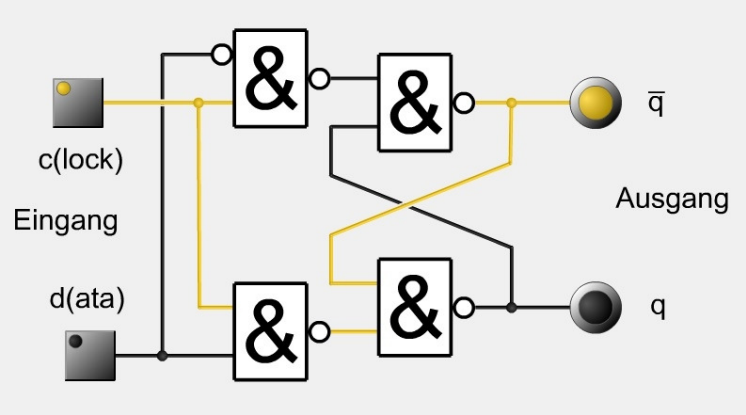
Zahlensysteme
und
Logische Schaltungen



Begleitmaterial für den Informatikunterricht

Differenzierungskurs Informatik

2014

Dipl.-Inform. Klaus Milzner

www.milzners.de

Inhaltsverzeichnis

✶ : für Fortgeschrittene / optional

Übersicht

Dieser Themenblock vermittelt die Grundlagen der digitalen Datenverarbeitung. Dabei stehen zunächst die wesentlichen Zahlensysteme in Verbindung mit den Grundrechenarten Addition und Subtraktion mittels Zweierkomplement im Vordergrund. In der Folge werden anhand von Wahrheitstafeln Schaltnetze aus logischen Grundbausteinen (UND, ODER, NICHT) entworfen. Hierzu gehören, aufbauend auf dem ersten Kapitel, insbesondere Additionsschaltungen. Abschließend werden Schaltwerke in Form von Elementen mit Speicherverhalten betrachtet (FlipFlops).

1 Zahlensysteme

1.1 Dezimalzahlen

Im *Dezimalsystem*, das jeder von uns täglich verwendet, werden *Zahlen* aus den 10 *Ziffern*

0, 1, 2, 3, …, 9

gebildet. Eine mehrstellige Zahl, hier als Beispiel die Zahl 358, setzt sich dabei *stellenweise* wie folgt zusammen: $358 = 3 * 100 + 5 * 10 + 8 * 1$

Alternativ kann man dies auch unter Verwendung der *Basis* des Zahlensystems in Exponentialdarstellung schreiben. Die Basis des Dezimalsystems ist, wie der Name „Dezimal“ bereits sagt, die Zahl 10. Damit ergibt sich: $358 = 3 * 10^2 + 5 * 10^1 + 8 * 10^0$

Stelle	10^4	10^3	10^2	10^1	10^0
Wert	10000	1000	100	10	1

Addiert man zwei *n-stellige Zahlen*, wie z.B. die beiden zweistelligen Zahlen 84 und 19, kann sich als Ergebnis eine *(n+1)-stellige Zahl* ergeben. In diesem Beispiel ergibt die Summe ein dreistelliges Ergebnis: $84 + 19 = 103$.

Bei der Addition ist in jeder Stelle ein möglicher *Übertrag* aus der nächstkleineren Stelle zu berücksichtigen. In obigem Beispiel ergibt sich ein Übertrag aus der Einerstelle ($4 + 9 = 13$) in die Zehnerstelle und als Folge auch in die Hunderterstelle.

1.2 Dualzahlen

Ein Rechner kann, wie *jedes* digitale System, nicht *direkt* im Dezimalsystem rechnen. Der Grund hierfür ist, dass seine internen Bausteine nur zwei unterschiedliche Zustände unterscheiden können. Dies kann man sich stark vereinfacht wie eine Ansammlung von miteinander verbundenen Schaltern vorstellen, die sich gegenseitig steuern und jeweils entweder AN oder AUS sein können. Hierdurch ergeben sich je Schalter die beiden Möglichkeiten *Schalter bzw. Strom aus* und *Schalter bzw. Strom an*. Diese beiden möglichen Zustände werden auch durch die beiden Ziffern 0 und 1 gekennzeichnet.

Ein Computer kann daher Zahlen (und auch alle anderen Informationen !) nur auf der Grundlage der beiden Ziffern 0 und 1 darstellen. Damit ergibt sich als *Basis* des Zahlensystems die Zahl 2 und jede Stelle einer mehrstelligen Zahl als eine Potenz zu dieser Basis.

Dieses System wird *Dualsystem*, die Zahlen *Dualzahlen* genannt[1].

Abgesehen von der anderen Basis ist das Dualsystem damit ein *Stellenwertsystem* wie das uns vertraute Dezimalsystem. Die folgende Tabelle veranschaulicht die ersten acht Stellen des Dualsystems und ihre jeweilige Wertigkeit als Dezimalzahl:

Stelle	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Wert	128	64	32	16	8	4	2	1

[1] In der Literatur findet man auch die äquivalenten Begriffe *Binärsystem* bzw. *Binärzahlen*.

Jede Stelle wird dabei als ein *Bit* bezeichnet (Bit 1, Bit 2, …). Ein *Bit* ist die kleinste Informationseinheit und kann den Wert 0 oder 1 annehmen. Acht Bit bilden jeweils ein *Byte*. Mit einem *Byte* können daher Dezimalzahlen im Bereich von 0_{10} (00000000_2) bis 255_{10} (11111111_2) dargestellt werden. Die Indizes an den Werten weisen jeweils auf die Basis des Zahlensystems hin.

Wir werden uns im folgenden aus Gründen der Übersichtlichkeit weitgehend auf maximal acht Bit breite Zahlen beschränken.

Beispiel: Die Dezimalzahl 167_{10} als Dualzahl geschrieben lautet 10100111_2 , da

$$1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 167_{10}$$

oder in Kurzform

$$128 + 32 + 4 + 2 + 1 = 167_{10}$$

Entsprechend den Dezimalzahlen (s.o.) ergibt sich der Wert einer mehrstelligen Dualzahl damit auch als Summe der einzelnen Stellenwerte.

Folgende Tabelle gibt eine Übersicht über die ersten 30 Dualzahlen:

Dezimal	Dual
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001

Dezimal	Dual
10	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
16	00010000
17	00010001
18	00010010
19	00010011

Dezimal	Dual
20	00010100
21	00010101
22	00010110
23	00010111
24	00011000
25	00011001
26	00011010
27	00011011
28	00011100
29	00011101

Aufgabe: Wandle die Dualzahlen 10111111_2 und 11110000_2 in Dezimalzahlen.

1.2.1 Umwandlung von Dezimal- in Dualzahlen

Um eine Dezimalzahl in eine entsprechende Dualzahl umzuwandeln, kann man schrittweise möglichst große Potenzen zur Basis 2 von der Zahl subtrahieren, die entsprechende Stelle der Dualzahl auf 1 setzen und dann mit dem Rest jeweils genauso verfahren bis dieser Null wird.

Oder man verwendet den folgenden einfachen *Algorithmus* (Rechenvorschrift):

Die Dezimalzahl fortlaufend durch zwei teilen. Die jeweiligen Reste ergeben die Dualstellen, und zwar erhält man die Dualstellen von den niederwertigen zu den höherwertigen.

Beispiel: Dezimalzahl 53_{10} = 110101_2 Dualzahl:

```
53 : 2 = 26  Rest 1
26 : 2 = 13  Rest 0
13 : 2 =  6  Rest 1
 6 : 2 =  3  Rest 0
 3 : 2 =  1  Rest 1
 1 : 2 =  0  Rest 1
```

niederwertigstes Bit

höchstwertigstes Bit

Aufgabe: Wandle die Dezimalzahlen 78_{10} und 250_{10} in Dualzahlen.

Tipp: Um Zahlendarstellungen ineinander umzurechnen oder seine Rechenergebnisse zu überprüfen ist der *Windows-Rechner* ein gutes Hilfsmittel. Dieser befindet sich im Ordner *Start→Programme→Zubehör*.
Nach dem Start des Rechners in die *Ansicht→Programmierer* wechseln. Hier kann direkt zwischen den unterschiedlichen Zahlendarstellungen *Hex*(adezimal), *Dez*(imal), *Okt*(al) und *Bin*(är) bzw. Dual umgeschaltet werden.

