Technische Universität Berlin

Technische Grundlagen der Informatik 1

Digitale Systeme

Technische Universität Berlin
Fakultät IV • Informatik und Elektrotechnik
Institut für Technische Informatik
FG Rechnertechnologie
Franklinstraße 28/29 • D-10587 Berlin

Vorwort

Wir haben uns in diesem Semester entschlossen ein Skript anzubieten, das mit etwas Glück diesen Namen auch verdient. Auch wenn der Inhalt des Skripts nahezu deckungsgleich mit den Folien der Vorlesung ist, so erhoffen wir uns von einem Skript dennoch Vorteile gegenüber dem Abdrucken der Folien (Handouts). Zum einen ist es natürlich kompakter, zum anderen hoffen wir, dass es auch etwas lesbarer ist und dadurch vielleicht eher einen Nachschlagecharakter hat. Beim Neugestalten haben wir auch versucht die Abbildungen aneinander anzugleichen und im Aussehen konsistenter zu machen, sodass ein höherer Wiedererkennungswert zwischen den einzelnen technischen Darstellungen in den unterschiedlichen Kapiteln gegeben ist. Bedanken wollen wir uns vor allem auch bei den Tutoren, die uns beim Umsetzen des Vorhabens unter die Arme gegriffen haben:

Sven-Garrit Czarnian
Carsten Dülsen
Thomas Faber
Matthias Hartmann
Evren Kücükbayraktar
Marcus Schubert
Jan Wetter

Wir hoffen, dass die Arbeit nicht umsonst war und wenigstens ein paar Studierenden gefällt.

Dr. Carsten Gremzow
Nico Moser

Inhaltsverzeichnis

1 Zweiwertige Logik

1.1 Grundbegriffe der Logik

1.1.1 Aussagen

Jede wörtlich formulierte Behauptung ist eine Aussage (z.B. “3 + 5 = 8”,“er läuft”). Von solchen Sätzen lässt sich im Allgemeinen bestimmen, ob sie zutreffen oder nicht, d.h. ob sie wahr oder falsch sind. Es lässt sich somit der Wahrheitswert einer Aussage bestimmen.

Definition 1. *Ein Satz, dem eindeutig ein Wahrheitswert zugeordnet werden kann, heißt Aussage. Ist A eine Aussage, dann ist W(A) der Wahrheitswert der Aussage mit: W(A) := wahr (falsch), falls Aussage A zutrifft (falls Aussage A nicht zutrifft).*

Verknüpfungen von Aussagen

Mehrere Aussagen lassen sich zu einer neuen Aussage zusammenfassen – auch dafür lässt sich ein Wahrheitswert finden. Diese Zusammenfassung von Einzelaussagen wird umgangssprachlich mit bestimmten Wörtern gebildet.

Beispiel 1. *Verknüpfungen von Aussagen*
5 ist eine Primzahl und 10 ist durch 5 teilbar.
Ist a durch 4 teilbar, dann ist a auch durch 2 teilbar.

Allgemein für A1 und A2 als zwei Aussagen gilt:

A1 und A2	Konjunktion
A1 oder A2	(einschließendes) Oder, Disjunktion
entweder A1 oder A2	ausschließendes Oder, Antivalenz
wenn A1 dann A2	Folgerung, Implikation
A1 genau dann, wenn A2	Äquivalenz

Die Bindewörter verknüpfen Aussagen. Jeder dieser aus Aussagen zusammengesetzte Satz ist selbst wieder eine Aussage. Wahrheitswerte der Einzelaussagen stehen dem Wahrheitswert der zusammengesetzten Aussagen gegenüber.

Wahrheitswert der Verknüpfung

Aussagenlogik betrachtet unter Vernachlässigung des Inhalts der Aussagen nur die Wahrheitswerte. Für entsprechende Verknüpfungen zwischen den Wahrheitswerten soll gelten:

Verknüpfung der Wahrheitswerte der Einzelaussagen =
Wahrheitswert der zusammengesetzten Aussage

Ist z. B. "A1 und A2" eine zusammengesetzte Aussage, dann soll gelten:

$$\mathsf{W(A1)\ und\ W(A2)\ =\ W(A1\ und\ A2)}$$

Diese Gleichung muss für alle möglichen Kombinationen der Wahrheitswerte von A1 und A2 gelten.

Wertetabelle, Wahrheitstabelle

Die Aussage "A1 und A2" ist nur dann eine wahre Aussage, wenn beide Einzelaussagen A1 und A2 zutreffen. Für die entsprechende Verknüpfung lässt sich eine Wertetabelle bzw. Wahrheitstabelle aufstellen. Für die Konjunktion (Symbol: $\wedge$) gilt folgende Wertetabelle:

W(A1)	W(A2)	W(A1) $\wedge$ W(A2)
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

Vereinfachung

Wahr und falsch wird durch w und f oder durch 1 und 0 dargestellt. w und f sind innerhalb der Aussagenlogik üblich, 1 und 0 in der formalen Logik. Für die Wertetabelle der Konjunktion oder UND-Verknüpfung gilt dann:

W(A1)	W(A2)	W(A1) $\wedge$ W(A2)
0	0	0
0	1	0
1	0	0
1	1	1

Logische Operationen

Entsprechend der Wertetabelle für die Konjunktion, lässt sich eine Tabelle für alle anderen zusammengesetzten Aussagen aufstellen. Diese Verknüpfungen werden auch logische Operationen genannt. Entsprechend der Anzahl der Wahrheitswerte oder Operanden, die miteinander verknüpft werden, spricht man von ein-, zwei- oder allgemein n-stelligen Operationen. Die Konjunktion ist ein Beispiel für eine zweistellige Operation.

Negation

Aussagen lassen sich auch verneinen. Dementsprechend gibt es in der Aussagenlogik die einstellige Operation der Negation (Symbol: $\neg$).

Erkenntnis

Die Aussagenlogik untersucht die formalen Zusammenhänge zwischen Aussagen. Diese können entweder wahr oder falsch sein.

1.1.2 Grundverknüpfungen

Logische Operationen

Die Grundverknüpfungen werden als logische Operationen durch Symbole dargestellt. Für die veränderlichen Operanden wird der Begriff der logischen Variablen eingeführt.

Definition 2. *Ein Zeichen, welches anstelle eines Elementes aus* $\{0, 1\}$ *steht, heißt logische Variable. Eine Zuordnung konkreter Werte zu den Variablen heißt Belegung der Variablen.*

Schaltalgebra

Eine Sonderform der Booleschen Algebra ist die Schaltalgebra. Mit ihrer Hilfe lassen sich Digitalschaltungen berechnen und vereinfachen. Die Schaltalgebra kennt Variablen und Konstanten. Es gibt nur zwei mögliche Konstanten: 0 und 1. Die Konstanten entsprechen den logischen Zuständen 0 und 1. Eine Variable kann entweder den Wert 0 oder 1 annehmen. Jede Größe, mit Wert 0 oder Wert 1, stellt eine Variable dar. Ausdrücke wie $(a \wedge b)$, die z. B. aus zwei Variablen bestehen, können wie eine Variable behandelt werden, denn ihr Wert kann ebenfalls nur 0 oder 1 sein. Eine Variable der Schaltalgebra ist also eine binäre Größe.

Operation	Symbolik	Bindewörter	Wahrheitstabelle
Negation	$\overline{a}$	nicht a	a | $\overline{a}$ 0 | 1 1 | 0
Konjunktion	$a \wedge b, a \cdot b$	a und b	a b | $a \wedge b$ 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
Disjunktion	$a \vee b, a + b$	a oder b	a b | $a \vee b$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Äquivalenz	$a \equiv b, a \leftrightarrow b$	a äquivalent b	a b | $a \equiv b$ 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1
Antivalenz	$a \not\equiv b, a \oplus b$	a antivalent b	a b | $a \not\equiv b$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Implikation	$a \rightarrow b$	a impliziert b	a b | $a \rightarrow b$ 0 0 | 1 0 1 | 1 1 0 | 0 1 1 | 1
Sheffer-Funktion (NAND)	$\overline{a \wedge b}, \overline{a \cdot b}, a \mid b$	a und b nicht	a b | $\overline{a \wedge b}$ 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
Peirce-Funktion (NOR)	$\overline{a \vee b}, \overline{a + b}, a \downarrow b$	weder a noch b	a b | $\overline{a \vee b}$ 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

2-stellige Operationen

Neben den Grundverknüpfungen sind weitere Verknüpfungen zwischen a und b denkbar. In einer Wertetabelle für zwei Operanden kann jede beliebige Kombination von

a	b		$a < b$		a		b	$a \not\equiv b$	$a > b$	$\overline{a > b}$	$a \equiv b$	$\overline{b}$		$\overline{a}$	$a \rightarrow b$	$\overline{a < b}$	
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Tabelle 1.1: mögliche Kombinationen 2-stelliger logischer Operationen

0-1-Werten eingesetzt werden, was eine zweistellige Operation ergibt. Nicht alle diese Operationen haben einen Namen oder eine symbolische Bezeichnung. Es ergeben sich $2^{2^2} = 16$ Möglichkeiten (siehe Tabelle 1.1).

1-stellige Operationen

Für einstellige Operationen, wie z.B. die Negation ergeben sich $2^{2^1} = 4$ Möglichkeiten:

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

0-stellige Operationen

Konstanten werden als nullstellige Operationen bezeichnet. Es existieren $2^{2^0} = 2$ verschiedene Anordnungen von 0 und 1. Die Konstanten haben somit die Werte 0 (Kontradiktion/nie) oder 1 (Tautologie/immer).

1.1.3 Ausdrücke

n-stellige Operationen

Verknüpfungen von nur zwei Operanden sind im Allgemeinen nicht ausreichend und bei mehrstelligen Operationen wächst die Zahl der möglichen Verknüpfungen rasch an. Bei n-stelligen Operationen ergeben sich 2^n verschiedene Möglichkeiten für die Belegung der Variablen. Diesen Belegungen können wieder 2^{2^n} 0-1-Kombinationen als Ergebnisse zugeordnet werden. Mehr als zweistellige Operationen lassen sich durch 1- und 2-stellige Operationen ausdrücken. Die Grundverknüpfungen genügen, um sämtliche weiteren logischen Zusammenhänge zu beschreiben. Dazu kombiniert man verschiedene Operationen, was zum Begriff des Ausdrucks führt.

Definition 3. *Eine Verknüpfung von endlich vielen Konstanten und logischen Variablen mittels Grundverknüpfungen heißt ein Ausdruck. Die Reihenfolge, in der Operationen anzuwenden sind, wird durch Klammern bestimmt. Ausdrücke die nur aus den Operationen $\overline{\ }$ (Negation), $\wedge$ (Konjunktion) und $\vee$ (Disjunktion) zusammengesetzt sind, heißen boolesche Ausdrücke.*

Beispiel 2. *Ausdrücke*
$\overline{a}$
$(a \wedge b) \vee 1$
$(a \rightarrow b) \wedge (c \downarrow a)$
$((((a \vee \overline{b} \leftrightarrow c) \oplus a) \vee c)$
$((a \wedge b) \vee (\overline{a} \wedge \overline{b}))$
$(((\overline{a \wedge b}) \vee (b \wedge \overline{c})) \wedge 1$
kein Ausdruck ist:
$a \rightarrow b \oplus c$
weil die Reihenfolge der Operationen nicht eindeutig ist

Ermittlung des Wahrheitswerts eines Ausdrucks

Die aktuellen Variablenwerte werden in den Ausdruck eingesetzt und schrittweise der Klammerung entsprechend abgearbeitet. Dabei werden jeweils die Grundverknüpfungen durch den Wahrheitswert ersetzt, der durch die zugehörige Wertetabelle gegeben ist.

Beispiel 3. *Mit* $a = 1$, $b = 0$, $c = 1$ *gilt für den Wahrheitswert des Ausdrucks* $(a \rightarrow b) \wedge (c \downarrow a)$ *:*

$$(a \rightarrow b) \wedge (c \downarrow a) \;=\; (1 \rightarrow 0) \wedge (1 \downarrow 1) \;=\; 0 \wedge 0 \;=\; 0$$

Soll der Ausdruck vollständig durch eine Wertetabelle beschrieben werden, ist es meist sinnvoll, auch Zwischenschritte mit in der Tabelle zu notieren.

Beispiel 4. *Die Verknüpfung* $(a \rightarrow b) \wedge (c \downarrow a)$ *soll durch eine Wertetabelle vollständig beschrieben werden:*

a	b	c	$(a \rightarrow b)$	$(c \downarrow a)$	$(a \rightarrow b) \wedge (c \downarrow a)$
0	*0*	*0*	*1*	*1*	*1*
0	*0*	*1*	*1*	*0*	*0*
0	*1*	*0*	*1*	*1*	*1*
0	*1*	*1*	*1*	*0*	*0*
1	*0*	*0*	*0*	*0*	*0*
1	*0*	*1*	*0*	*0*	*0*
1	*1*	*0*	*1*	*0*	*0*
1	*1*	*1*	*1*	*0*	*0*

Äquivalente Ausdrücke

Definition 4. *Zwei Ausdrücke A1, A2 heißen äquivalent (A1 = A2) genau dann, wenn sie bei gleicher Belegung gemeinsamer Variablen stets gleiche Wahrheitswerte annehmen. A1 = A2 heißt dann eine logische Gleichung.*

Dabei ist nicht gefordert, dass beide Ausdrücke genau dieselben Variablen enthalten.

Beispiel 5. *Seien* $\mathsf{a}, \mathsf{b}, \mathsf{c}$ *logische Variable.*
Behauptung: $\mathsf{a} \vee (\mathsf{b} \wedge \overline{\mathsf{b}}) = \mathsf{a} \vee (\mathsf{c} \wedge \overline{\mathsf{c}})$
Beweis (durch Wertetabelle):

a	b	c	$(\mathsf{a} \vee (\mathsf{b} \wedge \overline{\mathsf{b}}))$	$\mathsf{a} \vee (\mathsf{c} \wedge \overline{\mathsf{c}})$
0	*0*	*0*	*0*	*0*
0	*0*	*1*	*0*	*0*
0	*1*	*0*	*0*	*0*
0	*1*	*1*	*0*	*0*
1	*0*	*0*	*1*	*1*
1	*0*	*1*	*1*	*1*
1	*1*	*0*	*1*	*1*
1	*1*	*1*	*1*	*1*

Beide Ausdrücke hängen weder von b noch von c ab, und gerade dann gilt auch nur die Äquivalenz unter derartigen Ausdrücken. Solche überflüssigen (redundante oder fiktive) Variablen können eliminiert werden.

Vorrangregeln

Bei komplexeren Ausdrücken kann die Zahl der Klammern stark zunehmen. Zur Vermeidung zahlreicher Klammern werden unter den wichtigsten Grundverknüpfungen, Vorrangregeln eingeführt. Das geschieht ähnlich wie unter den arithmetischen Rechenarten („Punktrechnung vor Strichrechnung", etc.).

Definition 5. *Bei der Ausführung von Operationen in logischen Ausdrücken gelten folgende Prioritäten: Geklammerte Verknüpfungen haben Vorrang vor Negation* ($\overline{\ }$) *hat Vorrang vor Konjunktion* ($\wedge$) *hat Vorrang vor allen anderen Operationen. Reicht das Negationszeichen* ($\overline{\ }$) *über mehrere Variable oder Konstante, gelten diese als geklammert und die Negation gilt für den Klammerausdruck. Bei mehreren aufeinanderfolgenden zweistelligen Grundverknüpfungen gleicher Vorrangstufe, also bei Hintereinanderausführung gleicher Verknüpfungen, wird von links nach rechts abgearbeitet. Das Konjunktionszeichen kann weggelassen werden, wenn keine Verwechslungen mit anderen Variablen möglich sind.*

Beispiel 6. *Seien* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$ *logische Variablen.*
$\mathsf{a} \vee (\mathsf{b} \wedge \mathsf{c}) = \mathsf{a} \vee \mathsf{b} \wedge \mathsf{c} = \mathsf{a} \vee \mathsf{bc}$,
$(\mathsf{a} \wedge \overline{\mathsf{b}}) \vee (\overline{\mathsf{c} \wedge \mathsf{d}}) = \mathsf{a}\overline{\mathsf{b}} \vee \overline{\mathsf{c}}\overline{\mathsf{d}}$,
$\mathsf{a} \to \mathsf{b} \to \mathsf{c} = (\mathsf{a} \to \mathsf{b}) \to \mathsf{c}$,
$(\mathsf{a} \vee \mathsf{b}) \wedge (\mathsf{a} \vee \mathsf{c}) = (\mathsf{a} \vee \mathsf{b})(\mathsf{a} \vee \mathsf{c})$,
aber:
$\overline{\mathsf{a}} \wedge \overline{\mathsf{b}} = \overline{\mathsf{a}}\ \overline{\mathsf{b}} \neq \overline{\mathsf{a}}\overline{\mathsf{b}}$

1.1.4 Umformung logischer Ausdrücke

Regeln zur Vereinfachung und Umformung gegebener Ausdrücke

Es wird sich auf boolesche Ausdrücke beschränkt, weil es für diese Ausdrücke eine entsprechende algebraische Struktur gibt, nämlich die Boolesche Algebra. Andererseits lassen sich alle anderen Operationen als boolesche Ausdrücke schreiben. Beweisen lassen sich diese Regeln in einfachster Weise durch Vergleich der Wertetabellen der Ausdrücke auf beiden Seiten der Gleichung.

Satz 1. *Seien* $\mathsf{a}, \mathsf{b}, \mathsf{c}$ *logische Variablen, dann gelten folgende Regeln:*

Name	*Gesetzmäßigkeit*
Negation der Negation	$\overline{\overline{\mathsf{a}}} = \mathsf{a}$
Kommutativgesetz	$\mathsf{a} \wedge \mathsf{b} \wedge \mathsf{c} = \mathsf{c} \wedge \mathsf{a} \wedge \mathsf{b}$ $\mathsf{a} \vee \mathsf{b} \vee \mathsf{c} = \mathsf{c} \vee \mathsf{a} \vee \mathsf{b}$
Assoziativgesetz	$\mathsf{a} \wedge (\mathsf{b} \wedge \mathsf{c}) = (\mathsf{a} \wedge \mathsf{b}) \wedge \mathsf{c}$ $\mathsf{a} \vee (\mathsf{b} \vee \mathsf{c}) = (\mathsf{a} \vee \mathsf{b}) \vee \mathsf{c}$
Distributivgesetz	$\mathsf{a} \wedge (\mathsf{b} \vee \mathsf{c}) = (\mathsf{a} \wedge \mathsf{b}) \vee (\mathsf{a} \wedge \mathsf{c})$ $\mathsf{a} \vee (\mathsf{b} \wedge \mathsf{c}) = (\mathsf{a} \vee \mathsf{b}) \wedge (\mathsf{a} \vee \mathsf{c})$
Idempotenzgesetz	$\mathsf{a} \wedge \mathsf{a} = \mathsf{a}$ $\mathsf{a} \vee \mathsf{a} = \mathsf{a}$
Komplementgesetz	$\mathsf{a} \wedge \overline{\mathsf{a}} = 0$ $\mathsf{a} \vee \overline{\mathsf{a}} = 1$
0-1-Gesetz	$\mathsf{a} \wedge 1 = \mathsf{a}$ $\mathsf{a} \wedge 0 = 0$ $\mathsf{a} \vee 1 = 1$ $\mathsf{a} \vee 0 = \mathsf{a}$
Absorptionsgesetze	$\mathsf{a} \wedge (\mathsf{a} \vee \mathsf{b}) = \mathsf{a}$ $\mathsf{a} \vee (\mathsf{a} \wedge \mathsf{b}) = \mathsf{a}$ $(\mathsf{a} \wedge \mathsf{b}) \vee (\mathsf{a} \wedge \overline{\mathsf{b}}) = \mathsf{a}$ $(\mathsf{a} \vee \overline{\mathsf{b}}) \wedge \mathsf{b} = \mathsf{a} \wedge \mathsf{b}$ $(\mathsf{a} \wedge \overline{\mathsf{b}}) \vee \mathsf{b} = \mathsf{a} \vee \mathsf{b}$ $(\mathsf{a} \vee \mathsf{b}) \wedge (\mathsf{a} \vee \overline{\mathsf{b}}) = \mathsf{a}$
De Morgan'sche Regeln	$\overline{\mathsf{a} \wedge \mathsf{b}} = \overline{\mathsf{a}} \vee \overline{\mathsf{b}}$ $\overline{\mathsf{a} \vee \mathsf{b}} = \overline{\mathsf{a}} \wedge \overline{\mathsf{b}}$

Beweis 1. *Der Beweis erfolgt durch Wertetabelle. Exemplarisch soll er hierfür das vorletzte Absorptionsgesetz durchgeführt werden.*
$(a \wedge \overline{b}) \vee b = a \vee b$

a	b	$a \wedge \overline{b}$	$(a \wedge \overline{b}) \vee b$	$a \vee b$
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*1*
1	*0*	*1*	*1*	*1*
1	*1*	*0*	*1*	*1*

Die Spalten für $(a \wedge \overline{b}) \vee b$ *und* $a \vee b$ *stimmen überein (was zu zeigen war).*

Vereinfachung komplexer boolescher Ausdrücke

Die angeführten Regeln erlauben, komplizierte boolesche Ausdrücke häufig zu vereinfachen.

Beispiel 7. $(\overline{a} \vee b)\overline{a\ b}((\overline{a} \vee b)(\overline{a} \vee \overline{b}) \vee (ca \vee \overline{c}))$ *De Morgan'sche Regel*
$= (\overline{a} \vee b)(\overline{a} \vee \overline{b})((\overline{a} \vee b)(\overline{a} \vee \overline{b}) \vee (ca \vee \overline{c}))$ *Absorptionsgesetz*
$= \overline{a}(\overline{a} \vee (ca \vee \overline{c}))$ *Absorptionsgesetz*
$= \overline{a}$

Beispiel 8. $x \vee yz \vee (\overline{x} \wedge \overline{yz})$
$= yz \vee x \vee (\overline{x} \wedge \overline{y}z)$
$= yz \vee (x \vee \overline{x}) \wedge (x \vee \overline{yz})$
$= yz \vee 1 \wedge (x \vee \overline{yz})$
$= yz \vee (x \vee \overline{yz})$
$= yz \vee \overline{yz} \vee x$
$= 1 \vee x$
$= 1$

Logische Gleichungen lassen sich in analoger Weise beweisen, denn bei mehr als 3 Variablen wird der Beweis mittels Wertetabelle recht aufwändig.

1.1.5 Formal wahre, formal falsche Ausdrücke

Ein häufig beschrittener Weg zur Vereinfachung von Ausdrücken ist die Suche nach "Teilausdrücken", die immer logisch 0 oder immer 1 sind, um anschließend die 0-1-Gesetze anzuwenden.

Definition 6. *Ein Ausdruck heißt formal wahrer Ausdruck, wenn er für jede Belegung den Wahrheitswert 1 liefert. Ein Ausdruck heißt formal falsch, wenn er für jede Belegung den Wahrheitswert 0 liefert.*

Beispiel 9.

Formal wahre Ausdrücke:	*Formal falsche Ausdrücke:*
$a \vee \overline{a} = 1$	$a \wedge \overline{a} = 0$
$ab \vee a\overline{b} \vee \overline{a}b \vee \overline{a}\overline{b} = 1$	$\overline{(ab \vee a\overline{b} \vee \overline{a}b \vee \overline{a}\overline{b})} = 0$
$(a \to b) \vee (b \to a) = 1$	$a \oplus a = 0$
$(a \leftrightarrow b) \vee (\overline{a} \leftrightarrow b) = 1$	$a \wedge b \wedge c \wedge 0 = 0$
$(a \wedge b) \vee 1 = 1$	

Tautologie, Kontradiktion

Bei formal wahren/falschen Ausdrücken handelt es sich in der Aussagenlogik um Aussagen, die immer zutreffen müssen (formal wahr) oder nie zutreffen können (formal falsch). Die Behauptung "n=m oder n $\neq$ m" ($a \vee \overline{a}$) trifft immer zu, unabhängig von den Werten der Einzelaussagen "(n = m)" und "(n $\neq$ m)". Immer falsch dagegen ist die Aussage "n=m und n $\neq$ m" ($a \wedge \overline{a}$). Also schon aus der Form des Ausdrucks lässt sich der Wahrheitswert bestimmen. Solche Ausdrücke werden auch Tautologien oder Allgemeingültigkeiten genannt, wenn sie formal wahr sind, bzw. Kontradiktionen im Falle formal falscher Ausdrücke.

Einfache Sätze zur Umformungen von Ausdrücken

Der folgende Satz ist eine Verallgemeinerung der De Morgan'schen Regel.

Satz 2. *Ein boolescher Ausdruck lässt sich negieren, indem man alle Operatoren $\vee$ durch $\wedge$ und alle $\wedge$ durch $\vee$ ersetzt und die Konstanten und Variablen negiert. Die Klammerungen bleiben dabei erhalten.*

Mit Hilfe de DeMorgan'schen Regeln lässt sich die Gültigkeit gut veranschaulichen:

Beispiel 10.
$$\overline{(a_1 \wedge a_2 \wedge a_3 \wedge ... a_n)} = \overline{a_1} \vee \overline{a_2} \vee \overline{a_3} \vee ... \vee \overline{a_n}$$
$$\overline{(a_1 \vee a_2 \vee a_3 \vee ... \vee a_n)} = \overline{a_1} \wedge \overline{a_2} \wedge \overline{a_3} \wedge ... \wedge \overline{a_n}$$
$$\overline{((a_1a_2 \vee \overline{a_2}a_3)(a_4 \vee \overline{a_5}))} = (\overline{a_1} \vee \overline{a_2})(a_2 \vee \overline{a_3}) \vee \overline{a_4}a_5$$
$$\overline{((1 \vee a_1a_2) \wedge ((\overline{a_1} \wedge 1) \vee 0)} = (0 \wedge (\overline{a_1} \vee \overline{a_2})) \vee ((a_1 \vee 0) \wedge 1)$$
$$= 0(\overline{a_1} \vee \overline{a_2}) \vee \overline{(a_1 \vee 0)1}$$
$$(\overline{a_1}\ \overline{a_2} \vee \overline{a_1}a_2 \vee a_1\overline{a_2} \vee a_1a_2) = (a_1 \vee a_2)(a_1 \vee \overline{a_2})(\overline{a_1} \vee a_2)(\overline{a_1} \vee \overline{a_2})$$

Beweis 2.
$$\overline{(a_1a_2a_3...a_n)} = \overline{((a_1a_2a_3...a_{n-1})a_n)} \text{ } \textit{(wegen Vorrangr.)}$$
$$= \overline{(a_1a_2...a_{n-1})} \vee \overline{a_n} \text{ } \textit{(De Morgansche Regel)}$$
$$= \overline{((a_1...a_{n-2})a_{n-1})} \vee \overline{a_n}$$
$$= \overline{a_1} \vee \overline{a_2} \vee \overline{a_3} \vee ... \vee \overline{a_n}$$

Satz 3. *Eine Konjunktion $a_1 \wedge a_2 \wedge a_3 \wedge .... \wedge a_n$ ist für eine Belegung $(a_1, a_2, ..., a_n)$ falsch genau dann, wenn es (mindestens) ein i gibt mit $a_i = 0, i \in 1, 2, ..., n$. Eine Disjunktion $a_1 \vee a_2 \vee ... \vee a_n$ ist für eine Belegung $(a_1, a_2, ..., a_n)$ richtig genau dann, wenn es (mindestens) ein i gibt mit $a_i = 1, i \in 1, 2, ..., n$.*

Der Satz kann natürlich auch umgekehrt formuliert werden:

Satz 4. *Eine Konjunktion* $a_1 \wedge a_2 \wedge .. \wedge a_n$ *ist genau dann wahr, wenn alle Variablen* a_i *mit 1 belegt sind. Eine Disjunktion* $a_1 \vee a_2 \vee .. \vee a_n$ *ist genau dann falsch, wenn alle Variablen* a_i *mit 0 belegt sind.*

Diese Aussagen werden später bei der Behandlung der Normalformen benötigt.

1.2 Logische Funktionen und Funktionsbündel

1.2.1 Logische Funktionen

Begriff der logischen Funktionen

Ein logischer Ausdruck beschreibt eine Zuordnung zwischen einer Belegung der Variablen und den Wahrheitswerten 0 und 1. Das Ergebnis ist eine Funktion der (unabhängigen) Variablen.

Definition 7. *n-stellige logische Funktion*
Sei $D \subset \{0,1\}^n, n \in \mathbb{N}$ *und* $(a_1, a_2, ..., a_n) \in D$ *eine Belegung.* $\{0,1\}^n$ *ist Menge aller* $n-$Tupel $(a_1, a_2, ..., a_n)$ *mit* $a_i \in \{0,1)\}$ *für alle* $i = 1, 2, ..., n$ *Eine Zuordnungsvorschrift* $f : D \to \{0,1\}$ *mit* $(a_1, \ldots, a_n) \mapsto f(a_1, a_2, \ldots, a_n)$ *heißt n-stellige logische Funktion oder logische Funktion genau dann, wenn f eindeutig ist, d.h. jeder Belegung* $(a_1, a_2, ..., a_n) \in D$ *wird genau ein Wert* $f(a_1, a_2, ..., a_n) \in \{0,1\}$ *zugeordnet.* $f(a_1, a_2, ..., a_n)$ *heißt Funktionswert, D heißt Definitionsbereich, die Elemente aus D heißen Argumente der Funktion. Ist* $D = \{0,1\}^n$*, so heißt f vollständig definiert. Ist* $D \subset \{0,1\}^n$*, so heißt f partiell definiert.*

Partiell definierte logische Funktionen

Eine logische Funktion muss also nicht für jede Belegung ihrer Variablen definiert sein ($D \subset \{0,1\}^n$, partiell definierte logische Funktionen), was später im Zusammenhang mit Don't-care-Bedingungen in der Schaltalgebra interessant wird. Eine logische Funktion kann praktisch durch eine Wertetabelle (auch Funktionstabelle genannt) oder durch einen logischen Ausdruck angegeben werden.

Beispiel 11. *Vollständig definierte Funktionen*

x_1	x_2	$g(x_1, x_2)$
0	*0*	*0*
0	*1*	*1*
1	*0*	*0*
1	*1*	*0*

$f(a, b, c) = \overline{a \vee b} \to c$

Durch logische Ausdrücke können nur vollständig definierte Funktionen angegeben werden.

Funktionsgleichung

Eine Funktion kann auch durch eine abhängige Variable definiert werden, die selbst in einer Funktion als Variable auftritt. Auf diese Art lassen sich logische Funktionen verschachteln oder verknüpfen.

Beispiel 12. *Funktionsgleichung*
$$\begin{aligned} y &= f(a,b) \\ z &= g(a,c,y) \\ &= g(a,c,f(a,b)) \end{aligned}$$

Man spricht hierbei auch von Funktionsgleichungen. Diese werden im Zusammenhang mit der Analyse und Synthese von größeren Schaltnetzen zur besseren Übersicht benutzt.

Dezimales Äquivalent

Unter der Voraussetzung, nur vollständig definierte Funktionen zu betrachten, gilt, dass es 2^{2^n} verschiedene n-stellige Funktionen gibt. Wegen der hohen Anzahl hat man Kurzschreibweisen für diese Funktionen eingeführt. Es wird der Begriff des dezimalen Äquivalents einer Belegung (oder eines logischen Vektors) eingeführt.

Definition 8. *Dezimales Äquivalent*
Sei $(a_1, \ldots, a_n)$ *eine Folge logischer Konstanten.* $\delta(a_1 a_2 \ldots a_n) = i \in \mathbb{N}_0$ *heißt das dezimale Äquivalent zu* $(a_1, a_2, \ldots, a_n)$*, wobei*
$i = \sum_{j=0}^{n-1} b_j 2^j$ und $b_j = \begin{array}{l} 1 \; falls \; a_{n-j} = 1 \\ 0 \; falls \; a_{n-j} = 0 \end{array}$ mit $b_j \in \mathbb{N}_0$
$\mathbb{N}_0$: *Menge der natürlichen Zahlen, einschließlich der Null*

Die Definition sagt, dass die Folge logischer Werte so wie sie dasteht als Dualzahl $b_{n-1} b_{n-2} \ldots b_0$ aufgefasst wird und als natürliche Zahl im Dezimalsystem wiedergegeben wird. Umweg über b_j ist nötig, um Konfusionen zwischen den Wahrheitswerten 0, 1 und den natürlichen Zahlen 0, 1 zu vermeiden. Die lexikographische Anordnung der Belegungen logischer Variablen bedeutet nur, die Belegungen als Dualzahlen zu lesen und sie dann in aufsteigender Reihenfolge zu schreiben (siehe Kapitel 2).

Beispiel 13. *Dezimale Äquivalente*
$\delta(000) = 0$
$\delta(0010) = 2$
$\delta(10010) = 18$

1.2.2 Vollständige Systeme

Darstellung der Grundverknüpfungen

Eine logische Funktion kann durch verschiedene zueinander äquivalente Ausdrücke angegeben werden.

Satz 5. *Äquivalenzen zu Grundverknüpfungen*
Seien x_1, x_2 *logische Variable, dann gilt:*
$f(x_1, x_2) = x_1 \wedge x_2 = \overline{(\overline{x_1} \vee \overline{x_2})}$
$f(x_1, x_2) = x_1 \oplus x_2 = \overline{x_1}x_2 \vee x_1\overline{x_2} = \overline{(x_1 \vee \overline{x_2})} \vee \overline{(\overline{x_1} \vee x_2)}$
$f(x_1, x_2) = x_1 \vee x_2 = \overline{(\overline{x_1} \wedge \overline{x_2})}$
$f(x_1, x_2) = \overline{x_1 \vee x_2} = \overline{x_1} \wedge \overline{x_2}$
$f(x_1, x_2) = x_1 \leftrightarrow x_2 = \overline{x_1x_2} \vee x_1x_2 = \overline{(x_1 \vee x_2)} \vee \overline{(\overline{x_1} \vee \overline{x_2})}$
$f(x_1, x_2) = x_1 \rightarrow x_2 = \overline{(x_1 \vee \overline{x_2})} = \overline{x_1} \vee x_2$
$f(x_1, x_2) = \overline{x_1 \wedge x_2} = \overline{x_1} \vee \overline{x_2}$

Beweisen lassen sich diese Behauptungen durch Wertetabellen und die logischen Gesetze. Die Gleichungen zeigen, dass sich alle Grundverknüpfungen durch $\wedge, \vee, ^-$ ausdrücken lassen, dass sogar $\vee$ und $^-$ genügen, um alle Grundverknüpfungen darzustellen.

Satz 6. *Boolescher Ausdruck*
Seien $a_1, a_2, \ldots, a_n$ *logische Variable. Jede vollständig definierte logische Funktion* $g(a_1, a_2, \ldots, a_n)$ *lässt sich als logischer Ausdruck, der nur aus den Grundverknüpfungen* $\wedge, \vee, ^-$ *zusammengesetzt ist, d.h. als boolescher Ausdruck angeben.*

Definition 9. *Eine Menge logischer Operationen* $(O_1, O_2, \ldots, O_k)$ *heißt vollständiges System, wenn sich jede vollständig definierte logische Funktion nur aus diesen Operationen bilden lässt.*

Durch $(\wedge, \vee, ^-)$ sind alle booleschen Ausdrücke erfasst. Mit booleschen Ausdrücken lassen sich also alle logischen Zusammenhänge beschreiben. Es gibt natürlich noch weitere vollständige Systeme. Um Vollständigkeit zu beweisen genügt es zu zeigen, dass mit den betrachteten Operationen die Grundverknüpfungen $\wedge, \vee, ^-$ ausgedrückt werden können.

Satz 7. *Vollständige Systeme*
$(\wedge, ^-)$ *und* $(\vee, ^-)$ *sind vollständige Systeme.*

Beweis 3. *Seien* a, b *logische Variablen.*
$a \vee b$ *bzw.* $a \wedge b$ *lassen sich mit* $\wedge$ *und* $^-$ *bzw.* $\vee$ *und* $^-$ *angeben.*
$(\wedge, ^-)$*:* $a \vee b = \overline{\overline{a} \wedge \overline{b}}$ *(De Morgansche Regel)*
$(\vee, ^-)$*:* $a \wedge b = \overline{\overline{a} \vee \overline{b}}$ *(De Morgansche Regel)*

Satz 8. *NOR und NAND bilden jeweils ein vollständiges System.*

Mit einer einzigen Verknüpfung lassen sich alle vollständig definierten Funktionen ausdrücken, was die technische Realisierung vereinfacht. Es muss prinzipiell nur ein Verknüpfungsglied gebaut werden. Allerdings erhöht sich die Anzahl der Verknüpfungsglieder dadurch in der Regel. Dennoch genießt das System $(\wedge, \vee, ^-)$ eine gewisse Vorzugsstellung, weil es über diese Verknüpfungen eine algebraische Struktur gibt.

1.2.3 Funktionsbündel

In der Praxis sind häufig Funktionsbündel zu betrachten. Es handelt sich dabei um eine (endliche) Menge logischer Funktionen, die das gleiche Argument haben.

Beispiel 14.
Funktionsbündel werden z.B. bei der Umcodierung eines Binärcodes in einen anderen, wie es z.B. bei der Umformung des BCD-Codes in den 1-aus-10-Code der Fall ist, benötigt. Zur Beschreibung der Ausgänge werden zehn verschiedene Funktionen benötigt, die vier gemeinsame Variablen an den Eingängen haben. z. B.
$y_1 = x_1x_2x_3 \vee x_1x_4$
$y_2 = x_1\overline{x_4} \vee x_1x_2x_3$

Definition 10. *Eine Zusammenfassung σ logischer Funktionen $f_1, f_2, \ldots, f_m$, die das gleiche Argument ($a_1, \ldots, a_n$) haben ($m, n \in \mathbb{N}$), heißt ein Funktionsbündel oder eine Vektorfunktion.*
Schreibweise:
$\sigma(a_1, a_2, \ldots, a_n) = (y_1, y_2, \ldots, y_m)$

$$\begin{array}{ll} & y_1 = f_1(a_1, \ldots, a_n) \\ mit & y_2 = f_2(a_1, \ldots, a_n) \\ & \ldots \\ & y_m = f_m(a_1, \ldots, a_n) \end{array}$$

Darstellung von Funktionsbündeln

Die Darstellung von Funktionsbündeln kann durch eine Anzahl m verschiedener logischer Ausdrücke gegeben sein oder durch eine Wertetabelle.

Beispiel 15. *Funktionsbündel* $\sigma(a, b, c) = (x, y, z, t)$

a	b	c	x	y	z	t
0	*0*	*0*	*1*	*1*	*1*	*0*
0	*0*	*1*	*1*	*1*	*1*	*1*
0	*1*	*0*	*1*	*0*	*1*	*0*
0	*1*	*1*	*0*	*0*	*1*	*0*
1	*0*	*0*	*0*	*0*	*0*	*1*
1	*0*	*1*	*1*	*0*	*0*	*0*
1	*1*	*0*	*1*	*0*	*0*	*0*
1	*1*	*1*	*1*	*1*	*0*	*1*

Anwendung

Durch das Zusammenfassen mehrerer Funktionen zu einem Funktionsbündel kann beispielsweise der Hardwareaufwand minimiert werden. Gibt es in mehreren der beteiligten Funktionen (logischen Ausdrücken) gemeinsame Teile, kann dieser Teil von mehreren Funktionen gleichzeitig benutzt werden. Dies ermöglicht eine Einsparung von Hardware

und Verlustleistung bei der technischen Realisierung. Bisher existiert kein allgemeiner Algorithmus zur effizienten Berechnung optimaler Lösungen von mehrstufigen Schaltnetzen.

1.3 Das Karnaugh-Veitch-Diagramm

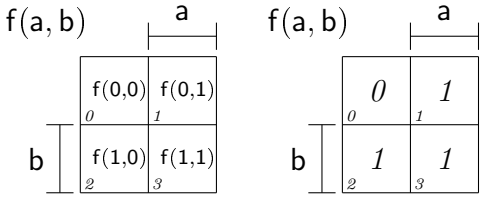
Graphische Darstellung

Zur Veranschaulichung logischer Funktionen eignen sich graphische Darstellungen bzw. Diagramme. Eine gute Verbildlichung ist das Karnaugh-Veitch-Diagramm (KV-Diagramm). Diese grafische Form ist vorteilhaft bei der Behandlung der Normalformen, die daraus zum Teil direkt abgelesen werden können. Das KV-Diagramm ist eine Matrix von Wahrheitswerten. Jedem Element der Matrix ist eindeutig eine Belegung der Variablen der Funktionen zugeordnet, die am Rand des Schemas eingetragen werden.

Beispiel 16. *KV-Diagramm mit zwei Variablen*

$\delta(ba)$	b	a	f(a, b)
0	*0*	*0*	*0*
1	*0*	*1*	*1*
2	*1*	*0*	*1*
3	*1*	*1*	*1*

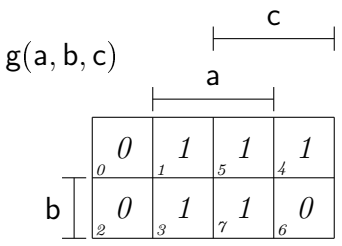
$f(a, b) = a \vee b$



Beispiel 17. *KV-Diagramm mit drei Variablen*

$\delta(cba)$	c	b	a	g(a, b, c)
0	*0*	*0*	*0*	*0*
1	*0*	*0*	*1*	*1*
2	*0*	*1*	*0*	*0*
3	*0*	*1*	*1*	*1*
4	*1*	*0*	*0*	*1*
5	*1*	*0*	*1*	*1*
6	*1*	*1*	*0*	*0*
7	*1*	*1*	*1*	*1*

$g(a, b, c) = ab \vee a\overline{b} \vee \overline{a}\overline{b}c$



Anordnung bei mehr als einer Variable pro Zeile /Spalte

Bei mehr als einer Variable pro Spalte oder Zeile kann man die Bezeichnung auch als Vektor der Eingangvariablen betrachten (im Beispiel a und c bei den Spalten). Die Belegungen der Vektoren werden nicht lexikographisch (d.h. entsprechend dem Dualcode) angeordnet, sondern entsprechend dem Graycode ($[a, c]_{Spalte} \Rightarrow [0, 0]_0, [0, 1]_1, [1, 1]_2, [1, 0]_3$) ähnlich wie der Dualcode ist auch der Graycode eine Codierung mit 0 und 1. Der Graycode hat aber die Eigenschaft, sich in der Darstellung von benachbarten Zahlen nur in

einer Stelle zu unterscheiden (Hammingdistanz gleich 1). In den Zeilen bzw. Spalten stehen nur Belegungen nebeneinander, die sich nur in einer Stelle unterscheiden. Das gilt sogar zwischen dem ersten und letzten Element einer Zeile bzw. Spalte. Man kann Anfang und Ende einer Zeile oder Spalte als benachbart ansehen. Gerade diese systematische Anordnung ermöglicht die Benutzung dieser Diagramme zur Vereinfachung logischer Ausdrücke (Minimierung). Mit Zunahme der Variablenzahl werden die Matrizen abwechselnd nach unten und rechts "umgeklappt", wodurch sich die Zahl ihrer Elemente jeweils verdoppelt.

1-Muster oder 0-Muster

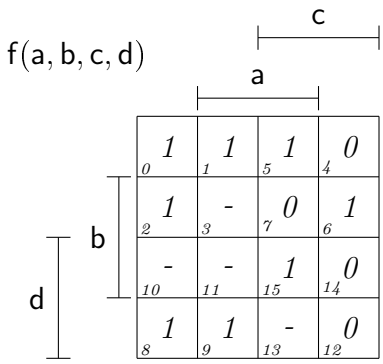
Um Wahrheitswerte eindeutig in ein KV-Diagramm einzutragen, genügt es, entweder nur die 1-Werte oder nur die 0-Werte einzutragen. Es entsteht ein sogn. 1-Muster oder 0-Muster. Vielfach schraffiert man auch nur die Kästchen, die eine 1 tragen.

Übertragung einer Funktion in ein KV-Diagramm

Eine Funktion kann durch Wertetabelle oder logischen Ausdruck angegeben werden. Liegt die Funktion als Wertetabelle vor, ist die Übertragung in das KV-Diagramm einfach. Man sucht zu jeder Belegung die entsprechende Stelle und trägt den Wahrheitswert der Funktion ein. Bei unvollständig definierten Funktionen, werden in den entsprechenden Stellen "-" ("don't-care") eingetragen

Beispiel 18. *Nicht vollständig definierte Funktion*

δ(cba)	d	c	b	a	f(a, b, c, d)
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	-
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	-
11	1	0	1	1	-
12	1	1	0	0	0
13	1	1	0	1	-
14	1	1	1	0	0
15	1	1	1	1	1

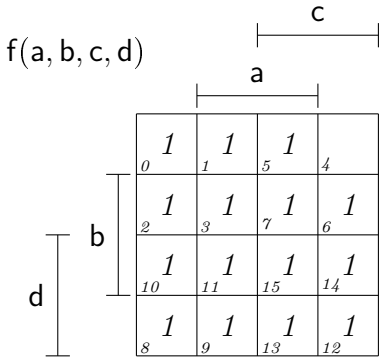


Vorgehensweise bei der Funktionsbeschreibung

Sind boolesche Ausdrücke gegeben, um eine Funktion zu beschreiben, lässt sich jeder Wert einzeln bestimmen und eintragen. Prinzipiell ist jeder boolesche Ausdruck entweder als eine Disjunktion oder als eine Konjunktion anzusehen, abgesehen von einer Negation des gesamten Ausdrucks. Eine Disjunktion ist immer dann wahr, wenn mindestens eine der Komponenten, aus der die Disjunktion besteht, gleich 1 ist. Somit können für die einzelnen Komponenten einer Disjunktion die 1-Werte nacheinander eingetragen werden.

Beispiel 19. *Disjunktive Funktion*

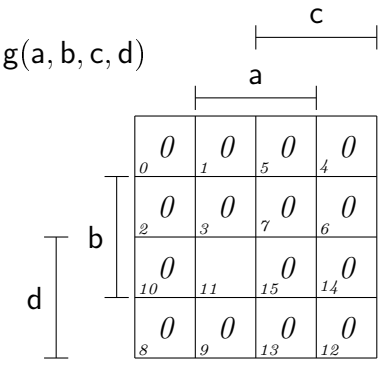
$f(a, b, c, d) = a + b + \bar{c} + c \cdot d$



Eine Konjunktion ist immer dann falsch, wenn mindestens eine der Komponenten der Konjunktion gleich 0 ist. Analog zu Disjunktionen werden nur die 0-Werte zu jeder einzelnen Komponente eingetragen.

