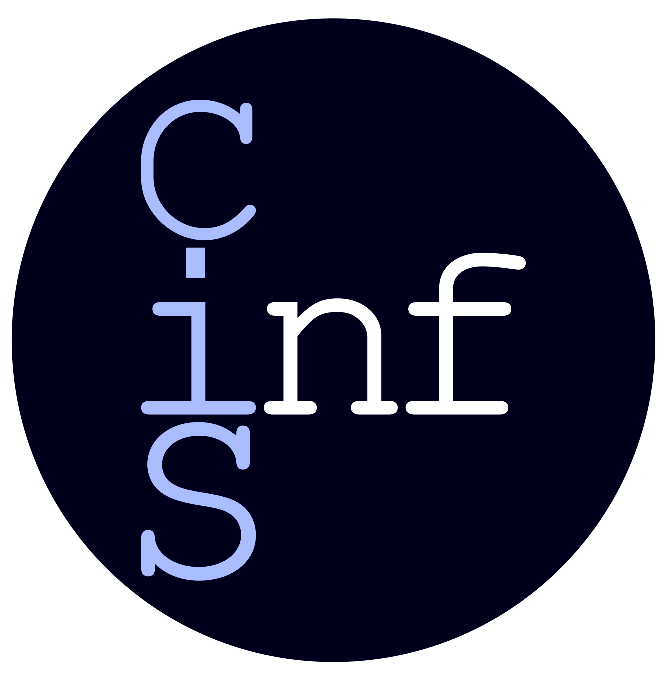


Rechenerstrukturen und Betriebssysteme

Alan Kniep, Livi Franke

12. Januar 2021

Inhaltsverzeichnis

1 Organisatorisches **1**

2 Einführung **2**

3 Informationsverarbeitung **5**

4 Ziffern und Zahlen **7**

4.1 Dualsystem . . . 8

4.1.1 Aritmethik . . . 9

4.2 Stellenwertsysteme . . . 10

4.3 Umrechnung zwischen Stellenwertsystemen . . . 11

4.3.1 vorberechnete Potenztabellen . . . 11

4.3.2 Divisionsrestverfahren . . . 12

4.3.3 Horner-Schema . . . 13

4.4 Präfixe . . . 15

4.5 Festkommadarstellung . . . 16

4.6 Negative Zahlen . . . 17

4.6.1 Betrag und Vorzeichen . . . 17

4.6.2 Exzess-Codierung (Offset) . . . 17

4.6.3 Komplementdarstellung . . . 17

5 Arithmetik **20**

5.1 Overflow . . . 20

5.2 Sign Extension . . . 20

5.3 Subtraktion . . . 20

5.4 Division (Integer Division) . . . 21

5.5 Reziprokwert . . . 21

5.6 Heron-Verfahren . . . 21

5.7 Informationstreue . . . 22

6 Zeichen und Text **23**

6.1 Zeichensätze . . . 23

6.1.1 ASCII . . . 23

6.1.2 ISO-8859 . . . 23

6.1.3 Microsoft: Codepages . . . 23

6.1.4 Unicode . . . 23

7 Logische Operationen **25**
7.1 Boole'sche Algebra . . . 25
7.2 Logische Operationen, Ring der Algebra . . . 27
7.3 Bitweise Operationen . . . 28

8 Codierung **30**
8.1 Einschrittiger Code . . . 31
8.2 Graycode . . . 32
8.3 Optimalcodes . . . 33
8.3.1 Shannon-Fano Coding . . . 33
8.3.2 Huffman Coding . . . 33
8.3.3 Kraft-Ungleichung . . . 34
8.4 Informationstheorie . . . 35
8.4.1 Entropie . . . 35
8.4.2 Redundanz . . . 36
8.4.3 Kanalkapazität . . . 37
8.4.4 Fehlertypen . . . 37
8.4.5 Begriffe zur Fehlerbehandlung . . . 38
8.4.6 Verfahren zur Fehlerbehandlung . . . 38
8.4.7 Binärpolynome . . . 42
8.4.8 Zyklische Codes (CRC) . . . 43

9 Schaltfunktionen **45**
9.1 Definition . . . 45
9.2 Darstellung . . . 46
9.2.1 Funktionstabellen . . . 46
9.2.2 Verhaltensbeschreibung . . . 46
9.2.3 Strukturbeschreibung . . . 46
9.3 Normalformen . . . 47
9.3.1 Disjunktive Normalform (DNF) . . . 47
9.3.2 Allgemeine disjunktive Form . . . 47
9.3.3 Konjunktive Normalform (KNF) . . . 47
9.3.4 Allgemeine konjunktive Form . . . 48
9.3.5 Reed-Muller Form . . . 48
9.4 Entscheidungsbäume und OBDDs . . . 50
9.5 Minimierung . . . 51
9.5.1 Algebraische Minimierungsverfahren . . . 51
9.5.2 Grafische Minimierungsverfahren . . . 51
9.5.3 Quine-McCluskey Algorithmus . . . 52

10 Schaltnetze **53**
10.1 Definition . . . 53
10.2 Gatter . . . 53
10.3 Schaltpläne . . . 54

10.4 Grundlegende Schaltnetze 54
10.5 Addierer 55
10.6 Multiplizierer 57
10.7 Prioritätsencoder 57
10.8 Shifter, Rotator, etc. 57
10.9 Arithmetisch-logische Einheit (ALU) 58
10.10 Zeitverhalten von Schaltungen 58
10.11 Impulsdiagramme 59
10.12 Hazards 60

11 Schaltwerke **61**
11.1 Definition 61
11.2 Deterministische Endliche Automaten 61
11.3 Zeitglieder / Flipflops 62

12 Glossar **64**

1 Organisatorisches

Vorlesung:

- 14 Vorlesungswochen
- ca. 3h Vorlesungsmaterial pro Woche
- Videokonferenz: Freitags ab 12:15 Uhr

Übungsgrupen:

- 13 Übungstermine
- 12 Aufgabenblätter (+1 Präsenzaufgabenblatt)
- Gruppenarbeit (maximal 3 Studierende)
- Scheinkriterien:
  - regelmäßige aktive Teilnahme
  - höchstens zweimal unentschuldigt fehlen
  - mind. zweimal vorrechnen
  - jeder Übungszettel bearbeitet: mind 30% der Punkte Pro Blatt
  - insgesamt mind. 50% der Punkte

2 Einführung

Informatik:

"Die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (→ Computer)."

Brockhaus-Enzyklopädie

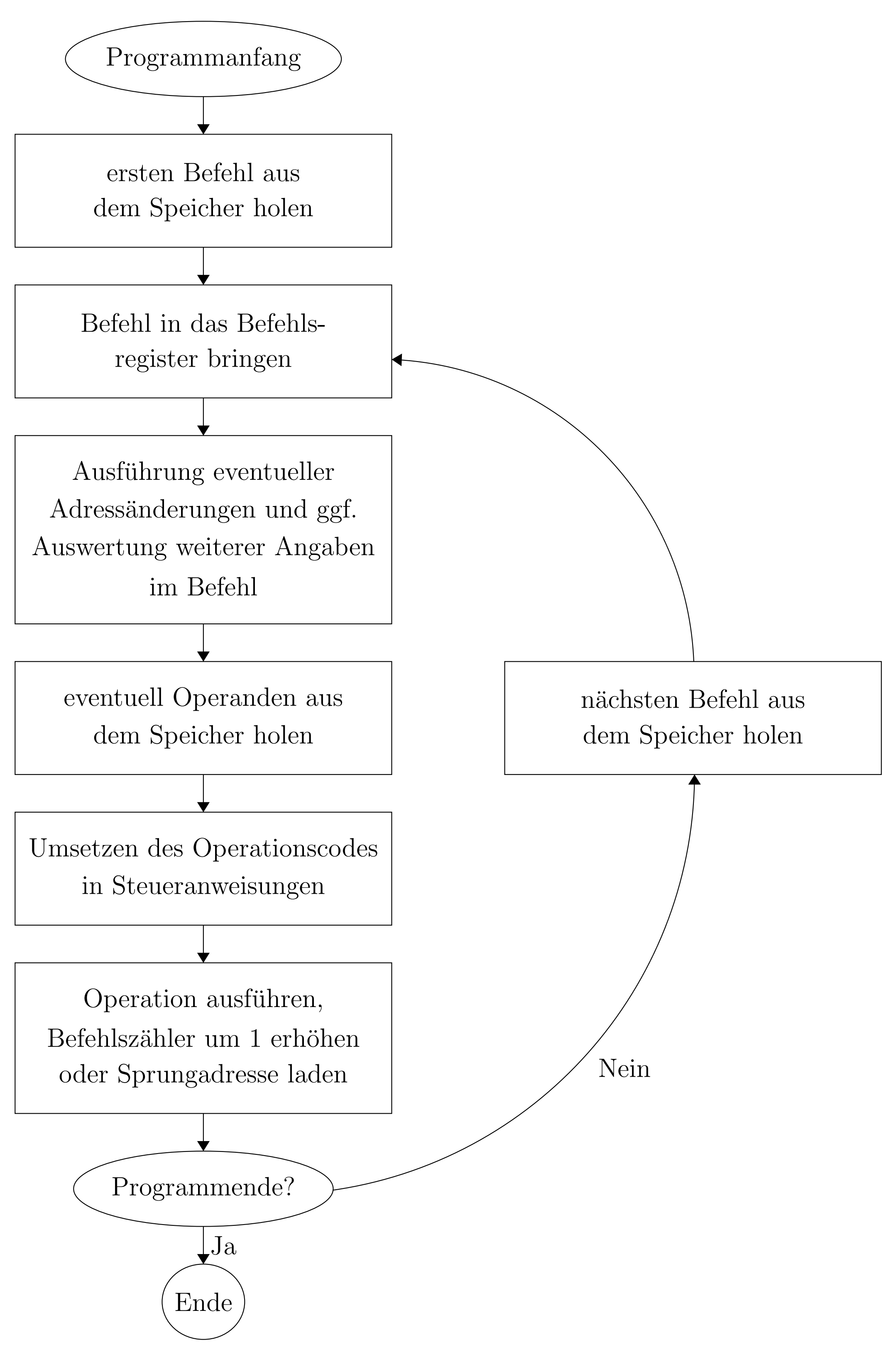
Das Grundverständnis von der Interaktion zwischen Software und Hardware ist erforderlich für **performante Software** und für **Sicherheitsaspekte**.

- Systemsicht/Variantenvielfalt von Mikroprozessorsystemen
- Supercomputer, Server, Workstations, PCs
- Medienverarbeitung, Mobile Geräte
- RFID-Tags, Wegwerfcomputer

Technische Fortschritte in Mikro- und Optoelektronik mit exponentiellen Wachstum:

- Rechenleistung von Prozessoren ("Performanz")
- Speicherkapazität Hauptspeicher (DRAM, SRAM, FLASH)
- Speicherkapazität Langzeitspeicher (Festplatten, FLASH)
- Bandbreite (Netzwerke)

von-Neumann Konzept:
Eine Maschine mit von-Neumann Konzept hat einen minimalen Hardwareaufwand, da es sich um ein abstraktes System mit Prozessor, Speicher und Peripheriegeräten handelt. Hierbei wird der Speicher für Programme und Daten gemeinsam genutzt und fortlaufend adressiert, sodass Programme wie Daten bzw. Daten wie Programme behandelt werden können. Das von-Neumann Konzept nutzt den Fetch-Decode-Execute Befehlszyklus und ist dadurch sehr flexibel, was auch bedeutet, dass es sehr vielfältige Architekturvarianten, Befehlssätze u.Ä. gibt, die dadrauf basieren. Alle aktuellen Rechnern basieren auf diesem Prinzip.


Programmanfang
ersten Befehl aus
dem Speicher holen
Befehl in das Befehls-
register bringen
Ausführung eventueller
Adressänderungen und ggf.
Auswertung weiterer Angaben
im Befehl
eventuell Operanden aus
dem Speicher holen
Umsetzen des Operationscodes
in Steueranweisungen
Operation ausführen,
Befehlszähler um 1 erhöhen
oder Sprungadresse laden
Programmende?
Ja
Ende
Nein
nächsten Befehl aus
dem Speicher holen

erste Computer, ca. 1950

- zunächst noch kaum Softwareunterstützung
- nur zwei Schichten:
  - 1. Programmierung in elementarer Maschinensprache (ISA level)
  - 2. Hardware in Röhrentechnik (device logic level)
    - Hardware kompliziert und unzuverlässig

Mikroprogrammierung

- Programmierung in komfortabler Maschinensprache
- Mikroprogramm-Steuerwerk (Interpreter)
- einfache, zuverlässigere Hardware
- Grundidee der sog. CISC-Rechner (68000, 8086, VAX)

erste Betriebssysteme

- erste Rechner jeweils nur von einer Person benutzt
- Anwender = Programmierer = Operator
- Programm laden, ausführen, Fehler suchen usw.

→ Maschine wird nicht gut ausgelastet

→ Anwender mit lästigen Details überfordert

Einführung von Betriebssystemen

- "system calls"
- Batch-Modus:
  - Programm abschicken
  - Warten
  - Resultate am nächsten Tag

Moore's Law:
Exponetielles Wachstum:

$$L(t) = L(0) \cdot 2^{t/18}$$

$$\begin{aligned} L(t) &= \text{Leistung zum Zeitpunkt } t \\ L(0) &= \text{Leistung zum Zeitpunkt } 0 \end{aligned}$$

3 Informationsverarbeitung

"Ein Computer ist eigentlich eine dumme Kiste," daher ist die Informationsverarbeitung nicht immer fehlerfrei und nicht immer eindeutig. So ist es mathematisch nicht schwer rauszukriegen, dass $0.1+0.1+0.1 = 0.3$ ist, für einen Rechner ist diese Information aber deutlich abstrakter, wodurch die obige Rechnung 0.30000000000000004 ergeben würde.

Ein Rechner hat mehrere Ebenen an (virtuellen) Maschinen M_n mit den akzeptierten Sprachen L_n, wobei M_0 die Sprache L_0 akzeptiert und die unterste Ebene direkt von der Maschine (Hardware) verstanden wird. Es gilt: $L_0 < L_1 < L_2 < ...$

Virtuelle Maschinen sind meistens langsamer als die echte Maschine.

Mächtigere Sprachen werden mithilfe eines **Interpreters** oder **Compilers** in eine jeweils niedrigere Ebene übersetzt. Ein Compiler erzeugt dabei ein neues Programm, welches separat ausgeführt wird, wobei jeder L_n-Befehl in eine dazugehörige Folge an $L_(n-1)$ Befehlen übersetzt wird, während ein Interpreter zu jedem eingelesenen L_n-Befehl die zugehörige $L_(n-1)$-Befehlsfolge direkt ausführt. L_n-Befehle sind für Interpreter einfache Daten, welche verarbeitet werden. Interpreter erlauben die Simmulierung jeder beliebigen Maschiene.

Tabelle 3.1: Folie 87, Einteilung der Ebenen

Ebenenname	RSB?	Funktion
Anwendungsebene	-	Hochsprachen
Assemblerebene	RSB	low-level Anwendungsprogrammierung
Betriebssystemebene	RSB	Betriebssysteme, Systemprogrammierung
Rechnerarchitektur	RSB	Schnittstelle HW & SW, Befehlssatz, Datentypen
Mikroarchitektur	RSB	Steuer- & Operationswerk, Register, ALU, etc.
Logikebene	(RSB)	Grundschaltungen; Gates, Flipflops, ...
Transistorebene	-	Elektrotechnik, Transistoren, Chip-Layout
Physikalische Ebene	-	Geometrien, Chip-Fertigung

Tabelle 3.2: Folie 88, Sechs Ebenen eines x86 Rechners

E_5	Problemorientierte Sprache
E_4	Assemblersprache
E_3	Betriebssystemmaschine
E_2	Befehlssatzarchitektur (ISA)
E_1	Mikroarchitektur
E_0	Digitale Logik

Bei der Informationsverarbeitung unterscheidet man zwischen der **Information** einer **Aussage**, auch **Nachricht** genannt, welche teilweise Raum zur **Interpretation** lassen, sodass man sie in Kombination mit einer Bedeutung **Verstehen** kann.

