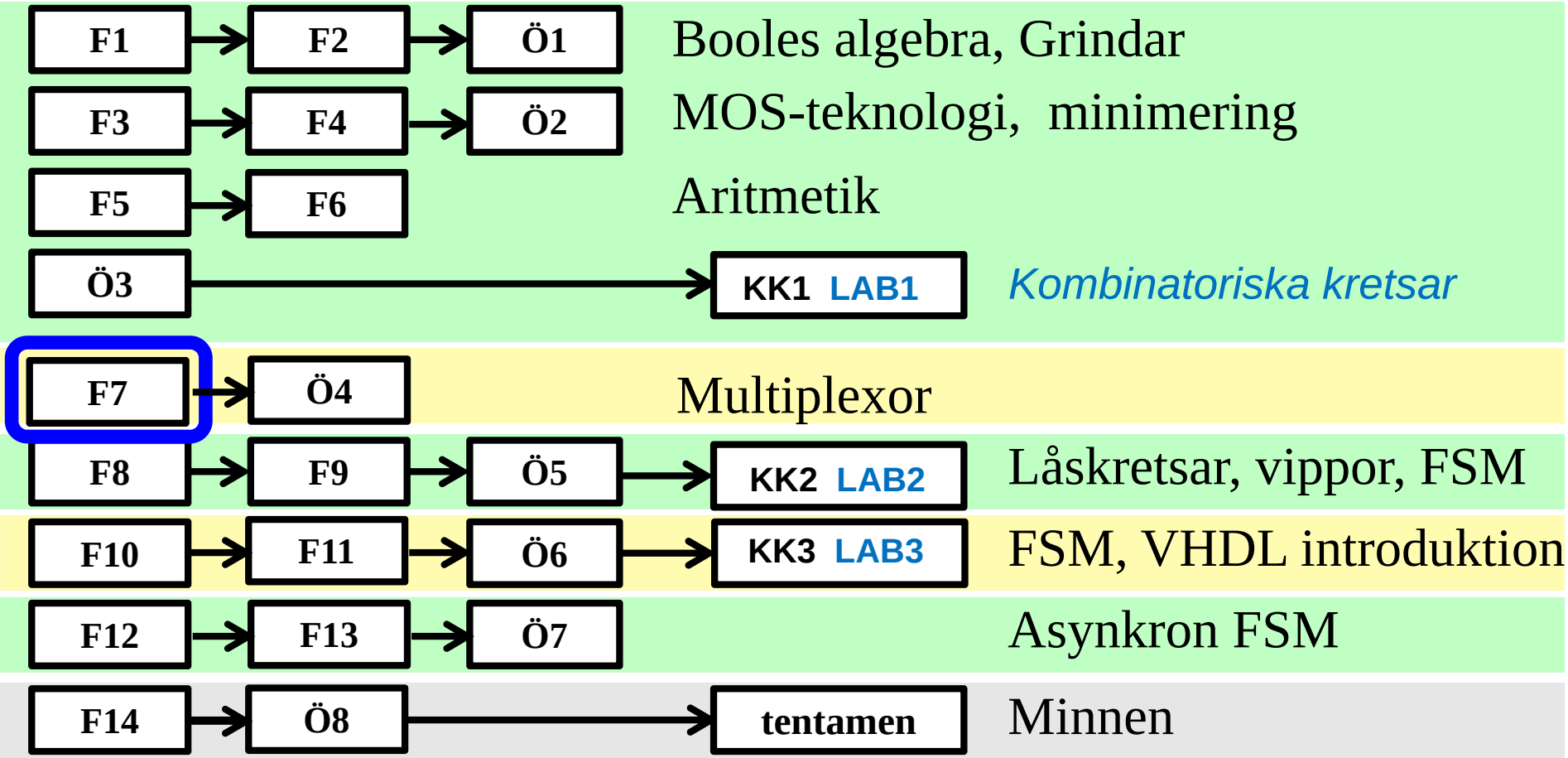
Digital Design IE1204

Föreläsningsbilder av William Sandqvist

F7 Kombinatorik-kretsar

Carl-Mikael Zetterling
bellman@kth.se

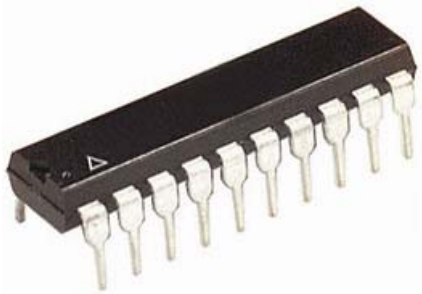
IE1204 Digital Design



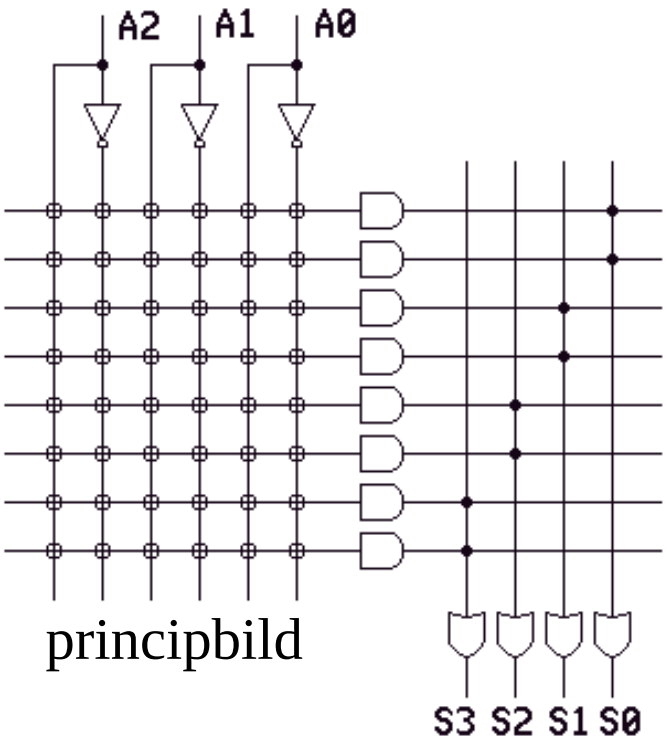
Föreläsningar och övningar bygger på varandra! Ta alltid igen det Du missat!
Läs på i förväg – delta i undervisningen – arbeta igenom materialet efteråt!

Detta har hänt i kursen …

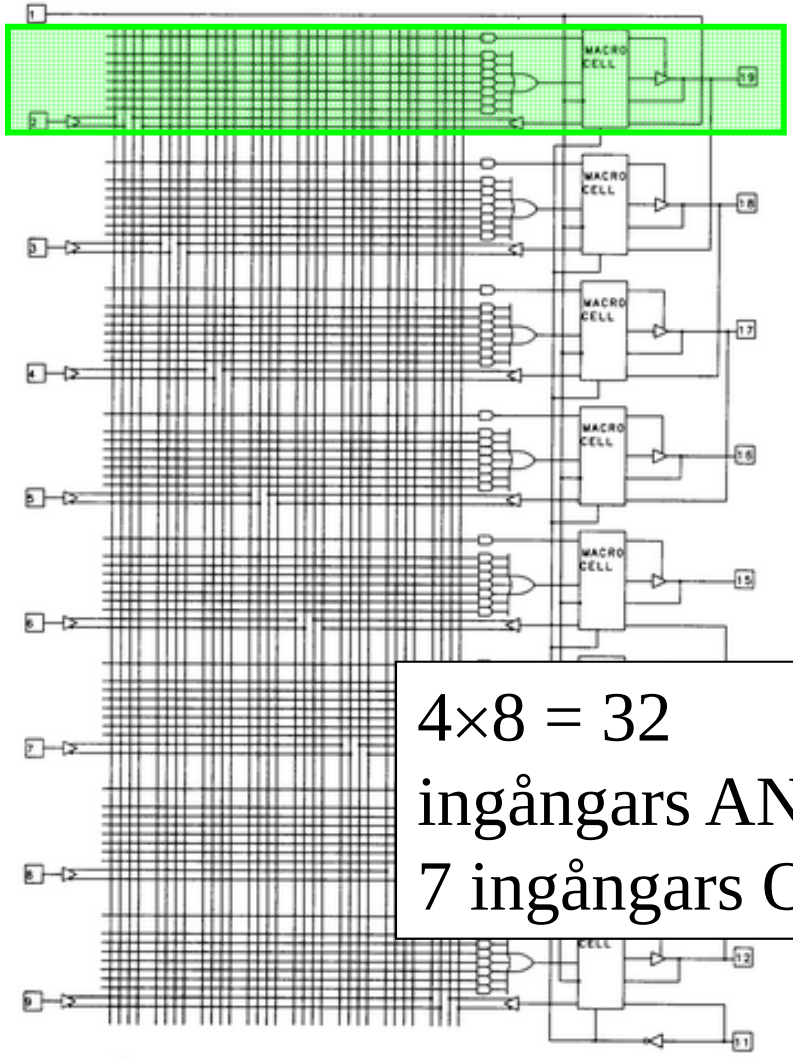
Decimala, hexadecimala, oktala och binära talsystemen
AND OR NOT EXOR EXNOR Sanningstabell, mintermer Maxtermer PS-form
Booles algebra SP-form deMorgans lag Bubbelgrindar Fullständig logik
NAND NOR CMOS grindar, standardkretsar Minimering med Karnaugh-diagram 2, 3, 4, 5, 6 variabler
Registeraritmetik tvåkomplementrepresentation av binära tal
Additionskretsar Multiplikationskrets Divisionskrets



PLD (tex. **PAL**)



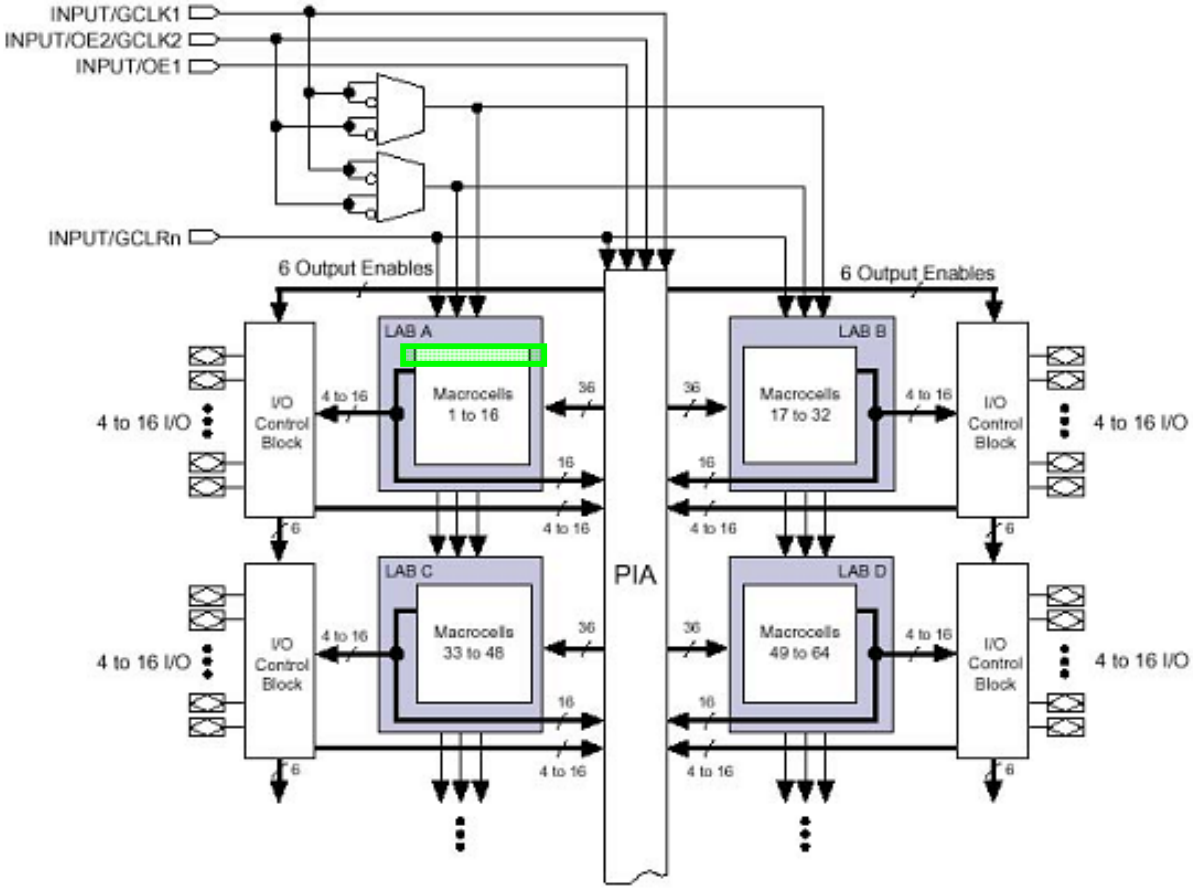
principbild



4×8 = 32
ingångars AND
7 ingångars OR

Typiskt **8** st logikelement
Teknik: **AND-OR** array

CPLD (tex. MAX)

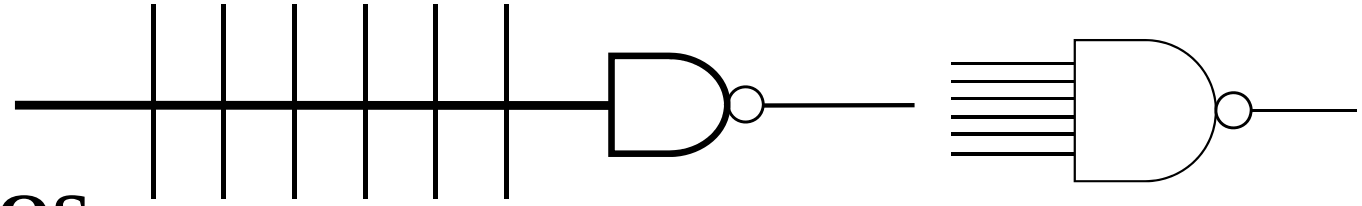


Typiskt **64** Macroceller

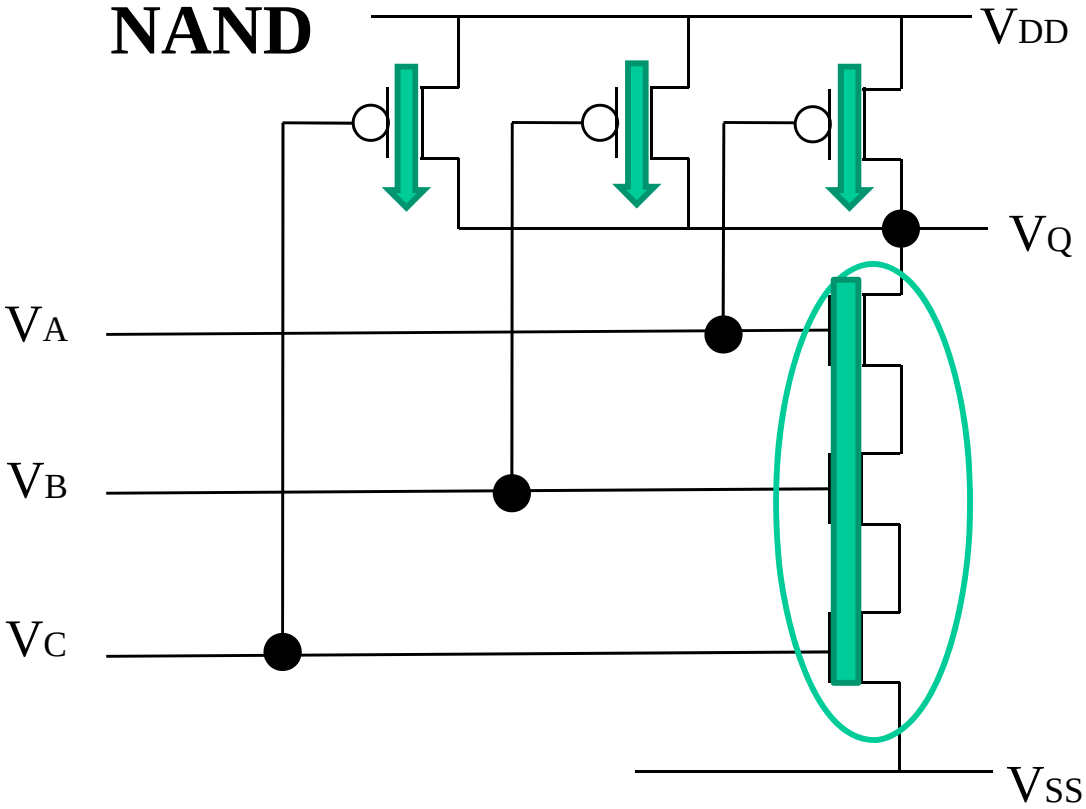
Teknik: **AND-OR** array

(större MAX bytte senare till MUX-tree teknik)

Grindar med många ingångar?