Beispiel 20. *Konjunktive Funktion*

$g(a, b, c, d) = a \cdot b \cdot \bar{c} \cdot (c + d)$



2 Zahlendarstellung

2.1 Stellenwertsysteme (B-adische Systeme)

2.1.1 Darstellung natürlicher Zahlen

Jede natürliche Zahl ist durch die folgende Summenformel darstellbar:
$N = a_{n-1} \cdot B^{n-1} + a_{n-2} \cdot B^{n-2} + \cdots + a_1 \cdot B^1 + a_0 \cdot B^0 = \sum_{i=0}^{n-1} a_i \cdot B^i$

- N = Zahl im entsprechenden Stellenwertsystem mit der Stellenanzahl n
- B = Basis (Radix), z.B. 10 (dezimal), 2 (dual)
- B^i = Wertigkeit (Gewicht) der i-ten Stelle
- a_i = Ziffer (Koeffizient) der i-ten Stelle aus der Menge der Ziffern {0, 1, 2, ...,B-1 }

2.1.2 Zerlegung nach dem Horner-Schema

$N = \sum_{i=0}^{n-1} a_i \cdot B^i = (\ldots(a_{n-1} \cdot B + a_{n-2}) \cdot B + \cdots + a_1) \cdot B + a_0$
Man erhält die Ziffern $a_0, a_1, \ldots, a_{n-1}$ dadurch, dass man die Zahl N durch B dividiert und man nach der Division den Rest abtrennt und ihn notiert, d.h. man erhält zuerst a_0. Mit dem Quotienten wird dann in derselben Weise verfahren, bis er nach wiederholter Durchführung verschwindet. Die notierten Ziffern a_i stellen dann die Zahl im Stellenwertsystem zur Basis B dar.

Beispiel 21. *Konvertierung ins Dualsystem*
Ausgehend vom Dezimalsystem und unter Verwendung der Dezimalarithmetik ist es möglich mit dem beschriebenen Verfahren Dualzahlen zu erzeugen:

$167_{10} \rightarrow$

167 / 2 = 83	*Rest 1*	*Least Significant Bit, LSB*
83 / 2 = 41	*Rest 1*	
41 / 2 = 20	*Rest 1*	
20 / 2 = 10	*Rest 0*	
10 / 2 = 5	*Rest 0*	
5 / 2 = 2	*Rest 1*	
2 / 2 = 1	*Rest 0*	
1 / 2 = 0	*Rest 1*	*Most Significant Bit, MSB*

$\rightarrow 10100111_2$

Beispiel 22. *Rückkonvertierung ins Dezimalsystem*
Einsetzen der Ziffern (Koeffizienten) in die allgemeine Gleichung zur Zahlendarstellung mit B*=2 (Summenformel oder Horner-Schema). Im folgenden Beispiel in Form der Summenformel:*
$10100111_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 167_{10}$

2.2 Vorzeichenlose Zahlen

Vorzeichenlose Dualzahlen (unsigned binary numbers, unsigned integers) werden als natürliche Zahlen im Stellenwertsystem 2 mit n Stellen a_i folgendermaßen dargestellt:
$D = \sum_{i=0}^{n-1} a_i \cdot 2^i = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$

2.2.1 Addition

Die Addition von Dualzahlen wird durchgeführt indem paarweise die Stellen der Summanden addiert werden (mit Übertrag). Angewendet werden folgende Rechenregeln:
0+0 = 0
0+1 = 1
1+0 = 1
1+1 = **1**0

Bei 1+1=**1**0 entsteht eine "1" als Übertrag (Carry c), die eine zusätzliche Stelle benötigt.

Beispiel 23. *Addition:*

$$\begin{array}{rl} 5+6=11 & \rightarrow\ 0101 \\ & \quad +0110 \\ & \overline{\quad \overset{\circ}{1}011} \end{array}$$

$$\begin{array}{rl} 5+12=17 & \rightarrow\ 0101 \\ & \quad +1100 \\ & \overline{\quad \overset{\circ}{1}\overset{\circ}{0}001} \end{array}$$

2.2.2 Subtraktion

Die Subtraktion von Dualzahlen wird durchgeführt indem paarweise die Stellen des Subtrahenden von denen des Minuenden subtrahiert werden (mit Übertrag). Angewendet werden folgende Rechenregeln:
0-0 = 0
0-1 = 1
1-0 = 1
1-1 = 0

Bei 0-1=1 wird von der nächsthöheren Stelle eine "1" als Übertrag geborgt (Borrow b).

Beispiel 24. *Subtraktion:*

$$\begin{array}{rl} 12-5=7 & \rightarrow\ 1100 \\ & \quad -0101 \\ & \overline{\quad \overset{\circ}{0}\overset{\circ}{1}\overset{\circ}{1}1} \end{array}$$

$$\begin{array}{rl} 4-5=-1 & \rightarrow\ 0100 \\ & \quad -0101 \\ & \overline{\quad \overset{\circ}{1}\overset{\circ}{1}\overset{\circ}{1}\overset{\circ}{1}1} \end{array}$$

2.2.3 Zahlenbereich

Der Zahlenbereich von Dualzahlen kann als Zahlenring dargestellt werden. In der Abbildung 2.1 ist die Zahlenbereichsgrenze eingezeichnet. Wird der Zahlenbereich überschritten, so ist das an dem Übertragsbit des höchstwertigen Bits zu erkennen (Carry am Most Significant Bit).

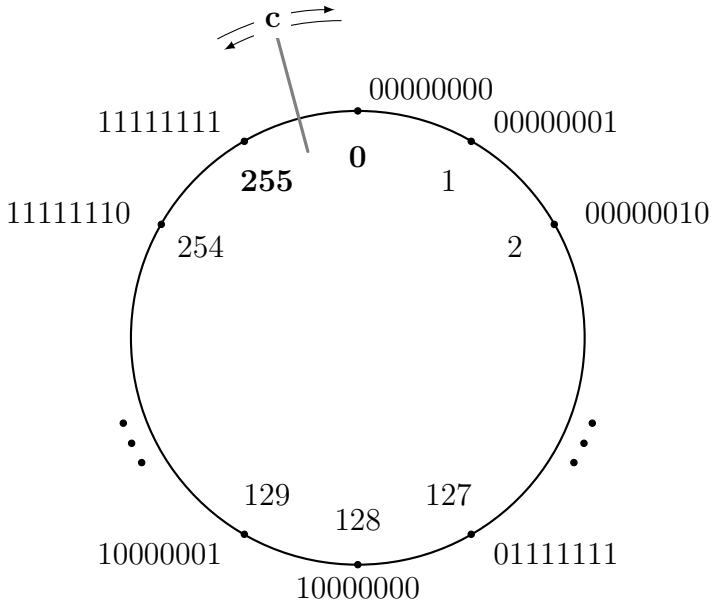


Abbildung 2.1: Zahlenring für vorzeichenlose Dualzahlen

2.3 Vorzeichenbehaftete Zahlen

Es gibt unterschiedliche Möglichkeiten negative Dualzahlen darzustellen. Ausgangspunkt bei der Codierung negativer Zahlen ist die Codierung positiver Dualzahlen. Im Folgenden werden die verbreitetsten Varianten aufgelistet:

- Vorzeichen-/Betrags-Zahlen (VB-Zahlen):
  Höchstwertiges (n-tes) Bit bestimmt das Vorzeichen: “0” zeigt eine positive , “1” eine negative Zahl an. Die restlichen n-1 Bits sind codiert wie vorzeichenlose Dualzahlen und bestimmen den Betrag der Zahl.

- 1-Komplement-Zahlen (1K-Zahlen):
  Negative Zahlen erhält man durch das bitweise Invertieren einer vorzeichenlosen Dualzahl mit selben Betrag. Höchstwertiges (n-tes) Bit impliziert das Vorzeichen: “0” zeigt eine positive , “1” eine negative Zahl an.

- 2-Komplement-Zahlen (2K-Zahlen):
  Negative Zahlen werden erzeugt indem die entsprechenden positiven Dualzahl bitweise invertiert wird und anschließend eine "1" addiert wird. Höchstwertiges (n-tes) Bit (Negative n) impliziert das Vorzeichen: "0" zeigt eine positive , "1" eine negative Zahl an.

Beispiel 25. *Zahlen mit 4-Bit-Wortlänge:*

positive Zahlen		*negative Zahlen*			
dez.	*dual*	*dez.*	*VB-Zahl*	*1K-Zahl*	*2K-Zahl*
+0	*0 000*	*-0*	*1 000*	*1 111*	—
+1	*0 001*	*-1*	*1 001*	*1 110*	*1 111*
+2	*0 010*	*-2*	*1 010*	*1 101*	*1 110*
+3	*0 011*	*-3*	*1 011*	*1 100*	*1 101*
+4	*0 100*	*-4*	*1 100*	*1 011*	*1 100*
+5	*0 101*	*-5*	*1 101*	*1 010*	*1 011*
+6	*0 110*	*-6*	*1 110*	*1 001*	*1 010*
+7	*0 111*	*-7*	*1 111*	*1 000*	*1 001*
		-8	—	—	*1 000*

2.4 2-Komplement-Zahlen

In der Praxis werden (nahezu) ausschließlich vorzeichenbehaftete Zahlen (signed binary numbers, signed integers) in 2K-Codierung benutzt. Im Stellenwertsystem 2 mit n Stellen a_i werden sie als ganze Zahlen folgendermaßen dargestellt:
$Z = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$ (d.h. mit negativem Gewicht für die Vorzeichenstelle n-1).

2.4.1 Bildung

Das Darstellen einer 2-Komplement-Zahl, d.h. die 2-Komplement-Bildung, folgt der Vorschrift:

$$\begin{aligned} -Z &= -(-a_{n-1} \cdot 2^{n-1} + \textstyle\sum_{i=0}^{n-2} a_i \cdot 2^i) = a_{n-1} \cdot 2^{n-1} - \sum_{i=0}^{n-2} a_i \cdot 2^i \\ &= a_{n-1} \cdot 2^{n-1} - \textstyle\sum_{i=0}^{n-2} a_i \cdot 2^i + (2^{n-1} - 1) - (2^{n-1} - 1) \\ &= a_{n-1} \cdot 2^{n-1} - 2^{n-1} - \textstyle\sum_{i=0}^{n-2} a_i \cdot 2^i + \sum_{i=0}^{n-2} 2^i + 1 \\ &= -(1 - a_{n-1}) \cdot 2^{n-1} + \textstyle\sum_{i=0}^{n-2} (1 - a_i) \cdot 2^i + 1 \end{aligned}$$

Beispiel 26. *2-Komplement-Bildung:*

$$\begin{aligned} & 01101001_2 && = +105_{10} \\ \textit{Invertierung} : \; & 10010110_2 \\ \textit{Addition von Eins} : \; & + \quad\quad 1 \\ \textit{2-Komplement} : \; & \overline{10010111_{2K}} && = -105_{10} \end{aligned}$$

und zurück:

$$\begin{aligned} & 10010111_{2K} && = -105_{10} \\ \textit{Invertierung} : \; & 01101000_2 \\ \textit{Addition von Eins} : \; & + \quad\quad 1 \\ \textit{2-Komplement} : \; & \overline{01101001_{2K}} && = +105_{10} \end{aligned}$$

2.4.2 Addition

Genau wie bei den vorzeichenlosen Dualzahlen, findet die Addition der 2K-Zahlen stellenpaarweise statt und es gibt einen Übertrag. Auch bei 2K-Zahlen kann es zu einer Bereichsüberschreitung kommen, diese wird aber zur Unterscheidung von der bei den vorzeichenlosen Dualzahlen Überlauf (Overflow v) genannt.

Beispiel 27. *Addition:*

$$\begin{aligned} 5 + (-4) = 1 \; \rightarrow \; & 0101 \\ & +1100 \\ & \overline{\mathring{1}\mathring{0}0001} \end{aligned}$$

Es gibt keinen Überlauf, da die beiden 'obersten' Überträge 1 sind ($v = 1 \oplus 1 = 0$). Es handelt sich um das korrekte positive Ergebnis: $0001_{2K} \equiv +1_{10}$.

2.4.3 Subtraktion

Äquivalent zur Addition erfolgt die Subtraktion der 2K-Zahlen genau so wie bei den vorzeichenlosen Zahlen. Auch hier findet eine Bereichsüberschreitung bei einem Überlauf statt.

Beispiel 28. *Subtraktion:*

$$\begin{aligned} (-128) - 127 = -255 \; \rightarrow \; & 1000\,0000 \\ & -0111\,1111 \\ & \overline{\mathring{0}\mathring{0}\mathring{0}\mathring{0}\,\mathring{0}\mathring{0}\mathring{0}\mathring{1}} \end{aligned}$$

Es gibt einen Überlauf, da nur einer der beiden 'obersten' Überträge 1 ist ($v = 0 \oplus 1 = 1$). Es ensteht ein falsches positives Ergebnis: $0000\,0001_{2K} \equiv +1_{10}$.

2.4.4 Zahlenbereich

Der Zahlenkreis verdeutlicht wie sich der Zahlenbereich, und somit die Grenze an der eine Bereichsüberschreitung stattfinden kann, verschoben hat.

Die ‘0’ markiert im Gegensatz zu den vorzeichenlosen Zahlen nicht mehr das untere Ende des Zahlenbereiches, sondern die Mitte, und die Grenze ist zu der anderen Stelle des Zahlenkreises gewandert, an der das höchstwertigste Bit (MSB, negative Bit) seinen Wert ändert. Die Zahlenbereichsüberschreitung ist am Überlauf (Overflow v) zu erkennen.

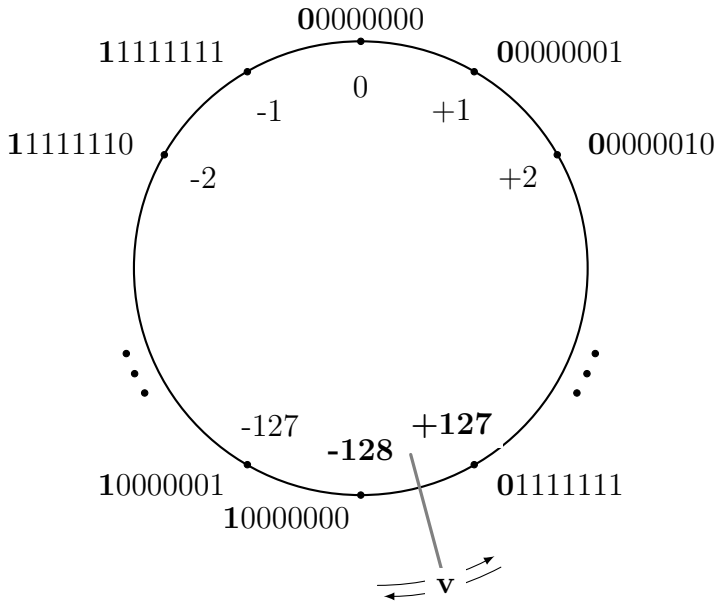


Abbildung 2.2: Zahlenring für 2-Komplement-Zahlen

2.5 Gray-Code

Neben der Codierung zur Informationsdarstellung und -verarbeitung gibt es weitere Codierungsmöglichkeiten bei deren Verwendung andere Randbedingungen im Vordergrund stehen. Bei dem Gray-Code sind diese Bedingungen Sicherheit und Zuverlässigkeit, die darin Berücksichtigung finden, dass benachbarte Codes sich jeweils durch genau ein Bit unterscheiden. Der Übergang von einem Code zum nächsten findet somit durch eine wohldefinierte Veränderung statt. Für die Erzeugung des Gray-Code gibt es eine rekursive Bildungsvorschrift:

$$[G_{i+1}]_{2n\times(i+1)} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}_n \cup \begin{bmatrix} G_i \end{bmatrix}_{n\times i} , \quad G_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}_{2\times 1}, \qquad \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_n \cup \begin{bmatrix} G_i^R \end{bmatrix}_{n\times i}$$

$$\begin{array}{lcl} i & \geqslant & 1 \\ n & \ldots & \text{Anzahl der Codes in Stufe i} \\ G_i^R & \ldots & G_i \text{ gespiegelt} \end{array}$$

Den reflektierten Code erhält man durch Spiegelung in der Zeilenmitte.

Beispiel 29. *Gray-Code für 3 Bit:*

Dez.	*Gray-Code*
0_{10}	*000*
1_{10}	*001*
2_{10}	*011*
3_{10}	*010*
4_{10}	*110*
5_{10}	*111*
6_{10}	*101*
7_{10}	*100*

3 Logische Normalformen und Primtermdarstellung

Logische Normalformen

Eine vollständig definierte logische Funktion kann durch einen logischen Ausdruck angegeben werden. Der logische Ausdruck ist aber nicht eindeutig bestimmt, d. h. es gibt mehrere äquivalente Darstellungen für eine Funktion. Für Verfahren oder Algorithmen, die auf eine Funktion angewendet werden sollen, ist es aber meist nötig, von einer Standarddarstellung der Funktion auszugehen. Solche standardisierten Darstellungsformen sind die logischen Normalformen. Die Notwendigkeit einer solchen Vereinheitlichung wird z. B. notwendig, wenn die Äquivalenz von Ausdrücken gezeigt werden soll. Abgesehen vom Vergleich mittels Wertetabellen, müssten die Ausdrücke durch äquivalente Umformungen auf eine gemeinsame Form gebracht werden. Dabei sollte die prinzipielle Gestalt der Ausdrücke nach Umformung bekannt sein, um vergleichen zu können.

Beispiel 30. *Aufzeigen der Äquivalenz zweier Funktionen mittels einer Normalform*

$$\begin{aligned} f_1 &= (a + b) \to a \cdot c \\ &= \overline{a + b} + a \cdot c \\ &= \overline{a} \cdot \overline{b} + a \cdot c \end{aligned} \qquad \begin{aligned} f_2 &= (\overline{a} + \overline{c}) \to \overline{a + b} \\ &= (\overline{a} + \overline{c}) \to \overline{a} \cdot \overline{b} \\ &= \overline{\overline{a} + \overline{c}} + \overline{a} \cdot \overline{b} \\ &= a \cdot c + \overline{a} \cdot \overline{b} \\ &= \overline{a} \cdot \overline{b} + a \cdot c \end{aligned}$$

also : $f_1 = f_2$

Definition 11. *Term*
Einen logischen Ausdruck oder einen Teil eines logischen Ausdruckes, der selbst ein logischer Ausdruck ist, nennt man auch Term. Eine Konjunktion $X_1 \cdot X_2 \cdot \dots \cdot X_n$ *heißt n-stelliger Konjunktionsterm, wenn alle* $X_i, i \in 1, \dots, n$*, entweder logische Variablen oder negierte logische Variablen sind und jede Variable nur einmal vorkommt. Ein 0-stelliger Konjunktionsterm soll mit der logischen Konstanten 1 identisch sein. Eine Disjunktion* $X_1 + X_2 + \dots + X_n$ *heißt n-stelliger Disjunktionsterm, wenn jedes* $X_i, i \in 1, \dots, n$*, entweder logische Variablen oder negierte logische Variablen sind und jede Variable nur einmal vorkommt. Ein 0-stelliger Disjunktionsterm soll mit der Konstanten 0 identisch sein.*

Umformung äquivalenter Ausdrücke

Jeder Ausdruck lässt sich äquivalent in einen booleschen Ausdruck umformen, da $(\cdot, +, \overline{\ })$ ein vollständiges System ist. Für dieses System stehen eine Menge von Umformungsre-

geln zur Verfügung, mit denen gerechnet werden kann. Als Ausgangsposition werden bestimmte Formen boolescher Ausdrücke genannt. Dabei unterscheidet man hauptsächlich zwei Typen von Normalformen: die disjunktive Normalform (DNF) und die konjunktive Normalform (KNF).

3.1 Disjunktive Normalform

3.1.1 Allgemeine disjunktive Normalform

Definition 12. *Disjunktive Normalform (DNF)*
Ein logischer Ausdruck der Form $\mathsf{K}_1 + \mathsf{K}_2 + \cdots + \mathsf{K}_k, k \in \mathbb{N}$*, heißt disjunktive Normalform (DNF), wenn alle* $\mathsf{K}_i, i \in \{1, 2, \ldots, k\}$*, paarweise verschiedene Konjunktionsterme sind. Gilt* $f(a_1, a_2, \ldots, a_n) = \mathsf{K}_1 + \mathsf{K}_2 + \cdots + \mathsf{K}_k, k, n \in \mathbb{N}$*, so heißt* $\mathsf{K}_1 + \mathsf{K}_2 + \cdots + \mathsf{K}_k$ *eine disjunktive Normalform der Funktion f.*

Beispiel 31. *Disjunktive Normalformen sind:*
$a \cdot b \cdot \overline{c} + a \cdot c + \overline{a} \cdot c$
$x_1 \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot \overline{x_3} \cdot \overline{x_4} + \overline{x_1} \cdot \overline{x_2} + x_4$
$\overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot \overline{z} + \overline{x} \cdot y \cdot z + x \cdot \overline{y} \cdot \overline{z} + x \cdot \overline{y} \cdot z + x \cdot y \cdot \overline{z}$

Eine DNF ist für eine gegebene vollständig definierte Funktion i. A. nicht eindeutig bestimmt.

Beispiel 32. *Umformung in DNF*

$$
\begin{aligned}
f_{(a,b,c,d)} &= b \cdot (c \rightarrow a \cdot d) \\
&= b \cdot (\overline{c} + a \cdot d) \\
&= b \cdot \overline{c} + a \cdot b \cdot d \\
&= b \cdot \overline{c} + \cancel{a \cdot b \cdot c \cdot d} + a \cdot b \cdot c \cdot d \\
&= \overline{a} \cdot b \cdot \overline{c} + a \cdot b \cdot \overline{c} + a \cdot b \cdot c \cdot d \\
&= \overline{a} \cdot b \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot \overline{c} \cdot d + a \cdot b \cdot \overline{c} + a \cdot b \cdot c \cdot d \\
&= \overline{a} \cdot b \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot \overline{c} \cdot d + a \cdot b \cdot \overline{c} \cdot \overline{d} + a \cdot b \cdot \overline{c} \cdot d + a \cdot b \cdot c \cdot d
\end{aligned}
$$

Umformung eines logischen Ausdrucks in eine mögliche DNF

1. Umformung des logischen Ausdrucks in einen Booleschen Ausdruck (Eliminierung der Verknüpfungen $\rightarrow, \equiv, \oplus, \mathsf{NAND}, \mathsf{NOR}$).

2. Anwendung der De Morganschen Regeln ($\overline{a \cdot b} = \overline{a} + \overline{b}$, $\overline{a + b} = \overline{a} \cdot \overline{b}$) bis die Negation nur noch bei Variablen auftritt.

3. Anwendung des Distributivgesetzes, des Komplementgesetzes, des 0-1-Gesetzes und des Idempotenzgesetzes.

4. Mehrfach auftretende Konjunktionsterme werden nach dem Idempotenzgesetz $(a + a = a)$ zusammengefasst.

Das Komplementgesetz und das 0-1-Gesetz sind nur zwischen Variablen anzuwenden, die schon durch Konjunktionszeichen miteinander verbunden sind. Es werden dabei mehrfach auftretende Variablen innerhalb dieses Terms eliminiert. Nur so erhält man die geforderten Konjunktionsterme. Mit dem Idempotenzgesetz wird erreicht, dass kein Konjunktionsterm mehrfach auftritt. Dieser letzte Schritt erübrigt sich jedoch meist.

Beispiel 33. *Umformung in DNF*

$$\begin{aligned}
f_{(a,b,c,d)} &= a \cdot b \equiv \overline{c \cdot d} \rightarrow (\overline{a \cdot b}) \\
&= (a \cdot b \cdot \overline{c \cdot d} + \overline{a \cdot b} \cdot c \cdot d) \rightarrow (\overline{a} + \overline{b}) \\
&= \overline{(a \cdot b \cdot \overline{c \cdot d} + \overline{a \cdot b} \cdot c \cdot d)} + (\overline{a} + \overline{b}) \\
&= \overline{(a \cdot b \cdot \overline{c \cdot d}} \cdot \overline{\overline{a \cdot b} \cdot c \cdot d)} + \overline{a} + \overline{b} \\
&= (\overline{a} + \overline{b} + c \cdot d) \cdot (a \cdot b + \overline{c} + \overline{d}) + \overline{a} + \overline{b} \\
&= \overline{a} \cdot \overline{c} + \overline{a} \cdot \overline{d} + \overline{b} \cdot \overline{c} + \overline{b} \cdot \overline{d} + a \cdot b \cdot c \cdot d + \overline{a} + \overline{b}
\end{aligned}$$

Sonderfälle

Eine DNF kann auch aus nur einem einzigen Konjunktionsterm bestehen. Besteht der umzuwandelnde logische Ausdruck nur aus der Konstanten 1, entspricht dies der DNF eines formal wahren Ausdrucks. Für formal falsche Ausdrücke gibt es keine DNF.

3.1.2 Kanonische disjunktive Normalform

Problem

Die DNF eines logischen Ausdrucks i. A. nicht eindeutig bestimmt ist, was zu Schwierigkeiten beim Vergleich logischer Funktionen führt. Es gibt aber eine ausgezeichnete oder kanonische disjunktive Normalform, die diese Schwierigkeit umgeht.

Definition 13. *Kanonische disjunktive Normalform (KDNF)*
Erweiterung einer disjunktiven Normalform zu einer vollständig definierten n-stelligen logischen Funktion heißt kanonisch (oder ausgezeichnet) (KDNF), wenn jeder der vorhandenen Konjunktionsterme alle n Variablen enthält und nur einmal auftritt.

Umformung der DNF zu KDNF

Die kanonische disjunktive Normalform (KDNF) ist für jede vollständig definierte logische Funktion eindeutig bestimmt. Sie hat aber die unangenehme Eigenschaft, die „längste“ DNF der jeweiligen Funktion zu sein. Jeder Konjunktionsterm ist n-stellig, falls die Funktion n-stellig ist, enthält also alle Variablen. Die kanonische disjunktive Normalform kann aus jeder beliebigen DNF gewonnen werden. Die Konjunktionsterme,

die noch nicht alle Variablen enthalten, werden mit dem Komplementgesetz ($a + \overline{a} = 1$) um die nicht vorhandenen Variablen erweitert. Dabei entsteht natürlich noch jeweils ein weiterer Konjunktionsterm. Mehrfach auftretende Konjunktionsterme werden nach dem Idempotenzgesetz ($a + a = a$) zusammengefasst.

Schritte zur Umformung DNF in KDNF

Gegeben sei eine DNF einer n-stelligen Funktion.

1. Jeder Konjunktionsterm, der nicht n-stellig ist, wird konjunktiv mit dem Term $(a_j + \overline{a_j})$ verknüpft, wobei a_j eine Variable ist, die in diesem Konjunktionsterm nicht enthalten ist.

2. Auf den so erhaltenen neuen Konjunktionsterm wird das Distributivgesetz ($K \cdot (a_j + \overline{a_j}) = K \cdot a_j + K \cdot \overline{a_j}$) angewandt.

3. Mehrfach auftretende Terme werden nach den Idempotenzgesetzen zusammengefasst

Wiederholte Anwendung der vorstehenden Schritte liefert die KDNF.

Beispiel 34. *Umformung DNF in KDNF*

$$\begin{aligned} f_{(a,b,c)} &= a + b \\ &= a \cdot (b + \overline{b}) + (a + \overline{a}) \cdot b \\ &= a \cdot b + a \cdot \overline{b} + a \cdot b + \overline{a} \cdot b \\ &= a \cdot b + a \cdot \overline{b} + \overline{a} \cdot b \\ &= a \cdot b \cdot (c + \overline{c}) + a \cdot \overline{b} \cdot (c + \overline{c}) + \overline{a} \cdot b \cdot (c + \overline{c}) \\ &= a \cdot b \cdot c + a \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot c + a \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot b \cdot c + \overline{a} \cdot b \cdot \overline{c} \end{aligned}$$

Aufstellung der KDNF für Funktionen, die durch Wertetabellen gegeben sind

Die Konjunktionsterme einer KDNF werden auch Minterme genannt. Es gibt 2^n Minterme bei n Variablen, nämlich so viele Minterme, wie es Belegungen gibt.

Definition 14. *Ein n-stelliger Konjunktionsterm über n Variablen* $a_1, a_2, \ldots, a_n$ *heißt n-stelliger Minterm.*

Beispiel 35. *Wertetabellen einiger Minterme*

	d	c	b	a	$a\bar{b}cd$	$\bar{a}\bar{b}\bar{c}d$
0	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*0*	*0*	*1*	*0*	*0*
2	*0*	*0*	*1*	*0*	*0*	*0*
3	*0*	*0*	*1*	*1*	*0*	*0*
4	*0*	*1*	*0*	*0*	*0*	*0*
5	*0*	*1*	*0*	*1*	*0*	*0*
6	*0*	*1*	*1*	*0*	*0*	*0*
7	*0*	*1*	*1*	*1*	*0*	*0*
8	*1*	*0*	*0*	*0*	*0*	*1*
9	*1*	*0*	*0*	*1*	*0*	*0*
10	*1*	*0*	*1*	*0*	*0*	*0*
11	*1*	*0*	*1*	*1*	*0*	*0*
12	*1*	*1*	*0*	*0*	*0*	*0*
13	*1*	*1*	*0*	*1*	*1*	*0*
14	*1*	*1*	*1*	*0*	*0*	*0*
15	*1*	*1*	*1*	*1*	*0*	*0*

Mit dieser Kenntnis über Minterme lässt sich nun leicht eine KDNF aus der Wertetabelle einer vollständig definierten Funktion ableiten. Da ein Minterm immer nur für eine Belegung den Wert 1 annimmt, steht jeder Minterm in einer KDNF für genau eine Belegung, bei der die gesamte disjunktive Normalform eine 1 liefert (0-1-Gesetz). Die Anzahl der Minterme gibt dabei an, wie oft der Wahrheitswert 1 im Ergebnisvektor der zugehörigen Funktion steht.

Aufstellung einer kanonischen disjunktiven Normalform aus der Wertetabelle:

Gegeben sei die Wertetabelle einer vollständig definierten logischen Funktion, die $\neq 0$ ist.

1. Zu jeder Belegung, die den Funktionswert 1 hat, wird der Minterm ermittelt, indem die Variablen, die in dieser Belegung 0 sind, als negierte Variablen und die, die 1 sind, als einfache Variablen konjunktiv verknüpft werden.

2. Die Disjunktion dieser Minterme ist die KDNF der gegebenen Funktion.

Die KDNF wird um so länger, je häufiger der Wert 1 im Ergebnisvektor auftritt. Wegen des Kommutativgesetzes ist die Reihenfolge der Minterme in der KDNF gleichgültig. Es ist aber üblich, die Minterme nach aufsteigenden Indizes i aufzuschreiben. Die Reihenfolge der Variablen in den Mintermen ist dabei stets gleich.

Beispiel 36. *KDNF aus Wertetabelle:*

	c	b	a	f(a, b, c)	*Minterm*
0	*0*	*0*	*0*	*0*	
1	*0*	*0*	*1*	*0*	
2	*0*	*1*	*0*	*1*	$\bar{a}b\bar{c}$
3	*0*	*1*	*1*	*1*	$ab\bar{c}$
4	*1*	*0*	*0*	*0*	
5	*1*	*0*	*1*	*1*	$a\bar{b}c$
6	*1*	*1*	*0*	*0*	
7	*1*	*1*	*1*	*1*	abc

$$f(a, b, c) = \bar{a}b\bar{c} + ab\bar{c} + a\bar{b}c + abc$$

3.2 Konjunktive Normalform

3.2.1 Allgemeine konjunktive Normalform

Definition 15. *Ein logischer Ausdruck der Form* $D_1 \cdot D_2 \cdot \dots \cdot D_k, k \in \mathbb{N}$, *heißt konjunktive Normalform (KNF), wenn alle* $D_i, i \in \{1, 2, \dots, k\}$ *paarweise verschiedene Disjunktionsterme sind.*
Gilt $f(a_1, a_2, \dots, a_n) = D_1 \cdot D_2 \cdot \dots \cdot D_k, k, n \in \mathbb{N}$, *so heißt* $D_1 \cdot D_2 \cdot \dots \cdot D_k$ *eine konjunktive Normalform der Funktion f.*

Beispiel 37. *Konjunktive Normalformen sind:*

1. $(a + b + \bar{c}) \cdot (a + c) \cdot (a + \bar{b} + c)$
2. $(x_1 + x_2 + \overline{x_3} + \overline{x_4}) \cdot (x_1 + \overline{x_3} + x_4) \cdot (x_2 + x_3 + x_4) \cdot x_1$
3. $(\bar{x} + \bar{y} + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot (\bar{x} + y + \bar{z}) \cdot (\bar{x} + y + z) \cdot (x + \bar{y} + \bar{z}) \cdot (x + \bar{y} + z) \cdot$
   $(x + y + \bar{z}) \cdot (x + y + z)$

Auch die KNF ist für eine Funktion i. A. nicht eindeutig bestimmt.

Beispiel 38.

$$\begin{aligned}
f(a, b, c, d) &= b \cdot (c \to a \cdot d) \\
&= b \cdot (\bar{c} + a \cdot d) \\
&= b \cdot \bar{c} + a \cdot b \cdot d \\
&= (b \cdot \bar{c} + a) \cdot (b \cdot \bar{c} + b) \cdot (b \cdot \bar{c} + d) \\
&= (b + a) \cdot (\bar{c} + a) \cdot b \cdot (b + d) \cdot (\bar{c} + d) \cdot (\bar{c} + b) \\
&= (a + b + c) \cdot (a + b + \bar{c}) \cdot (\bar{c} + a) \cdot b \cdot (b + d) \cdot (\bar{c} + d) \\
&= (a + b + c + d) \cdot (a + b + c + \bar{d}) \cdot (a + b + \bar{c}) \cdot (\bar{c} + a) \cdot b \cdot (b + d) \cdot (\bar{c} + d)
\end{aligned}$$

Umformung eines logischen Ausdrucks in eine mögliche KNF

1. Umformung des logischen Ausdrucks in einen Booleschen Ausdruck (Eliminierung der Verknüpfungen $\rightarrow, \equiv, \oplus, \mathsf{NAND}, \mathsf{NOR}$).

2. Anwendung der De Morganschen Regeln, bis Negation nur noch bei einzelnen Variablen auftritt.

3. Anwendung des Distributivgesetzes, des Komplementgesetzes, des 0-1-Gesetzes und des Idempotenzgesetzes.

4. Mehrfach auftretende Disjunktionsterme werden nach dem Idempotenzgesetz zusammengefasst. Komplementgesetz und 0-1-Gesetz müssen angewandt werden, um innerhalb der einzelnen Disjunktionen zu erreichen, dass jede Variable nur einmal auftritt.

Sonderfälle

Die konjunktive Normalform kann auch aus nur einem Disjunktionsterm bestehen. Andererseits braucht ein Disjunktionsterm nicht mehr als eine Variable zu enthalten. Das bedeutet aber, dass die DNF $\mathsf{f}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}) = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c} \cdot \overline{\mathsf{d}}$ ebenfalls eine konjunktive Normalform ist, und zwar sogar die "kürzeste". Die Konstante 0 ist eine KNF eines formal falschen Ausdrucks. Für formal wahre Ausdrücke lässt sich keine KNF konstruieren. Die angegebenen Regeln zur Gewinnung der Normalformen führen nicht automatisch zur minimalen Form.

3.2.2 Kanonische konjunktive Normalform

Definition 16. *Eine konjunktive Normalform einer vollständig definierten n-stelligen logischen Funktion heißt kanonisch (oder ausgezeichnet)(KKNF), wenn jeder der auftretenden Disjunktionsterme alle n Variablen enthält und nur einmal auftritt.*

Die Herleitung der KKNF aus einer beliebigen KNF geschieht auch hier durch Erweitern der Disjunktionsterme um nicht vorhandene Variablen. Anschließend werden mehrfach auftretende Disjunktionsterme wieder zusammengefasst.

Schritte zur Umformung KNF in KKNF

Gegeben sei eine KNF einer n-stelligen Funktion.

1. Jeder Disjunktionsterm, der nicht n-stellig ist, wird disjunktiv mit dem Term $(\mathsf{a_j} \cdot \overline{\mathsf{a_j}})$ verknüpft, wobei $\mathsf{a_j}$ eine Variable ist, die in diesem Disjunktionsterm nicht enthalten war.

2. Auf den so erhaltenen neuen Disjunktionsterm wird das Distributivgesetz $\mathsf{D} + (\mathsf{a_j} \cdot \overline{\mathsf{a_j}}) = (\mathsf{D} + \mathsf{a_j}) \cdot (\mathsf{D} + \overline{\mathsf{a_j}})$ angewandt.

3. Mehrfach auftretende Terme werden nach den Idempotenzgesetzen zusammengefasst.

Wiederholte Anwendung der vorstehenden Regeln liefert die KKNF.

Beispiel 39. *Umformung von KNF in KKNF*

$$\begin{aligned}
f(a,b,c) &= a \cdot b \\
&= (a + (b \cdot \overline{b})) \cdot (b + (a \cdot \overline{a})) \\
&= (a + b) \cdot (a + \overline{b}) \cdot (b + a) \cdot (b + \overline{a}) \\
&= (a + b) \cdot (a + \overline{b}) \cdot (\overline{a} + b) \\
&= ((a + b) + (c \cdot \overline{c})) \cdot (a + \overline{b} + (c \cdot \overline{c})) \cdot ((\overline{a} + b) + (c \cdot \overline{c})) \\
&= (a + b + c) \cdot (a + b + \overline{c}) \cdot (a + \overline{b} + c) \cdot (a + b + \overline{c}) \cdot (\overline{a} + \overline{b} + c) \cdot (\overline{a} + b + \overline{c}) \cdot
\end{aligned} \tag{3.1}$$

Den Mintermen bei den KDNF entsprechen die Maxterme oder Volldisjunktionen bei den KKNF. Die Disjunktionsterme einer KKNF heißen Maxterme und liefern für genau eine Belegung eine 0, sonst immer 1.

Beispiel 40. *Wertetabellen einiger Maxterme*

	d	c	b	a	$a + \overline{b} + c + d$	$\overline{a} + \overline{b} + \overline{c} + d$
0	*0*	*0*	*0*	*0*	*1*	*1*
1	*0*	*0*	*0*	*1*	*1*	*1*
2	*0*	*0*	*1*	*0*	*0*	*1*
3	*0*	*0*	*1*	*1*	*1*	*1*
4	*0*	*1*	*0*	*0*	*1*	*1*
5	*0*	*1*	*0*	*1*	*1*	*1*
6	*0*	*1*	*1*	*0*	*1*	*1*
7	*0*	*1*	*1*	*1*	*1*	*0*
8	*1*	*0*	*0*	*0*	*1*	*1*
9	*1*	*0*	*0*	*1*	*1*	*1*
10	*1*	*0*	*1*	*0*	*1*	*1*
11	*1*	*0*	*1*	*1*	*1*	*1*
12	*1*	*1*	*0*	*0*	*1*	*1*
13	*1*	*1*	*0*	*1*	*1*	*1*
14	*1*	*1*	*1*	*0*	*1*	*1*
15	*1*	*1*	*1*	*1*	*1*	*1*

Bei n Variablen gibt es 2^n Maxterme (entsprechend der Zahl möglicher Belegungen), wobei jeder Maxterm ein n-stelliger Disjunktionsterm ist.

Definition 17. *Ein n-stelliger Disjunktionsterm über n Variablen* $a_1, a_2, \ldots, a_n$ *heißt n-stelliger Maxterm*

Satz 9. *Eine konjunktive Normalform einer vollständig definierten n-stelligen logischen Funktion ist kanonisch, wenn die Disjunktionsterme paarweise verschiedene n-stellige Maxterme sind.*

Gewinnung der KKNF aus einer Wertetabelle

Eine Konjunktion $D_1 \cdot D_2 \cdot \cdots \cdot D_k$ wird genau dann 0, wenn mindestens ein $D_i = 0$ ist. Ein Maxterm wird für genau eine Belegung 0 und dementsprechend steht jeder Maxterm in einer KKNF für genau eine Belegung, bei der diese Normalform 0 ist. Es treten so viele Maxterme auf, wie im Ergebnisvektor der Wahrheitswert 0 auftritt.

Aufstellung einer kanonischen konjunktiven Normalform aus der Wertetabelle:

Gegeben sei die Wertetabelle einer vollständig definierten logischen Funktion, die nicht identisch 1 ist.

1. Zu jeder Belegung, die den Funktionswert 0 hat, wird der Maxterm ermittelt, indem die Variablen, die in der entsprechenden Belegung 1 sind, als negierte Variablen und die, die 0 sind, als einfache Variablen disjunktiv verknüpft werden.

2. Die Konjunktion dieser Maxterme ist die KKNF der Funktion.

Beispiel 41. *KKNF aus Wertetabelle:*

	c	b	a	f(a, b, c)	*Maxterm*
0	*0*	*0*	*0*	*0*	$a + b + c$
1	*0*	*0*	*1*	*0*	$\overline{a} + b + c$
2	*0*	*1*	*0*	*1*	
3	*0*	*1*	*1*	*0*	$\overline{a} + \overline{b} + c$
4	*1*	*0*	*0*	*0*	$a + b + \overline{c}$
5	*1*	*0*	*1*	*1*	
6	*1*	*1*	*0*	*1*	
7	*1*	*1*	*1*	*1*	

$f(a, b, c) = (a + b + c) \cdot (\overline{a} + b + c) \cdot (\overline{a} + \overline{b} + c) \cdot (a + b + \overline{c})$

Praktischer Hinweis

Wie bei der KDNF ist es auch hier üblich, die Maxterme nach aufsteigenden Indizes i innerhalb einer KKNF anzugeben. Die Reihenfolge der Variablen in den einzelnen Maxtermen ist stets gleich. Sie ist festgelegt durch die Anordnung in der Wertetabelle oder durch die Angabe des Argumentes der Funktion oder – bei logischen Ausdrücken – durch die lexikographische Reihenfolge.

3.3 Zusammenhang zwischen den Normalformen

Überführung kanonische disjunktive ⇔ kanonische konjunktive Normalformen

Häufig muss eine DNF in die KNF umgeformt werden und umgekehrt. Diese Umrechnung wird oft recht umfangreich nach den bisher bekannten Regeln. Für die Umformung einer kanonischen Normalform in die andere lassen sich die Kenntnisse über Min- und Maxterme anwenden. Die KKNF besteht aus k Maxtermen M_i.

$i \in I \subset \{0, 1, 2, \ldots, 2^n - 1\}, k = |I| \leqslant 2^n$
Jedem Maxterm entspricht eine „0“ im Ergebnisvektor der zugehörigen Funktion. Die KDNF besteht dementsprechend aus $2^n - k$ Mintermen m_j, denn jedem Minterm entspricht eine 1 im Ergebnisvektor. Dabei treten nur Minterme mit solchen Indizes j auf, die in der KKNF nicht auftreten.
$j \in J \subset \{0, 1, 2, \ldots, 2^n - 1\}$ und $J \cap I = \varnothing$ und $J \cup I = \{0, 1, ..., 2^n - 1\}$

Umformung KDNF $\Leftrightarrow$ KKNF

Gegeben sei eine KDNF (KKNF).

1. Die KDNF (KKNF) wird mittels Mintermsymbolen m (Maxtermsymbolen M) dargestellt.
2. Die KKNF (KDNF) erhält man als Konjunktion (Disjunktion) derjenigen Maxterme (Minterme), deren Indizes nicht in der KDNF (KKNF) vorkommen.

Beispiel 42. $f_{KKNF} = (a + b + \overline{c}) \cdot (a + \overline{b} + \overline{c}) \cdot (\overline{a} + b + c) = M_1 \cdot M_3 \cdot M_4$
Es gibt $2^3 = 8$ *Minterme und ebensoviele Maxterme (*$0 \leqslant i \leqslant 7$*). Die in der KKNF nicht auftretenden Indizes* ($i = 0, 2, 5, 6, 7$) *sind die Indizes der Minterme.*
$f_{KDNF} = m_0 + m_2 + m_5 + m_6 + m_7 = \overline{a} \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot c + a \cdot b \cdot \overline{c} + a \cdot b \cdot c$

Bemerkung:

Die angegebenen Regeln zur Gewinnung der Normalform beruhen nur auf der Anwendung der bisher bekannten Umformungsregeln und Sätze. Danach ist eine Normalform eine äquivalente Darstellung einer vollständig definierten logischen Funktion. Es fehlt bisher der Beweis der Eindeutigkeit der kanonischen Normalform, der hier an dieser Stelle nicht aufgeführt wird.

3.4 Normalformen und partiell definierte Funktionen

Partiell definierte Funktionen

Ein logischer Ausdruck stellt eine vollständig definierte Funktion dar. In der Praxis treten jedoch auch partiell definierte Funktionen auf, die nicht für jede Belegung einen Funktionswert haben. Sie sind fast immer durch Wertetabellen gegeben. Die Realisierung als Schaltnetz erfordert jedoch eine Funktion als booleschen Ausdruck.

Vorgehensweise

Ist eine Funktion für eine Belegung nicht definiert, tritt diese Belegung entweder nie auf, oder es ist in diesem Moment gleichgültig, welcher Funktionswert auftritt („don't care“).

Beispiel 43. *Partiell definierte Funktion*
Funktion zum Erkennen einer bestimmten Ziffer des BCD-Codes. Bei einem 4-stelligen Argument gibt es 6 Belegungen, die nicht vorkommen können (die Dualzahlen 10 bis 15)

Werden nun den nicht definierten Belegungen der Funktion beliebige Funktionswerte zugewiesen, so ist dies ohne praktische Relevanz.

Vervollständigung einer Wertetabelle zur Gewinnung einer KDNF oder KKNF

Das Ergebnis der Vervollständigung liefert eine neue, vollständig definierte Funktion, die mit der ursprünglichen Funktion in den definierten Belegungen übereinstimmt. Um möglichst kurze kanonische Normalformen zu erhalten, ergänzt man die Wertetabelle: mit 0, falls eine KDNF gewonnen werden soll, mit 1, falls eine KKNF gewonnen werden soll. Dies gilt aber nur für kanonische Normalformen. Für möglichst kurze Normalformen müssen die fehlenden Werte gezielt ergänzt werden.

Beispiel 44. *Partiell definierte logische Funktion*

	c	b	a	$f(a,b,c)$	$f_{KDNF}(a,b,c)$	$f_{KKNF}(a,b,c)$
0	*0*	*0*	*0*	*1*	*1*	*1*
1	*0*	*0*	*1*	-	*0*	*1*
2	*0*	*1*	*0*	*0*	*0*	*0*
3	*0*	*1*	*1*	-	*0*	*1*
4	*1*	*0*	*0*	*0*	*0*	*0*
5	*1*	*0*	*1*	*0*	*0*	*0*
6	*1*	*1*	*0*	*1*	*1*	*1*
7	*1*	*1*	*1*	-	*0*	*1*

$f_{KDNF}(a,b,c) = m_0 + m_6 = \overline{a} \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot b \cdot c$
$f_{KKNF}(a,b,c) = M_2 \cdot M_4 \cdot M_5 = (a + \overline{b} + c) \cdot (a + b + \overline{c}) \cdot (\overline{a} + b + \overline{c})$

3.5 Karnaugh-Veitch-Diagramm und Normalformen

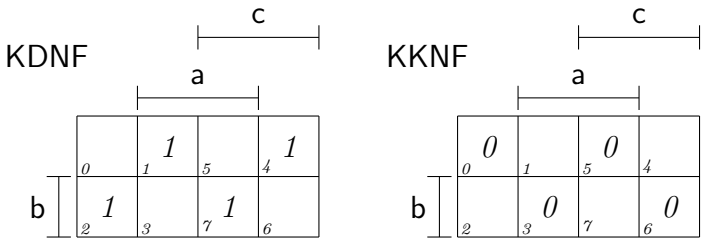
Abbildung boolescher Ausdrücke in Karnaugh-Veitch-Diagramme

Die Abbildung boolescher Ausdrücke in KV-Diagramme vereinfacht sich beträchtlich, wenn man von kanonischen Normalformen ausgeht. Jedem Feld des Diagramms ist genau ein Minterm bzw. Maxterm zugeordnet. Das dezimale Äquivalent der Belegung, die zu einem Feld des Diagramms gehört, stimmt mit dem Index i des zugehörigen Min- bzw. Maxterms überein. Bei einer KDNF wird für jeden Minterm eine 1 in das entsprechende Feld eingetragen. Bei einer KKNF verfährt man entsprechend, indem man für jeden Maxterm eine 0 einträgt.

