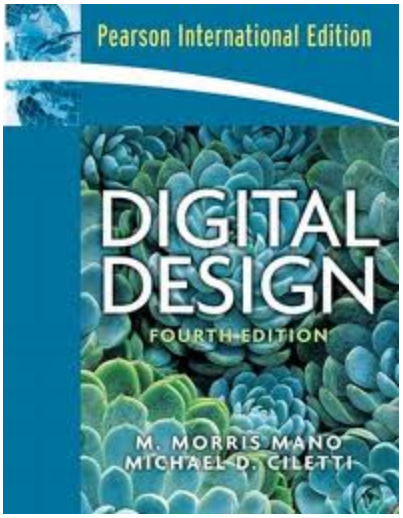
PENSUM

INF1400

H11



Digital Design, M. Morris Mano, 4th edition

Joakim Myrvoll Johansen

STIKKORDREGISTER:

INF1400

INTRO

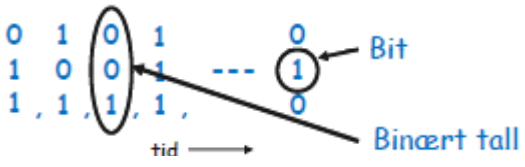
- ✗ MÅL:
  1) Forstå hvordan tall, bilder, lyd og andre signaler representeres ved hjelp av strøm/spenning styrt av transistorer
  2) Forstå hvordan de sentrale delene av en datamaskin er bygd opp helt ned til transistornivå
  3) Designe digitale elektroniske systemer helt nede på port/transistor nivå
  4) Designe din egen lille datamaskin (CPU)
  5) Kjenne til den Boolske algebraen som er grunnlaget for digitale tall/systemer
- ✗ TEMAER:
  1) Forståelse av grunnleggende digitalteknikk
  2) Transistor / CMOS teknologi
  3) Simulering
  4) Forenklinger basert på Boolsk algebra
  5) Kombinatorisk og sekvensiell logikk
  6) Tilstandsmaskiner
  7) Minne
  8) Asynkron logikk
  9) Automatisk systemkonstruksjon (VHDL)
- ✗ Datamaskinnivåer:
  1) Høynivåkode – Java/Python/PhP/C++/C#/Ruby…
  2) Assembler – maskinkode
  3) Horisontal - vertikal mikrokode
  4) Digital hardware – direkte og VHDL

  } INF1400

KAPITTEL 1

Digitalteknikk

- ✗ Hva betyr digital?
  - Binære tall:
    Tall som kun er representert ved symbolene 0 og 1 (bit). Hvert tall kan besta av ett eller flere bit (siffer)
  - Digitale signaler:
    - Sekvenser av binære tall i tid.



- Digitale systemer:
  - Systemer som håndterer digitale signaler

✗ Den digitale revolusjon I

- Digitale systemer:
  - CPU
    - Datamaskinen
      - Internett

✗ Den digitale revolusjon II

- Digitale systemer tar over analog signalbehandling
  - Eksempler:
    - Stereoanlegg (CD / vinylplate)
    - Video (DVD / VHS)
    - Foto (digitale kamera / 35mm)
    - Mobiltelefon (GSM)
    - osv...

✗ Den digitale revolusjon III

- Trådløse sensorsystemer:
  Millioner av små hardware "duppeditter"-online (internett)
  - Eksempler:
    - Sensorer/display i klær/stoffer/bøker
    - "Intelligent støv"
    - Fysiske "widgets"
    - Kroppsmonitorering
    - osv...

✗ Den digitale revolusjon IV

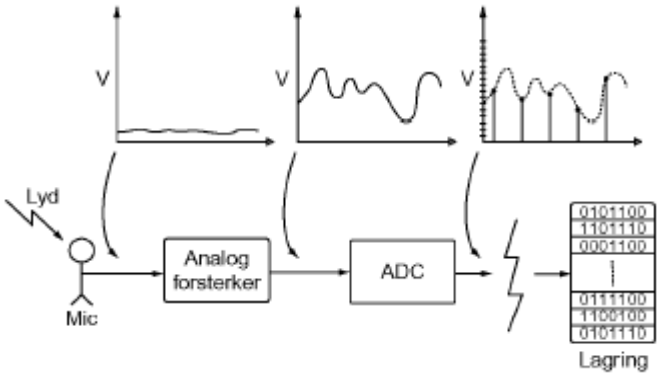
- Silisium er biokompatibelt !! ("Braingate")

✗ Den digitale revolusjon V

- Sammensmelting av mikroelektronikk og nanoteknologi
  (Hva blir forskjellen på maskin og biologi?)

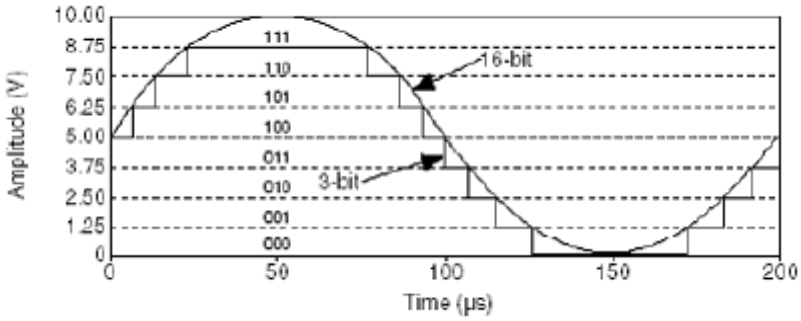
- ✗ Hvorfor er digitale systemer bedre enn analoge systemer?
  - Tapsfri signalbehandling
  - Tapsfri lagring av aignaler
  - Kraftigere muligheter for manipulering (filtrering)
  - Enklere design
  - Billigere
  - ...
- ✗ Det digitale "egget"
  Verden vi lever i er "stort sett" analog.
  Skal vi bruke digitale systemer til å håndtere analoge fenomen, trenegr vi en analog-til-digital-konverter + digital-til-analog-konverter

  - Eksempel:
    Innspilling av lyd



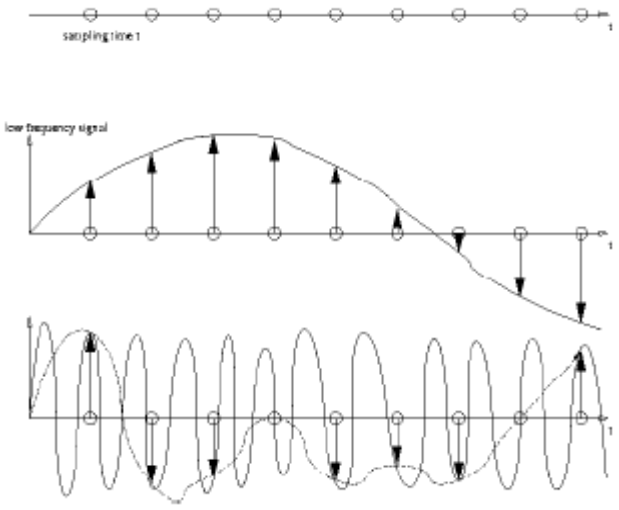
✗ To typer ADC-feil:

1. Kvantisering (avrundingsfeil)



16 bit – 2^{16} = 65536 mulige spenning/tallverdier.
Er 16 bit digital oppløsning av lyd "nok"?
"State of the art" - ADCs gir rundt 24 bit

2. For lav samplingshastighet



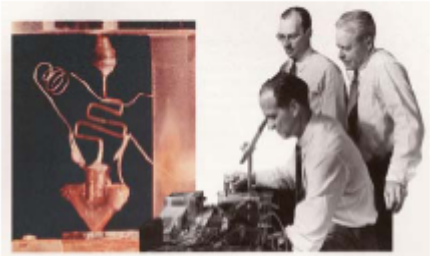
Nyquist teorem:
Mulighet for perfekt rekonstruksjon hvis man sampler raskere enn to ganger den høyeste frekvensen i signalet.

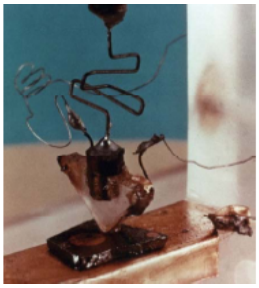
✗ Hvordan representere 0 og 1?

- Elektrisk strøm/spenning/ladning
- Lys
- Magnetisme
- Fysisk geometri
- osv...

✗ Transistoren

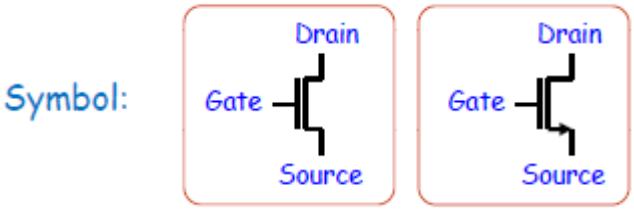
- Første transistor (erstatter radiorør):
  - Bell Laboratories, 1947 – William Shockley



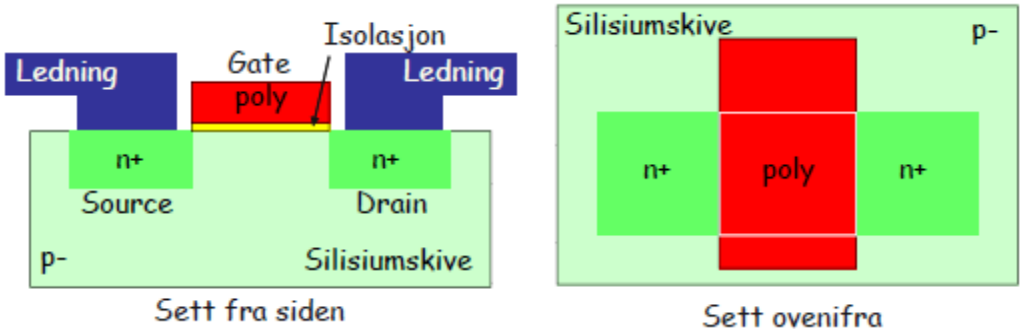


✗ NMOS transistoren

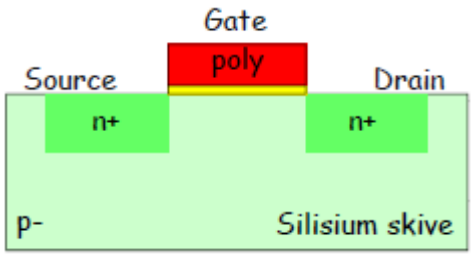
- NMOS (Negative doped Metal Oxide Silicon)
  - En 3 (4) terminals komponent



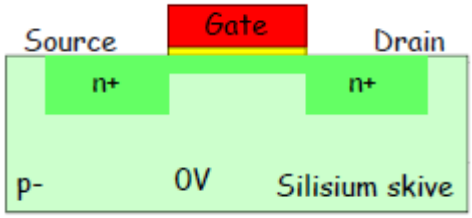
  - Spenningen på gate bestemmer om transistoren leder strøm mellom drain og source terminalene.



p- : Svakt positivt dopet silisium
n+ : Sterkt dopet silisium (ledende)
poly : Polykrystalinskt silisium (ledende)



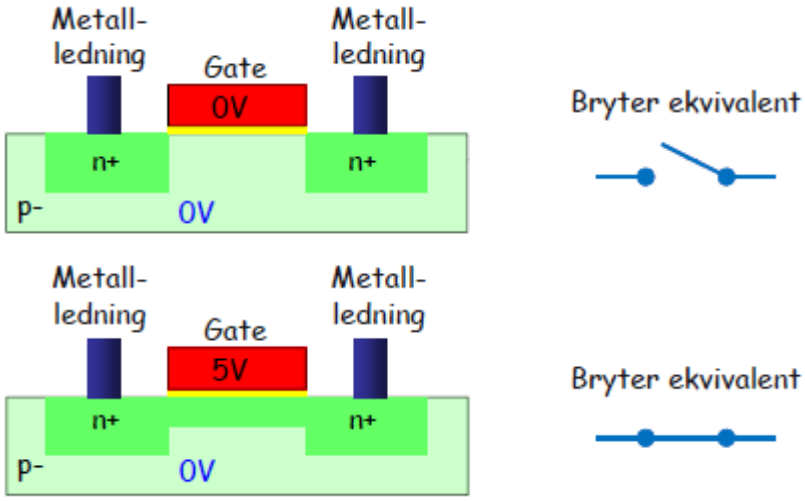
n+ og poly leder strøm, p- leder også strøm til en viss grad.
Strøm (elektroner) kan ikke gå fra p- til n+ materiale
Strøm kan derfor i utgangspunktet ikke gå fra source til drain



Hvis man setter en positiv spenning på gate terminalen (5V)* i forhold til silisiumskiven, dannes det er n+ lag under gate terminalen
Nå kan det gå strøm mellom source og drain.

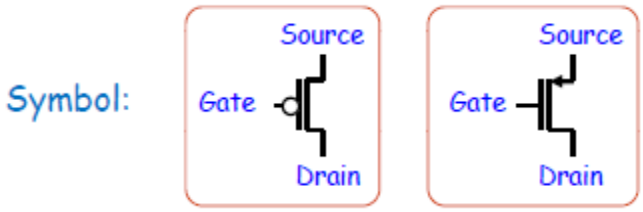
(* Forutsetter en 5V prosess i alle påfølgende forklaringer)

- NMOS brukt som styrt bryter (digital anvendelse):

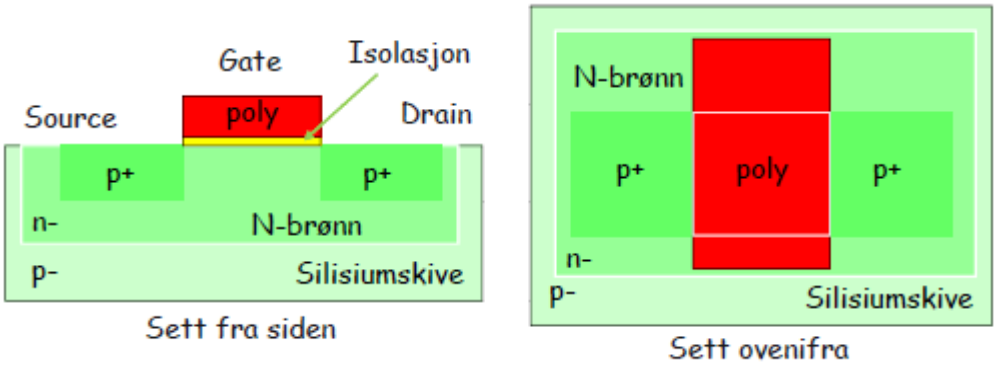


✗ PMOS transistoren

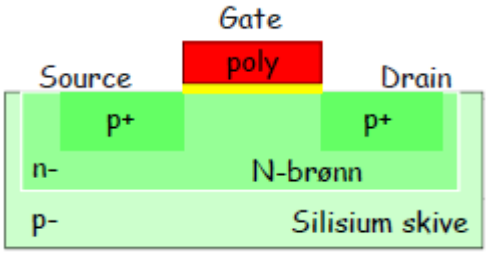
- PMOS (Positive doped Metal Oxide Silicon)
  - En 3 (4) terminals-komponent



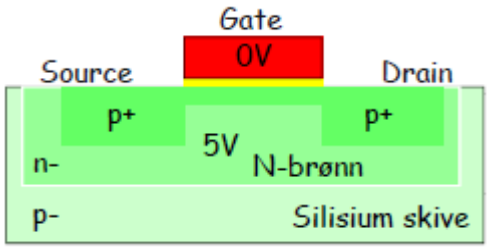
  - Spenningen på gate bestemmer om transistoren leder strøm i mellom drain og source terminalene



n- : Svakt negativt dopet silisium
p+ : Sterkt positivt dopet silisium (ledende)
poly : Polykrystalinskt silisium (ledende)

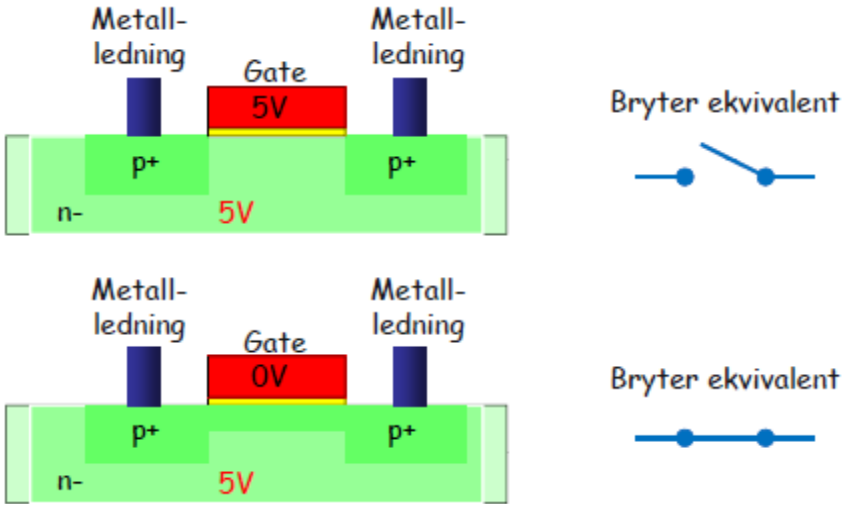


p+ og poly leder strøm, n- leder også strøm, til en viss grad
Strøm (elektroner) kan ikke gå fra p+ til n- materiale
Strøm kan derfor i utgangspunktet ikke gå fra drain til source



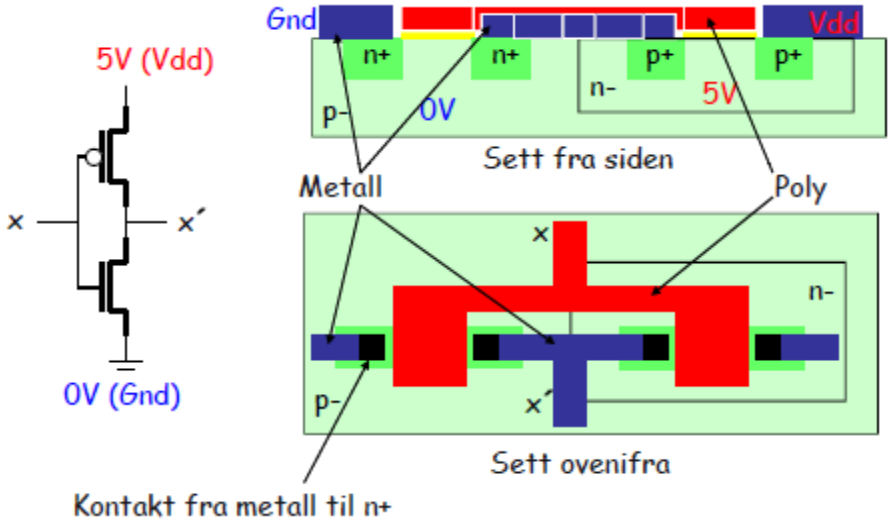
Hvis man setter en negatiiv spenning på gate-terminalen (-5V) i fohold til brønnen, dannes det et p+ lag under gate-terminalen
Nå kan det gå strøm mellom drain og source

- PMOS brukt som styrt bryter (digital anvendelse):

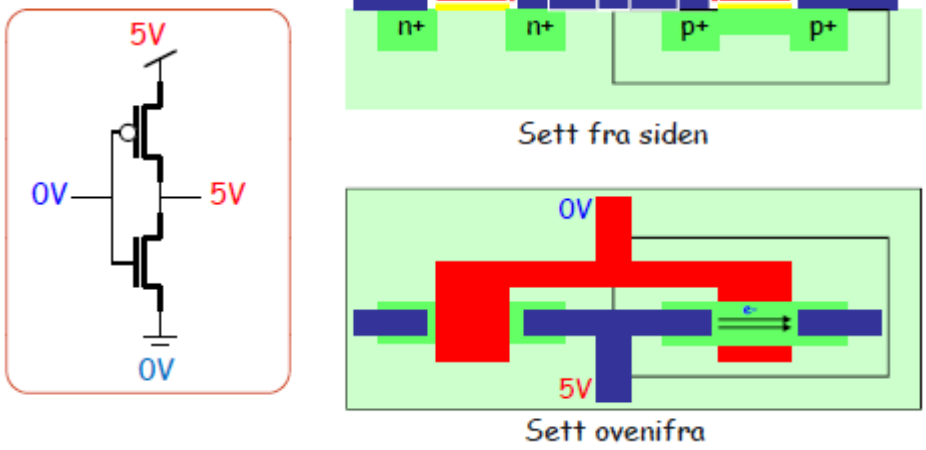


✗ CMOS kretser

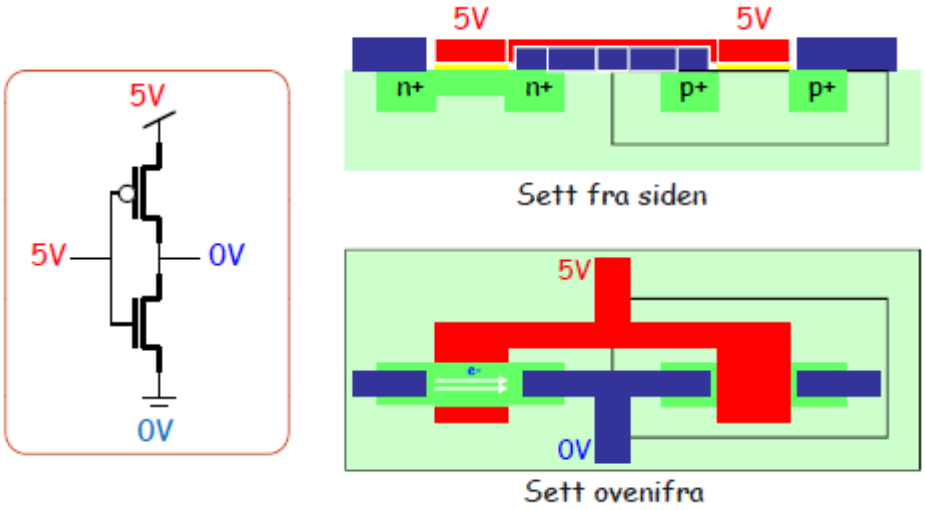
– CMOS (Complementary MOS) inverter



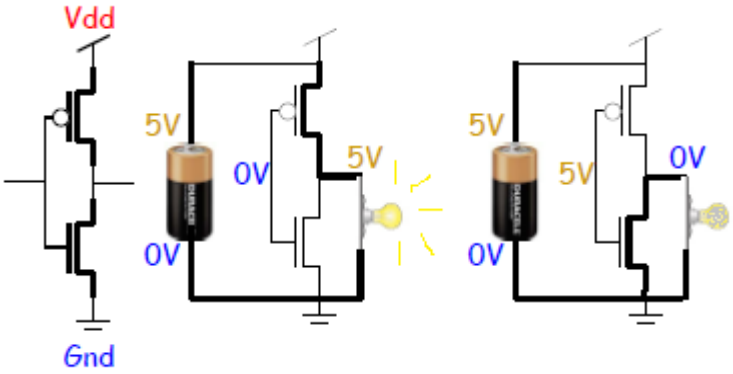
- Tilstand 1:
  0V inn, 5V ut



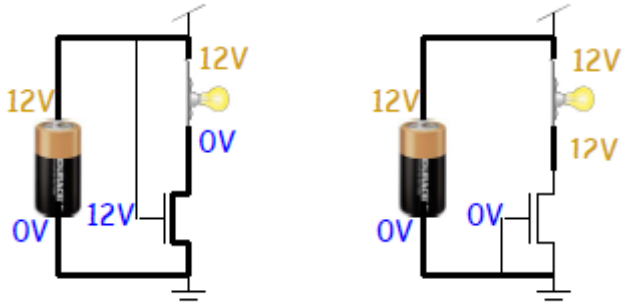
- Tilstand 2:
  5V inn, 0V ut



- Bryterekvivalent:



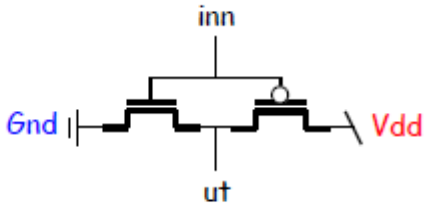
- Demo:



Ohms lov:
Spenning over motstand (lampe) =
Motstandsverdi * strøm =
Motstandsverdi * 0 = 0

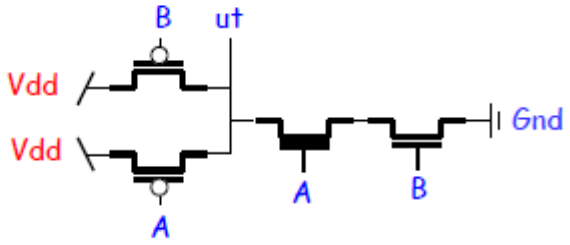
- Sannhetstabell:
  Lister opp alle mulige inngangsverdier + hva man får ut

inn	ut
0	1
1	0

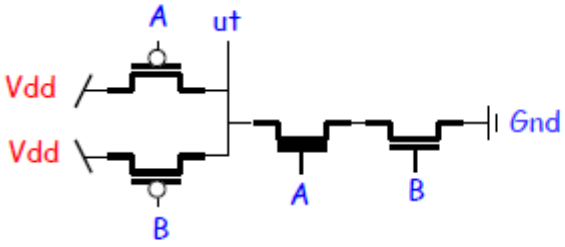


✗ CMOS NAND-krets

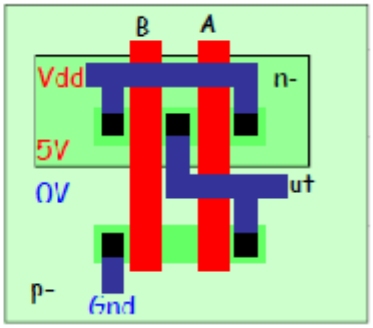
AB	ut
00	1
01	1
10	1
11	0



Både A og B må være 5V for å koble utgangen ned 0V



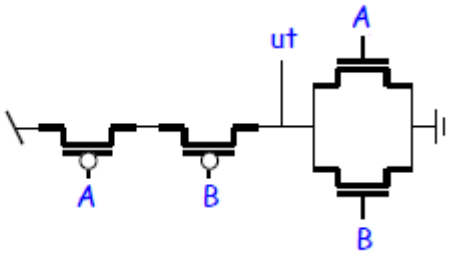
Skjema



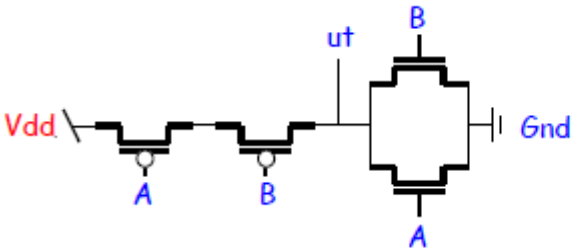
Utlegg

✗ CMOS NOR-krets

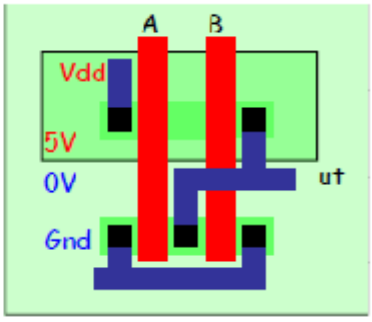
AB	ut
00	1
01	0
10	0
11	0



Det holder at enten A eller B er 5V for å koble utgangen ned til 0V



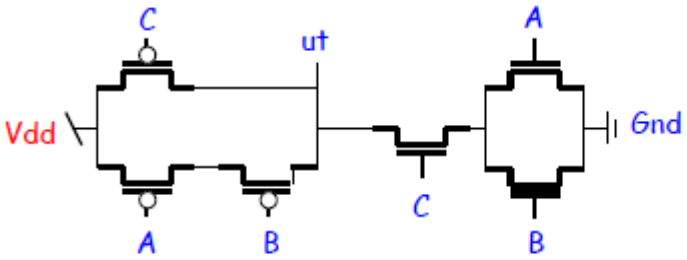
Skjema

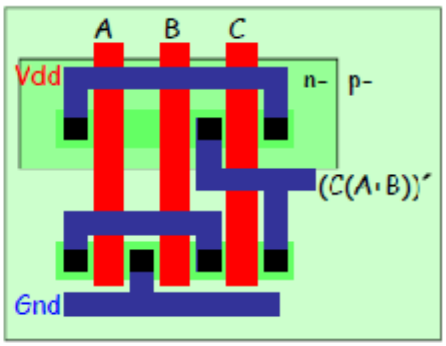


Utlegg

✗ CMOS-kretser
En enkel CMOS port kan implementere generelle funksjoner
Eksempel: F = (C(A+B))'

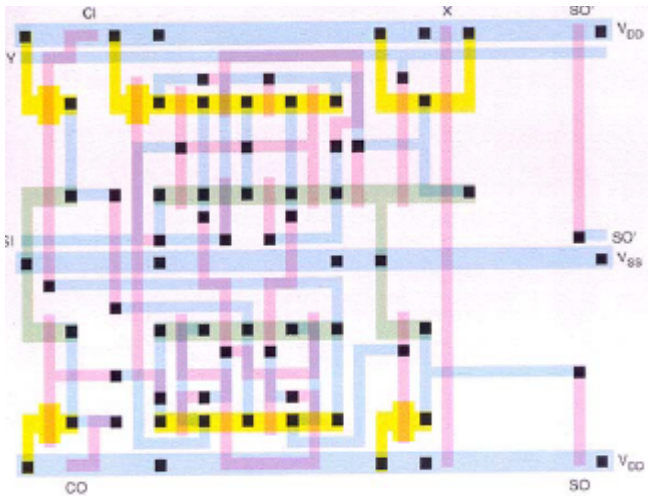
ABC	ut
000	1
001	1
010	1
011	0
100	1
101	0
110	1
111	0





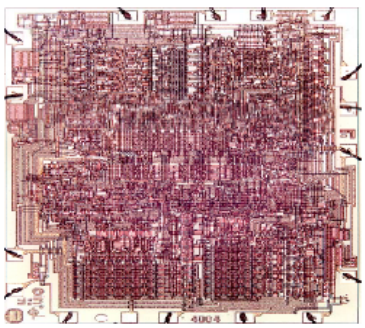
✗ CMOS-kretser

- Eksempel: Fulladder

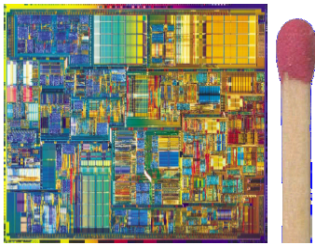


✗ Mikroelektronikk:

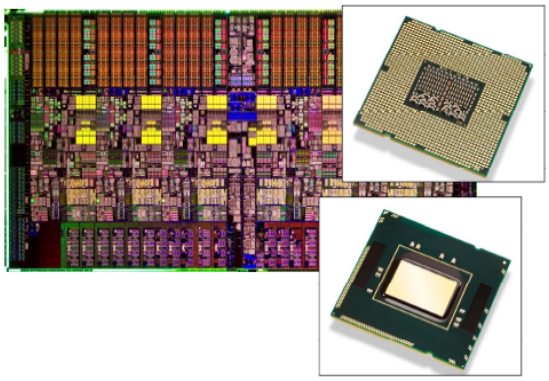
- Den egentlige årsaken til teknologiutviklingen
- Integrerte kretser
  - Texas Instruments – 1958
    - Germanium
  - Fairchild
    - Første kommersielle krets i silisium
  - Intel
    - Første mikroprosessor i 1971
      (kanskje tidenes mest imponerende og viktigste industrielle revolusjon)



- Intell 4004, mikro-chip
  - 1971
  - 2300 transistorer
  - 108kHz klokke
  - max 648 byte minne
  - 4 bit bus
- Pentium 4
  - >42 000 000 transistorer
  - >3GHz klokke



- Intel core 7i
  - Eksempel:
    "Gulftown" 1.17 milliarder transistorer



✗ Teknologi-utviklingen:

- Moore's lov:
  Max antall transistorer på en chip dobles/18 mnd.

