INF2270 — Spring 2010

Philipp Häfliger

Lecture 2: Boolean Functions, Combinational Logic

UNIVERSITETET I OSLO

content

Boolean Functions

- Leftover from Last Lecture
- Simplification with Karnaugh Maps

Combinational Logic Circuits

- Concepts
- Encoder/Decoder
- Multiplexer/Demultiplexer

content

Boolean Functions
- Leftover from Last Lecture
- Simplification with Karnaugh Maps

Combinational Logic Circuits
- Concepts
- Encoder/Decoder
- Multiplexer/Demultiplexer

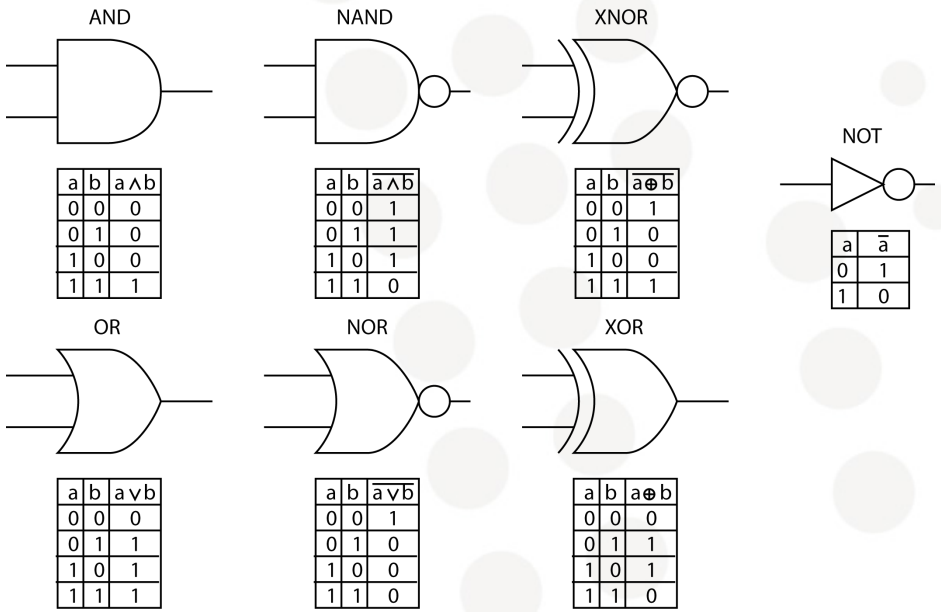
content

Boolean Functions
- Leftover from Last Lecture
- Simplification with Karnaugh Maps

Combinational Logic Circuits
- Concepts
- Encoder/Decoder
- Multiplexer/Demultiplexer

Representing Boolean Functions

1. expression with variables and operators
2. truth table
3. graphically with logic gates (combinational logic)



a	b	a ∧ b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$\overline{a \wedge b}$
0	0	1
0	1	1
1	0	1
1	1	0

a	b	$\overline{a \oplus b}$
0	0	1
0	1	0
1	0	0
1	1	1

a	$\bar{a}$
0	1
1	0

a	b	a ∨ b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$\overline{a \vee b}$
0	0	1
0	1	0
1	0	0
1	1	0

a	b	a ⊕ b
0	0	0
0	1	1
1	0	1
1	1	0

Examples: simplify

Solutions

$$
\begin{aligned}
& a \wedge b \vee a \wedge \bar{b} \\
= \quad & a \wedge (b \vee \bar{b}) \\
= \quad & a \wedge 1 \\
= \quad & a
\end{aligned}
$$

$$
\begin{aligned}
& a \wedge b \wedge c \vee \bar{a} \wedge b \wedge c \quad \vee \quad \bar{a} \wedge b \wedge \bar{c} \wedge (a \vee c) \\
= \quad & (a \vee \bar{a}) \wedge b \wedge c \quad \vee \quad \bar{a} \wedge b \wedge \bar{c} \wedge a \vee \bar{a} \wedge b \wedge \bar{c} \wedge c \\
= \quad & 1 \wedge b \wedge c \quad \vee \quad 0 \vee 0 \\
= \quad & b \wedge c
\end{aligned}
$$

Examples: representation with truth tables (1/2)

Solution

$F = a \wedge b \vee \bar{a} \wedge \bar{b}$

a	b	F

Examples: representation with truth tables (1/2)

Solution

$F = a \wedge b \vee \bar{a} \wedge \bar{b}$

a	b	F
0	0	
0	1	
1	0	
1	1	

Examples: representation with truth tables (1/2)

Solution

$F = a \wedge b \vee \bar{a} \wedge \bar{b}$

a	b	F
0	0	1
0	1	
1	0	
1	1	

Examples: representation with truth tables (1/2)

Solution

$F = a \wedge b \vee \bar{a} \wedge \bar{b}$

a	b	F
0	0	1
0	1	0
1	0	
1	1	

Examples: representation with truth tables (1/2)