Beispiel 45.

$$\begin{aligned} f(a,b,c) &= \overline{a} \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot \overline{c} + a \cdot b \cdot c \\ &= m_1 + m_2 + m_4 + m_7 \\ &= (a + b + c) \cdot (a + \overline{b} + \overline{c}) \cdot (\overline{a} + b + \overline{c}) \cdot (\overline{a} + \overline{b} + c) \\ &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \end{aligned}$$

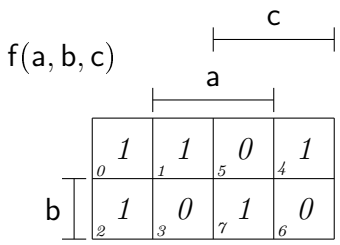
Für die Min- bzw. Maxterme ergeben sich folgende Diagramme:



Bestimmung eines logischen Ausdrucks aus dem KV-Diagramm einer vollständig definierten Funktion

Um die KKNF oder KDNF zu gewinnen, benutzt man das eben beschriebene Verfahren in umgekehrter Richtung. KDNF: Jede 1 im KV-Diagramm entspricht einem Minterm der KDNF. KKNF: Jede 0 im KV-Diagramm entspricht einem Maxterm der KKNF.

Beispiel 46. *Bestimmung der KDNF und KKNF aus KV-Diagramm*



Also liest man ab:
KDNF: $\mathsf{f(a,b,c)} = \mathsf{m_0} + \mathsf{m_1} + \mathsf{m_2} + \mathsf{m_4} + \mathsf{m_7}$
KKNF: $\mathsf{f(a,b,c)} = \mathsf{M_3} \cdot \mathsf{M_5} \cdot \mathsf{M_6}$

Mit KV-Diagramm lassen sich kürzere nicht kanonische Normalformen ermitteln. Benachbarte Einsen ergeben in der KDNF stets zwei Minterme, die sich zu einem n-1 stelligen Konjunktionsterm zusammenfassen lassen.

3.6 Primtermdarstellung logischer Funktionen

Für technische Realisierungen logischer Funktionen ist es wünschenswert, Darstellungen als möglichst kurze boolesche Ausdrücke zur Verfügung zu haben. Zu deren Gewinnung sollen jedoch zunächst als technische Hilfsmittel spezielle Konjunktionsterme, nämlich die Primterme, eingeführt werden.

3.6.1 Implikanten

Vorbemerkung

Bekannt sind aus der Logik die Sprechweisen „A impliziert B“ oder „aus A folgt B“, wobei A und B z. B. mathematische Aussagen sind. Eine solche Implikation muss formal

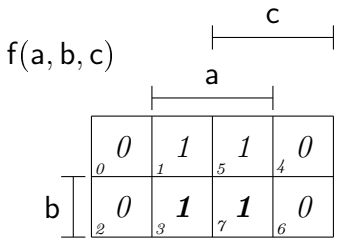
wahr sein, wenn sie in einem Theorem oder Satz auftritt. Sind zwei n-stellige logische Funktionen g und f gegeben und soll beschrieben werden, dass für jede Belegung, für die g den Wert 1 hat, auch f in ihrem Definitionsbereich D den Wert 1 annimmt, so zeigt die Wertetafel der Implikation, dass
$g(a_1, \ldots, a_n) \rightarrow f(a_j, \ldots, a_n) = 1$ für alle $(a_1, \ldots, a_n) \in D \subset \{0,1\}^n$
gesetzt werden muss, d. h. der Ausdruck $g(a_1, \ldots, a_n) \rightarrow f(a_1, \ldots, a_n)$ muss formal wahr sein. In diesem Falle wird als Abkürzung auch einfach $g \rightarrow f = 1$ geschrieben.

Definition 18. *Implikant*
Seien f *und* i *n-stellige logische Funktionen,* $i(a_1, \ldots, a_n)$ *ein Konjunktionsterm und* $i \rightarrow f = 1$*, dann heißt* $i(a_1, \ldots, a_n)$ *Implikant von* $f(a_1, \ldots, a_n)$ *oder auch kurz von* f*.*

Bei einer Darstellung im KV-Diagramm würde das bedeuten, dass das 1-Muster der Funktion f das 1-Muster des Implikanten überdeckt.
Man sagt daher auch: f überdeckt i.

Beispiel 47. *f überdeckt i*
$f = a$, $i = a \cdot b$

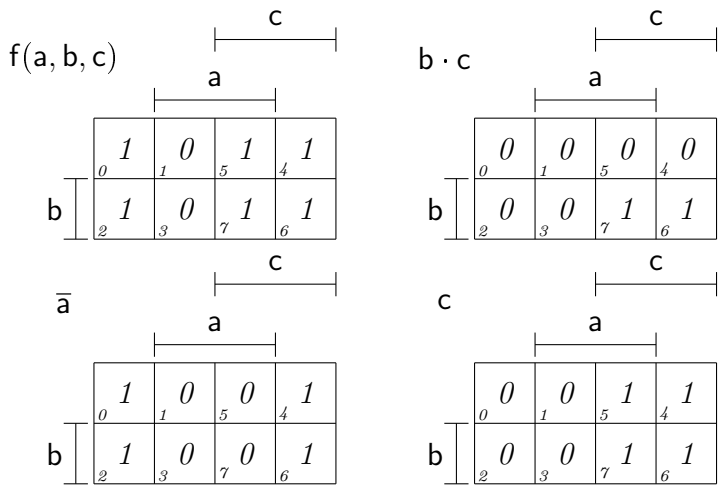
$$a \cdot b \rightarrow a = \overline{a \cdot b} + a = \overline{a} + \overline{b} + a = 1$$



Beispiel 48. *Implikanten der Funktion f*
Gegeben: $f(a,b,c) = m_0 + m_1 + m_2 + m_3 + m_5 + m_7$
und die Terme: m_5, $b \cdot c$, $\overline{a}$, b, c, $\overline{a} + c$

	c	b	a	f(a, b, c)	m_5	$b \cdot c$	$\overline{a}$	b	c	$\overline{a} + c$
0	0	0	0	1	0	0	1	0	0	1
1	0	0	1	0	0	0	0	0	0	0
2	0	1	0	1	0	0	1	1	0	1
3	0	1	1	0	0	0	0	1	0	0
4	1	0	0	1	0	0	1	0	1	1
5	1	0	1	1	1	0	0	0	1	1
6	1	1	0	1	0	1	1	1	1	1
7	1	1	1	1	0	1	0	1	1	1

m_5, $b \cdot c$, $\overline{a}$, c *sind Implikanten von f, b ist kein Implikant von f, da* $b \rightarrow f(a,b,c) = 0$ *für* $(a,b,c) = (1,1,0)$ *ist,* $\overline{a} + c$ *impliziert zwar* $f(a,b,c)$ *für alle Belegungen, ist jedoch kein Konjunktionsterm und daher kein Implikant von f.*
KV-Diagramme für $f(a,b,c)$, $b \cdot c$, $\overline{a}$, c*:*



Die 1-Muster der Implikanten werden vom 1-Muster der zugehörigen Funktion f überdeckt.

Zusammenfassung von Implikanten

Offenbar ist jeder Minterm, der zur KDNF einer Funktion gehört, ein Implikant dieser Funktion. Zwei Minterme, die sich nur in einer Variablen unterscheiden, lassen sich innerhalb der DNF zu einem neuen kürzeren Konjunktionsterm zusammenfassen (Absorptionsgesetz: $K \cdot a + K \cdot \bar{a} = K$). Dieser neu entstandene Term ist wieder Implikant der Funktion. Dies gilt nicht nur für Minterme, sondern allgemein für Implikanten, die sich mit Hilfe der Absorptionsgesetze ($K \cdot a + K \cdot \bar{a} = K$ und $K + K \cdot \bar{a} = K$) zu einem neuen kürzeren Konjunktionsterm zusammenfassen lassen.

Beispiel 49. *Zusammenfassung von Implikanten*
Implikant c im letzten Beispiel kann entstanden sein durch:
$m_1 = \bar{a} \cdot \bar{b} \cdot c$
$m_3 = \bar{a} \cdot b \cdot c$
$m_5 = a \cdot \bar{b} \cdot c$
$m_7 = a \cdot b \cdot c$

$m_1 + m_3 = \bar{a} \cdot c$ *und* $m_5 + m_7 = a \cdot c$ *ergeben*
$\bar{a} \cdot c + a \cdot c = c$

oder:

$m_1 + m_5 = \bar{b} \cdot c$ *und* $m_3 + m_7 = b \cdot c$ *ergeben*
$\bar{b} \cdot c + b \cdot c = c$

Die zwei Implikanten, aus denen sich der neue kürzere Implikant bilden lässt, sind auch Implikanten dieses neuen Konjunktionsterms.
Also gilt:
$m_1 \to \bar{a} \cdot c, m_3 \to \bar{a} \cdot c, m_5 \to a \cdot c, m_7 \to a \cdot c, \bar{a} \cdot c \to c, \bar{a} \cdot c \to c, c \to f(a, b, c)$

Satz 10. *Zusammenfassung von Implikanten*
Seien f eine n-stellige logische Funktion und $i_1(a_1, \ldots, a_n), i_2(a_1, \ldots, a_n)$ *Implikanten von* f.

1. *Lassen sich* $i_1(a_1, \ldots, a_n)$ *und* $i_2(a_1, \ldots, a_n)$ *nach den Absorptionsgesetzen zu einem neuen kürzeren Konjunktionsterm* $i_3(a_1, \ldots, a_n)$ *zusammenfassen, so ist* $i_3(a_1, \ldots, a_n)$ *wieder Implikant von* f.

2. $i_1(a_1, \ldots, a_n)$ *und* $i_2(a_1, \ldots, a_n)$ *sind Implikanten von* $i_3(a_1, \ldots, a_n)$.

3. *Jeder Konjunktionsterm einer DNF von* f *ist Implikant von* f.

4. *Jede DNF von* f *ist eine Disjunktion von Implikanten von* f.

3.6.2 Primimplikanten

Vorbemerkung

Im Beispiel wird deutlich, dass es in der Menge der Implikanten einer Funktion solche gibt, die sich nicht mehr disjunktiv zu neuen Implikanten verknüpfen lassen. Im Beispiel waren es $\overline{a}$ und c. Gerade solche, nicht weiter zusammenfassbare Implikanten sind für Minimierungsverfahren von Interesse.

Definition 19. *Primimplikant (Primterm)*
Sei $p(a_1, \ldots, a_n)$ *Implikant einer n-stelligen logischen Funktion f.* $p(a_1, \ldots, a_n)$ *heißt Primimplikant oder Primterm von* f*, wenn es keinen von* $p(a_1, \ldots, a_n)$ *verschiedenen Implikanten* $i(a_1, \ldots, a_n)$ *mit* $p \rightarrow i = 1$ *gibt.*

Folgerung

Primimplikanten einer vollständig definierten n-steligen logischen Funktion f sind genau die Implikanten von f, die sich mit keinem anderen Implikanten von f zu einem von diesem Primimplikanten verschiedenen Implikanten von f disjunktiv zusammenfassen lassen.

Beispiel 50. *Implikanten und Primimplikanten*
$f(a, b, c) = a \cdot \overline{b} + a \cdot b \cdot \overline{c} + a \cdot b \cdot c$
Folgende Implikanten von f lassen sich erkennen:

$a \cdot \overline{b}$, $a \cdot b \cdot \overline{c}$, $a \cdot b \cdot c$	*direkt aus* f
$a \cdot b$	*wegen* $a \cdot b \cdot \overline{c} + a \cdot b \cdot c = a \cdot b$
a	*wegen* $a \cdot b + a \cdot \overline{b} = a$

Die vollständige Liste der Implikanten von f *lautet jedoch:*
$a \cdot \overline{b} \cdot \overline{c}$, $a \cdot \overline{b} \cdot c$, $a \cdot b \cdot \overline{c}$, $a \cdot b \cdot c$, $a \cdot \overline{b}$, $a \cdot \overline{c}$, $a \cdot c$, $a \cdot b$, a
a *ist ein Primimplikant von* f.

Beispiel 51. *Implikanten und Primimplikanten*
$h(a, b, c) = \overline{a} \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c} = m_0 + m_3 + m_6$
Minterme $\overline{a} \cdot \overline{b} \cdot \overline{c}, \overline{a} \cdot b \cdot c, a \cdot b \cdot \overline{c}$ *sind die einzigen Implikanten von* h*. Da sie sich nicht zusammenfassen lassen, handelt es sich um Primimplikanten.*

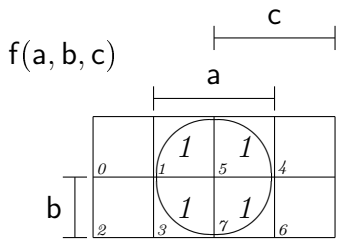
Satz 11. *Primtermdarstellung einer Funktion*
Sei f *eine vollständig definierte n-stellige logische Funktion und* $p(a_1, \ldots, a_n), i = 1, \ldots, k$ *seien alle Primimplikanten von* f.
Dann gilt:
$f(a_1, \ldots, a_n) = p_1(a_1, \ldots, a_n) + \cdots + p_k(a_1, \ldots, a_n)$.

Offenbar ist die Disjunktion aller Primterme eine spezielle disjunktive Normalform. Diese spezielle DNF, auch Primtermdarstellung der Funktion genannt, lässt sich mit den Absorptionsgesetzen nicht weiter vereinfachen. Diese Primtermdarstellung ist nicht in jedem Falle schon die kürzeste. Es kann Primimplikanten geben, auf die man bei der Darstellung der Funktion verzichten kann.

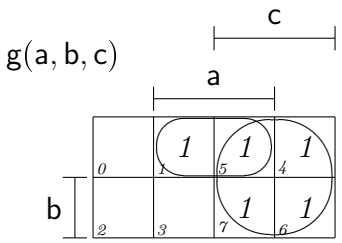
Implikanten und Primimplikanten im KV-Diagramm

Beispiel 52. *Implikanten und Primimplikanten*

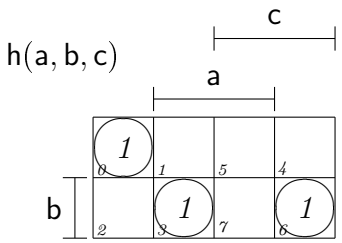
1. $f(a, b, c) = a \cdot \overline{b} + a \cdot b \cdot \overline{c} + a \cdot b \cdot c$



2. $g(a, b, c) = \overline{a} \cdot c + a \cdot c + a \cdot \overline{b} \cdot \overline{c}$



3. $h(a, b, c) = \overline{a} \cdot \overline{b} \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c}$



Aspekte zu Primimplikanten

Werden im KV-Diagramm 1-, 2-, $\ldots, 2^i$-Gruppen beieinanderliegender Einsen gebildet, so entspricht jede dieser Gruppen einem Implikanten. Die disjunktive Verknüpfung zwei-

er Implikanten zu einem neuen bedeutet dann die Zusammenlegung zweier gleichgroßer Gruppen zu einer nächstgrößeren. Lässt sich eine Gruppe nicht mehr vergrößern, liegt ein Primimplikant vor. Im letzten Beispiel gibt es keine benachbarten Felder, die mit „1“ belegt sind, deshalb sind die drei Minterme auch die einzigen Implikanten und daher Primimplikanten. Die von Implikanten belegten Felder von Einsen sind nicht immer disjunkt, auch die von Primimplikanten nicht, wie Beispiel (2) zeigt. Mit dem KV-Diagramm lassen sich also Primterme ermitteln. Alle Primimplikanten lassen sich aus anderen Implikanten kombinieren oder sind selbst Minterme. (2) zeigt aber, dass die benötigten Implikanten nicht immer von vornherein bekannt sind.

Ermittlung der Primimplikanten einer logischen Funktion

1. Aufstellung der KDNF der Funktion. (Alle in der KDNF auftretenden Minterme sind Implikanten).
2. Alle bisher vorhandenen Implikanten der Funktion werden – soweit möglich – paarweise disjunktiv zu neuen Implikanten verknüpft.
3. Hat man neue Implikanten erhalten, fährt man mit 2. fort.
4. Implikanten, die sich nicht mehr mit anderen zu neuen verknüpfen lassen, sind dann die Primimplikanten der Funktion.

Nur wenn man von der KDNF ausgeht, erhält man so alle Implikanten.

Beispiel 53. $g(a,b,c) = \overline{a}\cdot c + a\cdot c + a\cdot\overline{b}\cdot\overline{c}$

1. $g(a,b,c) = \overline{a}\cdot b\cdot c + \overline{a}\cdot\overline{b}\cdot c + a\cdot b\cdot c + a\cdot\overline{b}\cdot c + a\cdot\overline{b}\cdot\overline{c}$

2. Implikanten sind:

init	*2nd*	*3rd*
~~$\overline{a}\,b\,c$~~ ~~$\overline{a}\,\overline{b}\,c$~~	~~$\overline{a}\,c$~~	c
~~$\overline{a}\,b\,c$~~ ~~$a\,b\,c$~~	~~$b\,c$~~	c
~~$a\,b\,c$~~ ~~$a\,\overline{b}\,c$~~	~~$a\,c$~~	
~~$a\,b\,c$~~ ~~$a\,b\,\overline{c}$~~	~~$a\,b$~~	
~~$a\,\overline{b}\,c$~~ ~~$a\,\overline{b}\,\overline{c}$~~	$a\cdot\overline{b}$	

Es wurden die Implikanten durchgestrichen, die sich mit einem anderen disjunktiv zu einem neuen verknüpfen ließen.

3. Primterme sind die nicht markierten Implikanten $a\cdot\overline{b}$ *und* c.

Das beschriebene Verfahren wird sehr umfangreich und schnell unübersichtlich, wenn die zu untersuchende Funktion mehr als drei Variable und folglich eine „lange“ KDNF hat. Rechnergestütztes Verfahren: Quine McCluskey.

4 Analyse und Synthese von Schaltnetzen

Dieses Kapitel behandelt die Synthese von zweistufigen AND-OR-Schaltnetzen (disjunktive und konjunktive Normalform) unter Anwendung von Wahrheitstafeln und der De-Morgan'schen Gesetze. Im weiteren Verlauf werden auch Schaltnetze mit mehreren Stufen behandelt.

Das Ziel der Synthese ist die Begrenzung der Anzahl der Gattereingänge (Gate-Array-Technik, Vollkundenentwurf oder Standardbausteine) und die Optimierung der Laufzeiten und der Verlustleistung

4.1 Begriffe

Eine boolesche Funktion ist die Abbildung von n binären Variablen auf einen Wert (0 oder 1). Die einzelnen Variablensymbole (z.B. x, $\overline{x}$) werden Literale genannt. Variablen können drei Werte annehmen: 0, 1 und – (unbestimmt oder don't care). Literale lassen sich zu Min- und Maxtermen zusammensetzen. Ein Minterm ist ein Konjunktionsterm und definiert einen einzelnen Punkt innerhalb eines Wertefeldes von Variablen. Ein Maxterm hingegen definiert ein ganzes Wertefeld bis auf einen Punkt. Ein Implikant definiert eine Gruppe von 1-Punkten.

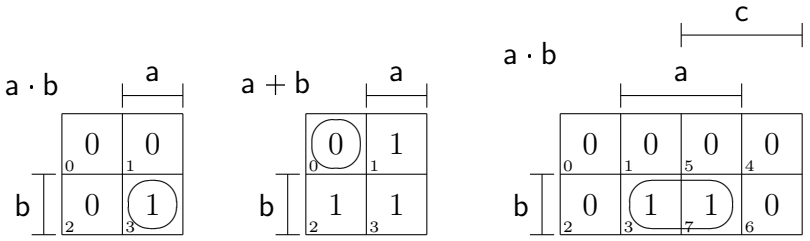


Abbildung 4.1: Ein Minterm, Maxterm und Implikant dargestellt in einem KV-Diagramm

Der Primimplikant ist ein Implikant, der so groß wie möglich ist, ohne dabei vollständig Teil anderer Implikanten zu sein. Dabei darf durch weglassen einer einzigen Variablen der Wert der Funktion nicht verändert werden.

Ein wichtiges Ziel der Analyse und Synthese ist die Optimierung der Funktion. Je nach Sichtweise kann „Optimierung" verschiedenes bedeuten. Aus mathematischer Sicht kann beispielsweise ein Ergebnis mit möglichst wenig Gattern (Termen) und möglichst wenig

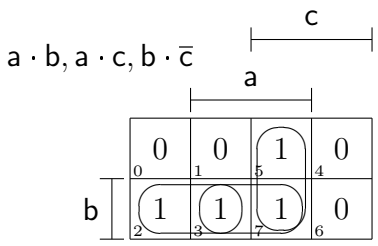


Abbildung 4.2: Primimplikanten in einem KV-Diagramm dargestellt

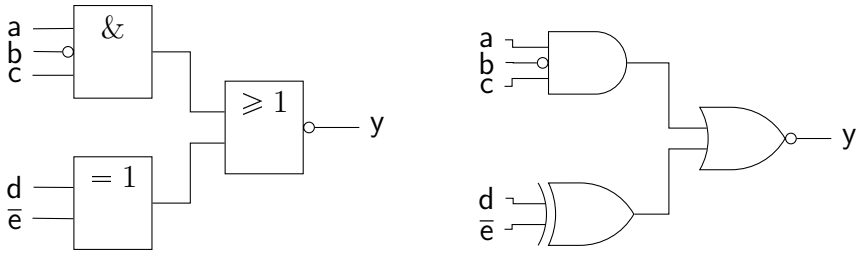
Eingängen (Literalen) als ideal angesehen werden. Aus physikalischer oder elektrotechnischer Sicht ist eine möglichst kleine Chipfläche (also wenige Bauelemente und gute Verdrahtbarkeit), eine minimale Laufzeit (wenige Gatterebenen, Fanin, Fanout) und eine minimale Verlustleistung (Gatterstruktur, Anzahl der Gatterebenen, Fanin, Fanout) optimal. Ein Problem ist, dass sich spezielle Schaltungstechniken wie dynamische Logik und Pass-Logik nur schwer in Logikansätze integrieren.

4.2 Gattersymbolik

Neben Funktionen, Wertetabellen und KV-Diagrammen ist noch eine weitere Darstellung etabliert - die Gattersymbolik. Bei der Gattersymbolik handelt es sich um eine grafische Darstellung, der die Grundstruktur einer Schaltung zu entnehmen ist. Dabei werden Terme der realisierten Funktion durch Grundgatter symbolisiert, deren Funktionalität den Grundfunktionen der Booleschen Algebra entsprechen. Die Anzahl der in den jeweiligen Term involvierten Eingangsvariablen spiegelt sich in der Anzahl der Eingänge eines jeden Gatters wieder. Die Negation von Eingangsvariablen und Termen wird durch eingangs- oder ausgangsseitige Kreise symbolisiert. In der Abbildung 4.3 wird die Funktion

$$y = \overline{(a \cdot \overline{b} \cdot c) + (d \oplus \overline{e})}$$

dargestellt.



(a) DIN-Symbole (b) US-Symbole

Abbildung 4.3: Beispiele für die Verwendung unterschiedlicher Gattersymbolik

Neben der DIN Symbolik aus Abbildung 4.3a, die auch im weiteren Verlauf des Skripts Verwendung findet, gibt es unter anderem auch die in Abbildung 4.3b gezeigte. Auf diese

stößt man hauptsächlich in englischsprachigen Publikationen, aber auch in Entwurfswerkzeugen, die im Digitalentwurf Gebrauch finden.

4.3 Elementare Verfahren zur Schaltnetzsynthese

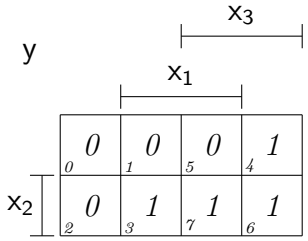
Zur Synthese von Schaltungen betrachten wir drei Möglichkeiten:

- Die Realisierung durch Gatter mit Hilfe von kanonischen Abbildungen
- Die Realisierung durch Schalter mit Hilfe des Entwicklungssatzes von Shannon
- Iterative Strukturen

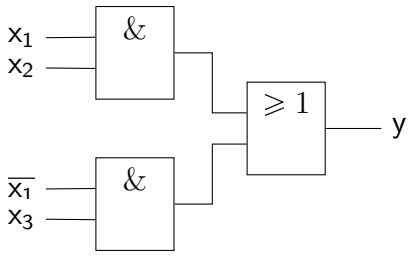
4.3.1 Kanonische Abbildungen

Bei der Bildung von kanonischen Abbildungen verwenden wir Normalformen. Dabei wird unterschieden zwischen der disjunktiven Normalform mit AND/OR-Strukturen (Darstellung der 1-Felder—NAND) und der konjunktiven Normalform mit OR/AND-Strukturen (Darstellung der 0-Felder—NOR).

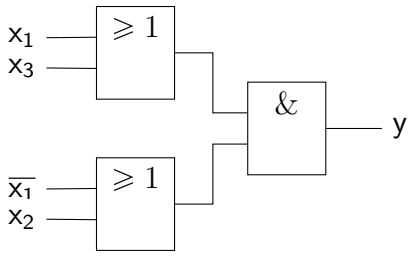
Beispiel 54. *Disjunktive und konjunktive Normalform*



$$y = x_1 \cdot x_2 + \overline{x_1} \cdot x_3 \qquad \textit{disjunktive Normalform}$$
$$\overline{y} = \overline{x_1} \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \qquad \textit{Umformung nach DeMorgan}$$
$$y = (x_1 + x_3) \cdot (\overline{x_1} + x_2) \qquad \textit{konjunktive Normalform}$$



(a) Gatterrealisierung einer DNF



(b) Gatterrealisierung einer KNF

Abbildung 4.4: Gatterrealisierungen unterschiedlicher Normalformen

4.3.2 Entwicklungssatz (Shannon)

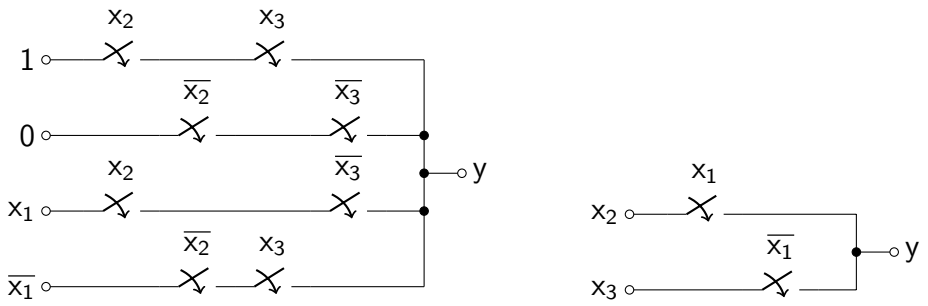
Der Entwicklungssatz nach Shannon erlaubt das Herausheben einer oder mehrerer Variablen aus einer Schaltungsfunktion.

Satz 12. *Entwicklungssatz (Shannon)*

$$y = f(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$$
$$y = x_i \cdot f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) + \overline{x_i} \cdot f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

Beispiel 55. *Shannon*

$y = x_1 \cdot x_2 + \overline{x_1} \cdot x_3$
$y = x_2 \cdot x_3 \cdot (1) + x_2 \cdot \overline{x_3} \cdot (x_1) + \overline{x_2} \cdot x_3 \cdot (\overline{x_1}) + \overline{x_2} \cdot \overline{x_3} \cdot (0)$ *nach* x_2, x_3 *entwickelt*
$y = x_1 \cdot (x_2) + \overline{x_1} \cdot (x_3)$ *nach* x_1 *entwickelt*



(a) nach x_2, x_3 entwickelt (b) nach x_1 entwickelt

Abbildung 4.5: Schaltermatrizen für Entwicklung nach unterschiedlichen Variablen

4.3.3 Iterative Strukturen

Einfache iterative Netzwerke bestehen aus eindimensionalen Arrays von Gattern. Damit lassen sich jedoch nicht alle Funktionen abbilden.

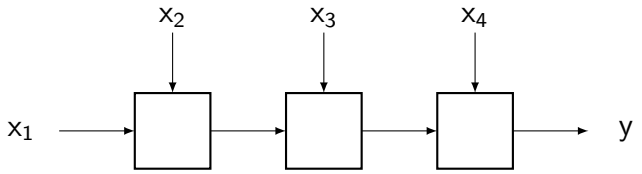
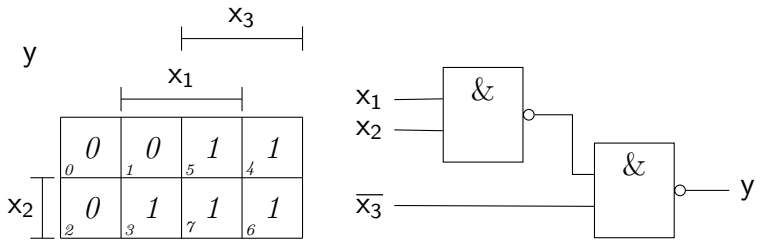


Abbildung 4.6: Grundstruktur eines eindimensionalen iterativen Schaltnetzes

Beispiel 56. *Iterative Implementierung*

$$y = x_1 \cdot x_2 + x_3$$



4.4 Minimale zweistufige AND-OR-Schaltnetze

Zur Minimierung einer Funktion in ein zweistufiges AND-OR-Schaltnetz müssen zuerst alle Primimplikanten der Funktion f bestimmt werden. Aus diesen werden dann die Primimplikanten ausgewählt, die alle 1-Punkte der Funktion f überdecken und minimale Kosten aufweisen.

4.4.1 Kosten

Kosten beschreiben die Zahl der Gatter und Eingänge z.B. bei der disjunktiven Normalform. Die Kosten eines Primimplikanten stehen in Relation zum Optimierungsziel (Zahl der Literale). Es wird davon ausgegangen, dass Literale sowohl komplementiert als auch nicht komplementiert zur Verfügung stehen.

Bei der Minimierung nach Gatteranzahl hat jeder Primimplikant mit zwei oder mehr Variablen einen Kostenwert von 1. Ein Primimplikant mit nur einer Variable hat einen Kostenwert von 0. Wenn mehrere Primimplikanten verknüpft werden, erhöhen sich die Kosten zusätzlich um 1. Der Kostenwert entspricht also der Anzahl der Gatter.

Soll nach Anzahl der Gattereingänge minimiert werden, so gilt für einen Primimplikanten mit einem Eingang ein Kostenwert von 1 und für Primimplikanten mit $n > 1$ Eingängen ein Kostenwert von $n + 1$.

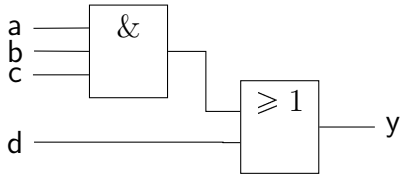


Abbildung 4.7: Schaltnetz mit zwei Gattern

4.4.2 Bestimmung der Primimplikanten

Bestimmung der Primimplikanten (Quine/McCluskey)

Eine Möglichkeit zur Bestimmung der Primimplikanten ist der Quine-McCluskey-Algorithmus:

1. Aufstellung einer Tabelle aller 1- und don't-care-Punkte ($-$) der Funktion f, sortiert nach der Anzahl der Variablen mit Wert 1, in die Gruppen $S_0, S_1, \ldots, S_n$. Um don't-cares richtig zu beachten muss dabei auch der Funktionswert in die Tabelle eingetragen werden.

2. In allen benachbarten Gruppen S_i und S_{i+1}, für alle i mit $0 \leqslant i < n$, werden Paare gesucht, bei denen sich nur eine Variable unterscheidet. Aus diesen Paaren werden neue Implikanten gebildet, bei denen jeweils die unterschiedliche Variable auf den Wert „unbestimmt" gesetzt wird. Der Funktionswert ist genau dann don't care, wenn beide kombinierten Terme an dieser Stelle don't care angeben, sonst 1. Die so erzeugten Implikanten werden in die Gruppe S'_i eingeordnet, die kombinierten Terme werden markiert.

3. Der zweite Schritt wird wiederholt, um beispielsweise S'_i mit S'_{i+1} zu S''_i zu kombinieren. Dabei müssen unbestimmte Variablen (don't care) immer übereinstimmen.

4. Wenn keine weiteren Paare gebildet werden können, sind alle Terme, die nie kombiniert wurden (also nicht markiert sind) Primimplikanten.

Beispiel 57. *Bestimmung der Primimplikanten nach Quine/McCluskey*

Eingangsvariable: x_3, x_2, x_1

1-Punkte: $\{(1,0,0),(0,1,0),(0,0,1),(1,1,0)\}$; m_4, m_2, m_1, m_6

(–)-Punkte: $\{(1,1,1)\}$; m_7

		x_3	x_2	x_1	f	
	m_4	1	0	0	1	*!*
S_1	m_2	0	1	0	1	*!*
	m_1	0	0	1	1	
S_2	m_6	1	1	0	1	*!*
S_3	m_7	1	1	1	–	*!*
		1	–	0	1	
S'_1		–	1	0	1	
S'_2		1	1	–	1	

Ergebnis: Primimplikanten: $\{x_1 \cdot \overline{x_2} \cdot \overline{x_3}, \overline{x_1} \cdot x_3, \overline{x_1} \cdot x_2, x_2 \cdot x_3\}$

Bestimmung der Primimplikanten (Tison 1965)

Eine weitere Möglichkeit zur Bestimmung der Primimplikanten ist der Algorithmus von Tison:

1. Die disjunktive Funktion $f = A + B + \ldots + G$ mit $S = \{A, B, \ldots, G\}$ ist als Ausgangspunkt gegeben. Zuerst werden alle Terme gelöscht, die vollständig überdeckt werden. Dann wird eine Variable ausgewählt, die in negierter und nichtnegierter Form vorkommt.

2. Bilden der Überdeckungsmengen aus jedem Term mit der nichtnegierten Variable mit jedem Term der negierten Variable. Die Überdeckungsmengenterme werden zur Menge S hinzugefügt und vollständig überdeckte Terme werden gelöscht.

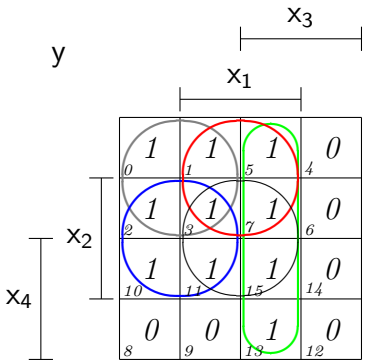
3. Wiederholung von Schritt 2 mit allen anderen Variablen, die in negierter und nichtnegierter Form vorkommen. Wenn die Voraussetzung für eine Variable entfällt, entfällt auch dieser Schritt für diese Variable.

4. S enthält jetzt alle Primimplikanten.

Beispiel 58. *Bestimmung der Primimplikanten nach Tison*

$$y = x_1 \cdot x_2 \cdot x_4 + x_1 \cdot x_3 + \mathbf{x_2 \cdot \overline{x_3}} + \overline{x_3} \cdot \overline{x_4}$$

Die beiden Variablen die sowohl negiert als auch nichtnegiert auftreten sind x_3 *und* x_4*. Von diesen wird zunächst* x_3 *gewählt. Ein mögliches Termpaar zum Bilden der Überdeckungsmengen ist* $x_1 \cdot x_3$ *und* $x_2 \cdot \overline{x_3}$*. Der Überdeckungsmengenterm lautet* $x_1 \cdot x_2$*. Er entsteht durch das „verunden" der beiden Terme bei Vernachlässigung der vorher gewählten Variable* x_3*. Das zweite und letzte mögliche Paar ist* $x_1 \cdot x_3$ *und* $\overline{x_3} \cdot \overline{x_4}$*. Daraus folgt analog* $x_1 \cdot \overline{x_4}$*. Der Term* $x_1 \cdot x_2 \cdot x_4$ *der Ursprungsfunktion wird durch den neu gefundenen Term* $x_1 \cdot x_2$ *überdeckt und entfällt somit.*



Ergebnis: Primimplikanten: $x_1 \cdot x_3$, $x_2 \cdot \overline{x_3}$, $\overline{x_3} \cdot \overline{x_4}$, $x_1 \cdot x_2$, $x_1 \cdot \overline{x_4}$

Vergleich der Verfahren zur Bestimmung der Primimplikanten

Während die Tison-Methode vor allem bei großer Variablenzahl flexibel ist, kann die Quine-McCluskey-Methode don't cares berücksichtigen, ist günstiger für rechnergestützte Verfahren und vorteilhaft, wenn die Funktion durch Minterme vergegeben ist.

4.4.3 Kostenoptimierung (minimale Überdeckung)

Die gefundenen Menge an Primimplikanten ist in der Regel nicht frei von Redundanz. Zur weiteren Kostenoptimierung müssen die Primimplikanten bestimmt werden, die zwar alle 1-Punkte der Funktion abdecken, sich dabei aber minimal Überdecken und somit minimale Kosten verursachen. Die Minimierung von Hand kann in tabellarischer Form durchgeführt werden. Dazu wollen wir vereinbaren, dass x_1 das niederwertigste Bit und x_n das höchstwertigste Bit ist.

Für eine Reihe der folgenden Verfahren können die bestimmenden Faktoren (Primimplikanten, Minterme) nach Kosten gewichtet werden. Da diese Gewichtung für das Verständnis der Verfahren jedoch unerheblich ist, gehen wir bei der Erläuterung der Verfahren von Primimplikanten und Mintermen mit egalisierter Gewichtung aus.

Wir werden vier Verfahren behandeln, die in der Regel in einer Kombination angewandt werden:

- wesentlicher Primimplikant
- Reihendominanz
- Spaltendominanz
- Branching

Der Einsatz einiger dieser Verfahren ist in der Praxis erst bei sehr komplexen Funktionen, die jedoch nicht didaktisch zur Erläuterung geeignet sind, ratsam. Im Folgenden werden daher Beispiele bemüht, die in dieser Form nicht unmittelbar aufgestellt werden können (zum Beispiel als Ergebnis eines Quine-McCluskey). Ihr Zustandekommen kann mit einer vorher erfolgten, in dem jeweiligen Beispiel aber nicht aufgeführten Anwendung anderer Minimierungsverfahren erklärt werden.

Methode der wesentlichen Primimplikanten

		m_1	m_2	m_4	m_6	m_7
P_1	$x_1 \cdot \overline{x_2} \cdot \overline{x_3}$	1				
P_2	$\overline{x_1} \cdot x_3$			1	1	
P_3	$\overline{x_1} \cdot x_2$		1		1	
P_4	$x_2 \cdot x_3$				1	–

Abbildung 4.8: Besetzungstabelle der Primimplikanten aus dem Beispiel 57

Ein wesentlicher Primimplikant überdeckt als einziger einen Minterm einer Funktion. Wenn P_i als einziger Primimplikant m_i überdeckt, muss P_i ein Bestandteil der minimalen Überdeckung sein. Alle Spalten (Minterme), die von P_i überdeckt werden, können dann eliminiert werden.

Beispiel 59. *Methode des wesentlichen Primimplikanten anhand Abb. 4.8*
Der Primimplikant P_3 *überdeckt als einziger den Minterm* m_2 *und ist damit ein wesentlicher Primimplikant.* P_3 *überdeckt weiterhin* m_6*. Es können also insgesamt zwei Spalten entfernt werden. Mit der Eliminierung von* m_6 *kann auch* P_4 *entfernt werden, denn der Funktionswert von* m_7 *ist lediglich don't care. Es bleibt damit die reduzierte Besetzungstabelle:*

	m_1	m_4
P_1	1	
P_2		1

Somit ist die minimale Überdeckung der Primimplikanten P_1, P_2, P_3 *bzw.* $x_1 \cdot \overline{x_2} \cdot \overline{x_3}$, $\overline{x_1} \cdot x_3$, $\overline{x_1} \cdot x_2$

Methode der Reihendominanz

Nachdem die Menge an Primtermen beispielsweise durch die Methode der wesentlichen Primimplikanten reduziert wurde, kann oft noch eine weitere Vereinfachung erfolgen.

Wie der Name Reihendominanz schon suggeriert, wird bei dieser Methode eine Reihe P_i gesucht, die eine andere Reihe P_j dominiert. Dominieren bedeutet, dass die Reihe P_i alle 1-Punkte der Reihe P_j abdeckt und zudem geringere (oder gleiche) Kosten aufweist ($C(P_i) \leqslant C(P_j)$). In diesem Fall kann die dominierte Reihe P_j entfernt werden, da so die minimale Kostenfunktion durch P_i erhalten bleibt.

Zusammengefasst: Seien $\mathbf{M_{P_x}}, \mathbf{M_{P_y}}$ Mengen aller durch P_x oder P_y abgedeckten Minterme. Dann gilt für $\mathbf{M_{P_x}} \subset \mathbf{M_{P_y}}$: P_y dominiert P_x.

Beispiel 60. *Methode der Reihendominanz*

	m_a	m_b	m_c	m_d	m_e
P_1	**1**	**1**			
P_2	**1**	**1**	1		1
P_3			1	1	1
P_4	1			1	

In diesem Beispiel dominiert die Reihe P_2 *die Reihe* P_1*, die somit eliminiert werden kann. Daraus ergibt sich die reduzierte Überdeckungstabelle:*

	m_a	m_b	m_c	m_d	m_e
P_2	**1**	**1**	1		1
P_3			1	1	1
P_4	1			1	

In dieser ist P_2 *ein wesentlicher Primimplikant für* m_b*. Nach Anwenden der Methode des wesentlichen Primimplikanten bleibt nur die Spalte* m_d *übrig:*

	m_d
P_3	1
P_4	1

Jetzt dominiert P_3 P_4*, allerdings dominiert* P_4 *auch* P_3*. Der Primimplikant wird nach Kosten ausgewählt. In diesem Beispiel sind die minimalen Überdeckungen also:* $\{\mathsf{P}_2, \mathsf{P}_3\}$ *oder* $\{\mathsf{P}_2, \mathsf{P}_4\}$*.*

Methode der Spaltendominanz

Dominanz lässt sich auch auf Spalten anwenden, dabei muss jedoch eine andere Methode verwendet werden. Eine Spalte m_i dominiert eine Spalte m_j, wenn jeder Primimplikant, der m_j überdeckt, maximal auch m_i überdeckt. Zum Beispiel kann m_j 1-Punkte in allen Reihen haben, in denen auch m_i 1-Punkte hat. Die Spalte m_j kann eliminiert werden, denn jede Überdeckungsmenge der reduzierten Tabelle überdeckt auch m_i.

Zusammengefasst: Seien $\mathbf{P}_{\mathbf{m_x}}, \mathbf{P}_{\mathbf{m_y}}$ Mengen aller Primterme, die m_x oder m_y abdecken. Dann gilt für $\mathbf{P}_{\mathbf{m_x}} \subseteq \mathbf{P}_{\mathbf{m_y}}$: m_x dominiert m_y.

Beispiel 61. *Methode der Spaltendominanz*

	m_a	m_b	m_c	m_d	m_e
P_1	**1**	**1**			
P_2	**1**	**1**	1		1
P_3			1	1	1
P_4	1			1	

m_b *dominiert* m_a*. Außerdem dominiert* m_c m_e *und umgekehrt. Es können also* m_a *und* m_e *(oder* m_c*) eliminiert werden.*

	m_b	m_c	m_d
P_1	1		
P_2	1	1	
P_3		1	1
P_4			1

P_2 *dominiert* P_1 *und* P_3 *dominiert* P_4 *(Reihendominanz). Das Ergebnis lautet somit:* $\{\mathsf{P}_2, \mathsf{P}_3\}$ *ist die minimale Überdeckung*

Methode des Branching

Die drei bisher vorgestellten Methoden führen nicht zum Ziel, wenn keine der entsprechenden Kriterien zutreffen. Diese Tabellen werden als zyklisch bezeichnet und können mit der Methode des Branching reduziert werden.

Dazu wird eine Teilmenge R von Reihen ausgewählt, so dass jede Lösung ein Element von R enthalten muss. Danach wird P_i aus R gewählt. Die Spalten, die von P_i überdeckt werden, werden eliminiert. P_i ist nun ein Bestandteil der Lösung. Dieser Ablauf wird mit einer anderen Reihe von R wiederholt. Gegebenenfalls muss zum Auffinden des Minimums jedes Element von R gewählt werden.

Beispiel 62. *Methode des Branching*

	m_a	m_b	m_c	m_d	m_e
P_1	1	1			1
P_2		1	1		
P_3	1		1		
P_4				1	1
P_5			1	1	
P_6		1		1	

Es wird die Spalte m_a *ausgewählt. Aus* m_a *ist ersichtlich, dass jede Überdeckungsmenge* P_1 *und/oder* P_3 *enthalten muss. Daraus folgt Branching um* m_a*. Es gibt zwei Möglichkeiten:*

- P_1 *wird gewählt und alle Spalten von* P_1 *werden eliminiert.*
- P_3 *wird gewählt und alle Spalten von* P_3 *werden eliminiert.*

$\mathbf{P_1}$	m_c	m_d
P_2	1	
P_3	1	
P_4		1
P_5	1	1
P_6		1

$\mathbf{P_3}$	m_b	m_d	m_e
P_1	1		1
P_2	1		
P_4		1	1
P_5		1	
P_6	1	1	

Beide Tabellen können nach den bisherigen Regeln weiter reduziert werden. Die minimalen Überdeckungsmengen sind $\{P_1, P_5\}$*,* $\{P_3, P_6, P_1\}$*,* $\{P_3, P_6, P_4\}$ *oder* $\{P_3, P_1, P_4\}$*. Gegenüber den anderen drei Lösungen weist* $\{P_1, P_5\}$ *die geringeren Kosten auf und ist deshalb die optimale Lösung dieser Spaltenauswahl.*

Hinweis Branching kann mit jeder Spalte durchgeführt werden. Jedoch sollten aus Gründen minimaler Rechenzeit die Spalten mit geringer Anzahl von 1-Punkten gewählt werden.

4.5 Schaltnetze mit mehreren Stufen

Schaltnetze mit mehreren Stufen bieten einen Kompromiss aus Hardwareaufwand und Geschwindigkeit. Sie ermöglichen es, den Ein- und Ausgangslastfaktor zu beschränken (fanin, fanout, z.B. Gate-Array-Technik). Dies wird durch die Aufspaltung von Gattern mit vielen Eingängen möglich, denn dadurch kann die Zahl der Eingänge beschränkt werden. Die Ausgangslast sinkt somit.

