TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Betriebssysteme

Laura Morgenstern

Skript

zur Vorlesung

Einführung in die Funktionsweise von Computersystemen

Stand: Frühjahr 2018

Inhaltsverzeichnis

Abkürzungen

ALSU Arithmetic/Logic/Shifting Unit
ALU Arithmetic Logic Unit
ANSI American National Standards Institute
ARM Advanced RISC Machines
ASA American Standards Association
ASCII American Standard Code for Information Interchange
BS Betriebssystem
CISC Complex Instruction Set Computer
CMOS Complementary metal-oxide-semiconductor
CPU Central Processing Unit
EBCDIC Extended Binary Coded Decimals Interchange Code
EDVAC Electronic Discrete Variable Automatic Computer
IA-32 Intel Architecture 32-Bit
IC Integrated Circuit
IEC International Electrotechnical Commission
ISO International Standards Organisation
ISR Interrupt Service Routine
LSB Least Significant Bit
MAR Memory Address Register
MDR Memory Data Register
MIPS Microprocessor without Interlocked Pipeline Stages
MSB Most Significant Bit
OS Operating System
PCB Process Control Block
RAM Random-Access Memory
RISC Reduced Instruction Set Computer
ROM Read-Only Memory
SPARC Scalable Processor ARChitecture
TTL Transistor-Transistor-Logik

Abbildungsverzeichnis

Tabellenverzeichnis

Einführung

In der Lehrveranstaltung „Einführung in die Funktionsweise von Computersystemen“[1] setzen wir uns mit den technischen Vorgängen auseinander, die zur Ausführung eines Programms auf einem Rechner notwendig sind. Ergänzend dazu wird das Zusammenspiel von Rechnern in Netzwerken betrachtet. In Vorlesung und Übung werden demnach Themen aus den Bereichen Rechnerarchitektur, Rechnerorganisation, Betriebssysteme und Rechnernetze behandelt, um die Frage „Wie funktioniert ein Computer?“ zu beantworten.

[1] In einigen Studiengängen noch: „Grundlagen der Anwendungsunterstützung“

1 Information und ihre Darstellung

Wie bereits am Begriff „Informatik“ zu erkennen ist, beschäftigt sich die Informatik mit Information, genauer gesagt mit der systematischen Verarbeitung von Informationen durch Rechner. Um im Verlauf der Vorlesung die Verarbeitung von Informationen verstehen zu können, betrachten wir in diesem Kapitel zunächst, was Information ist und wie sie dargestellt werden kann.

1.1 Information

„Information“ ist ein sehr vielschichtiger Begriff, dessen Definition in den verschiedenen Wissenschaftsdisziplinen sowie im Alltag variiert. Laut *Duden -- Das Herkunftswörterbuch* hat das Wort „Information“ die Bedeutung „Nachricht, Auskunft, Belehrung“ und wurde vom Substantiv „informatio“ aus dem Lateinischen entlehnt. [Dud07] Im alltäglichen Sprachgebrauch bringen wir den Begriff seiner Abstammung entsprechend mit dem Erlangen neuen Wissens bzw. der Mitteilung von Sachverhalten in Verbindung. Information kann demnach als extern repräsentiertes bzw. übermitteltes Wissen bezeichnet werden und ist u.a. textuell, grafisch oder audio-visuell darstellbar. Liegen Informationen in einem für die (automatische) Verarbeitung günstigen Format vor, so sprechen wir von Daten. In den nachfolgenden Abschnitten betrachten wir, wie Informationen, insbesondere numerischer bzw. alphanumerischer Art, für die Verarbeitung durch Rechner dargestellt werden können.

1.2 Alphabete

Informationen können mithilfe von Alphabeten dargestellt werden. Ein Alphabet ist dabei eine endliche Menge von paarweise verschiedenen Zeichen. Alphabete in diesem Sinne sind beispielsweise:

- Dezimalziffern: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Buchstaben: $\{a, b, c, ..., A, B, C, ...\}$
- Jahreszeiten: $\{Frühling, Sommer, Herbst, Winter\}$
- Farben: {■, ■, ■, ■, ■, ...}

Um Information technisch darstellen zu können, sollte das zur Repräsentation gewählte Alphabet möglichst klein sein. Das kleinste Alphabet besteht aus zwei Zeichen und ist hier als Menge $\{\texttt{0}, \texttt{1}\}$ definiert. Die Zeichen `0` und `1` können zur Repräsentation unterschiedlicher technischer Zustände verwendet werden. So kann `0` beispielsweise für „Spannung liegt nicht an“ und `1` für „Spannung liegt an“ stehen. Analog dazu könnte der elektrische Ladungszustand (bspw. eines Kondensators) mit `0` für „ungeladen“und `1` für „geladen“ dargestellt werden. Auch die beiden Zustände eines Schalters – „geöffnet“ und „geschlossen“ können so beschrieben werden. Allgemein betrachtet können wir mithilfe dieses Alphabets demnach Fragen, die dem „Ja-Nein-Schema“ folgen, beantworten. Da das Alphabet $\{\texttt{0}, \texttt{1}\}$ genau zwei Zeichen enthält und der Repräsentation von Zuständen dient, sprechen wir auch von einem Binärcode.

1.3 Codes

Ein Code ist eine Abbildung (Zuordnungsvorschrift) zwischen Alphabeten. D.h. wir ordnen Zeichen bzw. Zeichengruppen des einen Alphabetes Zeichen bzw. Zeichengruppen des anderen Alphabets zu. Durch die Möglichkeit, ein einzelnes Zeichen durch eine Zeichengruppe zu kodieren, kann durch ein Alphabet mit kleinerer Zeichenanzahl ein Alphabet mit größerer Zeichenanzahl abgebildet werden. Damit sind wir in der Lage, Symbole aller denkbaren Alphabete mithilfe von Binärcodes (also bspw. dem Alphabet $\{\texttt{0}, \texttt{1}\}$ aus Abschnitt 1.2) auszudrücken. Wie in den nachfolgenden Beispielen erkennbar ist, müssen die Zeichengruppen eines Alphabets, welche jeweils ein Zeichen eines anderen Alphabets codieren, dabei nicht die gleiche Länge aufweisen:

- Kleinbuchstaben:
  - a → `00000`, b → `00001`, c → `00010`, d → `00011`
  - a → • — , b → — • • • , c → — • — • , d → — ••
- Zahlen:
  - 0 → `0`, 1 → `1`, 2 → `10`, 3 → `11`, 4 → `100`
  - 0 → ☺, 1 → ☻, 2 → ☻☺, 3 → ☻☻, 4 → ☻☺☺

Rechner basieren i. d. R. auf zwei elektrischen bzw. magnetischen Zuständen. Aus diesem Grund werden Informationen im Rechner mittels Binärzahlen codiert, wir sprechen von Binärcodierung. Rechner speichern und verarbeiten Daten demnach binär.

1.3.1 Darstellung von Schriftzeichen

In diesem Abschnitt betrachten wir Codes zur rechnergeeigneten Darstellung von alphanumerischen Zeichen, also u. a. zur Darstellung von Ziffern und Buchstaben verschiedener

Alphabete. Dabei ist zu beachten, dass beispielsweise die Zahl 5 im mathematischen Sinne und das Schriftzeichen „5“durch einen Rechner völlig unterschiedlich interpretiert werden können, weswegen sie u. U. unterschiedlich codiert werden.
Zur Darstellung von Text werden die benötigten Buchstaben, Ziffern, Satzzeichen und Steuerzeichen in Form von Bitfolgen (d. h. Folgen von Nullen und Einsen) codiert. Eine Bitfolge kann als Binärzahl interpretiert werden.

American Standard Code for Information Interchange (ASCII)

Die erste Version des 7-Bit-Codes ASCII wurde 1963 von der American Standards Association (ASA) (heute: American National Standards Institute (ANSI)) herausgegeben. ASCII codiert insgesamt 128 Zeichen – die Schriftzeichen der englischen Sprache sowie einige für die Formatierung bzw. Übertragung notwendige Steuerzeichen. Der Code entsteht durch die Zuordnung je einer 7-stelligen Bitfolge zu jedem Zeichen. Diese Zuordnung wiederum entspricht dem Durchnummerieren aller darzustellenden Zeichen mit den Dezimalzahlen `0` bis `127`. ASCII folgt dabei den Prinzipien, die wir aus dem Alltag gewohnt sind: Sowohl die Ziffern `0` bis `9` als auch die Großbuchstaben `A` bis `Z` sowie die Kleinbuchstaben `a` bis `z` sind mit aufeinanderfolgenden Zahlen codiert. Aus Tabelle 1.1 können wir für jedes Zeichen den zugehörigen ASCII-Wert in binärer Form entnehmen.

Tabelle 1.1: ASCII-Zeichentabelle

Code	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
000_	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
001_	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
010_	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
011_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
100_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
101_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
110_	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
111_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Aus Tabelle 1.1 können wir beispielsweise für den Großbuchstaben `A` die die Bitfolge `0100 0001` ablesen. Diese Bitfolge wiederum entspricht der rechnerinternen Repräsentation des Zeichens `A`.
Um nicht nur einzelne Zeichen, sondern auch Zeichenketten („Strings“) darzustellen, reihen wir die Codes der einzelnen Zeichen aneinander. In Abbildung ist exemplarisch die Codierung des Strings `Hello!` dargestellt. Aus Gründen der Lesbarkeit stellen wir diese Binärzahlen hier auch durch ihr hexadezimales (wesentlich kürzeres) Äquivalent dar. In Abschnitt 1.3.2 finden Sie nähere Informationen über die Konvertierung zwischen Hexadezimal- und Binärsystem.

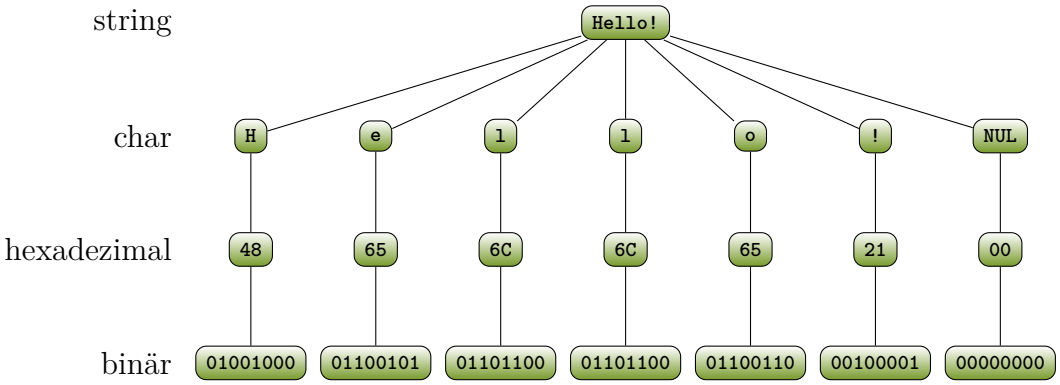


Abbildung 1.1: Binäre Darstellung von `Hello!` nach ASCII

Im Beispiel ist zu erkennen, dass ASCII neben Schriftzeichen, Satzzeichen, Sonderzeichen und Ziffern auch nicht druckbare Steuerzeichen codiert. Diese werden im Code durch die Hexadezimalwerte `00` bis `1F` sowie `7F` repräsentiert und sind in Tabelle 1.1 jeweils durch ein Mnemonic („Kürzel“) dargestellt. In Abbildung 1.1 ist exemplarisch das Nullzeichen „NUL“ zu finden, das u. a. in der Programmiersprache `C` Strings terminiert.

ISO 8859

ISO 8859 ist eine Normenfamilie der International Standards Organisation (ISO), die die Erweiterung von ASCII auf nationale Alphabete standardisiert. Die Erweiterung ist möglich, weil Computer Daten i. d. R. in Vielfachen von 8 Bit ($\hat{=}1 Byte$) speichern und verarbeiten, ASCII allerdings nur die letzten 7 Bit eines Bytes nutzt. In den ISO 8859 Normen entsprechen die Codes 0 bis 127 den ASCII-Zeichen. Die Codes 128 bis 255 können demnach für Codierung nationaler Sonderzeichen genutzt werden. Die Norm ISO 8859-1 ist auch als ISO Latin-1 bekannt. ISO Latin-1 ist in Europa verbreitet und codiert u. a. die deutschen Umlaute.

Extended Binary Coded Decimals Interchange Code (EBCDIC)

Die 8-Bit-Codierung EBCDIC wurde ungefähr zeitgleich zu ASCII von IBM entwickelt und 1964 durch den IBM 360 Computer erstmals verwendet. Eine auf die Abstammung des Codes von der IBM-Lochkarte (siehe Abbildung 1.2) zurückzuführende Eigenart von EBCDIC ist, dass nicht alle Buchstaben des lateinischen Alphabets aufeinanderfolgende Bitkombinationen besetzen. So werden die Buchstaben `A` bis `I` zwar durchgehend durch die Bitmuster `1100 0001` ($\hat{=}$ $\texttt{C1}_{\texttt{hex}}$) bis `1100 1001` ($\hat{=}$ $\texttt{C9}_{\texttt{hex}}$) repräsentiert, die Buchstaben `J` bis `R` folgen dann allerdings erst in den Bitmustern `1101 0001` ($\hat{=}$ $\texttt{D1}_{\texttt{hex}}$) bis `1101 1001` ($\hat{=}$ $\texttt{D9}_{\texttt{hex}}$).

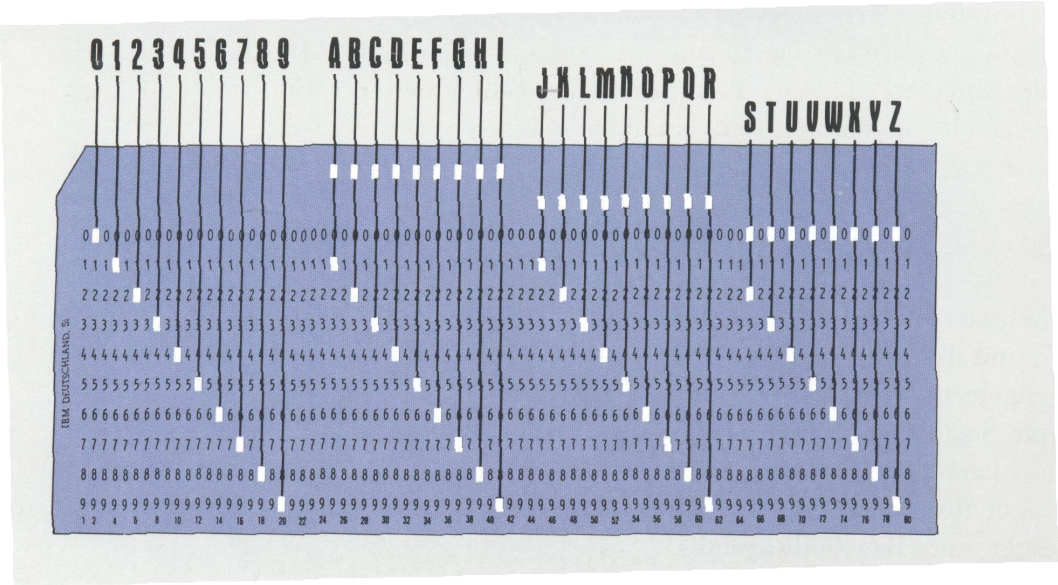


Abbildung 1.2: IBM Lochkarte[1]

Des Weiteren bleiben im EBCDIC, im Gegensatz zu ASCII, einige Bitkombinationen ungenutzt, d. h. sie codieren kein Zeichen.
Da IBM Rechner weltweit vertreibt, wurde die Entwicklung diverser, länderabhängiger Ausprägungen („Codepages“) des Codes notwendig. Während Codepage 273 beispielsweise den Zeichensatz für Deutschland und Österreich codiert, codiert die in Tabelle 1.2 dargestellte Codepage 500 den internationalen Zeichensatz.

Tabelle 1.2: EBCDIC 500[2]

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4_			â	ä	à	á	ã	å	ç	ñ	[	.	<	(	+	!
5_	&	é	ê	ë	è	í	î	ï	ì	ß	]	$	*	)	;	^
6_	-	/	Â	Ä	À	Á	Ã	Å	Ç	Ñ	¦	0	%	_	>	?
7_	ø	É	Ê	Ë	È	Í	Î	Ï	Ì	‘	:	#	@	’	=	
8_	Ø	a	b	c	d	e	f	g	h	i	«	»	ð	ý	þ	±
9_	°	j	k	l	m	n	o	p	q	r	ª	º	æ	¸	Æ	¤
A_	µ	~	s	t	u	v	w	x	y	z	¡	¿	Ð	Ý	Þ	®
B_	¢	£	¥	·	©	§	¶	¼	½	¾	¬	|	¯	¨	´	×
C_	{	A	B	C	D	E	F	G	H	I		ô	ö	ò	ó	
D_	}	J	K	L	M	N	O	P	Q	R	¹	û	ü	ù	ú	ÿ
E_	\	÷	S	T	U	V	W	X	Y	Z	²	Ô	Ö	Ò	Ó	Õ
F_	0	1	2	3	4	5	6	7	8	9	³	Û	Ü	Ù	Ú	

[1] http://kkraftonline.de/Museum/data/images/IBM%20Lochkarte.jpg (25.04.2016)
[2] ftp://ftp.software.ibm.com/software/globalization/gcoc/attachments/CP00500.pdf (18.04.2016)

Bedingt durch den Umstand, dass unterschiedliche Sprachen auch unterschiedliche Zeichensätze erfordern, existieren heute mehr als 100 solcher Codepages. Diese Menge an EBCDIC-Varianten führt allerdings zu Schwierigkeiten, wenn eine mittels einer bestimmten Codepage codierte Datei in einem Land mit einer anderen Codepage darstellbar sein soll. EBCDIC kam hauptsächlich auf Mainframes zum Einsatz und konnte sich letztlich nicht gegen ASCII und dessen Erweiterungen behaupten.

Unicode

In Abschnitt 1.3.1 haben wir gesehen, dass die Codierung nationaler Sonderzeichen durch separate Normen bzw. Codepages hinsichtlich des Datentransfers zwischen Systemen problematisch ist. Aus diesem Grund versucht das Unicode-Konsortium nicht nur alle nationalen Alphabete, sondern auch phonetische, mathematische und weitere Zeichen in einem einheitlichen Zeichensatz – dem Unicode – zu vereinen. Unicode ist im Prinzip ein „Über-Alphabet“, das jedem Zeichen unabhängig von Plattform, Programm oder Sprache einen eindeutigen Zahlenwert („Codepoint“) zuordnet. Der deutsche Umlaut „ä“ ist beispielsweise dem Codepoint `U+00E4` zugeordnet. Dabei steht `U+` für „Unicode“ und `00E4` ist ein Hexadezimalwert. Die im Unicode enthaltenen Alphabete sind in Blöcken organisiert. So entspricht beispielsweise der Block „Basic Latin (ASCII)“mit den Codepoints `U+0000` bis `U+007F` den Zeichen des ASCII-Codes. Der in Tabelle 1.3 dargestellte Block zeigt beispielhaft die Zeichen des georgischen Alphabets und die zugehörigen Codepoints im Unicode.

Tabelle 1.3: Unicode-Block „Georgian“ (U+10A0 bis U+10FF, grüne Felder sind nicht belegt)

	10A	10B	10C	10D	10E	10F
0	Ⴀ	Ⴐ	Ⴠ	ა	რ	ჰ
1	Ⴁ	Ⴑ	Ⴡ	ბ	ს	ჱ
2	Ⴂ	Ⴒ	Ⴢ	გ	ტ	ჲ
3	Ⴃ	Ⴓ	Ⴣ	დ	უ	ჳ
4	Ⴄ	Ⴔ	Ⴤ	ე	ფ	ჴ
5	Ⴅ	Ⴕ	Ⴥ	ვ	ქ	ჵ
6	Ⴆ	Ⴖ		ზ	ღ	ჶ
7	Ⴇ	Ⴗ	Ⴧ	თ	ყ	ჷ
8	Ⴈ	Ⴘ		ი	შ	ჸ
9	Ⴉ	Ⴙ		კ	ჩ	ჹ
A	Ⴊ	Ⴚ		ლ	ც	ჺ
B	Ⴋ	Ⴛ		მ	ძ	
C	Ⴌ	Ⴜ		ნ	წ	ჼ
D	Ⴍ	Ⴝ	Ⴭ	ო	ჭ	ჽ
E	Ⴎ	Ⴞ		პ	ხ	ჾ
F	Ⴏ	Ⴟ		ჟ	ჯ	ჿ

Nachdem wir mit Unicode ein universales Alphabet vorliegen haben, benötigen wir nun noch eine Möglichkeit, dieses für die Verarbeitung mit Rechnern zu codieren. Naheliegend ist beispielsweise die Codierung der Codepoints, die jeweils einer Hexadezimalzahl mit 4 Stellen entsprechen, in Form von 2 Bytes ($\hat{=}16Bit$). Die zu diesem Ansatz gehörende Codierung ist UTF-16. Mit dieser Methode können $16^4 (= 65536)$ Zeichen dargestellt werden. In der englischsprachigen Welt, die fast ausschließlich die Unicode-Zeichen `U+0000` bis `U+007F` nutzt, führt diese Variante allerdings dazu, dass ein Byte nur Nullen enthält und somit Speicherplatz „verschwendet“ wird. Aus diesem Grund wurde die UTF-8 Kodierung entwickelt. UTF-8 ist eine Mehrbyte-Codierung, bei der die ursprünglichen ASCII Zeichen mit nur einem Byte ($\hat{=}8Bit$) codiert werden. Somit werden besonders häufig genutzte Zeichen mit einer besonders kurzen Bitfolge codiert, was zu geringerem Speicheraufwand führt. Seltener genutzte Zeichen werden in UTF-8 durch 2- bis 6-Byte-Codes codiert.

1.3.2 Darstellung von Zahlen

Nachdem wir nun Möglichkeiten zur rechnergeeigneten Darstellung von Text kennengelernt haben, betrachten wir in den folgenden Abschnitten die Repräsentation von Zahlen. In diesem Abschnitt beschäftigen wir uns zunächst allgemein mit zwei Varianten zur Darstellung von Zahlen – den „Additionssystemen“ und den „Positionsystemen“.
In Additionssystemen errechnet sich der Wert einer Zahl durch Addition der einzelnen Ziffern, wobei die Positionen der Ziffern innerhalb der Zahl keine Rolle spielen. Ein aus dem Alltag bekanntes Beispiel für ein Additionssystem ist die Strichliste, die als einzige Ziffer den Strich „|“ mit einer Wertigkeit von Eins verwendet. Auch das Römische Zahlensystem ist ein Additionssystem, wenn auch ergänzt um eine Subtraktionsregel für bestimmte Ziffernkombinationen, um die Schreibweise zu verkürzen. Im Römischen Zahlensystem werden die lateinischen Buchstaben I (Eins), V (Fünf), X (Zehn), L (Fünfzig), C (Hundert), D (Fünfhundert) und M (Tausend) als Ziffern verwendet.
Bei Positions- bzw. Stellenwertsystemen hingegen ist der Wert einer Zahl nicht nur durch die Werte der einzelnen Ziffern bestimmt, sondern auch durch deren Positionen innerhalb der Zahl. Auch wenn wir im Alltag fast ausschließlich das Dezimalsystem, welches ein Beispiel für ein Positionssystem ist, nutzen, so gibt es dennoch unendlich viele Möglichkeiten, ein Positionssystem zu konstruieren. Dazu betrachten wir, wie sich der Dezimalwert einer Zahl in einem Positionssystem im Allgemeinen zusammensetzt:

$$v = \sum_{i=0}^{n} (b_i \cdot \beta^i) \tag{1.1}$$

Der Index i läuft mit $i = 0$ von rechts beginnend über alle n Ziffern der Zahl. b_i entspricht dem Wert der Ziffer an Position i, während β der gewählten Basis des Positionssystems

entspricht. Wir betrachten ein Beispiel anhand des Positionssystems mit Basis 10, dem Dezimalsystem:

$$v(\mathbf{12345}_{DEC}) = \mathbf{1} \cdot 10^4 + \mathbf{2} \cdot 10^3 + \mathbf{3} \cdot 10^2 + \mathbf{4} \cdot 10^1 + \mathbf{5} \cdot 10^0 = 12345_{DEC}$$

Die Zahl 12345 kann also auch als die Summe von 5 Einern, 4 Zehnern, 3 Hundertern, 2 Tausendern und einem Zehntausender beschrieben werden. Weitere gängige Systeme neben dem Dezimalsystem sind das Dual- bzw. Binärsystem (BIN) mit Basis 2 und Ziffernmenge $\{0, 1\}$, das Oktalsystem (OCT) mit Basis 8 und Ziffernmenge $\{0, 1, ..., 7\}$ sowie das Hexadezimalsystem (HEX) mit Basis 16 und Ziffernmenge $\{0, 1, ..., 9, A, B, ..., F\}$. Wie die folgenden Beispiele zeigen, funktioniert die Berechnung des Dezimalwertes für diese Systeme analog:

$$\begin{aligned} v(\mathbf{101010}_{BIN}) &= \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{0} \cdot 2^0 = 42_{DEC} \\ v(\mathbf{52}_{OCT}) &= \mathbf{5} \cdot 8^1 + \mathbf{2} \cdot 8^0 = 42_{DEC} \\ v(\mathbf{2A}_{HEX}) &= \mathbf{2} \cdot 16^1 + \mathbf{A} \cdot 16^0 = 42_{DEC} \end{aligned}$$

Wie die Beispiele zeigen, können wir anhand von Formel 1.1 Zahlen aus jedem beliebigen Positionssystem in das Dezimalsystem überführen. Im Folgenden betrachten wir nun die Gegenrichtung dazu sowie weitere nützliche Konvertierungen zwischen Dual-, Oktal-, Dezimal- und Hexadezimalsystem.

Konvertierung von Dezimal- zu Dual-, Oktal- und Hexadezimalsystem Um eine Dezimalzahl in ein beliebiges anderes Positionssystem zu überführen, wird die Ausgangszahl zunächst durch die Basis des Zielsystems geteilt und sowohl das Ergebnis, als auch der Rest der Division, notiert. Anschließend wird das Ergebnis der vorhergehenden Division wiederum durch die Basis des Zielsystems geteilt und Ergebnis und Rest notiert. Dieser Vorgang wird wiederholt, bis wir als Ergebnis der Division Null erhalten. Die notierten Rest-Werte entsprechen nun, in umgekehrter Reihenfolge ihrer Entstehung betrachtet, der in das Zielsystem konvertierten Zahl. Im folgenden Beispiel wird die Vorgehensweise anhand der Überführung der Dezimalzahl 42 in das Dualsystem demonstriert:

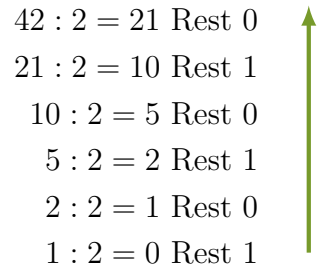


Abbildung 1.3: Konvertierung von 42_{DEC} ins Dualsystem

Die Basis des Dualsystems ist 2. Aus diesem Grund wird 2 im Beispiel als Divisor verwendet. Nach Durchführung der Divisionen wird die binäre Darstellung von 42_{DEC} in Pfeilrichtung von den ermittelten Rest-Werten abgelesen. Das Resultat der Umwandlung ist demnach 101010_{BIN}. Analog dazu funktioniert die Konvertierung einer Dezimalzahl in das Oktal- bzw. Hexadezimalsystem. Hierfür muss lediglich der Divisor durch die Basis des entsprechenden Zielsystems (beispielsweise 8 oder 16) ersetzt werden.

Konvertierungen zwischen Dual-, Oktal- und Hexadezimalsystem Um die nachfolgend vorgestellten Konvertierungen verstehen zu können, sind folgende Überlegungen hilfreich: In einer Oktalzahl kann jede Stelle durch eine von 8 Ziffern besetzt werden. Um 8 Ziffern binär, also als Folge von Nullen und Einsen zu codieren, werden wegen $\log_2(8) = 3$ genau drei Bit benötigt. Die acht dreistelligen Bitfolgen zur Codierung der Ziffern 0 bis 7 sind dann folgende:

$$
\begin{aligned}
000 &\rightarrow 0\\
001 &\rightarrow 1\\
010 &\rightarrow 2\\
011 &\rightarrow 3\\
100 &\rightarrow 4\\
101 &\rightarrow 5\\
110 &\rightarrow 6\\
111 &\rightarrow 7
\end{aligned}
$$

Analog dazu können wir uns überlegen, dass für die Darstellung der 16 Ziffern des Hexadezimalsystems wegen $\log_2(16) = 4$ vier Bit benötigt werden. Die zugehörigen vierstelligen

Bitfolgen sind:

$$\begin{aligned}
0000 &\rightarrow 0 \qquad 1000 \rightarrow 8 \\
0001 &\rightarrow 1 \qquad 1001 \rightarrow 9 \\
0010 &\rightarrow 2 \qquad 1010 \rightarrow A \\
0011 &\rightarrow 3 \qquad 1011 \rightarrow B \\
0100 &\rightarrow 4 \qquad 1100 \rightarrow C \\
0101 &\rightarrow 5 \qquad 1101 \rightarrow D \\
0110 &\rightarrow 6 \qquad 1110 \rightarrow E \\
0111 &\rightarrow 7 \qquad 1111 \rightarrow F
\end{aligned}$$

Mit diesen Vorüberlegungen und (bestenfalls) den Bitfolgen für die entsprechenden Ziffern im Hinterkopf fällt die Konvertierung leicht: Wir unterteilen die Dualzahl von rechts beginnend in 3- bzw. 4-Bit-Blöcke und notieren zu jedem Block die zugehörige Oktal- bzw. Hexadezimalziffer und erhalten so die äquivalente Oktal- bzw. Hexadezimaldarstellung. Im folgenden Beispiel konvertieren wir die Dualzahl 101010 auf dem beschriebenen Weg ins Oktal- und Hexadezimalsystem:

$$\underbrace{101}_{5}\underbrace{010}_{2} \rightarrow 52_{OCT}$$

$$\underbrace{0010}_{2}\underbrace{1010}_{\mathrm{A}} \rightarrow 2A_{HEX}$$

Sollte, wie im Beispiel für die Hexadezimaldarstellung geschehen, die Anzahl der Stellen der Dualzahl nicht ohne Rest durch die Blockgröße teilbar sein, so wird die Dualzahl von links um die entsprechende Anzahl Nullen ergänzt.
Dieses Vorgehen funktioniert auf ähnliche Weise auch in die Gegenrichtung, d. h. für die Konvertierung einer Zahl vom Oktal- bzw. Hexadezimalsystem in das Dualsystem. Hierfür werden die Ziffern der Zahl vereinzelt und durch ihr binäres Äquivalent ersetzt:

$$\underbrace{5}_{101}\underbrace{2}_{010} \rightarrow 101010_{BIN}$$

$$\underbrace{2}_{0010}\underbrace{A}_{1010} \rightarrow (00)101010_{BIN}$$

Die führenden Nullen, die sich im Beispiel für die Hexadezimaldarstellung durch die Blockschreibweise ergeben, können im Ergebnis weggelassen werden. In Abschnitt 1.3.1 wurden Hexadezimalwerte als Kurzschreibweise für Bitfolgen verwendet. Diese Schreibweise ist

praktisch, weil wir bei einer Hexadezimalzahl mittels des hier vorgestellten Verfahrens die zugehörige Bitfolge ohne großen Rechenaufwand direkt „sehen“ können.

Vorzeichenlose Dualzahlen

Nachdem die Darstellung von Zahlen durch Positionssysteme bekannt ist, betrachten wir nun darauf aufbauend die rechnergeeignete Darstellung von vorzeichenlosen, ganzen Zahlen. Dass prinzipiell beliebige Dezimalzahlen als Dualzahlen dargestellt werden können, haben wir bereits gesehen. Für die Praxis sind allerdings zusätzlich der Wertebereich bzw. die Stellenanzahl der Dualzahlen relevant – beispielsweise um im Speicher liegende Bitfolgen voneinander abzugrenzen.
Die Anzahl der durch eine n-stellige Dualzahl darstellbaren Werte berechnet sich zu 2^n. Welcher Wertebereich durch diese 2^n Bitfolgen abgedeckt wird, ist theoretisch frei wählbar. Ein Beispiel: Mit vier Bit können wir $2^4 = 16$ verschiedene Bitfolgen darstellen. Intuitiv können damit z. B. die Natürlichen Zahlen 0...15 dargestellt werden. Allerdings könnten die Bitfolgen auch auf die Zahlen 1...16 oder 16...31 abbilden.
Ebenso wie bei der Codierung alphanumerischer Zeichen, benötigen wir auch hier Konventionen zur definierten Darstellung. Diese Konventionen werden in Programmiersprachen durch Datentypen realisiert. Beispiele für Datentypen sind:

- `char` ($n = 8, 0...255$)
- `int, short` ($n = 16, 0...65535$)
- `int, long` ($n = 32, 0...4294967295$ *oder* $n = 64, 0...18446744073709551615$)

Wie in den Beispielen ersichtlich wird, decken wir durch die Bitfolgen üblicherweise den Wertebereich $0...2^n - 1$ ab.

Vorzeichenbehaftete Dualzahlen

In diesem Abschnitt widmen wir uns nun der Darstellung von vorzeichenbehafteten, ganzen Zahlen. Dabei ist zu beachten, dass in Rechnern, wie in Abschnitt 1.2 erläutert, zur Darstellung von Information nur zwei Zustände (`0` und `1`) zur Verfügung stehen. Entsprechend ist es nicht möglich einer Dualzahl ein Vorzeichen in Form von „+“ bzw. „-“ (analog zum Dezimalsystem) voranzustellen. Passenderweise nimmt allerdings auch das Vorzeichen einer Zahl jeweils einen von zwei Zuständen (positiv oder negativ) an, weswegen es zunächst naheliegend ist ein Bit zur Codierung des Vorzeichens zu nutzen. Auf diesem Weg sind mit n Bits alle ganzen Zahlen im Bereich $-1 \cdot (2^{n-1} - 1)...2^{n-1} - 1$ abbildbar.
Im Beispiel entspricht das am weitesten links stehende Bit („Most Significant Bit“) dem Vorzeichen. Dabei steht `0` für ein positives und `1` für ein negatives Vorzeichen. Alle weiteren

Bits entsprechen dem Wert der Zahl (sowie ggf. führenden Nullen, um eine einheitliche Länge der Zahlen zu erreichen):

0000 → +0	1000 → -0
0001 → +1	1001 → -1
0010 → +2	1010 → -2
0011 → +3	1011 → -3
0100 → +4	1100 → -4
0101 → +5	1101 → -5
0110 → +6	1110 → -6
0111 → +7	1111 → -7

Im Beispiel wird eine problematische Eigenart dieser Darstellungsvariante deutlich: Es gibt zwei Darstellungen für Null, nämlich `0000` und `1000`. Das führt dazu, dass anstelle der mit vier Bit möglichen 16 Werte nur 15 Werte codiert werden können. Diese Darstellung erschwert außerdem die Konstruktion eines Rechenwerks für die Addition, weil einige Fallunterscheidungen notwendig werden: So wird zunächst festgestellt, ob die Summanden gleiche oder unterschiedliche Vorzeichen aufweisen. Sind die Vorzeichen gleich, werden die Beträge addiert und das Vorzeichen übernommen. Sind die Vorzeichen hingegen unterschiedlich, wird der kleinere Betrag vom größeren Betrag subtrahiert und das Vorzeichen des größeren Betrags übernommen.
Um diese Probleme zu umgehen, betrachten wir in den beiden folgenden Abschnitten alternative Darstellungen vorzeichenbehafteter Dualzahlen.

Einerkomplement Im Einerkomplement werden positive Dualzahlen wie bisher (vgl. Abschnitt 1.3.2) dargestellt. Um das negative Pendant darzustellen, werden alle Bits der positiven Dualzahl invertiert, d. h. aus `0` wird `1` und umgekehrt. Die Gegenrichtung, d. h. die Umwandlung einer negativen Zahl in ihr positives Gegenstück, funktioniert ebenso. Auch Zahlen im Einerkomplement weisen eine feste Bitanzahl auf, was wiederum zum Auffüllen mit Nullen führen kann. Bei der Umwandlung einer positiven Dualzahl in eine negative Dualzahl werden diese Nullen ebenfalls invertiert. Im dargestellten Zahlenkreis des 4-Bit-Einerkomplements wird die Invertierung durch Pfeile veranschaulicht:

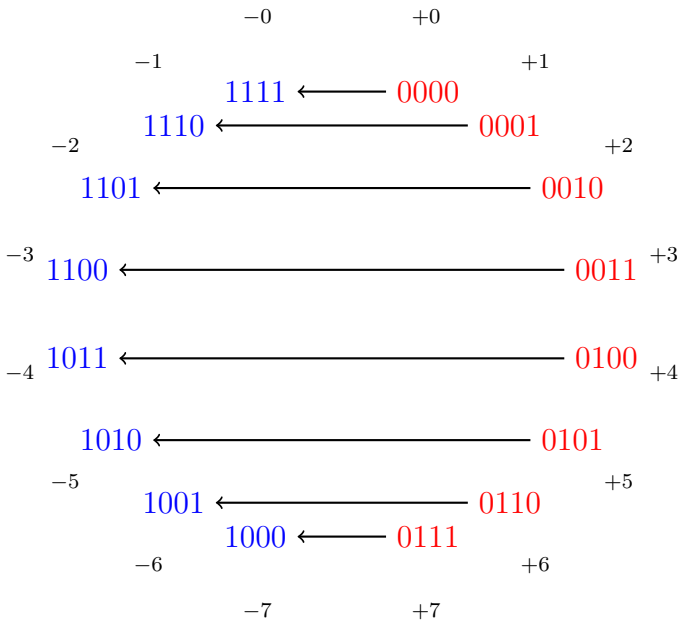


Abbildung 1.4: Zahlenkreis 4-Bit-Einerkomplement

In Abbildung 1.4 wird ersichtlich, dass die positiven Zahlen (rot) von +0 bis +7 jeweils mit 0 beginnen. Die negativen Zahlen (blau) von −0 bis −7 beginnen hingegen mit 1. D.h. wir können auch hier, analog zur Vorzeichenbit-Darstellung, das Vorzeichen direkt am Most Significant Bit ablesen. Der symmetrische Wertebereich entspricht auch hier $-1 \cdot (2^{n-1}-1) \ldots 2^{n-1}-1$. Wie im Beispiel zu sehen, ist auch die Einerkomplementdarstellung nicht eindeutig – es gibt auch hier zwei Darstellungen für Null. Auch die Konstruktion eines Rechenwerks für die Addition wird durch das Einerkomplement nicht einfacher. Da diese Schwierigkeiten so nicht gelöst werden, dient das Einerkomplement lediglich als Basis für das nachfolgend vorgestellte Zweierkomplement.

Zweierkomplement Die Darstellung der positiven Zahlen ändert sich auch im Zweierkomplement nicht. Um das negative Gegenstück einer positiven Zahl zu erhalten, gibt es zwei Möglichkeiten:

1. Einerkomplement der positiven Dualzahl bilden und anschließend 1 addieren.
2. Positive Dualzahl von rechts durchlaufen und alle Nullen sowie die erste Eins abschreiben. Anschließend die übrigen Stellen invertieren und übernehmen.

Beide Varianten sind ebenso für die Umwandlung einer negativen Zahl in ihr positives Pendant verwendbar.
Abbildung 1.5 zeigt alle mit 4 Bit darstellbaren Zahlen im Zweierkomplement:

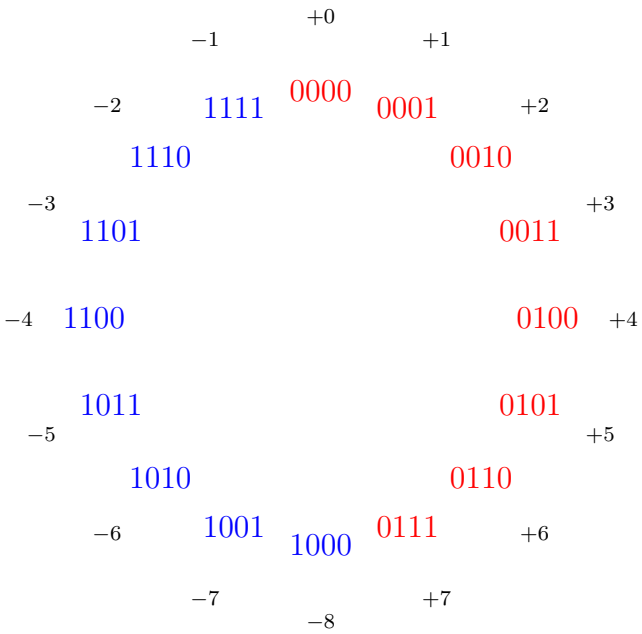


Abbildung 1.5: Zahlenkreis 4-Bit-Zweierkomplement

Im Gegensatz zum Einerkomplement gibt es im Zweierkomplement auch für Null nur eine Darstellung, nämlich 0000. Das Zweierkomplement ist somit eindeutig und weist den unsymmetrischen Wertebereich $-1 \cdot (2^{n-1})...2^{n-1} - 1$ auf. Da das Zweierkomplement demnach die Nachteile des Einerkomplements beseitigt, ist es die gebräuchlichste Methode zur Darstellung vorzeichenbehafteter Dualzahlen.

