# — Computer Engineering —
# Module 3: Combinatorial Circuits

Universität Zürich[UZH]

CSG Communication Systems Group



## Modul 3: Combinatorial Circuits

- Formal Basics of Logics
  - Boolean Algebra
- Normalized and Minimized Forms
- Implementation of Combinatorial Circuits by Gates
- Design of Combinatorial Circuits
  - Minimization and KV Diagrams
  - Function Programming
- Delay Effects



## Boolean Algebra Instantiated

- The instantiated Boolean algebra is defined explicitly by the following table:

| Boolean Algebra | Instantiated |
|---|---|
| V | {0,1} |
| ⊕ | ∨ |
| ⊗ | ∧ |
| **n** | 0 |
| **e** | 1 |
| $\bar{a}$ | $\bar{a}$ |

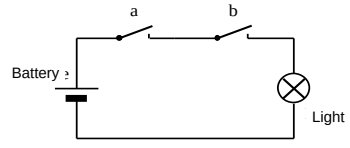
Alternative representations:

a + b for a ∨ b or naming as “OR/ODER”
a & b, ab for a ∧ b or naming as “AND/UND”



## Implementation of Logical Conjunctions (1)

- A technical implementation of logical conjunctions uses simple switching models instead of logical elements:
- AND:

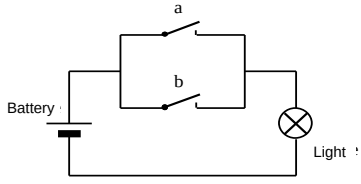



- The light is only switched “on” (function’s value equals 1), if the two switches are shut (a AND b equal 1), otherwise the light is switched “off” (function’s value equals 0).



## Implementation of Logical Conjunctions (2)

- OR:
  The light is switched “on”, if the two switches are shut.





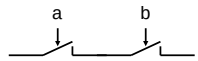

## Boolean Algebra Instantiated (1)

- All conjunctions are represented in tables:

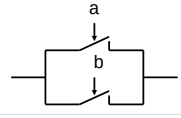
| a | b | a ∧ b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| a | b | a ∨ b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

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




Technical implementation with switches




no synonym available

## Boolean Algebra Instantiated (2)

- Huntington Axioms:

- H1: Commutative Law
  $a \vee b = b \vee a$
  $a \wedge b = b \wedge a$

- H2: Distributive Law
  $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
  $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

- H3: One Element
  $a \wedge 1 = a$
  $a \vee \overline{a} = 1$

- H4: Zero Element
  $a \wedge \overline{a} = 0$
  $a \vee 0 = a$



## Boolean Algebra Instantiated (3)

- From those 4 Huntington's Axioms additional clauses are derived:

- G1: Associative Law
  $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
  $(a \vee b) \vee c = a \vee (b \vee c)$

- G2: Idempotence Law
  $a \wedge a = a$
  $a \vee a = a$

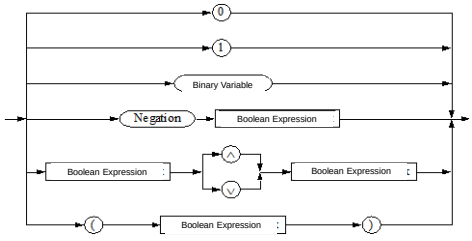
- G3: Law of Absorption
  $a \wedge (a \vee b) = a$
  $a \vee (a \wedge b) = a$

- G4: DeMorgan Law
  $\overline{a \wedge b} = \overline{a} \vee \overline{b}$
  $\overline{a \vee b} = \overline{a} \wedge \overline{b}$
  Also named as: NAND NOR



## Boolean Expressions (1)

- A Boolean expression is defined as a sequence of characters, which contains binary variables, the operators $\wedge$ and $\vee$, and brackets, and which follows the syntactical rules, defined by the syntax diagramm:






## Boolean Expressions (2)

- Examples:
  - Syntactical correct expressions: $a \vee b$, 0, $( a \wedge b ) \vee \overline{c}$
- No Boolean expressions, due to their syntactical incorrectness:
  - $a \vee \vee a$, 1 0, $( ) \vee c$

- For the Boolean Algebra instantiated the constants 0 and 1 are termed sometimes – closely related to statements – as Boolean Truth Values:
  - 0 : false 1 : true

- A Boolean expression by itself does not represent a Boolean value, since it may contain Boolean variables!
  - Only an assignment of a binary value with a Boolean value allows for the calculation of this Boolean expression's Boolean truth value



## Representation of Boolean Functions

1. Function table, truth table ("Funktionstabelle")
2. Algebraic expression (symbolic form)
3. Graph

Function Table

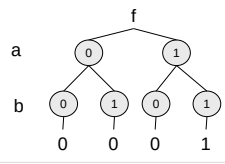
| a | b | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Symbolic form

$f = a \wedge b$

Graph

(*e.g.*, Shannon Tree)






## Full Operators System (1)

- Definition:
  A system of operators, with which all Boolean functions can be represented, is termed full operators system.

- The operators ($\wedge$, $\vee$, $\overline{\ }$) do form such a full operators system.

- Example:
  $a \leftrightarrow b$ equivalence with the same result as $( a \wedge b ) \vee ( \overline{a} \wedge \overline{b} )$.

  $\leftrightarrow$ thus, replaced by basic operations $\wedge$, $\vee$, and $\overline{\ }$.

## Full Operators System (2)

| Operators System | Representation of ... | | |
|---|---|---|---|
| | Negation | Conjunction | Disjunction |
| $(\wedge, \vee, \overline{\ })$ | $\bar{a}$ | $a \wedge b$ | $a \vee b$ |
| $(\wedge, \overline{\ })$ | $a\bar{a}$ | $a \wedge b$ | $\overline{\bar{a} \wedge \bar{b}}$ |
| $(\vee, \overline{\ })$ | $\bar{a}$ | $\overline{\bar{a} \vee \bar{b}}$ | $a \vee b$ |
| $(\barwedge)$ | $a \barwedge a$ | $(a \barwedge b) \barwedge (a \barwedge b)$ | $(a \barwedge a) \barwedge (b \barwedge b)$ |
| $(\barvee)$ | $a \barvee a$ | $(a \barvee a) \barvee (b \barvee b)$ | $(a \barvee b) \barvee (a \barvee b)$ |
| $(\wedge, \leftrightarrow\!\!\!\!/)$ | $a \leftrightarrow\!\!\!\!/\ 1$ | $a \wedge b$ | $a \leftrightarrow\!\!\!\!/\ b \leftrightarrow\!\!\!\!/\ (a \wedge b)$ |

Hint: The operand "$\wedge$" may not be required in writing: $a \wedge b \leftrightarrow ab$



## Tautology (1)

- In which cases two expressions A and B define the exact same Boolean function?
- Similarly noted as: Is $A \leftrightarrow B$ a tautology?
- Two Boolean functions are defined as follows:
  $f_1(a,b) = (a \wedge b) \vee (\bar{a} \wedge \bar{b})$
  $f_2(a,b) = (a \vee \bar{b}) \wedge (\bar{a} \vee b)$
- Is $f_1$ identical to $f_2$
  or
  is $(a \wedge b) \vee (\bar{a} \wedge \bar{b}) \leftrightarrow (a \vee \bar{b}) \wedge (\bar{a} \vee b)$ a tautology?



