VL Digitale Rechenanlagen

Insitut für Computerwissenschaften
Universität Salzburg
WS 01/02

Vorlesungsunterlagen
Version 0.2

Helmut A. MAYER
Salzburg, Herbst 2001

Über diese Unterlagen

Die vorliegenden Vorlesungsunterlagen sind eine **Ergänzung** zur Vorlesung, decken aber sicher nicht alle in der VL aufgeworfenen Fragen ab. Die Unterlagen sollen Sie auch motivieren, ihr Wissen durch das Lesen von unterstützender und weiterführender Literatur zu vertiefen. Der Text ist mit etlichen Fragen und kleinen Aufgaben versehen, deren Lösung mit Sicherheit zu einem tieferen Verständnis des Stoffes beitragen.

Der Prüfungsstoff zur Vorlesung wird durch diese Unterlagen definiert. Dies bezieht sich nur auf die speziellen Inhalte, aber nicht auf einzelne Prüfungsfragen, d.h. es muss nicht jede Prüfungsfrage explizit im Text vorkommen oder beantwortet sein.

Ich bin für jegliche Hinweise dankbar, die die Qualität dieser Unterlagen heben, insbesondere würde ich Sie ersuchen, mich über fehlerhafte oder missverständliche Textstellen zu informieren (am besten per E–Mail: `helmut@cosy.sbg.ac.at`) (die konsequente Ersetzung von "ß" mit "ss" ist als Beitrag zu einer möglichen Reform der Rechtschreibreform gedacht..;-)

Ich hoffe, dass diese Unterlagen eine sinnvolle Hilfe für Sie sein werden, möchte aber auch bei aller Ernsthaftigkeit darauf hinweisen, dass Spass und Freude an Ihrem Studium ein wesentlicher Motivationsfaktor während Ihrer gesamten Studienzeit sein sollten. Entgegen der landläufigen Meinung verlangt nämlich die wissenschaftliche Beschäftigung mit Informatik und Computerwissenschaften neben solider technischer Grundausbildung ein grosses Mass an Kreativität, die Sie sich erhalten und auch ausbauen sollten. In diesem Sinne

Viel Spass,

Helmut Mayer

Inhaltsverzeichnis

Kapitel 1

Codierung und Informationstheorie

Der Begriff *Information* steht im Zentrum der *Informatik*, wie schon die Begriffsbildung zeigt, die auf die Beschäftigung mit Information und deren Verarbeitung mit wissenschaftlichen und technischen Methoden verweist. Daher wird uns auch in diesem Kapitel die Frage nach der Definition, Darstellung und Messbarkeit von Information beschäftigen. Was aber ist nun eigentlich Information?

Information wird durch jene Signale repräsentiert, die zwei (oder mehrere) Systeme miteinander austauschen. Information benötigt also eine Repräsentation, wie sie zum Beispiel die Schrift darstellt, die eine der grössten Kulturleistungen des Menschen ist, und einen ganz wesentlichen Anteil an der exponentiellen technischen Entwicklung der letzten Jahrhunderte beigetragen hat. Ganz allgemeine Beispiele für Systeme und von diesen verwendete Signale sind in Tabelle 1.1 angeführt.

Systeme	Signale
Massen	Schwerkraft
Sonnen	Licht
Pflanzen	Farbe, Geruch
Tiere	Laute, Gestik
Mensch	Sprache, Schrift
Computer	Spannung, Licht

Tabelle 1.1: Allgemeine Beispiele für Informationen und deren Repräsentation (Medium).

Eine etwas technischere Aufstellung von Systemen und den zum Informa-

tionsaustausch verwendeten Signalen gibt Tabelle 1.2.

Technische Systeme	Signale
Feuerstelle	Rauchzeichen
Telegraph	Morsezeichen
Rundfunk	Amplitudenmodulation
Compact Disc	Pulse Code Modulation
Digitale Videos	MPEG

Tabelle 1.2: Technische Beispiele für Informationen und deren Repräsentation (Codierung).

Der wesentliche Unterschied zwischen Tabelle 1.1 und Tabelle 1.2 besteht in der unterschiedlichen Kategorisierung der Signale. Während in Tabelle 1.1 rein das physikalische Medium, das zur Übermittlung der Signale notwendig ist, angeführt ist, ist in Tabelle 1.2 die *Codierung* der Signale in einem Medium angegeben. Die Codierung dient einerseits dazu komplexere Nachrichten versenden zu können, andererseits ermöglicht sie eine Anpassung des Nachrichtentyps an das verwendete Medium (es wäre wenig vorteilhaft digitale Videos mit natürlicher Sprache zu beschreiben (= codieren) und die sprachliche Beschreibung zu übermitteln).

Am Beispiel der Informationsübertragung mittels Morsezeichen ist leicht zu sehen, dass Codierung nichts anderes als die "Übersetzung" einer speziellen Repräsentation (Schrift) in eine andere (Morsezeichen) ist (Tabelle 1.3).

Buchstabe	Morsezeichen
A	.-
E	.
K	-.-
T	-
3	...- -
SOS	... - - - ...

Tabelle 1.3: Beispiele der Morsecodierung.

Im Rahmen dieser Vorlesung werden wir uns fast ausschliesslich mit *Binärcodierungen* beschäftigen, die jegliche Information auf eine Kette aus den Zeichen **0** und **1** abbildet, die in digitalen Rechenanlagen wiederum als "Spannung/Strom aus" und "Spannung/Strom ein" repräsentiert und verarbeitet werden. Hier soll auch erwähnt werden, dass alles Leben auf unserer

Erde im wesentlichen durch die *quaternäre Codierung* mit den Zeichen **A**, **T**, **C**, **G** repräsentiert wird.

1.1 Codes und Codierung

Unsere allgemeinen Überlegungen sollen nun etwas formaler zusammengefasst werden (Broy, 1993).

Definition 1.1.1 *Eine (endliche) Menge A von Zeichen heisst* Zeichenvorrat. *Ein linear geordneter Zeichenvorrat A heisst* Alphabet.

Ein elementarer Zeichenvorrat ist die Menge $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ (auch $\mathbb{B} = \{\mathbf{L}, \mathbf{O}\}$) [1] Ein Element der Menge $\mathbb{B}$ nennen wir *Binärzeichen* oder *Binary Digit* (Bit). Die lineare Ordnung wird durch $\mathbf{0} \leq \mathbf{1}$ definiert.

Definition 1.1.2 *Die Menge A^* der endlichen Zeichenfolgen über einem Zeichenvorrat A nennen wir die Menge der* Wörter *über A.*

Die Menge $\mathbb{B}^* = \{0, 1\}^*$ bezeichnen wir als Menge der *Binärwörter*. Die Elemente der Menge $\mathbb{B}^n$ nennen wir *n–Bit–Wörter*, die oft selbst wiederum einen Zeichenvorrat bilden können.

Definition 1.1.3 *Eine Abbildung zwischen zwei Zeichenvorräten A und B $c : A \to B$ bzw. eine Abbildung von Wörtern über diesen Zeichenvorräten $c : A^* \to B^*$ nennen wir* Code *oder* Codierung.

Für spezielle Codelängen (Anzahl der Zeichen) $m, n \in \mathbb{N}$ ergibt sich (als Beispiel) die Abbildung $c : A^m \to B^n$, wobei es durchaus üblich ist, dass nicht alle möglichen Wörter (wieviele sind das?) aus A^m und B^n codiert werden. Ist in unserem Beispiel $m = 1$ und n beliebig, dann nennt man diese spezielle Codierung auch *Chiffrierung*.

Da ein sinnvoller Code auch umkehrbar (*decodierbar*) sein sollte, müssen wir von unserer Codierung $c : A \to B$ fordern, dass sie eine *injektive* Abbildung darstellt. Es müssen also verschiedene Zeichen (Wörter) aus A auf verschiedene Zeichen (Wörter) aus B abgebildet werden. Damit ist die Umkehrabbildung

$$d : \{c(a) \mid a \in A\} \to A$$

eindeutig und es gilt dann für alle Zeichen $a \in A$

$$d(c(a)) = a.$$

Diese Umkehrabbildung nennen wir *Decodierung*.

[1] Anstatt $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ werden wir in Hinkunft die einfachere Schreibweise $\mathbb{B} = \{0, 1\}$ verwenden. Verwechslungen der Binärzeichen mit Dezimalzahlen sollten durch den jeweiligen Kontext (fast) ausgeschlossen sein.

1.2 Eigenschaften von Codes

Prinzipiell unterscheiden wir zwischen Codes *fester* und Codes *variabler* Länge. Für die Verarbeitung von Information in digitalen Rechenanlagen finden fast ausschliesslich Codes fester Länge Verwendung, da die zugrundeliegende Hardware (z.B. Register) dies erfordert. Codes variabler Länge werden vor allem dann eingesetzt, wenn man Komprimierung von Daten wünscht, und die Codelänge mit der Häufigkeit der zu codierenden Zeichen (Wörter) korreliert. Zuerst betrachten wir aber eine wichtige Eigenschaft von Codes fester Länge.

Definition 1.2.1 *Der* Hammingabstand H_d *(hamming distance) zweier (binärer) Codewörter* r, s *der Länge* n *ist gegeben durch*

$$H_d(r,s) = \sum_{i=1}^{n} d_i, \quad mit \quad d_i = \begin{cases} 1 & if\ r_i \neq s_i \\ 0 & otherwise \end{cases}$$

Der Hammingabstand einer Codierungsabbildung $c : A \to \mathbb{B}^n$ ist dann durch den kleinsten Hammingabstand aller codierten Wörter aus A gegeben:

$$H_d(c) = \min\{H_d(c(r), c(s)) \mid r, s \in A \wedge r \neq s\}.$$

Die Hammingdistanz hat vor allem bei Codes für *Fehlererkennung* und *Fehlerkorrektur* grosse Bedeutung und ist eng mit dem Begriff der *Redundanz* verknüpft. Je grösser der Hammingabstand eines Codes ist, desto leichter ist es, einen Fehler zu erkennen, da Fehler in einem Codewort auf ungültige Codewörter, i.e. die Hammingdistanz eines fehlerhaften Codewortes ist kleiner als die des Codes, führen können. Eine grössere Hammingdistanz bedingt aber auch das Mitführen von immer mehr "sinnloser" (redundanter) Information.

1.3 Spezielle Codes

Einer der einfachsten Codes zur Fehlersicherung ist das *Paritätsbit* p (parity bit). Angenommen wir hätten einen Code $c_1 : A \to \mathbb{B}^n$ mit $H_d(c_1) = 1$ (warum bereitet ein Code mit $H_d = 0$ wenig Freude?). Aus diesem Code konstruieren wir einen Code $c_2 : A \to \mathbb{B}^{n+1}$ mit $H_d(c_2) = 2$ wie folgt:

$$c_2(a) = c_1(a) \circ p(c_1(a)),$$

wobei das Symbol $\circ$ die *Konkatenation* von Codewörtern repräsentiert, und die Abbildung $p : A\mathbb{B}^n \to \mathbb{B}^1$ durch

$$p(b) = \begin{cases} 0 & \text{if } \sum_{i=0}^{n} b_i \text{ is even} \\ 1 & \text{if } \sum \text{ is odd} \end{cases}$$

gegeben ist (gerade Parität, bei ungerader Parität vertauschen 1 und 0). Das Paritätsbit p wird also von der Anzahl der 1–Bits in b abgeleitet. Ein Beispiel für diese Konstruktion findet sich in Tabelle 1.4.

Buchstabe	7–Bit ASCII even parity
0	0011000**0**
A	0100000**1**
z	1111101**0**
LF	0000101**0**

Tabelle 1.4: Beispiele von 7–Bit ASCII mit geradem Paritätsbit.

An diesem Beispiel ist leicht zu ersehen, dass ein 8–Bit Wort mit ungerader Anzahl von 1–Bits kein gültiges Codewort sein kann, und ein solches Wort als fehlerhaft erkannt wird. Dieser ASCII [2] Code hat daher eine Hammingdistanz $H_d = 2$.

Ein weiterer (einfacher) Code fester Länge ist der *Gray Code*, der dadurch definiert ist, dass zwei in ihrer Ordnung aufeinanderfolgende Zeichen aus einem Alphabet A derart codiert sind, dass ihre Hammingdistanz $H_d = 1$ ist. Als Beispiel ist in Tabelle 1.5 die Codierung der Dezimalziffern $\{0, \ldots, 9\}$ mit einem 4–Bit Gray Code angegeben.

Ziffer	Gray Code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1000

Tabelle 1.5: 4–Bit Gray Code für Dezimalziffern.

[2] ASCII steht für *American Standard Code for Information Interchange* und wurde 1968 eingeführt.

Der Code in Tabelle 1.5 ist zudem ein *zyklischer* Gray Code, da der Code des ersten und letzten Zeichens der Ordnung sich ebenfalls nur an einer Stelle unterscheidet. Für unser Beispiel gibt es verschiedene Gray Codierungen (finden Sie eine andere) und es werden auch nicht alle möglichen Codewörter verwendet. Es exisitieren auch analytische Möglichkeiten Gray Codes zu konstruieren. Eine Methode erzeugt *Binary Reflected Gray Codes* (BRGC), die durch folgende (induktiv definierte) Transitionssequenz T erstellt werden können:

$$T(n+1) = T(n), n+1, T(n) \quad mit \quad T(1) = 1.$$

Die Transitionssequenz gibt jene Stellen eines binären Codeworts der Länge $n+1$ an, die hintereinander verändert werden müssen, um einen (zyklischen) Gray Code zu erhalten. Für $n+1 = 3$ erhält man so die Sequenz $T = 1, 2, 1, 3, 1, 2, 1$ (überzeugen Sie sich, dass dies ein Gray Code ist). Eine der Motivationen zur Entwicklung von Gray Codes war die Minimierung des Einflusses von Fehlern bei der Nachrichtenübermittlung. Die Grundidee dabei ist, dass ein Einzelfehler im Codewort (eher) zu benachbarten Wörtern führt, i.e. das fehlerhafte Codewort also weniger dramatische Auswirkungen hat (Übermittlung ihres Kontostandes).

Ein weiterer einfacher Code fester Länge ist der k–aus–n Code, bei dem ein Codewort der Länge n genau k 1–Bits aufweist. Als Beispiel ist in Tabelle 1.6 eine Codierung der Dezimalzahlen mit Hilfe eines 1–aus–10 Codes angegeben.

Ziffer	1–aus–10 Code
0	0000000001
1	0000000010
2	0000000100
3	0000001000
4	0000010000
5	0000100000
6	0001000000
7	0010000000
8	0100000000
9	1000000000

Tabelle 1.6: 1–aus–10 Code für Dezimalziffern.

Auch hier ist leicht zu erkennen, dass eine Erhöhung der Redundanz die Fehlererkennung erleichtert. Interessant ist bei dem Beispiel in Tabelle 1.6 allerdings, dass die Hammingdistanz des Codes $H_d = 2$ nicht grösser als die des

Paritätsbitcodes (Tabelle 1.4) ist. Trotzdem können mehrere Fehler erkannt werden (warum?). Der k–aus–n Code hat vor allem bei der Umwandlung von analogen in digitale Signale Bedeutung (A/D–Wandler).

Codes variabler Länge bieten die Möglichkeit die Codewortlänge an die Häufigkeit des Auftretens eines zu codierenden Zeichens (Wortes) anzupassen, was zu effizienteren Codes führt. Allerdings steht diesem Vorteil auch der Nachteil einer aufwendigeren Technik gegenüber [3], die dafür Sorge tragen muss, dass Anfang und Ende jedes Codeworts korrekt erkannt wird. Am Beispiel des Morsecodes aus Tabelle 1.3 ist ersichtlich, dass die Zeichenfolge "−.−" entweder als "K" oder als "TET" interpretiert werden kann. Um die einzelnen Codes eindeutig zu machen, musste man daher ein Signal einführen, dass die einzelnen Codes trennt (im Morsecode ein drittes Signal, das einfach durch eine kleine Pause zwischen den einzelnen codierten Buchstaben definiert wurde).

Eine andere Möglichkeit wäre, die einzelnen Codes nicht *seriell*, sondern *parallel* zu übertragen. Bei einer parallelen Übertragung wird je ein Zeichen eines codierten Zeichens auf einer eigenen Leitung übertragen. Da alle "Teile" eines Codewortes zur gleichen Zeit beim Empfänger ankommen ist die Decodierung der Zeichen eindeutig (wenn man Störungsfreiheit der Leitungen voraussetzt). Zusätzlich erhöht sich die Geschwindigkeit der Zeichenübertragung, was aber durch höhere Hardwarekosten erkauft werden muss (siehe Abb. 1.1).

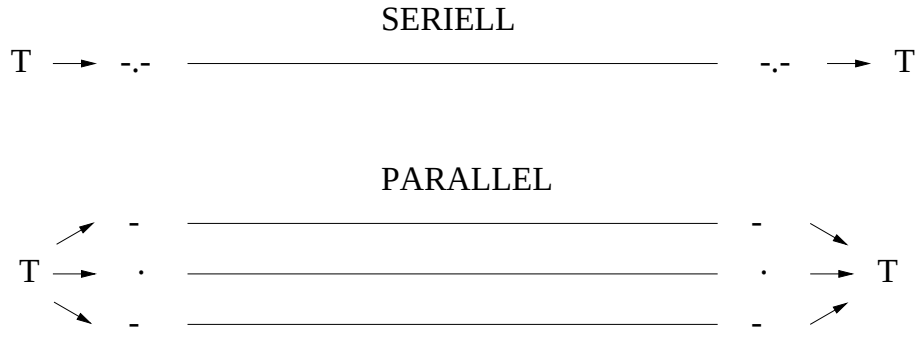

Abbildung 1.1: Serielle und parallele Übertragung des Morsecodes für den Buchstaben "T".

Ein (einfaches) Beispiel für einen Code variabler Länge ist in Tabelle 1.7 angeführt.

Zur Decodierung serieller Codes variabler Länge sind *Codebäume* ein praktisches Hilfsmittel (siehe Abb. 1.2).

[3] Dieses Grundprinzip technischer Systeme wird meist als *trade off* bezeichnet.

Zeichen	Code
A	0
B	10
C	11
D	001

Tabelle 1.7: Ein Code variabler Länge.

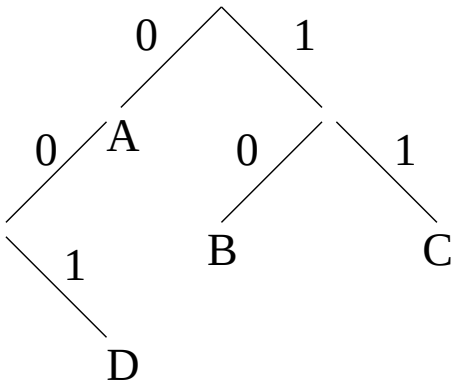

Abbildung 1.2: Ein Codebaum für den Code aus Tabelle 1.7.

Um ein codiertes Zeichen zu decodieren wird dabei einfach (von der Wurzel beginnend) dem Pfad im Baum gefolgt, der durch das jeweils empfangene Zeichen ausgewählt wird. Erreicht man einen Knoten, bei dem ein decodiertes Zeichen eingetragen ist, notiert man dieses und beginnt wieder an der Wurzel. In unserem Beispiel haben wir nun das oben angesprochene Problem, denn wir würden in unserem Baum nie zu dem Blatt mit dem Zeichen "D" gelangen, obwohl für dieses Zeichen ein Code existiert. Der vorliegende Code ist also nicht eindeutig. Er wäre dies nur dann, wenn decodierte Zeichen (hier Buchstaben) nur an den Blättern (Endknoten) des Baums stehen würden. Diese Erkenntnis für Codes variabler Länge ist so fundamental, dass sie (in anderer Formulierung) einen eigenen Namen trägt. Die *Fano*–Bedingung besagt folgendes:

Bedingung 1.3.1 *Kein Codewort darf Anfang eines anderen Codeworts sein (*Fano–*Bedingung).*

Wird ein Code, der die Fano–Bedingung erfüllt seriell übertragen, so ist die Decodierbarkeit garantiert (ohne zusätzliche Start/Stop–Zeichen). Zu beachten ist auch, dass die Fano–Bedingung hinreichend für die Umkehrbarkeit der Codierung, aber nicht notwendig ist (was ist daran eigentlich unterschiedlich?).

1.4 Informationsgehalt und Entropie

Bisher haben wir uns vornehmlich mit der Repräsentation von Information beschäftigt, i.e. einer qualitativen Beschreibung von Zeichen und Wörtern. Nun wollen wir uns aber der Frage zuwenden, ob und wie wir Information quantifizieren (oder messen) können. Was bedeutet überhaupt "viel" oder "wenig" Information? Dazu einige Informationen:

a) Frau Müller wird nächste Woche im Lotto sechs Richtige haben.

b) Morgen geht die Sonne auf.

c) Heute entfällt die Vorlesung Digitale Rechenanlagen.

Wie würden Sie den *Informationsgehalt* dieser drei Aussagen reihen (was unterscheidet c) von a) und b))? Offenbar spielt bei der Bewertung des Informationsgehalts die *Wahrscheinlichkeit* mit der eine Aussage eintritt die zentrale Rolle. Der Informationsgehalt einer Nachricht (Zeichen, Wort) ist umso höher, je kleiner die Wahrscheinlichkeit ihres Eintretens ist, oder der Informationsgehalt ist umgekehrt reziprok zur Wahrscheinlichkeit. Der Informationsgehalt einer "trivialen" Aussage ist nahezu (oder gleich) null. Einen negativen Informationsgehalt schliessen wir aus, da diesem keine intuitiv einsichtige Rolle zukommt. Kleiden wir diese Überlegungen in ein formales Gewand, so stellen wir folgende Forderungen an den Informationsgehalt (Carlson, 1975) $\mathcal{I}_A$ einer Nachricht A mit der Wahrscheinlichkeit p_A:

$$\mathcal{I}_A = f(p_A) \tag{1.1}$$

$$\mathcal{I}_A \geq 0 \quad mit \quad 0 \leq p_A \leq 1 \tag{1.2}$$

$$\lim_{p_A \to 1} \mathcal{I}_A = 0 \tag{1.3}$$

$$\mathcal{I}_A > \mathcal{I}_B \Leftrightarrow p_A < p_B \tag{1.4}$$

Viele Funktionen $\mathcal{I}_A = f(p_A)$ erfüllen die Gleichungen 1.1 bis 1.4. Es bleibt aber die Frage wie wir den Informationsgehalt von zwei Nachrichten A und B messen. Dazu benötigen wir noch den Begriff der *Unabhängigkeit* von Nachrichten. In obiger Beispielnachricht c) ist es unbedingt erforderlich, dass eine zweite Nachricht B über den Wochentag mitgeteilt wird, da sie sonst nutzlos ist. Der Informationsgehalt dieser Nachricht ist also *abhängig*

von einer anderen und wir können den Informationsgehalt beider Nachrichten **nicht** als Summe der einzelnen Informationsgehalte angeben. Sind zwei Nachrichten jedoch unabhängig (z. B a) und b)), dann ist es sinnvoll zu verlangen, dass die Summe der einzelnen Informationsgehalte den Informationsgehalt beider Nachrichten ergibt. Für die Wahrscheinlichkeiten zweier unabhängiger Ereignisse (Nachrichten) A und B gilt überdies

$$p(AB) = p_A p_B, \tag{1.5}$$

und für den Informationsgehalt dieser unabhängigen Nachrichten fordern wir

$$\mathcal{I}_{AB} = \mathcal{I}_A + \mathcal{I}_B. \tag{1.6}$$

Durch Einbeziehung letzterer Bedingung 1.6 lässt sich zeigen (Ash, 1965), dass genau eine Funktion $\mathcal{I}_A = f(p_A)$ existiert, die die Bedingungen 1.1 bis 1.4 erfüllt, und zwar die Funktion

$$\mathcal{I}_A = log_b \frac{1}{p_A} = -log_b\ p_A, \tag{1.7}$$

der Logarithmus zur Basis b. Es bleibt noch die Frage nach der Wahl von b. Übliche Kandidaten wären die Zahl 10 oder die *Eulersche* Zahl e. In diesem Fall wurde jedoch $b = 2$ gewählt und wir schreiben dann für $log_2\ x = ld\ x$ (logarithmus dualis). Dies hat den einfachen Grund, dass dann ein Zeichen aus dem Zeichenvorrat $\mathbb{B}$ mit der Wahrscheinlichkeit $p = \frac{1}{2}$ den Informationsgehalt von 1 *bit* (Einheit des Informationsgehalts) aufweist. In desem Fall ist also der Informationsgehalt von einem binären Zeichen (Bit) genau 1 bit. Zu beachten ist aber, dass im allgemeinen der Informationsgehalt von einem Bit (binary digit) **ungleich** 1 bit ist.

Betrachten wir eine Nachrichtenquelle, die viele verschiedene Nachrichten (Zeichen, Wörter) sendet, dann wird zur Charakterisierung der Quelle zumeist der *mittlere Informationsgehalt* verwendet.

Definition 1.4.1 *Eine Nachrichtenquelle, bei der zu jedem Zeitpunkt die Wahrscheinlichkeit des nächsten zu sendenden Zeichens gleich der mittleren Häufigkeit dieses Zeichens ist, heisst* stochastische *oder* Shannonsche *Nachrichtenquelle.*

Die mittlere Häufigkeit h_z eines Zeichens ist dabei einfach durch die Anzahl n_z der gesendeten Zeichen z und die Gesamtanzahl N der gesendeten Zeichen gegeben.

$$h_z = \frac{n_z}{N} \tag{1.8}$$

Aus Definition 1.4.1 folgt, dass die Zeichen einer stochastische Nachrichtenquelle statistisch unabhängig sind, und damit gilt für den Informationsgehalt von N Zeichen dieser Quelle

$$Np_1\mathcal{I}_1 + Np_2\mathcal{I}_2 + \ldots + Np_m\mathcal{I}_m = \sum_{i=1}^{m} Np_i\mathcal{I}_i, \tag{1.9}$$

wenn die Quelle m verschiedene Zeichen erzeugt. Für die Wahrscheinlichkeiten dieser Zeichen muss $\sum_{i=1}^{m} p_i = 1$ gelten (warum?). Der mittlere Informationsgehalt der Quelle, den wir auch als *Entropie* [4] $\mathcal{H}$ der Quelle bezeichnen ist dann einfach

$$\mathcal{H} = \sum_{i=1}^{m} p_i\mathcal{I}_i = \sum_{i=1}^{m} p_i \; ld \; \frac{1}{p_i} = -\sum_{i=1}^{m} p_i \; ld \; p_i. \tag{1.10}$$

Um uns eine bessere Vorstellung von der Bedeutung der Entropie zu machen, betrachten wir eine binäre (stochastische) Nachrichtenquelle deren Entropie nach Gleichung 1.10 durch

$$\mathcal{H} = p_1 \; ld \; \frac{1}{p_1} + p_2 \; ld \; \frac{1}{p_2} \tag{1.11}$$

gegeben ist. Da auch hier $p_1 = 1 - p_2$ gelten muss, ist $\mathcal{H} = \mathcal{H}(p_1)$ und kann somit als Funktion einer Veränderlichen dargestellt werden (Abb. 1.3).

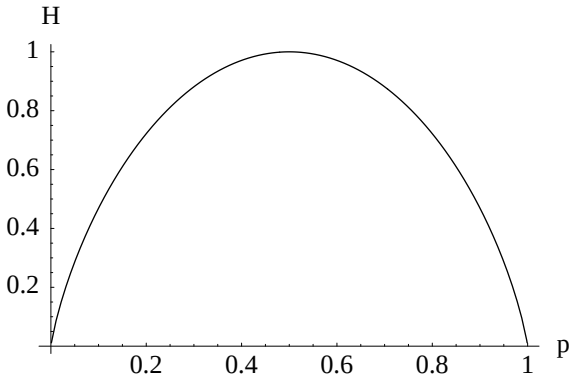

Abbildung 1.3: Entropie einer binären Nachrichtenquelle.

[4] Dieser Begriff stammt eigentlich aus der Thermodynamik.

Wir sehen, dass die Entropie an den Stellen $p = 0.0$ und $p = 1.0$ gegen 0 geht (versuchen Sie, das aus Gleichung 1.11 abzuleiten), was nicht weiter überrascht, da diese Fälle einer Nachrichtenquelle entsprechen, die nur ein einziges Zeichen sendet. Das Maximum der Entropie (finden Sie es analytisch?) befindet sich an der Stelle $p = 0.5$, da die einzelnen Zeichen dann maximalen Informationsgehalt haben. Weiters ist zu bemerken, dass dieses Maximum flach ist, da bei einer Wahrscheinlichkeit von $p = 0.3$ die Entropie immer noch $H = 0.88$ bit beträgt.

1.5 Optimalcodierung

Wir können nun die Entropie dazu verwenden, um die Effizienz einer Codierung quantitativ zu beschreiben. Dazu benötigen wir noch den Begriff der mittleren Codewortlänge eines Binärcodes

$$\mathcal{L} = \sum_{i=1}^{m} p_i \; l_i, \tag{1.12}$$

wobei m die Anzahl der Codewörter, l_i die Länge (Anzahl der Bits) des Codeworts m_i, und p_i dessen Wahrscheinlichkeit (und damit des codierten Zeichens) ist. Die Einheit der mittleren Wortlänge ist das *bit*. Nimmt man nämlich an, dass jedes Bit den Informationsgehalt von 1 bit hat (Maximalfall), dann sieht man, dass dies der maximalen Entropie entspricht, die man mit dieser Codierung übertragen kann. Umgekehrt versucht man bei gegebener Entropie einer Quelle, die Codierung derart zu wählen, dass die mittlere Wortlänge möglichst nahe an die Entropie der Quelle kommt. Dass dies möglich ist, besagt der *Fundamentalsatz der Codierung* (Lochmann, 1995) oder das

Theorem 1.5.1 Shannonsches Codierungstheorem: *Für beliebige binäre Codierungen gilt $\mathcal{H} \leq \mathcal{L}$ und jede Nachrichtenquelle kann durch binäre Codierung so codiert werden, dass der positive Wert $\mathcal{L} - \mathcal{H}$ beliebig klein wird.*

Besonders der zweite Teil dieses Theorems ist für die Praxis von grosser Bedeutung, da es einen optimalen Code postuliert, der nicht weiter verbessert werden kann. Der Optimalcode ist einfach durch die Bedingung $\mathcal{L} = \mathcal{H}$ gegeben. Da ein solcher Optimalcode keine "überflüssige" Information beinhaltet, wird die Redundanz $\mathcal{R}$ eines Codes mit

$$\mathcal{R} = \mathcal{L} - \mathcal{H} \tag{1.13}$$

definiert. Die *relative Redundanz* r und *Codeeffizienz* η_C ist dann durch

$$r = \frac{\mathcal{R}}{\mathcal{L}} = 1 - \frac{\mathcal{H}}{\mathcal{L}} = 1 - \eta_C \tag{1.14}$$

gegeben.

Beispiel: Welche Redundanz weist die deutsche Sprache auf? Wenn wir annehmen, dass die 26 Zeichen (ohne Umlaute, ß und Zwischenraum) gleichwahrscheinlich auftreten, dann erhält man die (maximale) Entropie $\mathcal{H}_0 = ld\ 26 = 4.7\ bit$ (pro Zeichen). Berücksichtigt man statistische Abhängigkeiten innerhalb von Silben, Wörtern und Sätzen, so gelangt man zu einer (realen) Entropie $\mathcal{H}_\infty \approx 1.6\ bit$ (Küpfmüller, 1954). Eine zu Gleichung 1.14 äquivalente Formulierung der relativen Redundanz ist dann $r = \frac{\mathcal{H}_0 - \mathcal{H}_\infty}{\mathcal{H}_0} = 0.6595$. In der deutschen Sprache sind also $\frac{2}{3}$ der transportierten Information "überflüssig" (ist das sinnvoll?). ◊

Wenn wir uns die Erkenntnis aus dem Shannonschen Codierungstheorem vor Augen halten, begegnet man einem häufig auftretenden Problem in den technischen Wissenschaften. Es ist zwar sehr erfreulich, dass die Redundanz eines Codes beliebig klein werden kann, aber diese Erkenntnis hat nur dann praktischen Nutzen, wenn es auch Methoden gibt, einen solchen (optimalen) Code zu konstruieren. Glücklicherweise wurden solche Methoden schon entwickelt, e.g. der *Shannon–Fano–Algorithmus*. Wir wollen uns hier näher mit einem ähnlichen Verfahren beschäftigen, der *Optimalcodierung nach Huffman* (Huffman, 1952)

Die Grundidee, die auch der Huffman–Codierung zugrunde liegt, ist uns schon bekannt: Die Wortlänge eines codierten Wortes sollte indirekt proportional zu der Wahrscheinlichkeit des Auftretens des Wortes sein. Zur Illustration des Verfahrens sind in Tabelle 1.8 die Wahrscheinlichkeiten des Auftretens von Zeichen einer stochastischen Nachrichtenquelle mit dem Zeichenvorrat $Z = \{A, B, C, D, E, F\}$ angegeben.

Zeichen	Wahrscheinlichkeit
A	0.1250
B	0.1250
C	0.2500
D	0.0625
E	0.1875
F	0.2500

Tabelle 1.8: Auftrittswahrscheinlichkeit der Zeichen einer stochastischen Nachrichtenquelle mit dem Zeichenvorrat Z.

Man verfährt nun folgendermassen (siehe Abb. 1.4):

a. Man ordnet die Symbole untereinander nach fallenden Wahrscheinlichkeiten.

b. Von unten beginnend ordnet man dem Symbol mit der kleinsten Wahrscheinlichkeit das Bit 0, dem nächstgrösseren das Bit 1 zu.

c. Man summiert die Wahrscheinlichkeiten der beiden Symbole und ordnet sie quasi als neues Einzelsymbol entsprechend der Summenwahrscheinlichkeit in die vertikale Symbolfolge ein.

d. Wenn die Summenwahrscheinlichkeit $p < 1.0$ ist, dann fährt man mit b. fort, ansonsten ist der Code fertig konstruiert.

e. Die einzelnen Codewörter liest man vom horizontalen Ende des entstandenen Pfades zu den jeweiligen Symbolen ab, indem man alle dabei auftretenden Bits der Reihe nach notiert.

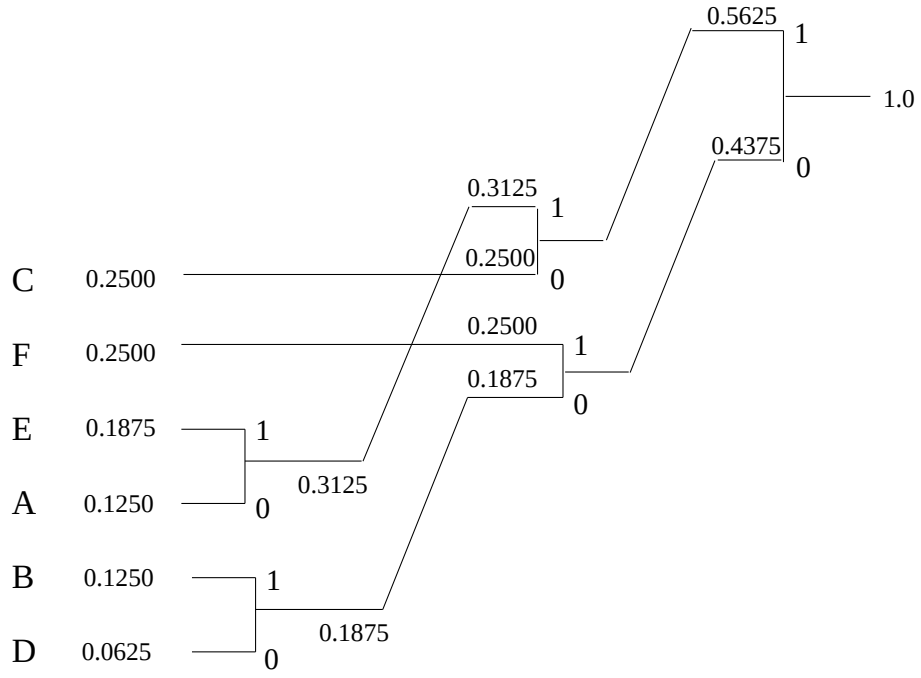

Abbildung 1.4: Konstruktion eines optimalen Codes nach Huffman.

Der für unser Beispiel konstruierte Code ist in Tabelle 1.9 dargestellt.

An diesem Beispiel sind noch folgende Dinge bemerkenswert. Die Zeichen mit der höchsten Auftrittswahrscheinlichkeit haben den kürzesten Code. Die Konstruktion des Huffman–Codes liefert immer einen Code, der die Fano–Bedingung erfüllt. Die Konstruktion ist in diesem Beispiel nicht eindeutig,

denn wir hätten einen anderen Huffman–Code erhalten, wenn wir eine Summenwahrscheinlichkeit, die einer anderen Wahrscheinlichkeit in der Tabelle gleich ist, nicht über dem Symbol, sondern unter dem Symbol eingeordnet hätten (z.B. bei E mit 0.1875).

Wenn Sie zur Übung die Redundanz des entstehenden Codes berechnen, werden Sie feststellen, dass diese ungleich 0 ist. Ein *redundanzfreier* Code (R = 0) ist nur unter folgender Bedingung zu erreichen:

Bedingung 1.5.1 *Für einen redundanzfreien Code muss für die Auftrittswahrscheinlichkeiten p_i eines Zeichens $p_i = \frac{1}{2^{n_i}}$ mit $n_i \in \mathbb{N}$ gelten.*

Die Herleitung dieser Bedingung wird zur Übung empfohlen. Wie können wir aber die Redundanz unseres Huffman–Codes noch weiter verbessern? Dies muss möglich sein, da das Shannonsche Codierungstheorem 1.5.1 besagt, dass $\mathcal{R}$ beliebig klein werden kann.

Dies ist in der Tat durch die Methode der *Codeerweiterung* möglich. Hierbei werden nicht die einzelnen Zeichen der Quelle codiert, sondern zwei oder mehrere zugleich. In unserem Beispiel in Tabelle 1.8 könnte man immer zwei Zeichen zu einem Wort zusammenfassen (wieviele Wörter gibt es?) und die so entstandenen Wörter codieren. Bei statistischer Unabhängigkeit der Zeichen ergibt sich dann für ein Wort aus zwei Zeichen die Auftrittswahrscheinlichkeit $p_{XY} = p_X p_Y$, was zu einer feineren Aufspaltung der Wahrscheinlichkeiten führt und eine effizientere (Huffman)Codierung erlaubt. Diese Vorgangsweise ist auch die Basis eines konstruktiven Beweises des Shannonschen Codierungstheorems 1.5.1.

Die Huffman–Codierung hat auch durchaus praktische Anwendungen. So ist eine modifizierte Form der Huffman–Codierung in allen heutigen Faxgeräten implementiert.

Zeichen	Optimalcode
A	110
B	001
C	10
D	000
E	111
F	01

Tabelle 1.9: Huffman–Code für den Zeichenvorrat aus Tabelle 1.8.

1.6 Nachrichtenkanal und Leitungscodierung

Bisher haben wir uns mit Information und deren Codierung ausschliesslich unter informationstheoretischen Aspekten beschäftigt. Nun wollen wir noch kurz einige wesentliche technische Aspekte berücksichtigen, die für die Übertragung von Information und deren Codierung wichtig sind.

Um Nachrichten von einem *Sender* (Quelle) zu einem *Empfänger* (Senke) zu senden, benötigen wir einen *Kanal*, in dem diese Nachrichten gesendet werden. Ein (sehr einfaches) Diagramm eines Übertragungsssytems ist in Abb. 1.5 dargestellt.

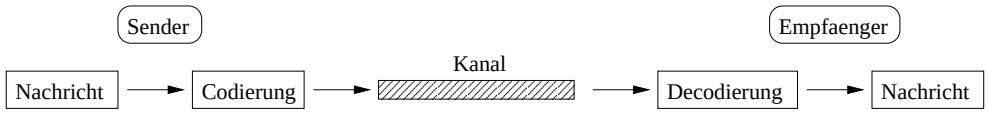

Abbildung 1.5: Prinzip eines Nachrichtenübertragungssystems.

Dabei versteht man unter Kanal nicht nur eine Verbindung im herkömmlichen Sinn (Kupferleitung, Glasfaserkabel), sondern auch technisch abgegrenzte Bereiche in einem Medium (Frequenzkanäle für elektromagnetische Wellen, die sich aber alle im selben Medium Luft ausbreiten, z. B. Radio).

Der "natürliche Feind" jedes Nachrichtensystems ist das *Rauschen* (noise). Dieser Begriff rührt zwar ursprünglich vom akustischen Phänomen des Rauschens her, doch meint man damit jedwede Störungsquelle, die ein Nachrichtensignal verändert und damit Teile der übertragenen Information zerstört. Diese Verallgemeinerung geht sogar soweit, dass dieser Begriff auch in den Computerwissenschaften verwendet wird, wenn Daten fehlerhaft sind (verrauschte Daten), oder Ergebnisse nur näherungsweise berechnet werden (verrauschte Berechnungen).

In einem Nachrichtensystem gibt es viele, verschiedene Rauschquellen in Sender (z. B. thermisches Rauschen von elektrischen Widerständen), Kanal (z. B. Verunreinigungen in Glasfasern), und Empfänger (z. B. Schrotrauschen in Halbleitern). Modellhaft wird jegliches Rauschen dem Kanal zugeordnet, womit wir eine genauere Formulierung des Fundamentaltheorems 1.5.1 angeben können.

Theorem 1.6.1 *Für eine Nachrichtenquelle mit der Entropierate $\mathcal{H}'$ und einem Übertragungssystem mit der Kanalkapazität $\mathcal{C}$ gibt es für den Fall $\mathcal{H}' \leq \mathcal{C}$ eine Codierung, die die Übertragung der Nachricht über den rauschbehafteten Kanal mit beliebig kleiner Fehlerrate erlaubt.*

Die Entropierate $\mathcal{H}'$ ist einfach als Entropie pro Zeiteinheit definiert. Daraus folgt, dass auch die Kanalkapazität $\mathcal{C}$ die Einheit $\frac{bit}{sec}$ haben muss

(warum?). Die Kanalkapazität stellt die über diesen Kanal maximal übertragbare Entropierate dar, die aber wiederum vom Kanalrauschen abhängt (welche Kapzität hätte ein durchtrenntes Kabel?).

Das Problem mit Theorem 1.6.1 ist nun aber wieder eine geeignete Codierung zu finden. In Zusammenhang mit der Übertragung von Signalen über einen Kanal (oder Leitung) spricht man auch von *Leitungscodierung* der Signale. Die (meist schon) codierte Nachricht muss in der Regel einer weiteren Codierung unterworfen werden, um die Signale an die physikalischen Eigenschaften des Kanals anzupassen. Im Falle elektrischer Signale muss ein Leitungscode die folgenden wesentlichen Bedingungen erfüllen.

i) Gleichstromfreiheit (Übertrager)

ii) Optimierung des Leistungsdichtespektrums (Nebensprechen)

iii) Störabstand (Rauschen)

iv) Taktinformation (Synchronisation)

Die Bedeutung dieser Bedingungen wollen wir an einigen Beispielen kurz erläutern. Wenn wir einen binären Code ganz einfach mit zwei Spannungszuständen codieren, sodass dem Bit 0 die Spannung von 0V und dem Bit 1 die Spannung von 5V entspricht, dann erhalten wir ein Leitungssignal wie es in Abb. 1.6 dargestellt ist.

