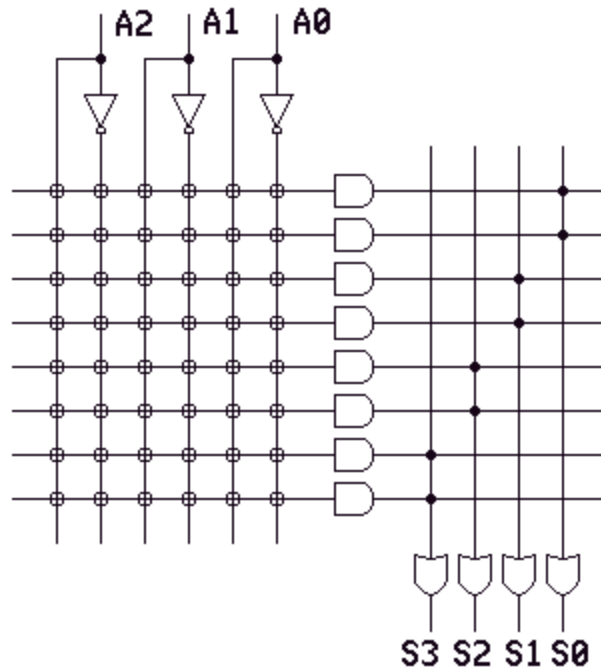
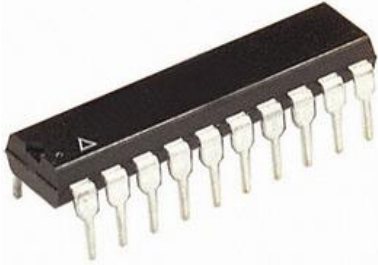


IE1205 Digital Design:

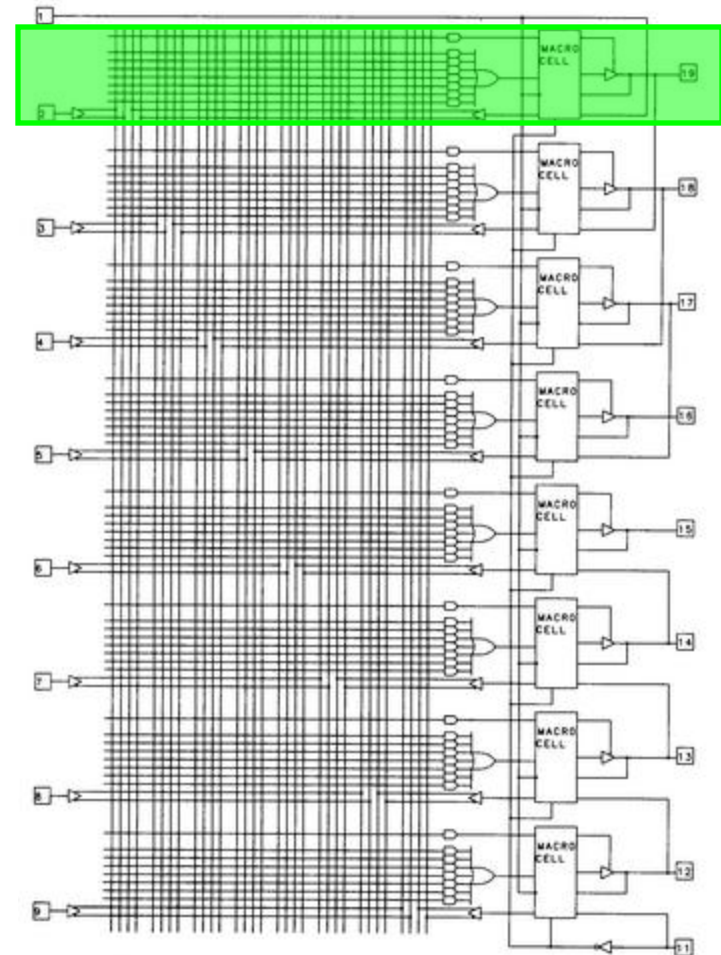
F7 : Kombinatorik

PLD (tex. PAL)

Programmable Logic Device (Programmable Array Logic)

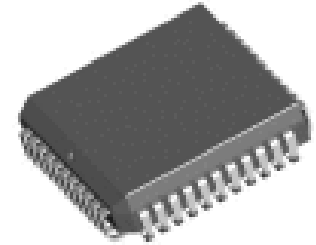
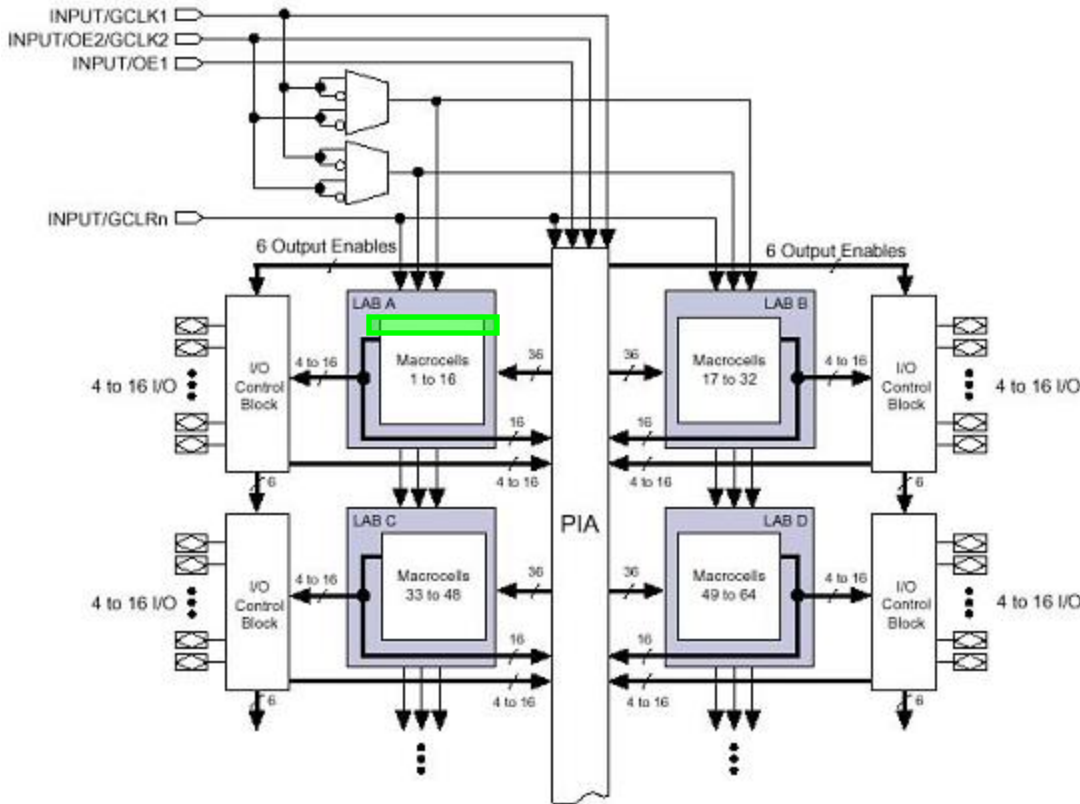


Typiskt 8 st logikelement
Teknik: AND-OR array



CPLD (tex. MAX)

Complex Programmable Logic Device

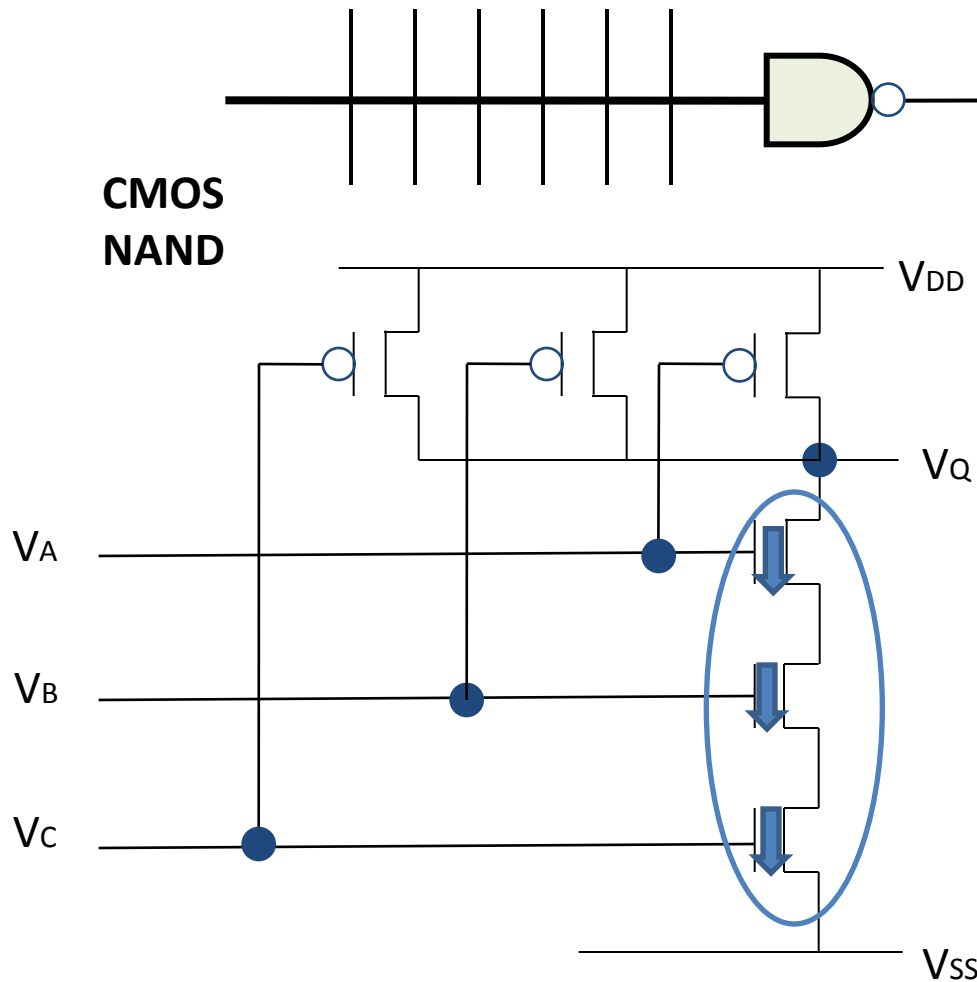


Typiskt **64** Macroceller

Teknik:
AND-OR array

(större MAX har MUX-
tree teknik)

Grindar med många ingångar?

