Zum Geleit

Im Laufe der vergangenen Jahrzehnte hat sich die Informatik als eine Disziplin herausgebildet, die sich mit dem Entwurf und der Bewertung von Computern beschäftigt. Rechnerstrukturen stehen dabei im Zusammenhang mit der Architektur und dem Verhalten digitaler Computer und Informationssysteme. Während ständig neue Konzepte den Markt mit leistungsfähigeren Produkten überschütten, gibt es jedoch auch Grundlagen und -strukturen, die alle Architekturen gemeinsam haben und die sich im Laufe der Zeit nicht ändern. Und genau diese Thematik ist es, die im vorliegenden Werk behandelt wird.

Doch zunächst etwas zur Entwicklung der Informatik. Während diese Wissenschaftsrichtung als sehr jung angesehen wird, handelt es sich — streng genommen — um eine Disziplin, deren Prinzipien so alt sind, wie die Menschheit selbst. Nicht zuletzt kann jedes menschliche Wesen selbst Funktionen eines Computers ausüben. Auch wenn danach der Abacus und Rechenschieber dem Menschen Arbeit erleichterten (jedoch manuelle Arbeiten nicht ersetzten), so wird als Beginn der *digitalen* Rechentechnik das Jahr 1944 angesehen, in dem John von Neumann den ersten programmgesteuerten Rechenautomaten konstruierte. Neben den damit verbundenen Erleichterungen war die Basis für eine sowohl schnellere als auch fehlerunanfälligere Rechenleistung geschaffen.

Somit bietet es sich an, die Bestandteile eines Rechners auf die menschlichen Papier-und-Bleistift-Schulrechnungen abzubilden, um einen ersten Einstieg in die Begriffswelt der Rechnerstrukturen zu erhalten. Der Zweck des Papiers ist der eines *Informationsspeichers*. Die auf dem Papier gespeicherten Informationen können dabei sowohl *Daten* (Zahlen, Variablen, Wörter, ...) als auch Listen mit Instruktionen und Rechenvorschriften, d.h. *Algorithmen* oder *Programme,* enthalten. Während der Berechnungen werden sowohl Zwischenergebnisse als auch das Endergebnis auf dem Papier notiert, bzw. im Informationsspeicher gespeichert. Der eigentliche Rechenprozeß wird im Gehirn des Menschen ausgeführt, er ist vergleichbar mit dem *Prozessor* des Rechners. Dabei werden vom Gehirn im wesentlichen zwei Funktionen übernommen: eine *Kontrollfunktion*, die Anweisungen interpretiert und dafür sorgt, daß diese in der richtigen Reihenfolge ausgeführt werden, und eine *Ausführungsfunktion*, die das eigentliche (stupide) Rechnen, insbesondere die Rechenoperationen der Addition, ausführt. Bei der Funktionalität der Ausführungsfunktion wird das menschliche Gehirn heutzutage auch von Taschenrechnern unterstützt.

Die Grundkomponenten eines Rechners sind ähnlich. Eine *Memory Unit* oder ein Speicher übernehmen die Funktion des Papiers, um Daten zu speichern. Die *Control Unit*, d.h. eine Steuereinheit, interpretiert und speichert Instruktionen und Daten und die *Arithmetic Logical Unit*, d.h. die Arithmetisch-Logische Einheit führt diese Instruktionen aus. Die Steuer- und die Arithmetisch-Logische

Einheit bilden zusammen die *Central Processing Unit*, also CPU, deren Funktionalität weitestgehend den Aufgaben des menschlichen Gehirns entsprechen könnte.

Zu den vielen Unterschieden, die es zwischen Mensch und Rechner in diesem Zusammenhang gibt, gehört zum Beispiel die Art, in der Daten und Instruktionen dargestellt werden. Während der Mensch natürliche Sprachen verwendet und eine große Anzahl von Symbolen, Bildern und dezimalen Zahlen interpretieren kann, verwendet der Rechner prinzipiell nur binäre Formen, die als Bits bezeichnet werden. Um die menschlichen Informationen in solche zu konvertieren, die für den Computer brauchbar sind (und umgekehrt), wird sogenanntes *Input/Output-Equipment*, d.h. eine Ein-/Ausgabe-Einheit verwendet.

Das vorliegende Büchlein umfaßt diese genannten Begriffe mehr oder weniger. Neben allgemeinen Konzepten wird insbesondere auf die Arithmetisch-Logische Einheit eingegangen. Der vorliegende Stoff wurde in Anlehnung an die von Prof. Dr. Otto Spaniol im Sommersemester 1995 gehaltene Vorlesung "Rechnerstrukturen" zusammengestellt. Die editoriellen Arbeiten des Skripts wurden im wesentlichen von Herrn cand. inform. Frank Anstötz übernommen, bei dem wir uns in diesem Zusammenhang für die geleistete Arbeit besonders bedanken möchten.

Aachen, im Juli 1995

Inhaltsverzeichnis

1. Grundlegende Konzepte

Wir befassen uns in dieser Vorlesung mit digitalen und programmgesteuerten Rechensystemen.

Der Rechner akzeptiert digitale **Eingabedaten** (Eingangsargumente), verarbeitet sie mit einem **Programm**, das in seinem Speicher steht, und produziert digitale **Ausgabedaten** (Ausgangsargumente).
Nimmt man vereinfachend an, daß die Eingabedaten im Speicher stehen und die Ergebnisse ebenfalls im Speicher abgelegt werden, dann bewirkt ein Programm eine **Transformation** des alten Speicherinhaltes in einen neuen Speicherinhalt.

Rechnersysteme unterscheiden sich grundlegend voneinander bezüglich Größe, Geschwindigkeit, Zahl anschließbarer beziehungsweise bedienbarer Geräte oder Benutzer und ihrer Kosten.
Man unterscheidet folgende Klassen von Rechnersystemen:

- Supercomputer
- Mainframes
- Superminis
- Minicomputer
- Workstations
- Prozeßrechner
- Personal Computer.

Die Grenzen zwischen diesen Klassen sind fließend.

1.1 Funktionelle Einheiten eines Rechners

Ein Rechnersystem besteht aus folgenden Einheiten, die sich aufgrund ihrer Funktion innerhalb des Systems voneinander unterscheiden.

Die **Eingabe-** und die **Ausgabeeinheit** bilden die Schnittstellen zur Außenwelt.
Der **Prozessor** (CPU — *Central Processing Unit*) hat Aufgaben der Steuerung und Arithmetik. Er vereinigt demnach die **Steuereinheit** (*Control Unit*) und die **Recheneinheit** (ALU — *Arithmetic Logical Unit*). Eine weitere wichtige Einheit ist der **Speicher**. Auf ihn wird später eingegangen. Abbildung 1.1 veranschaulicht die Zusammenarbeit der einzelnen Komponenten.

Es gibt erweiterte Konzepte, bei denen mehrere Rechner zusammenarbeiten, wobei diese dann durch ein **Kommunikationsnetz** verbunden sind.

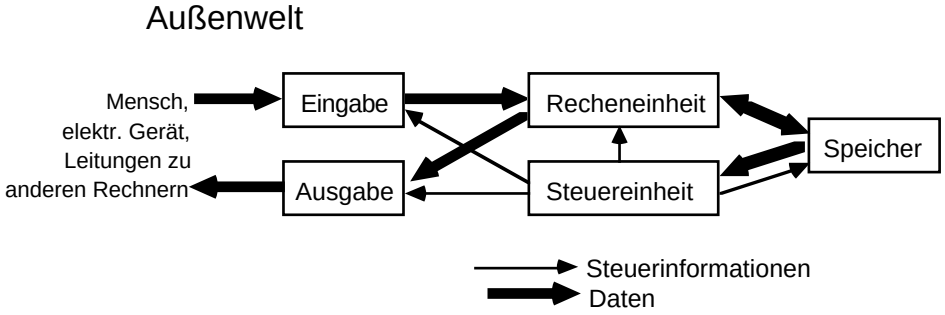


Abbildung 1.1: Rechnereinheiten

Ein Rechensystem verarbeitet **Informationen**, die im Speicher gehalten werden. Man unterscheidet zwei Arten von Informationen. Zum einen **Befehle**, die Aktivitäten des Prozessors anstoßen, und auf der anderen Seite **Daten,** die manipuliert werden. Ein Programm ist dabei eine Folge von Befehlen. Typische Daten sind zum Beispiel Zahlen und Zeichen, die in geeigneter Codierung im Speicher abgelegt werden. Da sich ein Programm meist auch im Speicher befindet, sind die Codierungen der Befehle auch als Daten auffaßbar.

Der Prozessor verarbeitet Informationen im einfachsten Fall nach folgendem Ablauf:

1. Befehl (aus dem Speicher) holen
2. Befehl interpretieren
3. Aufgrund der Interpretation des Befehls nötige Operanden holen
4. Befehl (Operation) ausführen

Anschließend wird dieser Ablauf mit dem nächsten Befehl durchgeführt.

Die Bearbeitung erfolgt **sequentiell**, das heißt, die Befehle werden nacheinander und jeder für sich zu einem Zeitpunkt ausgeführt.
Es gibt Befehle, deren Aufgabe es ist, nicht den nachfolgenden, sondern einen an anderer Stelle im Speicher befindlichen Befehl anzustoßen (sogenannter **Sprungbefehl**). Ferner dienen Vergleichsbefehle dazu, Fallunterscheidungen zu bilden, so daß aufgrund eines Speicherzustandes eine bestimmte Aktion ausgeführt wird.

Üblicherweise werden ein Programm (oder auch mehrere) und die zugehörigen Daten im Speicher gehalten. Zu diesem Zweck enthält der Speicher eine Menge von adressierbaren Speicherzellen (Worten).
Zur Unterscheidung zwischen Daten und Befehlen muß man entweder die Speicherzelle markieren (zum Beispiel ein gesondertes Bit des Wortes zur Verfügung stellen wie: 1 steht für Datum und 0 steht für Befehl) oder den aktuellen CPU-Zustand interpretieren.

Abbildung 1.2 zeigt den Speicher, der in zwei Abschnitte zur Aufnahme von Programmen und Rechendaten aufgeteilt ist.

Die Unterscheidung zwischen Befehl und Datum kann zum Beispiel durch ein Bit an erster Stelle des Speicherwortes erfolgen. Ferner kann man sich auch beide Bereiche ineinander verschränkt denken.

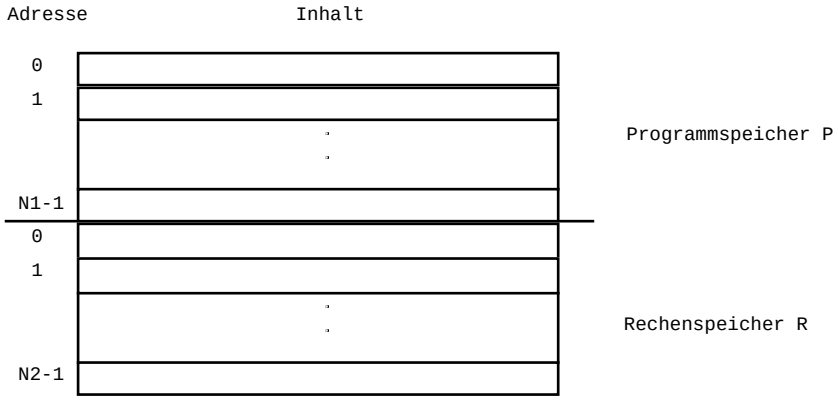


Abbildung 1.2: Programm- und Rechenspeicher

Wir nehmen also ohne wesentliche Einschränkung an, daß wir zwei Speicher P und R haben, die jeweils von Null ab adressiert werden. Den Inhalt der Speicherzellen kennzeichnen wir im Fall eines

Befehls mit π (i) (aktueller Inhalt von Zelle i in P),
Datums durch ρ (j) (aktueller Inhalt der Zelle j in R).

Speichertypen

Man unterscheidet im wesentlichen drei Arten von Speichern.

Der **Hauptspeicher** (*Primary Memory, Main Memory*) ist direkt adressierbar mit nahezu gleicher Zugriffszeit pro Zelle (typische Werte liegen zwischen 100 und 500 Nanosekunden). Er wird demnach auch als *Random Access Memory* (RAM) bezeichnet.
Einen **Hintergrundspeicher** (*Secondary Memory*) verwendet man zur Speicherung relativ selten benötigter Informationen. Er ist langsamer bezüglich der Zugriffszeit. Da er nicht die teuren und schnellen Speicherbausteine verwendet, ist er wesentlich billiger und daher auch in größerem Umfang verfügbar. Als Beispiele sind Platten- oder Magnetbandspeicher zu nennen.
Der **Pufferspeicher** (*Cache Memory*) zeichnet sich dadurch aus, daß er besonders schnell, dafür aber auch teuer ist. Aus diesem Grund ist er relativ klein. Seine besondere Eignung liegt in der Pufferung (vom Prozessor) besonders häufig benutzter Befehle und Daten.

Register

Neben den Speichertypen gibt es die **Register.** Es handelt sich dabei um Zellen mit besonders kurzen Zugriffszeiten. Den Registern werden bestimmte Sonderaufgaben zugeordnet, wie sie im folgenden beispielhaft aufgeführt sind.

Dabei unterscheiden wir der Einfachheit halber nicht zwischen Register**name** und Register**inhalt**.

Das **Akkumulatorregister** (der Akkumulator, Bezeichnung durch α) dient der kurzfristigen Speicherung von Informationen. Es wird für den Transfer vom und zum Speicher benutzt. Verwendung findet es zum Beispiel bei der Ausführung arithmetischer Operationen.
Eine typische Befehlsfolge unter Verwendung des Akkumulators ist die folgende:

$\alpha := \rho(0)$
$\alpha := \alpha + \rho(1)$
$\alpha := \alpha + \rho(2)$
$\rho(0) := \alpha$

Dieses Programm sorgt für eine Addition der Inhalte der ersten drei Rechenspeicherzellen. Die Summe wird in die erste Speicherzelle geschrieben.

Das **Indexregister** (bezeichnet mit γ) wird bei der indizierten Adressierung verwendet, die in Kapitel 2 eingeführt wird.
Nicht zu verwechseln sind das **Befehlsregister** (*Instruction Register*, abgekürzt durch IR) und das **Befehlszählregister** (*Program Counter* — PC, auch mit β bezeichnet). Das Befehlsregister enthält den Befehl, der momentan ausgeführt wird. Dahingegen steht im Befehlszählregister die Adresse des zur Zeit ausgeführten Befehls. Ist die Ausführung des einen Befehls abgeschlossen und soll der nächste (im Speicher nachfolgende) Befehl aufgerufen werden, so wird der Inhalt des Befehlszählregisters inkrementiert ($\beta := \beta + 1$).

Das **Speicheradreßregister** (*Memory Address Register* — MAR) enthält die Adresse einer Speicherzelle, während das **Speicherdatenregister** (*Memory Data Register* — MDR) das Datum dieser durch das Speicheradreßregister angesprochenen Speicherzelle enthält.
Darüber hinaus gibt es noch eine architekturabhängige Vielzahl von Allzweckregistern, die meist mit R0, R1, R2 usw. bezeichnet werden. Sie werden für unterschiedliche Zwecke eingesetzt, zum Beispiel für die Durchführung von Rechenoperationen.

1.2 Programmausführung

Es gibt verschiedene Konzepte zur Ausführung von Befehlen und somit zur Manipulation von Daten.

Von-Neumann-Prinzip

Die konventionelle Programmausführung erfolgt nach dem **Von-Neumann-Prinzip**: Es wird zu einem Zeitpunkt ein Befehl auf ein Datum angewandt. Man nennt den zugehörigen Rechner auch **SISD-Rechner** (*Single Instruction Single Data*).

Die Ausführung eines Befehls geschieht in vier Schritten:

1. Hole nächsten Befehl in das Befehlsregister (IR).

Die Adresse des Befehls steht im Befehlszählregister β; demzufolge enthält β zu Beginn der Programmausführung die Startadresse.

2. Interpretiere den Befehl, das heißt, der Befehl wird decodiert. Es wird der Typ des Befehls ermittelt und die Anzahl nötiger Operanden bestimmt.
3. Hole den beziehungsweise die benötigten Operanden.
4. Führe den Befehl aus.

Der „Hole"-Befehl (Laden, engl.: *Load)* beschreibt Datentransfer zwischen Speicher und Prozessor. Er hat die Form: $\alpha := \rho(i)$ und bedeutet: Lade Datum von Speicherzelle i in den Akkumulator. Entsprechend lautet der Befehl für einen „Speichere"-Befehl (engl.: *Store*): $\rho(k) := \alpha$.
Auf den Aufbau und die Darstellung eines Befehls wird in Kapitel 2 eingegangen.

Alternative Modelle

Neben dem SISD-Prinzip gibt es folgende andere Typen, die hier nur kurz genannt werden:

Der **SIMD**-Rechner (*Single Instruction Multiple Data*) gestattet die Anwendung eines Befehls zu einem Zeitpunkt auf mehrere Daten. Es gibt Rechnertypen, die hierzu mehrere Prozessoren verwenden, welche zum Beispiel einen gemeinsamen Speicher nutzen (*Shared Memory Model*). Ein weiteres Modell stellt der **MIMD**-Rechner dar (*Multiple Instruction Multiple Data*), selten wird der **MISD**-Rechner (*Multiple Instruction Single Data*) verwendet.

Pipelining

Die Befehlsausführung nach dem Von-Neumann-Prinzip in vier Schritten kann beschleunigt werden, wenn man Schritt 1 und 2 des nachfolgenden Befehls mit den Schritten 3 und 4 des aktuell auszuführenden Befehls kombiniert. Abbildung 1.3 zeigt, daß der Prozessor nach der Interpretation des Befehls i (Schritt 2) zusätzlich zu dessen weiterer Bearbeitung (Schritt 3 und 4) den Befehl (i+1) holt und interpretiert.

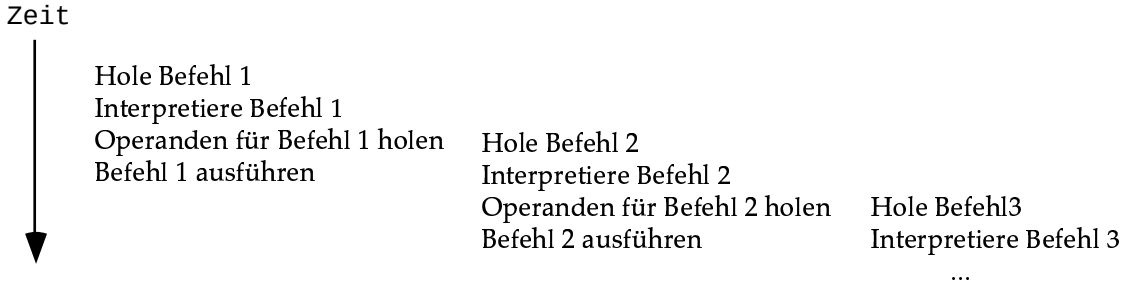


Abbildung 1.3: Pipelining

Zu beachten ist allerdings, daß die Ausführung von Befehl i (Schritt 4) unter Umständen erst den Befehl (i+1) festlegen kann. In diesem Fall ist irrtümlich der falsche Befehl (i+1) ausgeführt worden, was zu ungewünschten Folgen führen kann. Der nun richtige Befehl (i+1) muß, sofern noch möglich, ausgeführt werden.

Ausführungssteuerung

Neben Befehlen zum Datentransfer zwischen Prozessor und Speicher werden auch die **Ein-/Ausgabegeräte** (E/A, *Input/Output — I/O*) über E/A-Befehle vom Prozessor angesteuert. Ihre Tätigkeit können solche Endgeräte dann weitgehend unabhängig vom Prozessor ausführen. Eine wichtige Aufgabe hat der Prozessor bei der Koordination dieser Geräte. Hierzu senden die Endgeräte sogenannte **Interrupt-Signale**, die den Prozessor in der Ausführung eines anderen Programms unterbrechen. Auf diese Weise kann von der „normalen" Befehlsausführung abgewichen werden.

Beispiele für derartige Unterbrechungen des Prozessors sind eine Alarmmeldung, die eine Reaktion innerhalb einer nicht zu überschreitenden Zeitspanne (*Deadline*) erfordert, oder eine Fertigmeldung eines langsamen Endgerätes. Der vergleichsweise schnelle Prozessor bedient normalerweise viele langsame Endgeräte gleichzeitig, er muß also zwischen gleichzeitig laufenden Programmen umschalten.

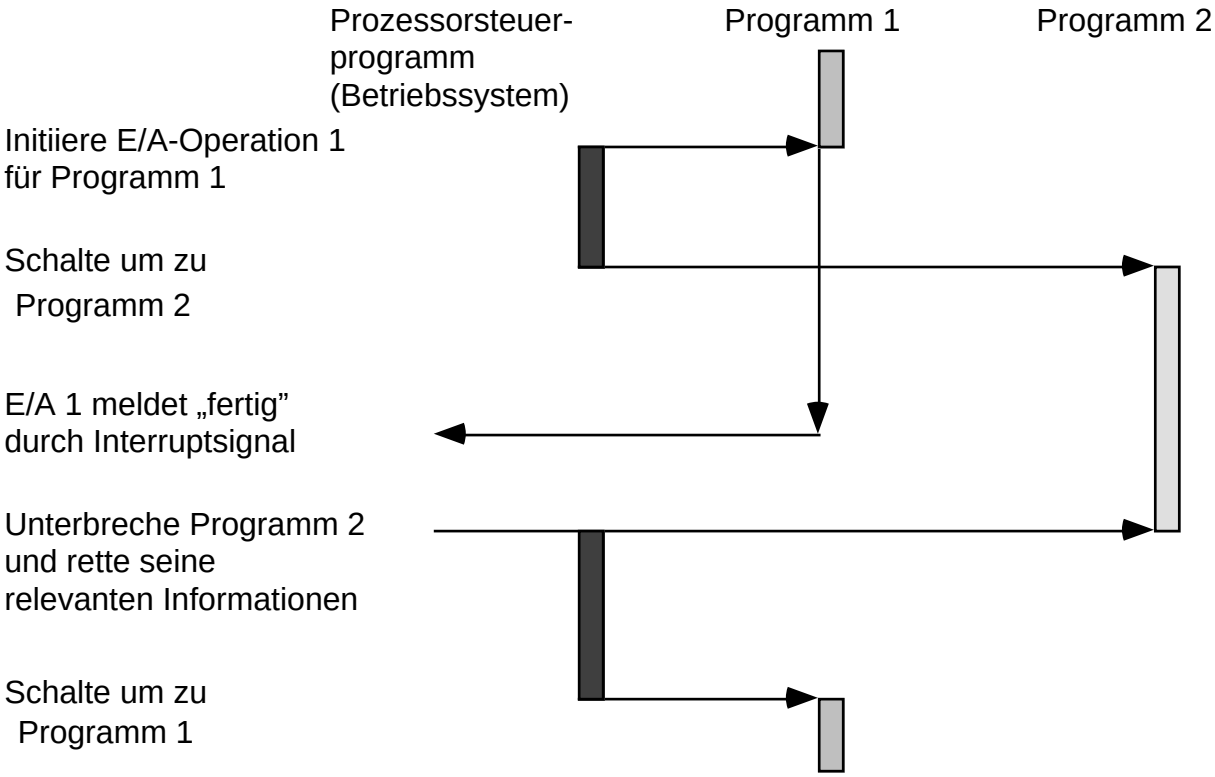


Abbildung 1.4: Ausführungssteuerung mit Interruptsignal

1.3 Datenübertragung über Busse

Leitungen

Die Kommunikation besteht im wesentlichen im Transfer von Daten und Steuerinformationen zwischen den funktionellen Einheiten des Rechners, insbesondere zwischen Prozessor, Speicher und E/A-Geräten.
Sie erfolgt über Leitungen, welche die Einheiten verbinden. Physikalisch gesehen sind Leitungen meist drahtgebunden. Aber auch andere Möglichkeiten sind gegeben wie etwa Funkstrecken.
Informationstransport auf einer Leitung erfolgt in der Regel sequentiell, das heißt, ein Bit wird nach dem anderen auf die Leitung gebracht und ebenso empfangen. Der **parallele** Transport bedeutet, daß mehrere Bits gleichzeitig über ein „Bündel" von Leitungen gesendet werden. Ein solches „Bündel" wird auch als **Bus** bezeichnet.

Busse

Ein Rechnerbus besteht aus einer Menge von Leitungen. Ein Bus ist im Rechner als „öffentliches Transportmedium" für mehrere Einheiten aufzufassen, die über diesen miteinander verknüpft sind. Der Zugriff auf den Bus, also die Busbenutzung, wird daher auch als ***Multiple Access*** (MA) bezeichnet und muß durch Konventionen (in Form von „Protokollen") geregelt werden.

Bustypen

Busse lassen sich nach den Einheiten charakterisieren, die sie miteinander verbinden. Ebenso gibt es verschiedene **Busarchitekturen**, welche die Zuordnung von Bussen an Einheiten beschreiben.
Abbildung 1.5 zeigt eine Architektur mit zwei Bustypen: Der **Speicherbus** dient der Kommunikation zwischen Prozessoreinheit und dem Speicher. Ein eigener **Rechnerbus** sorgt für die Kommunikation des Prozessors mit den E/A-Geräten.

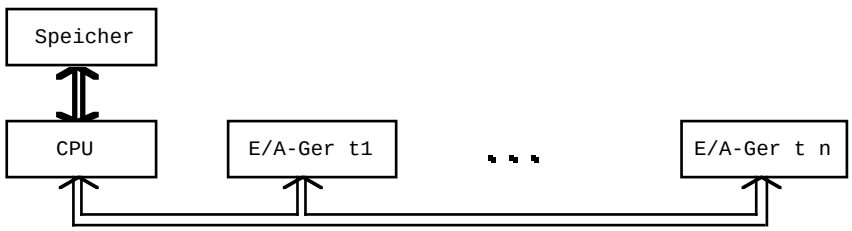


Abbildung 1.5: Zwei-Bus-Architektur

Einfacher, aber weniger leistungsfähig als diese Zwei-Bus-Architektur ist die Verwendung eines gemeinsamen Busses (sogenannter **Unibus**) für alle Kommunikationsaufgaben zwischen allen Einheiten.

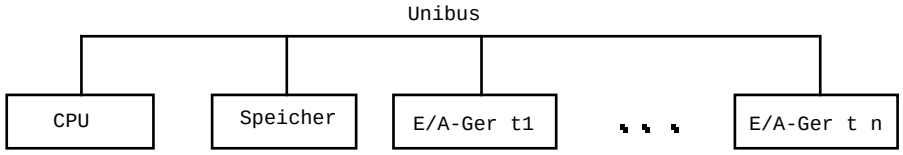


Abbildung 1.6: Unibus-Architektur

Ein grundlegendes Problem ist die unterschiedliche Geschwindigkeit, mit der die Komponenten arbeiten, die an einen gemeinsamen Bus angeschlossen sind, zum Beispiel: schnelle CPU und langsames E/A-Gerät. Benutzt eine langsame Einheit den Bus, so ist er für eventuell längere Zeit für andere Einheiten blockiert. Dies erklärt auch, warum der Unibus nicht sehr leistungsfähig ist.
Erforderlich sind also Hilfsmittel, welche die Geschwindigkeitsunterschiede glätten, um weniger Busse verwenden zu können. Ein Lösungsansatz besteht im Einsatz von Pufferregistern zwischen dem Bus und den angeschlossenen Geräten.

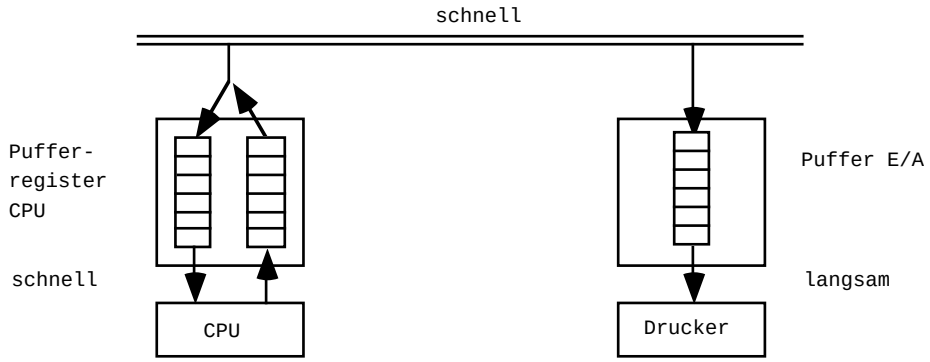


Abbildung 1.7: Pufferung des Buszugangs

Auf diese Weise kann zum Beispiel eine schnelle CPU viele langsame Endgeräte wechselweise bedienen. Der Bus ist allerdings nur solange nicht blockiert, bis der Puffer voll ist.

2. Informationsdarstellung, Adressierung und maschinennahe Programmierung

In diesem Kapitel werden zunächst Darstellungsformen und Codierungen für Daten - insbesondere Zahlen und Zeichen - sowie für Befehle eingeführt. Ferner werden verschiedene Arten der Speicheradressierung besprochen. Schließlich wird erklärt, was es heißt, Programme und Unterprogramme auszuführen.

Die Darstellung eines Datums oder Befehls findet meist in einem Speicherwort statt. Es sind auch Halb- oder Doppelwortdarstellungen gängig. Ein Speicherwort ist somit als Informationseinheit des Rechners zu verstehen. Die Wortlänge ist dabei abhängig vom Rechner; es handelt sich in der Regel um Potenzen von 2 (16 bzw. 32 Bit beim PC). Bei Großrechnern findet man auch „krumme" Werte wie 50- oder 60-Bit-Darstellungen.

2.1 Darstellung von Daten

Wir betrachten in diesem Abschnitt sowohl gebräuchliche Formen der Darstellung von ganzen und Gleitkommazahlen als auch Codierungen von Zeichen, denen keine arithmetische Bedeutung zugeordnet ist.

Allgemeines zur Darstellung ganzer Zahlen

Es gilt, eine Darstellung für ganze Zahlen zu finden, so daß sich eine „vernünftige" Zuordnung zwischen Codierung und dem Wert der Zahl ergibt.
Man kann einige Forderungen an die Codierung stellen:

- Zunächst sollten alle positiven und negativen Zahlen eines Intervalls [–x; y] dargestellt werden können.
- Dabei sollten in etwa gleich viele positive wie negative Zahlen darstellbar sein.
- Die Darstellung sollte — abgesehen von wenigen Ausnahmen — eindeutig sein, das heißt, zwei unterschiedliche Codeworte beschreiben auch zwei unterschiedliche Werte.
- Eine wichtige Forderung ist die Durchführbarkeit von Rechenoperationen in der Codierung.
- Eine eher technische Forderung liegt darin, daß die Unterscheidung zwischen positiver und negativer Zahl möglichst am Anfang des Codeworts getroffen werden kann. Der Rest wird als Codierung des Stellenwerts aufgefaßt.

Die Stellenwertcodierung ist ein gebräuchliches Beispiel für eine „vernünftige" Zuordnung. Das bekannteste Beispiel ist das Dezimalsystem.

Man ordnet einem Codewort $(a_{(n-1)}, \ldots, a_i, \ldots, a_0)$ einen Wert zu, der sich ergibt als:

$$Wert(a_{(n-1)}, \ldots a_i, \ldots a_0) := \backslash I\backslash SU(i=0;n-1;\ a_i \cdot d^i)$$

Dabei bezeichnet man d als **Basis** und a_i als **Stellenwert**.
Es muß gelten: $0 \le a_i \le d-1$.

Gebräuchliche Basen sind: d = 2; 8; 10; 16. Das Zahlensystem wird entsprechend der durch die Basis gegebenen Codierung als **Binär-** (2), **Oktal-** (8), **Dezimal-** (10) oder **Hexadezimal-** (16) System bezeichnet. Die Werte für a im binären Fall schreibt man als „0" und „1". Diese **Binärzeichen** eignen sich besonders bei der rechnerinternen Darstellung: Man unterscheidet sie dann physikalisch durch fließenden Strom im Falle der „1" und nicht fließenden (schwachen) Strom im Falle der „0". Im Hexadezimalsystem notiert man für a: 0, 1, 2, 3, ... 9, A, B, ... F. Jede Hexadezimalziffer wird in binärer Darstellung durch vier Bits dargestellt. Im folgenden betrachten wir jedoch das Binärsystem.

Wir verwenden nun die folgende Stellenwertfunktion, die eine Unterscheidung zwischen negativer und positiver Zahl aufgrund des ersten Bits $a_{(n-1)}$ zuläßt:

$$W: \{0,1\}^n \rightarrow Z$$
$$W(a_{(n-1)}, \ldots a_i, \ldots a_0) := (a_{(n-1)}, S_{(n-2),0}), \text{ mit: } S_{j,i} := \backslash I\backslash SU(k=i;j;\ a_k \cdot 2^k)$$

Man bezeichnet $a_{(n-1)}$ als **Vorzeichenbit** und legt fest:

$$W(a_{(n-1)}, \ldots a_i, \ldots a_0) \ge 0, \quad \text{wenn } a_{(n-1)} = 0,$$
$$W(a_{(n-1)}, \ldots a_i, \ldots a_0) \le 0, \quad \text{wenn } a_{(n-1)} = 1.$$

Diese Codierung erlaubt eine Darstellung von maximal 2^n verschiedenen Zahlen. Da durch das erste Bit aber eine Unterscheidung zwischen positiven und negativen Zahlen getroffen wird und hier die „0" sowohl als positive als auch negative Zahl interpretiert wird, ist die betragsgrößte Zahl $2^{(n-1)}-1$.

In den folgenden Abschnitten werden drei Codierungsformen für ganze Zahlen eingeführt. Ihnen gemeinsam ist die Darstellung positiver ganzer Zahlen. Sie unterschieden sich in der Codierung negativer Zahlen und der „0", der damit zusammenhängenden Symmetrie und der Größe des Zahlenbereiches. Ferner sind die Rechenoperationen auf diesen Codierungen unterschiedlich zu handhaben. Als Rechenoperationen werden zunächst nur die Addition und Subtraktion vorgestellt. In einem weiteren Abschnitt über Gleitkommazahlen werden auch die Multiplikation und Division eingeführt.

Betrag- und Vorzeichendarstellung ganzer Zahlen (B+V)

Ausgangspunkt ist eine n-stellige Binärzahl der Form $(b_{(n-1)}, \ldots, b_0)$. Ferner wird die Summe $S_{j,i}$ des letzten Abschnitts verwendet.

Positive Zahlen

Eine positive Zahl liegt dann vor, wenn $b_{(n-1)} = 0$ ist. Dann ergibt sich der Wert der Binärzahl als:

$$W_{„B+V"}(b_{(n-1)}, \ldots, b_0) := + S_{(n-2), 0}.$$

Beispiel:

Sei n= 6.
Dann ist $W_{„B+V"}(001101) = + 13$.

Negative Zahlen

Bei negativen Zahlen ist das Vorzeichenbit auf 1 gesetzt: $b_{(n-1)} = 1$. Der Wert ist dann:

$$W_{„B+V"}(b_{(n-1)}, \ldots, b_0) := - S_{(n-2), 0}.$$

Beispiel:

Sei n= 6.
Dann ist $W_{„B+V"}(101110) = - 14$.

Eine negative betragsgleiche Zahl erhält man aus einer positiven, indem man das Vorzeichenbit invertiert, das heißt, von 0 auf 1 setzt.

Beispiel:

Sei $(b_2, b_1, b_0) = (011)$ gegeben mit $W_{„B+V"}(011) = +3$.
Dann ergibt sich für (111) der Wert $W_{„B+V"}(111) = -3$.

Addition

Die „B+V"-Darstellung ganzer Zahlenist ungünstig bezüglich der Addition. Eine Möglichkeit, diese durchzuführen, besteht darin, die Summanden a und b in die Darstellung im **1-Komplement** (übernächster Abschnitt) zu transformieren und dort die Addition durchzuführen. Die Summe s liegt dann im 1-Komplement vor und wird auf gleiche Weise zurück in die „B+V"-Darstellung überführt.

Transformation: „B+V" ↔ „1-Komplement":

Sei $b = (b_{(n-1)}, \ldots b_0)$ eine Binärzahl in der „B+V"-Darstellung. Wenn das Vorzeichenbit $b_{(n-1)} = 1$ ist, kippt man alle Bits bis auf das Vorzeichenbit. Formal führt man ein „Exklusives Oder" (Operationszeichen: $\oplus$) zwischen $b_{(n-1)}$ und b_i für alle Stellen $i \in \{0, \ldots (n-2)\}$ durch: Die invertierte Zahl $c = (c_{(n-1)}, \ldots c_0)$ im „1-Komplement" ergibt sich also durch:

$$c_{(n-1)} := b_{(n-1)} \text{ und für alle } i \in \{0, \ldots (n-2)\}: \; c_i := b_{(n-1)} \oplus b_i.$$

Diese Transformation funktioniert auch in umgekehrter Richtung, also vom 1-Komplement zur „B+V"-Darstellung.

Beispiel:

Sei b = (10110) mit $W_{„B+V"}(10110) = -6$.
Die Binärzahl im 1-Komplement lautet: c = (11001)

Die Addition im 1-Komplement wird im übernächsten Abschnitt erklärt.

Subtraktion

Die Subtraktion kann auf die Addition zurückgeführt werden, indem man a–b als a+(–b) auffaßt. Also muß b in eine negative Zahl umgeformt werden. Anschließend gehe man wie unter „Addition" geschildert vor.

Zahlenbereich

Der Zahlenbereich in der „B+V"-Codierung ist **symmetrisch** und reicht bei einer n-stelligen Binärzahl von $-(2^{(n-1)}-1)$ bis $+(2^{(n-1)}-1)$.
Daher hat die „0" zwei Darstellungen, nämlich $(b_{(n-1)}, ..., b_0) = (0, .., 0)$ und $(b_{(n-1)}, ... b_0) = (1, 0, ... 0)$.

Zahlenverlängerung

Will man eine n-stellige Binärzahl auf m Stellen (m>n) erweitern, so füllt man die Zahl nach dem Vorzeichenbit (an der Stelle (n–2)) mit Nullen auf.

Beispiel:

Sei n=5 mit a = (11010) gegeben.
Die auf m=7 Stellen erweiterte Zahl lautet: a´ = (1**00**1010).

2-Komplement-Darstellung ganzer Zahlen

Ausgangspunkt ist eine n-stellige Binärzahl der Form $(b_{(n-1)}, ... , b_0)$. Ferner wird die Summe $S_{j,i}$ des letzten Abschnittes verwendet.

Der Wert $W_{„2\text{-Kompl.}"}$ einer ganzen Zahl ergibt sich nach folgender Formel:

$$W_{„2\text{-Kompl.}"}(b_{(n-1)}, ... , b_0) := S_{(n-2),\,0} - b_{(n-1)} \cdot 2^{(n-1)}.$$

Im Falle, daß $b_{(n-1)} = 1$ ist, wird also durch Subtraktion eine negative Zahl errechnet.

Positive Zahlen

Positive ganze Zahlen haben demnach die gleiche Binärdarstellung im „2-Komplement" wie in der „B+V"-Codierung.

Negative Zahlen

Eine negative Zahl ergibt sich aus der obigen Formel, wenn $b_{(n-1)} = 1$ ist. Andererseits erhält man den Wert einer negativen Binärzahl in „2-Komplement"-Darstellung wie folgt:

Gegeben sei $b = (b_{(n-1)}, ... , b_0)$ mit $W_{„2\text{-Kompl.}"}(b_{(n-1)}, ... , b_0) < 0$.

1. Man kippe alle Bits von b, das heißt: $b_i := 1$, falls $b_i = 0$, $b_i := 0$, sonst.
   Es ergibt sich b_{neu}.
2. Eine „1" addieren: $b_{neu} := b_{neu} + 1_n$, mit $1_n = (c_{(n-1)}, ... , c_0) = (0, ... 0, 1)$.
3. Zugehörigen Wert in „B+V" bestimmen: $U := W_{„B+V"}(b_{neu})$.
4. Der Dezimalwert von b ist dann –U: $W_{„2\text{-Kompl.}"}(b) = -U$.

Diese Art der Berechnung versagt nur, wenn $(b_{(n-1)}, \ldots, b_0) = (1, 0, \ldots 0)$. Nach der zweiten Methode ergibt sich der Wert –0. Der Dezimalwert nach obiger Formel ist jedoch: $W_{„2\text{-Kompl.}“}(1, 0, \ldots 0) = -2^{(n-1)}$

Beispiel:

Sei n=7 und b = (b6, ... , b0) := (1010110).
Nach der Formel ergibt sich:

$$W_{„2\text{-Kompl.}“}(b_6, \ldots, b_0) = 1\cdot2^1 + 1\cdot2^2 + 1\cdot2^4 - 1\cdot2^6 = 22 - 64 = \mathbf{-42}.$$

Nach der zweiten Methode ergibt sich:

1. Kippen: b_{neu}:= 0101001,
2. „1" addieren: b_{neu}:= 0101010,
3. U:= $W_{„B+V“}(b_{neu}) = 1\cdot2^1 + 1\cdot2^3 + 1\cdot2^5 = 42$,
4. Dezimalwert ist $W_{„2\text{-Kompl.}“}(b) = -U = \mathbf{-42}$.

Addition

Zwei Binärzahlen $a = (a_{(n-1)}, \ldots, a_0)$ und $b = (b_{(n-1)}, \ldots, b_0)$ werden addiert, indem man sie (nach Schulmethode) komponentenweise addiert. Im Falle, daß die Summation der (n–1)-ten Stellen einen Übertrag liefert, wird dieser Gesamtübertrag ignoriert.

Beispiel:

Seien die Summanden a = (1101101) und b = (0011001) gegeben. Die Summe s ergibt sich dann wie folgt:

	1101101	$W_{„2\text{-Kompl.}“}(a) = -19$
	0011001	$W_{„2\text{-Kompl.}“}(b) = +25$
1	0000110	$W_{„2\text{-Kompl.}“}(0000110) = +6$.

↑ Übertrag wird ignoriert.

Subtraktion

Die Subtraktion kann auf die Addition zurückgeführt werden, indem man a–b als a+(–b) auffaßt. Also muß b in eine negative Zahl umgeformt werden. Anschließend gehe man wie unter „Addition" geschildert vor.

Zahlenbereich

Der Zahlenbereich im „2-Komplement" ist **asymmetrisch**. Er reicht bei einer n-stelligen Binärzahl von $-(2^{(n-1)})$ bis $+(2^{(n-1)}-1)$. Die „0" hat also nur eine Darstellung, nämlich: $(b_{(n-1)}, \ldots, b_0) = (0, \ldots, 0)$.

Zahlenverlängerung

Will man eine n-stellige Binärzahl auf m Stellen (m>n) erweitern, so füllt man die Zahl vorne (an der Stelle n) mit dem Vorzeichenbit auf.

Beispiel:

Sei n=5 mit a = (11010) gegeben.
Die auf m=7 Stellen erweiterte Zahl lautet: a´ = (**1**1**1**1010).