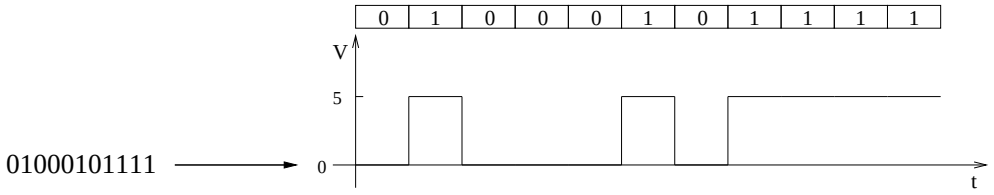

Abbildung 1.6: Einfache Leitungscodierung.

Diese einfache Leitungscodierung hat gravierende Schwächen, da der Gleichanteil der Spannung ungleich null ist, was zu technischen Problemen bei der Übertragung führt. Weiters ist es nicht einfach aus diesem Signal den Takt (die Frequenzrate mit der die Bits übertragen werden) zu regenerieren, da eine längere Kette von 0 oder 1 immer gleichen Spannungspegel liefert. Eine wesentliche Verbesserung stellt der *Manchester*–Code dar, der schematisch in Abb. 1.7 gezeigt wird.

Der Manchester–Code ist ein *bipolarer* Code, d. h. die beiden Binärzustände werden auf zwei Spannungspegel abgebildet, die in Summe 0 ergeben, was zur Folge hat, dass der Gleichanteil verschwindet. Das Bit 0 wird so codiert, dass

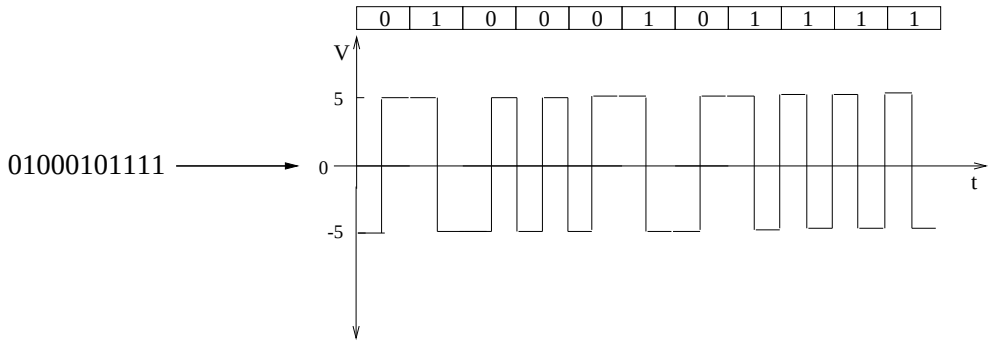

Abbildung 1.7: Leitungscodierung mit Manchester–Code.

in der ersten Hälfte des Bitintervalls (Periode) der negative Spannungspegel und in der zweiten Hälfte, der positive Spannungspegel verwendet wird. Beim Bit 1 ist es genau umgekehrt, was technisch einer Phasenverschiebung um 180° entspricht. Ausserdem ist die Taktregeneration hier sehr viel einfacher, da es nur zwei Zeiten zwischen Taktflanken geben kann: die Periodendauer (des Bitintervalls) T_0 oder $\frac{T_0}{2}$.

Ein Nachteil des Manchester–Codes resultiert aber auch aus letzter Eigenschaft, denn die benötigte *Bandbreite* ist doppelt so gross wie bei einem Rechtecksignal konstanter Periodendauer T_0. Die Bandbreite eines Signals ist jener Frequenzbereich, den das Signal benötigt um (näherungsweise) unverformt beim Empfänger anzukommen (Rauschfreiheit vorausgesetzt). Prinzipiell gilt, dass die Bandbreite $\Delta B \approx .\frac{1}{T_0}$ ist. Je kürzer die Periodendauer also ist (höhere Frequenz, höhere Übertragungsgeschwindigkeit), desto grösser ist die benötigte Bandbreite, die der Kanal zur Verfügung stellen muss. Dies ist auch der Grund warum einfache Kupferkabel nur eine relativ geringe Übertragungsgeschwindigkeit zulassen, da deren maximale Bandbreite relativ klein ist (siehe klassische Telephonleitung).

Den Zusammenhang zwischen zeitlichem Verlauf eines Signals und der Bandbreite bzw. des Frequenzspektrums liefert die *Fouriertransformation*, die in der Nachrichtentechnik und ganz allgemein in der Signalverarbeitung (Fernerkundung, Bildverarbeitung, Spektroskopie, etc.) von fundamentaler Bedeutung ist.

Kapitel 2

Zahlendarstellung

Zu Beginn der Entwicklung der digitalen Rechenanlagen mit ersten elektrischen Bauteilen [1] war man fast ausschliesslich damit befasst, einfache numerische Berechnungen auf diesen Maschinen durchzuführen. Heute löst diese Aufgaben (und noch viel mehr) zwar jeder Taschenrechner in viel kürzerer Zeit, doch ein zentrales Problem muss in beiden Systemen und in allen digitalen Rechenanlagen gelöst werden: wie repräsentiert man die Zahlen, mit denen man rechnet?

Im Grunde können wir die Frage mit unserem Wissen aus Kapitel 1 beantworten, denn die Repräsentation von Information ist gleichbedeutend mit der Codierung von Zeichen. Eine wesentliche Einschränkung gibt es für das Rechnen mit Zahlen in digitalen Rechenanlagen, die wir auch schon angedeutet haben. Die Hardware eines Rechners ist (heute noch) statisch, d. h. der Code für die Darstellung von Zahlen muss eine feste Länge aufweisen.

Wenn wir von Zahlen sprechen, dann meinen wir fast immer *Dezimalzahlen*, da dies das weltweit dominierend Zahlensystem ist. Digitale Rechenanlagen verfügen aber definitionsgemäss nur über zwei darstellbare Zustände mit Zeichen aus dem Alphabet $\mathcal{B}$. Unsere Aufgabe ist es daher eine Codierung zu finden, die eine Untermenge der Dezimalzahlen aus dem Alphabet $\mathcal{D}$ (mit Wörtern aus Dezimalziffern(zeichen)) auf $\mathcal{B}^n$ abbildet.

Mit Binärwörtern der Länge n können wir maximal 2^n verschiedene Dezimalzahlen codieren. Wir könnten daher für die Zahlen $1-16$ alle Möglichkeiten von 4–Bit–Wörtern verwenden und jeder Dezimalzahl ein (beliebiges) Binärwort zuordnen. Selbst wenn wir diese willkürliche Zuordnung weltweit normieren würden, wäre diese Vorgangsweise ineffizient, da wir dann zur Decodierung eine Tabelle von (allgemein) 2^n Einträgen brauchen würden. Anstelle einer expliziten Codierung jeder Zahl hat man daher versucht, ana-

[1] Konrad Zuse begann 1931 mit dem Bau seines elektromechanischen Rechners Z–1.

lytische Methoden zur Konstruktion eines Binärcodes für Dezimalzahlen zu finden.

2.1 Einfache Binärdarstellung

Die analytische Vorgangsweise zur (De)Codierung von Dezimalzahlen beruht auf folgender Definition.

Definition 2.1.1 *In einem* polyadischen *Zahlensystem mit der Basis* $B \in \mathbb{N}$ *mit* $B > 1$ *wird eine Dezimalzahl* $m \in \mathbb{N}$ *als Summe von gewichteten Potenzen von* B *folgendermassen dargestellt:*

$$m = \sum_{i=0}^{n-1} b_i B^i = b_0 + b_1 B + b_2 B^2 + \ldots + b_{n-1} B^{n-1}. \tag{2.1}$$

Die Gewichte b_i sind hier die Ziffern des Zahlensystems zur Basis B, wobei die Ziffer b_0 ganz rechts steht und die Zahl dann folgendes Aussehen hat: $< b_{n-1} b_{n-2} \ldots b_1 b_0 >$. Für unsere Zwecke ist das *dyadische* Zahlensystem (Binärsystem) von grösster Bedeutung, daneben gibt es aber auch noch andere Zahlensysteme, die praktischen Nutzen im Bereich der digitalen Rechenanlagen haben (Tabelle 2.5).

Zahlensystem	Basis	Ziffern
Binär	2	$0, 1$
Oktal	8	$0, 1, 2, 3, 4, 5, 6, 7$
Dezimal	10	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
Hexadezimal	16	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

Tabelle 2.1: Einige wichtige Zahlensysteme.

Da es bei der Verwendung unterschiedlicher Zahlensysteme leicht zu Verwechslungen kommen kann, werden wir das verwendete Zahlensystem (wenn nötig) immer als Index angeben. Z. B. ist die Zahl 1000 erst dann eindeutig festgelegt, wenn man das Zahlensystem mit angibt ($1000_{(10)}$ oder auch $1000_{(2)}$, welcher Dezimalzahl entspricht $1000_{(5)}$?).

Beispiel: Die Zahl $10101_{(2)}$ lässt sich mit Definition 2.1.1 sofort in das Dezimalsystem umwandeln.

$$10101_{(2)} = 1 + 2^2 + 2^4 = 21_{(10)}$$

Ganz analog lassen sich Zahlen in anderen Systemen umwandeln.

$$3147_{(8)} = 7 + 4 \times 8 + 8^2 + 3 \times 8^3 = 1639_{(10)}$$

$$1A2C_{(16)} = 12 + 2 \times 16 + 10 \times 16^2 + 16^3 = 6700_{(10)}$$

$\Diamond$

Wie können wir aber nun Dezimalzahlen in ein anderes Zahlensystem umwandeln? Dazu dividieren wir Gleichung 2.1 durch die Basis B und erhalten

$$\frac{m - b_0}{B} = b_1 + b_2 B + \ldots + b_{n-1} B^{n-2} = m'. \tag{2.2}$$

Daraus folgt $m = m'B + b_0$ und man sieht, dass die Ziffer b_0 der Rest bei Division durch B ist. Verfährt man dann mit m' in gleicher Weise, dann ist der nächste Rest die Ziffer b_1 usw, bis $m' = 0$. Man spaltet so der Reihe nach alle Ziffern der Zahl zur Basis B ab, was dem Verfahren seinen Namen gibt: die Methode der *sukzessiven Division*.

Beispiel: Wir wollen mittels sukzessiver Division die Zahl $31_{(10)}$ in das ternäre Zahlensystem ($B = 3$) umwandeln.

m	m'	b_i
31	10	1
10	3	1
3	1	0
1	0	1

Tabelle 2.2: Sukzessive Division durch 3.

Wir erhalten als Ergebnis $31_{(10)} = 1011_{(3)}$ (überprüfen Sie die Korrektheit des Resultats). $\Diamond$

Wir überlegen uns noch kurz, wie gross die grösste darstellbare Zahl im polyadischen Zahlensystem zur Basis B mit n Stellen ist. Ausgehend von Gleichung 2.1 erhalten wir,

$$m_{max} = b_{max} \sum_{i=0}^{n-1} B^i = b_{max}(1 + B + B^2 + \ldots + B^{n-1}), \tag{2.3}$$

wobei b_{max} die Ziffer mit dem grössten Zahlenwert der Basis B ist. Wir sehen, dass die Summe in Gleichung 2.3 eine *geometrische Reihe* darstellt, für die

$$\sum_{i=0}^{n-1} q^i = \frac{1-q^n}{1-q} \tag{2.4}$$

gilt. Mit $b_{max} = B - 1$ (warum?) erhalten wir dann für Gleichung 2.3

$$m_{max} = b_{max} \sum_{i=0}^{n-1} B^i = (B-1)\frac{1-B^n}{1-B} = B^n - 1. \tag{2.5}$$

Dieses Ergebnis hätten wir auch einfacher erhalten können, wenn wir einfach die Anzahl der darstellbaren Zahlen zur Basis B mit n Stellen betrachtet hätten (versuchen Sie es), doch zeigt dieses Beispiel sehr schön, auf welch unterschiedlichen Wegen man an ein Ziel gelangen kann.

Bisher haben wir nur positive Zahlen betrachtet, doch benötigen wir natürlich auch die binäre Darstellung von negativen Zahlen.

2.2 Darstellung negativer Zahlen

Negative Dezimalzahlen werden einfach durch Voranstellen eines Minuszeichens dargestellt. Wir könnten dies auch für Binärdarstellungen übernehmen, doch gibt es technische Schwierigkeiten bei Zahlenoperationen, wenn wir derartige Binärdarstellungen in digitalen Rechenanlagen verwenden würden.

Wenn wir z. B. die Addition $-1_{(10)} + -1_{(10)}$ betrachten, so haben wir gelernt, dass das Ergebnis eine negative Zahl sein muss. Würden wir eine 2–Bit–Zahl mit einem dritten Bit als Vorzeichen betrachten, dann könnten wir diese Addition als $101_{(2)} \oplus 101_{(2)}$ (mit $\oplus$ für die Binäraddition) darstellen. Allerdings wäre das "naive" Ergebnis dieser Addition die Binärzahl $010_{(2)}$, welche $+2_{(10)}$ entsprechen würde. Wir müssten also das binäre Minuszeichen gesondert behandeln, was eben zusätzlichen technischen Aufwand bedeuten würde.

Eine elegantere Methode beruht auf der Idee, ein Vorzeichenbit nicht nur zur Indikation des Vorzeichens zu verwenden, sondern auch zur Darstellung der Zahlen Wir haben gesehen, dass die grösste darstellbare (natürliche) Binärzahl mit n Stellen $2^n - 1$ (Gleichung 2.5) ist. Wir verwenden nun diese n Stellen einfach für die Darstellung von Zahlen aus dem Intervall $[-2^{n-1}+1,\ 2^{n-1}-1]$ und interpretieren das *Most Significant Bit* (MSB, hier das Bit linksaussen) als Vorzeichenbit.

Um einfache Addition (und Subtraktion) von Zahlen in dieser Darstellung zu ermöglichen Müssen wir das Gewicht des MSB geeignet wählen. Eine Möglichkeit ist die *1–Komplement*–Darstellung für eine Zahl $z \in \mathbb{Z}$

$$z = -b_{n-1}(2^{n-1} - 1) + \sum_{k=0}^{n-2} b_k B^k. \tag{2.6}$$

Aufmerksame Leserinnen werden schon hier feststellen, dass es für $z = 0$ zwei Darstellungen gibt. Ist das MSB gesetzt (gleich 1), dann hat diese Stelle also den Wert $-2^{n-1} + 1$. Daraus folgt aber, dass man zur Darstellung der Zahl -1 nicht einfach das Vorzeichenbit (MSB) und das *Least Significant Bit* (LSB, hier das Bit rechtsaussen) setzen kann (welcher Zahl entspräche das?), sondern man muss auch die Wertigkeit des MSB berücksichtigen.

Ist $0 \leq z \leq 2^{n-1} - 1$ ergibt sich die einfache binäre Darstellung mit $b_{n-1} = 0$ (positives Vorzeichen). Für $-2^{n-1} + 1 \leq z \leq 0$ sucht man das positive $z' = z + 2^{n-1} - 1$ und stellt dieses in einfacher Binärdarstellung mit $b_{n-1} = 1$ (negatives Vorzeichen) dar.

Beispiel: Einige spezielle Zahlen im 1–Komplement für $n = 8$ (8–Bit–Zahlen).

$z_{(10)}$	$z'_{(10)}$	$k_1(z)_{(2)}$
1	-	00000001
-1	126	11111110
127	-	01111111
-127	0	10000000
0	-	00000000
-0	127	11111111

Tabelle 2.3: 8–Bit–Zahlen im 1–Komplement.

$\Diamond$

Aus den Zahlen in Tabelle 2.3 ist jetzt auch ersichtlich woher der Name 1–Komplement herrührt. Hier gilt nämlich für eine Zahl $k_1(z) = b_{n-1} \ldots b_1 b_0$

$$k_1(-z) = \neg b_{n-1} \ldots \neg b_1 \neg b_0, \tag{2.7}$$

wobei $\neg 0 = 1$ und $\neg 1 = 0$ die binäre Komplementbildung darstellt (Beweis von Gleichung 2.7 zur Übung).

2.2.1 Addition im 1–Komplement

Die 1–Komplement–Darstellung erlaubt die Rückführung der Subtraktion auf die Addition nach dem Muster $a - b = a + (-b)$. Dabei sind jedoch Sonderfälle zu beachten. Wir müssen vor allem das Bit b_n (links vom MSB) im Auge behalten, das bei einer Addition auch verändert werden kann. Addieren

wir zwei Zahlen im 1–Komplement, dann müssen wir dieses Bit an der Stelle b_0 hinzuaddieren (*Einserrücklauf*). Dieser Übertrag wird aber nicht zu den ursprünglichen Zahlen addiert, sondern zu den Zahlen mit jeweils "verdoppeltem" MSB. Dann ist nämlich ein *Überlauf* durch $b_n \neq b_{n-1}$ gekennzeichnet.

Beispiel: Die "triviale" Addition von $-0 + -0$ ist sehr lehrreich.

-0		11111111
-0	$\oplus$	11111111
Übertrag		**1**11111110
		(1)11111111
	$\oplus$	(1)11111111
	$\oplus$	00000001
-0		(1)**11**11111111

Es tritt ein Übertrag aber kein Überlauf ($b_n = b_{n-1}$) auf. Addieren wir $100 + 28$, so ist offensichtlich, dass wir den darstellbaren Zahlenbereich $[-127,\ 127]$ verlassen.

100		01100100
28	$\oplus$	00011100
Überlauf		**01**0000000

Wir sehen, dass $b_n \neq b_{n-1}$ ist (formal müssten wir den Übertrag 0 noch zu den um 0 "verdoppelten" Zahlen addieren, doch verändert dies das angegebene Resultat nicht), und stellen damit einen Überlauf fest. $\Diamond$

Die etwas komplizierte Eigenschaft des Einserrücklaufs ist eine Folge der doppelten Darstellung der 0 und damit ein ein Nachteil der 1–Komplement–Darstellung. Ein kleiner Kunstgriff eliminiert diesen Nachteil und wir gelangen zur *2–Komplement*–Darstellung für eine Zahl $z \in \mathbb{Z}$

$$z = -b_{n-1}2^{n-1} + \sum_{k=0}^{n-2} b_k B^k. \tag{2.8}$$

Wir sehen, dass hier die 0 nur eine Darstellung hat (warum?), was die Darstellung einer zusätzlichen Zahl ermöglicht. Dies reultiert in dem darstellbaren Intervall $[-2^{n-1},\ 2^{n-1} - 1]$.

Analog zum 1–Komplement ergibt sich für $0 \leq z \leq 2^{n-1} - 1$ die einfache binäre Darstellung mit $b_{n-1} = 0$ (positives Vorzeichen). Für $-2^{n-1} \leq z < 0$ sucht man das positive $z' = z + 2^{n-1}$ und stellt dieses in einfacher Binärdarstellung mit $b_{n-1} = 1$ (negatives Vorzeichen) dar.

Beispiel: Einige spezielle Zahlen im 2–Komplement für $n = 8$ (8–Bit–Zahlen).

$z_{(10)}$	$z'_{(10)}$	$k_2(z)_{(2)}$
1	-	00000001
-1	127	11111111
127	-	01111111
-127	1	10000001
-128	0	10000000

Tabelle 2.4: 8–Bit–Zahlen im 2–Komplement. ◊

Auch hier kann man aus Tabelle 2.3 auf die (etwas kompliziertere) Bildungsvorschrift negativer Zahlen $k_2(z) = b_{n-1} \ldots b_1 b_0$ schliessen. Man sieht, dass

$$k_2(-z) = \neg b_{n-1} \ldots \neg b_1 \neg b_0 \oplus 0 \ldots 01. \tag{2.9}$$

2.2.2 Addition im 2–Komplement

Der Einserrücklauf wird hier schon bei der Komplementbildung berücksichtigt, was zu Vereinfachung der Addition/Subtraktion führt, wenn man hierbei das MSB der beiden Summanden zu einer zusätzlichen Stelle "verdoppelt". Wir können dann prüfen, ob ein Überlauf stattfindet, was wir wieder an der Bedingung $b_n \neq b_{n-1}$ erkennen. Tritt kein Überlauf auf, so ist das Ergebnis direkt an den Stellen der Summe $b_{n-1} \ldots b_1 b_0$ abzulesen.

Beispiel: Wir berechnen $17 - 2 = 17 + (-2)$:

2		00000010
-2		11111110
17		(0)00010001
-2	$\oplus$	(1)11111110
15		(1)**00**0001111

Wir sehen, dass $b_n = b_{n-1}$ (b_{n+1} bleibt unberücksichtigt), und damit kein Überlauf auftritt. Das Ergebnis ist positiv (warum?) und wir können die binäre Summe direkt als korrektes Resulat $15_{(10)}$ identifizieren. ◊

Der wesentliche Punkt bei der Addition in beiden Darstellungen ist die einfache Feststellung eines Überlaufs mittels Vergleich von b_n mit b_{n-1}. Das Ergebnis dieses Vergleichs wird in vielen *Rechenwerken* (Basis aller digitalen

Rechenanlagen) als *Flag* (Bit) in einem Statusregister angeführt, das Auskunft über die letzte Rechenoperation gibt. Trat dabei ein Überlauf auf, so wird das *Overflow–Flag* automatisch gesetzt. Das Steuerprogramm (Software oder Hardware) kann so einfach erkennen, ob das Reultat gültig ist oder nicht.

2.2.3 Multiplikation von Binärzahlen

Bei der Multiplikation von Binärzahlen können wir auf bekannte Methoden aus dem Dezimalsystem zurückgreifen. Die bekannteste Methode ist sicher die folgende.

Beispiel: Wir berechnen das Produkt $7_{(10)} \times 3_{(10)}$ in 4–Bit–Binärdarstellung.

7×3	$0111 \otimes 0011$
	0000
	0000
	0111
$\oplus$	0111
21	10101

Die Multiplikation wird also mit den einzelnen Zahlen des Multiplikators ausgeführt und die Einzelergebnisse entsprechend ihrer Wertigkeiten addiert. Man kann die Binärmultiplikation relativ einfach auf mehrere Additionen zurückführen, da bei den einzelnen Multiplikationen nur zwei Teilergebnisse möglich sind: der Multiplikand oder die 0. Diese Eigenschaften werden bei der Konstruktion von Multiplizierern (Hardware) auch ausgenützt. $\Diamond$

Ein Hauptproblem bei der Multiplikation ist, dass die Anzahl der Stellen des Produkts $p = d + k$ ist (d Stellenanzahl des Mutliplikanden, k Stellenanzahl des Multiplikators, Beweis?), was auch heute noch zu Problemen in Rechnern führen kann, da Zahlen üblicherweise mit konstanter Stellenanzahl dargestellt werden (Hardware). Das Problem kann natürlich mit Software gelöst werden, doch führt dies wiederum zu Geschwindigkeitsverlusten.

Obwohl die Multiplikation von Zahlen ein triviales Problem zu sein scheint, schadet auch hier längeres Nachdenken oder Geschichtsstudium nicht. Die Ägypter haben nämlich schon eine sehr effiziente Methode der Multiplikation entwickelt, die sogar schon Binärarithmetik beinhaltet. Wir wollen die *ägyptische Multiplikation* an folgendem Beispiel erläutern:

Beispiel: Wir berechnen das Produkt 22×21.

Man schreibt also einfach die beiden zu multiplizierenden Zahlen nebeneinander, verdoppelt in jeder neuen Zeile die linke Zahl und halbiert die rech-

22	21
~~44~~	~~10~~
88	5
~~176~~	~~2~~
352	1
462	

te (ganzzahlig), streicht jene Zeilen in denen die rechte Zahl gerade ist und addiert die verbleibenden linken Zahlen. Diese Methode beinhaltet implizit auch die uns schon bekannte *sukzessive Division* (wo genau?). ◊

Diese Methode hat die für uns besondere Eigenschaft, dass wir neben der Addition nur Verdopplung und Halbierung von Zahlen benötigen. Diese beiden Operationen lassen sich aber mit Binärzahlen sehr einfach realisieren, da eine Verschiebung um eine Stelle nach links (*shift left*) einer Verdopplung und eine Verschiebung nach rechts (*shift right*) einer Halbierung entspricht.

Beispiel:

$$27 \times 2 = shl(11011) = 110110 = 54$$

$$33 \div 2 = shr(100001) = 10000 = 16$$

Bei den Operationen *shl* und *shr* ist zu beachten, dass die freiwerdende Stelle mit dem Bit 0 belegt wird. Im Prinzip haben wir mit diesen beiden Operationen schon Befehle einer gängigen Maschinensprache (*Assemblerprogrammierung*) kennengelernt. ◊

2.3 Darstellung rationaler Zahlen

Für die meisten numerischen Berechnungen ist die Verwendung rationaler Zahlen $q \in \mathbb{Q}$ ("Kommazahlen") unumgänglich. Wenn wir uns die Gleichung 2.1 in Erinnerung rufen, so erkennt man, dass eine einfache Erweiterung die Darstellung (gewisser) rationaler Zahlen ermöglicht.

$$q = \sum_{k=-m}^{n-1} b_k B^k = b_{-m}B^{-m} + b_{-m+1}B^{-m+1} + \ldots + b_0 + \ldots + b_{n-1}B^{n-1}. \quad (2.10)$$

2.3.1 Festpunktdarstellung

Da sich jede rationale Zahl q als Summe einer natuerlichen Zahl $u \in \mathbb{N}_0$ und einer rationalen Zahl v $(0 \leq v < 1)$ mit $q = u + v$ angeben lässt, können

wir ein einfaches Darstellungsformat für rationale Zahlen, die *Festpunkt–*Darstellung, angeben.

$$q_{(10)} = (u_n u_{n-1} \dots u_0 \odot v_{-1} \dots v_{-m})_{(2)},$$

wobei u_n das Vorzeichenbit ist, u eine ganze n–Bit Zahl und v eine rationale m–Bit Zahl ist. Für die Festpunktdarstellung verwenden wir die Schreibweise $(n.m)$ für Zahlen ohne und $(-n.m)$ für Zahlen mit Vorzeichenbit.

Beispiel: Wir wandeln die Zahl 0.125 in eine Binärzahl in (2.4)–Festpunktdarstellung um.

$$0.125 = 0 + \frac{1}{8} = 0 + 2^{-3} = 00.0010$$

◊

Im obigen Beispiel war die Umwandlung sehr einfach, da die Zahl durch eine einzige (negative) 2–Potenz darstellbar ist. Für beliebige rationale Zahlen wenden wir analog zur sukzessiven Division die *sukzessive Multiplikation* mit der Basis B an, wobei der Reihe nach die Ziffern $b_{-1} \dots b_{-m}$ abgespaltet werden (Gleichung 2.10).

Beispiel: Wir wandeln die Zahl $-1.3_{(10)}$ in eine Binärzahl in (-2.8)–Festpunktdarstellung um. Zuerst ermitteln wir die Nachkommastellen.

m	m'	b_k
0.3	0.6	0
0.6	1.2	1
0.2	0.4	0
0.4	0.8	0
0.8	1.6	1
0.6	1.2	1

Tabelle 2.5: Sukzessive Multiplikation mit 2.

Die Rolle des Rests übernimmt hier die Vorkommastelle, die allgemein von 0 bis $B-1$ gehen kann (Beweis?). Die erste abgespaltene Ziffer b_k entspricht b_{-1}. Weiters ist zu beachten, dass das Verfahren nur dann terminiert, wenn die rationale Zahl als Summe von negativen Zweierpotenzen darstellbar ist. In unserem Beispiel ist dies nicht der Fall, daher erhalten wir eine *periodische* Binärzahl, die laut Voraussetzung aber auf 8 Stellen beschränkt bleibt.

$$0.3 = 0.0\dot{1}00\dot{1} = 0.0\overline{1001}$$

Unter Beachtung des Vorzeichens und der vollständigen Zahl -1.3 ergibt sich daher $-1.3_{(10)} = 101.01001100$. (wie gross ist der Fehler?) ◊

Offensichtlich können wir die Zahl im obigen Beispiel nicht exakt darstellen, auch wenn wir beliebig viele Stellen zur Verfügung hätten. Dies ist eine sehr wesentliche Erkenntnis für numerische Berechnungen in digitalen Rechenanlagen, die in der Praxis aber oft vergessen wird.

Es gibt natürlich Versuche, die Genauigkeit der Berechnungen mit Zahlen in Festpunktdarstellung zu verbessern. Eine Möglichkeit ist die *Normierung* (Volkert, 1999), bei der die Nachkommastellen vor jedem Rechenschritt solange nach links geschoben wird $b_{-1} \neq 0$. Dies führt zwar meist zu genauerer Berechnung, dafür ist aber erforderlich, dass alle Normierungen aufgezeichnet werden, um dann das Resultat korrekt *denormieren* zu können.

2.3.2 Gleitpunktdarstellung

Eine weitere Möglichkeit der genaueren (effizienteren) Darstellung von rationalen Zahlen ist die Gleitpunktdarstellung (*floating point representation*). Die Grundidee dieser Darstellung wird erkennbar, wenn man die Darstellung einer sehr kleinen Zahl z. B. 2^{-16} betrachtet. In Festpunktdarstellung würden wir dafür mindestens 16 Nachkommastellen benötigen. Was aber, wenn wir eine Zahl q in der Form

$$q = MB^C \tag{2.11}$$

darstellen würden? Wählen wir $B = 2$ und $M = 1.0$, dann würden wir zur Darstellung von 2^{-16} nur 5 Bit (C) plus 1 Bit (M) benötigen. Alternativ könnten wir $m = 2.0$ wählen und so dieselbe Zahl auf mehrere Arten darstellen. C wird in dieser Darstellung meist als *Charakteristik* bezeichnet. Um auch negative Hochzahlen darstellen zu können, würden wir für eine 5–Bit Zahl $C' = C - 16$ wählen, d. h. unsere Zahl würde dann als 1 00000 codiert.

Diesen Prinzipien folgend wurden einige Floating–point–Formate standardisiert. Eine verbreiteter Standard ist das *32–Bit–IEEE–Format* [2]

$$q_{(2)} = v\ c_7 \ldots c_0\ m_{22} \ldots m_0$$

mit

$$q_{(10)} = (-1)^v 2^{C-127}(1.M),$$

wobei v das Vorzeichenbit, C die 8–stellige Charakteristik, und M die 23–stellige *Mantisse* ist. Zu beachten ist neben dem *bias* (oder *offset*) der Charakteristik (zur Darstellung negativer Hochzahlen) die Annahme, dass die

[2] IEEE (zu Deutsch "ei–tripl–i") steht für *Institute of Electrical and Electronics Engineers* und ist weltweit eine der grössten technisch–wissenschaftlichen Vereinigungen.

einzige Vorkommastelle der Mantisse *immer* gleich 1 ist, was eine effizientere Codierung erlaubt.

Die Bezeichnung "floating point" ist insofern etwas irreführend, da in dieser Darstellung das Komma gar nicht explizit auftritt. Damit soll aber ausgedrückt werden, dass so unterschiedliche Zahlen wie 2.5 und -233783.485482 mit derselben Darstellung repräsentiert werden.

Die 0 als eine der wichtigsten Zahl ist in diesem Format nicht exakt darstellbar (warum?), daher hat man für diese und einige andere spezielle Zustände folgende Darstellungen definiert.

$$C = 0, M = 0 \rightarrow q = (-1)^v 0.0$$
$$C = 0, M \neq 0 \rightarrow q = (-1)^v 2^{-126}(0.M)$$
$$C = 255, M = 0 \rightarrow q = (-1)^v \infty$$
$$C = 255, M \neq 0 \rightarrow q = NaN$$

Hier ist NaN (*Not a Number*) und symbolisiert ungültige Ergebnisse (z. B. Division durch 0). Die Darstellung mit 32 Bits erlaubt nur eine eingeschränkte Genauigkeit (wie gross ist diese?), daher wurde auch ein *64–Bit–IEEE–Format* definiert, wobei

$$q_{(2)} = v\ c_{10} \ldots c_0\ m_{51} \ldots m_0$$

mit

$$q_{(10)} = (-1)^v 2^{C-1023}(1.M).$$

Beispiel: Wir stellen die Zahl -3.5 im 32–Bit–IEEE–Format und im 64–Bit–IEEE–Format dar. Dazu müssen wir diese Zahl zuerst in die gewünschte Form bringen.

$$-3.5 = -1 \times 2^1 \times 1.75$$

und daher für 32–Bit–IEEE

$$-3.5_{(10)} = 1\ 10000000\ 11000000000000000000000_{(32,IEEE)}$$

und für 64–Bit–IEEE

$$-3.5_{(10)} = 1\ 10000000000\ 1100000000000000000000000000000000000000000000000000_{(64,IEEE)}$$

Aufbauend auf diesen Formaten (es gibt andere ähnliche) exisitieren heute in fast allen Programmiersprachen die Typen *float* (einfache Genauigkeit, meist 32 Bit) und *double* ("doppelte" Genauigkeit, meist 64 Bit) für die Darstellung von rationalen Zahlen.

Kapitel 3

Logische Schaltungen

Wir haben im letzten Kapitel gesehen, wie man Zahlen (binär) codieren kann, sodass mit diesen Zahlen Berechnungen in digitalen Rechenanlagen durchgeführt werden können. Nun wenden wir uns der Frage zu, welche Bauteile wir zu diesem Zweck benötigen und wie diese strukturiert sind. Wir werden sehen, dass wir numerische Berechnungen auf einfache logische Verknüpfungen zurückführen können, die auch im Alltag meist grössere Bedeutung haben als das Rechnen mit Zahlen. Betrachten wir als Beispiel folgende Aussage:

Wenn die Sonne scheint und es windstill ist, fahre ich an den See.

Das wesentliche an dieser gesamten Aussage ist, dass die einzelnen Aussagen nur zwei *Wahrheitswerte* annehmen können, nämlich *wahr* (W) oder *falsch* (F) (z. B. die Sonne scheint (W) oder sie scheint nicht (F)). Diese Tatsache lässt uns natürlich sofort ahnen, dass wir Aussagen dieses Typs binär codieren können und damit in digitalen Rechenanlagen verknüpfen und bearbeiten können. Den theoretischen Unterbau zu dieser Verknüpfung von Aussagen liefert ein Teilgebiet der *Mathematischen Logik*, das als *Aussagenlogik* oder auch als *Formale Logik* bezeichnet wird (Mendelson, 1982).

3.1 Aussagenlogik

Gehen wir alle Möglichkeiten für obiges Beispiel durch, so können wir eine *Wahrheitswertetabelle* (kurz Wahrheitstabelle) aufstellen (Tabelle 3.1). Wir bezeichnen die Aussage "Sonne scheint" mit **A**, die Aussage "windstill" (= "kein Wind") mit $\neg$ **B**, wobei das Symbol "$\neg$" die *Negation* darstellt, und die Aussage "See fahren" mit **C**.

A	¬ **B**	**C**
F	F	F
F	W	F
W	F	F
W	W	W

Tabelle 3.1: Wahrheitstabelle für den Seeausflug.

3.1.1 Aussagenlogische Operationen

Aus Tabelle 3.1 erkennen wir, dass die Negation den Wahrheitswert einer Aussage einfach umkehrt. Ist z. B. ¬ **B** wahr (es ist windstill), dann ist **B** falsch (kein Wind). Die Negation ist das einfachste Beispiel einer *aussagenlogischen Operation*. Obige Wahrheitstabelle beschreibt die Operation der *Konjunktion* (logisches UND) von **A** und **B'** = ¬ **B**, die meist als **A** & **B'** dargestellt wird. Wir können erkennen, dass **A** & **B'** nur dann wahr wird, wenn **A** und **B'** wahr sind, was eine Formalisierung unseres Seeausflugs darstellt.

Eine weitere wichtige aussagenlogische Operation ist die *Disjunktion* (auch Adjunktion) von zwei Aussagen **A** und **B**, die mit **A** ∨ **B** bezeichnet wird. Die Wahrheitstabelle ist in Tabelle 3.2 dargestellt.

A	**B**	**A** ∨ **B**
F	F	F
F	W	W
W	F	W
W	W	W

Tabelle 3.2: Wahrheitstabelle für die Disjunktion.

Diese Operation wird auch als logisches ODER (auch OR) bezeichnet. Wir sehen, dass es genügt, dass **A oder B** wahr werden, damit die Disjunktion wahr wird. Auch bei diesen elementaren Operationen ist zu beachten, dass die sprachliche Verwendung nicht unbedingt mit der mathematischen Bedeutung übereinstimmen muss. Im Falle der Disjunktion verwenden wir das Wort "oder" sprachlich nicht nur *inklusiv* (Tabelle 3.2), sondern auch *exklusiv* wie z. B. in der Aussage "ich kaufe Käse oder Wurst".

Letzteres entspricht der Operation "exklusives ODER" (auch XOR) **A** + **B**, die wir in Tabelle 3.3 sehen.

Eine weitere Operation ist die Subjunktion **A** → **B** (Tabelle 3.4), die fälschlicherweise oft auch als Implikation bezeichnet wird.

A	**B**	**A + B**
F	F	F
F	W	W
W	F	W
W	W	F

Tabelle 3.3: Wahrheitstabelle für XOR.

A	**B**	**A $\rightarrow$ B**
F	F	W
F	W	W
W	F	F
W	W	W

Tabelle 3.4: Wahrheitstabelle für die Subjunktion.

Die Subjunktion ist ein Beispiel dafür, dass die umgangssprachliche Bedeutung von $\mathbf{A} \rightarrow \mathbf{B}$ "wenn **A**, dann **B**" nicht unbedingt auf die aussagenlogische Operation schliessen lässt. Trotzdem ist diese Definition "logisch", da für den Fall, dass **A** falsch ist, die Subjunktion immer wahr ist (mit der Idee, dass ich mit einer falschen Voraussetzung alles behaupten kann). Für den Fall, dass **B** wahr ist, ist die Subjunktion auch immer wahr (wenn die Folgerung wahr ist, ist die Voraussetzung unerheblich). Die dritte Zeile der Wahrheitstabelle 3.4 entspricht der umgangssprachlichen Verwendung (finden Sie eine zur Subjunktion äquivalente Verknüpfung mit $\neg$ und $\vee$).

Wir können also zwei Aussagevariablen **A** und **B** auf verschiedene Arten verknüpfen (wieviele gibt es?), und jede dieser Verknüpfungen mit einem *Junktor* (Verknüpfungszeichen) darstellen. Bereits bekannte Junktoren sind die Zeichen $\neg$, &, $\vee$, $+$, und $\rightarrow$.

Eine Verknüpfung von *Aussagevariablen* mit ein oder mehreren Junktoren nennen wir *Aussageform*. Eine Aussageform definiert eine *aussagenlogische Wahrheitsfunktion*.

Beispiel: Den Wert einer Aussageform können wir mit einer Wahrheitstabelle berechnen, indem wir den einzelnen Aussagevariablen- und formen Wahrheitswerte zuweisen. Wir berechnen die Aussageform $(\neg \mathbf{A} \,\&\, \mathbf{B}) \vee (\mathbf{A} \,\&\, \neg \mathbf{B})$ in Tabelle 3.5.

(Finden Sie eine äquivalente (einfachere) Aussageform.) $\Diamond$

A	**B**	$\neg$ **A** & **B**	**A** & $\neg$ **B**	($\neg$ **A** & **B**) $\vee$ (**A** & $\neg$ **B**)
W	W	F	F	F
W	F	F	W	W
F	W	W	F	W
F	F	F	F	F

Tabelle 3.5: Wahrheitstabelle zur Berechnung einer Aussageform.

3.1.2 Aussagenlogische Grundbegriffe

Wir betrachten noch einige wichtige Definitionen der Aussagenlogik.

Definition 3.1.1 *Eine Aussageform* **A** *heisst* Tautologie*, wenn sie für jede beliebige Bewertung ihrer Variablen immer wahr ist. Man sagt dann:* **A** *ist* allgemeingültig.

Beispiel: **A** $\rightarrow$ **A** ist eine Tautologie. Hier wird auch klar, warum die Subjunktion für **A** ist falsch als wahr definiert ist (ist **A** & **A** allgemeingültig?). $\Diamond$

Definition 3.1.2 *Eine Aussageform* **A** *heisst* Kontradiktion*, wenn sie für jede beliebige Bewertung ihrer Variablen immer falsch ist. Man sagt dann:* **A** *ist ein* Widerspruch.

Beispiel: **A** & $\neg$ **A** ist eine Kontradiktion. (ist **A** $\rightarrow$ $\neg$ **A** ein Widerspruch?). $\Diamond$

Definition 3.1.3 *Eine Aussageform* **A** impliziert *die Aussageform* **B** *genau dann, wenn eine Variablenbewertung, die* **A** *wahr macht, auch* **B** *wahr macht.*

Beispiel: **A** & **B** impliziert **A** (impliziert (**A** & **B**) (**A** $\vee$ **B**)?). $\Diamond$

Definition 3.1.4 *Zwei Aussageformen* **A** *und* **B** *heissen logisch* äquivalent*, wenn jede Variablenbewertung* **A** *und* **B** *gleichwertig macht.*

Wir haben den Begriff der Äquivalenz schon weiter oben angesprochen und intuitiv das Richtige gemeint. Eine äquivalente Definition zu Definition 3.1.4 ist die, dass die Wahrheitswerte von **A** und **B** in der Wahrheitstabelle zeilenweise übereinstimmen müssen.

Beispiel: Ist die *Bijunktion* **A** $\leftrightarrow$ **B** äquivalent zu (**A** $\rightarrow$ **B**) & (**B** $\rightarrow$ **A**)? Dazu stellen wir die Wahrheitstabelle auf, in der auch die Bijektion definiert ist.

$\mathbf{A}$	$\mathbf{B}$	$\mathbf{A} \leftrightarrow \mathbf{B}$	$\mathbf{A} \rightarrow \mathbf{B}$	$\mathbf{B} \rightarrow \mathbf{A}$	$(\mathbf{A} \rightarrow \mathbf{B})$ & $(\mathbf{B} \rightarrow \mathbf{A})$
W	W	W	W	W	W
F	W	F	W	F	F
W	F	F	F	W	F
F	F	W	W	W	W

Tabelle 3.6: Wahrheitstabelle zur Überprüfung einer Äquivalenz.

Die genannten Aussagenformen sind äquivalent (was wir auch erwartet haben). ◊

Eine von vielen wichtigen Äquivalenzen sind die *deMorganschen Regeln*:

$$\neg(\mathbf{A} \vee \mathbf{B}) \equiv \neg\mathbf{A} \;\&\; \neg\mathbf{B} \tag{3.1}$$

$$\neg(\mathbf{A} \;\&\; \mathbf{B}) \equiv \neg\mathbf{A} \vee \neg\mathbf{B}, \tag{3.2}$$

die wie die *Idempotenz*

$$(\mathbf{A} \;\&\; \mathbf{A}) \equiv \mathbf{A} \quad \mathbf{A} \vee \mathbf{A} \equiv \mathbf{A}, \tag{3.3}$$

oder die *doppelte Negation*

$$\neg\neg\mathbf{A} \equiv \mathbf{A} \tag{3.4}$$

für Umformungen von Aussageformen Verwendung finden, wie z. B. für den Nachweis der Kommutativität, Assoziativität und Distributivität von Aussageformen bezüglich spezifischer Junktoren.