CMOS NAND



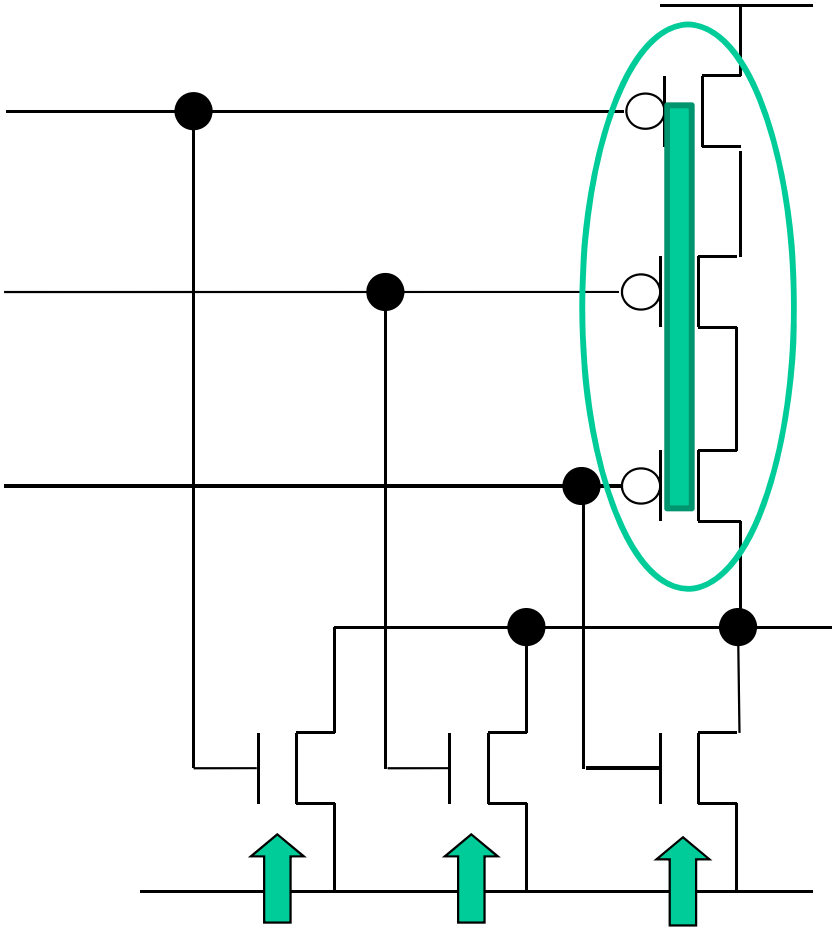
PAL-matrisen har grindar med så många ingångar att man måste rita dem med ett ”förenklat ritsätt”

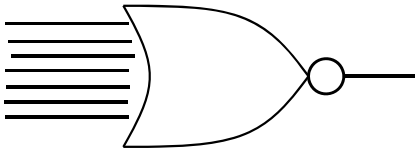
Alla måste leda för ”0” för liten spänning över varje transistor

= **långsamt** när det är många ingångar

Lika illa med CMOS NOR

CMOS NOR





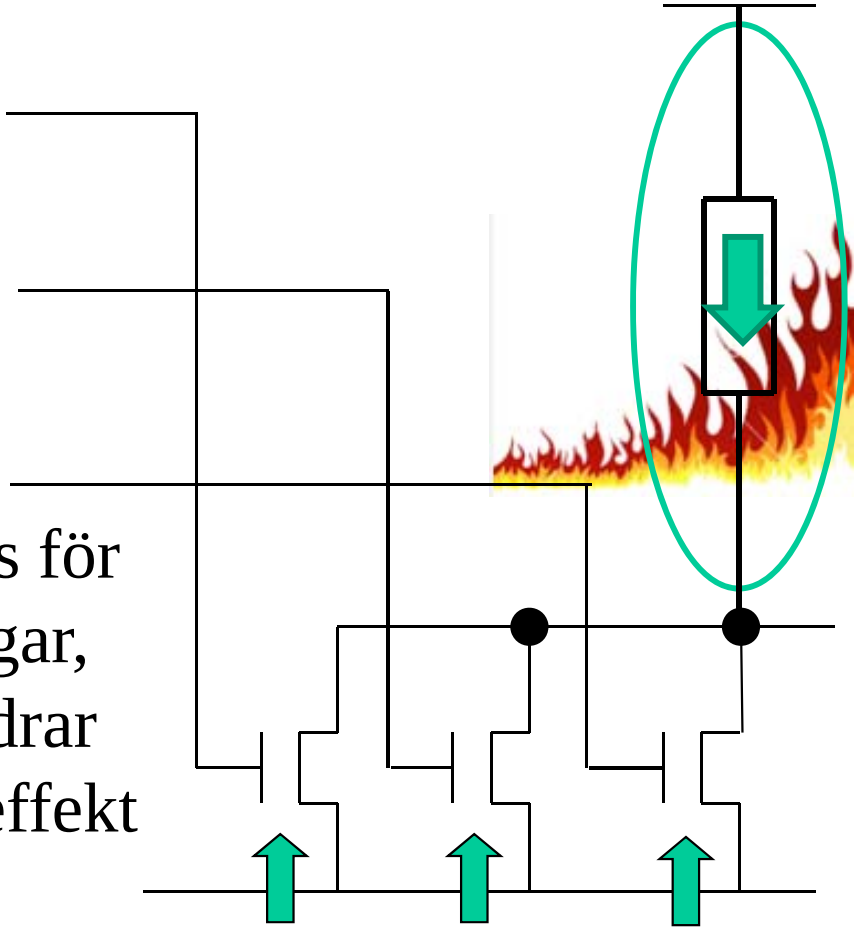
Alla måste leda för ”1” = för liten spänning över varje transistor **långsamt** när det är många ingångar

(En måste leda för ”0” = snabbt)

Snabbt men hög effektförlust

NMOS NOR

Kan användas för många ingångar, men NMOS drar mycket mer effekt än CMOS!



Inte CMOS

”Pull-Up” resistor ger ”1” = **snabbt** men ”**Power Hungry**” vid ”0”

Bara ***en*** måste leda för ”0” = **snabbt**

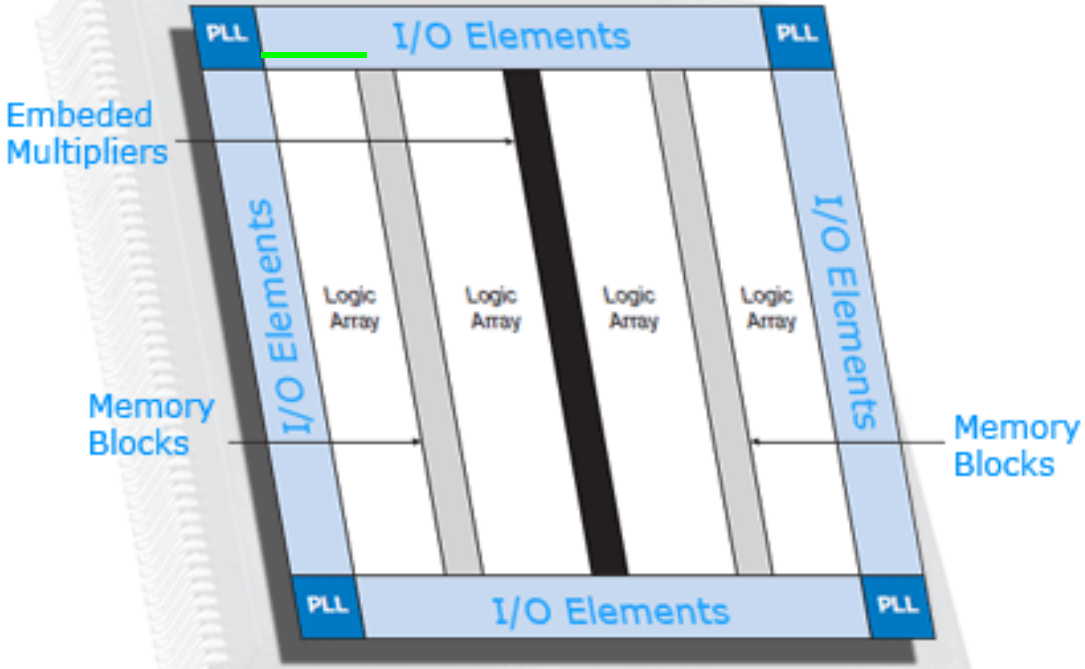
Detta användes till PAL-kretsarna, det blir en kompromiss mellan – **höga effektförluster** eller **långsamma kretsar**.

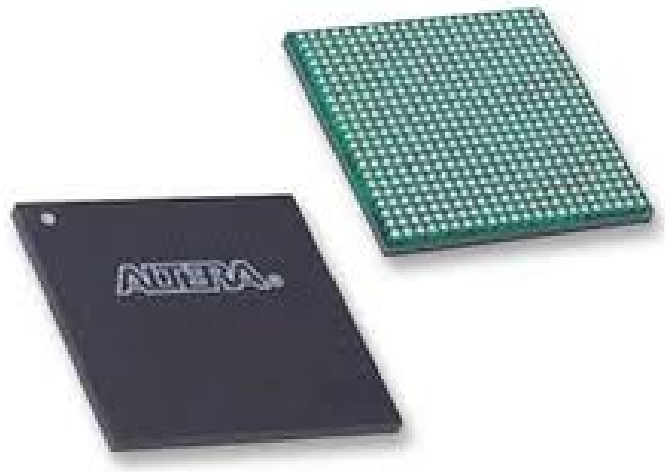
Stora programmerbara kretsar

Det behövs således någon annan teknik som inte bygger på grindar med* måna ingångar, *för att man ska kunna bygga stora programmerbara kretsar i CMOS-teknik!

FPGA (tex. Cyclone II)

Field Programable Gate Array





Stor programmerbar logikkrets

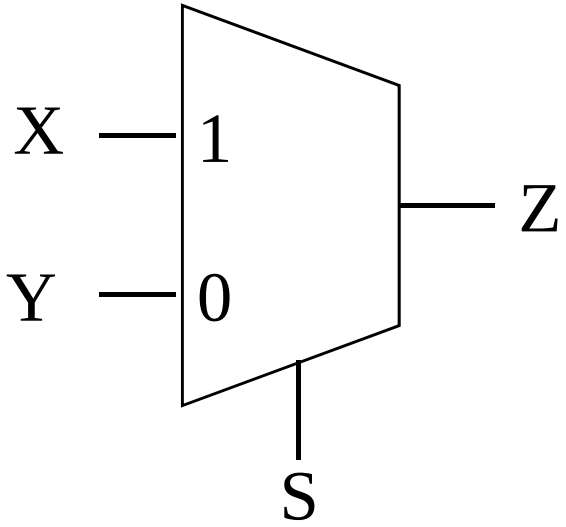
Typiskt **50000** logikelement

Teknik: **MUX** tree

• Multiplexorn MUX

Med multiplexorn kan man välja vilken ingång man ska koppla till utgången.

MUX är numera ”standardkomponenten” vid framtagandet av Digital logik.



$$Z = SX + \overline{S}Y$$

• *Kom Du på att det var en MUX som var den ”**hemliga**” kretsen vid LAB1?*

Logiska funktioner med **MUX**

Hur kan följande funktioner implementeras med en 2:1 Multiplexor?

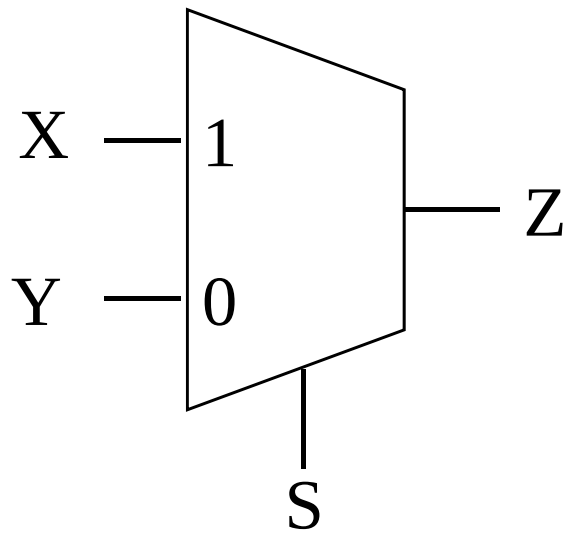
$$Z = \overline{x}_0 \quad NOT$$

$$Z = x_1 \cdot x_0 \quad AND$$

$$Z = x_1 + x_0 \quad OR$$

$$Z = x_1 \oplus x_0 \quad XOR$$

?

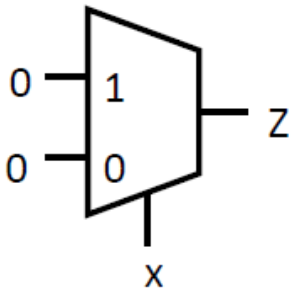


$$Z = SX + \overline{S}Y$$

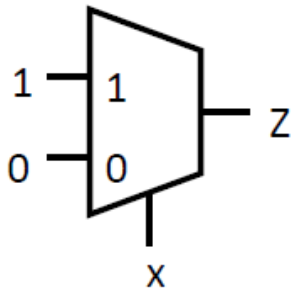
Snabbfråga

*Hur skall vi koppla ingångarna för att implementera en **inverterare** med en MUX?*

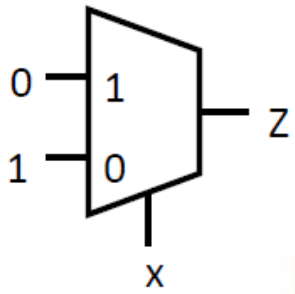
Önskad funktion: $z = \overline{x}$



Alt: A



Alt: B



Alt: C

Invertering **NOT** med **MUX**

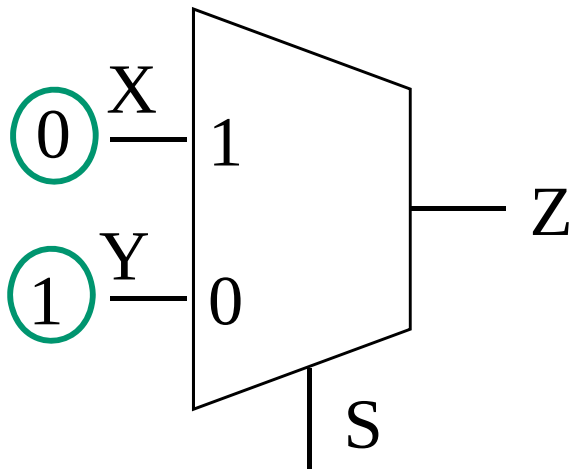
Specification:

```
if input = '1' then result <= '0'
if input = '0' then result <= '1';
```

x_0	0	1
	1	0

$$Z = S \cdot X + \overline{S} \cdot Y =$$

$$= x_0 \cdot \boxed{0} + \overline{x}_0 \cdot \boxed{1} = \overline{x}_0 \quad NOT$$

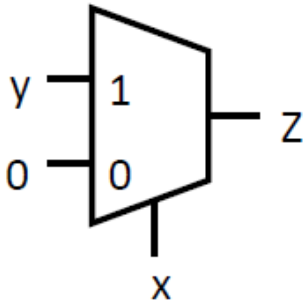


Input (x_0)

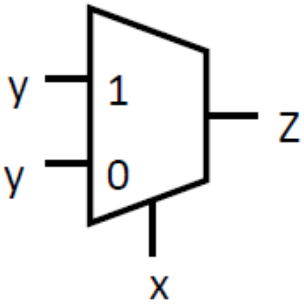
Snabbfråga

*Hur skall vi koppla ingångarna för att implementera en **AND grind** med en MUX?*

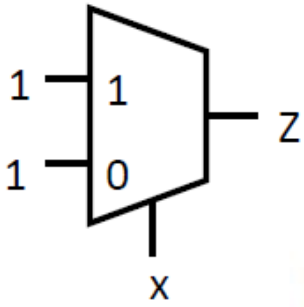
Önskad funktion: z = xy



Alt: A



Alt: B



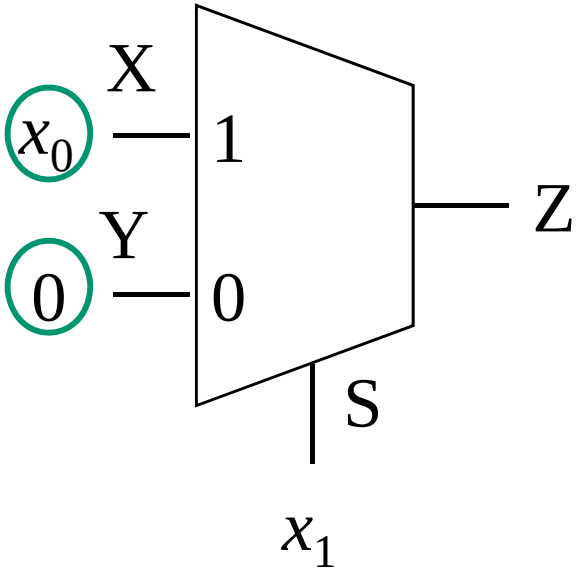
Alt: C

AND-funktion med **MUX**

Specification:

$x_1 \backslash x_0$	0	1
0	0	0
1	0	1

$$Z = x_1 \cdot x_0 \quad AND$$



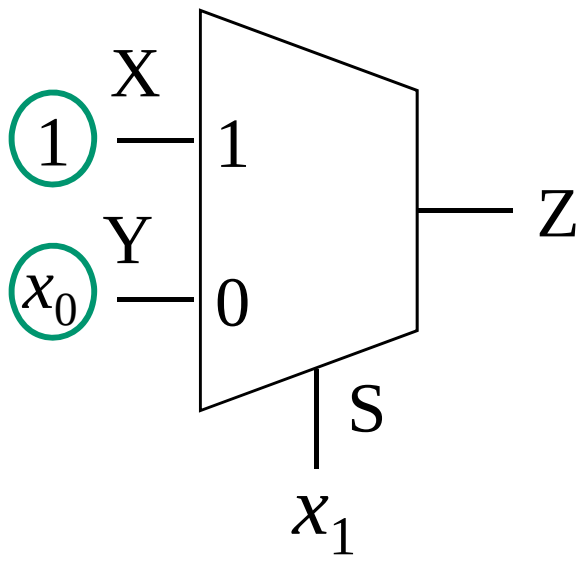
$$Z = SX + \overline{S}Y = x_1 \cdot \boxed{x_0} + \overline{x}_1 \cdot \boxed{0} = x_1 \cdot x_0$$

OR-funktion med **MUX**

Specification:

$x_1 \backslash x_0$	0	1
0	0	1
1	1	1

$$Z = x_1 + x_0 \quad OR$$



$$Z = x_1 x_0 + \bar{x}_1 x_0 + x_1 \bar{x}_0 =$$

$$= \{SX + \bar{S}Y\} = x_1(x_0 + \bar{x}_0) + \bar{x}_1 \cdot x_0 =$$

$$= x_1 \cdot \boxed{1} + \bar{x}_1 \cdot \boxed{x_0}$$

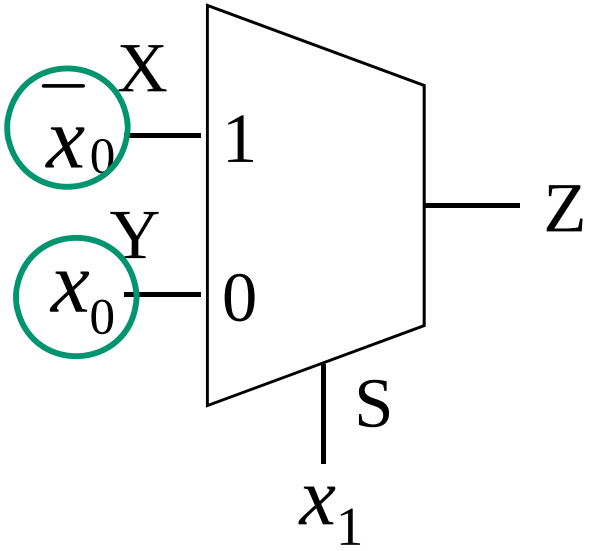
XOR-funktion med **MUX**

Specification:

$x_1 \backslash x_0$	0	1
0	0	1
1	1	0

$$Z = x_1 \oplus x_0$$

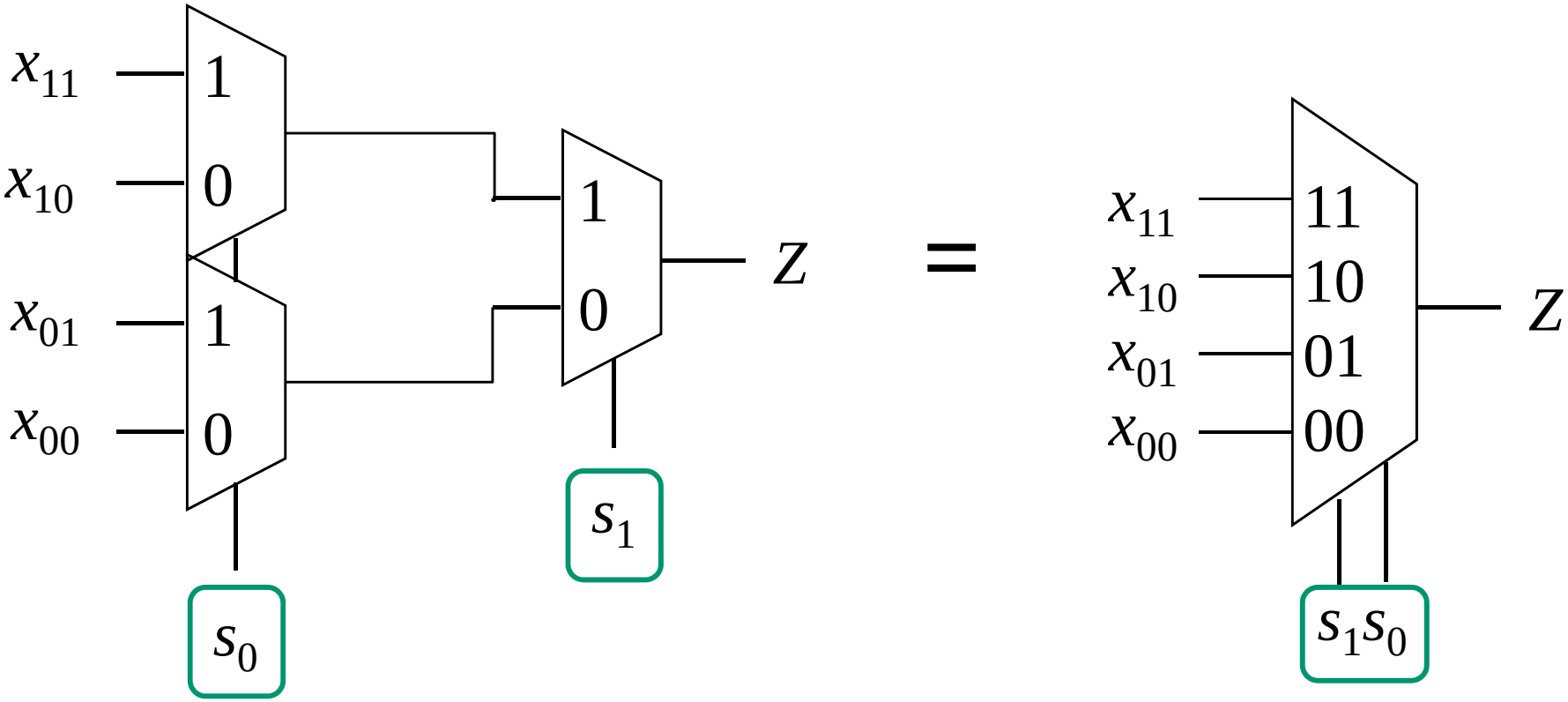
XOR



$$Z = SX + \overline{S}Y =$$

$$= x_1 \cdot \boxed{\overline{x}_0} + \overline{x}_1 \cdot \boxed{x_0} = x_1 \oplus x_0$$

Hierarkier av Muxar

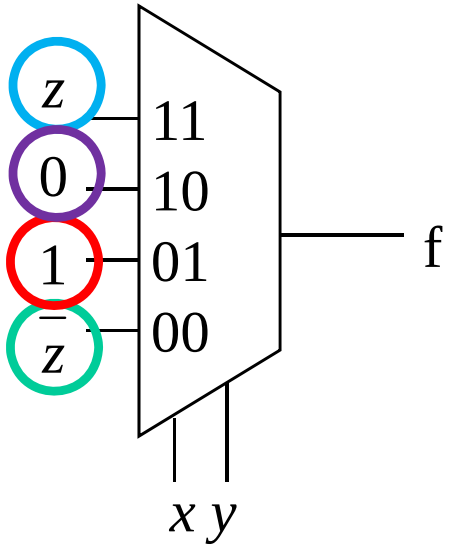


Större funktioner med MUXar

Välj några av ingångarna som address-ingångar ...

z \ $x\,y$	00	01	11	10
0	1	1	0	0
1	0	1	1	0

$$f = \bar{z}\,\bar{x} + \bar{x}y + zy$$



...och minimera/implementera funktionen som uppstår
för varje ingång. Rita nya Karnaugh-diagram om det behövs.

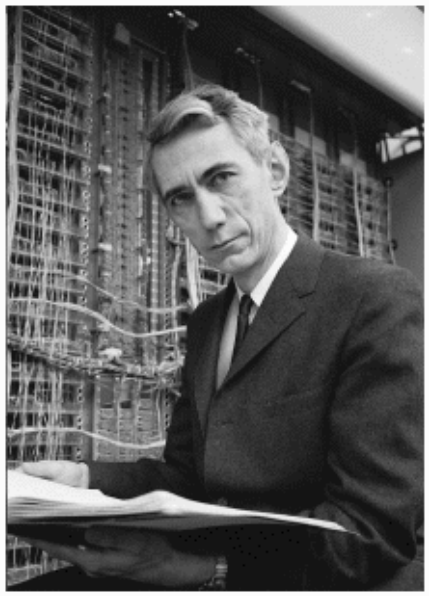
En (n+1)-input funktion kan alltid implementeras med en mux som har n select-ingångar!

Varning! Detta saknas i ”Hemert”

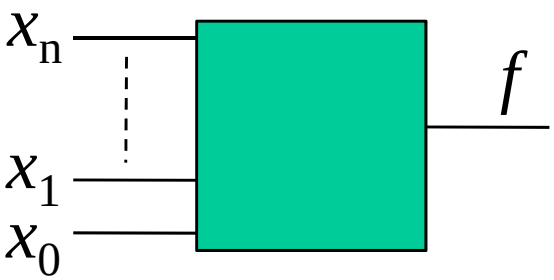
Det finns inget avsnitt om Shannon dekomposition i den svenska boken **Digitala Kretsar**.

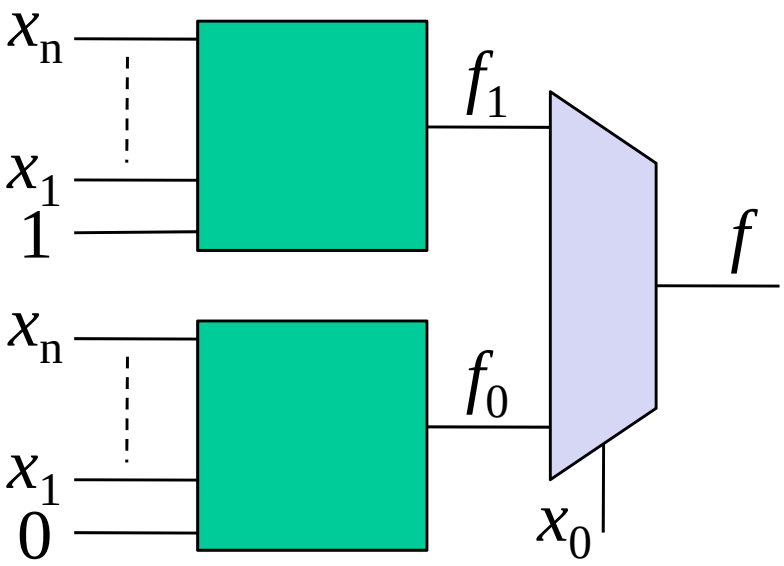
Läs föreläsningsmaterial och övningsmaterial om Du inte använder boken **Digital Logic**.

Shannon dekomposition



Claude Shannon
matematiker/elektrotekniker
(1916 –2001)



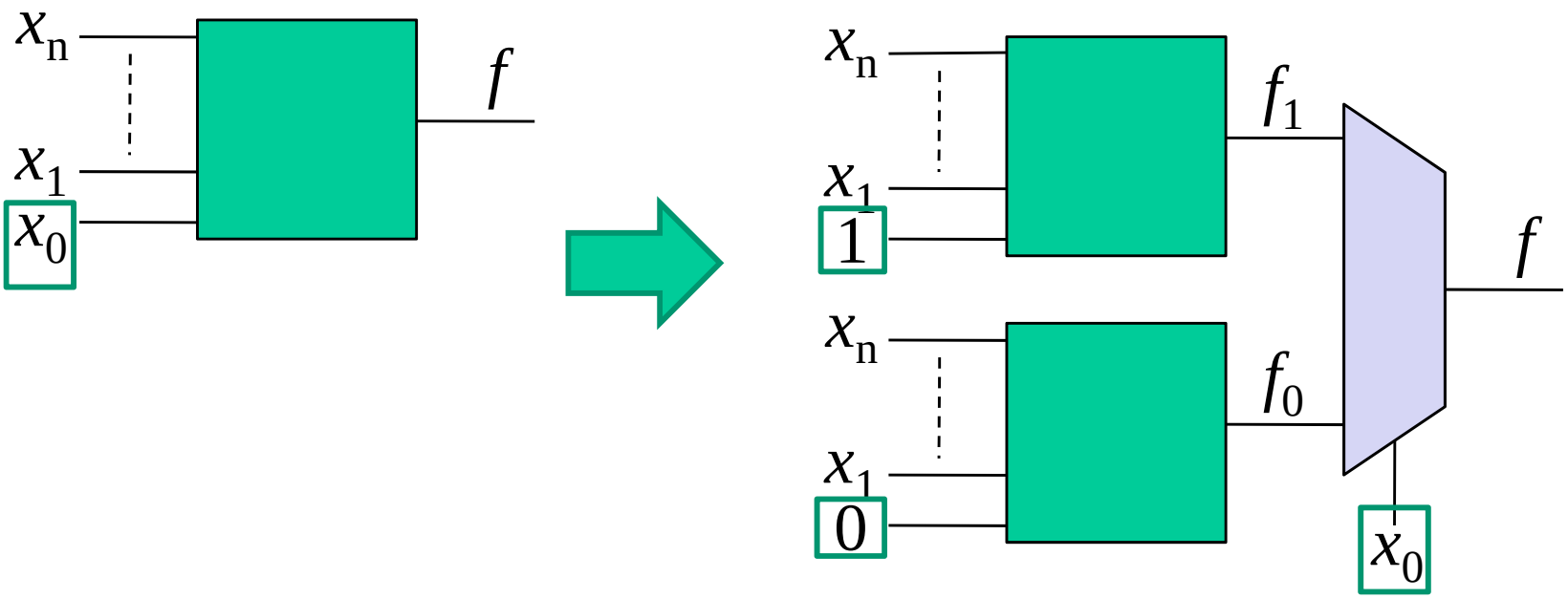


Shannon dekomposition

En boolesk funktion $f(x_n, \ldots, x_1, x_0)$ kan delas upp enligt

$$f(x_n,\ldots,x_1,\boxed{x_0}) = \boxed{x_0}\cdot f(x_n,\ldots,x_1,\boxed{1}) + \boxed{\overline{x_0}}\cdot f(x_n,\ldots,x_1,\boxed{0})$$

Funktionen kan sedan implementeras med en multiplexer.