So kann die Information "13" verschieden verstanden werden. Ihre Repräsentation kann bspw. als Zahl zur Basis 10 gesehen werden (13_{10}) oder als Text in der deutschen Sprache ("dreizehn"), aber auch symbolisch oder sonst wie. Eine Notation ist daher ummer nur eine Repräsentation, nie eine tatsächliche Information. Es benötigt Absprachen fürs Verständnis von Information.

(Folie 100, analog 1/0, vielleicht interessant, aber wohl nicht relevant)
(Folie 101, Ambiguity)

Informationsübertragung von a nach b erfolgt in mehreren Schritten. (1) Zunächst muss die Information als Nachricht verpackt werden mithilfe einer Abbildung α, anschließend (2) muss diese Nachricht von a nach b übertragen werden, wobei **Störungen** auftreten können und (3) muss die Nachricht nun wieder mit einer Abbildung α^{-1} zurückübersetzt werden zu einer Information. Die Abbildungen α und α^{-1} sind dabei abgesprochen, sodass am Ende die Eingangsinformation äquivalent zur Ausgangsinformation ist.

(Folie 105f, Informationstreue)

4 Ziffern und Zahlen

Zahlen dienen als Abstraktion zum Zaählen, Rechenen und Speichern. Das erste **Stellenwersystem**, das Sexagesimalsystem mit der Basis 60 wurde vor etwa 40 000 Jahren von den Babyloniern entwickelt, bei einem Stellenwertsystem ist die Wertigkeit einer Ziffer abhängig von ihrer Position.

Radixdarstellung

Bei der Radixdarstellungbezeichnet b die Basis des jeweiligen Stellenwertsystems (10 beim Hexadezimalsystem, 2 Beim Dualsystem, etc.)
Die Menge der entsprechenden Ziffern ist dann gegeben durch: $\{0, 1, \ldots, b-1\}$

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i$$

n ist die Anzahl der benötigten Stellen.

4.1 Dualsystem

Das Dualsystem ist das Stellenwertsystem zu der Basis 2 und benötigt ca. dreimal mehr Stellen zur Darstellung einer Zahl und ist deswegen für Menschen “unbequem”. Aus technischer Sicht hingegen ist es besonders einfach zu implimentieren, da nur zwei verschiedene Fälle unterschieden werden müssen (bspw. zwei Spannungen, Beleuchtungsstärken etc.).
Die Vorteile von nur zwei verschiedenen Zuständen sind aus technischer Sicht, die geringe Anfälligkeit gegenüber von Rauschen und Störungen und die einfache und effiziente Umsetzung von Arithmetik.

Position	$Name_{10}$	$Zifferwert_{10}$	$Name_2$	$Zifferwert_2$
$\underline{1}$	Einer	$(0 \sim 9) * 10^0$	Einer	$(0 \sim 1) * 2^0$
$\underline{1}0$	Zehner	$(0 \sim 9) * 10^1$	Zweier	$(0 \sim 1) * 2^1$
$\underline{1}00$	Hunderter	$(0 \sim 9) * 10^2$	Vierer	$(0 \sim 1) * 2^2$
$\underline{1}\ 000$	Tausender	$(0 \sim 9) * 10^3$	Achter	$(0 \sim 1) * 2^3$
$\underline{1}0\ 000$	Zehntausender	$(0 \sim 9) * 10^4$	Sechzehner	$(0 \sim 1) * 2^4$
...	...	...	...	...

Tabelle 4.1: Potenztablle - Dualsystem

Stelle	Wert im Dualsystem	Wert im Dezimalsystem
2^0	1	1
2^1	10	2
2^2	100	4
2^3	1000	8
2^4	10000	16
2^5	100000	32
2^6	1000000	64
2^7	10000000	128
2^8	100000000	256
2^9	1000000000	512
2^{10}	10000000000	1024
2^{11}	100000000000	2048
2^{12}	1000000000000	4096
...	...	...

4.1.1 Aritmethik

Additionsmatrix

+	0	1
0	0	1
1	1	10

Schriftliche Addition Beispiel:

	1	0	1	1	1	0	=	46
+			1	0	1	0	=	10
Ü		1	1	1			=	
	1	1	1	0	0	0	=	56

Multiplikationsmatrix

$\cdot$	0	1
0	0	0
1	0	1

Schriftliche Multiplikation$= 46 \cdot 13$ Beispiel:

		1	0	1	1	1	0				$\cdot$	1	1	0	1	$= 46 \cdot 13$
		1	0	1	1	1	0					1				
			1	0	1	1	1	0					1			
				0	0	0	0	0	0					0		
					1	0	1	1	1	0					1	
$Ü_2$	1	1	1	10	1	1										
	1	0	0	1	0	1	0	1	1	0						$= 598$

4.2 Stellenwertsysteme

System	Basis	Zeichensatz	C-Schreibweise
Dual	2	$\{0, 1\}$	
Oktal	8	$\{0, 1, 2, 3, 4, 5, 6, 7\}$	Präfix: 0
Dezimal	10	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$	
Hexadezimal	16	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$	Präfix: 0x

Folie 133

4.3 Umrechnung zwischen Stellenwertsystemen

$$\begin{array}{ll} \text{Base16} & \overbrace{A}\ \overbrace{B}\ \overbrace{C}\ .\ \overbrace{6} \\ \text{Base2} & 101010111101\ .\ 0110 \\ \text{Base8} & \underbrace{5}\ \underbrace{2}\ \underbrace{7}\ \underbrace{5}\ .\ \underbrace{6} \end{array}$$

4.3.1 vorberechnete Potenztabellen

Beispiel Dezimalzahl in Dualzahl $Z = (163)_{10}$

$$\begin{array}{rrrr} & 163 & & & \\ - & 128 & 2^7 & & 10000000 \\ \hline & 35 & & & \\ - & 32 & 2^5 & + & 100000 \\ \hline & 3 & & & \\ - & 2 & 2^1 & + & 10 \\ \hline & 1 & & & \\ - & 1 & 2^0 & + & 1 \\ \hline & 0 & & & 1010\ 0011 \end{array}$$

$Z = (163)_{10} \leftrightarrow (1010\ 0011)_2$

Beispiel Dualzahl in Dezimalzahl $Z = (1010\ 0011)_2$

$$\begin{array}{rrrrr} & 10100011 & & & \\ - & 1100100 & 1 \cdot 10^2 & & 100 \\ \hline & 00111111 & & & \\ - & 111100 & 6 \cdot 10^1 & + & 60 \\ \hline & 11 & & & \\ - & 11 & 3 \cdot 10^0 & + & 3 \\ \hline & 0 & & & 163 \end{array}$$

$Z = (1010\ 0011)_2 \leftrightarrow (163)_{10}$

4.3.2 Divisionsrestverfahren

Beispiel Dezimalzahl in Dualzahl $Z = (163)_{10}$

$$
\begin{array}{rcrcrlcl}
163 & : & 2 & = & 81 & \text{Rest} & 1 & 2^0 \\
81 & : & 2 & = & 40 & \text{Rest} & 1 & 2^1 \\
40 & : & 2 & = & 20 & \text{Rest} & 0 & 2^2 \\
20 & : & 2 & = & 10 & \text{Rest} & 0 & 2^3 \\
10 & : & 2 & = & 5 & \text{Rest} & 0 & 2^4 \\
5 & : & 2 & = & 2 & \text{Rest} & 1 & 2^5 \\
2 & : & 2 & = & 1 & \text{Rest} & 0 & 2^6 \\
1 & : & 2 & = & 0 & \text{Rest} & 1 & 2^7 \\
\hline
\multicolumn{8}{c}{(163)_{10} \leftrightarrow (1010\ 0011)_2}
\end{array}
\quad \uparrow \text{Leserichtung}
$$

Beispiel Dualzahl in Dezimalzahl $Z = (1010\ 0011)_2$

$$
\begin{array}{rcrcrlrcrl}
(1010\ 0011)_2 & : & (1010)_2 & = & 1\ 0000 & \text{Rest} & (11)_2 & = & 3 & 10^0 \\
(1\ 0000)_2 & : & (1010)_2 & = & 1 & \text{Rest} & (110)_2 & = & 6 & 10^1 \\
(1)_2 & : & (1010)_2 & = & 0 & \text{Rest} & (1)_2 & = & 1 & 10^2 \\
\hline
\multicolumn{10}{c}{(1010\ 0011)_2 \leftrightarrow (163)_{10}}
\end{array}
\quad \uparrow \text{Leserichtung}
$$

4.3.3 Horner-Schema

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i = (\dots((a_{n-1} \cdot b + a_{n-2}) \cdot b + a_n - 3) \cdot b + \dots + a_1) \cdot b + a_0$$

Umwandlung Dezimal- in Dualzahl

(1) Darstellung als Potenzsumme

$$Z = (163)_{10} = (1 \cdot 10 + 6) \cdot 10 + 3$$

(2) Faktoren und Summanden im Zielzahlensystem

$$\begin{aligned} (10)_{10} &\leftrightarrow (1010)_2 \\ (6)_{10} &\leftrightarrow (110)_2 \\ (3)_{10} &\leftrightarrow (11)_2 \\ (1)_{10} &\leftrightarrow (1)_2 \end{aligned}$$

(3) Arithmetische Operationen

$$\begin{array}{ccccccccc} 1 & \cdot & 1010 & = & 1\,010 & & & & \\ & & & + & 110 & & & & \\ \hline & & & & 10\,000 & \cdot & 1010 & = & 1010\,0000 \\ & & & & & & & + & 11 \\ \hline & & & & & & & & 1010\,0011 \end{array}$$

Umwandlung Dual- in Dezimalzahl

(1) Darstellung als Potenzsumme

$$\begin{aligned} Z &= (1010\,0011)_2 \\ &= ((((((1 \cdot 10_2 + 0) \cdot 10_2 + 1) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2 + 1) \cdot 10_2 + 1 \end{aligned}$$

(2) Faktoren und Summanden im Zielzahlensystem

$$
\begin{aligned}
(10)_2 &\leftrightarrow (2)_{10} \\
(1)_2 &\leftrightarrow (1)_{10} \\
(0)_2 &\leftrightarrow (0)_{10}
\end{aligned}
$$

(3) Arithmetische Operationen

$$
\begin{array}{l}
1 \cdot 2 = 2 \\
\quad \underline{+\ 0} \\
\quad\ \ 2 \cdot 2 = 4 \\
\qquad\quad \underline{+\ 1} \\
\qquad\qquad 5 \cdot 2 = 10 \\
\qquad\qquad\qquad \underline{+\ \ 0} \\
\qquad\qquad\qquad\ \ 10 \cdot 2 = 20 \\
\qquad\qquad\qquad\qquad\quad \underline{+\ \ 0} \\
\qquad\qquad\qquad\qquad\qquad 20 \cdot 2 = 40 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \underline{+\ \ 0} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad 40 \cdot 2 = 80 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \underline{+\ \ 1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 81 \cdot 2 = 162 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \underline{+\ \ \ \ 1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ 163
\end{array}
$$

4.4 Präfixe

Dezimalsystem

Faktor	Name	Symbol
10^{24}	yotta	Y
10^{21}	zetta	Z
10^{18}	exa	E
10^{15}	peta	P
10^{12}	terra	T
10^{9}	giga	G
10^{6}	mega	M
10^{3}	kilo	K
10^{2}	hecto	H
10^{1}	deka	da

Faktor	Name	Symbol
10^{-24}	yocto	y
10^{-21}	tepto	z
10^{-18}	atto	a
10^{-15}	femto	f
10^{-12}	pico	p
10^{-9}	nano	n
10^{-6}	micro	μ
10^{-3}	milli	m
10^{-2}	centi	c
10^{-1}	dezi	d

Dualsystem

Faktor	Name	Symbol	
2^{80}	yobi	Yi	yottabinary
2^{70}	zebi	Zi	zettabinary
2^{60}	exbi	Ei	exabinary
2^{50}	pebi	Pi	petabinary
2^{40}	tebi	Ti	terabinary
2^{30}	gibi	Gi	gigabinary
2^{20}	mebi	Mi	megabinary
2^{10}	kibi	Ki	kilobinary

4.5 Festkommadarstellung

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i + \sum_{i=-m}^{-1} a_i \cdot b^i = \sum_{i=-m}^{n-1} a_i \cdot b^i$$

Alle Dualbrüche sind im Dezimalsystem exakt darstellbar, umgekehrt gilt nicht, dass alle Dezimahlbrüche exakt als Dualbrüche Darstellbar sind.

$B = 10$	$B = 2$
0.1	$0.0\overline{0011}$
0.2	$0.\overline{0011}$
0.3	$0.0\overline{1001}$
0.4	$0.\overline{0110}$
0.5	0.1
0.6	$0.\overline{1001}$
0.7	$0.1\overline{0110}$
0.8	$0.\overline{1100}$
0.9	$0.1\overline{1100}$

$B = 2$	$B = 10$
0.001	0.125
0.01	0.25
0.011	0.375
0.1	0.5
0.101	0.625
0.11	0.75
0.111	0.875

Potenztabelle

$2^{-1} = 0.5$	$2^{-7} = 0.0078125$
$2^{-2} = 0.25$	$2^{-8} = 0.00390625$
$2^{-3} = 0.125$	$2^{-9} = 0.001953125$
$2^{-4} = 0.0625$	$2^{-10} = 0.0009765625$
$2^{-5} = 0.03125$	$2^{-11} = 0.00048828125$
$2^{-6} = 0.015625$	$2^{-12} = 0.000244140625$

Umrechnung mittels Potenztabelle

$$\begin{aligned}
(0,3)_{10} &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + \dots \\
&= 2^{-2} + 2^{-5} + 2^{-6} + 2^{-9} + \dots \\
&= (0,0\overline{1001})_2
\end{aligned}$$

Umrechnung mittels Divisionsrestverfahren

$$\begin{array}{rcrcrcl}
2 \cdot & 0,59375 & = & 1,1875 & \rightarrow 1 & 2^{-1} & \\
2 \cdot & 0,1875 & = & 0,375 & \rightarrow 0 & 2^{-2} & \\
2 \cdot & 0,375 & = & 0,75 & \rightarrow 0 & 2^{-3} & \downarrow \text{Leserichtung} \\
2 \cdot & 0,75 & = & 1,5 & \rightarrow 1 & 2^{-4} & \\
2 \cdot & 0,5 & = & 1,0 & \rightarrow 1 & 2^{-5} & \\
\hline
\multicolumn{7}{c}{(0,59375)_{10} \leftrightarrow (0,10011)_2}
\end{array}$$

4.6 Negative Zahlen

4.6.1 Betrag und Vorzeichen

Bei dieser Notation wird das erste Bit als **Vorzeichenbit** interpretiert. Ist dieses 0, so ist das Vorzeichen $+$, andernfalls $-$.
Der Nachteil an diesem System ist, dass es die Null doppelt gibt und somit immer ein Sonderfall behandelt werden müsste.

0000	$+0$	1000	-0
0001	$+1$	1001	-1
0010	$+2$	1010	-2
...	...	...	...

4.6.2 Exzess-Codierung (Offset)

Für Exzess-`Offset` gilt: $z = c - \texttt{offset}$

Bei dieser Notation wird von der negativesten Zahl (Offset) im System bis zur höchsten Zahl hochgezählt. Der Bitvektor, welcher nur aus Nullen besteht 0000_2 ist demnach in der Exzess-8-Notation $(0-8)_{10} = -8_{10}$, also -8 und demnach, ausgehend von einer 4-Bit Breite, der Bitvektor $1111_2 = (15-8)_{10} = 7_{10}$.