## Tautology (2)

- Proof by utilizing (a) function tables or
  (b) logical transformations by applying algebraic laws.
  - (b) See next slide
  - (a) Two expressions are equivalent, if their results for all interpretations and, thus, all possible combinations of assignments for variables are identical.

| a b | $(a \wedge b) \vee (\bar{a} \wedge \bar{b})$ | $(a \vee \bar{b}) \wedge (\bar{a} \vee b)$ | $A \leftrightarrow B$ |
|---|---|---|---|
| 0 0 | 1 | 1 | 1 |
| 0 1 | 0 | 0 | 1 |
| 1 0 | 0 | 0 | 1 |
| 1 1 | 1 | 1 | 1 |



## Tautology (3)

- Transformations by applying algebraic laws:

$$(a \wedge b) \vee (\bar{a} \wedge \bar{b}) = [(a \wedge b) \vee \bar{a}] \wedge [(a \wedge b) \vee \bar{b}]$$
(Distributive Law)

$$= [(a \vee \bar{a}) \wedge (b \vee \bar{a})] \wedge [(a \vee \bar{b}) \wedge (b \vee \bar{b})]$$
(Distributive Law)

$$= [1 \wedge (b \vee \bar{a})] \wedge [(a \vee \bar{b}) \wedge 1]$$
(Inverse Element)

$$= (b \vee \bar{a}) \wedge (a \vee \bar{b})$$
(Neutral Element)

$$= (\bar{a} \vee b) \wedge (a \vee \bar{b})$$
(Commutative Law)



## Modul 3: Combinatorial Circuits

- Formal Basics of Logics
  - Boolean Algebra
- Normalized and Minimized Forms
- Implementation of Combinatorial Circuits by Gates
- Design of Combinatorial Circuits
  - Minimization and KV Diagrams
  - Function Programming
- Delay Effects



## Normal Forms

- A Boolean function may be represented by different Boolean expressions.
- A standardized representation of Boolean functions within a full operator's system such as $(\wedge, \vee, \overline{\ })$ is defined by the conjunctive normal form (CNF) – sometimes called "clausal normal form" – and the disjunctive normal form (DNF).
- Definition:
  A literal $L_i$ is defined as either a positive variable $x_i$ or its negation (negative variable) $\bar{x}_i$, thus, $L_i \in \{x_i, \bar{x}_i\}$
  - Only positive or negative literals exist!

## Clauses (Product Terms)

- Definition:
  A conjunctive clause (or product term) $K(x_1,...,x_m)$ is defined as the conjunction of literals

  $\bigwedge_{i\in\{1,...,m\}} L_i = L_1 \wedge \ldots \wedge L_m$

  or the constants "0" or "1 "
- Examples: $a \wedge a \wedge b$ $\quad \bar{x}_i \wedge x_i$
- Every conjunctive clause $K(x_1,...,x_m) = \bigwedge_{i\in\{1,...,m\}} L_i$ may be represented in a form such that a single variable x only appears at most once in one literal.



## Literals and Clauses

- If $L_h = x$, $L_j = x$, $h \neq j$ (multiple "positive" appearances)
  $\Rightarrow L_h \wedge L_j = x$
  $\Rightarrow K(x_1,...,x_m) = \bigwedge_{i=1}^{m} L_i$
- Multiple appearances of x can be reduced due to the Idempotence Law ( $x \wedge x = x$).
- If $L_h = x$ und $L_j = \bar{x}$ (mixed "positive" and "negated" appearances)
  $\Rightarrow L_h \wedge L_j = 0$
  $\Rightarrow K(x_1,...,x_m) = 0$ (clause equals 0)



## Implicant and Minterm

- Definition:
  A conjunctive clause $K(x_1,...,x_n)$ is termed implicant of a Boolean function $y(x_1,...,x_n)$, if $K \rightarrow y$
  - K implies y (K is an implicant of y), if y also takes the value 1 whenever K equals 1
  - *I.e.*, for every assignment $B \in \{0,1\}^n$ the following holds true:
    If K(B) = 1, y(B) = 1 holds as well.
- Definition:
  An implicant of a Boolean function $y(x_1,...,x_n)$ is termed minterm, if a literal of each variable $x_i$ within the function y of implicants exists exactly once.
- Examples
  - Minterms of a Boolean function $y(x_1,...,x_4)$: $x_1 \wedge x_2 \wedge x_3 \wedge x_4$ or $x_1 \wedge \bar{x}_2 \wedge x_3 \wedge x_4$
  - No minterms of the Boolean function $y(x_1,...,x_4)$: $x_1 \wedge x_2$ or $x_1 \wedge \bar{x}_2 \wedge x_3 \wedge x_3 \wedge x_4$



## Disjunctive Normal Form

- Definition:
  Assuming a Boolean function $y(x_1,...,x_n)$, a Boolean expression is termed to be disjunctive normal form (DNF) of the function y, if it exists as a disjunction of all minterms $K_i$.
  $y = K_0 \vee K_1 \vee ..... \vee K_k$, $\quad k \leq 2^n\text{-}1$
  - No two conjunctions $K_i$, $K_j$ exist with $i \neq j$ , such that they are equivalent
  - Any Boolean expression is in its disjunctive normal form, iff it is either a single conjunctive clause or a series of conjunctive clauses joined by "$\vee$".
- Examples
  - f ( a,b,c ) = $a\,b\,c \vee a\,\bar{b}\,c \vee \bar{a}\,\bar{b}\,c$ determines a valid DNF
  - f ( a,b,c ) = $a\,b\,c \vee a\,\bar{b} \vee c\,a\,b \vee a\,(\,b\,c \vee \bar{b}\,\bar{c}\,)$ does not determine a DNF, since
    - $a\,\bar{b}$ does not contain all variables
    - $a\,b\,c$ and $c\,a\,b$ are equivalent
    - $a\,(\,b\,c \vee \bar{b}\,\bar{c}\,)$ does not define a "pure" conjunction



## Implicat and Maxterm

- Definition:
  A disjunctive clause $D(x_1,...,x_n)$ is termed implicat of a Boolean function $y(x_1,...,x_n)$, if $D \rightarrow y$
  - D implies y (D is an implicant of y), if $\bar{y}$ also takes the value 0 whenever D equals 0
  - *I.e.*, for every assignment $B \in \{0,1\}^n$ the following holds true:
    If D(B) = 0, y(B) = 0 holds as well.
- Definition:
  An implicat of a Boolean function $y(x_1,...,x_m)$ is termed maxterm, if a literal of each variable $x_i$ within the function y of implicants exists exactly once.
- Examples
  - Maxterms of the Boolean function $y(x_1,...,x_3)$:
    $x_1 \vee x_2 \vee x_3$ or $x_1 \vee \bar{x}_2 \vee x_3$