1-Komplement-Darstellung ganzer Zahlen

Ausgangspunkt ist eine n-stellige Binärzahl der Form $(b_{(n-1)}, \ldots, b_0)$.
Ferner wird die Summe $S_{j,i}$ des letzten Abschnittes verwendet.

Der Wert $W_{„1\text{-Kompl.}“}$ einer ganzen Zahl ergibt sich nach folgender Formel:

$$W_{„1\text{-Kompl.}“}(b_{(n-1)}, \ldots, b_0) := S_{(n-2),0} - b_{(n-1)} \cdot (2^{(n-1)} - 1).$$

Im Falle, daß $b_{(n-1)} = 1$ ist, wird also durch Subtraktion eine negative Zahl errechnet:

$$\begin{aligned} W_{„1\text{-Kompl.}“}(1, b_{(n-2)}, \ldots, b_0) &= S_{(n-1),0} - (2^n - 1) \text{ modulo } (2^n - 1) \\ &= S_{(n-2),0} + b_{(n-1)} \cdot 2^{(n-1)} - (2^n - 1) \\ &= S_{(n-2),0} + 2^{(n-1)} - (2 \cdot 2^{(n-1)} - 1), \text{ da } b_{(n-1)} = 1 \\ &= S_{(n-2),0} - (2^{(n-1)} - 1). \end{aligned}$$

Positive Zahlen

Positive ganze Zahlen haben die gleiche Binärdarstellung im 1-Komplement wie in der „B+V"-Codierung.

Negative Zahlen

Eine negative Zahl ergibt sich aus der obigen Formel, wenn $b_{(n-1)} = 1$ ist. Andererseits erhält man den Wert einer negativen Binärzahl in 1-Komplement-Darstellung wie folgt:

Gegeben sei $b = (b_{(n-1)}, \ldots, b_0)$ mit $W_{„1\text{-Kompl.}“}(b_{(n-1)}, \ldots, b_0) < 0$.

1. Man kippe alle Bits von b, das heißt: $b_i := 1$, falls $b_i = 0$, $b_i := 0$, sonst.
   Es ergibt sich b_{neu}.
2. Zugehörigen Wert in „B+V" bestimmen: $U := W_{„B+V“}(b_{neu})$.
3. Der Dezimalwert von b ist dann –U: $W_{„1\text{-Kompl.}“}(b) = -U$.

Hier entfällt der Schritt 2 aus der 2-Komplement-Darstellung.

Beispiel:
Sei n=7 und $b = (b_6, \ldots, b_0) := (1001010)$.
Nach der Formel ergibt sich:

$$W_{„1\text{-Kompl.}“}(b_6, \ldots, b_0) = 1 \cdot 2^1 + 1 \cdot 2^3 - 1 \cdot (2^6 - 1) = 10 - 63 = -53.$$

Nach der zweiten Methode ergibt sich:

1. Kippen: $b_{neu} := 0110101$,
2. $U := W_{„B+V“}(b_{neu}) = 1 \cdot 2^0 + 1 \cdot 2^2 + 1 \cdot 2^4 + 1 \cdot 2^5 = 53$,
3. Dezimalwert ist $W_{„1\text{-Kompl.}“}(b) = -U = -53$.

Addition

Zwei Binärzahlen $a = (a_{(n-1)}, \ldots, a_0)$ und $b = (b_{(n-1)}, \ldots, b_0)$ werden zu einer Summe $s = (s_n, \ldots, s_0)$ addiert, indem man sie (nach Schul-Methode) komponentenweise addiert. Im dem Fall, daß die Summation der (n–1)-ten Stellen einen Übertrag ($s_n = 1$) liefert, wird ein sogenannter ***End-Around-Carry*** durchgeführt. Die herausfallende Stelle ($s_n = 1$)wird wie in der 2-Komplement-Darstellung gestrichen, dann allerdings auf die bisherige Summe s addiert: $s := s + 1_n$, mit $1_n = (0, \ldots, 0, 1)$. Wenn im 1-Komplement ein Gesamtübertrag $s_n = 1$ entsteht, dann gilt, daß dieser sowohl den Wert $s_n \cdot 2^n$ als auch den Wert $s_n \cdot 2^0 = s_n \cdot 1$ hat, weil man „modulo $(2^n–1)$" rechnet, denn:

$$s_n \cdot 2^n = s_n \cdot (2^n–1) + s_n = s_n \text{ modulo } (2^n–1) = s_n \cdot 2^0 \text{ modulo } (2^n–1).$$

Beispiel:

Seien die Summanden a = (1101101) und b = (0011001) gegeben. Die Summe s ergibt sich dann wie folgt:

	1101101	$W_{„1\text{-Kompl.}"}(a) = –18$
	0011001	$W_{„1\text{-Kompl.}"}(b) = +25$
1	0000110	$W_{„1\text{-Kompl.}"}(0000110) = +6$ mit s = (0000110).

↑

Übertrag $s_n = 1$ zu s addieren:

0000111	$W_{„1\text{-Kompl.}"}(s) = 7$.

Subtraktion

Die Subtraktion kann auch hier wieder auf die Addition einer negativen Zahl zurückgeführt werden.

Zahlenbereich

Die Asymmetrie der 2-Komplement-Darstellung ist behoben. Der symmetrische Zahlenbereich reicht nun von $–(2^{(n-1)}–1)$ bis $+(2^{(n-1)}–1)$.

Zahlenverlängerung

Die Zahlenverlängerung im 1-Komplement funktioniert genauso wie im 2-Komplement: Vorne das Vorzeichenbit beliebig oft anfügen.

Überlaufproblematik

Wie die vorangehenden Abschnitte zeigten, kann es bei der Addition zu einem Überlauf kommen.

Beispiel:

Die Summe von a = (01010) und b = (01100) ist s = (10110). Im „2-Komplement" ist der Dezimalwert $W_{„2\text{-Kompl.}"}(s) = –10$. Das korrekte Ergebnis lautet aber: $W_{„2\text{-Kompl.}"}(a) + W_{„2\text{-Kompl.}"}(b) = +22$. Zwischen beiden „Ergebnissen" liegt eine Differenz von $32 = 2^n = 2^5$. Dies läßt sich damit begründen, daß im „2-Komplement" „modulo 2^n" gerechnet wird. Das Ergebnis $W_{„2\text{-Kompl.}"}(s) = –10$ ist also bis auf Vielfache von 2^n richtig.

Abhilfe gegen einen Überlauf

Man führe eine **Zahlenverlängerung** durch. Bei der 2-Komplement- und 1-Komplement-Darstellung verdoppelt man das Vorzeichenbit (siehe vorige Abschnitte). Jeder Summand beginnt dann mit zwei gleichen Bits. Sind die ersten beiden Bits der Summe verschieden, ist das Ergebnis sicherheitshalber zu prüfen.

Beispiel:

Verlängere a = (01010) zu a´ = (**0**01010) und b = (01100) zu b´ = (**0**01100).
Die Summe ist dann s = (**01**0110).
Eine Überprüfung ergibt, daß das Ergebnis korrekt ist.

Der Nachteil dieser Vorgehensweise liegt in einer **höheren Redundanz**; man verschenkt ein Bit und kann somit nur halb so viele Zahlen darstellen, wie die Bitwortlänge erlaubte. Vorteilhaft ist die Einfachheit dieser Methode.

Ein Kriterium für die Unmöglichkeit eines Überlaufs ist ein Vergleich der Vorzeichen beider Summanden: Ein Überlauf ist unmöglich, wenn die Vorzeichen unterschiedlich sind, oder formal:
Sind $a = (a_{(n-1)}, \ldots, a_0)$ und $b = (b_{(n-1)}, \ldots, b_0)$ die zwei Summanden, dann ist ein Überlauf genau dann unmöglich, wenn $a_{(n-1)} \neq b_{(n-1)}$. Anschaulich läßt sich dies dadurch begründen, daß die Summe einer negativen und einer positiven Zahl nicht den Zahlenbereich überschreiten wird.

Darstellung rationaler Zahlen

Die obigen Abschnitte haben die Darstellung von ganzen Zahlen eingeführt. Die zugehörigen Rechenoperationen sind die Addition und Subtraktion, wobei letztere sich auf die Addition zurückführen läßt.

Will man nun Zahlen dividieren, so reichen ganze Zahlen nicht aus. In diesem Abschnitt werden Darstellungsformen rationaler Zahlen eingeführt.

Festkommazahlen

Es gibt zwei Möglichkeiten der Festkommadarstellung:
Zu einer Binärzahl $b = (b_{n-1}, \ldots b_0, b_{-1}, \ldots, b_{-m})$ stellt man sich ein Komma „zwischen" b_0 und b_{-1} vor. Der Wert ergibt sich dann entsprechend der obigen Abschnitte. Ganze Zahlen haben in dieser Darstellung das gedachte Komma ganz rechts, das heißt, nach der Stelle b_{-m} in obiger Schreibweise.
Eine andere Möglichkeit besteht darin, den Zahlenbereich auf ein offenes Intervall (–1, +1) einzuschränken. Die Binärzahl hat die Form:

$$b = (b_0, b_{1,} \ldots, b_n),$$

wobei b_0 das Vorzeichenbit ist und der Dezimalwert von b der folgenden Formel genügt:

$$W_{„Festk."}(b) = + \text{ , falls } b_0 = 0,$$
$$W_{„Festk."}(b) = - \text{ , falls } b_0 = 1.$$

Die Werte –1 und 1 sind auf diese Weise nicht darstellbar, denn die größte darstellbare Zahl ist $1 - 1/2^n$.

Gleitkommazahlen

Abbildung 2.1 zeigt die Darstellung einer Gleitkommazahl.

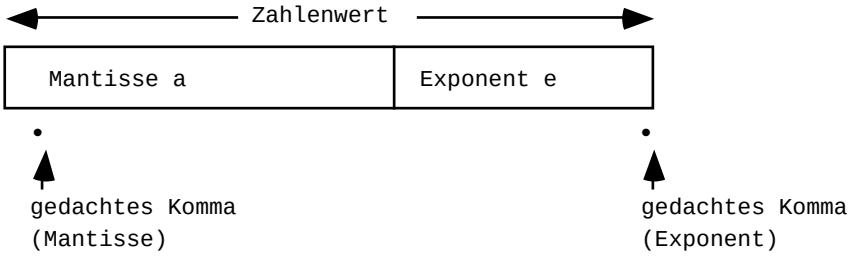


Abbildung 2.1: Codierung einer Gleitkommazahl

Der Darstellung einer rationalen Zahl durch Mantisse und Exponent liegt der folgende Satz für reelle Zahlen zugrunde.

Satz:

1) Jede Zahl z ist **darstellbar** in der Form:

$$z = a \cdot d^e$$

mit der Mantisse a, dem Exponenten e und
der Basis der Zahlendarstellung d.

2) Eine Zahl $z \neq 0$ ist **eindeutig darstellbar** als

$$z = a \cdot d^e,$$

wobei $\backslash F(1;d) \leq |a| < 1$.
Diese Darstellung heißt dann:
normalisierte Darstellung mit Mantisse a und Exponent e.

Beispiel:

Es soll die Dezimalzahl 5 in die normalisierte Darstellung zur Basis 2 gebracht werden:

$$\begin{aligned} 5 &= 5 \cdot 2^0 \\ &= 2{,}5 \cdot 2^1 \\ &= 1{,}25 \cdot 2^2 \quad (1.01000 \mid 0010) \\ &= 0{,}625 \cdot 2^3 \quad (0.10100 \mid 0011) \quad \textbf{normalisierte Darstellung} \\ (&= 0{,}3125 \cdot 2^4 \quad (0.01010 \mid 0100) \ldots) \end{aligned}$$

Somit hat die Dezimalzahl 5 die normalisierte Darstellung:

$5 = (2^{-1} + 2^{-3}) \cdot 2^3$ mit:
Mantisse a = (0.10100) und
Exponent e = (0011)
zur Basis 2,

denn $1/2 \leq |0{,}625| < 1$.

Dieses Beispiel legt die Basis 2 zugrunde. Es sind aber auch andere Basen möglich, wie zum Beispiel d=16 (hexadezimal). Intern wird jede hexadezimale Ziffer auf 4 Bits dargestellt. Man bezeichnet sie mit 0,1,2, ... 9, A, B, ... F, wobei die Binärzahl (0110) hexadezimal 6 ist und $(1110)_2 = E_{16}$. Hexadezimal sind größere Zahlen darstellbar als bei binärer Basis (d=2).

Für d=2 erhält man folgende größte, kleinste und betragskleinste Zahl, wenn man den Exponenten im 2-Komplement fünfstellig und die Mantisse siebzehnstellig darstellt:

Exponent:

größter:	01111,	$W_{„2\text{-Kompl.}“}(01111) = +15$
kleinster:	10000,	$W_{„2\text{-Kompl.}“}(10000) = -16$

Sei für $\alpha \cdot 2^x$ $\alpha \in [-1; +1)$ und $x \in [-16; +15]$.

Binäre Darstellung:

größte Zahl:	$(1\text{-}1/2^{16}) \cdot 2^{15}$
kleinste Zahl:	$-1 \cdot 2^{15}$
betragskleinste Zahl:	$1 \cdot 2^{-16}$

Die Zahlen liegen mit zunehmender Entfernung von Null weiter auseinander, wohingegen sie um die Null extrem dicht liegen, vorausgesetzt, der Zahlenbereich ist symmetrisch gehalten. Wegen der Binärinterpretation des Exponenten läßt sich dieser dichte Dezimalzahlen-bereich verschieben. Denkt man sich zum Beispiel den Exponenten um jeweils 8 größer (also $x \in [-8; +23]$), so ist der dichte Bereich ins Positive verschoben.

Die **Mantissenlänge** erweitert den Zahlenbereich nicht, sie erhöht die Genauigkeit der Darstellung. Irrationale Zahlen sind aber immer nur ungenau darstellbar.

Arithmetik bei normalisierten Gleitkommazahlen

Addition und Subtraktion

Bei der Addition zweier Zahlen $z_1 = a_1 \cdot d_1^{e_1}$ und $z_2 = a_2 \cdot d_2^{e_2}$ ist zu beachten, daß die Exponenten gleich sind.
Daher geht man wie folgt vor:

1. Exponenten anpassen (den kleineren an den größeren),
2. Operation (gemäß der vorangehenden Abschnitte) ausführen,
3. Postnormalisieren (Summendarstellung normalisieren).

Auf diese Weise kann es zu einem Genauigkeitsverlust (1.) und einer Teilauslöschung bei begrenzter Stellenzahl kommen.

Beispiel:

Seien $z_1 = (0.1101) \cdot 2^4$ und $z_2 = (0.1011) \cdot 2^2$ im „2-Komplement" gegeben.

1. Exponentenanpassung: $z_2 := (0.0010)\mathbf{11} \cdot 2^4$,
   wobei die herausfallenden Einsen auf die letzte Stelle gerundet werden: $z_2 := (0.0011)$.
2. Addition: $s := (0.1101) \cdot 2^4 + (0.0011) \cdot 2^4 = (1.0000) \cdot 2^4$
3. Postnormalisierung ergibt: $s := (0.1000) \cdot 2^5$, da
   $|W_{„2\text{-Kompl.}"}(1.0000)| = 1$.

Der sich ergebende Dezimalwert ist ungenau:

$$W_{„\text{Gleitk.}"}((0.1000) \cdot 2^5) = 0{,}5 \cdot 2^5 = \mathbf{16}.$$

Der korrekte Wert ist aber:

$$W_{„\text{Gleitk.}"}((0.1101) \cdot 2^4) + W_{„\text{Gleitk.}"}((0.1011) \cdot 2^2)$$
$$= (1/2 + 1/4 + 1/16) \cdot 2^4 + (1/2 + 1/8 + 1/16) \cdot 2^4 = 13 + 2{,}75 = \mathbf{15{,}75.}$$

Beispiel:

Die Zahl $z_2 = 0.9986 \times 10^{39}$ soll von $z_1 = 0.1002 \cdot 10^{40}$ subtrahiert werden.
Exakt ergibt sich:

$$(1.002 - 0.9986) \cdot 10^{39} = 0.0034 \cdot 10^{39} = 3{,}4 \cdot 10^{36} = \mathbf{0{,}34 \cdot 10^{37}}.$$

1. Anpassung von z_2 an den größeren Exponenten:
   z_2 wird zu $0.09986 \cdot 10^{40}$,
   aufgrund der begrenzten Stellenzahl wird gerundet: $z_2 := 0.0999$.
2. Die Subtraktion ergibt: $0.1002 \cdot 10^{40} - 0.0999 \cdot 10^{40} = 0.0003 \cdot 10^{40}$.
3. Nach Postnormalisierung: $\mathbf{0.3000 \cdot 10^{37}}$.

Im Vergleich zum exakten Wert liegt ein großer Fehler vor.

Im Extremfall wird die Mantisse durch Rundungsfehler zu Null und ist nicht mehr postnormalisierbar. Gleichzeitig kann der Exponent groß bleiben. In diesem Fall spricht man von einer sogenannten ***Dirty Zero***, da die Zahl dann Null ist, obwohl sich der echte Wert von Null unterscheidet.

Multiplikation und Division
Für die Multiplikation beziehungsweise Division eignet sich die Darstellung mit Mantisse und Exponent besser. Hier muß keine Exponentenanpassung stattfinden.
Man geht wie folgt vor:

1. Mantissen multiplizieren beziehungsweise dividieren
2. Exponenten addieren beziehungsweise subtrahieren
3. Postnormalisierung (wie oben)
   Hierzu reicht maximal ein Linksshift bei der Multiplikation beziehungsweise ein Rechtsshift bei der Division aus.

Formal:

$$a_1 \cdot d^{e_1} \cdot a_2 \cdot d^{e_2} = (a_1 \cdot a_2) \cdot d^{\,e_1 + e_2}, \text{ mit } \backslash F(1;d^2) \leq \backslash B\backslash BC\backslash |(a_1 \cdot a_2) < 1$$

$$\backslash F(a_1 \cdot d^{e_1}; a_2 \,.\, d^{e_2}) = \backslash F(a_1; a_2) \cdot d^{\,e_1 - e_2}, \text{ mit: } \backslash B\backslash BC\backslash |(\backslash F(a_1;a_2)) < d$$

Beispiel:

$$\backslash F(-0.9 \cdot 10^5; 0.75 \cdot 10^8) = -1.2 \cdot 10^{-3} = -0.12 \cdot 10^{-2}$$

Der letzte Schritt entsteht durch Postnormalisierung, indem die Mantisse nach rechts geshiftet und der Exponent korrigiert wird.

Darstellung von Zeichen

In diesem Abschnitt werden Darstellungsformen von Zeichen eingeführt. Es werden hier keine Forderungen an die Interpretierbarkeit (wie Rechenoperationen) dieser Zeichen gestellt.
Zeichen sind zum Beispiel **Buchstaben**, **Ziffern**, **Sondersymbole** (anderer Alphabete) und **Steuerzeichen**. Zeichen werden als Folge von Bits (Bitstring) codiert. Ein Wort (Speicherelement, siehe Einleitung von Kapitel 2) enthält meist mehrere Zeichen in codierter Form.
Es gibt zwei grundsätzlich unterschiedliche **Codierungsarten**, die in den folgenden Abschnitten vorgestellt werden. Die zugrundeliegende Frage ist hierbei: Wieviele Bits benötigt man, um ein Zeichen eindeutig darzustellen?

Codierung mit Umschaltsymbolen

Bei dieser Codierung gibt es keine eindeutige Zuordnung zwischen Zeichen und dem codierenden Bitstring. Man verwendet kurze Bitstrings und kann hiermit wenige Zeichen codieren. Um weitere Zeichen darstellen zu können, faßt man diese als eine weitere Gruppe von Zeichen auf. Ferner verwendet man Umschaltsymbole, die anzeigen, welcher Gruppe die Zeichen angehören, die die nächsten Bits des Bitstrings codieren.

Beispiel:
Darzustellen seien die 26 Buchstaben des Alphabets (o.B.d.A. in Form von Großbuchstaben) und die zehn Ziffern sowie weitere Sonderzeichen.

Will man jedes Zeichen durch fünf Bits codieren, so kann man $2^5 = 32$ verschiedene Zeichen codieren. Dies reicht offenbar nicht aus, um der Anforderung gerecht zu werden.
Nach der obigen Methode führe man zwei Umschaltsymbole (11111) und (11010) ein, die anzeigen, daß von jetzt ab nur Buchstaben beziehungsweise nur Ziffern oder Sonderzeichen im Bitstring folgen. Trifft man also auf ein Umschaltsymbol, so werden die im Bitstring folgenden 5-Bit-Folgen entsprechend dem vorangehenden Umschaltsymbol interpretiert. Diese Codierung erlaubt die Darstellung von 30 Buchstaben sowie 30 Ziffern und Sonderzeichen, wenn man die beiden Umschaltsymbole berücksichtigt.
Eine Erweiterung wäre die Unterscheidung zwischen Groß- und Kleinschreibung. Hierzu müßten zwei weitere weitere Umschaltsymbole eingeführt werden: Umschalten Groß- nach Kleinschreibung und Klein- nach Großschreibung bei Buchstaben. Günstiger ist in diesem Fall die Einführung von insgesamt drei Umschaltsymbolen für Großbuchstaben, Kleinbuchstaben und Sonderzeichen bzw. Ziffern.

Zu beachten ist bei dieser Codierung, daß ohne Kenntnis des im Bitstring zuletzt vorgekommenen Umschaltsymbols einer Bitfolge kein eindeutiges Zeichen zugeordnet werden kann.
Dem Vorteil der Darstellbarkeit vieler Zeichen als kurze Bitfolgen steht die Verlängerung des Bitstrings durch die eingestreuten Umschaltsymbole gegenüber. Zudem verkleinern viele Umschaltsymbole die Gruppen der tatsächlich zu codierenden Zeichen.

Eine Aufgabe der Informationstheorie ist die Darstellung einer gegebenen Informationsmenge mit möglichst wenigen Bits.
Eine weitere Idee liegt darin, häufig vorkommende Zeichen oder Zeichenkombinationen wie zum Beispiel „e" oder „en" mit kurzen Bitfolgen zu codieren, für seltene Zeichen wie „q" verwendet man längere Folgen.

Die **Huffman-Codierung** basiert auf der relativen Häufigkeit des Gebrauchs von Zeichen. Die Häufigkeit deutscher Buchstaben ergibt abnehmend das Kunstwort „**E N R I S T U D A**".

Ein eindeutiger Bitstring pro Zeichen

Bei dieser Codierung ordnet man jeder n-stelligen Bitfolge genau ein Zeichen zu. Somit sind 2^n verschiedene Zeichen codierbar.

Beispiel:

Die Darstellung der 26 Großbuchstaben, 26 Kleinbuchstaben und 10 Ziffern ist auf sechs Bits möglich.

Standardzeichensätze verwenden sieben beziehungsweise acht Bits (ein Byte) pro Zeichen. Zu nennen ist der **ASCII-Code** (*American Standard Code of Information Interchange)*, welcher sieben Bits verwendet. Der **EBCDIC-Code** (*Extended Binary Coded Decimal Interchange Code*) verwendet acht Bits. In Tabellen lassen sich diese Codierungen nachschlagen.
Die Darstellung von ASCII-Zeichen auf Bytebasis (Erweiterung auf 8 Bits) ermöglicht die Einrichtung einer **Übertragungskontrolle**. In der 8-stelligen

Bitfolge (b_7, ... , b_0) codieren die Bits b6, ... , b0 das Zeichen, während das sogenannte **Parity-Bit** b_7 die Zahl der Einsen \I\SU(i=0;6; b_i) auf eine gerade Zahl ((b_0 + ... + b_7) modulo 2 = 0) ergänzt. Für die Folge (b_6, ... , b_0) = (001101) ist b_7 = 1. Ein **Bitfehler** ist definiert als „Kippen" eines Bits. Ein solcher Fehler ist dadurch erkennbar, daß die Summe der b_i (modulo 2) gleich eins ist. Somit sind „Doppelfehler", das heißt Kippen von zwei Bits einer Bitfolge, nicht erkennbar. Erkennbar sind: 1-Bit-, 3-Bit-, ... (2n + 1)-Bit-Übertragungsfehler.

Darstellung von Dezimalziffern

Zur Darstellung der zehn Ziffern 0, 1, ... 9 benötigt man mindestens vier Bits, da mit drei Bits nur 2^3 = 8 Zeichen codierbar sind.

BCD-Code

Die Ziffern werden wie folgt im **BCD-Code** (*Binary Coded Decimal*) dargestellt: 0 durch 0000, 1 durch 0001, ... und 9 durch 1001, wobei sechs Redundanzen 1010, ... 1111 auftreten.

In ASCII und EBCDIC sind die Ziffern so dargestellt, daß sie bezüglich der Bits $b_3b_2b_1b_0$ gerade den BCD-Code haben. Bei aufeinanderfolgenden Dezimalziffern ist eine Unterdrückung von $b_7b_6b_5b_4$ möglich, womit eine dichtere Packung erreicht werden kann.

3-Excess-Code

Der redundante Bereich wird hier auf die Bitfolgen (0000), (0001) und (0010) sowie auf (1101), (1110) und (1111) verteilt. Eine Dezimalzahl i wird hier also im BCD-Code als Dezimalzahl (3 + i) dargestellt.

0000;0001;0010 }	redundant
0011	codiert die „0"
0100	codiert die „1"
0101	codiert die „2"
:	
1100	codiert die „9"
1101;1110;1111 }	redundant.

Ein Vorzug dieser Darstellung ist die Möglichkeit, eine Ziffer a in die komplementäre Ziffer b durch einfaches Kippen der Bits zu überführen, wobei a komplementär zu b ist, wenn a+b = 9.

Gray-Code

Der Gray-Code ist formal folgendermaßen definiert:
Eine Abbildung aller Dezimalzahlen i mit $0 \le i \le 2^n-1$ für $n \ge 1$ durch Bitfolgen (Bitworte) $W^{(n)}(i)$ der Länge n sei durch Induktion über n wie folgt erklärt:

1.) $W^{(1)}(0) := 0$ und $W^{(1)}(1) := 1$,

2.) $W^{(n+1)}(i) :=$ \B\LC\{(\A\AL\CO2\HS9 (0$W^{(n)}$(i);falls $0 \le i \le 2^n - 1$;1$W^{(n)}(2^{(n+1)} - 1 - i)$;falls $2^n \le i \le 2^{(n+1)} - 1$))

Anschaulich konstruiert man eine Folge von Gray-Code-Ziffern, indem man einen Block von untereinander stehenden Ziffern nach unten spiegelt und vor die Binärzahlen des oberhalb der Spiegelachse befindlichen Blocks „0" und unterhalb der Spiegelachse „1" schreibt:

;0;0;0;:;0 Block A	;	;1;:;1;1;1 Block A;gespie-;gelt

Beispiel:

Für n = 2 Bits lange Worte $W^{(2)}$ ergibt sich:

;0;0 ;0;1	;0;1	;	;1;1 ;1;0	;2;3

Für n = 3 Bits lange Worte $W^{(3)}$ erhält man:

;0;0;0;0 ;00;01;11;10	;0;1;2;3	;	;1;1;1;1 ;10;11;01;00	;4;5;6;7

Das Charakteristikum dieser Codierung besteht darin, daß sich beim Übergang von der Dezimalzahl i nach (i+1) die zugehörigen Binärzahlen im Gray-Code an genau einer Stelle unterscheiden.

Erkennung und Behebung von Bitfehlern

Die Behandlung von Bitfehlern spielt insbesondere im Bereich der Datenkommunikation eine große Rolle, da die Übertragung von Bits häufig durch äußere Einflüsse beeinflußt und gestört wird. Daher hat man Codes eingeführt, durch die man in der Lage ist, auftretende Bitfehler

- zu erkennen
- zu erkennen und zu beheben.

Bei einer entsprechenden Codierung verwendet man aus der Menge aller möglichen (binären) Folgen der Länge n Folgen mit speziellen Eigenschaften, also eine Teilmenge der möglichen Codeworte.

Ein einfaches Beispiel für einen fehlererkennenden und -behebenden Code ist das Hinzufügen von Paritätsbits. Diese zusätzlichen Bits werden gerade so gewählt, daß sie sich mit der Gruppe der Bits, auf die sie sich jeweils beziehen, derart ergänzen, daß sich eine gerade Anzahl von Einsen pro Gruppe ergibt. Analog zu dieser geraden Parität kann man natürlich auch ungerade Parität vereinbaren. Man unterscheidet Längsparität und Querparität, je nachdem, ob es sich bei der Gruppe um eine aufeinanderfolgende Bitreihe handelt oder aber um alle i-ten Positionen innerhalb einer größeren Anzahl von Bitreihen. Gemeinsam angewendet bieten Längs- und Querparität ein Mittel, einzelne Bitfehler zu lokalisieren und zu korrigieren. Bis zu drei Bitfehler werden sicher erkannt. Es kann allerdings in "pathologischen Sonderfällen" und einer darüber hinausgehenden Häufung von Fehlern dazu kommen, daß diese nicht erkannt werden.

Beispiel für gerade Parität:

10**1** 11001	1
00101000	0
10010001	1
Querparität	

Längsparität

Wird nun aus der fettgedruckten Eins aufgrund eines Übertragungsfehlers eine Null, so kann dieser Fehler anhand der zugehörigen Längs- und Querparitätsbits erkannt und behoben werden. Kippen die fettgedruckten Bits des folgenden Beispiels, so wird dies nicht erkannt:

10**1** 11**0** 01	1
00**1** 01**0** 00	0
10 01 00 01	1
Querparität	

Längsparität

Hamming-Distanz

Zwischen 2 binären Folgen kann man ein Abstandsmaß definieren, welches **Hamming-Distanz** genannt wird. Darunter versteht man de Anzahl der Stellen, an denen sich die Folgen unterscheiden. Formal:
Seien $a = (a_1,...,a_n)$ und $b = (b_1,...,b_n)$ binäre Folgen der Länge n. Dann ist

$$d(a,b) = d((a_1,...,a_n), (b_1,...,b_n)) := \sum_{i=1}^{n} |a_i - b_i|$$

die Hamming-Distanz der Folgen. Je größer d(a,b) ist, desto mehr "Störungen" sind also notwendig, um a in b zu verwandeln.
Die Hamming-Distanz eines Codes wird definiert als

$$HD = \min (d,(a,b)), \quad \text{wobei a und b zulässige Codeworte und } a \neq b.$$

Bei der Paritätsbitcodierung gilt also HD = 2. Im Extremfall sind die zulässigen Codewörter {(0,...,0);(1,...,1)}, somit gilt hier HD=n, wenn n die Länge der Wörter beschreibt.
Wenn ein Code eine Hammingdistanz von 2t+1 hat, dann gibt es um jedes Codewort einen "Einzugsbereich" der Größe t.

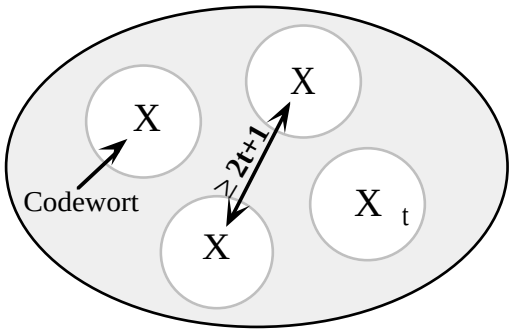


Abbildung 2.2: Hammingdistanz

Die Folgen innerhalb der t-Kugel entsprechen dabei Störungen des Codeworts von maximal t Bits. Geht man davon aus, daß Störungen weniger wahrscheinlich sind als korrekte Übertragungen, so kann man eine empfangene Folge dem nächstliegenden Codewort zuordnen (*Maximum Likelihood Decision*). Damit hat man die Möglichkeit, bis zu t Bitfehler zu korrigieren. Treten jedoch mehr als t Bitfehler auf, so wird die Entscheidung fehlerhaft.
Bei Codes der Hammingdistanz 2t sind bis zu t-1 Bitfehler erkennbar und korrigierbar. t Bitfehler sind erkennbar, aber nicht mehr korrigierbar. Das einfachste Beispiel hierfür ist wieder die Paritätsbitcodierung mit t=1.

Beispiel:

Es werde der folgende Code definiert:

A	00000000
E	11000011
R	00000111
S	11100000
T	00111010
U	11111111

Dann ist die Hamming-Distanz des Codes 3. Somit gilt hier t=1, es können also 2-Bit-Fehler erkannt und 1-Bit Fehler korrigiert werden.

2.2 Darstellung von Befehlen

Die Speicherung eines Befehls findet meist in einem Wort statt, manchmal auch in einem Halbwort oder Doppelwort. Die Interpretation einer Bitfolge setzt in einem Speicherwort eine Strukturierung voraus, die den gewünschten Befehl identifiziert.

Aufbau eines Befehlswortes

Den Aufbau eines Befehlswortes erkennt man in der folgenden Abbildung:

Abbildung 2.3: Struktur eines Befehlswortes

Format

Das Format legt einerseits die Strukturder **Adreßcodierung** fest; es gibt an, an welcher Binärstelle die Adressenbeginnen und wie lang sie sind. Andererseits legt es Regeln zur Befehlsausführung fest, dasheißt, die Zahl der Operanden, Vorrangregelungen oder etwa implizite Operanden wie zumBeispiel den Akku oder Stack.

Bemerkung: Oft ist das Format implizit durch den Opcode gegeben.

Opcode

Der Opcode definiert die auszuführende Operation. Aufgrund seines Codes kann zum Beispiel ein Holebefehl (Code sei 001011) von einem Additionsbefehl (011011) unterschieden werden.
Man unterscheidet Operationen nach ihrer Stelligkeit, das heißt, anhand der zu ihrer Ausführung benötigten Operanden.

Beispiel:

Nullstellige Operation: HALT (Stop der Programmausführung)
Einstellige Operation: a := –b
Zweistellige Operation: a := b op c
Mehrstellige Operation: Vektoroperation

Adreßteile

Man unterscheidet zwei Typen von Adressen:
Operandenadressen geben die Quell- und Zieladressen der Operanden an. Dabei kann die Adresse zum Beispiel eine Konstante sein.

Beispiel:

a := b + c
Hier sind b und c Quelladressen, und a ist eine Zieladresse.

Die **Folgebefehlsadresse** zeigt an, an welcher Adresse sich die nach diesem Befehl auszuführende Instruktion befindet.

Die Quell- und Zieladressen können fehlen, wenn sie sich auf spezielle Register oder auf den Stack beziehen (dies wird aus dem Opcode ersichtlich). Die Folgebefehlsadresse fehlt, wenn von der sequentiellen Befehlsausführung nicht abgewichen wird.

Anhand der Operanden-Zahl lassen sich Befehlsstrukturen unterscheiden (Abbildung 2.4). Die Bezeichnung (x+y)-Befehl bedeutet, daß der Befehl x Operandenadressen (ohne Register) und y Folgebefehlsadressen verwendet.

Befehlsstruktur	Adressen	Wirkung (Beispiel)	Folgebefehl
3 + 1	a, b, c, d	ρ(a) := ρ(b) op ρ(c)	d
3 + 0	a, b, c	ρ(a) := ρ(b op ρ(c)	β + 1
2 + 1	a, b, d	ρ(a) := ρ(b) op ρ(a)	d
2 + 0	a, b	ρ(a) := ρ(b) op ρ(a)	β + 1
1 + 1	a, d	α := α op ρ(a)	d
1 + 0	a	α := α op ρ(a)	β + 1
0 + 1	d	goto d	d
0 + 0	keine	Stackbefehl	β + 1

Abbildung 2.4: Befehlsstrukturen

Bezeichnungen:
α steht für den Akkumulator(-inhalt),
β bezeichnet den Befehlsfolgezähler, und
ρ(i) ist der Wert (Datum) in Speicherzelle i

Bemerkung:
Häufig benutzt und ökonomisch (da nur auf maximal eine Adresse zugegriffen werden muß) sind (1+0)-Befehle und (0+0)-Adreßbefehle.

Beispiele eines (1+0) -Befehls sind:

α := ρ(i)	Opcode 1, Operand i
α := α + ρ(j)	Opcode 2, Operand j
ρ(k) := α	Opcode 3, Operand k

Dabei spricht man von **Mnemonischer Darstellung** ("Assemblerbefehl") und dem **Quellcode,** bzw. von **Binärer Darstellung** ("Objektform") und **Objektcode.**
Der Unterschied zwischen Quellcode und Objektcode liegt unter anderem in der besseren Lesbarkeit der Programme im Quellcode, während die Darstellung im Objektcode „maschinennäher" ist.
Ein **Assembler** ist ein Programm, welches Quellcode in Objektcode umwandelt. Mit diesem Begriff identifiziert man auch häufig die **Assemblersprache**.

(2+y)-Adreßbefehle sind aufwendiger und ungebräuchlich, sie lassen sich außerdem in eine Folge von (1+y)- und (0+1)-Adreßbefehlen umwandeln.

Beispiel:
Der (2 + 1)-Befehl „α: = ρ(i) - ρ(k); goto h" läßt sich durch folgende Sequenz simulieren:

α := ρ(i)	(1+0)-Befehl
α := α - ρ(k)	(1+0)-Befehl
goto h	(0+1)-Befehl

Außerdem gibt es den **„Stackbefehl".** Hierzu sei der **Stack** kurz vorgestellt. Es handelt sich um eine Datenstruktur, die nach dem **LIFO-Prinzip** (*Last In First Out*) organisiert ist. Ein Stack ist ein Stapel von Informationseinheiten, in die Informationen (Elemente) abgelegt werden. Dabei kann nur auf das oberste Element zugegriffen werden. Es gibt zwei Operationen zur Veränderung des Stacks: **Push** legt ein neues Element auf den Stack, das heißt, an die oberste Position (τ, *Top of Stack*). **Pop** entnimmt das oberste Element, wobei das entnommene Element ausgegeben wird. Es gibt auch die Variante, daß dieses Element nicht explizit angegeben wird; in diesem Fall verwendet man zusätzlich die Operation **Top**, welche es erlaubt, das oberste Element des Stacks zu lesen (dabei wird es aber nicht entfernt).

Ein Stack wird als Folge σ = (σ(1), σ(2), ... σ(τ)) (Abbildung 2.5) notiert.

τ	$\sigma(\tau)$
:	:
i	σ(i)
:	:
2	$\sigma(2)$
1	$\sigma(1)$

Abbildung 2.5: Stack

Beispiel: Simulation von Stackbefehlen

Der Push-Befehl vollzieht folgende Instruktionen:

$$\tau := \tau + 1$$
$$\sigma(\tau) := \alpha$$

Pop läßt sich folgendermaßen ausdrücken:

$$\alpha := \sigma(\tau)$$
$$\tau := \tau - 1$$

Sei nun **Stackadd** die Operation, welche die obersten Stackelemente addiert, eliminiert und die Summe als oberstes Element ablegt:

$$\alpha := \sigma(\tau)$$
$$\tau := \tau - 1$$
$$\sigma(\tau) := \alpha + \sigma(\tau)$$

Anwendung finden Stacks bei der Auswertung arithmetischer Ausdrücke. Die Ausführung arithmetischer Operationen in Infixnotation (a op b) hängt von Präzedenzregeln und der Assoziativität der Operationen ab, wenn man von einer Klammerung absieht. Die **Umgekehrt Polnische Notation** (UPN) legt die Auswertung von Ausdrücken eindeutig durch die Reihenfolge der Operanden und Operationen fest. Bei dieser sukzessiven Auswertung im UPN-Ausdruck geht man von links nach rechts in Verbindung mit Push, Pop und Stackϕ vor, wobei $\phi \in$ {add, sub, mult, div, exp} ist. Zur Auswertung eines Infixausdrucks wandelt man diesen zunächst in einen UPN-Ausdruck um. Hier stehen die Operationen hinter den Operanden.

Beispiel:

Sei der Infix-Ausdruck x: = a + [(b – c) • d] $\uparrow$ (e • f) – g gegeben.
In UPN: x: = a b c – d • e f • $\uparrow$ + g –.
Stackoperationen (zu Beginn sei der Stack leer):

Push a
Push b
Push c
Stacksub
Push d
Stackmult

```
Push e
Push f
Stackmult
Stackexp
Stackadd
Push g
Stacksub
Pop x
```

Befehlstypen

Man unterscheidet folgende drei Befehlstypen nachihren unterschiedlichen Aufgaben:

a) Datentransfer

Für den Datentransfer gibt es den Ladebefehl $\alpha := \rho(i)$, der Daten aus dem Speicher in den Akkumulator holt, und den Befehl $\rho(j) := \alpha$ zum Speichern für die umgekehrte Richtung.

b) Arithmetische und logische Operationen

Beispiel für eine arithmetische Operation ist die Addition $\alpha := \alpha + \rho(i)$.
Der Shiftbefehl zum Schieben von Bits in einem Speicherwort kann ebenfalls als arithmetische Operation aufgefaßt werden. Ein Shift nach links verdoppelt den Wert (wenn man von Überträgen absieht).

Beispiel:

SHR α, i führt zu einem Rechts-Rund-Shift des Akkumulatorinhalts um i Bits:

SHR, 2, (0110111) = (1101101).

c) Steuerbefehle

Steuerbefehle sind:

- ein unbedingter Sprung goto j
- ein bedingter Sprung if <Bedingung> then goto k
  <Bedingung> ist zum Beispiel: „$\alpha < 0$" oder „ $\alpha = \gamma$".
- ein Vergleich und Überspringen der nächsten Operation (Skip) bei Gleichheit
- ein Unterprogrammaufruf
  call <Unterprogrammname/-adresse>
- ein Unterprogrammrücksprung return

Befehlssätze

Eine Menge von Befehlen eines Rechnermodells bezeichnet man als **Befehlssatz**. Die meisten Befehle sind durch (eine Folge von) anderen Befehlen ersetzbar („simulierbar").

Es gibt nun die Tendenz zu möglichst großen Befehlssätzen mit komplexen Operationen. Hier lassen sich kurze Programme bei Ausnutzung des vollen Befehlssatzes schreiben, denn es ist unnötig, umfangreichere Befehle durch eine Folge von weniger mächtigen Befehlen zusammenzusetzen. Ein Rechner, der dieser Tendenz entspricht, ist der *Complete Instruction Set Computer"* **(CISC)**.
Die **RISC**-Architektur (*Reduced Instruction Set Computer*) verfolgt gerade die umgekehrte Tendenz. Kleinere Befehlssätze lassen eine effizientere und schnellere Realisierung des Rechners zu (es werden kürzere Leitungen zur Hardwarerealisierung benötigt, wodurch eine höhere Taktfrequenz möglich ist).
Ferner geht man davon aus, daß komplexe Befehle selten eingesetzt werden und bei Bedarf simuliert werden können; bei Beachtung des „seltenen" Eintretens ist der Leistungsverlust gering. Die schnelle Realisierung der häufigen Grundoperationen führt im allgemeinen zu einer Leistungsverbesserung.

