KTH Informations- och kommunikationsteknik

IE1204 Digital Design

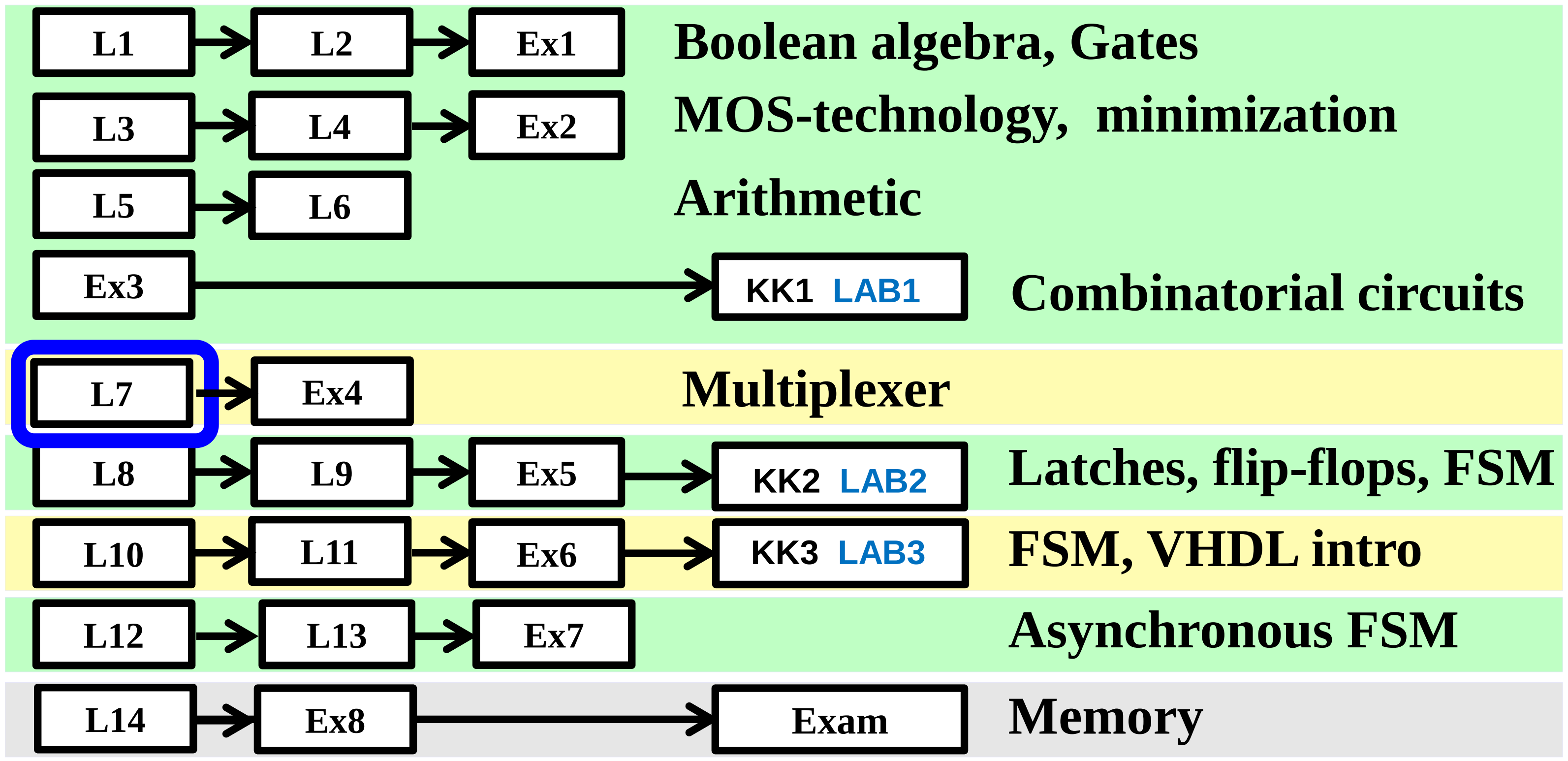
L7: Combinational circuits, Introduction to VHDL

Masoumeh (Azin) Ebrahimi

KTH/ICT

mebr@kth.se

IE1204 Digital Design

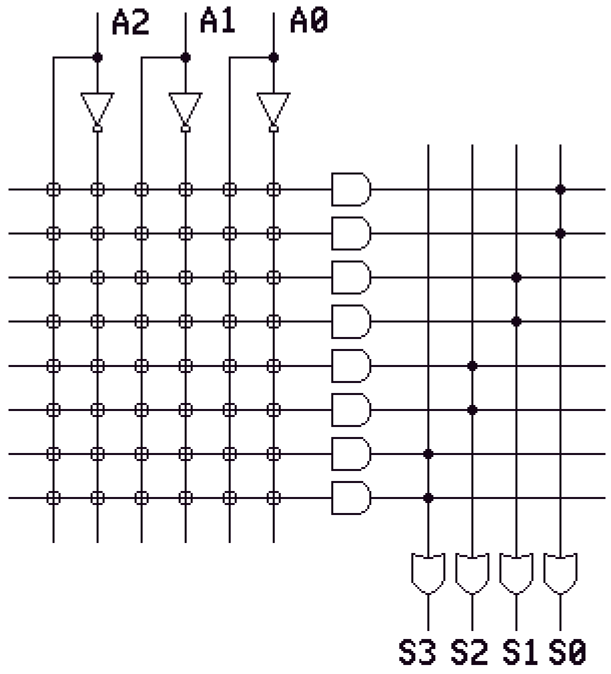


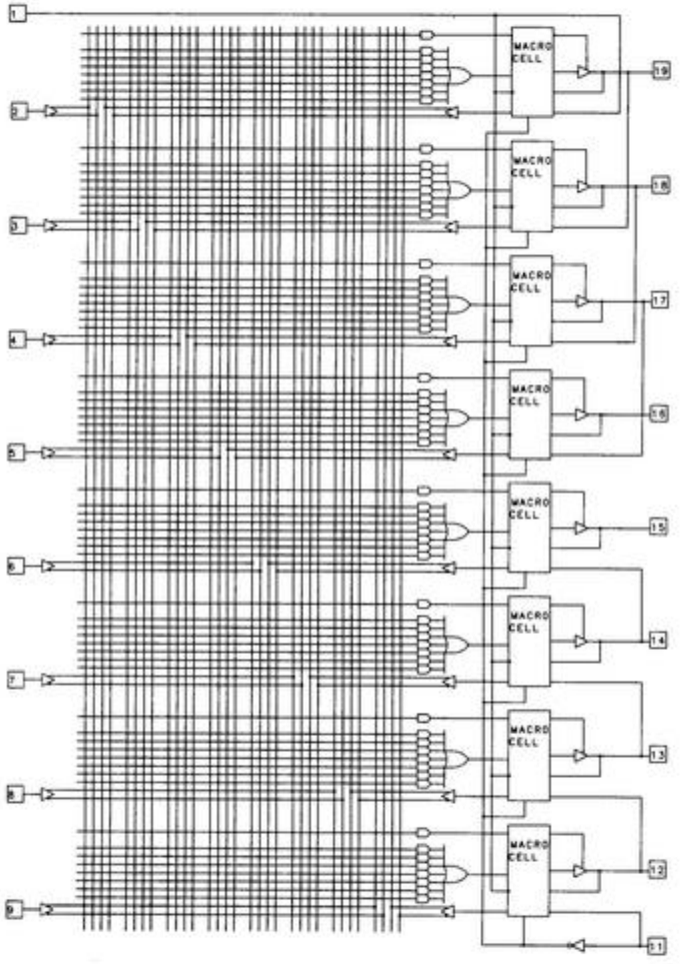
This lecture

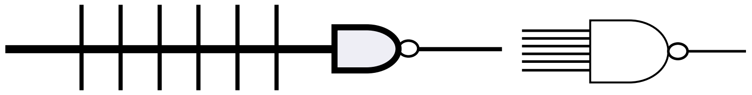
- BV 318-339, 60-65, 280-291,341-365

PLD (eg. PAL)

Technology: AND-OR array





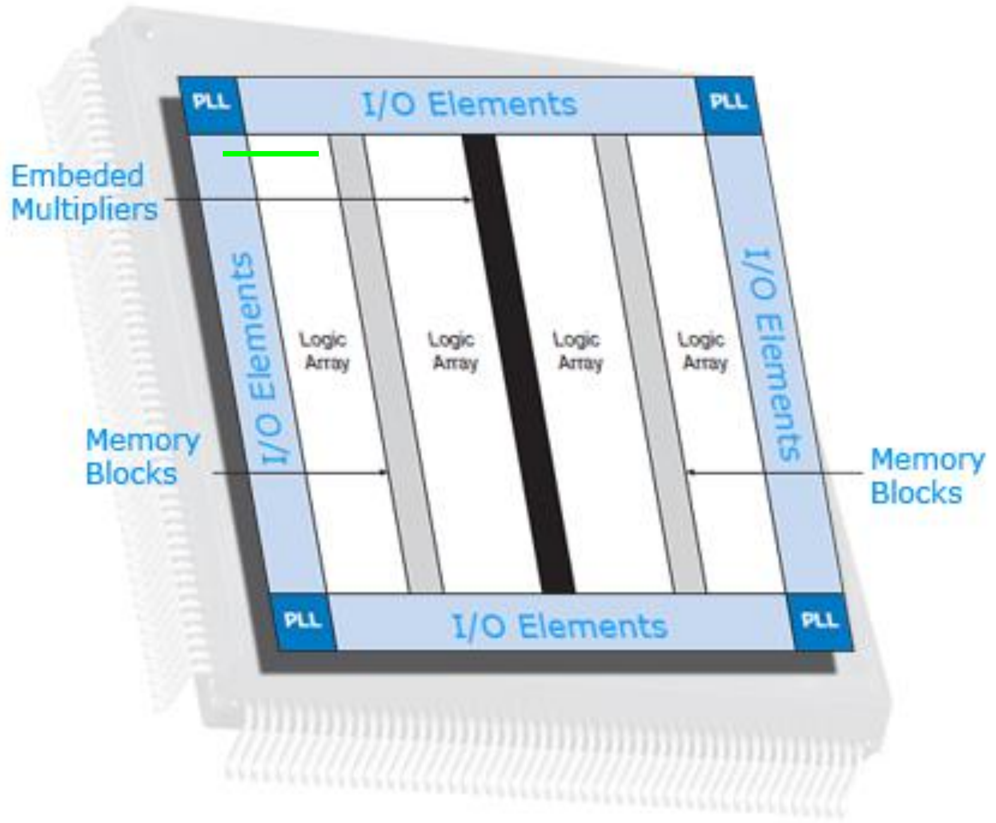


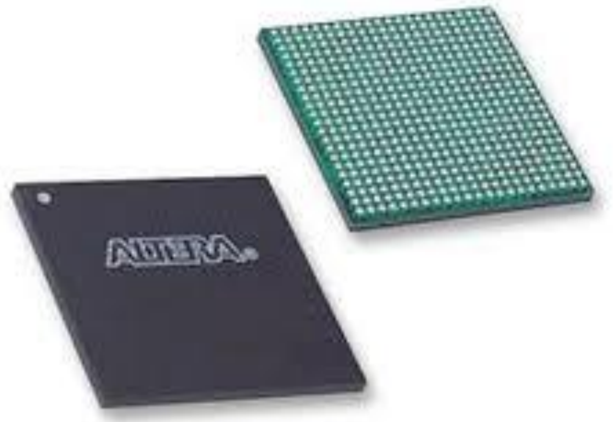
Fan-in issue

Large programmable circuits

Therefore in order to be able to build large programmable circuits in CMOS technology there is a need for other techniques that are not based on gates with many inputs!

