Jörg Keller, Wolfram Schiffmann

Computersysteme I

Kurseinheit 1-4

mathematik und informatik

FernUniversität in Hagen

Vorwort

Allgemeines

Wir begrüßen Sie herzlich zum Kurs *01608 Computersysteme I*. Dieser Kurs führt Sie in die Grundlagen der Computer-Hardware ein: Schaltfunktionen, Schaltnetze, Speicher, Schaltwerke und Mikroprozessoren. Im Folgekurs *01609 Computersysteme II* werden Sie die Architektur von Hochleistungsprozessoren und –Speichersystemen kennenlernen. Die beiden Kurse können entweder beide in einem Semester oder in zwei aufeinanderfolgenden Semestern belegt werden, da der Kurs 01609 zeitversetzt beginnt. Beide Kurse werden in jedem Semester angeboten.

Die beiden Kurse 01608 und 01609 bilden im Bachelor-Studiengang Informatik das Modul *Computersysteme*, das mit einer Prüfungsklausur endet, die am Ende jedes Semesters stattfindet[1]. Das Modul Computersysteme ist mit 10 Leistungspunkten bewertet, was einem *durchschnittlichen* Arbeitsaufwand eines Studierenden von 300 Stunden entspricht. Zur Bearbeitung des Kurses 01608 müssen Sie also im Mittel mit 150 Stunden Aufwand rechnen. Wenn Sie mit dem Kurs gut zurecht kommen, ist der Aufwand vermutlich geringer. Wenn Sie Schwierigkeiten mit den Kursinhalten haben ist der Aufwand vermutlich höher.

Die Inhalte des Kurses sind unseres Erachtens mit dem Studium des Kurstextes zu erschließen. Allerdings hilft zum vertieften Verständnis, gerade bei schwierigen Kursteilen, die Konsultation von Sekundärliteratur, um einen etwas anderen Blickwinkel auf die gleiche Materie zu erhalten. Hierzu existiert eine Fülle von Lehrbüchern, eine kleine Auswahl zeigt das Literaturverzeichnis. Die meisten dieser Bücher sind in der Universitätsbibliothek der FernUniversität und anderer Universitäten verfügbar. Zur Ausleihe von Büchern konsultieren Sie bitte die Webseiten der Bibliothek.

Kursbeleger fragen oft, warum Studierende der Informatik Kenntnisse über Hardware erwerben sollen, auch wenn sie voraussichtlich nie Prozessoren oder Rechner entwerfen werden. Hierfür gibt es unseres Erachtens mehrere Gründe.

1. So selten sind hardware-nahe Tätigkeiten gar nicht! Eine der größten Branchen in Deutschland ist der Maschinen– und Automobilbau, und heutige Maschinen und Autos beziehen einen großen Teil ihrer Funktionalität aus sogenannten *eingebetteten Systemen*, d.h. in darin integrierten

[1] Zur Verwendung in anderen Studiengängen konsultieren Sie bitte Ihre studiengangsspezifischen Studien– und Prüfungsordnungen.

Hard– und Softwaresystemen. Bei der Entwicklung eingebetteter Systeme findet auch heute noch[2] die Software-Entwicklung sehr hardware-nah statt, so dass Kenntnisse der zugrunde liegenden Hardware und Maschinensprache notwendig sind.

2. Auch bei der klassischen Software-Entwicklung sind Kenntnisse der Hardware, auf der die Software ausgeführt werden soll, wichtig um die Leistung der Software zu steigern. Dies reicht vom Einstellen der korrekten Compiler-Switches bis zur Optimierung von Datenstrukturen in Bezug auf die Nutzung der Prozessor-Caches. Außerdem ist ein grundlegendes Verständnis der Bereiche, die an das eigene Arbeitsfeld angrenzen, stets nützlich, um über den „Tellerrand“ schauen zu können. Schließlich verschmelzen bei den zunehmend verwendeten rekonfigurierbaren Schaltungsbausteinen die Bereiche der Software- und Hardware-Entwicklung, so dass auch hier Kenntnisse beider Bereiche notwendig sind.

3. Viele Konzepte, die innerhalb ihres Anwendungsbereichs bei Computersystemen vorgestellt werden, sind in der gesamten Informatik wichtig. Als Beispiel sollen *endliche Automaten* genannt werden, die nicht nur als Zustandsmaschinen komplexer Schaltwerke sondern auch als Konzept zur Erkennung regulärer Sprachen in der theoretischen Informatik und als Steuerungsalgorithmen in der praktischen Informatik verwendet werden.

Der Kurs 01608 *Computersysteme I* besteht aus vier Kurseinheiten.

Kurseinheit 1 beschreibt Schaltfunktionen und ihre verschiedenen Darstellungen, speziell die Beschreibung durch Boole'sche Ausdrücke.

Kurseinheit 2 beschreibt Schaltnetze, die Schaltfunktionen in Hardware realisieren, sowie dafür geeignete Zahlendarstellungen.

Kurseinheit 3 beschreibt Speicher sowie Schaltwerke, d.h. Kombinationen von Speichern und Schaltnetzen.

Kurseinheit 4 beschreibt komplexe Schaltwerke, d.h. Schaltwerke in denen zwischen Daten– und Kontrollsignalen unterschieden wird, und erweitert diese zum Grundkonzept eines einfachen Prozessors.

Zu jeder Kurseinheit gibt es Einsendeaufgaben, die im 2-Wochenrhythmus zu bearbeiten sind. Die genauen Termine entnehmen Sie bitte den Seiten des Kurses in der LVU.

Die ersten beiden Kurseinheiten sind gemessen an ihrem Inhalt recht formal gehalten. Dies geschieht weder um Studierende zu verwirren noch ist es Selbstzweck. Rein textuelle Beschreibungen erklären zwar die häufigen Betriebsfälle einer Schaltung anschaulicher als Formeln, gleichzeitig bleiben oft Unklarheiten über das Verhalten der Schaltung in Ausnahmefällen. Solche Unklarheiten werden bei einer formalen Beschreibung vermieden, da dort in der Regel auffällt, wenn ein Fall fehlt, oder wenn verschiedene Fälle nicht disjunkt sind und damit die Beschreibung nicht widerspruchsfrei ist. Auch das Erlernen dieser Betrachtungsweise ist über den Bereich der Computersysteme hinaus wichtig, da auch Software-Projekte oft unter unentdeckten unvollständigen oder widersprüchlichen Anforderungen leiden. Schließlich erlaubt die formale Beschreibung auch

[2] Dies geschieht aus Ressourcen– und Effizienzgründen.

das Führen von Beweisen, und führt so über die bloße, „kochbuchartige“ Vermittlung von Fakten (die Tiefe des Carry-Chain-Addierers ist zum Beispiel linear zur Bitbreite der Eingaben) zu einem Verständnis warum dies so ist (der Übertrag läuft beim Carry-Chain-Addierer eventuell über alle Stellen). Schließlich übt das formale Vorgehen auch die aus den Kursen der Mathematik kommenden Methoden im Informatik-Umfeld.

Die Inhalte des Moduls Computersysteme werden in Kursen der Kataloge B2 und M2 vertieft. Prozessoren für eingebettete Rechensysteme werden im Kurs 01706 *Anwendungsorientierte Mikroprozessoren* behandelt. Der Entwurf von integrierten Schaltungen wird im Kurs 01721 *VLSI-Entwurfsalgorithmen* vermittelt. Arithmetische Schaltnetze und Schaltwerke werden in Kurs 01726 *Rechnerarithmetik* ausführlich behandelt. Praktische Aspekte von PC-Systemen werden im Kurs 01744 *PC–Technologie* vertieft.

Literatur

1. B. Becker, R. Drechsler, P. Molitor. Technische Informatik. Pearson Studium 2005.

2. K. Gotthardt. Aufgaben zur Informationstechnik Teil I. Logos 2003.

3. H. P. Gumm, M. Sommer. Einführung in die Informatik. 6. Auflage. Oldenbourg 2004.

4. G. Hotz. Einführung in die Informatik. Teubner 1990.

5. J. Keller, W. J. Paul. Hardware Design, 3. Auflage. Teubner 2005.

6. W. Oberschelp, G. Vossen. Rechneraufbau und Rechnerstrukturen. Oldenbourg 2000.

7. W. Schiffmann, R. Schmitz. Technische Informatik I+II. 5. Auflage, Springer 2005.

8. A. Tanenbaum. Computerarchitektur. 5. Auflage. Pearson 2005.

9. H.-D. Wuttke, K. Henke. Schaltsysteme. Pearson 2003.

Inhaltsverzeichnis

Kurseinheit 1

Schaltfunktionen und Boole'sche Ausdrücke

Kapitelinhalt

Zusammenfassung

Schaltfunktionen bilden die formale Grundlage von Schaltnetzen. In dieser Kurseinheit wird die Darstellung von Schaltfunktionen durch Boole'sche Ausdrücke behandelt. Hierbei wird Wert auf die Umwandlung von Boole'schen Ausdrücken gelegt, insbesondere das Finden von Minimalpolynomen.

Lernziele

Die Lernziele dieser Kurseinheit sind:

- Kenntnis der verschiedenen Darstellungen von Schaltfunktionen,
- Sicherheit im Umgang mit und Kenntnis der Eigenschaften von Boole'schen Ausdrücken,
- Fähigkeit zur Bestimmung der Kosten von Boole'schen Ausdrücken, speziell von Minimalpolynomen.

1.1 Vorbemerkungen

1.1.1 Mengen und Funktionen

In diesem Abschnitt befassen wir uns mit Funktionen auf endlichen Mengen. Diese Funktionen werden sich durch Schaltnetze berechnen lassen. Wir setzen ein allgemeines Verständnis von Mengen und Funktionen voraus, und werden hier einige speziellere Sachverhalte wiederholen. Wir bezeichnen[1] mit

$$\mathbf{N} = \{1, 2, 3, \ldots\} \text{ bzw. } \mathbf{N}_0 = \mathbf{N} \cup \{0\}$$

die *Menge der natürlichen Zahlen* (ohne bzw. mit Null) und mit $\mathbf{R}$ die *Menge der reellen Zahlen*. Mit Z_n bezeichnen wir die Menge der Zahlen von 0 bis $n-1$. Weitere Mengen bezeichnen wir mit Großbuchstaben, z.B. $M = \{1, 2, 5\}$. Die leere Menge bezeichnen wir mit $\emptyset$.

natürliche Zahlen
reelle Zahlen
leere Menge

Wir bezeichnen die Mächtigkeit einer Menge M mit $\#M$. Bei einer endlichen Menge ist $\#M \in \mathbf{N}_0$. Zum Beispiel ist $\#\{1, 2, 5\} = 3$. Zwei Mengen M und N sind gleichmächtig, wenn es eine Bijektion $f : M \to N$ gibt, d.h. wenn jedem Element von M eineindeutig ein Element aus N zugeordnet werden kann. Während diese Festlegung für endliche Mengen (bei denen man ja lediglich die Anzahlen vergleichen müsste) aufwändig erscheint, ist sie bei unendlichen Mengen angebracht. Eine unendliche Menge, die gleichmächtig wie $\mathbf{N}$ ist, heißt *abzählbar unendlich*, ansonsten *überabzählbar unendlich* . Zum Beispiel ist die Menge der reellen Zahlen überabzählbar unendlich, die Menge M der geraden natürlichen Zahlen ist hingegen abzählbar unendlich, da die Funktion $f : \mathbf{N} \to M$, $f(x) = 2x$ eine Bijektion zwischen diesen Mengen darstellt.

abzählbar unendlich
überabzählbar unendlich

Das *kartesische Produkt* $M \times N$ zweier Mengen M und N ist definiert als

kartesisches Produkt

$$M \times N = \{(a, b) \ : \ a \in M, b \in N\} \ .$$

Hierbei gilt für endliche Mengen M und N: $\#(M \times N) = \#M \cdot \#N$. Beispielsweise ist

$$Z_2 \times Z_3 = \{0, 1\} \times \{0, 1, 2\} = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)\} \ .$$

und

$$\#(Z_2 \times Z_3) = 6 = 2 \cdot 3 = \#Z_2 \cdot \#Z_3 \ .$$

In gleicher Weise kann man das kartesische Produkt aus $n \in \mathbf{N}$ Mengen $M_0, \ldots, M_{n-1}$ definieren. Sind alle Mengen $M_i = M$ identisch, dann schreibt man statt $M \times \cdots \times M$ auch M^n, und es gilt $\#M^n = (\#M)^n$. Beispielsweise ist

$$\begin{aligned} \{0, 1\}^3 &= \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), \\ &\quad (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\} \ , \\ \#\{0, 1\}^3 &= (\#\{0, 1\})^3 = 2^3 = 8 \ . \end{aligned}$$

[1] Die Sprechweisen der wichtigsten vorkommenden Notationen sind in einem Anhang angegeben.

Korrekterweise müsste man das n-fache kartesische Produkt induktiv definieren und hätte dann sehr viele Klammern zu setzen. Wir setzen allerdings nur die äußeren Klammern, d.h. statt

$$a = (\cdots(a_0, a_1), a_2)\cdots), a_{n-1})$$

schreiben wir

$$a = (a_0, a_1, a_2, \ldots, a_{n-1}) .$$

Folge

Wir nennen a eine *Folge der Länge* n, in Zeichen $l(a) = n$. Weiterhin vereinbaren wir, wenn $\cup$ die Vereinigung von Mengen bezeichnet, die Notation

$$A^+ = \bigcup_{i \in \mathbf{N}} A^i .$$

Alphabet
Zeichenreihe
leeres Wort

Eine endliche Menge A von Symbolen oder Zeichen nennen wir ein *Alphabet*. Die Folgen über A, d.h. die Elemente aus A^+ nennen wir *Zeichenreihen*.

Die eindeutige Zeichenreihe mit Länge 0 wird *das leere Wort* genannt und häufig mit ϵ abgekürzt. Für ein beliebiges Alphabet A bezeichnet man mit A^0 die Menge, die nur das leere Wort enthält, also

$$A^0 = \{\epsilon\} .$$

Man bezeichnet mit

$$A^* = A^+ \cup \{\epsilon\} = \bigcup_{i \in \mathbf{N}_0} A^i$$

die Menge aller Zeichenreihen mit Zeichen aus A einschließlich des leeren Worts.

Damit ist $\{0,1\}^* = \{\epsilon, 0, 1, (0,0), (0,1), (1,0), (1,1), \ldots\}$.

Für die Anzahl von Zeichenreihen der Länge höchstens n gilt, da die vereinigten Mengen disjunkt sind

$$\#\left(\bigcup_{i=0}^{n} A^i\right) = \#A^0 + \#A^1 + \cdots + \#A^n = \sum_{i=0}^{n} \#A^i = \sum_{i=0}^{n} (\#A)^i . \qquad (1.1)$$

Um die Summe der rechten Seite zu vereinfachen, benötigen wir das folgende Lemma 1.1.

Lemma 1.1 *Seien x und n zwei natürliche Zahlen mit $x \neq 1$. Dann gilt*

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1} .$$

Beweis: Sei $s = \sum_{i=0}^{n} x^i$. Dann gilt

$$x \cdot s = \sum_{i=0}^{n} x^i \cdot x = \sum_{i=1}^{n+1} x^i .$$

Weiterhin ist $(x-1) \cdot s = x \cdot s - s$. Wir setzen für s und für $x \cdot s$ jeweils die obigen Summenformeln ein und erhalten

$$(x-1) \cdot s = \sum_{i=1}^{n+1} x^i - \sum_{i=0}^{n} x^i = x^{n+1} - 1 .$$

Dividiert man die linke und rechte Seite der Gleichungskette durch $x - 1$, so erhält man die Behauptung. ∎

Den Ausdruck $\sum_{i=0}^{n} x^i = x^0 + x^1 + \cdots x^n$ nennt man *geometrische Reihe.*

Damit ist die Anzahl der Zeichenketten der Länge höchstens 2 über dem Alphabet $\{0,1\}$, also gerade der oben ausformulierte Teil der Menge $\{0,1\}^*$, $(2^3-1)/(2-1) = 7$. geometrische Reihe

Eine Folgerung aus Lemma 1.1 ist

$$\sum_{i=m}^{n-1} x^i = \frac{x^n - x^m}{x-1} \,. \tag{1.2}$$

wobei m eine natürliche Zahl kleiner als n sein soll. Zum Beispiel ist

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1 \text{ und } \sum_{i=m}^{n-1} 2^i = 2^n - 2^m \,.$$

Selbsttestaufgabe 1.1 *Beweisen Sie Gleichung (1.2).*

Lösung auf Seite 35

Eine *Funktion* kann nun ebenfalls über ein kartesisches Produkt definiert werden.

Definition 1.1 *Es seien X und Y Mengen. Eine* Funktion *f von X nach Y ist eine Teilmenge f von $X \times Y$, für die gilt: für jedes $x \in X$ gibt es genau ein $y \in Y$ mit $(x,y) \in f$. Dieses y heißt der* Funktionswert *von f an der Stelle x. Die Menge X heißt* Definitionsbereich *von f, die Menge Y heißt* Wertebereich *von f.* Funktion, Funktionswert, Definitionsbereich, Wertebereich

Statt $(x,y) \in f$ schreibt man gewöhnlich $f(x) = y$. Statt „$f \subseteq X \times Y$ ist eine Funktion“ schreibt man gewöhnlich $f : X \to Y$.

1.1.2 Gerichtete Graphen

Zusammenhänge zwischen Elementen einer Menge werden in der Informatik häufig durch Graphen dargestellt. Wir werden die wichtigsten Begriffe hier einführen.

Definition 1.2 *Ein* endlicher gerichteter Graph *wird spezifiziert durch ein Paar $G = (V,E)$. Hierbei gilt* gerichteter Graph

- *V ist eine endliche Menge. Die Elemente von V heißen die* Knoten *des Graphen.*
- $$E \subseteq V \times V = \{(u,v) \mid u,v \in V\} \,.$$

  Ein Element $(u,v) \in E$ heißt eine gerichtete Kante *von u nach v. Ist $(u,v) \in E$, so heißt u* direkter Vorgänger *von v und v heißt* direkter Nachfolger *von u.* Kante, Vorgänger, Nachfolger

Da V endlich ist, ist notwendig auch E endlich. Wir betrachten bis auf weiteres weder unendliche Graphen noch ungerichtete Graphen. Wir schreiben deshalb statt „endlicher gerichteter Graph“ meistens einfach „gerichteter

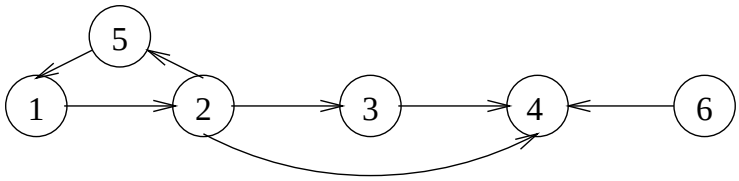


Abbildung 1.1: Beispiel eines Graphen

Graph" oder „Graph". Man zeichnet gerichtete Graphen, indem man die Knoten $v \in V$ als Kreise oder Punkte mit Beschriftung v und gerichtete Kanten (u, v) als Pfeile von u nach v malt.

Definition 1.3 *Sei $G = (V, E)$ ein gerichteter Graph. Eine Folge von Kanten*

$$e_i = (v_i, w_i) \in E, i = 1, \ldots, l \ ,$$

Pfad
Zyklus

heißt Pfad*, falls $v_{i+1} = w_i$ für $i = 1, \ldots, l-1$. Wir nennen l die* Länge *des Pfades. Gilt $w_l = v_1$, so heißt der Pfad* Zyklus*. Gerichtete Graphen, in denen es keine Zyklen gibt, heißen* zykelfrei.

Definition 1.4 *Sei $G = (V, E)$ ein gerichteter Graph. Für einen Knoten $v \in V$ ist der* Ingrad *$indeg(v)$ und der* Outgrad *$outdeg(v)$ definiert als*

Ingrad
Outgrad

$$\begin{aligned} indeg(v) &= \#\{u \mid (u, v) \in E\} \\ outdeg(v) &= \#\{u \mid (v, u) \in E\} \ . \end{aligned}$$

Quelle
Senke

Knoten v mit $indeg(v) = 0$ heißen Quellen *des Graphen. Knoten v mit $outdeg(v) = 0$ heißen* Senken *des Graphen. Für den gesamten Graphen definiert man*

$$\begin{aligned} indeg(G) &= \max\{indeg(v) \mid v \in V\} \\ outdeg(G) &= \max\{outdeg(v) \mid v \in V\} \end{aligned}$$

Beispiel 1.1 *Abbildung 1.1 zeigt einen gerichteten Graphen mit sechs Knoten und sieben Kanten. Es gibt einen Pfad der Länge 3 und einen Pfad der Länge 2 von Knoten 1 nach Knoten 4, es gibt einen Zyklus der Länge 3 aus den Kanten $(1, 2)$, $(2, 5)$, $(5, 1)$. Der Ingrad von Knoten 1 ist 1, der Outgrad von Knoten 2 ist 3. Knoten 4 ist eine Senke, Knoten 6 eine Quelle.*

Tiefe

Definition 1.5 *Sei $G = (V, E)$ ein gerichteter Graph und $v \in V$. Die* Tiefe *$\mathrm{T}(v)$ von v wird definiert als die Länge eines längsten Pfades von einer Quelle zu v, falls ein solcher längster Pfad existiert. Andernfalls ist die Tiefe von v nicht definiert.*

Wir fassen einzelne Knoten auf als (unechte) Pfade der Länge 0. Damit können wir die obige Definition noch auf die Quellen gerichteter Graphen ausdehnen und ihre Tiefe als 0 definieren. Die Tiefe von Knoten, die auf einem Zyklus liegen, ist nicht definiert. Man kann zeigen, dass in einem zyklenfreien Graphen jeder Knoten eine Tiefe hat. Für zykelfreie gerichtete Graphen $G = (V, E)$ können wir nun die *Tiefe* $\mathrm{T}(G)$ des Graphen G definieren als

$$\mathrm{T}(G) = \max\{\mathrm{T}(v) \mid v \in V\} \ .$$

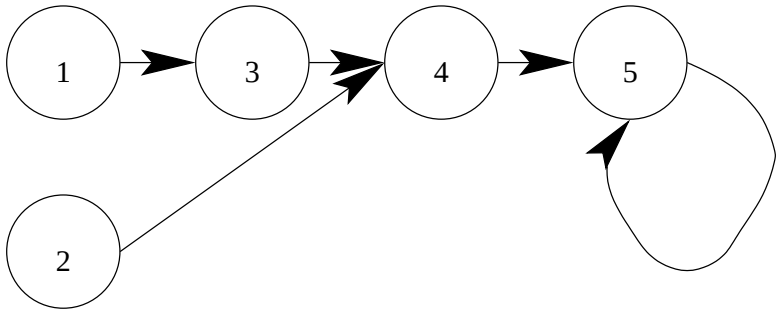


Abbildung 1.2: Zu analysierender Graph der Selbsttestaufg. 1.2

Beispiel 1.2 *In Abbildung 1.1 hat Knoten 6 als Quelle die Tiefe 0, und für Knoten 4 gilt* $T(4) = 1$*. Die anderen Knoten sind nicht von einer Quelle aus zu erreichen, ihre Tiefe ist nicht definiert.*

Selbsttestaufgabe 1.2 *Bestimmen Sie im Graphen aus Abbildung 1.2 die Quellen und die Senken. Bestimmen Sie für alle Knoten den Ingrad, den Outgrad und die Tiefe.*

Lösung auf Seite 35

Definition 1.6 *Ein* Baum *ist ein gerichteter zykelfreier Graph* G *für den gilt:* Baum

1. $outdeg(G) = 1$ *und*

2. G *hat genau eine Senke.*

Die Quellen eines Baums nennt man Blätter*, die Senke des Baums nennt man die* Wurzel *des Baums. Alle Knoten die keine Blätter sind heißen* innere Knoten*. Ein* binärer Baum *ist ein Baum, in dem alle Knoten außer den Quellen Ingrad höchstens 2 haben. Ein Baum ist* balanciert *wenn er minimale Tiefe hat und die Pfade von allen Blättern zur Wurzel sich in der Länge höchstens um 1 unterscheiden.*

Blatt eines Baumes
Wurzel eines Baumes
binärer Baum
balancierter Baum

Wir weisen darauf hin, dass man einen Baum ebenfalls definieren kann, wenn man die Rollen von Quellen und Senken sowie von Ingrad und Outgrad vertauscht.

Beispiel 1.3 *Der Graph aus Abbildung 1.3(a) ist ein binärer Baum mit Blättern* 1 *und* 4 *bis* 6 *und Wurzel* 0*. Die Graphen aus Abbildung 1.3(b) und (c) sind keine Bäume. Bei (b) gibt es zwei Senken, bei (c) hat ein Knoten Outgrad 2.*

Man kann zeigen, dass ein balancierter binärer Baum, bei dem die Pfade von Blättern zur Wurzel alle Länge n haben, aus 2^n Blättern und $2^n - 1$ inneren Knoten besteht (siehe Kurseinheit 2).

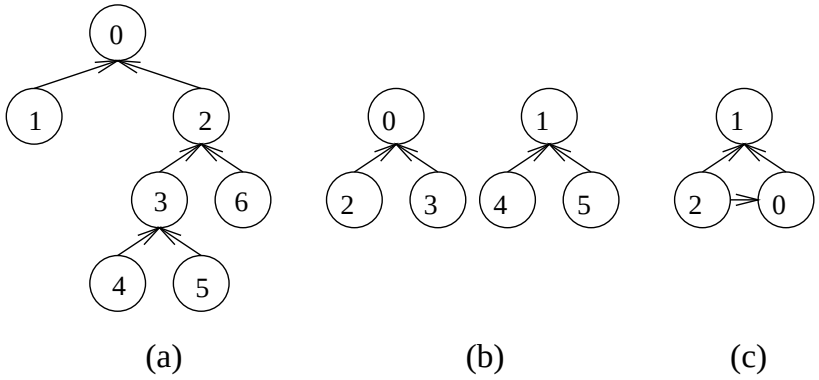


Abbildung 1.3: Beispiel und Gegenbeispiele für Bäume

1.2 Boole'sche Ausdrücke

Sei $n \in \mathbf{N}$. Wir interessieren uns für Schaltungen mit n Eingängen $X_1, \ldots, X_n$ und einem Ausgang Y. An jedem Eingang sowie am Ausgang sollen nur zwei Signale vorkommen können, die wir der Einfachheit halber mit 0 und 1 bezeichnen. Das Signal am Ausgang soll durch die Signale an den Eingängen eindeutig festgelegt sein. Das Ein–/Ausgabeverhalten einer solchen Schaltung lässt sich dann offensichtlich beschreiben durch eine Funktion $f : \{0,1\}^n \to \{0,1\}$, die jeder Kombination von Eingangssignalen das hierdurch festgelegte Ausgangssignal zuordnet. Dies motiviert die folgende Definition 1.7.

Schaltfunktion

Definition 1.7 *Sei $n \in \mathbf{N}$. Eine n-stellige* Schaltfunktion *ist eine Abbildung $f : \{0,1\}^n \to \{0,1\}$.*

Lemma 1.2 *Es gibt 2^{2^n} n-stellige Schaltfunktionen.*

Beweis: Die Mächtigkeit des Definitionsbereichs $Z_2^n = \{0,1\}^n$ ist $m = 2^n$. An jeder Stelle des Definitionsbereichs kann einer von zwei möglichen Funktionswerten genommen werden. Damit gibt es insgesamt, $2^m = 2^{2^n}$ Möglichkeiten, die Funktion zu definieren. ∎

Zum Beispiel beträgt die Anzahl der 5-stelligen Schaltfunktionen $2^{2^5} = 2^{32} \approx 4$ Milliarden. Zum Schätzen der Größenordnung von Zweierpotenzen benutzt man den Zusammenhang $2^{10} = 1024 \approx 10^3 = 1000$, siehe auch Kurseinheit 2.

Um eine Schaltfunktion jemandem anderen mitteilen zu können, muss man sie auf irgendeine Art und Weise darstellen. Hierzu sind die folgenden Methoden gebräuchlich, wobei die Liste keinen Anspruch auf Vollständigkeit erhebt:

- Wertetabellen,
- Karnaugh-Diagramme,
- Boole'sche Ausdrücke,
- Schaltnetz-Zeichnungen,

Tabelle 1.1: Wertetabelle einer 2-stelligen Schaltfunktion

x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1

- geordnete binäre Entscheidungsbäume (OBDDs).

Eine sehr allgemeine Art der Darstellung ist die *Wertetabelle*. Man schreibt in der linken Spalte alle Elemente des Definitionsbereichs untereinander, und in der rechten Spalte schreibt man neben jedes Element des Definitionsbereichs den zugehörigen Funktionswert. Während Wertetabellen für beliebige Funktionen anwendbar sind, sind die weiteren Darstellungen auf Schaltfunktionen spezialisiert. Tabelle 1.1 zeigt die Wertetabelle einer 2-stelligen Schaltfunktion, die gerade dann den Wert 1 annimmt, wenn beide Variablen den Wert 1 haben. Diese Schaltfunktion wird uns in Kürze unter dem Namen Konjunktion oder UND-Verknüpfung wiederbegegnen. Wertetabelle

Selbsttestaufgabe 1.3 *Bestimmen Sie die Anzahl der 2-stelligen Schaltfunktionen, und tragen Sie alle diese Funktionen in einer Wertetabelle zusammen. Welche dieser Schaltfunktionen können auch als 1-stellige Schaltfunktion des ersten Arguments betrachtet werden, da ihre Funktionswerte vom zweiten Argument unabhängig sind?*

Lösung auf Seite 35

Ein *Karnaugh-Diagramm*, das auch Karnaugh-Veitch-Diagramm oder KV-Diagramm genannt wird, ist anwendbar für $n \leq 4$ Variablen, es ist ein Rechteck mit 2^n Feldern, und einer Markierung der Seiten mit Variablenwerten, so dass jedes Feld eineindeutig einem Wert des Definitionsbereichs zugeordnet werden kann. In jedes Feld schreibt man den zugehörigen Funktionswert. Abbildung 1.4 zeigt ein Beispiel eines Karnaugh-Diagramms für eine 4-stellige Schaltfunktion. Abbildung 1.5 zeigt eine übliche verkürzende Schreibweise für Karnaugh-Diagramme, bei der nur die Spalten bzw. Zeilen mit einer Variablen markiert werden, bei denen die Variable den Wert 1 hat. Welche Variable an welcher Seite steht, ist eigentlich egal. Allerdings muss garantiert sein, dass sich von jeder Zeile zur nächsten und von jeder Spalte zur nächsten jeweils genau ein Variablenwert ändert, da ansonsten die Eindeutigkeit nicht gelten kann. Karnaugh-Diagramm

Selbsttestaufgabe 1.4 *Erstellen Sie ein Karnaugh-Diagramm für die Schaltfunktion aus Tabelle 1.1.*

Lösung auf Seite 36

Mit Boole'schen Ausdrücken wollen wir uns im Folgenden beschäftigen, so dass wir hier auf eine Erklärung verzichten. Gleiches gilt für Schaltnetz-Zeichnungen, die wir in Kurseinheit 2 besprechen werden.

	$X_1 = 1$	$X_1 = 1$	$X_1 = 0$	$X_1 = 0$	
$X_2 = 1$	1	1	1	0	$X_4 = 0$
$X_2 = 1$	1	1	1	1	$X_4 = 1$
$X_2 = 0$	1	1	1	0	$X_4 = 1$
$X_2 = 0$	0	1	0	0	$X_4 = 0$
	$X_3 = 0$	$X_3 = 1$	$X_3 = 1$	$X_3 = 0$	

$$f(X_1, X_2, X_3, X_4) = \begin{cases} 1 & \text{falls mindestens zwei der } X_i = 1, \\ 0 & \text{sonst} \end{cases}$$

Abbildung 1.4: Karnaugh-Diagramm für $n = 4$

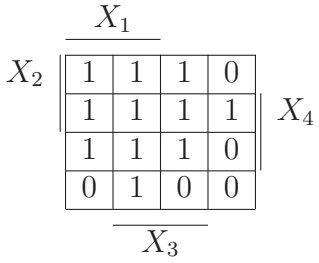


Abbildung 1.5: Vereinfachtes Karnaugh-Diagramm für $n = 4$

Die letzte Darstellung die wir kurz ansprechen wollen ist eine Darstellung als spezieller Graph, nämlich als *geordneter binärer Entscheidungsbaum* (ordered binary decision diagram, OBDD). Hierbei handelt es sich um einen balancierten binären Baum, d.h. um einen Baum, der von der Wurzel zu den Blättern hin gerichtet ist, bei dem jeder innere Knoten 2 Kinder hat, und bei denen alle Blätter die gleiche Tiefe haben. Alle inneren Knoten der gleichen Tiefe sind mit der gleichen Variable, und die je zwei Kanten die einen Knoten verlassen mit 0 und 1 markiert. Die Blätter sind mit den Funktionswerten markiert. Möchte man den Funktionswert an der Stelle $(a_1, \ldots, a_n)$ herausfinden, so startet man an der Wurzel, und bei jedem inneren Knoten folgt man der linken, mit 0 markierten Kante, falls die Variable X_i, mit der der Knoten markiert ist, den Wert $a_i = 0$ hat, sonst folgt man der rechten Kante.

Beispiel 1.4 *Abbildung 1.6 zeigt ein OBDD für die Schaltfunktion aus Tabelle 1.1. Um den Funktionswert an der Stelle $X_1X_2 = 01$ festzustellen, folgt man in der Wurzel der linken Kante, da die Wurzel mit X_1 markiert ist und $X_1 = 0$. In dem linken Knoten der Tiefe 1 folgt man der rechten Kante, da er mit X_2 markiert ist und $X_2 = 1$. Der Funktionswert beträgt 0.*

partiell definiert

Man kann eine Schaltfunktion auch nur auf einer Teilmenge von $\{0,1\}^n$ definieren, man nennt sie dann *partiell definiert*. In diesem Fall lässt man die betreffenden Zeilen in der Wertetabelle weg; im Karnaugh-Diagramm markiert man die Felder, die nicht zum Definitionsbereich gehören mit dem Symbol X.

Konjunktion
Disjunktion
Negation

Spezielle Schaltfunktionen sind die *Konjunktion* $\wedge : \{0,1\}^2 \to \{0,1\}$, die *Disjunktion* $\vee : \{0,1\}^2 \to \{0,1\}$ und die *Negation* $\sim : \{0,1\} \to \{0,1\}$. Ihre

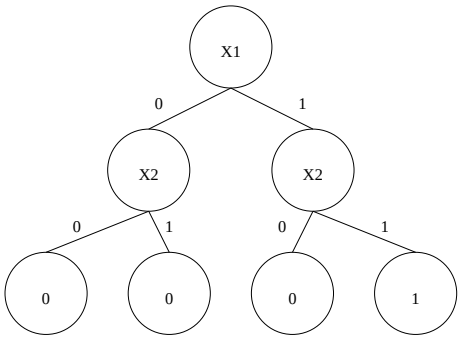


Abbildung 1.6: Geordneter binärer Entscheidungsbaum für eine 2-stellige Schaltfunktion

Tabelle 1.2: Wertetabellen von Konjunktion, Disjunktion und Negation

x_1	x_2	$x_1 \wedge x_2$	$x_1 \vee x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

x_1	$\sim x_1$
0	1
1	0

Wertetabellen sind in Tabelle 1.2 angegeben.

Offensichtlich gilt $\wedge(X_1, X_2) = X_1 \wedge X_2 = 1 \Leftrightarrow X_1 = 1$**und** $X_2 = 1$. Eine Schaltung mit diesem Ein-Ausgabeverhalten heißt deshalb AND-*Gatter*. Weiter gilt $\vee(X_1, X_2) = X_1 \vee X_2 = 1 \Leftrightarrow X_1 = 1$**oder** $X_2 = 1$. Eine Schaltung mit diesem Ein-Ausgabeverhalten heißt deshalb OR-*Gatter*. Schließlich gilt $\sim X_1 = 1 \Leftrightarrow X_1 \neq 1$. Eine Schaltung mit diesem Ein-Ausgabeverhalten heißt deshalb NOT-*Gatter* oder auch *Inverter*. Diese Gatter kann man in Halbleiterschaltungen leicht realisieren. Man benutzt sie als Bausteine zur Konstruktion von komplizierteren Schaltungen (s. Kurseinheit 2).

Gatter

Inverter

Sei nun $V = \{X_1, \ldots, X_n\}$ eine im Folgenden feste Menge von Variablen. Gelegentlich werden wir diese Variablen zu einem Vektor $X = (X_1, \ldots, X_n)$ zusammenfassen. Wir werden im Rest dieses Abschnitts folgendes tun:

1. Wir definieren mit Hilfe von $\wedge, \vee$ und $\sim$ die Menge der Boole'schen Ausdrücke mit Variablen in V. Die Menge $O_1 = \{\wedge, \vee, \sim\}$ heißt auch *Operatorensystem*.

   Operatorensystem

2. Wir ordnen jedem solchen Ausdruck e eine Schaltfunktion $f_e : \{0,1\}^n \to \{0,1\}$ zu. Die Funktion f_e heißt die durch Ausdruck e *berechnete Funktion*.

3. Wir geben Regeln für das Rechnen und Lösen von Gleichungen mit Boole'schen Ausdrücken an.

4. Ein Operatorensystem O heißt *vollständig*, wenn es zu jeder n-stelligen Schaltfunktion f einen Ausdruck e mit Operatoren aus O gibt mit $f = f_e$. Wir zeigen mit dem Darstellungssatz 1.6, daß es zu jeder n-stelligen

Schaltfunktion f einen Boole'schen Ausdruck e gibt mit $f = f_e$. Das gewählte Operatorensystem O_1 ist also vollständig.

Damit haben wir zweierlei erreicht. Zum einen besitzen wir Schaltfunktionen als *Spezifikation* des Ein–/Ausgabeverhaltens von Schaltnetzen. Zum anderen besitzen wir Boole'sche Ausdrücke, die als Beschreibungen oder Realisierungen einer Schaltfunktion dienen können, da sie alle Schaltfunktionen beschreiben können. Da es zu jeder Schaltfunktion mehrere Boole'sche Ausdrücke gibt, wird man je nach Einsatzzweck den einen oder anderen nehmen. Dies entspricht der Vorgehensweise in der Software-Entwicklung, wo man zunächst die Aufruf-Semantik einer Prozedur festlegt, und erst später entscheidet, wie die Prozedur intern realisiert wird, d.h. ob ein langsamer aber speicherplatzsparender Algorithmus verwendet wird oder ein schnellerer Algorithmus, der aber mehr Speicherplatz verbraucht.

1.2.1 Vollständig geklammerte Ausdrücke

Sei

$$A = \{0, 1, \wedge, \vee, \sim, X_1, \ldots, X_n, (,)\} .$$

Boole'scher Ausdruck

Die Menge B der *vollständig geklammerten Boole'schen Ausdrücke* ist eine Menge von Zeichenreihen in A^*. Sie wird auf folgende Weise induktiv definiert:

Definition 1.8 *Es ist $B_1 = \{0, 1, X_1, \ldots, X_n\}$. Sei $i \in \mathbf{N}$, und seien $a, b \in B_i$. Dann liegen folgende Zeichenreihen in B_{i+1}:*

1. a
2. $(\sim a)$
3. $(a \vee b)$
4. $(a \wedge b)$

Eine Zeichenreihe $z \in A^+$ liegt genau dann in B, wenn z in einer der Mengen B_i liegt, d.h. $B = \cup_{i \in \mathbf{N}} B_i$.

Beispiel 1.5 *Der Ausdruck $((0 \vee (\sim 1)) \wedge (X_2 \vee (\sim(\sim X_{52}))))$ ist ein vollständig geklammerter Boole'scher Ausdruck, denn es gilt*

$$\begin{aligned} 0, 1, X_2, X_{52} &\in B_1, \\ (\sim 1), (\sim X_{52}) &\in B_2, \\ (0 \vee (\sim 1)), (\sim(\sim X_{52})) &\in B_3, \\ (X_2 \vee (\sim(\sim X_{52}))) &\in B_4 \quad \textit{und} \\ ((0 \vee (\sim 1)) \wedge (X_2 \vee (\sim(\sim X_{52})))) &\in B_5. \end{aligned}$$

Die obige Definition hat für das praktische Rechnen einen entscheidenden Schönheitsfehler: wir rechnen nicht allein mit Boole'schen Ausdrücken, die nur mit den drei Funktionen $\wedge$, $\vee$ und $\sim$ gebildet sind. Vielmehr definieren wir in vielfältiger Weise neue Funktionen f und bilden dann mit Hilfe dieser Funktionen Ausdrücke wie zum Beispiel

$$X_1 \wedge f(X_2, 0, (X_1 \vee X_3)) .$$

Deshalb definieren wir nun die Menge EB der *erweiterten Boole'schen Ausdrücke*. Hierfür sei $F = \{f_1, f_2, \ldots\}$ eine abzählbare Menge von Funktionsnamen für Schaltfunktionen. Mit Hilfe der Funktion $s : F \to \mathbf{N}_0$ ordnen wir jeder Funktion $f \in F$ eine *Stelligkeit* zu, die einfach die Anzahl der Argumente von Funktion f angibt. Das unendliche Alphabet A wird definiert durch

erweiterter Boole'scher Ausdruck
Stelligkeit

$$A = V \cup F \cup \{0, 1, \wedge, \vee, \sim, (,), , \} .$$

Es enthält insbesondere das Komma. Die Menge EB der vollständig geklammerten *erweiterten Boole'schen Ausdrücke* wird nun induktiv definiert.

Definition 1.9 *Es ist* $\mathrm{EB}_1 = \{0, 1\} \cup V$. *Sei* $i \in \mathbf{N}$, *und seien* $e_1, e_2, \ldots \in \mathrm{EB}_i$. *Dann liegen folgende Zeichenreihen in* EB_{i+1}.

1. e_1
2. $(\sim e_1)$
3. $(e_1 \wedge e_2)$
4. $(e_1 \vee e_2)$
5. $f(e_1, \ldots, e_{s(f)})$ *für alle* $f \in F$.

Man beachte, dass ein Ausdruck, der sich in EB_i befindet, sich auch wegen der ersten Regel in $\mathrm{EB}_{i+1}, \mathrm{EB}_{i+2}, \ldots$ befindet. Daraus folgt, dass die verschiedenen EB_i nicht disjunkt sind. In der Praxis bedeutet es, dass man, um zu zeigen, dass zum Beispiel $(\sim e_1) \in \mathrm{EB}_{i+1}$ ist, nur zeigen muss, dass $e_1 \in \mathrm{EB}_j$ für irgendein $j \leq i$.

Beispiel 1.6 *Sei* $s(f) = 3$, *d.h.* f *bezeichnet eine 3-stellige Schaltfunktion. Dann ist*

- $X_1, X_2, X_3, 0 \in \mathrm{EB}_1$
- $(X_1 \vee X_3) \in \mathrm{EB}_2$
- $f(X_2, 0, (X_1 \vee X_3)) \in \mathrm{EB}_3$
- $(X_1 \wedge f(X_2, 0, (X_1 \vee X_3))) \in \mathrm{EB}_4$.

Für den späteren Gebrauch verabreden wir noch die Abkürzung $f(X)$ für $f(X_1, \ldots, X_{s(f)})$.

Selbsttestaufgabe 1.5 *Zeigen Sie, dass* $(((X_1 \wedge X_2) \wedge X_3) \vee f_1(X_1, X_2))$ *in* EB *liegt, wenn* $s(f_1) = 2$. *Wie müsste man den Ausdruck* $f_1(X_1, X_2) \vee (X_1 \wedge X_2 \vee X_3))$ *ergänzen, damit er in* EB *liegt?*

Lösung auf Seite 36

1.2.2 Einsetzungen

Jeder Boole'sche Ausdruck $e \in B$ kann als Vorschrift zur Berechnung einer Funktion $f_e : \{0,1\}^n \to \{0,1\}$ aufgefaßt werden: für $a = (a_1, \ldots, a_n) \in \{0,1\}^n$ berechnet man den Wert der Funktion f_e an der Stelle a, indem man für alle i die Konstante a_i für die Variable X_i einsetzt und dann auf die übliche Art auswertet. Die folgenden Definitionen formalisieren diese Vorgehensweise. Das mag manchem Leser überflüssig erscheinen, aber ohne präzise Definitionen kann man eben nichts beweisen.

Einsetzung

Definition 1.10 *Eine* Einsetzung *ist eine Abbildung* $\phi : V \to \{0,1\}$.

Für alle i ist $\phi(X_i) \in \{0,1\}$ gerade die Konstante, die für die Variable X_i eingesetzt werden soll. Durch eine Einsetzung ϕ ist bereits für *jeden* Boole'schen Ausdruck e der *Wert von* e *an der Stelle* $(\phi(X_1), \ldots, \phi(X_n))$ festgelegt. Wir nennen diesen Wert $\phi(e)$ und definieren ihn formal, indem wir induktiv die Funktion ϕ von $V \subseteq \mathrm{EB}$ auf die ganze Menge EB ausdehnen. Wir definieren $\phi(0) = 0$ und $\phi(1) = 1$. Damit ist ϕ auf EB_1 erklärt.

Definition 1.11 *Seien* $e_1, e_2, \ldots \in \mathrm{EB}$ *und* $f_i \in F$. *Wir definieren*

$$\begin{aligned}
\phi(\sim e_1) &= \sim\phi(e_1), \\
\phi((e_1 \wedge e_2)) &= \phi(e_1) \wedge \phi(e_2), \\
\phi((e_1 \vee e_2)) &= \phi(e_1) \vee \phi(e_2), \\
\phi(f_i(e_1, \ldots, e_{s(f_i)})) &= f_i(\phi(e_1), \ldots, \phi(e_{s(f_i)})) \text{ für alle } i \in \mathbf{N} .
\end{aligned}$$

Man beachte, daß wir hier die Bezeichner $\wedge$, $\vee$ und $\sim$ sowie f_i in zweifacher Weise verwendet haben. Auf der linken Seite stellen sie ein Zeichen in einem Boole'schen Ausdruck dar, auf der rechten Seite sind sie eine Aufforderung zum Auswerten von Funktionen.

Für die Funktionen '$\wedge$','$\vee$' und '$\sim$' enthält Tabelle 1.2 die Regeln zum Auswerten. Für irgendwelche weiteren Funktionen f_i muß man zuerst Auswertungsvorschriften festlegen, bevor man die obige Definition konkret anwenden kann.

Beispiel 1.7 *Sei* ϕ *eine Einsetzung mit* $\phi(X_1) = 1$, $\phi(X_2) = 0$ *und* $\phi(X_3) = 1$. *Für den Ausdruck* $((X_1 \wedge X_2) \vee X_3)$ *gilt dann* $\phi((X_1 \wedge X_2) \vee X_3) = \phi(X_1 \wedge X_2) \vee \phi(X_3)$. *Für den ersten Teilausdruck ergibt sich* $\phi(X_1 \wedge X_2) = \phi(X_1) \wedge \phi(X_2) = 1 \wedge 0 = 0$. *Damit folgt* $\phi((X_1 \wedge X_2) \vee X_3) = 0 \vee 1 = 1$.

Beispiel 1.8 *Die 3-stellige Schaltfunktion* f *sei durch die Funktionstabelle 1.3 definiert. Wie im vorigen Beispiel sei* ϕ *eine Einsetzung mit* $\phi(X_1) = 1$, $\phi(X_2) = 0$ *und* $\phi(X_3) = 1$. *Für den Ausdruck* $X_1 \wedge f(X_2, 0, (X_1 \vee X_3))$ *gilt dann*

$$\phi(X_1 \wedge f(X_2, 0, (X_1 \vee X_3))) = \phi(X_1) \wedge \phi(f(X_2, 0, (X_1 \vee X_3))) .$$

Für den zweiten Teilausdruck gilt

$$\phi(f(X_2, 0, (X_1 \vee X_3))) = f(\phi(X_2), \phi(0), \phi(X_1 \vee X_3)) .$$

Tabelle 1.3: Wertetabelle der Funktion f aus Beispiel 1.8

a_1	a_2	a_3	$f(a_1, a_2, a_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Da $\phi(X_1 \vee X_3) = \phi(X_1) \vee \phi(X_3) = 1 \vee 1 = 1$*, gilt für den zweiten Teilausdruck* $\phi(f(X_2, 0, (X_1 \vee X_3))) = f(0, 0, 1) = 0$*. Damit gilt für den gesamten Ausdruck*

$$\phi(X_1 \wedge f(X_2, 0, (X_1 \vee X_3))) = 1 \wedge 0 = 0 \; .$$

Ein subtiler Punkt ist an dieser Stelle die Tatsache, daß durch Definition 1.11 jedem Ausdruck e ein und *nur* ein Wert $\phi(e)$ zugewiesen wird. Hierfür muß man zeigen, daß es zu jedem vollständig geklammerten Ausdruck eine und *nur* eine Zerlegung in Teilausdrücke gibt, auf die man Definition 1.11 anwenden kann. Das ist im Wesentlichen der Inhalt des Zerlegungssatzes, den wir hier nicht ausführen wollen.

Selbsttestaufgabe 1.6 *Bestimmen Sie den Wert des Ausdrucks* $(((X_1 \wedge X_2) \wedge X_3) \vee f_1(X_1, X_2))$ *an den Stellen* $a = (1, 1, 1)$ *und* $b = (1, 0, 1)$*, d.h. für die Einsetzungen* ϕ_a *und* ϕ_b *mit* $\phi_a(X_1) = \phi_a(X_2) = \phi_a(X_3) = 1$ *und* $\phi_b(X_1) = \phi_b(X_3) = 1$*,* $\phi_b(X_2) = 0$*. Hierbei soll* $f_1(1, 0) = 1$ *und* $f_1(0, 0) = f_1(0, 1) = f_1(1, 1) = 0$ *gelten.*

Lösung auf Seite 36

1.2.3 Identitäten und Ungleichungen

Ausdrücke kann man nicht nur auswerten, man kann auch mit ihnen rechnen. Dabei verfolgt man meistens eine der zwei folgenden Aktivitäten:

1. man formt Ausdrücke äquivalent um oder
2. man löst Gleichungen.

Definition 1.12 *Es seien* $e_1, e_2 \in$ EB *erweiterte Boole'sche Ausdrücke. Es gilt* $e_1 \equiv e_2$ *genau dann, wenn* $\phi(e_1) = \phi(e_2)$ *für alle Einsetzungen* ϕ *gilt.*

Für $e_1 \equiv e_2$ sagt man auch „e_1 und e_2 sind *äquivalent*“, und man nennt die Zeichenreihe „$e_1 \equiv e_2$“ eine *Identität*. Identität

Beim konkreten Rechnen schreibt man häufig statt '$e_1 \equiv e_2$' einfach '$e_1 = e_2$' und sagt '$e_1 = e_2$ gilt identisch' oder noch einfacher 'e_1 gleich e_2'. Dabei mißhandelt man strikt gesprochen das Gleichheitszeichen, denn man setzt Ausdrücke

einander gleich, die als Zeichenreihen betrachtet in der Regel *nicht* gleich sind. Wir werden jedoch gelegentlich die Schreibweise '$e_1 \equiv e_2$' verwenden.

Satz 1.3 *Sei $e \in$ EB ein erweiterter Boole'scher Ausdruck. Dann gibt es genau eine n-stellige Schaltfunktion f so daß $f(X) \equiv e$ gilt.*

Die Funktion f mit $f(X) \equiv e$ heißt die durch Ausdruck e *berechnete Funktion.*

Beweis: Wir definieren zuerst die Funktion f. Für $a = (a_1, \ldots, a_n) \in \{0,1\}^n$ sei $\phi_a : V \to \{0,1\}$ die Einsetzung mit $\phi_a(X_i) = a_i$ für alle i. Damit $f(X) \equiv e$ gilt, muß $\phi(f(X)) = \phi(e)$ für alle Einsetzungen ϕ gelten, also insbesondere für $\phi = \phi_a$. Es folgt

$$\phi_a(e) = \phi_a(f(X)) = f(\phi_a(X_1), \ldots, \phi_a(X_n)) = f(a) \; .$$

Also ist f eindeutig bestimmt, und um den Wert der Funktion f an der Stelle a zu berechnen muß man einfach:

- für jede Variable X_i die Konstante a_i einsetzen (ϕ_a bilden) und dann
- auswerten ($\phi_a(e)$ bilden).

Sei nun $\phi : V \to \{0,1\}$ eine beliebige Einsetzung. Dann ist für $a = (\phi(X_1), \ldots, \phi(X_n))$ auch $\phi = \phi_a$. Es folgt

$$\phi(e) = \phi_a(e) = f(a) = \phi_a(f(X)) = \phi(f(X)) \; ,$$

also gilt $e \equiv f$. ∎

Beispiele für Identitäten liefert der folgende

Satz 1.4

(B1)	$(X_1 \wedge X_2) \equiv (X_2 \wedge X_1)$	*Kommutativität*
	$(X_1 \vee X_2) \equiv (X_2 \vee X_1)$	
(B2)	$((X_1 \vee X_2) \vee X_3) \equiv (X_1 \vee (X_2 \vee X_3))$	*Assoziativität*
	$((X_1 \wedge X_2) \wedge X_3) \equiv (X_1 \wedge (X_2 \wedge X_3))$	
(B3)	$(X_1 \wedge (X_2 \vee X_3)) \equiv ((X_1 \wedge X_2) \vee (X_1 \wedge X_3))$	*Distributivität*
	$(X_1 \vee (X_2 \wedge X_3)) \equiv ((X_1 \vee X_2) \wedge (X_1 \vee X_3))$	
(B4)	$(X_1 \vee (X_1 \wedge X_2)) \equiv X_1$	
	$(X_1 \wedge (X_1 \vee X_2)) \equiv X_1$	
(B5)	$(X_1 \vee (X_2 \wedge (\sim X_2))) \equiv X_1$	
	$(X_1 \wedge (X_2 \vee (\sim X_2))) \equiv X_1$	
(B6)	$(X_1 \vee (\sim X_1)) \equiv 1$	
	$(X_1 \wedge (\sim X_1)) \equiv 0$	
(B7)	$(X_1 \vee 1) \equiv 1$	
	$(X_1 \vee 0) \equiv X_1$	
	$(X_1 \wedge 1) \equiv X_1$	
	$(X_1 \wedge 0) \equiv 0$	
(B8)	$(\sim(X_1 \vee X_2)) \equiv ((\sim X_1) \wedge (\sim X_2))$	*Morgan-Formeln*
	$(\sim(X_1 \wedge X_2)) \equiv ((\sim X_1) \vee (\sim X_2))$	
(B9)	$(\sim(\sim X_1)) \equiv X_1$	
(B10)	$(X_1 \vee X_1) \equiv X_1$	
	$(X_1 \wedge X_1) \equiv X_1$	

Tabelle 1.4: Beweis von Identität (B6)

$\phi(e_1)$	$\phi(\sim e_1)$	$\phi((e_1 \vee (\sim e_1)))$
0	1	1
1	0	1

Man kann Satz 1.4 beweisen, indem man für jede der Identitäten ganz stur die höchstens acht verschiedenen Belegungen der vorkommenden Variablen aufzählt und für jede der Belegungen den Wert beider Seiten der Identitäten auswertet. Das kann man ganz schematisch in Tabellenform tun. Für die erste der Identitäten (B6) haben wir das in Tabelle 1.4 ausgeführt. Mit Hilfe der Identitäten aus Satz 1.4 kann man bis auf die vielen Klammern schon fast in gewohnter Weise rechnen. Mit Hilfe von (B3) kann man beispielsweise rechnen:

$$(X_7 \vee (X_1 \wedge (X_4 \wedge (1 \vee X_2)))) = (X_7 \vee (X_1 \wedge ((X_4 \wedge 1) \vee (X_4 \wedge X_2)))).$$

Hierbei haben wir zwei Dinge getan, nämlich:

1. Wir haben in (B3) die Variablen umbenannt und teilweise durch Konstanten ersetzt und
2. wir haben in einem Ausdruck einen Teilausdruck durch einen äquivalenten Ausdruck ersetzt.

In der Schule wurde beim Rechnen mit arithmetischen Ausdrücken die Regel 'Punktrechnung geht vor Strichrechnung' vereinbart. Der einzige Sinn dieser Regel ist das Sparen von Schreibarbeit, da man Ausdrücke nun nicht mehr vollständig klammern muß. Für Boole'sche Ausdrücke verabreden wir die Regeln

- $\sim$ bindet stärker als $\wedge$ und
- $\wedge$ bindet stärker als $\vee$.

Insbesondere behandeln wir also '$\vee$' wie '$+$' (Strichrechnung) und '$\wedge$' wie '$\cdot$'. (Punktrechnung). Nun können wir in gewohnter Weise Klammern weglassen.

Beispiel 1.9 *$X_1 \vee \sim X_2 \wedge X_3 \vee X_4$ ist Abkürzung für $((X_1 \vee ((\sim X_2) \wedge X_3)) \vee X_4)$.*

unvollständig geklammerter Ausdruck

Die *unvollständig geklammerten Ausdrücke*, die durch das Weglassen von Klammern entstehen, sind nichts weiter als Abkürzungen für die ursprünglichen — hoffentlich eindeutig rekonstruierbaren — vollständig geklammerten Ausdrücke. Eine strenge Beschreibung und Rechtfertigung dieses Vorgehens ist mit erheblichem Aufwand verbunden. Der interessierte Leser findet die entsprechenden Konstruktionen und Sätze zum Beispiel in Kapitel 1 von Keller/Paul: Hardware Design.

Wir vereinfachen die Schreibweise noch weiter. Ist e ein erweiterter Boole'scher Ausdruck, so schreibt man statt $\sim e$ oft $\bar{e}$.

Beispiel 1.10 *Statt $\sim(X_1 \wedge X_2)$ schreibt man oft $\overline{X_1 \wedge X_2}$.*

In arithmetischen Ausdrücken läßt man oft das Multiplikationszeichen '$\cdot$' weg. Ebenso läßt man in Boole'schen Ausdrücken oft das '$\wedge$' weg.

Beispiel 1.11 *Statt $X_1 \wedge X_2 \wedge \overline{X_3}$ schreibt man oft $X_1X_2\overline{X_3}$.*

Selbsttestaufgabe 1.7 *Nutzen Sie Regel (B3), um den Ausdruck $X_1(X_2 \vee X_3)$ in einen unvollständig geklammerten Ausdruck zu transformieren, in dem bei Beachtung der Punkt-vor-Strich-Regel überhaupt keine Klammern mehr notwendig sind. Nutzen Sie die Regeln (B6) und (B7) sowie falls notwendig weitere Regeln, um den Ausdruck X_1X_3 so zu transformieren, dass auch die Variable X_2 vorkommt, und keine Klammern notwendig sind.*

Lösung auf Seite 36

allgemeine Morgan–Formeln

Aus den Morgan-Formeln von Satz 1.4 kann man durch Induktion direkt die *allgemeinen Morgan-Formeln*

$$\begin{array}{rcl} \overline{X_1 \vee \cdots \vee X_n} & \equiv & \overline{X_1} \wedge \cdots \wedge \overline{X_n} \\ \overline{X_1 \wedge \cdots \wedge X_n} & \equiv & \overline{X_1} \vee \cdots \vee \overline{X_n} \end{array} \tag{1.3}$$

herleiten. Außerdem folgen aus Regeln (B3) und (B6) von Satz 1.4 die sogenannten *Resolutionsregeln*

$$\begin{array}{rcl} X_1X_3 \vee X_2\overline{X_3} & \equiv & X_1X_3 \vee X_2\overline{X_3} \vee X_1X_2 \\ (X_1 \vee X_3)(X_2 \vee \overline{X_3}) & \equiv & (X_1 \vee X_3)(X_2 \vee \overline{X_3})(X_1 \vee X_2) \end{array} \tag{1.4}$$

In Analogie zur Summennotation von arithmetischen Ausdrücken verabreden wir für erweiterte Boole'sche Ausdrücke $e_1, \ldots, e_m$ die Schreibweisen

$$\begin{array}{rcl} \bigwedge_{i=1}^{m} e_i & = & e_1 \wedge \ldots \wedge e_m, \\ \bigvee_{i=1}^{m} e_i & = & e_1 \vee \ldots \vee e_m \, . \end{array}$$

Für den Sonderfall, daß man das UND bzw. ODER von einer leeren Menge von Ausdrücken bildet, verabreden wir

$$\bigwedge_{i \in \emptyset} e_i = 1 \quad \text{und} \quad \bigvee_{i \in \emptyset} e_i = 0 \, . \tag{1.5}$$

Wir definieren die Relation $\leq$ auf der Menge $\{0, 1\}$ wie bei den natürlichen Zahlen, d.h. $0 \leq 0$, $0 \leq 1$, $1 \leq 1$, aber $1 \not\leq 0$. Wir erweitern diese Relation auf die erweiterten Boole'schen Ausdrücke.

Definition 1.13 *Es seien e_1 und e_2 erweiterte Boole'sche Ausdrücke. Es gilt $e_1 \leq e_2$ genau dann, wenn $\phi(e_1) \leq \phi(e_2)$ für alle Einsetzungen ϕ gilt.*

Aus der Definition schließt man unmittelbar für Ausdrücke $a, a', b, b' \in$ EB:

1. aus $a \leq b$ und $a' \leq b$ folgt $a \vee a' \leq b$ und

2. aus $a \leq b$ und $a' \leq b'$ folgt $a \vee a' \leq b \vee b'$.

Beispiel 1.12 *Es ist* $\wedge_{i=1}^{3} X_i = X_1 X_2 X_3$. *Durch Betrachten der Funktionstabelle 1.2 erkennt man, dass* $X_1 \wedge X_2 \leq X_1 \vee X_2$ *gilt.*

1.2.4 Lösen von Gleichungen

Das Gleichheitszeichen zwischen verschiedenen Ausdrücken e_1 und e_2 kommt außer beim äquivalenten Umformen noch in einem ganz anderen Zusammenhang vor, nämlich beim Lösen von Gleichungen.

Definition 1.14 *Eine* Gleichung *ist eine Zeichenreihe der Form '*$e_1 = e_2$*', wobei* e_1 *und* e_2 *beliebige Ausdrücke sein dürfen. Man* löst *eine Gleichung, indem man alle Einsetzungen* $\phi : V \to \{0, 1\}$ *bestimmt, so daß* $\phi(e_1) = \phi(e_2)$ *gilt.* Gleichung

Beispiel 1.13 *Die Gleichung* $X_1\overline{X_2} \vee \overline{X_1}X_2 = 1$ *hat zwei Lösungen, nämlich*

1. $\phi(X_1) = 1, \phi(X_2) = 0$ *und*
2. $\phi(X_1) = 0, \phi(X_2) = 1$.

Wir leiten einige Regeln zum Lösen von Gleichungen her. Es seien $e_1, \ldots, e_n$ vollständig geklammerte Boole'sche Ausdrücke und es sei ϕ eine Einsetzung. Aus Definition 1.11 und Tabelle 1.2 folgt direkt:

$$\begin{aligned} \phi((e_1 \wedge e_2)) = 1 &\Leftrightarrow \phi(e_1) \wedge \phi(e_2) = 1 \\ &\Leftrightarrow \phi(e_1) = 1 \text{ und } \phi(e_2) = 1 \, . \end{aligned}$$

Durch Induktion über n folgt:

$$\phi((e_1 \wedge \ldots \wedge e_n)) = 1 \Leftrightarrow \phi(e_i) = 1 \text{ für alle } i \in \{1, \ldots, n\}$$

Dem Leser wird auffallen, daß man eine Menge Schreibarbeit sparen kann, wenn man beim Gleichungslösen die ϕ's einfach wegfallen läßt. Aus dem Zusammenhang des Gleichungslösens geht dann hervor, daß man statt den Ausdrücken e in Wirklichkeit die Werte $\phi(e)$ meint. Das ist in der Tat gängige Praxis, der wir auch folgen werden. Nur bei ganz seltenen Anlässen muß man sich daran erinnern, daß man diese Vereinfachung vorgenommen hat. Insbesondere hätte man oben ohne Bezugnahme auf ϕ *nicht* folgern können:

$$(e_1 \wedge e_2) = 1 \Leftrightarrow e_1 = 1 \text{ und } e_2 = 1 \, .$$

Nach dem gleichen Muster beweist man das folgende Lemma. Es ist in der vereinfachten Form formuliert, aber für den Induktionsanfang der Beweise muß die Vereinfachung rückgängig gemacht werden.

Lemma 1.5 *Seien* $e_1, \ldots, e_n$ *vollständig geklammerte Boole'sche Ausdrücke. Dann gilt:*

1. $e_1 \wedge \ldots \wedge e_n = 1 \Leftrightarrow e_i = 1$ *für alle* $i \in \{1, \ldots, n\}$

2. $e_1 \wedge \ldots \wedge e_n = 0 \Leftrightarrow e_i = 0$ *für (mindestens) ein* $i \in \{1, \ldots, n\}$

3. $e_1 \vee \ldots \vee e_n = 1 \Leftrightarrow e_i = 1$ *für ein* $i \in \{1, \ldots, n\}$

4. $e_1 \vee \ldots \vee e_n = 0 \Leftrightarrow e_i = 0$ *für alle* $i \in \{1, \ldots, n\}$

5. $\overline{e_1} = 1 \Leftrightarrow e_1 = 0$

Beispiel 1.14 *Für den Ausdruck* $X_1(X_2 \vee X_3)$ *finden wir alle Einsetzungen, bei denen der Ausdruck den Wert 1 hat, d.h. wir lösen die Gleichung* $X_1(X_2 \vee X_3) = 1$. *Zumindest muss für jede solche Einsetzung* $\phi(X_1) = 1$ *gelten, denn ansonsten gilt wegen der zweiten Regel von Lemma 1.5, dass der Ausdruck den Wert 0 hat. Auch der Ausdruck in der Klammer muss den Wert 1 haben, was wegen der dritten Regel des Lemmas dann der Fall ist, wenn mindestens eine der Variablen* X_2 *und* X_3 *den Wert 1 hat. Also gibt es drei verschiedene solcher Einsetzungen:*
$\phi_1(X_1) = 1,\ \phi_1(X_2) = 1,\ \phi_1(X_3) = 0,$
$\phi_2(X_1) = 1,\ \phi_2(X_2) = 0,\ \phi_2(X_3) = 1,$
$\phi_3(X_1) = 1,\ \phi_3(X_2) = 1,\ \phi_3(X_3) = 1.$

Selbsttestaufgabe 1.8 *Zeigen Sie, dass der Ausdruck* $(\bar{X}_1 \vee \bar{X}_2) \wedge X_1 X_2$ *unter keiner Einsetzung den Wert 1 annehmen kann.*

Lösung auf Seite 36

1.2.5 Der Darstellungssatz

Der zentrale Satz dieses Abschnitts läßt sich nun sehr leicht herleiten. Für Variablen $X_i \in V$ und $\epsilon \in \{0, 1\}$ verabreden wir die Schreibweise

$$X_i^\epsilon = \begin{cases} \overline{X_i} \text{ falls } \epsilon = 0 \\ X_i \text{ falls } \epsilon = 1 \ . \end{cases}$$

Offensichtlich gilt

$$X_i^\epsilon = 1 \Leftrightarrow X_i = \epsilon \ .$$

Literal

Boole'sche Ausdrücke der Form X_i^ϵ nennt man *Literale.*

Definition 1.15 *Für* $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ *definieren wir die Boole'schen Ausdrücke* $m(a)$ *und* $c(a)$ *durch*

$$\begin{aligned} m(a) &= \bigwedge_{i=1}^{n} X_i^{a_i} \ , \\ c(a) &= \bigvee_{i=1}^{n} X_i^{\bar{a}_i} \ . \end{aligned}$$

Minterm
Maxterm

Der Ausdruck $m(a)$ *heißt der zu* a *gehörige* Minterm *und* $c(a)$ *der zu* a *gehörige* Maxterm.

Beispiel 1.15 *Es ist* $m(0, 1, 0) = \overline{X_1} X_2 \overline{X_3}$ *und* $c(0, 1, 0) = X_1 \vee \overline{X_2} \vee X_3$.

Aus Lemma 1.5 folgt

$$\begin{aligned} m(a) = \bigwedge_{i=1}^{n} X_i^{a_i} = 1 &\Leftrightarrow X_i^{a_i} = 1 \text{ für alle } i \in \{1,\ldots,n\} \\ &\Leftrightarrow X = (X_1,\ldots,X_n) = a \qquad (1.6) \\ c(a) = 0 &\Leftrightarrow X = a \qquad (1.7) \end{aligned}$$

Für n-stellige Schaltfunktionen f heißt die Menge

$$\mathrm{Tr}(f) = \{a \in \{0,1\}^n \mid f(a) = 1\}$$

der *Träger* von f. Offenbar ist

$$\mathrm{Tr}(f) = f^{-1}(1) \text{ und } \{0,1\}^n \setminus \mathrm{Tr}(f) = f^{-1}(0)\ .$$

Hierbei liefert f^{-1} die Urbilder der Funktion f, d.h. $f^{-1}(y)$ ist die Menge aller x für die $f(x) = y$ gilt. Es gilt

Träger

Satz 1.6 (Darstellungssatz) *Sei $f : \{0,1\}^n \to \{0,1\}$ eine Schaltfunktion. Dann gilt*

$$\begin{aligned} f(X) &\equiv \bigvee_{a \in \mathrm{Tr}(f)} m(a) \\ f(X) &\equiv \bigwedge_{a \notin \mathrm{Tr}(f)} c(a) \end{aligned}$$

Die erste Darstellung heißt die *kanonische disjunktive Normalform* von f, die zweite Darstellung die *kanonische konjunktive Normalform*. Der Ausdruck „kanonisch" rührt daher, dass diese Formen jeweils bis auf die Reihenfolge Min– bzw. Maxterme eindeutig sind.

kanonische disjunktive Normalform

kanonische konjunktive Normalform

Beispiel 1.16 *Sei f die in Tabelle 1.3 definierte Funktion. Dann gilt*

$$\begin{aligned} f(X) &\equiv \overline{X_1}X_2\overline{X_3} \vee X_1X_2\overline{X_3} \\ &\equiv (X_1 \vee X_2 \vee X_3) \wedge (X_1 \vee X_2 \vee \overline{X_3}) \wedge (X_1 \vee \overline{X_2} \vee \overline{X_3}) \\ &\quad \wedge (\overline{X_1} \vee X_2 \vee X_3) \wedge (\overline{X_1} \vee X_2 \vee \overline{X_3}) \wedge (\overline{X_1} \vee \overline{X_2} \vee \overline{X_3})\ . \end{aligned}$$

Beweis des Darstellungssatzes: Es gilt

$$\begin{aligned} \bigvee_{a \in \mathrm{Tr}(f)} m(a) = 1 &\Leftrightarrow m(b) = 1 \text{ für ein } b \in \mathrm{Tr}(f) \\ &\Leftrightarrow X = b \text{ für ein } b \in \mathrm{Tr}(f)\ . \end{aligned}$$

Behauptung 1 folgt nun direkt aus Lemma 1.5. Behauptung 2 beweist man ebenso. ∎

Selbsttestaufgabe 1.9 *Bestimmen Sie den Träger der Funktion $\vee$. Bestimmen Sie die zu den Elementen des Trägers gehörigen Minterme und die kanonische disjunktive Normalform von $\vee$.*

Lösung auf Seite 37

1.2.6 Kosten von Ausdrücken

Wir suchen im Folgenden sehr oft zu einer vorgegebenen Schaltfunktion f möglichst *einfache* Ausdrücke e, die f berechnen. Hierbei messen wir die Kompliziertheit eines Ausdrucks einfach durch die folgende Kostenfunktion.

Definition 1.16 *Sei $e \in B$ ein Boole'scher Ausdruck. Die* Kosten $L(e)$ *von e sind definiert als die Anzahl von Vorkommen der Zeichen $\wedge$, $\vee$ und $\sim$ in e.*

Beispiel 1.17 $L(X_1 \wedge \sim X_2 \wedge X_3) = 3$.

Die obige Definition scheint wörtlich genommen nur sinnvoll zu sein für Ausdrücke e, bei denen wir gewisse vereinfachte Schreibweisen nicht verwenden. Wir erinnern jedoch daran, daß für uns vereinfacht aufgeschriebene Ausdrükke ebenso wie unvollständig geklammerte Ausdrücke bloß Abkürzungen für vollständig geklammerte Ausdrücke aus B sind. Es ist deshalb

$$L(X_1\overline{X_2}X_3) = L(X_1 \wedge \sim X_2 \wedge X_3) = L((X_1 \wedge ((\sim X_2) \wedge X_3))) = 3\ .$$

Offenbar ist $L(X_i^\epsilon) \in \{0, 1\}$, d.h. Literale haben stets Kosten 0 oder 1. Sei nun f eine n-stellige Schaltfunktion. Jeder Minterm m der kanonischen disjunktiven Normalform von f besteht aus genau n Literalen und $n-1$ $\wedge$–Zeichen. Es folgt $L(m) \leq 2n-1$. Sei nun p die kanonische disjunktive Normalform von f. Dann besteht p aus genau $\#\mathrm{Tr}(f)$ Mintermen. Für die Anzahl v der $\vee$–Zeichen in p gilt

$$v = \begin{cases} \#\mathrm{Tr}(f) - 1 \text{ falls } \#\mathrm{Tr}(f) \geq 2 \\ 0 \text{ falls } \#\mathrm{Tr}(f) \leq 1 \end{cases}$$

Wegen $\#\mathrm{Tr}(f) \leq \#\{0,1\}^n = 2^n$ folgt

$$L(p) \leq n2^{n+1}\ .$$

Für jede n-stellige Schaltfunktion f gibt es also einen Boole'schen Ausdruck e mit Kosten höchstens $n2^{n+1}$, der f berechnet. Wir wären natürlich gern in der Lage, zu jeder vorgegebenen Schaltfunktion einen *billigsten* Ausdruck mit dieser Eigenschaft sowie seine Kosten zu bestimmen.

Definition 1.17 *Für Schaltfunktionen f heißt die Zahl*

$$L(f) = \min\{L(e) \mid e \in B, e \equiv f(X)\}$$

Formelgröße *die* Formelgröße *(engl. formula size) von f.*

Aus dem oben Gesagten folgt sofort

Satz 1.7 *Für jede n-stellige Schaltfunktion f gilt $L(f) \leq n2^{n+1}$.*

Genau genommen folgt aus der obigen Konstruktion $L(f) \leq n2^{n+1} - 1$. Allerdings ist die Aussage aus Satz 1.7 natürlich auch richtig. Sie ist außerdem besser zu merken und enthält die wichtige Information, dass man eine obere Schranke angeben kann, die exponentiell in der Anzahl n der Variablen ist und einen linearen Vorfaktor hat. Oft möchte man, wenn man das Wachstum von

Funktionen beschreibt, auch von konstanten Faktoren abstrahieren, da man an der „Größenordnung“ des Wachstums interessiert ist. Um den Begriff „größenordnungsmäßig“ formal zu fassen, führen wir Notationen für asymptotisches Wachstum ein.

Definition 1.18 *Seien $f, g : \mathbf{N} \to \mathbf{N}$ Funktionen. Wir sagen f* ist asymptotisch durch g beschränkt*, in Zeichen $f \leq_a g$, falls es ein $n_0 \in \mathbf{N}$ gibt mit $f(n) \leq g(n)$ für alle $n \geq n_0$. Wir definieren*

$$\begin{aligned} O(g) &= \{f \mid \exists k \in \mathbf{N} : f \leq_a k \cdot g\}, \\ \Omega(g) &= \{f \mid \exists k \in \mathbf{N} : g \leq_a k \cdot f\}, \\ \Theta(g) &= O(g) \cap \Omega(g) \quad . \end{aligned}$$

Normalerweise schreibt man $f = O(g)$ statt $f \in O(g)$.

Beispiel 1.18 *Es ist $3n^2 - 4n + 5 \in O(n^3)$, $e^n = \Omega(n^{10})$ und $4n^5 = \Theta(n^5)$. Allerdings ist $2n \notin O(\log_2 n)$.*

Damit läßt sich Satz 1.7 umformulieren zu:

Für jede n-stellige Schaltfunktion f gilt $L(f) = O(n2^n)$.

Selbsttestaufgabe 1.10 *Bestimmen Sie eine möglichst gute obere Schranke für die Formelgröße von $f(X_1, X_2, X_3) = X_1 \wedge (X_2 \vee X_3)$. Geben Sie auch die Schranke aus Satz 1.7 an.*

Lösung auf Seite 37

1.3 Minimalpolynome

1.3.1 Polynome und Primimplikanten

Wir untersuchen im Folgenden besonders einfache Mengen von Boole'schen Ausdrücken, nämlich die sogenannten *Boole'schen Polynome* und die *konjunktiven Normalformen.*

Definition 1.19

- *Ein* Literal *ist ein Ausdruck der Form X_i^ϵ mit $X_i \in V$ und $\epsilon \in \{0, 1\}$.* — Literal
- *Ein* Monom *oder* Konjunktionsterm *ist ein Ausdruck der Form $\bigwedge_{i \in I} L_i$, wobei die L_i Literale sind für alle i in einer endlichen Indexmenge I.* — Monom, Konjunktionsterm
- *Ein (Boole'sches)* Polynom *oder* disjunktive Normalform (DNF) *ist ein Ausdruck der Form $\bigvee_{i \in I} M_i$, wobei die M_i Monome sind für alle i in einer endlichen Indexmenge I.* — Boole'sches Polynom, disjunktive Normalform
- *Eine* Klausel *oder* Disjunktionsterm *ist ein Ausdruck der Form $\bigvee_{i \in I} L_i$, wobei die L_i Literale sind für alle i in einer endlichen Indexmenge I.* — Klausel, Disjunktionsterm
- *Eine* konjunktive Normalform *(KNF) ist ein Ausdruck der Form $\bigwedge_{i \in I} C_i$, wobei die C_i Klauseln sind für alle i in einer endlichen Indexmenge I.* — konjunktive Normalform

Alle Minterme sind Monome, und alle Maxterme sind Klauseln. Jede kanonische disjunktive Normalform ist ein Polynom, und jede kanonische konjunktive Normalform ist eine konjunktive Normalform.