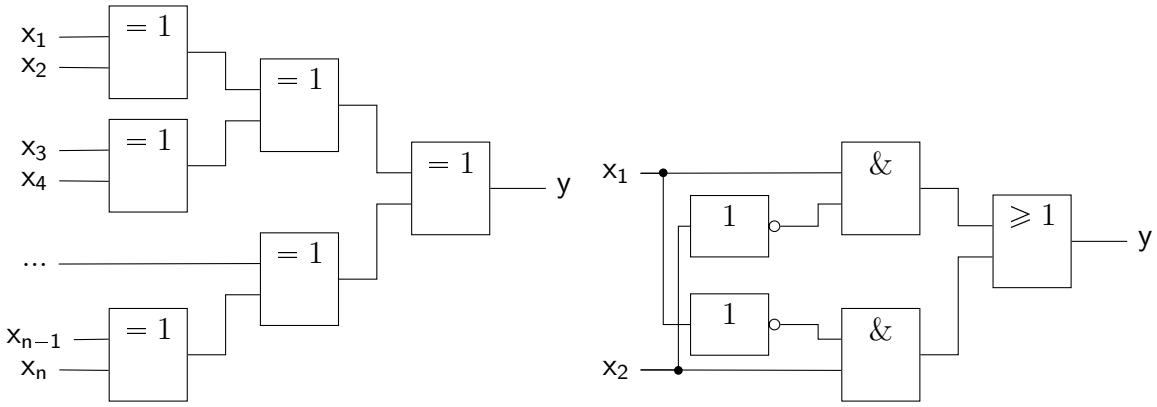
4.5.1 Baumstruktur

Eine Schaltung, bei der die Ausgänge eines jeden Gatters nur mit einem Gattereingang verknüpft werden, wird als Baumstruktur bezeichnet.

Beispiel 63. *Mehrstufiges Schaltnetz: Parityfunktion*

$$f(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

Ist das Ziel eine Realisierung mit möglichst wenigen Stufen, so erfordert diese Realisierung die Verwendung von 2^{n-1} *AND-Gatter mit jeweils* n *Eingängen, ein OR-Gatter mit* 2^{n-1} *Eingängen und* n *Interver. Unter Berücksichtigung der Inverter ergeben sich insgesamt drei Stufen. Der Aufwand lässt sich jedoch reduzieren, indem Parityschaltungen für 2 Bit kaskadierend miteinander verbunden werden. Es sind dazu* $\log_2 n$ *bzw. unter Berücksichtigung der internen Struktur der einzelnen Stufen* $3 \log_2 n$ *Stufen notwendig. Der Aufwand verringert sich damit auf* $3(n-1)$ *NAND-Gatter mit 2 Eingängen und* $2(n-1)$ *Inverter. Die Erhöhung der Stufenzahl führt jedoch zur Erhöhung der Laufzeiten.*



(a) Parity-Baum (b) Parity Stufe für 2 Bit

Abbildung 4.9: Parity-Baum

4.5.2 Faktorisierung

Die Umformung in Schaltungen mit mehr als zwei Stufen kann durch Faktorenzerlegung erfolgen.

Beispiel 64. *Zerlegung in Faktoren*

$$f = x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4} + x_1 \cdot x_2 \cdot \overline{x_3} \cdot x_4$$

Die Funktion kann wegen des gemeinsamen Faktors $x_1 \cdot x_2$ *zerlegt werden in:*

$$f = x_1 \cdot x_2 \cdot (x_3 \cdot \overline{x_4} + \overline{x_3} \cdot x_4)$$

Die Stufenzahl erhöht sich dabei von 2 auf 3, die Zahl der Eingänge pro Gatter reduziert sich von 4 auf 2. Es ergibt sich jedoch das Problem der unterschiedlichen Pfadlängen.

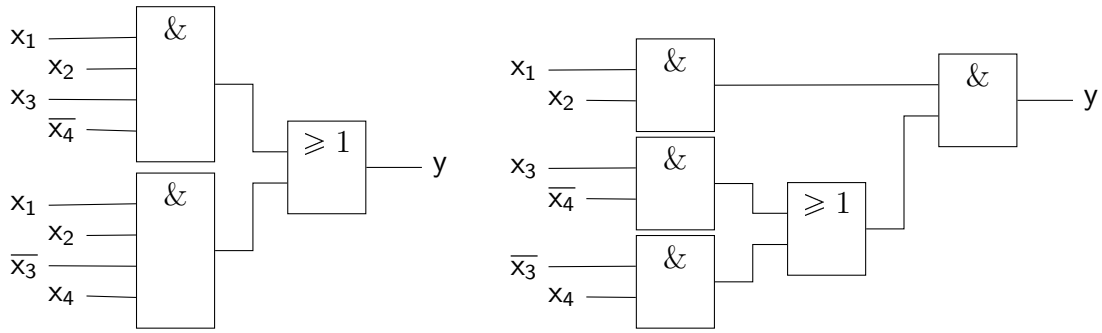


Abbildung 4.10: zweistufige Realisierung und dreistufige Realisierung mit zerlegten Faktoren

4.5.3 Zusammenfassung

Jedes Schaltnetz kann in eine äquivalente Schaltung umgeformt werden, um die Randbedingungen bezüglich der Eingangs- und Ausgangsfaktoren (fanin, fanout) zu erfüllen.

Beispiel 65. *n-Eingangs-AND-Gatter*
Ein AND-Gatter mit n *Eingängen kann durch einen Baum von 2-Eingang-AND-Gatter ersetzt werden. Die Anzahl der Stufen beträgt* $\log_2 \mathsf{n}$ *für* $\mathsf{n} = 2^{\mathsf{N}}$ $\mathsf{n}, \mathsf{N} \in \mathbb{N}$ *bzw.* $\log_2(\mathsf{n}+1)$ *für* $\mathsf{n} \neq 2^{\mathsf{N}}$.

Ein Ausgangslastfaktor von n kann ersetzt werden durch eine Baumstruktur mit $\log_2 \mathsf{n}$ Stufen, wobei jedes Gatter einen Eingang und ein Fanout von 2 aufweist (Verstärkerstufe). Bisher existiert allerdings kein allgemeiner Algorithmus zur effizienten Berechnung optimaler Lösungen von mehrstufigen Schaltnetzen.

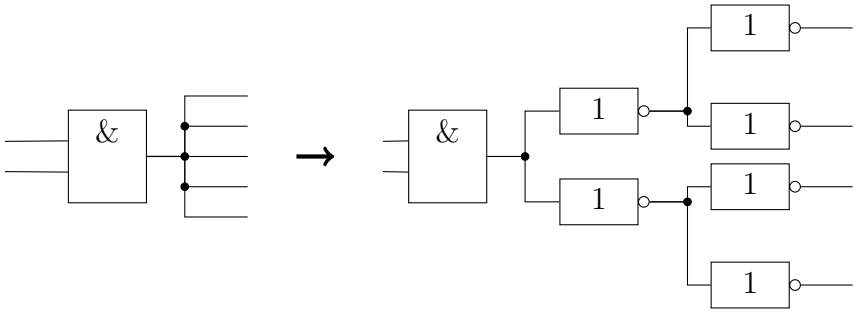


Abbildung 4.11: Reduzierung der Ausgangslast durch Baumstruktur

5 Komplexe Schaltnetze

5.1 Codierer

Funktion

Ein Codierer entspricht einem Schaltnetz, welches eine Menge von Eingangswerten in eine Menge von Ausgangswerten übersetzt (**Codeumsetzung**).

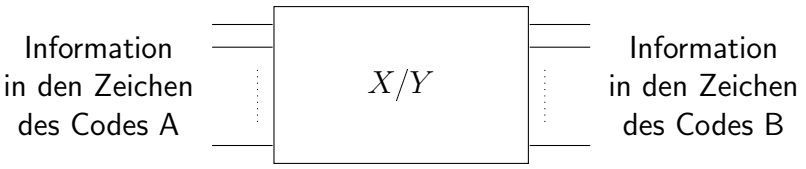


Abbildung 5.1: Schaltzeichen eines Codierers

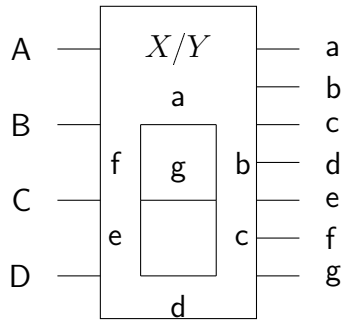
Das Schaltzeichen deutet an, dass ein Codierer ein Schaltnetz ist, das eine Vektorfunktion realisiert ($\overrightarrow{y} = \mathsf{f}(\overrightarrow{x})$). Die Anzahl der Ein- und Ausgänge eines Codeumsetzers ist abhängig von der Wortlänge des Binärcodes, in dem die Information dargestellt ist.

Beispiel 66. *Schaltnetzentwurf für die 8421-BCD zu 7-Segment-Umsetzung*

Aufgabe:
Dezimalziffern werden häufig in einem ***BCD-Code*** *(Binary Coded Decimal) dargestellt und in 7-Segmenteinheiten eingesetzt. Dies kann durch einen Codeumsetzer realisiert werden.*

Annahme:
Die Dezimalziffern liegen im 8421-BCD-Code vor.

Codeumsetzung:
1. Die Zuordnung (Codierung) der Dezimalziffern vom 8421-BCD-Code zu den Segmenten der 7- Segment-Anzeige wird in einer Wertetabelle festgelegt.



(a) Blockschaltbild

Dezimal-ziffer	8421-BCD-Code D	C	B	A	7-Segment-Code a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10	1	0	1	0	–	–	–	–	–	–	–
11	1	0	1	1	–	–	–	–	–	–	–
12	1	1	0	0	–	–	–	–	–	–	–
13	1	1	0	1	–	–	–	–	–	–	–
14	1	1	1	0	–	–	–	–	–	–	–
15	1	1	1	1	–	–	–	–	–	–	–

(b) Wertetabelle

Abbildung 5.2: 8421-BCD-Code zu 7-Segment Codierer

2. Für die Ausgangsvariablen $\mathsf{a}, \mathsf{b}, ..., \mathsf{g}$ *werden die Funktionsgleichungen aus der Wertetabelle in der DNF hergeleitet. Nimmt man an, dass die Pseudotetraden nicht vorkommen (Zustand 10 bis 15), können diese bei der Vereinfachung im KV-Diagramm als don't care-Terme benutzt werden.*

3. Nach der Minimierung lauten die Funktionsgleichungen für die Ausgangsvariablen $\mathsf{a}, \mathsf{b}, ..., \mathsf{g}$ *in DNF:*

$$\mathsf{a} = \mathsf{D} + \overline{\mathsf{A}}\ \overline{\mathsf{C}} + \mathsf{A}\ \mathsf{C} + \mathsf{A}\ \mathsf{B}$$
$$\mathsf{b} = \overline{\mathsf{C}} + \mathsf{A}\ \mathsf{B} + \overline{\mathsf{A}}\ \overline{\mathsf{B}}$$
$$\mathsf{c} = \mathsf{A} + \overline{\mathsf{B}} + \mathsf{C}$$
$$\mathsf{d} = \overline{\mathsf{A}}\ \mathsf{B} + \overline{\mathsf{A}}\ \overline{\mathsf{C}} + \mathsf{B}\ \overline{\mathsf{C}} + \mathsf{A}\ \overline{\mathsf{B}}\ \mathsf{C}$$
$$\mathsf{e} = \overline{\mathsf{A}}\ \mathsf{B} + \overline{\mathsf{A}}\ \overline{\mathsf{C}}$$
$$\mathsf{f} = \mathsf{D} + \overline{\mathsf{A}}\ \overline{\mathsf{B}} + \overline{\mathsf{A}}\ \mathsf{C} + \overline{\mathsf{B}}\ \mathsf{C}$$
$$\mathsf{g} = \mathsf{D} + \overline{\mathsf{A}}\ \mathsf{B} + \overline{\mathsf{B}}\ \mathsf{C} + \mathsf{B}\ \overline{\mathsf{C}}$$

5.2 Decodierer

Funktion

Decodierer sind Codierer mit mehreren Ein- und Ausgängen, bei denen für jede Kombination von Eingangssignalen immer **nur je ein Ausgang** ein Signal abgibt. Am Ausgang eines Decodierers liegt die Information in den Zeichen eines 1-aus-n-Codes vor.

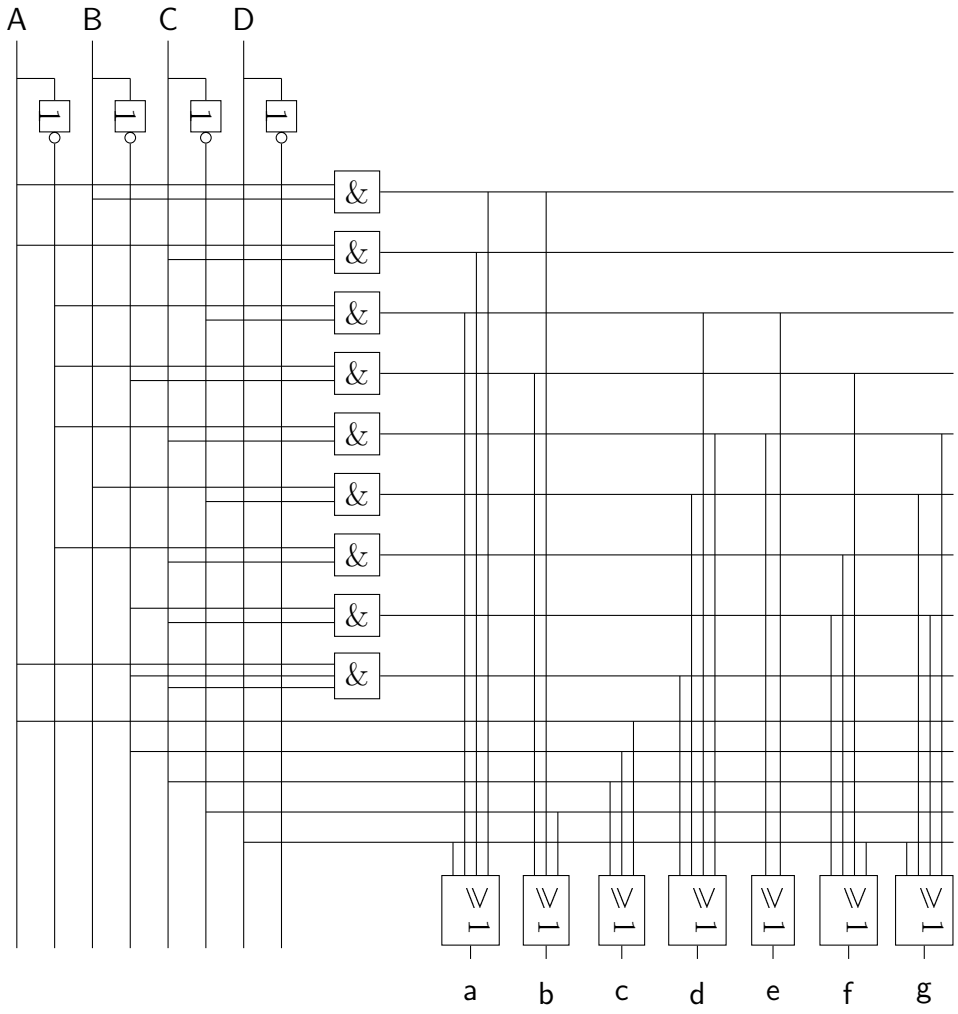


Abbildung 5.3: Schaltnetz für einen 8421-Code in 7-Segment Codierer

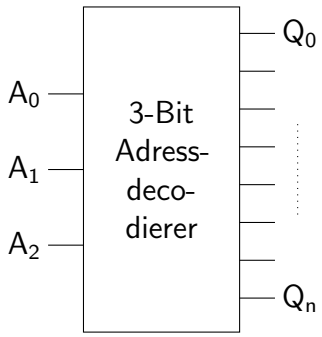
Definition 20. *1-aus-n-Code*
Ein 1-aus-n-Code ist dadurch gekennzeichnet, dass die Anzahl der Binärstellen gleich der Anzahl der darzustellenden Zeichen ist. D.h. führt eine Bitstelle des Codewortes ein 1- (bzw. 0-) Signal, dann führen alle anderen Stellen ein 0- (bzw.1-) Signal.

Beispiel 67. *3-Bit Adressdecodierer*

Über die Adresseingänge wird ein Ausgang des Decodierers angewählt, der dann 1-Signal führt (siehe Wertetabelle). Hat ein Adressdecodierer n*-Eingänge, dann können* 2^n*-Ausgänge angewählt werden.*

Anwendungen

Decodierer finden als Adressdecodierer Anwendung in digitalen Rechensystemen. Beispielsweise werden Peripheriegeräte oder Speicherzellen mit einer Adresse angewählt.



(a) Blockschaltbild

Eingang			Ausgang							
A_2	A_1	A_0	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**

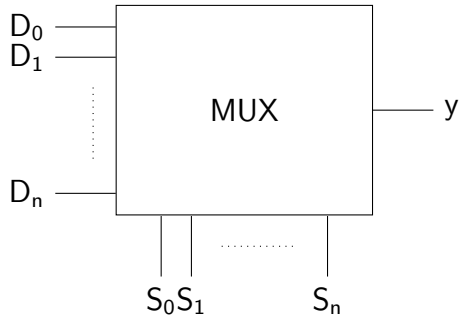
(b) Wertetabelle

Abbildung 5.4: 3-Bit Adressdecodierer

5.3 Multiplexer

Funktion

Ein Multiplexer ist ein **auswählendes** Schaltnetz. Über Steuereingänge wird einer der Dateneingänge auf den Ausgang durchgeschaltet.



(a) Blockschaltbild

$D_0, ..., D_n$	Dateneingänge
$S_0, ..., S_m$	Steuereingänge
Y	Ausgang
G	Steuereingänge steuern Dateneingänge durch UND-Verknüpfung
$\overline{E}$	Aktivierungseingang Enable (**Low-aktiv**)

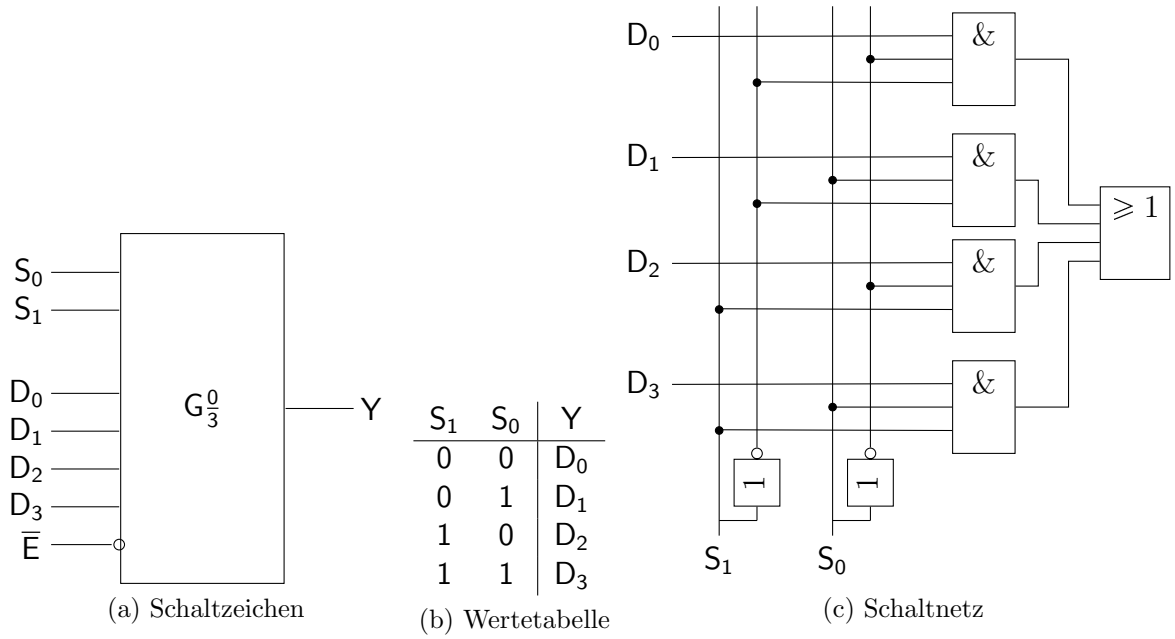
(b) Ein- und Ausgänge

Abbildung 5.5: Multiplexer

Beispiel 68. *4-zu-1 Multiplexer*
Die Funktion eines 4-zu-1 Multiplexers wird durch eine Wertetabelle beschrieben.

Logikfunktion
Jeweils ein Dateneingang wird mit dem entsprechenden Steuerwort UND-verknüpft und auf den Ausgang geschaltet.

$$Y = \overline{S_0}\ \overline{S_1}\ D_0 + S_0\ \overline{S_1}\ D_1 + \overline{S_0}\ S_1\ D_2 + S_0\ S_1\ D_3$$



(a) Schaltzeichen (b) Wertetabelle (c) Schaltnetz

Abbildung 5.6: 4-zu-1 Multiplexer

Schaltnetz
Aus der Logikfunktion ergibt sich das Schaltnetz (siehe Abbildung).

Multiplexer mit größerer Wortbreite

Typische Datenwortlängen, die in einem Rechenwerk verarbeitet werden, sind 4 bis 64 Bits. Ein Multiplexer, der Datenworte von einem auswärtigen Register auf die ALU durchschaltet, muss auch 4 bis 64 Bit-Eingangsdaten auf den Ausgang mit entsprechender Wortlänge schalten.

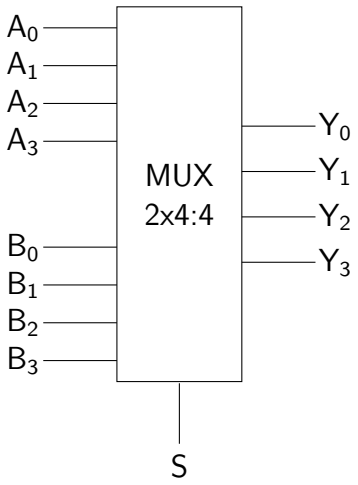
Beispiel 69. *2x4-zu-4 Multiplexer*

Es werden die Dateneingänge $A_0, .., A_3$ *oder* $B_0, .., B_3$ *auf den Ausgang* Y *durchgeschaltet. Die Funktion des 2x4-Bit zu 4-Bit Multiplexers wird durch die Wertetabelle beschrieben. Die Auswahl der Eingangsdatenworte wird durch die Steuervariable S festgelegt.*

5.4 Demultiplexer

Funktion

Der Demultiplexer ist ein **verteilendes** Schaltnetz. In Abhängigkeit vom Steuerwort wird ein Dateneingang auf einen der möglichen Datenausgänge geschaltet. Mit n Steuereingängen kann auf einen von 2^n Datenausgängen verteilt werden.

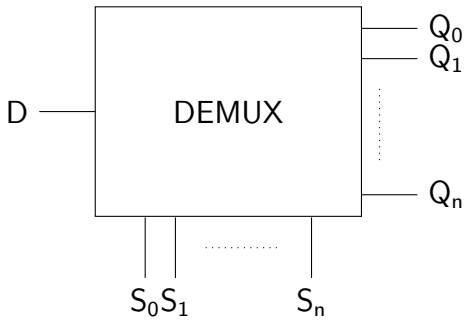


(a) Blockschaltbild

S	Y
0	$A_{0...3}$
1	$B_{0...3}$

(b) Wertetabelle

Abbildung 5.7: 2x4-zu-4 Multiplexers



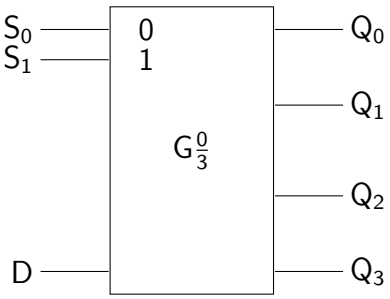
(a) Blockschaltbild

D	Dateneingang
$S_0, ..., S_m$	Steuereingänge
$Q_0, ..., Q_n$	Ausgänge
G	Steuereingänge steuern Dateneingang durch UND-Verknüpfung

(b) Ein- und Ausgänge

Abbildung 5.8: Demultiplexer

Beispiel 70. *1-zu-4 Demultiplexer*



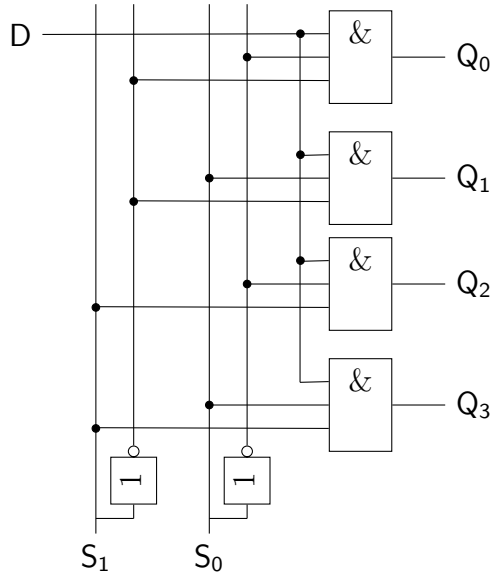
(a) Schaltzeichen

S_1	S_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

(b) Wertetabelle

$$Q_0 = \overline{S_0}\ \overline{S_1}\ D$$
$$Q_2 = \overline{S_0}\ S_1\ D$$
$$Q_1 = S_0\ \overline{S_1}\ D$$
$$Q_3 = S_0\ S_1\ D$$

(c) Schaltfunktionen



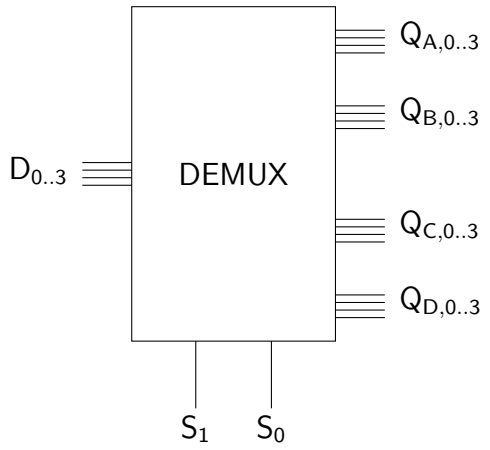
(d) Schaltnetz

Abbildung 5.9: 1-zu-4 Demultiplexer

Die Steuerworteingänge und der Dateneingang sind wie beim Multiplexer UND-verknüpft. Aus der Wertetabelle ergeben sich für die Ausgänge die entsprechenden Schaltfunktionen.

Mehrfach-Demultiplexer

Um ein Mehr-Bit Dateneingangswort $D_{0...3}$ auf verschiedene Mehr-Bit Ausgangskanäle $Q_{A,...,D}$ schalten zu können, müssen entsprechende Demultiplexer vorhanden sein. Die Funktion eines 1x4-Bit zu 4x4-Bit Demultiplexers wird durch folgende Wertetabelle beschrieben:



(a) Blockschaltbild

S_1	S_0	$Q_{A_{0...3}}$	$Q_{B_{0...3}}$	$Q_{C_{0...3}}$	$Q_{D_{0...3}}$
0	0	$D_{0...3}$	0	0	0
0	1	0	$D_{0...3}$	0	0
1	0	0	0	$D_{0...3}$	0
1	1	0	0	0	$D_{0...3}$

(b) Wertetabelle

Abbildung 5.10: 1x4-zu-4x4 Demultiplexers

5.5 Komparatoren

Funktion

Komparatoren **vergleichen** analoge oder binäre Signale. In Digitalsystemen sind Komparatoren Schaltnetze, die zwei Binärzahlen miteinander vergleichen. Für zwei Binärzahlen a und b gelten z. B. die Vergleichskriterien $\mathbf{a{=}b}$, $\mathbf{a{<}b}$ und $\mathbf{a{>}b}$.

Komparatorschaltnetz für zwei einstellige Binärzahlen

Aus der Wertetabelle können die Schaltfunktionen direkt angegeben werden (siehe Abbildung 5.11).

Komparatorschaltnetz für zwei zweistellige Binärzahlen

Sollen mehrstellige Binärzahlen auf $=$, $>$ oder $<$ untersucht werden, dann sind verschiedene Schaltungskonfigurationen denkbar. Mithilfe der Stellenwertigkeit der beiden Binärzahlen kann der Vergleich in einer Wertetabelle dargestellt werden. Aus der Wertetabelle wiederum ergeben sich nach Vereinfachung die entsprechenden Schaltfunktionen, die dann als Schaltnetz realisiert werden können.

Beispiel 71. *2-Bit Komparator*

Für zwei zweistellige Binärzahlen $\mathsf{A} = \mathsf{a_1a_0}$ *und* $\mathsf{B} = \mathsf{b_1b_0}$*, wobei* $\mathsf{a_0}$*,* $\mathsf{b_0}$ *den Stellenwert* 2^0 *haben, und* $\mathsf{a_1}$*,* $\mathsf{b_1}$ *den Stellenwert* 2^1*, ergibt sich die Wertetabelle:*

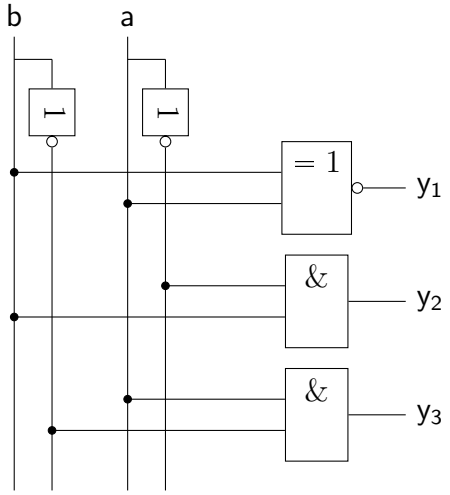
(a) Blockschaltbild

b	a	y_1 a = b	y_2 a>b	y_3 a<b
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

(b) Wertetabelle

$y_1 = a \equiv b$
$y_2 = \overline{a}\ b$
$y_3 = a\ \overline{b}$

(c) Schaltfunktionen



(d) Schaltnetz

Abbildung 5.11: 1-Bit Komparator

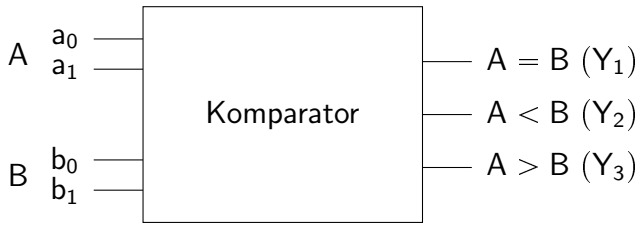


Abbildung 5.12: 2-Bit Komparator

B		A		Y_1	Y_2	Y_3
b_1	b_0	a_1	a_0	A = B	A > B	A < B
0	0	0	0	1	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	1	0	0

Aus der Wertetabelle ergeben sich nach Vereinfachung die Schaltfunktionen in der DNF:

$Y_2 = \overline{a_1}\ b_1 + \overline{a_1}\ \overline{a_0}\ b_0 + \overline{a_0}\ b_1\ b_0$ *für* A < B
$Y_3 = a_1\ \overline{b_1} + a_0\ \overline{b_1}\ \overline{b_0} + a_1\ a_0\ \overline{b_0}$ *für* A > B
$Y_1 = \overline{Y_2}\ \overline{Y_3} = \overline{Y_2 + Y_3}$ *für* A = B

Diese Schaltfunktionen können als Schaltnetz realisiert werden.

Vergleich mehrstelliger Binärzahlen

Es wird ein Algorithmus angewandt, der schrittweise alle Bit-Stellen miteinander vergleicht. Der Vergleich kann mit der MSB-Stelle (werthöchste) oder der LSB-Stelle (wertniedrigste) beginnen. Die Realisierung führt zu **mehrstufigen** Schaltnetzen.

5.6 Arithmetische Schaltnetze (Addierer)

In einem Computer gehören Addierer zur arithmetisch/logischen Einheit (engl.: Arithmetic Logic Unit [ALU]). Addierer sind Schaltnetze, die zwei Dualzahlen miteinander

addieren. Dualzahlen werden wie Dezimalzahlen stellenweise addiert, beginnend bei der niederwertigsten Stelle (least significant bit [LSB]).

5.6.1 Halbaddierer

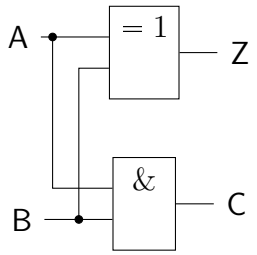
Die Addition zweier einstelliger Dualzahlen A und B ergibt vier Wertekombinationen. Das Ergebnis wird mit **Summe Z** und **Übertrag C** gekennzeichnet. Diese vier Kombinationen können in eine Wertetabelle übertragen werden, aus der man die Schaltfunktionen für Z und C in eine DNF übertragen kann.

A	B	Z	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

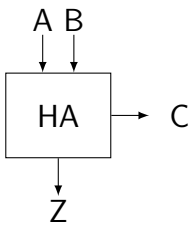
(a) Wertetabelle

$$Z = A\,\overline{B} + \overline{A}\,B = A \oplus B$$
$$C = A\,B$$

(b) Schaltfunktionen



(c) Schaltnetz



(d) Schaltsymbol

Abbildung 5.13: Halbaddierer

Überträgt man diese Funktion in ein Schaltnetz, erhält man die Gatterstruktur für einen einfachen Addierer. Betrachtet man nun den Addierer als ein Gatter mit den zwei einstelligen Eingängen A und B und den Ausgängen Z und C, so erhält man einen **Halbaddierer (HA)**.

5.6.2 Volladdierer

Bei Addition von zwei **mehrstelligen** Dualzahlen (A und B) werden nicht zwei sondern drei Bit addiert, weil der Übertrag (C_i) von der nächst niedrigeren Stelle hinzukommt. Ein Schaltnetz, das drei Bit addieren kann und daraus die Summe Z und den Übertrag C_{i+1} bildet, wird **Volladdierer (VA)** genannt. Solche Additionen können nicht mit einem Halbaddierer durchgeführt werden.

Auch für den Volladdierer gibt es eine Wertetabelle und Schaltfunktion, die zu einem wesentlich komplexeren internen Aufbau des Volladdierers im Vergleich zum Halbaddierer führen.

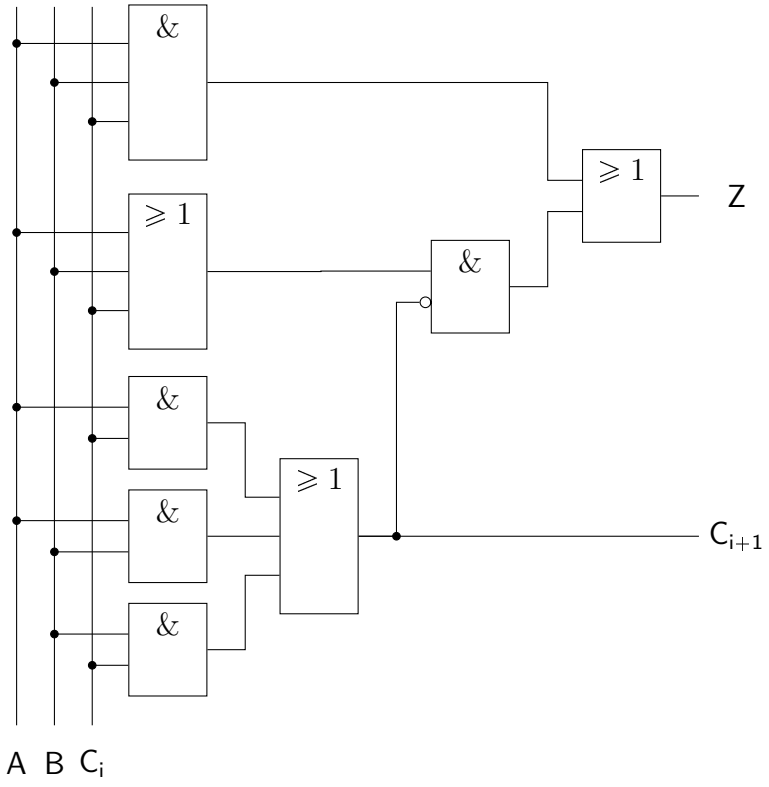
C_i	A	B	Z	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) Wertetabelle

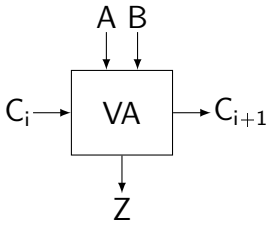
$$Z = A\ \overline{B}\ \overline{C_i} + \overline{A}\ B\ \overline{C_i} + \overline{A}\ \overline{B}\ C_i + A\ B\ C_i$$
$$Z = A \oplus B \oplus C_i$$

$$C_{i+1} = A\ B + A\ C_i + B\ C_i$$
$$C_{i+1} = A\ B + (A + B)C_i$$

(b) Schaltfunktionen



(c) Schaltnetz



(d) Schaltsymbol

Abbildung 5.14: Volladdierer

5.6.3 Mehrstellige Addierer

Addition zweier mehrstelliger Dualzahlen kann bitseriell oder bitparallel erfolgen. Man spricht daher auch von **Serienaddierer** und **Paralleladdierer**. Beide Addiernetze unterscheiden sich wesentlich im Hardwareaufwand und in der Addierzeit. Serienaddierer führen pro Taktschritt nur **eine Stelle** der Addition aus (mit Speicherelementen). Paralleladdierer hingegen führen während eines Taktschrittes die Addition **aller Stellen** aus.

5.7 Arithmetisch/Logische Einheit (ALU)

Die Basisfunktionalität von programmgesteuerten datenverarbeitenden Geräten wird von arithmetisch-logischen Einheiten abgebildet. Sie stellen somit den Kern des datenverarbeitenden Operationswerks dar. Ihre Komplexität ist hoch genug um mehrere elementare Operationen durchführen zu können. Diese lassen sich - wie man dem Namen der Einheit entnehmen kann - in arithmetische und logische Operationen gruppieren. Trotz der Vielzahl an durchführbaren Operationen ist die strukturelle Komplexität (der Aufbau) einer ALU, bedingt durch die Ähnlichkeiten zwischen den einzelnen Operationen, relativ gering.

Der wesentliche Unterschied zwischen den arithmetischen und logischen Operationen besteht darin, dass erstere auf 2-Komplement-Zahlen arbeiten, letztere aber auf einzelnen Bits, wobei diese jedoch zu Bitvektoren zusammengefasst werden können.

Es können alle 16 logischen Operationen (siehe Tabelle 1.1) durchgeführt werden, die mit zwei Variablen möglich sind. Bei den arithmetischen Operationen sind nur elementare realisierbar (Zählen, Komplementieren, Addieren, Subtrahieren). Komplexere Operationen (Multiplizieren, Dividieren, Radizieren) werden durch zusätzliche Programme, Schaltwerke oder Schaltnetze abgebildet.

Herleiten der ALU-Struktur

Bei der Verarbeitung von 2-Komplement-Zahlen ist es wichtig, Strukturen vorzusehen, die den Austausch von Informationen über die einzelnen Bitstellen hinweg ermöglichen. Bei dem Volladdierer ist eine solche Struktur mit dem Übertrags-Bit vorhanden. Mit Hilfe der Volladdiererergleichung und einigen Umformungen lässt sich aus einem Volladdiererglied ein ALU-Glied ableiten.

Die Volladdiererergleichungen:

$$\begin{array}{ll} c_{i+1} = a_i \cdot b_i + (a_i + b_i) \cdot c_i & \text{Übertragungsfunktion} \\ z_i \quad = a_i \equiv b_i \equiv c_i & \text{Summenfunktion} \end{array}$$

lassen sich durch folgende Ersetzungen:

$$\begin{array}{ll} G_i = a_i \cdot b_i & \text{Generate (Übertrag generieren)} \\ P_i = a_i + b_i & \text{Propagate (Übertrag propagieren)} \end{array}$$

umschreiben zu:

$$\begin{aligned} c_{i+1} &= G_i + P_i \cdot c_i \\ z_i &= a_i \equiv b_i \equiv c_i \\ &= (a_i \cdot b_i + \overline{a_i + b_i}) \equiv c_i \\ &= (G_i + \overline{P_i}) \equiv c_i \end{aligned}$$

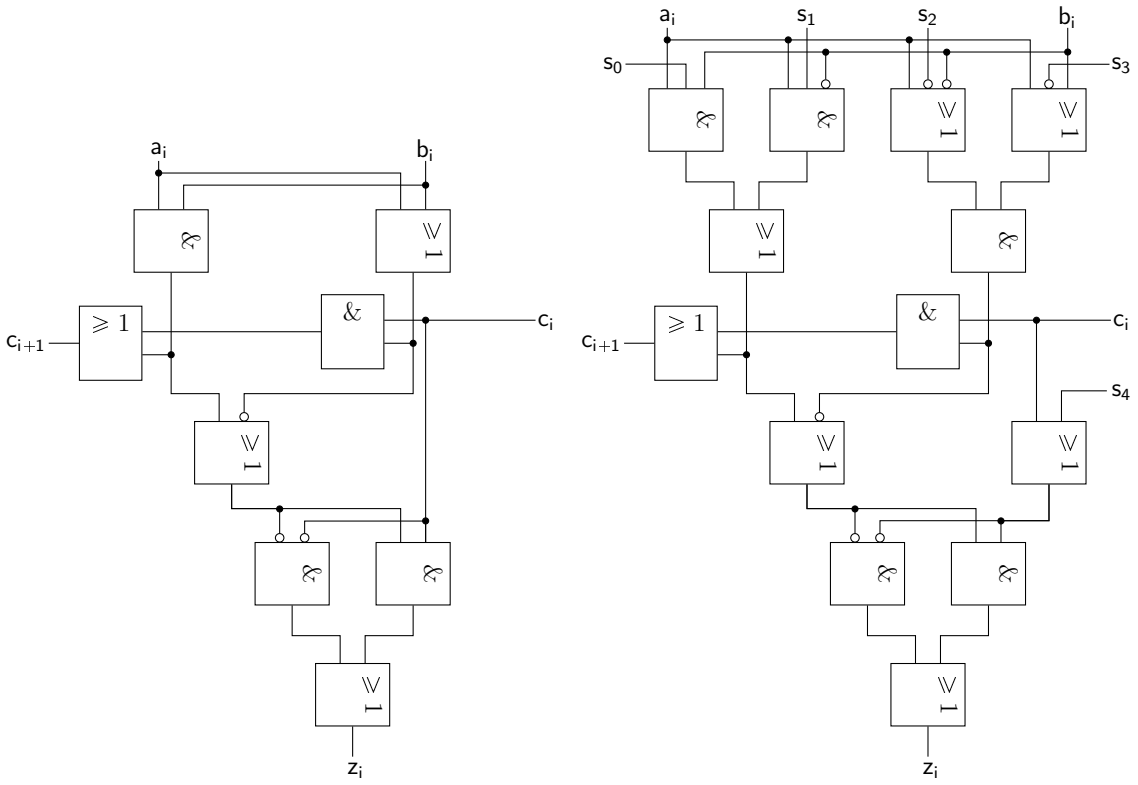
Aus dem einfachen 1-Bit-Volladdierer entsteht eine 1-Bit-ALU, indem die so entstandenen Gleichungen um einen Steuervektor s erweitert wird. Dieser dient dazu die Generate- und Propagate-Funktions-Variable erzeugen zu können sowie die Funktionalität um logische Operationen zu erweitern.

Mit dem Steuervektor $s = [s_0 s_1 s_2 s_3 s_4]$ lauten die Gleichungen des ALU-Glieds:

$$\begin{aligned} G_i &= s_0 \cdot a_i \cdot b_i + s_1 \cdot a_i \cdot \overline{b_i} \\ P_i &= (\overline{s_2} + a_i + \overline{b_i}) \cdot (\overline{s_3} + a_i + b_i) \\ c_{i+1} &= G_i + P_i \cdot c_i \\ z_i &= (G_i + \overline{P_i}) \equiv (s_4 + c_i) \end{aligned}$$

Durch Variation von $s_0 \ldots s_4$ können zusätzlich zur Addition (Steuervektor $s = [10010]$) eine Vielzahl weiterer Operationen gebildet werden. s_4 dient dazu zwischen den beiden Operationsgruppen zu unterscheiden. Mit $s_4 = 0$ ist das Ergebnis z_i abhängig vom Übertrag c_i, weil sich die Gleichung wieder auf $z_i = (G_i + \overline{P_i}) \equiv c_i$ reduzieren lässt. $s_4 = 0$ kennzeichnet somit die arithmetischen Operationen. Demzufolge werden die logischen Operation durch $s_4 = 1$ gekennzeichnet, denn dadurch wird das Ergebnis $(z_i = G_i + \overline{P_i})$ völlig unabhängig vom Übertrag c_i.

Sämtliche ALU-Operationen mit n-stelligen 2-Komplement-Zahlen bzw. Bitvektoren lassen sich der Tabelle 5.1 entnehmen.



(a) VA-Glied (b) ALU-Glied

Abbildung 5.15: Gegenüberstellung eines Volladdierer-Gliedes mit einem ALU-Glied

s_0	s_1	s_2	s_3	arithmetische Operationen	logische Operationen
0	0	0	0	$Z = -1 + c_0$	$Z = 0$
0	0	0	1	$Z = A \vee B + c_0$	$Z = \overline{A \vee B}$
0	0	1	0	$Z = A \vee \overline{B} + c_0$	$Z = \overline{A} \wedge B$
0	0	1	1	$Z = A + c_0$	$Z = \overline{A}$
0	1	0	0	$Z = A \wedge \overline{B} - 1 + c_0$	$Z = A \wedge \overline{B}$
0	1	0	1	$Z = (A \vee B) + (A \wedge \overline{B}) + c_0$	$Z = \overline{B}$
0	1	1	0	$Z = A - B - 1 + c_0$	$Z = A \not\equiv B$
0	1	1	1	$Z = (A \wedge \overline{B}) + A + c_0$	$Z = \overline{A \wedge B}$
1	0	0	0	$Z = (A \vee B) - 1 + c_0$	$Z = A \wedge B$
1	0	0	1	$Z = A + B + c_0$	$Z = A \equiv B$
1	0	1	0	$Z = (A \vee \overline{B}) + (A \wedge \overline{B}) + c_0$	$Z = B$
1	0	1	1	$Z = (A \wedge B) + A + c_0$	$Z = \overline{A} \vee B$
1	1	0	0	$Z = A - 1 + c_0$	$Z = A$
1	1	0	1	$Z = (A \vee B) + A + c_0$	$Z = A \vee \overline{B}$
1	1	1	0	$Z = (A \vee \overline{B}) + A + c_0$	$Z = A \vee B$
1	1	1	1	$Z = A \times 2 + c_0$	$Z = -1$

Tabelle 5.1: Operationen einer ALU. Anmerkung: logische Operationen werden hier mit $\vee$ und $\wedge$ dargestellt, um sie von den arithmetischen (Addition $+$, Multiplikation $\times$) unterscheiden zu können.

5.8 Schaltketten

Schaltketten sind mehrstufige Schaltnetze, die vorwiegend aus identischen bzw. generischen Teilschaltnetzen aufgebaut sind. Sie treten bei der Realisierung rekursiver Funktionen auf. Charakteristisch für solche Funktionen ist, dass auf beiden Seiten der Funktionsgleichung Variablen mit gleichem Namen (nur durch einen Index unterschieden) auftreten. Zum Beispiel:

$$w_{i+1} = f(a_i, b_i, w_i), \text{ für } i = 0, 1, \ldots, n-1$$

Dies gilt als Kurzschreibweise für

$$w_n = f(a_{n-1}, b_{n-1}, f(a_{n-2}, b_{n-2}, \ldots, f(a_0, b_0, w_0))) \text{ mit } w_0 = 1 \text{ oder } w_0 = 0$$

Mit rekursiven Funktionen lassen sich Operationen zwischen Dualzahlen (Addition, Multiplikation, Vergleich) gut beschreiben. Derartige Schaltungen haben Bedeutung in Rechenanlagen. Ein Typisches Beispiel für rekursive Schaltfunktionen ist die Übertragsfunktion bei der Addition. Eine Verknüpfung von Volladdierern zum Zwecke der Addition zweier n-Stelliger Zahlen ergibt dabei eine Schaltkette. Wir vereinbaren dazu, dass Dualziffern ohne Umbenennung als Schaltvariablen behandelt werden.