- ✗ Kunstig hjerne
  - Den "uungåelige" sammenligningen:

Mikrochips	Den menneskelige hjerne
Design: Synkron binær logikk	Design: Asynkron binær logikk
Virkemåte: Elektrisk strøm styrt av transistorer	Virkemåte: Elektrisk strøm styrt av nevroner
Antall “brytere”: Singlechip: >3 000 000 000 transistorer MCM: >100*3 000 000 000 transistorer	Antall “brytere”: Vanlig hjerne: >70 000 000 000 nevroner

  - Kan vi lage kunstige hjerner?
    - 2011:
      Teknologi --> ok
      Kan sette sammen flere 1000 prosessorer og minnebrikker på kretskort (tynnfilm) (evt. Wafer scale)
    - Utfordring: Design/arkitektur
      Vet ikke nok om arkitekturen til hjernen, enda...
    - Løsning?
      Selvorganiserende design, kunstig evolusjon
- ✗ Definisjoner:
  - VLSI (Very-Large-Scale-Integrated-Cicuits)
    - Mer enn 100 000 porter på samme chip
  - LSI (Large-Scale-Integrated-Cicuits)
    - Noen få tusen porter på samme chip
  - SSI (Small-Scale-Integrated-Circuits)
    - Noen få portes på samme chip

- ✗ Digitale design-metoder I
  - Hardware basert design:
    - SSI design:
      - Setter sammen SSI pakker på kretskort
    - FPGA (Field Programmable Gate Array)
      - Programmerbar logikk, eks.
        Xilinx, Altera, osv..
    - ASIC (Application Specified Integrated Circuit)
      - Skreddersydd logikk (designer på transistor-nivå)
      - Strømforbruk/hastighet/areal/spesialfunksjoner/pris/
        kombinert analog-digital/andre spesielle formål
- ✗ Digitale design-metoder II
  - Software/hardware basert design:
    - MicroController
      - Mikroprosessor med tilhørende I/O og minne på en brikke (datamaskin på en brikke)
    - DSP (Digital Signal Processor)
      - Microcontroller spesialbygd for rask signalbehandling, eks. video, audio, osv..
    - PC / mobiltelefon / ...
      - Digitale operasjoner kan utføres ved hjel av I/O hardware

TALLSYSTEMER

- ✗ Desimale tall:
  Et desimalt tall er representert ved symbolene 0, 1, 2, ..., 9
  - kodingen er posisjons-bestemt

  Eksempel:

  $$(7392)_{dec} = 7 \cdot 10^3 + 3 \cdot 10^2 + 9 \cdot 10^1 + 2 \cdot 10^0$$

- ✗ Binære tall:
  Tall må generelt ikke representeres ved 10 symboler (antall fingre)
  Eksempel: binøre tall
  Et binært tall representert ved symbolene 0 og 1
  - kodingen er posisjons-bestemt

  Eksempel:

  $$(101)_{bin} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

✗ Oktale tall
Et oktalt tall er representert ved symbolene 0, 1, 2, ..., 7

- kodingen er posisjonsbetinget med grunntall 8

Eksempel:

$$(252)_{okt} = 2\cdot8^2 + 5\cdot8^1 + 2\cdot8^0$$

✗ Heksadesimale tall
Et heksadesimalt tall er representert ved symbolene 0, 1, 2, ..., 8, 9, A, B, C, D, E, F

- kodingen er posisjonsbetinget med grunntall 16

Eksempel:

$$(2B9)_{heks} = 2\cdot16^2 + 11\cdot16^1 + 9\cdot16^0$$

✗ Telling:

Desimal	Binær	Oktal	Heksadesimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

✗ Generelt:

$$(...a_2a_1a_0,a_{-1}a_{-2}...)r = ... + a_2\bullet r^2 + a_1\bullet r^1 + a_0\bullet r^0 + a_{-1}\bullet r^{-1} + a_{-2}\bullet r^{-2} + ...$$

Eksempel:

$$(1A5,1C)_{16} = 1\bullet16^2 + 10\bullet16^1 + 5\bullet16^0 + 1\bullet16^{-1} + 12\bullet16^{-2} = (421,1133)_{des}$$

✗ Konvertering fra desimal til binær

- Prosedyre:
  1. Del det desimale tallet på 2
  2. Resten etter divisjon, multiplisert med 2 blir LSB*
  3. Del det nye desimale tallet på 2
  4. Resten etter divisjon, multiplisert med 2 blir neste bit
  5. Osv.

  (* Least significant bit)
- Eksempel:

  Konverter tallet $(41)_{des}$ til binær:

41/2	=	20 + 1/2	$a_0 = 1$	LSB
20/2	=	10 + 0/2	$a_1 = 0$	
10/2	=	5 + 0/2	$a_2 = 0$	
5/2	=	2 + 1/2	$a_3 = 1$	
2/2	=	1 + 0/2	$a_4 = 0$	
1/2	=	0 + 1/2	$a_5 = 1$	

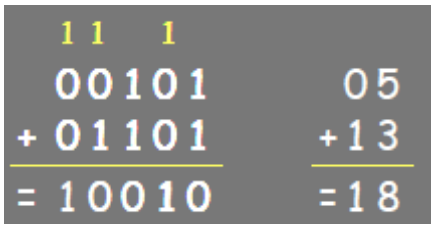
  Dermed: $(41)_{des} = (101001)_{bin}$

✗ Konvertering fra desimal til grunntall "r"

- Gjenta prosedyren fra forrige side. Bytt ut grunntallet 2 med r.
  Resten multiplisert med r blir det aktuelle sifferet

✗ Binær addisjon

- Prosedyren for binær addisjon er identisk med prosedyren for desimal addisjon
- Eksempel
  - Adder 5 og 13:



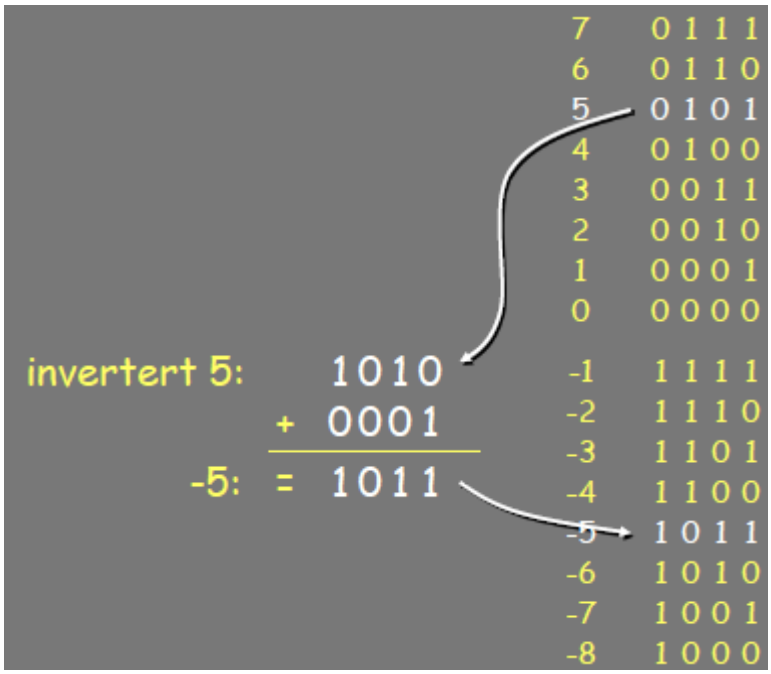
✗ Negative binære tall

- Mest vanlig representasjon:
  2'er komplement (signed)
  - Lar mest signifikante bit være 1 for negative tall
  - Dette må være "avtalt" på forhånd
  - Eksempel:
    4 bit kan representere tallene -8 til +7

7	0 1 1 1	-1	1 1 1 1
6	0 1 1 0	-2	1 1 1 0
5	0 1 0 1	-3	1 1 0 1
4	0 1 0 0	-4	1 1 0 0
3	0 0 1 1	-5	1 0 1 1
2	0 0 1 0	-6	1 0 1 0
1	0 0 0 1	-7	1 0 0 1
0	0 0 0 0	-8	1 0 0 0

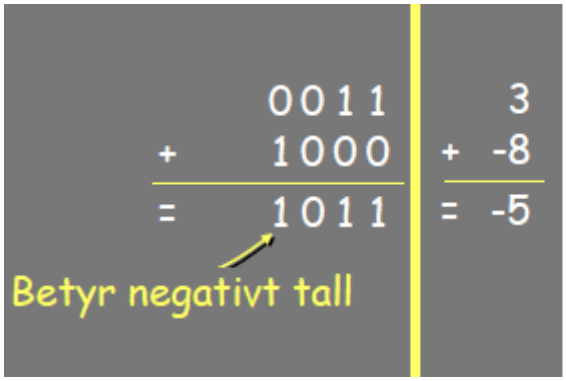
✗ 2'er komplement (signed)

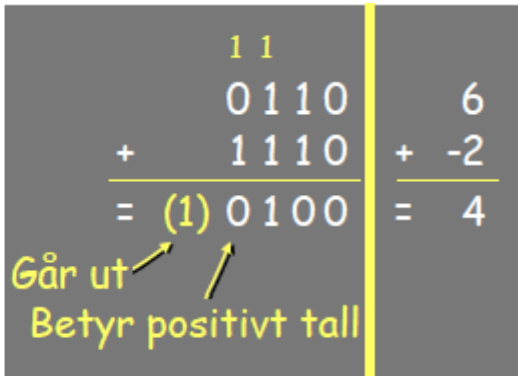
- Setter minus foran et binært tall ved å invertere alle bit'ene og plusse på 1
- Eksempel:
  Finner -5



✗ Binær substraksjon

- Fremgangsmåte for tall representert ved 2'er kompliment:
  - adder tallene på vanlig måte.
- Eksempel:
  6 – 2





✗ Utvidelse av antall bit

- For å øke antall bit på et "signed" binært tall kopierer man MSB til de nye plassene
- Eks 4 til 6 bit:

0110	->	0000110
1011	->	111011

✗ BCD-kode

- Binary coded decimal (BCD)
  - En kode som er en mellomting mellom binærkode og desimalkode
  - Lett å visualisere flere siffer på desimale display

Desimal	BCD
00	00000000
01	00000001
02	00000010
03	00000011
04	00000100
05	00000101
06	00000110
07	00000111
08	00001000
09	00001001
10	00010000
11	00010001
12	00010010
⋮	⋮
21	00100001
22	00100010
23	00100011
⋮	⋮
51	01010001
52	01010010
53	01010011

- ✗ Gray-kode
  - Kun ett bit forandrer verdi når tallet inkrementeres/dekrementeres
  - Spesielt gunstig ved overførsel av data mellom asynkrone system hvis man ikke har request/acknowledge

Desimal	Gray
0 0	0 0 0 0
0 1	0 0 0 1
0 2	0 0 1 1
0 3	0 0 1 0
0 4	0 1 1 0
0 5	0 1 1 1
0 6	0 1 0 1
0 7	0 1 0 0
0 8	1 1 0 0
0 9	1 1 0 1
1 0	1 1 1 1
1 1	1 1 1 0
1 2	1 0 1 0
1 3	1 0 1 1
1 4	1 0 0 1
1 5	1 0 0 0

KAPITTEL 2

Boolsk algebra og logiske porter

- ✗ Algebra, definisjon
  - Algebraisk system:
    - Mengde av elementer
    - Et sett av operasjoner på elementene (funksjoner) som tilfredsstiller en liste av aksiomer/postulater
  - Aksiom/postulat:
    Regler som ikke skal bevises (utgangspunkt for algebraisk system)
  - Eksempler på algebraiske system:
    "vanlig" algebra, Boolsk algebra, matrise algebra, osv...
  - Algebra – studiet av algebraiske system

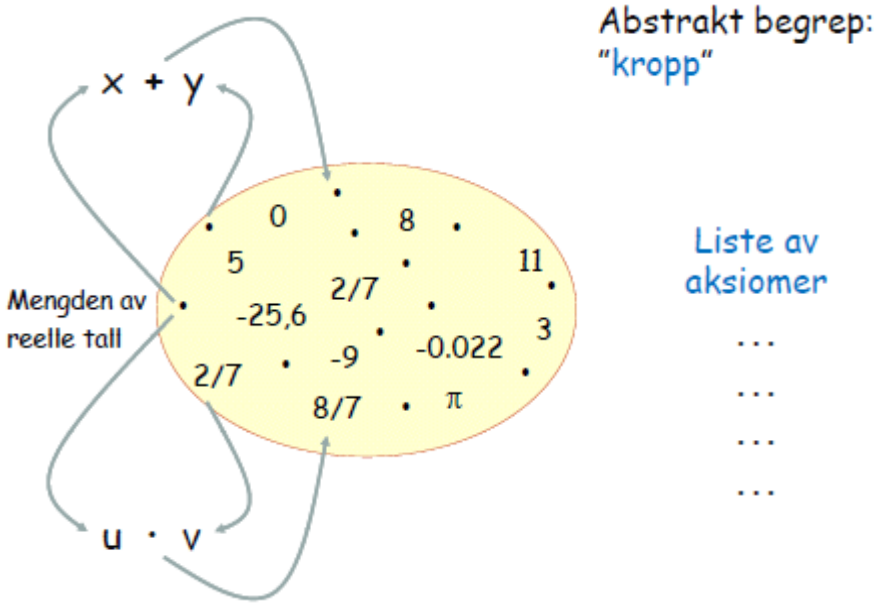
✗ Digitale system, anvendelser

Vanlig regning med tall:	Logiske operasjoner:
Beskrives av "vanlig algebra" Eksempler: • ALU-PC • Signalprosessorer • osv... NB: "vanlig algebra" inkluderer også binære tall, siden eneste forskjell er representasjonen	Beskrives av Boolsk algebra Eksempler: • Generering av binære utfall gitt ett sett med binære hendelser • Oppbygning av aritmetiske funskjoner

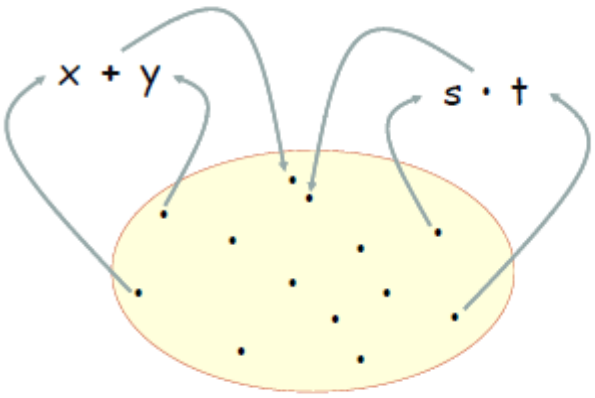
✗ Hvorfor trenger vi Boolsk algebra?

- Muliggjør design av komplekse digitale system
- Muliggjør analyse av komplekse digitale system
- Forenkling av logiske uttrykk
  - Gir enklere fysisk implementasjon

✗ Eksempel: "vanlig" algebra:



✗ Eksempel: Boolsk algebra:



Generell mengde av element

"+" og " •" er generelle operasjoner

Det fins mange Boolske algebraer

George Boole 1854

Liste av aksiomer (Huntington)

...

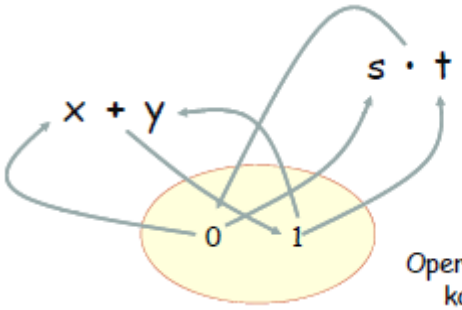
...

...

...

✗ "Toverdi" Boolsk algebra

Et konkret eksempel på Boolsk algebra (Shannon 1938)



Operatorene "+" og " •" kaller vi OR / AND

Mengden av elementene "0" og "1"

(C.E. Shannon 1938)

Huntington's postulater

...

...

...

...

✗ Huntington postulater – produserer toverdi Boolsk algebra

–	P1: Mengden {0,1} er lukket under "+" og "•"		lukket
–	P2: x + 0 = x	x • 1 = x	3 ident. el.
–	P5: x + x' = 1	x • x' = 0	3 komplem.
–	P6: 0 ≠ 1		minst 2 el.
–	P3: x + y = y + x	x• y = y • x	kommutativ
–	P4: x•(y+z) = x•y + x•z	x+(y•z) = (x+y)•(x+z)	distributiv

Dualitet for postulatene: kan bytte ”•” med ”+” hvis man bytter ”0” med ”1”
Presedens: først utføres ”( )”, så ”•” og til slutt ”+”

✗ Toverdi Boolsk algebra i praksis:
Hva gjør operatorene +/•

– Sannhetstabell:

AND

x y	x•y
0 0	0
0 1	0
1 0	0
1 1	1

OR

x y	x+y
0 0	0
0 1	1
1 0	1
1 1	1

– Husk! +/• har ingenting med addisjon og multiplikasjon å gjøre

– Vi kan velge å se på eksistens av identitetselement som en mapping fra x til x', får da en ny operator vi kan kalle NOT

NOT

x	x´
0	1
1	0

✗ Basis-byggeblokker for våre digitale system:

AND

X Y	Z
0 0	0
0 1	0
1 0	0
1 1	1

OR

X Y	Z
0 0	0
0 1	1
1 0	1
1 1	1

NOT

X	Y
0	1
1	0

(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

✗ Teorem/postulatliste:

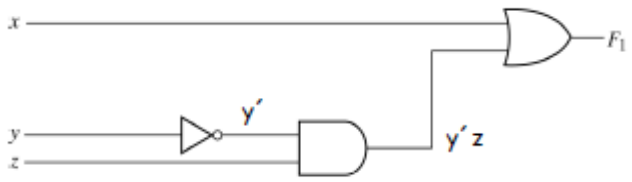
P2	x + 0 =x	P2	x • 1 = x
P5	x + x' = 1	P5	xx' = 0
P3	x + y = y + x	P3	xy = yx
	x + (y + z) = (x + y) + z		x (yz) = (xy) z
P4	x(y + z) = xy + xz	P4	x + (yz) = (x + y)(x + z)
T1a	x + x = x	T1b	x • x = x
T2a	x + 1 = 1	T2b	x • 0 = 0
	x + xy = x		x(x + y) = x
DeMorgans	(x + y)' = x'y'	DeMorgans	(xy)' = x' + y'

✗ Boolske funksjoner:

- Tilordner en binær variabel en verdi bestemt av verdien på en eller flere andre binære variabler
- Eksempel:

  $F_1 = x + y'z$

  Direkte port-implementasjon:



✗ Sannhetstabell:

- En boolsk funksjon kan visualiseres i en sannhetstabell
  Eksempel:

  F = x + y'z

- En gitt funksjon har kun en sannhetstabell

  Men, en gitt sannhetstabell har uendelig

  mange funksjonsuttrykk

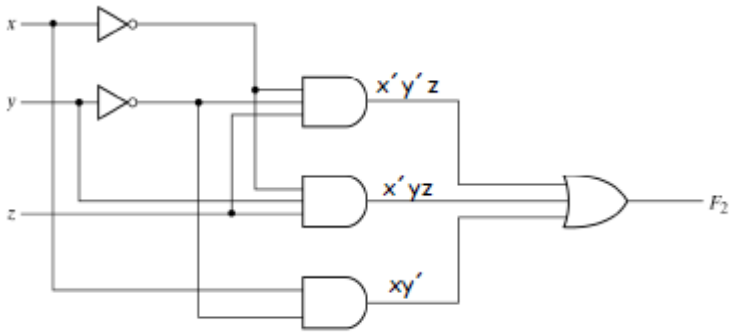
  Stikkord: forenkling av funksjonsuttrykk

x y z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

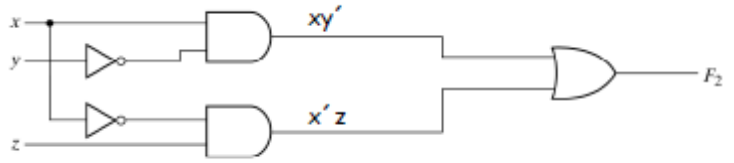
✗ Forenkling av uttrykk:

Eksempel:	Eksempel:	Eksempel:	Eksempel:	Eksempel:
F = x(x'+y)	F = x+x'y	F = (x+y)(x+y')	F = xy+x'z+yz	F = (x+y)(x'+z)(y+z)
F = xx'+xy	F = (x+x')(x+y)	F = x + (yy')	F = xy+x'z+yz(x+x')	F = (x+y)(x'+z) Dualitet
F = 0+xy	F = 1(x+y)	F = x + 0	F = xy+x'z+xyz+x'yz	
F = xy	F = x+y	F = x	F = xy(1+z)+x'z(1+y)	
			F = xy+x'z	

✗ Port-implementasjon av F_2



(a) $F_2 = x'y'z + x'yz + xy'$



(b) $F_2 = xy' + x'z$

✗ Definisjoner: minterm

– I en funksjon kan en binær variabel x opptre som x eller x'
En funksjon kan være gitt på "sum av produkt" form
Eksempel:

F = xy + xy' + x

Hvert "produktledd" som inneholder alle variablene, kalles en minterm

For to variable fins det 2^2 forskjellige mintermer:
xy + xy' + x'y + x'y'

For 3 variable fins det 2^3 forskjellige mintermer:
xyz + xyz' + xy'z + xy'z'+x'yz + x'yz' + x'y'z + x'y'z'

✗ Definisjoner: maksterm

- En funksjon kan være gitt på ”produkt av sum” form
  Eksempel:

$$F = (x+y)(x+y')y$$

Hvert ”summeledd” som inneholder alle variablene kalles en maksterm

For to variable fins det 4 forskjellige makstermer:
(x+y)(x+y’)(x’+y)(x’+y’)

For n variable fins det 2^n forskjellige makstermer

✗ Sannhetstabell/mintermer

- Hvis man genererer en funksjon ut i fra sannhetstabellen får man en sum av mintermer
- Eksempel:

$$F = x'y'z + xy'z' + xyz'$$

x y z	F
0 0 0	0
0 0 1	**1**
0 1 0	0
0 1 1	0
1 0 0	**1**
1 0 1	0
1 1 0	**1**
1 1 1	0

- En sannhetstabell kan sees på som en liste av mintermer.

✗ Notasjon

- Mintermer / maxtermer har notasjon m_x / M_x

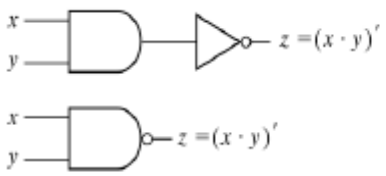
			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

- Funksjoner som bare består av sum/produkt av min/maxtermer (kanonisk form) har følgende notasjon.

  Eksempel:

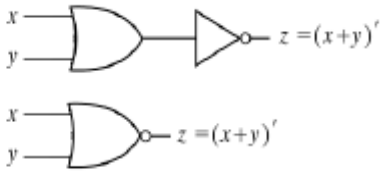
$$F(x,y,z) = \Sigma(m_3, m_6) = \Sigma(3, 6) = x'yz+xyz'$$
$$F(x,y,z) = \Pi(M_3, M_6) = \Pi(3, 6) = (x+y'+z')(x'+y'+z)$$

✗ NAND / NOR



(a) Two-input NAND gate

NAND	
XY	Z
0 0	1
0 1	1
1 0	1
1 1	0



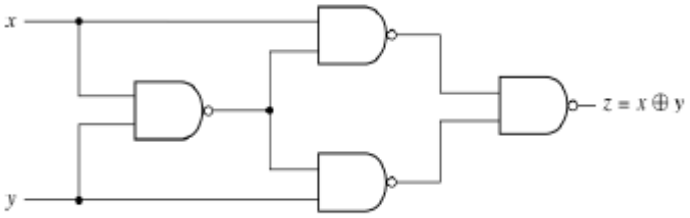
(a) Two-input NOR gate

NOR	
XY	Z
0 0	1
0 1	0
1 0	0
1 1	0

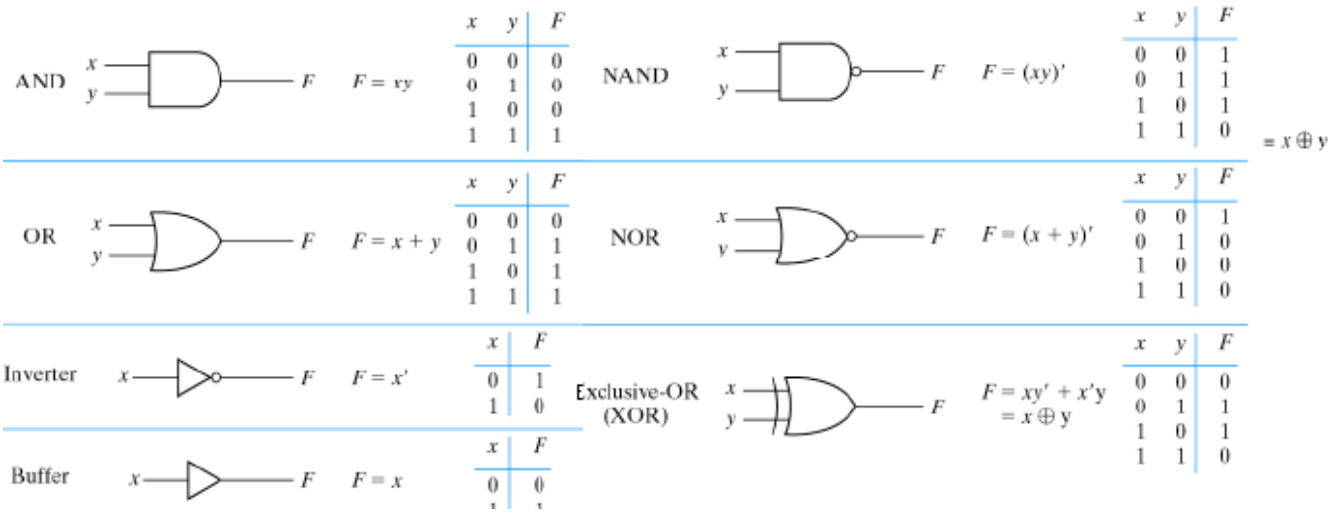
✗ XOR

Exclusive-OR Implementations

XOR	
XY	Z
0 0	0
0 1	1
1 0	1
1 1	0



✗ 2-inputs byggeblokker, oversikt:



✗ Forenkling på portnivå:
Det er ikke alltid at det enkleste funksjonsuttrykket resulterer i den enkleste port-implementasjonen.
Ved forenkling på portnivå må man vite hvilke porter man har til rådighet, og så justere funksjonsuttrykket mot dette.
(håndverk)

✗ Generell design prosedyre

1. Bestem hvilke signal som er innganger og utganger
2. Sett opp sannhetstabell for alle inngangskombinasjoner
3. Generer funksjonsuttrykket som sum av mintermer
4. Tilpass /forenkle funksjonsuttrykket mot aktuelle porter

KAPITTEL 3

Karnaughdiagram

✗ Karnaughdiagram

Grafisk metode for forenkling av Boolske uttrykk

- Uttrykket må være representert ved sumav mintermer (mx).Disse leser vi direkte ut av sannhetstabellen
- Metoden egner seg for funksjoner med 2-4(5) variable

– Eksempel, 2 variable:

- $F= m_1+ m_3= a'b+ ab$

– Eksempel, 4 variable:

- $F =m_0+m_1+m_{15} = A'B'C'D' + A'B'C'D + ABCD$

✗ Don't care
I noen tilfeller har man inngangskombinasjoner som aldri dukker opp.
I andre tilfeller bryr man seg ikke om utfallet for visse inngangskombinasjoner.

Slike kombinasjoner kalles don't care kombinasjoner og markers med "X"

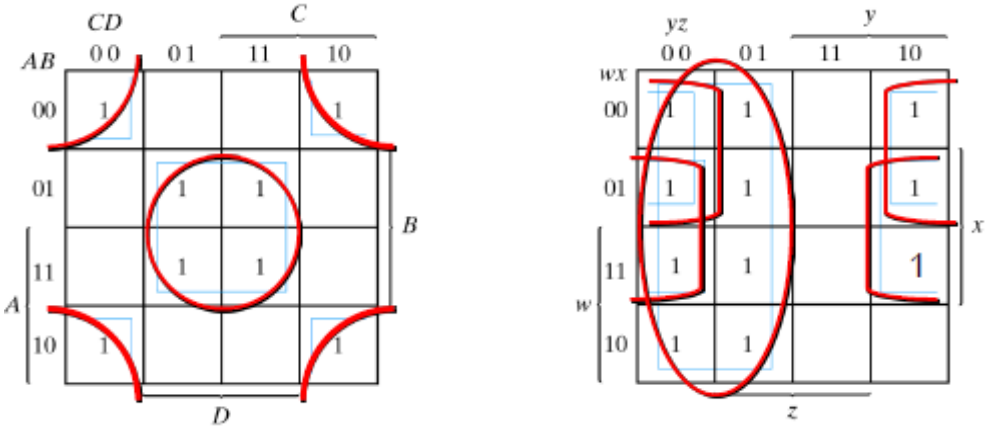
✗ Grupperingsregler:

Velg så store grupper av elementer som mulig. Antall elementer må være potens av 2.

Du kan velge å gruppere 1'erne og få F eller 0'erne og få F'
("Don't care" kan grupperes etter eget ønske).

Mintermene plasseres slik at kun 1 variabel varierer i mellom hver vannrette/loddrette naborute

Eksempel:



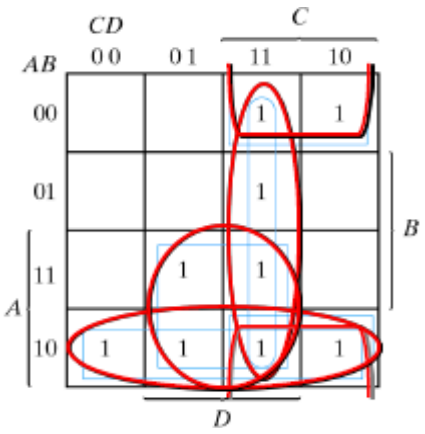
✗ Utlesningsregler:

Les av hver gruppe slik at den mintermen som ikke varierer gjelder.

Diagrammets funksjon blir da summen av hvert gruppeledd.