Der Nachteil an diesem System ist, dass die Null abhängig vom Offset ist und somit nur in Exzess-0 ein Bitvektor mit ausschließlich Nullen ist.
Hingegen von Vorteil ist, dass ein Größenvergleich von zwei Zahlen einfach bleibt.

Bitmuster	Binärcode	Exzess-8	Exzess-6
0000	0	-8	-6
0001	1	-7	-5
0010	2	-6	-4
...	...	...	...
1111	15	7	9

4.6.3 Komplementdarstellung

Es gibt mehrere Arten der Komplementdarstellung, man nennt diese x-Komplement, wobei x in Abhängigkeit zur Basis b des Stellenwertsystems ist.

Eine Variante ist dabei das **b-Komplement**, welches in den meisten Computersystemen Anwendung findet. Im Dualsystem ist es also genauer das 2-Komplement, im Dezimalsystem das 10-Komplement.

Eine Zahl z in diesem System lässt sich mit folgender Gleichung errechnen:

$$K_b(z) = b^n - z, \qquad \text{für } z \neq 0$$
$$= 0, \qquad \text{für } z = 0$$

Dabei ist n die Anzahl der zu berücksichtigenden Vorkommastellen. Es gilt:

$$K_b(z) + z = b^n$$

Stellenwertschreibweise

$$z = -a_{n-1} \cdot b^{n-1} + \sum_{i=-m}^{n-2} a_i \cdot b^i$$

$b = 10$	$n = 4$	$K_{10}(3\ 763)_{10}$	$= 10^4 - 3\ 763_{10}$	$= 6\ 237_{10}$
	$n = 2$	$K_{10}(0.3763)_{10}$	$= 10^2 - 0.3763_{10}$	$= 99.6237_{10}$
	$n = 0$	$K_{10}(0.3763)_{10}$	$= 10^0 - 0.3763_{10}$	$= 0.6237_{10}$
$b = 2$	$n = 2$	$K_2(10.01)_2$	$= 2^2 - 10.01_2$	$= 01.11_2$
	$n = 8$	$K_2(10.01)_2$	$= 2^8 - 10.01_2$	$= 1111\ 1101.11_2$

Einfacher als mit der obigen Gleichung lässt sich das System durch Invertieren aller Bits/Ziffern und anschließendem Addieren von 1 an der niedrigsten Stelle (bei Fließkommazahl ist dies die letzte Nachkommastelle) herleiten:

$$\text{für } n = 2: \ -10.01_2 \quad \to 01.10_2 \quad + 00.01_2 \quad = 01.11_2$$
$$\text{für } n = 0: \ -0.3763_{10} \quad \to 0.6236_{10} \quad + 0.0001_{10} \quad = 0.6237_{10}$$

Eine andere Variante ist das **$b - 1$-Komplement** bzw. das 1-Komplement (Dual-) und das 9-Komplement (Dezimalsystem). Allerdings hat dieses System auch wieder den Nachteil, dass es zwei Nullen gibt. Es gilt:

$$K_{b-1}(z) = b^n - z - b^{-m}, \qquad \text{für } z \neq 0$$
$$= 0, \qquad \text{für } z = 0$$

Hierbei ist n ebenfalls die Anzahl der zu berücksichtigenden Vorkommastellen und hinkommend ist m die Anzahl der Nachkommastellen. Es gilt:

$$K_{b-1}(z) + z + b^{-m} = b^n$$

Diese Variante verhält sich ähnlich wie das b-Komplement, sodass sie einfach durch das Invertieren aller Bits/Ziffern erfolgt:

$$\text{für } n = 8: \ -1100\ 1001_2 \quad \to 0011\ 0110_2$$
$$\text{für } n = 2: \ -24\ 453_{10} \quad \to 75\ 546_{10}$$

```
  0   /     0  =  NaN
  1   /     0  =  Inf
 -1   /     0  =  -Inf
  1   /   Inf  =  0.0
Inf   +   Inf  =  Inf
Inf   -   Inf  =  NaN
Inf   *  -Inf  =  -Inf
Inf   +   NaN  =  NaN
Inf   *     0  =  NaN
sqrt(-1)       =  NaN
  0   +   NaN  =  NaN
NaN  ==   NaN  =  false
Inf   >   NaN  =  false
```

5 Arithmetik

5.1 Overflow

Sobald eine Zahl größer ist als die Wortlänge w, kommt es zu einem **Overflow**.
Für `unsigned` meint dies, dass (1) entweder das oberste Bit außerhalb der Wortlänge ist und damit verloren geht, sodass ein deutlich niedrigerer Wert heraus kommt oder (2) vom niedrigsten Wert (0) abgezogen wird, sodass ein höherer Wert heraus kommt als zu erwarten ist.

Für `signed` (2-Komplement) hingegen meint dies, dass das Vorzeichenbit überschrieben wird und somit bspw. durch Addition zweier sehr großer Zahlen eine negative Zahl heraus kommen kann und anders herum.
Erkennbar ist dies, wenn sich das Vorzeichen entgegen der Erwartung ändert.

Außerdem spricht man von einem Overflow, wenn man versucht eine Zahl der Wortlänge w_1 in die Wortlänge w_2 zu konvertieren (s. Sign Extension), wobei $w_2 < w_1$, sodass es möglich ist, dass ein signifikantes, oberes Bit verloren gehen kann.

5.2 Sign Extension

Sign Extension ist das Konvertieren einer Zahl der Wortlänge w_1 zur Wortlänge w_2 ($w_2 > w_1$, andernfalls kommt es möglicherweise zu Fehlern). Dabei wird jeweils das oberste Bit angeschaut und falls es 1 ist werden $w_2 - w_1$ Einsen vorgeschoben, andernfalls so viele Nullen, damit das Vorzeichen erhalten bleibt.

$$
\begin{array}{r|rcr}
\text{8-bit signed} & +6 & = & 0000\ 0110 \\
\text{16-bit signed} & +6 & = & 0000\ 0000\ 0000\ 0110 \\
& & & \\
\text{8-bit signed} & -2 & = & 1111\ 1110 \\
\text{16-bit signed} & -2 & = & 1111\ 1111\ 1111\ 1110
\end{array}
$$

5.3 Subtraktion

Die **Subtraktion** $a - b$ unterscheidet sich zwischen dem 1- und dem 2-Komplement; so addiert man bei beiden den ersten Operanden a mit dem Komplement des zweiten Operanden b, wobei beim 1-Komplement anschließend noch 1 aufs Ergebnis addiert wird.

5.4 Division (Integer Division)

Bei der Ganzzahldivision kommt beim Ergebnis grundsätzlich wieder ein Integer heraus, es werden also keine Nachkommastellen berechnet. Stattdessen wird ein Rest mitgegeben, mit dem weiter gerechnet werden kann.

$(125/5)_{10} = (111\ 1101/101)_2$

```
 1111101  /  101  =  0011001
 1                   0
 11                   0
 111                   1
-101
---------------------------------------
  10                    0
  101                    1
- 101
---------------------------------------
    01                    0
     10                    0
     101                    1
   - 101
---------------------------------------
        0                       0 (Rest)
```

5.5 Reziprokwert

Berechnung des multiplikativen Inversem durch: $y_{i+1} = y_i \cdot (2 - x \cdot y_i)$ Zur Berechnung wird ein geeigneter Schätzwert y_0 benötigt.
Beispiel: Inverses von 3; $y_0 = 0.5$:

$$\begin{aligned}
y_1 &= 0.5 \cdot (2 - 3 \cdot 0.5) && = 0.25 \\
y_2 &= 0.25 \cdot (2 - 3 \cdot 0.25) && = 0.3125 \\
y_3 &= 0.3125 \cdot (2 - 3 \cdot 0.3125) && = 0.33203125 \\
y_4 &= 0.3332824 \\
y_5 &= 0.3333333332557231 \\
y_6 &= 0.3333333333333333
\end{aligned}$$

5.6 Heron-Verfahren

Approximation von $\sqrt{x}$:

$$y_{n+1} = \frac{y_n + \frac{x}{y_n}}{2} \tag{5.1}$$

5.7 Informationstreue

Es gilt:

$$\forall x \in \texttt{float} : x^2 \geq 0$$

aber da dabei ein Overflow entstehen kann nicht:

$$\forall x \in \texttt{signed int} : x^2 \geq 0$$

Ebenso gilt:

$$\forall x, y, z \in \texttt{int} : (x + y) + z = x + (y + z)$$

aber da dabei die Genaugkeit verloren gehen kann nicht:

$$\forall x, y, z \in \texttt{float} : (x + y) + z = x + (y + z)$$

6 Zeichen und Text

6.1 Zeichensätze

6.1.1 ASCII

In ASCII wird jedes Zeichen mit 7 bit codiert. So lassen sich insgesamt $2^7 = 128$ Zeichen darstellen. Von diesen 128 Zeichen sind 95 Druckbar und 33 dienen als Steuerzeichen.

6.1.2 ISO-8859

Die Zeichensätze der ISO-8859 Familie erweitern den ASCII Zeichensatz um Sonderzeichen und Umlaute. Dabei wird jedes Zeichen mit 8 bit codiert, Folglich lassen sich maximal 256 Zeichen darstellen.
Das reichte allerdings nicht aus und folglich wurden für verschiedene Regionen verschieden Zeichensätze definiert; bspw. für Westeuropa Latin-1, für Mitteleuropa Latin-2 und so weiter.
So können innerhalb eines Sprachraums besser Dokumente verfasst werden, jedoch sind mehrsprachige Dokumente so trotzdem nicht wirklich realisierbar, da bspw. eine 8 bit-Sequenz in dem Zeichensatz Latin-1 möglicherweise ein anderes Zeichen in dem Zeichensatz Latin-3 repräsentiert.

6.1.3 Microsoft: Codepages

Die Microsoft Codepages sind ebenfalls eine Erweiterung des ASCII-Zeichensatzes auf einen 8 bit Code. Die Microsoft Codepages enthalten neben einigen Umlauten auch Grafiksymbole und waren der Zeichensatz des IBM-PC ab 1981.

6.1.4 Unicode

Die Motivation des Unicodezeichensatzes war, dass aufgrund der zunehemenden Vernetzung und Globalisierung ein Standard gebraucht wurde zum internationalen Datenaustausch bspw. über mehrsprachige Dokumente, der Untererstützung aller gängiger Sprachen (asiatische/orientalische Sprachen), etc.
Mittlerweile sind im Unicode-Zeichensatz (auch UCS [Universal Character Set] gennant) über 130 000 Zeichen enthalten. Darunter Sprachzeichen, Hieroglyphen, Satzzeichen, Währungen, Pfeile, mathematische Symbole, technische Symbole, Braille, Noten, Emojis, etc.

Schreibweise:
Ursprünglich wurde ein Zeichen mit 16 bit codiert, mittlerweise gibt es jedoch über 2^{16} Zeichen. Unicode Zeichen können zur Basis 16 als `U+xxxx` angegeben werden, so ergibt sich ein Bereich von `U+0000` bis `U+FFFF`; dabei entspricht der Bereich `U+0000` bis `U+007F` dem ASCII-Zeichensatz.

Der Nachteil an der Unicode-Codierung besteht darin, dass ein mit dieser Codierung verfasster Text doppelt so viel Speicherplatz benötigt, wie derselbe Text, wenn er in ISO-8859 codiert würde.
Aus diesem Grund wird häufig **UTF-8** oder **UTF-16** verwendet, da dort Zeichen nach ihrer häufigkeit in 1 bis 4 Byte codiert werden. Die ist besonders effizient bei der Codierung von Texten "westlicher" Sprachen.

UTF-8 Algorithmus

Unicode-Berreich (hexadezimal)	UTF-Codierung (binär)	Anzahl (benutzt)
`0000 0000-0000 007F`	`0*** ****`	128
`0000 0000-0000 07FF`	`110* **** 10** ****`	1 920
`0000 0000-0000 FFFF`	`1110 **** 10** **** 10** ****`	63 488
`0000 0000-0000 00FF`	`1111 0*** 10** **** 10** **** 10** ****`	bis 2^{21}

- Die ersten 128 Zeichen sind mit ASCII kompatibel
- Die Sonderzeichen westlicher Sprachen werden mit je 2 Byte codiert
- Die führende `1` markiert ein Multi-Byte Zeichen
- Die Anzahl der führenden Einsen gibt die Anzahl der Byte-Gruppen an

Ein Vorteil dieser eindeutigen Codierung ist, dass im Falle eines Informationsverlustes klar ist wo das nächste Zeichen anfängt.

7 Logische Operationen

7.1 Boole'sche Algebra

George Boole untersuchte und definierte 1850 eine logische Algebra, welche logische Aussagen mit genau zwei möglichen Werten nutze; true (wahr) und false (falsch). Hierüber wurden drei grundlegende Funktionen definiert, nämlich die Negation ($\neg x$, $\overline{x}$, $\sim x$), die Konjunktion ($x \land y$, $x \& y$) sowie die Disjunktion ($x \lor y$, $x \mid y$) und die exklusive Disjunktion (x xor y, $x \oplus y$, $x\hat{\ }y$).

Diese **Boole'sche Algebra** wurde in 1937 von Claude Shannon in Form von binärer Logik als Schaltfunktionen realisiert.

NOT(x)

x	
0	1
1	0

AND(x,y)

x \ y	0	1
0	0	0
1	0	1

OR(x,y)

x \ y	0	1
0	0	1
1	1	1

XOR(x,y)

x \ y	0	1
0	0	1
1	1	0

Binäre Funktionen

x =	0	1	0	1	Bezeichnung	Notationen			
y =	**0**	**0**	**1**	**1**					
	0	0	0	0	Nullfunktion	0			
	0	0	0	1	AND(x,y)	x∧y	x∩y	x&&y	
	0	0	1	0	Inhibition(x,y)	x<y			
	0	0	1	1	Identität(y)	y			
	0	1	0	0	Inhibition(x,y)	x>y			
	0	1	0	1	Identität(x)	x			
	0	1	1	0	XOR(x,y)	x⊕y	x≠y	x!=y	x xor y
	0	1	1	1	OR(x,y)	x∨y	x∪y	x||y	
	1	0	0	0	NOR(x,y)	$\overline{x \lor y}$	¬(x∪y)	!(x||y)	
	1	0	0	1	Äquivalenz(x,y)	x↔y	¬(x⊕y)	x=y	x==y
	1	0	1	0	NOT(x)	¬x	$\overline{x}$	!x	
	1	0	1	1	Implikation(x,y)	x→y	x≤y	x<=y	
	1	1	0	0	NOT(y)	¬y	$\overline{y}$	!y	
	1	1	0	1	Implikation(y,x)	x←y	x≥y	x=>y	
	1	1	1	0	NAND(x,y)	$\overline{x \land y}$	¬(x∩y)	!(x&&y)	
	1	1	1	1	Einsfunktion	1			

Eine Algebra wird gebildet durch einen 6-Tupel wie diesen:

$$(\{0,1\}, \vee, \wedge, \neg, 0, 1)$$

Hierbei ist das erste Element eine Menge mit Elementen, im Falle der Boole'schen Algebra sind dies genau zwei; das zweite Element ist die "Addition"; das dritte die "Multiplikation"; gefolgt vom "Komplement"; dem Nullelement (neutrales Element der Addition); und dem Einselement (neutrales Element der Multiplikation).