Rationale Zahlen

Bisher haben wir uns ausschließlich mit der Darstellung ganzer Zahlen beschäftigt. Allerdings spielen, insbesondere in wissenschaftlichen Anwendungen, auch rationale Zahlen (gebrochene Zahlen) eine wichtige Rolle. In diesem Abschnitt betrachten wir deshalb zwei Möglichkeiten zur computergeeigneten Darstellung von rationalen Zahlen mithilfe der Binärdarstellung, welche für die Trennung von Vor- und Nachkommateil kein Trennzeichen vorsieht.

Festkommazahlen Eine Festkommazahl sowie der entsprechende Vor- bzw. Nachkommateil haben eine festgelegte Anzahl an Stellen, welche für alle darzustellenden Zahlen gleichermaßen gilt. Durch die Festlegung der Stellenanzahlen entfällt die Darstellung des Kommas. Die binäre Darstellung des Vorkommateils erfolgt wie in Abbildung 1.6 dargestellt. Die Darstellung des Nachkommateils erfolgt nach dem selben Schema – allerdings mit inversen Zweierpotenzen. Das Beispiel zeigt die Darstellung der Dezimalzahl $10,625$ als binäre 8-Bit-Festkommazahl mit vier Vorkomma- und vier Nachkommastellen:

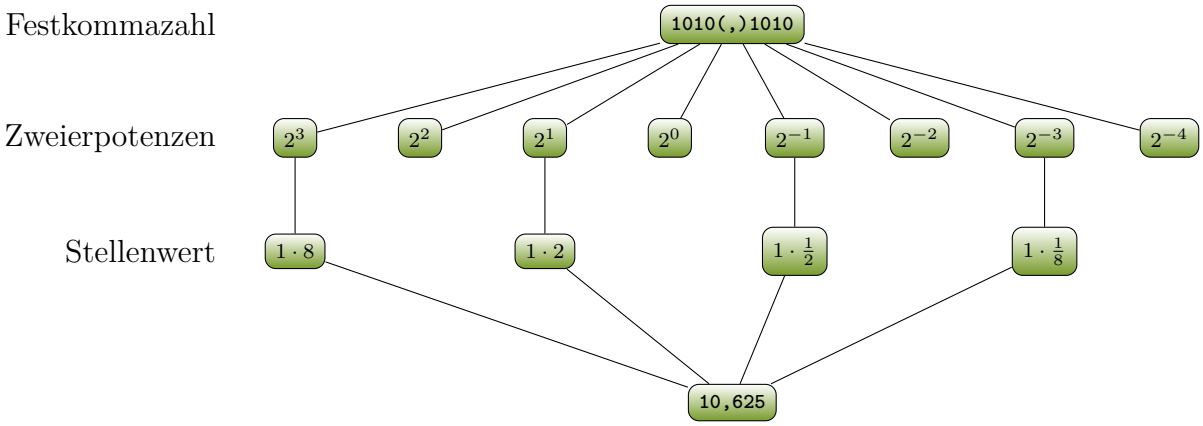


Abbildung 1.6: $10,625_{DEC}$ als 8-Bit-Festkommazahl

Problematisch am Festkommaformat ist, dass wir uns durch die Festlegung einer einheitlichen Kommaposition für alle Zahlen zwischen großem Wertebreich und hoher Genauigkeit entscheiden müssen. Verschieben wir das (gedachte) Komma nach rechts, wächst der Wertebereich, aber die Genauigkeit sinkt. Verschieben wir das Komma nach links, wächst die Anzahl der Nachkommastellen und wir erhlaten somit eine höhere Genauigkeit. Allerdings veringert sich dann der Wertebereich. Die Entscheidung zwischen großem Wertebereich und hoher Genauigkeit lässt sich in der Praxis kaum sinnvoll treffen: Während beispielsweise Astronomen zur Darstellung der Sonnenmasse auf große Wertebreiche angewiesen sind, benötigen Physiker und Chemiker hohe Genauigkeiten zur Abbildung von Naturkonstanten und Messwerten. Prinzipiell wäre dieses Problem zu lösen, in dem die Stellenanzahl der Festkommazahlen entsprechend groß gewählt wird. So könnten wir den Astronomen die benötigten Vorkommastellen bieten, ohne dass Physiker und Chemiker Nachkommastellen bzw. Genauigkeit einbüßen. Diese Lösung führt allerdings u. U. zu massiver Verschwendung von Speicherplatz (z. B. weil Astronomen im Extremfall gar keine Nachkommastellen benötigen und somit sämtliche Nachkommabits der Zahl Nullbits sind). Um den Konflikt Wertebreich vs. Genauigkeit zu lösen, betrachten wir im folgenden Abschnitt das Konzept der Gleitkommazahlen.

Gleitkommazahlen Eine Gleit- bzw. Fließkommazahl besteht ebenso wie eine Festkommazahl aus einer festgelegten Anzahl an Stellen. Allerdings ist die Position des Kommas hier nicht fest vorgegeben, sondern verschiebbar, wodurch wir uns nicht mehr zwischen großem Wertebereich und hoher Genauigkeit entscheiden müssen. Wie das Komma innerhalb einer Kommazahl „verschoben“ werden kann, ohne den Wert der Zahl zu verändern, ist Ihnen vermutlich bereits von der „Exponetialdarstellung“ bekannt. So zeigt das Beispiel äquivalente Darstellungen der Dezimalzahl $3,14$:

$$314 * 10^{-2} = 31,4 * 10^{-1} = \mathbf{3{,}14} = 0,314 * 10^{1} = 0,0314 * 10^{2}$$

Wie im Beispiel ersichtlich wird, ist die Gleitkommadarstellung nicht eindeutig. Um sicherzustellen, dass ein Bitmuster dennoch für genau eine Zahl steht, normalisieren wir die Zahl. Wir stellen die Zahl also so dar, dass die erste Ziffer, die direkt links vom Komma steht, keine Null ist. Im Beispiel entspricht $3,14$ der normalisierten Zahl.
Um eine Zahl im Gleitkommaformat darzustellen speichern wir folglich nur noch das Vorzeichen (sign s), die Ziffernfolge ohne führende Nullen (Mantisse m) und den Exponenten e, der die Verschiebung des Kommas ermöglicht. Die Basis b wird für alle Zahlen festgelegt. Der Wert z der Zahl ergibt sich dann nach:

$$z = (-1)^s \cdot m \cdot b^e$$

Dabei kann s entweder den Wert Null oder den Wert Eins annehmen. Für den Fall, dass z positiv ist, gilt $s = 0$, wegen $(-1)^0 = 1$. Ist z hingegen eine negative Zahl, so gilt $s = 1$, da $(-1)^1 = -1$. s wird im Prinzip analog zum Vorzeichenbit verwendet.
Für die computergeeignete Darstellung von Gleitkommazahlen ist die Basis b wiederum 2, womit die Mantisse m aus den Ziffern 0 und 1 zusammengesetzt ist. Die von Rechnern verwendeten Gleitkommaformate sind durch die Norm IEEE 754 standardisiert. Das einfach genaue Zahlenformat („single“; 32 Bit) und das doppelt genaue Zahlenformat („double“; 64 Bit) sind dem Standard nach wie in Abbildung 1.7 gezeigt aus Bitblöcken zusammengesetzt.

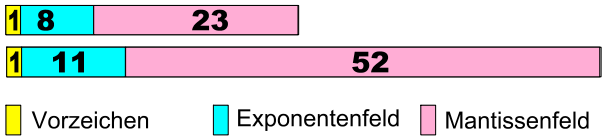


Abbildung 1.7: Gleitkommaformate „single“ und „double“

Das Vorzeichenbit wird für positive Zahlen auf `1` und für negative Zahlen auf `0` gesetzt. Zur Repräsentation des Exponenten stehen im single-Format acht Bit zur Verfügung, d. h. es sind 256 unterschiedliche Exponenten darstellbar. Für die Darstellung von negativen Exponenten (Verschiebung des Kommas nach links) stehen die Bitmuster `00000000` bis `01111111` zur Verfügung. Die Bitmuster `10000000` bis `11111111` repräsentieren positive Exponenten (Verschiebung des Kommas nach rechts). Diese Aufteilung gilt analog mit elf Bit für das double-Format. Auf den Exponenten folgt die normalisierte Darstellung der Mantisse mit 23 bzw. 52 Bit. Die Normalisierung einer Mantisse in Binärschreibweise führt günstigerweise dazu, dass die erste Ziffer links vom Komma, die keine Null ist, (fast) immer eine Eins ist. Diese führende Eins muss demnach nicht gespeichert werden und wird deshalb auch als „Hidden Bit“ bezeichnet. Aus der normalisierten Mantisse `1,0110` wird demnach `,0110`. Einzige Ausnahme ist dabei die Mantisse der Zahl Null.

Da Null, bedingt durch das Hidden Bit nicht darstellbar wäre, ist das Bitmuster, welches ausschließlich Nullen enthält, für die Zahl Null reserviert. Mithilfe des einfach genauen Formats können auf diese Weise Zahlen im Bereich von 10^{-38} bis 10^{38} dargestellt werden. Für das doppelt genaue Format liegen die darstellbaren Zahlen im Bereich 10^{-308} bis 10^{308}.

1.4 Boolesche Algebra

Nachdem wir uns bisher mit der Darstellung von Information beschäftigt haben, wollen wir in diesem Abschnitt die Basis für die Verarbeitung von Information beschreiben. Diese Basis, mit deren Hilfe wir Informationen verknüpfen können, ist die „Boolesche Algebra“. Die folgende Definition, die allgemein beschreibt, was eine Boolesche Algebra ist, stammt aus *Grundlagen der Technischen Informatik* [Hof14] und geht auf den Mathematiker Edward Vermilye Huntington zurück:

> Gegeben seien eine nicht leere Menge V sowie die beiden binären Operatoren $+ : V \times V \to V$ und $\bullet : V \times V \to V$. Das Tripel $(V, +, \bullet)$ ist genau dann eine *Boolesche Algebra*, wenn die folgenden vier *Huntington'schen Axiome* erfüllt sind:
>
> - Kommutativgesetze:
> Die Reihenfolge der Operanden darf vertauscht werden:
>
> $$x \bullet y = y \bullet x$$
> $$x + y = y + x$$
>
> - Distributivgesetze:
> Gleiche Elemente dürfen ausgeklammert werden:
>
> $$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$$
> $$x + (y \bullet z) = (x + y) \bullet (x + z)$$
>
> - Neutrale Elemente:
> Es existieren Elemente $e, n \in V$, für die gilt:
>
> $$x \bullet e = x$$
> $$x + n = x$$
>
> - Inverse Elemente:
> Für alle $x \in V$ existiert ein x^{-1}, für das gilt:
>
> $$x \bullet x^{-1} = n$$
> $$x + x^{-1} = e$$

Gegeben ist demnach eine Menge V mit beliebigen Elementen, beispielsweise $\{0, 1\}$, sowie die Operatoren $+$ und $\bullet$, die je zwei dieser Elemente so miteinander verknüpfen, dass wir

wieder ein Element aus der Menge V als Ergebnis erhalten. Wenn nun für die Menge in Verbindung mit den beiden Operatoren die genannten Axiome (Grundsätze/Festlegungen, die nicht herleitbar sind) gelten, so bilden $V, +$ und $\bullet$ eine Boolesche Algebra.

Versuchen wir nun, die Definition zum besseren Verständnis mit Leben zu füllen. Interpretieren wir beispielsweise $\bullet$ als gewöhnliches Multiplikationszeichen und $+$ als gewöhnliches Additionszeichen für das Rechnen mit reellen Zahlen, dann gilt die Kommutativität wie in der Definition beschrieben. Auch das zuerst genannte Distributivgesetz (Einklammern/Ausklammern) gilt beim Rechnen mit reellen Zahlen. Das zweite Distributivgesetzt gilt für die reellen Zahlen aufgrund der Punkt-vor-Strich-Regel allerdings nicht,d. h. die Menge der reellen Zahlen in Kombination mit den Operatoren $+$ und $\cdot$ ist keine Boolesche Algebra.
Eine andere Variante wäre die Interpretation von $+$ als Vereinigungsmenge $\cup$ und $\bullet$ als Schnittmenge $\cap$. Als Menge V bietet sich die Potenzmenge (Menge aller Teilmengen) einer beliebigen Grundmenge G an, um sicherzustellen, dass das Ergebnis der Operationen $\cup$ und $\cap$ wiederum in V liegt. Auch hier sind beide Kommutativgesetze gültig. Zusätzlich gelten hier, in Ermangelung eines Pendants zur Punkt-vor-Strich-Regel, beide Distributivgesetze. Als neutrales Element e der Schnittmenge bietet sich die Grundmenge G an, da der Operand x aus der Potenzmenge V gewählt ist und somit immer einer Teilmenge von G entspricht, wodurch das Ergebnis der Schnittmenge immer x ist. Das neutrale Element n der Vereinigung ist die leere Menge $\{\}$: Wenn wir zur Menge x „Nichts“ hinzufügen, erhalten wir als Ergebnis x unverändert zurück. Als inverses Element x^{-1} verwenden wir die Komplementärmenge $G \setminus x$ von x bzgl. der Grundmenge (also die Grundmenge G ohne die Elemente von x). Bei Verwendung der Komplementärmenge als inverses Element erhalten wir beim Schnitt von x mit $G \setminus x$ die leere Menge (entspricht n), weil es keine Überschneidung zwischen x und $G \setminus x$ gibt. Vereinigen wir x mit der Menge $G \setminus x$ so fügen wir im Prinzip zu x alle fehlenden Elemente aus G hinzu und erhalten somit G (entspricht e). Die Mengenlehre ist demnach eine Anwendung der Booleschen Algebra.

Neben der im Exkurs vorgestellten Mengenlehre ist auch die Aussagenlogik eine Anwendung der Booleschen Algebra. Da die Aussagenlogik die Grundlage der Schaltalgebra ist, die wir in Kapitel 2 für die Konstruktion von Logikschaltungen zur Informationsverarbeitung benötigen, betrachten wir sie im folgenden Abschnitt näher.

1.4.1 Anwendung: Aussagenlogik

Die klassische Aussagenlogik ist ein Teilgebiet der Logik in dem Eigenschaften von Aussagen, die mittels Aussagenverknüpfung aus anderen Aussagen entstehen, untersucht wer-

den. Aussagen sind beispielsweise:

1. 1 Meter entspricht 100 Zentimetern. ($\rightarrow$ `WAHR`)
2. Alle Menschen sind grün. ($\rightarrow$ `FALSCH`)

Anhand der Beispielaussagen wird bereits deutlich, dass in der klassischen Aussagenlogik das *Prinzip der Zweiwertigkeit* gilt:

Jede Aussage hat entweder den Wert `WAHR` oder den Wert `FALSCH`.

Es gibt also ausschließlich die Wahrheitswerte `WAHR` und `FALSCH`. Daraus ergibt sich, dass die Aussagenlogik eine zweielementige Boolesche Algebra mit der Menge $V = \{\texttt{WAHR}, \texttt{FALSCH}\}$ ist.
Des Weiteren gilt der *Satz vom ausgeschlossenen Dritten (tertium non datur)*:

Jede Aussage ist immer entweder wahr oder falsch.

Eine Aussage hat also in jedem Fall einen Wert, es gibt keine Aussagen mit unbestimmtem Wert. Außerdem gilt der *Satz vom ausgeschlossenen Widerspruch*:

Keine Aussage ist zugleich wahr und falsch.

Um mit der Aussagenlogik arbeiten zu können, muss außerdem das *Prinzip der Extensionalität* gelten:

Der Wahrheitswert einer Aussageverknüpfung hängt ausschließlich von den Wahrheitswerten ihrer Bestandteile ab.

Im Prinzip der Extensionalität ist die Rede von „Aussageverknüpfung“, d. h. wir benötigen Operatoren, um Aussagen (gemäß den Vorgaben der Booleschen Algebra) miteinander zu verknüpfen. Diese Operatoren sind die UND-Verknüpfung ($\wedge$) und die ODER-Verknüpfung ($\vee$), welche der Verknüpfung von je zwei Aussagen dienen. Zur Bildung des inversen Elements wird außerdem der Negationsoperator ($\neg$) benötigt, der den Wahrheitswert einer Aussage umkehrt. In den folgenden Abschnitten beschäftigen wir uns mit der Verknüpfung von Aussagen im Rahmen der Booleschen Algebra.

1.4.2 Boolesche Funktionen

Eine Boolesche Funktion ist eine Abbildungsvorschrift, die jedem n-Tupel von Zahlen aus der Menge $\{0, 1\}$ eindeutig ein m-Tupel von Zahlen aus $\{0, 1\}$ zuordnet. Wir ordnen also jeder der 2^n mit n Bit darstellbaren Dualzahlen eindeutig eine andere mit m Bit darstellbare Dualzahl zu. Die formale Definition für Boolesche Funktionen ist demnach Folgende:

Sei $B = \{0, 1\}$. Funktionen $f : B^n \to B^m$ mit $n, m \geq 1$ heißen Boolesche Funktionen.

Für den Fall, dass $m = 1$ gilt, sprechen wir von einer echten Booleschen Funktion. Eine echte Boolesche Funktion ordnet jedem n-Tupel von Elementen aus $\{0, 1\}$ eindeutig ein Element aus $\{0, 1\}$ zu. Dieses Element ist entweder `0` oder `1` und heißt Wahrheitswert. Tabelle 1.4 gibt einen Überblick über wichtige, echte, einstellige Boolesche Verknüpfungsfunktionen. „Einstellig“ bedeutet hier, dass die Funktion nur eine, mit `0` oder `1` zu belegende, Variable als Eingabeparameter erhält.

Tabelle 1.4: Einstellige Boolesche Verknüpfungsfunktionen

Bezeichnungen	Notation für $f(x)$	$f(0)$	$f(1)$
Identität	$f(x) = x$	0	1
Negation, not, nicht	$f(x) = \neg x = \bar{x}$	1	0
Einsfunktion	$f(x) = 1(x)$	1	1
Nullfunktion	$f(x) = 0(x)$	0	0

Die nachfolgende Tabelle 1.5 zeigt wichtige Beispiele für zweistellige, echte Boolesche Verknüpfungsfunktionen.

Tabelle 1.5: Zweistellige Boolesche Verknüpfungsfunktionen

Bezeichnungen	Funktion $f(x, y)$	$f(0,0)$	$f(1,0)$	$f(0,1)$	$f(1,1)$
Und, and, Konjunktion	$f(x, y) = x \wedge y = x \bullet y = xy$	0	0	0	1
Oder, or, Disjunktion	$f(x, y) = x \vee y = x + y$	0	1	1	1
Nicht-Und, nand	$f(x, y) = \overline{x \wedge y}$	1	1	1	0
Nicht-Oder, nor	$f(x, y) = \overline{x \vee y}$	1	0	0	0
Antivalenz, Exklusiv-Oder, xor	$f(x, y) = x \oplus y$	0	1	1	0
Äquivalenz	$f(x, y) = x \Leftrightarrow y$	1	0	0	1
Implikation	$f(x, y) = x \Rightarrow y$	1	0	1	1

In Tabelle 1.5 finden sich auch die Operatoren $\wedge$, $\vee$ und $\neg$ der Aussagenlogik wieder. Operatoren sind demnach Funktionen, die Aussagen verknüpfen und den entsprechenden Wahrheitswert berechnen. $\{\wedge, \vee, \neg\}$ ist ein logisch vollständiges Operatorensystem, weil mit den enthaltenen Operatoren alle anderen Funktionen dargestellt werden können. Weitere logisch vollständige Operatorensysteme Ω sind:

$\Omega = \{\vee, \neg\}$
$\Omega = \{\wedge, \neg\}$
$\Omega = \{nand\}$
$\Omega = \{nor\}$
$\Omega = \{Einsfunktion, \oplus, \wedge\}$

1.4.3 Nützliche Regeln

Zum Abschluss des Kapitels betrachten wir noch einige Rechenregeln der Aussagenlogik, die sich aus den Huntington'schen Axiomen (vgl. Abschnitt 1.4) herleiten lassen und sich für die in Kapitel 2 verwendete Schaltungslogik als praktisch erwiesen haben. Neben den Axiomen selbst sind folgende Regeln gültig:

- De Morgansche Regeln:

  $\neg(x \wedge y) = \neg x \vee \neg y$
  $\neg(x \vee y) = \neg x \wedge \neg y$

- Assoziativgesetze:

  $(x \vee y) \vee z = x \vee (y \vee z) = x \vee y \vee z$
  $(x \wedge y) \wedge z = x \wedge (y \wedge z) = x \wedge y \wedge z$
  $(x \oplus y) \oplus z = x \oplus (y \oplus z) = x \oplus y \oplus z$

- Idempotenz:

  $x \vee x = x$
  $x \wedge x = x$

- Absorptionsregeln:

  $x \vee \neg x = 1$
  $x \wedge \neg x = 0$
  $x \vee (x \wedge y) = x$
  $x \wedge (x \vee y) = x$
  $x \oplus x = 0$
  $x \oplus \neg x = 1$

- Substitution von Konstanten:

  $x \vee 0 = x$
  $x \vee 1 = 1$
  $x \wedge 0 = 0$
  $x \wedge 1 = x$
  $x \oplus 0 = x$
  $x \oplus 1 = \neg x$

Zusammenfassung

In diesem Kapitel haben Sie erfahren, dass Rechner Informationen binär darstellen und verarbeiten. Darauf aufbauend haben wir Möglichkeiten zur binären Darstellung von Text und Zahlen aufgezeigt. Zur Darstellung von Text haben wir die Codes ASCII, ISO 8859 und Unicode (sowie den weniger gebräuchlichen EBCDIC) kennengelernt, die einem Zeichen jeweils eindeutig eine binäre Darstellung zuweisen. Die Darstellung von Zahlen erfolgt auf Basis des Dualsystems sowie einiger Anpassungen desgleichen. Während positive ganze Zahlen lediglich durch einige Divisionen mit Rest in duale Zahlen überführt werden mussten, wurde zur Darstellung negativer ganzer Zahlen das Zweierkomplement herangezogen. Die Darstellung gebrochener Zahlen erfolgte im Gleitkommaformat. In Abschnitt 1.4 haben Sie schließlich die Boolesche Algebra und eine ihrer Anwendungen, die Aussagenlogik, kennengelernt. Aufbauend auf die Aussagenlogik und die in Abschnitt 1.4.3 vorgestellten Rechenregeln werden wir uns im nächsten Kapitel mit dem Aufbau logischer Schaltungen für die Informationsverarbeitung beschäftigen.

Aufgaben

Aufgabe 1

Dekodieren Sie folgenden ASCII-kodierten Text:
01001001 01101110 01100110 01101111 01110010 01101101 01100001 01110100 01101001 01101111 01101110

Aufgabe 2

Füllen Sie die Tabelle aus, indem Sie zwischen den Zahlensystemen umrechnen.

Dez	Bin	Okt	Hex
			4F
84			
77			
			52
	01000101		
			50
67			
		125	

Zusatzaufgabe

Wie lautet die in der Tabelle aus Aufgabe 2 versteckte Nachricht?

2 Von der Schaltungslogik zur Informationsverarbeitung

In diesem Kapitel wollen wir die in Kapitel 1.1 vorgestellten Funktionen der Aussagenlogik nutzen, um Schaltungen zur Verarbeitung von Information zu entwerfen. Abbildung 2.1 veranschaulicht unser Ziel anhand elektrischer Schaltkreise. Diese Schaltkreise bestehen aus Schaltern und Glühlampen, wobei der Zustand der Glühlampen (`aus`$\hat{=}$`0`/`an`$\hat{=}$`1`) von der Schalterstellung (`offen`$\hat{=}$`0`/`geschlossen`$\hat{=}$`1`) abhängt. Durch die Anordnung der Schalter wird dabei jeweils eine boolesche Funktion implementiert: Die Schalterstellung entspricht dabei den Eingabewerten und der Zustand der Glühlampe dem Ausgabewert.

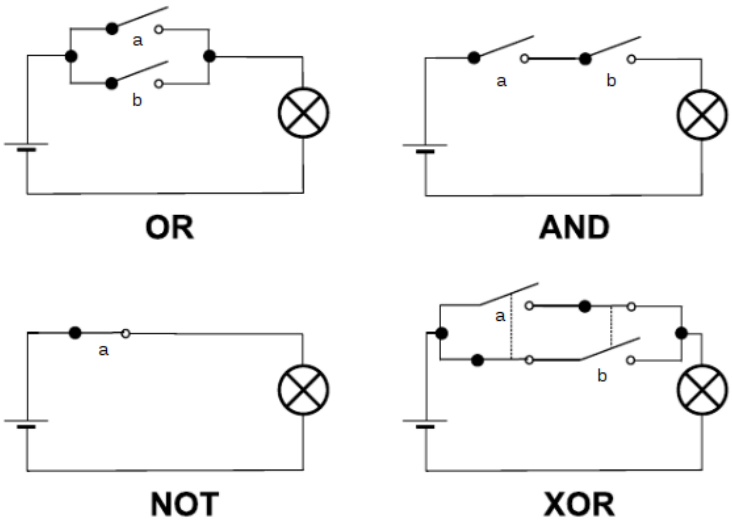


Abbildung 2.1: Elektrische Schaltungen für OR, AND, NOT und XOR

Die Glühlampe im OR-Schaltkreis leuchtet, sobald einer der Schalter geschlossen ist. Sie leuchtet auch, wenn beide Schalter geschlossen sind („OR“ entspricht hier also nicht dem umgangssprachlichen Konzept von „oder“, welches eher im Sinne von „entweder oder“ gebraucht wird). Im AND-Schaltkreis müssen beide Schalter (d. h. Schalter `a` *und* Schalter `b`) geschlossen sein, damit die Glühlampe leuchtet. Im XOR-Schaltkreis muss entweder `a` oder `b` geschlossen sein, damit die Lampe leuchtet (sind beide Schalter gleichzeitig offen (bzw. geschlossen), leuchtet die Lampe nicht). Die anderen beiden Schalter sind nicht als Eingabevariablen zu betrachten – sie dienen lediglich der Implementation der XOR-

Funktionalität. Um den Zusammenhang zwischen den gezeigten Schaltungen und den zugehörigen booleschen Funktionen zu verinnerlichen, ist das Betrachten der entsprechenden Wahrheitstabellen sinnvoll:

Tabelle 2.1: $f(a, b) = a \vee b$

a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 2.2: $f(a, b) = a \wedge b$

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 2.3: $f(a) = \neg a$

a	f
0	1
1	0

Tabelle 2.4: $f(a, b) = a \oplus b$

a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

Bezogen auf die Beispiele in Abbildung 2.1 werden in den Wahrheitstabellen 2.1 bis 2.4 alle möglichen Schalterkombinationen für die Schalter a und b dem daraus resultierenden Zustand f der Lampe gegenübergestellt.
Anhand des Beispiels haben wir gesehen, dass boolesche Funktionen durch elektrische Schaltkreise realisierbar sind. Grundsätzlich können boolesche Funktionen nicht nur durch elektrische Schaltkreise, sondern durch jede beliebige andere Technologie implementiert werden. So sind beispielsweise auch mechanische oder chemische Rechner denkbar. Rechner arbeiten in der Praxis natürlich nicht mit Schaltern und Lampen, sondern mit Transistoren. Auf die Funktionsweise der Transistorlogik auf elektronischer Ebene wollen wir hier nicht näher eingehen. Konzeptuell lässt sich das Ganze allerdings wie folgt betrachten: Transistoren sind elektronische Bauteile, die Schaltern ähneln. Ebenso wie Schalter haben auch Transistoren zwei Zustände: „sperrend“ und „leitend“. Als Eingabewerte werden die Eingänge einer Transistorschaltung mit Spannungswerten (`low`/`high`) belegt. Der Ausgabewert einer solchen Schaltung ist wiederum einer der beiden Spannungswerte. Die konkreten, physikalischen Spannungswerte für `low` und `high` hängen von der verwendeten Transistortechnologie (z. B. Transistor-Transistor-Logik (TTL) oder Complementary metal-oxide-semiconductor (CMOS)) ab.
Von diesem Punkt an ignorieren wir die technische Umsetzung von booleschen Funktionen und beschäftigen uns ausschließlich mit dem Entwurf von Schaltsystemen mithilfe der in Abbildung 2.2 gezeigten Logikgatter, die jeweils für eine bestimmte boolesche Funktion stehen:

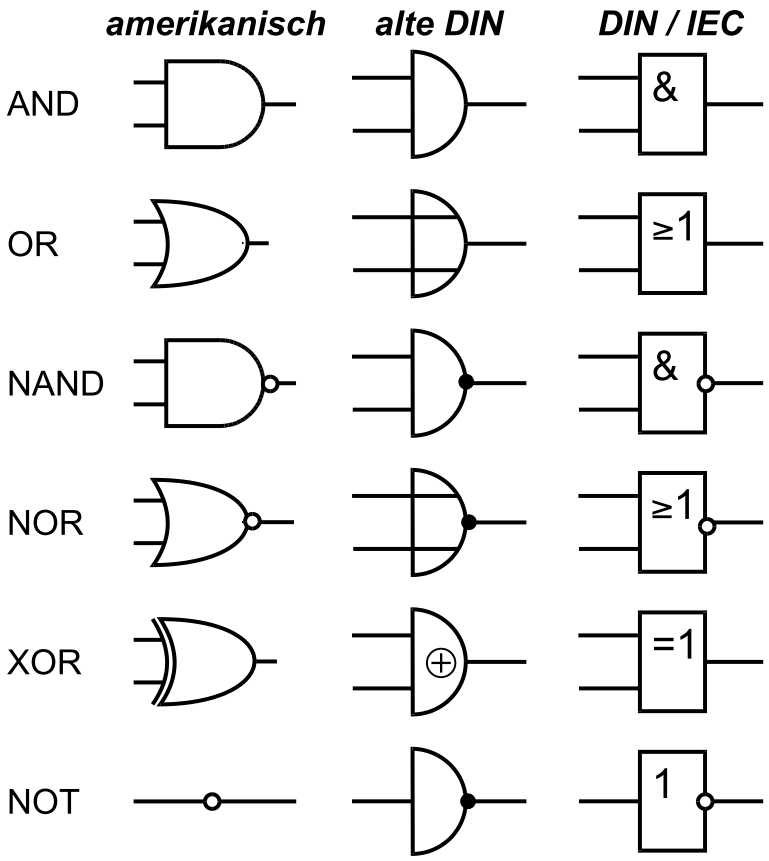


Abbildung 2.2: Logikgatter

Im amerikanischen bzw. englischen Raum finden fast ausschließlich die amerikanischen Symbole Verwendung. Im europäischen Raum werden heute hauptsächlich die Symbole nach International Electrotechnical Commission (IEC) gebraucht. Auch in diesem Skript werden wir die Symbole nach IEC verwenden.
Um Schaltsysteme zu entwerfen und die Untergliederung dieses Kapitels zu verstehen, betrachten wir zunächst die Taxonomie der Schaltsysteme, wie sie in Abbildung 2.3 dargestellt ist.

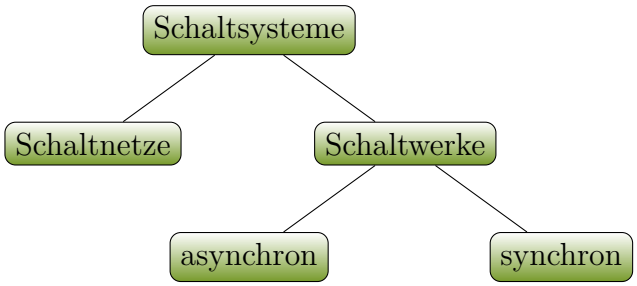


Abbildung 2.3: Taxonomie Schaltsysteme

Logikschaltungen bzw. Schaltsysteme sind Zusammensetzungen von Gattern (vgl. Abbildung 2.2). Schaltsysteme können anhand ihres Speicherverhaltens in Schaltnetze und

Schaltwerke unterschieden werden. Schaltwerke werden zusätzlich anhand ihres Zeitverhaltens in asynchrone und synchrone Schaltwerke unterteilt. In den folgenden Abschnitten betrachten wir die Eigenschaften von Schaltnetzen und Schaltwerken näher.

2.1 Schaltnetze

Schaltnetze sind zustandslose Logikschaltungen, also aus Logikgattern zusammengesetzte Schaltungen ohne Speicherverhalten und damit ohne Rückkopplungen. Eine Rückkopplung ist das Zurückführen eines Ausgangs auf einen Eingang der Schaltung, beispielsweise um den Ausgabewert innerhalb der Schaltung zu speichern und bei Bedarf wieder abzurufen. Die Ausgabe eines Schaltnetzes hängt demnach zu jedem Zeitpunkt ausschließlich von den Eingabewerten ab.
Im folgenden Beispiel beschäftigen wir uns mit dem Entwurf eines Schaltnetzes, d. h. wir wollen eine boolesche Funktion mithilfe einer Logikschaltung beschreiben. Wir entwerfen ein Schaltnetz, welches die boolesche Funktion XOR mit den Eingangsvariablen a und b sowie Ausgabe f realisiert. Allerdings stehen für die Umsetzung ausschließlich NAND-Gatter zur Verfügung. Wie wir wissen, ist es mithilfe logisch vollständiger Operatormengen (vgl. Abschnitt 1.4.2) möglich, jede beliebige boolesche Funktion zu realisieren. Die Funktion NAND ist für sich genommen bereits eine logisch vollständige Operatormenge. Um ein Schaltnetz zu zeichnen, benötigen wir zunächst die entsprechende Funktionsgleichung. Dazu wird die gewünschte Funktion $f = a \oplus b$ anhand der Rechenregeln der booleschen Algebra (vgl. 1.4.3) in eine äquivalente Gleichung transformiert, die als Operator ausschließlich den NAND-Operator enthält:

$$
\begin{aligned}
x \oplus y &= (x \wedge \overline{y}) \vee (\overline{x} \wedge y) && \textit{Absorptionsgesetz} \\
&= (x \wedge \overline{y}) \vee (\overline{x} \wedge y) \vee (x \wedge \overline{x}) \vee (y \wedge \overline{y}) && \textit{Distributivgesetz} \\
&= (x \wedge (\overline{x} \vee \overline{y})) \vee (y \wedge (\overline{x} \vee \overline{y})) && \textit{DeMorgan} \\
&= (x \wedge (\overline{x \wedge y})) \vee (y \wedge (\overline{x \wedge y})) && \textit{DeMorgan} \\
&= \overline{\overline{(x \wedge (\overline{x \wedge y}))} \wedge \overline{(y \wedge (\overline{x \wedge y}))}}
\end{aligned}
$$

Im ersten Schritt transformieren wir den Ausdruck $x \oplus y$ in einen äquivalenten Ausdruck, der nur die Operatoren $\wedge$, $\vee$ und $\neg$ enthält, um die Rechenregeln aus Abschnitt 1.4.3 anzuwenden. Den Absorptionsregeln entsprechend fügen wir diesem Ausdruck im nächsten Schritt die Terme $(x \wedge \overline{x})$ ($\hat{=}0$) und $(y \wedge \overline{y})$ ($\hat{=}0$) hinzu. Mithilfe des Distributivgesetzes werden nun die beiden Terme, die ein nicht negiertes x enthalten, zu einem Term zusammengefasst. Selbiges geschieht auch mit den beiden Termen, die ein nicht negiertes y

enthalten. In der nächsten Zeile wird die De Morgansche Regel auf die innersten beiden Klammerausdrücke angewendet, womit wir diese Terme bereits mittels des gewünschten NAND-Operators ausdrücken. Abschließend wird die De Morgansche Regel auf den gesamten Ausdruck angewandt, um auch den $\vee$-Operator in der Mitte in ein NAND zu überführen. Anhand des so erstellten Ausdrucks wird die Logikschaltung, beispielsweise wie in Abbildung 2.4, erstellt:

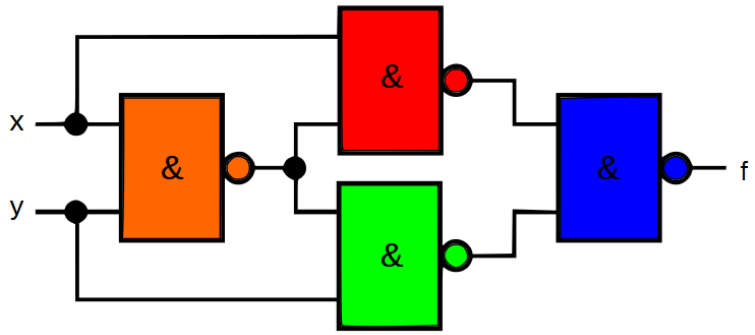


Abbildung 2.4: XOR

Die Farben der NAND-Gatter in der Abbildung entsprechen den Farben der Terme im Ausdruck. Im booleschen Ausdruck fällt auf, dass der gelbe Term zweimal auftritt. Aus diesem Grund sind für die Logikschaltung nicht die durch die Gleichung suggerierten fünf Gatter notwendig sondern es genügen durch den doppelt auftretenden Term bereits vier.

2.2 Schaltwerke

Schaltwerke sind im Gegensatz zu Schaltnetzen zustandsbehaftete Logikschaltungen, also Schaltungen mit speicherndem Verhalten. Die Ausgabe eines Schaltwerks hängt demnach nicht nur von den anliegenden Eingabewerten ab, sondern auch von intern gespeicherten Werten. Ein Schaltwerk enthält immer ein Schaltnetz und mindestens eine Rückkopplung. Sofern ein Schaltwerk als Eingabe ein Taktsignal erhält, sprechen wir von einem synchron arbeitenden Schaltwerk. Hat das Schaltwerk keinen Takteingang, so arbeitet das Schaltwerk asynchron.

2.2.1 Taktsteuerung

Das Taktsignal ist ein binäres Signal, durch das die Zustände `Low`, logisch `0`, und `High`, logisch `1`, abgebildet werden können. Taktsignale werden aufgrund ihres Verlaufs auch als Rechtecksignale bezeichnet. Abbildung 2.5 zeigt ein Taktsignal.

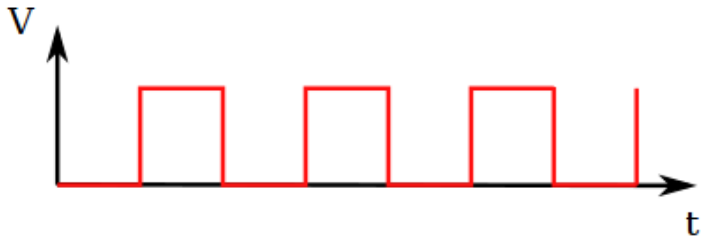


Abbildung 2.5: Rechtecksignal

In Schaltsystemen dient die Taktsteuerung der Synchronisation von Abläufen innerhalb bzw. zwischen Schaltwerken. Die Steuerung eines Systems kann entweder taktzustandsgesteuert oder taktflankengesteuert erfolgen. Abbildung 2.6 zeigt sowohl die Zustände des Signals als auch die Flanken.

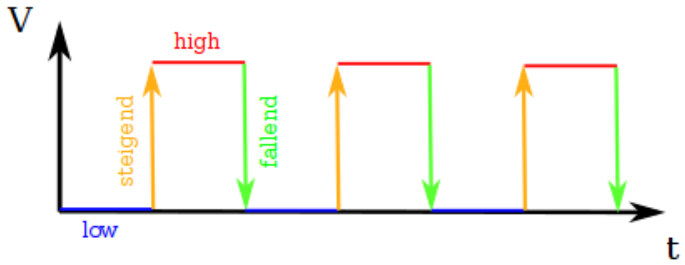


Abbildung 2.6: Zustände und Flanken eines Taktsignals

Taktzustandsgesteuert bedeutet, dass eine Schaltaktion stattfindet, solange ein aktives Taktsignal anliegt, d. h. der Takt den Wert `High` (rot dargestellt) hat. Taktflankengesteuert bedeutet hingegen, dass eine Schaltaktion genau dann ausgeführt wird, wenn ein Zustandsübergang des Taktsignals stattfindet. Ein Übergang von `Low` nach `High` wird als „steigende Taktflanke“ (orange dargestellt) bezeichnet. Der Zustandswechsel von `High` nach `Low` heißt „fallende Taktflanke“ (grün dargestellt).