Jo større ruter/grupper desto enklere uttrykk

Eksempel:

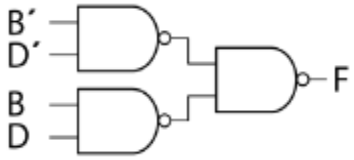


F = AD + AB' + CD + B'C

✗ Algebraisk reduksjon til NAND / NOR

- I noen tilfeller har man kun NAND og NOR kretser tilrådighet
- Eksempel:

  F =B'D'+BD

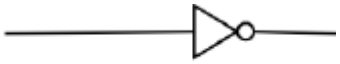


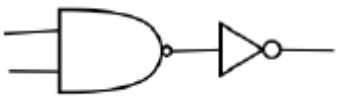
- DeMorgan:

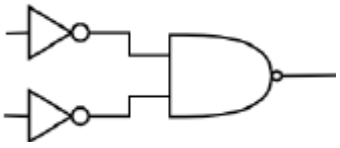
  F =((B'D')' • (BD)')'

✗ NAND reduksjon

- All kombinatorisk logikk kan implementeres ved hjelp av NAND

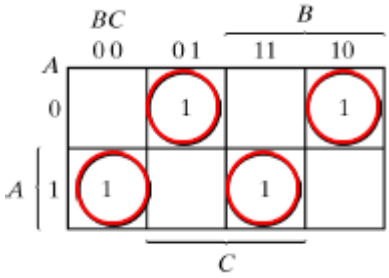
Inventerer 

AND 

OR 

Spesialteknikker

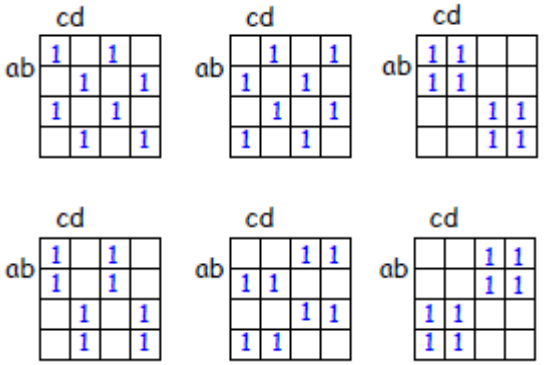
- ✗ XOR
  - XOR funksjonen dekker maksimalt "uheldige" 1'er plasseringer i diagrammet
    Har man XOR porter til rådighet bruker man disse
  - Eksempel:



(a) Odd function
$F = A \oplus B \oplus C$

Her kan 1 stk. 3-inputs XOR realisere F

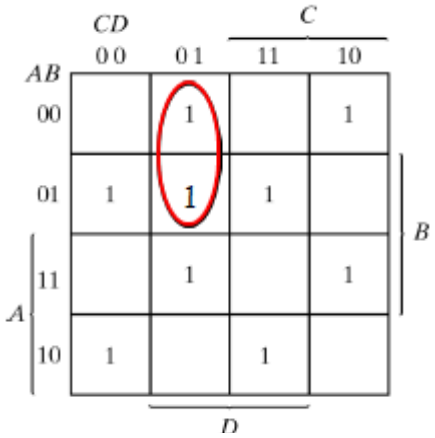
  - XOR av inverterte innganger, a', b' osv. gir andre diagramkonfigurasjoner
  - XOR av 3 eller 2 innganger i et 4-variabeldiagram gir nye konfigurasjoner osv.



  - XOR funksjonen kan kombineres med andre ledd.

    Eksempel:

    - F = A⊕B⊕C⊕D⊕ + A'C'D

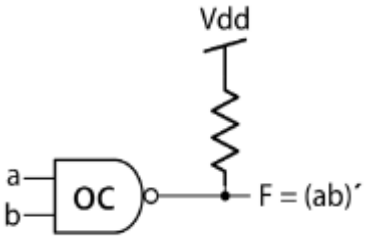


- XOR som paritetssjekker
  - Paritetsjekk bruker når det kan oppstå feil i overføringen av bit
  - En bit paritet kan enkelt implementeres med XOR, og detekterer 50 % av feilene
  - Even-parity:

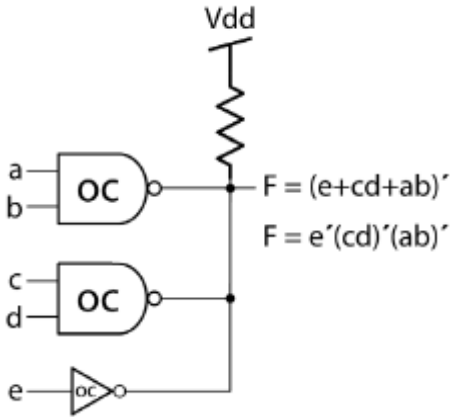
$$P = a_1 \oplus a_2 \cdots \oplus a_n$$

✗ Open drain (wired AND)

- En port med open drain(collector)-utgang kan bare trekke utgangsspenningen ned til Gnd / "0", ikke dra den opp til Vdd / "1"
- Fordel: Utganger fra flere porter kan kobles sammen (ingen logisk/elektrisk konflikt)
- Andre egenskaper:
  - For å kunne få "1" ut, bruker vi en ekstern motstand mot "Vdd"
  - Nye funksjoner blir generert i utgangsledningene. (Kan resultere i færre porter)
- Eksempel 1:



- Eksempel 2:
  Får 3-inputd NOR "gratis"
  Evt. Får 3-inputs AND "gratis"



KAPITTEL 4

Binær adder

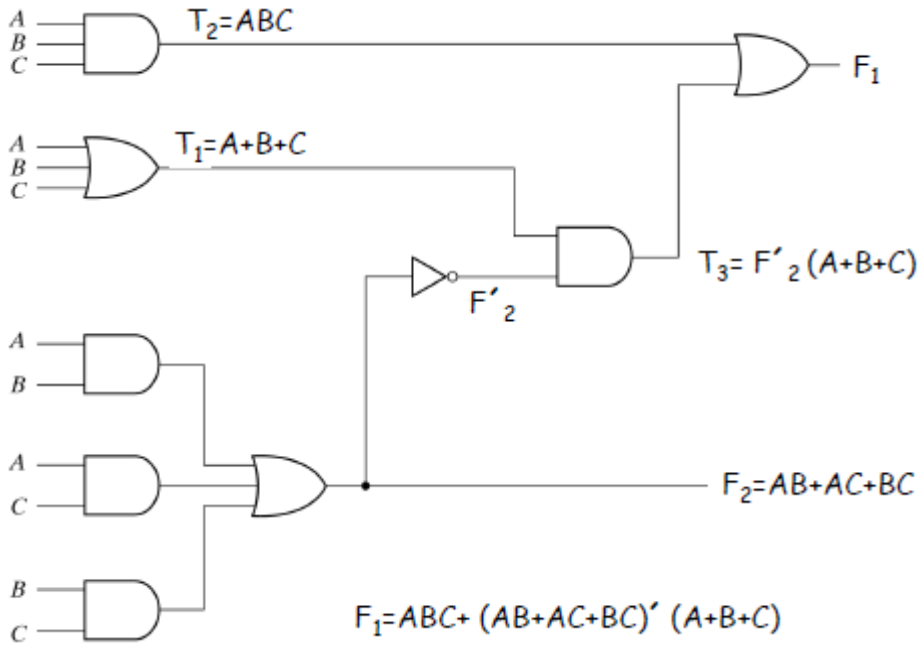
- ✗ Generell analyseprosedyre for digitale kretser

  1)Sett funksjonsnavn på ledningene

  2)Finn funksjonene

  3)Kombiner funksjonsuttrykkene

  - Eksempel:



- ✗ Binær adder
  - En av de mest brukte digitale kretser
  - Vanlige anvendelser:

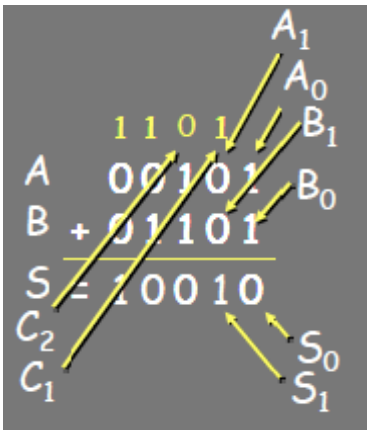
    Mikroprosessor ALU / Xbox / mikserbord / digitalt kommunikasjonsutstyr / AD-DA omformere osv...

    - Basis for addisjon / subtraksjon / multiplikasjon / divisjon og mange andre matematiske operasjoner
    - All form for filtrering / signalbehandling

- Ønsker å designe en generell binær adder

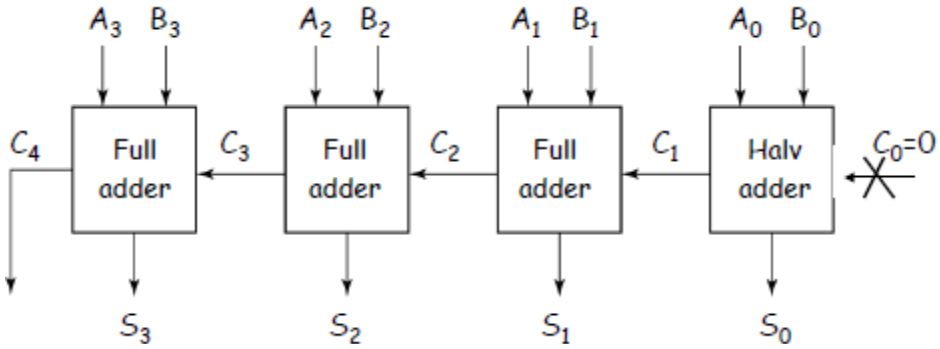
  Funksjonelt eksempel:

  Adder to tall A = 5 og B = 13:



✗ Et adder-system

- Systemelementer:
  - Halvadder: tar ikke inn mente
  - Fulladder: tar inn mente



✗ Halvadder (ingen mente inn)

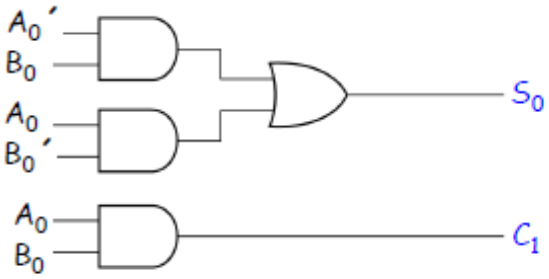
- Adderer sammen de to minst signifikante bit'ene A_0 og B_0. Elementet har 2 innganger og 2 utganger

  $S_0 = A_0'B_0 + A_0B_0' = A_0 \oplus B_0$

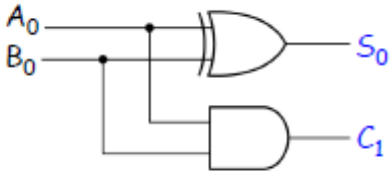
  $C_1 = A_0B_0$

A_0 B_0	S_0 C_1
0 0	0 0
0 1	1 0
1 0	1 0
1 1	0 1

- Halvadder implementasjon



$S_0 = A_0' B_0 + A_0 B_0'$
$C_1 = A_0 B_0$



$S_0 = A_0 \oplus B_0$
$C_1 = A_0 B_0$

✗ Fulladder (mente inn)

- Adderer sammen bit A_n , B_n med evt. mente inn

  Elementet har 3 innganger og 2 utganger

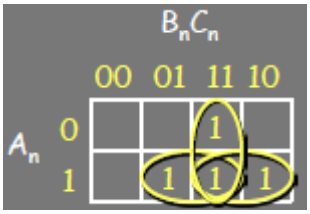
A_n	B_n	C_n	S_n	C_{n+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

  $S_n = A_n \oplus B_n \oplus C_n$ (oddefunksjon)

  $C_{n+1} = A_n'B_nC_n + A_nB_n'C_n + A_nB_nC_n' + A_nB_nC_n$

- Forenkling

  forenkler C_{n+1} vrd Karnaughdiagram

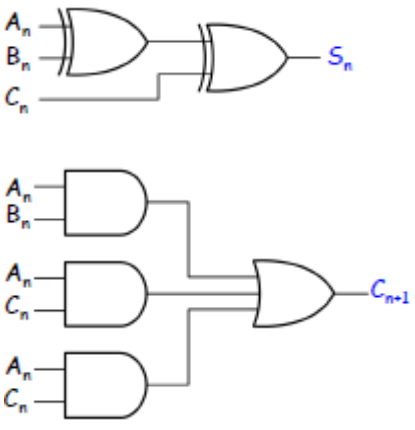


  $C_{n+1} = A_n'B_nC_n + A_nB_n'C_n + A_nB_nC_n' + A_nB_nC_n$
  $C_{n+1} = A_nB_n + A_nC_n + B_nC_n$

- Implementasjon I

  Rett fram implementasjon

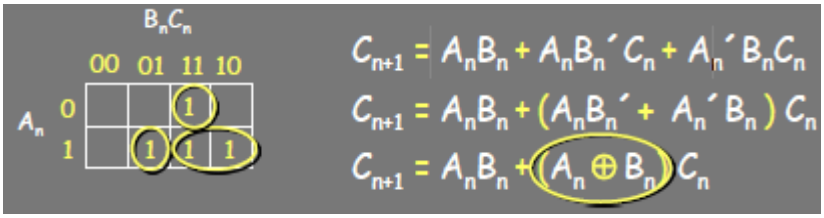
  - $S_n = A_n \oplus B_n \oplus C_n$
  - $C_{n+1} = A_nB_n + A_nC_n + B_nC_n$



- Implementasjon II

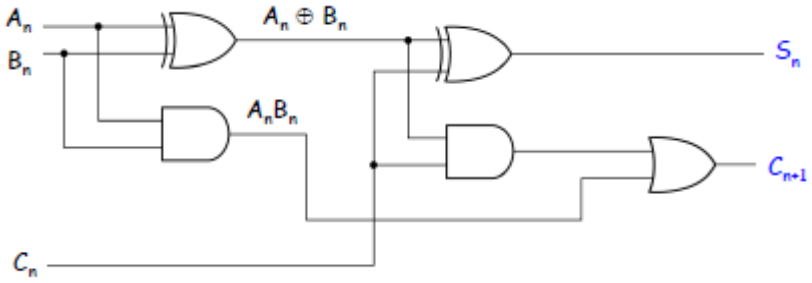
  Forenklet implementasjon av C_{n+1} basert på gjenbruk av porter fra S_n

  Leser ut C_{n+1} fra karnaughdiagram på nytt

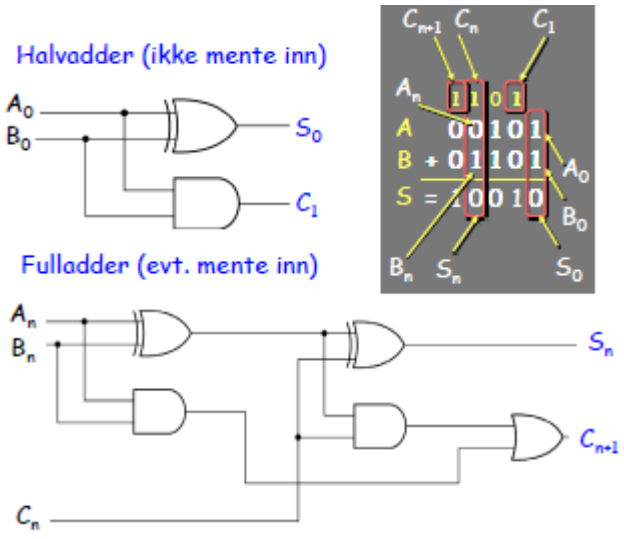


  Vanlig implementasjon av en-bits fulladder

  - $S_n = (A_n \oplus B_n) \oplus C_n$
  - $C_{n+1} = A_nB_n + (A_n \oplus B_n)\,C_n$

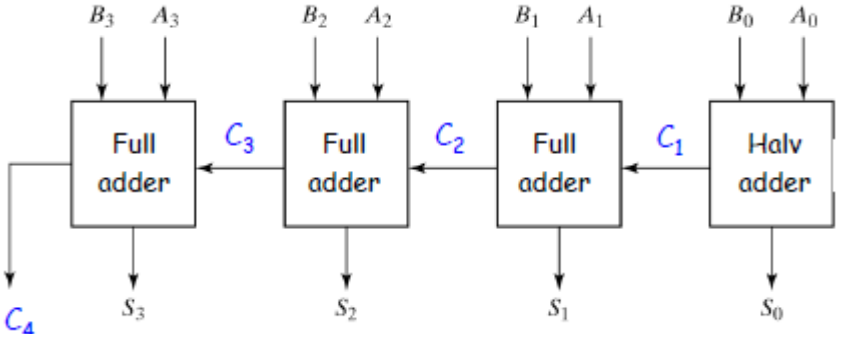


✗ Binær adder



✗ Menteforplantning

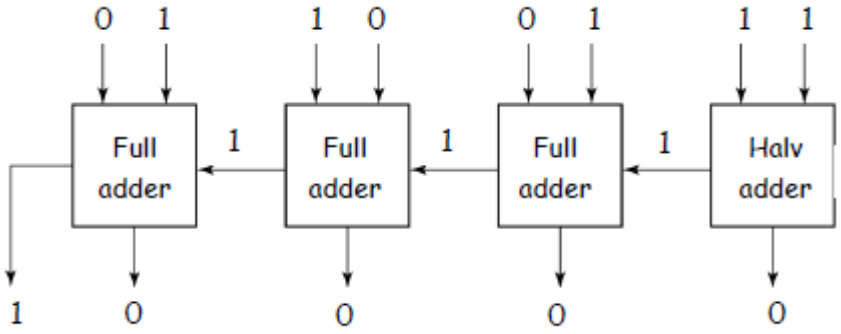
4-bits binær adder



- Portforsinkelse gir menteforplantning (rippeladder)

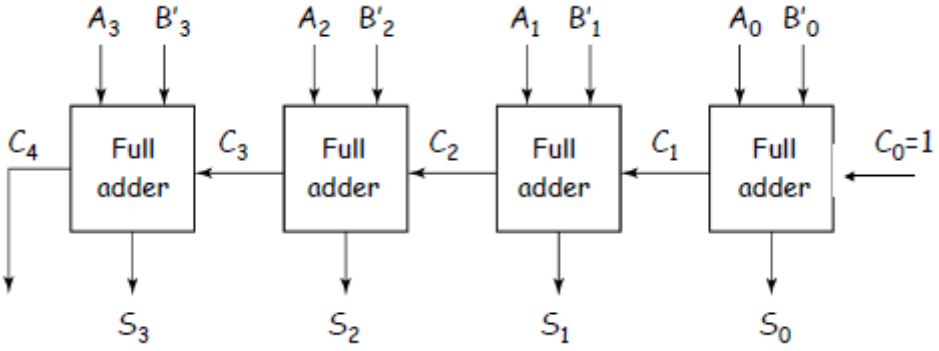
  Eksempel:

  Adderer 0101 og 1011



✗ Binær Substraksjon

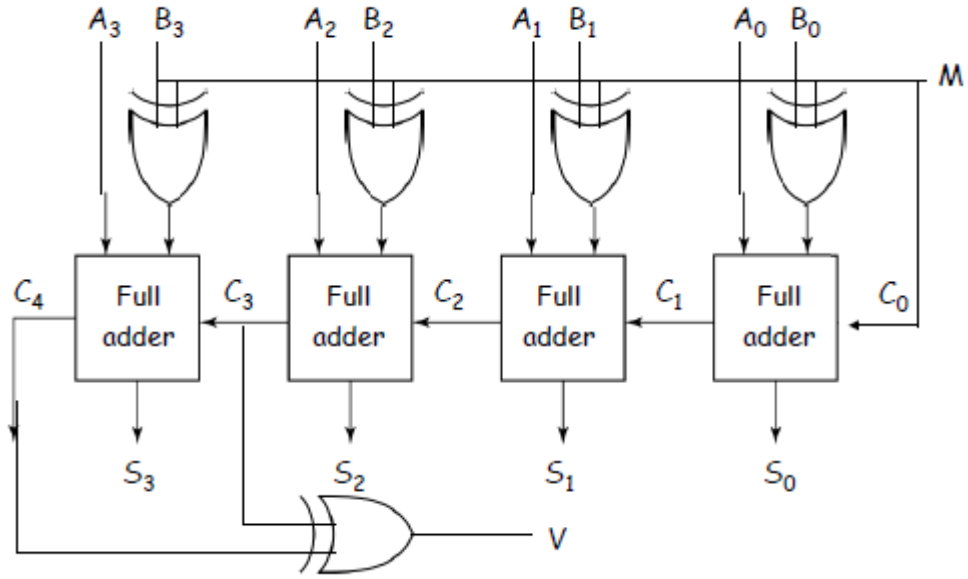
- For å regne ut A-B kan man bruke en adderer krets med noen modifikasjoner
- Man må ta 2'ers komplement av B, dvs. B invers + 1.
  Man kan legge til en ved å sette C_0= 1



✗ Binær adder/subtrakter

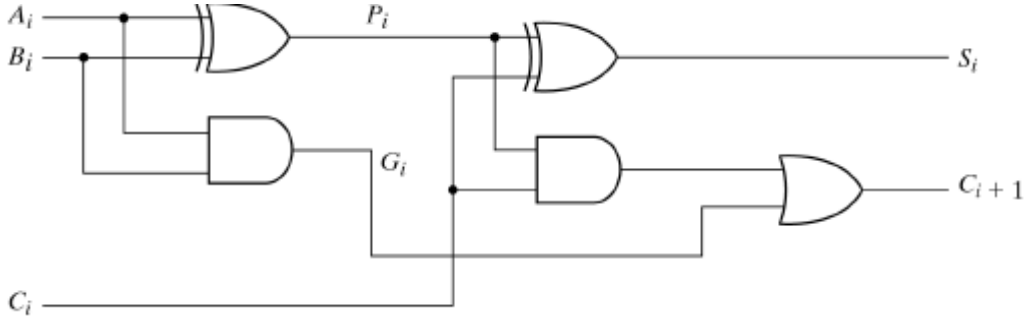
M : 0 --> addisjon , 1 --> substraksjon

V : overflyt hvis 1



✗ "Carry Lookahead"

- Ønsker å unngå menteforplantning – gir økt hastighet
  - G_i – generate: brukes i menteforplantningen
  - P_i – progagate: påvirker ikke menteforplantningen



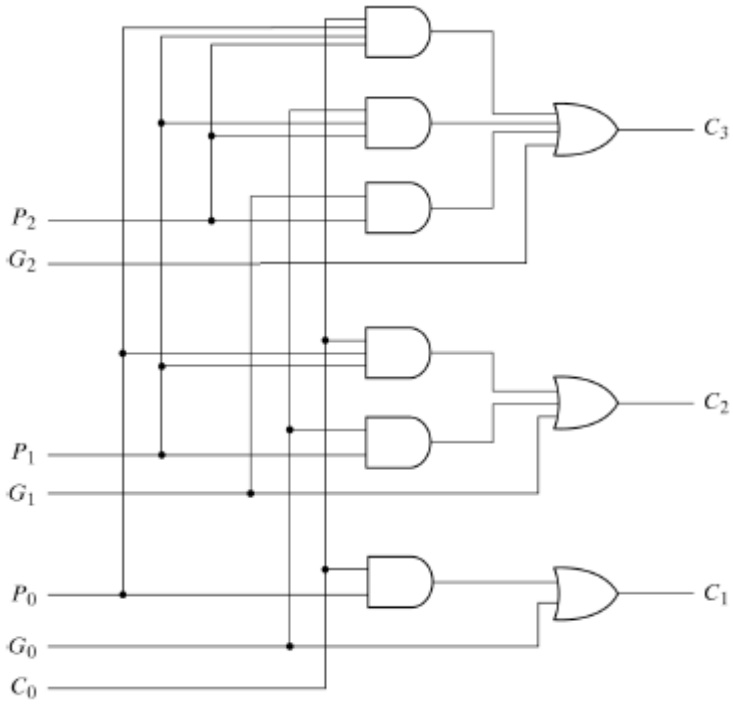
  - $S_i = P_i \oplus C_i$
  - $C_{i+1} = G_i + P_iC_i$
- For en 4-bits adder bestående av 4 fulladdertrinn har vi:
  - $S_i = P_i \oplus C_i$
  - $C_{i+1} = G_i + P_iC_i$

  Uttrykker C_1, C_2 og C_3 rekursivt
    - $C_1 = G_0 + P_0C_0$

- $C_2 = G_1 + P_1C_1 = G_0 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$
- $C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

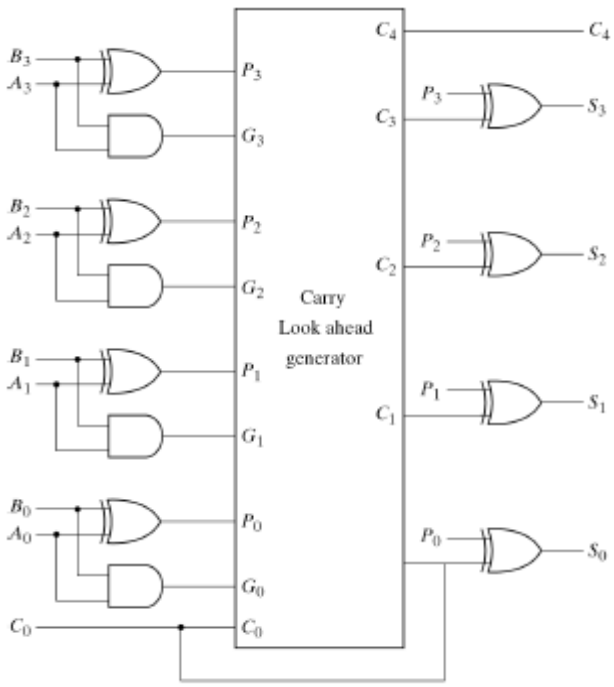
✗ "Carry Lookahead" generator

– Rett fram implementasjon av C_1, C_2, C_3



✗ "Carry Lookahead" adder

– 4-bits Carry Lookahead adder med input carry C_0



✗ Komparator

- Komparator – sammenligner to tall A og B
  - 3 utganger: A=B, A>B og A<B
- Eksempel: 4-bits komparator
  - Utgang A=B
    - Slår til hvis $A_0 = B_0$ og $A_1 = B_1$ og $A_2 = B_2$ og $A_3 = B_3$
    - Kan skrives: $(A_0 \oplus B_0)'(A_1 \oplus B_1)'(A_2 \oplus B_2)'(A_3 \oplus B_3)'$
  - Utgang A>B slår til hvis:

    $(A_3>B_3)$ eller

    $(A_2>B_2$ og $A_3=B_3)$ eller

    $(A_1>B_1$ og $A_2=B_2$ og $A_3=B_3)$ eller

    $(A_0>B_0$ og $A_1=B_1$ og $A_2=B_2$ og $A_3=B_3)$

    Kan skrives:

    $(A_3B_3') + (A_2B_2') (A_3 \oplus B_3)' + (A_1B_1') (A_2 \oplus B_2)'(A_3 \oplus B_3)' +$

    $(A_0B_0')(A_1 \oplus B_1)'(A_2 \oplus B_2)'(A_3 \oplus B_3)'$
  - Utgang A<B slår til hvis:

    (A3<B3) eller

    $(A_2<B_2$ og $A_3=B_3)$ eller

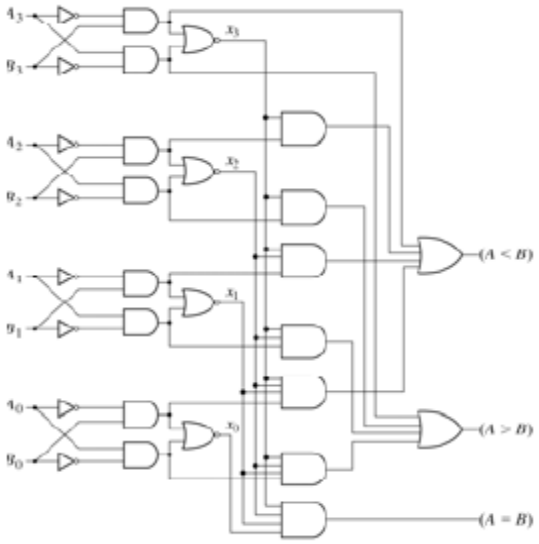
    $(A_1<B_1$ og $A_2=B_2$ og $A_3=B_3)$ eller

    $(A_0<B_0$ og $A_1=B_1$ og $A_2=B_2$ og $A_3=B_3)$

    Kan skrives:

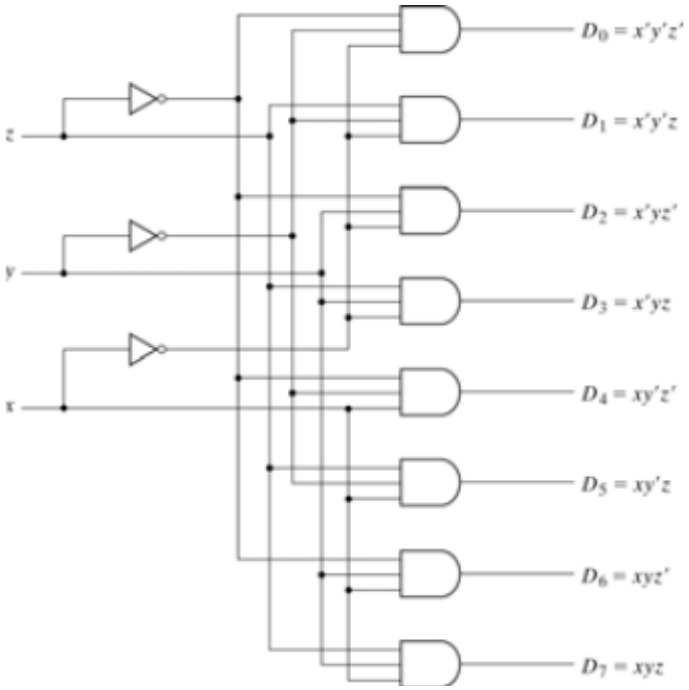
    $(A_3'B_3) + (A_2'B_2) (A_3 \oplus B_3)' + (A_1'B_1) (A_2 \oplus B_2)'(A_3 \oplus B_3)' +$

    $(A_0'B_0)(A_1 \oplus B_1)'(A_2 \oplus B_2)'(A_3 \oplus B_3)'$
- Komparator-eksempel:



✗ Dekoder:

- Dekoder – tar inn et binært ord, gir ut alle mintermer
- Eksempel:
  3 bit inn / 8 bit ut

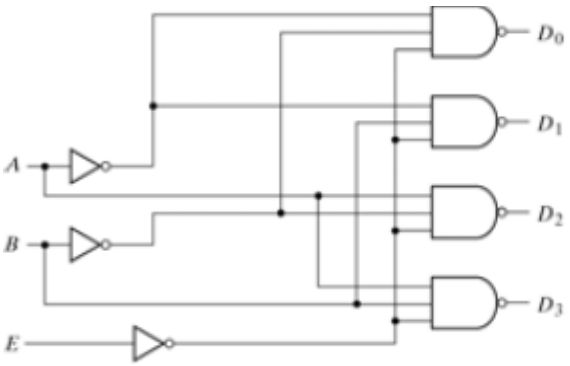


- Sannhetstabell:
  Eksempel: 3 bit inn

Innganger			Utganger							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- Varianter:
  - Enable input:
    - Enable aktiv --> normal operasjon
    - Enable inaktiv --> alle utganger disablet