Beispiel für eine Simulation:

Ein bedingter Sprung soll durch einen „Vergleich und Skip"-Befehl ersetzt werden.
Die Wirkung von „Skip α+" besteht darin, den nächsten Befehl zu überspringen, wenn $\alpha \geq 0$.
Zu simulieren ist: if $\alpha < 0$ then goto r
Simulation durch: [skip α+; goto r]; falls also $\alpha \geq 0$ ist, wird „goto r" übersprungen.

Verwendung zusätzlicher Register und Registerbefehle

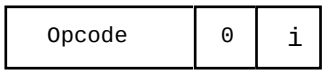
Zusätzlich zum Akkumulator können weitere Register R_0, R_1, ... zu speziellen Zwecken benutzt werden, zum Beispiel als Indexregister γ.
In **Registerbefehlen** können bis zu drei dieser Register angesprochen werden:

0-Registerbefehle

Hierzu zählen die bisherigen Befehle. Im Falle der Verwendung des Akkumulators (Register) wird dieser nicht explizit genannt, sondern durch den Opcode implizit angesprochen.

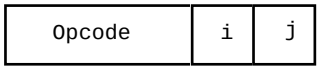
1-Registerbefehle

Beispiel eines solchen Befehls ist die Besetzung des Indexregisters zur indizierten Adressierung: $\gamma := \rho(i)$. Der Opcode hat die Form:



2-Registerbefehle

Ein Beispiel ist die Anwendung von Rechenoperationen auf Registerinhalte: $R_i := R_i - R_j$ oder die Zuweisung (Laden) von Registerinhalten: Load R_i, R_j, mit der Wirkung: $R_i := R_j$. Der Opcode ist folgendermaßen aufgebaut:

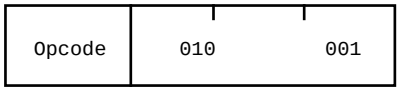


3-Registerbefehle

Analog läßt sich auch mit drei Registern rechnen: $R_i := R_j + R_k$.
Die Verwendung von 3-Adreßregisterbefehlen ist sinnvoll, weil diese Befehle schnell ausführbar sind und kurze Adressen haben.

Beispiel:

Seien 8 Register R0, ... R7 gegeben.
Die Registernummern werden auf drei Adreßteile mit je 3 Bits codiert, also insgesamt auf $3 \cdot 3 = 9$ Bits.
Dementsprechend ist der Opcode wie folgt aufgebaut:



2.3 Adressierung

Während der letzte Abschnitt die Darstellung von Informationen (Daten und Befehlen) behandelte, sollen hier Techniken vorgestellt werden, wie man diese Informationen im Speicher ansprechen (adressieren) kann.

Direkte Adressierung (absolute Adressierung)

Bei der direkten Adressierung wird eine Speicherzelle durch Angabe der **Adresse** adressiert. Hat diese eine Länge von k Bits, so können 2^k Adressen angesprochen werden.
Man notiert diese direkte Adressierung in der Form: $\alpha := \rho(100)$, wobei hier beispielhaft der Akkumulator den Inhalt der Speicherzelle 100 erhält.
Der Vorteil dieser Technik liegt im einfachen und übersichtlichen Zugriff auf den Speicher.
Ein schwerwiegender Nachteil besteht aber darin, daß ein zu kleiner Adreßraum ansprechbar ist. Außerdem ist der Anteil der Adresse am Befehlswort relativ hoch.

Indirekte Adressierung durch Basisregister und Relative Distanz

Diese Technik behebt die Nachteile der direkten Adressierung. Abbildung 2.6 zeigt die Funktionsweise.
Ein Befehl ist folgendermaßen aufgebaut:

Dabei stehen für die Nummer des Basisregisters m Bits und für die relative Distanz r Bits zur Verfügung. Ist m = 3, so können die Register R_0, ..., R_7 der Länge h als Basisregister fungieren.
Die angesprochene Adresse ergibt sich als folgende Summe:

(Basisadresse des Bereichs für R_i)
+ (2^r · Inhalt von R_i)
„Einteilung des Speicherblocks für R_i in 2^h Blöcke der Größe 2^r"
+ j „Relative Distanz"

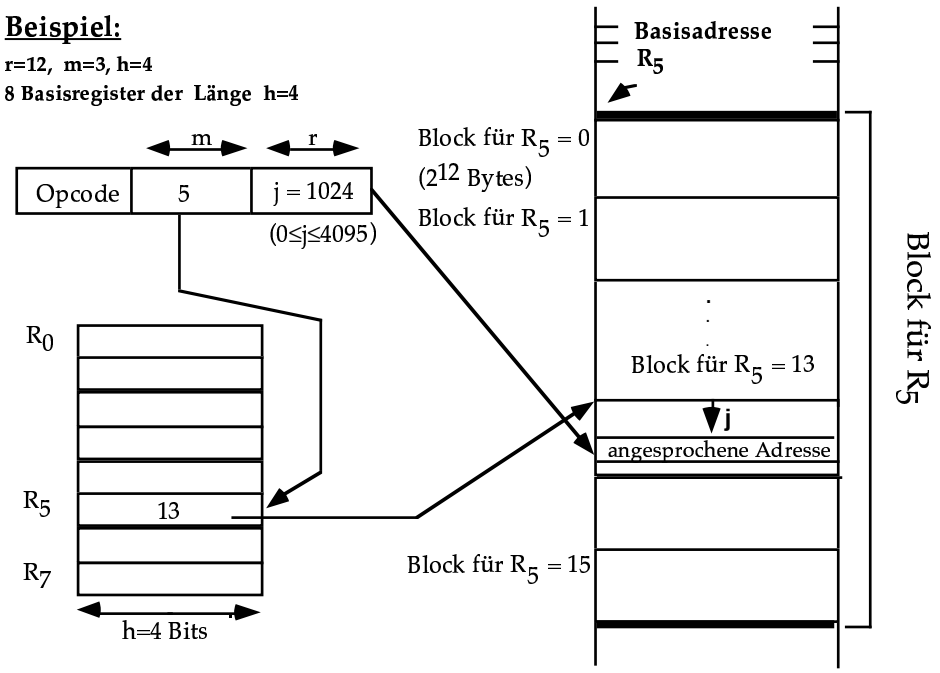


Abbildung 2.6: Indirekte Adressierung mit Basisregister und relativer Distanz

Die **Größe des adressierbaren Speichers** ergibt sich somit aus:

Registerzahl:	2^m	m = 3	-> 8 Register
Registerinhalt:	2^h	h = 4	-> 16 Blöcke pro Register
Rel. Verschiebung:	2^r	r = 12	-> $0 \le j \le 4095$

Es sind demnach $2^m \cdot 2^h \cdot 2^r$ Speicherzellen mit einer m+r Bit langen Adresse ansprechbar (im Beispiel also 2^{19}). Mit direkter Adressierung sind dagegen nur 2^{m+r} Adressen ansprechbar. Die Wortlänge des Befehls wurde eigentlich um h Bits verlängert durch den Zugriff auf das h-Bit-Register R_i.
Durch Verwendung „vernünftig großer" Registerlängen ist ein fast unbeschränkt großer Speicherbereich auf diesem Wege adressierbar.

Bemerkung:
Bisher war die Basisadresse fest. Durch Änderung dieser Adressen ist eine weitere Flexibilisierung möglich.

Indexregister γ zur indirekten Adressierung

Sei γ ein Indexregister. Durch geeignete Belegungen von γ lassen sich praktisch beliebige Adressen angeben. Im Zusammenhang mit diesem Register kann man zum Beispiel folgende Befehlstypen verwenden:

- $\alpha := \rho(\gamma)$
- $\rho(\gamma) := \alpha$
- $\gamma := \gamma - 1$

Anwendung findet das Indexregister bei der Bearbeitung von linearen Listen, Schleifen und ähnlichem ohne Änderung des Adreßteils eines Befehls.

Beispiel:

```
            γ := 1000;          Vorbesetzung
ANFANG:     .
            .                   Andere Operationen denkbar
            .
            α := ρ(γ)           Nacheinander werden ρ(1000),
            .                   ρ(999), ..., ρ(1) angesprochen
            .
            .
            γ := γ – 1
            if γ > 0 then goto ANFANG
```

Indexregister mit Adreßmodifikation

Um zwischen direkter und indizierter Adressierung wählen zu können, bietet sich diese Technik an. Der Befehl hat die folgende Struktur:

Opcode	j	Adresse

j kann dabei zum Beispiel auf 3 Bits codiert sein und gibt folgende Situationen an:

$j = 0$	Befehl ohne Adreßmodifikation, Befehl: $\alpha := \rho(\text{Adresse})$
$j \in \{1, ... 7\}$	Adreßmodifikation mit Indexregister γ_j Befehl: $\alpha := \rho(\text{Adresse} + \gamma_j)$

Unterschiedliche Adreßmodi

Ferner sind Modifikationen der Adresse denkbar. So wird je nach Modus zum Beispiel das Indexregister automatisch inkrementiert, wenn über dieses eine Adressierung vorgenommen wird.

Beispiel: PDP 11

Sei j = Nummer des Indexregisters γ_j, hier: $j \in \{0, ... 7\}$.

I = \B\LC\{(\A\AL\CO2\HS9(0;direkte Adressierung;1;indirekte Adressierung (siehe nächster Abschnitt)))

Modus	Befehl	Wirkung (Beispiel)
00	Register (-befehl)	$\rho(x) := \gamma_j$
01	Autoinkrement	$\rho(x) := \rho(\gamma_j); \gamma_j := \gamma_j + 1$
10	Autodekrement	$\gamma_j := \gamma_j - 1; \rho(x) := \rho(\gamma_j)$
11	Indiziert	$\rho(x) := \rho(\gamma_j + \rho(F))$ F ist das Wort, welches auf den Befehl folgt. Bem.: Mischung von Befehlen und Daten!

Indirekte Adressierung

Eine weitere Variante ist die indirekte Adressierung, bei der die Adresse einer Speicherzelle als Inhalt einer anderen Speicherzelle ermittelt wird. Ein Holebefehl hat dann die Form:

$$\alpha := \rho(\rho(i)),$$

während er bei direkter Adressierung „$\alpha := \rho(i)$" lautete.
In einem Befehl ist die Kennzeichnung, ob „$\alpha := \rho(i)$" oder „$\alpha := \rho(\rho(i))$" gemeint ist, durch das I-Bit möglich; zum Beispiel zeigt I=0 die direkte und I=1 die indirekte Adressierung an.

Beispiel:

Bearbeitung der Zellen 100, 99, .., 1 mit indirekter Adressierung:

```
        α := 100;
START   ρ(200) := α              Hilfsspeicherzelle 200
        α := ρ(ρ(200))
        .
        .
        .
        α := ρ(200)
        α := α – 1
        if  α > 0  then goto START
```

Modifizierung von Befehlen

Der Inhalt des Adreßteils eines Befehls wird hier modifiziert, also etwa inkrementiert oder dekrementiert.
Die Wirkung besteht darin, daß das Programm während seines Durchlaufs verändert wird. Dies ist zum Teil undurchschaubar und daher gefährlich.

2.4 Leistungsfähigkeit von Adressiertechniken, (notwendiger) Umfang von Assemblersprachen

Satz:

Die Adreßtechniken, welche indirekte Adressierung erlauben, besitzen die volle Funktionalität der absoluten Adressierung, bzw. sind sogar noch stärker.

Beweisidee:

1.) Jede der Techniken kann durch jede andere simuliert werden.
2.) Problem: (siehe nachfolgende Vermutung)

Vermutung: Direkte Adressierung ist schwächer als indirekte Adressierung.

Ist die direkte (absolute) Adressierung „schwächer" als eine indirekte Adressierung? Unter „schwächer" ist dabei zu verstehen, daß es ein Problem gibt, zu dessen Lösung ein Programm mit indirekter Adressierung existiert, aber keines, welches allein mit direkter Adressierung auskommt.
Es stellt sich heraus, daß diese Behauptung **nicht allgemein** gilt, sondern von der Endlichkeit des Rechnermodells abhängt.

Berechnung durch ein Programm

Zur Diskussion der Vermutung müssen wir uns mit der Frage befassen, was ein Programm eigentlich berechnet.
Ein (terminierendes) **Programm** soll eine Fragestellung universell lösen, das heißt, ein und dasselbe Programm wird auf verschiedene Argumentkombinationen angesetzt (siehe Beispiel: Additionsprogramm). Es bewirkt „im wesentlichen" eine **Transformation** eines alten Rechenspeicherinhalts in einen neuen Rechenspeicherinhalt. „Im wesentlichen" soll heißen: Ein- und Ausgabe nur vom oder zum Speicher; Register, Flags und Statusinformationen sind nur für Hilfszwecke vorgesehen, also unerheblich für das Start- beziehungsweise Endergebnis.
Damit kann man ein Programm als **Abbildung** vom Rechenspeicher in den Rechenspeicher (RSP) auffassen:

$$f: RSP \rightarrow RSP.$$

Beispiel: Additionsprogramm

```
0:  γ := 100
1:  α := 0                  Initialisierung: Anfangssumme
2:  α := α + ρ(γ)           Indirekte Adressierung
3:  γ :=γ – 1
4:  if γ > 0 then goto 2
5:  ρ(1) := α
6:  HALT
```

In diesem kurzen Programm sei es erlaubt, Zahlen als Label zu verwenden, die an sich keine Aussage über die Programmstruktur zulassen. Daher sollte man bei längeren Assemblerprogrammen aussagekräftige Label verwenden.
Die Wirkung des Programms ist in Abbildung 2.7 gezeigt.

Fall A: Endlichkeit der realen Maschine

Wir betrachten zunächst die Berechenbarkeit, die auf der „Endlichkeit" einer realen Maschine basiert. Jede Rechenspeicherzelle ist endlich groß, daher ist auch jedes Programmspeicherwort endlich groß. Diese obere Schranke soll nicht vergrößert werden können (etwa durch längere Worte).
Die Information einer Rechenspeicherzelle sei o.B.d.A. eine ganze Zahl, die kleinste darstellbare Zahl sei mit **MINZAHL** und die größte darstellbare Zahl mit **MAXZAHL** bezeichnet.

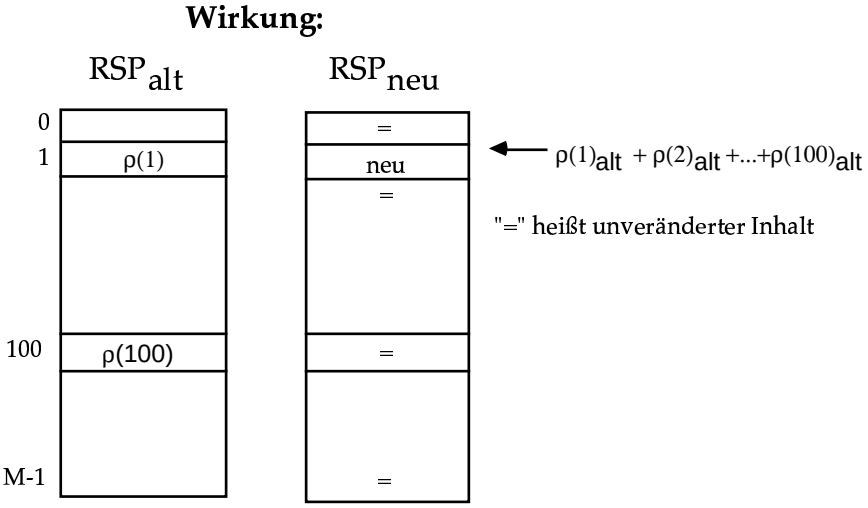


Abbildung 2.7: Programm als Funktion f: RSP → RSP.

Ein Programm, das M Speicherzellen verwendet, ist auffaßbar als Abbildung

$$f: D \rightarrow D', \text{ wobei } D, D' \subseteq [\text{MINZAHL} : \text{MAXZAHL}]^M$$

Beispiel:

Das vorige Additionsprogramm hatte 100 Eingangsargumente und 1 Ausgangswert, also gilt hier

$D = [\text{MINZAHL} : \text{MAXZAHL}]^{100}$ und

$D' = [\text{MINZAHL} : \text{MAXZAHL}]^1$.

Satz:

Jede Abbildung $f: D \rightarrow D'$ mit $D, D' \subseteq [\text{MINZAHL} : \text{MAXZAHL}]^M$ ist realisierbar durch ein Programm, welches (außer Steuerbefehlen wie START, STOP, ...) auskommt mit:

$\alpha := \rho(i)$,	für $i, j \in \{0, ..., M-1\}$
$\rho(j) := \alpha$,	i und j sind absolute Adressen, keine Variablen
$\alpha := k$,	k ist Konstante
$\alpha := \alpha \pm 1$,	In-/Dekrementierung des Akkumulatorinhalts

if $\alpha < 0$ then goto j

if $\alpha = 0$ then goto j

if $\alpha > 0$ then goto j

Bemerkung:

Dieses Programm kommt insbesondere ohne indirekte Adressierung und mit einem sehr kleinen Befehlssatz aus.

Beweis (für jedes Programm, also für jede Abbildung f) :

Die Abbildung f bewirkt: $f(\rho(0)_{alt}, ..., \rho(M-1)_{alt}) = (\rho(0)_{neu}, ..., \rho(M-1)_{neu})$, wobei: $\rho(k)_{neu} = h_k(\rho(0)_{alt}, ..., \rho(M-1)_{alt})$. Das heißt, der neue Wert ergibt sich aus den alten Werten (im Extremfall aus allen alten Werten).

Beispiel (voriges Additionsprogramm):

$k \neq 1$: $\rho(k)_{neu} = h_k(...) = \rho(k)_{alt}$

$k = 1$: $\rho(1)_{neu} = h_1(\rho(1)_{alt}, ... \rho(100)_{alt}) = \rho(1)_{alt} + ... + \rho(100)_{alt}$
also:

$$h_1(\rho(1)_{alt}, ... \rho(100)_{alt}) = \begin{cases} 5050; \text{falls } \rho(i)_{alt} = i & \text{für alle } 1 \le i \le 100; \\ 0; \text{falls für alle } 1 \le i \le 100; ...; \text{andere Fälle} \end{cases}$$

Die Programmwirkung ist also beschreibbar durch Abbildungen h_0, h_1, ... h_{M-1}. Abbildung 2.8 zeigt, wie die Eingangsargumente $\rho(i)_{alt}$ schichtweise im Baum abgefragt werden. Somit ist jede Kombination von Eingabeargumenten durch einen Pfad im Baum gegeben. An den Blättern erhält man aufgrund der Funktionen h_1 bis $h_{(M-1)}$ und der durch den Pfad gegebenen Eingangswerte die Ausgabeargumente.

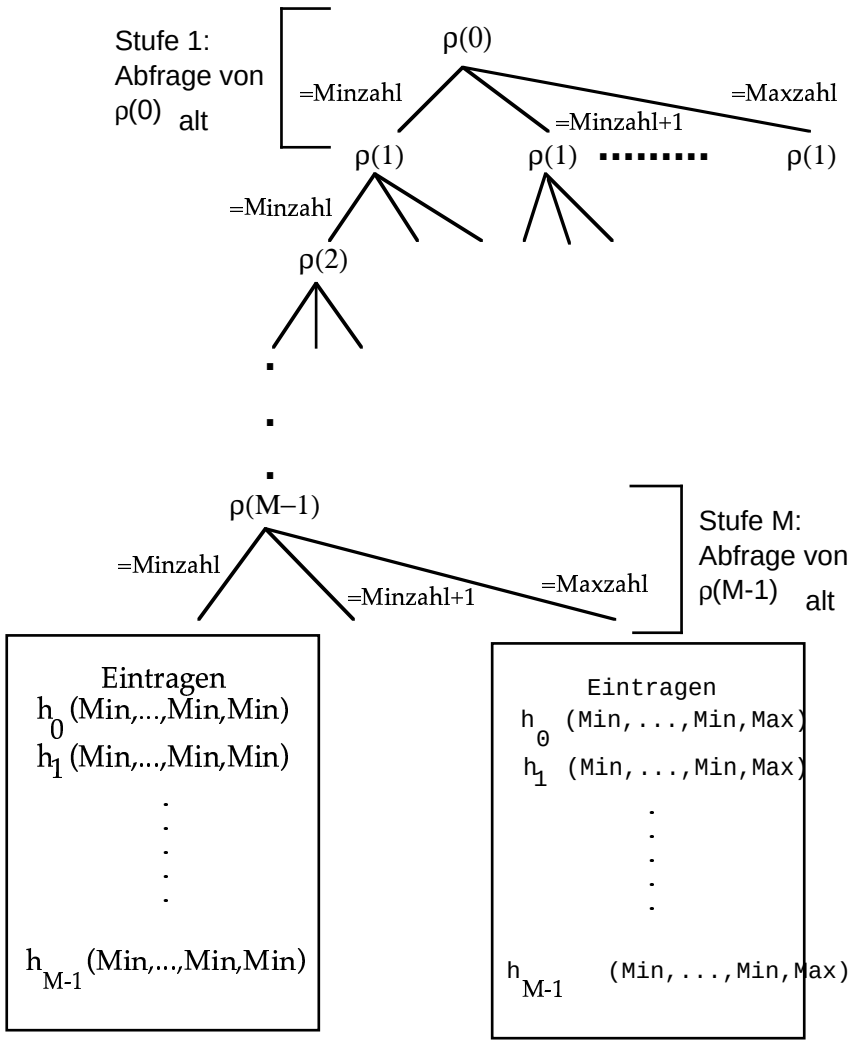


Abbildung 2.8: Algorithmus zur Berechnung von h0, h1, ...

Das folgende Programm realisiert diese schichtenweise Abfrage der Eingangsargumente im wesentlichen unter Verwendung bedingter Sprünge. Die Ausgabeargumente werden im letzten Abschnitt des Programms eingetragen.
Sei zur Abkürzung: MINZAHL = – Q und MAXZAHL = + Q.

```
0:        α := ρ(0)
1:        if α = 0 then goto [Stufe 2, Wert 0]      Label mit Nummer 4Q + 1
2:        if α < 0 then goto (2Q + 2)
3:        α := α – 1
4:        if α = 0 then goto [Stufe 2, Wert 1]
5:        α := α – 1
6:        if α = 0 then goto [Stufe 2, Wert 2]
.         .
.         .
.         .
2Q:       if α = 0 then goto [Stufe 2, Wert Q – 1]
2Q + 1:   goto [Stufe 2, Wert Q]
2Q + 2:   α:= α + 1
2Q + 3:   if α = 0 then goto [Stufe 2, Wert – 1]
.         .
.         .
.         .
4Q – 1:   if α = 0 then goto [Stufe 2, Wert – Q + 1]
4Q:       goto [Stufe 2, Wert – Q]
4Q + 1    [Baumorganisation Stufe 1, 2, ..., M]
```

Letzte Stufe zur Eintragung der Ergebnisse:

```
α := h0 (–Q,... –Q)                 zum Beispiel: α := +17
ρ(0) := α
α := h1 (–Q,... –Q)                 zum Beispiel: α := –3
ρ(1) := α
.
.
.
α := h(M–1)(–Q,... –Q)
ρ(M–1) := α

α := h0(–Q, –Q, –Q, –Q + 1)
ρ(0) := α
.
.
.
```

Fall B: „Beliebig große" Funktionswerte sollen berechenbar sein

Ein Beispiel ist die Additionbeliebig großer Zahlen.
Das Programm ist jetzt eine Abbildung:

$$f: [\text{MINZAHL:MAXZAHL}]^* \rightarrow [\text{MINZAHL:MAXZAHL}]^*.$$

Das „*" zeigt an, daß es beliebig viele Rechenspeicherzellen gibt.

Satz:
Es gibt Abbildungen, die nur durch Programme **mit** indirekter Adressierung berechenbar sind.

Bemerkung:
Diese Programme enthalten also Befehle der Art $\rho(\rho(i)):=\alpha$ oder $\rho(\gamma):=\alpha$.

Beweis:
Ein Programm ohne indirekte Adressierung spricht nur feste Rechenspeicherzellen durch Befehle der Form $\rho(i):=\alpha$ an. Jedes Programm P hat eine maximale, von ihm angesprochene Rechenspeicheradresse ADMAX(P). „Hinter" ADMAX(P) kann im Rechenspeicher nichts geändert werden. Daher sind alle Probleme, die hinter ADMAX(P) etwas ändern, von P nicht bearbeitbar.

Beispiel für Existenz solcher Probleme:
„Suche die erste 0 im Rechenspeicher und ersetze sie durch 1"
Hier kann nämlich die erste 0 hinter ADMAX(P) liegen.

Annahme:
Es existiere ein Programm P ohne indirekte Adressierung.
Man betrachte die Rechenspeicherbelegung, bei der keine 0 in den Speicherzellen mit einer Adresse kleiner/gleich ADMAX(P) steht.

Für diese Speicherbelegung versagt das „Suchprogramm" P.
Widerspruch.

Mit indirekter Adressierung geht es sehr einfach:

```
0:  γ := −1
1:  γ := γ+1
2:  α := ρ(γ)
3:  if (α ≠ 0) then goto 1
4:  ρ(γ) := 1
5:  HALT
```

2.5 Unterprogramme

Beim Programmaufbau aus Modulen werden oft benutzte Teile als **Unterprogramm (UP)** zusammengefaßt. Diese können auch in Form einer Unterprogrammbibliothek abgerufen werden, was der Wiederverwendbarkeit dient. In diesem Abschnitt werden nun Probleme aufgezeigt, die sich bei der Handhabung solcher Unterprogramme auf maschinennaher Ebene ergeben.

Ein Unterprogramm habe die folgende Form:

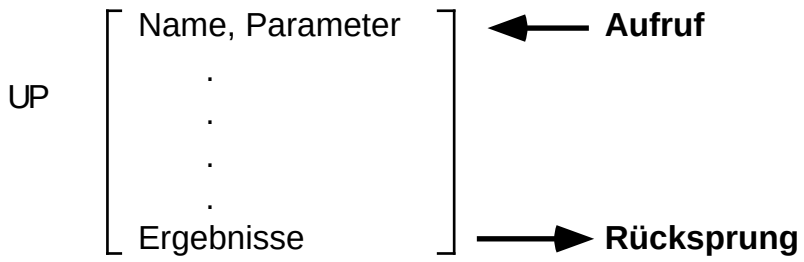


Abbildung 2.9: Aufbau eines Unterprogramms

Unterprogramme sind möglicherweise mehrstufig **verschachtelt**, das heißt, ein Unterprogramm ruft ein anderes oder sich selbst **(Rekursion)** auf. Einen Ablauf einer solchen Aufrufkette zeigt die folgende Abbildung:

Abbildung 2.10: Aufruf und Beenden von Unterprogrammen

Die Pfeile „→" kennzeichnen die Unterbrechung des (Unter-)Programms der Ebene i und den Aufruf des Unterprogramms der Stufe (i+1). Die Pfeile in umgekehrter Richtung zeigen die Beendigung des Unterprogramms der Stufe (i+1) und den Wiedereintritt in das aufrufende (Unter-)Programm der Stufe i an.
Ein Problem ist es nun, die richtige **Einsprungstelle** für den Wiedereintritt zu verwalten. Die unterschiedlichen Techniken bedingen folgende Kategorien von Unterprogrammen.

Einstufige, nichtrekursive Unterprogramme

Die Speicherposition des aufrufenden Programms wird im Register δ gespeichert. Ein Unterprogrammaufruf durch „call" legt die Adressse des „call"-Befehls, also des aufrufenden Programms, in δ ab. Ein „return" aus dem Unterprogramm führt zu einem Sprung an die nächste Adresse nach der „geretteten" Programmadresse in δ.
Weil nur eine Zelle (Register) zur Verwaltung der Adresse des aufrufenden Programms verwendet wird, kann auch nur eine Verzweigung in ein Unterprogramm durchgeführt werden. Daher nennt man diese Programme „einstufig". Insbesondere ist Rekursion unmöglich.

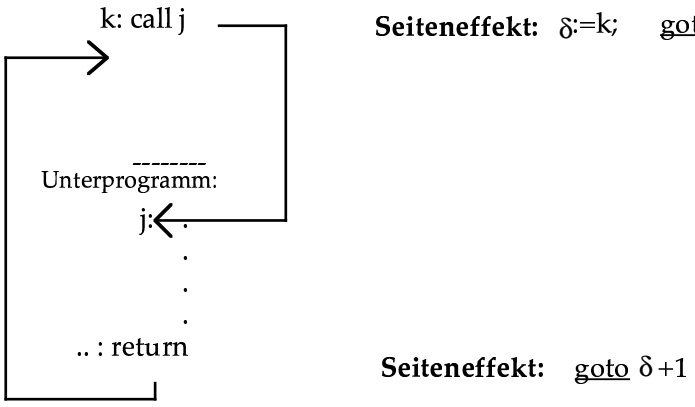


Abbildung 2.11: Einstufiger Unterprogrammaufruf

Mehrstufige, nichtrekursive Unterprogramme

Bei dieser Technik hat jedes Unterprogramm eine eigene Rücksprungadresse. Sie ist an einer speziellen Stelle dessen gespeichert und gibt an, an welcher Stelle in dem aufrufenden (Unter-) Programm fortzufahren ist, wenn das aufgerufene Unterprogramm selbst beendet ist. In Abbildung 2.12 ist diese Rücksprungadresse an der ersten Stelle des aufgerufenen Unterprogramms abgelegt (ρ(j)).

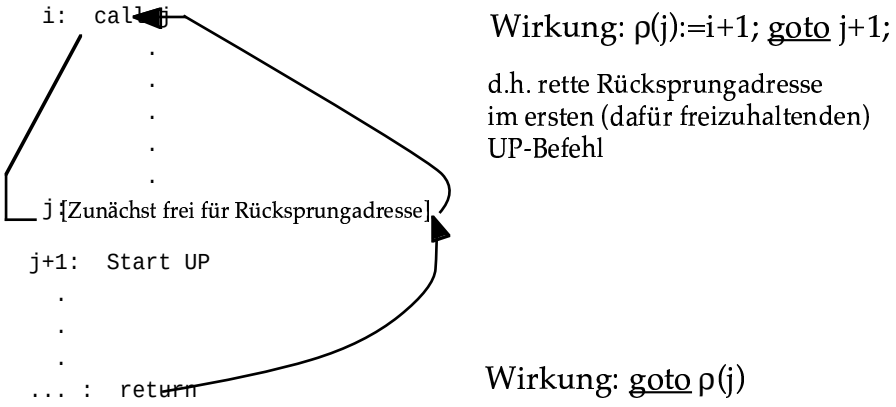


Abbildung 2.12: Mehrstufiger Unterprogrammaufruf

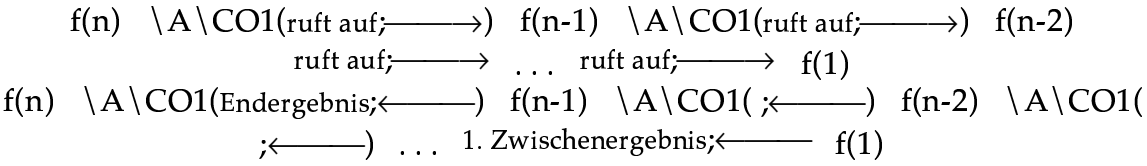
Eine Voraussetzung für die Funktionsfähigkeit dieser Technik besteht darin, daß das Unterprogramm sich nicht selbst aufruft (**reentrent** ist); in diesem Falle würde die Rücksprungadresse überschrieben werden.

Mehrstufige, eventuell rekursive Unterprogramme

Zunächst wird ein Beispiel eines typischen rekursiven Unterprogramms gegeben: Es handelt sich um die rekursive Berechnung der Fakultätsfunktion (n!).

Algorithmus: f(n):= if n=0 or n=1 then 1 else n·f(n-1)

Ein Aufruf von f(n) zieht die folgende Kette von rekursiven Unterprogrammaufrufen nach sich:



Die Technik des letzten Abschnitts ist nicht anwendbar, da nur eine Speicherzelle für die Rücksprungadresse vorgesehen ist. An dieser Stelle müßten also „beliebig viele" Stellen für Rücksprungadressen zur Verfügung stehen, anhand derer man ablesen kann, welche die nächste Rücksprungadresse ist. Hierfür eignet sich der Stack, wobei die letzte Rücksprungadresse das Topelement ist.
Rekursive Aufrufe werden also über einen Stack mit den Operationen Push und Pop verwaltet, wobei folgende Register verwendet werden:

- δ_1: Register, dessen Inhalt auf die oberste Stackposition (Topelement) zeigt (für Push und Pop).
- δ_2: Befehlszählregister (enthält Adresse des „aktuellen" Befehls).
- R_L: Das sogenannte Linkageregister ist ein beliebiges Register zum Auslesen der Parameter im Unterprogramm und Speichern der Rücksprungadresse.

Der „call"- und „return"-Befehl haben dann die Wirkung:

- call R_L, dst [Push R_L; R_L := δ_2+1; goto dst]
- return R_L [Pop R_L; δ_2 := R_L]

Dabei ist „dst" die erste Adresse des aufgerufenen Unterprogramms:

\B\RC\}(\A\AL\CO1(dst: .; <Unterprogramm-Anweisungen> ; .)) R_L, Autoinkrement

Der „call"-Befehl sichert den aktuellen Wert des Linkageregisters auf dem Stack und läßt R_L auf den ersten Parameter verweisen.
Das Unterprogramm liest die Parameter mittels Autoinkrementadressierung über R_L, so daß R_L anschließend auf den nächsten Befehl zeigt. Dieser ist die Rücksprungadresse. Der „return"-Befehl setzt das Befehlszählregister auf die Rücksprungadresse und holt den alten Wert von R_L vom Stack.

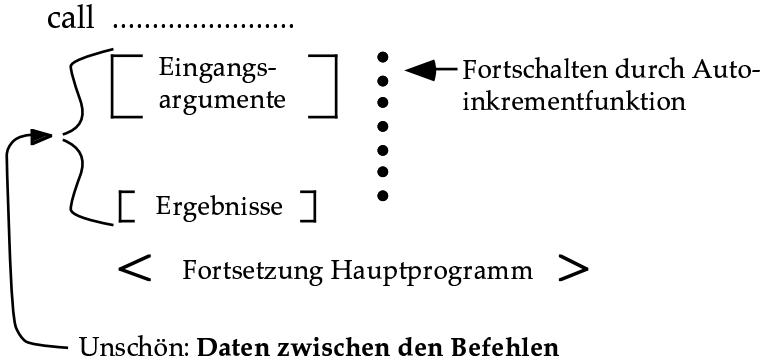


Abbildung 2.13: Organisation mit Linkageregister

Eine andere Organisationsform für die Verwaltung der Eingabe- und Ausgabeparameter verwendet zwei besondere Register: R_{EING} und R_{RESULT} sollen die Rechenspeicheradressen enthalten, in denen die Eingangsparameter beginnen beziehungsweise ab wo die Ergebnisse zurückzuschreiben sind. Diese müssen (bei Aufruf des Unterprogramms) geeignet vorbesetzt werden. Die Eingangsparameter werden sofort übernommen (das heißt, es ist hierfür kein Stack notwendig). R_{RESULT} wird über einen Stack verwaltet.
Abbildung 2.14 zeigt schematisch den zugehörigen Unterprogrammaufruf.

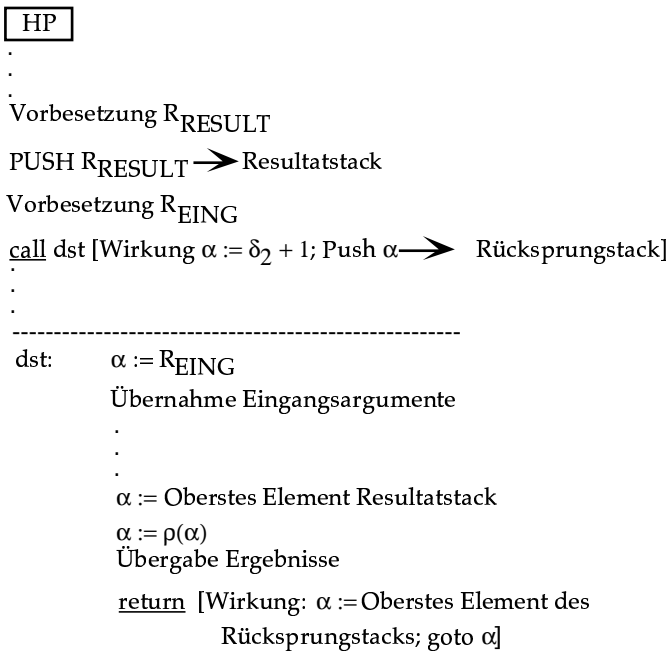


Abbildung 2.14: Organisation mit R_{EING} und R_{RESULT}

R_{EING} und R_{RESULT} verweisen auf den Beginn der Eingangs- beziehungsweise Ergebnisparameterblöcke.
Damit man weiß, wieviele Argumente zu übergeben sind, kann man in die erste Zelle dieser Bereiche jeweils die Länge (Anzahl der zu übergebenden Parameter plus Länge selbst) speichern.

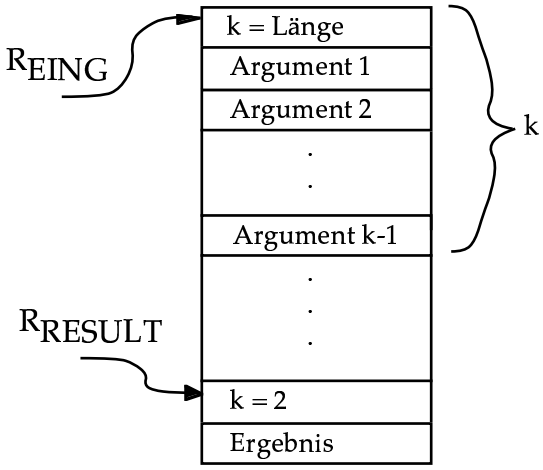


Abbildung 2.15: Verwaltung der Eingangs- und Ergebnisparameter

2.6 Prozessorzustand, Programmstatuswort

Der **Prozessorzustand** bezeichnet den aktuellen Inhalt der

- Register (Akkumulator, Indexregister, andere Allzweckregister),
- Programmstatusregister und
- Flags (spezielle 1-Bit-„Register", zum Beispiel zur Markierung von „Eingeschaltet / Nicht eingeschaltet", „Fehler / kein Fehler")

Das **Programmstatuswort** hält Informationen über den Prozessorzustand (Modus), Prozeßpriorität (Interruptverwaltung) oder über arithmetische Zustände. Zum Beispiel sei das Programmstatuswort der älteren PDP-11 erwähnt:

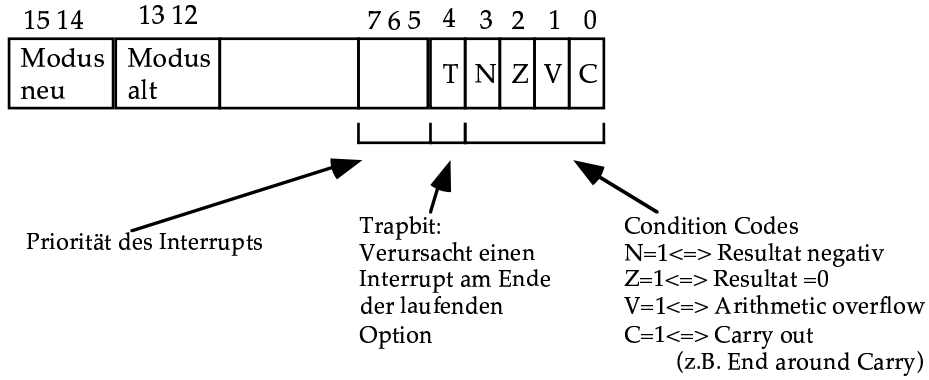


Abbildung 2.16: Programmstatuswort PDP-11

Bemerkungen:

- Der Prozessormodus gibt Auskunft über die Priorität eines Prozesses, zum Beispiel: „user" hat niedrige Priorität, „kernel" hat hohe Priorität.

- Im *User Mode* ist der Zugang nur zu bestimmten Befehlen, bestimmten Speicherbereichen und bestimmten Endgeräten erlaubt.
- „Modus neu" ist jetziger Prozessormodus.
- „Modus alt" ist voriger Prozessormodus.

Diese Thematik wird in der Vorlesung über „Systemprogrammierung" vertieft.

3. Bausteine und Komponenten von Rechensystemen

Im Kapitel 2 ist die Darstellung und Adressierung von Informationenbehandelt worden. In diesem Kapitel wird die rechnerinterne Verarbeitung von Informationen mittels Schaltungen, Schaltkreisen und Schaltwerken eingeführt.
Im ersten Abschnitt wird der Begriff der Schaltfunktion definiert. Die Verknüpfung solcher Schaltfunktionen führt zu den Bausteinsystemen. Eingettet werden die Begriffe in die Boolesche Algebra. Der zweite Abschnitt führt in die Normalformen von Booleschen Ausdrücken ein. Abschnitt drei behandelt die Synthese von Schaltkreisen und deren Minimierung. Im letzten Abschnitt dieses Kapitels werden Schaltwerke und Speicherelemente besprochen.

3.1 Schaltfunktionen, Bausteinsysteme und Boolesche Algebra

Einführung

Die Aufgabe einer Schaltung besteht in der **Transformation** (binär dargestellter) **Eingangssignale** in (binär dargestellte) **Ausgangssignale**.
Eine Schaltung ist daher mit einem Programm vergleichbar, mit dem Unterschied, daß die Transformation durch die Schaltung fest verdrahtet ist. Eine Schaltfunktion beschreibt die Zuordnung von Ausgangswerten zu Eingangswerten.

Beispiel: LED-Dezimalzähler

Als erstes Beispiel für eine Schaltung soll ein LED-Zähler betrachtet werden. Der LED-Dezimalzähler besteht aus 7 „Strichen", die leuchten oder nicht leuchten; siehe Abbildung 3.1.

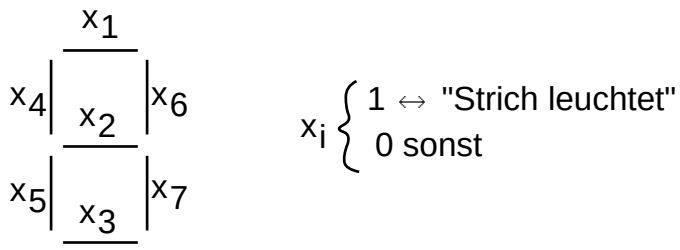


Abbildung 3.1: LED-Zähler

Dabei läßt sich die „7" durch Aufleuchten der „Striche" x_1, x_6 und x_7 „erzeugen". Dies entspricht dem Setzen von Bits, so daß: $(x_1, x_2, \ldots x_7) = (1,0,0,0,0,1,1)$. Dem entsprechend ist der „4" (0,1,0,1,0,1,1) zugeordnet. Ein LED-Zähler soll nun von 0 bis 9 modulo 10 zählen. Er muß also zur aktuellen Zahl i (Eingangswert) den Nachfolger ((i+1) modulo 10) (Ausgangswert) erzeugen, wie Abbildung 3.2 veranschaulicht.

