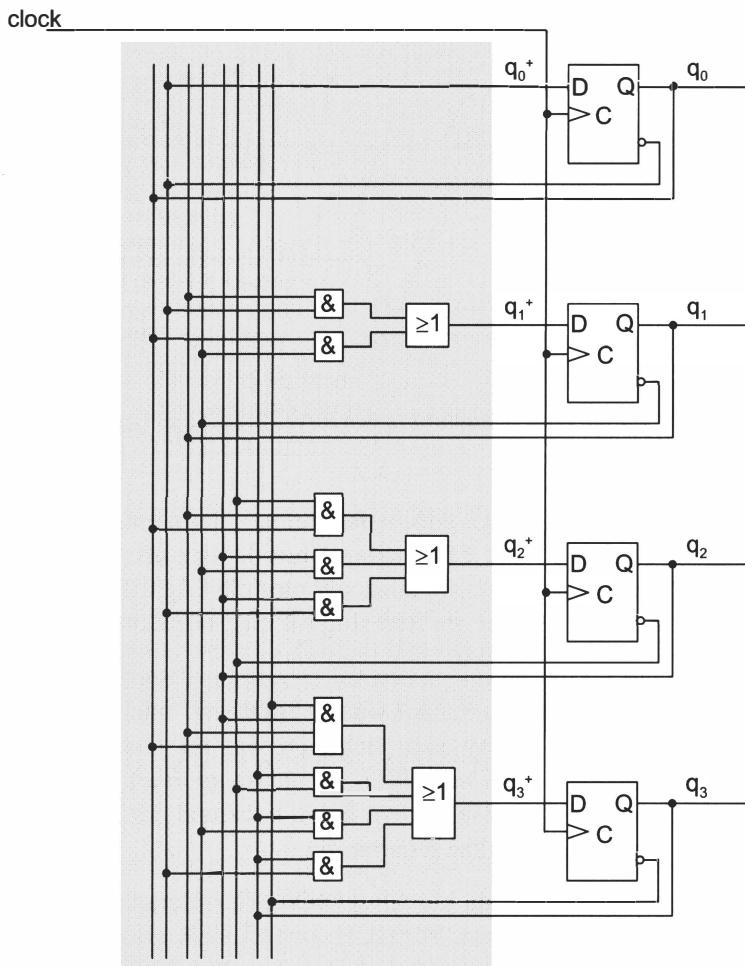
$$q_1^+ = q_1' q_0 + q_1 q_0'$$

$$q_2^+ = q_2' q_1 q_0 + q_2 q_1' + q_2 q_0'$$

$$q_3^+ = q_3' q_2 q_1 q_0 + q_3 q_2' + q_3 q_1' + q_3 q_0'$$

$q^+$ -funktionerna ovan har en regelbunden uppbyggnad. Studera tillståndstabellen i figur 5.43 ovan och identifiera principen för hur  $q^+$ -funktionerna konstrueras.

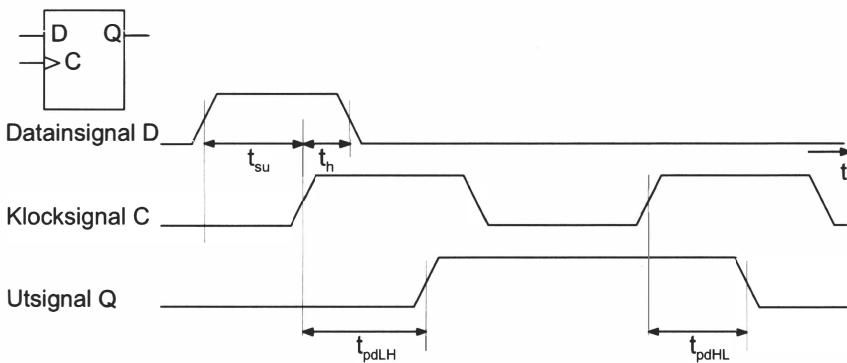
### Kretsschema för räknaren



Figur 5.45 Kretsschema med D-vippor för binärräknaren modulo-16 med 4 bitar.

## Beräkning av maximal klockfrekvens för räknaren

Maximal klockfrekvens bestäms främst av födröjningen i D-vippan och grindnätet som genererar nästa tillstånd. Det finns dock för D-vippan även några andra tidsparametrar än födröjningen som måste beaktas. I figuren nedan visas viktiga tidsparametrar för en D-vippa.

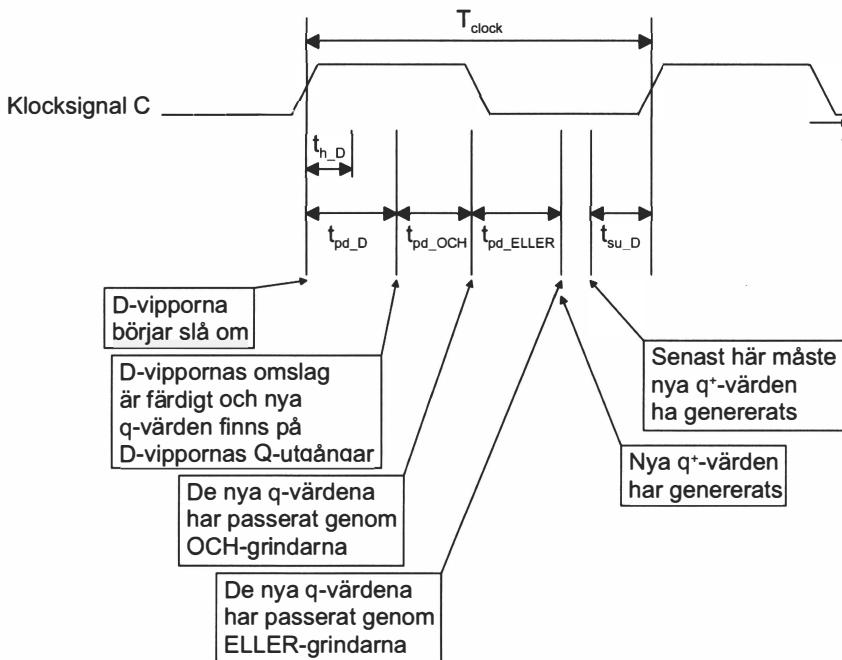


Figur 5.46 Inställningstid  $t_{su}$ , hålltid  $t_h$  och födröjning  $t_{pdLH}$ ,  $t_{pdHL}$  för en D-vippa.

Födröjningen  $t_{pd}$  (eng. *propagation delay time*), definieras som tiden från klocksignalens aktiva flank till omslaget hos D-vippans utsignal. Ibland anges födröjningen dels då utsignalen slår om från Låg till Hög, betecknad  $t_{pdLH}$ , dels då utsignalen slår om från Hög till Låg, betecknad  $t_{pdHL}$ .

Det är självklart viktigt att insignalen till D-vippans dataingång D är stabil under klocksignalens aktiva flank. Detta är emellertid inte tillräckligt, utan insignalen kan också behöva vara stabil en viss tid före klocksignalens aktiva flank, benämnd *inställningstid*  $t_{su}$  (eng. *set-up time*) och hållas stabil en viss tid efter klocksignalens aktiva flank, benämnd *hålltid*  $t_h$  (eng. *hold time*). Båda dessa tider är alltså *minimitider*.

Låt oss nu beräkna maximala klockfrekvensen för räknaren, utgående från kretsschemat i figur 5.45 ovan. För OCH- och ELLER-grindarna betecknas födröjningarna  $t_{pd\_OCH}$  respektive  $t_{pd\_ELLER}$ . För D-vippan betecknas födröjning, inställningstid och hålltid  $t_{pd\_D}$ ,  $t_{su\_D}$  respektive  $t_{h\_D}$ .



Figur 5.47 Tidsdiagram för ett omslag hos räknaren.

Hålltiden  $t_{h\_D}$  i diagrammet ovan behöver inte beaktas vid beräkning av maximala klockfrekvensen. Ur diagrammet fås att

$$T_{clock} \geq t_{pd\_D} + t_{pd\_OCH} + t_{pd\_ELLER} + t_{su\_D}$$

$$f_{clock} = 1 / T_{clock} \leq 1 / (t_{pd\_D} + t_{pd\_OCH} + t_{pd\_ELLER} + t_{su\_D})$$

Antag som exempel  $t_{pd\_D} = 0,7$  ns,  $t_{su\_D} = 0,2$  ns,  $t_{h\_D} = 0$  ns och  $t_{pd\_OCH} = 0,3$  ns,  $t_{pd\_ELLER} = 0,4$  ns.

Då blir enligt ovan

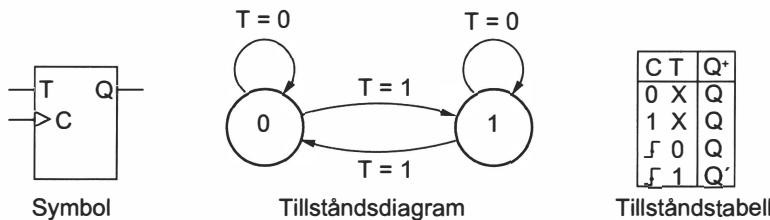
$$T_{clock} \geq 0,7 + 0,3 + 0,4 + 0,2 = 1,6 \text{ ns}$$

$$f_{clock} = 1 / T_{clock} \leq 1 / 1,6 = 0,625 \text{ GHz}$$

## Realisering av räknaren med T-vippor

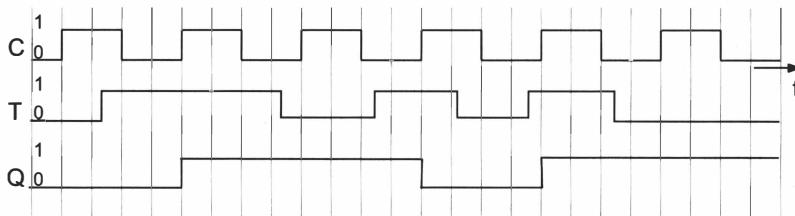
Låt oss gå tillbaka och studera booleska uttrycken för  $q^+$ -funktionerna hos räknaren realiserad med D-vippor. Vi ser där att  $q_0^+$  har en produktterm,  $q_1^+$  två,  $q_2^+$  tre och  $q_3^+$  fyra produkttermer. Generellt gäller för en binärräknare modulo- $2^n$  med n bitar att antalet produkttermer hos  $q_i^+$ -funktionerna blir lika med  $(i+1)$  och sålunda att MSB  $q_{n-1}^+$  har n produkttermer. Vid realisering av räknare med många bitar blir det stora antalet produkttermer ett problem. Ett sätt att lösa problemet är att dela upp räknaren i ett antal räknare med färre antal bitar och kaskadkoppla dem, något som kommer att visas längre fram. Ett annat sätt är att realisera räknaren med T-vippor, som har ett beteende speciellt lämpat för realisering av räknare, som vi nu skall se. Vi börjar med en genomgång av T-vippan.

### T-vippan



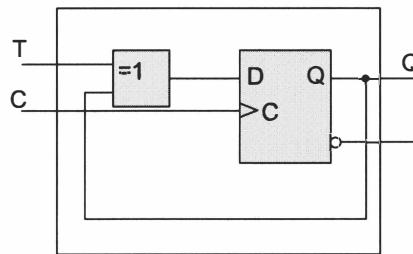
Figur 5.48 T-vippan beskriven i ett tillståndsdiaagram och en tillståndstabell.

T-vippan fungerar så att  $T = 0$  ger ingen tillståndsändring, "står kvar", medan  $T = 1$  ger tillståndsändring, "slår om". Bokstaven T i namnet kommer från det engelska ordet "*toggle*", som betyder "slå om, ändra tillstånd" och som förekommer i ordet "*toggle switch*", vippomkopplare. I figuren nedan visas ett tidsdiagram för T-vippan.



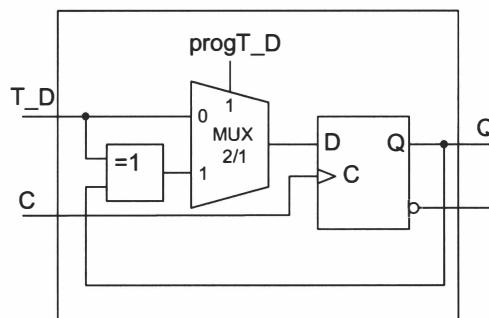
Figur 5.49 T-vippans funktion i tidsplanet.

T-vippan kan konstrueras med en D-vippa och en XOR-grind enligt figuren nedan (se också övningsuppgift 2.15 i kapitel 2). För  $T = 0$  kommer värdet på D-vippans utgång att återmatas till D-ingången och ingen tillståndsändring sker, medan för  $T = 1$  kommer värdet på D-vippans utgång att invereras i XOR-grinden och D bli  $Q'$ , som ger tillståndsändring.



Figur 5.50 T-vippa realiseras med en D-vippa och en XOR-grind.

I PLD är det ofta möjligt att programmera vipporna till D- eller T-vippor, just för att kunna realisera räknare med T-vippor. Kopplingsprincipen visas i figuren nedan, där typ av vippa väljs med signalen  $\text{progT\_D}$  som kan programmeras till 0 eller 1. Med  $\text{progT\_D} = 0$  väljs kanal 0 på MUX:en och den programmerbara vippans insignal  $T_D$  går då direkt via MUX:en in i D-vippan medförande att den programmerbara vippan blir en D-vippa. Med  $\text{progT\_D} = 1$  väljs kanal 1 på MUX:en och XOR-grindens utsignal går via MUX:en in i D-vippan medförande att vippan blir en T-vippa.



Figur 5.51 Vippa programmerbar som D- eller T-vippa.

Låt oss nu realisera räknaren med T-vippor. För D-vippan gäller att  $D = q^+$  och när vi realiserade räknaren med D-vippor så bestämde vi minimala booleska uttryck för nästa tillstånd  $q^+$  som realiseras i grindnät vars utsignal anslöts som insignal till D-vipporna. För T-vippan gäller inte att T är lika med  $q^+$ , utan i stället gäller att  $T = q^+ \oplus q$ . Nästa tillstånd kommer inte att finnas explicit i kretsen såsom fallet är i sekvenskretsmodellen där tillståndsregistret är realisrat med D-vippor. Grindnäten som tidigare generade nästa tillstånd skall nu i stället generera T-signaler. Vi bestämmer nu minimala booleska uttryck för T-signalerna.

### Booleska uttryck för T-signaler

		$q_1 q_0$				
		00	01	11	10	
		00	0	0	0	0
		01	0	0	1	0
		11	0	0	1	0
		10	0	0	0	0
		$T_3$				

		$q_3 q_2$				
		00	01	11	10	
		00	0	0	1	0
		01	0	0	0	0
		11	0	0	0	0
		10	0	0	0	0
		$T_2$				

		$q_1 q_0$				
		00	01	11	10	
		00	0	1	1	0
		01	1	1	0	0
		11	1	1	0	0
		10	1	1	0	0
		$T_1$				

		$q_3 q_2$				
		00	01	11	10	
		00	1	1	1	1
		01	1	1	1	1
		11	1	1	1	1
		10	1	1	1	1
		$T_0$				

Figur 5.52 Karnaughdiagram för T-signaler.

Booleska uttrycken för T-signalerna blir

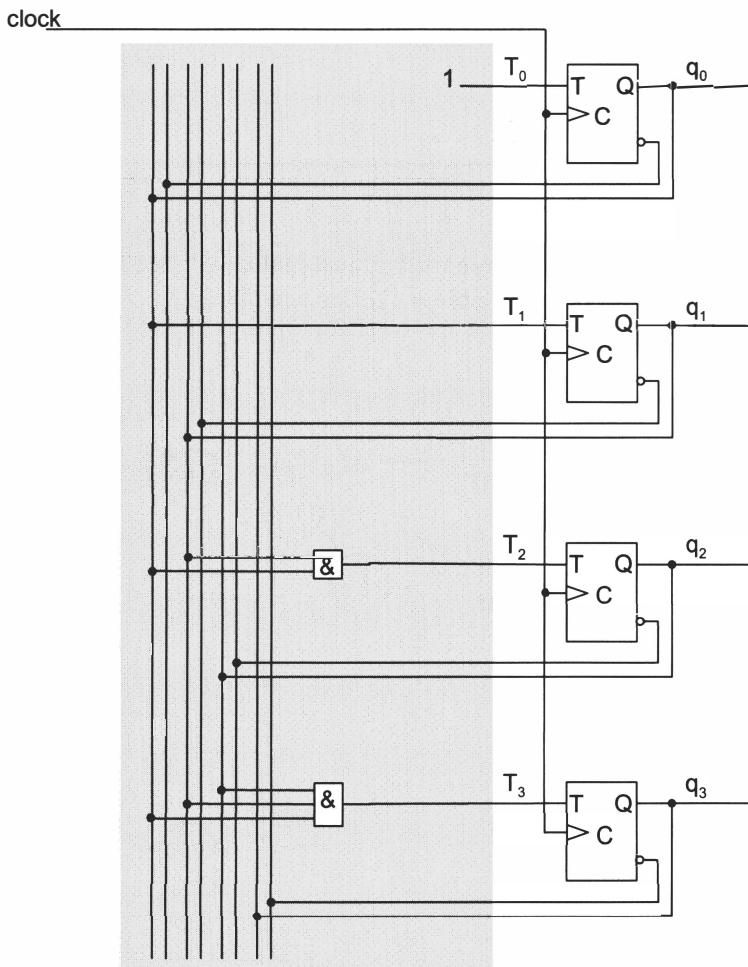
$$T_0 = 1$$

$$T_1 = q_0$$

$$T_2 = q_1 q_0$$

$$T_3 = q_2 q_1 q_0$$

### Kretsschema för räknaren



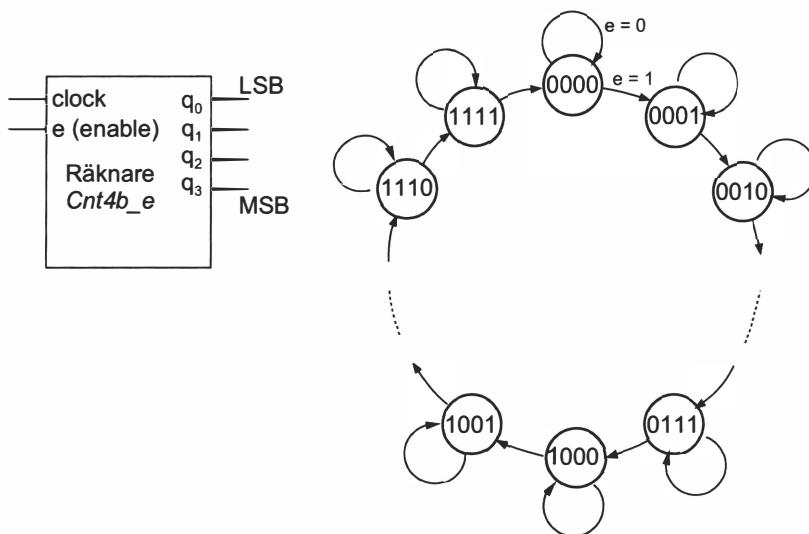
Figur 5.53 Kretsschema med T-vippor för binärräknaren modulo-16 med 4 bitar.

Vi konstaterar att komplexiteten är avsevärt mindre vid realisering av räknaren med T-vippor jämfört med realisering med D-vippor. Booleska uttrycken för T-funktionerna innehåller endast *en* produktterm. Beträffande booleska uttrycken så hade vi kunnat skriva upp dem direkt utgående från räknarens tillståndstabell utan att gå via Karnaughdiagram. I tillståndstabell-

len ser vi att  $q_0$  alltid skall ”slå om” vilket ger  $T_0 = 1$ ,  $q_1$  skall ”slå om” när  $q_0 = 1$  vilket ger  $T_1 = q_0$ ,  $q_2$  skall ”slå om” när  $(q_1 = 1)$  och  $(q_0 = 1)$  vilket ger  $T_2 = q_1 q_0$ ,  $q_3$  skall ”slå om” när  $(q_2 = 1)$  och  $(q_1 = 1)$  och  $(q_0 = 1)$  vilket ger  $T_2 = q_2 q_1 q_0$ , osv, principen gäller vid utbyggnad av räknaren till godtyckligt antal bitar.

### Räknare med *enable*

Vi bygger nu ut binärräknaren med en insignal *enable* med vilken man styr om räknaren skall räkna (*enable* = 1) eller inte räkna (*enable* = 0).



Figur 5.54 Binärräknare modulo-16 med 4 bitar och med en insignal enable.

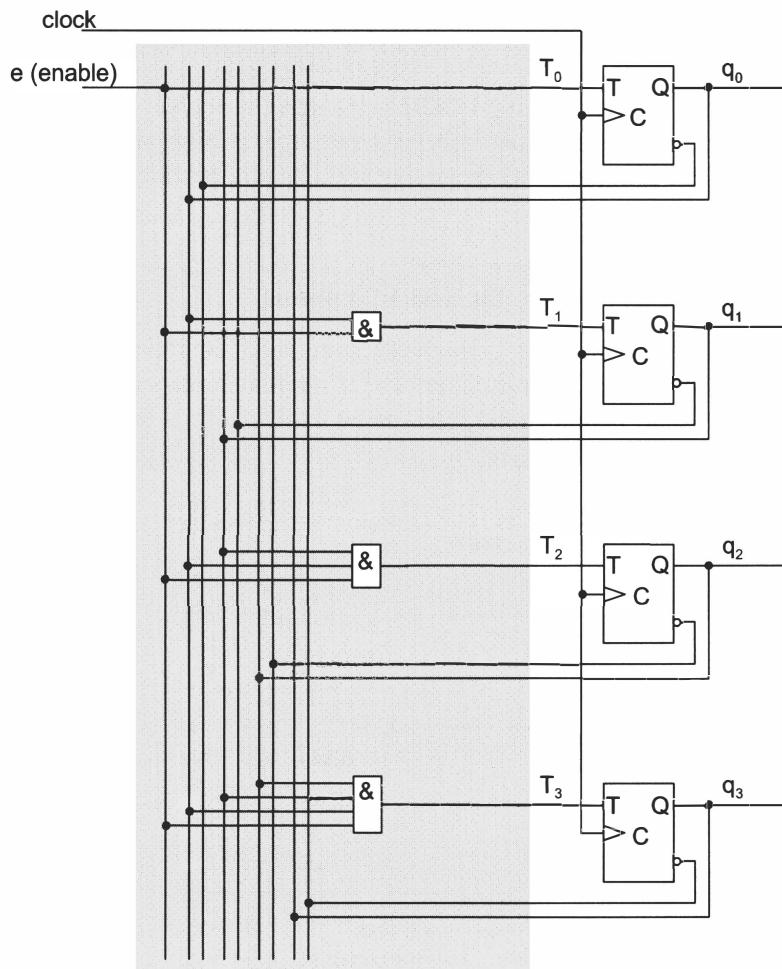
Det är enkelt att bygga ut räknaren, realiseras med T-vippor, med *enable* (*e*). Orsaken är T-vippans funktion. För *e* = 0 skall räknaren inte räkna, innebärande  $T = 0$  hos samtliga vippor, medan för *e* = 1 skall räknaren räkna, innebärande oförändrade T-signaler. De nya T-signaler kan sålunda bildas genom en OCH-operation mellan *enable* (*e*) och de ursprungliga T-signalerna enligt nedan.

$$T_0 = e$$

$$T_1 = eq_0$$

$$T_2 = eq_1q_0$$

$$T_3 = eq_2q_1q_0$$



Figur 5.55 Kretsschema för binärräknaren modulo-16 med enable ( $e$ ).

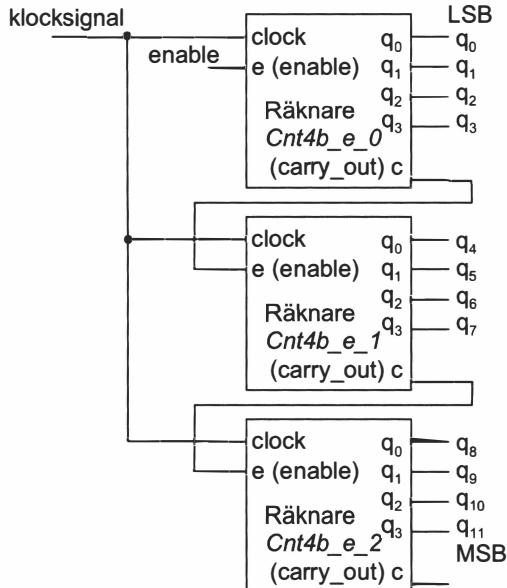
Det är mer komplicerat att bygga ut räknaren realiseras med D-vippor med *enable* (*e*). Betecknar vi de tidigare  $q^+$ -funktionerna utan enable med  $q^+_{old}$  så blir booleska uttrycken för de nya  $q^+$ -funktionerna med enable

$$q^+ = eq^+_{old} + e'q$$

För  $e = 1$  blir booleska uttrycket ovan  $q^+ = q^+_{old}$  och räknaren räknar enligt tidigare. För  $e = 0$  blir  $q^+ = q$  och det aktuella  $q$ -värdet återmatas till D-ingången och räknaren räknar inte. Utbyggnaden av räknaren med enable sker således genom utökning av OCH-ELLER-nätten som genererar D-signaler, dels med en extra ingång i OCH-grindarna för *enable* (första produkten i booleska uttrycket ovan), dels med en extra OCH-grind och extra ingång i ELLER-grinden för återmatning av  $q$  (andra produkten i booleska uttrycket ovan).

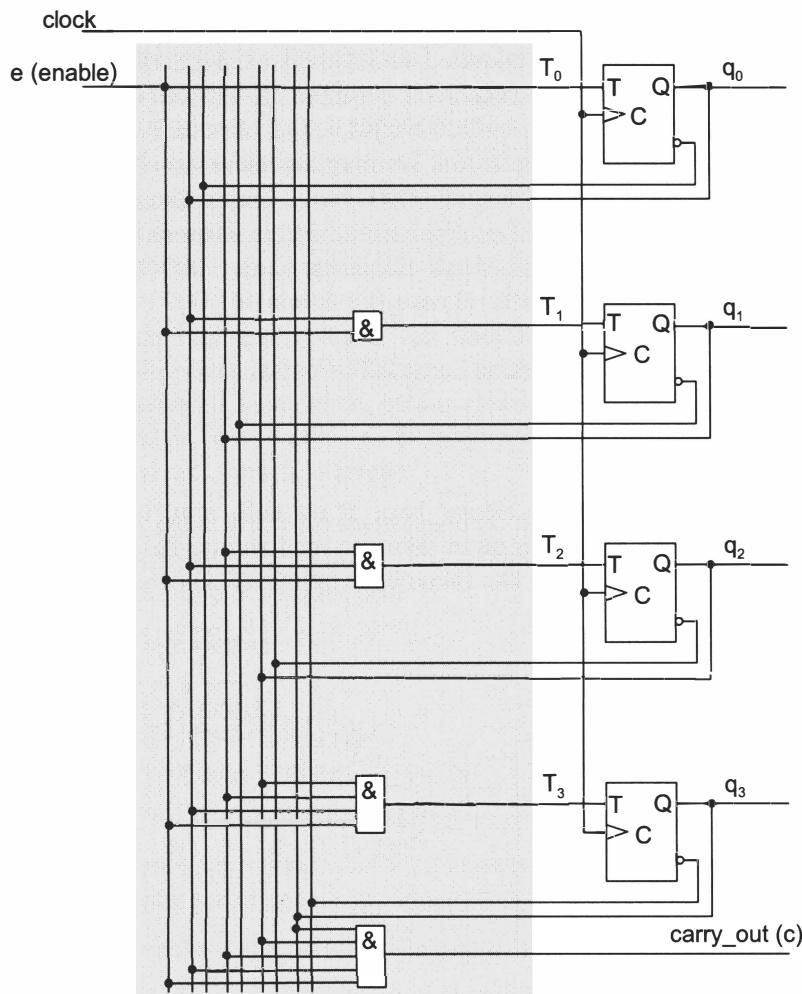
### Räknare med *carry\_out* (*c*) för kaskadkoppling

I samband med realisering av binärräknaren med D-vippor tidigare och konstaterandet att antalet produkttermer i  $q^+$ -funktionerna växer dramatiskt med antalet bitar, nämndes möjligheten att kaskadkoppla räknare med ett färre antal bitar.



Figur 5.56 Kaskadkoppling av tre binärräknare 4 bitar till en räknare 12 bitar.

Principen visas i figuren ovan där en binärräknare modulo-4096, med 12 bitar byggs upp med tre kaskadkopplade binärräknare modulo-16, 4 bitar. I figuren nedan visas hur utsignalen *carry\_out (c)* genereras.



Figur 5.57 Binärräknare med en utsignal *carry\_out (c)* för kaskadkoppling.

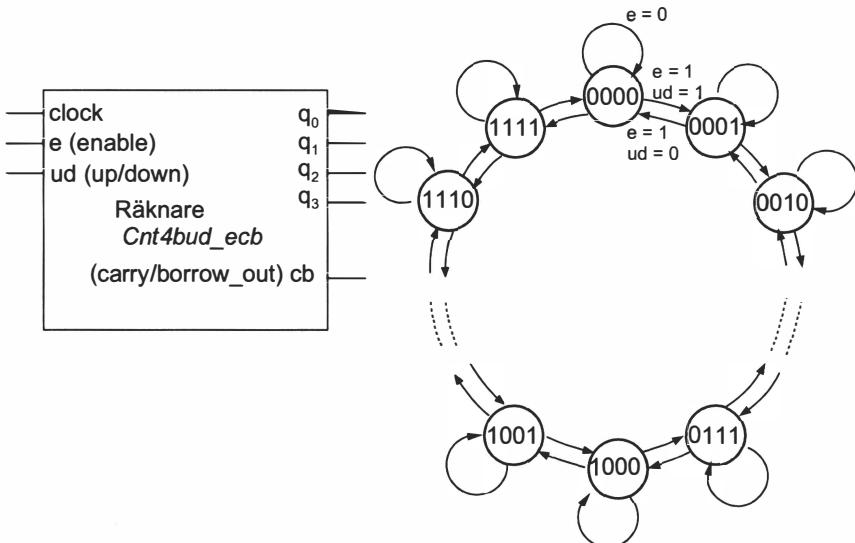
Utsignalen *carry\_out* (*c*) är bildad som

$$c = \text{eq}_3\text{q}_2\text{q}_1\text{q}_0$$

innebärande att  $c = 1$  om och endast om  $e = 1$  och räknaren har sitt maxvärde  $\text{q}_3\text{q}_2\text{q}_1\text{q}_0 = 1111$ . Antag att de tre kaskadkopplade räknarna i figur 5.56 är nollställda och att enable = 1 till räknare 0. Endast räknare 0 kommer att räkna, eftersom räknare 1 och 2 har  $e = 0$  p.g.a. räknare 0 och 1 inte har sitt maxvärde. När räknare 0 kommer till sitt maxvärde och 12-bitars räknaren alltså har värdet 000000001111 så blir  $c = 1$  från räknare 0 och sålunda  $e = 1$  till räknare 1 som kommer att räkna vid nästa klockpuls samtidigt som räknare 0 slår om till 0000 och 12-bitars räknaren får värdet 000000010000. Räknare 1 räknar sedan inte förrän räknare 0 kommer till sitt maxvärde 1111 och 12-bitars räknaren värdet 000000011111, då räknare 1 räknar ett steg och räknare 0 slår om till 0000 och 12-bitars räknaren får värdet 000000100000, osv. Det är viktigt att  $e$  ingår i uttrycket för *carry\_out*, eftersom räknarna bara skall räkna om  $e = 1$  till räknare 0.

### Upp/ned-räknare

De hittills visade räknarna räknar bara åt ett håll, *upp*, medan i tillståndsdiagrammet. Vi bygger nu ut räknaren med en insignal *up/down* (*ud*) som styr räknaren så att den kan räkna både upp (*ud* = 1) och ned (*ud* = 0).



Figur 5.58 Upp/ned-räknare.

En upp/ned-räknare kan enkelt konstrueras utgående från den tidigare konstruerade binärräknaren. Om vi vandrar med sols i tillståndsdiagrammet till binärräknaren i figur 5.58 och betraktar inverserna,  $q'$ -signalerna, så ser vi en nedräkning i dessa. Nu skall vi ju ha nedräkning i  $q$ -signalerna, men det kan vi alltså åstadkomma genom att räkna upp i  $q'$ -signalerna! Om vi byter ut  $q$ -signalerna som är anslutna till OCH-grindarna i räknaren i figur 5.53 mot  $q'$ -signalerna, så får vi en räknare som räknar ned i  $q$ -signalerna. Nu vill vi ha en räknare som både kan räkna upp och ned och som skall styras med signalen up/down (ud). I räknaren måste då finnas de ursprungliga OCH-grindarna till vilka  $q$ -signalerna är anslutna, som skall ge T-signaler för uppräkning, och de nya OCH-grindarna till vilka  $q'$ -signalerna är anslutna, som skall ge T-signaler för nedräkning. Val av T-signaler skall ske med signalen up/down (ud) och skulle kunna göras med MUX 2-1 anslutna till T-ingångarna. För att inte i onöдан öka gränddjupet i kombinationskretsarna som genererar T-signaler och därmed minska maximala klockfrekvensen, så splittrar vi upp MUX:arna så att multiplexerns ELLER-grind bibrålls ansluten till T-ingången, medan multiplexerns OCH-grindar sammansmälts med de redan befintliga OCH-grindarna. Kretsschemat för upp/ned-räknaren visas i figur 5.59 nedan.

I stället för att utöka OCH-grindarna med en ingång för signalen up/down (ud) så har enable (e) och up/down (ud) grändats samman så att det blir en enable-signal för upp och en för ned.

T-signaler är genererade enligt

$$T_0 = e$$

$$T_1 = (e(ud))q_0 + (e(ud))'q_0'$$

$$T_2 = (e(ud))q_1q_0 + (e(ud))'q_1'q_0'$$

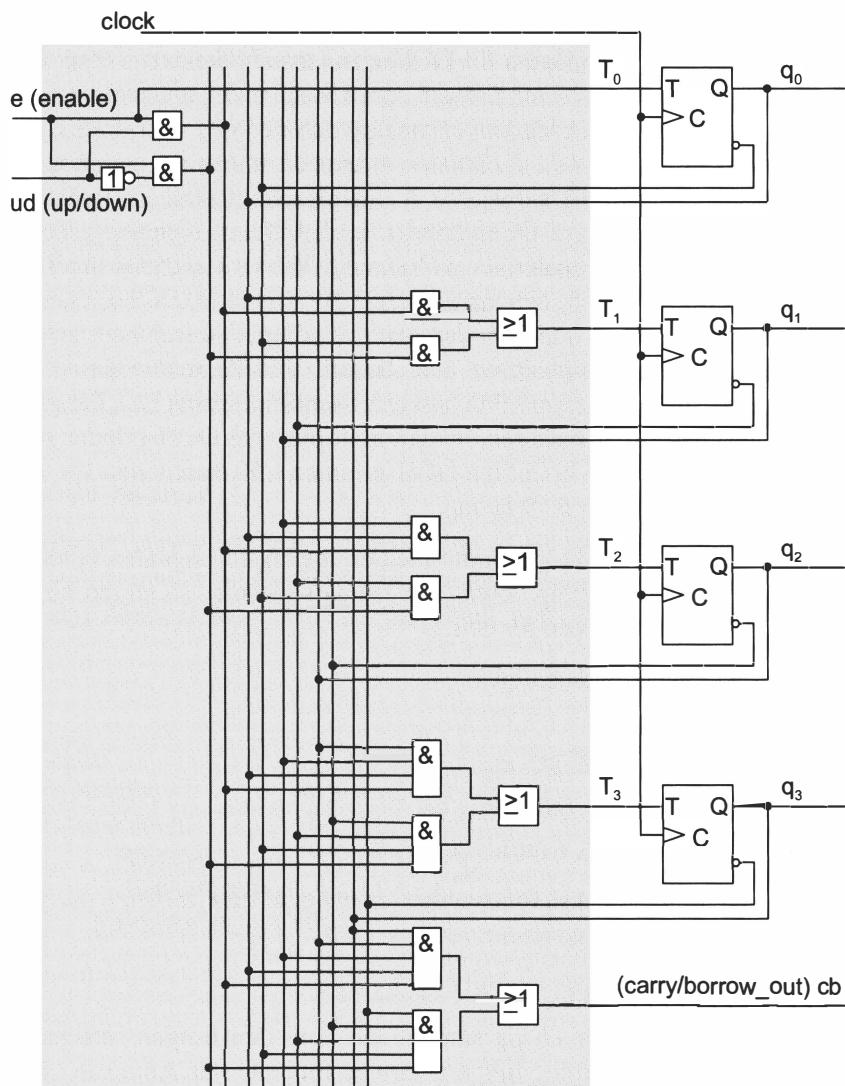
$$T_3 = (e(ud))q_2q_1q_0 + (e(ud))'q_2'q_1'q_0'$$

Upp/ned-räknaren har också försetts med en utsignal *carry/borrow\_out (cb)* för kaskadkoppling som genererats som

$$cb = (e(ud))q_3q_2q_1q_0 + (e(ud))'q_3'q_2'q_1'q_0'$$

Vid uppräkning fungerar *cb* på samma sätt som den tidigare utsignalen *carry\_out (c)*. Vid nedräkning fungerar den på följande sätt. Antag att vi har tre kaskadkopplade upp/ned-räknare och att 12-bitarsräknaren befinner sig i tillståndet 010011000000 och att det är nedräkning. Eftersom räknare 0 har värdet 0000 blir *cb* = 1, vilket ger räknare 1 enable = 1 och den räknar ned

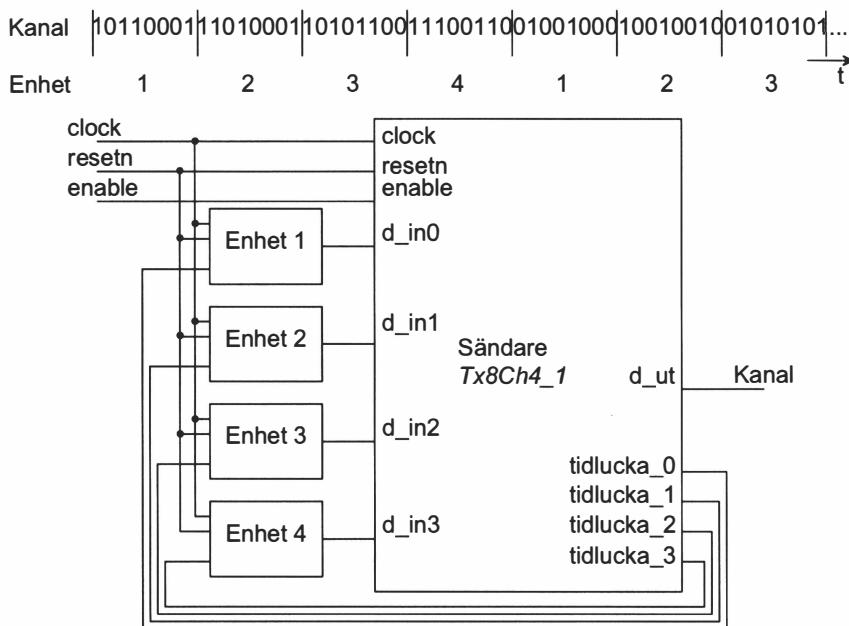
ett steg samtidigt som räknare 0 räknar ned till 1111 och 12-bitars räknaren får värdet 010010111111.



Figur 5.59 Kretsschema upp/ned-räknare.

**Exempel 5.4**

Fyra enheter skall sända 8-bitars ord i serieform på en gemensam kanal. Orden skall sändas direkt efter varandra i tur och ordning i var sin *tidlucka* (eng. *timeslot*) omfattande åtta klocksignalintervall enligt figuren nedan. En sändare *Tx8Ch4\_1* skall i tur och ordning låta enheterna sända sina ord i respektive tidlucka och skicka ut dem på kanalen. Utsignalerna *tidlucka*, aktiv höga, ger information till respektive enhet när den får sända sitt 8-bitars ord. Konstruera sändaren med standardkomponenter.

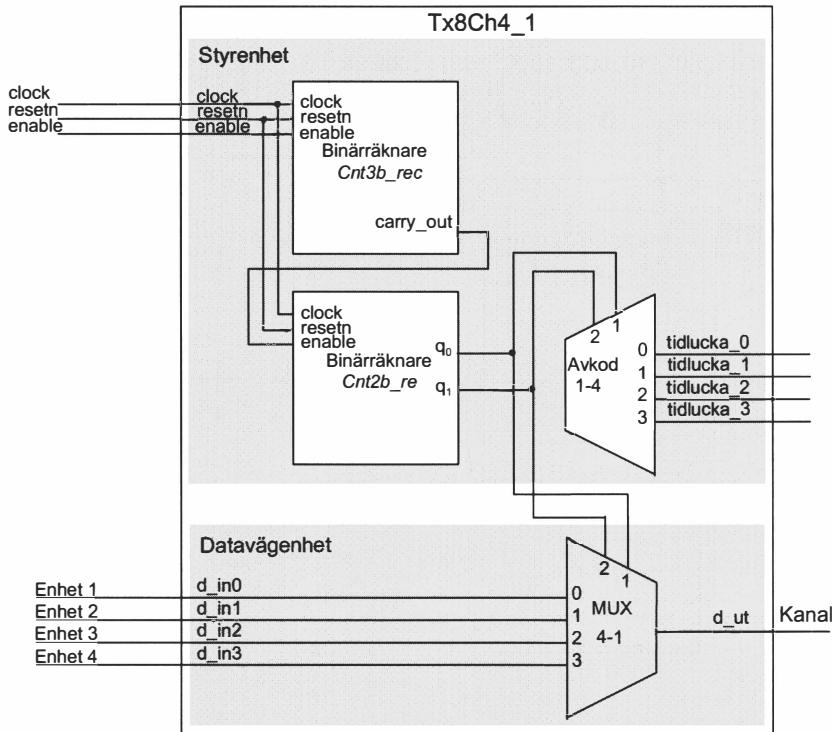


*Figur 5.60 Sändare för sändning av 8-bitars ord på en gemensam kanal.*

**Lösning**

En MUX4\_1 används för att bilda en dataväg mellan respektive enhet och den gemensamma kanalen. MUX:en adresseras med en räknare modulo-4 *Cnt2b\_re*, som räknar 00, 01, 10, 11, 00, 01... . Denna räknare skall räkna ett steg var åttonde klockpuls. För att styra denna räknare används en

räknare modulo-8 *Cnt3b\_rec*, som var åttonde klockpuls genererar en *carry\_out* som används som *enable* till räknaren *Cnt2b\_re*. Utsignalerna *tidlucka* genereras med en avkodare *Avkod1\_4*, som adresseras av räknaren *Cnt2b\_re*. Schemat för sändaren visas nedan. Notera hur sändaren är uppdelad i två enheter, en *styrenhet* och en *datavägenhet*. Styrenheten består av räknarna och avkodaren. Datavägenheten består av multiplexern. Styrenheten styr datavägenheten och ger även styrsignaler till enheterna 1-4.

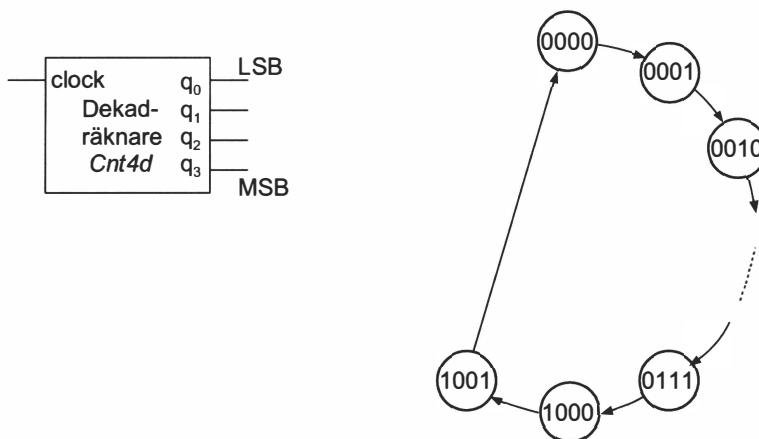


Figur 5.61 Blockschema för sändaren *Tx8Ch4\_1*.



## Dekadräknare

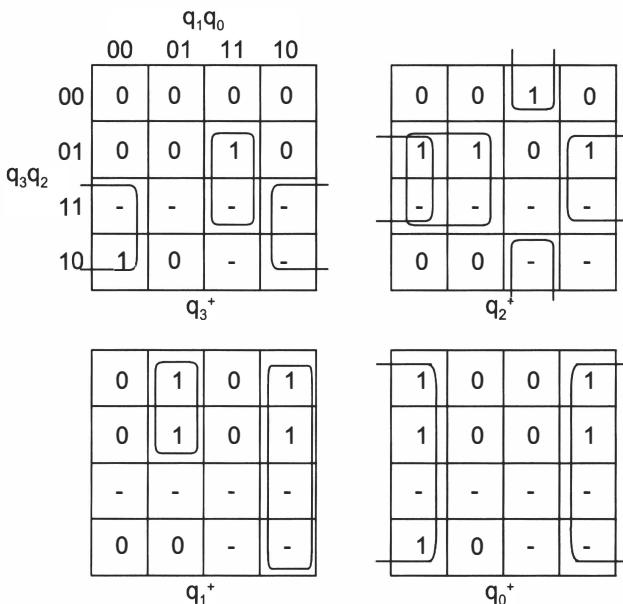
Vi har hittills studerat n-bitars räknare som genomlöpt samtliga tillstånd i en cirkel och räknat modulo- $2^n$ . En annan klass av intressanta räknare är räknare modulo-10. Sådana räknare benämnes *dekadräknare* och räknar 0, 1, 2,..., 9, 0, 1... En dekadräknare måste uppenbart ha minst 4 tillståndsvariabler. Naturligast är att låta en dekadräknare räkna i BCD-kod, innebärande att räknecykeln innehåller de 10 första tillstånden i den tidigare behandlade 4-bitars binärräknaren. Tillståndsdiagrammet blir enligt figuren nedan.



Figur 5.62 Tillståndsdiagram för en dekadräknare som räknar i BCD-kod.

Dekadräknaren med 4 tillståndsvariabler har totalt  $2^4 = 16$  tillstånd och det finns sålunda 6 icke använda tillstånd 1010, 1011, 1100, 1101, 1110 och 1111, som måste beaktas vid realiseringen. Vi väljer här att sätta nästa tillstånd för de ospecifierade tillstånden till don't care. Efter bestämning av de minimala  $q^+$ -funktionerna kan man sedan avgöra om tillståndsövergångarna för de ospecifierade tillstånden blivit acceptabla. Ett kriterium för de ospecifierade tillstånden kan vara att de alltid skall leda in i den normala räknecykeln.

Karnaughdiagrammen för  $q^+$ -funktionerna blir enligt figuren nedan.

Figur 5.63 Karnaughdiagram för  $q^+$ -funktionerna till dekadräknaren.

De minimala booleska uttrycken för  $q^+$ -funktionerna blir

$$q_0^+ = q_0'$$

$$q_2^+ = q_3' q_1' q_0 + q_1 q_0'$$

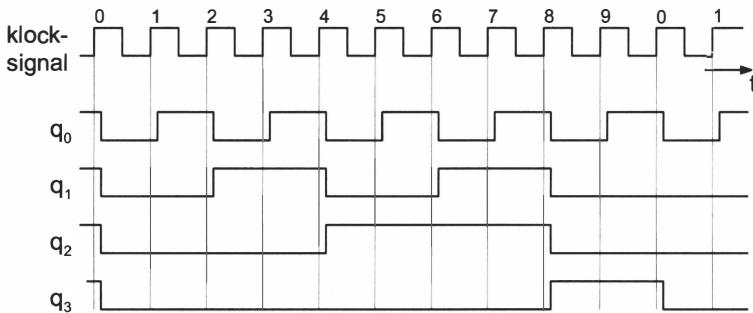
$$q_2^+ = q_2' q_1 q_0 + q_2 q_1' + q_2 q_0'$$

$$q_3^+ = q_3 q_0' + q_2 q_1 q_0$$

Dekadräknaren realiseras med D-vippor får samma struktur som den tidigare realiseraade binärräknaren modulo-16 i figur 5.45, och vi ritar därför inte upp schemat –  $q^+$ -funktionerna ovan ger tillräcklig information om räknaren.

En användning av dekadräknaren är som frekvensdelare och det kan därför vara intressant att studera räknarens  $q$ -signaler i tidsplanet, som visas i figuren nedan (räknaren har antagits realiseras med vippor som slår om på

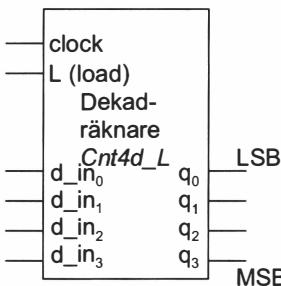
positiv flank). Vi ser där att frekvensen hos utsignalen  $q_3$  är  $f_{\text{klock}}/10$ , och dekadräknaren kan alltså användas för *frekvensdelning med 10*. Nu ser vi emellertid också i signaldiagrammet att utsignalen  $q_3$  inte är symmetrisk, den är Låg under 8 klockpulsperioder och Hög under 2 klockpulsperioder. Orsaken till osymmetrin är naturligtvis kodningen av dekadräknarens 10 tillstånd, som här är gjord i BCD-kod. Om man önskar en symmetrisk signal, så får en annan tillståndskodning användas, t.ex. en kod med positionsvikterna 5-4-2-1, se tabell 1.3 i kapitel 1. Positionen med vikten 5 blir symmetrisk.



Figur 5.64 Tidsdiagram för en dekadräknare som räknar i BCD-kod.

I vissa tillämpningar är det önskvärt att kunna *förinställa*, *ladda*, en räknare med ett godtyckligt starttillstånd i räknerekvensen och sedan räkna från detta tillstånd. En  $n$ -bitars räknare förses då med  $n$  ingångar på vilka det aktuella tillståndet appliceras och med en speciell signal  $L$ , *Ladda* (eng. *Load*) laddas sedan vipporna med detta tillstånd.

I figuren nedan visas blockschemat för en 4-bitars dekadräknare med förinställning. Med insignalen  $L = 1$  tilldelas D-vipporna värdet på ingångarna  $d_{\text{in}}$ . Med insignalen  $L = 0$  påverkar inte ingångarna  $d_{\text{in}}$  räknaren. Laddningen kan realiseras som asynkron eller synkron beroende på önskemål och vad tillgängliga vippor medger. Asynkron laddning förutsätter att vipporna är försedda med asynkrona insignaler *Preset* och *Clear*. Synkron laddning verkställs av klockpulsen på klockpulsens aktiva flank om insignalen  $L = 1$ . Låt oss se hur dekadräknaren ovan kan förses med synkron laddning.



Figur 5.65 Dekadräknare med förinställning, laddning, av starttillstånd.

Laddningen kan realiseras med en multiplexer MUX 2-1 ansluten till D-ingången hos vardera D-vippa. Insignalen L ansluts som adress till multiplexern, dekadräknarens ursprungliga D-funktion enligt ovan som insignal till multiplexerns ingång 0 och insignalen d\_in som insignal till multiplexerns ingång 1. De nya D-funktionerna med laddning, betecknade D<sub>mL</sub>, skrivna som funktion av de ursprungliga D-funktionerna (D), laddningsignalen (L) och laddningsvärdena (d\_in) blir

$$D_{mL} = L'D + L(d_{in})$$

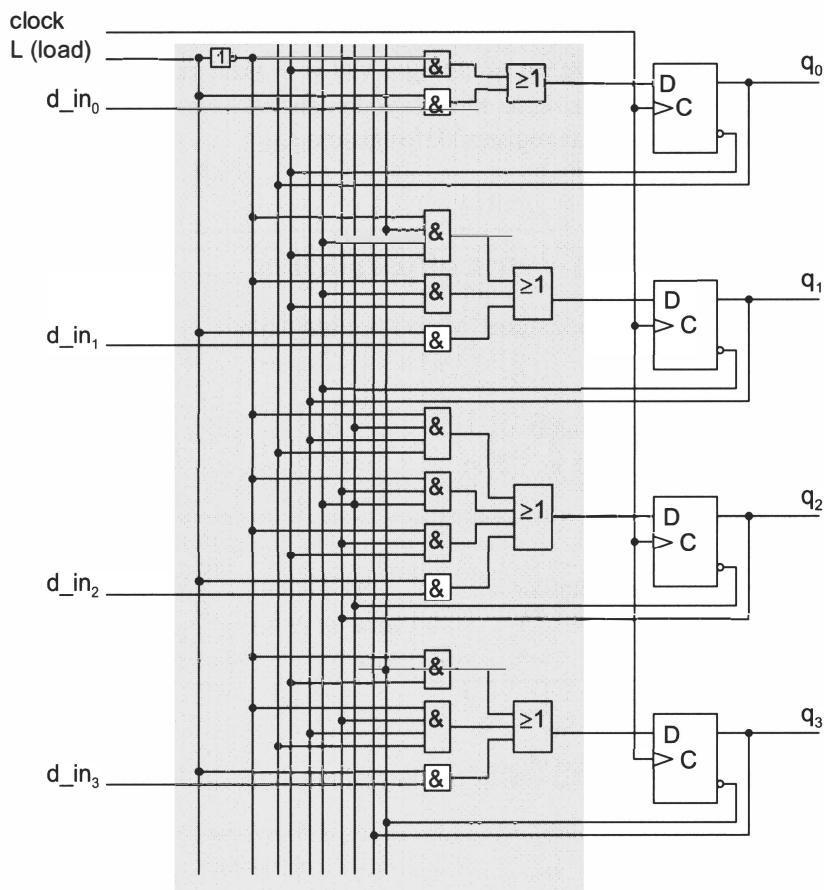
För att inte öka grinddjupet och därmed minska räknarens maximala klockfrekvens, så splittrar vi upp multiplexern och låter den sammansmälta med OCH-ELLER-näten för D-funktionerna ovan, motsvarande att vi ”multiplicerar in” L' i D-funktionerna, innebärande att OCH-grindarna får en extra ingång för L' och låter ELLER-grindarna få en extra ingång för funktionen L(d\_in) som realiseras i en extra OCH-grind. De nya D-funktionerna blir då enligt nedan och realiseringen av räknaren enligt figur 5.66.

$$D_{0mL} = L'q_0' + L(d_{in_0})$$

$$D_{1mL} = L'q_3'q_1'q_0 + L'q_1q_0' + L(d_{in_1})$$

$$D_{2mL} = L'q_2'q_1q_0 + L'q_2q_1' + L'q_2q_0' + L(d_{in_2})$$

$$D_{3mL} = L'q_3q_0' + L'q_2q_1q_0 + L(d_{in_3})$$



Figur 5.66 Kretsschema för dekadräknare med förinställning, laddning.

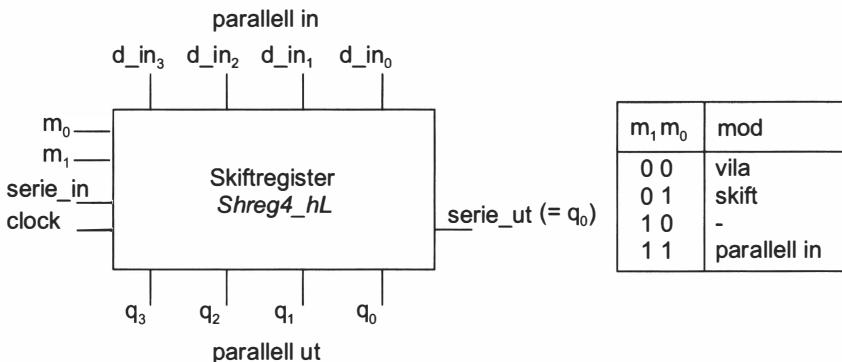
Med denna dekadräknare med förinställning avslutas sektionen om räknare, där vi har studerat räknare modulo- $2^n$  och modulo-10, två viktiga klasser av räknare, och sett hur räknare kan förses med olika faciliteter, såsom enable, carry\_out, up/down, carry/borrow\_out och load. I programmerbar logik har man i princip möjlighet att realisera räknare för vilka modultal som helst och med vilka faciliteter som helst.

## 5.3 Register och skiftregister

Några enkla register och skiftregister visades i kapitel 2. I dessa styrdes parallell inmatning av data och skift enbart med klocksignalen. Ofta vill man ha speciella styrsignaler som anger vad som skall ske, t.ex. laddning eller skift, och endast använda klocksignalen för att verkställa aktiviteten. Nedan visas några sådana register/skiftregister.

### Serie in/parallel in – serie ut/parallel ut

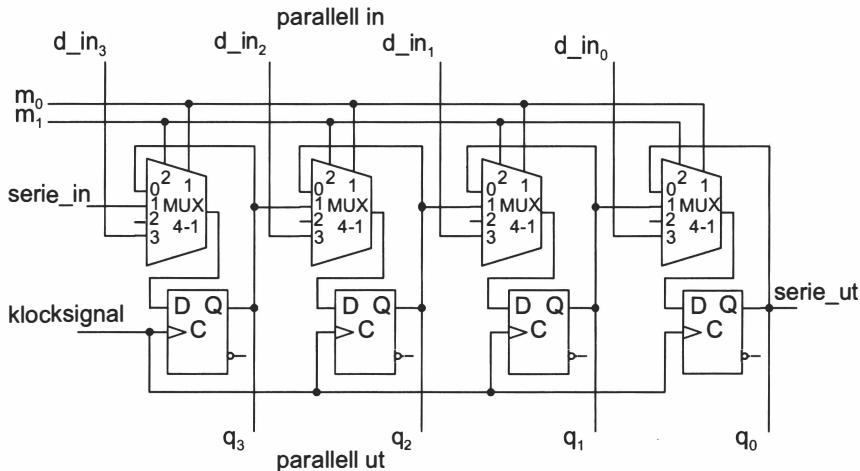
Blockschema och funktionstabell visas i figuren nedan.



Figur 5.67 Blockschema och funktionstabell för skiftregistret *Shreg4\_hL*.

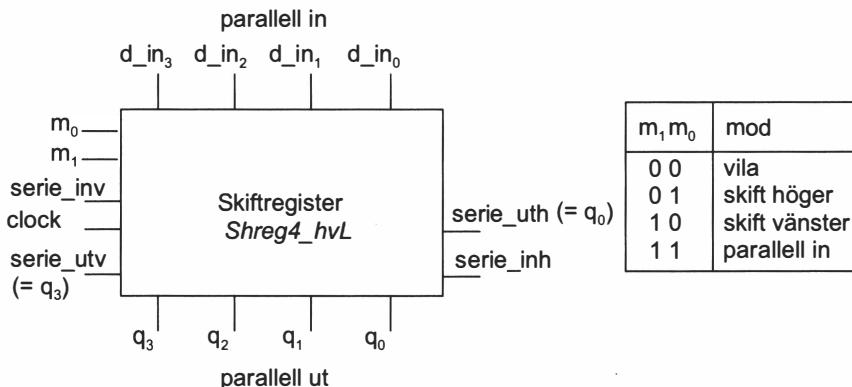
Skiftregistret styrs med två *modsignaler*  $m_1$  och  $m_0$  som bestämmer *mod, arbetssätt*. Vi bygger upp skiftregistret med D-vippor och multiplexrar MUX 4-1 vars adress är modsignalerna  $m_1$  och  $m_0$  och vars utgångar är anslutna till D-vippornas ingångar enligt figuren nedan. För mod *vila*,  $m_1m_0 = 00$ , väljs kanal 0 på multiplexern. Eftersom klocksignalen är kopplad till alla D-vippor, som då alltså klockas för varje klockpuls, så innebär vila att D-vippan skall behålla sitt värde. Detta åstadkoms genom att D-ingången tilldelas värdet hos Q-utgången via anslutning av D-vippans utgång till kanal 0 på multiplexern. För mod *skift*,  $m_1m_0 = 01$ , väljs kanal 1 på multiplexern. Skift innebär skift höger och åstadkoms genom att serie-ingången ansluts till kanal 1 på multiplexern längst till vänster och till

övriga multiplexrars kanal 1 ansluts Q-utgången från D-vippan till vänster. För mod *parallel in*,  $m_1 m_0 = 11$ , väljs kanal 3 på multiplexern. Laddning av D-vipporna med värdet på ingångarna  $d_{\_in}$  åstadkoms genom att ingångarna  $d_{\_in}$  ansluts till kanal 3 på multiplexarna.



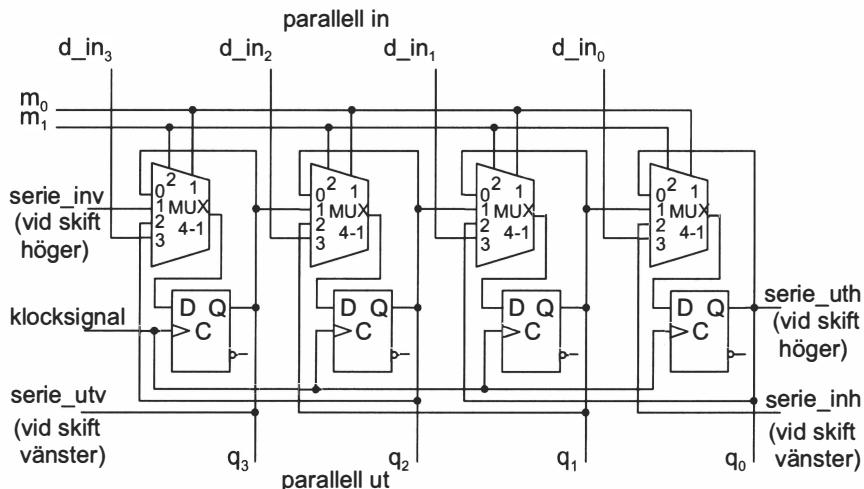
Figur 5.68 Kretsschema för skiftregistret Shreg4\_hL.

## Serie in/parallel in – serie ut/parallel ut – skift höger/skift vänster



Figur 5.69 Blockschema och funktionstabell för skiftregistret Shreg4\_hvL

Vi bygger ut skiftregistret med ytterligare en funktion, möjlighet till skift vänster. Detta kan enkelt ske med hjälp av den icke utnyttjade kanal 2 hos multiplexern och sålunda erhålls mod *skift vänster* för  $m_1 m_0 = 10$ . Insignalen *serie\_inh* ansluts till kanal 2 på multiplexern längst till höger och till övriga multiplexrars kanal 2 ansluts Q-utgången från D-vippan till höger.



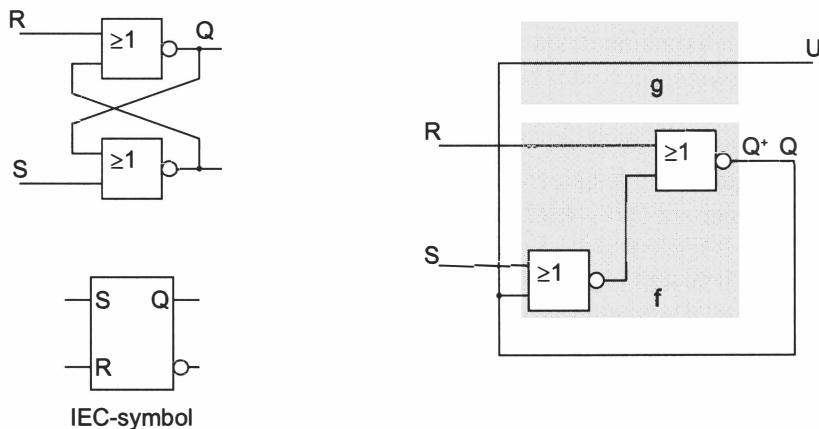
Figur 5.70 Kretsschema för skiftregistret Shreg4\_hvL.

## 5.4 Latchar

Latchar är en viktig klass av sekvenskretsar. De är liksom vippor minneskretsar, men är enklare till sin uppbyggnad och mer primitiva i sin funktion.

### SR-latchen

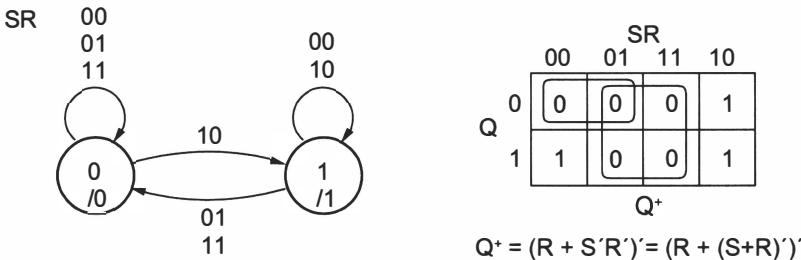
SR-latchen är en *asynkron* sekvenskrets. Den används bl.a. som minneskrets i statiska halvledarminnen. I sådana minneskapslar finns flera miljoner SR-latchar, som vardera lagrar en bit. – Vanligen brukar man se SR-latchens kretsschema ritat som två korskopplade grindar enligt nedan till vänster. SR-latchen är alltså ett grindnät, men till skillnad från de kombinationskretsar vi sett realisera med grindar, så innehåller SR-latchens grindnät återkoppling, utsignalerna från grindarna är återkopplade som insignaler. Till höger visas samma schema omritat så att det skall antyda likhet med sekvenskretsmodellen typ Moore. Det finns som synes inget tillståndsregister mellan  $Q^+$  och  $Q$  och ingen klocksignal. Trots detta blir kretsen en sekvenskrets, en asynkron sekvenskrets, som kan anta två olika tillstånd  $Q = 0$  och  $Q = 1$ . I asynkrona sekvenskretsar kan tillstånd vara instabila beroende på att det inte finns något tillståndsregister som spärrar vägen i återkopplingen. Låt oss analysera beteendet.



Figur 5.71 SR-latch.

Tillståndsdiagram och tillståndstabell visas nedan. Vi ser där hur varje insignal kombination har ett stabilt tillstånd (understrukna i tabellen).

Nuvarande tillstånd Q	Nästa tillstånd $Q^+$			
	Insignaler SR			
	00	01	11	10
0	0	0	0	1
1	1	0	0	1

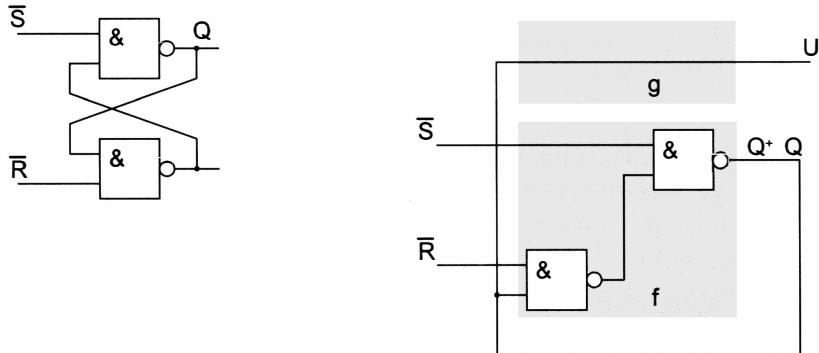


Figur 5.72 Tillståndsdiagram, tillståndstabell och  $Q^+$  för SR-latchen.

För insignal kombinationen  $SR = 00$  befinner sig SR-latchen i vad som kan kallas *viloläge* eller *minnesfas* och stannar kvar i det aktuella tillståndet. Om vi antar att latchen är 0-ställd, dvs  $Q = 0$ , och att  $SR = 00$ , kan latchen 1-ställas genom att vi gör  $S = 1$ , vilket enligt tillståndstabellen medföljer att nästa tillstånd blir 1. Låter vi sedan  $S$  återgå till 0 så förblir latchen 1-ställd och befinner sig åter i viloläge. Om vi nu med latchen 1-ställd, dvs  $Q = 1$ , och  $SR = 00$ , gör  $R = 1$ , så blir enligt tillståndstabellen nästa tillstånd 0 och latchen blir 0-ställd. Låter vi sedan  $R$  återgå till 0 så förblir latchen 0-ställd och befinner sig åter i viloläge.  $S$  är alltså en 1-ställningssignal och  $R$  är en 0-ställningssignal, båda aktivt höga. Signalbeteckningarna  $S$  och  $R$  kommer från de engelska orden *Set* och *Reset*. Ordet *latch* är det engelska ordet för *läs*, och benämningen SR-latch motiveras av att kretsen efter att ha blivit 0-ställd respektive 1-ställd läses fast i det aktuella tillståndet med insignal kombinationen  $SR = 00$ . Insignal kombinationen  $SR = 11$  har vi inte dis-

kuterat och den brukar inte användas, dels därför att då nästa tillstånd bestämmes av den insignal som längst bibehåller värdet 1, dels framförallt därför att de två utgångarna hos SR-latchen då har samma värde 0, till skillnad från fallet för de andra insignalkombinationerna då utgångarna alltid är varandras invers Q och  $\bar{Q}$ .

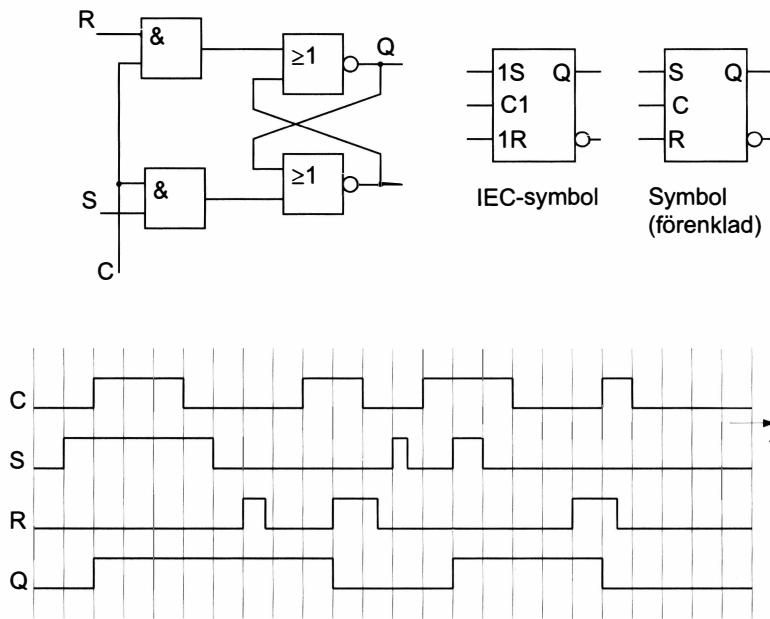
SR-latchen förekommer också i en variant med aktivt låga ingångar och konstrueras då med NAND-grindar enligt figuren nedan. Strecket över S och R hör till själva signalbeteckningen och markerar att signalen är aktivt låg. Denna latch har sitt viloläge för insignalkombinationen 11 och 0-ställs och 1-ställs med insignalen 0. Analysen med tillståndsdiagram och tillståndstabell lämnas till en övningsuppgift.



Figur 5.73  $\bar{S}\bar{R}$ -latch.

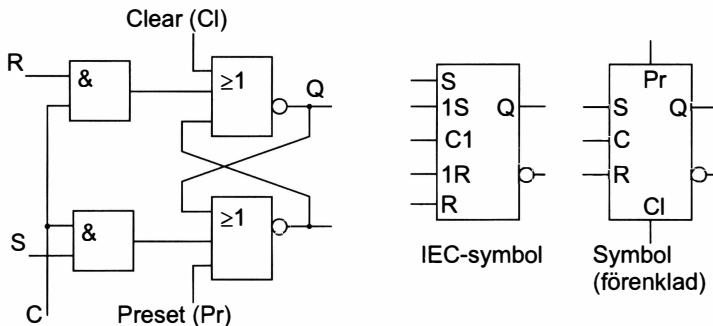
SR-latchen kan förses med en klocksignal C som verkställer tillståndsändringen så man får kontroll över när tillståndsändring skall ske. I figuren nedan visas SR-latchen med aktivt höga ingångar, försedd med en sådan klocksignal. Funktionen är lätt att förstå. För C = 0 är utsignalen från OCH-grindarna lika med 0 och den ursprungliga SR-latchen befinner sig i viloläget, medan för C = 1 är OCH-grindarna öppna och insignalerna S och R kan påverka SR-latchen på vanligt sätt. Där visas också den klockade SR-latchens IEC-symbol. Siffran 1 framför S och R i symbolen markerar att de är beroende av klocksignalen med samma nummer, dvs C1. Vi har för D-vippan inte följt IEC-standarden och satt ut siffror vid D- och klocksignalen och för latcharna väljer vi i fortsättningen av bekvämlighet att heller inte

sätta ut siffrorna, det bör inte leda till missförstånd. Funktionen hos den klockade SR-latchen exemplifieras i ett tidsdiagram.



Figur 5.74 Klockad SR-latch.

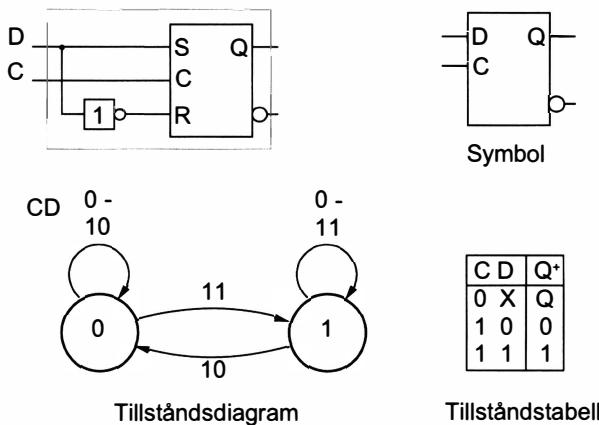
Den klockade SR-latchen kan förses med ingångar med vilka latchen kan 0-ställas och 1-ställas utan medverkan av klocksignalen. En sådan SR-latch visas i figuren nedan, där som synes dessa ingångar ansluts direkt i själva SR-latchen. I IEC-symbolen är dessa insignalerna S och R ej försedda med siffran 1 eftersom de påverkar latchen oberoende av klocksignalen. Som tidigare nämnts för D-vippan så brukar sådana insignaler för 0-ställning och 1-ställning oberoende av klocksignal, på engelska benämns *Clear (Cl)* respektive *Preset (Pr)*.



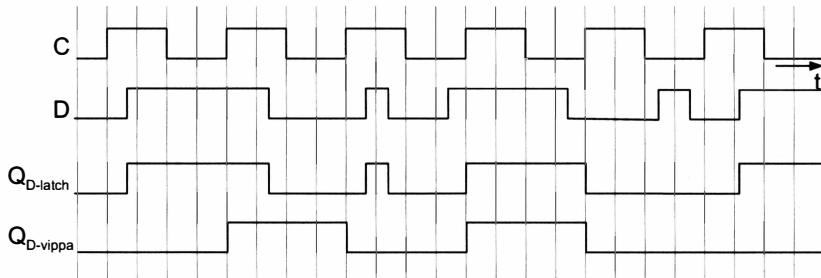
Figur 5.75 Klockad SR-latch med Preset och Clear.

## D-latchen

D-latchen är liksom SR-latchen en minneskrets som kan läsas i tillståndet 0 eller 1, men till skillnad från SR-latchen som har en insignal för 0-ställning och en insignal för 1-ställning, så har D-latchen endast en insignal D som bestämmer tillståndet och en klocksignal C som öppnar och stänger latchesn. D-latchen förekommer endast klockad. D-latchen kan konstrueras med en klockad SR-latch och en inverterare enligt figuren nedan. Funktionen är enkel. För C = 0 befinner sig D-latchen i sitt viloläge. För C = 1 blir för D = 0 SR-latchen 0-ställd eftersom SR = 01, och för D = 1 blir SR-latchen 1-ställd eftersom SR = 10. Funktionen kan sammanfattas som att när klocksignalen C = 1, så följer utsignalen Q värdet hos insignalen D, i princip direkt förbindelse mellan ingång och utgång, och D-latchen brukar därför karakteriseras som *transparent*. När klocksignalen C blir 0, läses utgångsvärdet Q till det värde insignalen D har i detta ögonblick. D-ingången kan tolkas som Dataingång, data 0 eller 1 som skall matas in i D-latchen, läggs på denna ingång. D-latchens symbol skiljer sig från den flanktriggade D-vippans symbol genom att den inte har någon pil på ingången för klocksignalen. Studera som jämförelse D-vippans tillståndstabell i figur 2.42 i kapitel 2. Nedan visas ett tidsdiagram för en D-latch där det som jämförelse också i samma diagram visas tidsdiagram för en positivt flanktriggad D-vippa.

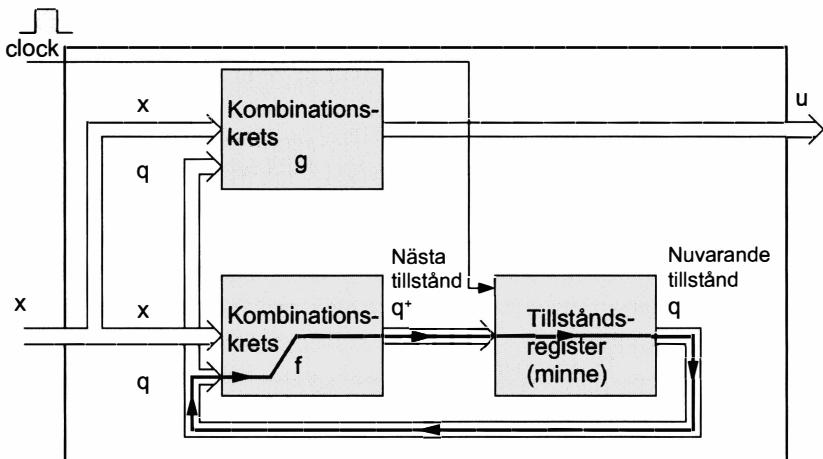


Figur 5.76 D-latch.



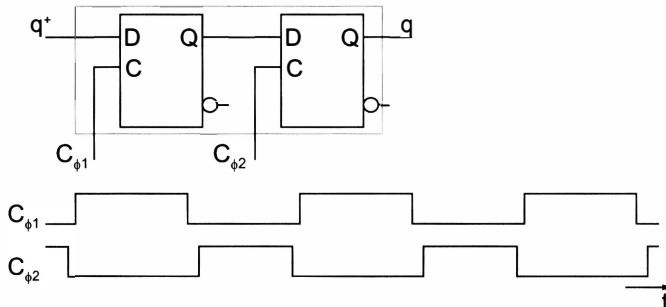
Figur 5.77 Tidsdiagram för en D-latch och en positivt flanktriggad D-vippa.

I modellerna för synkrona sekvenskretsar av typ Mealy och Moore ingår ett tillståndsregister. Tillståndsregistret fungerar ju så, att på klockpulsens aktiva flank tilldelas utsignalen  $Q$  värdet hos insignalen  $D$ , dvs  $Q^+$ . Nu efter att ha studerat D-latchen kan man kanske, utan närmare eftertanke, tycka att tillståndsregistret borde kunna realiseras med D-latchar, en för varje tillståndsvariabel och med gemensam klocksignal. Detta kommer emellertid inte att fungera p.g.a. att D-latchen är transparent! När D-latchen öppnas med klocksignalen  $C = 1$ , kommer det att bildas en sluten väg från D-latchens ingång genom D-latchen och kombinationskretsen  $f$  till D-latchens ingång enligt figuren nedan.



Figur 5.78 Sluten väg  $q^+$  till  $q$  om tillståndsregistret realiseras med D-latchar.

Ett sätt att lösa detta problem är att i stället för en D-latch för varje tillståndsvariabel i tillståndsregistret, koppla två D-latchar efter varandra enligt figuren nedan.

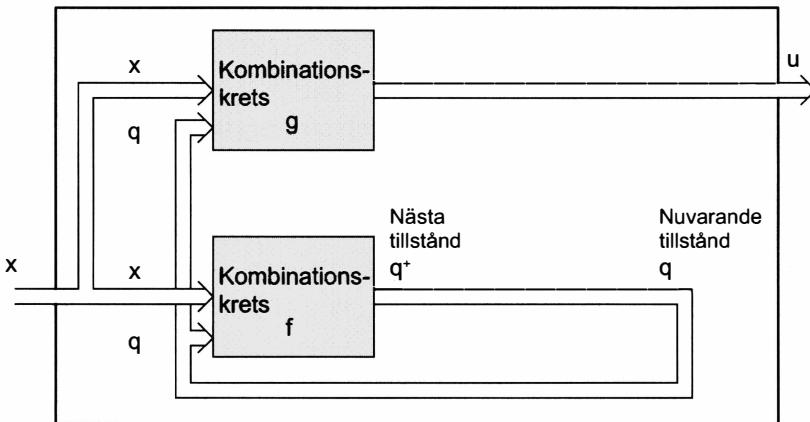


Figur 5.79 Två kaskadkopplade D-latchar med tvåfasklockning.

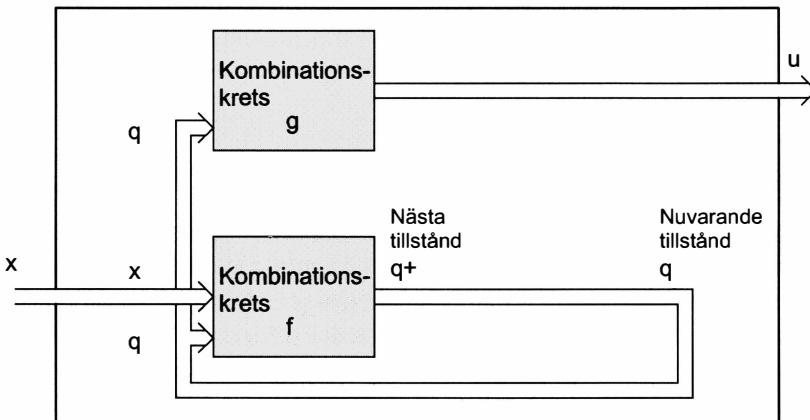
$q^+$  slussas över till  $q$  i två steg. Först öppnas den vänstra D-latchen och stängs sedan igen, medförande att  $q^+$  befinner sig mellan D-latcharna. Därefter öppnas den högra D-latchen och stängs sedan igen, medförande att  $q^+$  nu nått utgången  $q$ . Utsignalen  $q$  har således tilldelats  $q^+$  utan att det varit en öppen väg mellan ingången  $q^+$  och utgången  $q$ . D-latcharna klockas med var sin klocksignal, enligt figuren, s k *tvåfasklockning*, där klocksignalerna aldrig är 1 samtidigt. Principen går ibland på engelska under benämningen *master-slave*, slaven (den andra D-latchen) följer alltid mästaren (den första D-latchens), utsignal.

## 5.5 Asynkrona sekvenskretsar

I en asynkron sekvenskrets är minnet normalt invävt i kretsens struktur och ej direkt urskiljbart såsom i en synkron sekvenskrets, där minnet utgörs av ett tillståndsregister styrt av en klocksignal. Asynkrona sekvenskretsar kan också representeras med modellerna typ Mealy och Moore, enda skillnaden jämfört med synkrona motsvarigheterna är att tillståndsregistret är borta.

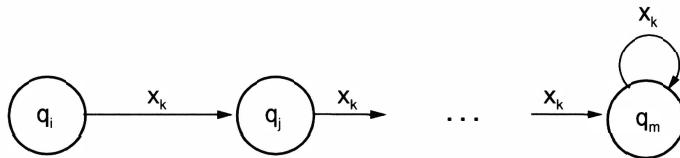


Figur 5.80 Modell för en asynkron sekvenskrets typ Mealy.