In den obigen Definitionen sind auch leere Indexmengen I erlaubt. Es folgt, daß 0 sowohl ein Polynom und als auch eine Klausel ist, und daß 1 sowohl ein Monom als auch eine konjunktive Normalform ist.

Selbsttestaufgabe 1.11 *Ist der Ausdruck $X_1(X_2 \vee X_3)$ eine disjunktive oder konjunktive Normalform? Falls nicht, formen Sie ihn um.*

Lösung auf Seite 37

Naturgemäß interessiert man sich zu einer vorgegebenen Schaltfunktion f für *billigste* Polynome p, die f berechnen.

Definition 1.20 *Sei f eine Schaltfunktion und p ein Boole'sches Polynom. Dann heißt p ein* Minimalpolynom *oder* kürzeste disjunktive Normalform *von f, falls die folgenden beiden Bedingungen gelten:*

1. *$p \equiv f(X)$, d.h. p berechnet f.*
2. *$L(p) = \min\{L(q) \mid q$ ist Boole'sches Polynom und $q \equiv f(X)\}$, d.h. p ist ein billigstes Polynom mit dieser Eigenschaft.*

Wir werden im Folgenden zu vorgegebener Funktion f die Monome, die in Minimalpolynomen von f auftreten können, charakterisieren, und wir werden angeben, wie man diese Monome finden kann. Im unmittelbaren Anschluß daran stoßen wir schon auf das mit Abstand berühmteste offene Problem der Informatik.

Teilmonom

Definition 1.21 *Seien m und m' Monome. Dann heißt m'* Teilmonom *von m, falls die folgenden beiden Bedingungen gelten:*

1. *jedes Literal in m' kommt auch in m vor, oder $m' = 1$;*
2. *in m kommt mindestens ein Literal vor, das nicht in m' vorkommt.*

Beispiel 1.19 *Die Monome X_1X_4, 1 und $X_2\overline{X_3}$ sind Teilmonome des Monoms $X_1X_2\overline{X_3}X_4$, die Monome $X_1X_2X_3X_4$ und $X_1X_2\overline{X_3}X_4$ hingegen nicht.*

Lemma 1.8 *Es sei m' Teilmonom von m. Dann gilt $m \leq m'$.*

Beweis: Falls $m' = 1$ dann ist $m \leq m'$ offensichtlich wegen $m \leq 1$. Es sei also $m' = \bigwedge_{i \in J} L_i$, wobei die L_i Literale sind und J eine nicht-leere Indexmenge. Dann ist $m = \bigwedge_{i \in I} L_i$, wobei $I \supset J$, da jedes Literal aus m' nach Definition auch in m enthalten ist. Auch können wir $I = J$ ausschließen, da es nach Definition mindestens ein Literal in m geben muss, das nicht in m' enthalten ist. Wir betrachten im folgenden nur den Fall dass m bei einer Variablenbelegung den Wert 1 annimmt, denn wenn es den Wert 0 annimmt ist offensichtlich $m \leq m'$ wegen $m' \geq 0$. Aus Lemma 1.5 folgt dann unter Berücksichtigung von $J \subset I$:

$$\begin{aligned} m = 1 &\Leftrightarrow L_i = 1 \text{ für alle } i \in I \\ &\Rightarrow L_i = 1 \text{ für alle } i \in J \\ &\Leftrightarrow m' = 1\ . \end{aligned}$$

Also gilt auch in dem Fall, dass m bei einer Variablenbelegung den Wert 1 annimmt, $m \leq m'$. ∎

Beispiel 1.20 *Die Monome $m'_1 = X_1X_4$, $m'_2 = 1$ und $m'_3 = X_2\overline{X_3}$ sind Teilmonome von $m = X_1X_2\overline{X_3}X_4$. Das Monom $m = X_1X_2\overline{X_3}X_4$ nimmt den Wert 1 nur an der Stelle $X_1X_2X_3X_4 = 1101$ an. Sonst nimmt es den Wert Null an. An der Stelle 1101 haben auch die Teilmonome den Wert 1. Damit gilt für jedes Teilmomom m'_i: $m \leq m'_i$. An der Stelle $X_1X_2X_3X_4 = 1111$ haben die ersten beiden Teilmonome den Wert 1 aber m den Wert 0, an der Stelle 0100 haben die Teilmonome m'_2 und m'_3 den Wert 1, $m$aber nicht*

Definition 1.22 *Sei f eine Schaltfunktion und m ein Monom. Dann heißt m ein* Implikant *von f falls $m \leq f(X)$ gilt, d.h. falls aus $m = 1$ auch $f(X) = 1$ folgt. Ein Implikant von f heißt ein* Primimplikant *oder* Primterm *von f falls kein Teilmonom von m Implikant von f ist.*

Implikant

Primimplikant

Die konstante Funktion f mit $f(a) = 0$ für alle a hat nur ein Minimalpolynom, nämlich 0. Für alle anderen Schaltfunktionen f werden die Implikanten, die in Minimalpolynomen von f vorkommen können, charakterisiert durch

Satz 1.9 *Es sei f eine Schaltfunktion, und f sei nicht identisch gleich 0. Es sei p ein Minimalpolynom von f. Dann besteht p nur aus Primimplikanten von f.*

Beweis: Ist f identisch gleich 1, so hat f nur ein Minimalpolynom, nämlich 1, und der Satz gilt offensichtlich. Andernfalls gilt

$$f(X) \equiv p = m_1 \vee \ldots \vee m_s$$

für ein $s \in \mathbf{N}$ und Monome m_i, $i \in \{1, \ldots, s\}$. Jedes der Monome m_i ist ein Implikant von f, da es sonst eine Einsetzung ϕ gäbe mit $\phi(f) = 0$, aber $1 = \phi(m_i) = \phi(p)$.

Wir nehmen nun an, daß mindestens eins der Monome m_i kein Primimplikant von f ist. Ohne Beschränkung der Allgemeinheit können wir $i = 1$ annehmen (sonst numerieren wir die Monome um.) Sei nun m'_1 Teilmonom von m_1 und Implikant von f. Wir bilden das Polynom p', indem wir in p das Monom m_1 durch das billigere Monom m'_1 ersetzen:

$$p' = m'_1 \vee m_2 \vee \ldots \vee m_s \; .$$

Offensichtlich ist dann $L(p') < L(p)$. Wegen Lemma 1.8 ist $m_1 \leq m'_1$ und deshalb $p \leq p'$. Andererseits gilt $m'_1 \leq f$ und $m_i \leq f$ für alle i, denn sowohl m'_1 als auch alle m_i sind Implikanten von f. Es folgt $p' \leq f \equiv p$. Es folgt $p' \equiv p \equiv f$. Also war p kein Minimalpolynom von f. ∎

Selbsttestaufgabe 1.12 *Sind $X_1\bar{X}_2$, $\bar{X}_1X_2$ und X_1X_2 Implikanten der Schaltfunktion $f : \{0,1\}^2 \to \{0,1\}$, die den Wert 1 genau dann annimmt, wenn genau eines ihrer Argumente den Wert 1 hat? Falls ja, sind es Primimplikanten?*

Lösung auf Seite 37

1.3.2 Bestimmung von Minimalpolynomen

Um ein Minimalpolynom zu bestimmen, gibt es eine Reihe von Verfahren. Bei den meisten bildet man ausgehend von der Wertetabelle oder der kanonischen DNF zunächst alle Primimplikanten. Hierunter ist das bekannteste das Verfahren von Quine und McCluskey. Wir werden hier lediglich exemplarisch zeigen, wie man die Primimplikanten mittels des Karnaugh-Diagramms bestimmt.

Hierzu erinnern wir daran, dass jedes Feld des Karnaugh-Diagramms eineindeutig einem Element des Definitionsbereichs entspricht. Damit entspricht jedes mit 1 markierte Feld einem Minterm. Zum Beispiel entspricht in Abbildung 1.4 auf Seite 10 das Feld in der linken oberen Ecke dem Minterm $X_1X_2\bar{X}_3\bar{X}_4$ und das Feld rechts daneben entspricht dem Minterm $X_1X_2X_3\bar{X}_4$. Da sich beim Wechsel der Zeile oder der Spalte der Wert genau einer Variable ändert, haben die Minterme zweier nebeneinanderliegender mit 1 markierter Felder die Form m_1X_i und $m_1\bar{X}_i$, wobei m_1 ein Monom ist, das die Variable X_i nicht enthält. Damit entspricht das 2×1-Rechteck, das aus diesen beiden Feldern gebildet wird, dem Monom $m_1X_i \vee m_1\bar{X}_i = m_1(X_i \vee \bar{X}_i) = m_1$. Dieses Monom ist offensichtlich Teilmonom der beiden Monome (in diesem Fall Minterme) aus denen es entstanden ist. Die beiden Minterme sind Implikanten, also ist auch das resultierende Monom ein Implikant.

Auf die gleiche Weise kann man natürlich auch zwei 2×1-Rechtecke die mit Einsen markiert sind weiter zusammenfassen, und erhält wiederum ein Teilmonom das ein Implikant ist. Zum Beispiel bilden die vier Einsen der ersten und zweiten Zeile und Spalte in Abbildung 1.4 ein 2×2-Rechteck, das dem Monom X_1X_2 entspricht.

Insgesamt können wir festhalten, dass jedes Rechteck in einem Karnaugh-Diagramm, dessen Seitenlängen Zweierpotenzen sind, einem Monom entspricht. Folglich ist jedes dieser Rechtecke, das in einem Karnaugh-Diagramm nur Einsen überdeckt, ein Implikant. Hierbei ist zu beachten, dass man das Karnaugh-Diagramm so interpretieren muss, als sei es „rundgeklebt", d.h. wenn man am linken Rand herausfällt, macht man am rechten Rand weiter, ebenso mit oberem und unterem Rand. Lässt sich kein größeres Monom-Rechteck finden, das das gegenwärtige Rechteck enthält, und nur Einsen überdeckt, dann ist das gegenwärtige Rechteck schon ein Primimplikant.

Beispiel 1.21 *In dem Karnaugh-Diagramm aus Abbildung 1.4 (Seite 10) lassen sich sechs Primimplikanten finden: die zweite Zeile bildet ein Rechteck mit Seitenlängen 4 und 1. Sie entspricht dem Monom X_2X_4. Die zweite Spalte bildet ein Rechteck mit Seitenlängen 1 und 4, sie entspricht dem Monom X_1X_3. In dem linken oberen 3×3-Block aus Einsen finden sich vier Quadrate mit Seitenlänge 2. Sie entsprechen den Monomen X_1X_2, X_2X_3, X_1X_4, X_3X_4.*

Beispiel 1.22 *In dem Karnaugh-Diagramm aus Abbildung 1.7 finden sich drei ungewöhnliche Primimplikanten. Die vier Ecken bilden wegen des Rundklebens ein Quadrat mit Seitenlänge 2, das dem Monom $\bar{X}_3\bar{X}_4$ entspricht. Die erste und letzte Zeile der ersten und zweiten Spalte bilden ein Quadrat mit Seitenlänge 2, das dem Monom $X_1\bar{X}_4$ entspricht. Die ersten beiden Zeilen der ersten und*

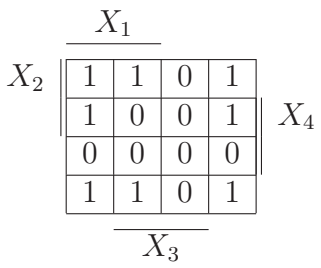


Abbildung 1.7: Vereinfachtes Karnaugh-Diagramm für $n = 4$

letzten Spalte bilden wiederum ein Quadrat mit Seitenlänge 2, das dem Monom $X_2\bar{X}_3$ entspricht.

Ist die Schaltfunktion nur partiell definiert, so kann das Symbol X im Karnaugh-Diagramm als 1 oder 0 interpretiert werden, je nachdem wie es besser passt.

Selbsttestaufgabe 1.13 *Stellen Sie eine Wertetabelle auf für die Schaltfunktion $f : \{0,1\}^4 \to \{0,1\}$, die genau dann den Wert 1 annimmt, wenn höchstens zwei ihrer vier Argumente den Wert 1 annehmen, und die undefiniert ist, wenn genau 3 ihrer Argumente den Wert 1 annehmen. Bestimmen Sie den Träger und erstellen Sie die kanonische disjunktive Normalform. Bestimmen Sie die Kosten der KDNF. Übertragen Sie die Wertetabelle in ein Karnaugh-Diagramm. Bestimmen Sie die Primimplikanten aus dem Karnaugh-Diagramm.*

Lösung auf Seite 37

Es bleibt das auf den ersten Blick einfache Restproblem, aus den Primimplikanten einer Schaltfunktion ein Minimalpolynom zusammenzubauen. Ein solches Minimalpolynom wird i.A. nicht aus allen Primimplikanten bestehen. Man muß deshalb eventuell unter den Primimplikanten eine Auswahl treffen. Hierfür beschreiben wir im weiteren Regeln.

Definition 1.23 *Sei e ein Boole'scher Ausdruck und $a \in \{0,1\}^n$. Wir sagen e* überdeckt *a genau dann, wenn $\phi_a(e) = 1$ gilt. Ist $A \subseteq \{0,1\}^n$ mit $A \neq \emptyset$, dann überdeckt e die Menge A genau dann, wenn e jedes $a \in A$ überdeckt. Der Ausdruck e überdeckt eine Funktion f, wenn e den Träger $\mathrm{Tr}(f)$ der Funktion überdeckt. Ist M eine Menge von Monomen und $F \subseteq \{0,1\}^n$, so heißt die Abbildung $I : M \times F \to \{0,1\}$,* Überdeckung

$$I(m,a) = \begin{cases} 1 \text{ falls } m \text{ überdeckt } a \\ 0 \text{ falls sonst} \end{cases}$$

die Implikantentafel *von M und F.* Implikantentafel

Der Name 'Implikantentafel' kommt daher, daß man I als Matrix aufschreiben kann, deren Zeilen mit den Elementen $m \in M$ und deren Spalten mit den Elementen $a \in F$ indiziert sind. Ist M die Menge der Primimplikanten einer Schaltfunktion f und ist $F = \{a \mid f(a) = 1\}$ der Träger der Schaltfunktion, so heißt I die *Primimplikantentafel* oder *Primtermtabelle* von f. Die

Tabelle 1.5: Funktionstabellen der Schaltfunktionen f_1 und f_2

a_1	a_2	a_3	a_4	$f_1(a_1,a_2,a_3,a_4)$	$f_2(a_1,a_2,a_3)$
0	0	0	0	1	0
0	0	0	1	1	
0	0	1	0	1	1
0	0	1	1	1	
0	1	0	0	0	1
0	1	0	1	0	
0	1	1	0	0	1
0	1	1	1	1	
1	0	0	0	0	1
1	0	0	1	0	
1	0	1	0	0	1
1	0	1	1	0	
1	1	0	0	0	1
1	1	0	1	0	
1	1	1	0	0	0
1	1	1	1	1	

Tabelle 1.6: Primimplikantentafeln für f_1 und f_2

(a)

f_1	0000	0001	0010	0011	0111	1111
$\overline{X_1}X_3X_4$	0	0	0	1	1	0
$X_2X_3X_4$	0	0	0	0	1	1
$\overline{X_1}\,\overline{X_2}$	1	1	1	1	0	0

(b)

f_2	001	010	011	100	101	110
$\overline{X_1}X_3$	1	0	1	0	0	0
$\overline{X_2}X_3$	1	0	0	0	1	0
$\overline{X_1}X_2$	0	1	1	0	0	0
$X_2\overline{X_3}$	0	1	0	0	0	1
$X_1\overline{X_3}$	0	0	0	1	0	1
$X_1\overline{X_2}$	0	0	0	1	1	0

Primimplikantentafeln der beiden Schaltfunktionen $f_1 : \{0,1\}^4 \to \{0,1\}$ und $f_2 : \{0,1\}^3 \to \{0,1\}$, deren Funktionstabellen in Tabelle 1.5 zu sehen sind, findet man in Tabelle 1.6.

Jeder Menge S von Monomen ordnen wir das Polynom

$$p(S) = \bigvee_{m \in S} m$$

zu. Eine Implikantentafel I definiert ein zugehöriges *Überdeckungsproblem*: finde eine Teilmenge $S \subseteq M$ von Monomen, so daß gilt: $p(S)$ überdeckt f. Eine solche Teilmenge heißt eine *Lösung* des Überdeckungsproblems. Ist I die Prim-

Tabelle 1.7: Primimplikantentafel zur Schaltfunktion aus Abb. 1.4

	0011	0101	0110	0111	1001	1010	1011	1100	1101	1110	1111
X_1X_2								1	1	1	1
X_1X_3						1	1			1	1
X_1X_4					1		1		1		1
X_2X_3			1	1						1	1
X_2X_4		1		1					1		1
X_3X_4	1			1			1				1

implikantentafel von f, so gilt offensichtlich

$$p(S) \equiv f(X)$$

für alle Lösungen S von I, da sie gerade alle Implikanten der Funktion f enthält. Die Lösungen S, für die $p(S)$ minimale Kosten hat, sind offensichtlich gerade die Minimalpolynome von f.

Beispiel 1.23 *Wir erstellen die Primimplikantentafel zu der Schaltfunktion aus Abbildung 1.4 (Seite 10). Hierzu bestimmen wir zunächst den Träger der Funktion, d.h. alle Belegungen $a \in \{0,1\}^4$ mit $f(a) = 1$. Das sind nach Definition der Schaltfunktion gerade alle a die mindestens zwei Einsen enthalten. Die Primimplikanten haben wir in Beispiel 1.21 bereits bestimmt. Die Primimplikantentafel ist in Tabelle 1.7 dargestellt. Zur Verdeutlichung haben wir nur die Einträge mit 1 dargestellt. Die Einträge mit 0 sind leer. Man sieht, dass jeder Primimplikant gerade vier Monome überdeckt, nämlich die vier, mittels derer man ihn im Karnaugh-Diagramm identifiziert hat.*

Um eine Lösung eines Überdeckungsproblems zu finden, bestimmen wir zunächst die Monome, die in der Lösung unbedingt enthalten sein müssen.

Definition 1.24 *Sei $I : M \times F \to \{0,1\}$ ein Überdeckungsproblem und $m \in M$. Dann heißt m* wesentlich*, falls es ein $a \in F$ gibt, so daß a nur von m und keinem anderen Monom in M überdeckt wird.* wesentliches Monom

Die wesentlichen Primimplikanten der Primimplikantentafel nennt man auch *Kernimplikant*. Einen Kernimplikanten entdeckt man in der Primimplikantentafel dadurch, dass eine der Einsen in seiner Zeile die einzige Eins in der betreffenden Spalte ist. Kernimplikant

Beispiel 1.24 *Wie man aus Tabelle 1.6 erkennt sind für die Funktion f_1 aus Tabelle 1.5 die Monome $\overline{X_1}\,\overline{X_2}$ wegen der ersten drei Einsen und $X_2X_3X_4$ wegen der letzten Eins wesentlich. Monom $\overline{X_1}X_3X_4$ ist nicht wesentlich. Bei Funktion f_2 ist kein Monom wesentlich. In der Tabelle 1.7 sind alle Primimplikanten wesentlich.*

Das Überdeckungsproblem $I' = I(m) : M' \times F' \to \{0,1\}$ enstehe aus I durch Entfernen von m aus M und durch Entfernen aller von m überdeckten Tupel a aus F. Anschaulich gesprochen entfernen wir alle Spalten, in denen in der Zeile von m eine 1 war, und dann die Zeile von m.

Tabelle 1.8: Vereinfachte Primimplikantentafel für f_1

f_1	0111	1111
$\overline{X_1}X_3X_4$	1	0
$X_2X_3X_4$	1	1

Eine Teilmenge S' von M' ist genau dann eine Lösung $S(m)$ von $I(m)$ wenn $S' \cup \{m\}$ Lösung von I ist. Eine billigste Lösung von Problem I kann also stets aus einer billigsten Lösung des kleineren Problems I' gewonnen werden.

Wir lösen also das Überdeckungsproblem, indem wir alle wesentlichen Monome nacheinander entfernen und eine billigste Lösung für das Restproblem suchen.

Beispiel 1.25 *Sei I das Überdeckungsproblem aus Tabelle 1.6(a) und $m = \overline{X_1}\,\overline{X_2}$. Dann ist I' das Problem aus Tabelle 1.8.*

Wir können natürlich das gleiche Kriterium nochmals anwenden, um in diesem Fall die einzige Lösung minimaler Kosten zu bestimmen.

Tabelle 1.8 illustriert auch ein Kriterium mit dem man Hinweise gewinnt, wie man eine billigste Lösung des Restproblems nach Entfernen der wesentlichen Monome findet:

Sei $I : M \times F \to \{0,1\}$ ein Überdeckungsproblem, und es seien $m, m' \in M$. Wir sagen, daß m das Monom m' *dominiert*, falls $L(m) \leq L(m')$ und falls jedes Tupel a, das von m' überdeckt wird auch von m überdeckt wird.

Beispiel 1.26 *In Tabelle 1.8 dominiert $X_2X_3X_4$ das Monom $\overline{X_1}X_3X_4$.*

Wird m' von m dominiert, so kann man m' in jeder Lösung von I durch m ersetzen. Man erhält wieder eine Lösung, und diese ist nicht teurer als die alte Lösung. Deshalb kann man in diesem Fall m' einfach aus M entfernen.

Man kommt leider häufig in Situationen, in denen das Kriterium nicht anwendbar ist, beispielsweise in Tabelle 1.6(b). Im Allgemeinen kommt man an dieser Stelle nur noch mit roher Gewalt weiter: Man sucht für *alle* $m \in M$ eine billigste Lösung $S(m)$ des Problems $I(m)$ und sucht dann unter den Lösungen $m \vee \bigvee_{r \in S(m)} r$ eine billigste aus. Da man bei den entstehenden kleineren Problemen immer wieder in die gleiche Situation geraten kann, wird das sehr schnell extrem aufwendig:

Für $k \in \mathbf{N}$ sei $i(k)$ die größte Zahl von Überdeckungsproblemen, die insgesamt generiert werden, wenn man mit einem Überdeckungsproblem mit k Monomen startet. Dann ist

$$i(1) = 1,$$

und für $k > 1$ können wir nach dem oben Gesagten $i(k)$ nur abschätzen durch

$$i(k) \leq k \cdot i(k-1) \; .$$

Durch Induktion über k folgt

$$i(k) \leq k! = 1 \cdot 2 \cdot \ldots \cdot k = \Omega(2^k) \; .$$

	m_1	m_2	m_3	m_4	m_5	m_6
pi_1	1	1				
pi_2		1	1	1		
pi_3			1	1	1	
pi_4					1	1
pi_5				1		1

Abbildung 1.8: Primimplikantentafel zu Selbsttestaufg. 1.14

Ob man solche Probleme sehr viel schneller lösen kann, d.h. ob es eine Lösung gibt mit einem Aufwand der polynomiell in k bleibt, ist eine offene Frage, deren Bedeutung weit über die Grenzen der Optimierung von Boole'schen Ausdrücken hinausreicht. In der theoretischen Informatik werden Ihnen solche Fragestellungen als NP-vollständige Probleme wieder begegnen.

In der Praxis ist das Restproblem oft klein, so dass man durch Ausprobieren direkt eine Lösung findet. Enthält das Restproblem zum Beispiel noch drei Primimplikanten, so kann man zunächst alle Möglichkeiten suchen, zwei dieser Primimplikanten zu kombinieren, so dass sie die restlichen Monome vollständig überdecken. Unter diesen sucht man dann die billigste Variante.

Beispiel 1.27 *Die vereinfachte Primimplikantentafel für Schaltfunktion f_2 aus Tabelle 1.5 entspricht ihrer Primimplikantentafel aus Tabelle 1.6(b). Hier gibt es sechs Primimplikanten, von denen keiner den anderen dominiert. Alle haben gleiche Kosten 2, so dass diese bei der Auswahl keine Rolle spielen. Da jeder Primimplikant zwei Monome überdeckt, und sechs Monome zu überdecken sind, braucht eine Lösung des Restüberdeckungsproblems mindestens drei Primimplikanten. Durch genaues Hinsehen findet man drei solche Primimplikanten auch schnell, zum Beispiel $\bar{X}_1X_3$, $X_2\bar{X}_3$ und$X_1\bar{X}_2$. Ein Minimalpolynom der Schaltfunktion f_2 lautet also*

$$p(X_1, X_2, X_3) = \bar{X}_1X_3 \vee X_2\bar{X}_3 \vee X_1\bar{X}_2 \ .$$

Selbsttestaufgabe 1.14 *Bestimmen Sie aus der Primimplikantentafel der Abbildung 1.8 die Kernimplikanten und stellen Sie die vereinfachte Primimplikantentafel auf. Die Primimplikanten sind dort mit pi_i gekennzeichnet, die Monome des Trägers mit m_j. Vereinfachen Sie diese Tafel mit den Kriterien der Wesentlichkeit und der Dominanz. Bilden Sie eine Lösung des Restproblems. Geben Sie ein Minimalpolynom an. Gehen Sie davon aus, dass die Primimplikanten alle gleiche Kosten haben.*

Lösung auf Seite 38

1.4 Exkurs: Unverfügbarkeit von Systemen

Der folgende Abschnitt ist ein Exkurs und damit nicht relevant für die Klausuren am Ende des Kurses. Er soll illustrieren, dass Schaltfunktionen eine Bedeutung haben, die über das Konstruieren von Schaltnetzen weit hinausreicht. Als

Beispiel dient die Modellierung der Unverfügbarkeit (Wahrscheinlichkeit des Ausfalls) eines technischen Systems, wenn man die Fehlerwahrscheinlichkeiten seiner Komponenten kennt. Im wesentlichen stützt sich diese Ausarbeitung auf Schriften von Herrn Prof. Dr. Winfrid Schneeweiss, dem Emeritus am Lehrgebiet Rechnerarchitektur der FernUniversität, s. `http://www.lilole-verlag.de/`.

George Boole hat interessanterweise nicht die uns geläufigen Operatoren $\wedge$, $\vee$ und $\sim$ benutzt. Er hat die Operationen UND, ODER und NOT arithmetisch ausgedrückt:

$$\begin{aligned} a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ \sim a &= 1 - a \end{aligned}$$

Hierbei meinen die Symbole auf der rechten Seite tatsächlich Addition, Subtraktion und Multiplikation bei ganzen (oder reellen) Zahlen. Die Korrektheit kann man leicht nachrechnen.

Am unangenehmsten ist hierbei das ODER, da dabei die Terme a und b doppelt auftauchen. Weiß man allerdings, dass die Terme a und b nie gleichzeitig den Wert 1 annehmen können, dann kann man schreiben $a \dot{\vee} b = a + b$, wobei das Symbol $\dot{\vee}$ ausdrückt, dass die Terme a und b disjunkt sind, d.h. nicht gleichzeitig den Wert 1 annehmen können.

Man kann boolesche Ausdrücke benutzen, um die Fehlerhaftigkeit eines Systems zu beschreiben. Jede Variable X_i ist eine Komponente, und $X_i = 1$ bedeutet dass die Komponente fehlerhaft ist. Hat man zwei Komponenten X_i und X_j hintereinandergeschaltet, dann funktioniert das System nur, wenn beide Komponenten funktionieren, das heißt es fällt aus wenn die eine oder die andere Komponente (oder beide) aufallen und die Fehlerfunktion des Systems ist $X_i \vee X_j$. Bei Parallelschaltung zweier redundanter Komponenten X_i und X_j funktioniert das System solange eine der Komponenten funktioniert, es fällt also aus wenn beide Komponenten ausfallen, und die Fehlerfunktion des Systems ist $X_i \wedge X_j$. Kompliziertere Systeme haben eine Fehlerfunktion, die ein boolescher Ausdruck ist.

Wenn die Komponenten X_i unabhängig voneinander mit Wahrscheinlichkeiten p_i fehlerhaft werden, kann man die Unverfügbarkeit, d.h. die Wahrscheinlichkeit dass das System ausfällt, nach folgendem Verfahren berechnen: man stellt zunächst die Fehlerfunktion als boole'schen Ausdruck auf wie oben beschrieben. Nun transformiert man die Fehlerfunktion so dass man nur noch UND, NOT, und disjunkte ODER hat, ersetzt diese Operatoren durch ihr arithmetisches Äquivalent, ersetzt die Variablen durch ihre Fehlerwahrscheinlichkeiten und rechnet aus.

Wegen des gerade beschriebenen Zusammenhangs hat es viele Ansätze gegeben, boole'sche Ausdrücke zu transformieren, so dass es nur disjunkte ODERs gibt. Einer der einfachsten Ansätze ist das Aufstellen der kanonischen disjunktiven Normalform (KDNF) der Fehlerfunktion, denn zwei verschiedene Minterme haben niemals gleichzeit den Wert 1. Allerdings ist die KDNF typischerweise sehr lang. Ein anderer Ansatz besteht in der Anwendung des sogenannten *Entwicklungssatzes von Shannon*:

$$f(X_1, \ldots, X_n) = X_i \wedge f(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_n)$$

$$\dot{\vee} \bar{X}_i \wedge f(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n) \; .$$

Ist hierbei $f(X_1, \ldots, X_n)$ durch einen beliebigen boole'schen Ausdruck e dargestellt, so bildet die rechte Seite das disjunkte ODER zweier Terme. Diese Terme erhält man, indem man in e jedes Vorkommen der Variablen X_i durch 1 (bzw. 0) ersetzt, die Terme mit den bekannten Rechenregeln vereinfacht und schließlich mit X_i bzw. $\bar{X}_i$ UND-verknüpft. Fährt man mit der Entwicklung so lange rekursiv in den beiden Termen fort, bis nichts mehr zu entwickeln ist, setzt ein und löst auf, erhält man eine disjunktive Normalform, bei der alle ODER tatsächlich disjunkte ODER sind. Man nennt eine solche DNF auch *DDNF, disjunkte disjunktive Normalform.*

1.5 Anhang: Sprechweisen für Notationen

1.5.1 Vorbemerkungen

Notation	**Aussprache**
$a \in A$	a Element (von) A
$A \subseteq B$	A ist Teilmenge von B
$A \subset B$	A ist echte Teilmenge von B
$A = \{a, b, c\}$	A ist die Menge aus/aus den Elementen/der Elemente a, b und c
$\exists a : f(a) = b$ $\exists a.f(a) = b$ $\exists a \vert f(a) = b$	Es existiert ein a mit (der Eigenschaft) f von a gleich b
$A \cup B$	A vereinigt mit B Vereinigung von A und B
$\#A$	Mächtigkeit von A, Anzahl der Elemente von A
$f : A \to B$	f von A nach B, f bildet A auf B ab
$\{(a, b) : a \in A, b \in B\}$	Die Menge aller (Tupel) a, b mit (der Eigenschaft) a Element A, b Element B
$A \times B$	A kreuz B, kartesisches Produkt der Mengen A und B
A^n	A hoch n, das n-fache kartesische Produkt der Menge A
$\bigcup_{i \in \mathbf{N}} A^i$	Vereinigung aller A hoch i mit i Element $\mathbf{N}$
A^+	A Plus
A^*	A Stern
$\sum_{i=0}^{n} x^i$	Summe von x hoch i, für i gleich 0 bis n
$f\,(x) = y$	Die Funktion f hat an der Stelle x den Wert y, f von x gleich y
$indeg(v)$	Indegree von v, Ingrad von v
$outdeg(v)$	Outdegree von v, Outgrad von v

1.5.2 Boole'sche Ausdrücke

Notation	**Aussprache**
$X_1 \wedge X_2$	X_1 und X_2, X_1 and X_2
$X_1 X_2$	X_1 X_2, X_1 und X_2, X_1 and X_2
$X_1 \vee X_2$	X_1 oder X_2, X_1 or X_2
$\sim X_1$, $\neg X_1$	nicht X_1, not X_1
$\overline{e}$	e quer, nicht e, not e (e ist dabei ein erweiterter Boole'scher Ausdruck)
$X_1 \neq X_2$	X_1 ungleich X_2
f_e	Die von e berechnete Funktion
$\phi(e)$	Phi von e
$e_1 \equiv e_2$	e_1 identisch mit e_2, e_1 und e_2 sind äquivalent
$\bigwedge_{i=0}^{n} e_i$	Konjunktion über e_i für i gleich 0 bis n
$\bigvee_{i=0}^{n} e_i$	Disjunktion über e_i für i gleich 0 bis n
$\bigvee_{i \in A} e_i$	Disjunktion über e_i für alle i in A
Aussage 1 $\Leftrightarrow$ Aussage 2	Aussage 1 (ist) äquivalent zu Aussage 2
$\mathrm{Tr}(f)$	Träger von f
$m(a)$	m von a, Minterm zu a
$c(a)$	c von a, Maxterm zu a
$L(e)$	L von e, Kosten von e (falls e Boole'scher Ausdruck)
$L(f)$	L von f, Formelgröße von f (falls f Schaltfunktion)
$f \leq_a g$	f ist asymptotisch durch g beschränkt f ist asymptotisch kleiner gleich g
$O(g)$	O von g
$\Omega(g)$	Omega von g
$\Theta(g)$	Theta von g

1.6 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 1.1 von Seite 5

Es gilt

$$\sum_{i=m}^{n-1} x^i = \sum_{i=0}^{n-1} x^i - \sum_{i=0}^{m-1} x^i \ .$$

Die beiden Summen auf der rechten Seite lassen sich mittels Lemma 1.1 ausdrücken als

$$\frac{x^n - 1}{x - 1} \text{ und } \frac{x^m - 1}{x - 1} \ .$$

Subtrahiert man diese beiden Brüche, erhält man die rechte Seite von Gleichung (1.2).

Selbsttestaufgabe 1.2 von Seite 7

Die Knoten 1 und 2 bilden die Quellen. Es gibt keine Senke im Graphen, da alle Knoten einen Outgrad größer als Null haben. Es gilt $T(1) = T(2) = 0$, da die beiden Knoten Quellen sind, und $T(3) = 1$, da dieser Knoten mit einem Pfad der Länge 1 von Quelle 1 aus erreichbar ist. Knoten 4 ist von Quelle 1 aus mit einem Pfad der Länge 2 erreichbar, und von Quelle 2 aus mit einem Pfad der Länge 1 aus erreichbar. Die Tiefe von Knoten 4 beträgt also 2, da der längste Pfad zählt. Die Tiefe von Knoten 5 ist nicht definiert, da dieser einen Zyklus der Länge 1 mit sich selbst bildet.

Selbsttestaufgabe 1.3 von Seite 9

Die Anzahl der 2-stelligen Schaltfunktionen beträgt $16 = 2^{2^2} = 2^4$. Sie lauten:

$X_1\ X_2$	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Als 1-stellige Schaltfunktion des ersten Arguments können die Funktionen $f_0 = 0$, $f_3 = X_1$, $f_{12} = 1 - X_1$, $f_{15} = 1$ interpretiert werden. Man erhält gerade die Anzahl möglicher 1-stelliger Schaltfunktionen: $4 = 2^{2^1} = 2^2$. Man erkennt die Unabhängigkeit vom zweiten Argument daran, dass die Funktionswerte bei 00 und 01 sowie die Funktionswerte bei 10 und 11 gleich sind.

Selbsttestaufgabe 1.4 von Seite 9

Das Karnaugh-Diagramm hat 4 Felder und bildet also ein 2×2-Quadrat. Um Eindeutigkeit zu erzielen, muss jeweils die Hälfte der Spalten und die Hälfte der Zeilen mit einer Variable markiert werden. Die genaue Zuordnung ob X_1 die Spalten oder die Zeilen markiert, und ob die linke bzw. rechte Spalte markiert wird, spielt in diesem Fall keine Rolle. Ein mögliches KV-Diagramm zeigt die folgende Abbildung.

	$\overline{X_2}$
0	0
0	1

X_1

Selbsttestaufgabe 1.5 von Seite 13

Es gilt $X_1, X_2, X_3 \in \mathrm{EB}_1$ und damit über die erste Regel von Definition 1.9 auch in allen weiteren EB_i. Dann sind $(X_1 \wedge X_2)$ und $f_1(X_1, X_2)$ in EB_2, $((X_1 \wedge X_2) \wedge X_3) \in \mathrm{EB}_3$ und $(((X_1 \wedge X_2) \wedge X_3) \vee f_1(X_1, X_2)) \in \mathrm{EB}_4$. In dem zweiten Ausdruck fehlt die äußere öffnende Klammer zu Anfang des Ausdrucks, sowie eine innere Klammer zur Strukturierung von $(X_1 \wedge X_2 \vee X_3)$. Die letzte Ergänzung ist allerdings nicht eindeutig. Die zwei möglichen Ausdrücke aus EB sind

$$(f_1(X_1, X_2) \vee ((X_1 \wedge X_2) \vee X_3)) \text{ und } (f_1(X_1, X_2) \vee (X_1 \wedge (X_2 \vee X_3))) .$$

Selbsttestaufgabe 1.6 von Seite 15

An der Stelle $a = (1, 1, 1)$ gilt $\phi_a(X_1) = \phi_a(X_2) = 1$, also $\phi_a(X_1 \wedge X_2) = 1 \wedge 1 = 1$. Weiterhin ist $\phi_a(f_1(X_1, X_2)) = f_1(\phi_a(X_1), \phi_a(X_2)) = f_1(1, 1) = 0$. Hieraus folgt $\phi_a((X_1 \wedge X_2) \wedge X_3) = \phi_a(X_1 \wedge X_2) \wedge \phi_a(X_3) = 1 \wedge 1 = 1$ und $\phi_a(((X_1 \wedge X_2) \wedge X_3) \vee f_1(X_1, X_2)) = \phi_a(((X_1 \wedge X_2) \wedge X_3)) \vee \phi_a(f_1(X_1, X_2)) = 1 \vee 0 = 1$

An der Stelle $b = (1, 0, 1)$ gilt $\phi_b(X_1 \wedge X_2) = 1 \wedge 0 = 0$ und damit $\phi_b((X_1 \wedge X_2) \wedge X_3) = 0 \wedge 1 = 0$. Gleichzeitig gilt $\phi_b(f_1(X_1, X_2)) = f_1(1, 0) = 1$. Damit folgt $\phi_b(((X_1 \wedge X_2) \wedge X_3) \vee f_1(X_1, X_2)) = \phi_b(((X_1 \wedge X_2) \wedge X_3)) \vee \phi_b(f_1(X_1, X_2)) = 0 \vee 1 = 1$.

Selbsttestaufgabe 1.7 von Seite 18

Wir wenden die erste Regel unter (B3) an und erhalten $X_1(X_2 \vee X_3) \equiv X_1X_2 \vee X_1X_3$. Beim zweiten Ausdruck wenden wir zunächst die dritte Regel unter (B7) an und erhalten $X_1X_3 \equiv X_1 \wedge X_3 \wedge 1$. Nun ersetzen wir die 1 mittels der zweiten Regel unter (B6) und erhalten $X_1 \wedge X_3 \wedge 1 \equiv X_1 \wedge X_3 \wedge (X_2 \vee \bar{X}_2)$. Schließlich nutzen wir wieder die erste Regel unter (B3) und erhalten $X_1 \wedge X_3 \wedge (X_2 \vee \bar{X}_2) \equiv X_1X_2X_3 \vee X_1\bar{X}_2X_3$.

Selbsttestaufgabe 1.8 von Seite 20

Wir formen den Ausdruck zunächst mittels der Morgan-Formel um und erhalten $\overline{X_1X_2} \wedge (X_1X_2)$. Wir ersetzen nun zur Verdeutlichung X_1X_2 durch e und

erhalten $\bar{e} \wedge e$, was aber nach Regel (B6) identisch zu Null ist. Damit kann es keine Einsetzung geben, unter der der Ausdruck den Wert 1 hat.

Selbsttestaufgabe 1.9 von Seite 21

Nach Tabelle 1.2 gilt $\mathrm{Tr}(\vee) = \{(0,1),(1,0),(1,1)\}$. Damit gilt $m(0,1) = \bar{X}_1 X_2$, $m(1,0) = X_1 \bar{X}_2$, und $m(1,1) = X_1 X_2$. Schließlich ist die kanonische disjunktive Normalform von $\vee$:

$$\bar{X}_1 X_2 \vee X_1 \bar{X}_2 \vee X_1 X_2 \; .$$

Selbsttestaufgabe 1.10 von Seite 23

Die gegebene Beschreibung mittels des Ausdrucks $X_1(X_2 \vee X_3)$ liefert bereits $L(f) \leq 2$. Hier sieht man auch, dass die Schranke aus Satz 1.7 mit $3 \cdot 2^4 = 48$ oft sehr unscharf ist.

Selbsttestaufgabe 1.11 von Seite 24

Der Ausdruck $X_1(X_2 \vee X_3)$ ist eine konjunktive Normalform, da X_1 und $X_2 \vee X_3$ Klauseln sind. Er ist keine disjunktive Normalform, da er kein Monom darstellt. Durch Anwendung des Distributionsgesetzes (B3) kann man ihn aber in die disjunktive Normalform $X_1 X_2 \vee X_1 X_3$ umformen.

Selbsttestaufgabe 1.12 von Seite 25

Die Funktion f mit $f(00) = f(11) = 0$, $f(01) = f(10) = 1$ heißt auch exklusives Oder. Wir prüfen für jedes der Monome, welchen Wert es an den Stellen annimmt, an denen die Funktion den Wert 0 annimmt, denn nur dort kann eine Verletzung der Implikanteneigenschaft '$\leq$' vorkommen. Das Monom $\bar{X}_1 X_2$ hat an den Stellen 00 und 11 den Wert 0, das Monom $X_1 \bar{X}_2$ ebenfalls. Diese beiden Monome sind also Implikanten. Das Monom $X_1 X_2$ hat an der Stelle 11 den Wert 1, also ist $X_1 X_2 \not\leq f$, und dieses Monom ist kein Implikant. Die beiden ersten Monome stellen auch Primimplikanten dar, denn ihre Teilmonome sind X_1, $\bar{X}_1$, X_2, und $\bar{X}_2$, und alle diese Ausdrücke sind keine Implikanten, da sie entweder an der Stelle 00 oder an der Stelle 11 den Wert 1 annehmen, die Funktion f hingegen nicht.

Selbsttestaufgabe 1.13 von Seite 27

Die Wertetabelle ist in Tabelle 1.9 dargestellt. Der Träger der Funktion ist die Menge

$$\mathrm{Tr}(f) = \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 1000, 1001, 1010, 1100\} \; .$$

Die kanonische disjunktive Normalform lautet

$$\begin{aligned} f(X) \;=\; & \bar{X}_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 \vee \bar{X}_1 \bar{X}_2 \bar{X}_3 X_4 \vee \bar{X}_1 \bar{X}_2 X_3 \bar{X}_4 \vee \bar{X}_1 \bar{X}_2 X_3 X_4 \\ & \vee \bar{X}_1 X_2 \bar{X}_3 \bar{X}_4 \vee \bar{X}_1 X_2 \bar{X}_3 X_4 \vee \bar{X}_1 X_2 X_3 \bar{X}_4 \vee X_1 \bar{X}_2 \bar{X}_3 \bar{X}_4 \\ & \vee X_1 \bar{X}_2 \bar{X}_3 X_4 \vee X_1 \bar{X}_2 X_3 \bar{X}_4 \vee X_1 X_2 \bar{X}_3 \bar{X}_4 \; . \end{aligned}$$

Tabelle 1.9: Wertetabelle einer zu analysierenden Schaltfunktion f

$X_1X_2X_3X_4$	$f(X)$
0000	1
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	X
1000	1
1001	1
1010	1
1011	X
1100	1
1101	X
1110	X
1111	0

Die KDNF enthält 10 $\vee$-Operatoren, $33 = 11 \cdot 3$ $\wedge$-Operatoren, und 28 Inverter, ihre Kosten betragen also 71.

Das Karnaugh-Diagramm ist in Abbildung 1.9 abgebildet. Hier ist es sinnvoll, die mit X markierten Felder als 1 zu interpretieren, da sich hierdurch größere Rechtecke bilden lassen. Es ergeben sich vier Primimplikanten. Die erste und die letzte Zeile bilden ein 4×2-Rechteck, das dem Monom $\bar{X}_4$ entspricht. Die dritte und vierte Zeile bilden ein 4×2-Rechteck, das dem Monom $\bar{X}_2$ entspricht. Die erste und die letzte Spalte bilden ein 2×4-Rechteck, das dem Monom $\bar{X}_3$ entspricht. Die dritte und die vierte Spalte bilden ein 2×4-Rechteck, das dem Monom $\bar{X}_1$ entspricht. Keiner dieser Primimplikanten ist ein Kernimplikant, da jede 1 im Karnaugh-Diagramm von mehreren Primimplikanten abgedeckt ist. Zwar sind die X nur jeweils von einem Primimplikanten abgedeckt, allerdings haben sie auf die potentielle Eigenschaft des Kernimplikanten keinen Einfluss, da in der Primtermtabelle nur die Träger-Elemente als Spalten auftauchen. Jeweils drei Primimplikanten überdecken alle Einsen, so dass es vier Minimalpolynome gibt. Eines davon ist

$$f(X) = \bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3 \; .$$

Selbsttestaufgabe 1.14 von Seite 31

Der einzige Kernimplikant ist pi_1, da die Spalte m_1 als einzige nur eine 1 enthält. Die vereinfachte Primimplikantentafel entsteht durch Streichung der Spalten m_1 und m_2 sowie der Zeile pi_1 und ist in Abbildung 1.10 zu sehen. In dieser Tafel wird pi_2 durch pi_3 dominiert, und die betreffende Zeile kann weggelassen werden. Dann ist pi_3 aber wieder wesentlich, und wir können die Spalten m_3,

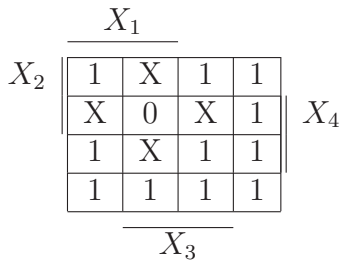


Abbildung 1.9: Karnaugh-Diagramm

	m_3	m_4	m_5	m_6
pi_2	1	1		
pi_3	1	1	1	
pi_4			1	1
pi_5		1		1

Abbildung 1.10: Vereinfachte Primimplikantentafel zu Selbsttestaufg. 1.14

m_4 und m_5 sowie die Zeile mit pi_3 entfernen. Übrig bleibt die Tabelle des Restproblems in Abbildung 1.11. Die beiden Primimplikanten dominieren sich gegenseitig. Da sie gleiche Kosten haben, wählen wir einen aus, zum Beispiel pi_4. Das resultierende Minimalpolynom ist

$$pi_1 \vee pi_3 \vee pi_4 \ .$$

	m_6
pi_4	1
pi_5	1

Abbildung 1.11: Primimplikantentafel des Restproblems zu Selbsttestaufg. 1.14

Kurseinheit 2

Schaltnetze und Zahlendarstellungen

Kapitelinhalt

Zusammenfassung

In dieser Kurseinheit werden Schaltnetze als Realisierungen von Schaltfunktionen und Boole'schen Ausdrücken behandelt. Hierbei wird Wert auf arithmetische Schaltnetze und die dabei verwendeten Zahlendarstellungen gelegt.

Lernziele

Die Lernziele dieser Kurseinheit sind:

- Verständnis der Definition und Verwendung von Schaltnetzen,
- Kenntnis grundlegender Zahlendarstellungen,
- Verwendung einfacher arithmetischer Schaltnetze.

2.1 Schaltnetze

2.1.1 Gatter

Wir haben bereits in Abschnitt 1.2 über Gatter gesprochen. Das sind Schaltungen mit wenigen Eingängen und einem Ausgang, die gewisse einfache Schaltfunktionen berechnen. Wir gehen ab jetzt davon aus, dass uns Gatter zur Berechnung der folgenden Schaltfunktionen zur Verfügung stehen[1]:

1. die bereits bekannten Schaltfunktionen $\wedge$, $\vee$ und $\sim$.
2. NAND : $\{0,1\}^2 \to \{0,1\}$ mit $\text{NAND}(x,y) = \overline{x \wedge y}$ für alle x, y .
3. NOR: $\{0,1\}^2 \to \{0,1\}$ mit $\text{NOR}(x,y) = \overline{x \vee y}$ für alle x, y .
4. $\oplus : \{0,1\}^2 \to \{0,1\}$ mit

$$\oplus(x,y) = 1 \Leftrightarrow x + y = 1 \text{ für alle } x, y \ .$$

Hierbei stellt das Symbol '+' das arithmetische Plus-Symbol dar.

Die letzte Schaltfunktion heißt auch *Antivalenz*, *exklusives ODER (EXOR)* oder *Plus modulo zwei*, da

exklusives Oder

$$\oplus(x,y) = x + y \bmod 2$$

für alle x, y gilt. Die Funktion EXOR nimmt also genau dann den Wert 1 an, wenn eines ihrer Argumente den Wert 1 und das andere den Wert 0 hat. Haben beide Argumente den gleichen Wert, so nimmt EXOR den Wert 0 an. Statt $\oplus(x,y)$ schreibt man gewöhnlich $x \oplus y$ oder $x \not\equiv y$. Die Negation der Antivalenz heißt *Äquivalenz*, in Zeichen $x \equiv y$.

Äquivalenz

Da die Funktion $\oplus$ assoziativ ist, kann man in Ausdrücken wie $(x \oplus (y \oplus z))$ die Klammern weglassen. Wir bemerken noch, dass die Schaltfunktionen $\wedge$, $\vee$, NAND und $\oplus$ alle kommutativ sind.

Ist f eine Schaltfunktion, so nennt man Gatter, die f berechnen *f-Gatter*. Uns stehen also jetzt $\wedge$-Gatter (AND-Gatter), $\vee$-Gatter (OR-Gatter), $\sim$-Gatter (Inverter), NAND-Gatter, NOR-Gatter und $\oplus$-Gatter (EXOR-Gatter) zur Verfügung. Die Menge der direkt durch Gatter realisierbaren Funktionen fassen wir in der Menge

$$K = \{\wedge, \vee, \sim, \text{NAND}, \text{NOR}, \oplus\}$$

zusammen. Man verwendet für Gatter üblicherweise die *Schaltsymbole* aus Abbildung 2.1. Die obere Reihe zeigt Schaltsymbole nach DIN (Deutsche Industrienorm), die untere Reihe Schaltsymbole wie sie im wissenschaftlichen Bereich verwendet werden[2]. Die untere Reihe entspricht bis auf das OR-Gatter dem amerikanischen IEEE (Institute of Electrical and Electronics Engineers) Standard. Wir werden sowohl die Schaltsymbole der oberen wie der unteren Reihe benutzen, allerdings in einer Schaltnetz-Zeichnung nur Symbole einer Reihe.

Schaltsymbol

[1] Die Aussprachen der wichtigsten Notationen sind wie in Kurseinheit 1 in einem Anhang zusammengestellt.

[2] Das Symbol des NOR-Gatters wird analog zum NAND-Gatter als OR-Gatter mit nachgeschaltetem Inverter-Kringel gebildet.

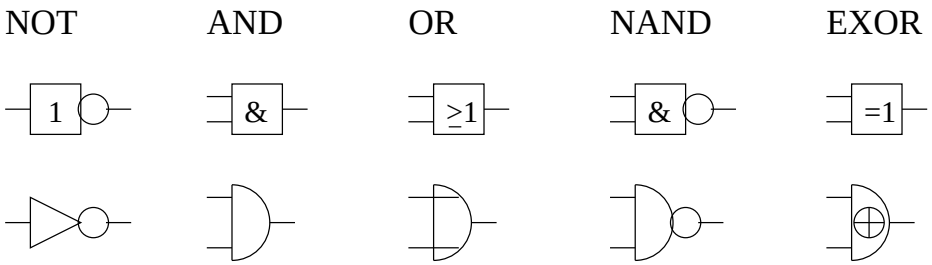


Abbildung 2.1: Schaltsymbole

2.1.2 Schaltnetze

Schaltnetze erhält man nun, indem man Gatter auf spezielle Weise zusammenschaltet. Man geht dabei in vier Schritten vor.

Eingang

1. Man spezifiziert eine endliche Menge $X = \{X_1, \ldots, X_n\}$ von *Eingängen*. Diese Eingänge werden eine ähnliche Rolle wie die Variablen in Boole'schen Ausdrücken spielen.

2. Man spezifiziert einen zykelfreien Graphen $G = (V, E)$ mit den folgenden Eigenschaften:

   - $\{0, 1\} \cup \{X_1 \ldots, X_n\} \subseteq V$, d. h. jeder Eingang ist Knoten des Graphen G. Zusätzlich gibt es zwei spezielle Knoten 0 und 1. Diese Knoten werden später die konstanten Signale 0 und 1 liefern.
   - Die Menge $\{0, 1\} \cup \{X_1, \ldots, X_n\}$ der Eingänge bildet die Quellen von G.
   - Jeder Knoten aus $I = V \setminus (\{X_1, \ldots, X_n\} \cup \{0, 1\})$ hat Ingrad 1 oder 2.

   Die Menge I heißt die Menge der *Gatter*. Die Kanten des Graphen geben die Verdrahtung der Gatter untereinander an. Da Gatter einen oder zwei Eingänge haben, muß auch der Ingrad jedes Gatters 1 oder 2 sein.

3. Man spezifiziert eine Abbildung $g : I \to K$, die für jedes Gatter angibt, welche Funktion es berechnet. Diese Funktion muß sich mit dem Ingrad des Gatters vertragen, d. h. es muß gelten:

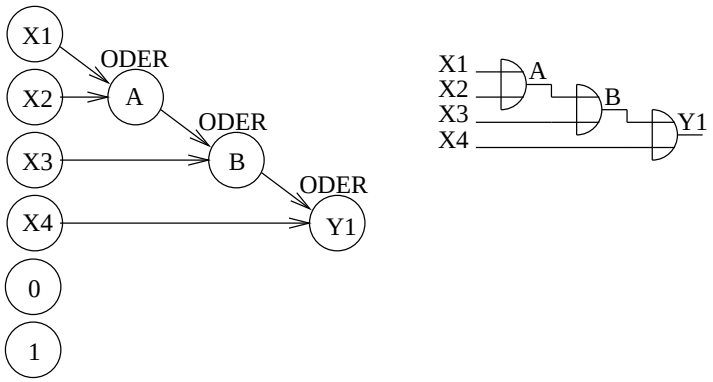
$$g(v) \in \begin{cases} \{\wedge, \vee, \text{NAND}, \oplus\} & \text{falls } indeg(v) = 2 \\ \{\sim\} & \text{falls } indeg(v) = 1 \end{cases}$$

Ausgang

4. Man zeichnet eine Menge $Y = \{Y_1, \ldots, Y_n\}$ von Knoten $Y_i \in V$ als *Ausgänge* aus. Ist $Y = \{Y_1\}$, dann identifizieren wir oft Knoten und Menge.

Schaltnetz

Jedes 4-Tupel $S = (X, G, g, Y)$ mit den oben genannten Eigenschaften spezifiziert ein *Schaltnetz*. Manchmal wird auch der Begriff *Schaltkreis* als Synonym benutzt, obwohl Schaltnetze gerade keine Zyklen enthalten.



(a) Graph (b) Schaltnetz

Abbildung 2.2: Zeichnen von Schaltnetzen

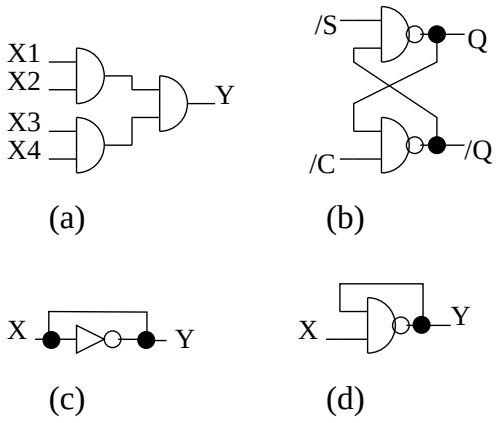


Abbildung 2.3: Schaltnetz oder nicht?

Man kann ein Schaltnetz S zeichnen, indem man den Graphen G zeichnet, und jedes Gatter v zusätzlich mit $g(v)$ beschriftet. Ein Beispiel findet man in Abbildung 2.2(a). Statt ein Gatter v mit $g(v)$ zu beschriften zeichnet man in der Regel jedoch direkt das zugehörige Schaltsymbol. Den Namen v des Gatters schreibt man an den Ausgang des Schaltsymbols. Weil aus den Schaltsymbolen die Richtung der Kanten hervorgeht, spart man sich beim Zeichnen die Spitzen der Pfeile. Die Kreise um die Quellen $0, 1, X_1, \ldots, X_n$ läßt man weg. Werden die speziellen Knoten 0 und 1 nicht als Eingänge von Gattern oder als Ausgänge des Schaltnetzes benutzt, läßt man sie in Zeichnungen ebenfalls einfach weg. Aus Abbildung 2.2(a) entsteht so Abbildung 2.2(b).

Beispiel 2.1 *Abbildung 2.3 zeigt vier weitere Schaltungen. Davon ist nur (a) ein Schaltnetz, die Schaltungen (b), (c) und (d) nicht. Bei (b),(c) und (d) gibt es einen Zyklus. Überdies ist bei (c) der Inverter durch eine Parallelverbindung kurzgeschlossen.*

Man kann natürlich jede der Schaltungen aus Abbildung 2.3 physikalisch aufbauen und den Strom anschalten. Wir werden in Kurseinheit 3 sehen, dass eine dieser Schaltungen sogar sehr nützliche Arbeit leistet. Andere fangen eher an zu qualmen und gehen kaputt.

Selbsttestaufgabe 2.1 *Zeichnen Sie das Schaltnetz* $S = (X, G, g, Y)$ *das durch* $X = \{X_1, X_2, X_3\}$, $Y = \{Y_1\}$, $G = (V, E)$ *mit* $V = X \cup \{A, B, C\} \cup Y$ *und* $E = \{(X_1, A), (X_2, B), (X_3, B), (X_1, C), (X_2, C), (C, A), (A, Y_1), (B, Y_1)\}$ *sowie* $g : \{A, B, C, Y_1\} \to \{NAND\}$, $g(i) = NAND$ *für alle Gatter gegeben ist. Die Knoten 1 und 0 haben wir weggelassen, da wir sie nicht brauchen.*

Lösung auf Seite 87

Es fällt auf, dass die Knoten aus Y laut obiger Konstruktion nicht unbedingt Senken des Graphs sein müssen. Besteht Y nur aus einem Element, so sollte dies eigentlich selbstverständlich sein, denn wenn es eine Senke $u \notin Y$ gibt, dann stellt der Ausgang des Gatters u einen Ausgang des Schaltnetzes dar, der nicht als Ausgang genutzt wird, da $u \notin Y$. Somit wäre dieser Teil des Schaltnetzes sinnlos (es sei denn man will an das Schaltnetz später ein weiteres Schaltnetz „anbauen“, s. Abschnitt 2.2.3). Besteht Y hingegen aus mehreren Elementen, dann könnte ein Element aus Y allein schon deshalb einen Ausgangsgrad größer als Null haben, weil es zur Bestimmung eines anderen Ausgangs der Schaltung gebraucht wird.

Unsere Festlegung lässt nicht zu, dass beide Eingänge eines Gatters mit dem Ausgang des gleichen Gatters verbunden sein können, denn Mehrfachkanten lassen sich in der von uns verwendeten Mengenschreibweise von E nicht darstellen, und bei Verwendung einer Einzelkante wäre der Ingrad des Gatters lediglich 1, obwohl es zwei Eingänge hat. Da in diesem Fall bei einem UND– oder ODER-Gatter die Identität, bei einem NAND-Gatter die Inversion, und bei einem EXOR-Gatter der konstante Wert 0 berechnet wird, verzichten wir auf eine (mathematisch aufwändige) formale Erweiterung.

2.2 Rechnen mit Schaltnetzen

2.2.1 Einsetzungen

Wir definieren nun die Arbeitsweise von Schaltnetzen. Sei $S = (X, G, g, Y)$ ein Schaltnetz, $X = \{X_1, \ldots, X_n\}$ und $G = (V, E)$. Weiter sei $\phi : \{X_1, \ldots, X_n\} \to \{0, 1\}$ eine Einsetzung (auch Eingangsbelegung genannt), die jedem Eingang X_i ein Signal $\phi(X_i) \in \{0, 1\}$ zuordnet. Wir definieren nun auf ziemlich offensichtliche Weise für jeden Knoten $v \in V$ den im Schaltnetz S durch v bei Einsetzung ϕ *berechneten Wert* $\phi(v)$. Der durch Knoten Y berechnete Wert ist der vom Schaltnetz berechnete Wert bei Einsetzung ϕ.

Eingangsbelegung

Weil G zykelfrei ist, hat jeder Knoten $v \in V$ eine Tiefe. Wir setzen für die speziellen Knoten 0 und 1

$$\phi(0) = 0 \text{ und } \phi(1) = 1 \, .$$

Damit ist $\phi(v)$ definiert für alle Knoten v mit Tiefe 0. Wir definieren nun $\phi(v)$ durch Induktion über t für alle Gatter v.

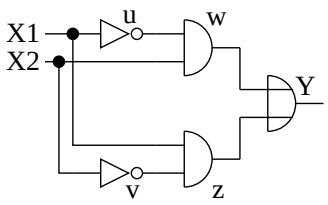


Abbildung 2.4: Schaltnetz zur Berechnung von EXOR

Tabelle 2.1: Berechnete Werte im EXOR Schaltnetz

i	$\phi_i(X_1)$	$\phi_i(X_2)$	$\phi_i(u)$	$\phi_i(v)$	$\phi_i(w)$	$\phi_i(z)$	$\phi_i(Y)$
1	0	0	1	1	0	0	0
2	0	1	1	0	1	0	1
3	1	0	0	1	0	1	1
4	1	1	0	0	0	0	0

Sei $t \in \mathbf{N}$, und $\phi(u)$ sei definiert für alle Gatter u mit Tiefe $t-1$. Es sei v ein Gatter mit Tiefe t. Dann sind zwei Fälle möglich.

1. Ist $indeg(v) = 1$, so hat v einen direkten Vorgänger u mit Tiefe $t-1$, es ist $g(v) = \sim$ und wir definieren

$$\phi(v) = \sim (\phi(u)) \; .$$

2. Ist $indeg(v) = 2$, so hat v zwei direkte Vorgänger u_1 und u_2. Beide haben höchstens Tiefe $t-1$ und wir definieren

$$\phi(v) = \begin{cases} \phi(u_1) \wedge \phi(u_2) & \text{falls } g(v) = \wedge \\ \phi(u_1) \vee \phi(u_2) & \text{falls } g(v) = \vee \\ \overline{\phi(u_1) \wedge \phi(u_2)} & \text{falls } g(v) = \text{NAND} \\ \phi(u_1) \oplus \phi(u_2) & \text{falls } g(v) = \oplus \end{cases}$$

oder kürzer

$$\phi(v) = g(v)(\phi(u_1), \phi(u_2)) \; .$$

Obwohl aus der formalen Definition von Schaltnetz S nicht hervorgeht, welcher der Knoten u_1 und u_2 mit dem rechten Eingang des $g(v)$-Gatters v verbunden ist und welcher mit dem linken Eingang, ist $\phi(v)$ in jedem Fall wohldefiniert. Das liegt an der Kommutativität der Funktionen $\wedge$, $\vee$, NAND und $\oplus$.

Beispiel 2.2 *Wir zeigen, dass das Schaltnetz in Abbildung 2.4 das exklusive Oder aus X_1 und X_2 berechnet. Hierzu berechnen wir für die vier möglichen Einsetzungen ϕ_i, $i = 1, \ldots, 4$, den Wert von $\phi_i(Y)$. Das Resultat ist in Tabelle 2.1 zu sehen.*

Die obige Definition schlägt fehl in den Beispielen aus Abbildung 2.3(b) bis (d) wegen der dort vorkommenden Zyklen.

Durch einen trivialen Induktionsbeweis über die Tiefe von Knoten v zeigt man

Lemma 2.1 *Zur Berechnung von $\phi(v)$ werden als Zwischenergebnisse nur Werte $\phi(u)$ von Knoten u benutzt, die auf einem Pfad von den Eingängen zu v liegen.*

Selbsttestaufgabe 2.2 *Bestimmen Sie für das Schaltnetz aus Selbsttestaufgabe 2.1 die berechneten Werte bei allen Einsetzungen.*

Lösung auf Seite 87

2.2.2 Identitäten und berechnete Funktionen

Sei $S = (X, G, g, Y)$ mit $G = (V, E)$ ein Schaltnetz. Wir haben oben für jede Belegung $\phi : \{X_1, \ldots, X_n\} \to \{0, 1\}$ und jedes Gatter v einen Wert $\phi(v) \in \{0, 1\}$ definiert. Eine solche Konstruktion haben wir früher statt mit Knoten $v \in V$ schon mit erweiterten Boole'schen Ausdrücken $e \in \mathrm{EB}$ durchgeführt. Zusammen mit der Definition der Äquivalenz von Ausdrücken war diese Konstruktion der Dreh– und Angelpunkt für die Herleitung der Regeln für das 'gewöhnliche' Rechnen. Man könnte deshalb hoffen, dass man mit Gattern eines vorgegebenen Schaltnetzes genauso rechnen kann wie mit erweiterten Boole'schen Ausdrükken.

Das läßt sich sogar sehr leicht rechtfertigen: Man definiert die Menge $\mathrm{EB}(S)$ der *zu S gehörigen erweiterten Ausdrücke*, indem man einfach in der Definition der gewöhnlichen erweiterten Ausdrücke die Menge $\mathrm{EB}_1 = \{0, 1, X_1, \ldots, X_n\}$ durch die gesamte Menge V ersetzt. Damit hat man gerade die Gatter des Schaltnetzes zusätzlich in die Menge $\mathrm{EB}_1(S)$ aufgenommen.

Für $f, g \in \mathrm{EB}(S)$ definieren wir

$$f \equiv_S g$$

genau dann, wenn

$$\phi(f) = \phi(g) \text{ für alle Einsetzungen } \phi : \{X_1, \ldots, X_n\} \to \{0, 1\}$$

gilt.

Wenn klar ist, in welchem Schaltnetz wir rechnen, schreiben wir ab jetzt statt '$\equiv_S$' einfach '$\equiv$' oder — um Schreibarbeit zu sparen — einfach '$=$'. Aus den Definitionen des Abschnitts 2.2.1 folgt sofort:

Lemma 2.2 *Sei $v \in V$. Hat v nur einen direkten Vorgänger u, so gilt*

$$v \equiv_S \sim u \; .$$

Hat v zwei direkte Vorgänger u_1 und u_2, so ist

$$v \equiv_S \begin{cases} u_1 \wedge u_2 \ \textit{falls}\ g(v) = \wedge \\ u_1 \vee u_2 \ \textit{falls}\ g(v) = \vee \\ \overline{u_1 \wedge u_2} \ \textit{falls}\ g(v) = \mathrm{NAND} \\ u_1 \oplus u_2 \ \textit{falls}\ g(v) = \oplus \end{cases}$$

oder kürzer

$$v \equiv_S g(v)(u_1, u_2) \; .$$

Beispiel 2.3 *Für das Schaltnetz aus Beispiel 2.2 gilt*

$$\begin{aligned} u &= \sim X_1 \\ v &= \sim X_2 \\ w &= u \wedge X_2 \\ &= \sim X_1 \wedge X_2 \\ z &= v \wedge X_1 \\ &= X_1 \wedge \sim X_2 \\ Y &= w \vee z \\ &= (\sim X_1 \wedge X_2) \vee (X_1 \wedge \sim X_2) \end{aligned}$$

Gilt $v = e$ für einen Knoten v und einen Ausdruck $e \in \mathrm{EB}(S)$, so sagen wir: *v berechnet e.*

Wir gewinnen auch sofort:

Satz 2.3 *Sei S ein Schaltnetz mit n Eingängen. Dann gibt es zu jedem Knoten v in S genau eine n-stellige Schaltfunktion f_v mit $f_v(X) \equiv_S v$. Sie heißt die von v* berechnete Funktion.

Beweis: Für $a \in \{0,1\}^n$ berechnet man $f_v(a)$, indem man für alle i am Eingang X_i das Signal a_i anlegt und dann auswertet, d. h.

$$f_v(a) = \phi_a(v)$$

mit

$$\phi_a(X_i) = a_i \text{ für alle } i \ .$$

■

Beispielsweise berechnet das Schaltnetz aus Abbildung 2.4 die Funktion $\oplus$.

Hat das Schaltnetz S die Ausgänge $\{Y_1, \ldots, Y_m\}$ für $m \geq 2$, so heißt die Funktion

$$f : \{0,1\}^n \to \{0,1\}^m \text{ mit}$$
$$f(a) = (\phi_a(Y_1), \ldots, \phi_a(Y_m))$$

für alle $a \in \{0,1\}^n$ die von Schaltnetz S *berechnete Funktion*. Wir verallgemeinern damit Definition 1.7 und betrachten ab jetzt jede Funktion $f : \{0,1\}^n \to \{0,1\}^m$ als *Schaltfunktion*.

Selbsttestaufgabe 2.3 *Bestimmen Sie die in Selbsttestaufgabe 2.1 berechnete Schaltfunktion.*

Lösung auf Seite 87

2.2.3 Anfangsschaltnetze

Nun können wir in einem festen Schaltnetz S schon rechnen, wie wir das gewöhnt sind. Gehen wir zu einem neuen Schaltnetz S' über, das völlig anders aufgebaut ist als S, dann können wir nicht erwarten, dass Rechnungen mit Knoten aus S uns irgendetwas über Knoten in S' verraten, und wir müssen im Allgemeinen von vorn anfangen zu rechnen. Wenn man das Schaltnetz S'

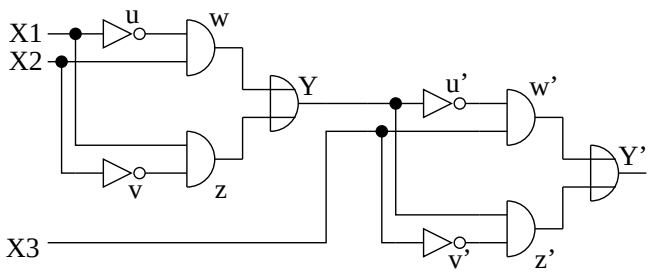


Abbildung 2.5: Berechnung von EXOR mit drei Eingängen

jedoch dadurch gewinnt, dass man an das Schaltnetz S so anbaut, dass die Verbindungen von Schaltnetz S mit den Eingängen intakt bleiben, dann sollte man Rechnungen mit Knoten in S für das neuen Schaltnetz S' wiederverwerten können. Das läßt sich in der Tat leicht rechtfertigen:

Definition 2.1 *Es seien $S = (X, G, g, Y)$ mit $G = (V, E)$ und $S' = (X', G', g', Y')$ mit $G' = (V', E')$ Schaltnetze mit $X \subseteq X'$, $V \subseteq V'$, $E \subseteq E'$ und $g(v) = g'(v)$*
Anfangsschaltnetz *für alle $v \in V$. Dann heißt S ein* Anfangsschaltnetz *von S'.*

Beispiel 2.4 *Das Schaltnetz aus Beispiel 2.2 ist ein Anfangsschaltnetz des Schaltnetzes aus Abbildung 2.5, der ein exklusives ODER mit drei Eingängen berechnet.*

Ist S ein Anfangsschaltnetz von S', dann kann man das Schaltnetz S' konstruieren, indem man zuerst das Schaltnetz S konstruiert und dann anbaut. Das folgende Lemma besagt, dass man dabei die Pfade von den Eingängen zu den Knoten in V nicht verändert.

Lemma 2.4 *Ist S ein Anfangsschaltnetz von S', dann gibt es keinen Pfad von einem Knoten in $V' \setminus V$ zu einem Knoten $v \in V$.*

Beweis durch Induktion über die Tiefe von v: Die Aussage ist offensichtlich richtig für Knoten v der Tiefe 0, da diese alle keine Vorgänger haben.

Sei nun $v \in V$ ein Knoten mit Tiefe $t + 1$. Da S ein Schaltnetz ist, hat v einen direkten Vorgänger in V falls $g(v) = \sim$ oder zwei direkte Vorgänger in V falls $g(v) \neq \sim$. Da S' Schaltnetz ist und $g'(v) = g(v)$ gilt hat v in V' keine zusätzlichen Vorgänger. Also gibt es keine Kante von $V' \setminus V$ nach V. Nach Induktionsvoraussetzung gibt es aber auch keinen Pfad von $V' \setminus V$ zu den direkten Vorgängern von v. ∎

Aus Lemma 2.1 und Lemma 2.4 folgt direkt

Lemma 2.5 *Sei S Anfangsschaltnetz von S', $v \in V$ und $\phi : \{X_1, \ldots, X_n\} \to \{0, 1\}$ eine Einsetzung. Dann führt die Berechnung des Wertes $\phi(v)$ in S und in S' zum gleichen Ergebnis*[3].

Das liefert aber sofort:

[3] $\phi^S(v) = \phi^{S'}(v)$ wenn wir uns in Abschnitt 2.2.1 den Index nicht geschenkt hätten.

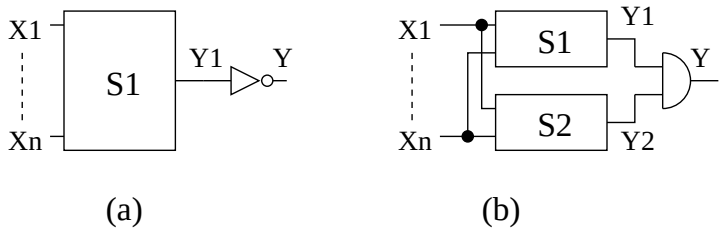


Abbildung 2.6: Schaltnetze zu gegebenen Boole'schen Ausdrücken

Satz 2.6 *Ist S Anfangsschaltnetz von S' und sind $f, g \in \mathrm{EB}(S) \cap \mathrm{EB}(S')$, dann gilt $f \equiv_S g$ genau dann, wenn $f \equiv_{S'} g$ gilt.*

Rechnungen für das Anfangsschaltnetz S können also für S' wiederverwertet werden.

2.2.4 Darstellungssatz

Wir übertragen Satz 1.6 auf Schaltnetze. Zunächst folgern wir mit einem sehr leichten Beweis

Satz 2.7 *Zu jedem Boole'schen Ausdruck $e \in B$ gibt es ein Schaltnetz S mit Eingängen $\{X_1, \ldots, X_n\}$ und mit einem einzigen Ausgang Y so dass*

$$e \equiv_S Y$$

gilt.

Beweis durch Induktion über den Aufbau der Boole'schen Ausdrükke: Für $e \in B_0 = \{0,1\} \cup \{X_1, \ldots, X_n\}$ braucht man gar keine Gatter. Man setzt einfach $Y = e$, d. h. man macht einfach den passenden Eingang oder speziellen Knoten des Schaltnetzes zum Ausgang.

Sei nun $i \in \mathbf{N}_0$ und $e \in B_{i+1}$. Ist $e = \sim e_1$ mit $e_1 \in B_i$, so gibt es nach Induktionsvoraussetzung ein Schaltnetz S_1 mit Ausgang Y_1 so dass $Y_1 \equiv_{S_1} e_1$. Sei S das Schaltnetz aus Abbildung 2.6(a). Schaltnetz S_1 ist Anfangsschaltnetz von S. Also gilt nach Satz 2.6

$$Y_1 \equiv_S e_1$$

und somit

$$\begin{aligned} Y &\equiv_S \sim Y_1 \text{ wegen Lemma 2.1} \\ &\equiv_S \sim e_1 \text{ wegen Satz 2.6} \\ &= e\,. \end{aligned}$$

Ist $e = e_1 \circ e_2$ mit $\circ \in \{\wedge, \vee\}$, so gibt es nach Induktionsvoraussetzung Schaltnetze S_1, S_2 mit Ausgängen Y_1, Y_2 so dass $Y_1 \equiv_{S_1} e_1$ und $Y_2 \equiv_{S_2} e_2$. Sei S das Schaltnetz aus Abbildung 2.6(b). Die Schaltnetze S_1, S_2 sind beide Anfangsschaltnetze von S und wir folgern

$$y \equiv_S Y_1 \circ Y_2$$

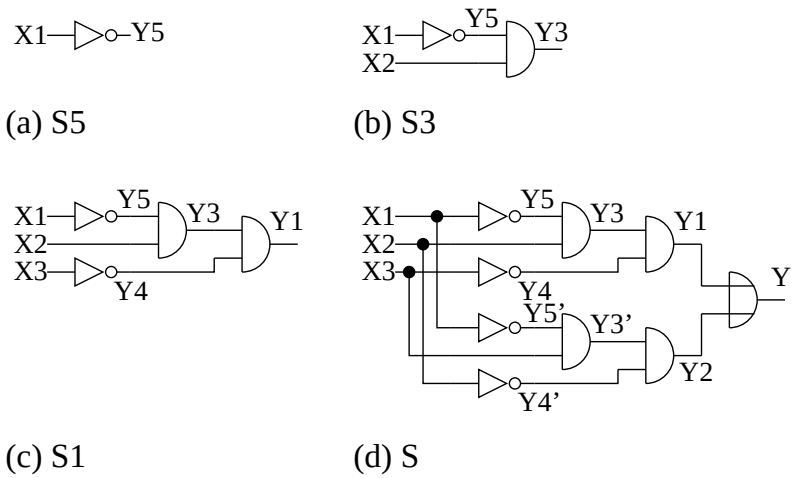


Abbildung 2.7: Konstruktion von Schaltnetz S zu Polynom $p(X)$

$$\begin{aligned} &\equiv_S && e_1 \circ e_2 \\ &= && e\ . \end{aligned}$$

■

Beispiel 2.5 *Wir konstruieren ein Schaltnetz S, das das Polynom*

$$p(X) = \overline{X_1}X_2\overline{X_3} \vee \overline{X_1}X_3\overline{X_2}$$

berechnet. Hierzu konstruieren wir zuerst die Schaltnetze S_1 und S_2, die die Monome $e_1 = \overline{X_1}X_2\overline{X_3}$ und $e_2 = \overline{X_1}X_3\overline{X_2}$ berechnen.