Solution

$F = a \wedge b \vee \bar{a} \wedge \bar{b}$

a	b	F
0	0	1
0	1	0
1	0	0
1	1	

Examples: representation with truth tables (1/2)

Solution

$F = a \wedge b \vee \bar{a} \wedge \bar{b}$

a	b	F
0	0	1
0	1	0
1	0	0
1	1	1

Examples: representation with truth tables (2/2)

a	b	c	G

solution

$G = a \wedge b \wedge c \vee \bar{a} \wedge c \vee a \wedge \bar{b}$

Examples: representation with truth tables (2/2)

a	b	c	G
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

solution

$G = a \wedge b \wedge c \vee \bar{a} \wedge c \vee a \wedge \bar{b}$

Examples: representation with truth tables (2/2)

a	b	c	G
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	1

solution

$G = a \wedge b \wedge c \vee \bar{a} \wedge c \vee a \wedge \bar{b}$

Examples: representation with truth tables (2/2)

a	b	c	G
0	0	0	
0	0	1	1
0	1	0	
0	1	1	1
1	0	0	
1	0	1	
1	1	0	
1	1	1	1

solution

$G = a \wedge b \wedge c \vee \bar{a} \wedge c \vee a \wedge \bar{b}$

Examples: representation with truth tables (2/2)

a	b	c	G
0	0	0	
0	0	1	1
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	
1	1	1	1

solution

$G = a \wedge b \wedge c \vee \bar{a} \wedge c \vee a \wedge \bar{b}$

Examples: representation with truth tables (2/2)

a	b	c	G
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

solution

$G = a \wedge b \wedge c \vee \bar{a} \wedge c \vee a \wedge \bar{b}$

content

Boolean Functions

- Leftover from Last Lecture
- Simplification with Karnaugh Maps

Combinational Logic Circuits

- Concepts
- Encoder/Decoder
- Multiplexer/Demultiplexer

Simplification Example 2 Revisited

$$F = a \wedge b \wedge c \vee \bar{a} \wedge b \wedge c \vee \bar{a} \wedge b \wedge \bar{c} \wedge (a \vee c) \longrightarrow$$

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Transscribing a Truth Table into a K-Map

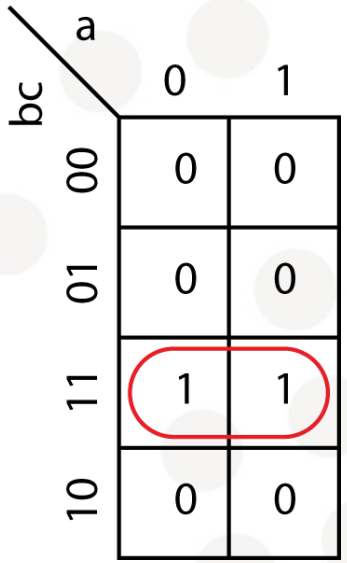
a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

bc \ a	0	1
00	0	0
01	0	0
11	1	1
10	0	0

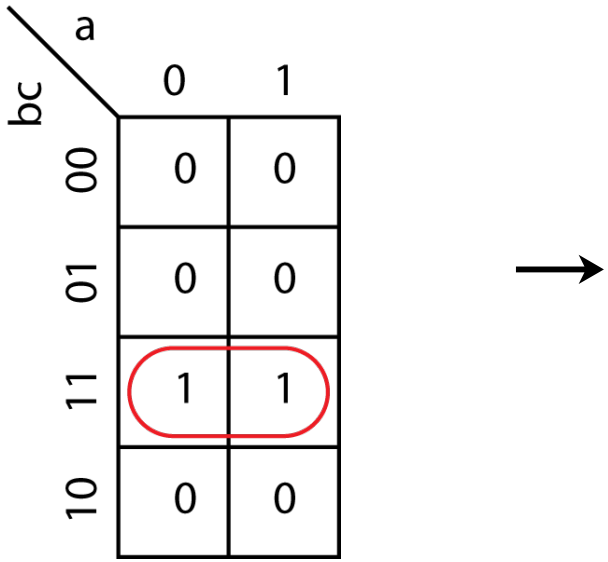
The input variables are listed on the x- and y-axis as Gray-code: only one bit shift between two subsequent positions.

Identify the 'minterms'

Identify rectangles filled with 1's and containing 2^n 1's (i.e. 1,2,4,8 ... elements). Find a set of a minimal number of such rectangles that covers all 1's. Overlap of the rectangles is allowed, actually desired to maximize their size. Note that the K-map wraps around at its boundaries, i.e rectangles can be formed across the map edges.