3.1.3 Normalformen

Bisher sind wir beispielhaft immer von zwei Aussagevariablen-(formen) ausgegangen, doch ist klar, dass wir Aussageformen aus beliebig vielen Variablen mit allen möglichen Junktoren bilden können.

Beispiel: Eine Aussageform mit den Variablen $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ und $\mathbf{D}$ und den Junktoren $\neg$, &, $\vee$, $+$ und $\rightarrow$.

$$(\neg\mathbf{A} + \mathbf{B}) \rightarrow ((\mathbf{C} \vee \mathbf{D}) \;\&\; (\neg\mathbf{B} \vee \mathbf{D}))$$

Es stellt sich die Frage, ob man die Darstellung beliebiger Aussageformen vereinheitlichen kann, da dies eine systematische Bearbeitung dieser Aussagen erleichtern würde. Dazu definieren wir

Definition 3.1.5 *Ein* Konjunktionsterm *bezeichnet eine Aussageform, die entweder aus einer negierten oder nicht–negierten Variablen, oder der Konjunktion zweier oder mehrerer negierter oder nicht–negierter Variablen besteht.*

Beispiel: Die Terme **A**, $\neg$ **B** und **C** & **A** & $\neg$ **D** sind Konjunktionsterme. $\Diamond$

Definition 3.1.6 *Sind* **A** *und* **B** *zwei Konjunktionsterme, so "ist* **A** *in* **B** *enthalten", wenn alle negierten und nicht–negierten Variablen von* **A** *auch in* **B** *vorkommen.*

Beispiel: Die Aussageform **A** & $\neg$ **B** ist in **A** & $\neg$ **B** & **C** enthalten, während **A** & **B** nicht enthalten ist. $\Diamond$

Definition 3.1.7 *Eine Aussageform liegt in* Disjunktiver Normalform *(DNF) vor, wenn diese entweder ein Konjunktionsterm, oder die Disjunktion von zwei oder mehreren Konjunktionstermen, wobei keiner in einem anderen enthalten sein darf, ist.*

Von grosser Bedeutung ist der folgende Spezialfall einer DNF:

Definition 3.1.8 *Eine DNF heisst* vollständig*, wenn in jedem Konjunktionsterm* jede *Aussagevariable der Aussageform vorkommt. Die Konjunktionsterme einer vollständigen DNF heissen* Minterme.

Beispiel: Die DNF **A** & $\neg$ **B** $\vee$ $\neg$ **B** & **C** ist nicht vollständig. Die DNF **A** & $\neg$ **B** & **C** $\vee$ **A** & **B** & **C** ist vollständig. $\Diamond$

In dualer Weise lassen sich die *Konjunktive Normalform* (KNF) und die *Vollständige* KNF definieren, wenn man die Begriffe Disjunktion und Konjunktion vertauscht. Die Disjunktionsterme einer vollständigen KNF heissen *Maxterme.*

Mit diesen Definitionen kann man nun zeigen:

Theorem 3.1.1 *Jede nicht–kontradiktorische Aussageform kann auf eine vollständige DNF gebracht werden.*

Dieser Satz spielt eine zentrale Rolle in der Konstruktion von logischen Schaltungen. Haben wir nämlich eine Wahrheitsfunktion (= gewünschte Funktion einer logischen Schaltung) in Form einer Wahrheitstabelle vorliegen, so können wir daraus einfach eine Aussageform in DNF ableiten. Bisher haben

x_1	x_2	$f(x_1, x_2)$
W	W	F
F	W	W
W	F	W
F	F	W

Tabelle 3.7: Wahrheitstabelle und Ableitung einer DNF.

wir das Aufstellen einer Wahrheitstabelle aufgrund einer Aussageform betrachtet, aber in der Praxis tritt fast immer der umgekehrte Fall auf (man kennt die Funktion, aber nicht die Aussageform).

Beispiel: Wir wollen die Wahrheitsfunktion in Tabelle 3.7 realisieren.

Dazu betrachten wir jene Zeilen, deren $f(x_1, x_2)$ wahr ist, und bilden aus den Variablen Konjunktionsterme, wobei die Variable negiert wird, wenn ihr Wert falsch ist. Wir erhalten so

$$\neg \mathbf{A} \ \& \ \mathbf{B} \vee \mathbf{A} \ \& \ \neg \mathbf{B} \vee \neg \mathbf{A} \ \& \ \neg \mathbf{B},$$

$\Diamond$

und sehen, dass jeder Konjunktionsterm genau dann wahr wird, wenn die Variablen die Wahrheitswerte annehmen, die die Wahrheitsfunktion wahr werden lassen. Die Disjunktion dieser Aussagen wird dann wahr, wenn zumindest einer der Konjunktionsterme wahr wird. Eine Disjunktion mit einem Term, dessen Wert falsch ist (erste Zeile in Tabelle 3.7) verändert den Wert der Aussageform nicht, daher bleiben diese Zeilen unberücksichtigt. Wir erkennen überdies, dass aus einer Kontradiktion keine DNF abgeleitet werden könnte (warum?), was auch im Theorem 3.1.1 zum Ausdruck kommt.

Eine wichtiger Begriff, den wir später noch brauchen werden ist folgendermassen definiert.

Definition 3.1.9 *Ein Konjunktionsterm ψ heisst Primimplikant einer Aussageform* **A** *d. u. n. d., wenn ψ die Form* **A** *logisch impliziert und* **A** *durch keinen anderen in ψ enthaltenen Konjunktionsterm impliziert wird.*

Beispiel: $(\mathbf{A} \ \& \ \mathbf{B}) \vee (\mathbf{A} \ \& \ \neg \ \mathbf{B} \ \& \ \mathbf{C})$ ist eine DNF, die durch den Primimplikanten $\mathbf{A} \ \& \ \mathbf{C}$ impliziert wird (suchen Sie weitere Primimplikanten). Ein Wegnehmen einer beliebigen Variablen des angegebenen Primimplikanten impliziert die spezielle DNF nicht. $\Diamond$

3.1.4 Verknüpfungsbasen

Wir haben gesehen, wie wir von einer Wahrheitsfunktion zu einer Aussageform gelangen, haben uns aber nicht näher damit beschäftigt, welche und wieviele Junktoren wir dazu benötigen (implizit haben wir das, wo?). Mit

Definition 3.1.10 *Als* Verknüpfungsbasis *bezeichnen wir die Menge der Junktoren, die ausreicht, jede Wahrheitsfunktion als Aussageform darzustellen.*

Aus Theorem 3.1.1 folgt dann sofort, dass die Menge $\{\neg, \&, \vee\}$ eine Verknüpfungsbasis ist.

Auch diese Erkenntnis hat grosse praktische Bedeutung, da man im Prinzip nur drei verschiedene logische Bauteile benötigt, um alle Aussagenformen in Hardware abbilden zu können. Aber es kommt noch besser, denn man kann zeigen, dass auch folgende *einelementige* aussagenlogische Operationen eine Verknüpfungsbasis darstellen: die *Shefferbasis* $\{|\}$ und die *Piercebasis* $\{\downarrow\}$.

Die *Sheffersche Wahrheitsfunktion* ist in Tabelle 3.8 angegeben.

A	**B**	**A | B**
W	W	F
F	W	W
W	F	W
F	F	W

Tabelle 3.8: Wahrheitstabelle für die Sheffer–Funktion.

Man sieht, dass $\mathbf{A} \mid \mathbf{B} \equiv \neg(\mathbf{A} \,\&\, \mathbf{B})$ ist (weisen Sie es nach), was auch der Ausgangspunkt zum Nachweis der Eigenschaft der Verknüpfungsbasis ist. Wegen obiger Äquivalenz wird der entsprechende logische Baustein auch als NAND (not and) bezeichnet.

Die *Nicodsche Wahrheitsfunktion* ist in Tabelle 3.9 dargestellt.

A	**B**	**A $\downarrow$ B**
W	W	F
F	W	F
W	F	F
F	F	W

Tabelle 3.9: Wahrheitstabelle für die Nicod–Funktion.

Wenn wir aus Tabelle 3.9 eine DNF konstruieren, erhalten wir direkt $\mathbf{A} \downarrow \mathbf{B} \equiv \neg\mathbf{A} \,\&\, \neg\mathbf{B} \equiv \neg(\mathbf{A} \vee \mathbf{B})$. Letztere Aussageform gibt dem logischen Baustein für diese Funktion die Bezeichnung NOR (not or).

Die praktische Bedeutung dieser einelementigen Verknüpfungsbasen liegt darin, dass für die Realisierung *aller* logischer Operationen ein einziger Grundbaustein benötigt wird. Dies bringt vor allem in der Fertigung technologische Vorteile, heisst aber nicht, dass die Verwendung eines einzigen Bausteins optimal hinsichtlich der Anzahl der Bausteine für eine Schaltung ist.

3.2 Boolesche Algebra

Da es fast schon Allgemeinwissen ist, dass die *Boolesche Algebra* [1] eines der wichtigsten Werkzeuge zum Entwurf logischer Schaltungen ist, werden auch wir sie hier kurz betrachten. Auf den Zusammenhang zwischen der Booleschen Algebra und dem Schaltungsdesign wurde zuerst von *Shannon* in seiner Diplomarbeit hingewiesen!

Ganz allgemein wird in der Mathematik unter einer Algebra eine *Trägermenge* und ein oder mehrere Verknüpfungen von Elementen dieser Menge verstanden, die genau definierten Axiomen genügen. Wir wollen den Begriff der Verknüpfung genauer betrachten.

Definition 3.2.1 *Eine* n–stellige *(n–äre) Operation in einer Menge Y ist jede Funktion f, die jedes n–Tupel von Y mit einem Element $f \in Y$ verknüpft. Man sagt dann auch, dass Y* bezüglich f abgeschlossen *ist.*

Beispiel: In der Menge der reellen Zahlen ist $f(x) = x + 1$ eine unäre und $f(x, y) = x + y$ eine binäre Operation (*Anm.:* Der Begriff "binär" bezeichnet hier **nicht** die Zahldarstellung, sondern die Anzahl der verknüpften Elemente). Für die Zahlen $m, n \in \mathbb{N}$ ist $f(m, n) = m - n$ keine binäre Operation (warum?).

Somit können wir schon das Axiomensystem der Booleschen Algebra angeben.

Definition 3.2.2 *Als* Boolesche Algebra *bezeichnen wir eine Menge B, wenn in B zwei binäre Operationen $\wedge$ und $\vee$, eine unäre Operation $'$ und zwei spezielle Elemente 0 und 1 existieren, und folgende Axiome gelten:*

(1) $\forall x, y \in B: \quad x \vee y = y \vee x$

(2) $\forall x, y \in B: \quad x \wedge y = y \wedge x$

(3) $\forall x, y, z \in B: \quad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

[1] Sie geht zurück auf George Boole (1815–1864).

(4) $\forall x, y, z \in B: \quad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

(5) $\forall x \in B: \quad x \vee 0 = x$

(6) $\forall x \in B: \quad x \wedge 1 = x$

(7) $\forall x \in B: \quad x \vee x' = 1$

(8) $\forall x \in B: \quad x \wedge x' = 0$

(9) $0 \neq 1$

Wir nennen $x \wedge y$ das *Produkt* von x und y, $x \vee y$ die *Summe* von x und y, x' das *Komplement* von x, 0 das *Nullelement* und 1 das *Einselement*.

Eine Boolesche Algebra kennzeichnen wir mit $\langle B, \wedge, \vee, ', 0, 1\rangle$ (ein Sechstupel).

Beispiel: Wir kennen die Begriffe Summe und Produkt von unseren Rechenoperationen in den reellen Zahlen. Wenn wir nun 0.0 als Nullelement und 1.0 als Einselement verwenden, könnten wir annehmen, dass das Tupel $\langle \mathbb{R}, ., +, ', 0.0, 1.0\rangle$ eine Boolesche Algebra sei. Obwohl wir das Komplement noch nicht definiert haben, sehen wir schnell, dass diese Annahme Axiom (4) verletzt (warum?). $\Diamond$

Beispiel: $\mathbb{B}_0 = \langle \{\emptyset, \{\emptyset\}\}, \cap, \cup, ^-, \emptyset, \{\emptyset\}\rangle$ ist eine zweielementige Boolesche Algebra mit den bekannten mengentheoretischen Operatoren Durchschnitt, Vereinigung und Komplement (Nachweis?). $\Diamond$

Nun kann man auch relativ einfach nachweisen, dass die bezüglich Konjunktion, Disjunktion und Negation abgeschlossene Menge der Aussagen eine Boolesche Algebra ist. Bei einem exakten Beweis muss man allerdings berücksichtigen, dass z. B. das Nullelement die Aussageform $\mathbf{A}$ & $\neg\mathbf{A}$ sein könnte. Damit sind aber auch alle logisch äquivalenten Aussagen ein Nullelement, und somit gäbe es mehrere Nullelemente, was im Widerspruch zur Definition einer Booleschen Algebra ist.

Man führt daher Klassen von logisch äquvialenten Aussagen ein, was einfach zur Klasse der wahren K_W und zur Klasse der falschen Aussagen K_F führt. Damit ist $\langle \{K_W, K_F\}, \&, \vee, \neg, K_F, K_W\rangle$ eine Boolesche Algebra, und die gesamte Theorie darüber kann für die Aussagenlogik und somit auch für logische Schaltungen verwendet werden. Umgekehrt müssen damit natürlich auch alle Gesetze, die wir bisher in der Aussagenlogik kennengelernt haben, auch für Boolesche Algebren gültig sein (wie z. B. doppeltes Komplement, Idempotenz, deMorgan).

Wir werden in Hinkunft nicht ganz überraschend für K_W das Element **1** und für K_F das Element **0** verwenden. Das Produkt (Konjunktion) werden wir mit '.', die Summe mit '+', und das Komplement (Negation) mit $^-$

bezeichnen. Die Aussagevariablen werden wir mit Bezeichnern versehen, die meist in der Informatik für Variable verwendet werden (z. B. x_1, y_2, z_3).

Beispiel: Die DNF **A** & ¬ **B** ∨ ¬ **C** & **D** kann dann als $x_1\overline{x_2} + \overline{x_3}x_4$ geschrieben werden. ◊

3.3 Schaltnetze

Bisher haben wir Aussageformen und davon abgeleitete Wahrheitsfunktionen rein formal betrachtet, doch der Schritt zu einer logischen Realisierung dieser *Booleschen Funktionen* (im folgenden auch *Schaltfunktionen* genannt) ist nicht mehr allzu gross. Mit logischer Realisierung meinen wir hier die graphische Darstellung von Schaltfunktionen, die aus logischen Grundbausteinen zusammengesetzt sind. Die technische Realisierung solcher Bauteile würde den Rahmen dieser Vorlesung sprengen, da eine Beschäftigung damit weitreichende Kenntnisse der Elektrotechnik voraussetzen würde. Trotzdem wird an einigen Stellen versucht werden, Grundideen der technischen Realisierung zu vermitteln.

Ganz allgemein verstehen wir unter einer Schaltfunktion eine Abbildung der Form $f : \mathbb{B}^n \to \mathbb{B}^m$. Es werden also n *Eingänge* auf m *Ausgänge* abgebildet, wobei die Menge $\mathbb{B}$ aus den *booleschen* Elementen **0** und **1** besteht, die wir im vorigen Kapitel mit den Wahrheitswerten falsch und wahr identifiziert haben.

Die graphische Darstellung einer Schaltfunktion nennen wir *Schaltnetz*. Dies ist ein gerichteter, azyklischer Graph, dessen Knoten aus logischen Bausteinen (Schaltfunktionen) besteht. Ein Eingang ist dann eine Kante, an die ein *logisches Signal* angelegt wird. Ein Ausgang ist eine Kante, die ein logisches Signal als Resultat liefert. Das Grundelement jeder logischen Schaltung ist der *Schalter*, der die zwei booleschen Elemente als "Schalter geschlossen" und "Schalter geöffnet" realisiert. Diese logische Schaltung ist unabhängig von der technischen Realisierung, für die ein Schalter ein (mechanisches) Ventil, ein (elektromechanisches) Relais, oder ein (elektronischer) Transistor sein könnte.

In abstrakter Form können wir Schalter in einem Schaltnetz wie in Abb. 3.1 gezeigt darstellen.

Bei dieser Darstellung wird angenommen, dass ein einzelner Schalter einen Stromfluss unterbrechen kann. In beiden Schaltnetzen in Abb. 3.1 kann kein Strom fliessen, daher ist am Ausgang y_1 jeweils der Zustand "kein Strom" (0) festzustellen. Identifiziert man die Schalterzustände "geschlossen" (Strom fliesst) mit 1 und den Zustand "offen" (Stromfluss unterbrochen) mit 0, so kann man sofort die entsprechenden Wahrheitstabellen für die Schaltfunktio-

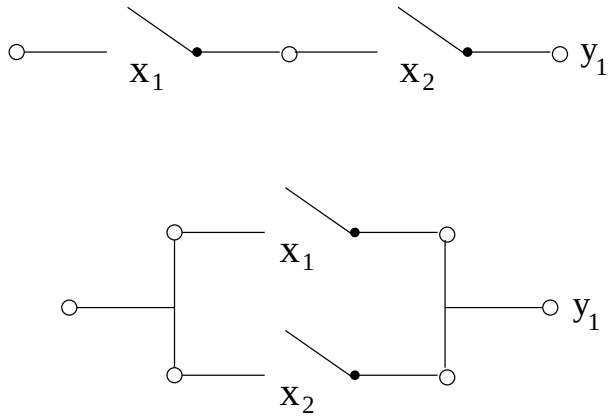

Abbildung 3.1: Einfache Schaltnetze für die Konjunktion $x_1 x_2 = y_1$ (oben) und die Disjunktion $x_1 + x_2 = y_1$ (unten).

nen angeben.

Weiters ist zu erkennen, dass die *Serienschaltung* von Schaltern einer Konjunktion, und die *Parallelschaltung* einer Disjunktion der jeweiligen Variablen entspricht. Ein etwas komplexeres Schaltnetz, wobei der Schalter für die Variable x_4 eine Negation darstellt, zeigt Abb. 3.2 (geben Sie die dazugehörige Schaltfunktion an).

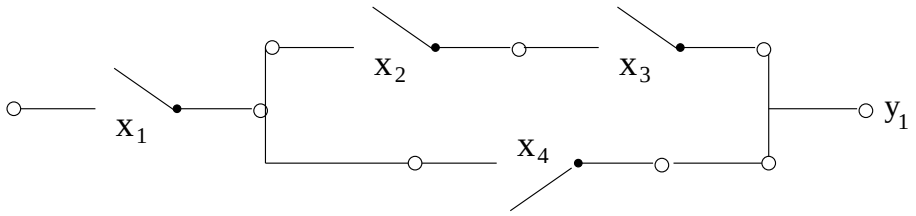

Abbildung 3.2: Ein Schaltnetz aus mehreren Schaltern.

In obigen Schaltnetzen wird erkennbar, dass die logischen Variablen den Schalterzuständen entsprechen, die einen Stromfluss beeinflussen. Streng genommen sind es also zwei physikalische Phänomene (Schalter, Strom), die ein solches Schaltnetz realisierbar machen. Eine elektronische Ausführung eines solchen Schaltnetzes verwendet Bauelemente aus Halbleitern (Transistoren, Dioden) als Schalter. Diese Schalter werden aber dann ihrerseits von Strömen (Spannungen) geschaltet, sodass man sämtliche logische Variablen als Strom/Spannung darstellen kann.

3.3.1 Logische Gatter

Um zu einer kompakteren Darstellung von Schaltnetzen zu gelangen, wird die Schalterdarstellung der logischen Basisfunktionen in logische Grundbausteine, die als *Gatter* bezeichnet werden, überführt. In Abb. 3.3 sind die wichtigsten Gatter (sie bilden eine Verknüpfungsbasis) dargestellt.

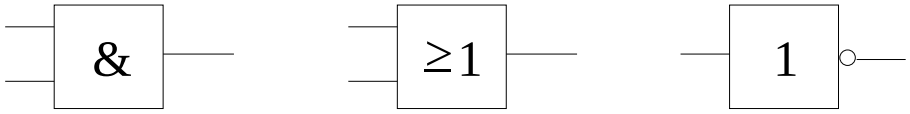

Abbildung 3.3: Die Gatter für Konjunktion (links), Disjunktion (Mitte) und Negation (rechts).

Diese Gatter sind Realisierungen logischer Funktionen der Form $f(x_1, x_2) = y_1$ (im Fall der Negation stimmt das nicht, warum?). Es werden also zwei boolesche Werte (Eingänge) auf einen Ausgang abgebildet. Hat man mehrere Ausgänge, so bildet man für jeden einzelnen Ausgang eine logische Funktion (Schaltnetz) aus diesen Gattern und kombiniert diese dann zu einer einzigen Schaltung (wobei man dann aus Kostengründen versucht, logische Untereinheiten, die für mehrere Ausgangsfunktionen verwendbar sind zu finden).

Mit den angegebenen Gattern kann dann das Schaltnetz aus Abb. 3.2 eleganter gezeichnet werden (Abb. 3.4).

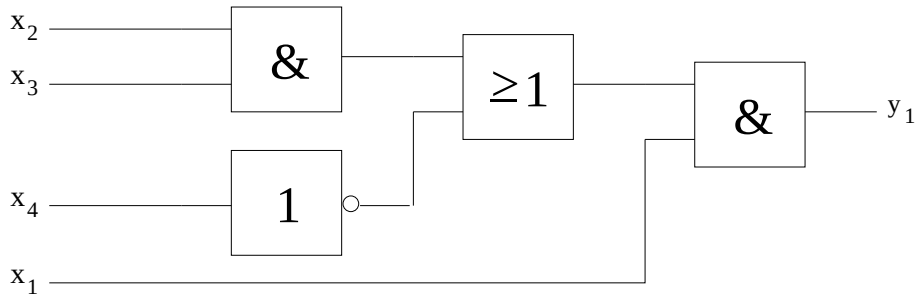

Abbildung 3.4: Die Gatterdarstellung des Schaltnetzes aus Abb. 3.2.

Obiges Schaltnetz kann in einem nächsten Abstraktionsschritt als neuer Grundbaustein verwendet werden (wenn genau diese logische Funktion häufig in einem Schaltnetz vorkommt), den wir (wahlweise) als **DR** bezeichnen (Abb. 3.5).

Durch immer weitere Abstraktionen kann man so immer komplexere Schaltungen aufbauen. Auch der modernste heutige Prozessor besteht nur aus solchen einfachen logischen Bausteinen (allerdings in sehr grosser Anzahl).

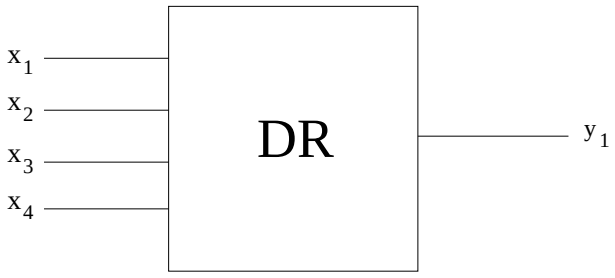

Abbildung 3.5: Der logische Baustein **DR** repräsentiert das Schaltnetz aus Abb. 3.4.

Heute hält man bei Schaltungen in VLSI (*Very Large Scale Integration*)–Technologie bei Bauteildichten von 10^5 Transistoren pro mm^2. Hierbei sind grosse Bauteilgruppen auf einem einzigen Chip integriert.

Obige Darstellung logischer Bausteine ähnelt schon sehr dem Erscheinungsbild eines elektronischen Chips. Die logischen Eingänge entsprechen dann elektrischen Eingängen, die mit Ausgängen anderer Chips oder Signalquellen (z. B. Sensoren wie Lichtschranke, Rauchmelder, Temperaturfühler) verbunden sind.

3.3.2 Komposition von Schaltnetzen

Wir betrachten die Möglichkeiten der Komposition von Gattern (oder Schaltnetzen) zu Schaltnetzen noch etwas genauer und führen eine funktionale Beschreibung der einzelnen Kompositionsformen ein, die eine direkte Repräsentation des Schaltnetzes ist (Broy, 1993). Zwei der wichtigsten Kompositionsformen – die parallele und die serielle Komposition – haben wir schon kurz in Kapitel 3.3 betrachtet.

Die *Parallele Komposition* von zwei als Schaltnetzen realisierten Schaltfunktionen

$$f_1 : \mathbb{B}^{n_1} \to \mathbb{B}^{m_1} \quad f_2 : \mathbb{B}^{n_2} \to \mathbb{B}^{m_2}$$

bezeichnen wir mit

$$(f_1 || f_2) : \mathbb{B}^{n_1+n_2} \to \mathbb{B}^{m_1+m_2}, \tag{3.5}$$

die in Abb. 3.6 abgebildet ist.

Die beiden Schaltnetze f_1 und f_2 werden hier unzusammenhängend kombiniert.

Die *Sequentielle (Serielle) Komposition* von zwei Schaltfunktionen

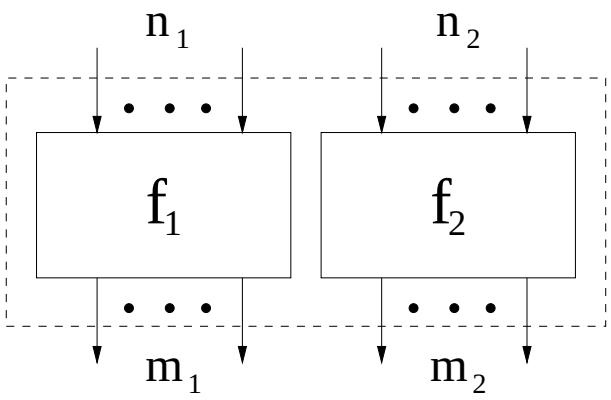

Abbildung 3.6: Parallele Komposition von Schaltnetzen.

$$f_1 : \mathbb{B}^n \to \mathbb{B}^k \quad f_2 : \mathbb{B}^k \to \mathbb{B}^m$$

bezeichnen wir mit

$$(f_1.f_2) : \mathbb{B}^n \to \mathbb{B}^m. \tag{3.6}$$

Die Ausgänge des Schaltnetzes f_1 werden also zu den Eingängen des Schaltnetzes f_2 (Abb. 3.7).

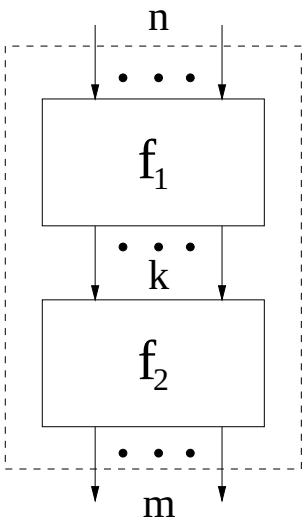

Abbildung 3.7: Serielle Komposition von Schaltnetzen.

Die *Projektion* schaltet genau eine von n Eingangsleitungen auf den Ausgang durch. Wir schreiben dafür

$$\Pi_i^n : \mathbb{B}^n \to \mathbb{B}. \tag{3.7}$$

Die in Abb. 3.8 gezeigte Projektion wird in der Schaltungstechnik als *Multiplexer* bezeichnet, der eben aus mehreren Eingängen einen auswählt. Selektiert man jeden der Eingänge für eine definierte Zeiteinheit (*Time Slice*), so ist dieses Schaltnetz Grundbaustein eines *Time–Division–Multiplexing*–Systems, wie es bei digitaler Übertragung von Daten verwendet wird. Ein *Multi–Tasking*–Betriebssystem basiert auf einem analogen Mechanismus, der da jedoch meist in Software realisiert ist.

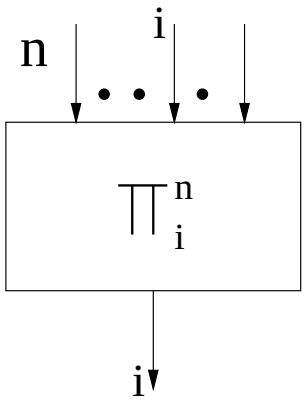

Abbildung 3.8: Projektion (Multiplexer).

Mit obigen drei Kompositionsformen können wir bereits alle Schaltnetze beschreiben, doch zur Vereinfachung der Darstellung von Schaltnetzen geben wir noch einige Kompositionsformen an, die aber alle aus den bisherigen abgeleitet werden können.

Das *Tupeling* von zwei Schaltfunktionen

$$f_1 : \mathbb{B}^n \to \mathbb{B}^{m_1} \quad f_2 : \mathbb{B}^n \to \mathbb{B}^{m_2}$$

wird definiert durch

$$[f_1, f_2] : \mathbb{B}^n \to \mathbb{B}^{m_1+m_2}. \tag{3.8}$$

Wie in Abb. 3.9 dargestellt, werden n Leitungen als Eingang für zwei (oder mehrere) Schaltfunktionen verwendet.

Für das Tupeling der Schaltfunktionen f_1 und f_2 gilt

$$[f_1, f_2] = [\Pi_1^n, \ldots, \Pi_n^n, \Pi_1^n, \ldots, \Pi_n^n].(f_1||f_2), \tag{3.9}$$

was aber offensichtlich eine wesentlich aufwendigere Darstellung ist (finden Sie alternative Darstellungen der Schaltfunktion in Gleichung 3.9). Auch

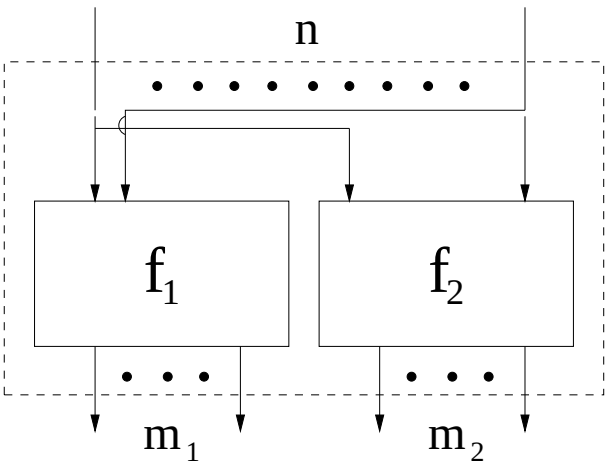

Abbildung 3.9: Tupeling zweier Schaltnetze.

die *Identität*, die *Permutation* und die *Verzweigung* lassen sich als Tupeling von Projektionsfunktionen darstellen.

Die Identität

$$I_n : \mathbb{B}^n \to \mathbb{B}^n$$

entspricht einer einfachen Leitung, die am Ausgang i einfach dem Eingang i entspricht. Die Identität kann als

$$I_n = [\Pi_1^n, \ldots, \Pi_n^n] \tag{3.10}$$

dargestellt werden.

Die Permutation

$$P_n : \mathbb{B}^n \to \mathbb{B}^n$$

reiht den ersten Eingang hinter den letzten und kann mit

$$P_n = [\Pi_2^n, \ldots, \Pi_n^n, \Pi_1^n] \tag{3.11}$$

beschrieben weden. Durch sequentielle und parallele Komposition von (dieser spezifischen) Permutation mit Identitäten lassen sich alle möglichen Permutationen beschreiben.

Die Verzweigung

$$V_n : \mathbb{B}^n \to \mathbb{B}^{2n}$$

verdoppelt (verzweigt) jede einzelne Leitung und damit jedes Tupel von Eingängen wie folgt:

$$V_n = [\Pi_1^n, \ldots, \Pi_n^n, \Pi_1^n, \ldots, \Pi_n^n]. \tag{3.12}$$

Zwei wichtige Spezialfälle wollen wir auch noch kurz anführen. Die *Senke* S entspricht einer Abbildung $S : \mathbb{B}^n \to \mathbb{B}^0$, mit $\mathbb{B}^0$ als leerer Menge. In einer logischen Schaltung entspricht dies einem (oder mehreren) "hängenden" Ausgang einer Schaltfunktion, d.h. ein Ausgang, der mit keinem weiteren Eingang verbunden ist, oder nicht als Ausgang des gesamten Schaltnetzes verwendet wird.

Die *Konstante* K kann mit $K : \mathbb{B}^0 \to \mathbb{B}^n$ beschrieben werden. Dies entspricht einer *Signalquelle*, d.h. das logische Signal auf einer Eingangsleitung wird nicht durch vorhergehende Schaltfunktionen bestimmt, sondern hat immer den konstanten Wert **0** oder **1**.

Die Darstellung eines Schaltnetzes (oder ganz allgemein eines Graphen) mit diesen oder ähnlich definierten Kompositionen wird auch *Termdarstellung* genannt. Diese Form der Darstellung hat mehrere Vorteile. Sie bietet eine effiziente Möglichkeit, Graphen in Computern zu speichern und zu bearbeiten. Insbesondere kann eine Termdarstellung in andere umgerechnet werden, wodurch sich auch die Möglichkeit bietet, Schaltnetze zu vereinfachen. Das Feld der Computerwissenschaften, das sehr tief in der Mathematik fundiert ist, das sich mit diesen Darstellungen beschäftigt heisst *Term–* oder *Graph Rewriting*.

3.3.3 Arithmetische Schaltnetze

Wir wollen uns nun der Realisierung von Schaltnetzen zuwenden, die grosse praktische Bedeutung für *Arithmetische Schaltnetze* und damit für digitale Rechenanlagen haben. Entsprechend dem Prinzip des Aufbaus von komplexeren Bausteinen aus Grundbausteinen beschäftigen wir uns zuerst mit der Addition von zwei Einbitzahlen. Dies mutet zwar sehr einfach an, doch werden wir bald sehen wie wichtig dieser Grundbaustein ist.

Zuerst stellen wir die Wahrheitstabelle für diese einfache Addition auf und berücksichtigen dabei, dass bei Addition der Bits x_1 und x_2 eine Summe s und ein Übertrag c (für Carry) auftreten (Tabelle 3.10).

Aus der Wahrheitstabelle leiten wir nun die DNF für die beiden Ausgänge s unbd c ab. Wir erhalten

$$s = \overline{x_1} x_2 + x_1 \overline{x_2}$$

und

$$c = x_1 x_2.$$

x_1	x_2	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabelle 3.10: Wahrheitstabelle für die Addition zweier Einbitzahlen.

Die DNF–Darstellung der Schaltfunktionen s und c lässt direkt in ein Schaltnetz mit den logischen Gattern AND, OR und NOT übertragen (Abb. 3.10).

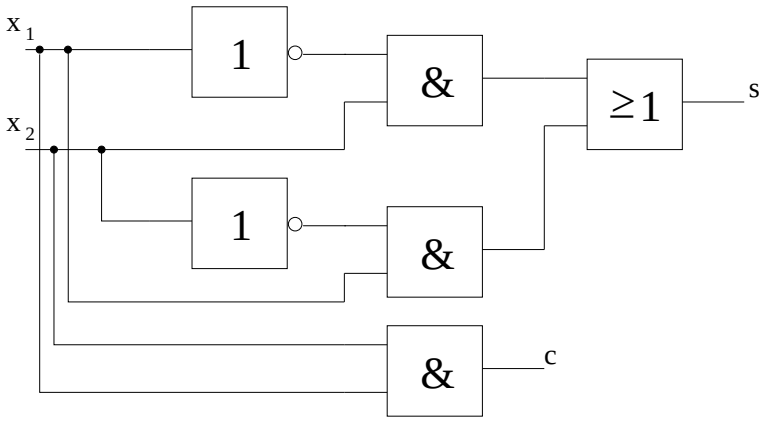

Abbildung 3.10: Ein Halbaddierer.

Für diese doch recht einfache Operation benötigen wir also sechs logische Gatter. Mit den algebraischen Operationen, die für Boolesche Algebren gelten, können wir die beiden DNF vereinfachen und damit auch s als Funktion von x_1, x_2 und c ausdrücken. Durch geeignete Umformung (prüfen Sie diese nach) erhalten wir z. B. den Ausdruck

$$s = \overline{c}(x_1 + x_2),$$

den wir in Abb. 3.11 in eine logische Schaltung übertragen haben.

Wir sehen, dass wir nur noch vier Bausteine (damit auch weniger Leitungen) benötigen, um exakt dieselbe Funktion darzustellen. Der theoretische Unterbau der Booleschen Algebra bewährt sich hier in ganz praktischer Weise zur Vereinfachung von Schaltfunktionen.

Bei der Addition von Binärzahlen sind eigentlich nicht zwei, sondern drei Einbitzahlen zu berücksichtigen, da wir zum Ergebnis der Addition an einer Stelle immer noch den Übertrag der vorigen Stelle hinzuaddieren müssen.

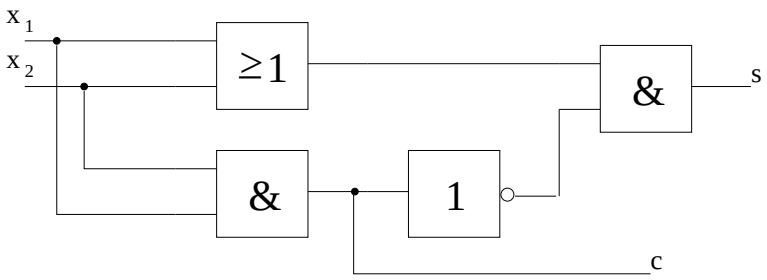

Abbildung 3.11: Vereinfachter Halbaddierer.

Aus diesem Grund heisst die bisher betrachtete Schaltfunktion *Halbaddierer* aus dem sich der in Abb. 3.12 dargestellte *Volladdierer* aufbauen lässt.

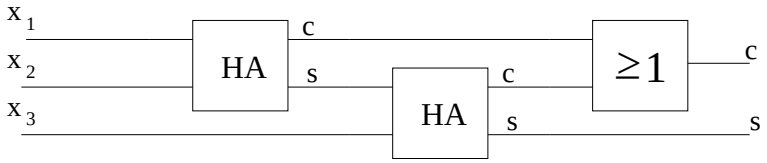

Abbildung 3.12: Ein Volladdierer.

Num haben wir also als neuen Grundbaustein den Volladdierer mit drei Eingängen und zwei Ausgängen zur Verfügung. Wenn man den Eingang x_3 als Übertrag der Addition der vorhergehenden Stelle verwendet, kann man durch Komposition von Volladdierern ein Rechenwerk aufbauen, das Binärzahlen korrekt addiert. Wir haben also die Addition auf ganz einfache logische Operationen zurückgeführt.

In ganz analoger Weise lassen sich alle möglichen arithmetischen Netze aufbauen, womit wir (theoretisch) schon jetzt in der Lage sind, grosse Teile von heutigen Prozessoren zu entwerfen. Das eher praktische Problem dabei ist die auftretende Komplexität der Schaltungen, die bei heutigen Entwürfen nur mehr durch geeignete CAD (*Computer Aided Design*)–Programme bewältigt werden kann.

Um die Komplexität von digitalen Schaltungen einigermassen in den Griff zu bekommen, versucht man jede Schaltfunktion zu minimieren, d.h. so umzuformen, dass möglichst wenig Bauteile (und Leitungen) dafür verwendet werden müssen. Zwei exakte Verfahren zur Minimierung von Schaltfunktionen werden wir gleich näher kennenlernen.

3.4 Schaltungsminimierung

Bevor wir uns der Frage zuwenden, wie wir beliebige Schaltfunktionen minimalisieren können (und damit auch die Anzahl der Bauteile in den entsprechenden logischen Schaltungen), wollen wir noch kurz alternative Darstellungsformen von Schaltfunktionen betrachten.

Wir haben Schaltfunktionen schon in die Form logischer Schaltungen gebracht, wobei logische Verknüpfungen durch spezielle logische Bausteine dargestellt werden. Mit diesen Bausteinen lässt sich ein reales, physikalisches Modell der Aussageform, die die Schaltungsfunktion definiert, erstellen. Zur einfachen Visualisierung von Schaltfunktionen gibt es aber auch andere Möglichkeiten.

3.4.1 Gebietsdarstellung

Die *Gebietsdarstellung* ist der Darstellung von Mengen entlehnt und kann für Schaltfunktionen von bis zu drei Variablen verwendet werden. Dabei begrenzt ein Rechteck das Gesamtgebiet der Schaltfunktionen, und jeder Variable wird ein Kreis zugeordnet. Die einzelnen Kreise müssen sich jeweils gegenseitig überlappen. Weiters wird zwischen schraffierten und unschraffierten Flächen unterschieden. Die schraffierten Flächen entsprechen einer wahren Aussage, während die unschraffierten den Wahrheitswert falsch aufweisen.

Die Konjunktion zweier Variablen x_1 und x_2 kann dann wie in Abb. 3.13 dargestellt werden.

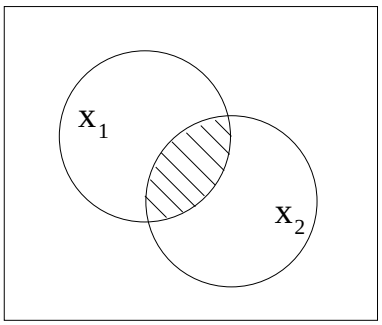

Abbildung 3.13: Die Konjunktion $y = x_1 x_2$ in Gebietsdarstellung.

Die Konjunktion wird also als gemeinsamer Bereich (Schnitt) von x_1 und x_2 dargestellt. Die Darstellung der Disjunktion und des Komplements sind anhand der Gleichung $y = \overline{x_1} + x_2$ erklärt (Abb. 3.14).

Die Gebietsdarstellung von zwei Variablen lässt genau vier Flächen unterscheiden, die allen möglichen Mintermen entsprechen. Analog existieren

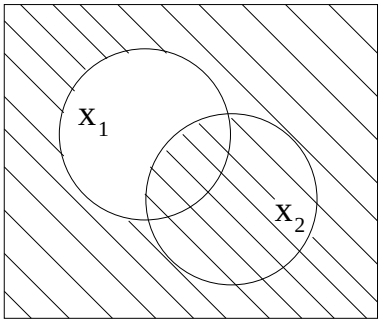

Abbildung 3.14: Die Disjunktion $y = \overline{x_1} + x_2$ in Gebietsdarstellung.

in der Gebietsdarstellung von drei Variablen acht unterscheidbare Flächen.

Beispiel: Wie lautet die Schaltfunktion der Gebietsdarstellung in Abb. 3.15?

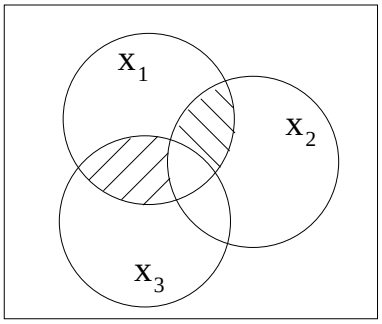

Abbildung 3.15: Eine Schaltfunktion mit den Variablen x_1, x_2 und x_3.

◊

Über die Gebietsdarstellung können Schaltfunktionen vereinfacht werden, indem man versucht die schraffierten Gebiete alternativ zu beschreiben.

Beispiel: Versuchen Sie die Schaltfunktion $y = (x_1+x_2)(x_1+x_3)+(x_1x_2x_3)$ über eine Gebietsdarstellung zu vereinfachen und verifizieren Sie das Ergebnis durch Umformen von y. ◊

Die Gebietsdarstellung hat eher anschaulichen Charakter, da die Anzahl der Variablen in der Schaltfunktion auf drei limitiert ist, und die Vereinfachung der Schaltfunktion nicht auf einem Algorithmus sondern auf Inspektion der Darstellung beruht. Eine ähnliche (geometrische) Darstellung, die wir nun behandeln wollen, bringt einige wesentliche Verbesserungen.