Zur Konstruktion von S_1 brauchen wir Schaltnetze S_3 und S_4, die $e_3 = \overline{X_1}X_2$ und $e_4 = \overline{X_3}$ berechnen. Die Konstruktion von S_2 kann analog erfolgen durch Vertauschung der Rollen von X_2 und X_3.

Zur Konstruktion von S_3 brauchen wir ein Schaltnetz S_5, das $e_5 = \overline{X_1}$ berechnet. Die Konstruktion von S_4 kann analog zu der von S_5 erfolgen, indem man die Rolle von X_1 durch X_3 ersetzt.

Die Schaltnetze sind in Abbildung 2.7 zu sehen.

Zusammen mit Satz 1.6 folgt sofort:

Satz 2.8 *Zu jeder Schaltfunktion $f : \{0,1\}^n \to \{0,1\}$ gibt es ein Schaltnetz S, das f berechnet.*

Sei nun $f : \{0,1\}^n \to \{0,1\}^m$ eine Schaltfunktion mit Wertebereich $\{0,1\}^m$. Dann gibt es m Schaltfunktionen $f_i : \{0,1\}^n \to \{0,1\}$ so dass

$$f(a) = (f_1(a), \ldots, f_m(a))$$

für alle $a \in \{0,1\}^n$. Nach Satz 2.8 gibt es zu jeder Funktion f_i ein Schaltnetz S_i mit Ausgang Y_i, der f_i berechnet. Bilden wir mit diesen Schaltnetzen das Schaltnetz S aus Abbildung 2.8, so gilt: S berechnet f.

Wir haben damit

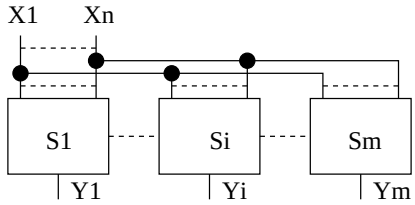


Abbildung 2.8: Konstruktion von Schaltnetz S aus Schaltnetzen S_i

Tabelle 2.2: Funktionstabelle der Beispielfunktion

X_1	X_2	X_3	f_2	f_1
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Satz 2.9 *Zu jeder Schaltfunktion $f : \{0,1\}^n \to \{0,1\}^m$ gibt es ein Schaltnetz S, das f berechnet.*

Man kann also jede Schaltfunktion durch ein Schaltnetz berechnen.

Beispiel 2.6 *Wir konstruieren ein Schaltnetz S zur Berechnung der Funktion $f : \{0,1\}^3 \to \{0,1\}^2$. Die Funktionstabelle von f ist in Tabelle 2.2 zu sehen.*

Das Schaltnetz S_1 zur Berechnung von f_1 ist gerade das Schaltnetz aus Abbildung 2.5. Weiterhin gilt

$$f_2 = \overline{X_1}X_2X_3 \vee X_1\overline{X_2}X_3 \vee X_1X_2\overline{X_3} \vee X_1X_2X_3 \ .$$

Das Schaltnetz S_2 zur Berechnung von f_2 ist in Abbildung 2.9(a) zu sehen, das Schaltnetz S in Abbildung 2.9(b).

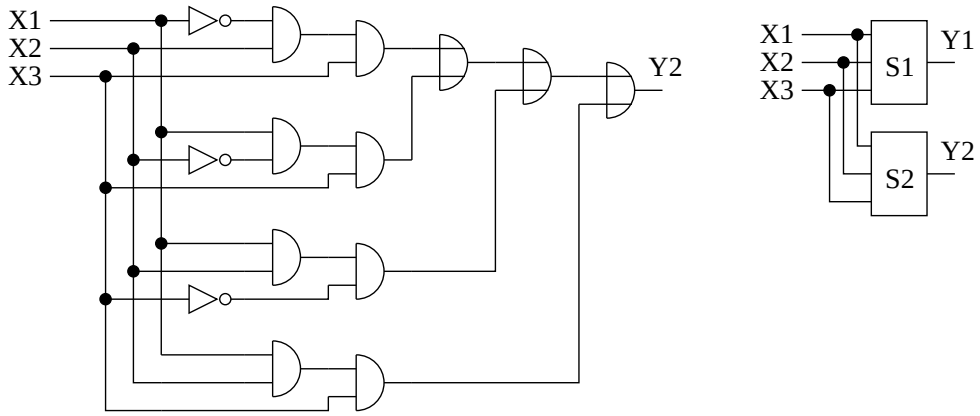
Selbsttestaufgabe 2.4 *Konstruieren Sie ein Schaltnetz zur Berechnung der Schaltfunktion $f(X_1, X_2, X_3, X_4, X_5) = (X_1X_2 \vee X_2X_3 \vee X_1X_3)(X_4 \vee X_5)$.*

Lösung auf Seite 87

2.3 Schaltnetzkomplexität

2.3.1 Komplexitätsmaße

Wir messen die Kompliziertheit von Schaltnetzen $S = (X, G, g, Y)$ durch zwei Maße:



(a) S2 (b) S

Abbildung 2.9: Schaltnetz zur Berechnung der Beispielfunktionen f_2 und f

Kosten
Tiefe

Definition 2.2 *Die* Kosten $\mathrm{C}(S)$ *des Schaltnetzes S sind gleich der Anzahl der Gatter von S. Die* Tiefe $\mathrm{T}(S)$ *des Schaltnetzes S ist gleich der Tiefe des Graphen $G = (V, E)$.*

Beispiel 2.7 *Die Kosten der Schaltnetze in den Abbildungen 2.2(b) und 2.3(a) sind jeweils 3, die Tiefen sind 3 bzw. 2.*

Realisiert man ein Schaltnetz S durch physikalische Gatter, von denen jedes d Euro kostet, so kosten alle Gatter im Schaltnetz zusammen gerade $d \cdot \mathrm{C}(S)$ Euro. Schaltnetzkosten modellieren also gewöhnliche Kosten in Euro.

Gatter, die man physikalisch realisiert, schalten auch nicht unendlich schnell. Vielmehr machen sich Änderungen an den Eingängen eines Gatters erst nach einer gewissen Verzögerungszeit am Ausgang bemerkbar. Hat jedes Gatter eine Verzögerungszeit von t Sekunden, so dauert es höchstens $t \cdot \mathrm{T}(S)$ Sekunden, bis eine Änderung von Signalen an den Eingängen eines Schaltnetzes sich an den Ausgängen bemerkbar macht. Die Tiefe von Schaltnetzen modelliert also Verzögerungszeiten.

Analog zur Formelgröße können wir jetzt Schaltnetzkomplexität und Tiefe von Schaltfunktionen erklären:

Definition 2.3 *Sei f eine Schaltfunktion. Dann heißt*

$$\mathrm{C}(f) = \min\{\mathrm{C}(S) \mid S \textit{ berechnet } f\}$$

die Schaltnetzkomplexität *von f, und*

$$\mathrm{T}(f) = \min\{\mathrm{T}(S) \mid S \textit{ berechnet } f\}$$

heißt die Tiefe *von f.*

Offensichtlich gilt für alle Schaltnetze S die triviale Abschätzung $\mathrm{T}(S) \leq \mathrm{C}(S)$. Hieraus folgt

$$\mathrm{T}(f) \leq \mathrm{C}(f)$$

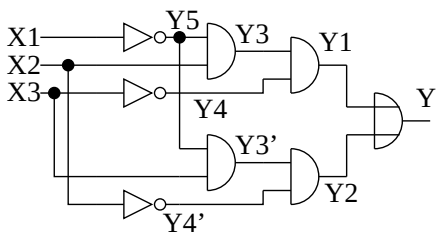


Abbildung 2.10: Vereinfachung durch einmalige Berechnung invertierter Literale

für alle Schaltfunktionen f.

Im Beweis von Satz 2.7 wurde für jedes Funktionszeichen in Ausdruck e ein Gatter benutzt. Für das so konstruierte Schaltnetz S gilt also $\mathrm{C}(S) = L(e)$.

Zusammen mit Satz 1.7 erhalten wir

Satz 2.10 *Für alle $f : \{0,1\}^n \to \{0,1\}$ gilt*

$$\mathrm{T}(f) \leq \mathrm{C}(f) \leq n2^{n+1} .$$

Bei der Herleitung von Satz 2.10 haben wir kanonische disjunktive Normalformen

$$p = \bigvee_{a \in \mathrm{Tr}(f)} m(a)$$

mit Hilfe von Satz 2.8 in Schaltnetze umgewandelt. In diesen Schaltnetzen hat jedes Gatter Outgrad 1, d. h. wir haben den Ausgang eines jeden Gatters nur an einer Stelle verwendet und nie den Ausgang eines Gatters an mehreren Stellen genutzt. Insbesondere werden alle Literale der Form $\overline{X_i}$ für jedes Monom $m(a)$ in p getrennt berechnet. Natürlich genügt es in einem Schaltnetz, jedes solche Literal mit einem einzigen Inverter zu berechnen und dann das Ergebnis nötigenfalls an mehreren Stellen zu verwenden. Aus dem Schaltnetz in Abbildung 2.7(d) wird so das Schaltnetz in Abbildung 2.10.

Diese Konstruktion ist von größter praktischer Bedeutung, beispielsweise beim Aufbau der *Kontroll-Logik* von Rechnern. Allgemein reichen zum Berechnen einer n-stelligen Schaltfunktion

- n Inverter um von jeder Variable das Inverse zu bestimmen,
- $(n-1)2^n$ $\wedge$-Gatter zur Bildung der Minterme und
- $2^n - 1$ $\vee$-Gatter zur Veroderung der Minterme

und wir haben

Satz 2.11 *Für alle $f : \{0,1\}^n \to \{0,1\}$ gilt*

$$\mathrm{T}(f) \leq \mathrm{C}(f) \leq n2^n + n .$$

Selbsttestaufgabe 2.5 *Bestimmen Sie die Kosten des Schaltnetzes aus Abbildung 2.9(b). Lassen sich diese Kosten einfach verringern?*

Lösung auf Seite 88

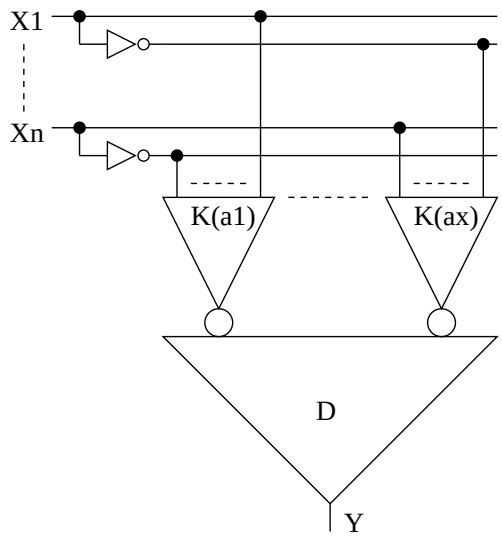


Abbildung 2.11: Teilgraphen des Schaltnetzes zu einem Polynom

2.3.2 Assoziativität und balancierte Bäume

Alle bisherigen Abschätzungen über die Tiefe von Schaltfunktionen folgen aus der trivialen Abschätzung $\mathrm{T}(S) \leq \mathrm{C}(S)$. Um diese Abschätzungen zu verbessern, benutzen wir ein einfaches graphentheoretisches Konzept.

Definition 2.4 *Es seien $G = (V, E)$ und $G' = (V', E')$ gerichtete Graphen. Dann heißt G'* Teilgraph *von G falls*

$$V' \subseteq V, \quad E' \subseteq E \; .$$

Teilbaum *Ist G' ein Baum, so heißt G' ein* Teilbaum *von G.*

Die Graphen der Schaltnetze S, die wir im vorigen Abschnitt aus Polynomen

$$p = \bigvee_{a \in \mathrm{Tr}(f)} m(a)$$

konstruiert haben, enthalten die folgenden Teilgraphen (siehe Abbildung 2.11):

1. die Eingänge $X_1, \ldots, X_n$ und die Inverter zur Berechnung von $\overline{X_1}, \ldots, \overline{X_n}$.
2. für jedes a mit $f(a) = 1$ einen binären Baum $K(a)$ mit n Blättern aus der Menge $\{X_1, \ldots, X_n, \overline{X_1}, \ldots, \overline{X_n}\}$ und $n-1$ vielen $\wedge$-Gattern, dessen Wurzel das Monom $m(a)$ berechnet.
3. einen binären Baum D, dessen Blätter die Wurzeln der Bäume $K(a)$ sind. Dieser Baum hat $\#\mathrm{Tr}(f) = \#\{a \mid f(a) = 1\}$ viele Blätter sowie $\#\mathrm{Tr}(f) - 1$ viele $\vee$-Gatter, und seine Wurzel berechnet p.

Da jeder Pfad durch das Schaltnetz höchstens einen Inverter und nur einen der Bäume $K(a)$ trifft, folgt sofort

$$\begin{aligned} \mathrm{T}(f) &\leq 1 + \max\{\mathrm{T}(K(a)) \mid a \in \{0,1\}^n \text{ und } f(a) = 1\} + \mathrm{T}(D) \quad (2.1) \\ &\leq 1 + n - 1 + 2^n - 1 \; . \end{aligned}$$

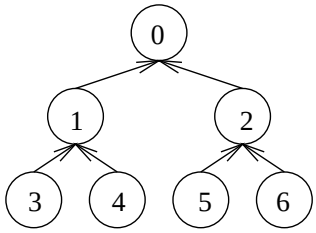


Abbildung 2.12: Verringerung der Tiefe eines Baumes

Da die Funktion $\wedge$ assoziativ ist, können wir in S jeden der Bäume $K(a)$ durch irgendeinen anderen binären Baum mit den gleichen Blättern und $n-1$ vielen $\wedge$-Gattern ersetzen, ohne dass sich die von S berechnete Funktion ändert. Beispielsweise können wir den Baum in Abbildung 1.3(a) durch den Baum aus Abbildung 2.12 ersetzen.

Ebenso können wir D durch einen beliebigen Baum mit den gleichen Blättern und $\#\mathrm{Tr}(f)-1$ vielen $\vee$-Gattern ersetzen. Wir interessieren uns deshalb für binäre Bäume mit n Knoten und möglichst geringer Tiefe. Solche Bäume nennt man *balanciert*.

balancierter Baum

Lemma 2.12 *Für jedes $n \in \mathbf{N}$ gibt es einen binären Baum B_n mit n Blättern und Tiefe $\lceil \log n \rceil$.*

Die Zeichen $\lceil \quad \rceil$ bedeuten dabei die Aufrundung einer reellen Zahl zur nächstgrößeren Ganzzahl. Den Logarithmus bilden wir zur Basis 2.

Beweis durch Induktion über n: Der Baum B_1 besteht aus einem einzigen Knoten. Für den Induktionsschritt sei nun $n > 1$, und es sei

$$p = \max\{2^k \mid 2^k < n \text{ und } k \in \mathbf{N}_0\},$$

d. h. p ist die größte Zweierpotenz, die kleiner als n ist.

Dann ist

$$n = p + n'$$

mit $n' \in \{1, \ldots p\}$. Es folgt

$$p < n \leq 2p\ .$$

Logarithmieren liefert

$$\lceil \log p \rceil = \log p < \log n \leq \log(2p) = \log p + 1,$$

also

$$\lceil \log n \rceil = \log p + 1 = \lceil \log p \rceil + 1\ . \tag{2.2}$$

Wir konstruieren B_n aus den Bäumen B_p und $B_{n'}$ wie in Abbildung 2.13 angegeben. Dann ist

$$\mathrm{T}(B_n) = \mathrm{T}(B_p) + 1.$$

Das Lemma folgt nun direkt aus (2.2). ∎

Als Beispiel zeigt Abbildung 2.14 die Bäume B_n für $n = 1, 2, 3, 4$ und 7.

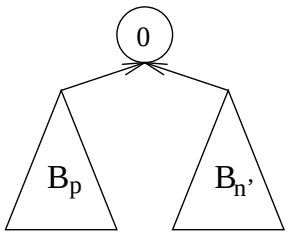


Abbildung 2.13: Konstruktion von B_n aus B_p und $B_{n'}$

Aus Lemma 2.12 und (2.1) folgt sofort

$$\mathrm{T}(f) \leq 1 + \lceil \log n \rceil + \lceil \log(2^n) \rceil \ .$$

Also gilt

Satz 2.13 *Für alle* $f : \{0,1\}^n \to \{0,1\}$ *gilt*

$$\mathrm{T}(f) \leq n + \lceil \log n \rceil + 1 \ .$$

Dies ist ein erstaunliches Ergebnis. Wir haben, ohne zusätzliche Kosten zu erhalten, die Tiefe eines Schaltnetzes von $O(n2^n)$ auf $O(n)$ reduziert, was einen gewaltigen Unterschied macht.

Selbsttestaufgabe 2.6 *Konstruieren Sie ein Schaltnetz zur ODER-Verknüpfung von 14 Variablen* $X_1, \ldots, X_{14}$*. Welche Kosten hat dieses Schaltnetz? Welche Tiefe hat es im schlechtesten Fall, welche Tiefe hat es im besten Fall?*

Lösung auf Seite 88

2.3.3 Boole'sche Ausdrücke und korrespondierende Schaltnetze

Gegeben sei ein Boole'scher Ausdruck e. Es ist nun trivial, ein Schaltnetz $S(e)$ so zu konstruieren, dass die von e und $S(e)$ berechneten Funktionen übereinstimmen, und dass $S(e)$ die gleiche Struktur hat wie e. Damit ist — umgangssprachlich ausgedrückt — gemeint, dass es zu jedem Operator in e ein korrespondierendes Gatter in $S(e)$ gibt, und dass die Verdrahtung der Gatter gerade die Struktur der Klammerung, d.h. der Teilausdrücke von e, spiegelt. Wir nennen $S(e)$ das zu e *korrespondierende Schaltnetz*. Insbesondere gilt $\mathrm{C}(S(e)) = L(e)$, wobei $L(e)$ die Kosten von e aus Kurseinheit 1 sind. Daraus folgt für die von e und $S(e)$ berechnete Funktion f natürlich auch $\mathrm{C}(f) \leq L(f)$. Oft erlaubt man, dass das Inverse einer Variable im korrespondierenden Schaltnetz nur einmalig durch einen Inverter berechnet werden muss, auch wenn die invertierte Variable in der Formel mehrfach benutzt wird. Man beschränkt so die Strukturgleichheit auf die 2-stelligen Operatoren. In einem solchen Fall gilt $\mathrm{C}(S(e)) < L(e)$.

Ist umgekehrt ein Schaltnetz S mit einem Ausgang gegeben, dann gibt es zwar stets einen Boole'schen Ausdruck e, der die gleiche Funktion wie S berechnet, aber es ist nicht offensichtlich, ob es immer einen zu S *korrespondierenden*

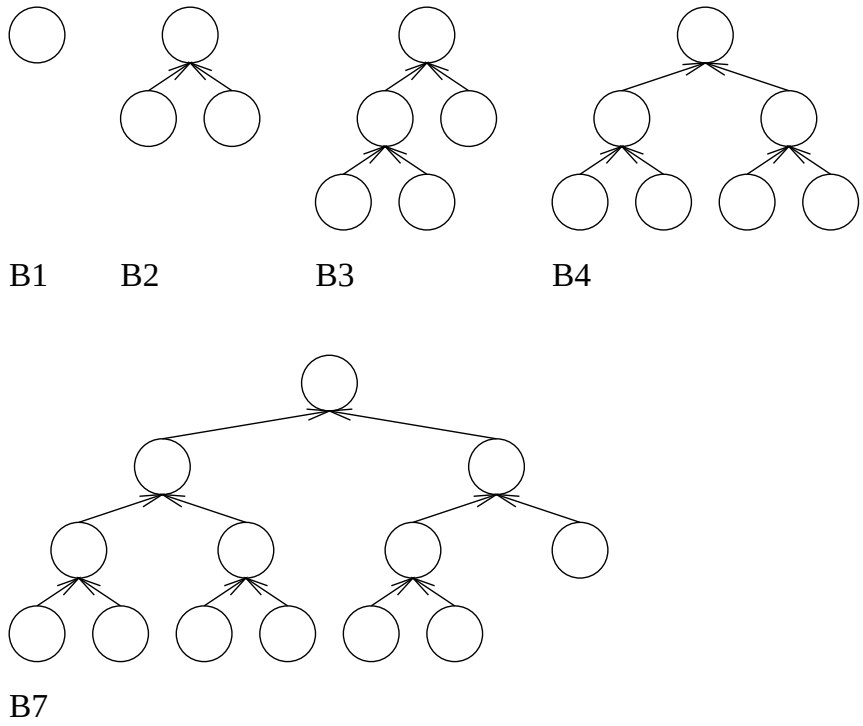


Abbildung 2.14: Bäume B_n für $n = 1, 2, 3, 4, 7$

Ausdruck $e(S)$ gibt, d.h. einen Ausdruck der zusätzlich die gleiche Struktur hat wie e. Das folgende Lemma zeigt, wann es korrespondierende Ausdrücke gibt.

Lemma 2.14 *Entfernt man in einem Schaltnetz S alle Inverter, deren eingehende Kante von einem Eingang kommt, sowie alle Eingänge, dann gibt es einen zu S korrespondierenden Ausdruck genau dann, wenn das modifizierte Schaltnetz S' ein Baum ist.*

Man entfernt neben den Eingängen auch die mit ihnen verbundenen Inverter, da in einem Boole'schen Ausdruck nicht nur Variablen, sondern auch invertierte Variablen beliebig oft vorkommen können. Beim korrespondierenden Ausdruck gilt $L(e(S)) = \mathrm{C}(S)$, beziehungsweise $L(e(S)) = \mathrm{C}(S) + c$, wenn k invertierte Variablen in e insgesamt $k + c$ mal verwendet werden. Auch hier beschränkt man also die Strukturgleichheit auf die Gatter mit zwei Eingängen. Ist das verbleibende Schaltnetz kein Baum, dann wird ein Gatterausgang mehrfach als Eingang anderer Gatter benutzt, was sich in einem Boole'schen Ausdruck nicht widerspiegeln lässt.

Existiert zu einem Schaltnetz S kein korrespondierender Ausdruck, dann existiert ein *quasi-korrespondierender Ausdruck* e'. Diesen findet man, indem man das Schaltnetz S so „umbaut“, dass es nach Lemma 2.14 einen korrespondierenden Ausdruck zu dem umgebauten Schaltnetz gibt. Der Umbau von S erfolgt dadurch, dass man, vom Ausgang (der bei einem Schaltnetz mit einem Ausgang auch die einzige Senke darstellt) rückwärts startend, stets Gatter, die mehrfach benutzt werden, samt dem Anfangsschaltnetz, das in ihnen endet, entsprechend oft repliziert. Da man in jedem Schritt Gatter mit geringerer

Tiefe bearbeitet, und in einem Schaltnetz keine Zyklen vorkommen, terminiert dieses Verfahren. Allerdings gilt dann natürlich $L(e') > \mathrm{C}(S)$.

Was man bereits vermutet, stimmt tatsächlich: man kann Funktionen angeben bei denen die Formelgröße, d.h. die Anzahl der Operatoren in der kürzesten Formel, echt größer ist als die Schaltnetzkomplexität, also die Anzahl der Gatter des billigsten Schaltnetzes. Die Konstruktion ist aufwändig, man findet sie zum Beispiel in Kapitel 3.5 von Keller/Paul: Hardware Design.

2.4 Darstellungen für ganze Zahlen

Zahlen sind abstrakte Konstruktionen der Mathematik. Um über Zahlen sprechen zu können, müssen Menschen sie in eine Form bringen, die man Zahlendarstellung nennt. Eine sehr einfache Darstellung ist die *unäre* Darstellung, bei der jeder natürlichen Zahl z eine Reihe von z Strichen entspricht. Formal wird eine n-stellige unäre Darstellung so definiert, dass die von einer Zeichenreihe $a_{n-1}, \ldots, a_0 \in \{0,1\}^n$ dargestellte Zahl gerade $k \in \{0, \ldots, n-1\}$ ist, wenn $a_k = 1$ und $a_j = 0$ für $j \neq k$. Hat die Zeichenreihe eine andere Form (entweder Null oder mehr als ein Bit gesetzt), dann ist die dargestellte Zahl nicht definiert. Alternativ könnte man die n-stellige unäre Darstellung der Zahl k auch durch $n-k$ Nullen, gefolgt von k Einsen definieren. Diese alternative Definition verträgt sich allerdings nicht gut mit der Definition der Decoder und Coder in Abschnitt 2.5.2. Die folgende Tabelle zeigt die 4-stelligen unären Darstellungen und die durch sie dargestellten Zahlen.

unäre Darstellung

$a_3a_2a_1a_0$	k
0001	0
0010	1
0100	2
1000	3

Allgemein üblich ist heute die Verwendung von *Stellenwertsystemen*. Aus der Schulzeit bekannt ist das Dezimalsystem mit der Ziffernmenge $Z_{10} = \{0, 1, 2, \ldots, 9\}$. Eine Zahl, genauer gesagt ihre n-stellige Darstellung im Dezimalsystem, besteht dann aus einer Folge $a = a_{n-1}, \ldots, a_0 \in Z_{10}^n$ und die dargestellte Zahl z berechnet sich zu

$$z = \sum_{i=0}^{n-1} a_i \cdot 10^i \ . \tag{2.3}$$

Die Zahl 10 heißt die *Basis* des Dezimalsystems. Man schreibt für Gleichung (2.3) auch abkürzend

$$z = \langle a_{n-1}, \ldots, a_0 \rangle_{10} \ .$$

Es hat sich als Artefakt der Übertragung aus dem Arabischen durchgesetzt, die Folge $a = (a_i)_{i=0\ldots n-1}$ so zu schreiben, dass die höchstwertige Stelle a_{n-1} links und die Stelle a_0 mit der niedrigsten Wertigkeit rechts steht.

Da Schaltnetze nur mit den Werten 0 und 1 rechnen, benötigt man auch Zahlendarstellungen, die auf der Ziffernmenge $Z_2 = \{0, 1\}$ basieren. Die *n-stellige Binärdarstellung* einer Zahl z ist eine Folge $a = a_{n-1}, \ldots, a_0 \in Z_2^n$ mit

$$z = \langle a \rangle_2 = \sum_{i=0}^{n-1} a_i \cdot 2^i \ . \tag{2.4}$$

Ist klar, welche Basis zur Anwendung kommt, so kann man den Index auch weglassen.

Beispiel 2.8 *Es ist* $\langle 1001 \rangle_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9$. *Dieser Wert ist aber verschieden von* $\langle 1001 \rangle_{10}$.

Lemma 2.15 *Die Abbildung* $\langle \ \ \rangle_2 : Z_2^n \to \{0, \ldots, 2^n - 1\} =: R_n$ *ist eine Bijektion, d.h. zu jeder Zahl* z *aus* R_n *gibt es genau eine n-stellige Binärzahl* a *mit* $z = \langle a \rangle_2$.

Beweis: Die Mengen Z_2^n und R_n haben die gleiche Mächtigkeit 2^n. Damit muss lediglich gezeigt werden, dass die Abbildung $\langle \ \ \rangle_2$ injektiv ist, woraus unter obiger Voraussetzung die Bijektivität folgt. Man führt den Beweis durch die Annahme des Gegenteils, nämlich dass $\langle \ \ \rangle_2$ nicht injektiv sei, und führt diese Annahme zum Widerspruch.

Wäre die Abbildung nicht injektiv, dann gäbe es zwei Darstellungen $a, a' \in Z_2^n$ mit $a \neq a'$, so dass $\langle a \rangle_2 = \langle a' \rangle_2$. Sei j der größte Index, so dass $a_j \neq a'_j$, wobei ohne Beschränkung der Allgemeinheit $a_j < a'_j$, also $a_j - a'_j = -1$. Dann würde aus

$$\langle a \rangle - \langle a' \rangle = \sum_{i=0}^{n-1} (a_i - a'_i) \cdot 2^i = 0$$

folgen, dass

$$\sum_{i=0}^{j-1} (a_i - a'_i) \cdot 2^i = 2^j \ .$$

Andererseits gilt aber für $i < j$ dass $a_i - a'_i \leq 1$, so dass die linke Seite durch $\sum_{i=0}^{j-1} 2^i = 2^j - 1$ nach oben abgeschätzt werden kann, was zu einem Widerspruch führt.

■

Die Umkehrabbildung zu $\langle \ \ \rangle_2$, die also jeder Zahl $z \in R_n$ ihre n-stellige Binärdarstellung zuweist, heißt bin_n. Um die Binärdarstellung zu finden, dividiert man z durch 2^{n-1}, das Ergebnis ist a_{n-1}. Mit dem Rest der Division verfährt man analog und erhält so nacheinander auch a_{n-2} bis a_0. Um für eine beliebige Zahl $z \in \mathbf{N}_0$ eine möglichst kurze Binärdarstellung zu finden, bestimmt man zunächst das kleinste n mit $z \in R_n$, d.h. man sucht das kleinste n mit $z < 2^n$. Man erhält

$$n = \lceil \log_2(z + 1) \rceil$$

wobei die Funktion $\lceil \ \ \rceil$ zur nächsten Ganzzahl aufrundet. Dann bestimmt man $\text{bin}_n(z)$ wie oben.

Tabelle 2.3: Liste von Zweierpotenzen

i	2^i
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Selbsttestaufgabe 2.7 *Bestimmen Sie die kürzeste Binärdarstellung für* $z = 90$.

Lösung auf Seite 88

Man merkt hier, dass es hilfreich ist, die gängigsten Zweierpotenzen zu kennen. Tabelle 2.3 listet einige Zweierpotenzen. Alternativ kann man sich die gewünschte Zweierpotenz auch durch fortgesetztes Verdoppeln und Abzählen des Exponenten mit den Fingern herleiten.

Manchmal ist es schwer, sich die Größenordnung von Zweierpotenzen im Dezimalsystem vorzustellen. Eine Daumenregel nutzt die Tatsache, dass $2^{10} = 1024 \approx 1000 = 10^3$. Die Zahl $2^{33} = 2^{10} \cdot 2^{10} \cdot 2^{10} \cdot 2^3$ ist demnach ungefähr $10^3 \cdot 10^3 \cdot 10^3 \cdot 8 = 8 \cdot 10^9$, also 8 Milliarden.

Einige Zusammenhänge bei Binärdarstellungen gibt das folgende Lemma an:

Lemma 2.16 *Sind a und b n-stellige bzw. m-stellige Binärdarstellungen von $z \in \mathbf{N}_0$, wobei $m > n$ ist, dann gilt $b = \underbrace{0, \ldots, 0}_{m-n}, a$.*

Ist $a_{n-1}, \ldots, a_0 \in Z_2^n$ und ist $1 \leq l \leq n-1$, dann gilt

$$\langle a_{n-1}, \ldots, a_0 \rangle = \langle a_{n-1}, \ldots, a_l \rangle \cdot 2^l + \langle a_{l-1}, \ldots, a_0 \rangle; .$$

Als Spezialfall gilt: Ist $a \in Z_2^{n-l}$ und $z \in \mathbf{N}_0$ mit $z = \langle a \rangle$, dann ist $\left\langle a, \underbrace{0, \ldots, 0}_{l} \right\rangle = z \cdot 2^l$.

Beispiel 2.9 *Das Lemma besagt, dass man Nullen voranstellen darf, ohne dass man den Wert verändert, also* $\langle 1001 \rangle_2 = 9 = \langle 01001 \rangle_2$. *Fügt man hingegen am Ende eine Null an, dann verdoppelt sich der Wert, also* $\langle 10010 \rangle_2 = 16 + 2 = 18 = 2 \cdot \langle 1001 \rangle_2$. *Teilt man eine Binärzahl in zwei Teile, dann müssen die höherwertigen Bits entsprechend gewichtet werden:* $\langle 1001 \rangle_2 = \langle 10 \rangle_2 \cdot 2^2 + \langle 01 \rangle_2 = 2 \cdot 4 + 1 = 9$.

Will man auch negative Zahlen darstellen, dann gibt es drei Möglichkeiten: Betrag und Vorzeichen, Einser-Komplement, und Zweier-Komplement.

Bei der Darstellung mit Betrag und Vorzeichen hat man n Stellen um den Betrag der Zahl darzustellen, und eine weitere Stelle, die das Vorzeichen angibt: 0 entspricht hierbei einem positiven Vorzeichen, 1 entspricht einem negativen Vorzeichen.

Beim Einser-Komplement gibt es ebenfalls ein zusätzliches Bit, das als Vorzeichen dient. Die restlichen n Stellen stellen aber nur im positiven Fall den Betrag der darzustellenden Zahl dar. Im negativen Fall ($a_n = 1$) ist die darzustellende Zahl

$$-2^n + 1 + \langle a_{n-1}, \ldots, a_0 \rangle \ .$$

In der Regel wird das Zweier-Komplement benutzt. Hier gilt

$$z = [a_n, \ldots, a_0]_2 := -a_n \cdot 2^n + \langle a_{n-1}, \ldots, a_0 \rangle \ , \tag{2.5}$$

wobei gilt:

$$[\quad]_2 : Z_2^{n+1} \to \{-2^n, \ldots, 2^n - 1\} \ .$$

Die Funktion $[\quad]$ ist, wie man leicht einsieht, bijektiv. Das Bit a_n ist ebenfalls ein Vorzeichenbit. Im Falle $a_n = 0$ ist die Zahl z also nicht-negativ, und die restlichen n Stellen repräsentieren die Zahl. Im Falle $a_n = 1$ ist die Zahl z negativ, und die restlichen Stellen bilden die Zahl, die zu -2^n addiert wird, um die eigentliche Zahl zu bilden.

Sucht man also die $(n+1)$-stellige Zweier-Komplement-Darstellung für eine positive Zahl z, so bildet man zunächst $\mathrm{bin}_n(z)$ und setzt eine Null davor. Sucht man die entsprechende Darstellung für eine negative Zahl z, so bildet man $z + 2^n$, was eine nicht-negative Zahl ist, berechnet $\mathrm{bin}_n(z + 2^n)$, und setzt davor eine Eins.

Beispiel 2.10 *Um die Zahl $z = -9$ im Zweier-Komplement darzustellen, wählen wir $n = 4$. Wir bilden $z + 2^n = -9 + 2^4 = 7$ und bestimmen* $\mathrm{bin}_4(7) = 0111$. *Dann setzen wir noch eine 1 davor. Es ist* $[10111]_2 = -16 + 7 = -9$.

Die Korrektheit dieses Verfahrens ergibt sich aus Gleichung (2.5). Ist $z < 0$, so muss $a_n = 1$ sein, damit die Gleichung eine Lösung haben kann. Durch Addition von 2^n auf beiden Seiten der Gleichung ergibt sich

$$z + 2^n = \langle a_{n-1}, \ldots, a_0 \rangle \ ,$$

was wegen der Bijektivität von $\langle \quad \rangle$ gleichbedeutend ist mit

$$\mathrm{bin}_n(z + 2^n) = a_{n-1}, \ldots, a_0 \ .$$

Alternativ kann man die Zweier-Komplement-Darstellung einer negativen Zahl z finden, indem man die Zweier-Komplement-Darstellung der positiven Zahl $|z|$ bildet, dann alle Bits einschließlich des Vorzeichenbits invertiert, die Darstellung kurz als Binärdarstellung interpretiert, und 1 dazuaddiert. Die Addition von Binärzahlen werden wir in Abschnitt 2.6 betrachten.

Die Zweier-Komplement-Darstellung unterscheidet sich vom Einser-Komplement und der Darstellung mit Betrag und Vorzeichen dadurch, dass es keine

doppelte Darstellung der Null mit positivem und negativem Vorzeichen gibt, und dass der Zahlenbereich unsymmetrisch ist. Es können alle Zahlen von -2^n bis $+2^n - 1$ dargestellt werden.

Die Umkehrabbildung von $[\ \ ]$, die einer Zahl ihre Darstellung im Zweier-Komplement zuordnet, bezeichnen wir auch mit $\mathrm{twoc}(z)$.

Selbsttestaufgabe 2.8 *Bestimmen Sie die* 8*-stellige Zweier-Komplement-Darstellung für* $z = -116$*. Bestimmen Sie* $[110001]_2$ *und* $[01001]_2$*.*

Lösung auf Seite 89

Will man eine Zweier-Komplement-Darstellung um $l \in \mathbf{N}$ Bit verlängern, dann gilt

$$[a_n, \ldots, a_0] = [\underbrace{a_n, \ldots, a_n}_{l}, a_n, \ldots, a_0] \ . \tag{2.6}$$

Hier muss man also im Gegensatz zu Binärzahlen, bei denen Nullen vorangestellt wurden, das Vorzeichenbit vervielfachen. Bei positiven Zahlen ist allerdings kein Unterschied festzustellen.

Selbsttestaufgabe 2.9 *Zeigen Sie durch Verwendung der Gleichungen (2.4) und (2.5) die Korrektheit von Gleichung (2.6).*

Lösung auf Seite 89

Binary Coded Decimals

Neben den Binär– und Zweier-Komplement-Darstellungen gibt es auch noch die sogenannten *Binary Coded Decimals (BCD)*. Bei diesen wird eine Dezimaldarstellung Stelle für Stelle binär kodiert. Für jede Dezimalstelle werden 4 Bit benötigt.

Wie in Gleichung (2.4) kann man auch Darstellungen zu anderen Basen bilden. Bekannt sind hier das *Oktalsystem* zur Basis 8 mit den Ziffern 0 bis 7 und das *Hexadezimalsystem* zur Basis 16 mit den Ziffern $0, \ldots, 9$,A,...,F, wobei die Buchstaben für die Ziffern mit den Dezimalwerten 10 bis 15 stehen. Will man eine Binärdarstellung in eine dieser Darstellungen umrechnen, macht man sich zunutze, dass die Basen dieser Systeme Potenzen der Basis 2 der Binärdarstellung sind. So gilt

$$\begin{aligned} z &= \langle a_{4n-1}, \ldots, a_0 \rangle_2 = \sum_{i=0}^{4n-1} a_i \cdot 2^i \\ &= \sum_{j=0}^{n-1} (a_{4j+3} \cdot 2^3 + a_{4j+2} \cdot 2^2 + a_{4j+1} \cdot 2^1 + a_{4j} \cdot 2^0) \cdot 2^{4j} \\ &= \sum_{j=0}^{n-1} \langle a_{4j+3}, \ldots, a_{4j} \rangle_2 \cdot 16^j \\ &= \langle \langle a_{4n-1}, \ldots, a_{4n-4} \rangle_2, \ldots, \langle a_3, \ldots, a_0 \rangle_2 \rangle_{16} \ , \end{aligned}$$

das heißt man erhält die Hexadezimaldarstellung dadurch, dass man jeweils vier Stellen der Binärdarstellung zusammenfasst.

Beispiel 2.11 *Es ist* $\langle 10011011 \rangle_2 = \langle 9B \rangle_{16} = 9 \cdot 16 + 11 = 155$*, da* $\langle 1001 \rangle_2 = 9$ *und* $\langle 1011 \rangle_2 = 11$*.*

2.5 Häufig benutzte Schaltnetze

Während es schon für kleine n sehr viele verschiedene n-stellige Schaltfunktionen und damit Schaltnetze gibt, gibt es einige Schaltnetze, die man häufig benötigt. In der Regel dienen Sie zur gesteuerten Auswahl aus einer Menge von Signalen, oder zur gesteuerten Auswahl aus einer Vielzahl von Ausgangsmöglichkeiten für ein Signal. Da einige der Schaltnetze Zahlendarstellungen verwenden, können wir sie erst jetzt vorstellen.

Wenn die Anzahl der Eingangs– und Ausgangsvariablen einer Schaltfunktion größer wird, ist die Wertetafel keine günstige Spezifikationsgrundlage mehr. Dies gilt insbesondere, wenn man eine Spezifikation parametrisieren will. Wie man in diesem Fall vorgeht zeigt die nun folgende Definition 2.5.

2.5.1 Multiplexer und Demultiplexer

Ein Multiplexer ist ein Schaltnetz, das mittels des Werts einer Steuerleitung einen von zwei Eingängen auf einen Ausgang weiterleitet. Dabei kann ein solcher Eingang durchaus mehr als ein Bit umfassen. Dies lässt sich mit folgender Definition formalisieren.

Definition 2.5 *Ein n-Bit* Multiplexer *(MUX_n) ist ein Schaltnetz, das die folgende Funktion $m : \{0,1\}^{2n+1} \to \{0,1\}^n$ berechnet:* Multiplexer

$$m(a^0_{n-1}, \ldots, a^0_0, a^1_{n-1}, \ldots, a^1_0, s) = \begin{cases} (a^1_{n-1}, \ldots, a^1_0) & \textit{falls} \quad s = 1 \\ (a^0_{n-1}, \ldots, a^0_0) & \textit{falls} \quad s = 0 \; . \end{cases}$$

Wenn wir für $i = 0, 1$ die Abkürzungen $a^i = a^i_{n-1}, \ldots, a^i_0$ vereinbaren, dann können wir die obige Formel auch kürzer schreiben:

$$m(a^0, a^1, s) = a^s \; .$$

Es wird gerade der n-bit Vektor ausgewählt, dessen oberer Index dem Wert des Steuersignals s entspricht. Diese Abkürzung werden wir nochmals benutzen, wenn wir weiter unten Multiplexer definieren, die aus mehr als zwei Eingängen auswählen.

Ein n-Bit Multiplexer kann durch das Schaltnetz in Abbildung 2.15(a) realisiert werden. Wenn $s = 1$ ist, dann leitet das jeweilige linke AND-Gatter wegen $a^1_j \wedge 1 = a^1_j$ das Signal weiter, und das jeweilige rechte AND-Gatter sperrt wegen $a^0_j \wedge \bar{s} = a^0_j \wedge 0 = 0$. Im Falle $s = 0$ ist es gerade umgekehrt. Das OR-Gatter erhält also jeweils ein Eingangssignal a^i_j und eine Null als Eingänge. Wegen $x \vee 0 = 0 \vee x = x$ leitet es dieses Eingangssignal an den Ausgang.

Wir werden im Weiteren das Schaltsymbol aus Abbildung 2.15(b) benutzen. Hierbei symbolisiert der waagerechte, mit n markierte Strich an den Eingängen, dass es sich hierbei eigentlich um ein Bündel aus n Leitungen handelt.

Aus Abbildung 2.15(a) ergeben sich Kosten und Tiefe eines n-Bit Multiplexers zu

$$\mathrm{C}(\mathrm{MUX}_n) = 3n + 1 \text{ und } \mathrm{T}(\mathrm{MUX}_n) = 3 \; .$$

Analog kann für $t \in \mathbf{N}$ ein 2^t-Wege n-Bit Multiplexer $\mathrm{MUX}_{n,2^t}$ definiert werden als Schaltnetz das aus 2^t Eingängen, von denen jeder ein n-Bit Vektor

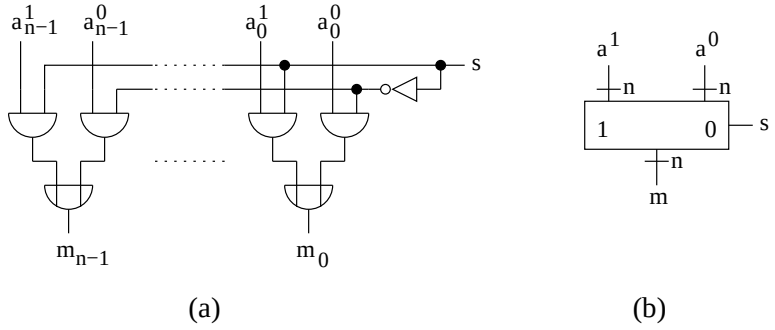


Abbildung 2.15: Schaltnetz und Symbol für Multiplexer

ist, einen auswählt nach Maßgabe eines Steuersignals das in diesem Fall die Binärdarstellung einer Zahl zwischen 0 und $2^t - 1$ ist.

Ein 2-Wege n-Bit Multiplexer ist dabei gerade der eben definierte n-Bit Multiplexer. Für $t \geq 2$ definieren wir den 2^t-Wege n-Bit Multiplexer als Schaltnetz zur Berechnung der Funktion $m : \{0,1\}^{2^t n + t} \to \{0,1\}^n$ mit

$$m(a^0, \ldots, a^{2^t-1}, s_{t-1}, \ldots, s_0) = a^{\langle s_{t-1}, \ldots, s_0 \rangle} .$$

Dabei sind die $a^i \in \{0,1\}^n$ für $i \in \{0, \ldots, 2^t - 1\}$ gerade die 2^t n-Bit Vektoren, und s ist das t-Bit Steuersignal, das als t-stellige Binärdarstellung einer Zahl aus dem Bereich 0 bis $2^t - 1$ interpretiert wird und so den betreffenden Eingang selektiert.

Selbsttestaufgabe 2.10 *Zeigen Sie, dass ein 2^t-Wege n-Bit Multiplexer als balancierter binärer Baum aus $2^t - 1$ vielen n-Bit Multiplexern konstruiert werden kann, wenn man nur die Datenleitungen betrachtet. Berechnen Sie die Kosten* $\mathrm{C}(\mathrm{MUX}_{n,2^t})$ *und Tiefe* $\mathrm{T}(\mathrm{MUX}_{n,2^t})$ *dieses Schaltnetzes.*

Lösung auf Seite 89

Ein Schaltnetz das genau das Umgekehrte erreicht ist ein Demultiplexer. Er hat einen Dateneingang (der aus einem n-Bit Vektor besteht) und 2^t Ausgänge (ebenfalls zu je n Bit), und leitet die Signale des Dateneingangs auf den Ausgang, der durch das Steuersignal angegeben wird. Wir formalisieren dies in der folgenden Definition.

Definition 2.6 *Ein 2^t-Wege n-Bit* Demultiplexer *ist ein Schaltnetz zur Berechnung der Funktion*

$$dm : \{0,1\}^{n+t} \to \{0,1\}^{2^t n}$$

mit

$$dm(b, s_{t-1}, \ldots, s_0) = (a^0, \ldots, a^{2^t-1})$$

wobei

$$a^i = \begin{cases} b & \text{falls } i = \langle s_{t-1}, \ldots, s_0 \rangle \\ 0 & \text{sonst.} \end{cases}$$

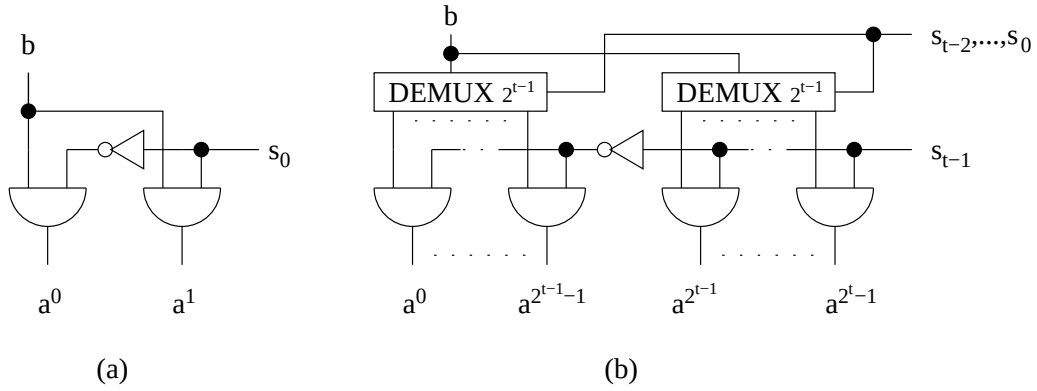


Abbildung 2.16: Konstruktion von Demultiplexern

Man gibt also mit den Steuerleitungen s an, zu welchem Ausgangsvektor a^i der Eingangsvektor b geleitet werden soll. Abbildung 2.16 zeigt für $n = 1$ die Konstruktion eines 2-Wege Demultiplexers und die rekursive Konstruktion eines 2^t-Wege Demultiplexers aus zwei 2^{t-1}-Wege Demultiplexern. Unter Rekursion wollen wir hier verstehen, dass wir bei einem Schaltnetz für einen 2^t-Wege Demultiplexer Schaltnetze für die 2^{t-1}-Wege Demultiplexer bestimmen und dann in die Konstruktion von Abbildung 2.16(b) einsetzen. Die Schaltnetze für die 2^{t-1}-Wege Demultiplexer erhalten wir auf die gleiche Weise, es sei denn $t = 2$, dann ist $t - 1 = 1$, es folgt $2^{t-1} = 2^1 = 2$, und wir können das Schaltnetz aus Abbildung 2.16(a) einsetzen.

Wir verzichten an dieser Stelle auf einen formalen Beweis der Korrektheit und stellen nur folgende Überlegung an. Beim 2-Wege Demultiplexer leitet im Falle $s_0 = 1$ das rechte AND-Gatter die Eingabe weiter, und das linke AND-Gatter sperrt. Im Falle $s_0 = 0$ ist es umgekehrt. Beim 2^t-Wege Demultiplexer sperren im Falle $s_{t-1} = 1$ die linke Hälfte der AND-Gatter und die rechte Hälfte der AND-Gatter leitet die Ausgaben des rechten 2^{t-1}-Wege Demultiplexers weiter. Im Falle $s_{t-1} = 0$ ist es umgekehrt. In diesem Fall ist wegen

$$\langle 0, s_{t-2}, \ldots, s_0 \rangle = \langle s_{t-2}, \ldots, s_0 \rangle$$

offensichtlich, dass bei korrekter Funktionsweise des linken 2^{t-1}-Wege Demultiplexers der Eingang auf den richtigen Ausgang geleitet wird. Bei $s_{t-1} = 1$ wird der j-te Ausgang des rechten 2^{t-1}-Wege Demultiplexers auf den Ausgang $2^{t-1} + j$ geleitet. Wegen $\langle 1, s_{t-2}, \ldots, s_0 \rangle = 2^{t-1} + \langle s_{t-2}, \ldots, s_0 \rangle$ ist auch dies korrekt.

2.5.2 Decoder und Coder

Ein Decoder wandelt eine t-stellige binäre Zahlendarstellung in eine unäre Zahlendarstellung um. Bilden also die t Eingangssignale die Binärdarstellung der Zahl i aus dem Bereich 0 bis $2^t - 1$, dann erhält Ausgang i den Wert 1 und alle anderen Ausgänge den Wert 0. Wir formalisieren dies in folgender Definition.

Definition 2.7 *Ein t-Bit* Decoder *ist ein Schaltnetz zur Berechnung der Funktion $dc : \{0,1\}^t \to \{0,1\}^{2^t}$ mit*

$$dc(s_{t-1}, \ldots, s_0) = (a_{2^t-1}, \ldots, a_0)$$

Decoder

wobei

$$a_i = \begin{cases} 1 & \textit{falls } i = \langle s_{t-1}, \ldots, s_0 \rangle \\ 0 & \textit{sonst.} \end{cases}$$

Decoder-Schaltnetze finden zum Beispiel bei der Realisierung von Speichermatrizen Verwendung. Man findet Sie auch in endlichen Automaten (s. Kurseinheiten 3 und 4).

Selbsttestaufgabe 2.11 *Zeigen Sie dass ein t-Bit Decoder aus einem 2^t-Wege 1-Bit Demultiplexer konstruiert werden kann, indem der Eingang b fest mit dem Wert 1 belegt wird.*

Lösung auf Seite 90

Ein Coder oder Encoder ist das Gegenstück eines Decoders. Er wandelt eine unäre Zahlendarstellung in eine binäre Zahlendarstellung um.

Coder

Definition 2.8 *Ein* signalisierender t-Bit Coder *oder* Encoder *ist ein Schaltnetz zur Berechnung der Schaltfunktion $cd : \{0,1\}^{2^t} \to \{0,1\}^{t+2}$ mit*

$$cd(a_{2^t-1}, \ldots, a_0) = (b_{t+1}, \ldots, b_0) \ ,$$

wobei

$$\langle b_{t-1}, \ldots, b_0 \rangle = \begin{cases} i & \textit{wenn } a_i = 1 \textit{ und alle } a_j = 0 \textit{ für alle } j \neq i \\ \textit{beliebig} & \textit{sonst.} \end{cases}$$

Im ersten Fall sind $b_{t+1} = b_t = 0$. Sind alle $a_i = 0$, so ist $b_{t+1} = 0$ und $b_t = 1$. Sonst ist $b_{t+1} = 1$ und der Wert von b_t beliebig.

Hier haben wir also ein Beispiel einer Schaltfunktion, die nur auf einer Teilmenge ihres möglichen Definitionsbereichs $\{0,1\}^{2^t}$, nämlich den unären Zahlendarstellungen, etwas Sinnvolles tun kann. Ansonsten wird lediglich durch das Ausgangssignal b_{t+1} eine Warnung ausgegeben, dass mehr als ein Bit in der Eingabe gesetzt ist, bzw. durch b_t eine Warnung ausgegeben, dass kein Bit der Eingabe gesetzt ist. Ein Coder wandelt nämlich nur eine gültige unäre Darstellung in eine Binärdarstellung um, sonstige Muster führen zur Warn-Ausgabe. Die Bedeutung des extra Signals b_t, das anzeigt, dass die Eingabe nur aus Nullen besteht, wird aus der Definition nicht ganz ersichtlich, hat aber in der Realisierung eine wichtige Rolle, wie man gleich sieht.

Man kann einen t-Bit Coder aus zwei $(t-1)$-Bit Codern, einem t-Bit Multiplexer sowie etwas Logik zur Erzeugung des Warnsignals b_{t+1} und des Nullflags b_t konstruieren wie in Abbildung 2.17(b) gezeigt. Abbildung 2.17(a) zeigt die Konstruktion eines 1-Bit Coders. Wir haben also wie beim Demultiplexer eine rekursive Konstruktion.

Man kann die Korrektheit des 1-Bit Coders unmittelbar einsehen: er liefert ein korrektes Resultat ohne Fehlerflag nur bei den Eingaben 01 oder 10. In

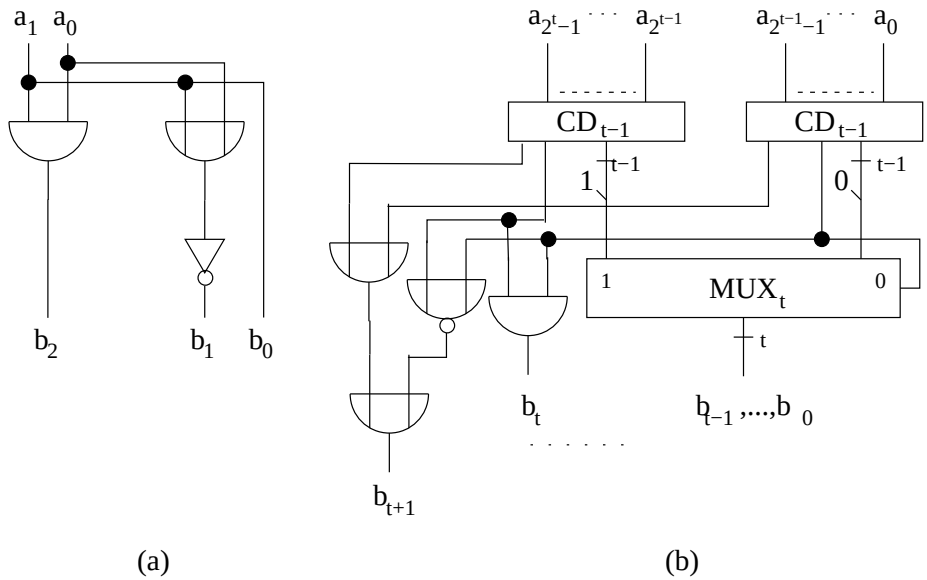


Abbildung 2.17: Konstruktion von Codern

diesem Falle entspricht die 1-stellige Binärcodierung gerade dem Wert von a_1. Haben beide Eingänge den Wert 0, dann wird das Nullflag gesetzt. Im letzten verbleibenden Fall, wenn beide Eingänge den Wert 1 haben, wird das Warnflag gesetzt; der Wert von b_0 ist in diesem Falle egal.

Zur Korrektheit des t-Bit Coders muss man ein wenig mehr Aufwand treiben. Eine unäre Eingangscodierung aus 2^t Bits liegt genau dann vor, wenn die eine Hälfte der Eingänge eine unäre Eingangscodierung darstellt, und die andere Hälfte der Bits alle den Wert 0 haben. In diesem Fall steuert das Nullflag der unteren Hälfte der Eingänge den Multiplexer, denn wenn diese Hälfte aus Nullen besteht, dann enthält die obere Hälfte der Eingänge eine gültige unäre Kodierung. Enthält keine der Hälften eine gültige Kodierung, so ist die Auswahl egal.

Ist die untere Hälfte der Eingänge eine unäre Codierung, so ergibt sich die t-stellige Binärausgabe als die um eine Null verlängerte $(t-1)$-stellige Binärausgabe des unteren $(t-1)$-bit Coders gemäß Lemma 2.16. Ist hingegen die obere Hälfte der Eingabe eine unäre Codierung, so ergibt sich die korrekte Binärausgabe dadurch, dass zum Wert 2^{t-1} addiert wird, also $b_{t-1} = 1$ gesetzt wird. Die Erweiterung der Bündel aus $t-1$ Leitungen um ein Signal 0 bzw. 1 zu Bündeln aus t Leitungen wird dabei in der Schaltnetz-Zeichnung durch den schrägen Strich mit Markierung 0 bzw. 1 am Leitungsbündel gekennzeichnet.

Das Nullflag hat dann den Wert 1, wenn beide Nullflags der $(t-1)$-bit Coder den Wert 1 haben. Das Warnflag hat dann den Wert 1, wenn entweder eines der Warnflags der $(t-1)$-bit Coder den Wert 1 hat, oder wenn keines der beiden Nullflags gesetzt ist. Im letzteren Fall enthalten nämlich beide Hälften der Eingänge je ein gesetztes Bit.

Tabelle 2.4: Funktionstabellen von Halb– und Volladdierer

a_0	b_0	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a)

a_0	b_0	c	s_1	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b)

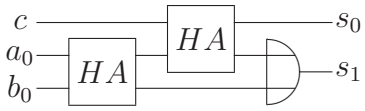


Abbildung 2.18: Aufbau eines Volladdierers aus Halbaddierern

Selbsttestaufgabe 2.12 *Konstruieren Sie einen 3-Bit Coder und einen 1-Bit 4-Wege Demultiplexer.*

Lösung auf Seite 90

2.6 Schaltnetze für Ganzzahl-Arithmetik

Für Binärzahlen kann man ähnlich wie für Dezimalzahlen die Grundrechenarten definieren. Für die Addition einstelliger Binärzahlen gilt: $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, $1 + 1 = 10$. Das Schaltnetz, das dieses Verhalten implementiert, nennt man *Halbaddierer (HA)* (engl. half adder), weil er zwar einen Ausgangsübertrag produzieren, aber keinen Eingangsübertrag verarbeiten kann. Ein solches Schaltnetz nennt man *Volladdierer (VA)* (engl. full adder, FA). Die Wertetafeln für beide Schaltnetze sind in Tabelle 2.4 zu sehen. Die Wertetafel stellt eine *Spezifikation*, d.h. eine Beschreibung des Ein– und Ausgabeverhaltens eines Schaltnetzes dar. Diese ist zu unterscheiden von der konkreten Realisierung des Schaltnetzes. Bereits für den Halbaddierer gibt es mehrere Möglichkeiten, die davon abhängen, ob man EXOR-Gatter erlaubt oder nicht. Für den Volladdierer gibt es ebenfalls mehrere Möglichkeiten, darunter eine, die aus zwei Halbaddierern und einem ODER-Gatter besteht (s. Abb. 2.18). Die Schaltsymbole von Halb– und Volladdierer sind in Abbildung 2.19 zu sehen.

Halbaddierer

Volladdierer

Im Allgemeinen wird es stets mehrere Möglichkeiten zur Realisierung eines bestimmten Ein–/Ausgabeverhaltens geben, die sich in den Kosten und der Tiefe unterscheiden. Welche Möglichkeit gewählt wird, hängt dann von den Einsatzbedingungen ab. Wir werden im Weiteren $\mathrm{C}(HA) = 2$, $\mathrm{T}(HA) = 1$, $\mathrm{C}(FA) = 5$ und $\mathrm{T}(FA) = 3$ verwenden.

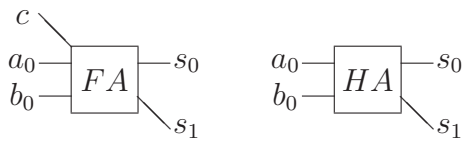


Abbildung 2.19: Symbole für Halb– und Volladdierer

Definition 2.9 *Ein n-Bit Addierer, d.h. ein Addierer für n-stellige Binärzahlen, ist ein Schaltnetz, das folgende Funktion* $add : Z_2^{2n+1} \to Z_2^{n+1}$ *berechnet:*

$$add(a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, c_0) = c_n, s_{n-1}, \ldots, s_0$$

mit

$$\langle c_n, s_{n-1}, \ldots, s_0 \rangle = \langle a_{n-1}, \ldots, a_0 \rangle + \langle b_{n-1}, \ldots, b_0 \rangle + c_0 \ .$$

Hierbei heißen $a = a_{n-1}, \ldots, a_0$ *und* $b = b_{n-1}, \ldots, b_0$ *die* Summanden*,* c_0 *der* Eingangsübertrag*,* $s = s_{n-1}, \ldots, s_0$ *die* Summe *und* c_n *der* Ausgangsübertrag.

Eingangsübertrag

Ausgangsübertrag

Wir haben mit dieser Definition unendlich viele Schaltnetze spezifiziert, ohne allerdings einen Hinweis darauf zu geben, wie diese aussehen könnten.

Im Folgenden wollen wir deshalb zwei Schaltnetze angeben, die die obige Definition 2.9 erfüllen.

2.6.1 Carry–Chain Addierer

Sobald man das kleine Einsundeins mit Überträgen beherrscht, kann man zumindest mit Papier und Bleistift lange Zahlen a und b nach der Schulmethode addieren. Das geht bei Binärzahlen genauso wie bei Dezimalzahlen. Man schreibt die Zahlen untereinander und arbeitet die Stellen von rechts nach links ab. Für jede Stelle i addiert man den Übertrag c_{i-1} von der vorherigen Stelle und die Ziffern a_i und b_i der Summanden ($c_{-1} = 0$). Bei der Addition von Binärzahlen kann man das durch Nachsehen in der Tabelle 2.4(b) (Wertetabelle des Volladdierers) tun. Von dem zweistelligen Ergebnis (c_i, s_i) schreibt man die hintere Stelle als Teil des Gesamtergebnisses hin und behält die vordere Stelle c_i als Übertrag für die nächste Stelle 'im Sinn'. Ein Beispiel hierfür ist

c	-	1	1	0	-
a	-	1	0	1	1
b	-	0	1	1	0
s	1	0	0	0	1

Wir spezifizieren nun Schaltnetze, die genau dieses Verfahren durchführen. Dass man mit dem Verfahren tatsächlich korrekt addiert[4], wird im Beweis von Satz 2.17 implizit mitbewiesen.

Satz 2.17 *Das Schaltnetz* A_1 *bestehe aus einem einzigen Volladdierer* FA*. Für* $n > 1$ *entstehe das Schaltnetz* A_n *aus* A_{n-1} *und* FA *wie in Abbildung 2.20 angegeben. Dann ist für alle* n *das Schaltnetz* A_n *ein n–Bit Addierer.*

[4]Für Dezimalzahlen haben wir in der Grundschule gelernt, das zu glauben.

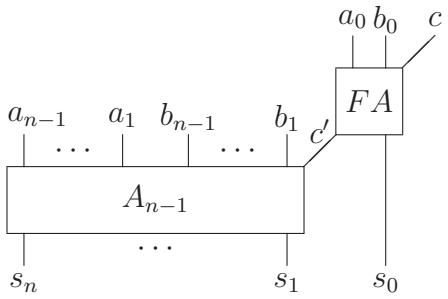


Abbildung 2.20: Rekursiver Aufbau von A_n

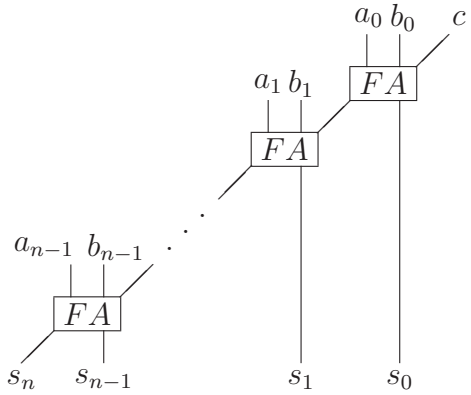


Abbildung 2.21: Aufbau eines Carry–Chain Addierers

Carry–Chain Addierer

Löst man die in Satz 2.17 angegebene Rekursion auf, so erhält man einen Addierer wie in Abbildung 2.21. Einen solchen Addierer nennt man *Carry–Chain Addierer* oder *Carry–Ripple Addierer*, da der Übertrag die Kette aller Volladdierer durchläuft.

Beweis von Satz 2.17: Wir führen den Beweis durch Induktion über n.

$n = 1$**:** In diesem Fall ist A_n einfach ein Volladdierer, also nach Definition ein 1–Bit Addierer.

$n - 1 \to n$**:** Sei die Eingabe $(a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, c)$ mit

$$\langle a_{n-1}, \ldots, a_0 \rangle + \langle b_{n-1}, \ldots, b_0 \rangle + c = S \; .$$

Wir zeigen, dass das Schaltnetz A_n den Wert $(s_n, \ldots, s_0)$ ausgibt mit $\langle s_n, \ldots, s_0 \rangle = S$.

Nach Definition des Volladdierers berechnet FA den Wert (c', s_0) mit

$$\langle c', s_0 \rangle = a_0 + b_0 + c \ .$$

Man beachte, dass $\langle c', s_0 \rangle = 2 \cdot c' + s_0$ gilt. Nach Induktionsvoraussetzung berechnet A_{n-1} den Wert $(s_n, \ldots, s_1)$ mit

$$\langle s_n, \ldots, s_1 \rangle = \langle a_{n-1}, \ldots, a_1 \rangle + \langle b_{n-1}, \ldots, b_1 \rangle + c' \ .$$

Es folgt

$$\begin{array}{rcll} S & = & \langle a_{n-1}, \ldots, a_0 \rangle + \langle b_{n-1}, \ldots, b_0 \rangle + c & \\ & = & 2^1 \cdot \langle a_{n-1}, \ldots, a_1 \rangle + 2^1 \cdot \langle b_{n-1}, \ldots, b_1 \rangle + a_0 + b_0 + c & \text{nach L. 2.16} \\ & = & 2^1 \cdot \langle a_{n-1}, \ldots, a_1 \rangle + 2^1 \cdot \langle b_{n-1}, \ldots, b_1 \rangle + 2 \cdot c' + s_0 & \\ & = & 2^1 \cdot (\langle a_{n-1}, \ldots, a_1 \rangle + \langle b_{n-1}, \ldots, b_1 \rangle + c') + s_0 & \\ & = & 2^1 \cdot \langle s_n, \ldots, s_1 \rangle + s_0 & \text{nach Vor.} \\ & = & \langle s_n, \ldots, s_0 \rangle & \text{nach L. 2.16} \end{array}$$

■

Die Kosten eines n–Bit Carry–Chain Addierers A_n betragen

$$\mathrm{C}(A_n) = n \cdot \mathrm{C}(FA) = 5n \ ,$$

die Tiefe beträgt

$$\mathrm{T}(A_n) \leq n \cdot \mathrm{T}(FA) = 3n \ .$$

Für $i \in 0, \ldots, n-1$ sei c_i der Übertrag zwischen Volladdierer i und dem Volladdierer $i+1$ im Carry-Chain Addierer falls $i < n-1$ und der Ausgangsübertrag des Carry-Chain Addierers falls $i = n-1$. Dies ist nichts anderes als der Übertrag von Stelle i nach Stelle $i+1$.

Da die Kette der Volladdierer $i, \ldots, 0$ einen $(i+1)$-Bit Addierer bildet, gilt für alle i:

$$\langle a_i, \ldots, a_0 \rangle + \langle b_i, \ldots, b_0 \rangle + c = \langle c_i, s_i, \ldots, s_0 \rangle \ . \tag{2.7}$$

Es folgt

$$c_i = 1 \Leftrightarrow \langle a_i, \ldots, a_0 \rangle + \langle b_i, \ldots, b_0 \rangle + c \geq 2^{i+1} \ .$$

Selbsttestaufgabe 2.13 *Bestimmen Sie die Kosten und die Tiefe eines 4-Bit Carry-Chain Addierers.*

Lösung auf Seite 90

Wir werden im Folgenden häufig die Addierer der vorigen Abschnitte als Bausteine für weitere Schaltnetze verwenden. Für das Zeichnen dieser Schaltnetze treffen wir die folgende Verabredung: Wir verwenden für n-Bit Addierer mit Eingängen $a = a_{n-1}, \ldots, a_0$, $b = b_{n-1}, \ldots, b_0$ und c und Ausgängen $s = s_n, \ldots, s_0$ das Symbol aus Abbildung 2.22.

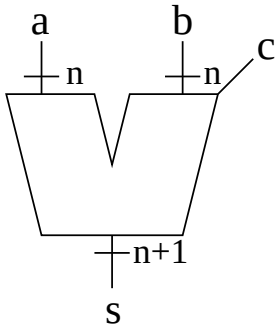


Abbildung 2.22: Symbolvereinbarung

2.6.2 Conditional–Sum Addierer

Wir fragen nun, ob es vielleicht billigere Addierer oder Addierer mit geringerer Tiefe als den Carry-Chain Addierer gibt. Ein einfaches Argument zeigt, dass man die Kosten jedenfalls nicht unter $2n$ senken kann. Der Ausgangsübertrag s_n hängt von allen $2n+1$ Argumenten ab. Also muß jedes Schaltnetz, das die Funktion $+_n$ berechnet einen Baum B mit Ausgang s_n enthalten, der mit allen Eingängen verbunden ist. In diesem Baum müssen $2n$ Gatter vorkommen.

Ein einfacher Induktionsbeweis zeigt, dass jeder binäre Baum mit Tiefe t höchstens 2^t Blätter hat. Hieraus folgt, dass die Tiefe des Baums B mindestens $\log n+1$ sein muß. Also gilt für die Tiefe von n-Bit Addierern $\mathrm{T}(A_n) = \Omega(\log n)$.

Es ist verlockend zu argumentieren, dass jeder n-Bit Addierer mindestens die Tiefe n haben muß, „*weil der Übertrag ja über alle n Stellen laufen muß*“. Dies ist allerdings ein Irrtum. Der Übertrag zwischen zwei Stellen kann nur den Wert 0 oder 1 annehmen. Man kann also ab diesem Punkt mit zwei Varianten arbeiten: eine mit Eingangsübertrag 0 und eine mit Eingangsübertrag 1. Hat man den tatsächlichen Übertrag berechnet, so kann man das richtige der beiden Ergebnisse auswählen. Die Auswahl erledigt ein Multiplexer.

Sei n gerade, und es sei $A_{n/2}$ ein $(n/2)$-Bit Addierer. Dann kann man aus drei Addierern $A_{n/2}$ und einem $(n/2+1)$-Bit Multiplexer das Schaltnetz A_n aus Abbildung 2.23 konstruieren. Wir zeigen, dass A_n ein Addierer ist.

Im Schaltnetz A_n werden die höherwertigen Bits $a_h = a_{n-1}, \ldots, a_{n/2}$ und $b_h = b_{n-1}, \ldots, b_{n/2}$ der Operanden in zwei $(n/2)$-Bit Addierer gegeben, einmal mit Eingangsübertrag 0 und einmal mit Eingangsübertrag 1. Von diesen Addierern berechnet einer die höherwertigen Bits $s_h = s_n, \ldots, s_{n/2}$ der Summe, falls der Übertrag c' von Stelle $n/2-1$ nach Stelle $n/2$ gleich 0 ist, der andere berechnet die höherwertigen Bits der Summe falls $c' = 1$ gilt.

Die niederwertigen Bits $a_l = a_{n/2-1}, \ldots, a_0$ und $b_l = b_{n/2-1}, \ldots, b_0$ werden mit Eingangsübertrag c in einen dritten $(n/2)$-Bit Addierer gegeben. Dieser Addierer berechnet die niederwertigen Bits $s_l = s_{n/2-1}, \ldots, s_0$ des Ergebnisses und den Übertrag c'. Die Auswahl der höherwertigen Bits des Ergebnisses erfolgt durch den Multiplexer, kontrolliert durch Signal c'.

Für Zweierpotenzen n definieren wir: das Schaltnetz A_1 sei ein einzelner Volladdierer FA. Für $n > 1$ entstehe A_n aus $A_{n/2}$ wie in Abbildung 2.23 angegeben.

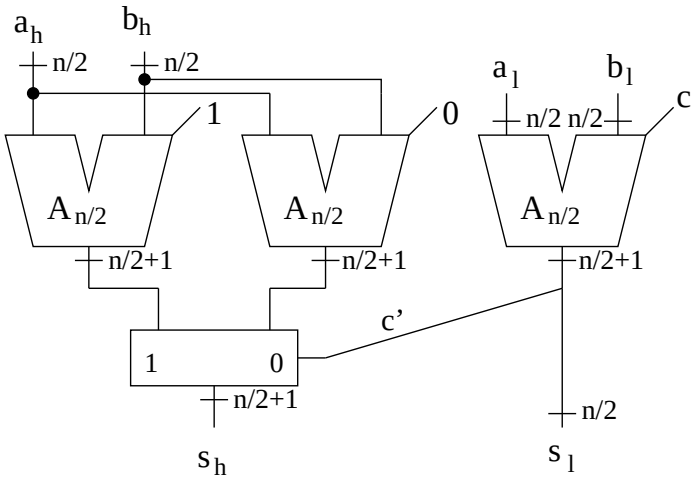


Abbildung 2.23: Aufbau eines Conditional-Sum Addierers

Die hierdurch definierten n-Bit Addierer heißen *Conditional-Sum Addierer*. Wir bezeichnen ihre Kosten und Tiefe mit

Conditional–Sum Addierer

$$\begin{aligned} c(n) &= \mathrm{C}(A_n) \text{ und} \\ t(n) &= \mathrm{T}(A_n) \end{aligned}$$

Aus den Definitionen folgt direkt

$$\begin{aligned} c(1) &= \mathrm{C}(FA) = 5 \\ c(n) &= 3 \cdot c(n/2) + \mathrm{C}(\mathrm{MUX}_{n/2+1}) \\ &= 3 \cdot c(n/2) + 3n/2 + 4 \text{ für } n > 1 \end{aligned} \tag{2.8}$$

und

$$\begin{aligned} t(1) &= \mathrm{T}(FA) = 3 \\ t(n) &= t(n/2) + \mathrm{T}(\mathrm{MUX}_{n/2+1}) \\ &= t(n/2) + 3 \text{ für } n > 1\,. \end{aligned} \tag{2.9}$$

Man nennt Gleichungen der Form (2.8) und (2.9) *Differenzengleichungen*. Wir lösen zuerst das System (2.9). Hierzu müssen wir eine geschlossene Formel für $t(n)$ raten und ihre Korrektheit durch Induktion beweisen. Für Differenzengleichungen gibt es eine einfache Methode, um solche Formeln zu raten. Man wendet die Differenzengleichung mehrfach auf sich selbst an, im obigen Beispiel etwa

Differenzengleichung

$$\begin{aligned} t(n) &= t(n/2) + 3 \\ &= t(n/4) + 3 + 3 \\ &= t(n/8) + 3 + 3 + 3 \\ &\vdots \end{aligned}$$

und hofft, dass man etwas sieht. Im obigen Beispiel sieht man

$$t(n) = t(n/2^k) + k \cdot 3$$

für alle k mit $n/2^k \geq 1$. Für $k = \log n$ erhält man für $t(n)$ die geschlossene Formel

$$t(n) = t(1) + 3\log n = 3 + 3\log n \ .$$

Dass wir hiermit tatsächlich die Lösung des Systems (2.9) geraten haben, zeigt ein einfacher Induktionsbeweis, denn es ist

$$t(1) = \mathrm{T}(FA) = 3 + 3\log 1$$

und für $n > 1$ ist

$$\begin{aligned} t(n) &= t(n/2) + 3 \\ &= 3 + 3\log(n/2) + 3 \qquad \text{nach Ind.Vor.} \\ &= 3 + 3 \cdot (\log n - 1) + 3 \\ &= 3 + 3\log n \ . \end{aligned}$$

.

Damit haben wir

Satz 2.18 *n-Bit Conditional-Sum Addierer haben Tiefe $O(\log n)$.*

Zwar könnte man den Satz auch formulieren „Conditional-Sum Addierer haben Tiefe $3 + 3\log n$“, aber das Interessanteste an diesem Ergebnis ist, dass die Tiefe tatsächlich nur logarithmisch in n ist, und nicht linear wie beim Carry-Chain Addierer. Asymptotisch betrachtet haben wir nach den obigen Ausführungen zu Bäumen und Tiefe ein Optimum erreicht.

