Informatik I und II für Ingenieure

Skript zur Vorlesung

Organisation von Berechnungsabläufen und Rechnern
Jahrgang 2002/2003
Zuletzt aktualisiert: 13.04.2003

Gerald Sommer (gs@ks.informatik.uni-kiel.de)

Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik

Vorwort

Dieses Skript hat eine Geschichte, die eng gekoppelt ist an die Geschichte der Vorlesung "Informatik für Ingenieure". Im Wintersemester 1996/97 wurde diese spezielle Vorlesung für Ingenieure neu eingeführt. Folgende Dozenten haben seitdem diese Vorlesung gehalten:

Informatik I	WS 1996/97	G. Sommer
Informatik II	SS 1997	G. Sommer
Informatik I	WS 1997/98	G. Sommer
Informatik II	SS 1998	P. Kandzia
Informatik I	WS 1998/99	G. Sommer
Informatik II	SS 1999	G. Sommer
Informatik I	WS 1999/2000	G. Sommer
Informatik II	SS 2000	G. Sommer
Informatik I	WS 2000/01	R. Koch
Informatik II	SS 2001	R. Koch
Informatik I	WS 2001/02	R. Koch
Informatik II	SS 2002	R. Koch
Informatik I	WS 2002/03	R. v. Hanxleden
Informatik II	SS 2003	G. Sommer

Wie alle gedruckten Werke von größerem Umfang war das Skript nie frei von Schreib- und orthografischen Fehlern. Wir haben stets an deren Beseitigung gearbeitet. Viel wesentlicher ist aber, dass mit der Zeit auch inhaltliche Korrekturen und Verschiebungen der Vorlesung erfolgten, die sich im Skript niederschlugen. Ich danke allen an der Vorlesung beteiligten Dozenten, wissenschaftlichen Mitarbeitern und Studenten für diese Verbesserungen. Sie tragen dazu bei, dass auch heute das vorliegende Skript lebt und stets offen ist für Korrekturen und für Vorschläge zur weiteren Optimierung des Stoffes.

Gerald Sommer April 2003

Inhaltsverzeichnis

4 Vom Programm zur Maschine

In diesem Kapitel werden wir lernen, wie sichergestellt wird, daß die in einer höheren Programmiersprache formulierte Problemlösung rechnerseitig realisiert wird.
Die Betrachtung erfolgt

- bezüglich der Architektur eines Rechners nach dem von Neumann-Prinzip und
- bezüglich der Organisation von Berechnungsabläufen auf einer solchen Architektur.

Dabei werden wir uns bis auf die Ebene der Speicherorganisation und der Maschinenbefehle (einschließlich der Varianten von Adressierungen) herabbegeben.
Zwischen der Maschinensprache und höheren Programmiersprachen ist die *Assemblersprache* angesiedelt, die eine Befehlsmnemonik und symbolische Adressierung nutzt.
Mit der Beschreibung der Art und Weise, wie aus einem Assemblerprogramm ein lauffähiges Programm wird, beenden wir auch die globale Sicht der Rechnerorganisation. Das heißt, die mit dem Betriebssystem bereitgestellten Ressourcen zur Programmabarbeitung und die Grundlagen der Programmübersetzung werden nicht behandelt.

4.1 Rechnerorganisation und Rechnerarchitektur

4.1.1 Die Ansichtsweisen eines Rechners

Die Problemlösung auf einem Rechner stellt ein komplexes Organisationsproblem dar:

- es findet auf unterschiedlichen Ebenen statt (virtuelle Maschinen)
- es hat unterschiedliche Facetten, je nach der Sicht auf das Problem (Systemsicht, Programmiersicht, Gerätesicht)
- es nutzt unterschiedliche Ressourcen (Gerätesicht: Komponenten der Rechnerarchitektur, Systemsicht: Betriebssystem)

a) Rechnerorganisation

Die *Rechnerorganisation* befaßt sich mit der Realisierung von Berechnungsvorgängen auf zwei Ebenen.