FPGA (eg. Cyclone II)



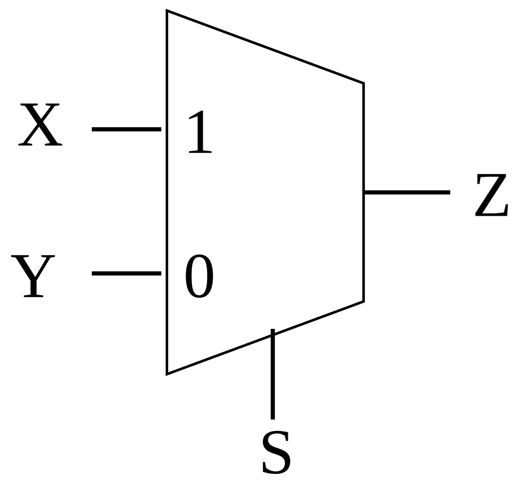


Typically 50000 logic elements

Technology : MUX tree

The multiplexer (MUX)

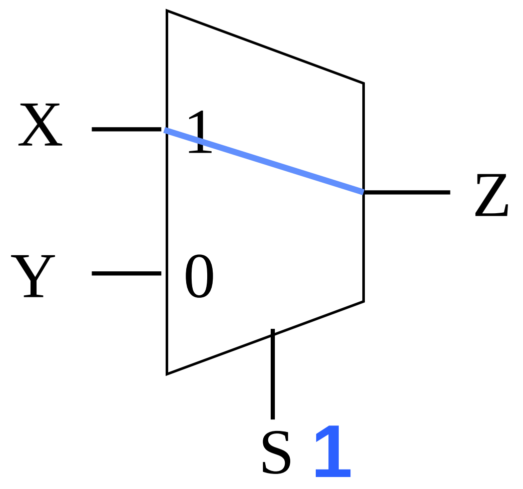
- The multiplexer can select which input you are going to connect to the output
- ”If S then X, else Y”



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

The multiplexer (MUX)

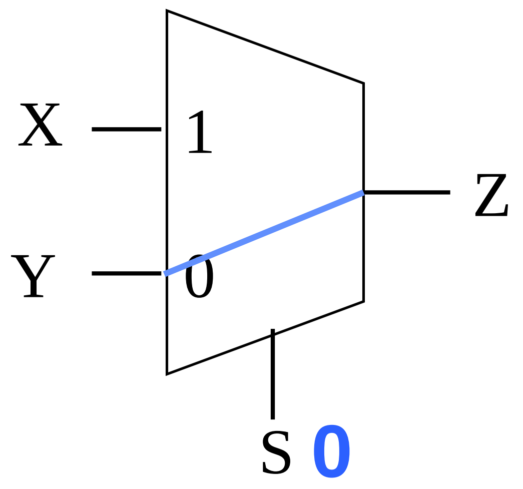
- The multiplexer can select which input you are going to connect to the output
- ”If S then X, else Y”



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

The multiplexer (MUX)

- The multiplexer can select which input you are going to connect to the output
- ”If S then X, else Y”

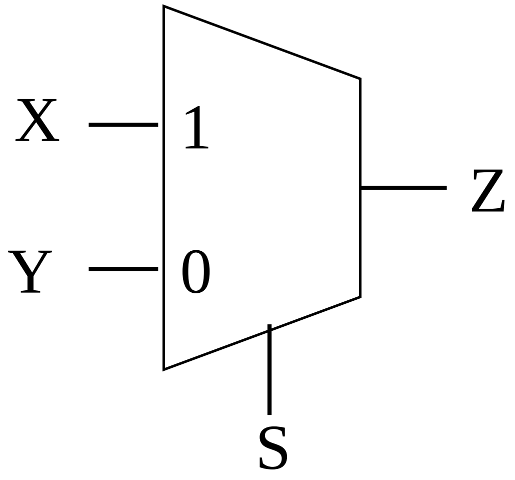


$$Z = SX + \overline{S}Y$$

Implementation of functions using MUXes

How can the following functions be implemented with a 2:1 multiplexer?

- $Z = \overline{B}$ (INV)
- $Z = AB$ (AND)
- $Z = A + B$ (OR)
- $Z = A \oplus B$ (XOR)

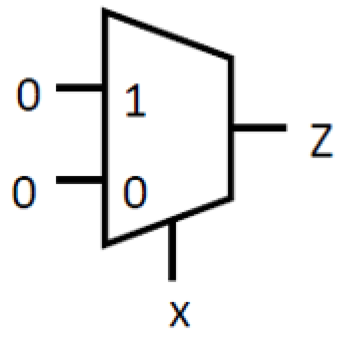


$$Z = SX + \overline{S}Y$$

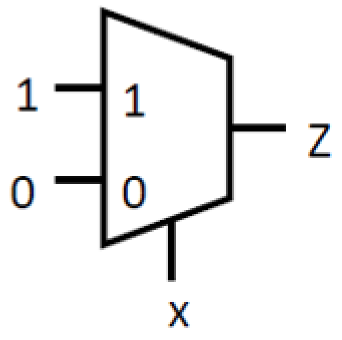
Quickie Question …

- ***How to connect the inputs of the MUX in order to implement an inverter?***

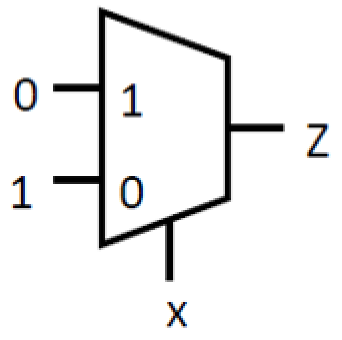
Desired function: $z = \overline{x}$



Alt: A



Alt: B

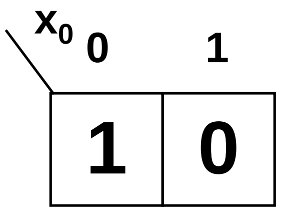


Alt: C

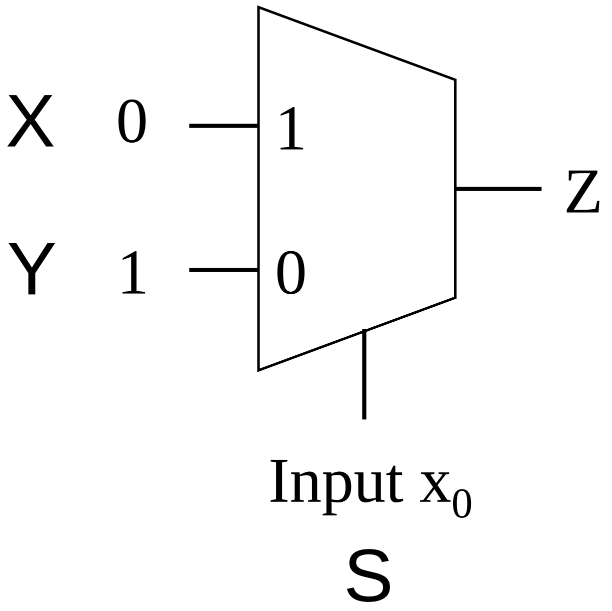
Inverter implemented with a MUX

Specification:

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



NOT



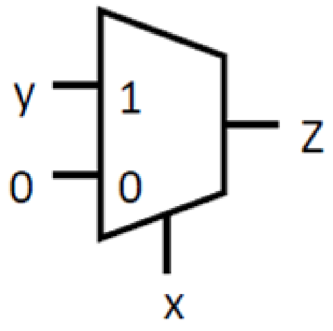
$$Z = S \cdot X + \overline{S} \cdot Y =$$

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

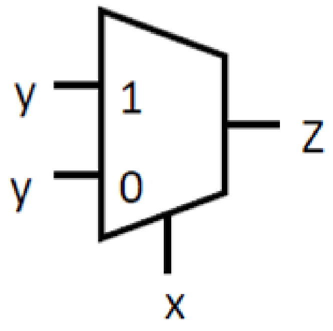
Quickie Question …

- ***How to connect the inputs of the MUX in order to implement an AND gate?***

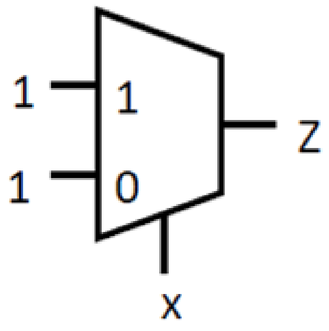
Desired function: z = xy



Alt: A



Alt: B

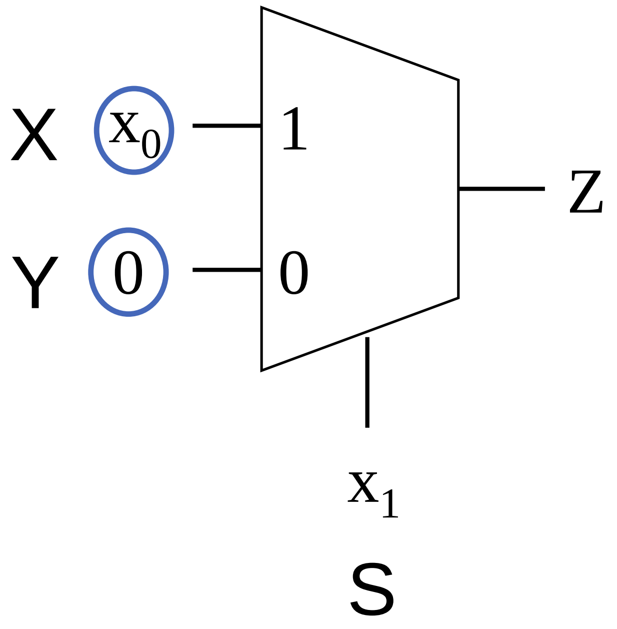


Alt: C

AND implemented with a MUX

- Specification:

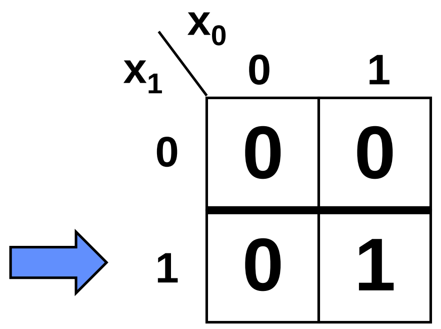
$x_1 \backslash x_0$	0	1
0	0	0
1	0	1

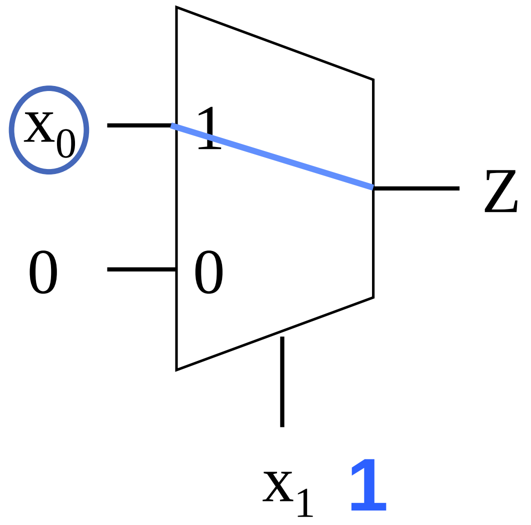


$$Z = SX + \bar{S}Y = x_1 \cdot \boxed{x_0} + \bar{x}_1 \cdot \boxed{0} = x_1 \cdot x_0$$

AND implemented with a MUX

- Specification:

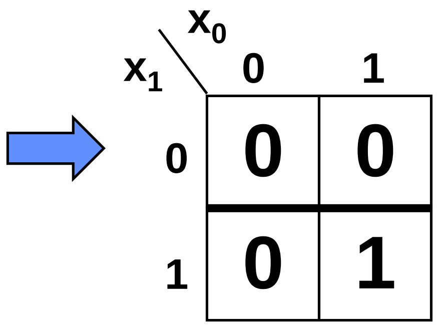




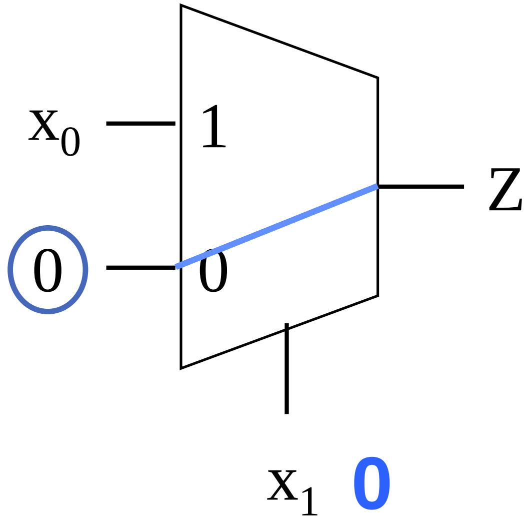
$$Z = SX + \bar{S}Y = x_1 \cdot \boxed{x_0} + \bar{x}_1 \cdot 0 = x_1 \cdot x_0$$

AND implemented with a MUX

- Specification:



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

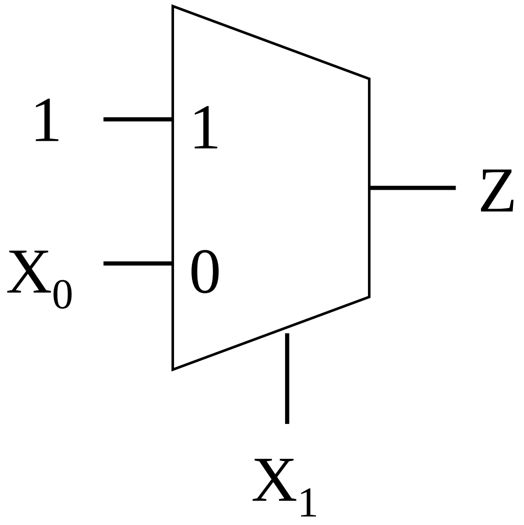


$$Z = SX + \bar{S}Y = x_1 \cdot x_0 + \bar{x}_1 \cdot \boxed{0} = x_1 \cdot x_0$$

OR implemented with a Mux

- Specification:

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



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

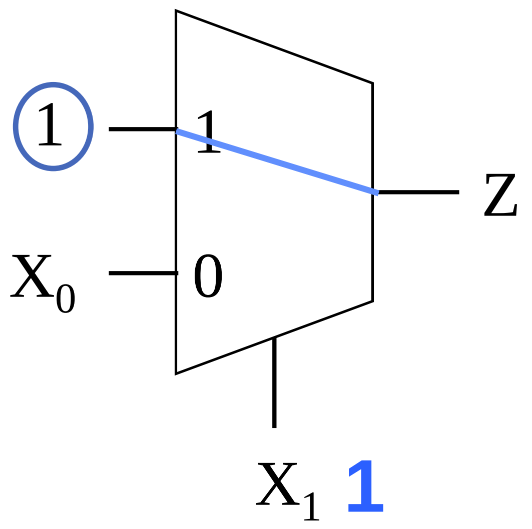
$$= \{SX + \bar{S}Y\} = x_1(x_0 + \bar{x}_0) + \bar{x}_1 \cdot x_0 =$$

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

OR implemented with a Mux

- Specification:

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



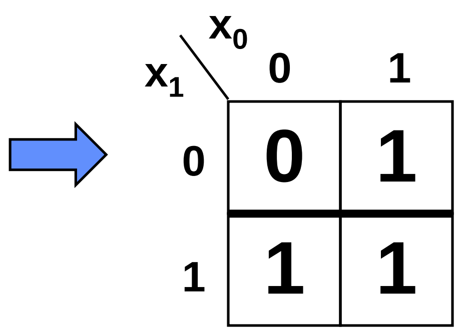
$$Z = x_1x_0 + \bar{x}_1x_0 + x_1\bar{x}_0 =$$