Das Gleichungssystem (2.8) lösen wir nach dem gleichen Rezept, wir müssen nur etwas mehr rechnen. Das System hat die Form

$$f(n) = a \cdot f(n/b) + g(n), f(1) = c \ .$$

Wir betrachten nur den Fall $n = b^k$. Mehrfaches Einsetzen der Differenzengleichung in sich selbst liefert

$$\begin{aligned} f(n) &= g(n) + a \cdot f(n/b) \\ &= g(n) + a \cdot g(n/b) + a^2 \cdot f(n/b^2) \\ &\ \ \vdots \\ &= \sum_{i=0}^{j-1} a^i \cdot g(n/b^i) + a^j \cdot f(n/b^j) \end{aligned}$$

für $j \leq k$. Einsetzen von $j = k$ und $f(1) = c$ liefert die Induktionsbehauptung des folgenden Lemmas, die man auf keinen Fall auswendig lernen sollte. Es ist viel leichter sich zu merken, wie man sie herleitet.

Lemma 2.19 *Sei $f : \mathbf{N} \to \mathbf{N}$ eine Funktion mit $f(1) = c$ und $f(n) = a \cdot f(n/b) + g(n)$ für alle Potenzen $n = b^k$ von b. Dann gilt*

$$f(n) = a^{\log_b n} \cdot c + \sum_{i=0}^{\log_b n - 1} a^i \cdot g(n/b^i)$$

für alle Potenzen n von b.

Beweis durch Induktion über k:

$k = 0$**:** Für $n = b^0 = 1$ ist $\log_b n = 0$ und damit $f(1) = a^0 \cdot c = c$.

$k \to k+1$**:** Für $n = b^{k+1}$ ist $\log_b n = k+1$. Wir zeigen, dass

$$f(n) = a^{k+1} \cdot c + \sum_{i=0}^{k} a^i \cdot g(b^{k+1-i})$$

gilt. Per Definition ist

$$f(n) = f(b^{k+1}) = a \cdot f(b^k) + g(b^{k+1}) \; .$$

Aus der Induktionsvoraussetzung folgt

$$\begin{aligned} f(b^{k+1}) &= a \cdot \left(a^k \cdot c + \sum_{i=0}^{k-1} a^i \cdot g(b^{k-i}) \right) + g(b^{k+1}) \\ &= a^{k+1} \cdot c + \sum_{i=1}^{k} a^i \cdot g(b^{k-i+1}) \; + g(b^{k+1}) \\ &= a^{k+1} \cdot c + \sum_{i=0}^{k} a^i \cdot g(b^{k+1-i}) \; . \end{aligned}$$

■

Mit Lemma 2.19 lassen sich auch die Kosten von Conditional-Sum Addierern bestimmen. Wir setzen

$$a = 3, \; b = 2, \; c = 5, \; g(n) = \mathrm{C}(\mathrm{MUX}_{n/2+1}) = (3/2) \cdot n + 4$$

und erhalten

$$c(n) = 3^{\log n} \cdot 5 + \sum_{i=0}^{\log n - 1} 3^i \cdot \left(\frac{3}{2} \cdot \frac{n}{2^i} + 4 \right) \; .$$

Wir treiben Potenzrechnung: für positive reelle Zahlen x gilt:

$$\begin{aligned} x^{\log n} &= (2^{\log x})^{\log n} \\ &= 2^{(\log x \cdot \log n)} \\ &= (2^{\log n})^{\log x} \\ &= n^{\log x} \; . \end{aligned} \tag{2.10}$$

Der erste Summand läßt sich so umformen zu

$$5 \cdot n^{\log 3} \approx 5 \cdot n^{1.585} \; .$$

Die Summe spalten wir in zwei Teilsummen auf:

$$(3n/2) \cdot \sum_{i=0}^{\log n - 1} (3/2)^i \text{ und } 4 \cdot \sum_{i=0}^{\log n - 1} 3^i \; .$$

Jede der Teilsummen bildet eine geometrische Reihe (vergleiche Abschnitt 1.1). Mit der dort errechneten Formel $\sum_{i=0}^{n-1} x^i = (x^n - 1)/(x - 1)$ ergibt sich

$$\begin{aligned} c(n) &= 5n^{\log 3} + \frac{3n}{2} \cdot \frac{(3/2)^{\log n} - 1}{(3/2) - 1} + 4 \cdot \frac{3^{\log n} - 1}{3 - 1} \\ &= 5n^{\log 3} + 3n^{\log 3} - 3n + 2n^{\log 3} - 2 \\ &= 10n^{\log 3} - 3n - 2 \; . \end{aligned}$$

Der Conditional-Sum Addierer hat damit zwar eine geringe Tiefe, aber hohe Kosten.

Selbsttestaufgabe 2.14 *Bestimmen Sie die Kosten und die Tiefe eines 4-Bit Conditional-Sum Addierers.*

Lösung auf Seite 90

2.6.3 Multiplizierer

Seien $a \in \{0,1\}^n$ und $b \in \{0,1\}^m$. Dann ist $0 \leq \langle a \rangle < 2^n$ und $0 \leq \langle b \rangle < 2^m$. Für das Produkt $z = \langle a \rangle \cdot \langle b \rangle$ gilt dann $0 \leq z < 2^{n+m}$, also hat z eine $(n+m)$-stellige Binärdarstellung.

Multiplizierer

Definition 2.10 *Ein* (n,m)-Multiplizierer *ist ein Schaltnetz zur Berechnung der Funktion* $mul^{n,m} : \{0,1\}^{n+m} \to \{0,1\}^{n+m}$ *mit*

$$\begin{aligned} mul^{n,m}(a_{n-1}, \ldots, a_0, b_{m-1}, \ldots, b_0) &= s_{n+m-1}, \ldots, s_0 \text{ mit} \\ \langle s_{n+m-1}, \ldots, s_0 \rangle &= \langle a_{n-1}, \ldots, a_0 \rangle \cdot \langle b_{m-1}, \ldots, b_0 \rangle \; . \end{aligned}$$

Wir nennen (n,n)*-Multiplizierer einfach* n*-Bit Multiplizierer.*

Für die Multiplikation von n-stelligen Dezimalzahlen a mit m-stelligen Dezimalzahlen b lernt man in der Grundschule das folgende Verfahren:

1. Für alle $i \in \{0, \ldots, m-1\}$ multipliziert man den Multikand a mit jeder Stelle b_i des Multiplikators und schiebt das Ergebnis d_i um i Stellen nach links (was einer stillschweigenden Multiplikation mit 10^i entspricht.)
2. Man addiert die Summanden $d_i \cdot 10^i$ auf.

Ein Beispiel findet man in Beispiel 2.12(a). Wenn man nicht weiß, wie man mehrere Summanden $d_i \cdot 10^i$ in einem Schritt zusammenaddiert, kann man sich immer noch helfen: man erzeugt durch einfache Additionen sukzessive

$$\begin{aligned} D_1 &= d_0 + d_1 \cdot 10^1 \\ D_2 &= D_1 + d_2 \cdot 10^2 \text{ usw.} \end{aligned}$$

Ein Beispiel für dieses Verfahren findet man in Beispiel 2.12(b).

Das zweite Verfahren läßt sich sofort auf Binärzahlen übertragen. Das Problem, das kleine Einmaleins auswendig zu lernen entfällt, da man für die Erzeugung der Summanden d_i nur mit Stellen $b_i \in \{0,1\}$ multiplizieren muß. Ein Beispiel findet man in Beispiel 2.12(c).

Beispiel 2.12 **(a)** *Wir multiplizieren zwei Dezimalzahlen* $a = 348$ *und* $b = 529$*. Dann ergibt sich*

$$\begin{aligned} d_0 &= 348 \cdot 9 = 3132 \\ d_1 &= 348 \cdot 2 = 696 \\ d_2 &= 348 \cdot 5 = 1740 \end{aligned}$$

Durch aufsummieren der Summanden erhält man das Produkt $a \cdot b$*:*

$$\begin{array}{r} 3132 \\ 696 \\ 1740 \\ \hline 184092 \end{array}$$

(b) *Wir wählen* a *und* b *wie im Teil (a). Anstatt alle Summanden zu addieren, errechnen wir schrittweise:*

$$\begin{aligned} D_1 &= 3132 + 696 \cdot 10^1 = 10092 \\ D_2 &= 10092 + 1740 \cdot 10^2 = 184092 \end{aligned}$$

(c) *Wir multiplizieren zwei Binärzahlen* $a = 110$ *und* $b = 101$ *nach der Methode aus Teil (b). Wir erhalten* $d_0 = d_2 = 110$ *und* $d_1 = 000$*. Um das Produkt zu erhalten, berechnen wir*

$$\begin{aligned} D_1 &= 110 + 0000 = 110 \\ D_2 &= 110 + 11000 = 11110 \end{aligned}$$

Die Schaltnetze $M^{n,m}$, die wir im Folgenden konstruieren, multiplizieren genau auf die eben beschriebene Weise. Wir bezeichnen ihre Kosten und Tiefe mit $c(n,m)$ und $t(n,m)$.

Für $m = 1$ gilt

$$mul^{n,1}(a,1) = a \text{ und } mul^{n,1}(a,0) = 0 \ .$$

Dieser Multiplizierer bildet eine Ausnahme, da für das Produkt eine n-stellige anstatt der erwarteten $(n+1)$-stelligen Binärdarstellung genügt. Bei einer Darstellung als Schaltsymbol verwenden wir trotzdem $n+1$ Ausgänge, von denen der oberste stets den Wert 0 hat. Ein solcher Multiplizierer läßt sich durch n AND-Gatter realisieren:

$$m_i = a_i \wedge b_0 \text{ für } 0 \leq i < n, m_n = 0 \ .$$

Es gilt somit

$$c(n,1) = n \text{ und } t(n,1) = 1 \ . \tag{2.11}$$

Für $m > 1$ gilt wegen Lemma 2.16

$$\begin{aligned} \langle a \rangle \cdot \langle b_{m-1}, \ldots, b_0 \rangle &= \langle a \rangle \cdot (\langle b_{m-1}, \ldots, b_1 \rangle \cdot 2 + b_0) \\ &= \langle a \rangle \cdot \langle b_{m-1}, \ldots, b_1 \rangle \cdot 2 \ + \langle a \rangle \cdot b_0 \ . \end{aligned}$$

Eine weitere Anwendung von Lemma 2.16 liefert

$$\langle x_{p-1}, \ldots, x_0\rangle \cdot 2 = \langle x_{p-1}, \ldots, x_0, 0\rangle, \tag{2.12}$$

Also braucht man für die Multiplikation mit Zwei keine Gatter, und man kann einen (n, m)-Multiplizierer $M^{n,m}$ wie in Abbildung 2.24 gezeigt konstruieren aus

- einem $(n, m-1)$-Multiplizierer $M^{n,m-1}$,
- einem $(n, 1)$-Multiplizierer, d. h. n AND-Gattern, und
- einem $(n+m)$-Bit Addierer, den wir mit A^m bezeichnen.

Wir verwenden als Addierer A^m einen $(m+n)$-Bit Carry-Chain Addierer und erhalten für die Kosten die Abschätzung

$$\begin{aligned} c(n,m) &= c(n,m-1) + (n+m)\cdot \mathrm{C}(FA) + n \\ &= c(n,m-1) + 5\cdot(n+m) + n \\ &= c(n,m-1) + 5m + 6n\ . \end{aligned} \tag{2.13}$$

Aus (2.11) und (2.13) folgt durch Induktion über m sofort

$$\begin{aligned} c(n,m) &= mn + \sum_{i=2}^{m} 5(n+i) \\ &= mn + 5n(m-1) + 5\cdot\left(\sum_{i=1}^{m} i\right) - 5\ . \end{aligned}$$

Die Formel für $1 + \cdots + m$ haben wir natürlich vergessen, weil es leichter ist, sich ihre Herleitung zu merken:

$$\begin{array}{rcccccc} & 1 & + & 2 & + & \cdots & + & m \\ + & m & + & m-1 & + & \cdots & + & 1 \\ = & m\cdot(m+1)\ , & & & & & & \end{array}$$

also

$$\sum_{i=1}^{m} i = m\cdot(m+1)/2\ . \tag{2.14}$$

Es folgt

$$\begin{aligned} c(n,m) &= mn + 5n(m-1) + \frac{5}{2}m(m+1) - 5 \\ &= 6mn + \frac{5}{2}m^2 + \frac{5}{2}m - 5n - 5 \end{aligned} \tag{2.15}$$

Für $m = n$ haben wir damit Kosten $O(n^2)$.

Für die Tiefe dieser Multiplizierer liest man aus Abbildung 2.24 direkt die folgende Abschätzung ab:

$$t(n,m) \le t(n,m-1) + (n+m)\cdot \mathrm{T}(FA)\ .$$

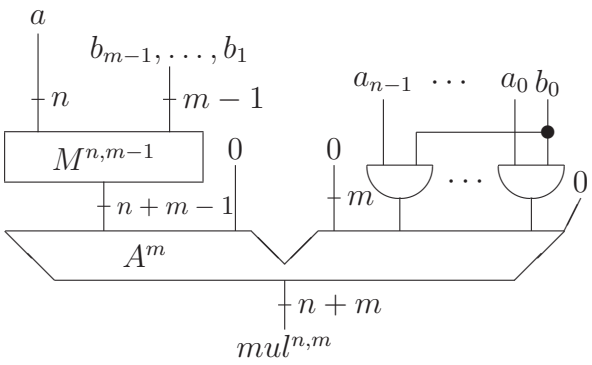


Abbildung 2.24: Rekursive Definition eines einfachen (n, m)-Multiplizierers

Wie oben folgt

$$\begin{aligned} t(n,m) &\leq 1 + \sum_{i=2}^{m} 3(n+i) \\ &= 1 + 3n(m-1) + \frac{3}{2}m(m+1) - 3 \\ &= 3mn + \frac{3}{2}m^2 + \frac{3}{2}m - 3n - 2 \,. \end{aligned}$$

Diese Abschätzung ist zwar korrekt, aber viel zu grob. Wir betrachten noch einmal Abbildung 2.24 und machen die folgenden Beobachtungen:

1. Jeder Pfad von einem Eingang zu einem Ausgang berührt genau eines der AND-Gatter der $(m, 1)$-Multiplizierer.

2. Für $i \in \{2, \ldots, m\}$ und $j \in \{0, \ldots, n+i-1\}$ bezeichne FA^i_j den Volladdierer an Stelle j im $(n+i)$-Bit Carry-Chain Addierer A^i. Dann gibt es von FA^i_j höchstens Kanten zu den folgenden Volladdierern:

   (a) FA^i_{j+1} falls $j \neq n+i-1$ und

   (b) FA^{i+1}_{j+1} falls $i \neq m$.

Zeichnen wir nur die Volladdierer FA^i_j des Multiplizierers und ihre Verbindungen untereinander, so ergibt sich das Gitter aus Abbildung 2.25. Durch Induktion über j folgt nun sofort die Behauptung:
Ist v ein Gatter in Volladdierer FA^i_j, dann ist

$$\mathrm{T}(v) \leq (j+1)\mathrm{T}(FA) + 1 \,.$$

Hieraus folgt

$$t(n,m) \leq (m+n) \cdot \mathrm{T}(FA) + 1 = 3m + 3n + 1 \,.$$

Die Kosten der obigen Konstruktion kann man leicht verbessern. In jedem der Addierer A^i hat der Volladdierer FA^i_0 zwei Eingänge, die immer Null sind, und bei den i Volladdierern FA^i_n bis FA^i_{n+i} ist ein Eingang stets Null. Diese

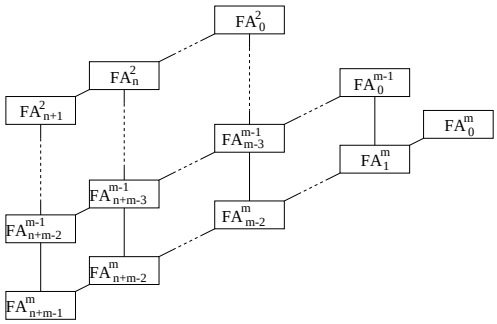


Abbildung 2.25: Aufbau der Volladdierer im einfachen Multiplizierer

Volladdierer kann man in dem Schema von Abbildung 2.25 durch Halbaddierer ersetzen, bzw. im Fall des FA_0^i sogar weglassen. Dadurch spart man in jedem der $m-1$ Carry Chain Addierer A^i Kosten

$$i(\mathrm{C}(FA) - \mathrm{C}(HA)) + \mathrm{C}(FA) = 3i + 5 \; .$$

Für die Kosten $c'(n,m)$ der so modifizierten Multiplizierer gilt dann

$$\begin{aligned} c'(n,m) &= c(n,m) - \sum_{i=2}^{m}(3i+5) \\ &= 6mn + m^2 - 4m - 5n + 3 \; . \end{aligned}$$

Wir fassen die bisherigen Ergebnisse für $m = n$ zusammen in

Satz 2.20 *Die Implementierung der Schulmethode liefert n-Bit Multiplizierer mit Kosten $7 \cdot n^2 - 9 \cdot n + 3$ und Tiefe $6n + 1$.*

Wenn man die Kosten noch genauer abschätzen will, dann lässt man im $M^{n,2}$ den Volladdierer FA_{n+1}^2 ebenfalls weg, da das Multiplikationsergebnis des $M^{n,1}$ nur n Bit statt $n+1$ Bit umfasst, und der FA_{n+1}^2 somit zwei Eingänge mit dem Wert 0 hat. Man kann sogar noch weiter gehen: da man die Volladdierer FA_0^i weglässt, hat der Volladdierer FA_1^i einen Eingang mit dem Wert 0 und kann durch einen Halbaddierer ersetzt werden. Diese beiden zusätzlichen Maßnahmen werden im folgenden Beispiel illustriert.

Beispiel 2.13 *Ein $(4,2)$-Multiplizierer nach Schulmethode besteht aus zwei $(4,1)$-Multiplizierern, die $(a_3,\ldots,a_0)$ mit b_1 und b_0 multiplizieren, und einem 6-Bit Carry-Chain Addierer, der die beiden Teilergebnisse zusammenfasst. Die Kosten dieses Multiplizierers bestehen aus $2 \cdot 4 = 8$ AND-Gattern für die Teilergebnisse, und $6 \cdot \mathrm{C}(FA) = 30$ für die Bestimmung des Gesamtergebnisses. Nach den obigen Optimierungen können der unterste und der oberste Volladdierer weggelassen werden, und der zweitunterste und zweitoberste Volladdierer kann jeweils durch einen Halbaddierer ersetzt werden, wodurch sich eine Einsparung von 16 ergibt, so dass die Gesamtkosten 22 betragen.*

2.7 Darstellungen für rationale Zahlen

Zur Darstellung rationaler Zahlen gibt es zwei Möglichkeiten: *Festkommadarstellungen* und *Fließkommadarstellungen*. Festkommadarstellungen mit t Nachkommastellen kann man mit $(n+t)$-stelligen Dualzahlen vergleichen, bei denen allerdings der Wert mit 2^{-t} multipliziert wird. Der Wert einer Festkommazahl $a_{n-1}, \ldots, a_0.a_{-1}, \ldots, a_{-t}$ ist damit gegeben als

Fest–, Fließkommadarstellungen

$$\langle a_{n-1}, \ldots, a_0, a_{-1}, \ldots, a_{-t} \rangle = \sum_{i=-t}^{n-1} a_i \cdot 2^i \ .$$

Addierer für Festkommazahlen entsprechen damit Addierern für Dualzahlen.

Bei Fließkommadarstellungen versucht man, Zahlen in einem großen Zahlenbereich stets mit der gleichen Genauigkeit darzustellen. Fließkommazahlen sind standardisiert im IEEE Standard 754. Der Wert einer normalisierten Darstellung $(s, c, a) \in \{0,1\}^{1+m+n}$ mit Mantisse $a = a_{-1}, \ldots, a_{-n}$ und Charakteristik $c = c_{m-1}, \ldots, c_0$, $1 \leq \langle c \rangle \leq 2^m - 2$ sowie Vorzeichen s ist gegeben durch

$$(-1)^s \cdot \langle 1.a \rangle \cdot 2^{\langle c \rangle - \text{bias}} \ ,$$

wobei bias $= 2^{m-1} - 1$. Der Wert $2^m - 1$ der Charakteristik dient mit $\langle a \rangle = 0$ zur Darstellung von $+\infty$ und $-\infty$. Ist $\langle a \rangle \neq 0$, dann wird keine gültige Zahl dargestellt, der Standard nennt dies *NaN, not a number*. Der Wert 0 der Charakteristik dient zur Darstellung der *denormalisierten Zahlen*

$$(-1)^s \cdot \langle 0.a \rangle \cdot 2^{1 - \text{bias}} \ .$$

denormalisierte Zahl

Der IEEE Standard definiert eine einfache Genauigkeit (single precision) mit $n = 23$ und $m = 8$ (bias $= 127$) sowie eine doppelte Genauigkeit (double precision) mit $n = 52$ und $m = 11$ (bias $= 1023$). Damit ist die größte darstellbare Zahl in einfacher Genauigkeit

$$z_{max} = (-1)^0 \cdot \left\langle 1.\underbrace{1 \ldots 1}_{23} \right\rangle \cdot 2^{254-127} = (2 - 2^{-23}) \cdot 2^{127} \approx 2^{128} \ .$$

Die Zahl 0 hat eine denormalisierte Darstellung mit $c = 0$ und $a = 0$.

Gleichzeitig ist klar, dass nicht alle natürlichen Zahlen zwischen 0 und z_{max} dargestellt werden können. Jeder Wert einer Charakteristik definiert ein halboffenes Intervall $[2^{\langle c \rangle - 127} \ldots 2^{\langle c \rangle - 126})$, in dem 2^{23} verschiedene Zahlen dargestellt werden können. Ist zum Beispiel $\langle c \rangle = 151$, so sind die darstellbaren Zahlen $2^{24}, 2^{24} + 2, 2^{24} + 4, \ldots, 2^{25} - 2$.

Zum Rechnen definiert der IEEE Standard 754, dass eine Rechenoperation (Addition, Subtraktion, usw.) so durchzuführen ist, dass zunächst das exakte Ergebnis bestimmt wird, und dann zu einer darstellbaren Zahl gerundet wird. Hierzu sind vier Rundungsmodi definiert, zum Beispiel *Round towards zero*, bei dem immer zur nächsten betragsmäßig kleineren darstellbaren Zahl gerundet wird.

Rundungsmodus

Selbsttestaufgabe 2.15 *Bestimmen Sie das Ergebnis der Addition von* 2^{24} *und* 1 *in einfacher Genauigkeit, wenn als Rundungsmodus Round towards zero benutzt wird.*

Lösung auf Seite 91

2.8 Anhang: Sprechweisen für Notationen

2.8.1 Schaltnetze

Notation	Aussprache
$X_1 \oplus X_2$	X_1 exor X_2

2.8.2 Rechnen mit Schaltnetzen

Notation	Aussprache
$f \equiv_S g$	f ist bezüglich S äquivalent zu g f ist im Schaltnetz S äquivalent zu g

2.8.3 Schaltnetzkomplexität

Notation	Aussprache
$C(S)$	C von S, Kosten von S, Kosten des Schaltnetzes S
$T(S)$	T von S, Tiefe von S, Tiefe des Schaltnetzes S
$C(f)$	C von f, Schaltnetzkomplexität von f (wobei f Schaltfunktion ist)
$T(f)$	T von f, Tiefe von f
$\log n$	Log n, Logarithmus von n (zur Basis 2)

2.8.4 Darstellungen für ganze Zahlen

Notation	Aussprache
$\langle a_{n-1}, ..., a_0 \rangle_{10}$	die im Dezimalsystem durch die Ziffernfolge $a_{n-1}, ..., a_0$ dargestellte Zahl
$\langle a_{n-1}, ..., a_0 \rangle_2$	die im Binärsystem durch die Ziffernfolge $a_{n-1}, ..., a_0$ dargestellte Zahl
$\text{bin}_n(z)$	die n-stellige Binärdarstellung von z

2.8.5 Häufig benutzte Schaltnetze

Notation	Aussprache
MUX_n	Mux n, n-Bit-Multiplexer
$\text{MUX}_{n,2^t}$	2^t Wege-n-Bit-Multiplexer
DEMUX_{2^t}	2^t-Wege-Demultiplexer
CD_{t-1}	$t-1$-Bit-Coder

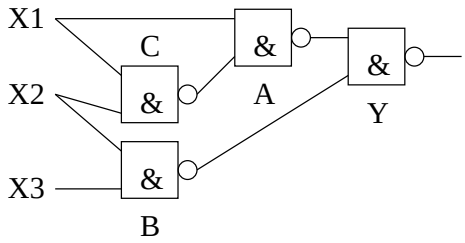


Abbildung 2.26: Schaltnetz zu Aufgabe 2.1

Tabelle 2.5: Berechnete Werte im Schaltnetz aus Abbildung 2.26

i	$\phi_i(X_1)$	$\phi_i(X_2)$	$\phi_i(X_3)$	$\phi_i(B)$	$\phi_i(C)$	$\phi_i(A)$	$\phi_i(Y_1)$
1	0	0	0	1	1	1	0
2	0	0	1	1	1	1	0
3	0	1	0	1	1	1	0
4	0	1	1	0	1	1	1
5	1	0	0	1	1	0	1
6	1	0	1	1	1	0	1
7	1	1	0	1	0	1	0
8	1	1	1	0	0	1	1

2.9 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 2.1 von Seite 46

Das Schaltnetz ist in Abbildung 2.26 dargestellt.

Selbsttestaufgabe 2.2 von Seite 48

Wir berechnen für jede der acht möglichen Belegungen der Eingangssignale die Werte der Ausgänge aller Gatter nach aufsteigender Tiefe. Hierbei haben B und C die Tiefe 1, A die Tiefe 2 und Y_1 die Tiefe 3. Das Ergebnis ist in Tabelle 2.5 zu sehen.

Selbsttestaufgabe 2.3 von Seite 49

Es gilt $B = \overline{X_2 \wedge X_3}$ bzw. $\bar{B} = X_2 \wedge X_3$. Außerdem ist $C = \overline{X_1 \wedge X_2}$. Nun ist $A = \overline{X_1 \wedge C}$ bzw. $\bar{A} = X_1 \wedge C = X_1 \wedge \overline{X_1 \wedge X_2} = X_1 \wedge \bar{X}_2$. Schließlich ist $Y = \overline{A \wedge B} = \bar{A} \vee \bar{B} = X_1 \wedge \bar{X}_2 \vee X_2 \wedge X_3$.

Selbsttestaufgabe 2.4 von Seite 53

Wir konstruieren zunächst vier Schaltnetze für die Terme X_1X_2, X_1X_3, X_2X_3 und $X_4 \vee X_5$, wobei die ersten drei Schaltnetze nur ein AND-Gatter enthalten, das letzte Schaltnetz hingegen ein OR-Gatter. Dann verbinden wir die ersten

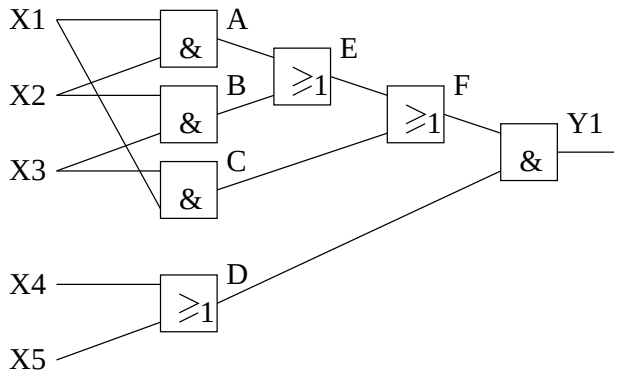


Abbildung 2.27: Schaltnetz zu Selbsttestaufgabe 2.4

drei Schaltnetze mittels zweier OR-Gatter, und schließlich verbinden wir diese mit dem letzten Schaltnetz des letzten Terms über ein AND-Gatter. Das resultierende Schaltnetz ist in Abbildung 2.27 zu sehen.

Selbsttestaufgabe 2.5 von Seite 55

Die Kosten des Schaltnetzes aus Abbildung 2.9(b) ergeben sich als Summe der Kosten der Schaltnetze S_1 und S_2. Die Kosten des Schaltnetzes S_2 lassen sich aus Abbildung 2.9(a) ablesen, sie betragen 14. Die Kosten des Schaltnetzes S_1 lassen sich aus Abbildung 2.5 ablesen, sie betragen 10. Insgesamt betragen die Kosten also $\mathrm{C}(S) = 24$. Die Kosten lassen sich einfach reduzieren, weil sowohl in S_1 wie auch in S_2 jede Variable in einem Inverter verarbeitet wird. Drei dieser Inverter kann man also sparen. Außerdem werden in S_1 mittels der Gatter w und z die Terme $\bar{X}_1 X_2$ und $X_1 \bar{X}_2$ berechnet, gleiches passiert auch in S_2. Also kann man auch zwei AND-Gatter sparen. Schließlich wird in S_2 zweimal $X_1 X_2$ gebildet, eines dieser AND-Gatter kann man weiterhin sparen. Damit reduzieren sich die Kosten um 6 auf 18.

Selbsttestaufgabe 2.6 von Seite 58

Das Schaltnetz zur ODER-Verknüpfung $X_1 \vee \cdots \vee X_{14}$ besteht aus 13 OR-Gattern, unabhängig davon wie diese verschaltet sind. Im ungünstigsten Fall sind die Gatter alle hintereinander geschaltet und die Tiefe beträgt 13. Im günstigsten Fall hat man einen balancierten Baum und die Tiefe beträgt $\lceil \log_2(14) \rceil = 4$.

Selbsttestaufgabe 2.7 von Seite 62

Es gilt $n = \lceil \log_2 91 \rceil = 7$, man benötigt also eine 7-stellige Binärzahl. Teilt man 90 durch $2^6 = 64$, dann erhält man $a_6 = 1$ Rest 26. Teilt man 26 durch $2^5 = 32$, so erhält man $a_5 = 0$ Rest 26. Teilt man 26 durch $2^4 = 16$, so erhält man $a_4 = 1$ Rest 10. Weiterhin erhält man $a_3 = 1$, $a_2 = 0$, $a_1 = 1$, $a_0 = 0$. Es gilt also $\mathrm{bin}_7(90) = 1011010$. Als Probe berechnet man $\langle 1011010 \rangle_2 = 2{+}8{+}16{+}64 = 90$.

Selbsttestaufgabe 2.8 von Seite 64

Da eine 8-stellige Zweier-Komplement-Darstellung gesucht ist, gilt $n = 7$. Wegen $z < 0$ gilt $a_7 = 1$, und man bestimmt zunächst $z + 2^n = -116 + 128 = 12$. Es gilt $\text{bin}_7(12) = 0001100$. Also ist $\text{twoc}(-116) = 10001100$.

Für den ersten Ausdruck gilt $n = 5$. Durch Einsetzen in Gleichung (2.5) erhält man

$$[110001]_2 = -2^5 + 2^4 + 2^0 = -15 \; .$$

Für den zweiten Ausdruck gilt $n = 4$ und man erhält $[01001]_2 = 2^3 + 2^0 = 9$.

Selbsttestaufgabe 2.9 von Seite 64

Wegen Gleichung (2.5) ist die rechte Seite von Gleichung (2.6) identisch mit

$$-a_n \cdot 2^{n+l} + \left\langle \underbrace{a_n, \ldots, a_n}_{l-1}, a_n, \ldots, a_0 \right\rangle \; .$$

Wegen Gleichung (2.4) ist dies wiederum identisch mit

$$-a_n \cdot 2^{n+l} + \sum_{i=n}^{n+l-1} a_n \cdot 2^i + \sum_{i=0}^{n-1} a_i \cdot 2^i \; .$$

Klammert man nun bei den beiden ersten Summanden a_n aus und benutzt $\sum_{i=j}^{k} 2^i = 2^{k+1} - 2^j$, dann erhält man

$$-a_n \cdot 2^n + \sum_{i=0}^{n-1} a_i \cdot 2^i = [a_n, \ldots, a_0] \; .$$

Selbsttestaufgabe 2.10 von Seite 66

Man kann einen 2^t-Wege Multiplexer konstruieren, indem man zunächst jeweils die Hälfte der Eingangsvektoren (2^{t-1} viele mit je n Bit) in einem 2^{t-1}-Wege Multiplexer auswählt, von denen man dann zwei Stück braucht, und die beiden Ausgänge dieser Multiplexer nochmals in einem n-Bit Multiplexer (der auch als 2-Wege Multiplexer bezeichnet werden kann) zusammenfasst. Der Multiplexer erhält als Steuersignal s_{t-1}, die beiden 2^{t-1}-Wege Multiplexer erhalten jeweils die Steuersignale $s_{t-2}, \ldots, s_0$. Die Korrektheit dieser Konstruktion kann man durch vollständige Induktion beweisen. Rollt man diese rekursive Konstruktion auf, indem man immer wieder 2^{t-i}-Wege Multiplexer durch einen Multiplexer und zwei 2^{t-i-1}-Wege Multiplexer ersetzt, bis schließlich $2^{t-i-1} = 2$ bei $i = t-2$ gilt, so erhält man schließlich, wenn man nur die Datenleitungen betrachtet, einen balancierten Binärbaum der Tiefe t, in dem jeder Knoten aus einem n-Bit Multiplexer besteht.

Für die Kosten gilt nun

$$\text{C}(\text{MUX}_{n,2^t}) = (2^t - 1) \cdot \text{C}(\text{MUX}_n) = (2^t - 1) \cdot (3n + 1) \; .$$

Für die Tiefe gilt wegen der Baumeigenschaft und der Tatsache, dass auf jedem Pfad durch den Baum nur ein Inverter durchlaufen werden muss:

$$\text{T}(\text{MUX}_{n,2^t}) = 2t + 1 \; .$$

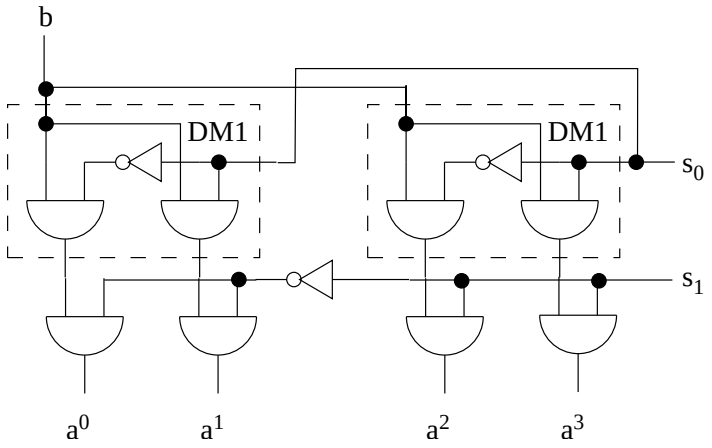


Abbildung 2.28: 1-Bit 4-Wege Demultiplexer

Selbsttestaufgabe 2.11 von Seite 68

Setzt man in Definition 2.6 $n = 1$ und $b = 1$ ein, so erhält man gerade Definition 2.7.

Selbsttestaufgabe 2.12 von Seite 70

Der 1-Bit 4-Wege Demultiplexer wird konstruiert, indem man in Abbildung 2.16(b) bei $t = 2$ die beiden DEMUX(2^{t-1}) jeweils durch einen 2-Wege Demultiplexer aus Abbildung 2.16(a) ersetzt. Damit ergibt sich das Schaltnetz aus Abbildung 2.28.

Der 3-Bit Coder wird konstruiert, indem man zunächst einen 2-Bit Coder konstruiert. Dieser wiederum wird konstruiert, indem man in Abbildung 2.17(b) mit $t = 2$ für die beiden CD_{t-1} jeweils einen 1-Bit Coder aus Abbildung 2.17(a) einsetzt. Zwei dieser 2-Bit Coder setzt man nun in in der Konstruktion von Abbildung 2.17(b) mit $t = 3$ für die CD_{t-1} ein. Das resultierende Schaltnetz sieht man in Abbildung 2.29.

Selbsttestaufgabe 2.13 von Seite 73

Die Kosten eines 4-Bit Carry-Chain Addierers ergeben sich zu $\mathrm{C}(A_4) = 4 \cdot \mathrm{C}(FA) = 4 \cdot 5 = 20$. Die Tiefe eines 4-Bit Carry-Chain Addierers lässt sich abschätzen durch $4 \cdot \mathrm{T}(FA) = 4 \cdot 3 = 12$.

Selbsttestaufgabe 2.14 von Seite 78

Die Kosten eines n-Bit Conditional-Sum Addierers betragen $10 \cdot n^{\log_2 3} - 3n - 2 = 10 \cdot 3^{\log_2 n} - 3n - 2$. Für $n = 4$ ergeben sich Kosten von $10 \cdot 3^2 - 3 \cdot 4 - 2 = 76$. Damit sind die Kosten schon bei einem solch kleinen Addierer fast viermal so hoch wie bei einem Carry-Chain Addierer gleicher Breite (s. Selbsttestaufgabe 2.13). Die Tiefe des Conditional-Sum Addierers beträgt $3 + 3 \log n = 3 + 3 \cdot 2 = 9$, und damit nur 3/4 der Tiefe des Carry-Chain Addierers.

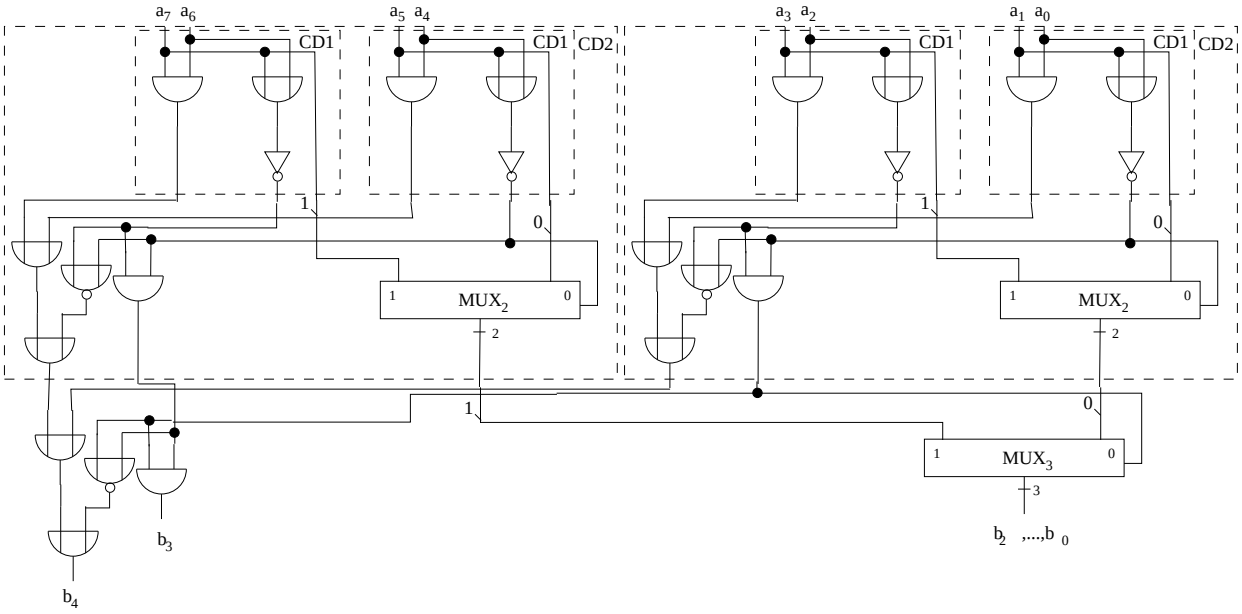


Abbildung 2.29: 3-Bit Coder

Selbsttestaufgabe 2.15 von Seite 84

Das exakte Ergebnis der Addition lautet $2^{24} + 1$. Diese Zahl ist allerdings in einfacher Genauigkeit nicht darstellbar, da dazu 25 Mantissenbits notwendig wären. Die beiden nächsten darstellbaren Zahlen sind 2^{24} und $2^{24} + 2$, so dass bei der Rundung zur Zahl 2^{24} gerundet wird.

Kurseinheit 3

Speicherglieder und Schaltwerke

Kapitelinhalt

Zusammenfassung

Charakteristisches Merkmal von Schaltnetzen ist, dass die von ihnen erzeugten Ausgangssignale ausschließlich von den anliegenden Eingangssignalen abhängen. Es gibt jedoch viele Aufgabenstellungen, bei denen ein digitales System selbstständig (autonom) oder in Abhängigkeit von Eingangssignalen eine Abfolge von Ausgangssignalen erzeugen soll. Beispiele sind Zähler oder Steuerungsschaltungen wie z.B. eine Ampelsteuerung.

In dieser Kurseinheit werden wir uns deshalb den Schaltwerken zuwenden. Bei diesen ist die Ausgabe von den Eingabesignalen und von einem inneren Zustand bestimmt. Dieser Zustand kann sich wiederum in Abhängigkeit der Eingangssignale zeitlich ändern. Zur Speicherung des inneren Zustands werden Speicherglieder benötigt. Diese werden zuerst unter Betrachtung ihres Aufbaus und ihrer Kenngrößen eingeführt. Indem man die Ausgänge von Speichergliedern zusammen mit zusätzlichen Eingabesignalen in einem Schaltnetz verknüpft und dann dessen Ausgangsvektor wieder auf die Eingänge derselben Speicherglieder zurückführt, erhält man ein Schaltwerk. Struktur und Verhalten von Schaltwerken kann man mit Hilfe endlicher Automaten modellieren. Neben der Analyse und der Synthese von Schaltwerken werden in der vorliegenden Kurseinheit auch verschiedene Arten der Implementierung vorgestellt.

Lernziele

Die Lernziele dieser Kurseinheit sind:

- Verständnis von Aufbau und Funktionsweise verschiedener Speicherglieder,
- Kenntnis von Automatenmodellen und deren Anwendung auf Schaltwerke,
- Verständnis des Zeitverhaltens und der Funktionsgrenzen von Schaltwerken,
- Fähigkeit zur Analyse und Synthese von Schaltwerken,
- Kenntnis und Anwendung der Möglichkeiten zur Implementierung von Schaltwerken.

3.1 Motivation

Bevor wir uns näher mit der Realisierung der benötigten Speicherglieder befassen, wollen wir ein einfaches Beispiel für ein Schaltwerk betrachten. Dazu dient ein Vorwärtszähler, der eine Wortbreite von zwei Bit haben soll. Wir gehen davon aus, dass nach dem Einschalten der Betriebsspannung beide Ausgänge der Speicherglieder den Wert 0 haben. Die beiden Bits kann man zu einem Wort Q_1Q_0 zusammenfassen, das einen stellengewichteten Wert darstellt. Ein Vorwärtszähler muss zu diesem Wert 1 addieren, um den nachfolgenden Zählerstand zu ermitteln. Diese Addition erfolgt mit Hilfe eines Addier-Schaltnetzes, das auch als *Inkrementierer* bezeichnet wird (siehe Kurseinheit 1).

Ausgehend vom Anfangszustand 00 durchläuft dann unser Schaltwerk die *Folge*zustände 01, 10 und 11. Danach wechselt es wieder in den Anfangszustand und beginnt von vorne. Es können also insgesamt vier verschiedene Zustände angenommen werden und man bezeichnet ein solches Schaltwerk als *Modulo-4-Zähler*.

Für den Modulo-4-Zähler benötigen wir zwei Speicherglieder, die man auch *Flipflops* nennt. Ein Flipflop ist in der Lage, einen von zwei Zuständen einzunehmen. Damit die Zustandsübergänge bei allen Flipflops gleichzeitig ausgeführt werden, verfügen sie über einen Takteingang, der meist von einem zentralen Taktsignal angesteuert wird.

In Abbildung 3.1 ist das Schaltbild unseres einfachen Schaltwerks dargestellt. Die Schaltung stellt ein *autonomes* Schaltwerk dar, da keine externen Steuereingänge vorhanden sind. Ein allgemeines Schaltwerk verfügt über solche Eingänge. Mit einem externen Steuereingang X könnte man den Zähler z.B. zu einem umschaltbaren Vor-/Rückwärtszähler erweitern.

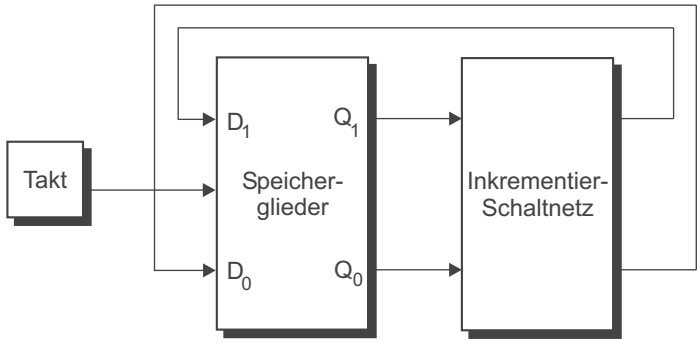


Abbildung 3.1: Aufbau eines Modulo-4-Zählers.

Als nächstes werden wir verschiedene Arten von Speichergliedern kennen lernen. Wir beginnen mit einem so genannten *SR-Latch*, das *asynchron* arbeitet. Die Grundschaltung besitzt keinen Takteingang, der zeitgleiche (*synchrone*) Zustandsübergänge ermöglicht.

Das *SR*-Latch kann leicht um eine Taktsteuerung erweitert werden und man erhält ein *taktzustandsgesteuertes SR*-Latch. Daraus kann dann das *D*-Latch abgeleitet werden. Das gemeinsame Merkmal dieser beiden Latches ist, dass die Eingangssignale sich unmittelbar auf die Ausgänge auswirken, wenn der

Takteingang mit dem Wert 1 belegt ist.

Weil Latches in diesem Betriebszustand *transparent* sind, können sie *nicht* zum Aufbau von Schaltwerken benutzt werden. Wir werden dies im Abschnitt 3.5 noch ausführlich erläutern. Auf Basis von Latches können jedoch Flipflops konstruiert werden, die eine zeitliche Trennung zwischen Ein- und Ausgabe ermöglichen und die daher als Speicherglieder für den Aufbau von Schaltwerken nach Abbildung 3.1 geeignet sind.

Selbsttestaufgabe 3.1 (Vor-/Rückwärtszähler)

a) Ihnen steht ein Halbaddierer und ein Volladdierer zur Verfügung. Entwerfen Sie damit das Inkrementier-Schaltnetz nach Abbildung 3.1.

b) Ersetzen Sie das Inkrementier-Schaltnetz durch ein umschaltbares Inkrementier-/Dekrementier-Schaltnetz, das über ein externes Steuersignal X umgeschaltet werden kann. Der Modulo-4-Zähler soll für $X = 0$ vorwärts und für $X = 1$ rückwärts zählen.

Lösung auf Seite 143

3.2 Speicherglieder

3.2.1 SR-Latch

Das einfachste Speicherglied ist ein SR-Latch. Es besitzt zwei Eingänge, die jeweils zum Setzen (S) und Rücksetzen (R) des Speicherzustands verwendet werden. Aufgrund seines internen Aufbaus wird sowohl ein Ausgang Q als auch dessen Komplement $\overline{Q}$ bereitgestellt (Abbildung 3.2).

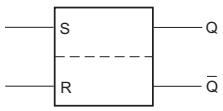


Abbildung 3.2: Schaltbild eines SR-Latches.

Im gesetzten Zustand ist $Q = 1$ und $\overline{Q} = 0$. Im rückgesetzten Zustand ist $Q = 0$ und $\overline{Q} = 1$. Solange an den beiden Steuereingängen 0-Signale anliegen, bleibt der aktuelle Speicherzustand erhalten. Mit $SR = 10$ kann das Latch gesetzt und mit $SR = 01$ rückgesetzt werden. Die Belegung $SR = 11$ ist nicht zulässig, da das Latch nicht gleichzeitig gesetzt und rückgesetzt werden darf. Die Funktionen des SR-Latches werden in Tabelle 3.1 zusammengefasst.

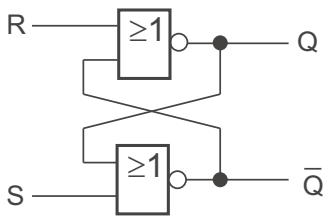


Abbildung 3.3: SR-Latch aus NOR-Schaltgliedern.

In Abbildung 3.3 ist ein SR-Latch dargestellt, das aus NOR-Schaltgliedern aufgebaut ist[1]. Der Ausgang des einen Schaltglieds wird jeweils auf einen der zwei Eingänge des jeweils anderen Schaltglieds zurückgeführt.

Im Folgenden werden wir die Funktionsweise dieses SR-Latches analysieren.

Tabelle 3.1: Funktionen des SR-Latches auf Basis von NOR-Schaltgliedern. ($\times$ steht für eine beliebige Belegung mit 0 oder 1.)

S	R	$Q(t)$	$Q(t+1)$	$\overline{Q}(t+1)$	Funktion
0	0	0	0	1	Speichern
0	0	1	1	0	Speichern
1	0	$\times$	1	0	Setzen
0	1	$\times$	0	1	Rücksetzen
1	1	$\times$	0	0	unzulässig

Funktionsweise

Zunächst gehen wir davon aus, dass das Latch gesetzt ($Q = 1, \overline{Q} = 0$) und dass $SR = 00$ ist. Da $\overline{Q} = 0$ und $R = 0$ sind, wird der Ausgang des oberen NOR-Schaltgliedes seine Belegung mit $Q = 1$ beibehalten. Da $Q = 1$ ist, wird auch das untere NOR-Schaltglied – unabhängig von der Belegung an S – als Ausgabe $\overline{Q} = 0$ liefern. Der Zustand $Q = 1, \overline{Q} = 0$ ist also *stabil* und bleibt gespeichert. Speichern

Nehmen wir nun an, dass sich die Belegung am Eingang R ändert, d.h. $R = 1$ wird. Der Ausgang Q wird nach der Schaltverzögerung des oberen NOR-Schaltglieds[2] 0 werden. Da nun beide Eingänge des unteren NOR-Schaltglieds 0 sind, wird nach dessen Schaltverzögerung der Ausgang $\overline{Q} = 1$. Das Latch ist nun zurückgesetzt. Es behält diesen Zustand auch bei, wenn der Eingang R wieder auf 0 zurückgenommen wird. Es genügt nämlich, dass einer der beiden Eingänge des oberen NOR-Schaltglieds auf 1 liegt, um $Q = 0$ zu erhalten. Rücksetzen

Die Überlegungen des vorigen Absatzes können analog für den Fall angestellt werden, dass zunächst $S = 1$ und anschließend wieder zurückgenommen wird. In diesem Fall geht das SR-Latch in den Setzzustand ($Q = 1, \overline{Q} = 0$) über. Setzen

Wenn beide Eingänge gleichzeitig 1-Signale erhalten, muss $Q = \overline{Q} = 0$ werden. Diese Kombination der Eingangssignale ist nicht zulässig, da sie zu einer widersprüchlichen Ausgangsbelegung führt.

Ferner ist beim gleichzeitigen Übergang von $SR = 11$ nach $SR = 00$ nicht vorherzusehen, welchen Zustand das SR-Latch einnehmen wird. Denn im Allgemeinen werden die beiden NOR-Schaltglieder verschiedene Schaltverzögerungen aufweisen. Eines der beiden Schaltglieder wird schneller schalten und daher seinen Ausgang zuerst auf 1 setzen. Diese 1-Belegung wird wegen der Rückkopplung den Ausgang des anderen Schaltglieds auf 0 setzen. Da aufgrund der Her- Unzulässige Ansteuerung

[1] Schaltglieder wurden in der Kurseinheit 1 auch *Gatter* genannt.

[2] Ein typischer Wert ist z.B. 1,5 ns.

stellungstoleranzen nicht vorhersehbar ist, welches NOR-Schaltglied „schneller“ ist, kann der Folgezustand des Latches nicht vorhergesagt werden.

Oszillation
critical race

Für den Fall, dass die beiden NOR-Schaltglieder exakt die gleiche Schaltverzögerung haben, entsteht sogar eine *Oszillation*, die man auch als *critical race* bezeichnet. Wenn die Schaltverzögerung z.B. 2 ns beträgt, so werden beide Ausgänge nach dieser Zeit ihren Wert von 0 auf 1 ändern. Wegen der Rückkopplung wird nun an je einem Eingang der beiden NOR-Schaltglieder 1-Pegel liegen, so dass sie nach weiteren 2 ns Schaltverzögerung die Ausgänge wieder auf 0 schalten. Dann wird sich der beschriebene Vorgang wiederholen. An den Ausgängen liegt ein (Takt-)Signal mit einer Periode von 4 ns an.

Zeitverhalten

Die Abbildung 3.4 zeigt das Zeitverhalten eines SR-Latches, das aus *verzögerungsfreien* (idealisierten) NOR-Schaltgliedern aufgebaut ist. Folgende Abläufe sind darin dargestellt:

- Setzen im Zeitintervall t_0 bis t_1,
- Rücksetzen im Zeitintervall t_2 bis t_3,
- Betrieb mit unzulässigen Steuersignalen im Zeitintervall t_5 bis t_6 und
- anschließender Rücksetzvorgang im Zeitintervall t_6 bis t_7.

Zum Zeitpunkt t_{10} wird das gleichzeitige Zurücknehmen bei unzulässigen Steuersignalen demonstriert. Es entsteht die bereits oben beschriebene Oszillation.

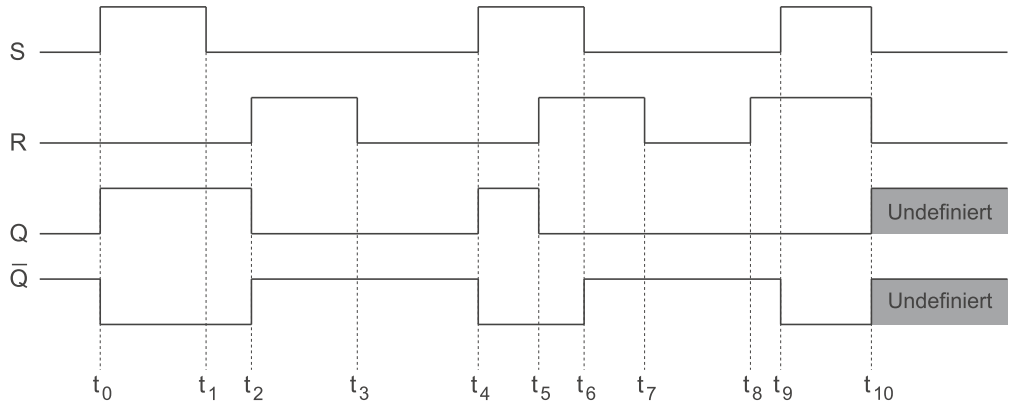


Abbildung 3.4: Zeitverhalten eines SR-Latches mit verzögerungsfreien NOR-Schaltgliedern.

Im Folgenden wollen wir das Zeitverhalten eines SR-Latches betrachten, das aus *verzögerungsbehafteten* NOR-Schaltgliedern aufgebaut ist. Wir nehmen an, dass die Verzögerungszeit pro Schaltglied genau 2 ns beträgt. Das Zeitdiagramm nach Abbildung 3.4 muss dann wie folgt verändert werden (vgl. auch Abbildung 3.3):

- Das Signal am Steuereingang zum Zeitpunkt t_0 wirkt sich erst nach 2 ns auf $\overline{Q}$ und erst nach 4 ns auf Q aus. Die Reaktionszeit des Latches entspricht also der Summe der Verzögerungszeiten der beiden NOR-Schaltglieder.
- Ähnlich wirkt sich auch das Rücksetzsignal zum Zeitpunkt t_2 nach 2 ns auf Q und erst nach 4 ns auf $\overline{Q}$ aus. Der Rücksetzzustand wird also ebenfalls erst 4 ns nach dem Steuersignal eingenommen.

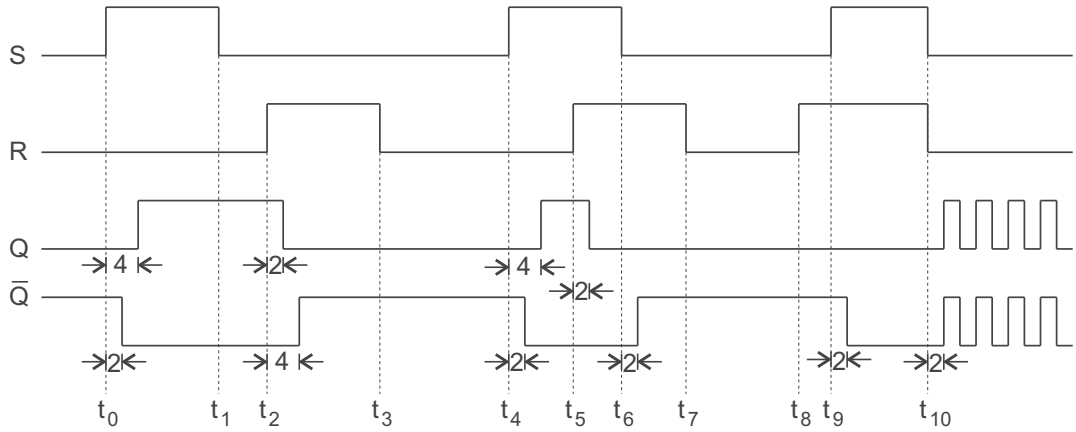


Abbildung 3.5: Zeitverhalten eines SR-Latches mit verzögerungsbehafteten NOR-Schaltgliedern (Zeitangaben in ns).

SR-Latch mit NAND-Schaltgliedern

Das SR-Latch kann auch mit Hilfe von NAND-Schaltgliedern realisiert werden. Im Gegensatz zu Abbildung 3.3 erhalten wir hier jedoch Steuereingänge, die bei einem 0-Pegel wirksam werden. Man bezeichnet derartige Steuereingänge auch als *active low input*. Die entsprechende Schaltung ist in Abbildung 3.6 dargestellt. Um ein SR-Latch gemäß Abbildung 3.2 zu erhalten, müssen den Eingängen $\overline{S}$ und $\overline{R}$ zusätzliche Inverter vorgeschaltet werden. Die Funktionstabelle für ein SR-Latch mit NAND-Schaltgliedern unterscheidet sich von Tabelle 3.1 nur in der letzten Zeile. Im Fall von NAND-Schaltgliedern nehmen für $S = R = 1$ die Ausgänge Q und $\overline{Q}$ statt 0 den Wert 1 an.

active low input

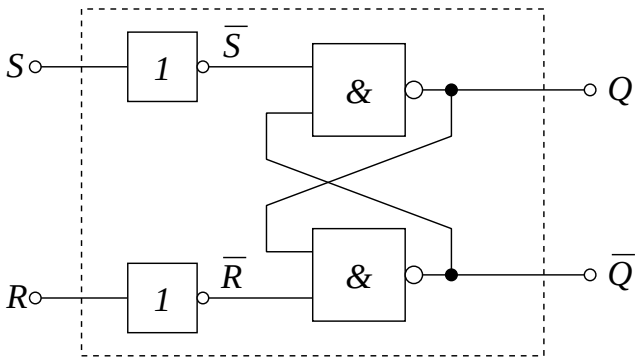


Abbildung 3.6: Aufbau eines SR-Latches auf Basis von NAND-Schaltgliedern.

Selbsttestaufgabe 3.2 (SR-Latch mit NAND-Schaltgliedern)

Skizzieren Sie analog zu den Abbildungen 3.4 und 3.5 Zeitdiagramme eines SR-Latches auf Basis von NAND-Schaltgliedern. Im Falle verzögerungsbehafteter Schaltglieder soll eine Verzögerungszeit von 2 ns pro Schaltglied angenommen werden.

Lösung auf Seite 144

3.2.2 Taktzustandsgesteuertes SR-Latch

Taktsignal

Beim SR-Latch nach Abbildung 3.2 werden die Steuersignale S und R unmittelbar wirksam. Oft ist es jedoch sinnvoll, dass die Wirksamkeit dieser Signale von einem zusätzlichen *Taktsignal* C (für Clock) abhängig gemacht wird. Wenn sich der Zustand des SR-Latches nur dann ändern kann, wenn $C = 1$ ist, so spricht man von einem *taktzustandsgesteuerten* SR-Latch (Abbildung 3.7). Die Taktzustandssteuerung erfolgt mit Hilfe zweier AND-Schaltglieder, die jeweils über zwei Eingänge verfügen. Je einer dieser beiden Eingänge ist mit dem gemeinsamen Takteingang C verbunden. Die beiden Steuersignale S und R werden nur dann an das nachgeschaltete SR-Latch weitergeleitet, wenn der Takteingang den Wert 1 hat. Sonst liegt auf den Leitungen S' und R' der Wert 0 an und das nachgeschaltete SR-Latch befindet sich im Speicherzustand. Dieses „normale“ SR-Latch kann wie oben beschrieben entweder mit NOR- oder mit NAND-Schaltgliedern realisiert werden. Die Funktionsweise wird in Tabelle 3.2 zusammengefasst. Dabei wird vorausgesetzt, dass das SR-Latch wie in Tabelle 3.1 mit NOR-Schaltgliedern realisiert wurde. Bei Verwendung von NAND-Schaltgliedern würden die Ausgänge im Fall von $S = R = 1$ beide den Wert 1 annehmen.

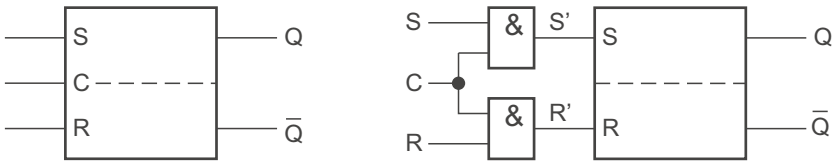


Abbildung 3.7: Schaltzeichen und Schaltbild eines taktzustandsgesteuerten SR-Latches.

3.2.3 Taktzustandsgesteuertes D-Latch

In den beiden letzten Abschnitten haben wir gesehen, dass beim SR-Latch die Steuersignale nicht gleichzeitig 1 werden dürfen, d.h. es muss stets $SR \neq 11$ gelten. Beim Entwurf von Digitalschaltungen ist diese Einschränkung aber nur schwer realisierbar. Ausgehend von dem taktzustandsgesteuerten SR-Latch können wir jedoch mit einem zusätzlichen Inverter erreichen, dass die o.g. unzulässige Kombination der Steuersignale nie auftreten kann (Abbildung 3.8).

zusätzlicher Inverter

Tabelle 3.2: Funktionen des taktzustandsgesteuerten SR-Latches auf Basis von NOR-Schaltgliedern.

C	S	R	$Q(t)$	$Q(t+1)$	$\overline{Q}(t+1)$	Funktion
0	X	X	0	0	1	Speichern
0	X	X	1	1	0	Speichern
1	0	0	0	0	1	Speichern
1	0	0	1	1	0	Speichern
1	1	0	X	1	0	Setzen
1	0	1	X	0	1	Rücksetzen
1	1	1	X	0	0	unzulässig

Dazu verbinden wir den Eingang des Inverters mit dem Steuereingang S und dessen Ausgang mit dem Steuereingang R und erhalten so ein taktzustandsgesteuertes D-Latch.

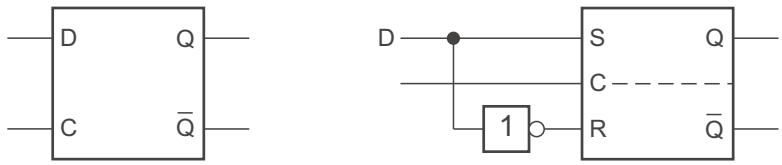


Abbildung 3.8: Schaltzeichen und Schaltbild eines taktzustandsgesteuerten D-Latches.

Der Inverter bewirkt, dass der Wert auf dem Steuereingang D während $C = 1$ vom nachgeschalteten Latch übernommen und am Ausgang Q ausgegeben wird. Wenn das Taktsignal von 1 nach 0 wechselt wird der momentane Wert von D bzw. Q solange im Latch gespeichert, bis das Taktsignal wieder den Wert 1 annimmt. Man nennt das D-Latch bei $C = 1$ *transparent*, da Änderungen des Signals am Eingang nach einer kleinen Verzögerung zu Änderungen des Signals am Ausgang führen. Es ist also so als wäre das D-Latch nicht vorhanden, wenn man von der kleinen Verzögerung absieht. Bei $C = 0$ nennt man das D-Latch *intransparent*, da Änderungen des Signals am Eingang nicht zu einer Veränderung am Ausgang führen. Stattdessen wird am Ausgang bis zur nächsten steigenden Flanke von C der im D-Latch gespeicherte Wert ausgegeben.

Die Funktion des D-Latches wird in Tabelle 3.3 zusammengefasst.

Tabelle 3.3: Funktionstabelle eines D-Latches.

C	D	$Q(t)$	$Q(t+1)$
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1

3.2.4 Setz- und Haltezeiten bei Latches

Wir haben bisher stets implizit angenommen, dass ein Signal lang genug an einem Latch anliegt, so dass das Latch korrekt funktioniert. Sei zum Beispiel τ die Durchlaufzeit eines NOR-Gatters, d.h. die Zeit, die nach einer Signaländerung am Eingang des Gatters bis zu einer Änderung am Ausgang vergeht. Dann muss beim SR-Latch das Setz-Signal S mindestens für die Zeit $2 \cdot \tau$ aktiviert werden, damit der Wert übernommen wird, denn um diese Zeitspanne wird das Signal $S = 1$ beim Durchgang durch beide NOR-Gatter verzögert, bis die 1 am oberen Eingang des unteren NOR-Gatters in Abbildung 3.3 ankommt, so dass die Änderung des Setz-Signals auf $S = 0$ keine unerwünschte Auswirkung mehr auf den Zustand des Latches hat. Daraus folgt auch, dass beim D-Latch für eine gewisse Zeit vor und nach der fallenden Flanke des Taktsignals das Signal D seinen Wert nicht ändern darf, da sonst der Wert nicht sicher übernommen werden kann.

Verkompliziert wird die Situation in der Praxis dadurch, dass man die Durchlaufzeit τ der Gatter nicht genau weiß, da sie von der aktuellen Temperatur, der aktuellen Spannungsversorgung der Schaltung und einigen weiteren physikalischen Gegebenheiten abhängt. Deshalb gibt man normalerweise einen Minimalwert τ_{min} und einen Maximalwert τ_{max} an, so dass die tatsächliche Durchlaufzeit τ sicher dazwischen liegt, d.h. dass

$$\tau_{min} \leq \tau \leq \tau_{max}$$

gilt.

Um unter diesen Umständen nicht ständig solche aufwändigen Betrachtungen wie eben anstellen zu müssen, und da man oft den inneren Aufbau von Latches gar nicht genau kennt, charakterisiert man das zeitliche Verhalten von Latches durch zwei Intervalle, die in Abbildung 3.9 grafisch erläutert sind.

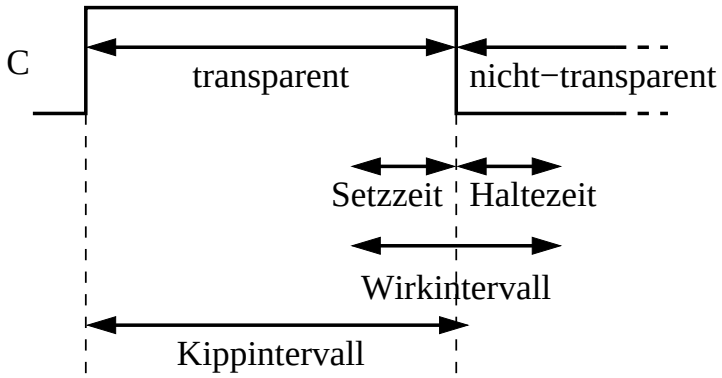


Abbildung 3.9: Zeitdiagramm zur Lage von Wirk- und Kippintervall bei einem taktzustandsgesteuerten D-Latch.

Wirkintervall

Das *Wirkintervall*, das um die Taktflanke des Latches herum gelagert ist, stellt den Zeitraum dar, in dem sich die Eingabesignale des Latches (bis auf das Taktsignal natürlich) **nicht** ändern dürfen, damit das Latch ordnungsgemäß funktioniert. Die Zeit vom Beginn des Wirkintervalls bis zur Taktflanke heißt

dabei *Setzzeit* (setup time), und die Zeit von der Taktflanke bis zum Ende des Wirkintervalls heißt *Haltezeit* (hold time). In Abbildung 3.10 sind diese Zeiten eingetragen. Das D-Eingabesignal darf sich während dieser Zeiten nicht ändern, da sonst der Wert nicht richtig in das D-Latch eingespeichert wird. Haltezeit

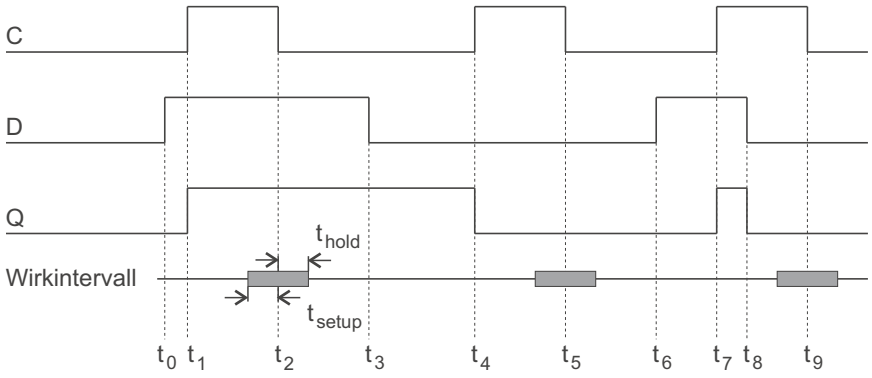


Abbildung 3.10: Zeitdiagramm eines taktzustandsgesteuerten D-Latches. In den Wirkintervallen muss der D-Eingang stabil sein.

Das *Kippintervall* stellt den Zeitraum dar, in dem sich der Ausgang des Latches ändern *kann*, d.h. außerhalb des Kippintervalls ändert sich der Ausgang sicher nicht. Hierbei wird angenommen, dass das Wirkintervall beachtet wird. Beim D-Latch überdeckt das Kippintervall die transparente Phase und reicht, je nach Aufbau, bis zum Ende des Wirkintervalls. Kippintervall

Im Falle eines taktzustandsgesteuerten D-Latches überlappen sich Wirk- und Kippintervall. Zum Aufbau eines sicher funktionierenden Schaltwerks ist es jedoch nötig, dass das Wirkintervall dem Kippintervall *vorausgeht*. Während des Wirkintervalls wird der nachfolgende Zustand (genauer das dem Folgezustand zugeordnete Bit) in das Latch eingeschrieben und gleichzeitig der aktuelle Zustand ausgegeben.

Im Folgenden werden wir zeigen, wie man die oben beschriebenen Latches erweitern muss, um ein derartiges Schaltverhalten zu erreichen. Wir wollen diese Art von Speichergliedern als *Flipflop*s bezeichnen. Flipflop

Es gibt zwei Möglichkeiten, Flipflops mit getrennten Wirk- und Kippintervallen zu realisieren. Wenn wir weiterhin eine Takt*zustands*steuerung verwenden, können wir die Trennung von Wirk- und Kippintervall mit einem zweiten Latch erreichen. Wir erhalten ein so genanntes *Master-Slave-Flipflop*. Eine andere Möglichkeit besteht darin, zu einer Takt*flanken*steuerung überzugehen. Hierbei wirken die Steuereingänge nur in einem kleinen Zeitfenster (Wirkintervall) um die steigende Taktflanke von C ($0 \to 1$) oder fallende Taktflanke von C ($1 \to 0$) auf das Flipflop ein. Master-Slave-Flipflop

Selbsttestaufgabe 3.3 (Schaltwerk mit Latches)
Weshalb würde der Vorwärtszähler nach Abbildung 3.1 mit D-Latches nicht funktionieren?

Lösung auf Seite 144

3.2.5 *Master-Slave-D*-Flipflop

In Abbildung 3.11 ist oben der Aufbau eines *Master-Slave-D*-Flipflops[3] dargestellt. Es besteht aus zwei *D*-Latches, die mit komplementärem Taktsignal betrieben werden. Der Eingang D ist mit dem *Master*-Latch verbunden, das auch direkt das Taktsignal C erhält. Das *Slave*-Latch wird dagegen mit dem invertierten Taktsignal angesteuert. Sein D-Eingang ist mit dem Ausgang Q_M des Master-Latches verbunden und sein Ausgang Q_S liefert das Ausgangssignal Q. Durch die Verwendung zweier hintereinander geschalteter D-Latches wird erreicht, dass das *MS*-Flipflop *nie* transparent ist. Es stellt daher ein ideales Speicherglied für Schaltwerke dar.

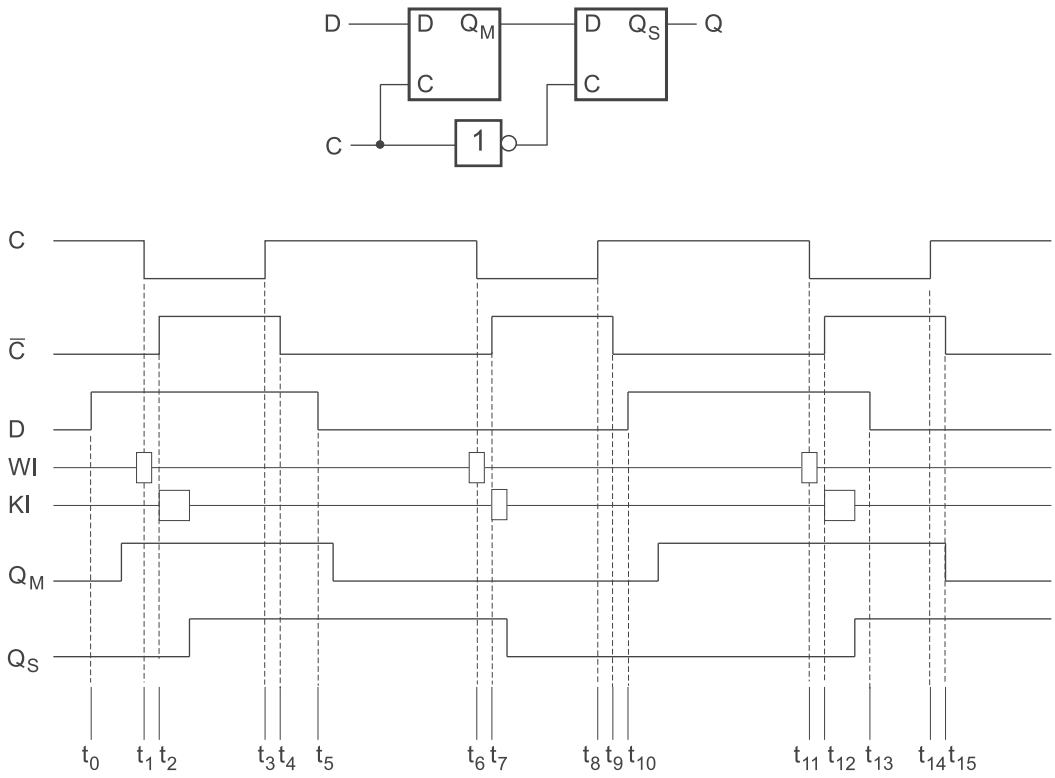


Abbildung 3.11: Aufbau und Zeitverhalten eines *Master-Slave-D*-Flipflops.

Hat das Taktsignal C den Wert 0, so ist das Master-Latch intransparent, und hat den Eingangswert D, der zum Zeitpunkt der fallenden Taktflanke anlag, gespeichert. Das invertierte Taktsignal hat dann aber den Wert 1, so dass das Slave-Latch transparent ist und der im Master-Latch gespeicherte Wert zum Ausgang Q durchgereicht wird. Hat das Taktsignal C eine steigende Flanke, so hat das invertierte Taktsignal eine fallende Flanke, und das Slave-Latch speichert den Wert, der bisher im Master-Latch gespeichert war. Während das Taktsignal den Wert 1 hat, bleibt dieser Wert im Slave-Latch gespeichert. Das Master-Latch ist jetzt zwar transparent, spielt aber wegen der Intransparenz des Slave-Latches keine Rolle. Bei der fallenden Flanke des Taktsignals wird der dann am Eingang D anliegende Wert wieder im Master-Latch gespeichert, und

[3]Künftig als *D-MS*-Flipflop abgekürzt.

der gerade beschriebene Zyklus beginnt aufs neue. Da stets eines der beiden Latches intransparent ist, ist das MS-D-Flipflop nie transparent.

Beim Zusammenschalten der beiden D-Latches muss man dafür sorgen, dass sich das Kippintervall des Master-Latches und das Wirkintervall des Slave-Latches nicht überlappen, da sonst das Einspeichern ins Slave-Latch bei der steigenden Flanke von C nicht richtig funktioniert. Die Zeitverzögerung durch den Inverter bewirkt genau diese Entkopplung. Das Slave-Latch wird erst dann transparent, wenn der Ausgangswert des Master-Latchs schon eingefroren ist.

Das Wirkintervall des MS-D-Flipflops entspricht dem Wirkintervall des Master-Latches. Das Kippintervall hingegen ist im Vergleich zu dem des Slave-Latches relativ schmal, da sich das Ausgabesignal Q nur ändert, wenn das Master-Latch intransparent und das Slave-Latch kurz darauf transparent wird.

Eine ausführliche Funktionsbeschreibung des *D-MS*-Flipflops erfolgt anhand des Zeitdiagramms in Abbildung 3.11. Dabei gehen wir von verzögerungsbehafteten Schalt- und Speichergliedern aus. Zum Zeitpunkt t_0 findet an D ein 0→1-Signalübergang statt. Q_M reagiert zeitverzögert. Da Setz- und Haltezeit im Beispiel gleichgroß sind, liegt die Mitte des Wirkintervalls symmetrisch um den $1 \rightarrow 0$-Signalübergang des Taktsignals. Der zum Ende eines Wirkintervalls „eingefrorene“ Wert des D-Eingangs wird später ins Slave-Latch übernommen.

Selbsttestaufgabe 3.4 (Takt-Inverter)
Um in Abbildung 3.11 das Wirkintervall symmetrisch zum $0 \rightarrow 1$-Übergang des Taktsignals zu positionieren, könnte man den Takt beim Slave-Latch direkt einspeisen und den Takt-Inverter vor dem Master-Latch platzieren. Dies würde weiterhin sicherstellen, dass beide Latches mit einem komplementären Taktsignal betrieben werden. Begründen Sie, warum ein solches D-MS-Flipflop nicht für den Aufbau von Schaltwerken geeignet ist!