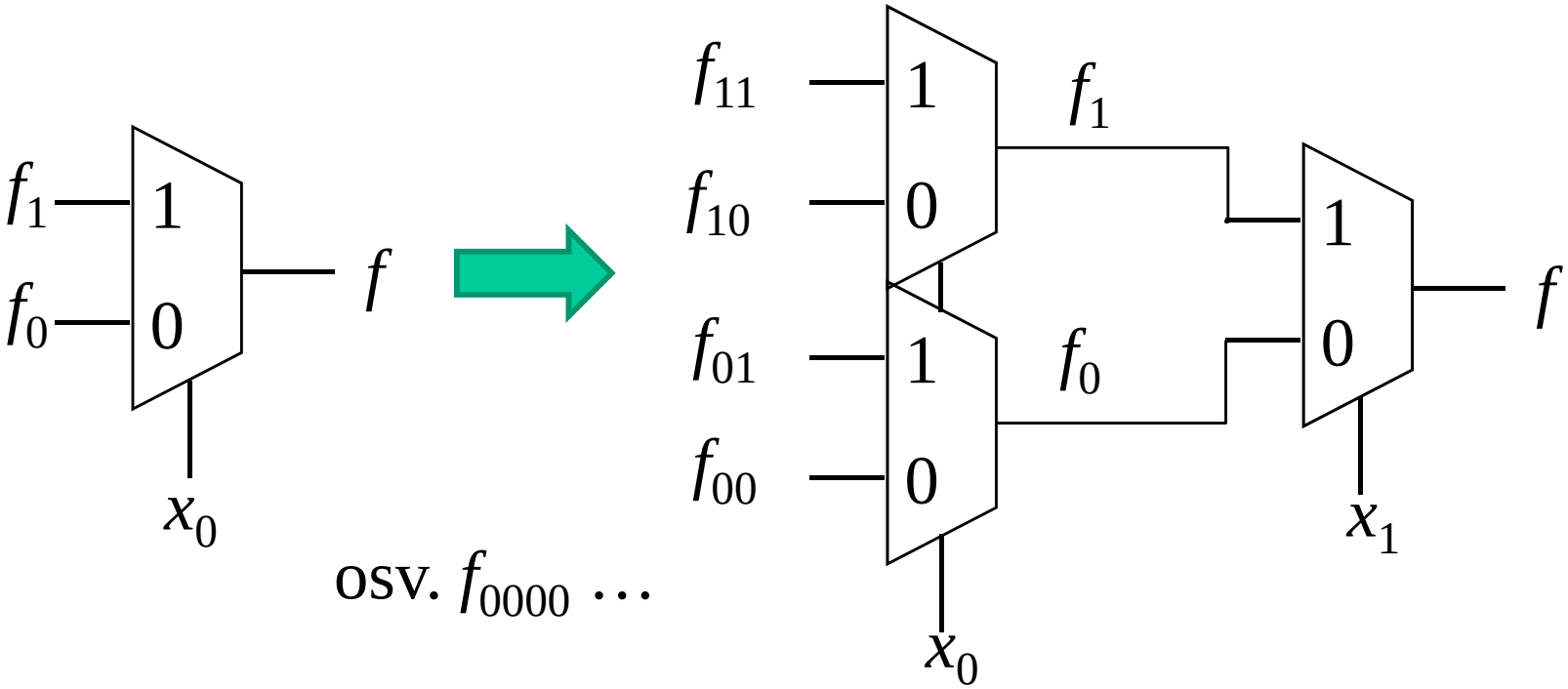


Rekursivt

Alla booleska funktioner $f(x_n, \ldots, x_1, x_0)$ kan delas upp (rekursivt) enligt

$$f(x_n,\ldots,x_1,x_0) = x_0 \cdot f_1(x_n,\ldots,x_1,\boxed{1}) + \overline{x}_0 \cdot f_0(x_n,\ldots,x_1,\boxed{0})$$

$$f_1(x_n,\ldots,x_1) = x_1 \cdot f_{11}(x_n,\ldots,x_2,\boxed{1}) + \overline{x}_1 \cdot f_{10}(x_n,\ldots,x_2,\boxed{0})$$

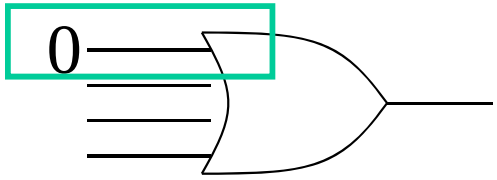


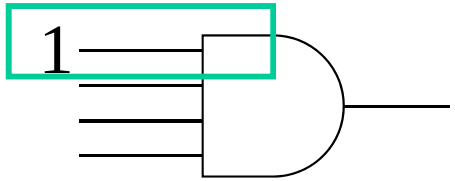
osv. f_{0000} …

0 och 1 till logiknät?

$$f(x_n,\ldots,x_1,x_0) = x_0 \cdot f(x_n,\ldots,x_1,\boxed{1}) + \overline{x}_0 \cdot f(x_n,\ldots,x_1,\boxed{0})$$

Att införa 1 eller 0 i ett logiknät är ett ”trix” för att göra beviset enkelt. Det kan man *alltid* göra i ett logiknät utan att det innebär någon egentlig inskränkning av detta (utan att det märks)!





Om shannon-dekompositionen tillämpas rekursivt på alla variablerna behöver man till sist bara **1** och **0**, *inga logiknät* (utöver multiplexorer) behövs!

Bevis

$$f(x_n,\ldots,x_1,x_0) = x_0 \cdot f(x_n,\ldots,x_1,1) + \overline{x}_0 \cdot f(x_n,\ldots,x_1,0)$$

Höger sida (eng. RHS):

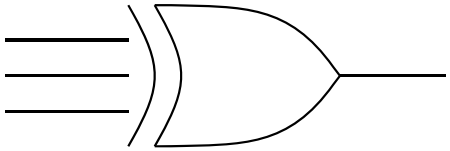
om x_0=1 så blir den högra termen noll. Då blir f lika med den vänstra termen.
om x_0=0 så blir den vänstra termen noll. Då blir f lika med den högra termen.

Vänster sida (eng. LHS):

om x_0=1 så blir f lika med $f(x_n,\ldots,x_1,1)$ (= vänstra termen på högra sidan)
om x_0=0 så blir f lika med $f(x_n,\ldots,x_1,0)$ (= högra termen på högra sidan)

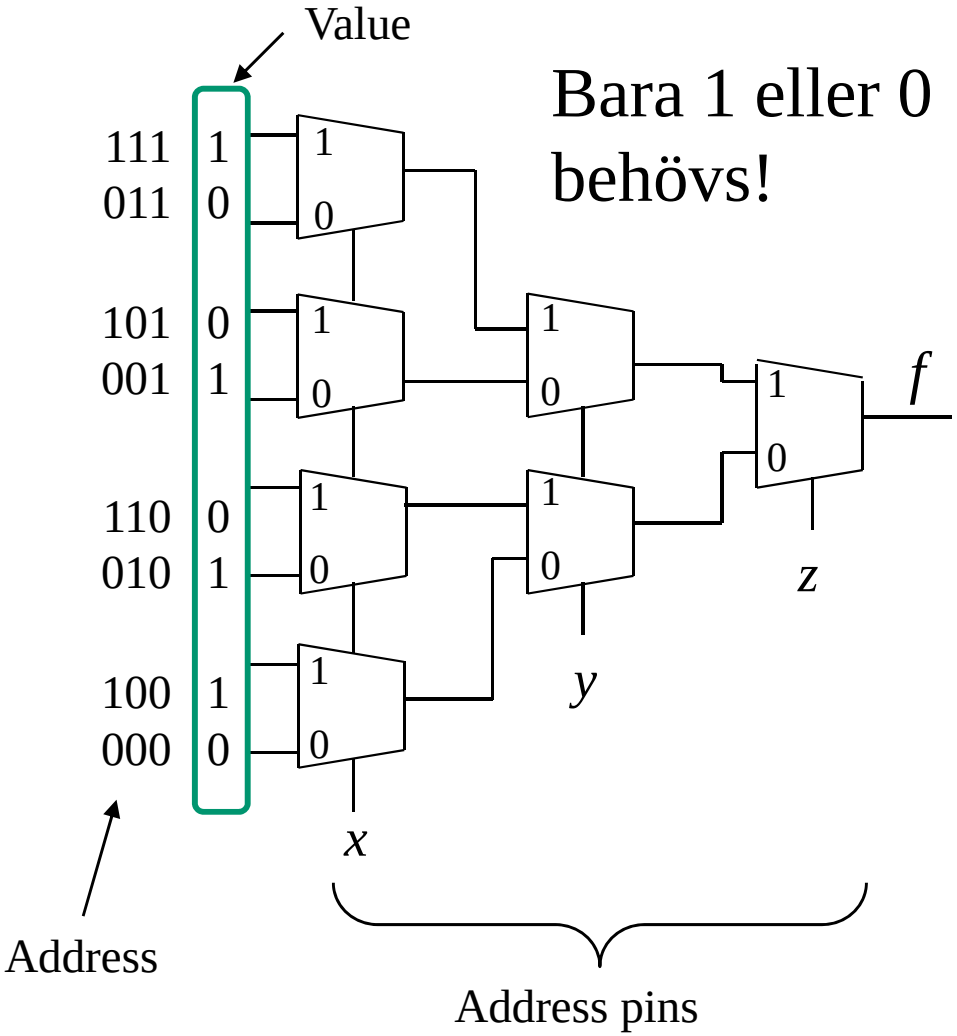
LHS=RHS

MUX networks (tree)

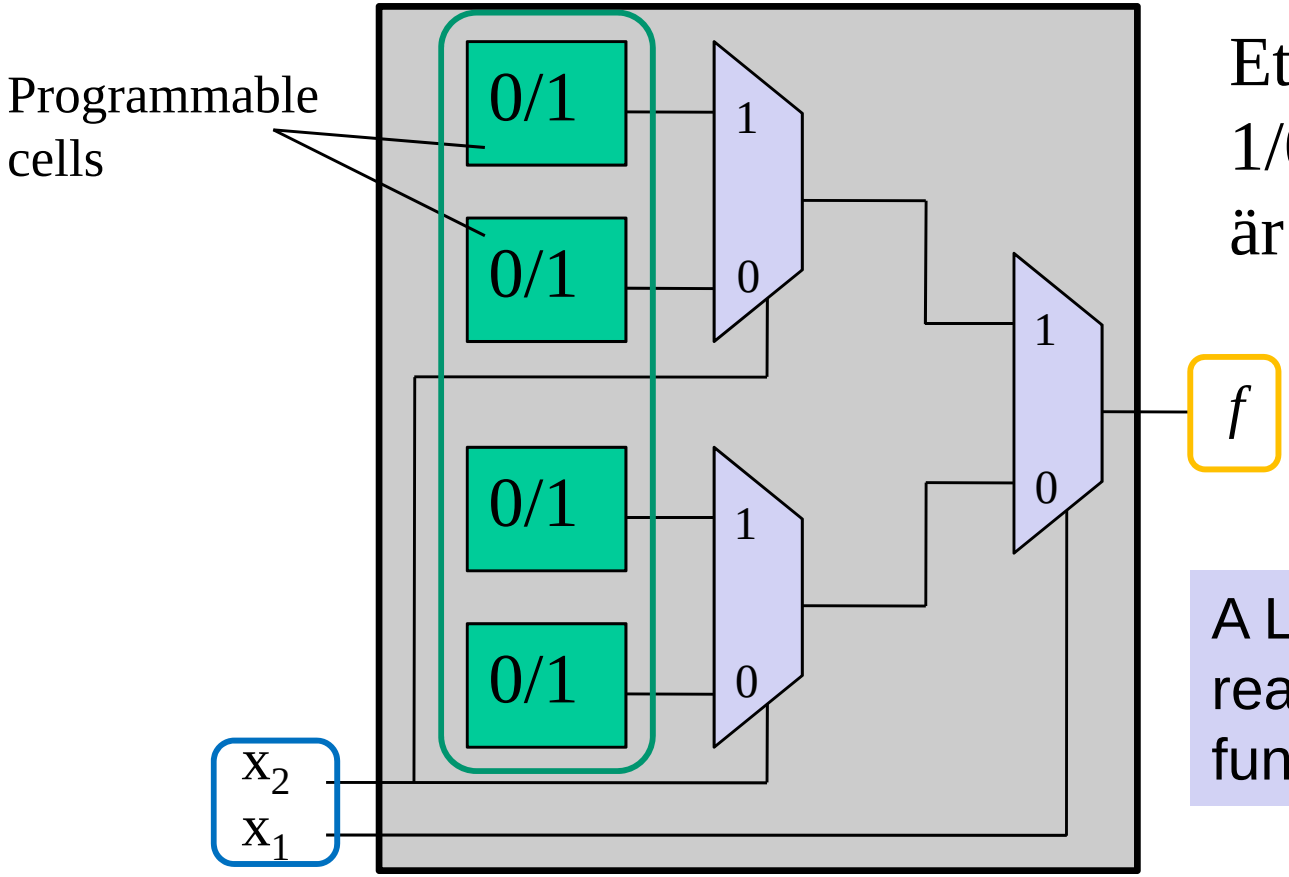


x \ yz	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Man kan se *xyz* som en adress, till rutorna i Karnaughdiagrammet. Med 1/0 från rutorna till muxens ingångar ”realiserar” man funktionen *f*.



Look-up-table (LUT)

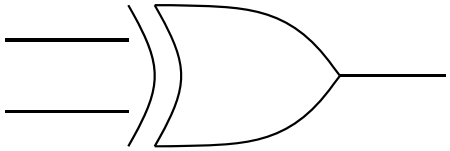


Two-input LUT

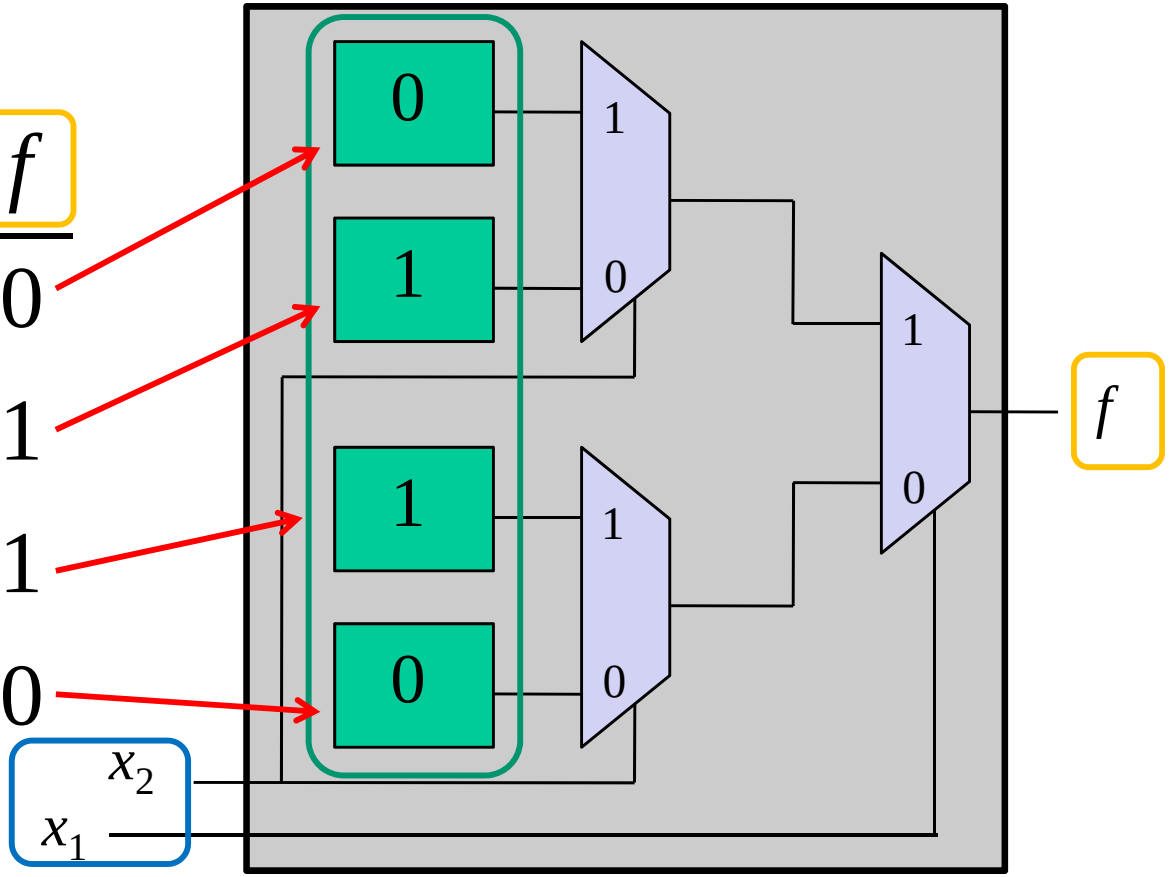
Ett MUX-tree med 1/0 på ingångarna är en LUT.

A LUT with *n* inputs can realize all combinational functions with *n* inputs

LUT för XOR-grind

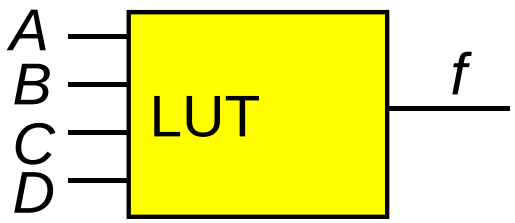


x_1	x_2	f
1	1	0
1	0	1
0	1	1
0	0	0

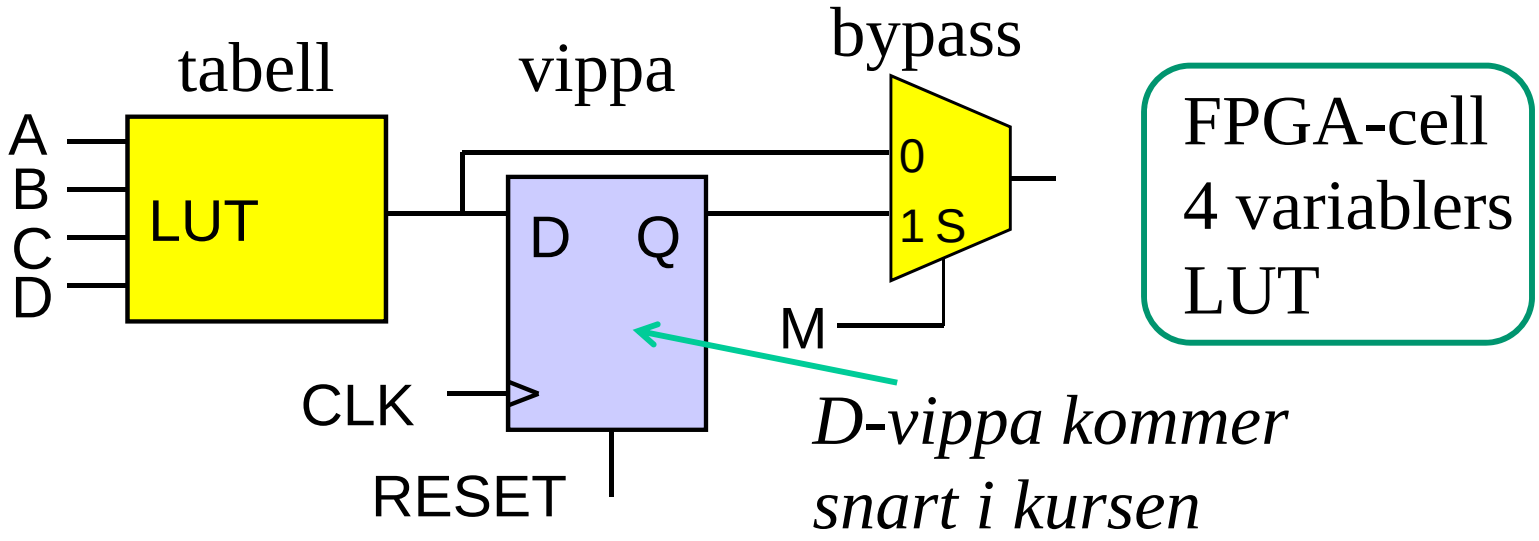


Two-input LUT

En enkel FPGA-cell



Den enklaste FPGA-cellen är uppbyggd av en enda tabell (eng. Look-Up-Table - LUT), en D-vippa och en bypass-Mux. D-vippan är en minneskrets för synkronisering – kommer senare i kursen. Med bypass-muxen kopplar man sig förbi D-vippan för de kretsar som inte behöver en sådan.