$i_{DEZIMAL} \rightarrow (i+1 \text{ modulo } 10)_{DEZIMAL}$

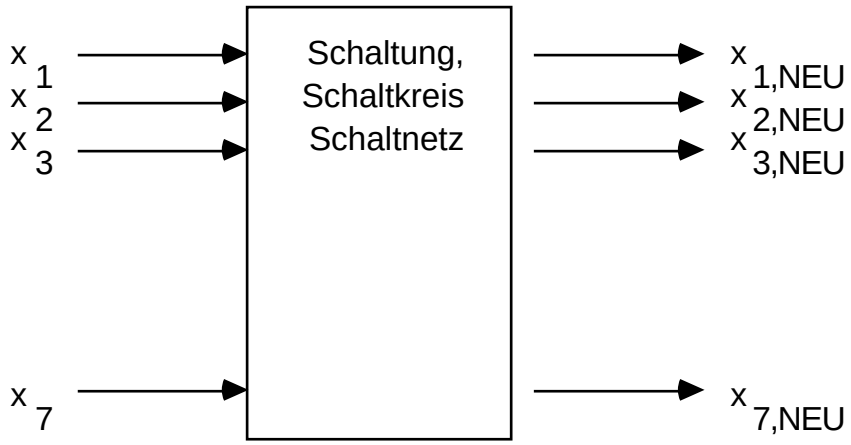


Abbildung 3.2: LED-Zähler als Transformation

Allgemeine **Definition einer Schaltung**:

Die Wirkung einer **Schaltung** wird durch eine n-stellige **Schaltfunktion** mit k Ausgängen beschrieben.

Bemerkung:
Die Schaltfunktion f hat n Eingänge und k Ausgänge, wobei jeder Ein- und Ausgang den Wert „0" oder „1" annimmt.

$$f: B^n \longrightarrow B^k, \text{ mit } B = \{0, 1\}$$

n Eingänge k Ausgänge

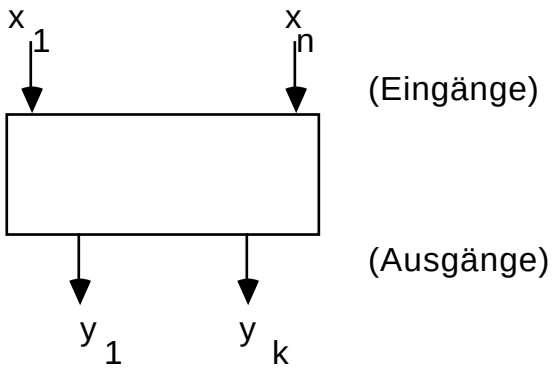


Abbildung 3.3: Symbol für eine n-stellige Schaltfunktion mit k Ausgängen

Möglicherweise kann nur ein Teil aller denkbaren Eingangssignale (Kombinationen der x_i in Form von $(x_1, x_2, \ldots, x_n)$) vorkommen.

In diesem Fall gilt: $f: B^n \supset D \rightarrow B^k$.

Auf dem Eingangsbereich $B^n \setminus D$ ist die Schaltfunktion undefiniert und somit in ihren Ausgangswerten frei wählbar. Diesen Bereich bezeichnet man auch als **„Don't Care"-Bereich**.

Beispiel:

Bei der LED-Anzeige werden nur 10 Kombinationen der 2^7 möglichen benutzt. Es ist zum Beispiel dem Leuchten der Striche drei, vier und sechs keine Zahl sinnvoll zuzuordnen.

Eine n-stellige Schaltfunktion kann in Form einer Tabelle angegeben werden, indem man in die linke Spalte die definierten Eingangswerte und in die rechte Spalte die die Funktion definierenden Ausgangswerte schreibt, vgl. Abbildung 3.4.

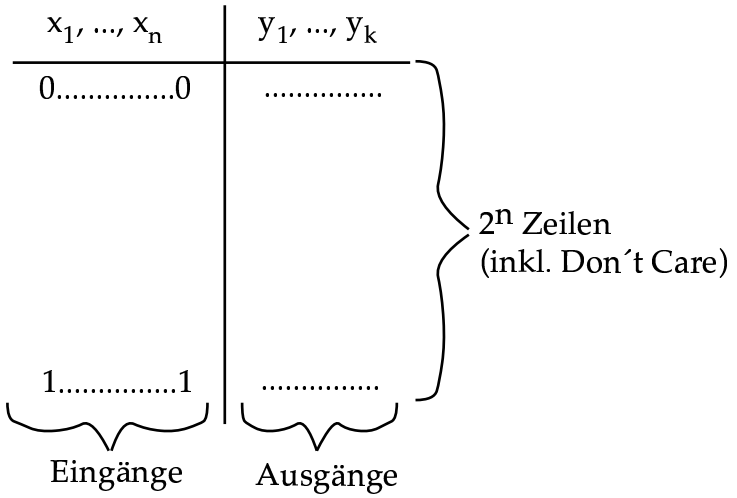


Abbildung 3.4: Durch Tabelle definierte Schaltfunktion

Im Falle, daß die Funktion n-stellig und jede Komponente binär ist, sind maximal 2^n Werte des Definitionsbereichs und dem entsprechend viele Ausgangswerte denkbar.

Beispiel: Rückwärtszähler im 3-Excess-Code

Die folgende Tabelle zeigt die Schaltfunktion für einen Rückwärtszähler im 3-Excess-Code, der im Abschnitt „Darstellung von Zeichen" im Kapitel 2 eingeführt wurde.

Die Aufgabe besteht in der Beschreibung der Funktion:

$$[i]_{3\text{-Excess}} \rightarrow [(i\text{-}1) \text{ modulo } 10]_{3\text{-Excess}}$$

Dezimaler Wert	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	Dezimaler Wert
0	0	0	1	1	1	1	0	0	9
1	0	1	0	0	0	0	1	1	0
2	0	1	0	1	0	1	0	0	1
3	0	1	1	0	0	1	0	1	2
4	0	1	1	1	0	1	1	0	3
5	1	0	0	0	0	1	1	1	4
6	1	0	0	1	1	0	0	0	5
7	1	0	1	0	1	0	0	1	6
8	1	0	1	1	1	0	1	0	7
9	1	1	0	0	1	0	1	1	8
	0	0	0	0					
	0	0	0	1					
	0	0	1	0	Don´t Care				
	1	1	0	1	Werte				
	1	1	1	0					
	1	1	1	1					

Abbildung 3.5: Als Tabelle angegebene Schaltfunktion

Eine Schaltfunktion mit k Ausgängen ist darstellbar durch k Schaltfunktionen auf dem gleichen Definitionsbereich mit je einem Ausgang:

$$f: B^n \mapsto B^k \quad \text{entspricht} \quad \{ f_1: B^n \mapsto B;:;f_k: B^n \mapsto B$$

Aus diesem Grund kann man sich im folgenden auf Schaltfunktionen mit einem Ausgang beschränken.

Entwurf und Realisierung von Schaltfunktionen:

In diesem Abschnitt werden „Bausteine" eingeführt, die es gestatten, binäre Eingangswerte in binäre Ausgangswerte zu überführen. Man kann sie als Operationen auf B auffassen.
Es bestehen nun die Fragen:

- Welche Bausteine sind notwendig, um bestimmte Eigenschaften der Schaltfunktion wiederzugeben?
- Wie kann man eine Schaltfunktion mit möglichst wenigen Bausteinen darstellen, um den Realisierungsaufwand (und damit die Kosten) gering zu halten?

Dabei soll der Entwurf möglichst systematisch vor sich gehen. Besonders einfache Schaltfunktionen lassen sich als Grundbausteine („Bausteinfunktionen") zur Zusammensetzung komplexerer Funktionen verwenden.
Die folgende Tabelle zeigt einige Symbole für solche Grundbausteine:

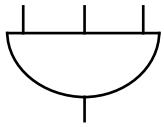
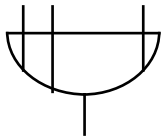
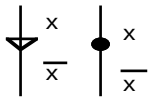
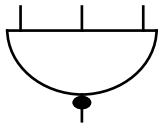
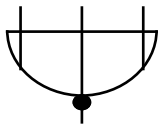
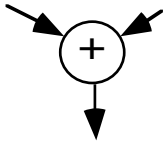
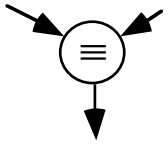
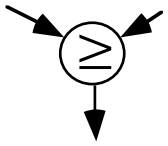
Name	Definition	Symbol
Konjunktion (AND)	$AND_n: B^n \rightarrow B$ $AND_n(x_1,...,x_n)=1$ $\Leftrightarrow x_1*...*x_n=1$	
Disjunktion (OR)	$OR_n: B^n \rightarrow B$ $OR_n(x_1,...,x_n)=0$ $\Leftrightarrow x_1+...+x_n=0$	
Negation (NOT)	$NOT : B \rightarrow B$ $NOT(x)=1-x=\bar{x}$	
NAND	$NAND_n: B^n \rightarrow B$ $NAND_n(x_1,...,x_n)=1$ $\Leftrightarrow x_1*...*x_n=0$	
NOR	$NOR_n: B^n \rightarrow B$ $NOR_n(x_1,...,x_n)=1$ $\Leftrightarrow x_1+...+x_n=0$	
EXOR	$EXOR_n: B^n \rightarrow B$ $EXOR_n(x_1,...,x_n)=$ $(x_1+...+x_n) \bmod 2$	
EQUIV	$EQUIV_n(X_1,...,x_n)=\overline{EXOR_n}$	
Schwellen-elemente	$\sigma_k: B^n \rightarrow B$ $\sigma_k (x_1,...,x_n)=1 \Leftrightarrow$ $x_1+...+x_n \geq k$	

Abbildung 3.6: Grundbausteinfunktionen und ihre Symbole

Die EXOR-Funktion (auch XOR) kann als Parityfunktion verwendet werden, wie zum Beispiel im Abschnitt über „Darstellung von Zeichen" in Kapitel 2 beschrieben. Die EXOR-, NOT- und EQUIV-Funktion sind keine Schwellenelemente. Für das logische „Und" schreibt man $\wedge$, $\cap$ oder $\cdot$. Das „Oder" wird als $\vee$, $\cup$ oder + geschrieben. „Und" bindet stärker als „Oder".

Definition:
Eine Menge von Bausteinfunktionen, aus denen sich beliebige Schaltfunktionen zusammensetzen lassen, heißt **Bausteinsystem.**

Beispiele:

In Abbildung 3.7 sind einige Bausteinsysteme aufgeführt.

Bausteinsystem	Nachweis
$\{AND_2, OR_2, NOT\}$	Disjunktive und konjunktive Normalform
$\{AND_2, NOT\}$	$OR_2(x,y) = NOT(\ AND_2(NOT(x), NOT(y))\ )$
$\{OR_2, NOT\}$	$AND_2(x,y) = NOT(\ OR_2(NOT(x), NOT(y))\ )$
$\{NAND_2\}$	$NOT(x) = NAND_2(x, x)$ $AND_2 = NOT(NAND_2)$
$\{NOR_2\}$	Analog zu $\{NAND_2\}$
$\{AND_2, EXOR_2, 1\}$	Komplementfreie Ringsummenentwicklung
$\{\sigma_k, NOT\}$	$\sigma_1 = OR_n$, $\sigma_n = AND_n$

Abbildung 3.7: Bausteinsysteme

Schaltfunktionen lassen sich mit diesen Grundbausteinen zu beliebigen neuen Schaltfunktionen zusammensetzen; siehe Abbildung 3.8.

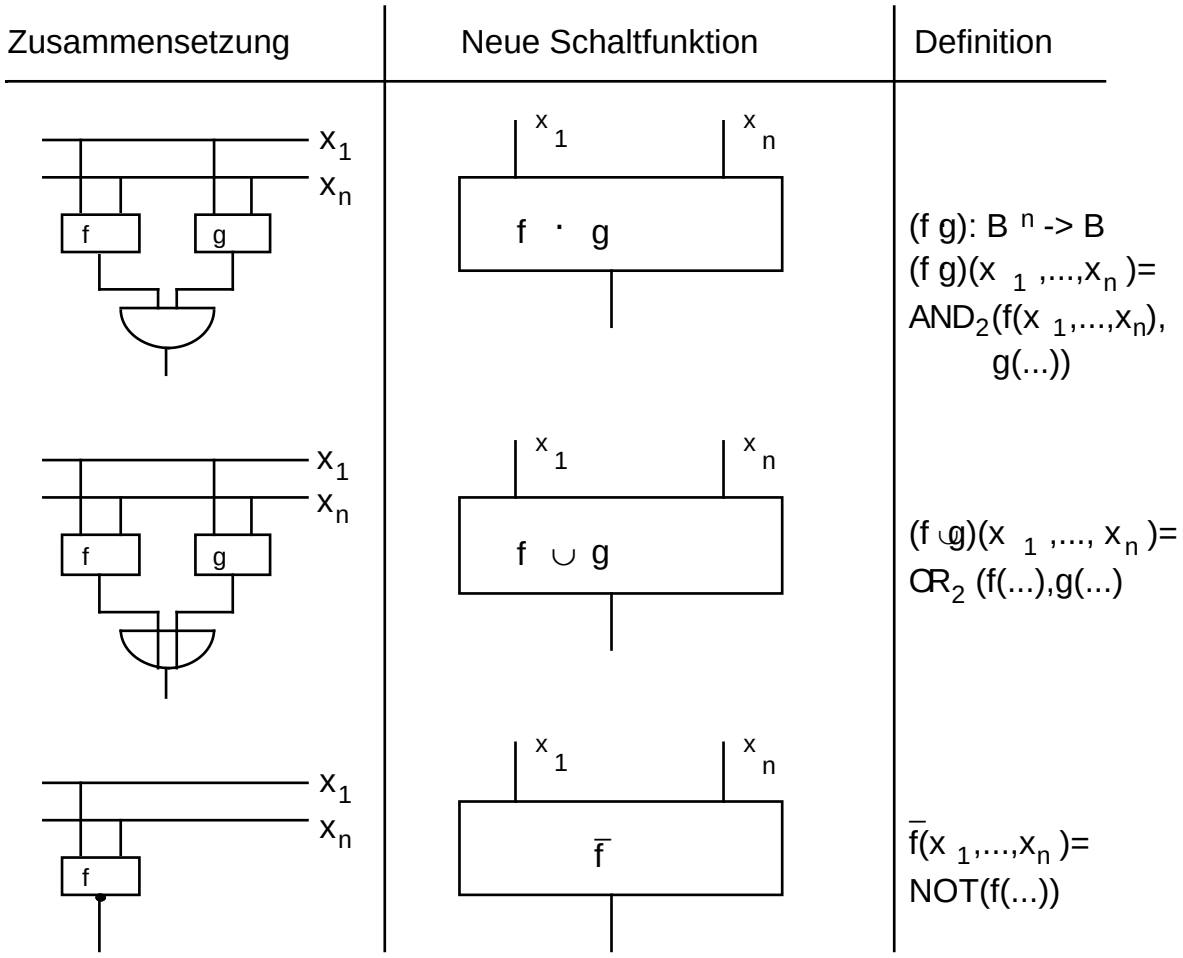


Abbildung 3.8: Zusammensetzung von Schaltfunktionen

Definition:
Zwei Schaltfunktionen f: $D \mapsto B$ und g: $D \mapsto B$ heißen **gleich** genau dann, wenn sie für alle zulässigen Argumentkombinationen das gleiche Ergebnis liefern:

$$f = g \Leftrightarrow f(x_1,...,x_n) = g(x_1,...,x_n), \text{ für alle } (x_1,...,x_n) \in D$$

Eigenschaften des Zusammensetzungsprozesses von Schaltfunktionen:

$f \wedge g = g \wedge f$	**(F1) Kommutativität**	$f \vee g = g \vee f$
$(f \wedge g) \wedge h = f \wedge (g \wedge h)$	**(F2) Assoziativität**	$(f \vee g) \vee h = f \vee (g \vee h)$
$f \wedge (g \vee h) = (f \wedge g) \vee (f \wedge h)$	**(F3) Distributivität**	$f \vee (g \wedge h) = (f \vee g) \wedge (f \vee h)$
$f \wedge (f \vee g) = f$	**(F4) Absorbtion**	$f \vee (f \wedge h) = f$
$f \wedge (g \vee \overline{\ };g) = f$	**(F5) Komplement**	$f \vee (g \wedge \overline{\ };g) = f$

Die Schaltfunktionen f, g und h erfüllen die Axiome einer Booleschen Algebra.

Definition:
Eine Menge $(A, \cdot, +, \overline{\ })$ mit drei Operationen: $\cdot, + : A \times A \mapsto A$ und $\overline{\ }: A \mapsto A$ heißt **Boolesche Algebra** $\Leftrightarrow$ Axiome (F1) bis (F5) sind erfüllt.

Folgerung:
Die Menge der Schaltfunktionen bildet mit den Operationen AND_2, OR_2 und NOT eine Boolesche Algebra.

Bemerkung:
Es sind auch andere Axiomensysteme denkbar, wie zum Beispiel eines, welches die Kommutativität (K), die Distributivität (D), das neutrale Element (N) und die Komplementfunktion (C - engl. *Complement*) umfaßt. Dabei gibt es Entsprechungen zwischen (K) und (F1) sowie (D) und (F3). Die anderen Axiome lauten:

(N) Es gibt neutrale Elemente $0,1 \in A$ mit $f + 0 = f$ und $f \cdot 1 = f$,
(C) Die Komplementfunktion $\overline{\ };f$ zu f erfüllt: $f + \overline{\ };f = 1$ und $f \cdot \overline{\ };f = 0$ für alle $f \in A$.

Satz:
Die Axiome (F1) bis (F5) sind äquivalent zu den Axiomen (K), (D), (N) und (C).

Bemerkung:
Andere Bausteinsysteme verwenden die Schaltfunktionen $\underline{0}$ und $\underline{1}$:

$\underline{0}: B^n \mapsto B$, mit $\underline{0}(x_1,...,x_n) = 0$
$\underline{1}: B^n \mapsto B$, mit $\underline{1}(x_1,...,x_n) = 1$
für alle $(x_1,...,x_n) \in B^n$.

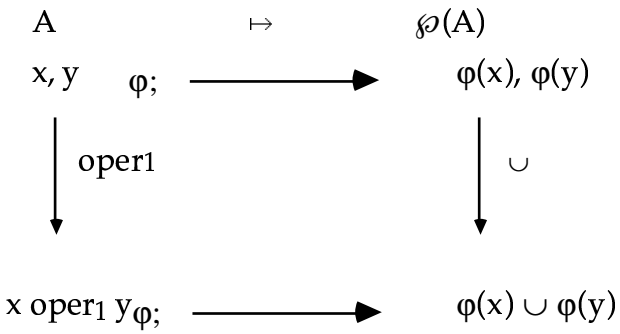
Andere Beispiele für Boolesche Algebren sind:

- Aussagenkalkül, wobei eine Aussage den Wahrheitswert „wahr" oder „falsch" annimmt. Dabei lassen sich Aussagen wie Schaltfunktionen zusammensetzen.
- Mengenalgebra mit den Operationen „Durchschnitt", „Vereinigung" und „Komplementmenge".

Satz von Stone:
Jede Boolesche Algebra $(A, oper_1, oper_2, oper_3)$ mit $|A| < \infty$ ist isomorph zu einer Mengenalgebra $(\wp(A), \cup, \cap, \bar{\ })$.
Das heißt, es gibt eine Bijektion $\varphi : A \mapsto \wp(A)$ mit:

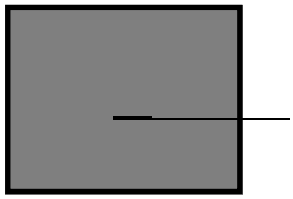
$$\begin{aligned} \varphi(x\ oper_1\ y) &= \varphi(x) \cup \varphi(y) \\ \varphi(x\ oper_2\ y) &= \varphi(x) \cap \varphi(y) \\ \varphi(oper_3\ x) &= \overline{\quad};\varphi(x) \end{aligned}$$

Bemerkung:
$\wp(A)$ ist die Potenzmenge von A.
Folgende Skizze veranschaulicht die Aussage des Satzes, wobei hier beispielhaft $oper_1$ und $\cup$ aufgeführt werden.



Das Diagramm ist kommutativ: man kann in der Mengenalgebra ebenso rechnen wie in der Algebra A.
Daher kann man alle Eigenschaften endlicher Boolescher Mengen als Eigenschaften von Mengenalgebren interpretieren. Dies kann dem leichteren Verständnis von Aussagen dienen.

Beispiel:
Die Aussage $(F3)_{DUAL}$ $f \vee (g \wedge h) = (f \vee g) \wedge (f \vee h)$ wird in Abbildung 3.9 deutlich.



1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1

Die Funktionen f_0 bis f_{15} tragen ihrer Bedeutung nach folgende Namen:

Funktion	Bezeichnung
f_0	NULL
f_1	AND
f_2	$\overline{x \to y}$
f_3	Projektion auf x
f_4	$\overline{y \to x}$
f_5	Projektion auf y
f_6	EXOR oder XOR
f_7	OR
f_8	NOR
f_9	EQUIV
f_{10}	Projektion auf $\bar{}$;y
f_{11}	$y \to x$
f_{12}	Projektion auf $\bar{}$;x
f_{13}	$x \to y$
f_{14}	NAND
f_{15}	EINS

Beispiel:

Wenn eine 4-stellige Schaltfunktion die folgende Tabelle hat:

x1	x2	x3	x4	...f18...
0	0	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	1	1	0

so gibt es mit $|D| = 5$ genau $2^5 = 32$ Schaltfunktionen $f_0, \ldots f_{31}$.

Atom, Minterm und Maxterm

In diesem Abschnitt betrachten wir kleinste und größte Schaltfunktionen.

Definition:

$a \in A$ mit $a \neq 0$ heißt **Atom einer Booleschen Algebra** $(A, \cdot, +, \bar{})$

$\Leftrightarrow a \cdot b = a$ oder $a \cdot b = 0$, für alle $b \in A$ $\quad (a \cdot b \neq 0 \Rightarrow a \cdot b = a)$

Bemerkung:

Atome einer Booleschen Algebra sind die „kleinsten Elemente", die multipliziert mit einem anderen entweder das Element selbst oder Null ergeben.
In Mengenalgebren $\wp(A)$ sind die Atome gerade die einelementigen Mengen mit $\cap$.

Definition:
Ein **Minterm** ist eine atomare Schaltfunktion, die an genau einer Stelle 1 ist.

Bemerkung:
Hat eine Schaltfunktion in ihrer Funktionstabelle genau eine 1, so ist sie ein Minterm:

$x_1 \dots x_n$	f
	:
$\varepsilon_1 \dots \varepsilon_n$	0
	1
	0
	:

Die Schaltfunktion f hat dann die Form:

$$f = x_1^{\varepsilon_1} \cdot x_2^{\varepsilon_2} \cdot \dots \cdot x_n^{\varepsilon_n}, \text{ wobei:}$$
$$x_i^{\varepsilon_i} = \begin{cases} x_i; \text{wenn } \varepsilon_i = 1; \\ \overline{x_i}; \text{wenn } \varepsilon_i = 0 \end{cases}$$

Beispiel:
$f_4 = \overline{x} \cdot y$ ist Minterm in obiger Tabelle.

Ein zum Minterm dualer Begriff ist der des Maxterms, also der „größten Schaltfunktion ungleich der 1-Funktion (EINS)".

Definition:
Ein **Maxterm** ist eine atomare Schaltfunktion, die an genau einer Stelle 0 ist.

Bemerkung:
Eine Schaltfunktion, die in ihrer Funktionstabelle genau eine 0 enthält, ist ein Maxterm:

$x_1 \dots x_n$	f
	:
$\varepsilon_1 \dots \varepsilon_n$	1
	0
	1
	:

Die Schaltfunktion f hat dann die Form:

$$f = \overline{x_1}^{\varepsilon_1} + \overline{x_2}^{\varepsilon_2} + \dots + \overline{x_n}^{\varepsilon_n}, \text{ wobei:}$$

$$x_i^{-;\varepsilon_i} = \begin{cases} \overline{\ };x_i;\text{wenn } \varepsilon_i=1 \\ x_i;\text{wenn } \varepsilon_i=0 \end{cases},$$

denn: $f(u_1, \ldots u_n) = 1 \Leftrightarrow u_1 = \overline{\ };\varepsilon_1 \vee u_2 = \overline{\ };\varepsilon_2 \vee \ldots \vee u_n = \overline{\ };\varepsilon_n$ und
$f(u_1, \ldots u_n) = 0 \Leftrightarrow u_1 = \varepsilon_1 \wedge u_2 = \varepsilon_2 \wedge \ldots \wedge u_n = \varepsilon_n.$

Minterme und Maxterme ermöglichen standardisierte Darstellungen (Normalformen) von Schaltfunktionen, welche im nächsten Abschnitt vorgestellt werden.

3.2 Boolesche Ausdrücke und Normalformen

Bisher wurden Schaltfunktionen über Funktionstabellen mit Eingängen (x_1, ..., x_n) und Ausgang ($f(x_1, ..., x_n)$) definiert. Jetzt werden x_1,..., x_n und f als **Variablen** interpretiert. Wir gehen also von Schaltfunktionen mit Eingängen x_1, ..., x_n zu Booleschen Funktionen in x_1, ..., x_n über. Da Schaltfunktionen eine Boolesche Algebra bilden, dürfen wir boolesche Ausdrücke mit den Rechenregeln der Booleschen Algebra umformen.

Definition:
Sei X:={0, 1, x_1, ..., x_n} eine Menge von Booleschen Variablen mit den Atomen „0" und „1".
Boolesche Ausdrücke k-ter Stufe über X sind induktiv definiert:

Stufe 0:	Elemente von X
Stufe (k+1):	$(w_1 \cdot w_2 \cdot \ldots \cdot w_r)$,
	$(w_1 + w_2 + \ldots + w_r)$ oder
	$\overline{\ };w$,

wobei w, w_i Boolesche Ausdrücke sind mit:
$;\text{Max}_i(\text{Stufe}(w_i)) = k$ und $\text{Stufe}(w) = k$.

Bemerkung:
Hierbei ist die Stufenzahl gleich der Klammertiefe und gleich der Zahl hintereinander zu durchlaufender AND-, OR- und NOT-Schaltungen.

Zusammenhang zwischen Booleschen Ausdrücken und Schaltfunktionen

Folgende Aussagen kann man aufstellen:

A. Jedem Booleschen Ausdruck entspricht eindeutig eine Schaltfunktion
B. Zu jeder Schaltfunktion gibt es unendlich viele Boolesche Ausdrücke.

Die folgende Aussage betrifft Optimierungsfragen:

C. Aus der Menge der Booleschen Ausdrücke kann man einen suchen, der die Schaltfunktion
   - am billigsten (Kostenmaß einführen) oder

• am schnellsten (Zeitmaß einführen)
realisiert.

Zu A:

Sei B(X) = { Boolesche Ausdrücke über X = $\{x_1,...x_n\}$ }
Sei $S(B^n)$ = { Schaltfunktionen f: $B^n \mapsto B$}
Sei eine Abbildung φ: $B(X) \mapsto S(B^n)$ induktiv definiert durch:

$\varphi(0)$	=	NULL (NULL ist eine Schaltfunktion: $\underline{0}$)
$\varphi(1)$	=	EINS (EINS ist eine Schaltfunktion: $\underline{1}$)
$\varphi(x_i)$	=	x_i Projektion auf i-te Komponente
$\varphi((w_1 \cdot w_2))$	=	$AND(\varphi(w_1), \varphi(w_2))$
$\varphi((w_1 + w_2))$	=	$OR(\varphi(w_1), \varphi(w_2))$
$\varphi((\overline{\ };w))$	=	$NOT(\varphi(w))$

Die Funktionstabelle eines Booleschen Ausdrucks ist eindeutig. Umgekehrt gilt dies aber nicht.

Zu B:

Zwei Boolesche Ausdrücke heißen äquivalent, wenn sie dieselbe Schaltfunktion definieren: a, b ∈ B(X), **a äquivalent zu b** $(a \equiv b) \Leftrightarrow \varphi(a) = \varphi(b)$.
Man sieht sofort:
a äquivalent b, sofern a in b mit Hilfe der Rechenregeln der Booleschen Algebra umwandelbar ist (zum Beispiel durch die Axiome (F1)-(F5)).

Beispiel:

	$x_1;\overline{\ } \cdot x_2 + x_1 \cdot x_2;\overline{\ } + x_1 \cdot x_2$	
≡	$x_1;\overline{\ } \cdot x_2 + x_1 \cdot x_2;\overline{\ } + x_1 \cdot x_2 + x_1 \cdot x_2$	(wegen a ≡ a + a)
≡	$x_1;\overline{\ } \cdot x_2 + x_1 \cdot x_2 + x_1 \cdot x_2;\overline{\ } + x_1 \cdot x_2$	(wegen (K))
≡	$(x_1;\overline{\ } + x_1) \cdot x_2 + x_1 \cdot (x_2;\overline{\ } + x_2)$	(wegen (D))
≡	$1 \cdot x_2 + x_1 \cdot 1$	(wegen $a + a;\overline{\ } = 1$)
≡	$x_2 + x_1$	(wegen (K) und $a \cdot 1 = a$)
≡	$x_1 + x_2$	(wegen (K))

Hierbei sind alle Booleschen Ausdrücke äquivalent. Außerdem handelt es sich bei dem letzten Ausdruck offenbar um einen besonders „einfachen" und „billig zu realisierenden" Booleschen Ausdruck.

Ergänzung für unvollständig definierte Schaltfunktionen:

Ist eine n-stellige Schaltfunktion nicht auf dem gesamten Bereich definiert:
$f : B^n \supset D \mapsto B$, $B^n \setminus D \neq \emptyset$, wenn also Don´t Care-Bedingungen vorliegen,
dann heißen zwei Boolesche Ausdrücke äquivalent, wenn sie auf D dieselbe Schaltfunktion realisieren.
Don´t Cares sind beliebig besetzbar, das heißt, man darf Minterme, die sich nur auf $B^n \setminus D$ beziehen (also auf D die Nullfunktion liefern), beliebig hinzufügen oder weglassen.

Zwei Boolesche Ausdrücke liefern dieselbe Schaltfunktion $f : D \mapsto B$,
$\Leftrightarrow$ sie sind ineinander überführbar durch:

a) Regeln (F1)-(F5) der Booleschen Algebra

b) Regeln $x_1^{\varepsilon_1} \cdot x_2^{\varepsilon_2} \ldots \cdot x_n^{\varepsilon_n} \equiv 0$, falls $(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) \notin D$
für die disjunktive Normalform (siehe später).

Beispiel:

Sei eine Schaltfunktion durch folgende Funktionstabelle gegeben:

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	Don´t Care
1	1	1	Don´t Care

Ein Boolescher Ausdruck für f ist zum Beispiel:

$x_1; \overline{\quad} \cdot x_2 \cdot x_3 + x_1 \cdot \overline{\quad}; x_2 \cdot \overline{\quad}; x_3 + x_1 \cdot \overline{\quad}; x_2 \cdot x_3$

Durch Umformungen erhält man einen einfacheren Ausdruck:

$\equiv$ $x_1; \overline{\quad} \cdot x_2 \cdot x_3 + x_1 \cdot \overline{\quad}; x_2 \cdot \overline{\quad}; x_3 + x_1 \cdot \overline{\quad}; x_2 \cdot x_3 + x_1 \cdot x_2 \cdot \overline{\quad}; x_3 + x_1 \cdot x_2 \cdot x_3$

Don´t Care hinzufügen, sie werden also mit „1" besetzt

$\equiv$ $x_1; \overline{\quad} \cdot x_2 \cdot x_3 + x_1 \cdot (\overline{\quad}; x_2 \cdot \overline{\quad}; x_3 + \overline{\quad}; x_2 \cdot x_3 + x_2 \cdot \overline{\quad}; x_3 + x_2 \cdot x_3)$

$\equiv$ $x_1; \overline{\quad} \cdot x_2 \cdot x_3 + x_1$

$\equiv$ $x_1; \overline{\quad} \cdot x_2 \cdot x_3 + x_1 + x_1 \cdot x_2 \cdot x_3$ Don´t Care hinzufügen

$\equiv$ $(x_1; \overline{\quad} + x_1) \cdot x_2 \cdot x_3 + x_1$

$\equiv$ $x_2 \cdot x_3 + x_1$

Bemerkung:
Hier werden Don´t Cares hinzugefügt. Es kann auch sinnvoll sein, sie wegzulassen.
Trivialbeispiel: NULL: $D \mapsto B$, hier wäre es sinnlos, Don´t Cares auf „1" zu setzen.

Normalformen

In diesem Abschnitt werden standardisierte Darstellungen von Schaltfunktionen durch Boolesche Ausdrücke eingeführt, die aus einer Funktionstabelle (automatisch) zu entnehmen sind.

(Vollständige) disjunktive Normalform (DNF)

Hier wird ein Boolescher Ausdruck zu einer Schaltfunktion als Vereinigung der Minterme dargestellt.

Definition: Einschlägiger Index
Die Zeilennummer ε einer Funktionstabelle zu einer Schaltfunktion f heißt **einschlägiger Index** zu f : $B^n \mapsto B$, falls $f(\varepsilon_1, ..., \varepsilon_n) = 1$ ist.

Definition: ε-ter Minterm
Sei ε ein Index von f.
Dann heißt die Funktion $m_\varepsilon : B^n \mapsto B$, definiert durch:

$$m_\varepsilon(x_1, ..., x_n) := x_1^{\varepsilon_1} \cdot x_2^{\varepsilon_2} \cdot ... \cdot x_n^{\varepsilon_n}$$

ε-ter Minterm von f.
Dabei sei: $x_j^{\varepsilon_j} = \begin{cases} x_j, \text{ falls } \varepsilon_j=1 \\ \overline{x_j}, \text{ falls } \varepsilon_j=0 \end{cases}$

Definition: Disjunktive Normalform (DNF)
Sei I die Menge der einschlägigen Indizes zu einer Schaltfunktion f.
Die **Disjunktive Normalform** zu f ist dann definiert als Disjunktion von Mintermen:

$$\bigcup_{\varepsilon \in I} m_\varepsilon$$

Ein Beispiel wird nach Einführung der anderen Normalformen vorgestellt.

(Vollständige) konjunktive Normalform (KNF)

Definition: ε-ter Maxterm
Sei ε Index von $f : B^n \mapsto B$, und sei m_ε der ε-te Minterm von f.
Dann heißt die Funktion $M_\varepsilon : B^n \mapsto B$ definiert durch:

$$M_\varepsilon(x_1, ..., x_n) := \overline{m_\varepsilon(x_1, x_2, ..., x_n)}$$

ε-ter Maxterm von f.
(Kurz schreibt man: $M_\varepsilon = \overline{m_\varepsilon}$)

Eine Schaltfunktion wird als Durchschnitt der Maxterme dargestellt.
Sie läßt sich aus der DNF herleiten:

$$\text{Aus NF(f)} = \bigcup_{\{\varepsilon \mid f(\varepsilon) = 1\}} x_1^{\varepsilon_1} \cdot x_2^{\varepsilon_2} \cdot ... \cdot x_n^{\varepsilon_n}$$

folgt:

$$\text{NF}(\overline{f}) = \bigcup_{\{\varepsilon \mid f(\varepsilon) = 0\}} x_1^{\varepsilon_1} \cdot x_2^{\varepsilon_2} \cdot ... \cdot x_n^{\varepsilon_n}$$

$$= \overline{\left(\prod (x_1 \cup \overline{\varepsilon_1}) \cup (x_2 \cup \overline{\varepsilon_2}) \cup ... \cup (x_n^{\varepsilon_n} \cup \overline{\varepsilon_n}) \right)} \quad \text{(De Morgan)}$$

$\{\varepsilon \mid f(\varepsilon) = 0\}$

Somit ergibt sich die konjunktive Normalform für f:

$$NF(f) = \prod_{\{\varepsilon \mid f(\varepsilon) = 0\}} (x_1^{\varepsilon_1}; \cup x_2^{\varepsilon_2}; \cup \ldots \cup x_n^{\varepsilon_n};)$$

Definition: Konjunktive Normalform (KNF)
Sei $J = \{\varepsilon \mid f(\varepsilon) = 0\}$ eine Menge von Indizes zu einer Schaltfunktion f.
Die **Konjunktive Normalform** zu f ist dann definiert als:

$$KNF(f) = \prod_{\varepsilon \in J} (x_1^{\varepsilon_1}; \cup x_2^{\varepsilon_2}; \cup \ldots \cup x_n^{\varepsilon_n};)$$

Bemerkung:
Wenn f nur auf D definiert ist, betrachtet man nur die Produkt- bzw. Vereinigungsbildung bezüglich D.

Komplementfreie Ringsummenentwicklung (RSE)

Aus der vollständigen disjunktiven Normalform läßt sich eine weitere Normalform herleiten, die aus „⊕", „•" und „1" besteht.
Dabei stellt ⊕ das Symbol für das exklusive Oder dar. „1" ist eine Basisfunktion.
Eine Schaltfunktion f hat dann die Darstellung:

$$\begin{aligned} RSE(f) = {} & a_0 \oplus a_1 \cdot x_1 \oplus \ldots \oplus a_n \cdot x_n \oplus \\ & a_{n+1} \cdot x_1 \cdot x_2 \oplus a_{n+2} \cdot x_1 \cdot x3 \oplus .... \\ & : \\ & : \\ & a_{2^n-1} \cdot x_1 \cdot \ldots \cdot x_n \end{aligned}$$

mit $a_i \in \{0,1\}$.

Bemerkung:
Es gibt somit 2^n Disjunktionsglieder. Ferner gibt es 2^{2^n} verschiedene Schaltfunktionen, da a_i entweder mit 0 oder 1 belegt ist. Liegt die disjunktive Normalform einer Schaltfunktion f vor, so erhält man nach folgender Vorgehensweise die RSE(f):

(1) Ersetze in DNF „∪" durch „⊕"
(erlaubt, da Minterme a,b die Bedingung $a \cdot b = 0$ für $a \neq b$ erfüllen)
(2) Ersetze $\overline{x_i}$ durch $(1 \oplus x_i)$
(3) Ausmultiplizieren
(4) Zusammenfassen [$a \oplus a = 0$; $a \cdot a = a$]

Das Ergebnis ist eindeutig bestimmt und von der oben angegebenen Form.
Zum Umgang mit dem exklusiven Oder „⊕" seien folgende Rechenregeln angegeben:

(a) $x \oplus 1 = \overline{x}$, $\quad x \oplus 0 = x$
(b) $x \oplus x = 0$, $\quad x \oplus \overline{x} = 1$

(c) $x \oplus y = y \oplus x$
(d) $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
(e) $x \cdot (y \oplus z) = x \cdot y \oplus x \cdot z$
(f) $0 \oplus 0 \oplus ... \oplus 0 = 0$
(g) $1 \oplus 1 \oplus ... \oplus 1 = \begin{cases} 1; & \text{wenn n ungerade} \\ 0; & \text{wenn n gerade} \end{cases}$
n ist dabei die Anzahl der Einsen.

Beispiel:

Zum Abschluß dieses Abschnitts werden die drei Normalformen durch ein Beispiel illustriert. Sei hierzu die Schaltfunktion durch folgende Funktionstabelle gegeben:

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Disjunktive Normalform (DNF):

Minterme sind:

$m_1(x, y, z) = \bar{x} \cdot \bar{y} \cdot z$,
$m_5(x, y, z) = x \cdot \bar{y} \cdot z$ und
$m_7(x, y, z) = x \cdot y \cdot z$

$DNF(f) = \bar{x} \cdot \bar{y} \cdot z + x \cdot \bar{y} \cdot z + x \cdot y \cdot z$

Konjunktive Normalform (KNF):

$KNF(f) = (x+y+z) \cdot (x+\bar{y}+z) \cdot (x+\bar{y}+\bar{z}) \cdot (\bar{x}+y+z) \cdot (\bar{x}+\bar{y}+z)$

Komplementfreie Ringsummenentwicklung:

$RSE(f) := \bar{x} \cdot \bar{y} \cdot z \oplus x \cdot \bar{y} \cdot z \oplus x \cdot y \cdot z$ (1)
$:= (1 \oplus x) \cdot (1 \oplus y) \cdot z \oplus x \cdot (1 \oplus y) \cdot z \oplus x \cdot y \cdot z$ (2)
$:= z \oplus y \cdot z \oplus x \cdot z \oplus x \cdot y \cdot z \oplus x \cdot z \oplus x \cdot y \cdot z \oplus x \cdot y \cdot z$ (3)
$:= z \oplus y \cdot z \oplus x \cdot y \cdot z$ (4)

Es gilt: $\varphi(DNF(f)) = \varphi(KNF(f)) = \varphi(RSE(f))$

3.3 Synthese von Schaltkreisen, Minimierung

Für jede Schaltfunktion gibt es unendlich viele Boolesche Ausdrücke und damit unendlich viele Schaltkreise, welche sie realisieren.

Beispiel:

$\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2 + x_1 \cdot x_2$ und $x_1 + x_2$ sind Ausdrücke für dieselbe

Schaltfunktion „Oder".

Um den einfachsten und „billigsten" Booleschen Ausdruck und den ihn realisierenden Schaltkreis (bzgl. AND, OR und NOT) zu finden, muß man ein **Kostenmaß** definieren.
Wir setzen dazu (sehr vereinfachend): Kosten Boolescher Ausdrücke = **Anzahl der Eingänge** in AND- beziehungsweise OR-Gatter des entsprechenden Schaltkreises.
Die Negationen, Überkreuzungen, Leitungsduplikationen, Leitungslängen, Ausgänge, Gatter usw. bleiben unberücksichtigt.

Definition:
Eine **Kostenfunktion** $k: B(X) \mapsto IN_0$ sei definiert durch:

- $k(x_i) = k(0) = k(1) = 0$ (Leitungen kostenlos)
- $k(w_1 + ... + w_r) = k(w_1 \cdot ... \cdot w_r) = \backslash I \backslash SU(i=1;r; k(w_i)) + r$
  (r = Zahl der Eingänge in das Gatter dieser Stufe)
- $k(\bar{}\,;w) = k(w)$ (Negation ist kostenlos)

Bemerkung:
Nach dieser Definition kostet eine Konjunktion (AND) beziehungsweise Disjunktion (OR) mit r Eingängen r Einheiten zuzüglich der Kosten vorgeschalteter Schaltkreise (Gatter).

Beispiel:

$k(x_1 \cdot (x_2+x_3) \cdot x_4 + x_1)$

$= k(x_1 \cdot (x_2+x_3) \cdot x_4) + k(x_1) + 2$
$= k(x_1) + k((x_2+x_3)) + k(x_4) + 3 + k(x_1) + 2$
$= k(x_1) + k(x_2) + k(x_3) + 2 + k(x_4) + 3 + k(x_1) + 2$
$= 0 + 0 + 0 + 2 + 0 + 3 + 0 + 2$
$= 7$

Allgemeines Minimierungsproblem

Zu einer Schaltfunktion $f : D \mapsto B$ gibt es unendlich viele Boolesche Ausdrücke a_1, a_2, ... mit $\varphi(a_i) = f$.
Gesucht ist nun ein Ausdruck $a^* \in \varphi^{-1}(f)$ mit den geringsten Kosten bezüglich obiger Definition: $k(a^*) \leq k(b)$ für alle $b \in \varphi^{-1}(f)$.
Dieses Problem ist bisher ungelöst.