Lösung auf Seite 144

3.2.6 Taktflankengesteuertes *D*-Flipflop

Eine Möglichkeit, die Daten mit der steigenden Taktflanke zu übernehmen, besteht darin, bei einem normalen D-Latch die Takt*zustands*steuerung durch eine Takt*flanken*steuerung (Flankentriggerung) zu ersetzen. Der Vorteil der Flankensteuerung liegt darin, dass man mit einem einzigen Latch auskommt. Um zu verhindern, dass der D-Eingang sich während des gesamten Zeitraums, in dem $C = 1$ ist, auf den Zustand des Latches auswirken kann, werden bei einem taktflankengesteuerten Flipflop nur kurzzeitige *Impulse* erzeugt, wenn das Taktsignal seine Belegung wechselt. Die AND-Schaltglieder, die beim D-Latch D bzw. $\overline{D}$ auf das nachgeschaltete SR-Latch weiterleiten, werden daher nur für eine kurze Zeit freigegeben.

Taktflankensteuerung

Der Steuerimpuls kann sowohl bei einem $0 \rightarrow 1$- als auch bei einem $1 \rightarrow 0$-Übergang erzeugt werden. Man spricht im ersten Fall auch von einer *positiven* und im zweiten Fall von einer *negativen* Flankensteuerung bzw. -triggerung. Der Steuerimpuls wird durch Laufzeiteffekte (propagation delay) in besonderen Schaltnetzen erzeugt. Solche Laufzeiteffekte sind normalerweise unerwünscht und werden auch als *Hazards* bezeichnet.

Hazards

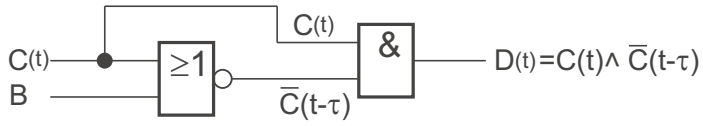


Abbildung 3.12: Impulserzeugung durch Laufzeiteffekte.

In Abbildung 3.12 ist das Prinzip der Impulserzeugung dargestellt. Für den Fall, dass $B = 0$ ist, arbeitet das NOR-Schaltglied als Inverter. Dieser Inverter liefert aber $\overline{C}$ erst mit einer Laufzeitverzögerung τ. Wenn an C ein $0 \rightarrow 1$-Übergang erfolgt, so sind die beiden Eingänge C und $\overline{C}$ des AND-Schaltglieds etwa für gerade diese Zeit τ mit 1 belegt. Dies führt dazu, dass kurz danach an D ein 1-Impuls für die Dauer von τ entsteht. Wenn dagegen an C ein $1 \rightarrow 0$-Übergang erfolgt, so hat die Laufzeitverzögerung des NOR-Schaltglieds keine Wirkung und D bleibt 0.

Betrachten wir nun den Fall, dass $B = 1$ ist. Wegen der Funktion des NOR-Schaltglieds wird $\overline{C}(t - \tau) = 0$ und daher bleibt auch hier D unabhängig von Signalwechseln an $C(t)$ konstant auf dem Wert 0.

Aus den bisherigen Erläuterungen folgt, dass mit dem Schaltnetz nach Abbildung 3.12 eine positive Flankensteuerung implementiert werden kann, die durch ein *active low*-Signal an B aktiviert wird.

In Abbildung 3.13 ist der Aufbau eines einflankengesteuerten D-Flipflops dargestellt, das auf dem oben beschriebenen Prinzip der Impulserzeugung durch Laufzeiteffekte beruht. Da NAND- anstelle von AND-Schaltgliedern benutzt werden, kann ein ebenfalls auf NAND basiertes SR-Latch als eigentlicher Zustandsspeicher nachgeschaltet werden.

Wenn an D der Wert 1 liegt, während das Taktsignal von 0 nach 1 wechselt, wird wegen $\overline{S} = 0$ an $\overline{S'}$ kurzzeitig ein 0-Impuls erzeugt. Da $\overline{R} = 1$ ist, liegt an $\overline{R'}$ permanent 1 an.

Die Lage des Wirkintervalls ist somit durch die $0 \rightarrow 1$-Flanke des Taktsignals bestimmt. Aufgrund der Laufzeitverzögerung τ der NAND-Schaltglieder vor dem SR-Latch erfolgt der Zustandswechsel am Ausgang stets zeitverzögert, d.h. das Kippintervall ist stets vom Wirkintervall getrennt. Es folgt allerdings mit einem sehr kurzen Zeitabstand unmittelbar auf das Wirkintervall. Beim Aufbau von Einregister-Schaltwerken kann dies jedoch zu Problemen führen (vgl. Abschnitt 3.5).

3.2.7 Zweiflankengesteuertes *D*-Flipflop

Ähnlich wie im Abschnitt 3.2.5 können wir auch zwei einflankengesteuerte D-Flipflops zu einem zweiflankengesteuerten *D-MS*-Flipflop zusammenschalten (Abbildung 3.14). Das Taktsignal des Slave-Flipflops wird wieder mit Hilfe eines verzögerungsbehafteten Inverters aus dem Taktsignal des Master-Flipflops abgeleitet.

Im Gegensatz zum taktzustandsgesteuerten *D-MS*-Flipflop ist diese Zeitverzögerung aber unkritisch für die Funktionsweise. Der Takt-Inverter bewirkt,

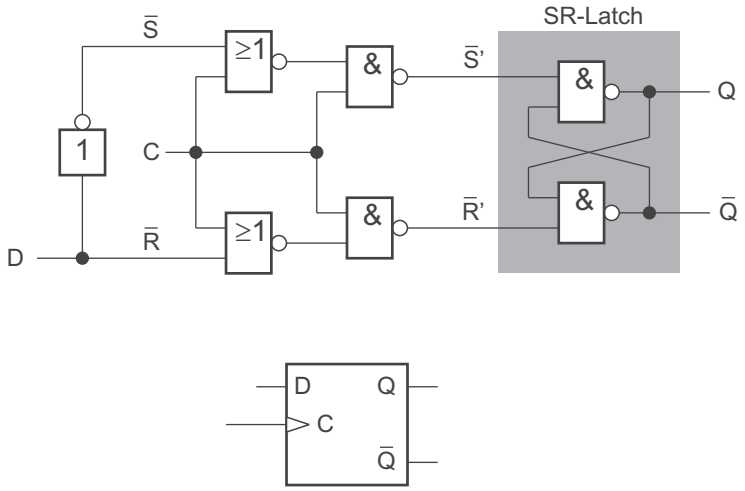


Abbildung 3.13: Aufbau eines einflankengesteuerten D-Flipflops.

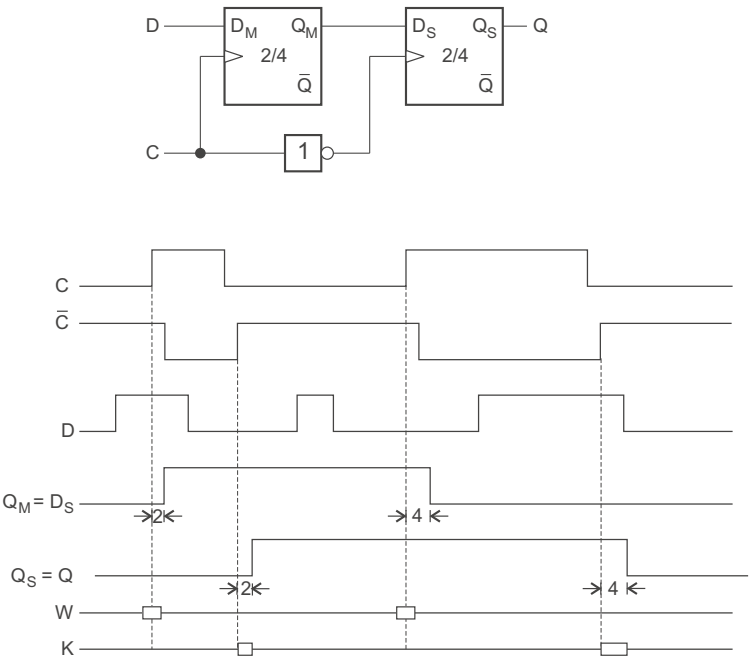


Abbildung 3.14: Aufbau eines zweiflankengesteuerten D-MS-Flipflops.

dass das Slave-Flipflop den Inhalt des Master-Flipflops mit der negativen Taktflanke übernimmt. Wie beim einflankengesteuerten D-Flipflop tastet das Master-Flipflop den Eingang D mit einem um die positive Flanke platzierten Wirkintervall ab. Kurz danach erscheint der abgetastete D-Wert am Ausgang Q_M. Das Slave-Flipflop übernimmt diesen Wert erst mit der negativen Flanke des Taktsignals.

Wir erkennen aus Abbildung 3.14, dass beim zweiflankengesteuerten D-Flipflop die Lage der Wirk- und Kippintervalle beliebig durch die Pulsweite des Taktsignals eingestellt werden kann.

3.2.8 *JK*-Flipflop

Da *MS*-Flipflops immer klar voneinander getrennte Wirk- und Kippintervalle besitzen, sind sie für die Implementierung von Schaltwerken sehr gut geeeignet. Bisher haben wir uns auf *MS*-Flipflops mit nur einem Eingang D beschränkt. Ähnlich zu Abbildung 3.11 können wir aber auch ein *SR-MS*-Flipflop konstruieren. Ein solches Flipflop hat jedoch den Nachteil, dass die Eingangskombination $SR = 11$ unzulässig ist. Dadurch wird der Schaltwerksentwurf erheblich erschwert. Im Falle eines *MS*-Flipflops können wir jedoch eine einfache Modifikation vornehmen, so dass auch bei $SR = 11$ ein klar definiertes Verhalten auftritt. Wie aus Abbildung 3.15 ersichtlich, werden dazu die beiden Eingänge SR zu JK umbenannt.

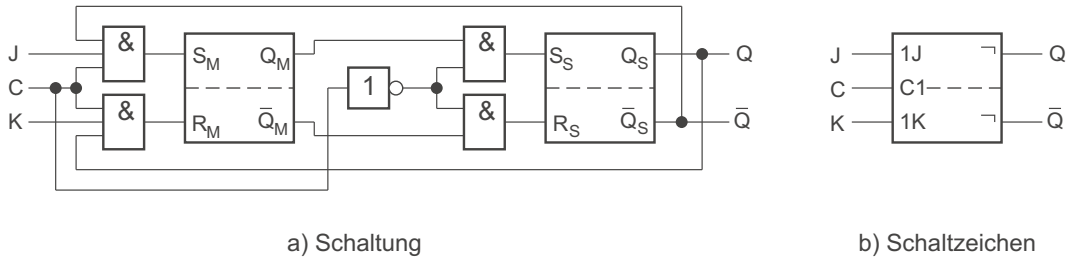


Abbildung 3.15: Aufbau und Schaltzeichen eines *JK-MS*-Flipflops.

Der Eingang J bzw. K entspricht in seiner Funktion dem Eingang S bzw. R. Mit der kreuzweisen Rückkopplung der Ausgänge Q bzw. $\overline{Q}$ werden diese Steuereingänge über AND-Schaltglieder derart modifiziert, dass am nachgeschalteten *SR-MS*-Flipflop auch für $JK = 11$ stets gültige Steuer-„Anweisungen“ ankommen.

Da die Ausgänge Q und $\overline{Q}$ immer komplementäre Belegungen führen, werden auch die Eingänge S_M und R_M ebenfalls zueinander komplementär sein. Ist das Flipflop gesetzt, so wird $R_M = 1$ und $S_M = 0$, d.h. im nächsten Wirkintervall wird das Master-Flipflop zurückgesetzt. Dieser neue Zustand wird anschließend ins Slave-Flipflop übernommen und erscheint im darauf folgenden Kippintervall an den Ausgängen. Man beachte, dass diese Funktionsweise unbedingt ein Master-Slave- bzw. Zweispeicher-Flipflop voraussetzt. Daher können wir den Zusatz MS im Namen des Flipflops auch weglassen. Analog zur obigen Beschreibung wird das Flipflop für $JK = 11$ vom rückgesetzten in den gesetzten Zustand gehen. Die Funktionsweise des JK-Flipflops wird in Tabelle 3.4 zusammengefasst.

T-Flipflop

Lässt man in der Schaltung nach Abbildung 3.15 die Eingänge J und K komplett weg, so entsteht ein so genanntes T-Flipflop. Das T steht dabei für „Toggle“, was übersetzt „umkippen“ bedeutet. T-Flipflops wechseln nach jedem Taktzyklus ihren Ausgangszustand. Dies bedeutet, dass an ihrem Ausgang ein neues, symmetrisches Taktsignal entsteht[4], das die *halbe* Frequenz des ursprünglichen Taktsignals hat.

[4] 0- und 1-Phase sind gleich lang.

Tabelle 3.4: Funktionstabelle eines JK-Flipflops.

J	K	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Mehrere hintereinander geschaltete T-Flipflops können daher als Frequenzteiler eingesetzt werden, die beliebige Teiler zur Basis 2 ermöglichen. Hierzu verbindet man jeweils den Ausgang Q mit dem Takteingang C des nachfolgenden T-Flipflops. Wir realisieren in diesem Fall ein *asynchrones* Schaltwerk, weil die Flipflops nicht mit dem gleichen Taktsignal betrieben werden.

3.2.9 Zusammenfassung der Flipflop-Typen

Wir haben gesehen, dass man die verschiedenen Flipflops bezüglich der Art der Taktsteuerung (zustands- oder flankengesteuert), nach der Wirkungsweise der Eingangssignale (*SR, D, JK, T*) und nach der Anzahl der internen Speicher (Einspeicher, Zweispeicher oder *MS*) unterscheiden kann.

Die Taktung und die Zahl der internen Speicher bestimmt die Lage der Wirk- und Kippintervalle, die letztendlich für die Realisierung funktionierender Schaltwerke wichtig ist. In Abbildung 3.16 werden die behandelten Flipflop-Typen mit verschiedener Taktsteuerung und Anzahl interner Speicher einander gegenüber gestellt.

Im Folgenden gehen wir davon aus, dass nur noch zweiflankengesteuerte *MS*-Flipflops zum Einsatz kommen. Hier können die Abstände zwischen Wirk- und Kippintervallen (T_{WK}, T_{KW}) mittels des Länge der Taktphasen (T_1, T_2) beliebig eingestellt werden.

Wir haben oben bereits die ausführlichen Funktionstabellen der drei wichtigsten Flipflop-Typen (SR, D, JK) angegeben. Die Funktionsweise dieser Flipflops kann aber auch mittels Boole'scher Funktionen oder so genannter *charakteristischer Gleichungen* und grafisch mit so genannten *Zustandsgraphen* dargestellt werden. Beim Zustandsgraph gibt es für jeden Zustand einen Knoten. Zustandsübergänge werden durch Kanten repräsentiert. Diese sind mit den Eingangsbelegungen beschriftet, bei denen dieser Zustandsübergang stattfindet.

charakteristische Gleichung

Zustandsgraph

Die beiden letztgenannten Darstellungsmöglichkeiten können leicht aus der Funktionstabelle abgeleitet werden und sind vor allem bei der *Analyse* von Schaltwerken hilfreich. Für die *Synthese* leisten so genannte *Ansteuertabellen* (excitation table) gute Dienste. Diese werden durch eine Umgruppierung der Ein- und Ausgänge der Funktionstabelle abgeleitet. Für jeden der vier möglichen Zustandsübergänge liefert die Ansteuertabelle die dafür erforderlichen

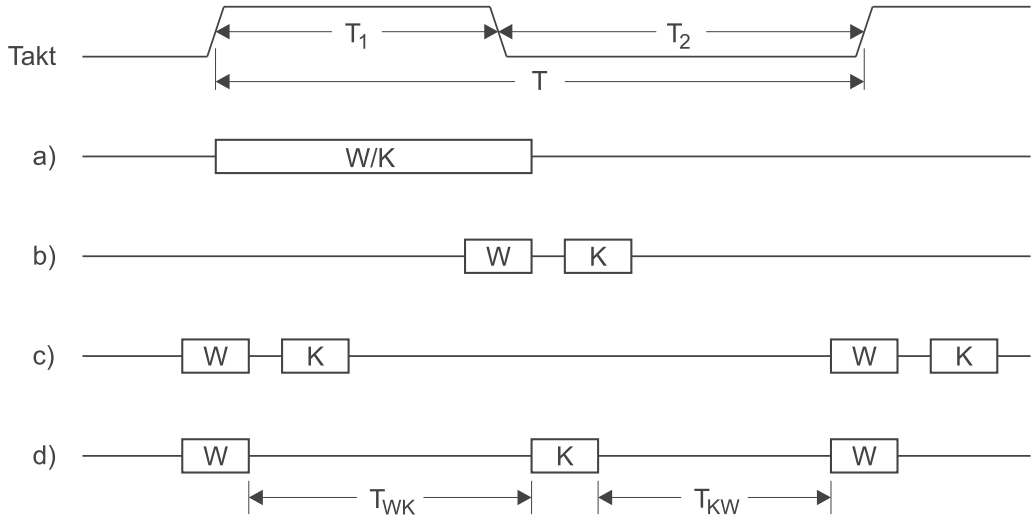


Abbildung 3.16: Lage von Wirk- und Kippintervall bei verschiedenen Flipflop-Typen: a) Latch, b) zustandsgesteuertes *MS*-Flipflop, c) einflankengesteuertes Flipflop, d) zweiflankengesteuertes *MS*-Flipflop.

Belegungen an den Steuereingängen ($\times$ steht für don't care, d.h. es kann den Wert 0 oder 1 annehmen).

Im Folgenden geben wir für jeden der drei wichtigen Flipflop-Typen die (verkürzte) Funktionstabelle, die charakteristische Gleichung, die Ansteuertabelle und den Zustandsgraphen an.

SR-Flipflop

- Funktionstabelle

Tabelle 3.5: Funktionstabelle des SR-Flipflops.

S	R	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	unzulässig

- Charakteristische Gleichung

$$Q(t+1) = S \vee \overline{R}Q(t) \tag{3.1}$$

- Ansteuertabelle

Tabelle 3.6: Ansteuertabelle des SR-Flipflops.

$Q(t)$	$Q(t+1)$	S	R
0	0	0	$\times$
0	1	1	0
1	0	0	1
1	1	$\times$	0

- Zustandsgraph

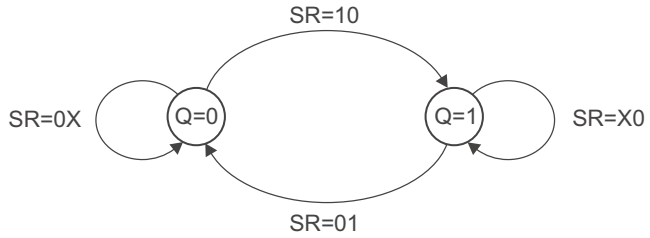


Abbildung 3.17: Zustandsgraph des SR-Flipflops.

JK-Flipflop

- Funktionstabelle

Tabelle 3.7: Funktionstabelle des JK-Flipflops.

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

- Charakteristische Gleichung

$$Q(t+1) = J\overline{Q}(t) \vee \overline{K}Q(t) \tag{3.2}$$

- Ansteuertabelle

Tabelle 3.8: Ansteuertabelle des JK-Flipflops.

$Q(t)$	$Q(t+1)$	J	K
0	0	0	$\times$
0	1	1	$\times$
1	0	$\times$	1
1	1	$\times$	0

- Zustandsgraph

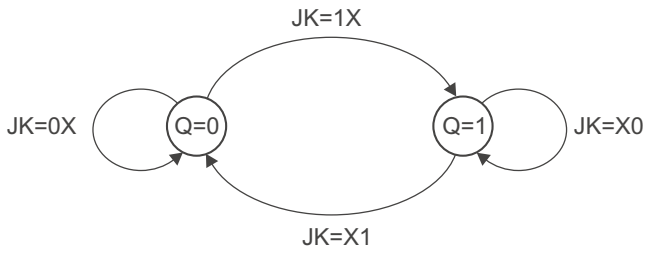


Abbildung 3.18: Zustandsgraph des JK-Flipflops.

D-Flipflop

- Funktionstabelle

Tabelle 3.9: Funktionstabelle des D-Flipflops.

D	$Q(t+1)$
0	0
1	1

- Charakteristische Gleichung

$$Q(t+1) = D \tag{3.3}$$

- Ansteuertabelle

Tabelle 3.10: Ansteuertabelle des D-Flipflops.

$Q(t)$	$Q(t+1)$	D
0	0	0
0	1	1
1	0	0
1	1	1

- Zustandsgraph

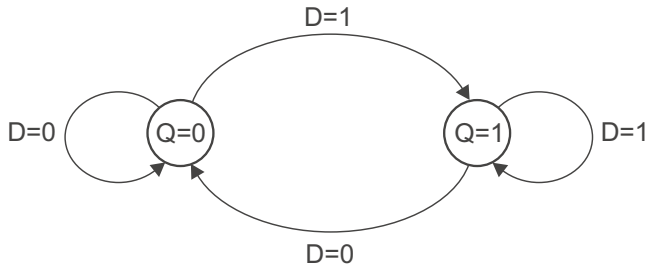


Abbildung 3.19: Zustandsgraph des D-Flipflops.

3.2.10 Asynchrone Setz- und Rücksetz-Eingänge

Um nach dem Einschalten der Betriebsspannung oder dem Drücken einer Taste definierte Speicherzustände vorgeben zu können, verfügen Flipflops neben den Steuer- und Takt-Eingängen auch über asynchrone Setz- und Rücksetz-Eingänge. Diese wirken unmittelbar auf den Speicherzustand eines Flipflops und werden normalerweise als *active low*-Steuersignale ausgelegt. In Abbildung 3.20 ist das Schaltbild eines flankengesteuerten D-Flipflops dargestellt, das über je einen active low-Setz- und Rücksetz-Eingang verfügt. Unabhängig vom Daten-Eingang D und dem Taktsignal C bewirkt eine 0 an $\overline{R}$, dass das Flipflop unmittelbar zurückgesetzt wird ($Q = 0$). Analog dazu bewirkt eine 0 an $\overline{S}$, dass das Flipflop gesetzt wird ($Q = 1$). Natürlich dürfen $\overline{R}$ und $\overline{S}$ nicht beide gleichzeitig mit dem Wert 0 angesteuert werden.

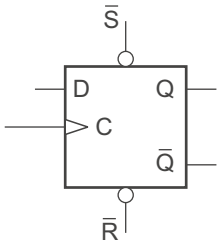


Abbildung 3.20: Schaltbild eines D-Flipflops mit asynchronen Setz- und Rücksetz-Eingängen.

3.3 Register

Register bestehen aus einer bestimmten Anzahl von D-Flipflops, die durch ein gemeinsames Taktsignal angesteuert werden. Die Wortbreite ist häufig eine Zweierpotenz, z.B. 32 Bit. Wie bei einzelnen Flipflops kann man auch bei Registern Wirk- und Kippintervalle definieren. Unter idealen Bedingungen würden alle D-Flipflops absolut gleichzeitig schalten. In der Praxis ist dies jedoch nicht der Fall. Zum einen gibt es aufgrund von Bauteil-Toleranzen bei der Herstellung Unterschiede zwischen den einzelnen Flipflops. Das Wirk- und Kippintervall eines Registers ergibt sich daher aus der Vereinigung der entsprechenden Intervalle der einzelnen Flipflops.

gemeinsames Taktsignal

Zusätzlich müssen aber auch die unterschiedlichen Laufzeitverzögerungen des Taktsignals auf den Verbindungsleitungen berücksichtigt werden. Das Taktsignal an den Eingängen zweier räumlich entfernter Flipflops ist daher leicht verschoben. Diesen Effekt bezeichnet man als *signal skew* bzw. *clock skew*. Die Signalübertragung auf idealen Leitungen erfolgt mit Lichtgeschwindigkeit. Damit ergibt sich rechnerisch eine Signalverzögerung von ca. 3,3 ns pro Meter. Bei verlustbehafteten Leitungen ist die Signalverzögerung jedoch etwa doppelt so groß und liegt bei ca. 7 ns pro Meter.

clock skew

Der zeitliche Versatz des Taktsignals zwischen zwei Flipflops an den Enden des Registers führt dazu, dass die Wirk- und Kippintervalle des Registers verbreitert werden. Wie aus Abbildung 3.16 hervorgeht, liegen bei taktzustands-

und einflankengesteuerten Flipflops die Wirk- und Kippintervalle sehr dicht zusammen. Die o.g. Effekte können bei Registern mit diesen Flipflop-Typen dazu führen, dass das Wirk- und Kippintervall des Registers sich überlappen. Nur bei den zweiflankengesteuerten Flipflops können durch clock skew bedingte Überlappungen mit einem entsprechend geformten Taktsignal wieder ausgeglichen werden.

Wie bei einzelnen Flipflops gibt es auch bei Registern asynchrone Steuereingänge zum Setzen und Rücksetzen des Registerinhalts. Hiermit kann z.B. nach dem Einschalten der Betriebsspannung ein vordefinierter Wert (Zustand) ins Register geschrieben werden.

Bei einfachen Schaltwerken wird ein einziges Register als zentrales Speicherelement verwendet (Einregister-Schaltwerk). In diesem Fall wird das Register mit jedem Taktzyklus neu beschrieben. In komplexen Schaltwerken (siehe Kurseinheit 4) findet man dagegen mehrere Register, die nur in bestimmten Taktzyklen beschrieben werden sollen. Daher wird ein zusätzlicher Steuereingang benötigt, mit dem zwischen Speichern und Laden des Registers umgeschaltet werden kann. Dieser Eingang wird häufig als *Enable* oder *Load* bezeichnet.

ladbare Register

Um ein solches *ladbares* Register zu realisieren, wird der Takt über ein AND-Schaltglied geleitet, an dessen zweitem Eingang das Load-Signal zugeführt wird. Mit $Load = 0$ kann das Taktsignal ausgeblendet werden und der Registerinhalt bleibt in dem betreffenden Taktzyklus unverändert.

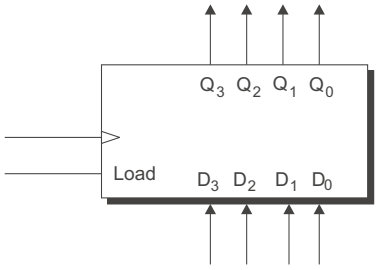


Abbildung 3.21: Schaltbild eines ladbaren 4-Bit-Registers.

Schieberegister

Eine weitere nützliche Variante stellt das *Schieberegister* dar. Im Gegensatz zu einem ladbaren Register sind die Flipflops lateral miteinander verbunden. Je nach Schieberichtung unterscheidet man Links- und Rechts-Schieberegister.

Ein Schieberegister besitzt einen Eingang, an dem mit jedem Taktzyklus ein Bit in das Register eingeschrieben wird. So ist z.B. bei einem 4-Bit-Rechts-Schieberegister mit der gleichen Anordnung der Flipflops wie nach Abbildung 3.21 der Eingang I mit dem Eingang D_3 des Flipflops am linken Rand verbunden. Intern sind die Ausgänge Q_n mit den Eingängen D_{n-1} verbunden. Der Ausgang Q_0 stellt den Ausgang Q des Rechts-Schieberegisters dar. Mit jedem Taktzyklus „verschwindet“ hier ein Bit.

Natürlich könnte man diesen Ausgang Q wieder mit dem Eingang I verbinden. Dann würden die gespeicherten Bits mit jedem Taktzyklus um eine Stelle nach rechts *rotieren*.

Häufig findet man auch *ladbare Schieberegister*, die über zwei Signale S_1S_0 gesteuert vier verschiedene Registerfunktionen bereitstellen.

Selbsttestaufgabe 3.5 (Steuerbares Schieberegister)
Entwerfen Sie ein 3-Bit-Schieberegister, das folgende vier Funktionen ausführt:

S_1	S_0	*Funktion*
0	*0*	*Rechtsschieben*
0	*1*	*Löschen*
1	*0*	*Parallel laden*
1	*1*	*Linksschieben*

Hinweis: Verwenden Sie zum Entwurf 4:1-Multiplexer und D-Flipflops.
Lösung auf Seite 145

3.4 Automatenmodelle für Schaltwerke

Wenn das Ausgabeverhalten einer Digitalschaltung nicht nur von ihrer momentanen Eingabe abhängt, sondern auch auf vorhergehende Eingaben reagieren soll, müssen wir sie als Schaltwerk implementieren. Zur abstrakten Darstellung des Verhaltens verwendet man das Modell eines endlichen Automaten. Dabei kann man zwei Automatentypen unterscheiden: Mealy- und Moore-Automaten.

Ein *Automat* ist ein Modell für ein diskretes, zeitveränderliches System. Automaten können formal durch ein 6-Tupel beschrieben werden:

$$< I, S, O, s_0, f, g > \tag{3.4}$$

Darin bezeichnen

- I die Menge der möglichen Eingabezeichen (Eingabealphabet),
- S die Menge der Zustände,
- O die Menge der möglichen Ausgabezeichen (Ausgabealphabet),
- s_0 den Startzustand,
- g die Übergangsfunktion und
- f die Ausgangsfunktion.

Wenn die Mengen I, S und O endlich sind, so spricht man von einem *endlichen* Automaten (*Finite State Maschine*, FSM). Finite State Maschine

Für jedes Paar aus einem Zustand und einem Eingabezeichen liefert die Übergangsfunktion g einen eindeutig bestimmten Folgezustand:

$$g : S \times I \rightarrow S \tag{3.5}$$

Die Ausgabezeichen werden durch die Funktion f bestimmt. Für diese Funktion gibt es *zwei* verschiedene Definitionen. Die erste Möglichkeit besteht darin, die Ausgabezeichen lediglich aus dem augenblicklichen Zustand abzuleiten:

$$f : S \rightarrow O \tag{3.6}$$

Moore-Automat

Diese Variante wird als *zustandsbasierter* endlicher Automat oder *Moore-Automat* bezeichnet.

Mealy-Automat

Eine andere Möglichkeit zur Definition von f besteht darin, zusätzlich das aktuelle Eingabezeichen einzubeziehen. Man bezeichnet diese Variante als *übergangsbasierter* endlicher Automat oder *Mealy-Automat*. In diesem Fall gilt:

$$f : S \times I \to O \tag{3.7}$$

Wenn die Elemente von I, S und O aus binären Zeichenketten bestehen, so stellen die Funktionen f und g Boole'sche Funktionen dar. Sie können daher durch Schaltnetze implementiert werden. Zur Speicherung der Zustände können Flipflops bzw. Register mit geeignetem Zeitverhalten benutzt werden.

Wenn der Eingabevektor X eines Schaltwerks aus m Elementen besteht, d.h. $X = (x_{m-1}, \ldots, x_1, x_0)$, so ergibt sich die Menge I der Eingabezeichen aus dem kartesischen Produkt der einzelnen Signalmengen. Im Falle binärer Eingaben gilt also

$$I = \{0, 1\}^m \tag{3.8}$$

Ähnlich erhalten wir die beiden anderen Mengen für ein Schaltwerk mit einem k Bit langen Zustandsvektor bzw. -register $Z = (z_{k-1}, \ldots, z_0)$ und einem n-dimensionalen Ausgangsvektor $Y = (y_{n-1}, \ldots, y_0)$. Im Falle binärer Signale gilt:

$$S = \{0, 1\}^k \tag{3.9}$$

$$O = \{0, 1\}^n \tag{3.10}$$

Vollständigkeit

Ein Automat heißt *vollständig*, wenn für jeden Zustand und alle möglichen Eingaben Zustandsübergänge (Kanten) spezifiziert sind. Eine Kante kann auch durch eine ODER-Verknüpfung zweier oder mehrerer möglicher Eingaben markiert sein.

Widerspruchsfreiheit

Ein Automat heißt *widerspruchsfrei*, wenn für jeden Zustand und alle möglichen Eingaben jeweils ein eindeutiger Folgezustand bestimmt ist, d.h. wenn g tatsächlich eine Funktion ist. Für den Zustandsgraphen bedeutet dies, dass es für jede mögliche Eingabe nur eine einzige auslaufende Kante aus einem Knoten (Zustand) gibt.

Der Aufbau eines synchronen Schaltwerks nach den beiden oben beschriebenen Automatenmodellen ist in Abbildung 3.22 dargestellt. Um eine korrekte Arbeitsweise des Schaltwerks zu gewährleisten, muss der Folgezustand Z^{t+1} stets durch nicht-transparente Speicherglieder (z.B. Register mit Master-Slave-Flipflops) vom aktuellen Z^t entkoppelt sein. Dies wird durch die verschieden schraffierten Bereiche des Registers angedeutet.

3.4.1 Darstellungsformen

Es gibt im Wesentlichen zwei Möglichkeiten, um das Verhalten eines Schaltwerkes darzustellen:

- Zustandstabellen (oder Zustandsdiagramme) und
- Zustandsgraphen.

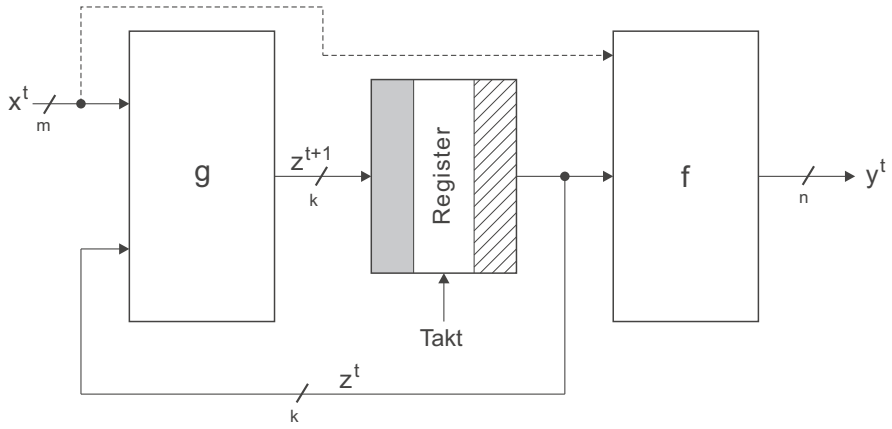


Abbildung 3.22: Aufbau eines Mealy-Schaltwerks. Wenn die gestrichelte Verbindung weggelassen wird, erhalten wir ein Moore-Schaltwerk.

Die *Zustandstabelle* enthält pro Zeile als Eingangsvariablen die Komponenten des Eingabevektors X^t und die Komponenten des Zustandsvektors Z^t, als Ausgangsvariablen die Komponenten des Ausgabevektors Y^t und die Komponenten des Folgezustandsvektors Z^{t+1}.

Eingangsvariablen						Ausgangsvariablen					
z^t_{k-1}	$\cdots$	z^t_0	x^t_{m-1}	$\cdots$	x^t_0	z^{t+1}_{k-1}	$\cdots$	z^{t+1}_0	y^t_{n-1}	$\cdots$	y^t_0

Die Werte der Ausgangsvariablen werden durch ein Schaltnetz aus den Werten der Eingangsvariablen gebildet. Für jede Ausgangsvariable kann eine minimierte Schaltfunktion in der DNF oder KNF bestimmt werden.

Ein *Zustandsgraph* beschreibt das Verhalten eines Schaltwerks in graphischer Darstellung. Er besteht aus Knoten und Kanten. Die Knoten werden als Kreise gezeichnet und stellen die inneren Zustände des Schaltwerks dar. Die Kanten werden als gerichtete Verbindungen zwischen den Knoten gezeichnet und stellen die Übergänge zwischen Zuständen dar (Abbildung 3.23).

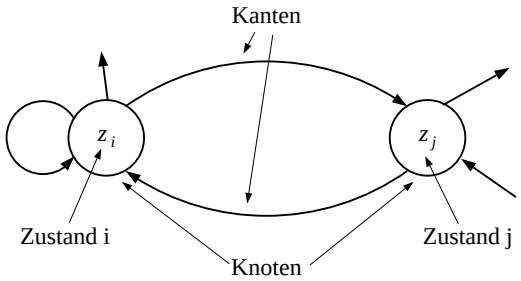


Abbildung 3.23: Grundform eines Zustandsgraphen.

Die Knoten (Kreise) enthalten die Zustandsnamen oder die zugehörigen Kombinationen der Zustandsvariablen. An die Kanten wird die Belegung des Eingabevektors geschrieben, die das Schaltwerk vom gegenwärtigen zum Folgezustand überführt. Beim Mealy-Automaten schreibt man zusätzlich hinter die Eingabekombination die zum gegenwärtigen Zustand gehörige Belegung des Ausgabevektors. Beim Moore-Automaten ist der Ausgabevektor eindeutig durch den momentanen Zustand (Knoten) bestimmt. Daher müssen die Ausgaben an allen von diesem Knoten ausgehenden Übergangskanten gleich sein. Alternativ schreibt man beim Moore-Automaten die Ausgabe deshalb auch unterhalb des Zustandsnamens in den Zustand und nicht an die Kanten.

Der Übergang zum Folgezustand ist zwar taktabhängig, jedoch wird der Takteingang nicht angegeben, da er kein Informationsträger ist. Eine auf den Ausgangsknoten zurückführende Kante gibt an, dass bei dieser Belegung des Eingabevektors keine Zustandsänderung auftritt.

Hat der Zustandsvektor eines Schaltwerkes k Variablen, dann hat der entsprechende Zustandsgraph höchstens 2^k Knoten. Bei einem Eingabevektor mit m Variablen können maximal 2^m Kanten (Verzweigungen) von jedem Knoten ausgehen.

3.4.2 Äquivalenz zwischen Mealy- und Moore-Automaten

Die beiden oben beschriebenen Automatenmodelle können so ineinander überführt werden, dass sie ein äquivalentes Ein-/Ausgabeverhalten aufweisen. Wir beginnen mit dem aufwändigeren Fall: der Überführung eines Mealy-Automaten in einen äquivalenten Moore-Automaten. Hier müssen wir dem äquivalenten Moore-Automaten zugestehen, dass im Vergleich zum Mealy-Automaten alle Ausgaben einen Takt verzögert erfolgen, und dass die Ausgabe im Startzustand nicht gewertet wird. Diese Einschränkung lässt sich nicht umgehen, da im Mealy-Automat die Ausgabe im ersten Takt vom Startzustand *und* von der Eingabe abhängig ist, während im Moore-Automat die Ausgabe im ersten Takt *nur* vom Startzustand abhängig ist, und die Eingabe während des ersten Taktes nur den Folgezustand beeinflussen kann, und so erst im folgenden Takt auf die Ausgabe wirken kann.

Bei der Transformation betrachten wir für jeden Knoten v des Mealy-Automaten die eingehenden Kanten. Sind diese alle mit dem gleichen Ausgabevektor Y markiert, so wird im Moore-Automat der Knoten v mit dem Ausgabevektor Y markiert. Gibt es auf den eingehenden Kanten verschiedene Markierungen mit Ausgabevektoren $Y_1, \ldots, Y_k$, so werden im Moore-Automat die Knoten v_1 bis v_k geschaffen, die mit den Ausgabevektoren Y_1 bis Y_k markiert werden, und als eingehende Kante jeweils die erhalten, die im Mealy-Automat mit dem betreffenden Ausgabevektor markiert war.

Abbildung 3.24 zeigt ein Beispiel für die Transformation eines Mealy-Automaten. Wir erkennen, dass der im linken oberen Teil dargestellte Mealy-Automat für den Zustand 1 unterschiedlich markierte einlaufende Kanten aufweist. Daher muss der Zustand 1 in zwei Zustände 1_0 und 1_1 aufgespalten werden. Über die in Abbildung 3.24 rechts oben dargestellte Zwischenstufe ist dann wieder eine einfache Transformation zu einem äquvalenten Moore-Automaten mög-

lich (Abbildung 3.24 unten). Dabei kann, wie in oben schon beschrieben, der Ausgabevektor im Zustand 0 beliebig gewählt werden, da er ignoriert wird.

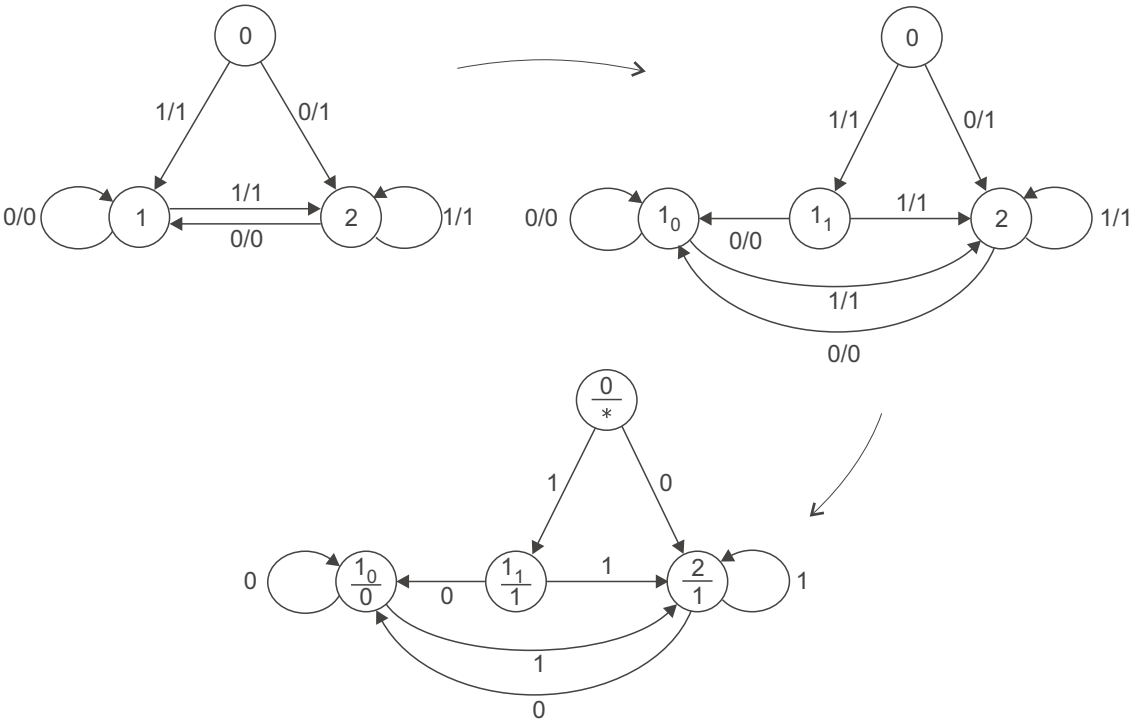


Abbildung 3.24: Umwandlung eines Mealy- in einen äquivalenten Moore-Automaten durch Einfügen zusätzlicher Zustände.

Bei der Überführung eines Moore-Automaten in einen äquivalenten Mealy-Automaten müssen wir prinzipiell nichts tun, da ein Moore-Automat auch als Mealy-Automat betrachtet werden kann, bei dem die Ausgangsfunktion f nicht von ihrem zweiten Argument, dem Eingabevektor, abhängt. In der Darstellung als Graph müsste man dann nur die Ausgabemarkierung jedes Zustands v als Ausgabemarkierung auf alle von v ausgehenden Kanten übertragen. Um eine Symmetrie zum Vorgehen bei der Transformation eines Mealy-Automaten in einen Moore-Automaten zu erhalten, kann man allerdings auch die Ausgabemarkierung jedes Zustands auf die *einlaufenden* Kanten übertragen. Dann entfällt die Ausgabe des Startzustands und alle Ausgaben des Mealy-Automaten erfolgen einen Takt früher als im Moore-Automaten.

Wir wollen dies an einem Beispiel demonstrieren. In der Abbildung 3.25 sehen wir auf der linken Seite den Zustandsgraphen eines Moore-Automaten. Zur Umwandlung in einen äquivalenten Mealy-Automaten müssen die Ausgaben aus den einzelnen Zuständen auf die *einlaufenden* Kanten übertragen werden. Das Ergebnis der Umwandlung ist im rechten Teil von Abbildung 3.25 zu sehen.

3.5 Rückkopplungsbedingungen

Im Folgenden werden wir das Zeitverhalten von synchronen (Einregister-) Schaltwerken genauer analysieren und Bedingungen ableiten, welche die Funktions-

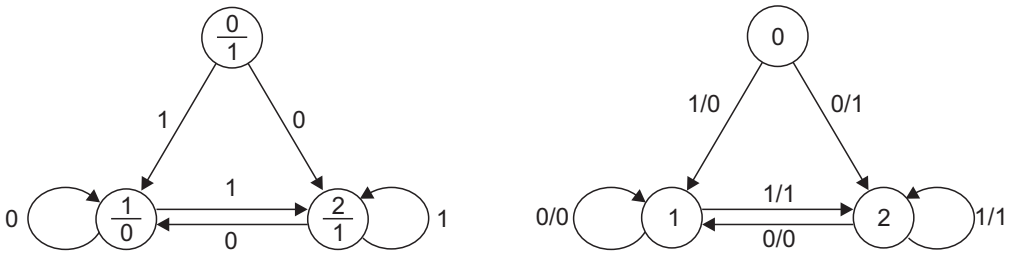


Abbildung 3.25: Umwandlung eines Moore- in einen äquivalenten Mealy-Automaten.

grenzen dieser Schaltwerke bestimmen.

Wie wir in Abbildung 3.22 erkennen, sind die Register-Ausgänge Z^t über das Übergangsschaltnetz g direkt auf die Register-Eingänge Z^{t+1} zurückgekoppelt. Durch diese Rückkopplung wird eine rekursive Funktion des Zustandsvektors Z^t erreicht.

Nach dem Einschalten der Betriebsspannung wird im Register über hier nicht eingezeichnete Setz- bzw. Rücksetz-Eingänge ein definierter Anfangszustand Z^0 hergestellt. Dieser Zustand wirkt, zusammen mit dem momentan anliegenden Eingabevektor X^0, auf das Übergangsschaltnetz g ein, welches dann nach einer gewissen Verzögerungszeit T_g den Folgezustand Z^1 liefert. Die Verzögerungszeit T_g ist durch die Tiefe des Schaltnetzes g bestimmt. Die Ausgänge dieses Schaltnetzes müssen spätestens zum Beginn des (Register-) Wirkintervalls den Folgezustand Z^1 liefern. Nur dann ist sichergestellt, dass er korrekt in das Register eingespeichert wird. Während des nachfolgenden Kippintervalls wird sich der Zustandsvektor Z^t von Z^0 zu Z^1 ändern und der oben beschriebene Ablauf beginnt von vorne. In dieser Weise durchläuft das Schaltwerk verschiedene Zustände. Abhängig von der durch das Schaltnetz g vorgegebenen Übergangsfunktion können einzelne Zustände auch mehrfach durchlaufen werden oder es gibt einen oder mehrere Endzustände, die nicht mehr verlassen werden können.

Im Folgenden wollen wir die Funktionsgrenzen eines Schaltwerks analysieren. Ein Schaltwerk funktioniert nur dann *sicher*, wenn die folgende Grundbedingung erfüllt ist: *Die Eingangsvariablen sämtlicher Flipflops müssen während des Wirkintervalls des (Schaltwerk-)Registers stabil sein.*

Diese Bedingung ist hinreichend, aber nicht notwendig. Ein Schaltwerk *kann* für eine ganz bestimmte Zustandsfolge funktionieren, wenn nur die Wirkintervalle einzelner Flipflops eingehalten werden (notwendige Bedingung). Daraus folgt aber nicht, dass ein solches Schaltwerk alle *möglichen* Zustandsübergänge realisieren kann.

Aus der oben genannten Bedingung lassen sich zwei *Rückkopplungsbedingungen* ableiten, um die zulässigen bzw. notwendigen Abstände zwischen den Wirk- und Kippintervallen eines Einregister-Schaltwerks zu ermitteln.

Dabei ist als dynamische Kenngröße die maximale Verzögerungszeit T_g des Rückkopplungsschaltnetzes g zu berücksichtigen. Die Zeit T_g hängt sowohl von der verwendeten Halbleitertechnologie als auch von der Zahl der Verknüpfungsebenen des Übergangsschaltnetzes ab. Sie setzt sich aus einem Totzeitanteil T_{gt}

und einem Übergangsanteil $T_{gü}$ zusammen. Änderungen der Eingangsbelegung des Schaltnetzes g wirken sich zunächst *nicht* auf die Ausgänge aus. Die Totzeit ist proportional zu der minimalen Anzahl von Gattern, die ein Signalpfad von einem beliebigen Eingang von g zu einem beliebigen Ausgang von g durchlaufen muss. Nach der Totzeit können sich die Ausgänge verändern. Die zu den neuen Eingangsbelegungen gehörenden Ausgangsbelegungen stellen sich spätestens nach Ablauf der Übergangszeit ein.

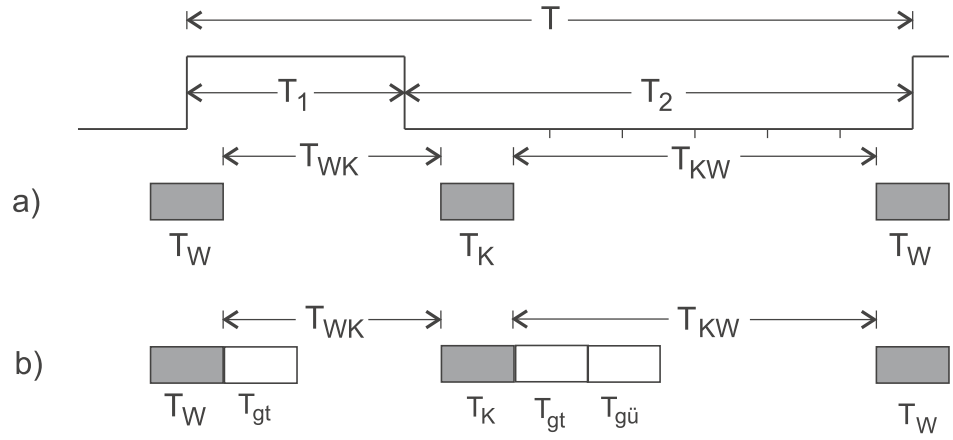


Abbildung 3.26: Zur Herleitung der Rückkopplungsbedingungen. Zeitverhalten a) *ohne* b) *mit* Tot- und Übergangszeiten.

Abbildung 3.26 zeigt die Lage von Wirk- und Kippintervallen des Zustandsregisters innerhalb eines Taktes. Die Zeit zwischen dem Ende des Wirkintervalls und dem Beginn des Kippintervalls ist dabei mit T_{WK} bezeichnet, die Zeit vom Ende des Kippintervalls bis zum Beginn des Wirkintervalls des nächsten Takts mit T_{KW}. Wir leiten nun Bedingungen für die minimale Länge dieser beiden Zeiten her. Aus diesen Bedingungen können wir mittels

$$T \geq T_W + T_{WK} + T_K + T_{KW}$$

eine minimale Zykluszeit (und als deren Kehrwert eine maximale Taktfrequenz) bestimmen, mit der das Schaltwerk sicher betrieben werden kann. Eine solche minimale Zykluszeit kann aber nur erreicht werden, wenn man tatsächlich ein Register findet, dessen Spezifikationen genau den minimalen Werten von T_{WK} und T_{KW} entsprechen. Liegen die Werte des Registers über den Minimalwerten, so muss die Zykluszeit des Schaltwerks entsprechend größer gewählt werden. Liegen sie darunter, ist das Register für das geplante Schaltwerk nicht brauchbar.

1. Rückkopplungsbedingung

Da der Ausgang des Zustandsregisters über das Schaltnetz g auf seinen eigenen Eingang zurückgekoppelt ist, muss sichergestellt werden, dass Kippvorgänge am Ausgang nicht unmittelbar auf den Eingang zurückwirken. Sofern keine Totzeiten berücksichtigt werden (Abbildung 3.26 a)), muss $T_{WK} \geq 0$ gelten.

Dies bedeutet, dass sich Wirk- und Kippintervall des Zustandsregisters nicht überlappen dürfen. Die Ausgangssignale des Zustandsregisters werden beim Durchgang durch das Schaltnetz g allerdings verzögert, und zwar mindestens um T_{gt}. Berücksicht man diese Totzeit, so ergibt sich

$$T_{WK} \geq -T_{gt} \; . \tag{3.11}$$

In diesem Fall kann T_{WK} also negativ sein, d.h. Wirk- und Kippintervall des Registers dürfen sich überlappen.

2. Rückkopplungsbedingung

Spätestens zum Ende des Kippintervalls steht der aktuelle Zustand am Ausgang des Zustandsregisters bereit. Die Berechnung des Nachfolgezustands, die längstens $T_{gt}+T_{g\ddot{u}}$ dauert, muss abgeschlossen sein, bevor das Wirkintervall des folgenden Takts beginnt, denn sonst könnte der Nachfolgezustand nicht sicher ins Register übernommen werden. Damit gilt aber gerade

$$T_{KW} \geq T_{gt} + T_{g\ddot{u}} = T_g \; . \tag{3.12}$$

Zur Realisierung von Einregister-Schaltwerken eignen sich zweiflankengesteuerte Master-Slave-Flipflops am besten. Durch Veränderung des Taktverhältnisses können sowohl T_{WK} als auch T_{KW} beliebig eingestellt werden. Man wird dadurch fast gänzlich unabhängig von den Flipflop-spezifischen Schaltzeiten. Die Taktphase T1 bestimmt im Wesentlichen die Zeit T_{WK} und die Taktphase T2 legt T_{KW} fest. Die maximal mögliche Taktfrequenz eines Schaltwerks wird vorwiegend durch die Verzögerungszeit T_g des Rückkopplungs-Schaltnetzes begrenzt. Diese Verzögerungszeit kann vor allem bei Schaltnetzen zur Berechnung arithmetischer Funktionen sehr groß werden. Daher ist es wichtig, dass die Laufzeiten bei arithmetischen Schaltnetzen durch geeignete Schaltungen minimiert werden.

Selbsttestaufgabe 3.6 (Maximale Taktfrequenz)
Wie hoch darf die maximale Taktfrequenz eines autonomen Schaltwerks sein, das durch die Kenngrößen $T_W = 2ns$, $T_K = 1ns$, $T_{gt} = 100ps$ und $T_g = 4ns$ beschrieben wird?

Lösung auf Seite 145

3.6 Analyse von Schaltwerken

Im Folgenden gehen wir davon aus, dass der Schaltplan eines Schaltwerks vorgegeben ist und wir die Aufgabe haben, sein Schaltverhalten zu analysieren. Hierzu müssen wir zunächst die Übergangsfunktion g und die Ausgangsfunktion f anhand der vorgegebenen (Rückkopplungs-)Schaltnetze herleiten. Dann wird ein Anfangszustand Z_0 angenommen und mit den möglichen Werten der Eingabevariablen und der Übergangsfunktion werden die erreichbaren Folgezustände bestimmt. Zusammen mit der Ausgangsfunktion können auf diese Weise

die Zeilen der Zustandstabelle erstellt werden. Mit Hilfe der vollständigen Zustandstabelle kann schließlich der Zustandsgraph gezeichnet werden, der das Verhalten des Schaltwerks anschaulich darstellt.

Wir wollen die Vorgehensweise anhand von zwei Beispielen erläutern.

3.6.1 Analyse eines Schaltwerks mit *D*-Flipflops

Wir beginnen mit einem Schaltwerk, das aus (Master-Slave-)D-Flipflops nach Abbildung 3.27 aufgebaut ist[5]. Zunächst einige allgemeine Bemerkungen zur Charakterisierung dieses Schaltwerks: Es handelt sich um ein synchron angesteuertes Schaltwerk. Der Eingabevektor X und der Ausgangsvektor Y bestehen aus je einer Variablen. Das Schaltwerk enthält zwei D-Flipflops als Speicherglieder, es hat also zwei Zustandsvariablen und kann daher maximal vier Zustände einnehmen. Die Komponenten z_0^+ und z_1^+ des Folgezustandsvektors Z^{t+1} werden durch ein Schaltnetz aus dem Eingabevektor X und aus den Komponenten z_0 und z_1 des Zustandsvektors Z zum Zeitpunkt t gebildet. Der Ausgangsvektor Y wird aus dem Eingabevektor X und den Komponenten des Zustandsvektors Z gebildet. Daraus folgt, dass das Schaltwerk einen Mealy-Automaten darstellt.

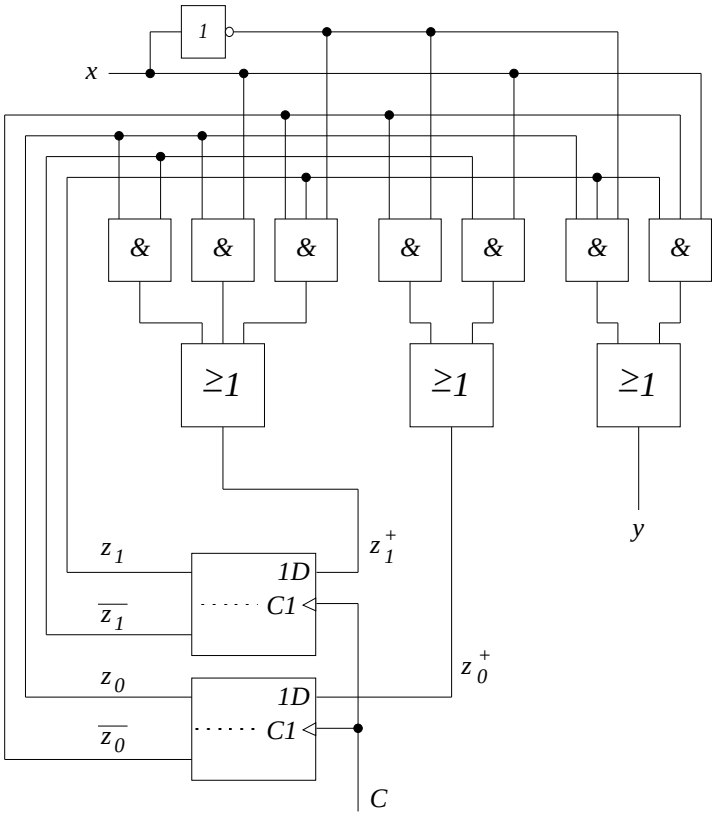


Abbildung 3.27: Schaltwerk mit D-Flipflops.

[5]Der Einfachheit halber markieren wir im Folgenden die Komponenten des Folgezustandsvektors mit einem hochgestellten „+“-Zeichen.

Aus der allgemeinen Charakterisierung ergibt sich eine *erste* Beschreibung des Schaltwerks durch Schaltfunktionen. Die Analyse des Schaltnetzes liefert die Komponenten des Folgezustandsvektors:

$$z_0^+ = (\overline{z}_0 \wedge \overline{x}) \vee (\overline{z}_1 \wedge x) \tag{3.13}$$
$$z_1^+ = (z_0 \wedge \overline{z}_1) \vee (z_0 \wedge x) \vee (\overline{z}_0 \wedge z_1 \wedge \overline{x}) \tag{3.14}$$

Für den Ausgangsvektor Y ergibt sich:

$$y = (z_0 \wedge z_1 \wedge \overline{x}) \vee (\overline{z}_0 \wedge z_1 \wedge x) \tag{3.15}$$

Beim Übergang vom Zeitpunkt t zu $t+1$ wird der Folgezustandsvektor zum neuen Zustandsvektor $Z(t) := Z(t+1)$. Mit dieser Zuweisung und den Schaltfunktionen nach (3.13) und (3.14) kann die *Zustandstabelle* erstellt werden. Wir gehen von einem Anfangszustand $z_0 = 0$, $z_1 = 0$ aus, d.h. die beiden Flipflops sollen beim Einschalten der Betriebsspannung zurückgesetzt werden. Die Eingangsvariable soll zuerst den Wert $x = 0$ haben. Mit (3.13) folgt $z_0^+ = 1$, mit (3.14) folgt $z_1^+ = 0$ und mit (3.15) folgt $y = 0$. Es ergibt sich die Zustandstabelle 3.11.

Tabelle 3.11: Zustandstabelle für das Schaltwerk mit D-Flipflops.

z_1	z_0	x	z_1^+	z_0^+	y
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	1	0
1	0	1	0	0	1
1	1	0	0	0	1
1	1	1	1	0	0

Mit der Zustandstabelle kann der Zustandsgraph gezeichnet werden (Abbildung 3.28).

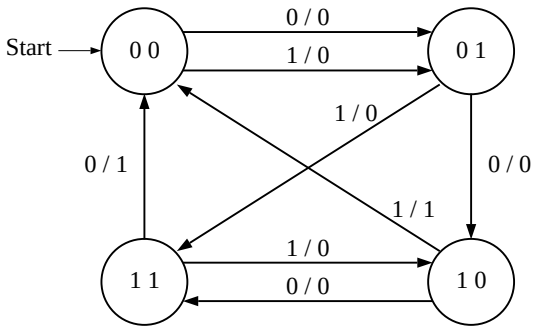


Abbildung 3.28: Zustandsgraph für das Schaltwerk mit D-Flipflops.

3.6.2 Analyse eines Schaltwerks mit *JK*-Flipflops

Im zweiten Beispiel nach Abbildung 3.29 werden zwei JK-Speicherglieder verwendet. Das Schaltwerk kann daher maximal vier Zustände einnehmen. Die Komponenten z_0^+ und z_1^+ des Folgezustandes werden zwar durch ein Schaltnetz aus dem Eingang x und den Komponenten z_0 und z_1 des Zustandsvektors gebildet, aber als getrennte Steuereingänge J und K an die Speicherglieder herangeführt. Der Folgezustand eines einzelnen Flipflops i hängt von den Belegungen der jeweiligen Steuereingänge J_i und K_i ab. Die Analyse des Schaltnetzes führt zu den *Schaltfunktionen*:

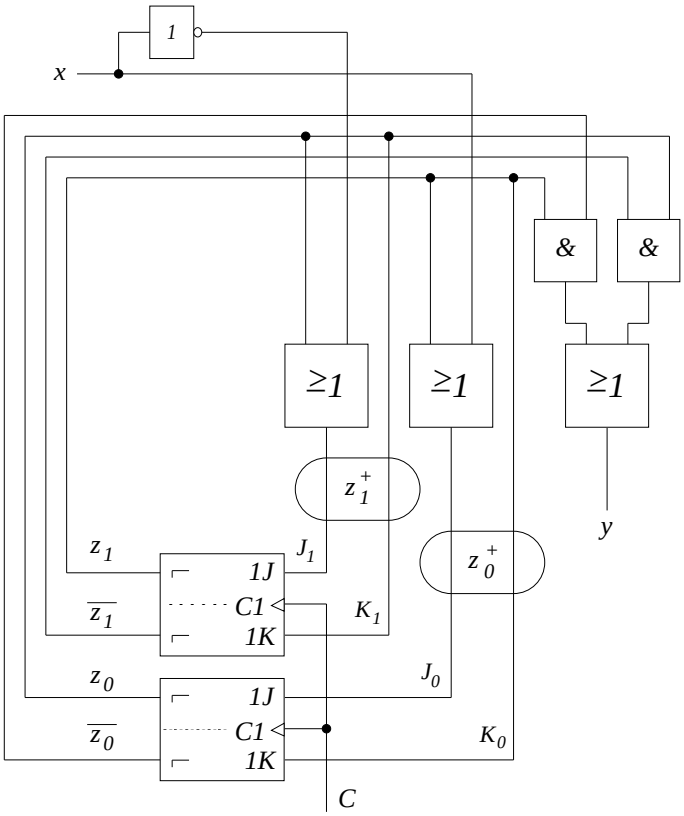


Abbildung 3.29: Schaltwerk mit JK-Flipflops.

$$
\begin{aligned}
J_0 &= x \vee z_1 && (3.16)\\
K_0 &= z_1 && (3.17)\\
J_1 &= \overline{x} \vee z_0 && (3.18)\\
K_1 &= z_0 && (3.19)\\
y &= (\overline{z}_1 \wedge z_0) \vee (z_1 \wedge \overline{z}_0) && (3.20)
\end{aligned}
$$

Der Ausgangsvektor y enthält nur die Variablen des Zustandsvektors Z, das Schaltwerk ist also ein Moore-Automat.

Die *Zustandstabelle* für das Schaltnetz enthält als Eingänge die Variablen des Eingabevektors X (mit nur einer Komponente x) und des Zustandsvektors Z,

als Ausgang die Variablen J und K der Steuereingänge für die Speicherglieder und die Variablen des Ausgangsvektors Y (Tabelle 3.12). Aus den Werten für J und K folgen das Schaltverhalten der Flipflops und daraus die Werte für die Komponenten des Folgezustands.

Tabelle 3.12: Zustandstabelle für das Schaltwerk mit JK-Flipflops.

z_1	z_0	x	K_1	J_1	K_0	J_0	z_1^+	z_0^+	y
0	0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	0	1	0
0	1	0	1	1	0	0	1	1	1
0	1	1	1	1	0	1	1	1	1
1	0	0	0	1	1	1	1	1	1
1	0	1	0	0	1	1	1	1	1
1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	1	1	0	0	0

Wir gehen wieder von einem Ausgangszustand $z_0 = 0$ und $z_1 = 0$ aus. Die Eingangsvariable soll den Wert $x = 0$ haben. Mit (3.16)–(3.19) folgt: $J_0 = 0$, $K_0 = 0$, $J_1 = 1$, $K_1 = 0$. Daraus ergeben sich die Werte für die Variablen des Folgezustandes $z_0^+ = 0$ und $z_1^+ = 1$. Für $x = 1$ wird $z_0^+ = 1$ und $z_1^+ = 0$. Es ergibt sich die Tabelle 3.12.

Aus der Zustandstabelle kann der Zustandsgraph gezeichnet werden (Abbildung 3.30).

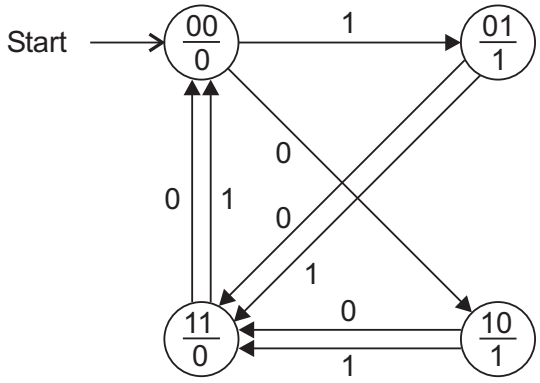


Abbildung 3.30: Zustandsgraph für das Schaltwerk mit JK-Flipflops.

Selbsttestaufgabe 3.7 (2-Bit-Synchronzähler)
Analysieren Sie den Synchronzähler aus Abbildung 3.31.

a) Bestimmen Sie, ausgehend vom Startzustand $Q_0 = 0$ und $Q_1 = 0$, den Zählzyklus für $X = 1$ und für $X = 0$ und erstellen Sie die Zustandstabelle!

b) Zeichnen Sie den Zustandsgraphen!

Lösung auf Seite 146

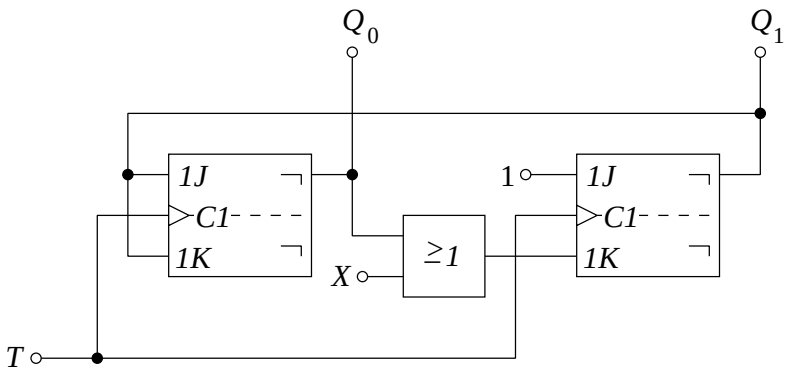


Abbildung 3.31: Synchronzähler mit zwei Ausgängen.

3.7 Synthese von Schaltwerken

Bei der Synthese wird aus einer (häufig verbal) gegebenen Aufgabenstellung ein Schaltwerk entworfen. Dazu ist es notwendig, die Aufgabenstellung mit den Beschreibungsmöglichkeiten eines Schaltwerks darzustellen. Es empfehlen sich folgende Schritte für das Vorgehen:

- Festlegen der Zustandsmenge, die das Schaltwerk einnehmen soll. Daraus ergibt sich die Anzahl der Zustandsvariablen und die Anzahl der erforderlichen Speicherglieder.

- Festlegen des Anfangszustandes. Hier wird meist der Nullvektor angenommen, der durch die Rücksetzeingänge der Flipflops beim Einschalten der Betriebsspannung erzwungen werden kann.

- Definition der Eingangs- und Ausgangsvariablen.

- Darstellung der *zeitlichen* Zustandsfolge in Form eines Zustandsgraphen.

- Aufstellen der Zustandstabelle.

- Herleitung und Minimierung der Übergangs- und Ausgangsfunktion in DNF bzw. KNF aus der Zustandstabelle.

- Darstellung des Schaltwerks in einem *Schaltplan*. Das bedeutet: Übertragen der Schaltfunktionen in ein Schaltnetz, Verbindungen mit den Flipflops herstellen, Kennzeichnen des Zustandsvektors und des Folgezustandsvektors.

- Implementierung des Schaltwerks.

Auch hier sollen wieder zwei Beispiele die Vorgehensweise verdeutlichen.

3.7.1 Umschaltbarer Gray-Code-Zähler

Im ersten Beispiel wollen wir einen zweistelligen, umschaltbaren Gray-Code-Zähler auf Basis von D-Flipflops entwerfen. Kennzeichen des Gray-Codes ist es, dass sich zwischen zwei aufeinander folgenden Codewörtern stets nur eine Bitstelle ändert. Die Umschaltung soll durch eine Eingangsvariable x erfolgen. Für $x = 0$ ist die Zählfolge

$$00, 01, 11, 10, 00 \quad \text{usw.}$$

und für $x = 1$ ist die Zählfolge

$$00, 10, 11, 01, 00 \quad \text{usw.}$$

festgelegt.

- Zuerst ermitteln wir die Anzahl der nötigen Zustände. Da es insgesamt vier verschiedene Codewörter gibt, sind genauso viele Zustände erforderlich. Zur Codierung dieser vier Zustände reichen zwei Zustandsvariablen aus, die in zwei D-Flipflops gespeichert werden.
- Das Schaltwerk beginnt mit dem Anfangszustand $Z(0) = 00$.
- Die Umschaltung erfolgt durch die Eingangsvariable x. Die Ausgangsvariablen sind identisch mit den Zustandsvariablen, weil der Zählzustand angezeigt werden soll. Wir können also auf ein Ausgangsschaltnetz f verzichten.
- Die zeitliche Zustandsfolge, die sich aus der obigen Beschreibung und den Festlegungen ergibt, ist in Abbildung 3.32 dargestellt.

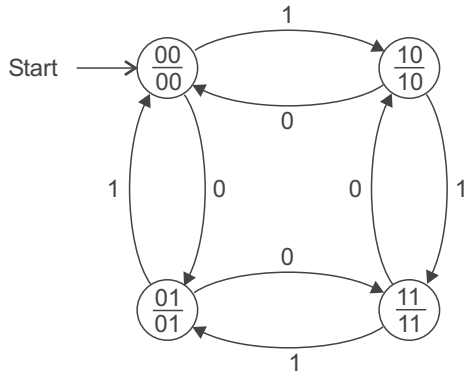


Abbildung 3.32: Zustandsgraph des umschaltbaren Gray-Code-Zählers.

- Zustandstabelle

  Aus dem Zustandsgraphen folgt unmittelbar die Zustandstabelle 3.13. Die linke Seite der Tabelle enthält alle Wertekombinationen, die die Eingangsvariable x und die Zustandsvariablen z_1, z_0 annehmen können. Die rechte Seite der Tabelle enthält die Werte der Folgezustände.

Tabelle 3.13: Zustandstabelle für den umschaltbaren Gray-Code-Zähler.

z_1	z_0	x	z_1^+	z_0^+
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	0	1

- Schaltfunktionen (Übergangsfunktionen)

  Aus der Zustandstabelle können wir die Übergangsfunktionen in der DNF aufstellen:

$$z_1^+ = (z_0 \wedge \overline{x}) \vee (\overline{z}_0 \wedge x) \quad (3.21)$$
$$z_0^+ = (\overline{z}_1 \wedge \overline{x}) \vee (z_1 \wedge x) \quad (3.22)$$

  Die beiden Schaltfunktionen können nicht weiter mit Karnaugh-Diagrammen (vgl. Kurseinheit 1) minimiert werden. z_1^+ entspricht der EXOR-Funktion und z_0^+ der negierten EXOR-Funktion (Äquivalenz-Funktion).

- Zeichnen des Schaltwerkes (Abbildung 3.33).

3.7.2 Zähler mit vorgegebener Zählfolge

Als zweites Beispiel soll ein Zähler mit *JK*-Flipflops entworfen werden. Der Zähler soll folgende Zählfolge durchlaufen:

$$0 \to 15 \to 4 \to 1 \to 5 \to 11 \to 3 \to 8 \to 6 \to 7 \to 0 \quad \text{usw.}$$

- Zuerst ermitteln wir wieder die Anzahl der nötigen Zustände: Da es insgesamt zehn verschiedene Zählerwerte gibt, sind genauso viele Zustände erforderlich. Zur Codierung dieser zehn Zustände sind vier Zustandsvariablen nötig, die in vier *JK*-Flipflops gespeichert werden.

- Das Schaltwerk beginnt mit dem Anfangszustand $Z(0) = 0000$.

- Es gibt *keine* Eingangsvariablen, d.h. wir entwerfen ein autonomes Schaltwerk. Die Ausgangsvariablen sind identisch mit den Zustandsvariablen, weil der Zählerwert direkt angezeigt werden kann. Wir können also wieder auf ein Ausgangsschaltnetz f verzichten.

- Die Abfolge der Zustände wird durch den Zustandsgraphen nach Abbildung 3.34 dargestellt.

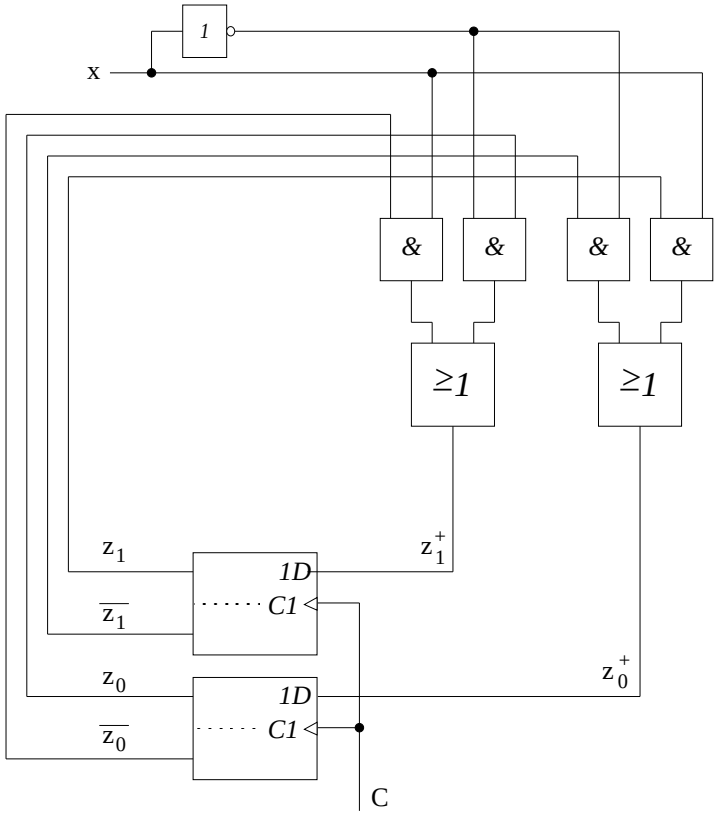


Abbildung 3.33: Schaltplan des umschaltbaren Gray-Code-Zählers.

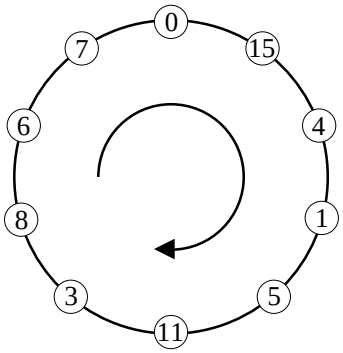


Abbildung 3.34: Zustandsgraph für den Zähler mit vorgegebener Zählfolge.

- Zustandstabelle

  Die Zustandstabelle erstellen wir speziell für JK-Flipflops. Aufgrund der Funktionsweise von JK-Flipflops können in der Zustandstabelle bei den Belegungen für die JK-Eingänge don't care-Symbole ($\times$) eingesetzt werden. Soll zum Beispiel der Ausgang Q_0 vom Zustand 0 in den Zustand 1 übergehen, so geschieht das sowohl mit der Belegung $J_0 = K_0 = 1$ (Kippen des Flipflops) als auch mit der Belegung $J_0 = 1$ und $K_0 = 0$ (Setzen des Flipflops). Daraus folgt, dass die Belegung von K_0 in diesem Fall beliebig ist.

Zum Entwurf des Schaltwerks tragen wir zuerst alle möglichen Zählzustände in die Spalte für den Zeitpunkt t ein. Anschließend werden die Folgezustände in die Spalte für den Zeitpunkt $t+1$ eingetragen. Nun sieht man, welche Ausgänge sich wie verändern müssen, und kann mit Hilfe der Tabelle 3.8 die J- und K-Eingänge belegen. Wir erhalten für das Schaltwerk die Zustandstabelle nach Tabelle 3.14.