Beispiel 72. *Additionsschaltkette für zwei Dualzahlen*
Mit Volladdierern als Zelle wird ein Addierer für zwei Dualzahlen $a = a_n a_{n-1} \ldots a_0$, $b = b_n b_{n-1} \ldots b_0$ *als Schaltkette folgendermaßen realisiert:*

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i = F(a_i, b_i, c_i) \qquad \textit{Übertragsfunktion}$$
$$z_i = (a_i \oplus b_i) \oplus c_i = G(a_i, b_i, c_i) \qquad \textit{Summenfunktion}$$

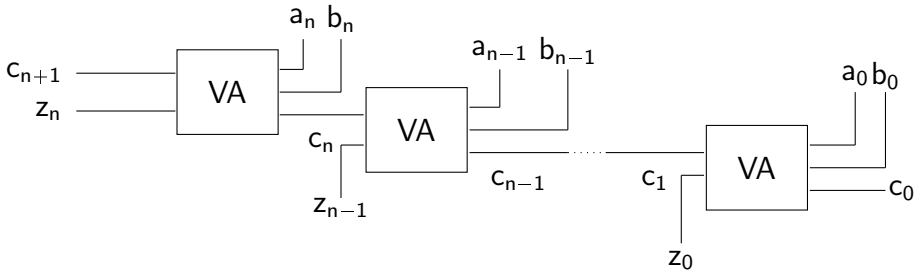


Abbildung 5.16: Additionsschaltkette für zwei Dualzahlen

Die Glieder einer Schaltkette werden auch Kettenglieder genannt. Sie haben stets das gleiche Schaltverhalten und verarbeiten neben Eingangsvariablen a_i, b_i auch Überträge c_i vom vorherigen Kettenglied, die durch die Übergangsfunktion F bestimmt sind. Die Überträge für das 1. Kettenglied ($c_0 = 0$) werden als Anfangswerte bezeichnet. Da Anfangswerte meist Schaltkonstanten sind, ist das 1. Glied der Kette häufig eine einfachere Schaltung. Jedes Kettenglied kann außerdem Ausgangsvariablen haben (hier z_i). Die dazugehörige Schaltfunktion (das zugehörige Funktionsbündel) heißt Ausgangsfunktion (G). Auch die Übergangsfunktion kann ein Funktionsbündel sein.

Beispiel 73. *Eindimensionale Schaltkette*

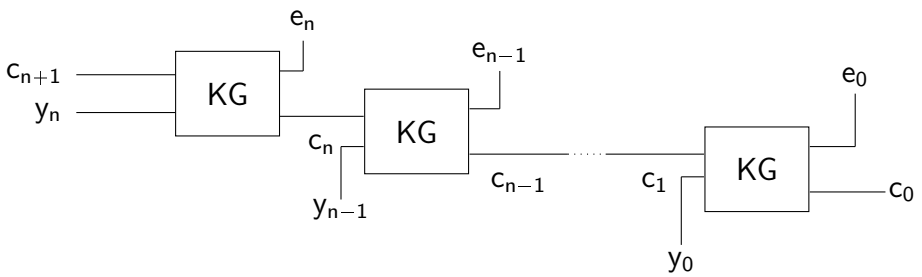


Abbildung 5.17: Verallgemeinerte eindimensionale Schaltkette

Es werden die folgenden n*-Tupel von Schaltvariablen gebildet:*

$$
\begin{aligned}
e_i &= (e_{1i}, e_{2i}, \ldots, e_{ri}) && \textit{Eingangsvektor}\\
c_i &= (c_{1i}, c_{2i}, \ldots, c_{ri}) && \textit{Übergangsvektor}\\
y_i &= (y_{1i}, y_{2i}, \ldots, y_{ri}) && \textit{Ausgangsvektor}\\
c_{i+1} &= F(e_i, c_i) && \textit{Übergangsfunktion}\\
y_i &= G(e_i, c_i) && \textit{Ausgangsfunktion}\\
c_0 & && \textit{Anfangsvektor}
\end{aligned}
$$

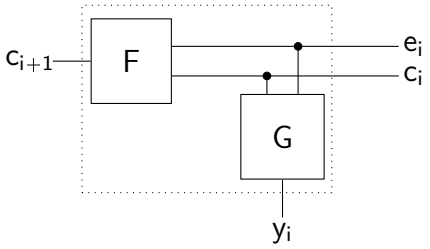


Abbildung 5.18: Kettenglied

Jedes Kettenglied kann selbst ein Schaltnetz sein. Beim Entwurf von Schaltketten ist zunächst eine schaltalgebraische Beschreibung des Problems mit rekursiver Übergangsfunktion und Ausgangsfunktion aufzustellen. Danach ist das Kettenglied als Schaltnetz zu entwerfen. Es ist jedoch nicht jedes schaltalgebraische Problem dazu geeignet, mit Schaltketten realisiert zu werden.

Beispiel 74. *Komparator für zwei Dualzahlen*
Entwurf einer Schaltkette, die zwei $n+1$*-stellige Dualzahlen auf Gleichheit prüft:*

$$
\begin{aligned}
a &= a_n a_{n-1} \ldots a_0\\
b &= b_n b_{n-1} \ldots b_0\\
w &= \begin{cases} 1, & \textit{für } a = b\\ 0, & \textit{für } a \neq b \end{cases}
\end{aligned}
$$

Zur Vereinfachung werden Äquivalenzgatter verwendet. Gleichheit tritt genau dann ein, wenn die Dualziffern an jeder Stelle paarweise gleich sind:

$$w = (a_n \equiv b_n) \cdot (a_{n-1} \equiv b_{n-1}) \cdot \ldots \cdot (a_0 \equiv b_0)$$

Dieser logische Ausdruck kann rekursiv abgearbeitet werden:

$$
\begin{aligned}
w_1 &= a_0 \equiv b_0 = (a_0 \equiv b_0) \cdot w_0 && \textit{mit } w_0 = 1\\
w_2 &= (a_1 \equiv b_1) \cdot w_1\\
w_{n+1} &= (a_n \equiv b_n) \cdot w_n\\
w &= w_{n+1}
\end{aligned}
$$

Die Übergangsfunktion ist:

$$w_{i+1} = F(a_i, b_i, w_i) = (a_i \equiv b_i) \cdot w_i \qquad i = 1, \ldots, n$$
$$w_1 = a_0 \equiv b_0$$

Das erste Kettenglied wurde mit dem Anfangswert $w_0 = 1$ *vereinfacht.*

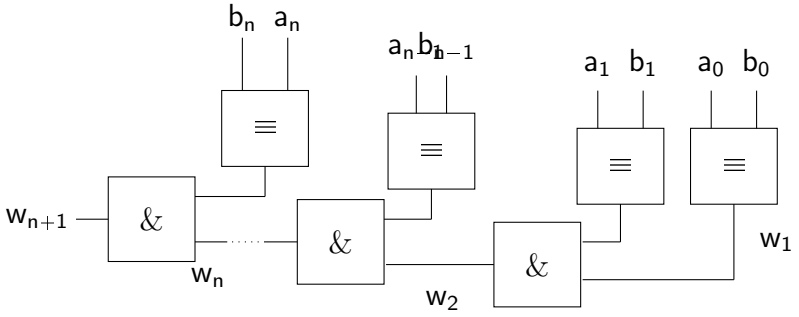


Abbildung 5.19: Komparator für zwei Dualzahlen

Laufzeitproblem

Schaltketten können beträchtliche Längen erreichen. Das bringt lange Signallaufzeiten und eventuell Hazards mit sich. Die Laufzeit ist das Produkt aus der Zahl der Kettenglieder und der Verzögerungszeit eines Gliedes. Die Nachteile sind nur teilweise durch schaltungstechnischen Mehraufwand auszugleichen. Ein 2-Stufiges Schaltnetz garantiert nicht die minimale Signallaufzeit. Es ist effizienter, einige Kettenglieder (meist 2, 4 oder 8) zusammenzufassen und 2-stufig zu realisieren. So werden beispielsweise 2-, 4- und 8-Bit-Volladdierer und 4- und 8-Bit-Vergleicher als Bausteine/Zellen eingesetzt, die sich dann wieder als Kettenglieder in Schaltketten verwenden lassen. Diese Bausteine sind allerdings nicht immer zweistufig realisiert.

Zwei- und mehrdimensionale Schaltketten

Bei zweidimensionalen Schaltketten empfängt jedes Kettenglied von zwei verschiedenen Kettengliedern Übergangssignale und gibt Signale an zwei Kettenglieder ab.

Beispiel 75. *Multiplikationsschaltkette*
Eine Schaltkette zur Multiplikation zweier 4-stelliger Dualzahlen

$$a = a_3a_2a_1a_0 \text{ } \textit{und} \text{ } b = b_3b_2b_1b_0$$

Die Multiplikation von Dualzahlen wird nach dem vom dezimalen Rechnen her geläufigen Algorithmus durchgeführt. Der Multiplikant wird mit jeder Ziffer des Multiplikators multipliziert. Das Endergebnis ergibt sich durch die stellenwertrichtige Aufsummierung der Teilprodukte.

					a_3	a_2	a_1	a_0	$\cdot$	b_0
$+$				a_3	a_2	a_1	a_0		$\cdot$	b_1
$+$			a_3	a_2	a_1	a_0			$\cdot$	b_2
$+$		a_3	a_2	a_1	a_0				$\cdot$	b_3
	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0	$=$	$a \cdot b$

Es gilt:

$$a_i \cdot b_i = \begin{cases} a, & \textit{falls } b_i = 1 \\ 0, & \textit{falls } b_i = 0 \end{cases}$$

Um das Problem rekursiv formulieren zu können, erfolgt die Einführung der Zwischensummen:

$$\begin{aligned} z_i &= z_{i7} z_{i6} \ldots z_{i0} \\ z_0 &= 0000000 \\ z_{i+1} &= z_i + 2^i a b_i \qquad \textit{für } i = 0,1,2,3 \end{aligned}$$

Damit gilt für das Produkt:

$$z_4 = m_7 m_6 m_5 m_4 m_3 m_2 m_1 m_0 = a \cdot b$$

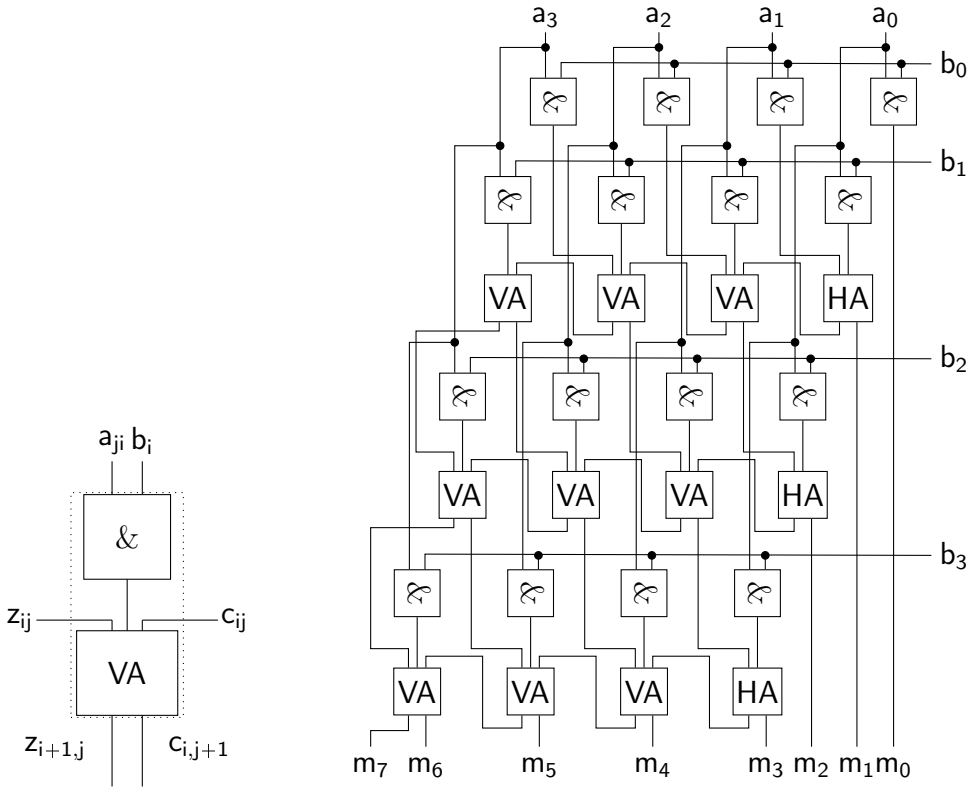
Für die Schaltalgebraische Beschreibung erhält man:

$$\begin{aligned} z_{i+1j} &= (z_{ij} \oplus (a_{j-i} \cdot b_i)) \oplus c_{ij} \\ c_{ij+1} &= z_{ij} \cdot (a_{j-i} \cdot b_i) + z_{ij} \cdot c_{ij} + (a_{j-i} \cdot b_i) \cdot c_{ij} \\ z_{ii+4} &= c_{i-1,i+4} \\ \left.\begin{aligned} c_{ii} &= 0 \\ s_{0J} &= 0 \end{aligned}\right\} & \textit{Anfangswerte für } i = 0,1,2,3 \textit{ und } j = i, i+1, i+2, i+3 \end{aligned}$$

Für die Dualziffern des Produktes $a \cdot b$ *gilt:*

$$\begin{aligned} m_j &= z_{j+1} \qquad && \textit{für } j = 0,1,2 \\ m_j &= z_{4j} && \textit{für } j = 3,4,5,6 \\ m_7 &= c_{37} \end{aligned}$$

Die Beschreibung der Produktziffern wurde so gewählt, um unnötige Kettenglieder einzusparen. Statt 16 Kettengliedern würden sonst 28 benötigt, von denen 12 nur 0 addieren. Ein Kettenglied stellt sich dann wie in Abbildung 5.20a dar. Das in 5.20b dargestellte Schaltnetz wurde durch Ausnutzung der Anfangswerte vereinfacht.



(a) Kettenglied (b) Multiplikationsschaltkette

Abbildung 5.20: Multiplikation zweier Dualzahlen

5.9 Schiebeeinheit

Eine weitere elementare Operation, neben den gezeigten arithmetsich-logischen Operationen, ist die Schiebeoperation (Shift). Auch wenn es unterschiedliche Ausprägungen von Schiebeoperationen gibt, so ist die prinzipielle Funktion immer die gleiche: bei einem Bitvektor wird jedes einzelne Bit von seiner ursprünglichen Stelle in diesem Vektor zu einer anderen verschoben.

Unterschieden werden die Schiebeoperationen zunächst nach der Richtung in welche geschoben wird und nach der Art wie mit den Bits am Anfang und Ende des Bitvektors umgegangen wird. Bits die sich an diesen Stellen befinden, werden durch die Schiebeoperation entweder aus dem Bereich des Vektors hinausgeschoben, oder Stellen müssen sinnvoll aufgefüllt werden. In Abbildung 5.21 sind die unterschiedlichen Varianten aufgeführt. Neben der Schieberichtung und -art kann man Shift-Einheiten auch noch um

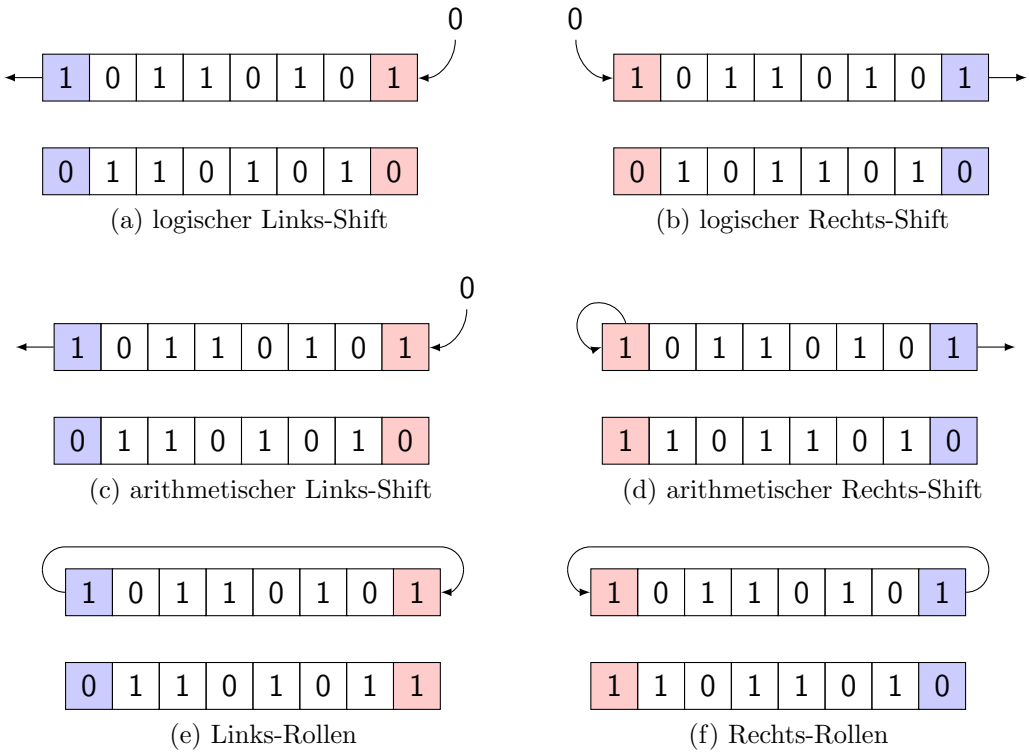


Abbildung 5.21: Die unterschiedlichen Ausprägungen der Schiebeoperation (Shift)

eine variable Shift-Weite ergänzen, die es ermöglicht einen Bitvektor um jede beliebige Stellenanzahl in jede Richtung zu verschieben. Schiebe-Operationen mit diesem Funktionsumfang sind in einer relativ einfachen ALU nicht realisierbar, da man beispielsweise den Übertrag eines Gliedes auch an die niederwertigen Glieder weitergeben können muss, um Schiebeoperationen nach rechts durchführen zu können. Deswegen werden Schiebeoperationen oft in extra Funktionseinheiten realisiert. Der bekannteste Vertreter einer Schiebeeinheit ist der Barrelshifter (siehe Abbildung 5.22).

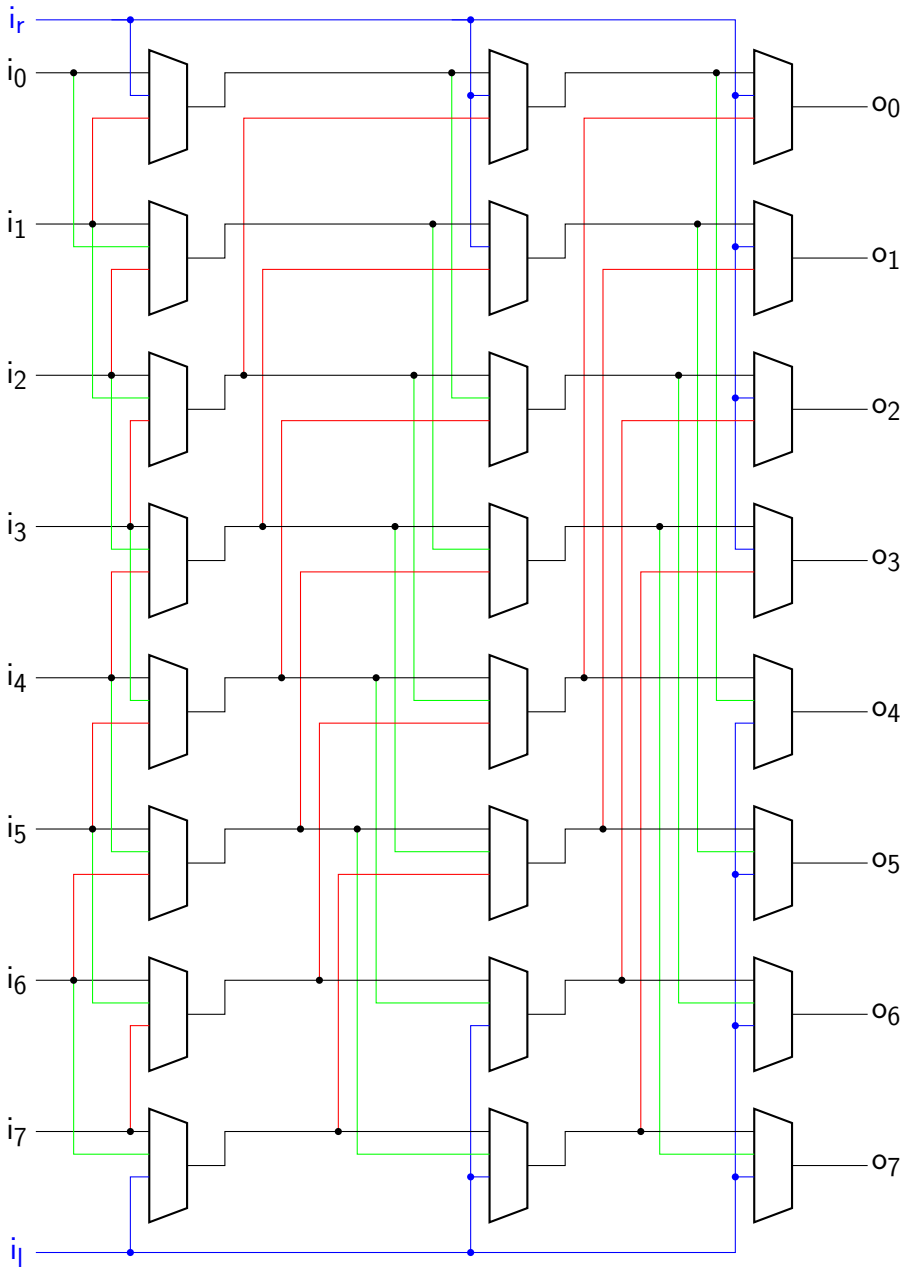


Abbildung 5.22: Multiplexerbasierte Implementierung eines Barrelshifters zur Realisierung von Schiebeoperationen mit variabler Schiebeweite

6 Speicherglieder

6.1 Begriffe und Kenngrößen

6.1.1 Speicherelement

Ein Speicherelement muss **zwei physikalische Zustände** besitzen, die den binären Werten 0 und 1 zugeordnet werden können. Beide Werte müssen von außen verändert werden können (Schreiben bzw. Speichern). Außerdem müssen die Speicherelemente nach bestimmten Kriterien wieder aufgefunden werden können (z.B. durch Adresse oder Informationsinhalt).

6.1.2 Speicherkapazität

Die Speicherkapazität gibt die Anzahl der Speicherelemente oder Zellen im Speicher an. Die gebräuchlichen Einheiten sind **Bit** und **Byte**, wobei 8 Bits 1 Byte entsprechen. Größere Speicherkapazitäten werden mit Größenfaktoren **Ki** (kibi) , **Mi** (mebi) und **Gi** (gibi) bezeichnet. Hierbei gilt, abweichend vom Dezimalsystem (Kilo = 10^3, Mega = 10^6, Giga = 10^9):
Ki = $2^{10} = 1024^1 = 1.024$
Mi = $2^{20} = 1024^2 = 1.048.576$
Gi = $2^{30} = 1024^3 = 1.073.741.824$
Die Abweichung beruht auf der dualen Speicheradressierung.

6.1.3 Speicherorganisation

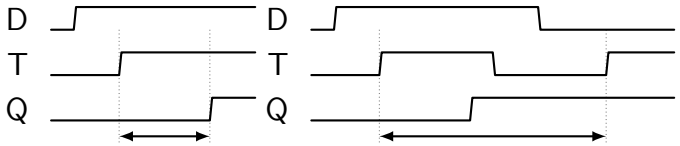
Die Speicherorganisation beschreibt die **Speicherkapazität** und die **Auswahlmöglichkeit**.

6.1.4 Arbeitsgeschwindigkeit

Die Arbeitsgeschwindigkeit kann je nach Speichertyp durch die Größen **Zugriffszeit**, **Zykluszeit** und **Datenrate/Übertragungsgeschwindigkeit** charakterisiert werden.

Zugriffszeit t_z

Die Zugriffszeit ist die Zeitspanne zwischen dem Adressieren einer Speicherzelle bis zum Anstehen der gesuchten Daten am Ausgang.



(a) Zugriffszeit (b) Zykluszeit

Abbildung 6.1: Ein-Bit-Speicherzelle

Zykluszeit t_c

Die Zykluszeit entspricht dem minimalem Zeitabstand mit dem zwei aufeinanderfolgende Daten eingeschrieben bzw. ausgelesen werden können. Im Grenzfall kann die Zykluszeit gleich der Zugriffzeit werden. Im Allgemeinen ist sie aber größer.

Datenrate oder Übertragungsgeschwindigkeit

Die Datenrate entspricht der Geschwindigkeit, mit der Daten ausgelesen und eingeschrieben werden können (meistens **Kehrwert der Zykluszeit**).

6.1.5 Weitere Kenngrößen

Neben den Kosten pro Bit (Bitpreis) wird Speicher auch nach der Speicherdichte klassifiziert, wobei letzteres in Bit pro mm^2 angegeben wird. Zudem gibt es dann noch die Zuverlässigkeit des Speichers. Diese wird durch die Ausfallrate und das Signal-Rauschverhältnis (engl.: Signal-to-Noise-Ratio [SNR]) angegeben.

6.2 Klassifizierung digitaler Speicher

6.2.1 Einteilung nach Arbeitsgeschwindigkeit und Kapazität

Die Hauptparameter **Arbeitsgeschwindigkeit**, **Kapazität** und **Kosten** können nicht gleichzeitig optimiert werden. Hohe Arbeitsgeschwindigkeit bedeutet beispielsweise hohe Kosten und wird meist mit Speichern kleinerer Kapazität realisiert. Es ist daher je nach Anwendungsfall ein Kompromiss zu schließen. Für Rechenanlagen wird ein **hierarchisches Speichersystem** eingesetzt.

6.2.2 Einteilung nach der Art des Zugriffs

Informationen können im Wesentlichen durch zwei Arten wiederaufgefunden werden. Zum einen kann durch Adressierung eines ortsadressierten Speichers der Inhalt erreicht werden. Die andere Möglichkeit ist das Auffinden von Dateninhalten in inhaltsadressierten Speichern (engl.: content adressed memory [CAM]).

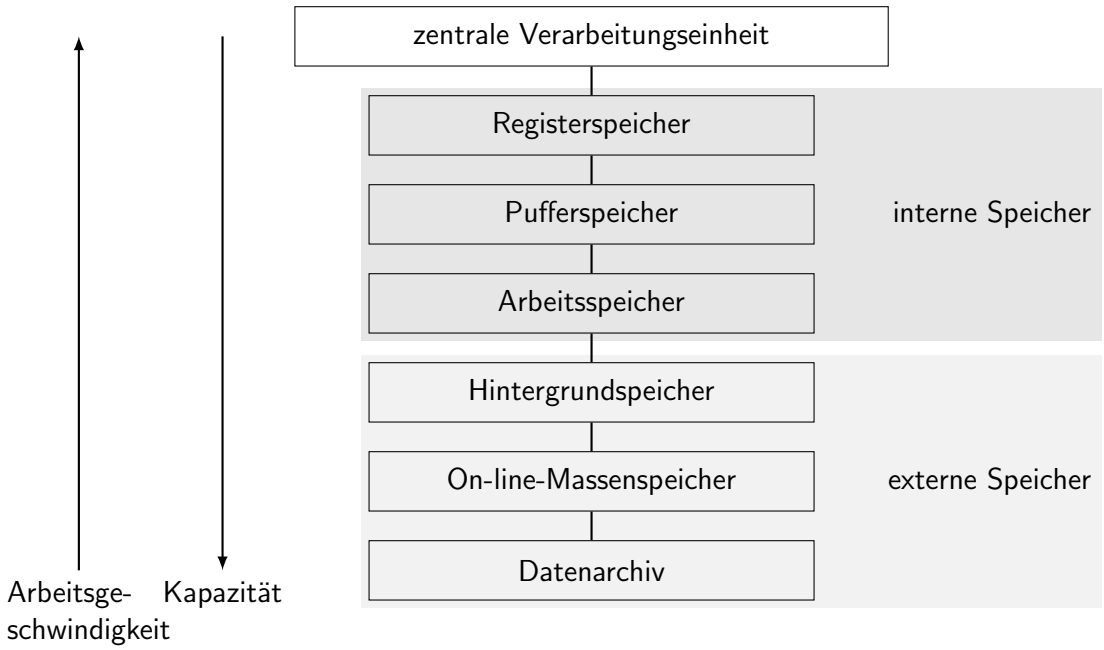


Abbildung 6.2: Aufbau der Speicherhierarchie

Adressierung

Beim Wiederauffinden der Information durch Adressierung wird zwischen **wahlfreiem Zugriff** (random access) und **seriellem Zugriff** (serial access) unterschieden.

Beim Random-Access kann jede Speicherzelle unabhängig von den Anderen adressiert, beschrieben und gelesen werden. Diese Zugriffsart ermöglicht die kürzeste Zugriffszeit. Sie findet beispielsweise im Hauptspeicher von Rechner Verwendung.

Der seriellen Zugriff ist abhängig von einer bestimmter Reihenfolge, so dass der Speicherplatz nicht direkt erreichbar ist. Der Durchlaufspeicher ist ein Beispiel dafür. Die zuerst angekommenen Daten werden auch wieder zuerst ausgegeben (first-in-first-out-Prinzip [FIFO]).

Dateninhalt

Typische Vertreter dieser **Assoziativspeicher** sind beispielsweise Caches und Renaming-Register wie sie in superskaleren Prozessoren verwendet werden aber auch das menschliche Gehirn.

6.2.3 Einteilung nach Art des Datenverkehrs

Schreib-Lese-Speicher

Die Daten können **mehrfach** gespeichert (Schreiben) und wieder abgerufen (Lesen) werden.

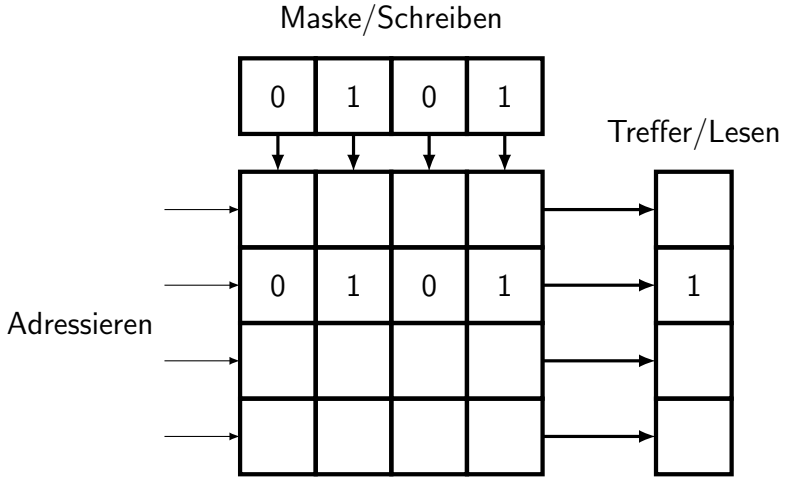


Abbildung 6.3: Zugriff auf inhaltsadressierte Speicher

Lesespeicher/Festwertspeicher

Das Einschreiben erfolgt einmalig, während des Herstellungsprozesses. Diese Speicher nennt man (nur) Lesespeicher (engl.: read only memory [ROM]). Ist diese einmalige Einspeicherung durch den Anwender möglich, so spricht man von programmierbaren Lesespeichern (programmable ROM, PROM, EPROM). Wenn die Einspeicherung ein reversibler Prozess ist, also mehrmals möglich ist,dann werden solche Speicher als EAROM, EEROM (electrical erasable ROM) bezeichnet (z. B. Flash-Speicher).

6.3 Halbleiterspeicher

6.3.1 Einleitung

Überblick

Der Bedarf große Datenmengen zu speichern führte in den letzten 30 Jahren zur Entwicklung zahlreicher Technologien. Bei Halbleiterspeichern konnten große Fortschritte bei der **Kapazitätserhöhung** und der **Kostensenkung pro Bit** erzielt werden. Die Kosten je Bit haben sich seither auf unter ein 1/100 reduziert und liegen 2007 etwa bei 10 Euro je GiBit. Die Kapazität hat 2GiByte je Chip erreicht.

Ein weiterer Grund für den Einsatz von Halbleiterspeichern ist die Kompatibilität zwischen Speicher und Prozessor. Hierbei sind die jeweiligen Betriebsspannungen und Pegel gleich, da die Speicher und Prozessoren in der gleichen Technologie hergestellt sind. Dadurch entfallen teure und störanfällige Interfaceschaltungen. Zusätzlich besteht die Möglichkeit größere Speichersysteme modular aus kleineren Bausteinen aufzubauen.

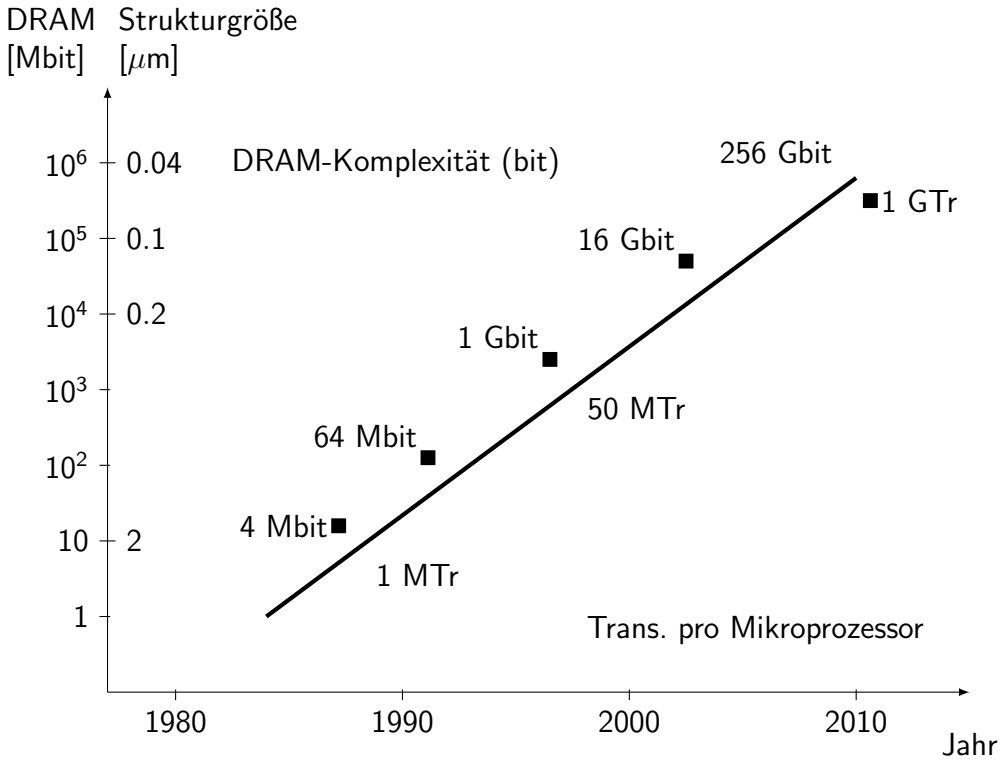


Abbildung 6.4: Überblick der Technologie-Generationen

Realisierung unterschiedlicher Organisationsformen

Die **Schreib-Lese-Speicher** können **statisch** oder **dynamisch** ausgeführt sein. Die statischen Zellen halten die Information solange, wie Versorgungsspannung anliegt. Die dynamischen Zellen erfordern in bestimmten Zeitabständen sogenannte Refresh-Impulse, damit die Daten nicht verloren gehen. **Festwertspeicher** sind **nichtflüchtige Speicher**.

Statische Halbleiterspeicher

Statische Halbleiterspeicher besitzen **kurze Zugriffszeiten**. Im Vergleich zu dynamischen Speichern ist aber für die Realisierung ein **höherer Schaltungsaufwand** notwendig. Anwendung finden die statische Halbleiterspeicher hauptsächlich in Speicherbausteinen kleiner und mittlerer Kapazität. Ihr Einsatz wird vorrangig durch eine kurze Zugriffszeit bestimmt. Realsiert werden diese Speicher in **Bipolar- und MOS-Technik**.

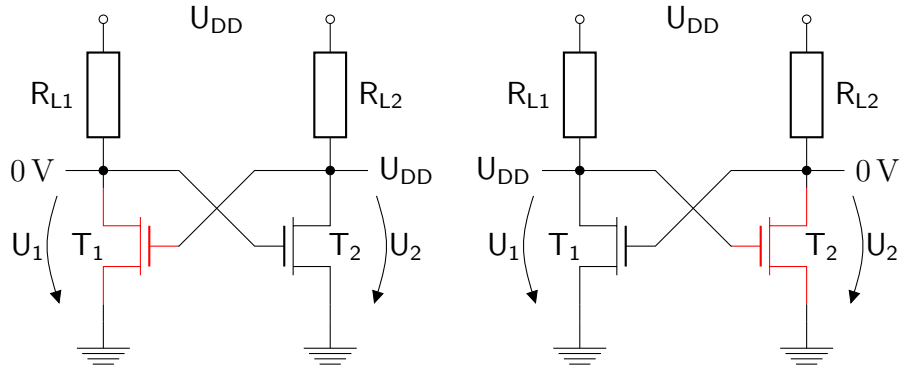
Dynamische Halbleiterspeicher

Dynamische Halbleiterspeicher kommen in komplexen Speichersystemen wegen ihrer hohen Integrationsdichte zum Einsatz. Hergestellt werden diese Speicher nur in **MOS-Technik**.

6.3.2 Bistabile Kippstufe

Funktionsprinzip der statischen Speicherung

Realisiert werden statische Speicher mit zwei Invertern, deren Ausgänge auf die Eingänge der jeweils anderen Stufe zurückgeführt sind. Diese Schaltung mit zwei kreuzgekoppelten Invertern wird als **bistabile Kippstufe** oder als **Flipflop** bezeichnet.



(a) Zustand 1 (b) Zustand 2

Abbildung 6.5: Bistabile Kippstufe

Funktionsweise der Kippstufe (Abbildung 6.5)

Für die Anfangsbetrachtung sei T_1 leitend, U_1 wird auf etwa 0 V gezogen. Diese Spannung liegt gleichzeitig am Gate von T_2 an, der somit sperrt. Wegen des Widerstandes R_{L2} stellt sich die Spannung U_2 auf U_{DD} ein. Da diese Spannung auch am Gate von T_1 anliegt, bleibt T_1 leitend.

6.3.3 Dynamischer MOS-Speicher

Funktion

Die einfachste dynamische Speicherzelle ist das 1-Transistorelement. Über den Transistor T wird eine Kapazität C geladen und die Spannung an ihr wieder abgefragt.

Schreiben

Der Transistor T wird mit einer „1“ auf der Wortleitung leitend und die Kapazität C wird auf den Pegel der Datenleitung aufgeladen.

Lesen

Der Transistor T wird mit einer „1“ auf der Wortleitung leitend und die Ladung der Kapazität C wird abgefragt. War eine „0“ gespeichert, befand sich keine Ladung auf C.

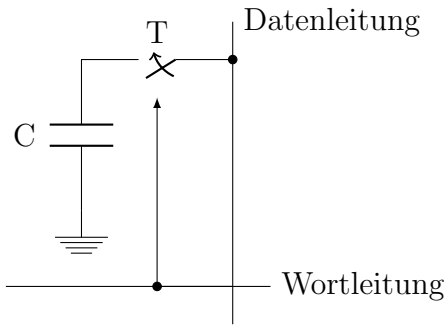
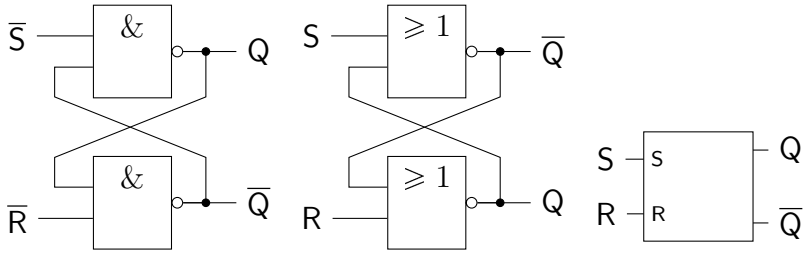


Abbildung 6.6: Dynamische 1-Transistor-Zelle

Da beim Lesen der Ladung von C entnommen wird, muss sie anschließend regeneriert werden.

6.3.4 Ein-Bit-Flipflopspeicher

Asynchrones RS-Flipflop



(a) NAND-Gattern (b) NOR-Gattern (c) Schaltzeichen

Abbildung 6.7: RS-Flipflop realisiert mit unterschiedlichen Gattertypen

Als Speicherglied wird eine bistabilen Kippstufe aus zwei kreuzgekoppelten NAND- oder NOR-Gattern eingesetzt. Jeweils ein Eingang der beiden Gatter wird für die Rückkopplung verwendet, die freien Eingänge dienen zum **Setzen (S)** bzw. **Rücksetzen (R)**. In der Praxis hat dieses RS-Flipflop als Einzelelement zur Speicherung wenig Bedeutung. Der Grund dafür liegt im asynchronen Verhalten, denn die Information wird geladen bzw. gelöscht, sobald am Setz- oder Rücksetzeingang ein Signal anliegt. Ein weiterer Nachteil ist durch die verbotenen Eingangskombinationen (NAND-Gliedern: $\overline{R} = \overline{S} = 0$, NOR-Gliedern $R = S = 1$) gegeben.

Unzulässige Eingangskombination (NOR)

Der Zustand der Ausgänge Q und $\overline{Q}$ ist nicht mehr negiert zueinander. Beide Eingangstransistoren werden für $R = S = 1$ leitend und schalten den „L“-Pegel auf beide Ausgänge.

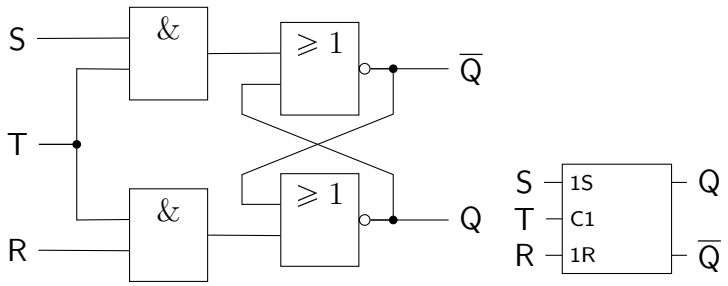
R	S	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	1	0
1	0	0	1
1	1	*	*

Abbildung 6.8: Wahrheitstabelle für NOR-Flipflop

Die mögliche Eingangskombination $R = 1, S = 1$ ergibt eine mehrdeutige Ausgangssituation $Q = 0$, $\overline{Q} = 0$ und ist deshalb **unzulässig**.

Getaktetes RS-Flipflop

Bei synchronen Schaltwerken soll eine bereits anliegende Information erst zu einem bestimmten Zeitpunkt in das Flipflop übernommen werden. Dieses Verhalten wird durch die jeweilige Verknüpfung des R- und S-Einganges mit einem **Steuertakt T** erzielt.



(a) Schaltung (b) Schaltzeichen

Abbildung 6.9: Getaktetes RS-Flipflop

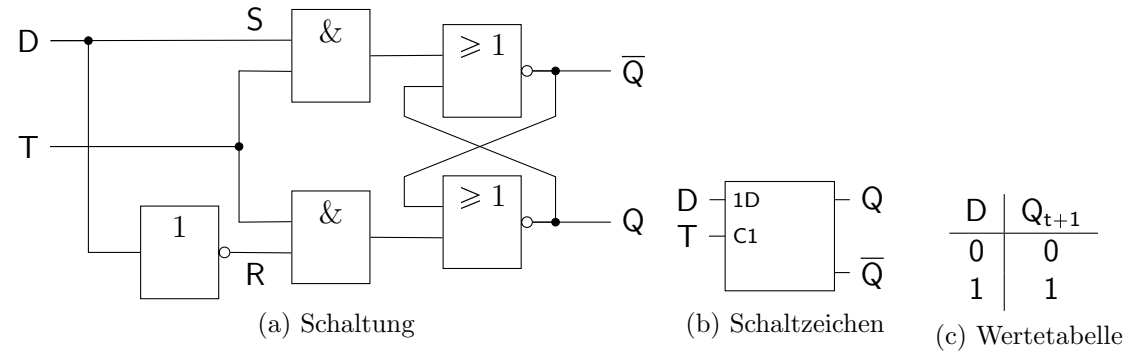
Die Wahrheitstabelle bleibt dieselbe wie für das asynchrone RS-Flipflop. Sie wird lediglich durch die Bedingung erweitert, dass sie nur für den Zeitraum $T = 1$ gilt.

Getaktetes D-Flipflop (Latch)

Der unzulässige Zustand $R = S = 1$ (bei NOR) bzw. $\overline{R} = \overline{S} = 0$ (bei NAND) lässt sich durch eine Beschaltung des Rückstelleingangs mit negiertem Signal des Setzeinganges umgehen. Die Kombination $\overline{R} = \overline{S} = 0$ bzw. $R = S = 1$ wird so prinzipiell vermieden. Dieses Flipflop bezeichnet man als **Delay-Flipflop** oder **Latch**.

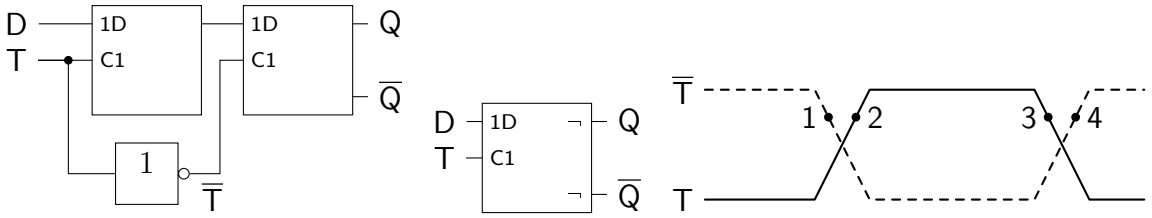
Master-Slave-D-Flipflop

Das einfache D-Latch weist den Nachteil auf, dass während $T = 1$, sich jede Änderung des D-Einganges auf den Ausgang überträgt (**Transparenz**). Dies kann durch Anwendung des **Master-Slave-Prinzips** vermieden werden. Es werden zwei Flipflops kaskadiert, wobei das zweite mit negiertem Takt angesteuert wird. Dadurch wird gewährleistet, dass immer nur eines der beiden Flipflops Daten übernehmen kann.



(a) Schaltung (b) Schaltzeichen (c) Wertetabelle

Abbildung 6.10: Getaktetes D-Flipflop



(a) Schaltung (b) Schaltzeichen (c) Taktimpuls

Abbildung 6.11: Master-Slave-D-Flipflop

Erläuterung der Arbeitsweise des MS-Flipflops am Verlauf der Taktflanken

Punkt	Beschreibung
1	Sperren der Eingänge des Slave-Flipflops
2	Übernahme der Information in das Master-Flipflop
3	Sperren des Dateneingangs des Master-Flipflop
4	Übernahme der Information vom Master-Flipflop in das Slave-Flipflop

Zwischen dem Setzen des Master-Slave-Flipflops und dem Anliegen der Daten an den Ausgängen vergeht eine halbe Taktperiode. Da die Übernahme vom Master zum Slave während einer Taktflanke erfolgt, werden Master-Slave-Flipflops als **taktflankengesteuerte Flipflops** bezeichnet.