1.2.2 Addition von Dualzahlen

Dualzahlen werden wie Dezimalzahlen stellenweise addiert. Ergibt eine Stellensumme eine Zahl größer als 1 erhalten wir einen Übertrag in die nächste Stelle:

```
0 + 0       =  0
0 + 1       =  1
1 + 0       =  1
1 + 1       = 10  =  0 + Übertrag in die nächste Stelle
1 + 0 + 1   = 10  =  0 + Übertrag in die nächste Stelle
1 + 1 + 1   = 11  =  1 + Übertrag in die nächste Stelle
        ↥ Übertrag aus der vorhergehenden Stelle
```

Beispiele:	allgemein:	dezimal:	dual:
	drei	3	011
	plus vier	+ 4	+ 100
	sieben	7	111
	elf	11	1011
	plus drei	+ 3	+ 0011
	vierzehn	14	1110
	siebenundzwanzig	27	00011011
	plus einundvierzig	+ 41	+ 00101001
	achtundsechzig	68	01000100

Aufgabe: Berechne

101010	101010	10101
+ 10101	+101010	+ 1011
?	?	?

1.2.3 Subtraktion von Dualzahlen mittels Zweierkomplement

Die Subtraktion einer positiven Zahl von einer anderen lässt sich auf die *Addition der (negativen) Gegenzahl* zurückführen. Dies bringt in Verbindung mit einem Rechner einen großen Vorteil: Der Rechner muss nur addieren können ! Da Rechenoperationen wie die Addition rechnerintern nicht durch Software, sondern durch spezielle Hardware in der *ALU* (*Arithmetic Logic Unit*) realisiert werden, verringert dies den Schaltungsaufwand auf einer begrenzten und teuren Chipfläche[2].

Es muss also ein geeigneter Weg gefunden werden, um eine positive Zahl in ihre Gegenzahl umzuwandeln. Hierbei sollen zu den einzelnen Bits der Zahl keine zusätzlich Symbole erforderlich sein (z.B. + und -), sowie alle Zahlen eindeutig sein (z.B. nur eine Darstellung für die Null).

Die gebräuchlichste Kodierung, die dies erfüllt, ist das so genannte *Zweierkomplement*.

[2] Aus dem gleichen Grund wird auch die Multiplikation / Division in wesentlichen Schritten auf die Addition zurückgeführt („schieben und addieren“).

In der Zweierkomplement-Darstellung einer Zahl wird das Vorzeichen der Zahl durch das höchstwertige Bit (das Bit ganz links…) der Zahl kodiert:

- eine 0 steht für eine positive Zahl,
- eine 1 für eine negative Zahl.

Hierdurch stehen für den möglichen Zahlenbereich einer n Bit langen Zahl nur noch (n-1) Bits zur Verfügung, sodass dieser sich halbiert (bei einer 8-Bit Zahl von 0 … 255 auf –128 … +127).

Positive Dualzahlen sind dabei in ihrer Darstellung unverändert. Um eine positive Dualzahl in ihre Gegenzahl bzw. ihr (negatives) Zweierkomplement umzuwandeln sind folgende zwei Schritte erforderlich:

1) Alle Stellen der Zahl *invertieren*, d.h. alle Einsen in Nullen sowie alle Nullen in Einsen ändern.
2) Zu der hieraus entstandenen Zahl *1 addieren*.

Beispiele:

			invertieren	*1 addieren*	*Ergebnis*	
$6_{10} = 0110_2$	$\Rightarrow$	$-6_{10} =$	$1001_2 +$	$0001_2 =$	1010_z	(4-stellig)
$= 00000110_2$	$\Rightarrow$	$= 11111001_2 +$	$00000001_2 =$		11111010_z	(8-stellig)

Subtraktion durch Addition der Gegenzahl:

dezimal:	dual:	Anmerkungen:
6	0110	Durch Abschneiden des höchsten Bits (Übertrag
+ (–6)	+ 1010	in die 5. Stelle, in Klammern) ergibt sich der
0	(1)0000	richtige Ergebniswert: 0000_2.
11	00001011	
+ (–6)	+ 11111010	
5	(1)00000101	Übertrag in die 9. Stelle s.o.
11	00001011	26 = 00011010
+ (–26)	+11100110	–26 = 11100110
–15	11110001	

Tipp: Ein schneller Weg für die Umwandlung einer positiven in eine negative Zahl von Hand ist das folgende Vorgehen:
Von rechts, dem niederwertigsten Bit, angefangen nach links alle Nullen inklusive der ersten Eins abschreiben, danach alle folgenden Stellen invertieren.

Um eine negative Dualzahl im Zweierkomplement (d.h. das höchstwertige Bit ist 1) in eine Dezimalzahl umzuwandeln, ist obiges Verfahren in umgekehrter Reihenfolge anzuwenden:

1) Von der Zweierkomplementzahl *1 subtrahieren* (durch Addition von –1).
2) Alle Stellen des Ergebnisses *invertieren* und das Resultat *in eine Dezimaldarstellung bringen*.
3) *Minuszeichen* der Dezimalzahl ergänzen.

Beispiel: $11110010_z \Rightarrow$

1) Eins subtrahieren:
$$\begin{array}{r} 11110010_z \\ +\ 11111111_z \\ \hline (1)11110001 \end{array}$$
2) invertieren und in Dezimalschreibweise umrechnen: $00001110_2 = 14_{10}$
3) Minuszeichen der Dezimalzahl ergänzen: $11110001_z = -14_{10}$

Wichtig:
Da bei der Zweierkomplement-Darstellung das Vorzeichenbit immer an der gleichen Position stehen muss, sollten bei der Addition alle verwendeten Zahlen in *einheitlicher Bitlänge* vorliegen. Falls erforderlich, können hierzu die fehlenden höherwertigen Stellen der kürzeren Zahl mit Nullen (positive Zahlen) oder Einsen (negative Zahlen) entsprechend aufgefüllt werden.

Beim Rechnen mit einer festen, vorgegebenen Stellenanzahl kann es, wie wir es eingangs bei den Dezimalzahlen gesehen haben, zu einer *Zahlenbereichsüberschreitung* kommen. Dies wird auch als *Überlauf* bezeichnet. So kann z.B. die Addition 123 + 127 = 250 mit einer 8 Bit breiten Dualzahl im Zweierkomplement nicht mehr dargestellt werden, da der mögliche Zahlenbereich von –128 … +127 überschritten wird. Diese Problematik wird an späterer Stelle in Verbindung mit Additionsschaltungen näher untersucht.

Aufgabe: 1) Stelle im Zweierkomplement dar: –54; –127; –128

2) Berechne im Zweierkomplement und überprüfe das Ergebnis durch Rückwandlung in die Dezimaldarstellung: 45 – 22; 81 – 107

1.3 Hexadezimalzahlen

Dualzahlen erreichen in der Informatik schnell eine Größe, die für den Menschen schlecht handhabbar ist. Für die Adressierung einer 1GB-Festplatte sind z. B. 30-stellige Dualzahlen erforderlich:
1GB = 2^{30} Byte = 1.073.741.824 Byte ≈ 1 Milliarde Byte

Hier ist es notwendig, eine kürzere und besser lesbare Zahlendarstellung zu verwenden. Dies lässt sich mittels Hexadezimalzahlen erreichen. Hierbei handelt es sich wiederum um ein Stellenwertsystem, diesmal zur Basis 16. Man fasst dabei beginnend mit den niederwertigen Stellen einer Dualzahl jeweils vier Stellen zu einer Hexadezimalstelle zusammen (da $2^4 = 16$). Dadurch ist die Hexadezimalzahl nur 1/4 so lang wie die entsprechende Dualzahl.

Das Hexadezimalsystem benötigt entsprechend der Basis 16 Ziffern:

Dezimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadezimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Beispiel: $10\,1001\,0111\,0000\,1011\,1010_2$ = 2 9 7 0 B A_{16}

Die ersten vier Stellen haben dabei folgende Wertigkeiten je Ziffer:

Stelle	16^3	16^2	16^1	16^0
Wert	4096_{10}	256_{10}	16_{10}	1_{10}

Ein bekanntes Beispiel für die Verwendung von Hexadezimalzahlen ist die RGB-Kodierung für 24Bit-Farben in HTML, die durch einen sechsstelligen Code festgelegt wird.

Aufgabe: 1) Wandle die Dualzahl $101100010110001111010 0_2$ in eine Hexadezimalzahl.

2) Wandle die Hexadezimalzahlzahl $A0B1C_{16}$ in eine Dualzahl.

3) Wandle die Hexadezimalzahlzahl $CAFE_{16}$ in eine Dezimalzahl.