1. *Programmausführungsebene*:
   Wie wird ein Anwenderprogramm (einschließlich Daten) über eine Folge von Zwischendarstellungen in eine Problemlösung überführt.
   - Agenten (der Operationen auf Objekten) sind Operatoren
   - Kanäle (des Transportes von Objekten zwischen Agenten) sind Speicherbereiche

   Entsprechend dem konkret angewendeten Programmier-Paradigma ist von Interesse,
   - wie Operanden an Operatoren zugeführt werden (*operationaler Aspekt*)
   - wie Operatoren aktiviert werden im Sinne der logischen Programmstruktur (*Kontrollaspekt*).
2. *Betriebssystemebene*:
   Wie werden Ablauf und Ressourcen der Prozesse der Programmausführungsebene verwaltet und geplant und wie wird die Funktionalität des Gesamtsystems sichergestellt.
   - Agenten sind Scheduler (Ablaufplaner), ihre Aufgabe ist die Steuerung und Überwachung der Prozeßabläufe
   - Kanäle sind Datenstrukturen, sie dienen der Darstellung von Prozeß- und Systemzuständen mittels Listen, Tabellen, Stacks, ...

b) Rechnerarchitektur

Die *Rechnerarchitektur* befaßt sich mit den elementaren Komponenten und Operationen zur Realisierung eines Berechnungsmodelles (einschließlich des Betriebssystemmodelles). Sie bestimmt somit das Erscheinungsbild der Ressourcen eines Rechners auf der untersten Ebene, die einem Programmierer zugänglich ist. Das Erscheinungsbild ist eine von der technischen Realisierung abstrahierte Systembeschreibung, die umfaßt:

- Instruktionssatz
- Adressierungsmodes
- elementare Datentypen
- Betriebsmittel und Zustandsbeschreibungen, die durch Nutzung von Instruktionen und Adressierungen zugänglich werden.

Architekturen definieren Systemfamilien. Die Modelle (Ausführungen) einer Systemfamilie sind in ihrem operationalen Verhalten äquivalent bzw. zumindest aufwärtskompatibel (Beispiel: SPARC-Prozessor). Aber nur, wenn sie sich auch der gleichen Rechnerorganisation bedienen, sind sie äquivalent bezüglich der Realisierung einer Aufgabe.

Forderung (nicht immer voll zu erfüllen):

Unabhängig von der Rechnerorganisation liefern die Realisierungen einer Rechnerarchitektur die gleichen Ergebnisse für eine Aufgabe. (Bsp.: Betriebssysteme für PCs: DOS, Windows, Linux)

c) Rechnertechnik (Computer Engineering)

Die *Rechnertechnik* befaßt sich mit der technischen Realisierung der Modelle einer Architekturfamilie. Hierbei sind von Interesse für Entwurfsentscheidungen:

- Leistungs- und Zuverlässigkeitsanforderungen
- technologische Randbedingungen
- Produktionsaufwendungen (Kosten).

Daraus abgeleitet werden die Details für Software-, Firmware- und Hardware-Implementierungen.

Folgende drei Ansichtsweisen sind für das Verständnis der Funktion eines Rechners zu unterscheiden:

1. Rechner als programmierte Maschine: Systemsicht des Nutzers
2. Rechner als programmierbare Maschine: Systemsicht des Programmierers
3. Rechner als komplexe materielle Maschine: Gerätesicht des Entwerfers, Programmierers, Nutzers

Aus der Sicht der Rechnerorganisation besteht ein Rechner aus mehreren abstrakten Maschinen (engines):

- eine oder mehrere *processing engines* (PEs) zur Ausführung benutzerspezifizierter Programme
- eine oder mehrere *input/output engines* (IOEs) zum Datentransfer mit peripheren Geräten bzw. für die Kommunikation in Netzen
- einer *operating system engine* (OSE) zur Überwachung und Steuerung der PEs und IOEs und zur Verwaltung deren Ressourcen.

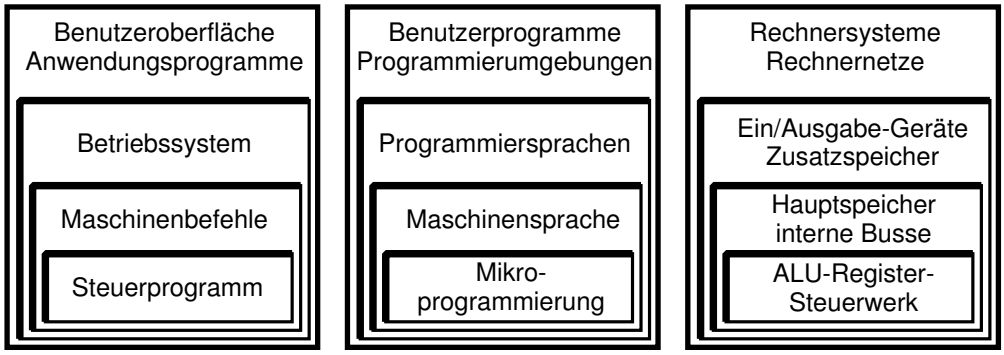


Abbildung 4.1: Drei Sichten des Rechners: Die Systemsicht, die Programmiersicht und die Gerätesicht