6.3.5 Speicherorganisation eines SRAMs

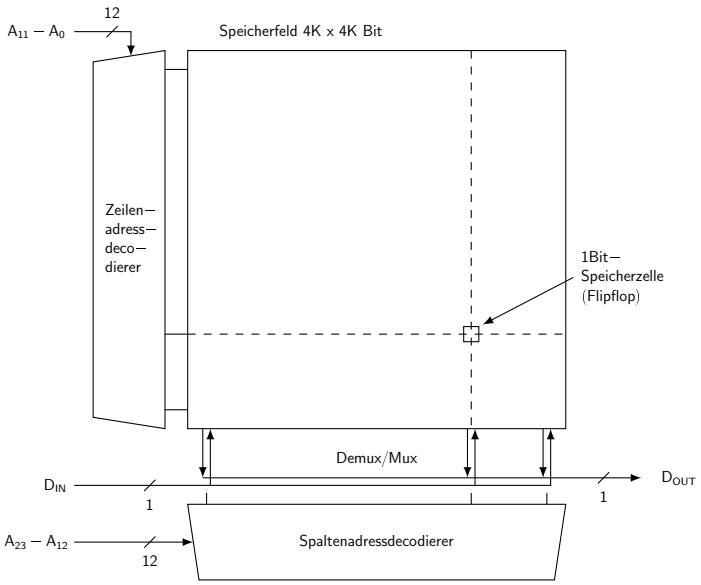


Abbildung 6.12: Speicherorganisation eines SRAMs

7 Programmierbare Logik

7.1 Einleitung

Für die Entwurfsmethodik digitaler Schaltungen gilt es mehrere Dinge zu beachten. Die Wahl der Schaltungsstruktur wird durch den Technologiefortschritt beeinflusst. Wurde früher die Minimierung der Transistoranzahl (Chipfläche) angestrebt, sind heute oft kurze Entwurfszeit, Low-Power oder Geschwindigkeit wichtiger. Der Trend orientiert sich weg von chipflächenoptimierter Schaltung hin zu synthetisierbaren Strukturen, Ausnahmen bilden komplexitäts- und geschwindigkeitsoptimierte Schaltungen.

7.2 Schaltnetze aus Multiplexern

Der Schaltnetzentwurf lässt sich mit Multiplexern (universal logic module ULM) vereinfachen, wenn weder Schaltgeschwindigkeit, Komplexität oder Verlustleistung kritisch sind. Ein Vorteil ist ihre hohe Programmierbarkeit (FPGAs). Mit Multiplexern für n Kontrollvariablen lassen sich alle Funktionen aus diesen Variablen bilden, für Funktionen mit mehr Variablen als die des Multiplexers erfolgt eine Kaskadierung von Multiplexern in zwei oder mehreren Ebenen (Datenselektor).

Der logische Entwurf mit Multiplexern erfolgt mit Hilfe des Erweiterungstheorems von Shannon. Die Funktion wird entsprechend der Multiplexerstruktur erweitert.
z. B. um x_1 und x_2:

$$\begin{aligned} f(x_1, x_2, \ldots, x_n) &= \overline{x_1} \cdot \overline{x_2} \cdot f(0, 0, x_3, \ldots, x_n) + \overline{x_1} \cdot x_2 \cdot f(0, 1, x_3, \ldots, x_n) + \\ &\quad x_1 \cdot \overline{x_2} \cdot f(1, 0, x_3, \ldots, x_n) + x_1 \cdot x_2 \cdot f(1, 1, x_3, \ldots, x_n) \end{aligned}$$

Beispiel 76. *Abbildung einer Funktion auf einen 3-Variablen-Multiplexer:*

$$\begin{aligned} f &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4 + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot x_3 \cdot x_4 + \\ &\quad \overline{x_1} \cdot x_2 \cdot x_3 \cdot \overline{x_4} + x_1 \cdot x_2 \cdot \overline{x_3} \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4} \end{aligned}$$

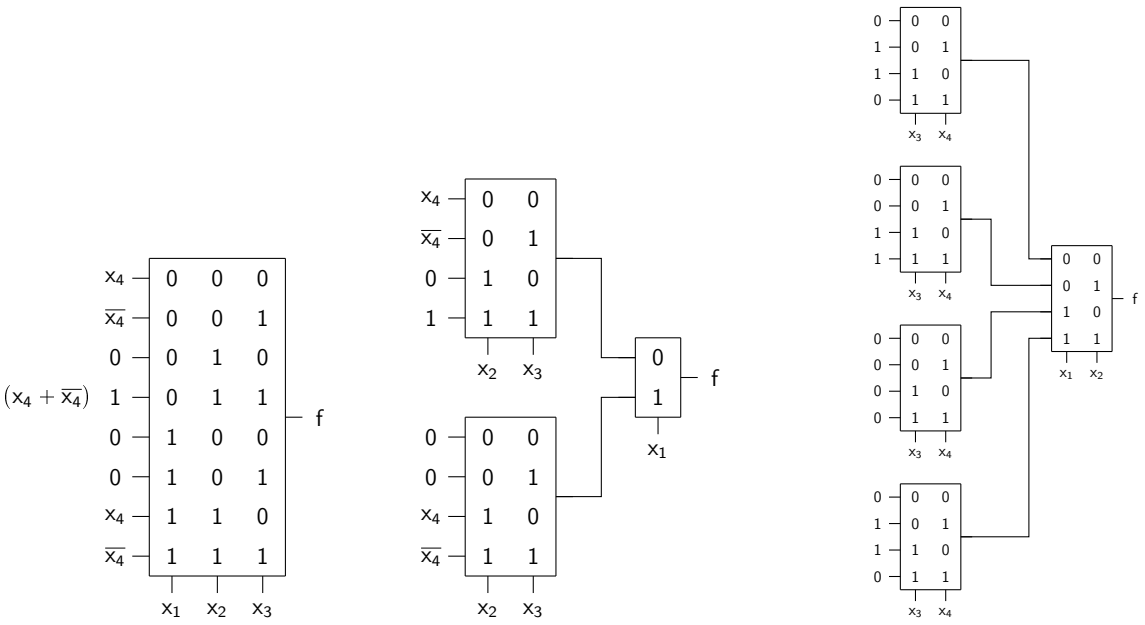
Bei der kaskadierten Erweiterung werden die Teilterme entsprechend erweitert, wobei die Variablen die Kontrollfunktion für die nachfolgenden Ebenen übernehmen.

Beispiel 77. *Implementierung mit 1-Variablen und 2-Variablen-Multiplexer:*

$$\begin{aligned} f &= \overline{x_1} \cdot [\overline{x_2} \cdot \overline{x_3} \cdot (x_4) + \overline{x_2} \cdot x_3 \cdot (\overline{x_4}) + x_2 \cdot \overline{x_3} \cdot (0) + x_2 \cdot x_3 \cdot (1)] + \\ & \quad x_1 \cdot [\overline{x_2} \cdot \overline{x_3} \cdot (0) + \overline{x_2} \cdot x_3 \cdot (0) + x_2 \cdot \overline{x_3} \cdot (x_4) + x_2 \cdot x_3 \cdot (\overline{x_4})] \end{aligned}$$

Beispiel 78. *Eine weitere Implementierungsvariante ist, alle Variablen mit Kontrollfunktion zu versehen (z. B. FPGA):*

$$\begin{aligned} f &= \overline{x_1} \cdot \overline{x_2} \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4(1) + x_3 \cdot \overline{x_4} \cdot (1) + x_3 \cdot x_4 \cdot (0)] + \\ & \quad \overline{x_1} \cdot x_2 \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (0) + x_3 \cdot \overline{x_4} \cdot (1) + x_3 \cdot x_4 \cdot (1)] + \\ & \quad \overline{x_1} \cdot \overline{x_2} \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (0) + x_3 \cdot \overline{x_4} \cdot (0) + x_3 \cdot x_4 \cdot (0)] + \\ & \quad x_1 \cdot x_2 \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (1) + x_3 \cdot \overline{x_4} \cdot (1) + x_3 \cdot x_4(0)] \end{aligned}$$



(a) 3-Variablen-Multiplexer

(b) 1- und 2-Variablen-Multiplexer

(c) 2-Variablen-Multiplexer mit voll expandierter Funktion

Abbildung 7.1: Implementierung der Funktion unter Verwendung verschiedener Multiplexervarianten

Implementierung von Funktionsbündeln

Stehen passende Multiplexer zur Verfügung, kann eine Minimierung entfallen, da die Funktionen vollständig expandiert zur Verfügung stehen.

Beispiel 79. *Funktionsbündel mit Multiplexern:*

$$\begin{aligned}
f_1 &= \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot\overline{x_4} + \overline{x_1}\cdot x_2\cdot\overline{x_3}\cdot\overline{x_4} + x_1\cdot\overline{x_2}\cdot x_3\overline{x_4} + \\
&\quad x_1\cdot\overline{x_2}\cdot x_3\cdot x_4 + x_1\cdot x_2\cdot\overline{x_3}\cdot\overline{x_4} + x_1\cdot x_2\cdot\overline{x_3}\cdot x_4 \\
f_2 &= \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot\overline{x_4} + \overline{x_1}\cdot x_2\cdot\overline{x_3}\cdot x_4 + x_1\cdot\overline{x_2}\cdot x_3\cdot\overline{x_4} + \\
&\quad x_1\cdot\overline{x_2}\cdot x_3\cdot x_4 + x_1\cdot x_2\cdot\overline{x_3}\cdot x_4 \\
f_3 &= \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\cdot x_4 + \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot\overline{x_4} + \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot x_4 + \\
&\quad x_1\cdot\overline{x_2}\cdot x_3\cdot\overline{x_4} + x_1\cdot\overline{x_2}\cdot x_3\cdot x_4 + x_1\cdot x_2\cdot\overline{x_3}\cdot\overline{x_4}
\end{aligned}$$

x_1, x_2 *und* x_3 *werden als Kontrollvariable eingesetzt:*

$$\begin{aligned}
f_1 &= \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\cdot(0) + \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot(\overline{x_4}) + \overline{x_1}\cdot x_2\cdot\overline{x_3}\cdot(\overline{x_4}) + \overline{x_1}\cdot x_2\cdot x_3\cdot(0) + \\
&\quad x_1\cdot\overline{x_2}\cdot\overline{x_3}\cdot(0) + x_1\cdot\overline{x_2}\cdot x_3\cdot(1) + x_1\cdot x_2\cdot\overline{x_3}\cdot(1) + x_1\cdot x_2\cdot x_3\cdot(0) \\
f_2 &= \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\cdot(0) + \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot(0) + \overline{x_1}\cdot x_2\cdot\overline{x_3}\cdot(1) + \overline{x_1}\cdot x_2\cdot x_3\cdot(0) + \\
&\quad x_1\cdot\overline{x_2}\cdot\overline{x_3}\cdot(0) + x_1\cdot\overline{x_2}\cdot x_3\cdot(1) + x_1\cdot x_2\cdot\overline{x_3}\cdot(x_4) + x_1\cdot x_2\cdot x_3\cdot(0) \\
f_3 &= \overline{x_1}\cdot\overline{x_2}\cdot\overline{x_3}\cdot(x_4) + \overline{x_1}\cdot\overline{x_2}\cdot x_3\cdot(1) + \overline{x_1}\cdot x_2\cdot\overline{x_3}\cdot(0) + \overline{x_1}\cdot x_2\cdot x_3\cdot(0) + \\
&\quad x_1\cdot\overline{x_2}\cdot\overline{x_3}\cdot(0) + x_1\cdot\overline{x_2}\cdot x_3\cdot(1) + x_1\cdot x_2\cdot\overline{x_3}\cdot(\overline{x_4}) + x_1\cdot x_2\cdot x_3\cdot(0)
\end{aligned}$$

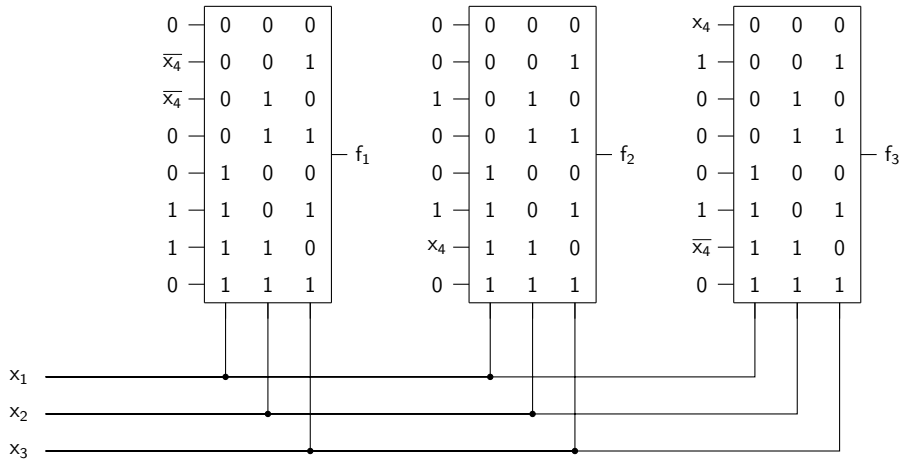


Abbildung 7.2: Implementierung des Funktionsbündels

7.3 Programmierbare Logik

7.3.1 Einleitung

Programmierbare Bausteine haben eine feste Struktur und können in ihrer Funktion speziell für eine Anwendung programmiert werden. Die programmierbare Logik nimmt bei

den Entwurfsstilen eine mittlere Stellung bezüglich Integrationsgrad und Entwicklungsdauer ein, wobei die Entwicklungskosten zum großen Teil von der Entwicklungsdauer beeinflusst werden. Die programmierbare Logik ist insbesondere wegen ihrer flexiblen Änderbarkeit attraktiv, ein Problem ist jedoch, dass die Programmierinfrastruktur die Integrationsdichte, Geschwindigkeit und Verlustleistung beeinträchtigt.

Programmierbare Schaltungen werden nach ihrer Struktur klassifiziert. Es gibt u.a.:

- Festwertspeicher (Read-Only Memories, ROMs): PROM, EPROM
- Schreib-/Lesespeicher: EPROM, EEPROM, EAPROM
- Programmierbare Logische Anordnungen (Programmable Logic Arrays, PLAs): PLA, PLD, GAL, PLM, FPLA, EPLD, Mehrfach-Array PLD
- strukturprogrammierbare Bausteine: FPGAs, PLSs (XILINX, ACTEL)

Programmierbare Array-Bausteine haben die Eigenschaft, dass die Komplexität wesentlich durch die Zahl der Anschlüsse und die Laufzeit begrenzt wird und dass der Logikentwurf der Synthese auf Gatterebene entspricht. Es gibt zwei verschiedene Möglichkeiten diese Bausteine zu programmieren. Bei irreversiblen Speichern (Festwertspeicher) werden Maskenprogrammierung und elektrische Programmierung eingesetzt, wobei die Maskenprogrammierung beim Halbleiterhersteller durchgeführt wird und nur bei großen Stückzahlen (> 1M) kostengünstig ist. Der Vorteil ist die hohe Schaltungskomplexität. Trotzdem werden maskenprogrammierte Bausteine heute weitgehend durch EPROM (OTP) ersetzt. Heutzutage ist jedoch die elektrische Programmierung eher vorzufinden.

Bei reversiblen Speichern (Schreib-/Lesespeicher) wird elektrisch geschrieben und entweder elektrisch oder durch UV-Licht gelöscht. Die elektrische Programmierung erfolgt beim Anwender im Labor und ist flexibel für kleinste Stückzahlen geeignet, die Nachteile sind der höhere Preis und die geringere Schaltungskomplexität und -geschwindigkeit.

7.3.2 ROM-Logik

Verschiedene Funktionsstrukturen haben verschiedene Einsatzbereiche, die Konjunktion ($f(x) = x_1 x_2 \overline{x_3} ... x_n$) z.B. dient zur Realisierung von Mikroprogrammsteuerwerken, zur Programmierung einer Schaltnetzfunktion und zur Darstellung von Wertetabellen und mathematischen Funktionen.

Die Spalten einer ROM-Matrix stellen NOR-Gatter mit N Eingängen dar, die aus den Eingangsvariablen A, B und C dekodiert werden (1 aus n Code). Die Ausgangsfunktionen werden wegen der quadratischen Anordnung ebenfalls dekodiert. Eine ROM-Matrix mit N-Bit Adresse und M-Bit Ausgangscode benötigt demnach M * N Speicherbits.

Ein Nachteil der ROM-Logik ist der Umstand, dass für alle N Zeilen und alle M Spalten Transistorplätze vorgesehen werden müssen, auch wenn sie zum Teil nicht benötigt werden. Meistens werden vom Schaltungsproblem auch nicht alle N Adressplätze (Zeilen) belegt, aufgrund der Redundanz ist die ROM-Logik also für die Implementierung einfacher Logikfunktionen ineffizient. Ein Vorteil liegt in der regelmäßigen Struktur der

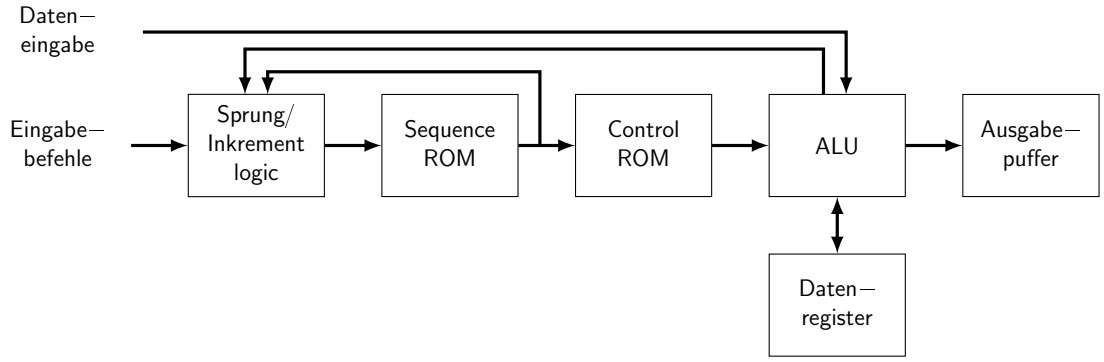


Abbildung 7.3: Blockschaltbild eines μP mit zwei ROMs: für Folgeadresse (Sequence ROM) und Steuerwort (Control ROM)

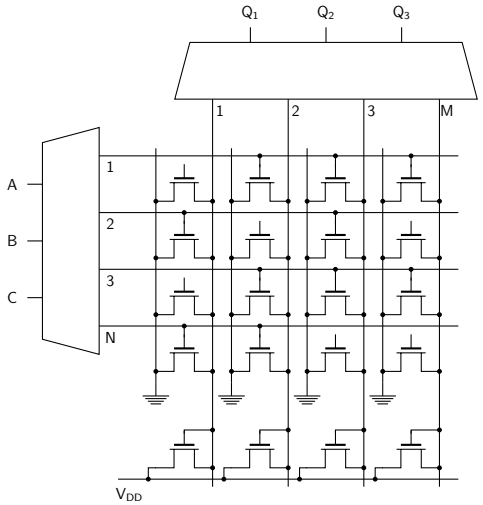


Abbildung 7.4: ROM-Matrix in MOS-Technik mit N Eingängen und M Ausgängen

ROM-Matrix, diese ermöglicht ein kompaktes Layout. Der Speicherinhalt kann leicht durch 1 bis 2 neue Masken programmiert werden, ohne dass eine Änderung der ROM-Architektur nötig wird.

Beispiel 80. $f_1 = x\overline{yz} + xyz$; $f_2 = \bar{x}z$; $f_3 = x\overline{yz} + \bar{x}z$

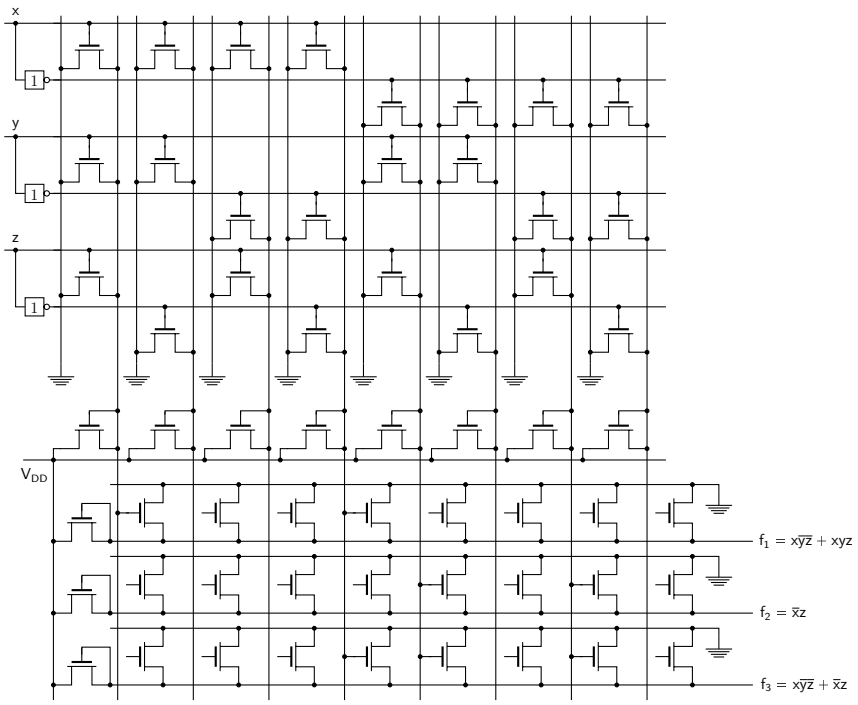
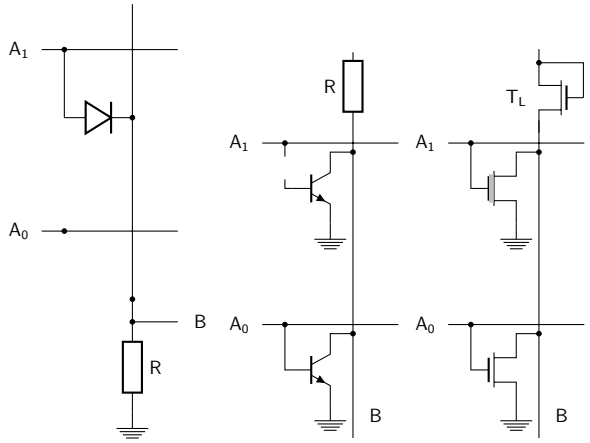


Abbildung 7.5: Beispiel einer ROM-Logik

7.3.3 Speicherzellen

Speicherzellen **irreversibler Festwertspeicher** sind aufgebaut aus Speicherelementen, die einer Kopplungsschaltung zwischen Wort- und Bitleitungen entsprechen, diese Koppelelemente bestehen entweder aus Dioden, Bipolar- oder MOSFET-Transistoren.



(a) Diode (b) Bipolar (c) MOS

Abbildung 7.6: ROM-Matrizen basierend auf unterschiedlichen Technologien

Bei maskenprogrammierbare Speicherelemente sind die Speicherzellen also z.B. Dioden- oder Bipolartransistormatrizen. Das Datenmuster dieser Matix wird durch die Verdrahtung festgelegt, dies geschieht am Ende des Herstellungsprozesses, bei dem mit einer sog. Verdrahtungsmaske die Verbindung festgelegt wird. Bei einer Transistormatrix in MOS-Technik wird durch Maskenprogrammierung die Dicke der Gateisolierschicht festgelegt, wobei eine dicke Isolierschicht ein Schalten des Transistors trotz angelegter Spannung verhindert.

Elektrisch programmierbare Festwertspeicher (PROM) werden durch das Durchschmelzen (Zerstören) eines Widerstandes (fusable link) oder einer Diode programmiert. Ein spezieller Widerstand wird zur Basis des Bipolartransistors oder zur Diode in Reihe geschaltet. An der entsprechenden Speicherzelle wird an diesen Widerstand mit Hilfe eines Programmiergerätes eine hohe Spannung (ca. 10 V) angelegt, die den Widerstand bzw. die Diode wie ein Sicherungselement durchschmilzt.

Reversible Festwertspeicher besitzen die Eigenschaften, dass ihr Urzustand wiederhergestellt werden kann, das Löschen ist je nach Speichertechnologie auf zwei Arten möglich. Einerseits durch UV-Licht-Bestrahlung des Speicherchips, dabei wird die gesamte Speichermatrix auf einmal gelöscht, andererseits durch elektrische Impulse, durch die jede Zelle einzeln gelöscht werden kann.

Erasable PROM (EPROM) folgen dem 1-Transistor-FAMOS-Speicherelement in n-Kanal-Technik Speicherprinzip, bei dem (FAMOS = Floating Gate Avalanche Injection MOS) das Gate des Speichertransistors nach außen völlig isoliert in seinem Potential „schwebt“. Der Speichereffekt beruht auf einer Verschiebung der Schwellenspannung eines MOS-Transistors, d.h zum Programmieren wird der Source-Gate-Übergang des Spei-

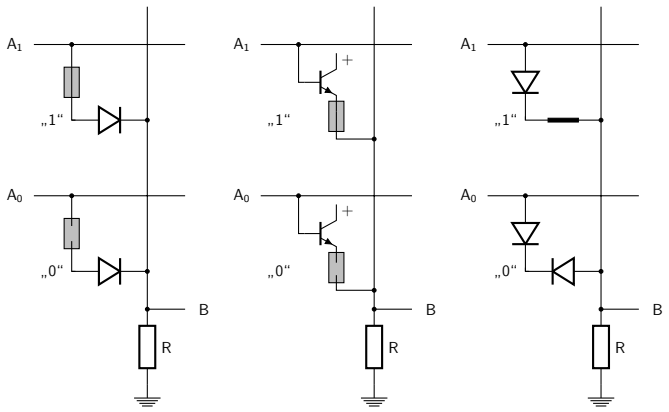


Abbildung 7.7: Elektrisch programmierbare Speicherelemente für Lesespeicher mit Schmelzsicherung

chertransistors mit hoher Spannung bis zum Avalanche-Durchbruch belastet. Der Speichertransistor arbeitet im programmierten und unprogrammierten Zustand als selbstsperrender Typ, die Löschung erfolgt mit UV-Licht. Die Bausteinkapazitäten liegt z.Z. bei 16 MBit.

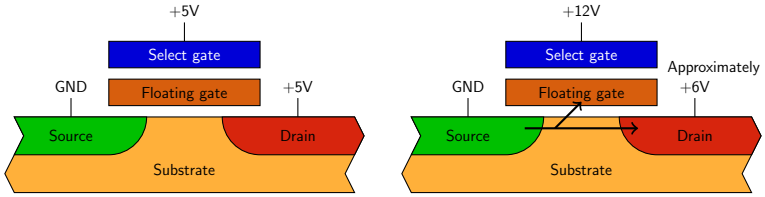


Abbildung 7.8: 1-Transistor-FAMOS-Speicherlement

Zur Neuprogrammierung im System wurden elektrisch löschbare Speicher (EAROM = Electrically Alterable ROM bzw. EEROM = Electrically Erasable ROM) entwickelt, bei denen der Speichereffekt durch Speicherung von Ladung in einem schwebenden Gate (FAMOS) erreicht wird. Die Nachteile dieser Speicher sind hohe Programmierspannungen und längere Zugriffzeiten.

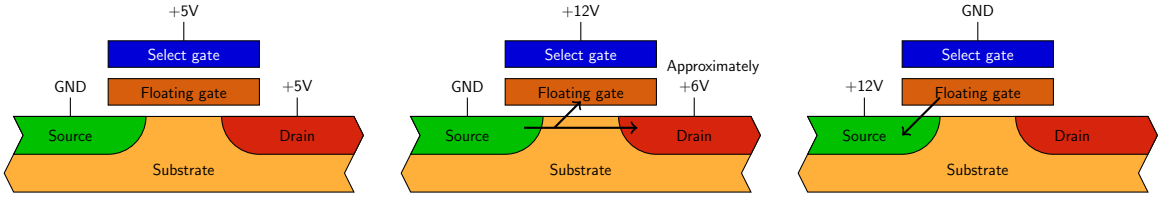


Abbildung 7.9: Programmierung einer FAMOS-EAPROM-Zelle

Die Vorteile der ROM-Logik liegen in leistungsfähigen Schaltnetzen mit mehrfachen Ein- und Ausgängen (Typische Anwendungen sind z.B. Codierer, Arithmetikschaltungen und ähnliches), der hohen Komplexität der Standardbausteine, und dass kein direkter Entwicklungsaufwand zur Minimierung der Logikfunktionen auf Gatterebene nötig

ist. Die Nachteile der ROM-Logik sind u.a. die langsamere Arbeitsgeschwindigkeit gegenüber irregulärer Logik (Kundenschaltung), diese Verzögerung wird verursacht durch eine zusätzliche Adressdecodierung und Matrixkapazitäten. Hinzu kommt eine recht schlechte Ausnutzung der Chipfläche, wegen unbenutzter Speicherbereiche und das Fehlen zusätzlicher Schaltungsmöglichkeiten, wie z. B. Negation oder Selektion von einzelnen Bits. Aufgrund dieser Nachteile ist die ROM-Logik weniger für kleine Schaltnetze geeignet, dafür aber für große kombinatorische Netzwerke mit regelmäßigen Strukturen (z.B. μP-Programme, Funktionstabellen).

7.3.4 Anwendungsbeispiele für ROM-Logik

Die algorithmische Berechnung arithmetischer Funktionen geschieht durch softwaremäßige Berechnung über Reihenentwicklungen, wenn keine Geschwindigkeit erforderlich ist:
$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - .....$
$S_{k+1} = \frac{1}{2}(S_k + \frac{N}{S_k})$ mit $\lim_{k\to\infty} S_k = \sqrt{N}$ (Newton-Raphson Verfahren)

Die algorithmische Berechnung kann für technisch-wissenschaftliche Bereiche zu langsam sein, dann erfolgt die tabellarische Berechnung auf Hardwareebene. Die einfachste Realisierung ist die Verwendung eines ROM als Tabellenspeicher, Argument x als Adresse und Speicherinhalt als Ergebnis des Funktionswertes von x Hierbei besteht aber der Nachteil, dass ein enormer Speicherbedarf bei gebräuchlichen Wortlängen benötigt wird, daher gibt es häufig Mischlösungen mit zusätzlicher Logik.

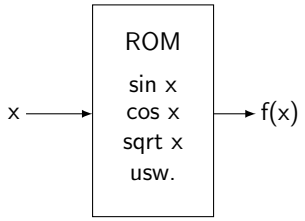


Abbildung 7.10: Struktur eines ROM-Schaltnetzes

7.4 Programmable Logic Array (PLA)

7.4.1 Überblick

Zur Klassifizierung programmierbarer Logikbausteine wird zwischen Array-Logikbausteinen wie PLA, PAL, PLD, GAL, PLM, FPLA, EPLD, Mehrfach-Array PLD und Strukturkonfigurierbaren Gate-Arrays wie XILINX, ACTEL unterschieden. Die physikalische Realisierung (Komplexität) programmierbarer Bausteine wird im wesentlichen durch die Zahl der Anschlüsse (Variablen) begrenzt. Ein Problem sind die mit wachsender Komplexität zunehmenden Leitungslaufzeiten. Der Logikentwurf wird von der Bausteinarchitektur geprägt, dies erfordert in der Regel eine firmenabhängige Entwurfssoftware.

7.4.2 Schaltungsprinzip

Vergleich von ROM und PLA: ROM erfordert für n Adresseingänge einen Decodierer mit 2n Ausgängen, ein PLA (programmable logic array) wird dagegen direkt adressiert und besteht aus einer UND- Matrix mit einer kaskadierten ODER-Matrix. In der UND-Matrix werden die Eingangsvariablen in konjunktiver Form programmiert, in der ODER-Matrix erfolgt die Verknüpfung dieser Produkte. Diese 2-Ebenen-Programmierung erlaubt eine Informationsverdichtung mit Hardware-Einsparung gegenüber der ROM-Logik.

Funktionsstruktur DNF: $f(x) = x_1 \cdot x_2 \cdot \overline{x_3} + x_4 \cdot x_5 ... x_n + x_6 \cdot \overline{x_7} + \ldots$

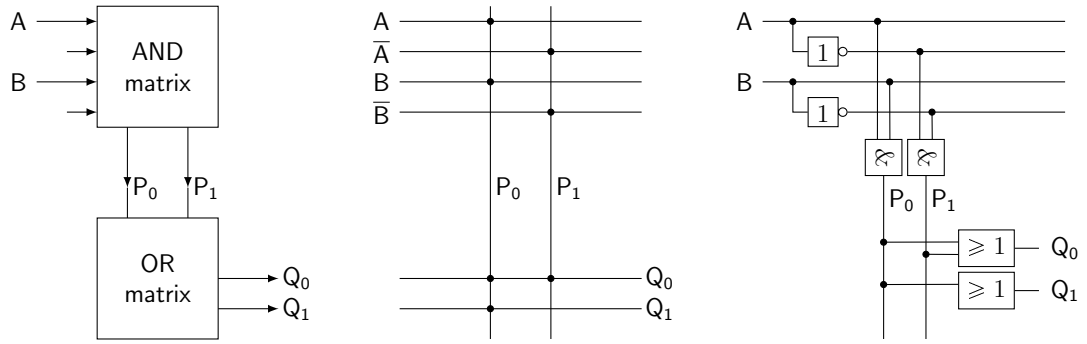


Abbildung 7.11: PLA: Blockdiagramm, Logikdiagramm, Logikschema

Zur PLA-Realisierung werden für beide Matrizen NAND- oder NOR-Gatter gewählt, die Umwandlung in NAND-Funktionen ist direkt möglich. Eine NOR-Realisierung erfordert entweder negierte Ausgangsfunktionen oder die Programmierung der negierten Werte, NOR-Matrizen werden in MOS-Technik wegen der Parallelschaltung bevorzugt.

Ein Vergleich mit Beispiel in der ROM-Logik zeigt eine erhebliche Aufwandsersparnis.

Es sind nur 27 gegenüber 72 Matrixpunkten erforderlich.

Die höhere Informationsdichte des PLA gegenüber dem ROM ist aber auch nachteilig. Im PLA ist die Anzahl der Zeilen und Spalten auf das Schaltungsproblem abgestimmt, so dass eine Logikänderung gleichzeitig eine Strukturveränderung des PLAs zur Folge haben. Da die PLA-Matrix kleiner ist als die ROM-Matrix, sind die Verzögerungszeiten kürzer.

Die Vorteile für PLAs gegenüber der Realisierung mit Standard-Gattern sind, dass keine zeitaufwendige Minimierung der Logik mehr nötig ist, aber eine Anpassung möglich ist, es bedarf auch keiner speziell entworfenen Logikgatter, außerdem ist die PLA-Struktur aufgrund ihrer einfacheren und regelmäßigeren Struktur leichter skalierbar.

Die Nachteile liegen in ihrer geringeren Flexibilität in der Programmierung gegenüber ROM, PLAs können keine komplexen Funktionen speichern (insbesondere keine mit vielen Produkttermen), ihre Struktur ist nicht geeignet für vorstrukturierte Hardware (Gate Arrays, Standardzellen, FPGAs) und eine Matrix verursacht längere Laufzeiten gegenüber optimierten Schaltnetzen.

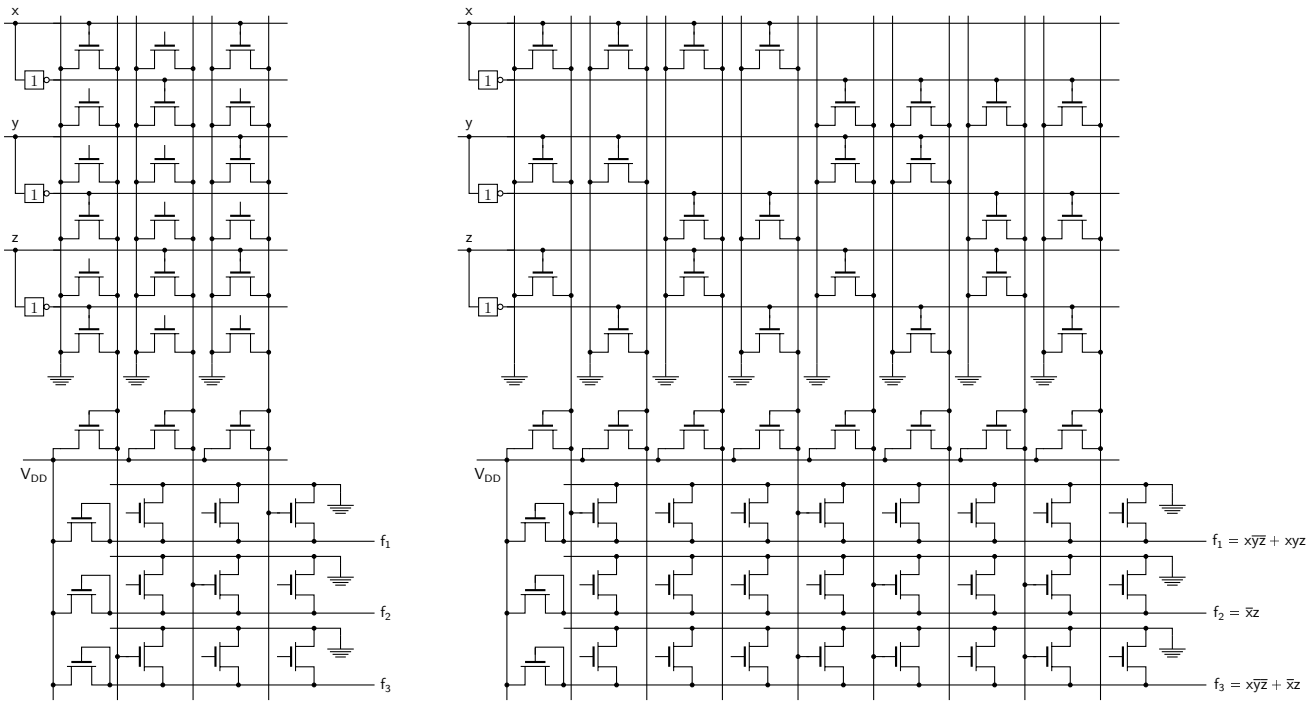


Abbildung 7.12: PLA-Beispiel

7.5 Andere programmierbare Logikschaltungen

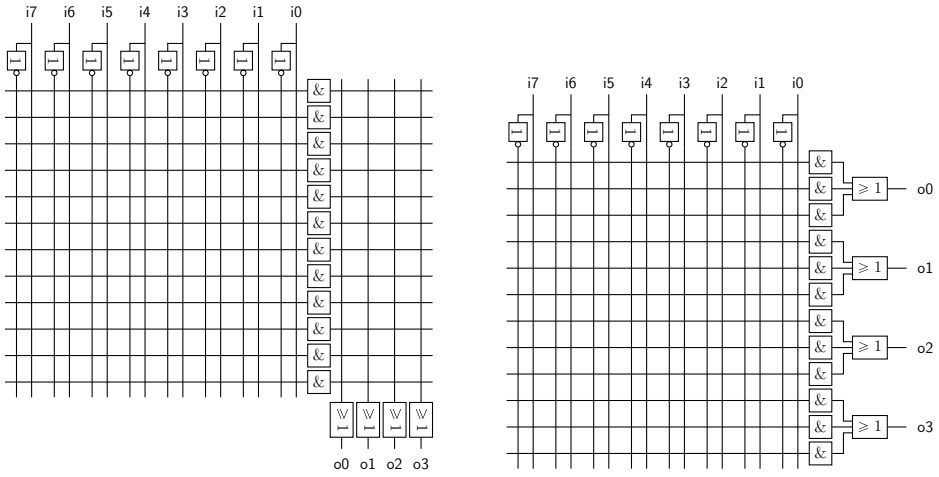
Neben PLAs gibt es noch weitere Strukturen auf dem Markt, es handelt sich in der Regel um elektronisch programmierbare Schaltungen:

Array-Logikbausteine PLA und PAL

Der PLA ist der Vorläufer aller Array-Bausteine (PLA ab 1973, FPLA ab 1975) und besteht aus einem programmierbaren UND-Array und einem programmierbaren ODER-Array. Jeder Produktterm des UND-Arrays kann mit jeder Ausgangsfunktion verknüpft werden.

Der PAL entspricht einer Vereinfachung der PLA-Struktur (ab 1978) durch ein fixiertes ODER-Array, auch GAL und PLD genannt. Hierbei werden die Produktterme einer festen Struktur der Ausgangsfunktion zugeordnet, das ODER-Array wird durch eine einfache feste Verdrahtungsstruktur ersetzt. Die Kapazität der PAL-Bausteine liegt im Bereich der von PLAs.

Weitere Unterschiede gibt es im Logikentwurf. Bei PLAs können Produktterme mehrfach von verschiedenen Ausgangsfunktionen genutzt werden, bei PALs ist das nicht möglich, für jede Ausgangsfunktion müssen eigene Produktterme implementiert werden. Daher benötigen PAL-Entwürfe in der Regel mehr Produktterme als PLA-Entwürfe. Beide ermöglichen eine zweistufige Implementierung jeder booleschen Funktion (UND/ODER, disjunktive Normalform), vorausgesetzt der Baustein enthält genügend



(a) PLA-Struktur (b) PAL-Struktur

Abbildung 7.13: Strukturen programmierbarer Logikschaltungen

Produktterme.

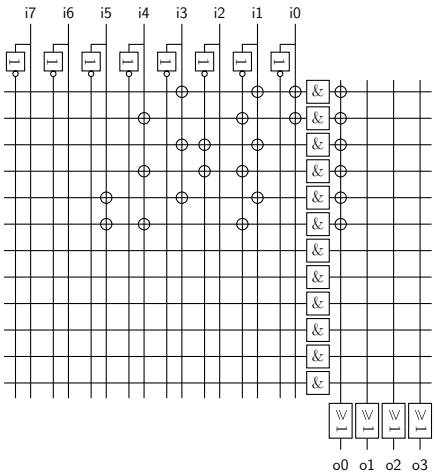


Abbildung 7.14: PLA-Beispiel

Ähnlich der Struktur von PLAs, PALs und PGAs ist die Struktur von programmierbaren Logik-Sequenzern. Flipflops können vom D- oder JK-Typ sein und auch Rückkopplungen sind möglich.

7.6 Programmierbares Gate Array (FPGA)

Benutzerprogrammierbare Gate Arrays bieten eine hohe Flexibilität, da auch Leitungssegmente programmiert werden können. Field Programmable Gate Arrays basieren auf dem Logikblock-Konzept. Ein Logikblock besteht aus einer kleinen Anzahl von Logikgattern, die flexibel programmierbar sind. Die Logikfunktion wird in Multiplexern

(Look-up tables, LUTs) oder RAM-Blöcken abgelegt, zusätzlich können noch Register existieren. Jedes FPGA enthält eine größere Anzahl von Logikblöcken (100 bis 500), jeder Block ist mit den anderen Blöcken und den I/O-Ports durch programmierbare Verknüpfung verbunden. Dadurch ist eine Verknüpfung der Logikblöcke untereinander sowie mit den I/O-Ports möglich. Es gibt verschiedene Programmierarten: EPROM/EEPROM, Schmelzsicherungen, SRAM.

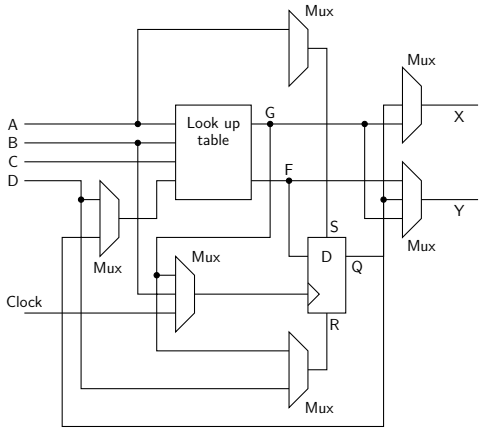


Abbildung 7.15: Struktur eines konfigurierbaren Logikblocks (Xilinx)

Die EPROM/EEPROM-Programmierung ist identisch mit der von PLDs, bei denen durch Floating-Gate-MOS-Transistor die zu programmierende Schalterverbindung gesteuert wird. Durch das Löschen des EPROM ist eine Wiederprogrammierung des FPGA's möglich.

Die Programmierung mittels Schmelzsicherungen ist ähnlich der Programmierung mit EPROMs, mit der Ausnahme, dass kein Löschen möglich ist. Zur Programmierung sind keine MOS-Transistoren erforderlich, so dass eine hohe Packungsdichte, geringe Laufzeiten und ein niedriger Preis erzielt werden können.

Bei der SRAM-Programmierung wird der programmierbare Schalter durch eine 1Bit-SRAM-Speicherzelle kontrolliert, bei Spannungsverlust verliert die SRAM-Zelle die Information und das FPGA seine Konfiguration. Zur Aufrechterhaltung der Konfiguration wird der Speicherinhalt in ein externes ROM/EPROM geladen, das dann mit einer speziellen Power-On-Routine das FPGA konfiguriert wird. Ein FPGA benötigt keine speziellen Programmiergeräte (nur für ROM/EPROM), allerdings Software zur Synthese. Die flexible Infrastruktur zur Programmierung verteuert das FPGA gegenüber dem PLD und die Software zur Synthese ist aufwendiger und damit teuer, außerdem sind die Verzögerungszeiten uneinheitlich und erst nach Plazierung und Verdrahtung bestimmbar.

Beispiel 81. *Konfiguration einer NOT-Funktion*
Der Logikblock hat 8 Eingänge und einen Ausgang, besteht aus drei 2:1 *Multiplexern und einem OR-Gatter, die NOT-Funktion wird mit einer 1 am A und S_0 Eingang und einer 0 am B Eingang festgelegt, die Variable X erzeugt so am S_A Eingang die NOT-Funktion am Y-Ausgang.*

Funktion	Gatterbedarf	benötigte Eingänge
NOT	9/14	4/8
2-AND	9/14	4/8
2-OR	14/14	6/8

Tabelle 7.1: Auslastungseffizienz einzelner Funktionen

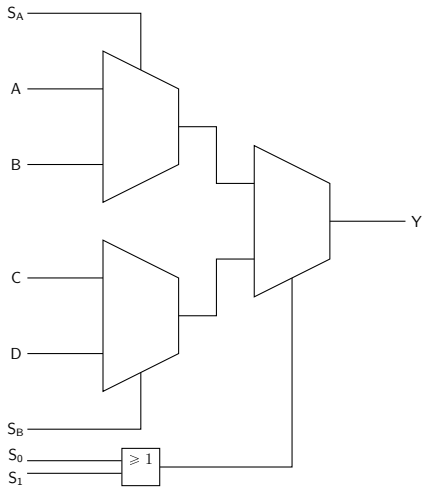


Abbildung 7.16: Struktur eines einfachen kombinatorischen Logikblocks

In einem Logikblock (XILINX CMOS LCA)besteht jede Zelle aus einem Schaltnetz und einem Speicherelement. Die Zellen können als jede Funktion von 4 Variablen oder als zwei Funktionen von 3 Variablen programmiert werden und werden als konfigurierbare Logikblöcke (CLB configurable logic block) bezeichnet.

Zellverknüpfung

Die Verknüpfung der Logikblöcke und E/A-Module geschieht über horizontal und vertikal verlaufende Verbindungen miteinander. Beim kleinsten Baustein (Actel A1010) mit 295 Logikblöcken erfolgt z.B. die Verknüpfung über 22 horizontale und 13 vertikale Spuren. Die Verbindung wird mittels Schmelzsicherungen/Transistoren hergestellt, die nach der Programmierung eine niederohmige (ca. $50\,\Omega$) Verbindung ergeben.