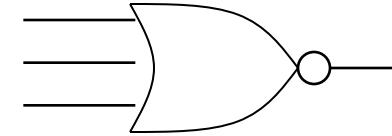
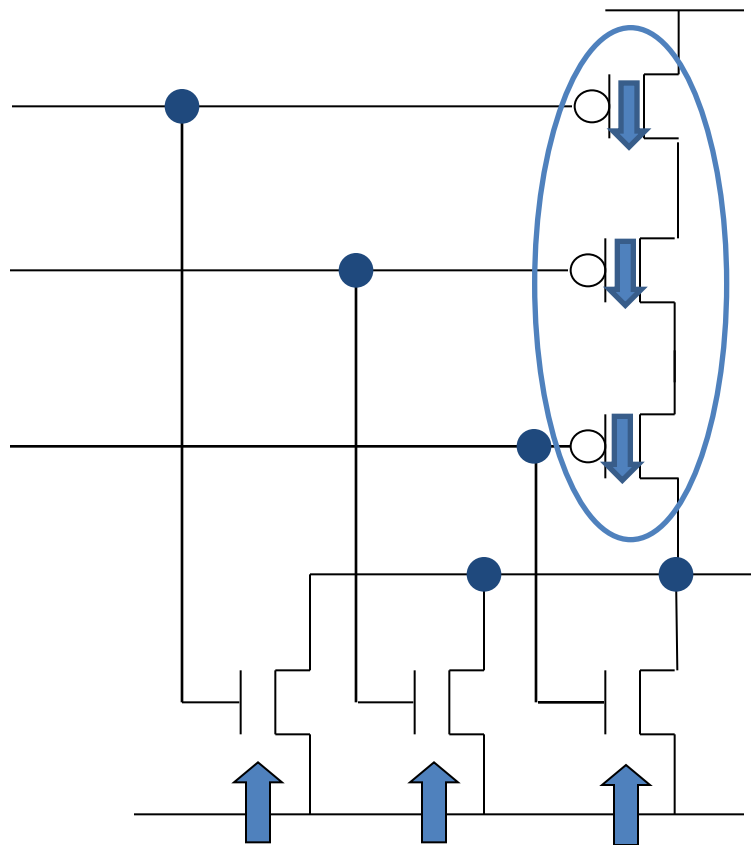


PAL-matrisen har grindar med så många ingångar att man måste rita dem med ett "förenklat ritsätt"

Alla måste leda för "0" = **långsamt** om det är många ingångar

Lika illa med CMOS NOR

**CMOS
NOR**



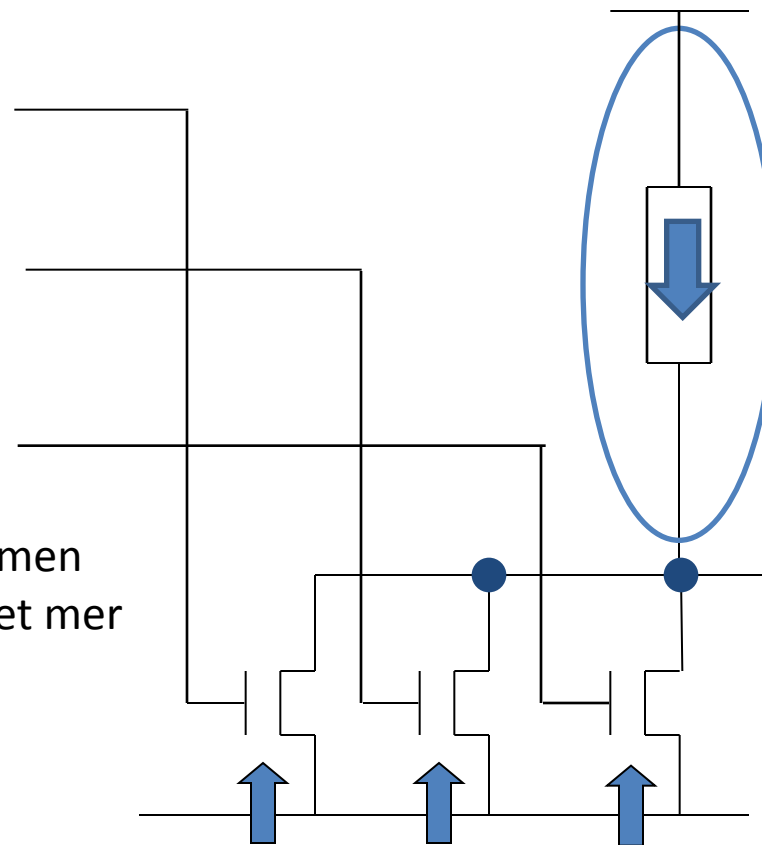
Alla måste leda för "1" =
långsamt om det är
många ingångar

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!

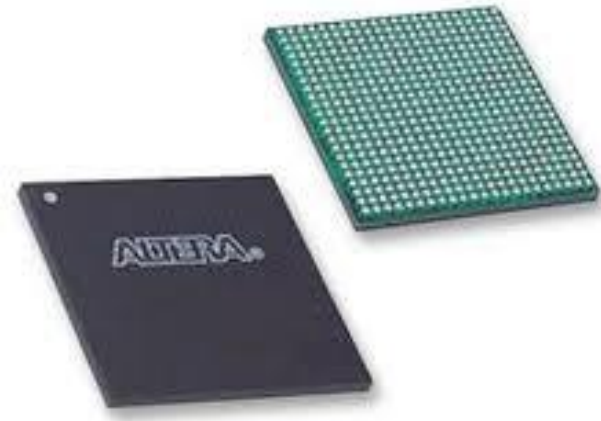
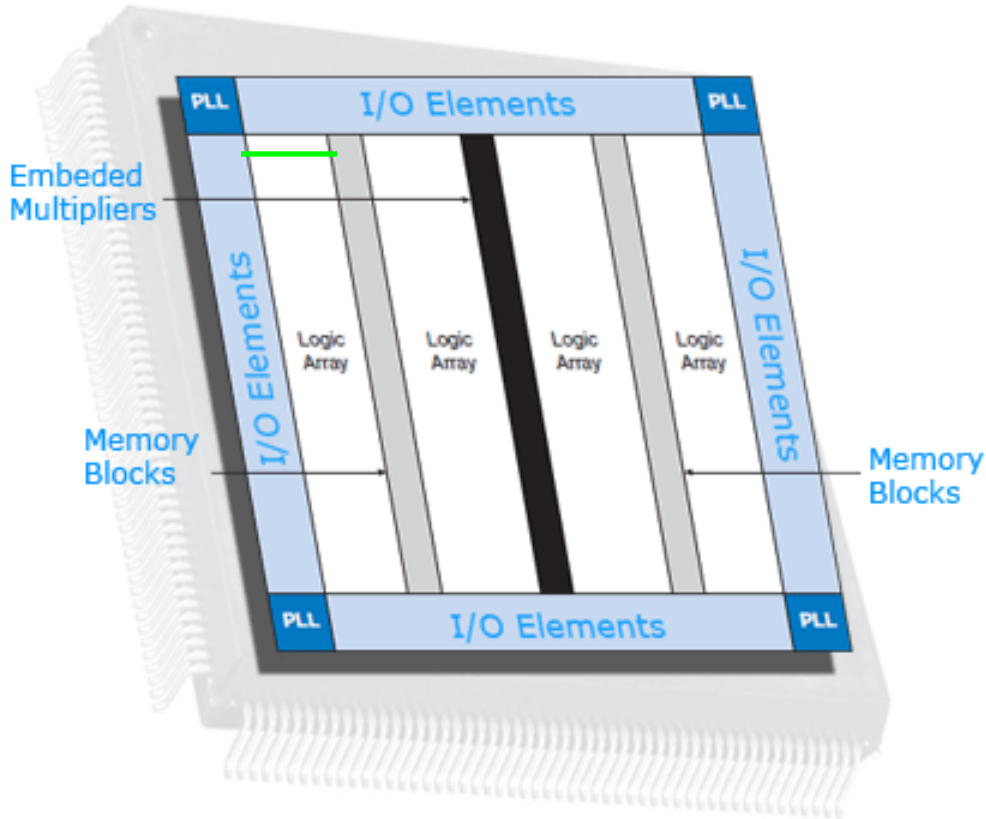


"Pull-Up" resistor ger "1"
= **snabbt** men "**Power
Hungry**" vid "0"

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

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

FPGA (tex. Cyclone II)



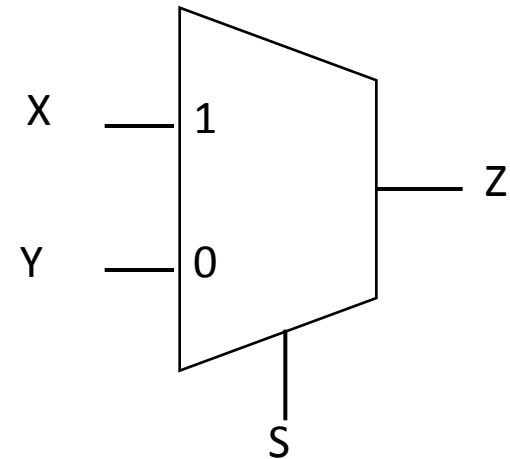
Typiskt **50000** logikelement

Teknik: **MUX** tree

Multiplexorn MUX

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

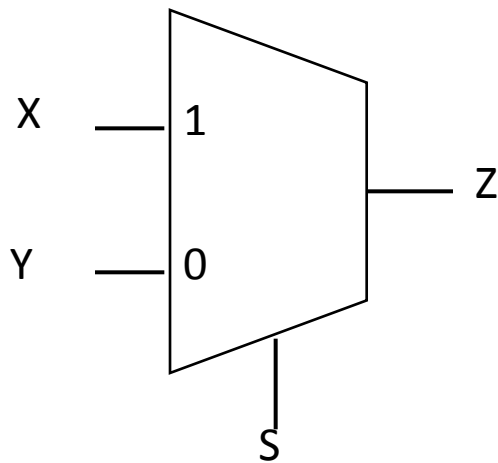
**MUX är numera
"standardkomponenten" vid
framtagandet av Digital logik.**