2 Logische Schaltungen

Wir haben in Verbindung mit Dualzahlen gesehen, dass digitale Rechenanlagen nur mit den beiden Zuständen 0 und 1 rechnen und sämtliche Informationsdarstellungen hierauf aufbauen (vereinfachtes Schaltermodell). Im folgenden werden wir uns nun damit beschäftigen, wie diese digitalen Informationen in Form von Bits in Rechenanlagen verarbeitet werden. Jedes Bit entspricht dabei einer elektrischen Leitung bzw. den daran vorhandenen Anschlüssen innerhalb eines Rechners.

Ausgehend von den logischen Grundschaltungen werden wir Schaltnetze und insbesondere Additionsschaltungen entwerfen, wie sie auch in Rechnern zum Einsatz kommen. Anschließend werden Schaltwerke in Form von Elementen mit Speicherverhalten betrachtet. Zum einfacheren Konstruieren, Testen und Visualisieren wird dabei jeweils eine spezielle Entwurfs- und Simulationssoftware namens *LogiFlash* eingesetzt.

2.1 Logische Grundfunktionen

Mit Hilfe logischer Grundfunktionen werden die grundlegenden Verknüpfungen von einzelnen Bits bzw. deren Zuständen beschrieben. Jedes Bit wird dabei durch eine *logische Variable*, die jeweils die Werte Null oder Eins annehmen kann, dargestellt. Aus wenigen Grundfunktionen können sämtliche komplexeren Strukturen eines Rechners bis hin zum Prozessor (engl. *CPU: Central Processing Unit*) und der Recheneinheit (engl. *ALU: Arithmetic Logic Unit*) ähnlich LEGO-Bausteinen zusammengesetzt werden.

Jede logische (Grund-)Funktion wird durch

1) eine *Funktionstabelle* (auch *Wahrheitstafel* genannt), in der die logischen Eingangsvariablen in *allen* möglichen Zustandskombinationen sowie die Ausgangsvariable als das jeweilige Verknüpfungsergebnis aufgelistet sind,
2) eine aus der Funktionstabelle abgeleitete mathematische Übertragungsfunktion in Verbindung mit einem spezifischen Operator- bzw. Rechenzeichen sowie
3) ein graphisches Schaltsymbol, das die jeweilige Funktion als Hardwarekomponente mit ihren Ein- und Ausgangsanschlüssen repräsentiert, und als Baustein für den Entwurf komplexerer Schaltungsstrukturen dient

beschrieben.

In der Literatur werden logische Grundfunktionen oft auch als *Logikgatter* bezeichnet.

2.1.1 Die ODER/OR-Funktion

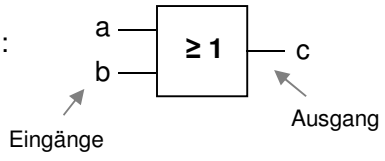
Die ODER-Verknüpfung (engl. *OR*) von zwei (oder mehr) logischen Variablen führt immer dann zu einer wahren Aussage und entsprechend zu einer Eins als Funktionsergebnis, wenn *mindestens eine* der Variablen den Wert Eins hat:

Funktionstabelle:

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

Übertragungsfunktion: $c = a \vee b$ („c gleich a oder b“)

Schaltsymbol:



Die ODER-Verknüpfung wird in der Literatur auch als *Disjunktion* bezeichnet.

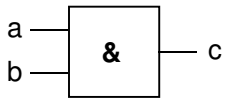
2.1.2 Die UND/AND- Funktion

Die UND-Verknüpfung (engl. *AND*) von zwei (oder mehr) logischen Variablen führt immer dann zu einer wahren Aussage und entsprechend zu einer Eins als Funktionsergebnis, wenn *alle* Variablen den Wert Eins haben.

Funktionstabelle: Übertragungsfunktion: $c = a \wedge b$ („c gleich a und b“)

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

Schaltsymbol:



Die UND-Verknüpfung wird in der Literatur auch als *Konjunktion* bezeichnet.

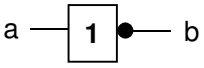
2.1.3 Die Negation

Durch eine Negation (engl. *NOT*) wird ein Bit *invertiert*, d.h. aus einer Eins wird Null und umgekehrt.

Funktionstabelle: Übertragungsfunktion: $b = \overline{a}$ („b gleich nicht a“)

a	b
0	1
1	0

Schaltsymbol:



Statt des oben gezeigten Schaltsymbols wird häufig nur der runde Punkt an einem Ein- oder Ausgang eines Gattersymbols verwendet, um eine Negation anzuzeigen. Dies ist z.B. bei mit *LogiFlash* (s.u.) gezeichneten Schaltplänen der Fall.

2.2 Abgeleitete Grundfunktionen

Neben den Grundfunktionen AND, OR und NOT gibt es hiervon abgeleitete Funktionen, die häufig Verwendung finden.

2.2.1 NAND (Nicht-UND) und NOR (Nicht-Oder)

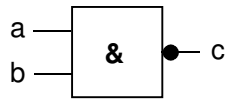
NAND- und *NOR*-Gatter setzen sich jeweils aus einem AND bzw. OR mit einer nachgeschalteten Negation zusammen (NOT AND bzw. NOT OR).

NAND Funktionstabelle: Übertragungsfunktion: $c = \overline{a \wedge b}$ („c gleich nicht (a und b)“)

$= \overline{a} \vee \overline{b}$ („c gleich nicht a oder nicht b“)

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

NAND Schaltsymbol:



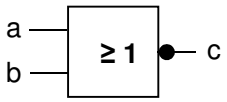
NOR Funktionstabelle:

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

Übertragungsfunktion: $c = \overline{a \vee b}$ („c gleich nicht (a oder b)“)

$= \overline{a} \wedge \overline{b}$ („c gleich nicht a und nicht b“)

NOR Schaltsymbol:



2.2.2 EXOR (Exklusiv-ODER)

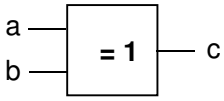
EXOR Funktionstabelle:

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

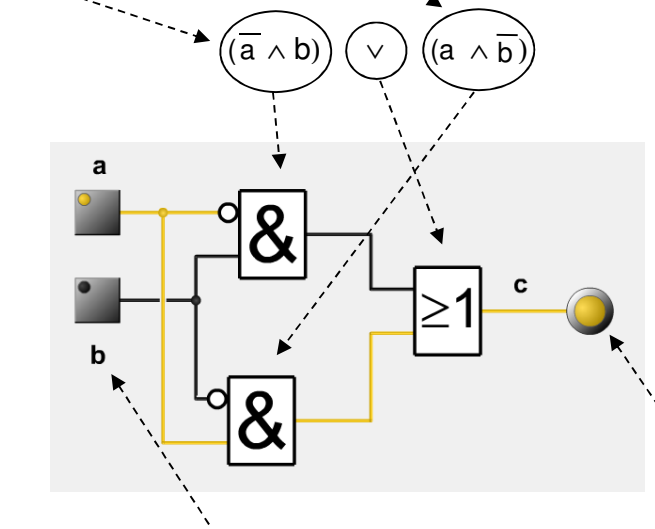
Übertragungsfunktion: $c = (\overline{a} \wedge b) \vee (a \wedge \overline{b})$

$= a \veebar b$ („c gleich entweder a oder b“)

EXOR Schaltsymbol:



Das EXOR-Gatter wird auch als Exklusiv-ODER bezeichnet.



Die Übertragungsfunktion eines EXOR zeigt, dass dieses durch die Verknüpfung von fünf Grundfunktionen (inkl. Negationen) realisiert werden kann.

Im Bild links ist der entsprechende, aus der Übertragungsfunktion hergeleitete *Schemaplan* dargestellt (erstellt mit *LogiFlash*, s.u.).

Jede Verknüpfung in der Übertragungsfunktion wird dabei in ein entsprechendes Logikgatter im Schemaplan abgebildet. Ein- und Ausgänge der Gatter werden durch Leitungen geeignet verbunden, um den gewünschten Signalfluss herzustellen.