## Conjunctive Normal Form

- Definition:
  Assuming a Boolean function $y(x_1,..., x_m)$, a Boolean expression is termed to be conjunctive normal form (CNF) of the function y, if it exists as a conjunction of all maxterms $D_i$.
  $y = D_0 \wedge D_1 \wedge ..... \wedge D_k$ , $k \leq 2^m\text{-}1$
  - No two disjunctions $D_i$, $D_j$ exist with $i \neq j$ , such that they are equivalent
  - Any Boolean expression is in its conjunctive normal form, iff it is either a single disjunctive clause or a series of disjunctive clauses joined by "$\wedge$".
- Examples
  - f ( a,b,c ) = $(a \vee b \vee c) \wedge (a \vee b \vee c) \wedge (a \vee b \vee c)$ determines a valid CNF
  - f ( a,b,c ) = $(a \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (c \vee a \vee b)$ does not determine a CNF, since
    - $a \vee \bar{b}$ does not contain all variables
    - $a \vee b \vee c$ and $c \vee a \vee b$ are equivalent

## DNF and CNF

- Any function with n variables *may* show up to $2^n$ minterms and maxterms, respectively.
- Example: For n = 3 those read in full and complete:

| | Minterm | Maxterm |
|---|---|---|
| 0 | $\bar{a}\ \bar{b}\ \bar{c}$ | $a \vee b \vee c$ |
| 1 | $a\ \bar{b}\ \bar{c}$ | $\bar{a} \vee b \vee c$ |
| 2 | $\bar{a}\ b\ \bar{c}$ | $a \vee \bar{b} \vee c$ |
| 3 | $a\ b\ \bar{c}$ | $\bar{a} \vee \bar{b} \vee c$ |
| 4 | $\bar{a}\ \bar{b}\ c$ | $a \vee b \vee \bar{c}$ |
| 5 | $a\ \bar{b}\ c$ | $\bar{a} \vee b \vee \bar{c}$ |
| 6 | $\bar{a}\ b\ c$ | $a \vee \bar{b} \vee \bar{c}$ |
| 7 | $a\ b\ c$ | $\bar{a} \vee \bar{b} \vee \bar{c}$ |



## "Reading" and Utilizing Normal Forms

- Every single minterm of a DNF equals one dedicated row in the functions table, which maintains always the function's value 1.
- Every single maxterm of a CNF equals one dedicated row in the functions table, which maintains always the function's value 0.
- Disjunctive and conjunctive normal forms determine unique representations!
  - Except for permutations, such as abc, acb, bac, bca, cab, or cba which are all equivalent.
- Functions represented by a single minterm only, determine for a single assignment only the function's value 1
  - Aside from the trivial null function, those functions show a minimal number of 1s!
- Functions represented by a single maxterm only, determine for a single assignment only the function's value 0
  - Aside from the trivial one function, those functions show a maximal number of 1s!



## Example: DNF and CNF

A Boolean function is uniquely specified by listing all minterms and maxterms, respectively.

| | c b a | y | Minterms | Maxterms |
|---|---|---|---|---|
| | 0 0 0 | 1 | $\bar{a}\ \bar{b}\ \bar{c}$ | |
| | 0 0 1 | 0 | | $\bar{a} \vee b \vee c$ |
| | 0 1 0 | 0 | | $a \vee \bar{b} \vee c$ |
| | 0 1 1 | 1 | $a\ b\ \bar{c}$ | |
| | 1 0 0 | 1 | $\bar{a}\ \bar{b}\ c$ | |
| | 1 0 1 | 0 | | $\bar{a} \vee b \vee \bar{c}$ |
| | 1 1 0 | 0 | | $a \vee \bar{b} \vee \bar{c}$ |
| | 1 1 1 | 1 | $a\ b\ c$ | |

DNF: $y = (\bar{a}\ \bar{b}\ \bar{c}) \vee (a\ b\ \bar{c}) \vee (\bar{a}\ \bar{b}\ c) \vee (a\ b\ c)$

CNF: $y = (\bar{a} \vee b \vee c)(a \vee \bar{b} \vee c)(\bar{a} \vee b \vee \bar{c})(a \vee \bar{b} \vee \bar{c})$



## Reminder of DNF and CNF

- Disjunctive and conjunctive normal forms determine unique representations!
  - Except for permutations, such as abc, acb, bac, bca, cab, or cba which are all equivalent.

Example: $y = a\ \bar{b} \vee c$

- DNF: $y = \bar{a}\ \bar{b}\ c \vee \bar{a}\ b\ c \vee a\ \bar{b}\ \bar{c} \vee a\ \bar{b}\ c \vee a\ b\ c$
- CNF: $y = (a \vee b \vee c) \wedge (a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee c)$



## Minimized Forms

- Goals
  - Deriving shortest possible expressions for a given Boolean function
  - Implementing such a function at the lowest possible costs!
- Similarly to the disjunctive and conjunctive normal forms, so-called minimal disjunctive normal forms ("Disjunktive Minimalform") and minimal conjunctive normal forms ("Konjunktive Minimalform") exist.
- Multiple minimal disjunctive and conjunctive normal forms may exist for the same Boolean function
  - Example:
    - $y = a\ \bar{b} \vee b\ \bar{c} \vee \bar{a}\ c$ and
    - $y = a\ \bar{c} \vee \bar{b}\ c \vee \bar{a}\ b$

    define an identical function and both are minimal disjunctive normal forms



## Deriving Minimized Forms

- In search of such a minimal disjunctive/conjunctive normal form the approach may become complex for functions with many variables
  - No "unique" or even optimal algorithm exists to find "the" normal form
  - Thus, sub-optimal solutions may be determined only by applying heuristics
- The minimization of forms operates in two steps
  - Step 1: Derive the set of implicants and implicats, respectively, for a given Boolean function f, while ensuring that the number of literals is kept as small as possible.
  - Step 2: Select from these sets of implicants and implicats, respectively, the smallest possible number of implicants and implicats, respectively, such that their disjunction and conjunction, respectively, determine this function f.
- Prerequisites for practical minimizations
  - Application of full operators systems with one operator only!

## NAND/NOR Conversion

- The ( $\overline{\wedge}$ ) system (NAND System) and ( $\overline{\vee}$ ) system (NOR System) are full operators systems (cf. before)
  ⇨ Any disjunctive and conjunctive expression can be represented within the NAND and NOR System
- Conversion options in four different cases
  Case 1: Function in disjunctive form ⇒ ( $\overline{\wedge}$ ) system
  Case 2: Function in disjunctive form ⇒ ( $\overline{\vee}$ ) system
  Case 3: Function in conjunctive form ⇒ ( $\vee$ ) system
  Case 4: Function in conjunctive form ⇒ ( $\overline{\wedge}$ ) system
- Why are exactly those conversions relevant?
  ⇨ Much simpler implementation in hardware!
  – NANDs or NORs are very simple to be implemented as combinatorial circuits.