7.2 Logische Operationen, Ring der Algebra

Für den **Ring der Algebra** gelten ähnliche Rechenregeln zum Ring der ganzen Zahlen:

Kommutativgesetz	$x + y = y + x$	$x \vee y = y \vee x$
	$x \cdot y = y \cdot x$	$x \wedge y = y \wedge x$
Assoziativgesetz	$(x + y) + z = x + (y + z)$	$(x \vee y) \vee z = x \vee (y \vee z)$
	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$
Distributivgesetz	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
		$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
Identitäten	$x + 0 = x$	$x \vee 0 = x$
	$x \cdot 1 = x$	$x \wedge 1 = x$
Vernichtung	$x \cdot 0 = 0$	$x \wedge 0 = 0$
Auslöschung	$-(-x) = x$	$\neg(\neg x) = x$
Inverses	$x + (-x) = 0$	—
Komplement	—	$x \vee \neg x = 1$
	—	$x \wedge \neg x = 0$
Idempotenz	—	$x \vee x = x$
	—	$x \wedge x = x$
Absorption	—	$x \vee (x \wedge y) = x$
	—	$x \wedge (x \vee y) = x$
De Morgan	—	$\neg(x \vee y) = \neg x \wedge \neg y$
	—	$\neg(x \wedge y) = \neg x \vee \neg y$

Programmiersprachen wie Java und C unterstützen alle der obigen Operatoren und Regeln, teilweise in abweichender Syntax und mit optionaler Klammerung. Logische Ausdrücke werden dabei für gewöhnlich von links nach rechts ausgewertet und Compiler benutzen oftmals Shortcuts basierend auf den obigen Regeln um unnötige Laufzeit zu reduzieren, indem eine Berechnung eines Ausdrucks aufhört, sobald ihr Wert feststeht. So würde bspw. der folgende Ausdruck aufgrund des Vorkommens einer Disjunktion mit Tautologie ab der zweite Hälfte nicht mehr überprüft werden müssen:

$$\underbrace{\texttt{(x||!x)}}_{\texttt{= 1}}\ \texttt{||}\ \texttt{y}$$

7.3 Bitweise Operationen

Zusätzlich gibt es **bitweise Logik-Operatoren**, welche jedes gleichwertige Bit zweier Zahlen miteinander vergleicht:

`NOT`	`~x`	`0011 1010`→`1100 0101`
`AND`	`x&y`	`0011&1010`→ `0010`
`OR`	`x|y`	`0011|1010`→ `1011`
`XOR`	`x^y`	`0011^1010`→ `1001`

Dazu kommen dann noch die **Schiebeoperationen**, welche es ermöglichen, Binärzahlen im Speicher bitweise verschieben kann:

Art	Operatoren		Beschreibung
Shift-Left	`shl`	`x<<n`	Schiebt n Nullen von rechts, linke Bits gehen verloren
Logical Shift-Right	`shl`	`x>>>n`	Schiebt n Nullen von links, rechte Bits gehen verloren
Arithmetic Shift-Right	`shl`	`x>>n`	Schiebt n MSB von links, rechte Bits gehen verloren
Rotate-Left	`shl`	kein[1]	Schiebt um n Stellen nach links, rechts Bits werden von links wieder rein geschoben
Rotate-Right	`shl`	kein[2]	Schiebt um n Stellen nach rechts, linke Bits werden von rechts wieder rein geschoben

Multiplikation ist meist ein langsamer Prozess, wenn er überhaupt auf dem Prozessor verfügbar ist als Befehl. Daher ist es teilweise deutlich schneller, eine Kombination aus Shifts (meistens 1 Takt) und Addition (typisch 1 Takt) zu nutzen anstelle von langsamer Multiplikation (viele Takte). So kann bspw. `9x = (8+1)x` ersetzt werden durch `(x<<3)+x`, ein Shortcut, welcher von vielen Compilern automatisch erkannt und angewandt wird.

Die meisten Prozessoren unterstützen einen `bit_set` und einen `bit_clear` Befehl, welcher an einer Position `p` der Binärzahl `x` den Bit setzen (`1`) oder löschen (`0`) kann. Dies kann allerdings auch nach obigem Prinzip gemacht werden und so in einer Programmiersprache in der die beiden Befehle nicht definiert sind, definiert werden:

[1] Java: `Integer.rotateLeft(int x, int n)`
[2] Java: `Integer.rotateRight(int x, int n)`

```
public int bit_set(int x, int p)
{
    return x | (1 << pos);
}

public int bit_clear(int x, int p)
{
    return x & ~(1 << pos);
}
```

8 Codierung

Mit **Codierung** (alternativ Kodierung) ist eine Übersetzen von einer Repräsentation `X` zu einer anderen `Y` gemeint. Dabei muss eine Interpretation von Y nach X immer eindeut sein. Ist die Interpretation umkehrbar eindeutig, spricht man von **Umcodierung**.

Als **Codewörter** werden die Wörter der Repräsentation `Y` aus einem Zeichenvorrat `Z` bezeichnet. Die Menge aller dieser Codewörter wird **Code** gennant und, sollten alle die gleiche Länge haben, spricht man genauer von **Blockcode**.

Enthält der Zeichenvorrat genau genau zwei Zeichen, also **Binärzeichen**), dann sind alle Codewörter **Binärwörter** und der **Code** genauer **Binärcode**.

Man verwendet Codes für verschiedene Zwecke, darunter sind effizientere Informationsdarstellungsformen und -verarbeitung, Kompression und Reduktion sowie Sicherheitsaspekte. So können Informationen (bzw. deren Repräsentationen) verkleinert werden, Fehlererkennung und -korrektur vereinfacht usw. werden.in Java und C nicht als Operator verfügbar ssel nicht ohne weiteres dekodierbar ist.

Man unterscheidet bei der Codierung zwischen **Quellencodierung**, also der Anpassung an die Quelle, **Kanalcodierung**, also der Anpassung an die Strecke und **Verarbeitungscodierung**, also der Anpassung im Rechner.

Zur Darstellung von Codes gibt es verschiedenste Methoden, darunter bspw. Wertetabellen im Speicher, Codebäume, logische Gleichungen und algebraische Ausdrücke.

Text wird auf verschiedenste Arten und Weisen codiert (s. Kapitel 4 und Kapitel 6), kann aber außerdem auch in unterschiedlichen Formatierungen gespeichert werden, so bspw. für Schriftart, Größe, Farbe, usw., aber auch für Dinge wie Markup-Sprachen (HTML, Markdown, ...) und deutlich mehr.

Die folgenden Begriffe bezüglich Binärcodes sind im Glossar erläutert: Minimalcode, redundanter Code, Gewicht, komplementär, einschrittig & zyklisch, zyklisch einschrittig.

8.1 Einschrittiger Code

Einschrittiger Code C mit n Codewörtern lässt sich durch folgende rekursive Konstruktion durchführen.

(1)	Starten mit $n/2$ Codewörtern	$\{0, 1\}$
(2)	Anhängen führender `0` vor alle bisherigen Codewörter	$\{00, 01\}$
(3)	Kopieren der Codewörter aus (1) in invertierter Reihenfolge und anschließend vorangestellter `1`	$\{00, 01, 11, 10\}$

Durch Darstellung gibt es **Karnaugh-Veitch Diagramme**, welche eine zweidimensionale Darstellung von einschrittigem Code ermöglichen, wobei jede benachbarte Zelle genau einen Schritt, also mit Änderung eines Zeichens erreichbar ist:

	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

8.2 Graycode

Ein Dualwort in ein Graywort lässt sich durch folgende Konstruktion durchführen:

(1)	Das MSB bleibt hierbei erhalten	$0 \to 0;\ 1 \to 1$
(2)	Man betrachte von links nach rechts je ein paar: Bei Zeichenänderung wird Stelle 1, andernfalls 0	$0011 \to 0010$

Umgekehrt lässt sich aus einem Graywort ein Dualwort durch folgende Konstruktion durchführen:

(1)	Das MSB bleibt hierbei erhalten	$0 \to 0;\ 1 \to 1$
(2)	Man betrachte von links nach rechts je eine Stelle: Ist diese 0, muss das Zeichen dem vorherigen entsprechen Ist dies 1, muss es sich vom vorherigen unterscheiden $0011 \to 0010$	$0110 \to 0100$

8.3 Optimalcodes

Optimalcodes sind Codes von variabler Länge, wobei die häufigsten Symbole die kürzesten Codewörter bekommen. Ziel dabei ist es, die Menge an bpsw. zu übertragenden Daten zu reduzieren. Hierbei ist es, anders als bei Blockcode, allerdings nicht möglich, Codewörter durch Abzählen zu trennen, sondern nur durch eindeutige Übersetzung.

Damit dies funktioniert ist es entweder notwendig, die **Fano-Bedingung** einzuhalten oder Codewörter durch ein eindeutiges Codewort, einen Maker zu trennen.

Die Fano-Bedingung (nach R. M. Fano, 1961) ist hierbei, dass kein Wort aus einem Code den Anfang eines anderen Codewortes bilden darf, auch **Präfix-Eigenschaft** genannt, welcher eindeutig decodierbar ist. Zu Präfix-Code zählt auch Blockcode.

8.3.1 Shannon-Fano Coding

Zunächst werden die sogenannten Urwörter a_i nach ihren Wahrscheinlichkeiten $p(a_i)$ (startend bei der höchsten Wahrscheinlichkeit) sortiert.

Anschließend werden die sortierten Urwörter in zwei Gruppen eingeteilt, wobei beide Gruppen möglichst die gleiche Gesamtwahrscheinlichkeit haben sollen. Die eine Gruppe bekommt hierbei als erste Codewortstelle eine 0, die andere eine 1.

Dies wird nun wiederholt, bis es keine Teilgruppe mit mehr als einem Element gibt. Dieses Verfahren lässt sich gut als Baumdiagram darstellen, wobei jede Gabelung maximal zwei Äste haben kann, einen 0- und einen 1-Ast.

Diese Codierung ist allerdings **nicht eindeutig** und funktioniert besser, je größer die Anzahl an Urwörtern ist und gar nicht, wenn ein Urwort eine Wahrscheinlichkeit $p(a_i) \geq 0.5$ aufweist.

8.3.2 Huffman Coding

Zunächst werden auch hier die Urwörter a_i nach ihren Wahrscheinlichkeiten $p(a_i)$ (allerdings startend bei der niedrigsten Wahrscheinlichkeit) sortiert.

Nun werden schrittweise die zwei Wörter der niedrigsten Wahrscheinlichkeit durch ein neues ersetzt und zurück in die Menge sortiert.

Dies wird nun wiederholt, bis es nur noch zwei Wörter in der Menge gibt. Auch hier lässt sich anschließend ein Baumdiagramm nach gleichem Prinzip erstellen, wobei je ein zusammengefasstes Wörtern aufgedröselt werden kann zu einer Verzweigung mit je zwei weiteren (zusammengefassten) Wörtern.

Dieses Verfahren ergibt die kleinstmöglichen mittleren Codewortlängen und wird daher für vieles genutzt, bspw. für JPEG, MPEG, etc. Aber auch dieser Verfahren funktioniert nicht, wenn ein Urwort eine Wahrscheinlichkeit $p(a_i) \geq 0.5$ aufweist.

Auf Basis dieses Verfahrens gibt es auch das sogenannte **Dynamic Huffman Coding** (nach Knuth, 1985), wobei ein Decoder den Codebaum bei jedem decodierten Zeichen anpasst.

8.3.3 Kraft-Ungleichung

Dies Kraft-Ungleichung (nach Kraft, 1949) ist eine notwendige sowie hinreichende Bedingung für die Existenz eines eindeutig decodierbaren Codes C mit s Elementen der Codelänge l_s (sortiert von klein nach groß) über einem q-nären Zeichenvorrat F:

$$\sum_{i=1}^{s} \frac{1}{q^{l_i}} \leq 1$$

8.4 Informationstheorie

Ein Symbol A_i wird zu einer Wahrscheinlichkeit p_i empfangen, wobei nicht gemeint ist, ob es korrekt übertragen wird, sondern dass genau dieses Zeichen übertragen wird. Hierbei schließen sich alle A_i gegenseitig aus.

Vor dem Empfang gibt es eine **Ungewissheit** über das Kommende. Je kleiner nun p_i ist, desto größer wird **der Informationsgewinn / die Überraschung**.

Der **Informationsgehalt/Entscheidungsgehalt** I eines Ereignisses A_i mit der Wahrscheinlichkeit p_i ist eine messbare, additive Größe der **Einheit 1 Bit**, ausgedrückt durch folgende Gleichung:

$$\begin{aligned} I(A_i) &= \log_2\left(\frac{1}{p_i}\right) \\ &= -\log_2(p_i) \end{aligned}$$

8.4.1 Entropie

Die durchschnittliche Information (auch Erwartungswert, mittlerer Informationsgehalt) beim Empfang eines Symbols bezeichnet man als **Entropie** des Systems.

Für die Summe aller Wahrscheinlichkeiten p_i gilt:

$$\sum_i p_i = 1$$

Die Entropie H lässt sich nun als Erwartungswert des Informationsgehaltes $I(A_i)$ wie folgt berechnen:

$$\begin{aligned} H &= E\{I(A_i)\} \\ &= \sum_i p_i \cdot I(A_i) \\ &= \sum_i p_i \cdot \log_2\left(\frac{1}{p_i}\right) \\ &= -\sum_i p_i \cdot \log_2(p_i) \end{aligned}$$

Es gelten folgende Eigenschaften:

- $H(p_1, \ldots, p_n)$ ist maximal, falls $p_i = 1/n$ für $1 \leq i \leq n$)
- H ist symmetrisch, falls für jede Permutation φ von $1, \ldots, n$ gilt: $H(p_1, \ldots, p_n) = H(p_{\varphi(1)}, \ldots, p_{\varphi(n)})$
- $H(p_1, \ldots, p_n) \geq 0$ mit $H(0, 0, \ldots, 0, 1, 0, \ldots, 0, 0) = 0$

- $H(p_1, \ldots, p_n, 0) = H(p_1, \ldots, p_n)$
- $H(1/n, \ldots, 1/n) \leq H(1/(n+1), \ldots, 1/(n+1))$
- H ist stetig in seinen Argumenten
- Additivität; seien $n, m \in \mathbb{N}\setminus\{0\}$:
  $H(\frac{1}{n \cdot m}, \ldots, \frac{1}{n \cdot m}) = H(\frac{1}{n}, \ldots, \frac{1}{n}) + H(\frac{1}{m}, \ldots, \frac{1}{m})$

Der **mögliche Informationsgehalt** H_0 ist durch die Symbolcodierung festgelegt und wie folgt definiert:

$$H_0 = \sum_i p_i \cdot \log_2 (q^{l_i})$$

Wobei für Dualcodes gilt:

$$H_0 = \sum_i p_i \cdot l_i$$

Und für binäre Blockcodes mit Wortlänge N bits:

$$H_0 = N$$

8.4.2 Redundanz

Die Redundanz R lässt sich wie folgt berechnen

$$R = H_0 - H$$

Dabei ist H_0 wieder der mögliche Informationsgehalt und H die Entropie.
Die **relative Redundanz** berechnet sich indem die Redundanz durch den möglichen Informationsgehalt dividiert wird:

$$r = \frac{R}{h_0} = \frac{H_0 - H}{H_0}$$

Binäre Blockcodes mit einer Wortlänge von N bits haben einen mittleren Informaionsgehalt von $H_0 = N$ Es gilt:

$$\begin{aligned} R &= H_0 - H \\ &= H_0 - \left(-\sum_{i=1}^{m} p(a_i) \cdot \log_2 \Big(p(a_i) \Big) \right) \\ &= N + \sum_{i=1}^{m} p(a_i) \cdot \log_2 \Big(p(a_i) \Big) \end{aligned}$$

8.4.3 Kanalkapazität

Die **Kanalkapazität** ist ein Maß dafür, wie wahrscheinlich es ist, dass ein Zeichen bei der Übertragung verfälscht wird und ist gegeben durch:

$$C = 1 - H(F)$$

Dabei ist $H(F)$ die Entropie des Fehlverhaltens.