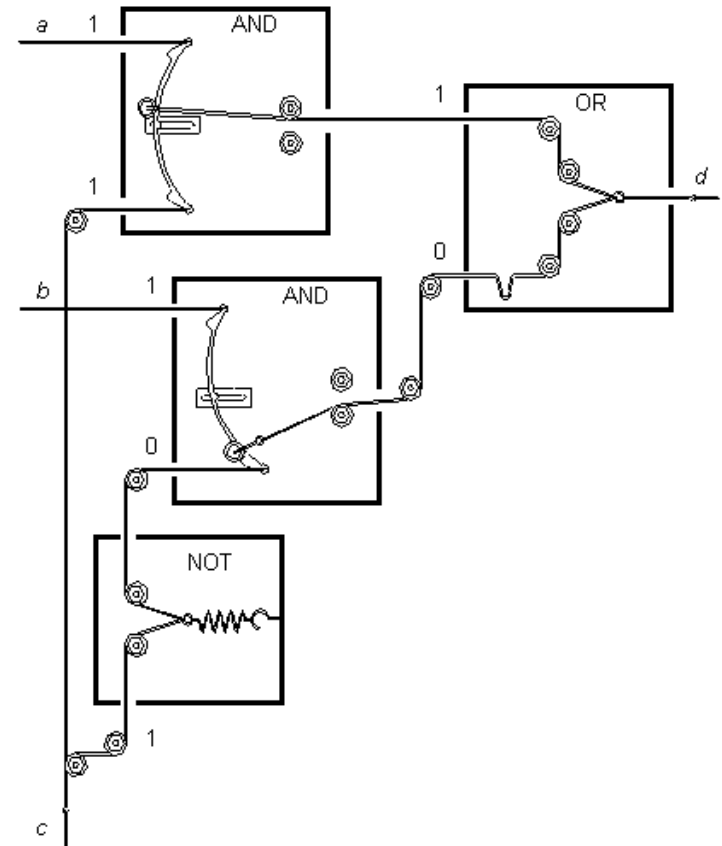
$$Z = SX + \bar{S}Y$$

Multiplexorn MUX

Till höger har vi en MUX i reptechnik – ett April-skämt från Scientific American!



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



Logiska funktioner med MUX

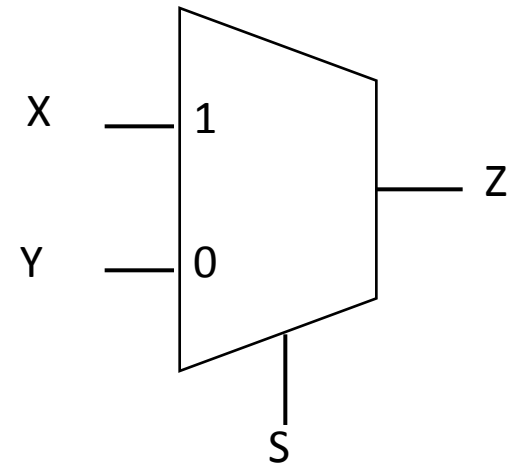
Hur kan följande funktioner implementeras med en 2:1 Multiplexor?

$$Z = \bar{x}_0 \quad \text{NOT}$$

$$Z = x_1 \cdot x_0 \quad \text{AND} \quad ?$$

$$Z = x_1 + x_0 \quad \text{OR}$$

$$Z = x_1 \oplus x_0 \quad \text{XOR}$$

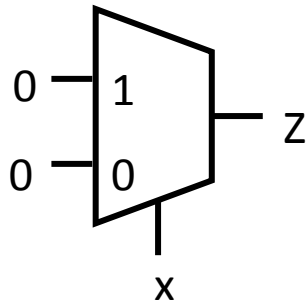


$$Z = SX + \bar{S}Y$$

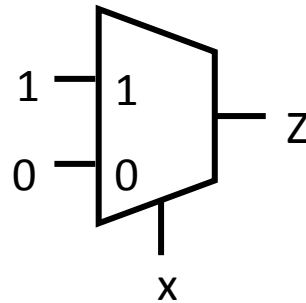
Snabbfråga

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

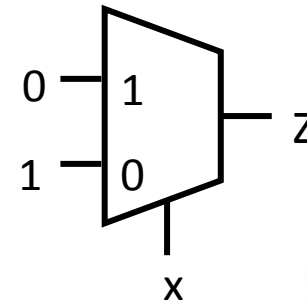
Önskad funktion: $z = \bar{x}$



Alt: A



Alt: B



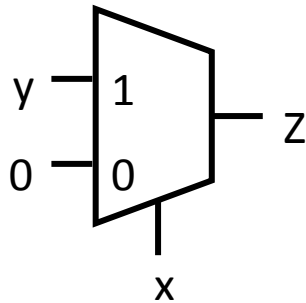
Alt: C



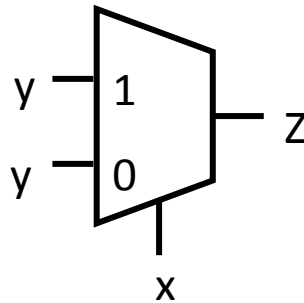
Snabbfråga

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

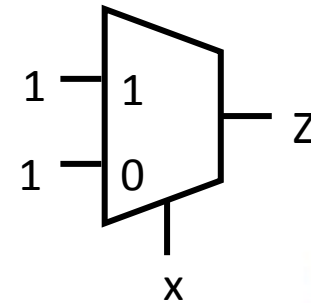
Önskad funktion: $z = xy$



Alt: A



Alt: B



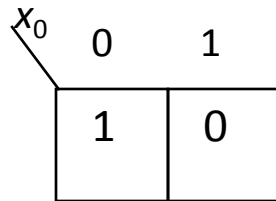
Alt: C



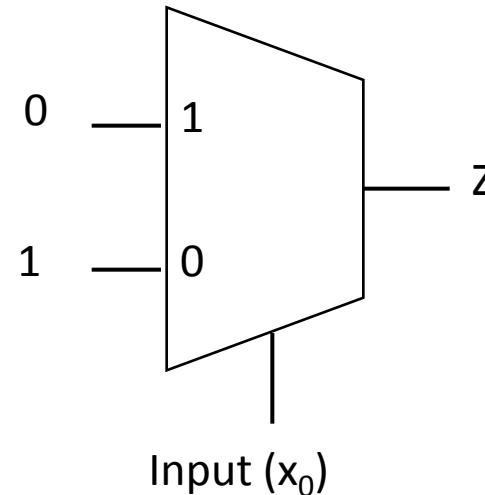
Invertering NOT med MUX

Specification:

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



$$Z = \overline{x_0} \quad \text{NOT}$$

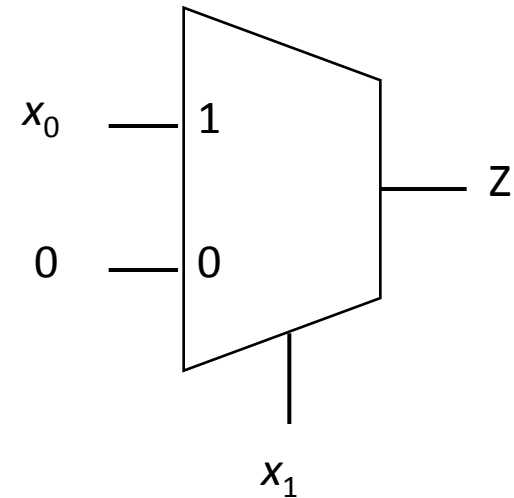


AND-funktion med MUX

Specification:

	x_0	0	1
x_1	0	0	0
1	1	0	1

$$Z = x_1 \cdot x_0 \quad \text{AND}$$



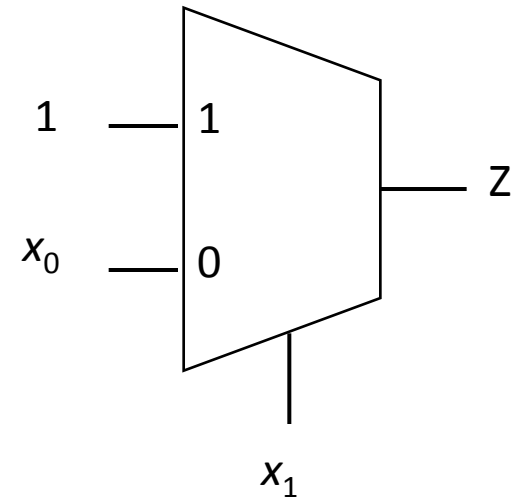
$$Z = SX + \bar{S}Y = x_1 \cdot x_0 + \bar{x}_1 \cdot 0 = x_1 \cdot x_0$$