Der Wert der Ausgangsvariablen wird durch eine Lampe visualisiert: AUS = 0; EIN = 1

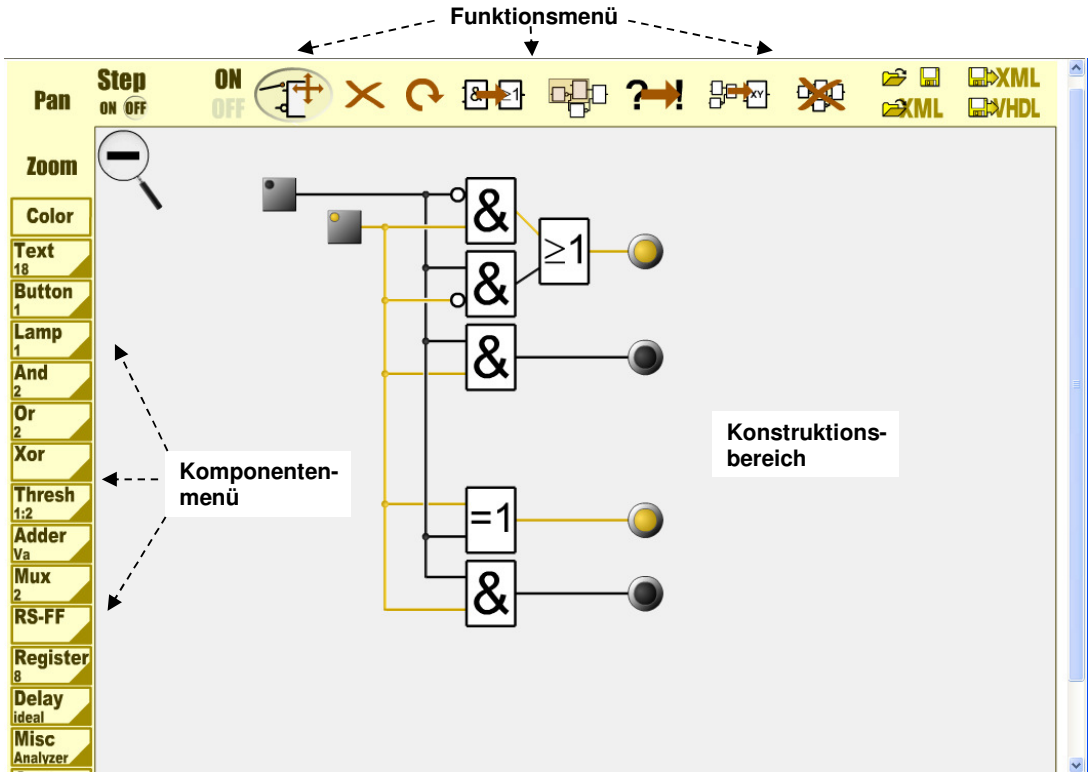
Mittels Schaltern können die Eingangsvariablen realisiert werden: 0 = AUS; 1 = EIN

2.3 LogiFlash – Ein Simulator für Digitalschaltungen

LogiFlash ist eine freie CAD-Software, die das Erstellen von Schemaplänen logischer Schaltungen und die Simulation ihrer Funktionsweise auf der Basis zweiwertiger Logik ermöglicht.

Hierdurch kann die Arbeitsweise dieser Schaltungen interaktiv und anschaulich nachvollzogen werden. Durch die im Simulationsmodus farbige Animation von Signalflüssen in Form von wechselnden logischen Zuständen an Ein- und Ausgängen aber auch der inneren Verschaltung bzw. den Leitungen wird die Schaltung sozusagen „lebendig“ (s.u.). Neben einer zeitdiskreten Steuerung ist dabei auch ein Einzelschrittmodus („Step“) verfügbar, in dem Details des Schaltungsverhaltens sozusagen in Zeitlupe betrachtet und analysiert werden können.

LogiFlash wurde vom Institut für Informatik der Johann-Wolfgang-Goethe-Universität in Frankfurt am Main für Ausbildungszwecke entwickelt und ist aufgrund seiner einfachen Handhabung auch für den Schulunterricht gut geeignet.



Die *LogiFlash*-Benutzeroberfläche

Obige Bildschirmansicht veranschaulicht die LogiFlash-Benutzeroberfläche. Am oberen Rand des Konstruktionsbereiches, in dem der Schaltplan ähnlich einem objektorientierten Zeichenprogramm konstruiert werden kann, befindet sich das Funktionsmenü (Simulation starten/stoppen, Komponenten erzeugen / verdrahten / löschen / drehen / ändern / etc.). Aus dem Komponentenmenü am linken Rand des Konstruktionsbereiches können sowohl die jeweils benötigte Art der logischen Schaltung (AND, OR, EXOR, Addierer, etc. in vielen Varianten) wie auch ergänzende Komponenten wie z.B. Schalter, Lampen und Anzeigen, Texte für das Beschriften von Komponenten, etc. ausgewählt werden. Ein Klick in den Konstruktionsbereich erzeugt und platziert die ausgewählte Komponente.

Eine weitergehende Anwendungsanleitung sprengt den Rahmen dieses Kapitels. Hier muss auf den Unterricht und / oder die zum Selbststudium geeignete Programmdokumentation verwiesen werden. Letztere besteht aus einer übersichtlichen Kurzanleitung für das Funktionsmenü (Datei *LogiFlash-Anleitung.htm* im LogiFlash-Verzeichnis) ergänzt um komponentenbezogene Beschreibungen in Verbindung mit Bespielen im Unterverzeichnis *Demo*.

Installation / Start:

Technisch gesehen handelt es sich um eine Macromedia Flash-Anwendung, die in jedem Internet-Browser mit dem entsprechenden Plugin lauffähig ist. Eine Installation im eigentlichen Sinne ist daher nicht notwendig.

Es gibt jedoch eine Randbedingung, die aus der Realisierung als Flash- bzw. ActiveX-Anwendung resultiert und die besondere Beachtung erfordert:

> *LogiFlash* kann Schaltungen bzw. Entwürfe *nur* aus dem Unterverzeichnis *circuits* laden (in dem sie natürlich zuvor vom Anwender gespeichert worden sein sollten). In diesem Punkt sollte die Kurzanleitung besonders genau gelesen werden !

Um nicht versehentlich die gleichnamige Datei des Nachbarn zu überschreiben und ein Durcheinander zu verursachen, muss daher jeder Kursteilnehmer das komplette LogiFlash-Verzeichnis vom zentralen Laufwerk V: (z.Z. *V:\Milzner\IF8\LogiFlashv3.01*) in ein privates Verzeichnis unter *Eigene Dateien* kopieren und nachfolgend auch nur diese Kopie verwenden.

LogiFlash wird durch einen Doppelklick auf die HTML-Datei *LogiFlash pur.htm* im LogiFlash-Hauptverzeichnis gestartet. Um den Zugriff zu vereinfachen, ist es sinnvoll, sich einen entsprechenden Verweis (engl. *Link*) direkt auf dem Desktop anzulegen.

2.4 Schaltnetze

Wir werden uns im folgenden mit dem Entwurf von *Schaltnetzen* auseinandersetzen. Der Begriff *Schaltnetze* bezeichnet logische Schaltungen, bei denen die Ausgangswerte *ausschließlich* von den jeweiligen Eingangswerten abhängen[3]. Dies sind alle Schaltungen, die über kein Speicherverhalten vorhergehender Zustände und damit nicht über Rückführungen von Aus- zu Eingängen verfügen.

2.4.1 Funktionstabelle und Übertragungsfunktion

Erste Beispiele für Schaltnetze haben wir bereits im Kapitel *2.2 Abgeleitete Grundfunktionen* mit NAND-, NOR- und EXOR-Gattern gesehen, ohne dabei jedoch die Herleitung der Übertragungsfunktion aus der Funktionstabelle näher zu betrachten. Dies wollen wir jetzt nachholen.

Die Tabelle unten beschreibt das Verhalten eines EXOR-Gatters. Da das Gatter zwei Eingänge haben soll (auch mehr sind möglich !), muss die Tabelle außer der Kopfzeile $2^2 = 4$ Zeilen aufweisen.

Um alle verschiedenen Zustandskombinationen der Variablen a und b zu erfassen, werden diese einfach dual von Null in der ersten Zeile bis vier in der letzten Zeile durchnummeriert. Die Spalte für die Ausgangsvariable c beschreibt das zugehörige Verknüpfungsergebnis, im Fall eines EXOR die Verknüpfung „*entweder a oder b*“.

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

$(\overline{a} \wedge b)$

$\vee$

$(a \wedge \overline{b})$

Die Herleitung der Übertragungsfunktion auf der Basis von logischen Grundfunktionen aus der Tabelle erfordert drei Schritte:

1) Markiere alle Zeilen, in denen die Ausgangsvariable den Wert 1 aufweist.
2) Bilde von *jeder* dieser Zeilen eine UND-Verknüpfung (Konjunktion) der einzelnen Eingangsvariablen.
3) Verknüpfe *alle* Zeilenterme durch ODER zu *einem* Gesamtterm (Disjunktion):

$$c = (\overline{a} \wedge b) \vee (a \wedge \overline{b})$$

Der hieraus entstandene Gesamtterm für die Übertragungsfunktion wird als *disjunktive Normalform* bezeichnet.