Eingeschränktes Minimierungsproblem

Bestimme eine billigste **zweistufige** Realisierung, das heißt, eine Realisierung, bei der nur maximal zwei AND- beziehungsweise OR-Gatter hintereinander durchlaufen werden. Die konjunktive und die disjunktive Normalform gehören zu diesen zweistufigen Realisierungen.

Im folgenden beschränken wir uns auf zweistufige Realisierungen, welche aus der disjunktiven Normalform ableitbar sind. Hierbei wird erst ein AND- und dann ein OR-Gatter durchlaufen. Solche Realisierungen heißen **Polynome**.

Definition:
P(X) ist die Menge der polynomialen Booleschen Ausdrücke über den Booleschen Variablen $X = \{x_1, \ldots, x_r\}$:

$P(X) = \{ p \mid p = a_1 + \ldots + a_m;\ p=1;\ p=0 \text{ mit: } a_i = x_{i_1}^{\varepsilon_{i_1}} \cdot \ldots \cdot x_{i_r}^{\varepsilon_{i_r}} \}$.

a_i heißt **Monom**.

Ein **billigstes Polynom** zur Realisierung von f heißt **Minimalpolynom** von f.

Ein Monom a heißt **Implikant** von $f \Leftrightarrow \varphi(a) \leq f$ („≤" komponentenweise)

Beispiel:

Ein solches Polynom hat die Form: $p = \bar{x}_1 + x_2 \cdot \bar{x}_3$.
Wie man sieht, handelt es sich um einen klammerfreien Ausdruck.

Zu der Schaltfunktion f, die durch die Tabelle

x_1	x_2	x_3	f	$\varphi(a_1)$	$\varphi(a_2)$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	**1**	0	0
0	1	1	**1**	0	0
1	0	0	**1**	**1**	0
1	0	1	0	0	**1**
1	1	0	**1**	**1**	0
1	1	1	0	0	0

gegeben wird, ist

$$a_1 = \ldots = x_1 \cdot \overline{\quad} ; x_3, \qquad a_2 = x_1 \cdot \overline{\quad} ; x_2 \cdot x_3.$$

a_1 ist Implikant von f; a_2 nicht: $\varphi(a_2(1,0,1)) = 1 \not\leq 0 = \varphi(f(1,0,1))$.

Bemerkung:
Alle Minterme einer Schaltfunktion sind auch Implikanten derselben. Kürzere Implikanten überdecken mehr Einsen von f und sind „billiger" als längere. Da ein Polynom von f gerade eine Überdeckung von f durch Implikanten ist, werden für eine billige Realisierung **kürzeste Implikanten** gesucht:

Definition:
a heißt **Primimplikant von f** $\Leftrightarrow$
a ist Implikant von f und kein Teilmonom von a ist Implikant.

Beispiel:

Seien $x \cdot \alpha \cdot y$ und $x \cdot \bar{\alpha} \cdot y$ Implikanten von f. Sie erfüllen also:

$$\varphi(\, x \cdot \alpha \cdot y \,) \le f \text{ und } \varphi(\, x \cdot \overline{\ }; \alpha \cdot y \,) \le f.$$

Dann erhält man einen minimalen Implikanten von f:

$\Rightarrow \varphi(x \cdot \alpha \cdot y + x \cdot \overline{\ }; \alpha \cdot y) \le f$	Kosten: $2 \cdot (k(x)+k(y)+k(\alpha)) + 8$
$\Rightarrow \varphi(x \cdot (\alpha + \overline{\ }; \alpha) \cdot y) \le f$	
$\Rightarrow \varphi(x \cdot y) \le f$	Kosten: $k(x)+k(y) + 2$

Quine-McCluskey-Algorithmus

Es wird nun ein Algorithmus vorgestellt, der eine Lösung des eingeschränkten Minimierungsproblems ist.

Methode:

a) Bestimme alle Primimplikanten von f.

b) Setze hieraus eine billigste Überdeckung von f zusammen, das heißt, suche eine billigste Gesamtüberdeckung der Minterme (ohne jene, die durch Don´t Cares entstanden sind) durch geschickte Auswahl einer Menge von Primimplikanten.

Zusatz zu a):
Man setze alle Don´t Cares von f auf 1. Dadurch erhält man mehr Einsen in f, wodurch man mehr beziehungsweise kürzere Primimplikanten angeben kann (siehe obiges Beispiel). Daher kann sich diese Verwendung der Don´t Cares allenfalls positiv auf die Realisierungskosten auswirken.
Für kleine Variablenzahl n ($f : B^n \supset D \mapsto B$) kann man Implikanten und somit Primimplikanten mit Hilfe von Diagrammen grafisch finden (Karnaugh-Diagramm). Für größere n braucht man eine systematischere Technik. Eine solche wird nun vorgestellt und anschließend an einem Beispiel erläutert.

Zu a): Prinzip zum Auffinden von Primimplikanten

Starte mit Mintermen und Don't Cares von f.
Nutze die Nachbarschaftsbeziehung — wie oben gezeigt:
$x \cdot \alpha \cdot y + x \cdot \overline{\ }; \alpha \cdot y \rightarrow x \cdot y$ — so oft wie möglich aus.

Dieses Verfahren liefert alle Primimplikanten.

Zur Systematik:

1. Ersetze $x_i^{\varepsilon_i}$ durch ε_i und ergänze fehlende Variablen durch „–" an der
entsprechenden Stelle.
$\overline{\ }; x_1 \cdot x_4 \cdot \overline{\ }; x_6$ entspricht „0 – – 1 – 0".
2. Nachbarn sind nur in Monomen möglich, die sich bezüglich ihres "Gewichts" (Anzahl der enthaltenen Einsen) um genau Eins unter-

scheiden. Daher sollte man die Monome nach dem Gewicht vorsortieren. Dazu kann z.B. das **Hamming-Gewicht** eines binären Vektors v_i genutzt werden:

$w(v) = \I\SU(i=1;n;v_i)$ (*weight*)

Zu b): Überdeckung von f aus Primimplikanten (PI):

1. Man erstelle eine Matrix mit:

   - Primimplikanten p_j als Zeilen und
   - Mintermen m_i der DNF (ohne Don't Cares) als Spalten.

   Der Matrixeintrag ist $a_{jr} = 1 \Leftrightarrow$ Primimplikant p_j überdeckt Minterm m_r: Der Minterm stimmt in allen Komponenten mit dem Primimplikanten überein. An Stellen, wo der Primimplikant ein „–" hat, kann im Minterm an gleicher Stelle beliebiges („1" oder „0") stehen.

Beispiel:

1–1– überdeckt 1110, 1010, 1011 und 1111.

00— überdeckt nicht 1011, denn es muß gelten:

$\varphi(\bar{\ };a\cdot\bar{\ };b) \geq \varphi(a\cdot\bar{\ };b\cdot c\cdot d)$.

Mit a=1, b=0, c=1 und d=1 ist die „rechte Seite", aber nicht die „linke Seite" erfüllt.

	m_1	m_2	m_3	. . .
p_1		1		
p_2			1	
p_3		1		
:				1

Abbildung 3.10: Überdeckungsmatrix

2. Ein Primimplikant heißt **wesentlich** $\Leftrightarrow$ Es gibt einen Minterm m_i, der **nur** von diesem Primimplikanten überdeckt wird. In der entsprechenden Matrixspalte befindet sich nur eine 1. Ein wesentlicher Primimplikant muß zum Minimalpolynom gehören, er ist unverzichtbar. Alle zusätzlich überdeckten Minterme der Matrix sind zu streichen, weil sie schon überdeckt sind.
3. Ist $\varphi(m_i) \leq \varphi(m_j)$, so streiche m_j, denn jede Überdeckung von m_i liefert m_j mit.

	. . .	m_i	. . .	m_j
p_1		1		1
p_2				
p_3		1		1
p_4				1
:				

Abbildung 3.10: Streiche m_j

4. Ist $\varphi(p_r) \leq \varphi(p_s)$ (p_s „leistet mehr" als p_r) und $k(p_r) \geq k(p_s)$, so streiche p_r, denn p_s ist billiger und leistungsfähiger.

	m_1	m_2	m_3	. . .
:				
p_r		1	1	
:				
p_s		1	1	1
:				

Abbildung 3.11: p_r streichen

Mit den Schritten 2, 3 und 4 (evtl. iteriert) wird die Bestimmung einer Überdeckung oft ein eindeutig bestimmter Vorgang. Manchmal müssen aber mehrere Möglichkeiten geprüft und eine insgesamt billigste Lösung ausgewählt werden.

Beispiel:

In diesem Beispiel sollen Boolesche Ausdrücke für die Realisierung eines Addierers zweistelliger Gray-Code-Zahlen x_1x_2 und y_1y_2 mittels Quine-McCluskey-Algorithmus bestimmt werden.

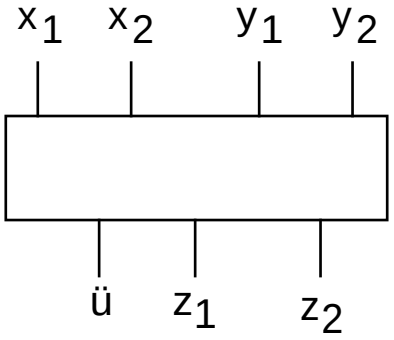


Abbildung 3.12: Gray-Code-Addierer

Die Schaltfunktion sei durch folgende Funktionstabelle gegeben:

x1	x2	y1	y2	Ü	z1	z2
0	0	0	0	0	0	0
0	0	0	1	0	0	**1**
0	0	1	1	0	**1**	**1**
0	0	1	0	0	**1**	0
0	1	0	0	0	0	**1**
0	1	0	1	0	**1**	**1**
0	1	1	1	0	**1**	0
0	1	1	0	**1**	**1**	0

1	1	0	0	0	**1**	**1**
1	1	0	1	0	**1**	0
1	1	1	1	**1**	**1**	0
1	1	1	0	**1**	**1**	**1**
1	0	0	0	0	**1**	0
1	0	0	1	**1**	**1**	0
1	0	1	1	**1**	**1**	**1**
1	0	1	0	**1**	0	**1**

Abbildung 3.13: Schaltfunktion für den Gray-Code-Addierer

Schaltfunktion und Minimalpolynom für Ü:
Sei I_n die Tabelle der Implikanten der Länge n, wobei sich die Länge aus der Zahl an „0"en und „1"en ergibt. Die Minterme sind in der Tabelle zeilenweise nach Gewicht sortiert.

I_4	I_3	I_2
1001	***10–1***	***1–1–***
1010	101–	
0110	**1–10**	
	–110	
1011	**1–11**	
1110	111–	
1111		

Man vergleiche jeden Minterm einer „Gewichtsgruppe" mit jedem der darunter befindlichen Gruppe. Falls sich diese an genau einer Stelle unterscheiden, so schreibe man an diese Stelle einen „–": 1001 und 1011 führen zu 10–1. Befinden sich in den Mintermen „–", so müssen diese an den gleichen Stellen auftreten: 1–10 und 1–11 ergibt 1–1–. Als Primimplikanten erhält man diejenigen Implikanten, welche nicht weiter reduziert werden können, hier : 10–1, –110 und 1–1–. Die Überdeckungsmatrix lautet:

	1001	1010	0110	1011	1110	1111
10–1	**1**			1		
–110			**1**		1	
1–1–		**1**		1	1	1

Wie man erkennt, sind alle Primimplikanten wesentlich.
Das Minimalpolynom ist also:

$$MP_{ü} = x_1 \cdot y_1 + x_1 \cdot x_2 \cdot \overline{y_2} + x_2 \cdot y_1 \cdot y_2 \cdot \overline{\ }.$$

Schaltfunktion und Minimalpolynom für z_1:

Zunächst die Primimplikanten ermitteln:

I_4	I_3	I_2
0010	001–	*0–1–*
1000	0–10	*1–0–*
	1–00	
	100–	
1100	110–	*11––*
0101	11–0	*–1–1*
1001	–101	*1––1*
0011	01–1	*––11*
0110	1–01	*–11–*
	10–1	
	0–11	
	–011	
	011–	
	–110	
1101	11–1	
0111	–111	
1011	1–11	
1110	111–	
1111		

Überdeckungsmatrix:

	0010	**1000**	1100	**0101**	1001	0110	0011	1011	1101	0111	1110	1111
0–1–	**1**					**1**	**1**			**1**		
1–0–		**1**	**1**		**1**				**1**			
11– –			1						1		1	1
–1–1				**1**					**1**	**1**		**1**
1– –1					1			1	1			1
– –11							1	1		1		1
–11–						1				1	1	1
								↑ nicht überdeckt			↑ nicht überdeckt	

Wesentliche Primimplikanten sind: 0–1–, 1–0– und –1–1.
Sie überdecken alle Minterme bis auf 1011 und 1110.

Nach Streichen der bisher überdeckten Minterme bleibt:

	1011	1110
11––		1
1––1	1	
––11	1	
–11–		1

Jede Überdeckung benutzt zwei Primimplikanten und ist gleich teuer,

so daß die Wahl beliebig ist.
Das Minimalpolynom hat die Form:

$$MP_{z_1} = x_1;\overline{\ }\cdot y_1 + x_1 \cdot y_1;\overline{\ } + x_2 \cdot y_2 + Y \text{ mit:}$$
$$Y \in \{x_1 \cdot x_2 + x_1 \cdot y_2,\ x_1 \cdot x_2 + y_1 \cdot y_2,\ x_2 \cdot y_1 + x_1 \cdot y_2,\ x_2 \cdot y_1 + y_1 \cdot y_2\}.$$

Schaltfunktion und Minimalpolynom für z_2:

Die Primimplikanten ergeben sich durch folgende Tabelle:

I_4	I_3
0100	*010–*
0001	*–100*
	0–01
	00–1
0101	*11–0*
0011	*101–*
1100	*1–10*
1010	*–011*
1011	
1110	

Alle Implikanten der letzten Spalte sind Primimplikanten.
Überdeckungsmatrix:

	0100	0001	0101	0011	1100	1010	1011	1110
010–	1		1					
–100	1				1			
0–01		1	1					
00–1		1		1				
11–0					1			1
101–						1	1	
1–10						1		1
–011				1			1	

In jeder Spalte stehen zwei (d.h. mehr als eine) Einsen, also liegt kein wesentlicher Primimplikant vor. Da kein Primimplikant wesentlich ist (alle gleich teuer), beginnen wir mit 010-:

	0100	0001	0101	0011	1100	**1010**	**1011**	1110	Kommentar
010–	1		1						1.) Start: 0100 und 0101 überdeckt
–100	1				1				
0–01		1	1						
00–1		1		1					2.) 0001 und 0011 überdeckt
11–0					1			1	3.) 1100 und 1110 überdeckt
101–						**1**	**1**		Noch nicht abgedeckt:

1–10						**1**		1	1010 und 1011
–011				1			**1**		

Von 101–, 1–10 und –011 wählt man 101–, weil dieser Primimplikant 1010 und 1011 überdeckt und diejenigen Minterme, welche 1–10 und –011 überdecken, bereits abgedeckt sind. Wesentliche Primimplikanten sind somit: 010–, 00–1, 11–0 und 101–. Ausgehend von 010– (entspricht $\overline{x_1}\,x_2\,\overline{y_1}$) ergibt sich folgendes eindeutiges Minimalpoly-nom:

$$MP_{z_2} = x_1 \cdot \overline{x_2} \cdot y_1 \cdot \overline{y_2} + x_1 \cdot \overline{x_2} \cdot y_2 + x_1 \cdot x_2 \cdot y_2 \cdot \overline{y_1} + x_1 \cdot x_2 \cdot \overline{y_1}.$$

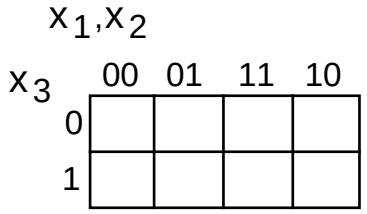
Beginnt man mit –100 statt mit 010–, dann entsteht in eindeutiger Weise:

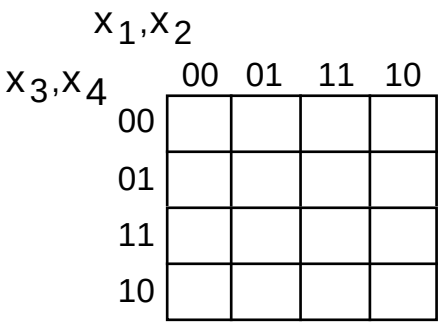
$$MP_{z_2} = x_2 \cdot y_1 \cdot \overline{y_2} \cdot \overline{y_1} + x_1 \cdot \overline{y_1} \cdot \overline{y_2} + x_2 \cdot \overline{y_1} \cdot y_2 + x_1 \cdot y_1 \cdot y_2 \cdot \overline{y_1}.$$

Beide Minimalpolynome sind gleich teuer.

Karnaugh-Diagramme

Ein Karnaugh-Diagramm einer Booleschen Funktion $f:B^n \rightarrow B$ mit $n \in \{3,4\}$ ist eine graphische Darstellung der Funktionstabelle von f durch eine (0-1)-Matrix der Größe 2×4 für n=3 bzw. 4×4 für n=4, deren Spalten mit den möglichen Belegungen der Variablen x_1 und x_2 und deren Zeilen mit den Belegungen der Variablen x_3 (n=3) bzw. x_3 und x_4 (n=4) beschriftet sind. Die Reihenfolge der Beschriftung erfolgt so, daß sich zwei zyklisch benachbarte Zeilen/Spalten nur in genau einer Komponente unterscheiden (**Gray-Code**).





Vorgehen zur Bestimmung eines Minimalpolynoms:

- An den Matrixelementen, deren Beschriftung ein einschlägiger Index für die Funktion f ist, eine 1 eintragen.
- Maximale 1er-Blöcke der Form $2^s * 2^r$ finden, so daß jede 1 mindestens einmal überdeckt ist. Dabei müssen auch zyklische Überdeckungen beachtet werden.
- Summe über den Termen, die den Blöcken entsprechen, bilden.
- Falls möglich, zusätzlich Don´t Cares ausnutzen, indem im ersten Schritt in den entsprechenden Feldern auch der Wert 1 zugeordnet wird. Damit sind mehr

Einsen vorhanden und evtl. größere Resolutionsblöcke möglich. Don´t Cares brauchen allerdings nicht vollständig überdeckt zu werden.

Bsp.:

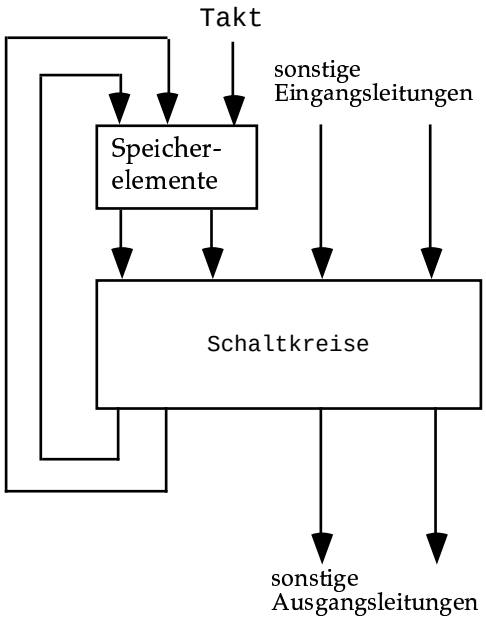


Abbildung 3.14: Schaltkreis und Schaltwerk

Schaltwerke befinden sich zu gegebener Zeit in einem Zustand Q. Ein solcher Zustand ist etwa durch die Werte der Ausgangsleitungen zu diesem Zeitpunkt charakterisiert. Zustände ändern sich nur zu bestimmten diskreten Zeitpunkten, bei Eintreffen eines Taktes, das heißt, wenn eine 1 auf der Taktleitung anliegt.

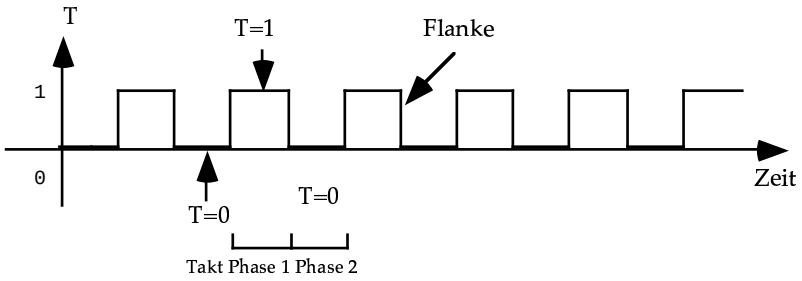


Abbildung 3.15: Takt

Speicherelemente

Binäre Speicherelemente sind kleinste logische Bausteine zur Aufbewahrung (Speicherung) einer Information. Es wird hier nur einer der Zustände „0" oder „1" gespeichert. Sie sind als Schaltwerke realisiert. Dabei ist ein Zustand $Q^{(n)}$ durch den gespeicherten Wert und durch die Werte auf den Ausgangsleitungen im n-ten Taktzyklus bestimmt. Der gespeicherte Wert steht über eine Ausgangsleitung zur Verfügung; meist gibt es eine zusätzliche Leitung, an der $\overline{Q^{(n)}}$ anliegt.

Speicherelement RS-Flipflop

Dieses Speicherelement besitzt zwei besondere Eingänge zum Setzen (auf „1"; Set) und Rücksetzen (auf „0"; Reset) des Inhalts.

Bemerkung:

Der Name Flipflop rührt von der Geräuschentwicklung der ersten Relais-Schaltwerke her, die zur Realisierung von Speicherelementen verwendet wurden.

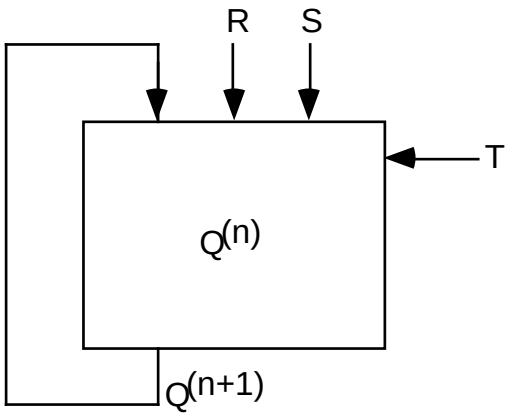


Abbildung 3.16: RS-Flipflop

Sei das RS-Flipflop durch folgende Tabelle beschrieben.
Sei T der Takt mit:

T=0: Zustand ändert sich nicht,
T=1: Zustand kann sich ändern.

R	S	$Q^{(n+1)}$
0	0	$Q^{(n)}$
0	1	1
1	0	0
1	1	Undefiniert und somit Don´t Care

Unter Einbeziehung des jeweils aktuellen Zustandes $Q^{(n)}$ ergibt sich folgende Tabelle für die Schaltfunktion von $Q^{(n+1)}$, wenn ein Takt anliegt (T=1), sonst (T=0) ändert sich nichts:

R	S	$Q^{(n)}$	$Q^{(n+1)}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	Undefiniert
1	1	1	und somit Don´t Care

Schaltfunktion für $Q^{(n+1)}$:

$$
\begin{aligned}
Q^{(n+1)} &= \overline{\ };T\cdot Q^{(n)} + T\cdot(S + \overline{\ };R\cdot Q^{(n)}) \qquad \text{(Torschaltung, if T=0 then...else...)}\\
&= \overline{\ };T\cdot Q^{(n)} + T\cdot S + T\cdot\overline{\ };R\cdot Q^{(n)} + \overline{\ };T\cdot\overline{\ };R\cdot Q^{(n)}\\
&= \overline{\ };T\cdot Q^{(n)} + T\cdot S + \overline{\ };R\cdot Q^{(n)}\\
&= (\overline{\ };T + \overline{\ };R)\cdot Q^{(n)} + T\cdot S
\end{aligned}
$$

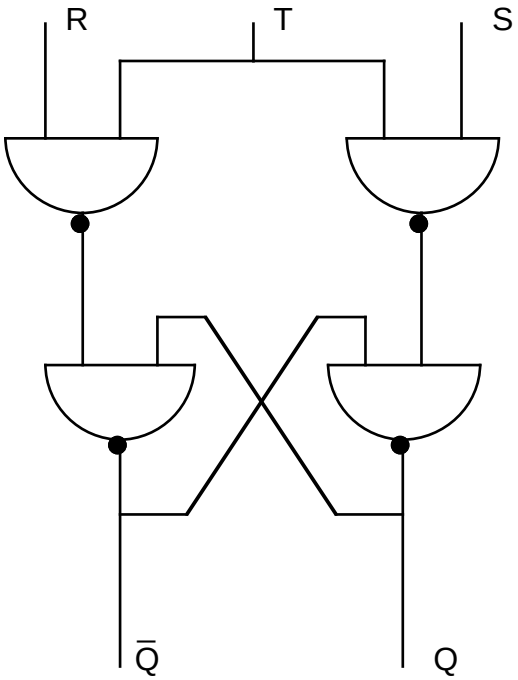
$$
= \overline{\qquad\qquad};\ \overline{\qquad\qquad};\ \overline{\qquad\qquad};(T\cdot R)\cdot Q^{(n)}\cdot\overline{\qquad};T\cdot S
$$

(NAND-Darstellung)

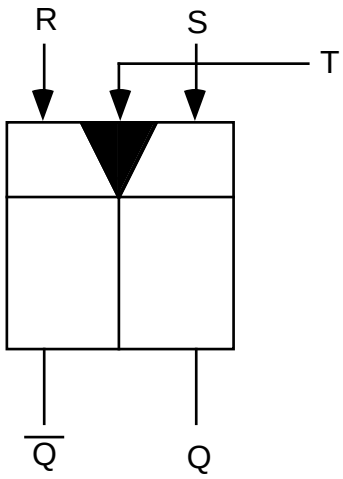
Analog für $\overline{\qquad};Q^{(n+1)}$:

$$
\overline{\backslash O(\qquad};Q^{(n+1)}) = \overline{\qquad\qquad};\ \overline{\qquad\qquad};\ \overline{\qquad\qquad};(T\cdot S)\cdot Q^{(n)};\ \overline{\qquad}\cdot\overline{\qquad\qquad};(T\cdot R)
$$

Schaltung:



Symbol:



oder einfach:

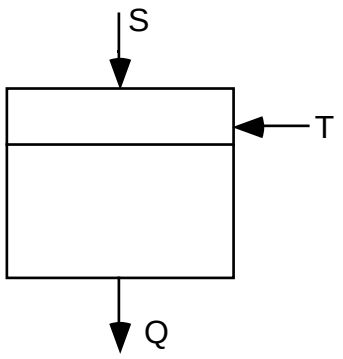


Abbildung 3.17: Schaltung und Schaltsymbol für das RS-Flipflop

Bemerkung:
Entscheidend für die Speichereigenschaft des Elements ist die **Rückkopplung** innerhalb der Schaltung. Der konstante Wert Q zirkuliert, solange T nicht geändert wird.

Speicherelement Earle Latch

Hierbei handelt es sich um ein vereinfachtes Speicherelement. Es speichert den alten Zustand solange T=0 und übernimmt den neuen Zustand von der Eingangsleitung D, sobald T=1. Dabei darf sich D nicht ändern, während T=1 ist.

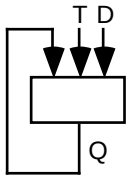


Abbildung 3.18: Earle Latch

$$Q^{(n+1)} := \{ Q^{(n)}; T=0; D; T=1$$

Die Schaltfunktion wird durch folgenden Booleschen Ausdruck beschrieben:

$$\begin{aligned} Q^{(n+1)} &= \bar{T};T \cdot Q^{(n)} + T \cdot D \\ &= \bar{T};T \cdot Q^{(n)} + T \cdot D + D \cdot Q^{(n)} \end{aligned}$$

$D \cdot Q^{(n)}$ ist mathematisch gesehen überflüssig, da er in den beiden anderen Termen abgedeckt wird. Er wird jedoch für die technische Realisierung — insbesondere im Hinblick auf den Takt — benötigt.

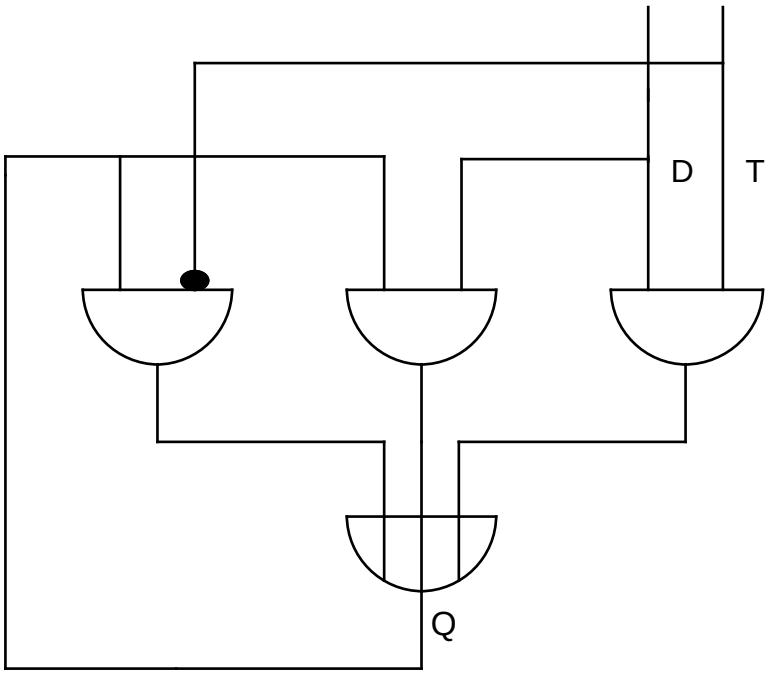


Abbildung 3.19: Schaltung des Earle Latch

Verwendung des Zusatzterms $D \cdot Q^{(n)}$:
Hierfür sind zwei Gründe zu nennen:

1) **Hazards:**
Eine Realisierung wird nie perfekt dem theoretischen Modell genügen. So können bei der Taktung Ungenauigkeiten (Laufzeitunterschiede) auftreten.

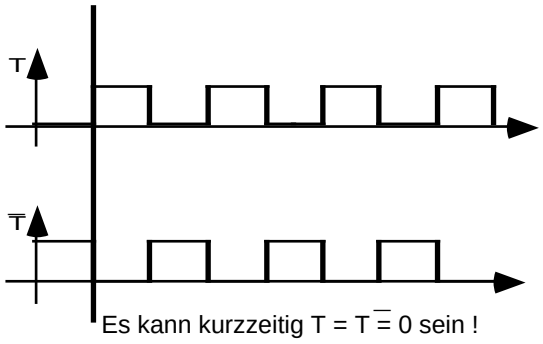


Abb. 3.20: Takt

2) **Nicht perfekte Taktflanken:**
Ein weiteres Problem besteht darin, daß die Taktflanken, die beim Wechsel des Taktes von T=0 auf T=1 entstehen, nicht senkrecht sind. Daher ist die Interpretation des Wertes von T ungenau über eine Zeitspanne nicht vernachlässigbarer Länge.

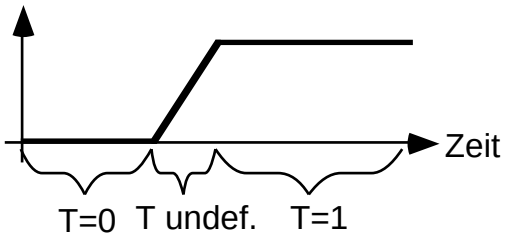


Abb. 3.21: Taktflanke

Ohne $D \cdot Q^{(n)}$ könnte nach der ersten Taktphase (T=1) kurzzeitig $T=\overline{\ };T=0$ gelten und damit $Q^{(n+1)} = \overline{\ };T \cdot Q + T \cdot D = 0$ unabhängig von D, in der zweiten Taktphase $(\overline{\ };T=1)$ gilt: $Q^{(n+1)} = \overline{\ };T \cdot Q \cup T \cdot D = 0$ (unabhängig von D).

4. Die Arithmetisch-Logische Einheit

In diesem Kapitel werden unter Anwendung der in Kapitel 3 eingeführten Begriffe Rechenwerke für die Addition, Multiplikation und Division besprochen. Dabei werden insbesondere für die Addition verschiedene Methoden vorgestellt.

4.1 Einführung

Zentraler Bestandteil der **Arithmetisch-Logischen Einheit** (ALU, *Arithmetic-Logical Unit*) ist eine schnelle Hardware zur Ausführung der arithmetischen Grundfunktionen:

- Addition (Subtraktion),
- Multiplikation und
- Division.

Interessant sind neben der Ausführungsgeschwindigkeit auch die Kosten der Bestandteile, also der Rechenwerke. Beide Kriterien lassen sich durch die Methode der Addition (beziehungsweise Multiplikation oder Division) entscheidend beeinflussen.
Die einfachste Additionsmethode bei Stellenwertcodierung (im 2-Komplement) ist aus der Schule bekannt. Man addiert zwei ganzzahlige, n-stellige Summanden komponentenweise und reicht den Übertrag zur nächsten Komponente weiter:

Summand 1: $a_{(n-1)}a_{(n-2)} \ldots a_0$,
Summand 2: $b_{(n-1)}b_{(n-2)} \ldots b_0$ und
Summe: $s_{(n-1)}s_{(n-2)} \ldots s_0$.

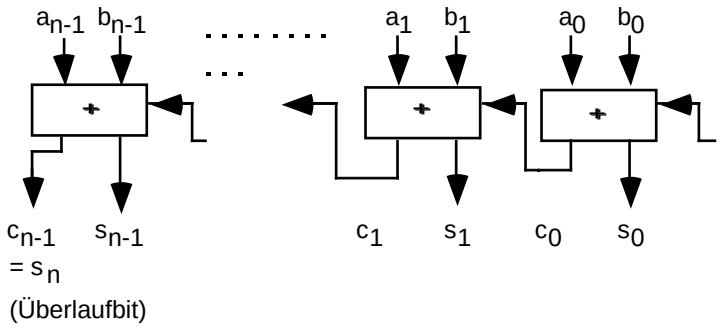


Abbildung 4.1: Addition nach „Schulmethode"

Beispiel:

```
  + 10110
    10011
 [1]01001
```

Bemerkung:
Es gibt Prinzipien für redundante Codierungen, in denen zum Beispiel ein Übertrag in Abschnitten aufgefangen wird.

4.2 Halb- und Volladdierer

Im folgenden werden zwei Additionskomponenten eingeführt, aus denen sich viele Rechenwerke zusammensetzen. Es handelt sich dabei um den **Halbaddierer** (*Halfadder* — HA) und den **Volladdierer** (*Fulladder* — FA).

Halbaddierer (Halfadder)

Der Halbaddierer wird als Schaltkomponente durch folgendes Symbol notiert:

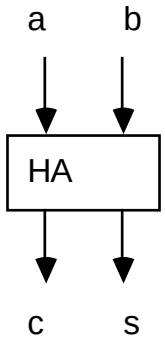


Abbildung 4.2: Symbol für einen Halbaddierer

Um den Halbaddierer durch eine Schaltung zu realisieren, muß zunächst ein Boolescher Ausdruck entwickelt werden:
Für den Übertrag c (*Carry*) gilt:

$$c = 1 \Leftrightarrow a = b = 1.$$

Also lautet das Minimalpolynom für c: $c = a \cdot b$.
Für die Summe s erhält man:

$$s = 1 \Leftrightarrow (a=1 \wedge b=0) \vee (a=0 \wedge b=1) \equiv a \oplus b.$$

Das Minimalpolynom lautet: $s = a \cdot \overline{b} + \overline{a} \cdot b$.

Obwohl es sich um eine Minimalpolynomdarstellung handelt, ist diese Darstellung nicht kostenminimal:

	Übertrag c	Summe s
Laufzeit (Gatterstufen)	1	2
Kosten nach Def. aus Kap. 3 (Zahl der Eingänge)	2	6

Eine bessere Realisierung wird erreicht durch:

$$s = a \cdot \overline{b} + \overline{a} \cdot b \equiv (a + b) \cdot (\overline{a} + \overline{b}) \equiv (a + b) \cdot \overline{c}, \text{ weil } c = a \cdot b.$$

Man beachte, daß es sich nicht um ein Minimalpolynom handelt, weil kein klammerfreier Ausdruck vorliegt.

	Übertrag c	Summe s
Laufzeit (Gatterstufen)	1	2
Kosten nach Definition aus Kapitel 3 (Zahl der Eingänge)	2	4

Der Halbaddierer läßt sich durch folgende **Schaltung** realisieren:

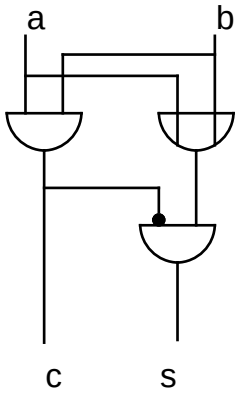


Abbildung 4.3: Schaltung für den Halbaddierer

Volladdierer (Fulladder)

Der Volladdierer verfügt über einen weiteren Eingang. Dieser ist verwendbar, um einen Übertrag aus der Summierung der vorhergehenden Komponenten zu verarbeiten. Ein Volladdierer wird durch zwei Halbaddierer realisiert (Klammerung: $(a+b)+c_{in}$).

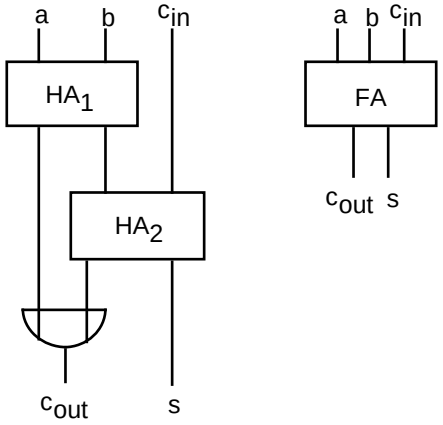


Abbildung 4.4: Schaltung und Schaltsymbol

Die **Kosten** ergeben sich aus den Kosten der Halbaddierer und des OR-Gatters:
$2 \cdot \text{Kosten}_{HA} + 2 = 14$.
Die **Laufzeit** hängt ebenso von der der Halbaddierer ab: 4 Stufen, wenn alle Eingänge gleichzeitig anliegen; 2 Stufen, wenn c_{in} später eintrifft.

Logik:

Als Boolesche Ausdrücke für den Übertrag und die Summe erhält man:

$$c_{out} = 1 \Leftrightarrow a + b + c_{in} \geq 2$$

also:

$$c_{out} = a \cdot b + (a \oplus b) \cdot c_{in} = a \cdot b + (a+b) \cdot c_{in}$$
$$s = (a \oplus b) \oplus c_{in} = \overline{a} \cdot \overline{b} \cdot c_{in} + \overline{a} \cdot b \cdot \overline{c_{in}} + a \cdot \overline{b} \cdot \overline{c_{in}} + a \cdot b \cdot c_{in}$$

Die Kosten sind k(s) = 16, das heißt, dieser Ausdruck ist nicht empfehlenswert.

4.3 Carry-Ripple-Addition

Die Carry-Ripple-Addition verwendet ein Addierwerk, welches gemäß der Abbildung aus n in Serie geschalteten Volladdierern aufgebaut ist. Es handelt sich um einen **Paralleladdierer**.
Addiert wird in der 2-Komplement-Darstellung. Bei einer Subtraktion werden alle Bits des zweiten Summanden b gekippt und c_{-1} auf 1 gesetzt. Also: a+b setzt voraus $c_{-1} := 0$ und a–b: $c_{-1} := 1$.

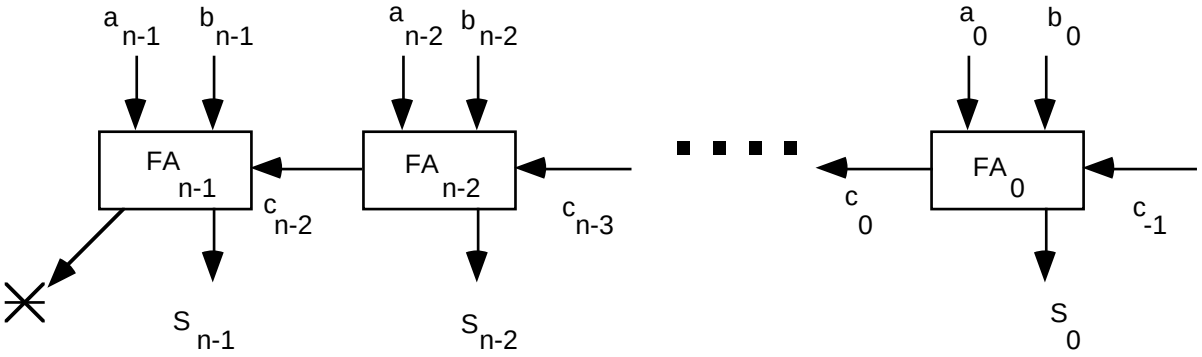


Abbildung 4.5: Carry-Ripple-Addierwerk

Logik

Der Boolesche Ausdruck für den Übertrag c_i zum Volladdierer $FA_{(i+1)}$ lautet:

$$c_i = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_{(i-1)}$$
$$= a_i \cdot b_i + (a_i + b_i) \cdot c_{(i-1)}$$

Sei zur Abkürzung: $k_i = a_i \cdot b_i$ (Konjuktion) und
$d_i = a_i + b_i$ (Disjunktion).

Dann hat der Boolesche Ausdruck die Form:

$$c_i = k_i + d_i \cdot c_{(i-1)}$$

Die verwendete Rekursionsformel hat ihren Start mit c_{-1} und ist eine Funktion:

$$c_i = f(k_i, d_i, c_{(i-1)}, ..., c_{-1}).$$

Laufzeit

Die Rekursionsformel zeigt an, daß sich Zwischenergebnisse durch Eintreffen eines Übertrags „von rechts" ändern können. Dies führt in der Schaltungzur Laufzeiterhöhung. Die **maximale Laufzeit** beträgt etwa$2 \cdot n$ Gatterstufen. Dies ist der Fall, wenn ein Übertrag ganz rechts entsteht und durch alle Stellen propagiert (fortgeleitet) wird.

Beispiel:

Sei b = (000101011) von a = (100101011) zu subtrahieren. Also ist $c_{in} = 1$, b wird gekippt und auf a addiert (2-Komplement):

```
 100101011
 111010100      cin = 1
1100000000
```

Eine **Propagationskette** ist ein Abschnitt zwischen gleichartigen Stellen 00 und 11. Innerhalb einer solchen Kette wird ein auftretender Übertrag weitergeleitet. Beim Komponentenpaar 00 wird der Übertrag aufgefangen.