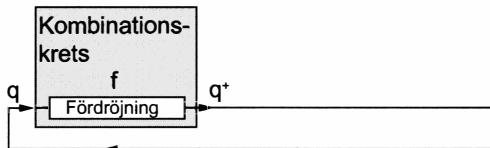


Figur 5.81 Modell för en asynkron sekvenskrets typ Moore.

I modellerna ovan ser vi att nästa tillstånd  $q^+$  är lika med nuvarande tillstånd  $q$ , dvs att  $q^+ = q$ , vilket vi också noterade för SR-latchen, som är en asynkron sekvenskrets. Detta måste dock nödvändigtvis så småningom inträffa efter en tillståndsändring, om inte kretsen ständigt skall ändra tillstånd och bete sig som en oscillator. I en asynkron sekvenskrets styrs tillståndsändringarna av ändringar i insignalen. Detta innebär i sekvenskretsmodellen, att i kombinationskretsen  $f$  bildas av det nya värdet av insignalen och nuvarande tillstånd, ett nytt tillstånd  $q^+$ , vilket tillsammans med fortfarande samma ingångsvärde, efter en viss födröjning i kombinationskretsen ger ett eventuellt nytt tillstånd  $q^+$  osv., tills slutligen nästa tillstånd  $q^+$  är lika med nuvarande tillstånd enligt principen i figuren nedan. För tillståndet  $q_m$  gäller att  $q^+ = f(q_m, x_k) = q_m$ . Tillståndet  $q_m$  är ett *stabilt tillstånd*. Tillståndsövergångar i en asynkron sekvenskrets måste alltid leda till ett stabilt tillstånd.



Figur 5.82 Princip för tillståndsövergång i en asynkron sekvenskrets.



Figur 5.83 Födröjning  $q \rightarrow q^+$  i kombinationskretsen  $f$  i en asynkron sekvenskrets.

Nedan visas en trivial asynkron sekvenskrets, som för insignalen  $x = 1$  inte intar ett stabilt tillstånd, utan ständigt ändrar tillstånd.



Figur 5.84 Asynkron sekvenskrets, instabil för insignalen  $x = 1$ .

I en asynkron sekvenskrets måste insignalen efter en ändring som leder till en tillståndsändring, självklart ha kvar sitt värde tills det nya tillståndet intagits. Vidare får *endast en insignal åt gången styra en tillståndsändring*. Om nämligen flera insignalen ändras samtidigt, kan helt felaktiga tillståndsvägar genomlöpas p.g.a. födröjningarna i kretsen. Om exempelvis en asynkron sekvenskrets har två insignalen som ändras från 00 till 11, så sker detta inte momentant, utan via någon av vägarna  $00 \rightarrow 01 \rightarrow 11$  eller  $00 \rightarrow 10 \rightarrow 11$ , och de mellanliggande insignalvärdena 01 och 10 kan då leda in på tillståndsvägar som ger felaktigt sluttillstånd.

*För en asynkron sekvenskrets måste sålunda gälla att:*

- Endast en insignal åt gången får styra en tillståndsändring
- Varje insignal kombination måste leda till ett stabilt tillstånd

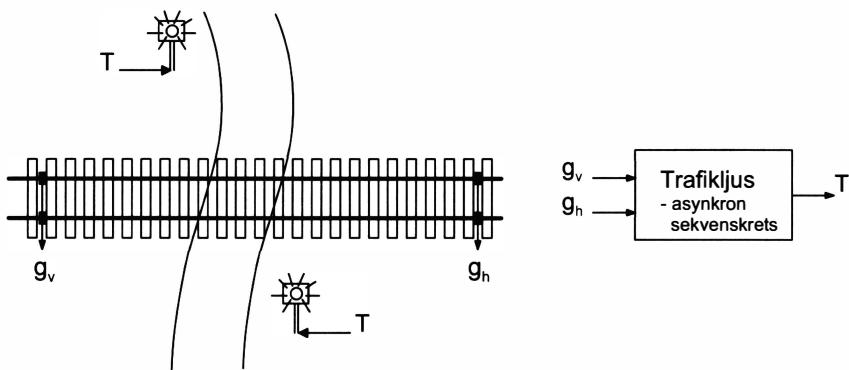
Det finns fler restriktioner som måste beaktas vid realisering av en asynkron sekvenskrets. Låt oss belysa detta i ett exempel.

## Realisering av en asynkron sekvenskrets

### Specifikation

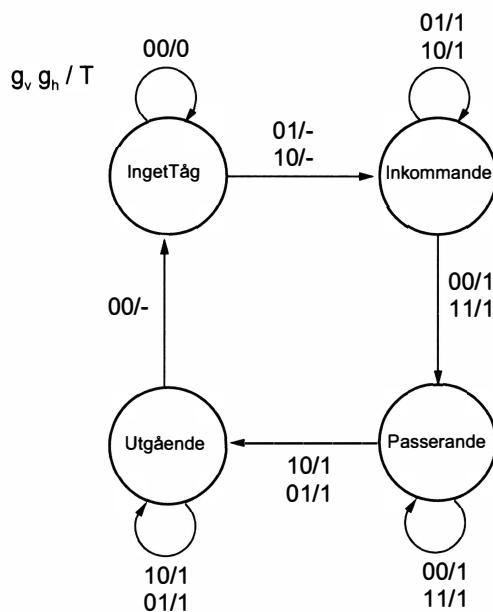
Vid en järnvägskorsning utan bommar där en väg korsar en järnväg finns ett trafikljus, som styrs av två givare  $g_v$  och  $g_h$  på rälsen, belägna på vänster respektive höger sida av vägen ca två kilometer från korsningen, se figuren nedan. Givaren ger utsignalen  $g = 1$  när tåget befinner sig över givaren och annars  $g = 0$ . Trafikljuset skall slå om från vitt blinkande sken till rött blinkande sken när början av tåget kommer in över den ena givaren och slå om till vitt blinkande sken igen när slutet av tåget lämnar den andra givaren. Funktionen skall vara densamma oavsett om tåget är kortare eller längre än avståndet mellan givarna. – En asynkron sekvenskrets Trafikljus som styr trafikljuset, skall konstrueras. Den skall ha två ingångar till vilka skall anslutas de två utsignalerna  $g_v$  och  $g_h$  från givarna, och en utgång  $T$ , som skall anslutas till trafikljusen. För utsignalen  $T$  gäller att  $T = 0$  ger vitt blinkande sken och  $T = 1$  ger rött blinkande sken.

Det kan antagas att ett tåg passerar korsningen fullständigt innan nästa tåg kommer in och att självklart inga tåg möts i korsningen. Dessa antaganden medför att kravet att endast en insignal åt gången får ändra sig, är uppfyllt.



Figur 5.85 Järnvägskorsning som skall styras med en asynkron sekvenskrets.

### Tillståndsdiagram



Figur 5.86 Tillståndsdiagram för trafikljuset.

Beteendet skall uppenbart vara detsamma oavsett om tåget kommer från höger eller från vänster. Tillståenden behöver därför ej innehålla någon information om höger eller vänster. Relativt enkelt kommer man fram till att tillståndsdiagrammet, se figuren ovan, bara behöver innehålla fyra tillstånd, som benämnts *IngetTåg*, *Inkommande*, *Passerande* och *Utgående*.

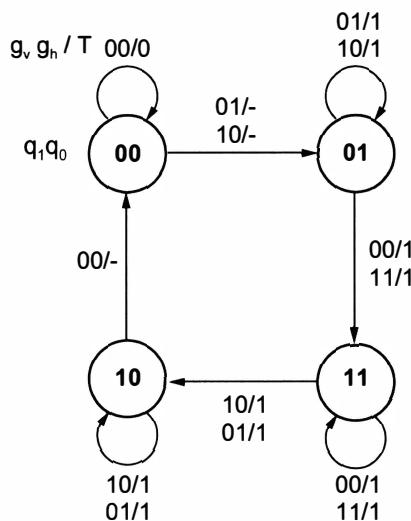
Det finns alltså bara ett tillstånd *Inkommande* och vi skiljer ej på om tåget kommer från höger eller från vänster, och motsvarande gäller för tillståndet *Utgående*.

Starttillstånd är *IngetTåg*, i vilket vi kvarstannar med utgångsvärdet  $T = 0$ , vitt blinkande sken, så länge inget tåg kommer, dvs. för ingångsvärdet  $g_{vh} = 00$ . När ett tåg kommer från höger eller från vänster, dvs. för  $g_{vh} = 01$  respektive  $g_{vh} = 10$ , går vi över till tillståndet *Inkommande*, i vilket vi stannar kvar med utgångsvärdet  $T = 1$ , rött blinkande sken, tills ingångsvärdet ändras. Utgångsvärdet under övergången från tillståndet *IngetTåg* till tillståndet *Inkommande* har satts till don't care, för att eventuellt ge en enklare realisering. Tillståndsövergången sker ju snabbt och det spelar därför ingen roll om utgångsvärdet blir  $T = 1$  redan under tillståndsövergången eller först då tillståndet *Inkommande* intas. När sedan ett tåg kortare än avståndet mellan givarna passerar mellan givarna, dvs. då  $g_{vh} = 00$ , eller ett tåg längre än avståndet mellan givarna passerar över båda givarna, dvs. då  $g_{vh} = 11$ , går vi över till tillståndet *Passerande*. Vi stannar kvar i detta tillstånd tills  $g_{vh} = 01$  eller  $g_{vh} = 10$ , dvs. då tåget är på utgående, då vi går över till tillståndet *Utgående*. I detta tillstånd kvarstannar vi tills tåget passerat, dvs tills  $g_{vh} = 00$ , då vi går över till tillståndet *IngetTåg*.

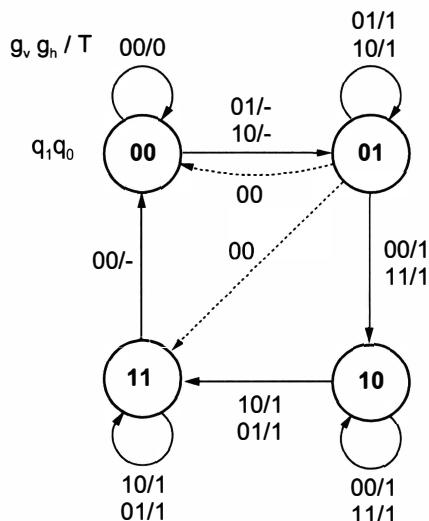
## Tillståndskodning

Vi har tidigare vid realisering av synkrona sekvenskretsar sett hur tillstånden i princip kan kodas fritt, men att kodningen har betydelse för kretsen komplexitet. Vid realisering av asynkrona sekvenskretsar måste tillståndskodningen ske på speciellt sätt för att inte kretsen funktion skall äventyras. *Tillståndskodningen måste vara kapplöpningsfri*, innebärande att bara en tillståndsvariabel åt gången ändrar sig vid varje tillståndsövergång.

I vårt fall kan kapplöpningsfri kodning göras med Gray-kod enligt figuren nedan.



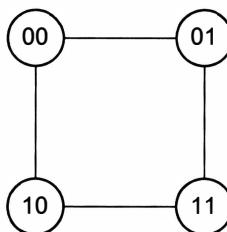
Figur 5.87 Kapplöpninsfri tillståndskodning med Gray-kod.

Figur 5.88 **Icke** kapplöpningsfri tillståndskodning.

Om vi i stället gör en *icke* kapplöpningsfri kodning enligt figuren ovan kan funktionen bli felaktig. Antag exempelvis att vi befinner oss i tillståndet 01 och lägger på ingångsvärdet  $g_v g_h = 00$ , varvid enligt tillståndsdiagrammet övergång skall ske till tillståndet 10. Om tillståndsvariabel  $q_0$  ändrar sig före  $q_1$  sker istället felaktig övergång till tillståndet 00 (streckad pil i tillståndsdiagrammet), i vilket man enligt den normala övergången kommer att kvarstanna med ingångsvärdet  $g_v g_h = 00$ . En felaktig tillståndsövergång har alltså inträffat. Likaså inträffar en felaktig tillståndsövergång till samma tillstånd 00 om i stället tillståndsvariabel  $q_1$  ändrar sig före  $q_0$ . Då sker först felaktig övergång till tillståndet 11 (streckad pil i tillståndsdiagrammet) och sedan övergång till tillståndet 00 enligt den normala vägen.

För vårt tillståndsdiagram var det enkelt att finna en kapplöpningsfri kodning. Om tillståndsdiagrammet innehållit diagonala tillståndsövergångar hade det inte varit möjligt att finna en kapplöpningsfri kodning med två tillståndsvariabler och sålunda totalt fyra tillstånd. Man hade blivit tvungen att öka antalet tillstånd och ändra strukturen i tillståndsdiagrammet för att kunna göra en kapplöpningsfri kodning. I stället för att då först konstruera tillståndsdiagrammet utan tanke på tillståndskodningen och därefter kanske sedan konstatera att en kapplöpningsfri kodning inte är möjlig att göra, är det lämpligt att redan från början konstruera tillståndsdiagrammet så att strukturen blir sådan att kapplöpningsfri kodning blir möjlig.

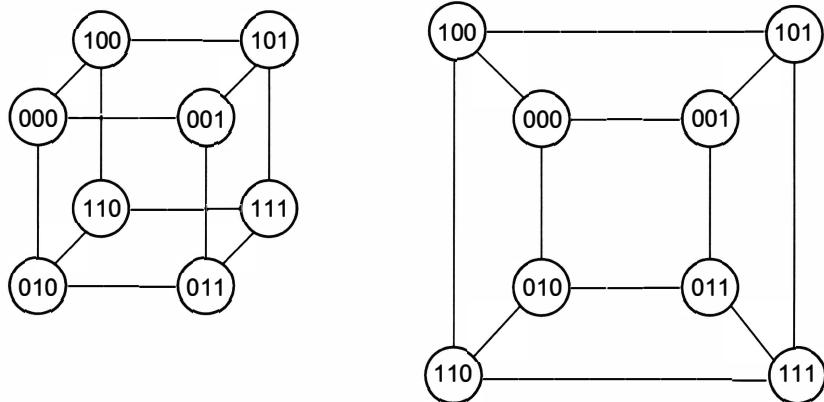
Ett tillståndsdiagram med tre eller fyra tillstånd måste ha en struktur och kodning med två tillståndsvariabler enligt figur 5.89 nedan. Tillståndsövergångarna får bara följa kvadratens kanter och ej gå diagonalalt.



Figur 5.89 Kapplöpningsfri tillståndskodning med två tillståndsvariabler.

Ett tillståndsdiagram med fem till åtta tillstånd måste ha en struktur och kodning med tre tillståndsvariabler enligt figuren nedan. Tillståndsöver-

gångarna får bara följa kubens kanter. Till höger i samma figur visas kuben tillplattad, en form mer lämpad för inplacering av ett aktuellt tillståndsdigram.



Figur 5.90 Kapplöpningsfri tillståndskodning med tre tillståndsvariabler.

### Tillståndstabell

Tillståndstabellen till den kapplöpningsfria kodningen i figur 5.87 ovan får följande utseende.

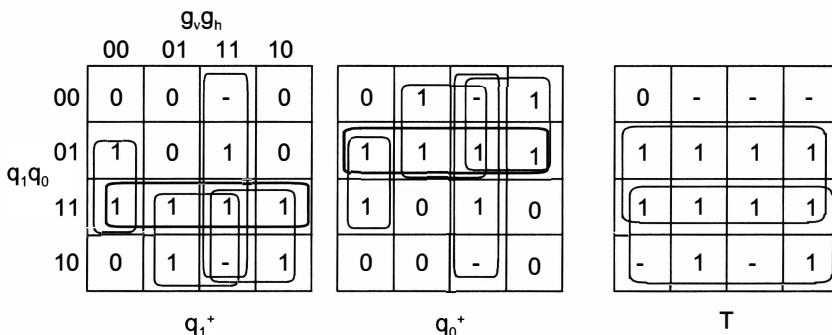
Tabell 5.5: Tillståndstabell till tillståndsdigrammet i figur 5.87.

Nuvarande tillstånd $q_1q_0$	Nästa tillstånd $q_1^+q_0^+$ / Utsignal T			
	Insignaler $g_v g_h$			
	00	01	11	10
00	<u>00</u> /0	01/-	--/-	01/-
01	11/1	<u>01</u> /1	11/1	<u>01</u> /1
11	<u>11</u> /1	10/1	<u>11</u> /1	10/1
10	00/-	<u>10</u> /1	--/-	<u>10</u> /1

Tidigare har konstaterats att i en asynkron sekvenskrets måste varje insignal kombination leda till ett stabilt tillstånd. I tillståndstabellen ovan innebär detta att det i varje insignal-kolumn måste finnas ett tillstånd sådant att  $q^+ = f(q, g_v g_h) = q$ . Sådana tillstånd är understrukna i tillståndstabellen.

### Booleska funktioner för nästa tillstånd och utgångsvärde

Ur tillståndstabellen ovan erhålls direkt Karnaughdiagrammen



Figur 5.91 Karnaughdiagram för  $q_1^+$ ,  $q_0^+$  och  $T$ .

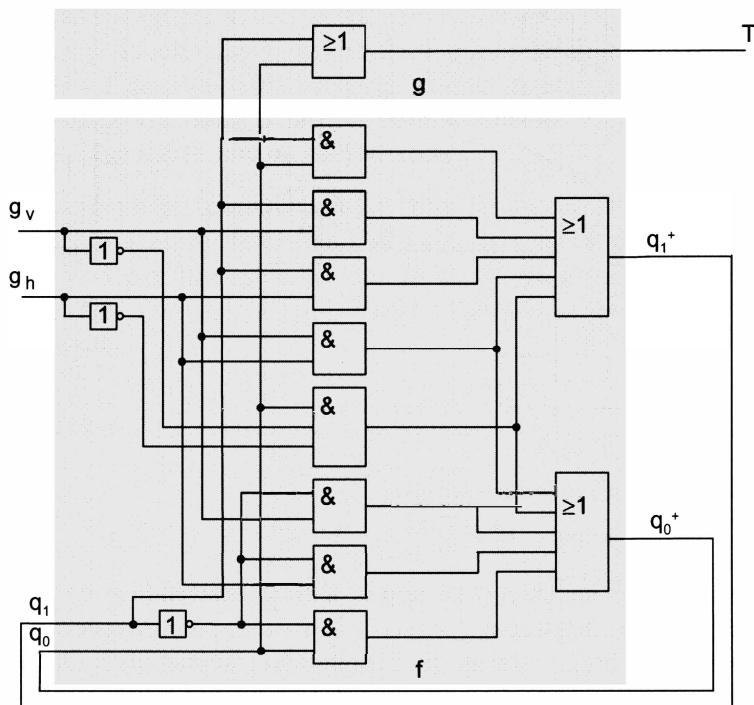
Ur Karnaughdiagrammen erhålls

$$\begin{aligned} q_1^+ &= g_v g_h + q_0 g_v' g_h' + q_1 g_v + q_1 g_h + q_1 q_0 \\ q_0^+ &= g_v g_h + q_0 g_v' g_h' + q_1' g_v + q_1' g_h + q_1' q_0 \end{aligned}$$

$$T = q_0 + q_1$$

Booleska funktionerna  $q_1^+$  och  $q_0^+$  har realiseras *hasardfria* med *samtliga primimplikatorer*. Det är viktigt att så sker, ty annars kan, som berörts tidigare i kapitel 4, Kombinationskretsar, spikar uppträda i utsignalerna från grindnäten som realisrar  $q_1^+$  och  $q_0^+$  och ge upphov till felaktiga tillståndsövergångar. De minimala SP-formerna till  $q_1^+$  och  $q_0^+$  innehåller bara de fyra första primimplikatorerna i uttrycken ovan, men vi har för hasardfrihet också tagit med resterande primimplikatorer. I detta fall en extra primimplikator i vardera funktion som är markerad med tjockare inringning i Karnaughdiagrammen och fet stil funktionsuttrycken.

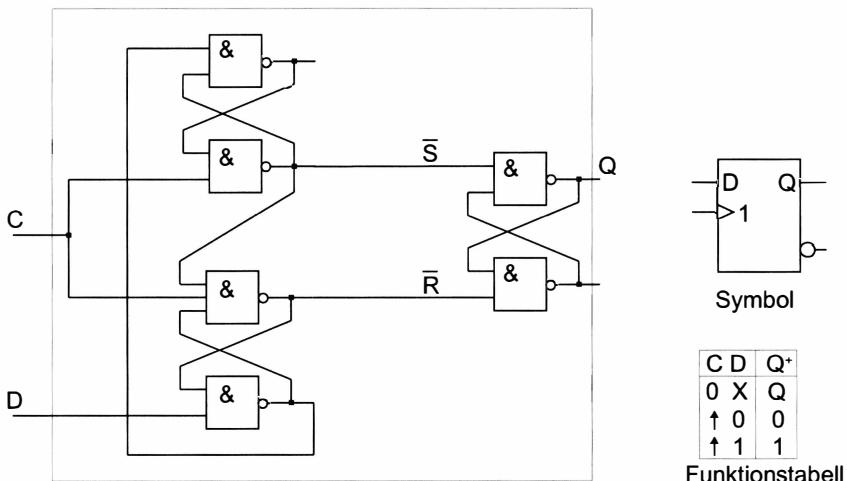
## Kretsschema



Figur 5.92 Kretsschema för asynkrona sekvenskretsen Trafikljus.

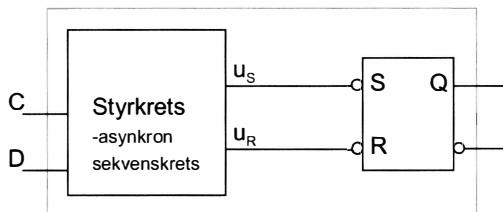
### Exempel 5.6

En flanktriggad D-vippa är en asynkron sekvenskrets. Den kan konstrueras på olika sätt. I figuren nedan visas en variant, som är positivt flanktriggad. I stället för att direkt utgående från kretsschemat analysera beteendet, väljer vi en annan väg där vi får utnyttja våra kunskaper om asynkrona sekvenskretsar. Vi skall utgå från beteendet för en positivt flanktriggad D-vippa och realisera en flanktriggad D-vippa med exakt samma utseende som den givna vippan.



Figur 5.93 D-vippa, positivt flanktriggad.

D-vippan ovan kan naturligt delas upp i två delar, bestående av en asynkron sekvenskrets på ingångssidan, som styr en SR-latch på utgångssidan, enligt figuren nedan. Vi börjar med att rita tillståndsdiagrammet till styrkretsen.



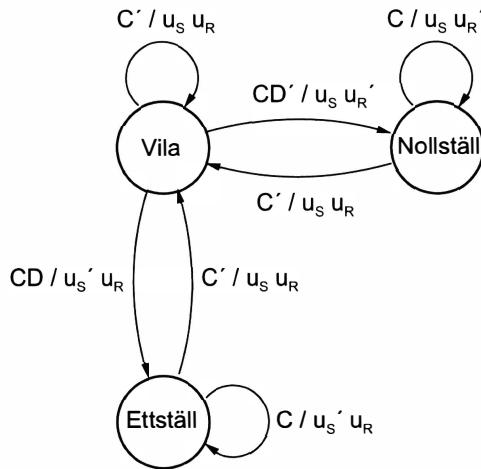
Figur 5.94 D-vippan uppdelad i en asynkron styrkrets och en SR-latch.

#### Tillståndsdiagram för styrkretsen

För klocksignalen  $C = 0$  skall SR-latchen vara i viloläget, innehållande att utsignalerna  $u_S, u_R$  hos styrkretsen skall vara  $u_S, u_R = 11$ . På klocksignalens positiva flank då  $C$  blir 1, skall SR-latchen påverkas så att, för  $D = 0$  skall den 0-ställas, innehållande att styrkretsen skall ge  $u_S, u_R = 10$ , medan för

$D = 1$  skall den 1-ställas, innebärande att styrkretsen skall ge  $u_S u_R = 01$ . Under klockpulsen då  $C = 1$ , skall SR-latchen ej påverkas av insignalen D. Sålunda efter exempelvis 0-ställning på klockpulsens positiva flank och under klockpulsen när  $C = 1$ , så får  $D = 1$  inte ge  $u_S u_R = 01$ , medan efter 1-ställning på klockpulsens positiva flank och under klockpulsen då  $C = 1$ , så skall  $D = 1$  ge  $u_S u_R = 01$ . Eftersom samma ingångsvärde skall ge olika utgångsvärden vid olika tidpunkter kan styrkretsen ej vara en kombinationskrets, utan måste vara en sekvenskrets.

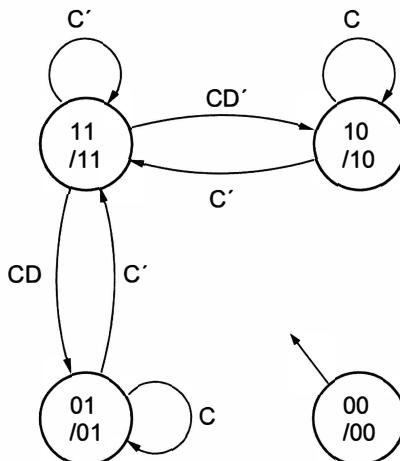
Resonemanget ovan leder relativt lätt fram till att den asynkrona sekvenskretsen behöver minst tre tillstånd, och det visar sig att det räcker med tre tillstånd. Vi döper tillstånden till *Vila*, *Nollställ* och *Ettställ*, som anger vad som sker med SR-latchen i tillstånden. Tillståndsdiagrammet blir enligt figuren nedan.



Figur 5.95 Tillståndsdiagram för den asynkrona sekvenskretsen Styrkrets.

Vi ser att tillståndsdiagrammet är av typ Moore. Tillstånden skall på vanligt sätt för en asynkron sekvenskrets kodas kapplöpningsfritt och i detta fall med tre tillstånd skall kodningen ske enligt kvadraten i figur 5.89 ovan. Det är här möjligt att koda tillstånden så att tillståndsvariablerna direkt kan användas som utsignaler. Tillståndsdiagrammet blir då enligt figuren nedan. Det icke använda tillståndet 00 måste beaktas och det är lämpligt att låta alla tillståndsövergångar för detta tillstånd leda bort från detta tillstånd,

vilket är markerat med en utgående pil från tillståndet. De ospecifierade tillstånden i tillståndstabellen nedan får alltså ej vara tillståndet 00.



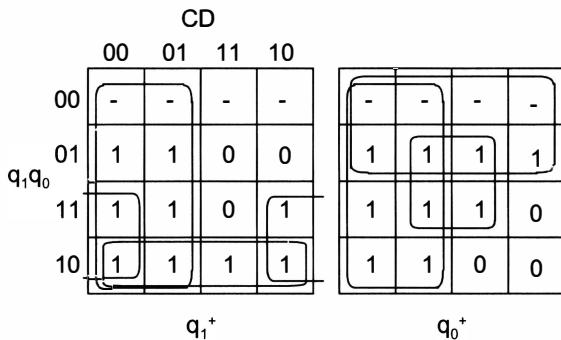
Figur 5.96 Tillståndsdiagrammet för styrkretsen med kapplöpningsfri kodning.

#### Tillståndstabell

$q_1 q_0$	Nästa tillstånd $q_1^+ q_0^+$				Utsignaler $u_S \ u_R$	
	Insignaler CD					
	00	01	11	10		
00	-	-	-	-	00	
01	11	11	01	01	01	
11	11	11	01	10	11	
10	11	11	10	10	10	

Inringningarna i Karnaughdiagrammen nedan för de ospecifierade värdena görs så att dels nästa tillstånd för tillståndet 00 ej blir 00, dels så att vi uppnår målet med konstruktionen av styrkretsen, nämligen att visa att de booleska funktionerna kan realiseras i en krets av exakt samma utseende som styrkretsen till den givna D-vippan. Det visar sig att inringningarna nedan ger önskat resultat.

### Karnaughdiagram



Figur 5.97 Karnaughdiagram för  $q_1^+$  och  $q_0^+$  till styrkretsen.

### Booleska funktioner

$$q_1^+ = C' + q_1 q_0' + q_1 D'$$

$$q_0^+ = C' + q_1' + q_0 D$$

Booleska uttrycken kan omformas enligt

$$\begin{aligned} q_1^+ &= C' + q_1 q_0' + q_1 D' = C' + q_1(q_0' + D') = C' + q_1(q_0 D)' = \\ &= (C(q_1(q_0 D)'))' \end{aligned}$$

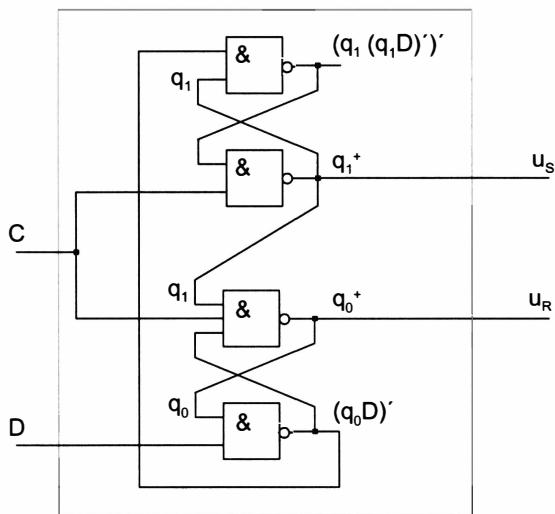
$$q_0^+ = C' + q_1' + q_0 D = (C q_1(q_0 D)')'$$

Styrkretsen har konstruerats så att utsignalerna  $u_S$  och  $u_R$  är lika med tillståndsvariablerna, se tillståndstabellen. Således gäller

$$u_S = q_1$$

$$u_R = q_0$$

Booleska funktionerna  $q_1^+$  och  $q_0^+$  kan realiseras i styrkretsen nedan. Vi kan se att den överensstämmer exakt med den önskade styrkretsen i den givna D-vippan och vi har alltså nått målet att konstruera en D-vippa likadan som den givna.

*Kretsschema*

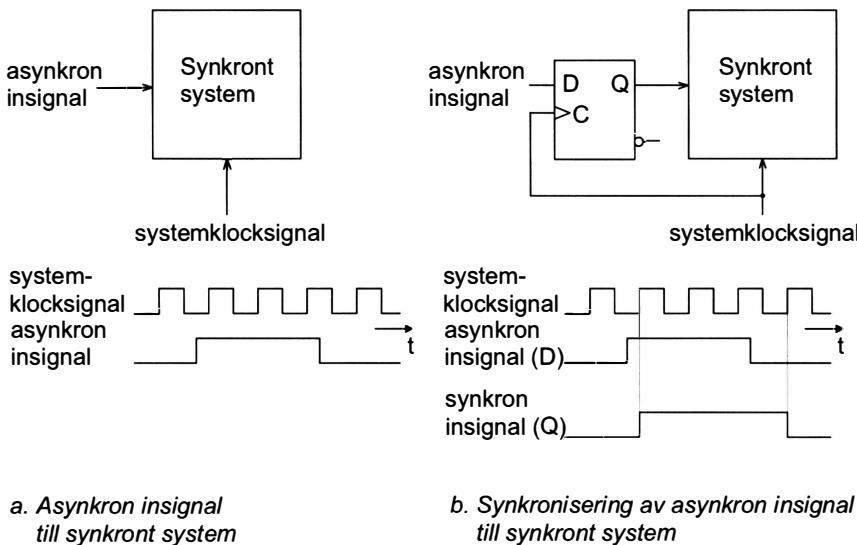
Figur 5.98 Kretsschema för den konstruerade styrkretsen.

□

## 5.6 Synkronisering av asynkrona signaler och metastabilitet

De allra flesta digitala system konstrueras som synkrona system. Alla signalövergångar inuti och ut från det synkrona systemet styrs av en systemklocksignal och sker synkront, t.ex. på klockpulsens positiva flank. För signaler inuti det synkrona systemet har man möjlighet att med uppgift om födröjningar se till så att kraven för de kritiska tiderna för vipporna, inställnings- och hålltider, blir uppfyllda. Problemet är när det synkrona systemet skall ta emot asynkrona insignaler utifrån, vilka exempelvis kan vara utsignaler från ett annat synkront system som har en annan systemklocksignal eller kan vara slumpmässigt uppträdande signaler. Dessa asynkrona insignaler kan förändra värde var som helst i klockpulsintervallet och ibland komma så olyckligt att de inte uppfyller villkoren för vippans inställningstid och hålltid, och då orsaka felaktiga tillståndsövergångar.

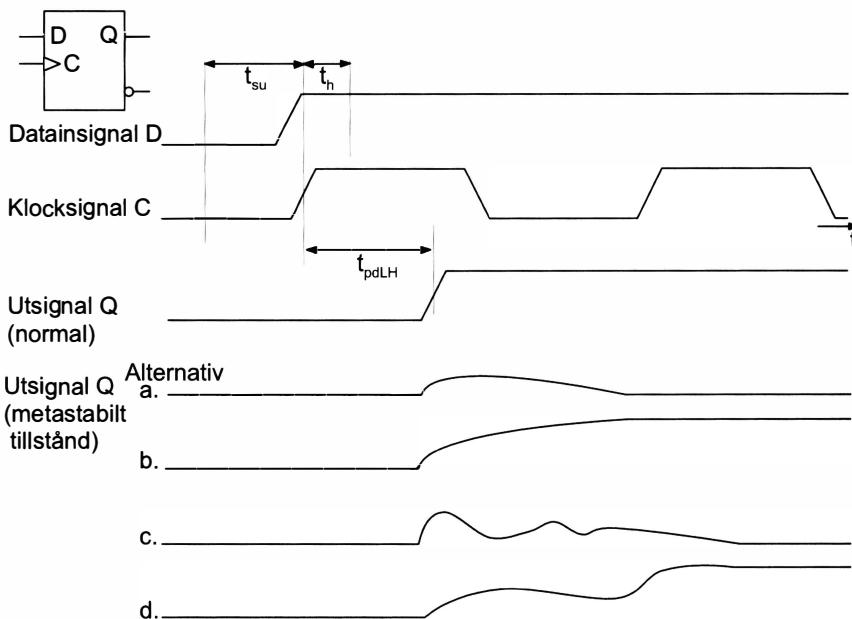
Ett vanligt sätt att synkronisera en asynkron insignal är att låta den passera en flanktriggad D-vippa som klockas med systemklocksignalen, enligt figuren nedan.



Figur 5.99 Asynkron signal till synkront system. Synkronisering.

Det synkrona systemet får nu en insignal som är synkron med systemklocksignalen, visserligen födröjd maximalt en klockpulsperiod, och synchroniseringssproblemet bör väl nu vara löst. Tyvärr är det inte fullt så enkelt. Problemet med inställnings- och hålltider har nu flyttats ut från det synkrona systemet till D-vippan utanför. Vad är nu faran med att D-vippans insignal ändrar sig innanför det otillåtna inställnings-hålltid-intervalliet? Det är inte att omslaget blir oförutsägbart 0 eller 1, ty det kommer vid nästa klockpuls att bli bestämt, utan det är att vippan kan gå över i ett s.k. *metastabil tillstånd* (eng. *metastable state*).

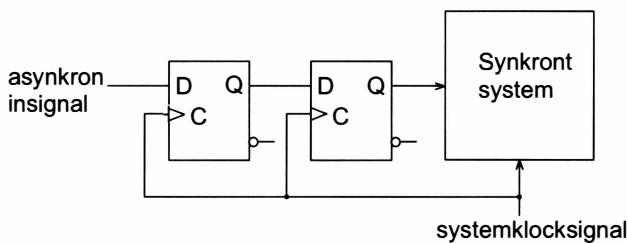
Ett *metastabil tillstånd* för en vippa innebär att vippans utsignal Q befinner sig i det förbjudna utsignalområdet ( $v_{OH\min} - v_{OL\max}$ ) under en tid som är avsevärt längre än den normala omslagstiden. Det är alltså den icke förutsägbara omslagstiden som är problemet. – I figuren nedan visas några olika möjliga alternativ a - d för utsignalen hos en D-vippa i ett metastabil tillstånd, som här tänkts orsakad av att insignalen D ändrats innanför otillåtet intervall för inställningstid  $t_{su}$ .



Figur 5.100 Utsignal hos D-vippa i ett metastabil tillstånd.

I figuren ovan visas utsignalen Q för ett normalt omslag från 0 till 1 samt för ett metastabilt tillstånd. I alternativ a påbörjas ett omslag till 1 som avbryts och återgång sker till 0 och i alternativ b fullföljs omslag till 1, i båda fallen med en omslagstid mycket större än normalt. I alternativ c och d avbryts omslaget respektive fullföljs, men i båda fallen under oscillering och med en omslagstid mycket större än normalt.

Tyvärr är det omöjligt att förhindra att metastabilitet inträffar vid synkronisering av asynkrona insignaler. Det finns emellertid möjlighet att minska sannolikheten för att metastabilitet skall inträffa. – Halvledarfabrikanterna är idag mycket observanta på problemet och brukar i sina databöcker diskutera vippornas beteende vid metastabilitet. De strävar efter att konstruera vipporna så att tidsfönstret där metastabilitet kan inträffa blir så litet som möjligt och när metastabilitet inträffar, tiden i det metastabila tillståndet blir så kort som möjligt. Systemkonstruktören kan minska sannolikheten för metastabilitet genom olika åtgärder. Synkronisering av den asynkrona insignalen kan göras med två kaskadkopplade D-vippor i stället för med en, enligt figuren nedan, till priset att fördöjningen av den asynkrona insignalen nu ökar till maximalt två klockpulsperioder. Vidare bör konstruktionen av det synkrona systemet vad avser de asynkrona insignalerna, följa samma regler som vid konstruktion av ett asynkront system, nämligen att endast en asynkron insignal i taget får påverka en tillståndsövergång och att vid en sådan tillståndsövergång endast en tillståndsvariabel får ändras. Systemklocksignalens frekvens bör också väljas så att den asynkrona signalens pulslängd är större än 2 perioder hos systemklocksignalen.



*Figur 5.101 Synkronisering av en asynkron insignal med två kaskadkopplade D-vippor.*

## 5.7 Övningsuppgifter

### 5.1 Generella sekvenskretsar

**5.1** Rita tillståndsdiagram för sekvenskretsarna beskrivna i a–g nedan. Insekvenserna kan vara godtyckligt långa. Vid sekvensens början antas sekvenskretsen befina sig i ett starttillstånd. Övergång till starttillståndet sker med en synkron signal resetn, som ej behöver beaktas.

**a)** Sekvenskrets typ Moore med en insignal  $x$  och en utsignal  $u$ , sådan att  $u = 1$  om och endast om de fem senaste insignalvärdena varit 11011 (överlappande sekvenser accepteras).

**b)** Sekvenskrets typ Mealy med betende enligt a).

**c)** Sekvenskrets typ Mealy med en insignal  $x$  och två utsignaler  $u_0$  och  $u_1$ . Utsignalen  $u = (u_1, u_0)$  skall ange antalet ettor i insekvensen, räknat modulo-3.

**d)** Sekvenskrets typ Moore med en insignal  $x$  och en utsignal  $u$ , sådan att  $u = 1$  om och endast om insignalen  $x = 1$  i *exakt tre* klocksignalin-tervall. Då en fjärde detta inkommer skall  $u$  återgå till 0.

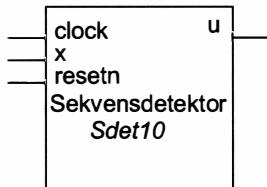
**e)** Sekvenskrets typ Moore med två ingångar  $x$  och  $y$  och två utgångar  $u_0$  och  $u_1$ . På ingångarna  $x$  och  $y$  inmatas två binära positiva heltal med MSB först. För utgångarna skall gälla att

$$\begin{aligned} u_1u_0 &= 00 \text{ om } x = y \\ &= 10 \text{ om } x > y \\ &= 01 \text{ om } x < y \end{aligned}$$

**f)** Sekvenskrets typ Moore med en ingång  $x$  och en utgång  $u$ , sådan att  $u = 1$  om och endast om insekvensen innehåller *exakt en* delsekvens innehållande en eller flera på varandra följande ettor.

**g)** Sekvenskrets typ Moore med två insignaler  $x_0$  och  $x_1$  och en utsignal  $u$ , sådan att  $u = 1$  om och endast om  $x_1x_0 = 11$  och det närmast föregående insignalvärdet varit 11 eller de två närmast föregående insignalvärdena varit i ordning 01, 10.

- 5.2** En sekvenskrets *Sdet10* med insignalerna x, resetn och clock samt en utsignal u skall konstrueras.



Sekvenskretsen skall detektera förekomst av *en* delsekvens 10 i en sekvens i insignalen x. Inmatning av en ny insekvens föregås alltid av en synkron resetn, aktiv låg, som överför sekvenskretsen till ett starttillstånd. Sekvenskretsen skall vara av typ Moore. Utsignalen skall vara 0 i starttillståndet och förbli 0 tills delsekvensen uppträder, då u skall bli 1 under ett klockpulsintervall för att sedan bli 0 under resten av sekvensen.

a) Rita tillståndsdiagram för Sdet10.

b) Bestäm booleska uttryck för nästa tillstånd  $q^+$  och u för tillstånds-kodningarna "binary", "Gray" och "one-hot". Koda tillstånden i ordning med början i starttillståndet.

- 5.3** En sekvensdetektor *Sdet010or0110* av typ Moore med insignalerna x, resetn och clock samt utsignalen u, skall konstrueras. Sekvensdetektor skall i en godtyckligt lång insekvens detektera förekomst av delsekvenser 010 och 0110. Inmatning av en sekvens föregås alltid av en synkron resetn, aktiv låg, som överför sekvenskretsen i ett starttillstånd. Utsignalen skall bli 0 i starttillståndet och förbli 0 tills någon av delsekvenserna uppträder, då u skall bli 1 under ett klocksignalintervall för att sedan åter bli 0 i nästa klocksignalintervall. Efter att en delsekvens detekterats måste innan en ny delsekvens kan detekteras, ha inkommit minst två på varandra följande ettor. – Exempel (efter resetn = 0):

x	110001011010000110011010
u	000000010000100000000010

Rita tilståndsdiagram för sekvenskretsen.

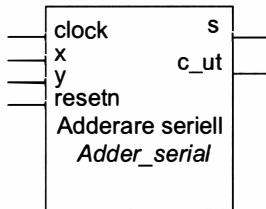
- 5.4 En sekvensdetektor *Sdet11100* av typ Mealy med insignalerna  $x$ , resetn och clock samt utsignalen  $u$ , skall konstrueras. Sekvensdetektorn skall detektera förekomst av delsekvenser 11100 i en godtyckligt lång insekvens. Inmatning av en sekvens föregås alltid av en synkron återställning med signalen resetn, aktiv låg, som överför sekvenskretsen i ett starttillstånd. Utsignalen skall bli 0 i starttillståndet och förblifft 0 tills sekvensen 11100 uppträder då utsignalen skall bli 1 för att sedan åter bli 0 i nästa klocksignalintervall. – Exempel (efter resetn = 0):

x	0010111000101111100001
u	00000000100000000001000

Rita tillståndsdiagram för sekvenskretsen. Koda tillståenden "binary" i ordning med början i starttillståndet. Realisera sekvenskretsen med NAND-grindar och D-vippor.

- 5.5 En sekvensgenerator *Sgen07* som skall generera sekvensen  $(q_2, q_1, q_0) = 000, 111, 001, 110, 010, 101, 011, 100, 000, \dots$ , skall konstrueras. Sekvensgeneratoren skall ha en synkron återställningssignal resetn, aktiv låg, som ger övergång till tillståndet 000. Sekvensgeneratorns utsignaler är tillståndsregistrets utsignaler. Realisera sekvensgeneratorn med NAND-grindar och D-vippor.
- 5.6 En sekvensgenerator *Sgen013764* som skall generera sekvensen  $(q_2, q_1, q_0) = 000, 001, 011, 111, 110, 100, 000, \dots$ , skall konstrueras. Sekvensgeneratoren skall ha en synkron återställningssignal resetn, aktiv låg, som ger övergång till tillståndet 000 samt en insignal enable, aktiv hög, som styr sekvensgenereringen. Sekvensgeneratorns utsignaler är tillståndsregistrets utsignaler. Realisera sekvensgeneratorn med NAND-grindar och D-vippor.
- 5.7 En 3-bitars sekvensgenerator *Sgen064325* som skall generera sekvensen 0, 6, 4, 3, 2, 5, 0, ... i vanlig binärkod i utsignalerna  $q_2, q_1, q_0$ , skall konstrueras. Sekvensgeneratorns utsignaler är tillståndsregistrets utsignaler. De icke använda tillstånden skall som nästa tillstånd ha tillståndet 000.
- a) Rita tillståndsdiagram och bestäm booleska uttryck för  $q^+$ -funktionerna.
- b) Utöka sekvensgeneratorn med synkrona insignaler enable och set6, båda aktiv hög. Signalen set6 skall ge övergång till tillståndet 6, oberoende av värdet hos enable. Komplettera de i a) beräknade  $q^+$ -funktionerna med signalerna enable och set6.

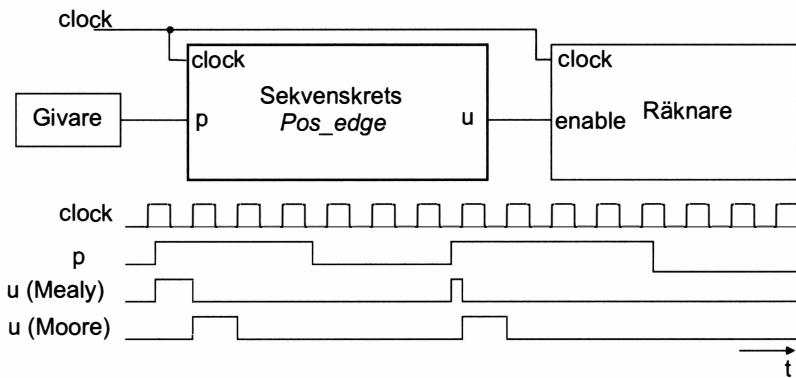
- 5.8** En adderare *Adder\_serial*, som adderar två binärtal i serieform skall konstrueras som en sekvenskrets.



Binärtalen som kan innehålla godtyckligt många bitar, inmatas bit för bit på ingångarna  $x$  och  $y$  med LSB först. Adderaren skall för varje inmatad bit av talen ge summan  $s$  och minnessiffran  $c_{ut}$ . En additon föregås av en resetn, aktiv låg. Rita tillståndsdiagram med lämpliga namn på tillstånden och realisera adderaren med D-vippor som en sekvenskrets av typ

- a) Mealy      b) Moore

- 5.9** En givare ger en fyrkantvåg, vars positiva flanker skall räknas i en räknare. En synkron sekvenskrets *Pos\_edge*, som för varje positiv flank i fyrkantvägen ger enable = 1 under ett klockpulsintervall till räknaren, skall konstrueras. Klocksignalens periodtid är mycket mindre än fyrkantvägens. Både sekvenskretsen och räknaren ändrar tillstånd på klockpulsens positiva flank.



- a) Rita tillståndsdiagram för Mealy-varianten och realisera den med D-vippor.  
b) Rita tillståndsdiagram för Moore-varianten och realisera den med D-vippor.

## 5.2 Räknare

**5.10** En 3-bitars modulo-8 räknare *Cnt3G* som räknar i Gray-kod, ( $q_2, q_1, q_0$ ): 000, 001, 011, 010, 110, 111, 101, 100, 000, ... skall konstrueras. Räknaren skall vara försedd en asynkron resetna, aktiv låg. Bestäm booleska uttryck för

- a) d-signalerna vid realisering med D-vippor.
- b) T-signalerna vid realisering med T-vippor.

**5.11** En 2-bitars räknare *Cnt2Gm* som räknar i Gray-kod skall vara försedd med två modsignaler  $m_0$  och  $m_1$  som styr betendet enligt nedan.

$m_1 m_0$	beteende
0 0	synkron nollställning, dvs övergång till tillståndet 00
0 1	uppräkning i Gray-kod, 00, 01, 11, 10, 00, ...
1 0	nedräkning i Gray-kod, 00, 10, 11, 01, 00, ...
1 1	synkron ettställning, dvs övergång till tillståndet 11

Rita tillståndsdiagram och bestäm booleska uttryck för d-signalerna vid realisering av räknaren med d-vippor.

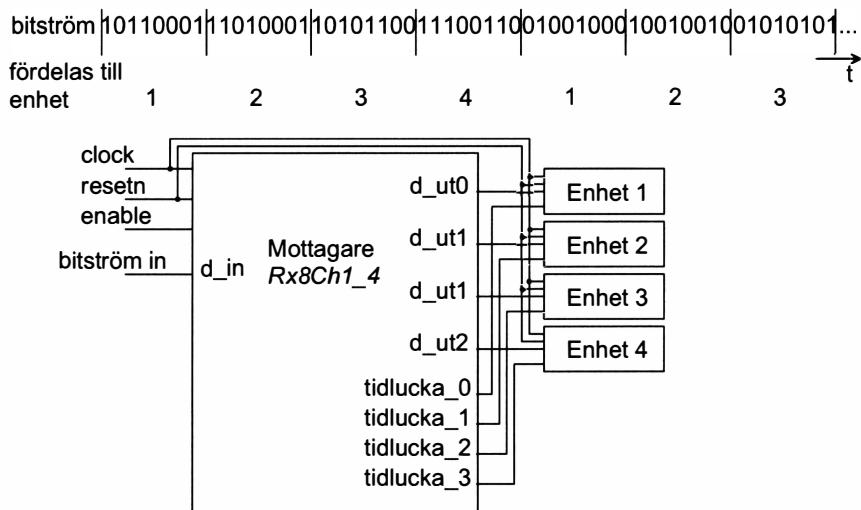
**5.12** En 4-bitars räknare *Cnt4bd*, som skall kunna räkna som binärräknare modulo-16, dvs räkna 0, 1, 2, ..., 15, 0, ... eller som dekadräknare i BCD-kod, dvs räkna 0, 1, 2, ..., 9, 0, ..., skall konstrueras. Med insignalen  $b_d$  skall styras mod binärräknare ( $b_d = 1$ ) eller dekadräknare ( $b_d = 0$ ). Räknaren skall ha en synkron resetn, aktiv låg. Realisera räknaren med NAND-grindar och T-vippor.

**5.13** En speciell 3-bitars binärräknare *Cnt3b\_plus3* skall konstrueras. Räknaren skall kunna arbeta i två olika moder, som väljs med insignalen *plus3*. För *plus3* = 0 skall räknaren arbeta som en konventionell 3-bitars binärräknare och räkna modulo-8 enligt 0, 1, 2, ..., 7, 0, .... För *plus3* = 1 skall räkning ske i vart tredje tillstånd i den normala räknecykeln enligt 0, 3, 6, 1, 4, 7, 2, 5, 0, 3, ... . Räknaren skall ha en synkron resetn, aktiv låg. Realisera räknaren med NAND-grindar och D-vippor.

**5.14** Ange för sändaren Tx8Ch4\_1 i exempel 5.4 vilka modifieringar i blockschemat i figur 5.61 som måste göras om

- a) enheterna skall sända 16-bitars ord i stället för 8-bitars ord.
- b) det är åtta enheter i stället för fyra.

**5.15** På en kanal sänds en bitström synkront med klocksignalen. Bitströmmen består av 8-bitars ord. Orden skall tas emot av en mottagare Rx8Ch1\_4 som skall fördela orden i tur och ordning till fyra enheter via fyra kanaler. Konstruera mottagaren med standardkomponenter.



**5.16** Ange för mottagaren Rx8Ch1\_4 i övningsuppgift 5.15 ovan vilka modifieringar i den föreslagna realiseringen som måste göras om

- a) bitströmmen består 16-bitars ord i stället för 8-bitars ord.
- b) orden skall fördelas till åtta enheter i stället för fyra.

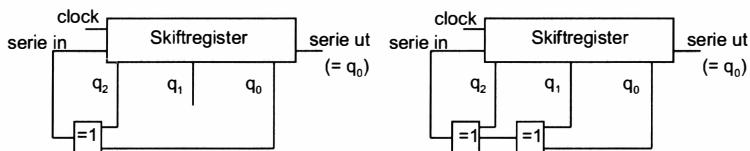
- 5.17** Bestäm maximal klockfrekvens för sekvensgeneratorn Sdet1\_mo i figur 5.8. Insignalen x får sitt nya värde samtidigt som D-vipporna får sina nya q-värden. Grindarna har fördröjningarna  $t_{pd\_Inv} = 0,5$  ns,  $t_{pd\_OCH} = 1$  ns,  $t_{pd\_ELLER} = 1$  ns. För D-vipporna gäller att  $t_{pd\_D} = 0,8$  ns,  $t_{su\_D} = 0,2$  ns och  $t_{h\_D} = 0$  ns.

### 5.3 Register och skiftregister

- 5.18** Skiftregister återkopplade med XOR-grindar enligt principen i figurerna a) och b) nedan, får tillståndsdiagram med speciellt utseende. Tillståndet 0 bildar uppenbart en cykel med sig självt. För de övriga tillstånden gäller att de bildar tillståndscykler av olika längd. – Rita tillståndsdiagram för sekvenskretsarna a) och b) nedan.

a)

b)



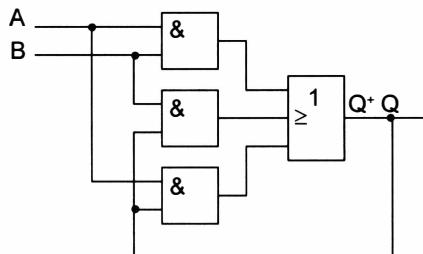
- c) Tillståndsdiagrammet i a) har ett speciellt utseende – samtliga tillstånd, bortsett från tillståndet 0, bildar en enda tillståndscykel. Detta medför att om sekvenskretsen startas i något av tillstånden i denna stora cykel, så kommer utsignalen att få en *maximal periodlängd*, i detta fall periodlängden  $2^3 - 1 = 7$ , och för ett n-bitars skiftregister kan en maximal periodlängd av  $2^n - 1$  erhållas. Verifiera periodlängden genom att bestämma 25 på varandra följande utgångsvärden. Återkopplade skiftregister enligt ovan, benämnes *linjära sekvenskretsar*. De kan bl.a. användas för att generera s.k. *pseudoslumpsekvenser*, sekvenser med stor periodlängd som ter sig slumpmässiga, men som naturligtvis inte är det. Linjära sekvenskretsar kan beskrivas matematiskt.

- 5.19** Bestäm maximal skiftfrekvens för skiftregistret i figur 5.68. Antag att fördröjningen i MUX4\_1,  $t_{pd\_MUX4\_1} = 1,5$  ns. För D-vipporna gäller att  $t_{pd\_D} = 2$  ns,  $t_{su\_D} = 0,5$  ns och  $t_{h\_D} = 0$  ns.

## 5.4 Latchar

**5.20** Analysera  $\overline{S}$   $\overline{R}$ -latchen i figur 5.73 och rita tillståndsdiagram och tillståndstabell.

**5.21 a)** Bestäm tillståndstabell för den asynkrona sekvenskretsen nedan.

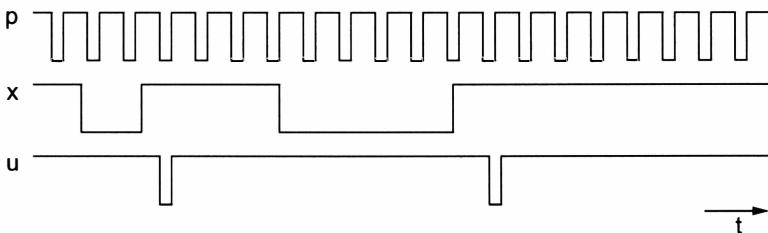
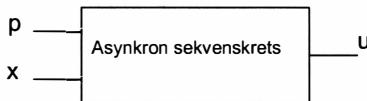


**b)** Sekvenskretsen skulle kunna användas som en latch, benämnd AB-latch. För vilka värden på insignalerna A och B befinner sig latches i sitt viloläge, respektive nollställs och ettställs den?

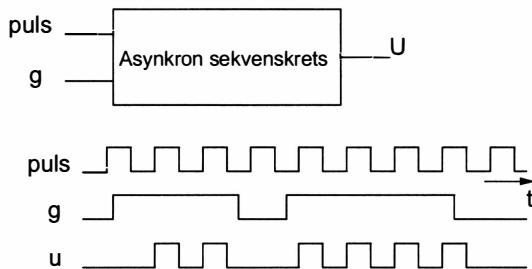
## 5.5 Asynkrona sekvenskretsar

**5.22** Konstruera D-latchen, utgående från tillståndsdiagrammet i figur 5.76, som en asynkron sekvenskrets.

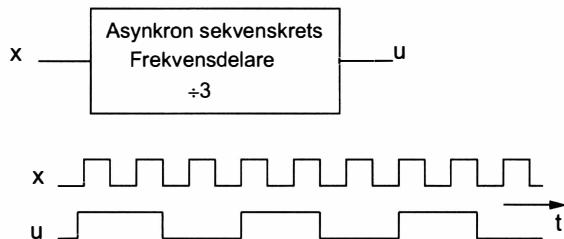
**5.23** En asynkron sekvenskrets med två insignaler p och x samt en utsignal u skall konstrueras. Insignalen p består av aktivt låga pulser. Direkt efter att insignalen x gått från låg (0) till hög (1) skall alltid en och endast en fullständig puls i insignalen p sändas ut i utsignalen u, som i viloläget har värdet hög (1). Insignalen x varierar långsamt i förhållande till insignalen p. Konstruera den asynkrona sekvenskretsen med enbart NAND-grindar.



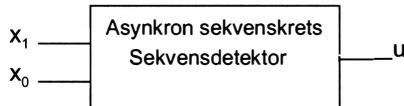
- 5.24** En asynkron sekvenskrets med två insignaler puls och g, samt en ut-signal u skall konstrueras. Insignalen puls utgörs av pulser (0 och 5 V) och insignalen g kommer från en studsfri återfjädrande kontakt, sådan att nedtryckt kontakt ger  $g = 1$  och icke nedtryckt kontakt ger  $g = 0$ . Så länge kontakten är nedtryckt skall fullständiga pulser sändas ut på utgången, fr.o.m. första möjliga fullständiga puls. Om kontakten släpps under pågående puls skall även denna sändas ut fullständigt (se figur). Manövreringen av kontakten g är långsam i förhållande till puls-frekvensen. – Konstruera tillståndsdiagram för sekvenskretsen och koda tillstånden.



- 5.25** En asynkron sekvenskrets som delar frekvensen hos en signal x med 3, skall konstrueras. Beteendet visas nedan. Konstruera kretsen med OCH-, ELLER-grindar och Inverterare.



**5.26** En sekvensdetektor med två insignaler  $x_0$  och  $x_1$  och en utsignal  $u$ , skall konstrueras som en asynkron sekvenskrets. För utsignalen  $u$  skall gälla att  $u = 1$  om och endast om de tre senaste insignalvärdena är  $x_1x_0: 00, 10, 11$ . Endast en insignal ändrar sig i taget. Konstruera kretsen med NAND-grindar.



Exempel.

$x_1:$  0 0 1 0 0 1 1 0 0 1 1 1

$x_0:$  0 1 1 1 0 0 1 1 0 0 1 0

$u:$  0 0 0 0 0 0 1 0 0 0 1 0

# 6 MOS-transistorn

## Grindar i CMOS och nMOS

I kapitel 2 behandlades grindar som logiska byggblock. I detta kapitel skall vi studera hur de realiseras med transistorer. Vi kommer då att hålla oss till uppbyggnad med MOS-transistorer, den totalt dominerande transistorn inom digitaltekniken.

Få digitaltekniker utför konstruktion av chip på transistornivå, normalt förekommer detta bara hos de stora halvledarfabrikena. För att kunna konstruera på denna nivå krävs omfattande kunskaper i bl. a. halvledarfysik och tillgång till mycket avancerade datorbaserade konstruktionsverktyg. Men även om man som digitaltekniker normalt inte konstruerar på transistornivå, utan på grindnivå eller ännu högre nivå, så behöver man elementära kunskaper om digitala kretsar på transistornivå för att kunna förstå och få känsla för begrepp som logiska nivåer, störmarginaler, belastbarhet, födröjningar m.m.

Kapitlet inleds med en kortfattad beskrivning av MOS-transistorns uppbyggnad och funktion. Det blir bara en ytlig betraktelse som resulterar i en mycket enkel modell för MOS-transistorn, i stort sett bara som switch, tillräcklig för förståelse av grindarnas funktion.

Digitala kretsar konstrueras i MOS-teknik huvudsakligen i två tekniker benämnda *CMOS* och *nMOS*. En Inverterare realiseras i CMOS respektive nMOS kommer att studeras och få illustrera struktur, statiska och dynamiska egenskaper för dessa två fundamentala MOS-tekniker. Därefter visas uppbyggnaden av andra grindar såsom NAND, NOR och XOR.

## 6.1 MOS-transistorn

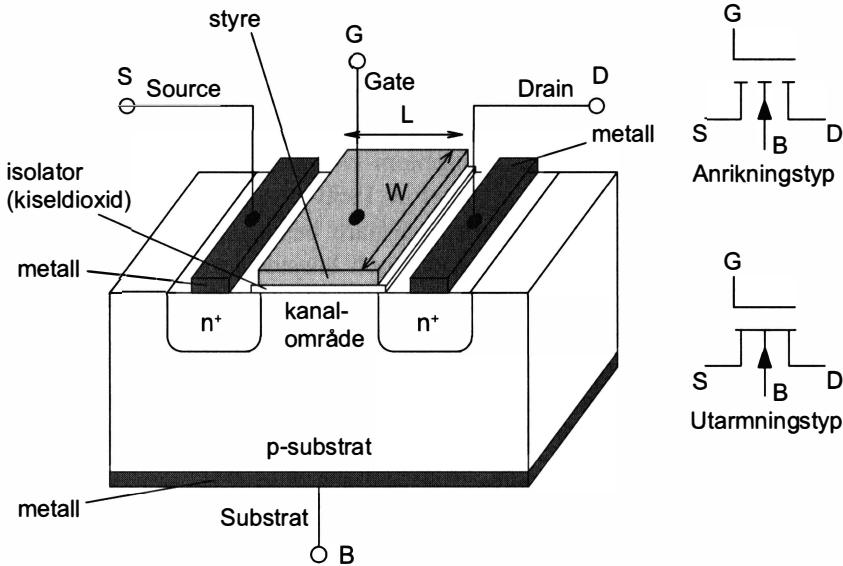
MOS-transistorn (eng. *Metal Oxide Semiconductor*) demonstrerades första gången år 1960 vid Bell Laboratories i USA. Den är en utveckling av FET-transistorn (*Fälteffektransistorn*) som konstruerades redan på 1950-talet, samtidigt som den bipolära transistorn, och dess fullständiga namn är egentligen MOSFET (eng. *Metal Oxide Semiconductor Field Effect Transistor*). Svårigheter att bemästra problem vid tillverkningen av MOS-transistorn gjorde att det dröjde till slutet av 1960-talet innan MOS-transistorn fick sitt kommersiella genombrott, närmare bestämt till år 1968, då det amerikanska företaget Intel grundades och började introducera halvledarminnen och mikroprocessorer och då det amerikanska företaget RCA introducerade den första CMOS-familjen CD 4000 av digitala standardkretsar. MOS-transistorn medger mycket hög packningstäthet, miljontals MOS-transistorer på ett chip och låg effektförbrukning. MOS-transistorn har varit förutsättningen för den fantastiska elektronikutvecklingen sedan 1970-talet.

### Uppbyggnad

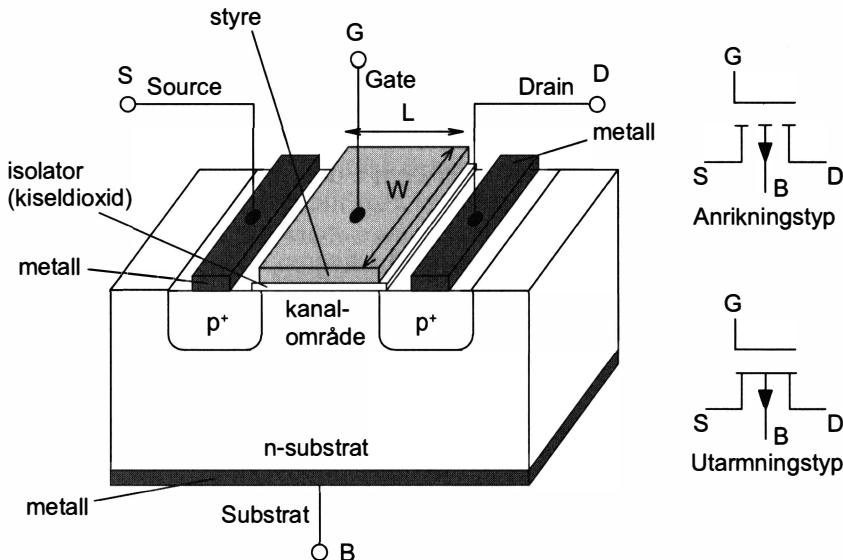
Det finns två huvudtyper av MOS-transistorn, *n-kanal (nMOS)* och *p-kanal (pMOS)*, vars schematiska uppbyggnad visas i figur 6.1 nedan. – MOS-transistorn har fyra elektroder, *emitter* (eng. *source*) S, *kollektor* (eng. *drain*) D, *styre* (eng. *gate*) G, och substrat (eng. *substrate*) B. Substratet, underlaget, benämnes på engelska även *body* och *bulk*, därav beteckningen B för substratelektroden. De engelska orden *source* och *drain* är så allmänt vedertagna att vi normalt använder dem i den följande framställningen i stället för *emitter* och *kollektor*.

MOS-transistorns symboler visas också i figuren och som synes finns det två symboler för respektive typ av MOS-transistor, vilket beror på att MOS-transistorn kan vara av antingen *anrikningstyp* (eng. *enhancement type*) eller *utarmningstyp* (eng. *depletion type*), begrepp som kommer att förklaras längre fram.

*nMOS-transistor:*



*pMOS-transistor:*



Figur 6.1 nMOS-transistor och pMOS-transistor

Source- och drain-områdena utgörs i nMOS-transistorn av kraftigt n-dopad kisel i ett substrat av p-dopad kisel, medan det i pMOS-transistorn är precis omvänt, dvs source- och drain-områdena utgörs av kraftigt p-dopad kisel i ett substrat av n-dopad kisel. Genomgående är pMOS ”tvärtom” i förhållande till nMOS, t.ex polaritet på matningsspänning, strömriktningar; nMOS och pMOS är *komplementära* på samma sätt som de bipolära transistorerna npn och pnp.

Styret är isolerat från kanalområdet med ett lager oledande kiseldioxid ( $\text{SiO}_2$ ), ett material som genomgående används som isolator vid uppbyggnaden av MOS-transistorn. Numera tillverkas styret normalt av det ledande materialet *polykristallinskt kisel* (eng. *polysilicon*), som har mycket god ledningförmåga, men ursprungligen tillverkades det av metall (aluminium), därav namnet MOS (*Metal Oxide Semiconductor*). P.g.a. det isolerande skiktet kiseldioxid, är *inresistansen i styret alltid mycket hög*, ca  $10^{12}$  ohm, oavsett polariteten hos inspänningen på styret.

Det isolerande oxidskiktet mellan styre och kanalområde har normalt en tjocklek av ca 0,05 - 0,1 µm. *Kanallängd (L)* och *kanalbredd (W)* vill man naturligtvis göra så små som möjligt för att få plats med så många transistorer som möjligt på ett chip.

Tillverkningsprocessen för integrerade kretsar är till stor del en fotografisk teknik med fotomasker, se figur 6.2 nedan, och man brukar ange minsta möjliga *linjebredden* som mått på vad tillverkningsprocessen medger i integrationsgrad. Idag är linjebredder (och kanallängd och kanalbredd) av storleksordningen 0,1 µm möjliga att åstadkomma i tillverkningsprocessen.

Kanallängd och kanalbredd har betydelse inte bara för packningstätheten, utan också för snabbheten hos transistorerna, ty ju mindre transistorer, desto mindre kapacitanser och därmed snabbare upp- och urladdningar (omslag från logiskt 0 till logiskt 1 och tvärtom). Kanallängd och kanalbredd har också betydelse för transistorernas *strömdrivningsförmåga*, ty kollektorströmmen  $i_D$  (strömmen genom kanalen) är proportionell mot kvoten W/L.

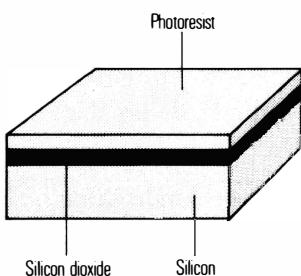


Figure 11.1

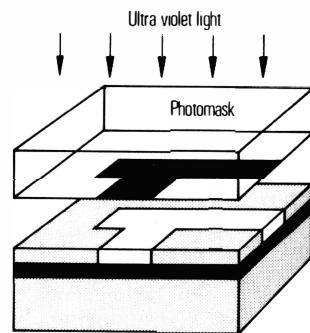


Figure 11.2

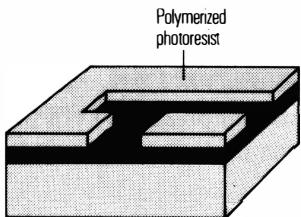


Figure 11.3

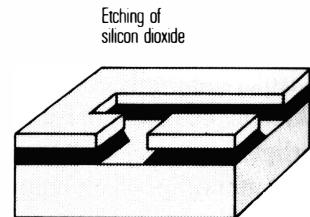


Figure 11.4

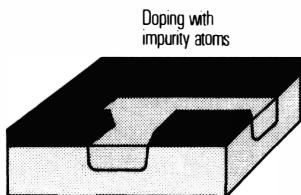
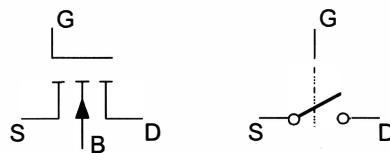


Figure 11.5

Figur 6.2 Schematisk bild av tillverkning av integrerade kretsar. (Siemens: "Components Technical Descriptions and Characteristics for Students, Edition 1986").

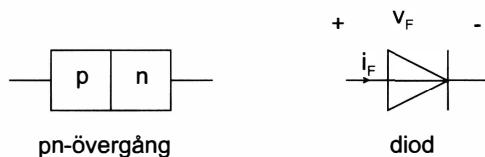
## Funktion

MOS-transistorn används i digitaltekniken som *switch* (strömbrytare) med två lägen. I det ena läget *Till* (eng. *On*) eller *Sluten* (eng. *Closed*) kan ström flyta genom kanalen. I det andra *Från* (eng. *Off*) eller *Öppen* (eng. *Open*) kan ström inte flyta genom kanalen. En analogi med en mekanisk switch visas nedan.



Figur 6.3 Analogi nMOS-transistor och mekanisk switch.

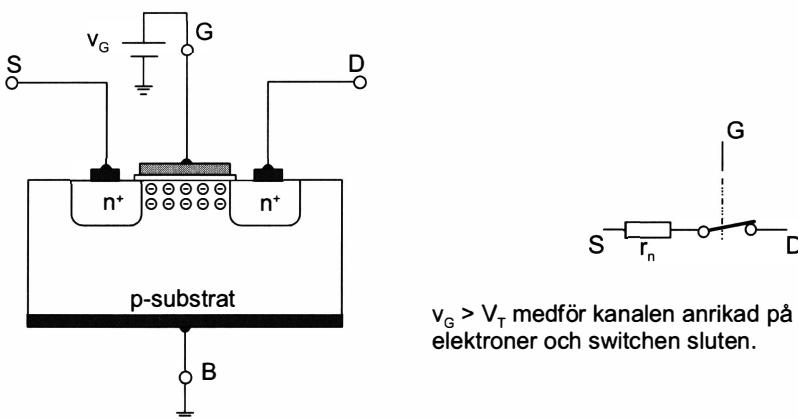
En nMOS-transistor av *anrikningstyp* är tillverkad så att den saknar negativa laddningsbärare, elektroner, i kanalområdet. Vägen mellan source- och drain-elektroderna utgörs av två motriktade dioder, np och pn, med p-området (kanalområdet) gemensamt, se den övre transistorn i figur 6.1 ovan. En pn-övergång är en diod enligt figur 6.4 nedan, en komponent som leder ström  $i_F$  i framriktningen (från p till n), medan den i backriktningen (från n till p) endast leder läckström. För att dioden skall leda i framriktningen krävs att framspänningen  $v_F$  över dioden är minst ca 0,7 V.



Figur 6.4 pn-övergång som diod.

Utan någon inspänning på styret så kan det alltså endast flyta läckström (diobbackström) mellan source och drain och switchen är öppen. För att ström skall kunna flyta mellan source och drain, dvs switchen skall bli sluten, så måste elektroner föras in i kanalen så att det blir en n-kanal mellan de n-do-

pade source- och drain-områdena. Om styret läggs på positiv potential i förhållande till substratet så kommer elektroner att attraheras in i kanalområdet från det p-dopade substratet och de n-dopade source- och drain-områdena. När potentialen på styret överskriden en viss positiv *tröskelspanning* (eng. *threshold voltage*)  $V_T$ , har det kommit in så många elektroner i kanalområdet att en kanal (ledare) bildats mellan source och drain och switchen är sluten, se figur 6.5 nedan. MOS-transistorn är en *icke-ideal* switch med en viss *till-resistans*, som i den mekaniska switchsymbolen markerats med en resistans  $r_n$  (index  $n$  för n-kanal) i serie med switchen (ideal switch).

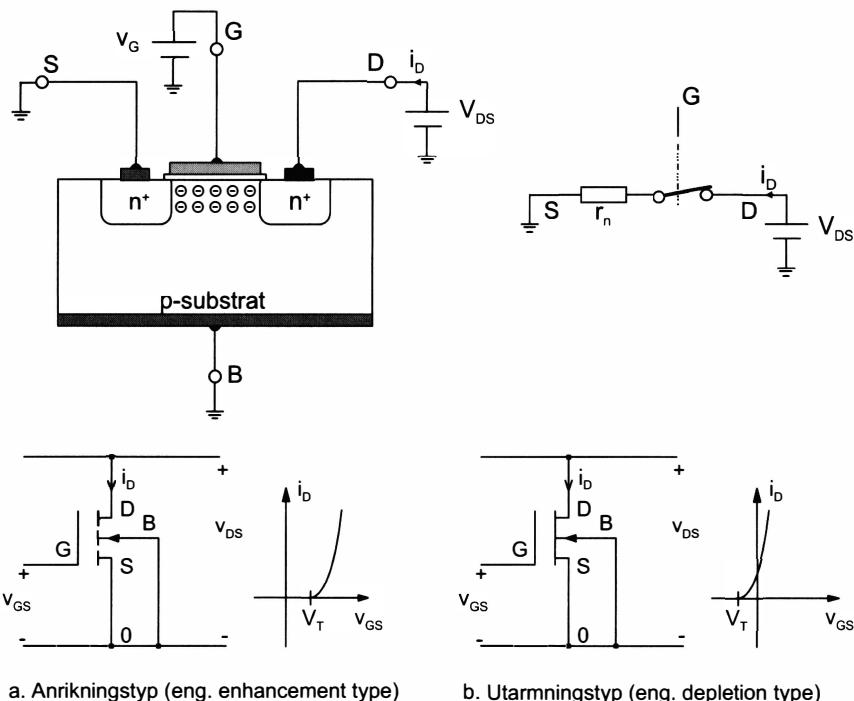


Figur 6.5 nMOS-transistor av anrikningstyp med kanalen anrikad på elektroner.

Med switchen sluten, elektroner i kanalen, så kan en ström flyta mellan source och drain om en spänning läggs över dem, se figur 6.6 nedan. Ström kan alltså flyta i båda riktningarna i kanalen. MOS-transistorn är symmetriskt uppbyggd och source och drain är egentligen definierade först när spänningen är pålagd över dem, elektroden med högsta potentialen blir drain. Med substrat och source jordade, som i figur 6.6, så kommer en kollektorström  $i_D$  att flyta från drain till source. Ökas inspänningen  $v_{GS}$  så ökar kollektorströmmen  $i_D$ , se figur 6.6a, vilket i den enkla modellen för nMOS-transistorn med en resistor i serie med en switch, motsvaras av att resistansen  $r_n$  minskar.

Vi kan sålunda konstatera att för denna typ av nMOS-transistor tillverkad så att kanalen saknar negativa laddningsbärare, krävs att spänningen  $v_{GS}$  mellan styre och source är större än en viss tröskelspanning  $V_T$ , för att kanalen skall bli *anrikad* på negativa laddningsbärare och ström kunna flyta mellan source och drain. Denna nMOS-transistor är av *anrikningstyp* (eng. *enhancement type*).

En nMOS-transistor kan också tillverkas med kanalområdet n-dopat, dvs. innehållande negativa laddningsbärare, vilket innebär att det utan inspänning på styret kan flyta ström mellan source och drain om en spänning läggs över dem. För att denna transistor skall bli strypt måste en negativ inspänning appliceras på styret, så att de negativa laddningsbärarna stötes bort från kanalområdet och kanalområdet *utarmas* på laddningsbärare. Denna nMOS-transistor, för vilken alltså tröskelspanningen  $V_T < 0$ , är av *utarmningstyp* (eng. *depletion type*).



Figur 6.6 nMOS-transistor,  $i_D = f(v_{GS})$ .

Transistorsymbolerna för anrikningstyp och utarmningstyp skiljer sig vad gäller linjen mellan source och drain. För anrikningstypen är linjen bruten, symboliserande att det utan inspänning på styret ( $v_{GS} = 0$ ), inte finns någon kanal mellan source och drain, inga laddningsbärare i kanalområdet, strömvägen mellan source och drain är bruten. För utarmningstypen är linjen hel, symboliserande att det utan inspänning på styret ( $v_{GS} = 0$ ) finns en kanal mellan source och drain, laddningsbärare i kanalområdet, strömvägen mellan source och drain är sluten.

Substratpilen i transistorsymbolen för nMOS-transistorn pekar in mot kanalområdet. Pilen markerar riktningen hos de pn-övergångar (dioder) som utgörs av p-substratet och n-området vid source respektive p-substratet och n-området vid drain. Dessa dioder måste alltid vara försända i backriktningen, dvs.  $V_{SB} \geq 0$  och  $V_{DB} \geq 0$ , så att ingen ström går i substratet (bortsett från backströmmen i dessa dioder). Därför måste en nMOS-transistor med substratet jordat som i figuren 6.6 ovan, ha en positiv matningsspänning.

Substratet får alltså i nMOS-transistorn inte ha högre potential än source och drain. En intressant fråga är vad som händer om substratet har lägre potential än source. Substratet kan betraktas som ett extra styre, som dock har avsevärt mindre inverkan på kanalen än det riktiga styret. Om substratet har negativ potential i förhållande till source, dvs spänningen  $v_{SB}$  mellan source och substrat är positiv, kommer transistorn att strypas hårdare och kräva ett högre värde på  $v_{GS}$  för att ström skall kunna flyta mellan source och drain, vilket är liktydigt med att nMOS-transistorns tröskelspanning  $V_T$  har blivit högre (förflyttats på  $v_{GS}$ -axeln åt höger i positiv riktning). Fenomenet brukar på engelska benämñas "*body-effect*". Tröskelspanningen  $V_T$  för nMOS-transistorn som funktion av spänningen  $v_{SB}$  mellan source och substrat (*body*), kan approximativt skrivas

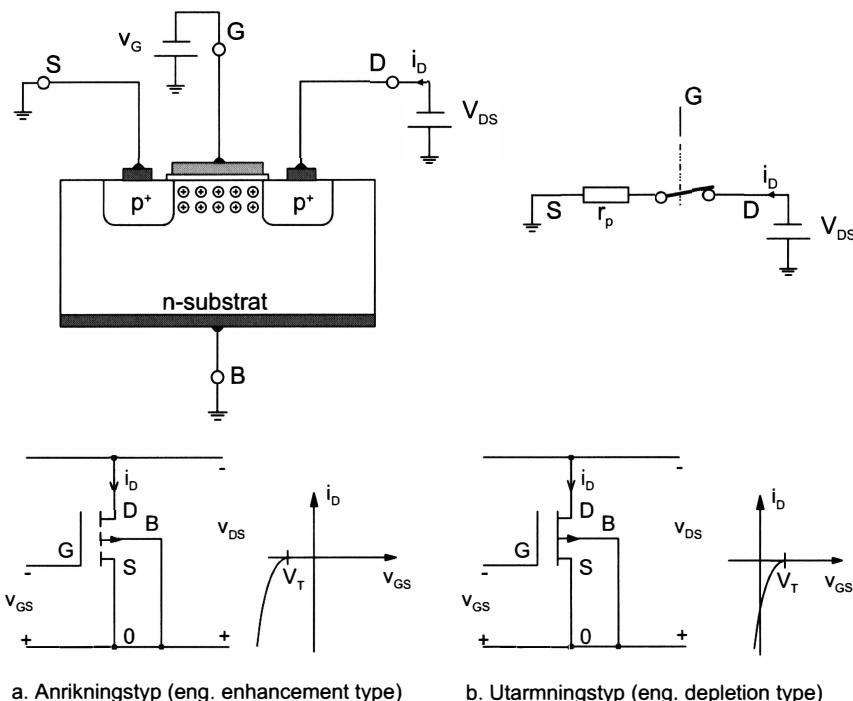
$$V_T \approx V_{T0} + \gamma \cdot \sqrt{v_{SB}} \quad (6.1)$$

där  $V_{T0}$  är tröskelspanningen för  $v_{SB} = 0$ .  $\gamma$  är en transistorparameter som normalt ligger i området 0,3 - 0,7.

pMOS-transistorn kan liksom nMOS-transistorn tillverkas som anrikningstyp respektive utarmningstyp. För att ström skall kunna flyta mellan source och drain i en pMOS-transistor av anrikningstyp måste positiva laddningsbärare, "hål", föras in i kanalen så att det blir en p-kanal mellan de p-dopade source- och drain-områdena. Om liksom för nMOS-transistorn substrat och source jordas, så kommer positiva laddningsbärare att attraheras in

i kanalen med en negativ inspänning på styret,  $v_{GS} < 0$ , se figur 6.7 nedan. Med inspänningen mer negativ än en viss tröskelspanning  $V_T$ , dvs.  $v_{GS} \leq V_T$ , så finns det så många positiva laddningsbärare i kanalen att en ström kan flyta mellan source och drain och switchen är sluten.

Om nu drain läggs på negativ potential så kommer en kollektorström att flyta från source till drain (med positiv strömmirktning markerad från drain till source i figuren, sålunda en negativ kollektorström  $i_D$ ). Ju större negativ inspänning  $v_{GS}$ , desto större kollektorström  $i_D$ , se figur 6.7a, vilket i den enkla modellen för pMOS-transistorn med en resistor i serie med en ideal switch, motsvaras av att resistansen  $r_p$  minskar.



Figur 6.7 pMOS-transistor,  $i_D = f(v_{GS})$ .