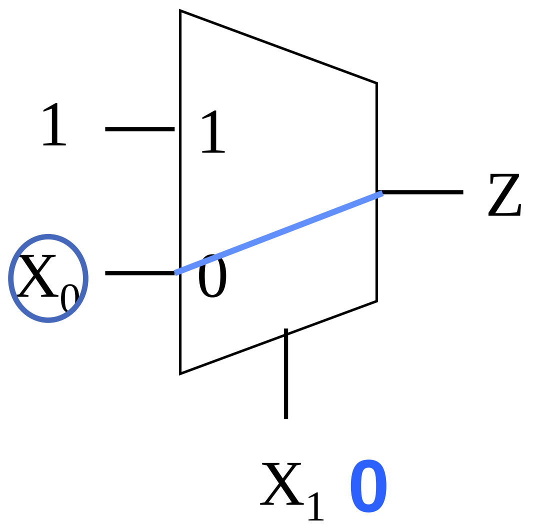
$$= \{SX + \bar{S}Y\} = x_1(x_0 + \bar{x}_0) + \bar{x}_1 \cdot x_0 =$$

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

OR implemented with a Mux

- Specification:





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

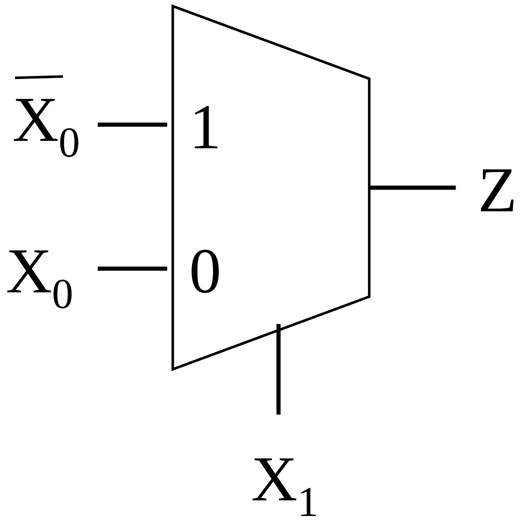
$$= \{SX + \bar{S}Y\} = x_1(x_0 + \bar{x}_0) + \bar{x}_1 \cdot x_0 =$$

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

XOR implemented with a Mux

- Specification:

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



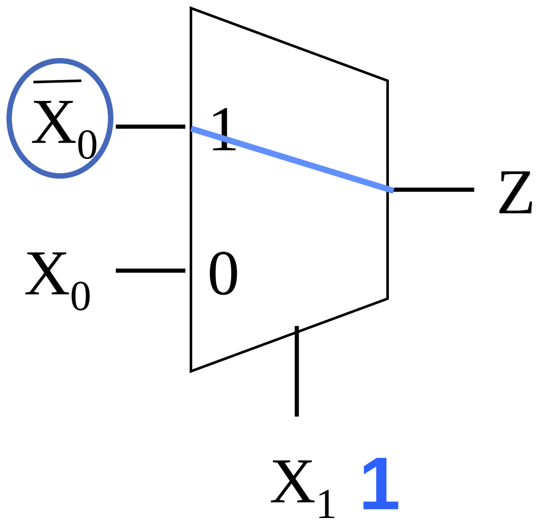
$$Z = SX + \overline{S}Y =$$

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

XOR implemented with a Mux

- Specification:

x_1 \ x_0	0	1
0	0	1
1	1	0



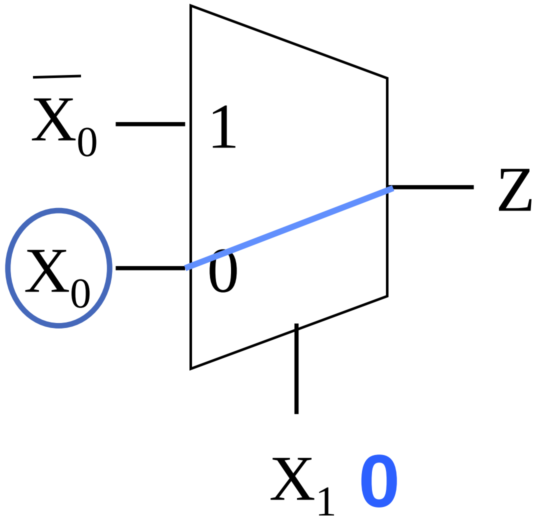
$$Z = SX + \bar{S}Y =$$

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

XOR implemented with a Mux

- Specification:

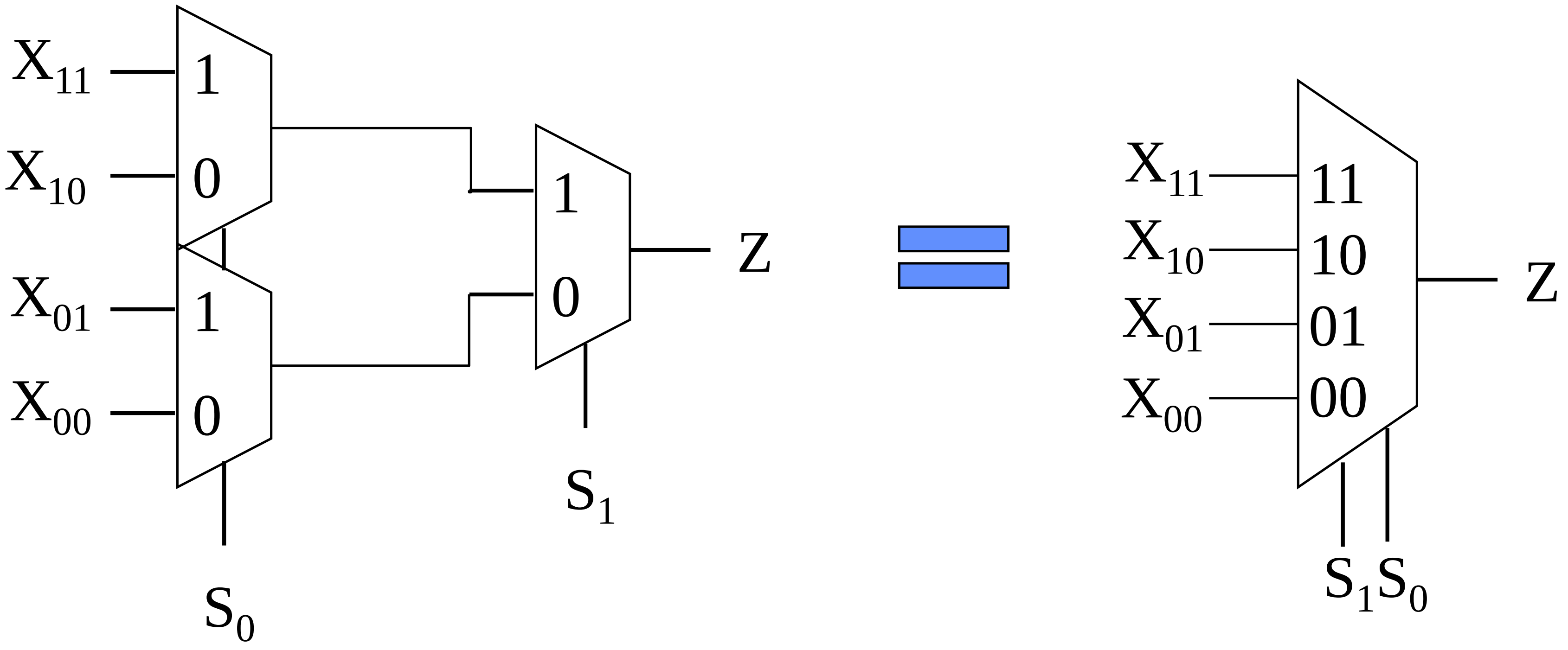
$x_1 \backslash x_0$	0	1
0	0	1
1	1	0



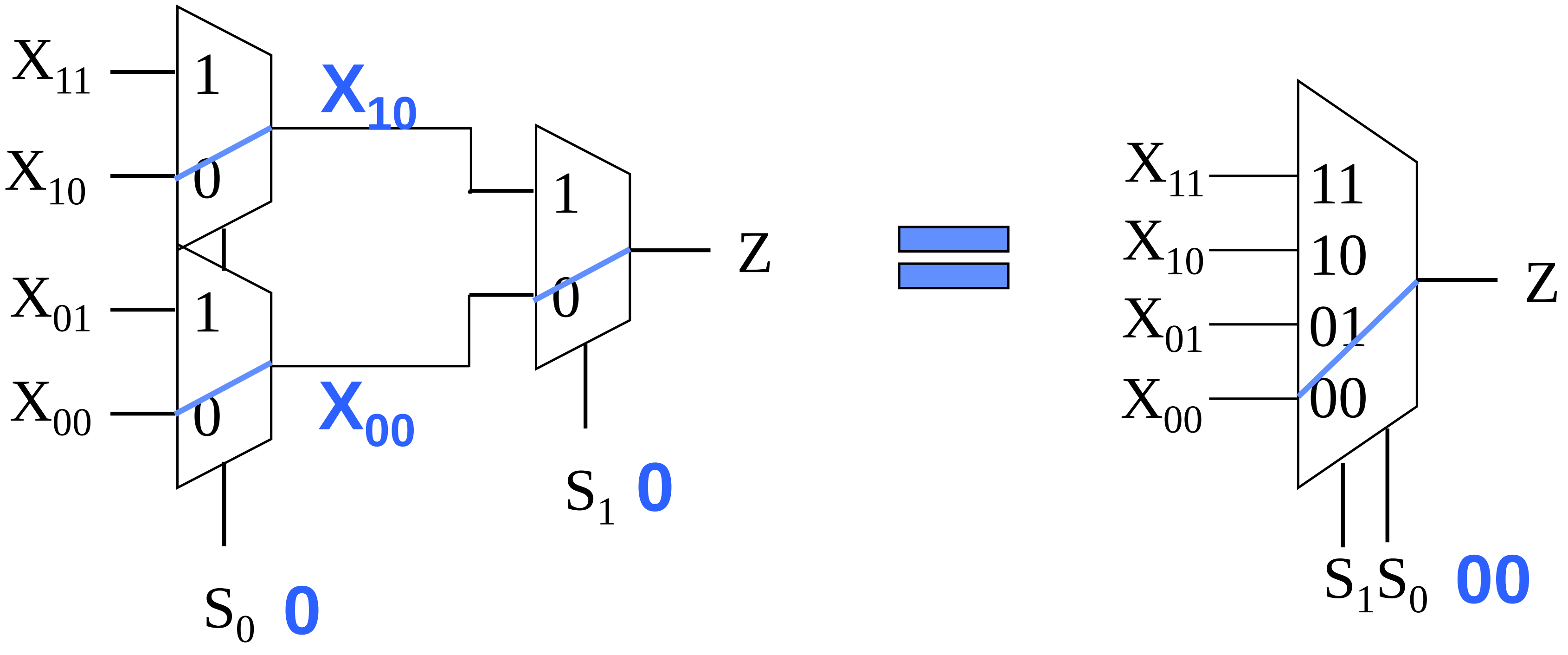
$$Z = SX + \overline{S}Y =$$

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

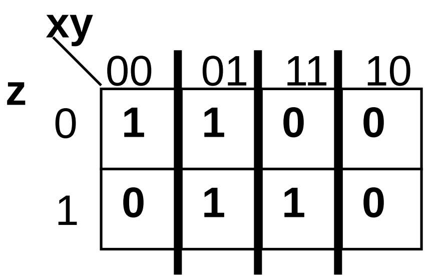
Hierarchy of MUXes



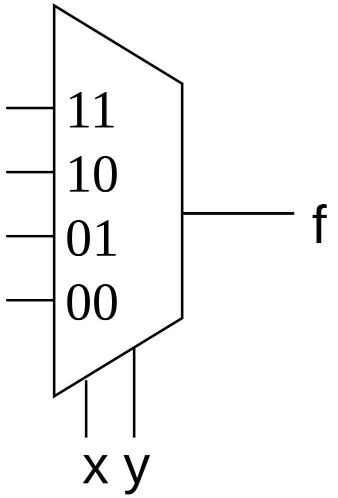
Hierarchy of MUXes



Implementation of larger functions with MUXes

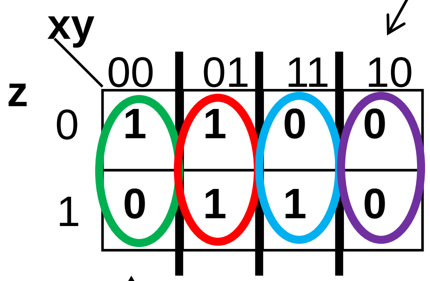


$$f = \bar{z}\,\bar{x} + \bar{x}y + zy$$

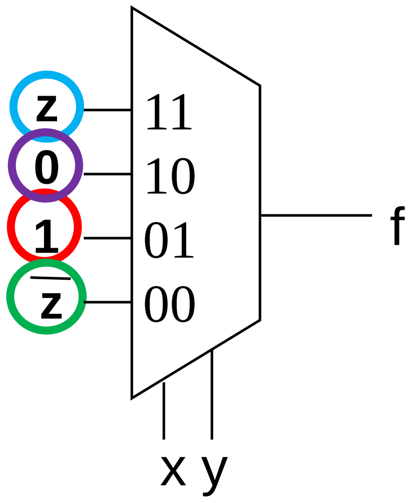


Implementation of larger functions with MUXes

Choose any of the inputs as address inputs ...



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



... And minimize/implement function for each input.
Draw new Karnaugh diagrams if necessary.