Beispiel:

Die Zahlen (0101101101) und (0110100111) haben folgende Propagationsketten:

```
0 1 0 1 1 0 1 1 0 1
0 1 1 0 1 0 0 1 1 1
 2   4   2 3   3
```

Die Laufzeit beträgt $4 \cdot 2 + 2$, weil die maximale Propagationskette die Länge 4 hat. Die 2 ergibt sich, da der rechte Volladdierer zwei Gatterstufen mehr braucht.

Falls 0 und 1 gleich häufig und unabhängig voneinander auftreten, gilt: Die Länge der längsten Propagationskette ist im Mittel $\log_2(n)$ und im schlechtesten Fall n lang.

4.4 Serielle Addition

Die serielle Addition entspricht dem handschriftlichen Addieren mit Übertrag. Die zwei n-stelligen Binärzahlen werden also komponentenweise in das Addierwerk gebracht, und dort werden die Komponenten a_i und b_i addiert, so daß man die Summenkomponente s_i und gegebenenfalls einen Übertrag c erhält.

Dieser Übertrag wird dann für die Addition der Komponenten $a_{(i+1)}$ und $b_{(i+1)}$ im nächsten Takt in den Volladdierer geleitet.

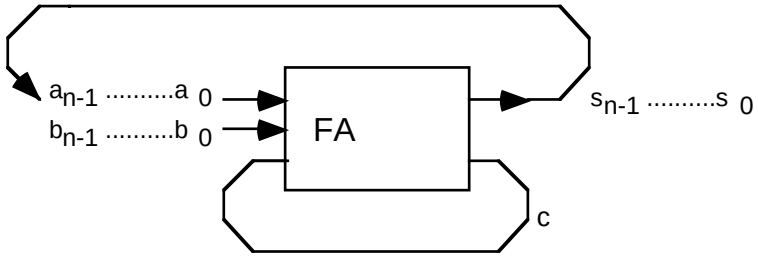


Abbildung 4.6: Serielles Addierwerk

Mikroprogramm:

Der Vorgang des seriellen Addierens läßt sich durch ein Mikroprogramm beschreiben. Sei hierzu Z eine Zählervariable, welche die Takte zählt. Sie wird mit n initialisiert. Die Steuervariable V gibt an, ob eine Addition (V=0) oder Subtraktion (V=1) durchgeführt werden soll. Durch V wird die Subtraktion im 2-Komplement garantiert. Zu beachten ist, daß die in eckigen Klammern stehenden Befehle parallel ausgeführt werden, also in einem Takt.

0: [$(a_{(n-1)} \ldots a_0)$:= Summand1; $(b_{(n-1)} \ldots b_0)$:= Summand2; Z := n; c := V]
1: [$b_i := b_i \oplus V$, für alle i = 0, ... (n–1)]
(b wird durch EXOR mit V komponentenweise gekippt;
Subtraktion im 2-Komplement.)
2: [$a_i := a_{(i+1)}$; $b_i := b_{(i+1)}$ (i=0, ... n–2);
c := Übertr. FA(a_0, b_0, c); $a_{(n-1)}$:= Summe FA(a_0, b_0, c); Z := Z–1; $b_{(n-1)} := 0$]
3: if Z > 0 then goto 2
4: Stop

In Programmzeile 2 werden (2·n+2) Operationen **parallel** ausgeführt.

Beispiel:

Seien a = (011) und b = (010) zu addieren:

1.Takt (Z=3):

01**1**	$s_1 = 1$
01**0**	c = 0

2.Takt (Z=2):

1 | 0**1**	$s_2 = 0$
0 | 0**1**	c = **1**

3.Takt (Z=1):

01 | **0**	$s_3 = 1$
00 | **0**	c = 0

Ergebnis:

101
000

4.5 Von-Neumann-Addition

Bei dem Von-Neumann-Addierwerk handelt es sich um ein einfaches getaktetes Paralleladdierwerk. Es ist aus n parallel arbeitenden Halbaddierern aufgebaut. Jedem Halbaddierer sind die Register a_i und b_i zugeordnet. Das Register 1 $a = (a_{(n-1)} \ldots a_0)$ dient der Speicherung der Zwischensumme. Es enthält auch die Endsumme. Das Register 2 $b = (b_{(n-1)} \ldots b_0)$ speichert die von den Halbaddierern gebildeten Überträge.
Summen und Überträge der HA werden solange in die Register 1 beziehungsweise Register 2 zurückgeschrieben, bis Register 2 den Wert Null enthält.

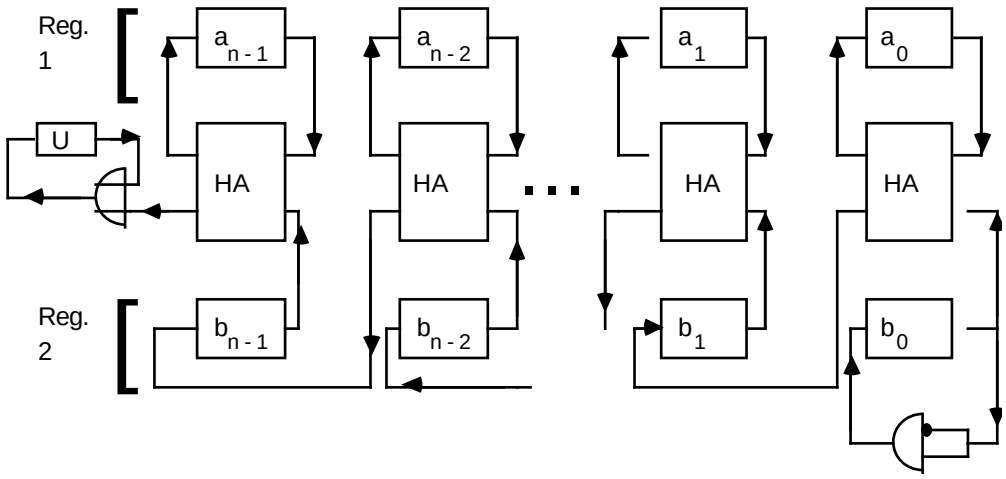


Abbildung 4.7: Von-Neumann-Addierwerk

Beispiel:

Sei dieses Verfahren auf zwei Dezimalzahlen angewandt. Sie befinden sich in den zwei Registern:

a = (98725),
b = (17282).

Es ergeben sich folgende Registerinhalte:

```
 98725
 17282
 05907
110100
115007
001000
116007
000000
```

Sei W(a) (bzw. W(b)) der Dezimalwert des Inhalts von Register a (bzw. b). Das Verfahren ist korrekt, weil die Invarianzeigenschaft „W(a) + W(b) = Summenwert" unabhängig vom aktuellen Takt gilt. Das Verfahren terminiert, weil pro

Takt mindestens ein Bit (von rechts gesehen) von Register 2 (b) zu Null wird. Man kann zeigen: Sind Nullen und Einsen in a und b gleichverteilt und unabhängig voneinander, so stoppt das Verfahren im Mittel nach $\log_2 n$ Takten. Es benötigt maximal (n+1) Takte.

Bei der **Subtraktion** im 2-Komplement werden alle Bits von b gekippt.
Im zweiten Takt wird $b_0 = 1$ gesetzt.

Beispiel:

Die Binärzahlen a = (11110101) (–11) und b = (11100101) (–27) sind zu subtrahieren (a-b).

Hierzu wird b zunächst invertiert: b´ = (00011010).
Addition nach von-Neumann:

11110101		Takt 1
00011010		
————		
11101111		Takt 2
00100001	c_{-1} = 1, denn Subtraktion	
————	im 2-Komplement	
11001110		Takt 3
01000010		
————		
10001100		Takt 4
10000100	Gesamtübertrag ignorieren	
————		
00001000		Takt 5
00001000		
————		
00000000		Takt 6
00010000		
————		
00010000	Die Summe hat den Wert W(a) = 16.	Takt 7
00000000	Stop, weil Register 2 den Inhalt (0 ... 0) hat.	

Realisierung und Ablaufsteuerung durch ein Mikroprogramm

Wie für die serielle Addition wird nun für die Von-Neumann-Addition ein Mikroprogramm angegeben. Anschließend wird dieses in einer Diodenmatrix „fest verdrahtet". Das Programm addiert (subtrahiert) binäre Zahlen in der 2-Komplement-Darstellung. Um Überläufe zu erkennen, wird das Vorzeichen verdoppelt. Wiederum werden die in eckigen Klammern stehenden Befehle parallel ausgeführt.

Mikroprogramm für die Von-Neumann-Addition:

0: [A = $(a_{(n-1)} \ldots a_0)$:= Summand 1;
B = $(b_{(n-1)} \ldots b_0)$:= Summand 2;
T := 0, falls Addition; T := 1, falls Subtraktion]

1: [$b_i := b_i \oplus T$, für alle i = 0, ... (n–1)]
(b wird durch EXOR mit V komponentenweise gekippt;
Subtraktion im 2-Komplement.)
2: if $B \cup T \neq 0$ then
[$a_i := a_i \oplus b_i$, für i = 0, ... (n–1);
$b_{(i+1)} := a_i \cdot b_i$, für i = 0, ... (n–2);
$b_0 := T$; $T := 0$;
goto 2]
else [$T := a_{(n-1)} \oplus a_{(n-2)}$;
goto 3]
3: if $T \neq 0$ then goto 5
4: Stop ohne Überlaufwarnung
5: Stop mit Überlaufmeldung

Ablaufsteuerung:

Für die Steuerung des Ablaufs eines Programms inhardwarenaher Form sind folgende Schritte umzusetzen:

α) Es muß festgestellt werden, welcher Takt vorliegt.
β) Man muß erkennen, welcher „Teiltakt" vorliegt (und welche Bedingungen aufgrund der Registerzustände gegeben sind).
γ) Es ist die Nummer des folgenden Taktes zu berechnen.
δ) Die Mikrooperationen (Befehle) sind auszuführen.
ε) Die durch die Mikrooperationen beeinflußten Register sind zu takten.

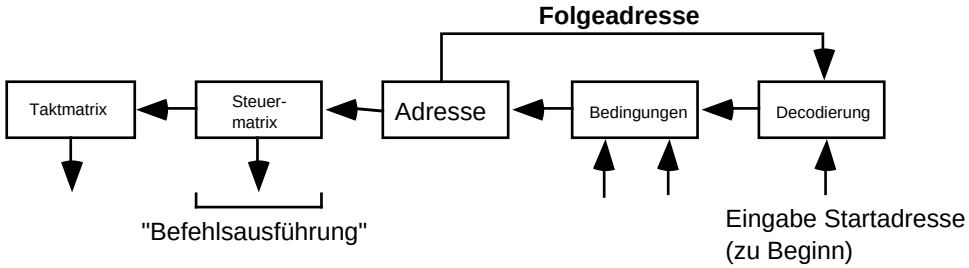


Abbildung 4.8: Ablaufsteuerung

Ablaufsteuerungen werden durch Diodenmatrizen oder **PLA**'s (***Programmable Logic Array***) realisiert. Dies sind Felder sich kreuzender Leitungen. Die Ausgangsleitungen stehen orthogonal zu den Eingangsleitungen. Eingangswerte werden mittels spezieller Dioden auf die Ausgangswerte umgesetzt. Der verknüpfende Kreis zwischen Eingangs- und Ausgangsleitungen bildet ein Oder, das heißt, eine auf der Eingangsleitung anliegende Eins wird auf die gesamte Ausgangsleitung „übertragen". Der Strich hingegen hat den Charakter eines logischen Und: Die Ausgangsleitung hat nur dann den Wert Eins, wenn mindestens alle über den Strich an sie angeschlossenen Eingangsleitungen eine Eins haben.

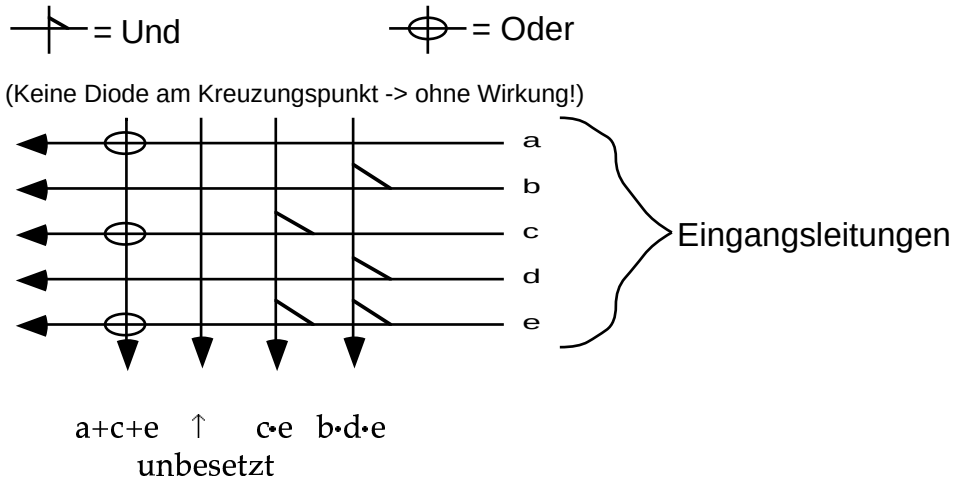


Abbildung 4.9: Dioden

Diesen Schritten sind in einer fest verdrahteten Realisierung eines Mikroprogramms folgende Teilmatrizen zugeordnet:

α) Decodiermatrix,
β) Bedingungsmatrix,
γ) Adreßmatrix für die Codierung der nächsten Programmadresse,
δ) Steuermatrix und
ε) Taktmatrix.

Beispiel:
Als Beispiel wird nun das Mikroprogramm der Von-Neumann-Addition in eine Diodenmatrix umgesetzt. Man trägt Bedingungen im Programm in die Bedingungsmatrix ein und verknüpft sie als Eingänge mit den vertikalen Eingangsleitungen.
Zu den Eingängen zählen auch die Leitungen der Decodiermatrix. Sie liefern die Programmadresse des Mikroprogrammbefehls. Auch sie werden über Und-Dioden mit den vertikalen Leitungen verknüpft. Die Verknüpfungspunkte entsprechen dabei der binär codierten Adresse. Zu den Ausgangsleitungenzählen die Taktmatrix, Steuermatrix und die Adreßmatrix.
Die Leitungen dieser Matrizen werden über Oder-Dioden mit den vertikalen Eingangsleitungen verknüpft. Die Adreßmatrix dient der Bestimmung der folgenden Programmadresse. In der Steuermatrix werden die Mikrooperationen der Mikroprogrammzeile angestoßen. Die Taktmatrix liefert den Takt für die in den angestoßenen Befehlen verwendeten Register. Die Ausgangsleitungen der letzten drei Matrizen sind deswegen durch Oder-Dioden mit den vertikalen Leitungen verknüpft, weil nur eine vertikale Leitung eine Eins zu führen braucht, um einen Mikroprogrammbefehl zu realisieren.

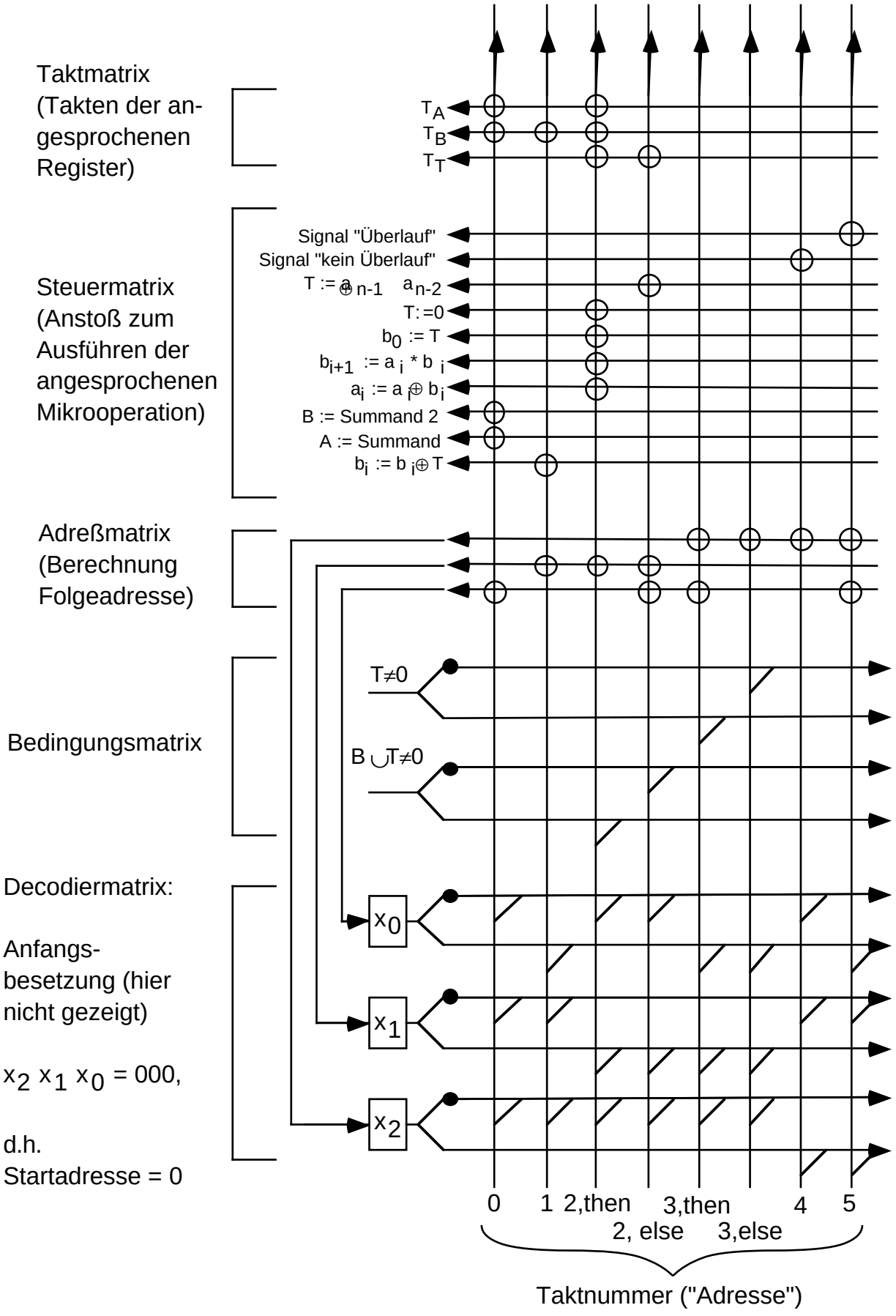


Abbildung 4.10: Diodenmatrix

4.6 Erweiterungen des Von-Neumann-Addierers

In diesem Abschnitt wird die Von-Neumann-Addition derart erweitert, daß man mehrere Summanden addieren kann. Dies wird durch Pipelining oder Parallelität (Adder-Tree) realisiert. Zunächst sei folgende abkürzende Notation eingeführt:

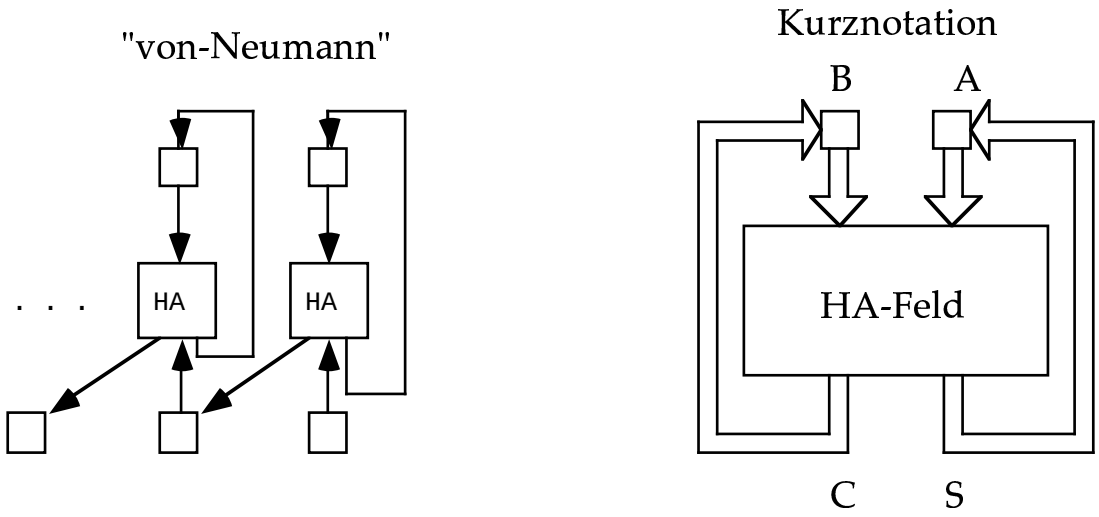


Abbildung 4.11: Kurznotation des Von-Neumann-Addierers

Carry-Save-Addierer

Eine erste Erweiterung besteht darin, imVon-Neumann-Addierer die Halbaddierer gegen Volladdierer auszutauschen. Aufgrund des somit gewonnenen Eingangs (siehe Kurznotation, insgesamt hat man nun 3·n Eingänge und 2·n Ausgänge) ist es möglich, nach jedem Takt einen neuen Summanden in das Addierwerk einzubringen.

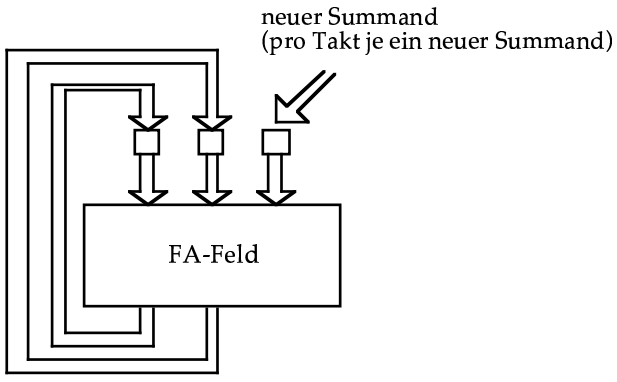
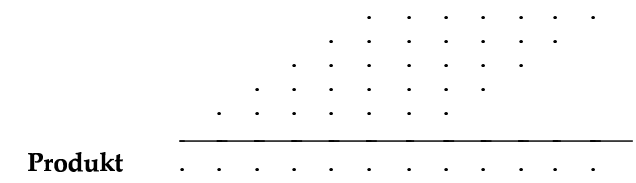


Abbildung 4.12: Carry-Save-Addierer

Das erhaltene Volladdiererfeld wird als **Carry-Save-Addierer** (CSA) bezeichnet (in der Kurznotation werden auch Einfachpfeile verwendet). Die Addition von m Zahlen ist nun in m Takten möglich, wenn man diese wie folgt in den Addierer einbringt:

Takt 1:	Die ersten drei Summanden werden bearbeitet.
Takt 2:	Summanden 1 bis 4 werden bearbeitet.
:	
Takt (m–2):	Summanden 1 bis m werden bearbeitet.
Takt (m–1):	„Auslaufen" der Addition nach Von-Neumann-Addition
Takt m:	Abhängig von der Länge der größten Propagationskette, weiteres „Auslaufen" der Addition; bei Wortlänge n im Mittel ($\log_2$ n) Takte.

Eine **Anwendung** besteht in der konventionellen Multiplikation durch eine Folge von Additionen (Schulmethode), wie im Abschnitt über die Multiplikation näher beschrieben. Skizze:



Adder-Tree (Wallace Tree)

Um einen Adder-Tree zur parallelen (und somit beschleunigten) Addition aufzubauen, bedarf es der Verwendung mehrerer CSAs, die miteinander verschaltet sind. Hierzu wird die Taktung aufgetrennt, es werden soviele CSAs hintereinandergeschaltet, wie es aufgrund der Summandenzahl erforderlich ist.

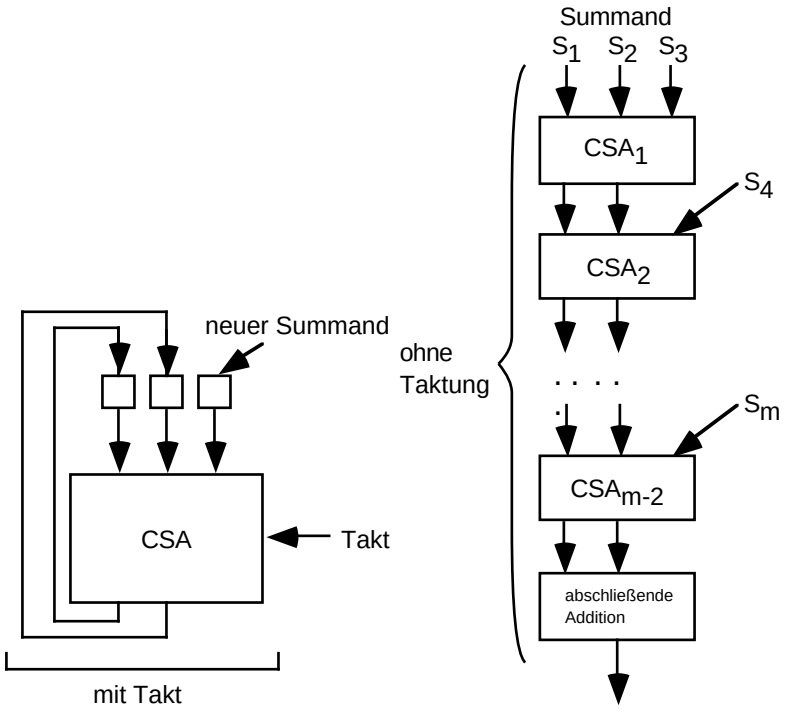


Abbildung 4.13: Lineare Verschaltung von CSAs

Auf diese Weise hat man bezüglich der Laufzeit keinen Gewinn gemacht. Ordnet man die CSAs in einem Baum an, so ist es möglich, die Addition in jeder Schicht des Baums parallel durchzuführen.

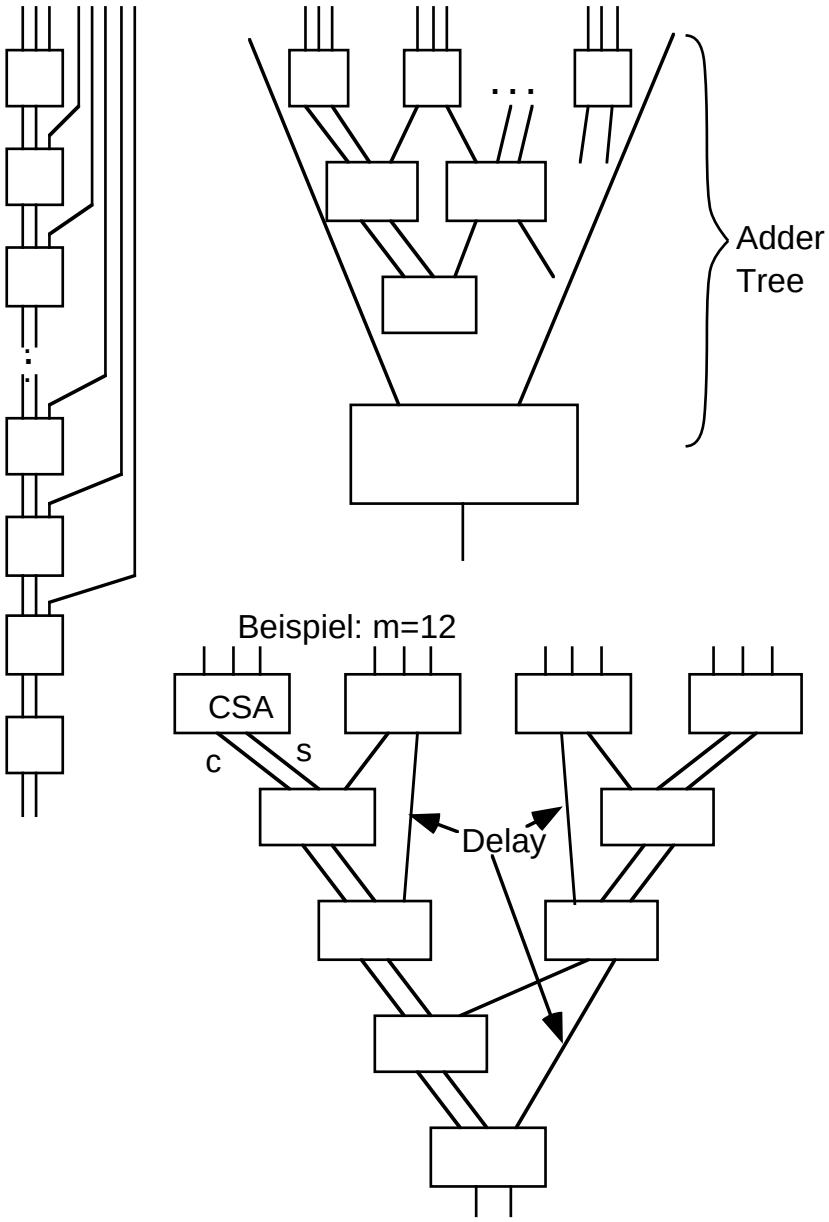


Abbildung 4.14: Adder-Tree (Wallace Tree)

Die **Laufzeit** des Adder-Trees für m Summanden ergibt sich dann wie folgt: Im wesentlichen werden durch jeden CSA 3 Eingänge auf 2 Ausgänge abgebildet, wobei die Anzahl der Eingänge nicht immer ein Vielfaches von 3 ist.
Sei m die Anzahl aller Eingänge (an den Blättern des Baums oben). Dann beträgt die Anzahl der Ausgänge nach der i-ten Stufe ungefähr $(2/3)^i \cdot m$. Für die Laufzeit ist die Stufenzahl x gesucht, so daß nach der x-ten Stufe zwei Ausgänge vorliegen: $(2/3)^x \cdot m = 2$.
Man erhält: $x \cdot \log_2 (2/3) = \log_2 (2/m) \Rightarrow x = \log_2 (m/2) \; / \; \log_2 (3/2) \approx 1{,}72 \cdot \log_2 m$.

4.7 Carry-Look-Ahead-Addition

Die bereits kennengelernte Carry-Ripple-Addition ist langsam, weil sie n Volladdierer in Serie verwendet. Hierbei entsteht ein Übertragsengpaß, weil jeder Volladdierer auf den Übertrag seines rechten Nachbarn warten muß, um sein korrektes Ergebnis bilden und weiterleiten zu können.

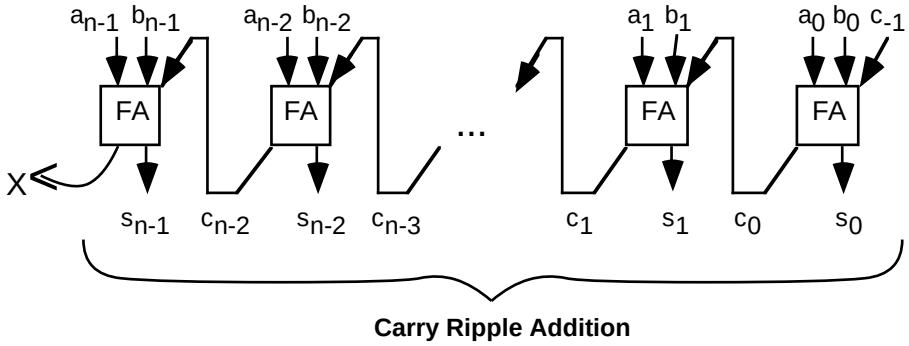


Abbildung 4.15: Carry-Ripple-Addition

Übertragsengpaß: $c_i = a_i \cdot b_i + (a_i + b_i) \cdot c_{(i-1)} = k_i + d_i \cdot c_{(i-1)}$

wobei: $d_i = a_i + b_i$ und $k_i = a_i \cdot b_i$.

Carry-Look-Ahead

Die Rekursionsformel für den Übertrag wird aufgelöst, um die Addition durch direkte Berechnung des Übertrags c_i aus den $(2 \cdot k+1)$ Werten $c_{(i-k)}$, a_j, b_j, $(j = (i-k+1), \ldots i)$ zu beschleunigen:

$$
\begin{aligned}
c_i &= k_i + d_i \cdot c_{(i-1)} \\
&= k_i + d_i \cdot (k_{(i-1)} + d_{(i-1)} \cdot c_{(i-2)}) \\
&= k_i + d_i \cdot k_{(i-1)} + d_i \cdot d_{(i-1)} \cdot c_{(i-2)} \\
&= \ldots = k_i + d_i \cdot k_{(i-1)} + d_i \cdot d_{(i-1)} \cdot k_{(i-2)} + d_i \cdot d_{(i-1)} \cdot d_{(i-2)} \cdot k_{(i-3)} + \ldots \\
&\quad + d_i \cdot \ldots \cdot d_1 \cdot k_0 + d_i \cdot \ldots \cdot d_0 \cdot c_{-1}
\end{aligned}
$$

Für große i ist diese Vorgehensweise zu teuer und unrealisierbar. Daher beschränkt man sich auf machbare Gruppen und verknüpft diese nach dem Prinzip der Carry-Ripple-Addition, wie in Abbildung 4.16 für Gruppengröße 4 gezeigt:

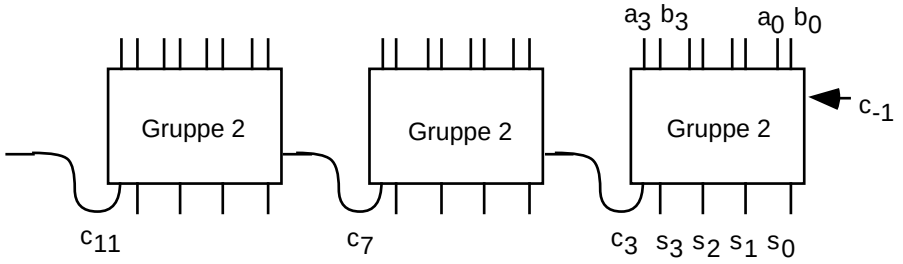


Abbildung 4.16: Carry-Look-Ahead-Addierer für Gruppengröße 4

Ein Carry-Look-Ahead-Addierer arbeitet im Gegensatz zu einem Carry-Ripple-Addierer also nicht mit einzelnen Bits, sondern mit ganzen **Bitgruppen**.

Formelbeispiel:

Der Übertrag c_7 hat die Form:

$$c_7 = k_7 + d_7 \cdot k_6 + d_7 \cdot d_6 \cdot k_5 + d_7 \cdot d_6 \cdot d_5 \cdot k_4 + d_7 \cdot d_6 \cdot d_5 \cdot d_4 \cdot c_3$$

Die Laufzeit wird jeweils im Worst-Case von $2 \cdot n$ bei Carry-Ripple-Addition auf $2 \cdot (n/g)$ bei der Carry-Look-Ahead-Addition herabgesetzt, wobei g die Gruppengröße und (n/g) die Gruppenanzahl ist. n bezeichnet wie oben die Stelligkeit der zu addierenden Zahlen und somit die Anzahl der Eingänge des Addierers. Sie wird um den Faktor der Gruppengröße verkürzt.

Die erzielte Beschleunigung wird bezahlt durch:

- höhere Kosten innerhalb der Gruppen und
- zusätzliche Vorbereitungslaufzeiten, um k_i und d_i bereitzustellen, weil diese sofort benötigt werden. Diese Vorbereitungslaufzeiten werden allerdings erst dann merkbar, wenn man das Prinzip auf die „2. Ordnung" erweitert, das heißt, zur Bildung von „Supergruppen" übergeht, indem man Carry-Look-Ahead-Addierer hintereinanderschaltet.

Beispiel: Carry-Look-Ahead

Die Fahrplanorganisation der SNCF (Paris Banlieue Ouest) ist ein anschauliches Beispiel für die Funktionsweise der Carry-Look-Ahead-Addition. Man betrachte die kursivgedruckten Ausschnitte in Abbildung 4.17.
Interpretiert man jeden Halt als Weitergabe eines Übertrags, so wird bei 5105 der Übertrag über die erste Gruppe — dargestellt durch den Zug 1538 — hinweggeleitet. Der Übertrag in Spalte 5833 überspringt die ersten zwei Gruppen bis KM 10.

	NUMERO	1538	5105	5833	2183	5107	1769	5471	1589	5109	5837
	PARTICULARITES										
	CIRCULATION										
KM											
0	PARIS ST LAZARE	*13 15*	*13 20*	*13 26*	13 29	13 35	13 35	13 41	*13 45*	*13 50*	*13 56*
2	PONT CARDINET	*13 17*					13 37		*13 47*		
4	CLICHY-LEVALLOIS	*13 20*					13 40		*13 50*		
5	ASNIERES	*13 22*					13 42		*13 52*		
6	BECON LES BRUYERES	*13 23*	*13 26*		13 34	13 41	13 43		*13 53*	*13 56*	
7	LES VALLEES		*13 28*			13 43				*13 58*	
8	LA GARENNE-BEZONS		*13 31*			13 46				*14 01*	
10	NANTERRE-UNIVERSITE		*13 33*	*13 34*		13 48		13 50		*14 03*	*14 05*
13	HOUILLES-CARRIERES/S			*13 38*				13 53			*14 08*
16	SARTROUVILLE			*13 41*				13 56			*14 11*
17	MAISONS LAFFITTE			*13 43*				13 58			*14 13*
24	ACHERES VILLE			*13 49*							*14 19*
33	CERGY PREFECTURE			*13 56*							*14 26*

Abbildung 4.17: Carry-Look-Ahead im SNCF-Fahrplan

4.8 Carry-Skip-Addition

Die Carry-Skip-Addition ist eine Verbesserung der Addition nach dem Carry-Look-Ahead-Prinzip. Die aufwendige Schaltlogik der Carry-Look-Ahead-Addition ist nicht in jeder der (n/g) Gruppen nötig. Hierbei bezeichnet n die Anzahl eingehender Bits, also die Stelligkeit der zu addierenden Zahlen und g die Anzahl der Eingänge pro Gruppe. Man berechnet die Überträge einer Gruppe wie bei Carry-Ripple in maximal 2·g Gatterstufen und leitet die Überträge beschleunigt durch einen einfachen Zusatzschaltkreis über die Gruppen hinweg.

Carry-Skip für eine konstante Gruppengröße g

Sei j der Gruppenindex, wobei j Werte zwischen 0 und (n/g)–1 annehmen kann. Ferner seien folgende Abkürzungen eingeführt:

- $D_j := d_{(j+1)g-1} \cdot \ldots \cdot d_{jg}$ und
- $K_j := k_{(j+1)g-1} \cup d_{(j+1)g-1} \cdot k_{(j+1)g-2} \cup \ldots \cup d_{(j+1)g-1} \cdot \ldots \cdot d_{jg+1} \cdot k_{jg}$

Hierbei sind d_i und k_i wie im letzten Abschnitt (Carry-Look-Ahead) definiert:

$$d_i = a_i \cup b_i \text{ und } k_i = a_i \cdot b_i$$

Ist $K_j = 1$, so ist in Gruppe j ein Übertrag entstanden. Es gilt für den Gesamtübertag C_j der Gruppe j:

$$C_j = K_j \cup D_j \cdot C_{(j-1)}$$

Bei Carry-Skip wird ein Übertrag der Gruppe j ($C_j = 1$) wie folgt berechnet, wobei zwei Fälle unterschieden werden:

a) $K_j = 1$ und $D_j \cdot C_{(j-1)} = 0$ (Übertrag entsteht erst in Gruppe j):
Dann ist $C_j = K_j = 1$ und wird wie bei Carry-Ripple in maximal 2·g Gatterstufen berechnet.

b) $D_j \cdot C_{(j-1)} = 1$ (Übertrag wird weitergeleitet):
Hier liegt der für die Gesamtlaufzeit ungünstigste Fall vor. Der Übertrag wird mit D_j in zwei zusätzlichen Gatterstufen weitergeleitet.

Abbildung 4.18 zeigt die Schaltung einer Gruppe des Carry-Skip-Addierers.

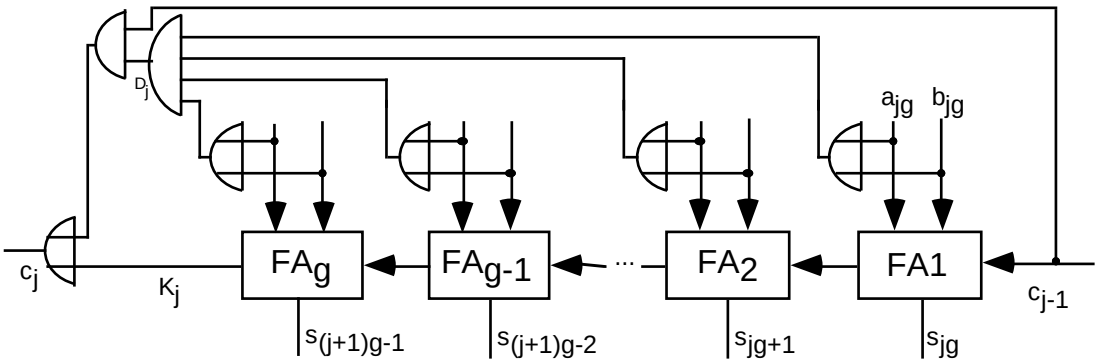


Abbildung 4.18: Gruppe eines Carry-Skip-Addierers

Betrachten wir nun die **Laufzeit und Kosten** des Addierers:

Der Worst-Case der Laufzeit liegt dann vor, wenn:

$$a_0 = b_0 = 1,\ c_{-1} = 0 \text{ und für alle } i = 1, \ldots (n\text{-}2) \text{ gilt: } a_i \oplus b_i = 1.$$

Das bedeutet, der Übertrag entsteht nur in Gruppe 0, wird nicht sofort weitergeleitet — weil $D_0 \cdot c_{-1} = 0$ — und läuft maximal lange.
In diesem Fall ist die Laufzeit:

τ	=	$2 \cdot g + 1$	+	$2 \cdot (n_1 - 2)$	+	$2 \cdot g$
		Berechnung von K_0, C_0		Übertrags-weiterleitung über (n_1-2) innere Gruppen mit $n_1 = (n/g)$		Berechnung von $s_{(n-1)}$

τ_{min} entsteht als Minimum (n_1 Einsetzen, nach g Ableiten, gleich Null setzen, ...) für:

$$g = g_{opt} = \backslash R(;\backslash F(n;2))$$

Man erhält:

$$\tau_{min} = 4 \cdot \sqrt{;2} \cdot \sqrt{;n} - 3.$$

Die Kosten lassen sich aus der Schaltung ablesen (vergleiche Abbildung 4.18). Zusätzlich zu den im Volladdierer enthaltenen OR-Gatternbenötigt man pro Gruppe ein AND-Gatter mit g Eingängen für D_j und zwei Gatter (OR, AND) mit jeweils zwei Eingängenfür die Ermittlung von C_j.
Die Kosten $k_{Carry\text{-}Skip}$ ergeben sich somit aus den Kosten $k_{Carry\text{-}Ripple}$ und den Zusatzkosten derGatter:

$k_{Carry\text{-}Skip} = k_{Carry\text{-}Ripple} + \backslash F(n;g) \cdot (g + 2 \cdot 2) = 14 \cdot n + n + \backslash F(4 \cdot n;g) = 15 \cdot n + \backslash F(4 \cdot n;g)$.