LUT funktionernas nummer

$$f(x_3, x_2, x_1, x_0) = \text{"}0110100110010110\text{"} = f_{6996H}$$

Bit # 15 Bit # 1 Bit # 0

LSB

$x_1 x_0$

x_3 x_2

MSB

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

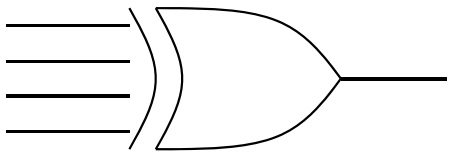
Känner Du igen funktionen … ?

Funktionerna som lagras i en LUT brukar numreras efter det tal som byggs upp av 1:orna i sanningstabellen /Karnaugh-diagrammet.

LUT funktionernas nummer

$$f(x_3, x_2, x_1, x_0) = \text{"0110100110010110"} = f_{6996H}$$

$x_3 x_2$ \ $x_1 x_0$	00	01	11	10
00	0: 0	1: 1	3: 0	2: 1
01	4: 1	5: 0	7: 1	6: 0
11	12: 0	13: 1	15: 0	14: 1
10	8: 1	9: 0	11: 1	10: 0



$$f_{6996H} = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

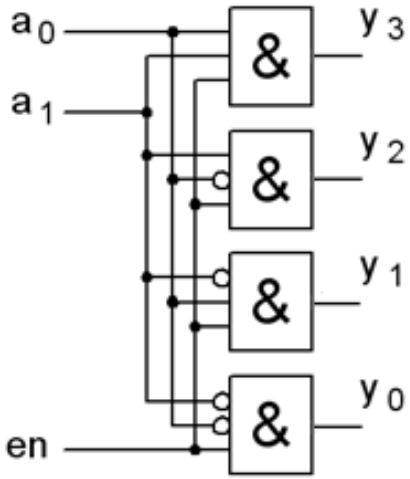
Nu vet Du vilken funktion som är **6996_{16}**!

Udda paritet!
Inga hoptagningar.

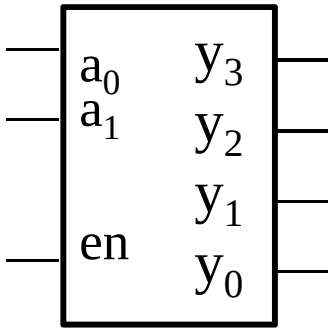
Med en LUT kan alla funktioner realiseras, därför är ingen av dem svårare att göra än någon annan!

Decoder (Avkodare)

Används mest som address-avkodare
Bara en utgång är aktiv när ’enable’ (en) är aktiv
Den aktiva utgången väljs med a_1a_0

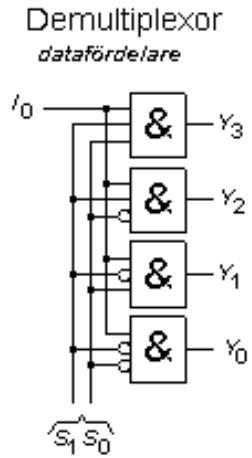


en	a_1	a_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	-	-	0	0	0	0



2-till-4 avkodare

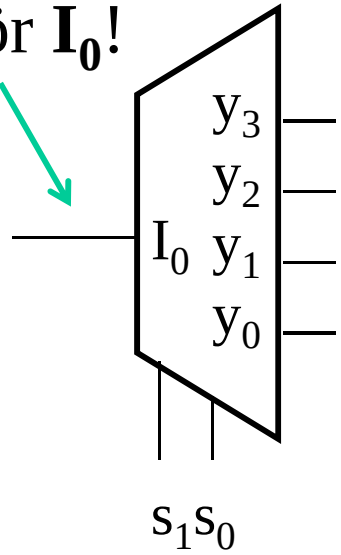
Demultiplexor (DMUX)



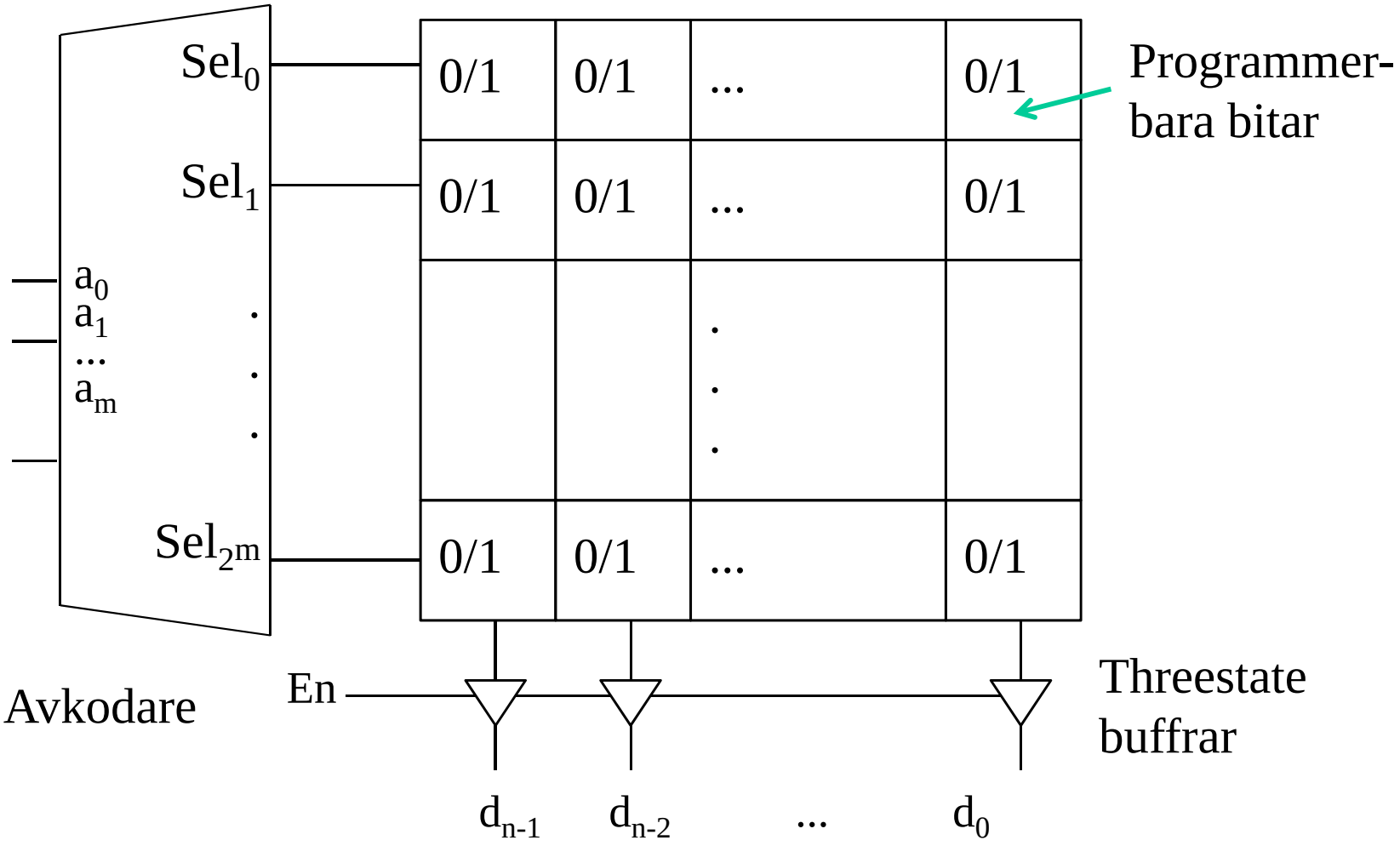
Demultiplexern har egentligen **samma** funktion som decodern, men ritas annorlunda ...
Ingången kopplas till en vald utgång

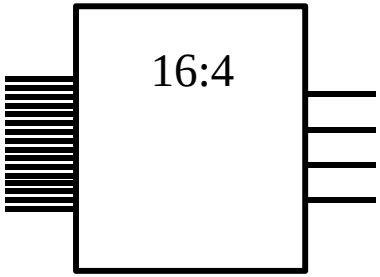
I_0	s_1	s_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	-	-	0	0	0	0

Nu kallas **en** för $\mathbf{I_0}$!



Read-only-memory (ROM)





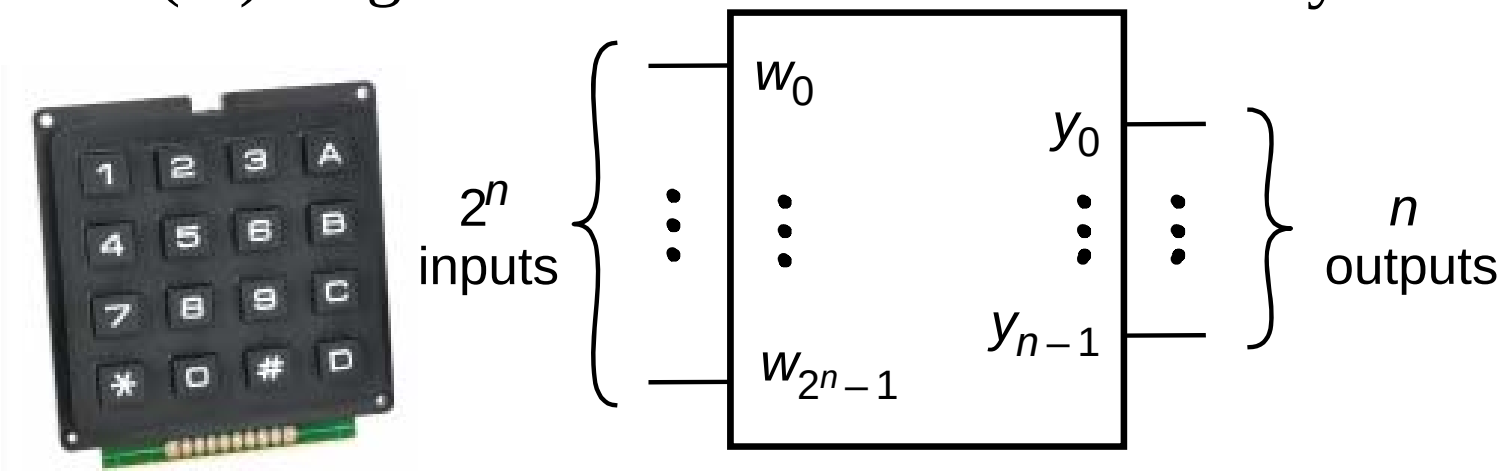
Encoder

Encoders har motsatt funktion som en decoder, dvs den översätter 2^N bitars input till en N-bitars kod.

- Informationen **koncentreras kraftigt**

Ex. Tangentbord med 16 (2^4) tangenter

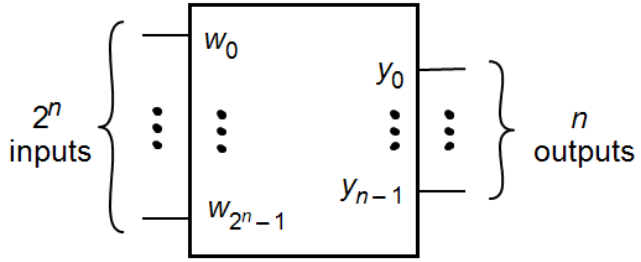
4-bitars HEX-kod för nedtryckt tangent



Prioritetsenkoder

En Priority Encoder ger tillbaka addressen på ingången med den lägsta (eller högsta) indexen som är satt till en etta (eller nolla beroende på vad man söker efter).
Om alla ingångar är 0 blir utgången $f = 0$, annars har f värdet = 1.

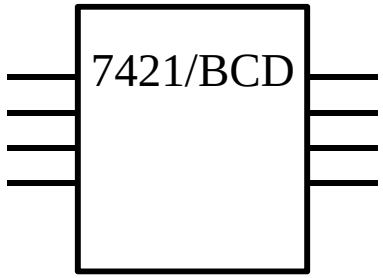
Tänk om man trycker på flera tangenter samtigt?



y_0	y_1	y_2	y_3	f	a_1	a_0
1	-	-	-	1	0	0
0	1	-	-	1	0	1
0	0	1	-	1	1	0
0	0	0	1	1	1	1
0	0	0	0	0	-	-

Nu blir det väldefinierat vad som ska hända om flera ingångar är aktiva.

ÖH 8.4 7-4-2-1 kod



Kodomvandlare 7-4-2-1-kod till BCD-kod.

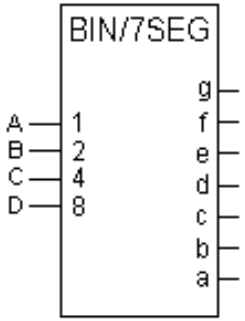
Vid kodning av siffrorna 0…9 användes förr ibland en kod med vikterna 7-4-2-1 i stället för den binära kodens vikter 8-4-2-1.

(en ***variant*** av 7-4-2-1 koden används i dag till butikernas streck-kod)

I de fall då en siffras kodord kan väljas på olika sätt väljs det kodord som innehåller minst antal ettor.

En sådan kodomvandlare konstruerar vi på övning 3

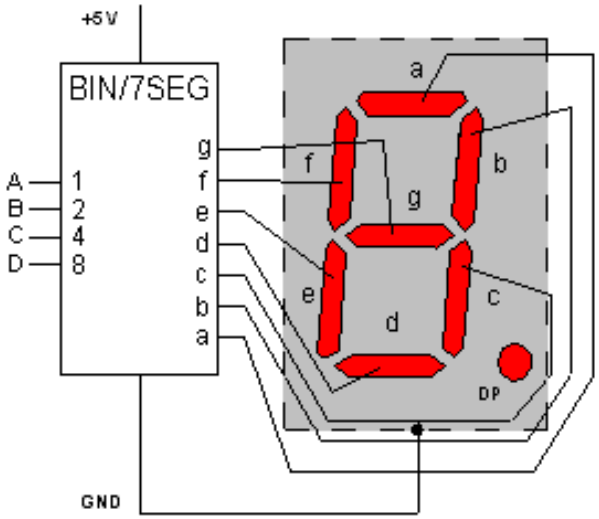
Kod-konverterare

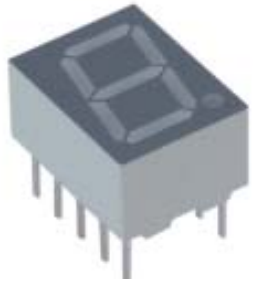


Kod-konverterare översätter från en kod till en annan.