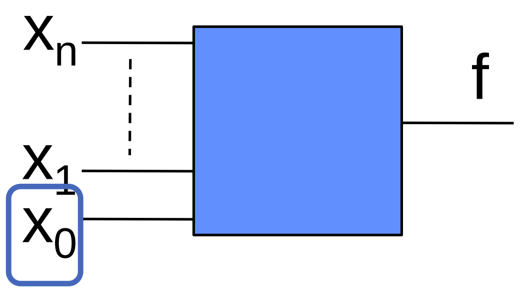
An (n + 1)-input function can be implemented with a MUX that has n select inputs!

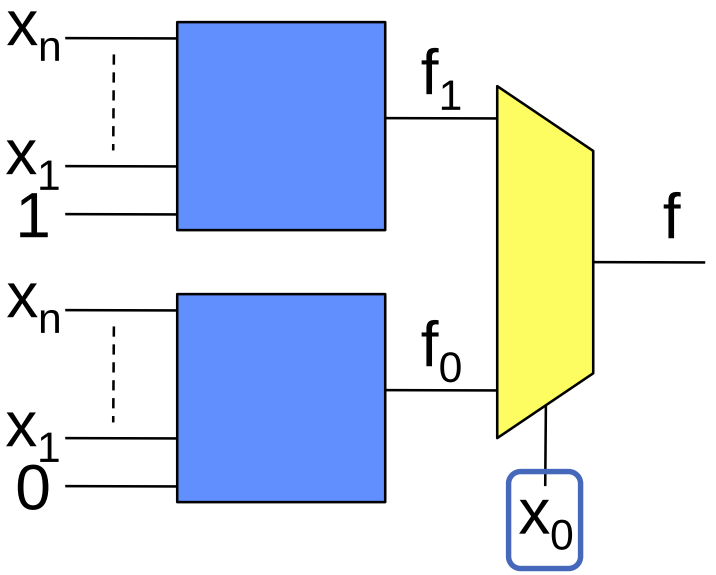
Mapping into MUXes: Shannon decomposition (BV 6.1.2)

- Any Boolean function f $(x_n, \ldots, x_1, x_0)$ can be partitioned as

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

- The function can then be implemented with a multiplexer



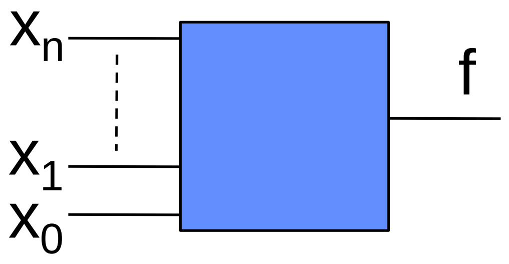


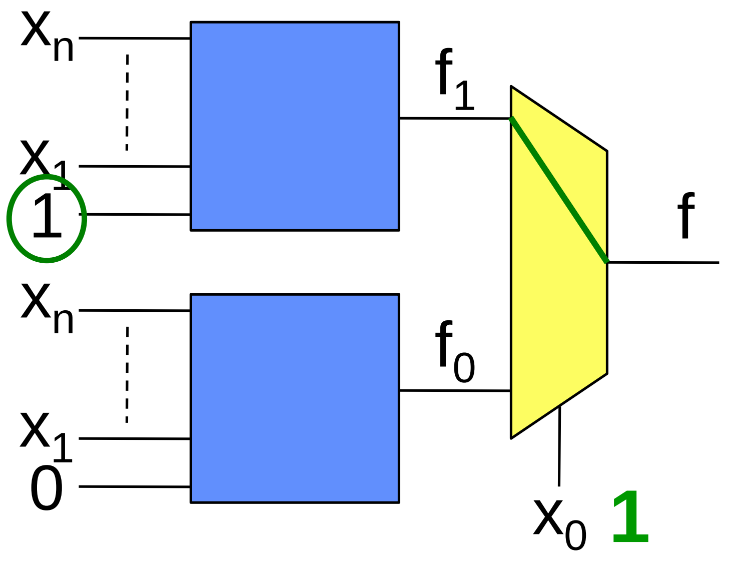
Mapping into MUXes: Shannon decomposition (BV 6.1.2)

- Any Boolean function f $(x_n, \ldots, x_1, x_0)$ can be partitioned as

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

- The function can then be implemented with a multiplexer



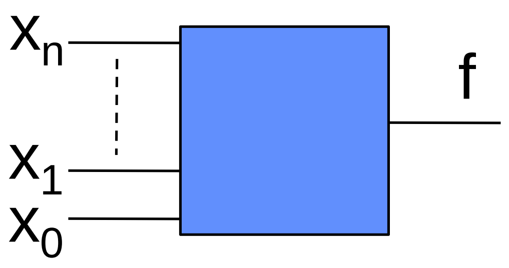


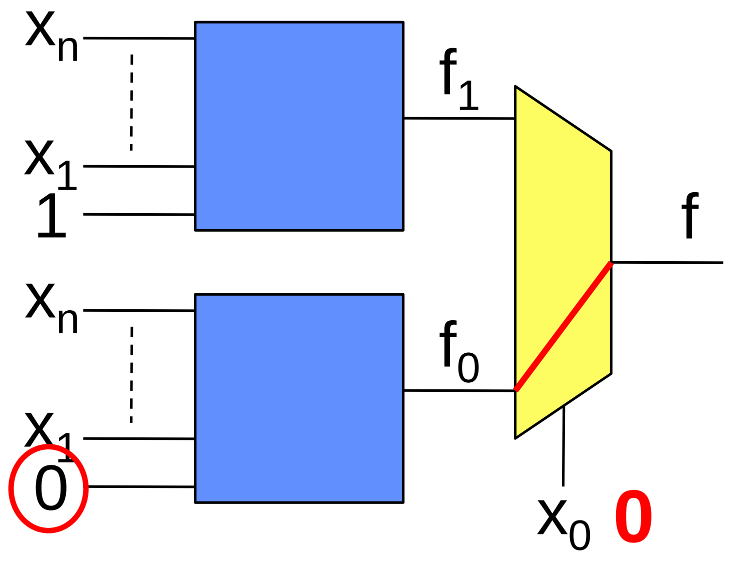
Mapping into MUXes: Shannon decomposition (BV 6.1.2)

- Any Boolean function f $(x_n, \ldots, x_1, x_0)$ can be partitioned as

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

- The function can then be implemented with a multiplexer

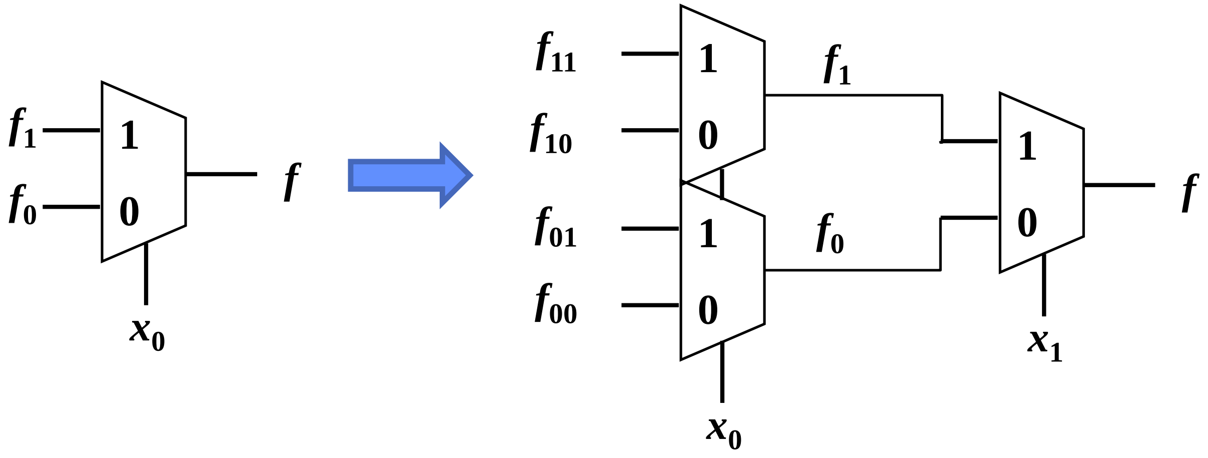




Mapping to MUXes: Shannon decomposition

- Any Boolean function f $(x_n, \ldots, x_1, x_0)$ can be decomposed (recursively) as

$$f(x_n,\ldots,x_1,x_0) = x_0\, f_1(x_n, \ldots, x_1,1) + \overline{x}_0\, f_0(x_n, \ldots, x_1,0)$$
$$= x_1 x_0\, f_{11}(x_n, \ldots, x_2,1,1) + x_1 \overline{x}_0\, f_{10}(x_n, \ldots, x_2,1,0)$$
$$+ \overline{x}_1 x_0\, f_{01}(x_n, \ldots, x_2,0,1) + \overline{x}_1 \overline{x}_0\, f_{00}(x_n, \ldots, x_2,0,0)$$



Proof

$$\overbrace{f(x_n,\ldots,x_1,x_0)}^{\text{Left-hand side}} = \overbrace{\underbrace{x_0 \cdot f(x_n,\ldots,x_1,1)}_{\text{Left term}} + \underbrace{\overline{x}_0 \cdot f(x_n,\ldots,x_1,0)}_{\text{Right term}}}^{\text{Right-hand side}}$$

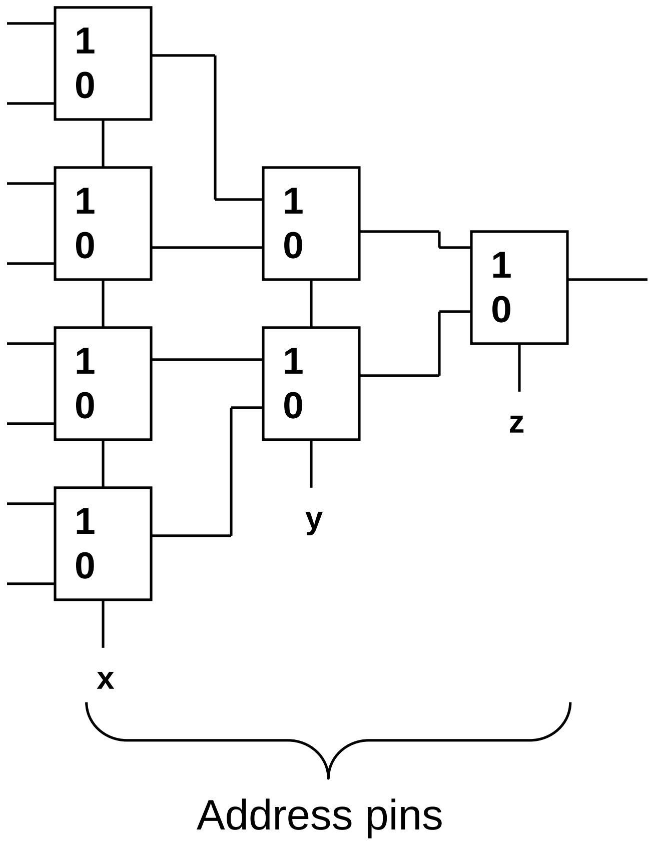
- Right-hand side:
  - If $x_0 = 1$ then the right term is zero. Then f is equal to the left term.
  - If $x_0 = 0$, the left term is zero. Then f is equal to the right term.

- Left-hand side:
  - if $x_0 = 1$, then f is equal to f $(x_n, \ldots, x_1,1)$ (= left term on the right-hand side)
  - if $x_0 = 0$ then f is equal to f $(x_n, \ldots, x_1,0)$ (= right term in the right-hand side)