    NAND logikk: Inverterte utganger



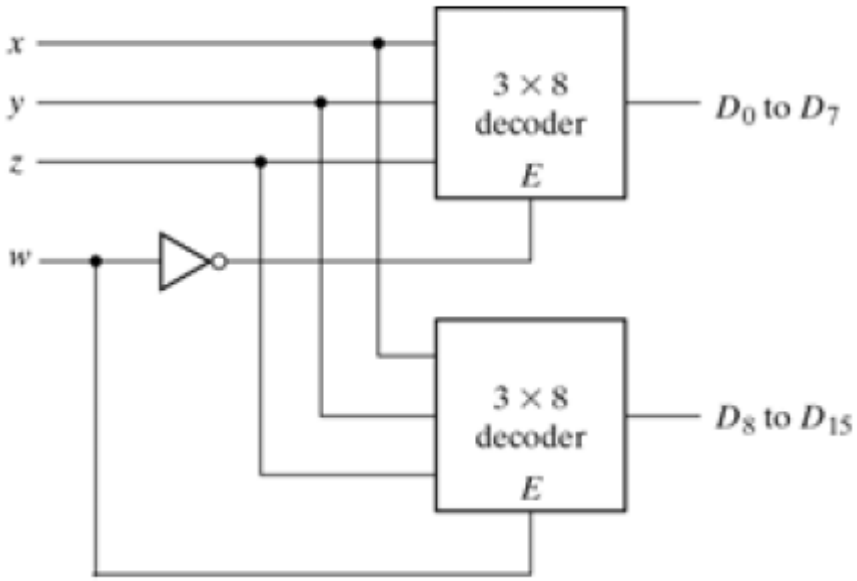
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Eksempel

Aktiv "lav" enable inngang

  - Parallellkobling

    Eksempel: Lager en 4x16 dekoder fra 2 stk 3x8 dekodere med enable innganger

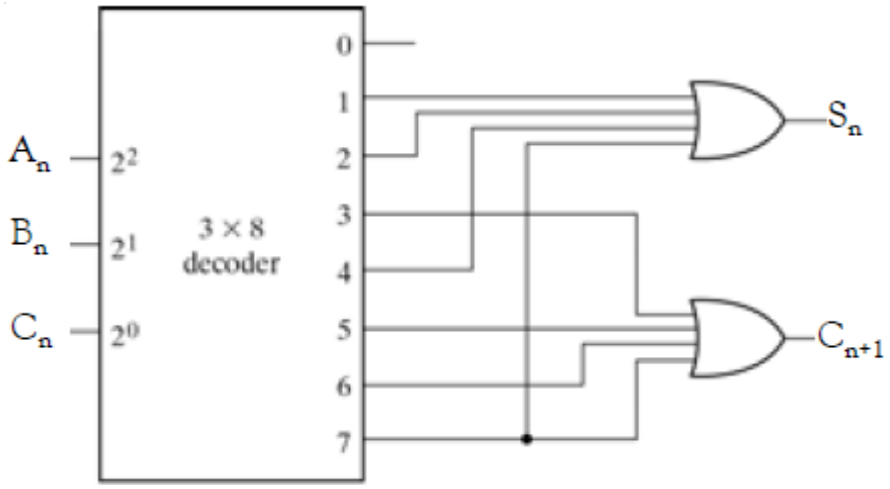


- Generering av logiske funksjoner:

  Dekoder – elektrisk sannhetstabell. Kan generere generelle logiske funksjoner direkte fra mintermene på utgangen

  Eksempel:

  Fulladder



✗ Enkoder (motsatt av dekoder)

- Eksempel: 8x3 enkoder

Innganger								Utganger		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$x = D_4 + D_5 + D_6 + D_7$

$y = D_2 + D_3 + D_6 + D_7$

$z = D_1 + D_3 + D_5 + D_7$

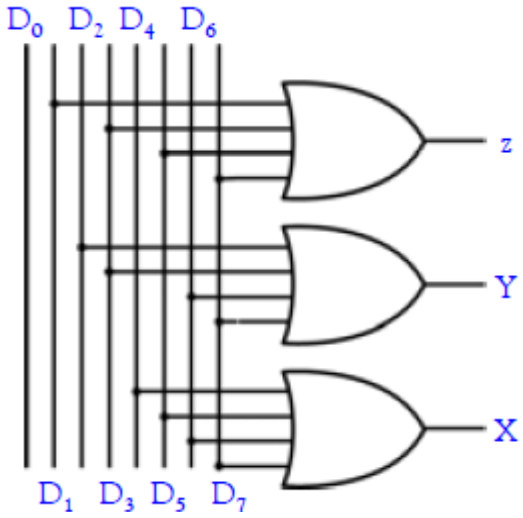
Antar at det ikke eksisterer andre inngangskombinasjoner

- Eksempel:

  $x = D_4 + D_5 + D_6 + D_7$

  $y = D_2 + D_3 + D_6 + D_7$

  $z = D_1 + D_3 + D_5 + D_7$



- Prioritets-enkoder

  Problem i enkodere: Hva hvis man får flere 1'ere inn samtidig?

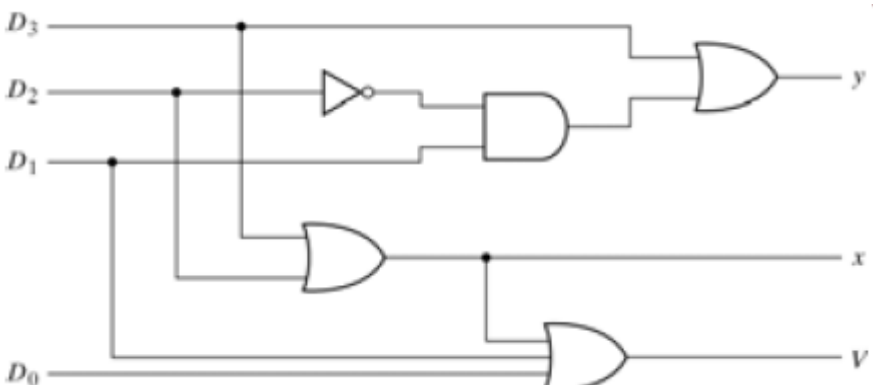
  Løsning: Prioritets-enkoder

  Hvis flere 1'ere inn ser kun på inngang med høyst indeks (prioritet)

  - Eksempel: 8x3 prioritets-enkoder

Innganger								Utganger		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
x	1	0	0	0	0	0	0	0	0	1
x	x	1	0	0	0	0	0	0	1	0
x	x	x	1	0	0	0	0	0	1	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	x	1	1	1	1

- Eksempel: 4x2 prioritets-enkoder med "valid" utgang

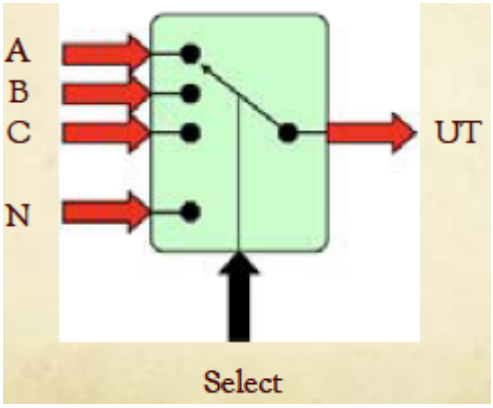


"V" signaliserer at minst en inngang er "1"

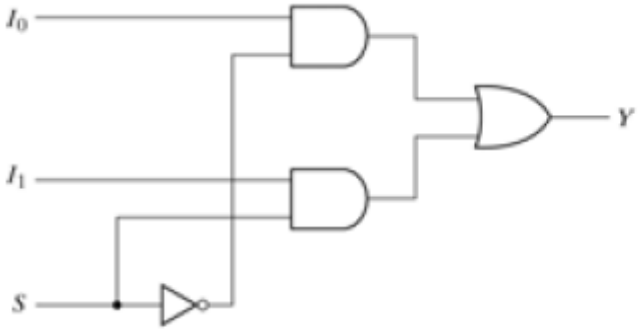
✗ Multiplekser

- Multiplekser (MUX) – velger hvilke innganger som slippes ut

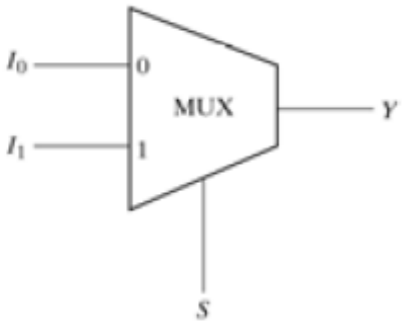
  Hver inngang kan bestå av ett eller flere bit



- Eksempel: 2-1 MUX

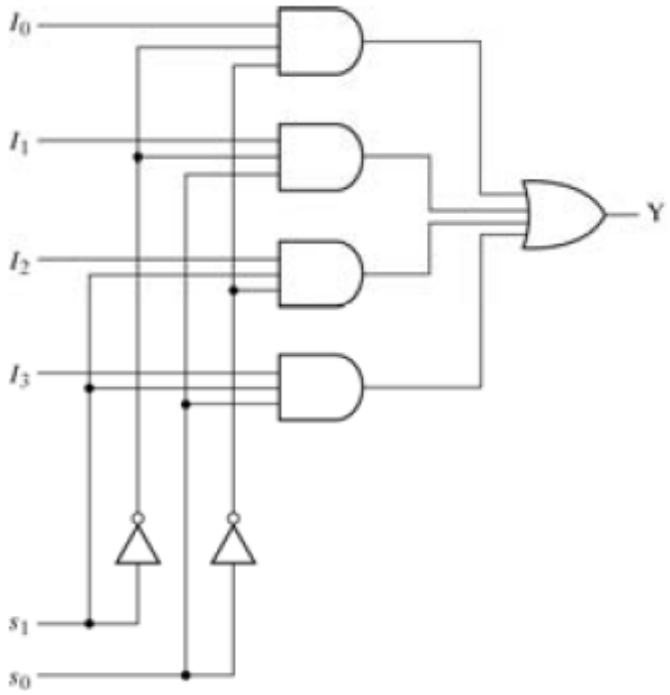


Implementasjon



Symbol

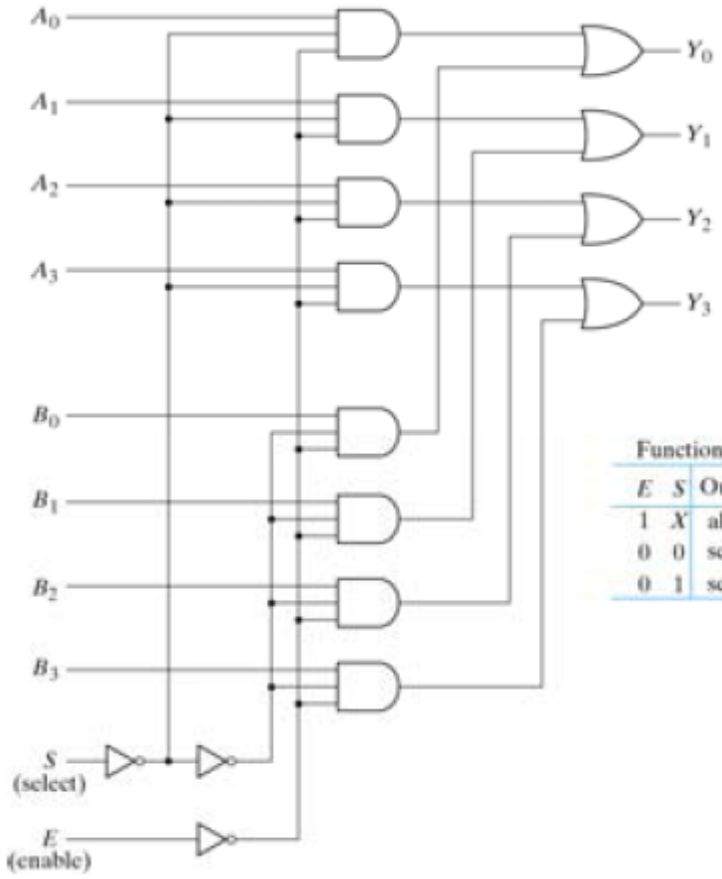
- Eksempel: 4-1 MUX



s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

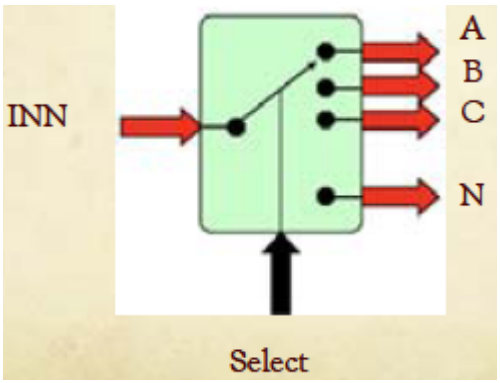
- Eksempel: 2-1 MUX



Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

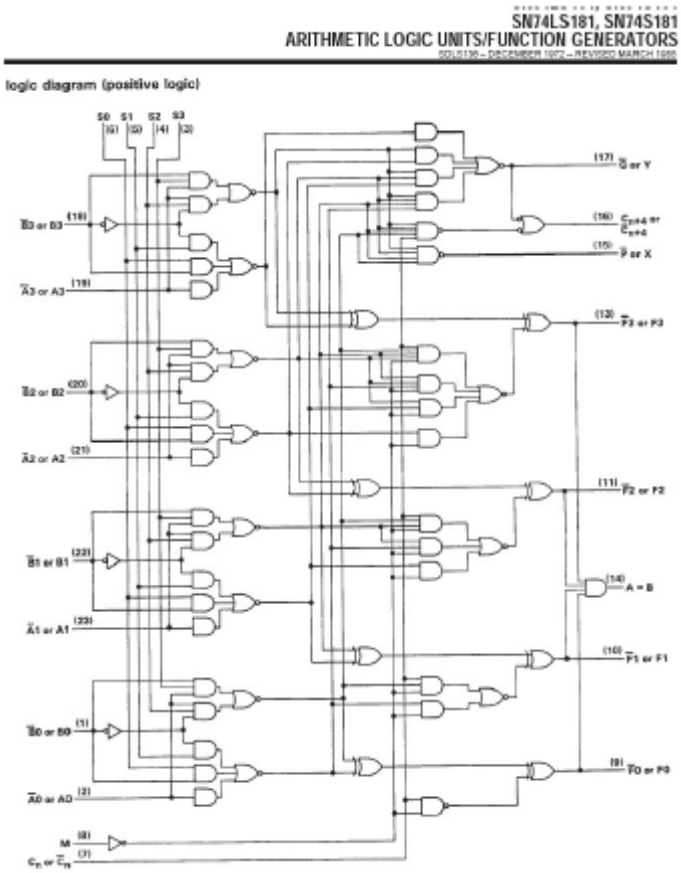
✗ Demultiplekser (motsatt av multiplekser)



CPU

✗ ALU (Aritmetic Logic Unit)

- Generell regneenhet
- Eksempel:
  SN74LS181
  4 bit utbyggbar
  ALU
  30 forskjellige operasjoner



- ALU – SN74LS181

SELECTION				ACTIVE-HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	$\overline{C_n}$ = H (no carry)	$\overline{C_n}$ = L (with carry)
L	L	L	L	F = $\overline{A}$	F = A	F = A PLUS 1
L	L	L	H	F = $\overline{A+B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	F = $\overline{A}B$	F = A + $\overline{B}$	F = (A + $\overline{B}$) PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = ZERO
L	H	L	L	F = $\overline{AB}$	F = A PLUS A$\overline{B}$	F = A PLUS A$\overline{B}$ PLUS 1
L	H	L	H	F = $\overline{B}$	F = (A + B) PLUS A$\overline{B}$	F = (A + B) PLUS A$\overline{B}$ PLUS 1
L	H	H	L	F = A ⊕ B	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
H	L	L	L	F = $\overline{A}$ + B	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	F = $\overline{A \oplus B}$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = (A + $\overline{B}$) PLUS AB	F = (A + $\overline{B}$) PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A†	F = A PLUS A PLUS 1
H	H	L	H	F = A + $\overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	F = (A + $\overline{B}$) PLUS A	F = (A + $\overline{B}$) PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

✗ FIFO (First in first out)

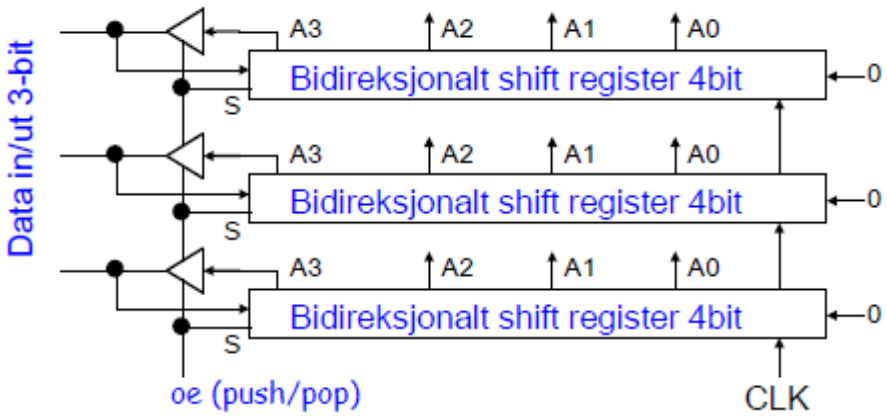
- mellomlagringsenhet, buffer

Eksempel: 4bit bred / 24bit dyp FIFO

✗ Stack

- mellomlagringsenhet for data, data kommer inn (push) og ut (pop) i samme ende

Eksempel: 3bit bred / 4bit dyp stack



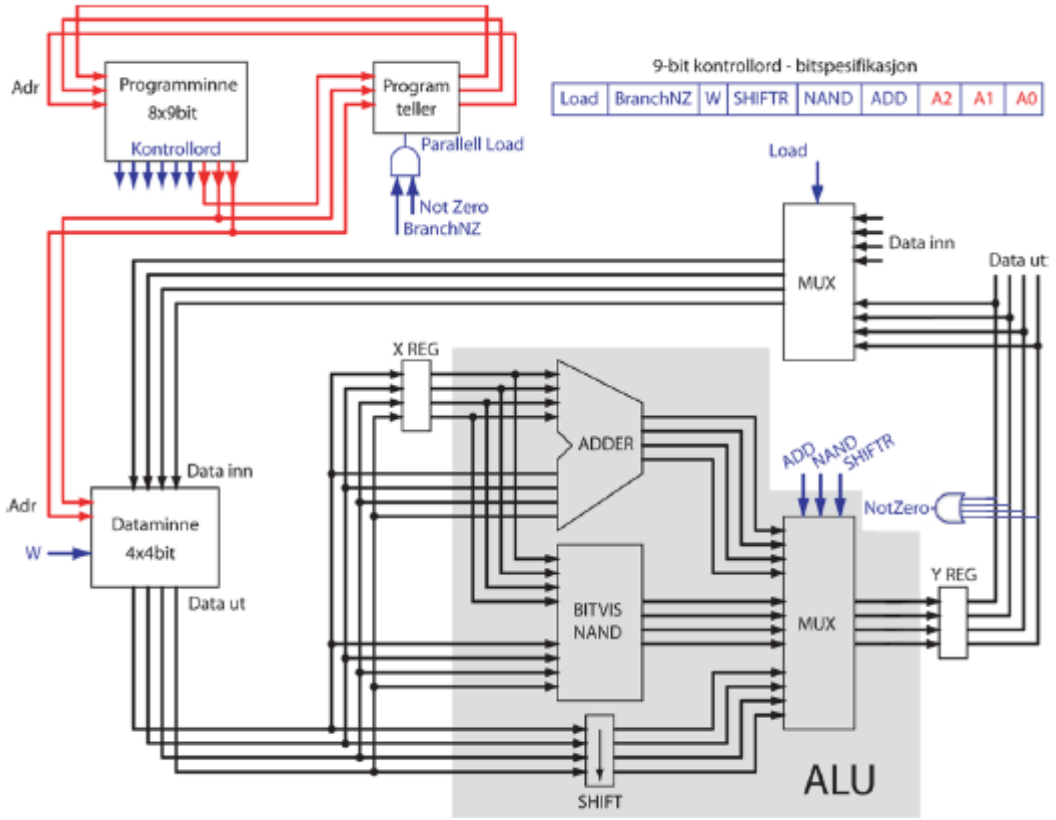
✗ CPU

- CPU (Central Processing Unit)
  - "Hjernen" i en vanlig seriell datamaskin
  - En CPU styres av programkode. Denne koden består av et sett med lavnivå (maskinkode) instruksjoner
  - Maskinkode-instruksjoner er det laveste nivået man kan programmere på
  - Programmering direkte i maskinkode kan i teorien gi optimale programmer (hastighet/plass), men blir fort ekstremt tungvint og uoversiktlig for større program
- RICC-CPU

  RISC – Reduced Instruction Set Computer (få maskinkode-instruksjoner)
  - RISC-prinsippet regnes i dag (av mange) som meget effektivt
  - Dess færre maskinkodeinstruksjoner man har til rådighet dess flere mellomoperasjoner må man utføre
  - Dess færre maskinkodeinstruksjoner man har dess raskere kan de utføres (til en viss grad)
  - RISC prosessorer tar i utgangspunktet mindre plass enn CISC (Compleks Instruction Set Computer) prosessorer

- En minimal RISC-CPU
  - Vi vil nå presentere en ekstrem RISC prosessor med kun 6 instruksjoner. (Pentium4 CISC har >200)
  - Den følgende RISC prosessoren er kraftig forenklet, men er fullt funksjonell, og med mer minne kan den utføre alle oppgaver man kan forvente av en vanlig CPU
  - Den følgende RISC prosessoren kunne ha vært forenklet ytterligere men dette er ikke gjort av pedagogiske grunner

Komplett CPU: 4-bit databuss / 3 bit adressebuss



  - Virkemåte
    - Alle flip-flopper, registre og programteller klokkes av ett felles klokkesignal
    - Innholdet i programtelleren angir adressen til neste instruksjon i programminnet.
    - Programtelleren øker instruksjonsadressen med 1 for hver klokkeperiode (hvis den ikke skal gjøre et hopp)
    - Når en ny instruksjon (kontrollord) lastes ut av programminnet vil de 6 første bittene i kontrollordet (OP-koden) direkte styre hva som skjer i systemet:

- Siste del av kontrollordet (3bit) brukes til å angi adresse i dataminnet for hvor data skal inn/ut evt. adresse i programminnet for hopp

- Kontrollord (OP kode)
  - Bit nr. 8 (load) styrer en MUX som velger om data inn til dataminne skal komme utenifra (IO) eller fra Y register
  - Bit nr. 7 (branchNZ) (hopp hvis resultat ikke er 0) avgjør om programteller skal telle opp eller laste inn ny adresse (parallell load) hvis siste beregning i ALU ikke gir 0
  - Bit nr. 6 (write) styrer om dataminnet skal lese ut data eller skrive inn ny data. (data som blir lest inn vil samtidig være tilgjengelig ut i dette systemet)
- Kontrollord (ALU)

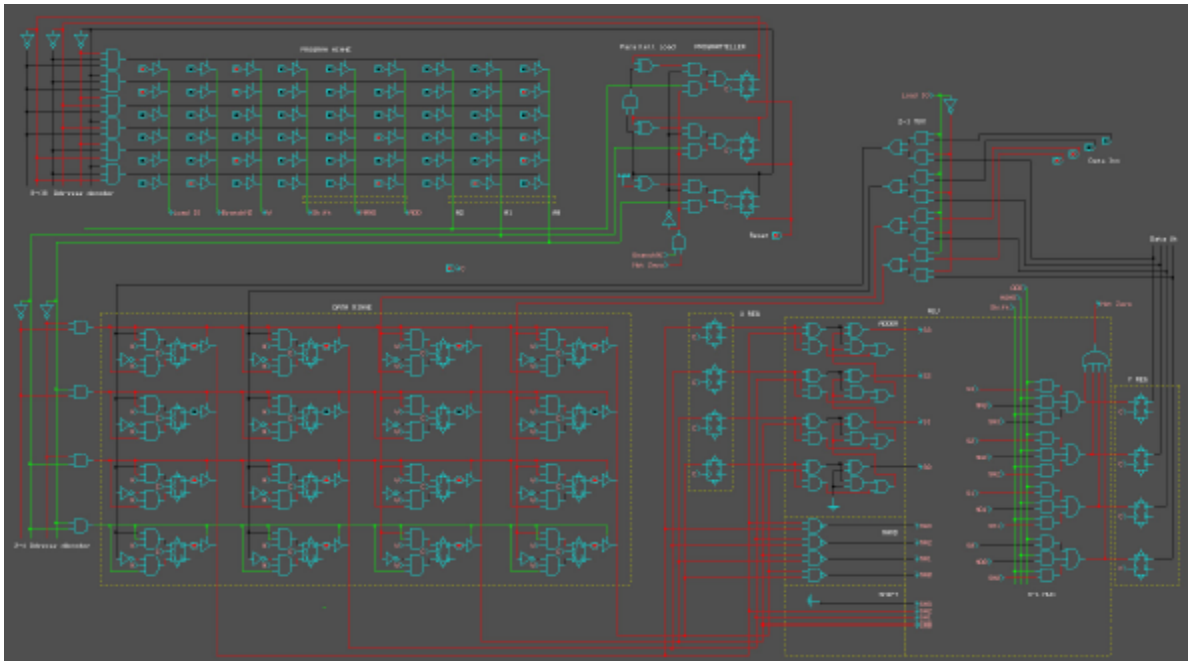
  ALU-en utfører 3 operasjoner samtidig

  1) Addisjon av nåværende data med forrige data (fra X registret)

  2) Bitvis NAND av nåværende data med forrige data

  3) Høyre shift av nåværende data
  - Bit nr. 5 (shift) velger det høyre-shiftede resultatet ut fra ALU-en
  - Bit nr. 4 (NAND) velger det bitvis NANDede resultatet ut fra ALU-en
  - Bit nr. 3 (ADD) velger addisjons resultatet ut fra ALU-en
- Kontrollord (adressedel)
  - Bit nr. 2 (A2) bit nr.2 i adresse for hopp
  - Bit nr. 1 (A1) bit nr.1 i adresse for hopp / data
  - Bit nr. 0 (A0) bit nr.0 i adresse for hopp / data
- Kommentarer:
  - Ordbredden på databussen er her satt til 4bit. Man kan uten videre utvide denne til 32-64bit hvis behov
  - Antall ord i data/programminnet (her valgt til 4/6) kan utvides til hva man måtte ønske
  - Lengden på OP-koden kan reduseres hvis man bruker binær koding
  - ALU-en kan utvides med flere direkte regneoperasjoner for å spare mellomregninger
  - Man kan legge inn muligheten av å direkte legge inn faste verdier (immediate) i dataminnet for å spare mellomregninger

- Komplett CPU: Funksjonell Doglog-implementasjon



- Maskinkode-instruksjoner (kontrollordet i programminnet)

Bit nr.	8	7	6	5	4	3	2	1	0
	Load	BranchNZ	W	ShiftR	NAND	ADD	A2	A1	A1
Hent data fra angitt minneadresse og utfor ADD med forrige data	0	0	0	0	0	1	X	A1	A0
Hent data fra angitt minneadresse og utfor BIT-NAND med forrige data	0	0	0	0	1	0	X	A1	A0
Hent data fra angitt minneadresse og shift data ett bit til høyre	0	0	0	1	0	0	X	A1	A0
Skriv data til angitt minneadresse	0	0	1	X	X	X	X	A1	A0
Load data fra IO til angitt minneadresse	1	0	1	X	X	X	X	A1	A0
Hopp til angitt programadresse hvis forrige data ikke ble 0	0	1	0	X	X	X	A2	A1	A0

- RISC program-eksempler
  - Eksempel 1: Ta inn to tall fra IO, adder tallene og gi svaret til IO.
    Løsning:

    0. Les inn første tall fra IO, og legg det i dataminne nr. 0
       Maskinkode: 101 000 000

    1. Les inn neste tall fra IO, og legg det i dataminne nr. 01

       Maskinkode: 101 000 001

    2. Hent ut siste tall fra dataminne 01 (legges i X reg)

       Maskinkode: 000 000 001

    3. Les ut første tall fra dataminne 00, og utfør ADD med forrige tall (som er innholdet i dataminne 01)

       Maskinkode: 000 001 000

    4. Skriv resultat til (for eksempel) dataminne nr. 10

       Maskinkode: 001 000 010

    (Databussen er direkte synlig for IO)

  - Eksempel 2: Implementasjon av for-løkke

Java kode	*Hva vi setter CPU til å gjøre*
uint k;	// velger dataadresse 10
k = IO.inInt();	// Henter k fra IO
for(;k > 0; k~)	
{	//
…	// Her kan man legge hva som
{	helst
	// Neste instruksjon

    *Vår RISC kan ikke subtrahere direkte. Legger derfor først inn 1111 (2er komplement for -1) i dataminne 00. Kan så addere -1 til k underveis i løkken

    Eksempel 2: Implementasjon av for-løkke (initialisering)

  - 000: Les inn ”-1” (1111) fra IO og legge dette på minne 00 101 000 000
  - 001: Les inn ”0” (0000) fra IO og legge dette på minn 01 101 000 001
  - 010: Les inn k fra IO og legge det til dataminne 10 101 000 010

Eksempel 2: Implementasjon av for-løkke (selve løkken)

- 011: Hente fram ”-1” fra dataminne 000 000 000
- 100: Hente fram ”k” fra dataminne og ADD med forrige data 000 001 001
- 101: Lagre resultatet til ”k” 001 000 010
- 110: Hente inn ”0” og ADD med ”k” 000 001 001
- 111: Hoppe hvis resultatet ikke ble 0 (dvs. K != 0) 010 000 011

Når man lagrer ”k” bli denne verdien også satt i X reg (linje 101). Når man da utfører ADD i linje 110 blir Y Reg satt til ”k”. I linje 111 sjekker man Y reg og hopper hvis resultatet ikke er 0.