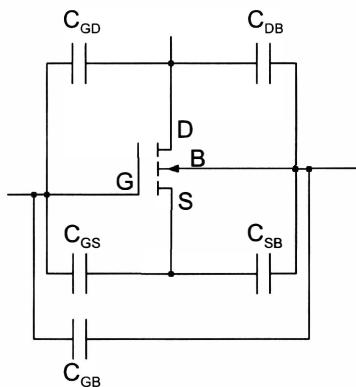
I transistorsymbolen för pMOS-transistorn är substratpilen riktad från kanalområdet ut från substratet. Liksom för nMOS-transistorn markerar pilen riktningen hos de pn-övergångar (dioder) som utgörs av p-området vid

source och n-substratet respektive p-området vid drain och n-substratet, vilka måste vara förspända i backriktningen, dvs.  $V_{SB} \leq 0$  och  $V_{DB} \leq 0$ . Därför måste en pMOS-transistor med substratet jordat ha en negativ matningsspänning.

## Kapacitanser

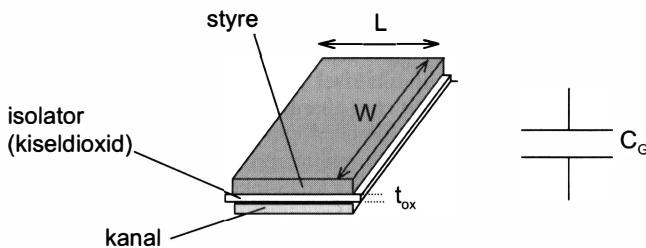
Digitala kretsars snabbhet, förmåga att snabbt slå om från logiskt 0 till logiskt 1 och tvärtom, bestäms av kapacitanserna i kretsen, som vid omslag skall laddas upp eller laddas ur. Kretsens kapacitanser utgörs av kapacitanserna hos transistorerna, ledningarna på chippet, ledningarna mellan kapslarna etc. Upp- och urladdningstid bestäms av strömmens storlek och kapacitansens storlek, ju större ström och mindre kapacitans, desto snabbare upp- och urladdning.

MOS-transistorn har själv många kapacitanser som visas i figur 6.8 nedan. Kapacitanserna  $C_{DB}$  och  $C_{SB}$  är diffusionskapacitanser i de backspända dioderna, pn-övergångarna, mellan substrat och drain respektive mellan substrat och source. Dessa kapacitanser beror inte bara på MOS-transistorns geometriska uppbyggnad och material, utan även på backspänningen i pn-övergången och varierar med denna spänning.



Figur 6.8 Kapacitanser hos en MOS-transistor.

Styret och kanalen kan betraktas som en plattkondensator enligt figuren nedan, där de ledande plattorna utgörs av styret som ju är en ledare och av kanalen som ju också är en ledare då den innehåller laddningsbärare.



Figur 6.9 Styret och kanalen betraktad som en plattkondensator.

En ideal plattkondensator enligt figuren ovan har kapacitansen

$$C_G = W \cdot L \cdot \frac{\epsilon_{ox}}{t_{ox}} \quad (6.2)$$

$\epsilon_{ox}$  [F/m] är dielektricitetskonstanten och  $t_{ox}$  [m] är tjockleken för isolatorn mellan plattorna. För isolatorn kiseldioxid är  $\epsilon_{ox} = 3,97 \cdot \epsilon_o$ . ( $\epsilon_o = 8,84 \cdot 10^{-12}$  [F/m], dielektricitetskonstanten för vakuum).

Styret och kanalen utgör inte en ideal plattkondensator, varför uttrycket (6.2) bara är ett approximativt uttryck för kapacitansen som dock antyder att ju mindre transistorn görs (ju mindre  $W \cdot L$ ), desto mindre blir kapacitansen (desto snabbare transistor). En orsak till att plattkondensatoren inte är ideal är att plattan som utgörs av kanalen inte har samma ledningsförmåga utefter kanalen. Laddningsbärarna är nämligen inte jämnt fördelade, utan är färre vid drain än vid source, varför kapacitansen varierar utefter kanalen.

Kapacitanserna  $C_{GD}$ ,  $C_{GS}$  och  $C_{GB}$  kan approximativt sammanföras till en kapacitans lika med kapacitansen  $C_G$  ovan, dvs  $C_G = C_{GD} + C_{GS} + C_{GB}$ . Noggrannare beräkningar av MOS-transistorers kapacitanser kräver tillgång till simuleringsprogram.

Kapacitanserna hos MOS-transistorerna och de interna metalledningarna på chippen är av storleksordningen 0,01–1 pF.

## Liten historik

MOS-transistorer började kommersiellt tillverkas under mitten av 1960-talet. De första MOS-transistorerna var av typ pMOS med aluminiumstyre. Nackdelarna var att de krävde flera höga matningsspänningar ( $-12\text{ V}$  och  $-27\text{ V}$ ), var långsamma (ca  $100\text{ ns/grind}$ ), hade relativt hög effektförbrukning (0,5 mW/grind) och var svåra att sammankoppla med TTL (eng. *Transistor Transistor Logic*; kretsfamilj uppbyggd med bipolära transistorer).

Nästa steg i utvecklingen blev pMOS med kiselstyre (eng. *silicon gate*), där aluminiumstyret ersattes med ett styre av polykristallinskt kisel (eng. *polysilicon*), som har mycket god ledningsförmåga. Kiselstyret gav jämfört med aluminiumstyret fördelar såsom mindre transistorer, ca 50% bättre packningstäthet, mindre fördräjning (ca  $50\text{ ns/grind}$ ), lägre effektförbrukning (faktor 0,7) och lättare sammankoppling med TTL. – Kiselstyrettekniken, som idag är helt dominerande och en viktig milstolpe i MOS-teknikens utveckling, utvecklades under slutet av 1960-talet av det amerikanska företaget Intel.

Nästa milstolpe blev nMOS med kiselstyre i början av 1970-talet. nMOS kom alltså flera år efter pMOS, beroende på att det tog tid innan man behärskade tillverkningsprocessen. nMOS med kiselstyre gav fördelar såsom bara en matningsspänning (5 V), större snabbhet (ca  $20\text{ ns/grind}$ ), ca 50 % bättre packningstäthet, lägre effektförbrukning (faktor 0,2) och fullt TTL-kompatibel.

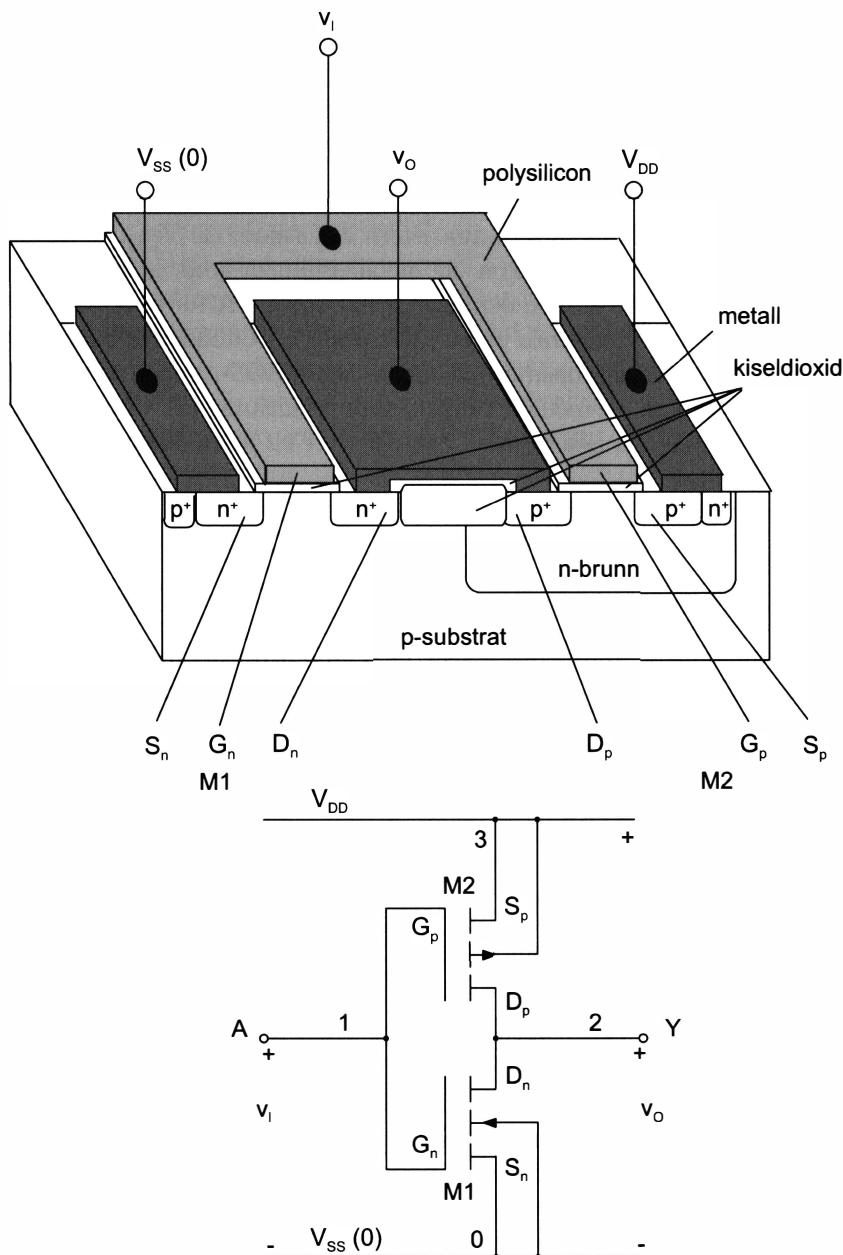
Utvecklingen av MOS-tekniken efter nMOS med kiselstyre karakteriseras i huvudsak av en alltmer förfinad tillverkningsteknik med möjlighet att tillverka allt mindre MOS-transistorer med därmed följande större snabbhet och högre packningstäthet.

Under årens lopp har det i MOS-tekniken dykt upp namn som HMOS, SMOS, XMOS, DMOS, VMOS, etc. Dessa namn står inte för någon ny revolutionerande MOS-teknik jämförbar med pMOS och nMOS, utan är bara att betrakta som olika fabrikanters sätt att förbättra prestanda hos den konventionella MOS-tekniken och profilera sin egna produkter. – Exempelvis står HMOS för High performance MOS, SMOS för Scaled MOS, DMOS för Double-diffused MOS och VMOS för en MOS-transistor med V-format styre.

## 6.2 CMOS-inverteraren

### Struktur

CMOS-inverterarens uppbyggnad visas i figur 6.10 nedan. Namnet CMOS står för *Complementary MOS*, innebärande att två *komplementära* transistorer alltid arbetar tillsammans, en pMOS- och en nMOS-transistor, båda av anrikningstyp. Transistorernas drain är sammankopplade och utgör CMOS-inverterarens utgång. Också transistorernas styren är sammankopplade och utgör CMOS-inverterarens ingång.



Figur 6.10 CMOS-inverteraren.

CMOS-inverteraren ovan är uppbyggd på ett p-substrat, med nMOS-transistorn i p-substratet och pMOS-transistorn i ett i p-substratet nedsänkt n-område, en s. k. *n-brunn* (eng. *n-well*). CMOS-inverteraren kan lika gärna byggas upp precis tvärtom, dvs på ett n-substrat med pMOS-transistorn i n-substratet och nMOS-transistorn i en p-brunn. En fördel med uppbyggnaden ovanpå ett p-substrat, är att då kan nMOS integreras tillsammans med CMOS.

Substrat och source hos pMOS-transistorn är anslutet till  $V_{DD}$ , se transistorstrukturen på chippet, genom att metalledningen som är ansluten till  $V_{DD}$  ligger över både source och n-brunnen, substratet för pMOS-transistorn, och över det sistnämnda på ett mera dopat  $n^+$ -område för bättre kontakt. På samma sätt är substrat och source hos nMOS-transistorn anslutet till  $V_{SS}$  (0), genom att metalledningen, som är ansluten till  $V_{SS}$  (0) ligger över både source och p-substratet, och där över ett mera dopat  $p^+$ -område för bättre kontakt.

## Logisk funktion

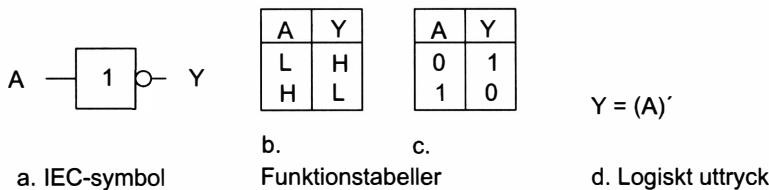
Låt oss nu studera transistorschemat för CMOS-inverteraren i figur 6.10 ovan. – Båda transistorerna är av anrikningstyp, innehärande att för nMOS-transistorn är tröskelspanningen  $V_{Tn} > 0$  och för pMOS-transistorn är tröskelspanningen  $V_{Tp} < 0$ . För insignalen  $v_I = 0$  är den undre transistorn, nMOS-transistorn, strypt, eftersom för denna  $v_{GSn} = 0 < V_{Tn}$ , medan den övre transistorn, pMOS-transistorn, är ledande, eftersom för denna  $v_{GSp} = v_I - V_{DD} = 0 - V_{DD} = -V_{DD} < V_{Tp}$ , och alltså blir utsignalen  $v_O \approx V_{DD}$ .

För insignalen  $v_I = V_{DD}$ , blir det tvärtom, dvs den undre transistorn är ledande, eftersom  $v_{GSn} = V_{DD} > V_{Tn}$ , medan den övre transistorn är strypt, eftersom  $v_{GSp} = v_I - V_{DD} = V_{DD} - V_{DD} = 0 > V_{Tp}$ , och således blir utsignalen  $v_O \approx 0$ . Vi konstaterar sålunda att CMOS-inverteraren *vänder om (inverterar), bildar komplementet till*, insignalen.

I CMOS-inverteraren arbetar nMOS- och pMOS-transistorn som switchar. nMOS-transistorn brukar benämñas *pull-down-transistor*, eftersom den har till uppgift att *dra ned* utsignalen till 0V, medan pMOS-transistorn benämnes *pull-up-transistor*, då den har till uppgift att *dra upp* utsignalen till  $V_{DD}$ .

Vid angivande av funktionen för digitala kretsar brukar symbolerna *L* och *H* användas för potentialerna *Låg* (*Low*) respektive *Hög* (*High*), i detta fall 0V ( $V_{SS}$ ) respektive  $V_{DD}$ . CMOS-inverterarens funktion kan då anges enligt funktionstabellen i figur 6.11b nedan. I figur 6.11a visas logiska symbolen för en inverterare enligt IEC-standard, där ringen på utgången symboliseras invertering.

I digitaltekniken använder man normalt de binära symbolerna 0 och 1. Det finns två sätt att representera potentialerna *L* och *H* med symbolerna 0 och 1, s.k. *positiv logik* med representationen  $L \leftrightarrow 0$  och  $H \leftrightarrow 1$ , (den naturligaste representationen), och *negativ logik* med representationen  $L \leftrightarrow 1$  och  $H \leftrightarrow 0$ . För inverteraren blir uppenbart funktionstabellen med de binära symbolerna 0 och 1 enligt figur 6.11c densamma, oavsett vilken logik som används. Inverteraren realiseras den logiska funktionen *ICKE* (*NOT*), ty ” $Y = 1$  om och endast om  $ICKE(A = 1)$ ”. I figur 6.11d anges logiska uttrycket för inverteraren, där operationen *ICKE* symboliseras med ett primtecken ( $\prime$ ), men som påpekades i kapitel 2 i genomgången av grindarna, så finns det flera andra symboler för operationen *ICKE*.



Figur 6.11 Inverterare.

Även om digitaltekniken till största delen handlar om 0:or och 1:or och logiska symboler, så är det som tidigare påpekats också viktigt att känna till komponenternas fundamentala elektriska egenskaper, såsom spännings- och strömnivåer för in- och utsignaler, fördräjningar, effektförbrukning osv. Vi skall nu studera CMOS-inverterarens *statiska* och *dynamiska* egenskaper. – Vad beträffar de statiska egenskaperna, så har vi ovan bara bestämt utsignalen  $v_O$  för två värden på insignalen, för  $v_I = 0$  V och  $v_I = V_{DD}$ , och skall nu gå vidare och studera överföringskarakteristiken,  $v_O = f(v_I)$  för  $0 \leq v_I \leq V_{DD}$ . När det gäller de dynamiska egenskaperna, så skall vi studera in- och utsignal i tidsplanet, stig- och falltider samt fördräjningar.

## Statiska egenskaper

### Överföringskarakteristiken $v_O = f(v_I)$

I figur 6.12 nedan visas  $v_O = f(v_I)$  och  $i_D = f(v_I)$  för en CMOS-inverterare för  $0 \leq v_I \leq 5$  V och  $V_{DD} = 5$  V samt  $V_{Tn} = 1$  V och  $V_{Tp} = -1$  V.

I schemat för CMOS-inverteraren i figur 6.10 ovan framgår att

$$v_O = V_{DSn}$$

$$v_{GSn} = v_I$$

$$v_{GSp} = v_I - V_{DD} = -(V_{DD} - v_I)$$

$$v_{DSP} = v_O - V_{DD} = -(V_{DD} - v_O)$$

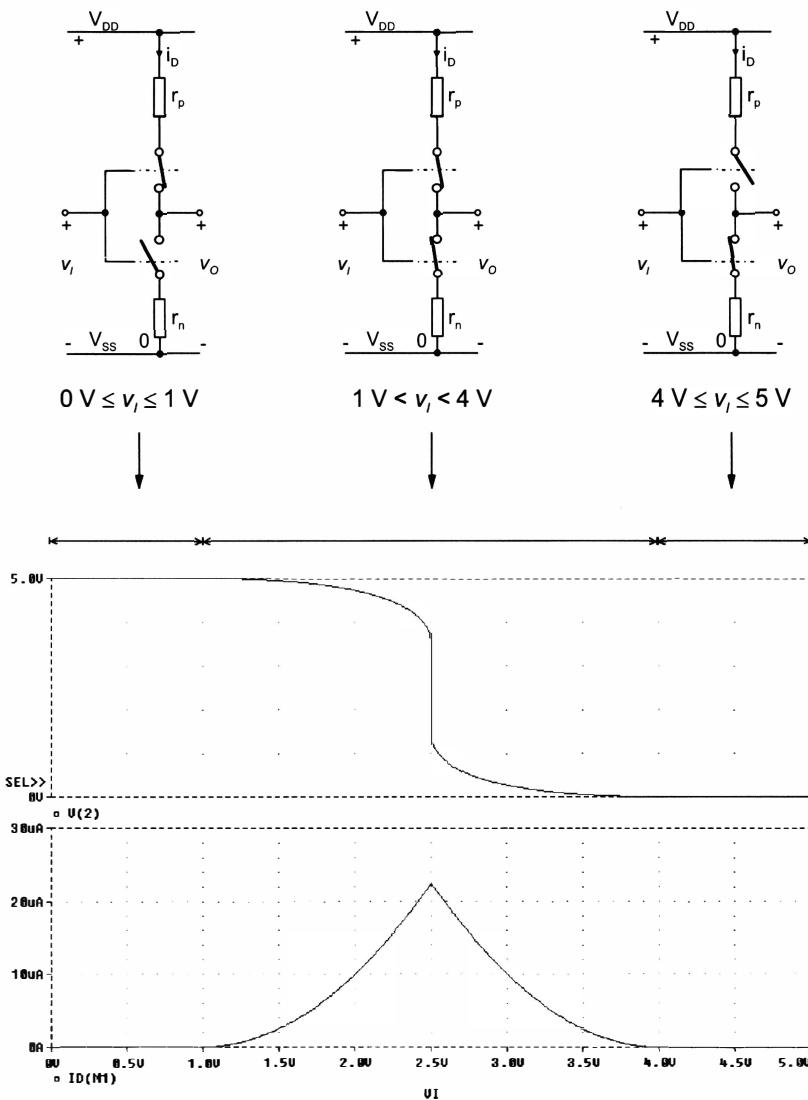
Låt oss nu analysera överföringskaraktersistiken.

$$0 \leq v_I \leq V_{Tn}$$

nMOS-transistorn M1 är strypt, ty  $v_{GSn} = v_I < V_{Tn}$ . pMOS-transistorn M2 leder, ty  $v_{GSp} = -(V_{DD} - v_I) < V_{Tp}$ . Strömmen  $i_D$  från  $V_{DD}$  genom de båda transistorerna till jord utgörs bara av läckströmmen genom nMOS-transistorn, som är mycket liten. Eftersom nMOS-transistorn är strypt och pMOS-transistorn ledande blir utspänningen  $v_O \approx V_{DD}$ .

$$V_{Tn} < v_I < V_{DD} - |V_{Tp}|$$

Då  $v_I > V_{Tn}$ , men nära  $V_{Tn}$ , så är resistansen  $r_n$  i nMOS-transistorn M1 mycket stor och resistansen  $r_p$  i pMOS-transistorn M2 liten, varför spänningssdelningen  $r_p$  i serie med  $r_n$  ger  $v_O \approx V_{DD}$ . När  $v_I$  ökar så minskar  $r_n$  och ökar  $r_p$ , och då  $v_I = V_{DD}/2$  så blir  $r_n = r_p$  och  $v_O = V_{DD}/2$  (förutsatt att nMOS- och pMOS-transistorerna geometriskt utformats för symmetrisk överföringskarakteristik). Ökar sedan  $v_I$  ytterligare mot  $V_{DD}$  så fortsätter  $r_n$  att minska och  $r_p$  öka, och när  $v_I = V_{DD} - |V_{Tp}|$  så stryps pMOS-transistorn och  $v_O \approx V_{SS} = 0$  V.



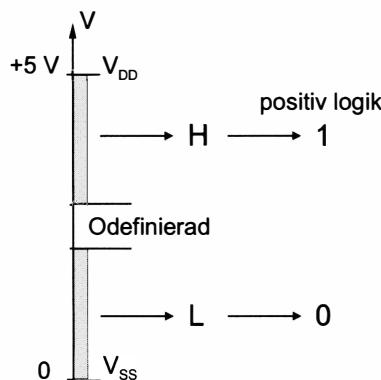
Figur 6.12  $v_O = f(v_I)$  och  $i_D = f(v_I)$  för en CMOS-inverterare – simulering med PSpice för  $0 \leq v_I \leq 5 \text{ V}$  och  $V_{DD} = 5 \text{ V}$  samt  $V_{Th} = 1 \text{ V}$  och  $V_{Tp} = -1 \text{ V}$ . (Kretsfilerna till simuleringarna visas i appendix 3.).

$$V_{DD} - |V_{Tp}| \leq v_I \leq V_{DD}$$

pMOS-transistorn M2 är strypt, ty  $v_{GSp} = v_I - V_{DD} \geq V_{Tp}$ . nMOS-transistorn M1 leder, ty  $v_{GSn} = v_I > V_{Th}$ . Strömmen  $i_D$  från  $V_{DD}$  genom de båda transistorerna till jord utgörs bara av läckströmmen genom pMOS-transistorn, som är mycket liten. Eftersom pMOS-transistorn är strypt och nMOS-transistorn ledande blir utspänningen  $v_O \approx V_{SS} = 0$  V.

## Logiska nivåer

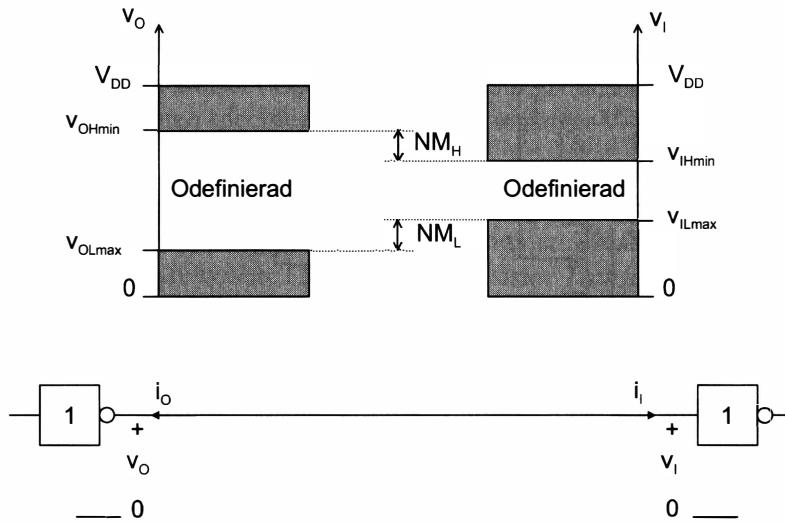
I inledningskapitlet berördes den fundamentala principen för representation med binära signaler, som åter visas nedan.



Figur 6.13 Principen för en binär signal.

Den binära signalen ovan är exemplifierad med en spänning i området  $V_{SS}$  (0) till  $V_{DD}$ , som är kvantiserad i två diskreta intervall avbildade på de binära storheterna L och H, och i positiv logik på de binära siffrorna 0 respektive 1. Intervallen är åtskilda av ett område där signalen inte är definierad, den är varken L eller H.

Vi skall nu analysera förutsättningarna för att utsignalen hos en CMOS-inverterare skall kunna vara insignal till en annan likadan CMOS-inverterare och ge korrekt logisk funktion. Analysen baseras på figur 6.14 nedan.



Figur 6.14 Signalnivåer  $v_O$  och  $v_I$ .

Ur figur 6.14 ovan framgår att *för korrekt logisk funktion måste gälla* (Index OH och OL står för **O**utput **H**igh respektive **O**utput **L**ow):

$$V_{IH\min} < V_{OH\min}$$

$$V_{IL\max} > V_{OL\max}$$

*Störmarginalerna* (eng. *noise margin*)  $NM_H$  och  $NM_L$  definieras:

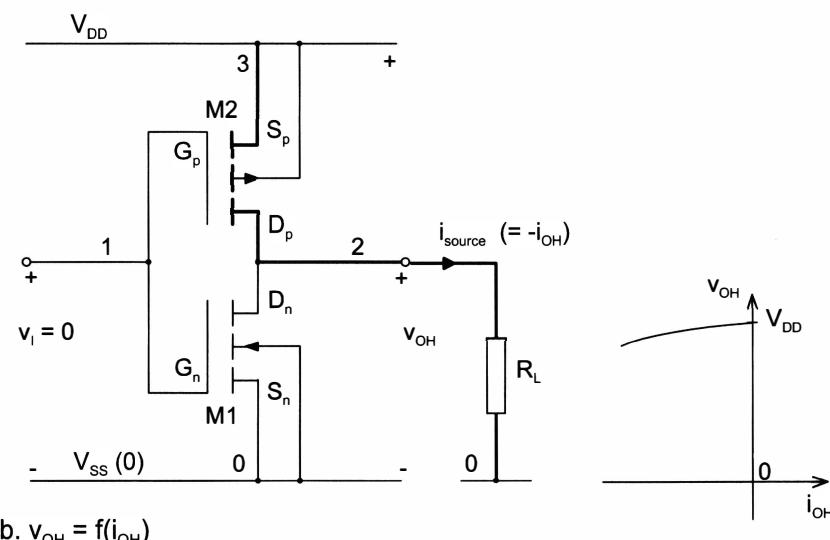
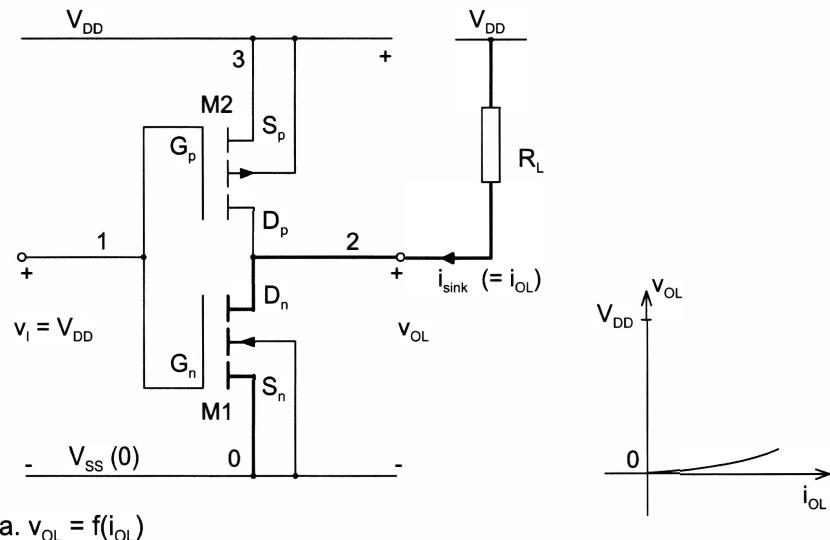
$$NM_H = V_{OH\min} - V_{IH\min}$$

$$NM_L = V_{IL\max} - V_{OL\max}$$

För säker funktion hos systemet är det självklart önskvärt med så stora störmarginaler som möjligt.

För CMOS-inverteraren i figur 6.10 ovan konstaterade vi att  $v_{OH} \approx V_{DD}$  för  $v_I = 0$  V och att  $v_{OL} \approx 0$  V för  $v_I = V_{DD}$ .  $v_{OH} \approx V_{DD}$  och  $v_{OL} \approx 0$  V *gäller bara om CMOS-inverteraren ej är belastad*. Om vid utsignalen Låg hos CMOS-inverteraren, ström *sänks* (eng. *sink*) genom nMOS-transistorn till jord, så ökar spänningen  $v_{DSn}$  över nMOS-transistorn (över  $r_n$  i modellen),

se figur 6.15a. Om vid utsignalen Hög hos CMOS-inverteraren, ström *utmatas* (eng. *source*) från  $V_{DD}$  genom pMOS-transistorn, så ökar spänningen  $v_{DSp}$  över pMOS-transistorn (över  $r_p$  i modellen) och  $v_{OH} = V_{DD} - v_{DSp}$  sjunker, se figur 6.15b.



Figur 6.15  $v_O = f(i_O)$ .

Utspanningarna  $v_{OH}$  och  $v_{OL}$  beror sålunda av belastningen. Om belastningen utgörs av bara CMOS-ingångar, vilka ju är mycket höghögmiga, så är belastningsströmmen mycket liten och  $v_{OH} \approx V_{DD}$  och  $v_{OL} \approx 0$  V – så är normalt fallet mellan grindarna inne på chippet. Vid anslutning av en krets (kapsel) till andra kretsar (kapslar) måste spännings- och strömnivåer beaktas. Kretsfabrikanten specificerar för kretsens utgångar  $V_{OHmin}$  och  $V_{OLmax}$  och för kretsens ingångar  $V_{IHmin}$  och  $V_{ILmax}$ .

Det är naturligtvis önskvärt med standardiserade spänningsnivåer som underlättar sammankoppling av kretsar. Det finns en sådan standard – s.k. *TTL-standard*. Akronymer TTL står för *Transistor Transistor Logic*, namnet på en kretsfamilj i bipolär teknik, som kom fram under mitten av 1960-talet. Den har haft mycket stor betydelse för digitalteknikens utveckling ända fram till mitten av 1980-talet. Kretsfamiljen TTL innehöll en stor mängd standardkretsar, såsom grindar, multiplexrar, avkodare, räknare, skiftregister, adderare. I och med genombrottet för de programmerade kretarna under mitten av 1980-talet, innebärande att en enda programmerbar krets kunde ersätta många TTL-kretsar, så förlorade TTL-familjen sin betydelse. Under TTL-eran var anpassning till TTL-nivåerna viktig för alla kretsfabrikanter, eftersom TTL-kretsar ingick i de flesta digitala system. TTL-nivåerna blev därför standard och kretsfabrikanter angav i sina datablad *TTL-kompatibel* (eng. *TTL compatible*), som tecken på att kretsen kunde sammankopplas med kretsar som uppfyllde kraven för TTL-nivåerna. Även om TTL-kretsar nu inte i någon nämnvärd utsträckning används i nykonstruktioner så lever TTL-standarden kvar.

TTL-nivåerna är:

$$V_{OHmin} = 2,4\text{V}$$

$$V_{IHmin} = 2,0\text{V}$$

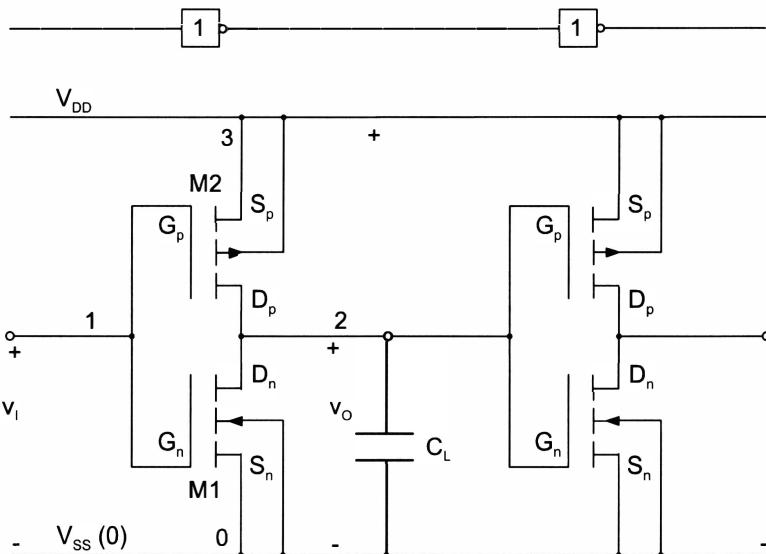
$$V_{OLmax} = 0,5\text{V}$$

$$V_{ILmax} = 0,8\text{V}$$

## Dynamiska egenskaper

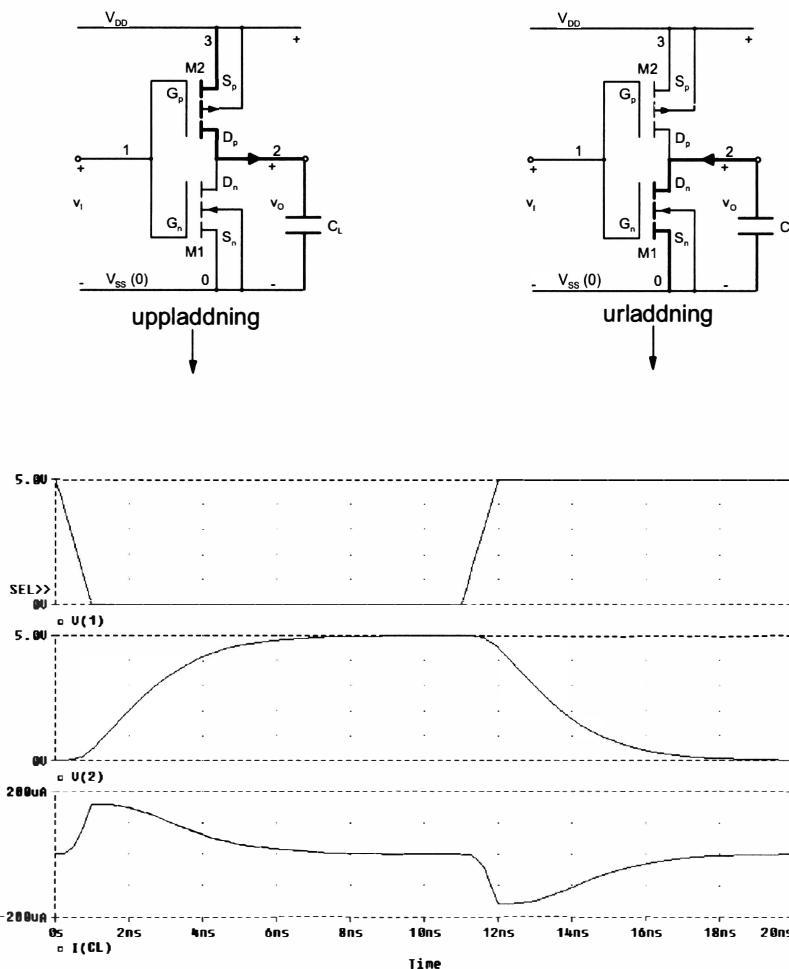
CMOS-inverterarens snabbhet, dvs. hur lång tid det tar för utspänningen  $v_O$  att slå om från Låg till Hög då inspänningen  $v_I$  slår om från Hög till Låg och vice versa är naturligtvis av stor betydelse. Omslagstiderna beror på CMOS-inverterarens förmåga att ladda upp och ladda ur diverse kapacitanser, såsom kapacitanser hos MOS-transistorer och metalledningar, som kort berördes i den tidigare genomgången av MOS-transistorn.

Låt oss analysera omslagstider för en CMOS-inverterare vars utgång är kopplad till en likadan inverterare på chippet enligt figuren nedan.



Figur 6.16 CMOS-inverterare med en belastningskapacitans  $C_L$

För att förenkla analysen av omslagstiderna har alla relevanta kapacitanser, såsom utgångskapacitanserna hos första CMOS-inverteraren, kapacitanserna hos metalledningarna som förbinder inverterarna, ingångskapacitanserna hos andra CMOS-inverteraren, ersatts med en enda *sammanslagen* (eng. *lumped*) kapacitans  $C_L$ , ansluten till den första CMOS-inverterarens utgång.



Figur 6.17 Transientanalys av CMOS-inverterare.

Under CMOS-inverterarens viloläge med utgången Låg eller Hög går endast läckström genom transistorerna och kapacitansen  $C_L$  är urladdad till  $v_{OL} \approx 0V$  eller uppladdad till  $v_{OH} \approx V_{DD}$ . Kapacitansens  $C_L$  storlek har ingen betydelse i viloläget, vilket den dock har vid omslag. När CMOS-inverterarens utgång skall slå om från Låg till Hög skall kapacitansen  $C_L$  uppladdas från  $V_{DD}$  genom pMOS-transistorn och när utgången skall slå

om från Hög till Låg skall kapacitansen  $C_L$  urladdas genom nMOS-transistor till jord.

Viktiga dynamiska parametrar för digitala kretsar är *fördräjning*  $t_p$  (eng. *propagation delay time*) mellan insignal och utsignal, *stigtid*  $t_r$  (eng. *rise time*) och *falltid*  $t_f$  (eng. *fall time*) hos utsignalen.

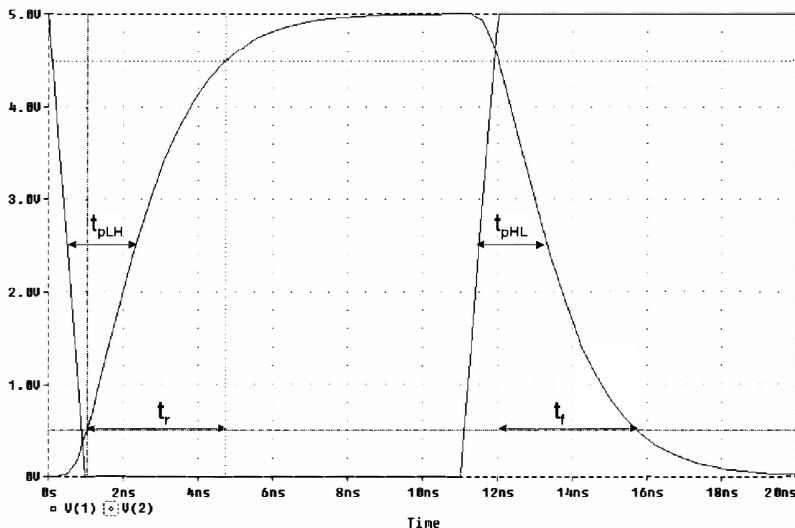
Fördräjningen mäts vid signalernas 50 %-nivå och brukar anges dels för då utsignalen går från Hög till Låg, betecknad  $t_{pHL}$ , dels för då utsignalen går från Låg till Hög, betecknad  $t_{PLH}$ . Stigtid och falltid mäts mellan signalens 10 %-nivå och 90 %-nivå. För transientanalysen i figur 6.17 är dessa tider för tydlighets skull markerade i en separat figur nedan. Tiderna uppmäts till

$$t_{PLH} = 1,8 \text{ ns}$$

$$t_r = 3,7 \text{ ns}$$

$$t_{pHL} = 1,8 \text{ ns}$$

$$t_f = 3,7 \text{ ns}$$



Figur 6.18 Fördräjningar, stigtid och falltid för transientanalysen i figur 6.17.

## Effektförbrukning

CMOS-inverterarens *statiska* effektförbrukning är mycket liten. I viloläge och obelastad går endast läckström genom pMOS- och nMOS-transistorn. Den väsentliga effektförbrukningen är i stället *dynamisk* och orsakas av omslagen från Låg till Hög och tvärtom. Vi har tidigare sett vid studium av CMOS-inverterarens överföringskarakteristik hur pMOS- och nMOS-transistorn samtidigt leder i omslagsområdet och ger upphov till en ström  $i_D$  genom båda transistorerna. Detta är dock bara en mindre del av den dynamiska effektförbrukningen, vars huvuddel i stället härrör från upp- och urladdningarna av belastningskapacitansen  $C_L$ . Denna effektförbrukning  $P$  är lika med

$$P = V_{DD}^2 \cdot C_L \cdot f$$

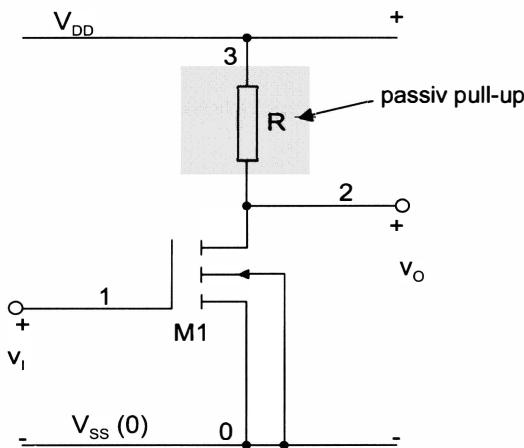
där  $V_{DD}$  är CMOS-inverterarens matningsspänning,  $C_L$  belastningskapacitansen och  $f$  omslagsfrekvensen.

Vi konstaterar alltså att *ju högre omslagsfrekvens, desto högre effektförbrukning*.

## 6.3 nMOS-inverteraren

nMOS-inverteraren, representanten för nMOS-tekniken, innehåller endast en typ av transistorer, nMOS-transistorer, till skillnad från CMOS-inverteraren, som innehåller en nMOS- och en pMOS-transistor. nMOS-inverteraren är uppbyggd av en nMOS-transistor, som arbetar som switch och pull-down-transistor, och en pull-up-komponent. I figur 6.19 nedan visas uppbyggnadsprincipen med en resistor R som *passiv* pull-up-komponent.

### nMOS-inverterare med en resistor som passiv pull-up



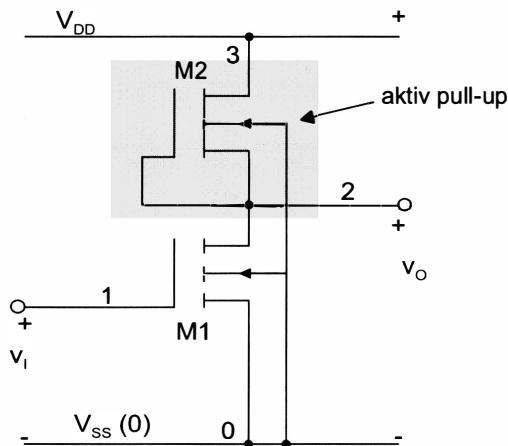
Figur 6.19 nMOS-inverterare med en resistor R som passiv pull-up.

För  $v_I = 0$  är switchtransistorn M1 strypt, endast läckström flyter genom M1, och alltså är utspänningen  $v_{OH} \approx V_{DD}$ . För  $v_I = V_{DD}$  är switchtransistorn M1 ledande och utspänningen  $v_{OL}$  bestäms av spänningsdelningen mellan  $r_n$  och R och med tillräckligt stort värde på R kan utspänningen fås att bli Låg. Kretsen är sålunda en inverterare.

$v_{OLmax}$  beror alltså av storleken på pull-up-resistorn R. En väsentlig skillnad mellan nMOS-inverteraren och CMOS-inverteraren, är att *i viloläge och obelastad flyter i nMOS-inverteraren en icke försumbar ström  $i_D$  vid Låg utgång, medan i CMOS-inverteraren endast flyter läckström.*

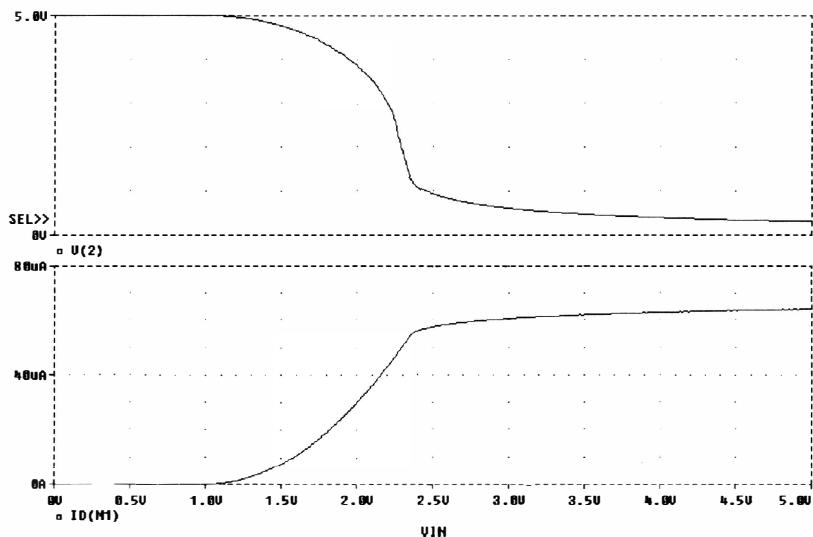
För att begränsa effektförbrukningen i nMOS-inverteraren krävs en pull-up-resistor av storleksordningen  $100\text{ k}\Omega$ . Realisering av en så stor resistor i kisel på chippet kräver en kiselyta mer än hundra gånger större än nMOS-transistorns. Normalt konstrueras därför inte nMOS-inverteraren med en resistor som pull-up, utan i stället med en transistor som aktiv pull-up, en nMOS-transistor av utarmningstyp som i princip uppför sig som en resistor.

### nMOS-inverterare med en transistor som aktiv pull-up

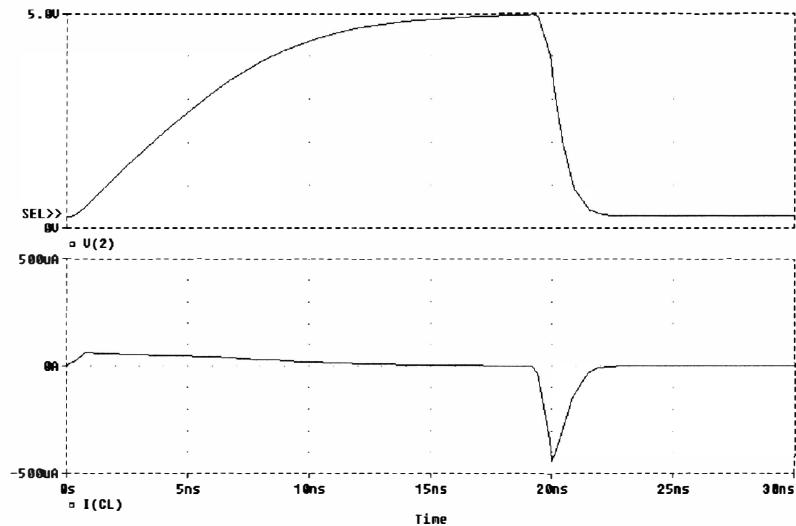


Figur 6.20 nMOS-inverterare med en transistor som aktiv pull-up.

I figur 6.21 och 6.22 nedan visas överföringskarakteristik och transientanalys, simulerad med PSpice, för en nMOS-inverterare med  $V_{Tn1} = 1\text{ V}$  och  $V_{Tn2} = -3\text{ V}$ . nMOS-transistorn av utarmningstyp som aktiv pull-up är alltid ledande eftersom  $v_{GS2} = 0 > V_{Tn2} = -3\text{ V}$ . (Observera att  $V_{Tn2}$  varierar p.g.a *body-effekten*, eftersom substratet (*body*) är anslutet till jord, medan potentialen hos source bestäms av spänningsdelningen mellan M2 och M1, som varierar.  $r_{n2}$  varierar därför med potentialen på source). Vi ser i överföringskarakteristiken att en ström  $i_D$  flyter i nMOS-inverteraren i viloläge vid Låg utgång. Vidare ser vi i transientanalysen att stigtiden är betydligt större än falltiden. Detta beror på att för att  $v_{OLmax}$  skall bli tillräckligt liten, så måste resistansen  $r_{n2}$  göras avsevärt större än resistansen  $r_{n1}$ .



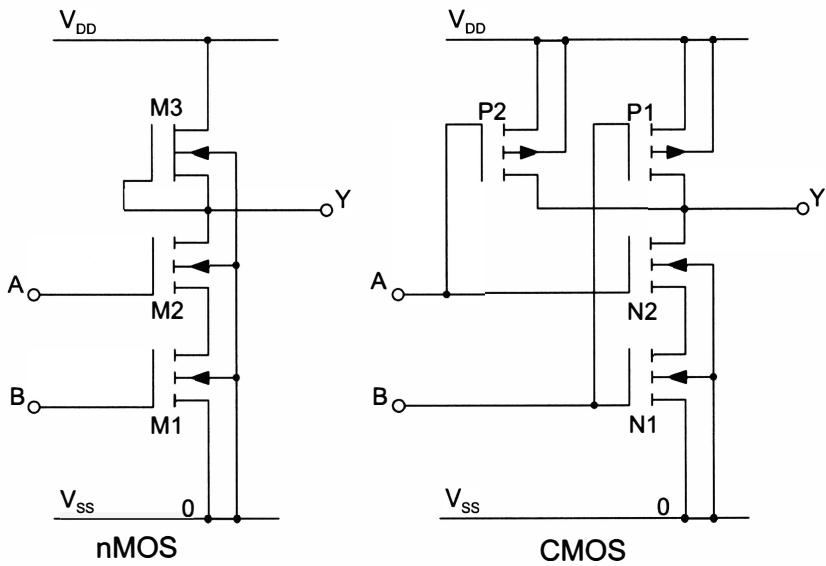
Figur 6.21 Överföringskarakteristik nMOS-inverterare.



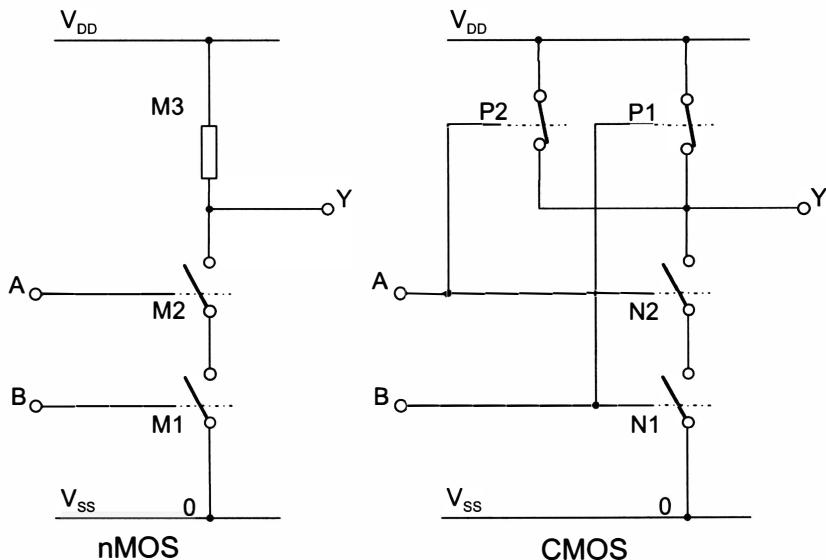
Figur 6.22 Transientanalys nMOS-inverterare.

## 6.4 Grindar i CMOS och nMOS

### NAND-grind



Enkla modeller av grindarna med switchar:



Figur 6.23 NAND-grind i nMOS och CMOS.

A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



$$Y = (AB)'$$

Figur 6.24 Sanningstabell för NAND-grinden i figur 6.23 ovan.

Ur transistorschematic för NAND-grinden framgår att förutsättningen för att utsignalen Y skall vara Hög är att inte båda pull-down-transistorerna M1 och M2 respektive N1 och N2 leder, dvs. att inte båda insignalerna A och B är Höga, eller formellt uttryckt med binära symboler att ” $Y = 1$  om och endast om  $ICKE(A = 1 \text{ OCH } B = 1)$ ”. Kretsen realiseras alltså den logiska operationen ICKE-OCH (eng. NOT-AND, NAND).

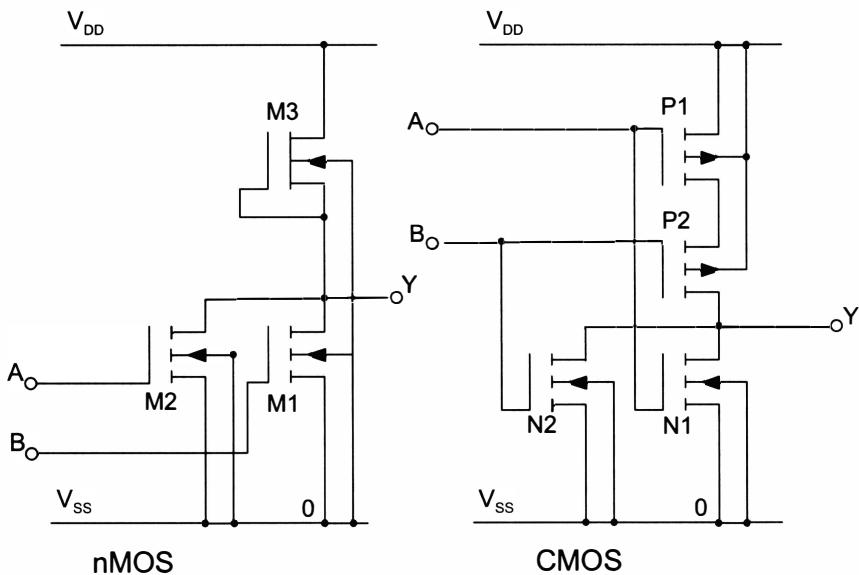
Tidigare har framhållits hur CMOS-inverteraren och nMOS-inverteraren är representativa för realisering av kretsar i CMOS- respektive nMOS-teknik och vi ser här hur realiseringen av NAND-grinden bygger på uppbyggnadsprincipen för respektive inverterare. NAND-grindar med tre eller flera ingångar realiseras enligt samma princip, för varje ny ingång tillfogas i nMOS en ny pull-down-transistor i serie med de tidigare, och i CMOS ett nytt komplementärt transistorpar med pull-down-transistor i serie med de tidigare och pull-up-transistor parallell med de tidigare. Observera alltså för NAND-grinden i CMOS hur pull-down-transistorerna ligger i serie medan pull-up-transistorerna ligger parallellt.

NAND-grindens utimpedans varierar med insignal kombinationerna. Exempelvis blir vid Hög utgång utimpedansen lika med  $r_{P2}$  för insignal kombinationen  $AB = 01$ , medan den blir  $r_{P2} // r_{P1}$  för insignal kombinationen  $AB = 00$ . I figur 6.25 nedan visas utimpedansen för samtliga insignal kombinationer. Utimpedansens variation med insignal kombinationerna medför att omslagsområdets läge varierar och upp- och urladdningstiderna varierar med insignal kombinationerna.

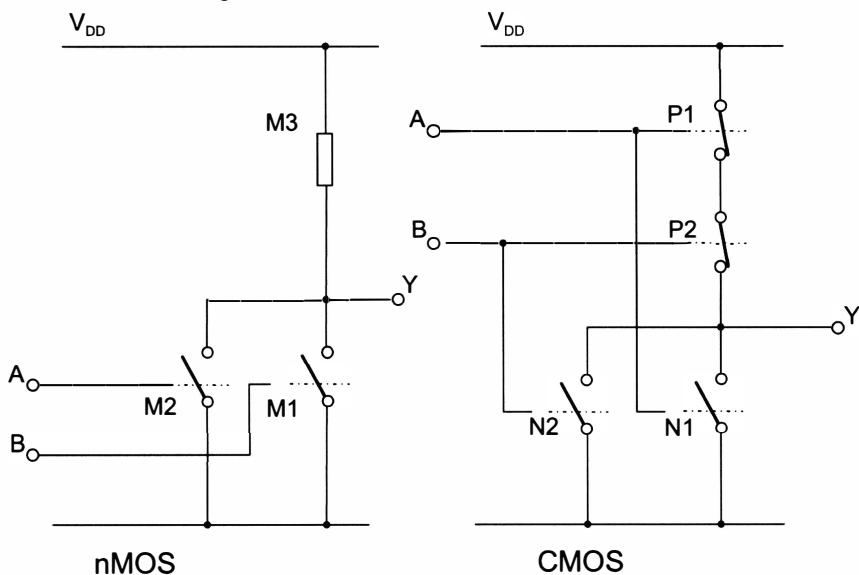
A	B	Y	$r_{ut}$
0	0	1	$r_{P2} // r_{P1}$
0	1	1	$r_{P2}$
1	0	1	$r_{P1}$
1	1	0	$r_{N1} + r_{N2}$

Figur 6.25 Utimpedansen hos NAND-grinden i CMOS för de olika insignal kombinationerna.

## NOR-grind



Enkla modeller av grindarna med switchar:

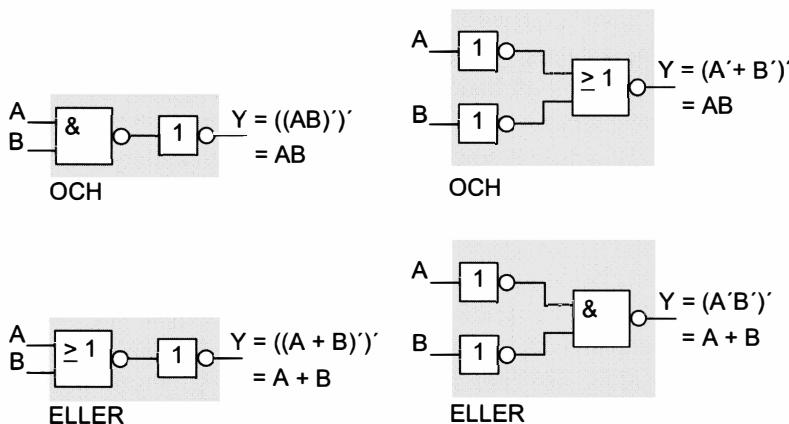


Figur 6.26 NOR-grind i nMOS och CMOS.

Ur transistorschemat för NOR-grinden framgår att utsignalen Y är Hög endast om inte någon av pull-down-transistorerna M1 och M2 respektive N1 och N2 leder, dvs. inte någon insignal A eller B är Hög, eller formellt uttryckt med binära symboler att ” $Y = 1$  om och endast om  $\text{ICKE}(A = 1 \text{ ELLER } B = 1)$ ”. Kretsen realiseras den logiska operationen ICKE-ELLER (eng. NOT-OR, NOR).

## OCH-grind och ELLER-grind

Operationerna OCH och ELLER är mer komplexa att realisera med transistorer än NAND och NOR. Detta beror på transistorns naturliga egenskap att invertera, som utnyttjas vid realiseringen av NAND och NOR, men som måste kompenseras med extra inverterare vid realiseringen av OCH och ELLER. Vi har redan i kapitlen 2 och 3 sett hur OCH och ELLER kan realiseras med NAND, NOR och inverterare enligt figur 6.27 nedan

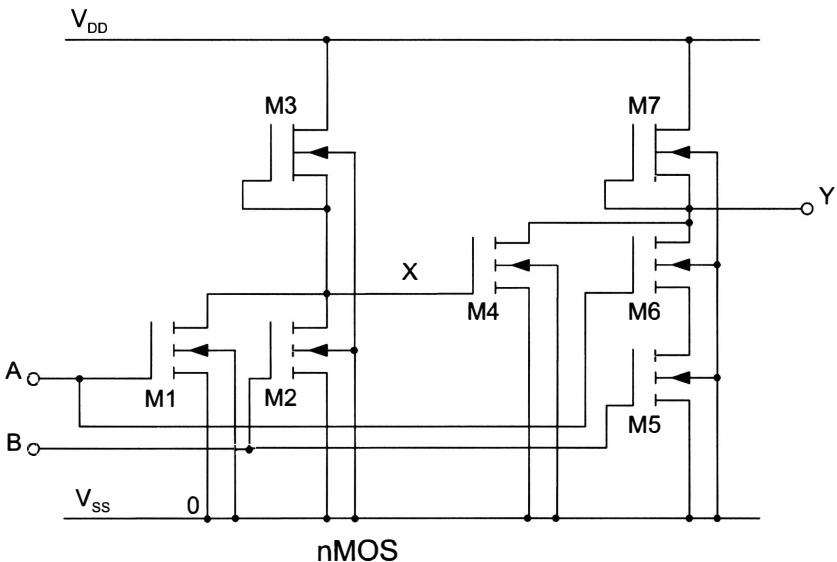


Figur 6.27 Realisering av OCH-grind och ELLER-grind.

Vi har också i kapitel 4 sett hur grindnät med strukturen OCH-ELLER är ekvivalenta med grindnät med strukturen NAND-NAND och hur grindnät med strukturen ELLER-OCH är ekvivalenta med grindnät med strukturen NOR-NOR. Realisering av logiska funktioner utförs därför så gott som uteslutande med NAND- och NOR-grindar. Även om fabrikanterna anger

funktionsschema med OCH- och ELLER-grindar, så är det normalt bara en beskrivning av den logiska funktion som realiseras, och inte hur den realiseras med grindar, vilket sannolikt är med NAND- och NOR-grindar.

## Exklusivt-ELLER-grind, XOR-grind



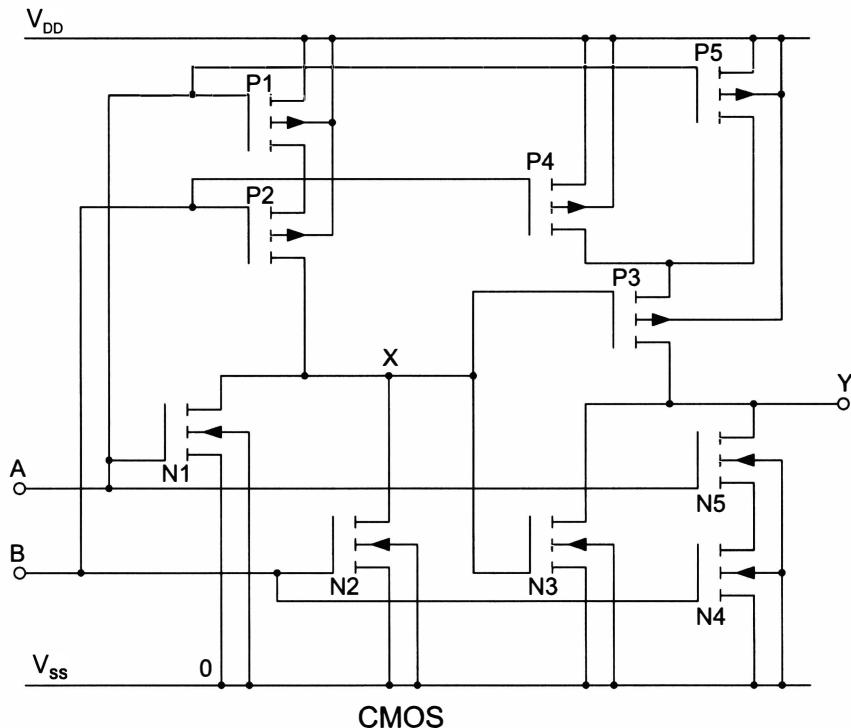
A	B	Y
L	L	L
L	H	H
H	L	H
H	H	L

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{array}{l} \text{A} \xrightarrow{=} 1 \\ \text{B} \end{array} \quad \text{Y}$$

$$Y = A \oplus B$$

Figur 6.28 XOR-grind i nMOS.

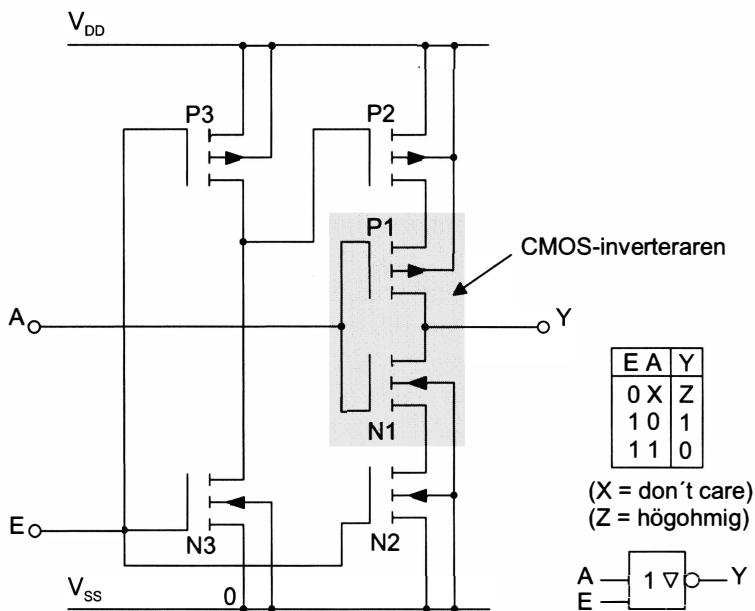


Figur 6.29 XOR-grind i CMOS.

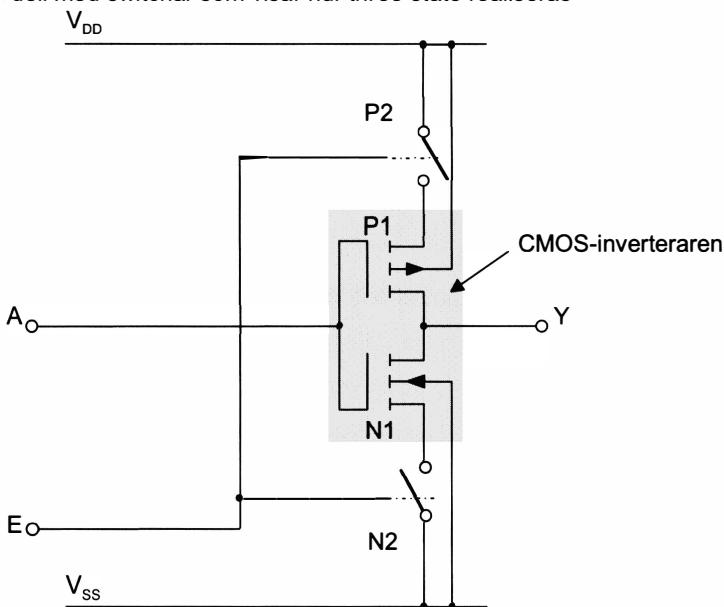
Vi ser att operationen XOR är relativt komplex att realisera i nMOS och CMOS. Analysen av de två XOR-grindarnas transistorscheman i figurerna 6.28 och 6.29 ovan lämnas till ett par övningsuppgifter.

## Three-state-utgång

Grindarna som hittills realiseras i nMOS och CMOS har haft två utgångsvärden (utgångstillstånd), Låg eller Hög. Många digitala kretsar har, som redan nämntes i kapitel 2, utgångar som förutom dessa två utgångstillstånd även kan anta ett tredje tillstånd, då den varken är Låg eller Hög, utan är *höghohmig, frisvävande*. Kretsen har alltså en utgång med tre utgångstillstånd (eng. *three-state output*).



Modell med switchar som visar hur three-state realiseras



Figur 6.30 CMOS-inverterare med three-state-utgång.

I figur 6.30 ovan visas hur en three-state-utgång kan realiseras i CMOS. Principen illustreras för en CMOS-inverterare, men kan användas vilken CMOS-krets som helst.

Själva CMOS-inverteraren består av transistorerna N1 och P1. Three-state-funktionen åstadkoms av transistorerna N2 och P2, vilka båda samtidigt antingen är Från (öppna), innebärande att inverterarens utgång då varken får förbindelse med Hög ( $V_{DD}$ ) eller Låg ( $V_{SS}$ ), utan blir frisvävande, eller Till (slutna) då inverterarens utgång kan anta Hög eller Låg. Three-state-funktionen styrs av insignalen E (eng. *Enable*). Som framgår av funktions-tabellen i figuren ovan, så är utgången högohmig (Z) när  $E = 0$ , ty då leder uppenbart inte N2 och heller inte P2, ty via inverteraren P3-N3 inverteras E och ger en Hög insignal till styret på P2. Inverteraren P3-N3 åstadkommer alltså att P2 är sluten när N2 är öppen och öppen när N2 är sluten.

## 6.5 Övningsuppgifter

### 6.4 Grindar i CMOS och nMOS

- 6.1** Fyll i funktionstabellerna nedan för XOR-grindarna i figurerna 6.28 och 6.29. Skriv i kolumnerna för transistorerna, T för Till och F för Från och i kolumnerna för signalerna X och Y, 0 eller 1.

a) XOR-grinden i figur 6.28

A	B	M1	M2	X	M4	M5	M6	Y
0	0							
0	1							
1	0							
1	1							

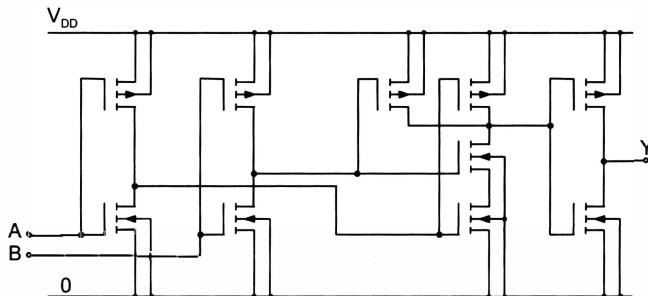
b) XOR-grinden i figur 6.29

A	B	N1	P1	N2	P2	X	N3	P3	N4	P4	N5	P5	Y
0	0												
0	1												
1	0												
1	1												

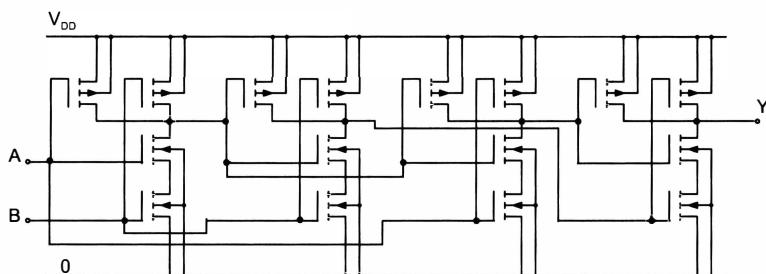
- 6.2** Rita transistorschema i nMOS för kretsar som realiseras logiska uttrycken
- not((A and B) or C).
  - (A and B) or C
  - not((A or B) and C))
- 6.3** Rita transistorschema i CMOS för kretsar som realiseras logiska uttrycken i uppgift 6.2.

**6.4 a)** Bestäm booleska uttrycket till utsignalen Y. Vilken grind realiseras?

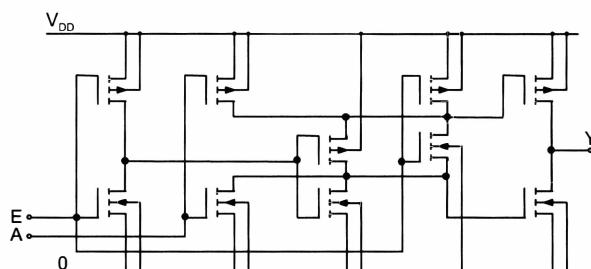
**b)** Grinden är realiserad på ett speciellt sätt. Vilket och varför?



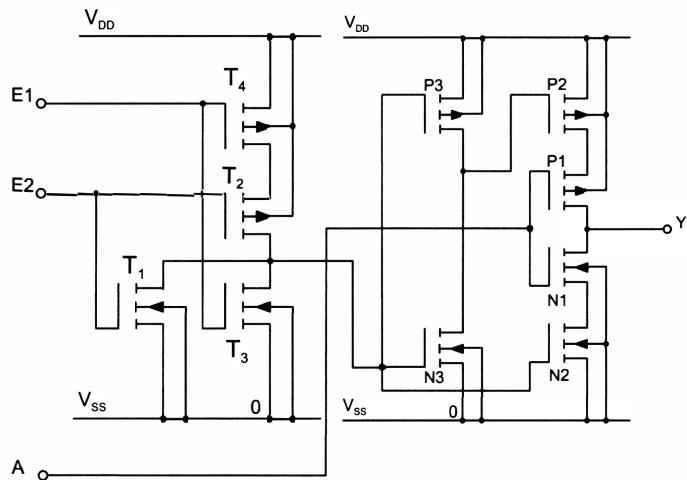
**6.5** Bestäm booleska uttrycket till utsignalen Y. Vilken grind realiseras? Vilken princip har använts vid realiseringen?



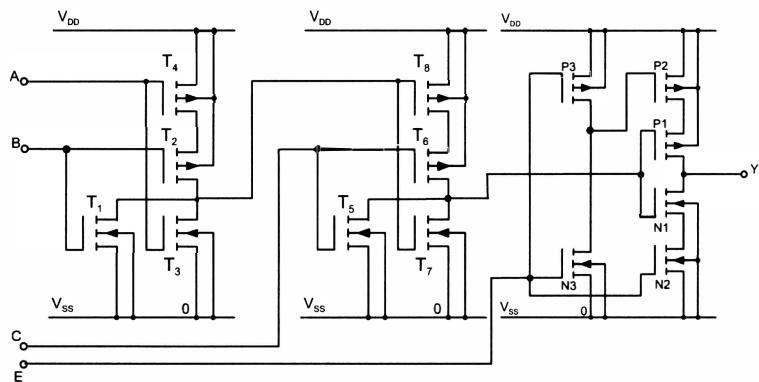
**6.6** Bestäm funktionstabellen till CMOS-kretsen nedan.



**6.7** Bestäm funktionstabellen till CMOS-kretsen nedan.



**6.8** Bestäm funktionen hos CMOS-kretsen nedan.



# 7 Halvledarminnen

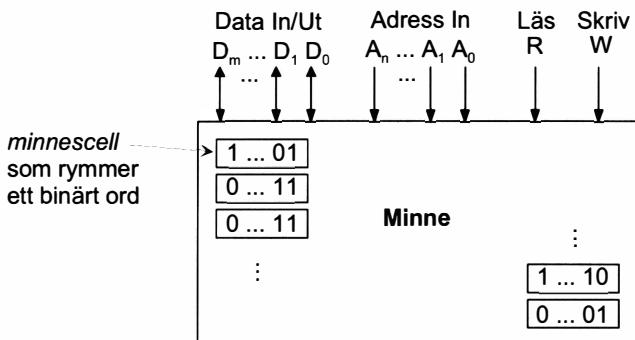
Kretsar med minnesfunktion har vi redan sett i form av vippor och latchar i kapitlen 2 och 5. Vippan och latchen är var och en ett litet minne i vilket kan lagras en bit. De kan sättas samman till ett minne med flera bitar, exempelvis till ett tillståndsregister för lagring av tillståndet i en sekvenskrets eller till ett generellt register för lagring av ett ord. Dessa minnen är mycket små minnen med en lagringskapacitet från några enstaka bitar till något tio-tal bitar och brukar inte benämñas minne utan i stället register. I detta kapitel skall vi studera egentliga minnen för lagring av tusentals, miljontals bitar. Enbart halvledarminnen kommer att behandlas. Det finns även andra viktiga klasser av minnen, såsom exempelvis magnetiska minnen typ hårddiskar. Framställningen inleds med en enkel minnesmodell och en översikt av halvledarminnen. Därefter följer en mer ingående behandling av olika typer av halvledarminnen.

## 7.1 Inledning – minnesmodell och en översikt av halvledarminnen

Den huvudsakliga användningen av halvledarminnen är i datorer. Persondatorernas enorma utveckling sedan introduktionen i början av 1980-talet, beror i lika hög grad på halvledarminnens som på mikroprocessorernas utveckling.- När det gäller datorer så tänker man väl först på *generella datorer*, typ persondatorer, arbetsstationer etc., som används för körsning av diverse olika program. Men det finns också en annaniktig klass av datorer, som omger oss överallt och som vi dagligen umgås med utan att tänka på det, datorer *inbyggda* (eng. *embedded*) i TV-apparater, videobandspelare, mobiltelefoner, bilar, diskmaskiner, tvättmaskiner, mikrovågsugnar, mätnstrument m.m. Dessa datorer, som benämnes *mikrostyrkretsar* (eng. *microcontroller*), är dedicerade för en specifik tillämpning och kör i huvudsak hela tiden ett och samma program under hela sin livslängd. Halvledarmin-

nenas utveckling har haft stor betydelse för också dessa datorers utveckling. I den inbyggda datorn där programmet skall ligga kvar ständigt, måste halvledarminnet kunna hålla kvar informationen även utan matningsspänning när datorn är avstängd, vilket inte behöver vara fallet i den generella datorn där det aktuella programmet vid en körning laddas in i halvledarminnet från ett annat lagringsmedium, t.ex. en hårddisk. Det ställs alltså olika krav på halvledarminnet beroende på tillämpningen och det finns en hel flora av olika halvledarminnen som vi skall se i det följande. Innan vi berör dessa olika typer ger vi en enkel minnesmodell, som diskussionen kan baseras på.

## Minnesmodell



Figur 7.1 Minnesmodell.

Ett minne är en ”enhet i vilken data kan lagras och från vilken data kan hämtas”. – Minnet i minnesmodellen i figuren ovan består av ett antal lika stora *celler*, vardera rymmande ett binärt *ord*. Data lagras i minnet i form av binära ord, som kan överföras till/från minnet på dataledningarna  $D_0$ ,  $D_1$ , ...,  $D_m$ , som antas dubbelriktade, dvs. de kan användas både som ingång och utgång. Varje minnescell har en bestämd adress och via adressledningarna  $A_0$ ,  $A_1$ , ...,  $A_n$ , *adresseras, utpekas*, den aktuella minnescellen. Med exempelvis 16 adressledningar  $A_0$ ,  $A_1$ , ...,  $A_{15}$ , är det möjligt att adressera  $2^{16} = 65536$  olika minneceller. I minnessammanhang brukar man beteckna  $2^{10} = 1024 = 1$  kilo = 1 k och sålunda kan då  $2^{16} = 65\,536$  i stället komprimerat skrivas 64 k, eftersom  $2^{16} = 2^6 \cdot 2^{10} = 64$  k. På samma sätt

brukar  $2^{20}$  ( $= 2^{10} \cdot 2^{10} = \text{kilo} \cdot \text{kilo}$ ) betecknas Mega (M) och  $2^{30}$  ( $= 2^{10} \cdot 2^{20} = \text{kilo} \cdot \text{Mega}$ ) betecknas Giga (G). Exempelvis med 32 adressledningar  $A_0, A_1, \dots, A_{31}$ , är det möjligt att adressera  $2^{32} = 2^2 \cdot 2^{30} = 4\text{G}$  olika minnesceller. I januari 1999 fastställdes det internationella standardiseringssorganet IEC (International Electrotechnical Commission) nya prefix. Om man avser de i datasammanhang vanliga prefix som bygger på multipler av två, rekommenderas *kibibyte* (förkortat *KiB*), *kibibit* (*Kibit*), *mebibibyte* (*Mibyte*), *mebibit* (*Mibit*), *gibibibyte* (*Gibyte*) respektive *gibibit* (*Gibit*). Ledet *bi* i dessa prefix ska ses som en förkortning av *binary* (*kibi* = "kilobinary" osv.). Närmare info om dessa och andra datatermer kan fås på Svenska datatermgruppens hemsida <http://www.nada.kth.se/dataterm/>.

I minnessammanhang används det engelska ordet *byte* som beteckning för en grupp av åtta bitar (egentligen betyder ordet *byte* "bitgrupp"). Kapaciteten hos ett minne brukar anges i *byte* och detta oavsett ord längden i minnet. Antag exempelvis att ett minne är organiserat med 32 bitars ord och att minnet har 32 dataledningar  $D_0, D_1, \dots, D_{31}$  samt att varje minnescell rymmer 32 bitar. Om detta minne har kapaciteten 2 Mega 32-bitars ord, så anges det normalt i *byte* som 8 Mbyte.

Minnets naturliga plats är sålunda i en dator och det kan vara motiverat att kort rekapitulera den enkla datormodellen som tidigare berördes i samband med ALU:n i kapitel 4. Modellen underlättar förståelsen av principen för hur data läggs in i och hämtas ut från minnet.

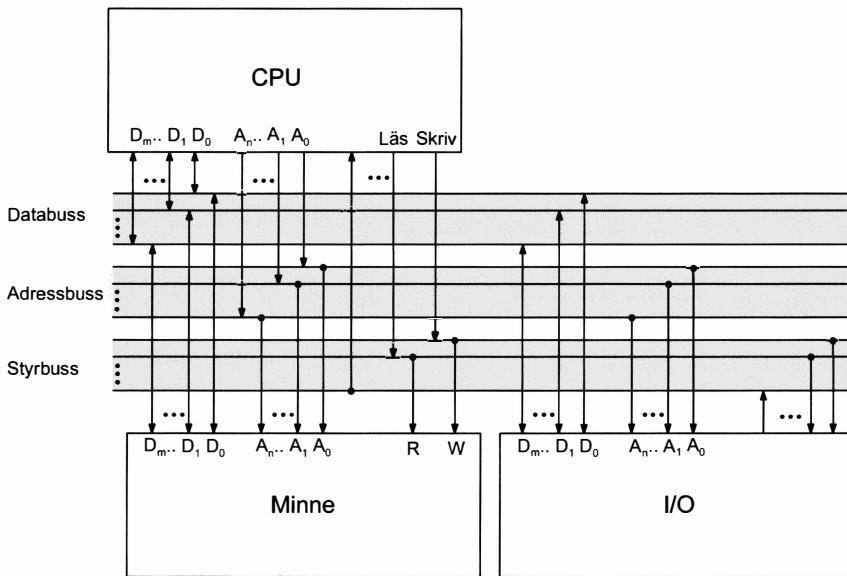
En *dator* består primärt av

- Centralenhets, CPU (eng. *Central Processing Unit*)
- Minne
- In/Ut-enheter, I/O (eng. *Input/Output units*)

I minnet lagras *program* och *data*. CPU:n, som också ofta benämnes *mikroprocessor*, exekverar programmet, utför informationsbehandlingen. Programmets minsta delar är *instruktioner* som CPU:n utför. En instruktion består av en *operation* och en *operand*, som ger CPU:n instruktion om vilken operation den skall utföra på operanden. CPU:n kan bara utföra mycket enkla, primitiva operationer, typ addera, subtrahera, flytta data.

Datorn byggs upp kring och kommunickerar via tre *bussar* (eng. *bus*), *databussen*, *adressbussen* och *styrbussen*, se figur 7.2 nedan. Databussen och adressbussen används som namnen säger, för överföring av data respektive adresser. Vanlig bredd på databussen är 8, 16 eller 32 bitar och på adressbussen 16, 24 eller 32 bitar. Styrbussen (eng. *Control bus*) överför diverse

styrsignaler mellan de olika enheterna. CPU:n hämtar ut, *läser*, ord från minnet och matar in, *skriver*, ord till minnet. I figur 7.3 och 7.4 nedan visas i tidsplanet principen för en *läsning* respektive *skrivning*.



Figur 7.2 Datormodell.