2.2.2 Flip-Flops

Ein Beispiel für Schaltwerke sind Flip-Flops. Flip-Flops sind einfache Speicherelemente, die die Zustände `0` und `1` speichern. Es gibt sowohl synchrone als auch asynchrone Flip-Flops. Die synchronen Flip-Flops werden außerdem in taktflankengesteuerte und taktzustandsgesteuerte Elemente unterschieden. Letztere werden auch als Latches bezeichnet. Wir benötigen Flip-Flops im weiteren Verlauf für Register und Arbeitsspeicher zum Speichern von Informationen durch den Rechner. Nachfolgend betrachten wir drei Arten von Flip-Flops.

Asynchrones RS-Flip-Flop

Abbildung 2.7 zeigt ein nicht taktgesteuertes RS-Flip-Flop (Reset-Set-Flip-Flop) und Tabelle 2.5 die zugehörige Wahrheitswerttabelle.

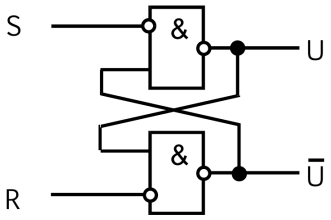


Abbildung 2.7: RS-Flip-Flop

Tabelle 2.5: Wahrheitswerttabelle RS-Flip-Flop

S	R	U_{neu}	$\overline{U_{neu}}$
1	0	1	0
0	1	0	1
0	0	U_{alt}	$\overline{U_{alt}}$
1	1	„verboten“	„verboten“

Die in Abbildung 2.7 dargestellte Variante des RS-Flip-Flops besteht aus zwei NAND-Gattern. Die Eingangssignale `S` (Setzen) und `R` (Rücksetzen) werden negiert in die Schaltung eingebracht. Das Flip-Flop ist ein Schaltwerk, weil es zwei Rückkopplungen enthält: Nämlich die Rückkopplung des Ausgangs U auf einen Eingang des unteren NAND-Gatters und eine Rückkopplung des Ausgangs $\overline{U}$ auf einen Eingang des oberen NAND-Gatters. Durch diese beiden Rückkopplungen wird der Ausgabewert U (sowie $\overline{U}$, der negierte Wert von U) in der Schaltung gespeichert. Betrachten wir nun die möglichen Eingangsbelegungen im Detail:

- Setzen: Ausgang U (also der durch das Flip-Flop gespeicherte Wert) wird auf `1` gesetzt.
- Rücksetzen: Ausgang U wird auf `0` gesetzt.
- Speichern: Der durch das Setzen bzw. Rücksetzen an U anliegende Wert bleibt durch die Rückkopplung erhalten.
- „Verboten“: Werden beide Eingänge gleichzeitig mit `1` beschaltet, so haben U und $\overline{U}$ den Wert `0`, was widersprüchlich ist. Zudem führt diese Eingangsbelegung dazu, dass der nachfolgende Zustand des Flip-Flops (und damit auch der Wert, der an U bzw. $\overline{U}$ anliegt) unbestimmt ist. Auch aus semantischer Sicht ist diese Belegung nicht sinnvoll, weil der zu speichernde Wert gleichzeitig `0` und `1` wäre.

Synchrones D-Flip-Flop

Beim in Abbildung 2.8 dargestellten D-Flip-Flop wird der am Dateneingang D anliegende Wert x (`0` oder `1`) auf die fallende Taktflanke des Taktsignals T als Speicherwert übernommen. Anschließend steht der gespeicherte Wert so lange an Ausgang Q zur Verfügung, bis sich der Wert an D ändert und der neue Wert wiederum mit der fallenden

Flanke übernommen wird. Die Wahrheitswerttabelle 2.6 fasst das beschriebene Verhalten zusammen.

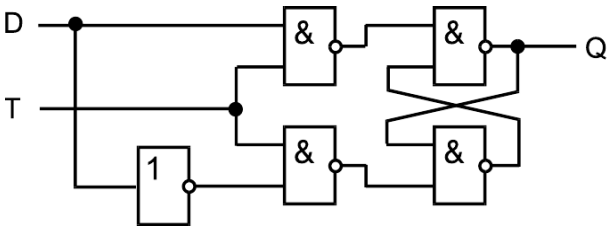


Abbildung 2.8: D-Flip-Flop

Tabelle 2.6: Wahrheitswerttabelle D-Flip-Flop

D	T	Q_{neu}
x	0	Q_{alt}
1	1	1
0	1	0

$T = 0$ bedeutet hier, dass gerade keine fallende Taktflanke anliegt und der Wert x an Eingang D demnach nicht übernommen wird. Der Ausgabewert Q bleibt in diesem Fall also erhalten. Liegt an Dateneingang D `1` an und die fallende Flanke kommt, so wird `1` übernommen und auf Q gelegt. Analog gilt für den Fall, dass an D `0` anliegt und die fallende Flanke kommt, dass der zu speichernde und über Q abzugreifende Wert anschließend `0` ist. Das durch die Wahrheitswerttabelle definierte Verhalten des Flip-Flops in Abhängigkeit des Takts zeigt Abbildung 2.9.

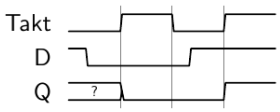


Abbildung 2.9: Ausgang Q in Abhängigkeit von Takt- und Dateneingang des D-Flip-Flops

Synchrones Master-Slave-Flip-Flop

Wir betrachten das in Abbildung 2.10 dargestellte Master-Slave-Flip-Flop.

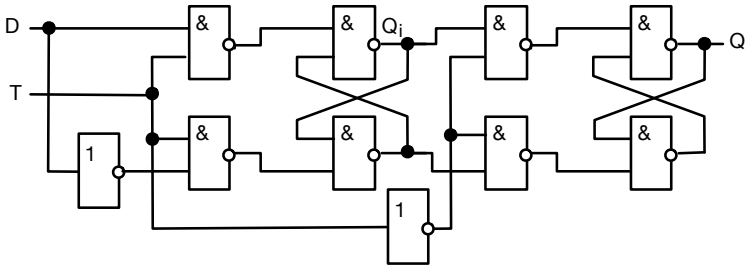


Abbildung 2.10: MS-Flip-Flop

Dieses MS-Flip-Flop besteht aus zwei über einen Negator zusammengeschalteten D-Flip-Flops. Mit der steigenden Taktflanke wird der an D anliegende Wert durch die erste Stufe des Flip-Flops, das Master-Flip-Flop, übernommen und steht anschließend an Ausgang

Q_i zur Verfügung. Q_i ist dabei der Ausgang der i-ten Stufe. Am Ausgang Q liegt noch immer der alte, zuvor gespeicherte Wert an. Mit der fallenden Taktflanke wird der Wert schließlich auch in die zweite Stufe, das Slave-Flip-Flop, übernommen. Erst jetzt liegt der neue Wert von D am Ausgang Q an. Aufgrund der beschriebenen Arbeitsweise werden MS-Flip-Flops als zweiflankengesteuert bezeichnet. Das Taktverhalten des MS-Flip-Flops in Abhängigkeit des Taktsignals zeigt Abbildung 2.11.

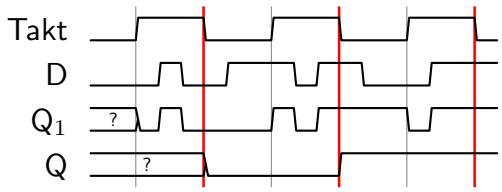


Abbildung 2.11: Ausgänge Q und Q_i des Master-Slave-Flip-Flops in Abhängigkeit des Taktsignals

2.2.3 Register

Im vorherigen Abschnitt haben wir uns mit der Funktionsweise von einfachen Speicherelementen vertraut gemacht, die genau 1 Bit speichern. Hier betrachten wir nun, wie wir diese Elemente so zu Registern verschalten, dass wir mehrere logisch zusammengehörende Bits (z. B. Dualzahlen) abspeichern können. Abbildung 2.12 zeigt ein solches, aus D-Flip-Flops zusammengesetztes Register und Abbildung 2.13 das zugehörige Blockschaltbild.

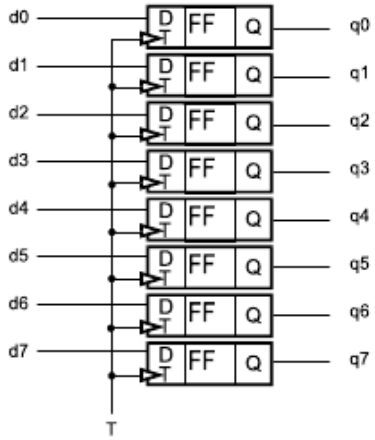


Abbildung 2.12: Register aus D-Flip-Flops

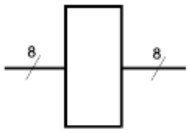


Abbildung 2.13: Blockschaltbild eines 8-Bit-Registers

Das dargestellte Register besteht aus acht unverbundenen, aber gemeinsam getakteten Flip-Flops. Da jedes der Flip-Flops ein Bit speichert, können wir im dargestellten Register eine achtstellige Dualzahl speichern.

2.2.4 Arbeitsspeicher

In den ersten Rechnern standen zur Verarbeitung von Informationen lediglich einige der in Abschnitt 2.2.3 vorgestellten Register zur Verfügung. Das auszuführende Programm und die benötigten Daten waren anfangs auf Lochkarten gespeichert. In heutigen Rechnern werden Programm und Daten durch den Arbeitsspeicher zur Verfügung gestellt. Abbildung 2.14 zeigt das Blockschaltbild eines Speichermoduls.

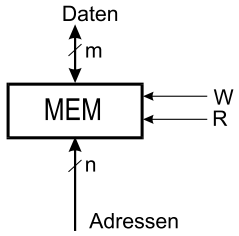


Abbildung 2.14: Blockschaltbild eines Speichermoduls

Abstrakt betrachtet ist der Arbeitsspeicher eine lineare Liste von Speicherzellen, auf die unter Angabe einer Adresse zugegriffen werden kann. Im Blockschaltbild entspricht „MEM“ der Liste der Speicherzellen. Eine Möglichkeit diese Liste von Speicherzellen technisch zu realisieren sind Halbleiterspeicher, bei welchen sich Millionen von Flip-Flops auf einem Chip befinden. Über den Eingang „Adressen“ suchen wir mithilfe eines n-stelligen Bitvektors die Speicherzelle aus, auf deren Inhalt zugegriffen werden soll. Über die beiden Eingänge „R“ (read) und „W“ (write) wird festgelegt, ob der Inhalt der gewählten Speicherzelle gelesen oder geschrieben werden soll. Liegt das `R`-Signal an, so wird der gelesene Speicherzelleninhalt auf den m Bit breiten Datenausgang gelegt. Liegt hingegen das `w`-Signal an, wird der am Dateneingang anliegende Wert an die entsprechende Adresse geschrieben. Typischerweise entspricht die Datenwortbreite m 32 oder 64 Bit und die Adressbreite n 24 Bit. Mit 24 Bit können $2^{24} \approx 17 Mio.$ Datenworte adressiert werden. Abbildung 2.15 zeigt eine kleine Multiplexerschaltung, die der Auswahl einer Speicherzelle über ihre Adresse dient.

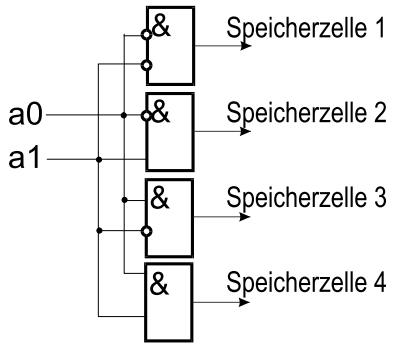


Abbildung 2.15: Adressdecodierung durch Multiplexerschaltung

Die Breite n des Adressvektors entspricht in diesem Fall exemplarischen 2 Bit (`a0` und `a1`). Über die $2^2 = 4$ Bitkombinationen können demnach 4 Speicherzellen angesprochen werden.

2.2.5 Schieberegister

Ein Schieberegister besteht aus mehreren, hintereinander geschalteten und gemeinsam getakteten Flip-Flops. Abbildung 2.16 zeigt ein aus MS-Flip-Flops bestehendes Schieberegister.

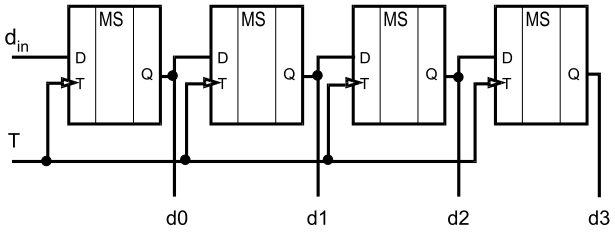


Abbildung 2.16: Schieberegister aus MS-Flip-Flops

Mit jedem Takt wird das Datenwort, bestehend aus den Werten der Ausgänge `d0`, `d1`, `d2` und `d3`, um eine Stelle verschoben und in das „freigewordene“ Flip-Flop der Wert des Eingangsbits `din` geschrieben. Tabelle 2.7 zeigt das Datenwort in Abhängigkeit von `din` nach jedem Zeitschritt.

Tabelle 2.7: Inhalt eines Schieberegisters in Abhängigkeit von Takt- und Dateneingang

t	din	d3 d2 d1 d0
0	1	0000
1	1	0001
2	0	0011
3	1	0110
4	1	1101

Schieberegister ermöglichen auf einfachem Wege die Multiplikation mit 2 und die Division durch 2. Für den Fall, dass `din` gleich `0` ist, entspricht das Schieben in Richtung Most Significant Bit (MSB) einer Multiplikation mit 2. In Tabelle 2.7 ist der Dateneingang in Takt 2 auf `0`. Mit dem nächsten aktiven Takt wird diese `0` an das Datenwort aus Takt 2 (`0011` $\hat{=}3_{DEC}$) angehängt. Das neue Datenwort in Takt 3 ist dann `0110`, was 6_{DEC} entspricht. Das Schieben in Richtung Least Significant Bit (LSB) führt analog zu einer Division durch 2.

2.2.6 Addition

Nachdem wir Information nun speichern können, betrachten wir in den nachfolgenden Abschnitten die Verarbeitung von Information. Abbildung 2.17 zeigt ein Schaltnetz, dass die Addition zweier Binärzahlen, `x` und `y`, durch logische Verknüpfungen realisiert.

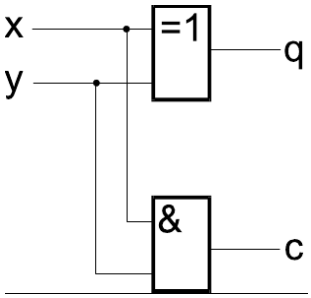


Abbildung 2.17: Halbaddierer

Tabelle 2.8: Wahrheitswerttabelle Halbaddierer

x	y	c	q
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Auf Ausgang q liegt das Ergebnis der Addition an und Ausgang c entspricht einem evtl. entstandenen Übertrag („Carry“), der in die nächsthöhere Stelle mit einfließt. Abbildung 2.18 zeigt das Blockschaltbild eines solchen Halbaddierers.

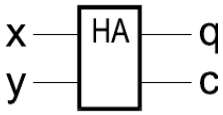


Abbildung 2.18: Halbaddierer

Zur vollständigen Berechnung einer Ergebnisstelle muss bei der Addition zweier mehrstelliger Dualzahlen zusätzlich der Übertrag aus der vorhergehenden Stelle berücksichtigt werden. Abbildung 2.19 zeigt eine mögliche Realisierung dieses Verhaltens.

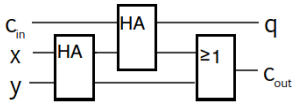


Abbildung 2.19: Volladdierer

Tabelle 2.9: Wahrheitswerttabelle Volladdierer

x	y	c_{in}	q	c_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Dem Schaltnetz in Abbildung 2.19 zufolge werden zunächst die beiden Eingabewerte x und y mithilfe eines Halbaddierers addiert. Anschließend wird zum so erhaltenen Ergebniswert der Übertrag c_{in} aus der vorherigen Addition addiert. Als Ergebnisstelle erhalten wir q. Mithilfe des OR-Gatters wird geprüft, ob bei mindestens einer der Additionen ein Übertrag entstanden ist. Wenn ein Übertrag entstanden ist, so liegt anschließend an c_{out} der Wert `1` an und sonst `0`.
Da wir nicht nur einzelne Stellen, sondern Dualzahlen beliebiger Breite addieren wollen, schalten wir mehrere Volladdierer, wie in Abbildung 2.20 dargestellt, zusammen. Die Dualzahlen werden dabei als Zahlen im Zweierkomplement aufgefasst.

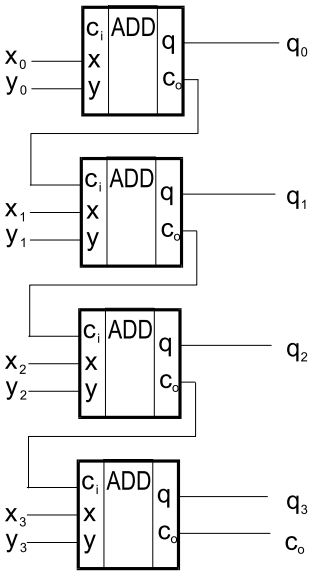


Abbildung 2.20: 4-Bit-Addierer

Das Carry-Bit c_o wird immer zur nächsthöheren Stufe weitergereicht. Deswegen muss das „Einschwingen“ aller n (in Abbildung: $n = 4$) Stufen abgewartet werden, bis ein gültiges Ergebnis anliegt.

2.2.7 Akkumulation

Akkumulation ist die fortgesetzte Addition beliebig vieler Zahlen in einem Akkumulator, d. h. einem Register, dass die Ergebnisse der vorherigen Additionen sammelt. Abbildung 2.21 zeigt eine Akkumulatorschaltung, bestehend aus der Addiererschaltung „ADD“und dem Akkumulatorregister „AC“. Als Eingabewert erhält die Schaltung einen n Bit breiten Vektor. Ausgabe sind der Wert des Akkumulatorregisters und das Carry-Bit `C`.

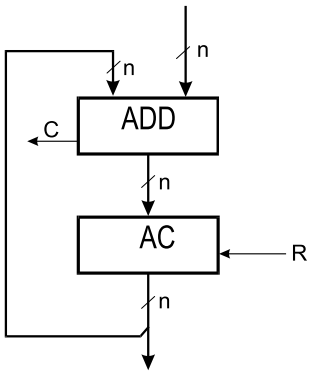


Abbildung 2.21: Akkumulator-Schaltung

Die Addition mehrerer Zahlen erfolgt durch die dargestellte Schaltung wie folgt:

1. Inhalt des Akkumulatorregisters AC mittels Eingang `R` (Reset) auf `0` setzen
2. Eingangswert x von ADD und den Inhalt von AC addieren sowie das Ergebnis in AC speichern
3. Wenn weitere Werte x aufaddiert werden sollen, gehe zu 2.

2.2.8 Multiplikation

Wir betrachten im Folgenden zwei Möglichkeiten der Multiplikation beliebiger Dualzahlen. Die erste Variante ist das Multiplikationsarray, die zweite Variante die Multiplikation mittels Schieberegister. Da das Verfahren mittels Multiplikationsarray auf der Multiplikation mit „Papier und Bleistift“ basiert, rufen wir uns zunächst anhand eines Beispiels die schriftliche Multiplikation ins Gedächtnis:

$$\begin{array}{cccccc} 1 & 1 & \cdot & 1 & 3 & \\ \hline & & & 1 & 1 & \\ & & + & 3 & 3 & \\ \hline & & & 1 & 4 & 3 \end{array}$$

Um zwei Zahlen schriftlich zu multiplizieren, multiplizieren wir also jede Stelle des einen Faktors mit jeder Stelle des anderen Faktors. Anschließend summieren wir die so erhaltenen Partialprodukte auf. Dieses Prinzip kann analog für die Multiplikation von Dualzahlen verwendet werden:

$$\begin{array}{ccccccccc} 1 & 0 & 1 & 1 & \cdot & 1 & 1 & 0 & 1 \\ \hline & & 1 & 0 & 1 & 1 & & & \\ & & & 1 & 0 & 1 & 1 & & \\ & & & & 0 & 0 & 0 & 0 & \\ + & & & & 1 & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

Dieses Prinzip der schriftlichen Multiplikation liegt der Schaltung in Abbildung 2.22 zugrunde. Jede der einzelnen Zellen dieser Schaltung ist wie in Abbildung 2.23 gezeigt aufgebaut.

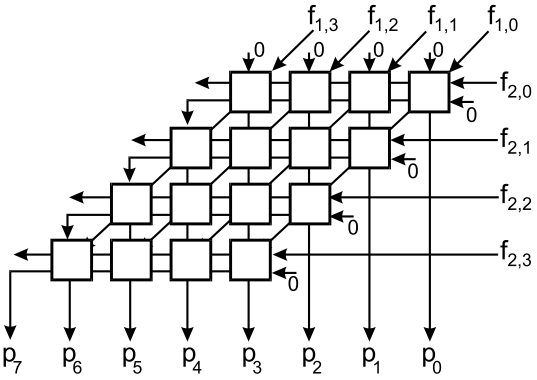


Abbildung 2.22: Multiplikationsarray

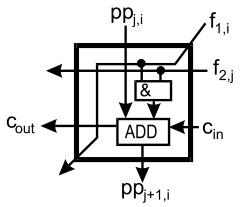


Abbildung 2.23: Einzelne Zelle des Multiplikationsarrays

Das Multiplikationsarray erhält als Eingabe die Faktoren f_1 und f_2 mit den Stellen 0 (LSB) bis 3. In jeder der quadratischen Zellen wird ein Partialprodukt $pp_{j,i}$, also das Produkt einer Stelle j von f_1 mit einer Stelle i von f_2, gebildet. Dieses Partialprodukt wird an die direkt unterhalb befindliche Zelle weitergereicht und zum Partialprodukt dieser Zelle addiert. Dabei entstehende Überträge werden über c_{out} an die linke Nachbarzelle weitergereicht, um in die Summe der nächsthöheren Stelle einzugehen. Diese Vorgehensweise führt allerdings zu hohem Rechenaufwand, da für die Multiplikation von Faktoren der Länge n n^2 Partialprodukte bzw. Zellen notwendig sind.
Aufgrund des hohen Rechenaufwandes bei der Multiplikation mittels Multiplikationsarray, betrachten wir ein weiteres Verfahren: die Multiplikation mit Schieberegister. Abbildung 2.24 zeigt die zugehörige Schaltung.

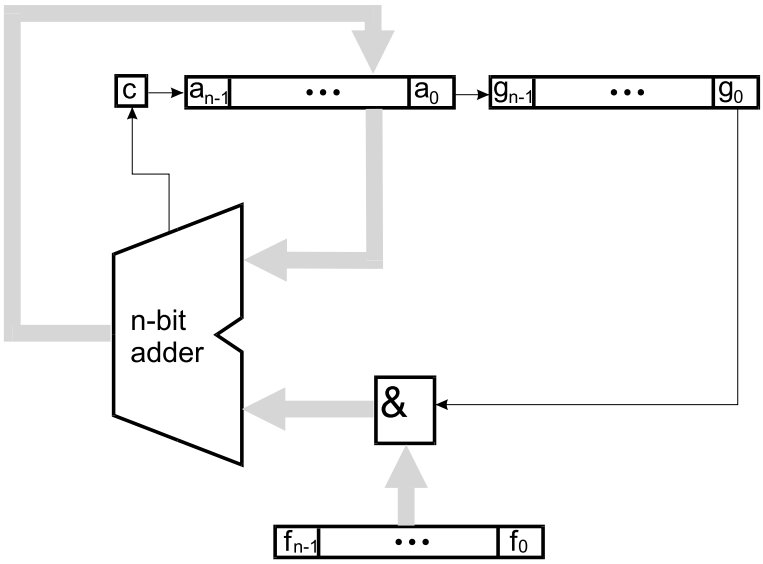


Abbildung 2.24: Multiplikation durch Schieberegister

Die Faktoren befinden sich zum Startzeitpunkt in den Registern `f` und `g`. Das Register `a` bildet mit `g` ein Doppelregister und wird mit `0` initialisiert. Der Übertrag `c` ist ebenfalls mit `0` initialisiert. Das Gesamtergebnis der Multiplikation wird am Ende der folgenden Prozedur im Doppelregister `ag` stehen:

1. Wenn g_0 gleich `1` ist, wird `f` nach `a` geladen.
2. Das Doppelregister `ag` wird um eine Stelle nach rechts geschoben, das LSB des Ergebnisses steht damit in g_{n-1}.
3. Wenn (das neue) g_0 gleich `1` ist wird `f` zu dem Wert in `a` addiert und das Ergebnis nach `a` geladen
4. Das Doppelregister `ag` wird um eine Stelle nach rechts geschoben. Gehe zu 1.

Zusammenfassung

In diesem Kapitel haben Sie die Realisierung von booleschen Funktionen mithilfe von Logikgattern kennengelernt. Darauf aufbauend haben wir den Entwurf und die Funktionsweise von diversen Schaltnetzen und Schaltwerken betrachtet, die als Speicher- und Rechentechnik in Computern zum Einsatz kommen können. Mithilfe der hier vorgestellten Komponenten, insbesondere Register, Arbeitsspeicher und n-Bit-Addierer, schauen wir uns im nächsten Kapitel Aufbau und Arbeitsweise eines Rechners an.

3 Von-Neumann-Rechner

Aus den vorangegangenen Kapiteln wissen wir, wie Daten dargestellt, gespeichert und logisch bzw. arithmetisch durch Schaltsysteme verarbeitet werden können. Allerdings haben wir diese Funktionalitäten durch die feste Verdrahtung von Schaltelementen erreicht, was problematisch ist, weil zum einen Algorithmen teilweise nur unter großem Aufwand verdrahtbar sind und zum anderen ein Computer nicht für jeden erdenklichen Algorithmus ein fest verdrahtetes Schaltsystem mitbringen kann. Unser Ziel in diesem Kapitel ist deswegen ein Allzweck-Rechner, der flexibel programmierbar und somit für die unterschiedlichsten Aufgaben und Eingabedaten einsetzbar ist. Was uns dafür noch fehlt, ist die Angabe und Ausführung eines Programmablaufs, ðdie Steuerung der Komponenten sowie deren Zusammenspiel durch eine frei wählbare Abfolge von Befehlen zur Informationsverarbeitung. Nachfolgend beschäftigen wir uns zunächst mit dem grundsätzlichen Ablauf von Informationsverarbeitung, um anschließend Aufbau und Arbeitsweise eines Rechners zu verstehen.

3.1 Informationsverarbeitung

Abbildung 3.1 zeigt die drei Stufen der Informationsverarbeitung.

Abbildung 3.1: Ablauf der Informationsverarbeitung

Abstrakt betrachtet läuft die Verarbeitung von Information demnach in den drei Phasen Eingabe, Verarbeitung und Ausgabe ab. Diese Phasen sollten sich in Geräten zur Informationsverarbeitung, wie dem im folgenden Abschnitt vorgestellten Von-Neumann-Rechner, widerspiegeln.

3.2 Von-Neumann-Rechner

Der grundsätzliche Aufbau eines universellen Computers geht auf den österreichisch-ungarischen Mathematiker John von Neumann (eigentlich Margittai Neumann János Lajos) zurück. Die nach ihm benannten Prinzipien des Rechnerentwurfs veröffentlichte er erstmals, nachdem er für das Manhattan-Projekt beim Entwurf des Electronic Discrete Variable Automatic Computer (EDVAC) mitgearbeitet hatte. Die meisten heute verwendeten Rechner haben, der Von-Neumann-Architektur entsprechend, den in Abbildung 3.2 gezeigten Aufbau.

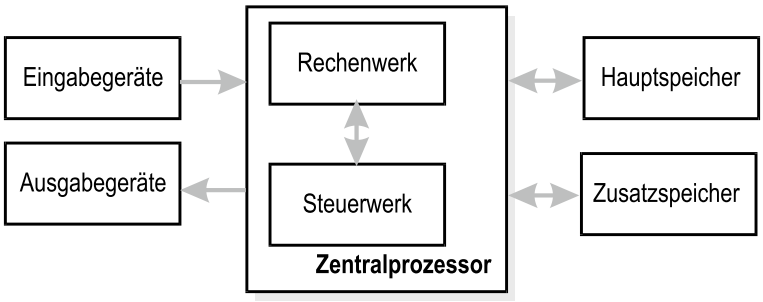


Abbildung 3.2: Von-Neumann-Architektur

Wie in der Abbildung deutlich wird, folgt die dargestellte Architektur tatsächlich den Stufen der Informationsverarbeitung. Zunächst findet die Eingabe von Daten über die Eingabegeräte statt. Anschließend werden die Daten durch den Zentralprozessor (auch bekannt als Central Processing Unit (CPU)), welcher aus Rechen- und Steuerwerk besteht, verarbeitet und gegebenenfalls durch die Speichermodule gespeichert. Die Ausgabe erfolgt anschließend über die Ausgabegeräte.
Neben dem bereits hier vorgestellten Aufbau des Von-Neumann-Rechners gibt es sieben weitere Prinzipien des Rechnerentwurfs nach von Neumann, die nachfolgend aufgezeigt werden.

3.2.1 Von-Neumann-Prinzipien

Auf die meisten der heute verwendeten Rechner treffen die folgenden Prinzipien des Rechnerentwurfs nach von Neumann zu:

- Ein Rechner besteht aus Rechenwerk, Steuerwerk, Speicher sowie Ein- und Ausgabegeräten.
- Die Zentraleinheit (bestehend aus Rechenwerk und Steuerwerk) arbeitet taktgesteuert.
- Die Signale werden binär codiert.

- Der Inhalt eines Speicherwortes wird über die Adresse des Speicherwortes angesprochen.
- Der Rechner verarbeitet externe Programme, die intern gespeichert werden.
- Programmbefehle und Daten werden im einheitlichen Hauptspeicher gespeichert.
- Programme und Daten werden sequentiell abgearbeitet. Der sequentielle Programmfluss kann dabei durch (bedingte und unbedingte) Sprünge verändert werden.
- Jede theoretisch mögliche Berechnung ist (im Rahmen der Kapazität des Rechners) berechenbar.

In den folgenden Abschnitten betrachten wir die Komponenten eines Rechners, der den genannten Prinzipien folgt, im Detail.

3.2.2 Bus

Ein Bus (bidirectional universal switch) ist eine Verbindungseinheit zwischen verschiedenen, logisch getrennten Funktionseinheiten. Abbildung 3.3 zeigt die Verbindung der Komponenten des Von-Neumann-Rechners über einen Adress- und einen Datenbus.

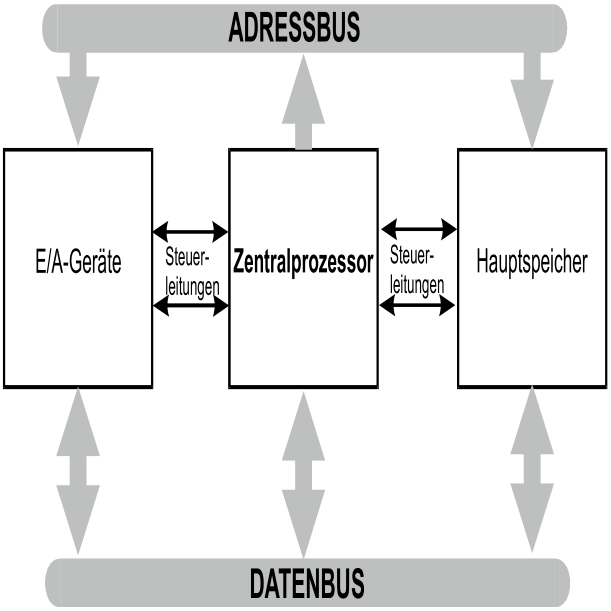


Abbildung 3.3: Von-Neumann-Architektur mit Daten- und Adressbus

3.2.3 Zentrale Verarbeitungseinheit

Die Zentrale Verarbeitungseinheit, auch Zentralprozessor oder CPU, ist das Kernstück jedes Von-Neumann-Rechners, in dem die eigentliche Informationsverarbeitung stattfindet. Die CPU besteht mindestens aus Rechenwerk und Steuerwerk und liegt häufig als integrierter Schaltkreis (Integrated Circuit (IC)), d. h. als elektronische Schaltung auf einem Chip, vor.

Bekannte Prozessorarchitekturen, die beispielsweise voneinander verschiedene Befehlssätze zur Programmierung aufweisen, sind u. a. :

- x86, Intel Architecture 32-Bit (IA-32), Intel-64
- m68k/m88k
- PowerPC
- Advanced RISC Machines (ARM)
- Microprocessor without Interlocked Pipeline Stages (MIPS)
- Scalable Processor ARChitecture (SPARC)

Während ARM-Prozessoren aufgrund ihres geringen Energiebedarfs beispielsweise in eingebetteten Systemen wie Smartphones und Tablets zum Einsatz kommen, sind Prozessoren mit x86-Architektur häufig in Desktop- und Server-Rechnern verbaut.
Neben der CPU können in einem Rechner weitere Prozessoren verbaut sein, die spezialisierte Aufgaben erfüllen. Diese zusätzlichen Prozessoren heißen Koprozessoren und können z. B. auf Gleitkommaarithmetik, Speichermanagement oder Buscontrolling spezialisiert sein.

Registersatz

In der CPU gibt es in der Regel ein oder mehrere Register Allzweckregister, die durch den Programmierer frei verwendbar sind. Des Weiteren weist eine CPU auch Spezialregister auf, die für bestimmte (u. U. CPU-interne) Aufgaben vorgesehen sind. Während Allzweckregister stets für den Programmierer sichtbar sind, wird bei den Spezialregistern zwischen sichtbaren und unsichtbaren Registern unterschieden. Abbildung 3.4 zeigt exemplarisch, welche Register für den Programmierer sichtbar und verwendbar sind und welche nicht.

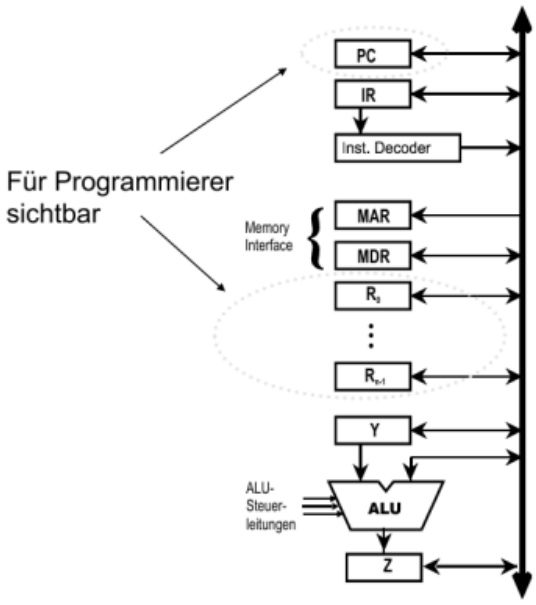


Abbildung 3.4: Sichtbarkeit von Registern

So ist beispielsweise das Register `PC` (program counter, auch instruction pointer bzw. Befehlszähler), welches die Adresse der Speicherzelle mit dem nächsten auszuführenden Befehl hält, für den Programmierer sichtbar. Der Programmierer kann den Inhalt dieses Spezialregisters anpassen, um den Programmablauf, z. B. durch Sprünge und Schleifen, zu steuern.
Das Spezialregister `IR` (instruction register bzw. Befehlsregister) ist für den Programmierer hingegen unsichtbar und dient CPU-intern dem Zweck, den gerade auszuführenden Befehl zu speichern. Auch die Register `MDR` (Memory Data Register) und `MAR` (Memory Adress Register) sind für den Programmierer unsichtbar. Sie dienen intern der Kommunikation mit dem Speicher, wobei das `MDR` die zu schreibenden bzw. aus dem Speicher gelesenen Daten hält, während das `MAR` die Adresse der Speicherzelle hält, die geschrieben oder gelesen werden soll.
Die Register R_0 bis R_{n-1} sind für den Programmierer sichtbare und frei verwendbare Allzweckregister.
Die Anzahl und die Einsetzbarkeit von Registern stellen ein wesentliches Merkmal einer Architektur dar. Die Gesamtheit der für den Programmierer nutzbaren Register wird Registersatz genannt.

Rechenwerk

Das Rechenwerk ist für die Ausführung verschiedener Operationen zuständig. Die Begriffe Rechenwerk und ALU sind nicht klar voneinander abzugrenzen. In einigen Quellen werden

die Begriffe synonym verwendet, während in anderen Quellen die ALU als das Kernstück des Rechenwerks betrachtet wird sowie selbst keine Register hat und ein reines Schaltnetz ist. Zu den Aufgaben des Rechenwerks gehört die Ausführung von:

- Arithmetischen Operationen (bspw. Addition, Subtraktion, Inkrementierung, Dekrementierung, Bildung von Zahlenkomplementen, etc.)
- Logischen Operationen (bspw. bitweise Negation, Disjunktion, Konjunktion, Antivalenz, etc.)
- Vergleichsoperationen zwischen Zahlen
- Schiebeoperationen (d. h. zyklisches und nicht-zyklisches Rechts- oder Linksschieben)

Bedingt durch die unterschiedlichen Operationstypen wird die ALU mitunter auch als Arithmetic/Logic/Shifting Unit (ALSU) bezeichnet.
Abbildung 3.5 zeigt eine einfache ALU.

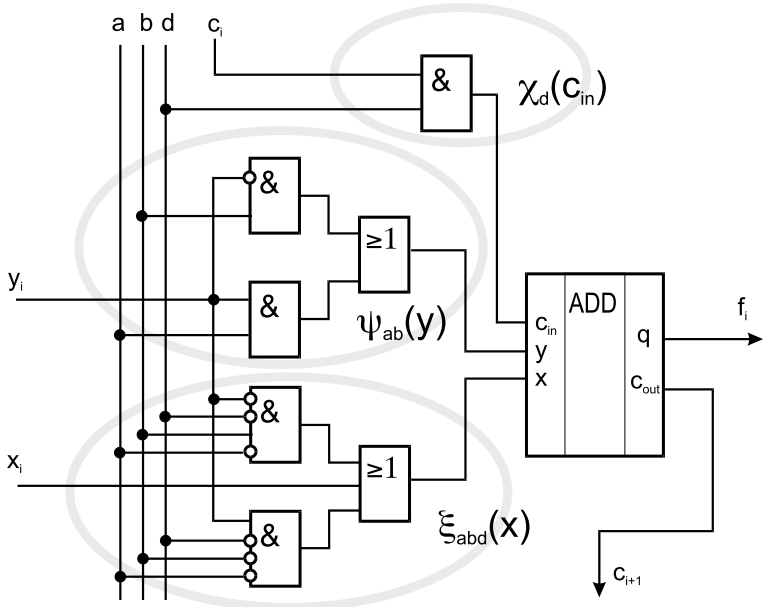


Abbildung 3.5: Einfache 1-Bit ALU

Die dargestellte ALU ist eine parametrisierte Schaltung, d. h. die auf den Operanden $\mathtt{x_i}$ und $\mathtt{y_i}$ auszuführende Operation kann mithilfe der Parameter `a`, `b` und `d` ausgewählt werden.
Wir betrachten in Abbildung 3.6 und Tabelle 3.1 zunächst die in Abbildung 3.5 mit $\psi_{ab}(y)$ bezeichnete Parameterfunktion zur Abbildung aller einstelligen Binärfunktionen für `y`.