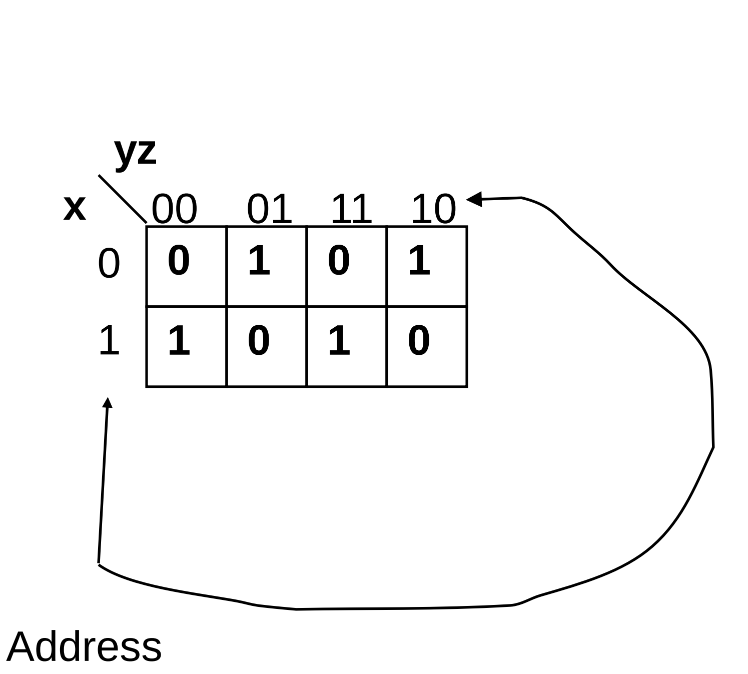
- Left-hand side = Right-hand side

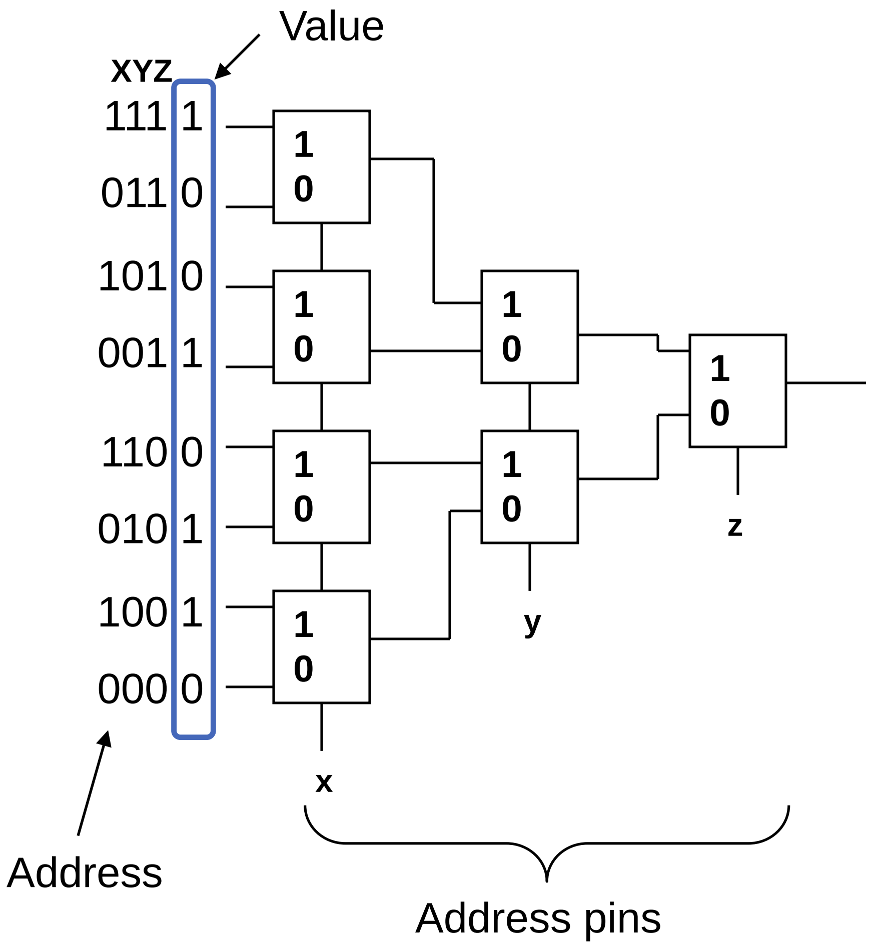
Mux circuits

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



Mux circuits

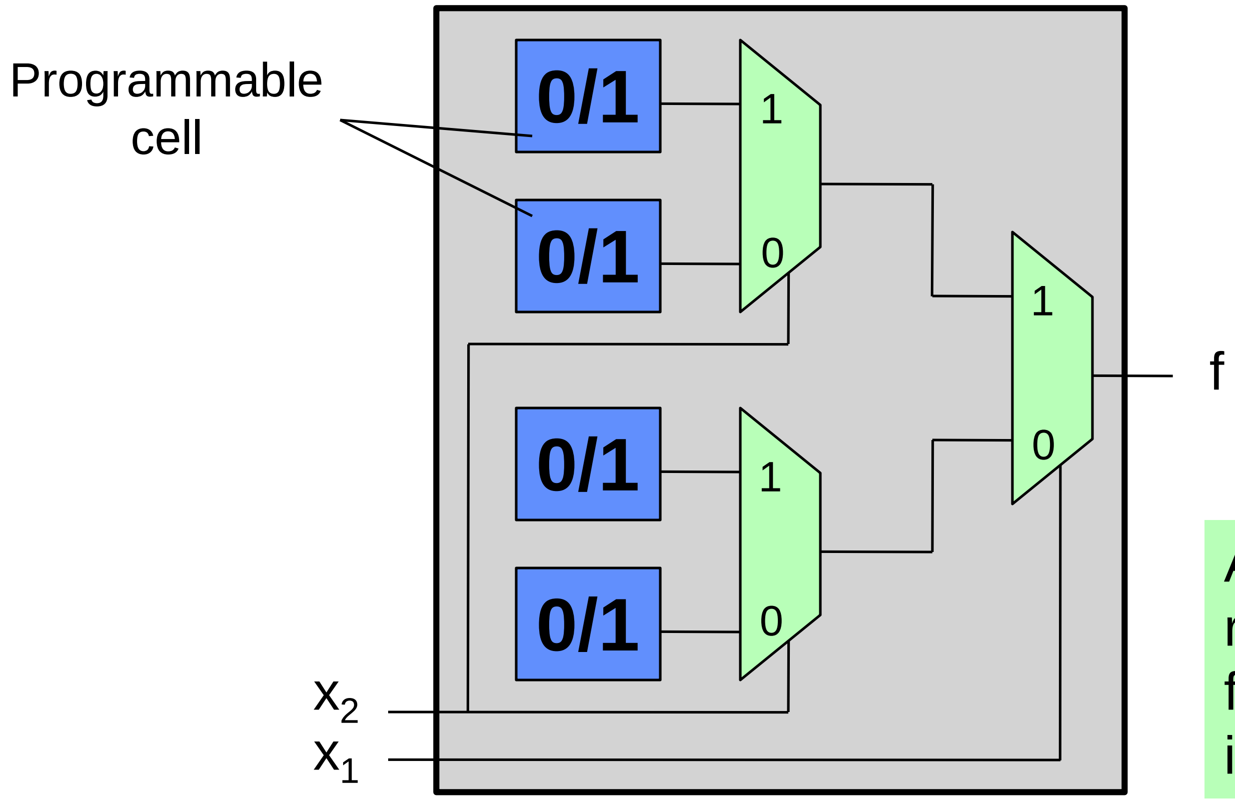




But this is a memory

(ROM, RAM ...)

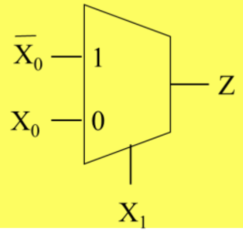
Look-up tables (LUT)



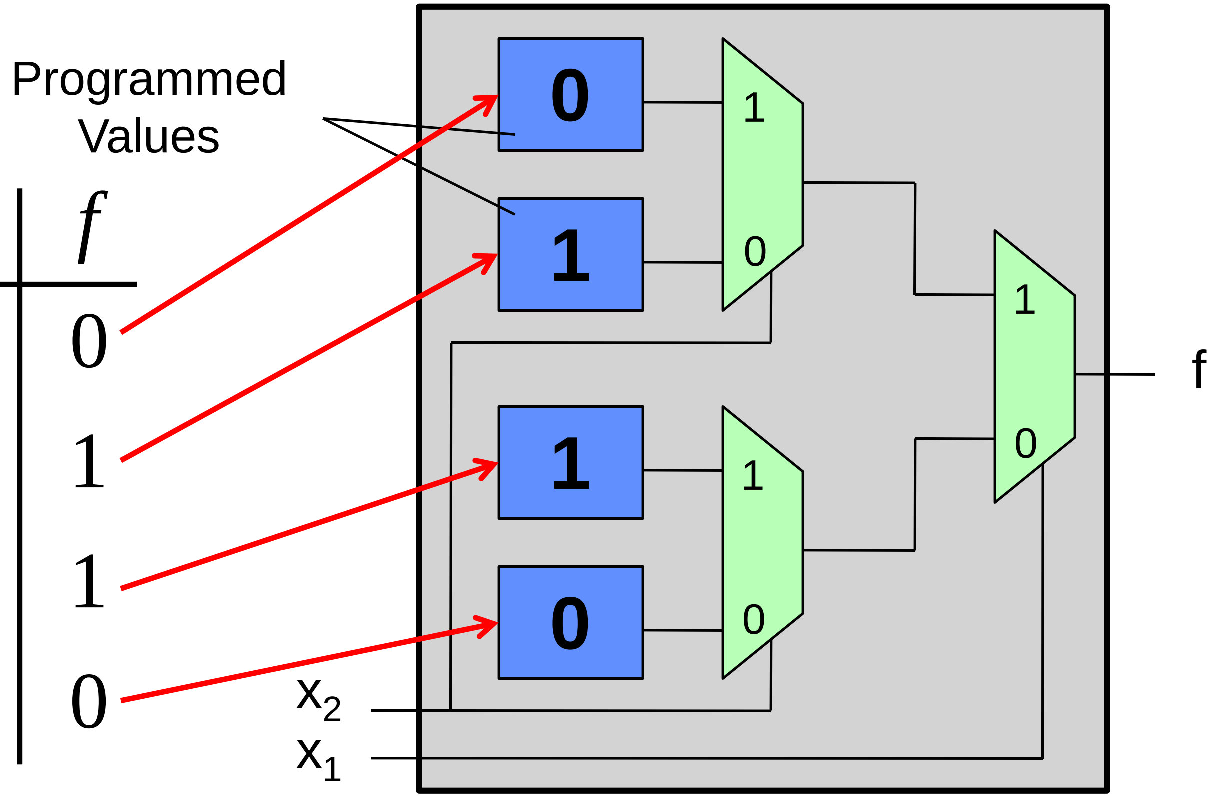
Two-input LUT

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

Example: XOR gate



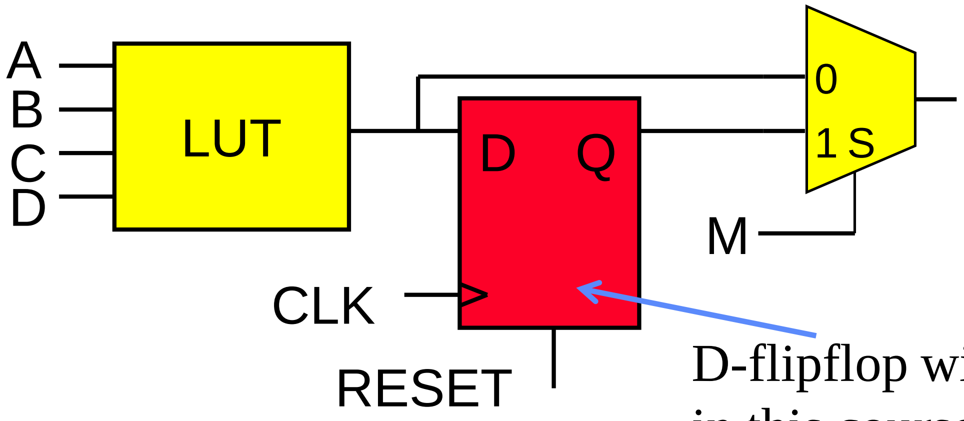
x_1	x_2	f
1	1	0
1	0	1
0	1	1
0	0	0



Two-input LUT

A simple FPGA cell

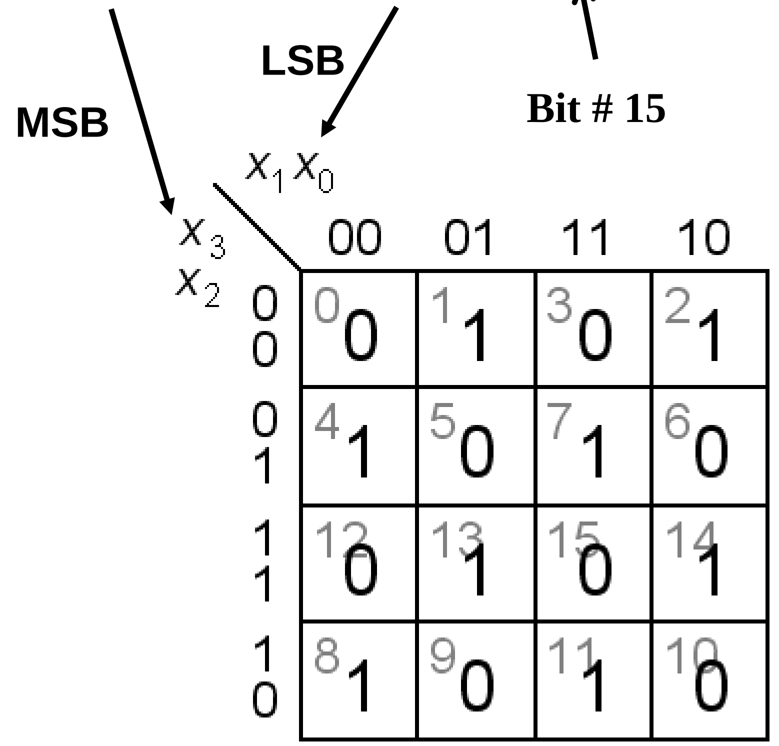
- The simplest FPGA cell consists of a single table (e.g. Look-Up-Table - LUT), a D flip-flop and a bypass MUX.



D-flipflop will be explained soon in this course

One way to identify functions...