3.4.2 Karnaugh–Diagramm

Auch im *Karnaugh*–Diagramm [2] (K–Diagramm) werden Mintermen Teilflächen in einem geometrischen Konstrukt (Rechteck) zugewiesen. Die einzelnen Minterme werden aber hier durch gleich grosse Teilflächen repräsentiert, und benachbarte Teilflächen weisen auch Ähnlichkeiten in der Struktur ihrer entsprechenden Minterme auf. Abb. 3.16 zeigt (wieder einmal) die Konjunktion $y = x_1 x_2$ in einem Karnaugh–Diagramm.

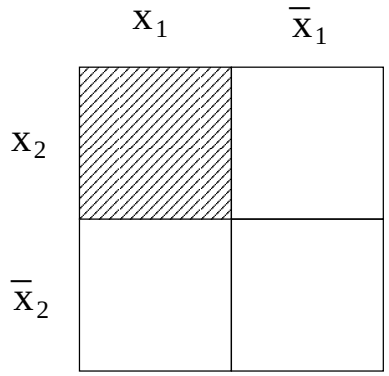

Abbildung 3.16: Das Karnaugh–Diagramm für die Konjunktion $y = x_1 x_2$.

Man erkennt, dass jede Variable in negierter und nicht–negierter Form an den Zeilen und Spalten des Diagramms aufgetragen sind. Die Kreuzungsflächen repräsentieren dann jeweils einen Minterm. Eine sehr wesentliche Eigenschaft ist dabei, dass sich die Minterme benachbarter Flächen (vertikal und horizontal) nur in einer Variablen unterscheiden. Dabei ist zu beachten, dass der Nachbar jedes Spalten– oder Zeilenendes die jeweilige erste Fläche in der Spalte oder Zeile ist. Auch für diese Nachbarn muss die erwähnte Beziehung der Minterme bestehen. Die Flächen jener Minterme, die die Schaltfunktion implizieren, werden dann gekennzeichnet (wie sieht das Karnaugh–Diagramm für die Disjunktion aus?).

Ist eine gesamte Zeile oder Spalte schraffiert, so heisst das, dass alle Minterme, die durch die einzelnen Flächen repräsentiert werden, zu der Variablen, die die Zeile oder Spalte beschreibt, zusammengefasst werden können. Dies ist auch die Grundidee zur Vereinfachung von Schaltungen mit dem Karnaugh–Diagramm.

Nun kann man auch Karnaugh–Diagramme mehrerer Variablen konstruieren, solange gewährleistet ist, dass alle möglichen Minterme (wieviele gibt es bei n Variablen?) genau durch eine Teilfläche repräsentiert werden, und die Nachbarschaftsbeziehungen eingehalten werden.

[2] Manchmal auch als *Karnaugh–Veitch–Diagramm* bezeichnet.

Die Schaltfunktion $y = (x_1 + x_2)(x_1 + x_3) + (x_1x_2x_3)$ ist im Karnaugh–Diagramm in Abb. 3.17 zu sehen.

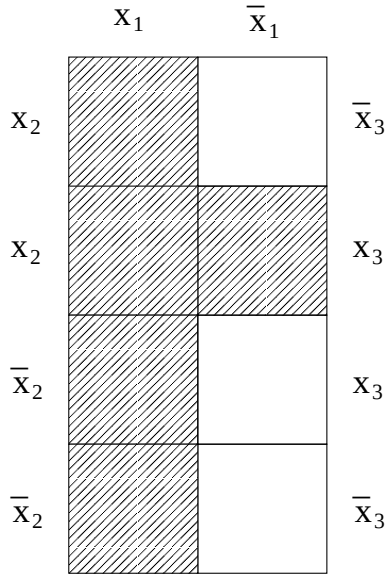

Abbildung 3.17: Das Karnaugh–Diagramm für $y = (x_1+x_2)(x_1+x_3)+x_1x_2x_3$.

Auch hier muss man wieder Sorge tragen, dass die Nachbarschaftsbeziehungen erfüllt sind (ändern Sie das Diagramm in Abb. 3.17 so, dass dies nicht mehr erfüllt ist). Dann nämlich kann man rasch erkennen, dass die gesamte linke Spalte durch x_1 und die zweite Zeile durch x_2x_3 dargestellt werden kann. Daher können wir y sofort zu $x_1 + x_2x_3$ vereinfachen.

Für 4 Variable benötigen wir ein Karnaugh–Diagramm mit 16 Flächen. Ab einer Anzahl von 5 Variablen kann man entweder zu einer 3–dimensionalen Darstellung übergehen, oder diese nach Abb. 3.18 "abwickeln".

Bei dieser Darstellung steht K_4 für das Karnaugh–Diagramm für die Variablen x_1, x_2, x_3 und x_4. Allerdings ist diese Darstellung für sechs Variable schon relativ unübersichtlich, daher wollen wir uns zunächst mit einem analytischen Verfahren zur Schaltungsminimierung beschäftigen, das auch für eine etwas grössere Anzahl von Variablen geeignet ist.

3.4.3 Das Verfahren von Quine–McCluskey

Überlegen wir einmal was Schaltungsminimierung im Kontext einer Schaltfunktion, die in DNF vorliegt, heisst. Um aus dieser Schaltfunktion zu einer "kleineren" Schaltung zu gelangen, müssten wir einzelne Variablen in speziellen Konjunktonstermen, oder aber gleich ganze Konjunktionsterme weglassen

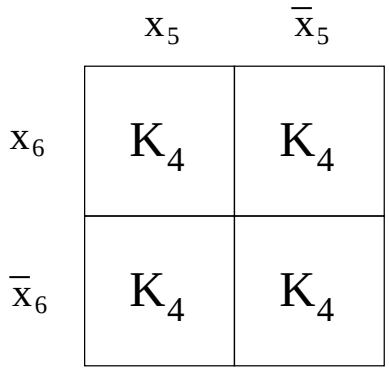

Abbildung 3.18: Das Karnaugh–Diagramm für 6 Variable mit K_4 als Unterdiagramm für 4 Variable.

können. Wenn wir die Gesamtzahl der negierten oder nicht–negierten Variablen in einer DNF mit v_Φ (jedes Auftreten wird gezählt) und die Anzahl der Konjunktionsterme mit k_Φ bezeichnen, dann können wir die Vereinfachung einer DNF formal definieren.

Definition 3.4.1 *Sind zwei DNFen Φ und Ψ gegeben, so heisst* Φ einfacher *als Ψ genau dann, wenn $v_\Phi \leq v_\Psi$ und $k_\Phi \leq k_\Psi$ und mindestens eine der Ungleichungen streng ($<$) gilt.*

Wenn wir von Schaltungsminimierung sprechen, dann möchten wir also die zugehörige DNF der Schaltfunktion so einfach wie möglich machen (minimieren).

Definition 3.4.2 *Die DNF Φ einer Schaltfunktion heisst* Disjunktive Minimalform *(DMF) genau dann, wenn keine einfachere logisch äquivalente Schaltfunktion existiert.*

Das heisst aber auch, dass die einzelnen Konjunktionsterme (Disjunktionsglieder) einer DMF nicht weiter vereinfacht werden können (warum?). Daher liegt die Vermutung nahe, dass die Disjunktionsglieder einer DMF Primimplikanten sind, was von folgendem Theorem bestätigt wird.

Theorem 3.4.1 *Jede DMF Φ der Schaltfunktion y ist eine Disjunktion von einem oder mehreren Primimplikanten.*

Das Theorem 3.4.1 (versuchen Sie den (kurzen) Beweis) zeigt auf, wie man von einer DNF die entsprechende DMF findet. Allerdings haben wir noch keine Methode, alle Primimplikanten einer DNF zu ermitteln, zur Verfügung. Dazu benötigen wir noch einige Erkenntnisse der Aussagenlogik.

Definition 3.4.3 *Wir betrachten eine DNF und zwei Konjunktionsterme ϕ_1 und ϕ_2. Wir sagen* ϕ_2 wird durch ϕ_1 bezüglich der DNF abgeschlossen*, wenn ϕ_2 in ϕ_1 enthalten ist, und ϕ_1 ein Minterm der DNF ist.*

Beispiel: Ausgehend von der DNF $x_1\overline{x_2} + x_2x_3$ und dem Term $y = x_1x_2$ stellen wir fest, dass jeder der Minterme $x_1x_2x_3$ und $x_1x_2\overline{x_3}$ y abschliesst. Daran erkennen wir auch den Zweck dieser Definition, denn wenn die beiden Minterme in der DNF enthalten sind, dann lassen sie sich durch den Term y ersetzen, und die DNF wird vereinfacht. $\Diamond$

Satz 3.4.1 *Ein Konjunktionsterm ϕ ist genau dann Primimplikant einer nicht allgemeingültigen vollständigen DNF Φ, wenn*

(i) alle Variablen von ϕ auch in Φ vorkommen und

(ii) alle Minterme, die ϕ bezüglich Φ abschliessen, Disjunktionsglieder von Φ sind, dies aber für jeden anderen in ϕ enthaltenen Term nicht gilt.

Der Satz 3.4.1 gibt uns jetzt die nötigen Hinweise, aus einer vollständigen DNF alle Primimplikanten zu bestimmen, was nach ihren Begründern als Verfahren von *Quine–McCluskey* bekannt ist. Dabei gehen wir von der vollständigen DNF $\Phi = \psi_1 + \psi_2 + \ldots + \psi_k$ aus.

(1) Die Minterme $\psi_1, \ldots, \psi_k$ werden in einer Liste zusammengestellt. Dabei ist es vorteilhaft, die Minterme in Gruppen einzuteilen, sodass benachbarte Gruppen sich nur in einer Variablen unterscheiden. Wir beginnen also z. B. damit, alle Minterme ohne negierte Variablen anzuschreiben (Klasse K_0), dann diejenigen Minterme mit einer negierten Variablen (K_1) etc.

(2) Wenn sich in der Liste zwei Minterme ψ_i und ψ_j nur in einer Variablen unterscheiden (negiert, nicht–negiert), was nur in benachbarten Klassen möglich ist, dann schreiben wir einen neuen Konjunktionsterm $\psi_{i,j}$ an, der durch Weglassen der unterschiedlichen Variablen entsteht. ψ_i und ψ_j werden dann als erledigt abgehakt.

(3) Wir führen Punkt (2) solange aus, bis wir nichts mehr zusammenfassen können. Dabei ist es auch möglich, bereits abgehakte Terme wieder neu zusammenzufassen.

(4) Die in der Liste verbleibenden nicht abgehakten Konjunktionsterme sind alle Primimplikanten von Ψ.

Wir wollen dieses Verfahren anhand eines Beispiels demonstrieren, das auch den ganzen Prozess des Entwurfs einer (kleinen) logischen Schaltung – von der Festlegung der gewünschten Funktion bis zur Minimalisierung der Schaltung – darlegt.

Beispiel: Wir wollen eine Schaltung zur Erkennung von *Pseudotetraden* in Zahlen, die im *Binary Coded Decimal* (BCD)–Format vorliegen, entwerfen (Dworatschek, 1970). Die BCD–Darstellung von Zahlen codiert jede Ziffer einer Dezimalzahl als binäre 4–Bit–Zahl.

So wird z. B. die Zahl $259_{(10)}$ zu $001001011001_{(BCD)}$ (prüfen Sie das nach). Offensichtlich existieren dann Codes, die keine Dezimalzahl repräsentieren (warum?). Diese "sinnlosen" Codes werden als *Pseudotetraden* bezeichnet (Tetrade steht hier für ein 4–Bit–Wort, das auch als *Nibble* bezeichnet wird).

Der BCD–Code wird z. B. zur Ansteuerung von Digitalanzeigen verwendet, da man zumeist Dezimalzahlen anzeigen will, deren Ziffern sich aus einem BCD–Code sehr einfach extrahieren lassen.

Wir beginnen also mit dem Aufstellen einer Wahrheitstafel (Tabelle 3.11, die einer korrekten BCD–Zahl $(x_3x_2x_1x_0)$ den Wert 0 und einer Pseudotetrade den Wert 1 zuweist.

x_3	x_2	x_1	x_0	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Tabelle 3.11: Wahrheitstabelle zur Erkennung von Pseudotetraden im BCD–Code.

Aus der Wahrheitstabelle 3.11 leiten wir die vollständige DNF

$$x_3\overline{x_2}x_1\overline{x_0} + x_3\overline{x_2}x_1x_0 + x_3x_2\overline{x_1x_0} + x_3x_2\overline{x_1}x_0 + x_3x_2x_1\overline{x_0} + x_3x_2x_1x_0$$

ab.

Nun bestimmen wir alle Primimplikanten der vollständigen DNF mit dem Verfahren von Quine–McCluskey. Dazu teilen wir die Minterme entsprechend der Anzahl ihrer negierten Terme in Klassen ein, die wir als $K_0, \ldots, K_3$ bezeichnen, und die Terme solange zusammenfassen und bei Verwendung abhaken, bis keine Vereinfachung mehr möglich ist.

$$
\begin{array}{lll}
K_0 : x_3x_2x_1x_0\surd & x_3x_2x_1\surd & x_3x_2 \\
 & x_3x_2x_0\surd & x_3x_1 \\
 & x_3x_1x_0\surd & x_3x_2 \\
 & & x_3x_1 \\
K_1 : x_3x_2x_1\overline{x_0}\surd & x_3x_2\overline{x_0}\surd & \\
\quad x_3x_2\overline{x_1}x_0\surd & x_3x_1\overline{x_0}\surd & \\
\quad x_3\overline{x_2}x_1x_0\surd & x_3x_2\overline{x_1}\surd & \\
 & x_3\overline{x_2}x_1\surd & \\
K_2 : x_3x_2\overline{x_1x_0}\surd & & \\
\quad x_3\overline{x_2}x_1\overline{x_0}\surd & &
\end{array}
$$

Wir erhalten als Ergebnis vier Primimplikanten, wobei allerdings jeweils zwei ident sind, was mit dem Idempotenzgesetz die DMF $x_3x_2 + x_3x_1$ ergibt, die sich sogar noch zu $x_3(x_2 + x_1)$ vereinfachen lässt (zeichnen Sie die entsprechende logische Schaltung). $\Diamond$

In obigem Beispiel sind die beiden gefundenen Primimplikanten *wesentlich*, da wir beide für die DMF unbedingt brauchen. Es gibt aber auch Fälle, bei denen nicht alle gefundenen Primimplikanten wesentlich sind, d. h. diese können auch noch weggelassen werden, um zur DMF zu gelangen. Wir benötigen daher noch ein Verfahren, das uns erlaubt, die wesentlichen Primimplikanten zu bestimmen. Dieses einfache Verfahren basiert auf dem

Satz 3.4.2 *Jeder Minterm der vollständigen DNF Φ enthält mindestens einen Konjunktionsterm der DMF Ψ.*

(Führen Sie den kurzen Beweis). Daraus sehen wir sofort, dass ein Primimplikant, der nur in einem einzigen Minterm enthalten ist, wesentlich sein muss, da ein Weglassen dieses Primimplikanten zur Verletzung von Satz 3.4.2 führt. Damit gelangen wir zur *Bestimmung der wesentlichen Primimplikanten:*

(1) Wir weisen jedem Minterme ϕ der vollständigen DNF eine Spalte in einer Tabelle zu. In jede Zeile dieser Tabelle tragen wir einen Primimplikanten von Φ ein.

(2) Enthält ein Minterm einen Primimplikanten, so kennzeichnen wir die entsprechende Stelle in der Tabelle mit einem Kreuz.

(3) Enthält eine Spalte nur ein Kreuz (wesentlicher Primimplikant), dann umgeben wir das Kreuz mit einem Kreis. Alle anderen Kreuze derselben Zeile umgeben wir mit einem Quadrat. Existiert keine Spalte mit nur einem Kreuz, dann umgeben wir ein beliebiges Kreuz einer Spalte mit einem Kreis.

(4) Erscheint dann in jeder Spalte ein Kreis oder ein Quadrat, ist das Verfahren beendet, andernfalls wiederholen wir (3) für eine noch unberücksichtigte Spalte. Es verbleiben jene Primimplikanten, in deren Zeile kein Kreis auftritt, als nicht wesentlich. Diese bleiben dann für die DMF unberücksichtigt.

Wir wollen dieses Verfahren mit einem Beispiel illustrieren.

Beispiel: Die vollständige DNF Φ sei gegeben mit $x_1\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1x_2\overline{x_3} + x_1x_2x_3$. Die Primimplikanten von Φ sind x_1x_2, x_1x_3 und $x_2\overline{x_3}$ (weisen Sie das nach). Wir stellen dann die Tabelle 3.12 zur Bestimmung der wesentlichen Primimplikanten auf und verfahren wie oben beschrieben.

	$x_1\overline{x_2}x_3$	$\overline{x_1}x_2\overline{x_3}$	$x_1x_2\overline{x_3}$	$x_1x_2x_3$
x_1x_2			x	x
x_1x_3	ⓧ			☒
$x_2\overline{x_3}$		ⓧ	☒	

Tabelle 3.12: Bestimmung der wesentlichen Primimplikanten.

Wir sehen, dass der Primimplikant x_1x_2 nicht wesentlich ist. Damit lautet die DMF $x_1x_3 + x_2\overline{x_3}$ (prüfen Sie das nach). ◊

Mit dem Verfahren von Quine–McCluskey und der Technik zur Auffindung der wesentlichen Primimplikanten ist es uns also möglich die DMF aus einer vollständigen DNF exakt zu bestimmen. Allerdings kann dieses Verfahren nur bis zu ca. 10 Variablen eingesetzt werden, da der Aufwand sonst zu hoch wird. Für komplexere Probleme wird auf *heuristische* Methoden zurückgegriffen. Man gibt sich also mit einer akzeptablen Lösung zufrieden, was im Lichte heutiger Integrationsdichten von VLSI–Schaltnetzen durchaus ausreichend ist.

3.4.4 Graphisches Verfahren mit Karnaugh–Diagrammen

Zur Schaltungsminimierung können wir auch das K–Diagramm heranziehen und dabei insbesondere seine Eigenschaft ausnutzen, dass benachbarte Fel-

der sich immer nur in einer Variablen eines Minterms unterscheiden. Unter benachbarten Feldern verstehen wir jene Felder, die eine gemeinsame Seite haben, wobei wir uns das Ende jeder Zeile/Spalte mit dem Anfang verbunden vorstellen.

Wir greifen wieder auf unser Beispiel der Auffindung von Pseudotetraden zurück, um die Schaltungsminimierung mit K–Diagrammen zu erläutern.

Beispiel: Aus der Wahrheitstabelle 3.11 erhielten wir die vollständige DNF

$$x_3\overline{x_2}x_1\overline{x_0} + x_3\overline{x_2}x_1x_0 + x_3x_2\overline{x_1x_0} + x_3x_2\overline{x_1}x_0 + x_3x_2x_1\overline{x_0} + x_3x_2x_1x_0.$$

Diese tragen wir nun in ein K–Diagramm ein, indem wir die Flächen, die die jeweiligen Minterme repräsentieren, kennzeichnen (Abb. 3.19).

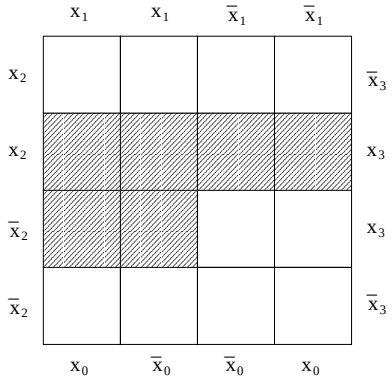

Abbildung 3.19: Karnaugh–Diagramm für das Beispiel der Pseudotetraden.

Wir fassen nun Blöcke von 2^n (hier $n = 1, \ldots, 4$) benachbarten Flächen so zusammen, dass jede Fläche mindestens einmal in einem Block vorkommt (logische Äquivalenz), die Anzahl der Blöcke minimal wird (Konjunktionsterme), und die Grösse der Blöcke maximal wird (Variable des Konjunktionsterms) (Abb. 3.20).

Wir erkennen, dass der quadratische 4–Block den Term x_1x_3 und der horizontale 4–Block x_2x_3 repräsentiert. Die Disjunktion dieser Terme führt wieder auf die gesuchte DMF $x_1x_2 + x_2x_3$. In Abb. 3.20 hätten wir auch mehrere 2–Blöcke bilden können, doch wäre dies eben nicht die minimale Anzahl von Blöcken gewesen, die wir zum Aufstellen der DMF brauchen. Trotzdem wäre auch dies eine Vereinfachung der ursprünglichen vollständigen DNF. Wir erkennen auch, dass dieses Zusammenfassen von Feldern voraussetzt, dass sich benachbarte Flächen nur in einer Variablen des Minterms unterscheiden. $\Diamond$

Dieses elegante und sehr anschauliche Verfahren zur Schaltungsminimierung hat aber auch den Nachteil, dass die Anzahl der Variablen auf 6 beschränkt ist, da es darüberhinaus zu Darstellungsproblemen kommt.

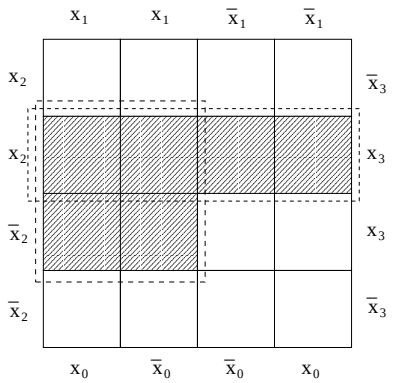

Abbildung 3.20: Minimalisierung mit dem Karnaugh–Diagramm.

3.5 Schaltwerke

Bisher haben wir mit den Schaltnetzen Systeme betrachtet, die Information (Bitfolgen) ausschliesslich in einer Richtung verarbeitet haben. Identische logische Signale an den Eingängen einer logischen Schaltung produzieren immer dieselben Zwischen–, und Ausgangswerte. Dieses stereotype (statische) Verhalten ist aber selbst für eine automatische Parkplatzschranke zuwenig.

Würde diese bei Ankunft eines Autos (und einem daraus abgeleiteten logischen Signal) immer aufmachen, käme es immer dann zu Problemen, wenn der Parkplatz voll ist. Daher muss sich dieses System *dynamisch* verhalten, i.e. die Schranke ändert ihre Aktionen abhängig von der Anzahl der Autos am Parkplatz. Die Schranke braucht also Information darüber was ihre (letzte) Aktion bewirkt hat. Dieser Vorgang heisst *Rückkopplung* oder *Feedback* und ist ein zentraler Begriff in der Informationstechnologie (Informatik, Elektrotechnik).

Beim Beispiel der Parkplatzschranke könnte diese Rückkopplung dadurch erfolgen, dass die Anzahl der Autos am Parkplatz immer mitgezählt wird. Erreicht diese Zahl das Limit, dann ändert sich das Verhalten der Schranke bei Ankunft eines Autos: sie bleibt geschlossen (und informiert hoffentlich über den Grund dieser Aktion). Die Anzahl der Autos muss in diesem System *gespeichert* sein. Es muss also eine Möglichkeit geben, sich an Aktionen in der Vergangenheit "erinnern" zu können, was technisch nur dann möglich ist, wenn Information gespeichert werden kann.

Um selbst solch einfach anmutende Aufgaben lösen zu können, muss ein Schaltnetz Informationen über den Ausgang seiner Aktionen bekommen. Anders ausgedrückt müssen Eingänge eines Schaltnetzes von Ausgängen desselben beeinflusst werden. Schaltnetze mit derartigen Rückkopplungen heissen *Schaltwerke*.

Ein Schaltwerk ist dann ein gerichteter Graph, an dessen Knoten sich Schaltwerke (oder Schaltnetze) befinden. Der wesentlich Unterschied zu einem Schaltnetz ist also, dass der Graph auch Zyklen enthält, die für die Rückkopplung verantwortlich sind (Abb. 3.21).

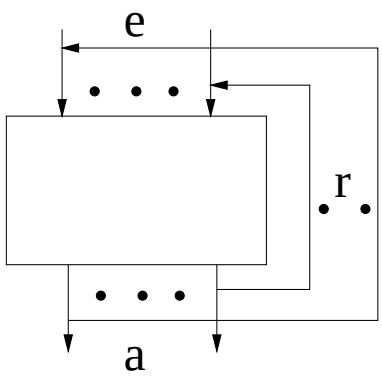

Abbildung 3.21: Ein Schaltwerk mit r rückgekoppelten Ausgängen.

Von a Eingängen des Schaltwerks in Abb. 3.21 sind r rückgekoppelte Ausgänge, die Information über die Vergangenheit des Schaltwerks geben, da dieser Ausgang vom vorherigen Eingang abhängig ist.

Ein wesentliches Problem bei dieser Rückkopplung ist, dass bei realen Systemen die Ausgänge sicher nie absolut gleichzeitig auf die Eingänge reagieren, sodass etwa Ausgang a_i schon an den Eingang e_j gelangt, obwohl der Ausgang a_r noch den alten Zustand an e_s überträgt. Dies würde aber bedeuten, dass die Information am Eingang eine sinnlose Mischung aus Gegenwart und Vergangenheit darstellt, was letztendlich zu einer Fehlfunktion des Schaltwerks führen würde.

3.5.1 Zeitverhalten

Daher wird für Schaltwerke zumeist ein Taktsignal benötigt, das als Zeitgeber fungiert, sodass sich die einzelnen Signale eindeutig gewissen Zeitspannen, oder –punkten zuordnen lassen. Dieser Vorgang wird auch als *Synchronisation* bezeichnet.

Da wir es in digitalen Schaltungen immer mit logischen Signalen mit genau zwei Zuständen zu tun haben, liegt es nahe, das Taktsignal (oder Takt) ebenfalls aus nur zwei Zuständen aufzubauen. Ein Rechtecksignal erfüllt genau diese Anforderung. Abbildung 3.22 zeigt die einfachste Möglichkeit ein Eingangssignal e_i mit einem Taktsignal t zu synchronisieren.

Dabei wird der Takt auf einen Eingang eines AND–Gatters gelegt. Ist der Taktpegel auf L (Lo(w)), so entspricht dies einer logischen 0. Wenn der Pegel

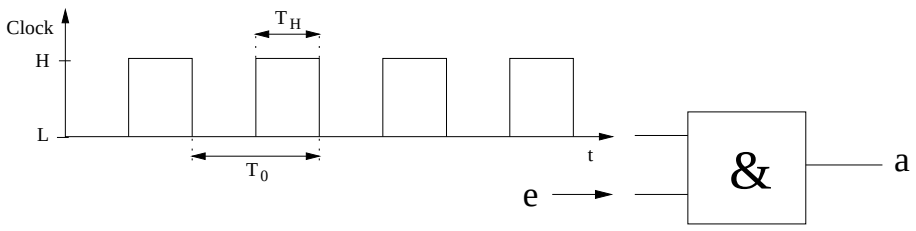

Abbildung 3.22: Ein zustandsgesteuertes Eingangssignal e.

auf H (Hi(gh)) ist, dann repräsentiert dieser Zustand eine logische 1 (man bezeichnet dies als *positive* Logik). Das bedeutet für das Eingangssignal e, dass es nur dann am Ausgang a erscheint, wenn der Takt H ist, andernfalls wird der Eingang "abgeschaltet". Man nennt diese Art der Synchronisation *Zustandssteuerung* (*State Triggering*), weil das Eingangssignal durch den Zustand des Takts für eine gewisse Zeitspanne definiert wird. In den L–Zeiten des Takts kann das Eingangssignal beliebig schwanken, doch der Ausgang bleibt davon unberührt.

Einige wichtige Begriffe im Zusammenhang mit dem Taktsignal (oft auch als *Clock* bezeichnet) wollen wir noch kurz betrachten. Die Periodendauer T_0 eines periodischen Signals genügt der elementaren Beziehung

$$f = \frac{1}{T_0} \quad [f] \ldots Hz. \tag{3.13}$$

Das *Tastverhältnis* α ist durch

$$\alpha = \frac{T_H}{T_0} \tag{3.14}$$

gegeben und gibt die relative Dauer des H–Pegels an. Wir können über den Taktpegel zwar Signale synchronisieren, doch verbleibt das Problem, dass innerhalb der Zeitdauer T_H der Eingang e ebenfalls schwanken könnte. Würden wir α sehr klein machen, dann liesse sich das Eingangssignal immer schärfer auf einen Zeitpunkt eingrenzen. Da aber viele Bauteilgruppen in digitalen Schaltungen einen Takt mit einem Tastverhältnis $\alpha \simeq 0.5$ benötigen, kann man mit wenigen Gattern einen sehr schmalen Impuls aus einem solchen Takt ableiten (Abb. 3.23).

Sehr interessant ist dabei, dass man genau jene Eigenschaft (nämlich unterschiedliche Signallaufzeiten in Schaltwerken), die man eigentlich in den Griff bekommen möchte, zu ihrer eigenen Bekämpfung verwendet. Ist der Takt auf L, dann ist der Ausgang des NAND–Gatters H. Springt der Takt auf H, dann braucht der Inverter etwas Zeit, um seinen Ausgang auf L zu setzen. Während dieser Zeit liefert das NAND–Gatter aber L und springt

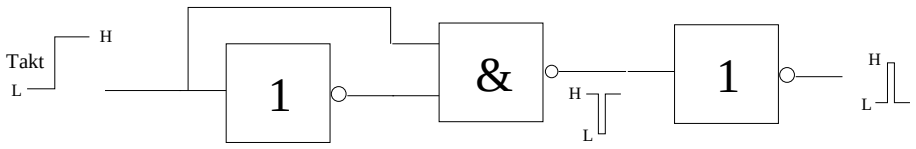

Abbildung 3.23: Ableitung eines Flankenimpulses aus dem Taktsignal.

dann gleich wieder auf H. Die Signallaufzeit durch den Inverter bestimmt also die Dauer des abgleiteten Impulses, der aber jedenfalls sehr viel kürzer als T_H des Takts sein wird. Der abschliessende Inverter stellt nur sicher, dass eine *positive Flanke* (Übergang L auf H) einen positiven Impuls generiert.

Die kleine Schaltung in Abb. 3.23 liefert also bei jeder positiven Flanke einen sehr kurzen Impuls ($\alpha <<$) (was passiert bei einer negativen Flanke?). Damit können wir nun "Momentaufnahmen" von logischen Signalen machen. Man nennt diese Technik *Flankensteuerung* (*Slope Triggering*).

3.5.2 Das Huffman–Modell

Generell halten wir fest, dass Schaltwerke ein Zeitverhalten aufweisen, das durch die allgemeine Gleichung

$$\vec{z}(t + \Delta t) = f(x(\vec{t}), y(\vec{t})) \tag{3.15}$$

beschrieben werden kann. Dabei ist $\vec{z}$ der Vektor aller Ausgänge, $\vec{x}$ repräsentiert die Eingänge, und $\vec{y}$ den (inneren) Zustand des Schaltwerks. Der Ausgangspunkt dieser Beschreibung ist das *Huffman–Modell* eines Schaltwerkes (Volkert, 1999). In diesem Modell besteht ein Schaltwerk aus zwei Blöcken:

- Kombinatorischer Schaltkreis (Schaltnetz)
- Speicher (zeitabhängiger Zustand)

In diesem Modell steuert der innere Zustand des Schaltwerks (Speicher), der auf den Eingang rückgekoppelt wird, die Verarbeitung der Eingänge $\vec{x}(t)$. In der Informatik bezeichnet man ein solches System auch als einen *Endlichen Automaten*, der auf externe Ereignisse $\vec{x}(t)$ je nach Zustand des Automaten unterschiedliche Aktionen setzt (dieses Modell wird vom Compilerbau über Software Engineering bis zur Robotik verwendet).

Für das Aufstellen eines Huffman–Modells und der Konstruktion eines Schaltwerks aus diesem Modell exisitieren keine analytischen Verfahren, daher spielen Erfahrung und daraus gewonnene Heuristiken eine grosse Rolle

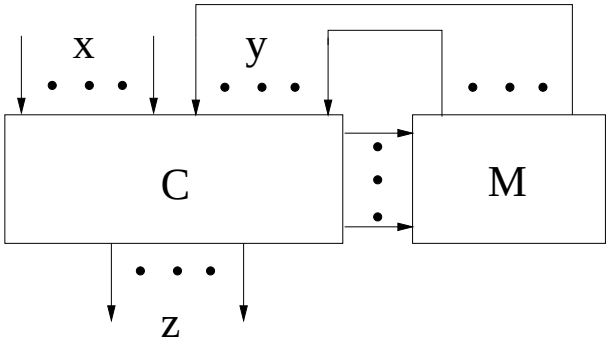

Abbildung 3.24: Huffman–Modell eines Schaltwerks mit den zwei Blöcken C (Schaltnetz) und M (Speicher).

bei der Realisierung von Schaltwerken. Natürlich kann man den Schaltnetzteil in gewohnter Weise behandeln, und auch auf Grundbausteine für den Speicher zurückgreifen. Grundelement aller logischen Bausteine mit Speichereigenschaft ist das *Flip–Flop*, das wir nun näher betrachten wollen.

3.5.3 Das R–S–Flip–Flop

Das einfachste speichernde Bauelement ist das *R–S–Flip–Flop*, wobei R für *Reset* (Rücksetzen) und S für *Set* (Setzen) steht (Newald and Lindner, 1985). Diese Bezeichnung rührt daher, dass man ein Flip–Flop als einen *Ein–Bit–Speicher* betrachten kann, dessen Inhalt man schreiben (setzen, rücksetzen) oder (unverändert) lesen kann. Abb. 3.25 zeigt die einfachste Realisierung eines R–S–Flip–Flops.

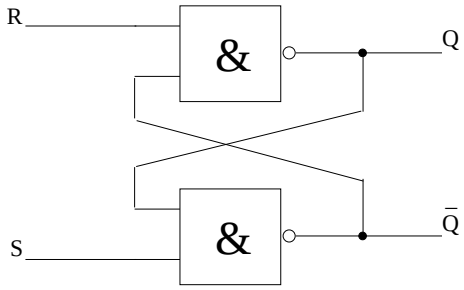

Abbildung 3.25: Einfaches R–S–Flip–Flop.

Wir sehen gleich, dass dieses einfache Bauelement die Ausgänge (Zustände) auf den Eingang rückkoppelt, was wir als wesentliches Merkmal eines Schaltwerks identifizert haben. Diese Rückkopplungen erschweren aber auch die Analyse der Schaltung. In diesem Fall schaffen wir es aber noch, wenn wir

uns vor Augen halten, dass eine 0 am Eingang eines NAND–Gatters eine 1 am Ausgang erzwingt.

Ist also $R = 0$, dann muss $Q = 1$ sein. Für $S = 1$ ergibt dies ein $\overline{Q} = 0$. Für den komplementären Fall $R = 1, S = 0$ vertauschen auch Q und $\overline{Q}$. Ist $R = S = 1$, so behalten Q und $\overline{Q}$ ihren alten Wert. Dieser Wert bleibt also gespeichert! Ist $R = S = 0$ wird $Q = \overline{Q} = 0$, was einerseits der Absicht widerspricht, komplementäre Ausgänge zu bilden, andererseits bei Übergang auf $R = S = 1$ einen instabilen Zustand bedingt. Daher wird letzterer Fall üblicherweise vermieden.

Tabelle 3.13 gibt die Wahrheitstabelle für dieses Schaltwerk an, die hier *Transitionstabelle* genannt wird, da darin auch zeitliche Zustandsübergänge eingetragen sind.

R	S	$Q(t)$	$Q(t+\Delta t)$
0	0	0	1*
0	0	1	1*
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Tabelle 3.13: Transitionstabelle des R–S–Flip–Flops.

Es wird ersichtlich, dass der Zustand des Schaltwerks, der sich zeitlich ändert, hier einfach der Ausgang Q ist. Das Symbol '*' steht für den unerwünschten instabilen Zustand. Die Funktion dieses Flip–Flops wird dann klar ersichtlich. Für $S = \overline{R}$ folgt der Ausgang Q immer S (unabhängig vom Zustand). Dies entspricht einer Schreiboperation: der Speicher wird mit S beschrieben. Ist $R = S = 1$, dann ist der neue Zustand $Q(t + \Delta t)$ gleich dem alten Zustand $Q(t)$. Der Inhalt des Speichers bleibt also unverändert, d.h. das Bit wird gespeichert.

Das R–S–Flip–Flop ist somit die Basis aller speichernden Bausteine, da man einen n –Bit–Speicher prinzipiell als Komposition von solchen 1–Bit–Speichern aufbauen kann.

3.5.4 Das D–Flip–Flop

Ein Problem des einfachen R–S–Flip–Flop aus Abb. 3.25 ist, dass jede Änderung von R und S sofort (innerhalb der Signallaufzeiten) den Zustand von Q

verändert (Ausnahme?). Nun möchte man in digitalen Rechenanlagen Daten meist zu einem gewissen Zeitpunkt lesen oder schreiben. Wir brauchen also wieder einen Taktgeber, der das Flip–Flop steuert. Daher versehen wir unser R–S–Flip–Flop mit einer kleinen Zusatzschaltung, die in Abb. 3.26 gezeigt ist.

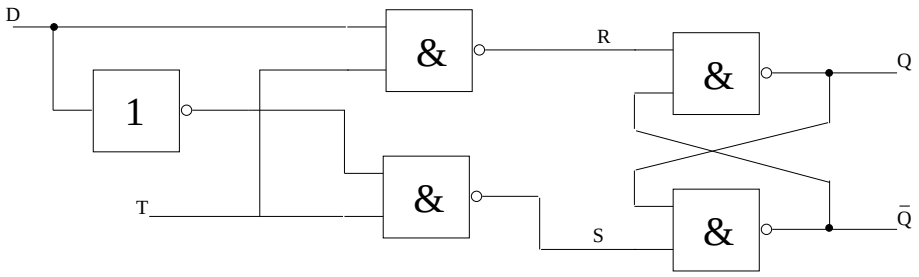

Abbildung 3.26: D–Flip–Flop (Latch).

Dieses *D–Flip–Flop* oder *Latch* (*Auffang–Flip–Flop*) übernimmt das an D anliegende Bit (Datum) nur dann, wenn der Takt T den Zustand H einnimmt. Ist $T = 0$, dann wird $R = S = 1$ (warum?) und das Flip–Flop speichert. In diesem Zustand bleibt jegliche Änderung von D unberücksichtigt. Ist $T = 1$, dann wird $Q = D$. Hier folgt der Ausgang Q dem Eingang D (Schreiboperation).

Der Takt muss nicht unbedingt ein periodisches Signal sein, sondern kann von einem anderen Schaltwerk kommen. Dieses kann dann dem Latch signalisieren, wann es ein neues Datum übernehmen soll. Das Latch kann z. B. zur Aufzeichnung und Ausgabe von Messdaten verwendet werden. Ist der Takt periodisch, dann wird bei jedem H–Pegel der Speicher neu beschrieben. Der Takt kann aber auch z. B. durch Ihren Knopfdruck erfolgen, womit Sie dem Gerät mitteilen, wann Sie einen neuen Messwert haben wollen.

Wenn wir mehrere Bits speichern wollen, ist obiges Latch sehr einfach erweiterbar, indem man einfach n dieser 1–Bit–Latches parallel komponiert und mit dem Takt synchronisiert. Wir haben damit ein *n–Bit–Register* konstruiert. Die Symbole für die Speicherbausteine R–S–Flip–Flop und D–Flip–Flop sind in Abb. 3.27 dargestellt.

Das Symbol 'C1' beim Takteingang des D–Flip–Flops steht für die Datenübernahme während des H–Pegels des Takts (logische 1 der Clock). Hier tritt nun wieder das schon bekannte Problem auf, dass es viele Anwendungen gibt, bei denen eine Änderung des Speicherzustands nicht zustandsgesteuert, sondern flankengesteuert erfolgen soll. Wir könnten dazu das Latch mit der kleinen Schaltung aus Abb. 3.23 versehen, oder aber das *Master–Slave–*Prinzip zur Anwendung bringen. Dies besteht darin, dass man zwei D–Flip–

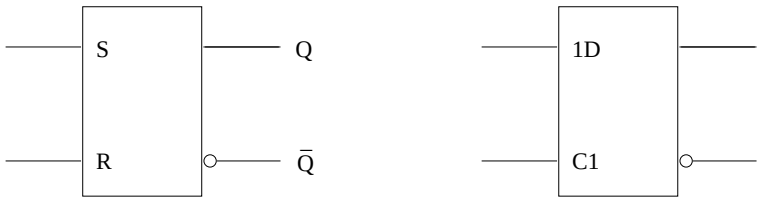

Abbildung 3.27: Schaltsymbole für R–S–Flip–Flop (links) und D–Flip–Flop (rechts).

Flops hintereinanderschaltet, wobei das nachgeschaltete (Slave) den Ausgang des vorgeschalteten (Master) nur zu einem gewissen Zeitpunkt (Taktflanke) übernimmt (Abb. 3.28).

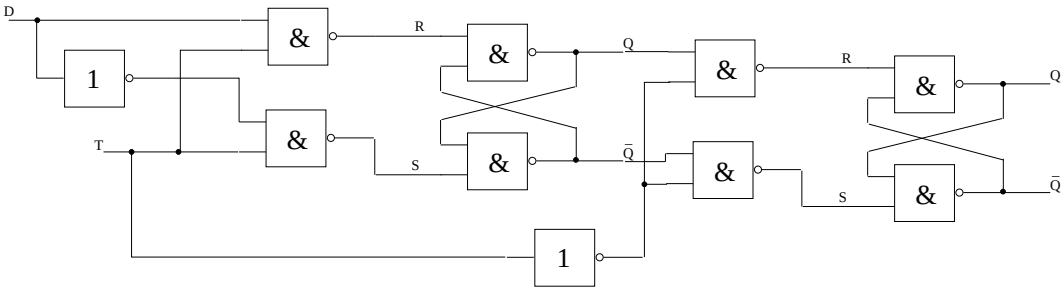

Abbildung 3.28: Zwei D–Flip–Flops in Master–Slave–Schaltung.

Liegt am Master–Flip–Flop der Takt 0 an, so haben wir die bekannte Speichersituation des Masters. Geht der Takt auf 1, so übernimmt der Master das Datum. Da der Takt für den Slave invertiert ist, ist dieser in Speicherstellung, d.h. der Ausgang bleibt unverändert. In dieser Situation kann sich der Eingang und damit der Ausgang des Masters ständig ändern, trotzdem reagiert der Slave nicht. Wenn jetzt der Takt am Master von 1 auf 0 geht (negative Taktflanke), geht der Takt am Slave auf 1 und der Slave übernimmt den Ausgang des Masters, der sich ab diesem Zeitpunkt in Speichersituation befindet. Das globale Verhalten dieser Master–Slave–Schaltung ist also die Übernahme des Master–Eingangs D auf den Slave–Ausgang Q mit der negativen Taktflanke. Damit haben wir erreicht, dass der Speicher nur zu exakt definierten Zeitpunkten beschrieben werden kann.

3.5.5 Das J–K–Flip–Flop

Ein Nachteil des R–S–Flip–Flops ist die nicht zulässige Eingangssituation $R = S = 0$, das die Operationsmodi dieses Flip–Flops auf Setzen, Rücksetzen und Speichern beschränkt. Ein zusätzlicher Modus *Kippen* (Toggle)

würde aber viele Vorteile bringen und so das Einsatzspektrum erweitern. Genau dies wird mit dem J–K–Flip–Flop erreicht, bei dem eine zusätzliche Rückkopplung vom Ausgang zum Eingang eines Master–Slave–Flip–Flops exisitiert (Abb. 3.29).