Tabelle 3.14: Zustandstabelle für den Zähler mit vorgegebener Zählfolge.

t									$t+1$
J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0	$Q_3 \dots Q_0$	$Q_3 \dots Q_0$
1	×	1	×	1	×	1	×	0000	1111
×	1	×	0	×	1	×	1	1111	0100
0	×	×	1	0	×	1	×	0100	0001
0	×	1	×	0	×	×	0	0001	0101
1	×	×	1	1	×	×	0	0101	1011
×	1	0	×	×	0	×	0	1011	0011
1	×	0	×	×	1	×	1	0011	1000
×	1	1	×	1	×	0	×	1000	0110
0	×	×	0	×	0	1	×	0110	0111
0	×	×	1	×	1	×	1	0111	0000

- Schaltfunktionen (Übergangsfunktionen)

  Mit Hilfe der Zustandstabelle können wir nun die acht Schaltfunktionen für die Steuereingänge der vier JK-Flipflops in einer DNF- oder KNF-Darstellung ableiten. Vor der Implementierung sollten diese Schaltfunktionen noch minimiert werden, um den Schaltungsaufwand zu reduzieren. Aus Platzgründen wollen wir dies hier aber nur für das Flipflop mit dem Index 1 tun. Wir beschränken uns also auf die Schaltfunktionen für die Eingänge J_1 und K_1.

  Um die Schaltfunktionen zu minimieren, wollen wir die in Kurseinheit 1 eingeführten *Karnaugh-Diagramme* anwenden. In dem hier vorliegenden Fall gibt es neben den 0/1-Einträgen auch noch die don't care-Belegungen (×). Da von den 16 möglichen Belegungen an den Ausgängen $Q_3Q_2Q_1Q_0$ nur 10 vorkommen, dürfen die restlichen sechs mit × markiert werden. Zusammen mit den don't care-Einträgen aufgrund der Ansteuertabellen der JK-Flipflops ergeben sich dadurch zusätzliche Möglichkeiten zur Vereinfachung[6].

  Karnaugh-Diagramme

[6] Man kann beim don't care wie bei einem „Joker" wählen, ob in dem betreffenden Feld eine 0 oder 1 stehen soll.

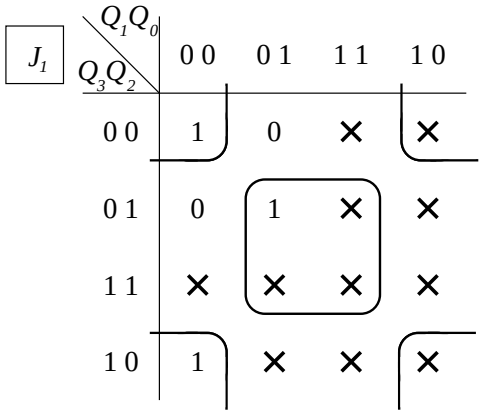


Abbildung 3.35: KV-Tafel für den Eingang J_1.

Wir wollen die beiden gesuchten Schaltfunktionen J_1 und K_1 in einer minimierten DNF darstellen. Dabei müssen wir darauf achten, dass lediglich die zur Päckchenbildung fehlenden Felder mit einer 1 gefüllt werden. Es ist überflüssig, große Päckchen mit don't care-Markierungen zu bilden, die entweder keine 1 abdecken (z.B. Abbildung 3.35, rechte Hälfte) oder die eine 1 doppelt abdecken (z.B. Abbildung 3.35, untere Hälfte).

Wir erhalten aus Abbildung 3.35 als minimale Gleichung für J_1:

$$J_1 = Q_0\, Q_2 \vee \overline{Q_0}\ \overline{Q_2} = Q_0\, Q_2 \vee (\overline{Q_0 \vee Q_2}) \tag{3.23}$$

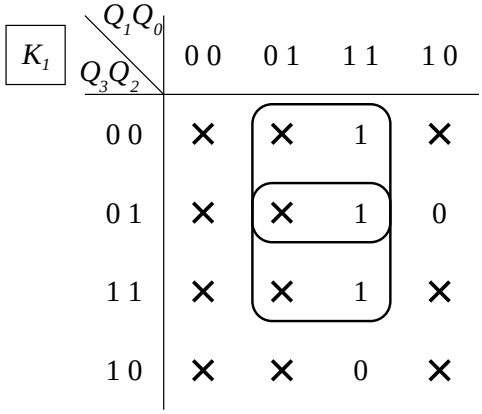


Abbildung 3.36: KV-Tafel für den Eingang K_1.

Aus der KV-Tafel des K_1-Eingangs (Abbildung 3.36) ergibt sich dessen minimale disjunktive Form:

$$K_1 = Q_0\, Q_2 \vee Q_0\, \overline{Q_3} \tag{3.24}$$

- Zeichnen des Schaltwerks

  Abbildung 3.37 zeigt das entsprechende Teilschaltwerk mit dem JK-Flipflop für die Stelle mit der Wertigkeit 2^1.

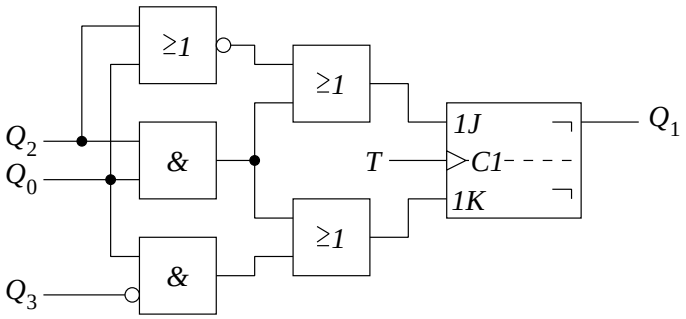


Abbildung 3.37: Teil des Schaltwerks für die Stelle mit der Wertigkeit 2^1.

Selbsttestaufgabe 3.8 (3-Bit-Zähler)

Entwerfen Sie einen 3-Bit-Synchronzähler mit folgender Zählfolge:

$$0 - 1 - 3 - 7 - 6 - 5 - 0 - \cdots$$

Erstellen Sie hierzu:

- *den Zustandsgraph,*
- *die Zustandstabelle,*
- *die minimierten Funktionsgleichungen und*
- *das Schaltbild des Schaltwerks.*

Lösung auf Seite 146

3.7.3 Zustands-Minimierung

Werden zwei Schaltwerke durch die gleiche Folge von Eingabevektoren angesteuert und liefern sie die gleiche Folge von Ausgangsvektoren, so sind sie bezüglich ihres Verhaltens äquivalent. Es kann aber durchaus sein, dass sie intern ganz anders arbeiten bzw. über eine unterschiedliche Zahl von Zuständen verfügen. Häufig enthalten die aus einer textuellen Beschreibung abgeleiteten Zustandgraphen redundante Zustände. Um den Entwurfs- und Hardwareaufwand zu minimieren, sollte man daher die Anzahl der Zustände im Zustandsgraphen minimieren.

redundante Zustände

Um redundante Zustände zu erkennen, muss man die Äquivalenz zweier bzw. mehrerer Zustände prüfen.

Zwei Zustände z_i und z_j sind äquivalent ($z_i \equiv z_j$), wenn sie das gleiche Ausgabeverhalten haben und – für alle Eingabevektoren – äquivalente Folgezustände einnehmen. Wir können folgende Äquivalenzrelation „$\equiv$“ definieren:

$$\begin{aligned} z_i \equiv z_j \quad \Leftrightarrow \quad & \forall x(f(x, z_i) = f(x, z_j)) && (3.25)\\ & \forall x(g(x, z_i) \equiv g(x, z_j)) \quad \text{d.h.} \quad \forall x(z_i^+ \equiv z_j^+) && (3.26)\\ & x \in I \end{aligned}$$

Wenn eine der Bedingungen (3.25) oder (3.26) nicht erfüllt ist, folgt, dass die Zustände z_i und z_j nicht äquivalent sind ($z_i \not\equiv z_j$). Die Bedingung in Gleichung (3.26) macht die Definition der Zustandsäquivalenz rekursiv, ohne dass man ein Rekursionsende sieht. Zum Verständnis ist es hilfreich, das Gegenteil zu betrachten: Zwei Zustände sind nicht äquivalent, wenn es eine natürliche Zahl r gibt, so dass nach r-facher Anwendung von Gleichung (3.26) eine Nicht-Äquivalenz durch Verletzung von (3.25) auftritt. Gibt es ein solches r nicht, dann sind die Zustände äquivalent. Wegen der endlichen Zahl $s = |S|$ von Zuständen kann man sich auf $r \leq s(s-1)/2$ einschränken.

Aus der Definition der Zustandsäquivalenz können wir ein einfaches Ausschluss-Verfahren zur Zustands-Minimierung ableiten.

Zu Beginn erstellen wir eine Liste sämtlicher Zustandspaare, die (3.25) erfüllen. Für jedes Paar ermitteln wir alle Folge-Zustandspaare $(g(x, z_i), g(x, z_j))$ und tragen diese hinter dem zugehörigen Ausgangs-Zustandspaar ein.

Wegen der Symmetrie-Eigenschaft der Äquivalenzrelation (aus $z_i \equiv z_j$ folgt $z_j \equiv z_i$) müssen nur die Zustandspaare mit $i < j$ für $i, j = 1, \cdots, n$ betrachtet werden.

Außerdem müssen nur Folge-Zustandspaare aufgenommen werden, die sich vom Ausgangs-Zustandspaar unterscheiden. Als nächstes wird überprüft, ob die eingetragenen Folge-Zustandspaare die Bedingung (3.25) der Äquivalenzrelation erfüllen. Wenn ein Folge-Zustandspaar nicht als Ausgangs-Zustandspaar vorkommt, ist es zu streichen. Gleichzeitig impliziert die Streichung eines Folge-Zustandspaares, dass auch das zugehörige Ausgangs-Zustandspaar nicht äquivalent sein kann und ebenfalls zu streichen ist.

Kennt man die Menge der äquivalenten Zustandspaare, so kann man mit der Transitivitäts-Eigenschaft eventuell neue Zustandspaare bestimmen. Wenn beispielsweise die Zustands-Paare (z_i,z_j) und (z_j,z_k) äquivalent sind, so ist auch das Zustandspaar (z_i,z_k) äquivalent und die drei Zustände bilden eine *Äquivalenzklasse*. Alle Zustände einer Äquivalenzklasse können durch einen einzigen Zustand ersetzt werden.

Äquivalenzklasse

Im Folgenden soll die Anwendung des gerade beschriebenen Verfahrens an einem Beispiel demonstriert werden. Das Schaltwerksverhalten sei durch die in Tabelle 3.15 gegebene Zustandstabelle gegeben. Es handelt sich um ein Moore-Schaltwerk mit einem Ein- und einem Ausgang.

Tabelle 3.15: Zustandstabelle für das Beispiel zur Zustands-Minimierung.

Zustand	Folge-Zustand für X=0	X=1	Ausgang
1	4	3	0
2	6	8	0
3	5	4	1
4	1	5	0
5	3	1	1
6	6	2	1
7	2	8	0
8	3	7	1

In Tabelle 3.16 sind drei Stufen zur Bestimmung der Äquivalenzklassen dargestellt. In Stufe 0 werden zunächst die Ausgangs-Zustandspaare ermittelt. Dann werden hinter jedes Ausgangs-Zustandspaar die entsprechenden Folge-Zustandspaare geschrieben. In Stufe 1 wird von oben nach unten geprüft, ob die rechts stehenden Folge-Zustandspaare auch als Ausgangs-Zustandspaare vorhanden sind. Wenn dies nicht der Fall ist, werden die betreffenden Folge-Zustandspaare und Ausgangs-Zustandspaare gestrichen. Bei der Stufe 1 wurden bereits Streichungen aus vorangehenden Zeilen berücksichtigt. So erfolgt z.B. in Zeile 9 und 10 die Streichung des Folge-Zustandspaars (1,2). In Stufe 2 verfährt man in gleicher Weise und stellt dann fest, dass bei einem weiteren Durchlauf keine Streichungen mehr möglich sind.

Tabelle 3.16: Ermittlung der Äquivalenzklassen.

Stufe 0	
(1,2)	(4,6) (3,8)
(1,4)	(3,5)
(1,7)	(2,4) (3,8)
(2,4)	(1,6) (5,8)
(2,7)	(2,6)
(3,5)	(1,4)
(3,6)	(5,6) (2,4)
(3,8)	(3,5) (4,7))
(4,7)	(1,2) (5,8)
(5,6)	(1,2) (3,6)
(5,8)	(1,7)
(6,8)	(3,6) (2,7)

Stufe 1	
~~(1,2)~~	~~(4,6)~~ (3,8)
(1,4)	(3,5)
(1,7)	(2,4) (3,8)
~~(2,4)~~	~~(1,6)~~ (5,8)
~~(2,7)~~	~~(2,6)~~
(3,5)	(1,4)
~~(3,6)~~	(5,6) ~~(2,4)~~
(3,8)	(3,5) (4,7)
~~(4,7)~~	~~(1,2)~~ (5,8)
~~(5,6)~~	~~(1,2)~~ (3,6)
(5,8)	(1,7)
~~(6,8)~~	~~(3,6)~~ (2,7)

Stufe 2	
~~(1,2)~~	~~(4,6)~~ (3,8)
(1,4)	(3,5)
~~(1,7)~~	~~(2,4)~~ (3,8)
~~(2,4)~~	~~(1,6)~~ (5,8)
~~(2,7)~~	~~(2,6)~~
(3,5)	(1,4)
~~(3,6)~~	(5,6) ~~(2,4)~~
~~(3,8)~~	(3,5) ~~(4,7)~~
~~(4,7)~~	~~(1,2)~~ (5,8)
~~(5,6)~~	~~(1,2)~~ (3,6)
~~(5,8)~~	~~(1,7)~~
~~(6,8)~~	~~(3,6)~~ (2,7)

Aus der Stufe 2 der Tabelle 3.16 können wir ablesen, dass die beiden Zustandspaare (1,4) und (3,5) äquivalent sind. Wir können daher pro äquivalentes Zustandspaar je einen Zustand (z.B. die Zustände 4 und 5) mit dem jeweils anderen Zustand vereinigen. Man beachte, dass die verbleibenden Zustände (1 und 3) wechselseitig Folgezustände voneinander sind.

Selbsttestaufgabe 3.9 (Benötigte Zustände ermitteln)

Eine Zustands-Minimierung ergab, dass die ursprünglich betrachteten acht Zustände (1 bis 8) folgende äquivalente Zustandspaare enthalten: (1,3), (4,5), (1,6) und (5,7). Wie viele Zustände werden insgesamt noch benötigt?

Lösung auf Seite 148

Nachdem wir die minimale Zahl der Zustände ermittelt haben, stellen sich folgende Fragen: Wie viele Bits (Flipflops) werden zu deren Codierung benötigt? Welche Codewörter sollen den jeweiligen Zuständen zugeordnet werden?

3.7.4 Zustands-Codierung

Nehmen wir an, dass nach der Zustandsminimierung n Zustände übrig bleiben. Zur binären Codierung dieser Zustände werden *mindestens* $\lceil \log_2 n \rceil$ Flipflops benötigt.

Gehen wir davon aus, dass n eine Zweierpotenz ist, so gibt es genau $n!$ mögliche Zustands-Codierungen. Wenn dagegen $\log_2 n < \lceil \log_2 n \rceil$ ist, so sind auch mehr[7] Binär-Codewörter als Zustände vorhanden. Folglich gibt es dann auch ein Vielfaches der $n!$ möglichen Zustands-Codierungen.

Schon bei wenig komplexen Zustandsgraphen müssen wir also viele Möglichkeiten zur Zustands-Codierung in Betracht ziehen. Die gewählte Zustands-Codierung hat großen Einfluss auf die Hardware-Komplexität (Kosten) und das Zeitverhalten der Rückkopplungsschaltnetze. Daher ist es notwendig, eine geeignete Zustands-Codierung zu finden. Um den exponentiellen Aufwand bei einer enumerativen Suche zu vermeiden, verwendet man dazu Heuristiken.

Wir wollen drei Heuristiken für die Zustandscodierung kurz skizzieren:

- Minimale Bit-Änderung,
- Priorisierte Nachbar-Codierungen,
- Hot-one-Codierung.

Minimale Bit-Änderung

Die grundlegende Idee dieser Heuristik besteht darin, dass sich bei Zustandsübergängen (Transitionen) nur möglichst wenige Bits ändern sollen. Man erhofft sich durch dieses Prinzip, dass dadurch auch nur wenig komplexe Rückkopplungsschaltnetze erforderlich sind. Auch unter dem Gesichtspunkt des Energieverbrauchs ist dieses Prinzip sinnvoll – schließlich verbraucht jeder Zustandswechsel bei einem Flipflop auch Energie, die in Wärme umgewandelt wird. Die Energie-Ökonomie ist besonders bei mobilen Geräten wichtig.

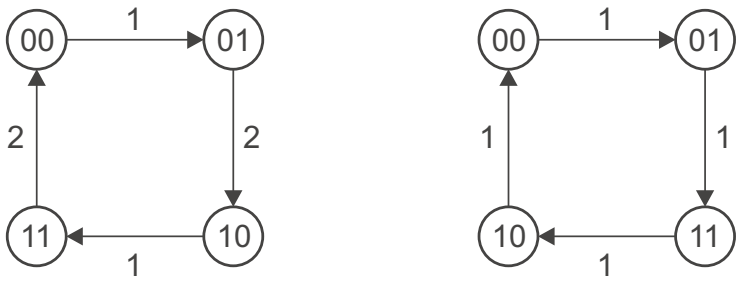


Abbildung 3.38: Zwei unterschiedliche Codierungen für einen 2-Bit-Zähler.

In Abbildung 3.38 werden für einen Modulo-4-Zähler zwei unterschiedliche Zustands-Codierungen gegenübergestellt. Auf der linken Seite wird den Zuständen der Dual-Code zugeordnet. Im Vergleich dazu sieht man auf der rechten Seite von Abbildung 3.38 die Codierung mit minimaler Bit-Änderung.

[7] redundante

Der dort verwendete Gray-Code ist ein einschrittiger Code, der gewährleistet, dass sich zwei benachbarte Codewörter jeweils nur um ein Bit unterscheiden. Der Gray-Code eignet sich besonders für Zustandsgraphen, die lange lineare Abfolgen von Zuständen enthalten. Zum besseren Vergleich ist in Abbildung 3.38 an den Transitionen jeweils angegeben, wie viele Bits sich beim Übergang zum nächsten Folgezustand ändern. Die Summe dieser Kantengewichte eines Zustandsgraphen ist ein Maß für die Güte der Codierung. Im Idealfall ist diese Summe gleich der Anzahl der Transitionen.

Priorisierte Nachbar-Codierungen

Im Gegensatz zu Abbildung 3.38 sind bei komplexeren Zustandsgraphen die Zustandsübergänge normalerweise unregelmäßig verteilt. Nach dem oben beschriebenen Idealfall sollten alle durch eine Transition verbundenen Zustände möglichst durch zwei *benachbarte* Codierungen dargestellt werden, die sich in nur einer Zustandsvariablen unterscheiden.

Es ist jedoch schwierig zu entscheiden, in welcher Reihenfolge den Zuständen diese benachbarten Codierungen zugeordnet werden sollen. Sobald einem Paar zwei benachbarte Codierungen zugeordnet wurden, ist man bei der Codierung weiterer Zustände, die ebenfalls mit einem oder beiden dieser Zustände verbunden sind, eingeschränkt.

Es wurden daher Regeln zur Priorisierung bei der Vergabe benachbarter Codierungen aufgestellt. Höchste Priorität erhalten dabei Zustände, die bei gleichem Eingabevektor auch den gleichen Folgezustand haben. Wenn sich die Codierungen solcher Zustände genau in einem Bit unterscheiden, wird in der Übergangsschaltfunktion g jeweils ein Literal eliminiert und damit der Hardwareaufwand reduziert.

Die zweithöchste Priorität haben Paare von Zuständen, die den gleichen Zustand als Vorgänger haben und bei denen die Eingabevektoren für die Transitionen sich genau um ein Bit unterscheiden. Hierbei ist ebenfalls zu erwarten, dass dadurch bei den Übergangsschaltnetzen Literale eliminiert werden können.

Die dritte und letzte Priorität haben schließlich Zustände, die für gleiche Eingabevektoren jeweils die gleiche Ausgabe liefern. Ähnlich wie bei der höchsten Priorität erhofft man sich hier eine Vereinfachung der Ausgangsschaltfunktion f und damit auch Einsparungen bei der Hardware zur Implementierung eines entsprechenden Schaltnetzes.

Hot-one-Codierung

Diese einfache, wenn auch redundante Codierung, ordnet jedem Zustand genau ein Flipflop zu. Dies bedeutet, dass stets ein Flipflop gesetzt ist, während alle anderen zurückgesetzt sind. Der Zustand des Schaltwerks entspricht dem gesetzten Flipflop. Für Moore-Automaten kann das Ausgangsschaltnetz durch diese Art der Zustandscodierung sehr einfach entworfen werden. Es besteht pro Ausgangsvariable aus einem OR-Schaltglied, dessen Eingänge mit allen Zuständen (d.h. Flipflop-Ausgängen) verbunden sind, bei denen diese Ausgangsvariable den Wert 1 liefern soll. Für Mealy-Automaten ist die Ansteuerung des OR-Schaltglieds deutlich komplizierter. Sie erfolgt durch die AND-Verknüpfung

mit einem Vorgängerzustand und der Decodierung des für eine 1-Ausgabe zugeordneten Eingangsvektors. Die benötigten Schaltfunktionen können sehr gut mit programmierbaren Logikbausteinen implementiert werden (vgl. Abschnitt 3.8.1).

Bei Verwendung von D-Flipflops kann auch die Übergangsfunktion sehr einfach bestimmt werden. Jeder Knoten des Zustandsgraphen ist genau einem Flipflop zugeordnet. Die Schaltfunktion des zugehörigen D-Eingangs ergibt sich aus einer OR-Verknüpfung von Konjunktionstermen, die jeweils einer einlaufenden Kante in diesen Knoten entsprechen. Mit jedem dieser Konjunktionsterme wird *eine* Bedingung für das Aktivieren des Zustands geprüft. Diese Bedingung enthält einerseits den vorausgehenden Zustand (Ausgang Q des zugeordneten Flipflops) und die Belegung des Eingabevektors, die diesen Zustandsübergang auslöst. Darin treten allerdings nur diejenigen Eingangsvariablen auf, die Einfluss auf den Zustandsübergang haben.

Die Hot-one-Codierung sollte wegen ihrer Hardware-Redundanz nur angewandt werden, wenn die Zahl der Zustände klein ist (z.B. < 10). Bei Schaltwerken mit einer größeren Zahl von Zuständen sollte man die oben beschriebenen Codierungen mit minimaler Anzahl von Bits (Flipflops) verwenden. Bei Problemstellungen mit einer sehr großen Zahl von Zuständen (z.B. > 1000) ist ein systematischer Entwurf nach der oben beschriebenen Methode nicht mehr möglich. Man verwendet in diesem Fall *kooperierende* Schaltwerke (vgl. Kurseinheit 4).

3.8 Implementierung von Schaltwerken

Wie wir gesehen haben, bestehen Schaltwerke aus einem Register und ein oder zwei Schaltnetzen für die Funktionen f und g. Zur Implementierung von Schaltwerken müssen also im Wesentlichen Schaltnetze aufgebaut werden, die diese beiden Funktionen realisieren. Diese Schaltnetze können entweder durch Verdrahten einzelner Schaltglieder (AND, OR, NOT) oder mit Hilfe programmierbarer Logikbausteine implementiert werden. Wir wollen im Folgenden die letztgenannte Möglichkeit genauer betrachten.

3.8.1 Programmierbare Logikbausteine

Wie aus Kurseinheit 1 bekannt, können alle Schaltfunktionen durch eine Normalform (DNF oder KNF) dargestellt werden. Es ist daher möglich, eine dieser beiden Normalformen als Grundlage für die Konstruktion eines universell verwendbaren Schaltnetzes auszuwählen. Den programmierbaren Logikbausteinen wird meist eine DNF zugrunde gelegt. Es handelt sich um *dreistufige* Schaltnetze, denn sie enthalten in der ersten Stufe Inverter für die Eingangsvariablen, in der zweiten AND-Verknüpfungen zur Bildung der Produktterme und schließlich in der dritten Stufe OR-Verknüpfungen dieser Produktterme.

Abbildung 3.39a zeigt den Strukturaufbau eines Schaltnetzes in der DNF. Der Eingabevektor X hat hier die Komponenten x_2, x_1, x_0 – sie liegen sowohl direkt als auch invertiert vor. In den AND-Gliedern werden die Produktterme $p_7, \ldots, p_0$ gebildet. Die Produktterme werden in den OR-Gliedern disjunktiv

verknüpft und bilden die Ausgangsvariablen. Die Menge der Kreuzungspunkte der Eingangsvariablen mit den Eingängen der AND-Glieder wird *AND-Matrix*, die Menge der Kreuzungspunkte der Produktterme mit den Eingängen der OR-Glieder *OR-Matrix* genannt. In den programmierbaren Logikbausteinen können die für das Schaltnetz erforderlichen Kreuzungspunkte der AND-Matrix und/oder der OR-Matrix programmiert werden.

AND-Matrix

OR-Matrix

Abbildung 3.39b zeigt einen programmierbaren Logikbaustein mit den einzelnen Verknüpfungsgliedern, Abbildung 3.39c ist eine vereinfachte Darstellung von Abbildung 3.39b. Hier sind die Eingangsleitungen der AND- und OR-Glieder nicht mehr dargestellt. Jeder Punkt in der AND-Matrix bedeutet, dass die entsprechende Eingangsvariable einen Beitrag zu der mit p_i gekennzeichneten AND-Verknüpfung liefert. Jeder Punkt der OR-Matrix bedeutet, dass dieser Produktterm einen Beitrag zur OR-Verknüpfung liefert. Nach der Programmierbarkeit der AND-Matrix und/oder der OR-Matrix unterscheidet man vier Arten von programmierbaren Logikbausteinen, die in Tabelle 3.17 dargestellt sind.

Tabelle 3.17: Arten von programmierbaren Logikbausteinen.

Art	AND-Matrix	OR-Matrix
ROM	fest	fest
PROM, EPROM, EEPROM	fest	programmierbar
PAL	programmierbar	fest
PLA	programmierbar	programmierbar

Während die oben beschriebene Grundstruktur je nach Art des programmierbaren Logikbausteins stets gleich bleibt, erfolgt die eigentliche Programmierung der zu implementierenden Schaltfunktionen, indem elektrische Verbindungen zwischen den Kreuzungspunkten hergestellt werden. Dies kann auf drei verschiedene Arten geschehen:

1. Bei der Herstellung wird die anwendungsspezifische Programmierung durch eine Maske mit einem entsprechenden Verbindungsmuster verwendet (*Maskenprogrammierung*). Die Herstellung von Logikbausteinen zur Implementierung verschiedener Schaltfunktionen unterscheidet sich also nur in diesem einen Fertigungsschritt. Beispiele sind so genannte *Read-Only Memories (ROM)*.

   Maskenprogrammierung

2. Die Programmierung erfolgt elektrisch und ist *irreversibel*. Die Logikbausteine werden alle in gleicher Weise hergestellt und enthalten *Schmelzsicherungen* (fuses), die durch kurzzeitiges Anlegen von Überspannungen zerstört werden. Dies bedeutet, dass zunächst alle möglichen Matrixverbindungen vorhanden sind und dass die nicht benötigten Verbindungen entfernt werden. Der Zustand dieser Matrixverbindungen wird mit Hilfe einer *fuse map* beschrieben, die zur Programmierung des Logikbausteins in einer Datei abgelegt wird. Beispiele sind so genannte *Programmable Read-Only Memory (PROM)*, *Programmable Array Logic (PAL)* und *Programmable Logic Array (PLA)*.

   Schmelzsicherungen

   fuse map

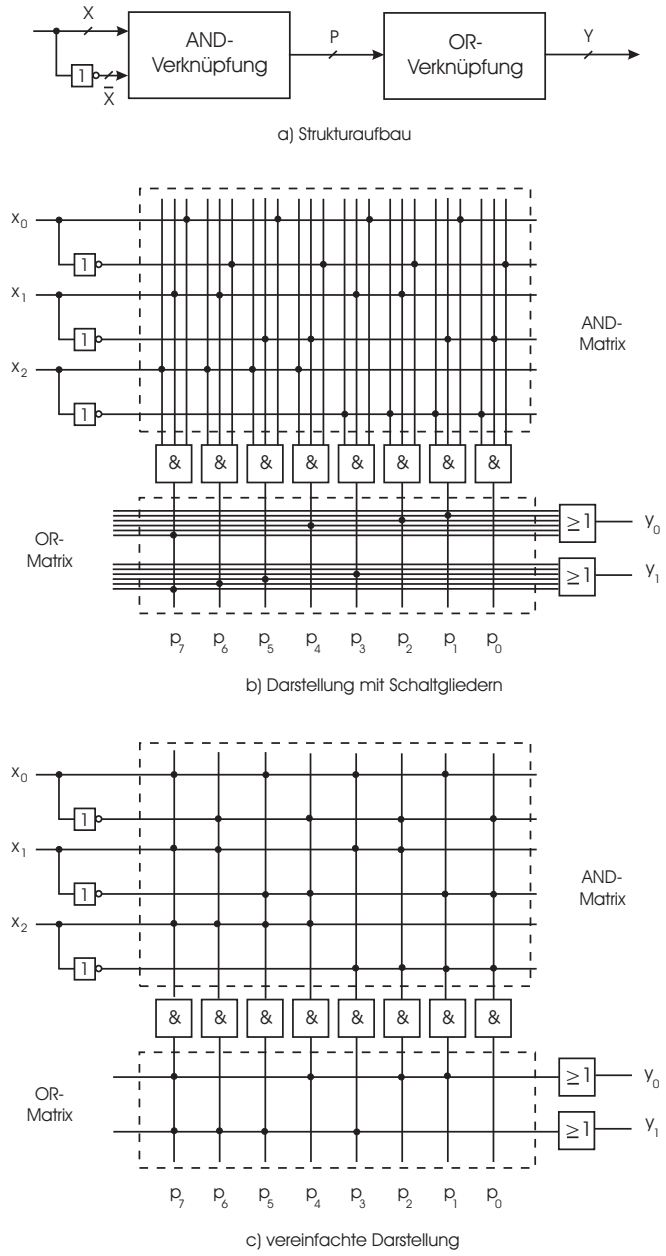


Abbildung 3.39: Struktur eines programmierbaren Logikbausteins.

löschbare Logikbausteine

3. Eine fehlerhafte Programmierung kann bei den Logikbausteinen der letztgenannten Art nicht korrigiert werden. Daher hat man *löschbare* (erasable) Logikbausteine entwickelt, bei denen die Programmierung entweder durch Bestrahlung mit UV-Licht oder auch elektrisch gelöscht werden kann. Beispiele sind so genannte *Erasable PROM (EPROM)* oder *Electrically Erasable PROM (EEPROM).*

Für die Programmierung von PROM, PAL, PLA und EPROM benötigt der Anwender ein spezielles Programmiergerät. Häufig wird er vom Hersteller durch zusätzliche Software unterstützt, die eine komfortable Eingabe der Schaltfunktionen, deren automatische Minimierung und eine optimale Abbildung auf den

Zielbaustein ermöglicht.

Eine Erweiterung der Funktionalität bieten so genannte GALs (*Gate Array Logic*), die neben einem programmierbaren PAL für jeden Ausgang auch noch ein Flipflop bereitstellen. Dessen Ausgang kann wiederum auf die Eingänge des Schaltnetzes zurück gekoppelt werden. Mit einem solchen Logikbaustein kann also ein komplettes Schaltwerk implementiert werden. Die Programmierung kann bei neueren Bausteinen (z.B. Lattice ispGALs[8]) sogar im laufenden Betrieb erfolgen und wird mit Hilfe von Hardwarebeschreibungssprachen unterstützt.

Gate Array Logic

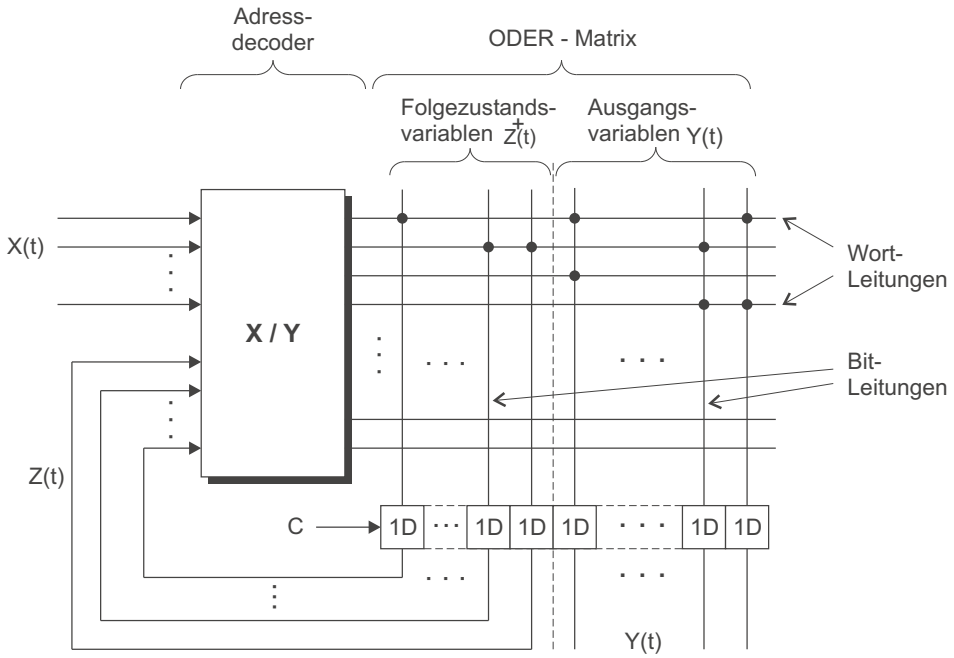


Abbildung 3.40: Mikroprogrammsteuerwerk auf Basis eines Speichers.

3.8.2 Mikroprogrammsteuerwerke

Mit einem Speicherbaustein[9] kann sehr leicht ein programmierbares Schaltwerk implementiert werden. Die beiden Schaltfunktionen f und g werden dann in zwei Teilbereichen des Speichers abgelegt. Da man ein derartiges Schaltwerk häufig zur Realisierung der Ablaufsteuerung in einem Prozessor verwendet, wird es auch als *Mikroprogrammsteuerwerk* bezeichnet. Sowohl das Holen als auch das Ausführen einzelner Maschinenbefehle kann mit Hilfe eines entsprechenden Mikroprogramms sehr komfortabel und flexibel implementiert werden. In der Kurseinheit 4 werden wir noch einige Beispiele hierfür vorstellen.

Der Adresseingang des Speichers wird durch die Konkatenation von Eingabevektor X und Zustandsvektor Z angesteuert. Zu jeder angelegten Adresse aktiviert der Adressdekoder genau eine Wortleitung mit 1 und steuert damit die

[8] *isp* steht für in-system programmable.

[9] ROM, PROM, EPROM oder EEPROM.

Bitleitungen an. Wenn der Vektor X den höherwertigen Adressbits zugeordnet wird, kann man damit im Speicher die Startadresse des Mikroprogramms festlegen. Bei einem Prozessor verwendet man hierzu den so genannten *Operationscode* oder kurz *Opcode*, der über einen weiteren programmierbaren Logikbaustein in die eigentliche Startadresse umgewandelt wird. Die Folgezustandsvariablen Z^+ bilden zusammen mit dem gleichzeitig zwischengespeicherten Ausgangsvektor[10] Y einen *Mikrobefehl*. Jeder Mikrobefehl kann über die Folgezustandsvariablen nachfolgende Mikrobefehle adressieren. Die Auswahl der in einem Mikrobefehl auszuführenden Operationen erfolgt durch die Belegung des zugehörigen Ausgangsvektors Y.

[10]Lässt man die D-Flipflops vor dem Ausgangsvektor Y weg, so erhält man ein Mealy-Schaltwerk.

3.9 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 3.1 von Seite 96

a)

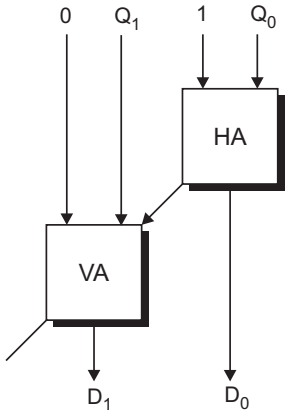


Abbildung 3.41: Aufbau des Inkrementier-Schaltnetzes für einen reinen Vorwärtszähler.

b)

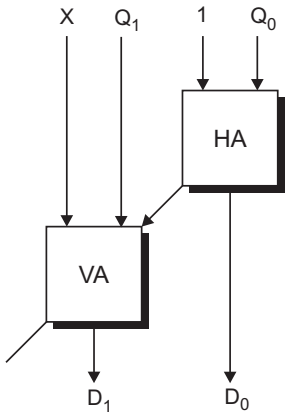


Abbildung 3.42: Aufbau des Schaltnetzes für einen umschaltbaren Vor-/Rückwärtszähler. Für $X = 0$ wird wie in a) imkrementiert. Für $X = 1$ wird die Zweierkomplement-Darstellung von -1 = 11 als zweiter Summand in den 2-Bit-Addierer eingespeist. Dadurch wird der Zustand mit jedem Takt dekrementiert.

Selbsttestaufgabe 3.2 von Seite 100

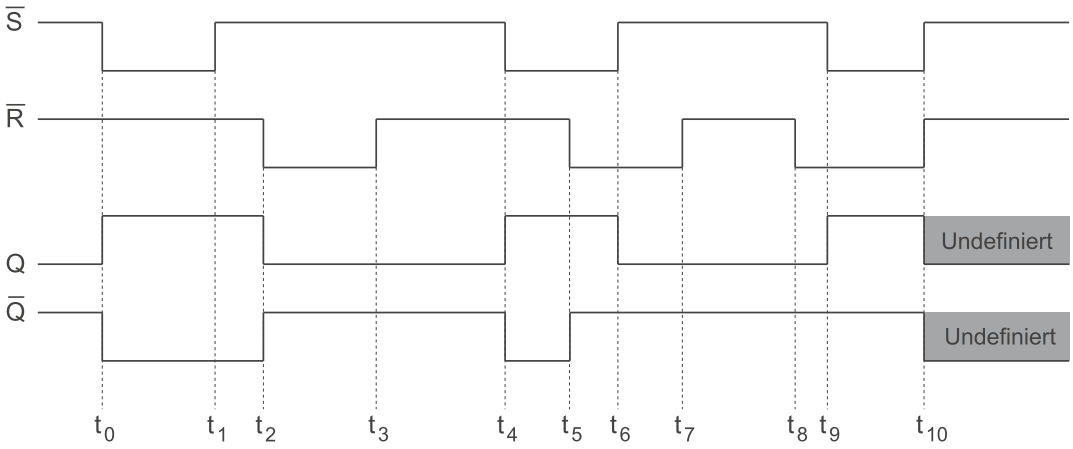


Abbildung 3.43: Zeitverhalten eines SR-Latches mit verzögerungs*freien* NAND-Schaltgliedern.

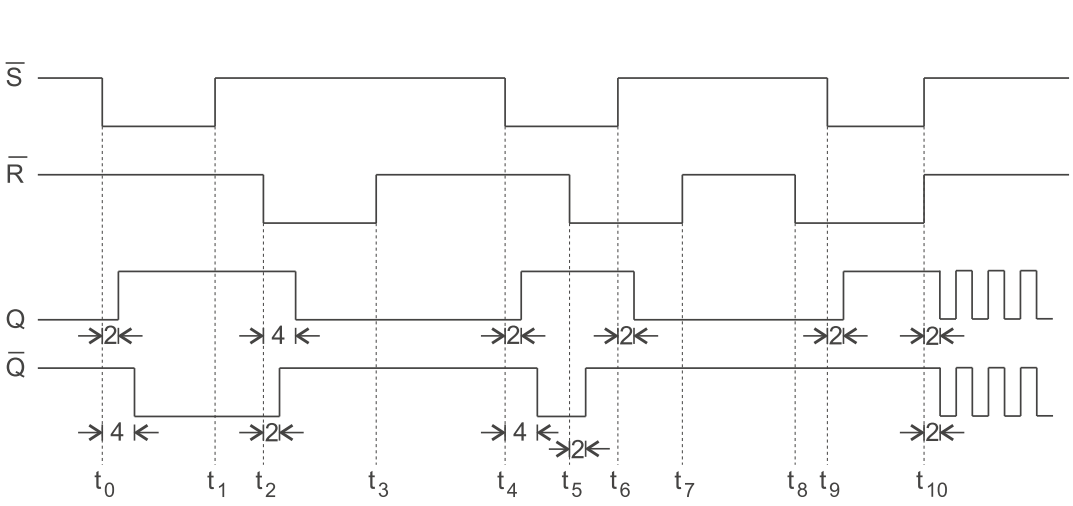


Abbildung 3.44: Zeitverhalten eines SR-Latches mit verzögerungs*behafteten* NAND-Schaltgliedern.

Selbsttestaufgabe 3.3 von Seite 103

Während der Takt 1 ist, sind die Ausgänge des Inkrementier-Schaltnetzes direkt mit seinen Eingängen verbunden. Nehmen wir an, dass der Inkrementierer zu jeder 2-Bit-Eingabe den um 1 inkrementierten Wert in genau 1 ns liefert, so zählt der Zähler mit ca. 1 GHz, solange das Taktsignal 1 ist. Da die Bits des inkrementierten Wortes aber mit unterschiedlichen Verzögerungszeiten an den Ausgängen der Latches erscheinen, ist in der Praxis das Schaltverhalten nicht mehr vorhersehbar.

Selbsttestaufgabe 3.4 von Seite 105

Wie wir bei der Beschreibung des *D-MS*-Flipflops gesehen haben, ist die Verzögerungszeit des Takt-Inverters unbedingt erforderlich, um Wirk- und Kipp-

intervall voneinander zu trennen. Wenn man den Takt-Inverter umdreht, wird das Slave-Latch *vor* dem Master-Latch freigegeben. Somit wäre in Abbildung 3.11 das Wirkintervall um die fallende Flanke von $\overline{C}$ positioniert und es würde durch das Kippintervall des Slave-Latches überdeckt, weil dieses schon mit dem $0 \rightarrow 1$-Übergang von C beginnen würde. Das resultierende *D-MS*-Flipflop wäre also transparent und somit nicht zum Aufbau eines Schaltwerks geeignet.

Selbsttestaufgabe 3.5 von Seite 115

Die Lösung ist in Abbildung 3.45 dargestellt. Die Multiplexereingänge werden von links nach rechts über $S_1S_0 = 00$, 01, 10 und 11 ausgewählt.

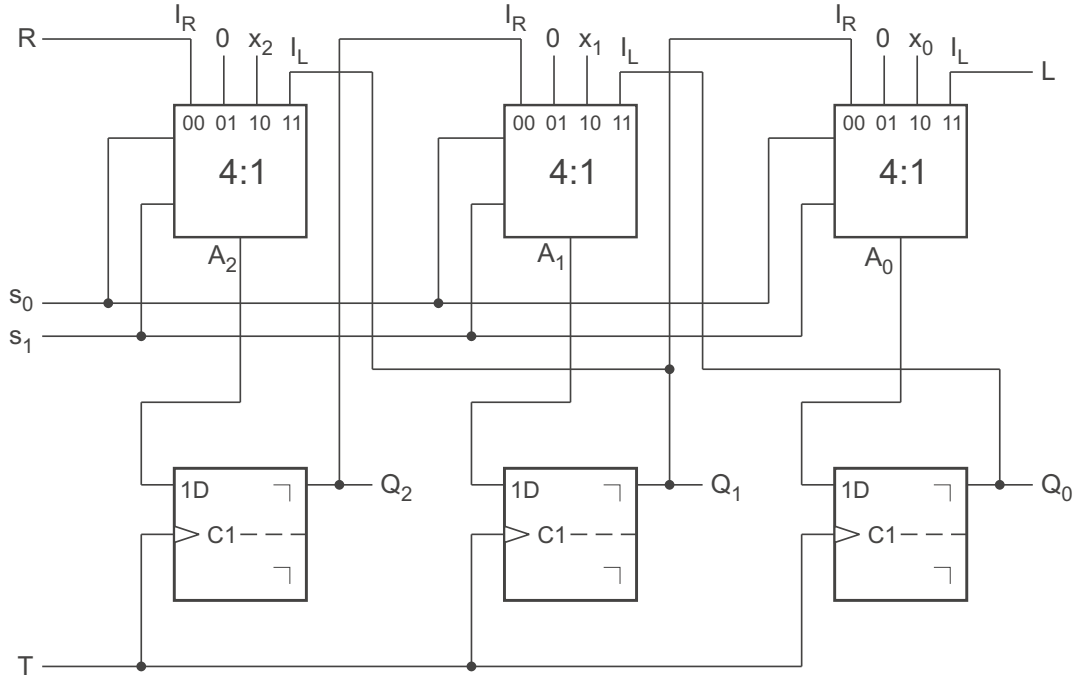


Abbildung 3.45: Steuerbares Schieberegister.

Selbsttestaufgabe 3.6 von Seite 122

Die minimale Periodendauer T_{min} ergibt sich durch

$$T_{min} = T_W + T_{WK} + T_K + T_{KW} \ .$$

Hierbei sind $T_W = 2\ ns$ und $T_K = 1\ ns$ in der Aufgabenstellung bereits vorgegeben. Aus der ersten Rückkopplungsbedingung ergibt sich $T_{WK} \geq -T_{gt} = -0,1\ ns$. Aus der zweiten Rückkopplungsbedingung ergibt sich $T_{KW} \geq T_g = 4\ ns$. Damit folgt

$$T_{min} = 2 - 0,1 + 1 + 4\ ns = 6,9\ ns \ .$$

Die maximale Taktfrequenz entspricht dem Kehrwert der minimalen Periodendauer, also

$$f_{max} = \frac{1}{6,9 \cdot 10^{-9} s} \approx 145 \text{ MHz}$$

Selbsttestaufgabe 3.7 von Seite 126

a) Wir erhalten die Zustandstabelle gemäß Tabelle 3.18.

Tabelle 3.18: Zustandstabelle für Selbsttestaufgabe 3.7.

X	J_0	K_0	J_1	K_1	Q_0^n	Q_1^n	Q_0^{n+1}	Q_1^{n+1}
0	0	0	1	0	0	0	0	1
0	1	1	1	0	0	1	1	1
0	1	1	1	1	1	1	0	0
1	0	0	1	1	0	0	0	1
1	1	1	1	1	0	1	1	0
1	0	0	1	1	1	0	1	1
1	1	1	1	1	1	1	0	0

Aus der Tabelle ergeben sich die Zählzyklen (wenn Q_0 dem Wert 2^0 und Q_1 dem Wert 2^1 entspricht):

- für $X = 0$ zählt das Schaltwerk in der Reihenfolge 0–2–3.
- für $X = 1$ zählt das Schaltwerk in der Reihenfolge 0–2–1–3.

b) Der Zustandsgraph ist eine grafische Darstellung der Zählfolgen. Die Knoten enthalten die jeweiligen (Zähl-)Zustände, während an den Kanten die Zustandsübergangsbedingungen eingetragen sind.

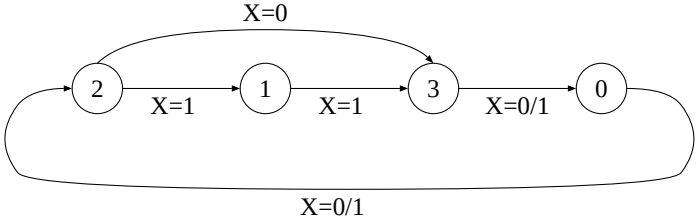


Abbildung 3.46: Zustandsgraph für das Schaltwerk von Selbsttestaufgabe 3.7.

Selbsttestaufgabe 3.8 von Seite 133

Den Zustandsgraphen für den Zähler zeigt Abbildung 3.47.

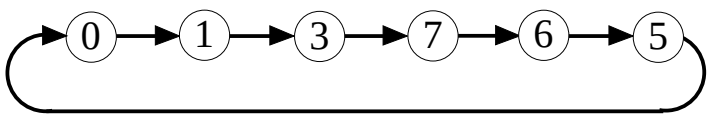


Abbildung 3.47: Zustandsgraph für den 3-Bit–Synchronzähler.

Da keine Vorgaben gemacht wurden, welche Flipflop-Typen zu verwenden sind, wählen wir D-Flipflops, weil sich damit wesentlich leichter synchrone Zähler aufbauen lassen als mit JK-Flipflops. Auch die Automatentabelle (Tabelle 3.19) wird sehr einfach. Die Spalten der Ausgänge des Zeitpunktes t_{n+1} sind mit den Spalten der D-Eingänge identisch.

Tabelle 3.19: Zustandstabelle für Selbsttestaufgabe 3.8.

t_n						t_{n+1}		
Q_2	Q_1	Q_0	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	1
0	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	0
1	1	0	1	0	1	1	0	1
1	0	1	0	0	0	0	0	0

Wir minimieren nun die Funktionsgleichungen für die D-Eingänge. Da die Zustände mit dem Wert 2 und 4 nicht auftreten, werden sie als *don't care* ($\times$) gekennzeichnet.

- D_0:

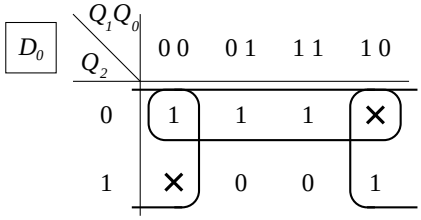


Abbildung 3.48: KV-Tafel für den Eingang D_0.

Wir erhalten: $D_0 = \overline{Q_2} \vee \overline{Q_0}$.

- D_1:

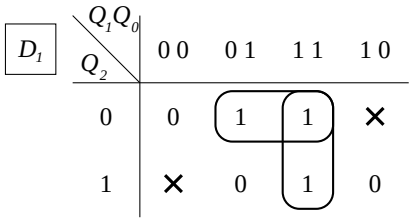


Abbildung 3.49: KV-Tafel für den Eingang D_1.

In diesem Fall nützen die *don't care*-Terme nichts, da sie weder zur Vergrößerung eines Päckchens noch zum Einbinden einer alleinstehenden 1 dienen können. Die minimale DF lautet: $D_1 = Q_0\, Q_1 \vee Q_0\, \overline{Q_2}$.

- D_2:

D_2	Q_1Q_0 / Q_2	00	01	11	10
	0	0	0	1	×
	1	×	0	1	1

Abbildung 3.50: KV-Tafel für den Eingang D_2.

Die minimale Gleichung für D_2 lautet: $D_2 = Q_1$.

Damit können wir die Schaltung angeben (Abbildung 3.51).

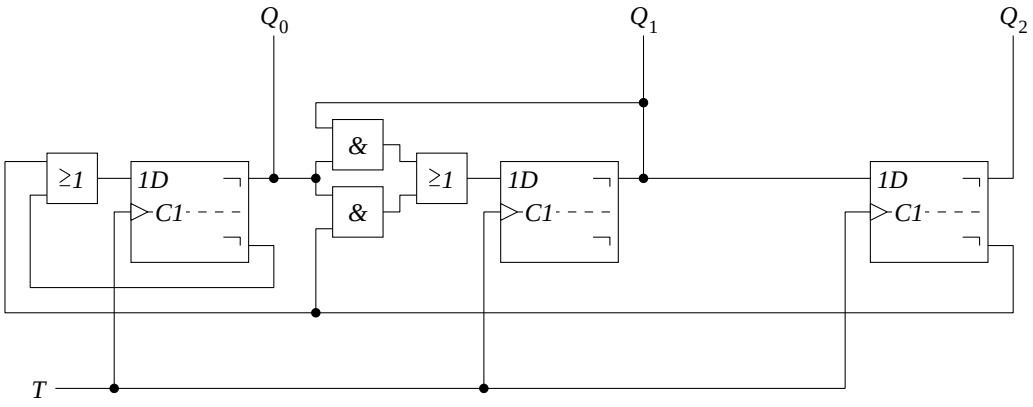


Abbildung 3.51: Schaltbild zur Lösung der Selbsttestaufgabe 3.8.

Selbsttestaufgabe 3.9 von Seite 135

Es gibt die Äquivalenzklassen (1,3,6) und (4,5,7). Die Zustände 3 und 6 können mit dem Zustand 1, die Zustände 5 und 7 mit dem Zustand 4 verschmolzen werden, d.h. es werden insgesamt nur noch 4 Zustände (1,2,4 und 8) benötigt.

Kurseinheit 4

Komplexe Schaltwerke, Grundlagen eines Computers

Kapitelinhalt

Zusammenfassung

In der letzten Kurseinheit haben wir Schaltwerke anhand von Zustandsgraphen bzw. -tabellen entworfen. Da bei der Mikroprogrammierung die Tabellen direkt in einen Speicher übertragen werden können, ist diese Methode wesentlich einfacher als der konventionelle Entwurf mit optimierten Schaltnetzen für die Funktionen f und g. Aber auch diese einfachere Entwurfsmethode kann nur angewandt werden, solange die Zahl der Zustände und Eingangsvariablen „klein“ ist.

Eine interessante Anwendung von Schaltwerken besteht darin, komplexere arithmetische Operationen auf der Basis von Schaltnetzen für einfacherere Operationen auszuführen und dadurch den Hardwareaufwand zu minimieren. So kann beispielsweise eine Multiplikation auf mehrere Additionen zurückgeführt werden. Schaltwerke für derartige Aufgabenstellungen können wegen der großen Zahl der Eingangsvariablen bzw. möglichen Zustände nicht mit den bisher behandelten Methoden entworfen werden. Bei arithmetischen Aufgabenstellungen ist außerdem die genaue Zahl der Zustände im Voraus nicht bekannt, weil die Anzahl der nötigen Berechnungsschritte von den Operanden abhängt.

Um bei solch komplexen Aufgabenstellungen einen übersichtlichen Schaltwerksentwurf zu erreichen, verwendet man ein *komplexes Schaltwerk*, das aus zwei kooperierenden Teilschaltwerken besteht. Ausgangspunkt für den Entwurf eines komplexen Schaltwerks ist ein *Hardware-Algorithmus* für die gewünschte Funktion. Die Umsetzung von Algorithmen in digitale Hardware erreicht man am einfachsten durch eine funktionale Aufspaltung in einen *steuernden* und einen *verarbeitenden* Teil. Ein komplexes Schaltwerk besteht daher aus einem *Steuer-* und einem *Operationswerk*.

Die Verwendung universeller Operationswerke führt uns zu den Grundlagen eines Computers, dessen wichtigste Komponente der Prozessor ist. Er besteht – wie ein komplexes Schaltwerk – aus zwei Teilschaltwerken: Leit- und Rechenwerk. Ein Prozessor stellt dem Anwender einen Satz nützlicher Maschinenbefehle zur Verfügung, mit deren Hilfe Programme zur Lösung verschiedenster Problemstellungen erstellt werden können. Diese Programme werden zusammen mit den benötigten Daten in einem Speicher abgelegt.

Lernziele

Die Lernziele dieser Kurseinheit sind:

- Verständnis von Aufbau und Funktion komplexer Schaltwerke.
- Fähigkeit, Hardware-Algorithmen für komplexe Problemstellungen zu formulieren und daraus komplexe Schaltwerke abzuleiten.
- Kenntnis der Grundlagen eines Computers und Fähigkeit, die Abläufe bei der Befehlsverarbeitung zu beschreiben.

4.1 Entwurf von Schaltwerken

In der letzten Kurseinheit haben wir Methoden zum Entwurf von Schaltwerken kennengelernt. Die wesentlichen Entwurfsschritte sollen hier kurz wiederholt werden.

Ausgehend von einer verbalen (informalen) Beschreibung, kann man das gewünschte Schaltverhalten mit einem Zustandsgraphen formal spezifizieren. Die Zahl der Zustände wird minimiert, indem man äquivalente Zustände zusammenfasst. Dadurch werden weniger Flipflops benötigt. Durch eine geeignete Zustands-Codierung kann der Aufwand zur Bestimmung der Folgezustände minimiert werden. Um die Schaltfunktionen für die Eingänge der D-Flipflops zu finden, überträgt man den Zustandsgraphen in eine *Zustandstabelle*. In dieser Tabelle stehen auf der linken Seite die Zustandsvariablen z_k^t und die Eingabevariablen x_m^t zum Zeitpunkt t. Auf der rechten Seite stehen die gewünschten Folgezustände z_k^{t+1} und die zugehörigen Ausgabevariablen y_n^t.

Die Entwurfsaufgabe besteht daher in der Erstellung der Zustandstabelle und der Minimierung der damit spezifizierten Schaltfunktionen. Im Gegensatz zu diesem konventionellen Entwurf kann bei Anwendung der Mikroprogrammierung der zweite Schritt entfallen. Beide Entwurfsmethoden sind aber nur anwendbar, solange die Zahl der Eingangsvariablen und möglichen inneren Zustände „klein“ bleibt.

Beispiel 4.1 *8-Bit-Dualzähler: Mit k Speichergliedern sind maximal 2^k Zustände codierbar. Um z.B. einen 8-Bit-Dualzähler nach der beschriebenen Methode zu entwerfen, müsste eine Zustandstabelle mit 256 Zeilen erstellt werden. Hiermit müssten acht Schaltfunktionen für die jeweiligen D-Flipflops ermittelt und minimiert werden. Obwohl es sich bei diesem Beispiel um eine relativ einfache Aufgabenstellung handelt und außer der Taktvariablen keine weiteren Eingangsvariablen vorliegen, ist ein Entwurf mit einer Zustandstabelle nicht durchführbar.*

Schaltwerke für komplizierte Aufgabenstellungen, d.h. Schaltwerke mit einer großen Zahl von Zuständen, wollen wir als *komplexe Schaltwerke* bezeichnen. Um bei komplexen Schaltwerken einen effektiven und übersichtlichen Entwurf zu erreichen, nimmt man eine Aufteilung in zwei kooperierende Teilschaltwerke vor: ein *Operationswerk* (data path) und ein *Steuerwerk* (control path). Die funktionale Aufspaltung in eine *verarbeitende* und eine *steuernde* Komponente vereinfacht den Entwurf erheblich, da beide Teilschaltwerke getrennt voneinander entwickelt und optimiert werden können.

komplexe Schaltwerke

Beispiele für die Anwendung komplexer Schaltwerke sind:

- Ampelsteuerung mit flexiblen Phasen,
- Umrechnung von Messwerten (z.B. Temperatur von Grad Celsius in Fahrenheit),
- Arithmetische Einheiten für Gleitkommazahlen (Multiplikation, Division).

Für die o.g. Aufgabenstellungen könnten wir sehr leicht eine Lösung in Form eines Algorithmus angeben und ein entsprechendes Programm schreiben. Aber auch wenn wir eine Hardware-Lösung für die Aufgabenstellung suchen, können wir einen Algorithmus als Ausgangspunkt für den Entwurf benutzen. Diesen *Hardware-Algorithmus* implementieren wir dann mit einem komplexen Schaltwerk.

4.2 Komplexe Schaltwerke

Abbildung 4.1 zeigt den Aufbau eines komplexen Schaltwerks. Wie bei einem gewöhnlichen Schaltwerk sind funktional zusammengehörende Schaltvariablen zu Vektoren zusammengefasst. Das Operationswerk bildet den verarbeitenden Teil, d.h. hier wird der Eingabevektor X schrittweise zum Ausgabevektor Y umgeformt. Der Eingabevektor, Zwischenergebnisse und der Ausgabevektor werden hierzu in Registern gespeichert und über Schaltnetze (arithmetisch oder logisch) miteinander verknüpft. Die Datenpfade zwischen den Registern und solchen *Operations*-Schaltnetzen können mit Hilfe von Multiplexern oder Bussen realisiert werden.

Steuervektor

Die in einem Taktzyklus auszuführenden Operationen und zu schaltenden Datenpfade werden durch die Belegung des *Steuervektors* bestimmt. Mit jedem Steuerwort wird ein Teilschritt des Hardware-Algorithmus abgearbeitet. Das Steuerwerk hat die Aufgabe, die richtige Abfolge von Steuerwörtern zu erzeugen. Dabei berücksichtigt es die jeweilige Belegung des *Statusvektors*, der besondere Ergebniszustände aus den vorhergehenden Taktzyklen anzeigt. Über eine Schaltvariable des Statusvektors kann z.B. gemeldet werden, dass ein Registerinhalt den Wert Null angenommen hat. Das Steuerwerk kann bestimmte Statusvariablen abfragen und anschließend auf deren momentane Belegung reagieren. So ist es beispielsweise möglich, eine Schleife im Steueralgorithmus zu verlassen, wenn das Register mit dem Schleifenzähler den Wert Null erreicht hat.

Statusvektor

Wir gehen im Folgenden davon aus, dass die beiden Teilschaltwerke synchron getaktet werden und die Rückkopplungsbedingungen stets erfüllt sind.

4.3 RTL-Notation

In diesem Abschnitt wollen wir die so genannte *RTL*-Notation (Register Transfer Level) zur kompakten Beschreibung von Hardwarekomponenten und deren Funktionen einführen. Wir bezeichnen Register durch Großbuchstaben. Sofern die Register eine besondere Funktion haben, wählen wir eine passende Abkürzung, z.B. AR für ein Adressregister. Allgemeine Register zur Speicherung von Daten können mit R bezeichnet und durchnummeriert werden.

Ein Register wird als Vektor definiert, indem man die „Randbits“ angibt. So bedeutet $R2(7:0)$, dass $R2$ ein 8-Bit-Register ist. Das rechte Bit hat die Wertigkeit 2^0 und wird daher auch als LSB (Least Significant Bit) bezeichnet. Das linke Bit hat die Wertigkeit 2^7 und wird als MSB (Most Significant Bit)

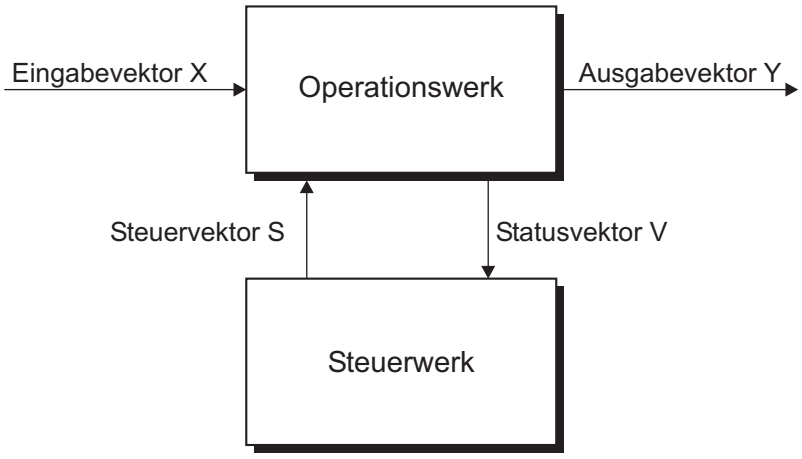


Abbildung 4.1: Aufbau eines komplexen Schaltwerks.

bezeichnet. Diese Anordnung der Bits heißt *big endian order*, weil man das höchstwertige Bit zuerst schreibt. Umgekehrt gibt es natürlich auch eine *little endian order*, die mit dem niederwertigen Bit beginnt. Häufig fasst man 8 Bit zu einem *Byte* zusammen und verwendet dieses anstatt einzelner Bits als Basisgröße (z.B. bei Speicherung oder Datenübertragung). Größere Wörter (z.B. 32 Bit) werden dann byteweise zerlegt bzw. zusammengesetzt. Beginnt man dabei z.B. mit dem höchstwertigen Byte $(D_{31}, \cdots, D_{24})$ so spricht man ebenfalls von einer Big- endian-Reihung.

Zuweisungen zu Registern werden durch den Ersetzungsoperator $\leftarrow$ beschrieben. So bedeutet $R_1 \leftarrow 0$, dass das Register R_1 zurückgesetzt wird, d.h. alle Bits werden Null. $R_1 \leftarrow R_1 + R_2$ bedeutet, dass der Inhalt von R_1 durch die Summe der in den Registern R_1 und R_2 gespeicherten Werte ersetzt wird.

Soll ein Registerinhalt **nicht** mit jedem Taktzyklus verändert werden, so kann man die Datenübernahme von einem Steuersignal abhängig machen. Dadurch erhalten wir eine bedingte Zuweisung in der Form:

$$\text{if } (S_1 = 1) \text{ then } (R_1 \leftarrow R_1 + R_2)$$

Schaltungstechnisch wirkt S_1 in diesem Fall als Steuervariable, die das Taktsignal modifiziert. Dies wird durch ein AND-Schaltglied wie in Abbildung 4.2 realisiert.

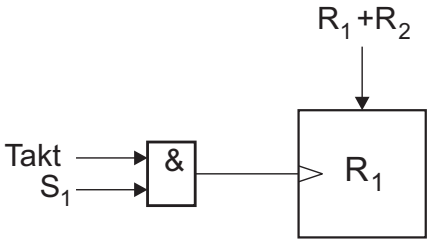


Abbildung 4.2: Bedingte Zuweisung durch Taktausblendung mit einem AND-Schaltglied.

Die o.g. if-Anweisung kann auch durch folgende verkürzte Schreibweise dargestellt werden:

$$S_1 : R_1 \leftarrow R_1 + R_2$$

Man beachte, dass wegen der Laufzeitverzögerung durch das AND-Glied ein Taktversatz (clock skew) zu den anderen Registern entsteht und somit die Wirk- und Kippintervalle verbreitert werden.

Sollen zwei oder mehr Registertransfers gleichzeitig stattfinden, so werden die Anweisungen in eine Zeile geschrieben und durch Kommata voneinander getrennt. Man beachte, dass die Reihenfolge der Anweisungen innerhalb einer Zeile keine Bedeutung hat, da alle gleichzeitig ausgeführt werden.

Zur Speicherung von Daten können neben einzelnen Registern auch Speicher eingesetzt werden. Diese werden durch den Großbuchstaben M charakterisiert und durch ein Adressregister angesprochen, das als Index in eckigen Klammern angegeben wird.

Beispiel 4.2 *Befehlsholephase eines Prozessors*

Wir werden später noch genauer untersuchen wie ein Prozessor Befehle verarbeitet. Ein wichtiger Teilschritt bei der Befehlsverarbeitung ist die Befehlsholephase, bei der ein Maschinenbefehl aus dem Hauptspeicher ins Leitwerk übertragen wird. Durch die Operation

$$IR \leftarrow M[PC], PC \leftarrow PC + 1$$

wird der durch den Befehlszähler (Programm Counter, PC) adressierte Befehl in das Befehlsregister (Instruction Register, IR) geladen. Gleichzeitig wird der Befehlszähler inkrementiert. Man beachte, dass am Ende eines Kippintervalls der noch unveränderte Befehlszähler zum Adressieren des Speichers benutzt wird und dass der geholte Befehl sowie der neue Befehlszählerstand erst mit dem Wirkintervall des nachfolgenden Takts übernommen werden. Das Inkrementieren des Befehlszählers kann auch durch die Schreibweise PC++ abgekürzt werden.

Die Register oder Speicherplätze auf der rechten Seite des Ersetzungssymbols $\leftarrow$ können durch Operationen miteinander verknüpft werden, die durch Boole'sche Schaltfunktionen beschrieben werden. Wir unterscheiden im Folgenden drei Kategorien und geben Beispiele für solche *Mikrooperationen* an.

1. Logische Mikrooperationen, bitweise ausgeführt
   - NOT $\bar{}$
   - AND $\wedge$
   - OR $\vee$
   - EXOR $\oplus$
2. Arithmetische Mikrooperationen
   - Addition +
   - Um 1 inkrementieren ++

- Subtraktion –
  (kann auf die Addition des Zweierkomplements zurückgeführt werden)
- Um 1 dekementieren – –
- Einerkomplement ¬
  (ist identisch mit der Negation)

3. Verschiebung und Konkatenation

   - Schiebe um n Bit nach links $\ll n$ bzw. nach rechts $\gg n$
   - Rotiere um n Bit nach links $\hookleftarrow n$ bzw. nach rechts $\hookrightarrow n$
   - Verbinde zwei Vektoren zu einem größeren Vektor $\|$

Mikrooperationen können durch Schaltnetze (z.B. Addition) oder einfach nur durch die An- bzw. Umordnung der Datenleitungen (z.B. Konkatenation, Schiebemultiplexer) implementiert werden. Durch Verkettung dieser elementaren Mikrooperationen in einem Hardware-Algorithmus können mit einem Operationswerk komplexere *Makrooperationen* gebildet werden. So kann beispielsweise ein einfaches Operationswerk mit einer 8 Bit breiten ALU (Arithmetic Logic Unit), die nur ganze Zahlen addieren bzw. subtrahieren kann, auch 32- oder 64-Bit-Gleitkommaoperationen ausführen. Man benötigt dazu einen Hardware-Algorithmus, der die Gleitkommaoperationen so zerlegt, dass sie durch eine entsprechende Zahl von 8-Bit-Operationen umgesetzt werden können. Die Anzahl und Abfolge dieser Mikrooperationen ist von den Daten abhängig und wird durch die Zwischenergebnisse (Statusbits) gesteuert. Die Anzahl der Mikrooperationen kann verringert werden, indem man die Wortbreite oder die Komplexität der verfügbaren Funktionsschaltnetze erhöht.

ALU

Nachdem wir nun die grundlegenden Mikrooperationen kennengelernt haben, werden wir im nächsten Abschnitt sehen, wie diese in einen Hardware-Algorithmus integriert werden.

4.4 ASM-Diagramme

Als Hilfsmittel zur Beschreibung von Hardware-Algorithmen für ein komplexes Schaltwerk verwendet man das so genannte **Algorithmic State Machine Chart** oder **ASM-Diagramm**. Ein ASM-Diagramm ähnelt einem Flußdiagramm, enthält aber zusätzliche Informationen über die zeitlichen Abläufe der benutzten Mikrooperationen. Es kann drei Arten von *Boxen* enthalten:

1. Zustandsboxen,
2. Entscheidungsboxen und
3. bedingte Ausgangsboxen.

Bei Moore-Schaltwerken gibt es keine bedingten Ausgangsboxen, da sie zustandsorientiert arbeiten. Dafür enthalten Moore-Schaltwerke in der Regel aber mehr Zustandsboxen als Mealy-Schaltwerke. Aus den o.g. Elementen werden wiederum *ASM-Blöcke* gebildet, die jeweils einen Zustand und dessen komplette Verzweigungsstruktur beinhalten.

4.4.1 Zustandsboxen

Die Zustandsbox wird durch einen symbolischen Zustandsnamen und einen Zustandscode bezeichnet (Abbildung 4.3). Innerhalb einer Zustandsbox können einem oder mehreren Registern Werte zugewiesen werden. Die Zuweisungen werden in der oben eingeführten RTL-Notation angegeben. In Zustandsboxen wird die Aktion festgelegt, die beim Erreichen dieses Zustands im Operationswerk ausgeführt werden soll.

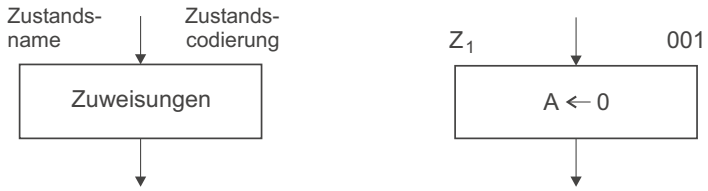


Abbildung 4.3: Zustandsbox, links: allgemeine Form, rechts: Beispiel.

In jedem ASM-Diagramm gibt es einen Anfangszustand Z_0, der nach dem Einschalten der Betriebsspannung eingenommen wird. Die dazu erforderlichen Datenpfade werden durch entsprechende Belegungen der Steuervariablen bestimmt, die wiederum durch den Anfangszustand des Steuerwerks festgelegt sind.

4.4.2 Entscheidungsboxen

Entscheidungsboxen testen Registerinhalte oder Eingänge auf besondere Bedingungen (Abbildung 4.4). Hierzu werden Status-Informationen durch einen Vergleicher (Komparator) abgefragt oder die Belegung von Eingängen mit bestimmten Werten wird getestet. Die abgefragten Bedingungen steuern den Kontrollfluß und legen damit die Ausgangspfade aus einem ASM-Block fest. Durch Entscheidungsboxen werden die Folgezustände festgelegt, die das Steuerwerk als nächstes einnehmen soll. Eine Entscheidungsbox hat nur zwei Ausgänge, da die abgefragte Bedingung nur wahr (1) oder falsch (0) sein kann.

Abbildung 4.4: Entscheidungsbox, links: allgemeine Form, rechts: Beispiel.

4.4.3 Bedingte Ausgangsboxen

Diese Komponente gibt es nur bei ASM-Diagrammen für Mealy-Schaltwerke. Wie bei einer Zustandsbox können in der bedingten Ausgangsbox Variablen

oder Ausgängen Werte zugewiesen werden[1]. Es werden jedoch nur diejenigen Ausgangsboxen ausgeführt, die auf dem genommenen Pfad liegen. Die Zuweisungen der aktivierten bedingten Ausgangsboxen werden zeitgleich zu den Zuweisungen in der vorgelagerten Zustandsbox ausgeführt. Zur Unterscheidung von einer Zustandsbox hat die bedingte Ausgangsbox abgerundete Ecken (Abbildung 4.5). Außerdem wird weder ein Zustandsname noch ein Zustandscode zugeordnet.

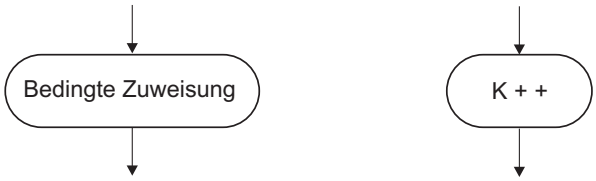


Abbildung 4.5: Bedingte Ausgangsbox, links: allgemeine Form, rechts: Beispiel.

Selbsttestaufgabe 4.1 (Erforderliches Steuerwerk)
Begründen Sie, dass bei Vorliegen einer bedingten Ausgangsbox in einem ASM-Diagramm das Steuerwerk des betreffenden komplexen Schaltwerks als Mealy-Automat ausgeführt werden muss.

Lösung auf Seite 205

4.4.4 ASM-Block

Ein ASM-Block enthält genau eine Zustandsbox, die als Eingang in den Block dient. Ein Netzwerk aus hinter- bzw. nebeneinandergeschalteten Entscheidungsboxen, das im Falle von Mealy-Schaltwerken auch bedingte Ausgangsboxen enthalten kann, führt zu anderen Zuständen im ASM-Diagramm. Die ASM-Blöcke können dadurch gekennzeichnet werden, dass man sie durch gestrichelte Linien einrahmt oder grau hinterlegt. Das Netzwerk der ASM-Blöcke bildet schließlich das ASM-Diagramm.

Beim Arbeiten mit ASM-Diagrammen müssen folgende Regeln beachtet werden:

1. *Die Entscheidungsboxen müssen so miteinander verbunden sein, dass jeder durch das Netzwerk der Entscheidungsboxen genommene Pfad (abhängig von der jeweiligen Variablenbelegung) zu genau einem Zustand führt.*

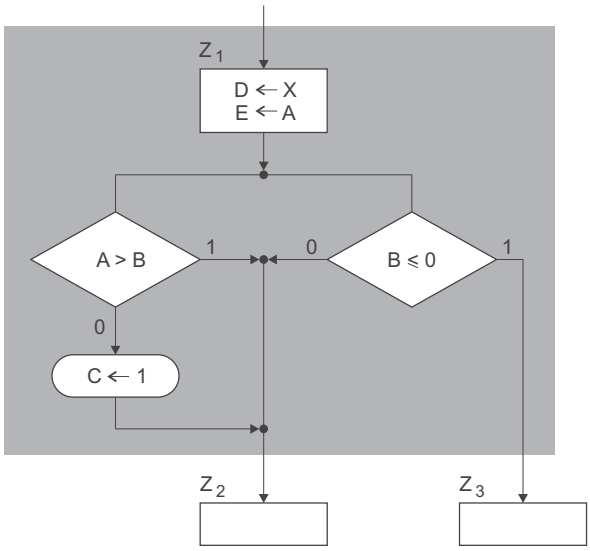
2. *Variablen in der Zustandsbox oder in bedingten Ausführungsboxen dürfen innerhalb eines ASM-Blocks nur einmal auf der linken Seite stehen, d.h. sie dürfen nur einmal pro Taktzyklus beschrieben werden.*

3. *Die Wertzuweisungen an die Variablen erfolgen erst am Ende eines Taktzyklus, d.h. innerhalb des Zustandes werden auf der rechten Seite von*

[1]Man könnte sie daher auch als bedingte Zustandsboxen bezeichnen.

Zuweisungen und in Entscheidungsboxen die Werte benutzt, die die Variablen am Anfang des Zustands haben. Die im Zustand neu zugewiesenen Werte sind also erst kurz nach Ende des Taktzyklus (am Anfang des Folgezustands) gültig.

Selbsttestaufgabe 4.2 (ASM-Diagramm) *Entscheiden Sie, ob der nachfolgende ASM-Block die in 1 angegebene Regel erfüllt.*



Lösung auf Seite 205

Im Gegensatz zu einem normalen Flußdiagramm ordnet das ASM-Diagramm den einzelnen Anweisungen und Verzweigungen Zustände zu und definiert so, welche Speicher, Funktionsschaltnetze und Datenwegschaltungen im Operationswerk benötigt werden. Da gleichzeitig auch der Kontrollfluß visualisiert wird, eignet sich das ASM-Diagramm zur vollständigen Spezifikation eines komplexen Schaltwerks. Anhand der Darstellung mit einem ASM-Diagramm können sowohl das Operations- als auch das Steuerwerk synthetisiert werden.