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



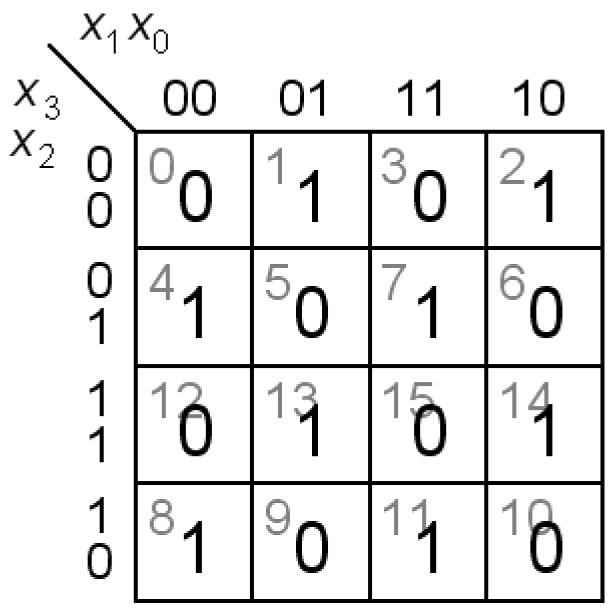
$$\begin{array}{c|cccc}
x_3x_2 \backslash x_1x_0 & 00 & 01 & 11 & 10 \\
\hline
00 & {}^{0}0 & {}^{1}1 & {}^{3}0 & {}^{2}1 \\
01 & {}^{4}1 & {}^{5}0 & {}^{7}1 & {}^{6}0 \\
11 & {}^{12}0 & {}^{13}1 & {}^{15}0 & {}^{14}1 \\
10 & {}^{8}1 & {}^{9}0 & {}^{11}1 & {}^{10}0
\end{array}$$

The functions that are stored in a LUT are usually numbered after the number that is made up of the 1's in the truth table / Karnaugh map.

n inputs => $2^{(2^n)}$ possible different Boolean functions

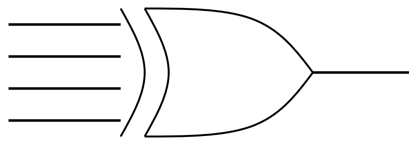
LUT function number

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



$x_3x_2 \backslash x_1x_0$	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

Odd parity

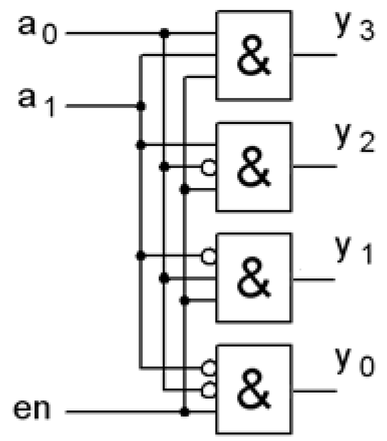


$$f_{6996} = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

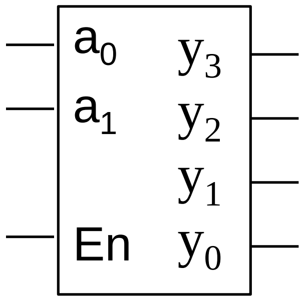
With a LUT, all functions are realized in the same way, so all of them have the same cost

Decoder

- Mostly used as address decoders
- Only one output is active when the 'enable' (En) signal is active
- The active output is selected by a_1a_0



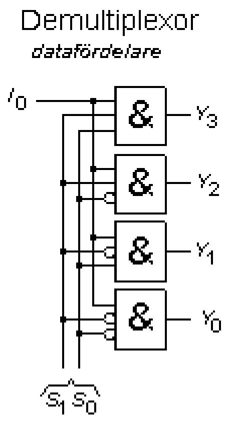
En	a_1	a_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	-	-	0	0	0	0



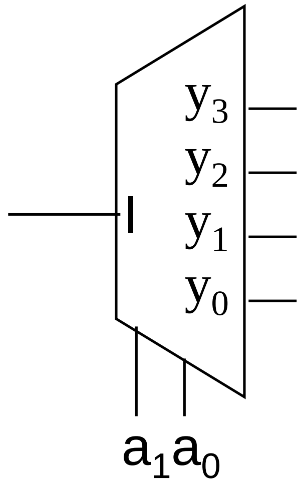
2-to-4 decoder

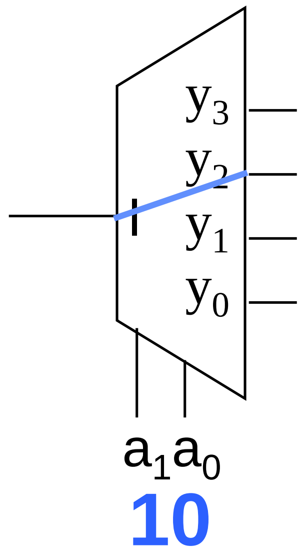
Demultiplexer

- The demultiplexer has basically the same function as the decoder, but it is drawn differently
- The input I is connected to a selected output

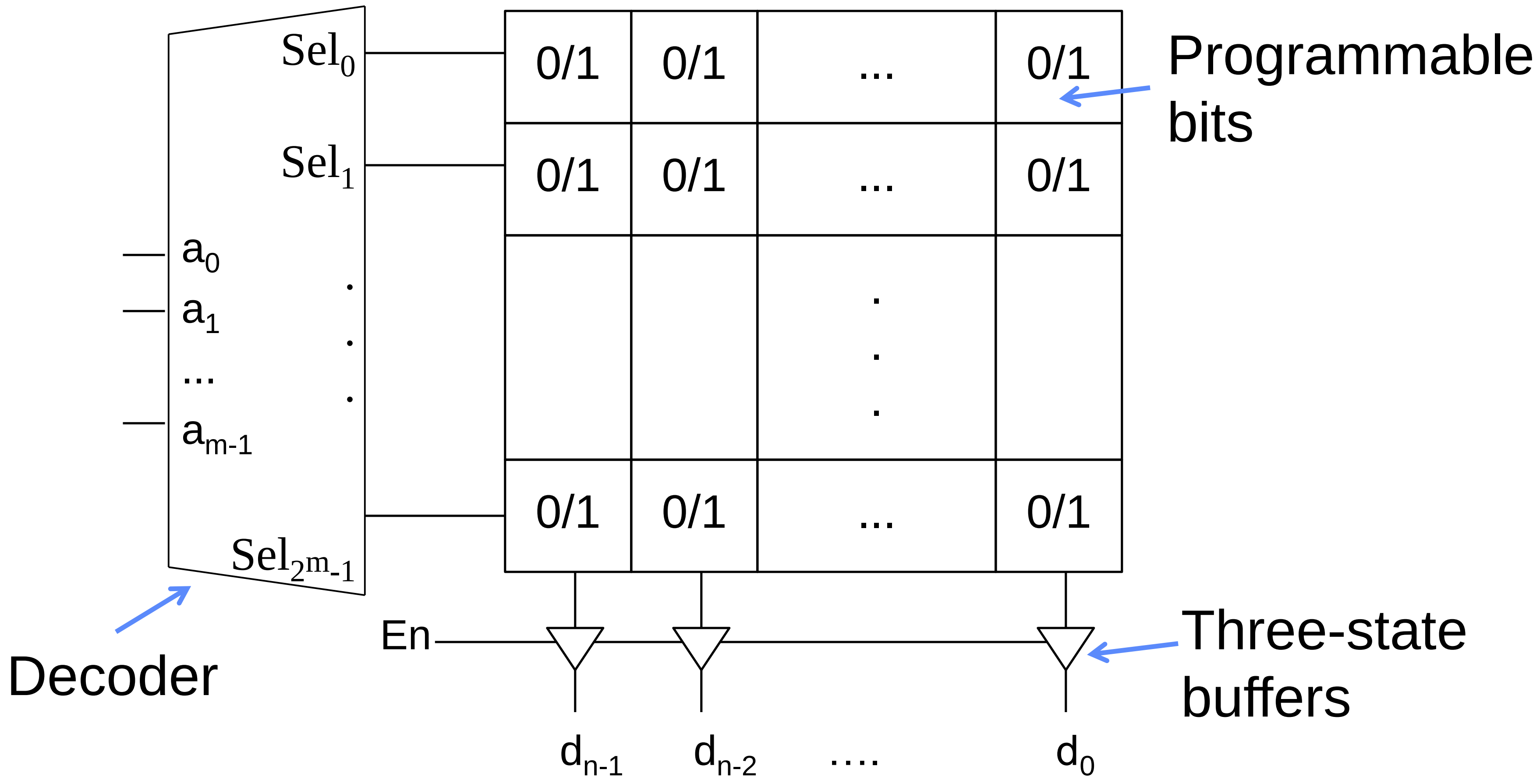


I	a_1	a_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	-	-	0	0	0	0





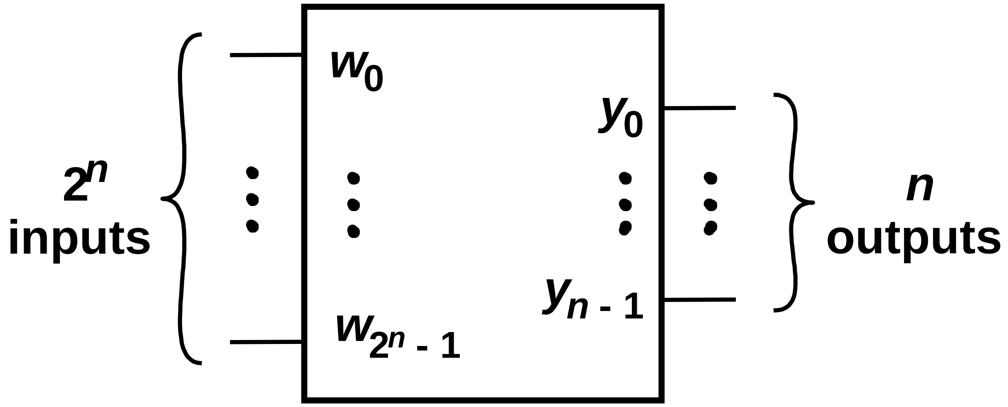
Read-Only Memory



Encoders

- Encoder has the opposite function to a decoder, i.e. it translates 2^N bit input into an N-bit code.
  - The information is greatly reduced

w_0	w_1	w_2	w_3	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1



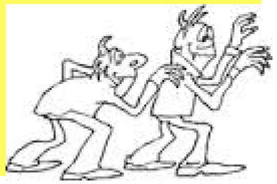
Priority Encoder

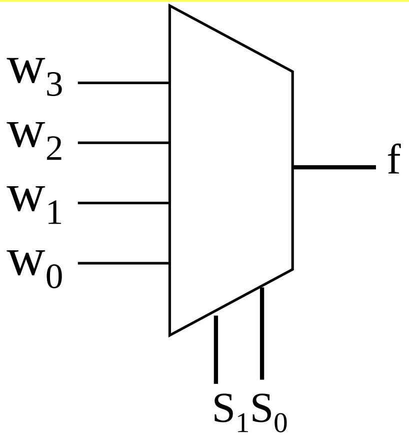
- A Priority Encoder gives back the address of the input with the lowest (or highest) indices that are set to 1 (or 0 depending on what you are looking for)
- If all inputs are 0, the output z = 0, else z = 1

w_0	w_1	w_2	w_3	z	y_1	y_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	-	-

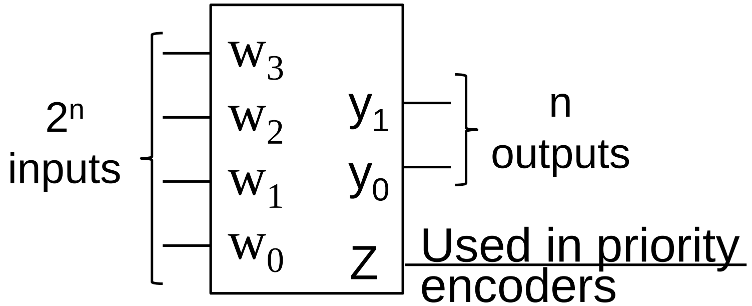
The output is well-defined even if several inputs are active at the same time.

Overview

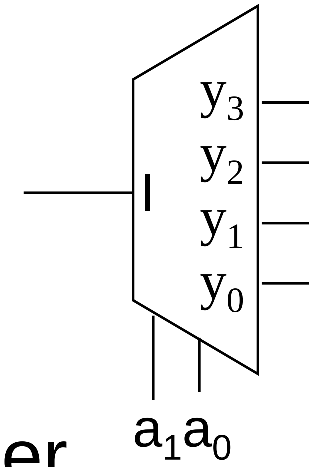




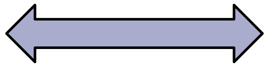
Multiplexer

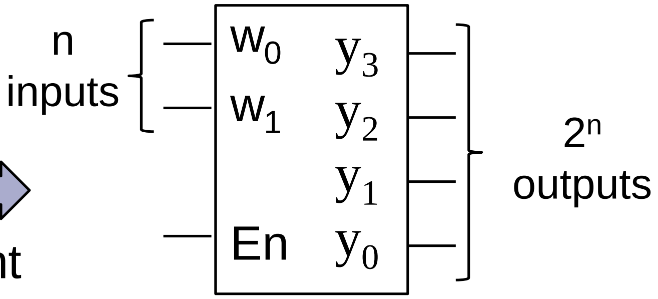


Encoder



Demultiplexer

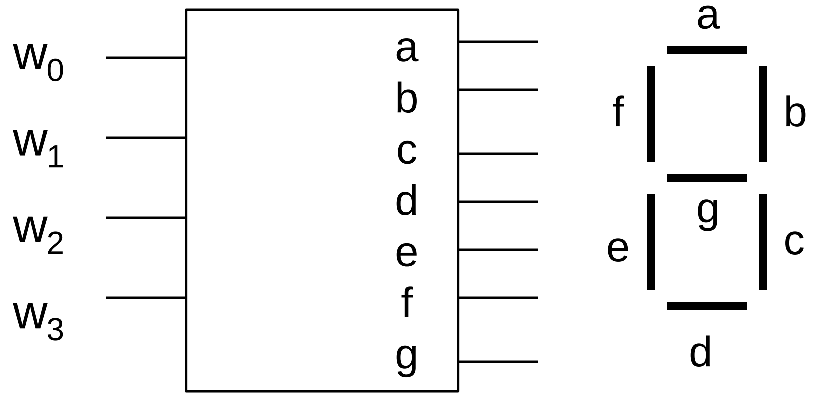




Decoder

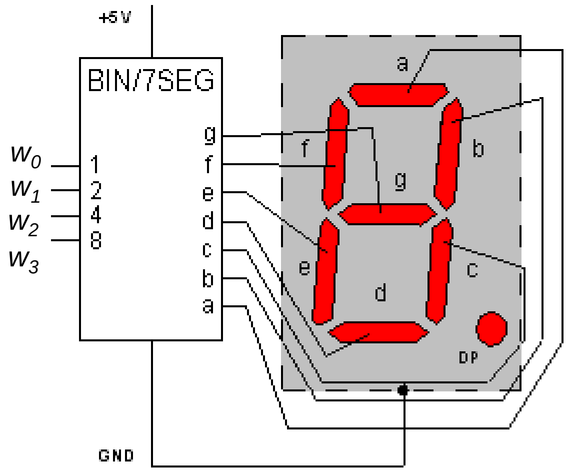
Code converters

- A code converter translates from one code to another. Typical examples are:
  - Binary to BCD (Binary-Coded Decimal)
  - Binary to Gray code
  - 7-4-2-1 code
  - BCD to seven-segment decoder