Bei einem binärem symmetrischem Kanal sind die 1 und 0 gleich wahrscheinlich.
Die Wahrscheinlichkeit P, dass bei einer Übertragung aus einer 1 eine 0 wird ist gleich der Wahrscheinlichkeit, dass aus einer 0 eine 1 wird, daher gibt ergibt sich für die Entropie des Fehlverhaltens:

$$H(F) = P \cdot \log_2\left(\frac{1}{P}\right) + (1 - P) \cdot \log_2\left(\frac{1}{1 - P}\right) \tag{8.1}$$

Sei $P = 0.5$, so beträgt die Kanalkapazität 0, die empfangene Bitsequenz lässt sich dann nicht mehr von einer zufälligen unterscheiden.

Shannon-Theorem
Falls die Übertragungsrate R kleiner als $C(P)$ ist, findet man zu jedem $\varepsilon > 0$ einen Code C mit der Übertragungsrate $R(C)$ und $C(P) \geq R(C) \geq R$ und der Fehlerdecodierwahrscheinlichkeit $< \varepsilon$
- Script "64-040- Modul InfB-RSB: Rechnerstrukturen und Betriebssysteme", Folie 404

Mit anderen Worten, wenn Kanalkapazität C größer der Übertragungsrate R ist können beliebig zuverlässige Codes gefunden werden.

8.4.4 Fehlertypen

Es gibt bspw. folgende Fehlertypen:

- Verwechlung eines Zeichens
- Vertauschen benachbarter Zeichen
- Vertauschen entfernter Zeichen
- Zwillings-/Bündelfehler

8.4.5 Begriffe zur Fehlerbehandlung

Blockcode: k-Informationsbits werden in n-Bits codiert.

Faltungscode: Ein Bitstrom wird in einen Codebitstrom einer höheren Bitrate codiert. Dabei erzeugt der Bitstrom eine Folge von Automatenzuständen und wird decodiert mithilfe bedingter Wahrscheinlichkeiten bei Zustandsübergängen. Sie sind prinzipiell linear, aber nicht nach der folgenden Definition.

linearer (n, k)-Code: Ein k-dimensionaler Unterraum des $\mathrm{GF}(2)^n$.

modifizierter Code: Systematische Veränderung mindestens einer Stelle eines linearen Codes, also invertiert im GF(2). Null- und Einsvektor gehören nicht mehr zum Code.

nichtlinearer Code: Weder linear, noch modifiziert.

Galoisfeld: GF(2); Galois-Feld mit zwei Elementen; ein Körper aus den zwei Verknüpfungen $\wedge$ (`AND`, Multiplikation) und $\oplus$ (`XOR`, Addition mod2) und dem additiven Inverses $x \oplus x = 0$.

systematischer Code: Wenn eine zu codierende Information direkt im Codewort (als Substring) enthalten ist.

zyklischer Code: Blockcode, bei dem für jedes Codewort gilt, dass auch alle zyklischen Verschiebungen dessen, also Rotationen (`rol`, `ror`, etc.) Codeworte sind.

8.4.6 Verfahren zur Fehlerbehandlung

Automatic Repeat Request: Ein fehlerhaftes Symbol wird vom Empfänger erkannt und fordert vom Sender eine Übertragung an.
Der Nachteil an dieser Fehlerbahndlung ist, dass eine bidirektionale Kommunikation erfoderlich ist und dies besonders bei Echtzeitanforderungen und großen Entfernungen unpraktisch ist.

Vorwärtsfehlerkorrektur (Forward Error Correction, FEC): Durch bspw. Prüfziffern wird die Redundanz erhöht. Dadurch kann der Empfänger fehlerhafte Codewörter erkennen und selbstständig korrigieren.

Die beiden Verfahren lassen sich auch verknüpfen.

Hamming-Abstand

Der **Hamming-Abstand** gibt die Anzahl der Stellen an, an denen sich zwei gleichlange Binärcodewörter unterscheiden.
Das **Hamming-Gewicht** eines Binärcodeworts gibt den Hamming-Abstand des Worts zum Null-Wort an, also die Anzahl der Einsen.

Um eine fehlerhafte Übertragung und eine Korrektur zu ermöglichen ist Redundanz notwendig.
Werden die Codewörter so gewählt, dass alle paarweise mindestens einen Hamming-Abstand von d aufweisen, so ist eine:

- Fehlererkennung bis zu $(d-1)$ fehlerhaften Stellen
- Fehlerkorrektur bis zu $\left(\frac{d-1}{2}\right)$ fehlerhaften Stellen

möglich.

Zur Fehlererkennung und Fehlerkorrektur werden den Daten Prüfinformationen hinzugefügt.

Paritätscode

Bei dem **Paritätscode** wird ein Paritätsbit an das Binärcodewort angehängt. Bei der geraden Patität, wird das Paritätsbit so angehängt, dass die Anzahl der Einsen gerade ist, bei der ungeraden Parität hingegen so, dass die Anzahl der Einsen gerade ist.
Der Paritätscode ist die einfachste Methode zur Erkennung von Einbitfehlern.
In der Praxis kommt eher die ungerade Parität zum Einsatz, damit mindestens eine 1 gesendet wird. So wird bspw. auch ein Strom- oder Systemausfall als Fehler erkannt.

Der Hamming-Abstand zweier Wörter im Paritätscode beträgt immer mindestens $d = 2$.

Zweidimensionale Parität

Wird die Parität zweidimensional verwendet, so lassen sich 1-bit Fehler finden und korrigieren. Dazu werden die Daten in einer Informationsmatrix angeordnet und für jede Zeile und jede Spalte ein Paritätsbit angefügt. Dies kann wie folgt aussehen:

H	1	0	0	1	0	0	0	0
A	1	0	0	0	1	0	1	0
M	1	1	0	1	1	0	1	0
M	1	0	0	1	1	0	1	0
I	0	0	0	1	0	0	1	1
N	1	0	0	1	1	1	0	0
G	1	0	0	0	1	1	1	0
	1	0	0	1	0	0	0	1

H	1	0	0	1	0	0	0	0	
A	1	0	0	0	1	0	1	0	1
M	1	1	0	1	1	0	1	0	
M	1	0	0	1	1	0	1	0	
I	0	0	0	1	0	0	1	1	
N	1	0	0	1	1	1	0	0	
G	1	0	0	0	1	1	1	0	
	1	0	0	1	0	0	0	1	
					1				

Auf diese Weise lassen sich 1-bit Fehler immer korrigieren und teilweise auch n-bit Fehler. Dieses Prinzip lässt sich auch n-Dimensional verwenden.

Hamming-Code

Bei einem (N, n)-Hemming-Code wird ein Datenwort mit n-bits $(d_1, \ldots, d_n)$ um k-Prüfbits $(p_1, \ldots, p_k)$ ergänzt, sodass ein Codewort mit $N = n + k$ bits entsteht.
Hierbei ist eine Fehlerkorrektur von $2^k \geq N + 1$ gewährleistet, wobei 2^k die Anzahl an Kombinationen mit k-Prüfbits ist, 1 der fehlerfreie Fall und N die zu markierenden Bitfehler sind.

Ein Codewort eines (N, n)-Hemming-Codes lässt sich wie folgt bilden:

1. Bestimmen des kleinsten k mit $n \leq 2^k - k - 1$
2. Prüfbits an Bitpositionen $2^0, \ldots, 2^{k-1}$
3. Originalbits an den übrigen Positionen

Ein Prüfbit i berechnet sich als $\bmod 2$-Summe (`XOR`) der Bits, dessen i-te Position 1 ist.
Für folgenden Hamming-Code:

Position	1 (0001)	2 (0010)	3 (0011)	4 (0100)	5 (0101)	6 (0110)	7 (0111)	8 (1000)	9 (1001)	…
Bit	p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4	d_5	…

ergeben sich folgende Errechnungen der Prüfbits:

$$
\begin{aligned}
p_1 &= d_1 \oplus d_2 \oplus d_4 \oplus d_5 \oplus \ldots \\
p_2 &= d_1 \oplus d_3 \oplus d_4 \oplus d_6 \oplus \ldots \\
p_3 &= d_2 \oplus d_3 \oplus d_4 \oplus d_8 \oplus \ldots \\
p_4 &= d_5 \oplus d_6 \oplus d_7 \oplus d_8 \oplus \ldots \\
&\ldots
\end{aligned}
$$

Bei einem (N, n)**-Hamming-Code** benötigt man N Codebits für je n Datenbits; dabei handelt es sich um einen linearen (N, n)-Block-Code. Weiter gibt es für jeden Hamming-Code eine so genannte **Generatormatrix** mit welcher sich durch Multiplikation mit den Datenbits ein Codewort generieren lässt, welches sich mit einer **Prüfmatrix** überprüfen lässt.

Für einen $(7, 4)$-Hamming-Code gibt es folgende Generatormatrix G und Prüfmatrix H:

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}; \quad H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Daraus ergibt sich für die Datenbits $(1, 1, 0, 1)$ folgendes Codewort:

$$c = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Dies lässt sich nun durch Multiplikation mit der Prüfmatrix überprüfen, wobei für ungültige Codewörter gilt, dass kein Nullvektor herauskommt:

$$H \cdot c = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Sollte das Ergebnis nun kein Nullvektor sein, so lässt sich die Fehlerstelle einfach rausfinden, denn eine der Spalten der Prüfmatrix entspricht dem herausgekommenden Vektor, sodass der Spaltenindex dem Zeilenindex des Codewortvektors entspricht.

8.4.7 Binärpolynome

Jedem Wort mit n bits $(d_1, \ldots, d_n)$ lässt sich mindestens ein Polynom über dem Körper $\{0, 1\}$ zuordnen. Außerdem kann man mit diesen Polynomen rechnen (Addition, Subtraktion, Multiplikation, Division).
Den Wort 101 1001 lassen sich bspw. folgende Polynome zuordnen:

$$
\begin{aligned}
&1x^6 + 0x^5 + 1x^4 + 1x^3 + 0x^2 + 0x^1 + 1x^0 \\
&x^6 + x^4 + x^3 + x^0 \\
&x^0 + x^2 + x^3 + x^6 \\
&x^0 + x^{-2} + x^{-3} + x^{-6} \\
&\ldots
\end{aligned}
$$

8.4.8 Zyklische Codes (CRC)

Cyclic Redundancy Check (CRC); Kapitel 7 - Codierung, Folien 440ff.

1. Wahl eines Generatorpolynoms g $n \deg(g)$ ist der Grad des Polynoms.

   Bsp. $g =$ `110101` $= x^5 + x^4 + x^2 + 1$ (in der Realität deutlich größer.)
   $\deg(g) = 5$

2. Nutzdaten (auch Frame genannt). Meist sind die Nutzdaten deutlich größer als das Generatorpolynom.

   Bspw. `11011`

3. Multiplikation des Frames mit x^n Das ist nichts anderes als n-Nullen an das Frame anzuhängen. Diese Nullen am Ende des Frames werden Nullpaket genannt.

   in unserem Beispiel: Multiplikation mit x^5 : `1101100000`

4. Dividieren durch g, wobei nur der Rest interessant ist.

$$
\begin{array}{cccccccccccccccccc}
 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & : & 1 & 1 & 0 & 1 & 0 & 1 \\
 & 1 & 1 & 0 & 1 & 0 & 1 \\
\oplus & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 & & & & \oplus & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
 & & & & & 0 & 0 & 0 & 1 & 0 & 1 \\
 & & & & & & & & 1 & 0 & 1 \\
\hline\hline
\end{array}
$$

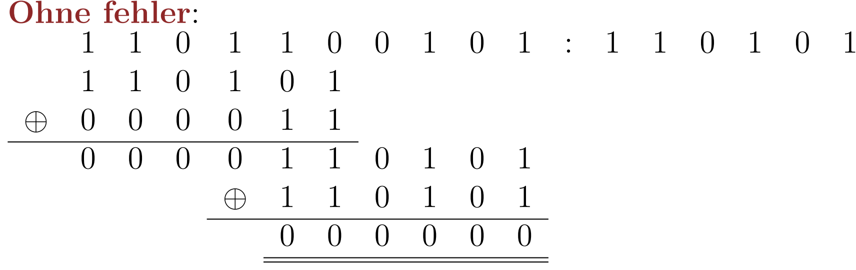
   `101` ist der Rest

5. Der Rest wird jetzt ins Nullpaket geschrieben. Dadurch bekommen wir eine Bitfolge, welche durch das Generatorpolynom geteilt einen Rest von 0 ergibt

   In unserem Beispiel bekommen wir dadurch folgende Bitfolge: `1101 1001 01`

6. wird die Bitfolge nun versand, teilt der Empfänger die Bitfolge durch g. Wobei wieder nur der Rest interessant ist.

   Ohne fehler:



$$
\begin{array}{cccccccccccccccccc}
 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & : & 1 & 1 & 0 & 1 & 0 & 1 \\
 & 1 & 1 & 0 & 1 & 0 & 1 \\
\oplus & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
 & & & & \oplus & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline\hline
\end{array}
$$

Mit fehler:

$$
\begin{array}{cccccccccccccccccc}
 & 1 & \color{red}{0} & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & : & 1 & 1 & 0 & 1 & 0 & 1 \\
 & 1 & 1 & 0 & 1 & 0 & 1 & & & & & & & & & & & \\
\oplus & 0 & 1 & 0 & 0 & 1 & 1 & & & & & & & & & & & \\
\hline
 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & & & & & & & & & & \\
 & \oplus & 1 & 1 & 0 & 1 & 0 & 1 & & & & & & & & & & \\
\hline
 & & 0 & 1 & 0 & 0 & 1 & 1 & 1 & & & & & & & & & \\
 & & \oplus & 1 & 1 & 0 & 1 & 0 & 1 & & & & & & & & & \\
\hline
 & & & 0 & 1 & 0 & 0 & 1 & 0 & 0 & & & & & & & & \\
 & & & \oplus & 1 & 1 & 0 & 1 & 0 & 1 & & & & & & & & \\
\hline
 & & & & 0 & 1 & 0 & 0 & 0 & 1 & 1 & & & & & & & \\
 & & & & \oplus & 1 & 1 & 0 & 1 & 0 & 1 & & & & & & & \\
\hline
 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & & \\
 & & & & & & 1 & 0 & 1 & 1 & 0 & & & & & & & \\
\hline\hline
\end{array}
$$

Da der Rest ungleich 0 ist, gab es ein Übertragungsfehler.

9 Schaltfunktionen

9.1 Definition

Eine **Schaltfunktion** meint eine eindeutige Zuordnungsvorschrift (Abbildung) f, die jeder Wertekombination von **Schaltvariablen** $(b_1, \ldots, b_n)$ einen Wert zuweist:

$$y = f(b_1, \ldots, b_n) \in \{0, 1\}$$

Hierbei ist eine **Schaltvariable** eine Variable mit endlich vielen möglichen Werten (typisch binäre Schaltvariablen).
Die **Ausgangsvariable** ist eine Schaltvariable, die den Wert y der Funktion am Ausgang annimmt.

Sei M eine Menge von Verknüfpungen über GF(2), so ist diese **funktional vollständig**, wenn die Funktionen $f, g \in T_2$ mit

$$f(x_1, x_2) = x_1 \oplus x_2$$
$$g(x_1, x_2) = x_1 \wedge x_2$$

ausschließlich mit den Verknüfpungen in M geschrieben werden können.