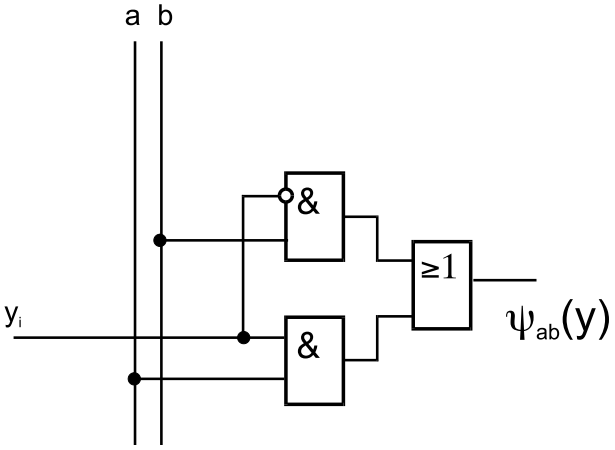


Abbildung 3.6: Schaltnetz für einstellige Binärfunktionen von `y`

Tabelle 3.1: Wahrheitswerttabelle $\psi_{ab}(y)$

a	b	$\psi_{ab}(y)$
0	0	0
0	1	$\overline{y}$
1	0	y
1	1	1

Tabelle 3.1 verdeutlicht, dass durch Kombination der Parameter `a` und `b` die auf `y` anzuwendende einstellige Binärfunktion aus der Menge aller möglichen einstelligen Binärfunktionen ausgewählt werden kann.
Abbildung 3.7 und Tabelle 3.2 beschreiben das parametrisierte Schaltnetz mit Ergebnisfunktion $\xi_{abd(x,y)}$ zur Abbildung zweistelliger Binärfunktionen von `x` und `y`. Auch dieses Schaltnetz ist Bestandteil der Gesamtschaltung in Abbildung 3.5 und trägt somit zur Vielfalt an verwendbaren Operationen bei.

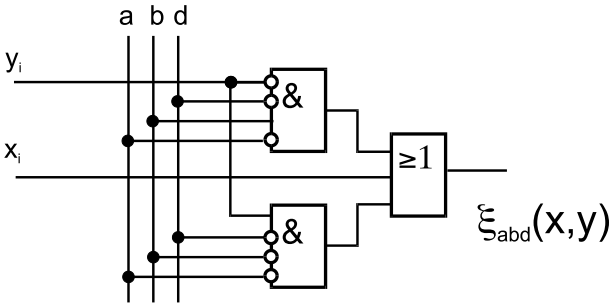


Abbildung 3.7: Schaltnetz für zweistellige Binärfunktionen von `x` und `y`

Tabelle 3.2: Wahrheitswerttabelle $\xi_{abd(x,y)}$

a	b	d	$\xi_{abd}(x,y)$
0	0	0	$x \vee y$
0	1	0	$x \vee \overline{y}$
1	0	0	x
1	1	0	x
-	-	1	x

Durch den $\vee$-Operator, der durch dieses Teilnetz der Operatorenmenge hinzugefügt wird, wird die ALU logisch vollständig (vgl. 1.4.2).
In Abbildung 3.8 und Tabelle 3.3 wird ersichtlich, dass mithilfe des Parameters `d` festgelegt werden kann, ob der Übertrag c_i in der Berechnung berücksichtigt wird.

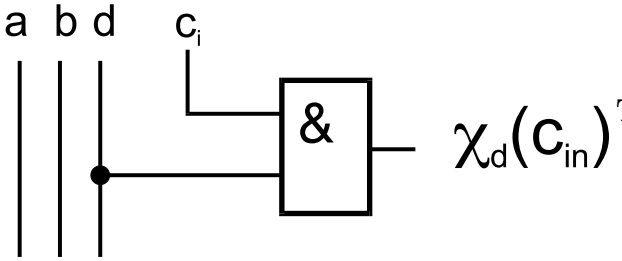


Abbildung 3.8: Schaltnetz für Umschaltung zwischen logischen und arithmetischen Operationen

Tabelle 3.3: Wahrheitswerttabelle $\chi_{abd(x,y)}$

a	b	d	$\xi_{abd}(x,y)$
0	0	0	$x \vee y$
0	1	0	$x \vee \overline{y}$
1	0	0	x
1	1	0	x
-	-	1	x

Ist `d=0`, so wird der Übertrag nicht berücksichtigt und aus arithmetischen Operationen werden logische (vgl. Tabelle 3.4), bei denen das Ergebnis einer Bitstelle nicht von anderen Bitstellen abhängt.
Tabelle 3.4 gibt zum einen einen Überblick über die gerade vorgestellten Parameterfunktionen und zeigt zum anderen, wie die Ergebniswerte dieser Funktionen mithilfe des Addierers in Abbildung 3.5 verknüpft werden müssen, um am Ausgang `f_i` das Gesamtergebnis der gewünschten Operationen zu erhalten.

Tabelle 3.4: Wahrheitswerttabelle ALU

a	b	c	d	$\psi_{ab}(y)$	$\chi_{abd}(x,y)$	Ergebnisfunktion $f_{abcd}(x,y)$	Bezeichnung
Logische Funktionen $f_{abcd}(x,y) = \psi_{a,b} \oplus \xi_{abd}(x,y)$							
0	0	-	0	0	$x \vee y$	$x \vee y$	Disjunktion
0	1	-	0	$\overline{y}$	$x \vee \overline{y}$	$x \wedge y$	Konjunktion
1	0	-	0	y	x	$x \oplus y$	Antivalenz
1	1	-	0	1	x	$\overline{x}$	Einerkomplement
Arithmetische Funktionen $f_{abcd}(x,1) = \psi_{ab}(y) + \xi_{abd}(x,y) + c$							
0	0	0	1	0	x	x	Identität (Transfer)
0	0	1	1	0	x	$x+1$	Inkrement
0	1	0	1	$\overline{y}$	x	$x-y$	Subtraktion (1er Kpl.)
0	1	1	1	$\overline{y}$	x	$x-y+1$	Subtraktion (2er Kpl.)
1	0	0	1	y	x	$x+y$	Addition
1	0	1	1	y	x	$x+y+1$	Addition mit Übtr.
1	1	0	1	1	x	$x-1$	Dekrement
1	1	1	1	1	x	x	Identität (Transfer)

Ähnlich der Zusammenschaltung von Volladdierern in Abbildung 2.20, können n ALUs zu einer n-Bit-ALU für die Verarbeitung von n-stelligen Operanden zusammen geschaltet werden.

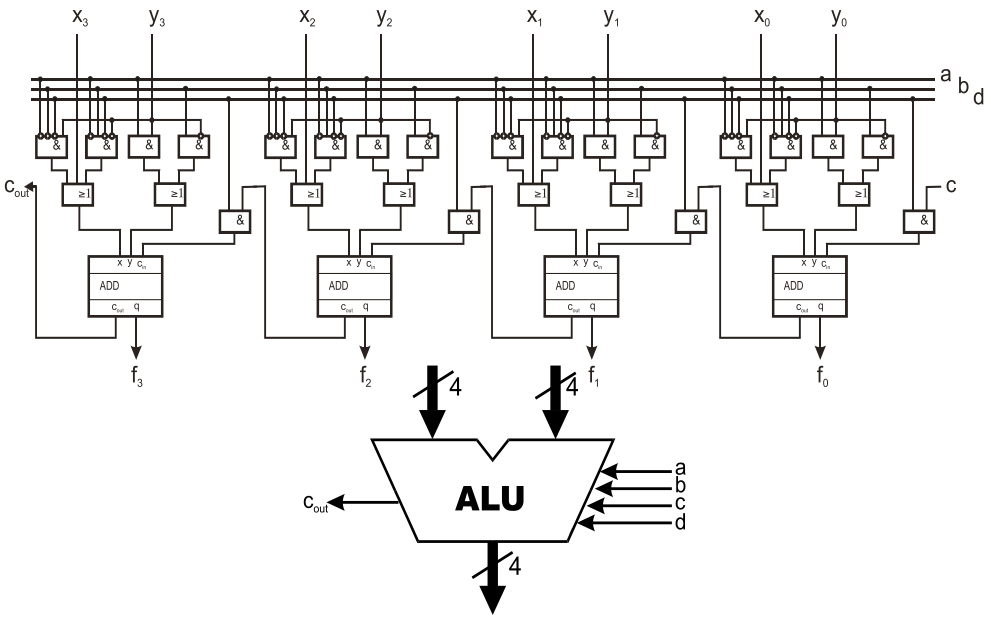


Abbildung 3.9: Einfache 4-Bit ALU

Abbildung zeigt eine auf diese Weise erzeugte, einfache 4-Bit-ALU und das zugehörige Blockschaltbild.
An diesem Punkt können wir mithilfe unserer einfachen ALU logische und arithmetische Operationen ausführen. Allerdings fehlt beispielsweise noch die Möglichkeit, Überläufe und Überträge zu berücksichtigen sowie Werte miteinander zu vergleichen. Deswegen betrachten wir nun in Abbildung 3.10 die Ergänzung der ALU um ein Statusregister (auch flag register), das im weiteren Verlauf auch der Realisierung von Vergleichsoperationen dient.

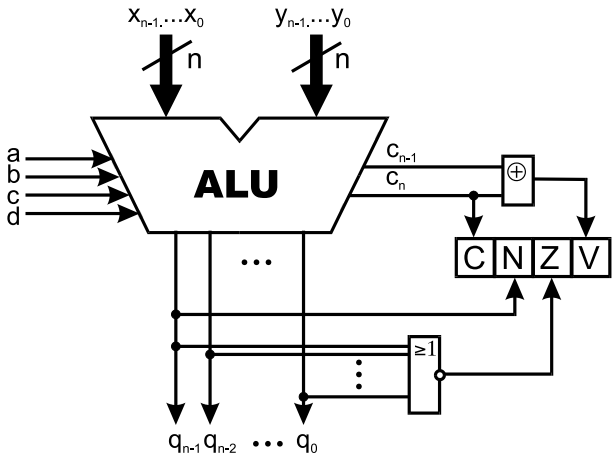


Abbildung 3.10: ALU mit Statusregister

Im Statusregister in Abbildung 3.10 werden die folgenden Flags (binäre Variablen, die als Statusindikatoren dienen) entsprechend dem Ergebnis der letzten Operation gesetzt:

- **Carry** ($C = c_n$): `C=1`, wenn eine arithmetische Operation einen Überlauf erzeugt hat.
- **Zero** ($Z = q_0 \vee ... \vee q_{n-1}$): `Z=1`, wenn ein Ergebnis ausschließlich aus Nullen besteht.
- **Negative** ($N = q_n$): `N=1`, wenn das Ergebnis eine negative Zahl ist.
- **Overflow** ($V = c_{n+1} \oplus c_n$): `V=1`, wenn eine Wertebereichsüberschreitung (Überlauf) bei einer Zweierkomplementrechnung aufgetreten ist.

Mithilfe der genannten Flags können wir nicht nur bestimmte Zustände der ALU detektieren, sondern auch Vergleichsoperationen ausführen. Die Auswertung von Vergleichen erfolgt durch Ausführung einer Subtraktion ohne Speichern des Rechenergebnisses. Das Ergebnis des Vergleichs ist anschließend, wie in Tabelle 3.5 zusammengefasst, an den Flags ablesbar.

Tabelle 3.5: Vergleichsoperationen

Relation	vorzeichenlose Zahlen	Zahlen im 2er-Komplement
$x = y$	Z=1	Z=1
$x \neq y$	Z=0	Z=0
$x \geq y$	C=1	N=V
$x < y$	C=0	N≠V
$x > y$	C=1 und Z=0	N=V und Z=0
$x \leq y$	C=0 oder Z=1	N≠V oder Z=1

Nachdem wir Aufbau und Funktionsweise des Kernstücks des Rechenwerks kennengelernt haben, fehlen uns zu einem einfachen, vollständigen Rechenwerk lediglich eine Komponente zum Ausführen von Schiebeoperationen, Register zum Ablegen der Operanden und einige Steuerleitungen. Abbildung 3.11 zeigt die Einbettung der ALU in das vollständige Rechenwerk.

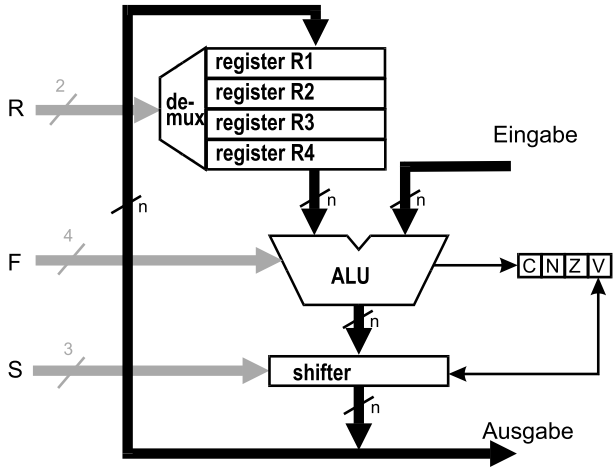


Abbildung 3.11: Vollständiges Rechenwerk

Über die schwarzen Bus erfolgt der Transport der Datensignale, d. h. der Transport der Informationen, die verarbeitet werden müssen. Der Transport der Steuersignale, die bestimmen, **wie** die Informationen verarbeitet werden, erfolgt über die grauen Busse. Dabei wird mithilfe des Steuersignals `R` das Register ausgewählt, dessen Inhalt als Operand dienen soll. Über das Steuersignal `F` wird festgelegt, welche Operation die ALU ausführen soll. Über das Steuersignal `S` kann die auszuführende Schiebeoperation gewählt werden.

Steuerwerk

Das Steuerwerk ist im Prinzip das „Nervenzentrum“ des Rechners. Es koordiniert das Zusammenspiel von Rechenwerk, Eingabegeräten, Ausgabegeräten und Speichersystem, indem es Steuersignale an die anderen Einheiten sendet sowie deren Status ergründet. Die beiden wesentlichen Funktionen des Steuerwerks sind das Sequencing, d. h. die Generierung von Steuersignalen zur Abarbeitung einer beliebigen, gegebenen Instruktion sowie die Steuerung und Überwachung von Speicher und Eingabe-/Ausgabe-System. Aufbau und Funktionsweise des Steuerwerks werden im Rahmen der Befehlsabarbeitung im nächsten Kapitel betrachtet.

3.2.4 Interaktion mit dem Speicher

Im Allgemeinen dient der Speicher zum Ablegen von Informationen, also Eingabedaten, Programmen und Ergebnissen. Für die Von-Neumann-Architektur ist besonders hervorzuheben, dass Programm und Daten in einem gemeinsamen Speicher liegen.
Was im Speicher liegt, ist aus Hardware-Sicht i. d. R. nicht zu erkennen, weswegen die Interpretation des Speicherinhalts der auf den Speicher zugreifenden Einheit obliegt. Für den Zugriff auf den Speicher ist u. a. von Bedeutung, dass es unterschiedliche Arten von Speicher gibt:

- Random-Access Memory (RAM): Speicher mit wahlfreiem Zugriff, d. h. Schreiben und Lesen sind möglich.
- Read-Only Memory (ROM): Speicher kann ausschließlich gelesen werden.

Unabhängig von der Art des Speichers muss zunächst ein Adresssignal erzeugt werden, um auf die entsprechende Speicherzelle des Speichers Zugriff zu erhalten. Außerdem wird ein Steuersignal benötigt, das besagt, ob die angegebene Adresse gelesen oder geschrieben werden soll.
Um mit dem Speicher kommunizieren zu können, weist die CPU mindestens die beiden für den Programmierer nicht sichtbaren Spezialregister Memory Address Register (MAR) und Memory Data Register (MDR) auf (vgl. Abbildung 3.4). Das MAR ist das Speicheradressierungsregister und hält die Adresse der Speicherzelle, auf die der lesende bzw.

schreibende Zugriff erfolgen soll. Das MDR speichert als Datenregister den Wert, der in den Speicher geschrieben werden soll bzw. der aus dem Speicher gelesen wurde. Wir greifen über diese beiden Register (anstatt direkt) auf den Speicher zu, weil Werte auf dem Bus u. U. länger anliegen müssen. Würden wir nicht den „Umweg“ über diese beiden Register gehen, würde das dazu führen, dass die CPU so lange blockiert wäre, bis die Speicheroperation abgeschlossen ist. Des Weiteren können Adressen aus komplexen Ausdrücken zusammengesetzt sein. In diesem Fall dient das MAR als Zwischenspeicher. Dazu kommt außerdem, dass Busse keine Werte speichern können, weswegen die Register auch als Busrepräsentation (Portal) dienen.
Die Verbindung der Register mit dem Speicher über entsprechende Steuerleitungen zeigt Abbildung 3.12.

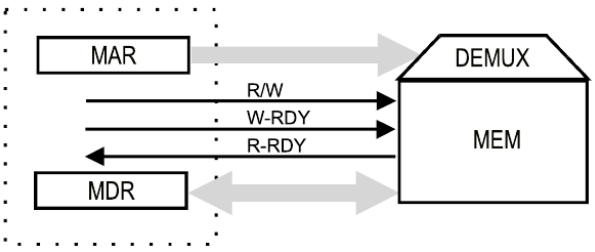


Abbildung 3.12: Vollständiges Rechenwerk

Ergänzend dazu beschreiben die nachfolgenden Flussdiagramme 3.13 und 3.14 den Ablauf des Lese- bzw. Schreibvorgangs.

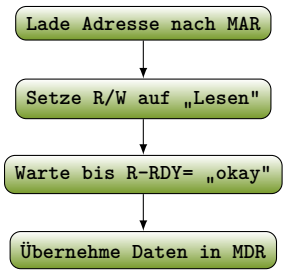


Abbildung 3.13: Lesezugriff

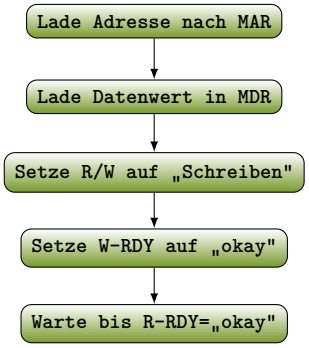


Abbildung 3.14: Schreibzugriff

3.2.5 Ein- und Ausgabe

Die Ein- und Ausgabe an Geräte funktioniert analog zum Lesen und Schreiben des Speichers. Entsprechend wird ein Gerät über eine Adresse angesprochen, wobei eine solche Adresse aus Sicht der CPU als Port bezeichnet wird. Ein auf einen Port geschriebener

oder von dort gelesener Wert wird, analog zum Prinzip des Speicherzugriffs, in bestimmten Registern zwischengespeichert. Da Ein-/Ausgabe und Speicherzugriff nach sehr ähnlichen Prinzipien ablaufen, wird in manchen Architekturen nicht zwischen Speicher und anderen Geräten unterschieden. Die Geräte weisen dort dann lediglich einen gesonderten Adressbereich auf.

3.3 Harvard-Architektur vs.Von-Neumann-Architektur

Neben der in den vorhergehenden Abschnitten sehr detailliert vorgestellten Von-Neumann-Architektur gibt es auch die in Abbildung 3.15 dargestellte Harvard-Architektur.

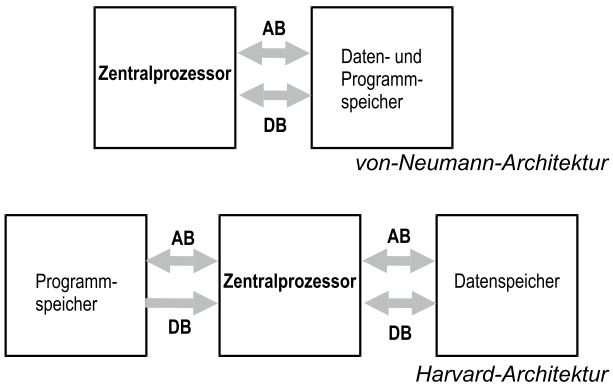


Abbildung 3.15: Harvard-Architektur

Der Unterschied zwischen beiden Architekturen besteht hauptsächlich darin, dass bei der Harvard-Architektur Programm und Daten in getrennten Speichern liegen, während sich Programm und Daten bei der Von-Neumann-Architektur in einem gemeinsamen Speicher befinden. Letzteres hat den unter dem Namen „Von-Neumann-Flaschenhals“ bekannten Nachteil, dass das Laden des nächsten Befehls und das Laden der Daten aus dem Speicher nicht parallel erfolgen können, weil Befehl und Daten über den selben Bus (und damit nacheinander) transportiert werden müssen. Bei der Harvard-Architektur können Befehl und Daten zwar parallel geladen werden, allerdings ist diese Architektur weniger flexibel, was das Mengenverhältnis zwischen Programmcode und Daten betrifft. Die Harvard-Architektur hat dann Vorteile, wenn man weiß, wie groß Programm- und Datenbereich sind, weil man die Größe der beiden Speichermodule so bereits hardwaretechnisch berücksichtigen kann. Letzteres ist für universelle Rechner allerdings nahezu unmöglich, weswegen die Harvard-Architektur meist in Spezialrechnern,z. B. für Steuerungsaufgaben oder Prozessdatenverwaltung, zum Einsatz kommt.

Zusammenfassung

Die drei Stufen der Informationsverarbeitung sind Dateneingabe, Verarbeitung und Datenausgabe. Diese drei Stufen spiegeln sich auch im Rechnerentwurf nach John von Neumann wider. Ein nach der Von-Neumann-Architektur konstruierter Rechner besteht dementsprechend aus Zentralprozessor mit Rechenwerk und Steuerwerk, einem gemeinsamen Speicher für Programm und Daten sowie Ein- und Ausgabegeräten. Die einzelnen Komponenten sind dabei über Daten- und Adressbus miteinander verbunden.

4 Von der Befehls- zur Programmausführung

Wir können nun prinzipiell arithmetische und logische Operationen ausführen und mit dem Speicher kommunizieren. In diesem Kapitel werden wir zunächst betrachten, was Befehle sind und anschließend analysieren, wie aus der Ausführung einzelner Befehle die Ausführung eines Programms wird. Wir beschäftigen uns also mit der Frage: „Was genau soll wann ausgeführt werden?“. Auf der Makroebene entspricht der Programmablauf schlicht der durch den Programmierer vorgegebenen Befehlsreihenfolge. Das Programm liegt hier in einer durch den Prozessor „lesbaren“ Form im Speicher vor. Nun stellt sich die Frage, welche Vorgänge auf Mikroebene ablaufen, damit der Prozessor das Programm „versteht“ bzw. die einzelnen Befehle korrekt interpretiert. Die Antwort auf diese Frage ist das durch das Steuerwerk ausgeführte „Sequencing“, welches in diesem Kapitel beschrieben wird.

4.1 Befehlssatz

Ein Befehl bzw. eine Instruktion ist eine Anweisung, die durch den Prozessor ausgeführt werden soll. Im Allgemeinen kann eine Operation bzw. ein Befehl als Funktion $z = op(x, y)$ aufgefasst werden, wobei x und y als Operanden bezeichnet werden und z als Ziel.
Die Menge an Befehlen, die ein Prozessor „versteht“ und damit ausführen kann, heißt Befehlssatz des Prozessors. Der Befehlssatz unterscheidet sich von Prozessor zu Prozessor. Die Befehle eines Befehlssatzes sind meist nach einem einheitlichen Schema aufgebaut und enthalten binär codiert mindestens den Op-Code (Name des Befehls), die Adressierungsart (vgl. Abschnitt 4.1.3) der Operanden und die Operanden selbst.

4.1.1 Typen von Instruktionen

Typischerweise enthält ein Befehlssatz folgende Arten von Instruktionen:

- Transfer-Operationen: Transfer von Daten zwischen Speicherorten (z.B. „Lade Registerinhalt in den Speicher!“, „Lege Wert auf Stack!“)

- arithmetische Operationen (z.B. „Inkrementiere Registerinhalt!“, „Addiere die obersten beiden Werte auf dem Stack!“)
- logische Operationen (z.B. „Bilde das Komplement des Akkumulatorinhalts!“, „Bilde die UND-Verknüpfung der obersten beiden Werte auf dem Stack!“)
- Operationen zur Programmsteuerung (z.B. „Springe falls `Z=1` nach Adresse `adr`!“, „Halte Programmausführung an!“, „Rufe Unterprogramm!“)
- Eingabe-/Ausgabe-Operationen (z.B. „Gib den Akkumulatorinhalt an Port `prt` aus!“)

4.1.2 Aufbau

Bei Unterscheidung von Prozessoren anhand der Anzahl der innerhalb von Befehlen verwendeten Adressen ergeben sich folgende Arten von Prozessoren:

- **Drei-Adress-Maschine**: Beide Operanden und das Ziel sind Bestandteil der Instruktion.
- **Zwei-Adress-Maschine**: Beide Operanden sind Bestandteil der Instruktion, ein Operand ist gleichzeitig das Ziel oder das Ziel ist implizit der Akkumulator.
- **Ein-Adress-Maschine**: Ein Operand in der Instruktion, zweiter Operand und Ziel in Akkumulator oder bestimmtem Register.

4.1.3 Adressierungsarten

Um Operanden innerhalb von Befehlen zu adressieren, gibt es die folgenden Möglichkeiten:

- **absolut/direkt**: Die Adresse des Operanden ist explizit als Teil der Instruktion gegeben.
- **unmittelbar**: Der Operand ist Teil der Instruktion.
- **indirekt**: Effektive Adresse des Operanden befindet sich in einem Register oder einer Hauptspeicherstelle, deren Adresse in der Instruktion angegeben wird.
- **indiziert**: Effektive Adresse des Operanden wird durch Addition einer Konstanten zu einem Registerinhalt generiert.
- **implizit**: Ein bestimmter Speicherort (z. B. ausgezeichnetes Register, Akkumulator, Stack) wird standardmäßig als Operand bzw. Ziel angenommen und somit nicht explizit im Befehl angegeben.

4.1.4 Entwurf von Befehlssätzen: RISC vs. CISC

Beim Entwurf von Befehlssätzen gibt es zwei (gegensätzliche) Entwurfsprinzipien: RISC und CISC. Tabelle 4.1 stellt die Eigenschaften beider Entwurfsprinzipien gegenüber. Beim Entwurf von Befehlssätzen für moderne Rechner wird versucht, einen guten Kompromiss zwischen RISC und CISC zu realisieren.

Tabelle 4.1: RISC vs. CISC

RISC	CISC
• Verzicht auf komplexe, bequeme Befehle • Konzentration auf wenige, dafür schneller ausführbare Befehle • Speicher wird meist nur bei Transferbefehlen benutzt, sonst Register • meist festverdrahtet implementiert	• Komplexe, mächtige und bequeme Befehle für Spezialzwecke • möglichst alle Adressierungsarten für alle Befehle • meist mittels Mikroprogramm implementiert

4.1.5 Maschinenbefehle

Wir wissen nun, dass Befehle Bitmuster sind, die der Prozessor „versteht“. Programme bestehen aus diesen Maschinenbefehlen und es entsteht die Illusion, dass der Prozessor diese Befehle unmittelbar ausführt.
Da es für Programmierer schwierig und aufwendig wäre, sich die Bitmuster der Maschinenbefehle zu merken, gibt es symbolische Abkürzungen für die Befehle. Diese menschenlesbaren Kürzel werden als „Mnemonics“ bezeichnet. Vorhandensein und Bedeutung der Menmonics sind von der Architektur abhängig. Einige typische Mnemonics, nach Befehlstyp sortiert, sind:

- **Transfer-Befehle**: `mov` (move), `ld` (load)
- **arithmetische Operationen**: `add`, `sub`, `mul`, `div`
- **logische Operationen**: `and`, `not`, `or`
- **Programmsteuerung**:

  `jmp` (unbedingter Sprung)
  `call` (Aufruf Unterprogramm)
  `jr` (unbedingter, relativer Sprung)
  `jrcc` (bedingter, relativer Sprung)
  `bra` (unbedingte Verzweigung)
  `bcc` (bedingte Verzweigung)

Damit aus den aufgelisteten Kürzeln vollständige Befehle werden, fehlen noch die entsprechenden Operanden. Die Reihenfolge von Quell- und Zieloperand ist von der verwendeten Assembler-Sprache und damit dem Prozessor abhängig:

- `mov ax,30H`: Kopiere den Wert 30_{HEX} ins Register `ax` (i80286-Assembler)
- `moveq #17, D0`: Kopiere den Wert 17 ins Register `D0` (MC68020-Assembler)

Auch die Adressierungsart ist vom Befehlssatz des Prozessors abhängig und wird durch das Mnemonic oder den Operanden angegeben. Während die Operanden im vorhergehenden Beispiel direkt adressiert wurden, ist u. a. auch indirekte Adressierung möglich. Indirekte Adressierung ist meist durch `()`, `[]` oder `@` gekennzeichnet:

- `mov ax,[bx]` (i80x86-Assembler)
- `ld a,(hl)` (Z80-Assembler)
- `moveb @R1,0x17` (PDP-11-Assembler)

Die Übersetzung der Mnemonics in Maschinencode übernehmen sog. Assembler-Programme. Heute werden Programme vielmals in Hochsprachen wie C, Pascal, Prolog oder Java (an Stelle von aus Mnemonics bestehender Assemblersprache) geschrieben. Programme zum Übersetzen von Hochsprachen, sog. Compiler, generieren heute oft direkt Maschinencode. Bei Compilern liegt im Gegensatz zu Assemblern keine 1-zu-1-Abbildung von (Hochsprach-)Befehl auf Maschinencode vor.

4.2 Abarbeitung von Befehlen

Ein Programm besteht aus im Speicher liegenden Bitmustern, die als Maschinenbefehle bezeichnet und nacheinander abgearbeitet werden. In diesem Abschnitt betrachten wir den Befehlszyklus, also den Ablauf zur Verarbeitung eines einzelnen Maschinenbefehls. Abbildung 4.1 zeigt den aus drei Schritten bestehenden Befehlszyklus.

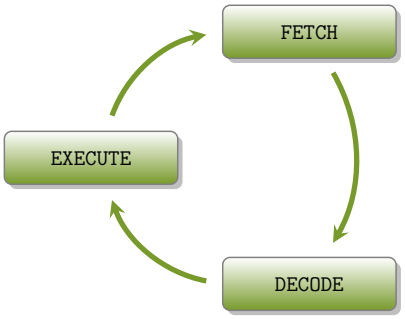


Abbildung 4.1: Befehlszyklus

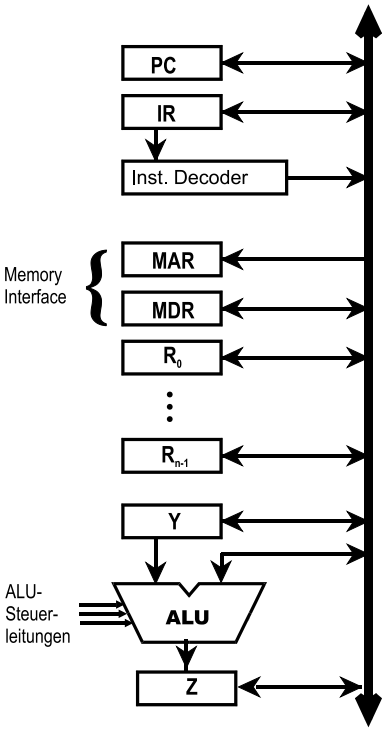


Abbildung 4.2: Registersatz

Für die Abarbeitung der einzelnen Phasen des Befehlszyklus wird der in Abschnitt 3.2.3 bereits vorgestellte und in Abbildung 4.2 dargestellte Registersatz benötigt. Die Phasen des Befehlszyklus im Detail:

1. **FETCH**: In dieser Phase wird der Befehl aus dem Hauptspeicher in den Prozessor geladen. Dazu wird die im Befehlszähler-Register `PC` enthaltene Adresse in das Speicheradressregister `MAR` geladen. Anschließend wird der an dieser Adresse befindliche Befehl aus dem Speicher in das Speicherdatenregister `MDR` gelesen und von dort aus in das Befehlsregister `IR` überführt.
2. **DECODE**: Hier wird der im `IR` befindliche Befehl dekodiert, d. h. es werden der Befehlscode, die Operanden und die Adressierungsart der Operanden ermittelt. Falls für die Operation Operanden aus dem Speicher geladen werden müssen, so geschieht das ebenfalls in dieser Phase.
3. **EXECUTE**: In dieser Phase wird die eigentliche Operation, bspw. mithilfe des Rechenwerks, ausgeführt. Falls die Operation dies verlangt, werden außerdem Ergebniswerte in den Speicher geschrieben.

Für die Ausführung des Befehlszyklus ist das Steuerwerk verantwortlich. Wie in Kapitel 3.2.3 versprochen, werden wir uns in diesem Abschnitt genauer mit dem Sequencing, also der Überführung des Befehlszyklus in eine Sequenz von Steuersignalen, beschäftigen. Die

Überführung des Befehlszyklus bzw. des einzelnen Befehls in eine Sequenz von Steuersignalen kann entweder durch ein fest verdrahtetes oder durch ein mikroprogrammierbares Steuerwerk geschehen. Hier betrachten wir Letzteres näher. Steuersignale sind CPU-interne Signale zur Steuerung der Komponenten. Über diese Signale wird beispielsweise die durch die ALU auszuführende Operation gewählt (Parametrisierung; vgl. Abschnitt 3.2.3). Außerdem werden so u. a. auch Registerein- und ausgaben gesteuert. Abbildung 4.3 zeigt die Komponenten des Steuerwerks.

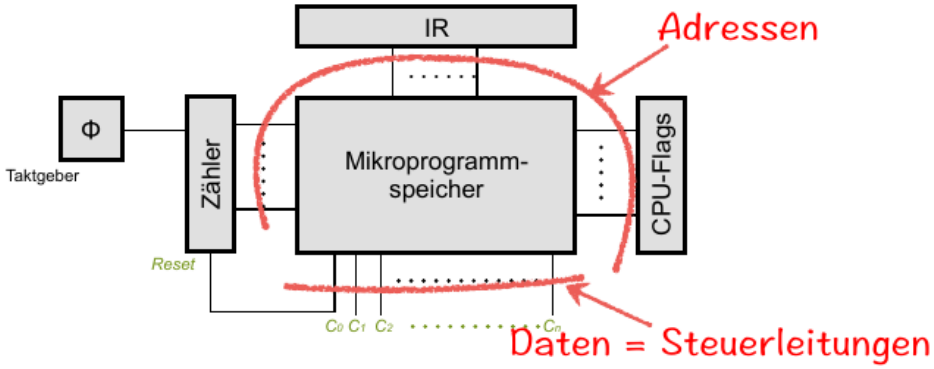


Abbildung 4.3: Aufbau Steuerwerk

Der Mikroprogrammspeicher enthält für jeden Makrobefehl (d. h. für jeden Maschinenbefehl) die notwendigen Kombinationen an Steuersignalen (sog. Mikrobefehle) $C_0 \ldots C_n$, die nacheinander für die Ausführung des Befehls realisiert werden müssen. Zum Abschluss der DECODE-Phase wird der (Befehls-)Zähler des Mikroprogrammspeichers auf den ersten Mikrobefehl, der für den in `IR` befindlichen Makrobefehl benötigt wird, gesetzt. In der EXECUTE-Phase wird dieser Zähler nun mit jedem Takt einen Mikrobefehl weiter gesetzt, bis die nötigen Mikrobefehle abgearbeitet sind und somit der Makrobefehl vollständig ausgeführt ist. Das nachfolgende Beispiel zeigt auf, welche Steuersignalkombinationen sequentiell realisiert werden müssen, um einen unbedingten, relativen Sprung auszuführen:

1. PC_{out} | MAR_{in} | read | clear Y | set carry-in to ALU | Add to Z
2. Z_{out} | PC_{in} | Wait for MFC
3. MDR_{out} | IR_{in}
4. PC_{out} | Y_{in}
5. $Adr(IR)_{out}$ | Add to Z
6. Z_{out} | PC_{in}

Für einen relativen Sprung sind bei unserer Beispiel-CPU also sechs nicht überlappende Zeiteinheiten (= Takte) notwendig.

4.3 Interrupts

Bis jetzt haben wir den regulären Befehlszyklus betrachtet, in welchem strikt sequentiell das im Speicher abgelegte Programm abgearbeitet wird. Allerdings können prinzipiell zu jedem Zeitpunkt der Programmausführung unvorhersehbare Ereignisse (bspw. Eingaben durch den Nutzer) eintreten, auf die möglichst schnell reagiert werden muss. Solche Ereignisse können auf verschiedene Art und Weise behandelt werden:

1. **Polling**: Im Programmcode wird ständig abgefragt, ob das Ereignis eingetreten ist. Vorteil dieser Variante ist, dass keine Hardware-Unterstützung notwendig ist, um Ergeigniss zu erkennen. Allerdings ist Polling sehr ineffizient, da es die CPU häufig in Anspruch nehmen muss, wenn ein Ereignis möglichst schnell detektiert werden soll.
2. **Interrupts**: Die Hardware erkennt ein Ereignis, sobald es eintritt. Im Gegensatz zum Polling sehr effizient, da die CPU nur in Anspruch genommen wird, wenn auch wirklich ein Ereignis eingetreten ist und darauf reagiert werden muss. Nachteil ist hier der größere Hardware-Aufwand.

Interrupts (d. h. Unterbrechungen der regulären Programmausführung) können auf Hardwareebene durch folgende Geräte angemeldet werden:

- Peripheriegeräte
- Koprozessoren
- die CPU selbst wegen Ausnahmebedingung (z.B. Fehler, wie Division durch Null)
- die CPU selbst wegen entsprechenden Befehls (Softwareinterrupt, auch Trap)

Wenn eine CPU Interrupt-Behandlung unterstützt, ändert sich der grundlegende Befehlszyklus zum Zyklus in Abbildung 4.4.

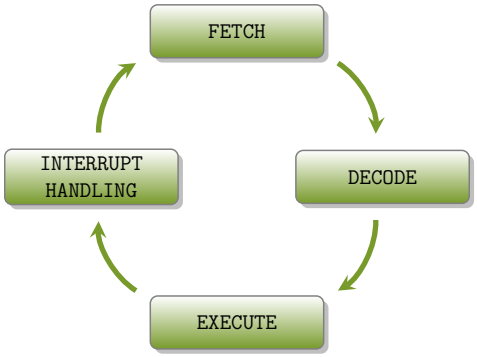


Abbildung 4.4: Befehlszyklus mit Interrupt-Behandlung

Die Behandlung eines Interrupts erfolgt durch ein gewöhnliches Programm. Um dieses Programm jedoch „während“ der regulären Ausführung eines anderen Programms ausführen

zu können, wird der Aufruf eines Unterprogramms nötig. Aus diesem Grund schauen wir uns hier kurz an, wie ein Unterprogrammaufruf i. d. R. funktioniert.
Viele Architekturen kennen neben dem frei adressierbaren Hauptspeicher einen Stack (auch Stapelspeicher). Ein Stack ist ein Speicher bei welchem, ähnlich wie bei einem Bücherstapel, ausschließlich das oberste Element entnommen werden kann. Ein Stack-Speicher bietet für den Zugriff mindestens die Befehle `push x` („Lege Element `x` auf den Stack!“) und `pop` („Entnimm das oberste Element vom Stack!“) an.
Beim Aufruf eines Unterprogramms wird der Stack genutzt, um Registerinhalte des aufrufenden Programms (beispielsweise den Befehlszähler) zu sichern und nach Ausführung des Unterprogramms anhand der gesicherten Werte den Programmablauf fortzusetzen. Abbildung 4.5 zeigt den schematischen Aufbau eines Programms im Speicher, welches das Unterprogramm „Aktivität Z“, dass sich ebenfalls einmal im Speicher befindet, mehrfach aufruft.

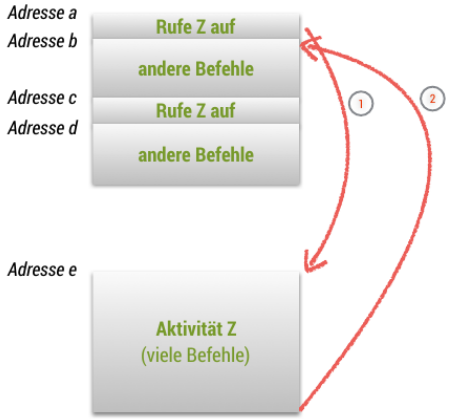


Abbildung 4.5: Schematische Darstellung eines Programms im Speicher mit Unterprogramm Z

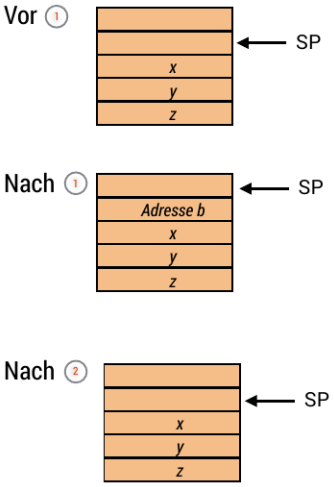


Abbildung 4.6: Entwicklung des Stacks bei Unterprogrammaufruf

Die Abbildung 4.6 zeigt passend dazu den Auf- und Abbau des Stacks. Für den Aufruf eines Unterprogramms steht i. d. R. der Befehl `call adr` zur Verfügung. Mittels `call adr` wird der aktuelle Befehlszähler (also der Inhalt des Registers `PC`) auf dem Stack gesichert und der Stack-Pointer `SP` inkrementiert. Anschließend wird das Register `PC` mit `adr` überschrieben und somit zur Startadresse des Unterprogramms gesprungen. Der Befehl `call adr` nutzt demnach den Befehl `push x`. In den Abbildungen entspricht dieses Vorgehen dem ersten Schritt. Nach Abarbeitung der Befehle des Unterprogramms kehren wir mittels `ret`, welches den Befehl `pop` nutzt, in das Hauptprogramm zurück. `ret` überführt den obersten Wert des Stacks in das Register `PC` und dekrementiert den Stack-Pointer. Der Befehlszeiger zeigt nun auf den nächsten auszuführenden Befehl des Hauptprogramms.