KAPITTEL 5

Sekvensiell logikk

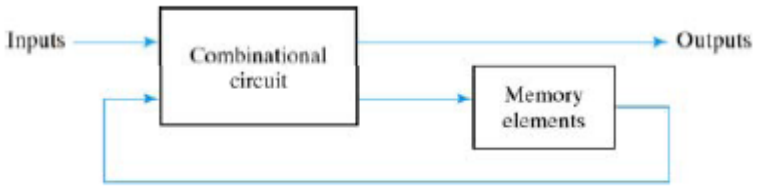
- ✗ Definisjoner
  - Kombinatorisk logikk

    Utgangsverdiene er entydig gitt av nåværende kombinasjon av inngangsverdier
  - Sekvensiell logikk

    Inneholder hukommelse (låsekretser)

    Utgangsverdiene er gitt av nåværende kombinasjon av inngangsverdier, samt sekvensen (tidligere inngangs-/utgangsverdier)
- ✗ Sekvensiell logikk

  Sekvensiell krets – generell oversikt



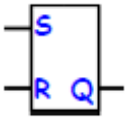
- ✗ Praktiske eksempler
  - Låsekrets

    Et eksempel basert på et vanlig behov innen digital design:

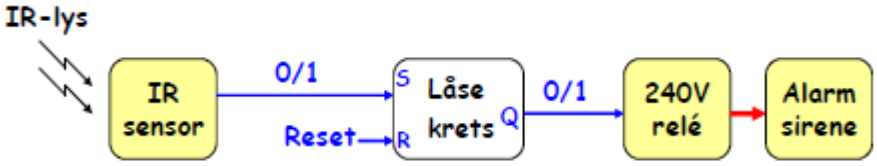
    • Ønsker å ha en to-inputs krets med følgende egenskaper:

    1) Kretsen skal sette utgangen Q til ”1” hvis den får ”1” på inngang S. Når inngang S går tilbake til ”0” skal utgangen forbli på ”1”

    2) Kretsen skal resette utgangen til ”0” hvis den får ”1” på inngang R. Når inngang R går tilbake til ”0” skal utgangen forbli på ”0”

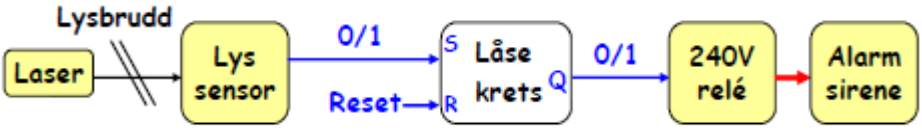


- Logikk som behandler signaler fra fysiske sensorer:
  - Varmeførende persondetektor



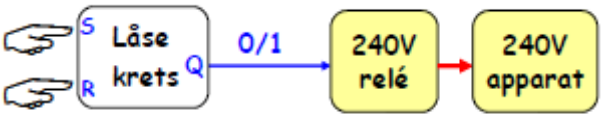
Når IR-lyset varierer mottar logikken et ”ras” av kortvarige 1'er pulser (µsek). Logikken skal sette sirenen permanent på på første mottatte puls

- Laserbasert tyveridetektor:



Når laserlyset blir brutt mottar logikken en eller flere 1'er pulser. Logikken skal sette sirenen permanent på første mottatte puls.

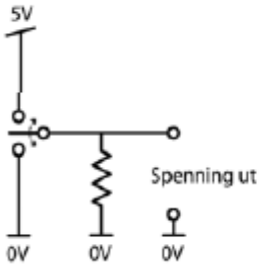
- Touch-bryter:

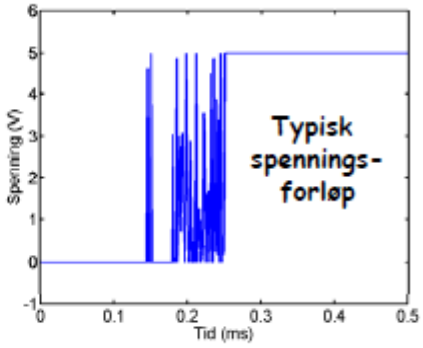


CMOS logikk har meget høy inngangsmotstand. Løse innganger reagerer derfor på statiske spenninger. Når man berører en løs inngang mottar inngangen et "ras" av spenningspulser ca. 0.6 til +5.5V. Dette kan brukes til å lage en touch-bryter.

- Kontaktprelling fra mekanisk bryter:

Mekaniske brytere gir ikke ”rene” logiske nivå ut i overgangsfasen. Slike signal må ofte ”renses” ved bruk av låsekretser.





✗ SR Latch

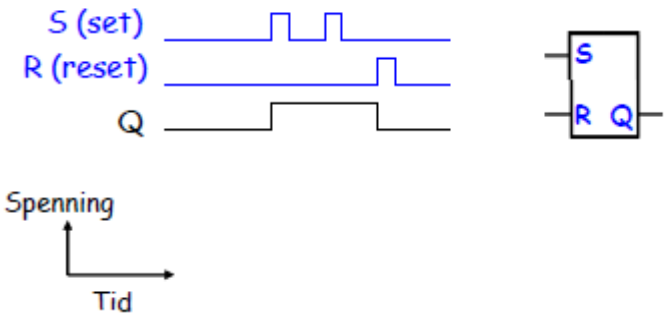
- Funksjonell beskrivelse:

  1) Kretsen skal sette Q til ”1” hvis den får ”1” på inngang S. Når inngang S går tilbake til ”0” skal Q forbli på ”1”

  2) Kretsen skal resette Q til ”0” når den får ”1” på inngang R Når inngang R går tilbake til ”0” skal Q forbli på ”0”

  3) Tilstanden ”1” på både S og R brukes normalt ikke

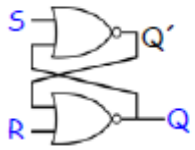
  • SR latch – en sekvensiell krets



SR

S	R	Q
0	0	låst
0	1	0
1	0	1
1	1	0

- SR Latch laget ved NOR



Øvre NOR

S	Q	Q´
0	0	1
0	1	0
1	0	0
1	1	0

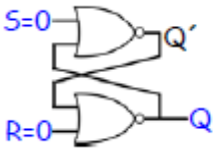
Nedre NOR

Q´	R	Q
0	0	1
0	1	0
1	0	0
1	1	0

*Signalet Q' er ikke invertert av Q for tilstand S = 1, R = 1

- SR – analyse

  • Tilstand S=0, R=0:
  En NOR port med fast ”0” inn på en av inngangene er ekvivalent med NOT



SR

S	R	Q
0	0	
0	1	
1	0	
1	1	

Øvre NOR

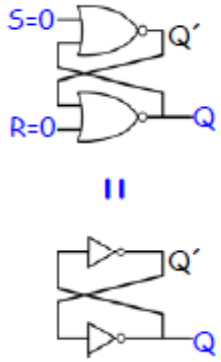
S	Q	Q´
0	0	1
0	1	0
~~1~~	~~0~~	~~0~~
~~1~~	~~1~~	~~0~~

Nedre NOR

Q´	R	Q
0	0	1
~~0~~	~~1~~	~~0~~
1	0	0
~~1~~	~~1~~	~~0~~

Ser bort i fra tilstand S = 1 og R = 1

- Tilstand S=0, R=0:
  En NOR port med fast ”0” inn på en av inngangene er ekvivalent med NOT



SR

S R	Q
0 0	låst
0 1	
1 0	
1 1	

Øvre NOR

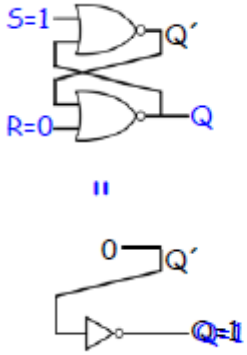
S Q	Q´
0 0	1
0 1	0
~~1 0~~	~~0~~
~~1 1~~	~~0~~

Nedre NOR

Q´R	Q
0 0	1
~~0 1~~	~~0~~
1 0	0
~~1 1~~	~~0~~

To inverter i ring: stabil låsekrets. Utgangsverdien holdes låst.

- Tilstand S=1, R=0:
  En NOR port med fast ”1” inn på en av inngangene gir alltid ut ”0”



SR

S R	Q
0 0	låst
0 1	
1 0	1
1 1	

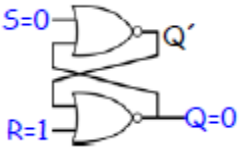
Øvre NOR

~~S Q~~	Q´
~~0 0~~	~~1~~
~~0 1~~	~~0~~
1 0	0
1 1	0

Nedre NOR

Q´R	Q
0 0	1
~~0 1~~	~~0~~
1 0	0
~~1 1~~	~~0~~

- Tilstand S=0, R=1:
  En NOR port med fast ”1” inn på en av inngangene gir alltid ut ”0”



SR

S R	Q
0 0	låst
0 1	0
1 0	1
1 1	

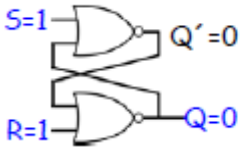
Øvre NOR

S Q	Q´
0 0	1
0 1	0
~~1 0~~	~~0~~
~~1 1~~	~~0~~

Nedre NOR

~~Q´R~~	Q
~~0 0~~	~~1~~
0 1	0
~~1 0~~	~~0~~
1 1	0

- Tilstand S=1, R=1:
  En NOR port med fast ”1” inn på en av inngangene gir alltid ut ”0”



SR

S R	Q
0 0	låst
0 1	0
1 0	1
1 1	0

Øvre NOR

SQ	Q´
~~0 0~~	~~1~~
~~0 1~~	~~0~~
1 0	0
1 1	0

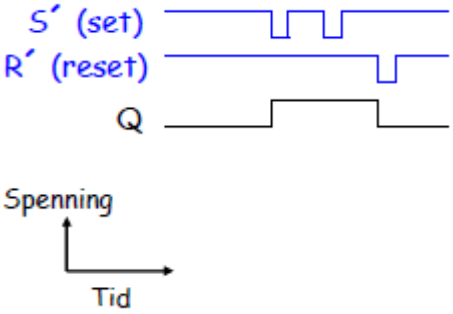
Nedre NOR

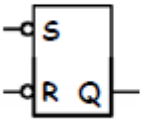
Q´R	Q
~~0 0~~	~~1~~
0 1	0
~~1 0~~	~~0~~
1 1	0

- I denne tilstanden er utgang Q' ikke den inverterte av Q.
  <u>Denne tilstanden brukes normalt ikke</u>

✗ S'R' Latch

Lik funksjon som SR latch, men reagerer på ”0” inn (negativ logikk).



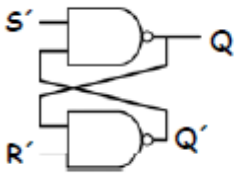


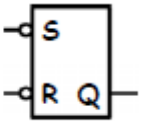
S´R´	Q
1 1	låst
1 0	0
0 1	1
0 0	1

Tilstanden S'=0, R'=0 brukes normalt ikke

✗ S'R' Latch laget ved NAND

Kretsen kan analyseres på samme måte som for SR latch





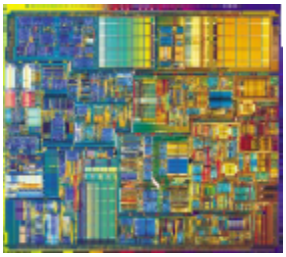
S´R´	Q
1 1	uforandret
1 0	0
0 1	1
0 0	1

✗ Synkron logikk

I større digitale system har man behov for å synkronisere dataflyten.
Til dette bruker vi et globalt klokkesignal.

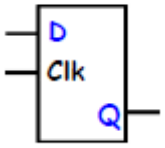
Alle store digitale systemer i dag er synkrone.
Eksempel: Pentium4

Uten global synkronisering ville vi hatt totalt kaos



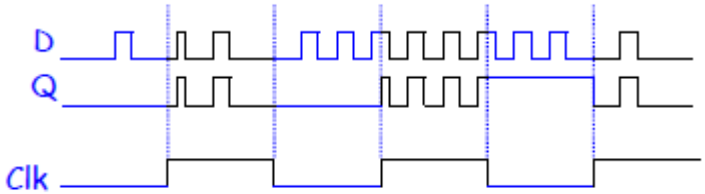
✗ D Latch

Dataflyten gjennom en D latch kontrolleres av et klokkesignal



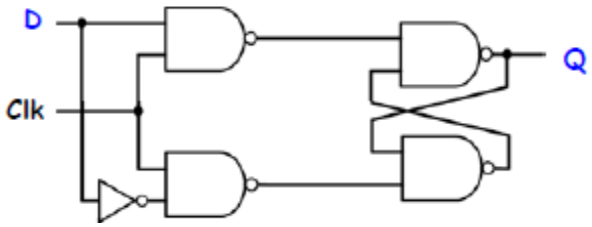
1) Slipper gjennom et digitalt signal så lenge klokkeinngangen er ”1” (transparent)
2) I det øyeblikk klokkeinngangen går fra ”1” til ”0” låser utgangen seg på sin nåværende verdi. Forandringer på inngangen vil ikke påvirke utgangsverdien så lenge klokkesignalet er ”0”.

- Signaler:
  Clk = 1 : kretsen slipper gjennom signalet
  Clk = 0 : kretsen holder (låser) utgangssignalet

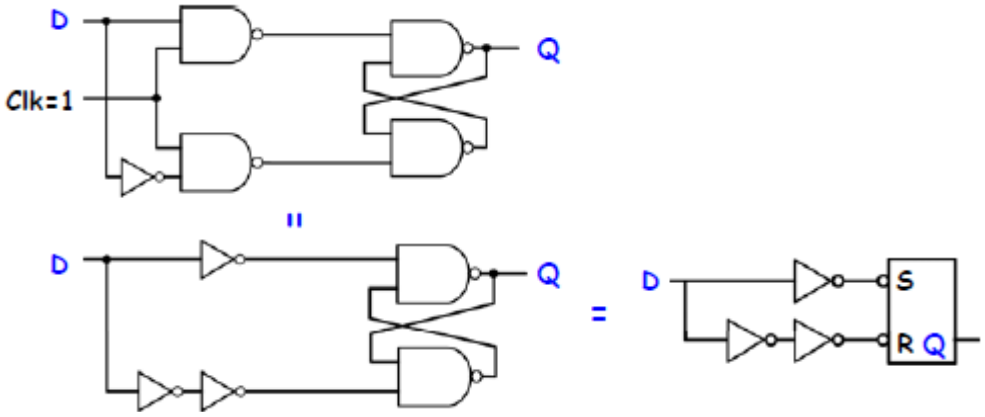


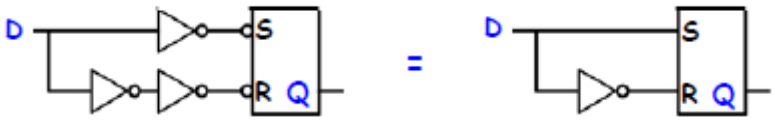
Logisk verdi på D i det øyeblikk Clk går i fra ”1” til ”0” bestemmer verdien som holdes på Q

- Implementasjon ved NAND porter

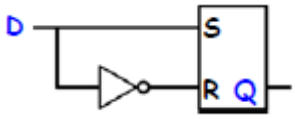


- Analyse:
  Analyserer virkemåten for 2 tilstander:
  - Tilstand 1: Clk=1
    NAND med fast "1" på en inngang er ekvivalent med NOT





Dette er en S'R' latch med inverterte innganger
Tilstand S' = R' oppstår aldri, og kretsen er ekvilalent med en S"R" eller SR latch

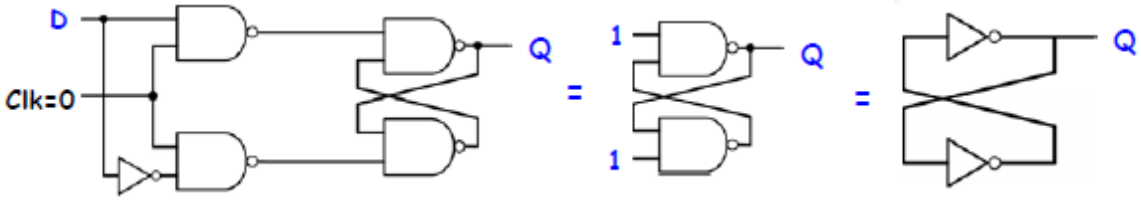


SR

S R	Q
~~0 0~~	~~uforandret~~
0 1	0
1 0	1
~~1 1~~	~~Q~~

S og R vil alltid være forskjellige.
Fra sannhetstabellen ser vi at Q=S=D, kretsen slipper signal D rett gjennom (transparent)

- Tilstand 2: Clk=0
  NAND med "0" på en inngang gir alltid ut "1"



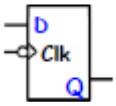
NAND med ”1” på en inngang er ekvivalent med NOT.
Får stabil låsekrets. Utgang holdes

✗ Flip-Flop'er

Flip-Flop’er kommer i to varianter:

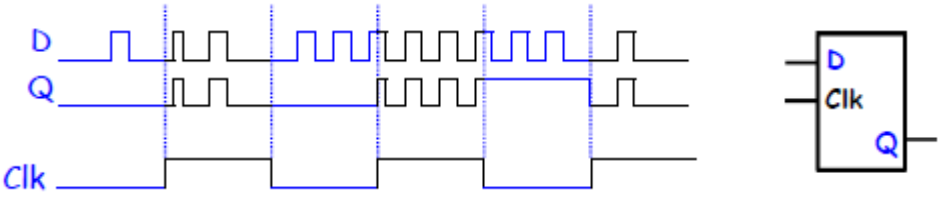
- Positiv flanketrigget
- Negativ flanketrigget

På en positiv flanketrigget Flip-Flop kan utgangen kun skifte verdi i det øyeblikket klokkesignalet går fra "0" til "1"

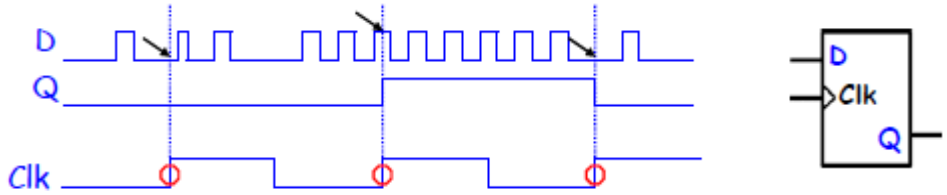


På en negativ flanketrigget Flip-Flop kan utgangen kun skifte verdi i det øyeblikket klokkesignalet går fra "1" til "0"

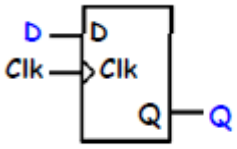
En D latch er transparent for Clk=1



En positiv flanketrigget D flip-flop sampler verdien på D i det øyeblikk Clk går fra ”0” til ”1” (positiv flanke). Denne verdien holdes fast på utgangen helt til neste positive flanke



- Karektaristisk tabell/ligning
  For flip-flop’er kan man generelt beskrive neste utgangsverdi Q(t+1) som funksjon av nåværende inngangsverdi(er), og nåværende utgangsverdi Q(t)



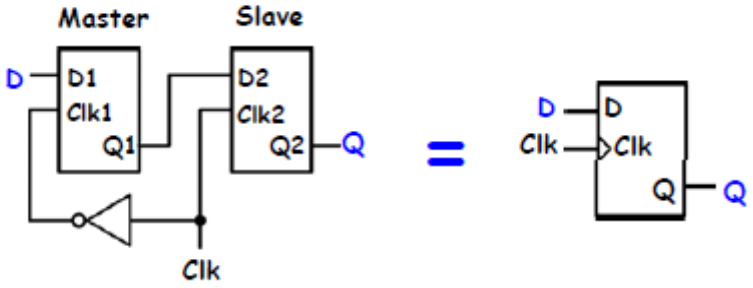
Karakteristisk tabell for D flip-flop

D	Q(t+1)
0	0
1	1

Karakteristisk ligning for D flip-flop

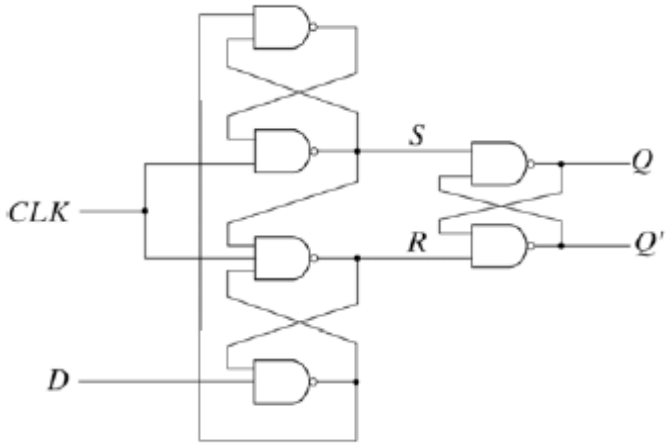
$Q(t+1) = D$

- En positiv flanketrigget D flip-flop kan lages av to D latcher (Master-Slave)

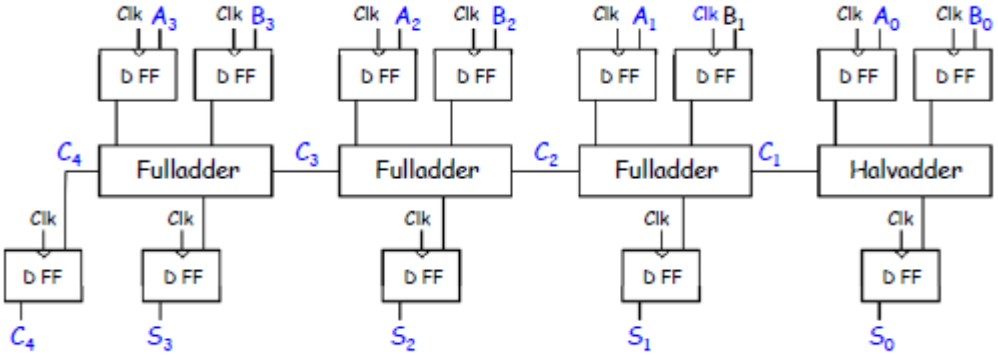


Under Clk=0 er første D latch (master) transparent
Under Clk=1 er siste D latch (slave) transparent

✗ D Flip-Flop
Kompakt versjon

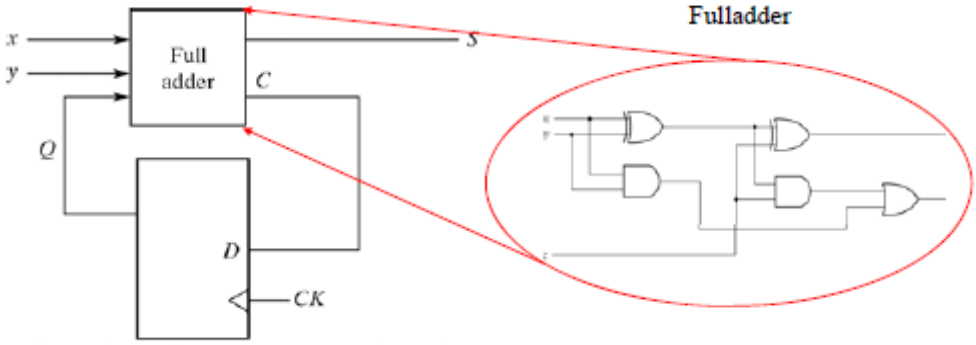


- Praktiske eksempler:
  - En rippeladder vil i et kort tidsrom gi gal sum ut.
    Styring av signalflyt med D flip-flops kamuflerer dette

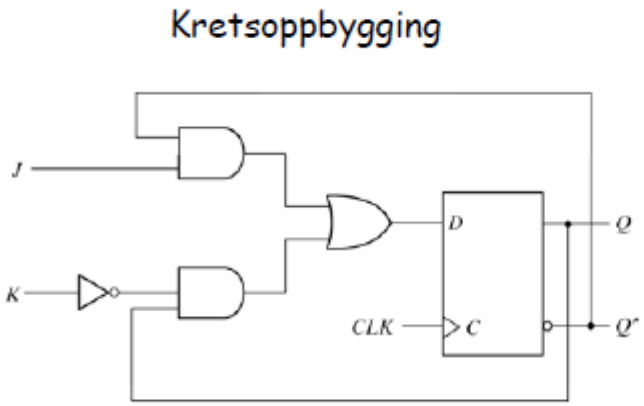


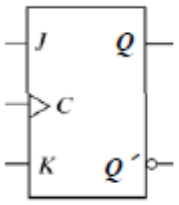
    På positiv Clk flanke kommer nye data inn til adderen. I samme øyeblikk leses forrige (stabiliserte) sum ut.

  - Seriell adder



✗ JK Flip-Flop





J K	Q(t+1)
0 0	Q(t)
0 1	0
1 0	1
1 1	Q´(t)

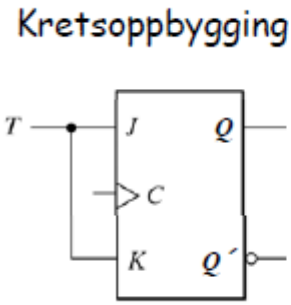
$Q(t+1) = JQ'(t) + K'Q(t)$

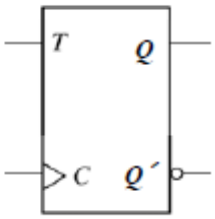
– En JK flip-flop har følgende egenskaper:
  - J=0, K=0: Utgang låst
  - J=0, K=1: Resetter utgang til ”0”
  - J=1, K=0: Setter utgang til ”1”
  - J=1, K=1: Inverterer utgang Q --> Q'

  Utgangen kan kun forandre verdi på stigende klokkeflanke

  En JK flip-flop er den mest generelle flip-floppen vi har.

✗ T Flip-Flop





En T flip-flop har følgende egenskaper
- T=0: Utgang låst
- T=1: Inverterer utgang Q --> Q'

Utgangen kan kun forandre verdi på stigende klokkeflanke
Det er lett å lage tellere av T flip-flop'er

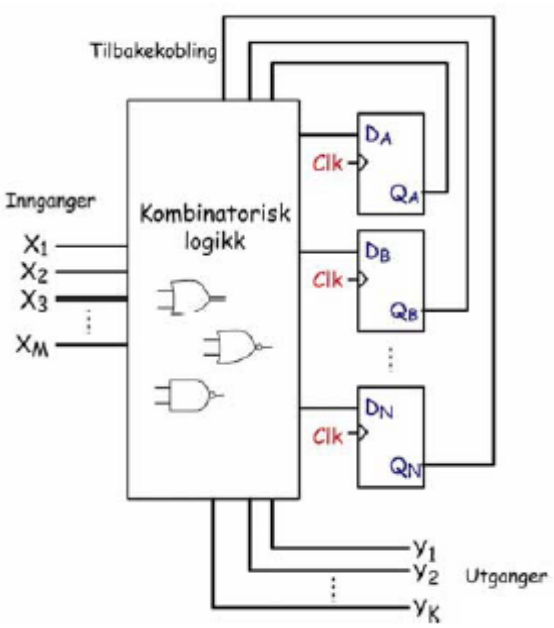
T	Q(t+1)
0	Q(t)
1	Q´(t)

$Q(t+1) = T \oplus Q(t)$

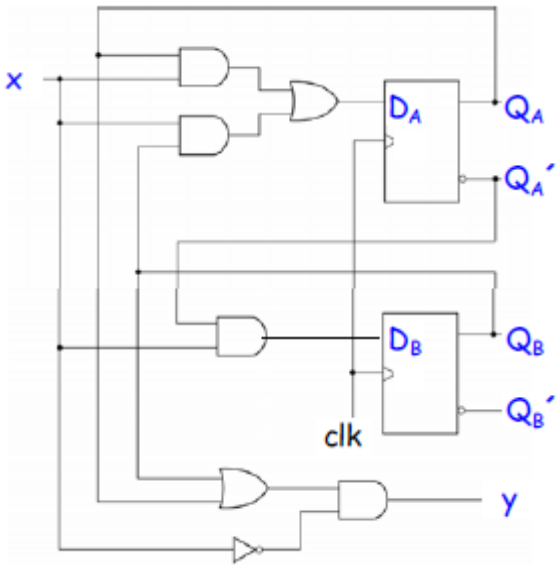
Tilstandsmaskin

- En tilstandsmaskin er et sekvensielt system som gjennomløper et sett med tilstander styrt av verdiene på inngangssignalene
  Tilstanden systemet befinner seg i, pluss evt. inngangsverdier bestemmer utgangsverdiene
  Tilstandsmaskins-konseptet gir en enkel og oversiktlig måte å designe avanserte system på

  Generell tilstandsmaskin basert på D flip-flops --> N-stk flip-flops gir 2^n forskjellige tilstander
  Utgangssignalene er en funksjon av nåværende tilstand pluss evt. inngangsverdier



- Eksempel nr. 1:
  Tilstandsmaskin der utgang y er en funksjon av tilstanden gitt av verdiene til Q_A og Q_B, samt inngangen x



✗ Tilstandstabell

Tilstandstabell = sannhetstabell for tilstandsmaskin

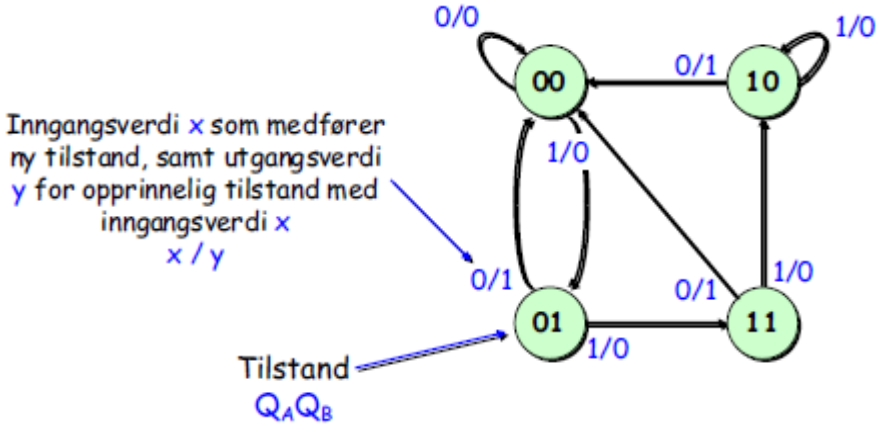
Eksempel nr.1:
En inngang, en utgang og 2 stk. D flipflops

Nåværende tilstand Q_A Q_B	Inngang x	Neste tilstand Q_A Q_B	Utgang for nåværende tilstand y
0 0	0	0 0	0
0 0	1	0 1	0
0 1	0	0 0	1
0 1	1	1 1	0
1 0	0	0 0	1
1 0	1	1 0	0
1 1	0	0 0	1
1 1	1	1 0	0

✗ Tilstandsdiagram

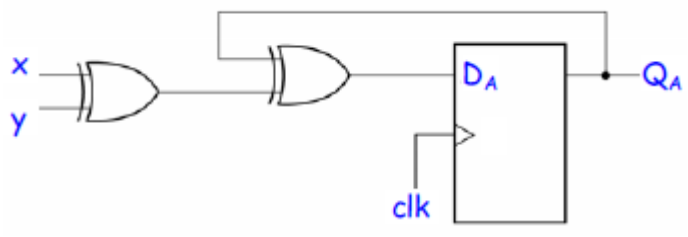
Tilstandsdiagram = grafisk illustrasjon av egenskapene til en tilstandsmaskin

Eksempel nr. 1:



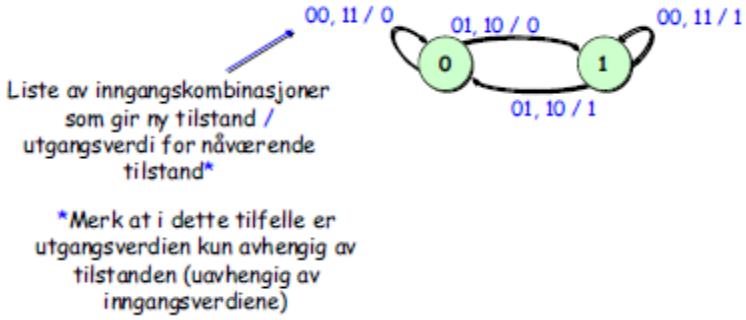
✗ Eksempel nr. 2:

To innganger x og y, en utgang som bare er gitt av tilstanden Q_A



Nåværende tilstand Q_A	Innganger x	y	Neste tilstand Q_A	Utgang for nåværende tilstand Q_A
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tilstandsdiagram:

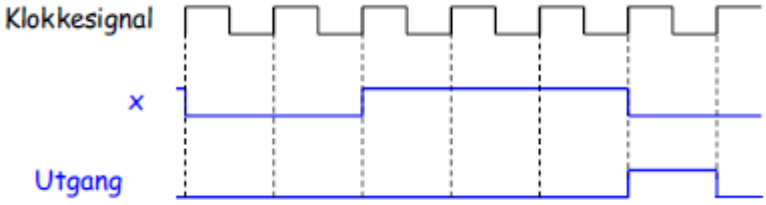


✗ Eksempel nr. 3:

Design av sekvensdetektor:

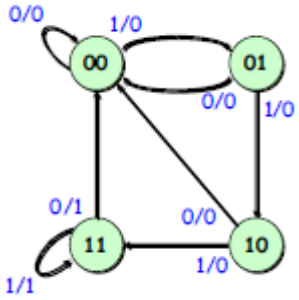
Ønsker å lage en krets som finner ut om det har forekommet tre eller flere ”1”ere etter hverandre i en klokket bit-sekvens x

Klokket bit-sekvens: Binært signal som kun kan skifte verdi synkront med et klokkesignal



Tilstandsdiagram:

Velger å ha 4 tilstander. Lar hver tilstand symbolisere antall ”1”ere som ligger etter hverandre i bit-sekvensen.