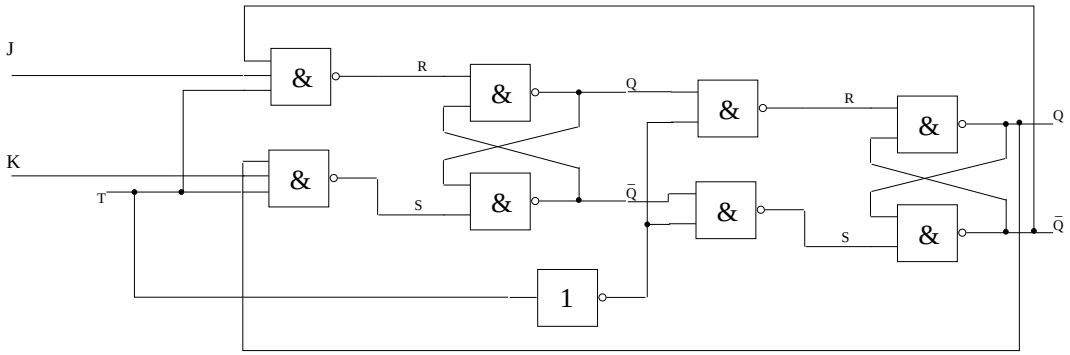

Abbildung 3.29: J–K–Flip–Flop.

Es wird also der Slave–Ausgang Q auf den Master–Eingang K und $\overline{Q}$ auf J rückgekoppelt. Dadurch wird am Master $R = \overline{J\overline{Q}}$ und $S = \overline{KQ}$, die durch T getaktet werden. Daraus sieht man, dass $J = K = 0$ ein $R = S = 1$ erzwingt, was den Master in Speichermodus versetzt.

Ist $J = \overline{K} = 1$, so erzwingt $Q = 1$ ein Speichern des Masters (Q bleibt also unverändert), ein $Q = 0$ führt zu $Q = 1$. Das heisst aber, dass für diesen Fall der Slave–Ausgang immer $Q = 1$ wird, was einem Setzen des Speichers entspricht. Analog führt $K = \overline{J} = 1$ zu einem Rücksetzen von Q.

Sind die Eingänge $J = K = 1$, so kippt Q immer in den negierten Zustand (prüfen Sie das nach). Damit kommt man zu der (vereinfachten) Transitionstabelle 3.14 für das J–K–Flip–Flop.

J	K	$Q(t + \Delta t)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\overline{Q(t)}$

Tabelle 3.14: Vereinfachte Transitionstabelle des J–K–Flip–Flops.

Die Schaltsymbole für Varianten des J–K–Flip–Flops zeigt Abb. 3.30.

Neben den J/K–Eingängen ist ein Eingang für den Takt und die daraus abgeleitete Flankensteuerung dargestellt. Eingänge ohne Negationssymbol haben definitionsgemäss positive Logik. Für den Takteingang bedeutet

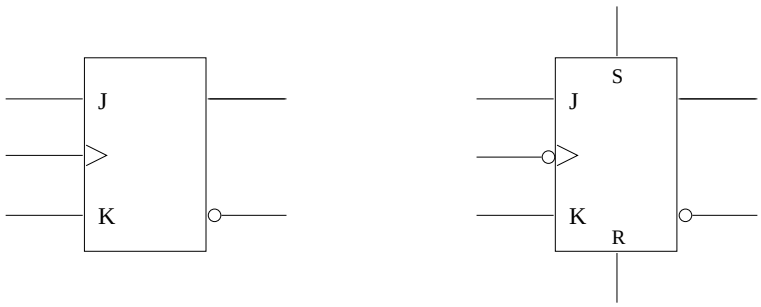

Abbildung 3.30: Schaltsymbole für R–S–Flip–Flop ohne (links) und mit asynchronen R/S–Eingängen (rechts).

diese positive Flankensteuerung (das Flip–Flop in Abb. 3.30 rechts hat negative Flankensteuerung). Die asynchronen R/S–Eingänge ermöglichen ein Setzen/Rücksetzen des J–K–Flip–Flops unabhängig von seinem aktuellen Zustand (z. B. für Initialisierungen).

Mit dem J–K–Flip–Flop lassen sich nun wiederum viele Grundbausteine digitaler Rechenanlagen aufbauen.

3.5.6 Parallelregister

Ein *Parallelregister* (Abb. 3.31) aus J–K–Flip–Flops übernimmt die anliegenden Daten gleichzeitig (flankengesteuert) und speichert diese bis zur nächsten Flanke (Schreiboperation).

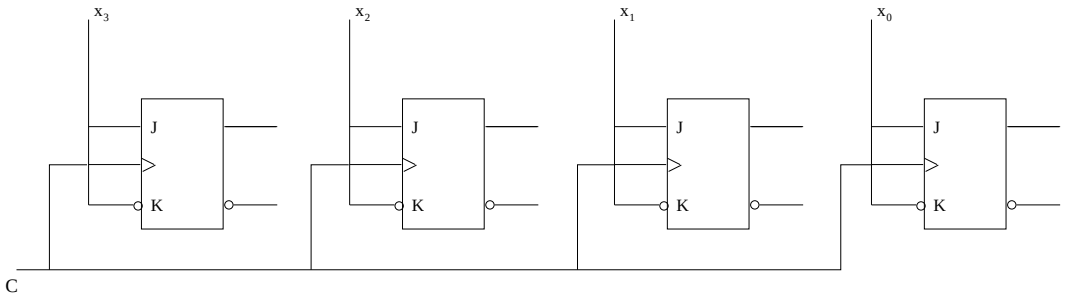

Abbildung 3.31: Ein 4–Bit Parallelregister.

Die Eingänge der J–K–Flip–Flops werden hier einfach so vorbereitet ($J = \overline{K}$, dass der Ausgang Q dem Eingang $J = x_i$ folgt.

3.5.7 Schieberegister

Ein *Schieberegister* (Abb. 3.32) verschiebt seinen Inhalt bei jeder Taktflanke in das nächste Flip–Flop. Diese Verschiebung (*Shift*) ist die Basis vieler verschiedener Operationen (Arithmetik, Codierung, Fehlerkorrektur, etc.).

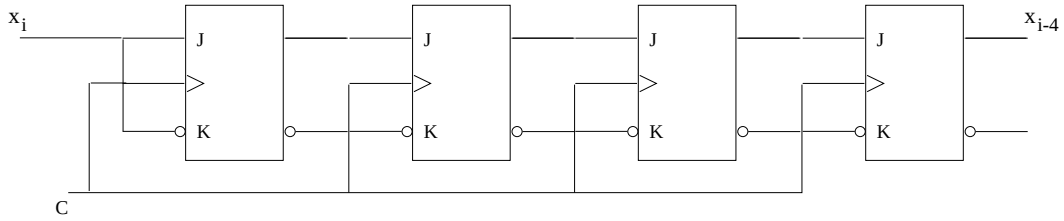

Abbildung 3.32: Ein 4–Bit Schieberegister.

Hier wird der Eingang x_i mit jeder Takftlanke in das nächste Flip–Flop geschrieben. Der Ausgang kann damit auch als zeitverzögerter Eingang interpretiert werden (digitales Verzögerungsglied).

3.5.8 Binärzähler

Ein weiterer sehr wichtiger Grundbaustein der meisten digitalen Rechenanlagen ist der *Zähler*. Er wird verwendet, um die Anzahl von Ereignissen (siehe Parkschranke) zu zählen, um Programme zu steuern, und er wird auch zur Zeitmessung herangezogen. Abb. 3.33 zeigt einen *Asynchronen Binären Vorwärtszähler*, der ebenfalls einfach aus J–K–Flip–Flops aufgebaut werden kann.

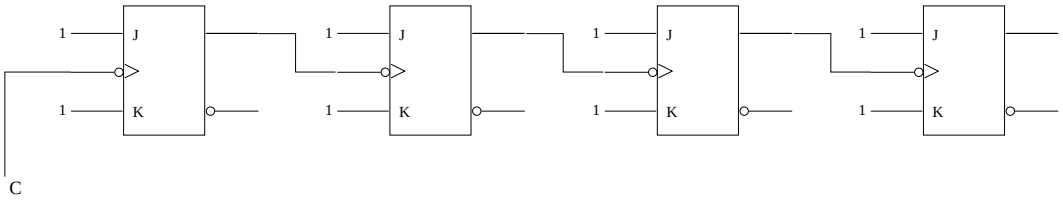

Abbildung 3.33: Asynchroner Binärer Vorwärtszähler.

Interessant dabei ist, dass hier ein externer Takt nur für das erste Flip–Flop (FF) Verwendung findet. Alle FFs sind über die konstanten Eingänge zum Kippen vorbereitet. Der Takt für die restlichen Flip–Flops wird aus den Ausgängen der vorherigen abgeleitet. Ist der Zähler in Abb. 3.33 in Ausgangsstellung 0000 (die man über asynchrone Setzeingänge erzwingen könnte), dann bewirkt die erste negative Flanke des Taktsignals ein Kippen des

ersten Flip–Flops (FF). Dieses ändert daher seinen Ausgang von 0 auf 1 (positive Taktflanke), was für das zweite FF nichts bewirkt. Beim nächsten Kippen des ersten FF geht der Ausgang von 1 auf 0 (negative Taktflanke), damit kippt der Ausgang des naechsten FF von 0 auf 1. Anders betrachtet verdoppelt sich nach jedem Flip–Flop die Periodendauer des Takts.

Behält man alle 4 Ausgänge der FFs im Auge, so erkennt man, dass bei jeder negativen Taktflanke am ersten FF (links) die binäre 4–Bit–Zahl um 1 hinaufgezählt wird (LSB ist das FF links, MSB rechts) (was geschieht bei Zählerstand 1111?).

Eigentlich übernimmt hier der Takt die Rolle des (einzigen) Eingangs des Zählers und die Schaltung zählt die Anzahl der negativen Flanken am Eingang. Damit hat man aber auch schon die Möglichkeit Zeit zu messen, da bei bekannter Taktfrequenz, die Zeitdauer zwischen zwei negativen Flanken (eines periodischen Signals) bekannt ist.

Der Zähler arbeitet asynchron, da sich die Zustände der einzelnen FFs nur *nacheinander* ändern können, aber nicht gleichzeitig. Es kommt daher beim Kippen eines FFs kurzzeitig immer zu Fehlanzeigen des Zählers, die aber meist keine Rolle spielen. Z. B. könnte man den Zählerstand mit jeder positiven Taktflanke in ein Latch übertragen, womit man mit sehr hoher Wahrscheinlichkeit gewährleistet, dass der Zählerstand im Latch gültig ist.

Trotzdem gibt es Anwendungsfälle bei dem die mangelnde Synchronizität des Zählers zum Problem wird. Daher konstruieren wir in folgendem Beispiel einen *Synchronen Binären Vorwärtszähler*.

Beispiel: Wir wollen einen *Synchronen Binären Vorwärtszähler* wieder aus 4 J–K–Flip–Flops aufbauen. Um den Zähler zu synchronisieren, muss der gemeinsame Takt an allen FFs anliegen. Die notwendigen Ausgangsänderungen müssen dann durch unterschiedliche Eingangswerte erzwungen werden. Man nennt daher die J/K–Eingänge auch *Vorbereitungseingänge*, die das FF auf eine Änderung (oder eine Speicherung) des Ausgangs vorbereiten soll.

Im Sinn des Huffman–Modells sind die FFs der Speicherteil des Schaltwerks und das Schaltnetz, das am Eingang den inneren Zustand (Zählerstand) erhält und daraus die Vorbereitungseingänge generiert, ist der kombinatorische Schaltkreis.

Im folgenden werden wir daher eine Transitionstabelle aufstellen, die alle möglichen Zählerstände (Zustandsübergänge) enthält, und für jeden Übergang die benötigten Vorbereitungseingänge angeben. Dazu überlegen wir uns zuerst einmal die möglichen elementaren Zustandsübergänge und die dazugehörigen Vorbereitungen (Tab. 3.15).

Das X in obiger Tabelle steht für beliebige Vorbereitung des jeweiligen Eingangs (ist also nicht die Totovorhersage für das kommende Wochenende). Mit diesem Wissen können wir die Transitionstabelle 3.16 aufstellen und die

Übergang	J	K
$0 \to 0$	0	X
$0 \to 1$	1	X
$1 \to 0$	X	1
$1 \to 1$	X	0

Tabelle 3.15: Vorbereitungen des J–K–Flip–Flops für Ausgangsübergänge.

benötigten Vorbereitungen daraus ableiten.

t	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Q_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Q_2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Q_1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Q_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
J_0	1	X	1	X	1	X	1	X	1	X	1	X	1	X	1	X
K_0	X	1	X	1	X	1	X	1	X	1	X	1	X	1	X	1
J_1	0	1	X	X	0	1	X	X	0	1	X	X	0	1	X	X
K_1	X	X	0	1	X	X	0	1	X	X	0	1	X	X	0	1
J_2	0	0	0	1	X	X	X	X	0	0	0	1	X	X	X	X
K_2	X	X	X	X	0	0	0	1	X	X	X	X	0	0	0	1
J_3	0	0	0	0	0	0	0	1	X	X	X	X	X	X	X	X
K_3	X	X	X	X	X	X	X	X	0	0	0	0	0	0	0	1

Tabelle 3.16: Transitionstabelle des synchronen 4–Bit–Vorwärtszählers.

Aus der Transitionstabelle können wir in diesem Fall die notwendigen Vorbereitungen einfach ablesen. Für alle Eingänge gilt, dass wir $J = K$ setzen können, da dies die Verteilung der X ermöglicht, für die wir wahlweise 0 oder 1 setzen können.

Aus $J_0 = K_0 = 1$ folgt sofort, dass das FF0 immer auf Kippen vorbereitet werden kann. Dies ist auch an der Struktur der Übergänge von Q_0 erkennbar und ist völlig analog zum asynchronen Fall.

Für das FF1 ergibt sich $J_1 = K_1 = Q_0$. Dieses FF erhält also einfach den Ausgang von FF0.

Für FF2 folgt $J_2 = K_2 = Q_0 Q_1$. Dies ist hier einfach aus der Transitionstabelle abzulesen, muss aber (wie behandelt) im allgemeinen Fall über die vollständige DNF und einem Minimierungsverfahren abgeleitet werden.

Ebenso leicht lässt sich fuer das FF3 die Vorbereitung $J_3 = K_3 = Q_0 Q_1 Q_2$ ableiten. Wir erhalten daher für den synchronen binären 4–Bit–Vorwärtszähler das Schaltwerk in Abb. 3.34.

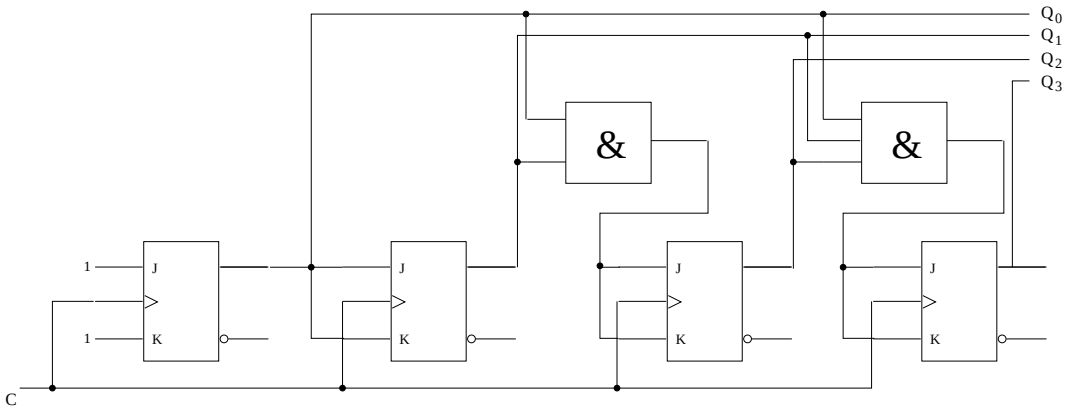

Abbildung 3.34: Synchroner Binärer Vorwärtszähler.

◊

In ganz ähnlicher Weise lassen sich verschiedene Zähler und andere Schaltwerke aufbauen. Als erster Schritt sollte dabei immer die prinzipelle Aufteilung in Schaltnetz und Speicher (Huffman–Modell) stehen. Anders ausgedrückt muss man unterscheiden, welche Grössen den Zustand des Schaltwerks beschreiben und welche Grössen den Ein/Ausgang des Systems beschreiben (im Spezialfall des Zählers sind Zustands– und Ausgangsvariablen identisch). Die Schaltnetze lassen sich dann aus den Transitionstabellen in bekannter Weise ableiten. Die Zustandsgrössen werden durch Flip–Flops gespeichert und verändert.

Kapitel 4

Rechnerarchitektur

Im Prinzip ist auch ein heutiger Rechner (*Computer*) ein einziges grosses Schaltwerk, das von einer zentralen Taktfrequenz gesteuert wird. Die Komplexität eines derartigen Schaltwerks ist allerdings so gross, dass man schon zu Beginn der Konstruktion von Computern einige wesentliche Teilschaltwerke identifizierte, und diese als Teile des Aufbaus von Rechnern, i. e. der *Rechnerarchitektur* sah. Die Grundstruktur fast aller heutigen Computer basiert auf der *von Neumann–Architektur*, die der ungarische Mathematiker mit US–amerikanischem Pass *John von Neumann* erstmals 1945 formulierte.

Der offensichtliche, enorme Fortschritt der Rechnerleistung, der seither erzielt wurde, beruht zum grössten Teil auf rein technologischen Verbesserungen. Die prinzipielle Architektur blieb jedoch unverändert. So konnte mit Einführung der Halbleitertechnologie der Platzbedarf, die Geschwindigkeit und die Speicherkapazität von Computern fast *exponentiell* gesteigert werden, was auch in *Moore's Law* zum Ausdruck kommt:

> The density of transistors on a chip will double every 18 months, thus increasing the price performance of computing power by a factor of two every 1 1/2 years. – Gordon Moore (Co–Founder, INTEL Corp.), 1978

Hier tritt auch schon das Problem der Quantifizierung der *Computing Power* auf, das wir später noch näher behandeln werden.

4.1 Eine kleine Computergeschichte

Bevor wir uns der Frage zuwenden, was ein (komplexes) Schaltwerk zu einem Computer macht, noch kurz einige Meilensteine der Geschichte der Computer (INFOTEC, 1998).

3000 v. Ch.: Sandabakus (wahrscheinlich in Babylonien)

700–800 n. Ch.: Arabische Zahlen verbreiten sich in Europa (Konzept der Null und der stellengewichteten Ziffer)

1457: Gutenberg erfindet die Druckerpresse

1642: Blaise Pascal baut erste numerische Rechenmaschine (Paris)

1833: Charles Babbage plant die Analytical Machine mit Programmen auf Lochkarten

1844: Samuel Morse sendet Nachricht von Washington nach Baltimore

1936: Konrad Zuse baut ersten Relais–Rechner *Z–1* (Berlin)

1943: John William Mauchly und John Presper Eckert beginnen Entwicklung von ENIAC (Electronic Numerical Integrator And Calculator), dem ersten elektronischen Allzweck–Rechner (Masse: 30m lang, 3m hoch, 2m breit, 18,000 Röhren, 1 Addition 200μs, University of Pennsylvania)

1945: John von Neumann beschreibt die *stored–program*–Architektur

1948: William Bradford Shockley, John Bardeen und Walter H. Brattain erfinden den Transistor

1953: Erstes Magnetbandgerät *IBM 726* (40 Zeichen/cm, 7500 Zeichen/sec)

1954: John Backus entwickelt FORTRAN (IBM)

1958: Seymour Cray baut ersten volltransistorisierten Computer *CDC 1604* (Control Data Corp.)

1970: Gilbert Hyatt reicht Patent für "Single Chip Integrated Circuit Computer Architectur" ein (Mikroprozessor)

1970: ARPANET mit 15 Knotenrechnern (Start des Internet)

1971: Intel Corporation präsentiert ersten Mikroprozessor *Intel 4004*

1977: *Apple*, *Commodore* und *Tally* verkaufen erste *Personal Computers* (PCs)

1981: *IBM* bringt den *IBM 5150 PC* mit *DOS*–Betriebssystem von *Microsoft* (4.77 MHz *Intel 8088* CPU, 4KB RAM, 40KB ROM, 5.25 floppy drive, Grundpreis 3000 US$)

1988: DEC beginnt Entwicklung von 64 bit, 150 MHz Alpha Prozessor

1989: 100,000 Rechner am Internet

1995: *Intel* präsentiert *Pentium Pro* mit 150 MHz

1996: 13,000,000 Rechner am Internet

2000: Erste 1 GHz–Prozessoren (*AMD*, *Intel*) in PCs

2000: 93,000,000 Rechner am Internet [1]

2001: 126,000,000 Rechner am Internet [1]

4.2 Die von Neumann–Architektur

Was macht nun ein Schaltwerk zu einem Computer? Nehmen wir als Beispiel einen einfachen Taschenrechner als (auch historische) Vorstufe zu einem Computer. Jener besitzt ein *Rechenwerk*, das verschiedene einfache arithmetische Operationen ermöglicht. Über eine Eingabe (*Input*) werden Zahlen und Operationen eingegeben, deren Ergebnis (*Output*) am *Display* angezeigt wird. Addieren wir zwei Zahlen $A + B$ und danach $A + C$, so ist das Ergebnis der ersten Addition vergessen, wenn der Taschenrechner keinen *Speicher* aufweist. Wir müssen also wiederum die beiden Zahlen und die Operation "Addition" eingeben, um das erste Ergebnis neuerlich zu erhalten.

Hätte unser Rechner einen Speicher, so könnten wir nicht nur die Ergebnisse speichern, sondern auch die ursprünglichen Zahlen A, B und C. Um diese Zahlen allerdings aus dem Speicher zu holen, müssen wir wissen, wo welche Zahl gespeichert ist, um die richtige Zahl anzusprechen. Wir benötigen also eine Kennzeichnung des Speichers, die *Adressierung* genannt wird. Jede Zahl ist damit an einer definierten Addresse "beheimatet".

Wenn wir jetzt mehrere verschiedene Operationen mit unseren drei Zahlen ausführen möchten, dann müssten wir bei Veränderung von A, B und C die Abfolge von Operationen wiederum eingeben und die jeweiligen Zahlen aus dem Speicher mit der richtigen Adresse holen. Alternativ könnten wir aber auch die Abfolge von Operationen – ein *Programm* P – im Speicher ablegen. Wenn sich unsere Zahlen – die *Daten* D – ändern, so bräuchten wir dem Rechner nur mitteilen "Führe das Programm P aus". Das Programm würde dann die Daten D von den entsprechenden Speicherstellen holen, sie verarbeiten, und die neuen Ergebnisse wiederum abspeichern. Auch das Programm selbst muss aus dem Speicher geholt werden. Das Prinzip der Speicherung

[1] Internet Software Consortium (http://www.isc.org/)

von Daten und Programm in einem gemeinsamen Speicher ist die Grundidee der *von Neumann–Architektur*, die auch als *stored–program*–Architektur bezeichnet wird (Abb. 4.1).

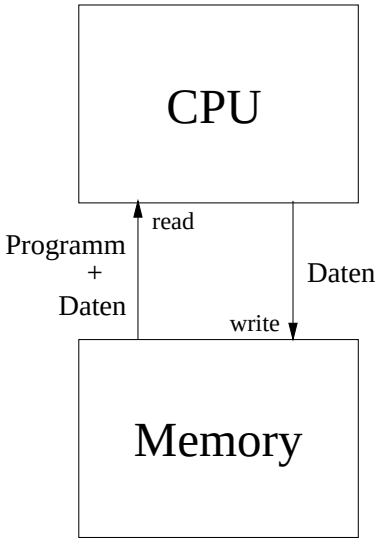

Abbildung 4.1: Prinzip der von Neumann–Architektur.

Hierbei holt eine *Central Processing Unit* (CPU) Programm und Daten aus dem gemeinsamen Speicher (*Memory*), führt die einzelnen Anweisungen (Operationen, *Instructions*) aus und schreibt Ergebnisdaten wieder in den Speicher zurück. Etwas genauer formuliert führt die CPU folgende Schritte aus:

1) Hole Anweisung aus dem Speicher

2) Hole die Daten für diese Anweisung

3) Führe die Anweisung aus

4) Speichere das Ergebnis der Anweisung

5) Gehe zu 1) oder Programmende

Die Leitungen, die die Verbindung zwischen CPU und Speicher herstellen, werden als *Bus* bezeichnet. Dabei unterscheidet man prinzipiell zwischen *Adress–* und *Datenbus*. Um ein Datum oder eine Anweisung aus dem Speicher zu holen, muss die CPU zuerst die Adresse des jeweiligen Speicherplatzes angeben. Die Binärdarstellung der Adresse wird an den Adressbus *angelegt*, was voraussetzt, dass die *Busbreite* (Anzahl der parallelen Leitungen)

diese Adresse aufnehmen kann. Die Busbreite bestimmt also den adressierbaren Speicherraum. Hat die CPU eine Adresse ausgewählt, so werden die Daten/Instruktionen über den Datenbus zurück an die CPU gesendet. Will man also z. B. ein Byte übertragen, so sollte der Datenbus aus 8 parallelen Leitungen bestehen. Ein 16–Bit–Wort müsste dann in zwei Schritten (von zwei verschiedenen Adressen) geholt werden, was doppelt so lange dauert. Ein (aufwendigerer) Datenbus aus 16 Leitungen würde für diesen Fall eine Verdopplung der Geschwindigkeit bewirken. Wir können hier schon die ganz allgemeine Erkenntnis ableiten, dass alle Hardwareverbesserungen die Geschwindigkeit eines Computersystems erhöhen.

4.3 Der Universalrechenautomat

Im folgenden werden wir uns immer weiter in das Innere der CPU, des Speichers, und des Bussystems vorarbeiten. Eine erste Verfeinerung der prinzipiellen von Neumann–Architektur ist der *Universalrechenautomat* nach Händler (Volkert, 1999), dessen Struktur in Abb. 4.2 gezeigt ist.

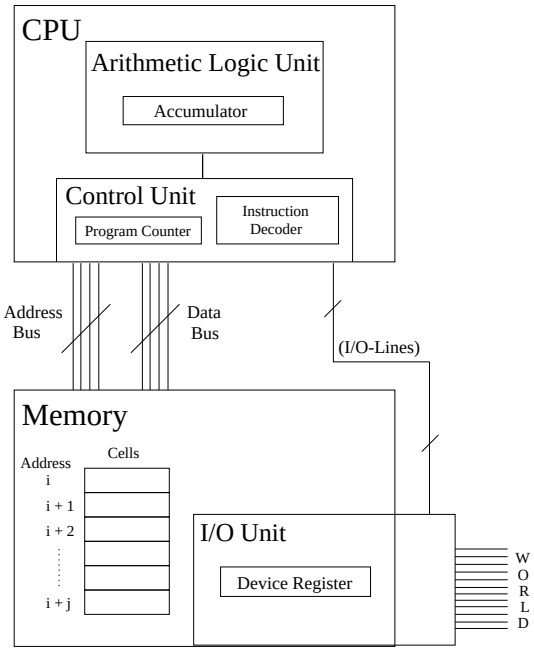

Abbildung 4.2: Ein Universalrechenautomat (URA).

Die Hauptbestandteile eines Computers – CPU und Speicher – werden hier folgendermassen unterteilt:

- CPU
  - Control Unit: Steuerwerk für Programmsteuerung und Befehlsdecodierung
  - Arithmetic Logical Unit: Rechenwerk für arithmetische und logische Operationen
- Speicher
  - Adressierbare Speicherzellen
  - I/O Unit: Ein/Ausgabe von Daten und Programmen (Tastatur, Monitor, Maus, Drucker, Scanner etc.)

Bei der Ansteuerung von I/O–Geräten (*Devices*) unterscheidet man zwischen *memory mapped I/O* und I/O Units, die über spezielle Leitungen mit der CPU verbunden sind. Heute wird fast nur mehr memory mapped I/O verwendet, bei dem die einzelnen Ein/Ausgabe–Geräte Teil des Adressraums des Speichers sind. Ein Gerät wird dann über eine Speicheradresse "beschrieben" und "gelesen", d.h. man schreibt entsprechende Befehlssequenzen in die Register, die an dieser Speicheradresse liegen, um das Gerät zu steuern. Vielfach übernehmen dann eigene Prozessoren in den Geräten die weitere Verarbeitung. Das Schreiben und Lesen einzelner *Device Register* unterscheidet sich aus Sicht der CPU nicht von dem Umgang mit einfachen Speicherzellen.

Die Funktionsweise des URA kann zusammenfassend beschrieben werden:

1) Der URA wird logisch und räumlich in die Teile CPU (Control Unit, ALU), Memory und I/O Unit zerlegt.

2) Das Programm des URA wird über die I/O Unit eingegeben und im Speicher abgelegt. Das Programm ermöglicht die Bearbeitung eines Problems, das von der Struktur des Rechners *unabhängig* ist.

3) Programm und Daten sind im selben Speicher.

4) Der Speicher ist eine numerierte Folge von Speicherzellen. Die Nummer einer Speicherzelle ist deren Adresse, über die der Inhalt der Speicherzelle gelesen oder geschrieben werden kann.

5) Ein Programm ist eine Folge von Befehlen, die in aufeinanderfolgenden Speicherzellen abgelegt sind. Wird ein Befehl von der Adresse a geholt und abgearbeitet, dann wird der nächste Befehl (im Normalfall) von der Adresse $a + 1$ geholt.

6) *Sprungbefehle* bewirken eine Änderung der Reihenfolge der Befehlsarbeitung, die bewirkt, dass der nächste Befehl nicht von der Adresse $a + 1$, sondern von $a \pm n$ geholt wird.

7) *Bedingte Sprünge* sind Sprungbefehle, die nur ausgeführt werden, wenn eine (programmierte) Bedingung erfüllt wird. Ist die Bedingung nicht erfüllt, wird der nächste Befehl (wie üblich) von der Adresse $a + 1$ geholt.

8) Programm und Daten sind binär codiert.

Von diesen acht wesentlichen Punkten ist vor allem Punkt 7) für die Flexibilität des URA verantwortlich. Diese Eigenschaft ermöglicht nämlich, dass sich das Programm in Abhängigkeit von den (zunächst unbekannten) Eingabedaten verhält. Ohne diese Eigenschaft wären z. B. Benutzeroberflächen (Interaktion mit dem Anwender) nicht möglich, und der Rechner damit nicht universell verwendbar.

4.4 Die Architektur des Befehlssatzes

Die primäre Schnittstelle zwischen einem Programmierer und einem Computer ist der Befehlssatz der verwendeten CPU. Verwendet die Programmiererin eine *Hochsprache*, so übernimmt ein *Compiler* die Übersetzung der Anweisungen in Maschinenbefehle. Ein Compiler (*Übersetzer*) ist also ein Hilfsmittel, das dem Programmierer erlaubt, eine CPU (*Maschine*) zu programmieren, ohne deren intrinsischen Befehlssatz zu kennen. Trotz der fortschreitenden Compilertechnologie gibt es auch heute noch Einsatzgebiete, die eine direkte Programmierung in *Maschinensprache* nötig machen (zeit–, und speicherkritische Anwendungen).

Die wichtigste Frage für den Entwickler einer CPU ist daher die Entscheidung über Art und Anzahl der verschiedenen Befehle einer Maschinensprache. Einerseits sollten die Befehle Programme aus allen möglichen Bereichen unterstützen, andererseits würden zuviele verschiedene Befehle die Komplexität einer CPU, und damit die Geschwindigkeit und Programmierbarkeit, negativ beeinflussen. Ganz wesentlich für das Aussehen des Befehlssatzes ist auch das Zusammenspiel zwischen CPU und Speicher.

4.4.1 Maschinenarchitektur

Die wesentlichste Unterteilung des Aufbaus einer CPU in *Maschinenarchitekturen* rührt vom Aufbau des *internen* Speichers einer CPU (Hennessy and

Patterson, 1996). Daten und Befehle, die vom Speicher geholt werden, müssen zwischengespeichert werden, um die Befehle abarbeiten zu können. Aufbau, Manipulierbarkeit und Kapazität des internen Speichers bestimmen somit wesentlich die Architektur des Befehlssatzes.

Zur Illustration der folgenden Maschinenarchitekturen werden wir unser erstes (sehr kleines) Maschinenprogramm schreiben. Unsere Aufgabe ist es zwei Zahlen A, B aus dem Speicher zu addieren und das Ergebnis in die Speicherstelle C abzulegen. Wir verwenden also symbolische Namen und keine Zahlen für die Speicherstellen, was eigentlich schon den Abstraktionsschritt von der Maschinenprogrammierung zur *Assemblerprogrammierung* darstellt. Ein *Assembler* übersetzt die *Mnemonics* der einzelnen Befehle und die symbolischen Adressen in Maschinensprache, was für den menschlichen Programmierer eine wesentliche Erleichterung darstellt, die mit Namen mehr anfangen kann als mit binären Zahlen.

Die Stackmaschine

Der interne Speicher einer *Stack–Maschine* wird durch einen *Stapelspeicher* (Stack) gebildet, dessen prinzipieller Aufbau in Abb. 4.3 dargestellt ist.

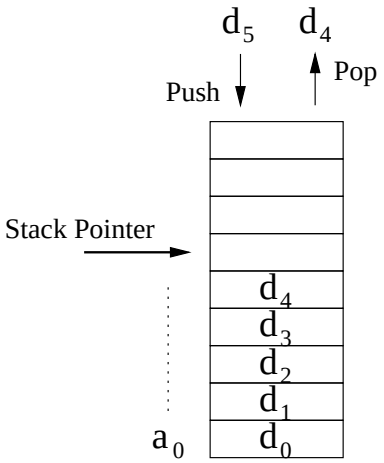

Abbildung 4.3: Ein Stack (Stapelspeicher).

Der *Stack Pointer* (ein internes Register) zeigt dabei immer auf die nächste freie Speicherstelle, in die ein neues Datum aus dem Memory mit dem Befehl `PUSH` geholt wird. Der Befehl `POP` holt das "oberste" Datum vom Speicher, was dieser Datenstruktur die Bezeichnung *LIFO*–Speicher (Last In First Out) gibt, der auch in der Softwaretechnik eine wichtige Rolle spielt. Diese Speicherorganisation unterstützt ganz elementar die *Umgekehrte Polnische Notation* (UPN), bei der arithmetische Operationen in einer speziellen Reihenfolge

ausgeführt werden. So wird die Addition $A + B$ als $AB+$ geschrieben, was auch zum Wegfall von eventuellen Klammern führt ($(A + B) * (C + D)$ wird in UPN zu $AB + CD + *$).

Unser Additionsprogramm für die Stackmaschine würde dann folgendermassen aussehen (mit erfundenen aber durchaus üblichen Assemblerbefehlen):

```
PUSH A     ; hole Datum von Speicherstelle A und lege es auf den Stack
PUSH B     ; hole B auf den Stack
ADD        ; addiere und speichere das Ergebnis am Stack
POP C      ; hole Datum vom Stack und lege es in Speicherstelle C
```

Während die ersten CPUs vielfach Stackmaschinen waren, ist diese Architektur bei den meisten heutigen CPUs nicht mehr anzutreffen. Der Hauptgrund dafür ist, dass immer nur das oberste Datum am Stack explizit angesprochen werden kann, und Daten oft mehrmals aus dem Speicher geholt werden müssen (in unserem Beispiel wäre A und B nach Addition nicht mehr in der CPU). Andererseits ist das Ausführen von Operationen sehr kompakt, da man keinerlei Adressen von Operanden mehr angeben muss, sobald diese auf dem Stack sind (siehe `ADD`). Die einzelnen Befehle produzieren daher einen *kompakten* Code, da nur die Operation, nicht aber die Adresse der Operanden codiert werden muss. Letzteres ist auch der Grund warum man die Stackmaschine auch als *Null–Adressmaschine* bezeichnet.

Eine *Virtual Machine* (VM) ist eine CPU, die nur aus Software besteht, d. h. es ist die Nachbildung einer physikalischen CPU. Die VM "lebt" daher im Speicher, wodurch der Nachteil des wiederholten Ladens von Daten in die CPU wegfällt, die effiziente Befehlsabarbeitung jedoch erhalten bleibt. Dies ist ein wesentlicher Grund dafür, dass die *Java Virtual Machine*, auf der Java–Programme exekutiert werden, als Stackmaschine implementiert ist.

Die Akkumulatormaschine

Viele frühe Maschinen hatten ein einziges internes Register, das vom Programmierer angesprochen werden konnte. Ein solches Register bezeichnet man als *Akkumulator*. Unser Beispielprogramm würde hier folgende Gestalt annehmen:

```
LOAD A     ; hole Datum von Speicherstelle A in den Akkumulator
ADD B      ; addiere B zum Akkumulatorinhalt
STORE C    ; schreibe Akkumulatorinhalt nach C
```

Der Code (Maschinenprogramm) für die Akkumulatormaschine ist zwar auch recht kompakt, da implizit jeder Datentransfer und jede Operation über den Akkumulator läuft, der nicht adressiert werden muss. Wie bei der Stackmaschine bleibt aber das Problem, dass Daten immer wieder geladen werden müssen, wenn sie mehrmals verwendet werden. Die Befehle für die Akkumulatormaschine weisen immer nur eine Adresse auf, daher wird sie auch als *Ein–Address–Maschine* bezeichnet.

Die Registermaschinen

Mit fortschreitender Technologie wurde es immer einfacher und billiger dem Programmierer einen Satz von allgemein verwendbaren Registern zur Verfügung zu stellen. Dies hat den wesentlichen Vorteil, dass oft verwendete Daten in der CPU zwischengespeichert werden können, was gleich mehrere Vorteile bringt: die Befehle für das Nachladen von Daten entfallen (damit das Holen und Ausführen dieser Befehle *und* das Nachladen). Die Bearbeitung dieser Daten in der CPU erfolgt schneller, da die *General Purpose Registers* in der CPU auf Geschwindigkeit optimiert werden können.

Die Registermaschinen können noch in *Register–Memory–*, und *Register–Register–*Maschinen unterteilt werden. Das Beispielprogramm für eine Register–Memory–Maschine lautet:

```
LOAD R1, A     ; hole Datum von Speicherstelle A nach Register R1
ADD R1, B      ; addiere B zu R1
STORE C, R1    ; speichere R1 nach C
```

Auch hier müsste man die Variablen A und B bei mehrmaliger Verwendung nachladen, doch durch einen zusätzlichen Befehl, der die Addition von Registern erlaubt (`ADD R1, R2`) könnte man Variablen und Teilergebnisse in der CPU zwischenspeichern. Allerdings muss der Programmierer explizit Befehle einfügen, damit dies auch wirklich geschieht. Da bei all diesen Operationen immer zwei Operanden addressiert werden müssen, heisst diese Maschine auch *Zwei–Adress–Maschine.*

Unsere kleine Addition auf einer Register–Register–Maschine:

```
LOAD R1, A      ; hole Datum von Speicherstelle A nach Register R1
LOAD R2, B      ; hole Datum von B nach R2
ADD R3, R2, R1  ; addiere Inhalt von R1 und R2 nach R3
STORE C, R3     ; speichere R3 nach C
```

Trotz der etwas aufwendigeren Befehle und Adressierungen stehen nach diesen Anweisungen alle Variablen und das Ergebnis auch nach dem Abspeichern in der CPU zur Verfügung. Da bei dieser Maschinenarchitektur

alle Daten in Register geladen werden, dort dann verarbeitet, und von den Registern wieder gespeichert werden, nennt man diesen Maschinentyp auch *Load–Store–Maschine*. Bei Registeroperationen müssen drei Register adressiert werden, daher heisst sie auch *Drei–Adress–Maschine*.

Theoretisch könnte man natürlich auch eine *Memory–Memory*–Maschine bauen, bei der man zwei Operanden "direkt" im Speicher addieren könnte. Trotz kompakten Codes wäre diese Maschine aber langsam, da bei *jeder* Addition zwei Operanden in den Speicher geholt, addiert, und das Ergebnis rückgespeichert werden müsste. Die meisten heutigen Maschinen sind daher vom Typ Register–Register.

Ein Vorteil der *Load–Store*–Architektur ist auch, dass alle Befehle ähnliche Ausführungszeiten haben. Die Ausführungszeit wird meist in *Clock Cycles* (Taktzyklen) angegeben. Geringe Unterschiede in den Ausführungszeiten ermöglichen ein effizientes *Pipelining* von Befehlen, das wir später noch ausführlich behandeln werden. Daher wählen wir für die hypothetische Maschine *DLX* (Kapitel 5) die *Load–Store*–Architektur.

4.4.2 Speicheradressierung

Wir wissen bereits, dass Speicherzellen über Nummern (Adressen) angesprochen werden, doch verbergen sich hinter dieser einfachen Feststellung viele wichtige Details. Zuerst stellt sich einmal die Frage wieviele Bits an einer Speicheradresse stehen. Aus historischen Gründen sind die meisten heutigen Prozessoren immer noch *byte–adressiert*, d. h. an jeder Adresse wird ein Byte gespeichert. Zusätzlich können heute auch Halbwörter (16 Bits), Wörter (32 Bits, *word*) und Doppelwörter (64 Bits) im Speicher angesprochen werden, die sich dann allerdings über entsprechend viele Adressen erstrecken.

Beispiel: In einem Speicher stehen ab Speicheradresse 1000 16 Bytes in hintereinanderliegenden Speicherzellen. Wir könnten dann von Adresse 1007 ein Byte lesen. Von Adresse 1004 könnten wir ein Wort lesen, wobei das nächste Wort an Adresse 1008 stehen würde. Von 1008 können wir aber auch ein Doppelwort (8 Bytes) lesen. (Was würde passieren, wenn wir das nächste Doppelwort lesen würden?) ◊

In obigem Beispiel stehen die einzelnen Daten an speziellen Speicherstellen. Man sagt, die Daten sind *ausgerichtet* (*aligned*). Für ausgerichtete Daten muss gelten $A \ mod \ s = 0$, wobei A die Adresse eines Datums und s die Grösse des Datums in Bytes ist. Viele Prozessoren sehen aus Hardwaregründen vor, dass die Daten ausgerichtet sein müssen. Ein Lesen/Schreiben eines Halbworts an einer ungeraden Adresse würde dann zu einem *Address Error* führen. Der Programmierer muss also dafür Sorge tragen, dass solche inkorrekten Zugriffe nicht auftreten.

Ein weiteres einfaches Problem, über das aber schon unzählige Programmierer gestolpert sind, ist die Ordnung der Bytes (*Byte Order*). Ist ein Halbwort an einer Adresse gespeichert, so gibt es genau zwei Möglichkeiten die beiden Bytes abzuspeichern. Entweder steht das MSByte an der Adresse, oder das LSByte. Wenn das höherwertige Byte (Bits 15–8) an der Adresse steht, so nennt man dies *Big Endian* (grosses Ende). Im anderen Fall sind die Bytes im *Little Endian*–Format (kleines Ende) gespeichert. Innerhalb einer Maschine bereiten diese Unterschiede meist kein Problem, da die Hardware auf das entsprechende Format vorbereitet ist. Tauschen zwei verschiedene Maschinen jedoch Daten aus (Netzwerk), dann muss die Byteordnung *unbedingt* berücksichtigt werden.