Funktional vollständige Mengen M sind:

- Boole'sche Algebra: $\{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$
- Reed-Muller Form: $\{\texttt{AND}, \texttt{XOR}, \texttt{1}\}$
- technisch relevant: $\{\texttt{NAND}, \texttt{NOR}\}$

9.2 Darstellung

Schaltfunktionen lassen sich unterschiedlich beschreiben, so sind textuelle Beschreibungen (formale Notationen, Schaltalgebra, etc.), tabellarische Beschreibungen (Funktionstabellen, KV-Diagramme, etc.) und graphische Beschreibungen (Kantorovic Baum, Schaltbild, etc.) sowie Verhaltensbeschreibungen ("Was?") und Strukturbeschreibungen ("Wie?") übliche Optionen.

9.2.1 Funktionstabellen

Diese bestehen aus Spalten für alle Eingänge x_i und den Ausgang $y = f(x)$. Die Zeilen werden dabei nach der Größe des Binärcodes sortiert:

x_3	x_2	x_1	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Die Kurschreibweise hiervon entspricht den Funktionswerten, also

$$f(x_3, x_2, x_1) = \{0, 0, 1, 1, 0, 0, 1, 0\}$$

Das Problem an Funktionstabellen ist, dass diese bei n Eingängen 2^n Einträge hat und somit exponentiell mit n wächst.

9.2.2 Verhaltensbeschreibung

Hier wird eine Funktion als Text über ihr Verhalten beschrieben. Das Problem hierbei ist, dass Formulierungen in Umgangssprache schnell mehrdeutig werden. Daher bieten sich hierfür logische Ausdrücke an (s. Programmiersprachen) oder der Einsatz spezieller Beschreibungssprachen (Verilog, VHDL, SystemC, etc.).

Die Mehrdeutigkeit kann bspw. entstehen durch Nutzung von Begriffen wie "oder", welche als `OR`, aber auch als `XOR` verstanden werden können.

9.2.3 Strukturbeschreibung

Hierbei wird die konkrete Realisierung einer Schaltfunktion spezifiziert. Dies können vollständig geklammerte algebraische Ausdrücke sein (bspw. $f = x_1 \oplus (x_2 \vee x_3)$), Datenflussgraphen, Schaltpläne, etc.

9.3 Normalformen

Da jede Funktion auf belieb viele Arten beschrieben werden kann, gibt es Standardformen, welche eine eindeutige Repräsentation für einfache Überprüfung auf Übereinstimmung ermöglichen.

Bspw. lassen sich alle ganzrationalen Funktionen darstellen durch

$$f(x) = \sum_{i=0}^{n} a_i x^i$$

Die **Normalform einer Boole'schen Funktion** ist analog zur Potenreihe definiert als Summe über die Koeffizienten $\hat{f} \in \{0,1\}$ und Basisfunktionen $\hat{B}_i$ einer Basis des T^n:

$$f = \sum_{i=1}^{2^n} \hat{f}_i \hat{B}_i, \quad \hat{f}_i \in \mathrm{GF}(2)$$

Sei V eine vollständige Menge der Verknüfpungen von $\{0,1\}$ und $\oplus, \otimes \in V$ assoziativ. Eine Form wird als Normalform und die Menge der $\hat{B}_i$ als Basis bezeichnet, wenn sich alle $f \in T^n$ in folgender Form schreiben lassen:

$$f = (\hat{f}_1 \otimes \hat{B}_1) \oplus \ldots \oplus (\hat{f}_{2^n} \otimes \hat{B}_{2^n})$$

Dabei sind es 2^n Basisfunktionen $\hat{B}_i$ und 2^{2^n} mögliche Funktionen f.

9.3.1 Disjunktive Normalform (DNF)

Bei der disjunktiven Normalform (auch kanonische disjunktive Normalform oder sum-of-products) werden alle Minterme m, welche den Funktionswert 1 haben disjunktiv verknüpft:

$$f = \bigvee_{i=1}^{2^n} \hat{f}_i \cdot m(i)$$

9.3.2 Allgemeine disjunktive Form

Bei der disjunktiven Form (sum-of-products) handelt es sich nicht um eine Normalform, da diese nicht eindeutig ist. Es handelt sich hingegen um die Zusammenfassung (Minimierung) der DNF.

9.3.3 Konjunktive Normalform (KNF)

Bei der konjunktiven Normalform (auch kanonische konjunktive Normalform oder product-of-sums) werden alle Maxterme μ, welche den Funktionswert 0 haben konjunktiv verknüpft:

$$f = \bigwedge_{i=1}^{2^n} \hat{f}_i \cdot \mu(i)$$

9.3.4 Allgemeine konjunktive Form

Bei der konjunktiven Form (product-of-sums) handelt es sich, wie bei der allgemeinen disjunktiven Form, nicht um eine Normalform, da diese nicht eindeutig ist. Es handelt sich hingegen um die Zusammenfassung (Minimierung) der KNF.

9.3.5 Reed-Muller Form

Die Reed-Muller Form ist eine Normalform, welche durch die additive (`XOR`) Verknüpfung aller Reed-Muller-Terme mit dem Funktionswert 1 erfolgt:

$$f = \bigoplus_{i=1}^{2^n} \hat{f}_i \cdot \mathrm{RM}(i)$$

Dabei sind $\mathrm{RM}(i)$ die folgenden Reed-Muller Basisfunktionen:

$\{1\}$	0 Variablen
$\{1, x_1\}$	1 Variable
$\{1, x_1, x_2, x_2x_1\}$	2 Variablen
$\{1, x_1, x_2, x_2x_1, x_3, x_3x_1, x_3x_2, x_3x_2x_1\}$	3 Variablen
...	...
$\{\mathrm{RM}(n-1), x_n \cdot \mathrm{RM}(n-1)\}$	n Variablen

Hierbei meint bspw. $x_2x_1 \leftrightarrow (x_2 \wedge x_1)$.

Für die **Umrechnung** von einem Ausdruck in die Reed-Muller Form gibt es folgende Gleichungen:

- $\overline{a} = a \oplus 1$
- $a \vee b = a \oplus b \oplus (a \wedge b)$
- $a \oplus a = 0$

Transformationsmatrix

Um eine Funktion f in die Reed-Muller Form umzurechnen kann die Transformationsmatrix A benutzt werden. Die Multiplikation der Transformationsmatrix mit den Funktionswerten der Funktion F als Vektor gibt den Koeffizientenvektor r der Reed-Muller Form.
Die Multiplikation der Transformationsmatrix A mit dem Koeffizientenvektor r gibt wieder die Funktionswerte als Vektor f

$$r = A \cdot f \qquad \text{und} \qquad f = A \cdot r$$

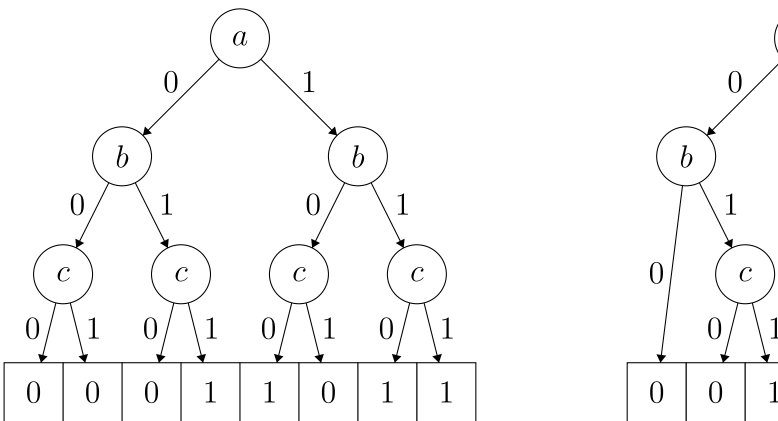
Dies gilt, da $A \cdot A$ die Einheitsmatrix $\mathbb{1}$ ist.

$$
\begin{aligned}
A_0 &= \begin{pmatrix} 1 \end{pmatrix} \\
A_1 &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \\
&\dots \\
A_n &= \begin{pmatrix} A_{n-1} & 0_{n,n} \\ A_{n-1} & A_{n-1} \end{pmatrix}
\end{aligned}
$$

9.4 Entscheidungsbäume und OBDDs

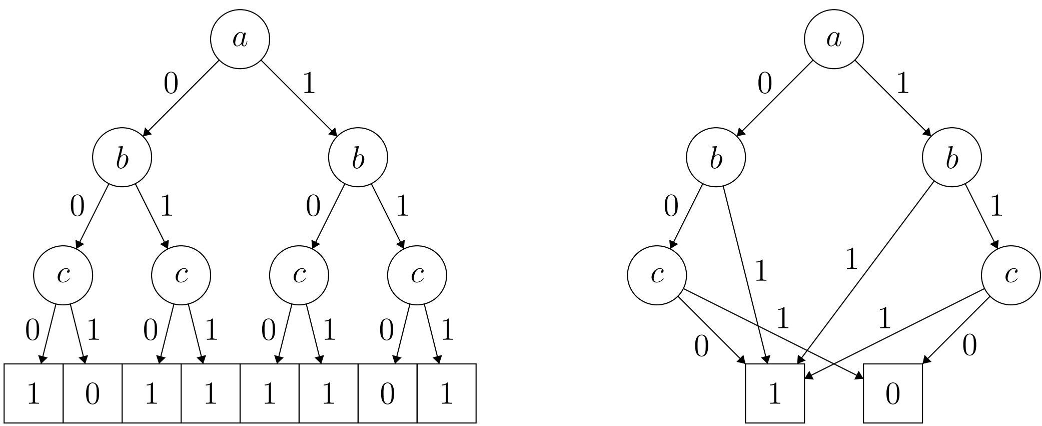
Schaltfunktionen lassen sich als Baum/Graph darstellen, wobei jeder Knoten einer Variable zugeordnet ist und jeder Knoten einem `if-else` entspricht.

Der Entscheidungsbaum der Funktion eines Multiplexers $f(a, b, c) = (a \wedge \overline{c}) \vee (b \wedge c)$ sieht wie folgt aus (links vollständig, rechts gekürzt):



Als Variante des Entscheidungsbaums gibt es **Reduces Ordered Binary-Decision Diagrams (ROBDD)**, welche eine Normalform für eine Funktion sind. Hierbei werden vorab die Variablenordnung gewählt (ordered) und redundante Knoten entfernt (reduced). Sie ermöglichen eine kompakte Darstellungsmöglichkeit, wobei die Anzahl der Knoten bei n Variablen zwischen $\mathcal{O}(n)$ und $\mathcal{O}(n^2)$ liegt. Allerdings mit der Ausnahme des n-bit Multiplizierers, bei dem die Anzahl der Knoten bei $\mathcal{O}(2^n)$ liegt.

Für die Funktion $f = (a \wedge b \wedge c) \vee (a \wedge \overline{b}) \vee (\overline{a} \wedge b) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c})$ sehen der Entscheidungsbaum (links) und das ROBDD (rechts) wie folgt aus:



Weitere Beispiele für ROBDD; Kapitel 8 - Schaltfunktionen, Folien 484ff.

9.5 Minimierung

Da es zur Realisierung einer Schaltfunktion (beliebig) viele Varianten gibt, lässt sich der Aufwand der Realisierung auch unterschiedlich minimieren.
Wie sehr und auf welche Art der Realisierungsaufwand minimiert wird, ist abhängig von verschiedenen Faktoren, wie z.B. den Hardwarekosten (Anzahl der Gatter), Hardwareeffizienz (`NAND` statt `XOR`), Geschwindigkeit, etc.

9.5.1 Algebraische Minimierungsverfahren

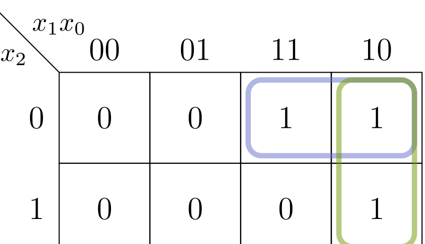
Schaltfunktionen lassen sich durch die Anwendung von den Gesetzen der Boole'schen Algebra vereinfachen, allerdings sind dafür keine allgemeingültigen Algorithmen bekannt und ohne Rechner das Vereinfachen sehr mühsam.

Allerdings gibt es heuristische (Schätz-) Verfahren, wie bspw. die Suche nach Primimplikanten (kürzeste Konjunktionsterme) oder das Quine-McCluskey-Verfahren und Erweiterungen.

9.5.2 Grafische Minimierungsverfahren

Schaltfunktionen lassen sich in KV-Diagrammen darstellen, wobei es möglich ist, daraus die KNF bzw. DNF herzuleiten bzw. abzulesen.

x_2	x_1	x_0	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



$$
\begin{aligned}
DNF &= (\overline{x_2} \wedge x_1 \wedge \overline{x_0}) \vee (\overline{x_2} \wedge x_1 \wedge x_0) \vee (x_2 \wedge x_1 \wedge \overline{x_0}) \\
&= (\overline{x_2} \wedge x_1) \vee (x_1 \wedge \overline{x_0})
\end{aligned}
$$

Don't-Care Terme sind eine Option um Werte und Terme zu markieren, bei denen spezifische Werte nicht relevant sind. Markiert wird dies typischerweise durch einen Asterisk (*) in Funktionstabellen und KV-Diagrammen.

Dies erlaubt das Setzen eines Wertes oder Termes auf 0 oder 1, wie es für die Minimierung am praktischten ist. Außerdem dürfen **Schleifen** Don't Cares enthalten.

9.5.3 Quine-McCluskey Algorithmus

Dieser Algorithmus dient zur Minimierung einer Schaltfunktion, wobei die Terme in Tabellen notiert werden. Prinzipiell entspricht dieser Verfahren dem Verfahren im KV-Diagramm, ist aber auch für mehr als sechs Variablen geeignet.

Zunächst werden hierbei die Terme nach ihrem Hamming-Abstand sortiert, unverzichtbare Terme erkannt und Gruppen von Termen mit einer Distanz von 1 aufgestellt. Nun können geeignete benachbarte Terme zusammengefasst werden.

10 Schaltnetze

10.1 Definition

Schaltznetze sind kombinatorische Schaltungen in Form eines digitalen Systems mit n Eingängen $(b_1, \ldots, b_n)$ und m Ausgängen $(y_1, \ldots, y_m)$. Hierbei sind die Ausgangsvariablen immer nur von den aktuellen Belegungen der Eingangsvariablen abhängig. Da die Ein- und Ausgänge durch Bitvektoren ausgedrückt werden, wird das Schaltnetz als Vektorfunktion (Gruppe von Schaltfunktionen) $\vec{y} = F(\vec{b})$ beschrieben.

10.2 Gatter

Schaltnetze sind eine Realisierung von Schaltfunktionen bzw. Bündeln, bestehend aus **logischen Gatterfunktionen** (Logicgates), welche triviale Funktionen mit einer kleinen Anzahl an Eingängen (2 - 4) sind. Hierbei müssen **Gatterlaufzeiten**, also wieviel Zeit Gatter zum vollständigen Schalten benötigen, beachtet werden. Grundgatter sind dabei `NOT`-, `AND`-, `OR`-Gates, usw., welche weiter kombiniert werden können.

Logische Gatter können unterschiedlich viele Eingänge haben (bspw. Inverter mit einem; `XOR/XNOR` mit zweien; `AND/OR` mit zweien oder mehr; etc.) und erfordern eine vollständige Basismenge, wobei `NAND/NOR` in Halbleitertechnologie besonders effizient sind.

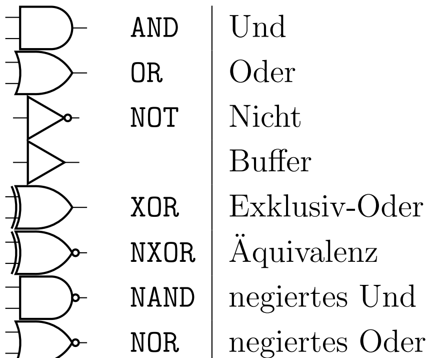
In der Regel haben Gatter maximal vier Eingänge, da höhere Eingangszahlen unpraktisch wären, da sie seltener Nutzung finden, als Varianten mit maximal vier Eingängen und sich höhere Eingangszahlen durch Kombination mehrerer Gatter erzeugen lassen. Bspw. kann man die Ausgänge von vier `AND3` in ein `AND4` laufen lassen um ein `AND12` zu erzeugen.