Typiska exempel är

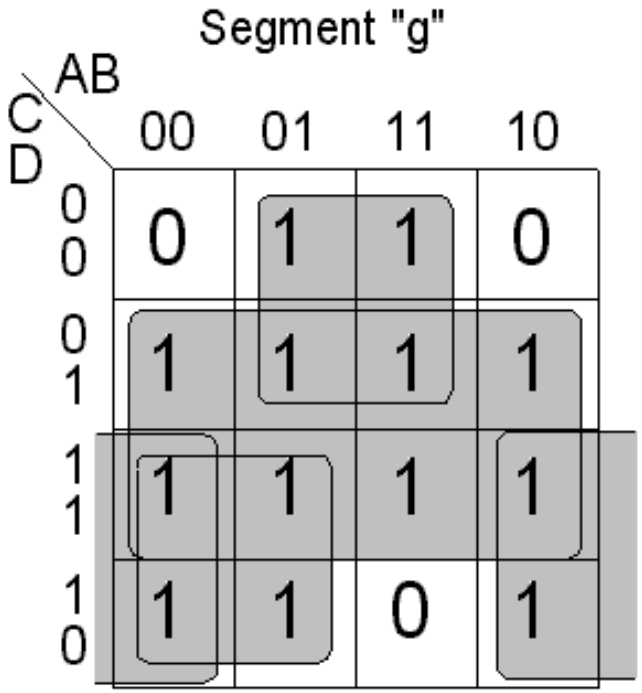
- Binär till BCD (Binary-Coded Decimal)
- Binär till Gray-kod
- BCD eller BIN till sju-segmentsavkodning





ÖH 8.5 Ett av segmenten ”g”

D C B A	a b c d e f	g
0 0 0 0	1 1 1 1 1 1	0
0 0 0 1	0 1 1 0 0 0	0
0 0 1 0	1 1 0 1 1 0	1
0 0 1 1	1 1 1 1 0 0	1
0 1 0 0	0 1 1 0 0 1	1
0 1 0 1	1 0 1 1 0 1	1
0 1 1 0	1 0 1 1 1 1	1
0 1 1 1	1 1 1 0 0 0	0
1 0 0 0	1 1 1 1 1 1	1
1 0 0 1	1 1 1 0 0 1	1
1 0 1 0	1 1 1 0 1 1	1
1 0 1 1	0 0 1 1 1 1	1
1 1 0 0	0 0 0 1 1 0	1
1 1 0 1	0 1 1 1 1 0	1
1 1 1 0	1 0 0 1 1 1	1
1 1 1 1	1 0 0 0 1 1	1



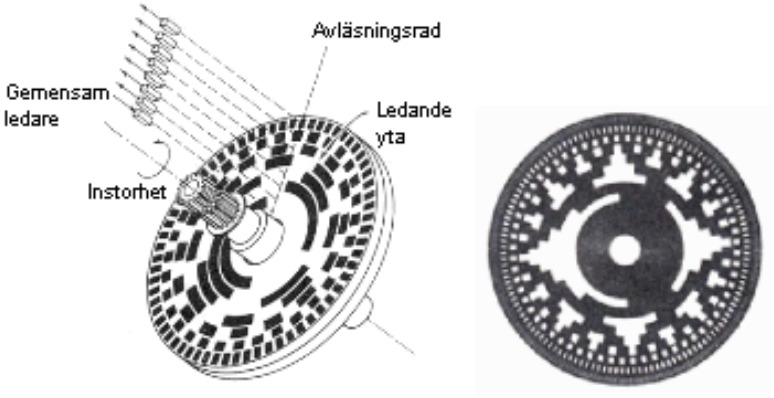
$$g = D + B\overline{C} + \overline{B}C + \overline{A}C$$

7-segmentavkodaren består av 7 olika kombinatoriska nät, ett för varje segment.

Man bör titta på Karnaughdiagrammen för ***alla*** segmenten samtidigt. Det kan ju finnas hoptagningar som är gemensamma för flera segment!

Den optimala 7-segmentavkodaren är nog redan uppfunnen!

Graykod eller Binärkod ?



Vinkelmätare med kodskiva. Till vänster binärkod, till höger Graykod.

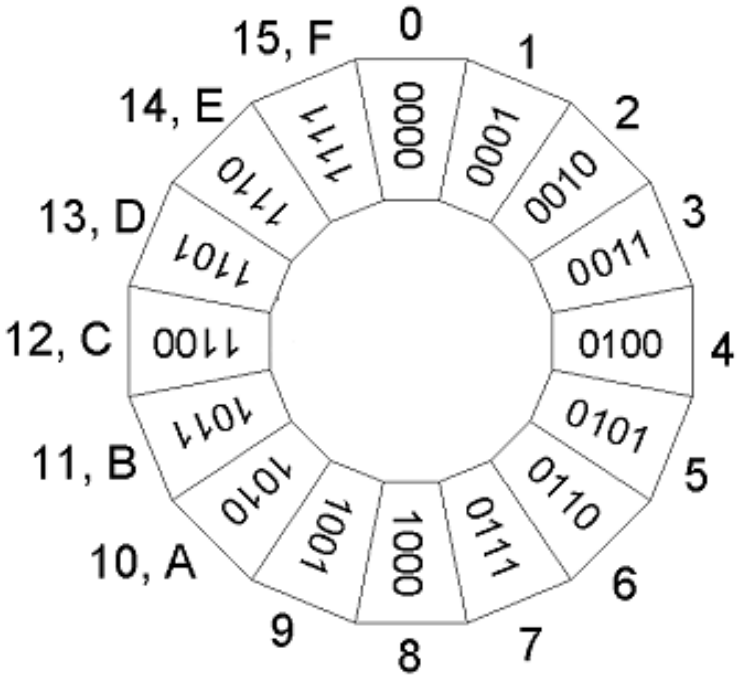
	Binär-kod	Gray-kod		Binär-kod	Gray-kod
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Tabell med Binärkod och Graykod.

Vindriktning *Vindhastighet*

Vindriktningsvisare brukar använda Gray-kod för att ge ***säker*** visning.

Binärkodens nackdel



Binärkod, angränsande koder
1-2 dubbeländring
3-4 trippeländring
5-6 dubbeländring
7-8 quadruppeländring!
9-A dubbeländring
B-C trippeländring!
D-E dubbeländring
F-0 quadruppeländring!

Men kan verkligen två bitar ändra sig ***exakt samtidigt****?*

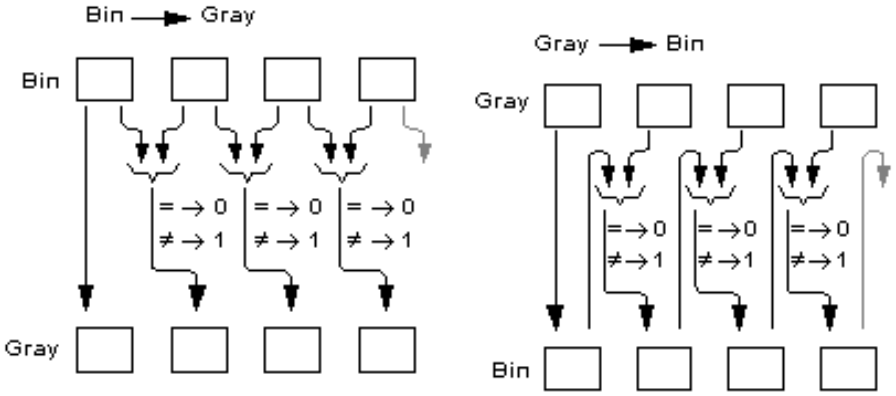
- Säker datafångst **Graykod**
- Databearbetning **Binärkod**

Graykod

Genom att ändra om den inbördes ordningen mellan kodorden kan man hitta koder där det aldrig är mer än ***en*** bit i taget som ändras vid övergångarna från ett kodord till nästa. Sådan koder kallas för Graykoder.

0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100
1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

Omvandling Binärkod-Graykod

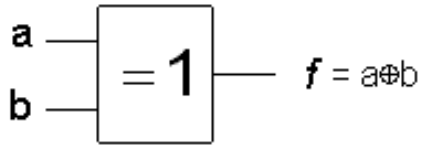


Binär → Gray:
Om Binärkodens bit b_n och bit b_{n-1} är ***olika***, är Graykodens bit g_{n-1} ”1", annars ”0".

Gray → Binär (den vanligaste omvandlingsriktningen):
Om Binärkodens bit b_n och Graykodens bit g_{n-1} är ***olika*** blir Binärkodens bit b_{n-1} ”1", annars ”0".

Logikkrets för kodomvandlingen

XOR-grindens utgång ger
”1” om ingångarna är *olika*!

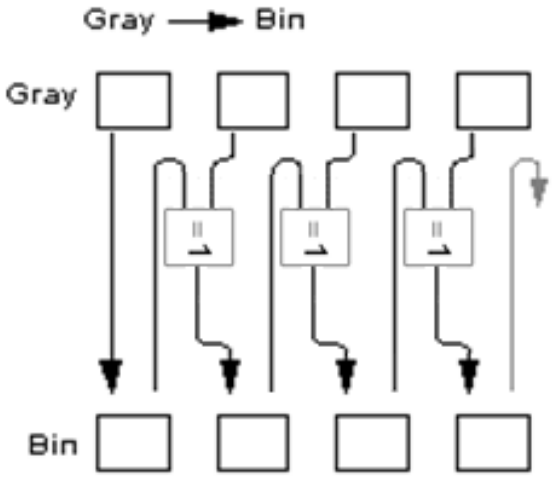


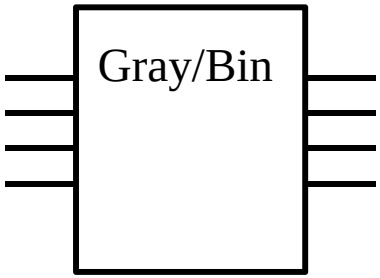
a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

	Binär-kod	Gray-kod		Binär-kod	Gray-kod
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Tabell med Binärkod och Graykod.

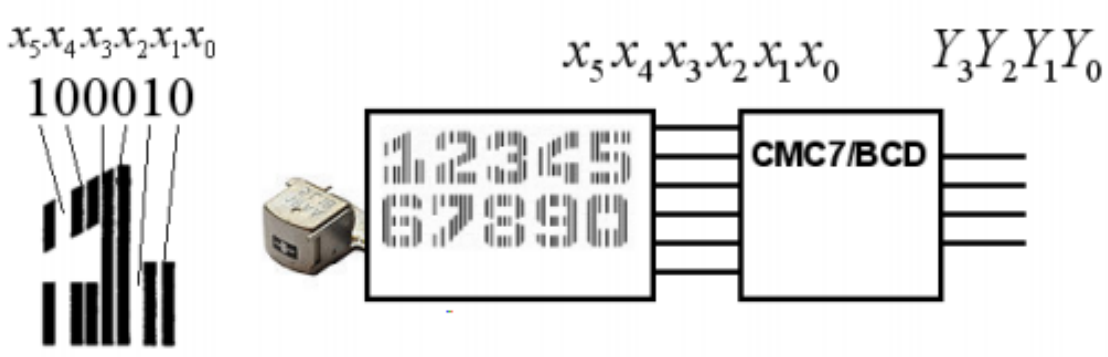
4 bits kodomvandlare
Graykod till Binärkod





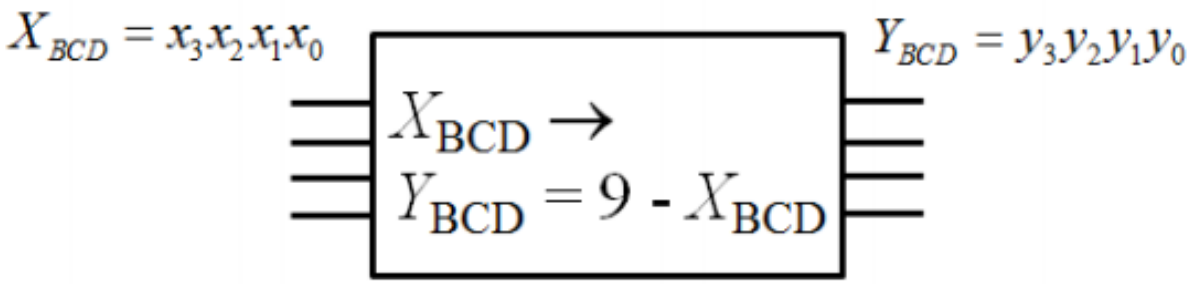
Kodomvandlare på extentor

CMC → BCD



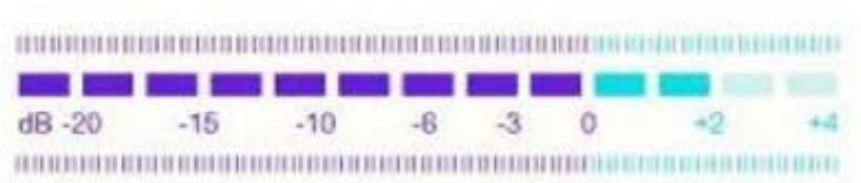
Kodomvandlare på extentor

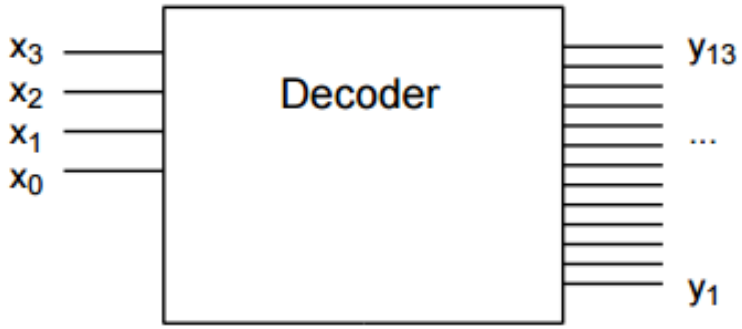
BCD 9- komplementerare



Kodomvandlare på extentor

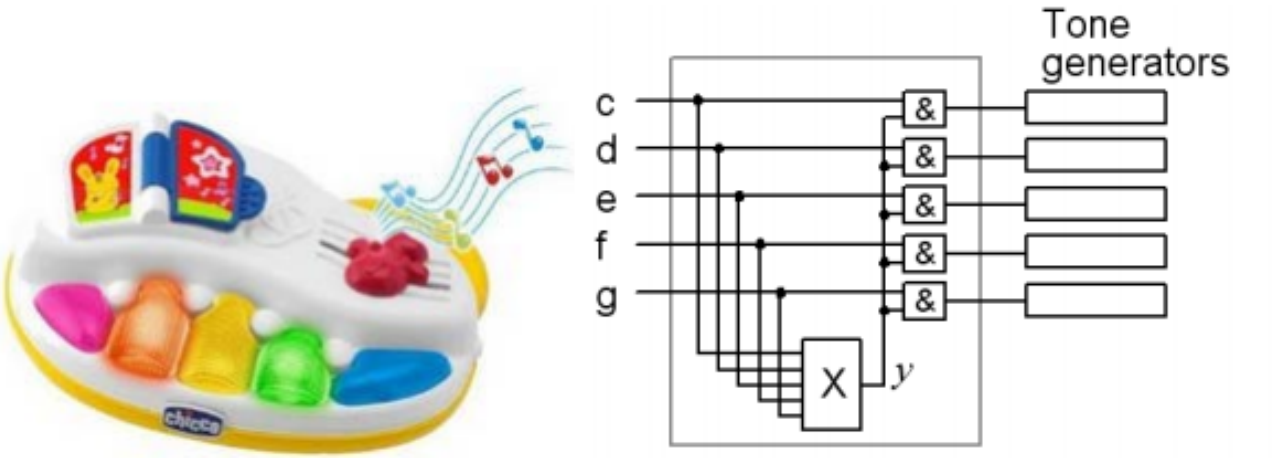
Bar graph decoder





Kodomvandlare på extentor

Dissonans-spärr



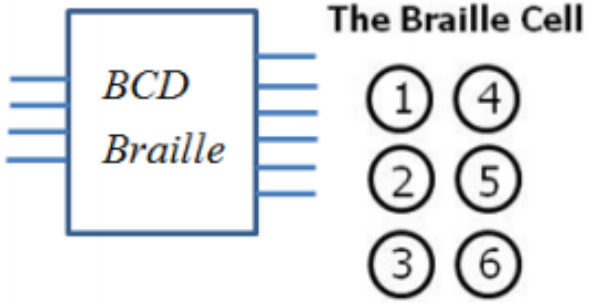
X tillåter endast väljudande ackord från barnen

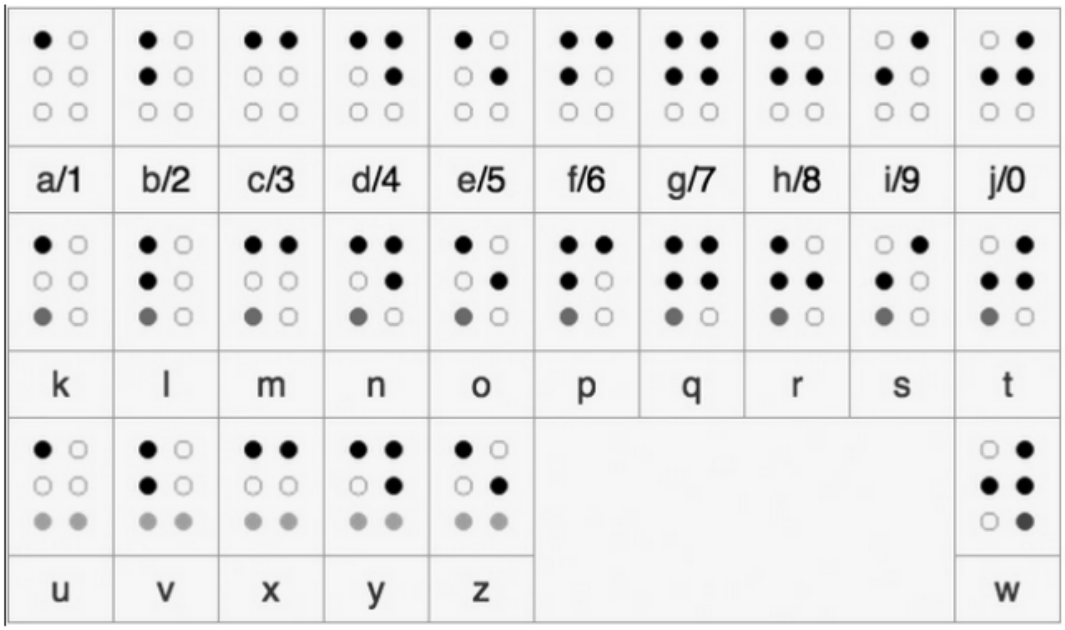
Tack för att Du konstruerar detta nät – många barnföräldrar blir tacksamma!