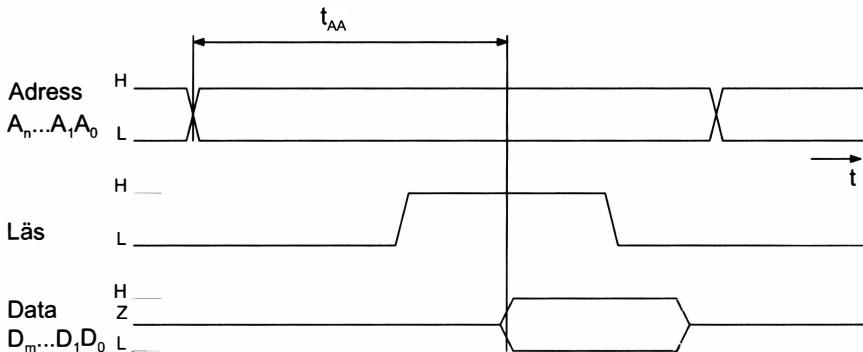
En *läsning* går till så att CPU:n först lägger ut adressen på adressbussen varvid den aktuella minnescellen adresseras, därefter ger CPU:n styrsignalen *Läs* på styrbussen, som verkställer läsningen i minnet och öppnar minnets three-state-buffert på datautgångarna och det adresserade ordet i minnet kommer ut på databussen varifrån CPU:n hämtar in ordet i något av sina register.

En *skrivning* går till så att CPU:n liksom vid en läsning börjar med att lägga ut adressen på adressbussen och adresserar den aktuella minnescellen i vilken data skall skrivas, lägger därefter ut ordet som skall skrivas in i minnet på databussen och ger sedan styrignalen *Skriv* på styrbussen, som verkställer skrivningen i minnet.

En viktig parameter för ett minne är snabheten. I ett datablad för ett halvledarminne anges ett flertal olika tider, varav speciellt intressanta är *lästdiden* och *skrivtiden*. Lästdiden, vanligen benämnd *accesstid*, *åtkomsttid*,

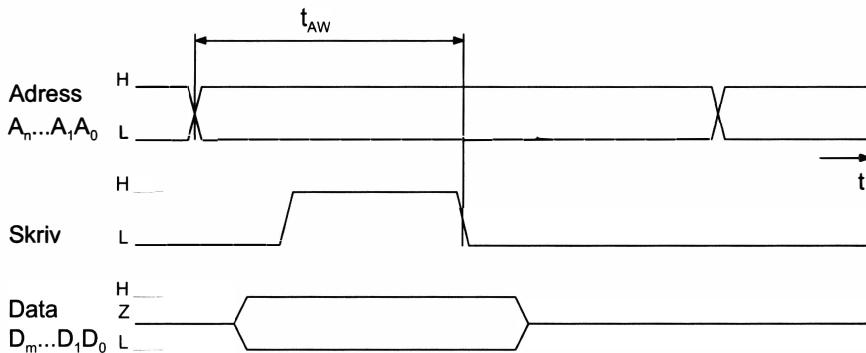
är i figur 7.3 nedan angiven som *adressaccesstid*  $t_{AA}$ , tiden från adressering av minnet tills data från den adresserade minnescellen finns tillgänglig på databussen. Skrivtiden är i figur 7.4 nedan angiven som *adressskrivtid*  $t_{AW}$ , tiden från adressering av minnet tills data är inskriven i den adresserade minnescellen. Intressant ur snabbhetssynpunkt är också hur snabbt på varandra följande läsningar respektive skrivningar kan göras. Detta anges av

## Läsning



Figur 7.3 Läsning från minnet.

## Skrivning



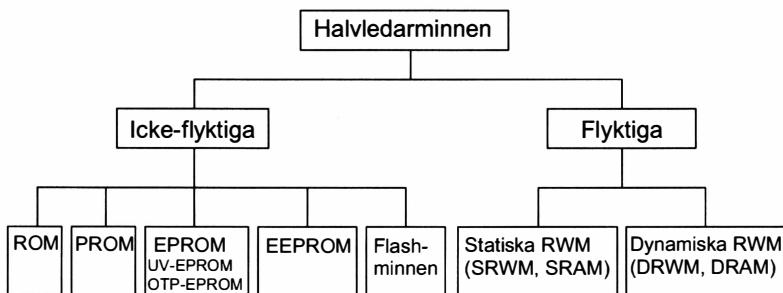
Figur 7.4 Skrivning till minnet.

*läscykletiden*  $t_{RC}$  (eng. *read cycle time*), definierad som minsta tillåtna tid mellan två på varandra följande läsningar, respektive *skrivcykletiden*  $t_{WC}$  (eng. *write cycle time*), definierad som minsta tillåtna tid mellan två på var-

andra följande skrivningar. Uppenbart måste gälla att  $t_{RC} \geq t_{AA}$  och  $t_{WC} \geq t_{AW}$  – Styrsignalerna *Läs* och *Skriv* i figurerna ovan är aktivt höga, ofta brukar de vara aktivt låga.

## Klassificering av halvledarminnen

Vi har redan berört att inbyggda datorer och generella datorer har olika krav på halvledarminnet som skall lagra programmet, vad avser minnets förmåga att behålla informationen utan matningsspänning, och detta leder fram till indelningen av halvledarminnena i klasserna *icke-flyktiga* (eng. *nonvolatile*) halvledarminnen och *flyktiga* (eng. *volatile*) halvledarminnen. Flyktigheten avser alltså minnets förmåga att *kvarhålla* (eng. *retain*) informationen utan matningsspänning och ett icke-flyktigt minne är sålunda ett minne som kvarhåller informationen utan matningsspänning, medan ett flyktigt minne förlorar informationen när matningsspänningen slås av. En naturlig fråga är då varför inte alla halvledarminnen görs icke-flyktiga. Svaret är att man inte klarar av att göra dessa minnen lika snabba som flyktiga minnen. Lästiden går att få lika bra, i området nanosekunder, medan skrivtiden blir relativt sett lång och ligger i området millisekunder. På grund av den jämfört med lästiden långa skrivtiden används de icke-flyktiga halvledarminnena huvudsakligen för läsning av permanent information, såsom *läsminnen*, där skrivning görs i förhållande till läsning sällan eller kanske aldrig.



Figur 7.5 Klassificering av halvledarminnen.

## Icke-flyktiga minnen

Läsminnena kan klassificeras med avseende på möjligheten och sättet för användaren att skriva in information i minnet. – Läsminnen med permanent information där det inte är möjligt att skriva in ny information, benämnes *ROM* (eng. *Read Only Memory*). Informationen i ett sådant minne skrivs in av halvledarfabrikanten vid tillverkningen och minnesinnehållet definieras av förbindningsstrukturen på chippet. Ett läsminne där användaren själv kan skriva in informationen en gång och sedan bara läsa informationen, benämnes *PROM* (eng. *Programmable Read Only Memory*), och även i detta minne definieras minnesinnehållet av förbindningsstrukturen på chippet.

Nästa klass av läsminnen är sådana där användaren inte bara kan skriva in informationen en gång, utan även kan radera (eng. *erase*) informationen och skriva in ny information. – I *EPROM* (eng. *Erasable Programmable Read Only Memory*), introducerade i början av 1970-talet, kan användaren radera informationen genom bestrålning av chippet med UV-ljus genom ett litet fönster i kapseln under ca 20 minuter. Inskrivning av informationen sker med kapseln i en speciell programmeringsutrustning. Minnesinnehållet ligger på chippet som elektriska laddningar. I många tillämpningar används EPROM på samma sätt som PROM, dvs. de programmeras bara en gång och raderas aldrig, och därför tillverkas en variant av EPROM utan fönster, s.k. *OTP-EPROM* (eng. *One Time Programmable EPROM*), som blir billigare än UV-EPROM, dels därför att det inte behövs något fönster, dels därför att det då går att använda en plastkapsel i stället för en keramikkapsel som måste användas när man har ett fönster för att man skall få ungefär samma utvidgningskoefficient för kapselmaterialet som för glaset. I *EEPROM* (eng. *Electrical Erasable PROM*), introducerade i slutet av 1970-talet, kan informationen raderas elektriskt på byte-nivå och ny byte skrivas in, med kapseln sittande i den aktuella tillämpningen. I *Flash-minnen*, introducerade i mitten av 1980-talet, kan informationen också raderas elektriskt, men här hela kapselinnehållet på en gång, och ny information skrivas in med kapseln sittande i den aktuella tillämpningen.

## Flyktiga minnen

Snabba halvledarminnen i området nanosekunder i vilka det går lika snabbt att skriva som att läsa går bara att tillverka som flyktiga minnen. De benämnes lämpligen *läs/skriv-minnen*, *RWM*, (eng. *Read Write Memory*), men vanligen benämnes de *RAM* (eng. *Random Access Memory*), som

betecknar ett minne sådant att för slumpmässigt (eng. *random*) valda adresser erhålls alltid samma accesstid (jfr en hårddisk, som inte är ett RAM, eftersom för denna accesstiden varierar och beror på var huvudena befinner sig i förhållande till den adresserade informationen).

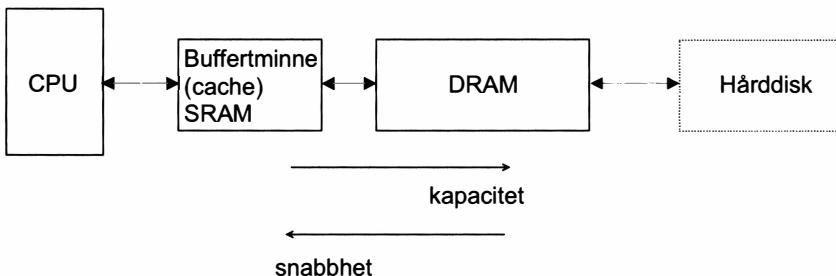
Läs/skriv-minnen, RWM, förekommer av två slag, *statiska RWM (SRWM, SRAM)* och *dynamiska RWM (DRWM, DRAM)*. I ett dynamiskt RWM lagras bitinformationen som en laddning i en kondensator. Efter en tid, ett antal millisekunder, laddas kondensatorn ur och informationen försvinner. För att inte minnesinnehållet skall gå förlorat måste det därför med jämma mellanrum uppförskas (eng. *refresh*), ungefär var 50:e ms, som sker genom att informationen läses och skrivs in igen. Nackdelen med dynamiska RWM är att det krävs speciella kretsar för att ombesörja uppförskningen, däremot är tiden för uppförskningen då minnet inte kan användas, inget större problem, då denna tid är kort och utgör mindre än en procent av tiden mellan uppförskningarna.

I ett statiskt RWM lagras bitinformationen i en latch, innebärande att informationen inte dör ut med tiden som i ett dynamiskt RWM. Fördelen med ett statiskt RWM är att man inte behöver uppförskningskretsar och att det är snabbare än ett dynamiskt RWM. En naturlig fråga är då varför inte alla läs/skriv-minnen, RWM, görs statiska. Svaret är att bitcellen i ett *dynamiskt RWM* kräver bara en transistor, medan bitcellen i ett statiskt RWM kräver 4-6 transistorer, medförande att dynamiska RWM kan tillverkas med mycket större kapacitet än statiska RWM.

I inbyggda datorer används ett större läsminne som programminne, medan läs/skriv-minnet normalt bara är ett *mindre minne* som används dels för uppbyggnad av en s.k. *stack* (speciellt minne för lagring av återhoppsadresser vid subrutinhopp och tillfällig lagring av registerinnehåll), dels för lagring av data (variabler). Läs/skriv-minnet i den inbyggda datorn utgörs av ett litet statiskt RWM.

I den generella datorn behövs däremot ett stort läs/skriv-minne som programminne och det utgörs därför av ett dynamiskt RWM. Dock används normalt också ett mindre statiskt RWM som buffertminne (eng. *cache memory*) närmast CPU för att snabbare dataöverföring mellan CPU och läs/skriv-minne skall erhållas. Minnena ordnas alltså i en hierarki enligt figur 7.6 nedan. Med intelligent, förutseende dataöverföring mellan de olika minnena, som sköts av en speciell *minnesadministrationsenhet* (eng. *memory management unit*), så skall normalt CPU hitta önskad data i det närmast

liggande snabba minnet. Principen benämnes *virtuell minneshantering*, CPU:n ser ett *skenbart, virtuellt*, minne som är både snabbt och stort.



Figur 7.6 Minneshierarki i en generell dator.

Historiskt sett kom de första halvledarminnena av betydelse i slutet av 1960-talet och början av 1970-talet. Det var då det amerikanska företaget **Intel** (**I**ntegrated **e**lectronics) introducerade några nu klassiska halvledarminnen i MOS-teknik:

- 1969 – Intel 1101, 256 bit (256×1) statiskt RWM
- 1971 – Intel 1103, 1 kbit (1k×1) dynamiskt RWM
- 1971 – Intel 1702, 2kbit (256×8) EPROM

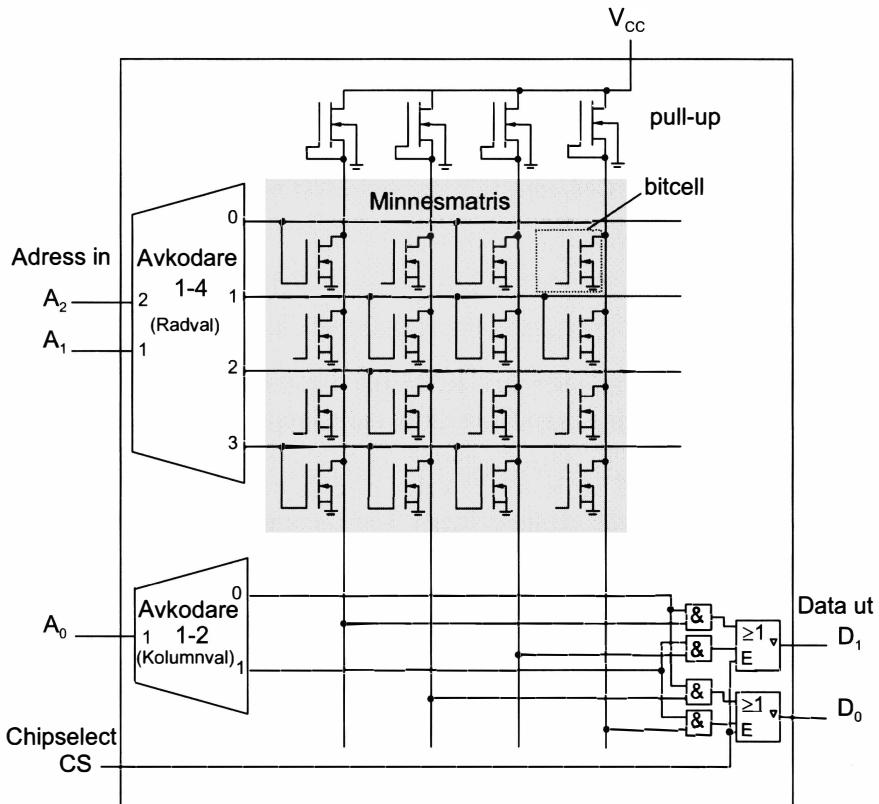
Datablad för dessa minnen, hämtade ur Intel data catalog, 1972, visas i appendix 5 för att ge perspektiv på utvecklingen.

Företaget Intel grundades 1968 och har alltsedan dess varit ledande inom mikrodatortekniken och halvledarminnestekniken. Företaget introducerade den första mikroprocessorn 1971, 4-bitars mikroprocessorn Intel 4004. – Halvledarminnet Intel 1702 innebar starten för en helt ny teknik, EPROM-tekniken, som haft mycket stor betydelse för mikrodatorerna. Intel 1103 var det första ”större” halvledarminnet av typ RWM (DRAM).

## 7.2 Läsminnen

### ROM

#### Principskiss



Figur 7.7 Principskiss på chipnivå för ett ROM med 8 ord à 2 bitar.

ROM:et innehåller 8 ord om vardera 2 bitar. För att kunna adressera 8 ord krävs en adress med 3 bitar ( $8 = 2^3$ ) och kapseln har sålunda 3

adresseringångar  $A_0$ ,  $A_1$  och  $A_2$ . Eftersom ordlängden är 2 bitar har minneskapseln 2 datautgångar  $D_0$  och  $D_1$ . I minnesmatrisen på chippet skall lagras 8 ord å 2 bitar, totalt alltså  $8 \cdot 2 = 16$  bitar. I detta lilla fiktiva ROM med bara 8 ord skulle det vara möjligt att placera orden direkt på chippet i ett bitmönster om 8 rader och 2 bitceller i varje rad. Om vi i stället har ett verkligt ROM med exempelvis 256 kbyte, dvs totalt  $256 \text{ k} \cdot 8 = 2 \text{ Mbit}$ , så är det uppenbart inte lämpligt att placera bitcellerna på chippet i ett mönster med  $256 \text{ k} = 262\,144$  rader med 8 bitceller i varje rad, chippet skulle då få en helt orimlig form. Halvledarfabrikanten strävar naturligtvis i stället alltid efter att ha ett så kvadratiskt chip i kapseln som möjligt och sålunda lägga bitcellerna i ett så kvadratiskt mönster som möjligt. I ROM:et på 2 Mbit, dvs  $2 \cdot 2^{20} = 2^{11} \cdot 2^{10}$  bitar, så lägger man bitcellerna i en  $2^{11} \cdot 2^{10} = 2048 \cdot 1024$  matris. För vårt lilla ROM med 16 bitar, dvs  $2^2 \cdot 2^2$  bitar, så lägger vi bitcellerna i en  $2^2 \cdot 2^2 = 4 \cdot 4$  matris.

## Funktion

Låt oss analysera funktionen hos ROM:et i figuren ovan. – Med adressbitarna  $A_2$  och  $A_1$  väljs i avkodaren *en* av radledningarna som blir lika med 1 (Hög). De transistorer i den utvalda raden som har styret anslutet till radledningen kommer då att leda och kolumnledningarna som dessa transistorer är anslutna till, får värdet 0 (Låg). Transistorerna i den utvalda raden som ej har styret anslutet till radledningen kommer inte att leda och tillhörande kolumnledningar får värdet 1 (Hög) via pull-up-transistorn. Kolumnledningarna är anslutna till två OCH-ELLER nät, vars utgångar utgör ROM-ets datautgångar  $D_1$  och  $D_0$ . Till OCH-ELLER-näten är också anslutna utgångarna från avkodaren som gör kolumnval. Adressbit  $A_0$  gör kolumnval genom att antingen skicka ut signalerna från de två vänstra kolumnerna (då  $A_0 = 0$ ) eller de två högra kolumnerna (då  $A_0 = 1$ ) till datautgångarna  $D_1$  och  $D_0$ . Det kan noteras att de två OCH-ELLER-näten till-sammans med avkodaren utgör två multiplexrar 2-1.

ELLER-grindarna på datautgångarna är försedda med three-state-utgång, som styrs med insignalen *CS* (eng. *Chip Select*).  $CS = 0$  ger högohmig utgång, medan  $CS = 1$  ger normal utgång, antingen Låg eller Hög. *CS*-signalen används som namnet säger till *kapselval*. Den är en typ av *Enable-signal*, som vi tidigare träffat på i många olika sammanhang och brukar också benämñas *CE* (eng. *Chip Enable*). *CS* används bl.a. då flera minneskapslar skall kopplas samman till ett större minne med fler ord, vilket strax skall demonstreras. Normalt påverkar den också kapselns effektför-

brukning på så sätt att när kapseln inte är vald så ligger kapseln i *standby* och förbrukar avsevärt mindre effekt än i läget aktiv. CS är här aktiv hög, ofta brukar den vara aktiv låg.

För de åtta olika adresserna kommer utsignalerna för ROM:et att bli enligt figur 7.8 nedan. ROM:et är uppenbart en kombinationskrets. Bitarna ligger lagrade i minnesmatrisen som ”anslutet styre”  $\leftrightarrow 0$  och ”icke anslutet styre”  $\leftrightarrow 1$ . Orden hörande till de olika adresserna ligger i matrisen på så sätt att ord 0 ligger i rad 0 i de två vänstra kolumnerna, ord 1 i rad 0 i de två högra kolumnerna, ord 2 i rad 1 i de två vänstra kolumnerna, ord 3 i rad 1 i de två högra kolumnerna osv., enligt figur 7.8 nedan.

	0	1	2	3
0	00	11	00	11
1	10	00	00	00
2	01	00	10	00
3	10	01	11	01
4	01	10	01	11
5	10	00	00	10
6	00	00	00	11
7	00	00	00	01
Placering av 2-bitars orden med adresserna 0 - 7 i bitmatrisen				

Adress in $A_2\ A_1\ A_0$	Data ut $D_1\ D_0$
0 0 0	0 1
0 0 1	0 1
0 1 0	1 0
0 1 1	0 0
1 0 0	1 0
1 0 1	1 1
1 1 0	0 0
1 1 1	0 1

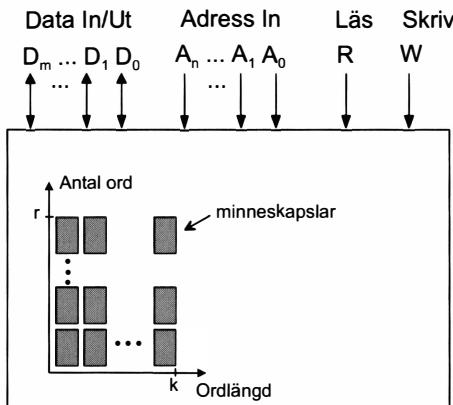
Figur 7.8 Minnesinnehåll, funktionstabell för ROM:et i figur 7.7.

Innehållet i minnesmatrisen i ett ROM läggs in av halvledarfabrikanten som ett ledningsmönster på chippet. Fabrikanten tillverkar ROM:et med alla transistorerna i matrisen fram till metalliseringsteget och lagrar dessa chip som halvfabrikat. Vid beställning av ROM från en halvledarfabrikant, så sänder beställaren över den önskade funktionstabellen på en diskett, cd el.dyl. till halvledarfabrikanten som låter ROM:halvfabrikaten gå in i tillverkningsprocessen och lägga på ledningsmönstret som förbinder styrenna med radledningarna på chippet enligt specifikationen i funktionstabellen. – Det krävs ganska stora volymer för att det skall löna sig att beställa ROM. En nackdel med att använda ROM i sina konstruktioner är att det blir besvärligt att göra modifieringar, nya ROM måste beställas vilket tar relativt lång tid och kanske många tidigare inköpta ROM kasseras. I stäl-

let för ROM används därför ofta PROM, EPROM, som ger användaren större flexibilitet genom att de kan lagerhållas oprogrammerade och snabbt programmeras.

## Expansion till ett minne med flera kapslar

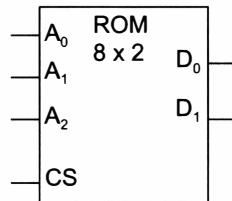
En minneskapsel innehåller ett visst antal ord av en viss ordlängd. Normalt organiseras minneskapslarna med en ordlängd av 1, 4, 8 eller 16 bitar och vanligast med 1 eller 8 bitar. Orsaken till att man väljer att ha relativt liten ordlängd i minneskapslarna trots att datorerna normalt arbetar med ordlängden 8, 16 eller 32 bitar är att man vill hålla nere antalet ben hos kapslarna, och om man måste hushålla med kapselbenen så är det optimalt att använda så många ben som möjligt till adress så att många ord kan adresseras i kapseln. Ett minne måste därför i regel byggas upp med flera kapslar enligt figur 7.9 nedan. Principen för expansion till ett minne med flera kapslar är enkel. Kapslarna kan arrangeras i ett 2-dimensionellt koordinatsystem, där expansion av ordlängden sker utefter x-axeln och expansion av antalet ord utefter y-axeln. Om det önskade minnet skall ha  $N$  ord med ordlängden  $M$  bitar och byggas upp med kapslar som har  $p$  ord med ordlängden  $q$  bitar, så skall antalet kapselrader ( $r$ ) utefter y-axeln, ordaxeln, vara  $r \geq N/p$  och antalet kapselkolumner ( $k$ ) utefter x-axeln, ordlängdsaxeln, vara  $k \geq M/q$ .



Figur 7.9 Princip för expansion till ett minne med flera kapslar.

**Exempel 7.1**

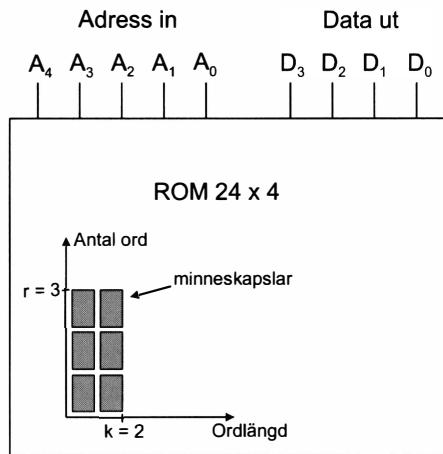
Ett minne, typ ROM, med 24 ord och ord längden 4 bitar, skall byggas upp med minneskapslar enligt figur 7.7 ovan, som har kapaciteten 8 ord å 2 bitar. Kapseln kan representeras med symbolen nedan.



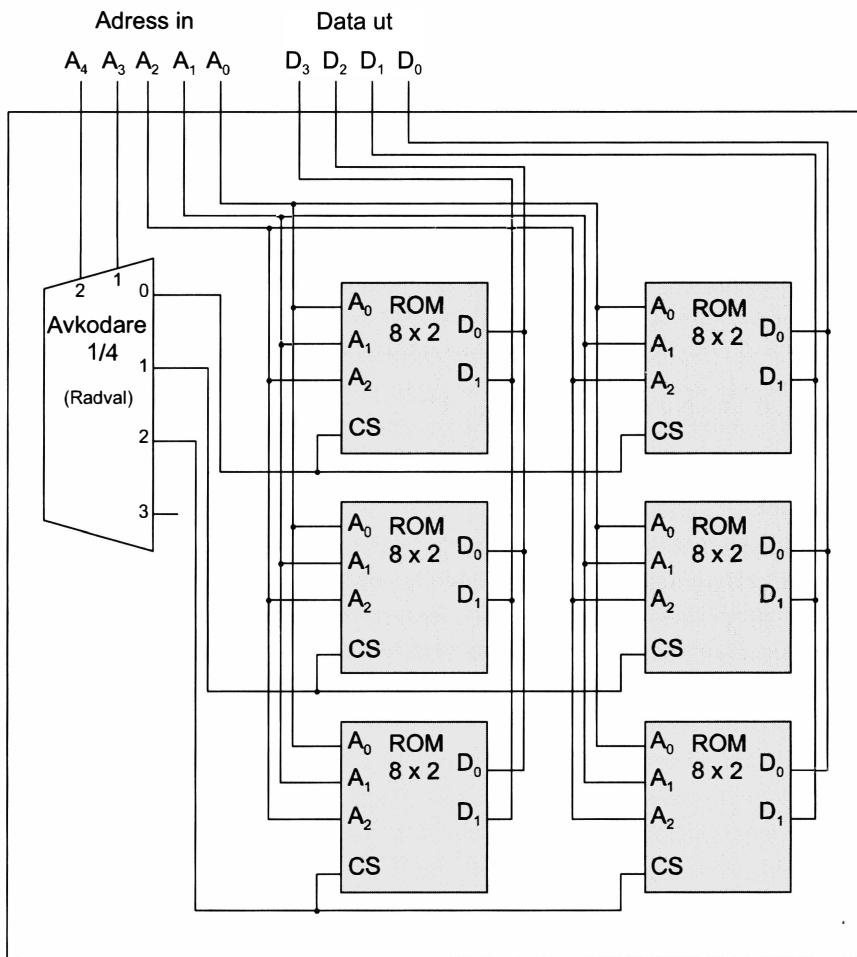
Figur 7.10 Symbol för ROM:et i figur 7.7.

**Lösning**

Antalet kapselrader (r) utefter y-axeln, ord-axeln, blir  $r = 24/8 = 3$  och antalet kapselkolumner (k) utefter x-axeln, ord längdsaxeln, blir  $k = 4/2 = 2$ . Minnet skall ha 24 ord, vilket kräver en 5-bitars adress, eftersom  $2^4 < 24 < 2^5$ . Principschema för minnet blir då enligt figur 7.11 nedan.



Figur 7.11 Principschema för ROM 24 x 4, uppbyggt med kapslar 8 x 2.



Figur 7.12 Blockschema för minne ROM  $24 \times 4$  med kapslar  $8 \times 2$ .

Kapslarnas respektive utgångar i en kolumn sammankopplas. Utgångarna är ju av typ three-state och endast en av kapselraderna får då vara aktiv och påverka utgångarna, medan övriga raders utgångar är höghömiga. CS-signalen hos kapslarna används för att styra om kapseln skall vara aktiv eller inte och eftersom *en* kapselrad i taget skall vara aktiv, så sammankopplas CS-ingångarna radvis. Kapslarnas tre adresseringångar ansluts till minnets tre

adresseringångar A<sub>0</sub>, A<sub>1</sub> och A<sub>2</sub>. Övriga två adressbitar A<sub>3</sub> och A<sub>4</sub> används för att generera CS till de tre raderna. Endast en av raderna i taget skall ha CS = 1 (aktiv) medan övriga rader skall ha CS = 0 (inaktiv) och det är uppenbart att en avkodare med A<sub>3</sub> och A<sub>4</sub> som insignalen bör användas för att generera CS-signaler. Adressområdet för de 24 orden blir 0000-10111, eller med hexadecimal representation som normalt används för adresser, området 00-17.

Generering av chip-select till kapselraderna med en avkodare enligt ovan är en generell princip som används vid expansion till ett minne med flera kapslar. Som tidigare nämnts så styr CS kapselns effektförbrukning, på så sätt att när kapseln inte är vald så ligger kapseln i standby och drar lite ström, och i minnet ovan så är det bara den valda kapselraden som är aktiv medan övriga rader ligger i standby.

□

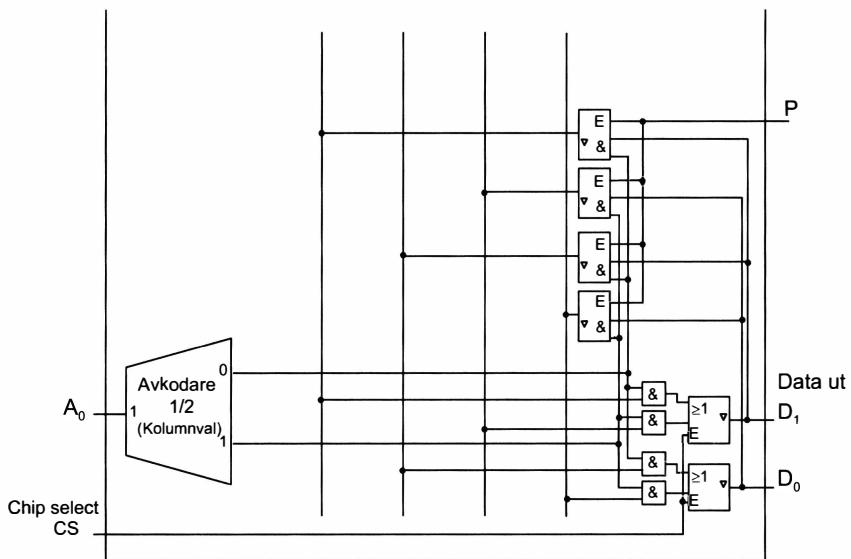
## PROM

Ett PROM är uppbyggt som ett ROM enligt principen i figur 7.7 ovan, med den skillnaden att samtliga transistorer är förbundna med tillhörande radledning. Programmeringen går till så att man i en speciell programmeringsutrustning bränner av förbindningen hos de transistorer som inte skall ha förbindelse med sin radledning. PROM tillverkades i bipolär teknik och har nu mest historiskt intresse. I figur 7.13 visas två varianter av bitcellen i ett PROM. I den vänstra cellen finns en *avbränningsbar länk, säkring* (eng *fusible link, fuse*), som kan brännas av genom att man väljer transistorn med radledningen och kör strömpulser genom transistorn, länken och kolumnledningen. I den högra cellen är radledning och kolumnledning förbundna med två motriktade dioder, som ger högohmig förbindelse mellan radledning och kolumnledning. Vid programmering läggs på radledningen så hög backspänning över den vänstra dioden att det blir ett genombrott i denna diod som ger en bestående kortslutning och sälunda en permanent förbindelse med en diod mellan radledning och kolumnledning. Vid programmering av ett PROM måste man vid val av cellerna i en rad styra vilka celler i raden som skall programmeras. Detta görs med PROM:ets datautgångar, som vid programmeringen används som dataingångar. Datautgångarna måste vara förbundna med kolumnledningarna via ingångsbuffertar som styrs av kolumnvalsavkodaren, som tillsammans med ingångsbuffertarna bildar en demultiplexer. Principen visas för dataut-

gångarna  $D_0$  och  $D_1$  i figur 7.14 nedan. Med programmeringssignalen  $P$  styrs three-state-utgångarna hos ingångsbufferna. Vid läsning skall  $P = 0$ , vilket ger högohmig utgång hos ingångsbufferna, medan vid programering skall  $CS = 0$ , som ger högohmig utgång hos ELLER-grindarna i utgångsnätet, och så skall  $P = 1$ , som öppnar ingångsbufferna så att data kan påverka kolumnledningarna. Principen visas i figur 7.14 nedan.



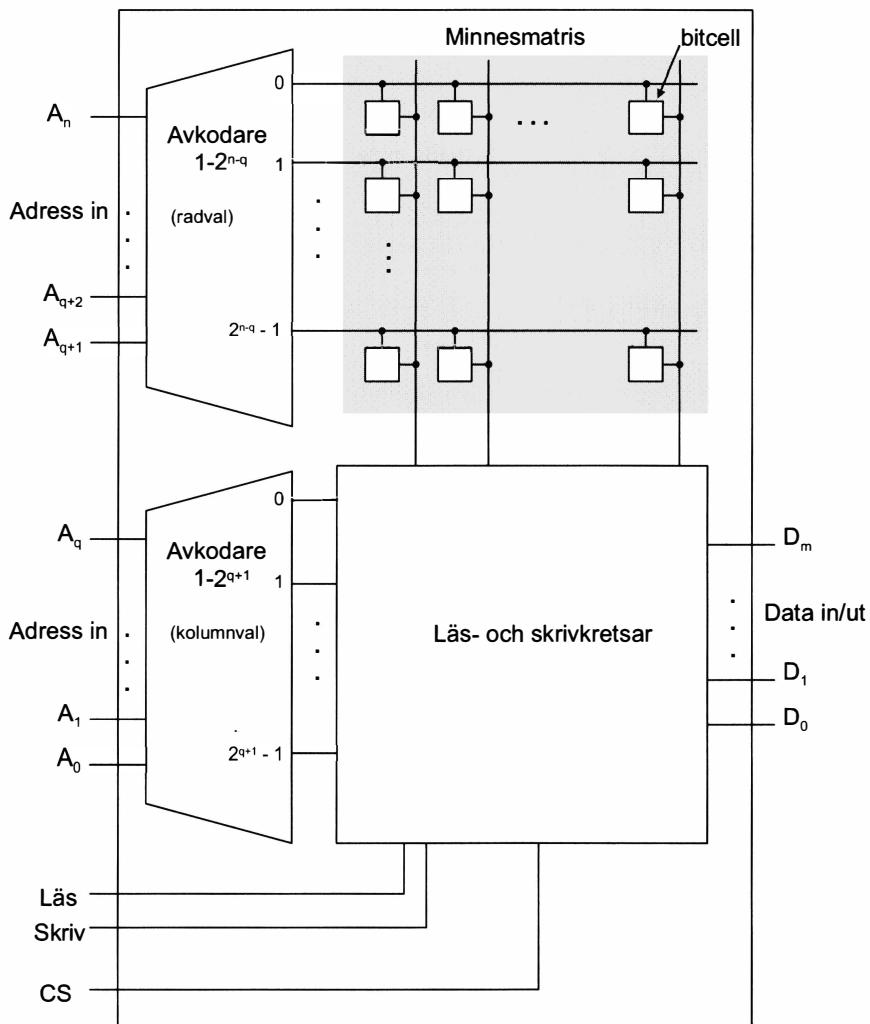
Figur 7.13 Bitceller i PROM.



Figur 7.14 Princip för inmatning av data via datautgångarna  $D_0$  och  $D_1$ .

## Modell för ett minneschip

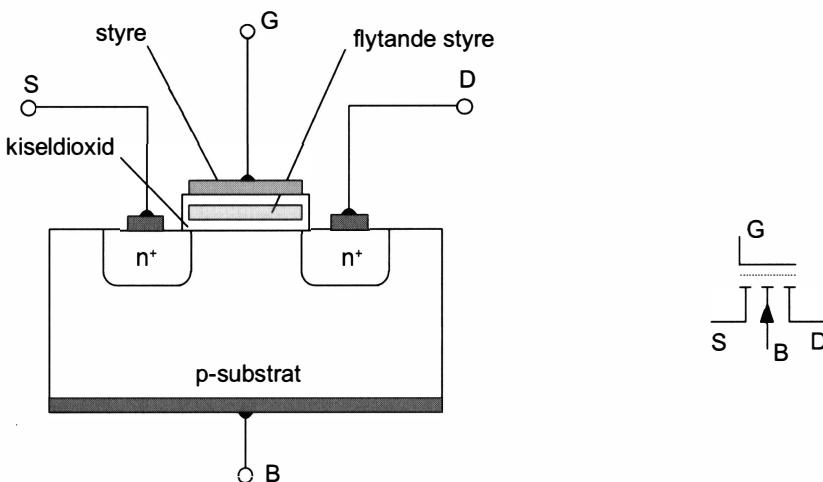
Vi har nu sett principer för radval och kolumnval i cellmatrisen på ett minneschip för både läsning och skrivning. Detta leder fram till en enkel modell för ett minneschip, oavsett typ av minne, enligt figur 7.15 nedan. Minnesmatrisen kan vara uppdelad i flera block, som i en del dynamiska RWM.



Figur 7.15 Modell för ett minneschip.

## EPROM

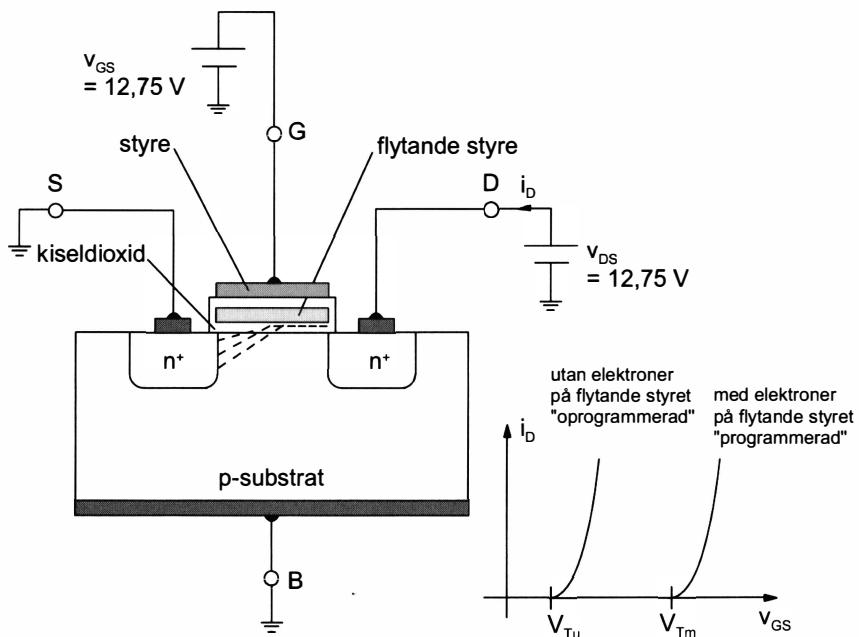
EPROM (eng. *Erasable Programmable Read Only Memory*) är ett raderbart programmerbart läsminne, som användaren själv kan programmera, radera genom bestrålning med UV-ljus och programmera på nytt. – EPROM-tekniken baseras på en speciell MOS-transistor, se figur 7.16 nedan, som har ett *flytande styre* (eng. *floating gate*) vilket ligger helt isolerat inne i kiseldioxiden mellan det vanliga styret och kanalen. Intel kallade transistorn *FAMOS* (*Floating gate Avalanche Injection MOS*).



Figur 7.16 MOS-transistor med flytande styre.

Det flytande styret används till att påverka transistorns tröskelspannning. Detta går till på följande sätt. MOS-transistorn arbetar normalt med spänningar i området 0 - 5 V. Ökar man spänningen  $v_{GS}$  på styret och spänningen på drain  $v_{DS}$  till (för moderna EPROM) 12,75 V, så kommer elektroner med hög energi att röra sig från source till drain. P.g.a. den högre spänningen på styret så kommer en del elektroner med hög energi, i databöcker på engelska benämnda "*hot electrons*", att tränga igenom det isolerande skiktet av kiseldioxid mellan kanalen och det flytande styret och hamna på det flytande styret. Detta blir då negativt laddat och kommer, då transistorn arbetar i det normala spänningsområdet 0 - 5 V, att stöta bort elektroner från kanalen och alltså motverka det vanliga styrets förmåga att

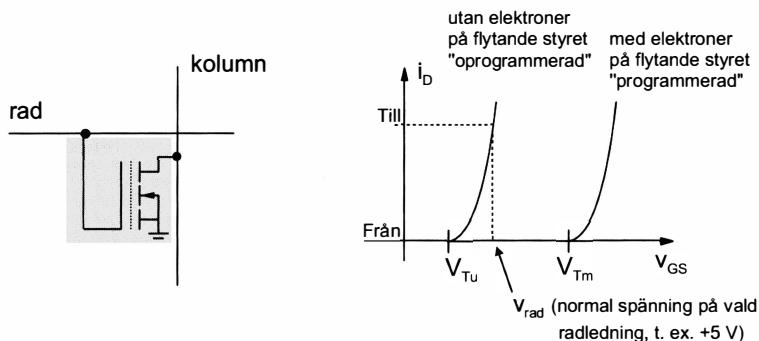
attrahera elektroner till kanalen. Med det flytande styret uppladdat så krävs alltså en högre spänning  $v_{GS}$  på det vanliga styret för att få lika mycket elektroner i kanalen som då det flytande styret ej är uppladdat. Elektroner på det flytande styret resulterar sålunda i en förskjutning åt höger av  $i_D = f(v_{GS})$ -kurvan och ett högre värde på tröskelspannningen  $V_T$ , enligt figur 7.17 nedan, där  $V_{Tu}$  och  $V_{Tm}$  betecknar tröskelspannningen utan respektive med elektroner på det flytande styret.



Figur 7.17 Programmering av MOS-transistor med flytande styre.

Bitcellen i minnesmatrisen i ett EPROM visas i figur 7.18 nedan. Den har i princip samma utseende som den tidigare visade bitcellen för ett ROM med styret anslutet till radledning, drain anslutet till kolumnledning och source jordad. Med det flytande styret oladdat så kommer transistorn då den väljs med spänningen Hög (5 V) på radledningen,  $v_{rad} = 5$  V, att leda (Till) och kolumnledningen bli Låg (0 V). Efter invertering i utgångskretsen blir kapselfens datautgång Hög, logiskt 1. I en raderad EPROM-kapsel innehåller alltså hela minnesmatrisen ettor. Med det flytande styret laddat så kommer

transistorn då den väljs med spänningen Hög på radledningen, inte att leda (*Från*) eftersom tröskelspannningen är högre än spänningen på radledningen och kolumnledningen förblir Hög via pull-up-transistorn och datautgången hörande till kolumnledningen efter invertering i utgångskretsen blir Låg, logiskt 0.



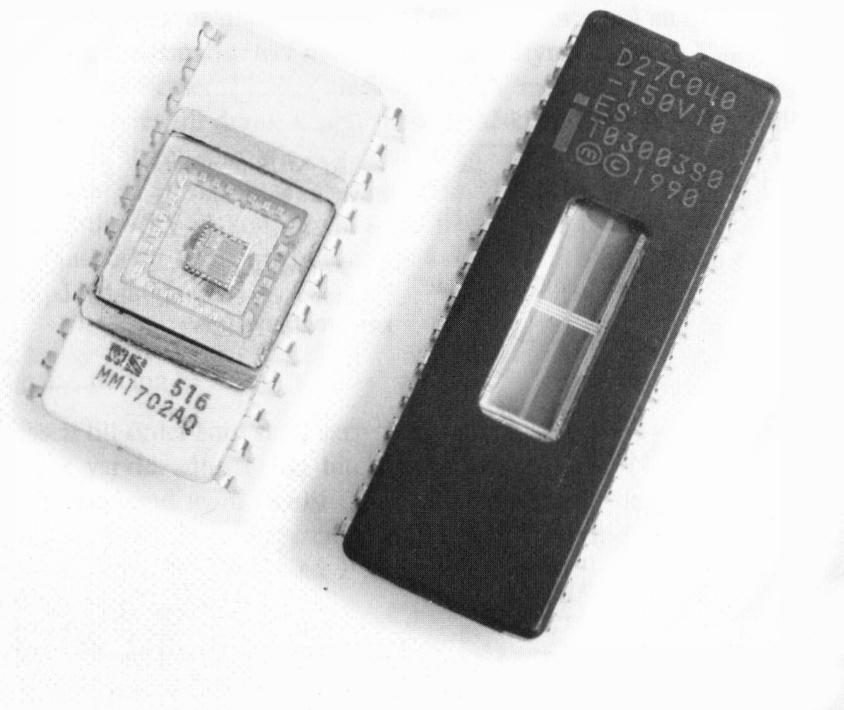
Figur 7.18 Bitcell i EPROM.

”Programmering” av elektroner på det flytande styret i bitcellen sker som redan antyts, genom att spänningen på radledningen och kolumnledningen höjs till 12,75 V under 100  $\mu$ s (tiden varierar beroende på version av EPROM) eller om det efter verifiering av programmeringen visar sig nödvändigt, ett antal multiplar av 100  $\mu$ s. Vid programmering av ett ord, 8 eller 16 bitar, läggs ordet på kapselns datautgångar som alltså vid programmeringen används som dataingångar. På kapselns adresseringångar appliceras adressen, som i matrisen väljer dels rad, dels de 8 eller 16 kolumner i matrisen motsvarande bitarna i ordet. Den valda raden läggs på 12,75 V och de valda kolumnerna vars dataingång har värdet 0 läggs på 12,75 V.

EPROM-epoken startade, som nämntes i inledningen, år 1971 med EPROM-kapseln Intel 1702, 2 kbit (256×8). EPROM-utvecklingen under 20-årsperioden 1971-1991 karakteriseras av att lagringskapaciteten har ökat i en takt av ca en *fördubbling* av antalet bitar vartannat år. Raden av EPROM-kapslar under denna period är 1702, 2708, 2716, 2732, 2764, 27128, 27256, 27512, 27010, 27020 och 27040. Kapselbeteckningarna följer ett system där 1:a siffran '2' anger att det är en minneskrets, 2:a siffran '7' att det är ett EPROM, och de därpå följande siffrorna anger kap-

citeten i kilobit. Exempelvis anger 27512 att kapaciteten är 512 kbit ( $64\text{k}\times 8$ ) och 27040 att kapaciteten är 4 Mbit ( $512\text{k}\times 8$ ).

EPROM 27040 blev Intels sista, då Intel 1992 beslutade att upphöra med utveckling av nya EPROM och i stället satsa på Flash-minnen som bygger på en snarlik teknik, men som är elektriskt raderbara.

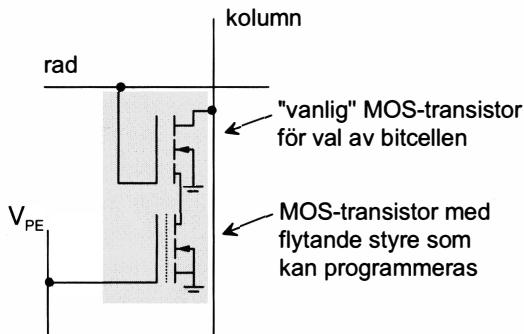


Figur 7.19 Intels första EPROM 1702 (256 byte) och sista EPROM 27040 (512 kbyte).

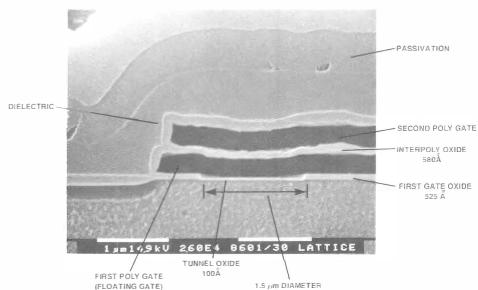
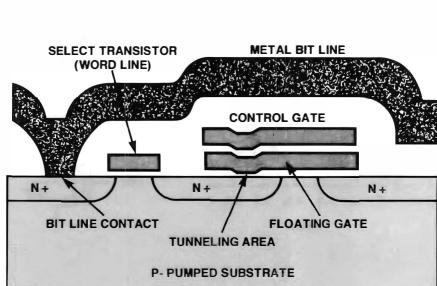
## EEPROM

EEPROM, även E<sup>2</sup>PROM, (eng. *Electrically Erasable PROM*) är ett läsminne som kan programmeras och raderas elektriskt i den aktuella tillämpningen. Radering och programering sker byte-vis.

E<sup>2</sup>PROM-bitcellen är en två-transistorcell, se figur 7.20 nedan, där den ena transistorn har ett flytande styre, medan den andra transistorn är en "vanlig" transistor som används för val av bitcellen.



Figur 7.20 EEPROM-bitcell.



Figur 7.21 EEPROM-bitcell (Lattice).

EEPROM-tekniken baseras liksom EPROM-tekniken på en MOS-transistor med flytande styre, men det flytande styret går in över drain med ett område som ligger på mycket litet avstånd, benämnt *tunneling area* i figur 7.21 ovan där en EEPROM-cell visas (figurerna är hämtade ur: Lattice "GAL Handbook"). Elektroner kan fås att passera genom det tunna oksidskiktet

mellan kanalen och drain, fenomenet benämnes *Fowler-Nordheim tunnelfekten*, som sålunda möjliggör tillförande av elektroner till flytande styret alternativt bortförande av elektroner från det flytande styret. Transistorn brukar benämns **FLOTOX** (*Floating gate tunnel oxide*).

## Radering

Radering sker ord för ord. Adressen till ordet som skall raderas läggs på kapselns adresseringångar varvid adresserade bitceller väljs med Hög (+5 V) på radledningen (benämnd *word line*) i figur 7.21 ovan, så att den "vanliga" transistorn, (benämnd *select transistor*), leder. Styrspänningen  $V_{PE}$  läggs på ca 12 V och adresserad kolumnledning på 0 V medförande att tunnelström flyter från styret till drain genom tunneloxidskiktet och elektroner tillförs det flytande styret som blir negativt laddat. Tröskelspanningen får nu ett högre värde så att när bitcellen adresseras vid en läsning och  $V_{PE}$  har sitt normalvärde +5 V så kommer inte transistorn med flytande styre att leda, vilket innebär att kolumnledningen med sin pull-up blir Hög och en 1:a erhålls på datautgången.

## Programmering

Adressen till ordet som skall skrivas in i minnet läggs på kapselns adresseringångar varvid adresserade bitceller väljs med Hög (+5 V) på radledningen (benämnd *word line*) i figur 7.21 ovan, så att den "vanliga" transistorn, (benämnd *select transistor*), leder. Radering utförs, se ovän. Styrspänningen  $V_{PE}$  läggs på 0 V. Ordet som skall skrivas in läggs på kapselns datautgångar.

### *Skrivning av en 0:a i bitcellen (inga elektroner på flytande styret)*

En 0:a i ordet som skall skrivas lägger adresserad kolumnledning på ca 12 V, medförande att tunnelström flyter från drain till styret genom tunneloxidskiktet och elektroner tillförs från flytande styret som blir oladdat. När bitcellen adresseras vid en läsning och  $V_{PE}$  har sitt normalvärde +5 V så kommer transistorn med flytande styre att leda, vilket innebär att kolumnledningen blir Låg och en 0:a erhålls på datautgången.

### *Skrivning av en 1:a i bitcellen (elektroner på flytande styret)*

En 1:a i ordet som skall skrivas in lägger adresserad kolumnledning på 0 V, varvid ingen tunnelström flyter mellan drain och styre och det flytande sty-

ret förblir negativt laddat. När bitcellen adresseras vid en läsning och  $V_{PE}$  har sitt normalvärdet +5 V så kommer inte transistorn med flytande styre att leda, vilket innebär att kolumnledningen med sin pull-up blir Hög och en 1:a erhålls på datautgången.

Skriftiden för ett ord i ett EEPROM är av storleksordningen någon milisekund. EEPROM-cellens behållning är bra som EPROM-cellens, men radering-skrivning ger utmattningsfenomen i det tunneloxidskiktet, så att antalet cykler radering-skrivning är begränsat.

## Flash-minnen

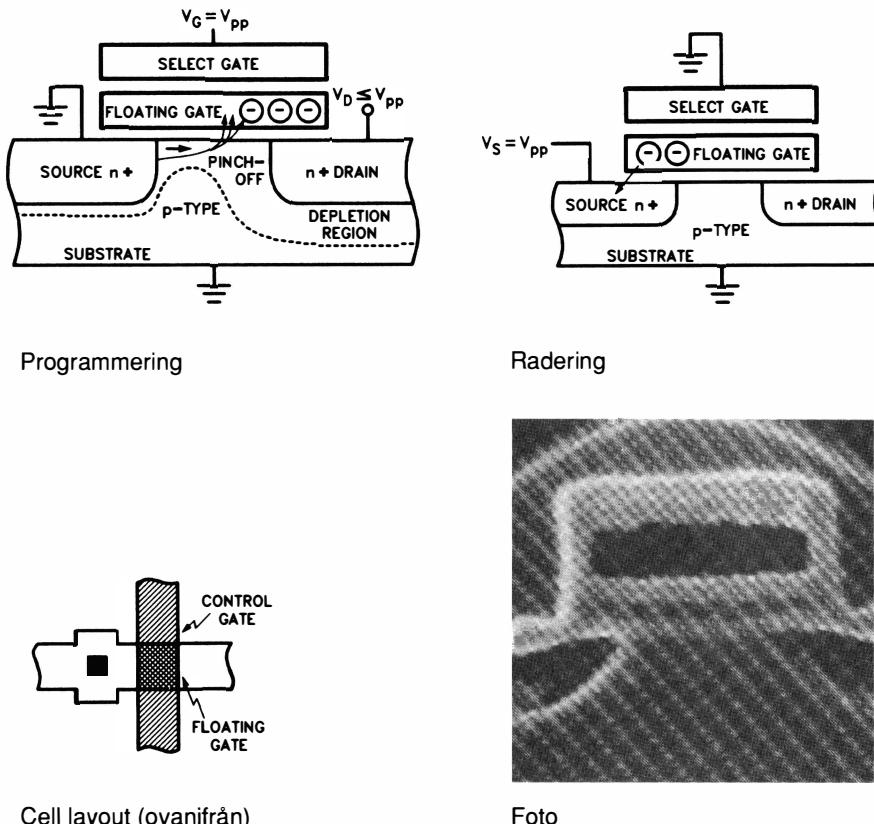
Flash-minnet är liksom EEPROM:et elektriskt raderbart, men till skillnad från det senare som är raderbart på byte-nivå, så är flash-minnet raderbart på blocknivå, hela kapsellinnehållet eller ett större block på en gång. – Flash-minnestekniken utvecklades först av det japanska företaget Toshiba i mitten av 1980-talet, tätt följt av företagen Intel och Seek Semiconductor.

	<b>EPROM</b>	<b>ETOX II Flash Memory</b>	<b>EEPROM</b>
<b>Normalized Cell Size</b>	1.0	1.2–1.3	3.0
<b>Programming:</b> Mechanism	Hot Electron Injection Byte	Hot Electron Injection Byte	Tunneling Byte 5 ms
Resolution Typ. Time	< 100 $\mu$ s	< 10 $\mu$ s	
<b>Erase:</b> Mechanism Resolution Typ. Time	UV Light Bulk Array 20 Min.	Tunneling Bulk Array < 1 Sec.	Tunneling Byte 5 ms

Figur 7.22 Jämförande data för celler EPROM, EEPROM och ETOX (Intel).

Minnescellen i ett flash-minne är en en-transistorcell med en MOS-transistor med flytande styre och ett tunneloxidområde, här mellan mellan styret och source. Intel kallar sin cell för ETOX (EPROM tunnel oxide). I figur 7.22 visas jämförande data för de olika cellerna EPROM, EEPROM och ETOX (ur Intel Memory Products, 1992).

Radering och skrivning skiljer sig något från motsvarande för EEPROM. I figur 7.23 nedan visas principerna för radering och skrivning samt layout och foto av en ETOX-cell.



*Figur 7.23 ETOX-cell (Intel).*

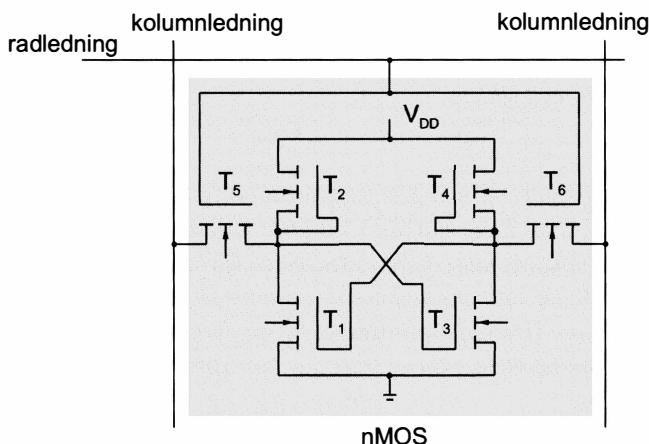
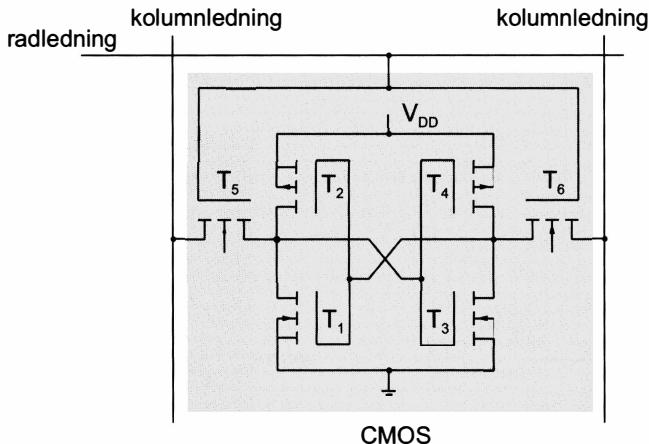
Som framgår av figuren ovan går programmering till på samma sätt som för EPROM-cellen, dvs. med vad som tidigare kallats *hot electrons*. Radering sker genom att source, som ju är gemensam för alla transistorer alternativt ett antal transistorer (ett minnesblock) på chippen, läggs Hög, ca 12 V, och styret på 0 V, medförande att tunnelström flyter från source till styret genom tunneloxidskiktet och elektroner bortförs från det flytande styret som blir oladdat.

Flash-minnen är det mest expansiva och dynamiska området av icke-flyktiga minnen. De används i många tillämpningar, såsom mobiltelefoner, mikrostyrkretsar, handdatorer, kortminnen "Flash Card" av storlek kreditkort som rymmer 100-tals Mb för användning i t.ex portabla datorer.

## 7.3 Läs/skriv-minnen

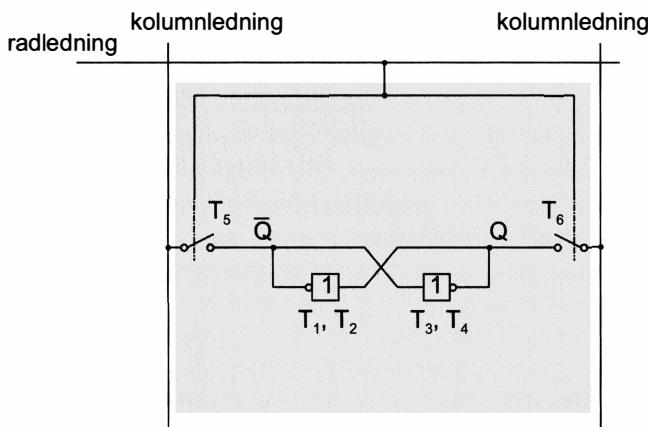
### Statiska RWM (SRAM)

I ett statiskt RWM utgörs bitcellen av en SR-latch. I figuren nedan visas bitcellen i ett statiskt RWM uppbyggd med 6 transistorer i CMOS respektive nMOS.



Figur 7.24 Bitcell i statiskt RWM (SRAM) i CMOS respektive nMOS.

Vi har studerat latchar i kapitel 6, Sekvenskretsar, och sett hur en SR-latch kan konstrueras med två NOR-grindar eller med två NAND-grindar. I bitcellerna ovan utgörs själva latchesen av två korskopplade inverterare, se figur 7.25 nedan, uppbyggda av transistorerna T<sub>1</sub>, T<sub>2</sub> respektive T<sub>3</sub>, T<sub>4</sub>. Transistorerna T<sub>5</sub>, T<sub>6</sub>, som styrs med radledningen, är switchar som används för adressering av bitcellen då innehållet skall läsas eller då en bit skall skrivas in. Vi ser att varje kolumn av bitceller har två kolumnledningar. Vid skrivning av en 0:a i latchen läggs kolumnledning Q = 0 och  $\bar{Q}$  = 1, medan vid skrivning av en 1:a läggs tvärtom.

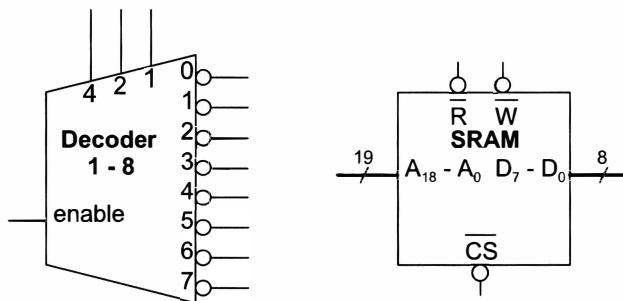
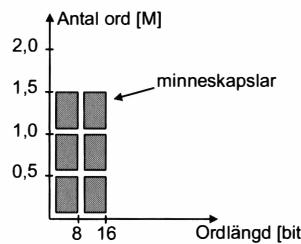


Figur 7.25 Logiskschema för bitcellerna i statiskt RWM (SRAM) i figur 7.24 ovan.

Viktiga användningar för statiska RWM (SRAM) är som snabbt buffertminne, *cache*, i datorer, se figur 7.6, och som variabel- och stackminne i mikrostyrkretsar.

**Exempel 7.2**

Till en mikroprocessor skall anslutas ett läs/skrivminne. Mikroprocessorn har en 16-bitars databuss och en 24-bitars adressbuss samt en styrbuss där det ingår styrsignaler för läsning och skrivning, *read* respektive *write*, aktivt låga. Läs/skrivminnet skall byggas upp med SRAM-minneskapslar på 4 Mbit ( $512 \text{ k} \times 8$ ), se symbolen nedan. Som framgår av symbolen så har kapslarna 19 adresssignaler  $A_0 - A_{18}$ , som ju möjliggör direkt adressering av  $512 \text{ k}$  ord, ty  $2^{19} = 2^9 \cdot 2^{10} = 512 \text{ k}$ . Mikroprocessorns 24-bitars adressbuss ger ett totalt adressområde 000000 till FFFFFFF, angivet i hexadecimal form. Läs/skrivminnet skall ha kapaciteten  $1,5 \text{ M} \times 16$  och placeras i adressområdet A80000 till BFFFFF. För generering av *chipselect* till kapslarna skall användas en avkodare 1-8 med aktivt låga utgångar och försedd med enable-ingång, aktivt hög. Till avkodaren skall anslutas adresssignaler  $A_{19} - A_{23}$  så att den genererar chipselect i hela adressområdet 800000 till BFFFFF och sålunda möjliggör framtida utökning av läs/skrivminnet större antal ord med fler kapslar. – Rita blockschema för läs/skrivminnet och använd då symbolerna nedan. Ange på avkodarens utgångar inom vilket adressområde den ger chipselect.

**Lösning**

Figur 7.26 Kapselkonfiguration för läs/skrivminnet  $1,5 \text{ M} \times 16$ .

I kapselkonfigurationen i figuren ovan har antalet kapslar utefter ord längd-axeln bestämts som

$$(\text{önskad ord längd}) / (\text{ord längd i kapseln}) = 16/8 = 2.$$

Antalet kapselrader utefter ordaxeln har bestämts som

$$(\text{önskat antal ord}) / (\text{antal ord i kapseln}) = 1,5 \text{ M} / 0,5 \text{ M} = 3.$$

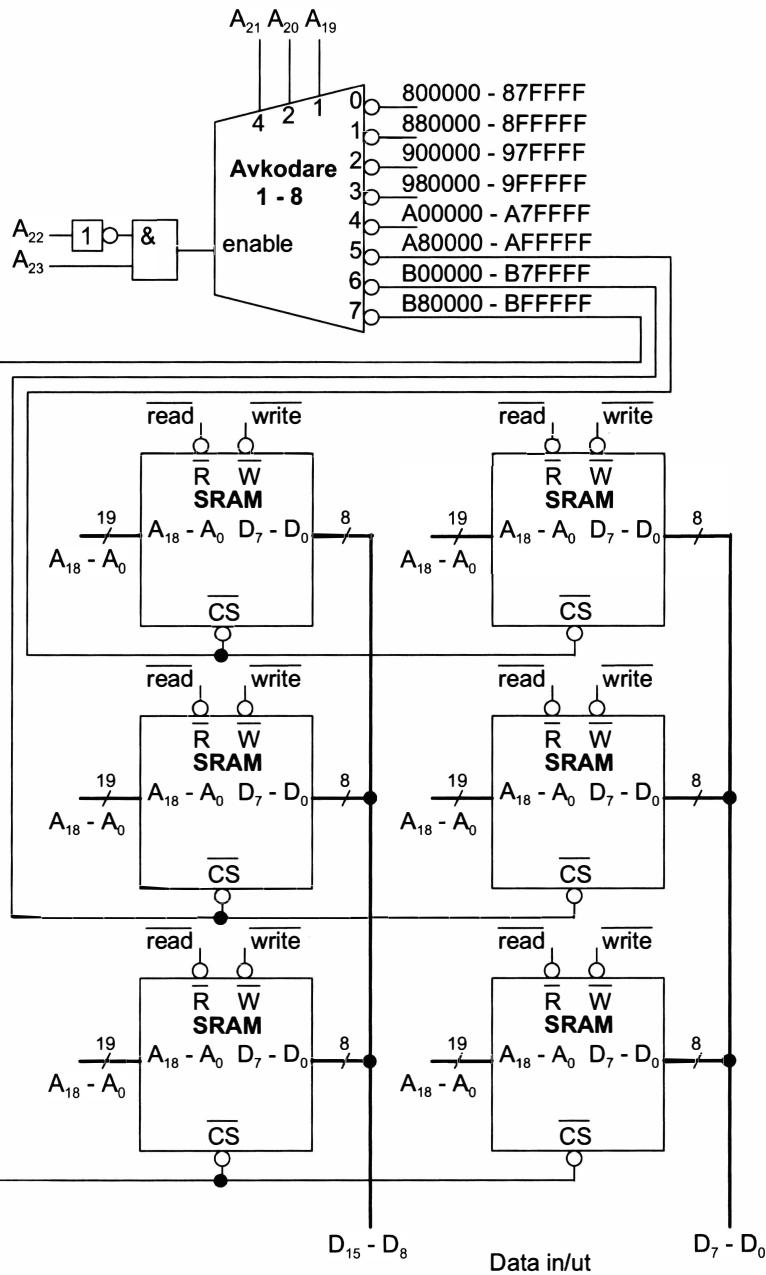
I figur 7.27 nedan visas i binär och hexadecimal form de för avkodaren och minnet specificerade adresserna. Adressbussens signaler  $A_0$  till  $A_{18}$  ansluts till samtliga kapslars motsvarande adressingångar. Övriga signaler  $A_{19}$  till  $A_{23}$  på adressbussen ansluts till avkodaren,  $A_{19}$ ,  $A_{20}$  och  $A_{21}$  till avkodarens adressingångar och  $A_{22}$  och  $A_{23}$  till avkodarens enable-ingång via lämpliga grindar. För första adressen till minnet A80000 (Hex) är  $A_{21}A_{20}A_{19} = 101_2 = 5_{10}$ , innehållande att utgång 5 hos avkodaren skall anslutas till chipselect på översta radens minneskapslar. För sista adressen till minnet BFFFFFF (Hex) är  $A_{21}A_{20}A_{19} = 111_2 = 7_{10}$ , innehållande att utgång 7 hos avkodaren skall anslutas till chipselect på nedersta radens minneskapslar. Utgång 6 hos avkodaren skall anslutas till chipselect på mittersta radens minneskapslar.

Ansluts till:	Avkodaren enable adress								Minneskapslarnas adressingångar															
Adressbitar:	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Första adress avkodare	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Första adress minne	1	0	1	0	5	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sista adress minne	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Sista adress avkodare	B				F				F				F			F		F						

Figur 7.27 Analys av specificerade adresser för avkodare och minne.

I blockschemat för läs/skrivminnet i figur 7.28 nedan har adressbussens signaler  $A_0$  till  $A_{18}$  anslutits till samtliga kapslars motsvarande adressingångar. Vidare har signalerna read och write från mikroprocessorns styrbuss anslutits till  $\bar{R}$  respektive  $\bar{W}$  på samtliga kapslar.

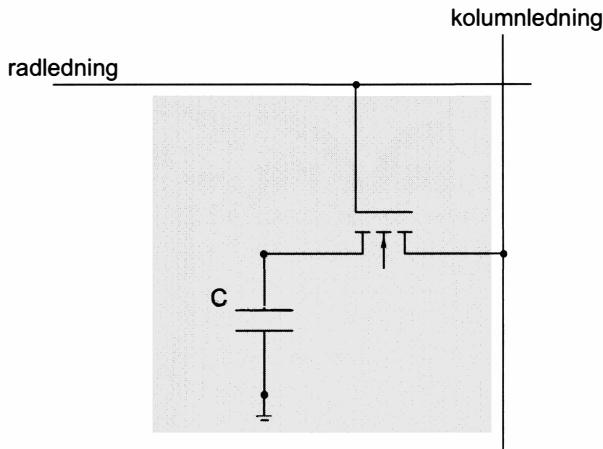
□



Figur 7.28 Blockschema för läs/skrivminnet i exempel 7.2.

## Dynamiska RWM (DRAM)

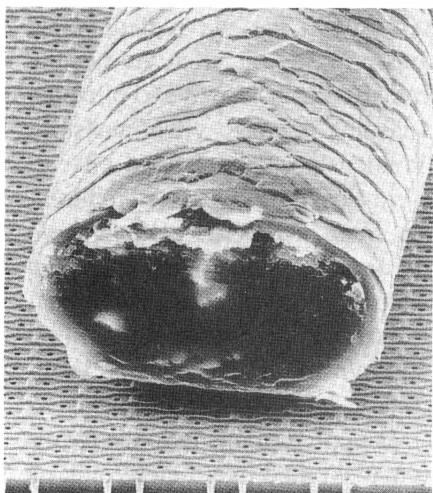
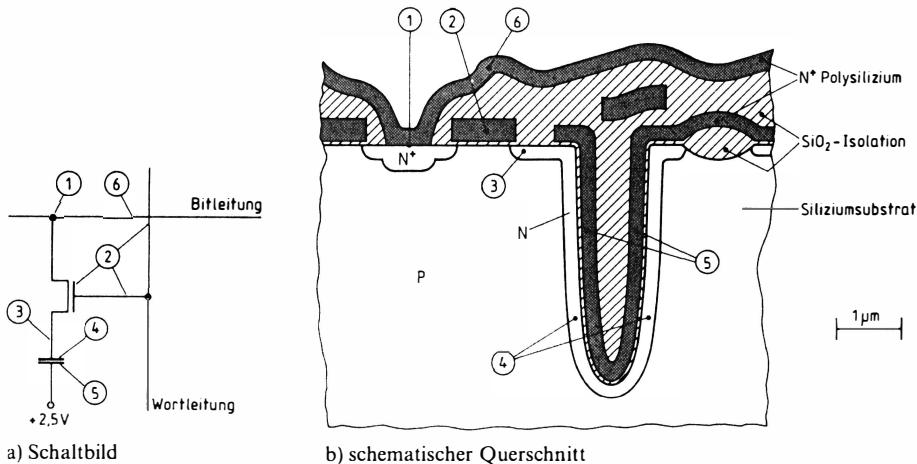
I ett dynamiskt RAM lagras informationen i bitcellen i en kondensator. Bitcellen är en *en-transistor-cell* enligt figuren nedan.



Figur 7.29 Bitcell i dynamiskt RWM (DRAM).

Funktionsprincipen är enkel – transistorn, som styrs via radledningen, tjänstgör som switch och används för adressering av bitcellen vid läsning och skrivning i cellen på samma sätt som transistorerna T5 och T6 i bitcellen i ett statiskt RWM i figur 7.25 ovan. Vid läsning öppnas switchtransistorn via radledningen och spänningen över kondensatoren, uppladdad eller ej uppladdad, avläses via kolumnledningen. Vid skrivning öppnas switchtransistorn via radledningen och informationen, biten, skrivs in i cellen genom att kondensatoren laddas upp eller laddas ur via kolumnledningen. P.g.a. den oundvikliga läckströmmen i switchtransistorn kommer kondensatoren att laddas ur efter ett antal millisekunder och därför måste informationen i ett dynamiskt RWM periodiskt *uppförskas* (eng. *refresh*), innebärande att informationen i alla celler läses och direkt skrivs in igen. I ett 256 Mbit dynamiskt RWM, som är standard idag (2001), är uppförskningsperioden 64 ms och tar 7,8 µs att genomföra och sker automatiskt med intern elektronik på chippet. I det första dynamiska RWM (1971), Intel 1103, 1 kbit, vars data-blad av historiska skäl visas i appendix 5, var uppförskningsperioden 2 ms. I detta första dynamiska RWM var bitcellen en *tre-transistor-cell*, bestående av en switchtransistor för läsning och en switchtransistor för skrivning samt

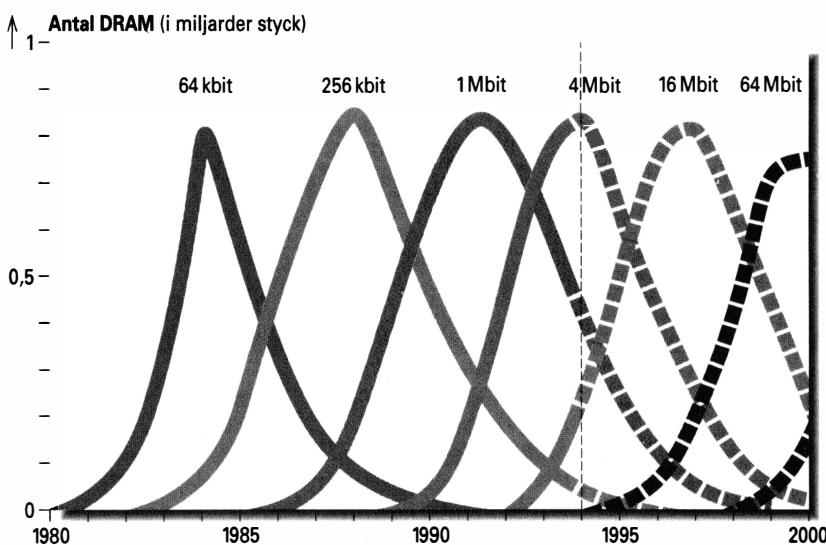
en transistor för lagring av biten, vilket skedde i ingångskapacitansen (parasit-kapacitansen) hos styret. När man sedan i efterföljande generationer dynamiska RWM minskade transistordimensionerna, så blev parasitkapacitanserna för små för att kunna lagra bitinformationen tillräckligt många millisekunder och man blev tvungen att integrera "riktiga" kondensatorer på chippen.



Figur 7.30 Kopplingsschema och schematiskt tvärsnitt av bitcell med kondensator samt ett foto av ett DRAM-chip på vilket ligger ett hårstrå.(Siemens).

Kondensatorn måste vara ca  $35\text{fF}$  (femtoFarad,  $10^{-15}\text{F}$ ) för att uppfriskningsperioden skall få en acceptabel storlek, men också ur störningssynpunkt, t.ex. från bitledningens egenkapacitans. Kondensatorerna integrerades horisontellt på chippet t.o.m. generationen 1 Mbit dynamiska RWM, men i efterföljande generationer är bitcellen så liten att kondensatorerna inte får plats i horisontalplanet, utan man har blivit tvungen att låta kondensatorerna gå ned vertikalt i chippet, med kondensatorplattorna utefter väggarna i ett dike (eng. *trench*; ty. *Graben*), som visas i figur 7.30 ovan. På fotot syns de vertikala kondensatorerna i framkanten av chippet.

Utvecklingen av dynamiska RWM sedan introduktionen av det första år 1971 har skett i en takt av ca en fyrdubbling av kapaciteten var tredje år. Idag (2001) är kapslar på 512 Mbit på väg ut på marknaden som volymprodukt. I figuren nedan visas livscykeln för olika generationer av dynamiska RWM.



Figur 7.31 Livscykeln för olika generationer av DRAM. (Siemens).

Endast några få halvledarfabrikanter i världen har idag resurser att utveckla dynamiska RWM, det krävs mycket stora investeringar för varje ny generation av minneskretsar. Det tyska företaget Siemens Semiconductor som

startade sin utveckling av dynamiska RWM i mitten av 1970-talet med 4 kbit dynamiska RWM, går idag i spetsen för utvecklingen – sedan 1999 under nytt namn, *Infineon Technologies*, se <http://www.infineon.com/>. Utvecklingen sker i samarbete med IBM och Toshiba. Marknaden för dynamiska RWM är enorm – den pågående utvecklingen av IT-samhället kräver kraftfulla persondatorer med mycket stora minnen.

I figur 7.30 nedan visas några jämförande data för Siemens dynamiska RWM från 4 kbit till 16 Mbit.

	4 KBit	16 KBit	64 KBit	256 KBit	1 MBit	4 MBit	16 MBit
Speicherbare DIN A4-Schreibmaschinen-Seiten	1/4	1	4	16	64	256	1000
Chipfläche in mm <sup>2</sup>	24	20	30	45	54	91	142
Kleinste laterale Strukturen in µm	6	4	2	1,5	1,2	0,8	0,6
Lithographiebedingte Minimalauflösung*	3,5	2,2	1,3	1,0	0,8	0,7	0,6
Zahl der Prozeßschritte	70	80	80	120	280	400	450
Produktionsaufnahme	1976	1980	1981	1985	1987	1989	voraussichtlich 1992

\* (g-Linie)

Figur 7.32 Jämförande data för Siemens dynamiska RWM 4kbit – 16Mbit.

Låt oss nu studera en konkret krets, Infineon HYB25D256400/800/AT 256 Mbit DDR SDRAM, se utdrag ur datablad i appendix 4. Kretsen är en fin representant för ”state-of-the-art” 2001 för DRAM, den tillverkas på kiselplattor (eng. *wafer*) med diametern 300 mm och med en linjebredd av 0,17 µm. I beteckningen SDRAM står ’S’ för *Synchronous*. De första generationerna av DRAM fram till 64 Mbit var *asynkrona*, innebärande att de inte klockades med processorns systemklocka. Vid läsning och skrivning fick processorn, efter att den lagt ut adress och styrsignalen *read*, respektive *write* och *data* (se princip i figurerna 7.3 och 7.4), vänta tillräckligt lång tid, beroende på typ av minneskapslar, på att läsningen respektive skrivningen skulle bli klar, i s.k. *wait states*, ett antal klocksignalperioder. Från och med generationen 64 Mbit DRAM har konstruerats *synkrona* minnen, innebärande att de klockas med systemklockan, t.ex. med frekvensen 133 MHz. De första synkrona DRAM var av typ SDR (Single Data Rate), innebärande

att data kan läsas eller skrivas en gång per period av systemklockan, på den positiva flanken av klocksignalen. Kretsen vars datablad visas här är av typen DDR (Double Data Rate), innebärande att data kan läsas och skrivas två gånger per period av klocksignalen, på både den positiva och den negativa flanken av klocksignalen.

Vår 256 Mbit-DRAM finns med två olika ord/ordlängd-organisationer, dels som  $64M \times 4$  (64 M ord à 4 bitar), dels som  $32M \times 8$  (32 M ord à 8 bitar). Låt oss närmare studera varianten  $32M \times 8$ . Adressering av 32 M ord kräver 25-bitars adress,  $32M = 2^5 \times 2^{20} = 2^{25}$ . I databladet page 2, Pin Configuration och page 6 Blockdiagram (32 Mb×8), ser vi att kapseln inte har så många adressingångar. Förklaringen är att adressering av ett ord sker genom att adressen delas upp i två halvor, som sänds in i kapseln efter varandra, i s.k. tidsmultiplex, och lagras i kapseln i två register (latchar), i blockschemat betecknade *Row-Address Latch* och *Column-Address Latch*. Orsaken till detta adresseringsförfarande är att man velat hålla nere antalet ben hos DRAM-kapslar. Den första kapseln Intel 1103, 1 kbit, hade 18 ben, se appendix 4, men man använde inte där tidsmultiplex. I efterföljande generationer övergick man till tidsmultiplex och 16-bens kapsel, som kunde behållas t.o.m 256 kbit-kapslar. 256 Mbit-DRAM-kapseln har 66 ben i en kapsel med dimensionen 10,16 mm×22,22 mm.

Ur blockschemat framgår att minnesmatrisen är uppdelad i fyra matriser (bankerna 0 till 3) på vardera 64 Mbit. Varje bank har dimensionen 8192 rader×8192 kolumner ( $= 2^{13} \times 2^{13} = 2^{26} = 2^6 \times 2^{20} = 64M$ ). De 8192 kolumnerna är angivna som  $512 \times 16$  för att markera 512 16-bitars ord.

Adresseringen av ett 8-bitars ord går till på följande sätt. Inmatningen av de två adresshalvorna styrs med de två signalerna  $\overline{RAS}$  (*Row Address Strobe*) och  $\overline{CAS}$  (*Column Adress Strobe*). Först läggs radadressen (13 bitar) på adressbenen A0 - A12 och klockas in i radadresslatchen (*Row-Address Latch*) via Row-Address MUX med  $\overline{RAS} = \text{Låg}$ . Val av bank görs med in-signalerna BA0 och BA1. Därefter läggs kolumnadressen (10 bitar) på adressbenen A0 - A9 och klockas in i kolumnadresslatchen (*Column-Address Latch*) med  $\overline{CAS} = \text{Låg}$ . Av kolumnadressens 10 bitar används 9 för att välja ett av de 512 16-bitars orden ( $2^9 = 512$ ) i den valda banken. Den återstående 10:e biten av kolumnadressen (COL0) används för att välja hälften av bitarna i 16-bitars ordet, dvs 8 bitar som skall skickas ut på utgångarna D0-D7 vid läsning eller skickas in vid en skrivning.

Uppfriskning av samtliga 256 M bitceller i matriserna måste göras periodiskt med perioden 64 ms. Uppfriskningen görs radvis samtidigt i alla fyra

bankerna, vilket ger snabb uppfriskning av hela kapselinnehållet. I detta fall krävs uppfriskning av totalt 8192 rader, som styrs av en adressräknare Refresh Counter som adresserar rad för rad i bankerna via Row-Address MUX. Uppfriskningen av hela minnet tar  $7,8 \mu\text{s}$ , vilket bara utgör 0,1 % av uppfriskningsperioden, dvs en mycket liten del av minnets normala verksamhet går åt till uppfriskning. I större minne som är uppbyggt med flera kapslar görs uppfrisksning av alla kapslar samtidigt, innebärande att uppfrisknings-tiden blir oberoende av antalet ord.