## NAND Conversion (Example for Case 1)

- Assume a function f provided in disjunctive form
  – To be converted into ⇒ ( $\overline{\wedge}$ ) system
- Conversion in two steps:
  Step 1: Double negation of entire f
  Step 2: Application of DeMorgan law
  – Resulting in an expression, which only contains NAND operators



## Example: NAND Conversion

$$y = \bar{a}\,b\,c \vee a\,\bar{b}\,c \vee a\,b\,\bar{c} \vee \bar{a}\,\bar{b}\,\bar{c}$$

$$= \overline{\overline{\bar{a}\,b\,c \vee a\,\bar{b}\,c \vee a\,b\,\bar{c} \vee \bar{a}\,\bar{b}\,\bar{c}}}$$

$$= \overline{\overline{\bar{a}\,b\,c} \wedge \overline{a\,\bar{b}\,c} \wedge \overline{a\,b\,\bar{c}} \wedge \overline{\bar{a}\,\bar{b}\,\bar{c}}}$$

$$= \text{NAND}_4(\text{NAND}_3(\bar{a}, b, c), \text{NAND}_3(a, \bar{b}, c), \text{NAND}_3(a, b, \bar{c}), \text{NAND}_3(\bar{a}, \bar{b}, \bar{c}))$$

$\text{NAND}_k(x_1,...,x_k)$ is defined as a k-ary function, for which:

$$\text{NAND}_k(x_1,...,x_k) = \begin{cases} 0 \text{ for } x_1 = ... = x_k = 1 \\ 1 \text{ otherwise} \end{cases}$$



## Example: $\text{NAND}_2$ Function

Representation of the $\text{NAND}_2$ function by the $\overline{\wedge}$ operator:

**Problem:**

Operators $\overline{\wedge}$ and $\overline{\vee}$ do not show an associative characteristic.

$$(x_1 \,\overline{\wedge}\, x_2) \,\overline{\wedge}\, x_3 \neq x_1 \,\overline{\wedge}\, (x_2 \,\overline{\wedge}\, x_3)$$

$$(x_1 \,\overline{\vee}\, x_2) \,\overline{\vee}\, x_3 \neq x_1 \,\overline{\vee}\, (x_2 \,\overline{\vee}\, x_3)$$

$$\text{NAND}_3(x_1, x_2, x_3) = \overline{x_1 \wedge x_2 \wedge x_3} = \overline{(x_1 \,\overline{\wedge}\, x_2)} \,\overline{\wedge}\, x_3$$

$$(x_1 \,\overline{\wedge}\, x_2) \,\overline{\wedge}\, x_3 = \overline{\overline{x_1 \wedge x_2} \wedge x_3} \neq$$

$$x_1 \,\overline{\wedge}\, (x_2 \,\overline{\wedge}\, x_3) = \overline{x_1 \wedge \overline{x_2 \wedge x_3}}$$



## Summary September 26, 2018

- Formal Basics of Logic
  - Boolean Algebra Instantiated
    - Huntington's axioms, Boolean laws, expressions, and functions
    - Full operators systems
    - Representation
      - Function's table (Truth Table), symbolic/algebraic form, Shannon tree (Graph)
- Normalized and Minimized Forms
  - Clause/product term and literal
  - Disjunctive Normal Form (DNF), unique
    - Implicant and minterm
      - Row of function table with function's value 1
  - Conjunctive Normal Form (CNF), unique
    - Implicat and maxterm
      - Row of function table with function's value 0
  - Minimal Conjunctive/Disjunctive Normal Form
    - Not unique, but "short" due to the use of "suitable" minterms and maxterms
  - $\text{NAND}_x$ function



## Modul 3: Combinatorial Circuits

- Formal Basics of Logics
  - Boolean Algebra
- Normalized and Minimized Forms
- Implementation of Combinatorial Circuits by Gates
- Design of Combinatorial Circuits
  - Minimization and KV Diagrams
  - Function Programming
- Delay Effects

## Implementation of Combinatorial Circuits

General hierarchy

- Register transfer level: logic circuits as elements
  - Multiplexer, shift registers, algebraic adder, ... **Examples to come**
- Gate level: logic gates
  - AND, OR, NAND, ... **Just discussed in M3 theoretically, ✓ now briefly transferred into gates**
- Electronic switch level: Transistors
  - Electrical engineering basics **Example to come**
- Layout level: Transistor technologies
  - Electrical engineering basics **Example mentioned in M1**

Bottom-up

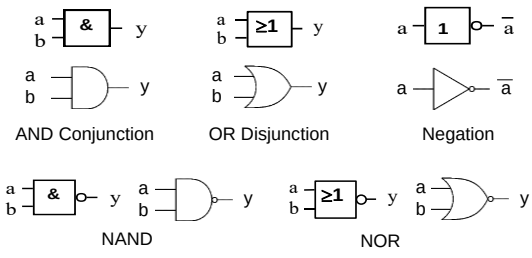


## Gate Level

- Full abstraction from any internal technical details of electrical engineering (almost)
- Full focus on logical behavior
  - Mapping of logic, Boolean functions onto circuits without knowledge of electrical engineering
- All circuits are represented in terms of symbols
- Algebraic and logic operators differ world-wide:
  - & different symbol for AND "∧"
  - ≥ 1 the output value equals 1, if at all ≥1-inputs a 1 is applied, thus, OR "∨"
  - ° different symbol for the negation "¬"
- Further different symbols and operators available on the Web.



## Symboles (DIN 40900 Teil 12, ANSI/IEEE-Standard 91-1984, IEC-Standard & US-Symbole)

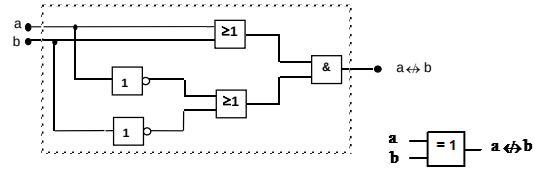



All circuits of these types are termed **gate(s)**.



## Exclusive OR Gate

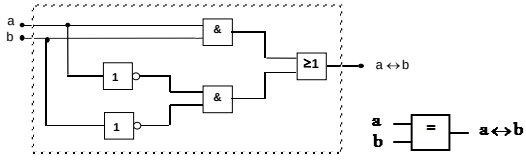
Simple gates are used to set-up hierarchically complex gates, which by themselves use (partially) dedicated symbols.




Exclusive OR (Antivalenz) composed out of five simpler gates. Applied to detect and determine the carry bit (Überlaufbit) in additions.