Carry-Skip mit variabler Gruppengröße

Aus dem letzten Abschnitt wird ersichtlich, daß folgender ungünstiger Fall auftreten kann: Der Übertrag entsteht in der Gruppe ganz rechts (in Abb. 4.18), läuft schnell über die inneren Gruppen und langsam in der letzten Gruppe ganz links.
Eine Idee zur Behebung der Laufzeitunterschiede zwischen den an den „Rändern" und in der Mitte liegenden Gruppen besteht nun darin, kleinere Gruppen an den beiden „Rändern" und höhere Gruppengrößen zur Mitte hin zu wählen. Dabei ist zu beachten, daß der Größenunterschied von Gruppe zu Gruppe nicht beliebig sein darf: Falls die Gruppengröße um mehr als 1 zwischen zwei benachbarten Gruppen differiert, entsteht ein anders gearteter „ungünstigster Fall".

Beispiel-Skizze:

Sei [i] eine Gruppe der Größe i. Unter den Kasten wird die Laufzeit innerhalb der Gruppe notiert. Die Gesamtlaufzeit wird für den Worst-Case angenommen.

Fall 1: Drei gleich große Gruppen

7	7	7	
14	2	15	Gesamtlaufzeit: 31

Fall 2: Drei Gruppen abnehmender Größe

8	7	6	
16	2	13	Gesamtlaufzeit: 31
16	15		Gesamtlaufzeit: 31

Fall 3: Abweichung der Gruppengröße um mehr als 1

9	7	5	
18	2	11	Gesamtlaufzeit: 31
18	15		Gesamtlaufzeit: 33

Somit liegt hier ein neuer Worst-Case vor, wenn man in der zweiten Gruppe beginnt.

Daher darf nur eine Steigerung der Gruppengröße von beiden Rändern nach innen pro Gruppe um jeweils den Wert 1 stattfinden. Dies ist aber nicht für alle Zahlenlängen (Längen der zu addierenden Zahlen) perfekt möglich: Die Summe der Gruppengrößen muß n ergeben, und gleichzeitig muß der Addierer laufzeitoptimal sein. Daher müssen am Rand gegebenenfalls einzelne Kleingruppen entfernt werden. Einige Beispiele zeigen auch, daß unter Umständen eine Gruppe aus der Mitte wegzunehmen ist.

Beispiele:

Gruppen werden mit ihrer Größe in einem Tupel notiert.

1) Sei n = 56:
   Hier liegt einer der wenigen optimalen Fälle vor: Gruppengrößen:
   (1, 2, 3, 4, 5, 6, 7, 7, 6, 5, 4, 3, 2, 1), Laufzeit: maximal 29 Gatterstufen.

2) Sei n = 50:
   In diesem Fall ist die Aufteilung nicht so günstig möglich. Man kann von der Situation im ersten Beispiel ausgehend die Gruppen an den Rändern entfernen:
   (3, 4, 5, 6, 7, 7, 6, 5, 4, 3)
   oder die drei Gruppen ganz links streichen:
   (4, 5, 6, 7, 7, 6, 5, 4, 3, 2, 1)
   oder die rechten drei Gruppen wegnehmen:
   (1, 2, 3, 4, 5, 6, 7, 7, 6, 5, 4).
   In allen drei Fällen hat man die gleiche Laufzeit von 29 Gatterstufen wie bei n = 56.

3) Sei n = 44:

Hier liegt ein Fall vor, bei dem man ausgehend von
(1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1) zum einen die Gruppen an den Rändern streichen und die mittlere 7er- gegen eine 6er-Gruppe tauschen sollte:
(2, 3, 4, 5, 6, 6, 6, 5, 4, 3)
oder an einem Rand Gruppen streicht und in der Mitte zwei Gruppen verkleinert:
(3, 4, 5, 6, 6, 5, 5, 4, 3, 2, 1).
Die Laufzeit beträgt 27 Gatterstufen wie für n = 49.

Die Laufzeit τ_n eines n-Bit-Carry-Skip-Addierers mit variabler Gruppengröße ist maximal:

$$\tau_n = \begin{cases} 2\cdot\lceil\sqrt{4\cdot n+1}\rceil - 1; & \text{wenn } m^2 < n \le m\cdot(m+1) \\ 2\cdot\lceil\sqrt{4\cdot n}\rceil - 1; & \text{wenn } (m-1)\cdot m < n \le m^2 \end{cases}$$

Insgesamt gilt also: $\tau_n \approx 4\cdot\sqrt{n}$.

Im Vergleich dazu war die minimale Laufzeit bei konstanter Gruppengröße:

$$\tau^{konst.}_{min} = 4 \cdot \sqrt{2} \cdot \sqrt{n} - 3.$$

Die Kosten sind in unserem Kostenmaß (siehe Kapitel 3) fast gleich.

Verallgemeinerungen und Kompromisse

Man kann Gruppen von Carry-Skip-Addierern hintereinanderschalten und erhält eine sogenannte Supergruppe. Aus mehreren solcher Supergruppen läßt sich dann ein Carry-Skip-Addierer zweiter Ordnung zusammensetzen.

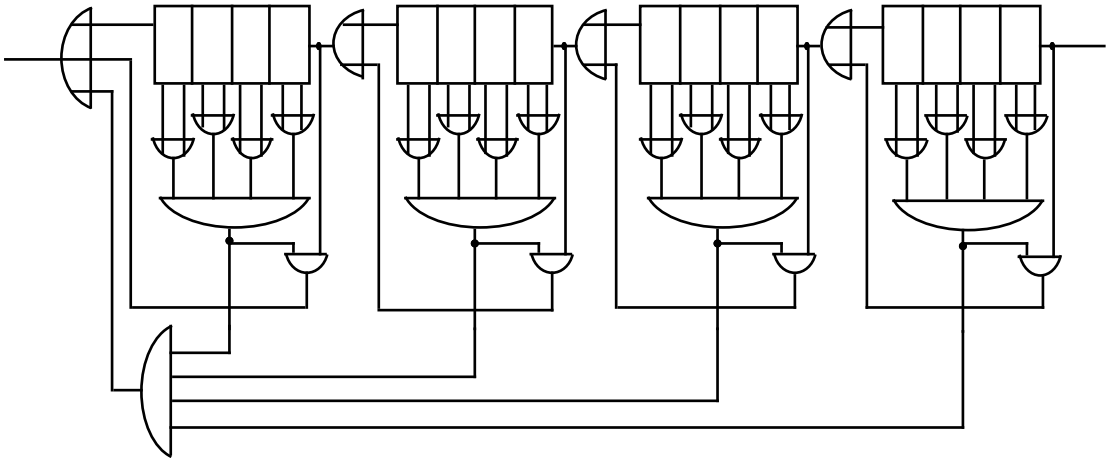


Abbildung 4.19: Schaltung zum Carry-Skip-Addierer zweiter Ordnung

Eine Ordnungserhöhung lohnt sich aber nur für sehr große Längen der zu addierenden Zahlen. Sonst fällt der Gewinn geringer aus als der Aufwand für die zusätzliche Logik, die zur Koordination nötig ist.

Ferner läßt sich auch die Technik des Carry-Skip-Addierers mit der eines Carry-Look-Ahead-Addierers kombinieren (siehe Abbildung 4.20). Die Anzahl der zusätzlichen Gatter ist hier jedoch größer.

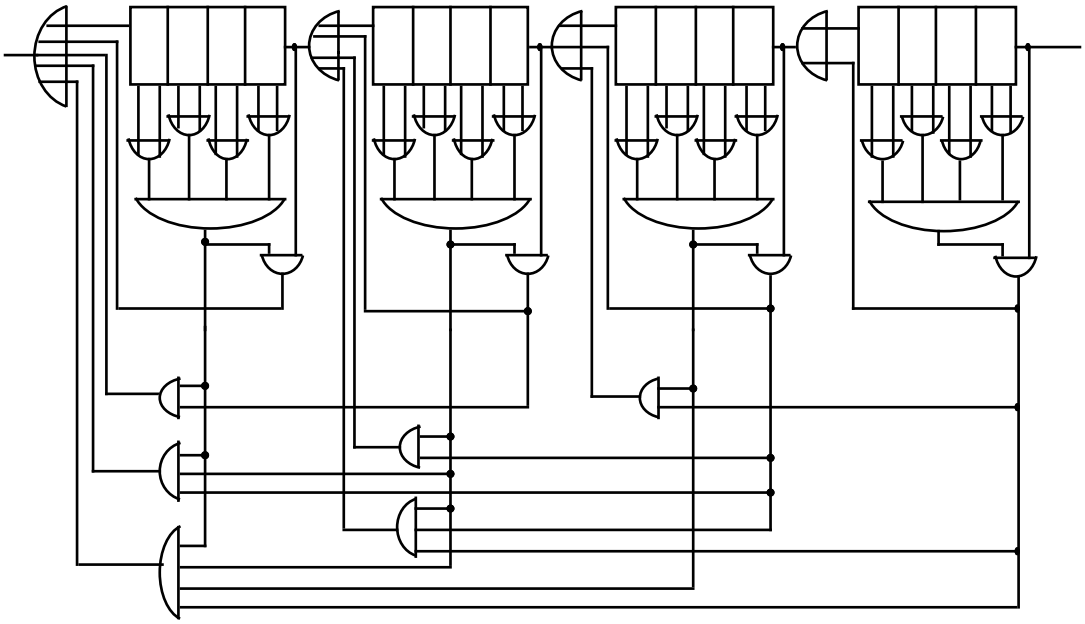


Abbildung 4.20: Kombination von Carry-Skip und Carry-Look-Ahead

4.9 Conditional-Sum-Addition

Wie bei der Carry-Look-Ahead-Addition werden hier ganze Gruppen bearbeitet, allerdings in verschiedenen Gruppengrößen, nämlich zuerst in 1er, dann in 2er, 4er, ... Gruppen. Es findet also immer eine Verdopplung der Gruppengröße statt, die durch die Zusammenfassung zweier benachbarter Gruppen erfolgt, wie man Abbildung 4.21 entnehmen kann. Diese Zusammenfassung kann dann parallel realisiert werden.

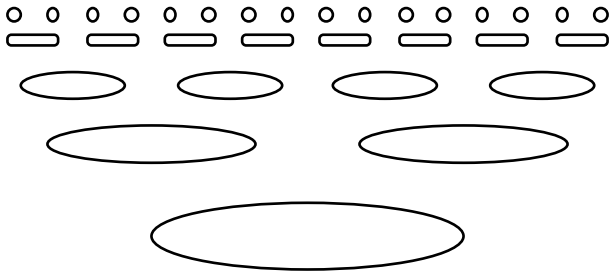


Abbildung 4.21: Gruppen bei der Conditional-Sum-Addition

Die Conditional-Sum-Addition zeichnet sich dadurch aus, daß sie bei der Zusammenfassung zweier Gruppen die Entstehung eines Übertrags berücksichtigt.

Für jede Gruppe gibt es zwei Möglichkeiten:

a) es kommt kein Übertrag von rechts oder

b) es kommt ein Übertrag von rechts.

Zusammenfassung zweier Gruppen:

Die Zusammenfassung zweier Gruppen geht sehr schematisch. Man stellt sich eine aus vier Feldern aufgebaute Gruppe vor (siehe Abbildung 4.22). Die linken Felder zeigen die Übertragssituation an und die rechten den davon abhängenden Wert. Informell lautet die Regel zur Zusammenfassung zweier Gruppen zu einer neuen:

- Für die rechte Hälfte (der neuen):
  Übernimm die „alten" Alternativen,
- Für die linke Hälfte:
  Übernimm Alternative „kein Übertrag", falls die rechte Hälfte dies anzeigt,
  übernimm Alternative „Übertrag", falls die rechte Hälfte dies anzeigt.

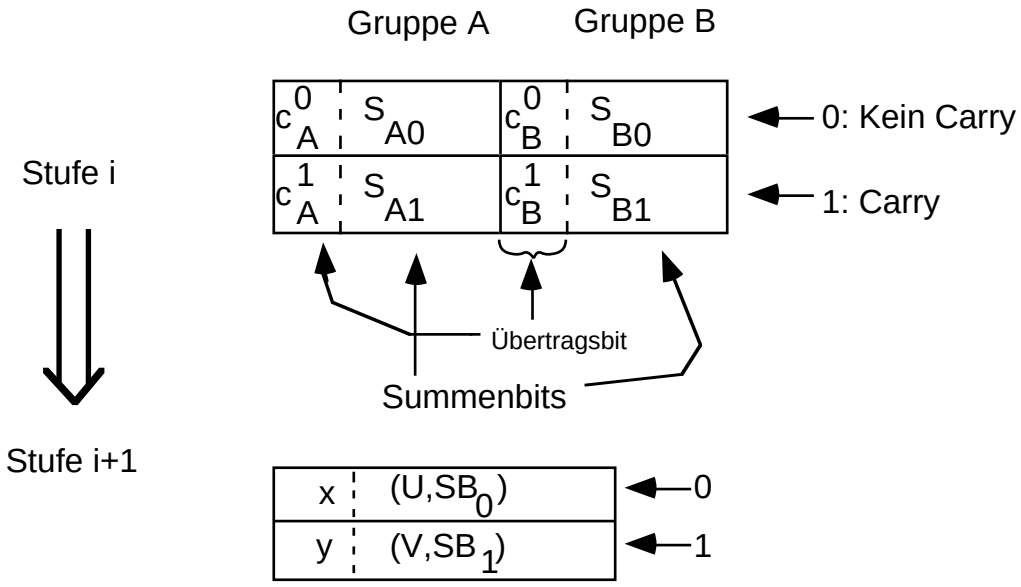


Abbildung 4.22: Zusammenfassung zweier Gruppen

Die formale Regel für die Zusammenfassung zweier Gruppen lautet:

- Für x:
  if $c_B;^0 = 0$ then $x := c_A;^0$
  else $x := c_A;^1$
- Für y:
  if $c_B;^1 = 1$ then $y := c_A;^1$
  else $y := c_A;^0$
- Für U:
  if $c_B;^0 = 0$ then $U := S_{A0}$
  else $U := S_{A1}$

- Für V:

  if $c_B;^1 = 1$ then $V := S_{A1}$
  else $V := S_{A0}$

Die Angabe einer Schaltlogik für diese Zusammenfassung ist einfach:
In der Stufe 0 ist jede Gruppe ein Bitpaar (a_i, b_i). Die Gruppengröße ist 1.
Die Logik für die Stufe 0 lautet:

$x = a_i \cdot b_i$,
$y = a_i + b_i$,
$S_{A0} = \overline{};S_{A1}$ und
$S_{A1} = a_i \oplus b_i \oplus 1 = a_i \cdot b_i + \overline{};a_i \cdot \overline{};b_i = x + \overline{};y$

Die Logik höherer Stufen bedeutet die Umsetzung des if-Konstruktes durch eine Torschaltung:

$x :=$ if $\alpha = 0$ then β else γ wird durch die Schaltung:
$x := \overline{};\alpha \cdot \beta + \alpha \cdot \gamma$ realisiert.

Die Kosten dieser Schaltung nach der Definition aus Kapitel 3 betragen 6.

Beispiel:
Die Zahlen a = (001101000110) und b = (110101011001) sind zu addieren bzw. zu subtrahieren, was durch c_{in} bzw. $c_{in} = 1$ ausgedrückt wird.

Stufe	a b	0 1		0 1		1 0		1 1		0 0		1 1		0 0		0 1		0 1		1 0		1 0		0 1		a b
0	0 1	1 0	0 1	1 0	0 1	1 0	1 1	0 1	0 0	0 1	1 1	0 1	0 0	0 1	0 1	1 0	0 1	1 0	0 1	1 0	0 1	1 0	0 1	1 0	0 1	$c_{in}=0$ $c_{in}=1$
1	0 1		1 0	1 0	1 1		0 0	0 1	0 0		1 1	0 1	0 0		0 1	1 0	0 1		1 0	1 0	0 1		1 0	1 0	0 1	0 1
2	1 1				0 0	0 0	0 0	0 1	0 0				1 1	0 0	0 1	1 0	0 1				1 0	1 0	1 0	1 0	0 1
3	1 1				0 0	0 0	0 0	0 1	0 0								1 1	0 0	0 1	1 0	1 0	1 0	1 0	1 0	0 1
4	1 1													0 0	0 0	0 0	0 0	1 1	0 0	0 1	1 0	1 0	1 0	1 0	1 0	0 1

Abbildung 4.23: Beispiel zur Conditional-Sum-Addition

Laufzeitbetrachtung:

Dieser Additionsalgorithmus ist fast optimal, denn es ist beweisbar, daß ($\log_2 n$) eine untere Schranke für die Laufzeit der Addition zweier n-stelliger Zahlen ist.

a) Man benötigt ($\lceil \log_2 n \rceil + 2$) Stufen für die Addition, wobei die letzte Stufe für die Auswahl des Endergebnisses nötig ist, das heißt, der Summe oder Differenz je nach c_{in}.

b) Man braucht $2 \cdot (\lceil \log_2 n \rceil + 2)$ Gatterstufen, denn es sind für jede Stufe je ein AND- und ein OR-Gatter mit jeweils zwei Eingängen nötig.

Neben der günstigen Laufzeit besteht ein weiterer Vorteil darin, daß gleichartige und billige Gatter verwendet werden (AND_2, OR_2). Außerdem ist ein symmetrischer und synchroner Ablauf möglich.

4.10 Multiplizierwerke

In diesem Abschnitt werden Verfahren der Multiplikation zweier Binärzahlen eingeführt. Zunächst wird die Multiplikation nach der Schulmethode betrachtet, anschließend werden beschleunigte Methoden vorgestellt.
Die Darstellung der Binärzahlen erfolgt in der „Betrag und Vorzeichen"-Codierung. Sie eignet sich gut, weil die Multiplikation auf die Addition zurückgeführt wird. Das Vorzeichenbit der Produktbinärzahl ergibt sich dadurch, daß man das exklusive Oder auf die Vorzeichenbits des Multiplikanden und Multiplikators anwendet.
Folgende Bezeichnungen werden vereinbart:

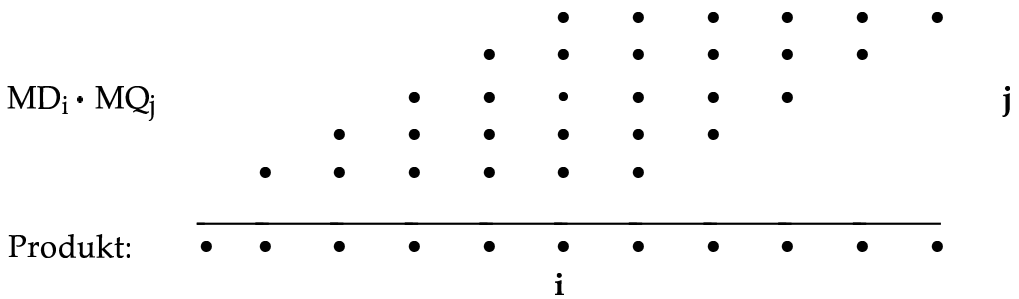
Faktor 1: Multiplikand $MD = (MD_{(n-1)} \ldots MD_0)$ und
Faktor 2: Multiplikator $MQ = (MQ_{(n-1)} \ldots MQ_0)$.

Beide Faktoren müssen **vorzeichenlos** sein. Um vorzeichenbehaftete Werte zu multiplizieren oder zu dividieren, geht man daher in drei Schritten vor:

1. Abfrage der Vorzeichen und Werte vorzeichenlos machen.
2. Multiplikation (Division) mit den vorzeichenlosen Werten.
3. Aus den bei Schritt 1 ermittelten Vorzeichen das Vorzeichen des Ergebnisses errechnen (EXOR).

Multiplikation als Folge von Additionen (Serielle Multiplikation)

Eine erste Idee besteht darin, die Multiplikation auf eine Folge von Additionen zurückzuführen. Dazu wird die **Multiplikationsmatrix** gebildet:



Die Zeilen werden addiert.

Beispiel:
Seien MD = 0111 und MQ = 1011.

Man schreibt in die j-te Zeile der Matrix MD — um j Bits nach links verschoben (Linksshift) —, wenn $MQ_j = 1$ ist. Sonst wird die Zeile mit entsprechend vielen Nullen besetzt. Die Zeilen werden komponentenweise addiert (Schulmethode).

```
              0  1  1  1
           0  1  1  1
        0  0  0  0
     0  1  1  1
-------------------------
  0  1  0  0  1  1  0  1
```

Realisierung

Für das doppelt so lange Produkt brauchen wir zwei miteinander verbundene Register der Länge n. Die einzelnen Additionen sind jeweils um ein Bit gegeneinander verschoben.
Addiert wird:

- Eine 0, falls das betreffende Multiplikatorbit (MQ_j) 0 ist.
- MD, wenn das betreffende Multiplikatorbit 1 ist.

Insgesamt also $MD \cdot MQ_i$, wobei i die Nummer des aktuellen Multiplikatorbits ist.

Um die oben skizzierte relative Verschiebung zu realisieren, kann man entweder das Zwischenergebnis festhalten und $MD \cdot MQ_i$ um ein Bit nach links verschieben, oder man hält $MD \cdot MQ_i$ fest und verschiebt das Zwischenergebnis um ein Bit nach rechts.
Wir wählen die zweite Möglichkeit und verwenden MQ als zweite Hälfte des Produktregisters. Dadurch wird der Multiplikator im Verlaufe der n Multiplikationsschritte überschrieben. Ferner steht das aktuelle Multiplikationsbit immer in MQ_0.

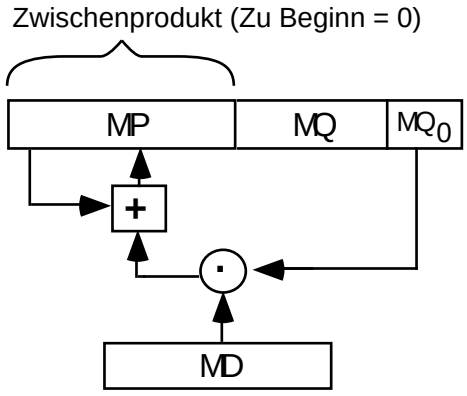


Abbildung 4.24: Schema zur sequentiellen Multiplikation

Das Mikroprogramm für die sequentielle Multiplikation lautet:

0: [MD := Multiplikand; MQ := Multiplikator; MP := 0; Z := n]
1: [Z := Z–1; if MQ_0=1 then MP := MP + MD]
2: [SHR(MP, MQ); if Z > 0 then goto 1]
3: Stop

Bemerkungen:
Die Mikrooperationen einer Programmzeile werden parallel ausgeführt. MP bezeichnet das Partialprodukt.
Die Operation SHR(MP, MQ) veranlaßt einen Rechtsshift des gekoppelten Registers (MP, MQ) um 1 Bit, links werden Nullen aufgefüllt (bei „Betrag & Vorzeichen"-Darstellung), rechts wird rausgeschoben. (Wenn nicht „Betrag & Vorzeichen", sondern 2-Komplement verwendet wird, dann wird links das Vorzeichen nachgefüllt. Die 1-Komplement-Codierung ist wegen End-Around-Carry ungeeignet.)
Das Mikroprogramm läuft n Zyklen lang.

Beispiel:
Seien wiederum MD = 0111 und MQ = 1011.
Am Anfang ist MP = 0000.
1. Durchlauf:
MP | MQ: 0000|1011
MD: 0111
2. Durchlauf:
MP :=
MP+MD: 0111|1011
Shift: 0011|1101
0111
3. Durchlauf:
MP :=
MP+MD: 1010|1101
Shift: 0101|011**0**
0111
4. Durchlauf:
Nur Shift: 0010|1011
0111
5. Durchlauf:
MP :=
MP+MD: 1001|1011
Shift: 0100|1101 Ergebnis.

0111

Beschleunigungsmöglichkeiten

Multiplikatorcodierung

Die sequentielle Multiplikation kann dadurch beschleunigt werden — wie im Mikroprogramm bereits getan —, daß man die Addition von „Nullzeilen" nicht ausführt.
Ein erweiterter Ansatz besteht darin, daß ein Shift über Blöcke von Nullen und Einsen des Multiplikators durchgeführt wird.
Hat der Multiplikator die Form MQ = ... 0^k1^r0 ... (..., k Nullen, dann r Einsen, 0, ...), so kann man unter Zuhilfenahme einer aufwendigen Zusatzlogik k Zyklen zusammenfassen.

Beispiel:

Sei MQ = ... 011110 ..., wobei die rechte Eins die Nummer u im Bitwort trägt.
Eine herkömmliche Berechnung hat die Form:

```
0   1   1   1   1 0...
    |   |   |   ↓ = 1·MD
    |   |   ↓ = 2·MD
    |   ↓ = 4·MD
    ↓ = 8·MD
```

Insgesamt wird also 15·MD addiert. Dies ist darstellbar als Addition von 16·MD und anschließender Subtraktion von 1·MD. Statt der vier Additionen wird man günstigerweise eine Subtraktion (–1) und eine Addition (+16) durchführen.

Man kann also wie folgt vorgehen:
MQ habe die Form MQ = ... 01^k0 ... . Trifft man auf die linke 0, so addiert man 2^k·MD auf das entsprechende Partialprodukt; die k Einsen überläuft man; tritt die Situation auf, daß man — auf einer 1 stehend — als nächstes Bit eine 0 erhält, so subtrahiert man MD.
Hier erkennt man das Problem, daß man wissen muß, wie lang der Einser-Block ist. Dies zu erkennen, ist im allgemeinen schwierig.
Eine einfache Variante ist das Verfahren von **Booth**. Hierbei handelt es sich um ein serielles Prüfen des Multiplikators, wobei man sich das vorherige Bit MQ_{-1} merkt. Die Aktionen sind in folgender Tabelle zusammengefaßt:

MQ_0	MQ_{-1}	Wirkung
0	0	No Operation (Shift über Nullen)
0	1	+ 1·MD (Shift)
1	0	– 1·MD (Shift)
1	1	No Operation (Shift über Einsen)

Bemerkung:
Das Verfahren ist nur dann günstig, wenn in MQ längere Blöcke von Nullen oder Einsen auftreten. Es ist ungünstig, wenn Nullen und Einsen häufig wechseln.

Das eben beschriebene Verfahren bezeichnet man auch als **Multiplikatorcodierung der Gruppengröße 1**.

Betrachten wir nun den allgemeinen Fall der Gruppengröße h:

$MQ_{(h-1)}$	...	MQ_0

Für die Multiplikatorcodierung muß das Korrekturbit MQ_{-1} betrachtet werden:

$MQ_{(h-1)}$	...	$MQ_0 \mid MQ_{-1}$

Falls $MQ_{(h-1)} = 1$, addiert man

$$x = 2^h - 2^{(h-1)} \cdot MQ_{(h-1)} + 2^{(h-2)} \cdot MQ_{(h-2)} + ... + MQ_0,$$

wobei 2^h im nächsten Zyklus korrigiert wird:

$$x' = -2^{(h-1)} \cdot MQ_{(h-1)} + 2^{(h-2)} \cdot MQ_{(h-2)} + ... + 2 \cdot MQ_1 + MQ_0 + MQ_{-1}$$

Aufgrund des Shifts über die Gruppe wird durch MQ_{-1} die Korrektur vorgenommen.
Das Verfahren von Booth arbeitet mit einer Gruppengröße von 1. Somit liegt hier x' zwischen –1 und 1: $-1 \le x' \le 1$.

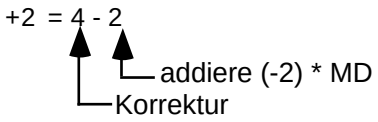
Multiplikatorcodierung mit Gruppengröße 2

Hier faßt man zwei Multiplikatorbits (MQ_1, MQ_0) zusammen.
Es gibt nun vier Möglichkeiten:

Wenn (MQ_1, MQ_0) = (0, 0), addiere 0 · MD,
= (0, 1), addiere 1 · MD,
= (1, 0), addiere 2 · MD und
= (1, 1), addiere 3 · MD.
und schiebe anschließend um 2 Bits nach rechts.

Somit wird die Zahl der Additionen halbiert. Im Falle, daß 2·MD addiert werden muß, führt man einen Linksshift um ein Bit und eine anschließende Addition von MD durch.
Der kritische Fall ist der letzte, wenn (MQ_1, MQ_0) = (1, 1) ist. Dann muß das Dreifache des Multiplikators addiert werden. Dieses Vielfache benötigt eine eigene Addition (2+1) zu seiner Berechnung. Die Lösung besteht nun darin, 3 als $4-1 = (+1) \cdot 2^2 + (-1) \cdot 2^0$ zu interpretieren. Die Wertigkeit von 2^2 ist im nächsten Zyklus — nach zweifachem Rechtsshift des Partialprodukts — 2^0. Man wird im

Fall $(MQ_1, MQ_0) = (1, 1)$ im Zyklus i fälschlicherweise $(-1)\cdot MD$ addieren. Dies wird im Zyklus (i+1) korrigiert, indem dann $4\cdot MD$ — zusätzlich zu der dort anliegenden Addition — addiert wird. Aufgrund des Rechtssshifts über die Gruppenlänge (2 Bits) ist dann aber nur $MQ_{-1}\cdot MD$ zu addieren (MQ_{-1} war im Zyklus i: MQ_1; +4 entspricht im nächsten Zyklus, der eine um den Faktor $2^2 = 4$ höhere Wertigkeit hat, genau dem Wert +1).
Dieses Verfahren läßt sich auch anwenden, falls $(MQ_1, MQ_0) = (1, 0)$; hier wird im Zyklus i $(-2)\cdot MD$ addiert und im Zyklus (i+1) durch $MQ_{-1}\cdot MD$ korrigiert:



Insgesamt kann man immer dann so vorgehen, wenn $MQ_1 = 1$ und somit im darauffolgenden Zyklus $MQ_{-1} = 1$ ist. Dies hat den Vorteil, daß nur ein Bit zu prüfen ist, statt wie im Fall $(MQ_1, MQ_0) = (1, 1)$ beide.
Die folgende Tabelle zeigt die Aktionen, die abhängig von

$$\backslash X(MQ_1, MQ_0 \mid MQ_{-1})$$

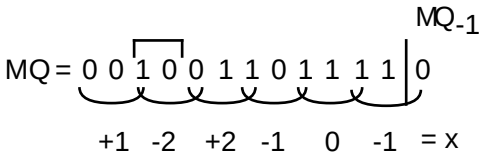
zu vollziehen sind.

MQ_1	MQ_0	MQ_{-1}	Operation: Addiere x·MD mit **x** = ...
0	0	0	**0**
0	1	0	**1**
1	0	0	**–2** (statt 2); •Korrektur: 4
1	1	0	**–1** (statt 3); •Korrektur: 4
0	0	1	**1** = 0 + Korrektur: 1 (4)
0	1	1	**2** = 1 + Korrektur: 1 (4)
1	0	1	**–1** = –2 (statt 2) + Korrektur: 1 (4); •Korrektur: 4
1	1	1	**0** = –1 (statt 3) + Korrektur: 1 (4); •Korrektur: 4

Abbildung 4.25: Multiplikatorcodierung mit Gruppengröße 2

Es gilt: $x = (-2\cdot MQ_1 + MQ_0 + MQ_{-1}) \cdot MD$ mit $-2 \le x \le 2$.
Somit sind alle x-Werte durch Inversion und Shift erhältlich.

Beispiel:



Adder-Tree und Pipelining

Um eine Beschleunigung der Multiplikation zu erhalten, kann durch Verwendung eines Adder-Trees die Addition — auf welche die Multiplikation zurückgeführt wird — parallelisiert werden. Die Multiplikationsmatrix wird zu je drei Zeilen in CSAs eines Adder-Trees gegeben.

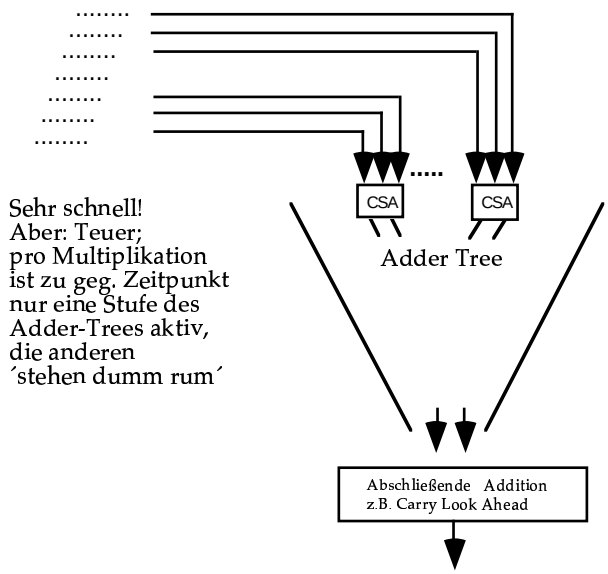


Abbildung 4.26: Multiplikation als Addition im Adder-Tree

Diese Vorgehensweise eignet sich, um **mehrere Multiplikationen nacheinander** auszuführen. Ein Beispiel ist die Bildung der Summe $\sum_{i=1}^{k} a_i \cdot b_i$. Hier werden die Multiplikationen stufenweise durch den Adder-Tree geführt (Pipelining).

Abbildung 4.27: Pipelining durch Adder-Tree

Pipeline pro Einzelmultiplikation zusammen mit Multiplikatorcodierung:

Um für eine einzelne Multiplikation ähnlich vorgehen zu können, wird der Multiplikator (zum Beispiel 60 Bits lang) in Gruppen zerlegt. Die Gruppen werden nacheinander in den Adder-Tree gegeben. Daher eignet sich besonders eine Gruppengröße von 12 Bits. Auf jede Gruppe wird die Multiplikatorcodierung angewendet, wodurch die Zahl der Vielfachen auf die Hälfte verringert wird, im Beispiel auf 6. Die Gruppen (im Beispiel 5) werden in einen kleinen Adder-Tree (6 Eingänge) gegeben. Der Adder-Tree muß Rückkopplungen

aufweisen, die den errechneten Wert (mit gleichzeitigem Rechtsshift um die Gruppengröße) mit dem nachfolgenden neuen Wert zusammen verarbeiten. Von den Rückkopplungen hängt die Laufzeit ab.
Ist die Rückkopplungsschleife bis oben zu den Blättern des Adder-Trees gelegt, so ist keine hohe Taktfrequenz möglich. Ein Adder-Tree mit kurzer Rückkopplungsleitung ist dagegen schneller.

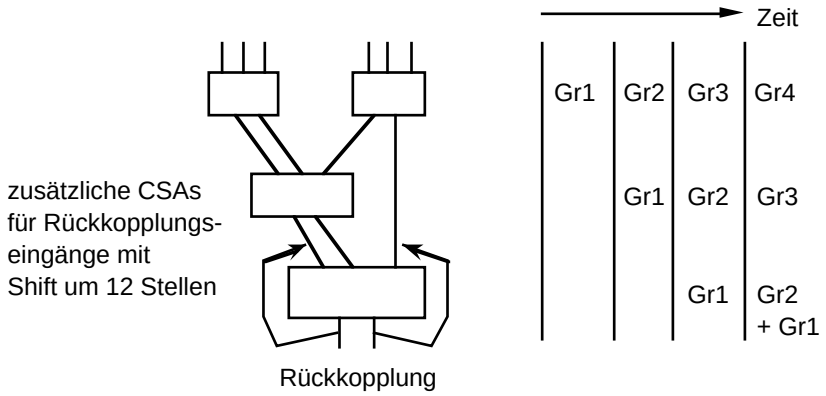


Abbildung 4.28: Adder-Tree für Einzelmultiplikation

4.11 Division

Wie bei der Multiplikation wird zunächst ein einfaches Verfahren zur Division vorgestellt (serielle Division). In einem weiteren Abschnitt wird mit der iterativen Division eine wesentlich elegantere Methode betrachtet, welche für die iterative Berechnung der Quadratwurzel modifiziert wird. Es seien folgende Abkürzungen eingeführt:
DR bezeichnet den Divisor, DD den Dividenden und DE den Quotienten.

Serielle Division

Die serielle Division stellt eine Analogie zur seriellen Multiplikation dar.
Sie entsteht durch Umkehrung des entsprechenden Vorgangs bei der Multiplikation:

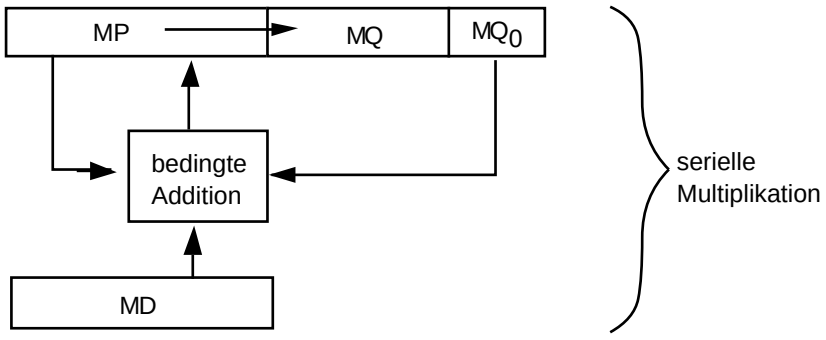


Abbildung 4.29: Serielle Multiplikation

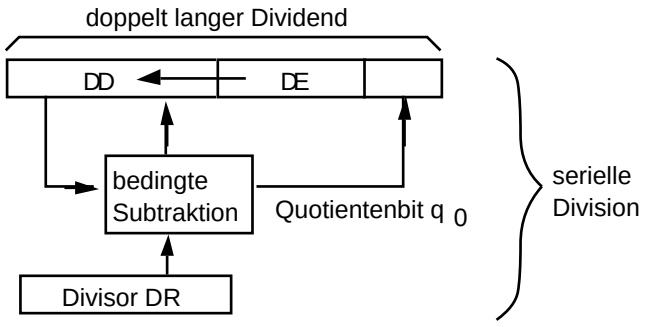


Abbildung 4.30: Serielle Division

Es besteht folgende Analogie:

\F(Produkt;Multiplikand) = Multiplikator \F(Dividend;Divisor) = Quotient

Die Division nach Schulmethode ist für den binären Fall durch das u.g. Mikroprogramm angegeben. Sei (DD,DE)=[DD_{n-1},...,DD_0,DE_{n-1},...,DE_0] das Dividendenregister, (DR)=[DR_{n-1},...,DR_0] das Divisorregister und (DE)=[DE_{n-1},...,DE_0] das Quotientenregister. (DD,DE) ist also doppelt so lang wie DR (ggfs. auffüllen). Die Zahlendarstellung erfolgt im 2-Komplement. Ferner wird vereinbart, daß Dividend und Divisor normiert und positiv sind. Es gilt: $0 \le w(DD) < w(DR) < 1$.
Die wesentliche Fallunterscheidung lautet: Ist DD ≥ DR, so subtrahiere DR von DD und setze das Quotientenbit q_0 gleich 1, ansonsten führe keine Subtraktion durch und setze q_0 auf 0.

Mikroprogramm:

```
0:  [ (DD, DE) := Dividend;  DR := Divisor;  Z := n ]
1:  [ Z := Z–1;
      if DD ≥ DR   then q := 1;  DD := DD–DR
                   else q := 0 ]
2:  [ DE0 := q;
      if Z > 0   then SHL( DD, DE );  goto 1
                 else SHL( DE ) ]
3:  Stop
```

Bemerkungen:
Am Ende der Programmausführung steht im Register DD der Partialrest der Division, und DE enthält den Quotienten. Der Divisionsrest ist halb so groß wie der Partialrest, daher wendet man den letzten Shift nur noch auf das Quotientenregister DE an. Für die Korrektheit des Verfahrens wurde vorausgesetzt, daß 0 ≤ Wert(DD, DE) < Wert(DR) < 1 gilt. Diese Variante wird auch als ***Non Performing-Division*** bezeichnet.
Eine andere Variante ist die ***Non Restoring-Division***. Sie unterscheidet sich von obiger Variante dadurch, daß in Zeile 1 immer DR von DD subtrahiert wird und diese Subtraktion gegebenenfalls wieder rückgängig gemacht wird. Anstelle der Abfrage, ob DD ≥ DR ist, mit der Konsequenz:

- falls ja, dann Subtraktion und q := 1,

- falls nein, dann nur q := 0,

kann man auch wie folgt vorgehen (für Wert(DR) > 0):

- Ist Wert(DD, DE) ≥ 0, dann subtrahiere DR.
- Ist Wert(DD, DE) < 0, dann addiere DR.
- Falls das Ergebnis (neuer Wert von (DD,DE)) ≥ 0, dann q := 1.
- Falls das Ergebnis (neuer Wert von (DD,DE)) < 0, dann q := 0.
- Wenn am Ende der entstehende Rest negativ ist, dann korrigiere durch Addtion von DR.

Mikroprogramm:

```
0:  [ (DD, DE) := Dividend;  DR := Divisor;  Z := n;  DE0 := 1 ]
1:  [ Z := Z–1;
      if DE0 = 1      then DD := DD – DR
                      else DD := DD + DR ]
2:  [ DE0 := ¬DD(n–1);
      if Z > 0  then SHL( DD, DE );  goto 1
                else SHL( DE );  if DD(n–1) = 1 then DD := DD + DR ]
3:  Stop
```

Bemerkung:
Zu beachten ist, daß die Mikrooperationen einer Zeile parallel ausgeführt werden, das heißt die linken Seiten der Zuweisungen werden mit einem Mal neu besetzt.
Zu Zeile 1: DE_0 ist das umgekehrte Vorzeichen des vorherigen Quotientenbits.

Korrektheitsüberlegungen:
Geht man davon aus, daß die erstgenannte Methode (*Non Performing*) korrekt arbeitet — Zahlen richtig dividiert —, so kann man zur Überprüfung der Korrektheit der zweiten Methode (*Non Restoring*) diese an der ersten „messen".
Nach der ersten Methode führt man folgende Operationen aus:

Quotientenbits	1	0	...	0	1
	↓ Vergleich: Subtraktion	↓ Vergleich: Keine Operation		↓ Vergleich: Keine Operation	↓ Zuletzt: Subtraktion
Der Partialrest ist stets nicht negativ.					

Nach der zweiten Methode geht man wie folgt vor:

Quotientenbits	1	0	...	0	1
	↓	↓	↓	↓	↓
Partialrest	positiv	negativ	...	negativ	positiv

	↓ Subtraktion	↓ Subtraktion	↓ Addition	↓ Addition	↓ Addition

Bei jedem Quotientenbit 1 sind die Partialreste beider Methoden gleich groß; an den Stellen, wo die Quotientenbits 0 sind, natürlich nicht:

Position	(k–1)	...	1	0
Quotientenbits	0	...	0	1
	↓ Keine Operation	↓ ...	↓ Keine Operation	↓ Subtraktion

Insgesamt ergibt sich: $-2^0 \cdot DR = -DR$.

Position	(k–1)	...	...	1	0
Quotientenbits	0	...	...	0	1
	↓ Subtraktion	↓ Addition	↓ ...	↓ Addition	↓ Addition

Hier erhält man: $(-2^{(k-1)} + 2^{(k-2)} + ... + 2^0) \cdot DR = -DR$.