Orsakerna till att läsning och skrivning i ett synkront DRAM kan ske snabbt, med 133 MHz, dvs med en period av 7,5 ns är flera. En väsentlig orsak är att läsningar och skrivningar sker i *skurar* (eng. *burst*). För vårt minne kan man välja burst-längderna 2, 4 eller 8. Efter inmatning av radadress och kolumnadress så läses eller skrivas automatiskt en burst av ord, 2, 4 eller 8 beroende vad man programmerat minnet för genom uppräkning av radadressen internt med Column-Address-Counter. En annan orsak är att när en bank används så kan en annan bank förberedas genom inmatning av radadress. En annan orsak är själva synkroniseringen med systemklockan, som ger effektivare utnyttjande av tiden. DRAM har i många generationer haft en accesstid på ca 50 ns. Om man bara läser/skriver ett ord i ett DDR SDRAM så blir accesstiden ungefär densamma som i äldre DRAM, förden här är alltså möjligheten att efter en adressering kunna läsa/skriva en skur av ord och förbereda adressering av efterföljande skurar.

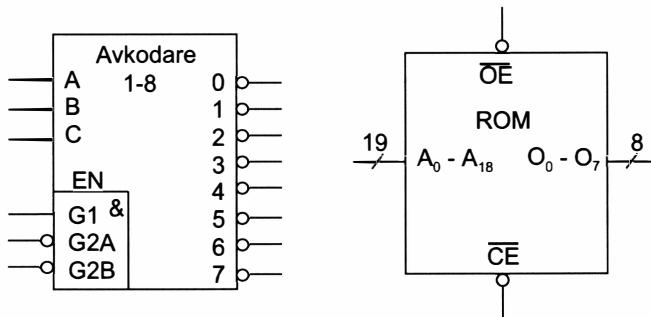
I asynkrona DRAM har man använt tekniker benämnda FPM (Fast Page Mode) och EDO (Extended Data Out) för att snabba upp på varandra följande läsningar och skrivningar.

Det finns ett konkurrerande system till DDR SDRAM, som heter Direct Rambus DRAM eller DRDRAM, utvecklat av företaget Rambus, som bygger på en 16-bitars buss med en klockfrekvens på 400 MHz, där överföringar sker på både positiv och negativ flank, vilket ger en överföringshastighet på 1,6 Gbyte/s.

Slutligen kan noteras i databladet för vårt minne att matningsspänningen  $V_{DD} = 2,5 \text{ V}$ . I äldre generationer var matningsspänningen 5 V för att sedan sänkas till 3,3 V och nu alltså till 2,5 V. – Minneskapslarna monteras i standardmoduler DIMM (Dual In-line Memory Module), 168 pin, se appendix 4, där visas 512 Mbyte DIMM, uppbyggda med 256 Mbit (32M $\times$ 8) SDRAM för 64 bitars ord längd utan paritetsbitar respektive 72 bitars ord längd (med paritetsbitar, s.k. ECC, Error Correction Circuit).

## 7.4 Övningsuppgifter

- 7.1** Ange vilka omkopplingar som skall göras i blockschemat till minnet i exempel 7.2 figur 7.28 för att adressområdet skall bli
- 980000 – AFFFFF
  - 480000 – 5FFFFFF
- 7.2** Ett minne enligt alternativen a) och b) skall realiseras med kapslar Infineon HYB25D256400/800/AT 256 Mbit ( $32M \times 8$ ) DDR SDRAM, se utdrag ur datablad i appendix 4. Ange hur många kapslar som erfordras utefter ord längdsaxeln respektive ordaxeln i figur 7.9.
- a)  $256M \times 64$       b)  $512M \times 72$
- 7.3** I ett minnessystem ingår läsminne och läs/skrivminne. Lässminnet skall byggas upp med minneskapslar  $4M$  ( $512k \times 8$ ), se symbol nedan. Minnessystemet adresseras med 24 adressbitar  $A_0$  –  $A_{23}$ , sålunda omfattande ett adressområde 000000 till FFFFFFF angivet i hexadecimall form. Lässminnet skall ha kapaciteten  $1\text{ M ord} \times 32\text{ bitar}$  och vara placerat i adressområdet 000000 till OFFFFF. Chipselect till lässminnet skall genereras med en avkodare 1-8, till vilken skall anslutas adressbitarna  $A_{19}$  –  $A_{23}$  så att den kan generera chip-select i hela adressområdet 000000 till 3FFFFFF. Rita blockschema för lässminnet och använd då symbolerna nedan. Ange på avkodarens utgångar inom vilket adressområde den ger chipselect. För Enable EN gäller att  $EN = (G1)(G2A)'(G2B)'$ .



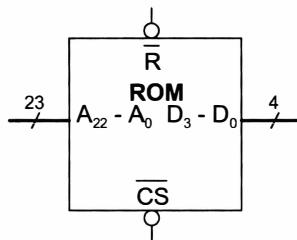
7.4 Enligt 7.3 ovan, men modifierat så att

a) läsminnet skall ha kapaciteten  $2M \text{ ord} \times 16 \text{ bitar}$  och vara placerat i adressområdet A00000 till BFFFFFF. Avkodaren skall generera chip-select i hela adressområdet 800000 – BFFFFFF.

b) läsminnet skall ha kapaciteten  $1 M \text{ ord} \times 24 \text{ bitar}$  och vara placerat i adressområdet 600000 till 6FFFFFF. Avkodaren skall generera chip-select i hela adressområdet 400000 – 7FFFFFF.

7.5 I ett minnessystem ingår läsminne och läs/skrivminne. Läsminnet skall byggas upp med minneskapslar 32Mbit ( $8M \times 4$ ), se symbol nedan. Minnessystemet adresseras med 32 adressbitar A0 – A31, sålunda omfattande ett adressområde 00000000 till FFFFFFFF.

Läsminnet skall ha kapaciteten  $16 M \text{ ord} \times 8 \text{ bitar}$  och vara placerat i adressområdet F3000000 till F3FFFFFF. Chipselect till läsminnet skall genereras med en avkodare 1-16 med aktivt låga utsignaler och en insignal enable, aktivt hög. Till avkodaren skall anslutas adressbitarna A23 - A31 så att den kan generera chip-select i hela adressområdet F0000000 till F7FFFFFF. Rita blockschema för läsminnet. Ange på avkodarens utgångar inom vilket adressområde den ger chipselect.



7.6 Ange vilka modifieringar i blockschemat till minnet i uppgift 7.5 som skall göras för att läsminnet skall ha kapaciteten  $24M \text{ ord} \times 16 \text{ bitar}$  och vara placerat i adressområdet A8800000 till A9FFFFFF. Avkodaren skall generera chip-select i hela adressområdet A8000000 – AFFFFFFF.

# 8 Programmerbara logiska kretsar

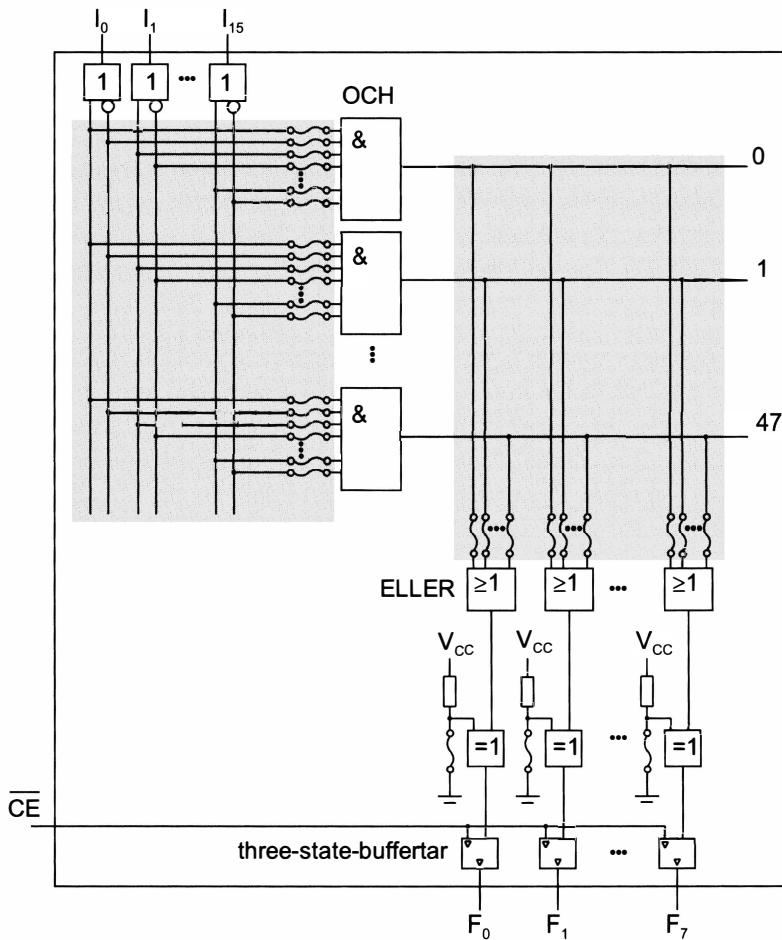
Akronymen *PLD* står för *Programmable Logic Device*, en klass av strukturprogrammerbara kretsar, men den kan också sägas stå för en ny filosofi att konstruera kretsar, där kretsens beteende står i förgrunden under konstruktionsprocessen och kretsens struktur till stor del överläts till syntesverktyget att bestämma.

PLD är ett mycket expansivt område, där kretsarna utvecklats mycket sedan den första egentliga PLDn kom i mitten av 1970-talet. Det finns en rik flora av PLD med olika uppbyggnadsfilosofi och komplexitet. Nya arter tillkommer i en strid ström och det är inte meningsfullt att försöka ge en heltäckande bild av området PLD, dels därför att bilden skulle bli alltför omfattande, dels därför att bilden ändrar sig så snabbt. Presentationen av PLD nedan görs, dels med en historisk översikt med redovisning av några milstolpar i utvecklingen, dels med en kort beskrivning av två kretsar, en CPLD och en FPGA, som får representera ”state of the art” 2001, sammantaget förhoppningsvis en grund för vidare fördjupning inom området som måste ske med informationsmaterial från fabrikanter.

**1975** introducerade företaget Signetics vad som kan anses vara den första egentliga PLDn, *FPLA (Field Programmable Logic Array)* 82S100, sedermera kretsen Signetics Philips PLS100, se principschema i figur 8.1 nedan. Den innehöll programmerbara OCH-ELLER-nät, där både OCH-nivån och ELLER-nivån var programmerbar med avbrännningsbara länkar, enligt samma princip som för PROM i figur 7.13. OCH-nivån hade 48 programmerbara 16-ingångars OCH-grindar. ELLER-nivån hade 8 programmerbara ELLER-grindar som kunde anslutas till önskade OCH-grindar. Med XOR-grinden kunde antingen funktionen eller funktionens invers realiseras, såsom tidigare demonstrerats i kapitel 4. Kretsen hade nackdel av hög effektförbrukning och relativt stor fördröjning.

Vi har i kapitel 7, Halvledarminnen, figurerna 7.7 och 7.8, sett hur en minneskrets kan betraktas som en kombinationskrets och sålunda hur en

boolesk funktion kan realiseras i en minneskrets. I FPLA 82S100 kunde realiseras booleska funktioner med 16 ingångar och 8 utgångar. Om sådana funktioner skulle realiseras i en minneskrets skulle det krävas en minneskapsel med kapaciteten  $2^{16}$  byte = 64 kbyte.



Figur 8.1 Första egentliga PLD, Signetics 82S100, principschema.

Skillnaden mellan realiseringen i minneskapseln och i FPLA är att i minneskapseln programmerar man in hela funktionstabellen, samtliga 64 kbyte, medan i FPLA programmerar man in 8 booleska funktioner på SP-form, där varje SP-form är begränsad till att innehålla en boolesk summa av

---

de tillgängliga 48 OCH-termerna, vardera med maximalt 16 variabler. Detta är ingen allvarlig begränsning då booleska funktioner som skall realiseras normalt har ett mindre antal OCH-termer och man har också möjlighet att minimera den booleska funktionen. Idag skulle vi i stället för FPLA 82S100 kunna använda en minneskrets på 64 k bytes, men det var inte var möjligt 1975, då vi bara befann oss i halvledarminnesteknikens barndom. Det krävs stor kapacitet hos minneskapslar när det gäller att realisera booleska funktioner. Antag att vi skall realisera en boolesk funktion med 24 ingångar och 16 utgångar och 10 produkttermer, vilket är möjligt i ett PLD. Vid realisering i en minneskapsel skulle behövas en kapsel med kapaciteten  $2^{24} \cdot 16$  bitar =  $2^4 \cdot 2^{20} \cdot 16 = 256$  Mbit. Vidare har minneskretsar normalt större fördräjning än PLD.

**1976** introducerade företaget Monolithic Memories (MMI) en snarlik PLD, *PAL (Programmable Array Logic)*, som liksom FPLA innehöll programmerbara OCH-ELLER-nät, men med den skillnaden att endast OCH-nivån var programmerbar. Dessa kretsar blev betydligt snabbare än FPLA, fördräjning 25ns, eftersom man slapp fördräjningen i länkarna i ELLER-nivån och vidare så krävde de mindre kiselyta än FPLA, eftersom man inte hade de utrymmeskrävande länkarna i ELLER-nivån.

**1978** publicerade MMI den första PAL-handboken med en hel familj av olika PAL. I appendix 5 visas ett utdrag ur denna första PAL-handbok "PAL Programmable Array Logic Handbook", First Edition 1978. Författare till boken är John Birkner, som är upphovsmannen till PAL. Beteckningssättet för kapslarna i PAL-familjen framgår under rubriken "Ordering information".

I utdraget visas logikschemat för en av kretsarna i familjen, PAL16R4, som belyser en del intressanta saker. – Till vänster är den programmerbara matrisen där de vertikala ledningarna är förbundna med de horisontella ledningarna med avbränningsbara länkar, som bildar en OCH-funktion, symboliskt visad med en OCH-grind med en ingång. Med alla länkarna intakta är utsignalen från OCH-grinden logiskt 0, eftersom alla insignalvariabler är anslutna både som icke-invers och som invers. Vi ser vidare att kapseln innehåller 4 D-vippor, som är anslutna till OCH-ELLER-näten enligt sekvenskretsmodellen. I kapseln kan alltså realiseras en sekvenskrets med  $2^4 = 16$  tillstånd. PAL-kretsarnas uppbyggnad enligt sekvenskretsmodellen fick konstruktörerna av digitala kretsar att börja tänka i denna modell jämfört med tidigare då det kanske varit lite av "vi tar en vippa här och en grind där ..", och PAL-kretsarna kan därför också sägas vara inledningen till en ny konstruktionsfilosofi.

Låt oss fortsätta studera logikschemat för PAL16R4. – De utgångar som ej har någon D-vippa har också en återkoppling till matrisen, vilken har flera intressanta användningar. Den används då man måste faktorisera en boolesk funktion p.g.a. att i de tillgängliga OCH-ELLER-näten, antalet OCH-grindar eller antalet ingångar hos dessa är för litet, varvid man då kan realisera en del av funktionen i ett av OCH-ELLER-näten utan D-vippa och sedan låta utsignalen från detta gå vidare som insignal till ett annat OCH-ELLER-nät. Den kan också användas då man vill realisera en asynkron sekvenskrets, då ju tillståndet ej lagras i ett tillståndsregister, utan återkopplas direkt, se kapitel 5, Sekvenskretsar, sektion 5.5. Vidare kan den användas för att skapa ytterligare en ingång till kapseln, vilket kan ske genom att man stänger av OCH-ELLER-näts förbindelse med utgångsstiftet genom att lägga three-state-buffertens utgång högohmig, varvid utgångsstiftet kan användas som ingång till matrisen via återkopplingen.

En allvarlig svaghet hos PLA och PAL de första åren var avsaknaden av utvecklingshjälpmödel, syntesverktyg. Konstruktören fick själv efter bästa förmåga skapa programmeringsmönstret, som sedan skrevs in på en blanskett i ett rutmönster av samma utseende som programmeringsmatrisen, där varje ruta svarade mot en länk i matrisen. I rutan markerades med ett kryss om länken skulle vara intakt eller med ett streck om länken skulle brännas av. Innehållet i rutmönstret matades sedan in i programmeringsutrustningen. En viktig milstolpe blev utvecklingshjälpmedlet *PALASM (PAL AssemBlér)* från MMI, som gjorde det möjligt att på dator beskriva kretsen som skall realiseras i PAL med booleska funktioner och sedan få datorn att utföra minimering och automatisk generering av programmeringsmönstret.

PAL har varit mycket populära kretsar och under årens lopp använts i många produkter, bl.a. fanns det i den första generationens persondatorer många PAL som utgjorde ”limmet” mellan de komplexa processorkretssarna. Kretsarnas betydelse avspeglas bl.a. av att ordet PAL ofta har använts som synonym för programmerbar logisk krets, och att andra fabrikanter som sedan konstruerat nya generationer PLD, ofta har sett till att deras kretsar kan *emulera*, efterlikna, härma (eng. *emulate*) PAL och direkt ersätta PAL i äldre konstruktioner.

**1983** kommer man överens om en JEDEC-standard för programmeringsinformationen, vilket underlättar för tillverkare av programmeringsutrustningar. – Syntesverktyg ej knutna till kretsar av ett speciellt fabrikat, generella kompilatorer, introduceras. CUPL (Universal Compiler for Program-mable Logic), från företaget Assisted Technology.

---

Företaget Advanced Micro Devices (AMD) introducerar PAL22V10, en andra generationens PLD, som innehåller programmerbara utgångsmakroceller. Kretsen har liksom tidigare PAL-kretsar en programmerbar OCH-matris, men också programmerbara utgångsmakroceller, som kan konfigureras med programmerbara multiplexrar, enligt samma princip som tidigare visats i kapitel 5, figur 5.51 för en vippa som kan programmeras som en D-vippa eller en T-vippa.

**1984** introducerar företaget Altera, då nyss bildat, de första UV-raderbara PLD, *EPLD (Erasable PLD)* i CMOS-teknik, som bygger på EPROM-tekniken. Alteras första krets EP300 var uppbyggd enligt samma princip som PAL22V10 ovan med makroceller och hade 8 makroceller.

Nya utvecklingshjälpmittel introduceras, såsom AMAZE (Automated Map And Zap Equations) från Signetics, samt en generell kompilator ABEL (Advanced Boolean Expression Language) från Data I/O, som tillverkar programmeringsutrustningar för bl.a. PLD.

**1985** introducerar företaget Lattice Semiconductor, GAL (Generic Array Logic), som är *EEPLD (Electrically Erasable PLD)*, elektriskt raderbara PLD, som bygger på samma typ av transistor som i EEPROM. GAL-kretsar är uppbyggda enligt samma princip som PAL22V10 ovan med makroceller. De har konstruerats med tanke att kunna emulera PAL, det engelska ordet *generic* betyder ungefär ”något som sammanfattar egenskaper hos en hel grupp”. Lattice GAL22V10 finns nu i en version ispGAL22V10, *isp = in-system-programmable*, som kan programmeras sittande i applicationen på kretskortet via kabel ansluten till parallelporten på en PC, ett programmeringsförfarande som nu är JEDEC-standard.

Företaget XILINX, som grundades 1984, introducerar *LCA (Logic Cell Array)*, PLD med en helt ny arkitektur. LCA består av en kvadratisk matris av *konfigurerbara* (programmerbara) logikblock, (eng. *Configurable Logic Block, CLB*), omgivna av en ring av In/Ut-block (eng. *Input/Output Block, IOB*). XILINX markerar att LCA har en speciell struktur som liknar en grindmatris, genom att kalla sina kretsar *FPGA* (eng. *Field Programmable Gate Array*). Programmeringsinformationen ligger här inte i EPROM- eller EEPROM-cell, som ju behåller sin information utan matningsspänning, utan den ligger i RWM-cell, som bara behåller informationen så länge det finns matningsspänning, och programmeringsinformationen måste alltså laddas in igen efter spänningsbortfall. LCA-kapseln kan själv ladda in programmeringsinformationen i seriell form från ett yttre seriellt EPROM.

**1988** introducerar företaget Actel en programmerbar grindmatris där programmeringsinformationen ligger kvar utan matningsspänning. Denna grindmatris liknar en ”vanlig grindmatris” och innehåller inte programmerbara logikblock typ XILINX LCA, utan man konstruerar med primitiver bestående av små multiplexrar.

**1990** börjar det komma VHDL-kompilatorer till syntesverktyg för PLD, bl.a. introducerar företaget Hardi Electronics AB i Lund, en VHDL-kompiler till syntesverktyget LOG/iC.

**1990-talet** karakteriseras av en kraftig utveckling av syntesverktygen och de programmerbara kretsarna samt strukturförändringar i branschen med företagsuppköp, sammanslagningar etc., som resulterat i ett färre antal och större företag.

**2001** är det fortfarande företagen Altera och XILINX, som alltsedan starten i mitten av 1980-talet, går i spetsen för utvecklingen av de programmerbara kretsarna och två kretsfamiljer från dessa företag får ge en antydan om ”state of the art” idag. I appendix 4 visas några data för Altera MAX 7000 och XILINX Virtex-II, som är exempel på två olika kretsfilosofier. Altera MAX 7000 är en CPLD (Complex PLD), en typ av PLD som består av några PLD:er på ett chip, som kan förbindas med varandra via switchmatriser. XILINX Virtex-II är en FPGA (Field-Programmable Gate Array) uppbyggd av ett mycket stort antal programmerbara logikblock utplacerade i ett matrismönster på ett chip.

Max 7000 arkitekturen baseras på Logic Array Blocks (LAB), se databladet Figure 1. MAX 7000A Device Block Diagram, som kan förbindas med varandra via PIA (Programmable Interconnect Area). Varje LAB består i sin tur av 16 makroceller, se Figure 2. MAX 7000A Macrocell. Makrocellen innehåller en vippa som kan programmeras som D/T-vippa och framförliggande kombinationskrets av typ OCH-ELLER-nät för beräkning av nästa tillstånd, jfr sekvenskrestmodellen i figur 5.5. I OCH-ELLER-nätet är anslutningen av insignalerna till OCH-grindarna programmerbar i en matris. Vidare kan OCH-grindarna kombineras med OCH-grindar från en annan LAB med hjälp av Product Term Select Matrix, se Figure 3. MAX 7000A Sharable Expanders. I LAB finns några block betecknade ”Select”. Det är MUX:ar som ger möjlighet att programmera olika signalvägar. Register Bypass möjliggör att leda signalen från OCH-ELLER-nätet förbi D/T-vippan och generera en ren kombinationskrets. Den mest komplexa kretsen innehåller 512 makroceller och har 212 I/O-stift. Kapseln kan program-

---

meras via en kabel ansluten till serieporten på en PC, som tidigare nämnts ett programmeringsförfarande som är JEDEC-standard.

XILINX Virtex-II är en kretsfamilj med mycket hög komplexitet, som framgår av Table 1. Virtex-II Field-Programmable Gate Array Family Members. De mest komplexa kretsarna har en komplexitet motsvarande 10 miljoner grindar och innehåller bl.a. 122880 D-vippor och kapseln har 1108 anslutningspunkter (BGA, Ball Grid Array) för I/O (In-Utgångar) i en kapsel av storleken 40×40 mm! I en sådan komplex krets kan man programmera in en mikroprocessor, typ IBM PowerPC 405 32-bit RISC CPU, en DSP (Digital Signal Processor) eller andra dylika kretsar som man kan köpa av fabrikanten, beskrivna i en fil, som färdiga s.k. IP-block (IP = Intellectual Property) och använda dessa kretsar tillsammans med de kretsar som man själv konstruerar. Kretsarna Virtex-II är uppbyggda i en matris, se Figure 1: Virtex-II Architecture Overview, av Programmerbara I/O block och konfigurerbara logikblock (CLB, Configurable Logic block) som kan förbindas med varandra via horisontella och vertikala ledningar mellan blocken. De konfigurerbara logikblocken CLB består av 4 ”slices”, se Figure 13: Virtex-II Slice Configuration och Figure 14: Virtex-II Slice (Top Half). I en slice finns två D-vippor/D-latchar och framförliggande kombinationskrets som gör beräkning av nästa tillstånd, jfr sekvenskrestmodellen i figur 5.5. Kombinationskretsen som beräknar nästa tillstånd är realiserad i en LUT (Look-Up Table), ett 16 bitars minne, som adresseras med en 4-bitars adress. I en LUT kan sålunda realiseras alla booleska funktioner av fyra variabler (adressen). Aktuell funktionstabell läggs in i minnet. En LUT kan alternativt användas som 16-bitars skiftregister eller 16-bitars RAM. Om man behöver booleska funktioner av fler än 4 variabler så kan flera LUT användas tillsammans med MUX:ar. I en slice kan realiseras en 2-bitars heladderare som kan användas för att realisera en större adderare och för det ändamålet finns också Fast Lookahead Carry Logic till carryaccelerator. En förutsättning för att kunna realisera stora kretsar i en Virtex-II kapsel är självklart tillgång till avancerade syntesverktyg som kan utföra Place and Route av den aktuella kretsen i kapseln

# 9 VHDL – en introduktion

VHDL är ett språk för *beskrivning* av digitala kretsar. VHDL är ett *standardiserat* språk. VHDL medger beskrivning av digitala kretsars *beteende* och *struktur* på olika abstraktionsnivåer. VHDL är ett mycket omfattande språk. Det finns kraftfulla simulatorer för kretsar beskrivna i VHDL. Det finns också kraftfulla syntesverktyg för kretsar beskrivna i VHDL. Syntes är dock jämfört med simulering mer komplicerat och syntesverktygen kan utföra syntes bara för en delmängd av VHDL. Dessutom stödjer olika syntesverktyg olika delmängder av VHDL. I detta kapitel kommer VHDL att introduceras med ett antal enkla exempel som illustrerar principer för beskrivning av kombinations- och sekvenskretsar.

## 9.1 VHDL – bakgrund

VHDL skapades i USA i början av 1980-talet. Initiativet kom från amerikanska försvarsdepartementet vilket, i egenskap av beställare av stora elektroniksystem med många företag inblandade, behövde ett språk för dokumentation av kretsarna i systemen, så att företagen lättare skulle kunna kommunicera med varandra och med beställaren, och så att kretsarna vid eventuella framtida modifieringar och utbyte skulle vara ordentligt dokumenterade.

Det var ett stort projekt för framställning av mycket komplexa digitala kretsar, stött av det amerikanska försvarsdepartementet, det s.k. VHSIC-projekttet (Very High Speed Integrated Circuit), som visade på nödvändigheten av ett gemensamt språk för digitala kretsar. Detta projektnamn finner vi också i akronymen *VHDL*, som står för *Very High Speed Integrated Circuit Hardware Description Language*. En första version kom 1985. Organisationen IEEE (Institute of Electrical and Electronic Engineers) började 1986 utarbeta en standard för VHDL, vilket december 1987 utmynnade i VHDL-87, VHDL-standard IEEE Std 1076-1987, som är dokumenterad i ”IEEE

Standard VHDL Language Reference Manual". Revidering skall ske vart 5:e år. Nästa standard kom 1993, VHDL-93, IEEE Std 1076-1993. VHDL-93 skiljer sig marginellt från VHDL-87.

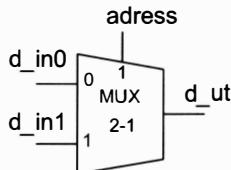
## 9.2 Beskrivning av kombinationskretsar

I kapitel 1 har redan de grundläggande begreppen *entity* och *architecture* introducerats i samband med beskrivningen av kombinationskretsen BCD-check i VHDL. Där presenterades också bl.a. *port()*, datatyperna *std\_logic* och *std\_logic\_vector()*, tilldelningsoperatorn  $<=$ , logiska operatorerna *not*, *and*, *or*, *nand*, *nor*, *xor* och *xnor*, samt satsen *when-else*. Vi skall börja med att rekapsitulera dessa begrepp i ett par små exempel.

### Parallelta (concurrent) signaltilldelningar

#### *Exempel 9.1*

Beskrivning av en MUX 2-1 (se kapitel 2, figur 2.9).



Figur 9.1 MUX 2\_1.

Lösning

```
--MUX2_1
--2001-07-27 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX2_1 is
  port(d_in0, d_in1, adress:in std_logic;
        d_ut: out std_logic);
end entity MUX2_1;

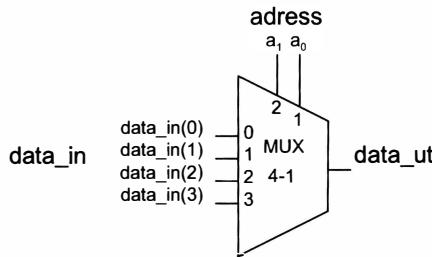
architecture beteende of MUX2_1 is
begin
  d_ut <= (not adress and d_in0) or (adress and d_in1);
end architecture beteende;
```

□

Beskrivningen av MUX:en i exemplet ovan är på en mycket låg abstraktionsnivå, en beskrivning av strukturen med grindar i realiseringen i figur 2.9.

### *Exempel 9.2*

Beskrivning av en MUX 4-1 (se kapitel 2, figur 2.12).



Figur 9.2 MUX 4-1.

### *Lösning*

--MUX4\_1  
--2001-07-27 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

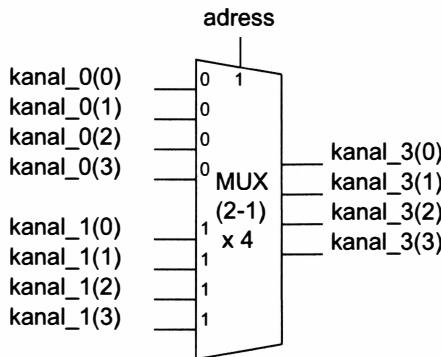
entity MUX4_1 is
  port(data_in: in std_logic_vector(3 downto 0);
       a:in std_logic_vector(1 downto 0);
       data_ut: out std_logic);
end entity MUX4_1;

architecture beteende of MUX4_1 is
begin
  data_ut:=(not a(1) and not a(0) and data_in(0)) or
            (not a(1) and a(0) and data_in(1)) or
            (a(1) and not a(0) and data_in(2)) or
            (a(1) and a(0) and data_in(3));
end architecture beteende;
```

□

**Exempel 9.3**

Beskrivning av en MUX (2-1)×4 (se kapitel 2, figur 2.16).



Figur 9.3 MUX (2-1)×4.

**Lösning**

```
--MUX2_1x4
--2001-07-27 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX2_1x4 is
    port(kanal_0, kanal_1: in std_logic_vector(3 downto 0);
          adress:in std_logic;
          kanal_3: out std_logic_vector(3 downto 0));
end entity MUX2_1x4;

architecture beteende of MUX2_1x4 is
begin
    kanal_3 <= kanal_0 when adress = '0' else
                    kanal_1
end architecture beteende;
```

□

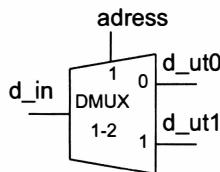
Beskrivningen i exempel 9.3 ovan med *when-else* är på en något högre abstraktionsnivå än de två första exemplen. Ett alternativ till *when-else* kan vara med *with-select-when* enligt nedan.

```
with adress select
    kanal_3 <= kanal_0    when '0',
                  kanal_1  when '1',
                  null      when others;
```

Samtliga alternativ måste finnas med i *with-select-when*. Om man vill samla alla återstående alternativ på sista raden i *with-select-when* kan man göra det med *when others*. Std\_logic innehåller mer än '0' och '1' och därför tillfogar vi *when-others*. *null* betyder ”gör ingenting”.

#### Exempel 9.4

Beskrivning av en DMUX 1-2 (se kapitel 2, figur 2.18).



Figur 9.4 DMUX 1-2.

#### Lösning

```
--DMUX1_2  
--2001-07-27 Lars-H Hemert
```

```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity DMUX1_2 is  
    port(d_in, adress:in std_logic;  
          d_ut: out std_logic_vector(1 downto 0));  
end entity DMUX1_2;  
  
architecture beteende of DMUX1_2 is  
begin  
    d_ut(0) <= (not adress and d_in);  
    d_ut(1) <= (adress and d_in);  
  
end architecture beteende;
```

□

I våra VHDL-beskrivningarna så följer efter de inledande kommentarerna:

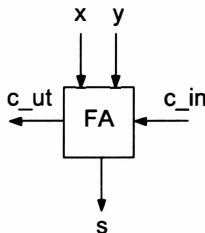
```
library IEEE;  
use IEEE.std_logic_1164.all;
```

std\_logic\_1164 är namnet på en s.k *package*, som är definierad av IEEE. En package är en fil som innehåller diverse definitioner som skall användas vid kompileringen av VHDL-beskrivningen. I std\_logic\_1164 är bl. a. datatyperna std\_logic och std\_logic\_vector definierade.

**use IEEE.std\_logic\_1164.all** anger att package std\_logic\_1164 som ligger biblioteket IEEE skall användas och att hela (*all*) package skall användas. Om bara en del skall användas anger man i stället för *all* namnet på den del som skall användas. Före direktivet *use* måste man ange vilka bibliotek som skall användas med direktivet *library*. I vårt fall skall användas biblioteket IEEE, varför vi skriver *library IEEE*.

### Exempel 9.5

Beskrivning av en heladderare (FA) (se kapitel 4, figur 4.45)



Figur 9.5 Heladderare (FA).

### Lösning

```
--FA, heladderare
--2001-07-27 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FA is
    port(x, y, c_in:in std_logic;
          s, c_ut: out std_logic);
end entity FA;

architecture beteende of FA is
begin
    c_ut  <= (x and y) or (x and c_in) or (y and c_in);
    s     <= x xor y xor c_in;
end architecture beteende;
```

□

## Process

Signaltilldelningssatserna i exemplen ovan är, som nämntes i kapitel 1, *parallella, samtidiga*, satser (eng. *concurrent statements*), som utförs när någon förändring inträffar hos signalerna i högerledet av tilldelningssatserna. Satsernas ordning i beskrivningen har därför ingen betydelse. De parallella satser vi sett exempel på har varit beskrivning av betenden på relativt låg nivå. Vid beskrivning av beteenden på högre nivå behöver man tillgång till satser som exekveras sekventiellt, speciellt gäller detta vid beskrivning av beteende för sekvenskretsar utgående från tillståndsdiagrammet.

Beskrivning med sekventiella satser görs i en *process*. I en process kan bl.a. användas *if-satser* och *case-satser*. En process är alltså lämplig för beskrivning av sekvenskretsar, men kan också med fördel p.g.a. tillgången till if-satser användas för beskrivning av kombinationskretsar på högre nivå. Låt oss direkt beskriva MUX:en i exempel 9.1 med en process och sedan kommentera beskrivningen.

### Exempel 9.6

Beskrivning av en MUX 2-1 i en process (se exempel 9.1).

*Lösning*

```
--MUX2_1
--2001-07-27 Lars-H Hemert
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX2_1 is
  port(d_in0, d_in1, adress:in std_logic;
        d_ut: out std_logic);
end entity MUX2_1;

architecture beteende of MUX2_1 is
begin
  process(d_in0, d_in1, adress)
  begin
    if adress = '0' then
      d_ut <= d_in0;
    else
      d_ut <= d_in1;
    end if;
  end process;
end architecture beteende;
```



Hela processen är själv en samtidig sats som kan ingå tillsammans med andra samtidiga satser eller andra processer i en architecture. Processen är *vilande* tills någon av signalerna inom parentesen i *process( )* förändras, då satserna i processen mellan *begin* och *end process*, utförs sekventiellt. Signalerna inom parentesen benämnes processens *känslighetslista* (eng. *sensitivity list*).

I processen ovan ingår en *if-sats* vars syntax är:

```
if logiskt uttryck then
    sats(-er);
elsif logiskt uttryck then
    sats(-er);
.
.
.
elsif logiskt uttryck then
    sats(-er);
else
    sats(-er);
end if;
```

Satserna *elsif* och *else* behöver ej vara med. If-satsen får ej användas vid parallella satser. Notera stavningen av *elsif* i ett ord och att *end if* är två ord.

Kompileringen av beskrivningen av MUX:en i processen ovan bör leda fram till samma booleska uttryck för syntesen som beskrivningen i exempel 9.1 med samtidiga satser. Skillnaden mellan de två beskrivningarna är att beskrivningen i processen är på en något högre nivå och mer en beskrivning av MUX:ens funktion (beteende), medan den första beskrivningen med logiska operatorer ligger närmare hur multiplexern realiseras med grindar (strukturen).

Låt oss nu illustrera användning av en *case*-sats vid beskrivning av MUX:en i exempel 9.2 med en process.

### **Exempel 9.7**

Beskrivning av en MUX 4-1 i en process (se exempel 9.2).

#### *Lösning*

```
--MUX4_1
--2001-07-27 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity MUX4_1 is
  port(data_in: in std_logic_vector(3 downto 0);
       a:in std_logic_vector(1 downto 0);
       data_ut: out std_logic);
end entity MUX4_1;

architecture beteende of MUX4_1 is
begin
  process(data_in, a)
  begin
    case a is
      when "00"    => data_ut <= data_in(0);
      when "01"    => data_ut <= data_in(1);
      when "10"    => data_ut <= data_in(2);
      when "11"    => data_ut <= data_in(3);
      when others => null;
    end case;
  end process;
end architecture beteende;
```



Syntaxen för *case*-satsen är:

```
case uttryck is
  when alternativ 1      => sats(-er);
  when alternativ 2      => sats(-er);
  .
  .
  when alternativ n      => sats(-er);
  when others            => sats(-er);
end case;
```

Samtliga alternativ av uttrycket måste finnas med i *case*-satsen. Eftersom std\_logic innehåller mer än '0' och '1' måste *when others* tillfogas. *null* betyder ”gör ingenting”.

I stället för att använda en process för att beskriva MUX:en skulle man kunna använda *with\_select\_when* enligt nedan.

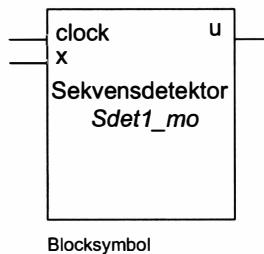
```
architecture beteende of MUX4_1 is
begin
  with a select
    data_ut <= data_in(0) when "00",
                data_in(1) when "01",
                data_in(2) when "10",
                data_in(3) when "11",
                <= null      when others;
end architecture beteende;
```

Beskrivningen blir dock mer lättläst med *case*-sats.

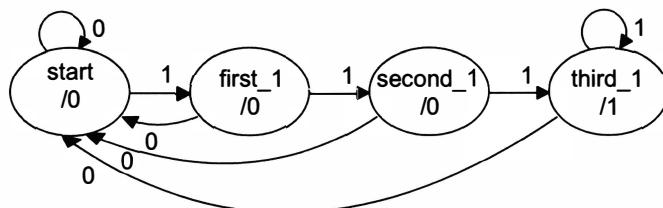
## 9.3 Beskrivning av generella sekvenskretsar

### Exempel 9.8

Beskrivning av sekvensdetektorn Sdet1\_mo (se kapitel 5, sektion 5.1).



Figur 9.6 Blocksymbol för sekvensdetektorn Sdet1\_mo.



Figur 9.7 Tillståndsdiagram för sekvensdetektorn Sdet1\_mo.

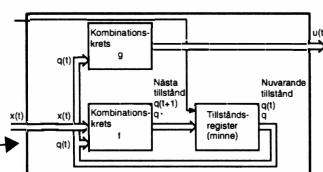
Lösning

```
--Sdet1_mo
--2001-07-28 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Sdet1_mo is
    port(x, clock: in std_logic;
         u: out std_logic);
end entity Sdet1_mo;

architecture beteende of Sdet1_mo is
    -- deklaration av tillståndstyp
    type state_type is (start, first_1, second_1, third_1);
    -- deklaration av nuvarande tillstånd och nästa tillstånd
    signal present_state, next_state: state_type;
```



```

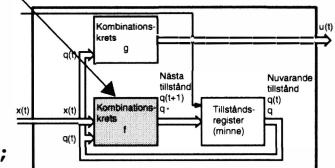
begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, x)
begin
    case present_state is
        when start => if x = '0' then
            next_state <= start;
        else
            next_state <= first_1;
        end if;

        when first_1 => if x = '0' then
            next_state <= start;
        else
            next_state <= second_1;
        end if;

        when second_1=> if x = '0' then
            next_state <= start;
        else
            next_state <= third_1;
        end if;

        when third_1=> if x = '0' then
            next_state <= start;
        else
            next_state <= third_1;
        end if;
    end case;
end process;

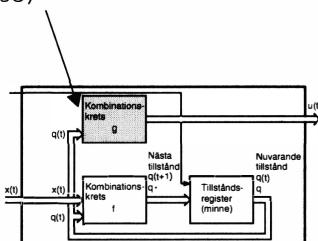
```



```

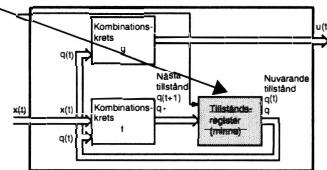
-- kombinatorisk process för beräkning av utsignaler
utsignaler_Moore: process(present_state)
begin
    case present_state is
        when start      => u <= '0';
        when first_1   => u <= '0';
        when second_1  => u <= '0';
        when third_1   => u <= '1';
    end case;
end process;

```



```
-- inmatning av nästa tillstånd i tillståndsregistret
state_register: process(clock)
begin
  if rising_edge(clock) then
    present_state <= next_state;
  end if;
end process;

end architecture beteende;
```



□

### Kommentarer till beskrivningen av sekvensdetektorn Sdet1\_mo

När det gäller att skriva ”vanliga” program i t.ex. Java eller C, så kan man skriva programmen på många olika sätt. Samma sak gäller för VHDL, det går att beskriva kretsar på många olika sätt. Ledstjärnan bör såväl i ”vanlig” programmering som i beskrivningar i VHDL vara *läsbarhet!* Beskrivningen ovan av sekvensdetektorn har gjorts direkt utgående från modellen av en sekvenskrets typ Moore i figur 5.5. I stället för att som ovan göra beskrivningen i tre olika processer, så skulle man kunna göra beskrivningen i en process. Det blir mindre att skriva, men till priset av sämre läsbarhet. Det kan vara värt besväret att göra beskrivningen som ovan och försöka följa den principen genomgående, det blir lättare att gå tillbaka och analysera tidigare gjorda beskrivningar om man följer en given mall.

Processerna har namnetts med en *etikett* (eng. *label*) följd av kolon (:), som är valfritt att göra, men kan vara lämpligt.

Tillståndsdiagrammet beskrivs som synes mycket tydligt med case-satser. *Nuvarande tillstånd* och *nästa tillstånd* har benämnts *present\_state* respektive *next\_state*, namn vi kommer att använda i fortsättningen. De har deklarerats som *signaler*. I signaldeklarationen skall anges *typ*, som här getts namnet *state\_type*, som vi också kommer att använda i fortsättningen, en egendefinierad typ som definierats ovanför signaldeklarationen. Typen *state\_type* omfattar de namn vi ger tillstånden. Den är en *uppräknad* (eng. *enumerated*) typ, ordningen i vilken vi anger tillståndsnamnen har alltså betydelse, t.ex. för tillståndskodningen.

I processen *state\_register* sker klockning av sekvenskretsen. *rising\_edge()* är en fördefinierad funktion i VHDL-93. Det är en funktion av typ *boolean*, som returnerar värdet ”true”, då signalen inom parentesen har en positiv flank. Motsvarande funktion för negativ flank heter *falling\_edge()*.

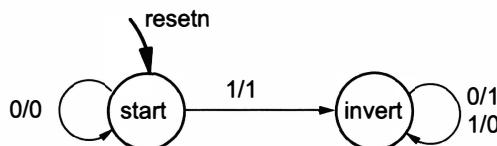
De tre processerna är vilande tills en förändring inträffar i någon signal i en sensitivity list (signalerna i processens parentes), då processen exekveras. Exempelvis då en positiv flank i klocksignalen inträffar exekveras processen *state\_register*, varvid nuvarande tillstånd (*present\_state*) får ett nytt värde. Om detta tillstånd skiljer sig från det föregående tillståndet exekveras processerna *state\_diagram\_Moore* och *utsignaler\_Moore*, eftersom nuvarande tillstånd (*present\_state*) ingår i processernas sensitivity list, och processerna beräknar nytt nästa tillstånd respektive nya utsignaler. På samma sätt exekveras processen *state\_diagram\_Moore* då insignalen *x* ändras.

### Exempel 9.9

Beskrivning av sekvenskretsen TwoComp av typ Mealy, som bildar 2-komplementet till ett binärtal.



Figur 9.8 Sekvenskretsen TwoComp som bildar 2-komplementet till ett binärtal.



Figur 9.9 Tillståndsdiagram för sekvenskretsen TwoComp av typ Mealy.

### Lösning

--TwoComp\_me  
--2001-07-28 Lars-H Hemert

```

library IEEE;
use IEEE.std_logic_1164.all;

entity TwoComp_me is
    port(x, resetn, clock: in std_logic;
         u: out std_logic);
end entity TwoComp_me;

```

```

architecture beteende of TwoComp_me is

-- deklaration av tillståndstyp
type state_type is (start, invert);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Mealy: process(present_state, x, resetn)
begin
if resetn = '0' then
    next_state <= start;
else
    case next_state is
        when start      => if x = '0' then
            next_state <= start;
        else
            next_state <= invert;
        end if;
        when invert     => next_state <= invert;
    end case;
end if;
end process;
```

```

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state, x)
begin
    case present_state is
        when start      => if x = '0' then
            u <= '0';
        else
            u <= '1';
        end if;
        when invert     => if x = '0' then
            u <= '1';
        else
            u <= '0';
        end if;
    end case;
end process;
```

```
-- inmatning av nästa tillstånd i tillståndsregistret
state_register: process(clock)
begin
  if rising_edge(clock) then
    present_state <= next_state;
  end if;
end process;

end architecture beteende;
```

□

### Synkron reset

I beskrivningen av TwoComp\_me ovan har vi behandlat synkron reset som en vanlig insignal. Syntesverktygen kan skilja sig åt hur de tolkar och kompilerar reset. För de flesta syntesverktygen kan det vara lämpligt att införa reset i klockningsprocessen state\_register enligt nedan.

```
-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron reset
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= start;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
```

Notera att signalen *resetn* ej skall finnas med i processens sensitivity list, eftersom processen skall ej exekveras när *resetn* ändras, utan på klocksignalens positiva flank (då en eventuell reset skall ske).

### Asynkron reset

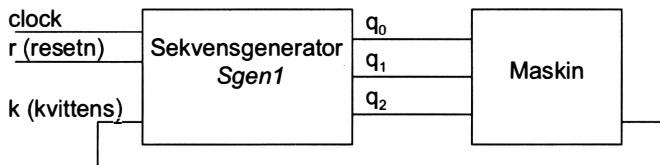
Asynkron reset kan införas i klockningsprocessen state\_register enligt nedan.

```
-- inmatning av nästa tillstånd i tillståndsregistret
-- asynkron reset
state_register: process(clock, resetn)
begin
  if resetn = '0' then
    present_state <= start;
  elsif rising_edge(clock) then
    present_state <= next_state;
  end if;
end process;
```

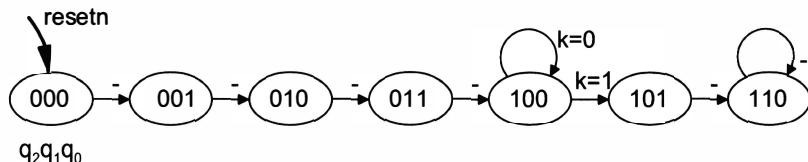
I processen ovan med asynkron reset skall resetn vara med i processens sensitivity list eftersom reset skall kunna ske när som helst utan medverkan av klocksignalen. Syntesverktygen kompilerar denna typ av reset som en asynkron reset, som ansluts till D-vippornas Clear.

### Exempel 9.10

Beskrivning av sekvensgeneratorn Sgen1 (se kapitel 5, exempel 5.3).



Figur 9.10 Sekvensgenerator Sgen1.



Figur 9.11 Tillståndsdiagram för sekvensgeneratorn Sgen1.

### Lösning

```
--Sgen1
--2001-07-28 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Sgen1 is
    port(kvittens, resetn, clock: in std_logic;
         q: out std_logic_vector(2 downto 0));
end entity Sgen1;

architecture beteende of Sgen1 is
    -- deklaration av tillståndstyp
    subtype state_type is integer range 0 to 6;
    -- deklaration av nuvarande tillstånd och nästa tillstånd
    signal present_state, next_state: state_type;
```

```
begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, kvittens)
begin
  case present_state is
    when 0 to 3    => next_state <= present_state +1;
    when 4          => if kvittens = '0' then
                           next_state <= present_state;
                         else
                           next_state <= present_state +1;
                         end if;
    when 5          => next_state <= present_state +1;
    when 6          => next_state <= present_state;
  end case;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,3);

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron reset
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= 0;
    else
      present_state <= next_state;
    end if;
  end if;
end process;

end architecture beteende;
```



### Kommentarer till beskrivningen av Sgen1

Typdeklarationen av tillstånden har gjorts som ett *heltalsområde* 0 till 6. Ett motiv för att definiera tillstånden som heltal är att det då går att göra aritmetiska operationer på tillstånden, som gjorts ovan i processen *state\_diagram*. Ett annat motiv är att tillståndskodningen är given och att utsignalerna skall utgöras av tillståndsregistrets utsignaler. Skulle man inte kunna definiera tillstånden som *std\_logic\_vector(2 downto 0)*? Nej, ty i *case-satsen* går det inte att med *when* ange ett område för en vektor.

Utsignalerna skall ju utgöras av tillståndsregistrets utsignaler. I VHDL är det i princip inte tillåtet att tilldela en signal värdet av en annan signal om de är av olika datatyp. Vi har definierat utsignalen *q* av typen *std\_logic\_vector*, medan *present\_state* är av typen *integer*. För att lösa sådana problem så finns det en del fördefinierade funktioner för typkonvertering i VHDL. De ingår i package *std\_logic\_arith* som ovan tillfogats i ett use-direktiv. Vi har använt konverteringsfunktionen

```
conv_std_logic_vector(arg:integer, size:integer),
```

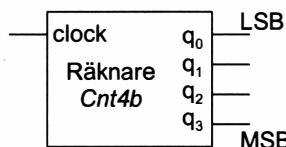
som returnerar en *std\_logic\_vector* motsvarande heltalet *arg* med *size* antal bitar. För omvandling åt andra hållet finns funktionen

```
conv_integer(arg:std_logic_vector)
```

## 9.4 Beskrivning av räknare

### *Exempel 9.11*

Beskrivning av binärräknaren *Cnt4b* som räknar modulo-16 (se kapitel 5, figur 5.43)



Figur 9.12 4-bitars binärräknare *Cnt4b* som räknar modulo-16.

### Lösning

```
--Cnt4b
--2001-07-29 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cnt4b is
  port(clock: in std_logic;
       q: out std_logic_vector(3 downto 0));
end entity Cnt4b;
```

```
architecture beteende of Cnt4b is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 15;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state)
begin
  if present_state = 15 then
    next_state <= 0;
  else
    next_state <= present_state + 1;
  end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,4);

-- inmatning av nästa tillstånd i tillståndsregistret
state_register: process(clock)
begin
  if rising_edge(clock) then
    present_state <= next_state;
  end if;
end process;

end architecture beteende;
```

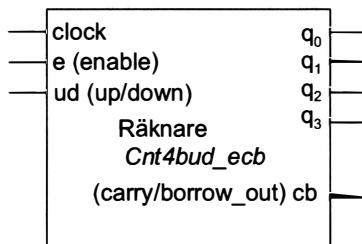


### Kommentar till beskrivningen

Notera hur kompakt en räknare kan beskrivas! Test av räknarens maxvärde måste göras annars ger simulatorn felindikering.

**Exempel 9.12**

Beskrivning av en 4-bitars upp/ned-binärräknare *Cnt4bud\_ecb* med enable och carry/borrow\_out som räknar modulo-16 (se kapitel 5, figur 5.58).



Figur 9.13 4-bitars upp/ned-binärräknare *Cnt4bud\_ecb*.

**Lösning**

```

--Cnt4bud_ecb
--2001-07-29 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cnt4bud_ecb is
    port(enable, up_down, clock: in std_logic;
         q: out std_logic_vector(3 downto 0);
         carry_borrow: out std_logic);
end entity Cnt4bud_ecb;

architecture beteende of Cnt4bud_ecb is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 15;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, enable, up_down)
begin
    if enable = '0' then
        next_state <= present_state;
    else

```

```
if up_down = '1' then
    if present_state = 15 then
        next_state <= 0;
    else
        next_state <= present_state + 1;
    end if;
else
    if present_state = 0 then
        next_state <= 15;
    else
        next_state <= present_state - 1;
    end if;
end if;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state, enable, ud)
begin
    if enable = '1' and
        ((present_state = 15 and up_down = '1') or
         (present_state = 0 and ud = '0')) then
        carry_borrow <= '1';
    else
        carry_borrow <= '0';
    end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,4);

-- inmatning av nästa tillstånd i tillståndsregistret
state_register: process(clock)
begin
    if rising_edge(clock) then
        present_state <= next_state;
    end if;
end process;

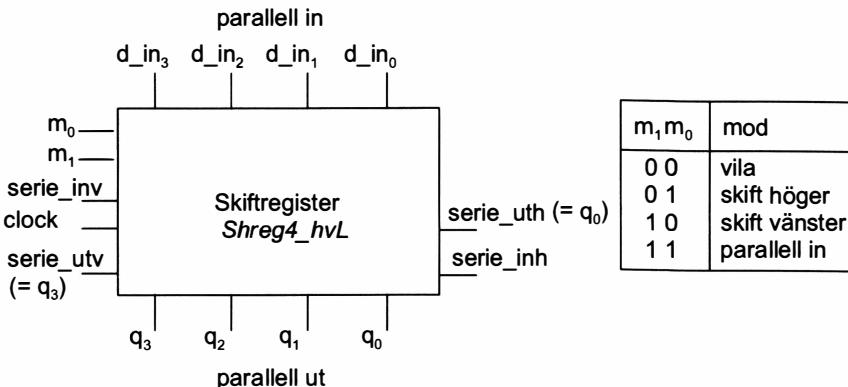
end architecture beteende;
```



## 9.5 Beskrivning av skiftregister

### Exempel 9.13

Beskrivning av skiftregister *Shreg4\_hvL* (se kapitel 5, figur 5.69).



Figur 9.14 Skiftregister *Shreg4\_hvL*.

### Lösning

--Shreg4\_hvL  
--2001-07-29 Lars-H Hemert

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Shreg4_hvL is
  port(serie_inv,serie_inh, clock: in std_logic;
        d_in: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        m: in std_logic_vector(1 downto 0));
end entity Shreg4_hvL;

architecture beteende of Shreg4_hvL is
  signal shift_reg: std_logic_vector(3 downto 0);
begin
  process(clock)
  begin
    if rising_edge(clock) then

```

```
if m = "00" then
    shift_reg(3 downto 0) <= shift_reg(3 downto 0);
elsif m = "01" then
    shift_reg(3 downto 0) <= serie_inv & shift_reg(3 downto 1);
elsif m = "10" then
    shift_reg(3 downto 0) <= shift_reg(2 downto 0) & serie_inh;
else
    shift_reg(3 downto 0) <= d_in(3 downto 0);
end if;
end if;
end process;
q <= shift_reg;
end architecture beteende;
```

□

### Kommentarer till beskrivningen av skiftregistret

Ett skiftregister är inte lämpligt att beskriva med tillstånd. I stället beskrivs på låg nivå hur skiftregistrets positioner kommunicerar med varandra, i princip en strukturbeskrivning hur D-vipporna är sammankopplade enligt figur 5.70. Utsignalerna q används som insignalen inne i skiftregistret. Utifrån betraktat så är q-signaler endast utsignalen och bör därför inte deklareras som *inout* utan som *out*. Vi löser detta problem genom att deklarera en intern signal *shift\_reg*, som kan användas både som insignal och utsignal inne i skiftregistret. Utsignalen q tilldelas sedan värdet hos denna signal med `q <= shift_reg;`. Notera också ovan hur vektortilldelningarna görs. Man kan som synes sammanbinda, hoplänka (eng. *concatenate*) två vektorer med operatorn *&*.

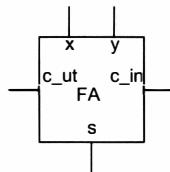
## 9.6 Strukturbeskrivning

Kretsarna som vi beskrivit, MUX:ar, DMUX:ar, räknare, skiftregister m.fl. kan användas som *komponenter* och sammankopplas till större kretsar. Beskrivningen av sammankopplingen kan göras i VHDL, i en *strukturbeskrivning*. En krets som man skapat genom sammankoppling av flera kretsar skall kanske i sin tur vara en komponent som skall sammankopplas med andra kretsar o.s.v. En större krets består av en hierarki av olika nivåer. Principen för strukturbeskrivning i VHDL skall illustreras med ett enkelt exempel nedan, där vi använder en heladderare FA, som tidigare beskrivits,

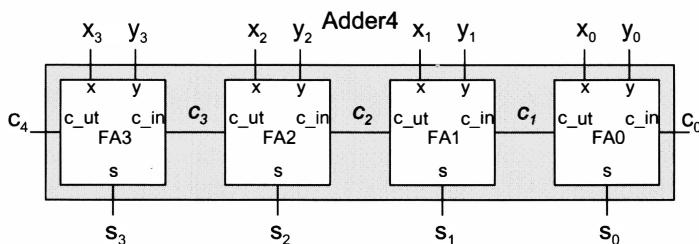
som en komponent för att bygga upp en 4-bitars adderare. Vi går alltså upp en nivå i hierarkin.

### Exempel 9.14

Beskrivning av en 4-bitars adderare *Adder4* uppbyggd av fyra heladderare FA, som tidigare beskrivits i exempel 9.5.



Figur 9.15 Tillgänglig komponent, heladderare FA.



Figur 9.16 4-bitars adderare Adder4, som shall beskrivas.

### Lösning

--Adder4  
--2001-07-29 Lars-H Hemert

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Adder4 is
    port(x, y: in std_logic_vector(3 downto 0);
          c0: in std_logic;
          s: out std_logic_vector(3 downto 0);
          c4: out std_logic);
end entity Adder4;

```

```
architecture struktur of Adder4 is

--komponentdeklaration
component FA
    port(c_in, x, y: in std_logic;
          c_ut, s: out std_logic);
end component;

--deklaration av interna signaler
signal c1, c2, c3: std_logic;

--beskrivning av komponentförbindningarna
begin
    FA0: FA
        port map(c_in => c0, x => x(0), y => y(0),
                  s => s(0), c_ut => c1);
    FA1: FA
        port map(c_in => c1, x => x(1), y => y(1),
                  s => s(1), c_ut => c2);
    FA2: FA
        port map(c_in => c2, x => x(2), y => y(2),
                  s => s(2), c_ut => c3);
    FA3: FA
        port map(c_in => c3, x => x(3), y => y(3),
                  s => s(3), c_ut => c4);
end architecture struktur;
```



### Kommentarer till beskrivningen av Adder4

Beskrivningen inleds på vanligt sätt med en *entity* som beskriver kretsen, i detta fall *Adder4*, sedd utifrån med sina portar. I *architecture* beskrivs här *strukturen* för *Adder4* och vi har därför valt namnet *struktur*. Architecture inleds med deklaration av vilka olika typer av *komponenter* som skall ingå i kretsen. I detta fall skall bara ingå en typ av komponet, en heladderare FA. Syntaxen för komponentdeklarationen är

```
component namn
    port();
end component;
```

I komponentdeklarationen måste `port()` ha exakt samma utseende som i entity för komponenten.

Efter komponentdeklarationen deklarerar *interna* signaler med vilka dataöverföring sker mellan komponenternas portar. I detta fall deklarerar interna signaler c1, c2 och c3 för överföring av carry.

Därefter följer s.k. *komponentinstansiering*. Komponenterna som deklarerats i komponentdeklarationen kan användas flera gånger i kretsen genom att man skapar *kopior*, "instansieringar" (eng. *instance*) av dem. I vår krets Adder4 använder vi fyra heladderare FA och har sålunda gjort fyra instansieringar av FA som vi kallat FA0, FA1, FA2 och FA3. Varje instansiering av en komponent skall ha ett unikt namn. För varje instansierad komponent beskrivs förbindningen med andra komponenter med `port map()`. Syntaxen för komponentinstansieringen är

```
instansnamn: komponentnam  
port map(p1 => s1, p2 => s2, ... )
```

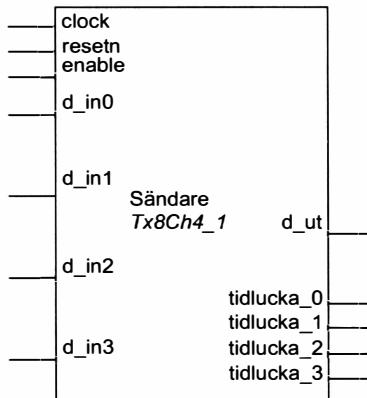
I `port map()` beskrivs *avbildningen* (eng. *map* = sv. *avbilda*) av komponentens portar p1, p2, ... på de interna signalerna och portarna s1, s2, ... hos den stora kretsen, i princip en *nätlista*.

Komponentinstansieringarna avslutar strukturbeskrivningen.

Vid kompilering av strukturbeskrivningen måste kompilatorn naturligtvis ha tillgång till de deklarerade komponenternas VHDL-beskrivningar via sökvägar till kataloger där de finns el. dyl.

### Exempel 9.15

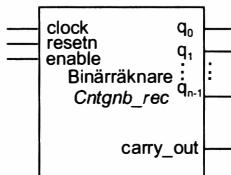
Beskriv strukturen för sändaren Tx8Ch4\_1 enligt schemat i figur 5.61.



Figur 9.17 Blocksymbol för sändaren Tx8Ch4\_1, enligt figur 5.60.

*Lösning*

I schemat för Tx8Ch4\_1 ingår två binärräknare, en 3-bitars och en 2-bitars. Vi har tidigare beskrivit en 4-bitars binärräknare. I stället för att behöva beskriva varje binärräknare med nytt antal bitar, vore det önskvärt att kunna göra en generell beskrivning där antalet bitar definieras av en parameter. Detta är möjligt i VHDL, man kan göra s.k. *generiska* (eng. *generic*) beskrivningar som innehåller parametrar, vars värde anges när man skall använda komponenten. Principen skall visas nedan för en binärräknare med n bitar, resetn, enable och carry\_out som vi benämner *Cntgnb\_rec* och som skall användas i konstruktionen av Tx8Ch4\_1.



*Figur 9.18 Generisk n-bitars binärräknare med resetn, enable och carry\_out.*

--Cntgnb\_rec  
--2001-07-30

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cntgnb_rec is
    generic(n: positive := 4;
            max_count: integer := 15);
    port(resetn, enable, clock: in std_logic;
          carry_out: out std_logic;
          q: out std_logic_vector((n-1) downto 0));
end entity Cntgnb_rec;

```

```

architecture beteende of Cntgnb_rec is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to max_count;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, enable)
begin
    if enable = '0' then
        next_state <= present_state;
    else
        if present_state = max_count then
            next_state <= 0;
        else
            next_state <= present_state + 1;
        end if;
    end if;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state, enable)
begin
    if enable = '1' and present_state = max_count then
        carry_out <= '1';
    else
        carry_out <= '0';
    end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,n);

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron reset
state_register: process(clock)
begin
    if rising_edge(clock) then
        if resetn = '0' then
            present_state <= 0
        else
            present_state <= next_state;
        end if;
    end if;
end process;
end architecture beteende;

```

Deklaration av generiska parametrar görs först i entity med **generic()**. I räknaren ovan har vi deklarerat parametrarna *n*, antal bitar hos räknaren, och *max\_count*, maximala värdet som räknaren skall kunna anta. Parametrarna har tilldelats värden (notera tilldelningsoperatorn :=) så att räknaren kan simuleras och kompileras. Parametrarna kan tilldelas andra värden med **generic map()**, som vi skall se senare, när räknaren skall användas som komponent och vi skall instansiera räknaren.

Multiplexern MUX4\_1 har vi beskrivit i exempel 9.2. Låt oss nu beskriva avkoden *Avkod1\_4*.

```
--Avkod1_4
--2001-07-30 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Avkod1_4 is
    port(adress:in std_logic_vector(1 downto 0);
          data_ut: out std_logic_vector(3 downto 0));
end entity Avkod1_4;

architecture beteende of Avkod1_4 is
begin
    process(adress)
    begin
        case(adress) is
            when "00"      => data_ut <= "0001";
            when "01"      => data_ut <= "0010";
            when "10"      => data_ut <= "0100";
            when "11"      => data_ut <= "1000";
            when others   => null;
        end case;
    end process;
end architecture beteende;
```

Vi har beskrivit alla komponenter som skall ingå i kretsen Tx8Ch4\_1, och övergår nu till att beskriva strukturen.

```

--Tx8Ch4_1
--2001-07-30 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Tx8Ch4_1 is
  port(resetn, enable, clock: in std_logic;
        d_in: in std_logic_vector(3 downto 0);
        d_ut: out std_logic;
        tidlucka: out std_logic_vector(3 downto 0));
end entity Tx8Ch4_1;

architecture struktur of Tx8Ch4_1 is

--komponentdeklarationer
component Cntgnb_rec
generic(n: positive := 4;
        max_count: integer := 15);
  port(resetn, enable, clock: in std_logic;
        carry_out: out std_logic;
        q: out std_logic_vector((n-1) downto 0));
end component;

component MUX4_1
port(data_in: in std_logic_vector(3 downto 0);
      a:in std_logic_vector(1 downto 0);
      data_ut: out std_logic);
end component;

component Avkodl_4
port(adress:in std_logic_vector(1 downto 0);
      data_ut: out std_logic_vector(3 downto 0));
end component;

--deklaration av interna signaler
signal icarry: std_logic;
signal iq: std_logic_vector(1 downto 0);

--beskrivning av komponentförbindningarna
begin
  Cnt3b_rec: Cntgnb_rec
    generic map(n => 3, max_count => 7)
    port map(clock => clock, resetn => resetn,
              enable => enable, carry_out => icarry,
              q => open);

```

```
Cnt2b_re: Cntgnb_rec
  generic map(n => 2, max_count => 3)
  port map(clock => clock, resetn => resetn,
            enable => icarry, q => iq, carry_out => open);

MUX4_1: MUX4_1
  port map(a => iq, data_in => d_in, data_ut => d_ut);

Avkod1_4:
  port map(adress => iq, data_ut => tidlucka);

end architecture struktur;
```

Med **generic map()** har vi ovan tilldelat parametrarna *n* och *maxcount* värden i instansieringarna för räknarna Cnt3b\_rec och Cnt2b\_re. Vidare har för utsignalerna *q* i räknaren Cnt3b\_rec och *carry\_out* i räknaren Cnt2b\_re, som ej skall anslutas, angetts *open*. De interna signalerna har vi gett namn som börjar med bokstaven *i* (intern), vilket kan öka läsbarheten.

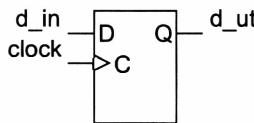
□

Ett alternativ till att göra strukturbeskrivningar i VHDL är att använda någon grafisk hierarkieditor. Syntesverktygen genererar symboler till VHDL-beskrivningarna av komponenterna som sedan kan användas vid uppritning av ett blockschema för en större krets. Blockschemat kan kompileras på samma sätt som kompileringen av strukturbeskrivningen i VHDL. Det är relativt bekvämt men nackdelen är att den grafiska beskrivningen blir knuten till ett syntesverktyg. Det kan vara en fördel att ha beskrivningarna genomgående i VHDL och därmed beskrivna i ett standardiserat språk.

## 9.7 Beskrivning av en D-vippa och en D-latch

Vi avslutar nu avsnittet om VHDL med beskrivning av en D-vippa och en D-latch. Beskrivningen av D-vippan blir ingen överraskning, ändemot kan det vara intressant att notera hur man beskriver en D-latch.

### Beskrivning av en D-vippa



Figur 9.19 D-vippa, positivt flanktriggad.

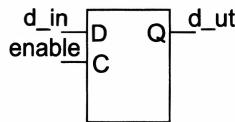
```
--D-vippa
--2001-07-30 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity D_vippa is
    port(d_in, clock: in std_logic;
          d_ut: out std_logic);
end entity D_vippa;

architecture beteende of D_vippa is
begin
    process(clock)
    begin
        if rising_edge(clock) then
            d_ut <= d_in;
        end if;
    end process;
end architecture beteende;
```

## Beskrivning av en D-latch



Figur 9.20 D-latch.

```
--D-latch
--2001-07-30 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity D_latch is
    port(d_in, enable: in std_logic;
          d_ut: out std_logic);
end entity D_latch;

architecture beteende of D_latch is
begin
    process(enable, d_in)
    begin
        if enable = '1' then
            d_ut <= d_in;
        end if;
    end process;
end architecture beteende;
```

En latch beskrivs alltså i en kombinatorisk process där samtliga insignaler är med i processens sensitivity list. Syntesverktygen kompilerar beskrivningen till en latch, eftersom inget sägs om vad utsignalen skall ha för värde när enable = '0', och då anser syntesverktyget att det gamla värdet skall ligga kvar vilket kräver en latch.

## 9.8 Övningsuppgifter

- 9.1** En kombinationskrets *Komb91* med sex insignalér  $x_0, x_1, \dots x_5$  och tre utsignalér  $u_0, u_1$  och  $u_2$  skall realiseras. För utsignalerna skall gälla:

$u_0 = 1$  om och endast om ”antingen både  $x_0$  och  $x_2$  är 0 eller  $x_4$  och  $x_5$  är olika”

$u_1 = 1$  om och endast om ” $x_0$  och  $x_1$  är lika och  $x_5$  är inversen av  $x_2$ ”

$u_2 = 0$  om och endast om ” $x_0$  är 1 och någon av  $x_1, x_2, \dots x_5$  är 0”

Beskriv *Komb91* i VHDL med parallella satser endast med operationerna *not*, *and*, *or* och *xor*.

- 9.2** Till en kombinationskrets *Komb92* inkommer två positiva binära heltal  $x = (x_3, x_2, x_1, x_0)$  och  $y = (y_3, y_2, y_1, y_0)$  i talområdet 0–15 på vardera fyra ingångar benämnda enligt bitarna i siffrorna. Kombinationskretsen har tre utgångar EQU, xGE4 och yLE4 för vilka gäller:

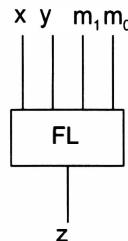
EQU = 1 om och endast om  $x = y$

xGE4 = 1 om och endast om  $x \geq 4$

yLE4 = 1 om och endast om  $y \leq 4$

Beskriv *Komb92* i VHDL med parallella satser endast med operationerna *not*, *and*, *or* och *xor*. Deklarera  $x$  och  $y$  som vektorer.

- 9.3** Beskriv i VHDL en *FL* (*Full Logic*), enligt figur 4.54, i en process med en case-sats. Deklarera  $m$  som en vektor.



- 9.4** Beskriv i VHDL en DMUX1\_4, enligt figur 2.20, i en process med en case-sats. Deklarera  $\text{data\_ut}$  som en vektor.

- 9.5** Beskriv i VHDL kombinationskretsen *Maxnumber* i övningsuppgift 4.9 i en process med en if-sats. Deklarera x och u som vektorer.
- 9.6** Beskriv i VHDL kombinationskretsen *Level* i övningsuppgift 4.8 i en process med en if-sats. Deklarera x och u som vektorer.
- 9.7** Beskriv i VHDL kombinationskretsen *Square\_x\_BCD* i övningsuppgift 4.7. Deklarera de decimala siffrorna x och msd och lsd som vektorer.
- 9.8** Beskriv i VHDL kombinationskretsen *Lower\_xyz* i övningsuppgift 4.15.
- 9.9** Beskriv i VHDL sekvenskretsarna i övningsuppgifterna 5.1 a) – g).
- 9.10** Beskriv i VHDL sekvensdetektorn *Sdet10* i övningsuppgift 5.2.
- 9.11** Beskriv i VHDL sekvensdetektorn *Sdet010or0110* i övningsuppgift 5.3.
- 9.12** Beskriv i VHDL sekvensdetektorn *Sdet1110* i övningsuppgift 5.4.
- 9.13** Beskriv i VHDL sekvensgeneratorn *Sgen07* i övningsuppgift 5.5.
- 9.14** Beskriv i VHDL sekvensgeneratorn *Sgen013764* i övningsuppgift 5.6.
- 9.15** Beskriv i VHDL sekvensgeneratorn *Sgen064325* i övningsuppgift 5.7.
- 9.16** Beskriv i VHDL den seriella adderaren *Adder\_serial* i övningsuppgifterna 5.8 a) och b).
- 9.17** Beskriv i VHDL sekvenskretsen *Pos\_edge* i övningsuppgifterna 5.9 a) och b).
- 9.18** Beskriv i VHDL räknaren *Cnt3G* i övningsuppgift 5.10.

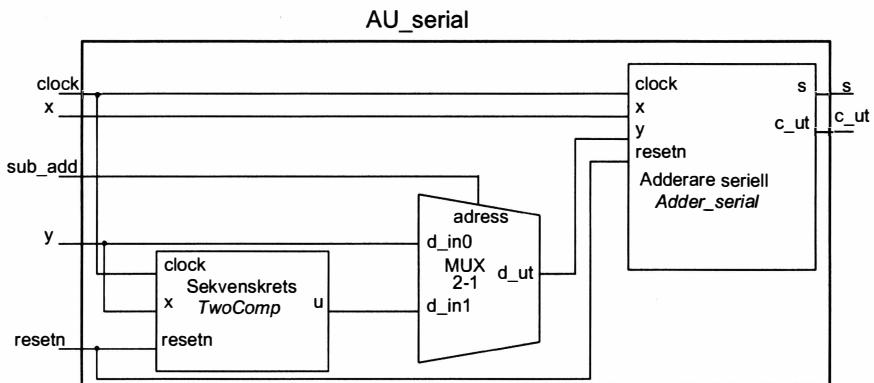
**9.19** Beskriv i VHDL räknaren *Cnt2Gm* i övningsuppgift 5.11.

**9.20** Beskriv i VHDL räknaren *Cnt4bd* i övningsuppgift 5.12.

**9.21** Beskriv i VHDL räknaren *Cnt3b\_plus3* i övningsuppgift 5.12.

**9.22** Beskriv i VHDL strukturen för mottagaren Rx8Ch1\_4 i övningsuppgift 5.15.

**9.23** Kretsen *AU\_serial* nedan är en aritmetisk enhet, som utför addition och subtraktion i serieform. Analysera funktionen. Beskriv i VHDL strukturen för kretsen.



# 10 D/A- och A/D-omvandlare

I detta avslutningskapitel skall vi studera Digital/Analog-omvandlare och Analog/Digital-omvandlare, kretsar i gränssnittet mellan den digitala och den analoga världen som förmedlar informationsutbyte mellan de två världarna. Den digitala världen växer sig ständigt större och tar över alltmer av informationsöverföring/behandling från den analoga världen. Exempelvis inom audioområdet har stora sådana förändringar skett de senaste åren, CD-spelare, digitalbandspelare, synthesizers, keyboards, digital stereo, digital radio, digital TV.

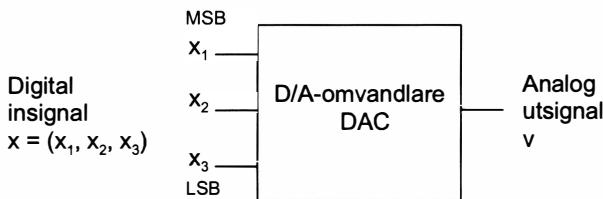
D/A-omvandlare och A/D-omvandlare är en mycket viktig klass av kretsar. Givare från fysikaliska förlopp, såsom temperatur, tryck, ljud, ljus osv. ger analoga utsignaler som måste omvandlas i A/D-omvandlare för att kunna behandlas i datorer. Inom medicinen finns många användningar av A/D-omvandlare, t.ex. bildbehandling vid magnetröntgen, analys av EKG-signaler och andra signaler. I inledningskapitlet nämndes användning av A/D- och D/A-omvandlare i digitala telefonsystem för omvandling av ljudsignaler.

D/A- och A/D-omvandlare förekommer i egna kapslar, men en kanske ännu viktigare förekomst är i mikrostyrkretsar, signalprocessorer och ASIC.

I D/A-omvandlare och A/D-omvandlare konfronteras man med problem, såsom linjäritet, noggrannhet m.m., som man inte behöver tänka på när det gäller rent digitala kretsar. Vi kommer här bara att studera fundamentala uppbyggnads- och funktionsprinciper och inte beröra problem som hör till analogtekniken.

## 10.1 D/A-omvandlare

En *Digital/Analog-omvandlare*, *D/A-omvandlare* (eng. *D/A Converter*, *DAC*) omvandlar, som namnet säger, ett digitalt ingångsvärde till ett analogt utgångsvärde. Den digitala insignalen kan tillhöra ett rent positivt talområde eller ett talområde med både positiva och negativa tal. Den analoga utsignalen kan vara en spänning eller en ström proportionell mot det digitala värdet.



Figur 10.1 Blockschema för en 3-bitars D/A-omvandlare.

En D/A-omvandlare består normalt av

- referensspänningskälla
- nät av precisionsresistorer
- transistorswitchar

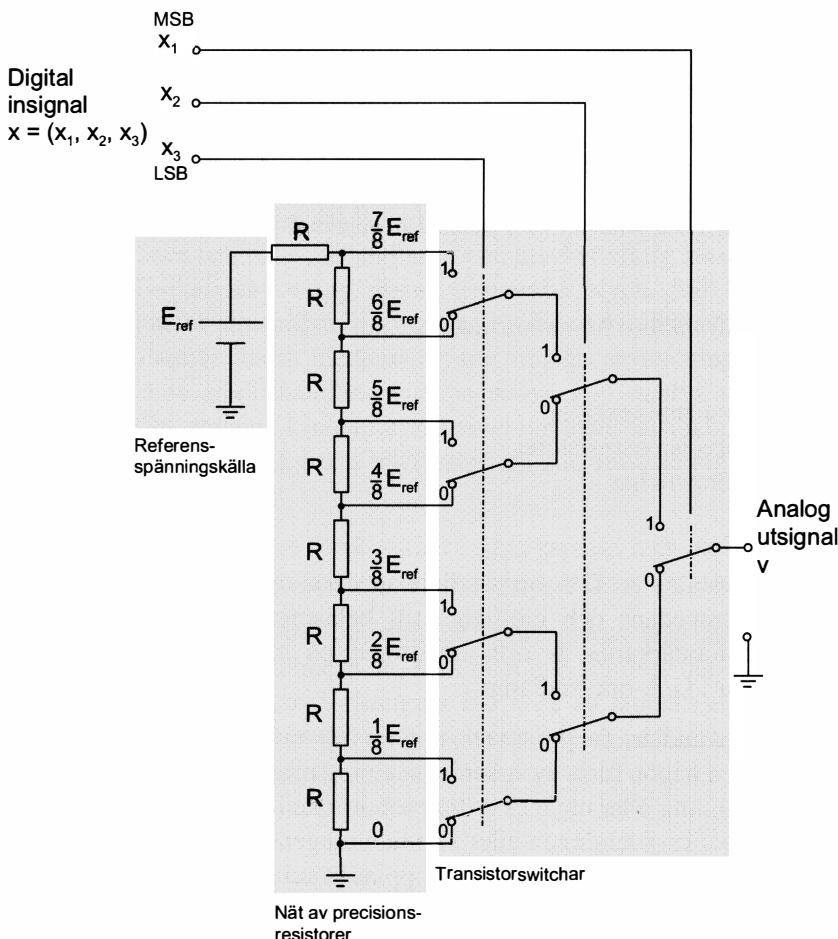
Normalt avslutas en D/A-omvandlare med en operationsförstärkare, som ger låg utimpedans och möjlighet till belastning av D/A-omvandlaren. Operationsförstärkaren är ofta integrerad på chippen tillsammans med övriga delar i D/A-omvandlaren.

En D/A-omvandlare fungerar i stort så, att från referensspänningskällan genereras, med någon form av spänningsdelning eller strömdelning, en analog utgångsspänning eller utgångsström proportionell mot det digitala ingångsvärdet. Spänningsdelningen eller strömdelningen görs med nätet av precisionsresistorer, i vilket resistorerna kopplas in och ur till referensspänningskällan med transistorswitcharna som styrs av den digitala insignalen.

Nätet av precisionsresistorer i D/A-omvandlaren förekommer i olika utseende. Naturligast vore väl i form av en vanlig spänningsdelare. Låt oss börja med att studera en sådan D/A-omvandlare.

## D/A-omvandlare med spänningsdelare

En 3-bitars D/A-omvandlare med spänningsdelare visas nedan.



Figur 10.2 D/A-omvandlare med spänningsdelare.

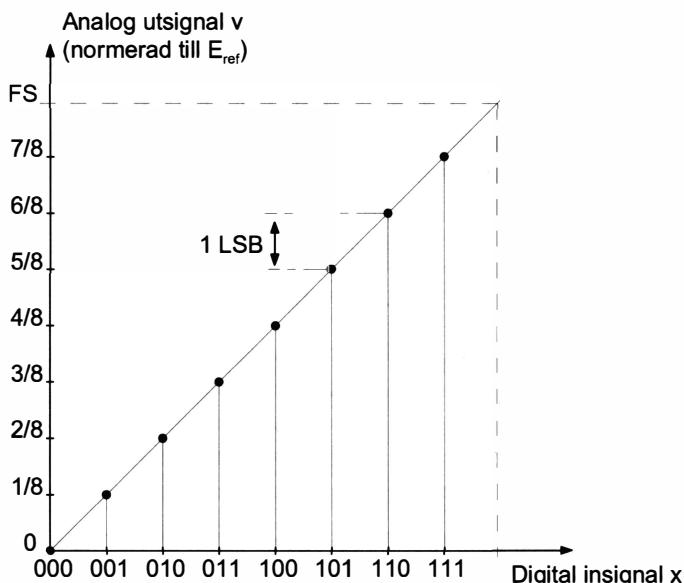
Den digitala insignalen  $x = (x_1, x_2, x_3)$  antas styra transistorswitcharna (symboliskt ritade som kontaktswitchar) så att bit  $x_i = 0$  medför switchen i läge 0 och bit  $x_i = 1$  medför switchen i läge 1. För  $x = 000$  är samtliga switchar i nedre läget och den analoga utsignalen blir  $v = 0$ . För  $x = 111$  är samtliga switchar i övre läget och utsignalen blir  $v = 7/8 \cdot E_{ref}$ , som sålunda är maximal analog utsignal. Den analoga utsignalen  $v$  som funktion av den digitala insignalen  $x = (x_1, x_2, x_3)$  och  $E_{ref}$  kan skrivas som

$$v = (x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + x_3 \cdot 2^{-3}) \cdot E_{ref} \quad (10.1)$$

Parentesen i uttrycket 10.1 ovan antar värdena 0, 1/8, 2/8, ..., 7/8 för  $x = 000, 001, 010, \dots, 111$ . Det är därför naturligt att betrakta bitarna  $x_1, x_2$ , och  $x_3$  som bitar till höger om binärpunkten i ett binärtal

$$x = 0.x_1x_2x_3 \quad 0 \leq x \leq 7/8$$

Den ideala karakteristiken för en 3-bitars D/A-omvandlare, den analoga utsignalen  $v$  som funktion av den digitala insignalen  $x$ , med  $v$  normerad till  $E_{ref}$ , visas nedan.