## Equivalence Gate




Equivalence (Äquivalenz) composed out of five simpler gates.



## Modul 3: Combinatorial Circuits

- Formal Basics of Logics
  - Boolean Algebra
- Normalized and Minimized Forms
- Implementation of Combinatorial Circuits by Gates
- Design of Combinatorial Circuits
  - Minimization and KV Diagrams
  - Function Programming
- Delay Effects

## Design of Combinatorial Circuits (1)

- Any practical design of circuits need to address
  - Gates are “ideal”, they generate heat to be diverted
  - Gates need space in terms of square micrometer [$\mu m^2$]
  - Gates show delay effects (see end of M3)
- Technical criteria
  - Power consumption, switching time, space, material, durability
  - Correct implementation with respect to static, dynamics (hazards see end of M3)
  - Constraints: #input lines, #output lines, maximal power of inputs/output
- Economic criteria
  - Effort for the design (salaries, simulations, data bases)
  - Effort of implementation (costs per gate, costs per square micrometer [$\mu m^2$])
  - Effort of operation (costs for tests, installation, maintenance at operation)



## Design of Combinatorial Circuits (2)

- Border line between technical and economic criteria not always clear
  - Selected criteria may be contradictory
    - Enhancing reliability ⇒ Increased costs (redundancy, space, …)
    - Lower design costs ⇒ Increased design costs
    - Higher density of functionality (less chips) ⇒ Increased design costs

➔ The design process has to determine alternatives and for a given problem the best possible compromise on functionality and costs has to be found and selected.

- Major goal: By respecting mandatory technical requirements to reach an economically favorable compromise such that overall full costs are minimized.



## Minimization

- Basic assumption
  - Implementation of circuits in two stages
- Representation of function to be found, which minimizes the costs of its implementation, termed in its result as
  - Minimal form
- The approach to generate this minimal form is termed
  - Minimization
- Minimization works in three different ways
  - Algebraic approach
  - Graphical approach
  - Table-based approach
- Algebraic and graphical approaches only work up to 5 or 6 variables, otherwise humans loose the comprehensive view, thus, table-based (maps) approaches are applied thereafter



## Reality Check

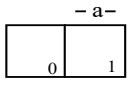
| Minimization Task | Prime Terms | Selection |
|---|---|---|
| Number of variables ≤ 6 | KV Diagram | KV Diagram<br>Coverage Maps |
| Assuming DF(CF), in search of DMF(CMF) | Consensus<br>Quine-McCluskey | Coverage Maps<br>(“Überdeckungstabelle”) |
| Assuming DF(CF) in search of DMF/CMF | Nelson | Coverage Maps |

Extended and powerful methods are accessible in selected literature (dedicated field)



## Graphical Approach

- KV Diagram (Karnaugh and Veitch)
  - Also termed Karnaugh map, Karnaugh diagram
- Rectangle, for which the right half is assigned to the variable a and the left half is assigned to $\bar{a}$:




**KV diagram for a single variable**

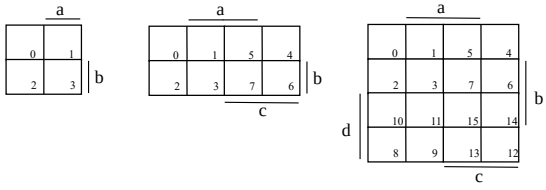


## KV Diagram (1)

- Numbered fields into determine the index of this variable‘s assignment
  - Index equals minterms, with the value 1
- By adding the truth values 0 or 1 in each of the KV diagrams fields, the Boolean function is characterized uniquely
- A KV diagram defines an additional representation of Boolean functions, especially considered as an alternative to function tables
- Examples:

## KV Diagrams (2)

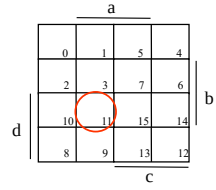
- KV diagrams for additional variables are defined via mirroring
  - Each additional variable doubles the number of possible assignments






## KV Diagrams (3)

- Every field is uniquely identified by reading the edges' names
  - Field 11 is assigned to: $a\ b\ \bar{c}\ d$
- The index with every field defines the index of this field's assignment of variables, while all variables are marked in in alphabetically reverse order
  - Field 11 = $1011_2$ = $d\ \bar{c}\ b\ a$






## KV Diagrams — Set-up (1)

- Assuming that the function *y* is defined by a function table
  - Each row of that function table corresponds to a single field in the KV diagram
  - ⇒ For each row of the functions table the corresponding indix will be determined and the function's value will be inserted
- Hint, to find all fields within the KV diagram:
  - Include all input variables in alphabetically reverse order into the table
  - In turn, the KV diagram can be completed according to their indices, which originate from the table's top to the table's bottom row



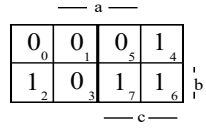
## KV Diagrams — Set-up (2)

Example: $y = \bar{a}\ b \vee b\ c \vee \bar{a}\ \bar{b}\ c$

*Function table:*

| Index | c | b | a | y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 1 |

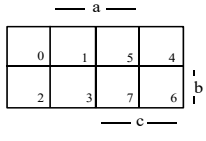
Resulting in KV diagram:






## Characteristics of KV Diagrams (1)

- Key characteristic
  - Minterms located symmetrically to any center line differ only in a single variable
- Example:




or

Minterm 0 = $\bar{c}\ \bar{b}\ \bar{a}$
Minterm 1 = $\bar{c}\ \bar{b}\ a$

or

Minterm 0 = $c\ \bar{b}\ \bar{a}$
Minterm 4 = $c\ \bar{b}\ \bar{a}$

or

Minterm 1 = $\bar{c}\ \bar{b}\ a$
Minterm 3 = $c\ \bar{b}\ a$

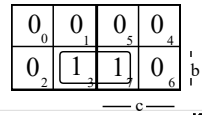


## Characteristics of KV Diagrams (2)

- Boolean algebra tells that terms differing in a single variable only may be minimized!
- Example:

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

- Thus, the resulting term does not contain the variable *c* anymore!
- In turn, minterms located symmetrically to any center line can be combined to a simpler term!

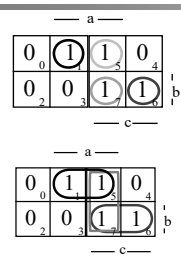


## Definition: Prime Implicant and Selection

- An implicant p is defined as a prime implicant, if no implicant q ≠ p exists, which is implied by p.
  $\forall q: \; q \neq p \Leftrightarrow \neg(p \rightarrow q)$
  - *I.e.*, p embraces the largest possible order (von größtmöglicher Ordnung).
- Proposition:
  Every function can be represented as a disjunction of its prime implicants.
- Prime implicants are accessible from any KV diagram
  - Select "manually" the largest possible blocks of ones, such that each block contains $2^k$ fields
- Example (continued next slide):
  $f = \bar{a}\ b\ c \vee a\ c \vee a\ \bar{b}\ \bar{c}$



## Example (1)




4 minterms:
($a\ \bar{b}\ \bar{c}$, $a\ \bar{b}\ c$, $a\ b\ c$, $\bar{a}\ b\ c$)

3 prime implicants:
first order implicants
($a\ \bar{b}$, $a\ c$, $b\ c$)

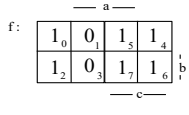
But ($a\ \bar{b}$, $b\ c$) would be sufficient!