Hinweis: Übertragungsfunktionen in disjunktiver Normalform können bei vielen „Einsen“ der Ausgangsvariablen sehr groß werden und sind in der Regel dann, im Gegensatz zu obigem Beispiel, nicht minimal, d.h. sie beinhalten Redundanzen. Sie sollten daher, bevor sie mittels Gattern realisiert werden, in eine Minimalform überführt werden. Hierzu gibt es verschiedene graphische oder algebraische Rechenverfahren (z.B. Karnaugh-Diagramme), die aber den Rahmen dieses Unterrichts sprengen und daher nicht behandelt werden. Falls erforderlich, werden wir statt dessen einen pragmatischen Ansatz des „scharfen Hinguckens“ verwenden (siehe die Kapitel Temperaturüberwachung / Volladdierer).

[3] In der Literatur werden *Schaltnetze* auch als *Kombinatorische Logik* bezeichnet.

2.4.2 Temperaturüberwachung

Als Beispiel für die Vereinfachung einer Schaltung bzw. des Terms der Übertragungsfunktion entwerfen wir eine Schaltung zur Temperaturüberwachung und Steuerung eines Lüfters, z.B. für den CPU-Lüfter eines Rechners oder den Kühlerventilator eines Autos.

Hierfür gelten die folgenden Randbedingungen:

1) Ein digitaler Temperatursensor misst die Temperatur und liefert den Temperaturwert in Form einer 3Bit-Dualzahl abc, also in einem Zahlenbereich von 0 bis 7. Bei einer Schrittweite von z.B. 10 Grad pro Wert würde ein Messbereich von 0 bis 70 Grad entstehen.
2) Der Lüfter soll eingeschaltet werden, wenn der Temperaturwert die Zahl 4 bzw. 40°C übersteigt.

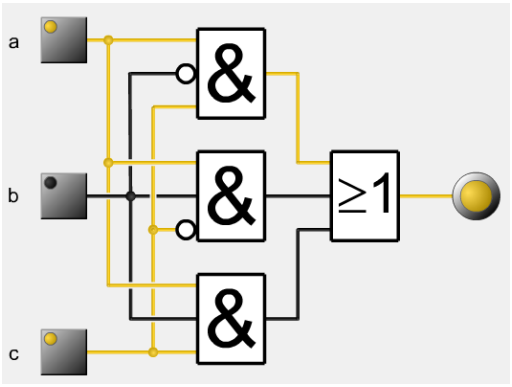
Bezeichnet man die einzelnen Ziffern der 3Bit-Dualzahl mit a, b und c, lässt sich damit die folgende Funktionstabelle aufstellen:

a	b	c	Lüfter
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Das oben beschriebene Verfahren zur Ermittlung der Übertragungsfunktion in disjunktiver Normalform ergibt:

$$\text{Lüfter} = (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c}) \vee (a \wedge b \wedge c)$$

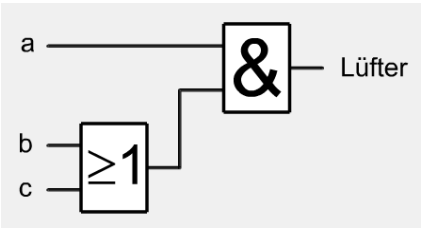
Diese Funktion kann mit Hilfe von drei UND-Gattern mit jeweils 3 Eingängen, einem Oder-Gatter mit ebenfalls 3 Eingängen sowie zwei Negationen direkt realisiert werden:



Wir wollen jedoch nun versuchen, die Funktion durch das Erkennen von Redundanzen weiter zu vereinfachen, um eine weniger aufwendige Schaltung zu erhalten.

Dazu betrachten wir zunächst die Variable a. Diese tritt in allen drei Teiltermen nur in dem Zustand 1 auf (siehe auch gestrichelter Kreis in der Funktionstabelle). Da sie ihren Zustand *nicht* wechselt, können wir sie getrennt behandeln und ausklammern (Distributivgesetz). Damit ergibt sich:

$$\text{Lüfter} = a \wedge ((\overline{b} \wedge c) \vee (b \wedge \overline{c}) \vee (b \wedge c))$$



Diese Form ist bereits kleiner als der Ausgangsterm.

Betrachten wir jetzt nur die Variablen b und c, d.h. die Teilterme in der Klammer, so kann man erkennen, dass diese einer ODER-Verknüpfung von b und c entsprechen (siehe auch Kapitel *2.1.1 Die ODER-Funktion*). Wir können den Term und damit auch die Schaltung also weiter deutlich vereinfachen:

$$\text{Lüfter} = a \wedge (b \vee c)$$

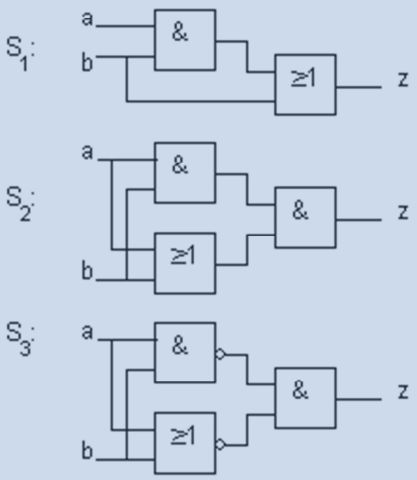
Damit ergibt sich als Ergebnis der links abgebildete vereinfachte Schemaplan.

Aufgaben: 1) Erstelle die Übertragungsfunktion und den Schemaplan für eine Schaltung zur Temperaturüberwachung, die den Lüfter ab einem Wert von 6 einschaltet.

2) Erstelle für die rechts gegebene Funktionstabelle
   a) die Übertragungsfunktion,
   b) den Schemaplan der zugehörigen logischen Schaltung und
   c) eine vereinfachte Schaltung durch Ausnutzung von Redundanzen.

a	b	c	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3) Gehe jetzt den umgekehrten Weg und leite jeweils die Übertragungsfunktion für $z(S_1)$, $z(S_2)$ und $z(S_3)$ aus den Schemaplänen für die unten abgebildeten Schaltnetze S_1, S_2 und S_3 ab.



4) Ergänze dann die unten stehende Funktionstabelle um die Werte der Ausgangsvariablen z für die einzelnen Schaltnetze.

a	b	$z(S_1)$	$z(S_2)$	$z(S_3)$
0	0			
0	1			
1	0			
1	1			

5) Erstelle die Schaltungen mit Hilfe von LogiFlash und überprüfe dein Ergebnis durch Simulation. Ergänze hierzu die Eingänge a und b um entsprechende Schalter, den Ausgang um eine Lampe als Anzeige.

2.4.3 Halbaddierer

Ein *Halbaddierer* ist ein Schaltnetz, mit dem zwei einstellige Dualzahlen (d.h. zwei Zahlen mit je einem Bit Länge) addiert werden können.

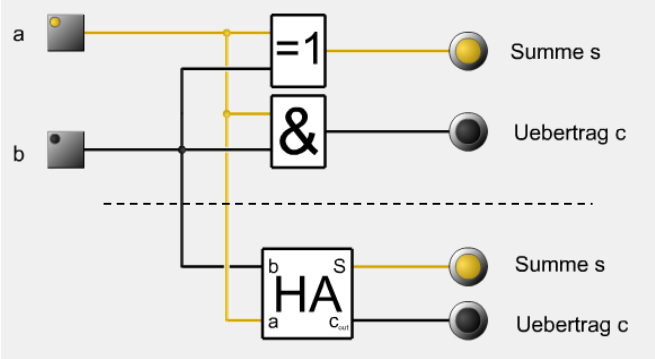
Das Ergebnis besteht aus der Summe der beiden Zahlen sowie einem möglichen Übertrag in die nächsthöhere Stelle einer mehrstelligen Zahl (siehe hierzu auch *Kapitel 1.2.2 : Addition von Dualzahlen*). Ein Halbaddierer muss daher zwei Eingänge für die beiden Zahlen a und b sowie je einen Ausgang für Summe und Übertrag haben. Damit ergibt sich folgende Funktionstabelle:

a_0	b_0	Summe s_0	Übertrag c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Das Vorhandensein von zwei Ausgängen bedeutet, dass es auch jeweils eine zugehörige Übertragungsfunktion geben muss. Durch Aufstellen der disjunktiven Normalform erhalten wir für Übertrag und Summe:

$c_1 = a \wedge b$

$s_0 = (\overline{a_0} \wedge b_0) \vee (a_0 \wedge \overline{b_0}) = a_0 \veebar b_0$



Da die Funktion der Summe einer EXOR-Verknüpfung und die Funktion für den Übertrag einer UND-Verknüpfung der beiden Eingänge entspricht, kann der Halbaddierer aus diesen beiden Gattern zusammengesetzt werden. Eine entsprechende Schaltung ist links in der oberen Hälfte des Schemaplans dargestellt. Alternativ könnte statt des EXOR-Gatters natürlich auch eine Schaltung aus zwei UND- und einem ODER-Gatter wie in Kapitel 2.2.2 gezeigt eingesetzt werden.