4.1.2 Der von Neumann-Rechner

Die wesentlichen Architektur- und Organisationsprinzipien fast aller heute verfügbaren Rechner gehen auf Vorschläge des ungarischen Mathematikers John von Neumann zurück (1946) (gemeinsam entwickelt mit Burks, Goldstine).

4.1.2.1 Prinzipien des von Neumann-Rechners

a) Physische Ressourcen

1. *Zentraleinheit* (*CPU*) mit einem oder mehreren Instruktionsprozessoren (IPUs), die sowohl PE- als auch OSE-Programme ausführen können; die CPU arbeitet taktgesteuert sequentiell.
2. Direkt adressierbares Speichermedium (*Arbeitsspeicher*), in dem Programmcode und Daten gehalten werden.
3. Ein oder mehrere *Ein-/Ausgabe-Prozessoren* (IOPUs) , die IOE-Programme ausführen können.
4. Ein oder mehrere *Bussysteme*, über die die Komponenten miteinander kommunizieren können.

b) Universalität

Die Struktur des Rechners ist unabhängig vom speziell zu bearbeitenden Problem. Für jedes Problem steht ein eigenes Programm, deshalb wird auch die Bezeichnung "programmgesteuerter Universalrechner" verwendet.

Biologische Systeme stehen dazu im Gegensatz: Die Struktur des Systems ist eine Verkörperlichung des Problems (andere Probleme erfordern andere Strukturen).

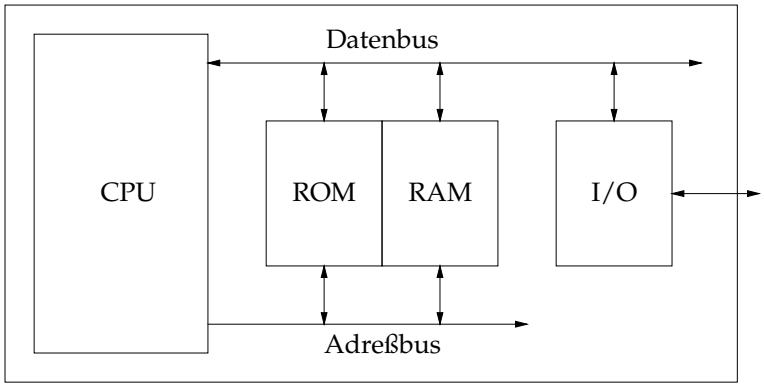


Abbildung 4.2: Struktur eines von Neumann-Rechners

c) Operationsprinzip

Das Operationsprinzip einer Rechnerarchitektur ist gekennzeichnet durch

1. verfügbare Maschinen-Datentypen:
   Sie definieren die Informationsstrukturen auf der Maschinenebene und die darauf ausführbaren Operationen.
2. realisierte Kontrollstruktur:
   Sie steuert die zeitliche Abfolge der Operationen auf der Informationsstruktur und überwacht den Datenzugriff.

Die Informationsstrukturen der von Neumann-Maschine werden wesentlich durch das realisierte Speicherkonzept bestimmt.

i) Speicherkonzept
   - Der Speicher ist sequentiell geordnet.
   - Der Speicher besteht aus Einheiten fester Wortlänge, die über ihre Adresse direkt angesprochen werden können.
   - Programme und Daten werden in demselben Speicher abgelegt.
   - Die binäre Repräsentation von Programm und Daten trägt keine Kennzeichnung, die den Inhalt einer Speicherzelle als Teil eines Programms oder als Datum identifiziert.