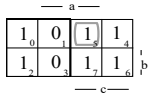
- $f = a\ \bar{b}\ \bar{c} \vee a\ \bar{b}\ c \vee a\ b\ c \vee \bar{a}\ b\ c$
  $= a\ \bar{b} \vee a\ c \vee b\ c = a\ \bar{b} \vee b\ c$



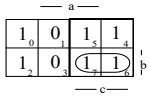
## Example (2)

f = minterms (0,2,4,5,6,7)

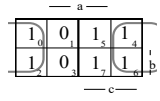



$g = a\ \bar{b}\ c$
determines an implicant




$g = b\ c$
determines an implicant

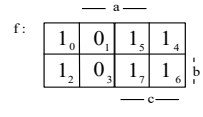


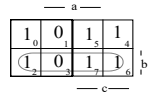

$g = \bar{a}$
determines an implicant



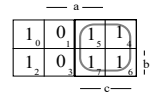
## Example (3)

f = minterms (0,2,4,5,6,7)

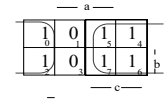



$g = b$
does not determine an implicant




$g = c$
determines an implicant



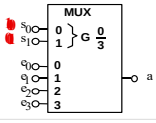

$g = \bar{a} \vee c$
does not determine an implicant

(Impl.)

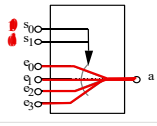


## Alternatives for Functions' Implementations

- Implementation of re-appearing functions by two alternatives:
  - By multiplexers/demultiplexers
  - Memory circuits
- A multiplexer (MUX) defines a circuit with multiple inputs and one output, for which by n control lines one of the $2^n$ inputs is selected and switched to the single output
- Multiplexers are classified according to their size
  - $2^n$:1 - Multiplexer (alternatively as a 1-out-of-$2^n$-multiplexer)
- Example 1-ouf-of-4-multplexer:




| $s_1$ | $s_0$ | a |
|---|---|---|
| 0 | 0 | $e_0$ |
| 0 | 1 | $e_1$ |
| 1 | 0 | $e_2$ |
| 1 | 1 | $e_3$ |






## Multiplexer-based Functions' Implementation

- Traditionally, a multiplexer controls data flows
- Here, it is applied to implement Boolean functions
  - A $2^n$: 1 - multiplexer can implement Boolean functions with n+1 variables
- An implementation table defines the behavior
  - $2^n$ columns for possible assignments of n control lines
  - 2 rows for the negated (negative) and non-negated (positive) (n+1)-st variable
- Content and assignment
  - The table contains all function's values depending on the input variables
  - Finally, each column is considered separately and gets assigned a 1-ary function $g \in \{0, 1, a, \bar{a}\}$, with which the input will be applied to according to the combination of all control variables

## Example

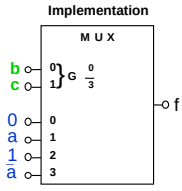
Implementing function f with a multiplexer:

$f = \bar{a}\, c \vee \bar{b}\, c \vee a\, b\, \bar{c}$

Implementation table in case of the selection of *b* and *c* as control inputs:

| c b | 0 0 | 0 1 | 1 0 | 1 1 |
|---|---|---|---|---|
| a = 0 | 0 | 0 | 1 | 1 |
| a = 1 | 0 | 1 | 1 | 0 |
| f = | 0 | a | 1 | $\bar{a}$ |

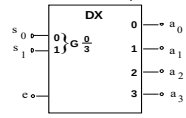
Implementation






## Demultiplexers/Decoders

- A circuit, which maps a single input depending on those *n* control lines onto one of the $2^n$ outputs, is termed demultiplexer
- Example 1-out-of-4 Demultiplexer:




| $s_1$ | $s_0$ | $a_0$ | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|---|---|
| 0 | 0 | e | 0 | 0 | 0 |
| 0 | 1 | 0 | e | 0 | 0 |
| 1 | 0 | 0 | 0 | e | 0 |
| 1 | 1 | 0 | 0 | 0 | e |

- Remarks
  - A demultiplexer offers an enable input *e* and *n* inputs $s_i$ for a binary number, which is decoded at the $2^n$ ouptputs $a_j$
  - If the enable input *e* = 0, all outputs are at 0, otherwise a 2-bit number is decoded
    - Example: By applying the number 2 ($s_1$= 1, $s_0$= 0), the ouput at $a_2$= 1 and all other outputs remain with 0
- Thus, a demultiplexer is termed also decoder



## Memory-based Functions' Implementation

- All circuits discussed so far, such as gates, multiplexers, or decoders, are defined by the function known in advance

  ➔ hardwired logic

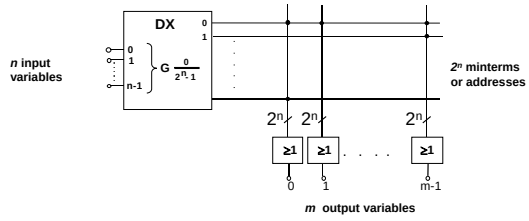
- However, circuits with multiple functions on board have to offer a higher flexibility to be adaptable to many application demands. Such an adaptation is termed personalization or programming

  ➔ micro-programmable logic
  (cf. M6 for CPUs)



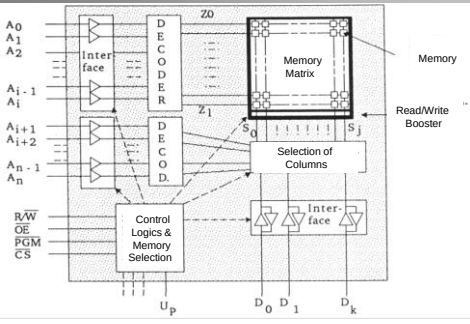
## Scheme of a Memory Cells

Scheme of memory cells, in which arbitrary function tables can be stored into:






## Organization of Memory Cells (1)






## Organization of Memory Cells (2)

- The input configuration (Eingangssignale) determines the selection of a memory cell (addressing) and the value stored at this address will be offered at the output configuration (Ausgangssignale)
- The output configuration of the decoder determines those min-terms of n input configurations (the rows of the function table)
- Storing the value 1 for a dedicated output configuration *i* means that this minterm is OR-connected to this *i*-th output; the value 0 indicates that this minterm will not be used