Wenn jede Adresse über eine eigene Nummer angesprochen werden kann, ist der gesamte Speicher für ein Programm zugänglich. Allerdings wird die Nummer grösser, je grösser der Speicher wird. Damit wird aber auch ein Befehl, der diese Speicherstelle anspricht grösser, da das *Adressfeld* mehr Bits bereitstellen muss (überlegen Sie wieviele Bits zur Adressierung eines 64 *MByte*–Speichers reserviert sein müssen). Dies würde aber wieder den Speicherbedarf von Programmen unnötig erhöhen. Ausserdem stehen Daten oft miteinander in Beziehung (Arrays, Matrizen), was man durch effiziente *Adressierungsmodi* unterstützen kann. Die gebräuchlichsten Möglichkeiten der Adressierung wollen wir nun näher betrachten.

Direkte Adressierung

Dieser Adressierungsmodus wird auch als *Absolute Adressierung* bezeichnet, da die absolute Speicheradresse (Nummer) angegeben wird. Als Beispiel für die verschiedenen Modi werden wir einen `MOVE`–Befehl betrachten, der ein Datum in das Register $R1$ transferiert.

```
MOVE R1, 1000     ; hole Inhalt der absoluten Speicheradresse 1000
```

Die Grösse des Datums hängt von der Spezifikation des `MOVE`–Befehls ab. Mit diesem Modus werden z. B. I/O–Devices angesprochen, die an einer systemabhängigen Speicherstelle angesiedelt sind. Ein Nachteil dieses Modus ist der Platzbedarf des Befehls (siehe oben). Viel günstiger ist es, wenn das Datum schon in einem Register steht.

```
MOVE R1, R2     ; hole Inhalt des Registers R2
```

Indirekte Adressierung

Dieser Modus entspricht dem *Pointer*–Konzept in Hochsprachen.

```
MOVE R1, (1000)     ; hole Inhalt der Adresse, die in 1000 steht
```

Es wird also das Datum an Adresse 1000 als weitere Adresse (Pointer) interpretiert, von der das Datum für `R1` geholt wird. Damit ist es möglich durch einen einzigen Befehl (Änderung des Pointers in 1000) auf verschiedene Datensätze zuzugreifen. Dieser spezielle Modus wird *Memory Deferred* bezeichnet, da der Pointer im Speicher abgelegt ist. Alternativ kann der Pointer in einem Register abgelegt sein (*Register Deferred*).

```
MOVE R1, (R2)     ; hole Inhalt der Adresse, die in R2 steht
```

Da man Register schneller und einfacher als Speicherstellen verändern kann, wird dieser Modus sehr häufig verwendet. Eine Spezialform dieser indirekten Adressierung sind *Autoincrement* und *Autodecrement*.

```
MOVE R1, (R2)+    ; hole Inhalt der Adresse in R2 und inkrementiere R2
MOVE R1, -(R2)    ; dekrementiere R2 und hole Inhalt der Adresse in R2
```

Mit diesen Anweisungen lassen sich sehr elegant Datenzugriffe in Schleifen programmieren. Eine sehr typische Anwendung dieser Anweisungen wäre auch die Verwendung von `R2` als Stackpointer (warum?). Erweiterungen der indirekten Adressierung über Register sind *Displacement*

```
MOVE R1, 100(R2) ; hole Inhalt der Adresse R2 + 100
```

und *Indexed*.

```
MOVE R1, (R2+R3) ; hole Inhalt der Adresse R2 + R3
```

Beide Varianten erlauben effiziente Programmierung in zweierlei Hinsicht. Die Befehlslänge (Anzahl der Bits eines Befehls) kann klein gehalten werden, da einige wenige Register sehr effizient zu adressieren sind. Ausserdem erlauben diese Modi die Generierung von sehr kompaktem Programmcode (Qualität des Programmierers/Compilers vorausgesetzt).

Immediate Adressierung

Es gibt auch Daten, die weder im Speicher, noch in einem Register stehen müssen, sondern direkt (*immediate*) im Programmcode angegeben sind.

```
MOVE R1, #100     ; lade R1 mit der Zahl 100
```

Diese Zahl muss dann natürlich explizit im Befehl angegeben werden. Der Platzbedarf hängt dann von der maximalen Grösse der Zahl ab, die vom Architekten des Befehlssatzes zugelassen wird. Stehen Wort–Register zur Verfügung (32 Bit), dann würde das Immediate Datum allein 32 Bit beanspruchen. Allerdings stellt sich die Frage, ob man derart grosse Zahlen jemals direkt eingeben wird.

In einem Befehlssatz einer konkreten Maschine sind selten alle möglichen Adressmodi unterstützt. Man muss daher Grundlagen für die Entscheidung über die Wahl einzelner Parameter erarbeiten. Dies geschieht, indem man *Benchmark*–Programme unter genau definierten Bedingungen auf Maschinen laufen lässt, die einen möglichst grossen Befehlssatz haben. Daraus lassen sich dann Statistiken erstellen, die Anhaltspunkte für die Architektur des Befehlssatzes geben.

Benchmarkmessungen

Für die Architektur unserer DLX–Maschine werden wir Displacement und Immediate als Adressmodi vorsehen, da diese $75\% - 99\%$ aller verwendeten Adressierungsmodi ausmachen. Die Grösse des Displacement wird auf 16 Bit gesetzt werden, da dies in 99% der gemessenen Fälle ausreicht. Auch die Grösse der Immediate–Adressierung wird auf 16 Bit gesetzt, was 80% aller Fälle abdeckt (Hennessy and Patterson, 1996).

4.4.3 Operationen des Befehlssatzes

Die Operationen eines Befehlssatzes können in Gruppen eingeteilt werden. Die wichtigsten davon, die auf jedem heutigen Prozessor implementiert sind, werden wir hier kurz betrachten.

Operationen für den *Datentransfer* sind elementare Voraussetzung, um Daten aus dem und in den Speicher zu bewegen. Operationen aus dieser Gruppe haben wir schon in unserem ersten kleinen Assemblerprogramm kennengelernt.

Operationen für *Arithmetik* und *Logik* umfassen die bekannten Integeroperationen wie Addition (siehe `ADD`), Multiplikation, oder logische Verknüpfungen der Operanden.

Floating Point–Operationen werden in einem speziellen Rechenwerk der CPU ausgeführt. Die Operanden für Addition, Subtraktion, Multiplikation, und Division sind auf den meisten heutigen Maschinen im IEEE–Format codiert. Manchmal gibt es Hardwareimplementierungen für das Berechnen von Quadratwurzeln, oder trigonometrischen Funktionen.

Control–Operationen sind jene Befehle, die den Programmablauf steuern. Man unterscheidet dabei zwischen

- Jump (unbedingter Sprung)
- Branch (bedingter Sprung)
- Call (Prozeduraufruf)
- Return (Rücksprung aus Prozedur)

Alle diese Befehle aus der Control–Gruppe veranlassen, dass das Programm *nicht* mit dem nächsten Befehl, sondern an einer anderen Stelle fortgeführt wird. Anders augedrückt verändern diese Befehle den Program Counter (PC), also jenes Register in der Control Unit, das die Adresse des nächsten auszuführenden Befehls beinhaltet.

Speziell bei Branches kann man beobachten, dass diese sehr oft nur einige Instruktionen überspringen (kleine Schleifen). Daher genügt es zur Adressierung der Sprünge meist die Distanz relativ zum PC anzugeben. In vielen Fällen genügen dafür 8 Bits. Mit diesen lassen sich Sprünge von ± 127 relativ zum PC realisieren.

Im Unterschied zu Jumps und Branches, sind die Sprungadressen von Call und Return nicht vor Programmausführung bekannt, d. h. sie müssen *dynamisch* bestimmt werden.

Beispiel: Wir betrachten drei Prozeduren A, B und C eines Programms. Die Prozedur C kann sowohl von A, als auch von B aufgerufen werden. Am Prozedurende von C steht die Anweisung `RETURN`. Ein Compiler (Assembler) kann hier nicht entscheiden, an welche Adresse zu springen ist, da dies davon abhängt, ob A oder B die Prozedur C aufgerufen hat. Die Rücksprungadresse kann daher erst zur Laufzeit des Programms gebildet werden. $\Diamond$

Die Rücksprungadresse wird je nach Maschine meist in einem speziellen Register oder auch auf einem Stack abgelegt, dessen Struktur sich hier als sehr vorteilhaft erweist. Die Rücksprungadresse ist jene Adresse, an der die Befahlsarbeitung nach Prozedurende fortgesetzt wird.

Branches sind Sprünge, die von einer Bedingung abhängig sind. Die Bedingung wird im Programm definiert, und muss dann zur Laufzeit interpretiert werden. Allerdings ist der Sprung trotzdem vor Programmablauf festlegbar, denn es gibt nur zwei Möglichkeiten: die Bedingung ist erfüllt (Sprung), oder sie ist nicht erfüllt (kein Sprung). Es gibt nun verschiedene Möglichkeiten, eine Bedingung zu prüfen, wobei meist nur eine dieser Möglichkeiten in einem Prozessor realisiert ist.

- Condition Code Register (in einem speziellen Register werden Statusflags gesetzt, die nach jedem Befehl aktualisiert werden)
- Register (ein beliebiges Register wird für Vergleiche benutzt, Condition Code muss vom Programm(ierer) erzeugt werden)
- Compare and Branch (ein einziger Befehl prüft die Bedingung *und* verzweigt)

Ein Nachteil des Condition Code Register ist, dass der Condition Code (wie Zero Flag, Overflow, etc.) bei jeder Operation gebildet wird, auch wenn kein Branch Befehl eine Bedingung prüft. Ein Compare and Branch ist relativ zeitaufwendig (im Vergleich zu anderen Befehlen, Probleme beim Pipelining). Die häufigste Bedingung für Branches ist das Prüfen auf Gleichheit/Ungleichheit zweier Zahlen (in Registern oder Speicherstellen).

Programmtechnisch ist bei Prozeduraufrufen zu beachten, dass Register oft als Zwischenspeicher verwendet werden. Wird z. B. `R1` für eine Variable x verwendet, dann darf eine aufgerufene Prozedur dieses Register nicht verändern. Eine Möglichkeit wäre dann, dieses Register gar nicht zu verwenden, was allerdings oft eine erhebliche Einschränkung darstellt. Daher ist entweder die aufrufende Prozedur (*Caller Save*) oder die aufgerufene Prozedur (*Callee Save*) dafür verantwortlich, dass dieses Register temporär gespeichert wird und nach dem Prozeduraufruf restauriert wird.

Wieviele und welche Befehle sollte nun ein Befehlssatz haben? Bechmarkmessungen für einen *Intel 80x86*–Prozessor ergaben Befehlshäufigkeiten, die in Tabelle 4.1 angeführt sind.

Befehl	Häufigkeit
load	0.22
branch	0.20
compare	0.16
store	0.12
add	0.08
and	0.06
sub	0.05
move	0.04
call	0.01
return	0.01
total	0.96

Tabelle 4.1: Befehlshäufigkeiten für Intel 80x86 bei Integer Benchmarks.

Wir sehen, dass 96% aller Instruktionen von 10 Befehlen abgedeckt werden! Daraus ergibt sich auch ein zentraler Leitsatz für die Architektur des Befehlssatzes: *Make the common case fast.* Die Befehle, die oft vorkommen, sollten also schnellstmöglich ausgeführt werden (Hardware). Überdies erkennt man, dass ein erstaunlich kleiner Befehlssatz genügt (überlegen Sie Gründe, warum in obiger Tabelle kein Multiplikationsbefehl vorkommt) .

Für die DLX ziehen wir also den Schluss, dass wir wenige elementare Befehle zur Verfügung stellen werden. Branches sollten zumindest mit 8 Bit relativ zum PC ermöglicht werden. Neben dieser Sprungadressierung muss diese Architektur aber auch indirekte Adressierung über Register für Rücksprungadressen vorsehen.

4.4.4 Operandentypen

Die Grösse der einzelnen Operanden von Instruktionen hängt elementar von der Rechnerarchitektur insbesondere von der Datenbusbreite ab. Operanden, deren Grösse der Breite des Datenbusses entsprechen können am einfachsten transferiert werden (in einem Taktzyklus). Beim Transfer Operanden unterschiedlicher Grösse erhöht sich der Hardwareaufwand. Es ist unmittelbar einsichtig, dass ein 32–Bit Datenbus mindestens zwei Taktzyklen braucht, um ein Doppelwort zu übertragen.

Der Typ des Operanden wird ganz einfach durch die Instruktion festgelegt. So gibt es eigene *Opcodes* (der Teil eines Befehls, der den Befehl selbst codiert) für Ganzzahl–, und Floating Point–Multiplikation. Dadurch wird sofort klar, welchen Typ die jeweiligen Operanden haben (müssen). Integers werden heute meist als Wort und Doppelwort repräsentiert. Für negative Zahlen wird fast ausschliesslich das 2–Komplement verwendet. Ähnliches Konvergenzverhalten kann man bei der Floating Point–Repräsentation beobachten, bei der durchwegs die beiden IEEE–Formate (32 und 64 Bit) verwendet werden. Manche Maschinen haben noch spezielle Befehle für String/Characteroperationen oder BCD–Zahlen.

Daraus folgt für die DLX–Maschine eine klare, einfache 32–Bit Architektur, die beide IEEE–Formate unterstützen wird.

4.4.5 Befehlssatzcodierung

Ein codierter Befehl kann in drei wesentliche Teile (*Felder*) unterschieden werden. Das eigentliche Befehlsfeld (*Opcode*) gibt den speziellen Befehl an (z. B. `ADD`), das Adressfeld (*Adress Field*) enthält eine Adresse, die in Abhängigkeit eines eventuellen *Address Specifier* interpretiert wird. Der Address Specifier kann entfallen, wenn die Spezifizierung der Adresse im Opcode erfolgt (z.

B. `LW` für "load word" enthält schon die Information darüber, dass ein Wort adressiert wird).

Die wesentliche Frage bei der Codierung des Befehlssatzes ist die nach der Länge der einzelnen Codewörter, die uns schon einmal in Kapitel 1.2 beschäftigt hat. Dort haben wir Codes fester und variabler Länge unterschieden, dessen Bedeutung für einen codierten Befehl in Abb. 4.4 gezeigt ist.

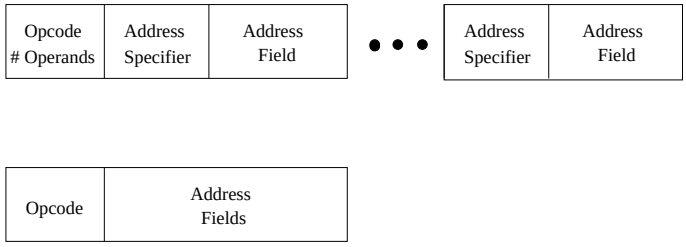

Abbildung 4.4: Befehlscode variabler (oben) und fester Länge (unten).

Wenn wir den Befehlssatz hinsichtlich Programmgrösse optimieren wollen, müssen wir einen Befehlscode variabler Länge vorsehen, da dann für jeden Befehl nur die jeweils nötige Zahl von Bytes gebraucht werden. Diese Variante stellt allerdings höhere Ansprüche an die Control Unit der CPU, da diese bei jedem Befehl aus dem Opcode die Anzahl der Operanden bestimmen muss und die entsprechende Anzahl von Bytes aus dem Speicher holen muss.

Die Programmsteuerung ist bei Befehlscodes fester Länge naturgemäss einfacher, da jedes Befehlswort gleiche Länge aufweist, was wiederum den Programmcode vergrössern wird, da es immer Befehle geben wird, die eigentlich nicht die vorgesehene Länge brauchen würden.

Für die DLX werden wir Befehlscodes fester Länge verwenden und dabei möglichst alle Bits des Codeworts verwenden.

4.4.6 Befehlssatz und Compiler

Da fast alle heutigen Programme in einer Hochsprache geschrieben werden, muss der Befehlssatz einer Maschine nicht nur auf die jeweilige Hardwarearchitektur, sondern auch auf grundlegende Compilertechnologien abgestimmt werden. Heutige Compiler versuchen Programmcode in vielerlei Hinsicht zu optimieren. Eine der wichtigsten Optimierungen betrifft die Verwendung von Registern für Variablen einer Hochsprache. Insbesondere innerhalb von Prozeduren wird versucht, lokale Variable in Registern zu halten, da dies die Ausführungsgeschwindigkeit erhöht. Die Entscheidung welche Variablen in Register kommen, um die Performance zu optimieren ist ein sehr komplexes

Problem. Generell gilt natürlich, dass bei steigender Zahl der Register immer mehr Variablen dort gehalten werden können, allerdings ist auch auf heutigen Maschinen die Anzahl der Register beschränkt. Eine häufig verwendete Methode in Compilern zur Registerallokierung ist das *Graph Coloring*. Diese Methode ist aber nur robust, wenn dem Compiler zumindest 16 Register zur Verfügung stehen.

Auch für den Compilerbau gilt ein ähnliches Prinzip wie in der Maschinenarchitektur: *Make the frequent cases fast and the rare case correct.* Die häufig auftretenden Codesequenzen sollten also auf Geschwindigkeit optimiert werden, während die seltenen "nur" korrekt sein sollten. Daraus ergeben sich einige prinzipielle Anforderungen an die Architektur des Befehlssatzes.

1) Orthogonalität des Befehlscodes (Operationen, Datentypen und Adressierungsmodi sollten beliebig kombinierbar sein)

2) Einfache Grundoperationen (manche Architekturen haben sehr spezielle Befehle, die aber selten von Compilern verwendet werden)

3) Einfache Abhängigkeiten (heute genügt es nicht mehr die Codegrösse zu optimieren, pipelines, branch prediction etc. beeinflussen den Code)

4) Statische Instruktionen (Befehle sollten keine Laufzeitabhängigkeiten haben, um sie zur Compilezeit exakt definieren zu können)

Aus obigen Punkten ergibt sich eindeutig die Forderung nach einem einfachen, klar strukturierten Befehlssatz. Interessanterweise setzte sich diese Erkenntnis aber erst ab 1980 durch. Vorher versuchte man nämlich durch sehr komplexe Befehlssätze (z. B. *VAX*), Maschinensprache an Hochsprache anzunähern. Heute ist aber die Ansicht weit verbreitet, dass eine RISC (*Reduced Instruction Set Computer*)–Architektur beträchtliche Geschwindigkeitsvorteile bringt. Diese gehen zwar auf Kosten der Codegrösse eines Programms, doch zeigt die heutige Entwicklung auf dem Speichersektor, dass dieses Problem an Bedeutung verliert.

Der DLX–Maschine wird daher eine RISC–Architektur zugrundegelegt werden.

4.5 Architekturperformance

Wir haben in diesem Kapitel oft von Verbesserungen von Komponenten der Rechnerarchitektur gesprochen, nun wollen wir versuchen, diese Verbesserungen zu quantifizieren und in Masszahlen zu fassen.

Ganz global betrachtet führt ein Computer(system) ein Programm in einer messbaren Zeit t_P aus. Führt man nun eine oder mehrere Verbesserungen, (z. B. schnellerer Speicher, schnellere Befehle, höhere Taktfrequenz, etc.) aus, so wird t_P (hoffentlich) verringert. Bezeichnen wir die Ausführungszeit auf der ursprünglichen Maschine mit t_P, old und jene auf der verbesserten Maschine mit t_P, new so ergibt sich der *Speedup* s zu

$$s = \frac{t_{P,old}}{t_{P,new}}. \tag{4.1}$$

Kennen wir den Anteil f_e am Gesamtprogramm von den Befehlen, auf die sich die Verbesserung auswirkt, und den Speedup s_e dieser speziellen Verbesserung, dann kommen wir über *Amdahl's Law* zum Speedup

$$s = \frac{1}{(1 - f_e) + \frac{f_e}{s_e}}. \tag{4.2}$$

Beispiel: Wir betrachten eine Verbesserung einer Maschine, die 40% aller Befehle betrifft. Diese Verbesserung erhöht die Geschwindigkeit dieser Befehle um einen Faktor 10. Wie gross ist der Speedup der Maschine? Wir setzen ein in Gleichung 4.2 und erhalten

$$s = \frac{1}{0.6 + \frac{0.4}{10}} \simeq 1.56.$$

$\Diamond$

Eine weitere wichtige Masszahl zur Beurteilung der Performance einer Rechnerarchitektur sind die *Clocks per Instruction*

$$CPI = \frac{n_C}{n_I}, \tag{4.3}$$

wobei n_C die Anzahl der Taktzyklen ist, die ein Programm zur Ausführung braucht, und n_I die Anzahl der Instruktionen des Programms ist. über die CPI-Rate kommt man einfach zur Ausführungszeit

$$t_P = CPI \times n_I \times t_C, \tag{4.4}$$

mit t_C als der Periodendauer eines Taktzyklus. Die CPI–Rate ist unabhängig von der Taktfrequenz und der Anzahl der Instruktionen eines Programms, was wesentlich zu einer Vergleichbarkeit der CPI–Raten beiträgt. Allerdings ist auch diese Masszahl von mehreren Parametern abhängig. Generell kann man folgende Abhängigkeiten identifizieren:

- Taktzykluszeit – Technologie der Hardware

- CPI – Organisation und Befehlssatzarchitektur
- n_C – Befehlssatzarchitektur und Compilertechnologie

Oft ist es auch nützlich, die CPI–Raten einzelner Befehle oder Befehlsgruppen zu kennen. Die gesamte CPI–Rate ergibt sich dann zu

$$CPI = \sum_{i=1}^{n} CPI_i \; f_i, \tag{4.5}$$

mit CPI_i als CPI–Rate einer Teilmenge des Befehlssatzes und f_i der Anteil dieser Teilmenge an der Anzahl der Gesamtinstruktionen. Die Summe ist dann über alle n Teilmengen zu bilden, d. h., es muss gelten $\sum_{i=1}^{n} f_i = 1$.

Eine andere oft gebrauchte Masszahl sind die *Million Instructions Per Second*

$$MIPS = \frac{n_I}{t_P \times 10^6} = \frac{1}{CPI \times t_C \times 10^6}. \tag{4.6}$$

Die MIPS–Rate ist jedoch eine sehr ungenügende Masszahl, da sie sogar bei derselben Rechnerarchitektur und demselben Programm unterschiedliche (compilerabhängige) Ergebnisse bringen kann. Es ist sogar möglich, dass die MIPS–Rate indirekt proportional zur Perfomance eines Systems ist. Ähnlich ist die Situation bei den *Million Floating Point Operations Per Second* (MFLOPS), bei denen nicht die Gesamtzahl der Instruktionen n_I, sondern nur die Anzahl der Floating Point–Operationen betrachtet wird.

Kapitel 5

Die DLX–Maschine

Mit den Überlegungen des vorigen Kapitels wollen wir nun einen konkreten Prozessor konstruieren. Die wesentlichen Designkriterien umfassen:

- Einfache *Load–Store*–Architektur
- Befehlssatz für efffiziente Pipeline (Befehlscodes fester Länge)
- Befehlssatz für effiziente Compiler

Die sich daraus ergebende Maschine ist zwar eine hypothetische (nicht kommerzielle erhältliche), doch stellt sie einen repräsentativen Durchschnitt vieler realer Prozessoren dar. Versucht man dies (in nicht ganz ernster Weise) zu formalisieren, so ergibt sich die Gleichung (Hennessy and Patterson, 1996) (AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation–1, Sun–4/110, Sun–4/260) / 13 = 560 = DLX.

Es werden also Elemente aller dieser Maschinen verwendet, um zur DLX (die Zahl 560 in römischen Ziffern) zu gelangen.

5.1 Der DLX–Befehlssatz

Wir wollen nun die Details der DLX–Maschine im einzelnen präsentieren.

5.1.1 Register

Für die DLX werden 32 32–Bit Register (*General Purpose Register*, GPR) vorgesehen, die mit `R0...R31` bezeichnet werden. Zusätzlich existieren 32 32–Bit Floating Point Register (FPR), die auch im 64–Bit Modus angesprochen werden können (*Double Precision*). Die FPRs werden mit `F0...F31`

bezeichnet. Falls sie als Double verwendet werden, so können alle geraden Registernummer verwendet werden, wobei die ungeraden dann das LSWord der Zahl im Double–Format beinhalten. Eine spezielle Rolle übernimmt das Register `R0`, das *immer* den Wert 0 hat. Ein Laden dieses speziellen Registers bleibt ohne Wirkung. Mit dieser kleinen Besonderheit lassen sich einige wichtige Befehle sehr einfach auf andere zurückführen, was eine Einsparung von Befehlen bedeutet.

5.1.2 Datentypen

Es können alle gängigen Datentypen – Byte, Halbwort, Wort, Doppelwort – adressiert werden. Die Operationen der DLX arbeiten aber ausschliesslich mit Operanden mit 32–Bit (Integer oder Float) und 64–Bit (Double). Werden Bytes oder Halbworte geladen, so werden die unberührten Stellen des Registers abhängig vom speziellen Befehl entweder mit 0 aufgefüllt (*unsigned*), oder vorzeichenrichtig erweitert (*signed*).

5.1.3 Adressierungsmodi

Es werden nur die beiden Adressierungen Displacement und Immediate mit jeweils 16–Bit im Adressfeld vorgesehen. Zwei weitere Modi lassen sich daraus einfach ableiten. Eine indirekte Adressierung über ein Register entspricht einfach einem Displacement von 0. Eine absolute Adressierung (allerdings nur mit 16 Bit) entspricht einem Displacement relativ zum Register `R0`. Der DLX–Speicher ist 32–Bit byteadressierbar, ausgerichtet (aligned) und verwendet das Big Endian Format.

5.1.4 Befehlsformat

Alle Befehle werden mit einer festen Länge von 32–Bit codiert. Der Opcode ist immer 6 Bit lang. Da es nur zwei Adressierungsmodi gibt, können diese im Opcode codiert werden. Das Format der Adressfelder kann abhängig vom jeweiligen Befehl in drei Gruppen unterteilt werden: I–, R–, und J–Instruktionen. Das Befehlsformat einer I–Instruktion ist in Abb. 5.1 dargestellt.

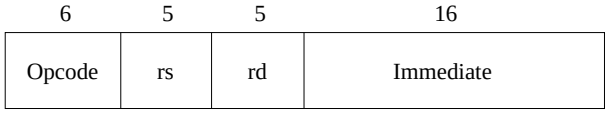

Abbildung 5.1: Instruktionstyp I der DLX.

Das Adressfeld `rs` bezeichnet das Quellregister (*Source*) und das Adressfeld `rd` das Zielregister (*Destination*). Befehle aus dieser Gruppe umfassen: Loads und Stores (Immediate ist dann das Displacement), alle Immediate–Befehle (`rd` ← `rs op immediate`), Branches (`rd` bleibt unbenützt, Immediate ist PC–relative Sprungadresse), und Jumps (mit `rd = 0` und `immediate = 0`).

Abb. 5.2 zeigt das Befehlsformat einer R–Instruktion.

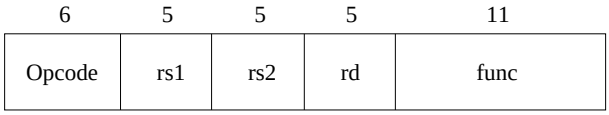

Abbildung 5.2: Instruktionstyp R der DLX.

Dieser Befehlstyp beinhaltet alle ALU Operationen, wobei die Operanden in den beiden Quellregistern `rs1` und `rs2` stehen, und das Resultat in `rd` abgelegt wird. In `func` wird die Operation spezifiziert, daher ist dieses Feld eigentlich Teil des Opcodes.

Das Format der J–Instruktionen zeigt Abb. 5.3.

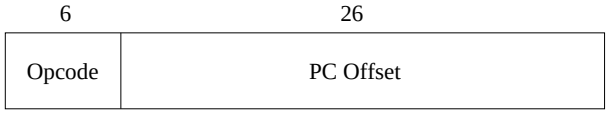

Abbildung 5.3: Instruktionstyp J der DLX.

Dieser Befehlstyp wird ausschliesslich für Jumps und Traps verwendet, wobei der Offset im Adressfeld zum PC addiert wird, um zur Sprungadresse zu gelangen. Bei einem *Jump and Link* wird zusätzlich die Rücksprungadresse in das Register *R31* geschrieben.

5.1.5 Befehlsfunktionen

Wir betrachten nun die einzelnen DLX–Befehle und geben einige Beispiele zu ihrer Verwendung an.

Datentransfer

`LB, LBU, SB`...lädt und speichert Bytes signed oder unsigned

```
LBU R1, 100(R2)     ; hole Byte von R2 + 100 in R1
```

`LH, LHU, SH`...lädt und speichert Halbwörter signed oder unsigned

```
  LH R3, 100(R1)     ; hole Halbwort von R1 + 100 nach R3
```

`LW, SW`... lädt und speichert Wörter

```
  SW R1, 100(R2)     ; speichere Wort aus R1 nach R2 + 100
```

`LF, LD, SF, SD`... lädt und speichert Floats und Doubles

```
  LD F2, 100(R2)     ; hole Double von R2 + 100 nach F2 und F3
```

`MOVI2S, MOVS2I`... transferiert Daten zwischen GPRs und Spezialregister
`MOVF, MOVD`... transferiert Daten zwischen FPRs

```
  MOVF F3, F7 ; kopiere F7 nach F3
```

`MOVFP2I, MOVI2FP`... transferiert Daten zwischen FPRs und GPRs

```
  MOVFP2I R1, F2     ; kopiere F2 nach R1
```

Arithmetik und Logik

`ADD, ADDI, ADDU, ADDUI`... Addiert Register signed, unsigned, immediate

```
  ADD R3, R2, R1     ; R1 + R2 nach R3 (signed)
```

`SUB, SUBI, SUBU, SUBUI`... Subtrahiert signed, unsigned, immediate

```
  SUBI R3, R2, #42     ; R2 - 42 nach R3 (signed)
```

`MULT, MULTU, DIV, DIVU`... multipliziert/dividiert FPRs signed, unsigned

```
  MULT F3, F2, F1     ; F1 * F2 nach F3 (signed)
```

`AND, ANDI`... Logisches UND (auch mit immediate)

```
  ANDI R3, R2, #16     ; Bit 4 von R2 nach R3
```

`OR, ORI, XOR, XORI`... Logisches ODER, XOR (immediate)
`LHI`... Lade oberes Halbwort immediate

```
  LHI R3, #12564     ; zur Generierung von 32-Bit Adressen
```

`SLL, SRL, SLLI, SRLI, SRA, SRAI`... logisches, arithmetisches Schieben

```
  SLLI R3, #3     ; verschiebe R3 um 3 Stellen nach links, 0 von rechts
  SRA R3, R2     ; verschiebe R3 um R2 nach rechts, Vorzeichen von links
```

`S__, S__I`... Prüfen einer Bedingung LT, GT, LE, GE, EQ, NE

```
  SEQ R3, R2, R1     ; setze R3, wenn R1 = R2
  SLTI R3, R2, #10     ; setze R3, wenn R2 <= 10
```

Steuerung (Control)

`BEQZ, BNEQZ`... Branch, wenn GPR gleich/ungleich 0

```
  BNEQZ R3, loop     ; springe nach Marke "loop", wenn R3 ungleich 0
```

`BFPT, BFPF`... Branch zufolge Compare Bit in FP Status Register

```
  BFPT loop     ; springe nach Marke "loop", wenn FPSR true
```

`J, JR`... Sprung absolut (26–Bit) oder Register

```
  JR R1     ; springe nach Adresse in R1
```

`JAL, JALR`... Sprung, speichere PC+4 in R31 (link)

```
  JAL proc     ; Rufe Prozedur "proc", R"ucksprungadresse in R31
```

`TRAP`... Springe auf systemabhängige Stelle (Betriebssystem)
`RFE`... Rücksprung aus Exception (Trap)

Floating Point

`ADDF, ADDD`... addiere Floats, Doubles

```
  ADDD F8, F4, F6     ; addiere Doubles in F4 und F6 nach F8
```

`SUBF, SUBD`... subtrahiere Floats, Doubles
`MULTF, MULTD`... multipliziere Floats, Doubles
`DIVF, DIVD`... dividiere Floats, Doubles
`CVTx2y`... konvertiere Integer, Float, Double (x != y)

```
  CVTF2I F2, F3     ; wandle Float in F3 zu Integer in F2
```

`_F, _D`... Vergleiche Floats, Doubles LT, GT, LE, GE, EQ, NE

```
  GED F4, F6     ; F4 >= F6? wenn ja, setze Compare Bit in FPSR
```

Besonderheiten

Die Multiplikation und Division von Zahlen wird ausnahmslos von der Floating Point Unit durchgeführt. Die Zahlen müssen daher in FPRs stehen.

Die Rücksprungadresse eines Prozeduraufrufs wird mit dem Befehl `JAL` *immer* in `R31` abgelegt.

Zwei wichtige Operationen sind nicht explizit im Befehlssatz der DLX: Laden einer Konstanten in ein Register und das Verschieben eines Datums von einem GRP in ein anderes (*Register Move*). Beide Befehle können mit dem Nullregister `R0` synthetisiert werden.

```
ADDI R1, R0, #5    ; lade R1 mit 5
ADD R1, R0, R2     ; kopiere R2 nach R1
```

Selbst auf einer RISC–Architektur werden abhängig vom verwendeten Compiler meist nur eine kleine Untermenge von Befehlen verwendet. Benchmarkmessungen von Integer–Programmen ergaben für die DLX–Maschine, dass 80% aller Befehle von nur fünf Befehlen abgedeckt wird: Load (26%), Branch (17%), Add (14%), Compare (14 %), Store (9%) (Hennessy and Patterson, 1996).

Kapitel 6

Pipelining

Nun haben wir am Beispiel der DLX–Maschine einen konkreten Befehlssatz eines Prozessors kennengelernt. Es ist klar, dass für die einzelnen Befehlsgruppen logische Einheiten am Prozessorchip vorgesehen werden müssen, die diese Befehle dann tatsächlich ausführen. Je schneller diese Einheiten arbeiten (je weniger Taktzyklen sie benötigen), desto schneller werden Programme abgearbeitet. Spätestens dann, wenn man es technologisch geschafft hat, alle (oder viele) Befehle auf einen Taktzyklus zu beschränken, kann man eigentlich nur noch die Taktfrequenz erhöhen, um den Prozessor zu beschleunigen. Techniker (vor allem gute) geben sich aber nicht mit solchen Erkenntnissen zufrieden, und man begann nachzudenken, ob man nicht mehrere Befehle quasi–gleichzeitig im Prozessor bearbeiten könne, was zur *Pipeline*–Technik führte.

Prinzipiell sind z. B. eine Addition von zwei Registern und das Laden einer Speicherstelle in ein Register zwei Befehle, die nicht voneinander abhängen. Während die ALU addiert, könnte die Control Unit den Speicherzugriff machen (überlegen Sie Gründe, warum das nicht ganz so einfach ist). Man könnte also diese beiden Befehle gleichzeitig ausführen, was eine Verbesserung der Rechengeschwindigkeit bedeuten würde. Zum Vergleich wird ein Produkt in einer modernen Anlage auch in mehreren Arbeitsschritten produziert, die gleichzeitig stattfinden. Würde man ein Auto so bauen, dass das nächste Auto erst dann zu bauen begonnen wird, wenn das vorige fertig ist, würden Autos Mangelware sein.

Wenn wir nun einen Maschinenbefehl mit einem Produkt vergleichen, so könnten wir für den Befehl auch verschiedene Arbeitsschritte definieren, in denen er bearbeitet wird. Wir können dann an mehreren Befehlen arbeiten, allerdings ist zu beachten, dass ein Arbeitsschritt immer nur an *einem* Befehl arbeiten kann. Trotzdem können wir unter der Annahme, dass alle Arbeitsschritte voneinander unabhängig sind (was in der Praxis selten eintritt) sehr

einfach den idealen Speedup s_P durch Pipelining angeben. Für n_P Arbeitsschritte (Pipelinestufen) ergibt sich

$$s_P = n_P. \tag{6.1}$$

Wenn es gelingt, jeden Arbeitsschritt in einem Taktzyklus zu absolvieren, dann würde man die optimale CPI–Rate von 1.0 erzielen. In der Realität wird aber der Speedup kleiner als der ideale sein, da nicht alle Arbeitsschritte für alle möglichen Befehle voneinander unabhängig sein werden. Wird z. B. ein gelbes nach einem orangen Auto lackiert, dann werden die zwei Lackiervorgänge länger dauern (durch Umrüstzeiten der Maschinen), als das Lackieren von zwei gelben Autos hintereinander. Die Arbeitsschritte können also auch zeitliche Abhängigkeiten haben.

Um diese Ideen nun auf einen Prozessor zu übertragen, müssen wir uns überlegen in welche Arbeitsschritte wir die Abarbeitung von Befehlen unterteilen können.

6.1 Einfache DLX–Implementierung

Wir haben schon bei der Behandlung der von–Neumann Architektur (4.2) einige wesentliche Stufen der Befehlsbearbeitung identifiziert: Holen eines Befehls, Decodieren, Ausführen, Ergebnis abspeichern. Wir wollen diese Sequenz noch für eine vereinfachte Version der DLX präzisieren. Wir beschränken uns auf Load/Store–Befehle, die nur mit Wörtern operieren, auf Branches und Integer ALU Operationen. Eine mögliche Folge der Abarbeitung besteht dann aus fünf Arbeitsschritten (zur Darstellung verwenden wir einen *Pseudocode* (Hennessy and Patterson, 1996)):

1) Instruction Fetch (IF)

```
IR <- Mem[PC]
NPC <- PC + 4
```

   Die Instruktion wird aus der Speicherstelle in das *Instruction Register* (IR) geholt, auf die der Program Counter (PC) zeigt. Dann wird der *New Program Counter* (NPC) auf die Adresse des nächsten Befehls gesetzt. PC und NPC sind spezielle Register auf der DLX. Hier zeigt sich schon der Vorteil von Befehlscodes fester Länge, die die Berechnung von NPC stark vereinfacht.

2) Instruction Decode / Register Fetch (ID)

```
A <- Regs[IR[6..10]]
B <- Regs[IR[11..15]]
Imm <- +/-IR[16..31]
```

Hier werden wieder drei Wortregister benötigt, um Operanden des Befehls zwischenzuspeichern. Aus dem IR werden die Adressen der beiden Quellregister und ein (möglicher) Immediate–Wert extrahiert. Dies wäre nicht für alle Befehle nötig, doch vereinfacht dies den Register Fetch (und kostet auch keine zusätzliche Zeit). Die Registerinhalte werden in Zwischenregistern `A, B` gespeichert, der Immediate–Wert vorzeichenerweitert in `Imm`. Der Instruction Decode kann parallel dazu erfolgen, da dieser nur die ersten sechs Bits des Befehlsworts benötigt. Auch hier ist das einfach aufgebaute und fix strukturierte Befehlswort von grossem Vorteil.

3) Execution / Effective Address (EX)

Nun wird der Befehl auch wirklich ausgeführt. Abhängig von der jeweiligen Instruktion (die im vorigen Schritt decodiert wurde), müssen wir vier Fälle unterscheiden.

- Speicherzugriff

  ```
  ALUOutput <- A + Imm
  ```

  Die absolute Adresse wird berechnet und im internen Register `ALUOutput` abgelegt.

- ALU Register–Register

  ```
  ALUOutput <- A func B
  ```

  Die ALU führt die Funktion aus, die durch den Funktionscode im Befehlswort definiert ist, und speichert das Ergebnis in `ALUOutput`.

- ALU Register–Immediate

  ```
  ALUOutput <- A op Imm
  ```

  Die ALU führt die Operation aus, die im Operationscode angegeben ist.

- Branch

  ```
  ALUOutput <- NPC + Imm
  cond <- (A op 0)
  ```

Die (mögliche) Sprungadresse wird berechnet. Ob tatsächlich gesprungen wird, wird in `cond` berechnet, das als Ergebnis den Vergleich des Registers mit 0 enthält. Die Operation `op` prüft (im Fall der DLX) nur auf (Un)Gleichheit.

4) Memory Access / Branch Completion (MEM)

   Der PC wird auf seinen neuen Wert gesetzt `PC <- NPC`, dann sind zwei Fälle zu unterscheiden:

   – Speicherzugriff

   ```
   LMD <- MEM[ALUOutput] (if load)
   Mem[ALUOutput] <- B (if store)
   ```

   Der Speicher wird entweder mit dem Quellregister, das in `B` abgelegt wurde, beschrieben, oder gelesen, wobei das Datum in das interne Register *Load Memory Data* (LMD) gespeichert wird.

   – Branch

   ```
   if (cond) PC <- ALUOutput
   ```

   Wenn die Bedingung für Branch erfüllt ist, dann wird der PC mit der vorhin berechneten Sprungadresse geladen.

5) Write Back (WB)

   – Speicherzugriff

   ```
   Regs[IR[11..15]] <- LMD
   ```

   Das Datum aus dem Speicher wird in das Zielregister transferiert.

   – ALU Register–Register

   ```
   Regs[IR[16..20] <- ALUOutput
   ```

   – ALU Register–Immediate

   ```
   Regs[IR[11..15]] <- ALUOutput
   ```

   In beiden Fällen wird das Ergebnis der ALU–Berechnung in das gewünschte Register engetragen (überlegen Sie, wie man die WB–Stufe vereinfachen könnte).

Die fünf Stufen IF, ID, EX, MEM und WB bilden nun die einzelnen Stufen der Befehlsabarbeitung. Allerdings kann diese Implementierung noch nicht für eine Pipeline verwendet werden. Würde man dies tun und zum Beginn der ID–Stufe schon der nächste Befehl geholt werden (IF), dann käme es z. B. zum Problem, dass in der WB–Phase ein Register über das IR adressiert würde, das aber nicht mehr jene Instruktion enthält, die gerade in WB ausgeführt wird!

Wenn es gelingt diese fünf Bearbeitungsschritte in je einem Taktzyklus durchzuführen, dann dauert ein DLX–Befehl maximal fünf Taktzyklen. Store,– und Branch–Befehle würden dann nur vier Taktzyklen benötigen, da keinerlei Arbeit in der WB–Phase für diese Instruktionen anfallen. Ausserdem könnte für ALU–Befehle die WB–Phase in die MEM–Phase vorgezogen werden, da diese Befehle in der MEM–Phase "arbeitslos" sind, was auch für ALU–Befehle eine Ausführungszeit von vier Taktzyklen bedeuten würde. Dies auch nur unter der gemachten Voraussetzung, nur Integer ALU–Befehle zu betrachten, da FP–Operationen meist nicht in einem Taktzyklus zu bewältigen sind. Als Hardwareeinsparung könnte die Berechnung von NPC in der ALU ausgeführt werden, die in der IF–Phase sonst nicht gebraucht wird.

6.2 Die DLX–Pipeline

Trotz der angesprochenen Probleme versuchen wir nun die einzelnen Abarbeitungsschritte als Stufen einer Pipeline zu verwenden, deren Prinzip in Abb. 6.1 dargestellt ist.

	Takt	1	2	3	4	5	6	7	8
Befehl	01	IF	ID	EX	MEM	WB			
Befehl	02		IF	ID	EX	MEM	WB		
Befehl	03			IF	ID	EX	MEM	WB	
Befehl	04				IF	ID	EX	MEM	WB

Abbildung 6.1: Einfache DLX–Pipeline.