Der untere Teil des Schemaplans zeigt das Symbol für einen Halbaddierer, wie es in LogiFlash zur Abstraktion der oberen Schaltung verwendet wird.

Aufgabe: 1) Erstelle den Schemaplan eines Halbaddierers ohne Verwendung eines EXOR-Gatters.

2) Begründe: Könnte der Halbaddierer durch Hintereinanderschalten für die Addition jeder Stelle einer 4-bit breiten Dualzahl verwendet werden ?

2.4.4 Volladdierer

Wollen wir *beliebige* Stellen zweier mehrstelliger Dualzahlen miteinander addieren, so gehen in die Addition nicht nur die Werte der beiden Ziffern ein, sondern auch ein möglicher Übertrag aus der nächst niederen Stelle. Wir müssen in diesem Fall also *drei* einstellige Dualzahlen addieren. Eine logische Schaltung, die dies leistet, heißt *Volladdierer*.

Das Ergebnis besteht, wie beim Halbaddierer (s.o.), aus der Summe der beiden Ziffern sowie einem möglichen Übertrag in die nächsthöhere Stelle der mehrstelligen Zahl. Ein Volladdierer hat daher drei Eingänge für die drei Zahlen a, b und c sowie je einen Ausgang für Summe und Übertrag.
Damit ergibt sich nach den Additionsregeln für Dualzahlen folgende Funktionstabelle für die Addition einer beliebigen Stelle n:

Übertrag
aus der nächstniederen Stelle in die nächsthöhere Stelle

a_n	b_n	c_{n-1}	Übertrag c_n	Summe s_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Wertet man die Einsen in den Ergebnisspalten aus, erhält man die beiden unten dargestellten Übertragungsfunktionen.

Übertragungsfunktion für den Übertrag

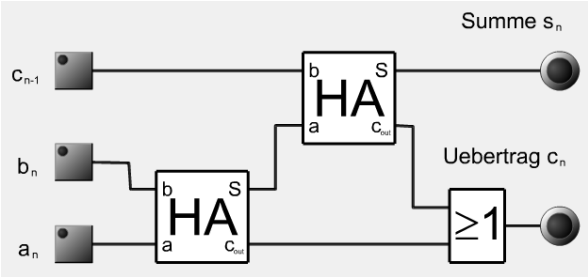
$$c_n = (\overline{a_n} \wedge b_n \wedge c_{n-1}) \vee (a_n \wedge \overline{b_n} \wedge c_{n-1}) \vee (a_n \wedge b_n \wedge \overline{c_{n-1}}) \vee (a_n \wedge b_n \wedge c_{n-1})$$

Diese Übertragungsfunktion kann weiter vereinfacht werden. Wenn wir uns die letzten beiden Teilterme bzw. Tabellenzeilen anschauen, erkennen wir, das a_n und b_n ihren Zustand nicht ändern und daher die Zustandsänderung bei c_{n-1} ist offensichtlich irrelevant ist und wegfallen kann:

$$c_n = (\overline{a_n} \wedge b_n \wedge c_{n-1}) \vee (a_n \wedge \overline{b_n} \wedge c_{n-1}) \vee (a_n \wedge b_n)$$

Um die Funktion noch weiter zu vereinfachen, betrachten wir jetzt die ersten beiden Teilterme (Tabellenzeilen 4 und 6). Hier kann man, da c_{n-1} hierin seinen Zustand nicht ändert, c_{n-1} ausklammern und erhält $c_{n-1} \wedge ((\overline{a_n} \wedge b_n) \vee (a_n \wedge \overline{b_n}))$. Der Term in der Klammer entspricht einer EXOR-Verknüpfung, sodass wir letztendlich für den Übertrag erhalten:

$$c_n = (c_{n-1} \wedge (a_n \veebar b_n)) \vee (a_n \wedge b_n)$$



Übertragungsfunktion für die Summe

$s_n = (a_n \veebar b_n \veebar c_n) \vee (a_n \wedge b_n \wedge c_{n-1})$

$= (a_n \veebar b_n) \veebar c_n$

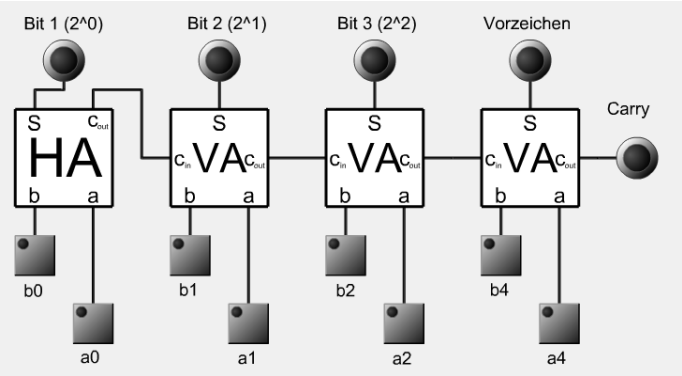
Mit Hilfe dieser beiden Übertragungsfunktionen kann ein Volladdierer aus zwei Halbaddieren sowie einem ODER-Gatter zusammengesetzt werden. Dabei ergibt sich der links stehende Schemaplan.

2.4.5 Addierer für 4Bit-Dualzahlen

Wir wollen nun mit Hilfe der zuvor betrachtenen Addiererschaltungen einen Addierer entwerfen, der nicht nur einstellige, sondern mehrstellige Dualzahlen addieren kann. Aus Gründen der Übersichtlichkeit beschränken wir uns dabei auf 4Bit-lange Dualzahlen im Zweierkomplement.

Für die niederwertigste Stelle, in die kein Übertrag aus vorhergehenden Stellen eingeht, ist ein Halbaddierer ausreichend. Für alle anderen Stellen muss ein Volladdierer verwendet werden, um stellenweise beide Ziffern plus einen möglichen Übertrag aus der vorhergehenden Stelle addieren zu können. Zum Weiterleiten von Überträgen von einer Stelle zur nächsten werden die entsprechenden Aus- und Eingänge für Überträge miteinander verbunden.

Damit ergibt sich ein Schemaplan wie unten dargestellt.



2.4.6 Addierer mit Überlauf-Erkennung (✶)

Ein *Überlauf* und damit ein falsches Additionsergebnis entsteht, wenn die Summe der beiden zu addierenden Zahlen mit der Anzahl der im Addierer vorhandenen Stellen nicht mehr darstellbar ist (siehe auch Kapitel *1.2.3 Subtraktion*, Hinweis am Ende).

Um die Merkmale für das Auftreten eines Überlaufs zu ermitteln, führen wir die drei unten gezeigten Additionen von 4Bit-langen Zahlen im Zweierkomplement durch. Hiermit lässt sich ein Zahlenbereich von

$$-(2^3) = -8 \quad \text{bis} \quad 2^3 - 1 = 7$$

darstellen. Die gewählten Zahlenbeispiele für die ersten beiden Additionen führen also zu einem provoziertem Überlauf, da das Ergebnis ausserhalb des darstellbaren Zahlenbereiches liegt und damit falsch ist. Das Ergebnis der dritten Addition liegt dagegen innerhalb des zulässigen Bereiches.

dezimal	dual
6	0110
+ 3	0011
9	1001 = -7_{10}

Fall 1: Übertrag *in* die Vorzeichenstelle

dezimal	dual
(-6)	1010
+ (-3)	1101
–9	(1)0111 = 7_{10}

Fall 2: Übertrag *aus* der Vorzeichenstelle

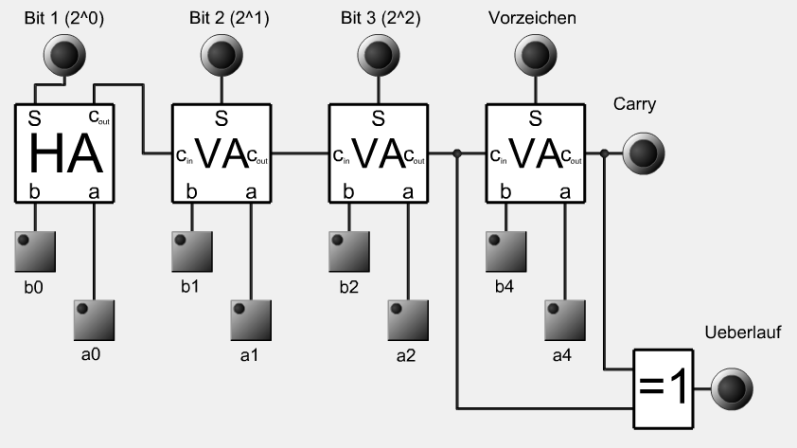
dezimal	dual
(-3)	1101
+ (-3)	1101
–6	(1)1010 = -6_{10}

Fall 3: Übertrag *in* die Vorzeichenstelle *und aus* der Vorzeichenstelle

Ein Überlauf kann nur durch einen nicht zulässigen Übertrag entstehen. Untersucht man die drei obigen Fälle hinsichtlich der aufgetretenen Überträge, so findet man heraus:

> Ein Überlauf hat genau dann stattgefunden, wenn die Überträge aus Vorzeichenstelle und höchster Stelle unterschiedlich sind oder, mit anderen Worten, wenn *entweder* ein Übertrag in die Vorzeichenstelle *oder* aus der Vorzeichenstelle aufgetreten ist.