The minterms define a 'sum' of 'products'



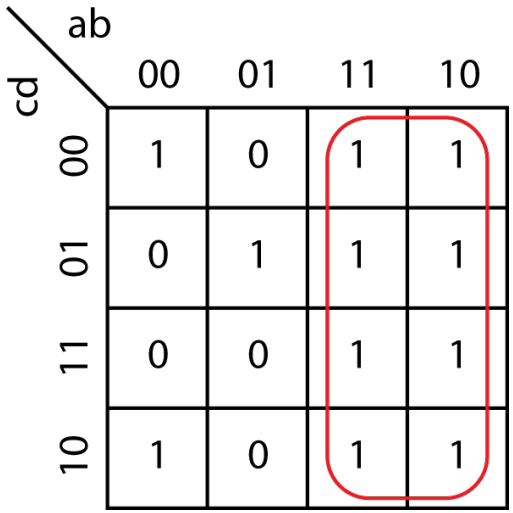
$$(b \wedge c)$$

Each rectangle defines a 'product' (elements and-ed) where the elements are the input variables that remain *constant* within the rectangle. Finally, all 'products' are or-ed. (Note: there is only one 'product' in this first example. See next example!)

K-map simplification: Example 2

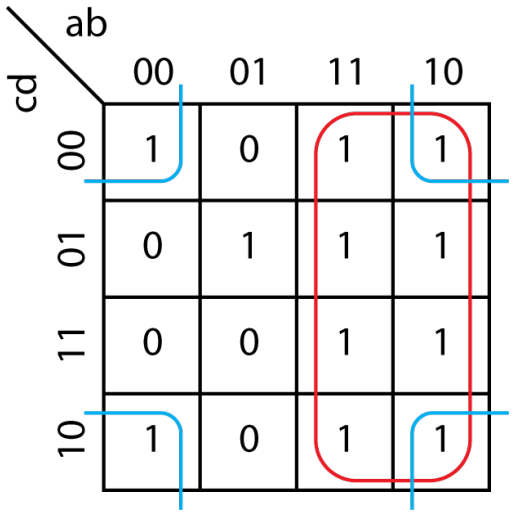
cd \ ab	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	0	0	1	1
10	1	0	1	1

K-map simplification: Example 2



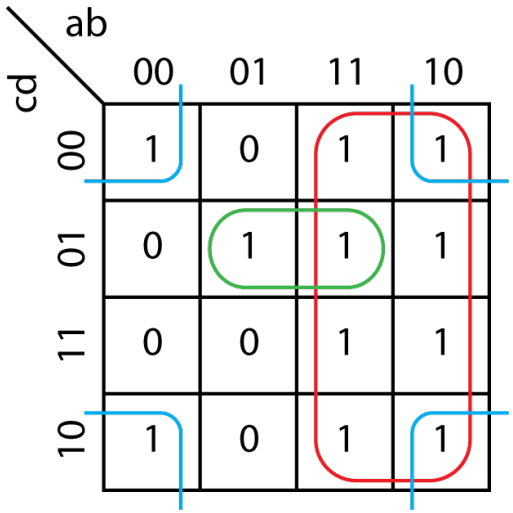
(a)

K-map simplification: Example 2



$$a \vee (\bar{b} \wedge \bar{d})$$

K-map simplification: Example 2



$$a \vee \bar{b} \wedge \bar{d} \vee (b \wedge \bar{c} \wedge d)$$

K-Maps Based on the '0's

One can also use the 0 to form the minterms and derive an expression for the inverse function $\bar{F}$ instead of F by exact same procedure. Then one can use the deMorgan theorem to turn the sum of product into a product of sums to derive F.

Properties of K-maps

- Karnaugh maps are useful to a size of up to 6 Boolean variables
- It is only possible to have up to two variables along one axis. Karnaugh maps with 5-8 variables become, thus, 3-dimensional. (Example in the weekly exercise for lecture 3)

content

Boolean Functions
- Leftover from Last Lecture
- Simplification with Karnaugh Maps

Combinational Logic Circuits
- Concepts
- Encoder/Decoder
- Multiplexer/Demultiplexer

content

Combinational Logic

Combinational logic circuits are feed-forward logic/digital circuits with no memory that can be described by Boolean functions.

Note what is implied here: logic gates can *also* be connected in ways that include feed-back connections that implement/include *memory* that *cannot* be described as Boolean functions! This is then not 'combinational logic', but 'sequential logic', of which we will talk later.

Design and Analysis of Digital Circuits

Design of a digital circuit is the process of assembling circuit blocks to form a bigger digital circuit.

Analysis of a digital circuit is the process of finding out what it is doing, e.g. (in the case of combinational logic!) by finding an equivalent Boolean function or a complete truth table.

A complete analysis is quite trivial for small digital circuits but neigh impossible for circuits of the complexity of a modern CPU. Hierarchical approaches in design and analysis provide some help.