Inngang: bit-sekvens x

Utgang: gitt av tilstanden, ”0” for tilstand 0-2, ”1” for tilstand 3

Tilstandstabell:

Bruker D flip-flops

D_A og D_B settes til de verdiene man ønsker at Q_A og Q_B skal ha i neste tilstand

Nåværende tilstand Q_A	Q_B	Inngang x	Neste tilstand Q_A	Q_B	Utgang for nåværende tilstand y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

$$D_A = Q_A'Q_Bx + Q_AQ_B'x + Q_AQ_Bx$$

$$D_B = Q_A'Q_B'x + Q_AQ_B'x + Q_AQ_Bx$$

$$y = Q_AQ_B$$

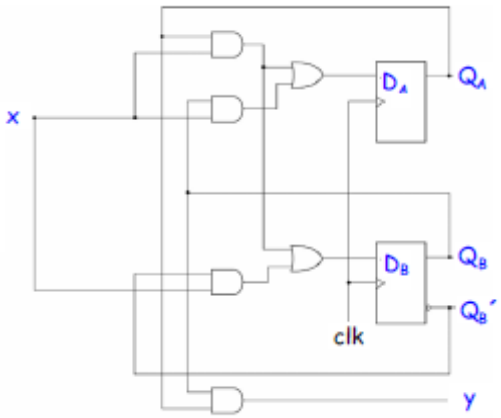
Forenkling:

Forenkler uttrykkene med Karnaugh-diagram

$$D_A = Q_Ax + Q_Bx$$

$$D_B = Q_Ax + Q_B'x$$

$$y = Q_AQ_B$$



✗ Reduksjon av tilstander

En tilstandsmaskin gir oss en eller flere utgangssignal som funksjon av en eller flere inngangssignal
Hvordan dette implementeres internt i maskinen er uinteressant sett utenifra
I noen tilfeller kan man fjerne tilstander (forenkle designet) uten å påvirke inngangs/utgangs-funksjonene

Hvis to tilstander har samme utgangssignal, samt leder til de samme nye tilstandene gitt like inngangsverdier, er de to opprinnelige tilstandene like. En tilstand som er lik en annen tilstand kan fjernes.

Eksempel:

Tilstand G er lik tilstand E

Nåværende tilstand	Inngang	Neste tilstand	Utgang
A	0	B	0
A	1	B	0
B	0	C	0
B	1	D	0
C	0	A	0
C	1	D	0
D	0	E	0
D	1	F	1
E	0	A	0
E	1	F	1
F	0	G	0
F	1	F	1
G	0	A	0
G	1	F	1

Fjerner tilstand G. Erstatter hopp til G med hopp til E

Nåværende tilstand	Inngang	Neste tilstand	Utgang
A	0	B	0
A	1	B	0
B	0	C	0
B	1	D	0
C	0	A	0
C	1	D	0
D	0	E	0
D	1	F	1
E	0	A	0
E	1	F	1
F	0	E	0
F	1	F	1

Nå er tilstand F lik tilstand D

Fjerner tilstand F

Nåværende tilstand	Inngang	Neste tilstand	Utgang
A	0	B	0
A	1	B	0
B	0	C	0
B	1	D	0
C	0	A	0
C	1	D	0
D	0	E	0
D	1	F	1
E	0	A	0
E	1	F	1
F	0	E	0
F	1	F	1

Har fjernet tilstand F

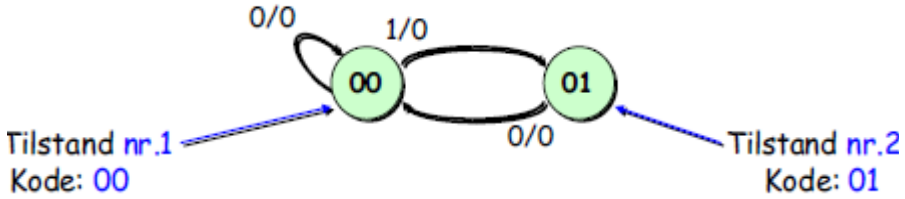
Nåværende tilstand	Inngang	Neste tilstand	Utgang
A	0	B	0
A	1	B	0
B	0	C	0
B	1	D	0
C	0	A	0
C	1	D	0
D	0	E	0
D	1	D	1
E	0	A	0
E	1	D	1

- Tilordning av tilstandskoder

  I en tilstandsmaskin med M tilstander må hver tilstand tilordnes en kode basert på minimum N bit der $2^N \geq M$

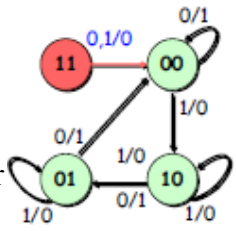
  Kompleksiteten til den kombinatoriske delen avhenger av valg av tilstandskode

  Anbefalt strategi for valg av kode: prøv-og-feil i tilstandsdiagrammet



- Ubrukte tilstander

  I en tilstandsmaskin med N flip-flopper vil det alltid finnes 2^N tilstander.
  Designer man for M tilstander der $M < 2^N$ vil det finnes ubrukte tilstander.



  Problem: Under oppstart (power up) har man ikke full kontroll på hvilken tilstand man havner i først. Havner man i en ubrukt tilstand som ikke leder videre til de ønskede tilstandene vil systemet bli låst.

Løsning: Design systemet slik at alle ubrukte tilstander leder videre til en ønsket tilstand.

✗ Generell designprosedyre basert på D flip-flops

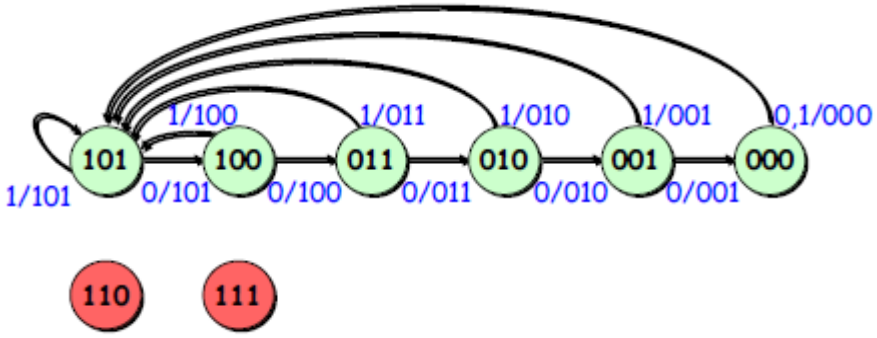
1) Definer tilstandene, inngangene og utgangene
2) Velg tilstandskoder, og tegn tilstandsdiagram
3) Tegn tilstandstabell
4) Reduser antall tilstander hvis nødvendig
5) Bytt tilstandskoder hvis nødvendig for å forenkle
6) Finn de kombinatoriske funksjonene
7) Skjekk at ubrukte tilstander leder til ønskede tilstander
8) Tegn opp kretsen

✗ Eksempel nr. 4

Design en teller som teller sekvensen 5,4,3,2,1,0. Etter 0 skal telleren gjenta sekvensen (telle rundt). Telleren skal kunne resettes til 5 med ett reset signal.

1) Velger en tilstand for hvert tall ut. Systemet har 1 reset inngang, og trenger 3 utganger for å representere tallene 5 til 0.
2) Velger tilstandskoder som direkte representerer tallene ut. Tallene ut blir gitt av tilstandene

Tilstandsdiagram:



Registrer at vi har to ubrukte tilstander

– Tilstandstabell:
  - Tegner tilstandstabell
  - Ingen reduksjonsmuligheter
  - Velger å ikke bytte tilstandskoder da utgangene i såfall må omformes

	Nåværende tilstand / utgang $Q_A\ Q_B\ Q_C$	Inngang R	Neste tilstand $Q_A\ Q_B\ Q_C$
	0 0 0	0	1 0 1
	0 0 0	1	1 0 1
	0 0 1	0	0 0 0
	0 0 1	1	1 0 1
	0 1 0	0	0 0 1
	0 1 0	1	1 0 1
	0 1 1	0	0 1 0
	0 1 1	1	1 0 1
	1 0 0	0	0 1 1
	1 0 0	1	1 0 1
	1 0 1	0	1 0 0
	1 0 1	1	1 0 1
Ubrukte tilstander	1 1 0	0	X X X
	1 1 0	1	X X X
	1 1 1	0	X X X
	1 1 1	1	X X X

- Karnaugh-diagram:

D_A

$Q_A Q_B$ \ $Q_C R$	00	01	11	10
00	1	1	1	0
01	0	1	1	0
11	x	x	x	x
10	0	1	1	1

D_B

$Q_A Q_B$ \ $Q_C R$	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	x	x	x	x
10	1	0	0	0

D_C

$Q_A Q_B$ \ $Q_C R$	00	01	11	10
00	1	1	1	0
01	1	1	1	0
11	x	x	x	x
10	1	1	1	0

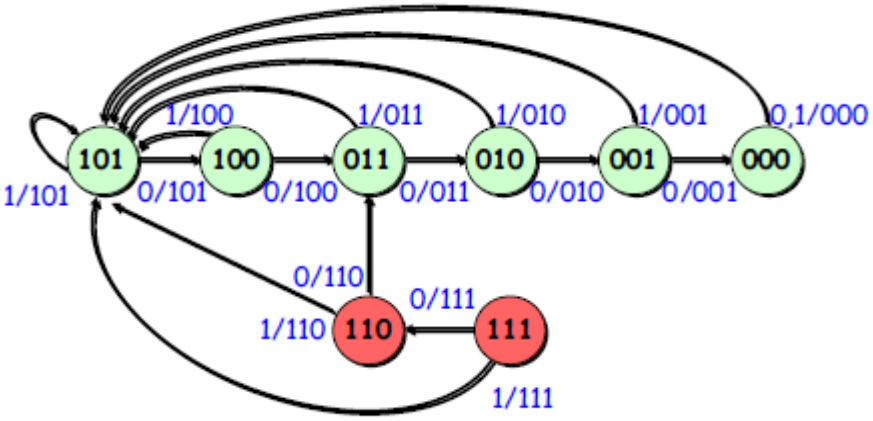
$$D_A = R + Q_A' Q_B' Q_C' + Q_A Q_C$$
$$D_B = Q_B Q_C R' + Q_A Q_C' R'$$
$$D_C = Q_C' + R$$

- Sjekker at ubrukte tilstander leder til ønskede tilstander

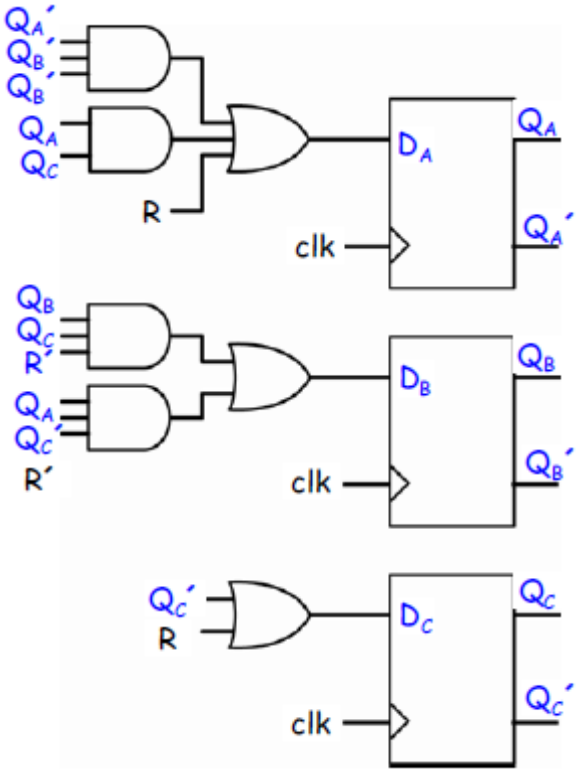
Nåværende tilstand / utgang Q_A	Q_B	Q_C	Inngang R	Neste tilstand Q_A	Q_B	Q_C
1	1	0	0	0	1	1
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	0	1

$$D_A = R + Q_A' Q_B' Q_C' + Q_A Q_C$$
$$D_B = Q_B Q_C R' + Q_A Q_C' R'$$
$$D_C = Q_C' + R$$

- Viser i diagram at ubrukte tilstander leder til ønskede tilstander

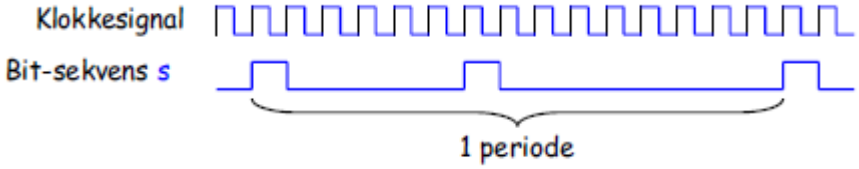


- Tegner krets:
  Q_A, Q_B og Q_C blir tellerens utganger
  Telleren resettes ved å sette R=1

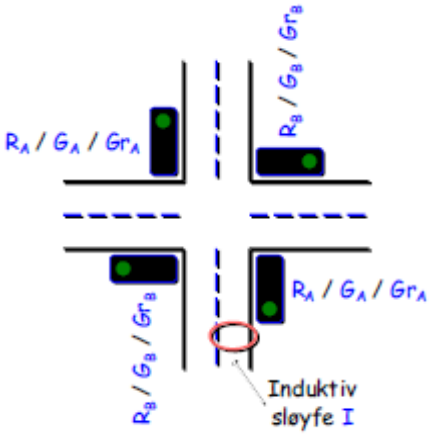


✗ Eksempel nr. 5

Ønsker å bruker tilstandsmaskin for å styre trafikklys
Krysset har to vanlige trafikklys A og B. Disse styres med de binære signalene R_A, G_A, Gr_A samt R_B, G_B, Gr_B.
Setter man G_A til "1" lyser det grønt i lys A osv.
For å generere lyssekvensene bruker vi en repeterende bit-sekvens s som vist under.
Avstanden mellom "1"er pulsene gir intervallene mellom skifte fra grønt i lys A tilgrønt i lys B og motsatt.



Systemet har en induktiv sensor i bakken som registrerer biler den ene veien.
Bil over sensoren gir I=1 ellers har vi I=0
Vi ønsker at bil registrert av sensoren skal gi grønt lys i A så fort som mulig



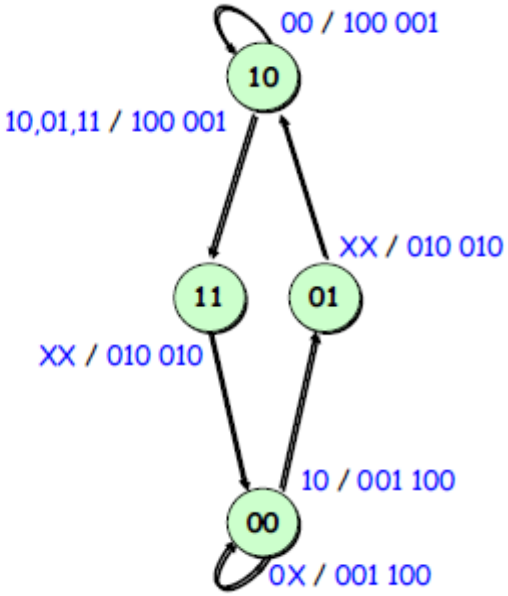
- Forenklede tilstander:
  - 00 - Grønt lys i A, rødt lys i B
  - 01 - Gult lys i A og B. Skifter mot grønt lys i B.
  - 10 - Rødt lys A, grønt lys i B
  - 11 - Gult lys i A og B. Skifter mot grønt lys i A.

  Innganger: s, I
  Utganger: R_A, G_A, Gr_A, R_B, G_B, Gr_B
  Lar utgangene kun være en funksjon av tilstanden

- Tilstandsdiagram

X - don't care

sI / $R_AG_AGr_A$ $R_BG_BGr_B$



- Tilstandstabell:

Finner kombinatoriske funksjoner

$D_A = Q_A \oplus Q_B$

$D_B = Q_AQ_B'I + Q_B'sI'$

$R_A = Q_AQ_B'$

$G_A = Q_B$

$Gr_A = Q_A'Q_B'$

$R_B = Gr_A$

$G_B = G_A$

$Gr_B = R_A$

Nåværende tilstand		Innganger		Neste tilstand		Utganger	
Q_A	Q_B	s	I	Q_A	Q_B	$R_AG_AGr_A$	$R_BG_BGr_B$
0	0	0	0	0	0	0 0 1	1 0 0
0	0	0	1	0	0	0 0 1	1 0 0
0	0	1	0	0	1	0 0 1	1 0 0
0	0	1	1	0	0	0 0 1	1 0 0
0	1	0	0	1	0	0 1 0	0 1 0
0	1	0	1	1	0	0 1 0	0 1 0
0	1	1	0	1	0	0 1 0	0 1 0
0	1	1	1	1	0	0 1 0	0 1 0
1	0	0	0	1	0	1 0 0	0 0 1
1	0	0	1	1	1	1 0 0	0 0 1
1	0	1	0	1	1	1 0 0	0 0 1
1	0	1	1	1	1	1 0 0	0 0 1
1	1	0	0	0	0	0 1 0	0 1 0
1	1	0	1	0	0	0 1 0	0 1 0
1	1	1	0	0	0	0 1 0	0 1 0
1	1	1	1	0	0	0 1 0	0 1 0

- Krets:

$$D_A = Q_A \oplus Q_B$$

$$D_B = Q_A Q_B' I + Q_B' s I'$$

$$R_A = Q_A Q_B'$$

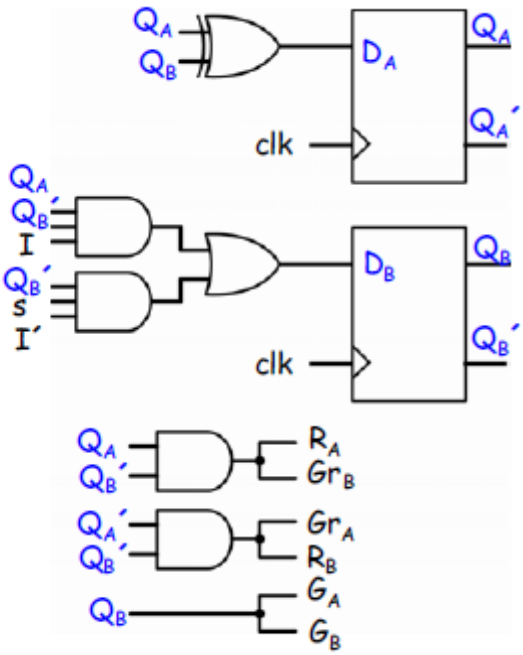
$$G_A = Q_B$$

$$Gr_A = Q_A' Q_B'$$

$$R_B = Gr_A$$

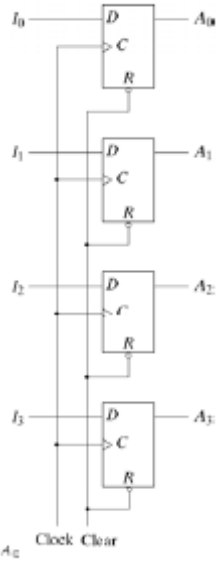
$$G_B = G_A$$

$$Gr_B = R_A$$



KAPITTEL 6

Registre og tellere



- Register:

  N bits register – parallellkobling av N stk. D flip-flopper

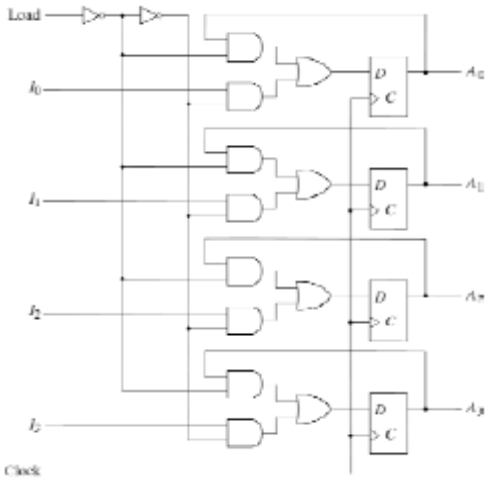
  Data I_0-I_3 slippes gjennom til utgang A_0-A_3 på stigende klokkeflanke

  Anvendelse: Kontroll av dataflyt

- Lagringsregister:

  Data I_0-I_3 lastes inn samtidig på stigende flanke og når Load er høy, ellers beholdes gammel verdi.

  Ekstra anvendelse: Lagring av data

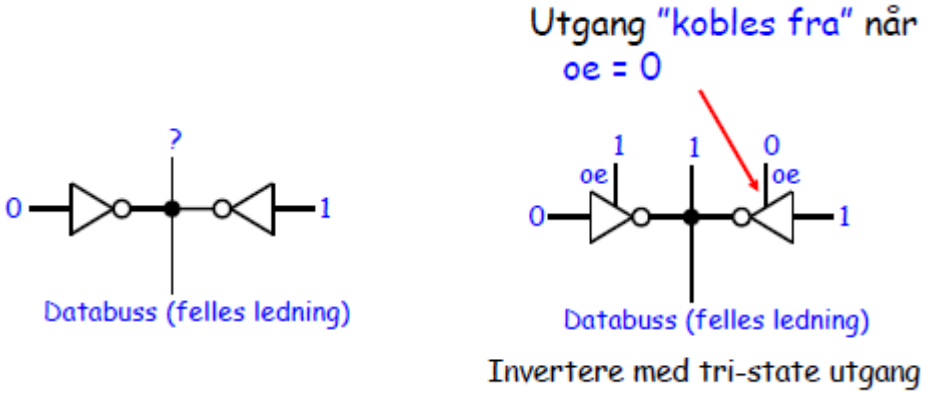


✗ Tri-state utgang

Ønske: Flere utganger skal kunne dele samme databuss

Problem: Porter som ikke er aktive kan ikke ”slås av” – gir elektrisk konflikt på utgangen

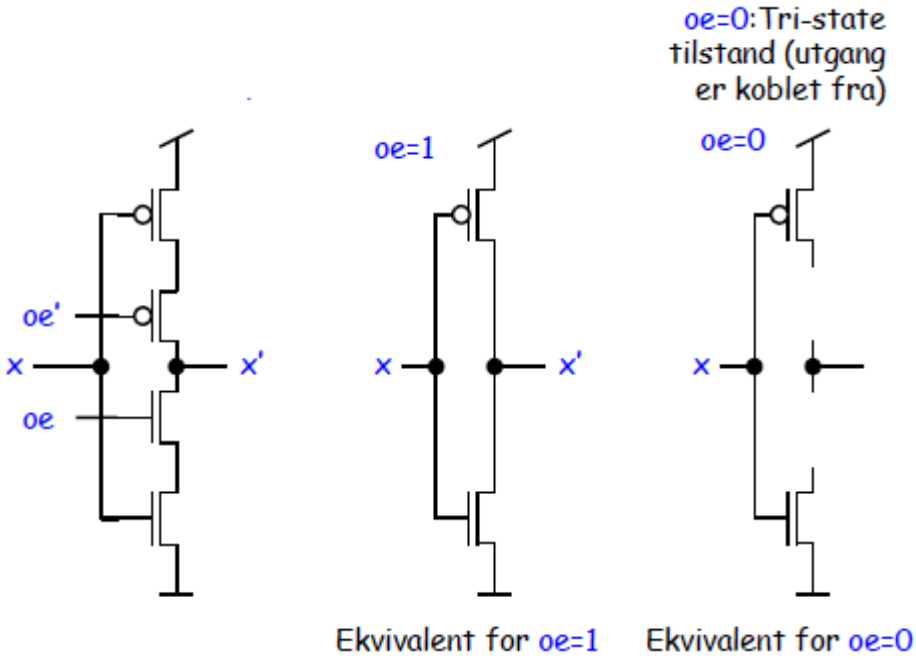
Løsning: Tri-state utgang



Invertere med tri-state utgang

– Implementasjon:

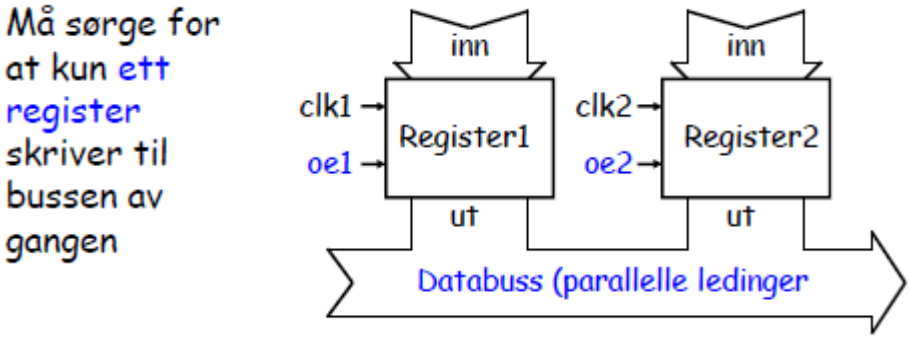
Mulig implementasjon av tri-state inverter (oe - output enable)



Ekvivalent for oe=1 Ekvivalent for oe=0

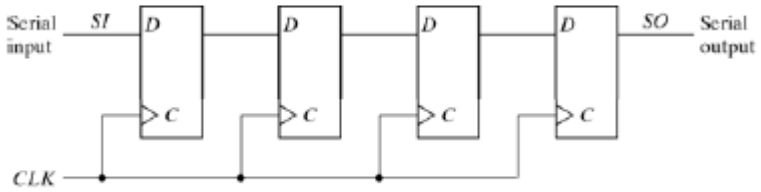
✗ Felles databuss

Hvis man bruker tri-state utganger kan flere registre dele samme databuss



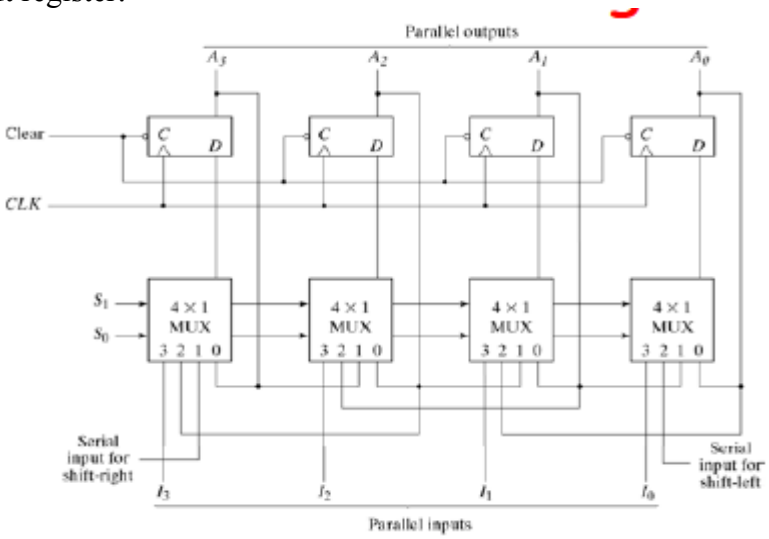
✗ Shift-register

Shiftregister – Seriell inngangsdata klokkes gjennom registeret ett trinn per klokkeperiode



(animasjoner: www.play-hookey.com)

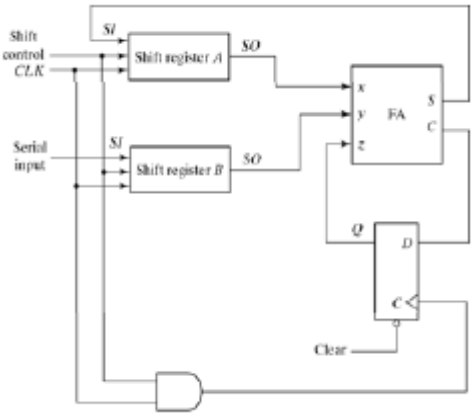
✗ Universelt shift register:



Eksempel:

Hver kolonne med bits adderes for seg, carrybiten mellomlagres i en D-ff

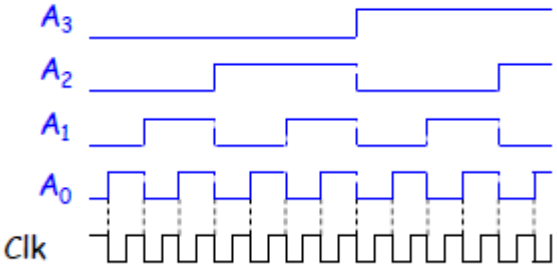
Resultatet sendes tilbake inn i Shift reg. A

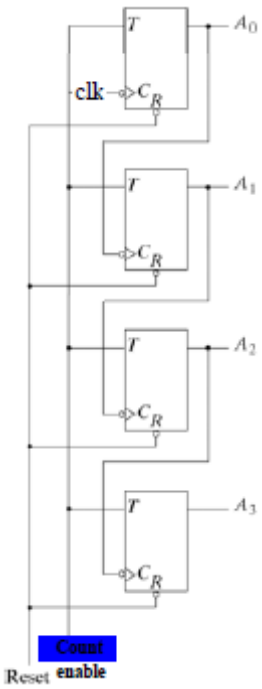


✗ Rippeltellere

Her laget med T flip-flopper.