Figur 10.3 Ideal karakteristik för en 3-bitars D/A-omvandlare.

I karakteristiken ovan har markerats ett värde *FS* (eng. *Full Scale*), ett *nominellt fullt utslag* (= 1, normerat till  $E_{ref}$ ), som aldrig inträffar, men som ändå kan vara lämpligt att ha med, då de andra värdena är relaterade till FS. I detta fall är *verkligt fullt utslag* lika med  $7/8 \cdot FS$ .

Den analoga utsignalen  $v$  hos 3-bitars D/A-omvandlaren ovan har  $2^3 = 8$  nivåer (eng. *level*). Med *upplösningen* (eng. *resolution*) hos en D/A-omvandlare menar man vikten hos den minst signifika biten (LSB), och i detta fall är alltså upplösningen  $2^{-3}$ . Ibland anges upplösningen som antalet bitar.

Antalet bitar och snabbheten är viktiga parametrar för en D/A-omvandlare. Snabbheten anges som *inställningstiden* (eng. *settling time*) och definieras som tiden från en ändring av den digitala insignalen tills utsignalen antagit sitt slutvärde, normalt inom  $\pm 1/2$  LSB. Inställningstiden bestäms bl.a. av transistorswitcharna och operationsförstärkaren.

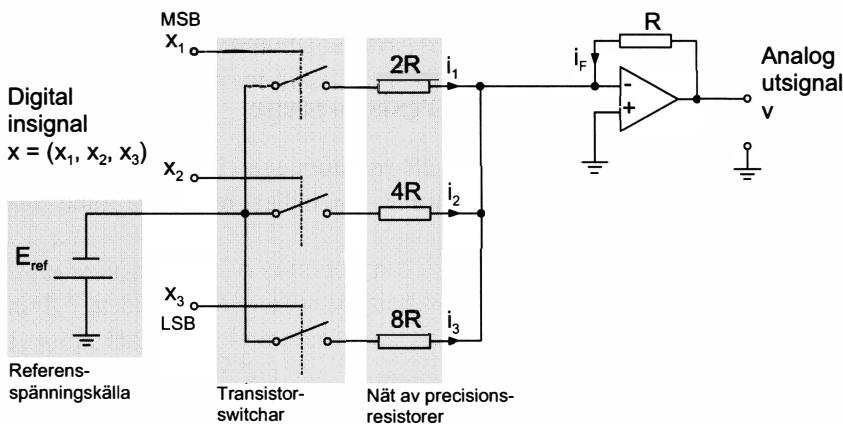
D/A-omvandlaren med spänningsdelare är enkel till sin uppbyggnad och funktion. En annan fördel är att alla resistorerna har samma värde. Ytterligare fördel är att den är *monoton* även om resistorerna avviker från det nominella värdet. En D/A-omvandlare definieras vara *monoton* om den analoga utsignalen aldrig minskar när den digitala insignalen ökar. Alla fördelar till trots så är D/A-omvandlaren med spänningsdelare svår att realisera i praktiken. Orsaken är att det krävs  $2^n$  resistorer och  $2^n - 1$  transistorswitchar för en  $n$ -bitars D/A-omvandlare. I 3-bitars D/A-omvandlaren ovan är det inget större problem med 8 resistorer och 7 transistorswitchar, men i en 16-bitars D/A-omvandlare blir antalet 65 536 respektive 65 535. D/A-omvandlaren med spänningsdelare används därför bara vid ett mindre antal bitar.

## D/A-omvandlare med viktade resistorer

En 3-bitars D/A-omvandlare med viktade resistorer visas i figuren nedan. Förutom de fundamentala delarna, referensspänningsskällan  $E_{ref}$ , transistorswitcharna och resistornätet med de viktade resistorerna  $2R$ ,  $4R$  och  $8R$ , ingår också en operationsförstärkare. För transistorswitcharna antas att bit  $x_i = 0$  medförs öppen switch och bit  $x_i = 1$  medförs sluten switch.

Operationsförstärkaren med återkopplingsresistorn  $R$  och de viktade resistorerna är i s.k. *inverterande koppling*, som förutsätts känd. Strömmotkopplingen medförs att potentialerna  $v_+$  och  $v_-$  på operationsförstärkarens plus- och minusingångar blir approximativt lika stora. Eftersom här plusingången

ligger på nollpotential så får också minusgången approximativt nollpotential, s.k. "virtuell jord".



Figur 10.4 D/A-omvandlare med viktade resistorer.

Summering av strömmarna i den virtuella jordpunkten ger

$$i_1 + i_2 + i_3 + i_F = 0$$

$$\frac{-x_1 \cdot E_{ref} - 0}{2R} + \frac{-x_2 \cdot E_{ref} - 0}{4R} + \frac{-x_3 \cdot E_{ref} - 0}{8R} + \frac{v - 0}{R} = 0$$

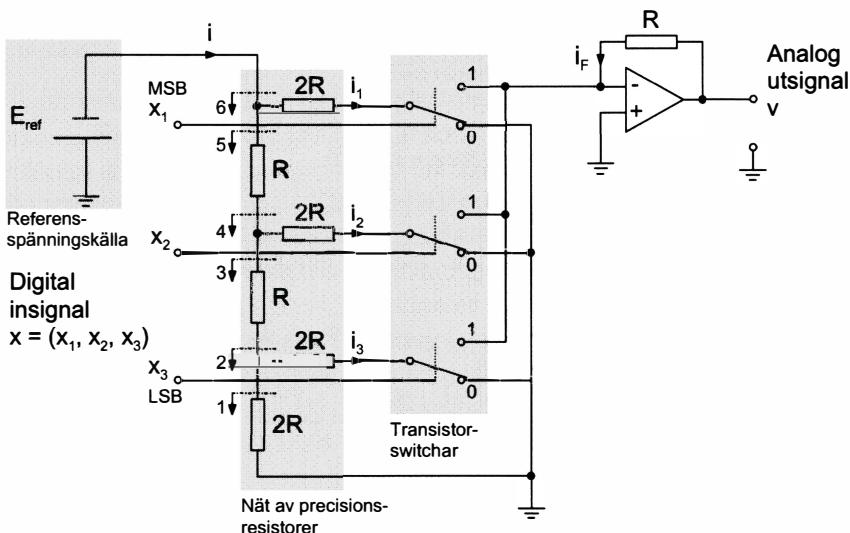
$$v = (x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + x_3 \cdot 2^{-3}) \cdot E_{ref} \quad (10.2)$$

D/A-omvandlaren ovan är uppbyggd med *viktade resistorer*, viktade med 2-potenser, vikterna i ett vanligt binärtal. Nätet av precisionsresistorer innehåller  $n$  resistorer jämfört med  $2^n$  för den förra D/A-omvandlaren med spänningsdelare, en klar fördel. En nackdel är att resistorerna inte har samma värde, t.ex. har en 16-bitars D/A-omvandlare med viktade resistorer ett nät med precisionsresistorer inom ett mycket stort resistansområde,  $2 \cdot R, 4 \cdot R, \dots, 2^{16} \cdot R$ . Resistorer inom ett så stort område är opraktiskt ur tillverkningssynpunkt, varje resistor lasertrimmas på chippet. Vidare får då inte resistorerna samma temperaturkoefficient, vilket omöjliggör tempera-

turkompensering. D/A-omvandlare konstrueras därför normalt inte med viktade resistorer enligt ovan, utan med ett precisionsresistornät i form av en s.k. *R-2R-resistorstege* (eng. *resistor ladder*), innehållande resistorer med bara två olika resistansvärden,  $R$  och  $2R$ .

## D/A-omvandlare med R-2R-resistorstege

En 3-bitars D/A-omvandlare med R-2R-resistorstege visas i figuren nedan, där alltså nätet av precisionsresistorer innehåller resistorer med bara värdena  $R$  och  $2R$ .



Figur 10.5 D/A-omvandlare med R-2R-resistorstege

Låt oss analysera funktionen, som inte är lika enkel som funktionen hos de förra D/A-omvandlarna. – Den digitala insignalen  $x = (x_1, x_2, x_3)$  antas styra transistorswitcharna så att bit  $x_i = 0$  medföljer switchen i läge 0 och bit  $x_i = 1$  medföljer switchen i läge 1. Transistorswitcharna i resistorstegen kopplar strömmarna  $i_1, i_2$  och  $i_3$  antingen till den ”riktiga” jordpunkten (switchen i läge 0) eller till den virtuella jordpunkten (switchen i läge 1), varför strömmarna är konstanta och oberoende av switcharnas lägen. Strömmen  $i$  från

referensspänningsskällan  $E_{ref}$  är sålunda också konstant och lika med  $E_{ref}$  dividerad med inresistansen till resistorstegen. Låt oss beräkna inresistansen, som också är konstant och oberoende av switcharnas lägen. I resistorstegen är markerat nivåer 1 till 6. Impedansen från nivå 1 till jord är  $2R$ . Impedansen från nivå 2 till jord är  $2R//2R = R$ , impedansen från nivå 3 till jord är  $R+R = 2R$ , impedansen från nivå 4 till jord är  $2R//2R = R$  osv. Inimpedansen till resistorstegen, från nivå 6 till jord, är alltså  $R$ . För strömmen i gäller då att

$$i = \frac{-E_{ref}}{R}$$

Inresistansen i nivå 6 åt höger mot switchen är  $2R$  och inresistansen neråt i nivå 5 är också  $2R$ , varför strömmen  $i$  kommer att halveras i nivå 6 så att  $i_1 = i/2$ . Halvering kommer sedan att ske i nivå 4 och 2, varför  $i_2 = i/4$  och  $i_3 = i/8$ .

Vi kan nu summa strömmarna i den virtuella jordpunkten på samma sätt som för den förra D/A-omvandlaren. Strömmen efter switchen in i den virtuella jordpunkten kan vi symboliskt teckna  $x \cdot i$ , där  $x = 0$  innebär switchen i det i figur 10.5 angivna läget och således strömmen in i den virtuella jordpunkten lika med  $0 \cdot i = 0$ , och där  $x = 1$  innebär switchen i det övre läget och strömmen in i den virtuella jordpunkten lika med  $1 \cdot i = i$ .

Summering av strömmarna i den virtuella jordpunkten ger

$$x_1 \cdot i_1 + x_2 \cdot i_2 + x_3 \cdot i_3 + i_F = 0$$

$$x_1 \cdot \frac{i}{2} + x_2 \cdot \frac{i}{4} + x_3 \cdot \frac{i}{8} + i_F = 0$$

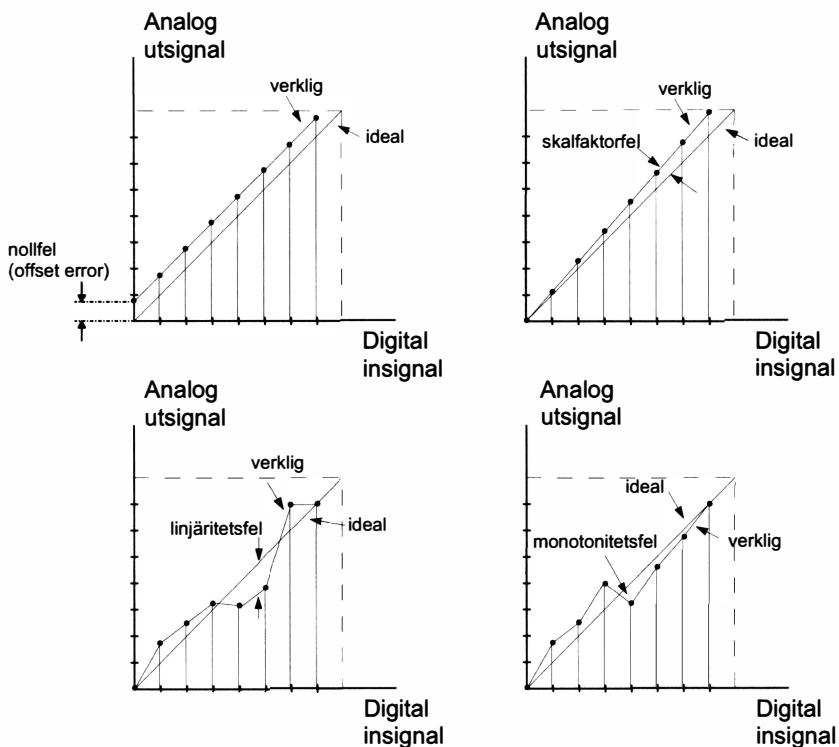
$$x_1 \cdot \frac{-E_{ref}}{2R} + x_2 \cdot \frac{-E_{ref}}{4R} + x_3 \cdot \frac{-E_{ref}}{8R} + \frac{v - 0}{R} = 0$$

$$v = (x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + x_3 \cdot 2^{-3}) \cdot E_{ref} \quad (10.3)$$

Vi ser alltså att uttryck (10.3) är samma som uttrycken (10.1) och (10.2) för de föregående D/A-omvandlarna.

## Fel i överföringskarakteristiken hos D/A-omvandlare

Noggrannheten hos en D/A-omvandlare påverkas av många olika faktorer. Totala felet vid en D/A-omvandling är sammansatt av flera olika typer av fel. I figuren nedan definieras några olika feltyper.



Figur 10.6 Olika typer av fel hos en D/A-omvandlare.

### Nollfel

Nollfelet (eng. *offset error*) är en parallellförflyttning av den verkliga karakteristiken i förhållande till den ideala karakteristiken. Felet kan normalt trimmas bort med en yttre potentiometer med vilken en ström, positiv eller negativ, tillförs den virtuella jordpunkten.

## Skalfaktorfel

Skalfaktorfelet, förstärkningsfelet (eng. *gain error*) är avvikelsen i lutning hos den verkliga karakteristiken i förhållande till den idealna karakteristiken. Karakteristikens lutning bestäms normalt av förstärkningen hos operationsförstärkaren och trimmas med en yttre resistor parallellt med eller i serie med återkopplingsresistorn.

## Linjäritetsfel

Linjäritetsfelet, icke-linjäriteten (eng. *non linearity*), är den maximala avvikelsen hos den aktuella karakteristiken från den idealna karakteristiken. Linjäritetsfelet anges normalt i procent av FS. Linjäritetsfelet beror på avvikelse i resistans från idealna värden hos resistorerna i nätet av precisionsresistorer på chippen och kan inte trimmas bort.

## Differentiellt linjäritetsfel

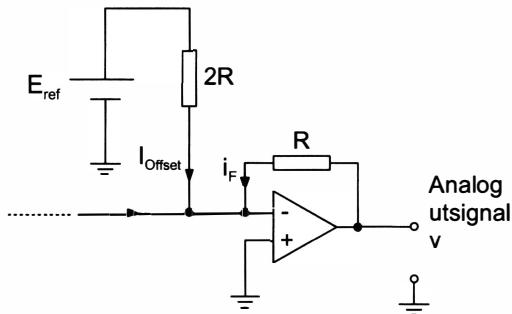
Ett *steg* i den idealna karakteristiken för en n-bitars D/A-omvandlare är FS/ $2^n$ . Det differentiella linjäritetsfelet definieras som den maximala skillnaden mellan ett steg i den aktuella karakteristiken och ett steg i den idealna karakteristiken.

## Monotonitetsfel

En D/A-omvandlare säges vara *monoton* om utsignalen aldrig minskar när insignalen ökar. Om det differentiella linjäritetsfelet har ett tillräckligt stort negativt värde blir D/A-omvandlaren *icke-monoton*. I figuren ovan erhålls en minskande analog utsignal för den digitala insignalen 100, som har en mindre analogt värde än den digitala insignalen 011. Monotonitetsfelet beror sannolikt på ett felaktigt värde hos resistorn i den mest signifikanta positionen.

## Unipolär och bipolär utsignal

De hittills visade 3-bitars D/A-omvandlarna är *unipolära* (eng *unipolar*), innehållande att de ger en analog utsignal av bara en polaritet. Vi har också förutsatt en digital insignal  $x$ , som tillhör ett positivt talområde  $0 \leq x \leq 7/8$ . En *bipolär* (eng. *bipolar*) D/A-omvandlare som ger både positiva och negativa utsignaler kan konstrueras av de föregående omvandlarna genom tillförande av en ström  $I_{\text{Offset}}$  i den virtuella jordpunkten som parallellförskjuter hela karakteristiken nedåt. I figuren nedan visas principen genom anslutning av referensspänningskällan  $E_{\text{ref}}$  till den virtuella jordpunkten via en resistor  $2R$ .



Figur 10.7 Parallelfförskjutning av karakteristiken för bipolär utsignal.

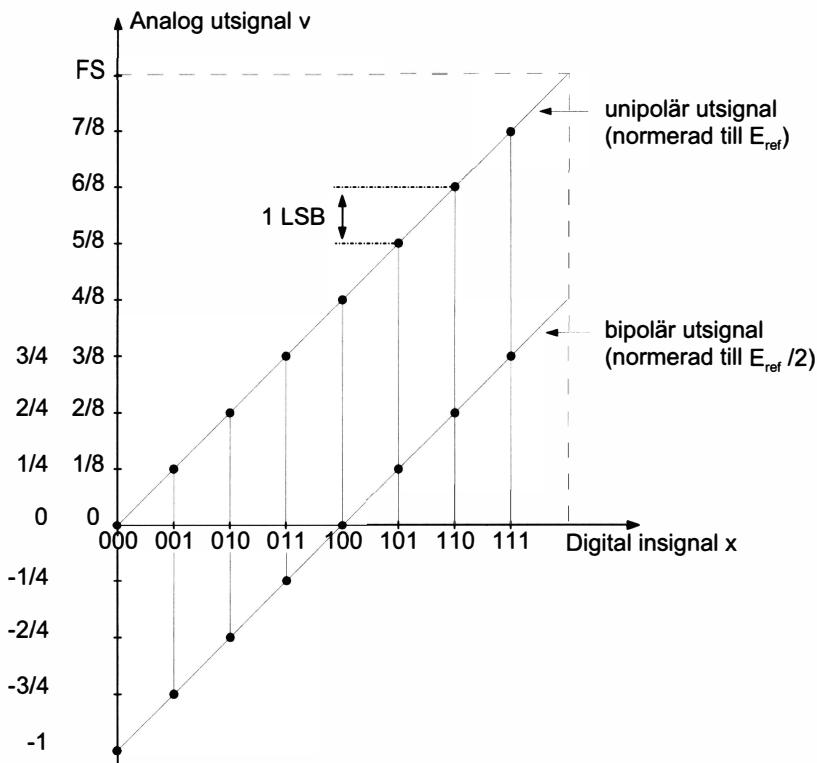
Strömmen  $I_{\text{Offset}}$  till den virtuella jordpunkten blir  $E_{\text{ref}}/2R$ , vilket förändrar uttryckena (10.2) och (10.3) för den analoga utsignalen  $v$  till

$$v = (x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + x_3 \cdot 2^{-3}) \cdot E_{\text{ref}} - \frac{E_{\text{ref}}}{2} \quad (10.4)$$

Den analoga utsignalen  $v$  ligger i området  $(-1) \cdot (E_{\text{ref}}/2) \leq v \leq (3/4) \cdot (E_{\text{ref}}/2)$  för den digitala insignalen  $000 \leq x \leq 111$ . I tabellen och figuren nedan visas  $v$ , normerad till  $E_{\text{ref}}/2$ , för de olika insignalkombinationerna.

Tabell 10.1: Utsignalen  $v$  för 3-bitars bipolära D/A-omvandlaren i figur 10.7.

Digital insignal x $x_1x_2x_3$	Analog utsignal v (normerad till $E_{ref}/2$ )
1 1 1	3/4
1 1 0	2/4
1 0 1	1/4
1 0 0	0
0 1 1	-1/4
0 1 0	-2/4
0 0 1	-3/4
0 0 0	-1

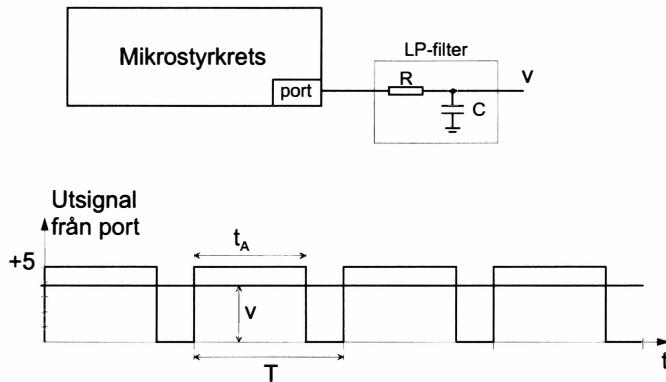


Figur 10.8 Karakteristik för unipolära och bipolära 3-bitars D/A-omvandlaren.

I karakteristiken ser vi att  $x_1$  kan betraktas som *teckenbit*, med  $x = 1$  för positiva tal och  $x_1 = 0$  för negativa tal. Denna representation kallas *binär offset* (eng. *binary offset*), en naturlig benämning med tanke på hur den skaps genom en offset, parallellförskjutning av karakteristiken. I kapitel 4, avsnittet representation av negativa tal med 2-komplement, nämndes denna representation i förbigående. Omvandling mellan binär offset och 2-komplement är enkel, den innebär endast invertering av teckenbiten.

## D/A-omvandling genom pulsbreddsmodulering (PWM)

I mikrostyrkretsar ingår normalt en A/D-omvandlare, men ofta inte någon D/A-omvandlare. D/A-omvandling, dvs. omvandling av ett digitalt värde i mikrostyrkretsen till ett analogt värde på en portutgång, lösas genom *pulsbreddsmodulering* (eng. *pulse width modulation*, PWM), enligt principen i figuren nedan.



Figur 10.9 Pulsbreddsmodulering, PWM.

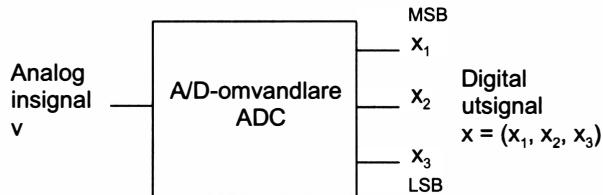
På mikrostyrkretsens portutgång kan utmatas en binär signal med nivåerna 0V och +5V. Antag att ett digitalt värde  $x$  i talområdet 0–255 skall matas ut och ge en analog signal  $v$  i området 0 till +5V proportionell mot det digitala värdet. Utmatningen görs då med PWM så att pulsens bredd  $t_A$  är proportionell mot det digitala värdet ( $T = \text{konstant}$ ).

$$t_A = \frac{x}{255} \cdot T \quad v = \frac{t_A}{T} \cdot 5 = \frac{x}{255} \cdot 5 \quad [\text{V}]$$

Den analoga signalen  $v$  (likspänningen) efter lågpassfiltret blir proportionell mot det digitala värdet. T.ex. ger  $x = 255$ ,  $v = 5\text{V}$  och  $x = 51$ ,  $v = 1\text{V}$ .

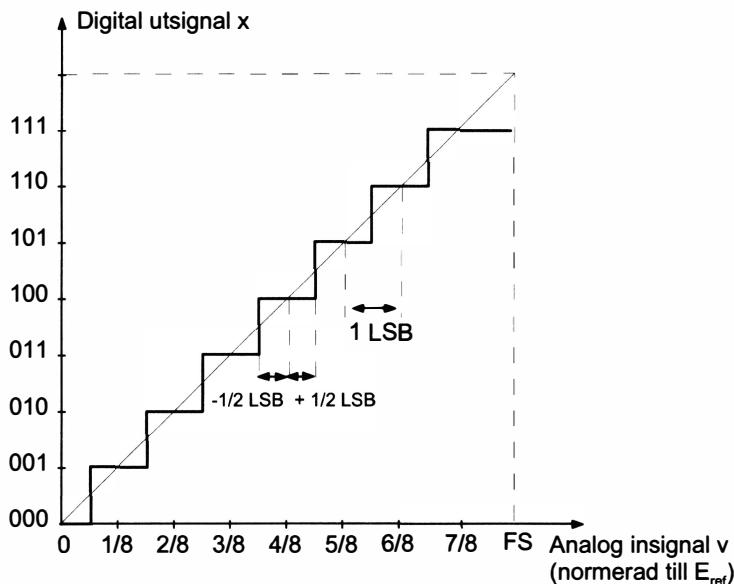
## 10.2 A/D-omvandlare

En *Analog/Digital-omvandlare*, *A/D-omvandlare* (eng. *A/D Converter*, *ADC*) omvandlar ett analogt ingångsvärde till ett digitalt utgångsvärde.



Figur 10.10 Blockschema för en 3-bitars A/D-omvandlare.

Den idealade karakteristiken för en 3-bitars A/D-omvandlare visas i figuren nedan. Karakteristiken är en trappstegskurva.



Figur 10.11 Ideal karakteristik för en 3-bitars A/D-omvandlare.

Den analoga insignalen kvantiseras i ett antal lika intervall, som vart och ett tilldelas ett digitalt värde, motsvarande värdet av den analoga signalen mitt i intervallet. En A/D-omvandlare fungerar i stort så, att den analoga insignalen *jämförs* med i A/D-omvandlaren genererade analoga referensnivåer, motsvarande ändpunkterna i kvantiseringsintervallen och för vilka det digitala värdet är känt. Vid jämförelsen bestäms i vilket intervall den analoga insignalen ligger och intervallets digitala värde matas ut som digital utsignal från A/D-omvandlaren.

A/D-omvandlare är mer komplicerade än D/A-omvandlare. Det finns ett helt spektrum av olika A/D-omvandlare som avspeglar att "tid är pengar" och att man måste kompromissa mellan antal bitar och snabbhet. I de snabbaste A/D-omvandlarna, s.k. *flash-converter*, utförs omvandlingen i parallell form, genom att alla de analoga referensnivåerna finns tillgängliga för en samtidig jämförelse med den analoga insignalen. I de längsammaste A/D-omvandlarna görs omvandlingen med tidsuppdelning, genom att de olika analoga referensnivåerna genereras en i taget och jämförs med den analoga insignalen.

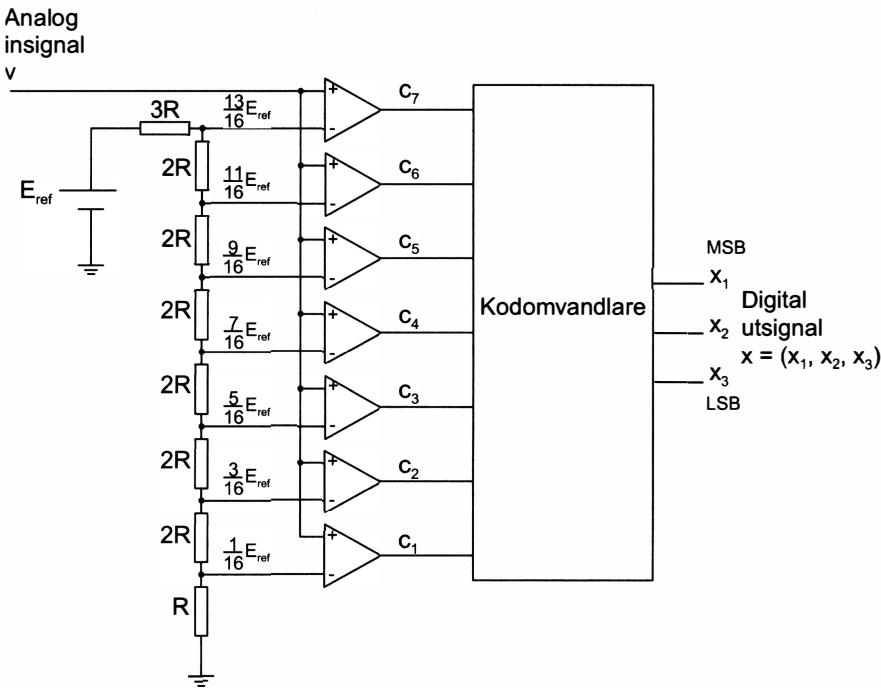
## A/D-omvandlare – typ parallel ("flash")

I figuren nedan visas en 3-bitars parallell A/D-omvandlare. De analoga referensnivåerna genereras i ett nät av precisionsresistorer (spänningssdelare). Jämförelsen av den analoga insignalen med de analoga referensnivåerna görs i analoga komparatorer, som i princip är snabba operationsförstärkare. Komparatorerna jämför två analoga insignaler  $v_+$  och  $v_-$  och ger utsignalen Hög ("1") om ( $v_+ > v_-$ ) och utsignalen Låg ("0") om ( $v_+ < v_-$ ). Komparatorn är en icke motkopplad operationsförstärkare som arbetar i öppen loop, och så fort insignalerna avviker från varandra (vilket de alltid gör) så kommer operationsförstärkaren p.g.a. den mycket höga förstärkningen att bli överstyrd åt det ena eller andra hålet, bli Hög eller Låg.

En parallell A/D-omvandlare har lika många komparatorer som kvantiseringsintervall i den analoga insignalen. För en  $n$ -bitars omvandlare  $2^n - 1$  stycken och i vår 3-bitars omvandlare alltså  $2^3 - 1 = 7$  stycken.

Komparatorernas minusgång är matad med det analoga värdet för ändpunkten av respektive kvantiseringsintervall. utsignalen från komparatorerna,  $c = (c_7, c_6, c_5, c_4, c_3, c_2, c_1)$ , är det digitala värdet för den analoga insignalen. För t.ex.  $5/16 < v < 7/16$  blir  $c = 0000111$ . Det digitala värdet erhålls

i en ”termometerkod”, där en rad ettor växer från höger i  $c$  när den analoga insignalen ökar. För  $v > 13/16$  blir  $c = 1111111$ . Termometerkoden är en oekonomisk kod och omvandlas i en kodomvandlare, en kombinationskrets, till en komprimerad binärkod.

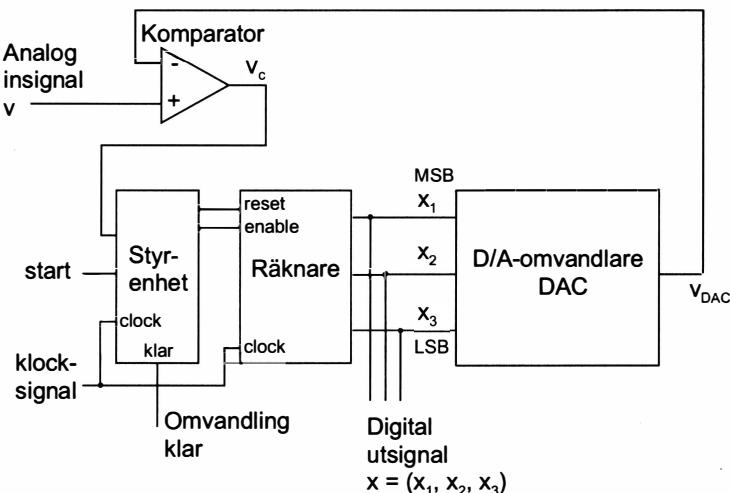


Figur 10.12 A/D-omvandlare – typ parallell (”flash”).

Problemet med parallell A/D-omvandling är antalet komparatorer. Exempelvis skulle i en 18-bitars parallell A/D-omvandlare behövas  $2^{18} - 1 = 262\,143$  komparatorer. Dagens teknik medger inte detta, idag ligger gränsen för en parallell A/D-omvandlare vid ca 10 bitars upplösning, dvs. 1023 komparatorer.

## A/D-omvandlare – typ nivåramp

A/D-omvandlare med nivåramp utför omvandlingen i extrem serieform. I stället för en komparator för varje analog referensnivå som i den föregående parallella A/D-omvandlaren, används här bara en komparator och de analoga referensnivåerna genereras en i taget och jämförs med den analoga insignalen. Ett naturligt sätt att generera referensnivåerna är i form av en ramp med användning av en D/A-omvandlare till vilken den digitala insignalen matas från en räknare enligt figuren nedan.



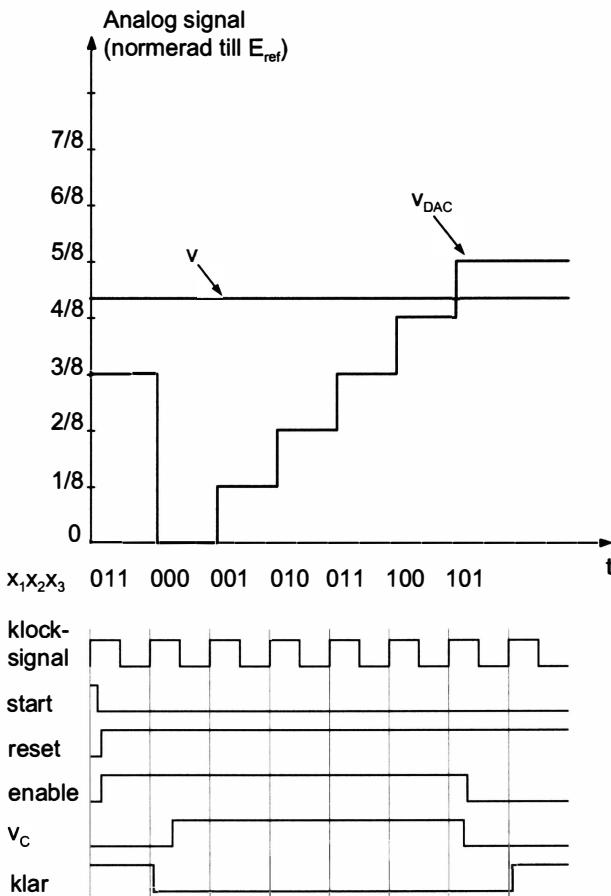
Figur 10.13 A/D-omvandlare – typ nivåramp.

En A/D-omvandling startas med signalen  $start = 0$  till styrenheten, vilken då ger *reset* och *enable* till räknaren, som börjar räkna upp från värdet 0. Vi antar synkron *reset* och att signalen *start* skall ha värdet 0 under hela omvandlingen samt att ny omvandling kräver att signalen *start* först ges värdet 1. Från D/A-omvandlaren erhålls då en ramp enligt figuren nedan. När utsignalen  $v_{DAC}$  blir större än den analoga insignalen  $v$  ger komparatorn  $v_c = 0$  till styrenheten, som då stoppar räknaren med *enable* = 0 och signalerar omvandling klar med utsignalen *klar* = 1. Det digitala värdet till den analoga insignalen kan läsas från räknaren. I figuren är  $4/8 < v < 5/8$  och A/D-omvandlingen ger som resultat  $x = 101$ .

Nackdelen med denna omvandlare är omvandlingstiden. För upplösningen  $n$  bitar blir maximala omvandlingstiden  $t_c$  lika med

$$t_c = 2^n \cdot T_{clock}$$

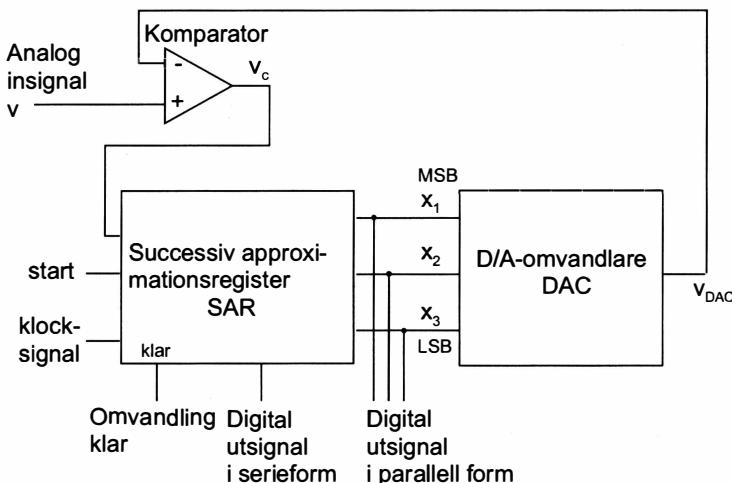
där  $T_{clock}$  är klocksignalens periodtid, som måste väljas med hänsyn till födröjningen i komparatorn + styrenheten + räknaren + D/A-omvandlaren.



Figur 10.14 Tidsdiagram för A/D-omvandling med ramp.

## A/D-omvandlare – typ successiv approximation

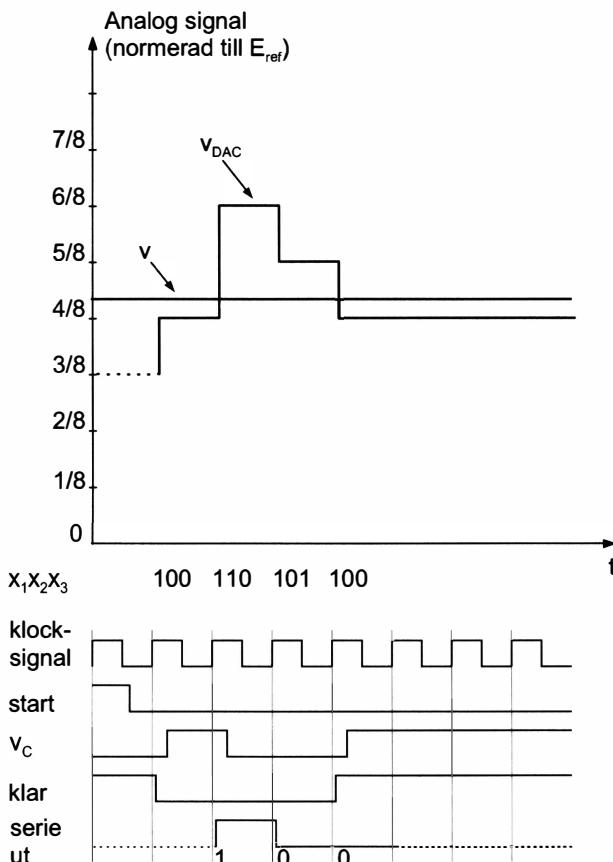
A/D-omvandlaren med successiv approximation genererar de analoga referensnivåerna mycket ”smartare” än den förra A/D-omvandlaren med nivåramp. Nivåerna genereras enligt samma princip som när man skall gissa ett tal som någon tänker på. Om talet är ett heltal mellan 1 och 100, så frågar man inte, är talet 1, är talet 2, är talet 3 osv., enligt principen nivåramp, utan man börjar med att fråga om talet är större än 50, och om svaret är ja frågar man om det är större än 75 osv. I figuren nedan visas uppbyggnaden av en A/D-omvandlare med successiv approximation.



Figur 10.15 A/D-omvandlare – typ successiv approximation.

Tidsdiagram för en omvandling visas i figuren nedan. – Den digitala insignalen till D/A-omvandlaren genereras av ett successiv-approximationsregister (SAR). Omvandlingen startas med signalen start = 0. Vi antar att start skall ha värdet 0 under hela omvandlingen samt att ny omvandling kräver först start = 1. På klockpulsens positiva flank lägger då SAR ut  $x_1x_2x_3 = 100$ . Utsignalen från D/A-omvandlaren blir då  $v_{DAC} = 4/8$  (normerad), som går in i komparatorn och jämförs med  $v$ ,  $4/8 < v < 5/8$ . Komparatoren ger  $v_c = 1$  till SAR, som information om att utsignalen från D/A-omvandlaren var för liten. SAR bibehåller därför bit  $x_1$  till  $x_1 = 1$  och lägger vid nästa positiva flank hos klocksignalen ut  $x_1x_2x_3 = 110$ . Utsig-

nalen från D/A-omvandlaren blir då  $v_{DAC} = 6/8$ , som går in i komparatorn och jämförs med  $v$ . Komparatoren ger  $v_c = 0$  till SAR, som information om att utsignalen från D/A-omvandlaren var för stor. SAR återställer därför bit  $x_2$  till  $x_2 = 0$  och lägger vid nästa positiva flank hos klocksignalen ut  $x_1x_2x_3 = 101$ . Utsignalen från D/A-omvandlaren blir då  $v_{DAC} = 5/8$ , som går in i komparatorn och jämförs med  $v$ . Komparatoren ger  $v_c = 0$  till SAR, som information om att utsignalen från D/A-omvandlaren var för stor. SAR återställer bit  $x_3$  till  $x_3 = 0$  och A/D-omvandlingen är klar med slutresultatet  $x_1x_2x_3 = 100$ . Utsignalen från komparatoren överensstämmer med bitarna i slutresultatet och kan därför efter varje jämförelse direkt användas som digital utsignal i serieform.



Figur 10.16 Tidsdiagram för A/D-omvandling med successiv approximation.

A/D-omvandlingen med successiv approximation för 3-bitars omvandlaren i exemplet ovan krävde 4 klockpulser, en för varje bit plus en för återställning av den sista biten. Omvandlingen skulle med en A/D-omvandlare med nivåramp kräva maximalt 7 klockpulser, ingen större skillnad, men om det i stället vore en 18-bitars omvandlare så skulle med successiv approximation åtgå  $18 + 1 = 19$  klockpulser, medan med nivåramp maximalt 262 143 klockpulser!

# Appendix 1 2-potenser

1	$2^0$	
2	$2^1$	
4	$2^2$	
8	$2^3$	
16	$2^4$	
32	$2^5$	
64	$2^6$	
128	$2^7$	
256	$2^8$	1/4k
512	$2^9$	1/2k
<b>1024</b>	<b><math>2^{10}</math></b>	<b>1k</b>
2048	$2^{11}$	2k
4096	$2^{12}$	4k
8192	$2^{13}$	8k
16384	$2^{14}$	16k
32768	$2^{15}$	32k
65536	$2^{16}$	64k
131 072	$2^{17}$	128k
262 144	$2^{18}$	256k
524 288	$2^{19}$	512k
<b>1048 576</b>	<b><math>2^{20}</math></b>	<b>1 M</b>
2097 152	$2^{21}$	2M
4 194 304	$2^{22}$	4M
8 388 608	$2^{23}$	8M
16 777 216	$2^{24}$	16M
33 554 432	$2^{25}$	32M
67 108 864	$2^{26}$	64M
134 217 728	$2^{27}$	128M
268 435 456	$2^{28}$	256M
536 870 912	$2^{29}$	512M
<b>1073 741 824</b>	<b><math>2^{30}</math></b>	<b>1G</b>

# Appendix 2 ASCII-koden

Bitarna 4 till 6 Hex (MSD)								
Bitarna 0 till 3 Hex (LSD)	0	1	2	3	4	5	6	7
0 NUL	DLE	SP	0	@	P		p	
1 SOH	DC1	!	1	A	Q	a	q	
2 STX	DC2	"	2	B	R	b	r	
3 ETX	DC3	#	3	C	S	c	s	
4 EOT	DC4	\$	4	D	T	d	t	
5 ENQ	NAK	%	5	E	U	e	u	
6 ACK	SYN	&	6	F	V	f	v	
7 BEL	ETB	,	7	G	W	g	w	
8 BS	CAN	(	8	H	X	h	x	
9 HT	EM	)	9	I	Y	i	y	
A LF	SUB	*	:	J	Z	j	z	
B VT	ESC	+	;	K	[	k	{	
C FF	FS	,	<	L	\	l		
D CR	GS	-	=	M	]	m	}	
E SO	RS	.	>	N	^	n	~	
F SI	US	/	?	O	-	o	DEL	

# Appendix 3 PSpice simulerings-filer

CMOS-inverterare. nMOS:W/L=1 pMOS: W/L=2,5.

Överföringskarakteristik

\* Lars-H Hemert 1996-01-09

```
* Modeller
.Model n-kanal nMOS VTO=1V KP=20u
.Model p-kanal pMOS VTO=-1V KP=8u
* Kretsbeskrivning
M1 2 1 0 0 n-kanal W=1u L=1u
M2 2 1 3 3 p-kanal W=2.5u L=1u
* Matning
VDD 3 0 DC 5V
* DC-svep
VI 1 0
.DC VI 0V 5V 0.01V
.PROBE
.END
```

CMOS-inverterare. nMOS:W/L=1 pMOS:W/L=2 CL=0,1 pF

Transientanalys

\* Lars-H Hemert 1996-01-09

```
* Modeller
.Model n-kanal nMOS VTO=1V KP=20u
.Model p-kanal pMOS VTO=-1V KP=8u
* Kretsbeskrivning
M1 2 1 0 0 n-kanal W=1u L=1u
M2 2 1 3 3 p-kanal W=2.5u L=1u
CL 2 0 0.1pF
* Matning
VDD 3 0 DC 5V
* Transientanalys
VI 1 0 PULSE(5V 0V 0nS 1nS 1nS 10nS 20nS)
.TRAN 0.1nS 20nS
.PROBE
.END
```

## NMOS-inverterare. Överföringskarakteristik

\* Lars-H Hemert 1996-01-09

```
* Modeller
.MODEL n-kanall nMOS      VTO=1V  KP=20u  GAMMA=0.5
.MODEL n-kanal2 nMOS      VTO=-3V  KP=15u  GAMMA=0.5
* Kretsbeskrivning
M1 2 1 0 0 n-kanall      W=3u  L=1u
M2 3 2 2 0 n-kanal2      W=1u  L=1u
* Matning
VDD 3 0 DC 5V
* DC-svep
VIN 1 0
.DC VIN 0V 5V 0.01V
.PROBE
.END
```

## nMOS-inverterare. Transientanalys

\* Lars-H Hemert 1996-01-09

```
* Modeller
.MODEL n-kanall nMOS      VTO=1V  KP=20u  GAMMA=0.5
.MODEL n-kanal2 nMOS      VTO=-3V  KP=15u  GAMMA=0.5

* Kretsbeskrivning
M1 2 1 0 0 n-kanall      W=3u  L=1u
M2 3 2 2 0 n-kanal2      W=1u  L=1u
CL 2 0 0.1pF

* Matning
VDD 3 0 DC 5V

* Transientanalys
VI 1 0 PULSE (5V 0V 0nS 1nS 1nS 18nS 40nS)
.TRAN 1ns 30ns

.PROBE

.END
```

# **Appendix 4 Halvledarminnen – datablad**

- Infineon HYB25D25640/800/AT 256 Mbit DDR SDRAM
- Infineon HYS 64/72V64220GU SDRAM DIMM Module Package
- Intel 1101, 256 bit ( $256 \times 1$ ) statiskt RWM
- Intel 1103, 1 kbit ( $1k \times 1$ ) dynamisk RWM
- Intel 1702, 2 kbit ( $256 \times 8$ ) EPROM

## Features

### CAS Latency and Frequency

CAS Latency	Maximum Operating Frequency (MHz)		
	DDR266A	DDR266B	DDR200
2	-7	-7.5	-8
2.5	133	125	100
	143	133	125

- Double data rate architecture: two data transfers per clock cycle
- Bidirectional data strobe (DQS) is transmitted and received with data, to be used in capturing data at the receiver
- DQS is edge-aligned with data for reads and is center-aligned with data for writes
- Differential clock inputs (CK and  $\overline{CK}$ )
- Four internal banks for concurrent operation

- Data mask (DM) for write data
- DLL aligns DQ and DQS transitions with CK transitions.
- Commands entered on each positive CK edge; data and data mask referenced to both edges of DQS
- Burst lengths: 2, 4, or 8
- CAS Latency: 2, 2.5
- Auto Precharge option for each burst access
- Auto Refresh and Self Refresh Modes
- 7.8  $\mu$ s Maximum Average Periodic Refresh Interval
- 2.5V (SSTL\_2 compatible) I/O
- $V_{DDQ} = 2.5V \pm 0.2V$  /  $V_{DD} = 2.5V \pm 0.2V$
- TSOP66 package

## Description

The 256Mb DDR SDRAM is a high-speed CMOS, dynamic random-access memory containing 268,435,456 bits. It is internally configured as a quad-bank DRAM.

The 256Mb DDR SDRAM uses a double-data-rate architecture to achieve high-speed operation. The double data rate architecture is essentially a  $2n$  prefetch architecture with an interface designed to transfer two data words per clock cycle at the I/O pins. A single read or write access for the 256Mb DDR SDRAM effectively consists of a single  $2n$ -bit wide, one clock cycle data transfer at the internal DRAM core and two corresponding  $n$ -bit wide, one-half-clock-cycle data transfers at the I/O pins.

A bidirectional data strobe (DQS) is transmitted externally, along with data, for use in data capture at the receiver. DQS is a strobe transmitted by the DDR SDRAM during Reads and by the memory controller during Writes. DQS is edge-aligned with data for Reads and center-aligned with data for Writes.

The 256Mb DDR SDRAM operates from a differential clock (CK and  $\overline{CK}$ ; the crossing of CK going HIGH and CK going LOW is referred to as the positive edge of CK). Commands (address and control signals) are registered at every positive edge of CK. Input data is registered on both edges of DQS, and output data is referenced to both edges of DQS, as well as to both edges of CK.

Read and write accesses to the DDR SDRAM are burst oriented; accesses start at a selected location and continue for a programmed number of locations in a programmed sequence. Accesses begin with the registration of an Active command, which is then followed by a Read or Write command. The address bits registered coincident with the Active command are used to select the bank and row to be accessed. The address bits registered coincident with the Read or Write command are used to select the bank and the starting column location for the burst access.

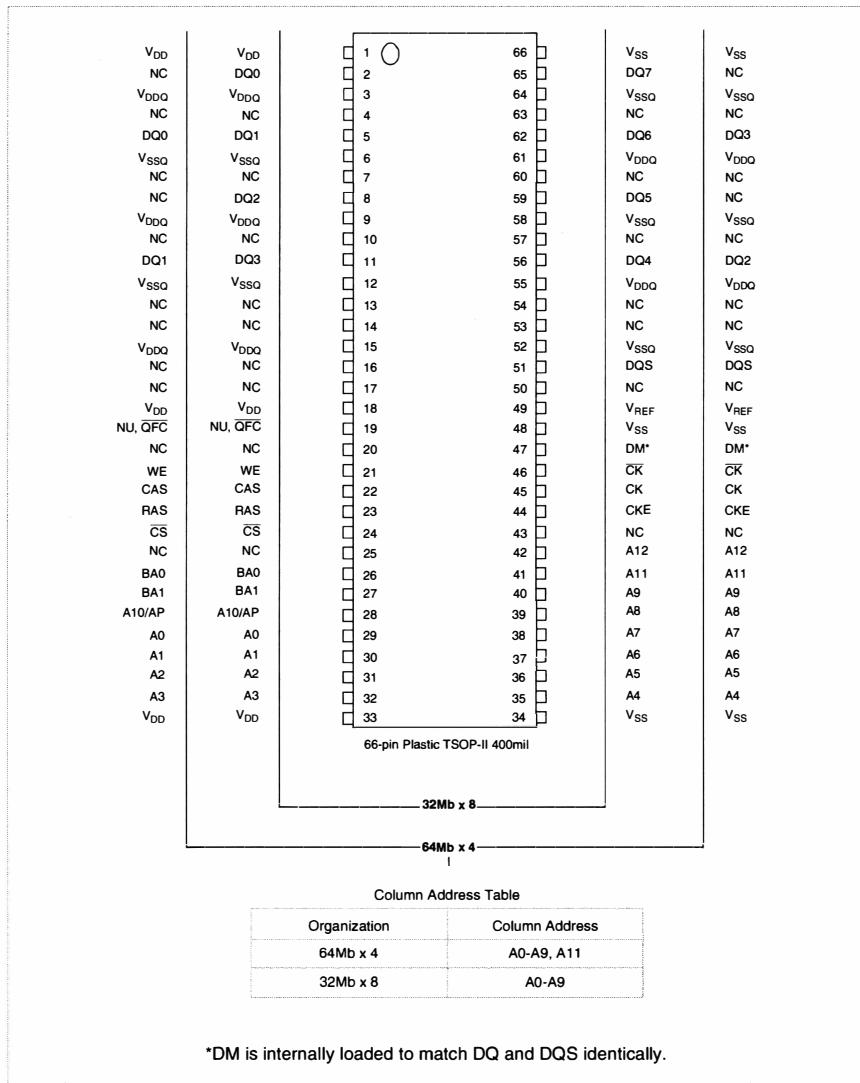
The DDR SDRAM provides for programmable Read or Write burst lengths of 2, 4 or 8 locations. An Auto Precharge function may be enabled to provide a self-timed row precharge that is initiated at the end of the burst access.

As with standard SDRAMs, the pipelined, multibank architecture of DDR SDRAMs allows for concurrent operation, thereby providing high effective bandwidth by hiding row precharge and activation time.

An auto refresh mode is provided along with a power-saving power-down mode. All inputs are compatible with the JEDEC Standard for SSTL\_2. All outputs are SSTL\_2, Class II compatible.

**Note:** The functionality described and the timing specifications included in this data sheet are for the DLL Enabled mode of operation.

### Pin Configuration



### Input/Output Functional Description

Symbol	Type	Function
CK, $\bar{CK}$	Input	<b>Clock:</b> CK and $\bar{CK}$ are differential clock inputs. All address and control input signals are sampled on the crossing of the positive edge of CK and negative edge of $\bar{CK}$ . Output (read) data is referenced to the crossings of CK and $\bar{CK}$ (both directions of crossing).
CKE	Input	<b>Clock Enable:</b> CKE HIGH activates, and CKE Low deactivates, internal clock signals and device input buffers and output drivers. Taking CKE Low provides Precharge Power-Down and Self Refresh operation (all banks idle), or Active Power-Down (row Active in any bank). CKE is synchronous for power down entry and exit, and for self refresh entry. CKE is asynchronous for self refresh exit. CKE must be maintained high throughout read and write accesses. Input buffers, excluding CK, $\bar{CK}$ and CKE are disabled during power-down. Input buffers, excluding CKE, are disabled during self refresh.
$\bar{CS}$	Input	<b>Chip Select:</b> All commands are masked when CS is registered HIGH. $\bar{CS}$ provides for external bank selection on systems with multiple banks. CS is considered part of the command code. The standard pinout includes one $\bar{CS}$ pin.
RAS, CAS, WE	Input	<b>Command Inputs:</b> RAS, CAS and WE (along with $\bar{CS}$ ) define the command being entered.
DM	Input	<b>Input Data Mask:</b> DM is an input mask signal for write data. Input data is masked when DM is sampled HIGH coincident with that input data during a Write access. DM is sampled on both edges of DQS. Although DM pins are input only, the DM loading matches the DQ and DQS loading.
BA0, BA1	Input	<b>Bank Address Inputs:</b> BA0 and BA1 define to which bank an Active, Read, Write or Precharge command is being applied. BA0 and BA1 also determines if the mode register or extended mode register is to be accessed during a MRS or EMRS cycle.
A0 - A12	Input	<b>Address Inputs:</b> Provide the row address for Active commands, and the column address and Auto Precharge bit for Read/Write commands, to select one location out of the memory array in the respective bank. A10 is sampled during a Precharge command to determine whether the Precharge applies to one bank (A10 LOW) or all banks (A10 HIGH). If only one bank is to be precharged, the bank is selected by BA0, BA1. The address inputs also provide the op-code during a Mode Register Set command.
DQ	Input/Output	<b>Data Input/Output:</b> Data bus.
DQS	Input/Output	<b>Data Strobe:</b> Output with read data, input with write data. Edge-aligned with read data, centered in write data. Used to capture write data.
$\bar{QFC}$	Output	<b>FET control:</b> Optional. Output during every Read and Write access. Is provided to control isolation switches on modules. Open drain output. Pullup resistor must be tied to $V_{DDQ}$ at second level of assembly. The QFC pin is present on this product version, but all timings parameters related to this pin are not tested on the final product and are only guaranteed by design.
NC		<b>No Connect:</b> No internal electrical connection is present.
$V_{DDQ}$	Supply	<b>DQ Power Supply:</b> $2.5V \pm 0.2V$ .
$V_{SSQ}$	Supply	<b>DQ Ground</b>
$V_{DD}$	Supply	<b>Power Supply:</b> $2.5V \pm 0.2V$ .
$V_{SS}$	Supply	<b>Ground</b>
$V_{REF}$	Supply	<b>SSTL_2 reference voltage:</b> $(V_{DDQ} / 2)$

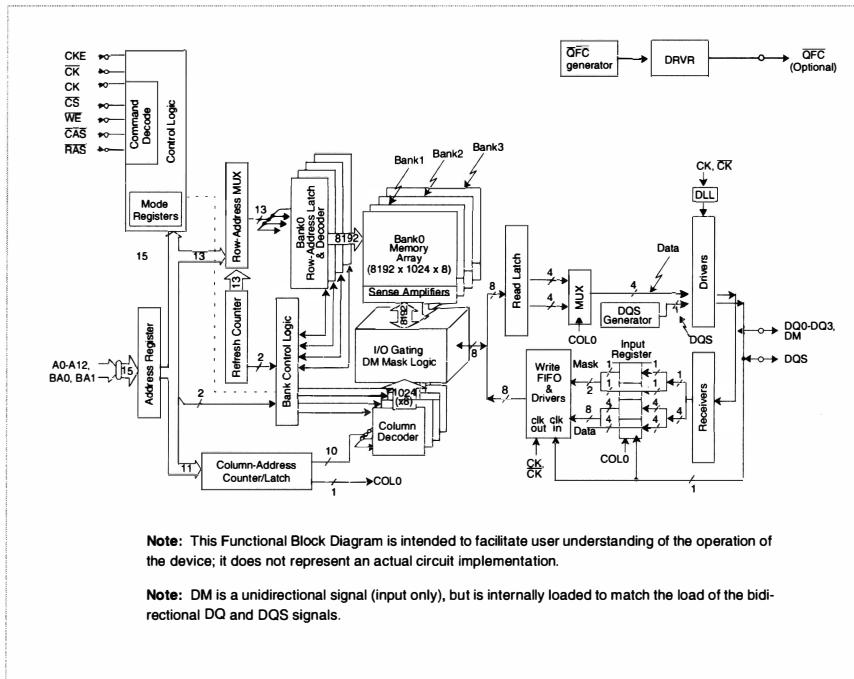


**HYB25D256400/800T/AT**  
**256-Mbit Double Data Rate SDRAM**

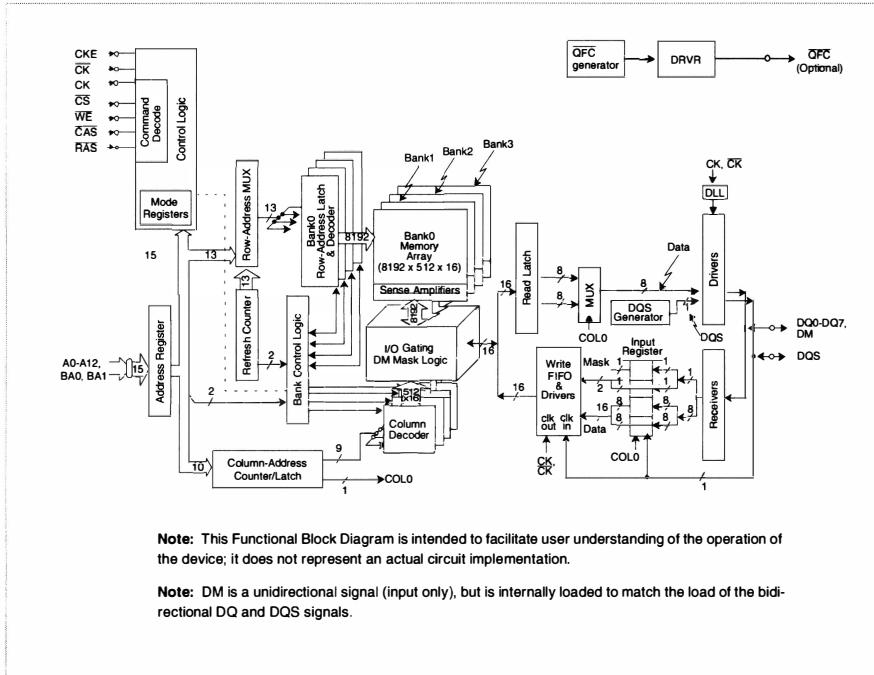
**Ordering Information**

Part Number (ASTC)	CAS Latency	Clock (MHz)	CAS Latency	Clock (MHz)	Speed	Org.	Package	
HYB25D256400T-7	2.5	143	2	133	DDR266A	x 4	66 pin TSOP-II	
HYB25D256800T-7				125	DDR266B	x 8		
HYB25D256400T-7.5				100	DDR200	x 4		
HYB25D256800T-7.5		133		125	DDR266B	x 8		
HYB25D256400T-8				100	DDR200	x 4		
HYB25D256800T-8				100	DDR200	x 8		

Part Number (WOS)	CAS Latency	Clock (MHz)	CAS Latency	Clock (MHz)	Speed	Org.	Package	
HYB25D256400AT-7	2.5	143	2	133	DDR266A	x 4	66 pin TSOP-II	
HYB25D256800AT-7				125	DDR266B	x 8		
HYB25D256400AT-7.5				100	DDR200	x 4		
HYB25D256800AT-7.5		133		125	DDR266B	x 8		
HYB25D256400AT-8				100	DDR200	x 4		
HYB25D256800AT-8				100	DDR200	x 8		

**Block Diagram (64Mb x 4)**


### Block Diagram (32Mb x 8)



## Functional Description

The 256Mb DDR SDRAM is a high-speed CMOS, dynamic random-access memory containing 268, 435, 456 bits. The 256Mb DDR SDRAM is internally configured as a quad-bank DRAM.

The 256Mb DDR SDRAM uses a double-data-rate architecture to achieve high-speed operation. The double-data-rate architecture is essentially a  $2n$  prefetch architecture, with an interface designed to transfer two data words per clock cycle at the I/O pins. A single read or write access for the 256Mb DDR SDRAM consists of a single  $2n$ -bit wide, one clock cycle data transfer at the internal DRAM core and two corresponding  $n$ -bit wide, one-half clock cycle data transfers at the I/O pins.

Read and write accesses to the DDR SDRAM are burst oriented; accesses start at a selected location and continue for a programmed number of locations in a programmed sequence. Accesses begin with the registration of an Active command, which is then followed by a Read or Write command. The address bits registered coincident with the Active command are used to select the bank and row to be accessed (BA0, BA1 select the bank; A0-A12 select the row). The address bits registered coincident with the Read or Write command are used to select the starting column location for the burst access.

Prior to normal operation, the DDR SDRAM must be initialized. The following sections provide detailed information covering device initialization, register definition, command descriptions and device operation.

## Initialization

DDR SDRAMs must be powered up and initialized in a predefined manner. Operational procedures other than those specified may result in undefined operation. The following two conditions must be met:

No power sequencing is specified during power up or power down given the following criteria

$V_{DD}$  and  $V_{DDQ}$  are driven from a single power converter output

$V_{TT}$  meets the specification

A minimum resistance of 42 ohms limits the input current from the  $V_{TT}$  supply into any pin and  $V_{REF}$  tracks  $V_{DDQ}/2$

or

The following relationship must be followed

$V_{DDQ}$  is driven after or with  $V_{DD}$  such that  $V_{DDQ} < V_{DD} + 0.3$  V

$V_{TT}$  is driven after or with  $V_{DDQ}$  such that  $V_{TT} < V_{DDQ} + 0.3$  V

$V_{REF}$  is driven after or with  $V_{DDQ}$  such that  $V_{REF} < V_{DDQ} + 0.3$  V

The DQ and DQS outputs are in the High-Z state, where they remain until driven in normal operation (by a read access). After all power supply and reference voltages are stable, and the clock is stable, the DDR SDRAM requires a  $200\mu s$  delay prior to applying an executable command.

Once the  $200\mu s$  delay has been satisfied, a Deselect or NOP command should be applied, and CKE must be brought HIGH. Following the NOP command, a Precharge ALL command must be applied. Next a Mode Register Set command must be issued for the Extended Mode Register, to enable the DLL, then a Mode Register Set command must be issued for the Mode Register, to reset the DLL, and to program the operating parameters. 200 clock cycles are required between the DLL reset and any read command. A Precharge ALL command should be applied, placing the device in the "all banks idle" state

Once in the idle state, two AUTO REFRESH cycles must be performed. Additionally, a Mode Register Set command for the Mode Register, with the reset DLL bit deactivated (i.e. to program operating parameters without resetting the DLL) must be performed. Following these cycles, the DDR SDRAM is ready for normal operation.

---

## Register Definition

### Mode Register

The Mode Register is used to define the specific mode of operation of the DDR SDRAM. This definition includes the selection of a burst length, a burst type, a CAS latency, and an operating mode. The Mode Register is programmed via the Mode Register Set command (with BA0 = 0 and BA1 = 0) and retains the stored information until it is programmed again or the device loses power (except for bit A8, which is self-clearing).

Mode Register bits A0-A2 specify the burst length, A3 specifies the type of burst (sequential or interleaved), A4-A6 specify the CAS latency, and A7-A12 specify the operating mode.

The Mode Register must be loaded when all banks are idle, and the controller must wait the specified time before initiating the subsequent operation. Violating either of these requirements results in unspecified operation.

### Burst Length

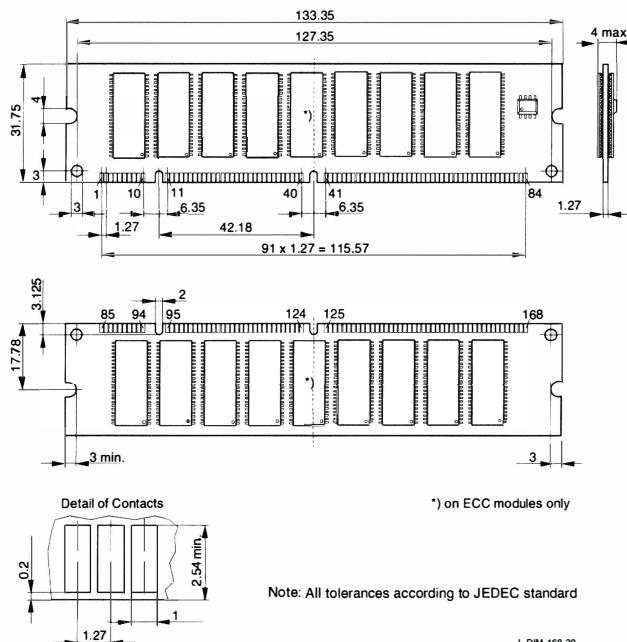
Read and write accesses to the DDR SDRAM are burst oriented, with the burst length being programmable. The burst length determines the maximum number of column locations that can be accessed for a given Read or Write command. Burst lengths of 2, 4, or 8 locations are available for both the sequential and the interleaved burst types.

Reserved states should not be used, as unknown operation or incompatibility with future versions may result.

When a Read or Write command is issued, a block of columns equal to the burst length is effectively selected. All accesses for that burst take place within this block, meaning that the burst wraps within the block if a boundary is reached. The block is uniquely selected by A1-Ai when the burst length is set to two, by A2-Ai when the burst length is set to four and by A3-Ai when the burst length is set to eight (where Ai is the most significant column address bit for a given configuration). The remaining (least significant) address bit(s) is (are) used to select the starting location within the block. The programmed burst length applies to both Read and Write bursts.

### Package Outlines

**L-DIM-168-30 (JEDEC MO-161-BA)**  
**SDRAM DIMM Module Package**  
**HYS 64/72V64220GU**



## 256 BIT FULLY DECODED RANDOM ACCESS MEMORY

- Access Time -- Typically Below 650 nsec - 1101A1, 850 nsec - 1101A
- Low Power Standby Mode
- Low Power Dissipation -- Typically less than 1.5 mW/bit during access
- Directly DTL and TTL Compatible
- OR-Tie Capability
- Simple Memory Expansion -- Chip Select Input Lead
- Fully Decoded -- On Chip Address Decode and Sense
- Inputs Protected -- All Inputs Have Protection Against Static Charge
- Ceramic and Plastic Package -- 16 Pin Dual In-Line Configuration

The 1101A is an improved version of the 1101 which requires only two power supplies (+5V and -9V) for operation. The 1101A is a direct pin for pin replacement for the 1101.

The Intel 1101A is a 256 word by 1 bit random access memory element using normally off P-channel MOS devices integrated on a monolithic array. It uses fully dc stable (static) circuitry and therefore requires no clocks to operate.

The 1101A is designed primarily for small buffer storage applications where high performance, low cost, and ease of interfacing with other standard logic circuits are important design objectives. The unit will directly interface with standard bipolar integrated logic circuits (TTL, DTL, etc.) The data output buffers are capable of driving TTL loads directly. A separate chip select (CS) lead allows easy selection of an individual package when outputs are OR-tied.

For applications requiring a faster access time we recommend the 1101A1 which is a selection from the 1101A and has a guaranteed maximum access time of 1.0  $\mu$ sec.

The Intel 1101A is fabricated with silicon gate technology. This low threshold technology allows the design and production of higher performance MOS circuits and provides a higher functional density on a monolithic chip than conventional MOS technologies.

Intel's silicon gate technology also provides excellent protection against contamination. This permits the use of low cost silicone packaging.

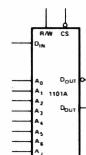
PIN CONFIGURATION



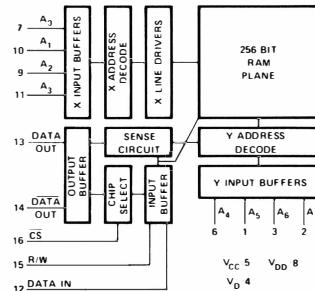
PIN NAMES

$D_{IN}$	DATA INPUT	CS	CHIP SELECT
$A_0 - A_7$	ADDRESS INPUTS R/W READ/WRITE INPUT	$D_{OUT}$	DATA OUTPUT

LOGIC SYMBOL



BLOCK DIAGRAM



## FULLY DECODED RANDOM ACCESS 1024 BIT DYNAMIC MEMORY

- Low Power Dissipation – Dissipates Power Primarily on Selected Chips
- Access Time – 300 nsec
- Cycle Time – 580 nsec
- Refresh Period... 2 milliseconds for 0–70°C Ambient
- OR-Tie Capability
- Simple Memory Expansion – Chip Enable Input Lead
- Fully Decoded – on Chip Address Decode
- Inputs Protected – All Inputs Have Protection Against Static Charge
- Ceramic and Plastic Package -- 18 Pin Dual In-Line Configuration.

The Intel 1103 is designed primarily for main memory applications where high performance, low cost, and large bit storage are important design objectives.

It is a 1024 word by 1 bit random access memory element using normally off P-channel MOS devices integrated on a monolithic array. It is fully decoded, permitting the use of an 18 pin dual in-line package. It uses dynamic circuitry and primarily dissipates power only during precharge.

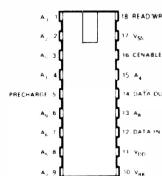
Information stored in the memory is non-destructively read. Refreshing of all 1024 bits is accomplished in 32 read cycles and is required every two milliseconds.

A separate **cenable** (chip enable) lead allows easy selection of an individual package when outputs are OR-tied.

The Intel 1103 is fabricated with **silicon gate technology**. This **low threshold** technology allows the design and production of higher performance MOS circuits and provides a higher functional density on a monolithic chip than conventional MOS technologies.

Intel's silicon gate technology also provides excellent protection against contamination. This permits the use of low cost plastic packaging.

PIN CONFIGURATION



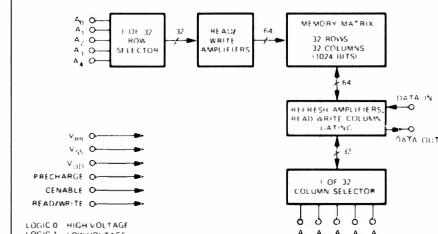
LOGIC SYMBOL



PIN NAMES

D <sub>IN</sub>	DATA INPUT	PRC	PREFECH INPUT
A <sub>0</sub> -A <sub>9</sub>	ADDRESS INPUTS	CE	CHIP ENABLE
R/W	READ/WRITE	D <sub>OUT</sub>	DATA OUTPUT

BLOCK DIAGRAM



## 2048 BIT FULLY DECODED READ ONLY MEMORY

### 1601/1701, 1602/1702/ ELECTRICALLY PROGRAMMABLE 1301 MASK PROGRAMMABLE

- Erasable and Field Reprogrammable (1701,1702)
- Field Programmable (1601,1602)
- All 2048 Bits Guaranteed Programmable – 100% factory tested (1601/1701, 1602/1702)
- Inputs and outputs DTL and TTL compatible
- Static and Dynamic Operation (1601,1701,1301)
- Static Only Operation (1602,1702)
- OR-tie capability
- Simple Memory Expansion – Chip Select input lead
- 24 pin dual-in-line hermetically sealed ceramic package (1601,1602,1301)
- 24 pin dual-in-line, quartz lid ceramic package (1701,1702)

The Intel 1601, 1602, 1701, and 1702 is a 256 word by 8 bit electrically programmable ROM ideally suited for uses where fast turnaround and pattern experimentation are important such as in prototype or in one of a kind systems. The 1601, 1602, 1701, and 1702 is factory reprogrammable which allows Intel to perform a complete programming and functional test on each bit position before delivery.

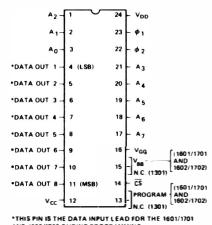
The four devices 1601, 1602, 1701, and 1702 use identical chips. The 1601 and 1701 is operable in both the static and dynamic mode while the 1602 and 1702 is operable in the static mode only. Also, the 1701 and 1702 has the unique feature of being completely erasable and field reprogrammable. This is accomplished by a quartz lid that allows high intensity ultraviolet light to erase the 1701 and 1702. A new pattern can then be written into the device. This procedure can be repeated as many times as required.

The 1301 is a pin-for-pin replacement part which is programmed by a metal mask and is ideal for large volume and lower cost production runs of systems initially using the 1601/1701 or the static only 1602/1702.

The dynamic mode of the 1601/1701 and 1301 refers to the decoding circuitry and not to the memory cell. Dynamic operation offers higher speed and lower power dissipation than the static operation.

The 1601, 1602, 1701, and 1702 is fabricated with silicon gate technology. This low threshold technology allows the design and production of higher performance MOS circuits and provides a higher functional density on a monolithic chip than conventional MOS technologies.

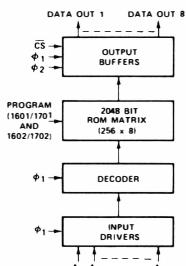
PIN CONFIGURATION



PIN NAMES

$A_0-A_7$	Address Inputs
$\phi_1, \phi_2$	Clocks for Dynamic Mode
CS	Chip Select Input
$D_{OUT1}-D_{OUT8}$	Data Outputs

BLOCK DIAGRAM



NOTE LOGIC 1 AT INPUT AND OUTPUT IS A HIGH AND LOGIC 0 IS LOW.

# Appendix 5 PLD – datablad

- Utdrag ur Monolithic Memories: "PAL Programmable Array Logic Handbook", First Edition 1978
- Altera MAX 7000
- XILINX Virtex-II

# Programmable Array Logic Family

## PAL Series 20 Data Sheet

Patent Allowed



### Features/Benefits

- Programmable replacement for conventional TTL logic.
- Reduces IC inventories substantially and simplifies their control.
- Reduces chip count by 4 to 1.
- Expedites and simplifies prototyping and board layout.
- Saves space with 20-pin Skinny DIP packages.
- High speed: 25ns typical propagation delay.
- Programmed on standard PROM programmers.
- Programmable three-state outputs.
- Special feature reduces possibility of copying by competitors.

### Description

The PAL family utilizes an advanced Schottky TTL process and the Bipolar PROM fusible link technology to provide user programmable logic for replacing conventional SSI/MSI gates and flip-flops at reduced chip count.

The family lets the systems engineer "design his own chip" by blowing fusible links to configure AND and OR gates to perform his desired logic function. Complex interconnections which previously required time-consuming layout are thus "lifted" from PC board etch and placed on silicon where they can be easily modified during prototype check-out or production.

The PAL transfer function is the familiar sum of products. Like the PROM, the PAL has a single array of fusible links. Unlike the PROM, the PAL is a programmable AND array driving a fixed OR array (the PROM is a fixed AND array driving a programmable OR array). In addition the PAL provides these options:

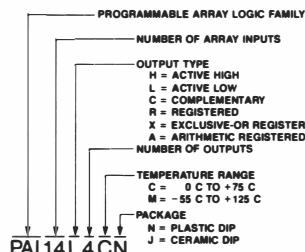
- Variable input/output pin ratio
- Programmable three-state outputs
- Registers with feedback
- Arithmetic capability

Unused inputs are tied directly to V<sub>CC</sub> or GND. Product terms with all fuses blown assume the logical high state, and product terms connected to both true and complement of any single input assume the logical low state. Registers consist of D type flip-flops which are loaded on the low to high transition of the clock. All registers are designed to power up to logical high state at the output pin. PAL Logic Diagrams are shown with all fuses blown, enabling the designer use of the diagrams as coding sheets. 8½ x 11 Logic Diagrams are available on request.

The entire PAL family is programmed on inexpensive conventional PROM programmers with appropriate personality and socket adapter cards. Once the PAL is programmed and verified, two additional fuses may be blown to defeat verification. This feature gives the user a proprietary circuit which is very difficult to copy.

PART NUMBER	DESCRIPTION
PAL10H	OCTAL 10 INPUT AND-OR GATE ARRAY
PAL12H	HEX 12 INPUT AND-OR GATE ARRAY
PAL14H	QUAD 14 INPUT AND-OR GATE ARRAY
PAL16H	DUAL 16 INPUT AND-OR GATE ARRAY
PAL16C1	16 INPUT AND-OR/AND-OR-INVERT GATE ARRAY
PAL10L8	OCTAL 10 INPUT AND-OR-INVERT GATE ARRAY
PAL12L6	HEX 12 INPUT AND-OR-INVERT GATE ARRAY
PAL14L4	QUAD 14 INPUT AND-OR-INVERT GATE ARRAY
PAL16L2	DUAL 16 INPUT AND-OR-INVERT GATE ARRAY
PAL16L8	OCTAL 16 INPUT AND-OR-INVERT GATE ARRAY
PAL16R8	OCTAL 16 INPUT REGISTERED AND-OR GATE ARRAY
PAL16R6	HEX 16 INPUT REGISTERED AND-OR GATE ARRAY
PAL16R4	QUAD 16 INPUT REGISTERED AND-OR GATE ARRAY
PAL16X4	QUAD 16 INPUT REGISTERED AND-OR-XOR GATE ARRAY
PAL16A4	QUAD 16 INPUT REGISTERED AND-CARRY-OR-XOR GATE ARRAY

### Ordering Information

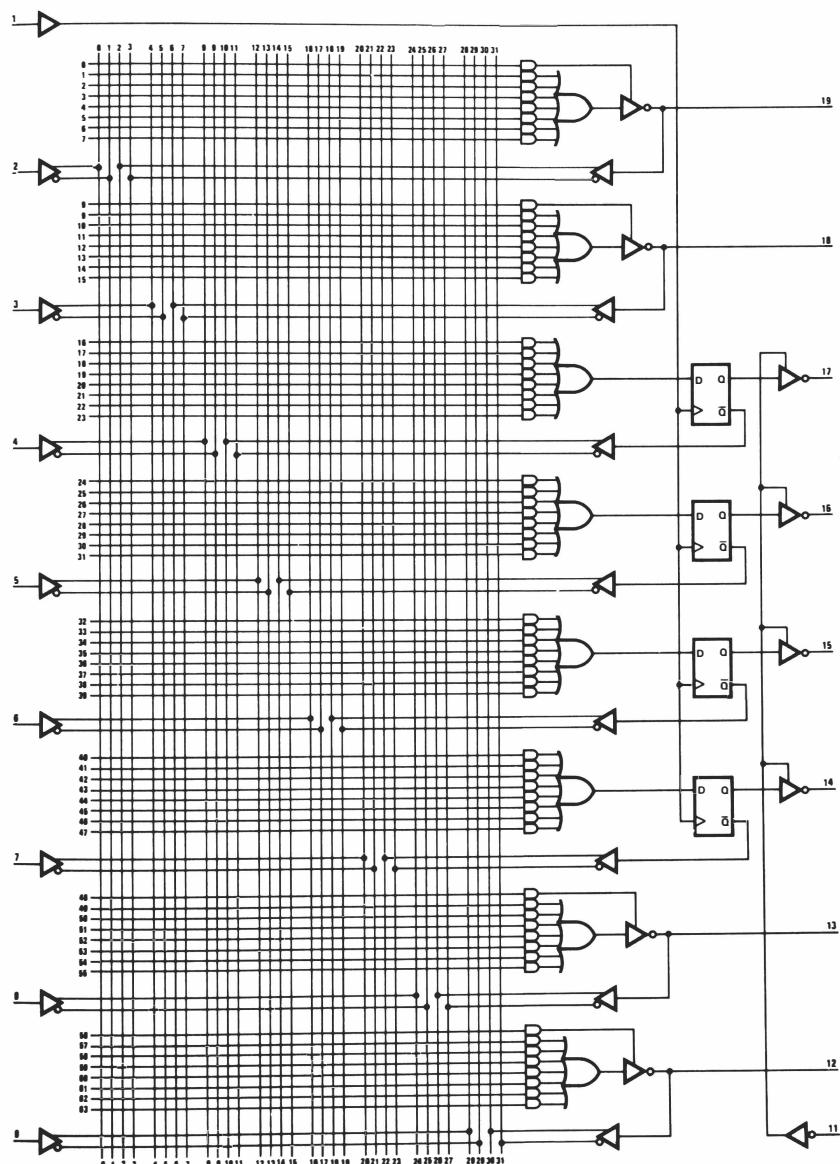


1165 East Arques Avenue, Sunnyvale, CA 94086 Tel: (408) 739-3535 TWX: 910-339-9229

**Monolithic Memories**

## PAL Family

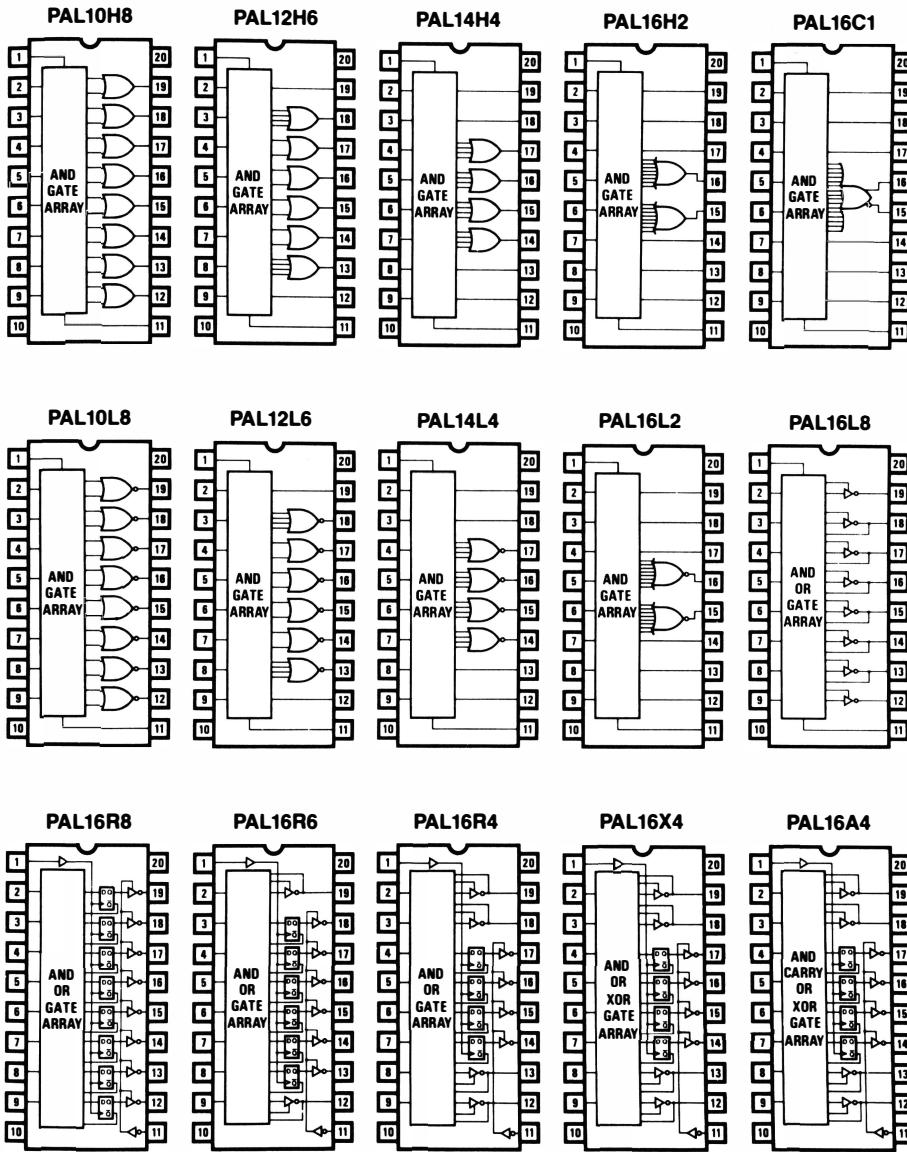
Logic Diagram PAL16R4

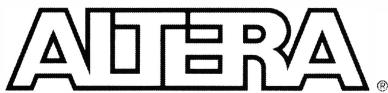


3-21

## PAL Family

### PAL Logic Symbols





Includes  
MAX 7000AE

# MAX 7000A

## Programmable Logic Device

April 2001, ver. 4.0

Data Sheet

### Features...

- High-performance 3.3-V EEPROM-based programmable logic devices (PLDs) built on second-generation Multiple Array Matrix (MAX®) architecture (see Table 1)
- 3.3-V in-system programmability (ISP) through the built-in IEEE Std. 1149.1 Joint Test Action Group (JTAG) interface with advanced pin-locking capability
  - MAX 7000AE device in-system programmability (ISP) circuitry compliant with IEEE Std. 1532
  - EPM7128A and EPM7256A device ISP circuitry compatible with IEEE Std. 1532
- Built-in boundary-scan test (BST) circuitry compliant with IEEE Std. 1149.1
- Supports JEDEC Jam Standard Test and Programming Language (STAPL) JESD-71
- Enhanced ISP features
  - Enhanced ISP algorithm for faster programming (excluding EPM7128A and EPM7256A devices)
  - ISP\_Done bit to ensure complete programming (excluding EPM7128A and EPM7256A devices)
  - Pull-up resistor on I/O pins during in-system programming
- Pin-compatible with the popular 5.0-V MAX 7000S devices
- High-density PLDs ranging from 600 to 10,000 usable gates
- 4.5-ns pin-to-pin logic delays with counter frequencies of up to 227.3 MHz

For information on in-system programmable 5.0-V MAX 7000 or 2.5-V MAX 7000B devices, see the *MAX 7000 Programmable Logic Device Family Data Sheet* or the *MAX 7000B Programmable Logic Device Family Data Sheet*.