OR-funktion med MUX

Specification:

	x_0	0	1
x_1	0	0	1
	1	1	1

$$Z = x_1 + x_0 \quad \text{OR}$$



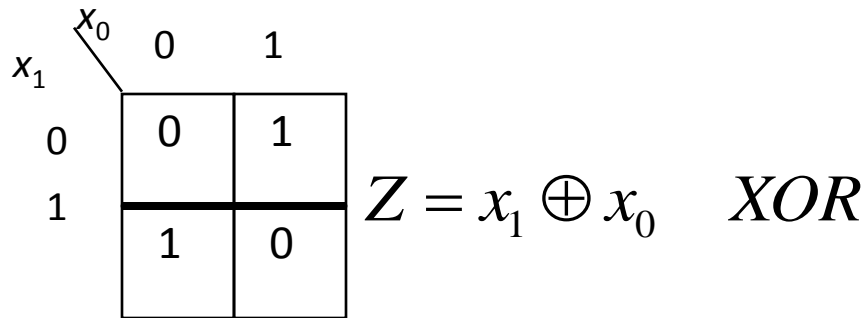
$$Z = SX + \bar{S}Y =$$

$$= x_1 \cdot 1 + \bar{x}_1 \cdot x_0 = x_1(x_0 + \bar{x}_0) + x_1 \cdot x_0 =$$

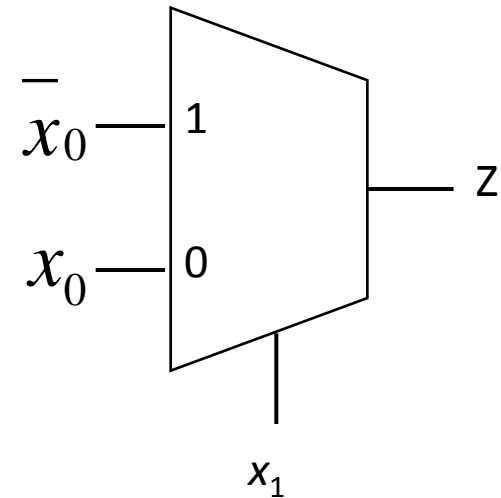
$$= x_1x_0 + \bar{x}_1x_0 + x_1\bar{x}_0 = x_1 + x_0$$

XOR-funktion med MUX

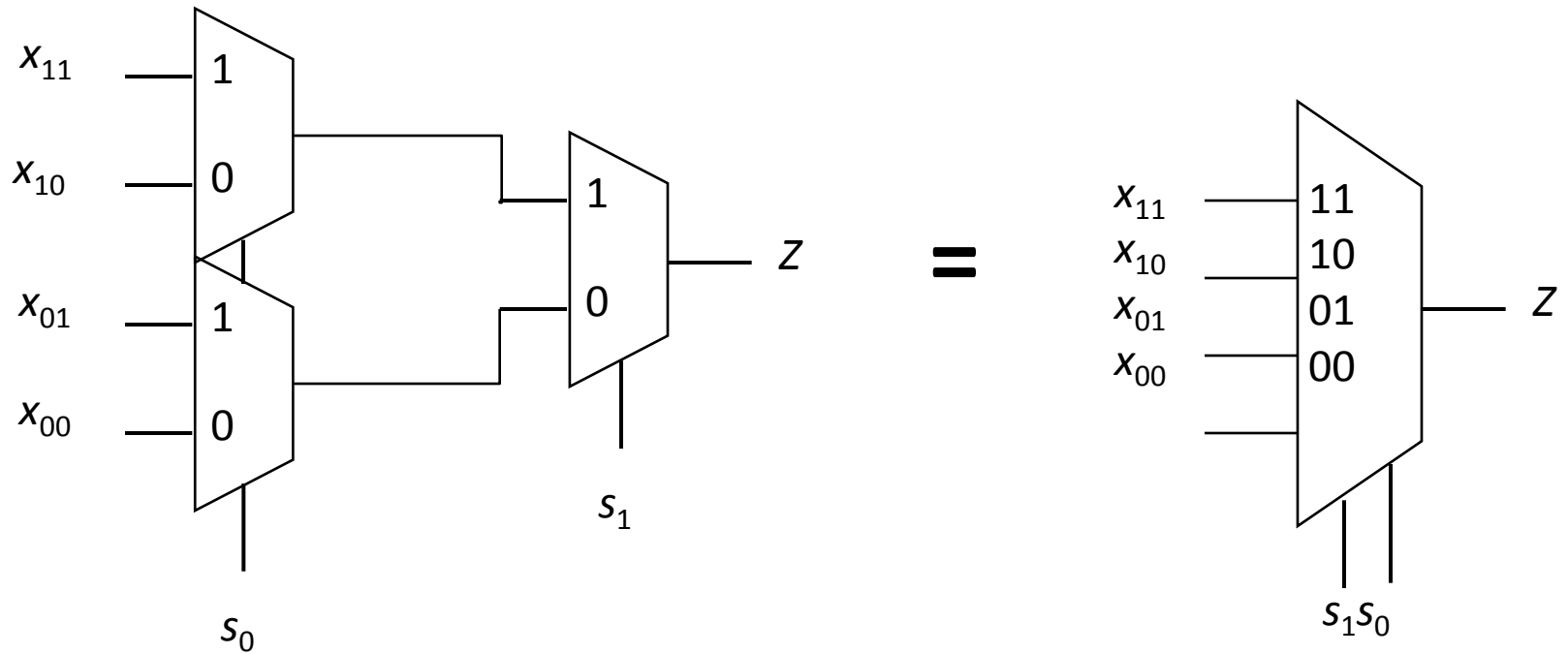
Specification:



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

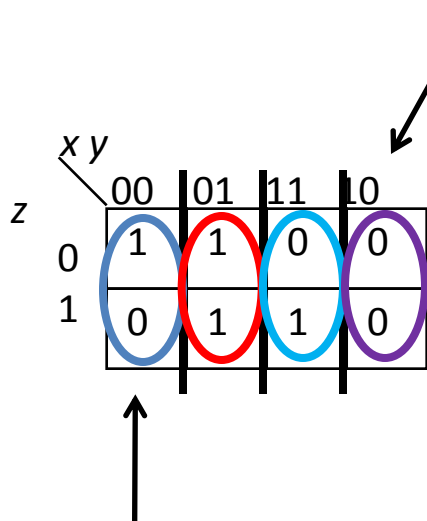


Hierarkier av Muxar

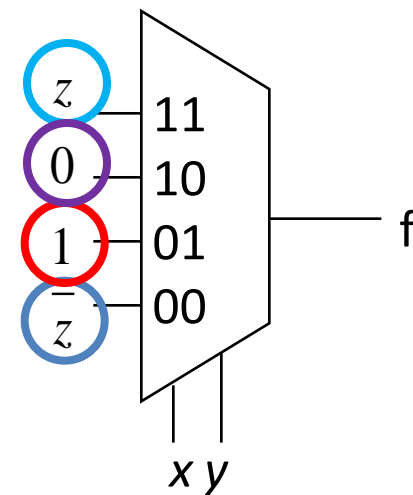


Större funktioner med MUXar

Välj några av ingångarna som address-ingångar ...



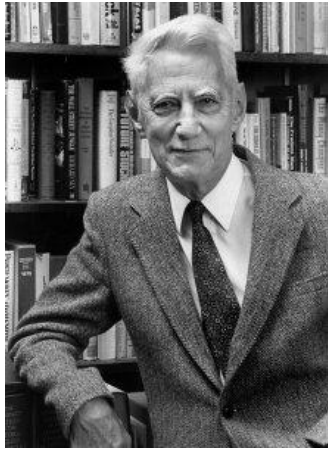
$$f = \bar{z}\bar{x} + \bar{x}y + zy$$



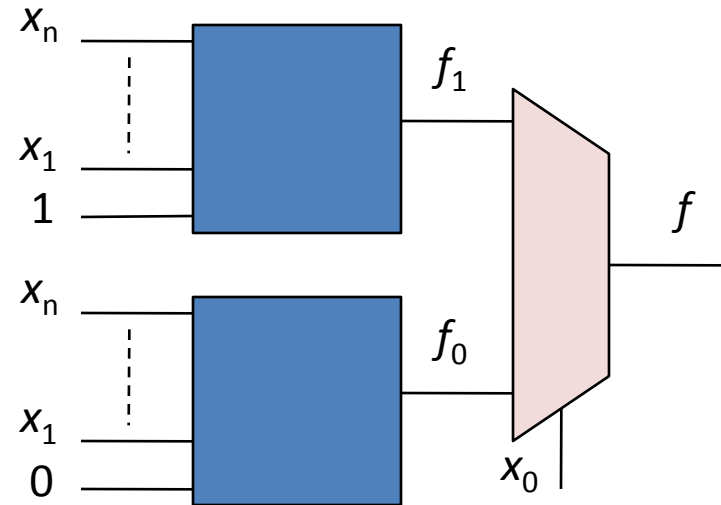
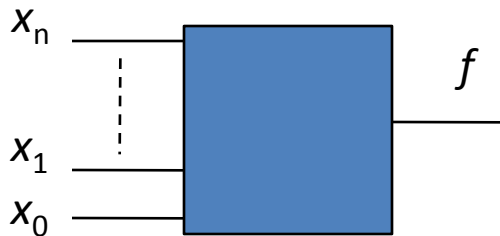
...och minimera/implementera funktionen som uppstår för varje ingång. Rita nya Karnaugh-diagram om det behövs.

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

Shannon dekomposition



Claude Shannon
matematiker/elektrotekniker
(1916 –2001)



Varning! Detta saknas i "Hemert"



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

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

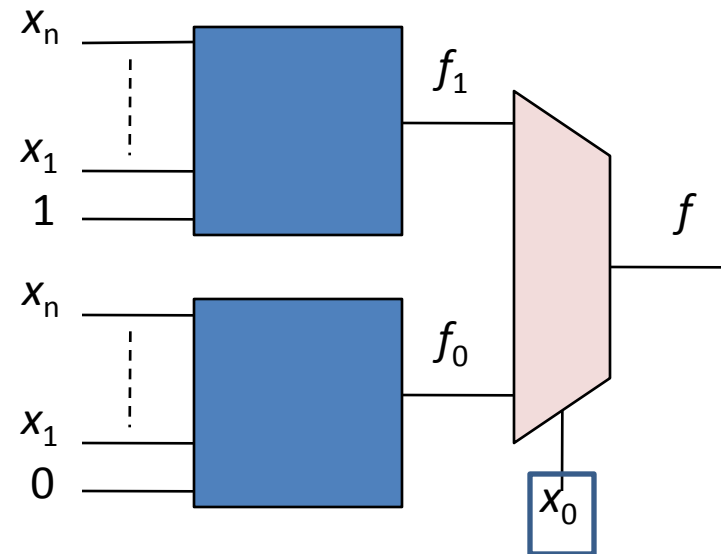
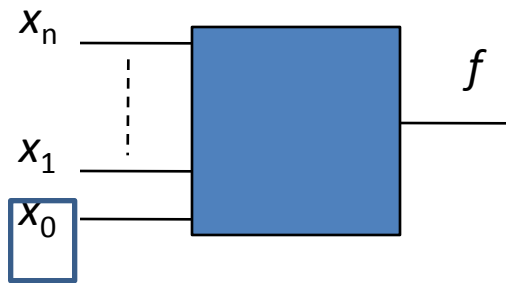


Shannon dekomposition

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

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

Funktionen kan sedan implementeras med en multiplexer.