Utgang ”toggler” toggler på negativ flanke

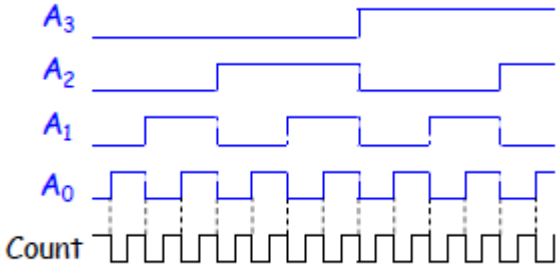


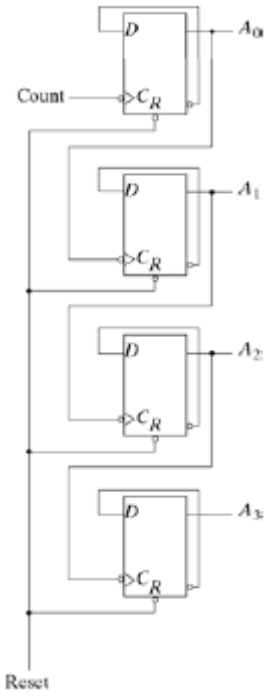


✗ Rippeltellere

Her laget med D flip-flopper.

Fører invertert utgang tilbake på negativ klokke



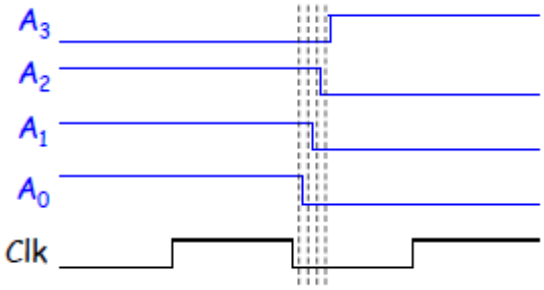


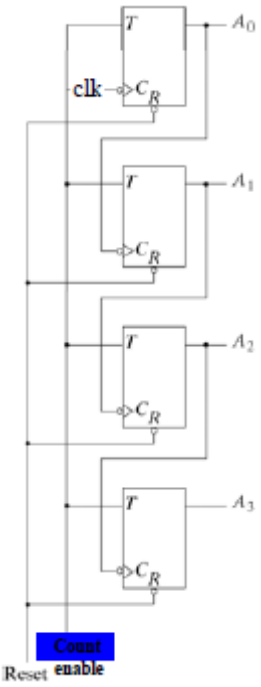
✗ Rippeltellere

Signalet ripler/propagerer gjennom trinnene

Problem:

Portforsinkelse gir temporært gale utgangsverdier

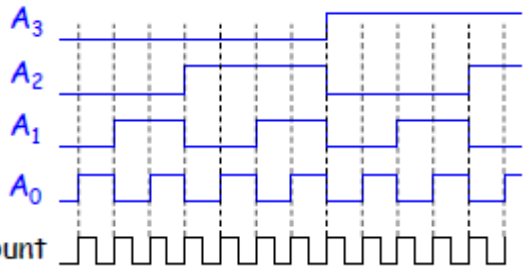


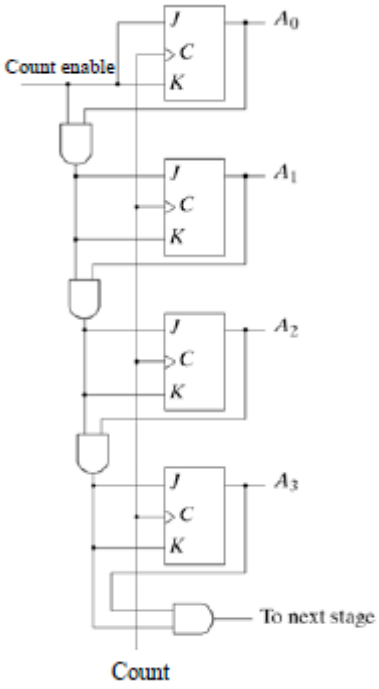


✗ Synkronteller:

Utganger skifter verdi samtidig / synkront

J	K	Q
0	0	uforandret
0	1	0
1	0	1
1	1	Q'

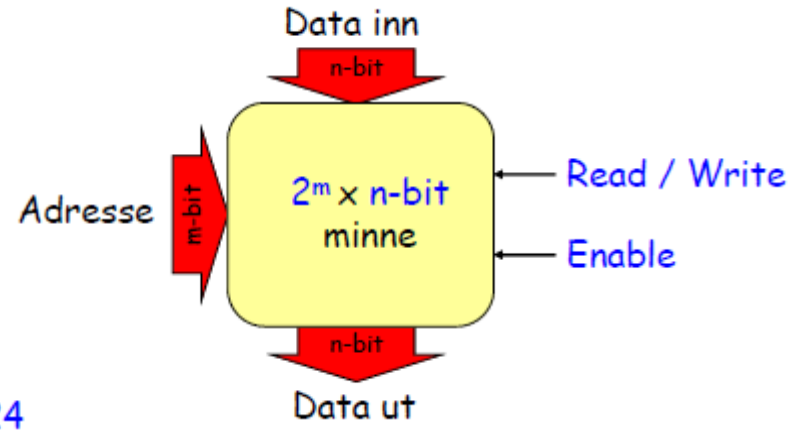




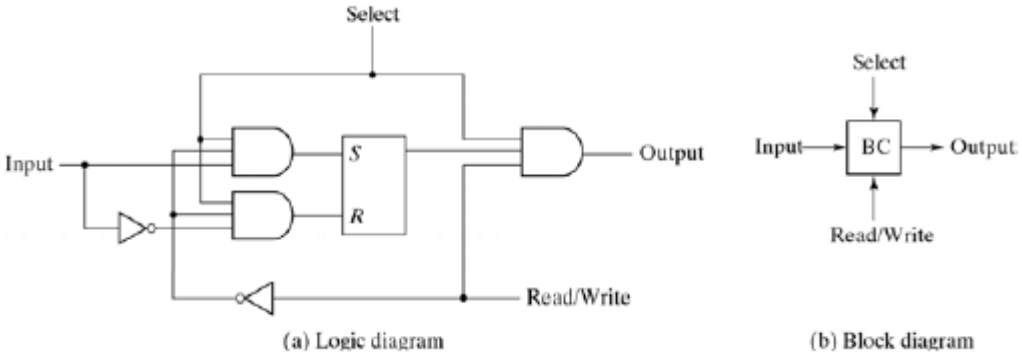
KAPITTEL 7

Minne

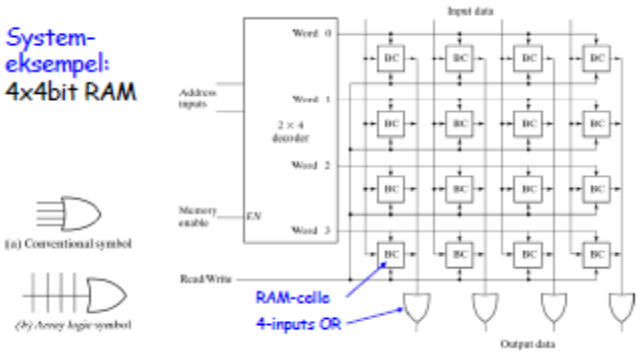
- ✗ Minne – generelt
  Hvert bit lagres i egen minne-celle



- ✗ Minne – teoretisk cellestruktur:
  RAM celle laget av standardkomponenter
  Separat datainngang/datautgang



- ✗ RAM – teoretisk system



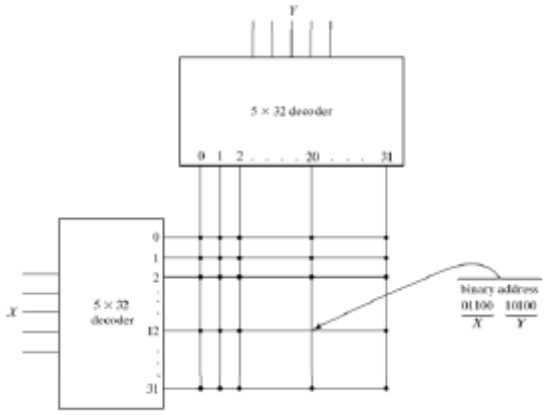
✗ Adresse-dekoding

Deler opp adressen i X og Y, sparer svært mye logikk

Eksempel:

X er de 5 mest signifikante adressebits.

Y er de 5 minst signifikante adressebits.



✗ SRAM (Static-Random-Access-Memory)

+ Raskt, <1ns – CPU cacheminne
+ Trenger ikke refresh, latch-basert

- Høyt strømforbruk
- Tar stor plass, 4 eller 6 transistorer / celle
- Flyktig (volatile), data forsvinner når strømforsyningen slås av

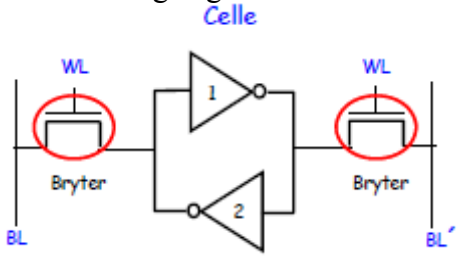
– Cellestruktur:

SRAM, 6-transistor latch-basert cellearkitektur

En celle lagrer ett bit

Ett bit representeres differensielt ved BL og BL´

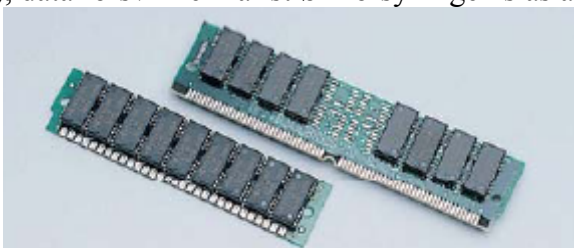
- Write: WL=1 lukker bryterne, spenningene BL og BL' overstyrer inverterne – spenningene lagres
- Read: BL og BL' slippes løs, WL=1 lukker bryterne, inverterne drar opp/ned spenningene på BL og BL', dette registreres av utlesningslogikken



✗ DRAM (Dynamic-Random-Access-Memory)

+ Kompakt/billig struktur, 1 transistor / celle

- Trenger oppfriskning (refresh) hver 20-30ms
- Flyktig (volatile), data forsvinner når strømforsyningen slås av

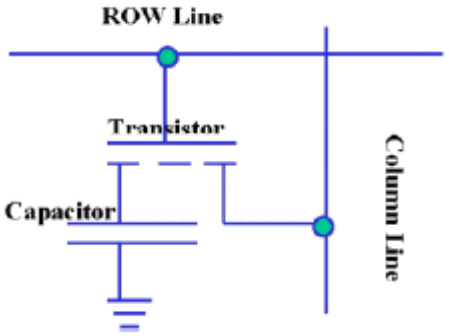


– Cellestruktur:

DRAM, en-transistor cellestruktur

Write: setter ROW Line til ”1”, legger ”0” eller ”1” på Column Line, transistoren kobler Column spenningen inn til kondensatoren

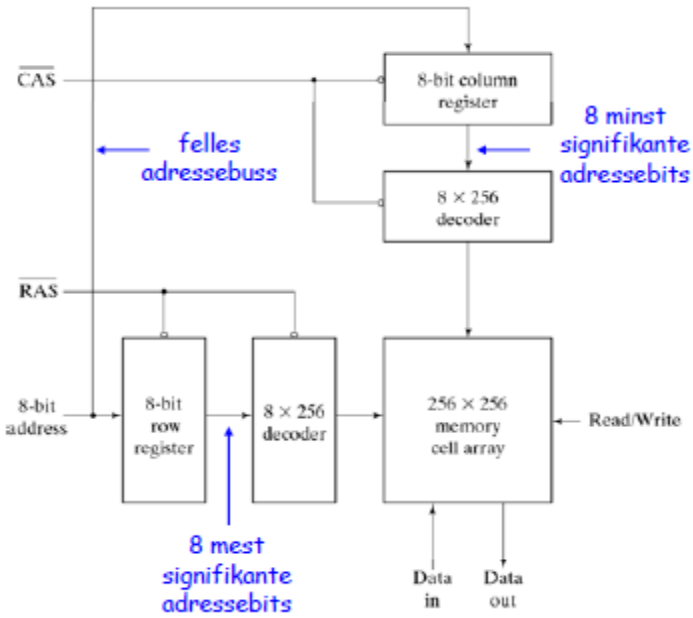
Read: setter ROW Line til ”1”,transistoren kobler kondensatorspenningen ut til Column Line



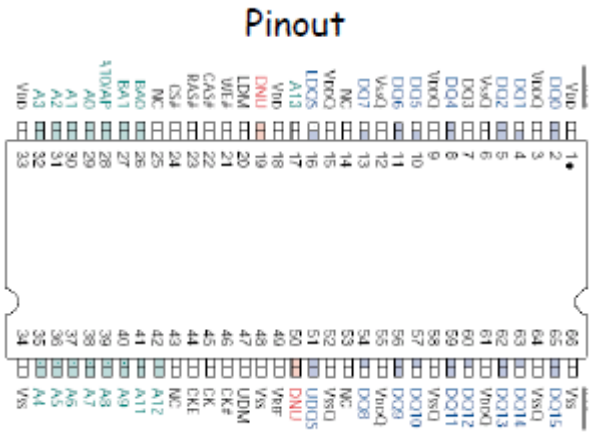
– Addressedekoding

Vanlig arraydekoding, men delt adressebuss

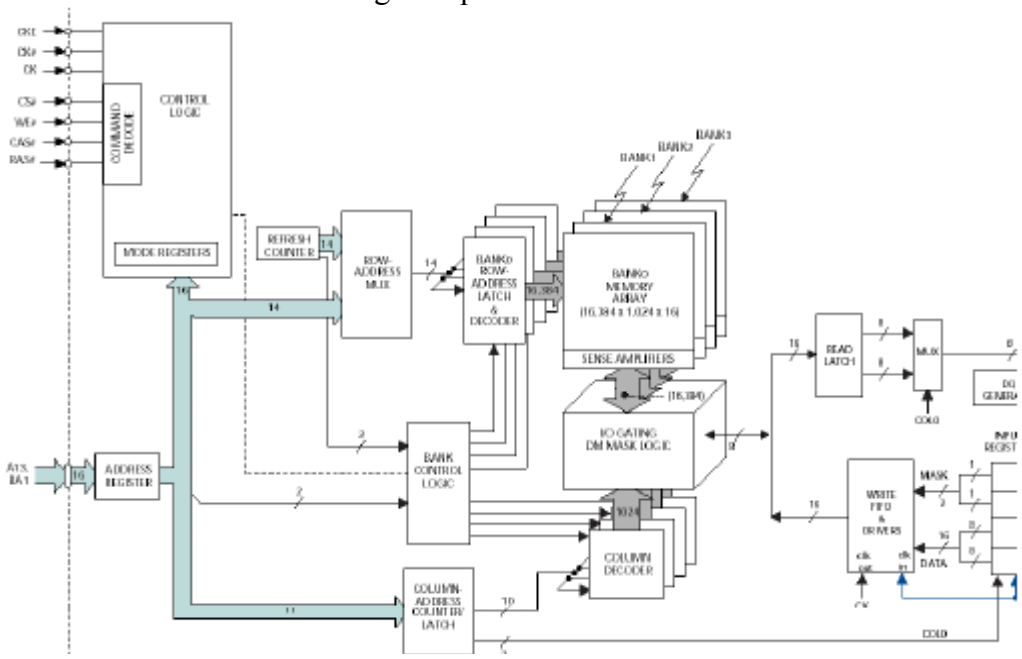
Fordeler adresser til rad/kolonne ved bruk av CAS/RAS-signal, sparer pinner



- Eksempel:
  - Micron – MT46V128M8
    1Gb single chip DDR RAM



✗ Micron – MT46V128M8 1Gb single chip DDR RAM



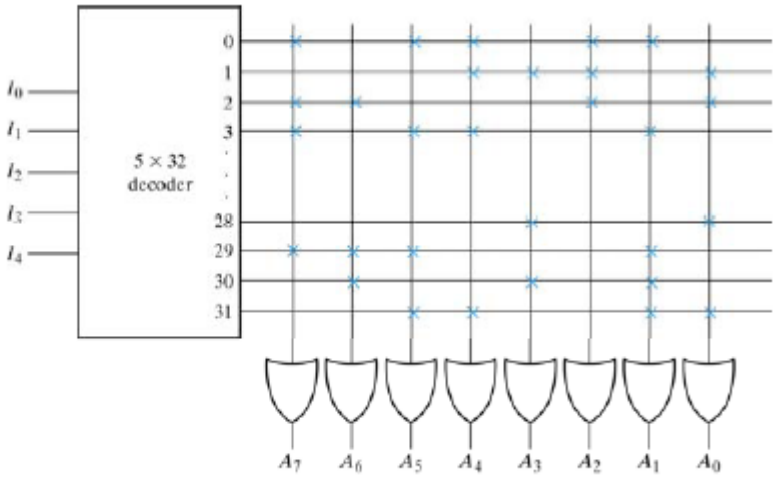
✗ ROM (Read-Only-Memory)

+ Ikke-flyktig (non-volatile), data vedvarer når strømforsyningen slås av
+ Rask operasjon <1ns
+ Lavt strømforbruk
+ Kan lages meget kompakt, med kun fysiske koblinger (sikringer)

- Må programmeres på fabrikken, aktuell for store produksjonskvanta

- Systemeksempel:

Hver celle som inneholder "1" er et fysisk koblingspunkt



✗ EPROM (Erasable-Programmable-Read-Only-Memory)

+ Ikke-flyktig (non-volatile), data vedvarer når strømforsyningen slås av

- Data kan legges inn av forbruker, data kan slettes ved bruk av ultrafiolett lys (eget lokk i pakken). Data må slettes (erase) før ny data kan legges inn

✗ Flash

Billig og rask variant av EEPROM. Meget aktuell hukommelse for mange anvendelser

+ Ikke-flyktig (non-volatile), data vedvarer når strømforsyningen slås av
+ Kan lages meget kompakt, hver celle består av kun en transistor

- Eksempel:

2003 - NAND flash

- Samsung
  K9W8G08U1M
  4Gb/chip !!
  >2 000 00 000 transistorer på samme chip

- CompactFlash
  CompactFlash – et lite kort med flere I/O protokoller, blant annet IDE interface
  CompactFlash kan erstatte IDE harddisk – ingen bevegelige deler
- Eksempel:
  NextFlash
  NX29F010-90PL

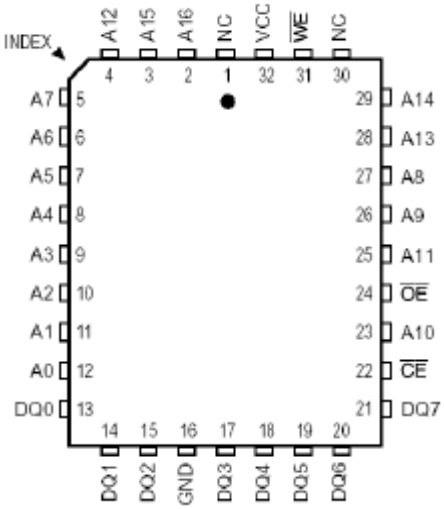


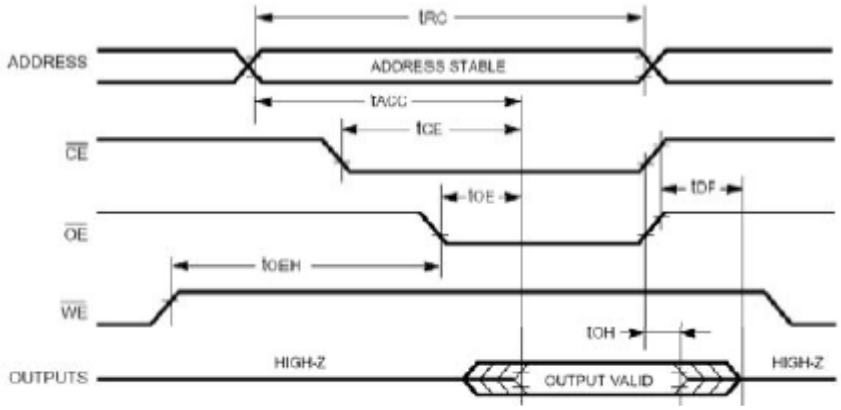
Table 1. Pin Descriptions

A0-A16	Address Inputs
DQ0-DQ7	Data Inputs/Outputs
$\overline{CE}$	Chip Enable Input
$\overline{OE}$	Output Enable Input
$\overline{WE}$	Write Enable Input
Vcc	Power Supply Voltage
GND	Ground
NC	No Internal Connection

- Eksempel:
  NX29F010-90PL

  Read: La (CE)' = 0, (OE)' = 0 og (WE)' = 1, data kommer senest ut tACC = 90ns etter ny adresse inn er stabil (verson 90PL)

  Erase / Write – kommandostyrt, se datablad

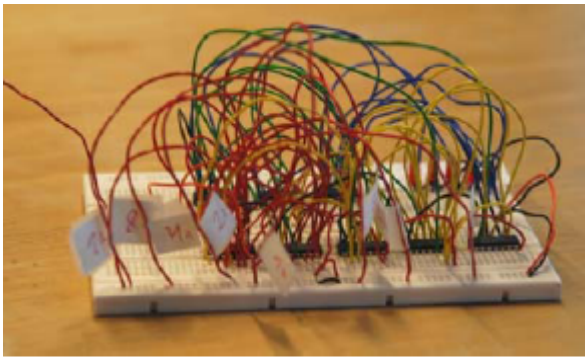


FPGA/CPLD-design

Xilinx

- ✗ Breadboard SS design

  Breadboard – grei måte å lage enkle test kretser på
  Ikke egnet for større digitale system



Aldri mer?

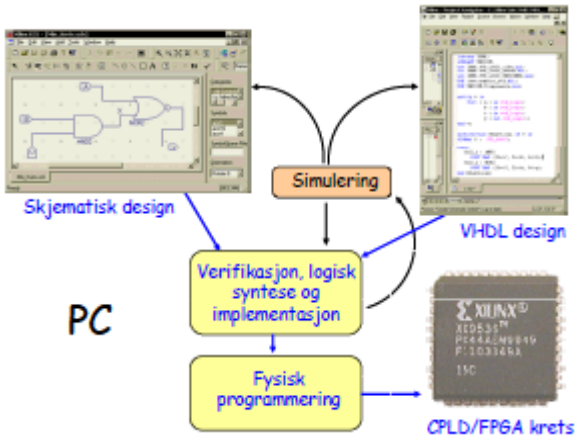
- ✗ Alternativ designmetode

  Større digitale system kan designes og verifiseres på PC - så lastes ned i programmerbare digitale kretser

  CPLD – Complex Programmable Logic Device
  - (V)LSI krets bestående av typisk 1k til >10k porter/flip-flopper som kan kobles sammen fra PC
  - Koblingsbeskrivelsen ligger lagret i FLASH lokalt i CPLD-brikke

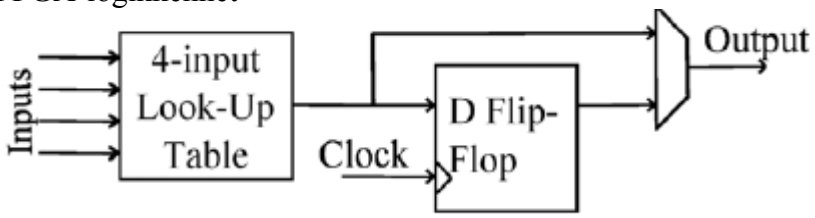
  FPGA – Field Programmable Gate Array
  - VLSI krets bestående av typisk 10k til >1M porter/flip-flopper som kan kobles sammen fra PC
  - Koblingsbeskrivelsen lagres vanligvis i en PROM- brikke som kobles til den SRAM-baserte FPGA-kretsen

  – CPLD/FPGA design



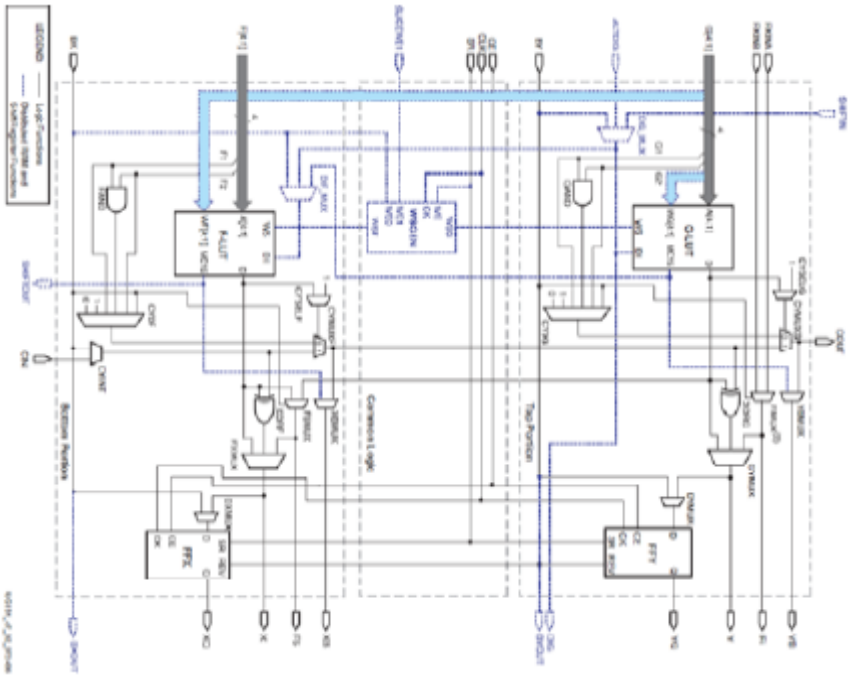
- Eksempel: Xilinx FPGA-er
  - Spartan3-familien
    - Opptil 3.4 millioner logiske porter (53k ”logic cells”)
  - Virtex5-familien
    - Opptil 330k ”logic cells” --> 21 millioner porter(?)
- FPGA-oppbygning

  En FPGA-brikke inneholder et gitt antall rekonfigurerbare logikkenheter
  Hver logikkenhet inneholder typisk en oppslagstabell (LUT) og en flip-flop
  Logikkenhetene kobles sammen slik softvaren finner hensiktmessig for å implementere designet vårt
- Generell FPGA-logikkenhet

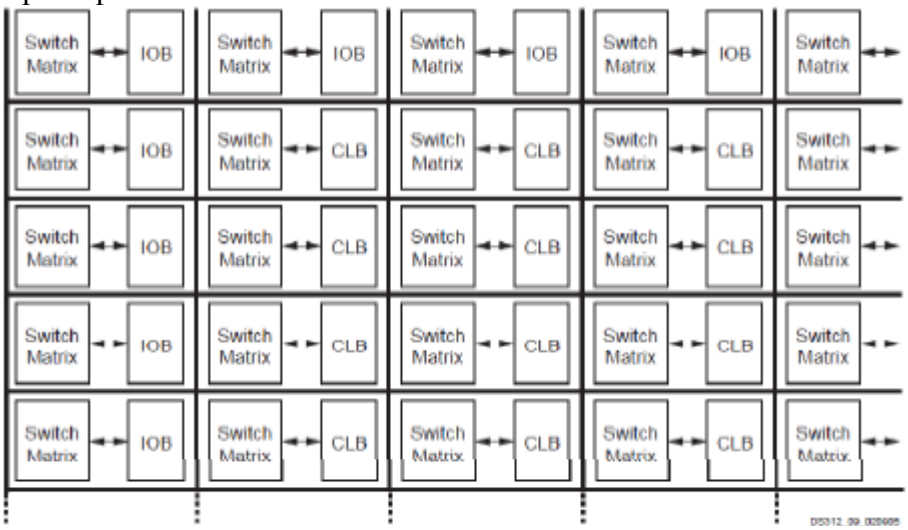


  LUT-en kan brukes til å implementere en hvilken som helst 4-variabels boolsk funksjon / kombinatorisk krets med 4 innganger og 1 utgang
- Eksempel:

  ”slice” i Xilinx Spartan3

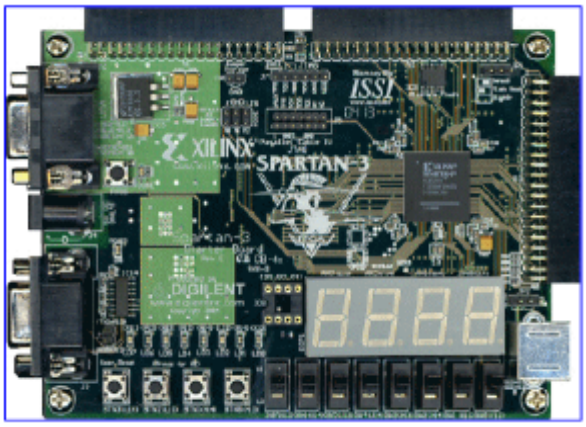


- Eksempel: Spartan3



Programmerbare brytermatriser styrer koblinger mellom logikkenheter

- Eksempel: Spartan3 utviklingskort



Bra for prototyping: RS232, VGA, PS/2, generelle pinner, minne, brytere, lysdioder

✗ Kretsbeskrivelse

Man kan generelt beskrive en krets på to måter

1) Skjematisk design

Man tegner opp kretsen bestående av porter, flip-flopper og/eller større komponenter på et tegneark i softvaren

2) Tekstlig beskrivelse

Man lager en tekstlig beskrivelse av funksjonaliteten til kretsen
Vanlig beskrivelsesspråk: VHDL (VHSIC* Hardware Description Language)
Alternativ: Verilog

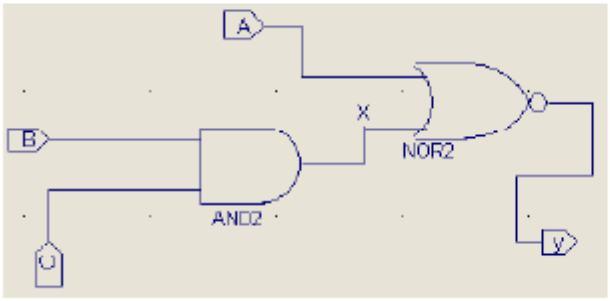
(*Very High Speed Integrated Circuit)

- Designmetoder

  1) Når kretsen er beskrevet kan den verifiseres og simuleres i softvaren
  2) Virker kretsen som den skal, lar man softvaren omforme beskrivelsen slik at den kan implementeres ved hjelp av sammenkoblinger av logikkenheter i FPGA/CPLD-brikken
  3) Man laster så ned den omformede beskrivelsen til FPGA/CPLD-brikken og man får et kompakt digitalt system

- Kretsbeskrivelse

Eksempel: 2 måter å beskrive samme krets på



Skjematisk beskrivelse

```
library IEEE;
LIBRARY UNISIM;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;
USE UNISIM.Vcomponents.ALL;

entity v is
    Port ( a : in std_logic;
           b : in std_logic;
           c : in std_logic;
           y : out std_logic);
end v;

architecture Behavioral of v is
SIGNAL X :  STD_LOGIC;

begin
   XLXI_1 : AND2
      PORT MAP (I0=>C, I1=>B, O=>X);
   XLXI_2 : NOR2
      PORT MAP (I0=>X, I1=>A, O=>y);
end Behavioral;
```