## Memory Types (1)

- The personalization of memory distuingishes a set of various different memory types with different characteristics:

- ROM (Read Only Memory)

- Memory circuits which can only be read during operation
- Programming was performed only once initially at the vendor‘s side (mask-programmable ROMs)
- Content of storage will remain unchanged for its entire lifetime



## Memory Types (2)

- PROM (Programmable Read Only Memory)

- Programmable ROMs, which are programmed by the user

- Programming:
  - Melting of fusible links (microscopic small connectors)
  - Stored charge based on dedicated physical processes, which maintain the charge within the circuit over years



## Memory Types (3)

- EPROM (Erasable Programmable Read Only Memory)

- A user-programmable memory circuit, which can be erased by applying UV light and which may be re-programmed again

- Very small quartz windows on top cover of circuit provide access

- Memory circuits with electrical erasing options (higher voltage):
  - EEPROM: Electrically Erasable PROM
  - EAROM: Electrically Alternable ROM
  - Application for memories of digital cameras (Compact Flash, Memory Stick)



## Memory Types (4)

- RAM (Random Access Memory)

- Memory, which enables reads and writes during normal operation

- A RAM will loose its programmed content once it will be decoupled from the power supply!

- Different implementations:
  - Dynamic RAM (DRAM)
  - Static RAM (SRAM)



## Comparison of RAM and ROM

- Key differences
  - RW: *read/write*
  - Access Times (AT) for R/W
  - Maintaining memory‘s content (operations OP) without power supply
  - Memory size (MS)

| | ROM | PROM | EPROM | Flash-PROM | DRAM | SRAM |
|---|---|---|---|---|---|---|
| RW | R | R | R ((W)) | R (W) | RW | RW |
| AT | + | + | + / – | + / – | ++ | +++ |
| OP | + | + | + | + | – | – |
| MS | + | + | + | + | + | + |



## Programmable Logic Array (PLA)

- Up to now the entire function table was stored in memory cells, while the function was implemented by its DNF (disjunctive normal form)

- Instead, by applying the DMF (minimal disjunctive normal form) many functions can be implemented efficiently (fewer memory cells)

- PLA (Programmable Logic Array):
  - In comparison to ROMs PLAs do see as the input configuration prime implicants for covering the function, instead of minterms
  - The decoder used up to now will be replaced by an AND matrix

## FPLA and PAL

- PLAs are being personalized during production (similar to ROMs)
- A user-programmable PLA with a fixed number of input configurations *n*, product terms *k*, and output configurations *m* is termed Field Programmable Logic Array (FPLA)
- Technical alternatives include PAL circuits, which did personalize already the AND matrix and the OR matrix at the time of production
- The role of PALs and FPLAs is determined by the demand for "similar" functions to be available in many devices, which may differ in their specific instantiations: reduction of costs, but no need to be fully flexible (programmable), but with fast operation

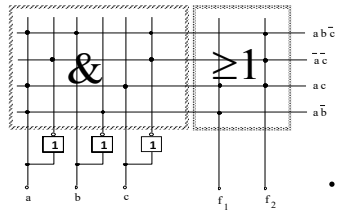


## Example: Implementation via PAL

Assume that the functions $f_1$ and $f_2$, already have been minimized:

$$f_1 = a\,\bar{b} \vee a\,c \quad \text{and} \quad f_2 = \bar{a}\,\bar{c} \vee a\,c \vee a\,b\,\bar{c}$$

Their PAL implementation reads as follows:




- Connector



## Modul 3: Combinatorial Circuits

- Formal Basics of Logics
  - Boolean Algebra
- Normalized and Minimized Forms
- Implementation of Combinatorial Circuits by Gates
- Design of Combinatorial Circuits
  - Minimization and KV Diagrams
  - Function Programming
- Delay Effects

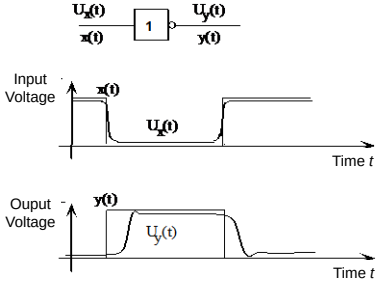


## Delay Effects

- The discussion of gates assumed so far that any logical operation was performed idealistically!
- In reality, gates are implemented by transistors, resistors (Widerstände), and capacitors, which are visible on the circuit's layout level.
- Thus, a temporal signal curve for an implemented gate differs from its ideal Boolean signal curve!



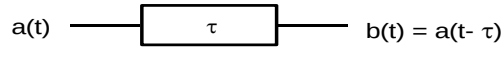
## Realistic and Ideal Signal Curve (Inverter)





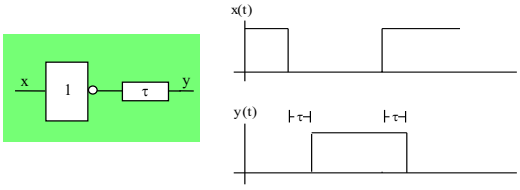

## Realistic Characterization of Gates

- Realistic effects on the gate level have been modeled in different models and complexities
- A simple model is termed Delay Model (Totzeitmodell)
  - Only signal delays are considered, which exist due to gates and links
  - A real gate is modeled by:
    - An ideal gate without any delays and
    - A delay element (Totzeitglied) with a defined delay, which depends on the switching time of the gate's implementation
    - The temporal behavior of a binary variable following the delay element is identical as of before the delay element, however only delayed by the time $\tau$



## Example: Delay Model of the Inverter/Negator

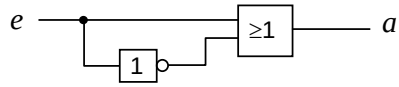
- Such a simple model allows for an investigation and a principle understanding of delay effects, however, the model is still too idealistic for concrete system designs






## Example: Application of the Inverter

**Assume:**

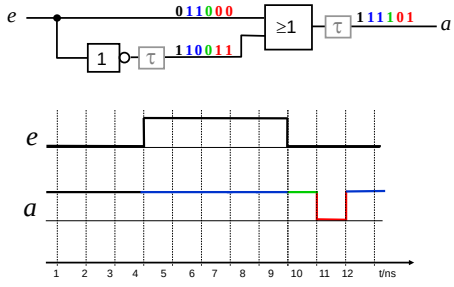



$$a = e \vee \overline{e} = 1$$

Both gates shall show a delay of 1 ns



## Time Chart






## Behavior of Combinatorial Circuits

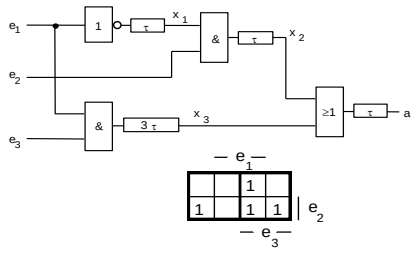
- Ideal combinatorial circuit
  - The output configuration does not change in case of previous and current assignments at the input utilizing the same values
  - The output configuration does change exactly once in case of previous and current assignments at the input utilizing different values
- Realistic combinatorial circuit
  - A change of the input configuration potentially results in different signal delays throughout the circuit
- Multiple changes of the output configuration, and thus the output signal curve, may occur until a stable final value is reached