Wenn durch die Hardware ein Interrupt signalisiert wird, so kann dessen Behandlung durch einen Aufruf der Interrupt Service Routine (ISR) erfolgen, die im Prinzip einem Unterprogramm entspricht. Wenn also ein Interrupt auftritt, so retten wir den Befehlszähler auf den Stack, arbeiten die ISR ab und holen den Befehlszähler zurück in das Register `PC`. Beachtenswert ist dabei, dass ein Interrupt zu jedem beliebigen Zeitpunkt auftreten kann, weswegen die ISR nichts vom gerade ausgeführten Programm „wissen“ kann und das Programm nichts von der ISR.
Da Interrupts zu jedem beliebigen Zeitpunkt auftreten können, können sie eben auch dann auftreten, wenn gerade ein anderer Interrupt behandelt wird. Diese Problem kann beispielsweise durch die „Maskierung“ dieser Interrupts behoben werden, d. h. wir weisen den Interrupt ab und führen damit die ISR für diesen Interrupt nicht aus. Eine andere Variante ist die Verschachtelung von Interrupts, d. h. wir starten die entsprechende ISR des Interrupts auch dann, wenn wir uns gerade in der ISR eines zuvor aufgetretenen Interrupts befinden. Eine dritte Möglichkeit ist die Verzögerung von Interrupts, d. h. die ISRs werden in der Reihenfolge des Auftretens der Interrupts sequentiell abgearbeitet.

4.4 „Lebenslauf“ eines Programms

Wir wissen nun nicht nur, wie Programme und ihre Befehle aufgebaut sind, sondern auch, wie sie durch die CPU automatisch interpretiert werden. Um das Verständnis der automatischen Ausführung von Programmen weiter auszubauen, betrachten wir nun den Weg von der Programmerzeugung bis zur vollständigen Abarbeitung:

1. **Quellcode schreiben (beispielsweise in der Hochsprache C)**:

   Listing 4.1: Einfaches C-Programm

```c
#include <stdio.h>
#define ANTWORT 42 //ein Makro

int main (void)
{
    printf("Hallo, die Antwort ist: %d", ANTWORT); //in stdio.h deklariert
    return 0;
}
```

2. **Generieren einer ausführbaren Binärdatei**:
   Aus dem Quellcode wird nun in mehreren Schritten maschinenlesbarer Code erzeugt.

   a) **Präprozessor: ersetzt Makros, inkludiert Header-Files (z. B. stdio.h) und entfernt Kommentare.**
   Nach dem Lauf des Präprozessors (mittels `gcc -E -o hello.i hello.c`) über

unser Beispielprogramm `hello.c` erhalten wir folgenden (um den Inhalt der Bibliothek `stdio.h` gekürzten) C-Quellcode:

Listing 4.2: Beispielprogramm nach Präprozessorverarbeitung

```
1 int main (void)
2 {
3     printf("Hallo, die Antwort ist: %d", 42);
4     return 0;
5 }
```

b) **Compiler: übersetzt Quellcode in Assemblercode.**
Ausführung von `gcc -S -o hello.s hello.i` führt zu folgendem Assemblercode:

Listing 4.3: Assemblercode

```
1         .file   "hello.c"
2     .section    .rodata
3 .LC0:
4     .string "Hallo, die Antwort ist: %d"
5     .text
6     .globl  main
7     .type   main, @function
8 main:
9 .LFB0:
10    .cfi_startproc
11    pushq   %rbp
12    .cfi_def_cfa_offset 16
13    .cfi_offset 6, -16
14    movq    %rsp, %rbp
15    .cfi_def_cfa_register 6
16    movl    $42, %esi
17    movl    $.LC0, %edi
18    movl    $0, %eax
19    call    printf
20    movl    $0, %eax
21    popq    %rbp
22    .cfi_def_cfa 7, 8
23    ret
24    .cfi_endproc
25 .LFE0:
26    .size   main, .-main
27    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.1) 5.4.0 20160609"
28    .section    .note.GNU-stack,"",@progbits
```

c) **Assembler: übersetzt Assemblercode in Maschinensprache.** Ergebnis dieses Vorgangs ist eine nicht (ohne Hex-Editor) menschenlesbare Binärdatei bzw. Objektdatei `hello.o`. Diese kann beispielsweise mittels `gcc -c -o hello.o hello.s` erzeugt werden.

d) **Linker: ergänzt unaufgelöste Symbole (z. B. Variablen- und Funktionsnamen.**
Das Linken bzw. Binden kann statisch oder dynamisch erfolgen. Beim sta-

tischen Linken müssen alle Symbole zur Linkzeit bekannt sein, während die Symbole beim dynamischen Linken erst zur Laufzeit aufgelöst werden. Ergebnis des Linkvorgangs ist eine ausführbare Objektdatei.

3. **Laden des Programms** (bzw. der ausführbaren Binär-Datei) in den Hauptspeicher
4. **Ausführen des Programms** (wobei aus dem Programm ein Prozess wird)
5. **Beenden des Prozesses**

Die Details der letzten drei Schritte werden wir in Kapitel 5.2 näher betrachten. Hier haben wir die Vorgänge bei der Übersetzung eines Quellcodes in Maschinensprache sehr detailliert betrachtet und jeden Schritt einzeln ausgeführt. Im Normalfall würde man zur Übersetzung eines Programms mithilfe der **GCC!** (**GCC!**) alle Schritte direkt mittels `gcc -v hello.c -o hello` durchführen, um die ausführbare Objektdatei `hello` zu erhalten. Das übersetzte Programm kann nun mittels `./hello` gestartet werden, was dazu führt, dass das Programm in den Speicher geladen und ein entsprechender Prozess erzeugt wird.

Zusammenfassung

Ein Programm ist aus Sicht der CPU eine im Speicher liegende Abfolge von Maschinenbefehlen (Bitmuster). Diese Maschinenbefehle werden durch ein mikroprogrammierbares Steuerwerk in Mikrobefehle überführt und dem Befehlszyklus Fetch-Decode-Execute-Interrupt Handling entsprechend von der CPU interpretiert und ausgeführt. Der Lebenslauf eines solchen Programms beginnt mit dem Schreiben des Quellcodes. Der Quellcode wird anschließend durch den Präprozessor verarbeitet, compiliert, assembliert und gelinkt. Die so erzeugte Objektdatei kann in den Speicher geladen und ausgeführt werden.

5 Systemsoftware: Prozesse und Prozesswechsel

In diesem Kapitel betrachten wir, wie aus einem Programm (vgl. Kapitel 4) ein Prozess wird. Dafür beschäftigen wir uns zunächst mit dem groben Aufbau und den Aufgaben eines Betriebssystems, weil dieses das Laden des Programms in den Hauptspeicher, die Programmausführung und das Beenden des Prozesses ermöglicht. Anschließend wird näher auf das Prozessmanagement durch das Betriebssystem (BS) eingegangen.

5.1 Betriebssystem

Das Laden von Programmen in den Hauptspeicher ist ein Dienst, der zur Laufzeit (im Gegensatz zur Übersetzungszeit) die Programmausführung unterstützt. Neben dem Laden von Programmen gibt es weitere Dienste, die bspw. für Prozessmanagement, Speichermanagement, Dateimanagement und I/O-Management verantwortlich sind. Diese Dienste werden zu einem Betriebssystem (engl.: Operating System (OS)) zusammengefasst.
Im Allgemeinen hat das BS zwei Aufgaben. Zum einen bietet es dem Programmierer „schönere“ Interfaces und Abstraktionen als die reale Computerhardware:

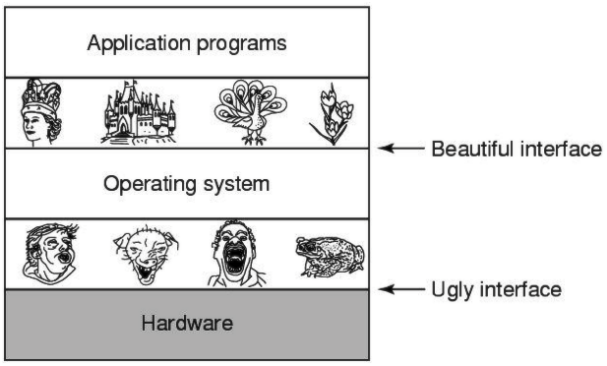


Abbildung 5.1: Betriebssystem vermittelt zwischen Anwendungsprogrammen und Hardware

Physisch betrachtet, liegen gespeicherte Inhalte auf der Festplatte nacheinander in Form von Datenblöcken fester Größe vor. Dank des Dateisystems, welches Bestandteil des BS

ist, sieht der Nutzer allerdings benannte Dateien variabler Größe. Zum anderen fungiert das BS als Ressourcenmanager, indem es den (konkurrierenden) Zugriff auf Ressourcen (z. B. Festplatte, CPU, Drucker) koordiniert. Pragmatisch zusammengefasst, umfasst ein BS also alle Softwarekomponenten, die die generische Ausführung von Anwendungen unterstützen und somit den Betrieb des Computersystems ermöglichen.

5.1.1 Arten von Betriebssystemen

Es werden u. a. folgende Arten von Betriebssystemen unterschieden:

- **Batch OS (Stapelbetriebssysteme)**: Verschiedene Programme (Jobs) werden nacheinander abgearbeitet und es findet meist keine Interaktion mit anderen Systemen bzw. Nutzern statt.
- **Multitasking OS (Mehrprogrammbetriebssysteme)**: Verschiedene Programme laufen nebenläufig, d. h. (scheinbar) gleichzeitig, ab. Außerdem sind Interaktionen möglich.
- **Multiuser OS (Mehrnutzerbetriebssysteme)**: Mehrere Nutzer können gleichzeitig interaktiv am System arbeiten (durch Time Sharing bzw. Zeitscheibenbetrieb).
- **Real-Time OS (Echtzeit- oder Prozesssteuerbetriebssysteme)**: Diese Betriebssysteme sind in der Lage, bestimmte zeitliche Anforderungen der Anwendungen zu erfüllen. Sie werden bspw. zu Steuerungszwecken in eingebetteten Systemen eingesetzt.

Heutige Betriebssysteme vereinigen meist mehrere Aspekte der genannten Betriebssytemarten.

5.1.2 Zweiteilung des Betriebssystems

Prozesse sind in Ausführung befindliche Programme, die u. U. untereinander interagieren können. Da Prozesse nicht physisch, d. h. nicht in Form von Hardware, vorhanden sind, müssen Prozesse und ihre Interaktionen durch eine Komponente zur Verfügung gestellt und verwaltet werden. Diese Komponente ist der „Kern“ bzw. „Kernel“ des Betriebssystems. Abbildung 5.2 zeigt die Grobgliederung des Betriebssystems in Kernbereich und Prozessbereich.

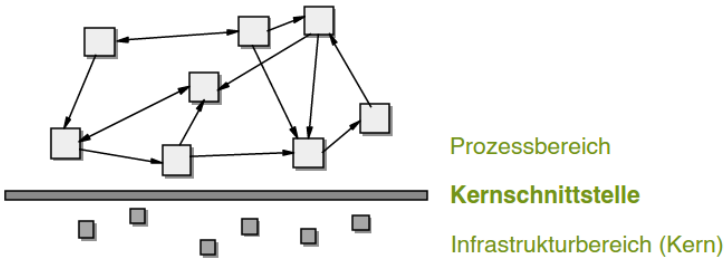


Abbildung 5.2: Zweiteilung des Betriebssystems

Während der Infrastrukturbereich bzw. Kern die grundlegende Infrastruktur für die Existenz von Prozessen bereitstellt, werden im Prozessbereich die eigentlichen, d. h. vom Anwender gewünschten, Funktionen erbracht.

5.2 Prozesse

Ein Prozess ist die Einheit der Ausführung (Aktivität), bei der ein Programm oder ein Teil eines Programms auf einem virtuellen Prozessor ausgeführt wird. Auf einem „virtuellen“ Prozessor deshalb, weil i. d. R. mehr Prozesse existieren, als reale Prozessoren zur Verfügung stehen. Der „virtuelle Prozessor“ wird implizit durch die Betriebssystem-Infrastruktur für den Prozess erzeugt. Jeder Prozess wird im Betriebssystem durch eine Datenstruktur, den Process Control Block (PCB), repräsentiert, um das Umschalten zwischen Prozessen zu ermöglichen. Der PCB enthält u. a. den Zustand des Prozesses (vgl. Abschnitt 5.2.2), Befehlszähler und Stackpointer.
Abbildung 5.3 veranschaulicht, warum wir überhaupt Prozesse und das Umschalten zwischen Prozessen benötigen.

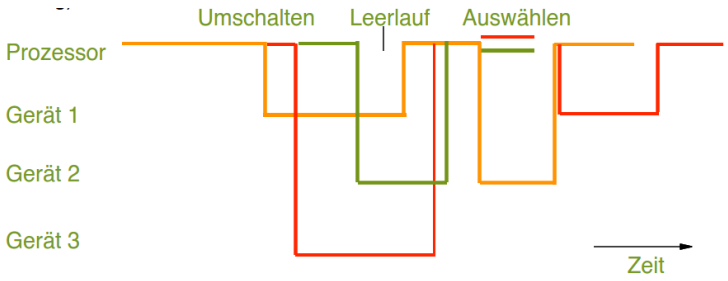


Abbildung 5.3: Ablaufdiagramm zur Nutzung der CPU durch mehrere Prozesse

Die Abbildung zeigt, dass Prozesse sehr häufig warten müssen, z. B. auf ein Gerät. Zur effizienten Nutzung der CPU ist es deswegen sinnvoll, in solchen Wartephasen, in denen der Prozess die CPU nicht nutzt, den Prozessor einem anderen Prozess zur Verfügung zu stellen. Dieser Prozesswechsel bedeutet einen Übergang von einer Befehlsfolge in eine

andere Befehlsfolge. Da der Prozessor aber nur eine Befehlsfolge abarbeiten kann, muss zwischen den Prozessen umgeschaltet werden. Das Umschalten geschieht u. a. durch das Ändern des Befehlszählers und wird in den folgenden Abschnitten diskutiert.

5.2.1 Prozesswechsel

Im einfachsten Fall kann das Umschalten direkt in die auszuführenden Programme „einprogrammiert“ werden, sodass an bestimmten Programmpunkten (z. B. während des Wartens auf ein Gerät) direkt zu einem anderen Prozess gesprungen werden kann. Anschließend soll der unterbrochene Prozess korrekt weitergeführt werden. Eine Umschaltstelle besteht demnach mindestens aus Sprungadresse (wo soll weitergemacht werden) und Fortsetzadresse (wo wurde die Arbeit unterbrochen). Für bestimmte Einsatzgebiete (z. B. Echtzeitsysteme) ist die zum Umschalten benötigte Zeit ein wichtiges Qualitätsmaß, weswegen das Umschalten effizient realisiert werden muss. Das soeben vorgestellte Umschalten durch direkten Sprung ist dabei die Minimallösung, die allerdings wenig flexibel und dadurch nur in speziellen, einfachen Fällen anwendbar ist.
Im Allgemeinen wird das Umschalten wesentlich aufwendiger sein, weil man bspw. häufig nicht weiß, von wo aus man wieder zum unterbrochenen Prozess zurückkehrt. Außerdem hält der Prozessor meist libnuma wesentliche Teile der Prozessbeschreibung (z. B. Registerinhalte), die durch das Umschalten zu einem anderen Prozess nicht verloren gehen dürfen. Dazu kommt, dass man nicht weiß, ob Prozesse die Sprungstelle überhaupt erreichen, sodass die CPU auch anderen Prozessen zur Verfügung steht. Außerdem ist der nächste Prozess, auf den man umschaltet, nicht immer der gleiche. Aus diesen Gründen führen wir den Prozesskontext und den darauf basierenden Kontextwechsel ein.

Prozesskontext

Außer dem Befehlszähler des gerade laufenden Prozesses enthält der Prozessor in seinen Registern eine Menge weiterer prozessspezifischer Daten. Dazu gehören bspw. die Inhalte von Rechen- und Indexregistern, die den Zustand der Programmabarbeitung repräsentieren sowie die Inhalte von Adressregistern, Segmenttabellen, Unterbrechungsmasken und Zugriffskontrollinformation, die die Ablaufumgebung des Prozesses darstellen. Die Gesamtheit aller durch den Prozessor gehaltenen Informationen wird als Prozesskontext bezeichnet. Dieser Prozesskontext muss im Rahmen des Umschaltens gerettet und beim Fortsetzen des Prozesses wieder hergestellt werden.

Kontextwechsel

Der Kontextwechsel ist der aufwendigste Teil des Umschaltens. Um ihn zu beschleunigen, kann von der Prozessorhardware Unterstützung angeboten werden, z. B. durch spezielle

Befehle, mit denen man einen kompletten Registersatz aus dem Prozessor in den Speicher schreiben kann (und umgekehrt) sowie durch die Bereitstellung mehrerer Registersätze auf dem Prozessor, sodass beim Umschalten u. U. nur das Register geändert werden muss, das die Nummer des gültigen Registersatzes angibt. Wenn nur die Rechenregister umgeladen werden müssen, also die Ablauf- bzw. Adressierungsumgebung dieselbe bleibt (Prozesswechsel innerhalb eines Adressraums durch Leichtgewichtsprozesse; vgl. Abschnitt 5.2.3), ist der Prozesswechsel relativ schnell möglich.

Automatisches Umschalten

Wie in Abschnitt 5.2.1 bereits erläutert, ist es in vielen Fällen nicht möglich, nicht sinnvoll oder zu aufwendig Umschaltstellen explizit in die Prozesse einzubauen. Insbesondere auch deshalb, weil sich dann jeder Programmierer bei der Erstellung von Programmen um das kooperative Umschalten kümmern müsste. Da wir i. d. R. davon ausgehen können, dass auf einem Rechner nicht nur ein einzelner Prozess läuft und entsprechend definitiv zwischen Prozessen umgeschaltet werden muss, ist es naheliegend, eine präemptive Vorgehensweise für das Umschalten zwischen Prozessen zu entwickeln. Das automatische Umschalten zwischen Prozessen wird mithilfe einer Intervalluhr (auch Timer bzw. Wecker) hardwareseitig realisiert. D. h. es gibt eine vorgegebene Frist, nach deren Ablauf der gerade laufende Prozess zugunsten eines wartenden Prozesses von der CPU verdrängt wird. Da das Umschalten von außen ausgelöst wird, können die Programme selbst unverändert bleiben – müssen sich also nicht kooperativ um das Umschalten kümmern. Allerdings kann das Umschalten jetzt auch zu jedem beliebigen Zeitpunkt innerhalb der Programmabarbeitung geschehen, was bei zeitkritischen Anwendungen bedacht werden muss.

Bedingtes Umschalten

Im Prozessablauf sind Situationen möglich, in denen vorübergehend nicht weitergearbeitet werden kann. Beispielsweise, wenn auf ein Gerät (z.B. die Festplatte, um Daten zu lesen/schreiben), auf die Freigabe einer Mutex-Variable oder auf den Eintritt in einen kritischen Codeabschnitt gewartet werden muss. In derartigen Situationen ist es sinnvoll, den Prozess (auch vor Ablauf der festgelegten Frist) vom Prozessor zu verdrängen. Denn in der Zeit, in der der wartende Prozess mit der Befehlsabarbeitung nicht vorankommt (busy wait), kann ein anderer Prozess den Prozessor verwenden. Die Vorgehensweise, dass nur dann zu einem Prozess umgeschaltet wird, wenn dieser gerade nicht auf das Eintreten einer Bedingung wartet, heißt bedingtes Umschalten. Abbildung 5.4 zeigt die Realisierung dieser Umschaltmethode mithilfe einer einfachen Binärvariable.

Abbildung 5.4: Bedingtes Umschalten realisiert durch die binäre Variable „Bedingungsbit“

5.2.2 Prozesszustände

Wenn ein Prozess auf eine Bedingung wartet, erfolgt beim bedingten Umschalten ein Umschalten auf einen anderen Prozess. Allerdings ist es möglich, dass auch der andere Prozess zu diesem Zeitpunkt nicht lauffähig ist, weil er bspw. auf eine E/A-Operation wartet. Das wiederum würde zu einem erneuten Umschaltversuch auf einen anderen Prozess führen. Dieses Vorgehen kann also zu einem länger andauernden Durchsuchen der Prozessmenge nach einem lauffähigen Prozess führen. Um die Suche nach einem fortsetzbaren Prozess zu beschleunigen, werden Prozesse anhand ihres Zustands zu Teilmengen zusammengefasst. Es ergeben sich die folgenden drei Prozesszustände:

- **Rechnend (Running)**: Der Prozess läuft gerade auf dem Prozessor.
- **Bereit (Ready)**: Der Prozess ist zwar lauffähig, läuft aber gerade nicht auf dem Prozessor (weil dieser durch einen anderen Prozess belegt ist).
- **Wartend (Waiting)**: Der Prozess ist nicht fortsetzbar, weil er auf das Eintreten einer Bedingung wartet.

Zustandsübergänge

Ein Prozess wechselt im Laufe seines Lebens zwischen den soeben beschriebenen Zuständen hin und her. Abbildung 5.5 zeigt die drei Prozesszustände und die entsprechenden Zustandsübergänge zwischen ihnen.

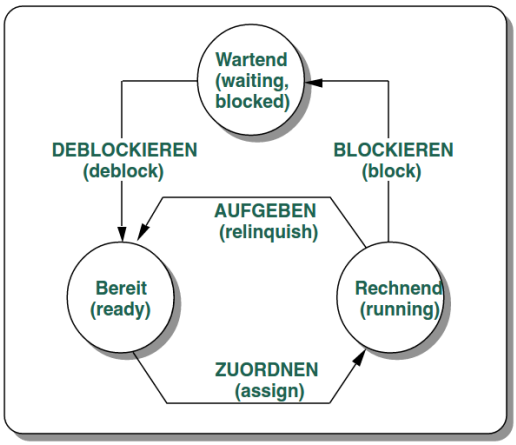


Abbildung 5.5: Prozesszustände und Zustandsübergänge

Für jeden der gezeigten Zustandsübergänge ist im Kern des Betriebssystems eine Zustandswechseloperation vorgesehen:

- **Aufgeben (Relinquish)**: Freiwilliges Umschalten auf einen anderen Prozess. Der bisher rechnende Prozess bleibt jedoch fortsetzbar, d. h. geht in den Zustand „bereit“ über.
- **Zuordnen (Assign)**: Nächsten Prozesses aus der „Bereit“-Menge auf dem Prozessor fortsetzen.
- **Blockieren (Block)**: Verlassen des Prozessors wegen Nichterfüllung einer Bedingung (bedingtes Umschalten). Da für die Fortsetzung die Bedingung erfüllt sein muss, geht der Prozess in den Zustand „wartend“ über.
- **Deblockieren (Deblock)**: Ist die Bedingung eingetreten, auf die der blockierte Prozess gewartet hat, so wird er wieder in die Menge der bereiten Prozesse eingefügt.

Dynamische Systeme

Bisher sind wir davon ausgegangen, dass im System eine feste Anzahl von Prozessen vorhanden ist. In diesem Abschnitt betrachten wir nun jedoch dynamische Systeme, d. h. Systeme, in denen die Anzahl der Prozesse variabel ist.
Dynamische Systeme können in zwei Schritten entstehen. Der erste Schritt entspricht dabei dem Hinzufügen des Zustands „Nicht aktiv“ zur Menge der möglichen Prozesszustände. Ein Prozess befindet sich im Zustand „Nicht aktiv“, wenn der Prozess zwar definiert ist (d. h. ein PCB existiert), aber der Prozess ruht. Im Zuge dieses Schritts werden die Zustandsübergänge „Aktivieren“ und „Deaktivieren“ zur Menge der Zustandsübergänge hinzugefügt. Im zweiten Schritt wird der Prozesszustand „Nicht existent“ zur Zustandsmenge hinzugefügt. Damit verbunden, ergeben sich die Zustandswechseloperationen Erzeugen und Löschen und damit die Möglichkeit einer veränderlichen Anzahl an Prozessen

im System. Diesem Schritt leigt die Annhame zugrunde, dass Prozesse bei Systemstart noch nicht existieren, sondern erst explizit erzeugt (und auch gelöscht) werden.
Abbildung 5.6 zeigt das um die neuen Zustände und Zustandsübergänge ergänzte Zustandsdiagramm.

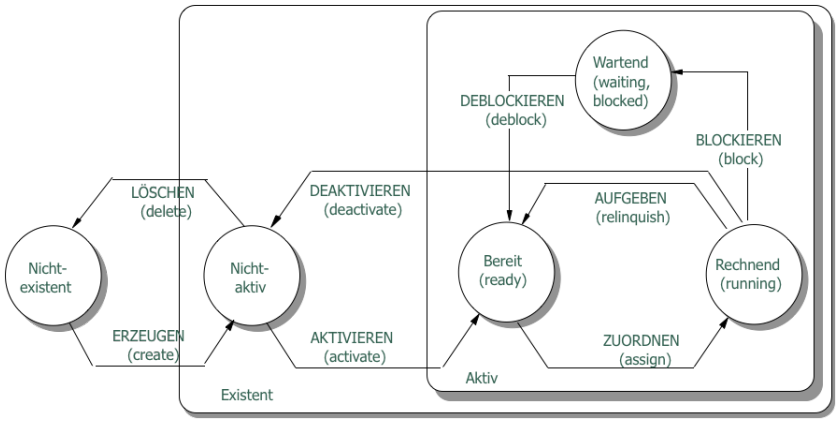


Abbildung 5.6: Vollständiges Zustandsdiagramm

Leerlaufproblem

Das Leerlaufproblem beschreibt die Möglichkeit, dass alle im System vorhanden Prozesse gerade auf den Eintritt einer Bedingung warten und damit nicht auf dem Prozessor ausgeführt werden können. Es stellt sich also die Frage, was der Prozessor in dieser Zeit tut, in der gerade kein Programm abgearbeitet werden kann. Da ein Anhalten des Prozessors ungünstig ist, weil jederzeit ein wartender Prozess fortsetzbar werden kann, ist eine elegante Lösung für das Leerlaufproblem der sogenannte Leerlaufprozess (idle process). Dieser Prozess ist nie blockierend, nie beendet (zyklischer Prozess) und muss jederzeit verlassen werden können. Die leere Schleife `while (true) {}` entspricht einer möglichen Implementation dieses Verhaltens. Die Ausführung des Leerlaufprozesses sollte jederzeit dynamisch gestoppt werden können. Dies kann bspw. durch einen Spezialbefehl (bspw. für x68 `HLT`) geschehen, der keinen Speicherzugriff durchführt und auf externe Signale reagiert.

5.2.3 Leichtgewichts- vs. Schwergewichtsprozesse

Wie in Abschnitt 5.2 beschrieben, ist ein Prozess ein in Ausführung befindliches Programm. Der so definierte, klassische Prozess entspricht dem hier vorgestellten Schwergewichtsprozess. Zu jedem Schwergewichtsprozess gehört eine Menge von Ressourcen, wie der Adressraum des Prozesses oder Dateideskriptoren (file-handles). Im Gegensatz dazu existieren auch Leichtgewichtsprozesse, sogenannte Threads. Im Falle der Leichtgewichtsprozesse teilen sich mehrere dieser Threads Ressourcen wie den Adressraum. Beim

Multithreading gibt es demnach keinen gegenseitigen Schutz, d. h. ein Thread kann auf die Daten der anderen Threads in seinem Adressraum zugreifen. Wie die Gegenüberstellung von Schwergewichts- und Leichtgewichtsprozess in Abbildung 5.7 zeigt, hat innerhalb des gemeinsamen Adressraums allerdings jeder Thread seinen eigenen Stack, um seine Registerinhalte und seinen Programm Counter zu sichern.

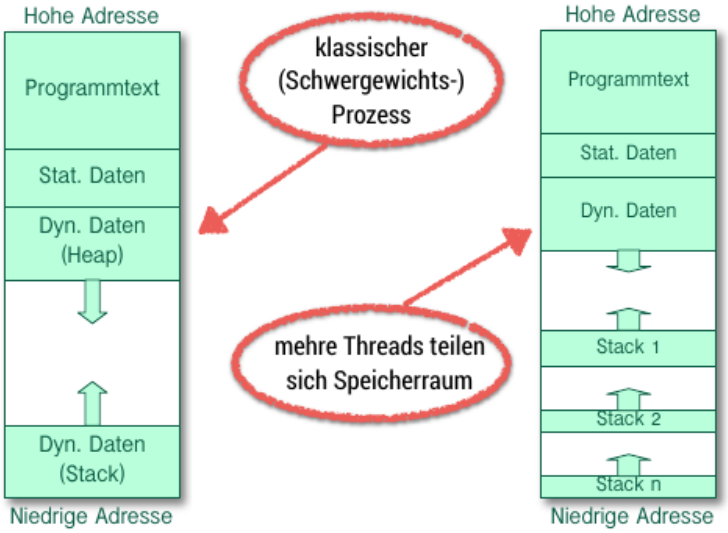


Abbildung 5.7: Aufteilung des Adressraums von Schwergewichts- und Leichtgewichtsprozess im Vergleich

Zusammenfassung

Das Betriebssystem umfasst alle Dienste, die für die generische Ausführung von Anwendungen notwendig sind. Neben Speicher-, I/O- und Dateimanagement gehört dazu auch das Prozessmanagement, welches wir in diesem Kapitel näher betrachtet haben. Ein Prozess ist dabei ein Programm in Ausführung. In einem dynamischen System existiert eine variable Anzahl an Prozessen, wobei diese Prozesse sich die Ressource Prozessor teilen müssen. Aus diesem Grund müssen Kontextwechsel stattfinden, weswegen sich ein Prozess zu jedem Zeitpunkt in einem der Zustände „Nicht existent“, „Nicht aktiv“, „Bereit“, „Wartend“ oder „Rechnend“ befindet. Abschließend haben wir die Unterteilung von Prozessen in Schwergewichts- und Leichtgewichtsprozesse kennengelernt. Während ein Schwergewichtsprozess (= klassischer Prozess) einen eigenen Adressraum besitzt, teilen sich mehrere Leichtgewichtsprozesse einen Adressraum und besitzen innerhalb dieses Adressraums lediglich einen eigenen Stack.

6 Ressource Prozessor: Scheduling

Wie in Kapitel 5.2 beschrieben, wird ein anderer, fortsetzungsfähiger Prozess ausgewählt, sobald ein Prozess den Prozessor aufgibt (d. h. den Zustand „laufend“ verlässt). Daraus ergibt sich die Fragestellung, welcher Prozess ausgewählt wird, wenn mehrere Prozesse gleichzeitig fortsetzungsfähig sind. Abbildung 6.1 veranschaulicht diese Fragestellung.

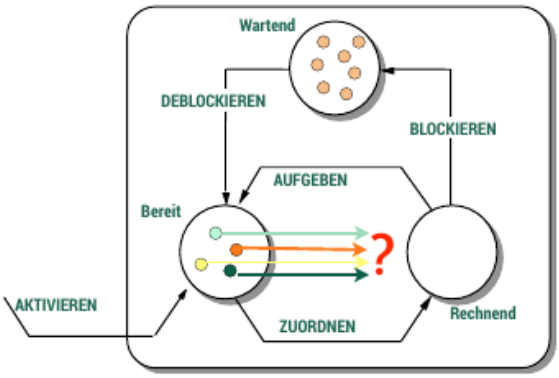


Abbildung 6.1: Mehrere Prozesse im Zustand „bereit“

Allgemeiner formuliert entspricht diese Fragestellung dem Problem der zeitlichen Zuordnung von Aktivitäten zu Ressourcen. Als Teilgebiet der Mathematik, welches älter als die Informatik ist, setzt sich die Schedulingtheorie („Scheduling“ bedeutet in etwa „Ablaufplanung“) mit der Beschreibung und Lösung dieses Problems auseinander.

6.1 Ziel

Abbildung 6.2 zeigt ein klassisches Schedulingproblem.

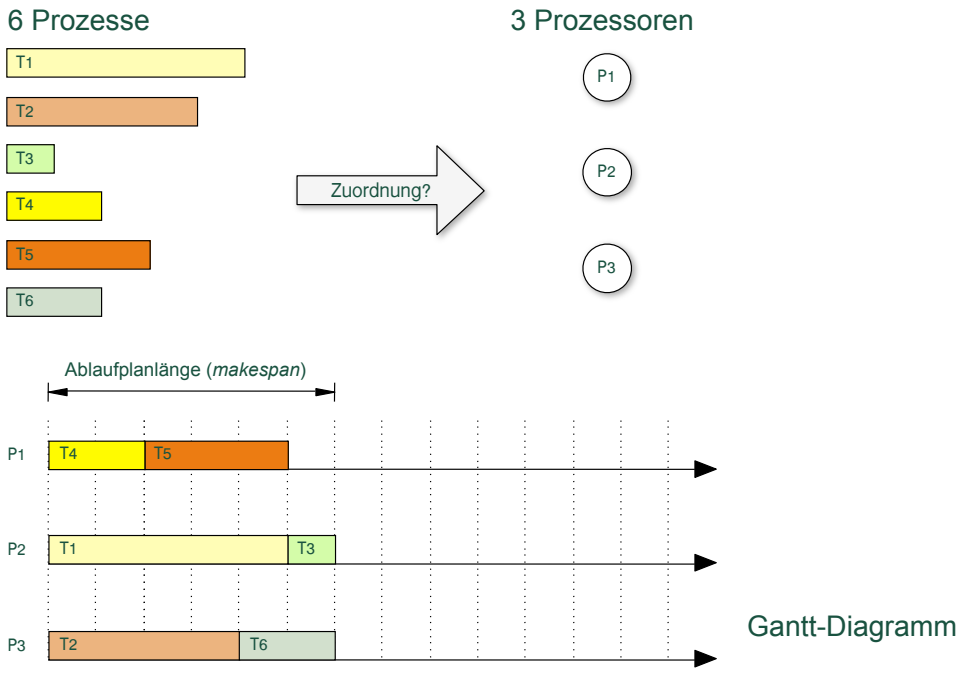


Abbildung 6.2: Klassisches Scheduling-Problem

Die sechs Prozesse T1 bis T6 sollen dabei den Prozessoren P1 bis P3 zugeordnet werden. Mit welchem Schedulingverfahren die Prozesse den Prozessoren zugeordnet werden, hängt vom Schedulingziel ab. Wie in Tabelle 6.1 aufgeführt, wird zwischen benutzer- und systemorientierten Zielen unterschieden.

Benutzerorientierte Ziele	Systemorientierte Ziele
Länge des Ablaufplans minimieren	Anzahl benötigter Prozessoren minimieren
Antwortzeit minimieren bzw. Frist einhalten	Durchsatz maximieren
Durchschnittliche Antwortzeit minimieren	Prozessorauslastung maximieren
Maximale Verspätung bzgl. einer Frist minimieren	
Anzahl verspäteter Prozesse minimieren	

Tabelle 6.1: Schedulingziele

Zu beachten ist, dass einige dieser Ziele miteinander in Konflikt stehen und somit nicht gleichzeitig erreicht werden können. Beispielsweise widerspricht das Minimieren der maximalen Verspätung dem gleichzeitigen Minimieren der Anzahl verspäteter Prozesse.

6.2 Standardstrategien

In den folgenden Abschnitten betrachten wir verschiedene Schedulingverfahren, um einen Prozess aus der „Bereit“-Menge entsprechend einem Schedulingziel auszuwählen und einem Prozessor zuzuordnen. Zur Illustration der vorgestellten Strategien verwenden wir die in Tabelle 6.2 gezeigte, beispielhafte Zusammenstellung von Prozessen.

Nr.	**Ankunft** a	**Bedienzeit** t_e	**Priorität** P
1	0	3	2
2	2	6	4
3	4	4	1
4	6	5	5
5	8	2	3

Tabelle 6.2: Scheduling-Beispiel

Jedem Prozess wird in Spalte „Nr.“ eine eindeutige ID zugewiesen. Die Ankunftszeit a gibt an, wann der Prozess in die Menge der zu planenden Prozesse eintritt. Die Bedienzeit t_e entspricht der Anzahl an Zeiteinheiten, die der Prozess im Zustand „rechnend“ verbringen muss (d. h. in Besitz des Prozessors sein muss) bis der Programmcode vollständig abgearbeitet ist. Des Weiteren können die Prozesse anhand der zugewiesenen Prioritäten nach Dringlichkeit sortiert werden. Dabei entspricht 1 der niedrigsten und 5 der höchsten Priorität. Bei Schedulingverfahren mit Beachtung der Priorität (vgl. 6.2.5) hat Prozess 4 demnach Vorrang vor Prozess 3.

6.2.1 First Come First Served (FCFS)

Bei Anwendung des FCFS-Verfahrens werden die Prozesse in der Reihenfolge abgearbeitet, in der sie in der Bereitliste ankommen. Wurde ein Prozess für die Ausführung auf dem Prozessor ausgewählt, so ist der Prozess solange in Prozessorbesitz, bis das Programm vollständig abgearbeitet ist oder der Prozess freiwillig aufgibt. D.h. ein Prozess wird nicht durch neu ankommende Prozesse vom Prozessor verdrängt. Das Verfahren ähnelt demnach der Vorgehensweise an einer Kasse im Supermarkt: Der Kunde (=Prozess), der sich zuerst an der Kasse (=Prozessor) anstellt, wird auch zuerst bedient. Zu beachten ist allerdings, dass Kunden mit wenigen Artikeln (= kurzer Bedienzeit) hier nicht der Vortritt gelassen wird.

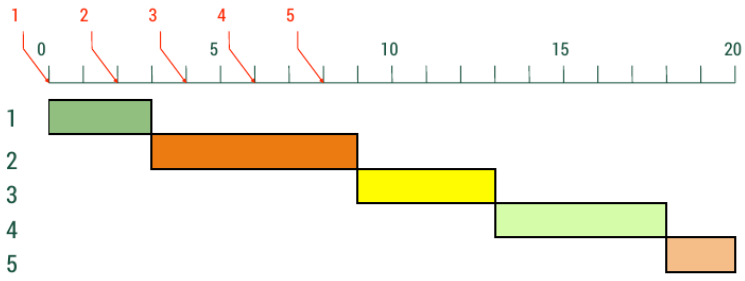


Abbildung 6.3: First Come First Served

Abbildung 6.3 zeigt den FCFS-Schedulingplan für die Prozesse aus 6.2. Links außen sind die Prozessnummern zu finden. Die roten Pfeile markieren die Ankunft des entsprechenden

Prozesses in der Bereitliste. Ist auf der Zeile eines Prozesses ein Balken zu finden, so bedeutet dies, dass der Prozess im zugehörigen Zeitabschnitt in Prozessorbesitz ist.

6.2.2 Last Come First Served (LCFS)

Das LCFS-Verfahren ist das Gegenstück zum FCFS-Verfahren. Hier werden die Prozesse in der umgekehrten Reihenfolge ihrer Ankunft in der Bereitliste abgearbeitet. Auch hier bleibt der Prozess solange in Prozessorbesitz, bis er freiwillig aufgibt oder das Programm vollständig abgearbeitet ist. Zudem kann der laufende Prozess nicht von einem anderen Prozess verdrängt werden.

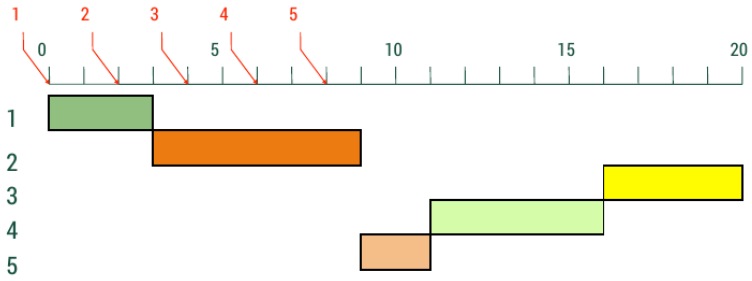


Abbildung 6.4: Last Come First Served

Abbildung 6.4 zeigt den zugehörigen Schedulingplan für das Prozessensemble in Tabelle 6.2. Zum Zeitpunkt $t = 0$ ist nur Prozess 1 in der Bereitliste zu finden, womit er der Prozess ist, der die Bereitliste als letztes erreicht hat. Während Prozess 1 abgearbeitet wird, kommt Prozess 2 in der Bereitliste an. Zum Zeitpunkt 3, an welchem Prozess 1 den Prozessor aufgibt, befindet sich ausschließlich Prozess 2 in der Bereitliste. Entsprechend geht Prozess 2 in den Zustand „laufend“ über. Während Prozess 2 läuft, erreichen die Prozesse 4 und 5 nacheinander die Bereitliste. Da die Prozesse nach LCFS in umgekehrter Reihenfolge abgearbeitet werden, wird nach der Abarbeitung von Prozess 3 zunächst Prozess 5 und danach Prozess 4 abgearbeitet. LCFS findet in dieser reinen Form eher selten Anwendung, weswegen wir im folgenden Abschnitt eine Erweiterung dieses Verfahrens betrachten.