Bevor wir den Entwurf mit ASM-Diagrammen an einem Beispiel demonstrieren, wollen wir einige allgemeine Konstruktionsregeln für Operations- und Steuerwerke vorstellen.

4.5 Konstruktionsregeln für Operationswerke

Da eine gegebene Aufgabenstellung durch viele verschiedene Algorithmen lösbar ist, gibt es für den Entwurf eines komplexen Schaltwerks keine eindeutige Lösung. Die *Kunst* des Entwurfs besteht darin, einen guten Kompromiß zwischen Hardwareaufwand und Zeitbedarf zu finden.

Ist die Architektur des Operationswerks vorgegeben, so muss der Steueralgorithmus daran angepaßt werden. Dies ist z.B. bei einem Rechenwerk eines Computers der Fall. Das Rechenwerk ist ein universelles Operationswerk, das zur Lösung beliebiger (berechenbarer) Problemstellungen benutzt werden kann.

Wenn dagegen die Architektur des Operationswerks frei gewählt werden darf, hat der Entwickler den größten Spielraum — aber auch den größten Entwurfsaufwand. Für diesen Fall sucht man ein an die jeweilige Problemstellung angepasstes Operationswerk. Wir werden im Folgenden Konstruktionsregeln für den Entwurf solcher *anwendungsspezifischer* Operationswerke angeben.

Ausgangspunkt ist stets ein Algorithmus, der angibt, wie der Eingabevektor zum Ausgabevektor verarbeitet werden soll. Wie wir oben gesehen haben, kann zur Beschreibung von Hardware-Algorithmen ein ASM-Diagramm benutzt werden. Es enthält Anweisungen mit Variablen, Konstanten, Operatoren, Zuweisungen und bedingte Verzweigungen. Hiermit kann dann die Architektur des Operationswerks wie folgt abgeleitet werden:

1. Für jede Variable, die auf der linken Seite einer Zuweisung steht, ist ein Register erforderlich. Um die Zahl der Register zu minimieren, können sich zwei oder mehrere Variablen ein Register teilen. Voraussetzung für eine solche Mehrfachnutzung ist aber, dass diese Variablen nur eine begrenzte „Lebensdauer“ haben und dass sie nicht gleichzeitig benutzt werden.

2. Wenn einer Variablen mehr als ein Ausdruck zugewiesen wird, muss vor die Eingänge des Registers ein Multiplexer (Auswahlnetz) geschaltet werden. Um die Zahl der Verbindungsleitungen (Chipfläche) zu reduzieren, können die Register auch in einem *Registerblock* zusammengefasst werden. Der Zugriff erfolgt in diesem Fall *zeitversetzt* über einen Bus. Soll auf zwei oder mehrere Register gleichzeitig zugegriffen werden, so muss der Registerblock über eine entsprechende Anzahl von Busschnittstellen verfügen, die auch als *Ports* bezeichnet werden. Man unterscheidet Schreib- und Leseports. Ein bestimmtes Register kann zu einem bestimmtem Zeitpunkt natürlich nur von einem Schreibport beschrieben werden. Es kann aber gleichzeitig an zwei oder mehreren Leseports ausgelesen werden.

   Registerblock

   Ports

3. Konstanten können fest verdrahtet sein oder sie werden durch einen Teil des Steuervektors definiert.

4. Die Berechnung der Ausdrücke auf den rechten Seiten von Wertzuweisungen erfolgt mit *Funktionsschaltnetzen*, welche die erforderlichen Mikrooperationen mit Hilfe Boole'scher Funktionen realisieren.

5. Wertzuweisungen an unterschiedliche Variablen können parallel (gleichzeitig) ausgeführt werden, wenn sie im Hardware-Algorithmus unmittelbar aufeinander folgen und wenn *keine Datenabhängigkeiten* zwischen den Anweisungen bestehen. Eine (echte) Datenabhängigkeit liegt dann vor, wenn die vorangehende Anweisung ein Register verändert, das in der nachfolgenden Anweisung als Operand benötigt wird.

   Datenabhängigkeiten

6. Zur Abfrage der Bedingungen für Verzweigungen müssen entsprechende Statusvariablen gebildet werden.

Die gleichzeitige Ausführung mehrerer Anweisungen verringert die Zahl der Taktzyklen und verkürzt damit die Verarbeitungszeit. Andererseits erhöht dies

aber den Hardwareaufwand, wenn in den gleichzeitig abzuarbeitenden Wertzuweisungen dieselben Mikrooperationen auftreten. Die Funktionsschaltnetze für die parallel ausgeführten Operationen müssen dann mehrfach vorhanden sein.

In Abschnitt 4.7 wird die Anwendung der oben angegebenen Konstruktionsregeln demonstriert.

Selbsttestaufgabe 4.3 (Einfaches Operationswerk)
Skizzieren Sie ein Operationswerk mit zwei Registern R_1 und R_2. Durch eine Steuervariable S soll es möglich sein, eine der beiden folgenden Mikrooperationen auszuwählen.

$S = 0$ *Tausche die Registerinhalte von R_1 und R_2.*

$S = 1$ *Addiere die Registerinhalte von R_1 und R_2 und schreibe die Summe in das Register R_1. Das Register R_2 soll dabei unverändert bleiben.*

Lösung auf Seite 205

4.6 Entwurf des Steuerwerks

Die Struktur eines Steuerwerks ist unabhängig von einem bestimmten Steueralgorithmus. Für den systematischen Entwurf haben wir drei Möglichkeiten kennengelernt:

1. Entwurf mit Hilfe von Zustandsgraph/-tabelle, Zustandsminimierung und optimierter Zustandscodierung,
2. wie 1., allerdings mit einem Flipflop pro Zustand (Hot-one-coding),
3. Entwurf als Mikroprogrammsteuerwerk.

Die beiden erstgenannten Methoden sind nur dann anwendbar, wenn die Zahl der Zustände nicht zu groß ist (z.B. bis 32 Zustände). Für umfangreichere Steueralgorithmen kommt praktisch nur die Methode der *Mikroprogrammierung* in Frage.

Ein verzweigungsfreier oder linearer Steueralgorithmus kann besonders leicht implementiert werden. Wir benötigen nur einen Zähler und einen *Steuerwort-Speicher* (Control Memory), der die einzelnen Steuerwörter aufnimmt und von den Zählerausgängen adressiert wird. Sobald der Eingabevektor anliegt, wird der Zähler zurückgesetzt und gestartet. Am Ausgang des Steuerwort-Speichers werden nun die Steuerwörter nacheinander ausgegeben und vom Operationswerk verarbeitet. Eine Schaltvariable des Steuervektors zeigt das Ende des Steueralgorithmus an und stoppt den Zähler. Danach kann ein neuer Zyklus beginnen.

Um *Verzweigungen* innerhalb des Steueralgorithmus zu ermöglichen, wird ein ladbarer Zähler verwendet (Abbildung 4.6). Der Einfachheit halber wollen wir annehmen, dass nur Verzweigungen auf die Bedingung „zero“, d.h. ein bestimmtes Register R des Operationswerks hat den Wert 0 angenommen, erlaubt sind. Diese Bedingung ist erfüllt, wenn alle Bits des Registers R den

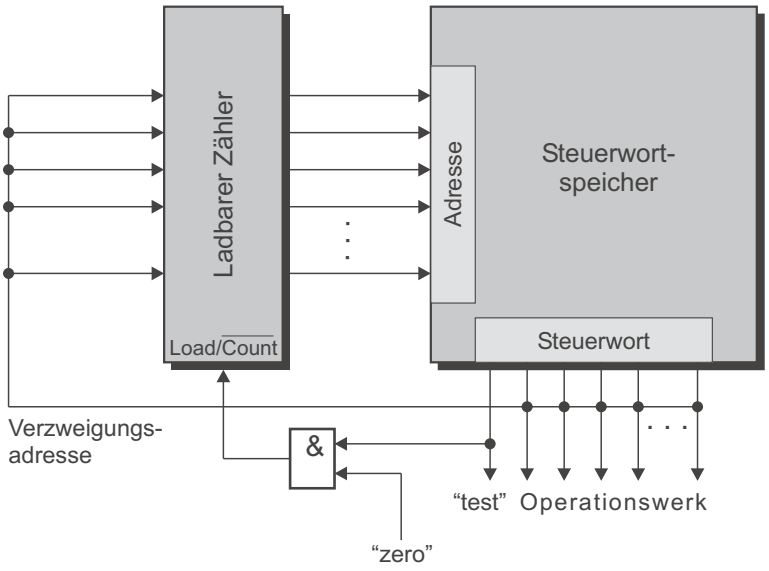


Abbildung 4.6: Prinzip eines mikroprogrammierbaren Steuerwerks, das bedingte Verzweigungen ausführen kann.

Wert 0 haben. Eine entsprechende Statusvariable „zero“ kann beispielsweise mit einem NOR-Schaltglied erzeugt werden, das alle Bits als Eingabe erhält. Durch eine Schaltvariable des Steuervektors wird diese Abfrage aktiviert. Falls das „zero“-Bit gesetzt ist, wird der (Mikroprogramm-)Zähler mit einer Verzweigungsadresse aus dem Steuerwort-Speicher geladen. Die Abfrage der Steuervariablen „zero“ wird durch eine 1 an der Steuervariablen „test“ aktiviert. Die AND-Verknüpfung liefert dann bei gesetztem „zero“-Flag den Wert 1 und schaltet über den $Load/\overline{Count}-$Eingang den Zähler vom Zähl- auf Ladebetrieb. Um Speicherplatz zu sparen, wird ein Teil des Steuerworts als Verzweigungsadresse interpretiert. Das Steuerwort darf deshalb während der Abfrage einer Verzweigungsbedingung nicht im Operationswerk wirksam werden. Mit Hilfe des Abfragesignals „test“ kann z.B. der Takt des Operationswerks für einen Taktzyklus ausgeblendet werden. Dadurch wird jeweils ein Wirk- und ein Kippintervall unterdrückt. Die durch die Verzweigungsadresse geschalteten Datenpfade und Operationen werden unwirksam, da die D-Flipflops die Ergebnisse nicht übernehmen.

Die Synchronisierung des Steuerwerks in Abbildung 4.6 erfolgt durch Rücksetzen des Zählers beim Start und durch ein Stop-Bit im Steuerwort. Beide Einrichtungen wurden der Übersicht halber nicht eingezeichnet.

Im Folgenden wollen wir an einem einfachen Beispiel verschiedene Entwürfe von komplexen Schaltwerken vorstellen.

4.7 Beispiel: Einsen-Zähler

Ein komplexes Schaltwerk soll die Anzahl der Einsen in einem n Bit langen Wort X bestimmen. Die Berechnung soll beginnen, sobald ein weiteres Eingangssignal *Start* den Wert 1 annimmt. Das Ergebnis wird auf den Ausgangsvektor Y gelegt

und eine 1 auf einem weiteren Ausgangssignal zeigt an, dass das Ergebnis am Ausgang anliegt.

Das hier vorliegende Problem ist für großes n sehr schwer zu lösen. Wir könnten mit einem Volladdierer jeweils 3-Bit-Blöcke auswerten und daraus eine 2-Bit-Dualzahl für die Anzahl der Bits in diesem 3-Bit-Block gewinnen. Bei $n = 12$ müssten wir dann aber vier „Wörter“ zu je 2 Bit in einem übergeordneten Addierschaltnetz aufsummieren.

Zur Lösung dieses Problems bietet sich ein ladbares Schieberegister A zur Aufnahme des n-Bit-Worts an. Die Anzahl der Einsen wird in einem Zähler K gespeichert, der zu Anfang zurückgesetzt wird. Der Zähler wird inkrementiert, wenn das niederwertige (rechte) Bit A_0 des Schieberegisters den Wert 1 hat. Dann wird das Register A um ein Bit nach rechts geschoben, wobei eine 0 von links nachgeschoben wird. Wenn alle Bits des Schieberegisters A den Wert 0 haben, kann K als Ergebnis an Y ausgegeben werden. Damit wir das Anliegen eines gültigen Ergebnisses erkennen, wird gleichzeitig der Ausgang $Fertig$ auf 1 gesetzt.

Selbsttestaufgabe 4.4 (12-Bit-Einsen-Zähler) *Entwerfen Sie ein Schaltnetz, um die Zahl der Einsen in einem 6 Bit langen Wort zu bestimmen. Erweitern Sie die Wortbreite mit zwei derartigen 6-Bit-Modulen auf 12 Bit.*

Lösung auf Seite 205

4.7.1 Lösung mit komplexem Moore-Schaltwerk

In Abbildung 4.7 ist das ASM-Diagramm für die zustandsorientierte Lösung dargestellt. Solange das *Start*-Signal den Wert 0 führt, bleibt das Schaltwerk im Zustand z_0. $Fertig = 0$ zeigt an, dass an Y noch kein gültiges Ergebnis anliegt. Wenn *Start* den Wert 1 annimmt, wird in den Zustand z_1 verzweigt. Dort wird der Eingangsvektor X in das Schieberegister A eingelesen. Gleichzeitig wird das Zählregister K zurückgesetzt. Im Zustand z_2 wird das niederwertige Bit A_0 des Schieberegisters abgefragt. Je nach Belegung wird K entweder inkrementiert oder es erfolgt ein direkter Zustandswechsel nach z_4. Dort wird A nach rechts geschoben und gleichzeitig geprüft, ob A noch Einsen enthält. Diese Statusinformation kann mit einem n-fach OR-Schaltglied erzeugt werden. Falls A noch mindestens eine 1 enthält, verzweigt das Schaltwerk in den Zustand z_2. Sonst wird für einen Taktzyklus die Zahl der Einsen auf Y ausgegeben. Dann springt das Schaltwerk in den Anfangszustand und wartet auf ein neues Startereignis.

4.7.2 Lösung mit komplexem Mealy-Schaltwerk

In Abbildung 4.8 ist die entsprechende übergangsorientierte Lösung dargestellt. Im Gegensatz zur Moore-Variante werden hier statt sechs nur vier Zustände benötigt. Zwei Zustände können durch die Verwendung bedingter Ausgangsboxen eingespart werden. Dies bedeutet, dass zur Zustandscodierung nur zwei statt drei Flipflops benötigt werden. Falls alle Bits des Eingangsvektors X mit dem Wert 1 belegt sind, erfolgt die Auswertung eines Bits bei der Mealy-Variante in

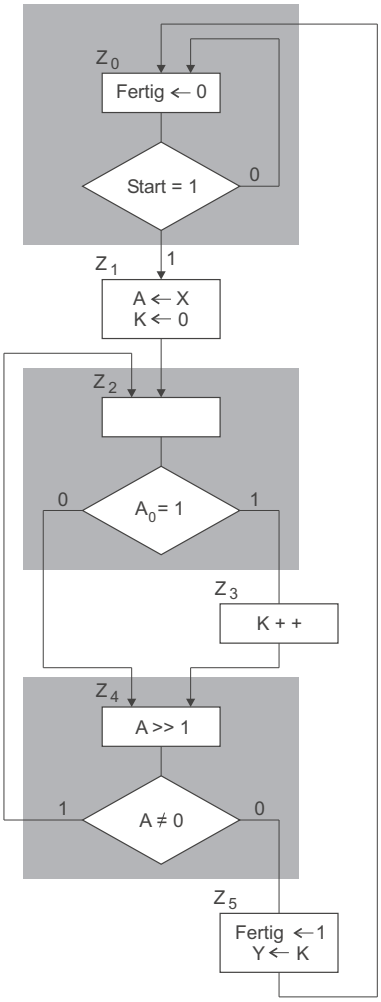


Abbildung 4.7: ASM-Diagramm für ein komplexes Moore-Schaltwerk.

einem statt in drei Taktschritten. Sie benötigt in diesem Fall auch nur ein Drittel der Zeit. Man beachte, dass die Bedingungen $A_0 = 1$ und $A \neq 0$ gleichzeitig ausgewertet werden, da sie dem gleichen ASM-Block zugeordnet sind.

4.7.3 Aufbau des Operationswerks

Bevor wir verschiedene Steuerwerkslösungen betrachten, entwerfen wir das Operationswerk (Abbildung 4.9). Das Operationswerk ist für beide Schaltwerks-Varianten identisch. Wir benötigen ein ladbares Schieberegister A mit einer Wortbreite von n Bit und ein Zählerregister K mit einer Wortbreite von $\lceil \log_2(n+1) \rceil$ Bit[2].

Zur Steuerung des Schieberegisters werden zwei Steuerleitungen benötigt: *Schieben* und *Aktiv* (vgl. Selbsttestaufgabe 4.5). Wenn *Schieben* $= 0$ und *Aktiv* $= 1$ ist, wird es mit einem parallel anliegenden Eingangsvektor geladen.

[2]Der $\lceil\ \rceil$-Operator rundet auf die nächste ganze Zahl auf.

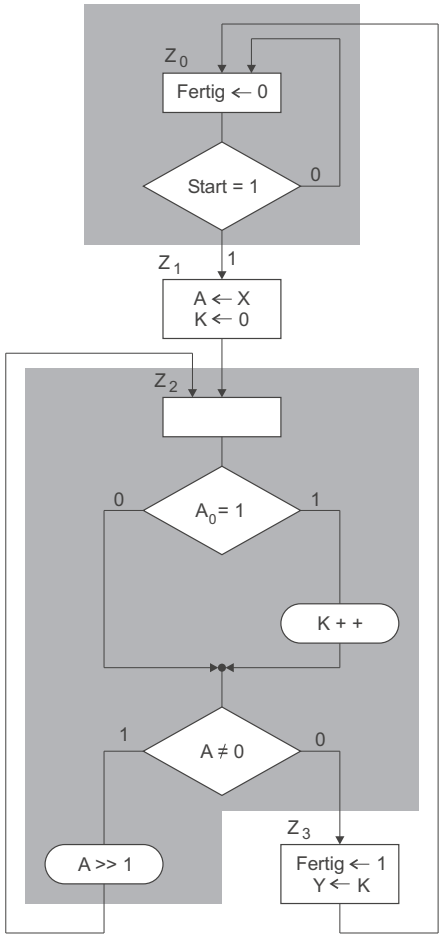


Abbildung 4.8: ASM-Diagramm für ein komplexes Mealy-Schaltwerk.

Bei der Belegung $Schieben = 1$ und $Aktiv = 1$ wird der Inhalt von A nach rechts geschoben.

Während A_0 direkt als Statusbit an das Steuerwerk weitergeleitet werden kann, muss die Statusinformation $A \neq 0$ aus dem Registerausgang von A durch ein n-fach OR-Schaltglied gebildet werden. Das Zählregister K kann mit $Reset = 1$ und $Inkrement = 0$ auf Null gesetzt werden. Bei Anliegen der Kombination $Reset = 0$ und $Inkrement = 1$ wird der Registerinhalt von K um 1 erhöht.

Zur Ausgabe des Ergebnisses auf Y wird ein *TriState*-Bustreiber vorgesehen, der über das $Fertig$-Signal gesteuert wird. Ein TriState-Bustreiber verstärkt im aktivierten Zustand ($Fertig = 1$) den an seinem Eingang anliegenden Bitvektor (ohne ihn jedoch zu verändern) und legt ihn auf einen Datenbus. Ein Datenbus kann von zwei oder mehreren Quellen zeitversetzt genutzt werden. Dazu darf zu einem bestimmten Zeitpunkt nur einer der Bustreiber aktiviert sein. Ein deaktivierter Bustreiber (hier: $Fertig = 0$) trennt die Quelle vollständig vom Bus ab, so dass eine andere Quelle Daten auf den Bus legen kann.

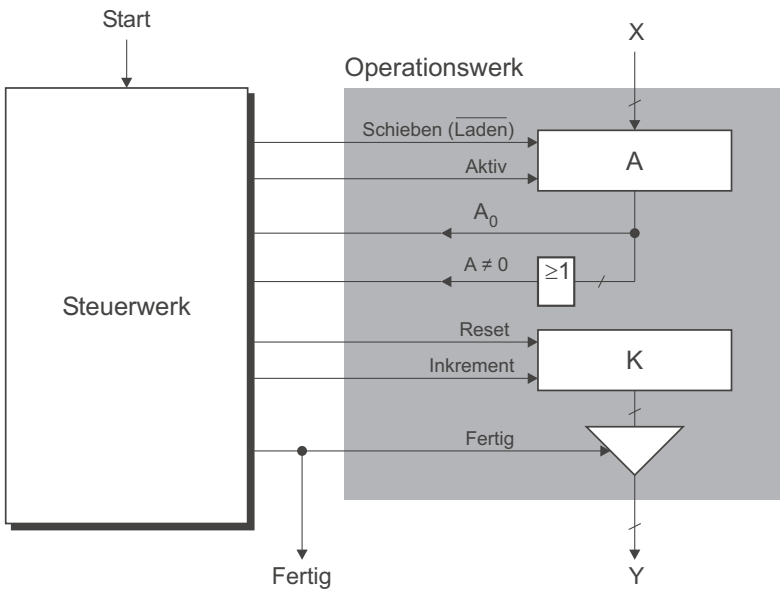


Abbildung 4.9: Aufbau des Operationswerks für den Einsen-Zähler.

4.7.4 Moore-Steuerwerk als konventionelles Schaltwerk

Aus dem ASM-Diagramm in Abbildung 4.7 sehen wir, dass sechs Zustände benötigt werden. Zunächst müssen wir entsprechende Zustandscodierungen festlegen. Wir wollen hierzu die Dualzahlen in aufsteigender Reihenfolge verwenden:

z^t	z_0	z_1	z_2	z_3	z_4	z_5
$Q_2Q_1Q_0$	000	001	010	011	100	101

Da die Codes 110 und 111 nicht vorkommen, können wir aus der obigen Tabelle diejenigen Codierungen, die zu diesen beiden Codes einen Hamming-Abstand[3] von 1 haben, weiter vereinfachen. Dies betrifft die Zustände z_2 bis z_5. Wir erhalten folgende Zustandscodierung, die gegenüber dem ersten Ansatz durch ein einfacheres Übergangsschaltnetz g implementiert werden kann.

z^t	z_0	z_1	z_2	z_3	z_4	z_5
$Q_2Q_1Q_0$	000	001	$\times$10	$\times$11	1$\times$0	1$\times$1

Das $\times$ bedeutet, dass das betreffende Bit i beim Berechnen des Zustands als Nachfolgezustand (D_i) den Wert 0 erhält, womit man Eindeutigkeit erzielt, dass man es aber bei der Abfrage des gegenwärtigen Zustands (Q_i) nicht berücksichtigen muss, was die Gleichungen vereinfacht.

Wir gehen davon aus, dass das Steuerwerksregister aus D-Flipflops aufgebaut ist. Die Folgezustände ergeben sich durch die Belegungen an den Eingängen der D-Flipflops:

$$z^{t+1} = D_2D_1D_0(z^t)$$

[3]Der Hamming-Abstand gibt an, um wie viele Bit sich zwei Codes voneinander unterscheiden.

Zusammen mit Abbildung 4.7 können wir nun die Schaltfunktionen für das Übergangsschaltnetz des Steuerwerks angeben.

D_0 muss den Wert 1 annehmen, wenn die Folgezustände z_1, z_3 oder z_5 sind. Der Zustand z_1 wird erreicht, wenn sich das Schaltwerk im Zustand z_0 befindet und $Start = 1$ wird. Der Zustand z_3 wird erreicht, wenn sich das Schaltwerk im Zustand z_2 befindet und $A_0 = 1$ wird. Schließlich wird der Zustand z_5 erreicht, wenn im Zustand z_4 $\overline{A \neq 0}$ gilt. Daraus erhalten wir folgende Schaltfunktion:

$$\begin{aligned} D_0 &= z_0 Start \vee z_2 A_0 \vee z_4 \overline{(A \neq 0)} \\ &= \overline{Q_2}\,\overline{Q_1}\,\overline{Q_0}\,Start \vee Q_1\,\overline{Q_0}\,A_0 \vee Q_2\,\overline{Q_0}\,\overline{(A \neq 0)} \end{aligned}$$

Aus der Zustandstabelle der vorigen Seite sehen wir, dass der Zustand z_2 durch die Kombination $Q_0 = 0$ und $Q_1 = 1$ eindeutig bestimmt ist und somit in der Gleichung für D_0 nur als $Q_1\overline{Q_0}$ auftaucht. Die Belegung von Q_2 spielt für die Definition des Zustands z_2 keine Rolle. Ähnlich verfährt man im Folgenden mit den Zuständen z_3 bis z_5.

D_1 muss nur für die Folgezustände z_2 und z_3 den Wert 1 annehmen, da für z_4 und z_5 das $\times$ mit 0 gleichgesetzt wird.

Aus Abbildung 4.7 sieht man, dass z_2 sich unbedingt aus z_1 oder aus z_4 und gleichzeitig $A \neq 0$ ergibt. Der Zustand z_3 wird erreicht, wenn in z_2 $A_0 = 1$ ist. Damit erhalten wir folgende Schaltfunktion:

$$\begin{aligned} D_1 &= z_1 \vee z_4(A \neq 0) \vee z_2 A_0 \\ &= \overline{Q_2}\,\overline{Q_1}\,Q_0 \vee Q_2\,\overline{Q_0}\,(A \neq 0) \vee Q_1\,\overline{Q_0}\,A_0 \end{aligned}$$

D_2 muss nur für die Folgezustände z_4 und z_5 den Wert 1 annehmen, da $\times$ für die Zustände z_2 und z_3 auf 0 gesetzt wird. Der Zustand z_4 ergibt sich, wenn in z_2 $A_0 = 0$ ist, oder unbedingt aus z_3. Der Zustand z_5 wird schließlich erreicht, wenn in z_4 $(A \neq 0) = 0$ ist. Damit folgt:

$$\begin{aligned} D_2 &= z_2\,\overline{A_0} \vee z_3 \vee z_4 \overline{(A \neq 0)} \\ &= Q_1\,\overline{Q_0}\,\overline{A_0} \vee Q_1\,Q_0 \vee Q_2\,\overline{Q_0}\,\overline{(A \neq 0)} \end{aligned}$$

Die o.g. Schaltfunktionen des Übergangsschaltnetzes definieren die Abfolge der Zustände in Abhängigkeit der vom Operationswerk gelieferten Statusbits. Um daraus die Steuersignale für das Operationswerk zu gewinnen, benötigen wir noch ein Ausgangsschaltnetz f. Dieses Schaltnetz wird durch folgende Schaltfunktionen bestimmt:

$$\begin{aligned} Schieben &= z_4 = Q_2\,\overline{Q_0} \\ Aktiv &= z_1 \vee z_4 = \overline{Q_2}\,\overline{Q_1}\,Q_0 \vee Q_2\,\overline{Q_0} \\ Reset &= z_1 = \overline{Q_2}\,\overline{Q_1}\,Q_0 \\ Inkrement &= z_3 = Q_1\,Q_0 \\ Fertig &= z_5 = Q_2\,Q_0 \end{aligned}$$

Selbsttestaufgabe 4.5 (Steuerung des Schieberegisters) *Warum ist es nicht möglich, das ladbare Schieberegister A mit einer einzigen Steuerleitung anzusteuern, d.h. weshalb wird die zweite Steuerleitung Aktiv benötigt?*

Lösung auf Seite 206

4.7.5 Moore-Steuerwerk mit Hot-one-Codierung

Die Zuordnung von bestimmten Zustandscodes kann entfallen, wenn man für jeden Zustand ein einzelnes D-Flipflop vorsieht. Die Schaltfunktionen für die Eingänge dieser Flipflops können *direkt* aus dem ASM-Diagramm abgelesen werden:

$$\begin{aligned}
D_0 &= Q_0\,\overline{Start} \vee Q_5 \vee \overline{Q_5}\,\overline{Q_4}\,\overline{Q_3}\,\overline{Q_2}\,\overline{Q_1}\,\overline{Q_0} \\
D_1 &= Q_0\,Start \\
D_2 &= Q_1 \vee Q_4\,(A \neq 0) \\
D_3 &= Q_2\,A_0 \\
D_4 &= Q_2\,\overline{A_0} \vee Q_3 \\
D_5 &= Q_4\,\overline{(A \neq 0)}
\end{aligned}$$

Damit das Steuerwerk nach Einschalten der Betriebsspannung sicher in den Zustand 0 gelangt, d.h. Q_0 gesetzt wird, muss in der Schaltfunktion für D_0 der Term $\overline{Q_5}\,\overline{Q_4}\,\overline{Q_3}\,\overline{Q_2}\,\overline{Q_1}\,\overline{Q_0}$ aufgenommen werden. Alle Flipflops müssen beim Einschalten der Betriebsspannung durch eine entsprechende *Power-on*-Schaltung auf 0 zurückgesetzt werden.

Die Steuersignale für das Operationswerk können direkt an den Flipflopausgängen abgenommen werden, nur zur Bestimmung von *Aktiv* wird ein zusätzliches OR-Schaltglied benötigt.

$$\begin{aligned}
Schieben &= Q_4 \\
Aktiv &= Q_1 \vee Q_4 \\
Reset &= Q_1 \\
Inkrement &= Q_3 \\
Fertig &= Q_5
\end{aligned}$$

4.7.6 Mealy-Steuerwerk als konventionelles Schaltwerk

Die Schaltfunktionen für das Mealy-Steuerwerk können wir wie im vorangehenden Abschnitt herleiten. Im ASM-Diagramm nach Abbildung 4.8 gibt es nur vier Zustände, so dass wir mit einem 2-Bit-Zustandsregister auskommen. Der Zustandsvektor wird wie folgt codiert:

z^t	z_0	z_1	z_2	z_3
Q_1Q_0	00	01	10	11

Wir gehen wieder davon aus, dass das Steuerwerksregister aus D-Flipflops aufgebaut ist. Die Folgezustände ergeben sich durch die Belegungen an den Eingängen der D-Flipflops:

$$z^{t+1} = D_1 D_0(z^t)$$

D_0 muss den Wert 1 annehmen, wenn der Folgezustand z_1 oder z_3 ist. z_1 erreicht man aus dem Zustand z_0, wenn $Start = 1$ ist, z_3 aus dem Zustand z_2, wenn $\overline{(A \neq 0)}$ ist. Damit erhalten wir:

$$\begin{aligned} D_0 &= z_0\, Start \vee z_2\, \overline{(A \neq 0)} \\ &= \overline{Q_1}\, \overline{Q_0}\, Start \vee Q_1\, \overline{Q_0}\, \overline{(A \neq 0)} \end{aligned}$$

D_1 muss den Wert 1 annehmen, wenn der Folgezustand z_2 oder z_3 ist. z_2 erreicht man aus dem Zustand z_1 oder aus dem Zustand z_2, wenn $(A \neq 0)$ ist. z_3 erhalten wir aus dem Zustand z_2, wenn $\overline{(A \neq 0)}$ gilt. Damit folgt:

$$\begin{aligned} D_1 &= z_1 \vee z_2\, (A \neq 0) \vee z_2\, \overline{(A \neq 0)} \\ &= z_1 \vee z_2 \\ &= \overline{Q_1}\, Q_0 \vee Q_1\, \overline{Q_0} \end{aligned}$$

Nachdem die Schaltfunktionen für das Übergangsschaltnetz g vorliegen, müssen noch die Schaltfunktionen für die Steuersignale des Operationswerks bestimmt werden (Ausgangsschaltnetz f):

$$\begin{aligned} Schieben &= z_2\, (A \neq 0) = Q_1\, \overline{Q_0}\, (A \neq 0) \\ Aktiv &= z_1 \vee z_2\, (A \neq 0) = \overline{Q_1}\, Q_0 \vee Q_1\, \overline{Q_0}\, (A \neq 0) \\ Reset &= z_1 = \overline{Q_1}\, Q_0 \\ Inkrement &= z_2\, A_0 = Q_1\, \overline{Q_0}\, A_0 \\ Fertig &= z_3 = Q_1\, Q_0 \end{aligned}$$

4.7.7 Mealy-Steuerwerk mit Hot-one-Codierung

Durch die Hot-one-Codierung kann der Entwurf erleichtert werden. Die vier Zustände werden dazu wie folgt codiert:

z^t	z_0	z_1	z_2	z_3
$Q_3Q_2Q_1Q_0$	0001	0010	0100	1000

Mit Hilfe von Abbildung 4.8 erhalten wir folgende Schaltfunktionen für die Flipflop-Eingänge:

$$\begin{aligned} D_0 &= Q_0\, \overline{Start} \vee Q_3 \vee \overline{Q_3}\, \overline{Q_2}\, \overline{Q_1}\, \overline{Q_0} \\ D_1 &= Q_0\, Start \\ D_2 &= Q_1 \vee Q_2\, (A \neq 0) \\ D_3 &= Q_2\, \overline{(A \neq 0)} \end{aligned}$$

Man beachte, dass der letzte Term in der Gleichung für D_0 wieder zur Initialisierung nötig ist. Die Schaltfunktionen für das Ausgangsschaltnetz lauten:

$$\begin{aligned} Schieben &= Q_2\ (A \neq 0) \\ Aktiv &= Q_1 \vee Q_2\ (A \neq 0) \\ Reset &= Q_1 \\ Inkrement &= Q_2\ A_0 \\ Fertig &= Q_3 \end{aligned}$$

4.7.8 Mikroprogrammierte Steuerwerke

Im Folgenden werden für die Moore- und Mealy-Variante mikroprogrammierte Steuerwerke vorgestellt. Um die Kapazität des Mikroprogrammspeichers optimal zu nutzen, sollten die ASM-Blöcke nur höchstens zwei Ausgänge (Folgezustände) haben. Das bedeutet, dass wir auch nur zwischen zwei möglichen Folgezustandsvektoren unterscheiden müssen. Im Mikroprogrammsteuerwerk ist dies durch einen 2:1-Multiplexer mit der Wortbreite des Zustandsvektors realisiert. Dieser Multiplexer wird durch ein Schaltnetz zur Abfrage der jeweiligen Testbedingung angesteuert. Eine Testbedingung wird über je ein AND-Schaltglied abgefragt, indem der zweite Eingang über eine Steuerleitung des Mikroprogrammspeicher aktiviert wird. Je nach Ergebnis dieses Tests wird dann aus einer Spalte des Mikroprogrammspeichers die Folgeadresse für das Zustandsregister ausgewählt. Parallel zur Erzeugung der Folgeadressen werden aus den Wortleitungen, die direkt den Zuständen zugeordnet sind[4], die Steuersignale für das Operationswerk abgeleitet.

In Abbildung 4.10 ist das Mikroprogrammsteuerwerk für die Moore-Variante dargestellt. Diese Lösung ist sehr ähnlich zu der Hot-one-Codierung aus 4.7.5. Dies gilt auch beim Mikroprogrammsteuerwerk für die Mealy-Variante, das in Abbildung 4.11 zu sehen ist. Auch dort können die Zustands- und Ausgangsschaltfunktionen direkt aus den Gleichungen aus Abschnitt 4.7.7 entnommen werden.

Zur Realisierung des „Einsen-Zählers“ haben wir ein speziell auf die Problemstellung abgestimmtes Operationswerk verwendet. Neben solchen anwendungsspezifischen Operationswerken kann man jedoch auch universell verwendbare Operationswerke konstruieren, die für viele verschiedene Problemstellungen eingesetzt werden können. Zur Lösung einer bestimmten Aufgabe mit Hilfe eines universellen Operationswerks müssen nur „genügend“ viele Register und Funktionsschaltnetze für „nützliche“ Operationen vorhanden sein. Da im allgemeinen nicht alle vorhandenen Komponenten vom Hardware-Algorithmus ausgenutzt werden, enthält ein universelles Operationswerk gegenüber einem anwendungsspezifischen Operationswerk stets ein gewisses Maß an *Redundanz*. Dafür eignen sie sich jedoch sehr gut zur Realisierung von Computern.

universelles Operationswerk

[4]Eine Optimierung der Zustandscodierung wie in 4.7.4 ist nicht erforderlich, da der Adressdecoder des Mikroprogrammspeichers alle Minterme der Adressen (Zustände) ermittelt.

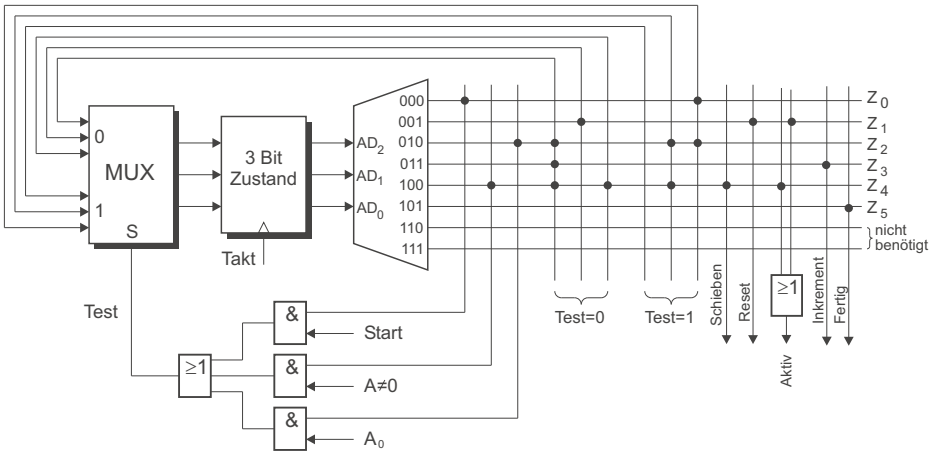


Abbildung 4.10: Mikroprogrammsteuerwerk der Moore-Variante.

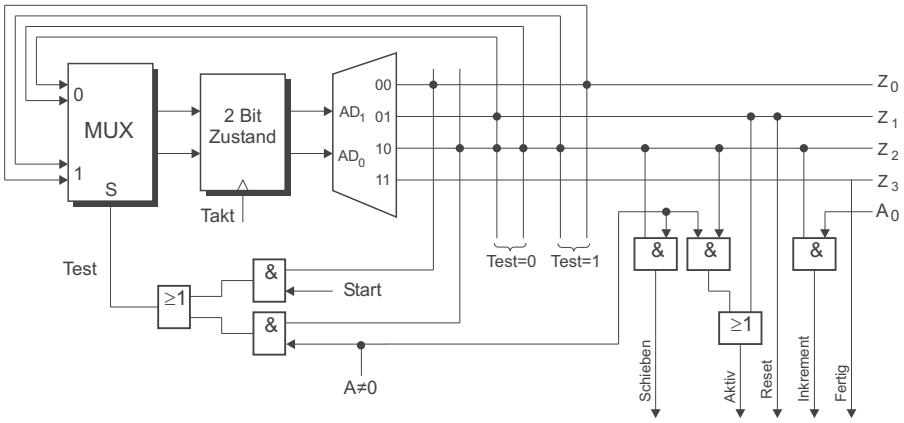


Abbildung 4.11: Mikroprogrammsteuerwerk der Mealy-Variante.

Selbsttestaufgabe 4.6 (Modulo-X-Zähler)

Entwerfen Sie ASM-Diagramm und Operationswerk für einen Modulo-X-Zähler. Der Wert soll über den Eingabevektor X vorgegeben werden und beträgt maximal 1000. Der Zähler soll beim Wert 0 starten. Die Zählerstände sollen in einem Register A gespeichert werden, das direkt mit dem Ausgabevektor Y verbunden ist. Zur Bestimmung des Statusvektors steht ein Vergleicher mit den Eingängen A und B zur Verfügung, der an seinem Ausgang A<B eine 1 ausgibt, sobald der Wert an A kleiner als B ist. Außerdem sei ein Addierschaltnetz der benötigten Wortbreite vorhanden.

1. *Wie groß ist die benötigte Wortbreite?*
2. *Skizzieren Sie das ASM-Diagramm!*
3. *Skizzieren Sie das Operationswerk!*
4. *In welchem Zustand nimmt Register A seinen maximalen Wert an?*

Lösung auf Seite 206

4.8 Grundlagen eines Computers

Ein Computer ergibt sich als Verallgemeinerung eines komplexen Schaltwerks. Im letzten Abschnitt haben wir gesehen, dass beim komplexen Schaltwerk (nach der Eingabe der Operanden) immer nur ein einziger Steueralgorithmus aktiviert wird. Angenommen, wir hätten Steueralgorithmen für die vier Grundrechenarten auf einem universellen Operationswerk entwickelt, das lediglich über einen Dual-Addierer verfügt. Durch eine Verkettung der einzelnen Steueralgorithmen könnten wir nun einen übergeordneten Algorithmus realisieren, der die Grundrechenarten als Operatoren benötigt. Die über Steueralgorithmen realisierten Rechenoperationen können durch *Operationscodes*, z.B. die Nummern 1-4, ausgewählt werden und stellen dem Programmierer Maschinenbefehle bereit, die er in seinem Programm benutzen kann.

Opcode

Die zentrale Idee eines Computers besteht nun darin, die Opcodes in einem *Speicher* abzulegen und sie vor der Ausführung durch das Steuerwerk selbst holen zu lassen. Es benötigt ein Befehlsregister für den Opcode und einen Befehlszähler für die Adressierung der Befehle im Speicher. Man bezeichnet ein solches Steuerwerk als *Leitwerk*. Ein universelles Operationswerk, das neben einer ALU auch mehrere Register enthält, nennt man *Rechenwerk*.

Speicher

Leitwerk und Rechenwerk bilden den *Prozessor* oder die *CPU* (*Central Processing Unit*). Damit der Prozessor nicht nur interne Berechungen ausführen kann, sondern auch von außen (Benutzer-)Daten ein- bzw. ausgegeben werden können, benötigt ein Computer eine Schnittstelle zur Umgebung, die als *Ein-/Ausgabe* bezeichnet wird.

Prozessor

Ein-/Ausgabe

Wir unterscheiden also insgesamt vier Funktionseinheiten: Rechenwerk, Leitwerk, Speicher und Ein-/Ausgabe. Das Blockschaltbild in Abbildung 4.12 zeigt, wie die einzelnen Komponenten miteinander verbunden sind. Die Verbindungen wurden entsprechend ihrer Nutzung gekennzeichnet. Hierbei steht D für Daten, B für Befehle, A für Adressen und S für Steuersignale. Die Zahl der Datenleitungen bestimmt die Maschinenwortbreite eines Prozessors.

Im Folgenden werden wir zunächst die grundlegende Arbeitsweise eines Computers beschreiben, die auf Arbeiten des Amerikaners John von Neumann und des Deutschen Konrad Zuse beruht. Im Anschluss an die Beschreibung der Grundlagen werden dann Erweiterungen vorgestellt, die sowohl die Implementierung als auch die Programmierung eines Computers vereinfachen. Danach wird ausführlicher auf den Aufbau von Rechen-, Leitwerk und Speicher eingegangen.

4.8.1 Rechenwerk

Ein Rechenwerk stellt ein *universelles Operationswerk* dar, das elementare arithmetische und logische Operationen beherrscht und über mehrere Register verfügt. Diese Register werden durch Adressen ausgewählt und nehmen die Operanden (Variablen oder Konstanten) auf, die miteinander verknüpft werden sollen. Die ALU kann jeweils zwei Registerinhalte arithmetisch oder logisch miteinander verknüpfen. Die Art der Verknüpfung wird durch ein Steuerwort bestimmt und das Ergebnis wird wieder in ein Register des Registerblocks zu-

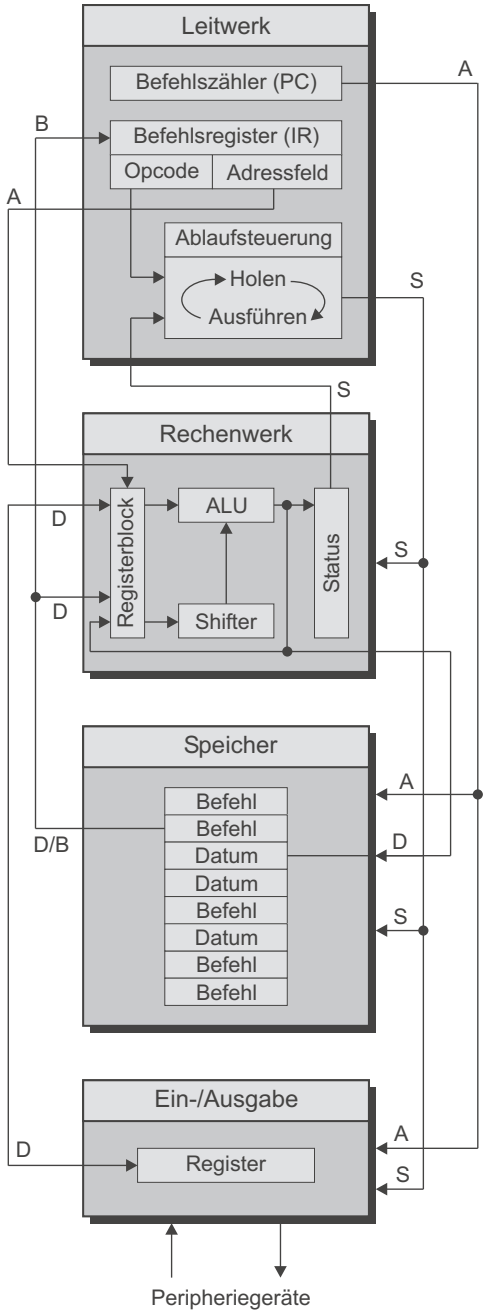


Abbildung 4.12: Blockschaltbild eines Computers.

rückgeschrieben. Meist ist auch eine Schiebeeinrichtung (Shifter) vorhanden, mit der die Datenbits um eine oder mehrere Stellen nach links oder rechts verschoben werden können. Der Shifter ist besonders nützlich, wenn zwei binäre Zahlen multipliziert oder dividiert werden sollen und die ALU nur addieren kann. Das Statusregister dient zur Anzeige besonderer Ergebnisse, die das Leitwerk auswertet, um bedingte Verzweigungen in Steueralgorithmen für Maschinenbefehle auszuführen. Die einzelnen Bits des Statusregisters bezeichnet man als *Flags*. Das Statusregister kann wie ein normales (Daten-)Register gelesen und beschrieben werden.

Status-Flags

4.8.2 Leitwerk

Das Leitwerk stellt ein *umschaltbares Steuerwerk* dar, das – gesteuert durch den *Operationscode* (Opcode) der Maschinenbefehle – die zugehörigen Steueralgorithmen auswählt. Es steuert den Ablauf eines Programms, indem es Maschinenbefehle aus dem Speicher holt, im *Befehlsregister* IR (Instruction Register) speichert und die einzelnen Operationscodes in eine Folge von Steuerwörtern (Steueralgorithmus) umsetzt. Es arbeitet zyklisch, d.h. die Holephase (fetch) und die Ausführungsphase (execute) wiederholen sich ständig (Abbildung 4.13). Der *Befehlszähler PC* (Program Counter) wird benötigt, um im Speicher die Befehle und – bei dem hier behandelten Grundprinzip – auch deren Operanden zu adressieren. Während der Holephase zeigt sein Inhalt auf den nächsten Maschinenbefehl. Nachdem ein neuer Maschinenbefehl geholt ist, wird der Befehlszähler erhöht.

Befehlsregister

Befehlszähler

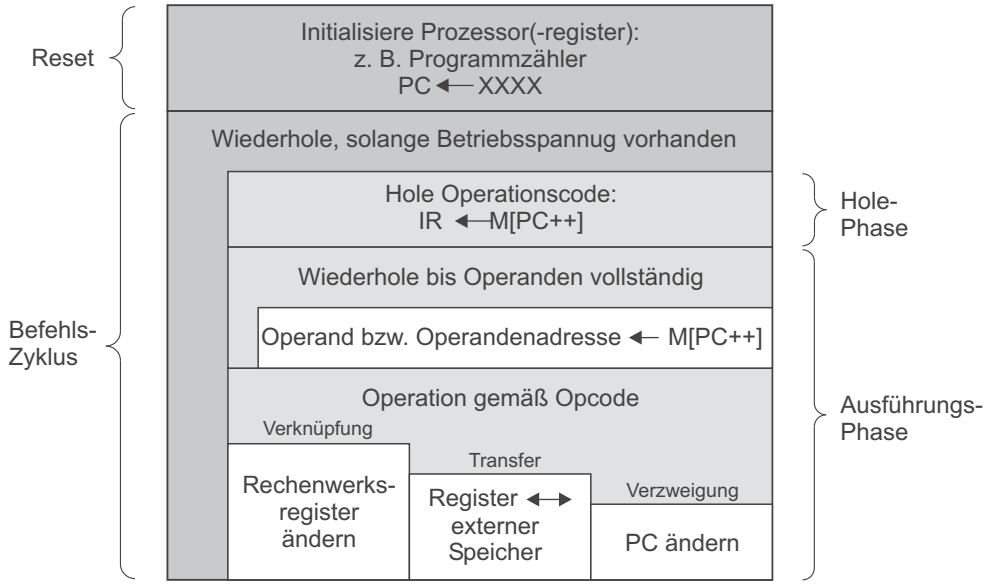


Abbildung 4.13: Befehlsabarbeitung in einem Computer.

Wie man sieht, wird ein Maschinenbefehl in mehreren Teilschritten abgearbeitet. Die Holephase ist für alle Befehle gleich. Damit ein neuer Maschinenbefehl in einem Taktzyklus geholt werden kann, setzt man eine Speicherhierarchie mit einem schnellen Cache-Speicher ein (siehe Abschnitt 4.14). Abhängig von der *Befehlssatzarchitektur*[5] des Prozessors kann die Ausführungsphase eine variable oder feste Anzahl von Taktschritten umfassen. Der Abbildung 4.13 entnimmt man, dass ein Prozessor im Wesentlichen drei Befehlsklassen unterstützt: Verknüpfungs-, Datentransfer- und Verzweigungsbefehle. Im Folgenden geben wir jeweils die notwendigen Verarbeitungsschritte für die Maschinenbefehle aus diesen Klassen an. Es wird davon ausgegangen, dass der Befehl bereits geholt wurde und während der Ausführungsphase im Befehlsregister gespeichert

[5]Der Befehlssatz bestimmt die Implementierung eines Prozessors durch eine so genannte *Mikroarchitektur*, die ebenfalls die Zahl der benötigten Taktschritte beeinflusst.

bleibt. Danach wird unter der *PC*-Adresse der nächste Befehl geholt, und dieser Zyklus wiederholt sich, solange der Prozessor mit Betriebsspannung versorgt wird[6].

1. Verknüpfung von Operanden

- Auslesen der beiden Operanden aus dem Registerblock,
- Verknüpfung in der ALU,
- Schreiben des Ergebnisses in den Registerblock,
- neuen *PC*-Wert bestimmen.

Die Registeradressen sind Bestandteil des Maschinenbefehls und werden neben dem Opcode in einem Adressfeld angegeben.

2. Datentransfer zwischen Speicher und Prozessor

Hier muss zwischen Lese- und Schreibzugriffen unterschieden werden:

a) Lesen (load)

- Bestimmung der Speicheradresse des Quelloperanden,
- Lesezugriff auf den Speicher (Speicheradresse ausgeben),
- Speichern des gelesenen Datums in das Zielregister,
- neuen *PC*-Wert bestimmen.

b) Schreiben (store)

- Bestimmung der Speicheradresse des Zieloperanden,
- gleichzeitig kann der Inhalt des Quellregisters ausgelesen werden,
- Schreibzugriff auf den Speicher (Speicheradresse und Datum ausgeben),
- Datum speichern,
- neuen *PC*-Wert bestimmen.

Die Speicheradresse wird als Summe eines Registerinhalts und einer konstanten Verschiebung (displacement, offset) bestimmt. Beide Angaben sind Bestandteil des Maschinenbefehls und werden im *Adressfeld* (s.u.) spezifiziert.

[6]Ausnahme: Bei manchen Prozessoren kann die Befehlsverarbeitung mit einem *HALT*-Befehl gestoppt werden.

3. Verzweigungsbefehle

Man kann in Maschinenprogrammen zwei Arten von Verzweigungen unterscheiden. Eine *unbedingte* Verzweigung oder *Sprung* (Jump) führt dazu, dass der *PC* mit einem neuen Wert überschrieben wird. Der neue *PC*-Wert wird entweder als Operand des Sprung-Befehls direkt vorgegeben (absolute Adresse) oder er wird berechnet, indem zum alten *PC*-Wert eine Adressverschiebung (Offset) addiert wird. Da die Addition des Zweierkomplements einer Subtraktion entspricht, kann man damit auch Rückwärtssprünge ausführen. Im Fall der Adressberechnung spricht man von *PC*-relativer sonst von absoluter Adressierung. Die *bedingte* Verzweigung (Branch) wird nur dann ausgeführt, wenn eine im Befehl vorgegebene Bedingung erfüllt ist. Im Falle der Ausführung wird sie analog zur unbedingten Verzeigung behandelt. Folgende Schritte werden bei Verzweigungen ausgeführt:

- Bestimmung des neuen *PC*-Werts für das Verzweigungsziel,
- Prüfung der Verzweigungsbedingung (entfällt bei Sprüngen),
- bei Sprüngen immer, bei Verzweigungen bedingt: Überschreiben des *PC* mit dem neuen Wert.

Damit ein Maschinenprogramm innerhalb des Adressraums verschiebbar bleibt, empfiehlt es sich, keine absolute, sondern nur *PC*-relative Adressierung zuzulassen.

Nicht alle aufgeführten Teilschritte der Ausführungsphase müssen in der angegebenen Reihenfolge abgearbeitet werden. So kann beispielsweise bei den beiden erstgenannten Befehlsklassen der neue PC-Wert gleichzeitig zum vorhergehenden Teilschritt ermittelt werden. Voraussetzung ist natürlich, dass eine zweite ALU vorhanden ist.

Die o.g. Teilschritte zur Befehlsausführung erfordern, abhängig vom jeweiligen Maschinenbefehl, *einen* oder *mehrere* Taktzyklen. Die Dauer einzelner Befehle hängt stark von den unterstützten *Adressierungsarten* ab. Diese legen fest, wie während der Befehlsausführung auf die Operanden zugegriffen wird. Je mehr Adressierungsmöglichkeiten zur Verfügung stehen, desto kürzer werden die Maschinenprogramme. Andererseits steigen aber der Hardwareaufwand und die Zahl der benötigten Taktzyklen zur Ausführung einzelner Befehle.

Adressierungsarten

Die Implementierung eines Befehlssatzes ist umso einfacher, je gleichartiger die Teilschritte der drei Befehlsklassen sind. Wenn es schließlich gelingt — als kleinsten gemeinsamen Nenner für alle Befehle — eine feste Zahl von Teilschritten zu finden, wird eine Implementierung durch eine *Befehlspipeline* möglich. Dabei können die Teilschritte mehrerer Befehle zeitlich überlappend ausgeführt werden. Wie bei einem Fließband kann dadurch der Befehls*durchsatz* erheblich gesteigert werden. Diese Philosophie wurde seit Mitte der achtziger Jahre konsequent bei so genannten *RISC-Prozessoren* (Reduced Instruction Set Computer) eingesetzt.

RISC-Prozessoren

Dagegen werden Prozessoren, die einen möglichst hohen Komfort bei der Programmierung in Maschinensprache bieten, als so genannte *CISC-Prozessoren*

CISC-Prozessoren

(Complex Instruction Set Computer) bezeichnet. Während RISC-Prozessoren Speicherzugriffe nur mit den o.g. Load- und Store-Befehlen ermöglichen, können CISC-Befehle die Operanden über vielfältige Adressierungsarten sogar direkt im Speicher ansprechen. Solche Befehle benötigen dann zwar keine expliziten Load- und Store-Operationen, aber ihr Implementierungsaufwand ist deutlich höher. Außerdem können die Befehle wegen ihrer unterschiedlichen Länge (die Zahl der Teilschritte ist variabel) nur nacheinander ausgeführt werden. Eine Fließbandverarbeitung wie bei RISC-Prozessoren ist daher nicht oder nur eingeschränkt möglich.

CISC-Prozessoren findet man heute vor allem in Mikrocontrollern, die beispielsweise in Haushalts- und Unterhaltungsgeräten oder in Fahrzeugen zum Einsatz kommen. In Personalcomputern oder Hochleistungsrechnern (Servern) werden dagegen überwiegend RISC-Prozessoren bzw. -Prozessorkerne verwendet.

Befehlsformate

Maschinenbefehle bestehen aus dem *Opcode* und einem *Adressfeld*, das entweder die Rechenwerksregister direkt adressiert, eine absolute Speicheradresse oder die Adresse eines *Indexregister*s sowie eine zusätzliche Adressverschiebung (Offset) zu dessen Inhalt enthält.

Ein-Wort-Befehle

Wenn die Operanden in Rechenwerksregistern stehen, genügt ein einziges Maschinenwort zur Darstellung eines Befehls. Solche *Ein-Wort-Befehle* findet man sowohl bei CISC- als auch bei RISC-Prozessoren, für die sie typisch sind. Ein Ein-Wort-Befehl hat folgendes Befehlsformat:

Opcode	Adressfeld

Mehr-Wort-Befehle

Wenn die Operanden oder Daten direkt im Speicher abgelegt sind, kann ein Befehl aus zwei oder drei Maschinenwörtern bestehen. Man findet solche Befehle ausschließlich bei CISC-Prozessoren. Ein typischer CISC-Befehl hat z.B. folgendes Format:

Opcode	Adressfeld
Adresse 2. Operand	
Adresse Ergebnis	

In diesem Beispiel steht der 1. Operand in einem Register, das über das Adressfeld des Befehls adressiert wird. Der 2. Operand wird aus dem Speicher geholt. Hierzu muss zunächst dessen Adresse aus dem Speicher geladen und in ein (für den Programmierer nicht sichtbares) Adressregister geschrieben werden. Mit dieser Adresse erfolgt dann ein weiterer Speicherzugriff, um den 2. Operanden in ein prozessorinternes Hilfsregister zu kopieren. Dann wird die Verknüpfung gemäß dem Opcode durchgeführt und das Ergebnis in einem Hilfsregister zwischengespeichert. Um es in den Speicher zu transportieren, muss zunächst wieder die Speicheradresse für das Ergebnis geholt und ins (für den Programmierer nicht sichtbare) Adressregister geladen werden. Der Speicher wird dann

über das Adressregister angesprochen und der Inhalt des Hilfsregisters in den Speicher kopiert. Wir sehen, dass ein solcher CISC-Befehl nicht nur sehr viele Teilschritte bzw. Taktzyklen, sondern auch zusätzliche Hardware erfordert (hier ein Hilfs- und Adressregister).

Zur symbolischen Darstellung von Maschinenbefehlen werden Abkürzungen benutzt, die sich ein Programmierer leichter einprägen kann als eine Funktionsbeschreibung oder gar den binären Opcode eines Befehls. Ein Programm, das aus solchen so genannten *Mnemonics* besteht, bezeichnet man als *Assemblerprogramm*. Es kann mit Hilfe einer Entwicklungssoftware, die man *Assembler* nennt, in die binäre Darstellung der Maschinenbefehle übersetzt werden. Assembler erlauben auch die Verwendung symbolischer Namen für Konstanten, Variablen und Sprungziele.

Mnemonic

Assembler

4.8.3 Speicher

Im Speicher eines Computers werden sowohl Maschinenbefehle als auch Daten abgelegt. Jeder Speicherplatz ist über eine binäre Adresse ansprechbar und kann ein Maschinenwort (z.B. 32 oder 64 Bit) aufnehmen. Bei modernen Computersystemen werden verschiedene Arten von Speichern benutzt, um möglichst geringe Kosten pro Bit und gleichzeitig hohe Zugriffsraten zu erreichen. Direkt mit dem Prozessor verbunden sind die Halbleiterspeicher, bei denen man Schreib-/Lese-Speicher (Random Access Memory, RAM) und Nur-Lese- Speicher (Read-Only Memory, ROM) unterscheidet. Jedes Computersystem enthält einen ROM-Speicher, in dem ein einfaches *Betriebssystem* (Monitor) oder kleines Urladeprogramm (Bootstrap Loader) permanent gespeichert ist. Bei den meisten Computern wird das Betriebssystem nach dem Einschalten von einem magnetomotorischen Speicher (Festplatte) geladen. Der Hauptspeicher wird im Allgemeinen mit *dynamischen* Speicherbausteinen (vgl. Abschnitt 4.14) aufgebaut, deren Zugriffszeiten etwa um einen Faktor bis 100 größer sind als die Taktzykluszeit des Prozessors. Durch den Einbau von schnellen Pufferspeichern zwischen Prozessor und Hauptspeicher können die Geschwindigkeitsverluste bei Speicherzugriffen verringert werden. Diese *Cache*-Speicher halten häufig benötigte Speicherblöcke aus dem Hauptspeicher für den Prozessor bereit.

4.8.4 Ein-/Ausgabe

Die Ein-/Ausgabe dient als Schnittstelle eines Computersystems zur Umwelt. Sie verbindet Peripheriegeräte wie z.B. Tastatur, Monitor oder Drucker mit dem Prozessor. Durch *direkten Speicherzugriff* (Direct Memory Access, DMA) oder einen *Ein-/Ausgabe-Prozessor* (Input/Output Processor, IOP) kann die CPU bei der Übertragung großer Datenblöcke entlastet werden. Die Ein-/Ausgabe erfolgt dann ohne zeitraubende Umwege über das Rechenwerk des Prozessors unmittelbar zwischen der Ein-/Ausgabe und dem Hauptspeicher. Dies ist besonders beim Anschluss von Festplatten wichtig, damit z.B. Daten von der Festplatten-Steuereinheit (Controller) ohne unnötige Zeitverzögerung abgeholt werden.

Die hier skizzierten Grundlagen eines Computers finden sich bis heute bei

integrierten Prozessoren (Mikroprozessoren) wieder. Im Laufe der technologischen Entwicklung wurden jedoch Modifikationen bzw. Erweiterungen vorgenommen, um die Implementierung zu vereinfachen, die Leistung zu erhöhen und die Programmierung zu erleichtern. Diese Maßnahmen werden im Folgenden vorgestellt.

4.9 Interne und externe Busse

Wir können die Zahl der Verbindungsleitungen für die Übertragung von Daten bzw. Befehlen nach Abbildung 4.12 reduzieren, wenn wir einen *Datenbus* einführen. Ein Bus besteht aus einem Bündel von Leitungen, die alle dem gleichen Zweck dienen. Durch die zeitlich versetzte Nutzung (Zeitmultiplex) werden die vorhandenen Leitungen besser ausgenutzt. Außerdem verringert sich auch die Gesamtzahl der Leitungen. Mit Hilfe von Adress- und Steuerleitungen, die Bustreiber (z.B. TriState) oder elektronische Schalter (CMOS[7]-Transmission Gates) ansteuern, können Datenpfade in zwei Richtungen (bidirektional) geschaltet werden. Der *interne Datenbus* ist i.a. auf die Maschinenwortbreite ausgelegt und kann von zwei verschiedenen Datenquellen angesteuert werden (Abbildung 4.14): mögliche Quellen sind die Rechenwerksregister und der *externe Datenbus*. Um die Zahl der Anschlusskontakte (Pins) eines Prozessorchips zu minimieren, ist der externe Datenbus jedoch nicht immer auf die volle Maschinenwortbreite ausgelegt. So verfügt z.B. der Motorola 68008 über einen internen Datenbus mit 32 Bit und einen externen Datenbus mit nur 8 Bit. Dem geringeren Verdrahtungsaufwand steht jedoch ein Verlust an Übertragungsgeschwindigkeit (Busbandbreite) gegenüber. Der externe Datenbus wird über bidirektionale Bustreiber an den internen Datenbus angekoppelt.

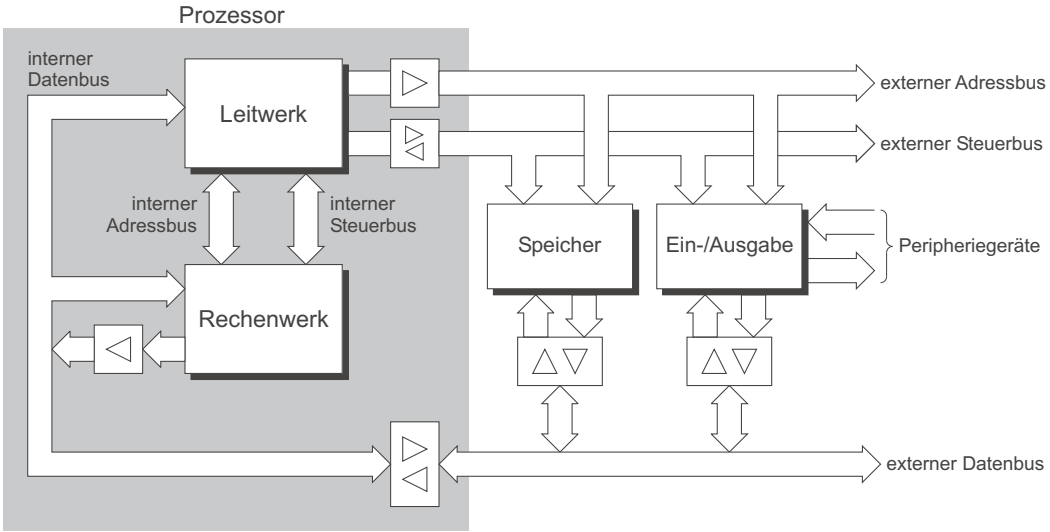


Abbildung 4.14: Interne und externe Busse erleichtern die Realisierung eines Prozessors.

[7]Complementary Metal Oxide Semiconductor.

Das Leitwerk versorgt das Rechenwerk mit Adressleitungen für die Register. Die internen Steuerleitungen bilden zusammen mit den Zustandsflags des Rechenwerks den *internen Steuerbus*. Über Bustreiber werden die Steuersignale und Adressen für das externe Bussystem verstärkt. Dies ist nötig, damit eine große Zahl von Speicher- und Ein-/Ausgabe-Bausteinen angeschlossen werden kann. Die Bustreiber wirken als Leistungsverstärker und erhöhen die Zahl der anschließbaren Schaltglieder. Der Ausgang eines TriState-Bustreibers hat wie sein Eingang entweder die logische Belegung 0 oder 1, oder er befindet sich im hochohmigen Zustand, d.h. der Ausgang ist (wie durch einen geöffneten Schalter) von der Busleitung abgekoppelt. Dieser (dritte) Betriebszustand kann über eine zusätzliche Steuervariable ausgewählt werden. Die bidirektionalen Bustreiber, mit denen die Speicher- und Ein-/Ausgabe-Bausteine an den externen Datenbus angeschlossen sind, werden durch die (externen) Steuer- und Adressleitungen geschaltet. Die Richtung der Datenübertragung wird mit einer $R/\overline{W}$-Steuerleitung (Read/Write) bestimmt.

Jeder externe Baustein wird durch eine Adresse bzw. innerhalb eines Adressbereichs angesprochen. Die Bustreiber-Ausgänge in Richtung Datenbus dürfen unter einer bestimmten Adresse nur bei *einem* einzigen externen Baustein aktiviert werden. Die hierzu erforderliche Adressdecodierung übernimmt ein besonderes Vergleicher-Schaltnetz. Bei einigen Prozessoren wird durch eine Steuerleitung $I/\overline{O}$ (Input/Output) zwischen einer Adresse für die Ein-/Ausgabe und Adressen für den Hauptspeicher unterschieden. Man kann aber die Ein-/Ausgabe auch wie einen Speicher bzw. -bereich behandeln. Dieses Verfahren wird als *Memory-Mapped Input/Output* bezeichnet.

Durch die Einführung interner und externer Busse wird die Realisierung von Mikrochips und deren Einsatz zum Aufbau von Computersystemen erleichtert. Die Gesamtheit von externem Adress-, Daten- und Steuerbus wird auch *Systembus* oder *Systembusschnittstelle* genannt. Wegen des geringeren Verdrahtungsaufwands ist es einfacher, den Schaltplan zu entflechten und ein Chip- oder Leiterplattenlayout zu erstellen. Ein weiterer Vorteil ist die geringere Zahl von Anschlüssen bei Prozessoren, Speicher- und Ein-/Ausgabe-Bausteinen.

Systembus

4.10 Prozessorregister

Die in einem Prozessor enthaltenen Register können in drei Klassen eingeteilt werden:

1. *Daten*register zur Aufnahme von Operanden und zur Speicherung von Ergebnissen,
2. *Adress*register zur Adressierung von Operanden,
3. *Steuer*register, die den Ablauf der Befehlsverarbeitung steuern und besondere Programmiertechniken unterstützen.

Die Datenregister dienen zur kurzzeitigen Speicherung von Variablen oder Konstanten eines Programms. Da die Datenregister im Gegensatz zum Hauptspeicher ohne zusätzliche Zeitverzögerungen benutzt werden können, ist es ratsam,

häufig benötigte Operanden in Datenregister zu übertragen, die Ergebnisse zu berechnen und dann in den Hauptspeicher zurückzuschreiben.

Beim Zugriff auf Operanden im Hauptspeicher wird nach dem Grundprinzip der Programmzähler als Adresse benutzt. Mit dieser Vorgehensweise können aber nur diejenigen Speicherplätze adressiert werden, die unmittelbar auf einen Maschinenbefehl folgen. Um Operanden überall im Hauptspeicher ablegen zu können, verwendet man daher Adressregister. Die verschiedenen *Adressierungsarten*, die man in Verbindung mit Adressregistern realisieren kann, werden im Kurs 1609 vorgestellt. Bei den meisten heutigen Prozessoren kann jedes Register aus dem Registerblock sowohl als Daten- als auch Adressregister verwendet werden.

Stackpointer

Zu den Steuerregistern zählt das Befehlsregister, das Statusregister, der Programmzähler und der *Stackpointer* (Stapelzeiger, SP), dessen Aufgabe es ist, im Hauptspeicher einen *Stack* (Stapelspeicher[8]) zu verwalten. Neben den genannten Steuerregistern gibt es bei manchen Prozessoren (z.B. Intel x86-Prozessoren) auch *Segmentregister*, welche die Speicherverwaltung durch das Betriebssystem unterstützen. Daneben verwaltet das Leitwerk auch prozessorinterne Register, auf die der Anwender nicht zugreifen kann.

Wir wollen im Folgenden auf die Anwendung des Stackpointers näher eingehen. Als Stackpointer wird ein besonderes Register bezeichnet, das zur Indizierung von Speicherzugriffen dient, um im Speicher einen Stack zu implementieren. Zum Schutz des Betriebssystems verwendet man häufig zwei getrennte Stackpointer, die man als *User-* bzw. *Supervisor-Stackpointer* bezeichnet.

4.11 Anwendungen des Stackpointers

Ein Stack ist ein wichtiges Hilfsmittel, das die Verarbeitung von *Unterprogrammen* (Subroutines) und *Unterbrechungen* (Interrupts) ermöglicht. Man kann einen Stack entweder direkt in Hardware auf dem Prozessorchip realisieren oder mit Hilfe eines Stackpointers in den Hauptspeicher abbilden. Die erste Methode ist zwar schneller, erfordert aber mehr Chipfläche als ein einziges SP-Register. Außerdem kann bei der Stackpointer-Methode die Speichertiefe des Stacks durch zusätzlichen Hauptspeicher beliebig vergrößert werden kann.

LIFO-Prinzip

Ein Stack arbeitet nach dem *LIFO-Prinzip* (Last In, First Out). Dabei sind nur zwei Operationen erlaubt: PUSH und POP. Mit der PUSH-Operation wird ein Maschinenwort auf den Stack gelegt und mit der POP-Operation wird es wieder zurückgeholt. Während des Zugriffs auf den Hauptspeicher wird der Stackpointer als Adresse (Zeiger) benutzt. Außerdem wird der Stackpointer durch die Ablaufsteuerung so verändert, dass ein Zugriff das LIFO-Prinzip erfüllt. Die Organisation eines Stacks ist prozessorspezifisch. Meist beginnt er am Ende des Hauptspeichers und „wächst“ in Richtung der niedrigeren Adressen. Für diesen Fall beginnt das Programm am Anfang des Hauptspeichers. An dieses schließt sich meist noch ein Datenbereich an. Der Stack darf niemals so groß werden, dass er den Daten- oder sogar Programmbereich überschreibt (Stack-Overflow). Wenn der Stack trotzdem überläuft (z.B. aufgrund eines Program-

[8] Oft auch Kellerspeicher genannt.

mierfehlers), muss das Betriebssystem das betreffende Programm abbrechen.

Es gibt zwei Möglichkeiten, den Stackpointer zu verändern:

1. vor der PUSH-Operation und nach der POP-Operation,
2. nach der PUSH-Operation und vor der POP-Operation.

Hier ein Beispiel für den ersten Fall. Der Stack soll am Ende des Hauptspeichers beginnen. Der Stackpointer muss deshalb mit der letzten Hauptspeicheradresse +1 initialisiert werden. Wenn das Register A auf den Stack gelegt werden soll (PUSH A), muss die Ablaufsteuerung zunächst den Stackpointer SP dekrementieren und dann den Inhalt von Register A in den Speicherplatz schreiben, auf den SP zeigt. In RTL-Schreibweise:

$$\begin{array}{ll} \text{PUSH } A\text{:} & SP \leftarrow SP - 1 \\ & M[SP] \leftarrow A \end{array}$$

Entsprechend dazu wird die POP-Operation definiert:

$$\begin{array}{ll} \text{POP } A\text{:} & A \leftarrow M[SP] \\ & SP \leftarrow SP + 1 \end{array}$$

Hier wird der Stackpointer zuerst zur Adressierung benutzt und erst inkrementiert, nachdem A vom Stack geholt wurde.

Der Stack kann auch benutzt werden, um andere Prozessorregister kurzzeitig zu speichern[9]. Dies ist z.B. nötig, wenn diese Register von einem Unterprogramm oder einer Unterbrechung benutzt werden. Zu Beginn eines solchen Programmteils werden die betreffenden Register mit PUSH-Operationen auf den Stack gebracht. Vor dem Rücksprung an die Aufruf- bzw. Unterbrechungsstelle werden sie dann durch POP-Operationen in umgekehrter Reihenfolge in den Prozessor zurückgeholt. Man beachte, dass die Zahl der PUSH- und POP-Operationen innerhalb eines Unterprogramms (bzw. Unterbrechung) stets gleich groß sein muss.

4.11.1 Unterprogramme

Wenn an verschiedenen Stellen in einem Programm immer wieder die gleichen Funktionen benötigt werden, faßt man die dazugehörenden Befehlsfolgen in einem *Unterprogramm* zusammen. Dadurch wird nicht nur Speicherplatz gespart, sondern auch die Programmierung *modularisiert*. Im Allgemeinen übergibt man Parameter an ein Unterprogramm, die als Eingabewerte für die auszuführende Funktion benutzt werden. Die Parameter und Ergebnisse einer Funktion können in bestimmten Registern oder über den Stack übergeben werden. Eine Sammlung von „nützlichen“ Unterprogrammen kann in einer (Laufzeit-)Bibliothek bereitgestellt werden. Sie enthält lauffähige Maschinenprogramme, die einfach in Anwenderprogramme eingebunden werden können. Ein Unterprogramm wird mit dem CALL-Befehl aufgerufen und mit einem RETURN-Befehl abgeschlossen.

[9] Wird oft auch als „retten“ bezeichnet.

CALL-Befehl

Der CALL-Befehl bewirkt eine Programmverzweigung in das Unterprogramm und speichert die *Rücksprungadresse* auf dem Stack. Die Ausführung eines CALL-Befehls in der Form *Call Ziel* läuft gewöhnlich in vier Schritten ab.

1. Nachdem der CALL-Befehl geholt wurde, wird der Operationscode dekodiert. Der Programmzähler wird inkrementiert, damit er auf den nachfolgenden Befehl zeigt.
$$PC{+}{+}$$

2. Nun wird die Verzweigungsadresse[10] in ein zusätzliches Adressregister AR gerettet.
$$AR \leftarrow Ziel$$

3. Damit der Prozessor das Hauptprogramm nach Abarbeiten des Unterprogramms an dieser Stelle fortsetzen kann, muss er diese Rücksprungadresse auf den Stack legen.
$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow PC$$

4. Im letzten Schritt wird die Startadresse des Unterprogramms aus dem Adressregister AR in den Programmzähler geladen.
$$PC \leftarrow AR$$

   Nun beginnt die Befehlsabarbeitung des Unterprogramms, indem der Opcode unter der neuen Programmzähleradresse geholt wird. Sollen Parameter an Unterprogramme über den Stack übergeben werden, so werden diese vor dem Call-Befehl auf den Stack gelegt. Beim Eintritt in das Unterprogramm muss die Rücksprungadresse vom Stack in ein freies Prozessorregister gerettet werden. Unmittelbar vor dem Rücksprung muss der Inhalt dieses Registers mit einer PUSH-Operation auf den Stack zurückgeschrieben werden. Eventuell werden zuvor auch noch Ergebnisse für das aufrufende Programm auf dem Stack abgelegt.

RETURN-Befehl

Ein Unterprogramm muss mit einem RETURN-Befehl (Return from Subroutine, RTS) abgeschlossen werden. Der RETURN-Befehl bewirkt eine Umkehrung von Schritt 3 des CALL-Befehls. Die Rücksprungadresse wird durch eine POP-Operation vom Stack geholt und in den Programmzähler geschrieben.

$$PC \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

[10] Startadresse des Unterprogramms.

Nachdem die Rücksprungadresse im Programmzähler wiederhergestellt ist, kann das Hauptprogramm, beginnend mit einer Befehlsholephase, fortgesetzt werden.

Verschachtelung und Rekursion. Das beschriebene Verfahren zur Unterprogramm-Verarbeitung ermöglicht auch den Aufruf von Unterprogrammen aus Unterprogrammen. Wenn sich Unterprogramme selbst oder gegenseitig aufrufen, spricht man von direkter oder indirekter Rekursion. Rekursive Aufrufe von Unterprogrammen sind nur dann zulässig, wenn die im Unterprogramm benutzten Registerinhalte zu Beginn mit PUSH-Befehlen auf den Stack gerettet und am Ende des Unterprogramms durch POP-Befehle (in umgekehrter Reihenfolge wie die PUSH-Befehle) wiederhergestellt werden. Die mögliche Verschachtelungstiefe hängt in allen Fällen von der Größe des für den Stack verfügbaren Hauptspeicherbereichs ab. Folglich muss auch ein entsprechendes Rekursionsende gewährleistet sein. Ist dies nicht der Fall, so kommt es zu einem Überlauf (Stack Overflow).