**Table 1. MAX 7000A Device Features**

Feature	EPM7032AE	EPM7064AE	EPM7128AE	EPM7256AE	EPM7512AE
Usable gates	600	1,250	2,500	5,000	10,000
Macrocells	32	64	128	256	512
Logic array blocks	2	4	8	16	32
Maximum user I/O pins	36	68	100	164	212
$t_{PD}$ (ns)	4.5	4.5	5.0	5.5	7.5
$t_{SU}$ (ns)	2.9	2.8	3.3	3.9	5.6
$t_{FSU}$ (ns)	2.5	2.5	2.5	2.5	3.0
$t_{CO1}$ (ns)	3.0	3.1	3.4	3.5	4.7
$f_{CNT}$ (MHz)	227.3	222.2	192.3	172.4	116.3

## ...and More Features

- MultiVolt™ I/O interface enables device core to run at 3.3 V, while I/O pins are compatible with 5.0-V, 3.3-V, and 2.5-V logic levels
- Pin counts ranging from 44 to 256 in a variety of thin quad flat pack (TQFP), plastic quad flat pack (PQFP), ball-grid array (BGA), space-saving FineLine BGA™, and plastic J-lead chip carrier (PLCC) packages
- Supports hot-socketing in MAX 7000AE devices
- Programmable interconnect array (PIA) continuous routing structure for fast, predictable performance
- PCI-compatible
- Bus-friendly architecture, including programmable slew-rate control
- Open-drain output option
- Programmable macrocell registers with individual clear, preset, clock, and clock enable controls
- Programmable power-up states for macrocell registers in MAX 7000AE devices
- Programmable power-saving mode for 50% or greater power reduction in each macrocell
- Configurable expander product-term distribution, allowing up to 32 product terms per macrocell
- Programmable security bit for protection of proprietary designs
- 6 to 10 pin- or logic-driven output enable signals
- Two global clock signals with optional inversion
- Enhanced interconnect resources for improved routability
- Fast input setup times provided by a dedicated path from I/O pin to macrocell registers
- Programmable output slew-rate control
- Programmable ground pins
- Software design support and automatic place-and-route provided by Altera's development systems for Windows-based PCs and Sun SPARCstation, and HP 9000 Series 700/800 workstations

- Additional design entry and simulation support provided by EDIF 200 and 300 netlist files, library of parameterized modules (LPM), Verilog HDL, VHDL, and other interfaces to popular EDA tools from manufacturers such as Cadence, Exemplar Logic, Mentor Graphics, OrCAD, Synopsys, Synplicity, and VeriBest
- Programming support with Altera's Master Programming Unit (MPU), MasterBlaster™ serial/universal serial bus (USB) communications cable, ByteBlasterMV™ parallel port download cable, and BitBlaster™ serial download cable, as well as programming hardware from third-party manufacturers and any Jam™ STAPL File (.jam), Jam Byte-Code File (.jbc), or Serial Vector Format File- (.svf) capable in-circuit tester

## General Description

MAX 7000A (including MAX 7000AE) devices are high-density, high-performance devices based on Altera's second-generation MAX architecture. Fabricated with advanced CMOS technology, the EEPROM-based MAX 7000A devices operate with a 3.3-V supply voltage and provide 600 to 10,000 usable gates, ISP, pin-to-pin delays as fast as 4.5 ns, and counter speeds of up to 227.3 MHz. MAX 7000A devices in the -4, -5, -6, -7, and some -10 speed grades are compatible with the timing requirements for 33 MHz operation of the PCI Special Interest Group (PCI SIG) *PCI Local Bus Specification, Revision 2.2*. See Table 2.

**Table 2. MAX 7000A Speed Grades**

Device	Speed Grade					
	-4	-5	-6	-7	-10	-12
EPM7032AE	✓			✓	✓	
EPM7064AE	✓			✓	✓	
EPM7128A (1)			✓	✓	✓	✓
EPM7128AE		✓		✓	✓	
EPM7256A (1)			✓	✓	✓	✓
EPM7256AE		✓		✓	✓	
EPM7512AE				✓	✓	✓

*Note:*

- (1) Altera does not recommend using EPM7128A or EPM7256A devices for new designs. Use EPM7128AE or EPM7256AE devices for these designs instead.

The MAX 7000A architecture supports 100% transistor-to-transistor logic (TTL) emulation and high-density integration of SSI, MSI, and LSI logic functions. It easily integrates multiple devices including PALs, GALs, and 22V10s devices. MAX 7000A devices are available in a wide range of packages, including PLCC, BGA, FineLine BGA, Ultra FineLine BGA, PQFP, and TQFP packages. See Table 3 and Table 4.

**Table 3. MAX 7000A Maximum User I/O Pins** Note (1)

Device	44-Pin PLCC	44-Pin TQFP	49-Pin Ultra FineLine BGA (2)	84-Pin PLCC	100-Pin TQFP	100-Pin FineLine BGA (3)
EPM7032AE	36	36				
EPM7064AE	36	36	41		68	68
EPM7128A (4)				68	84	84
EPM7128AE				68	84	84
EPM7256A (4)					84	
EPM7256AE					84	84
EPM7512AE						

**Table 4. MAX 7000A Maximum User I/O Pins** Note (1)

Device	144-Pin TQFP	169-Pin Ultra FineLine BGA (2)	208-Pin PQFP	256-Pin BGA	256-Pin FineLine BGA (3)
EPM7032AE					
EPM7064AE					
EPM7128A (4)	100				100
EPM7128AE	100	100			100
EPM7256A (4)	120		164		164
EPM7256AE	120		164		164
EPM7512AE	120		176	212	212

**Notes to tables:**

- (1) When the IEEE Std. 1149.1 (JTAG) interface is used for in-system programming or boundary-scan testing, four I/O pins become JTAG pins.
- (2) All Ultra FineLine BGA packages are footprint-compatible via the SameFrame™ feature. Therefore, designers can design a board to support a variety of devices, providing a flexible migration path across densities and pin counts. Device migration is fully supported by Altera development tools. See “SameFrame Pin-Outs” on page 15 for more details.
- (3) All FineLine BGA packages are footprint-compatible via the SameFrame feature. Therefore, designers can design a board to support a variety of devices, providing a flexible migration path across densities and pin counts. Device migration is fully supported by Altera development tools. See “SameFrame Pin-Outs” on page 15 for more details.
- (4) Altera does not recommend using EPM7128A or EPM7256A devices for new designs. Use EPM7128AE or EPM7256AE devices for these designs instead.

MAX 7000A devices use CMOS EEPROM cells to implement logic functions. The user-configurable MAX 7000A architecture accommodates a variety of independent combinatorial and sequential logic functions. The devices can be reprogrammed for quick and efficient iterations during design development and debug cycles, and can be programmed and erased up to 100 times.

MAX 7000A devices contain from 32 to 512 macrocells that are combined into groups of 16 macrocells, called logic array blocks (LABs). Each macrocell has a programmable-AND/fixed-OR array and a configurable register with independently programmable clock, clock enable, clear, and preset functions. To build complex logic functions, each macrocell can be supplemented with both shareable expander product terms and high-speed parallel expander product terms, providing up to 32 product terms per macrocell.

MAX 7000A devices provide programmable speed/power optimization. Speed-critical portions of a design can run at high speed/full power, while the remaining portions run at reduced speed/low power. This speed/power optimization feature enables the designer to configure one or more macrocells to operate at 50% or lower power while adding only a nominal timing delay. MAX 7000A devices also provide an option that reduces the slew rate of the output buffers, minimizing noise transients when non-speed-critical signals are switching. The output drivers of all MAX 7000A devices can be set for 2.5 V or 3.3 V, and all input pins are 2.5-V, 3.3-V, and 5.0-V tolerant, allowing MAX 7000A devices to be used in mixed-voltage systems.

MAX 7000A devices are supported by Altera development systems, which are integrated packages that offer schematic, text—including VHDL, Verilog HDL, and the Altera Hardware Description Language (AHDL)—and waveform design entry, compilation and logic synthesis, simulation and timing analysis, and device programming. The software provides EDIF 2.0.0 and 3.0.0, LPM, VHDL, Verilog HDL, and other interfaces for additional design entry and simulation support from other industry-standard PC- and UNIX-workstation-based EDA tools. The software runs on Windows-based PCs, as well as Sun SPARCstation, and HP 9000 Series 700/800 workstations.

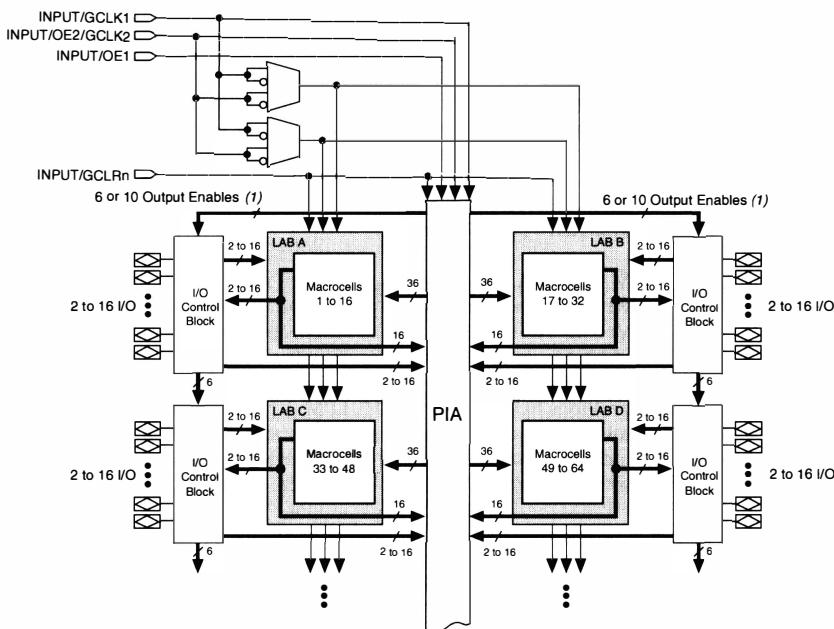
For more information on development tools, see the *MAX+PLUS II Programmable Logic Development System & Software Data Sheet* and the *Quartus Programmable Logic Development System & Software Data Sheet*.

## Functional Description

The MAX 7000A architecture includes the following elements:

- Logic array blocks (LABs)
- Macrocells
- Expander product terms (shareable and parallel)
- Programmable interconnect array
- I/O control blocks

The MAX 7000A architecture includes four dedicated inputs that can be used as general-purpose inputs or as high-speed, global control signals (clock, clear, and two output enable signals) for each macrocell and I/O pin. Figure 1 shows the architecture of MAX 7000A devices.

**Figure 1. MAX 7000A Device Block Diagram****Note:**

- (1) EPM7032AE, EPM7064AE, EPM7128A, EPM7128AE, EPM7256A, and EPM7256AE devices have six output enables. EPM7512AE devices have 10 output enables.

### Logic Array Blocks

The MAX 7000A device architecture is based on the linking of high-performance LABs. LABs consist of 16-macrocell arrays, as shown in Figure 1. Multiple LABs are linked together via the PIA, a global bus that is fed by all dedicated input pins, I/O pins, and macrocells.

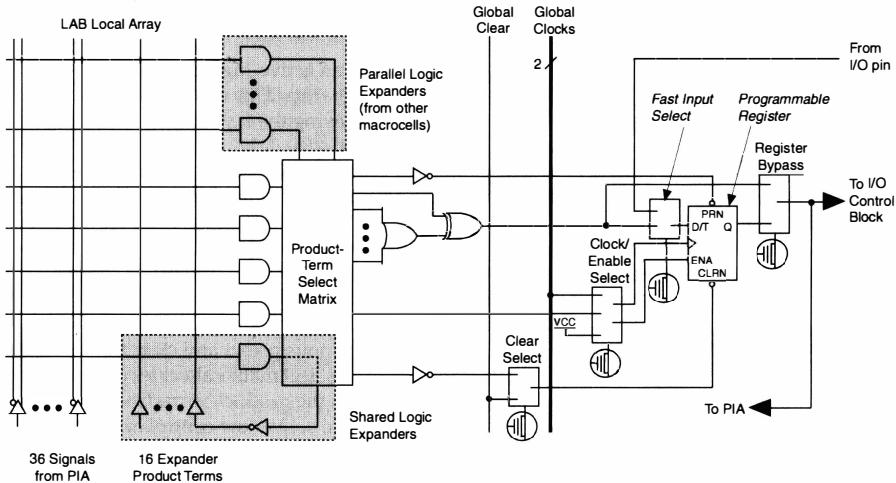
Each LAB is fed by the following signals:

- 36 signals from the PIA that are used for general logic inputs
- Global controls that are used for secondary register functions
- Direct input paths from I/O pins to the registers that are used for fast setup times

## Macrocells

MAX 7000A macrocells can be individually configured for either sequential or combinatorial logic operation. The macrocells consist of three functional blocks: the logic array, the product-term select matrix, and the programmable register. Figure 2 shows a MAX 7000A macrocell.

**Figure 2. MAX 7000A Macrocell**



Combinatorial logic is implemented in the logic array, which provides five product terms per macrocell. The product-term select matrix allocates these product terms for use as either primary logic inputs (to the OR and XOR gates) to implement combinatorial functions, or as secondary inputs to the macrocell's register preset, clock, and clock enable control functions.

Two kinds of expander product terms ("expanders") are available to supplement macrocell logic resources:

- Shareable expanders, which are inverted product terms that are fed back into the logic array
- Parallel expanders, which are product terms borrowed from adjacent macrocells

The Altera development system automatically optimizes product-term allocation according to the logic requirements of the design.

For registered functions, each macrocell flipflop can be individually programmed to implement D, T, JK, or SR operation with programmable clock control. The flipflop can be bypassed for combinatorial operation. During design entry, the designer specifies the desired flipflop type; the Altera software then selects the most efficient flipflop operation for each registered function to optimize resource utilization.

Each programmable register can be clocked in three different modes:

- Global clock signal. This mode achieves the fastest clock-to-output performance.
- Global clock signal enabled by an active-high clock enable. A clock enable is generated by a product term. This mode provides an enable on each flipflop while still achieving the fast clock-to-output performance of the global clock.
- Array clock implemented with a product term. In this mode, the flipflop can be clocked by signals from buried macrocells or I/O pins.

Two global clock signals are available in MAX 7000A devices. As shown in Figure 1, these global clock signals can be the true or the complement of either of the global clock pins, GCLK1 or GCLK2.

Each register also supports asynchronous preset and clear functions. As shown in Figure 2, the product-term select matrix allocates product terms to control these operations. Although the product-term-driven preset and clear from the register are active high, active-low control can be obtained by inverting the signal within the logic array. In addition, each register clear function can be individually driven by the active-low dedicated global clear pin (GCLRn). Upon power-up, each register in a MAX 7000AE device may be set to either a high or low state. This power-up state is specified at design entry. Upon power-up, each register in EPM7128A and EPM7256A devices are set to a low state.

All MAX 7000A I/O pins have a fast input path to a macrocell register. This dedicated path allows a signal to bypass the PIA and combinatorial logic and be clocked to an input D flipflop with an extremely fast (as low as 2.5 ns) input setup time.

## Expander Product Terms

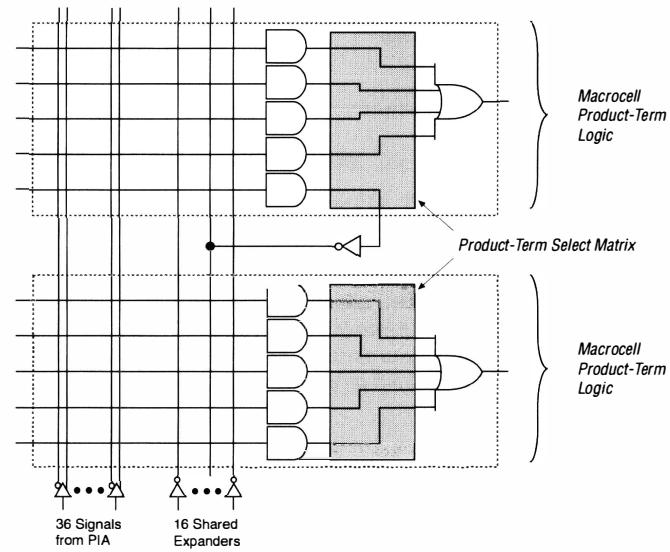
Although most logic functions can be implemented with the five product terms available in each macrocell, more complex logic functions require additional product terms. Another macrocell can be used to supply the required logic resources. However, the MAX 7000A architecture also offers both shareable and parallel expander product terms that provide additional product terms directly to any macrocell in the same LAB. These expanders help ensure that logic is synthesized with the fewest possible logic resources to obtain the fastest possible speed.

### Shareable Expanders

Each LAB has 16 shareable expanders that can be viewed as a pool of uncommitted single product terms (one from each macrocell) with inverted outputs that feed back into the logic array. Each shareable expander can be used and shared by any or all macrocells in the LAB to build complex logic functions. A small delay ( $t_{SEXP}$ ) is incurred when shareable expanders are used. Figure 3 shows how shareable expanders can feed multiple macrocells.

**Figure 3. MAX 7000A Shareable Expanders**

Shareable expanders can be shared by any or all macrocells in an LAB.



### Parallel Expanders

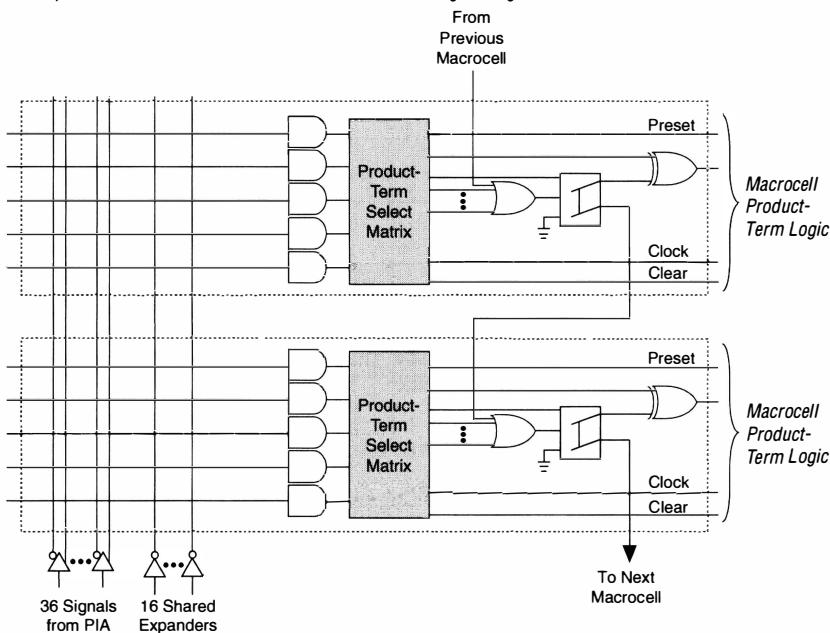
Parallel expanders are unused product terms that can be allocated to a neighboring macrocell to implement fast, complex logic functions. Parallel expanders allow up to 20 product terms to directly feed the macrocell OR logic, with five product terms provided by the macrocell and 15 parallel expanders provided by neighboring macrocells in the LAB.

The compiler can allocate up to three sets of up to five parallel expanders to the macrocells that require additional product terms. Each set of five parallel expanders incurs a small, incremental timing delay ( $t_{PEXP}$ ). For example, if a macrocell requires 14 product terms, the compiler uses the five dedicated product terms within the macrocell and allocates two sets of parallel expanders; the first set includes five product terms, and the second set includes four product terms, increasing the total delay by  $2 \times t_{PEXP}$ .

Two groups of eight macrocells within each LAB (e.g., macrocells 1 through 8 and 9 through 16) form two chains to lend or borrow parallel expanders. A macrocell borrows parallel expanders from lower-numbered macrocells. For example, macrocell 8 can borrow parallel expanders from macrocell 7, from macrocells 7 and 6, or from macrocells 7, 6, and 5. Within each group of eight, the lowest-numbered macrocell can only lend parallel expanders, and the highest-numbered macrocell can only borrow them. Figure 4 shows how parallel expanders can be borrowed from a neighboring macrocell.

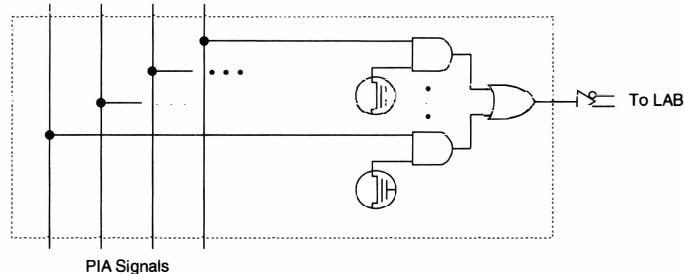
**Figure 4. MAX 7000A Parallel Expanders**

Unused product terms in a macrocell can be allocated to a neighboring macrocell.



### Programmable Interconnect Array

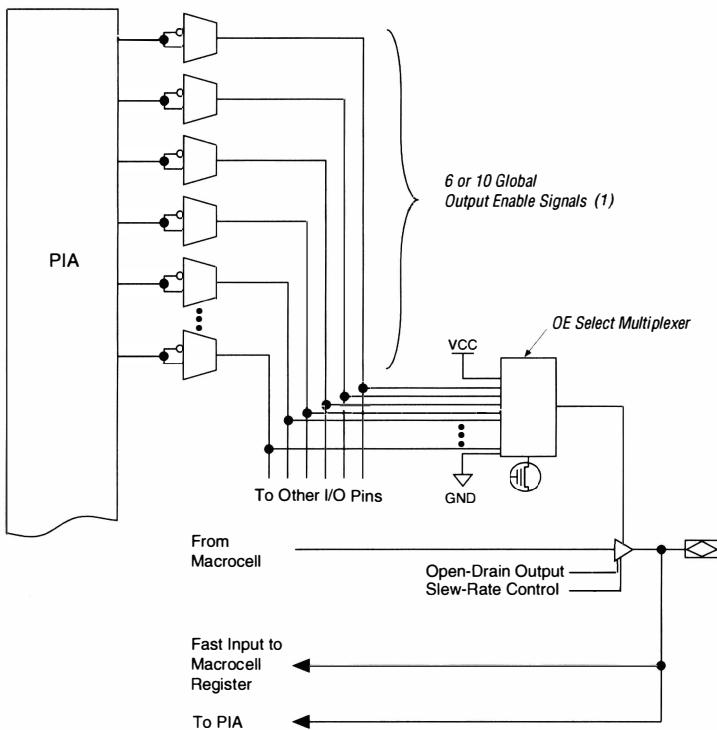
Logic is routed between LABs on the PIA. This global bus is a programmable path that connects any signal source to any destination on the device. All MAX 7000A dedicated inputs, I/O pins, and macrocell outputs feed the PIA, which makes the signals available throughout the entire device. Only the signals required by each LAB are actually routed from the PIA into the LAB. Figure 5 shows how the PIA signals are routed into the LAB. An EEPROM cell controls one input to a 2-input AND gate, which selects a PIA signal to drive into the LAB.

**Figure 5. MAX 7000A PIA Routing**

While the routing delays of channel-based routing schemes in masked or FPGAs are cumulative, variable, and path-dependent, the MAX 7000A PIA has a predictable delay. The PIA makes a design's timing performance easy to predict.

### I/O Control Blocks

The I/O control block allows each I/O pin to be individually configured for input, output, or bidirectional operation. All I/O pins have a tri-state buffer that is individually controlled by one of the global output enable signals or directly connected to ground or  $V_{CC}$ . Figure 6 shows the I/O control block for MAX 7000A devices. The I/O control block has 6 or 10 global output enable signals that are driven by the true or complement of two output enable signals, a subset of the I/O pins, or a subset of the I/O macrocells.

**Figure 6. I/O Control Block of MAX 7000A Devices****Note:**

- (1) EPM7032AE, EPM7064AE, EPM7128A, EPM7128AE, EPM7256A, and EPM7256AE devices have six output enable signals. EPM7512AE devices have 10 output enable signals.

When the tri-state buffer control is connected to ground, the output is tri-stated (high impedance) and the I/O pin can be used as a dedicated input. When the tri-state buffer control is connected to  $V_{CC}$ , the output is enabled.

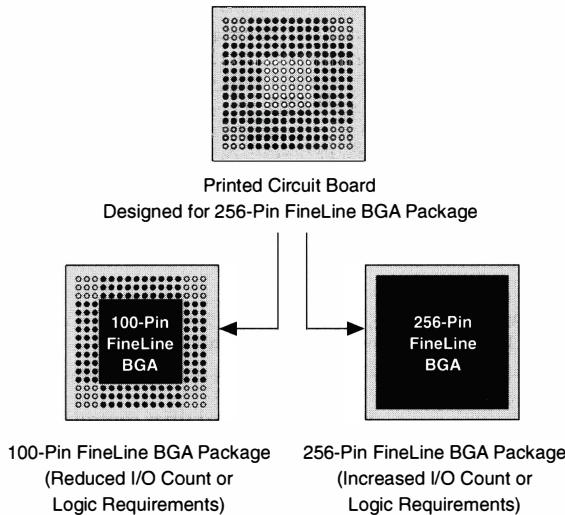
The MAX 7000A architecture provides dual I/O feedback, in which macrocell and pin feedbacks are independent. When an I/O pin is configured as an input, the associated macrocell can be used for buried logic.

## SameFrame Pin-Outs

MAX 7000A devices support the SameFrame pin-out feature for FineLine BGA packages. The SameFrame pin-out feature is the arrangement of balls on FineLine BGA packages such that the lower-ball-count packages form a subset of the higher-ball-count packages. SameFrame pin-outs provide the flexibility to migrate not only from device to device within the same package, but also from one package to another. A given printed circuit board (PCB) layout can support multiple device density/package combinations. For example, a single board layout can support a range of devices from an EPM7128AE device in a 100-pin FineLine BGA package to an EPM7512AE device in a 256-pin FineLine BGA package.

The Altera design software provides support to design PCBs with SameFrame pin-out devices. Devices can be defined for present and future use. The software generates pin-outs describing how to lay out a board to take advantage of this migration (see Figure 7).

**Figure 7. SameFrame Pin-Out Example**



## In-System Programmaticity

MAX 7000A devices can be programmed in-system via an industry-standard 4-pin IEEE Std. 1149.1 (JTAG) interface. ISP offers quick, efficient iterations during design development and debugging cycles. The MAX 7000A architecture internally generates the high programming voltages required to program EEPROM cells, allowing in-system programming with only a single 3.3-V power supply. During in-system programming, the I/O pins are tri-stated and weakly pulled-up to eliminate board conflicts. The pull-up value is nominally 50 kΩ.

MAX 7000AE devices have an enhanced ISP algorithm for faster programming. These devices also offer an ISP\_Done bit that provides safe operation when in-system programming is interrupted. This ISP\_Done bit, which is the last bit programmed, prevents all I/O pins from driving until the bit is programmed. This feature is only available in EPM7032AE, EPM7064AE, EPM7128AE, EPM7256AE, and EPM7512AE devices.

ISP simplifies the manufacturing flow by allowing devices to be mounted on a PCB with standard pick-and-place equipment before they are programmed. MAX 7000A devices can be programmed by downloading the information via in-circuit testers, embedded processors, the Altera MasterBlaster serial/USB communications cable, ByteBlasterMV parallel port download cable, and BitBlaster serial download cable. Programming the devices after they are placed on the board eliminates lead damage on high-pin-count packages (e.g., QFP packages) due to device handling. MAX 7000A devices can be reprogrammed after a system has already shipped to the field. For example, product upgrades can be performed in the field via software or modem.

In-system programming can be accomplished with either an adaptive or constant algorithm. An adaptive algorithm reads information from the unit and adapts subsequent programming steps to achieve the fastest possible programming time for that unit. A constant algorithm uses a pre-defined (non-adaptive) programming sequence that does not take advantage of adaptive algorithm programming time improvements. Some in-circuit testers cannot program using an adaptive algorithm. Therefore, a constant algorithm must be used. MAX 7000AE devices can be programmed with either an adaptive or constant (non-adaptive) algorithm. EPM7128A and EPM7256A device can only be programmed with an adaptive algorithm; users programming these two devices on platforms that cannot use an adaptive algorithm should use EPM7128AE and EPM7256AE devices.

The Jam Standard Test and Programming Language (STAPL), JEDEC standard JESD 71, can be used to program MAX 7000A devices with in-circuit testers, PCs, or embedded processors.

## Programming with External Hardware

For more information on using the Jam STAPL language, see *Application Note 88 (Using the Jam Language for ISP & ICR via an Embedded Processor)* and *Application Note 122 (Using Jam STAPL for ISP & ICR via an Embedded Processor)*.

ISP circuitry in MAX 7000AE devices is compliant with the IEEE Std. 1532 specification. The IEEE Std. 1532 is a standard developed to allow concurrent ISP between multiple PLD vendors.

MAX 7000A devices can be programmed on Windows-based PCs with an Altera Logic Programmer card, the MPU, and the appropriate device adapter. The MPU performs continuity checks to ensure adequate electrical contact between the adapter and the device.

For more information, see the *Altera Programming Hardware Data Sheet*.

The Altera software can use text- or waveform-format test vectors created with the Altera Text Editor or Waveform Editor to test the programmed device. For added design verification, designers can perform functional testing to compare the functional device behavior with the results of simulation.

Data I/O, BP Microsystems, and other programming hardware manufacturers provide programming support for Altera devices.

For more information, see *Programming Hardware Manufacturers*.

## IEEE Std. 1149.1 (JTAG) Boundary-Scan Support

MAX 7000A devices include the JTAG BST circuitry defined by IEEE Std. 1149.1. Table 5 describes the JTAG instructions supported by MAX 7000A devices. The pin-out tables, available from the Altera web site (<http://www.altera.com>), show the location of the JTAG control pins for each device. If the JTAG interface is not required, the JTAG pins are available as user I/O pins.



DS031-1 (v1.6) July 30, 2001

## Virtex-II 1.5V Field-Programmable Gate Arrays

### Advance Product Specification

#### Summary of Virtex®-II Features

- Industry First Platform FPGA Solution
- IP-Immersion™ Architecture
  - Densities from 40K to 10M system gates
  - 420 MHz internal clock speed (Advance Data)
  - 840+ Mb/s I/O (Advance Data)
- SelectRAM™ Memory Hierarchy
  - 3.5 Mb of True Dual-Port™ RAM in 18-Kbit block SelectRAM resources
  - Up to 1.9 Mb of distributed SelectRAM resources
  - High-performance interfaces to external memory
    - . DDR-SDRAM interface
    - . FCRAM interface
    - . QDR™-SRAM interface
    - . Sigma RAM interface
- Arithmetic Functions
  - Dedicated 18-bit x 18-bit multiplier blocks
  - Fast look-ahead carry logic chains
- Flexible Logic Resources
  - Up to 122,880 internal registers / latches with Clock Enable
  - Up to 122,880 look-up tables (LUTs) or cascadable 16-bit shift registers
  - Wide multiplexers and wide-input function support
  - Horizontal cascade chain and Sum-of-Products support
  - Internal 3-state bussing
- High-Performance Clock Management Circuitry
  - Up to 12 DCM (Digital Clock Manager) modules
    - . Precise clock de-skew
    - . Flexible frequency synthesis
    - . High-resolution phase shifting
  - 16 global clock multiplexer buffers
- Active Interconnect™ Technology
  - Fourth generation segmented routing structure
  - Predictable, fast routing delay, independent of fanout
- SelectI/O-Ultra™ Technology
  - Up to 1,108 user I/Os
  - 19 single-ended standards and six differential standards
  - Programmable sink current (2 mA to 24 mA) per I/O
- XCITE™ Digitally Controlled Impedance (DCI) I/O: on-chip termination resistors for single-ended I/O standards
- PCI-X @ 133 MHz, PCI @ 66 MHz and 33 MHz compliance
- Differential Signaling
  - 840 Mb/s Low-Voltage Differential Signaling I/O (LVDS) with current mode drivers
  - Bus LVDS I/O
  - Lightning Data Transport (LDT) I/O with current driver buffers
  - Low-Voltage Positive Emitter-Coupled Logic (LVPECL) I/O
  - Built-in DDR Input and Output registers
  - Proprietary high-performance SelectLink™ Technology
    - . High-bandwidth data path
    - . Double Data Rate (DDR) link
    - . Web-based HDL generation methodology
- Supported by Xilinx Foundation™ and Alliance™ Series Development Systems
  - Integrated VHDL and Verilog design flows
  - Compilation of 10M system gates designs
  - Internet Team Design (ITD) tool
- SRAM-Based In-System Configuration
  - Fast SelectMAP™ configuration
  - Triple Data Encryption Standard (DES) security option (Bitstream Encryption)
  - IEEE1532 support
  - Partial reconfiguration
  - Unlimited re-programmability
  - Readback capability
- Power-Down Mode
- 0.15 µm 8-Layer Metal process with 0.12 µm high-speed transistors
- 1.5 V ( $V_{CCINT}$ ) core power supply, dedicated 3.3 V  $V_{CCAUX}$  auxiliary and  $V_{CCO}$  I/O power supplies
- IEEE 1149.1 compatible boundary-scan logic support
- Flip-Chip and Wire-Bond Ball Grid Array (BGA) packages in three standard fine pitches (0.80mm, 1.00mm, and 1.27mm)
- 100% factory tested

© 2001 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and disclaimers are as listed at <http://www.xilinx.com/legal.htm>. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

DS031-1 (v1.6) July 30, 2001  
Advance Product Specification[www.xilinx.com](http://www.xilinx.com)  
1-800-255-7778Module 1 of 4  
1

Table 1: Virtex-II Field-Programmable Gate Array Family Members

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads <sup>(1)</sup>
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18-Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108
XC2V10000	10M	128 x 120	61,440	1,920	192	192	3,456	12	1,108

**Notes:**

- See details in Table 2, "Maximum Number of User I/O Pads".

## General Description

The Virtex-II family is a platform FPGA developed for high performance from low-density to high-density designs that are based on IP cores and customized modules. The family delivers complete solutions for telecommunication, wireless, networking, video, and DSP applications, including PCI, LVDS, and DDR interfaces.

The leading-edge 0.15µm / 0.12µm CMOS 8-layer metal process and the Virtex-II architecture are optimized for high speed with low power consumption. Combining a wide variety of flexible features and a large range of densities up to 10 million system gates, the Virtex-II family enhances programmable logic design capabilities and is a powerful alternative to mask-programmed gate arrays. As shown in Table 1, the Virtex-II family comprises 12 members, ranging from 40K to 10M system gates.

## Packaging

Offerings include ball grid array (BGA) packages with 0.80mm, 1.00mm, and 1.27mm pitches. In addition to traditional wire-bond interconnects, flip-chip interconnect is used in some of the BGA offerings. The use of flip-chip interconnect offers more I/Os than is possible in wire-bond versions of the similar packages. Flip-Chip construction offers the combination of high pin count with high thermal capacity.

Table 2 shows the maximum number of user I/Os available. The Virtex-II device/package combination table (Table 6 at the end of this section) details the maximum number of I/Os for each device and package using wire-bond or flip-chip technology.

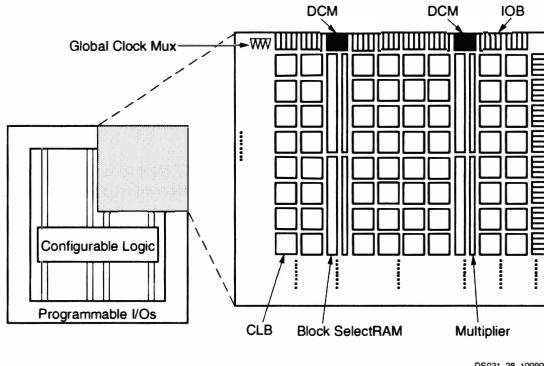
Table 2: Maximum Number of User I/O Pads

Device	Wire-Bond	Flip-Chip
XC2V40	88	
XC2V80	120	
XC2V250	200	
XC2V500	264	
XC2V1000	328	432
XC2V1500	392	528
XC2V2000	456	624
XC2V3000	516	720
XC2V4000		912
XC2V6000		1,104
XC2V8000		1,108
XC2V10000		1,108

## Architecture

### Virtex-II Array Overview

Virtex-II devices are user-programmable gate arrays with various configurable elements. The Virtex-II architecture is optimized for high-density and high-performance logic designs. As shown in Figure 1, the programmable device is comprised of input/output blocks (IOBs) and internal configurable logic blocks (CLBs).



*Figure 1: Virtex-II Architecture Overview*

DS031\_28\_100900

Programmable I/O blocks provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by the programmable IOBs.

The internal configurable logic includes four major elements organized in a regular array.

- Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.
- Block SelectRAM memory modules provide large 18-Kbit storage elements of True Dual-Port RAM.
- Multiplier blocks are 18-bit x 18-bit dedicated multipliers.
- DCM (Digital Clock Manager) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, coarse and fine-grained clock phase shifting, and electromagnetic interference (EMI) reduction.

A new generation of programmable routing resources called Active Interconnect Technology interconnects all of these elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general

routing matrix. The overall programmable interconnection is hierarchical and designed to support high-speed designs.

All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.

### Virtex-II Features

This section briefly describes Virtex-II features.

#### **Input/Output Blocks (IOBs)**

IOBs are programmable and can be categorized as follows:

- Input block with an optional single-data-rate or double-data-rate (DDR) register
- Output block with an optional single-data-rate or DDR register, and an optional 3-state buffer, to be driven directly or through a single or DDR register
- Bi-directional block (any combination of input and output configurations)

These registers are either edge-triggered D-type flip-flops or level-sensitive latches.

IOBs support the following single-ended I/O standards:

- LVTTL, LVCMSO (3.3 V, 2.5 V, 1.8 V, and 1.5 V)
- PCI-X at 133 MHz, PCI (3.3 V at 33 MHz and 66 MHz)
- GTL and GTLP

- HSTL (Class I, II, III, and IV)
- SSTL (3.3 V and 2.5 V, Class I and II)
- AGP-2X

The digitally controlled impedance (DCI) I/O feature automatically provides on-chip termination for each I/O element.

The IOB elements also support the following differential signaling I/O standards:

- LVDS
- BLVDS (Bus LVDS)
- ULVDS
- LDT
- LVPECL

Two adjacent pads are used for each differential pair. Two or four IOB blocks connect to one switch matrix to access the routing resources.

### **Configurable Logic Blocks (CLBs)**

CLB resources include four slices and two 3-state buffers. Each slice is equivalent and contains:

- Two function generators (F & G)
- Two storage elements
- Arithmetic logic gates
- Large multiplexers
- Wide function capability
- Fast carry look-ahead chain
- Horizontal cascade chain (OR gate)

The function generators F & G are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory.

In addition, the two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches.

Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.

### **Block SelectRAM Memory**

The block SelectRAM memory resources are 18 Kb of True Dual-Port RAM, programmable from 16K x 1 bit to 512 x 36 bits, in various depth and width configurations. Each port is totally synchronous and independent, offering three "read-during-write" modes. Block SelectRAM memory is cascadable to implement large embedded storage blocks. Supported memory configurations for dual-port and single-port modes are shown in Table 3.

**Table 3: Dual-Port And Single-Port Configurations**

16K x 1 bit	2K x 9 bits
8K x 2 bits	1K x 18 bits
4K x 4 bits	512 x 36 bits

A multiplier block is associated with each SelectRAM memory block. The multiplier block is a dedicated 18 x 18-bit multiplier and is optimized for operations based on the block SelectRAM content on one port. The 18 x 18 multiplier can be used independently of the block SelectRAM resource. Read/multiply/accumulate operations and DSP filter structures are extremely efficient.

Both the SelectRAM memory and the multiplier resource are connected to four switch matrices to access the general routing resources.

### **Global Clocking**

The DCM and global clock multiplexer buffers provide a complete solution for designing high-speed clocking schemes.

Up to 12 DCM blocks are available. To generate de-skewed internal or external clocks, each DCM can be used to eliminate clock distribution delay. The DCM also provides 90-, 180-, and 270-degree phase-shifted versions of its output clocks. Fine-grained phase shifting offers high-resolution phase adjustments in increments of 1/256 of the clock period. Very flexible frequency synthesis provides a clock output frequency equal to any M/D ratio of the input clock frequency, where M and D are two integers. For the exact timing parameters, see **Virtex™-II Electrical Characteristics**.

Virtex-II devices have 16 global clock MUX buffers, with up to eight clock nets per quadrant. Each global clock MUX buffer can select one of the two clock inputs and switch glitch-free from one clock to the other. Each DCM block is able to drive up to four of the 16 global clock MUX buffers.

### **Routing Resources**

The IOB, CLB, block SelectRAM, multiplier, and DCM elements all use the same interconnect scheme and the same access to the global routing matrix. Timing models are shared, greatly improving the predictability of the performance of high-speed designs.

There are a total of 16 global clock lines, with eight available per quadrant. In addition, 24 vertical and horizontal long lines per row or column as well as massive secondary and local routing resources provide fast interconnect. Virtex-II buffered interconnects are relatively unaffected by net fanout and the interconnect layout is designed to minimize crosstalk.

Horizontal and vertical routing resources for each row or column include:

- 24 long lines
- 120 hex lines
- 40 double lines
- 16 direct connect lines (total in all four directions)

### **Boundary Scan**

Boundary scan instructions and associated data registers support a standard methodology for accessing and configuring Virtex-II devices that complies with IEEE standards 1149.1 - 1993 and 1532. A system mode and a test mode are implemented. In system mode, a Virtex-II device performs its intended mission even while executing non-test boundary-scan instructions. In test mode, boundary-scan test instructions control the I/O pins for testing purposes. The Virtex-II Test Access Port (TAP) supports BYPASS, PRELOAD, SAMPLE, IDCODE, and USERCODE non-test instructions. The EXTEST, INTEST, and HIGHZ test instructions are also supported.

### **Configuration**

Virtex-II devices are configured by loading data into internal configuration memory, using the following five modes:

- Slave-serial mode
- Master-serial mode
- Slave SelectMAP mode
- Master SelectMAP mode
- Boundary-Scan mode (IEEE 1532)

A Data Encryption Standard (DES) decryptor is available on-chip to secure the bitstreams. One or two triple-DES key sets can be used to optionally encrypt the configuration information.

### **Readback and Integrated Logic Analyzer**

Configuration data stored in Virtex-II configuration memory can be read back for verification. Along with the configuration data, the contents of all flip-flops/latches, distributed SelectRAM, and block SelectRAM memory resources can be read back. This capability is useful for real-time debugging.

The Integrated Logic Analyzer (ILA) core and software provides a complete solution for accessing and verifying Virtex-II devices.

### **Power-Down Mode**

Activated by the power-down input, this mode reduces supply current and retains the Virtex-II device configuration.

### **Virtex-II Device/Package Combinations and Maximum I/O**

Wire-bond and flip-chip packages are available. Table 4 and Table 5 show the maximum possible number of user I/Os in wire-bond and flip-chip packages, respectively. Table 6 shows the number of available user I/Os for all device/package combinations.

- CS denotes wire-bond chip-scale ball grid array (BGA) (0.80 mm pitch).
- FG denotes wire-bond fine-pitch BGA (1.00 mm pitch).
- FF denotes flip-chip fine-pitch BGA (1.00 mm pitch).
- BG denotes standard BGA (1.27 mm pitch).
- BF denotes flip-chip BGA (1.27 mm pitch).

The number of I/Os per package include all user I/Os except the 15 control pins (CCLK, DONE, M0, M1, M2, PROG\_B, PWRDWN\_B, TCK, TDI, TDO, TMS, HSWAP\_EN, DXN, DXP, AND RSVD) and VATT.

**Table 4: Wire-Bond Packages Information**

Package	CS144	FG256	FG456	FG676	BG575	BG728
Pitch (mm)	0.80	1.00	1.00	1.00	1.27	1.27
Size (mm)	12 x 12	17 x 17	23 x 23	27 x 27	31 x 31	35 x 35
I/Os	92	172	324	484	408	516

**Table 5: Flip-Chip Packages Information**

Package	FF896	FF1152	FF1517	BF957
Pitch (mm)	1.00	1.00	1.00	1.27
Size (mm)	31 x 31	35 x 35	40 x 40	40 x 40
I/Os	624	824	1,108	684

Table 6: Virtex-II Device/Package Combinations and Maximum Number of Available I/Os (Advance Information)

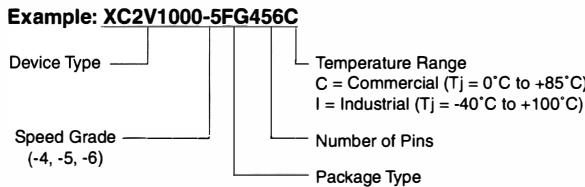
Package	Available I/Os											
	XC2V 40	XC2V 80	XC2V 250	XC2V 500	XC2V 1000	XC2V 1500	XC2V 2000	XC2V 3000	XC2V 4000	XC2V 6000	XC2V 8000	XC2V 10000
CS144	88	92	92									
FG256	88	120	172	172	172							
FG456			200	264	324							
FG676						392	456	484				
FF896					432	528	624					
FF1152								720	824	824	824	824
FF1517									912	1,104	1,108	1,108
BG575					328	392	408					
BG728							456	516				
BF957							624	684	684	684	684	684

**Notes:**

1. All devices in a particular package are pin-out (footprint) compatible. In addition, the FG456 and FG676 packages are compatible, as are the FF896 and FF1152 packages.

**Virtex-II Ordering Information**

Virtex-II ordering information is shown in Figure 2



DS031\_35\_033001

Figure 2: Virtex-II Ordering Information

## Slice Description

### Introduction

Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. As shown in Figure 13, each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element.

The output from the function generator in each slice drives both the slice output and the D input of the storage element. Figure 14 shows a more detailed view of a single slice.

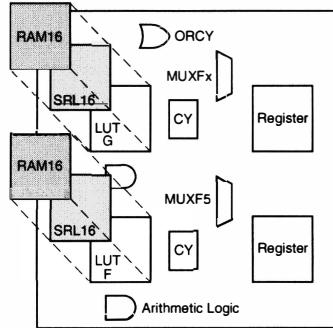


Figure 13: Virtex-II Slice Configuration

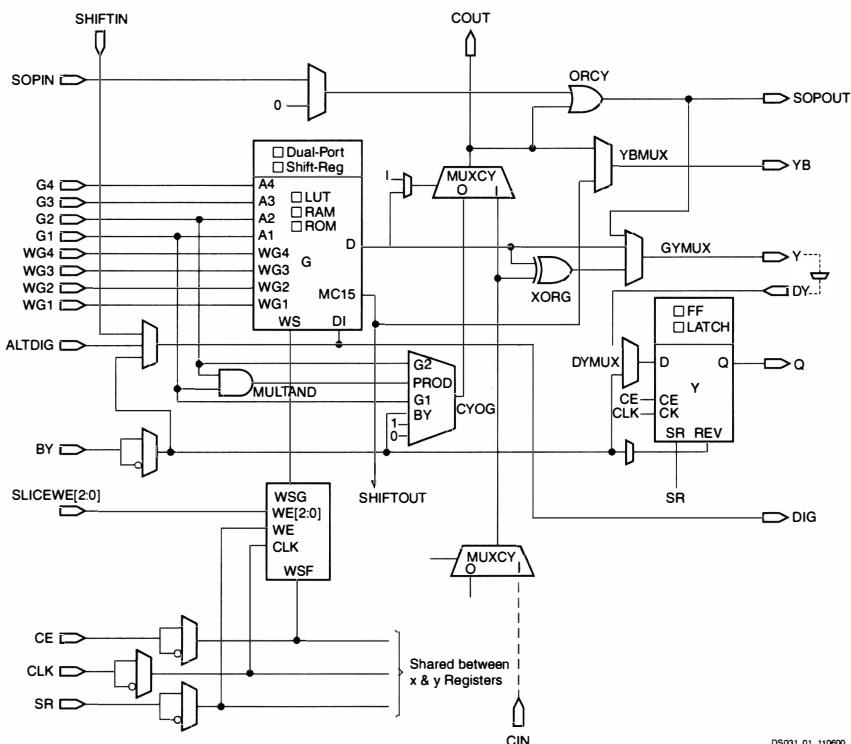


Figure 14: Virtex-II Slice (Top Half)

# Svar till övningsuppgifter

## 1 Inledning

### 1.3 Talsystem och koder

- 1.1** a) 20      b) 101      c) 273      d) 0,625      e) 0,4375      f) 0,15625
- 1.2** a) 26      b) 2572      c) 8383      d) 31      e) 124      f) 1073
- 1.3** a) 1010001      b) 10010110      c) 110010001  
d) 1010000101      e) 11100111110      f) 1010000111000
- 1.5** a) 0.1010      b) 0.0101      c) 0.0100
- 1.7** a) 14      b) 65      c) 111      d) 0.A      e) 0.7      f) 0.28
- 1.8** a) 24      b) 145      c) 421      d) 0.5      e) 0.34      f) 0.12
- 1.9** a) 110110011010.01      b) 1011000101011010.01111111001  
c) 1111100.11      d) 100110111.000001011
- 1.10** a) 170      b) 85      c) 51      d) 143      e) 76      f) 151  
g) 105      h) 126
- 1.11** a) 00011111      b) 00101011      c) 01100100  
d) 10001101      e) 11100001      f) 11111111
- 1.12** a) 02C1      b) 10F0      c) 7FFF      d) 8000
- 1.13** a) 0100      b) 00FF      c) 2000      d) 1FFF      e) FFFF
- 1.14** a) 0400      b) 0800      c) 1000      d) 2000      e) 4000      f) 8000  
g) 0200      h) 0C00      i) 1C00      j) 3C00      k) 7C00      l) FC00
- 1.15** a) 040000      b) 100000      c) 400000      d) 800000      e) F00000

**1.16 a)** 04000000**b)** 08000000**c)** 100000000**d)** 40000000**e)** C0000000**1.17a) 46b) CAc) 106d) EEd) 1000f) FFFE****g)** F100**1.18 a)** 22      **b)** F**c)** 6B**d)** E1**e)** EFFF**f)** 7001**g)** 1**1.19 a)** 1000 0001**b)** 0001 0101 0000**c)** 0100 0000 0001**d)** 0110 0100 0101**e)** 0001 1000 0101 0100**f)** 0101 0001 0111 0110**1.20**

Dec.	8	4	-2	-1
0	0	0	0	0
1	0	1	1	1
2	0	1	1	0
3	0	1	0	1
4	0	1	0	0
5	1	0	1	1
6	1	0	1	0
7	1	0	0	1
8	1	0	0	0
9	1	1	1	1

**1.21 a)** Kodordet för (9 - siffran, "9 minus siffran"), det s.k. 9-komplementet, erhålls lätt genom att man byter 0 mot 1 och tvärtom i kodordet för siffran. Exempel: Kodordet för siffran 1 är 0111 och kodordet för  $9 - 1 = 8$  blir enligt denna regel 1000, vilket enligt tabellen är kodordet för 8.

**1.21 b)** 00000111110000011111.., dvs 5 nollor, 5 ettor, 5 nollor, 5 ettor etc. Varje räkning 0 till 9 ger 5 nollor och 5 ettor, dvs. symmetri (jfr motsv. för BCD-koden, som ger 8 nollor och 2 ettor, dvs. osymmetri).

**1.21 c)** Kodorden innehåller minimalt antal ettor, som mest två ettor.

## 1.4 Realisering av en liten krets i VHDL - en introduktion till VHDL

### 1.22

--Complb  
--2001-07-04 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Complb is
    port(x, y: in std_logic;
          u: out std_logic);
end entity Complb;

architecture beteende of Complb is
begin
    u    <= (not x and not y) or (x and y);
end architecture beteende;
```

### 1.23

--HA  
--2001-07-04 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity HA is
    port(x, y: in std_logic;
          c_ut, s: out std_logic);
end entity HA;

architecture beteende of HA is
begin
    c_ut<= x and y;
    s    <= (not x and y) or (x and not y);

end architecture beteende;
```

### 1.24

--Nr\_ones  
--2001-07-04 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Nr_ones is
  port(x: in std_logic_vector(2 downto 0);
       u1, u0: out std_logic);
end entity Nr_ones;