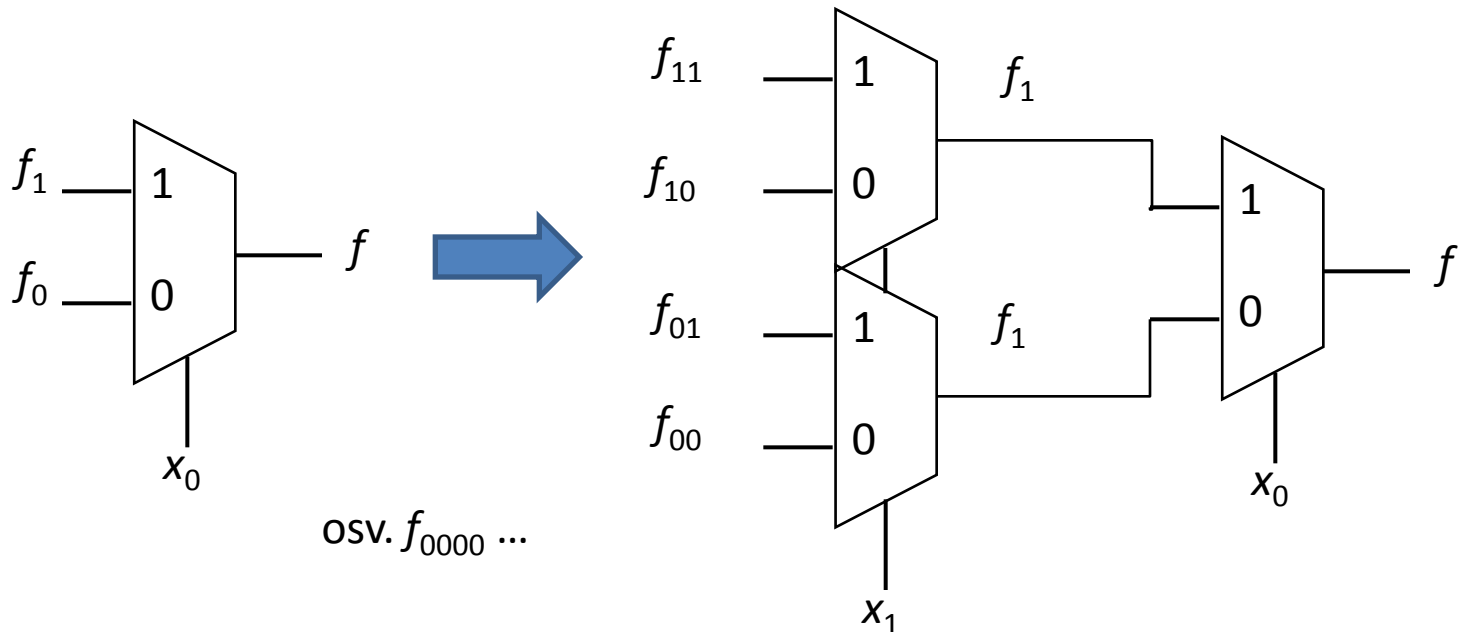


Rekursivt

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

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

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



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

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 $f(x_n, \dots, x_1, 1)$ (= vänstra termen på högra sidan)

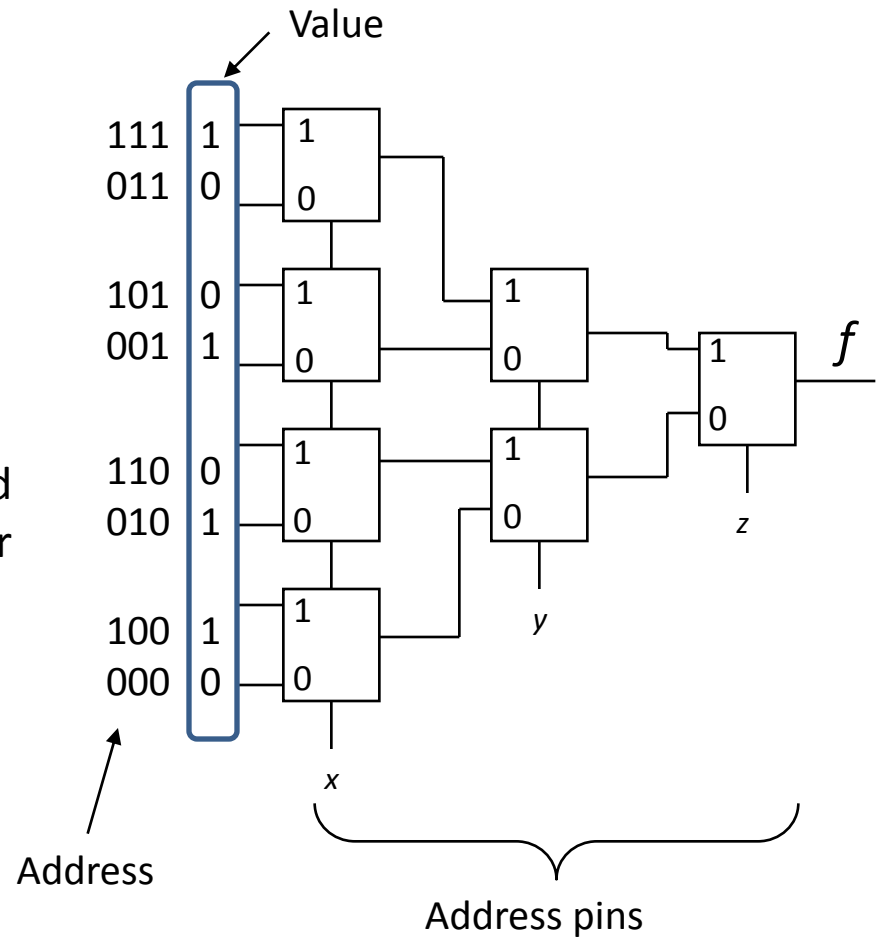
om $x_0=0$ så blir f lika med $f(x_n, \dots, x_1, 0)$ (= högra termen på högra sidan)

LHS=RHS

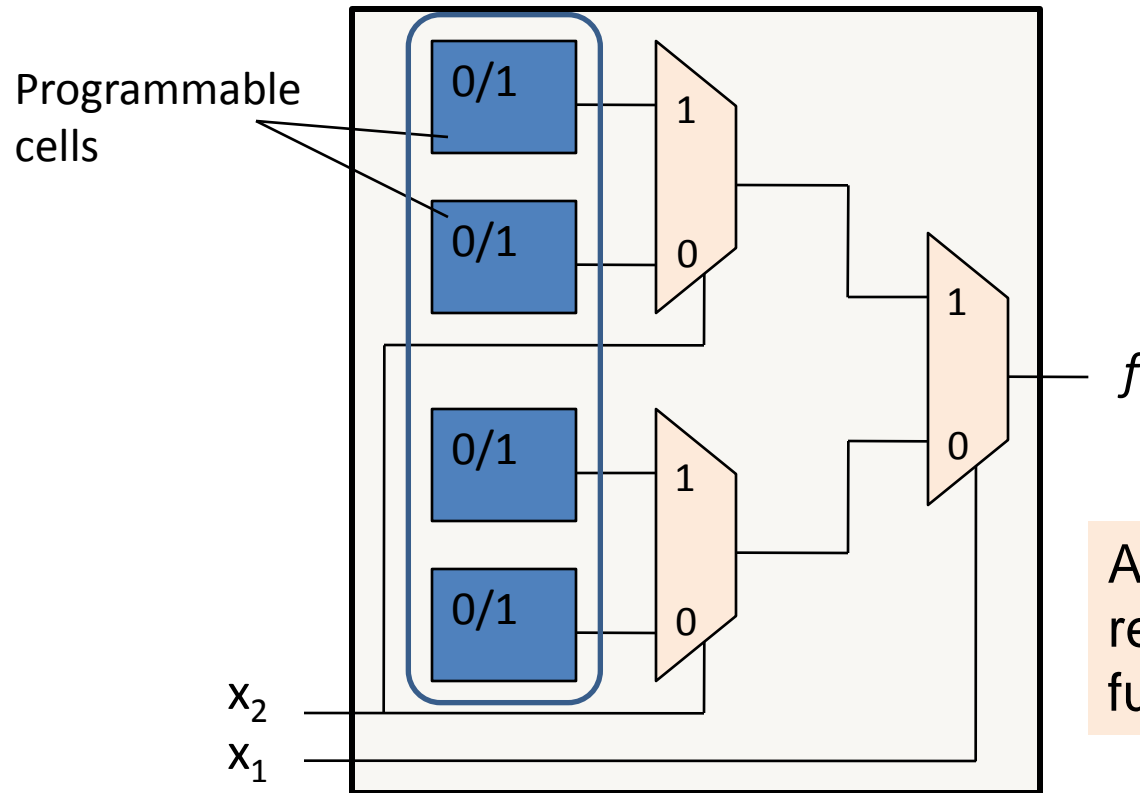
MUX networks

		f			
		00	01	11	10
x	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 .