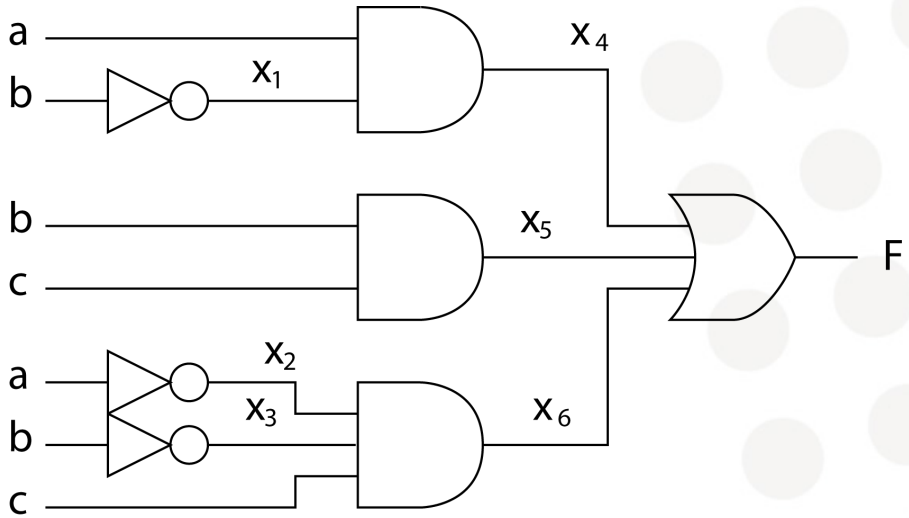
The first Pentium on the market had a mistake in its floating point unit.

> *After the Intel 286 there was the 386 and then the 486, but the 585.764529 was then dubbed 'Pentium' for simplicity sake.*

Logic Gate Signals

The logic gates introduced earlier are most often implemented to operate on voltages as input and output signals: a certain range of input voltage is defined as 'high' or logic '1' and another range is defined as 'low' or '0'. E.g. in a digital circuit with a 1.8V supply one can, for instance, guarantee an input voltage of 0V to 0.5V to be recognised as '0' and 1.2V to 1.8V as '1' by a logic gate. On the output side the gate can guarantee to deliver a voltage of either >1.75V or <0.05V.
These safety margins between input and output make (correctly designed!) digital circuits very robust (which is necessary with millions of logic gates in a CPU, where a single error might impair the global function!)

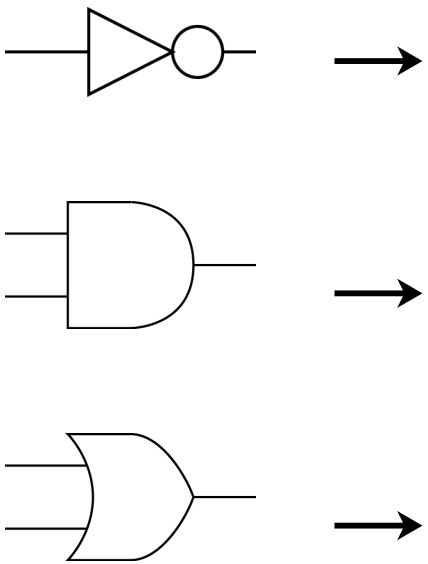
Combinational Logic Analysis: Example



$$\equiv \underbrace{a \wedge \bar{b}}_{x_4} \vee \underbrace{b \wedge c}_{x_5} \vee \underbrace{\bar{a} \wedge \bar{b} \wedge c}_{x_6}$$

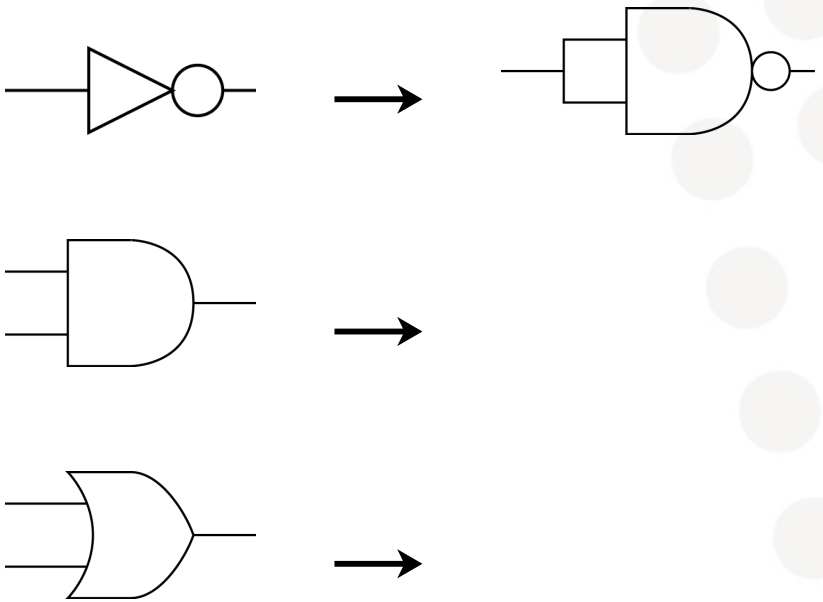
Universality of NAND and NOR

Any Boolean function can also be implemented using only NAND (or only NOR) gates. Use $\overline{a \wedge b} = \bar{a} \vee \bar{b}$ (deMorgan) and $\bar{\bar{a}} = a$ to proof!