Non Restoring Division — Beispiel:

Mikroprogramm:

```
0:  [ (DD, DE) := Dividend;  DR := Divisor;  Z := n;  DE0 := 1]
1:  [ Z := Z–1;
      if DE0 = 1      then DD := DD – DR
                      else DD := DD + DR ]
2:  [ DE0 := ¬DD(n–1);
      if Z > 0  then SHL( DD, DE );  goto 1
                else SHL( DE );  if DD(n–1) = 1 then DD := DD + DR ]
3:  Stop
```

Beispiel:

Sei (DD, DE) = (0.10001110010), Wert(DD, DE) = $569/2^{10} \approx 0{,}55566...$,
DR = 0.111100, Wert(DR) = $30/2^5 \approx 0{,}9375$.
Es gilt: $0 \le$ Wert(DD, DE) $<$ Wert(DR) < 1.

DD	DE
010001 –DR	110011
110011 100111 +DR	10011 | **0**
000101 001011	0011 | 0**1**

–DR	
101101 011010 +DR	011 | 01**0**
111000 110000 +DR	11 | 010**0**
001110 011101 –DR	1 | 0100**1**
111111 +DR	010010 Shift von DE, da Rest negativ ist.
011101	010010
Rest: $\backslash F(29;30) \cdot \backslash F(1;2^5)$	Quotient: $\backslash F(18;2^5) = 0{,}5625$

Beschleunigungsversuche

A. Die **Multiplikatorcodierung** ist in entsprechend einfacher und leistungsfähiger Form nicht auf die Division übertragbar (niemand hat bisher ein solches Konzept gefunden).

B. Verwendung spezieller Divisorvielfacher
Die Division ist schnell, wenn der neue Partialrest möglichst betragsklein ist. Falls sich DD (alter Partialrest) und DR stark unterscheiden, bringt der nächste Zyklus {DD–DR oder DD+DR (oder NOOP)} noch nicht viel im Sinne eines kleinen Partialrests. Stellt man den Divisor (DR) dem Dividenden (DD) in einer Matrix gegenüber, so kann man drei Fälle unterscheiden:

	DR
DD	|DD| << DR: Bilde DD := DD – 0,5•DR, falls DD≥0 DD + 0,5•DR, falls DD< 0 Ist |DD| ≈ DR: Operation wie üblich: DD := DD – DR, falls DD ≥ 0, DD := DD +DR, falls DD < 0. |DD| >> DR: Bilde DD := DD – 2•DR, falls DD≥0 DD+ 2•DR, falls DD< 0

Abbildung 4.31: Divisorvielfache

Man versucht also, ein geeignetes Vielfaches von DR einzusetzen.

Beispiel:

Sei 0 < DD <<DR:

Bilde DD := DD – 0,5·DR und berechne zwei Quotientenbits (00 und 01) je nachdem, ob das Ergebnis negativ oder positiv ist. Hier ist das { \F(1;2), 1, 2 }-System verwendet worden. Es gibt auch Ansätze, ein { \F(3;4), 1, \F(3;2) }- oder ein { \F(5;8), 1, \F(5;4) }-System zu verwenden. Dabei ist der letzte Ansatz besser als der zweite, und dieser ist besser als das { \F(1;2), 1, 2 }-System in folgendem Sinn: Der Betrag des neuen Partialrests ist im Mittel möglichst klein. Daher ist die Zahl der im Mittel pro Zyklus berechenbaren Quotientenbits bei dem { \F(5;8), 1, \F(5;4) }-System am höchsten.

Ein Kompromiß ist die **Table-Look-Up-Methode.** Sie arbeitet mit einer Matrix über Divisor und Dividend. Es wird ein Näherungsquotient aufgrund der ersten Bits von Dividend und Divisor bestimmt.

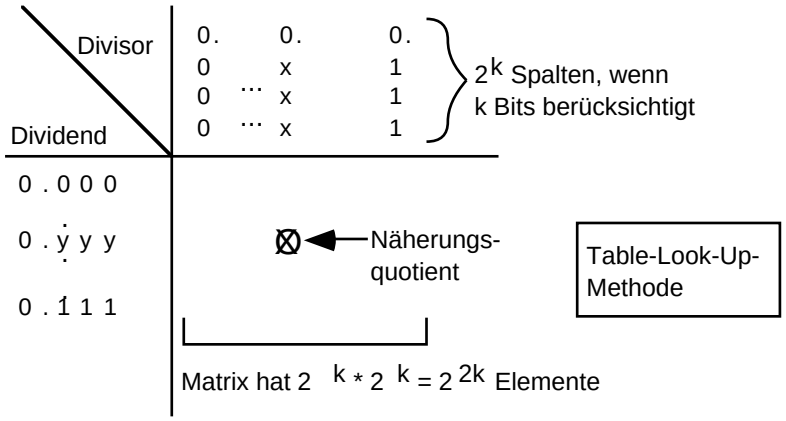


Abbildung 4.32: Table-Look-Up-Methode

Dieser Näherungsquotient kann als Startwert in iterativen Verfahren zur Division verwendet werden.

Iterative Division

In diesem Abschnitt wird ein iteratives Verfahren zur Division vorgestellt. Man geht von der Rückführung der Division auf eine Folge von Multiplikationen und Additionen aus. Da die Division im allgemeinen seltener benötigt wird als die Multiplikation, ist diese Rückführung der Konstruktion einer eigenen „Divisionshardware" vorzuziehen und rechtfertigt ein schnelles, aber teures Multiplizierwerk.

Das hier vorgestellte Verfahren ist iterativ:
Man geht von einem Startwert x_0 aus und wendet auf jeden Wert x_i eine rekursive Iterationsvorschrift φ an, welche den Wert $x_{(i+1)}$ liefert.

$$x_0 \rightarrow x_1 := \varphi(x_0) \rightarrow x_2 := \varphi(x_1) \rightarrow \ldots \rightarrow x_{(i+1)} := \varphi(x_i) \rightarrow \ldots$$

Unter geeigneter Wahl des Startwertes konvergiert φ gegen einen Grenzwert. Gilt $\bar{x} = \varphi(\bar{x})$, so heißt $\bar{x}$ **Fixpunkt** von φ.

Definition:
Sei x_0 ein geeignet gewählter Startwert.
φ **konvergiert mit Ordnung m gegen einen Grenzwert** $\bar{x}$ mit $m \in N_0 \Leftrightarrow$ für den Iterationsfehler $\varepsilon_i := x_i - \bar{x}$ gilt: $\varepsilon_{i+1} = O(\varepsilon_i^m)$.

Bemerkung:
Ist m = 1, so konvergiert φ mit linearer Geschwindigkeit gegen den Grenzwert; ist m = 3, so konvergiert φ mit kubischer Geschwindigkeit gegen $\bar{x}$.

Beispiel: Regula Falsi

Sei f eine Funktion, die eine Nullstelle besitzt. Man geht von zwei Stellen x_0 und x_1 aus, für die gilt: $f(x_0) < 0$ und $f(x_1) > 0$ oder umgekehrt. In jedem Schritt legt man eine Gerade durch die Punkte, welche durch die zwei Stellen und ihre Funktionswerte gebildet werden. Dann betrachtet man den Schnittpunkt der Geraden mit der x-Achse.

Man wendet folgende Iterationsvorschrift an:

Seien x_i und $x_{(i+1)}$ die betrachteten Stellen mit $x_i < x_{(i+1)}$.

if f(Schnittstelle) = 0 then Nullstelle := Schnittstelle;
if f(Schnittstelle) < 0 then x_i := Schnittstelle;
if f(Schnittstelle) > 0 then $x_{(i+1)}$:= Schnittstelle;

Das Verfahren konvergiert mit linearer Geschwindigkeit, ist also von der Ordnung m = 1.

Beispiel: Newton-Verfahren

Dieses Verfahren ist aus der Schule bekannt (oder ?). Es dient ebenfalls der iterativen Ermittlung einer Nullstelle einer Funktion f. Man geht von einer Startstelle x_0 aus und erhält $x_{(i+1)}$ durch Anwendung der Iterationsvorschrift $\varphi(x) = x - \backslash F(f(x);f\,'(x))$ auf x_i.

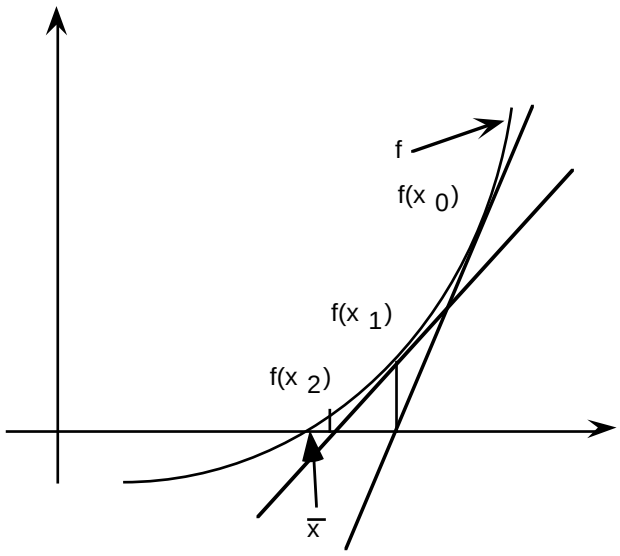


Abbildung 4.33: Newton-Verfahren

Das Newtonverfahren konvergiert quadratisch, also mit Ordnung m=2 gegen eine Nullstelle $\bar{x}$ von f(x), sofern x_0 geeignet gewählt wird und $f'(\bar{x}) \neq 0$ ist.
Das Newton-Verfahren wird nun verwendet, um ein elegantes Verfahren zur Division zu konstruieren. Dabei wird wie folgt $1/b$ aus b errechnet:

- Suche eine Funktion f(x), die $1/b$ als Nullstelle besitzt.
- Berechne $\varphi(x)$, wobei $\varphi(x)$ keine Division enthalten soll.
- Führe Iteration $x_{(i+1)} = \varphi(x_i)$ so oft wie nötig durch, bis Konvergenz im Rahmen der erzielbaren Genauigkeit eintritt.

Versuch 1:

Geht man von einer Funktion $f(x) = x - 1/b$ mit Nullstelle $1/b$ aus, so erhält man keine brauchbare Iterationsvorschrift:

$$\varphi(x) = x - \backslash F(f(x);f'(x)) = x - \backslash F(x-1/b;1) = 1/b$$

Der Fixpunkt $1/b$ muß vorher bekannt sein.

Versuch 2:

Sei $f(x) = -1/x + b$. Diese Funktion hat auch die Nullstelle $1/b$.
Die Iterationsvorschrift lautet:

$$\begin{aligned}\varphi(x) = x - \backslash F(f(x);f'(x)) &= x - \backslash F(-\backslash F(1;x) + b;\backslash F(1;x^2)) \\ &= x - (-x + b \cdot x^2) \\ &= 2 \cdot x - b \cdot x^2 \\ &= (2 - b \cdot x) \cdot x\end{aligned}$$

Mit dieser Iterationsvorschrift hat man eine divisionsfreie Formel erhalten. Die Division ist hiermit auf zwei Multiplikationen und eine Subtraktion zurückgeführt worden, die zum Beispiel im 2-Komplement leicht zu handhaben ist.

Für $b \in [1/2 : 1]$ liegt zum Beispiel die Mantisse einer normalisierten Gleitkommazahl vor.

Die Divisionsvorschrift ist also:

Startwert: $x_0 := 1$ oder
$x_0 :=$ Wert nach Table-Look-Up (Näherungswert für $1/b$)

Iterationsvorschrift:

$$\varphi_{Div}(x) = (2 - b \cdot x) \cdot x$$

Beispiel:
Seien $b = 0{,}85$, $x_0 = 1$.
Dann ergeben sich folgende Näherungswerte, wobei an diesen das quadratische Verhalten des Newton-Verfahrens zu erkennen ist:

$x_1 = (2 - 0{,}85 \cdot 1) \cdot 1,$
$x_2 = 1{,}\underline{17}5875$
$x_3 = 1{,}\underline{1764}703$
$x_4 = 1{,}\underline{1764706}..$

x_4 stellt einen relativ genauen Wert von $1/0{,}85$ dar.

Ein besserer Startwert ist 0,8, denn hier erhält man obige Näherungswerte schon einen Schritt früher:

$x_1^* = 1{,}\underline{17}1875$

$x_2^* = 1{,}\underline{1764}526$
$x_3^* = 1{,}\underline{1764705}..$

Die unterstrichenen Ziffern sind exakt.

φ_{Div} konvergiert quadratisch gegen $1/b$. Der Iterationsfehler des (i+1)-ten Schrittes ist kleiner/gleich Null (es liegt also **quadratische Konvergenz von unten** vor):

$$\varepsilon_{(i+1)} = x_{(i+1)} - \bar{}; x = (2 \cdot b \cdot x_i) \cdot x_i - 1/b = -b \cdot (x_i - 1/b)^2 = -b \cdot e_i^2 \leq 0, \text{ wenn } 0 < b < 1.$$

Anschaulich bedeutet dies eine Verdopplung der Zahl korrekter Bits (Stellen, Ziffern) bei jedem Übergang von x_i zu $x_{(i+1)}$.
Der (i+1)-te Schritt bringt soviel wie die ersten i Schritte zusammen, der Start mit einem guten Näherungswert aus der Table-Look-Up-Tabelle ist sehr zweckmäßig.

Bemerkungen:
Das Verfahren konvergiert nur bei hinreichend genauem Ausgangswert (dann allerdings schnell!).
Bei verkürzter ungenauer Berechnung der Multiplikationen $x_{(i+1)} = (2 - b \cdot x_i) \cdot x_i$ ist die Zeit pro Iterationsschritt kürzer. Die Konvergenzgeschwindigkeit ist dafür aber etwas schlechter als quadratisch.

Für eine kürzere Multiplikation verwendet man die Schreibweise $(2–b\cdot x_i)_{Tr}$ (*Truncated*). $(2–b\cdot x_i)_{Tr}$ enthält nur die relevantesten Bits von $(2–b\cdot x_i)$.
Ein Vorteil besteht darin, daß $(2–b\cdot x_i)_{Tr}$ mit Multiplikatorcodierung auf wenige (z.B. 6) Vielfache codiert und in einem kleinen Adder-Tree verarbeitet werden kann. Ferner wird die Bildung von $(2–b\cdot x_i)$ vereinfacht.

AEGP (Anderson-Earle-Goldschmidt-Power)

Ein Nachteil der bisherigen Iterationsvorschrift $\varphi_{Div}(x) = (2 - b\cdot x) \cdot x$ besteht darin, daß die zwei aufeinanderfolgenden Multiplikationen pro Iterationsschritt eine Parallelisierung verhindern.
Ziel ist es, eine gleichschnelle Konvergenz durch zwei parallele Multiplikationen zu erzielen. Eine Idee besteht darin, daß die erste Multiplikation entfällt, wenn b = 1 ist. Das Verfahren konvergiert dann für jeden sinnvollen Startwert zum Grenzwert 1.
Das Verfahren lautet:

d_0 = Startwert.
$d_{(i+1)} = (2–d_i) \cdot d_i$.

Beispiel:
$d_0 = 1{,}5$
$d_1 = 0{,}5 \cdot 1{,}5 = 0{,}75$
$d_2 = 1{,}25 \cdot 0{,}75 = 1 – (0{,}25)^2 = 1 – {}^1/_{16}$
Ein unbrauchbarer Startwert ist $d_0 = 5$.

Für die Berechnung eines Quotienten ${}^a/_b$ betrachtet man zwei parallele Folgen:
- Folge 1 mit Startwert b und
- Folge 2 mit Startwert a.

Prinzipiell führt man solange eine Multiplikation der aktuellen Werte von Folge 1 und 2 mit dem gleichen Faktor durch, bis Folge 1 den Grenzwert Eins erreicht hat (im Rahmen der Genauigkeit). Wenn dies der Fall ist, dann wurde durch Folge 2 der Wert des Quotienten ${}^a/_b$ approximiert:

$$\backslash F(a;b) = \backslash F(a\cdot r_0;b\cdot r_0) = \backslash F(a\cdot r_0\cdot r_1;b\cdot r_0\cdot r_1) = \ldots = \backslash F(a\cdot r_0\cdot r_1\cdot \ldots r_i;b\cdot r_0\cdot r_1\cdot \ldots r_i)$$

$a\cdot r_0\cdot \ldots r_i$ konvergiert gegen $\backslash F(a;b)$, wenn $b\cdot r_0\cdot \ldots r_i$ gegen 1 konvergiert.
Es gilt insgesamt:

	Folge 1	Folge 2
Startwert	$d_0 := b$	$d_0 := a$
Iterationsvorschrift	$d_{(i+1)} := d_i \cdot (2–d_i)$	$x_{(i+1)} := x_i \cdot (2–d_i)$
Konvergenz	quadratisch gegen	
	$d_E = 1$	$x_E = {}^a/_b$

In einer Realisierung sind nun zwei Multiplikationen ($d_i \cdot (2–d_i)$ und $x_i \cdot (2–d_i)$) pro Iterationsschritt umzusetzen. Sind zwei Multiplizierwerke vorhanden, so kann man die Multiplikationen parallel ausführen. Sonst könnte man das Pipelineprinzip verwenden:

Pipeline A:	d_i	x_i	$d_{(i+1)}$	$x_{(i+1)}$	$d_{(i+2)}$
Pipeline B:	$x_{(i-1)}$	d_i	x_i	$d_{(i+1)}$	$x_{(i+1)}$

——————————————————Zeit→

Ferner kann man die verkürzten Formen $(d_i \cdot (2-d_i))_{Tr}$ und $(x_i \cdot (2-d_i))_{Tr}$ in einen Adder-Tree geben, wobei man dann zunächst ungenau aber schnell rechnet und im letzten Schritt die Folge 2 genau berechnet.

4.12 Iterative Berechnung von Quadratwurzeln

Um die Quadratwurzel einer Zahl N zu berechnen, kann man ähnlich vorgehen wie bei der iterativen Division. Man verwendet wieder das Newton-Verfahren mit einer geeigneten Funktion f, welche $\sqrt{;N}$ als Nullstelle besitzt.

- Iterationsvorschrift für f: $x_{(i+1)} = \varphi(x_i) = x_i - \backslash F(f(x_i);f'(x_i))$

Wählt man $f(x) = x^2 - N$, so ergibt sich folgende Iterationsvorschrift für das „iterative Wurzelziehen":

- $x_{(i+1)} = \varphi(x_i) = x_i - \backslash F(f(x_i);f'(x_i)) = x_i - \backslash F(x_i^2 - N;2 \cdot x_i) = \backslash F(1;2) \cdot (x_i + \backslash F(N;x_i))$

Das Verfahren konvergiert quadratisch gegen $\sqrt{;N}$.

Beispiel: Quadratwurzel aus 2

Sei N=2 und $x_0 = 100$ (ziemlich schlechter Startwert).

Es ergibt sich folgende Näherungsfolge:

$x_0 = 100$
$x_1 = {}^1/_2 \cdot (100 + {}^2/_{100}) \approx 50$ (ungenau)
$x_2 \approx 25$
$x_3 \approx 12$
$x_4 \approx 6$
$x_5 \approx 3$
$x_6 \approx {}^1/_2 \cdot (3 + {}^2/_3) \approx 1{,}83$ (ungenau gerechnet ergäbe sich: 1,5)
$x_7 \approx 1{,}46$
$x_8 \approx 1{,}4149...$ (schon recht genau)

Ein großer Nachteil des Verfahrens besteht darin, daß es nicht divisionsfrei ist (N / x_i). Daher kommt man wieder auf eine AEGP-verwandte Methode zurück (siehe Abschnitt über iterative Division).
Man verwendet hier die Folgen (d_i) und (x_i) mit:

- $d_0 := N$, $\quad d_{(i+1)} := d_i \cdot r_i^2$,
- $x_0 := N$, $\quad x_{(i+1)} := x_i \cdot r_i$,

wobei $r_i = 1 + {}^1/_2 \cdot (1-d_i)$ ist. Hier liegt aber keine eigentliche Division vor, denn man kann stattdessen einen Rechtssshift durchführen.
Es gilt, daß $d_n = d_0 \cdot r_0^2 \cdot r_1^2 \cdot \ldots r_{(n-1)}^2$ quadratisch gegen 1 konvergiert, wobei auch hier wieder ein geeigneter Startwert vorausgesetzt sei.
Die Folge (x_i) konvergiert quadratisch gegen $\sqrt{;N}$:

$$x_n = x_0 \cdot r_0 \cdot r_1 \cdot \ldots r_{(n-1)} \rightarrow x_0 \cdot \backslash F(1;\backslash R(;d_0)) = \backslash F(N;\backslash R(;N)) = \backslash R(;N)$$

Beispiel: Quadratwurzel aus 2

Sei N = 2 und somit $d_0 = x_0 = 2$.
Man erhält die Folgen (d_i), (x_i) und (r_i):

$d_0 = 2$	$x_0 = 2$	$r_0 = 1 + {}^1/_2 \cdot (1-2) = {}^1/_2$
$d_1 = 2 \cdot {}^1/_4 = {}^1/_2$	$x_1 = 2 \cdot {}^1/_2 = 1$	$r_1 = 1 + {}^1/_2 \cdot (1-{}^1/_2) = {}^5/_4$
$d_2 = {}^{25}/_{32}$	$x_2 = {}^5/_4$	$r_2 = 1 + {}^7/_{64}$
$d_3 = 0{,}9614943$	$x_3 = 1{,}3867187$	$r_3 = 1{,}0192528$
$d_4 = 0{,}998872$	$x_4 = 1{,}4134169$	$r_4 = 1{,}0005635$
$d_5 = 0{,}999989$	$x_5 = 1{,}4142133$	$r_5 = 1{,}0000005$

Beweis:
Es ist zu zeigen, daß $d_{(i+1)}$ gegen 1 konvergiert.
Zunächst ist: $d_{(i+1)} = d_i \cdot (1 + {}^1/_2 \cdot (1-d_i))^2 = d_i \cdot (2 - d_i + {}^1/_4 \cdot (1-d_i)^2)$.

Hilfssatz:
Eine Iterationsvorschrift $x_{(i+1)} := \varphi(x_i)$ konvergiert mit Ordnung m gegen $\bar{x}$, [das heißt: Für $\varepsilon_i = x_i - \bar{x}$ gilt: $\varepsilon_{(i+1)} = O(\varepsilon_i^m)$] $\Leftrightarrow$

$\varphi(\bar{x}) = \bar{x}$ und $\varphi'(\bar{x}) = \ldots = \varphi^{(m-1)}(\bar{x}) = 0$ und $\varphi^{(m)}(\bar{x}) \neq 0$

Taylorentwicklung des Fehlers:

$$\varepsilon_{(i+1)} = x_{(i+1)} - \bar{x} = \varphi(x_i) - \bar{x} = \varphi(\bar{x} - \varepsilon_j) - \bar{x}$$

$$= \sum_{k=0}^{\infty} \frac{\varphi^{(k)}(x)}{k!} \cdot \varepsilon_i^k - \bar{x} = \sum_{k=1}^{\infty} \frac{\varphi^{(k)}(x)}{k!} \cdot \varepsilon_i^k$$

Daher: $\varepsilon_{(i+1)} = O(\varepsilon_i^m) \Leftrightarrow \varphi'(\bar{x}) = \ldots = \varphi^{(m-1)}(\bar{x}) = 0$ und $\varphi^{(m)}(\bar{x}) \neq 0$. q.e.d.
Es bleibt noch zu zeigen, daß $d_{(i+1)} = d_i \cdot (2 - d_i + {}^1/_4 \cdot (1-d_i)^2)$ quadratisch gegen $\bar{x} = 1$ konvergiert.
Die Iterationsvorschrift lautet hier:

$d_{(i+1)} = \varphi(d_i)$ mit $\varphi(x) = x \cdot (2 - x + {}^1/_4 \cdot (1-x)^2)$,

Ableitungen:

$\varphi'(x) = 1 \cdot (2 - x + {}^1/_4 \cdot (1-x)^2) + x \cdot (-1 + {}^1/_4 \cdot 2 \cdot (1-x) \cdot (-1))$
$\varphi''(x) = -2 + \backslash F(6 \cdot x;4)$

(d_i) konvergiert gegen 1, weil $\varphi(1) = 1$, $\varphi'(1) = 0$ und $\varphi''(1) \neq 0$ ist:

$\varphi(1) = 1 \cdot (2 - 1 + 0) = 1 = \bar{x}$,
$\varphi'(1) = 1 \cdot (2 - 1 + 0) + 1 \cdot (-1 + 0) = 0$,
$\varphi''(1) = -{}^1/_2$.

4.13 Arithmetik bei redundanter Zahlendarstellung

Die letzten Abschnitte — insbesondere über die Addition — haben gezeigt, daß der Übertrag in ungünstigen Fällen eine lange Laufzeit der Rechenoperation bewirkt. Der Übertrag kann sich zum Beispiel auf die gesamte Addition auswirken. In diesem Abschnitt werden Zahlendarstellungen für die Addition und für die Division besprochen, die auf Kosten einer erhöhten Redundanz eine beschleunigte Durchführung der Operationen ermöglichen.

SDNR-Darstellung für die Addition

Die grundlegende Idee besteht darin, eine oder mehrere zusätzliche **Redundanzziffern** einzuführen, in denen sich der Übertrag „fängt", falls er überhaupt entsteht. Man führt zum Beispiel eine zusätzliche „2" zu den binären Ziffern „0" und „1" ein. Die Ausgangszahlen könnte man binär darstellen und das Ergebnis ternär interpretieren. Dies wird im folgenden Abschnitt präzisiert.

Die Zahlendarstellung ist in Stellenwertform zu einer Basis d. Dabei sind im codierenden Bitwort nun nicht mehr nur die binären Ziffern zulässig, sondern positive und negative Ziffern $\alpha_i \in [-r_1 : +r_2]$, $r_1, r_2 > 0$:

$$W_{SDNR}(\alpha_{(n-1)}, \alpha_{(n-2)}, \ldots, \alpha_0) = \sum_{i=0;n-1} (\alpha_i \cdot d^i)$$

Beispiel:

Sei $\bar{x} := -x$.

$W_{SDNR}(2\bar{}; 31\bar{}; 4) = 2000 - 300 + 10 - 4 = 1706$.

Definition: SDNR (*Signed Digit Number Representation*)

Eine Zahlendarstellung, die folgende Eigenschaften hat, heißt SDNR:

1. Symmetrie: $r_1 = r_2 = r$
2. Für α_i sollen mindestens d verschiedene Werte möglichsein. Der Übertrag (+1 oder –1) aus der Stelle (i–1) soll „sich in der Stelle i fangen".
   Dies ist erfüllt, wenn $2 \cdot r + 1 \geq d + 2$ gilt, also $\lfloor F(d;2) \rfloor + 1 \leq r$;
   $2 \cdot r + 1$ ist die Anzahl möglicher Ziffern.
3. Für α_i sollen höchstens d nicht-negative Werte zulässig sein, also $r \leq d-1$.

Bemerkung:
Das Vorzeichen des Wertes einer SDNR-Zahl ist gleich dem Vorzeichen der linken Ziffer der Zahl, die nicht 0 ist.

Konsequenz:
a) Für die Basis d = 2 gibt es keine SDNR-Darstellung, denn:
$\lfloor F(d;2) \rfloor + 1 \leq r \leq d-1$ (Bedingung 2. und 3.) liefert für d = 2:

$1+1 \le r \le 2-1$, also $2 \le 1$, was zu einem Widerspruch führt.

b) Für die Basis $d = 3$ ist eine SDNR-Darstellung möglich:
$\lfloor \backslash F(3;2) \rfloor + 1 \le r \le 3-1$, folglich: $r = 2$ ist hier eindeutig bestimmt.
Somit ist die Codierung festgelegt:

$$W_{SDNR}(\alpha_{(n-1)}, \alpha_{(n-2)}, ..., \alpha_0) = \backslash I \backslash SU(i=0;n-1;\ \alpha_i \cdot 3^i), \text{ mit } \alpha_i \in [-2 : +2].$$

Beispiel: Addition

Schreibweise: $\bar{}\,;x := -x$.

Es werden die Zahlen $(2\ \bar{}\,;1\ \bar{}\,;2\ \bar{}\,;2\ 0\ 1\ 2)$ und $(\bar{}\,;1\ \bar{}\,;2\ \bar{}\,;2\ 1\ 2\ 2\ 2)$ addiert. Dabei wird das jeweilige Summenbit in der oberen Zeile und das Übertragsbit in der unteren — um eine Ziffer nach links geschoben — notiert. Diese Notation wurde schon bei der Von-Neumann-Addition verwendet. Im ersten Schritt der Addition wird darauf geachtet, daß die Summenbits $s_i \in [-1 : +1]$ sind.

$$\begin{array}{r} 2\ \bar{}\,;1\ \bar{}\,;2\ \bar{}\,;2\ 0\ 1\ 2 \\ \underline{\bar{}\,;1\ \bar{}\,;2\ \bar{}\,;2\ 1\ 2\ 2\ 2} \\ 1\ \bar{}\,;0\ \bar{}\,;1\ \bar{}\,;1\ \bar{}\,;\mathbf{1}\ 0\ 1 \\ \underline{0\ \bar{}\,;1\ \bar{}\,;1\ 0\ \mathbf{1}\ 1\ 1\ 0} \\ 0\ \bar{}\,;1\ \bar{}\,;1\ 0\ 0\ 1\ 1 \end{array}$$

Die fettgedruckten Ziffern zeigen, wie man $s_2 = 2$ vermeidet:
$2 = 3 \cdot 1 - 1$. Auf diese Weise entsteht im zweiten Schritt der Addition kein Übertrag mehr.

Bei der Berechnung im ersten Schritt bildet man die Summenbits so, daß die Summenbits s_i nicht den Ziffernbereich „bis oben beziehungsweise unten" ausfüllen: $s_i \in [-(r-1) : +(r-1)]$. Der Übertrag $+1$ oder -1 fängt sich dann im zweiten Schritt.
Allgemein für die Basis d und $\lfloor \backslash F(d;2) \rfloor + 1 \le r \le d-1$ wird folgendermaßen addiert:

		$\alpha_{(n-1)}$	...		α_0	
+		$\beta_{(n-1)}$	...		β_0	
		$\sigma_{(n-1)}$	...		σ_0	Zwischensumme
+	$c_{(n-1)}$	$c_{(n-2)}$	...	c_0	0	Vorläufiger Übertrag
	s_n	$s_{(n-1)}$	...		s_0	Summe

Dabei sind: $\alpha_i, \beta_i, s_i \in [-r : +r]$, $\sigma_i \in [-(r-1) : +(r-1)]$ und $c_i \in [-1 : +1]$.
Für Schritt 1 sind folgende Formeln denkbar:

Sei $w_{max} \in [(d-r) : (r-1)]$, zum Beispiel also $w_{max} = r-1$.

Setze $c_i = \{ +1;\text{falls } \alpha_i+\beta_i > w_{max};0;\text{sonst};-1;\text{falls } \alpha_i+\beta_i < -w_{max}$

setze $\sigma_i = \alpha_i+\beta_i - c_i \cdot d$, wegen der Invarianzeigenschaft: $\alpha_i+\beta_i = c_i \cdot d + \sigma_i$.

Dann ist $|\sigma_i| < r$ für i = 0, ... (n–1) erfüllt und somit istSchritt 2 ohne Überträge möglich.

„Gemischtes" SDNR-Verfahren

Die Addition von Zahlen in der SDNR-Darstellung geht sehr schnell. Zu beachten ist allerdings, daß die Zahlen zunächst in dieser Form codiert sein müssen. Der Aufwand für diese Codierung muß daher berücksichtigt werden. Günstig ist es, das Verfahren zu verallgemeinern, so daß eine Addition von Zahlen durchgeführt werden kann, wobei eine Zahl in SDNR-Darstellung zur Basis d und die andere in normaler Stellenwertcodierung zur Basis d vorliegt:

		$A_{(n-1)}$	...		A_0	
+		$b_{(n-1)}$	...		b_0	
		$T_{(n-1)}$	...		T_0	Zwischensumme
+	$c_{(n-1)}$	$c_{(n-2)}$	...	c_0	0	Vorläufiger Übertrag
	s_n	$s_{(n-1)}$	...		s_0	Summe

Die großen Buchstaben kennzeichnen Ziffern einer SDNR-Zahl:
$A_i \in [-(d-1) : +(d-1)]$; die kleinen Buchstaben bezeichnen Ziffern einer d-nären Zahl, also $b_i \in [0 : (d-1)]$.
Die Formeln für Schritt 1 und Schritt 2 der Addition lauten:

Schritt 1:

Setze $c_i = \begin{cases} 1; \text{wenn } A_i+b_i \geq d-1 \\ 0; \text{sonst} \end{cases}$,

$T_i = A_i+b_i - c_i \cdot d \in [-(d-1) : +(d-2)]$.

Schritt 2:

Setze $S_i = T_i + c_{(i-1)} \in [-(d-1) : +(d-1)]$, ohne weiteren Übertrag.

Dieses „gemischte" Verfahren funktioniert auch bei Codierung zur Basis d=2.
Hier liegt obiges Schema mit entsprechender Wahl der Ziffernbereiche zugrunde:

		$A_{(n-1)}$	...		A_0	
+		$b_{(n-1)}$	...		b_0	
		$T_{(n-1)}$	...		T_0	Zwischensumme
+	$c_{(n-1)}$	$c_{(n-2)}$	...	c_0	0	Vorläufiger Übertrag
	s_n	$s_{(n-1)}$	...		s_0	Summe

$A_i \in \{-1, 0, +1\}$, $b_i \in \{0, 1\}$, $T_i \in \{-1, 0\}$, $c_i \in \{0, 1\}$ für i = 0, ... (n–1) und $S_i \in \{-1, 0, 1\}$ für i = 0, ... n.

A_i, T_i und c_i werden binär codiert (b_i liegt binär vor und S_i wird umgewandelt):

- A_i auf zwei Binärstellen, wobei sich „Betrag & Vorzeichen" eignet:

$a_{i,1}$	$a_{i,2}$	Ai
0	0	+0
0	1	+1
1	0	−0
1	1	−1

- c_i und T_i können auf einer Binärstelle codiert werden.

Für die Realisierung der Berechnung eignen sich folgende Formeln:

a) $A_i = a_{i,2} \cdot (1-2 \cdot a_{i,1})$, also $a_{i,2} = A_i$ modulo 2

b) $c_i = 1 \Leftrightarrow A_i + b_i \geq 1 \Leftrightarrow A_i = 1 \vee A_i = 0 \wedge b_i = 1$
Formel: $c_i = \overline{a_{i,1}} \cdot a_{i,2} + \overline{a_{i,2}} \cdot b_i$

c) $T_i = -1 \Leftrightarrow A_i + b_i = 1$ modulo $2 \Leftrightarrow a_{i,2} + b_i = 1$ modulo 2
Formel: $T_i = -(a_{i,2} \oplus b_i)$; $|T_i| = a_{i,2} \oplus b_i$

d) Formeln für $S_i = (s_{i,1}; s_{i,2})$:
$s_{i,2} = 1 \Leftrightarrow S_i \neq 0 \Leftrightarrow T_i + c_{(i-1)} = 1$ modulo $2 \Leftrightarrow |T_i| \oplus c_{(i-1)} = 1$,
also: $s_{i,2} = |T_i| \oplus c_{(i-1)} = a_{i,2} \oplus b_i \oplus c_{(i-1)}$
$s_{i,1} = |T_i|$ oder auch $s_{i,1} = \overline{c_{(i-1)}}$

(Die zwei unterschiedliche Möglichkeiten beruhen auf der Redundanz der Zahlendarstellung.)

SRT-Verfahren zur seriellen Division

Dieses Verfahren ist benannt nach Sweeney, Robertson und Tocher.
Es verbessert das Prinzip der seriellen Division in der Form, daß der neue Partialrest möglichst betragsklein gewählt wird. Das Prinzip der seriellen Division besteht — wie in vorhergehenden Abschnitten besprochen — aus folgenden Schritten:

- Subtrahiere vom Dividenden ein „geeignetes" Vielfaches des Divisors.
- Bestimme Quotientenbit(s) in Abhängigkeit von dieser Subtraktion.
- Wiederhole diesen Vorgang so lange mit jeweils neuem kleineren Partialrest, bis der Partialrest näherungsweise Null ist (bis also der Quotient vollständig berechnet ist).

Das Ziel ist, den Partialrest X möglichst betragsklein zu machen:

- Formel (für eine Basis d der Zahlendarstellung):
$X^{(n)} := W(DD, DE)$ (Wert von Register (DD, DE), s.o.)
$X^{(j)} := d \cdot (X^{(j+1)} - q_j \cdot W(DR))$, für alle $j = (n-1), \ldots 0$,
wobei $q_j \in \{-(d-1), \ldots, 0, \ldots, (d-1)\}$ ein Quotientenbit ist.
Dieses Quotientenbit ist nun so zu wählen, daß $X^{(j)}$ möglichst betragsklein wird, das heißt, für alle q_j' gilt: $|X^{(j)}(q_j)| \leq |X^{(j)}(q_j')|$.

Formeln für die SRT-Division im binären Fall

Man kann folgende zwei Varianten unterscheiden: Das SRT1-Verfahren ist ziemlich genau, aber aufwendiger als das SRT2-Verfahren, welches dafür ungenauer arbeitet.

SRT1:
Es sei vorausgesetzt, daß $^1/_2 \le W(DR) < 1$ ist.
Dann ist SRT1 definiert durch:

- $X^{(n)} := W(DD, DE)$
- $X^{(j)} := 2 \cdot (x^{(j+1)} - q_j \cdot W(DR))$, für $j = (n-1), \ldots 0$, mit:

$$q_j := \{ +1;\text{falls } X^{(j+1)} > {}^1/_2 \cdot W(DR);0;\text{sonst};-1;\text{falls } X^{(j+1)} < -{}^1/_2 \cdot W(DR)$$

SRT2:
Es ist wieder vorausgesetzt, daß $^1/_2 \le W(DR) < 1$ ist.
Dann ist SRT2 definiert durch:

- $X^{(n)} := W(DD, DE)$
- $X^{(j)} := 2 \cdot (x^{(j+1)} - q_j \cdot W(DR))$, für $j = (n-1), \ldots 0$, mit:

$$q_j := \{ +1;\text{falls } X^{(j+1)} \ge {}^1/_2;0;\text{sonst};-1;\text{falls } X^{(j+1)} < -{}^1/_2$$

Hier ist also die Bestimmung von q_j einfacher, denn es muß kein expliziter Vergleich durchgeführt werden, sondern es reicht, die ersten beiden Bits zu prüfen:
Sind $x_{(n-1)}^{(j+1)}$ und $x_{(n-2)}^{(j+1)}$ die ersten beiden Bits von X(j+1), dann wird q_j wie folgt bestimmt:

$$q_j := \B\BC\{(\A\AL\CO2\HS9(+1;\text{falls } x_{(n-1)}^{(j+1)} = 0 \text{ und } x_{(n-2)}^{(j+1)} = 1;0;\text{falls } x_{(n-1)}^{(j+1)} = x_{(n-2)}^{(j+1)};-1;\text{falls } x_{(n-1)}^{(j+1)} = 1 \text{ und } x_{(n-2)}^{(j+1)} = 0))$$
$$= -x_{(n-1)}^{(j+1)} + x_{(n-2)}^{(j+1)} .$$

Mikroprogramm zur SRT-Division

Es wird nun eine einfache Version eines Mikroprogramms für die SRT-Division angegeben. Hierzu seien folgende Bezeichnungen vereinbart:

- Register:
  (DD, DE): Doppelt langer Dividend,
  DR: Divisor,
  DE: Nimmt positive Quotientenbits auf,
  DH: Nimmt negative Quotientenbits auf,
  Für DE und DH gilt, daß sie — falls kein Quotientenbit aufzunehmen ist — eine 0 an der Stelle j enthalten: $DE_j, DH_j = 0$.
- Zähler: Z (wie früher)

Das Mikroprogramm für nicht-negative Faktoren ($0 \le W(DD, DE) < W(DR) < 1$) lautet:

Mikroprogramm:

0: [(DD, DE) := Dividend; DR := Divisor; DH := 0; Z := n]
1: Z := Z–1; q := $-DD_{(n-1)} + DD_{(n-2)}$;

```
        if DD(n–1) ≠ DD(n–2) then
              [ if DD(n–1) = 1  then DD := DD + DR else DD := DD – DR]
2:      (DE0, DH0) :=   if q = 1 then (1, 0)
                        elseif q = –1 then (0, 1) else (0, 0)
        if Z > 0      then [ SHL(DD, DE);  SHL(DH);  goto 1 ]
                      else [ SHL(DE);  SHL(DH) ]
3:      DE := DE – DH
```

Bemerkung:
In DH werden die negativen Quotientenbits als Einsen abgelegt. Daher kann im letzten Schritt subtrahiert werden, um das Ergebnis zu dekonvertieren.

5. Schlußbemerkungen

Dieses Buch gab einen Einblick in den Aufbau und die Funktionsweise von Computern. Dazu wurde zunächst beschrieben, aus welchen Komponenten Rechner aufgebaut sind und wie diese Einheiten miteinander interagieren.

Daten und Befehle werden in Computern binär gespeichert. Die wichtigsten Darstellungsformen für Zahlen und Befehle wurden erklärt und es wurde auf ihre Vor- und Nachteile hingewiesen. Die meisten Befehle eines Programms operieren auf Daten, die im Hauptspeicher des Rechners abgelegt sind. Es wurden verschiedene Adressierungsvarianten vorgestellt, mit denen die Zellen des Speichers angesprochen werden können.

Schaltkreise von Computern bestehen aus unzähligen Gattern, die - miteinander verbunden - sogenannte Schaltfunktionen realisieren. Mit Hilfe der Boolschen Algebra wurde gezeigt, wie solche Schaltfunktionen eindeutig dargestellt werden können. Außerdem wurden Techniken vorgestellt, mit denen man die Kosten einer Schaltfunktion, d.h. die Anzahl der benötigten Gatter, minimieren kann.

Die wichtigste Komponente eines Computers ist die CPU. Diese enthält neben der Steuereinheit die Arithmetisch-Logische Einheit, welche für die Rechenoperationen zuständig ist. Verschieden Verfahren, mit denen die ALU Grundrechenoperationen durchführt, wurden erläutert. Dabei wurde insbesondere gezeigt, daß im allgemeinen ein Trade-Off zwischen der Schnelligkeit von Rechenoperationen und ihrem Implementierungsaufwand besteht.

6. Literatur

Hayes J. P.: "Computer Architecture and Organisation", McGraw-Hill, 1988

Hennessy J. L., Patterson D. A.: "Rechnerarchitektur", Vieweg, 1994

Oberschelp W., Vossen G.:"Rechneraufbau und Rechnerstrukturen", Oldenbourg Verlag, 1993

Spaniol O.: "Arithmetik in Rechenanlagen", Teubner Studienbücher, 1976

7. Index