  ➔ Hazard (Hasardfehler)



## Example

Function: $a = \overline{e}_1 e_2 \vee e_1 e_3$






## Assumption: Change of Input

- The two following transitions (changes of input values) are considered:

a) Inputs $e_2$ and $e_3$ are constant at 1 and the input $e_1$ changes from 0 to 1

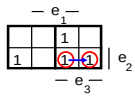
b) The inputs $e_2$ and $e_3$ are constant at 1 and the input $e_1$ changes from 1 to 0

## Functional Evaluation

**Function's values at the 2 transitions:**

**$(e_3,e_2,e_1) = (1,1,0) \Rightarrow a = 1$**

**$(e_3,e_2,e_1) = (1,1,1) \Rightarrow a = 1$**



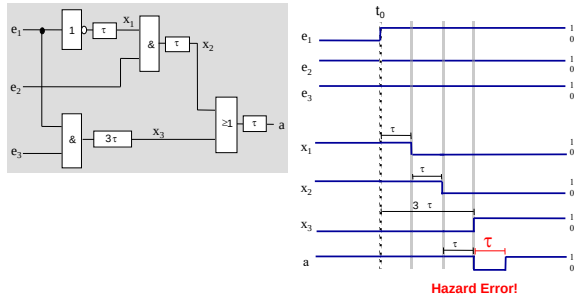

$\Rightarrow$ Correct behavior in these cases

For those transitions the value for *a* does not have to change under any circumstances, it needs to remain **constant 1** at all times

However, this behavior may not be guaranteed!



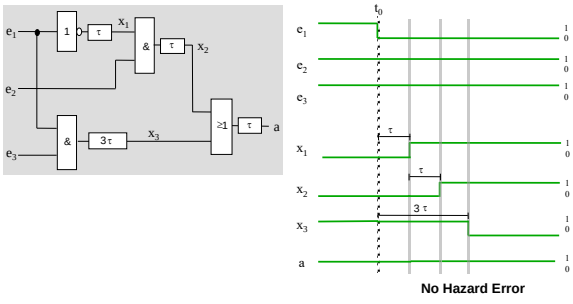
## Behavior Investigation Based on Delay Model




**Hazard Error!**



## Behavior Investigation Based on Delay Model




**No Hazard Error**



## Result and Interpretation

- For a change of input $e_1$ from 0 to 1 the output signal curve does not remain constantly at the correct function's value
  ➔ Hazard Error
- For a change of input $e_1$ from 1 to 0 the output signal curve is correct
  ➔ No worries!
- Interpretation of this example's findings:
  - Functions may inherently carry a hazard (which is instantiated in terms of the hazard error), which may only become visible if dedicated input changes are applied
  - Thus, incorrect behaviors may occur such that output values may change, although ideally that should not happen
  - Does a suitable mechanism exist (a) to find out, if a hazard error may occur and if so (b) to solve this problem in a realistic manner?



## Terminology Definitions

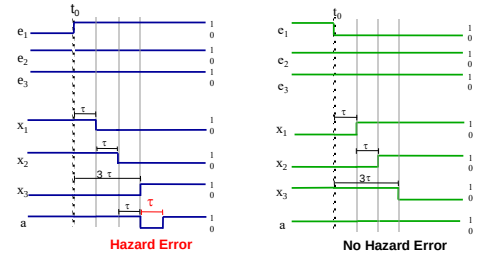
- A change of input(s) (Eingabewechsel) is defined as the change of one or multiple input configurations at a defined point in time
  - In case of a required change of multiple input configurations, these have to happen at the same point in time
- A transition (Übergang) determines the behavior of the combinatorial circuit, which is driven by the change of input(s).
  - A transitions starts at the time the input configuration changes and ends at the time the circuit has reached its static state (Ruhezustand)
- A hazard error is defined as multiple changes of an output configuration within a single transition
  - Cf. example in blue above
- A hazard is defined as the logical-structural prerequisite for a hazard error, while no detailed delay values for gates implementing the function are considered



## Hazard-afflicted Transitions

- Each hazard is a characteristics of one transition
- A hazard-afflicted transition is determined by
  - Logical function this combinatorial circuit implements
  - The structure of the same in terms of #gates and order of those (no delays)
- Given a combinatorial circuit and a transition, which ends up to be hazard-afflicted, the circuit contains a hazard
  Hazard error $\rightarrow$ Hazard
- The inverse is not correct: The existence of a hazard-afflicted transition does not lead necessarily to a hazard error
  Hazard $\wedge$ unfavorable delay values $\rightarrow$ Hazard error

## Example




Transition $(e_3,e_2,e_1)$ : (1,1,0) → (1,1,1) ist hazard-afflicted, since the potential to show a hazard error exists!



## Functional and Structural Hazards

- A functional hazard is a hazard, which originates from the function itself
  - Functional hazards always appear in any alternative combinatorial circuit implementing this function. It cannot be circumvented as such
  - Given an implementation as a combinatorial circuit, a suitable selection of delay values can prevent the functional hazard error to occur, but not the hazard itself
- A structural hazard is a hazard, which originates from the structure of the implemented circuit
  - Structural hazards can always be circumvented by developing an alternative combinatiorial circuit with a different structure
  - Thus, the selection of a different minimal disjunctive/conjunctive normal form for that function solves the problem in full



## Static 0-Hazard and 1-Hazard

- Compared to transitions, hazards are either dynamic or static
- A hazard within a static 0-transition is termed static 0-hazard
  - Examples

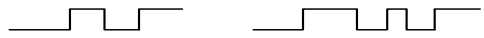
- A hazard within a static 1-transition is termed static 1-hazard
  - Examples

The transition (1,1,0) → (1,1,1) within the previous example contains a static 1-hazard

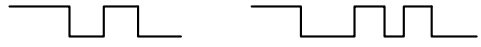


## Dynamic 01-Hazard and 10-Hazard

- A hazard within a dynamic 01-transition is termed dynamic 01-hazard
  - Examples

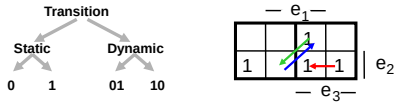


- A hazard within a dynamic 10-transition is termed dynamic 10-hazard
  - Examples





## Classifications of Hazards and Examples







**Static 1-transition:**

Transition $(e_3,e_2,e_1)$: (1,1,0) → (1,1,1)

**Dynamic 01-transition:**

Transition $(e_3,e_2,e_1)$: (0,1,1) → (1,0,1)

**Dynamic 10-transition**

Transition in inverse direction: (1,0,1) → (0,1,1)