VHDL beskrivelse

VHDL

- ✗ VHDL ((Very High Speed Integrated Circuits) Hardware Description Language)
  VHDL er en tekstlig beskrivelse av et digitalt design
  VHDL versus skjematisk design

  + Mulighet for å beskrive kretser på et høyere abstraksjonsnivå
  + Raskere design

  - Kan miste detaljkontroll på portnivå

- ✗ VHDL versus C(++)/Java
  - Forskjeller:
    - VHDL beskriver ekte parallelle fysiske systemer (med portforsinkelse) - konkurrens
    - C(++)/Java/assembler beskriver oppførsel til serielle fysiske systemer (mikroprosessor/mikrokontroller)

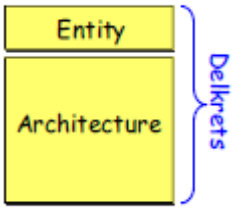
      + Parallelle systemer kan bli mye raskere enn serielle systemer

      - Design av ekte parallelle systemer er generelt mer krevende enn design av serielle systemer

      ± Design i VHDL krever vanligvis forståelse av den underliggende hardware

- ✗ Tre VHDL "abstraksjonsnivå"
  - "Behavioral"-beskrivelse:
    Beskriver i høynivå hvordan kretsen oppfører seg over tid. Beskrivelsen er ikke nødvendigvis tenkt å overføres til hardware
  - "Dataflow"-beskrivelse:
    Beskriver kretsen hovedsakelig ved bruk av Boolske uttrykk. Overlater til kompilator å velge porter og rekke ledninger
  - "Structural"-beskrivelse:
    - Beskriver sammenkoblingen av ferdigdefinerte porter/delkretser (Trekker kun ledninger mellom portene/delkretsene), (nettliste)
- ✗ Generell VHDL struktur – delkretser
  En VHDL beskrivelse består av en eller flere delkretser

  Hver delkrets har:
  1) Entity: Et grensesnitt som definerer delkretsens innganger utganger (pinout)
  2) Architecture: En beskrivelse av delkretsens interne virkemåte



og

- ”Dataflow”-beskrivelse
  - Beskriver ”flyten” av data imellom noder/registre
  - Beskriver ofte systemet ved hjelp av Boolske uttrykk
  - Har vanligvis ikke noe forhold til tidsbegrepet –
    systemet som beskrives er konkurrent (ekte parallelt, men med portforsinkelse)
  - Gir ofte godt samsvar mellom beskrivelse og hardware

- Eksempel – ”dataflow”-beskrivelse

VHDL "dataflow"-beskrivelse av en delkrets

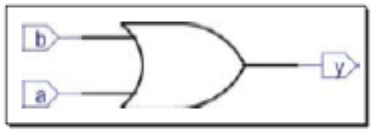
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity delkrets1 is
    Port ( a   : in std_logic;
           b   : in std_logic;
           y   : out std_logic);
end delkrets1;

architecture dataflow of delkrets1 is
begin

   y <= a or b;

end dataflow;
```

Bibliotek

Grensesnitt

Innhold

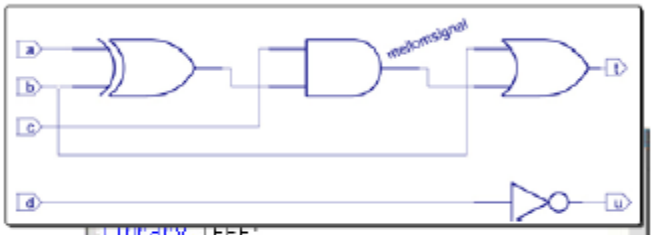
Delkrets



Ekvivalent skjematisk beskrivelse

- Eksempel II

  Interne signaler kan defineres lokalt inne i ”architecture” blokken




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity delkrets2 is
    Port ( a : in std_logic;
           b : in std_logic;
           c : in std_logic;
           d : in std_logic;
           t : out std_logic;
           u : out std_logic);
end delkrets2;

architecture behavioral of delkrets2 is

signal mellomsignal : std_logic;
begin

    mellomsignal <= (a xor b) and c;
               t <= mellomsignal or b;
               u <= not(d);

end behavioral;
```

Deklarasjon av internt signal

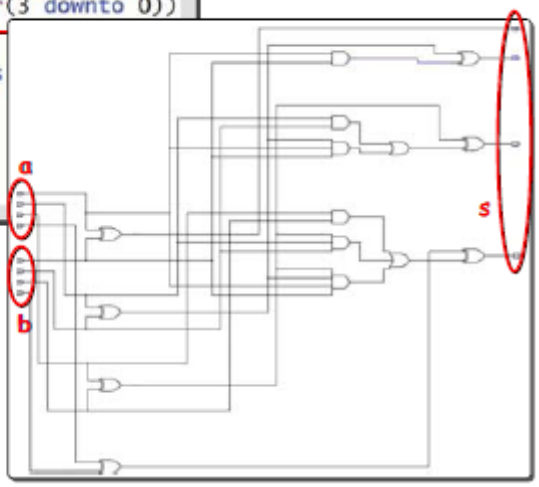
NB: Selv om program-linjene er sekvensielle definerer de et parallelt (konkurrent) system

✗ Eksempel III

3 bits adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity delkrets3 is
    Port ( a   : in std_logic_vector(3 downto 0);
           b   : in std_logic_vector(3 downto 0);
           s   : out std_logic_vector(3 downto 0));
end delkrets3;

architecture dataflow of delkrets3 is
begin

   s <= a + b;

end dataflow;
```

Lager binære ord (databusser) ved bruk av "_vector" deklarasjon



✗ Kondisjonell beskrivelse

Kondisjonell beskrivelse – when/else:

Syntaks:

```
Utgang <=   verdi1 when test1 else
            verdi2 when test2 else
            verdi3;
```

NB: alle tilstander bør defineres. Hvis ikke - har man mindre kontroll på det syntetiserte resultatet*

Hvis flere tester slår til vil første tilordning være gjeldende (prioritet)

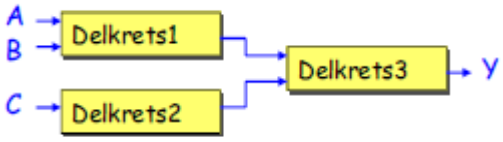
(*Noen synteseverktøy implementerer latcher (kan holde utgangsverdien), andre ikke)

- Eksempel, 4-input MUX:

```
entity muxen is
    Port ( in1 : in std_logic;
           in2 : in std_logic;
           in3 : in std_logic;
           in4 : in std_logic;
           sel : in std_logic_vector(1 downto 0);
           y   : out std_logic);
end muxen;

architecture Behavioral of muxen is
begin
   y <= in1 when sel="00" else
        in2 when sel="01" else
        in3 when sel="10" else
        in4;

end Behavioral;
```

✗ ”Structural”-beskrivelse

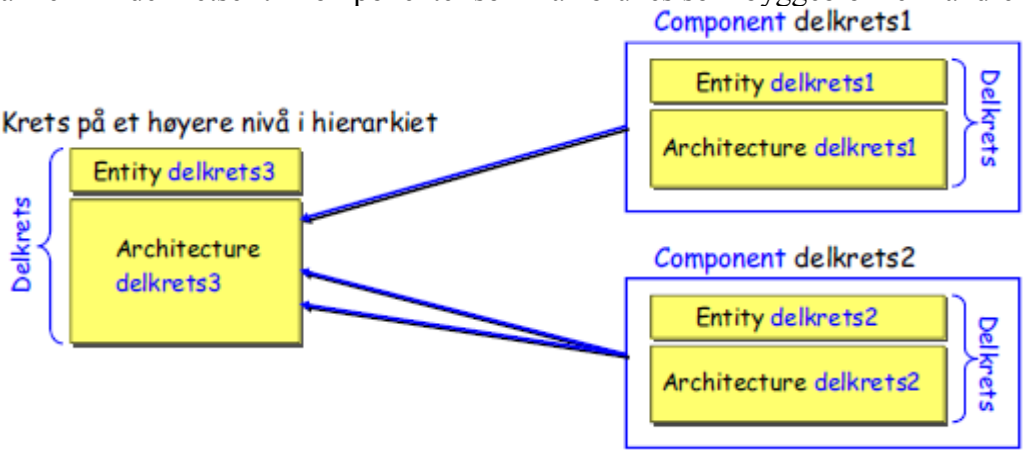
Beskriver sammenkoblingen av ferdigdefinerte delkretser (component’s)
(Trekker kun ledningene mellom delkretsene)



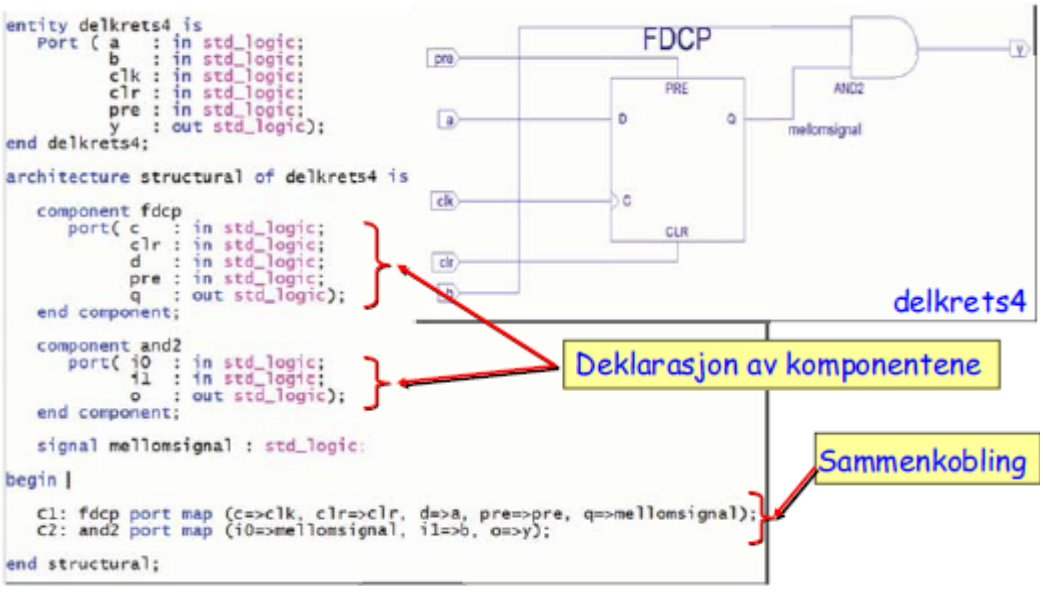
- Delkretsene kan man enten selv definere eller hente ferdig fra bibliotek
- Enkle standardkomponenter som and2..and8, or2.., fdcp (D flip-flop with clear and preset), osv.. finner man i biblioteket
- ”Structural”-beskrivelse kan benyttes på laveste nivå ved å koble sammen porter (evt. transistorer)
- ”Structural”-beskrivelse kan også brukes til å beskrive sammenkoblingen av store komplekse byggeblokker (eks. ALU / memory osv.). På denne måten kan man lage store komplekse systemer hierarkiske og oversiktelige

– ”Structural”-beskrivelse gir hierarkisk design

Pakker inn delkretser til komponenter som kan brukes som byggeblokker i andre delkretser

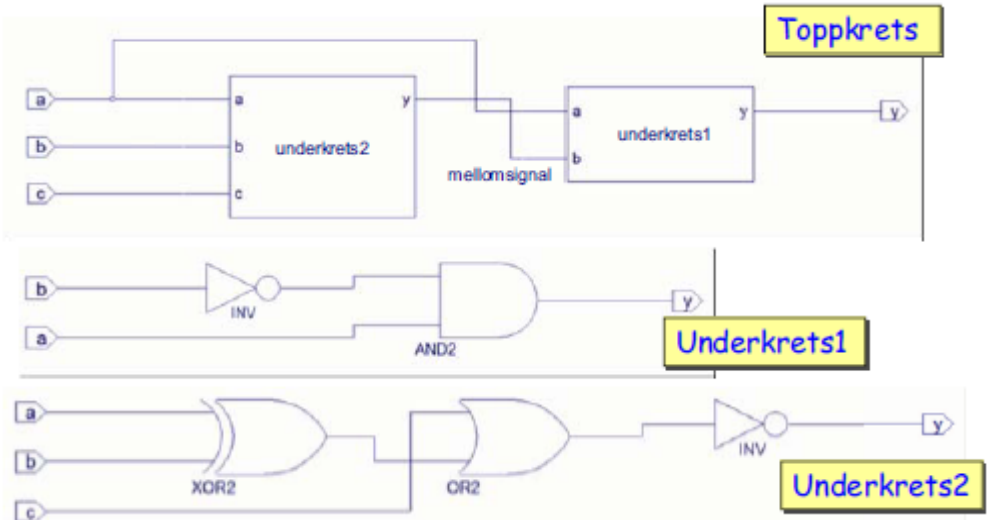


- Eksempel: Bruk av 2 ferdige bibliotekkomponenter – and2 og fdcp



- Eksempel II

  Lager 2 egne underkretser ved ”dataflow”. Setter sammen til krets på toppnivå ved ”structural”



- første del av beskrivelsen:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity underkrets1 is
   Port ( a   : in std_logic;
          b   : in std_logic;
          y   : out std_logic);
end underkrets1;

architecture dataflow of underkrets1 is
begin

   y <= a and not(b);

end dataflow;

library IEEE;                    -- Må inkluderes
use IEEE.STD_LOGIC_1164.ALL; -- før hver entity

entity underkrets2 is
   Port ( a   : in std_logic;
          b   : in std_logic;
          c   : in std_logic;
          y   : out std_logic);
end underkrets2;

architecture dataflow of underkrets2 is
begin

   y <= (a xor b) nor c;

end dataflow;
```

underkrets1

underkrets2

- Siste del av beskrivelsen

```
library IEEE;                  -- Må inkluderes
use IEEE.STD_LOGIC_1164.ALL;   -- før hver entity

entity toppkrets is
   Port ( a   : in std_logic;
          b   : in std_logic;
          c   : in std_logic;
          y   : out std_logic);
end toppkrets;

architecture structural of toppkrets is

   component underkrets1
      port( a   : in std_logic;
            b   : in std_logic;
            y   : out std_logic);
   end component;

   component underkrets2
      port( a   : in std_logic;
            b   : in std_logic;
            c   : in std_logic;
            y   : out std_logic);
   end component;

   signal mellomsignal : std_logic;

begin

   U1: underkrets1 port map (a=>a, b=>mellomsignal, y=>y);
   U2: underkrets2 port map (a=>a, b=>b, c=>c, y=>mellomsignal);

end structural;
```

Deklarasjon av underkrets1

Deklarasjon av underkrets2

Sammenkobling

✗ ”Behaviour”-beskrivelse
  - ”Behaviour”-beskrivelse er en høynivåbeskrivelse av systemets oppførsel over tid
  - Begrepet tid skiller ofte ”behaviour”-beskrivelse fra laverenivå-beskrivelse (”structural”/”dataflow”)
  - Komplekse ”behaviour”-beskrivelser kan ikke alltid overføres til hardware (syntetiseres)
  - Komplekse ”behaviour”-beskrivelser brukes ofte for å generere test bencher (simuleringsmodeller av systemet)
  - I komplekse ”behaviour”-beskrivelser kan man ”glemme” den underliggende hardwaren og beskrive systemet på en måte som nærmer seg Java/C(++)

✗ ”Process” – D latch eksempel Beskrivelse av en D latch

```
entity Dlatch is
    Port ( D   : in std_logic;
           clk : in std_logic;
           Q   : out std_logic);
end Dlatch;

architecture Behavioral of Dlatch is
begin

   process(clk,D)
   begin

      if (clk='1') then
         Q <= D;
      end if;

   end process;

end Behavioral;
```

Sensitivitets-liste

Processen starter opp hver gang clk eller D forandrer verdi (event)

Når clk går til '1' overføres verdien på D til Q

Når D forandres og clk='1' overføres verdien på D til Q (transparent latch).

✗ ”Process” – D flip-flop eksempel

- Beskrivelse av en D flip-flop

```
entity Dflipp is
    Port ( D   : in std_logic;
           clk : in std_logic;
           Q   : out std_logic);
end Dflipp;

architecture Behavioral of Dflipp is
begin

   process(clk)
   begin

      if (clk='1' and clk'event) then
         Q <= D;
      end if;

   end process;

end Behavioral;
```

Processen starter opp hver gang clk forandrer verdi

clk'event slår til når clk forandrer verdi

clk'event kan synes unødvendig, men er viktig for å garantere syntese av flip-flop og ikke latch

Merk at på positiv clk-flanke overføres verdien på D til Q. Ellers skjer det ikke noe

✗ ”Process” – MUX eksempel

```
entity Mux2 is
    Port ( A   : in std_logic;
           B   : in std_logic;
           Sel : in std_logic;
           Q   : out std_logic);
end Mux2;

architecture Behavioral of Mux2 is
begin

   process(A,B,Sel)
   begin
      if Sel='0' then
         Q <= A;
      else
         Q <= B;
      end if;

   end process;

end Behavioral;
```

Beskrivelse av en 2-1 MUX

Når Sel går til '0' overføres verdien på A til Q.

Når Sel går til '1' overføres verdien på B til Q.

Når verdien på A eller B forandres overføres selektert verdi til Q

Generelt: Når alle input-signaler er spesifisert i sensitivitets-listen og alle utganger alltid gis en verdi, syntetiseres en kombinatorisk krets

✗ Generelt om processer

En process kan beskrive både kombinatoriske kretser (med hukommelse) og sekvensielle kretser

Vil man beskrive en kombinatorisk krets:

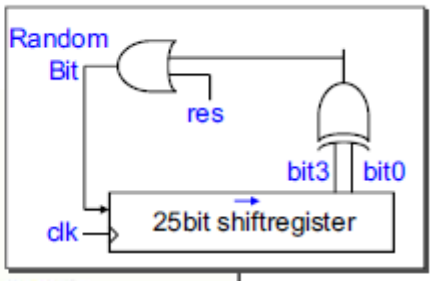
- Ta med alle inngangssignaler i sensitivitets-listen
- Pass på at utgangssignalene alltid blir tilordnet en verdi

Vil man beskrive en sekvensiell krets:

- Latcher bør generelt unngås: bruk flip-flop's ved å spesifisere flanketrigging ('event)
- Forsøk å gjøre den sekvensielle delen så liten og oversiktlig som mulig ved å "trekke" kombinatoriske element ut av processen

✗ Eksempel: pseudo-random generator

Mye brukt metode for pseudo-random tallgenerering



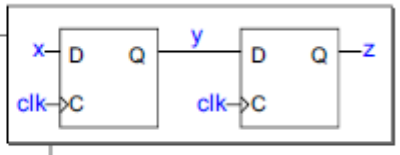

```
entity Rand is
    Port ( clk       : in std_logic;
           res       : in std_logic;
           randomBit : out std_logic);
end Rand;

architecture Behavioral of Rand is

signal ShiReg : std_logic_vector(24 downto 0);
signal tilbakeforing : std_logic;
begin

   process (clk)
   begin
      if clk='1' and clk'event then
         ShiReg <= tilbakeforing & ShiReg(24 downto 1);
      end if;
   end process;

   tilbakeforing <= (ShiReg(0) xor ShiReg(3)) or res;
   randomBit <= tilbakeforing;

end Behavioral;
```

Shift til høyre

Reset (oppstart)

✗ Tidspunkt for signaltilordning i processer

Viktig å vite: Inne i en "process" vil alle signaltilordninger (<=) utføres først i det øyeblikket processen terminerer ("end process" linjen utføres)




```
entity pipeline is
    Port ( x   : in std_logic;
           clk : in std_logic;
           z   : out std_logic);
end pipeline;

architecture Behavioral of pipeline is

signal y : std_logic;
begin

   process(clk)
   begin
      if clk='1' and clk'event then
      y <= x;
      z <= y;
      end if;
   end process;

end Behavioral;
```

Eksempel: signal z vil ikke nødvendigvis være lik signal x

✗ ”Enumerated” datatype

Kan definere datatyper selv ved bruk av ”type”

Syntaks:

```
type : egetTypeNavn is (hvasomhelst1, hvasomhelst2,… hvasomhelstN);
signal x : egetTypeNavn;
signal y : egetTypeNavn;
                :
```

hvasomhelst1 … hvasomhelstN syntetiseres til binære signaler

✗ Mer om tradisjonelle tester

For kondisjonell testing innen processer brukes ofte ”case”

Syntaks:

```
case : signalet is
when verdi1 =>
    uttrykk1;
when verdi2 =>
    uttrykk2;
when others =>
    uttrykk3;
end case;
```

Husk å spesifisere alle mulige verdier til signalet, evt. med ”others”

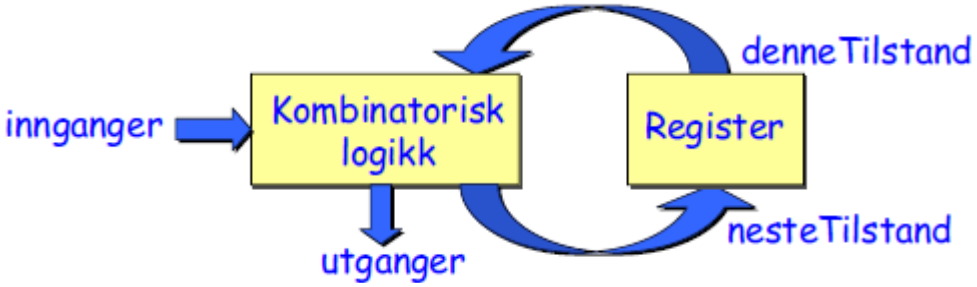
Pass på at verdi1, verdi2 er forskjellige

✗ Tilstandsmaskiner i VHDL

VHDL er spesielt egnet til å beskrive tilstandsmaskiner

En vanlig (og trygg) måte å spesifisere tilstandsmaskiner på er som følger:

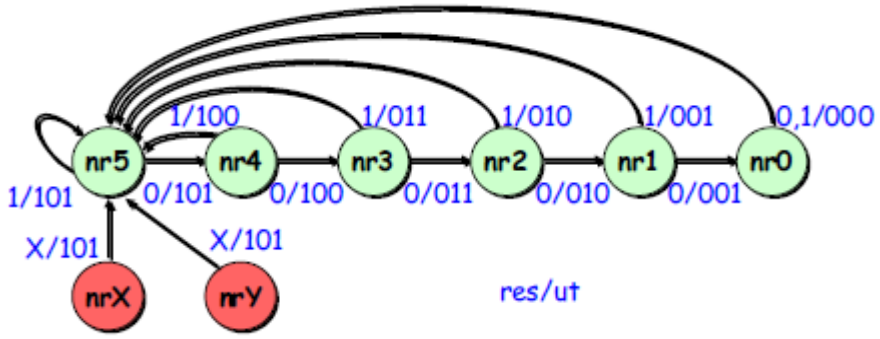
- Lag en egen process som implementerer all kombinatorisk logikk
- Lag en minimal process som kun implementerer registeret (flip-floppene)



✗ Eksempel på tilstandsmaskin

Lager en teller som teller sekvensen 5,4,3,2,1,0, 5,2…osv
Telleren har reset-inngang (res)

- Tilstandsdiagram:



Definerer navn på tilstander med ”type” (enumerated)
Må også i VHDL ta hensyn til ubrukte tilstander

- VHDL:

```
entity spesialTeller is
    Port ( Clk : in std_logic;
           Res : in std_logic;
           Ut  : out std_logic_vector(2 downto 0));
end spesialTeller;

architecture Behavioral of spesialTeller is

type S is (nr5, nr4, nr3, nr2, nr1, nr0);
signal TILSTAND_DENNE : S;
signal TILSTAND_NESTE : S;

begin

   process(Res, TILSTAND_DENNE) -- KOMBINATORISK del
   begin
      case TILSTAND_DENNE is
      when nr5 =>
         Ut <= "101";
         if Res='1' then
            TILSTAND_NESTE <= nr5;
         else
            TILSTAND_NESTE <= nr4;
         end if;
      when nr4 =>
         Ut <= "100";
         if Res='1' then
            TILSTAND_NESTE <= nr5;
         else
            TILSTAND_NESTE <= nr3;
         end if;
      when nr3 =>
         Ut <= "011";
         if Res='1' then
            TILSTAND_NESTE <= nr5;
         else
            TILSTAND_NESTE <= nr2;
         end if;
```

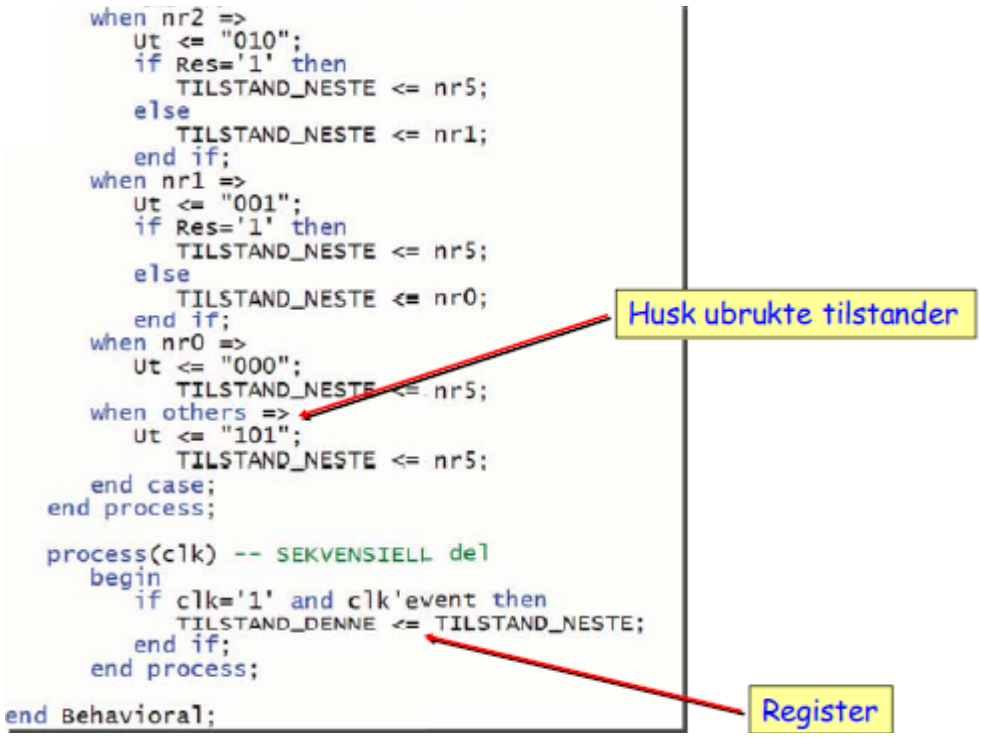
Spesifiserte tilstander

Ut fra register

Inn til register

Genererer utgangsverdi

Produserer neste tilstand basert på nåværende tilstand og inngangsverdi



✗ Datatyper i VHDL

Vanlige datatyper:

- Bit: '0', '1'
- Std_logic: '0', '1', '-', 'Z', 'W' + 4 andre
- Boolean true, false
- Type enumerated (brukerdefinert)
- Integer -2147483647 til +2147483647
- Time eksempel: 10ns
- *_vector(x downto y) vektor av Bit eller Std_logic

Std_logic er generelt foretrukket framfor bit
Alle ovenstående datatyper bortsett fra "time" kan syntetiseres til fysiske ledninger / registre

✗ Dataobjekter i VHDL

Vanlige dataobjekter i VHDL

- "Signal" - ledning/register. Tilordningssymbol: "<="
- "Variable" -temporær verdi brukt innenfor "process".
  Oppdateres umiddelbart (sekvensielt). Tilordningssymbol: ":="
- "Constant" - konstant definert for å lette oversikt

Alle objekter ovenfor kan inneholde datatypene definert på forrige side
Med tanke på syntese bør man bruke færrest mulig "variable". Bruk heller "signal" der det er mulig

✗ For – loop

Inne i processer kan man bruke kondisjonelle løkker av type ”for” – ”loop”
Syntaks:

```
for variabel1 in intervall loop
    uttrykk1;
    uttrykk2;
        :
end loop;
```

”variabel1” er av type variable (umiddelbar oppdatering) og blir automatisk deklarert i løkken

– Eksempel: for loop – partall tester

```
entity Parity is
    Port ( Data : in std_logic_vector(7 downto 0);
           P    : out std_logic);
end Parity;

architecture Behavioral of Parity is
begin

   process (Data)

   variable par: boolean;
   begin
      par := false;
      for i in 0 to 7 loop
         if Data(i) = '1' then
            par := not(par);
         end if;
      end loop;

      if par then
         P <= '1';
      else
         P <= '0';
      end if;

   end process;

end Behavioral;
```

P settes høy når antall '1'ere i Data bussen er et partall

Intern variable

Siden "par" er av type "variable" oppdateres par umiddelbart og systemet virker etter hensikten

✗ Operatorer:

Logical operators	and, or, nand, nor, xor, xnor, not
Relational operators	=, /=, <, <=, >, >=
Shift operators	sll, srl, sla, sra, rol, ror
Addition operators	+, -, &
Multiplying operators	*, /, mod, rem
Miscellaneous operators	**, abs

NB: ikke alle operatorer er definert for std_logic
NB: VHDL har ikke presedens for operatorer. ( ) må brukes

✗ Tips for VHDL design

VHDL kan beskrive både fysiske kretser samt test bencher for simulering.

- VHDL er et komplekst språk, og gir mange muligheter for å lage både gode og dårlige beskrivelser.
- Problemet med dårlige beskrivelse kan bli:
  - Forskjellig logisk oppførsel for syntese og hardware
  - Dårlig kode er ikke nødvendigvis portabel i mellom forskjellige synteseverktøy. Dette gjelder også for forskjellige simuleringsverktøy

✗ Tips for trygg syntese

- Bruk flip-flop's (registre) istedenfor latch'er der det er mulig
- Bruk " 'event " for å generere flip-flop'er
- Bruker man "process" til å lage kombinatorisk logikk – husk å inkludere alle inngangssignalene i sensitivitets-listen, samt å tilordne utgangssignalene verdier for alle kondisjonelle valg
- Ikke lag processer som trigger på begge klokkeflankene
- Separér ut kombinatorisk logikk fra processer som beskriver sekvensielle system
- For tilstandsmaskiner: husk å definere hopp ut av eventuelle udefinerte tilstander
- Husk at "if"-"else" som ikke spesifiserer alle muligheter genererer hukommelseselement (flip-flop'er)
- Ved bruk av "when"-"else", husk å spesifisere alle muligheter
- Minimaliser bruken av "variable". Begrens bruken til "for"-løkker
- Bruk helst typen "Std_logic" der det er mulig
- Ulike, men logisk like beskrivelser kan gi forskjellige hardware implementasjoner med hensyn på areal/strømforbruk/hastighet
- Unngå clock gating (manipulering av klokkesignalet)
- Unngå direkte kombinatorisk feedback. Feedback bør gå via registre
- Greit å vite: VHDL skiller ikke på små og store bokstaver
- Husk at dette er bare en introduksjon til VHDL. Språket inneholder mange flere mekanismer og egenskaper enn beskrevet her