A variant of the 7-4-2-1 code is used today to store the bar code

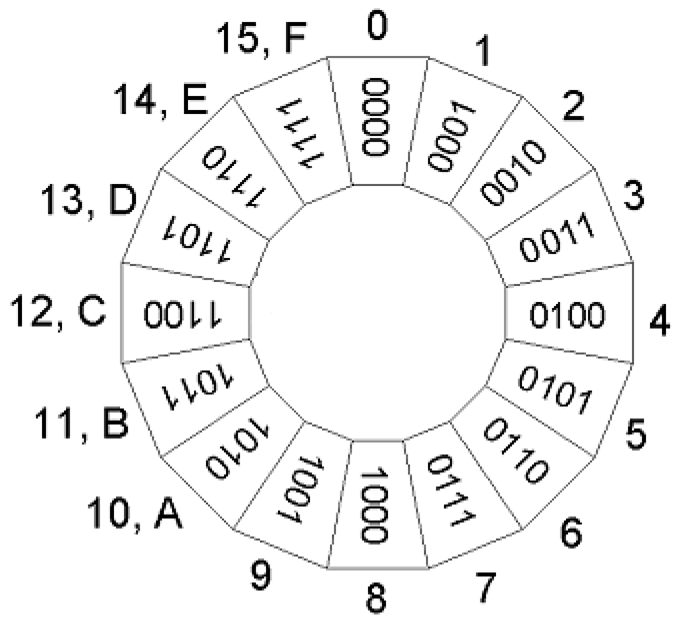
BCD-to-seven segment decoder



w_3	w_2	w_1	w_0	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	1	0	1	1

- BCD-to-7 segment decoder consists of 7 different combinatorial circuits, one for each segment
- To get optimal circuits, all 7 functions have to be minimized simultaneously so that common logic is shared

Disadvantage of binary codes



Binary code, adjacent code words:

1-2 double change
3-4 triple change
5-6 double change
7-8 quadruple change!
9-A double change
B-C quadruple change!
D-E double change
F-0 quadruple change!

Can two bits change at exactly the same time?

- For safe data registration use Gray code
- For data processing use binary code

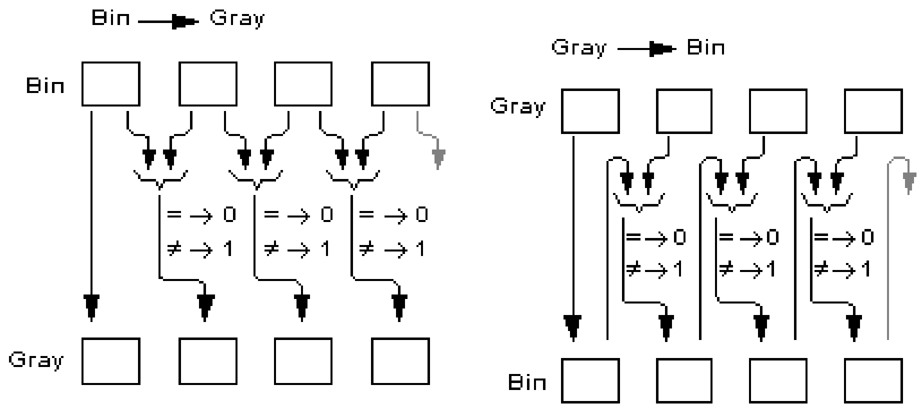
Gray code

- By changing the order of the codewords in a binary code, one construct codes in which no more than one bit is changing at a time
- Such codes are called Gray codes

$$\begin{tikzcd}
\mathbf{0000,} \arrow[r, bend left] & \mathbf{0001,} \arrow[r, bend left] & \mathbf{0011,} \arrow[r, bend left] & \mathbf{0010,} \arrow[r, bend left] & \mathbf{0110,} \arrow[r, bend left] & \mathbf{0111,} \arrow[r, bend left] & \mathbf{0101,} \arrow[r, bend left] & \mathbf{0100}
\end{tikzcd}$$

1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

Conversion between binary and Gray



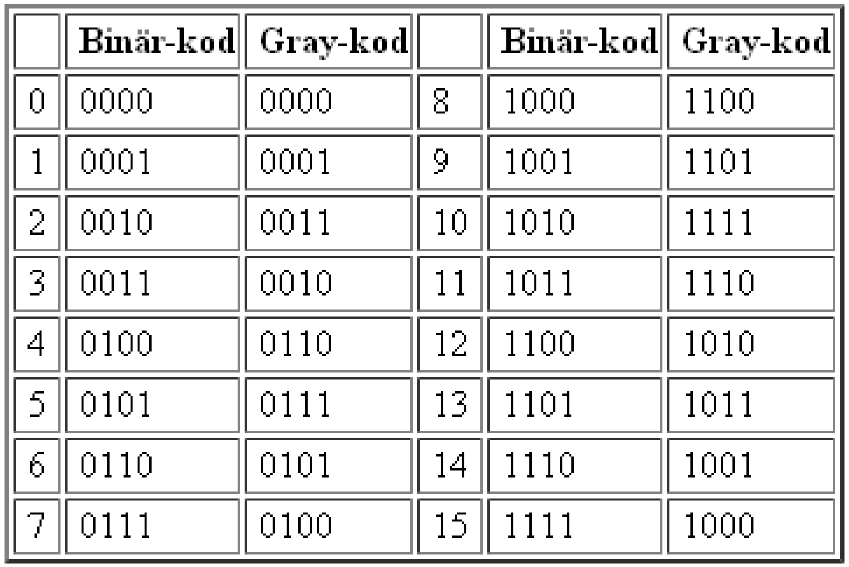
Binary → Gray:
If Binary bit b_n and bit b_{n-1} are *different*,
the Gray code bit g_{n-1} is ”1", else ”0".

Gray → Binary (most common transformation direction):
If Binary bit b_n and Gray code bit g_{n-1} are *different* the Binary bit b_{n-1} is ”1", else ”0".

Logic for Gray to Binary conversion

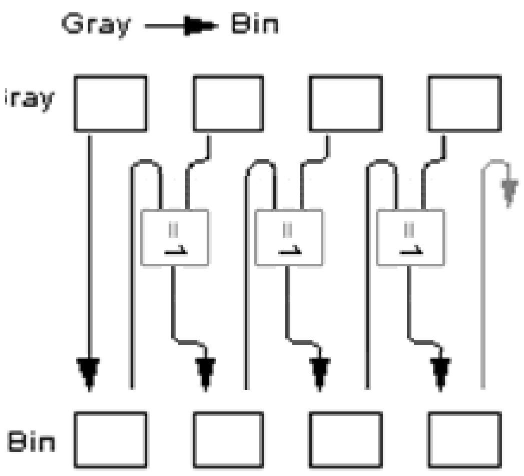
XOR-gate is ”1” if its inputs are *different*!

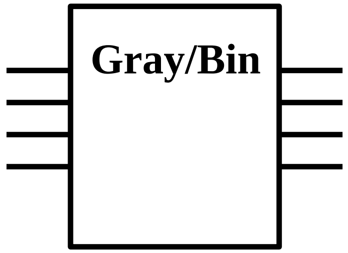
4 bit code converter
Gray code to Binary code



	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.

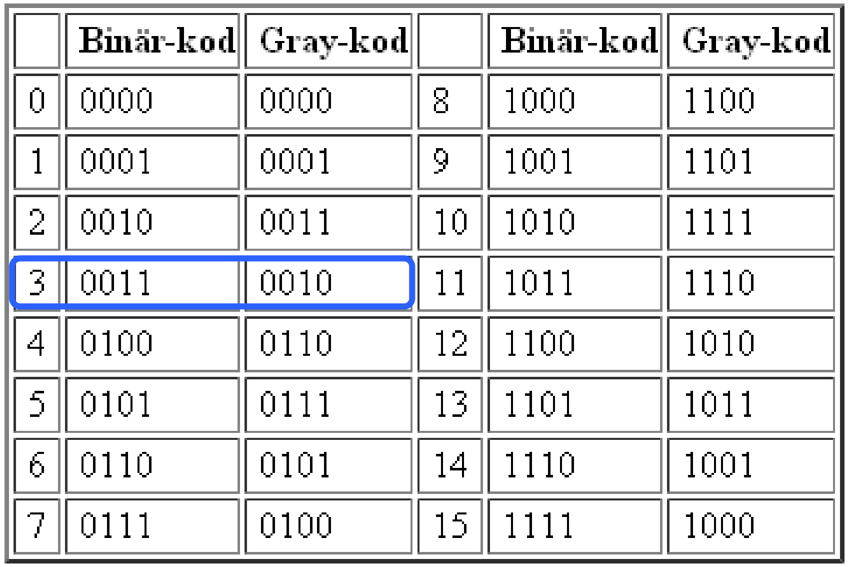




Logic for Gray to Binary conversion

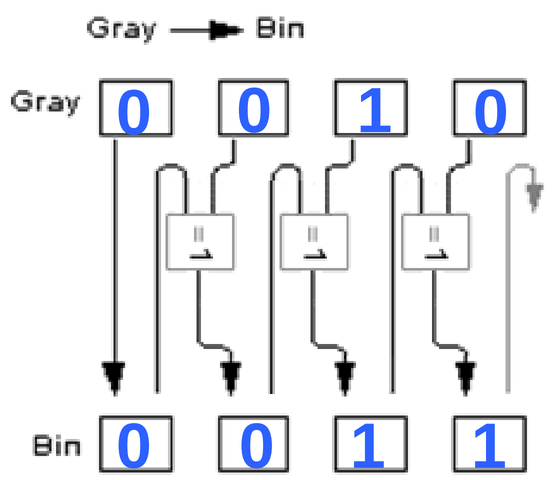
XOR-gate is ”1” if its inputs are *different*!

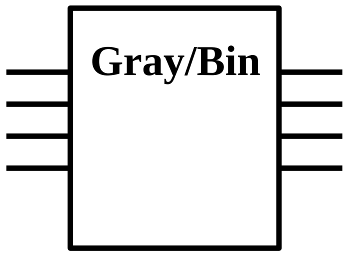
4 bit code converter
Gray code to Binary code



	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.





Introduction to VHDL

- VHDL is a language used to specify the hardware
  - – HDL - VHSIC Hardware Description Language
  - – VHSIC - Very High Speed Integrated Circuit
  - – Used mostly in Europe
- Verilog is another language used to specify the hardware
  - – Used mostly in the United States

Why VHDL?

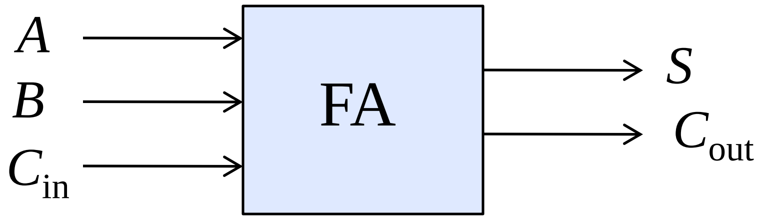
- VHDL is used to
  - – Verifies that you have connected right by simulating the circuit
  - – describes the large structures in a simple way and then generates the circuit by synthesis
  - – allows for structured descriptions of a circuit

VHDL increases the level of abstraction!

Types of VHDL code

- There are two types of VHDL code
  - **VHDL for synthesis:** The code is used as an input to a synthesis tool which converts it into an implementation (for example FPGA or ASIC)
  - **VHDL modeling and simulation** code is used to describe a system at an early stage. Since the code can be simulated so you can check whether the intended functionality is correct

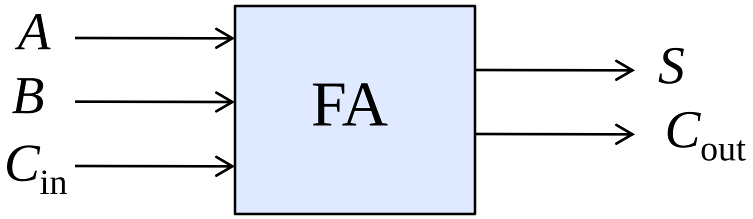
Entity (FA)

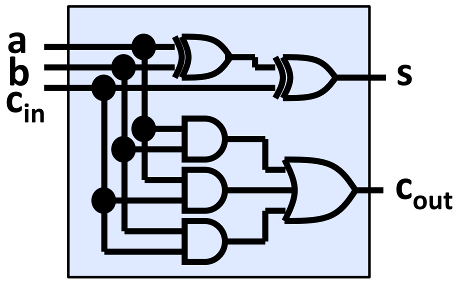


```
ENTITY fulladder IS
     PORT( A,B,Cin :  IN STD_LOGIC;
           S,Cout  : OUT STD_LOGIC);
END fulladder;
```

The ***entity*** describes the ports to the outside of the circuit.
The circuit as a block.