6.2.3 Last Come First Served – Preemptive Resume (LCFS-PR)

Das LCFS-PR Verfahren ist ein um das Verdrängungskonzept ergänztes LCFS-Verfahren. Kommt ein Prozess neu in der Bereitlist an, so verdrängt er den gerade laufenden Prozess vom Prozessor. Der verdrängte Prozess wird hinter dem verdrängenden Prozess in die Bereitliste eingeordnet. Falls in der Zeit, in der ein Prozess läuft, keine anderen Prozess ankommen, so wird die Bereitliste wie im Fall von LCFS ohne Verdrängung in umgekehrter Ankunftsreihenfolge abgearbeitet. Ziel dieser Vorgehensweise ist die Bevorzugung kurzer Prozesse, weil der kurze Prozesse die Chance haben, noch vor der nächsten Ankunft mit

der Abarbeitung fertig zu werden. Lange Prozesse können hingegen mehrfach verdrängt werden.

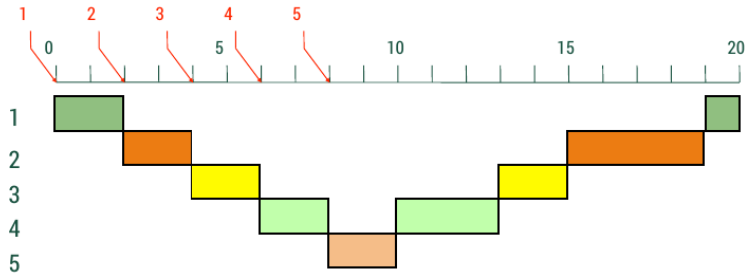


Abbildung 6.5: Last Come First Served – Preemptive Resume

Betrachten wir dazu den Schedulingplan in Abbildung 6.5. Prozess 1 hat eine Bedienzeit von 3 Zeiteinheiten. Innerhalb dieser 3 Zeiteinheiten erreicht Prozess 2 die Bereitliste, d. h. Prozess 2 verdrängt Prozess 1 vom Prozessor. Während Prozess 2 rechnet, erreicht Prozess 3 die Bereitliste und verdrängt wiederum Prozess 2. Durch die Ankunft von Prozess 4 wird anschließend Prozess 3 verdrängt und Prozess 4 muss selbst Prozess 5 weichen. Da während Prozess 5 läuft keine weiteren Prozesse ankommen, wird dieser fertig abgearbeitet. Anschließend wird Prozess 4 abgearbeitet, weil dieser zuvor durch Prozess 5 verdrängt wurde und somit hinter Prozess 5 in die Bereitliste eingeordnet wurde. Dieses Vorgehen gilt analog für die Abarbeitung der Prozesse 3, 2 und 1 in ebendieser Reihenfolge.

6.2.4 Round Robin (RR)

Die Prozesse werden entsprechend ihrer Ankunftsreihenfolge in die Bereitliste eingeordnet und abgearbeitet (vgl. 6.2.1. Nach Ablauf einer vorher festgesetzten Frist τ (Zeitscheibe, time slice, CPU-Quantum) wird der rechnende Prozess allerdings verdrängt und an das Ende der Bereitliste angefügt, sodass auf den nächsten Prozess umgeschaltet werden kann. Ziel des Verfahrens ist die gleichmäßige Verteilung der Prozessorkapazität und der Wartezeit auf die Prozesse unabhängig von Bedienzeit und Priorität.
Die Wahl der Zeitscheibenlänge τ ist Optimierungsproblem. Mit wachsendem τ nähert sich diese Strategie der Reihenfolgestrategie FCFS, weil sich mit wachsendem τ die Chance erhöht, dass Prozesse innerhalb einer Zeitscheibe fertig werden – womit die Prozesse im Extremfall schlicht entsprechend ihrer Ankunftsreihenfolge abgearbeitet werden. Mit fallendem τ steigt wiederum die Häufigkeit des Umschaltens, welches ebenso Zeit in Anspruch nimmt. Im Extremfall wäre dann die Zeit für das Umschalten auf einen Prozess länger, als die Zeit, die der Prozess auf dem Prozessor zubringt.

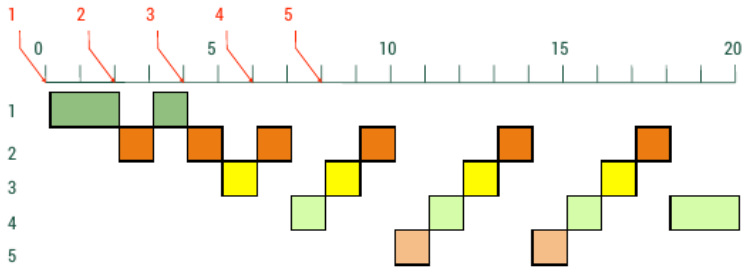


Abbildung 6.6: Round Robin mit $\tau = 1$

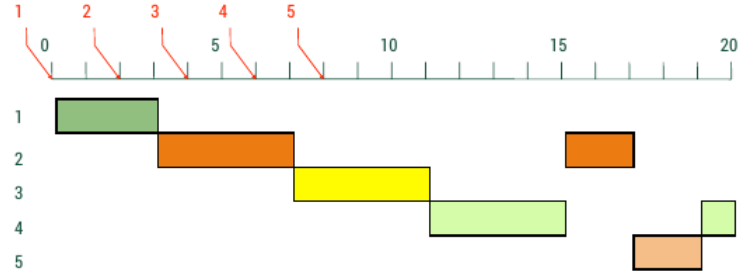


Abbildung 6.7: Round Robin mit $\tau = 4$

Die Abbildungen 6.6 und 6.7 illustrieren das Verfahren anhand unseres Eingangsbeispiels in Tabelle 6.2 für $\tau = 1$ und $\tau = 4$. Zu Beginn ist Prozess 1 der einzige Prozess in der Bereitliste, d. h. auch nach Ablauf der Frist von einer Zeiteinheit (Abbildung 6.6) ist Prozess 1 in Prozessorbesitz. Zum Zeitpunkt $t = 2$ wird Prozess 2 am Ende der Bereitliste eingefügt und zwar **vor** Prozess 1, d. h. vor der verdrängten Prozess. Nachdem die Zeitscheibe für Prozess 2 abgelaufen ist, ist wiederum Prozess 1 an der Reihe. Prozess 2 wird an das Ende der Bereitliste gestellt. Zum Zeitpunkt $t = 4$ tritt Prozess 3 in das Geschehen und wird hinter Prozess 2 in die Bereitliste gestellt. Durch den Ablauf der Frist für Prozess 1 (welcher aufgrund seiner Bedienzeit von 3 Zeiteinheiten die Bereitliste endgültig verlässt), kommt nun wieder Prozess 2 an die Reihe, welcher sich zunächst nur mit Prozess 3 abwechselt. Sukzessive werden dann neue Prozesse in die Bereitliste aufgenommen, während abgearbeitete Prozesse die Bereitliste verlassen. Wartezeit und Prozessorzeit werden indes gleichmäßig unter den Prozessen innerhalb der Bereitliste aufgeteilt. Abbildung 6.7 kommt durch das selbe Verfahren, jedoch mit höherer Zeitscheibenlänge zustande. Wie beschrieben, nähert sich RR mit steigendem τ dem Ablaufplan des FCFS (vgl. Abbildung 6.3) an.

6.2.5 Priorities – Non-Preemptive (PRIO-NP)

Das Schedulingverfahren PRIO-NP ordnet neu ankommende Prozesse nach ihrer Priorität in die Bereitlise ein. Nachdem auf einen Prozess umgeschaltet wurde, ist dieser Prozess solange in Besitz des Prozessors, bis der Prozess vollständig abgearbeitet ist oder freiwillig

aufgibt. Ein laufender Prozess kann also nicht durch einen anderen Prozess verdrängt werden.

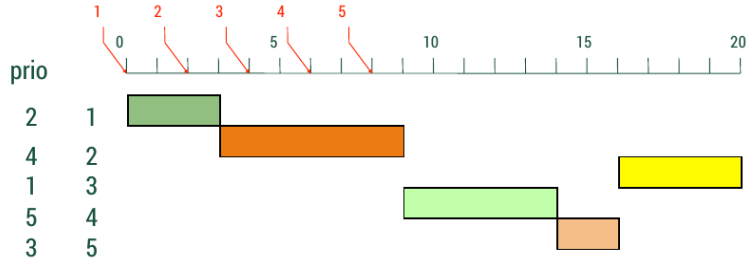


Abbildung 6.8: Priorities – Non-preemptive

Abbildung 6.8 verdeutlicht die Vorgehensweise anhand unseres Beispiels (vgl. Tabelle 6.2). Zum Zeitpunkt $t = 0$ ist Prozess 1 der einzige Prozess in der Bereitliste und erhält somit den Prozessor. Während Prozess 1 läuft kommt Prozess 2 – ein Prozess mit höherer Priorität – an und muss warten, bis Prozess 1 beendet ist bzw. freiwillig aufgibt. Die Wirkung der Priorität zeigt sich also erst, wenn sich mehrere Prozesse in der Bereitliste befinden – im Beispiel also 3, 4 und 5. Aus diesem Ensemble wird nach PRIO-NP zuerst Prozess 4 abgearbeitet, weil dieser mit einer Priorität von 5 die höchste Priorität aufweist. Anschließend kommt dann Prozess 5 mit einer Priorität von 3 und zum Abschluss Prozess 3 mit Priorität 1 an die Reihe.

6.2.6 Priorities – Preemptive (PRIO-P)

Das nun vorgestellte PRIO-P Verfahren funktioniert wie das soeben beschriebene PRIO-NP Verfahren, wird allerdings um das Prinzip der Verdrängung ergänzt, d. h. der rechnende Prozess wird verdrängt, wenn er eine geringere Priorität hat als der neu ankommende Prozess.

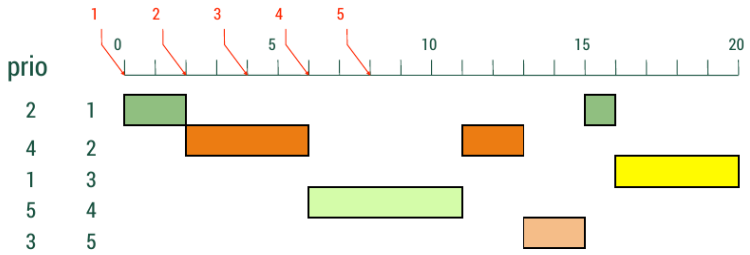


Abbildung 6.9: Priorities – Preemptive

Im Gegensatz zum Schedulingplan in Abbildung 6.8 wird Prozess 1 hier (Abbildung 6.9 durch den neu ankommenden und höher priorisierten Prozess 2 verdrängt. Damit verbleibt Prozess 1 solange in der Bereitliste, bis er der Prozess mit der höchsten Priorität innerhalb der Bereitliste ist.

6.2.7 Shortest Job Next (SJN)

SJN entspricht im Prinzip dem Verfahren PRIO-NP, wobei die Bedienzeit als Prioritätskriterium verwendet wird: Es wird stets der Prozess mit der kürzesten Bedienzeit bis zum Ende oder zur freiwilligen Aufgabe bearbeitet. Das Verfahren führt im Vergeleich mit FCFS zu kürzeren mittleren Antwortzeiten, weil es kurze Prozesse bevorzugt. Allerdings kann diese Vorgehensweise auch dazu führen, dass lange Prozess „verhungern“ (d. h. keinen Zugang zum Prozessor erhalten) – nämlich dann, wenn sich stets Prozesse mit kürzerer Bedienzeit in der Bereitliste befinden. In der Praxis ist mit diesem Verfahren auch die Problematik verbunden, dass man i.d.R. nicht weiß, welche Bedienzeit ein Prozess aufweist. Eine Möglichkeit, einen Wert für die Bedienzeit zu erhalten, ist eine Schätzung durch den Nutzer. Weitere Möglichkeiten, die Bedienzeit in das Scheduling der Prozesse einfließen zulassen (ohne die Bedienzeit von vornherein kennen zu müssen), werden in den Abschnitten 6.2.9 und 6.2.10 vorgestellt.

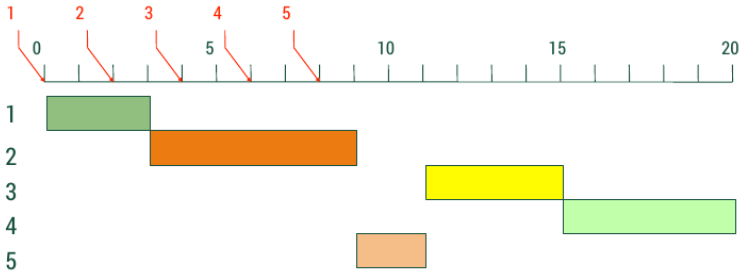


Abbildung 6.10: Shortest Job Next

Für den Ablaufplan in Abbildung 6.10 wurden die Bedienzeiten unserem Beispiel in Tabelle 6.2 entnommen.

6.2.8 Shortest Remaining Time Next (SRTN)

Im SRTN-Schedulingverfahren wird stets der Prozess der Bereitliste als nächster bearbeitet, der die kürzeste Restbedienzeit aufweist. Im Gegensatz zur SJN-Vorgehnsweise kann der rechnende Prozess hier allerdings verdrängt werden. Ebenso wie SJN hat auch dieses Verfahren den Nachteil, dass wir die Bedienzeit benötigen und dass längere Prozesse „verhungern“ können.

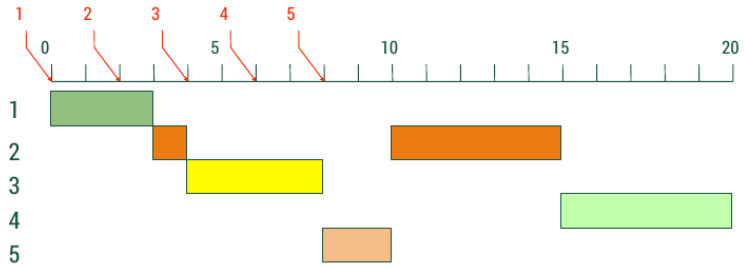


Abbildung 6.11: Shortest Remaining Time Next

Zur Verdeutlichung des Prinzips von SRTN betrachten wir in Abbildung 6.11 die Prozesse 2 und 3. Zum Zeitpunkt $t = 4$, zu dem Prozess 3 ankommt, ist Prozess 2 in Besitz des Prozessors. Prozess 2 mit einer Restbedienzeit von 5 Zeiteinheiten wird jetzt allerdings durch den neu ankommenden Prozess 3 verdrängt, weil dieser lediglich eine Restbedienzeit von 4 Zeiteinheiten aufweist.

6.2.9 Highest Response Ratio Next (HRN)

Das HRN-Verfahren basiert auf der dynamischen Berechnung der Response Ration (deutsch: Antwortverhältnis) $rr = \frac{Wartezeit+Bedienzeit}{Bedienzeit}$. Die so berechnete Response Ratio wird anschließend als Priorität verwendet und der Prozess mit der höchsten Response Ratio als nächster ausgewählt. Der ausgewählte Prozess wird bis zum Ende bzw. bis zur freiwilligen Aufgabe ausgeführt. Da eine kurze Bedienzeit zu einem kleinen Wert im Nenner von rr führt, werden hier, ebenso wie bei SJN, kurze Prozesse bevorzugt. Lange Prozesse müssen aber nicht ewig warten („verhungern“), sondern können durch Warten „Punkte sammeln“, weil mit der Wartezeit der Wert im Zähler von rr steigt.

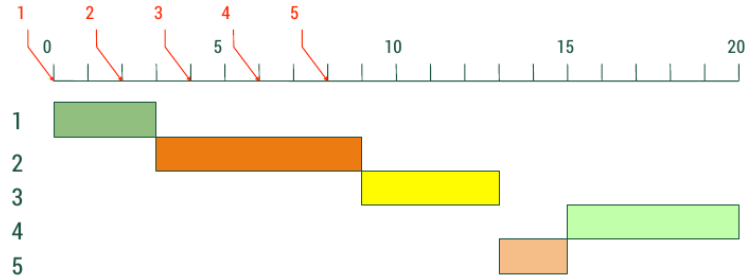


Abbildung 6.12: Highest Response Ratio Next

6.2.10 (Multilevel) Feedback (FB)

Kennt man die Bedienzeit a priori nicht (was in der Praxis häufig der Fall ist), möchte aber trotzdem lang laufende Prozesse benachteiligen, so kann man den Prozess nach jeder CPU-Benutzung in seiner „Priorität“ schrittweise herabsetzen. Dies kann bspw. mithilfe von Warteschlangen geschehen, die jeweils Prozesse einer Priorität aufnehmen. Die

Prozesse innerhalb der einzelnen Warteschlangen können dann bspw. mithilfe des Round Robin Verfahrens geplant werden. So ist es dann auch möglich, die Zeitscheibenlänge in Abhängigkeit der Warteschlange bzw. Priorität festzulegen. Prozesse mit niedriger Priorität (d. h. Prozesse, die schon häufig in Besitz des Prozessors waren) könnten dann als Ausgleich für die sinkende Priorität längere Zeitscheiben erhalten. Abbildung 6.16 zeigt den Aufbau einer Multilevel Feedback Warteschlange, bestehend aus den Warteschlangen 0 bis n mit absteigender Priorität. Je länger ein Prozess insgesamt in Besitz der CPU war, umso tiefer sinkt er in der Warteschlangenhierarchie. Mit jedem Level, das der Prozess fällt, sinkt seine Priorität und damit die Häufigkeit, mit der der Prozess dem Prozessor zugeordnet wird.

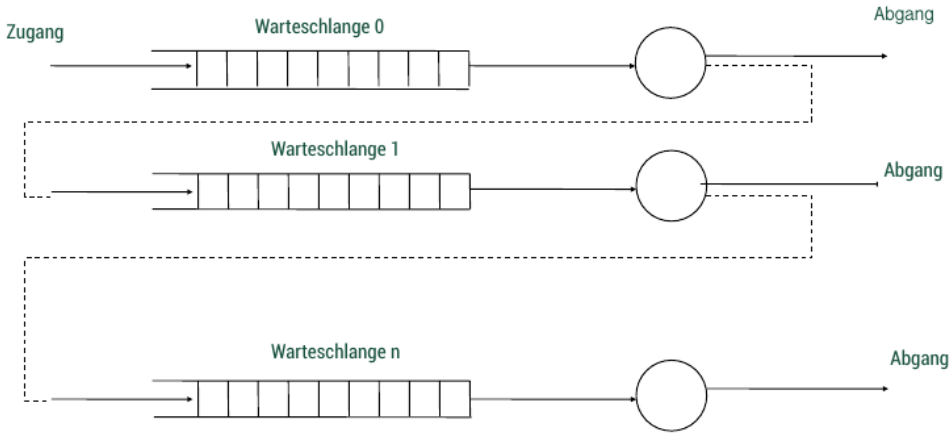


Abbildung 6.13: Multilevel Feedback Warteschlange

Abbildung 6.14 zeigt das Scheduling der Prozesse aus Tabelle 6.2 mittels Multilevel Feedback für den Fall, dass für die Zeitscheibenlänge τ für alle Warteschlangen $\tau = 1$ gilt. Die Werte innerhalb der Balken stehen für die Warteschlange i, in der sich der Prozess zu diesem Zeitpunkt befindet. Für jede Nutzung der CPU (d. h. nach jedem Ablauf der Zeitscheibe τ) wird i um Eins erhöht.

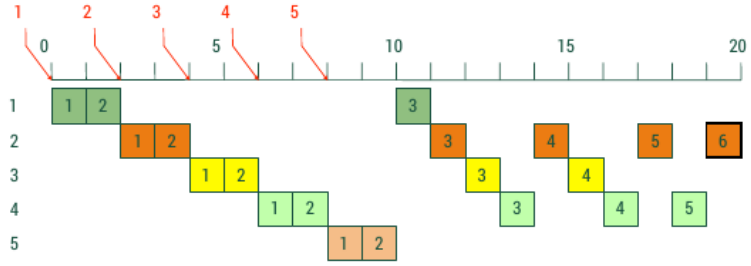


Abbildung 6.14: Multilevel Feedback mit $\tau = 1$

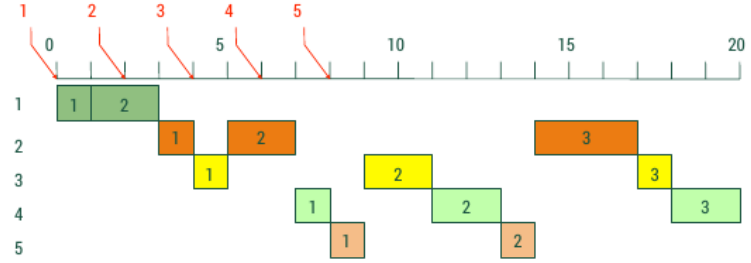


Abbildung 6.15: Multilevel Feedback mit $\tau = 2^i$

Abbildung 6.15 zeigt als Ergänzung dazu den Ablaufplan mittels Multilevel Feedback, wenn die Zeitscheibenlänge τ von der Priorität bzw. Warteschlange i abhängt. Da $\tau = 2^i$ ist, verdoppelt sich die Länge der Zeitscheibe mit sinkender Priorität. Prozesse mit niedrigerer Priorität kommen zwar immernoch seltener, aber dafür dann für längere Zeit zum Zug.

6.3 Fallbeispiel UNIX

In diesem Abschnitt betrachten wir abschließend das Prozessscheduling im klassischen UNIX (BSD, System V). Schedulingziel ist hier die Bevorzugung interaktiver Jobs, d. h. die möglichst schnelle Reaktion auf Aktionen des Nutzers. Die Darstellung der Prioritäten erfolgt mithilfe des Multilevel Feedback Verfahrens. Die Prioritätswerte laufen dabei von 0 bis 127, wobei 0 der höchsten Priorität und 127 der niedrigsten Priorität entspricht. Im Vergleich mit dem in Abschnitt 6.2.10 vorgestellten Verfahren besteht in UNIX der Unterschied, dass Prioritäten zusätzlich zu Klassen zusammengefasst werden. Eine Prioritätsklasse umfasst dabei 4 Prioritätswerte. Eine Warteschlange enthält jeweils Prozesse genau einer Prioritätsklasse. Bei 128 Prioritäten führt diese Vorgehensweise zu 32 Warteschlangen. Abbildung 6.16 veranschaulicht diesen Aufbau.

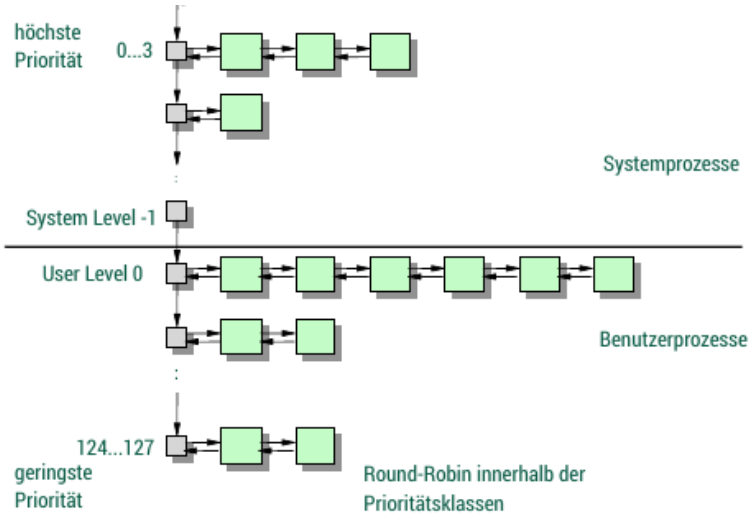


Abbildung 6.16: Multilevel Feedback Warteschlange unter UNIX

6.3.1 Berechnung der Priorität

Der Timer-Tick liegt typischerweise bei 10ms. Bei jedem vierten Tick, also alle 40ms, werden die Prozessprioritäten nach folgender Formel ermittelt:

$$P_{Proc} = P_{Basis} + floor\left(\frac{L_{CPU}}{4}\right) + 2 \cdot P_{nice}$$

Dabei ist P_{Basis} die Basispriorität, welche von Art und Zustand des Prozesses abhängt. Normale Nutzerprozesse starten dabei mit dem am höchsten priorisierten Nutzerprozess-Level (oft zwischen 50 und 60). Kernelprozesse haben in Abhängigkeit der Aufgabe höherere Prioritäten. L_{CPU} ist die Abschätzung der CPU-Zeit des Prozesses. Diese wird mit jedem Tick um Eins erhöht. P_{nice} ist eine vom Nutzer festlegbare Wichtigkeit des Prozesses. Ist der berechnete Wert für P_{Proc} größer als 127, so wird P_{Proc} auf 127 festgelegt.

6.3.2 Alterung

Als „Alterung“ bezeichnet man das Zunehmen des Prioritätswerts durch starke Prozessorbenutzung. Rechenintensive Prozesse altern und werden durch den steigenden Prioritätswert „benachteiligt“. I/O-intensive Prozesse, welche den Prozessor nur sehr selten benutzen, weil sie bspw. nur kurz einen I/O-Auftrag absetzen, werden hingegen bevorzugt – altern also langsamer. Durch das Konzept der Alterung wird eine hohe Parallelität zwischen den aktiven Rechnerkomponenten erreicht, d. h. zwischen CPU und Peripheriegeräten.

6.3.3 Glättung

Da der Wert L_{CPU} und damit die Priorität mit der Zeit wächst, hätten alle (hinreichend lange laufenden) Prozesse nach einiger Zeit sehr schlechte Prioritätswerte. Aus diesem Grund findet einmal pro Sekunde eine Glättung des Prozessornutzungswertes L_{CPU} statt, wodurch der Einfluss der letzten Rechenzeiten gedämpft wird:

$$L_{CPU} = \frac{2 \cdot l_{RQ}}{2 \cdot l_{RQ} + 1} \cdot L_{CPU} + P_{nice}$$

l_{RQ} ist dabei der Durchschnitt der Anzahl der Prozesse in der Bereitmenge innerhalb der letzten Minute. Umso geringer also die Anzahl der fortsetzungsfähigen Prozesse innerhalb der letzten Minute war, desto stärker wird L_{CPU} verringert. Eine Verringerung dieses Wertes führt wiederum zu besseren Prioritätswerten.

Zusammenfassung

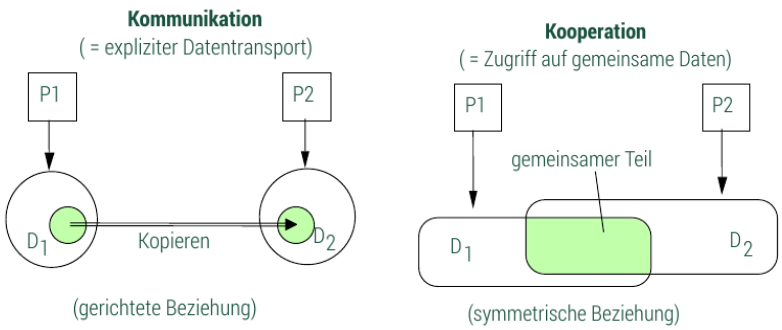
In diesem Kapitel haben Sie gelernt, dass die Wahl des Schedulingverfahrens vom Schedulingziel abhängt. Des Weiteren wurden diverse Schedulingverfahren vorgestellt und hinsichtlich der Schedulingziele diskutiert. Tabelle 6.3 gibt, gruppiert nach Verdrängung, Verwendung von Prioritäten und Bedienzeitabhängigkeit, einen Überblick über die erläuterten Schedulingverfahren.

	Ohne Verdrängung		Mit Verdrängung	
	Ohne Prioritäten	Mit Prioritäten	Ohne Prioritäten	Mit Prioritäten
BZ-unabhängig	FCFS, LCFS	PRIO-NP	LCFS-PR, RR, FB	PRIO-P
BZ-abhängig	SJN, HRN		SRTN	

Tabelle 6.3: Übersicht Standard-Umschaltstrategien

7 Prozesse im Zusammenspiel: Prozessinteraktion

Nachdem wir uns in den Kapiteln 5.2 und 6 mit der Verwaltung von Prozessen beschäftigt haben, betrachten wir in diesem Kapitel mit der Organisation der Prozessinteraktion einen weiteren Aufgabenbereich des Betriebssystems. Da Prozesse als Teile komplexer Systeme nebeneinander existieren, müssen sie u.U. Daten austauschen, d. h. miteinander interagieren. Das ist beispielsweise dann der Fall, wenn Prozesse sich aufrufen, aufeinander warten, sich auslösen oder abstimmen. Die Prozessinteraktion wird dabei durch einen zeitlichen und durch einen funktionalen Aspekt charakterisiert. Der zeitliche Aspekt entspricht dabei der Ablaufabstimmung bzw. Koordination des Datenaustauschs. Der Datenaustausch selbst entspricht dem funktionalen Aspekt, d. h. dem Zweck der Prozessinteraktion. Beim Austausch von Daten kann zwischen den beiden in Abbildung 7.1 dargestellten Interaktionsarten „Kommunikation“ und „Kooperation“ unterschieden werden.



(a) Kommunikation (b) Koordination

Abbildung 7.1: Interaktionsarten

Demnach ist die Koordination die elementarste Grundform der Interaktion, denn sowohl Kommunikation als auch Kooperation benötigen eine zeitliche Abstimmung zwischen den Interaktionspartnern (vgl. Abbildung 7.2). Im nachfolgenden Abschnitt betrachten wir deshalb zunächst die Koordination von Prozessen, um anschließend in den Abschnitten 7.2 und 7.3 dieses Wissen als Basis für Kommunikation und Kooperation zu nutzen.

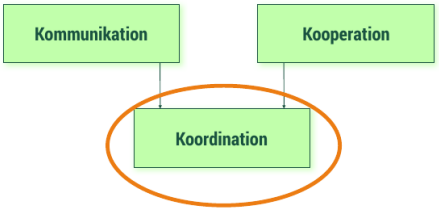


Abbildung 7.2: Koordination als Basis für Kommunikation und Kooperation

7.1 Koordination

Zunächst widmen wir uns der Frage, warum Koordination für die Interaktion von Prozessen notwendig ist. Dazu zeigt Abbildung 7.3 zwei Codebeispiele, die von je einem Prozess ausgeführt werden.

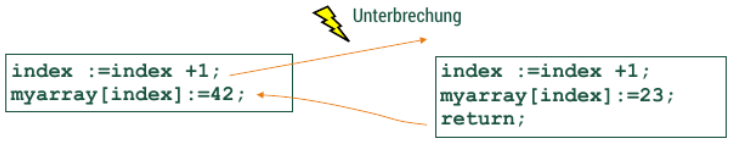


Abbildung 7.3: Racecondition

Bedingt durch den Prozessscheduler des Betriebssytems (vgl. Kapitel 6) kann die Ausführung eines Prozesses jederzeit unterbrochen werden. Dieses Verhalten kann zu sogenannten Raceconditions bzw. Wettlaufsituationen führen, wenn konkurrierende Prozesse auf die selben Datenstrukturen zugreifen. In Abbildung 7.3 ist genau das der Fall. Dort wird zunächst die erste Zeile des linken Codeschnipsels ausgeführt, d. h. `index` um Eins erhöht. Anschließend wird die Ausführung unterbrochen und der rechte Prozess fortgesetzt: `index` wird dabei erneut um Eins erhöht und `myarray` an der entsprechenden Position auf `23` gesetzt. Das `return`-Statement führt schließlich zur Fortsetzung des linken Prozesses und damit dazu, dass der soeben an Position `myarray[index]` geschriebene Wert `23` mit dem Wert `42` überschrieben wird. Abgesehen davon, dass der Wert `23` dadurch verloren geht, bleibt `myarray` an Position `index-1` unbestimmt, weil `index` durch die verzahnte Ausführung zweimal direkt nacheinander erhöht wurde. Von der Abbildung einmal abgesehen, besteht auch die Möglichkeit, dass zuerst der linke und anschließend der rechte Ausführungsstrang ohne Unterbrechnung ausgeführt wird. Des Weiteren kann sich dieses Szenario auch andersherum abspielen: Zuerst wird der rechte Ausführungsstrang komplett abgearbeitet und anschließend der linke. Dabei wird jedes dieser Ausführungsszenarios zu einem anderen Inhalt des Arrays führen. Das Verhalten bei der nebenläufigen Ausführung dieser beiden Codeschnipsel ist entsprechend unbestimmt, weil man nicht wissen kann, wann und ob die Prozesse in ihrer Ausführung unterbrochen werden, d. h. das Ergebnis der Ausführung ist nicht sicher vorhersagbar. Das Beispiel zeigt, dass die Interaktion von

Prozessen koordiniert erfolgen muss, um ein vorhersagbares Ergebnis zu erhalten.
Um die Problematik des konkurrierenden Zugriffs auf Daten zu lösen, betrachten wir im nächsten Abschnitt das Koordinationskonzept „kritischer Abschnitt“.

7.1.1 Konzept: Kritischer Abschnitt

Ein kritischer Abschnitt (critical section) ist ein Bereich (z. B. Daten, Befehle), auf den innerhalb einer bestimmten Zeitspanne keine konkurrierenden Zugriffe zulässig sind. Prozesse, die miteinander interagieren, haben häufig gemeinsame kritische Abschnitte (sie aktualisieren bspw. gemeinschaftlich eine Zählvariable). Ziel dieses Koordinationskonzepts ist entsprechend der gegenseitige Ausschluss (mutual exclusion), d. h. dafür Sorge zu tragen, dass sich zu einem Ausführungszeitpunkt nur ein einziger Befehlsstrom (Prozess, Thread) innerhalb eines kritischen Abschnitts aufhält.

Ein Beispiel für die Verwendung kritischer Abschnitte ist die Ausführung von Kernoperationen, die wir in diesem Exkurs näher betrachten werden. Da Kernoperationen häufig auf gemeinsame Datenstrukturen zugreifen, ist die Unterbrechung von Kernoperationen durch Kernoperationen ungünstig. Um Raceconditions zu vermeiden, müssten demnach alle potentiellen Konfliktstellen im Betriebssystemkernel identifiziert und zu kritischen Abschnitten erklärt werden. Weil im Kern wiederum eine Vielzahl an kritischen Abschnitten vorhanden ist und Kernoperationen i.d.R. wenig Zeit kosten, können wir uns diesen Aufwand sparen, indem wir den gesamten Kern als kritischen Abschnitt auffassen. Das unter gegenseitigen Ausschluss Stellen des gesamten Kerns wird als Kernausschluss bezeichnet. Im nachfolgenden Abschnitt betrachten wir die Realisierung eines solchen Kernausschlusses.

Realisierung des Kernausschlusses

Die Realisierung des Kernausschlusses hängt zum einen von der Anzahl der Prozessoren (auch „Kerne“, nicht zu verwechseln mit dem Betriebssystemkern) im System ab und zum anderen davon, ob das System einen Unterbrechungsmechanismus aufweist. Aus diesem Grund unterscheiden wir die folgenden vier Fälle:

1. Einkernsystem ohne Unterbrechungen
2. Einkernsystem mit Unterbrechungen
3. Mehrkernsystem ohne Unterbrechungen
4. Mehrkernsystem mit Unterbrechungen

In den nachfolgenden Abschnitten betrachten wir die Realisierung des Kernausschlusses für jeden dieser Fälle im Detail.

Fall 1: Einkernsystem ohne Unterbrechungen Da das System keinen Unterbrechungsmechanismus besitzt, kann auch keine Unterbrechung von Kernoperationen stattfinden. Demnach erfordert ein solches System keine speziellen Sicherungsmaßnahmen, um die Unterbrechung von Kernoperationen durch Kernoperationen zu verhindern.

Fall 2: Einkernsystem mit Unterbrechungen Für den Fall, dass ein Unterbrechungsmechnaismus vorliegt, ist es grundsätzlich möglich, dass eine Kernoperation durch eine andere Kernoperation unterbrochen wird. Entsprechend müssen wir den Kernausschluss durch eine geeignete Sicherungsmaßnahme realisieren. Eine Möglichkeit, den Kernausschluss für ein solches System (siehe Abbildung 7.4) zu implementieren zeigt Abbildung 7.5.

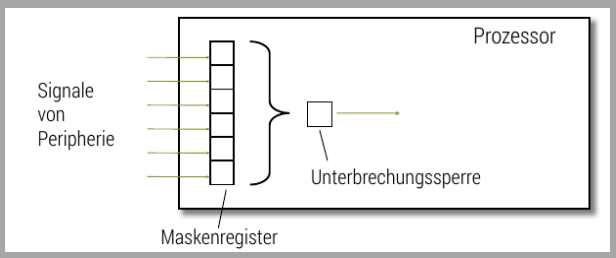


Abbildung 7.4: Einkernsystem mit Unterbrechungsmechanismus

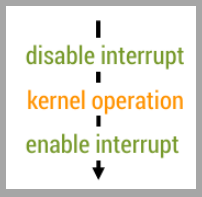


Abbildung 7.5: Realisierung des Kernausschlusses für Einkernsystem mit Unterbrechungsmechanismus

Wie im Ablaufdiagramm in Abbildung 7.5 zu sehen ist, wird mittels `disable interrupt` vor der Ausführung von Kernoperationen eine Unterbrechungssperre gesetzt, d. h. die Kernoperation kann nicht unterbrochen werden. Nach Abschluss der Kernoperation lösen wir die Unterbrechungssperre mittels `enable interrupt` wieder.

Fall 3: Mehrkernsystem ohne Unterbrechungen In Mehrkernsystemen kann es trotz Unterbrechungsverbot zu einer verzahnten Ausführung von Kernoperationen kommen. Das ist dann der Fall, wenn die Operationen simultan auf zwei Prozessoren ausgeführt werden und entsprechend verzahnt auf Datenstrukturen zugreifen.
Was wir benötigen, ist demnach eine Sperroperation, die Kernoperationen unter gegenseitigen Ausschluss stellt. Diese Operation muss zunächst prüfen, ob die Kernsperre gesetzt ist. Ist das der Fall, so darf keine andere Kernoperation durchgeführt werden. Ist die Kernsperre jedoch noch nicht gesetzt, so wird sie gesetzt und die Kernoperation kann durchgeführt werden. Das Problem hierbei ist, dass diese Operation selbst wiederum eine Kernoperation und damit ein kritischer Abschnitt ist, was dazu führt, dass sie unteilbar (atomar) laufen muss. Um eine derartige Kernsperre zu realisieren und damit diese pro-

blematische Rekursion aufzulösen, benötigen wir also einen Befehl, der in einer atomaren Operation den Wert einer Bedingungsvariable abfragt und sie gleichzeitig setzt. Dieser Befehl steht i. d. R. in Form eines Maschinenbefehls (`test_and_set`, `compare_and_swap`) zur Verfügung. Bei Aufruf von `test_and_set(Reg, x)` wird der Wert `x` in das Register `Reg` kopiert und `x` – unabhängig von seinem Wert – auf `1` gesetzt. War `x` also gleich `0`, so wird `x` auf `1` gesetzt, sonst bleibt der Wert unverändert bei `1`. Mithilfe dieses Befehls zum unteilbaren Abfragen und Schreiben einer Variable kann nun, wie in Abbildung 7.6 gezeigt, eine Kernsperre realisiert werden.