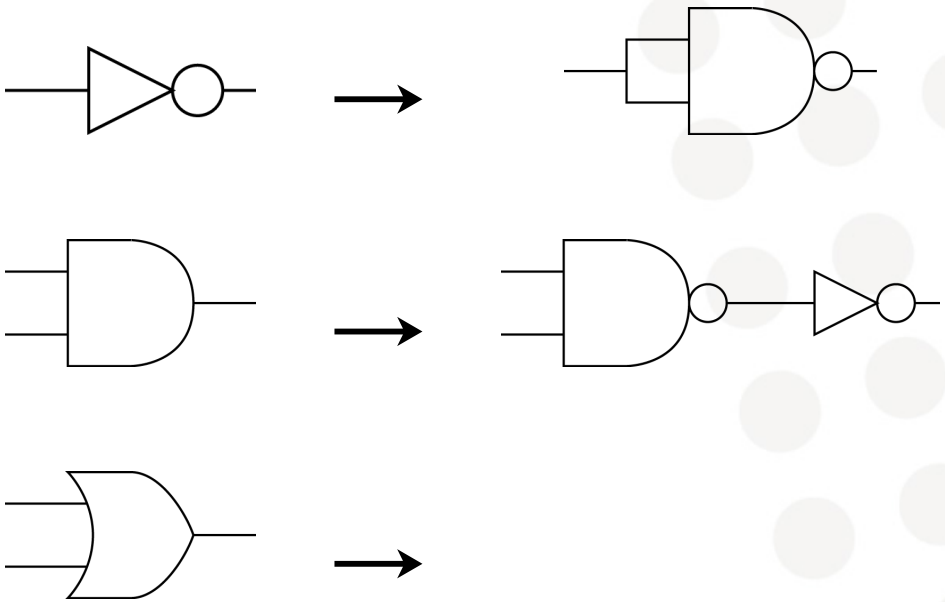
Universality of NAND and NOR

Any Boolean function can also be implemented using only NAND (or only NOR) gates. Use $\overline{a \wedge b} = \bar{a} \vee \bar{b}$ (deMorgan) and $\bar{\bar{a}} = a$ to proof!



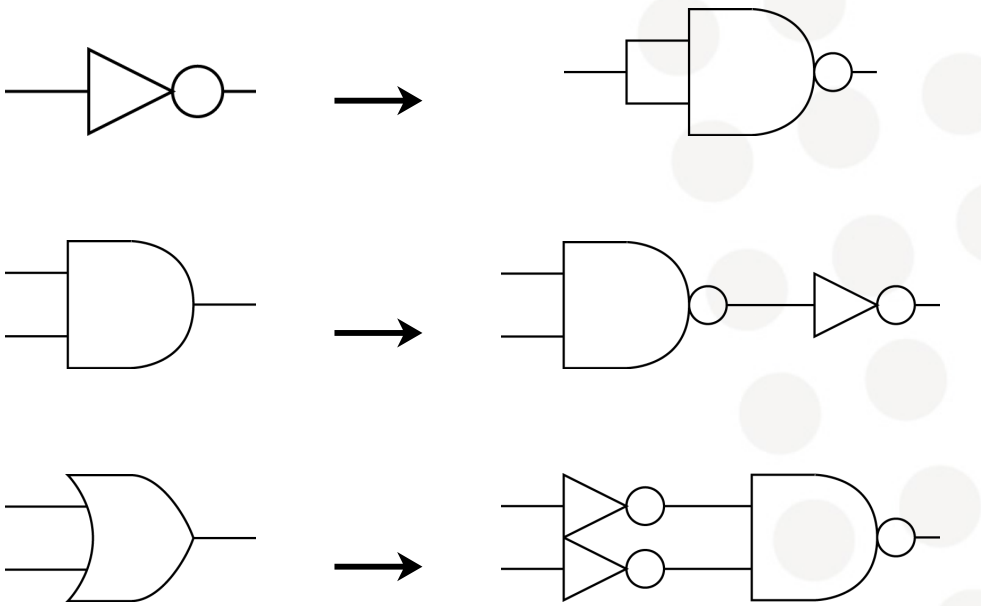
Universality of NAND and NOR

Any Boolean function can also be implemented using only NAND (or only NOR) gates. Use $\overline{a \wedge b} = \bar{a} \vee \bar{b}$ (deMorgan) and $\bar{\bar{a}} = a$ to proof!



Universality of NAND and NOR

Any Boolean function can also be implemented using only NAND (or only NOR) gates. Use $\overline{a \wedge b} = \bar{a} \vee \bar{b}$ (deMorgan) and $\bar{\bar{a}} = a$ to proof!