Dies entspricht einer Exklusiv-ODER-Verknüpfung dieser beiden Überträge. Damit kann das Schaltschema des 4Bit-Addierers einfach erweitert werden, um Überläufe direkt anzuzeigen:



Aufgabe: Berechne überschlägig, wie viele Grundgatter (AND. OR, NOT) benötigt werden, um einen Addierer für 64Bit-breite Zahlen zu konstruieren.

2.5 Schaltwerke

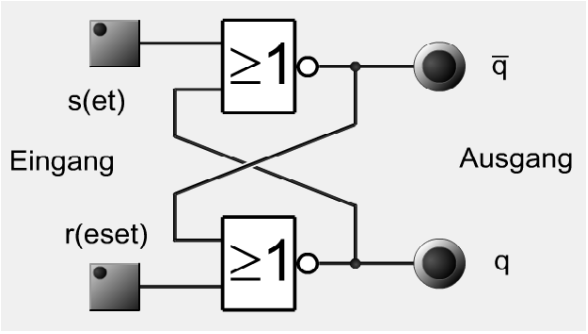
Der Begriff *Schaltwerke* bezeichnet im Gegensatz zu *Schaltnetzen* logische Schaltungen, bei denen die Ausgangswerte nicht nur von der aktuellen Belegung der Eingangswerte abhängen, sondern auch von deren Vorgeschichte bzw. Vorgängerzuständen. Hierzu sind Rückführungen von Aus- zu Eingängen erforderlich, um Zustände speichern zu können.

Wir werden uns im folgenden mit der zentralen Struktur eines Speichers für 1Bit beschäftigen. Diese wird lautmalerisch *Flip-Flop* genannt, da sie zwischen zwei stabilen Zuständen hin und her schalten kann. *Flip-Flops* gibt es in vielen Varianten, von denen wir nur die grundlegenden Ausführungen betrachten. *Flip-Flops* sind Basisbausteine in jedem Computer und werden in modernen Prozessoren millionenfach verwendet (z.B. für CPU-Register, Cache-Speicher, Adress-Latches, etc.).

2.5.1 RS-Flip-Flop

Die einfachste Form einer 1Bit-Speicherzelle ist das RS-Flip-Flop. Dieses kann z.B. mit zwei NOR-Gattern wie im Bild unten aufgebaut werden. Das Neue gegenüber bisherigen Schaltungen sind zwei „über Kreuz"-Rückführungen jeweils vom Ausgang eines Gatters zu einem Eingang des anderen.

Jedes RS-Flip-Flop hat, wie der Name schon sagt, die beiden Eingänge s (*set*) und r (*reset*) sowie die komplementären Ausgänge q und $\overline{q}$ („q quer"). Komplementär bedeutet, dass die Ausgänge jeweils versuchen, genau entgegengesetzte Zustände anzunehmen.



s	r	q	$\overline{q}$	Zustand
0	0	q_{-1}	$\overline{q}_{-1}$	← *hold*
0	1	0	1	← *reset*
1	0	1	0	← *set*
1	1	(0)	(0)	← *instabil*

RS-Flip-Flop mit NOR-Gattern

Das Verhalten kann anhand der Funktionstabelle nachvollzogen werden. Liegt an beiden Eingängen jeweils eine Null (Zeile 1), so nimmt der Ausgang jedes Gatters jeweils den Zustand an, den der andere Ausgang zu diesem Zeitpunkt hat. Da dies genau dem Zustand *vor* dem Wechsel der Eingangszustände auf Null/Null entspricht und sich an den Ausgangszuständen dabei nichts ändert, wird diese Eingangskombination als „Zustand halten" bezeichnet (engl. *hold*).

Wird an den Reset-Eingang eine Eins gelegt und der Set-Eingang auf Null belassen (Zeile 2), wird der Ausgang des unteren NOR-Gatters Null (q = 0) und der Ausgang des oberen NOR Eins ($\overline{q}$ = 1). Dies wird als Rücksetzen des Flip-Flop bezeichnet (engl. *reset*). Gehen beide Eingänge danach auf Null zurück, bleibt der Zustand erhalten (s.o.), d.h. der Ausgang q *speichert* den Wert Null.

Wird dagegen an den Set-Eingang eine Eins gelegt und der Reset-Eingang auf Null belassen (Zeile 3), geht der Ausgang des oberen NOR-Gatters auf den Wert Null und damit der Ausgang des unteren NOR-Gatters auf Eins (q = 1). Dies wird als Setzen des Flip-Flop bezeichnet (engl. *set*). Gehen nun beide Eingänge auf Null zurück, bleibt ebenfalls der Zustand erhalten und der Ausgang q *speichert* den Wert Eins.

Besondere Beachtung erfordert der letzte Zustand, in dem an beiden Eingängen eine Eins liegt (Zeile 4). Aufgrund der NOR-Funktionen gehen *beide* Ausgänge auf den Wert Null, d.h. sie sind nicht mehr wie erwartet komplementär. Gehen dann in der Folge beide Eingänge *gleichzeitig* auf Null zurück (d.h. in den Zustand *hold*, s.o.), so springen auch *beide* Ausgangszustände *gleichzeitig* auf den Wert Eins um. Aufgrund dieser Symmetrie (als Gegenteil von komplementär) „weiß" das Flip-Flop nicht, welchen Zustand es beibehalten bzw. annehmen soll und die Ausgänge springen wiederum auf den Wert Null zurück. Dieser Vorgang wiederholt sich schwingungsartig. Der Ausgangszustand ist daher *instabil* und nicht vorhersagbar.

In der Praxis ist die Verwendung eines RS-Flip-Flop in einer größeren Schaltung aufgrund des instabilen Zustands, der in jedem Fall vermieden werden sollte, problematisch. Es bildet daher in der Regel nur den Kern einer Speicherzelle und wird durch zusätzliche Logikschaltungen, die ein stabiles Verhalten sicherstellen sollen, ergänzt.

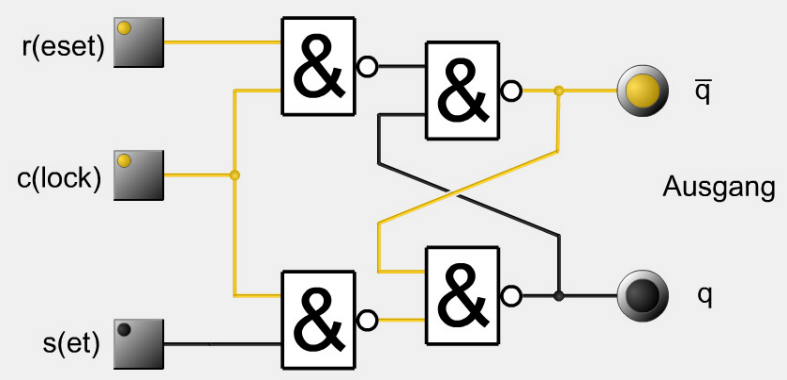
Tipp: Um den instabilen Zustand eines RS-Flip-Flop in LogiFlash zu visualisieren, zunächst den Simulationsmodus ausschalten, dann beide Eingänge auf Null setzen und danach den Simulationsmodus wieder einschalten. Resultat: beide Ausgänge flackern wild hin und her…

Aufgabe: 1) Erstelle den Schemaplan eines RS-Flip-Flops aus zwei NAND-Gattern.

2) Erstelle die Funktionstabelle für dieses RS-Flip-Flops. Was hat sich im Verhalten gegenüber der Ausführung mit NOR-Gattern geändert ?