architecture beteende of Nr_ones is
begin
  u1<= '1' when (x = "011") or
                (x = "101") or
                (x = "110") or
                (x = "111") else
                '0';
  u0  <=  (not x(2) and not x(1) and x(0)) or
          (not x(2) and x(1) and not x(0)) or
          (x(2) and not x(1) and not x(0)) or
          (x(2) and x(1) and x(0));
end architecture beteende;
```

## 2 Grindar, vippor, kombinations- och sekvenskretsar

### 2.1 Grindar och kombinationskretsar

**2.1 a)**  $y_1 = ab + c$

$$y_2 = ((ab)'c)'$$

$$y_3 = ((ab)'c')'$$

$$y_4 = ac + bc$$

$$y_5 = ((ac)'(bc)')'$$

$$y_6 = (a + b)c$$

$$y_7 = ((a + b)' + c)'$$

$$y_8 = ((a + b)' + c')'$$

$$y_9 = (a + c)(b + c)$$

$$y_{10} = ((a + c)' + (b + c)')'$$

**2.1 b)**

abc	$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$
000	0	1	0	0	0	0	0	0	0	0
001	1	0	1	0	0	0	0	0	1	1
010	0	1	0	0	0	0	1	0	0	0
011	1	0	1	1	1	1	0	1	1	1
100	0	1	0	0	0	0	1	0	0	0
101	1	0	1	1	1	1	0	1	1	1
110	1	1	1	0	0	0	1	0	1	1
111	1	1	1	1	1	1	0	1	1	1

**2.2**

a)  $y = (a + b) \oplus a$   
b)

**2.3**

a)  $y = ((b \oplus a)a)' \oplus (b \oplus a)$   
b)

**2.4**

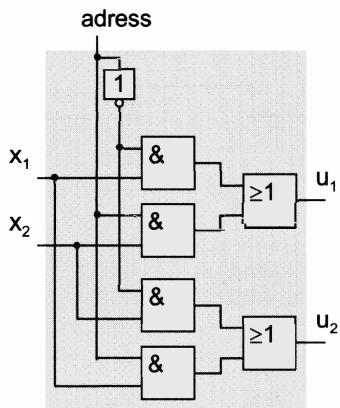
a)  $y = (a \oplus b) \oplus c$   
b)

ab	y
00	0
01	1
10	0
11	0

ab	y
00	1
01	0
10	1
11	1

abc	y
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

c) Utsignalen  $y = 1$  om och endast om ett udda antal av insignalerna har värdet 1.

**2.5****2.7 a)**

$$\begin{aligned} \text{data\_ut} = & a_1' a_0' \text{data\_in}(0) + \\ & a_1' a_0 \text{data\_in}(1) + \\ & a_1 a_0' \text{data\_in}(2) + \\ & a_1 a_0 \text{data\_in}(3) \end{aligned}$$

**2.7 b)**

$$\begin{aligned} \text{data\_ut}(0) &= a_1' a_0' \text{data\_in}(0) \\ \text{data\_ut}(1) &= a_1' a_0 \text{data\_in}(0) \\ \text{data\_ut}(2) &= a_1 a_0' \text{data\_in}(0) \\ \text{data\_ut}(3) &= a_1 a_0 \text{data\_in}(0) \end{aligned}$$

**2.8 a)** 0, 2, 1, 3**b)** 0, 2, 1, 3

<b>2.9 2.1) y:</b>	1	2	3	4	5	6	7	8	9	10
<b>ns</b>	2,7	1,6	1,6	2,7	1,6	2,7	1,8	1,8	2,7	1,8

**2.2) 3,2 ns    2.3) 4,6 ns    2.4) 3,8 ns****2.6**

- a)  $ab + cd$     b)  $((ab)'(cd)')'$   
c)

abcd	x	y
0000	0	0
0001	0	0
0010	0	0
0011	1	1
0100	0	0
0101	0	0
0110	0	0
0111	1	1
1000	0	0
1001	0	0
1010	0	0
1011	1	1
1100	1	1
1101	1	1
1110	1	1
1111	1	1

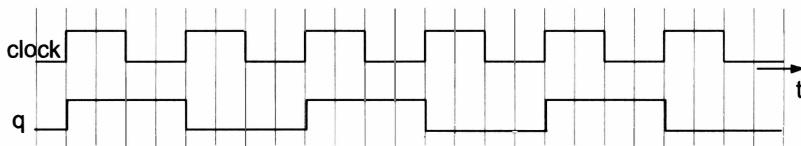
**2.6**

d) Sanningstabellerna är lika. I ett OCH-ELLER-nät kan samtliga grindar bytas mot NAND-grindar, utan att sanningstabellen ändras, och tvärtom så kan i ett NAND-NAND-nät NAND-grindarna i första nivån bytas mot OCH-grindar och NAND-grinden i andra nivån bytas mot en ELLER-grind, utan att sanningstabellen ändras.

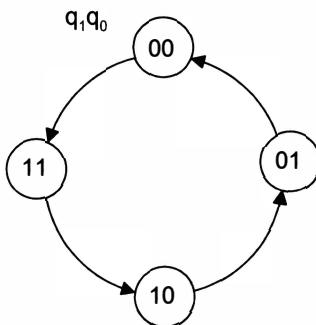
För logiska uttrycken gäller alltså  $ab + cd = ((ab)'(cd)')'$

## 2.2 Vippor och sekvenskretsar

2.10



2.11 a)



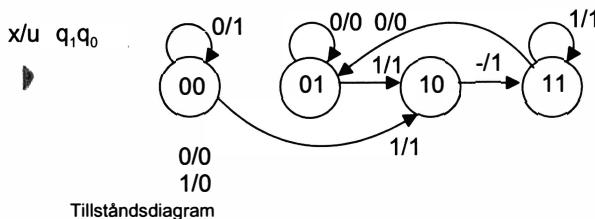
b) Binärräknare 2 bitar, som räknar baklänges.

2.12

x :	0 0 1 0 1 1 0 0 1 1 1 0 0 0 1 0 1 1 1 1 0 0
u :	0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0
$q_1$ :	0 0 0 0 0 0 1 1 0 0 1 1 1 0 0 0 0 0 1 1 1 1 0
$q_0$ :	0 0 0 1 0 1 1 0 0 1 1 1 0 0 1 0 1 1 1 1 0 0

Moore

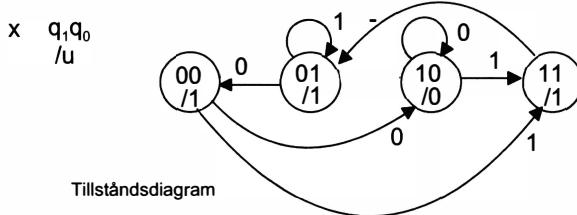
## 2.13 Mealy



Nuvarande tillstånd $q_1q_0$	Nästa tillstånd / Utsignal $u$ $q_1^+q_0^+/u$	
	x	
	0	1
00	00/1	10/1
01	01/0	10/1
10	11/1	11/1
11	01/0	11/1

Tillståndstabell

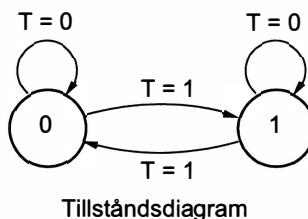
## 2.14 Moore



Nuvarande tillstånd $q_1q_0$	Nästa tillstånd $q_1^+q_0^+$		Utsignal $u$	
	Insignal x			
	0	1		
00	10	11	1	
01	00	01	1	
10	10	11	0	
11	01	01	1	

Tillståndstabell

**2.15 a)**



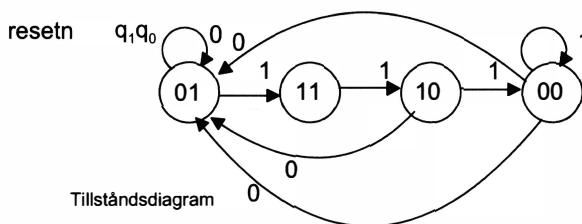
**b) Moore**

**c)**  $T = 0$  ger ingen tillståndsändring, "står kvar", medan  $T = 1$  ger tillståndsändring, "slår om".

**d)**

0	0	0	1	1	1	1	0	0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0	1
0	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1

**2.16**



Nuvarande tillstånd $q_1q_0$	Nästa tillstånd $q_1^+q_0^+$	
	Insignal resetn	
	0	1
00	01	00
01	01	11
10	01	00
11	01	10

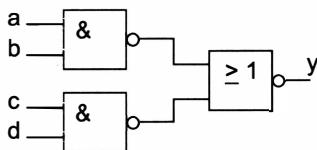
Tillståndstabell

### 3 Boolesk algebra

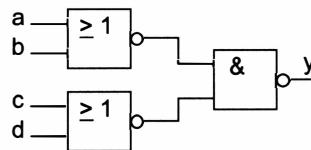
- 3.1**
- |                 |                  |        |               |            |               |
|-----------------|------------------|--------|---------------|------------|---------------|
| a) $ad$         | b) $ad$          | c) $a$ | d) $ab' + ac$ | e) $1$     | f) $a + b$    |
| g) $ac$         | h) $a'd + bc'd'$ |        | i) $ab$       | j) $c'$    | k) $a'c + bc$ |
| l) $ae + bc'e'$ | m) $a'b + cd$    |        | n) $ab'c$     | o) $a + b$ | p) $a'$       |
| q) $ac$         | r) $1$           |        |               |            |               |

**3.2**

$y_1 = ab + c$	$y_2 = ab + c'$	$y_3 = ab + c$	$y_4 = ac + bc$	$y_5 = ac + bc$
$y_6 = ac + bc$	$y_7 = ac' + bc'$	$y_8 = ac + bc$	$y_9 = ab + c$	$y_{10} = ab + c$

**3.4 a)**

$$y = ((ab)' + (cd)')' = abcd$$

**b)**

$$y = ((a+b)'(c+d)')' = a+b+c+d$$

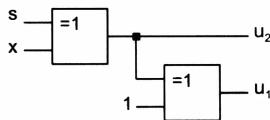
- 3.5** ICKE:  $B = 0, C = 0 \Rightarrow Y = A'$       NAND:  $C = 1 \Rightarrow Y = A' + B' = (AB)'$   
 ELLER:  $C = 0 \Rightarrow Y = A' + B$  (invertera A enl. ovan)  
 XOR:  $A = 1 \Rightarrow Y = BC' + B'C = B \oplus C$     OCH:  $Y = ((AB)')'$

**3.7**  $Y = A \oplus AB = A(1 \oplus B) = AB'$

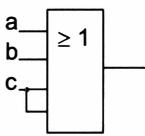
OCH:  $Y = A(B')'$

ICKE:  $A = 1 \Rightarrow Y = B'$

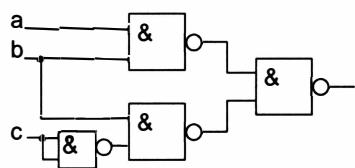
ELLER:  $Y = (A'B')'$

**3.8**

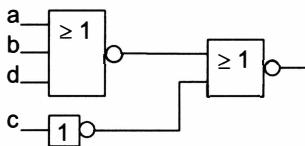
3.9



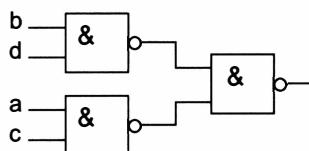
3.10



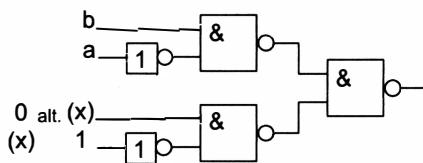
3.11



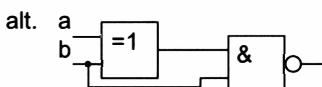
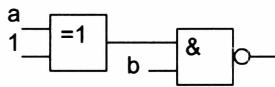
3.12



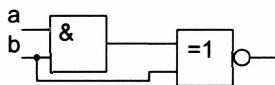
3.13



3.14



alt.



# 4 Kombinationskretsar

## 4.2 Booleska funktioner

4.1 a)  $\Sigma(1, 2, 3, 4, 5, 6), \Pi(0, 7)$

b)  $\Sigma(1, 3, 6, 7, 9, 11, 14, 15), \Pi(0, 2, 4, 5, 8, 10, 12, 13)$

c)  $\Sigma(1, 4, 5, 6, 7), \Pi(0, 2, 3)$

4.2 P.g.a. det angivna villkoret kan endast hälften  $2^n/2 = 2^{n-1}$  av samtliga  $2^n$  funktionsvärden väljas fritt.

## 4.3 Förenkling och realisering av booleska funktioner i grindnät

4.3 a)  $f = x_1'x_0' + x_2'x_0'$

$$f' = x_0 + x_2x_1$$

b)  $f = x_3'x_1 + x_3x_1'$

$$f' = x_3'x_1' + x_3x_1$$

c)  $f = x_3'x_1 + x_3x_1' + x_0'$

$$f' = x_3'x_1'x_0 + x_3x_1x_0$$

d)  $f = x_3'x_2'x_1'x_0 + x_3x_2x_1'x_0 + x_3'x_2x_0' + x_3'x_2x_1$

$$f' = x_2'x_0' + x_2'x_1 + x_3x_0' + x_3x_1 + x_3x_2' + x_3'x_2x_1'x_0$$

e)  $f = x_3'x_2'x_0 + x_3x_2x_0 + x_3'x_1x_0$  (alt.  $x_2x_1x_0$ )

$$f' = x_0' + x_3x_2' + x_3'x_2x_1'$$

f)  $f = x_4'x_3x_2 + x_4x_2'x_0' + x_3x_2'x_0$

$$f' = x_4'x_2'x_0' + x_4x_2 + x_3'x_0 + x_4'x_3' \text{ (alt. } x_3'x_2\text{)}$$

g)  $f = x_4'x_3x_2x_1 + x_4x_2'x_0' + x_4x_3x_2' + x_3x_2'x_1' + x_4'x_3x_2x_0$  (alt.  $x_4'x_3x_1'x_0$ )

$$f' = x_4'x_3' + x_4'x_2'x_1 + x_4x_2 + x_3'x_0 + x_2x_1'x_0'$$

h)  $f = x_5'x_4'x_3x_2' + x_5'x_4x_3'x_0 + x_5x_4'x_3x_2 + x_5x_4x_3x_2' + x_3x_2'x_0$

$$f' = x_4'x_3' + x_5'x_4x_0' + x_5x_4'x_2'x_0' + x_5x_3' + (x_5'x_4'x_2 + x_4x_3x_2)$$

(alt.  $x_5'x_4'x_2 + x_4x_3x_2$ ) (alt.  $x_5'x_3x_2 + x_4x_3x_2$ ) (alt.  $x_5'x_3x_2 + x_5x_4x_2$ )

**4.4 a)**  $f = x_1x_0 + x_3'x_2x_0$

$$f' = x_0' + x_2'x_1' + x_3x_1' \text{ (alt. } x_3x_2\text{)}$$

**b)**  $f = x_3'x_0 + x_2x_1'$ , alt.  $= x_3'x_0 + x_3'x_2$ , alt.  $= x_1'x_0 + x_2x_1'$   
alt.  $= x_1'x_0 + x_3'x_2$

$$f' = x_2'x_0' + x_1 \text{ (alt. } x_3\text{)}$$

**c)**  $f = x_4'x_3'x_0 + x_4'x_2'x_1 + x_4x_3x_2x_0 + x_3x_2x_1'x_0'$

$$f' = x_3'x_0' + x_1x_0' + x_2'x_1' + x_4x_2' + x_4x_3' + x_4'x_3x_2x_0$$

**d)**  $f = x_1'x_0 + x_2'x_0 + x_3x_0 + x_3x_2'x_1'$

$$f' = x_3'x_0' + x_1x_0' + x_2x_0' + x_3'x_2x_1$$

**e)**  $f = x_3x_0' + x_3'x_2x_1' + x_2x_1x_0$

$$f' = x_2' + x_3'x_1x_0' + x_3x_1'x_0$$

**f)**  $f = x_4'x_3'x_2' + x_4x_2'x_1 + x_4x_1x_0 + x_2x_1'x_0' + x_3'x_2x_1' + x_2'x_1x_0$  (alt.  $x_3x_1x_0$ )

$$f' = x_4'x_2x_1 + x_4'x_3x_0' + x_4x_2'x_1' + x_2x_1x_0' + x_3x_1'x_0$$

**4.5**  $u_5 = x_2x_1$        $u_5' = x_2' + x_1'$

$$u_4 = x_2x_1' + x_2x_0$$
       $u_4' = x_2' + x_1x_0'$

$$u_3 = x_2'x_1x_0 + x_2x_1'x_0$$
       $u_3' = x_0' + x_2'x_1' + x_2x_1$

$$u_2 = x_1x_0'$$
       $u_2' = x_1' + x_0$

$$u_1 = 0$$
       $u_1' = 1$

$$u_0 = x_0$$
       $u_0' = x_0'$

**4.6**  $z_4 = x_2x_1 + x_3$

$$z_3 = x_2x_1' + x_3 + x_2'x_1x_0$$

$$z_2 = x_2x_1' + x_2x_0 + x_2'x_1x_0'$$

$$z_1 = x_1'x_0 + x_1x_0'$$

$$z_0 = x_0$$

**4.7**  $d_{13} = x_3x_0$

$$d_{12} = x_3x_0' + x_2x_1x_0$$

$$d_{11} = x_3x_0' + x_2x_1'x_0 + x_2x_1x_0'$$

$$d_{10} = d_{01} = x_2x_0'$$

$$d_{03} = x_1x_0$$

$$d_{02} = x_3x_0' + x_2x_1' + x_1x_0'$$

$$d_{00} = x_0$$

**4.8**  $u_2 = x_4$

$$u_1 = x_4' x_2$$

$$u_0 = x_4' x_3 + x_2' x_1$$

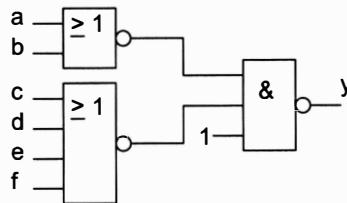
**4.9**  $u_2 = x_4' + x_5', \quad u_1 = x_5 x_4 x_3' + x_5 x_4 x_2', \quad u_0 = x_5' + x_4 x_3' + x_4 x_2 x_1'$

**4.10**  $z_5 = x_3 + x_2 x_1 x_0, \quad z_4 = x_2 x_0' + x_2 x_1', \quad z_3 = x_3 + x_2' x_1 + x_1 x_0' + x_2 x_1' x_0$

$$z_2 = x_2 x_0' + x_2' x_0, \quad z_1 = x_1, \quad z_0 = x_0$$

**4.11**  $f(x, y, z) = z(xy)' + z'xy = z(zxy)' + (zxy)'xy = ((z(zxy)')'((zxy)'xy)')$

**4.13**



**4.14 a)**  $f'(w, x, y, z) = w'xy'z + w'xyz' + wx'y'z + wx'yz'$

$$= w'x(y'z + yz') + wx'(y'z + yz')$$

$$= w'x(y \oplus z) + wx'(y \oplus z)$$

$$= (y \oplus z)(w'x + wx') = (y \oplus z)(w \oplus x)$$

$$\Rightarrow f = ((y \oplus z)(w \oplus x))'$$

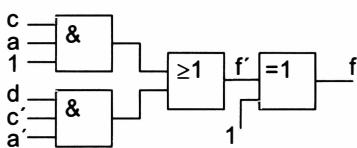
**b)**  $f(x, y, z) = \Sigma(4, 7) = xy'z' + xyz = x(y'z' + yz) = x(y \oplus z)'$

$$= (x' + (y \oplus z))' = ((1 \oplus x) + (y \oplus z))'$$

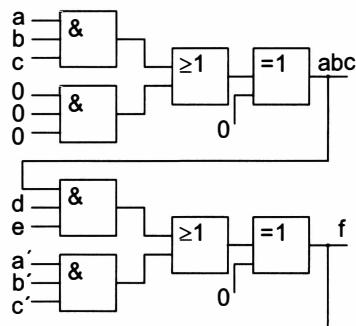
**4.15**  $u = z_1 y_1' y_0 x_1' x_0' + z_1 z_0 y_1 y_0' x_1'$

**4.16**

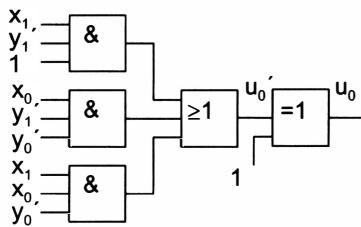
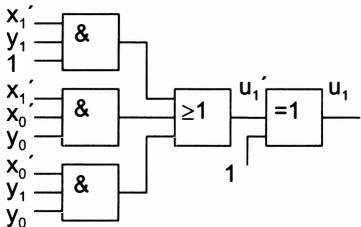
a)



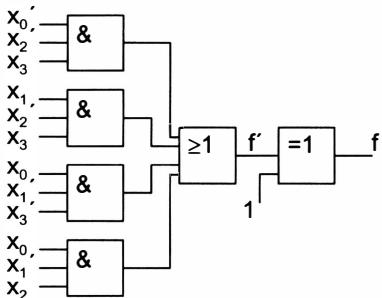
b)



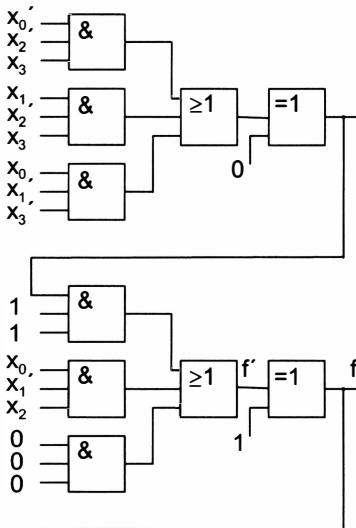
**4.17**



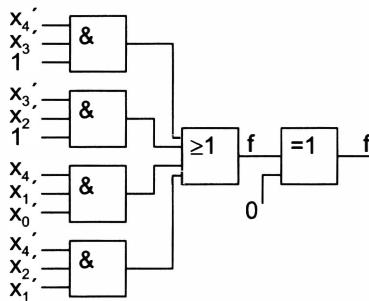
**4.18 a)**



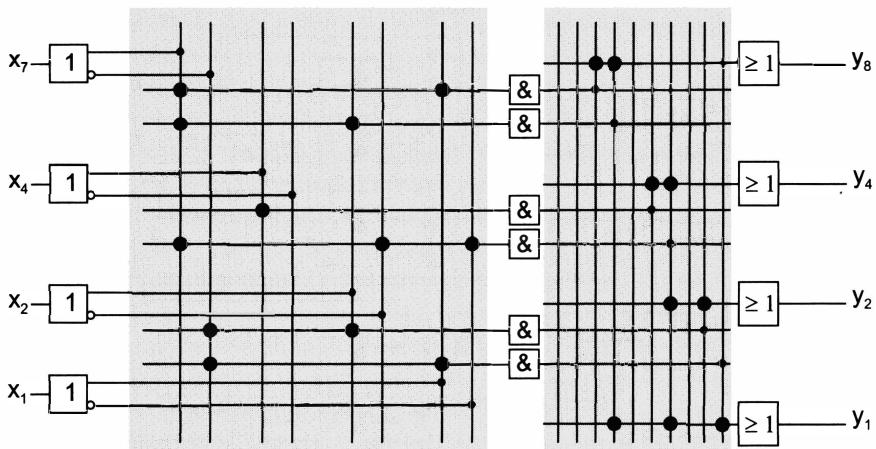
b)



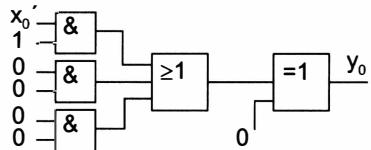
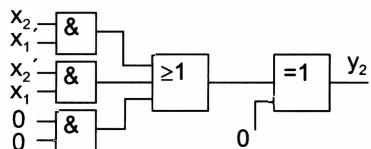
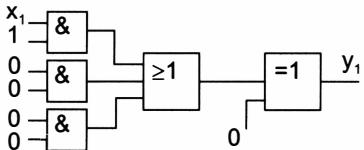
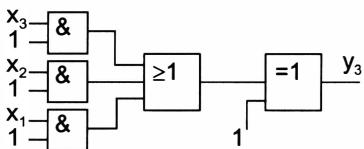
4.19



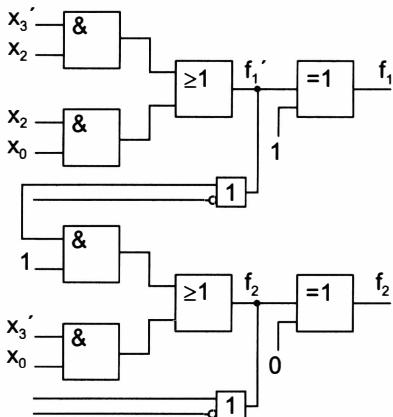
4.20



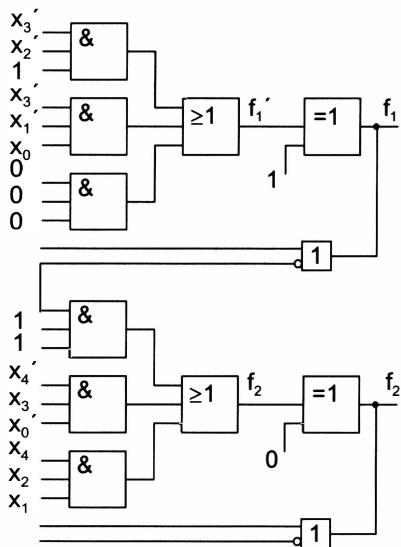
**4.21**



**4.22**



**4.23**



## 4.5 Kombinationskretsar i tidsplanet

**4.24 a)** Statisk 0-hasard:  $c = 0, b = d = 1$  och  $a: 0 \rightarrow 1 \rightarrow 0$   
 $a = 1, c = d = 0$  och  $b: 0 \rightarrow 1 \rightarrow 0$

Statisk 1-hasard:  $a = b = c = 1$  och  $d: 1 \rightarrow 0 \rightarrow 1$

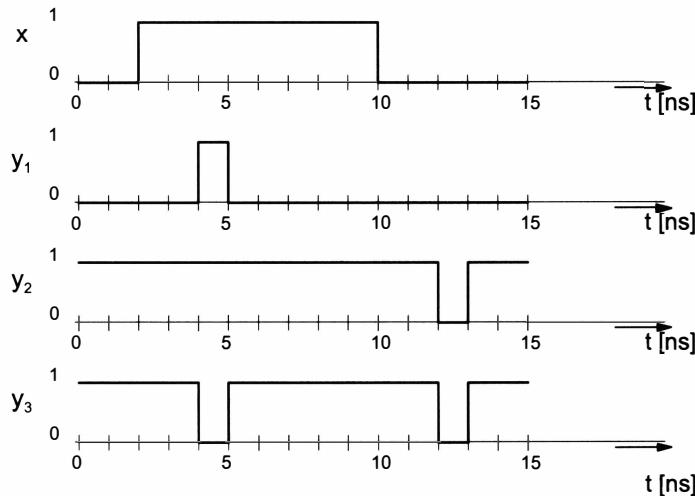
Hasardfritt två-nivå-nät:  $f = ab'd + acd + a'b'd' + bcd' + abc$

**b)** Statisk 0-hasard:  $a = b = c = 0$  och  $d: 0 \rightarrow 1 \rightarrow 0$

Statisk 1-hasard:  $a = 0, c = d = 1$  och  $b: 1 \rightarrow 0 \rightarrow 1$   
 $b = d = 0, c = 1$  och  $a: 1 \rightarrow 0 \rightarrow 1$

Hasardfritt två-nivå-nät:  $f = ad' + ab + bd + a'b'c + a'cd + b'cd'$

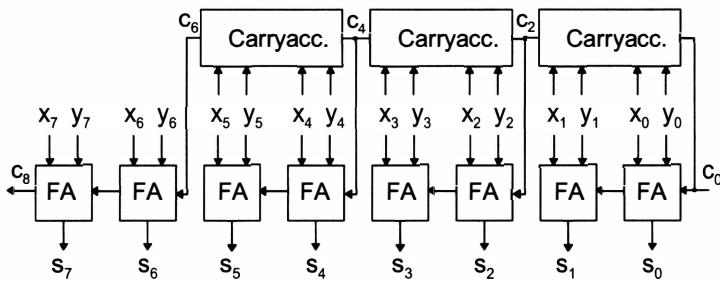
**4.25**



## 4.6 Adderare

**4.26**  $s_i = x_i'y_i'c_i + x_i'y_ic_i' + x_iy_i'c_i' + x_iy_ic_i = x_i'(y_i'c_i + y_ic_i') + x_i(y_i'c_i' + y_ic_i) = x_i'(y_i \oplus c_i) + x_i(y_i \oplus c_i)' = x_i \oplus y_i \oplus c_i$

**4.27 a)**



- b)  $c_{i+2} = x_{i+1}y_{i+1} + y_{i+1}x_iy_i + x_{i+1}x_iy_i + c_iy_{i+1}y_i + c_ix_{i+1}y_i + c_iy_{i+1}x_i + c_ix_{i+1}x_i$   
 c)  $c_0 \rightarrow c_2 \rightarrow c_4 \rightarrow c_6 \rightarrow c_7 \rightarrow s_7 = 2 + 2 + 2 + 2 + 3 = 11$  grindnivåer

## 4.7 Aritmetisk logisk enhet (ALU)

**4.28**

$s_{ut}$	Z	C	Of	PE	PO	S
8E	0	1	0	1	0	1
10	0	1	0	0	1	0
F0	0	0	0	1	0	1
00	1	1	0	1	0	0
86	0	0	1	0	1	1 talområdet överskrids
7A	0	1	1	0	1	0 talområdet överskrids
10	0	1	0	0	1	0
8E	0	1	0	1	0	1
72	0	0	0	1	0	0
00	1	1	0	1	0	0
86	0	0	1	0	1	1 talområdet överskrids
7A	0	1	1	0	1	0 talområdet överskrids

- 4.29 a)** OCH 10111110    **b)** ELLER 10001010    **c)** XOR 10110110

**4.30**

a)  $y_4 = x_5x_4'x_3'x_2'x_1'$      $y_{21} = x_5'x_2 + x_2x_1' + x_5x_2'x_1$      $y_0 = x_1$   
 $y_3 = x_5'x_4 + x_5x_4'x_3 + x_5x_4'x_1 + x_5x_4'x_2 + x_4x_3'x_2'x_1'$   
 $y_2 = x_5'x_3 + x_3x_2'x_1' + x_5x_3'x_1 + x_5x_3'x_2$

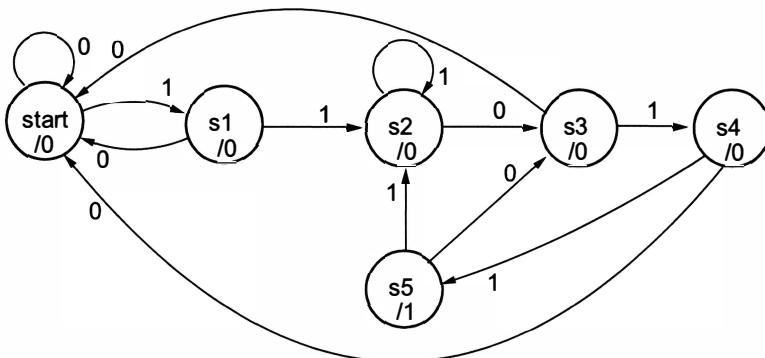
- b)** Invertera MSB.    **c)** Invertera MSB.

# 5 Sekvenskretsar

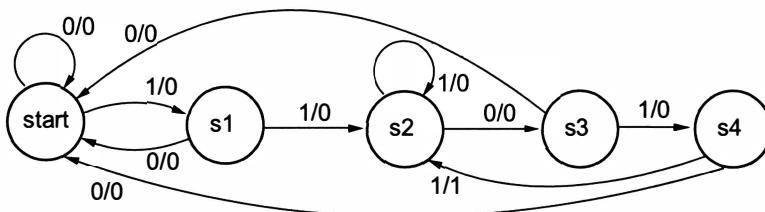
## 5.1 Genrella sekvenskretsar

5.1

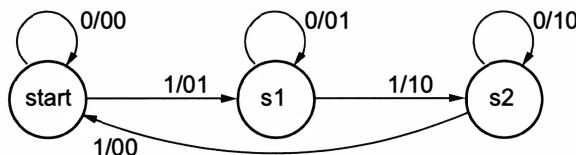
a)



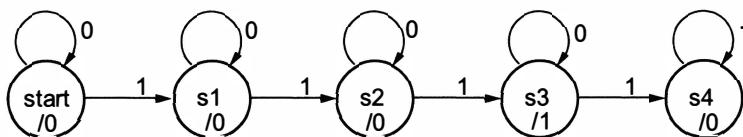
b)



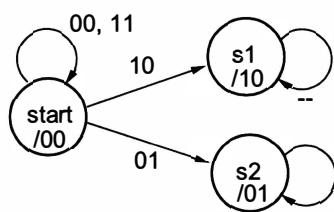
c)



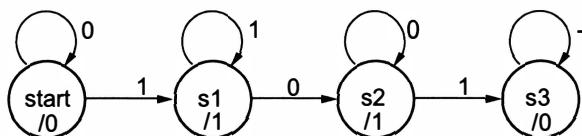
d)



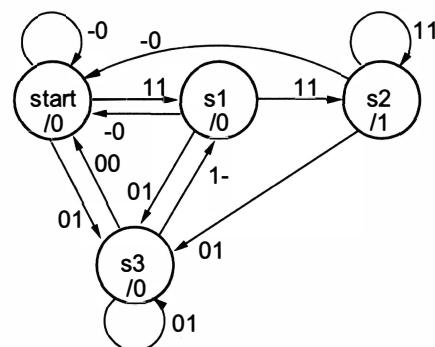
e)



f)

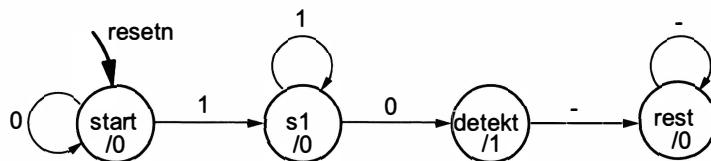


g)



## 5.2

a)

Kodning

Binär	00	01	10	11
Gray	00	01	11	10
"One-hot"	0001	0010	0100	1000

b)

Kodning *binary*

$$q_1^+ = q_1 + q_0 x' \quad q_0^+ = q_1 + x \quad u = q_1 q_0'$$

Kodning *Gray*

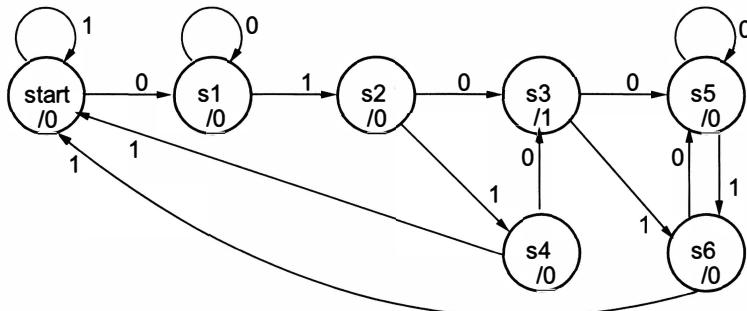
$$q_1^+ = q_1 + q_0 x' \quad q_0^+ = q_1' q_0 + q_1 x \quad u = q_1 q_0$$

Kodning *one-hot*

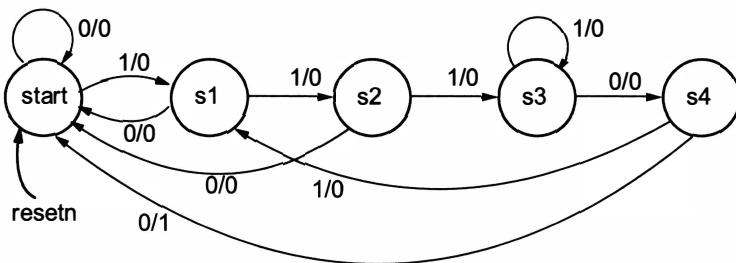
$$q_3^+ = q_3 + q_2 \quad q_2^+ = q_1 x' \quad u = q_2$$

$$q_1^+ = q_1 x + q_0 x' \quad q_0^+ = q_0 x'$$

## 5.3



5.4



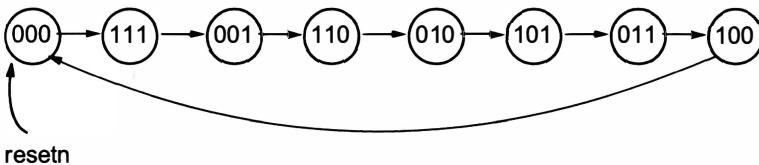
$$q_2^+ = q_1 q_0 x'$$

$$q_1^+ = q_1 x + q_0 x$$

$$q_0^+ = q_1 x + q_0' x$$

$$u = q_2 x'$$

5.5

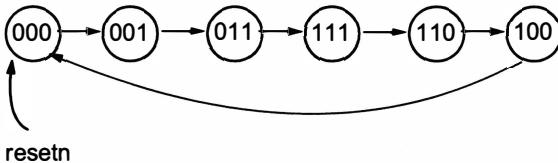


$$q_2^+ = q_2'$$

$$q_1^+ = q_2' q_1' + q_1' q_0 + q_2 q_1 q_0'$$

$$q_0^+ = q_2' q_0' + q_2 q_0$$

5.6

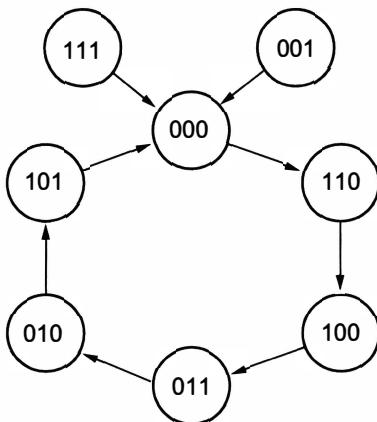


$$q_2^+ = q_1 \text{enable} + q_2 \text{enable}'$$

$$q_1^+ = q_0 \text{enable} + q_1 \text{enable}'$$

$$q_0^+ = q_2' \text{enable} + q_0 \text{enable}'$$

## 5.7a)



$$q_2^+ = q_2' q_0' + q_1 q_0 \quad q_1^+ = q_1' q_0' + q_2' q_1 q_0 \quad q_0^+ = q_2 q_1' q_0' + q_2' q_1 q_0$$

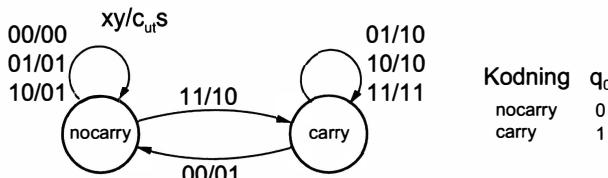
**b)**

$$q_{2se}^+ = q_2^+ \text{enable} + q_2 \text{enable}' + \text{set6}$$

$$q_{1se}^+ = q_1^+ \text{enable} + q_1 \text{enable}' + \text{set6}$$

$$q_{0se}^+ = q_0^+ \text{enable set6}' + q_0 \text{enable}' \text{set6}, \text{ där } q_2^+, q_1^+ \text{ och } q_0^+ \text{ är enligt a)}$$

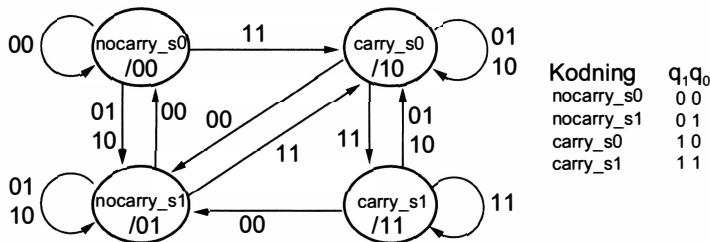
## 5.8

**a)**

$$q_0^+ = xy + q_0y + q_0x$$

$$c_{ut} = xy + q_0y + q_0x \quad s = q_0'x'y + q_0'xy' + q_0x'y' + q_0xy = q_0 \oplus x \oplus y$$

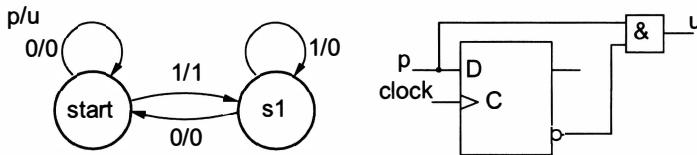
b)



$$q_1^+ = xy + q_1y + q_1x \quad q_0^+ = q_1'x'y' + q_1'xy' + q_1x'y + q_1xy = q_1 \oplus x \oplus y$$

$$c_{ut} = q_1 \quad s = q_0$$

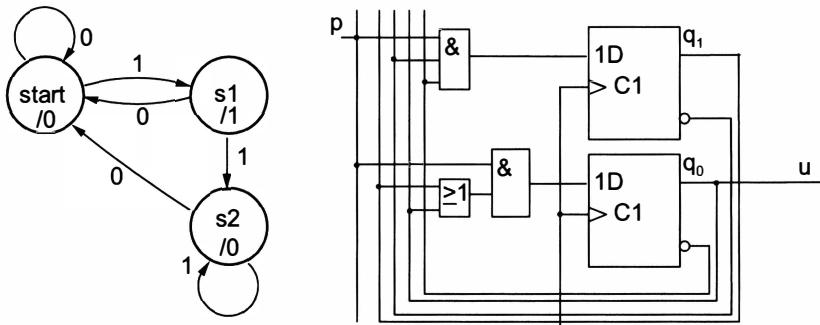
5.9a)



Kodning: start = 0, s1 = 1

$$q_0^+ = p \quad u = q_0'p$$

b)



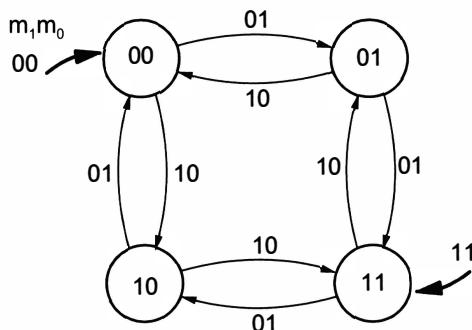
Kodning: start = 00, s1 = 01, s2 = 10

$$q_1^+ = q_1p + q_0p = (q_1 + q_0)p \quad q_0^+ = q_1'q_0'p$$

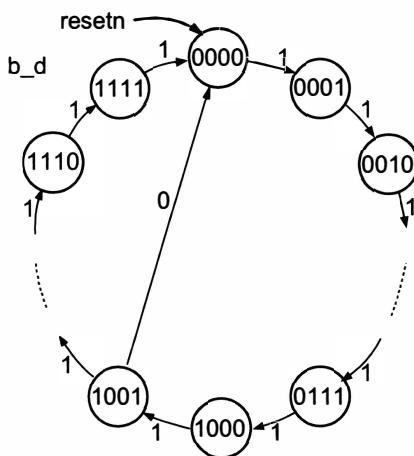
$$u = q_0$$

**5.10**

a)  $d_2 = q_2 q_0 + q_1 q_0'$        $d_1 = q_2' q_0 + q_1 q_0'$        $d_0 = q_2' q_1' + q_2 q_1$   
 b)  $T_2 = q_2' q_1 q_0' + q_2 q_1' q_0'$        $T_1 = q_2' q_1' q_0 + q_2 q_1 q_0$   
 $T_0 = q_2' q_1' q_0' + q_2' q_1 q_0 + q_2 q_1' q_0 + q_2 q_1 q_0$

**5.11**

$$d_1 = q_0 m_0 + q_0' m_1 \quad d_0 = q_1' m_0 + q_1 m_1$$

**5.12**

$$T_3 = q_3 q_0 b_{-}d' \text{resetn} + q_2 q_1 q_0 \text{resetn} + \text{resetn}' q_3 \quad T_2 = q_1 q_0 \text{resetn} + \text{resetn}' q_2$$

$$T_1 = q_3' q_0 \text{resetn} + q_0 b_{-}d \text{resetn} + \text{resetn}' q_1 \quad T_0 = \text{resetn} + \text{resetn}' q_0$$

**5.13**

$$q_2^+ = q_2' q_0 \text{plus}3 + q_2' q_1 \text{plus}3 + q_2' q_1 q_0 + q_2 q_0' \text{plus}3' + q_2 q_1' q_0' + q_2 q_1' \text{plus}3'$$

$$q_1^+ = q_1' q_0' \text{plus}3 + q_1' q_0 \text{plus}3' + q_1 q_0' \text{plus}3' + q_1 q_0 \text{plus}3$$

$$q_0^+ = q_0'$$

**5.14**

a) Räknaren Cnt3b\_rec ersätts med räknare modulo-16, Cnt4b\_rec.

b) Räknaren Cnt2b\_rec ersätts med en räknare modulo-8, Cnt3b-rec, avkodaren Avkod1\_4 ersätts med en avkodare Avkod1\_8 och multiplexern MUX4\_1 ersätts med en MUX8\_1.

**5.15**

I blockschemat för sändaren Tx8Ch4\_1 ersätts multiplexern MUX4\_1 med en DMUX1\_4.

**5.16**

a) Räknaren Cnt3b\_rec ersätts med räknare modulo-16, Cnt4b\_rec.

b) Räknaren Cnt2b\_rec ersätts med en räknare modulo-8, Cnt3b-rec, avkodaren Avkod1\_4 ersätts med en avkodare Avkod1\_8 och demultiplexern DMUX1\_4 ersätts med en DMUX1\_8.

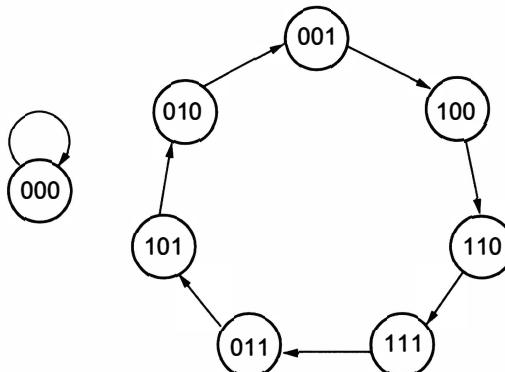
**5.17**

$$\begin{aligned} T_{\text{clock}} &\geq t_{\text{pd\_D}} + t_{\text{pd\_Inv}} + t_{\text{pd\_OCH}} + t_{\text{pd\_ELLER}} + t_{\text{su\_D}} = 0,8 + 0,5 + 1 + 1 + 0,2 \\ &= 3,5 \text{ ns} \end{aligned}$$

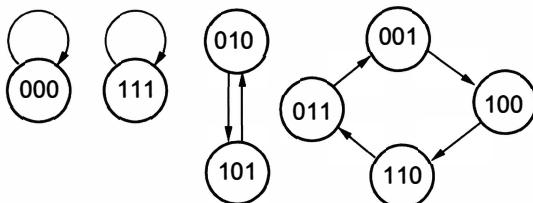
$$f_{\text{clock}} = 1 / T_{\text{clock}} \leq 28,6 \text{ MHz}$$

**5.18**

a)



b)



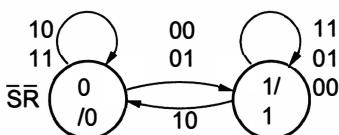
c)  $u = q_0 = 1001110100111010011101001$

**5.19**

$$T_{\text{clock}} \geq t_{\text{pd\_D}} + t_{\text{pd\_MUX}} + t_{\text{su\_D}} = 2 + 1,5 + 0,5 \\ = 4 \text{ ns}$$

$$f_{\text{clock}} = 1 / T_{\text{clock}} \leq 25,0 \text{ MHz}$$

**5.20**



**5.21**

a)

Q	Nuvarande tillstånd				Nästa tillstånd Q <sup>+</sup>			
	Ingångsvärde AB							
	00	01	11	10	0	0	1	0
0	0	0	1	0	0	1	1	0
1	0	1	1	1	0	1	1	1

b)

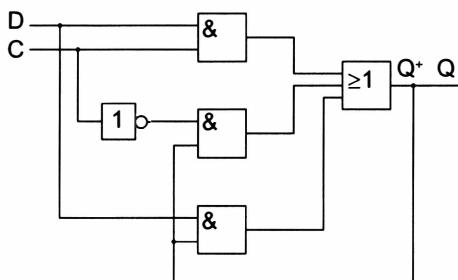
AB = 10 el. 01 är viloläge

AB = 00 ger nollställning

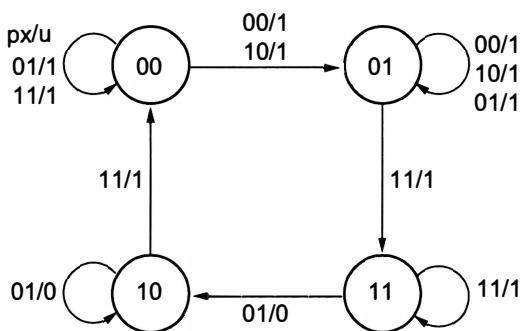
AB = 11 ger ettställning

**5.22**

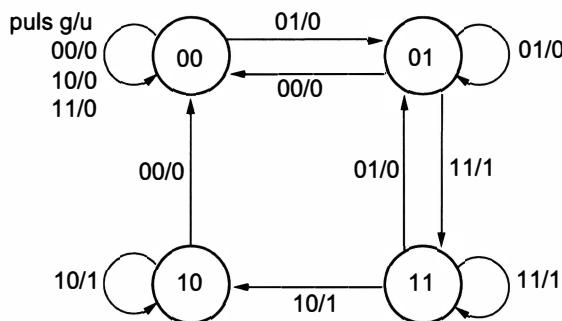
		DC					
		00	01	11	10		
Q	0	0	0	1	0		
	1	1	0	1	1		
						Q <sup>+</sup>	Q



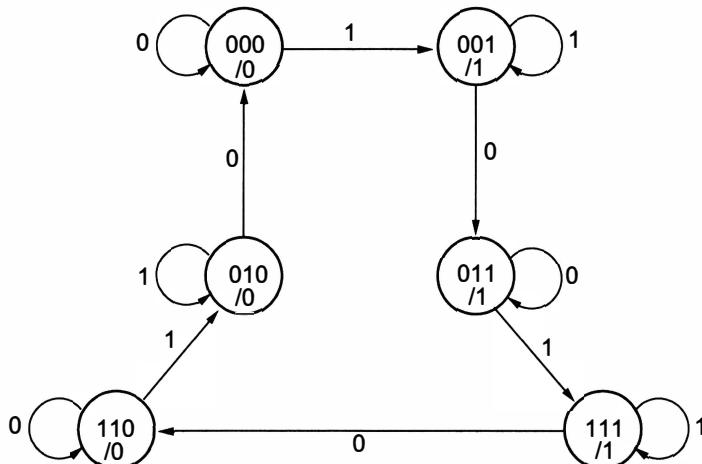
**5.23**



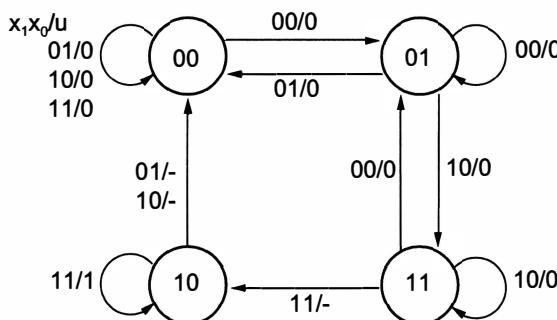
5.24



5.25



5.26



# 6 MOS-transistor

## Grindar i CMOS och nMOS

### 6.4 Grindar i CMOS och nMOS

6.1a)

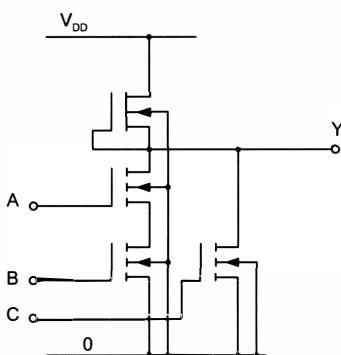
A	B	M1	M2	X	M4	M5	M6	Y
0	0	F	F	1	T	F	F	0
0	1	F	T	0	F	T	F	1
1	0	T	F	0	F	F	T	1
1	1	T	T	0	F	T	T	0

b)

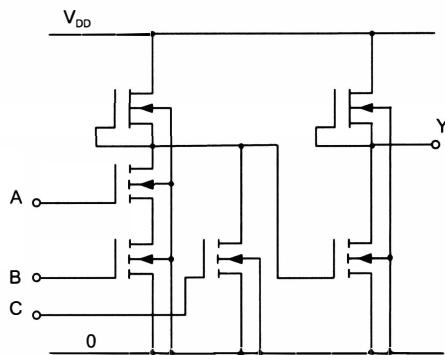
A	B	N1	P1	N2	P2	X	N3	P3	N4	P4	N5	P5	Y
0	0	F	T	F	T	1	T	F	F	T	F	T	0
0	1	F	T	T	F	0	F	T	T	F	F	T	1
1	0	T	F	F	T	0	F	T	F	T	T	F	1
1	1	T	F	T	F	0	F	T	T	F	T	F	0

6.2

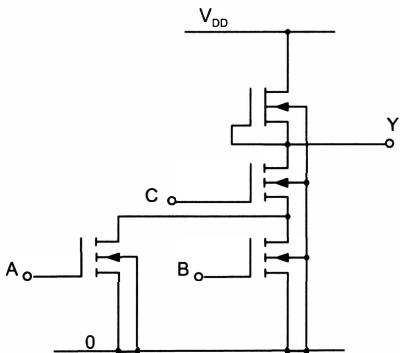
a)



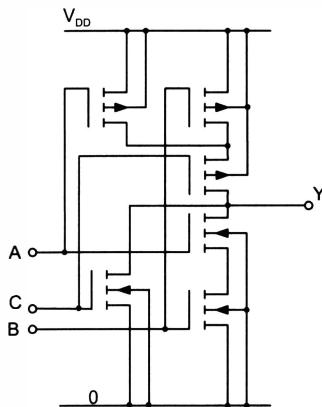
b)



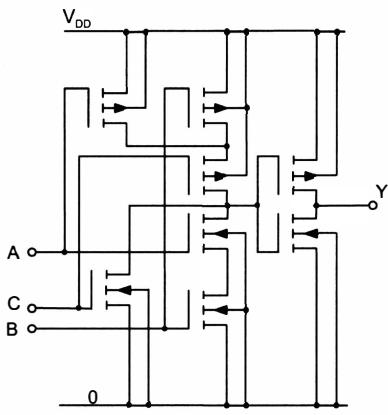
6.2 c)



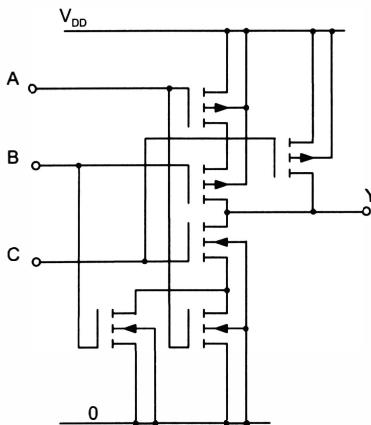
6.3



6.3 b)



c)



6.4 a)  $((A \cdot B)' \cdot')' = (A + B)'$ ; NOR-grind

b) Grinden har buffrade in- och utgångar. Utgången har konstant utresistans som inte varierar med insignal kombinationerna.

6.5  $Y = ((AB)'B)'((AB)'A)' = (AB)'B + (AB)'A = (A' + B')B + (A' + B')A = A'B + B'A = A \oplus B$ ; XOR-grind

6.6

E	A	Y
0	X	Z
1	0	0
1	1	1

**6.7**

E1	E2	A	Y
0	0	0	1
0	0	1	0
0	1	0	Z
0	1	1	Z
1	0	0	Z
1	0	1	Z
1	1	0	Z
1	1	1	Z

**6.8**

$$E = 0 \Rightarrow Y = Z$$

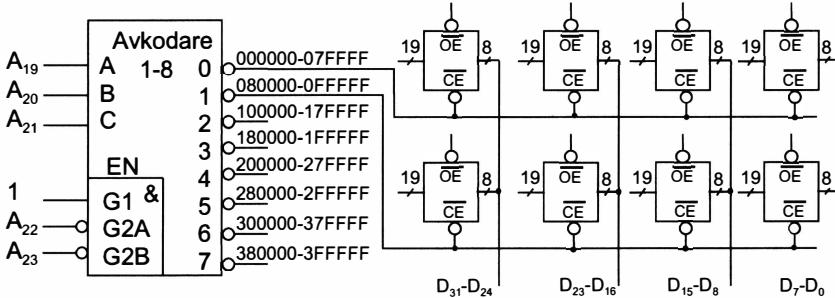
$$E = 1 \Rightarrow Y = (((A+B)'+C)')' =$$

$$(A+B)' + C = A'B' + C$$

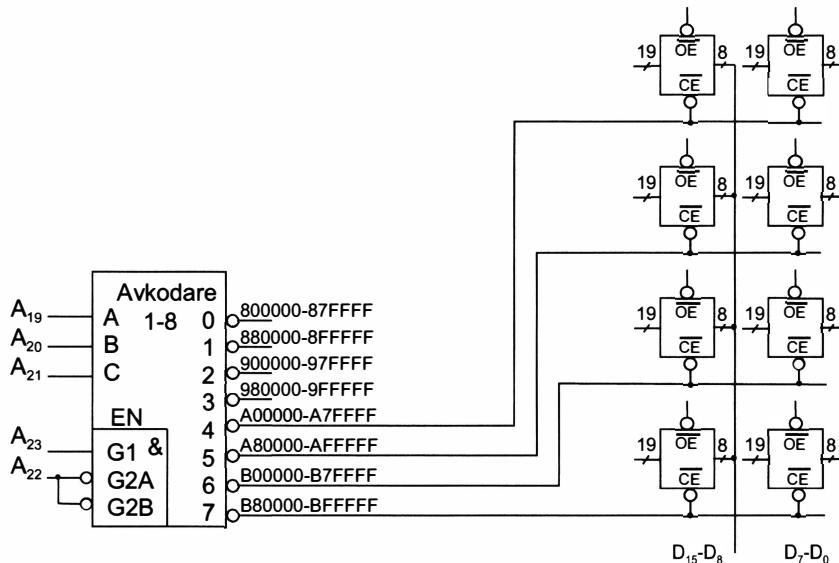
## 7 Halvledarminnen

- 7.1 a)** Flytta ledningarna till chipselect till avkodarens utgångar 3 - 5.
- b)** Skifta A<sub>22</sub> och A<sub>23</sub> samt flytta ledningarna till chipselect till avkodarens utgångar 1 - 3.
- 7.2 a)** 8 kapslar utefter ord längdsaxeln och 8 rader kapslar utefter ordaxeln.
- b)** 9 kapslar utefter ord längdsaxeln och 16 rader kapslar utefter ordaxeln.

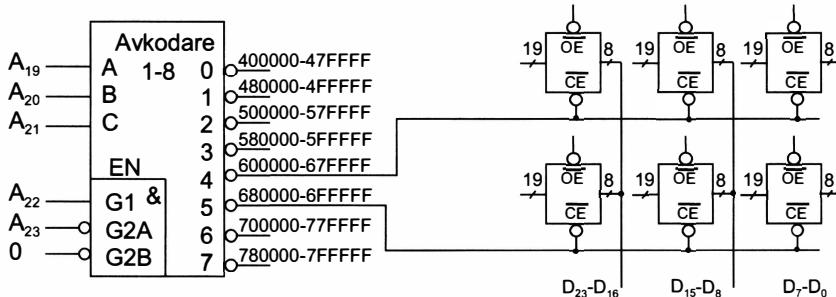
**7.3**



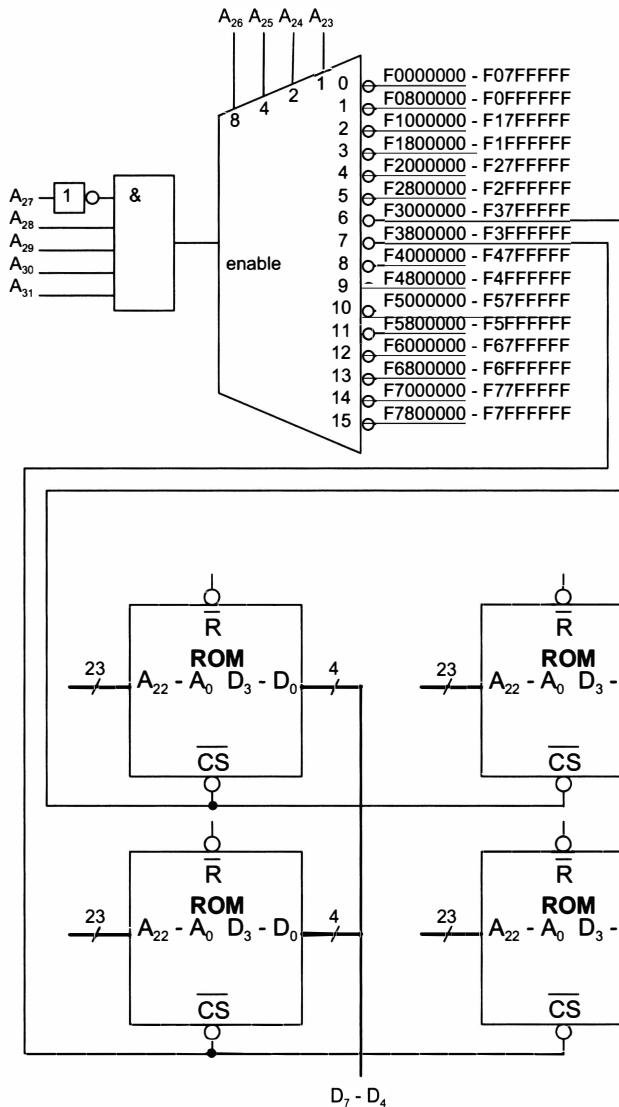
7.4 a)



7.4 b)



**7.5 a)**



**7.6 b)** Kapselkonfigurationen skall vara 4 kapslar utefter ordaxeln och 3 rader utefter ordaxeln. Till OCH-grinden vid avkodarens enableingång skall anslutas A<sub>31</sub>, A<sub>30</sub>', A<sub>29</sub>, A<sub>28</sub>' och A<sub>27</sub>. Ledningarna till chipselect skall anslutas till avkodarens utgångar 1 - 3.

# 9 VHDL – en introduktion

## 9.1

--Komb91  
--2001-01-06 Lars-H Hemert

```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity Komb91 is  
    port(x0,x1, x2, x3, x4, x5: in std_logic;  
          u0, u1, u2: out std_logic);  
end entity Komb91;  
  
architecture beteende of Komb91 is  
begin  
    u0 <= (not x0 and not x2) xor (x4 xor x5);  
    u1 <= not(x0 xor x1) and (x5 xor x2);  
    u2 <= not(x0 and (not x1 or not x2 or  
                      not x3 or not x4 or not x5));  
end architecture beteende;
```

## 9.2

--Komb92  
--2001-08-06 Lars-H Hemert

```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity Komb92 is  
    port(x, y: in std_logic_vector(3 downto 0);  
          EQU, xGE4, yLE4: out std_logic);  
end entity Komb92;  
  
architecture beteende of Komb92 is  
begin  
    EQU <= not((x(3) xor y(3)) or (x(2) xor y(2)) or  
               (x(1) xor y(1)) or (x(0) xor y(0)));  
    xGE4 <= x(2) or x(3);  
    yLE4 <= not y(3) and ((y(2) and not y(1) and not y(0)) or  
                           not y(2));  
end architecture beteende;
```

### 9.3

```
--FL
--2001-08-06 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity FL is
    port(x, y: in std_logic;
          m: in std_logic_vector(1 downto 0);
          z: out std_logic);
end entity FL;

architecture beteende of FL is
begin
    process(x, y, m)
    begin
        case m is
            when "00"      => z <= not x;
            when "01"      => z <= x and y;
            when "11"      => z <= x or y;
            when "10"      => z <= x xor y;
            when others   => null;
        end case;
    end process;
end architecture beteende;
```

### 9.4

```
--DMUX1_4
--2001-08-06 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DMUX1_4 is
    port(data_in: in std_logic;
          a:in std_logic_vector(1 downto 0);
          data_ut: out std_logic_vector(3 downto 0));
end entity DMUX1_4;

architecture beteende of DMUX1_4 is
begin
    process(data_in, a)
    begin
```

```
data_ut <= "0000" --default (grundvärde)
  case a is
    when "00"    => data_ut(0) <= data_in;
    when "01"    => data_ut(1) <= data_in;
    when "10"    => data_ut(2) <= data_in;
    when "11"    => data_ut(3) <= data_in;
    when others => null;
  end case;
end process;
end architecture beteende;
```

## 9.5

--Maxnumber  
--2001-08-06 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Maxnumber is
  port(x: in std_logic_vector(5 downto 1);
        u: out std_logic_vector(2 downto 0));
end entity Maxnumber;

architecture beteende of Maxnumber is
begin
  process(x)
  begin
    if x(5) = '0' then
      u <= "101";
    elsif x(4) = '0' then
      u <= "100";
    elsif x(3) = '0' then
      u <= "011";
    elsif x(2) = '0' then
      u <= "010";
    elsif x(1) = '0' then
      u <= "001";
    else
      u <= "000";
    end if;
  end process;
end architecture beteende;
```

### 9.6

```
--Level
--2001-08-06 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity Level is
  port(x: in std_logic_vector(4 downto 1);
       u: out std_logic_vector(2 downto 0));
end entity Level;

architecture beteende of Level is
begin
  process(x)
  begin
    if x(4) = '1' then
      u <= "100";
    elsif x(3) = '1' then
      u <= "011";
    elsif x(2) = '1' then
      u <= "010";
    elsif x(1) = '1' then
      u <= "001";
    else
      u <= "000";
    end if;
  end process;
end architecture beteende;
```

### 9.7

```
--Square_x_BCD
--2001-08-06 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Square_x_BCD is
  port(x: in std_logic_vector(3 downto 1);
       msd, lsd: out std_logic_vector(3 downto 0));
end entity Square_x_BCD;

architecture beteende of Square_x_BCD is
begin
```

```

process(x)
begin
    case x is
        when "0000" => msd <= "0000"; lsd <= "0000";
        when "0001" => msd <= "0000"; lsd <= "0001";
        when "0010" => msd <= "0000"; lsd <= "0100";
        when "0011" => msd <= "0000"; lsd <= "1001";
        when "0100" => msd <= "0001"; lsd <= "0110";
        when "0101" => msd <= "0010"; lsd <= "0101";
        when "0110" => msd <= "0011"; lsd <= "0110";
        when "0111" => msd <= "0100"; lsd <= "1001";
        when "1000" => msd <= "0110"; lsd <= "0100";
        when "1001" => msd <= "1000"; lsd <= "0001";
        when others => msd <= "----"; lsd <= "----";
    end case;
    end process;
end architecture beteende;

```

## 9.8

--Lower\_xyz  
--2001-08-06 Lars-H Hemert

```

library IEEE;
use IEEE.std_logic_1164.all;

entity Lower_xyz is
    port(x, y, z: in std_logic_vector(1 downto 0);
          u: out std_logic);
end entity Lower_xyz;

architecture beteende of Lower_xyz is
begin
    process(x, y, z)
    begin
        if (z = "11" and y = "10" and (x = "01" or x = "00")) or
            (z = "11" and y = "01" and x = "00") or
            (z = "10" and y = "01" and x = "00") then
            u <= '1';
        else
            u <= '0';
        end if;
    end process;
end architecture beteende;

```

## 9.9

```
--S51a_mo
--2001-08-06 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity S51a_mo is
    port(resetn, x, clock: in std_logic;
        u: out std_logic);
end entity S51a_mo;

architecture beteende of S51a_mo is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2, s3, s4, s5);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, x)
begin
    case present_state is
        when start      => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s1;
                            end if;
        when s1         => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s2;
                            end if;
        when s2         => if x = '0' then
                                next_state <= s3;
                            else
                                next_state <= s2;
                            end if;
        when s3         => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s4;
                            end if;
        when s4         => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s5;
```

```

        end if;
when s5      => if x = '0' then
                  next_state <= s3;
else
                  next_state <= s2;
end if;

end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Moore: process(present_state)
begin
  case present_state is
    when s5      => u <= '1';
    when others   => u <= '0';
  end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= start;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
end architecture beteende;

--S51b_me
--2001-08-06 Lars-H Hemert

```

```

library IEEE;
use IEEE.std_logic_1164.all;

entity S51b_me is
  port(resetn, x, clock: in std_logic;
       u: out std_logic);
end entity S51b_me;

```

```
architecture beteende of S51b_me is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2, s3, s4);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Mealy: process(present_state, x)
begin
    case present_state is
        when start      => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s1;
                            end if;
        when s1          => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s2;
                            end if;
        when s2          => if x = '0' then
                                next_state <= s3;
                            else
                                next_state <= s2;
                            end if;
        when s3          => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s4;
                            end if;
        when s4          => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s2;
                            end if;
    end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state)
begin
    case present_state is
        when s4          => if x = '1' then
                                u <= '1';
                            else
```

```

                u <= '0';
            end if;
when others      => u <= '0';
end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;

--S51c_me
--2001-08-06 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity S51c_me is
    port(resetn, x, clock: in std_logic;
         u: out std_logic_vector(1 downto 0));
end entity S51c_me;

architecture beteende of S51c_me is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Mealy: process(present_state, x)
begin
    case present_state is
        when start      => if x = '0' then
                                next_state <= start;
                            else
                                next_state <= s1;
                            end if;
        when s1         => if x = '0' then
                                next_state <= s1;
                            else
                                next_state <= s2;
                            end if;
        when s2         => if x = '0' then
                                next_state <= s2;
                            else

```

```
        next_state <= start;
    end if;
end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state)
begin
    case present_state is
        when start => if x = '0' then
            u <= "00";
        else
            u <= "01";
        end if;
        when s1 => if x = '0' then
            u <= "01";
        else
            u <= "10";
        end if;
        when s2 => if x = '0' then
            u <= "10";
        else
            u <= "00";
        end if;
    end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

--S51d\_mo  
--2001-08-06 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity S51d_mo is
    port(resetn, x, clock: in std_logic;
         u: out std_logic);
end entity S51d_mo;
```

---

```

architecture beteende of S51d_mo is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2, s3, s4);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, x)
begin
    case present_state is
        when start      => if x = '0' then
                                next_state <= start;
                                else
                                    next_state <= s1;
                                end if;
        when s1          => if x = '0' then
                                next_state <= s1;
                                else
                                    next_state <= s2;
                                end if;
        when s2          => if x = '0' then
                                next_state <= s2;
                                else
                                    next_state <= s3;
                                end if;
        when s3          => if x = '0' then
                                next_state <= s3;
                                else
                                    next_state <= s4;
                                end if;
        when s4          => next_state <= s4;
    end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Moore: process(present_state)
begin
    case present_state is
        when s3          => u <= '1';
        when others      => u <= '0';
    end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;

```

```
--S51e_mo
--2001-08-06 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity S51e_mo is
  port(resetn, x, y, clock: in std_logic;
        u: out std_logic_vector(1 downto 0));
end entity S51e_mo;

architecture beteende of S51e_mo is
  -- deklaration av tillståndstyp
  type state_type is (start, s1, s2);
  -- deklaration av nuvarande tillstånd och nästa tillstånd
  signal present_state, next_state: state_type;

begin
  -- kombinatorisk process för beräkning av nästa tillstånd
  state_diagram_Moore: process(present_state, x)
    begin
      case present_state is
        when start      => if (x = '0' and y = '0') or
                               (x = '1' and y = '1') then
                               next_state <= start;
                               elsif (x = '1' and y = '0') then
                               next_state <= s1;
                               else
                               next_state <= s2;
                               end if;
        when s1         => next_state <= s1;
        when s2         => next_state <= s2;
      end case;
    end process;

  -- kombinatorisk process för beräkning av utsignaler
  utsignaler_Moore: process(present_state)
    begin
      case present_state is
        when start      => u <= "00";
        when s1         => u <= "10";
        when s2         => u <= "01";
      end case;
    end process;
```

```
-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;

--S51f_mo
--2001-08-06 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity S51f_mo is
  port(resetn, x, clock: in std_logic;
        u: out std_logic);
end entity S51f_mo;

architecture beteende of S51f_mo is
  -- deklaration av tillståndstyp
  type state_type is (start, s1, s2, s3);
  -- deklaration av nuvarande tillstånd och nästa tillstånd
  signal present_state, next_state: state_type;

begin
  -- kombinatorisk process för beräkning av nästa tillstånd
  state_diagram_Moore: process(present_state, x)
    begin
      case present_state is
        when start      => if x = '0'   then
          next_state <= start;
        else
          next_state <= s1;
        end if;
        when s1         => if x = '1'   then
          next_state <= s1;
        else
          next_state <= s2;
        end if;
        when s2         => if x = '0'   then
          next_state <= s2;
        else
          next_state <= s3;
        end if;
        when s3         =>      next_state <= s3;
      end case;
    end process;
```

```
-- kombinatorisk process för beräkning av utsignaler
utsignaler_Moore: process(present_state)
begin
    case present_state is
        when s1 to s2 => u <= '1';
        when others      => u <= '0';
    end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

--S51g\_mo  
--2001-08-06 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity S51g_mo is
    port(resetn, clock: in std_logic;
          x: in std_logic_vector(1 downto 0);
          u: out std_logic);
end entity S51g_mo;

architecture beteende of S51g_mo is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2, s3);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, x)
begin
    case present_state is
        when start      => if x(0) = '0' then
            next_state <= start;
        elsif x = "01" then
            next_state <= s3;
        else
            next_state <= s1;
        end if;
```

```

when s1      => if x(0) = '0' then
                  next_state <= start;
                elsif x = "01" then
                  next_state <= s3;
                else
                  next_state <= s2;
                end if;
when s2      => if x(0) = '0' then
                  next_state <= start;
                elsif x = "01" then
                  next_state <= s3;
                else
                  next_state <= s2;
                end if;
when s3      => if x(1) = '1' then
                  next_state <= s1;
                elsif x = "01" then
                  next_state <= s3;
                else
                  next_state <= start;
                end if;
end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Moore: process(present_state)
begin
  case present_state is
    when s2      => u <= '1';
    when others   => u <= '0';
  end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;

```

**9.10**

--Sdet10  
--2001-08-06 Lars-H Hemert

```

library IEEE;
use IEEE.std_logic_1164.all;

```

```
entity Sdet10 is
  port(resetn, x, clock: in std_logic;
        u: out std_logic);
end entity Sdet10;

architecture beteende of Sdet10 is
-- deklaration av tillståndstyp
type state_type is (start, s1, detekt, rest);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
  -- kombinatorisk process för beräkning av nästa tillstånd
  state_diagram_Moore: process(present_state, x)
  begin
    case present_state is
      when start => if x = '0' then
                      next_state <= start;
                    else
                      next_state <= s1;
                    end if;
      when s1 => if x = '1' then
                      next_state <= s1;
                    else
                      next_state <= detekt;
                    end if;
      when detekt => next_state <= rest;
      when rest => next_state <= rest;
    end case;
  end process;

  -- kombinatorisk process för beräkning av utsignaler
  utsignaler_Moore: process(present_state)
  begin
    case present_state is
      when detekt => u <= '1';
      when others => u <= '0';
    end case;
  end process;

  -- inmatning av nästa tillstånd i tillståndsregistret
  --(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

**9.11**

```
--Sdet010or0110
--2001-08-06 Lars-H Hemert
library IEEE;
use IEEE.std_logic_1164.all;

entity Sdet010or0110 is
  port(resetn, x, clock: in std_logic;
       u: out std_logic);
end entity Sdet010or0110;

architecture beteende of Sdet010or0110 is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2, s3, s4, s5, s6);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, x)
begin
  case present_state is
    when start      => if x = '1' then
                           next_state <= start;
                           else
                           next_state <= s1;
                           end if;
    when s1         => if x = '0' then
                           next_state <= s1;
                           else
                           next_state <= s2;
                           end if;
    when s2         => if x = '0' then
                           next_state <= s3;
                           else
                           next_state <= s4;
                           end if;
    when s3         => if x = '0' then
                           next_state <= s5;
                           else
                           next_state <= s6;
                           end if;
    when s4         => if x = '0' then
                           next_state <= s3;
                           else
                           next_state <= start;
                           end if;
```

```
when s5      => if x = '0' then
                    next_state <= s5;
                else
                    next_state <= s6;
                end if;
when s6      => if x = '0' then
                    next_state <= s5;
                else
                    next_state <= start;
                end if;
end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Moore: process(present_state)
begin
    case present_state is
        when s3      => u <= '1';
        when others   => u <= '0';
    end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

## 9.12

--Sdet1110  
--2001-08-06 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Sdet1110 is
    port(resetn, x, clock: in std_logic;
         u: out std_logic);
end entity Sdet1110;

architecture beteende of Sdet1110 is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2, s3, s4);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;
```

```

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Mealy: process(present_state, x)
begin
  case present_state is
    when start      => if x = '0' then
      next_state <= start;
    else
      next_state <= s1;
    end if;
    when s1          => if x = '0' then
      next_state <= start;
    else
      next_state <= s2;
    end if;
    when s2          => if x = '0' then
      next_state <= start;
    else
      next_state <= s3;
    end if;
    when s3          => if x = '1' then
      next_state <= s3;
    else
      next_state <= s4;
    end if;
    when s4          => if x = '0' then
      next_state <= start;
    else
      next_state <= s1;
    end if;
  end case;
end process;

```

```

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state)
begin
  case present_state is
    when s4        => if x = '0' then
      u <= '1';
    else
      u <= '0';
    end if;
    when others   =>      u <= '0';
  end case;
end process;

```

```
-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

### 9.13

```
--Sgen07
--2001-07-28 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Sgen07 is
  port(resetn, clock: in std_logic;
        q: out std_logic_vector(2 downto 0));
end entity Sgen07;

architecture beteende of Sgen07 is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 7;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
  state_diagram: process(present_state)
    begin
      case present_state is
        when 0           => next_state <= 7;
        when 1           => next_state <= 6;
        when 2           => next_state <= 5;
        when 3           => next_state <= 4;
        when 4           => next_state <= 0;
        when 5           => next_state <= 3;
        when 6           => next_state <= 2;
        when 7           => next_state <= 1;
      end case;
    end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
  q <= conv_std_logic_vector(present_state,3);

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

**9.14**

--Sgen013764  
--2001-07-28 Lars-H Hemert

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Sgen013764 is
  port(resetn,enable, clock: in std_logic;
        q: out std_logic_vector(2 downto 0));
end entity Sgen013764;

architecture beteende of Sgen013764 is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 7;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, enable)
begin
  if enable = '0' then
    next_state <= present_state;
  else
    case present_state is
      when 0          => next_state <= 1;
      when 1          => next_state <= 3;
      when 3          => next_state <= 7;
      when 7          => next_state <= 6;
      when 6          => next_state <= 4;
      when 4          => next_state <= 0;
      when others     => null;
    end case;
  end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,3);

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;

```

### 9.15

--Sgen064325  
--2001-07-28 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Sgen064325 is
    port(resetn,enable,set6, clock: in std_logic;
          q: out std_logic_vector(2 downto 0));
end entity Sgen064325;

architecture beteende of Sgen064325 is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 7;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, enable, set6)
begin
    if set6 = '1' then
        next_state <= 6;
    elsif enable = '0' then
        next_state <= present_state;
    else
        case present_state is
            when 0      => next_state <= 6;
            when 1      => next_state <= 0;
            when 2      => next_state <= 5;
            when 3      => next_state <= 2;
            when 4      => next_state <= 3;
            when 5      => next_state <= 0;
            when 6      => next_state <= 4;
            when 7      => next_state <= 0;
        end case;
    end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,3);

-- inmatning av nästa tillstånd i tillståndsregistret
--(se S51a_mo i övn.uppgift 9.9)
end architecture beteende;
```

**9.16**

```

--Adder_serial_me
--2001-08-07 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity Adder_serial_me is
  port(resetn, x, y, clock: in std_logic;
        c_ut, s: out std_logic);
end entity Adder_serial_me;

architecture beteende of Adder_serial_me is
-- deklaration av tillståndstyp
type state_type is (nocarry, carry);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Mealy: process(present_state, x, y)
begin
  case present_state is
    when nocarry => if (x = '1' and y = '1') then
                           next_state <= carry;
                         else
                           next_state <= nocarry;
                         end if;
    when carry      => if (x = '0' and y = '0') then
                           next_state <= nocarry;
                         else
                           next_state <= carry;
                         end if;
  end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state, x, y)
begin
  case present_state is
    when nocarry => if (x = '0' and y = '0') then
                           c_ut <= '0';
                           s     <= '0';
                         elsif (x = '0' and y = '1') or
                               (x = '1' and y = '0') then
                           c_ut <= '0';
                           s     <= '1';
                         else
                           c_ut <= '1';
                           s     <= '0';
                         end if;
  end case;
end process;

```

```
        else
            c_ut    <= '1';
            s       <= '0';
        end if;
when carry      => if (x = '1' and y = '1') then
            c_ut    <= '1';
            s       <= '1';
        elsif (x = '0' and y = '1') or
            (x = '1' and y = '0') then
            c_ut    <= '1';
            s       <= '0';
        else
            c_ut    <= '0';
            s       <= '1';
        end if;
    end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
    if rising_edge(clock) then
        if resetn = '0' then
            present_state <= nocarry;
        else
            present_state <= next_state;
        end if;
    end if;
end process;
end architecture beteende;

--Adder_serial_mo
--2001-08-07 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;

entity Adder_serial_mo is
    port(resetn, x, y, clock: in std_logic;
          c_ut, s: out std_logic);
end entity Adder_serial_mo;

architecture beteende of Adder_serial_mo is
-- deklaration av tillståndstyp
type state_type is (nocarry_s0,nocarry_s1,carry_s0,carry_s1);
```

```
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, x, y)
begin
case present_state is
    when nocarry_s0=> if (x = '0' and y = '0') then
        next_state <= nocarry_s0;
    elsif (x = '0' and y = '1') or
        (x = '1' and y = '0') then
        next_state <= nocarry_s1;
    else
        next_state <= carry_s0;
    end if;
    when nocarry_s1=> if (x = '0' and y = '0') then
        next_state <= nocarry_s0;
    elsif (x = '0' and y = '1') or
        (x = '1' and y = '0') then
        next_state <= nocarry_s1;
    else
        next_state <= carry_s0;
    end if;
    when carry_s0 => if (x = '0' and y = '0') then
        next_state <= nocarry_s1;
    elsif (x = '0' and y = '1') or
        (x = '1' and y = '0') then
        next_state <= carry_s0;
    else
        next_state <= carry_s1;
    end if;
    when carry_s1 => if (x = '0' and y = '0') then
        next_state <= nocarry_s1;
    elsif (x = '0' and y = '1') or
        (x = '1' and y = '0') then
        next_state <= carry_s0;
    else
        next_state <= carry_s1;
    end if;
end case;
end process;
```

```
-- kombinatorisk process för beräkning av utsignaler
utsignalen_Moore: process(present_state, x, y)
begin
  case present_state is
    when nocarry_s0 => c_ut <= '0';
                           s     <= '0';
    when nocarry_s1 => c_ut <= '0';
                           s     <= '1';

    when carry_s0      => c_ut <= '1';
                           s     <= '0';
    when carry_s1      => c_ut <= '1';
                           s     <= '1';

  end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= nocarry_s0;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
end architecture beteende;
```

### 9.17

```
--Pos_edge_me
--2001-08-07 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Pos_edge_me is
  port(resetn, p, clock: in std_logic;
        u: out std_logic);
end entity Pos_edge_me;

architecture beteende of Pos_edge_me is
-- deklaration av tillståndstyp
type state_type is (start, s1);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;
```

```

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Mealy: process(present_state, p)
begin
    case present_state is
        when start      => if p = '0' then
            next_state <= start;
        else
            next_state <= s1;
        end if;
        when s1          => if p = '0' then
            next_state <= start;
        else
            next_state <= s1;
        end if;
    end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignaler_Mealy: process(present_state, p)
begin
    case present_state is
        when start      => if p = '0' then
            u <= '0';
        else
            u <= '1';
        end if;
        when s1          =>     u <= '0';
    end case;
end process;

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
    if rising_edge(clock) then
        if resetn = '0' then
            present_state <= start;
        else
            present_state <= next_state;
        end if;
    end if;
end process;
end architecture beteende;

```

```
--Pos_edge_mo
--2001-08-07 Lars-H Hemert
library IEEE;
use IEEE.std_logic_1164.all;

entity Pos_edge_mo is
  port(resetn, p, clock: in std_logic;
        u: out std_logic);
end entity Pos_edge_mo;

architecture beteende of Pos_edge_mo is
-- deklaration av tillståndstyp
type state_type is (start, s1, s2);
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram_Moore: process(present_state, p)
begin
  case present_state is
    when start      => if p = '0' then
                           next_state <= start;
                           else
                           next_state <= s1;
                           end if;
    when s1         => if p = '0' then
                           next_state <= start;
                           else
                           next_state <= s2;
                           end if;
    when s2         => if p = '0' then
                           next_state <= start;
                           else
                           next_state <= s2;
                           end if;
  end case;
end process;

-- kombinatorisk process för beräkning av utsignaler
utsignalera_Moore: process(present_state, p)
begin
  case present_state is
    when s1      => u <= '1';
    when others   => u <= '0';
  end case;
end process;
```

```
-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= start;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
end architecture beteende;
```

**9.18**

```
--Cnt3G
--2001-07-28 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cnt3G is
  port(resetn, clock: in std_logic;
        q: out std_logic_vector(2 downto 0));
end entity Cnt3G;

architecture beteende of Cnt3G is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 7;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state)
begin
  case present_state is
    when 0          => next_state <= 1;
    when 1          => next_state <= 3;
    when 2          => next_state <= 6;
    when 3          => next_state <= 2;
    when 4          => next_state <= 0;
    when 5          => next_state <= 4;
    when 6          => next_state <= 7;
    when 7          => next_state <= 5;
  end case;
end;
```

```
    end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,3);

-- inmatning av nästa tillstånd i tillståndsregistret
-- asynkron resetna
state_register: process(clock, resetna)
begin
    if resetna = '0' then
        present_state <= 0;
    elsif rising_edge(clock) then
        present_state <= next_state;
    end if;
end process;
end architecture beteende;
```

## 9.19

--Cnt2Gm  
--2001-07-28 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cnt2Gm is
    port(resetn, clock: in std_logic;
          m: in std_logic_vector(1 downto 0);
          q: out std_logic_vector(1 downto 0));
end entity Cnt2Gm;

architecture beteende of Cnt2Gm is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 3;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;
```

```

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state)
  begin
    if m = "00" then
      next_state <= 0;
    elsif m = "11" then
      next_state <= 3;
    else
      case present_state is
        when 0           => if m = "01" then
          next_state <= 1;
        else
          next_state <= 2;
        end if;
        when 1           => if m = "01" then
          next_state <= 3;
        else
          next_state <= 0;
        end if;
        when 2           => if m = "01" then
          next_state <= 0;
        else
          next_state <= 3;
        end if;
        when 3           => if m = "01" then
          next_state <= 2;
        else
          next_state <= 1;
        end if;
      end case;
    end if;
  end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,2);

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
  begin
    if rising_edge(clock) then
      if resetn = '0' then
        present_state <= 0;
      else
        present_state <= next_state;
      end if;
    end if;
  end process;

```

```
    end if;
end process;
end architecture beteende;
```

## 9.20

--Cnt4bd  
--2001-07-28 Lars-H Hemert

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cnt4bd is
    port(resetn, b_d, clock: in std_logic;
        q: out std_logic_vector(3 downto 0));
end entity Cnt4bd;

architecture beteende of Cnt4bd is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 15;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, b_d)
begin
    case present_state is
        when 0 to 8      => next_state <= present_state + 1;
        when 9           => if b_d = '0' then
                                next_state <= 0;
                            else
                                next_state <= 10;
                            end if;
        when 10 to 14   => next_state <= present_state + 1;
        when 15          => next_state <= 0;
    end case;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,4);
```

```
-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= 0;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
end architecture beteende;
```

**9.21**

```
--Cnt3b_plus3
--2001-07-28 Lars-H Hemert
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Cnt3b_plus3 is
  port(resetn, plus3, clock: in std_logic;
       q: out std_logic_vector(2 downto 0));
end entity Cnt3b_plus3;

architecture beteende of Cnt3b_plus3 is
-- deklaration av tillståndstyp
subtype state_type is integer range 0 to 7;
-- deklaration av nuvarande tillstånd och nästa tillstånd
signal present_state, next_state: state_type;

begin
-- kombinatorisk process för beräkning av nästa tillstånd
state_diagram: process(present_state, plus3)
begin
  if plus3 = '0' then
    case present_state is
      when 0 to 6 => next_state <= present_state + 1;
      when 7        => next_state <= 0;
    end case;
  else
    case present_state is
      when 0 to 4 => next_state <= present_state + 3;
      when 5       => next_state <= 0;
    end case;
  end if;
end process;
```

```
    when 6          => next_state <= 1;
    when 7          => next_state <= 2;
  end case;
end if;
end process;

-- utsignalerna tilldelas tillståndsregistrets utsignaler
q <= conv_std_logic_vector(present_state,3);

-- inmatning av nästa tillstånd i tillståndsregistret
-- synkron resetn
state_register: process(clock)
begin
  if rising_edge(clock) then
    if resetn = '0' then
      present_state <= 0;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
end architecture beteende;
```

## 9.22

```
--Rx8Ch4_1
--2001-08-7 Lars-H Hemert
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Rx8Ch1_4 is
  port(resetn, enable, clock: in std_logic;
        d_in: in std_logic;
        d_ut: out std_logic_vector(2 downto 0);
        tidlucka: out std_logic_vector(3 downto 0));
end entity Rx8Ch1_4;

architecture struktur of Rx8Ch1_4 is

--komponentdeklarationer
component Cntgnb_rec
generic(n: positive := 4;
        max_count: integer := 15);
  port(resetn, enable, clock: in std_logic;
        carry_out: out std_logic;
        q: out std_logic_vector((n-1) downto 0));
end component;
```

```
component DMUX1_4
port(data_in: in std_logic;
      a:in std_logic_vector(1 downto 0);
      data_ut: out std_logic_vector(3 downto 0));
end component;

component Avkodl_4
port(adress:in std_logic_vector(1 downto 0);
      data_ut: out std_logic_vector(3 downto 0));
end component;

--deklaration av interna signaler
signal icarry: std_logic;
signal iq: std_logic_vector(1 downto 0);

--beskrivning av komponentförbindningarna
begin
  Cnt3b_rec: Cntgnb_rec
    generic map(n => 3, max_count => 7)
    port map(clock => clock, resetn => resetn,
              enable => enable, carry_out => icarry,
              q => open);
  Cnt2b_re: Cntgnb_rec
    generic map(n => 2, max_count => 3)
    port map(clock => clock, resetn => resetn,
              enable => icarry, q => iq);

  DMUX1_4_1: DMUX1_4
    port map(a => iq, data_in => d_in, data_ut => d_ut);

  Avkodl_4_1: Avkodl_4
    port map(adress => iq, data_ut => tidlucka);

end architecture struktur;
```

### 9.23

```
--AU_serial
--2001-08-7 Lars-H Hemert

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity AU_serial is
    port(resetn, x, y, sub_add, clock: in std_logic;
          c_ut, s: out std_logic);
end entity AU_serial;

architecture struktur of AU_serial is

--komponentdeklarationer
component Adder_serial_me
port(resetn, x, y, clock: in std_logic;
      c_ut, s: out std_logic);
end component;

component MUX2_1
port(d_in0, d_in1, adress:in std_logic;
      d_ut: out std_logic);
end component;

component TwoComp_me
port(x, resetn, clock: in std_logic;
      u: out std_logic);
end component;

--deklaration av interna signaler
signal iu, id_ut: std_logic;

--beskrivning av komponentförbindningarna
begin
    TwoComp_me_1: Twocomp_me
        port map(clock => clock, resetn => resetn,
                  x => y, u => iu);

    MUX2_1_1: MUX2_1
        port map(d_in0 => y, d_in1 => iu,
                  adress => sub_add, d_ut => id_ut);
```

```
    Adder_serial_me_1: Adder_serial_me
      port map (s => s, c_ut => c_ut, clock => clock
                  x => x, y => id_ut, resetn => resetn);

end architecture struktur;
```

# Sakregister

- A/D 13, 434, 447  
A/D Converter *se A/D*  
A/D-omvandlare *se A/D*  
absorptionslagarna 105, 106  
accesstid 355  
ADC *se A/D*  
Adderare 171  
adressaccesstid 356  
adressbuss 354  
adressera 353  
adresskriftid 356  
aktiv låg 59  
aktiv pull-up 339  
alfanumeriska koder 36  
ALU 175, 190  
analog 11, 12  
analog komparator 448  
Analog/Digital-omvandlare *se A/D*  
AND gate 56  
anrikningstyp 317, 319  
Application Specific Integrated Circuits 11  
Application Specific Integrated Circuits  
*se ASIC*  
architecture 44, 399  
Arithmetic Unit, AU 171, 185  
Aritmetisk enhet 185  
Aritmetisk Logisk Enhet 175, 190  
Aritmetisk Logisk Enhet *se ALU*  
Aritmetiska enhet 171  
ASCII-kod 36  
ASIC 11, 20  
ASPLD 24  
assemblyspråk 191  
assignment statement *se tilldelningssats*  
asynchronous counters *se asynkron räknare*  
asynkron 86  
Asynkron reset 412  
asynkron räknare 250  
asynkron sekvenskrets 284  
avkoda 69  
Avkodare 69  
bas 25  
BCD-kod 35, 183  
beteende 46, 398, 405  
BGA, Ball Grid Array 397  
Binary Coded Decimal 183  
binary offset *se binär offset*  
binär 14  
binär offset 446  
binär signal 16  
binär-offset-representation 180  
bipolär 444  
bit 14  
bitcell i dynamiskt RWM 383  
bitcell i EPROM 372  
bitcell i statiskt RWM 378  
bitvis 188  
bitvisa logiska operationer 189  
body-effect 320, 339  
body-effekten *se body-effect*  
boolean function *se boolesk funktion*  
Boolesk algebra 103  
boolesk algebra 102  
boolesk funktion 120  
boolesk produkt 122

borrow	32, 175	CPU	191, 354
borrow	se <i>lånesiffra</i>	current state	82
burst	388	D flip flop	80
buss	354	D/A	13, 434, 435
byte	34, 354	D/A Converter	se D/A
cache	359	D/A-omvandlare	se D/A
carry	32, 172	DAC	se D/A
carry	se <i>minnessiffra</i>	datafördelare	18, 67
carry_out	263	dataväg	19
carryaccelerator	174	datavägenhet	268
CAS	387	dataväljare	18
case-satser	404	dator	354
Central Processing Unit	191	datormodell	355
Central Processing Unit	se CPU	DDR (Double Data Rate)	387
Centralenhet	191	DDR SDRAM	386
character	se <i>tecken</i>	De Morgans lagar	105, 109
Chip Enable	362	decoder	69
chipselect	380	decrement	191
chipselect	se <i>kapselval</i>	dekadräknare	269
Claude Shannon	14, 102	Demultiplexer	67
CLB, Configurable Logic block	397	demultiplexer	se DMUX
Clear	86	depletion type	se <i>utarmningstyp</i>
clear	280	differentiellt linjäriftetsfel	443
CMOS	312, 324	digital	11
CMOS-inverterare	324	Digital/Analog-omvandlare	se D/A
Column Adress Strobe	se CAS	digitala signaler	11
Column-Address Latch	387	DIMM (Dual In-line Memory	
combinational circuit	119	Module)	388
combinational circuits	80	Direct Rambus DRAM	388
comparator	se <i>jämförare</i>	diskreta	16
Compiler	50	distans	198
component	422	D-latch	281
concatenate	420	DMUX	17, 18, 67
concurrent statements	46, 404	don't care	82, 143
concurrent statements	se <i>samtidiga satser</i>	Double Data Rate	387
Configurable Logic Block	395	drain	se <i>kollektor</i>
Configurable Logic block	397	DRAM	se <i>dynamiskt RWM</i>
consensuslag	105, 107	DRDRAM	388
Control bus	354	DRWM	se <i>dynamiskt RWM</i>
counter	84, 250	dual	104
counter	se <i>räknare</i>	dubbelfel	199
		D-vippa	80, 293

dynamiskt RWM 359, 383  
E<sup>2</sup>PROM 374  
EDO (Extended Data Out) 388  
EEPLD 395  
EEPROM 358, 374  
ELLER-grind 58  
ELLER-OCH 124  
ELLER-summa 47  
embedded 352  
emitter *se source*  
emulate *se emulera*  
emulera 394  
Enable 77  
enable 70, 89, 260, 348  
Enable-signal 362  
enhancement type *se anrikningstyp*  
entity 44, 399  
EPLD 395  
EPROM 358, 370  
Erasable Programmable Read Only Memory *se EPROM*  
ETOX 376  
1-komplement 178  
even parity *se jämn paritet*  
execute 192  
Exklusivt-ELLER 111  
Exklusivt-ELLER-grind 74  
expansion till minne med flera kapslar 364  
Extended Data Out 388  
  
FA 172  
FA *se heladderare*  
faktorisering 152  
fall time 79  
fall time *se falltid*  
falling\_edge() 409  
falltid 79, 336  
FAMOS 370  
Fast Page Mode 388  
felrättande 196  
felrättande kod 199  
  
Felupptäckande 196  
felupptäckande kod 199  
fetch 192  
Field Programmable Logic Array *se FPLA*  
Finite State Machine 213  
FL 189  
Flaggor 195  
flaggor 192  
flash-converter 448  
Flash-minnen 358  
flash-minnet 376  
Floating gate Avalanche Injection MOS *se FAMOS*  
floating gate *se flytande styre*  
FLOTOX 375  
flyktiga 357  
flytande styre 370  
FPGA 24  
FPLA 391  
FPM (Fast Page Mode) 388  
Frekvensdelare 85  
frekvensdelning 271  
frequency divider 85  
frisvävande 77, 346  
FSM 213  
full adder 172  
full adder *se heladderare*  
full custom design 20  
Full Logic 189  
Födröjning 254  
fördröjning 78, 336  
förenkling 126  
förstärkningsfel *se skalfaktorfel*  
  
gain error *se skalfaktorfel*  
generic 424  
generic map() 426  
generisk 424  
George Boole 102  
glitch 168  
Gray 221  
gray-kod 37

- grind 21  
 grinddelning 145  
 grindmatris 21  
 grindnivåer 61  
 halvledarminne 352  
 hasard 167  
 heladderare 172  
 hexadecimal 26  
 hold time 254  
 hot electrons 370, 377  
 hålltid 254  
 högohmig 77, 346
- IC 195  
 ICKE 327  
 icke-flyktiga 357  
 icke-linjäriteten *se linjäritetsfel*  
 identifierare 46  
 IEC 47  
 ield Programmable Gate Array 24  
 if-sats syntax 405  
 if-satser 404  
 inbyggda system 352  
 increment 191  
 inringning 129  
 instance 423  
 Instruction counter 195  
 instruktioner 192, 354  
 Instruktionsavkodare 195  
 Instruktionsregister 195  
 Instruktionsräknare 195  
 inställningstid 254, 438  
 Intel 360  
 Intellectual Property 24, 397  
 interface 45  
 intervall 16  
 inverter 58  
 inverterar 326  
 Inverterare 58  
 IP-block 24, 397  
 JEDEC-standard 394
- jämförare 201  
 jämn paritet 197  
 kanalbredd 315  
 kanallängd 315  
 kapacitans 322  
 kapplöpning 169  
 kapplöpningsfri 288  
 kapselval 362  
 karnaughdiagram 127  
 kaskadkoppling av räknare 262  
 klassificering av integrerade kretsar 20  
 klockfrekvens, maximal 254  
 kollektor 313  
 kombinationskrets 118, 119  
 kombinationskretsar 80  
 Komparator 75, 201  
 komponent 420  
 komponentinstansiering 423  
 konfigurerbara logikblock 397  
 kundspecifierade kretsar 20  
 kvantisera 14
- ladda 271  
 latch 278  
 latchar 277  
 LCA 395  
 library 403  
 linjär sekvenskrets 308  
 linjäritetsfel 443  
 load *se ladda*  
 Logic Cell Array *se LCA*  
 Logic Unit, LU 188  
 logikblock 396  
 Logisk enhet 188  
 look-ahead carry generator 174  
 look-ahead carry generator *se carryaccelerator*  
 Look-Up Table 397  
 LSB 26  
 LSD 26  
 LU 188  
 LUT (Look-Up Table) 397

- Lånesiffra 175  
 lånesiffra 32, 175  
 läscykeltid 356  
 läsminne 361  
 läsning 355  
  
 makrocell 396  
 master-slave 283  
 max 90  
 maximal klockfrekvens 254  
 maxterm 122  
 Mealy 93, 94, 231  
 Mealy till Moore 239  
 Metal Oxide Semiconductor *se MOS-transistor*  
 metastabilitet 299  
 metastabilt tillstånd 300  
 metastable state *se metastabilt tillstånd*  
 microcontroller 352  
 microcontroller *se mikrostyrkrets*  
 mikrostyrkrets 352, 446  
 minnesmodell 353  
 minnessiffra 32, 172  
 minterm 121, 122  
 mneme 191  
 mnemonics 191  
 Modell för en sekvenskrets av typ Mealy 231  
 Modell för en sekvenskrets av typ Moore 218  
 modell för minneschip 369  
 modsignaler 274  
 modulo-2n 251  
 monoton 438, 443  
 monotonitetsfel 443  
 Moore 92, 218  
 Moores lag 78  
 MOS-transistor 312, 313  
 MSB 26  
 MSD 25  
 Multiplexer 62  
 multiplexer *se MUX*  
 MUX 17, 18, 62  
  
 NAND-grind 72, 341  
 NAND-grind i nMOS och CMOS 341  
 NAND-NAND-nät 139  
 negativ flank 81  
 negativ logik 14, 327  
 negativa tal 180  
 negative edge 81  
 negative edge triggered 81  
 negativt flanktriggad 81  
 next state 82  
 9-komplement 179  
 nivåramp 450  
 nMOS 312, 313  
 nMOS-inverterare 338  
 noise margin *se störmarginal*  
 nollfel 442  
 non linearity *se linjäritetsfel*  
 nonvolatile 357  
 nonvolatile *se icke-flyktiga*  
 NOR-grind 74  
 NOR-grind i nMOS och CMOS 343  
 NOR-grinden 344  
 normalform 121  
 NOR-NOR-nät 139  
 NOT *se ICKE*  
 null 402  
 nuvarande tillstånd 82  
 nästa tillstånd 82  
  
 OCH-ELLER 124  
 OCH-grind 56  
 OCH-produkt 47  
 odd parity *se udda paritet*  
 offset error *se nollfel*  
 ofullständigt specificerade funktioner 142  
 oktal 26  
 omslagstid 334  
 omvandlingstid 451  
 One Time Programmable EPROM *se OTP-EPROM*  
 one-hot 223

- operand 191, 192, 354  
 operation 191, 192, 354  
 operationsförstärkare 438  
 ord 353  
 ordlängd 34  
 OTP-E PROM 358  
 utnyttjade tillstånd 243  
 overflow 186
- package 402  
 PAL 393  
 PALASM 394  
 paritetsbit 196  
 paritetskontroll 196  
 Paritetskrets 196, 200  
 paritetskrets 199  
 Parity 192  
 parity bit *se paritetsbit*  
 parity check *se paritetskontroll*  
 passiv pull-up 338  
 PCM 15  
 Place and Route 50  
 PLD 391  
 pMOS 313  
 polysilicon 315  
 port map() 423  
 portar 45  
 positiv flank 81  
 positiv logik 14, 327  
 positive edge 81  
 positive edge triggered 81  
 positivt flanktriggad 81  
 present state 82  
 Preset 86  
 preset 280  
 primimplikator 131  
 Process 404  
 Programcounter 195  
 Programmable Array Logic *se PAL*  
 Programmable Logic Device *se PLD*  
 programmerbar D/T-vippa 257  
 Programmerbara I/O block 397  
 Programräknare 195
- PROM 358, 367  
 propagation delay 78  
 propagation delay time 254  
 propagation delay time *se fördöjning*  
 pseudoslumpsekvens 308  
 PS-normalform 123  
 pull-down-transistor 326  
 pull-up-transistor 326  
 puls width modulation *se pulsbreddsmodulering*  
 pulsbreddsmodulering 446  
 pulskodmodulering *se PCM*  
 PWM *se pulsbreddsmodulering*
- Quine-McCluskey 155
- R-2R-resistorstege 440  
 radadressbuffert 387  
 RAM 358  
 Rambus 388  
 Random Access Memory *se RAM*  
 RAS 387  
 read cycle time *se läscykeltid*  
 Read Write Memory *se RWM*  
 redundant 196  
 refresh *se uppfiska*  
 Register 274  
 register 82  
 Register Transfer Level 50  
 reset 87, 88  
 resistor ladder *se R-2R-resistorstege*  
 ripple counters *se asynkron räknare*  
 rise time 79  
 rise time *se stigtid*  
 rising\_edge() 409  
 ROM 358, 361  
 Row Address Strobe *se RAS*  
 Row-Address Latch 387  
 RTL 50  
 RT-nivå 50  
 RWM 358  
 Räknare 250  
 räknare 84, 250

- Räknare med carry\_out 262  
 Räknare med enable 260  
 sampel 13  
 samplingsteoremet 14  
 samtidiga satser 46  
 sanningstabell 57  
 SAR *se successiv-approximationsregister*  
 SDR (Single Data Rate) 386  
 Sekvensdetektor 214  
 sekvensgenerator 246  
 sekvenskrets 213  
 sekvenskretsar 80  
 sekvenskretsmodell, generell 249  
 sensitivity list 405  
 sequential circuits 80  
 settling time *se inställningstid*  
 set-up time 254  
 shift register 83  
 Sign 192  
 sign bit *se teckenbit*  
 signal 11  
 signalnivå 331  
 Single Data Rate 386  
 sink 331  
 skalfaktorfel 443  
 skiftregister 83, 274  
 skrivcykeltid 356  
 skrivning 355  
 source 313, 332  
 SP-form 122  
 spik 168  
 SP-normalform 123  
 SRAM *se statiskt RWM*  
 SR-latchen 277  
 SRWM *se statiskt RWM*  
 stabilt tillstånd 285  
 standardceller 22  
 standardgrindnät i PLD 147  
 standardkretsar 20  
 starttillstånd 215  
 state 82  
 state assignment 219  
 state assignment *se tillståndskodning*  
 state diagram 82  
 state table 82  
 statiskt RWM 359, 378  
 Statusregister 195  
 std\_logic 45  
 stigtid 79, 336  
 struktur 46, 398, 405  
 Strukturbeskrivning 420  
 strukturbeskrivning 52  
 styrbuss 354  
 styrenhet 19, 268  
 störmarginal 331  
 substratpil 320, 321  
 successiv approximation 452  
 successiv-approximationsregister 452  
 summa *se boolesk produkt*  
 switch 317  
 switchmatris 396  
 synchronous counters *se synkron räknare*  
 Synkron reset 412  
 synkron räknare 250  
 synkrona minnen 386  
 synkronisering av asynkrona signaler och metastabilitet 299  
 tecken 36  
 tecken-belopp-representation 180  
 teckenbit 180  
 termometerkod 449  
 three-state output 77  
 Three-state-utgång 77  
 three-state-utgång 346  
 threshold voltage *se tröskelspanning*  
 tidlucka 267  
 tidsmultiplex 17, 19, 387  
 tilldelningssats 46  
 tillstånd 82  
 tillståndsdiagram 82  
 Tillståndskodning 219  
 tillståndskodning 236

tillståndsmaskin 213  
tillståndsminimering 229  
tillståndstabell 82  
tillståndsvariabel 236  
10-komplement 179  
toggle 256  
transformering 239  
transparent 281  
trench 385  
tri-state 77  
truth table 57  
tröskelspanning 318  
TTL-kompatibel 333  
TTL-nivå 333  
tunneleffekten 375  
T-vippa 256  
tvåfasklockning 283  
2-komplement 176  
två-nivå-nät 124

udda paritet 197  
unipolär 444  
Upp/ned-räknare 264  
uppfriska 359, 383  
uppräknebar 219  
utarmningstyp 319  
utgångsmakrocell 395

VHDL 41, 44, 52, 398  
vikt 25  
viktade resistorer 439  
Vippor 80  
virtuellt minne 360  
volatile 357  
volatile *se flyktiga*  
väsentlig primimplikator 131

when-else 401  
with-select-when 401  
word length *se ord längd*  
write cycle time *se skrivcykeltid*





**Lars-Hugo Hemert** är universitetslektor vid Malmö högskola. Han har skrivit flera böcker inom områdena digitalteknik och mikrodatorteknik.

## Lars-Hugo Hemert **Digitala kretsar**

Denna bok behandlar digitalteknik från grunden, från binära talsystem, boolesk algebra fram till beskrivning av digitala kretsar i det standardiserade hårdvarubeskrivande språket VHDL. Konstruktion, syntes, av digitala kretsar sker idag med hjälp av mycket avancerade datorbaserade syntesverktyg. Under konstruktionsprocessen från beskrivning av kretsen i språket VHDL till färdig krets arbetar konstruktören interaktivt med syntesverktyget och måste ta ställning till många frågor och fatta många beslut, vilket kräver ingående kunskaper om fundamentala digitala kretsars beteende och struktur. Syntesverktygen ger konstruktörerna möjlighet att konstruera komplexa kretsar, men ställer samtidigt stora krav på konstruktörernas grundkunskaper i digitalteknik.

Boken inleds med en analys av begreppen analog och digital och en elementär framställning av talsystem. Därefter följer en ingående behandling av kombinationskretsar och sekvenskretsar, de två klasser av kretsar som digitaltekniken baseras på. Många av kretsarna beskrivs i språket VHDL och blir på så sätt en introduktion till detta språk. Utöver kombinations- och sekvenskretsar behandlas även halvledarminnen och Digital/Analog- och Analog/Digital-omvandlare. Vidare visas hur digitalteknikens minsta byggstenar, grindarna, realiseras med MOS-transistorer, den komponent som varit och är förutsättningen för den fantastiska utvecklingen av digitala kretsar.

*Digitala kretsar* är avsedd att användas som kurslitteratur i digitalteknik på högskolenivå. Den kan även användas av den som vill lära sig digitalteknik på egen hand eller av den som vill förbättra sina grundkunskaper och ta steget in i språket VHDL. Inga särskilda förkunskaper i matematik och elektronik krävs. I boken ingår ett stort antal övningsuppgifter med svar.

Filer till de beskrivna kretsarna och övningsuppgifterna i kapitlet om VHDL kan hämtas från bokens hemsida: [www.studentlitteratur.se/3454](http://www.studentlitteratur.se/3454)

**Tredje upplagan**

Art.nr 3454

I S B N 978-91-44-01918-5

9 789144 019185

[www.studentlitteratur.se](http://www.studentlitteratur.se)