E/A Module

Die externe Schnittstelle wird von E/A-Modulen übernommen, E/A-Module lassen sich zu vier Konfigurationen programmieren: Eingang, Ausgang, Tri-state-Ausgang, bidirektionales Buffer.

Die LCA Architektur (Logic Cell Array) besteht aus einer Matrix identischer Logik-Zellen, die von Ein-/Ausgabezellen umgeben ist. Das Verbindungsnetzwerk besteht aus Leitungssegmenten, die horizontal und vertikal entlang in Kanälen zwischen den Zellen

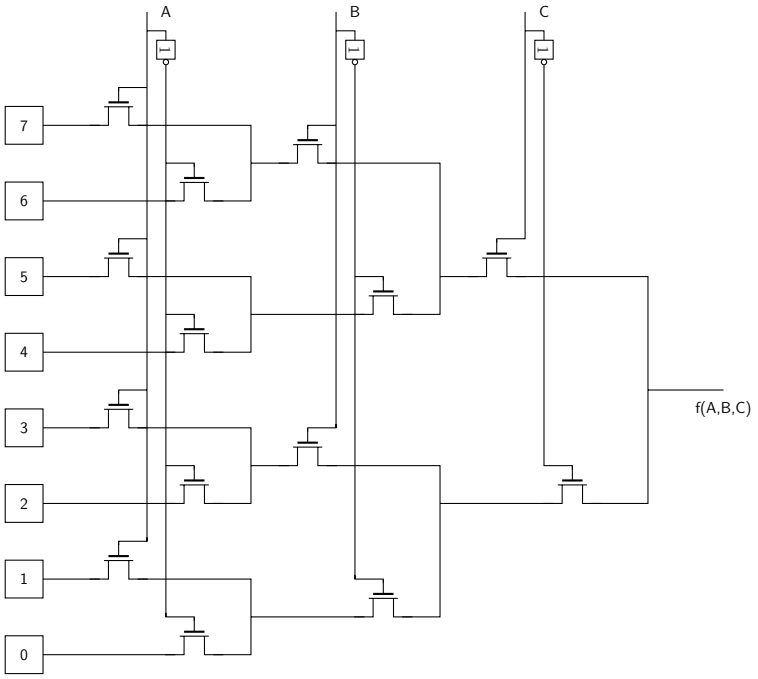


Abbildung 7.17: Look-up Funktionsgenerator mit 8 Speicherzellen und 8 zu 1 Multiplexer

verlaufen. Die Programmierung der Verdrahtung erfolgt durch die Schaltermatrix, die sich an den Kreuzungspunkten zwischen den Reihen und Spalten befindet.

Programmierung

Die Logikzellen, Peripheriezellen und die Verdrahtung sind vom Benutzer programmierbar. Die Information zur Konfiguration der Logik und Verdrahtung wird im Speicher gehalten. Die Programmierung des Speichers erfolgt mittels eines Entwurfssystems, das die Schaltungseingabe, Makrobibliothek, Simulation und Schaltungsumsetzung unterstützt. Die Plazierung und Verdrahtung erfolgt automatisch oder interaktiv manuell, dafür ist spezielle Entwurfssoftware erforderlich, die den entsprechenden IC konfiguriert. Die Schaltungskonfiguration kann im EPROM/ROM auf der Platine oder im Rechner gespeichert werden.

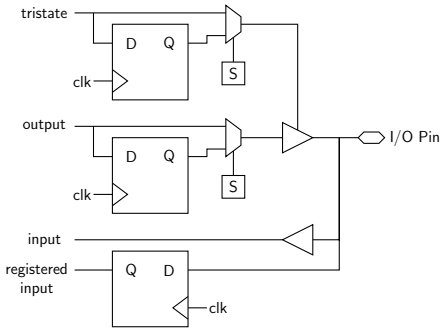


Abbildung 7.18: Prinzipieller Aufbau einer Ein-/Ausgabezelle

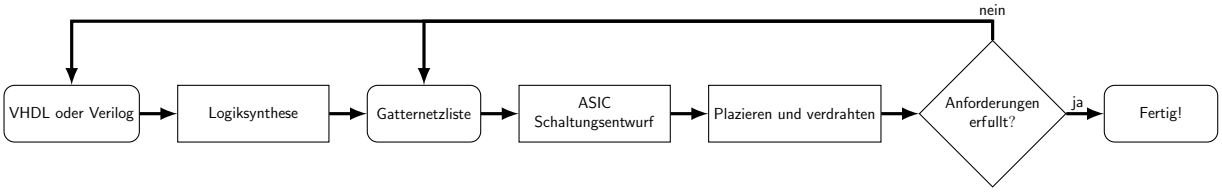


Abbildung 7.19: Programmierungs-Workflow

8 Technische Aspekte des Logikentwurfs

8.1 Realisierung der Schaltalgebra durch Logikgatter

Überblick

Anstelle der Schalterrealisierung durch Reihen- und Parallelschaltung lassen sich die Operationen $+$, $\cdot$, $\overline{\ }$ mit sogenannte Logikgattern oder digitalen Verknüpfungsgliedern durchführen. Es handelt sich dabei meist um elektronische Schaltungen, die auch Schaltkreis genannt werden.
Ein Gatter ist als Blackbox mit einem oder mehreren Eingängen und einem Ausgang gegeben. Von einer solchen Blackbox ist nur das Schaltverhalten bekannt. Dabei bezeichnet das Schaltverhalten das Verhalten der Spannung am Ausgang in Abhängigkeit von den an den Eingängen angelegten Spannungen. Die am Eingang angelegten Spannungen werden als Eingangssignale und die resultierende Spannung am Ausgang als Ausgangssignal bezeichnet.

Zuordnung von Signalen zu Logikwerten

Als Eingangssignale sind bei digitalen Gattern nur zwei verschiedene Spannungswerte (Stromwerte) innerhalb gewisser Toleranzgrenzen zugelassen. Dies gilt auch für das Ausgangssignal. Beiden Spannungswerten werden jeweils ein Schaltzustand zugewiesen, z.B. entspricht „0V“ dem Schaltzustand 0 und „5V“ dem Zustand 1. Sind somit z.B. 0V und 5V die zulässigen Spannungswerte, lässt sich zu jedem Schaltgatter eine entsprechende Schalttabelle angeben.

8.1.1 Positive Und Negative Logik

Die Realisierung logischer Funktionen durch Hardware erfordert die Darstellung der binären Variablen durch geeignete elektrische Größen. In digitalen Schaltkreisen erfolgt somit die Darstellung der boolschen Werte 0 und 1 durch zwei unterschiedliche Spannungspegel bzw. Spannungspegelbereiche. Diese Zuordnung der High-(H) und Lowpegel(L) kann entweder durch positive oder negative Logik geschehen.

	Pegel		phys. Größe	
Wert	pos. Logik	neg. Logik	pos. Logik	neg. Logik
0	L	H	z.B. 0V	z.B. 0V
1	H	L	z.B. +5V	z.B. -12V

Die Zuordnung der physikalischen Größen kann dabei willkürlich erfolgen, da die mathematische Logik keine Unterscheidung in positive und negative Logik besitzt. Diese ist somit nur erforderlich, wenn man die technische Realisierung der logischen Funktionen betrachtet.

8.1.2 Pegelbereiche

Problemstellung

Die Spannungspegel innerhalb digitaler Schaltungen unterliegen Streuungen. Die Ursachen liegen in Bauelementtoleranzen, Temperaturschwankungen, Betriebsspannungsschwankungen und Störsignalen.

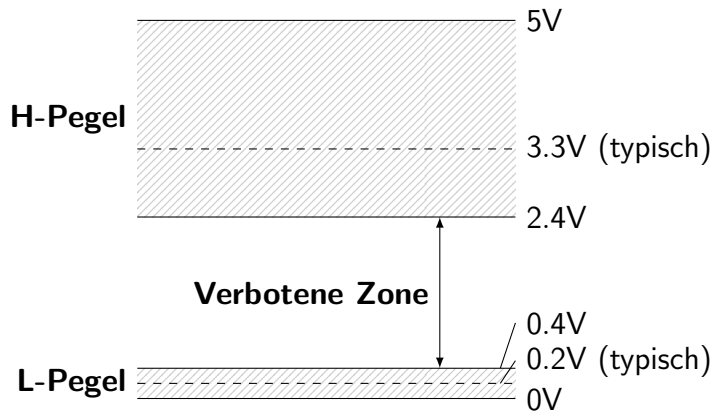


Abbildung 8.1: Spannungspegelbereiche für den H- und L-Pegel

Eine Verringerung des Einflusses der Toleranzen geschieht durch die Zuordnung der Spannungspegel H und L zu einem jeweils relativ breiten Pegelbereich. Beide Pegelbereiche werden durch eine verbotene undefinierte Zone getrennt.

8.1.3 Übertragungs(Transfer)Kennlinie

Elektronisches Verhalten

Die Beziehung zwischen Aus- und Eingangsgröße - z.B. Spannung (U_A , U_E) - eines Gatters heißt Transfer- oder Übertragungskennlinie. Sie stellt die wichtigste statische Eigenschaft eines Gatters dar. Die Transferkennlinie hängt von der Schaltungsstruktur, den gewählten Bauelementen, der Belastung und der Temperatur ab. Sie zeigt den typische Logikpegel und den statischen Störabstand.

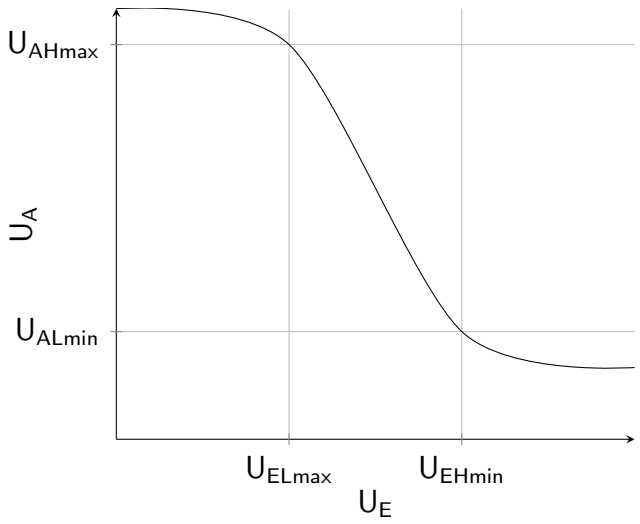


Abbildung 8.2: Übertragungskennlinie eines Inverters

8.1.4 Statischer Störabstand

Definition des statischen Störabstandes U_S

Der statische Störabstand U_S beschreibt den zulässigen Störspannungshub, der den logischen Zustand eines Gatters noch nicht ändert. Die Voraussetzung dafür ist, dass das Störsignal länger als die mittlere Verzögerungszeit dauert. Er definiert auch die Störspannung, die dem Ausgangssignal einer Stufe überlagert sein darf, ohne bei einer nachgeschalteten gleichen Stufe den zulässigen Eingangswert zu überschreiten. Die Störabstände für L- und H-Pegel sind verschieden und werden getrennt (U_{SL}, U_{SH}) angeben.

$$\boxed{\begin{aligned} U_{SH} &= U_{AHmin} - U_{EHmin} \\ U_{SL} &= U_{ELmax} - U_{ALmax} \end{aligned}}$$

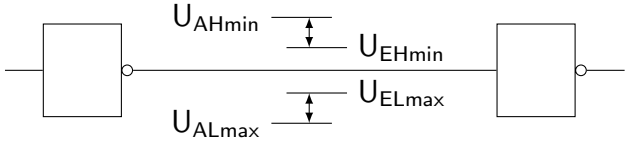


Abbildung 8.3: Definition des statischen Störspannungsabstandes

8.1.5 Dynamischer Störabstand

Problemstellung

Ist die Eingangssignaldauer kleiner als die Gatterverzögerungszeit, ist die Gatterfunktion nicht mehr gewährleistet.

Definition des dynamischen Störabstandes

Der dynamische Störabstand ist die minimal erforderliche Impulslänge t_{min} am Eingang eines Gatters, bei der das Gatter noch korrekt schaltet.

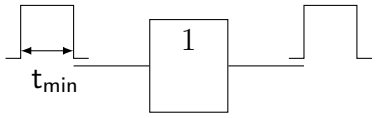


Abbildung 8.4: Dynamischer Störabstand

8.1.6 Belastbarkeit

Die Gattereingänge bilden Last für angeschlossene Ausgänge. Bei bipolaren Schaltungen handelt es sich um eine Strombelastung, MOSFET-Schaltungen weisen hingegen nur eine kapazitive Last auf. Meistens interessiert nur, wieviele Eingänge von nachfolgenden Gattern an einen Gatterausgang angeschlossen werden können. Man rechnet daher nicht mit Strömen, sondern mit festgelegten Lasteinheiten (unit load), in denen sämtliche „worst-case“-Bedingungen berücksichtigt sind. Die Berechnung der Lastfaktoren erfolgt getrennt für H- und L-Pegel. In Datenblättern wird dann der größere bzw. kleinere Wert angegeben.

I_{E_H} Eingangsstrom bei H-Pegel	I_{E_L} Eingangsstrom bei L-Pegel
I_{A_H} Ausgangsstrom bei H-Pegel	I_{A_L} Ausgangsstrom bei L-Pegel

Abbildung 8.5: Definition der Gatterströme

Eingangslastfaktor (fan in)

Faktor 1 ist die Belastung des Eingangs eines elementaren Grundgatters.

$$(\text{fanin})_H = \frac{I_{E_H}}{I_{E_{H_E}}} \quad (\text{fanin})_L = \frac{I_{E_L}}{I_{E_{L_E}}}$$

Ausgangslastfaktor (fan out)

Anzahl von Eingangslastfaktoren 1 der gleichen Schaltkreisfamilie, mit der ein Schaltkreisausgang belastet werden darf.

$$(\text{fanout})_H = \frac{I_{A_H}}{I_{E_{H_E}}} \quad (\text{fanout})_L = \frac{I_{A_L}}{I_{E_{L_E}}}$$

8.1.7 Verlustleistung

Mittlere statische Verlustleistung P_V

Fast alle Schaltkreisfamilien haben unterschiedliche Leistungsaufnahmen bei einem Ausgangspegel L bzw. H.

$$P_V = \frac{P_{VH}+P_{VL}}{2} \text{ arithmetisches Mittel der Verlustleistung}$$

dynamische Verlustleistung

Bei verschiedenen Schaltkreisfamilien tritt eine hohe dynamische Verlustleistung auf, die durch den Signalwechsel verursacht wird.

$$P_{VCMOS} \sim f \text{ mit } f \triangleq \text{Taktfrequenz}$$

8.1.8 Schaltzeiten

Problemstellung

Das dynamische Schaltverhalten digitaler Schaltkreise wird durch die Verzögerungszeit t_p, die Anstiegszeit t_r und Abfallzeit t_f bestimmt. Die Schaltzeiten hängen somit unter anderem von der Art der Beschaltung des Ausgangs ab.

Verzögerungszeit (propagation delay, Durchlaufzeit) t_p

Man unterscheidet zwischen den Verzögerungszeiten t_{pHL} und t_{pLH} entsprechend der HL- oder LH-Ausgangsflanke. Die Verzögerungszeiten werden bei 50% des Signalspannungspegels gemessen. Somit ist t_{pLH} das propagation delay von L nach H und t_{pHL} von H nach L. Die Gatterverzögerungszeit errechnet sich aus dem arithmetischen Mittelwert $t_{pd} = \frac{1}{2} \cdot (t_{p_{HL}} + t_{p_{LH}})$.

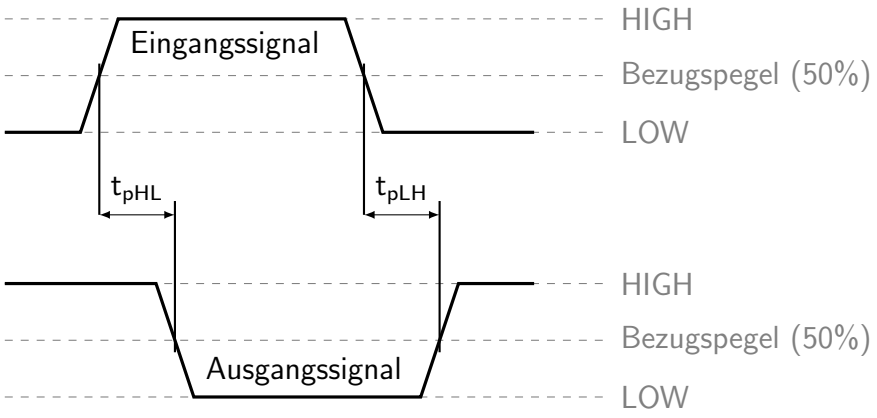


Abbildung 8.6: Propagation-Delay

Abfallzeit(fall time) und Anstiegszeit(rise time)

Die Anstiegs- bzw. Abfallzeit liegt zwischen 10% und 90% des Spannungshubs bei einer Ansteuerung mit einem idealen Rechteckimpuls am Eingang.

Symbol	eng. Beschreibung	Taktflankenänderung	deutsche Beschreibung
t_r	rise time	L → H	Anstiegszeit
t_f	fall time	H → L	Abfallzeit

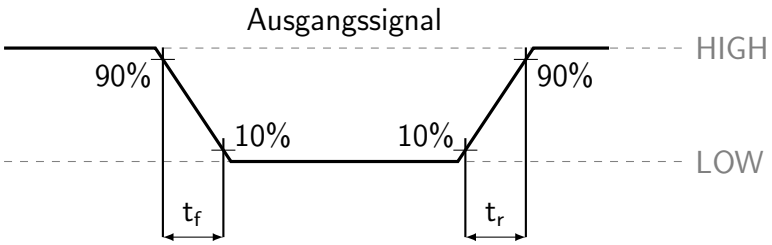


Abbildung 8.7: Definition Schaltzeiten

8.1.9 Geschwindigkeit-Leistungs-Produkt

Das Geschwindigkeits-Leistungs-Produkt W_G dient häufig als Bewertungskriterium für das dynamische Verhalten logischer Schaltkreise.Dieses Produkt beschreibt das Verhältnis zwischen der Verlustleistung(P_V) und der Verzögerungszeit(t_{pd}). Beschleunigt man also eine Anwendung oder Schaltung geschieht das auf Kosten der steigenden Verlustleistung oder anders herum. Das Geschwindigkeits-Leistungs-Produkt kann man auch mit der Formel $W_G = P_V \cdot t_{pd}$ beschreiben.

8.2 Dynamisches Verhalten von Schaltnetzen

8.2.1 Hazards

Die unterschiedlichen Verzögerungen einzelner Eingangssignale können in einem Schaltnetz Eingangskombinationen verursachen, die kurzzeitig zu fehlerhaften Ausgangskombinationen führen können. Diesen Effekt bezeichnet man auch als „Hazard“. Ein Hazard ist ein Übergang zwischen zwei Eingangskombinationen, bei dem die Möglichkeit besteht, dass während der Übergangsphase auf Grund unterschiedlicher Signalverzögerungen falsche Ausgangsignale auftreten. Ob dann tatsächlich der Fehler auftritt, hängt von den realen Verzögerungszeiten der einzelnen Signale ab. Man spricht hierbei dann von einem hazardbehafteten Schaltnetz.

Gegeben sei folgender boolscher Ausdruck: $y = x_2 \cdot \overline{x_0} + \overline{x_2} \cdot x_1$.

Als Beispiel kann man hier den Übergang von der Eingangskombination (x_2, x_1, x_0) 110 nach 001 betrachten. Im ersten Fall treten im Übergangsintervall kurzzeitig fälschlicherweise die Signale 0 und 1 auf. Bei den angenommen Verzögerungen hat das hazardbehaftete Schaltnetz auch einen Hazard zur Folge. Im zweiten Fall erfolgt in der

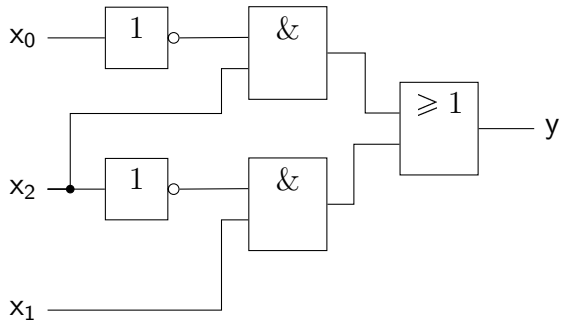
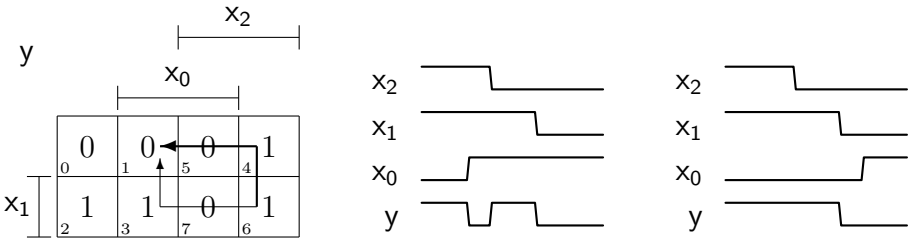


Abbildung 8.8: Hazardbehaftete Schaltung



(a) KV Diagramm (b) Verzögerungsvariante 1 (c) Verzögerungsvariante 2

Abbildung 8.9: Entstehung von Hazards

Übergangsphase ein eindeutiges Umschalten vom Wert 1 auf den Wert 0. Es ergibt sich keine falsche Aufeinanderfolge von Werten der Ausgangssignale. In diesem Fall liegt also für das gleiche Schaltnetz kein Hazard vor.
Bezüglich der Werte der Ausgangssignale, die vor und nach dem Übergang gefordert sind, unterscheidet man zwischen statischen und dynamischen Hazards.

Statischer Hazard

In einer Schaltung y(x) heißt ein beim Übergang zwischen den Eingangskombinationen x_1 und x_2 auftretender Hazard statischer Hazard, wenn bezüglich der Werte der Ausgangssignale gilt: $y(x_1) \rightarrow y(x_2)$.
Bei statischen Hazards unterscheidet man noch zwischen 0-Hazards und 1-Hazards. Ein statischer 0-Hazard (bzw. 1-Hazard) ist ein Hazard, bei dem das Ausgangssignal vor und nach dem Übergang gleich 0 (bzw. gleich 1) ist.

Dynamischer Hazard

In einer Schaltung y(x) heißt ein beim Übergang zwischen den Eingangskombinationen x_1 und x_2 auftretende Hazard dynamischer Hazard, wenn bezüglich der Werte der Ausgangssignale gilt: $y(x_1) \rightarrow \overline{y(x_2)}$.

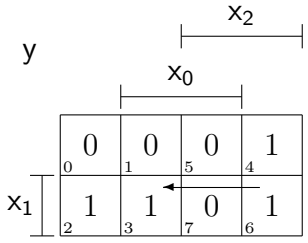
(a) statischer 0- und 1-Hazard (b) dynamischer 0- und 1-Hazard

Abbildung 8.10: Typen von Hazards

8.2.2 Funktionshazards

Statischer Funktionshazard

Ein Schaltnetz y(x) ist bei einem Übergang zwischen den Eingangskombinationen x_1 und x_2 mit einem statischen Funktionshazard behaftet, wenn es eine Wertekombination g für die Variablen x gibt, die während des Übergangs auftritt und für die gilt: $y(x_1) \rightarrow \overline{y(g)} \rightarrow y(x_2)$.
Als Kriterien zur Erkennung eines statischen Funktionshazards dienen zum einen der Funktionswert vor und nach dem Übergang, der gleich sein muss. Zum anderen muss es Wertekombinationen des Eingangsvektors g geben, die während des Übergangs momentan auftreten und die eine Änderung des Funktionswerts bewirken.
Betrachtet man in der Schaltung $y = x_2 \cdot \overline{x_0} + \overline{x_2} \cdot x_1$ z.B. den Übergang von 110 nach 011, dann tritt ein statischer Funktionshazard auf.



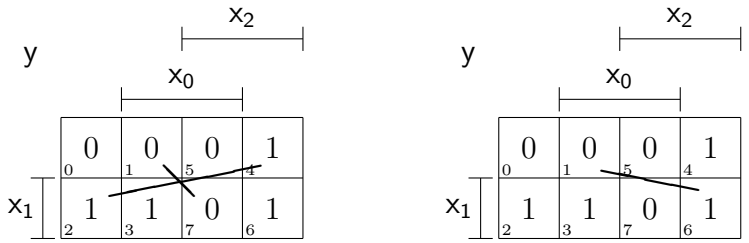
Dynamischer Funktionshazard

Ein Schaltnetz y(x) enthält beim Übergang zwischen den Eingangskombinationen x_1 und x_2 einen dynamischen Funktionshazard, wenn es zwei Wertekombinationen g_1 und g_2 von x gibt, die, ausgehend von x_1 zeitlich nacheinander auftreten und für die gilt: $y(x_1) \rightarrow \overline{y(g_1)} \rightarrow y(g_2) \rightarrow \overline{y(x_2)}$.
In den KV-Diagrammen sind einige Funktionshazards durch Linien gekennzeichnet.

8.2.3 Strukturhazards

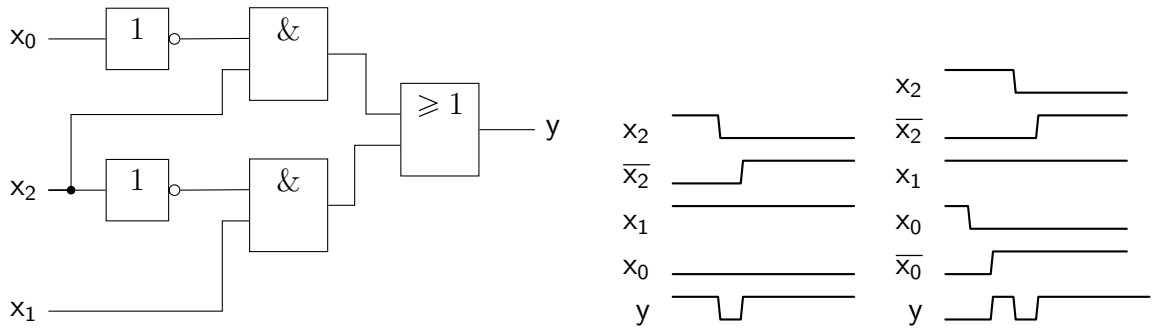
Ursache

Beim Übergang von 110 nach 010 entsteht kein Funktionshazard, da sich nur ein Variablenwert ändert. Dennoch kann während dieses Übergangs ein falsches Ausgangssignal enstehen. Die Ursache des falschen Ausgangssignals sind die Verzögerungen, die innerhalb eines Schaltnetzes zwischen einem Signal und seiner Negation wirksam sind.



(a) statisch (b) dynamisch

Abbildung 8.11: Funktionshazard



(a) Schaltbild (b) Entstehung eines statischen 1-Strukturhazards (c) Entstehung eines dynamischen Strukturhazards

Abbildung 8.12: Entstehung eines Strukturhazards

Das Ausgangssignal nimmt kurzzeitig den Wert 0 an, obwohl es den Wert 1 haben soll. Gemäß der Definition liegt ein statischer 1-Hazard vor.
Dieser Hazard wird allerdings in diesem Fall Strukturhazard genannt. Die Ursache dieses Strukturhazards ist, dass die Wertänderungen der Signale x und $\bar{x}$ nicht genau zeitgleich erfolgen. Es gibt also Zeitpunkte, zu denen beide Signale x und $\bar{x}$ gleich 0 oder gleich 1 sind. Hat x_1 den Wert 1 und x_0 den Wert 0 und wird der Wert von x_2 geändert (Übergang von 110 nach 010), tritt in der Schaltung ein statischer Strukturhazard auf. Dieser statische 1-Strukturhazard ist in dem KV-Diagramm durch eine Linie gekennzeichnet.

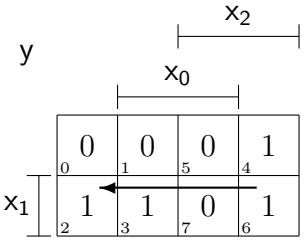


Abbildung 8.13: Statischer Strukturhazard

Die Veranschaulichung eines dynamischen Strukturhazards (siehe Abbildung 8.12c) zeigt den Übergang von 111 nach 010. Dabei wurde angenommen, dass die Verzögerung von x_2 größer ist als die Verzögerung von x_0. man erkennt, dass der Wert von y sich zunächst von 0 nach 1 ändert, dann noch einmal nach 0 zurückgeht und erst dann wieder 1 wird. Es ensteht also ein zusätzlicher Einsimpuls.

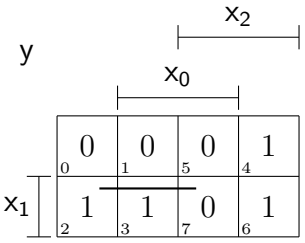


Abbildung 8.14: Dynamischer Strukturhazard

8.2.4 Vermeidung von Hazards

Übersicht

Ob die in Schaltnetzen auftretenden Hazards für eine nachgeschaltete Anordnung kritisch sind, hängt im wesentlichen von deren Zeitkonstanten ab. Deshalb ist es notwendig durch geeignete Maßnahmen die verschiedenartigen Hazardeinflüsse zu vermeiden.

Wichtigste Maßnahme zur Vermeidung von Hazards

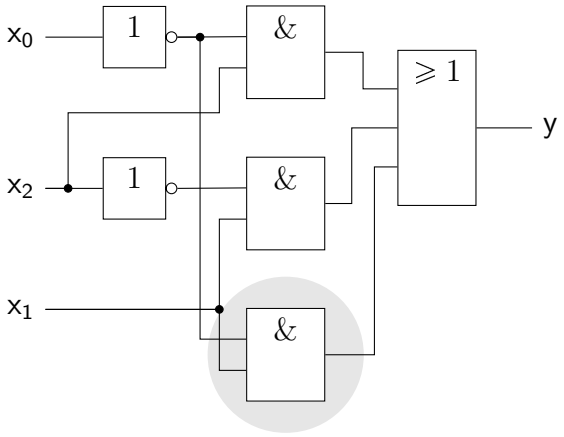
Es gibt drei wichtige Maßnahmen die man ergreifen kann, um Hazards zu vermeiden. Zum einen verhindert ein Taktbetrieb (eine Synchronisierung der Schaltung), dass aufge-

tretene Hazards sich in nachfolgenden Schaltungsteilen ausbreiten können. Zum anderen kann man durch gezieltes Hinzufügen von Verzögerungsgattern in die Eingangsleitung einen Ausgleich von vorhandenen Verzögerungen bewirken. Darüber hinaus kann auch eine Änderung der Schaltstruktur - also Umformung der boolschen Algebra - Hazards verhindern.

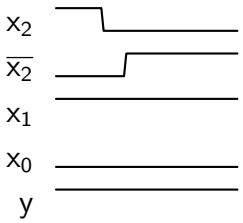
Folgende Funktion beschreibt ein Schaltnetz in dem keine statischen Strukturhazards auftreten.

$$y = x_2 \cdot \overline{x_0} + \overline{x_2} \cdot x_1 + x_1 \cdot \overline{x_0}$$

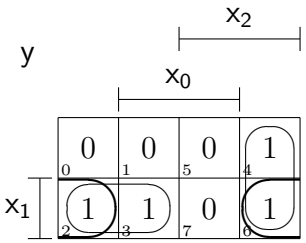
Es kann gezeigt werden, dass durch die Hinzunahme der redundanten Primimplkanten $x_1 \cdot \overline{x_0}$ auch die beiden dynamischen Strukturhazards beseitigt werden.



(a) zusätzliches UND-Element einfügen



(b) zeitlicher Ablauf



(c) Darstellung als KV-Diagramm

Abbildung 8.15: Beseitigung des Strukturhazards

Eine Schaltung die durch eine disjunktive Normalform beschrieben wird, ist genau dann frei von allen statischen und dynamischen Strukturhazards, wenn die disjunktive Normalform aus allen Primimplikanten der betroffenen Funktion aufgebaut ist. Die Vermeidung von Strukturhazards durch hinzufügen von Primimplikanten ist somit sehr aufwendig.

9 Synchrone Schaltwerke

9.1 Einleitung

Sequentielle Schaltungen (Schaltwerke)

Ein Schaltwerk besteht aus einem **Schaltnetz** und einem **Speicherteil** zur Sicherung vorangegangener Informationen. Die Ausgangsvariablen (Steuervariablen) z^t hängen von den Eingangsvariablen x^t und von den gespeicherter Information (aktueller Zustand) y^t ab. Als Speicherelemente kommen hauptsächlich Master-Slave-D-Flipflops (CMOS) zum Einsatz (siehe 6.3.4). Zur Synchronisation wird der Speicher getaktet (synchrones Schaltwerk).

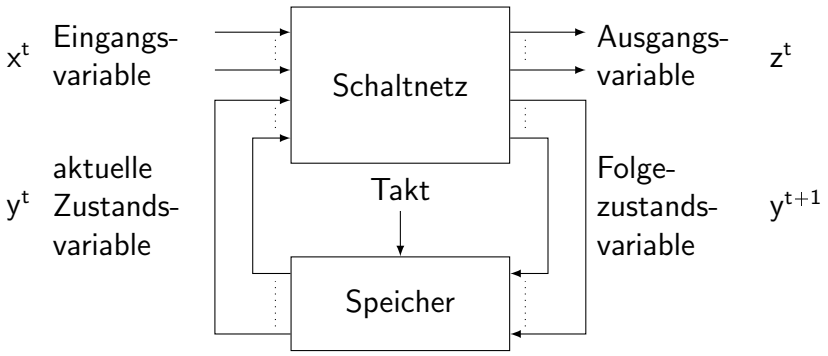


Abbildung 9.1: Schaltwerksstuktur

Darstellungen für Schaltwerke

Schaltwerke können durch Zustandstabellen bzw. Zustandsgraphen/Übergangsgraphen dargestellt werden. Bei Zustandstabellen werden ausgehend vom aktuellen Zustand y^t in Verbindung mit den Eingangsvariablen x^t die Folgezustände y^{t+1} und die Ausgangsvariablen z^t aufgezeigt. Zustands-/Übergangsgraphen visualisieren die Funktionsweise eines Schaltwerks graphisch. Hierbei repräsentieren die Knoten die Zustände. Die Kanten zeigen die Übergänge mit den zugehörigen Eingangs- und Ausgangssignalen an. Die Ausgangssignale können den Knoten oder den Kanten zugeordnet werden.

Arten von Automaten

Im Wesentlichen kann man zwei Arten von Schaltwerken/Automaten unterscheiden. Der **Mealy-Automat** ist dadurch gekennzeichnet, dass die Ausgangsvariablen z^t eine Funk-

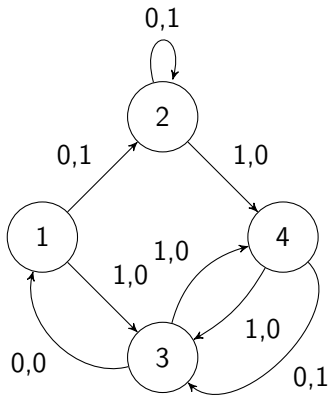
tion der Eingangsvariablen x^t und des aktuellen Zustands y^t ist. Die Folgezustände y^{t+1} sind eine Funktion der Eingangsvariablen x^t und des aktuellen Zustands y^t.

$$y^{t+1} = g(x^t, y^t)$$
$$z^t = f(x^t, y^t)$$

Beispiel 82. *Mealy-Automat mit Zustandstabelle und Zustandsgraphen*

	x = 0		x = 1	
y^t	y^{t+1}	z	y^{t+1}	z
1	2	1	3	0
2	2	1	4	0
3	1	0	4	0
4	3	1	3	0

(a) Zustandstabelle



(b) Zustandsgraph

Abbildung 9.2: Mealy-Automat

Der zweite Automatentyp wird **Moore-Automat** genannt. Dieses Schaltwerk bestimmt die Ausgangsvariablen z^t nur anhand des aktuellen Zustandes y^t. Der Folgezustand y^{t+1} bleibt eine Funktion der Eingangsvariablen x^t und des aktuellen Zustands y^t.

$$y^{t+1} = g(x^t, y^t)$$
$$z^t = f(y^t)$$

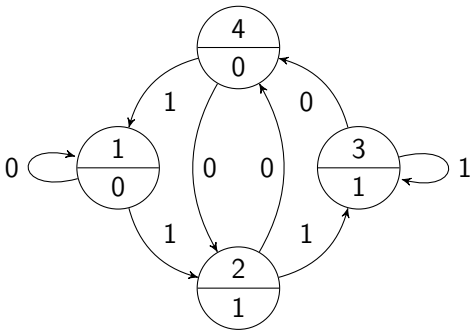
Beispiel 83. *Moore-Automat mit Zustandstabelle und Zustandsgraphen*

Eigenschaften der Automatentypen

Beide Steuerwerksarten sind äquivalent und jeweils in den anderen umwandelbar. Die Synthese beider Automatentypen ist ähnlich und muss dementsprechend nicht separat erklärt werden. Moore-Automaten erfordern in der Regel mehr Zustände. Mealy-Automaten hingegen können komplexere Übergangsfunktionen aufweisen. Die Steuerfunktionen können beim Mealy-Automaten asynchron reagieren, wenn sie nicht zwischengespeichert werden.

y^t	$x = 0$ y^{t+1}	$x = 1$ y^{t+1}	z
1	1	2	0
2	4	3	1
3	4	3	1
4	2	1	0

(a) Zustandstabelle



(b) Zustandsgraph

Abbildung 9.3: Moore-Automat

9.2 Schaltwerkssynthese

Schritte zur Schaltwerkssynthese

Die Schaltwerkssynthese kann im Wesentlichen in zwei Schritten realisiert werden. Zu Beginn wird dabei aus der Verhaltensbeschreibung einer sequentiellen Funktion eine Zustandstabelle bzw. ein Zustandsgraph abgeleitet. Die Ableitung einer Zustandstabelle aus der Verhaltensbeschreibung (High-Level-Synthese) bietet bisher noch nicht umfassend effiziente Lösungen und bleibt z. Z. aktueller Forschungsgegenstand. Der zweite Schritt beinhaltet die Realisierung der Zustandstabelle durch ein Schaltwerk. Die Umsetzung der Zustandstabelle in ein Schaltwerk ist ein weitgehend erforschter Bereich mit effizienten Lösungsansätzen.

Zustandszuordnung (state assignment)

Als eine Zustandszuordnung (state assignment) bezeichnet man eine Zuweisung einer Kombination von Zustandsvariablen y_i zu einem Zustand des Schaltwerks. Mit zweiwertigen Speicherelementen sind für n Zustände des Schaltwerks $\log_2 n$ Speicherelemente erforderlich. Daraus ergeben sich dann $n!$ verschiedene Zustandszuordnungen, deren Realisierungen zu unterschiedlich hohem Aufwand führen kann. Es sind verschiedene Algorithmen bekannt, die zu einer optimalen Zustandszuordnung führen. Ziel sind dabei z.B. eine minimale Anzahl von Gattereingängen, eine minimale Verzögerung oder minimale Verlustleistung. Eine Minimierung ist aber nur bei Schaltnetzrealisierung relevant, nicht bei ROM-Implementierung.

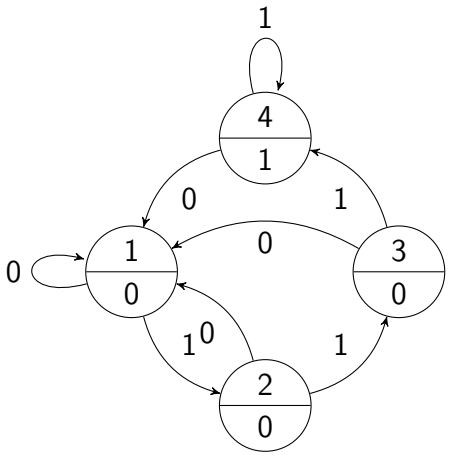
Beispiel 84. *Sequentieller serieller Bitprüfer als Moore-Automat mit MS-D-Flipflops*

Verhaltensbeschreibung
Die Schaltung soll für die Steuerfunktion z eine 1 erzeugen, wenn in einem bitseriellen Datenstrom 3 oder mehr aufeinanderfolgende Einsen auftreten.

Ableitung einer Zustandstabelle bzw. eines Zustandsgraphen
Zu Beginn muss gemäß der Verhaltensbeschreibung ein Ablaufschema entwickelt werden. Als mögliche Darstellungsform eignen sich eine Zustandstabelle und/oder ein Zustandsgraph.

aktueller Zustand y^t	neuer Zustand y^{t+1} $x=1$	$x=0$	Ausgang z
1	2	1	0
2	3	1	0
3	4	1	0
4	4	1	1

(a) Zustandstabelle



(b) Zustandsgraph

Abbildung 9.4: Bitprüfer

Realisierung der Zustandstabelle durch ein Schaltwerk
Auf Grundlage der Verhaltensanalyse durch die Zustandstabelle bzw. den Zustandsgraphen, ist eine Zustandscodierung vorzunehmen. Für die Codierung der vier vorgesehenen Zustände benötigt man entsprechend zwei Variablen (y_2, y_1). Folgerichtig ist die Zustandstabelle neu aufzustellen.

Zustand	y_2	y_1
1	0	0
2	0	1
3	1	1
4	1	0

(a) Zustandszuordnung (Gray-Code)

		$x=1$		$x=0$		
$y_2{}^t$	$y_1{}^t$	$y_2{}^{t+1}$	$y_1{}^{t+1}$	$y_2{}^{t+1}$	$y_1{}^{t+1}$	z
0	0	0	1	0	0	0
0	1	1	1	0	0	0
1	1	1	0	0	0	0
1	0	1	0	0	0	1

(b) neue Zustandstabelle

Abbildung 9.5: Zustandstabelle nach Codierung

Aus der neuen Tabelle können nun die Übertragungsfunktionen für den Folgezustand ($y_2{}^{t+1}, y_1{}^{t+1}$) und den Ausgang (z) ermittelt werden.

$$y_1{}^{t+1} = x\,\overline{y_2{}^t} \qquad y_2{}^{t+1} = x\,y_1{}^t + x\,y_2{}^t = x\,(y_1{}^t + y_2{}^t) \qquad z = \overline{y_1{}^t}\,y_2{}^t$$

NOR-NF: $\quad y_1{}^{t+1} = \overline{\overline{x} + y_2{}^t} \qquad y_2{}^{t+1} = \overline{\overline{x} + \overline{(y_1{}^t + y_2{}^t)}} \qquad z = \overline{y_1{}^t + \overline{y_2{}^t}}$

Der Bitprüfer soll durch ein Schaltwerk aus MS-D-Flipflops realisiert werden. Man benötigt also entsprechende Ansteuerfunktionen für die Flipflops. Hierbei ist anzumerken, dass die Ansteuerung direkt aus der Übertragungsfunktion abgeleitet werden kann ($D = Q^{t+1}$). Man setzt also D_1 für ${y_1}^{t+1}$ und D_2 für ${y_2}^{t+1}$ ein. Außerdem wird zur Vereinfachung ${y_1}^t$ durch y_1 und ${y_2}^t$ durch y_2 ersetzt.

Ansteuerungsfunktionen:

$D_1 = \overline{\bar{x} + y_2}$ $\qquad$ $D_2 = \overline{\bar{x} + \overline{(y_1 + y_2)}}$ $\qquad$ $z = \overline{y_1 + \overline{y_2}}$

Das resultierende Schaltwerk ist in der nachfolgenden Abbildung skizziert.

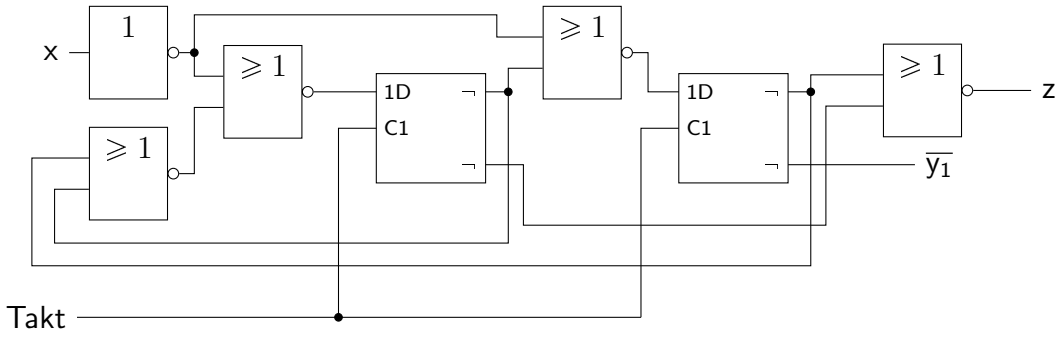


Abbildung 9.6: Prüfschaltung mit MS-D-Flipflops

Entwurf mit Festwertspeichern (ROMs)

Ein Schaltwerk kann gegebenenfalls auch mit einem ROM realisiert werden. Hierbei wird die Zustandstabelle durch den Festwertspeicher abgebildet. Die Eingangsvariablen x^t und Zustandsvariablen y^t werden als Adressbits verwendet. Die Folgezustandsvariablen y^{t+1} und die Ausgangsvariablen z^t werden im ROM gespeichert. Durch die Rückkopplung der Folgezustände auf den Eingang des ROMs wird ein sequentiellen Ablauf erreicht. Die Synchronisation wird mit MS-D-Flipflops oder Latches realisiert.

Entwurfsschritte:
Ausgangspunkt ist die Zustandszuordnungs- und die Ausgangstabelle. Die Tabelle wird als ROM-Übergangstabelle umgeformt und das Schaltwerk ist komplett realisiert. Häufig ist der ROM-Baustein aber größer (mehr Adressen (Zustände) und größere Wortbreite) als für ein spezielles Problem erforderlich. In diesen Fällen wird das ROM nur teilweise genutzt.