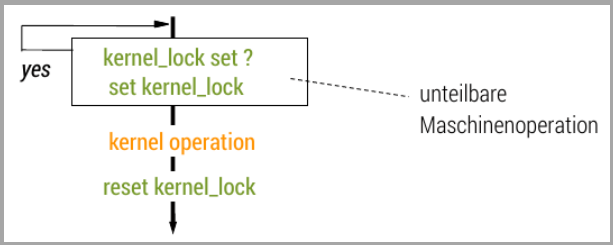


Abbildung 7.6: Realisierung des Kernausschlusses für Mehrkernsystem ohne Unterbrechungsmechanismus

Solange die Kernsperre belegt ist, wird in einer Schleife der Wert der Kernsperre immer wieder abgefragt. Es wird also darauf gewartet, dass die Kernsperre gelöst wird und somit für eine andere Kernoperation erneut gesetzt werden kann. Dieses Vorgehen wird als aktives Warten („busy waiting“) bezeichnet und eine so implementierte Sperre heißt auch „spin lock“. Aktives Warten führt zu einer gewissen Verschwendung von Rechenkapazität, weil der Prozessor stets damit beschäftigt ist, den Wert einer Variablen abzufragen, wobei kein echter Programmfortschritt zustande kommt. Da Kernoperationen relativ kurz sind, kann das aktive Warten an dieser Stelle allerdings toleriert werden. Diese Vorgehensweise wird häufig auch in Einkernsystemen mit Unterbrechungsmechanismus genutzt.

Fall 4: Mehrkernsystem mit Unterbrechungen Um den Kernausschluss für Mehrkernsysteme mit Unterbrechungsmechanismus zu realisieren, müssen nun die beiden Techniken – Unterbrechungssperre und Kernsperre – aus den vorhergehenden Abschnitten kombiniert werden. Das Ablaufdiagramm hierfür zeigt Abbildung 7.7

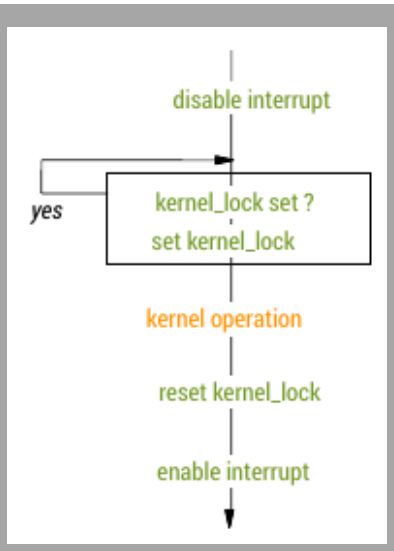


Abbildung 7.7: Realisierung des Kernausschlusses für Mehrkernsystem mit Unterbrechungsmechanismus

Wie im Ablaufdiagramm gezeigt, setzen wir zuerst die Unterbrechungssperre und dann erst die Kernpserre. Das Setzen der Sperren muss in exakt dieser Reihenfolge geschehen, weil auch Unterbrechungsbehandlungen Kernoperationen sind. Wenn zuerst die Kernsperre und anschließend die Unterbrechungssperre gesetzt würde, könnte dieser Umstand zum „Hängenbleiben“ des Prozesses führen: Nämlich genau dann, wenn nach dem Setzen der Kernsperre eine Unterbrechung auftritt. In diesem Fall würde die Routine für die Unterbrechungsbehandlung vergeblich versuchen, die bereits gesetzte Kernsperre zu setzen.

Softwarelösung für gegenseitigen Ausschluss

Während im Exkurs zum Kernausschluss 7.1.1 auf Hardware-Unterstützung in Form von atomaren Operationen zurückgegriffen wird, betrachten wir hier eine reine Softwarelösung zur Realisierung des gegenseitigen Ausschlusses. Die Grundidee für den in Listing 7.1 vorgestellten Ansatz geht auf Dekker zurück und wurde von Peterson weiterentwickelt. Das Listing zeigt die Implementation des gegenseitigen Ausschlusses für zwei konkurrierende Prozesse.

Listing 7.1: Softwarelösung für gegenseitigen Ausschluss nach Dekker/Peterson

```
1 const int N = 2; // Anzahl von Konkurrenten
2 volatile int turn; // Wer ist dran?  (shared)
3 volatile bool interested[N]; // Wer will? (shared)
4 void enter_section(int processID) // Wer ruft? - 0 oder 1
5 {
6   int other = 1 - processID; // ID des anderen Prozess
7   interested[processID] = true; // Ich will
8   turn = processID;
```

```
9   while ((turn == processID) && (interested[other] == true)); //warten
10 }
11
12 void leave_section(int processID) // Wer tritt aus krit. Bereich aus? - 0 oder 1
13 {
14   interested[processID] = false;
15 }
```

7.1.2 Koordination auf Prozessebene

Während wir uns im vorherigen Abschnitt mit dem gegenseitigen Ausschluss auf Kernebene beschäftigt haben, betrachten wir hier den gegenseitigen Ausschluss auf Prozessebene, da auch Prozesse kritische Abschnitte miteinander teilen können. Die Koordination auf Prozessebene kann durch den Zugriff auf kritische Abschnitte bzw. gemeinsame Daten über Kernaufrufe realisiert werden. Hierfür stehen auf Prozessebene abstraktere Konzepte zur Verfügung, die wir nachfolgend betrachten.

Konzept: Signalisierung

Das Konzept der Synchronisierung dient der Herstellung einer Reihenfolgebeziehung. In Abbildung 7.8 soll beispielsweise der Abschnitt A in Prozess P1 vor dem Abschnitt B in Prozess P2 ausgeführt werden.

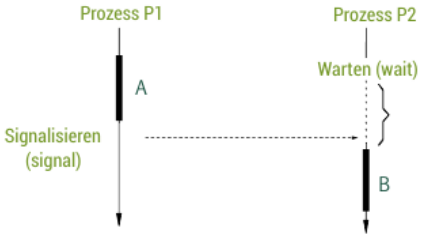


Abbildung 7.8: Signalisierung

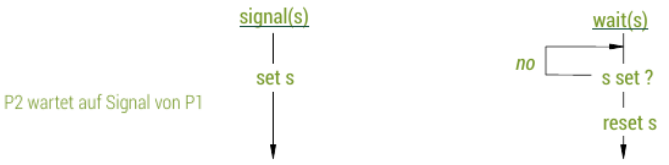


Abbildung 7.9: Einfache Realisierung von `signal(s)` und `wait(s)`

Zu diesem Zweck bietet der Betriebssystemkern die Operationen `signal(s)` und `wait(s)` an, die eine gemeinsame, geschützte, binäre Variable `s` verwenden. In der Abbildung wird also zunächst Abschnitt A in Prozess 1 ausgeführt, während Prozess P2 mittels `wait(s)` auf ein Signal von Prozess P1 wartet. Mittels `signal(s)` in P1 wird anschließend der Abschluss der Ausführung von Abschnitt A signalisiert, wodurch die Ausführung von Prozess P2 fortgesetzt werden und Abschnitt B betreten werden kann.
Abbildung 7.9 zeigt eine einfache Implementation der Operationen `signal` und `wait` durch aktives Warten an der Signalisierungsvariable `s`. Da die Wartezeit auf Prozessebene beliebig lang sein kann, ist aktives Warten hier ungünstig. Ist die Wartezeit zu hoch, so sollte

der Prozessor besser für andere Prozesse freigegeben werden. Eine mögliche Realisierung dieses Verhaltens zeigt Abbildung 7.10.

Abbildung 7.10: Signalisierung

In der Operation `wait(s)` wird zunächst geprüft, ob die Signalisierungsvariable `s` bereits gesetzt ist. Ist das der Fall, wird `s` zurückgesetzt und die Ausführung ohne Verzögerung fortgesetzt. Andernfalls wird der wartende Prozess blockiert (vgl. Abschnitt 5.2.2). In `signal(s)` wird zunächst die Signalisierungsvariable `s` gesetzt. Anschließend wird geprüft, ob ein Prozess auf das Signal wartet. Ist das der Fall, so wird der wartende Prozess deblockiert.
Nachfolgend betrachten wir den Einsatz des Signalisierungskonzepts zu Synchronisierungs- und Sperrzwecken.

Wechselseitige Synchronisierung Mithilfe des symmetrischen Einsatzes von `signal` und `wait`, wie in Abbildung 7.11 illustriert, können Prozesse an bestimmten Ausführungsstellen miteinander synchronisiert werden.

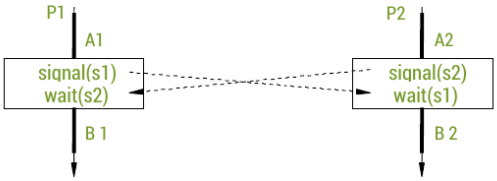


Abbildung 7.11: Wechselseitige Synchronisierung

Abbildung 7.12: Zusammenfassung von `signal(s)` und `wait(s)` in `sync(s)`

Durch die wechselseitige Synchronisierung wird im gezeigten Beispiel sichergestellt, dass sowohl Abschnitt A1 als auch Abschnitt A2 ausgeführt sind, bevor die Abschnitte B1 und B2 ausgeführt werden. Betrachten wir als Beispielszenario den Fall, dass Prozess P1 mit der Ausführung von Abschnitt A1 fertig ist, bevor Prozess P2 Abschnitt A2 abschließt. In diesem Fall wird also zunächst A1 ausgeführt. Anschließend setzt P1 die Signalisierungsvariable `s1` mittels `signal(s1)` und wartet in `wait(s2)` auf das Setzen von `s2` durch P2. Sobald nun Prozess P2 den Abschnitt A2 abgeschlossen hat, setzt P2 die Signalisierungsvariable `s2`, was dazu führt, dass P1 fortgesetzt werden kann. Da `s1` bereits

gesetzt ist, kann auch P2 die Ausführung direkt fortsetzen. Das Operationspaar `signal` und `wait` kann zu einer Operation `syncs` zusammengefasst werden (siehe Abbildung 7.12), weil sich P1 und P2 an dieser Stelle synchronisieren. Die wechselseitige Synchronisierung wird auch als Rendezvous bezeichnet, weil die Prozesse aufeinander warten.

Sperren Neben der wechselseitigen Synchronisierung dient das Signalisierungskonzept auch der Sicherung von kritischen Abschnitten auf Prozessebene. Abbildung 7.13 illustriert dazu den Einsatz von `signal` und `wait`, um die nebenläufige Ausführung der Abschnitte A und B zu verhindern.

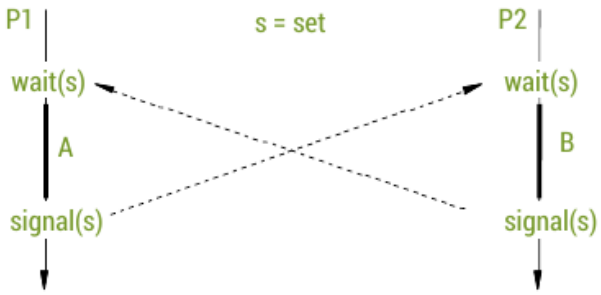


Abbildung 7.13: Einfache Realisierung von `signal(s)` und `wait(s)`

Für die Implementation einer Sperre auf Prozessebene genügt die Verwendung einer einzelnen Signalisierungsvariable `s`. Vor der Ausführung der zu sichernden Abschnitte ist `s` bereits gesetzt, d. h. sobald ein Prozess in `wait(s)` eintritt, kann die Ausführung sofort fortgesetzt werden. Innerhalb der Operation `wait(s)` wird `s` zurückgesetzt. Das führt dazu, dass der andere Prozess, sobald er in `wait(s)` eintritt, auf die Signalisierung von `s` warten muss. Die Signalisierung von `s` wiederum geschieht erst nach der Ausführung des kritischen Abschnitts (A oder B). Die Ausführungen von A und B schließen sich demnach gegenseitig aus.
Um dem Begriff „Sperren“ auch durch die Operationsnamen gerecht zu werden, verwenden wir hier statt `signal(s)` und `wait(s)` `unlock(s)` (entsperren) und `lock` (sperren).

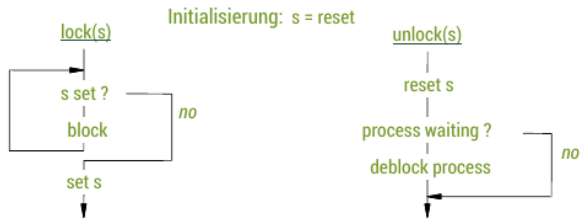


Abbildung 7.14: Realisierung von `lock(s)` und `unlock(s)`

Strukturell entspricht das Sperren `wait(s)` und das Entsperren `signal(s)` (vgl. Abbildung 7.14 und 7.10. Im Unterschied zu `signal(s)` wird hier jedoch die Variable `s` in einer

Schleife abgefragt, um den Fall zu berücksichtigen, dass zwischen dem Deblockieren des auf die Sperre wartenden Prozesses und dem Setzen der Sperre ein weiterer Prozess die Sperre setzen könnte.

7.2 Kommunikation

In diesem Abschnitt betrachten wir die Kommunikation zwischen Prozessen. Kommunikation bedeutet hier, dass die Prozesse explizit Nachrichten austauschen und nicht – wie im Falle der Kooperation (vgl. Abschnitt 7.3) – auf gemeinsamem Speicher arbeiten. Während das Arbeiten auf dem Speicher aus Sicht des Prozesses der „Normalfall“ ist, muss für den Nachrichtenaustausch erst die Möglichkeit geschaffen werden. Dazu führen wir nachfolgend das Konzept des Kanals ein.

7.2.1 Konzept: Kanal

Ein Kanal ist ein Datenobjekt, das die Operationen Senden (`send`) und Empfangen (`receive`) zur Verfügung stellt. Abbildung 7.15 zeigt einen solchen Kanal CO (channel object).

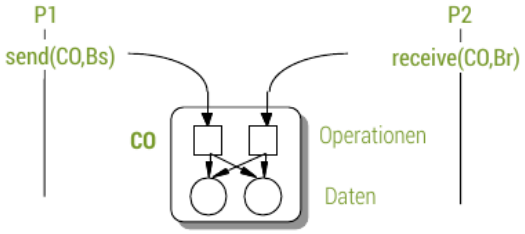


Abbildung 7.15: Kanal

Sende- und Empfangsoperation erhalten als Parameter den Namen des Kanals (hier CO) und die Adresse eines Behälters. Für den Sender ist letztere die Adresse des Quellbehälters Bs (buffer send), welcher die Adresse der zu versendenden Nachricht (bzw. die Nachricht selbst) enthält. Der Empfänger erhält an dieser Stelle hingegen die Adresse des Zielbehälters Br (buffer receive), welcher die Adresse enthält, an die die empfangene Nachricht geschrieben werden soll.
Anhand der Abbildung wird ersichtlich, dass das Kanalkonzept keine zeitliche Ordnung impliziert. D.h. Sender und Empfänger können ihre Operationen zu beliebigen Zeitpunkten aufrufen, womit zwei Fälle zu unterscheiden sind:

1. Erst Senden, dann Empfangen.
2. Erst Empfangen, dann Senden.

Wenn die aufrufenden Prozesse in den Operationen nicht blockiert werden sollen, besteht die Notwendigkeit der Zwischenspeicherung der Quell- und Zieladresse (bzw. der Nachricht) im Kanal. Wird zuerst die Sendeoperation ausgeführt, so wird entsprechend die Nachricht bzw. ihre Adresse im Kanal abgelegt. Diese kann durch die später erfolgende Empfangsoperation abgeholt werden. Wird hingegen die Empfangsoperation zuerst ausgeführt, so wird die Adresse des Zielpuffers im Kanal zwischengespeichert. Bei der später erfolgenden Sendeoperation kann die Nachricht anschließend dort hin kopiert werden. Für die Zwischenspeicherung dieser Daten hält der Kanal Variablen vor. Die soeben beschriebene Kommunikation ohne eine zeitliche Abstimmung zwischen Sender und Empfänger wird auch als asynchrone Kommunikation bezeichnet. Im nachfolgenden Abschnitt wird im Gegensatz dazu die koordinierte Kommunikation betrachtet, d. h. wir fordern eine gewisse zeitliche Abstimmung zwischen Sender und Empfänger ein.

7.2.2 Koordinierte Kommunikation

In der Praxis ist es häufig so, dass bspw. der Empfänger dringend auf den Empfang einer Nachricht angewiesen ist, weil er erst weiterarbeiten kann, wenn die Nachricht eingetroffen ist. Solange die Nachricht also nicht eingetroffen ist, blockiert der Empfänger in der Empfangsoperation. Da der Empfänger also auf den Sender wartet, synchronisiert sich der Empfänger an dieser Stelle mit dem Sender. Dieses Verhalten wird als synchrones Empfangen bezeichnet. Analog dazu ist auch ein synchrones Senden möglich. Dabei wird der Sender solange blockiert, bis die zugehörige Empfangsoperation aufgerufen wurde.
Durch Kombination von synchronen und asynchronen Sende- und Empfangsoperationen ergeben sich die in Abbildung 7.16 dargestellten Koordinationsvarianten der Kommunikation.

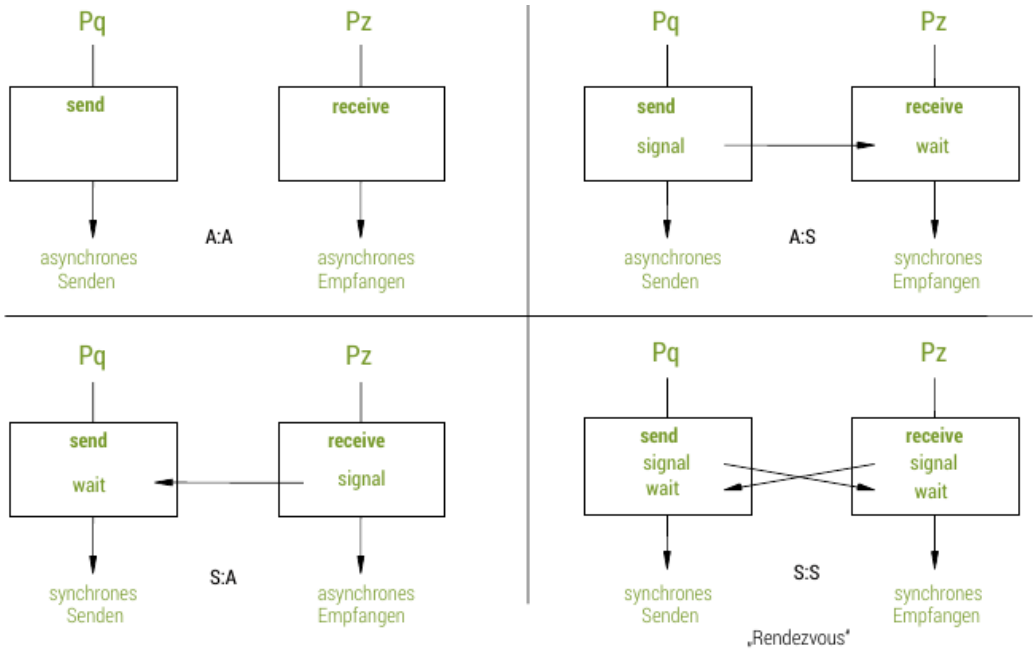


Abbildung 7.16: Koordinationsvarianten der Kommunikation

7.3 Kooperation

Damit mehrere Prozesse auf die selben Daten zugreifen können, müssen die Prozesse miteinander kooperieren. Da Speicherzugriffe an sich aus Prozesssicht etwas Gewöhnliches sind, müssen an dieser Stelle keine neuen Konzepte erdacht werden. Allerdings müssen beim Zugriff auf gemeinsamen Speicher Probleme wie Inkonsistenzen und Kapazitätsbeschränkungen bedacht werden. Abbildung 7.17 zeigt beispielhaft die Entstehung einer inkonsistenten Datensicht während des Einfügens eines Elements in eine doppelt verkettete Liste.

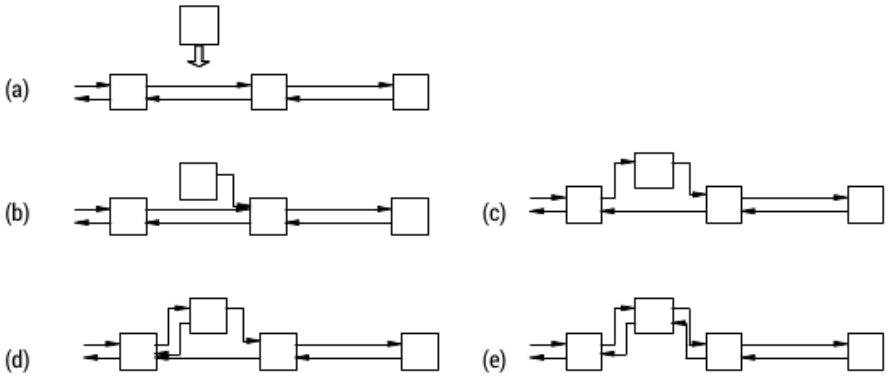


Abbildung 7.17: „Einfügen" in doppelt verkettete Liste in Einzelschritte aufgelöst

Während der Situationen c) und d) ist die Listenstruktur inkonsistent, weil ein konkurrierend auf die Liste zugreifender Prozess eine fehlerhafte Datenstruktur sehen würde.

Die Kooperation von Prozessen auf gemeinsamen Daten fällt offensichtlich mit dem Problem des kritischen Abschnitts bzw. des gegenseitigen Ausschlusses zusammen (vgl. 7.1.1), welches wir bereits im Rahmen der Koordination betrachtet haben. Zur Sicherung kritischer Abschnitte können demnach die bereits eingeführten Sperroperationen `lock(s)` und `unlock(s)` verwendet werden.
Ein Kooperationsabschnitt ist bis dato also dadurch gekennzeichnet, dass sich zu einem Zeitpunkt genau ein Prozess darin aufhält. Dieses Prinzip kann erweitert und auch für andere Kapazitäten verwendet werden. Wir können also sowohl für die Anzahl der Prozesse im Kooperationsabschnitt als auch für die Anzahl der wartenden Prozesse beliebige Obergrenzen festlegen. Gründe für die Begrenzung der Anzahl von Prozessen in einem bestimmten Bereich sind bspw. Platzmangel, Leistungsabfall und ohne Begrenzung entstehende Inkonsistenzen.
Als Beispiel für eine konsistenzbedingte Problematik der Kooperation betrachten wir das Leser-Schreiber-Problem. Denken wir dazu zurück an das Beispiel der doppelt verketteten Liste in Abbildung 7.17, stellen wir fest, dass sich in einem Kooperationsabschnitt durchaus mehrere Prozesse aufhalten können, wenn alle Prozesse lediglich lesend auf die Datenstruktur zugreifen. Solange kein Element in die Liste eingefügt oder aus der Liste entfernt wird, kann schließlich auch keine inkonsistente Datensicht entstehen. Während sich bei Abwesenheit eines schreibenden Prozesse beliebig viele lesende Prozesse im Kooperationsabschnitt aufhalten dürfen, so darf während ein Prozess schreibt, kein anderer Prozess (weder lesend, noch schreibend) im Kooperationsabschnitt sein. Im Kooperationsabschnitt dürfen sich also entweder genau ein Schreiber oder beliebige viele Leser befinden.
Betrachten wir nun einen Ringpuffer (siehe Abbildung 7.18) als Beispiel für eine kapazitätsbedingte Kooperationsproblematik.

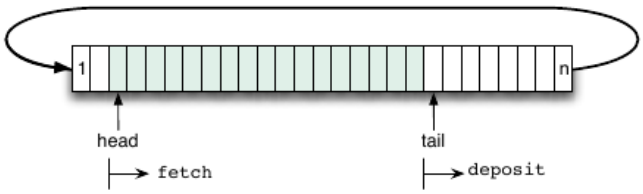


Abbildung 7.18: Ringpuffer

Der dargestellte Ringpuffer wird von mehreren Prozessen als gemeinsamer Pufferbereich genutzt. Prozesse können dort mittels `deposit(data)` Daten ablegen und mittels `fetch(data)` Daten abholen. Im dargestellten Ringpuffer können n Datenelemente abgelegt werden. Bereits belegte Datenfelder sind dabei hellblau gekennzeichnet. Der Head-Pointer zeigt entsprechend auf das erste belegte Datenfeld, der Tail-Pointer auf das erste nicht belegte Datenfeld. Neben der Sicherstellung des gegenseitigen Ausschlusses um eine inkonsisten-

te Datensicht zu vermeiden, muss hier zusätzlich beachtet werden, dass die Operation `deposit(data)` nur dann aufgerufen werden darf, wenn noch genügend Platz im Puffer vorhanden ist. Analog gilt für die Operation `fetch(data)`, dass sie nur aufgerufen werden darf, wenn der Puffer nicht leer ist.

7.3.1 Konzept: Semaphor

Das Semaphoren-Konzept wurde 1965 durch E. W. Dijkstra als Mechanismus für die Kooperation von Prozessen entwickelt. Ein Semaphor bzw. eine Zählsperre `S` ist eine Datenstruktur, die aus einer Zählvariable und zwei Operationen `P(S)` („P“ steht für niederländisch *probeer te verlagen*, zu deutsch: *versuchen, zu reduzieren*) und `V(S)` („V“ steht für niederländisch *vrijgave*, zu deutsch: *freigeben*) besteht. Die Zählvariable wird dabei so initialisiert, dass ihr Wert der Anzahl an Prozessen entspricht, die gleichzeitig auf die gemeinsamen Daten bzw. den zu sichernden Abschnitt zugreifen dürfen. Wenn ein Prozess einen durch einen Semaphor gesicherten Abschnitt betreten möchte, so ruft er die Operation `P(S)` auf. Für den Fall, dass die Zählvariable größer Null ist, wird die Zählvariable dekrementiert und der Prozess darf den Abschnitt betreten. Ist die Zählvariable allerdings kleiner gleich Null, wird der aufrufende Prozess blockiert. Diese Operation ist also eine erweitere Form der in Abschnitt 7.1.2 vorgestellten Operation `lock`. Wenn ein Prozess einen kritischen Abschnitt verlässt, so ruft er die Operation `V(S)`, durch welche der Zähler inkrementiert und ein evtl. blockierter Prozess deblockiert wird. Die Operation verhält sich demnach analog zur `unlock`-Operation.
Unter Unix (System 5) werden Semaphoren nicht als Einzelobjekte, sondern in Form von Arrays, die mehrere Semaphoren enthalten, zur Verfügung gestellt. Diese Arrays wiederum werden im System durch die Semaphore Table verwaltet. Um ein solches Array von Semaphoren anzulegen, kann der Systemruf `semget(key, nsems, semflg)` verwendet werden. Der Parameter `key` ist dabei eine Zahl, die die zu erzeugenden (bzw. bereits vorhandenen) Semaphoren identifiziert. Sie wird von den Prozessen vereinbart, die über die Semaphoren kommunizieren wollen. `nsems` entspricht der Anzahl an Semaphoren, die das Array enthalten soll. Über den Parameter `semflg` kann bspw. festgelegt werden, was passiert, wenn ein Semaphorenarray mit dem übergebenen `key` bereits existiert. `semget` liefert als Rückgabewert den Identifier des erzeugten Semaphorenarrays (also eine ganze Zahl) zurück. Das Bearbeiten von Semaphoren erfolgt mithilfe des Systemrufs `semop(semid,sops,nsops)`. Dabei entspricht `semid` dem Identifier eines Semaphorenarrays, welches im System existiert. `nsops` gibt an, wie viele Operationen auf dem angegebenen Semaphorenarray auszuführen sind, während `sops` die entsprechenden Operationen (und auf welchem der Semaphore im Array sie auszuführen sind) enthält. Der Systemruf ermöglicht also das gleichzeitige Bearbeiten aller Semaphore in einem Semaphorenarray.

Durch `semop` können die Semaphore um beliebige Werte inkrementiert und dekrementiert werden. An dieser Stelle stellt sich allerdings die Frage, was passiert, wenn ein Prozess terminiert, während er einen Semaphor hält. Die Lösung für dieses Problem ist eine „Undo Table“ als ergänzende Verwaltungsstruktur zur Semaphore Table. In der Undo Table werden für jeden Prozess die vorgenommenen Änderungen an betretenen Semaphoren protokolliert (Spiegelung). Bei Terminierung des Prozesses werden die betroffenen Semaphore um die protokollierten Spiegelwerte korrigiert.

Zusammenfassung

In diesem Kapitel haben Sie Konzepte zur Realisierung der Prozessinteraktion kennengelernt. Prozessinteraktion dient dem Datenaustausch zwischen Prozessen und basiert auf einer zeitlichen Abstimmung zwischen den interagierenden Prozessen – der Koordination. Basierend auf den erläuterten Koordinationskonzepten – kritischer Abschnitt und Signalisierung – haben wir die Interaktionsarten Kommunikation und Kooperation diskutiert. Kommunikation entspricht dabei dem expliziten Austausch von Nachrichten und kann mithilfe des Kanal-Konzepts implementiert werden. Die Kooperation entspricht hingegen dem konkurrierenden Zugriff auf gemeinsame Daten bzw. Ressourcen.

8 Konflikte

Sobald es in einem System mehrere nebenläufige Aktivitäten gibt, können diese miteinander in Konflikt[1] geraten. Dies gilt beispielsweise für Prozesse auf einem Rechner, die um Rechenzeit auf dem Prozessor konkurrieren, aber auch für Nutzer, die sich einen Rechner teilen oder Rechner in einem Netzwerk, die u. U. zeitgleich über eine Verbindung kommunizieren wollen. Konflikte können also immer dann entstehen, wenn mehrere Akteure (z. B. Prozesse, Nutzer, Netzwerkkarten) um bestimmte Betriebsmittel (z. B. Prozessor, Speicher, Bandbreite) konkurrieren, da Betriebsmittel knapp sind und ihre Benutzung exklusiv (d. h. zu einem Zeitpunkt durch genau einen Benutzer) erfolgt. Um Konflikte aufzulösen ist es deswegen u. U. sinnvoll, eine Betriebsmittelverwaltung einzuführen.

8.1 Betriebsmittelverwaltung

Die Verwaltung von Betriebsmitteln kann als allgemeines Konzept völlig losgelöst von Rechnern oder Rechnersystemen betrachtet werden, denn wo die gemeinsame Nutzung von Ressourcen organisiert werden muss, wird die Nutzung i. d. R. von Verwaltungsoperationen geklammert.

Abbildung 8.1: Klammerung der Nutzung eines Betriebsmittels durch Verwaltungsoperationen

In Abbildung 8.2 ist dieses Konzept sowohl allgemein (links) als auch am Beispiel einer Pkw-Vermietung (rechts) dargestellt. Zunächst melden mehrere Akteure den Bedarf an,

[1] In der Vorlesung werden die Konflikte im Rahmen der Koordination diskutiert. Es handelt sich aber um ein allgemeines Problem, weswegen im Skript ein eigenes Kapitel dazu gibt, dass auch die Hintergründe diskutiert.

das Betriebsmittel (im Beispiel: das Auto) verwenden zu wollen. Durch die Verwaltungsinstanz wird nun entschieden, welcher Akteur die Ressource als nächster nutzen darf. Im Anschluss an die Nutzung (im Beispiel: Auto fahren) wird die Ressource durch eine weitere Verwaltungsoperation wieder frei gegeben (im Beispiel: Auto zurückgeben).
Mit der Betriebsmittelverwaltung können unterschiedliche, teils gegensätzliche, Ziele verfolgt werden. So soll eine Betriebsmittelverwaltung beispielsweise korrekte Abläufe garantieren sowie Verklemmungen und Verhungern verhindern. Des Weiteren sind aber auch eine hohe Anzahl an nebenläufig möglichen Aktivitäten und eine hohe Auslastung des Betriebsmittels von Interesse. In der Praxis ist die Gestaltung einer Betriebsmittelverwaltung deshalb oft mit der Lösung komplexer Optimierungsprobleme verbunden.

8.1.1 Betriebsmittelprobleme im Alltag

In diesem Abschnitt betrachten wir einen Straßenengpass als einen im Alltag auftretenden Ressourcenkonflikt und leiten mögliche Lösungsstrategien aus diesem Konfliktszenario ab. Wir müssen also für den Fall, dass zwei Fahrzeuge gleichzeitig den Straßenengpass durchqueren wollen, eine Konfliktlösung finden.

Lösung 1: Zentrale Instanz Abbildung 8.2 zeigt den Straßenengpass sowie Ampeln und Schranken, die durch eine zentrale Instanz (Betriebsmittelverwalter) gesteuert werden, und somit vorgeben, welches der Fahrzeuge den Engpass zuerst durchfahren darf.

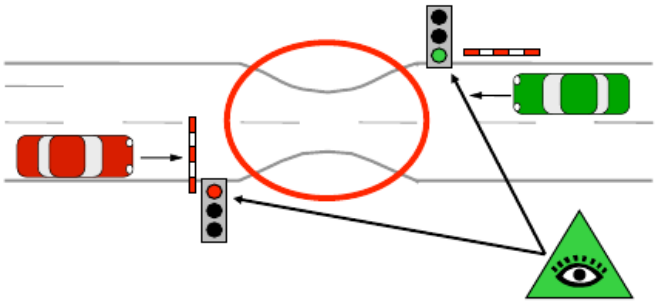


Abbildung 8.2: Zentrale Instanz zur Regelung der Vorfahrt

Allgemein betrachtet ist ein Betriebsmittelverwalter also eine zwischengeschaltete Instanz, die die exklusive Nutzung von Betriebsmitteln organisiert (vgl. 8.3).

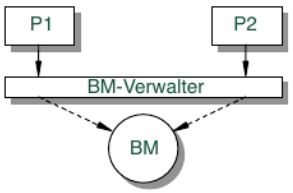


Abbildung 8.3: Betriebsmittelverwalter als Lösung für Ressourcenkonflikt

Beispiele für die Lösung von Betriebsmittelkonflikten in Rechnersystemen mittels zentraler Instanzen sind u.a. folgende:

- **Zwei-Phasen-Sperren in Datenbanksystemen**: Der Zugriff auf Daten (hier das Betriebsmittel) ist nur über den Scheduler (hier der Betriebsmittelverwalter) möglich.
- **Hauptspeicherverwaltung**: Zugriffe sind nur innerhalb der zugewiesenen Segmente möglich.
- **Monitor** (ein weiteres Synchronisationskonzept): Der Aufruf einer Entry-Prozedur ist nur möglich, wenn der Monitor frei ist.
- **Drucken**: Der Zugriff auf den Drucker als Betriebsmittel ist nicht unmittelbar möglich, sondern nur über spezielle Software, den „Treiber“, der als Verwalter fungiert.

Lösung 2: Verständigung der Teilnehmer Ein anderes Konzept zur Lösung von Betriebsmittelkonflikten ist die Verständigung der Konfliktteilnehmer. Bezogen auf die Konfliktsituation am Straßenengpass könnten die beteiligten Autofahrer beispielsweise per Handzeichen, Lichthupe oder „Berg- vor Talfahrt“ miteinander abstimmen, wer den Engpass zuerst befahren darf.

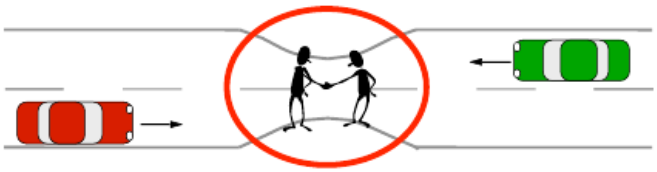


Abbildung 8.4: Verständigung der Verkehrsteilnehmer über die Vorfahrt

Das Prinzip der Verständigung als Konfliktlösung beruht im Allgemeinen auf Regeln, Verhandlung und Protokollen, mithilfe derer die beteiligten Teilnehmer den Konflikt untereinander (d. h. ohne Hilfe einer zentralen Instanz) lösen.

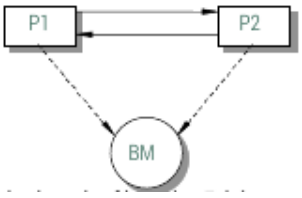


Abbildung 8.5: Verständigung als Lösung für Ressourcenkonflikt

In Rechnersystemen wird die Verständigung als Lösung für Konflikte mithilfe von Protokollen implementiert und kommt u.a. in folgenden Gebieten zum Einsatz:

- **Kritischer Abschnitt**: Die beteiligten Prozesse vereinbaren, den kritischen Abschnitt durch Nutzung von Sperren unter gegenseitigen Ausschluss zu stellen.

- **Globale Serialisierung in verteilten Systemen**: Die Rechner melden per Broadcast Bedarf an der Nutzung eines Betriebsmittels an. Im anschließenden Abstimmungsvorgang wird basierend auf logischer Zeit die Zugriffsreihenfolge auf das Betriebsmittel festgelegt.
- **Dezentrale Bus-Arbitrierung**: Der Zugriff auf den Bus als gemeinsames Übertragungsmedium muss geregelt werden. Die sendewilligen Komponenten (d. h. Komponenten, die einen „bus request“ abgesetzt haben) legen in einem speziellen Koordinationsprotokoll („Bus-Arbitrierung“) fest, wer als nächstes senden darf.

Lösung 3: Unkoordinierte Nutzung In diesem Fall ist keine Lösung, auch eine Lösung: Wir ergreifen keinerlei Maßnahmen, um auftretende Konflikte zu lösen und lassen es zur Kollision kommen, falls mehrere Akteure gleichzeitig auf ein Betriebsmittel zugreifen. Auch wenn sie im Straßenverkehr (vgl. Abbildung 8.6) definitiv keine Lösung ist, so kann die unkoordinierte Nutzung in Rechnersystemen durchaus ihren Zweck erfüllen.

Abbildung 8.6: Unkoordinierte Nutzung des Straßenengpasses führt u.U. zu Kollisionen

Ohne Abstimmung der Akteure über die Zugriffsreihenfolge besteht bei unkoordinierter Nutzung stets Kollisionsgefahr. BM Wenn es zu einer Kollision kommt, muss diese entsprechend in geeigneter Weise aufgelöst werden. Die unkoordinierte Nutzung sollte demnach nur dort zum Einsatz kommen, wo Kollisionen unwahrscheinlich sind bzw. selten auftreten und der durch eine Kollision entstandene Schaden reparabel ist.

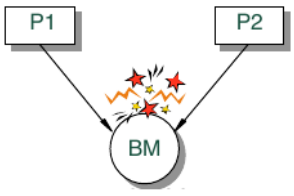


Abbildung 8.7: Unkoordinierte Nutzung von Betriebsmitteln

Da der Aufwand für seltene Kollisionsauflösung geringer sein kann als der permanente Aufwand für eine vorherige Abstimmung, kommt die unkoordinierte Nutzung beispielsweise in folgenden Anwendungen zum Einsatz:

- **Optimistische Synchronisation von Transaktionen (Validierung)**: Transaktionen setzen keine Sperren, sondern greifen einfach auf die benötigten Daten zu.

Diese Zugriffe werden in einem Log protokolliert und zum Abschluss der Transaktion („commit“) darauf überprüft, ob sie in Konflikt mit anderen Operationen standen (Validierung, Gültigkeitsprüfung). War das der Fall, so wird die Transaktion abgebrochen (und neu gestartet).

- **Kollisionsbehaftete Protokolle in lokalen Netzen** (z. B. Ethernet): Sobald eine Station im Netzwerk etwas senden möchte, hört sie kurz die Leitung ab. Ist die Leitung dabei frei, so sendet die Station. Sollten zwei Stationen gleichzeitig senden, kollidieren die Datenpakete und werden zerstört. Beide Stationen müssen ihre Daten dann erneut senden. Dies kann beispielsweise nach Ablauf einer zufällig bestimmten Zeit für jede der Stationen erfolgen, damit es möglichst unwahrscheinlich ist, dass die Stationen erneut gleichzeitig senden.

8.2 Verklemmung

Eine Verklemmung wird auch als Deadlock bezeichnet und beschreibt eine zyklische Wartesituation, in der jeder Akteur auf die Freigabe eines Betriebsmittels, welches exklusiv von einem anderen Akteur belegt ist, wartet. In einer solchen Situation ist für die Akteure kein Vorankommen mehr möglich. Im Alltag kann eine Verklemmung beispielsweise an einer Rechts-vor-Links-Kreuzung auftreten: Nämlich dann, wenn vier Fahrzeuge gleichzeitig die Kreuzung erreichen, weil dann jedes Fahrzeug entsprechend auf seinen rechten Nachbarn warten muss. Damit ergibt sich eine zyklische Wartesituation, die allein anhand der Vorfahrtsregelung nicht aufzulösen ist und dazu führt, dass keines der Fahrzeuge weiter fahren kann.

Abbildung 8.8: Deadlock bei Prozesssynchronisation mittels `signal()` und `wait()`

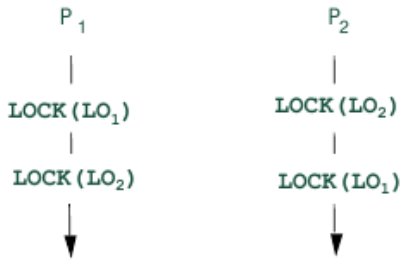


Abbildung 8.9: Deadlock bei Prozesssynchronisation mittels `lock()` und `unlock()`

Die Abbildungen 8.8 und 8.9 zeigen Beispiele aus der Prozesssynchronisation, die zu einem Deadlock führen. Die Prozesse blockieren sich in den Abbildungen gegenseitig, weil jeder Prozess auf die Freigabe einer Signalisierungs- bzw. Sperrvariable wartet, die nur durch den jeweils anderen Prozess erfolgen kann.