Für jeden Taktzyklus sind die gerade bearbeiteten Befehle in den verschiedenen Pipelinestufen tabellarisch dargestellt. Wenn die vier Befehle in Abb. 6.1 ohne Probleme mit Pipelining bearbeitet werden können, dann würden sie insgesamt 8 Taktzyklen benötigen, was eine CPI–Rate von 2.0 ergibt. Vergleichen wir diesen Wert mit der CPI–Rate für die DLX ohne Pipeline (> 4.0) so sehen wir, dass wir die Rechengeschwindigkeit mehr als verdoppelt

haben, doch vom theoretischen Maximum eines Speedups von 5 doch weit entfernt sind (warum?).

Betrachten wir aber Abb. 6.1 genauer, so erkennen wir bald einige der vielen Probleme, die das Pipelining auch mit sich bringt. Sehen wir uns einmal die Speicherzugriffe näher an. Die betroffenen Pipelinestufen sind hierbei IF und MEM. In den Taktzyklen 4 und 5 werden diese beiden Stufen gleichzeitig bearbeitet. Was passiert aber nun, wenn IF einen Befehl aus dem Speicher holen möchte und ein Load–Befehl einen Speicherzugriff in MEM nötig macht? Üblicherweise haben wir nur ein Busssytem als Verbindung zum Speicher zur Verfügung. Ein gleichzeitiger Zugriff ist daher nicht möglich. Eine mögliche Abhilfe wäre hier zwei separate Speicher in der CPU, wobei der eine Befehle und der andere Daten speichert (das löst aber das Problem auch nicht ganz, warum?).

Ein weiteres Problem kann die gleichzeitige Bearbeitung von ID und WB aufwerfen, falls dieselben Register angesprochen werden. Klarerweise kann ein Register nicht gleichzeitig gelesen und geschrieben werden (theoretisch schon, allerdings wäre der Inhalt dann undefiniert). Auch ein *Taken Branch* (Bedingung für Branch ist erfüllt), der in der MEM–Stufe berechnet wird bereitet ein Problem für Pipelinedesigner. Es wurden nämlich mittlerweile schon drei falsche Befehle aus dem Speicher geholt.

Wie oben schon angedeutet hat unsere einfache DLX–Implementierung auch den Nachteil, dass sie zuwenige interne Register aufweist. Wird z. B. in der MEM–Stufe der Wert im Zwischenregister `B` in den Speicher geschrieben, so stammt der Wert in `B` von dem Befehl, der in der vorigen ID–Stufe behandelt wurde, aber nicht vom ursprünglichen Store–Befehl. Dies führt dazu, dass eine Pipelineversion eines Prozessors zwischen den einzelnen Pipelinestufen *Pipeline Register* haben muss, die die benötigten Informationen für die nächste Stufe zur Verfügung stellen. In unserem Beispiel würde jeweils ein Register zwischen ID und EX (`ID/EX.B`) und EX und MEM (`EX/MEM.B`) dafür sorgen, dass der richtige Wert abgespeichert wird.

Konsequenz dieser Überlegungen ist eine mögliche (aber nicht perfekte) Pipelineversion der DLX in Abb. 6.2.

Der primäre Informationsfluss geht von der IF–, zur WB–Stufe. Verbindungen, die zumindest eine Stufe überspringen, sind Rückkopplungen (z. B. `MEM/WB.IR` von WB nach ID), während strichlierte Linien Steuerleitungen der Multiplexer (MUX) symbolisieren. Zwischen den einzelnen Pipelinestufen sind die Pipelineregister (z. B. wird IR von Stufe zu Stufe weitergereicht, um jeweils die korrekte Aktion über Multiplexer zu schalten) (überlegen Sie welche Probleme auftreten würden, wenn das IR nur bis zur ID–Stufe weitergereicht würde).

Die Aktionen, die in dieser Implementierung von der IF–Stufe durch-

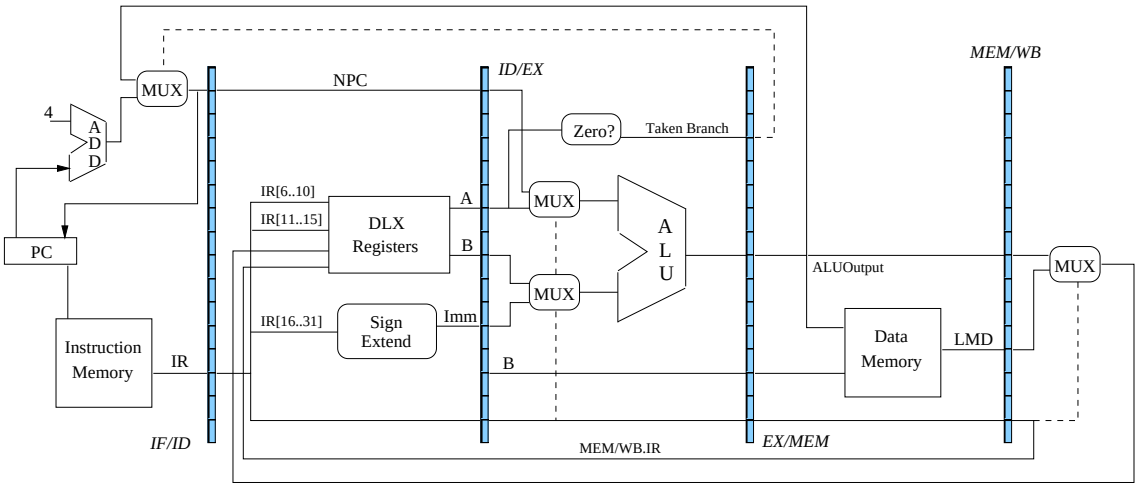

Abbildung 6.2: Implementierung einer DLX–Pipeline (vereinfacht).

geführt werden, sind:

```
IF/ID.IR <- Mem[PC]
IF/ID.NPC <- if (EX/MEM.cond) {EX/MEM.ALUOutput} else {PC + 4}
```

Wir sehen, dass bei einem Taken Branch die in der MEM–Stufe berechnete Sprungadresse als neuer Program Counter (PC) verwendet wird. Ist die Branchbedingung nicht erfüllt, dann wird der PC einfach inkrementiert und damit das Programm sequentiell abgearbeitet. Die Aktionen in den übrigen Stufen entsprechen im wesentlichen denen der DLX–Impementierung ohne Pipeline unter Berücksichtigung der zusätzlichen Pipelineregister.

Nun sehen wir auch, warum es wichtig ist, dass Befehle und auch die Pipelinestufen möglichst gleiche Ausführungszeiten haben sollten, denn die Taktzykluszeit muss sich nach der langsamsten Pipelinestufe richten. Bei grösseren Unterschieden zwischen den Stufen, würden die schnellen ständig auf die langsamen Stufen warten müssen. Die Pipelineregister stellen nicht nur zusätzliche Hardware dar, sie verlangsamen auch den Signalfluss durch die Pipeline (*Register Delay*). Zusätzlich limitiert *Clock Skew* (das asynchrone Eintreffen des Taktsignals an den Stufen) die Taktzykluszeit. Dies alles führt dazu, dass reale Speedups durch eine Pipeline nie an das theoretische Maximum heranreichen, auch wenn es gelänge, sämtliche Abhängigkeiten zwischen den Pipelinestufen zu eliminieren.

6.3 Pipeline Hazards

Kann eine Abhängigkeit zwischen Pipelinestufen nicht beseitigt werden, oder wäre eine Lösung zu aufwendig, so tritt ein Konflikt zwischen den betroffenen Pipelinestufen auf. Einen solchen Konflikt nennen wir *Hazard*, den wir auf drei prinzipielle Ursachen zurückführen können.

1) *Structure Hazards* entstehen durch die gleichzeitige Verwendung von Hardwareeinheiten durch zwei oder mehrere Stufen.

2) *Data Hazards* werden durch Abhängigkeiten von Daten verursacht, die von einzelnen Befehlen zu einem Zeitpunkt benötigt werden, an denen diese nicht zur Verfügung stehen.

3) *Control Hazards* treten durch verzögerte Erkennung von Befehlen auf, die den Program Counter und damit die Programmabarbeitung verändern.

Im Falle des Auftretens eines Hazards muss die Pipeline dafür sorgen, dass keine undefinierten (oder falschen) Zustände während der Befehlsabarbeitung eintreten. Die einfachste Methode besteht darin, die Stufe zu blockieren, die im Signalfluss vor der Stufe steht, mit der ein Konflikt auftritt. Ein derartiges Blockieren wird als *Stall* bezeichnet. Alle Befehle, die nach dem Stall an die Reihe kommen würden, müssen dann auch blockiert werden. Diejenigen Befehle, die schon in der Pipeline sind und vor dem blockierenden Befehl begonnen wurden, werden weiter durch die Pipeline geschickt.

6.3.1 Structure Hazards

Ein Beispiel für einen Structure Hazard ist der schon angesprochene Konflikt beim Speicherzugriff zwischen den Stufen IF und MEM. Exisitert nur ein Hauptspeicher, dann wird dieser Konflikt durch einen Stall gelöst (Abb. 6.3).

	Takt	1	2	3	4	5	6	7	8	9	10
Befehl	**Load**	IF	ID	EX	MEM	WB					
Befehl	02		IF	ID	EX	MEM	WB				
Befehl	03			IF	ID	EX	MEM	WB			
Befehl	04				**stall**	IF	ID	EX	MEM	WB	
Befehl	05						IF	ID	EX	MEM	WB

Abbildung 6.3: Structure Hazard Stall einer DLX–Pipeline.

Der Befehl 04 löst einen Stall aus, da der IF einen Speicherzugriff gleichzeitig mit dem Load Befehl in der MEM–Stufe bedinge würde. Alle nachfolgenden Befehle werden auch blockiert und um einen Taktzyklus verzögert ausgeführt. Man könnte auch für Befehl 05 einen Stall in das Diagramm 6.3 eintragen, allerdings würde das den Eindruck erwecken, als ob zwei Stalls auftreten würden. Es werden aber alle nachfolgenden Befehle um genau einen Taktzyklus verzögert, was alles eine Konsequenz des einzigen Stalls bei Befehl 04 ist (welche Folge hätte ein Store–Befehl 03?).

Beispiel: Um die Konsequenzen eines derartigen Structure Hazard abzuschätzen, nehmen wir an, dass eine Maschine ohne Auftreten von Hazards den idealen CPI–Wert 1.0 hätte. Eine Vergleichsmaschine hat separate Speicher (Caches) für Befehle und Daten. Die zusätzliche Hardware bedingt eine Verringerung der Taktrate um 5% gegenüber der ursprünglichen Maschine. Welche der beiden Maschinen ist schneller, wenn in einem Programm 40% aller Befehle Loads sind?

Wir bezeichnen die Maschine mit einem gemeinsamen Speicher mit dem Index m (Memory) und die verbesserte Maschine mit c (Cache). Für die CPI–Raten ergibt sich dann

$$CPI_m = 1 + 0.4 = 1.4 \qquad CPI_c = 1.0,$$

und für die Ausführungszeiten eines Programms mit n Befehlen

$$t_m = nT_m CPI_m \quad t_c = nT_c CPI_c,$$

und mit Berücksichtigung von $f_c = 0.95 f_m$ ein Speedup

$$s = \frac{t_m}{t_c} = \frac{f_c\, CPI_m}{f_m\, CPI_c} = \frac{0.95 \times 1.4}{1.0} = 1.33.$$

Die Maschine mit zusätzlicher Hardware, die den speziellen Structure Hazard vermeidet ist (wie erwartet) schneller. Allerdings erhöhen sich die Hardwarekosten und es ist im Einzelfall abzuschätzen, ob dieser Aufwand gerechtfertig ist. ◊

6.3.2 Data Hazards

Dieser Konflikttyp wird durch Vertauschen der Reihenfolge von Read/Write–Operationen von Daten (z. B. in Registern) in der Pipeline verursacht. Bei einer Maschine ohne Pipeline ist garantiert, dass ein Befehl vollständig abgearbeitet ist, bevor der nächste bearbeitet wird. Damit steht das Ergebnis des vorhergehenden Befehls absolut sicher zur Verfügung. In einer Pipeline ist dies nicht mehr garantiert wie folgendes Bespiel zeigt.

```
ADD  R1, R2, R3
LW   R4, 0(R1)
SW   12(R1), R4
```

Wenn wir diese Befehlssequenz in ein *Taktzyklendiagramm* der Pipeline eintragen, dann erkennen wir, dass allein diese drei Instruktionen einige Probleme bereiten (Abb. 6.4).

Takt	1	2	3	4	5	6	7
ADD R1, R2, R3	IF	ID	EX	MEM	WB		
LW R4, 0(R1)		IF	ID	EX	MEM	WB	
SW 12(R1), R4			IF	ID	EX	MEM	WB

Abbildung 6.4: Beseitigung von Data Hazards durch Forwarding (...).

Zunächst soll der `ADD`–Befehl das Ergebnis in `R1` ablegen. Dieser Wert, der erst nach WB in `R1` zur Verfügung steht (Takt 6) wird aber schon in der EX–Stufe des Load–Befehls (Takt 4) benötigt. Entweder die Pipeline fügt zwei Stalls ein, oder wir lassen uns eine elegantere Lösung einfallen. Auch beim Store–Befehl benötigen wir `R1` in der EX–Phase (Takt 5), in der der korrekte Wert auch noch nicht zur Verfügung steht. Schliesslich möchte der Store–Befehl den Wert von `R4` abspeichern (Takt 6), der aber erst nach dem WB des Load–Befehls (Takt 7) zur Verfügung steht.

Eine (relativ) einfache Möglichkeit, diese Data Hazards zu beseitigen, wäre das Einfügen von Stalls. Dies scheint zwar für dieses simple Beispiel sehr aufwendig, doch durch Beseitigen des ersten Hazards (zwei Stalls bei Load) wird auch der zweite beseitigt. Der dritte Konflikt muss allerdings wieder durch einen Stall gelöst werden. In Summe würden die drei Stalls dann eine Ausführungszeit von 10 Taktzyklen bewirken, was einer Erhöhung um 43% entspricht. Diese eklatante Verlangsamung bei einer sehr häufig auftretenden Befehlssequenz verlangt eine bessere Lösung.

Für den Load–Befehl brauchen wir den korrekten Wert von `R1`. Dieser Wert steht zwar erst nach WB im gewünschten Register, der Wert selbst aber ist schon am Ende der EX–Stufe berechnet. Wenn wir nun entsprechende Hardware vorsehen, die erkennt, dass der Befehl für die EX–Stufe im nächsten Takt dieses Ergebnis benötigt, dann könnte der Wert direkt übergeben, und damit Stalls vermieden werden. Diese Technik nennt man *Forwarding* (auch *Bypassing, Short–Circuiting*)

In unserem Beispiel (Abb.6.4) werden drei verschiedene Forwardings benötigt, um die Sequenz ohne Stalls abarbeiten zu können, EX→EX, MEM→EX und

MEM→MEM. Technisch geschehen geschieht das Forwarding am Taktende der übergebenden Stufe. Es wird also z. B. am Ende der EX–Stufe das Ergebnis in die Pipelineregister ID/EX geschrieben, und so der Pipeline "vorgetäuscht", dass der Wert korrekt aus dem Register geholt wurde.

Natürlich muss die Pipeline zuerst erkennen, ob dieses Überschreiben nötig ist. Dazu muss verglichen werden, ob das Zielregister des ersten Befehls ein Quellregister des folgenden Befehls ist. Im Falle des Forwarding EX→EX muss diese Bedingung aus den Pipelineregistern EX/MEM.IR und ID/EX.IR extrahiert werden (überlegen Sie wie und versuchen Sie auch für die anderen Forwardings entsprechende Hardware in das Blockschaltbild 6.2 einzufügen).

Üblicherweise findet Forwarding immer nur von einem Takt zum nächsten statt, da sonst wieder zusätzliche Hardware erforderlich wäre, um Ergebnisse zwischenzuspeichern. Wie effizient diese Technik aber sein kann, ist an Abb. 6.4 zu erkennen, wo durch Forwarding alle Stalls vermieden werden können. Dass letzteres nicht immer möglich ist, zeigt das nächste Beispiel

```
LW   R1, 0(R2)
SUB  R4, R1, R5
AND  R6, R1, R7
OR   R8, R1, R9
```

mit dem zugehörigen Taktzyklendiagramm (Abb. 6.5).

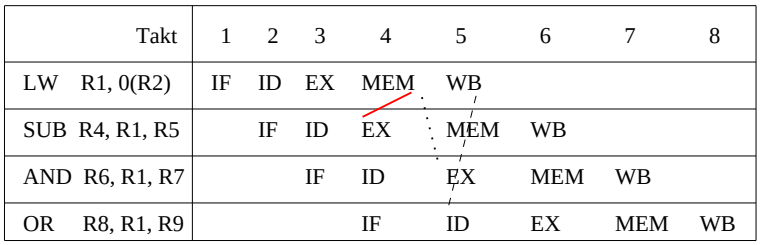

Abbildung 6.5: Takthalbierung WB|ID und "Backwarding" (nicht möglich, daher Stall in Takt 4).

Bei der Betrachtung dieser Abbildung ist Vorsicht geboten, denn sie impliziert die Möglichkeit, dass wir Daten aus der Zukunft in die Gegenwart übertragen können (von MEM in Takt 4 zu EX in Takt 4). Dies ist daher ein Beispiel dafür, dass auch Forwarding nicht immer einen Stall vermeiden kann. Hier müsste also die EX–Stufe des `SUB`–Befehls um einen Takt verzögert werden.

Würde im `SUB`-Befehl `R1` mit `R2` ersetzt sein, würde kein Stall erforderlich sein. Dann bleibt das Forwarding von MEM (T4) auf EX (T5), und in Takt

5 tritt ein neues Problem ein. Der `OR`–Befehl benötigt `R1`, das jedoch erst nach Takt 5 zur Verfügung steht. Man könnte nun ein weiteres Forwarding MEM→ID einführen. Einfacher jedoch ist die Technik der *Takthalbierung*.

Wenn in der ersten Hälfte eines Taktes die Register in WB beschrieben werden, und in der zweiten Hälfte dieses Taktes die Register in ID gelesen werden, dann erspart man sich zusätzliche Forwarding–Hardware.

Abhängig von der Reihenfolge des Read/Write–Zugriffs auf ein Datum, lassen sich die Data Hazards in drei Kategorien unterteilen. Die Bezeichnung dieser Kategorien geht von der *richtigen* Reihenfolge der Zugriffe in der Befehlssequenz aus.

RAW (Read after Write)

Dieser Data Hazard tritt auf, wenn eine Instruktion j ein Datum lesen will, das von einer vorhergehenden Instruktion i berechnet, aber noch nicht geschrieben worden ist. Dies ist der häufigste Konflikt und alle Data Hazards in Abb. 6.4 und Abb. 6.5 sind von diesem Typ.

WAW (Write after Write)

Wenn eine Instruktion j ein Datum schreibt, bevor es von einer vorhergehende Instruktion i beschrieben wird, dann steht nach diesen beiden Operationen der falsche Wert (der Instruktion i) im Speicher/Register. In der vorgestellten Implementierung einer DLX–Pipeline kann dieser Hazard nicht auftreten, da nur in der WB–Phase ein Register (bzw. in der MEM–Phase der Speicher) beschrieben wird.

Wenn wir aber zwei Veränderungen der DLX–Pipeline ins Auge fassen, dann könnte es zu einem WAW–Hazard kommen. Die WB–Stufe eines ALU–Befehls könnte in die MEM–Stufe vorgezogen werden, und ein Load–Befehl könnte zwei MEM–Stufen benötigen (Abb. 6.6).

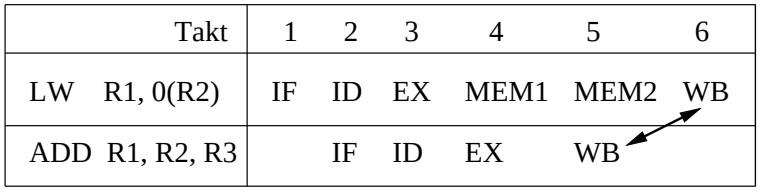

Abbildung 6.6: WAW Hazard einer modifizierten DLX–Pipeline.

WAR (Write after Read)

Ein Befehl j schreibt ein Datum bevor dieses vom vorhergehenden Befehl i gelesen wird, was dazu führt, dass i den falschen Wert liest. Dieser

Konflikt tritt in einer einfachen Pipeline selten auf, da typischerweise in den ersten Stufen gelesen und in den letzten geschrieben wird. Betrachten wir unsere modifizierte DLX–Pipeline und nehmen wir an, dass die Quelle eines Store–Befehls erst in MEM2 gelesen wird, dann ist ein WAR–Hazard möglich (Abb. 6.7).

Takt	1	2	3	4	5	6
SW 0(R1), R2	IF	ID	EX	MEM1	MEM2	WB
ADD R2, R3, R4		IF	ID	EX	WB	

Abbildung 6.7: WAR Hazard einer modifizierten DLX–Pipeline.

Wird das Quellregister des Store–Befehls in der zweiten Hälfte von MEM2 und das Zielregister in der ersten Hälfte von WB geschrieben (es bedarf schon ziemlicher Anstrengung einen WAR Hazard in DLX zu konstruieren), dann würde dieser Konflikt auftreten.

RAR (Read after Read)

Dies stellt keinen Konflikt dar (warum?).

Mit allen Varianten des Forwarding und der Takthalbierung WB|ID bleibt für die DLX–Pipeline ein einziger Data Hazard bestehen, der aber recht häufig auftreten wird: ein Load–Befehl mit einem folgenden Befehl, der das geladene Datum verwenden möchte (RAW–Stall). Man bezeichnet diesen Effekt als *Load Delay* mit einer Latenzzeit von einem Taktzyklus *Single–Cycle Latency*, was nur ausdrückt, dass ein Load–Befehl genau einen Stall auslösen kann. Tiefere Pipelines (viele Stufen) oder Pipelines in denen einige Operationen (wie Floating Point) als Stufen ausserhalb der Pipeline organisiert sind haben üblicherweise mehrere Delay–Befehle mit einer längeren Latenzzeit.

Für die DLX–Pipeline ist es relativ einfach einen bevorstehenden Load Delay zu erkennen. Dazu müssen einfach das Zielregister des Load–Befehls (ID/EX.IR) und die möglichen Quellregister des nächsten Befehls (IF/ID.IR) überprüft werden. Wird ein Konflikt entdeckt, so wird im nächsten Takt der Opcode von ID/EX.IR auf eine *No Operation* (z. B. `ADD R0, R0, R0`) gesetzt und IF/ID.IR nicht verändert. Dadurch wird der Load–Befehl weiter in der Pipeline bearbeitet, während der nachfolgende Befehl blockiert (Stall).

Bisher sind wir immer von Befehlssequenzen ausgegangen, die absichtlich so gewählt waren, dass Probleme bei der Bearbeitung in der Pipeline auftreten. Daraus folgt aber auch, dass wir Befehle so an–, und umordnen könnten, dass es zu möglichst wenigen Problemen kommen wird.

6.3.3 Static Scheduling

Wir wollen dieses Umordnen von Befehlen zur Vermeidung von Data Hazards an einem Beispiel illustrieren. Diese Technik, die auch *Pipeline Scheduling* genannt wird, ist ein komplexer Bestandteil jedes modernen Compilers. Da dieses Umordnen zur Compilezeit stattfindet, ist diese Methode statisch, da sie nicht dynamisch auf zur Laufzeit auftretende Befehlssequenzen reagieren kann. Wir übernehmen kurz die Arbeit eines Compilers.

Beispiel: Das Programmfragement in Pseudocode

```
a = b + c;
d = e - f;
```

soll in DLX–Code übersetzt werden. Bei (naiver) sequentieller Vorgangsweise würden wir folgende Befehlssequenz erhalten:

```
LW   R2, b
LW   R3, c
ADD  R1, R2, R3      ; a = b + c
SW   a, R1
LW   R5, e
LW   R6, f
SUB  R4, R5, R6      ; d = e - f
SW   d, R4
```

Diese Sequenz würde zwei Load Delays (Stalls) verursachen, die wir aber leicht eliminieren können:

```
LW   R2, b
LW   R3, c
LW   R5, e           ; delay slot
ADD  R1, R2, R3      ; a = b + c
LW   R6, f
SW   a, R1           ; delay slot
SUB  R4, R5, R6      ; d = e - f
SW   d, R4
```

An die Stelle nach einem Load–Befehl – der *Delay Slot* – wird ein Befehl eingefügt, der das geladene Datum *nicht* verwendet. Dies setzt natürlich voraus, dass es Befehle gibt, die sich, ohne das Programm semantisch zu ändern, verschieben lassen. Hier wird auch sehr deutlich, dass heutige Compiler sehr komplexe Aufgaben bewältigen müssen, da es nicht einmal für diese simple

Sequenz einfach ist, ein Programm zu schreiben, das diese Umordnungen automatisiert vornimmt.

Ein Compiler muss also entscheiden können, welche Befehle *parallel* sind und welche nicht. Parallele Befehle sind solche, die keinerlei Abhängigkeiten voneinander haben und daher in einer Pipeline sicher nicht zu Stalls führen. Durch Ausnutzen des *Instruction Level Parallelism* (ILP) lassen sich Programme umordnen und Stalls vermeiden. Wir unterscheiden drei Kategorien von Abhängigkeiten zwischen Befehlen:

- *Data Dependence*

  tritt dann auf, wenn eine Instruktion j ein Datum benötigt, das von Instruktion i produziert wird (möglicher RAW–Hazard). Dies zu erkennen, wäre noch relativ einfach, allerdings kann es auch zu Ketten von Abhängigkeiten kommen (j hängt von k ab und k von i usw.), was das Problem wesentlich kompliziert.

- *Name Dependence*

  bezeichnet die Verwendung derselben Register oder Speicherstellen (*Name*) durch zwei oder mehrere Befehle, ohne dass hierbei ein Datenfluss zwischen den Befehlen existiert. Wir können diese Abhängigkeit noch differenzieren in

  - *Antidependence*, die auftritt, wen Befehl i ein Datum liest und Befehl j auf denselben Namen schreibt (möglicher WAR–Hazard). Instruktion j übernimmt das Datum also nicht, trotzdem kann ein Konflikt auftreten.

    Beispiel: Hier liegt eine Antidependence der Befehle vor.

    ```
    ADD  R3, R5, R2
    SUB  R2, R4, R6
    ```

  - *Output Dependence*, bezeichnet einen möglichen WAW–Hazard bei Beschreiben desselben Namens durch Befehle i und j.

  Eine einfache aber wirkungsvolle Möglichkeit Name Dependences zu eliminieren, ist das *Register Renaming*, bei dem der Compiler nach Möglichkeit ein anderes Register verwendet, um den Konflikt aufzulösen. Bei Speicherstellen ist diese Technik nur sehr bedingt möglich (warum?).

- *Control Dependence*

  Klarerweise liegt immer eine Abhängigkeit vor, wenn Befehle nur bedingt ausgeführt werden. Diese Befehle können im allgemeinen nicht vor

einen Branch umgeordnet werden. Analog können Befehle, die immer ausgeführt werden, nicht in einen Branch umgeordnet werden. Wir werden bei der Behandlung der Control Hazards noch Methoden kennenlernen, wie man den Einfluss dieser Abhängigkeiten verringern kann.

Generell limitieren die Data Dependences die Möglichkeiten des Static Scheduling, da diese Abhängigkeiten die Semantik des Programms repräsentieren, die der Compiler unter keinen Umständen ändern darf. Ein Nachteil des Static Scheduling ist die grosse Abhängigkeit zwischen Compiler und Hardware, da der Compiler nur dann sinnvoll umordnen kann, wenn er alle Details der Hardware, insbesondere der Pipeline, kennt. Wird ein Prozessor weiterentwickelt (z. B. die Anzahl der Pipelinstufen erhöht), so muss auch der Compiler angepasst werden, da sich sonst Verbesserungen für den alten Prozessor in Verschlechterungen für den neuen verwandeln können. Daher gab es schon seit Beginn der Prozessorentwicklung die Idee, das Umordnen von Befehlen während der Laufzeit des Programms dem Prozessor zu überlassen.

6.3.4 Dynamic Scheduling

Im Gegensatz zu einem Compiler kann der Prozessor einzelne Befehle nur lokal analysieren und sich nicht das ganze Programm "ansehen". Trotzdem könnte die Hardware relativ leicht feststellen, dass zwei Befehle, die hintereinander aus dem Speicher geholt werden, parallel sind. Diese Feststellung würde uns bei der einfachen DLX–Pipeline aber auch nicht helfen, da diese nicht mehrere funktionale Einheiten (z. B. ALU) hat. Hätten wir diese, dann könnten wir zwei oder mehrere Befehle auf zwei oder mehrere funktionale Einheiten aufteilen (wenn die Befehle parallel sind), und so Befehle im wahrsten Sinne des Wortes parallel ausführen. Allerdings führt dies unweigerlich zu neuen Problemen, da es dann passieren kann, dass ein Befehl j den Befehl i "überholt" und früher abgearbeitet ist als der Befehl, der eigentlich vor ihm steht.

Wir brauchen daher zusätzliche Kontrollhardware, die die parallele Abarbeitung der Befehle veranlasst und überwacht. Wir werden hier auf die Grundzüge zweier Verfahren eingehen, die ein Dynamic Scheduling ermöglichen (Hennessy and Patterson, 1996). Während ein System alle Befehle zentral koordiniert und synchronisiert, versucht das andere auch die Kontrolle zu parallelisieren (und damit zu dezentralisieren).

Zentraler Punkt aller Verfahren für Dynamic Scheduling ist das Aufteilen der ID–Stufe einer Pipeline in folgende zwei Stufen:

1) Issue

Hier wird der Befehl decodiert und überprüft, ob eine funktionale Einheit frei ist, die diesen Befehl ausführen kann (möglicher Structure Hazard). Ist eine Einheit für diesen Befehl frei, so wird der Befehl an diese Einheit zur Bearbeitung übergeben.

2) Read Operands

In der funktionalen Einheit wartet der Befehl solange, bis sichergestellt ist, dass keine Data Hazards auftreten können, liest dann die Operanden und beginnt die Ausführung (EX–Stufe).

Das *Scoreboard*–Verfahren für Dynamic Scheduling sieht eine Hardwareeinheit vor, die alle Befehle, die abgesetzt wurden (issued), bis zu ihrer Beendigung überwacht. Das Scoreboard muss alle möglichen Hazards feststellen und vermeiden können. Man könnte das Scoreboard auch als *CPU Manager* bezeichnen, der die Befehlsarbeitung zentral überwacht und synchronisiert. Wie schon oben erwähnt benötigen wir mehrere funktionale Einheiten, um so Befehle parallel bearbeiten zu können. Ein Beispiel für eine Architektur mit mehreren funktionalen Einheiten und einem Scoreboard zeigt Abb. 6.8.

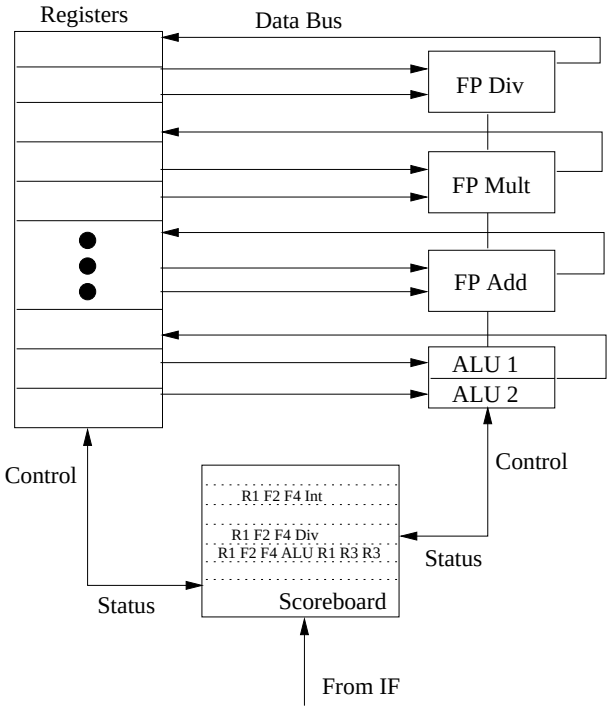

Abbildung 6.8: Dynamic Scheduling Architektur mit Scoreboard, drei FP Einheiten und zwei Integer ALUs.

Die Instruktionen kommen aus der IF–Stufe in das Scoreboard (Hardwareeinheit), das nun vier wesentliche Schritte für jeden Befehl überwacht. Diese Schritte ersetzen die ID–, EX–, und WB–Stufen unserer einfachen DLX–Pipeline.

1) Issue

   Das Scoreboard erhält den nächsten Befehl von der IF–Stufe und überprüft, ob eine funktionale Einheit für die Bearbeitung frei ist (Structure Hazard Check). Ist eine verfügbar, so wird auch geprüft, ob zur Zeit ein Befehl aktiv ist, der dasselbe Zielregister hat (WAW–Hazard Check). Entsteht kein Konflikt, so wird der Befehl an die funktionale Einheit abgesetzt (issued). Besteht ein Konflikt, so gibt es auch hier einen Stall. Befindet sich zwischen IF und Scoreboard ein Puffer, so können aber auch während des Stalls Instruktionen aus dem Speicher geholt werden.

2) Read Operands

   Das Scoreboard überwacht, ob eine andere Einheit Register schreiben wird, die von dem neu abgesetzten Befehl benötigt werden (RAW–Hazard Check). Ist dies der Fall, so wird solange gewartet bis die Register geschrieben sind, und erst dann wird die Ausführung des Befehls auf der Einheit gestartet. Dies hat aber auch zur Konsequenz, dass ein später abgesetzter Befehl früher bearbeitet werden kann (*execution out of order*). Man sieht hier klar, dass ein Stall nur den jeweiligen Befehl betrifft, der nächste Befehl aber abgesetzt werden kann (wenn er parallel ist), und die Befehle dynamisch umgeordnet werden (können).

3) Execution

   Die Einheit führt die Berechnung durch und signalisiert dem Scoreboard deren Ende. Die Durchführungszeit ist abhängig von der jeweiligen Einheit, was hier aber kein Problem darstellt, da mehrere Einheiten parallel arbeiten. Dieser Schritt entspricht der EX–Phase der einfachen DLX–Pipeline.

4) Write Result

   Unter der Bedingung, dass kein vorher abgesetzter Befehl das Register benötigt und noch nicht gelesen hat (Read Operands) wird das Zielregister beschrieben. Wenn dieser WAR Hazard entdeckt wird, blockiert das Scoreboard das Schreiben des Registers solange, bis der Konflikt aufgelöst ist. Dieser Schritt implementiert die WB–Stufe der einfachen DLX–Pipeline.

Um die einzelnen Überprüfungen machen zu können, muss das Scoreboard folgende Einträge aufweisen:

- Instruction Status

  Enthält für jede Instruktion, in welchem Schritt sie gerade ist.

- Functional Unit Status

  Der Zustand der funktionalen Einheiten wird hier protokolliert. Als Beispiel seien neun mögliche Einträge angeführt. *Busy* (Einheit belegt), *Op* (Operation, die ausgeführt werden soll), *Fi* (Zielregister), *Fj, Fk* (Quellregister), *Qj, Qk* (Einheiten, die Quellregister produzieren) und *Rj, Rk* (Flags, die anzeigen, ob Quellregister fertig).

- Register Result Status

  Gibt für jedes Register an, welche Einheit ein Ergebnis dorthin schreiben wird (Nulleintrag, wenn keine Einheit assoziiert).

Die Geschwindigkeitsgewinne, die man mit dieser Variante des Dynamic Scheduling erzielen kann, hängen (wie zumeist) von vielen Faktoren ab. Wenn jede Instruktion von der vorigen abhängt, kann Dynamic Scheduling gar nichts bewirken, da immer auf das Ende des jeweiligen Befehls gewartet werden muss. Mit anderen Worten heisst das, dass im Programm möglichst viele parallele Befehle vorkommen sollten, um den Prozessor mit Dynamic Scheduling beschleunigen zu können.

Die Anzahl der Einträge, die das Scoreboard aufnehmen kann, bestimmt die Anzahl von Befehlen (*Window*), die (implizit) auf Parallelität getestet werden können. Ausserdem treten Probleme auf, wenn Branches im Window enthalten sind, da die folgenden Befehle nur bedingt ausgeführt werden.

Die Anzahl und die Typen der funktionalen Einheiten bestimmen die Häufigkeit des Auftretens von Structure Hazards.

Schliesslich beeinflussen Antidependences und Output Dependences die Anzahl der Stalls in den einzelnen Einheiten.

Der Hardwareaufwand für ein Scoreboard kann überraschend gering gehalten werden und ist vergleichbar mit dem einer Funktionseinheit. Allerdings ist auch das zusätzliche Bussystem zu beachten, dass die Daten zwischen Registern und den einzelnen Einheiten transferiert.

Als Weiterentwicklung der Scoreboard–Technik kann das Dynamic Scheduling nach *Tomasulo* [1] betrachtet werden. Die wesentlichen Änderungen

[1] Robert Tomasulo war Entwickler der IBM 360/91.

dabei sind die Verteilung der Hazard Kontrolle (Dezentralisierung) und *Register Renaming* über einen *Common Data Bus*, der hier zentraler Bestandteil einer Methode ist, die dem uns schon bekannten Forwarding gleicht. Abb. 6.9 zeigt ein Blockschaltbild einer Floating Point Einheit, die nach Tomasulo's Prinzip organisiert ist.

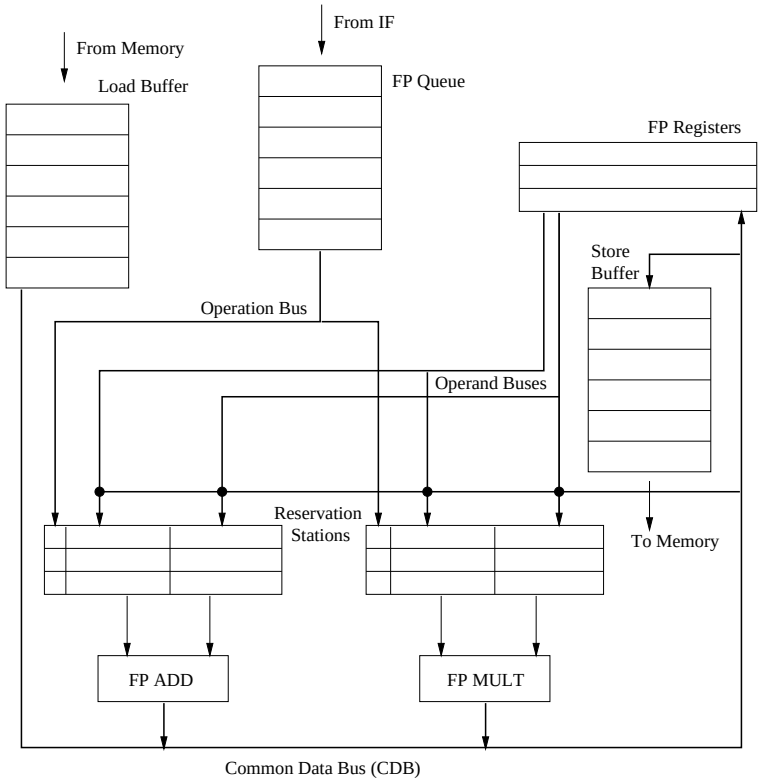

Abbildung 6.9: Dynamic Scheduling einer FP Einheit nach Tomasulo.

Die wesentliche Änderung zur Scoreboard–Technik sind die *Reservation Stations*, die die Kontrolle über die ihnen zugeordnete funktionale Einheit übernehmen. Über diese Reservation Stations (RS) wird auch das Register Renaming implementiert. Stehen Operanden nicht in einem Register zur Verfügung, dann wartet eine RS nicht auf das Schreiben des Registers, sondern die Hardware gibt dieser RS die Information, welche andere RS den benötigten Operanden gerade berechnet. Ist die Berechnung abgeschlossen, dann wird dieser Operand über den Common Data Bus direkt an die wartende RS übertragen. In diesem Sinne übernimmt also der Name der RS den Namen des Registers (Register Renaming). Gibt es mehr RS als Register, so kann diese Methode Hazards vermeiden, die ein Compiler auch theoretisch nicht verhindern könnte, da dieser nur Register umbenennen kann.

Wir können hier drei Schritte identifizieren, die die ID–, EX–, und MEM–Phasen einer einfachen DLX Pipeline ausführen. Wir betrachten hier neben FP Operationen auch Load–, und Store–Befehle eines FP Operanden, die in einer eigenen funktionalen Einheit behandelt werden.

1) Issue

   In der *FP Queue*, die von der IF–Stufe mit FP Instruktionen versorgt wird, wird der nächste Befehl analysiert. Ist es eine FP Operation, so wird überprüft, ob eine Reservation Station bereit ist, den Befehl zu übernehmen. Ist es ein Load/Store, dann wird geprüft, ob im jeweiligen Buffer noch ein Eintrag frei ist, um den Befehl auszuführen. Hier werden auch die Register umbenannt, indem einer RS (oder einem Load/Store) mitgeteilt wird, aus welcher RS die Operanden kommen. Dadurch werden WAW–, und WAR–Hazards eliminiert.

2) Execute

   Die RS stellt fest, ob alle Operanden für den Befehl zur Verfügung stehen. Falls Operanden noch nicht zur Verfügung stehen, wartet die RS, bis über den Common Data Bus die Operanden eingetroffen sind (RAW–Hazard Check). Sind die Operanden vorhanden, dann wird der Befehl an die funktionale Einheit übergeben.

3) Write Result

   Wenn ein Operand verfügbar ist (fertig berechnet oder geladen), dann wird er über den CDB an alle wartenden RS, in das Register, und in einen wartenden Store Buffer geschrieben. Dies minimiert die Wartezeiten, da ein Ergebnis an alle wartenden Einheiten sofort und gleichzeitig übermittelt wird.

Da in Tomasulo's Methode die Kontrolle über die Abarbeitung der Befehle verteilt ist, müssen auch mehrere Einheiten Statusinformationen protokollieren.

- Reservation Station

  Hier müssen sechs Felder für wichtige Informationen vorgesehen werden: *Op* (Operation, die ausgeführt werden soll), *Qj, Qk* (die Nummer der RS, die die Operanden generiert, 0 falls schon vorhanden), *Vj, Vk* (der Wert der Operanden), *Busy* (die RS ist belegt).

- Register File

Qi enthält die Nummer der RS, die ein Resultat für ein Register produziert. Diese Information genügt, um WAW–, und WAR–Hazards zu eliminieren (warum?).

- Load/Store Buffers

  Diese funktionalen Einheiten müssen Information darüber bereitstellen, ob ein Eintrag frei ist (*Busy*). Zusätzlich muss ein Store Buffer auch ein Feld *Qi* aufweisen, das wie beim Register File die produzierende RS anzeigt.

Das Dynamic Scheduling nach Tomasulo kann sehr guten Geschwindigkeitsgewinn bringen, allerdings auf Kosten von umfangreicher Zusatzhardware (RS, CDB). Auch bei dieser Methode stellen Branches das grösste Problem dar. Deshalb wollen wir uns nun den dadurch verursachten Controll Stalls zuwenden, und statische und dynamische Techniken zur Vermeidung oder Verringerung dieses Konflikttyps behandeln.