Natürlich ist es auch bei nicht-rekursiven Unterprogrammen ratsam, die darin benutzten Register zu Beginn des Unterprogramms auf den Stack zu retten und vor dem Rücksprung ins aufrufende Programm von dort wiederherzustellen. Der damit verbundene Zeitaufwand ist allerdings nur dann gerechtfertigt, wenn das aufrufende Programm die Bewahrung des Prozessorzustands auch tatsächlich erfordert. Dies trifft für die während rekursiver Unterprogrammaufrufe benutzten Register immer zu. Wenn das aufrufende Programm jedoch bestimmte Prozessorregister nicht oder nur als Hilfsregister nutzt, müssen diese von den Unterprogrammen auch nicht gerettet werden.

Zeitbedarf. Der dynamische Aufbau einer Verbindung zwischen dem Haupt- und Unterprogramm mit Hilfe der Befehle CALL und RETURN erhöht den Zeitbedarf zur Ausführung einer Funktion. Dieser zusätzliche Zeitaufwand ist der Preis für den eingesparten Speicherplatz. Bei zeitkritischen Anwendungen kann es erforderlich sein, auf Unterprogramme zu verzichten und stattdessen *Makros* zu verwenden. Hier werden an die Stelle eines CALL-Befehls alle Maschinenbefehle einer Funktion eingefügt. Obwohl sich dadurch der Speicherbedarf erhöht, wird die Zeit für den Aufruf eines Unterprogramms und für die Rückkehr zum Hauptprogramm eingespart. Dies ist insbesondere bei einfachen Funktionen günstiger. Assembler, welche die Verwendung von Makros unterstützen, werden *Makroassembler* genannt. Der Programmierer kann einer Folge von Maschinenbefehlen einen symbolischen Namen zuordnen. Immer wenn dieser Name im weiteren Verlauf des Programms auftritt, wird er durch diese Befehlsfolge ersetzt. Neben Makroassemblern erlauben auch einige problemorientierte Sprachen wie z.B. „C“ die Verwendung von Makros.

Makros

4.11.2 Interrupts

Besondere Ereignisse inner- oder außerhalb des Prozessors können *Interrupts* erzeugen, die zu einer Unterbrechung des normalen Programmablaufs führen. Der Prozessor verzweigt zu einem Programmteil, der auf das eingetretene Ereignis reagiert (Interrupt Service Routine). Interrupts werden vom Prozessor ähn-

lich wie Unterprogramme behandelt. Sie unterscheiden sich von Unterprogrammen lediglich durch die Art des Aufrufs. Interrupts werden nicht durch einen CALL-Befehl ausgelöst, sondern durch intern oder extern erzeugte Signale. Sie können an jeder beliebigen Stelle (asynchron zum Prozessortakt) während der Abarbeitung eines Programms eintreffen. Wenn der Prozessor einen Interrupt akzeptiert, erfolgt eine (mehr oder weniger) schnelle Reaktion auf das durch das Signal angezeigte Ereignis.

Anwendungen

Es gibt eine Vielzahl von Anwendungen für Interrupts. Wir wollen im Folgenden einige Beispiele kurz beschreiben.

Ein-/Ausgabe. Bei der Ein-/Ausgabe muss sich der Prozessor mit den angeschlossenen Peripheriegeräten synchronisieren. Eine Möglichkeit hierzu besteht darin, ständig den Zustand der Ein-/Ausgabe-Bausteine abzufragen (Busy-Waiting). Wesentlich günstiger ist jedoch die Interrupt-gesteuerte Ein-/Ausgabe. Hier wird der Prozessor nur dann unterbrochen, wenn der Ein-/Ausgabe-Baustein bereit ist, Daten zu senden oder zu empfangen. Erst wenn diese Bereitschaft vorhanden ist (Ereignis), wird sie mittels eines Signals gemeldet. In der Zwischenzeit kann der Prozessor nützlichere Dinge tun, als auf das Peripheriegerät zu warten.

Betriebssysteme. In so genannten Multitasking-Systemen schaltet der Prozessor ständig zwischen verschiedenen Benutzern (bzw. Prozessen) um. Jeder Benutzer erhält den Prozessor nur während einer bestimmten Zeitspanne (Time Slice). Die Zuteilung des Prozessors wird durch den Betriebssystemkern vorgenommen, der nach dem Ablauf einer Zeitscheibe durch einen Interrupt aufgerufen wird. Die Erzeugung dieser Interruptsignale erfolgt i.Allg. mit Hilfe von programmierbaren Zeitgebern (Timer). Mit Zeitgebern können genau bestimmbare Zeitverzögerungen erzeugt werden. Sie sind daher auch für die Echtzeitprogrammierung wichtig. Im Gegensatz zu einfachen Betriebssystemen muss bei Echtzeitbetriebssystemen der Zeitbedarf zur Abarbeitung eines Programms genau eingehalten werden.

Software-Interrupts

Software-Interrupts sind Maschinenbefehle, welche die gleiche Wirkung haben wie durch Hardware ausgelöste Interrupts. Mit Hilfe von Software-Interrupts kann der Benutzer bestimmte Ein-/Ausgabe-Operationen aufrufen, die das Betriebssystem bereitstellt (System Calls) und die nur im System-Modus (privileged Mode oder Supervisor Mode) erlaubt sind.

Fehlerbehandlung. Sowohl Hardware- als auch Softwarefehler können zu kritischen Zuständen eines Computersystems führen, die sofort bereinigt werden müssen. Die Fehler werden durch eine geeignete Hardware erkannt. Diese löst dann einen Interrupt aus, der in jedem Fall vom Prozessor akzeptiert und unverzüglich bearbeitet wird. Typische Softwarefehler sind die Division durch Null (Divide by Zero), die Überschreitung des darstellbaren Zahlenbereichs (Overflow) und das Ausführen nicht existierender Maschinenbefehle (Illegal Instruction). Der zuletzt genannte Fehler kann z.B. auftreten, wenn fälschlicherweise in einem Unterprogramm eine unterschiedliche Zahl von PUSH- und POP-

Befehlen auf dem Stack ausgeführt wurden. Die angeführten Interrupts entstehen innerhalb des Prozessors und werden häufig als *Traps* (Falle) oder *Exceptions* bezeichnet. Zu den externen Fehlern zählen z.B. Speicherdefekte, die durch eine Paritätsprüfung erkannt werden können. Weitere Fehlerquellen dieser Art sind Einbrüche der Betriebsspannung oder die Verletzung von Zeitbedingungen bei Busprotokollen. Um solche Fehler zu erkennen, benutzt man so genannte *Watchdog*-Schaltungen.

Exceptions

Verarbeitung eines einzelnen Interrupts

Moderne Prozessoren besitzen ein kompliziertes Interrupt-System, das mehrere Interrupt-Quellen unterstützt. Wir wollen jedoch zunächst annehmen, dass es nur eine einzige Interrupt-Quelle gibt und erst im nächsten Abschnitt zu komplizierteren Interrupt-Systemen mit mehreren Interrupt-Quellen übergehen.

Um mit einem Prozessor Interrupts verarbeiten zu können, muss die Ablaufsteuerung der Holephase erweitert werden. Nur so können externe oder interne Signale auf die Befehlsverarbeitung des Prozessors Einfluß nehmen. Da eine Interrupt-Anforderung IRQ (Interrupt Request) zu jedem beliebigen Zeitpunkt während eines Befehlszyklus eintreffen kann, muss sie zunächst in einem Flipflop zwischengespeichert werden. Bis zur Annahme und Bearbeitung durch den Prozessor bleibt die Interrupt-Anforderung anhängig (pending). Die notwendige Erweiterung der Holephase ist in Abbildung 4.15 dargestellt. Dabei wurde angenommen, dass der Stack am Ende des Hauptspeichers (RAM-Bereich) beginnt. M[- -SP] bedeutet, dass der Stackpointer zuerst dekrementiert und dann zur Adressierung des Speichers benutzt wird. Der Interrupt-Opcode legt implizit die Startadresse der Service-Routine fest, die auch *Interrupt-Vektor* genannt wird. Mit dem Quittungssignal $INTA$ (Interrupt Acknowledge) bestätigt der Prozessor die Annahme eines Interrupts.

Interrupt-Vektor

Mit diesem Signal kann das IRQ-Flipflop zurückgesetzt werden. Bevor die eigentliche Service-Routine beginnt, muss jedoch das Statusregister auf den Stack gerettet werden. Damit die während der Service-Routine möglicherweise veränderten Flags sich nicht auf den Programmfluß des unterbrochenen Programms auswirken, wird der zuvor gesicherte Inhalt des Statusregisters vor dem Rücksprung ins unterbrochene Programm wieder vom Stack zurückgeholt. Die Sicherung des Statusregisters wird normalerweise automatisch durch die Ablaufsteuerung (Mikroprogramm) erledigt. Am Ende eines Interrupts steht ein *RETI-Befehl* (Return from Interrupt), um – wie bei Unterprogrammen – die Rücksprungadresse vom Stack zu holen. Da zusätzlich auch das Statusregister zurückgeschrieben wird, unterscheidet sich der RETI-Befehl von einem Unterprogramm-RETURN-Befehl. Bei älteren Prozessoren (z.B. Intel i8085) muss der Programmierer selbst die Service-Routine in PUSH- und POP-Befehle einbetten, die den Inhalt des Statusregisters für das unterbrochene Programm retten.

RETI-Befehl

Mehrere Interrupts

Wenn ein Prozessor in der Lage sein soll, mehrere Interrupt-Quellen zu berücksichtigen, müssen zwei Probleme gelöst werden:

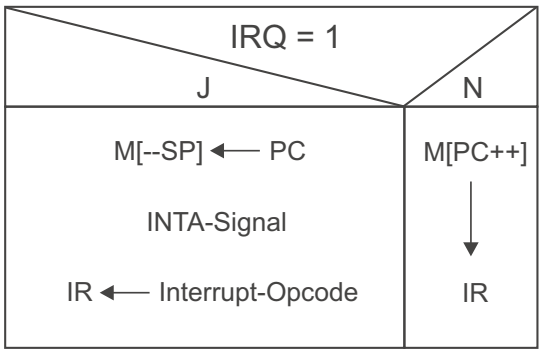


Abbildung 4.15: Erweiterung der Holephase zur Bearbeitung eines Interrupts.

1. Jedem Interrupt muss eine eigene Startadresse für die Service-Routine zugeordnet werden.

2. Wenn mehrere Interrupts gleichzeitig hängend sind, muss eine Entscheidung getroffen werden, welcher Interrupt vorrangig bearbeitet wird.

Prioritäten

Die Einführung von *Prioritäten* für die einzelnen Interrupts kann durch Hardware oder Software erfolgen. Außerdem ist es sinnvoll, zwei Kategorien von Interrupts zu unterscheiden: *maskierbare* und *nicht maskierbare* Interrupts. Die maskierbaren Interrupts können vom Programmierer freigegeben oder gesperrt werden. Während zeitkritischer Programmabschnitte ist es z.B. ratsam, alle Interrupts dieser Art zu sperren, damit keine zusätzlichen Verzögerungszeiten durch Service-Routinen entstehen. Nicht-maskierbare Interrupts (Non Maskable Interrupts, NMI) können nicht gesperrt werden. Sie dienen zur Fehlerbehandlung und müssen deshalb unverzüglich bearbeitet werden. Ein weiteres Beispiel für eine unaufschiebbare Fehlerbehandlung ist die Reaktion auf einen Abfall der Betriebsspannung, die von einer Watchdog-Schaltung ausgelöst wird. Beispiele für Prozessor-interne Ereignisse (Traps) wurden bereits beschrieben. Traps sind ebenfalls nicht maskierbar und haben daher stets die höchste Priorität.

Startadresse der Service-Routine. Um die Startadresse der Service-Routine zu bestimmen, gibt es drei Möglichkeiten:

1. Abfragemethode,

2. Vektormethode,

3. Codemethode.

Die *Abfragemethode* (Polling) hat den geringsten Hardwareaufwand, da am Prozessor weiterhin nur ein Interrupt-Eingang nötig ist. Die Interrupt-Anforderungen der einzelnen Ein-/Ausgabe-Bausteine werden durch eine OR-Funktion miteinander verknüpft, die dann diesen Eingang IRQ ansteuert. Wenn während der Holephase eine hängende Interrupt-Anforderung erkannt wird ($IRQ = 1$), unterbricht der Prozessor das laufende Programm, indem er die zu dem (einzigen) Interrupt gehörende Service-Routine aufruft. Dieses Programm verwaltet „alle“ Interrupts und wird deshalb *Interrupt Handler* genannt. Zunächst

Interrupt Handler

muss die Interrupt-Quelle ermittelt werden. Dazu werden die Statusregister der einzelnen Ein-/Ausgabe-Bausteine nacheinander gelesen und geprüft, ob das Interrupt-Flag gesetzt ist. Der Interrupt Handler kennt die Startadressen der Service-Routinen für jeden einzelnen Baustein. Sobald die Abfrage des Interruptflags positiv ist, wird durch einen unbedingten Sprung zu der (dem Ein-/Ausgabe-Baustein) entsprechenden Service-Routine verzweigt. Dieses *Device Handler*-Programm muss dann mit dem RETI-Befehl abgeschlossen sein. Die Reihenfolge der Abfrage entspricht der Priorität der angeschlossenen Ein-/Ausgabe-Bausteine. Wechselnde Prioritäten sind durch Änderung dieser Reihenfolge leicht zu realisieren. Ein großer Nachteil der Abfragemethode ist jedoch der hohe Zeitbedarf zur Abfrage der einzelnen Ein-/Ausgabe-Bausteine und Ermittlung des richtigen Device Handlers.

Die *Vektormethode* (vectored Interrupts) verursacht den größten Hardwareaufwand, da für jede mögliche Interrupt-Anforderung ein eigener Eingang am Prozessor vorhanden sein muss. Die eintreffenden Interrupt-Anforderungen werden in einem besonderen Register $ISRQ$ (Interrupt Service ReQuest) aufgefangen und mit einer Interrupt-Maske IM, die Bestandteil des Statusregisters ist, bitweise AND-verknüpft. Nehmen wir zunächst einmal an, dass sich die Interrupt-Anforderungen wechselseitig ausschließen. Wenn ein freigegebener Interrupt erkannt wird, kann die Ablaufsteuerung der betreffenden Anforderungsleitung einen Maschinenbefehl zuordnen, der *implizit* die Startadresse für eine Service-Routine festlegt. Mit der Vektormethode können Interrupt-Anforderungen sehr schnell beantwortet werden, da die Startadresse des Device Handlers spätestens nach einem Befehlszyklus bekannt ist. Nachteilig ist jedoch, dass jeder Anforderungsleitung nur ein Device Handler zugeordnet werden kann. Abhängig von ihrem momentanen Zustand benötigen Peripheriegeräte jedoch verschiedene Device Handler. Um trotzdem mit nur einer Anforderungsleitung auszukommen, bietet sich die direkte Abfrage eines Codes für die gewünschte Service-Routine an.

Die *Codemethode* wird von sehr vielen Prozessoren angeboten, da sie ein hohes Maß an Flexibilität bietet. Jeder Anforderungsleitung IRQ_i wird ein Quittungssignal $INTA_i$ zugeordnet. Wenn ein Ein-/Ausgabe-Baustein auf eine Interrupt-Anforderung hin ein Quittungssignal erhält, antwortet er mit einem Codewort auf den Datenbus[11]. Aus diesem Codewort wird dann die Startadresse des gewünschten Device Handlers bestimmt. Unterstützt der Prozessor vektorisierte Interrupts, so entspricht dem Codewort eine Vektornummer. Das Codewort kann aber auch einen CALL-Befehl darstellen, und der Ein-/Ausgabe-Baustein übergibt im Anschluss daran die Startadresse direkt an den Prozessor (vgl. Intel i8085).

Prioritäten lösen Konflikte. Wenn mehrere Interrupt-Anforderungen gleichzeitig hängend sind, liegt eine Konfliktsituation vor. Dieser Fall kann z.B. eintreten, wenn in einem Befehlszyklus zwei Interrupts gleichzeitig oder kurz hintereinander eintreffen. Welcher der beiden Interrupts wird dann zuerst bearbeitet? Oder nehmen wir an, der Prozessor sei gerade in einer Service-Routine und ein neuer Interrupt tritt auf. Soll die Service-Routine zuerst beendet werden oder wird sie selbst unterbrochen? Um solche Konfliktsituationen zu beheben, müs-

[11] Das Codewort hat meist nicht die volle Maschinenwortbreite.

sen Prioritäten definiert werden, welche die Bedeutung der einzelnen Interrupts bewerten. Wie wir bereits weiter oben gesehen haben, können Prioritäten sehr leicht durch die Abfragereihenfolge in einem Interrupt Handler implementiert werden. Die Abfragemethode ist jedoch sehr langsam.

Interrupt-Controller

Wir wollen im Folgenden den Aufbau und die Funktionsweise eines *Interrupt-Controllers* vorstellen. Eine Hardware-Lösung ist wesentlich schneller, da sie den Prozessor nicht belastet. Die Priorität einer Anforderungsleitung wird durch einen auf dem Chip integrierten Interrupt-Controller festgelegt. Jeder Interrupt-Eingang kann mit Hilfe eines externen Controllers in weitere Prioritätsebenen unterteilt werden. Sowohl interne als auch externe Interrupt-Controller haben prinzipiell den gleichen Aufbau.

In Abbildung 4.16 ist ein Interrupt-Controller für vier Prioritätsebenen dargestellt.

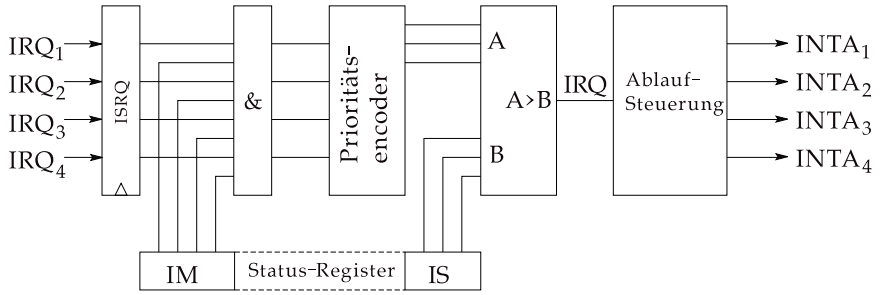


Abbildung 4.16: Aufbau eines Interrupt-Controllers für vier Prioritätsebenen.

Wenn alle Interrupts freigegeben sind ($IM = 1111$), gelangen hängende Anforderungen zu einem *Prioritätsencoder*, der die Nummer der Anforderung höchster Priorität als Dualzahl ausgibt. Für den Fall, dass keine Anforderung hängend ist, wird eine Null ausgegeben. Deshalb ist auch ein 3-Bit-Code notwendig. Nehmen wir an, dass IRQ_4 die höchste Priorität hat, so ergibt sich folgende Funktionstabelle für den Prioritätsencoder[12]:

IRQ_4	IRQ_3	IRQ_2	IRQ_1	Code
1	×	×	×	100
0	1	×	×	011
0	0	1	×	010
0	0	0	1	001
0	0	0	0	000

Der vom Prioritätsencoder ausgegebene Code wird mit den Interrupt-Statusbits IS des Statusregisters verglichen. Diese Bits codieren die Prioritätsebene, in der sich der Prozessor gerade befindet. Ist der vorliegende Prioritäts-Code kleiner als der IS-Code oder gleich groß, so wird keine Interrupt-Anforderung an die Ablaufsteuerung weitergeleitet. Alle vorliegenden Interrupt-Anforderungen

[12] × steht für *don't care*.

bleiben hängend. Falls der am Vergleicher-Eingang A anliegende Prioritäts-Code größer als der IS-Code ist, wird die Anforderung mit der höchsten Priorität akzeptiert und wie folgt bearbeitet:

1. Mit Hilfe des Prioritäts-Codes wird die Startadresse der Service-Routine nach einer der oben genannten Methoden bestimmt und ein $INTA$-Signal wird ausgegeben.

2. Die Rücksprungadresse wird zusammen mit dem alten IS-Code auf den Stack gerettet.

3. Die Startadresse der Service-Routine wird in den Programmzähler geladen und der IS-Teil des Statusregisters wird mit dem aktuellen Prioritätscode[13] überschrieben.

4. Nun kann die Abarbeitung der Service-Routine erfolgen. Sie wird mit einem RETI-Befehl abgeschlossen. Da dieser Befehl das alte Statusregister wiederherstellt, bewirkt er eine Rückkehr in die unterbrochene Prioritätsebene.

Mit dem beschriebenen Interrupt-Controller können beliebig ineinander verschachtelte Interrupts verarbeitet werden. Dies wollen wir an einem konkreten Beispiel noch einmal verdeutlichen.

Beispiel. Betrachten wir zwei Peripheriegeräte, die unterschiedliche Geschwindigkeitsanforderungen an eine Service-Routine stellen. Das Lesen eines Sektors bei einer Festplatte erfordert die sofortige Reaktion des Prozessors, sobald sich die gewünschte Spur unter dem Schreib-/Lesekopf befindet. Die zugehörige Service-Routine *HD* (Hard Disk) darf nicht unterbrochen werden, da sonst Daten verloren gehen. Wir ordnen deshalb der Festplatte die Interrupt-Anforderung höchster Priorität, hier IRQ_4, zu. Als zweites Peripheriegerät soll ein Drucker betrachtet werden, der zeilen- oder blockweise Daten empfängt und seine Bereitschaft zur Aufnahme neuer Daten durch einen Interrupt signalisiert. Die Service-Routine LP (Line Printer) kann jederzeit unterbrochen werden, da sich dadurch lediglich die Zeit zum Ausdrucken verlängert. Der Ein-/Ausgabe-Baustein zum Anschluss des Druckers wird daher mit der Anforderungsleitung niedrigster Priorität, hier IRQ_1, verbunden. Wir wollen annehmen, dass während der Abarbeitung des Hauptprogramms je ein Prozess zum Drucken und zur Datei-Eingabe aktiv sind und vom Betriebssystem blockiert wurden, d.h. diese Prozesse warten auf Interrupts der Peripheriegeräte. Wenn der Drucker über IRQ_1 Bereitschaft zum Drucken signalisiert, kann der Prozessor zum Device Handler LP verzweigen. Die Rücksprungadresse und das Statusregister werden auf den Stack gerettet und der IS-Code auf 001 gesetzt. Trifft während der Bearbeitung von LP ein IRQ_4 ein, so wird LP unterbrochen, da der aktuelle Interrupt-Prioritätscode 100 größer ist als der IS-Code. Nach der Verzweigung zum Programm HD ist der IS-Code 100. Der Device Handler der Festplatte kann demnach nicht unterbrochen werden, da alle anderen Interrupts niedrigere Prioritätscodes haben[14]. Sobald das Programm HD abgeschlossen ist,

[13] Vom Eingang A des Vergleichers.

[14] Ausgenommen NMIs.

setzt der Prozessor die Bearbeitung des Device Handlers LP fort. Nach dem Rücksprung aus HD ist der IS-Code wieder 001. Man beachte, dass beide Interrupts durch die Interrupt-Maske IM freigegeben sein müssen. Um Fehler zu vermeiden, dürfen Änderungen der Interrupt-Maske nur vom Betriebssystem vorgenommen werden. Die meisten Prozessoren bieten daher hardwaremäßig die zwei Betriebsarten *User* und *Supervisor Mode*. Die Interrupt-Maske kann nur im Supervisor Mode verändert werden.

Die oben beschriebene Hardware eines Interrupt-Controllers ist i.Allg. auf dem Prozessorchip integriert. Zusätzliche Prioritätsebenen können durch die Abfragemethode (Software), externe Interrupt-Controller oder durch eine Verkettung der $INTA$-Signale einer Ebene erreicht werden (Daisy-Chain). Die letztgenannte Methode ist besonders häufig, da sie nur geringen Mehraufwand an Hardware verursacht. Sie wird meist mit der Codemethode zur Ermittlung der Startadresse kombiniert. Die Daisy-Chain-Priorisierung führt eine *ortsabhängige* Prioritätenfolge innerhalb einer Ebene ein, indem das $INTA$-Signal vom Prozessor über hintereinandergeschaltete Busmodule weitergereicht wird. Jedes Busmodul stellt eine mögliche Interrupt-Quelle dar, die über eine Sammelleitung eine Interrupt-Anforderung erzeugen kann. Das erste Busmodul, das einen Interrupt-Service wünscht, antwortet dem Prozessor mit einem Codewort zur Bestimmung der Startadresse der Service-Routine, sobald es ein $INTA$-Signal registriert. In diesem Fall reicht das betreffende Busmodul das $INTA$-Signal nicht an nachfolgende Busmodule weiter. Somit hat das direkt auf das Prozessormodul folgende Busmodul die höchste Priorität.

4.12 Rechenwerk

Wie schon gesagt, besteht das Rechenwerk im Wesentlichen aus Registern, Multiplexern und einer ALU. Im Folgenden sollen diese Bestandteile näher betrachtet werden.

4.12.1 Daten- und Adressregister

Registerarchitektur

Wenn im Rechenwerk adressierbare Datenregister vorhanden sind, spricht man von einer *Registerarchitektur*. Die Datenregister bilden einen *Register-Block* oder ein *Register File*[15]. Bei Registerarchitekturen unterscheidet man Ein-, Zwei- und Drei-Adress-Maschinen. Der Adresstyp wird von der Zahl der internen Daten- und Adressbusse bestimmt, die den Registerblock mit der ALU und dem Speicher verbinden. Je mehr Datenregister gleichzeitig ausgewählt werden können, umso weniger Taktzyklen werden zur Ausführung eines Maschinenbefehls benötigt.

Oft wird der Begriff X-Adress-Maschine auch auf das Format eines Maschinenbefehls bezogen. Da die Zahl der Adressen, die in einem Befehl angegeben werden können, keine Auskunft über die Hardware des Rechenwerks (Maschine) gibt, sollte man in diesem Zusammenhang besser von einem X-Adress-*Befehlssatz* sprechen.

[15] Wenn sehr viele Register vorhanden sind (z.B. bei RISC-Prozessoren).

Neben Registerarchitekturen gibt es die *Stackarchitekturen* oder *Null-Adress-Maschinen*. Anstelle eines Register Files wird hier ein Stack benutzt, um Operanden oder Ergebnisse zu speichern. Eine echte Stackarchitektur verfügt über einen auf dem Prozessor integrierten LIFO-Speicher. Die Realisierung des Stacks im Hauptspeicher (mit Hilfe eines Stackpointers) ist nicht empfehlenswert, da die hohe Zahl von Speicherzugriffen die Prozessorleistung herabsetzt. Null-Adress-Befehle enthalten keine direkten Adressangaben, sondern benutzen die Daten, die oben auf dem Stack liegen, als Operanden oder als Adressen (für Zugriffe auf den Hauptspeicher). Diese Daten werden kurzzeitig in Latches zwischengespeichert und durch die ALU miteinander verknüpft. Das Ergebnis wird dann wieder auf den Top of Stack (TOS) gelegt. Diese Vorgehensweise entspricht der Berechnung eines arithmetischen Ausdrucks in der *umgekehrten Polnischen Notation* (Reverse Polnish Notation, RPN). Erst müssen mit PUSH-Befehlen die Operanden auf den Stack gebracht werden und danach wird die gewünschte Operation angegeben. Da die Adresse fehlt, sind Maschinenbefehle von Stackarchitekturen sehr kompakt. Zur Lösung einer bestimmten Aufgabe müssen jedoch viele Maschinenbefehle verwendet werden. Obwohl es stackorientierte Programmiersprachen (z.B. FORTH) gibt, die durch entsprechende Stackmaschinen unterstützt werden, hat sich die Stackarchitektur nicht durchsetzen können. Der überwiegende Anteil heutiger Prozessoren gehört zur Klasse der Registerarchitekturen.

Stackarchitektur

4.12.2 Datenpfade

In Abbildung 4.17 ist ein Rechenwerk dargestellt, das ALU und Registerfeld nur mit *einem* internen (bidirektionalen) Datenbus verbindet. Nachteilig ist an dieser Datenpfadstruktur, dass die Abarbeitung eines Maschinenbefehls drei Taktzyklen erfordert: Zunächst müssen die beiden Operanden in zwei Hilfs-Register gebracht werden. Da immer nur ein Operand pro Takt aus dem Register File gelesen werden kann, benötigt man hierzu zwei Taktzyklen. Im dritten Taktzyklus wird das Ergebnis der Verknüpfung vom F(unction)-Ausgang der ALU ins Register File geschrieben.

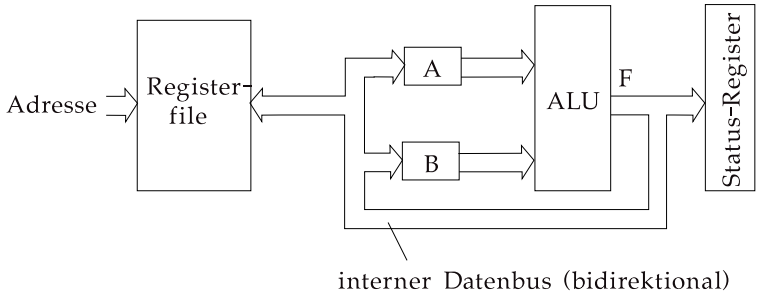


Abbildung 4.17: Rechenwerk einer Ein-Adress-Maschine.

Beispiel. Angenommen, der Drei-Adress-*Befehl SUB R1,R2,R3* soll auf einer Ein-Adress-*Maschine* ausgeführt werden. Die Realisierung dieses Befehls erfordert die folgenden Mikrooperationen:

Taktzyklus	Operation
1	$R1 \rightarrow A$
2	$R2 \rightarrow B$
3	$F \rightarrow R3$

Bei einer Zwei-Adress-Maschine kann der gleiche Befehl in zwei Taktzyklen ausgeführt werden, da die Latches A und B gleichzeitig geladen werden. In Abbildung 4.18 wird die Datenpfad-Struktur einer Drei-Adress-Maschine dargestellt, die typisch ist für RISC-Prozessoren. Der angeführte Beispiel-Befehl kann mit einem solchen Rechenwerk in einem Taktzyklus ausgeführt werden, da sowohl für die Operanden als auch für das Ergebnis eigene Datenbusse bereitstehen, die separat adressierbar sind.

RALU

Das Rechenwerk einer Drei-Adress-Maschine wird auch als Register-ALU oder kurz $RALU$ bezeichnet. Auf der Kursseite der LVU (Lernraum Virtuelle Universität) finden Sie eine gleichnamige Software, die eine 16-Bit RALU simuliert. Wir möchten Sie ermutigen, die in der Dokumentation angegebenen Beispiele selbst zu erproben und so mehr über die Mikroprogrammierung einer RALU zu lernen.

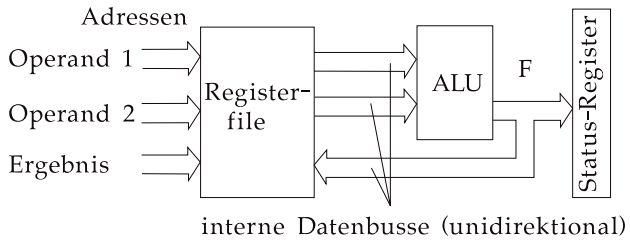


Abbildung 4.18: Rechenwerk einer Drei-Adress-Maschine.

4.12.3 Schiebemultiplexer

Schiebeoperationen sind z.B. für die Multiplikation oder Division nützlich. Sie können durch einen Schiebemultiplexer (Shifter) an einem Eingang der ALU realisiert werden. Der Schiebemultiplexer besitzt drei Eingänge mit Maschinenwortbreite (Abbildung 4.19). Ein Eingang ist ganz normal mit dem internen Datenbus verbunden. Bei den beiden anderen Eingängen sind die Datenleitungen einerseits um eine Stelle nach links, andererseits um eine Stelle nach rechts verschoben. Über zwei Steuerleitungen wird die gewünschte Positionierung eines ALU-Operanden ausgewählt. Wenn der Operand verschoben wird, muss von links oder rechts ein Bit nachgeschoben werden (Eingänge: Left Input, LI, und Right Input, RI). Das herausfallende Bit steht je nach Schieberichtung an den Ausgängen (Left Output, LO, bzw. Right Output, RO). Oft wird bei Schiebeoperationen das Carry Flag benutzt, um entweder das herausfallende Bit zwischenzuspeichern oder um den Inhalt des Carry Flags nachzuschieben.

Wenn die Schiebe-Eingänge mit den zugehörigen Ausgängen verbunden werden[16], *rotieren* die Bits um eine Stelle nach links oder rechts. Mit mehreren Rotationsoperationen können Bitgruppen in jede gewünschte Position gebracht werden. Oft ist es auch möglich, über das Carry Flag zu rotieren, d.h. der Schiebe-Ausgang wird mit dem Eingang des Carry Flags verbunden und dessen Ausgang mit dem Schiebe-Eingang.

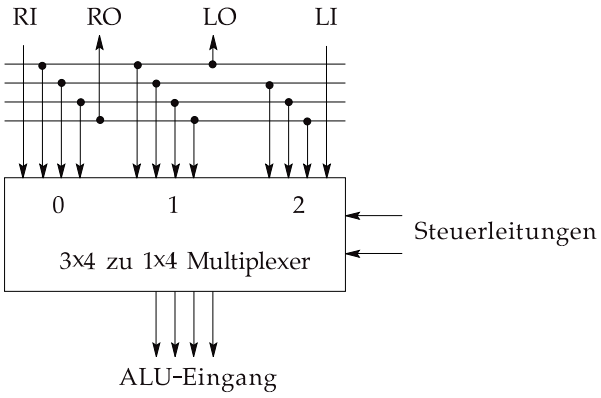


Abbildung 4.19: 4-Bit-Schiebemultiplexer (Shifter) vor einem ALU-Eingang.

4.12.4 Logische Operationen

In der zweiten Kurseinheit haben wir schon Schaltungen für arithmetische Operationen kennengelernt. Wir wollen im Folgenden kurz auf logische Operationen eingehen. Da bei logischen Operationen jede Stelle eines Maschinenworts unabhängig von den anderen Stellen verarbeitet werden kann, ist die Entwicklung entsprechender Schaltnetze unkritisch. Die Verzögerungszeiten logischer Operationen sind im Vergleich zu arithmetischen Operationen vernachlässigbar und haben folglich keinen Einfluß auf die Taktrate des Prozessors. Mit zwei Variablen x_i und y_i können 16 verschiedene logische Verknüpfungen gebildet werden. Erzeugt man alle vier Minterme und ordnet jedem Minterm eine Steuervariable zu, so können mit dem Steuerwort alle logischen Operationen ausgewählt werden.

$$z_i = s_3 x_i y_i \vee s_2 x_i \overline{y_i} \vee s_1 \overline{x_i} y_i \vee s_0 \overline{x_i}\,\overline{y_i}$$

Als Beispiele sollen die Steuerwörter für die drei Boole'schen Grundoperationen angegeben werden:

1. Negation

   $$z_i \;=\; \overline{x}_i = \overline{x}_i(y_i \vee \overline{y}_i) = \overline{x}_i y_i \vee \overline{x}_i \overline{y}_i \;\Rightarrow\; s_3 s_2 s_1 s_0 = 0011$$

2. Disjunktion

[16] $LO \to LI$ bzw. $RO \to RI$.

$$\begin{aligned} z_i &= x_i \vee y_i \\ &= x_i(y_i \vee \overline{y}_i) \vee (x_i \vee \overline{x}_i)y_i \\ &= x_i y_i \vee x_i\overline{y}_i \vee x_i y_i \vee \overline{x}_i y_i \;\Rightarrow\; s_3 s_2 s_1 s_0 = 1110 \end{aligned}$$

3. Konjunktion

$$z_i = x_i y_i \;\Rightarrow\; s_3 s_2 s_1 s_0 = 1000$$

Durch eine weitere Steuervariable s_4 kann man auch die Überträge aus einem Carry Lookahead Generator miteinbeziehen und so zwischen arithmetischen und logischen Operationen umschalten.

Die inverse EXOR-Funktion ($\overline{\oplus}$) wird *Äquivalenz*-Funktion genannt und mit dem Symbol $\equiv$ notiert. Für jede Stelle wird die folgende Funktion gebildet:

$$f_i = z_i \equiv (s_4 \vee c_i)$$

Für $s_4 = 1$ gilt dann

$f_i = z_i \equiv 1 = z_i$ $\Rightarrow$ logische Operationen wie oben

Für $s_4 = 0$ folgt

$f_i = z_i \equiv c_i$ $\Rightarrow$ arithmetische Operationen

Zur Addition zweier N-stelliger Dualzahlen berechnet sich die Stellensumme wie folgt:

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus c_i \\ &= x_i \equiv y_i \equiv c_i \\ &= (x_i y_i \vee \overline{x}_i\overline{y}_i) \equiv c_i \\ & \quad i = 0 \ldots N-1,\; c_0 = 0 \end{aligned}$$

Die Addition wird demnach mit dem Steuerwort $s_4 s_3 s_2 s_1 s_0 = 01001$ ausgewählt, d.h. $f_i = s_i$.

Die Subtraktion kann auf die Addition des Zweierkomplements des Subtrahenden zurückgeführt werden. Dieses kann durch stellenweise Invertierung und Addition von 1 gebildet werden. Daraus folgt für die Stellendifferenz in der i-ten Stelle:

$$\begin{aligned} d_i &= x_i \equiv \overline{y}_i \equiv c_i \\ &= (x_i\overline{y}_i \vee \overline{x}_i y_i) \equiv c_i \\ & \quad i = 0 \ldots N-1,\; c_0 = 1 \end{aligned}$$

Die Subtraktion wird mit dem Steuerwort $s_4 s_3 s_2 s_1 s_0 = 00110$ ausgewählt, d.h. $f_i = d_i$. Mit dem beschriebenen Steuerprinzip können 16 logische und 32 arithmetische Operationen ($c_0 = 1/0$) realisiert werden. Dabei entstehen auch „exotische“ Funktionen, die selten oder gar nicht benutzt werden.

Nun wollen wir zeigen, wie die wichtigsten Status-Flags bestimmt werden.

4.12.5 Status-Flags

Die Ausgänge $f_{N-1}, \ldots, f_0$ bilden den Ergebnisbus der ALU (vgl. Abbildung 4.17 und Abbildung 4.18). Aus der Belegung dieser Bits und der Belegung des Übertrags c_N werden die Status-Flags gebildet. Die folgenden Flags sind bei fast allen Prozessoren zu finden, da sie sehr einfach bestimmt werden können:

1. *Carry C:* Übertrag in der höchsten Stelle, $C = c_N$

2. *Zero Z:* alle Bits von F sind Null, $Z = \overline{f_{N-1} \vee f_{N-2} \vee \ldots \vee f_0}$

3. *Minus M:* negatives Vorzeichen bei Zweierkomplement-Darstellung, $M = f_{N-1}$

4. *Overflow (bzw. Underflow) V:* das Ergebnis ist zu groß (klein), um mit N-Bit-Wortbreite dargestellt zu werden, $V = c_N \oplus c_{N-1}$

Die Entstehung eines Overflows (bzw. Underflows) soll an einem Beispiel erläutert werden. Wir betrachten dazu die Addition von 2- Bit-Zahlen in der Zweierkomplement-Darstellung (vgl. Abbildung 4.20). Ein Overflow (bzw. Underflow) liegt vor, wenn bei der Addition die Grenze von +1 nach -2 (bzw. -2 nach +1) überschritten wird. Die Erkennung einer solchen Fehlersituation ist äußerst wichtig, da sonst falsche Ergebnisse zurück geliefert werden. Sie kann (mit Hilfe eines Interrupt Handlers) korrigiert werden kann, indem die Berechnung nochmal mit doppelter Wortbreite ausgeführt wird.

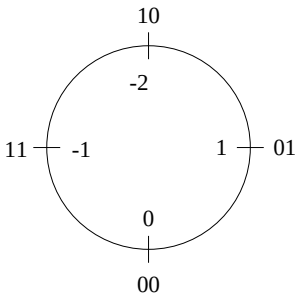


Abbildung 4.20: Zweierkomplement-Darstellung einer 2-Bit-Zahl; innen: Dezimalwert, außen: duale Codierung.

Bei folgender Addition entsteht ein Overflow:

$$1 + 1 \rightarrow -2 \qquad \begin{array}{ccc} & 0 & 1 \\ & 0 & 1 \\ \hline 0 & 1 & \\ \hline 0 & 1 & 0 \end{array} \qquad \text{mit} \qquad \begin{array}{ccc} & x_1 & x_0 \\ & y_1 & y_0 \\ \hline c_2 & c_1 & c_0 \\ \hline s_2 & s_1 & s_0 \end{array}$$

Ein Underflow entsteht bei folgenden Subtraktionen (durch Addition des Zweierkomplements):

$$(-1) + (-2) \rightarrow 1 \qquad \begin{array}{ccc} & 1 & 1 \\ & 1 & 0 \\ \hline 1 & 0 & \\ \hline 1 & 0 & 1 \end{array}$$

$$(-2) + (-2) \rightarrow 0 \qquad \begin{array}{ccc} & 1 & 0 \\ & 1 & 0 \\ \hline 1 & 0 & \\ \hline 1 & 0 & 0 \end{array}$$

Im Folgenden soll die oben angegebene Bedingung zur Erkennung eines Overflows hergeleitet werden. Bei der Zweierkomplement-Darstellung zeigt das höchstwertige Bit x_{N-1} einer N-stelligen Zahl das Vorzeichen an:

$x_{N-1} = 0$ positiver Wert (einschließlich 0)
$x_{N-1} = 1$ negativer Wert

Wenn zwei positive (negative) Summanden addiert werden und ein (kein) Übertrag aus der Stelle $N-2$ besteht, d.h. $c_{N-1} = 1$ ($c_{N-1} = 0$), so muss das Ergebnis auf jeden Fall positiv (negativ) sein.

Tabelle 4.1: Zur Bestimmung der Schaltfunktion V.

x_{N-1}	y_{N-1}	c_{N-1}	s_{N-1}	V
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

Betrachtet man nun die Funktionstabelle des Volladdierers in der höchstwertigen Stelle (Tabelle 4.1), so sieht man, dass diese beiden Fälle vom Volladdierer *falsch* ausgewertet werden. Sie müssen durch zusätzliche Hardware als Overflow V erkannt werden[17]. Aus Tabelle 4.1 erhalten wir folgende Schaltfunktion:

$$V = \overline{x}_{N-1}\overline{y}_{N-1}c_{N-1} \vee x_{N-1}y_{N-1}\overline{c}_{N-1}$$

Die rechte Seite kann wie folgt umgeschrieben werden:

$$= (\overline{x_{N-1} \vee y_{N-1}} \vee \overline{c}_{N-1})\overline{x_{N-1}y_{N-1}}c_{N-1} \vee$$

$$(x_{N-1}y_{N-1} \vee (x_{N-1} \vee y_{N-1})c_{N-1})\overline{c}_{N-1}$$

Mit

$$c_N = x_{N-1}y_{N-1} \vee c_{N-1}(x_{N-1} \vee y_{N-1})$$

bzw.

$$\overline{c}_N = \overline{x_{N-1}y_{N-1}}(\overline{c}_{N-1} \vee \overline{x_{N-1} \vee y_{N-1}})$$

ergibt sich schließlich

$$V = \overline{c}_N c_{N-1} \vee c_N \overline{c}_{N-1} = c_N \oplus c_{N-1}$$

Das gleiche Ergebnis erhält man, wenn man in der Tabelle 4.1 c_N ergänzt und damit V in Abhängigkeit von c_N und c_{N-1} bestimmt.

[17] Als Alternative könnte man auch ein spezielles Schaltnetz zur Addition der höchstwertigen Stelle konstruieren.

4.13 Leitwerk

Das Leitwerk hat die Aufgabe, Maschinenbefehle (Makrobefehle) aus dem Hauptspeicher ins Befehlsregister zu laden (Holephase) und anschließend zu interpretieren (Ausführungsphase). Ein Makrobefehl wird in eine Folge von Steuerwörtern (Mikrobefehle) für Rechenwerk, Speicher und Ein-/Ausgabe umgesetzt. Diese Steuerwörter werden durch eine Ablaufsteuerung erzeugt, die entweder festverdrahtet ist oder als Mikroprogramm-Steuerwerk aufgebaut wird. CISC-Prozessoren verfügen über einen sehr umfangreichen Befehlssatz, der nur mit Hilfe der Mikroprogrammierung implementiert werden kann. RISC-Prozessoren kommen dagegen mit einem festverdrahteten Leitwerk aus, das oft nur aus einem Decodierschaltnetz für den Opcode besteht.

4.13.1 Mikroprogrammierung

Mikroprogramm-Steuerwerke können leichter entwickelt und gewartet werden als festverdrahtete Steuerwerke. Sie sind aber auch wesentlich langsamer als diese, da sie die Mikrobefehle erst aus dem Steuerwort-Speicher holen müssen. Um die Chipfläche und die Kosten eines Prozessors zu minimieren, muss die Speicherkapazität des Steuerwort-Speichers optimal ausgenutzt werden. Die dabei angewandten Techniken werden im Folgenden behandelt. Der Befehlssatz eines Prozessors und die Fähigkeiten des Rechenwerks werden durch Mikroprogramme miteinander verknüpft. Zu jedem Makrobefehl gibt es einen Bereich im Steuerwort-Speicher, der die zugehörigen Mikrobefehle enthält und den man als *Mikroprogramm* bezeichnet. Der Befehlssatz eines Prozessors wird durch die Menge sämtlicher Mikroprogramme definiert. Die meisten Mikroprozessoren haben einen festen Befehlssatz, d.h. der Anwender kann die Mikroprogramme (Firmware) nicht verändern. Mikroprogrammierbare Rechner verfügen über RAM-Speicher zur Aufnahme der Mikroprogramme. Die Mikroprogrammierung bietet viele Möglichkeiten bei der Rechnerentwicklung und Anwendung, wie z.B. die Emulation anderer Rechner. Dies ist ein weiterer Vorteil von Mikroprogramm-Steuerwerken gegenüber festverdrahteten Steuerwerken. Ein Mikroprogramm-Steuerwerk ist ein Hardware-Interpreter für den Maschinenbefehlssatz. Um Mikroprogramme zu entwickeln, gibt es symbolische Sprachen, die mit Assemblern vergleichbar sind. Man nennt sie daher auch *Mikroassembler*. Sie werden gebraucht, um symbolische Mikroprogramme in Steuerspeicher-Inhalte zu übersetzen. Der „Mikro-“Programmierer braucht detaillierte Kenntnisse über den Hardware-Aufbau des Prozessors.

Die grundlegende Struktur eines Mikroprogramm-Steuerwerks haben wir bereits in Kurseinheit 3 kennengelernt. In Abbildung 4.21 wurde das in Abbildung 3.40 dargestellte Mikroprogramm-Steuerwerk so erweitert, dass es als Leitwerk verwendet werden kann.

Der Opcode wird im Befehlsregister abgelegt und durch ein Schaltnetz in die Startadresse des zugehörigen Mikroprogramms umgeformt. Der Steuerwort-Speicher wird über das Control Memory Address Register ($CMAR$) adressiert, das auf Mikroprogramm-Ebene die gleiche Funktion hat wie der Programmzähler für Makroprogramme. Dieser *Mikroprogrammzähler* wird zu Beginn mit

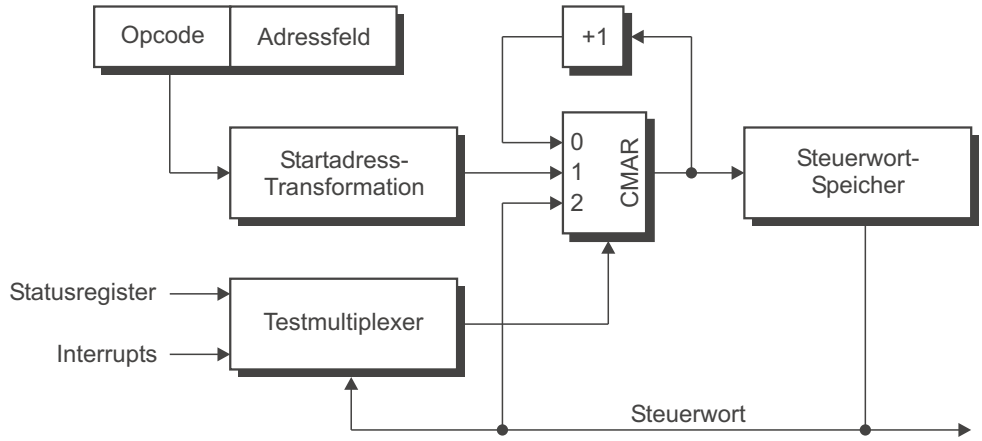


Abbildung 4.21: Aufbau eines reagierenden Mikroprogramm-Steuerwerks zur Ablaufsteuerung.

der Startadresse geladen und bei einem linearen Mikroprogramm mit jedem Taktzyklus um eins inkrementiert. Am Ende *jedes* Mikroprogramms muss das Befehlsregister mit dem nächsten Maschinenbefehl geladen werden und der beschriebene Ablauf wiederholt sich. Das Mikroprogramm für diese Holephase wird somit am häufigsten durchlaufen.

4.13.2 Mikrobefehlsformat

Mikrobefehle enthalten nicht nur Steuerbits, sondern auch Information zur Adresserzeugung für den Mikrobefehlszähler. Da während eines Mikroprogramms Interrupts und Status-Flags zu Verzweigungen führen können, benötigt man ein *reagierendes* Mikroprogramm-Steuerwerk. Ein Mikrobefehl setzt sich im Wesentlichen aus zwei Teilen zusammen:

1. einem Steuerwort zur Auswahl der Operationen im Rechenwerk,
2. einem *Adressauswahlwort*, um die Adresse des nächsten Mikrobefehls festzulegen.

Der überwiegende Teil des Steuerworts wird zur Steuerung von Mikrooperationen im Rechenwerk benötigt. Wenn mehrere Funktionseinheiten im Rechenwerk vorhanden sind, können auch mehrere Mikrooperationen gleichzeitig ausgeführt werden. Es werden dann auch weniger Mikroprogramm-Schritte zur Ausführung eines Befehls gebraucht. Andererseits erhöht die Zahl der parallelen Mikrooperationen auch die Zahl der Steuer-Bits bzw. den Hardwareaufwand des Rechenwerks.

In der Praxis arbeitet man mit codierten Mikrobefehlen, da nur ein geringer Prozentsatz der theoretisch möglichen Steuerwörter sinnvolle Mikrooperationen bewirkt. Wenn anstelle von Multiplexern Schalter (z.B. CMOS Transmission Gates) oder Bustreiber vor den Registern benutzt werden, können bei uncodierter Ansteuerung Datenpfade geschaltet werden, die auf dasselbe Register führen. Dabei würden sich die Daten gegenseitig verfälschen. Um den Aufwand

zur Decodierung gering zu halten, unterteilt man das Steuerwort in mehrere voneinander unabhängige Felder. Jedes Steuerfeld codiert eine Mikrooperation, die gleichzeitig zu den Mikrooperationen anderer Steuerfelder ausgeführt werden kann. Man beachte, dass mit der Decodierung eine Zeitverzögerung verbunden ist, die sich zur Zugriffszeit des Steuerwort-Speichers addiert.

Allgemein können wir zwei grundlegende Mikrobefehlsformate unterscheiden: *horizontale* und *vertikale* Mikrobefehle. Horizontale Mikrobefehle sind gekennzeichnet durch viele Steuer-Bits und eine hohe Zahl paralleler Mikrooperationen. Vertikale Mikrobefehle benutzen nur ein Steuerfeld und haben einen hohen Decodierungsaufwand. Es wird in jedem Taktzyklus immer nur eine einzige Mikrooperation ausgeführt. Die genannten Mikrobefehlsformate findet man in der Praxis selten in ihrer Reinform. Sie liegen meist in einer Mischform vor. Wenn z.B. ein zweiter Speicher zur Decodierung langer horizontaler Steuerwörter benutzt wird, so spricht man von *Nanoprogrammierung*.

4.13.3 Adresserzeugung

Man kann drei Arten von Mikroprogrammadressen unterscheiden:

- Startadressen,
- Folgeadressen,
- unbedingte und bedingte Verzweigungsadressen.

Der Opcode kann in der Regel nicht direkt als *Startadresse* benutzt werden. Da die Mikroprogramme unterschiedlich lang sind, würden große Teile des Steuerspeichers brachliegen. Zum anderen wäre man bei der Wahl des Opcode-Formates zu sehr eingeengt und alle Opcodes müßten die gleiche Länge haben. Um diese Schwierigkeiten zu umgehen, benutzt man einen Festwertspeicher oder ein PLA (Programmable Logic Array) zur Bestimmung der zugehörigen Mikroprogramm-Startadresse. Bei einem typischen CISC-Prozessor mit 16-Bit-Befehlsformat variiert die Opcode-Länge von 4 – 16 Bit. Der Steuerwort-Speicher enthält jedoch selten mehr als $2^{12} = 4096$ Mikrobefehle. Der Mikroprogrammzähler muss also 12 Bit lang sein. Im Gegensatz zum ROM brauchen die Startadressen für „kurze“ Opcodes nicht mehrfach programmiert zu werden. Die überflüssigen Bit-Positionen erhalten beim PLA einfach den Wert X (don't care), d.h. die entsprechenden Eingangsleitungen werden nicht in die Bildung der Produktterme einbezogen. Beim Start eines neuen Mikroprogramms wird das *CMAR* mit den PLA-Ausgängen geladen. Ein Eingangsmultiplexer sorgt dafür, dass das *CMAR* auch von anderen Adressquellen geladen werden kann.

Folgeadressen können entweder aus einem besonderen Adressteil des Mikrobefehls oder aus dem momentanen $CMAR$-Wert gewonnen werden. Die letzte Möglichkeit ist effizienter, da sie Speicherplatz einspart. Am einfachsten wird die Folgeadresse mit einem Inkrementier-Schaltnetz bestimmt, das die Ausgänge des *CMAR* über einen Eingangsmultiplexer auf dessen Eingänge zurückkoppelt und dabei den Zählerstand um eins erhöht.

Verzweigungsadressen können während der Abfrage des Testmultiplexers aus einem Teil des Steuerworts gebildet werden. Durch diese kurzzeitige Zweckentfremdung des Steuerworts kann wertvoller Speicherplatz eingespart werden. Ein zusätzliches Steuerbit muss während eines solchen Mikrobefehls die angeschlossenen Rechenwerke abkoppeln. Bei unbedingten Verzweigungen schaltet das Adressauswahlwort den Eingangsmultiplexer des *CMAR* einfach auf die betreffenden Steuerleitungen um. Bei bedingten Verzweigungen wird über einen Testmultiplexer und ein entsprechendes Adressauswahlwort ein bestimmtes Status-Flag abgefragt. Ist die gewählte Bedingung erfüllt, so wird das *CMAR* über die Steuerleitungen mit der Verzweigungsadresse geladen.

Im anderen Fall wird das Mikroprogramm mit der Folgeadresse fortgesetzt. Der *CMAR*-Eingangsmultiplexer muss durch das Ausgangssignal des Testmultiplexers und einer geeigneten Steuerlogik zwischen den zwei genannten Adressquellen umgeschaltet werden.

Zwei weitere Möglichkeiten zur Erzeugung von Verzweigungsadressen sollen hier nur angedeutet werden:

- Relative Mikroprogrammsprünge erreicht man durch Addition eines vorzeichenbehafteten Offsets (z.B. von den Bits des Mikroprogrammspeichers) zur momentanen Mikrobefehlsadresse.
- Bestimmte Status-Flags oder Interruptleitungen können feste Verzweigungsadressen erzeugen (vgl. vektorisierte Interrupts).

Die vorangehende Beschreibung eines reagierenden Mikroprogramm-Steuerwerks bezieht sich auf Leitwerke von CISC-Prozessoren. Bei RISC-Prozessoren findet lediglich eine Umcodierung des Opcodes mit einem Schaltnetz statt, d.h. es wird keine *zentrale* Ablaufsteuerung benötigt. Da RISC-Prozessoren nach dem Pipeline-Prinzip arbeiten, kann man hier von einer *verteilten* Ablaufsteuerung sprechen.

4.14 Speicher

Jeder Computer enthält verschiedenartige Speicher, um Befehle und Daten für den Prozessor bereitzuhalten. Diese Speicher unterscheiden sich bezüglich Speicherkapazität, Zugriffszeit und Kosten. Es wäre wünschenswert, dass der Prozessor immer mit seiner maximalen Taktrate arbeitet. Leider sind entsprechend schnelle Speicher teuer und haben eine vergleichsweise geringe Speicherkapazität. Deshalb verwendet man in Computersystemen verschiedene Speicher, die nach unterschiedlichen physikalischen Prinzipien arbeiten.

Preiswerte Speicher haben zwar eine hohe Speicherkapazität, sind aber relativ langsam. Durch eine hierarchische Anordnung unterschiedlicher Speicherarten versucht man, dieses Problem zu lösen. Ziel ist eine hohe Speicherkapazität bei niedrigen Kosten und einer möglichst hohen Zugriffsrate für den Prozessor. Dazu wird die niedrige Zugriffsrate des langsamsten Speichers durch zwischengeschaltete Stufen schnellerer Speicherarten an die hohe Geschwindigkeit des Prozessors angepasst. Die einzelnen Stufen enthalten jeweils Kopien kleinerer Speicherbereiche der nächsthöheren Hierarchiestufe. Der Austausch

zwischen den Stufen wird durch eine spezielle Hardware (*Memory Management Unit*, *MMU*) und durch das Betriebssystem gesteuert.

MMU

Bei heutigen Computersystemen verwendet man eine Hierarchie mit dreistufigen *Cache*-Speichern, einem Haupt- und Hintergrundspeicher (Festplatte). Der Prozessor kommuniziert direkt mit dem L_1-Cache (First Level Cache), der wiederum über den L_2- und L_3-Cache blockweise Daten mit dem Hauptspeicher austauscht. Programmteile oder Daten, die momentan nicht benötigt werden, befinden sich auf dem Hintergrundspeicher und werden bei Bedarf in den Hauptspeicher geladen. Ein Cache enthält Kopien häufig benutzter Speicherblöcke (typische Blockgrößen reichen von 16 bis 128 Byte) des nächsten Speichers in der Speicherhierarchie. Ist die Kopie vorhanden (Treffer, Hit), ist der Zugriff schnell (1 Takt für on-chip L_1-Cache), ansonsten muss der Cache die Kopie beim nächsten Speicher anfordern (Fehlzugriff, Miss), und der Zugriff ist deutlich langsamer. Caches werden im Kurs 01609 behandelt.

Cache

Wir wollen uns nun den Halbleiterspeichern zuwenden, die als Cache- und Hauptspeicher verwendet werden. Innerhalb eines Prozessors realisiert man mit ihnen auch die Register, Puffer- und Mikroprogrammspeicher. Hochintegrierte Halbleiterspeicher für Computersysteme haben meist wahlfreien Zugriff (Random Access). Sie können weiter unterteilt werden in *flüchtige* und *nichtflüchtige* Speicher. Nach dem Abschalten der Stromversorgung verlieren flüchtige Speicher ihren Inhalt. Zu den nichtflüchtigen Speichern zählen die verschiedenen Varianten von ROMs, die wir schon in Kurseinheit 3 kennengelernt haben. In jedem Computersystem gibt es mindestens einen nichtflüchtigen Speicher, der den Prozessor unmittelbar nach Einschalten der Betriebsspannung mit Maschinenbefehlen versorgt.

Schreib-/Lesespeicher sind flüchtige Speicher und werden als *RAMs* (Random Access Memory) bezeichnet, obwohl die damit beschriebene Eigenschaft ebenso auch für die ROMs zutrifft. Je nach Speicherprinzip unterscheiden wir zwei Arten von RAMs: *statische S*RAMs und *dynamische D*RAMs. SRAMs basieren auf bistabilen Kippstufen und benötigen pro Speicherbit meist sechs Transistoren. DRAMs speichern die Bits dagegen als Ladungspakete und benötigen pro Bit nur einen Transistor. Bei gleicher Integrationsdichte und gleicher Fläche kann also mit DRAMs eine höhere Speicherkapazität bereitgestellt werden als mit SRAMs. Daher wird der Hauptspeicher eines Computersystems meist mit DRAMs realisiert. Das Auslesen der Daten zerstört bei DRAMs die Ladungspakete in den Zellen und erfordert ein zeitaufwendiges Wiedereinschreiben. Für Caches verwendet man daher SRAMs, da sie deutlich schneller als DRAMs sind.

Schreib-/Lesespeicher

4.14.1 Speicherorganisation

Register bestehen aus Flipflops, die durch einen gemeinsamen Takt gesteuert werden (vgl. Kurseinheit 3). Sie werden benötigt, um ein Datum kurzzeitig zu speichern. Man findet sie sowohl im Rechen- als auch Leitwerk eines Prozessors. Da sie direkt durch die Ablaufsteuerung angesprochen werden, ist die Zugriffszeit sehr gering. Externe Halbleiterspeicher haben dagegen eine weitaus höhere Speicherkapazität und müssen für den Zugriff auf die gespeicherten In-

formationen über eine geeignete *Speicherorganisation* verfügen.

Wir wollen im Folgenden untersuchen, wie RAM- und ROM-Speicher organisiert sind, d.h. wie man die Speicherzellen anordnet und ihren Inhalt liest bzw. verändert. Eine Speicherzelle nimmt die kleinste Informationseinheit von einem Bit auf. Je nach Herstellungstechnologie benutzt man unterschiedliche Speicherprinzipien. Allen gemeinsam ist jedoch die matrixförmige Anordnung der Speicherzellen, da hiermit die verfügbare Chipfläche am besten genutzt wird.

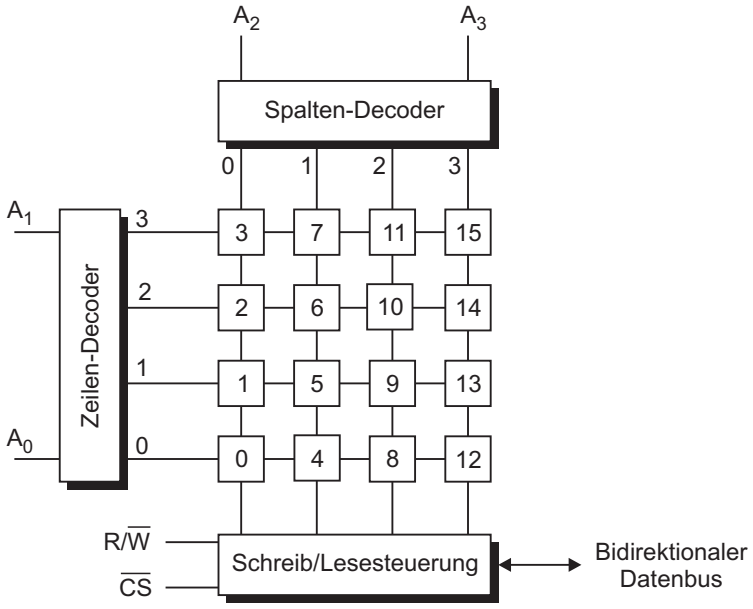


Abbildung 4.22: Aufbau eines bitorientierten Speichers (16×1).

Bitweise organisierte Halbleiterspeicher (Abbildung 4.22) adressieren immer nur jeweils eine einzige Speicherzelle. Die Adressleitungen werden in zwei Teile aufgespalten, die man dann als *Zeilen-* und *Spalten*adresse benutzt. Die decodierten Zeilen- und Spaltenadressen dienen zur Auswahlsteuerung für die Speichermatrix. Jeder möglichen Adresse wird genau ein Kreuzungspunkt der decodierten Zeilen- und Spaltenauswahl-Leitungen zugeordnet. Sind beide auf 1-Pegel, so ist die zugehörige Speicherzelle aktiviert. Die Schreib-/Lesesteuerung, die mit vertikal verlaufenden Daten-Leitungen verbunden ist, ermöglicht den Zugriff auf die adressierte Speicherzelle. Mit dem $R/\overline{W}$-Signal ($Read = 1$, $Write = 0$) wird zwischen einem Lese- oder Schreibzugriff unterschieden. Die Schreib-/Lesesteuerung enthält in der Regel auch TriState-Treiber für einen bidirektionalen Datenbus. Mit dem $\overline{CS}$-Signal ($ChipSelect = 0$) wird der TriState-Treiber aktiviert. Bei dynamischen RAMs enthält die Schreib-/Lesesteuerung eine Schaltung zum Auffrischen der in Form von Ladungspaketen gespeicherten Informationen.

Wortorganisierte Speicher adressieren mehrere Speicherzellen gleichzeitig. Setzt man eine quadratische Speichermatrix voraus, so reduziert sich dadurch die Zahl der Spaltenleitungen. Bei großen Wortbreiten kann dies zu einer *eindimensionalen* Adressierung führen. Die Speicherwörter sind dabei zeilenwei-

se angeordnet. Diese Organisationsform findet man vorwiegend bei EPROM-Speichern (Abbildung 4.23). Der Adressdecoder legt für jede Adresse genau eine Wortleitung auf 1-Pegel. Im Kreuzungspunkt Daten-/Wortleitung befinden sich Halbleiter-Koppelelemente wie Dioden oder Feldeffekt-Transistoren.

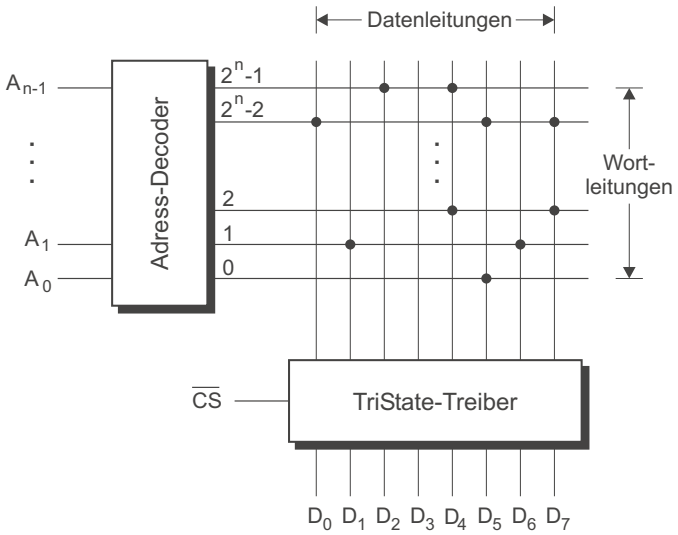


Abbildung 4.23: Eindimensionale Adressierung mit bei einem ROM mit 8-Bit-Wortbreite und einer Speicherkapazität von 2^n Wörtern.

4.14.2 Speichererweiterungen

Wenn die von Halbleiterherstellern angebotenen Speicherchips nicht die gewünschte Wortbreite und Kapazität haben, müssen wir mehrere solcher Speicherchips verwenden, um einen größeren System-Speicher aufzubauen. Wir können den Speicher auf drei Arten erweitern:

1. Wortbreite erhöhen,
2. Speicherkapazität erhöhen oder
3. beides gleichzeitig.

Die Vergrößerung der Wortbreite ist am einfachsten. Hierzu brauchen wir nur die Adress- und Steuerleitungen mehrerer Speicherchips parallel zu schalten. Um z.B. einen Speicher mit m Bit Wortbreite ($m = 8k, k \in \mathbb{N}$) aus Bausteinen mit einer Wortbreite von 8 Bit zu realisieren, müssen $m/8$ byteweise organisierte Speicherbausteine parallel geschaltet werden. Meist werden den Herstellern aber schon integrierte Speichermodule mit 32- oder 64-Bit-Wortbreite bereitgestellt, die dann über eine standardisierte Kontaktleiste in ein Computersystem eingesteckt werden können.

Die Speicherkapazität eines Speicherchips oder -moduls deckt meist nur einen kleinen Teil des mit dem Adressbus adressierbaren Bereichs ab. Um durch mehrere Speicherbausteine die Größe des verfügbaren Hauptspeichers zu erhöhen, werden die (active low) $\overline{CS}$-Auswahlleitungen benutzt.

Der mit Speicher belegte Adressraum sollte natürlich nahtlos abgedeckt werden. Die Zahl der Adressleitungen eines Speichermoduls entspricht dem Zweierlogarithmus seiner Speicherkapazität in Worten. Um jedes Speicherwort erreichen zu können, werden die Adressleitungen des Speichermoduls mit den niederwertigen Adressleitungen des Adressbusses verbunden. Wenn SA_i die Adressleitung des Speichermoduls mit der Wertigkeit 2^i bezeichnet, dann muss diese mit der gleichwertigen Leitung A_i des Adressbusses (als Teil des Systembusses) verbunden werden. Für ein Speichermodul mit der Speicherkapazität 2^N gilt dann:

$$SA_i = A_i \quad i = 0, 1, \cdots, N-1$$

Aus den verbleibenden höherwertigen Adressleitungen des Adressbusses muss nun mit einem Decoder-Schaltnetz für jedes Speichermodul j ein eigenes $\overline{CS_j}$-Signal erzeugt werden, das nur dann den Wert 0 liefert, wenn die anliegende Adresse A durch das jeweilige Speichermodul abgedeckt wird. Die $\overline{CS_j}$-Signale müssen sich also wechselseitig ausschließen, um die Adressbereiche der einzelnen Speichermodule nahtlos im Adressraum des Hauptspeichers aneinanderfügen.

Ähnlich wie bei den Speichermodulen werden auch einzelnen Ein-/Ausgabebausteine bestimmte (deutlich kleinere) Adressbereiche zugeordnet. Im Gegensatz zu den Speichern müssen diese Adressbereiche allerdings nicht unbedingt zusammenhängend sein. Trotzdem braucht man auch hier ein entsprechendes Decoder-Schaltnetz, um die sich wechselseitig ausschließenden $\overline{CS}$-Signale zu erzeugen.

4.15 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 4.1 von Seite 157

Die bedingte Ausgangsbox wird nur aktiviert, wenn sie auf einem Pfad durch vorgelagerte Entscheidungsboxen erreicht wird. D.h. das Schreibsignal (Teil des Steuervektors) des Registers der Variablen, die auf der linken Seite der Zuweisung in der bedingten Ausgangsbox steht, wird nur bei bestimmten Werten des Statusvektors (der die Ergebnisse der Entscheidungsboxen enthält) aktiviert. Der Steuervektor ist die Ausgabe des Automaten im Steuerwerk, der Statusvektor stellt die Eingabe dar. Damit ist in dem Automaten des Steuerwerks die Ausgabe abhängig von der Eingabe. Dies erfordert einen Mealy-Automaten, da bei einem Moore-Automaten die Ausgabe nur vom Zustand, nicht aber von der Eingabe abhängig ist.

Selbsttestaufgabe 4.2 von Seite 158

Für jeden Wert von A wird z_2 als Folgezustand gewählt. Solange $B > 0$ ist, besteht kein Konflikt. Sobald aber $B \leq 0$ wird, werden gleichzeitig z_2 und z_3 als Folgezustände ausgewählt. Damit wird gegen die angegebene Regel verstoßen.

Selbsttestaufgabe 4.3 von Seite 160

Der Aufbau des Operationswerks ist in Abbildung 4.24 dargestellt. Da das Register R_2 bei $S = 1$ nicht verändert werden darf, muss der Takt durch ein AND–Schaltglied „maskiert“ werden. Der Takt wird nur für $S = 0$ an R_2 weitergeleitet. In diesem Fall übernimmt das Register R_2 den Inhalt von Register R_1. Gleichzeitig wird über den Multiplexer vor R_1 das Register R_2 zum Einspeichern in R_1 ausgewählt. Für $S = 1$ wird die Summe von R_1 und R_2 ausgewählt und von R_1 übernommen.

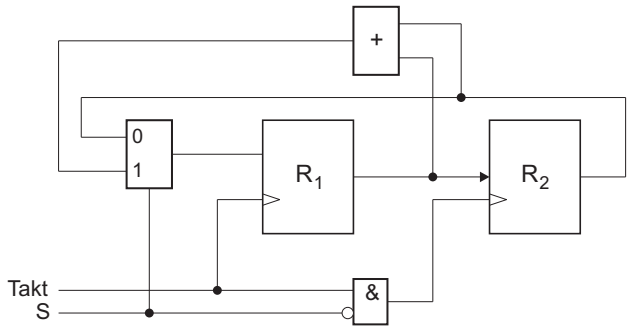


Abbildung 4.24: Aufbau des Operationswerks

Selbsttestaufgabe 4.4 von Seite 162

In Abbildung 4.25 ist der Aufbau der Schaltnetzlösung für 6 Bit dargestellt. Wenn wir mit zwei dieser Modulen einen 12-Bit-Vektor auswerten, erhalten wir

an den Modul-Ausgängen zwei 3-Bit-Wörter. Diese müssen mit einem 4-Bit-Addierschaltnetz aufsummiert werden, damit das größtmögliche Ergebnis 12 (=0C$_{hex}$) dargestellt werden kann.

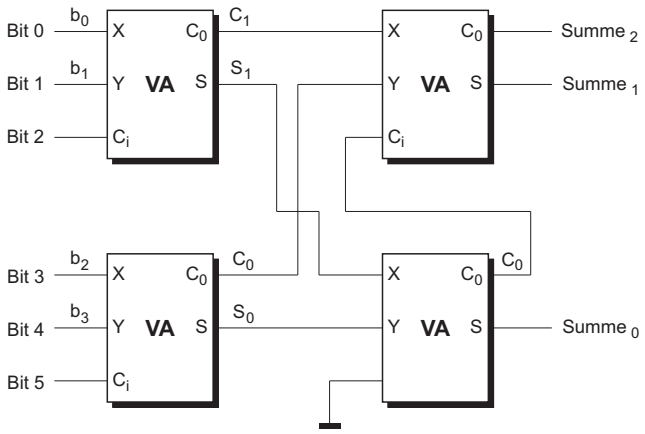


Abbildung 4.25: Aufbau eines Schaltnetzes zum Zählen von Einsen in einem 6-Bit-Wort.

Selbsttestaufgabe 4.5 von Seite 167

Aus dem ASM-Diagramm in Abbildung 4.7 entnehmen wir, dass nur im Zustand z_4 geschoben wird. Das bedeutet, dass in allen anderen Zuständen $Schieben = 0$ ist. Somit würde A nicht nur im Zustand z_1 mit X geladen, sondern auch in den Zuständen z_0, z_2 und z_3. Obwohl dies für die vorliegende Problemstellung im Zustand z_0 „unschädlich“ wäre, würde das Nachladen von A mit dem Wert X im Zustand z_3 zu einer Endlosschleife führen. Es würde dann nicht die durch das ASM-Diagramm spezifizierte Funktion ausgeführt und es entstehen unerwünschte Seiteneffekte. Das Schieberegister muss daher über ein weiteres Steuersignal *Aktiv* verfügen, das die über *Schieben* ausgewählte Funktion ein- bzw. abschalten kann.

Selbsttestaufgabe 4.6 von Seite 170

1. Die benötigte Wortbreite beträgt 10 Bit.

2. Siehe linke Seite von Abbildung 4.26.

3. Siehe rechte Seite von Abbildung 4.26. Über die Steuerleitung S_0 kann die Zuweisungsquelle an A gewählt werden. Mit $S_1 = 1$ kann man B nur im Zustand z_0 den aktuellen Wert des Eingabevektors X zuweisen. Im Zustand Z_1 ist $S_1 = 0$.

4. Damit A z.B. Werte von 0 bis 999 annimmt, muss die Abfrage $A < 998$ lauten. Bei $A = 998$ wird dann der Rücksetzzustand z_0 vorbereitet und beim Eintritt in den Rückssetzzustand gilt $A = 999$. Dort wird der

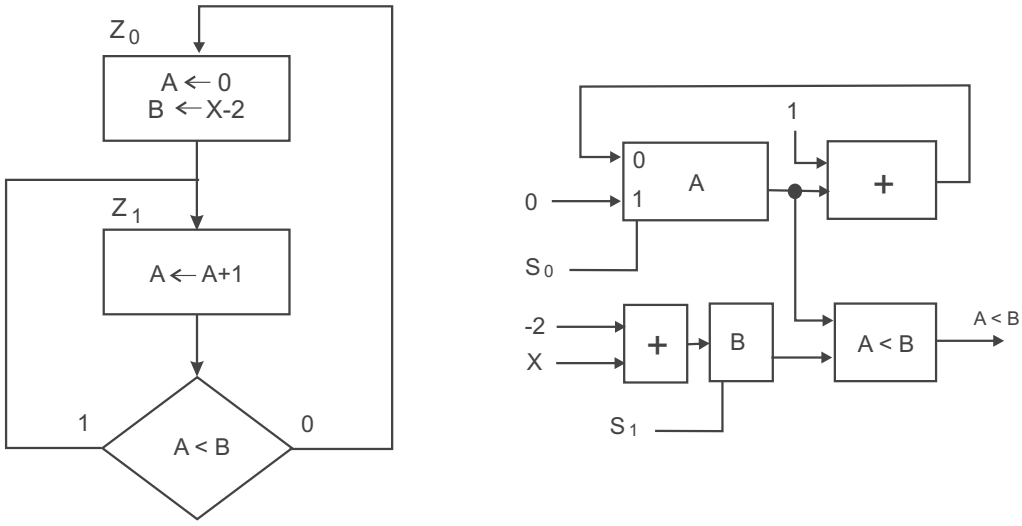


Abbildung 4.26: ASM-Diagramm und Operationswerk zum Modulo-X-Zähler.

Maximalwert erreicht und erst am Ende des Taktzyklus wieder auf 0 zurückgesetzt.

Index