Standard Combinational Logic

Some combinational logic (and of course also sequential logic → later) is often used in computational devices and are usually provided as 'black boxes' guaranteeing a defined function.
Examples:

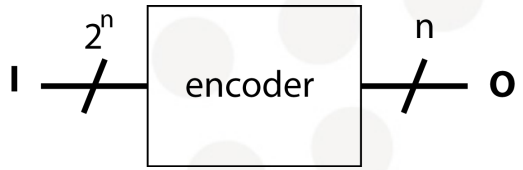
- encoder/decoder
- multiplexer/demultiplexer
- adders/multipliers

There are actually variations on how those functions are implemented, resulting in different processing speeds and/or power consumption and/or scalability (i.e. how easy it is to adapt the same function for more inputs).

content

3-bit Encoder Specification

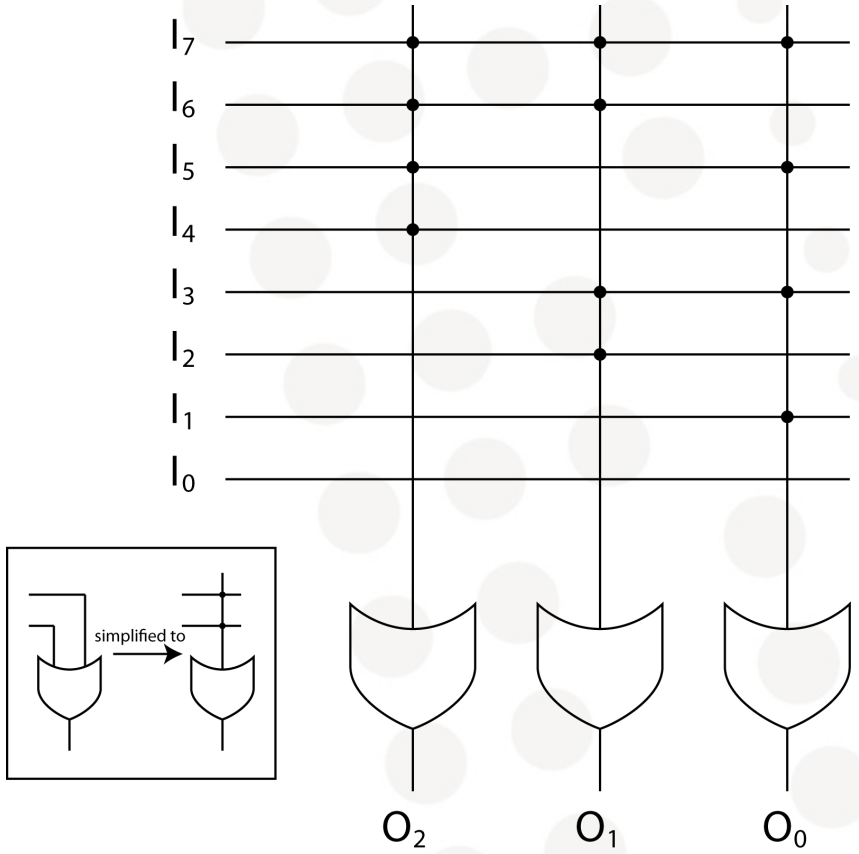
An encoder in digital electronics usually refers to a circuit that converts 2^n inputs into n outputs, as specified by the following truth table.



Simple 3-bit Encoder Truth Table

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

3-bit Encoder Implementation Variant



3-bit Encoder Remarks

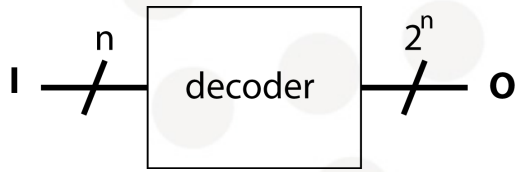
The truth table that was given is not complete: some inputs are ‘illegal’. Circuitry that produces the input should ensure to only produce legal states. In our specific digital circuit implementation we can of course deduct what the output in each illegal case would be, but other implementation may actually provide different outputs in those non-defined cases, and still be valid encoders!
The following truth table is a deterministic specification of an encoder, without ‘illegal’ inputs, where the ’highest’ active input bit determines the output. ’X’ in the table means ‘do not care’, or ‘for any state’ and allows to abbreviate the truth table. This would require a different implementation, but we will not present it here.