Zusätzlich gibt es noch sogenannte **Buffer**, dies sind Verstärker, welche dafür sorgen, dass wenn bspw. ein Output auf mehrere Inputs verteilt wird, jeder Input eine genügend starke Spannung erhält, damit keine Binärfehler entstehen. Zusätzlich sorgen Buffer aber auch für eine **Verzögerung**, da diese wie jedes Gatter eine gewisse Gatterlaufzeit besitzen. Buffer können also auch genutzt werden um sicherzustellen, dass verschiedene Signale zur (ungefähr) gleichen Zeit an einer Schaltungskomponente ankommen.

Gewisse Gatter lassen sich unterschiedlich realisieren, so kann ein `XOR` durch eine `AND-OR` Kombination, aber auch durch eine `NAND-NAND` Kombination und einen Multiplexer realisiert werden.

10.3 Schaltpläne

Schaltpläne sind eine standardisierte Art der Darstellung von Schaltungen, wobei es in verschiedenen Ländern verschiedene Normen für Komponentensymbole gibt.



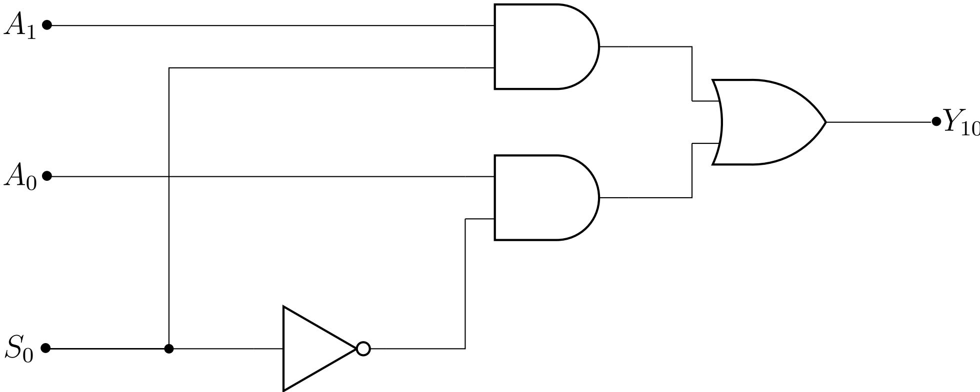
	AND	Und
	OR	Oder
	NOT	Nicht
		Buffer
	XOR	Exklusiv-Oder
	NXOR	Äquivalenz
	NAND	negiertes Und
	NOR	negiertes Oder

10.4 Grundlegende Schaltnetze

Aus den oben beschriebenen logischen Gattern lassen sich grundlegende Schaltnetze bauen, welche wiederum für komplexere Schaltnetzstrukturen benötigt werden.

Multiplexer
Multiplexer schalten zwischen verschiedenen Dateneingänge um. In der Regel hat ein Multiplexer n Dateneingänge und einen Datenausgang. Welcher Dateneingang gewählt wird wird über die Steuereingänge S entschieden.

2:1 Multiplexer
Hier sei beispielhaft das Schaltnetz eines 2:1 Multiplexer dargestellt:



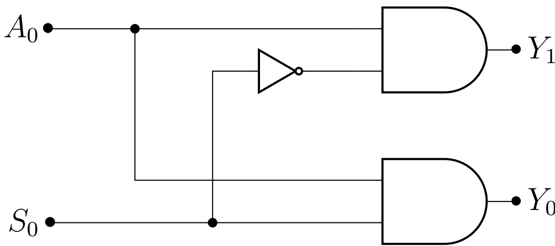
Die Funktionstabelle des 2:1 Multiplexer sieht wie folgt aus:

S_0	A_1	A_0	Y_{10}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S_0	A_1	A_0	Y_{10}
0	$*$	0	0
0	$*$	1	1
1	0	$*$	0
1	1	$*$	1

S_0	A_1	A_0	Y_{10}
0	$*$	A_0	A_0
1	A_1	$*$	A_1

Demultiplexer
Ein Demultiplexer hat in der Regel einen Dateneingang und n Datenausgänge. Durch einen Demultiplexer lässt sich auswählen auf welcher Leitung ein Ergebnis ausgegeben werden soll.
Hier sei beispielhaft das Schaltnetz eines 1:2 Demultiplexer dargestellt:



10.5 Addierer

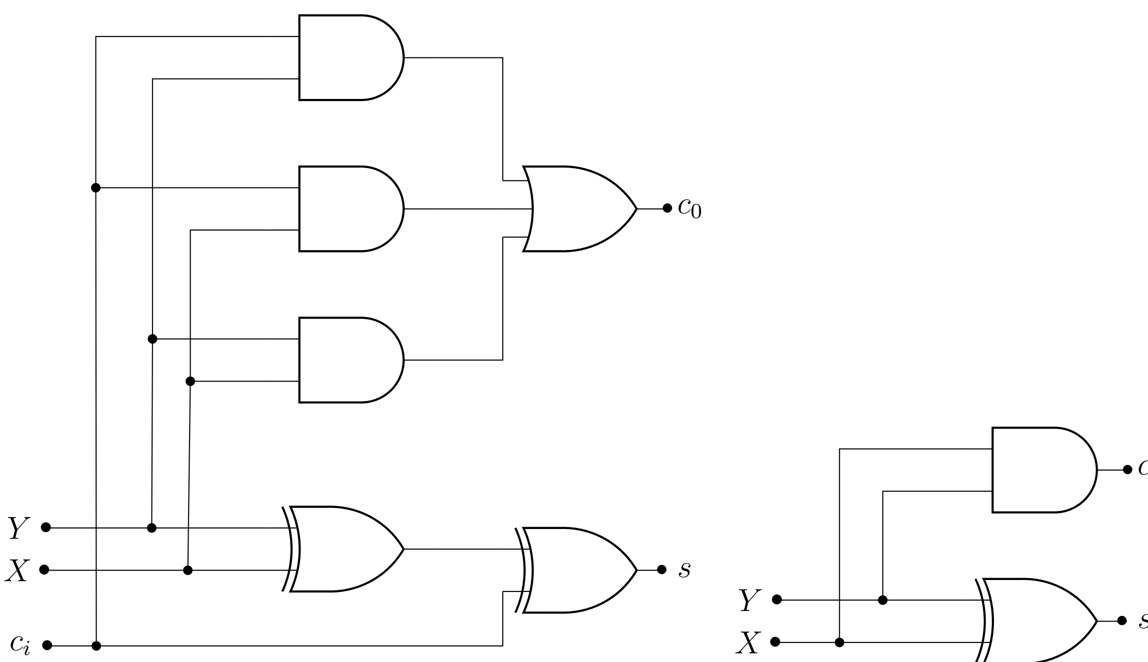
Es wird zwischen Halb- und Volladdierern unterschieden. **Halbaddierer** berechenen die 1-Bitsumme s und den dazugehörigen Übertrag bzw. Carry-Out c_0 von zwei Eingangsbits,
Volladdierer berechnen so wie Halbaddierer die 1-Bitsumme und den Carry-Out, allerdings aus zwei Eingangsbits und dem Carry-In.

Volladierer

X	Y	c_i	c_o	s
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

Halbadierer

X	Y	c_o	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Ripple-Carry Adder

Ein Ripple-Carry Addierer ist eine Kaskade aus mehreren Volladdierern, die hinter einander geschaltet sind. Der Carry-Out der Addition der niederwertigsten Bits wird als Carry-In für die Addition der nächsthöheren Bit verwendet.

Die Laufzeit eines Ripple-Carry Addierers, wächst mit $\mathcal{O}(n)$ wobei n der Wortlänge der zu addierenden Zahlen entspricht.

Andere Addierer sind bspw. der Carry-Select mit einer Laufzeit von $\mathcal{O}(\sqrt{n})$ oder der Carry-Lookahead mit einer Laufzeit von $\mathcal{O}(\ln n)$.

Die Subtraktion lässt sich durch Addition mit Zweierkomplementbildung realisieren.

10.6 Multiplizierer

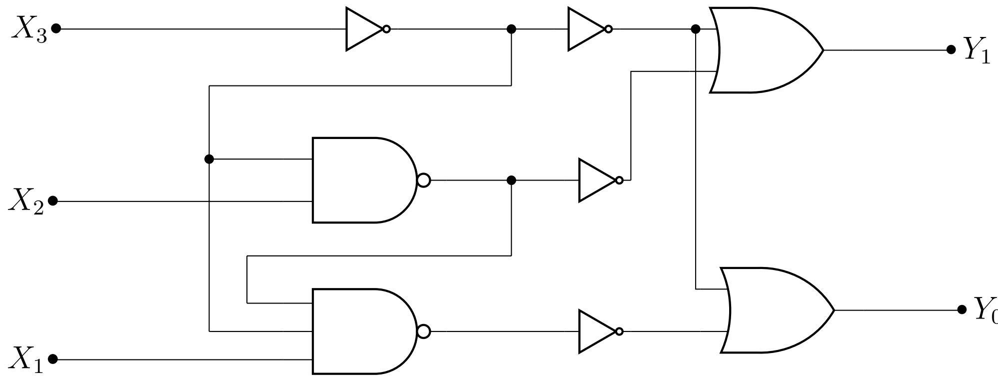
Multiplizierer mit Schaltnetzbeispielen; Kapitel 9 - Schaltnetze, Folien 576ff.

10.7 Prioritätsencoder

Ein Prioritätsencoder dient zur Priorisierung bspw. bei Interrupt-Signalen. Hierbei werden n Eingabebits in eine Dualcodierung konvertiert, wobei vom höchsten Bit an wie folgt priorisiert wird:

x_3	x_2	x_1	x_0	y_1	y_0
1	$*$	$*$	$*$	1	1
0	1	$*$	$*$	1	0
0	0	1	$*$	0	1
0	0	0	$*$	0	0

X_0 kann ignoriert werden, da der Ausgang unabhängig von X_0 ist.



10.8 Shifter, Rotator, etc.

Bei diesen Bauteilen ist Kaskadisierung möglich. Durch den Ausgangsbits vorangestellte Multiplexer wird festgelegt um wie weit geshiftet bzw. rotiert werden soll, indem versetze Leitungen von Eingangsbits zu allen, sich zur Shift-Seite befindenden Multiplexern existieren und so mittels Multiplexer ausgewählt werden kann, welcher Eingangsbit an welchem Ausgangsbit landet.

Bei Rotation wird zusätzlich dafür gesorgt, dass die äußersten Eingangsbits, damit diese beim Rotieren nicht verloren gehen, auf der anderen Seite wieder hineingeschoben werden.

10.9 Arithmetisch-logische Einheit (ALU)

Die ALU ist ein kombiniertes Schaltnetz, welche verschiedene arithmetische und logische Operationen in Form eines zentralen Rechenwerks zusammenfasst. Die verfügbaren Operationen variieren von Prozessor zu Prozessor, aber umfassen in der Regel die Addition und Subtraktion im 2-Komplement, bitweise logische Operationen wie `NOT`, `AND`, `OR`, `XOR`, etc. und Schiebeoperationen. Ab und zu auch Multiplikation, aber nur sehr selten (Integer-)Divison.

Subtraktion erfolgt über die gleiche Schaltung wie die Addition, wobei durch einen zusätzlichen Eingang festgelegt wird, ob für einen Wert das Komplement genutzt wird. Um eine vollständige Subtraktion zu haben, muss dann außerdem noch der Eingangscarry auf 1 gesetzt werden.

Die Auswahl einer ALU-Operation erfolgt über einen Decoder, welcher Op-Codes übersetzt um einen Multiplexer zu steuern, welche die jeweiligen Teile der ALU aktiviert, welche für die gewollte Operation verantwortlich sind.

10.10 Zeitverhalten von Schaltungen

Da Schaltungen in der Realität Zeit zum Schalten brauchen und Leitungen in Abhängigkeit von ihrer Länge Laufzeiten haben, müssen diese Zeiten in Erwägung gezogen werden, wenn ein Schaltnetz implementiert wird. Da es dabei aber viele Schwankungen, Ungenauigkeiten und äußere Einflüsse gibt, werden diese Zeiten für gewöhnlich approximiert, geschätzt oder abstrakt angegeben.

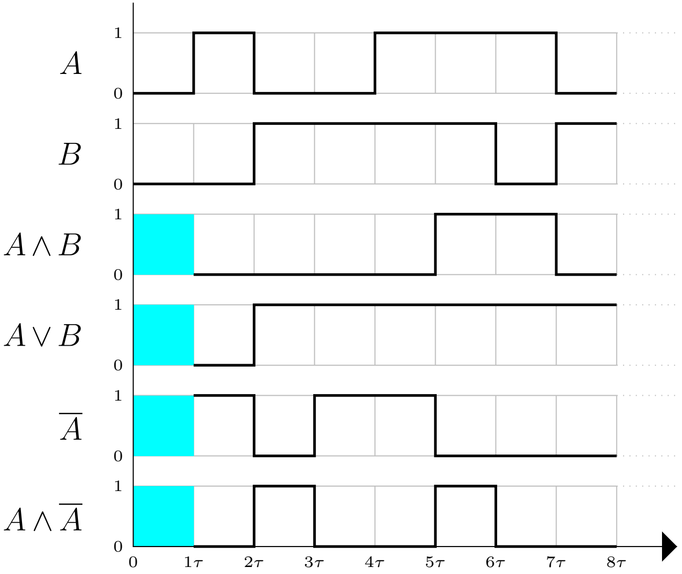
So nehmen wir im Rahmen der Vorlesung τ als eine **Einheitsverzögerung**, welche gewählt wird um Verzögerungen abstrakt anzugeben. So werden Gatterlaufzeiten als Vielfache von τ angegeben und Leitungslaufzeiten ignoriert.

Die maximal mögliche **Taktrate** einer Schaltung, unabhängig von den Verzögerungen der Bauteile lässt sich grob einschränken mithilfe der Geschwindigkeit von Licht c in Metallen und Halbleitern. So ist $c_{Luft} \approx 3 \cdot 10^8$ m / s $\approx$ 30 cm / ns, aber $c_{Metall} \approx$ 20 cm / ns. Dies bedeutet, dass die maximal mögliche Distanz pro Takt bei einer Taktrate von 1 GHz 20 cm beträgt.

10.11 Impulsdiagramme

Impulsdiagramme dienen zur Darstellung der logischen Werte einer Schaltfunktion über der Zeit. Hierbei können neben den Zuständen 1 und 0 auch zusätzliche Zustände wie Z, U und X für unbekannte oder spezielle Zustände angegeben werden. Üblich ist dabei ein X oder ein cyan-Ton um einen unbekannten Wert anzugeben.

Das folgende ist ein Beispiel für ein Impulsdiagramm, wobei jede Operation um genau τ verzögert ist. Am $A \wedge \overline{A}$ sieht man außerdem gut die Problematik der Zeitverzögerung.



10.12 Hazards

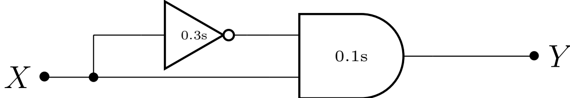
Hazards (engl. Glitch) meint eine Eigenschaft von Schaltfunktionen, bei der bestimmte Kombinationen von Zeitverzögerungen fehlerhafte bzw. nicht unbedingt erwartete Ausgaben produzieren. Diese Fehler nennt man auch **Hazardfehler**.