8.2.1 Problem: Speisende Philosophen

Das Philosophenproblem wurde als Fallbeispiel u.a. zur Erklärung der Verklemmungsgefahr von E.W. Dijkstra erdacht. Ausgangssituation dieses klassischen Koordinierungsproblems ist, dass fünf Philosophen an einem runden Tisch sitzen und stets entweder denken oder Spaghetti essen. Die Schwierigkeit ist nun, dass insgesamt nur fünf Gabeln vorhanden sind, jeder Philosoph um zu essen aber zwei Gabeln benötigt. Abbildung 8.10 illustriert diese Situation. Wenn sich ein Philosoph entscheidet zu essen, so ergreift er zuerst die eine Gabel und anschließend die andere. Die Gabeln werden also nie gleichzeitig vom Tisch genommen.

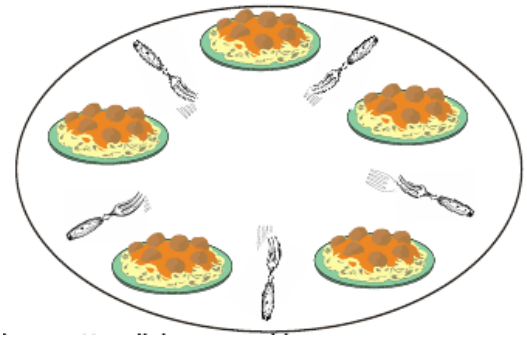


Abbildung 8.10: Aufbau des Philosophenproblems

Die Handlungsabfolge jedes einzelnen Philosophen sieht entsprechend wie folgt aus:

1. nachdenken
2. linke Gabel nehmen
3. rechte Gabel nehmen
4. essen
5. linke Gabel zurück legen
6. rechte Gabel zurück legen
7. gehe zu 1.

Da alle Philosophen nebenläufig diesem Algorithmus folgen, kann es dazu kommen, dass alle Philosophen die linke Gabel aufnehmen und anschließend (für immer) auf die rechte Gabel warten. Das passiert genau dann, wenn alle Philosophen im gleichen Moment auf die Idee kommen zu essen. Bedingt durch die so entstandene Verklemmung ist nun kein Fortschritt mehr möglich.

8.2.2 Wartegraph

Mithilfe von Wartegraphen können Wartesituationen formal dargestellt werden. Ein Wartegraph ist hier ein gerichteter Graph, dessen Knoten die Prozesse in einem System sind

und dessen Kanten den Wartebeziehungen zwischen diesen Prozessen entsprechen. Die Abbildungen 8.11 und 8.12 zeigen derartige Wartegraphen.

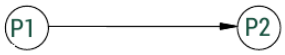


Abbildung 8.11: Wartegraph ohne zyklische Wartesituation

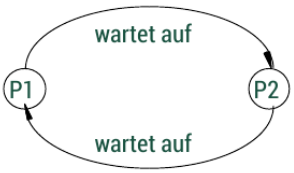


Abbildung 8.12: Wartegraph mit zyklischer Wartesituation

Abbildung 8.11 zeigt beispielsweise auf, dass Prozess 1 auf ein Ereignis (z. B. die Freigabe eines Betriebsmittels) wartet, welches Prozess 2 auslösen muss. Sobald Prozess 2 dieses Ereignis ausgelöst hat, ist die Wartesituation aufgelöst, d. h. die Kante im Wartegraph wird entfernt. Der Wartegraph in Abbildung 8.12 zeigt hingegen eine zyklische Wartesituation – Prozess 1 warte auf Prozess 2, während Prozess 2 auf Prozess 1 wartet. Enthält der Wartegraph einen solchen Zyklus, so liegt eine Verklemmung bzw. ein Deadlock vor.

8.2.3 Betriebsmittelgraph

In ähnlicher Art und Weise, wie mit Wartegraphen Wartesituationen abgebildet werden können, können mit Betriebsmittelgraphen Anforderungs- und Belegungssituationen formal abgebildet werden. Die Menge V der Knoten eines Betriebsmittelgraphen setzt sich aus der Menge P der Prozesse und der Menge BM der Betriebsmitteltypen in einem System zusammen. Im Betriebsmittelgraphen existieren des Weiteren zwei Arten von gerichteten Kanten:

1. Kante (p, b) von P-Knoten zu BM-Knoten: Prozess p fordert eine Einheit des Betriebsmitteltyps b an (Anforderungskante).
2. Kante (b, p) von BM-Knoten zu P-Knoten: Prozess p besitzt eine Einheit des Betriebsmitteltyps b (Belegungskante).

Abbildung 8.13 zeigt ein Beispiel für einen Betriebsmittelgraphen mit drei Prozessknoten $P1$, $P2$ und $P3$ sowie drei Betriebsmitteltypenknoten $B1$, $B2$ und $B3$.

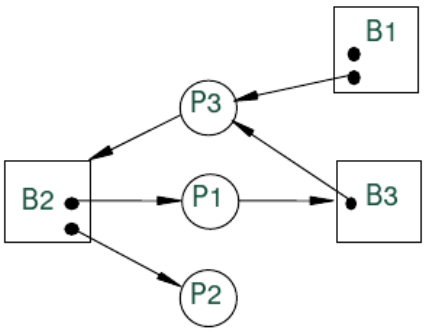


Abbildung 8.13: Betriebsmittelgraph mit je 3 Prozessen und Betriebsmitteltypen

Die schwarzen Punkte innerhalb der Rechtecke stehen für die verfügbaren Einheiten des entsprechenden Betriebsmitteltyps. Von Betriebsmitteltyp $B1$ stehen beispielsweise zwei Einheiten zur Verfügung, wovon eine bereits von Prozess $P3$ belegt ist. Von Betriebsmitteltyp $B2$ existieren insgesamt zwei Einheiten, welche im Beispiel bereits durch $P1$ und $P2$ belegt sind. Die Kante zwischen $P3$ und $B2$ zeigt entsprechend an, dass $P3$ eine Einheit dieses Betriebsmitteltyps anfordert. $P3$ muss im Beispiel also warten, bis mindestens einer der beiden anderen Prozesse eine Einheit von $B2$ freigegeben hat.
Ebenso wie im Wartegraphen, weist ein Zyklus im Betriebsmittelgraphen auf eine potentielle Verklemmungssituation hin. Allerdings ist ein Zyklus hier nur eine notwendige, aber keine hinreichende Bedingung für das Vorliegen einer Verklemmung, d. h. wenn der Betriebsmittelgraph einen Zyklus aufweist, heißt das nicht zwangsläufig, dass auch eine Verklemmung vorliegt (vgl. Abbildung 8.13 und Abbildung 8.14).
Für die sichere Erkennung einer Verklemmung anhand des Betriebsmittelgraphen gilt deswegen folgendes Theorem: *Eine Betriebsmittelsituation S ist verklemmt genau dann, wenn der dazugehörige Betriebsmittelgraph nicht vollständig reduzierbar ist.*
Um dieses Theorem anzuwenden, müssen wir verstehen, was „vollständig reduzierbar“ bedeutet. Jede Operation durch einen Prozess durchgeführte Operation (z. B. Anfordern, Belegen, Freigeben) bedeutet eine Veränderung des Betriebsmittelgraphen. Konkret: Das Hinzufügen oder Entfernen einer Kante. Ein Prozess kann nur dann eine Operation durchführen, wenn er nicht blockiert ist. Weiterhin kann ein Prozess dann beendet werden, wenn seine Gesamtanforderungen erfüllt sind. Wird ein Prozess beendet, so entspricht dies der Freigabe aller Betriebsmittel, die durch diesen Prozess belegt waren. Ist ein Prozess also weder isoliert noch blockiert, so reduziert er den Betriebsmittelgraphen durch Entfernen aller seiner Belegungskanten. Ein Betriebsmittelgraph heißt darauf aufbauend reduzierbar, wenn es einen Prozess gibt, der ihn reduzieren kann. Der BM-Graph heißt des Weiteren vollständig reduzierbar, wenn es eine Folge von Reduktionen gibt, die dazu führt, dass der Graph keine Kanten mehr enthält.
Abbildung 8.14 zeigt zur Verdeutlichung die vollständige Reduktion des Betriebsmittelgra-

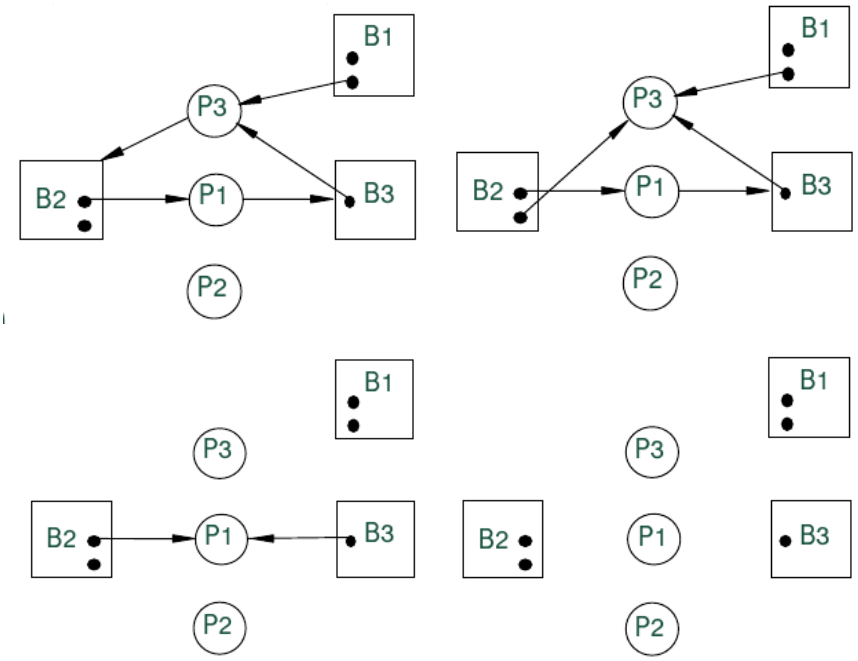


Abbildung 8.14: Vollständige Reduktion eines Betriebsmittelgraphen

phen aus Abbildung 8.13. Die gezeigte vollständige Reduktion des Betriebsmittelgraphen aus Abbildung 8.13, der einen Zyklus enthält, verdeutlicht auch den Sachverhalt, dass ein Zyklus im Betriebsmittelgraph nicht ausreichend ist, um eine Verklemmungssituation zu erkennen.
Allerdings gibt es den Spezialfall „Einexemplarbetriebsmittel“, in welchem ein Zyklus im Betriebsmittelgraph hinreichend ist, um eine Verklemmung zu erkennen. In diesem Fall gilt, dass von der Belegung/Anforderung eines Betriebsmitteltyps alle zugehörigen Einheiten betroffen sind. Damit lassen sich die Wartebeziehungen zwischen den Prozessen einfach feststellen und die Verklemmungsentdeckung reduziert sich zur Zyklenerkennung in Graphen.

8.2.4 Umgang mit Verklemmungen

Im Umgang mit Verklemmungen können verschiedene Vorbeugungs-, Vermeidungs-, Entdeckungs- und Auflösungsmaßnahmen ergriffen werden. Werden Verklemmungen nicht a priori vermieden, so besteht jede Verklemmungsauflösung darin, aus dem Wartezyklus auszubrechen. Ist hierzu ein (geordneter) Betriebsmittelentzug nicht möglich, so bleibt nur der Abbruch mindestens eines Prozesses, um wieder Fortschritt im System zu ermöglichen. Dabei stellt sich dann allerdings die Frage, welcher Prozess abgebrochen werden soll. Kriterien für die Auswahl des zwangsläufig abzubrechenden Prozesses können dabei u.a. folgende sein:

- Größe der Anforderung
- Umfang der belegten Betriebsmittel
- Dringlichkeit
- Benutzer-/Systemprozess
- Aufwand des Abbruchs
- Verlorengegangene Arbeit
- Restbedienzeit

In der Theorie können Verklemmungssituationen durchaus von vornherein verhindert werden, beispielsweise dadurch, dass auf die Betriebsmittel nur in einer strikten Ordnung zugegriffen wird. Bezogen auf das Problem der Speisenden Philosophen ist es zur Deadlock-Vermeidung beispielsweise ausreichend, wenn genau einer der Philosophen die Gabeln in umgekehrter Reihenfolge (d. h. zuerst die rechte, dann die linke) aufnimmt. In der Praxis sind derartige Ansätze allerdings oftmals aufwändig umzusetzen oder nur eingeschränkt verwendbar.

Zusammenfassung

In diesem Kapitel haben wir die Entstehung und Behandlung von Konfliktsituationen in Systemen mit mehreren nebenläufigen Aktivitäten betrachtet. Dabei haben wir drei Ansätze zur Verwaltung von Betriebsmitteln kennengelernt und diskutiert: die Einführung einer zentralen Instanz, die Verständigung der Akteure und die unkoordinierte Nutzung. Darüber hinaus haben wir die Gefahr von Verklemmungen in derartigen Systemen u.a. am Beispiel der Speisenden Philosophen betrachtet. Verklemmungen sind zyklische Wartesituationen, nach deren Eintreten kein Fortschritt für die Akteure im System mehr möglich ist. In diesem Kontext wurden des Weiteren Warte- und Betriebsmittelgraphen vorgestellt und erläutert, wie mithilfe dieser Graphen Verklemmungen erkannt werden können.

9 Über die Grenze: Rechner im Netzwerk

Bisher haben wir die Funktionsweise und das Zusammenspiel von Soft- und Hardware eines einzelnen Rechners betrachtet, um lokale Aufgaben zu lösen. In Kapitel 7 haben wir in diesem Rahmen die lokale Kommunikation zwischen Prozessen betrachtet. Da Aufgaben wie die Übertragung von Daten oder die Nutzung entfernter Rechenkapazitäten auch die Kommunikation zu entfernten Rechnern erfordert, betrachten wir in diesem Kapitel die Verbindung von Rechnern durch Netzwerke.

9.1 Netze

9.1.1 Kriterien für Netzwerke

Neben der Zuverlässigkeit bzw. der Ausfallsicherheit eines Netzwerkes sowie seiner räumlichen Ausdehnung sind die Geschwindigkeitsparameter Latenz und Durchsatz zwei der wichtigsten Kriterien zur Beurteilung der Leistungsfähigkeit von Kommunikationsnetzwerken. Die Latenz (auch „Verzögerung“, engl. „latency“) ist dabei die Zeit, die eine leere Nachricht braucht, um vom Sender zum Empfänger zu gelangen bzw. die Zeit zur Etablierung einer Verbindung. Der Durchsatz (auch Übertragungsrate, „transfer rate“) ist die Geschwindigkeit, mit der Daten übertragen werden, sobald eine Verbindung etabliert ist. Die Zeit, die für die Übertragung einer Nachricht benötigt wird, ergibt sich demnach zu:

$$Nachrichten\ddot{u}bertragungszeit = Latenz + \frac{Nachrichtenl\ddot{a}nge}{Durchsatz}$$

9.1.2 Topologien

Mithilfe der Netzwerktopologie wird die Struktur von Rechnernetzen beschrieben. Topologien sind Graphen, wobei die Knoten den Rechnern und die Kanten den Verbindungen zwischen den Rechnern entsprechen. Abbildung 9.1 zeigt typische Netzwerktopologien.

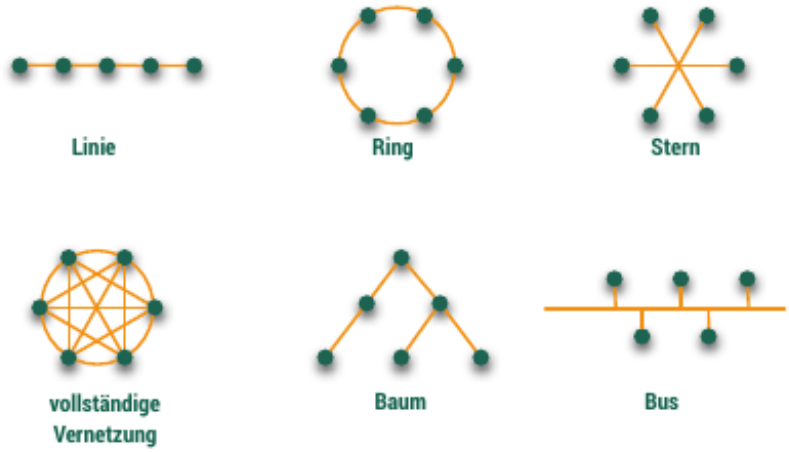


Abbildung 9.1: Netzwerktopologien

Auch Vermischungen dieser Topologien sowie irreguläre Strukturen sind denkbar und üblich. Die Wahl der Topologie wirkt sich direkt auf Ausfallsicherheit, Latenz und Durchsatz eines Netzwerkes aus. Haben die Netzwerkknoten beispielsweise viele Kanten, wie z.B. in der vollständigen Vernetzung, so ist die Ausfallsicherheit besonders hoch, weil viele alternative Nachrichtenpfade existieren.

9.1.3 Verbindungsarten

Beim Datentransport wird verbindungsorientierter und verbindungsloser Kommunikation unterschieden. Bei der verbindungsorientierten Kommunikation wird zwischen Sender und Empfänger ein virtueller Kanal (eine Verbindung) aufgebaut, der anschließend zur Übermittlung der Datenströme genutzt wird. Diese Vorgehensweise ist vergleichbar mit dem Telefonnetz: Zwischen Anrufendem und Angerufenem besteht für die Dauer des Telefonats eine feste Verbindung. Das Gespräch entspricht dabei den übertragenen Daten. Bei der verbindungslosen Kommunikation wird hingegen jede einzelne Nachricht (Datagramm) zu einem bestimmten Ziel geschickt. Schickt ein Sender also mehrere Nachrichten zu ein und demselben Ziel, so können sich die Wege dieser Nachrichten unterscheiden, weil keine feste Verbindung zwischen Sender und Empfänger etabliert wird. Diese Verbindungsart verhält sich also analog zum Versenden von Paketen mit der Post.

9.2 Schichtenmodelle

Architekturen, beispielsweise für die Kommunikation in Netzwerken, lassen sich in Hierarchieebenen untergliedern, die unterschiedliche Aufgaben erfüllen. Diese Hierarchieebenen werden auch als Schichten bezeichnet. Dabei hat jede Schicht einen bestimmten Zweck bzw. eine bestimmte Funktion. Jede der Schichten bietet den jeweils höheren Schichten

Dienste an, auf welche über eine Schnittstelle zugegriffen werden kann. Auf diese Art und Weise wird von den Implementierungsdetails abstrahiert, d. h. die konkrete Implementierung eines Dienstes ist für die Schichten, die den Dienst nutzen, irrelevant. Während eine Schicht Dienste anbietet, nutzt sie selbst Dienste tieferer Schichten, um ihre Funktion zu erfüllen. Schichtenmodelle dienen also durch Abstraktion der Reduzierung der Komplexität des Gesamtsystems. Abbildung 9.2 zeigt beispielhaft ein Vierschichtenmodell.

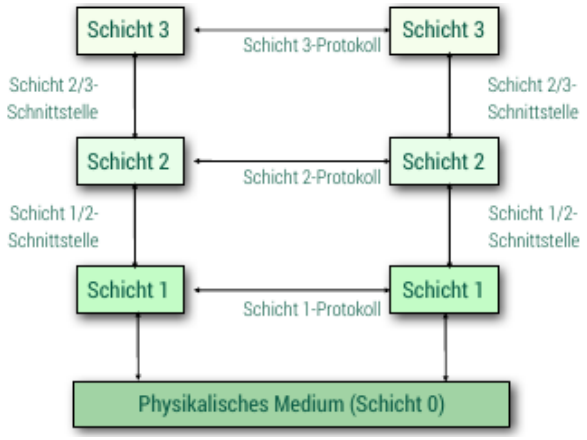


Abbildung 9.2: Schichtenmodell

Neben den vier Schichten und den zugehörigen Schnittstellen sind in der Abbildung auch „Protokolle“ erwähnt. Was es damit auf sich hat, betrachten wir im nachfolgenden Abschnitt.

9.2.1 Protokoll

Ein Protokoll legt die Regeln und Konventionen fest, wie die Schicht n der Maschine A mit Schicht n der Maschine B kommuniziert. Beim Senden leitet jede Schicht die Daten an die unterliegende Schicht weiter, bis die physikalische Schicht erreicht ist. Dabei können die Daten in jeder Schicht transformiert (z.B. aufgeteilt) sowie zusätzliche Informationen (z.B. Header, Trailer) hinzugefügt werden. Der Empfang erfolgt beginnend bei der physikalischen Schicht entsprechend umgekehrt, d. h. durch rückgängig machen der Transformationen und entfernen zusätzlicher Informationen. Eine Hierarchie von Protokollen wird auch als „Protocol Stack“ bezeichnet. Einen durch einen Protocol Stack definierten beispielhaften Informationsfluss durch ein Schichtenmodell zeigt Abbildung 9.3.

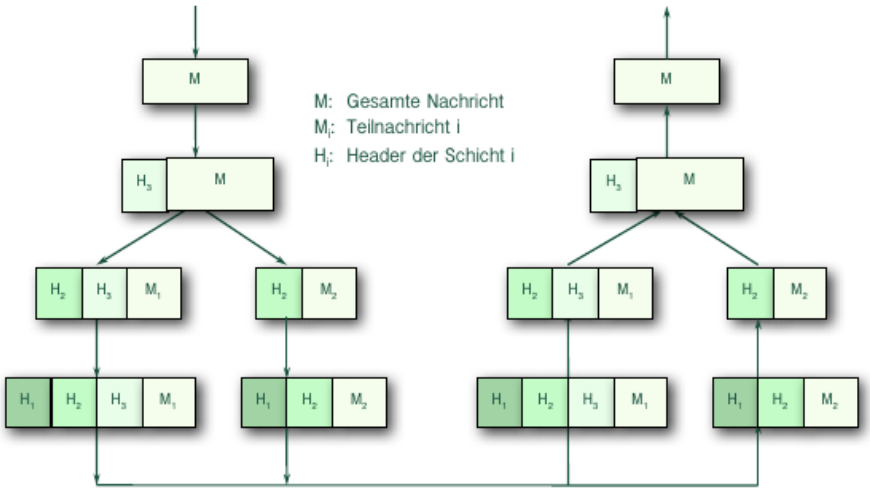


Abbildung 9.3: Informationsfluss

9.2.2 TCP/IP-Referenzmodell

Das TCP/IP-Referenzmodell ist ein Schichtenmodell, welches die Basis für die Kommunikation zwischen Knoten im Internet bildet. Abbildung 9.4 zeigt die vier Schichten des Modells.

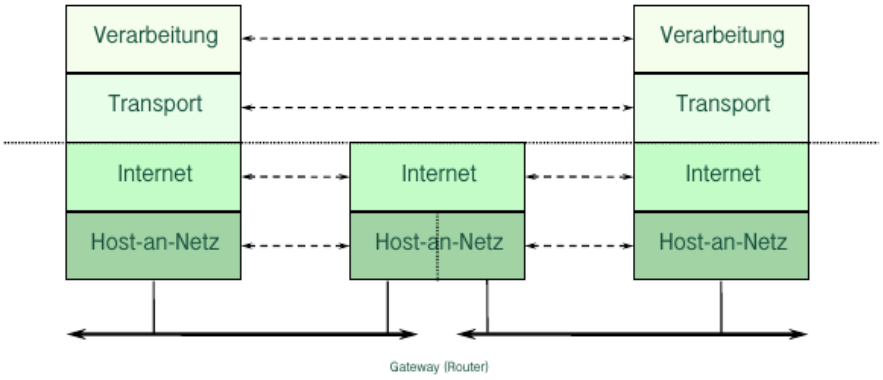


Abbildung 9.4: TCP/IP-Referenzmodell

Die Host-an-Netz- oder auch Netzzugangsschicht ist für das Senden, Weiterleiten und Empfangen einzelner Pakete zwischen direkt miteinander kommunizierenden Knoten zuständig.
Durch die Internetschicht wird die Ende-zu-Ende-Kommunikation zwischen Netzwerkknoten organisiert. Auf dieser Schicht wird zu einem empfangenen Paket das nächste Zwischenziel ermittelt und das Paket entsprechend weitergeleitet (Routing).
Die Transportschicht ist die Basis für die Kommunikation zwischen Anwendungen. Die beiden wohl bekanntesten Protokolle, die in diese Schicht einzuordnen sind, sind TCP und UDP. TCP ermöglicht dabei die zuverlässige, verbindungsorientierte Kommunikation

auf Basis von Byteströmen. UDP realisiert hingegen die unzuverlässige, verbindungslose Kommunikation auf Basis von Nutzdatenpaketen.
Die Verarbeitungsschicht ist die höchste Abstraktionsebene und stellt Anwendungsprotokolle wie HTTP, FTP und SMTP zur Verfügung.
Abbildung 9.5 zeigt ergänzend die Einordnung verschiedener Protokolle der Internetprotokollfamilie in die einzelnen Schichten des TCP/IP-Referenzmodells.

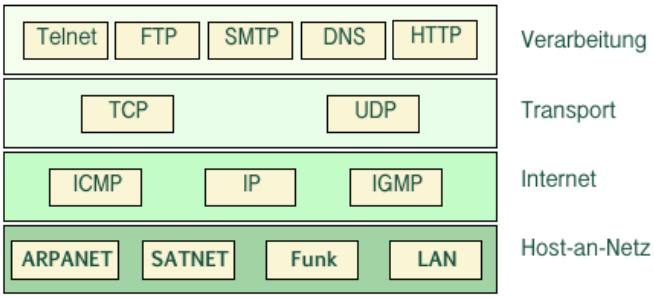


Abbildung 9.5: Einordnung von Protokollen in das TCP/IP-Referenzmodell

9.2.3 OSI-Referenzmodell

Das OSI-Referenzmodell ist ebenfalls ein Schichtenmdoell für Netzwerkprotokolle. Dieses Schichtenmodell besteht aus den sieben in Abbildung 9.6 gezeigten Schichten.

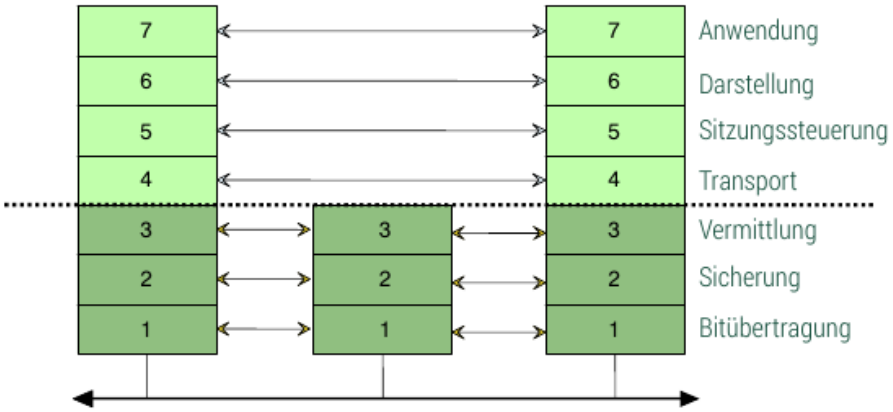


Abbildung 9.6: OSI-Referenzmodell

Nachfolgend beschreiben wir die Aufgaben der einzelnen Schichten:

1. **Bitübertragung (engl.: Physical)**: Übertragung ungesicherter Bitströme über den Kommunikationskanal
2. **Sicherung (engl.: Data Link)**: Sicherung der zu übertragenden Daten durch Aufteilung in Rahmen, Checksummen und Bestätigungen; Flusskontrolle
3. **Vermittlung (engl.: Network)**: Annehmen und Zustellen einzelner Pakete

4. **Transport (engl.: Transport)**: Entgegennehmen der Daten, Aufteilung in Pakete für die Vermittlungsschicht und Abliefern der vollständigen und korrekten Daten
5. **Sitzungssteuerung (engl.: Session)**: Synchronisation der Kommunikationspartner sowie Checkpointing bei langen Übertragungen
6. **Darstellung (engl.: Presentation)**: Festlegung von Syntax und Semantik der zu übertragenden Daten (Datenrepräsentation)
7. **Anwendung (engl.: Application)**: Angebot von Applikationsprotokollen wie HTTP, FTP und SMTP

9.2.4 TCP/IP- vs. OSI-Referenzmodell

Nachdem wir die Schichtenmodelle TCP/IP und OSI näher betrachtet haben, stellen wir in diesem Abschnitt einen Vergleich beider Modelle an. Abbildung 9.7 ordnet dazu die Schichten beider Modelle einander zu.

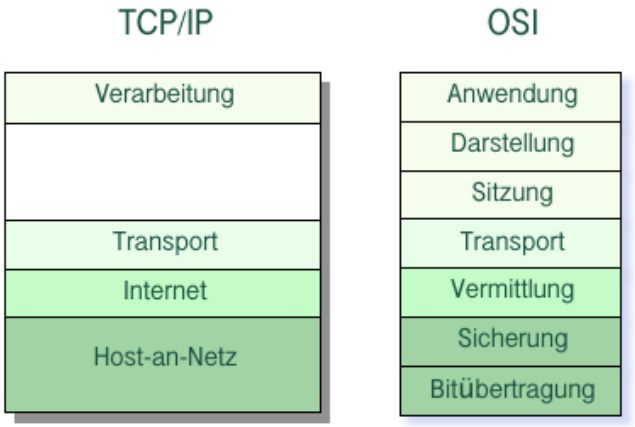


Abbildung 9.7: Zuordnung der Schichten von TCP/IP- und OSI-Referenzmodell

In Tabelle 9.1 ist eine Gegenüberstellung der Vor- und Nachteile beider Modelle zu finden.

TCP/IP	OSI
☺„Bottom-up“-Entwicklung getrieben aus der Praxis	☹Zähe „Top-down“-Entwicklung mit wenig Bezug zur Praxis
☺Einfachheit statt Universalität	☺Klar strukturiert und konzeptionell umfassend
☺Nur verbindungsloses Vermittlungsprotokoll	☹Ineffiziente Verarbeitung
☹Schwächen im Modell und in der Ebenenstruktur	☹Aufwendige, komplexe Implementierung
☹Keine klare Trennung zwischen Dienst, Schnittstelle und Protokoll	☹Redundante Funktionen
	☹Schlechtes Timing der Standardisierung

Tabelle 9.1: Vergleich der Referenzmodelle für Netzwerkprotokolle

Während das OSI-Referenzmodell sehr gut geeignet ist, um Netzwerke und Protokolle zu erörtern, hat sich in der Praxis das einfacher zu implementierende und zudem ältere TCP/IP-Referenzmodell durchgesetzt.

9.2.5 Fallbeispiel für Protokoll-Stack-Abarbeitung

In diesem Abschnitt betrachten wir anhand des Mausklicks auf einen Link die Abarbeitung des Protokoll-Stacks, um das Zusammenspiel der Protokollschichten zu verdeutlichen. Hierfür betrachten wir für die entsprechenden Schichten eine Auswahl der in der Praxis mitwirkenden Protokolle, wenn wir mithilfe eines Browsers (Firefox, Internet Explorer, Opera, etc.) ein auf einem entfernten Rechner liegendes HTML-Dokument aufrufen:

Anwendungsschicht

Das erste Protokoll der Anwendungsschicht ist das Hypertext Transfer Protocol (HTTP), welches das gemeinsame Protokoll (Anwendungsprotokoll) zwischen dem Webbrowser und dem Webserver (Programm, das z.B. HTML-Seiten zur Verfügung stellt.) ist. HTTP enthält Befehle wie `GET`, `PUT`, `HEADER` und `OPTIONS`. Mithilfe von `GET` fordert der Browser vom Server bestimmte Daten (z.B. eine HTML-Seite) an.
Mithilfe des Domain Name Service (DNS) werden hierzu für Menschen verständliche Domainnamen, wie `http://osg.informatik.tu-chemnitz.de`, in IP-Adressen konvertiert, um den entsprechenden Server anzusprechen.
Eine IP-Adresse ist eine internetweit eindeutige Gruppe von Nummern, die – ebenso wie Domainnamen – durch das Network Information Centre (NIC) vergeben wird. Mittlerweile existieren zwei Varianten von IP-Adressen: IPv4 und IPv6. IP-Adressen nach IPv4 haben eine Länge von 32 Bit und bestehen aus vier, durch Punkt getrennten 8-Bit-Dezimalzahlen, beispielsweise 134.109.193.2. IP-Adressen nach IPv6 sind hingegen 128 Bit

lang und bestehen aus acht, durch Doppelpunkt getrennten 16-Bit-Hexadezimalzahlen, beispielsweise 2001:0db8:85a3:08d3:1319:8a2e:0370:7344. Die ermittelte IP-Adresse wird nun in den unterliegenden Protokollschichten verwendet.

Transportschicht

Mit der korrekten IP-Adresse wendet sich der Browser an die Transportschicht, wo für den Transport die beiden Protokolle UDP und TCP (vgl. 9.2.2) zur Verfügung stehen. TCP übernimmt an dieser Stelle den Aufbau der virtuellen Verbindung zum Server, formt IP-Pakete (Datagramme), überprüft auf Fehler und nimmt ggf. eine Fehlerkorrektur (erneutes Senden) vor.

Vermittlungsschicht

Das Internet Protokoll (IP) gehört zur Vermittlungsschicht und ist in erster Linie für den Transport von Daten zwischen verschiedenen Netzwerken zuständig. IP zerlegt zum einen Datagramme, um die maximal zulässige Paketgröße eines Netzwerks zu erreichen und findet zum anderen einen Weg zum Ziel (Routing). Dabei greift das Protokoll auf eine Reihe weiterer Protokolle zurück, die auch von Netzwerk zu Netzwerk verschieden sein können.
Das IP-Routing erfolgt in zwei Schritten:

1. Lesen der Zieladresse eines Pakets
2. Network point of attachment (NPA) Adresse des Zielrechners oder nächsten Gateways anhand von Routingtabellen ermitteln

Im Routing wird das Internet als eine Menge von autonomen Systemen angesehen, die an das core backbone network (siehe Abbildung 9.8) angeschlossen sind.

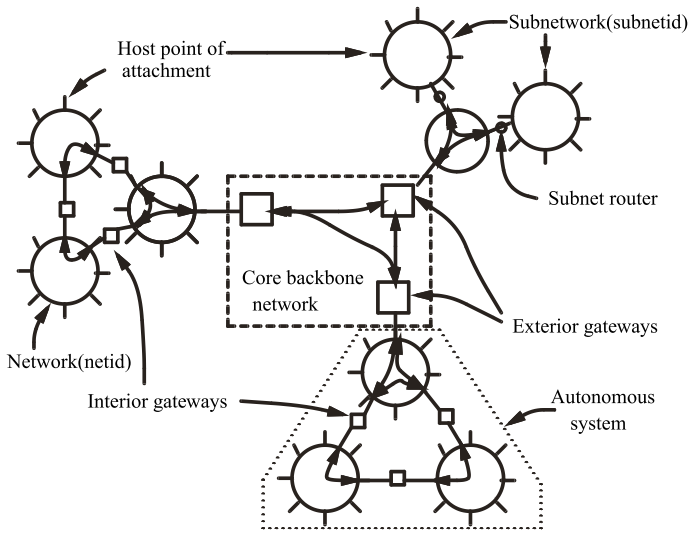


Abbildung 9.8: Internet: core backbone network und autonome Systeme

Innerhalb eines autonomen Systemes wird ein interior gateway protocol (IGP) benutzt, außerhalb das exterior gateway protocol (EGP). Verbreitete IGPs sind RIP (routing information protocol) und hello. Für die Adressauflösung nutzt IP das ARP (address resolution protocol).

Host-an-Netz-Schicht

Beim Netzzugang wird zwischen dem logischen und dem physischen Zugriff unterschieden. Der logische Medienzugriff (medium access control; MAC) wird durch MAC-Verfahren wie den Token Ring oder CSMA/CD (carrier sensing, multiple access with collision detection) geregelt.
Innerhalb von Netzwerken können verschiedene Medien zum Transport genutzt werden. Typisch sind beispielsweise elektromagnetische Wellen (vgl. Abbildung 9.9). Diese breiten sich wiederum innerhalb physikalischer Medien aus, auf welche schließlich der physische Zugriff erfolgt.

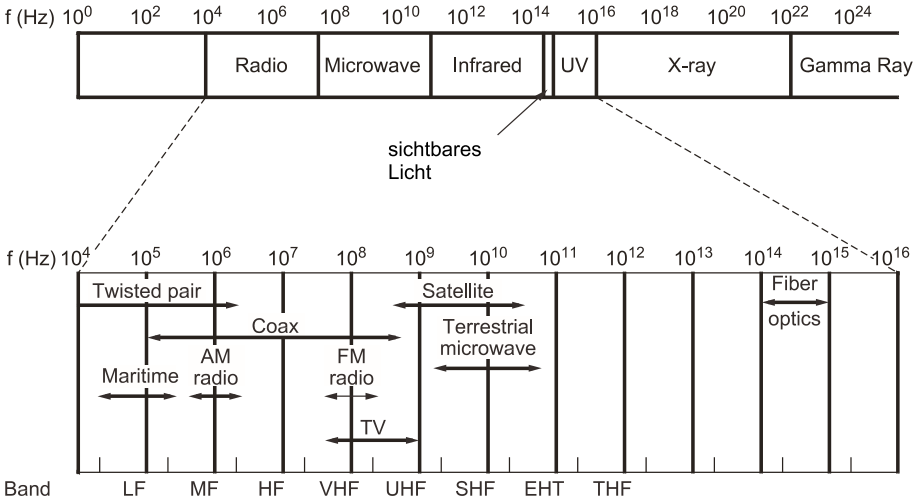


Abbildung 9.9: Elektromagnetisches Spektrum

Ziel erreicht

Ist ein Paket beim Empfänger angelangt, wird der in den vorherigen Abschnitten beschriebene Protokollstack rückwärts durchlaufen:

1. Die unteren Schichten lesen die Pakete vom Netz und liefern sie an das IP-Protokoll
2. IP setzt die Pakete zu Datagrammen zusammen und reicht sie an TCP
3. TCP überprüft auf Fehler, behebt sie (evtl. Kommunikation mit dem ursprünglichen Sender), setzt die Datagramme zu http-Nachrichten zusammen, und liefert diese an den WWW-Server aus
4. Webserver versteht die http-Nachricht, sucht das entsprechende Dokument und sendet es an den Browser zurück ...

5. ... und das Spiel beginnt von vorn.

Zusammenfassung

In diesem Kapitel haben wir die Verknüpfung von Rechnern durch Netzwerke erörtert. Hierzu haben wir Eigenschaften (Latenz und Durchsatz) und Topologien (Baum, Stern, Bus, etc.) von Netzwerken betrachtet. Darauf aufbauend haben wir uns mit Schichtenmodellen im Allgemeinen sowie dem TCP/IP- und OSI-Referenzmodell im Speziellen auseinandergesetzt. Abschließend haben wir uns das Zusammenspiel der Protokolle im TCP/IP-Protokoll-Stack anhand eines praktischen Beispiels angeschaut.

Literatur

[BDM05] Bernd Becker, Rolf Drechsler und Paul Molitor. *Technische Informatik – Eine Einführung*. Pearson Studium, 2005.

[Dud07] Dudenredaktion, Hrsg. *Duden – Das Herkunftswörterbuch*. Dudenverlag, 2007.

[EAS14] Martin Elstner, Jörg Axthelm und Alexander Schiller. *Süßeste Rechenmaschine der Welt – Chemischer Computer nutzt fluoreszierende Zuckerlösungen für Rechenoperationen*. http://www.scinexx.de/wissen-aktuell-17702-2014-06-24.html. [Stand: 15.04.2016]. 2014.

[GS13] Heinz-Peter Gumm und Manfred Sommer. *Einführung in die Informatik*. Zehnte. Oldenbourg, 2013.

[Hof14] Dirk W. Hoffmann. *Grundlagen der Technischen Informatik*. Vierte. Carl Hanser Verlag, 2014.

[PH05] David A. Patterson und John L. Hennessy. *Rechnerorganisation und -entwurf – Die Hardware/Software-Schnittstelle*. Dritte. Spektrum Akademischer Verlag, 2005.

[PP01] Yale N. Patt und Sanjay J. Patel. *Introduction to Computing Systems – From Bits and Bytes to C and Beyond*. McGraw-Hill Higher Education, 2001.