Kodomvandlare på extentor

$BCD \rightarrow Braille$

$x_3x_2x_1x_0$ $y_6y_5y_4y_3y_2y_1$

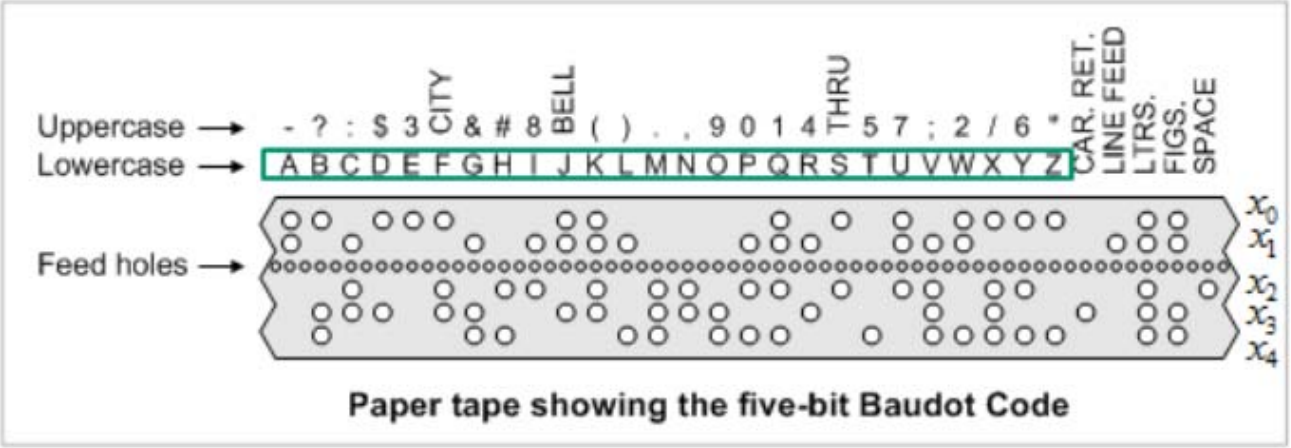




Blindskrift – indikator för siffrorna 0…9

Avkodare på extentor

Vokal - indikator



Tyvärr visste inte studenterna att vokalerna var
A O U E I Y !

VHDL-introduktion

VHDL är ett språk som används för att specificera hårdvara

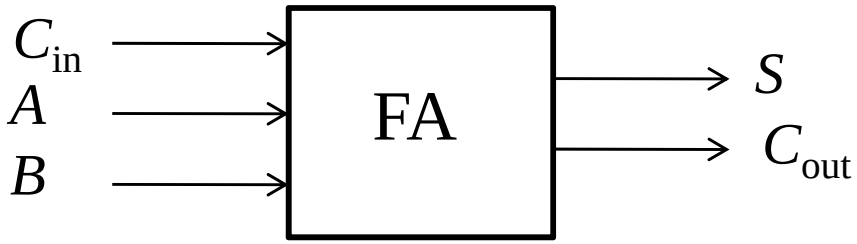
- HDL - Hardware Description Language
- VHSIC - Very High Speed Integrated Circuit

Används mest i Europa

Verilog är också ett språk som används för att specificera hårdvara

- Används mest i USA

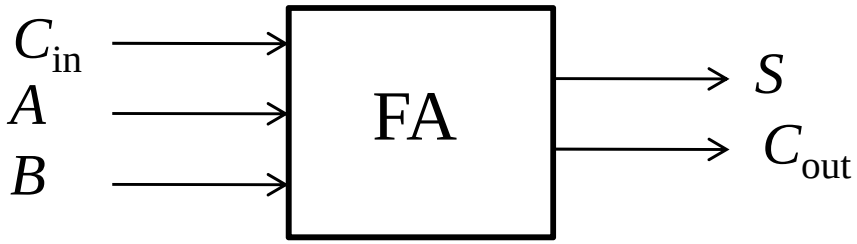
Entity




```
entity fulladder is
     port( A,B,Cin :  IN std_logic;
           S,Cout  : OUT std_logic);
end fulladder;
```

Entiteten beskriver portarna mot omvärlden för kretsen.
Kretsen som ett block.

Architecture




```
architecture behave of fulladder is
begin
   S <= A xor B xor Cin;
   Cout <= (A and B) or (A and Cin) or (B and Cin);
end behave;
```

Architecture beskriver funktionen inuti kretsen.

Varför VHDL?

VHDL används för att

- kunna kontrollera att man har tänkt rätt genom att ***simulera*** kretsen
- kunna beskriva stora konstruktioner på ett enkelt sätt och sedan generera kretsen genom syntes
- möjliggöra strukturerade beskrivningar av en krets

VHDL ökar abstraktionsnivån!

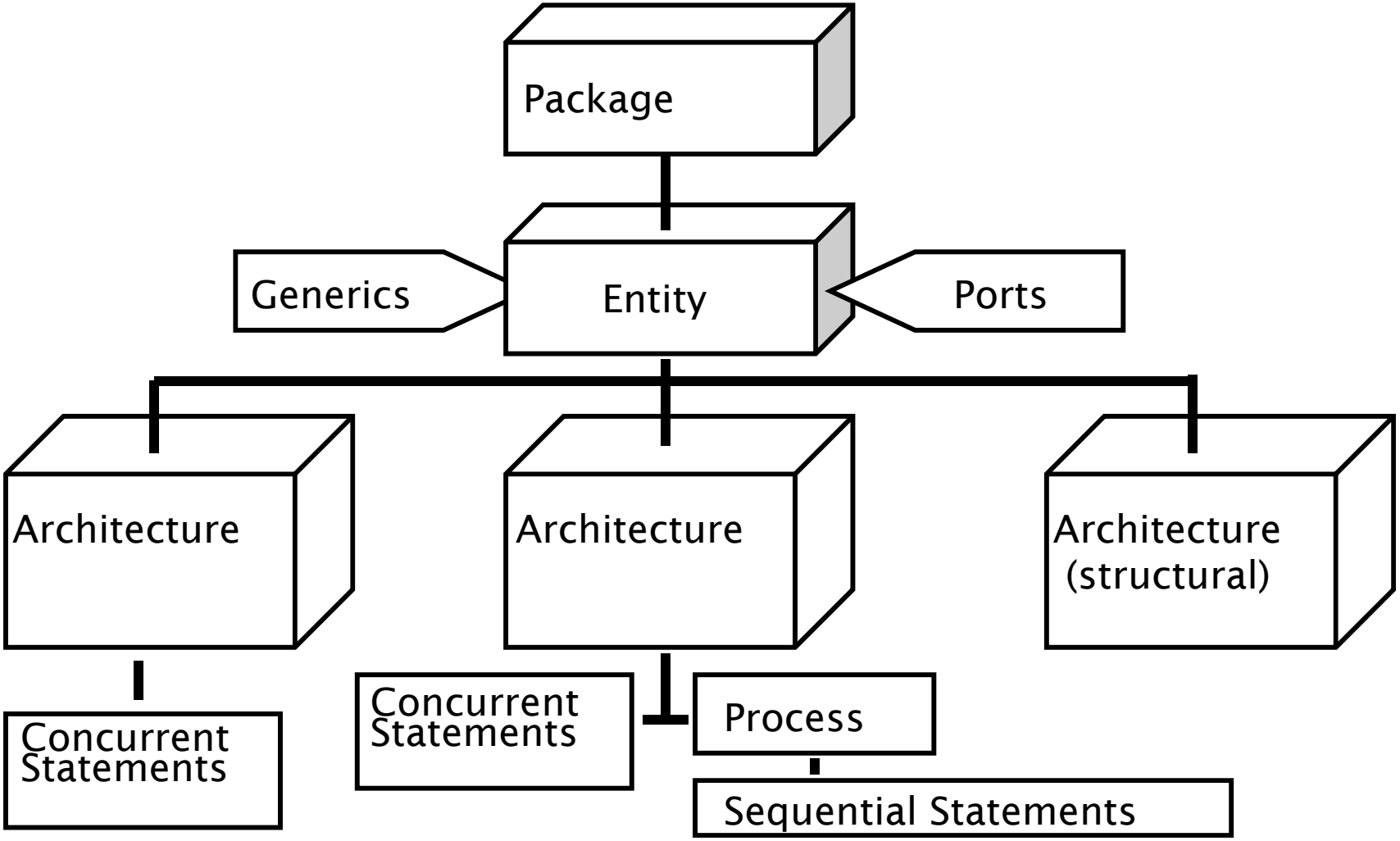
Grunder i VHDL

Det finns två typer av **VHDL-**kod

- VHDL för **syntes**: Koden ska vara input till ett syntesverktyg som omvandlar den till en implementering (t ex på en FPGA)

- VHDL för modellering och **simulering**: Koden används för att beskriva ett system i ett tidigt skede. Eftersom koden kan simuleras så kan man kontrollera om det tilltänkta funktionssättet är korrekt.

VHDL hiearkin



Entitet (eng. Entity)

- Den primära abstraktions-nivån i VHDL kallas för *entity*
- I en beteende-beskrivning definieras entiteten genom sina svar på signaler och ingångar
- En beteende-modell är samma sak som en ”svart låda"
  - Insidan syns inte från utsidan
  - Entitetens beteende definieras av den svarta lådans funktionalitet

Entity forts.

- En entitet beskriver en komponents *interface* med omvärlden
- PORT -deklarationen indikerar om det är en in eller utgång.
- En *entity* är en symbol för en komponent.

```
ENTITY xor_gate IS
    PORT( x, y: IN bit;
          q: OUT bit);
END xor_gate;
```

Använd engelska beteckningar för variabelnamn i koden!

VHDL Port

- PORT-deklarationen etablerar *gränssnittet (interfacet)* mellan komponenten och omvärlden.

- En PORT-deklaration innehåller tre saker:
  - Namnet på porten
  - Riktningen på porten
  - Portens datatyp

- Exempel:

```
ENTITY test IS
    PORT( namn : riktning data_typ);
END test;
```

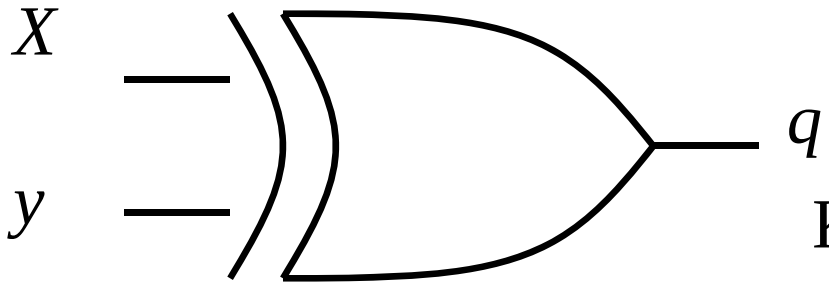
De vanligaste datatyperna

- Skalärer (envärda signaler/variabler)
  - bit (’0’,’1’)
  - std_logic (’U’,’0’,’1’,’X’,’Z’,’L’,’H’,’W’,’-’)
  - integer
  - real
  - time

- Vektorer (flervärda signaler/variabler)
  - bit_vector – vektor av bit
  - std_logic_vector – vektor av std_logic

Architecture

- En *architecture* beskriver komponentens funktion.
- En entitet kan ha många arkitekturer, men endast en kan vara aktiv i taget.
- En arkitektur motsvarar komponentens kopplingsschema eller beteende.

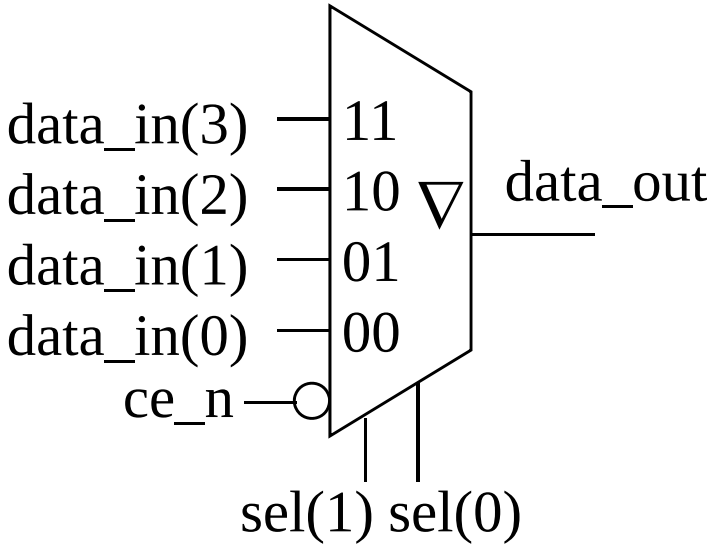


Kod för simulering

Betyder pil <=

```
ARCHITECTURE behavior OF xor_gate IS
BEGIN
    q <= a xor b after 5 ns;
END behavior;
```

VHDL-Exempel: 4/1 multiplexor




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Multiplexer_41 IS
PORT(ce_n     : IN std_logic; -- Chip En(active low)
     data_in  : IN std_logic_vector(3 DOWNTO 0);
     sel      : IN std_logic_vector(1 DOWNTO 0);
     data_out : OUT std_logic); -- TriState Output
END ENTITY Multiplexer_41;
```

VHDL-Exempel: 4/1 multiplexor

```
ARCHITECTURE RTL OF Multiplexer_41 IS
BEGIN
  PROCESS(ce_n, data_in, sel)
  BEGIN
    IF ce_n = '1' THEN
      data_out <= 'Z';
    ELSE
      CASE sel IS
        WHEN "00"=> data_out <= data_in(0);
        WHEN "01"=> data_out <= data_in(1);
        WHEN "10"=> data_out <= data_in(2);
        WHEN "11"=> data_out <= data_in(3);
        WHEN OTHERS => null;
      END CASE;
    END IF;
  END PROCESS;
END ARCHITECTURE RTL;
```

Threestate!

Syntesverktyget Quartus II

QuartusII

Simulator:
ModelSim

Kommer i LAB 3

Mer om VHDL

- Studiematerialet om syntes visar ett antal VHDL-konstruktioner och den resulterande hårdvaran

- Följande bilder innehåller extra material (överkurs)

- Kursboken ger många exempel och mer detaljerade förklaringar om VHDL

Signal deklarationen

Signal-*deklarationen* används inuti arkitekturer för att deklarera interna (lokala) signaler:

```
signal a,b,c,d : bit;
signal a,b,sum : bit_vector(31 downto 0);
```

Signal-*tilldelningen* (eng. *Signal assignment)* används för att beskriva beteendet:

```
sum <= a + b; -- assignment without delay
```

VHDL olika beskrivningsstilar

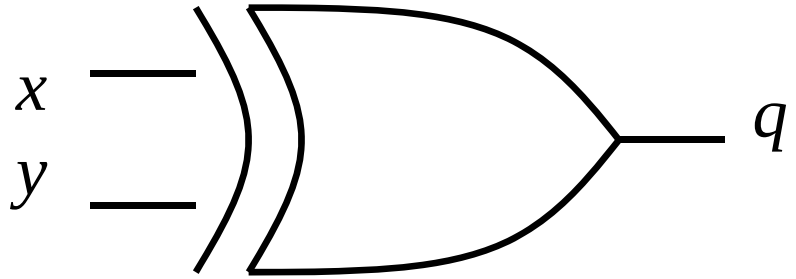
- **Strukturell**
  liknar hur man kopplar ihop komponenter
- **Sekvensiell**
  liknar hur man skriver vanliga datorprogram
- **Dataflöde**
  Parallella tilldelningar (eng. ”Concurrent assignments”)

Sekvensiell eller Parallell kod

- Det finns två typer av exekvering av kod i VHDL: sekventiell och parallell
- Hårdvara kan alltså modelleras på två olika sätt
  - VHDL supportar olika abstraktionsnivåer.
- **Sekvensiell kod** beskriver hårdvaran från en “programmerares” synvinkel och exekveras i den ordning den står i.
- **Parallell kod** exekveras oberoende av ordningen den står i och är *asynkron*.

Sekvensiell stil

XOR-gate

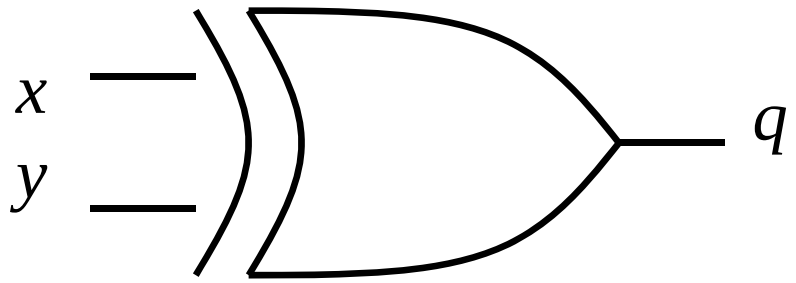