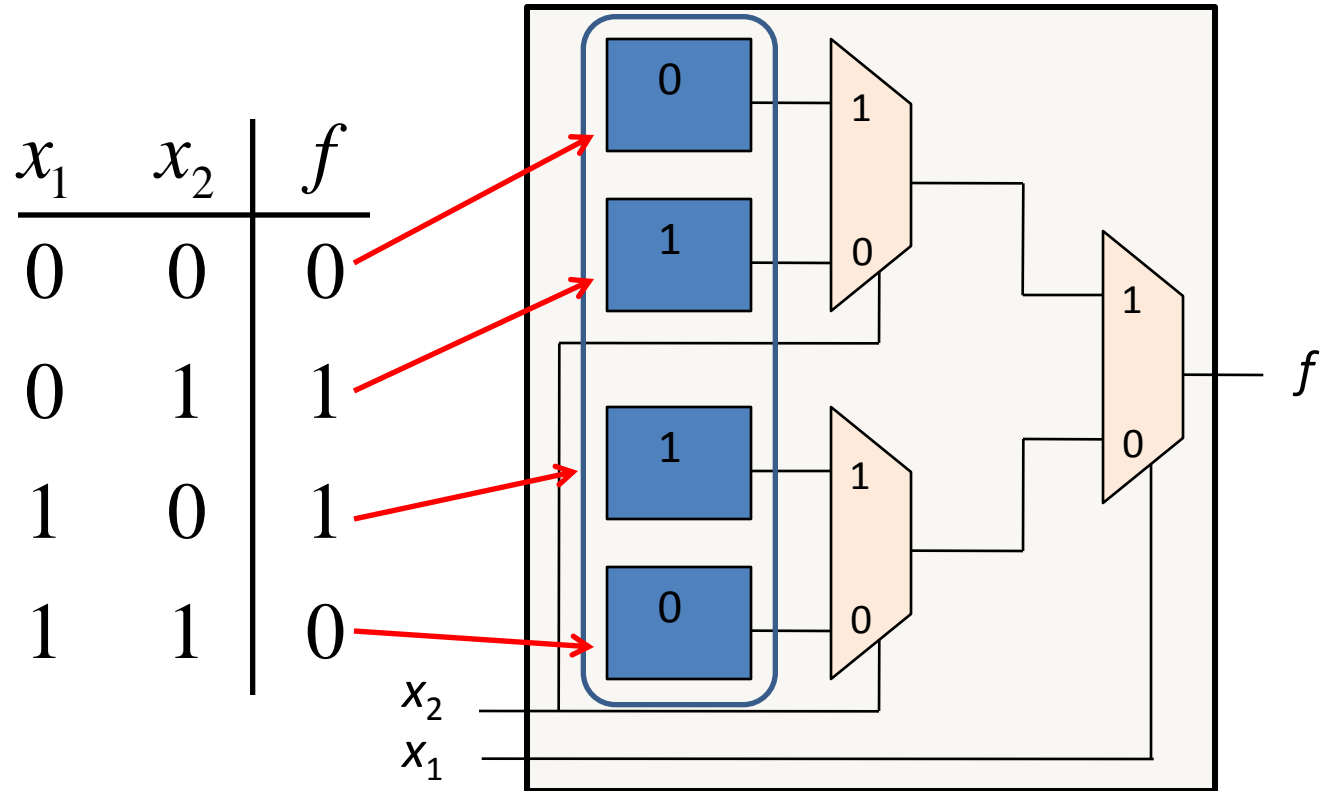
Look-up-table (LUT)



Two-input LUT

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

LUT för XOR-grind

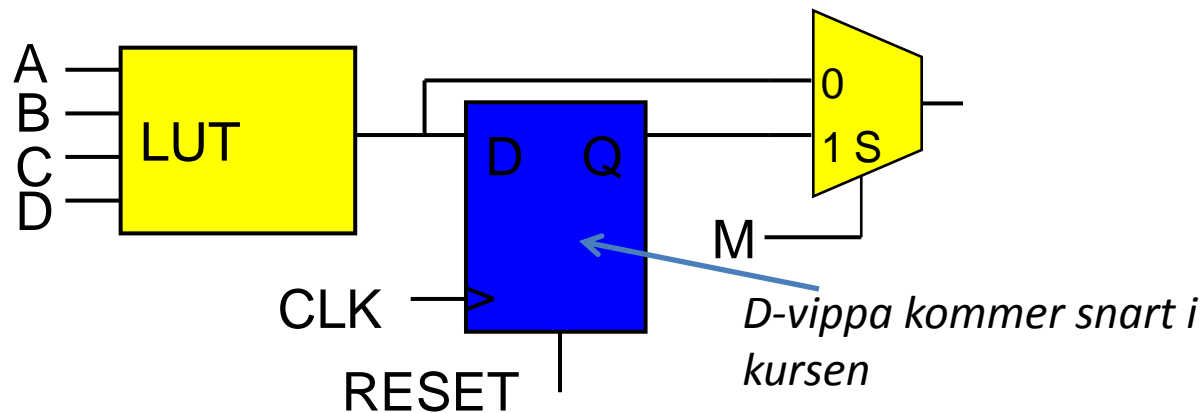


Two-input LUT

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

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

MSB x_3 x_2 $x_1 x_0$ LSB

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

*Känner Du
igen
funktionen ... ?*

Bit # 15
Bit # 1
Bit # 0

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

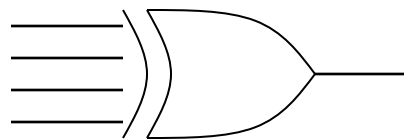
LUT funktionernas nummer

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

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

Udda paritet!

Inga hoptagningar.



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

Nu vet Du vilken funktion som är **6996**!

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

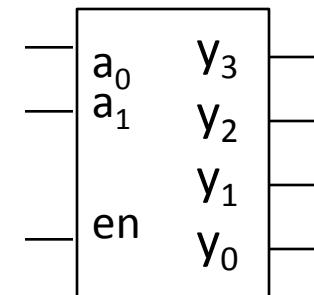
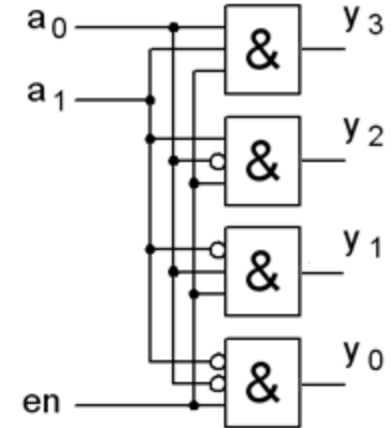
Decoder (Avkodare)

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

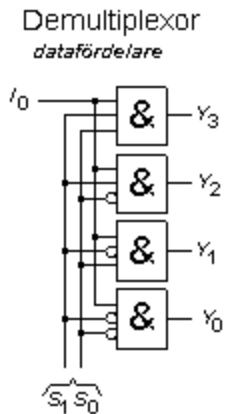


2-till-4 avkodare

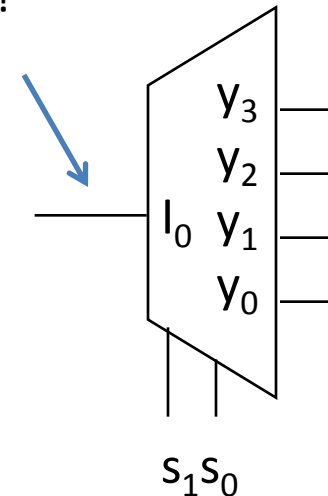
Demultiplexor (DMUX)

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

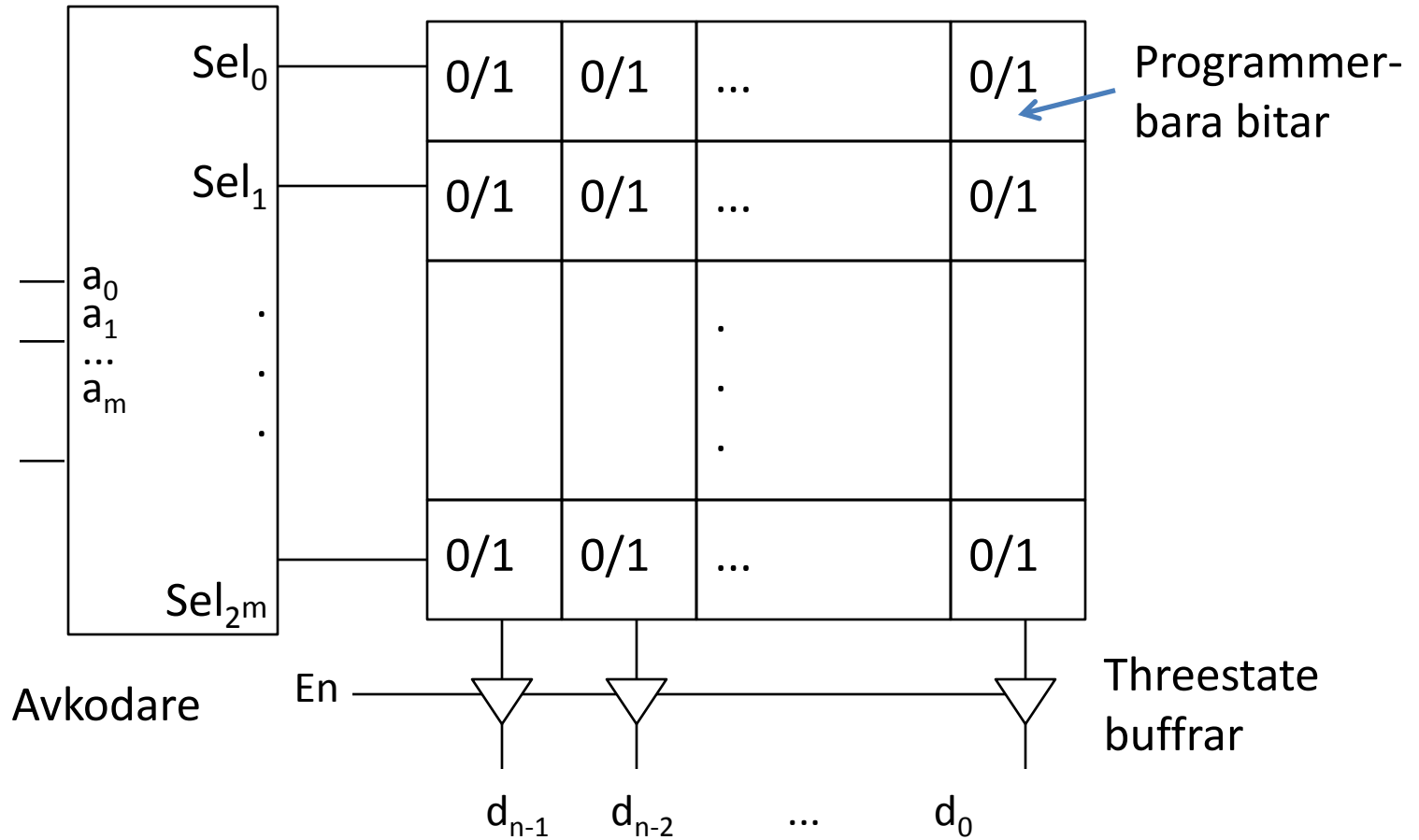
I_0	s_1	s_0	Y_0	Y_1	Y_2	Y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	-	-	0	0	0	0



Nu kallas en
för I_0 !