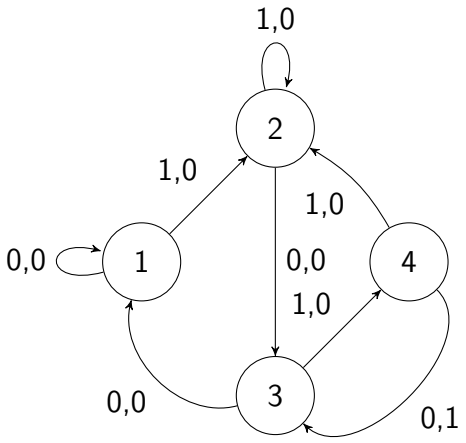
Beispiel 85. *Erkennung einer Bitsequenz*

Verhaltensbeschreibung:
Die Schaltung erzeugt die Steuerfunktion $z = 1$*, wenn die Bitsequenz* 1010 *erscheint. In allen anderen Fällen erscheint am Ausgang 0.*

Ableitung einer Zustandstabelle bzw. eines Zustandsgraphen:

	x = 0		x = 1	
y^t	y^{t+1}	z	y^{t+1}	z
1	1	0	2	0
2	3	0	2	0
3	1	0	4	0
4	3	1	2	0

(a) Zustandstabelle



(b) Zustandsgraph

Abbildung 9.7: Bitsequenz-Erkennung

Realisierung der Zustandstabelle durch ein Schaltwerk:

Zustandszuordnung (Gray-Code):

Zustand	y_2	y_1
1	0	0
2	0	1
3	1	1
4	1	0

neue Zustandstabelle:

		x = 0			x = 1		
$y_2{}^t$	$y_1{}^t$	$y_2{}^{t+1}$	$y_1{}^{t+1}$	z	$y_2{}^{t+1}$	$y_1{}^{t+1}$	z
0	0	0	0	0	0	1	0
0	1	1	1	0	0	1	0
1	1	0	0	0	1	0	0
1	0	1	1	1	0	1	0

ROM-Übergangstabelle:

Adresse			Ausgang		
A_2	A_1	A_0	D_2	D_1	D_0
y_2^t	y_1^t	x	y_2^{t+1}	y_1^{t+1}	z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	1	0	0

Durch eine Umsortierung nach Eingangs- und Ausgangsvariablen erhalten wir die benötigte ROM-Übergangstabelle. Der Eingang wird entsprechend als eine Adresse interpretiert. Schaltwerk:

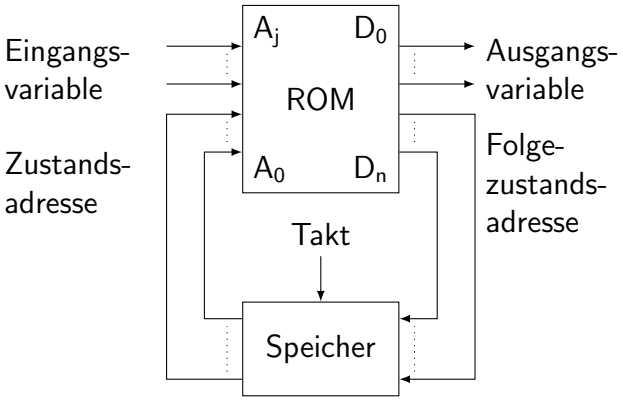


Abbildung 9.8: ROM-Implementierung

9.3 Register

9.3.1 Übersicht

Allgemeines

In Speicherorganisationen mit mehr als einem Bit Speicherkapazität erhalten die Speicherzellen eine gemeinsame Steuerung für die Datenein- und -ausgabe. Aufgebaut wird dieses System dann meist mit Flipflop-Speichern (Parallel- oder Serienspeicher). Speicher mit einer geringen Anzahl von Speicherzellen werden Register genannt. Register finden Anwendung als schnellen Zwischenspeicher für kleine Datenmengen.

Parallelspeicher

Die Datenein-/ausgabe erfolgt für alle Speicherzellen gleichzeitig. Die Steuerleitung T zur Datenübernahme ist parallel an alle Speicherzellen geschaltet. Parallelspeicher können

aus einfachen D-Latches oder aus MS-D-Flipflops aufgebaut werden.

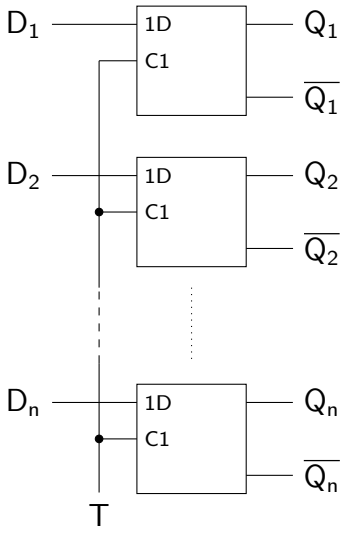


Abbildung 9.9: Parallelregister

Serienspeicher (Schieberegister)

Die Speicherzellen sind bei Serienspeichern in Reihe geschaltet. Jeder Ausgang der vorangehenden Zelle ist mit dem Eingang der nachfolgenden Zelle verknüpft. Die Daten werden beginnend von der ersten Zelle mit jedem Takt T in die nächste Zelle geschoben. Die Ausgabe der Daten erfolgt an der letzten Zelle der Speicherkette. Schieberegister sollten aus Master-Slave-D-Flipflops bestehen, da jede Speicherzelle gleichzeitig Daten übernehmen und weitergeben muss.

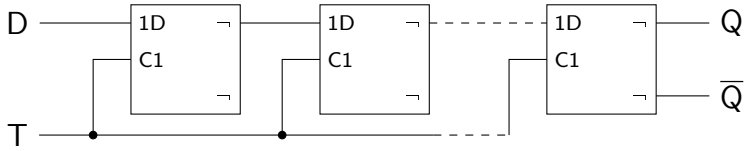


Abbildung 9.10: Serienspeicher (Schieberegister)

9.3.2 Technische Realisierung statischer Parallel- und Serienspeicher

Eigenschaften

Bei Bibliothekskomponenten handelt es sich meistens um Kombinationen aus Parallel- und Serienspeichern, um deren Anwendungsbereich zu erhöhen. Unterschieden werden

die Speicher nach Art der Ein- und Ausgabe der Daten an den Speicherzellen (parallel und/oder seriell). Weitere Eigenschaften der Speicherzellen können die Wahl der Schieberichtung nach rechts oder links sowie ein asynchrones Löschen oder Setzen sein.

Beispiel 86. *8-Bit-Schieberegister mit serieller Ein-/Ausgabe und paralleler Ausgabe*

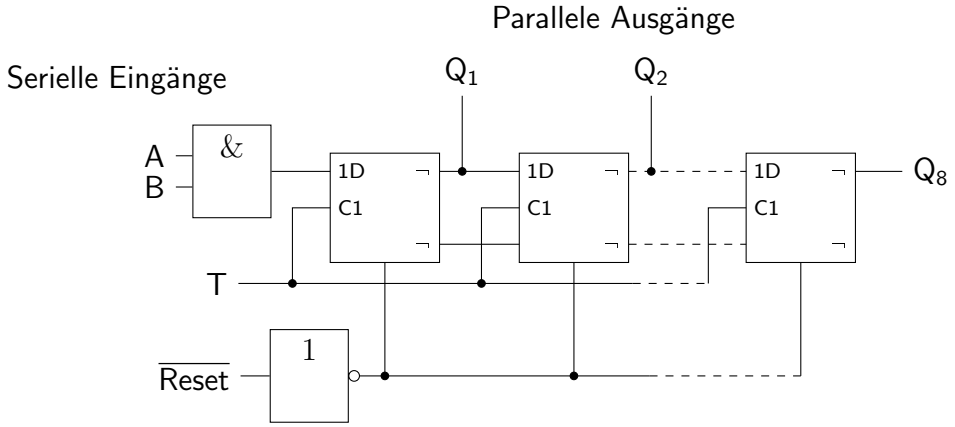


Abbildung 9.11: Schieberegister mit serieller Ein-/Ausgabe und paralleler Ausgabe

Diese Schaltung 9.11 ermöglicht die Serien-/Parallel-Umsetzung der Daten. Von den zwei seriellen Eingänge A *und* B *kann jeweils einer zu Steuerzwecken verwendet werden. Ist einer der beiden Eingänge '1', wird der andere durchgeschaltet.Die Daten werden während der positiven Taktflanke (Slave→Master) durch das Register geschoben und liegen gleichzeitig an den parallelen Ausgängen* $Q_1, ...Q_8$ *an. Das Register kann asynchron über den* Reset*-Eingang zurückgesetzt werden.*

Beispiel 87. *8-Bit-Schieberegister mit serieller Ein-/Ausgabe und paralleler Eingabe*
Diese Schaltung 9.12 ermöglicht ebenfalls die Serien-/Parallel-Umsetzung der Daten. Dieses Schieberegister ist umschaltbar zwischen Schiebebetrieb und parallelem Laden. Wenn das Signal Setzen $= 1$ *ist, dann wird derTakt an den Flipflops gesperrt und die parallel anliegenden Daten zu den Eingängen* $I_1, ...I_8$ *unabhängig vom Takt (asynchron) in die Flipflops übernommen. Andernfalls (*Setzen $= 0$*) sind die parallelen Eingänge gesperrt und der Takt ist freigegeben. Die Daten werden dann in Richtung des seriellen Ausgangs* Q_8 *abhängig vom Takt (synchron) verschoben. Bei gesetztem* Sperren*-Eingang wird der Schiebebetrieb unterbrochen und ein paralleles Laden ist ebenfalls nicht möglich.*

Beispiel 88. *Schieberegister mit zwei Schieberichtungen*
Bei arithmetischen Operationen sind oftmals beide Schieberichtungen erforderlich. Für Schiebeoperation von rechts nach links müssen die Ausgänge der Flipflops mit den Eingängen der links neben ihnen liegenden Flipflops verknüpft werden. Die steuerbare Schieberichtung erfordert einen Multiplexer, der je nach Schieberichtung den Ausgang mit dem Eingang des links oder rechts benachbarten Flipflops verbindet. Bei gesetztem R/L*-Signal wird das Schieben nach rechts vorgegeben. Die Daten von* D_R *werden übernommen und an* Q_4 *ausgegeben.* R/L $= 0$ *bewirkt folgerichtig das Schieben nach links. Das Verschieben der Daten erfolgt mit steigender Taktflanke (0→1). Während des Schiebens darf*

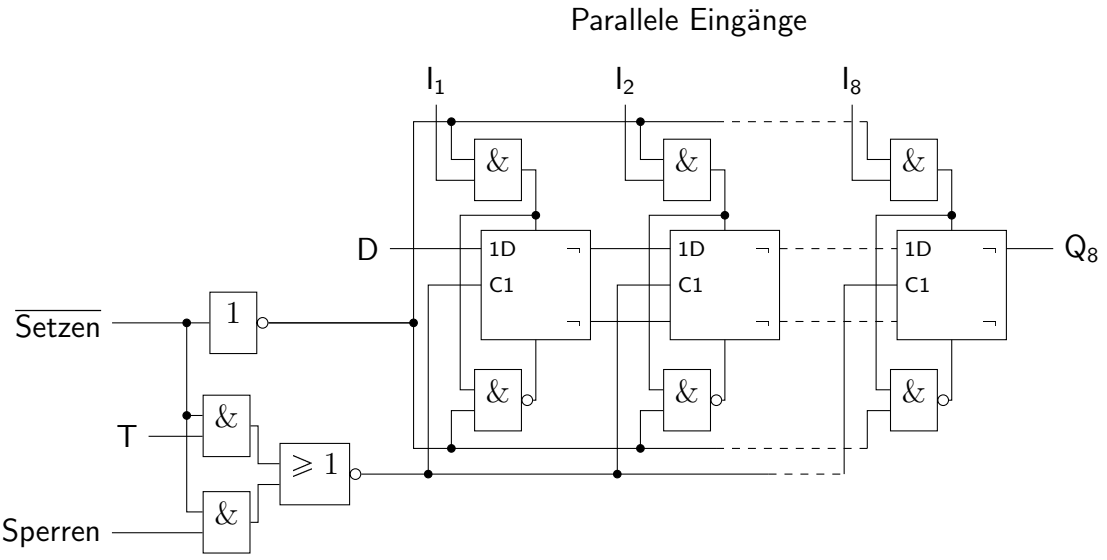


Abbildung 9.12: Schieberegister mit serieller Ein-/Ausgabe und paralleler Eingabe

die Schieberichtung nicht gewechselt werden, da sonst ein undefinierter Zustand auftritt. (siehe Abbildung 9.13)

9.4 Zähler und Untersetzer

9.4.1 Übersicht

Bei einem Zähler stellt jeder Zustand eine Zählerstellung dar. Eine einfache Sonderform der Zähler sind die Untersetzer (Frequenzteiler). Ein Untersetzter bewirkt nur eine Abfrage eines vorgegebenen Teilerverhältnis und ist meist einfacher aufgebaut. Die gebräuchlichsten Zähler sind reine Dual-Code-(modulo 2)-Zähler und der BCD-Code-Zähler, bei dem sechs Zustände des Dualcodes übersprungen werden. Unterschieden werden die Systeme durch ihre jeweilige Betriebsart (asynchron oder synchron). Beim asynchronen Betrieb werden die Flipflops immer von dem in der Zählfolge vorangehenden Flipflop getaktet. Bei der synchronen Variante werden alle Flipflops eines Zählers durch eine gemeinsame Clock gleichzeitig getaktet.

9.4.2 Untersetzer

Asynchroner Untersetzer

Ein asynchroner Untersetzer kann z. B. durch eine Kaskadierung von D-Latches realisiert werden. Mit der Beschaltung $D = \overline{Q}$ wirken die n D-Latches als Untersetzer im Untersetzungsverhältnis $2^n : 1$. Der Q-Ausgang jedes Flipflops wird mit dem Takteingang des folgenden Flipflops verbunden. Die Verzögerungszeiten t_{pd} addieren sich mit jeder zusätzlichen Untersetzerstufe (Nachteil). Bei Untersetzern mit einem geringeren

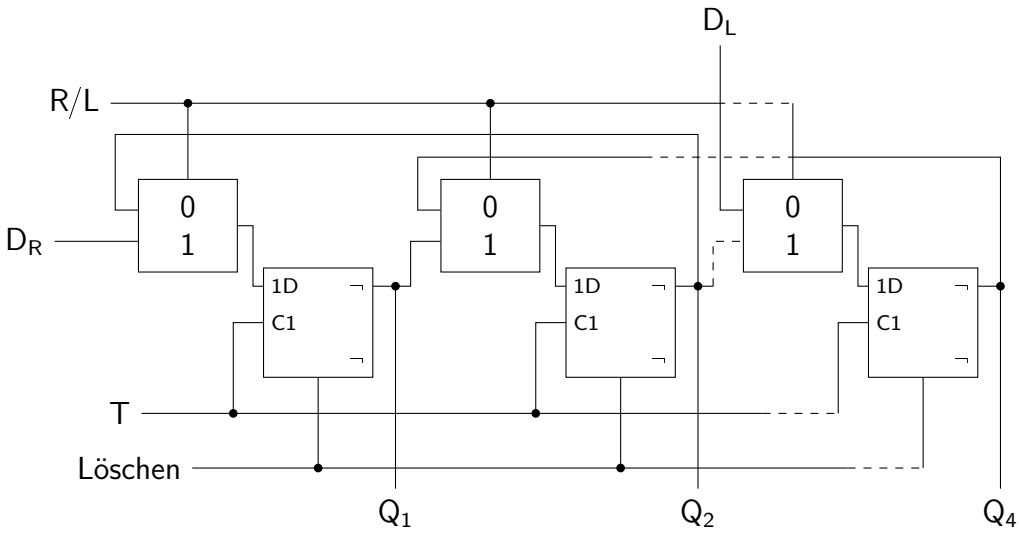
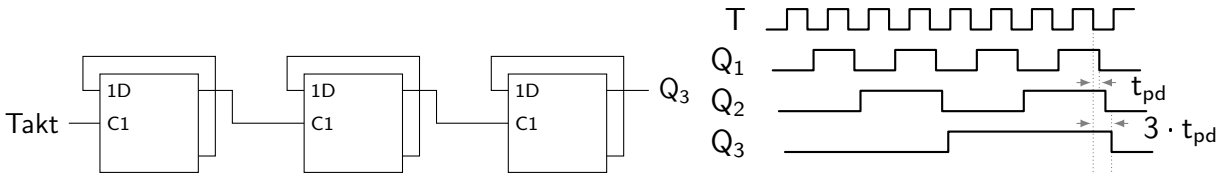


Abbildung 9.13: Schieberegister mit zwei Schieberichtungen



(a) Schaltung (b) Impulsdiagramm

Abbildung 9.14: Asynchroner Binäruntersetzer

Teilerverhältnis als $2^n : 1$ müssen durch Rückführungen entsprechend dem gewählten Teilerverhältnis Zählerzustände unterdrückt werden.

Synchroner Untersetzer

Wie für synchrone Systeme üblich werden alle Flipflops gleichzeitig vom gemeinsamen Takt gesetzt. Als Verzögerungszeit des gesamten Untersetzers tritt also nur die Laufzeit einer Stufe auf. Die Takteingänge können dadurch aber nicht zur Bestimmung des Teilerverhältnisses mitbenutzt werden. Damit ergibt sich ein höherer Schaltungsaufwand gegenüber den asynchronen Untersetzern.
In der Abbildung ist ein einfacher synchroner Untersetzer mit Schieberegister dargestellt. Die n Stufen ergeben ein Untersetzungsverhältnis von $n - 1$. Zusätzlich ist der Ausgang auf den Eingang zurückgeführt. Zu Beginn wird über das S-Signal das erste Flipflop gesetzt ($Q_1 = 1$) und die beiden anderen Flipflops zurückgesetzt. Danach wird mit jedem Taktimpuls die "1"jeweils um eine Stufe verschoben. Bei kreuzweiser Rückkopplung der Schieberegisterausgänge erhöht sich das Untersetzungsverhältnis auf $2n - 1$. Das Eingangsflipflop wird dann ebenfalls zu Beginn zurückgesetzt.

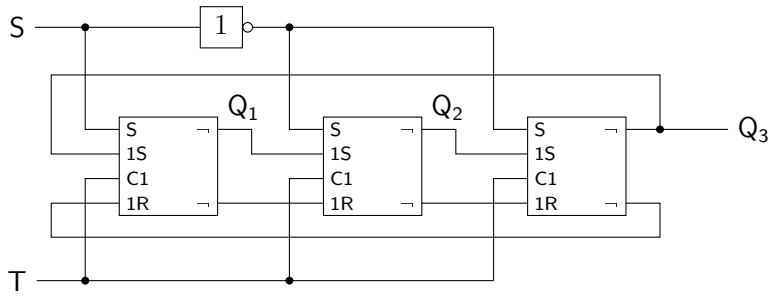


Abbildung 9.15: Synchroner Untersetzer

9.4.3 Zähler

Asynchroner Zähler

Asynchrone Zähler werden ähnlich wie asynchrone Untersetzer entworfen. Es werden entsprechend dem verwendeten Zählcode Zustände übersprungen. Die asynchronen Zähler weisen den gleichen Nachteil wie asynchrone Untersetzer auf. Trotzdem kommt dieser Zählertyp wegen seines geringen Aufwands zum Einsatz.Voraussetzung ist dabei aber, dass keine hohe Zählgeschwindigkeit erforderlich ist.

Beispiel 89. *BCD-Code-Zähler*

Zählerstand	Q_4	Q_3	Q_2	Q_1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

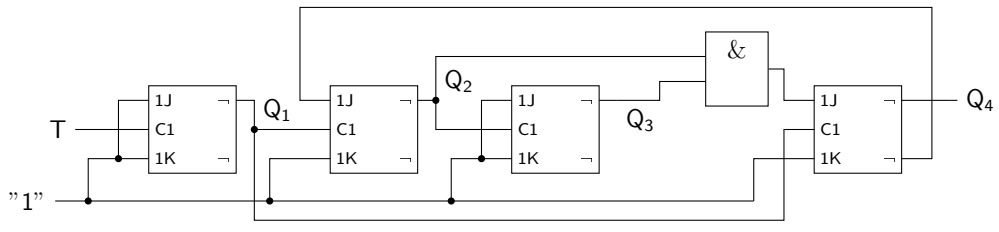
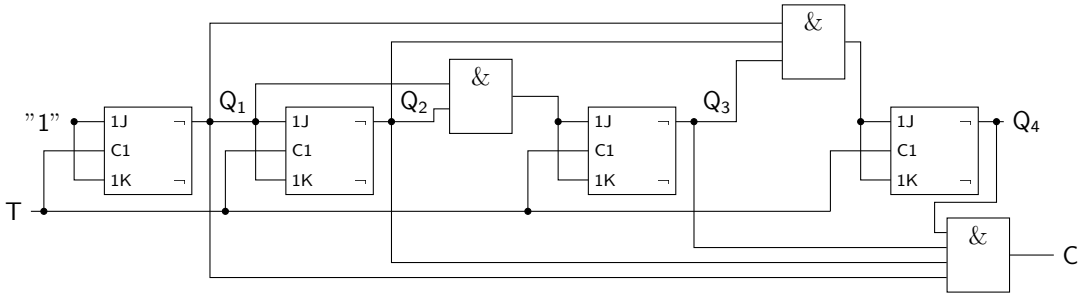


Abbildung 9.16: Asynchroner Zähler für den BCD-Code

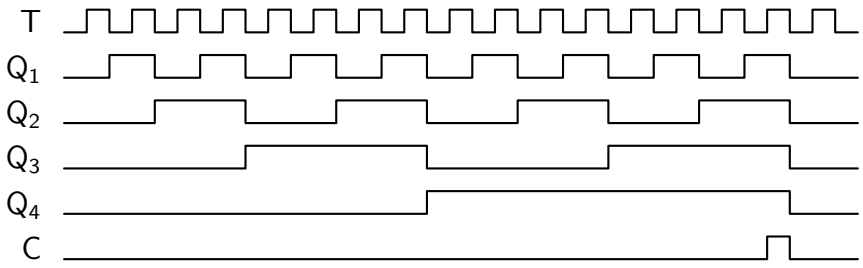
Synchroner Zähler

Die Zählfrequenz synchroner Zähler ist unabhängig von der Anzahl der Flipflops. Es addieren sich hierbei nicht die Flipfloplaufzeiten. Der große Nachteil ist wiederum der erhöhte Schaltungsaufwand. Die gebräuchlichsten Zähler arbeiten im reinen Dualcode oder im BCD-Code.

Beispiel 90. *Synchroner 4-Bit Vorwärtszähler im Dualcode*



(a) Schaltung



(b) Impulsdiagramm

Abbildung 9.17: Synchroner dualer 4-Bit-Vorwärtszähler

Die Schaltung 9.17 ist mit RS-Flipflops realisiert. Der Entwurf des Zählers erfolgt aus der Übergangstabelle, wobei die Zustände entsprechend dem Dualcode durchlaufen werden. Zur Kaskadierung mehrerer Zählerbausteine ist ein Übertragsausgang Ü vorhanden. Das Ü-Signal wird auf den Takteingang des nächsten Zählerbausteins geführt, so dass die Kaskadierung asynchron erfolgt.

Synchroner Rückwärtszähler

Die Vorwärtszählerschaltung kann in einen Rückwärtszähler abgeändert werden, wenn die Q-Ausgänge mit den $\overline{\text{Q}}$-Ausgängen vertauscht werden.

Beispiel 91. *Synchroner dualer 4-Bit Rückwärtszähler (siehe Abbidung 9.18)*

In integrierten Zählerbausteinen wird meistens die Vorwärts- und Rückwärtszählung durch eine ODER-Schaltung kombiniert. Die Zählrichtung kann dann über einen zusätzlichen Steuereingang bestimmt werden.

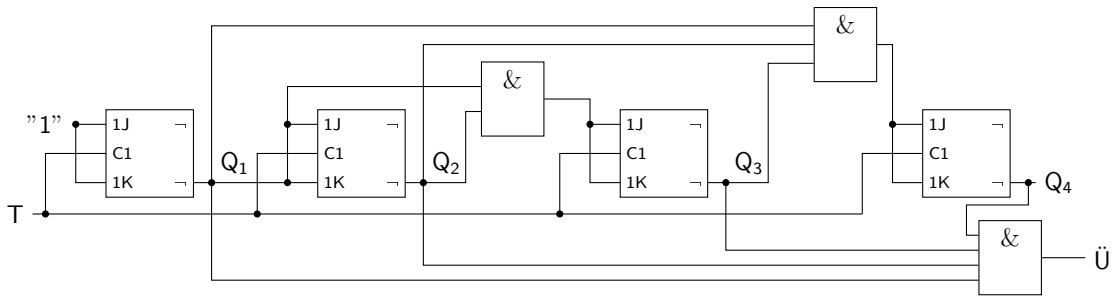


Abbildung 9.18: Synchroner dualer 4-Bit-Rückwärtszähler

10 Hardwarebeschreibungssprache VHDL

10.1 Motivation

Aktuelle Situation beim Entwurf elektronischer Systeme

- Integrationsgrad und Integrationsdichte steigen kontinuierlich
- Konkurrenzdruck und Kundenanforderungen bedingen kurze Entwicklungszeiten

Gründe für vollständige, widerspruchsfreie und verständliche Dokumentation

- Komplexität, Modularisierung
- Wiederverwendung von Entwurfsdaten und
- Wartung des fertigen Produktes

Kompatibilität der Entwurfsdaten

- Beschreibungsmittel müssen herstellerübergreifend normiert, rechnerunabhängig sein und mehrere Entwurfsebenen abdecken - Fixierung auf herstellerspezifische Beschreibungssprache kann hohes wirtschaftliches Risiko bedeuten

10.2 Entwurfssichten

Entwurfsfluss (Top-down-Entwurf) Der Entwurf komplexer elektronischer Systeme kann nur durch eine strukturierte Vorgehensweise beherrschbar gestaltet werden. Systemspezifikation wird die Schaltungsfunktion partitioniert (funktionale Dekomposition) und die Funktionen einzelnen Modulen zugeordnet. Schrittweise wird der Entwurf feiner strukturiert und zunehmend mit Implementierungsdetails versehen, bis die für die Fertigung notwendigen Daten vorliegen. Dieses sind z. B. Programmierdaten für Logikbausteine, Layouts für PCB‘s oder Maskenbänder für IC-Fertigung.

Sichtweisen beim Entwurf elektronischer Systeme

- Verhalten,
- Struktur und
- Geometrie.

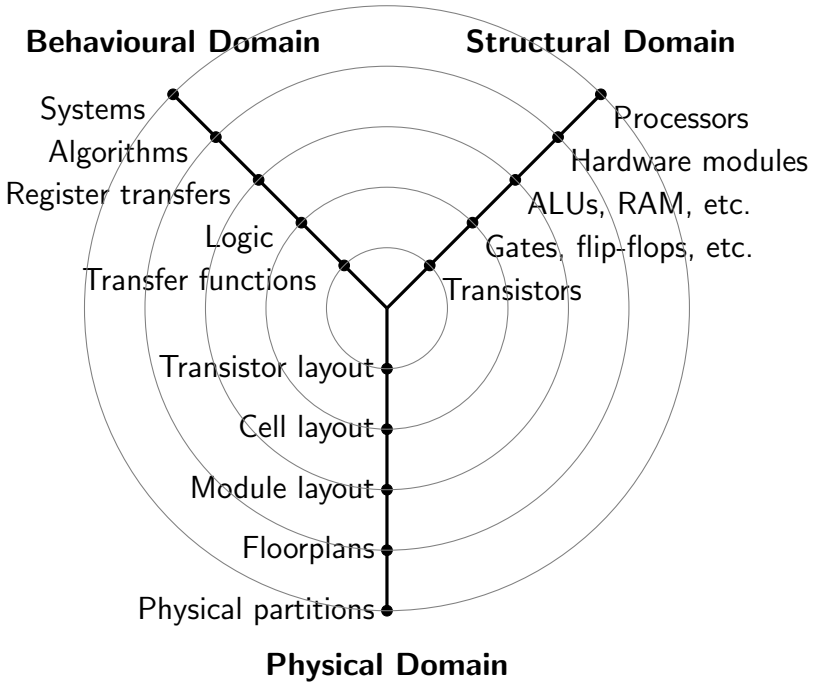


Abbildung 10.1: Gajski-Kuhn Y-Diagramm

Zu den drei Sichtweisen (Äste im Y-Diagramm) existieren verschiedene Abstraktionsebenen. Der Entwurf elektronischer Systeme kann als Reihe von Transformationen (Wechsel der Sichtweise) und Verfeinerungen (Wechsel der Abstraktionsebene innerhalb einer Sichtweise) im Y-Diagramm dargestellt werden.

10.3 Historische Entwicklung

1983 Start der VHDL-Entwicklung in den USA Arbeiten im Rahmen des VHSIC-Programms (Very High Speed Integrated Circuit). Ziel der Sprache: Austausch von Entwürfen zwischen Herstellern Anlehnung an Ada, da das Verteidigungsministerium diese Sprache in weitem Umfang einsetzt. 1987 Übernahme von VHDL als IEEE-Standard (IEEE 1076-1987). Dieser erste und bislang einzige IEEE-Standard für HDL definiert nur die Syntax und Semantik der Sprache, nicht jedoch ihre Anwendung bzw. einheitliches Vorgehen bei der Anwendung, insbesondere Synthese von Anfang 1992 bis Mitte 1993 Definition der Version IEEE 1076-1993 Dokumentation des neuen Standards (Language Reference

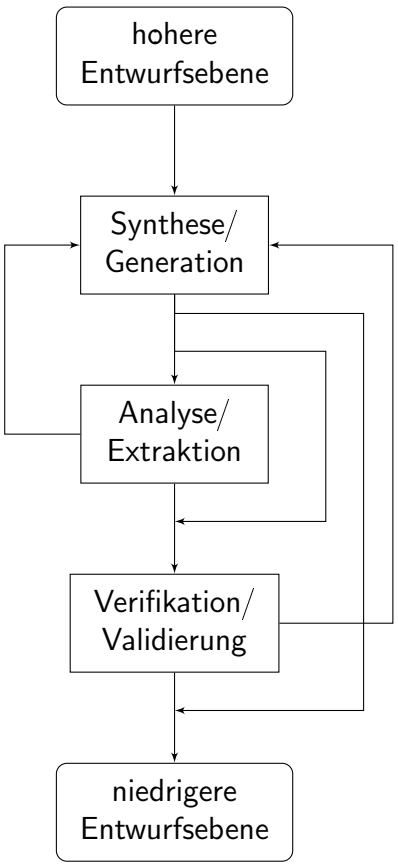


Abbildung 10.2: Prinzipieller Verlauf eines Entwurfsschrittes

Manual, LRM), wurde 1994 durch das IEEE herausgegeben neue Version enthält im wesentlichen kosmetische Änderungen (Systematisierung der Syntax), z.B. Einführung eines XNOR-Operators Seit Anfang 90er Jahre weltweiter Durchbruch für VHDL Heutige Konkurrenz von VHDL besteht vor allem in Verilog (USA) und UDL/I (Japan)

10.4 Aufbau einer VHDL-Beschreibung

10.4.1 Entity-Relationship-Modell

Datenmodelle zur Hardwarebeschreibung
Es gibt zwei unterschiedliche Datenmodelle. Zum einen das herkömmliche, strukturelle Datenmodell. In diesem gibt es hierarchische, netzartige und relationale Datenmodelle. Diese weisen je nach Anwendbarkeit als Hauptnachteil die ungenügende Modellierbarkeit komplexer Objekte auf. Semantischen Datenmodelle können zwar mächtige Datenstrukturen unterstützen. Allerdings lassen sich die Operationen oft nur unzulänglich definieren.

Allgemeines Grundkonzept des ER-Modells(semantisches Datenmodell
Man kann in diesem Modelltyp in zwei unterschiedliche Datentypen unterscheiden. Zum einen die Entitytypen (Objekttypen) mit denen man einzelne Objekte der realen Welt nachbildet. Diese sind im folgenden als Kästchen dargestellt. Zum anderen gibt es die Relationships, welcher in der Abbildung als Raute dargestellt ist. Diese beschreiben die Relation zwischen mehreren Entitäten.

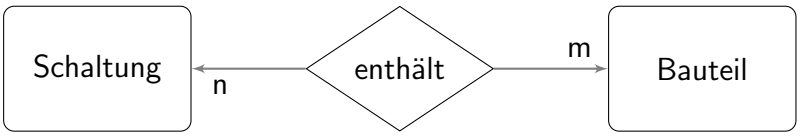


Abbildung 10.3: Einfaches ER-Diagramm.

Ein Beziehungstyp kann im ER-Modell über belieb vielen Entitäten definiert sein. Beim IC-Entwurf hat sich die Verwendung von Zellen als Grundstrukturmuster als zweckmäßig erwiesen. Bei der Beschreibung von Zellen unterschiedet man zwischen der Schnittstelle und dem Inhalt einer Zelle, die jeweils selbst wieder sehr komplexe Gebilde sein können.

10.4.2 Übersicht

Die wichtigsten Bestandteile eines VHDL-Modells ist die Instanz mit der Schnittstellenbeschreibung(entity), eine oder mehrere Architekturen(architecture), eine oder mehrere Konfigurationen(configuration), vordefinierte Funktionen, Prozeduren, Komponenten und Konstanten(package) und vordefinierte Bibliotheken(libraries).

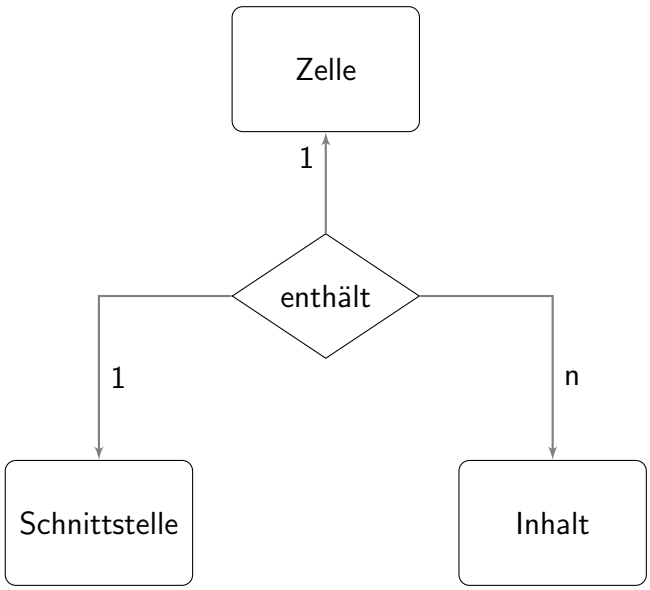


Abbildung 10.4: VHDL ER-Diagramm.

10.4.3 Schnittstellenbeschreibung

Schnittstelle der/des zu modellierenden Komponente/Systems Die Ein- und Ausgänge sowie deren Konstanten, Unterprogramme und sonstige Vereinbarungen, die auch für alle dieser Entity zugeordneten Architekturen gelten sollen.

```
0 entity entity_name is
1       [generic (generic_list)] [port (port_list);]
2       entity_declarative_part
3 [begin
4       passive_concurrent_statement]
5 end [entity] [entity_name] ;
```

Beispiel 92. *Schnittstellenbeschreibung (entity) eines OR-Gatters*

```
0 entity or2 is
1       port (in1,in2: in bit; out1: out bit);
2       -- definiere Pins als Signale vom Typ bit
3 end or2;
```

10.4.4 Architektur

Die Architektur - auch architecture genannt - ist eine Verhaltensbeschreibung oder kann einen strukturellen Charakter besitzen. Man kann mehrere dieser Funktionsbeschreibungen kombinieren. Somit kann man für eine Entity mehrere Architekturen deklarieren.

Dadurch können für eine Komponentenschnittstelle mehrere Beschreibungen auf unterschiedlichen Abstraktionsebenen oder verschiedene Entwurfsalternativen bestehen.

Syntax

```
architecture architecture_name of entity_name is
    architecture_declarative_part
begin
    all_concurrent_statements
end [architecture] [architecture_name];
```

Beispiel 93. *Architektur (architecture) eines OR-Gatters*

```
architecture or2_behavioral of or2 is
begin
    out1 <= in1 or in2; -- Verhaltensbeschreibung
end or2_behavioral;
```

10.4.5 Konfiguration

Eine Konfiguration(configuration) beschreibt welche Zuordnung für die möglicherweise verwendeten Submodule in der Architecture einer bestimmten Entity gelten sollen. Es können auch hierarchisch den untergeordneten Entities bestimmte Architekturen oder Parameterwerte zugeordnet werden.

Syntax

```
configuration configuration_name of entity_name is
    { use_clause. attribute_specification }
    { block-configuration }

    for component_specification
        [ use binding_indication ; ]
        [ block_configuration ]
    end for ;

end [configuration_name] ;
```

Beispiel 94. *Architektur (architecture) eines OR-Gatters*

```
configuration or2_config of or2 is
    for gate_level -- verknuepfe architecture gate_level
    end for;       -- mit entity or2
end or2_config;
```

10.4.6 Packages und Libraries

Packages fassen Typen oft gebrauchter Funktionen, Prozeduren, Komponenten oder Konstanten zusammen. Dazu zählen auch Anweisungen, wie Typ- oder Objektdeklarationen und Beschreibungen von Prozeduren und Funktionen, die in mehreren VHDL-Beschreibungen gebraucht werden. Es gibt vordefinierte Packages wie „standard“, welche den zweiwertigen Logiktyp „bit“ und häufig verwendete Funktionen und Typen beinhaltet.

Syntax

```
0 package name_of_the_package is
1       package_declarative_part
2 end [ package ] [ name_of_the_package ] ;
```

Bibliotheksobjekte werden in einer Library zusammengefasst. Die Bekanntgabe einer oder mehrerer dieser Bibiliotheken erfolgt durch das library-Tag. Dabei sind Bibliotheken wie std und work in VHDL allgemein bekannt.

Syntax

```
0 library library_name_1 {, library_name_i }; --Aufruf der Bibliothek
1
2 use library_name.all;
```

10.5 Entwurfssichten in VHDL

10.5.1 Verhaltensmodellierung

Das Verhalten einer Komponente wird durch die Reaktion der Ausgangssignale auf Änderungen der Eingangssignale beschrieben. Die Komponente verzweigt nicht weiter in Unterkomponenten.

Beispiel 95. *Verhaltensmodellierung eines Komparators*
Dieser Liefert eine boolsche „1“ wenn a größer ist als b. Außerdem lassen sich mit dem n beliebig breite Vektoren vergleichen.

```
0 entity compare is
1       generic (n : positive := 4);
2       port (a, b : in bit_vector (n-1 downto 0); result: out bit);
3 end compare;
4
5 architecture behavioral_1 of compare is
6 begin
7       process (a, b)
8       begin
9             if a > b then -- liefert 1 falls a > b
10                  result <= '1';
```

```
11            else
12                  result <= '0';
13            end if;
14      end process;
15 end behavioral_1;
```

Die beiden prinzipiellen Beschreibungsmittel in der Verhaltenssicht sind sequentielle(sequential) oder nebenläufige Anweisungen (concurrent statements). Bei den sequentiellen bzw. prozeduralen Beschreibungen werden Konstrukte wie Verzweigungen(if-then-else), Schleifen(loop) oder Unterprogrammaufrufe(function, procedure) verwendet.

Beispiel 96. *Modell eines Halbaddierers mit Schnittstelle und Architektur*
Anm.: Die Ergebnisse werden erst beim Verlassen des Prozesses sichtbar.

```
0 entity halfadder is
1       port (a, b : in bit; sum, carry : out bit);
2 end halfadder;
3
4 architecture behavioral_seq of halfadder is
5 begin
6       process (a, b)
7       begin
8             if(a = '1' and b = '1') then sum <= '0'; carry <= '1';
9             else
10                  if (a = '1' or b = '1') then sum <= '1'; carry <= '0';
11                  else sum <= '0'; carry <= '0';
12                  end if;
13            end if;
14      end process;
15 end behavioral_seq;
```

Bei den nebenläufigen Anweisungen ist es erlaubt parallel ablaufende Operationen zu beschreiben. Hiermit kann man spezifischer Eigenschaften von Hardware bzw. parallel arbeitenden Funktionseinheiten beschreiben.

Beispiel 97. *Architektur des Halbaddierers mit nebenläufigen Anweisungen*
Anm.: Die beiden Verknüpfungen XOR und AND können dabei gleichzeitig aktiv sein.

```
0 architecture behavioral_par of halfadder is
1 begin
2       sum <= a xor b;
3       carry <= a and b;
4 end behavioral_par;
```

10.5.2 strukturelle Modellierung

Das Wesen der strukturellen Modellierung besteht aus der Darstellung des inneren Aufbaus aus Unterkomponenten. Die Eigenschaften derUnterkomponenten werden in unabhängigen VHDL-Modellen beschrieben. Diese stehen kompiliert in Modellbibliotheken zur Verfügung. Eine eindeutige Zuordnung eines VHDL-Modells in eine der beiden Modellierungsarten ist oft schwierig. Daher ist es in VHDL erlaubt beide Beschreibungsarten innerhalb eines Modells zu benutzen.

Beispiel 98. *Halbaddierer, der aus einem XOR2- und einem AND2-Gatter aufgebaut ist*

```
0  architecture structural of halfadder is
1       component xor2
2            port (c1, c2 : in bit; c3 : out bit);
3       end component;
4
5       component and2
6            port (c4, c5 : in bit; c6 : out bit);
7       end component;
8
9  begin
10      xor_instance : xor2 port map (a, b, sum);
11      and_instance : and2 port map (a, b, carry);
12 end structural;
```

10.6 Entwurfsebenen in VHDL

10.6.1 Algorithmusebene

Die Schaltung im Beispiel soll immer dann, wenn sie von einem Controller eine Aufforderung erhält, eine Adresse aus einem internen Register nach frühestens 10ns auf den Bus legen. Dabei enthält die Beschreibung keine Angaben über die spätere Schaltungsstruktur und Takt- oder Rücksetzsignale.

Beispiel 99. *Beschreibung eines Schnittstellenbausteins auf Algorithmusebene*

```
0  architecture algorithmic_level of io_ctrl is
1  begin
2       ...
3       write_data_alg: process
4
5       begin
6            wait until adr_request = '1';
```

```
7             wait for 10 ns;
8             bus_adr <= int_adr;
9       end process write_data_alg;
10 end algorithmic_level;
```

10.6.2 Register-Transfer-Ebene

Im Unterschied zur Algorithmusebene wird in der Register-Transfer-Ebene ein zeitliches Ablaufschema der Operation vorgegeben und implizit eine Schaltungsstruktur beschrieben. Im Beispiel wird ein Taktsignal (clk) hinzugefügt und die Operation in Abhängigkeit dieses Signals beschrieben.

Beispiel 100. *Beschreibung eines Schnittstellenbausteins auf Algorithmusebene*

```
0 architecture rt_level of io_ctrl is
1 begin
2       ...
3       write_data_rtl : process (clk)
4             variable tmp : boolean;
5
6       begin
7             if (clk'event) and (clk = '1') then
8                   if((adr_request = '1') and (tmp = false))then tmp := true;
9                   elsif (tmp = true) then bus_adr <= int_adr; tmp := false;
10                  end if;
11            end if;
12      end process write_data_rtl;
13 end rt_level;
```

Wird bei aktiver Taktflanke ein gesetztes adr-request-Signal entdeckt, wird die temporäre Variable tmp gesetzt, damit bei nächster aktiver Taktflanke (Wartezeit!) die Adresse auf den Bus geschrieben werden kann. Eine geeignete Wahl der Taktperiode stellt die Wartezeit von mindestens 10ns sicher.

10.6.3 Logik(Gatter)ebene

Elektronische Systeme werden durch logische Verknüpfungen digitaler Signale und deren zeitlichen Eigenschaften (Verzögerungszeiten der Verknüpfungen) beschrieben. VHDL besitzt dazu vordefinierte Operatoren wie AND, OR, XOR, NOT, etc. für binäre Signale ('0', '1') und gestattet die Ergänzung durch benutzerdefinierte Operatoren. Die Konstrukte zur Modellierung zeitlicher Eigenschaften werden bereitgestellt.

Beispiel 101. *Halbaddierer-Architektur auf Logikebene in Verhaltenssichtweise*

```
0 architecture logic_1eve1 of halfadder is
1 begin
2       sum <= a xor b after 3 ns;
3       carry <= a and b after 2 ns;
4 end logic_level;
```

Die strukturelle Darstellung entspricht vorherigem Beispiel der Architektur structural. Die Verzögerungszeiten ergeben sich hier aus den internen Verzögerungszeiten der Subkomponenten.

10.7 Logiktypen

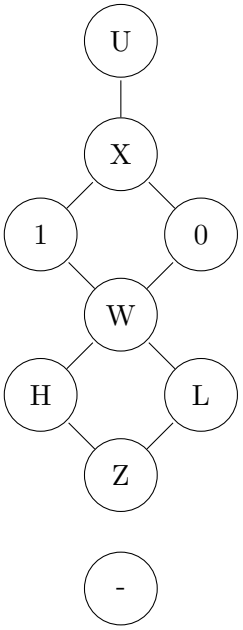
9-wertiges Logikwertesystem von VHDL `ieee.std_logic_1164` MVL9

Das Logikwertesystem ist nicht als Teil der Sprachdefinition definiert und kann an jeweilige Problemstellungen angepasst werden. Der Datenaustausch der VHDL-Modelle und der Einsatz von Synthesewerkzeugen erfordern aber eine Standardisierung. Für eine hardwarenahe Simulation wird häufig mit den 9 Zustand/Stärkewerten von MVL9 modelliert. Es beinhaltet eine Berücksichtigung der Signaltreiberstärke. Dies bedeutet, dass bei der Spannungsversorgung , Eingänge oder aktive Gatterausgänge stärkere Werte schwächere Stärken überschreiben. Dabei kann man folgende Festlegung der Signalwerte treffen. Es gibt zum einen die drei Signaltreiberstärken „stark“, „schwach“ und „hochohmig“. Dazu kommen die Logikpegel „0“, „1“ und „unbestimmt“. Darüber hinaus gibt es die zusätzlichen Werte „nicht initialisiert“ für die Simulation und ein „don't-care“ für Syntheseanwendungen.

Die Ausgänge der Gatter können mit einem Bus verbunden werden. Die Auflösung der zusammengeschalteten Signale an einem Bus erfolgt nach folgender Grafik und von links nach rechts in aufsteigender Reihenfolge.

Wert	deutsche Beschreibung	englische Beschreibung
U	nicht Initialisiert	uninitialized
X	stark unbestimmt	strong unknown
0	starker „0“-Pegel	strong low
1	starker „1“-Pegel	string high
Z	Hochohmig	tri-state
W	schwach unbestimmt	weak unknown
L	schwacher „0“-Pegel	weak low
H	schwacher „1“-Pegel	weak high
-	don’t-care	

(a) Übersicht über das Logikwertesystem



(b) Hasse Diagramm

Abbildung 10.5: 9-wertige Logik

VHDL-Konstrukte zur Laufzeitmodellierung

Bei der Laufzeitmodellierung in VHDL muss man verschiedene Delays mit einbauen damit die Simulation so realitätsnah wird wie irgend möglich. Das erste dieser Delays ist das „Transport-Delay“, bei dem jeder Impuls unabhängig von seiner Dauer verzögert am Ausgang erscheint. Das „Inertial-Delay“ hingegen gibt nur Impulse weiter deren Dauer größer als die angegebene Verzögerungszeit des angesteuerten Gatters sind. Es findet hier sowohl eine Impulsunterdrückung als auch eine Zeitverzögerung von gleicher länge statt. Im Gegensatz dazu wird bei dem „Reject-Inertial-Delay“ der Zeitbereich für die Impulsunterdrückung und die Zeitverzögerung separat angegeben.

```
0 entity nand2 is
1       port(a,b: in std_logic; o: out std_logic);
2 end nand2;
3
4 architecture several_delays of nand2 is
5 begin
6       --einfache Verzoegerung
7       o <= transport a nand b after 2 ns;
8
9       --Verzoegerung mit Impulsmindestloenge
10      o <= [inertial] a nand b after 2 ns;
11
12      --Verzoegerung mit seperater Impulsmindestlaenge
13      o <= reject 1 ns inertial a nand b after 2 ns;
14 end several_delays;
```

10.8 Testumgebung

Die VHDL-Simulation erfolgt in drei Phasen. Die Elaboration ist der Aufbau der Schaltung aus den compilierten VHDL-Modellen. Die Initialisierung beinhaltet die Signale, Variablen und Konstanten die Anfangswerte erhalten. Diese Werte ergeben sich aus expliziter Angabe durch die Typ-Spezifikation. Jeder Prozess wird einmal gestartet. Die Ausführung ist die Simulation bis zum Ende der spezifizierten Simulationsdauer. Zur Stimuli-Generierung und zur Ergebnisauswertung wird das zu testende Modell in eine Testbench eingebunden. Das Modell wird instanziiert und mit den Ein- und Ausgangssignalen verdrahtet.

Stichwortverzeichnis