2.5.2 Taktgesteuertes RS-Flip-Flop (✶)

Der erste Schritt zur Erweiterung eines RS-Flip-Flop ist das Vorschalten von zwei NAND-Gattern vor die beiden Flip-Flop-Eingänge (siehe Schemaplan unten). Hierdurch kann über einen zusätzlichen Eingang c (*clock*) kontrolliert werden, *wann* die beiden Eingänge r und s wirksam sein sollen. Daher wird dieser zusätzliche Eingang auch Takteingang und das Flip-Flop als Ganzes *taktgesteuertes RS-Flip-Flop* genannt.

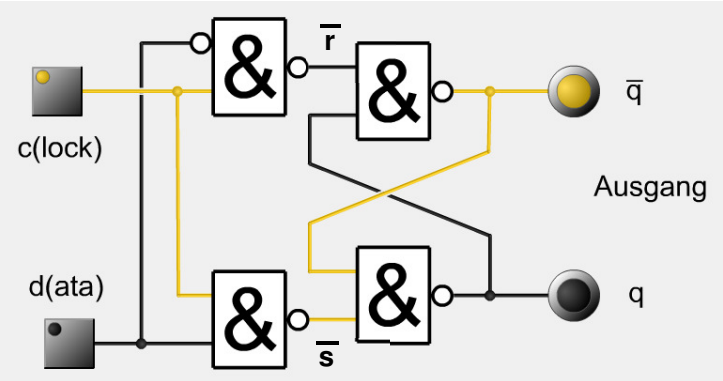


Für c = 0 sind die Eingänge r und s gesperrt und das Flip-Flop befindet sich im *hold*-Zustand (NAND-RS-Flip-Flop !).

Für c = 1 funktionieren beide Eingänge wie gewohnt.

2.5.3 D-Flip-Flop (✶)

Im nächsten Schritt wird die vorhergehende Schaltung geringfügig verändert. Die beiden Eingänge *set* und *reset* sind hierin jetzt über eine zusätzliche Negation zu einem einzigen Eingang d (*data*) zusammengefasst (siehe Schemaplan). Hierdurch wird ein instabiler Zustand sicher vermieden, da beide Eingänge der NAND-Gatter auf der linken Seite immer mit entgegengesetzten Werten belegt sind. Abhängig vom Wert des Takteinganges c können die Flip-Flop-Eingänge $\overline{r}$ und $\overline{s}$ damit auch nur entweder mit entgegengesetzten Werten belegt sein oder die Wertekombination Eins/Eins aufweisen (bei c = 0, entsprechend dem *hold*-Zustand eines NAND-RS-Flip-Flops).



c	d	q	$\overline{q}$
0	0	q_{-1}	$\overline{q}_{-1}$
0	1	q_{-1}	$\overline{q}_{-1}$
1	0	0	1
1	1	1	0

D-Flip-Flop

Die Funktionstabelle macht dieses Verhalten deutlich. Solange der Takteingang c den Wert c = 0 hat, befindet sich das gesamte Flip-Flop im *hold*-Zustand und die Ausgänge speichern den Vorgängerzustand q_{-1} bzw. q_{-1}. Dies ist derjenige Zustand, der beim *vorhergehenden Takt* (d.h. c = 1) am Eingang d vorgelegen hat. Wird ein neuer Taktimpuls angelegt, d.h. c erneut auf den Wert Eins gesetzt (Zeilen 3 und 4 der Tabelle), übernimmt der Ausgang q genau den Wert, der am Eingang d liegt.

Damit ist ein D-Flip-Flop eine Speicherzelle für 1Bit, die immer dann den Wert des Bits am Eingang d speichert, wenn der Takteingang c seinen Zustand von Null auf Eins ändert.

3 Lösungen zu den Übungen (erst selber versuchen!)

Kapitel 1 Zahlensysteme

Seite 4

Umwandeln von Dualzahlen: $10111111_2 = 191_{10}$

$11110000_2 = 240_{10}$

Umwandeln von Dezimalzahlen: $78_{10} = 01001110_2$

$250_{10} = 11111010_2$

Seite 5

Schriftliches Addieren:

101010	101010	10101
+ 10101	+101010	+ 1011
111111	1010100	100000

Seite 7

Darstellung in Zweierkomplement-Form: $-54_{10} = 11001010_z$

$-127_{10} = 10000001_z$

$-128_{10} = 10000000_z$

Berechnung im Zweierkomplement:

45	00101101	81	01010001
–22	+ 11101010	–107	+ 10010101
23	00010111_z	–26	11100110_z

00010111
$= 1 + 2 + 4 + 16 = 23_{10}$

11100101
00011010
$= 2 + 8 + 16 = 26_{10}$

Umwandlung in eine Hexadezimalzahl:

10	1100	0101	1000	1111	0100_2	$= 2C58F4_{16}$
2	C	5	8	F	4	

Umwandlung in eine Dualzahl:

$A0B1C_{16} =$	1010	0000	1011	0001	1100_2
	A	0	B	1	C

Umwandlung in eine Dezimalzahl:

$CAFE_{16} = 14 * 1 +$
$15 * 16 +$
$10 * 256 +$
$12 * 4096$
51966_{10}

Kapitel 2 Logische Schaltungen

Seite 14

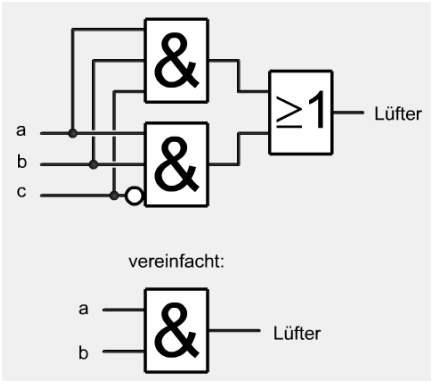
1)

a	b	c	Lüfter
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Übertragungsfunktion:

$$\text{Lüfter} = (a \wedge b \wedge \overline{c}) \vee (a \wedge b \wedge c)$$
$$= (a \wedge b)$$

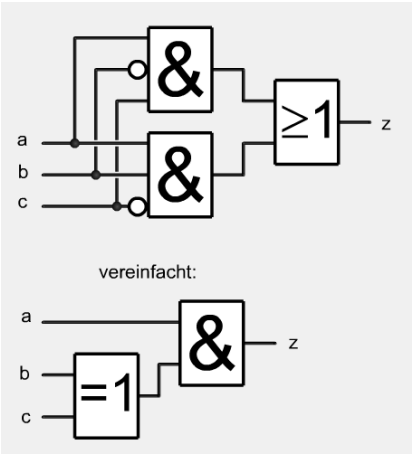
Schemaplan:



2) $z = (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c})$

$= a \wedge ((\overline{b} \wedge c) \vee (b \wedge \overline{c}))$

$= a \wedge (b \veebar c)$



3) $z(S_1) = (a \wedge b) \vee b$

$z(S_2) = (a \wedge b) \wedge (a \vee b)$

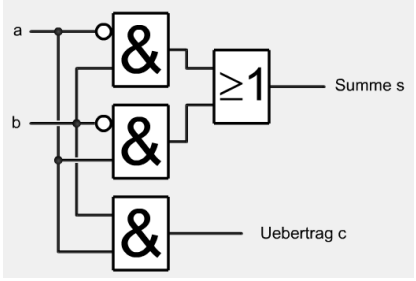
$z(S_3) = \overline{(a \wedge b)} \wedge \overline{(a \vee b)}$

4)

a	b	$z(S_1)$	$z(S_2)$	$z(S_3)$
0	0	0	0	1
0	1	1	0	0
1	0	0	0	0
1	1	1	1	0

Seite 15

1)



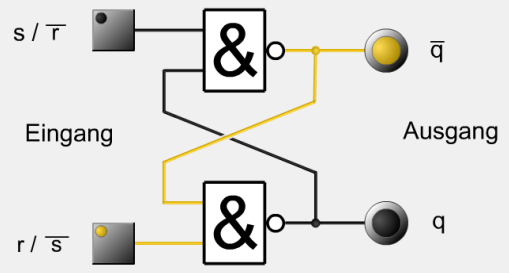
2) Nein, da ein Halbaddierer keinen Übertrag aus der nächstniederen Stelle berücksichtigen kann, kann er nur als Addierer für die niederwertigste Stelle verwendet werden (siehe Kapitel 2.4.5).

Seite 17

1 Halbaddierer: 3 AND, 1 OR, 2 NOT

63 Volladdierer: 126 Halbaddierer, 63 OR ⇒ 127 x 6 + 63 = 825 Gatter

Seite 19



s	r	q	$\overline{q}$	Zustand
0	0	(1)	(1)	← *instabil*
0	1	0	1	← *reset*
1	0	1	0	← *set*
1	1	q_{-1}	$\overline{q}_{-1}$	← hold

RS-Flip-Flop mit NAND-Gattern