3-bit Priority Encoder Truth Table

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

3-bit Decoder Specification

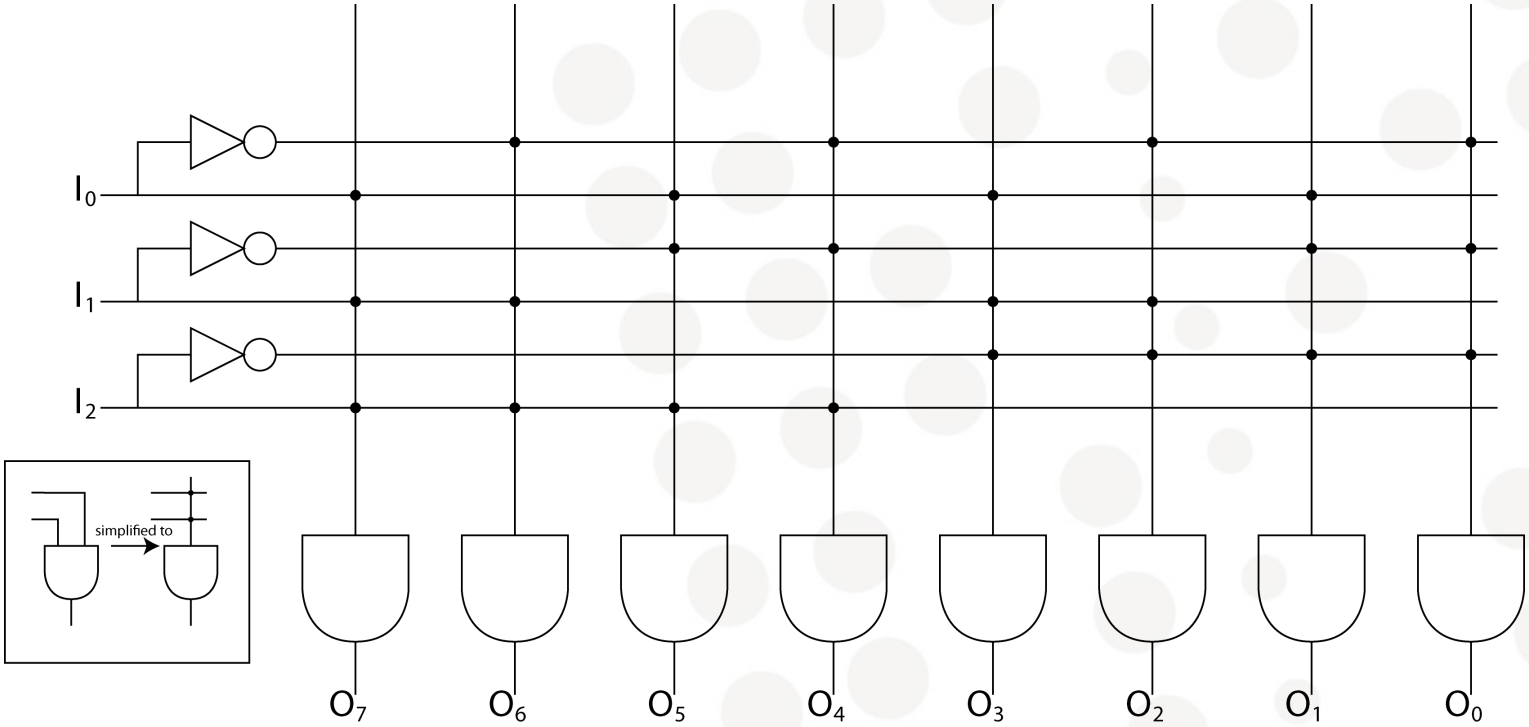
A decoder is the inverse function of a encoder, in digital circuits usually decoding n inputs into 2^n outputs.



3-bit Decoder Truth Table

I_2	I_1	I_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

3-bit Decoder Implementation Variant



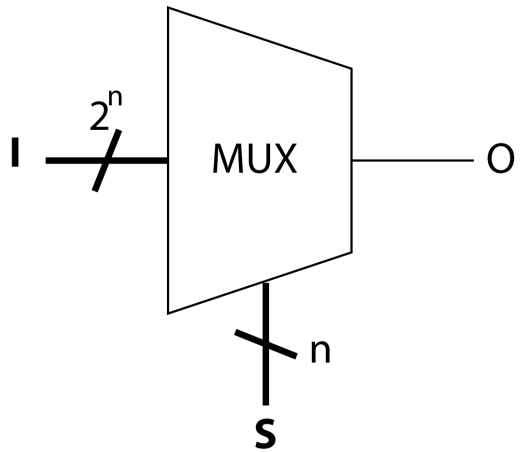
content

Boolean Functions
- Leftover from Last Lecture
- Simplification with Karnaugh Maps

Combinational Logic Circuits
- Concepts
- Encoder/Decoder
- Multiplexer/Demultiplexer