6.3.5 Control Hazards

Ein mindestens ebenso grosses Problem wie Data Hazards sind Control Hazards für eine Pipeline. Dieser Konflikt entsteht dadurch, dass erst nach einigen Pipelinestufen festgestellt werden kann, ob ein Branch verzweigt (*Branch Taken*) oder nicht (*Branch Untaken*). Bis das Ergebnis feststeht sind aber schon nachfolgend geholte Befehle in der Pipeline, die unter der impliziten Annahme branch untaken geholt wurden (die Pipeline wusste noch gar nichts von einem Branch). Wird aber ein taken branch erkannt, dann sind diese Befehle nicht nur falsch, sondern sie dürfen auch gar nicht ausgeführt werden, da sie Register verändern können, die dann von den Befehlen im taken branch verwendet werden. Einfachste Möglichkeit dies zu verhindern sind Stalls, die eingefügt werden, sobald die Pipeline erkennt, dass ein Branch–Befehl in der Pipeline ist.

Abb. 6.10 zeigt diese Methode zur Vermeidung eines Control Hazard für die einfache DLX Pipeline.

Takt	1	2	3	4	5	6	7	8	9	10
Befehl **Branch**	IF	ID	EX	MEM	WB					
Branch 02		IF	**stall**	**stall**	IF	ID	EX	MEM	WB	
Branch 03						IF	ID	EX	MEM	WB

Abbildung 6.10: Control Hazard Stalls in der DLX Pipeline.

Sobald feststeht, dass ein Branch–Befehl in der Pipeline ist (nach ID), werden zwei Stalls eingefügt, da erst nach der MEM–Stufe feststeht, ob und wohin gesprungen wird. Nach den beiden Stalls wird das IF für den Befehl nach dem Branch wiederholt, da beim ersten der falsche Befehl geholt worden sein kann, was einen *Branch Delay* von drei Taktzyklen zur Folge hat. Wenn man bedenkt, dass manche Programme bis zu 30% Branches aufweisen, dann ist klar, dass der Branch Delay eine massive Beeinträchtigung der Performance der Pipeline mit sich bringt.

In der DLX Pipeline wird der Branch zwar relativ rasch erkannt (in ID), allerdings würde uns das nur helfen, wenn wir auch wüssten ob und wohin gesprungen wird. Beides wird erst in MEM berechnet. Was wäre aber, wenn wir diese Berechnungen in die ID–Stufe vorziehen würden, denn dort wird auch schon das Register gelesen, das anzeigt, ob gesprungen wird. Auch wird in ID der Offset zum PC gelesen, der im Falle eines Sprungs zum PC addiert wird. Auch diese Massnahme kostet wieder an Hardware, da ein eigener Addierer für die Adressberechnung in ID verwendet werden muss. Ein Weiterverwenden der ALU für diese Berechnung würde unweigerlich zu einem Structure Hazard zwischen ID und EX führen.

Mit dieser Verbesserung muss nur noch ein Stall bei Erkennung eines Branch eingefügt werden. Während der Branch–Befehl in der ID–Stufe bearbeitet wird, wurde trotzdem schon der nächste Befehl (PC + 4) geholt, daher muss die IF–Stufe für den Befehl nach Branch wiederholt werden, was einem Stall von einem Taktzyklus entspricht. Mit einer kleinen weiteren Verbesserung könnten wir einen Teil dieser Stalls vermeiden (überlegen Sie wie). Durch das Vorziehen des Berechnens der Sprungadresse nach ID ergibt sich aber ein neues Problem. Generiert der Befehl vor dem Branch die Branchbedingung (in einem Register), dann tritt ein Data Hazard auf, der nur durch einen weiteren Stall behoben werden kann.

Die Grundidee weiterer Verbesserungen ist nun die, dass man versuchen könnte vorherzusagen, ob ein Branch tatsächlich verzweigt. Untersuchungen an Benchmark–Programmen zeigten allerdings, dass es erhebliche Unterschiede im Banchverhalten einzelner Programme gibt. Generell gilt aber, dass *Backward Branches* (Sprünge zu niedrigeren Adressen) häufiger verzweigen als *Forward Branches* (Hennessy and Patterson, 1996). Dies lässt sich dadurch erklären, dass Schleifen in Programmen meist als Backward Branch ausgeführt werden. In Zusammenarbeit mit dem Compiler lassen sich aber mit einfachen Vorhersagen über das Branchverhalten noch erhebliche Verbesserungen erzielen.

6.3.6 Static Scheduling

Diese Methoden versuchen mit einfachen Hardwareerweiterungen dem Compiler die Möglichkeit zu geben, den Code so anzuordnen, dass der *Branch Penalty* (Anzahl der Stalls für Branch–Befehle) minimiert wird. Dabei unterscheiden wir prinzipiell vier Methoden, deren Prinzip wir mit der bezüglich Branch verbesserten DLX Pipeline illustrieren wollen.

- *Freeze* Pipeline

  Dies entspricht der einfachsten Variante, die wir oben schon kennengelernt haben. Sobald ein Branch erkannt wird, wird die Abarbeitung in der Pipeline "eingefroren" bis der PC aktualisiert ist, und dann die Abarbeitung mit dem IF über den neuen PC fortgesetzt. Bei der DLX Pipeline kann der Compiler hier nur helfen, den möglichen Data Hazard vor dem Branch zu beseitigen.

- Predict–Not–Taken

  Mit dieser Variante wird von jedem Branch angenommen, dass er nicht verzweigt. D. h., die Abarbeitung wird beim nächsten Befehl (*Fall–Through*) fortgesetzt. Verzweigt der Branch tatsächlich nicht, dann ist kein Stall nötig. Andernfalls ist das Verhalten ident zum Freeze. Hier kann der Compiler die Pipeline unterstützen, indem er versucht, möglichst viele Branches so zu gestalten, dass sie meist nicht verzweigen.

- Predict–Taken

  Analog zum predict–not–taken wird hier angenommen, dass jeder Branch verzweigt. Dies wäre für die DLX Pipeline eine schlechte Idee, da zum Zeitpunkt der fertig berechneten Sprungadresse auch schon bekannt ist, ob gesprungen wird, und der fall–through schon in der Pipeline ist. Würde bei einem branch untaken jetzt der branch taken Befehl geholt (laut Annahme), würde der richtige Befehl mit einem falschen ersetzt werden, um dann zu bemerken, dass doch der andere geholt werden muss, was zu zwei Stalls führen würde. Bei einem branch taken ist die Vorhersage zwar richtig, bringt aber keinen Gewinn (warum?). Die Wahl der Richtung der Vorhersage hängt daher von der Organisation der Pipeline ab. Der Compiler kann auch hier die Branches wieder versuchen so einzustellen, dass die Vorhersage zutrifft.

- Delayed Branch

Wenn wir uns die simple Freeze–Technik vor Augen führen, ist klar, dass jeder Branch einen Stall auslöst, da wir nicht sicher sein können, ob der nächste Befehl in IF der richtige ist. Wenn wir aber einen Befehl direkt hinter den Branch plazieren, der unabhängig vom Branch bearbeitet werden kann, dann können wir den Stall vermeiden. Dem Compiler fällt also die wichtige Aufgabe zu, geeignete Befehle für den delay slot zu finden. Die Möglichkeiten dafür wollen wir noch etwas näher betrachten.

Beispiel: Wir betrachten ein Programmfragment eines Unterprogramms

```
      ADD R1, R2, R3
loop: SUB R4, R5, R6
      SUBI R10, R10, #1
      BEQZ R10, loop
      ADD R5, R8, R9
      OR R7, R8, R9
      JR R31
```

Unter der Voraussetzung, dass die Pipeline darauf vorbereitet ist, den Befehl nach dem Branch auf jeden Fall auszuführen, hat ein Compiler jetzt drei Möglichkeiten den delay slot zu füllen. In diesem Beispiel *muss* der Compiler den Code umordnen, da es sonst zu einem falsch ausgeführten Programm kommt (warum?).

Die erste Möglichkeit ist, einen Befehl vor der Schleife in den delay slot zu stellen.

```
loop: SUB R4, R5, R6
      SUBI R10, R10, #1
      BEQZ R10, loop
      ADD R1, R2, R3        ; delay slot filled from before
      ADD R5, R8, R9
      OR R7, R8, R9
      JR R31
```

Dies ist aber nur möglich, wenn es einen parallen Befehl vor der Schleife gibt. Existiert ein solcher, dann lässt sich zwar jeder Stall des vorhergehenden Branch–Befehls vermeiden, doch bei mehrmaligem Durchlaufen der Schleife wird der Befehl mehrmals ausgeführt. Dies stellt zwar kein Problem dar, doch ist dies dann eine nutzlose Operation.

Die nächste Möglichkeit ist dann die, dass der Compiler annimmt, dass branch taken eintritt und einen Befehl von der Sprungadresse in den delay slot stellt.

```
      ADD R1, R2, R3
loop: SUB R4, R5, R6
      SUBI R10, R10, #1
      BEQZ R10, loop + 4
      SUB R4, R5, R6        ; delay slot filled from target
      ADD R5, R8, R9
      OR R7, R8, R9
      JR R31
```

Wieder muss dieser Befehl parallel sein. Zusätzlich ist noch zu beachten, dass der Befehl in den delay slot *kopiert* werden sollte, da die Sprungadresse auch von einer anderen Stelle angesprungen werden kann. In unserem simplen Beispiel würde es aber genügen, den Befehl zu verschieben. Wird er aber kopiert, so kann auch die Sprungadresse optimiert werden (`loop + 4`). Verzweigt der Branch nicht, so wird zwar ein unnützer Befehl berechnet (ist effektiv ein Stall), was aber kein Problem darstellt, wenn er auch zu den folgenden Befehlen parallel ist.

Bleibt die Möglichkeit, den delay slot mit Befehlen aus dem fall–through zu besetzen.

```
      ADD R1, R2, R3
loop: SUB R4, R5, R6
      SUBI R10, R10, #1
      BEQZ R10, loop
      OR R7, R8, R9         ; delay slot filled from fall-through
      ADD R5, R8, R9
      JR R31
```

Hier *muss* der `OR`–Befehl vorgezogen werden, da bei einem taken branch `R5` verwendet wird, das im originalen Programmfragment im delay slot steht und somit ausgeführt werden würde. Dies stellt uns aber vor ein weiteres Problem, denn wenn wir keinen parallelen Befehl für den delay slot gefunden hätten, hätten wir nichts umordnen können. Wenn nun die Pipeline aber darauf vorbereitet ist, dass der Befehl im delay slot immer ausgeführt wird, dann könnten wir das Programmfragment gar nicht korrekt auf der DLX ausführen (es gibt meist Auswege, finden Sie einen). ◊

Eine bessere Lösung stellt ein *Cancelling Branch* dar (Abb. 6.11).

Hier wird der Befehl im delay slot nur dann ausgeführt, wenn der Branch in die vermutete Richtung verzweigt. Andernfalls wird der Befehl "eingefroren" und der nächste korrekte Befehl in die Pipeline geholt. Für die korrekte Bearbeitung unseres Beispiels bräuchten wir eine Pipeline mit *cancel if taken*,

Takt	1	2	3	4	5	6	7
Branch Taken	IF	ID	EX	MEM	WB		
Delay Slot		IF	ID	EX	MEM	WB	
Target 01			IF	ID	EX	MEM	WB

Takt	1	2	3	4	5	6	7
Branch Untaken	IF	ID	EX	MEM	WB		
Delay Slot		IF	**idle**	**idle**	**idle**	**idle**	
Fall-Through 01			IF	ID	EX	MEM	WB

Abbildung 6.11: Cancelling Branch (cancel if untaken) in der DLX Pipeline.

da dann der `ADD`–Befehl nach dem Branch nur bei branch untaken ausgeführt wird, was letztlich der Intention des Programms entspricht.

Oft implementieren Pipelines auch beide Varianten (regular and cancelling delay), was aber wieder erfordert, dass dies im Opcode des Branch codiert werden kann. Diese Technik erlaubt dem Compiler das effizienteste Scheduling, da er immer dann, wenn kein paralleler Befehl für den regular delay slot gefunden werden kann, einen nicht–parallelen Befehl im cancelling delay slot ausführen kann.

Beispiel: Die einfache DLX Pipeline mit getrennten Speichern für Instruktionen und Daten werde mit Hardware für delayed und cancelling branches ausgestattet (was genau benötigt man?). Benchmarkprogramme (Integer) werden mit einem Scheduler verbessert, indem zuerst die Load Delay Slots und dann die Branch Delay Slots gefüllt werden. Messungen ergeben, dass 5% aller Befehle zu einem Load Stall und 6% zu einem Branch Stall führen (der Compiler konnte also nicht alle Konflikte auflösen). Wie gross ist der Speedup dieser realen DLX–Pipeline im Vergleich zu einer DLX ohne Pipeline?

Wir kennzeichnen die CPI–Rate der realen Pipeline mit dem Index r und jene der DLX ohne Pipeline mit o. Dann ist

$$CPI_o = 5.0 \qquad CPI_r = 1.0 + 0.06 + 0.05 = 1.11.$$

Daraus folgt für den Speedup

$$s = \frac{CPI_o}{CPI_r} = \frac{5.0}{1.11} = 4.5$$

◊

Eine weitere Technik zur Eliminierung von branch delays ist die Eliminierung der Branches selbst. Diese Methode wird als *Loop Unrolling* bezeichnet und versucht Schleifen dadurch aufzulösen, indem der Code im Schleifenkörper einfach vervielfältigt wird. Auch dies ist nicht ganz einfach, da zur Compilezeit meist nicht bekannt ist, wie oft eine Schleife durchlaufen wird. Die prinzipielle Lösung dafür besteht darin, dass man die Schleife nicht ganz auflöst, sondern nur teilweise. Z. B. kann der Compiler den Schleifenkörper duplizieren (unter gewissen Voraussetzungen), die Schleifenvariable um jeweils zwei erhöhen und so die Hälfte der Branches einsparen (welches Problem haben wir dabei jetzt übersehen?).

Bei Maschinen mit grosser Anzahl von Pipelinestufen ist der branch delay üblicherweise grösser als ein Taktzyklus. Dadurch ist es dann für den Compiler schwer, genug parallele Befehle für die jetzt mehreren delay slots zu finden. Aus diesem Grund verzichten heute wieder viele Maschinen auf diese Technik und versuchen das Branchverhalten zur Laufzeit vorherzusagen. Je besser dies gelingt, desto eher können branch penalties vermieden werden.

6.3.7 Dynamic Branch Prediction

Branch History Buffers

Wenn wir die Vorhersage von Branches näher betrachten, dann ist klar, dass wir zwei Dinge vorhersagen könnten: die Branchbedingung und die Sprungadresse selbst. Das häufigste Mittel zur Vorhersage der Branchbedingung ist ein *Branch Prediction Buffer* (auch *Branch History Table*), der in seiner einfachsten Form aus einem einzigen Bit besteht. Dieses Bit gibt an, ob der letzte Branch verzweigte oder nicht. Mit diesem einzelnen Bit kann man allerdings schon viel erreichen.

Beispiel: Betrachten wir zwei Schleifen, die im Programm unmittelbar hintereinander ausgeführt werden. Die zweite Schleife wird neunmal durchlaufen und dann beendet. Wie gross ist die Vohersagegenauigkeit der zweiten Schleife, den wir mit dem 1–Bit–Buffer erreichen können?

Beim ersten Branch in der zweiten Schleife wird not taken vorhergesagt (warum?), was falsch ist. Dann werden acht Verzweigungen korrekt vorhergesagt, und der Schleifenabbruch wird nicht erkannt. Daraus folgt, dass von zehn Vorhersagen acht richtig waren, was einer Genauigkeit von 80% entspricht. ◊

Den Fehler beim Schleifenabbruch werden wir nicht vermeiden können, aber die Situation, dass dem Schleifeneintritt ein Austritt aus einer anderen Schleife vorhergeht, wird recht häufig eintreten. Wir benötigen daher

eine Lösung, die nicht nur den letzten Branch, sondern auch vorherige mitprotokolliert. Die einfachste Verbesserung ist dann ein 2–Bit–Buffer, dessen Vorhersage sich dann ändert, wenn zweimal hintereinander eine falsche Vorhersage abgegeben wurde. Für obiges Beispiel hiesse das, dass nach Austritt aus der ersten Schleife (falsche Vorhersage taken), die nächste Vorhersage immer noch taken lautet, was die Genauigkeit der Vorhersage auf 90% erhöhen würde.

Technisch könnte man das so realisieren, dass not taken mit 0 und taken mit 1 codiert wird und das MSB den vorletzten und LSB den letzten Branch repräsentieren. Dann ergeben sich für die 4 Zustände des 2–Bit Buffers die Vorhersagen 00 (not taken), 01 (not taken), 10 (taken), und 11 (taken). Interpretiert man die Zustände als Dezimalzahlen von 0 – 3, so erkennt man schnell, dass eine Zahl $z \geq 2$ eine taken–Vorhersage, und $z < 2$ eine not taken–Prognose codiert. Wird die Zahl bei taken inkrementiert und bei not taken dekrementiert, dann ergeben sich immer die richtigen Zustände (unter welcher Voraussetzung?)

Diese Erkenntnis kann man sehr einfach auf einen n–Bit Buffer übertragen, für den dann gilt, dass $z \geq 2^{n-1}$ taken und $z < 2^{n-1}$ not taken vorhersagt. Die Verbesserung, die mit solchen Prädiktoren zu erzielen ist, ist allerdings gering, weshalb man meist 2–Bit Buffer verwendet.

Es wäre allerdings zu schön, würden wir nur zwei einsame Bits benötigen, um Branchbedingungen gut vorherzusagen. In unserer Vereinfachung haben wir das Branchverhalten eines Branch global betrachtet. Sinnvoller ist es, für jeden Branch seine eigene Vorgeschichte zu protokollieren. Das erfordert aber, dass für jeden Branch jeweils 2 Bits zur Verfügung gestellt werden. Zusätzlich müssen wir den Branch aber auch identifizieren können. Eine eindeutige Identifizierung ist nur über die Adresse des Branch möglich, was aber bei einem 32–Bit System heissen würde, dass jeder Branch 32 Bit zur Identifikation und 2–Bit für seine Geschichte benötigen würde.

Daher verwendet man nur einige LSBs, um den Branch zu identifizieren, die man gleichzeitig zur Indizierung des branch prediction buffer verwenden kann. Nimmt man die 4 untersten Bits der Adresse, dann können wir 16 Einträge im Buffer adressieren. Es kann dann zwar passieren, dass wir die falsche Geschichte für einen Branch betrachten, doch muss das nicht bedeuten, dass die falsche Vorhersage getroffen wird. Durch Vergrösserung des Buffers kann man dieses Problem verkleinern (warum?).

Wenn wir unser letztes Beispiel noch einmal betrachten, dann ist es aber doch so, dass das Verhalten eines einzelnen Branches auch von der Vorgeschichte (eines anderen Branch) abhängt. Daher verwendet man auch *Correlating Predictors* (auch *Two–Level Predictors* genannt).

Der einfachste korrellierende Prädiktor besteht aus zwei Bits. Das erste

Bit steht z. B. für die Vorhersage unter der Bedingung, dass der vorherige Branch not taken war. Analog steht das zweite Bit für die Vorhersage unter der Bedingung taken. Dies ist also ein 1–Bit Prädiktor, der davon abhängt, was beim letzten (globalen) Branch geschah. War der letzte Branch taken, so sagt 0/1 taken vorher (was prognostiziert 0/0?).

Beispiel: Das Programmsegement

```
        ADDI R1, R0, #1     ; x = 1
        ADDI R2, R0, #0     ; y = 0
up:     BNEZ R1, next       ; b1 taken, if x != 0
        ADDI R2, R2, #1     ; ++y
next:   BNEZ R1, down     ; b2 taken, if x != 0
        ADDI R2, R2, #1     ; ++y
down:   XORI R1, R1, #1     ; flip x
        J up
```

ist zwar der "Tod" aller Prozessoren, aber wir können damit einen nichtkorrellierten mit einem korrellierten Prädiktor vergleichen.

Nehmen wir an, dass ein 1–Bit Prädiktor für jeden der Branches mit 0 (not taken) belegt sei, dann ergibt sich die Vorhersagetabelle 6.1 (für die ersten drei Jumps).

$b1_P$	$b1_A$	$b2_P$	$b2_A$
0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

Tabelle 6.1: Vorhersage (P) und Verhalten (A) der Branches $b1$ und $b2$ mit einem 1–Bit–Prädiktor.

Vergleichen wir obige Tabelle mit einem korrelierten 1–Bit–Prädiktor (linkes Bit ist Vorhersage unter der Bedingung not taken), der mit 0/0 belegt ist. Hier ist zu beachten, dass die Bedingung vom letzten globalen Sprung kommt, der hier eben der jeweils andere Branch ist. In Tabelle 6.2 beginnen wir mit einem letzten branch not taken.

Während der unkorrellierte Prädiktor alles inkorrekt vorhersagt, liegt der korrellierte nur bei den ersten beiden Vorhersagen daneben. ◊

Der korrellierte Prädiktor kann einfach zu einem (m, n)–Prädiktor verallgemeinert werden, wobei m die Anzahl der letzten (globalen) Sprünge ist, die Protokolliert werden, und n die Anzahl der Bits des Prädiktors. Die Anzahl der Einträge in den branch prediction buffer wird wieder einfach durch die

$b1_P$	$b1_A$	$b2_P$	$b2_A$
0/0	1	0/**0**	1
1/**0**	0	**0**/1	0
1/0	1	0/**1**	1
1/**0**	0	**0**/1	0

Tabelle 6.2: Vorhersage (fett) (P) und Verhalten (A) der Branches $b1$ und $b2$ mit einem korrellierten 1–Bit–Prädiktor.

Anzahl der Bits a', die man aus der Befehlsadresse ableitet, bestimmt (Fig. 6.12).

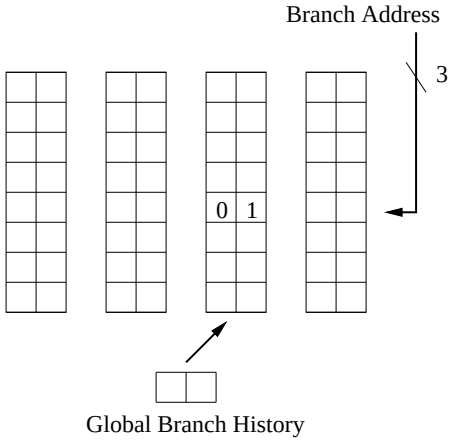

Abbildung 6.12: Korrellierender (2, 2)–Prädiktor mit acht Einträgen.

Die Gesamtanzahl der Bits n_B , die man für einen (m, n)–Prädiktor mit a' Einträgen benötigt, ergibt sich dann einfach zu

$$n_B = 2^m \times n \times 2^{a'}. \tag{6.2}$$

Der global history buffer lässt sich einfach mit einem Schieberegister implementieren. Viele Untersuchungen zeigten, dass man meist mit (0, 2)–Prädiktoren (unkorrelliert) oder (2, 2)–Prädiktoren ausreichend gute Ergebnisse erzielt. Selbst Prädiktoren, die ausschliesslich auf der globalen Geschichte beruhen ($a' = 0$) können gute Ergebnisse erzielen (z. B. ein (12, 2)–Prädiktor (Hennessy and Patterson, 1996)).

Wenn wir wieder einen Blick auf unsere einfache DLX–Pipeline werfen, dann erkennen wir, dass auch die (immer) korrekte Vorhersage eines Branch keine Verbesserung bringen würde, da Branchbedingung und Sprungadresse in der gleichen Stufe berechnet werden (ID). Selbst wenn wir in IF schon

wüssten, dass ein geholter Branch verzweigen wird, können wir nicht im nächsten Takt schon den richtigen Befehl holen, da die Sprungadresse noch nicht zur Verfügung steht. Um das Branchverhalten der DLX–Pipeline weiter zu verbessern müssen wir daher auch einen Weg finden, nicht nur die Branchbedingung, sondern auch die Sprungadresse vorherzusagen.

Branch Target Buffers

Eine einfache Idee, die aber wiederum einiges an Hardware kostet, ist, für jeden Branch die Sprungadresse in einem *Branch Target Buffer* zu speichern. Das ist insofern aufwendig, als jetzt ein exaktes Speichern der Branchadresse erforderlich ist. Würde man nur Teile dieser Adresse speichern, und damit Mehrdeutigkeiten erzeugen, dann würde die Vorhersagegenauigkeit stark beeinträchtigt werden, denn es ist sehr unwahrscheinlich, dass verschiedene Branches dasselbe Ziel haben. Das Prinzip besteht also einfach darin, eine Branchadresse zusammen mit der Sprungadresse abzuspeichern (Fig.6.13).

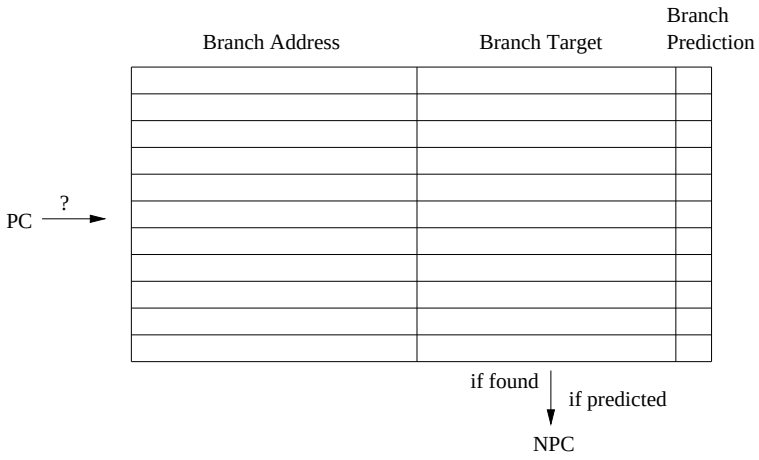

Abbildung 6.13: Branch Target Buffer mit (optionalem) Branch Prediction Buffer.

In der einfachsten Ausführung hat der Target Buffer nur Einträge für Branch–, und Sprungadresse. Immer wenn ein Befehl geholt wird, wird die Adresse im PC mit allen Branch Adressen im Target Buffer verglichen. Wird ein Eintrag gefunden, dann wird die zugehörige Sprungadresse in den PC geladen. Damit erspart man sich die explizite Decodierung des Befehls, da nur die Adressen von Branchbefehlen gespeichert sind. Weiters stellt dies implizit einen einfachen 1–Bit–Prädiktor dar, wenn die Sprungadresse immer nur dann gespeichert wird, wenn der Branch zuletzt auch ausgeführt wurde. Natürlich kann auch Speicher für komplexere Prädiktoren vorgesehen werden

(Fig. 6.13), womit sich eine Kombination aus Branch und Target Prediction realisieren lässt.

Wie kommen aber nun die Einträge in den Target Buffer? Zu Beginn eines ablaufenden Programms ist der Buffer leer. Immer dann, wenn ein Branchbefehl erkannt wird, der nicht im Buffer gefunden wurde, wird die Adresse des Befehls und dessen Sprungadresse eingetragen. Für den Fall eines Target Buffer ohne Branch Prediction wird dieses Adresspaar nur dann eingetragen, wenn der Branch verzweigte (impliziter 1–Bit–Prädiktor). Natürlich kann es hier auch zu falschen Vorhersagen kommen, wenn ein Branch im Buffer gefunden und die Sprungadresse geladen wurde, der Branch aber nicht verzweigt.

Für die einfache DLX–Pipeline bringt ein Branch Target Buffer, der in der IF–Stufe implementiert ist, einen möglichen Geschwindigkeitsgewinn (abhängig von der Vorhersagegenauigkeit). Im Idealfall steht jeder Branch korrekt im Target Buffer, was dazu führte, dass noch in der IF–Phase der PC den richtigen Wert für den nächsten Befehl enthält. Dies würde sämtliche Branch Stalls eliminieren. Allerdings ist auch zu bedenken, dass mit einer falschen Vorhersage die Probleme einer Pipeline mit Target Buffer grösser als die einer einfacheren Pipeline sind, da einige "Reparaturarbeiten" anfallen. In diesem Fall kann der Branch Penalty dann höher sein als für ein System ohne Branch Prediction Hardware. Alle möglichen Zustände der Vorhersage mit einem Target Buffer und die sich daraus ergebenden Branch Penalties sind in Abb. 6.14 gezeigt.

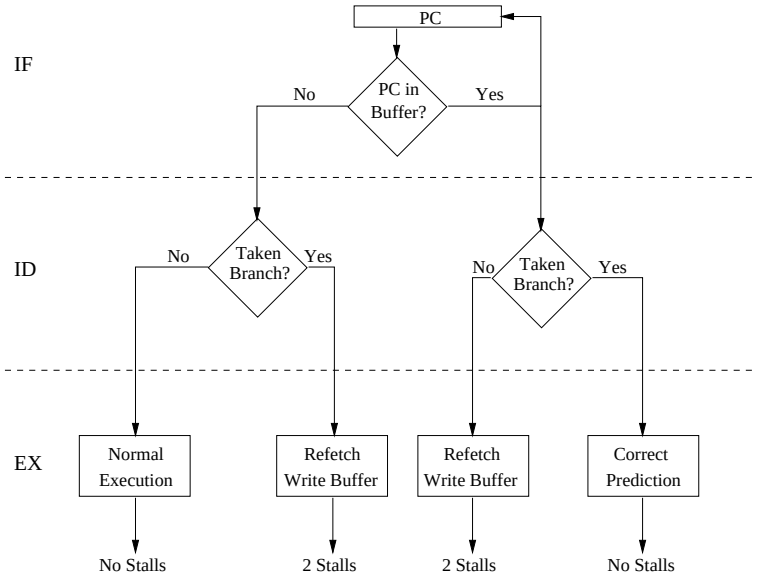

Abbildung 6.14: Operationen in der DLX Pipeline mit Branch Target Buffer.

Eine weitere Verbesserung ist möglich, wenn statt der Sprungadresse

gleich der Befehl an der Sprungadresse im Target Buffer steht. Wird der Branch richtig vorhergesagt, dann ist ein *Zero–Cycle–Branch* realisiert, da mit dem Holen des Branch–Befehls sofort der Befehl an der Sprungadresse zur Verfügung steht (*Branch Folding*). Heutige Prozessoren holen auch gleich mehrere Befehle von der Sprungadresse (gleichzeitig auch vom fall–through), um die Branch Penalties bei falscher Vorhersage zu minimieren.

Ein noch unbehandeltes Problem stellen *indirekte Sprünge* dar, wie sie zumeist bei Unterprogrammrücksprüngen (Returns) auftreten. Das Problem ist hier, dass eine Prozedur von mehreren Stellen im Programm aufgerufen werden kann. Es würden also für einen Branchbefehl mehrere Sprungadressen im Target Buffer gespeichert werden müssen. Aber auch dann müsste man entscheiden, welche von diesen Adressen im Augenblick die richtige ist. Eine Möglichkeit diese indirekten Sprünge vorherzusagen, ist ein *Return Stack*. Dies ist ein separater Speicher, der nur für indirekte Sprünge verwendet wird. Auf diesen Stack wird die aktuelle Rücksprungadresse gelegt. Da die Prozedurtiefe meist nicht grösser als 8 ist, ist hier mit wenig Zusatzhardware eine gute Verbesserung zu erzielen (mit einem kleinen Trick in einer DLX–Prozedur könnten Sie dieser Technik ein grosses Problem bereiten).

Kapitel 7

Interrupts

Ein weiterer Auslöser von möglichen Control Hazards sind *Interrupts* oder *Exceptions*. Beide Begriffe werden wahlweise verwendet, um denselben Sachverhalt zu beschreiben, doch definieren wir Interrupts hier als Unterbrechungsanforderung von externen Einheiten (z. B. I/O–Geräte) und Exceptions als eine Anforderung von internen Einheiten (Teile der CPU). Beides bewirkt eine Unterbrechung der normalen Programmarbeitung und veranlasst den Prozessor spezielle Prozeduren auszuführen, die feststellen welche Einheit die Unterbrechung angefordert hat, um dann die notwendigen Anweisungen auszuführen. Die einzelnen Prozeduren unterscheiden sich nicht von einem "normalen" Programm, doch werden sie eben nicht von der Software, sondern von der Hardware "aufgerufen".

Ein Mittelding sind *Software Exceptions* oder *Traps*, die durch spezielle Instruktionen in einem Programm ausgelöst werden. Sehr häufig werden diese Traps für Betriebsssystemaufrufe verwendet. Dies ermöglicht eine saubere Trennung von Anwenderprogrammen und Betriebssystem, die auch oft von zusätzlicher Hardware unterstützt wird. Durch einen Trap im Programm wird an eine fixe Adresse verzweigt, an der das Betriebssystem geladen ist. Diese Adresse muss nicht einmal im Hauptspeicher liegen, sondern kann auch einen *ROM*–Baustein (*Read–Only–Memory*) ansprechen, in dem das Betriebssytem liegt. Dadurch wird es möglich, Betriebssystemänderungen durch einfaches Austauschen des ROM–Bausteins vorzunehmen, ohne eine einzige Programmzeile in den Anwenderprogrammen ändern zu müssen.

Interrupts sind, obwohl sie den Programmablauf unterbrechen, ganz wesentlich für die Performance von Rechnersystemen verantwortlich. Ohne Interrupts müsste ein Programm (oder das Betriebssystem) laufend überprüfen, ob externe Einheiten aktiv werden wollen. Dieses Verfahren heisst *Polling* und wird mit folgendem Beispiel illustriert.

Beispiel: Auf der CPU eines PC läuft ein Programm, das eine Eingabe von

der Tastatur erwartet. Die Tastatur ist über eine serielle Schnittstelle mit dem PC verbunden. Das Statusregister der Schnittstelle hat eine Länge von einem Byte und liegt an Adresse 1000 im Hauptspeicher (memory–mapped I/O). Das Bit 2 (*Ready*) im Statusregister zeigt an, ob ein Datum (von der Tastatur) an der Schnittstelle bereit ist. Hat die Schnittstelle keine Interruptleitung (realistischere Annahme ist, dass die Leitung deaktiviert ist), so müssen wir die Schnittstelle ständig abfragen (pollen), ob sie bereit ist.

```
        ADDUI  R1, R0, #1000    ; Adresse Statusregister
poll:   LBU  R2, 0(R1)          ; Lade Statusregister
        ANDI  R2, R2, #4        ; Maskiere Ready--Bit aus
        BEQZ R2, poll           ; Datum von Tastatur?
        ........               ; hole Datum
```

Obwohl dieses Programmfragment sehr kurz ist, beschäftigt es die CPU ununterbrochen, solange keine Eingabe über die Tastatur erfolgt. Der Prozessor verbraucht seine ganze Leistung dafür, auf etwas zu warten, von dem nicht sicher ist, ob es überhaupt eintritt. Dies ist dann kein Problem, wenn es sich um ein Kleinsystem handelt, dass nur dafür gebaut wurde, Daten zu empfangen. Für heutige PCs wäre diese Lösung aber untragbar, da dort viele Prozesse (vom Betriebssystem verwaltet) laufen, die während dieser Wartezeit die CPU viel sinnvoller auslasten könnten. ◊

Wesentlich effizienter ist es, wenn die Schnittstelle selbständig ein Signal an den Prozessor abgibt, wenn ein Datum zur Verfügung steht. Dies wird mit einer Interruptleitung zur CPU ermöglicht. Nun hat aber ein Rechnersystem oft wesentlich mehr externe Einheiten, die Interrupts erzeugen können. Würde eine CPU nur einen Interrupteingang haben, könnte sie nicht unterscheiden woher der Interrupt kommt. Daher können heutige Systeme Interruptvektoren abgeben (auf mehreren Interruptleitungen), womit jede Einheit über ihren Vektor (binäre Zahl) identifizerbar ist.

Zusätzlich können Interrupts priorisiert werden, da ein Stromausfall viel schneller behandelt werden muss, als ein Drucker, der kein Papier mehr hat. Dies löst auch das Problem, was zu tun ist, wenn zwei Geräte gleichzeitig einen Interrupt auslösen, da dann einfach der mit der höheren Priorität zuerst behandelt wird. Natürlich ist für die Interruptlogik wieder eine eigene Hardwareeinheit notwendig (*Interrupt Controller*).

Wir können Interrupts und Exceptions nach mehreren Kriterien kategorisieren.

- Synchron/Asynchron

  Ist der Zeitpunkt für das Auftreten einer Unterbrechung definiert (i.e., kann nur zu bestimmten Zeitpunkten erfolgen), so nennt man diese

synchron. Im Prinzip gilt dies für alle Exceptions, während Interrupts asynchron auftreten. Beispiel für eine Exception ist ein *FP Error*, der immer nur nach einer FP Operation auftreten kann. Ein anderes Beispiel ist eine *Trace Exception*, bei der nach jedem Befehl unterbrochen wird, um die *Trace Routine* anzuspringen. Dies ist vor allem für *Debugger* sehr nützlich.

- Maskierbar/ Nicht Maskierbar

  Über die Interruptlogik lassen sich Interrupts maskieren (eigenes Statusregister in der CPU), d.h., diese Interrupts werden einfach nicht mehr beachtet. Dies wird häufig in Interruptprozeduren verwendet, um nicht durch andere Interrupts selbst unterbrochen zu werden (manche Programmierer vergessen dann bei Verlassen der Prozedur, die Interrupts wieder zu aktivieren).

- Innerhalb/Zwischen Befehlen

  Es erfordert weniger Hardwareaufwand, wenn nach Eintreffen eines Interrupts der bearbeitete Befehl noch zu Ende gebracht wird, und dann erst zur Interruptprozedur verzweigt wird. Allerdings kann es möglich sein, dass dies eine zu lange Antwortzeit bedingt. Dann muss die Hardware auch in der Lage sein, einen Befehl zu unterbrechen (und ihn dann wieder zu starten). In einer Pipeline könnten z. B. alle Befehle in der Pipeline noch abgearbeitet werden, um dann erst auf den Interrupt zu reagieren. Wenn dies zu langsam ist, dann müssen alle Befehle in der Pipeline beendet und nach dem Interrupt wieder neu geholt werden (aufwendigere Hardware).

- Abbruch/Unterbrechung

  Einfach wäre es, wenn bei Auftreten eines Interrupts das Programm einfach abgebrochen werden kann. Dies ist jedoch nur sehr selten der Fall, wie z. B. bei einem Hardwareproblem (Einheit ist fehlerhaft), oder bei einem Erkennen einer *Illegal Instruction* (wodurch könnte dies ausgelöst werden?). Meist muss aber das Programm genau dort fortgesetzt werden, wo es unterbrochen wurde, was auch erfordert, das alle Register, die eventuell von einer Interruptprozedur verwendet werden, wieder auf den ursprünglichen Wert zurückgesetzt werden (vom Programm oder hardwareunterstützt).

7.1 Exceptions in der DLX–Pipeline

Für die DLX betrachten wir nur Exceptions, die aber ausreichen die Probleme aufzuzeigen, die Unterbrechungen hervorrufen. Wir können den verschiedenen Pipeline–Stufen typische Exceptions zuordnen.

In der IF–Stufe findet ein Speicherzugriff statt, der einen *Page Fault, Bus Error* (Adresse exisitiert nicht), *Address Error* (nonaligned Zugriff), oder eine *Memory Protection Violation* (Bereich von Bestriebssystem/Hardware gesperrt) auslösen kann.

In der ID–Stufe kann eine *Illegal Opcode Exception* auftreten.

In der EX–Stufe löst ein *Arithmetic Error* (z. B. Division durch 0) eine Exception aus.

Die MEM–Stufe greift ebenso wie die IF–Stufe auf den Speicher zu, daher können dort dieselben Exceptions auftreten.

Die WB–Stufe kennt keine Unterbrechung (überlegen Sie die einzige Möglichkeit).

Bei Auftreten einer Exception muss die Pipelinehardware folgende Schritte ausführen:

1) Im nächsten Taktzyklus wird in IF ein `TRAP`–Befehl in das IR geschrieben.

2) Ab dem Befehl, der die Exception auslöste darf kein Befehl mehr irgendein Register beschreiben (warum?). Dies wird durch No Operations in den jeweiligen IRs der Pipelineregister erreicht.

3) Beim Ausführen des `TRAP`–Befehls wird an eine definierte Adresse *Exception Handling* gesprungen. Dabei muss der PC des Befehls gespeichert werden, der die Exception auslöste, um mit `RFE` (Return From Exception), die Programmausführung fortzusetzen (wenn dies sinnvoll ist).

Dies ist (wie zumeist) die prinzipielle Vorgangsweise, die aber noch durch ein paar Detaillösungen verbessert werden muss. Es könnte z. B. das Problem auftauchen, dass zwei Befehle, die hintereinander in der Pipeline sind, jeweils eine Exception auslösen. Wenn nun der zweite Befehl früher eine Exception auslöst als der erste Befehl (überlegen Sie Möglichkeiten dazu), dann würde der erste Befehl zu Ende bearbeitet werden, obwohl er fehlerhaft war und möglicherweise ein Register falsch beschrieben hat. Beim Fortführen des Programms würde aber jener Befehl wieder geholt, der die Exception auslöste (also der zweite) und angenommen, dass der vorige Befehl korrekt bearbeitet wurde.

Eine mögliche Lösung dafür ist das Protokollieren der Exceptions jedes einzelnen Befehls. Dazu muss ein Statusregister mitgeführt werden, das anzeigt, in welcher Stufe der Befehl eine Exception auslöste (theoretisch könnte er auch mehrere auslösen). Nur in der WB–Phase wird dann überprüft, ob der Befehl Exceptions ausgelöst hat. Da die WB–Phase des zweiten Befehls auf jeden Fall nach dem ersten Befehl ausgeführt wird, ist obiges Problem beseitigt. Zu bedenken ist aber, dass auf eine Exception dann nur mit einer gewissen Verzögerung reagiert wird, die für Exceptions tolerierbar ist, für Interrupts aber zu lang sein könnte.

Literaturverzeichnis

Ash, R. (1965). *Information Theory.* Wiley, New York, 1st edition.

Broy, M. (1993). *Rechnerstrukturen und maschinennahe Programierung.* Informatik, Eine grundlegende Einführung, Teil II. Springer, Berlin.

Carlson, A. B. (1975). *Communication Systems.* McGraw–Hill, Tokyo, 2nd edition.

Dworatschek, S. (1970). *Schaltalgebra und digitale Grundschaltungen.* de Gruyter, Berlin.

Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Francisco, 2nd edition.

Huffman, D. A. (1952). A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the IRE*, 40(10):1098–1101.

INFOTEC (1998). http://www.infotec.org/history.htm. WWW Repository, Association of Information Technology Professionals.

Küpfmüller, K. (1954). Die Entropie der deutschen Sprache. *FTZ*, 7(6):265–272.

Lochmann, D. (1995). *Digitale Nachrichtentechnik.* Verlag Technik, Berlin, 1st edition.

Mendelson, E. (1982). *Boolesche Algebra und Logische Schaltungen.* Schaum's Outline. McGraw–Hill, Hamburg.

Newald and Lindner (1985). Digitale Schaltwerke. Laborunterlagen, Institut für Datenverarbeitung, Technische Universität Wien.

Volkert, J. (1999). Digitale Rechenanlagen. Vorlesungsunterlagen, Institut für Technische Informatik und Telematik, Universität Linz.