```
process(x,y)
begin
 if (x/=y) then
   q <= '1';
 else
   q <= '0';
 end if;
end process;
```

Betyder not!

Dataflödes stil

XOR-gate



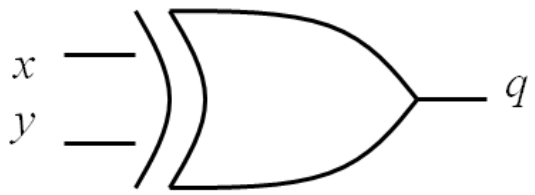

```
q <= a xor b;
```

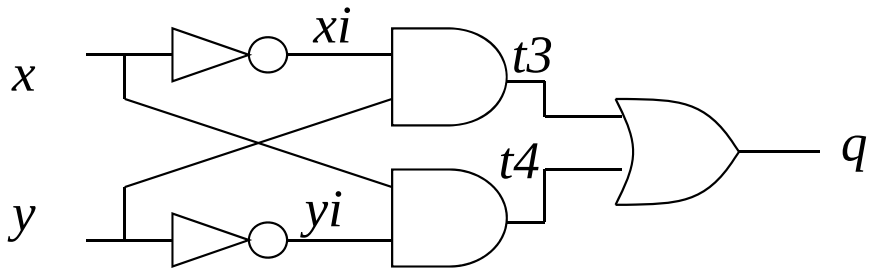
eller i ”**behavioural dataflow** style”

olikhet

```
q <= ‘1’ when a/=b else ‘0’;
```

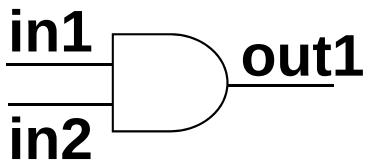
Strukturell stil






```
u1: not_gate port map (x,xi);
u2: not_gate port map (y,yi);
u3: and_gate port map (xi,y,t3);
u4: and_gate port map (yi,x,t4);
u5: or_gate port map (t3,t4,q);
```

Strukturell kod



- En **komponent** måste **deklareras** innan den kan användas

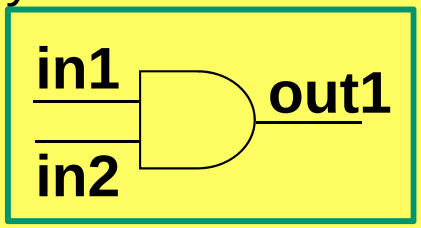
```
ARCHITECTURE test OF test_entity
   COMPONENT and_gate
     PORT ( in1, in2 : IN BIT;
            out1 : OUT BIT);
   END COMPONENT;
... more statements ...
```

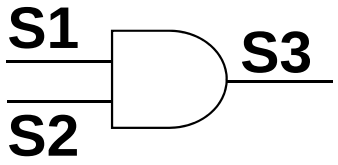
- Nödvändigt, om det inte är så att den redan finns i ett bibliotek någonstans

Instantiering

Komponent instantieringen kopplar ihop komponentens interface med signalerna i arkitekturen.

```
ARCHITECTURE test OF test_entity
    COMPONENT and_gate
      PORT ( in1, in2 : IN BIT;
             out1 : OUT BIT);
    END COMPONENT;
    SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : and_gate PORT MAP (S1,S2,S3);
END test;
```





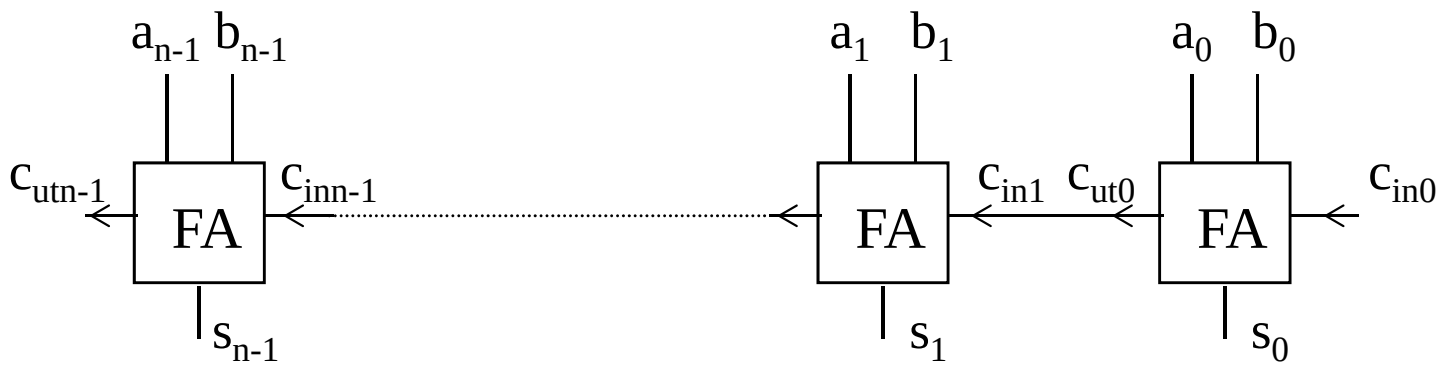
• generate

- Generate-statement kopplar ihop många likadana element

```
ENTITY adder IS
   GENERIC(N:integer)
   PORT(a,b:IN bit_vector(N-1 downto 0);
         sum:OUT bit_vector(N-1 downto 0));
END adder;
ARCHITECTURE structural OF adder IS
  COMPONENT full_adder
    PORT(a,b,cin:IN bit;cout,s:OUT bit);
  END COMPONENT;
  signal c:bit_vector(N-2 downto 0);
BEGIN
  G0:for i in 1 to N-2 generate
  U0:full_adder PORT MAP (a(i),b(i),c(i-1),c(i),s(i));
  end generate; -- G0
  U0:full_adder PORT MAP (a(0),b(0),'0',c(0),s(0));
  UN:full_adder PORT MAP (a(N-1),b(N-1),c(N-2),OPEN,s(N-1));
END structural;
```

Generera en n-bitsadderare!

`generate` n-bitsadderare



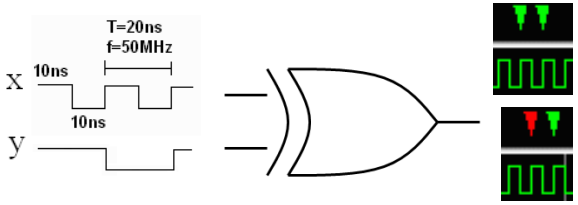
Fem rader kod genererar ripple-carry n-bitsadderaren från F5!

Testbänkar

ModelSim

- För att kunna test om ens konstruktion fungerar så måste man skapa en *testbänk*. Den har tre funktioner:
  - Generera stimuli för simulering
  - Applicera dessa stimuli till en entitet som skall testas
  - Jämföra utvärden med förväntade värden

Du kommer att använda en testbänk vid LAB 3. Ett tesbänksprogram kan outtröttligt prova igenom alla insignalskombinationer – det orkar inte Du!



Testbänk

ModelSim

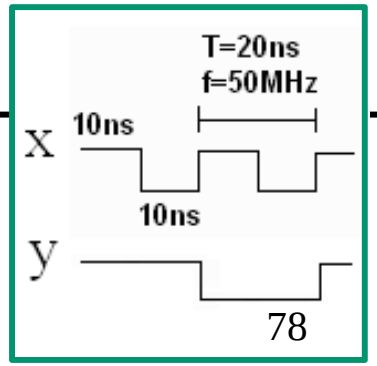
Testbänkens ENTITY är tom.

```
ENTITY testbench IS END testbench;

ARCHITECTURE xor_stimuli_1 of testbench IS
    COMPONENT xor_gate
      PORT(x,y:IN bit; q:OUT bit);
    END COMPONENT;
    signal x,y,u1,ut2,ut3:bit;
BEGIN
    x <= not(x) after 10 ns;
    y <= not(y) after 20 ns;
    U1:xor_gate PORT MAP (x,y,ut1);
    U2:xor_gate PORT MAP (x,y,ut2);
    U3:xor_gate PORT MAP (x,y,ut3);
END example;
```

Den krets som testas används som en komponent av testbänksprogrammet

Här genereras testsignalerna

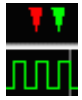


Testbänk

ModelSim

En testbänk kan markera när önskade händelser inträffar under körningen.

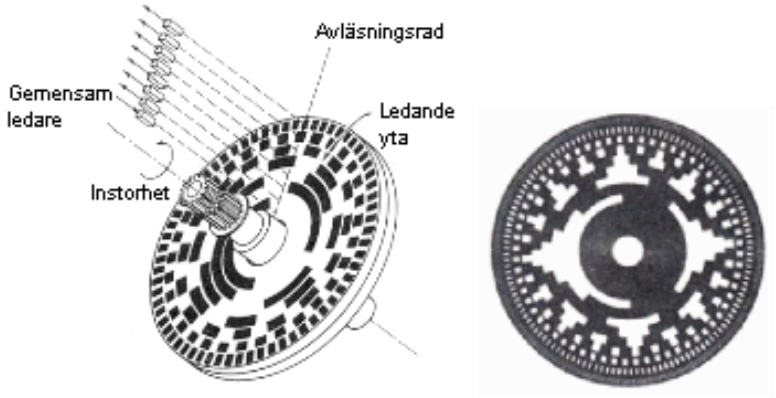
Eller markera när oönskade händelser uppträder



Resultatet av en körning med en testbänk kan sparas i en fil, som bevis att allt är **ok** – eller som hjälp vid felsökning om det nu inte gick bra.

Kodskiva/kodlinjal

Kodskivor:
(roterande rörelse)



Vinkelmätare med kodskiva. Till vänster binärkod, till höger Graykod.

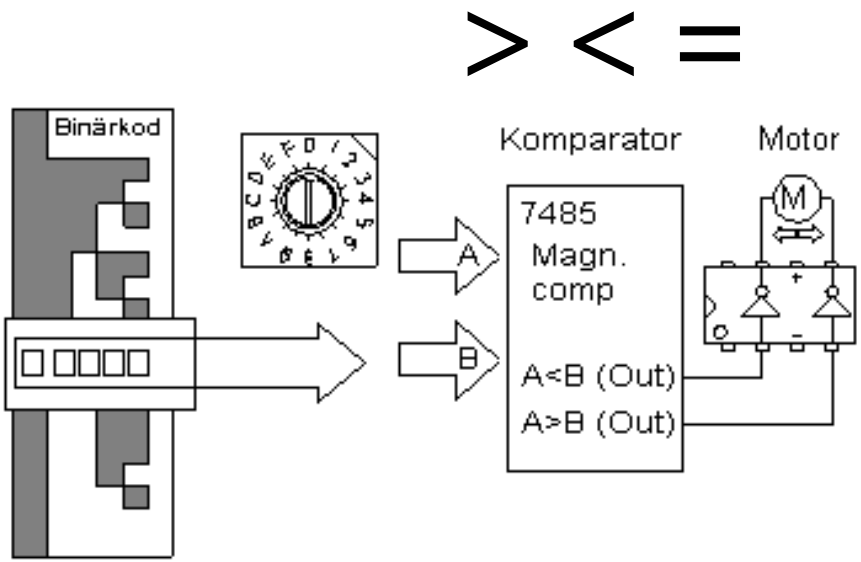
Kodlinjaler:
(linjär
rörelse)

För linjär rörelse används kodlinjaler. Överst binärkodad eller nederst Graykodad.

Experiment med kodlinjaler

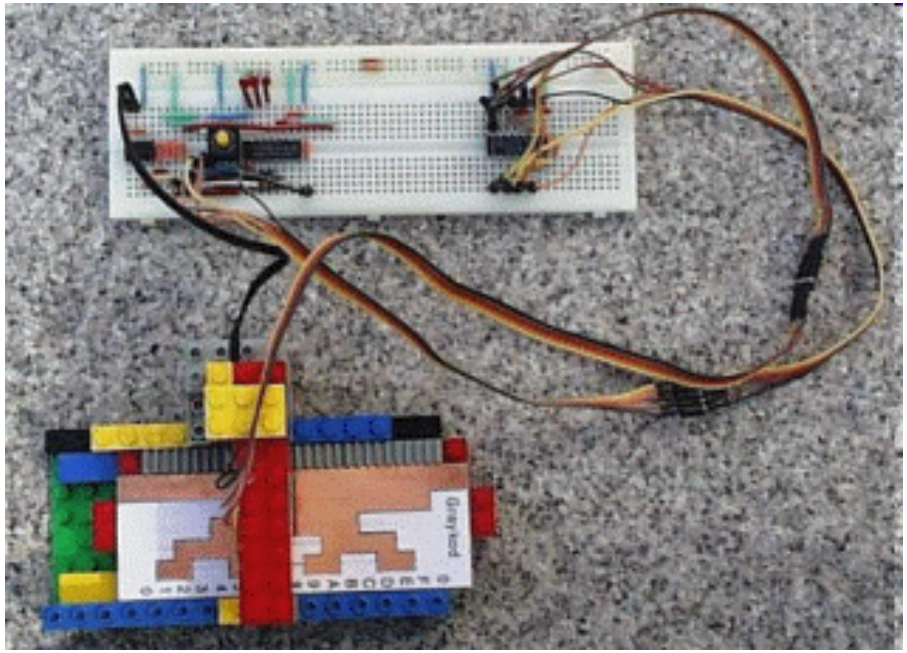
Koden från en **Hex-kodad vridomkopplare** jämförs med koden från en **kodlinjal** i en **digital komparator**. Komparatorns **större än**/**mindre än** utgångar driver en motor som förflyttar kodlinjalen tills de båda koderna blivit lika.

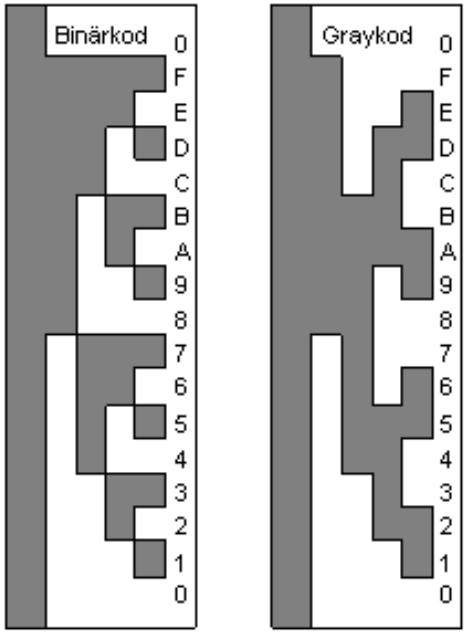
I teorin följer kodlinjalen lydigt med vridomkopplarens läge, men verkligheten blir något helt annat ...



Experimentutrustningen

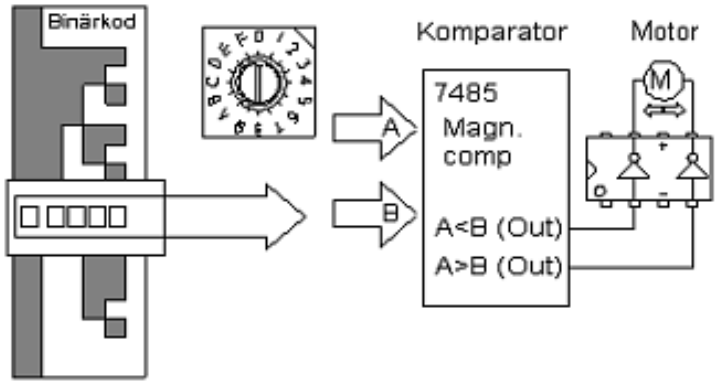
Följa John …



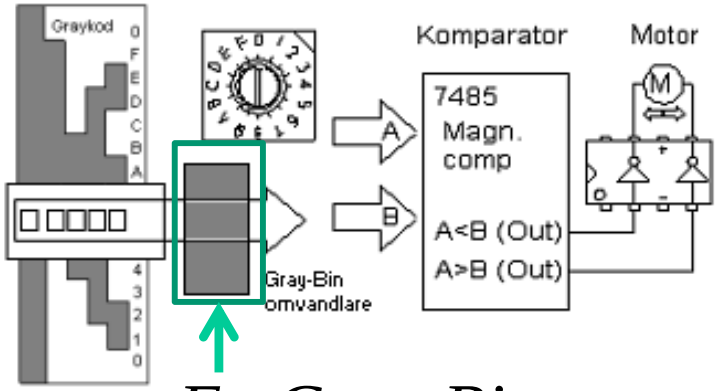


Jämför Binär-kod med Gray-kod

Binärkod:



Gray-kod:



Vad tror Du finns inuti Gray-Bin-omvandlaren?

En Gray-Bin-omvandlare behövs nu

Blir det problemfritt med Graykoden?