Man unterscheide in zwei Arten von Hazards, (1) einmal die **statischen Hazards**, wobei erwartet wird, dass sich der Ausgangswert nicht ändert, aber ein Wechsel auftritt; und (2) andererseits die **dynamischen Hazards**, bei welchen ein Wechsel bzw. eine bestimmte Anzahl an Wechseln am Ausgang erwartet wird, aber mehr Wechsel auftreten. Diese Hazards können in Kombination auftreten und mehrfach.

Die Notation von statischen und dynamischen Hazards variiert, aber im Rahmen der Vorlesung schreibt man "statischer 0-Hazards", wenn 0 erwartet wird, aber ein zwischenzeitiger Wechsel zur 1 entsteht; andersrum für einen "statischer 1-Hazard". Unter "dynamischer 0-Hazard" ist zu verstehen, dass ein Wechsel von 1 zu 0 erwartet wird, aber der Wert zwischenzeitig zu 0 und wieder zu 1 wechselt; für einen "dynamischer 1-Hazard" ist es andersrum.

Außerdem lassen sich Hazards auf zwei Bedingungen zurückführen, nach welchen sich diese weiter klassifizieren lassen. Da ist zum einen ein **Strukturhazard**, welcher durch die Struktur der Schaltung bedingt ist und durch eine Veränderung der Schaltstruktur manchmal beseitigt werden kann; zum anderen ist da der **Funktionshazard**, welcher durch die Schaltfunktion bedingt ist und nicht durch strukturelle Maßnahmen beseitigt werden kann, sondern nur durch Anpassen der Funktion, was nicht immer möglich ist; ein Funktionshazard entsteht durch zeitgleichen Wechsel verschiedener Eingänge.

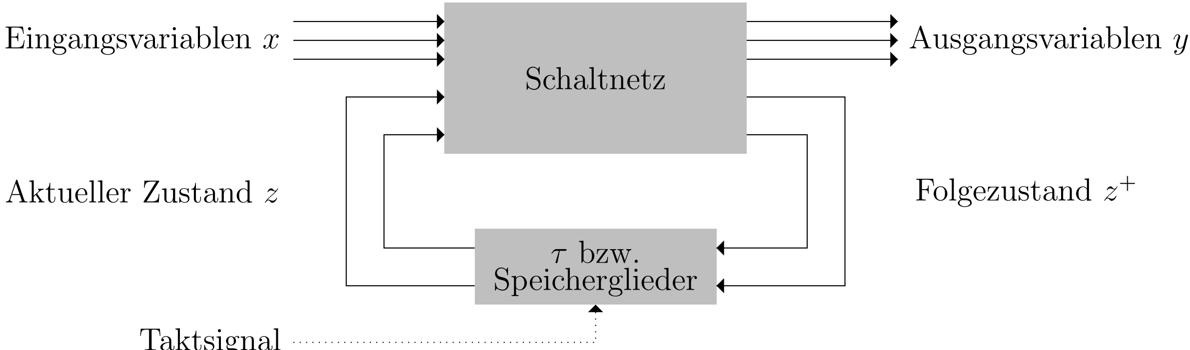
Hazards lassen sich in KV-Diagrammen darstellen; üblich ist dabei, den betroffenen Wechsel mit einem Pfeil zu markieren.



11 Schaltwerke

11.1 Definition

Ein Schaltwerk ist eine Schaltung mit RÜckkopplungen und Verzögerungen, bei der die Ausgangswerte neben den Eingangswerten auch abhängig vom internen Zustand ("Vorgeschichte") abhängig sind. Man kann Schaltwerke in Form von Blockschaltbildern wie folgendem allgemeinen darstellen:



Es wird zwischen synchronen und asynchronen Schaltwerken unterschieden:

Bei **synchronen Schaltwerken** wird der Zeitpunkt, an dem das Schaltwerk von einem Zustand zum nächsten übergeht, durch einen Taktgeber (engl. Clock) vorgegeben.

Da es bei **asynchronen Schaltwerken** keinen Taktgeber gibt, wirken sich alle Änderungen der Eingangssignale, nur durch die Gatterverzögerungen verzögert, auf das Ausgangssignal aus.
Somit wäre eine höhere Arbeitsgeschwindigkeit theoretisch möglich, allerdings wären solche Entwürfe sehr aufwändig und die Schaltwerke sehr fehleranfällig, da durch ändernde Außenbedingungen (Temperatur, Spannungsschwankungen, etc.) das Schaltwerk falsche Ergebnisse liefern würde.
Außerdem kann es bei asynchronen Schaltwerken **stabile** und **nicht-stabile** Zustände geben.

11.2 Deterministische Endliche Automaten

Bei Deterministischen endlichen Automaten (engl. Deterministic Finite State Maschine, FSM) gibt es zwei äquivalente Modelle; dass von **Mealy**, bei dem die Ausgabe von dem

Zustand und einer Eingabe abhängig ist; und dem Modell von **Moore**, bei dem die Ausgabe nur von dem Zustand abhängt.

In beiden Fällen lässt ich der endliche Automat durch einen 6-Tupel beschreiben: $\langle Z, \Sigma, \Delta, \delta, \lambda, z_0 \rangle$

Z Die Menge der Zustände

Σ Das Eingabealphabet

Δ Das Ausgabealphabet

δ Die Übergangsfunktion: $\delta : Z \times \Sigma \rightarrow Z$

λ Die Ausgabe Funktion:

Mealy-Modell: $\lambda : Z \times \Sigma \rightarrow \Delta$

Moore-Modell: $\lambda : Z \rightarrow \Delta$

z_0 Der Startzustand

11.3 Zeitglieder / Flipflops

Zeitglieder meint Bauelemente, welche den Zustand eines Schaltwerks speichern können. Dabei sind Flipflops Bauelemente mit zwei stabilen Zuständen (= 1 Bit). Sie stellen die elementaren Schaltwerke da und haben im Zustandsdiagramm zwei Knoten und vier Übergänge. Der Ausgang eines Flipflops wird dabei als Q bezeichnet und ist oftmals auch invertiert als $\overline{Q}$ verfügbar. Sie sind asynchron und dienen als Speicher- und Verzögerungselemente in synchronen Schaltwerken.

Es gibt verschiedene Arten von Flipflops:

Basis-Flipflop	"Reset-Set-Flipflop" (RS-Flipflop)	Folie 636 - 640
pegelgesteuertes D-Flipflop	"D-Latch"	Folie 641 - 643
flankengesteuertes D-Flipflop	"D-Flipflop"	Folie 644 - 647
Jump-Kill Flipflop	"JK-Flipflop"	Folie 648 - 651

Abbildung 11.1: RS-Flipflop (`NAND` links; `NOR` rechts)

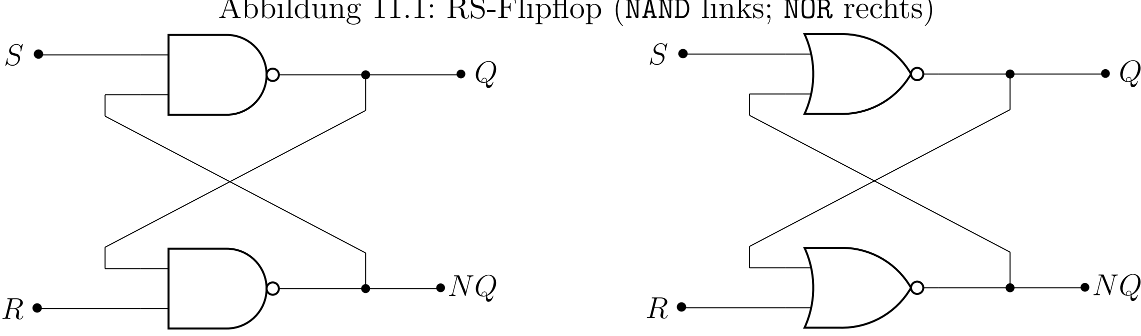


Abbildung 11.2: RS-Flipflop mit Takt

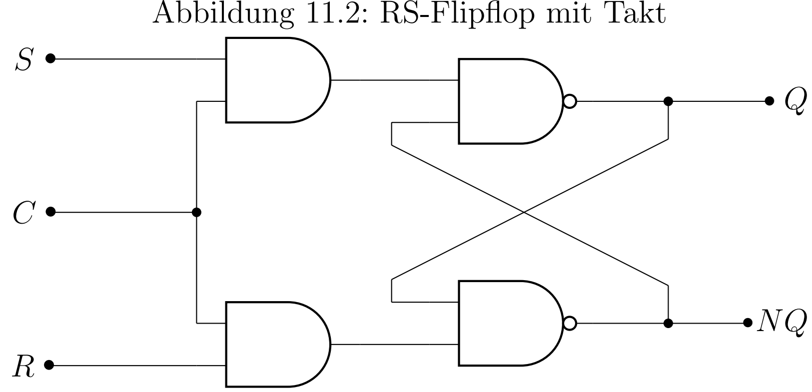
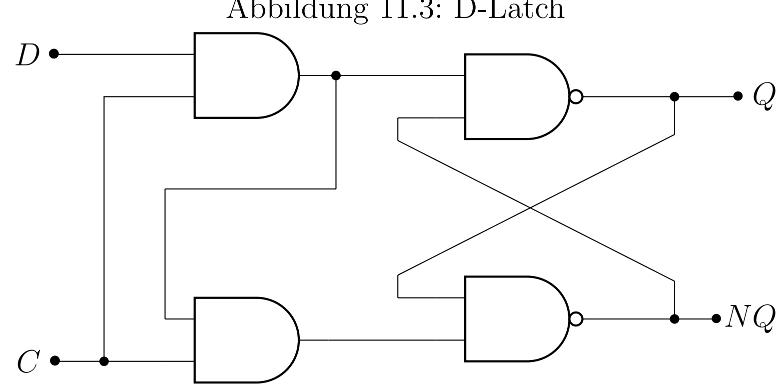
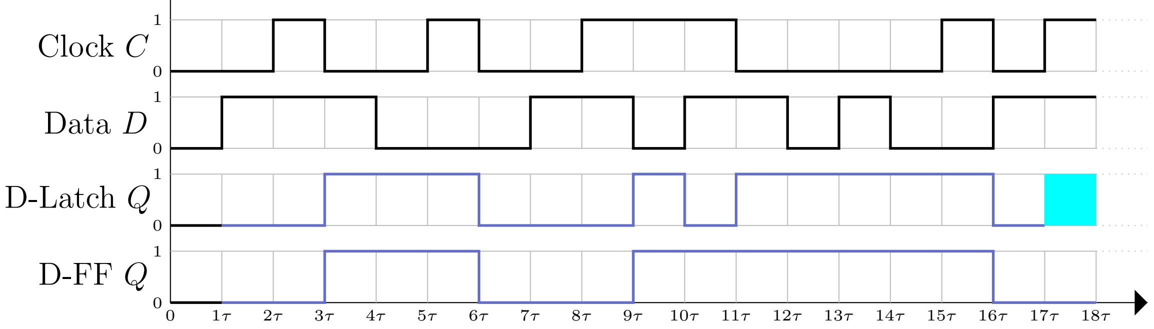


Abbildung 11.3: D-Latch





12 Glossar

Bezeichnung	**Beschreibung**
Alphabet	vereinbarter, geordneter Zeichensatz
alphanumerischer Zeichensatz	Menge an vereinbarten, numerischen sowie alphabetischen Zeichen
AMoore's Law	Alle zwei Jahre verliebt sich ein Informatiker über Paarship.
Aussage	Repräsentation von Information
Befehl	Maschinensprache, Einsen und Nullen
Binärcode	Hat eine Repräsentation einen Binärzeichenvorrat, ist der Code Binärcode
Binärwort	Hat eine Repräsentation einen Binärzeichenvorrat, sind die Wörter Binärwörter
Binärzeichensatz	Menge mit zwei Elementen, welche ein Zweizustandssystem beschreiben, bspw. "an" & "aus"
Blockcode	Code, wobei alle Codewörter die gleiche Länge haben
CISC	Complex Instruction Set Computer
Code	Menge aller Codewörter einer Repräsentation
Codewort	Wort aus dem Zeichenvorrat einer Repräsentation
Codierung	Übersetzen von einer Repräsentation zu einer anderen
einschrittig	aufeinanderfolgende Codewörter unterscheiden sich nur an einer Stelle
Faltungscode	Bitstrom wird in Codebitstrom höherer Bitrate codiert
FLOPS	Floating Point Operations Per Second
Galoisfeld	Galois-Feld mit zwei Elementen GF(2) ist ein Körper aus den zwei Verknüpfungen $\wedge$ (`AND`, Multiplikation) und $\oplus$ (`XOR`, Addition mod2) und hat das additive Inverse $x \oplus x = 0$

Bezeichnung	**Beschreibung**
Gewicht	Anzahl der Einsen in einem Codewort
Information	Abstrakter Gehalt einer Aussage
Interpretation	Ermitteln von Informationen aus Repräsentation
Kanalcodierung	Anpassung an die Strecke
komplementär	Zu jedem Codewort c existiert ein gültiges Codewort $\overline{c}$
linearer (n, k)-Code	k-dimensionaler Unterraum des $\mathrm{GF}(2)^n$
Maxterm	`OR`-Verknüpfung aller Schaltvariablen einer Schaltfunktion, wobei die Variablen negiert oder nicht negiert auftreten dürfen.
Minimalcode	Alle 2^n Codewörter bei Wortlänge n werden benutzt
Minterm	`UND`-Verknüpfung aller Schaltvariablen einer Schaltfunktion, wobei die Variablen negiert oder nicht negiert auftreten dürfen.
MIPS	Million Instructions Per Second
modifizierter Code	Mindestens eine Stelle eines linearen Codes wurde systematisch verändert, also invertiert im GF(2). Null- und Einsvektor gehören nicht mehr zum Code
Moore's Law	Alle zwei Jahre (je nach Quelle auch alle 12 oder 18 Monate) verdoppelt sich die Integrität von Schaltkreisen. In manchen Quellen ist auch die Rede von einer Verdopplung der Transitoren pro Flächeneinheit.
MSB	Der Most Segnificant Bit ist der Vorzeichenbit
Nachricht	Repräsentation im Kontext der Informationsverarbeitung und -übertragung
nichtlinearer Code	Weder linear, noch modifiziert
numerischer Zeichensatz	Menge an vereinbarten, numerisch zu verstehenden Zeichen
Programm	Sequentielle Abfolge von Befehlen
Quellencodierung	Anpassung an die Quelle
redundanter Code	nicht alle möglichen COdewörter werden benutzt
RISC	Reduced Instruction Set Computer
Stelle	Position eines Zeichens in einer Zeichenkette

Bezeichnung	**Beschreibung**
systematischer Code	Wenn zu codierende Information direkt im Codewort enthalten ist
ULP	Unit in Last Place ist der letzte genaue Bit; Abhängig vom Exponenten (Position des Kommas).
Verarbeitungscodierung	Anpassung im Rechner
Verstehen	Verbindung von Information mit Bedeutung
VLSI	Very Large Scale Integration; ab 10.000 Transitoren pro Chip.
Wort	Folge von Zeichen als Einheit (im Zusammenhang); ein 8 bit-Wort ist 1 Byte
Zeichen	z ist ein Zeichen, Z ein Zeichensatz oder Zeichenvorrat, es gilt: $z \in Z$
Zeichenkette	Folge von Zeichen, auch String genannt
Zeichensatz	Eine Menge an Zeichen
zyklisch	bei n geordneten Codewörtern ist $c_0 = c_n$
zyklisch einschrittig	Kombination aus zyklisch und einschrittig, wobei auch das erste und letzte Wort des Codes sich an genau einer Stelle unterscheiden
zyklischer Code	Blockcode bei dem für jedes Codewort gilt, dass auch alle zyklischen Verschiebungen dessen, also Rotationen (`rol`, `ror`, etc.) Codeworte sind.