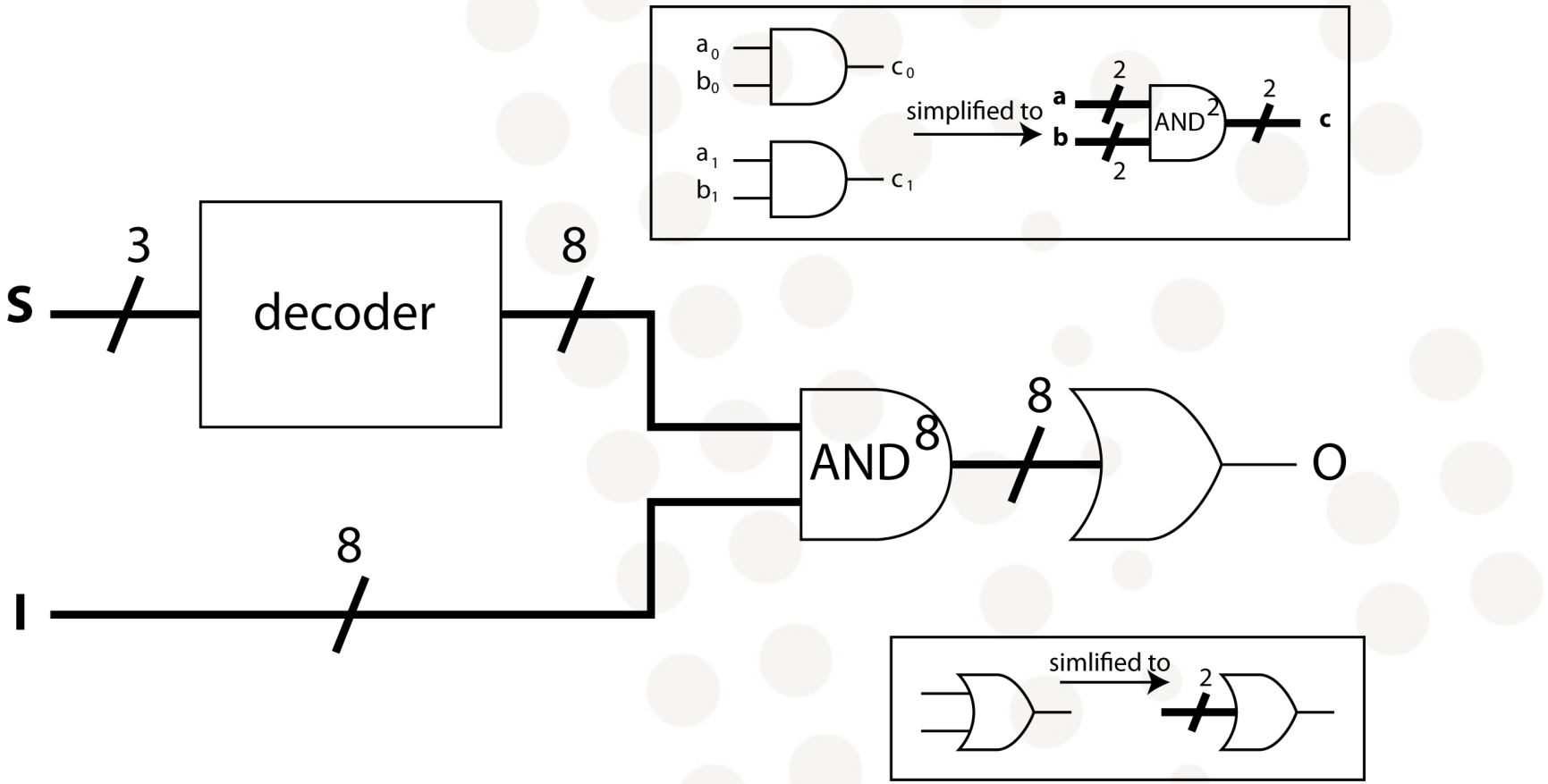
3-bit Multiplexer Specification

A multiplexer routes one of 2^n input signals as defined by the binary control number **S** to the output.



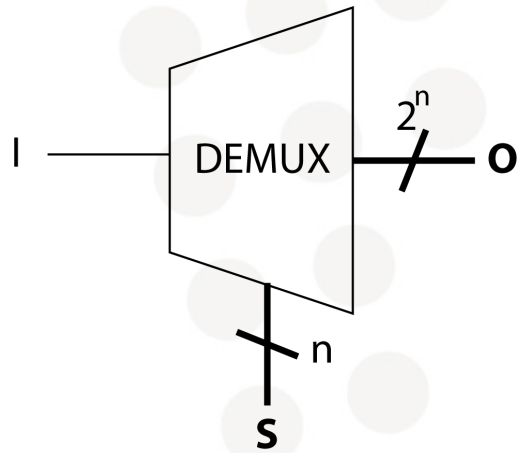
S_2	S_1	S_0	O
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

3-bit Multiplexer Implementation Variant



3-bit Demultiplexer Specification

A demultiplexer performs the inverse function of a multiplexer, routing one input signal to one of 2^n outputs as defined by the binary control number **S**



3-bit Demultiplexer Truth Table

S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	I
0	0	1	0	0	0	0	0	0	I	0
0	1	0	0	0	0	0	0	I	0	0
0	1	1	0	0	0	0	I	0	0	0
1	0	0	0	0	0	I	0	0	0	0
1	0	1	0	0	I	0	0	0	0	0
1	1	0	0	I	0	0	0	0	0	0
1	1	1	I	0	0	0	0	0	0	0

3-bit Demultiplexer Implementation Variant