Architecture (FA)





```
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 describes the function inside the circuit.

VHDL port

- PORT declaration establishes *interface* between the component and the outside world
- A port declaration contains three things:
  - The ***name*** of the port
  - The ***direction*** of the port
  - Port's ***datatype***

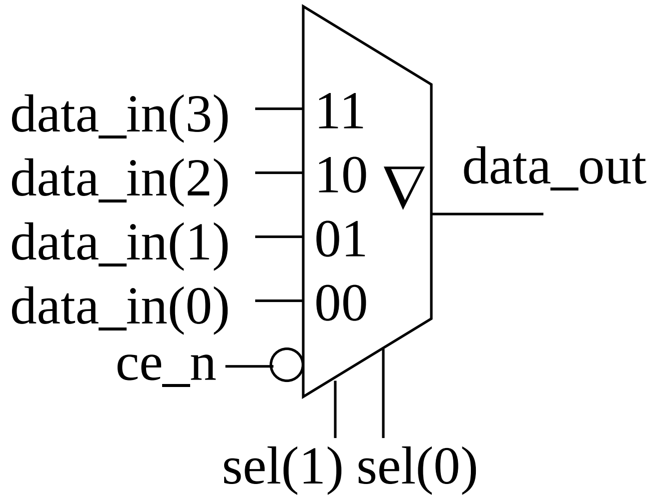
- Example:

```
ENTITY test IS
   PORT ( name : direction data_type);
END test;
```

The most common data types

- Scalars (single-variable signals)
  - Bit ("0", "1")
  - Std_logic ('U', '0', ’1', 'X', 'Z', 'L', 'H', 'W', '-')
  - Integer
  - Real
  - Time
- Vectors (many-variable signals)
  - BIT_VECTOR - vector of bits
  - STD_LOGIC_VECTOR - vector of std_logic

VHDL Example: 4/1 MUX

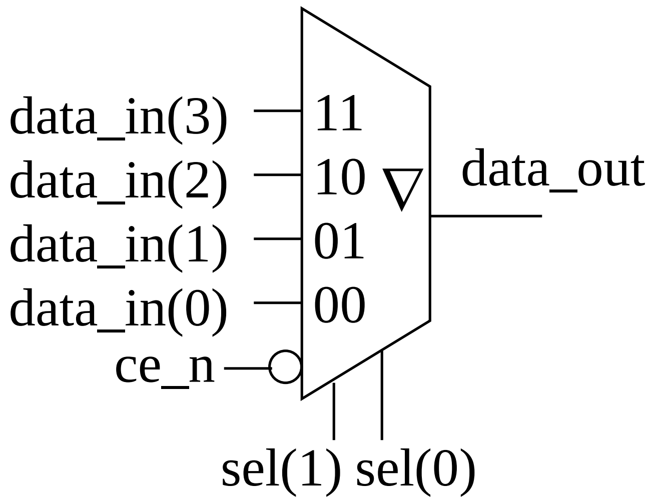


```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Multiplexer_41 IS
PORT(ce_n: IN std_logic; -- Chip Enable (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 Example: 4/1 MUX (cont.)

```
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;
```



More on VHDL

- The study material on synthesis shows a number of VHDL constructs and the resulting hardware
- The following slides contain extra material The book gives many examples and detailed explanations of VHDL

Synthesis tool Quartus

- The course textbook contains a CD with the synthesis tool, Quartus
- You will use Quartus in Lab 3

Summary

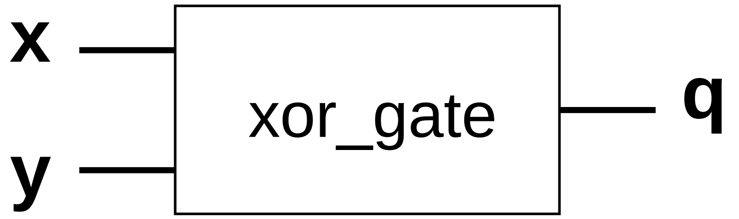
- Implementation of functions with MUXes
  - Shannon decomposition
- Look-up tables, ROM
- Decoder, encoder, code converters
- Introduction to VHDL
- Next lecture: BV pp. 383-418, 469-471

VHDL (Not part of the exam)

Entity (XOR)

- An entity describes a component's *interface* with the outside world
- PORT-declaration indicates if it is an input or an output
- An *Entity* is a symbol of a component.

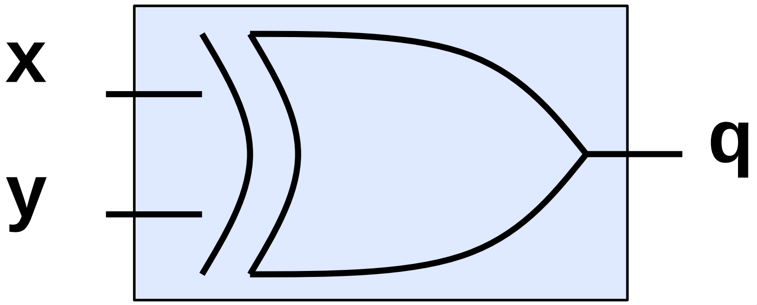
```
ENTITY xor_gate IS
    PORT (x, y: IN BIT;
          q: OUT BIT);
END xor_gate;
```



Use English names for variable names in the code!

Architecture (XOR)

- An *architecture* describes the operation of a component
- ***An entity can have many architectures***, but only one can be active at a time
- An architecture corresponds to the component diagram or behavior



Code for Simulation

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

Signal declaration

Signal-declaration is used inside architectures to declare internal (local) signals:

```
signal a, b, c, d: bit;

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

Signal-assignment is used to describe the behavior:

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

Sequential vs Parallel Code

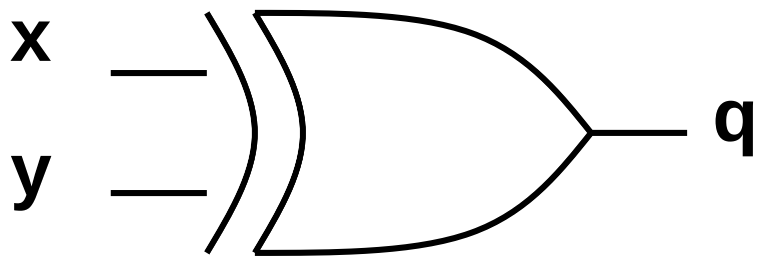
- There are two types of code execution in VHDL: sequential and parallel
  - Sequential code describes the hardware from a "programmer's" point of view and it is executed in the order which is defined
  - The parallel code is executed regardless of the order. It is *asynchronous*.

VHDL description styles

- Sequential (Behavioral)
  – similar to how to write desktop applications
- Data Flow (RTL)
  – Concurrent assignments
- Structural
  – similar to how to connect components

Sequential style

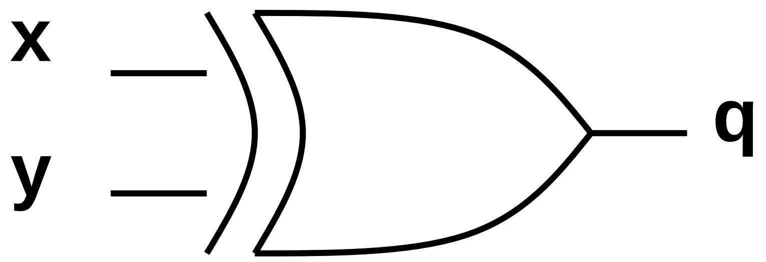
XOR gate



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

Data flow style

XOR gate

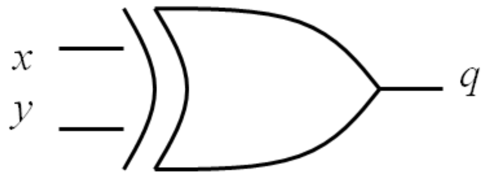


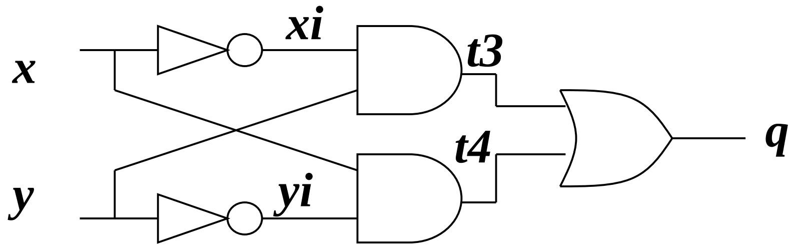
```
q <= a xor b;

- Or in behavioral dataflow style

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

Structural style





```
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);
```

Structural style
Component declaration

- A component must be declared before it can be used

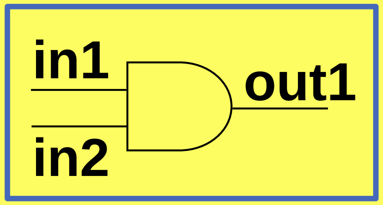
```
ARCHITECTURE Test OF test_entity
    COMPONENT and_gate
      Port (in1, in2: IN BIT;
               out1: BIT OUT);
    END COMPONENT;
... more statements...
```

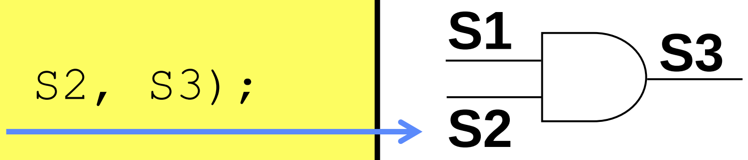
- It is necessary, unless it is not in a library somewhere

Structural style
Component instantiation

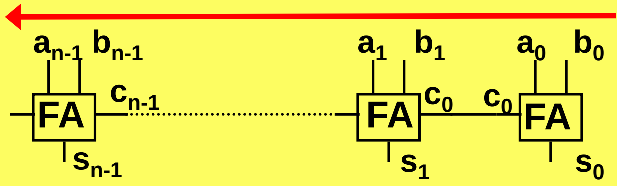
- Component *instantiation ring* connects the component interface with the signals in the architecture

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





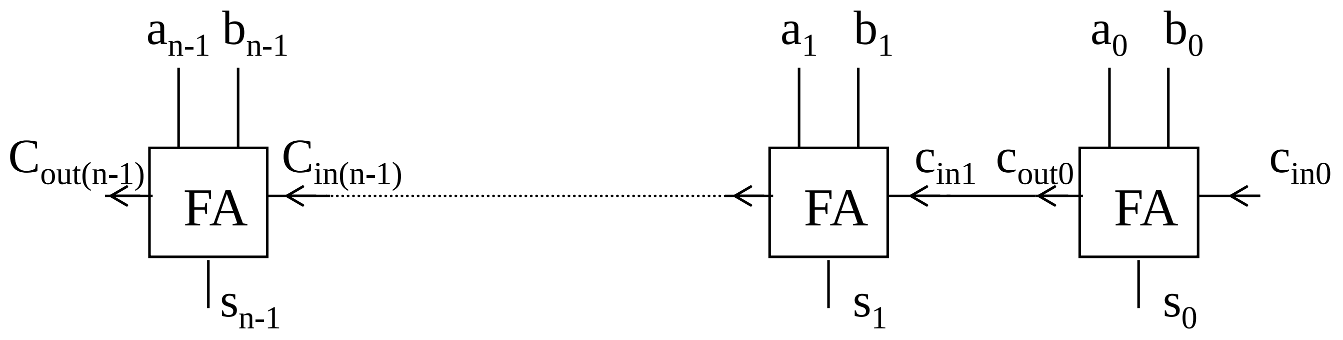
Generate



- Generate-statement couples many similar elements

```
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 OF structural 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), p (i));
   end Generate; - G0
   U0: full_adder PORT MAP (a(0), b(0), '0', c(0), p(0));
   UN: full_adder PORT MAP (a(n-1),b(n-1),c(n-2),OPEN, s(n-1);
END structural;
```

Generate n-bit adder



Five lines of code generates the ripple-carry n-bit adder from Lecture 5!

The test bench stimuli 1

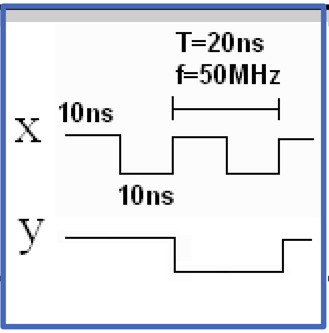
The ENTITY is empty!

```
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:bit;
BEGIN
    x <= not(x) after 10 ns;
    y <= not(y) after 20 ns;
    U1:xor_gate PORT MAP (x,y,ut1);
END example;
```

The circuit under test is used as a component of the test bench program

Here are the test signals generated



The test bench stimuli 2

```
ENTITY testbench IS END testbench;

ARCHITECTURE xor_stimuli_2 of testbench IS
    COMPONENT xor_gate
      PORT(x,y:IN bit;q:OUT bit);
    END COMPONENT;
    signal x,u1,u2,u3:bit; -- Endast en in-signal
    for U1:xor_gate use entity work.xor_gate(behave);
    for U2:xor_gate use entity work.xor_gate(data_flow);
    for U3:xor_gate use entity work.xor_gate(structural);
BEGIN
    x <= not(x) after 10 ns;
    U1:xor_gate PORT MAP (x,x,ut1);
    U2:xor_gate PORT MAP (x,x,ut2);
    U3:xor_gate PORT MAP (x,x,ut3);
END example;
```

Test bench

A test bench can mark when the desired events occur during the execution.

Or mark when unwanted events occur

The result of a run with a test bench can be saved in a file, as a proof that everything is ok - or as a troubleshooting aid if it did not go well.