Read-only-memory (ROM)



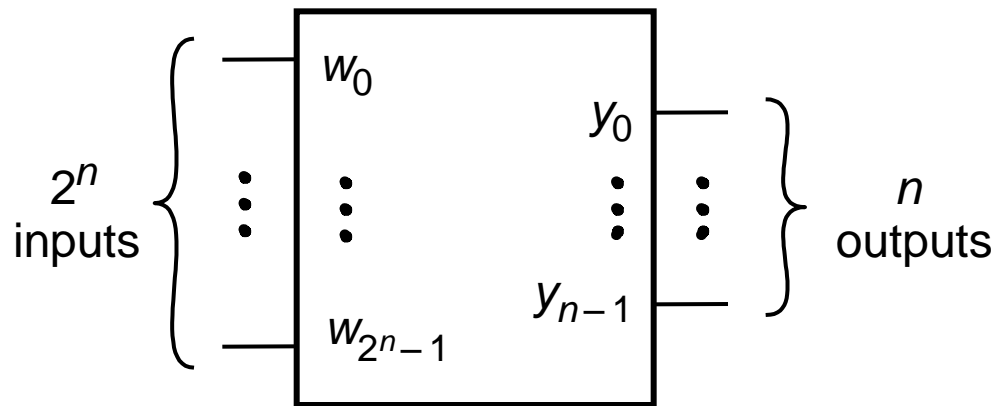
Encoder

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

- Informationen **koncentreras kraftigt**

Tangentbord med 256 (2^8) tangenter

8-bitars ASCII-kod för nedtryckt tangent



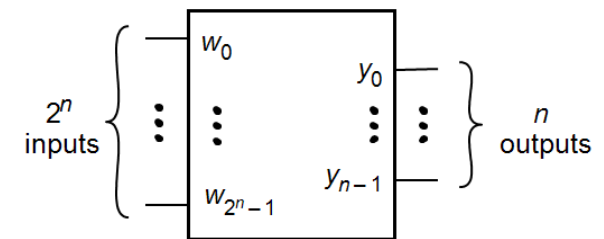
Prioritetsenkoder

En Priority Encoder ger tillbaka adressen 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 samtidigt?

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	-	-

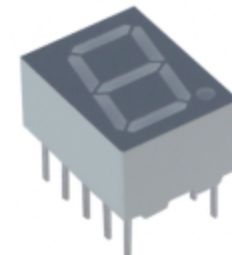
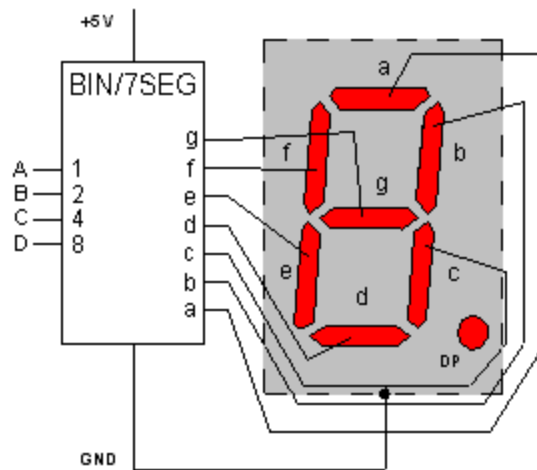


Nu blir det väldefinierat vad som ska hända om flera ingångar är aktiva.

Kod-konverterare

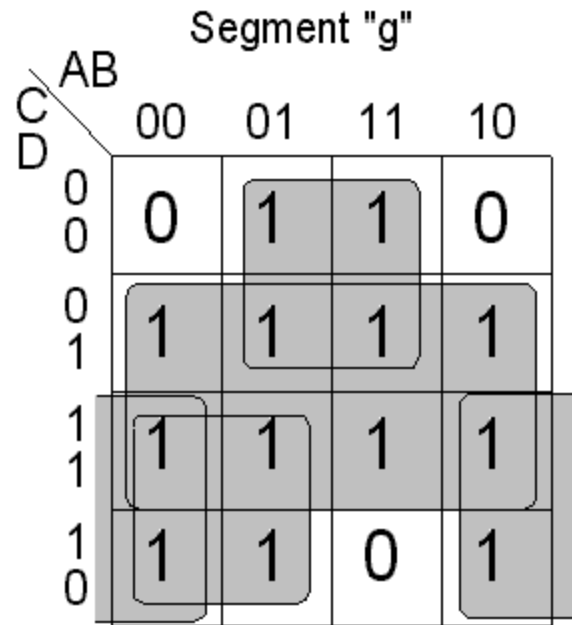
Kod-konverterare översätter från en kod till en annan. Typiska exempel är

- 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



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

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

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

Den optimala 7-segmentavkodaren är nog redan uppfunnen!

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

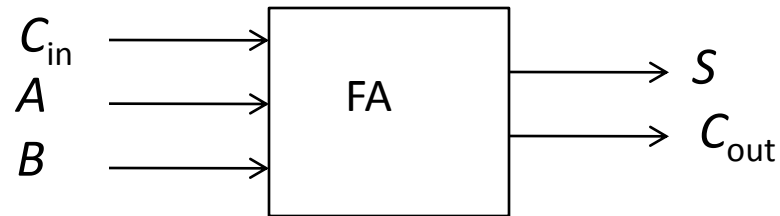
Verilog är också ett språk som används för att specificera hårdvara

- Används mest i USA

andra sätt att beskriva hårdvara:

System C, Matlab

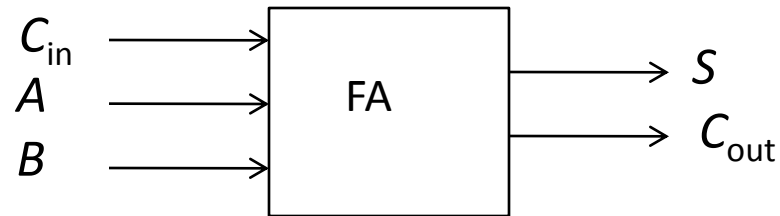
Entity



```
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

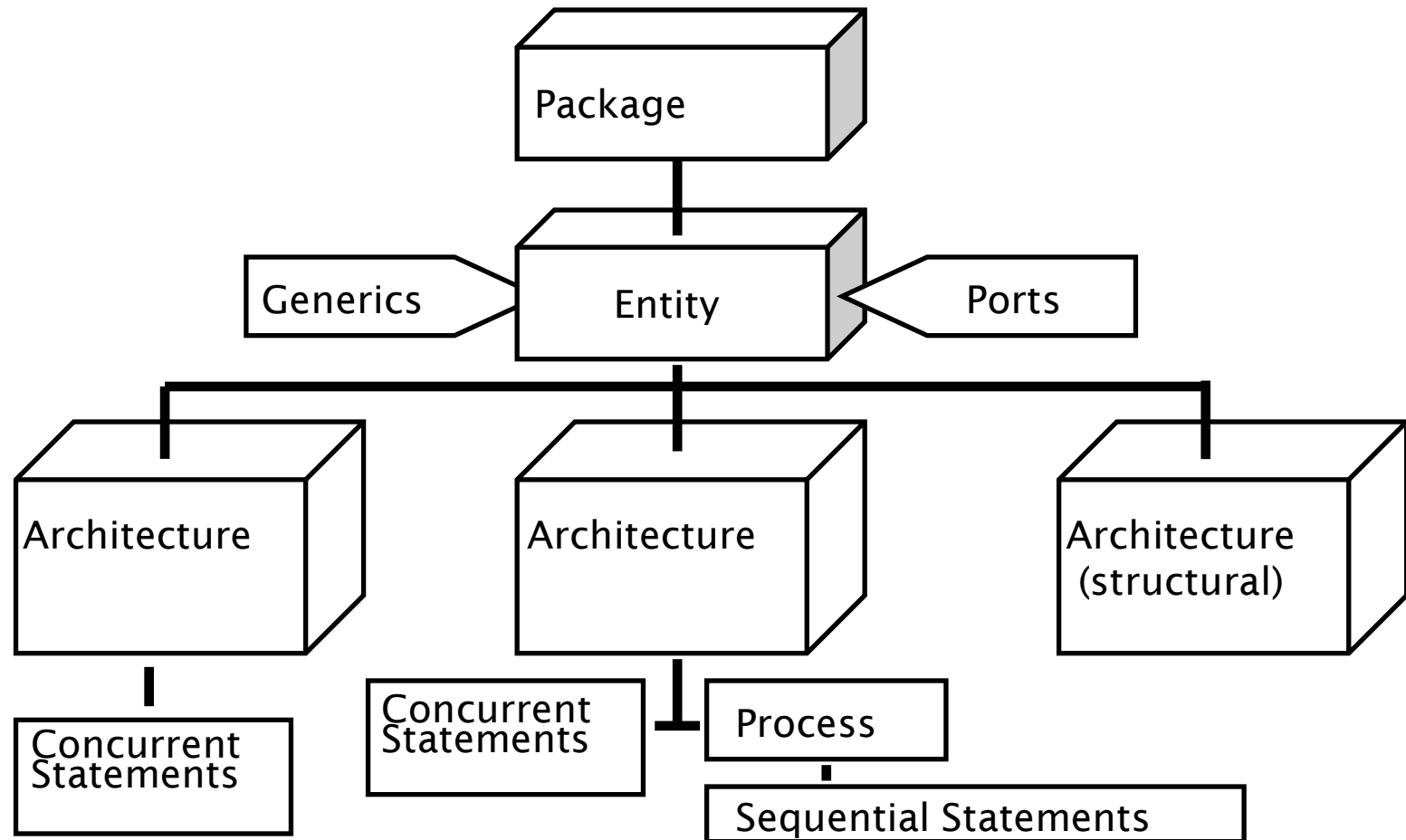
- kunna kontrollera att man har tänkt rätt genom att *simulera* kretsen
- kunna beskriva stora konstruktioner på ett enkelt sätt och sedan generera kretsen genom syntes
- möjliggöra strukturerade beskrivningar av en krets

VHDL ökar abstraktionsnivån!

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 hiarkin



Entitet (eng. Entity)



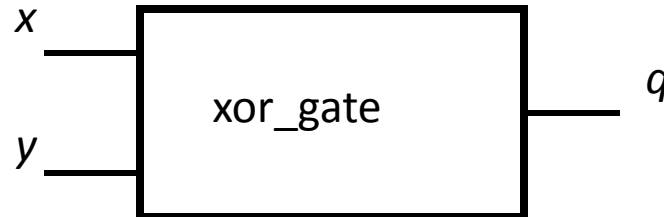
- Den primära abstraktions-nivån i VHDL kallas för *entity*
- I en beteende-beskrivning definieras entiteten genom sina svar på signaler och ingångar
- En beteende-modell är samma sak som en "svart låda"
 - Insidan syns inte från utsidan
 - Entitetens beteende definieras av den svarta lådans funktionalitet



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!

- 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

std_logic vs bit



bit:

'0' or '1'

std_logic:

'U': uninitialized. This signal hasn't been set yet.

'X': unknown. Impossible to determine this value/result.

'0': logic 0

'1': logic 1

'Z': High Impedance

'W': Weak signal, can't tell if it should be 0 or 1.

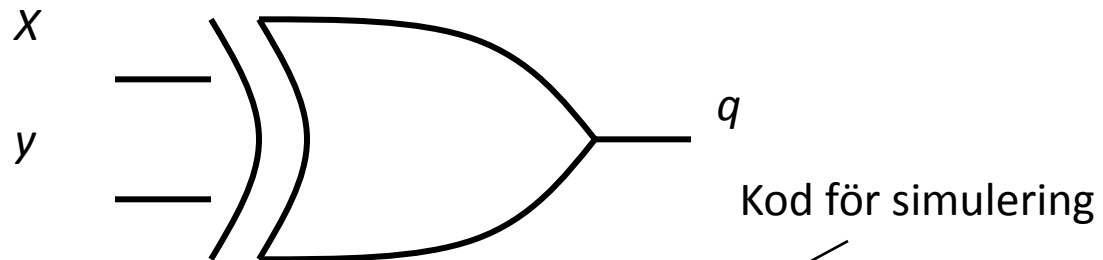
'L': Weak signal that should probably go to 0

'H': Weak signal that should probably go to 1

'-': Don't care.

Architecture

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

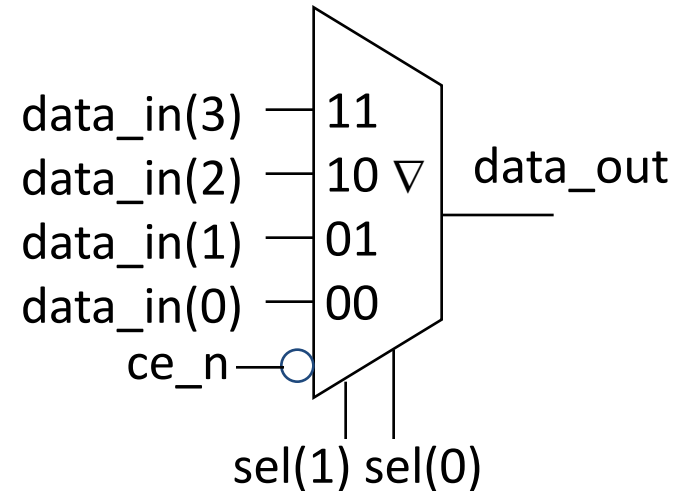


```
ARCHITECTURE behavior OF xor_gate IS
BEGIN
    q <= a xor b after 5 ns;
END behavior;
```

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'; ← Threestate!
    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;
```

Syntesverktøyet Quartus II



QuartusII



Mer utførligt i kommande kurser!

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

Överkurs VHDL



Signal deklARATIONEN



Signal-deklarationen används inuti arkitekturer för att deklarerera 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")

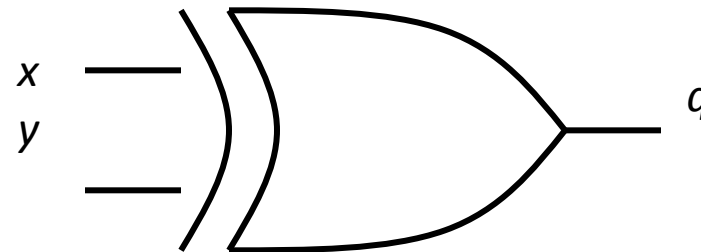
Sekvensiell eller Parallell kod



- 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

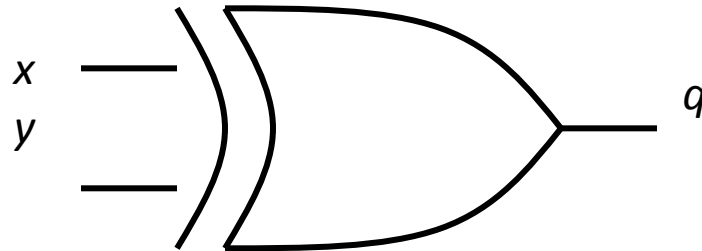


```
process (x, y)
begin
  if (x/=y) then
    q <= '1';
  else
    q <= '0';
  end if;
end process;
```

Betyder not!

Dataflödes stil

XOR-gate



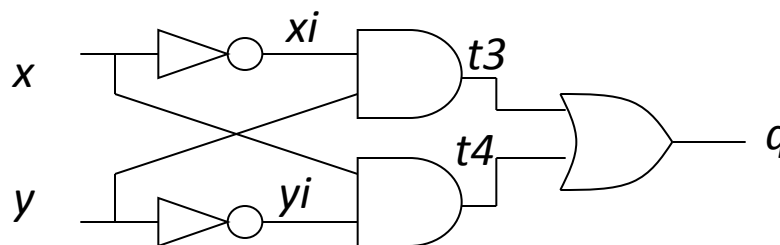
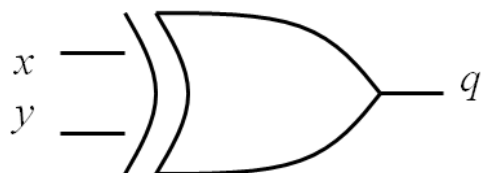
```
q <= a xor b;
```

eller i "behavioural dataflow style"

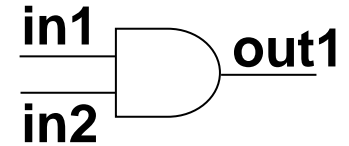
```
q <= '1' when a/=b else '0' ;
```

olikhet

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);
```



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

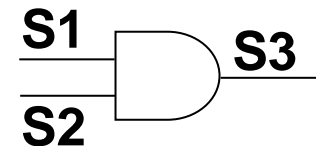
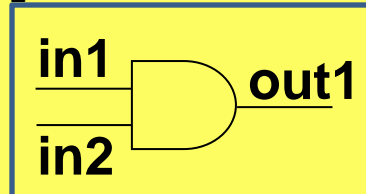
```
ARCHITECTURE test OF test_entity
  COMPONENT and_gate
    PORT ( in1, in2 : IN BIT;
           out1 : OUT BIT);
  END COMPONENT;
  ... more statements ...
```

- 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

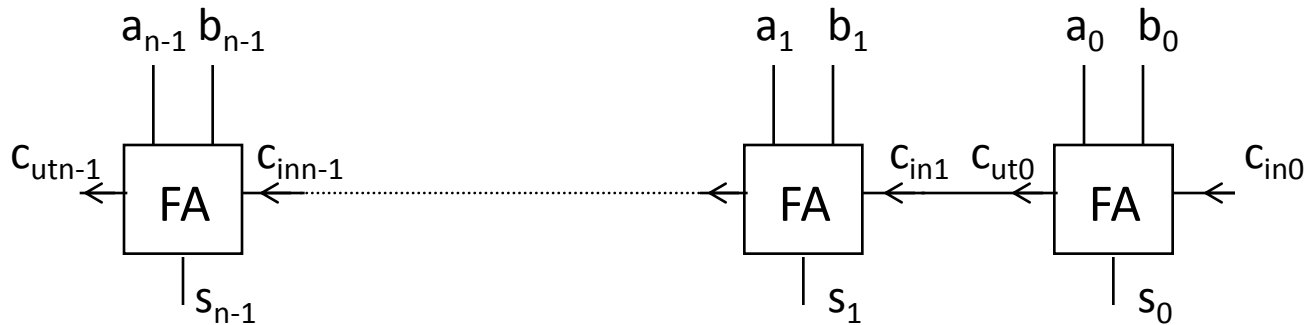


- Generate-statement kopplar ihop många likadana element

```
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!

- För att kunna testa 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 testbänksprogram kan outtröttligt prova igenom alla insignalskombinationer – det orkar inte Du!

Testbänk

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);
```

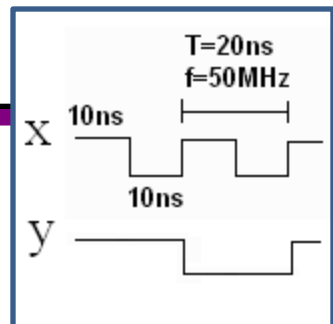
```
  U2:xor_gate PORT MAP (x,y,ut2);
```

```
  U3:xor_gate PORT MAP (x,y,ut3);
```

```
END example;
```

Den krets som testas används som en komponent av testbänksprogrammet

Här genereras testsignalerna



Testbänk



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.