ii) Semantische Lücke
   Hieraus folgt ein Problem, das als *semantische Lücke* bezeichnet wird. Auf der Anwenderprogramm-Ebene ist die elementare Einheit einer Zustandsbeschreibung die Programm-Variable mit der Informationsstruktur

$$\sigma_{P.V.} = (Identifikator, Wert, Typ).$$

Damit ist der Inhalt einer Speichereinheit als Variable eines bestimmten Namens und Typs semantisch eindeutig interpretierbar.
Auf der Maschinenebene der von Neumann-Architektur existiert hingegen eine Maschinen-Variable als Datenobjekt mit der Informationsstruktur

$$\sigma_{M.V.} = (Identifikator, Wert).$$

Dabei stehen *Identifikator* für eine logische oder physische Adresse auf dem Speicher und *Wert* für den Inhalt der Speicherzelle.
Die Transformationsvorschrift zur semantischen Interpretation ist selbst nicht codiert. Diesen Sachverhalt bezeichnet man als semantische Lücke.
Ein ähnliches Phänomen existiert im Gehirn: Neuronale Signale haben alle ein ähnliches Codierungsprinzip, so daß einer neuronalen Erregung nicht zu entnehmen ist, was sie codiert.
Lösung:

1. Nutzung des räumlichen Kontextes: Z.B. visuelle und Schallsignale werden an getrennten, wohl definierten Orten verarbeitet.
2. Nutzung des zeitlichen Kontextes: Einer Folge ähnlicher Erregungsmuster an unterschiedlichen Orten wird eine bestimmte Ursache zugeschrieben (diese Folge steht für etwas mit Bedeutung).

Die Interpretation eines Speicherwortes einer von Neumann-Maschine erfolgt nur aufgrund des Maschinenzustandes (durch die realisierte Kontrollstruktur).

iii) von Neumann-Maschinen-Datentypen
Die Datenobjekte (von Neumann-Variable) werden durch die Datentypen

- Befehl (bzw. Instruktion)
- Datum
- Adresse

repräsentiert.
Die auf Maschinenebene ausführbaren Operationen beziehen sich also auf Operanden dieser Typen. Maschinen-Operationen sind die auf Maschinenebene durch den Prozessor ausführbaren Operationen.
Das Ziel der Maschinen-Operationen besteht darin, eine Zustandsänderung zu bewirken, die semantisch äquivalent ist der logischen Struktur des Anwenderprogrammes.
Hierzu werden Speicherzell-Inhalte vom Typ Datum bzw. Adresse geändert. Dies geschieht in der *Execution-Phase* einer Operation über dem Typ Befehl. Daten und Adressen sind die Operanden der Operation "Befehl ausführen". Damit ein Befehl ausgeführt werden kann, ist der Abschluß der *Fetch-Phase* erforderlich, die der Operation "Befehl vorbereiten" entspricht.

iv) von Neumann-Kontrollstruktur
Die Kontrollstruktur der von Neumann-Maschine folgt einem *Zwei-Phasenkonzept*, indem sie bei der Befehlsverarbeitung ständig zwischen *Befehlszustand* und *Datumszustand* umschaltet:

1. *Fetch-Phase*: Befehlszustand der Maschine
Operation: "Befehl vorbereiten"
Teiloperationen:
   1. Befehl aus dem Speicher holen
   2. Befehlszähler erhöhen
   3. Befehl decodieren und Adresse des zugehörigen Datums berechnen
2. *Execution-Phase*: Datumszustand der Maschine
Operation: "Befehl ausführen"
Teiloperationen:
   1. Datum/Adresse aus dem Speicher holen
   2. Datum/Adresse verarbeiten/verändern
   3. Datum/Adresse in den Speicher zurückschreiben

Dies setzt voraus:

1. Eine von Neumann-Maschine bearbeitet zu jedem Zeitpunkt genau einen Befehl oder ein Datum (SISD-Prinzip, SISD=Single Instruction-Single Data).
2. Die sequentielle Speicherordnung der Programmbefehle entspricht der Reihenfolge ihrer Verarbeitung.

Die lineare Abarbeitung der Programmanweisungen auf Maschinenebene entspricht etwa der linearen Strukturierung des Anwenderprogramms bei imperativer Programmierung. Wie dort, wird auch auf Maschinenebene die lineare Abarbeitung unterbrochen durch

- Sprunganweisungen
- zustandsbedingte Verzweigungen (if-Anweisungen)
- Wiederholungen (while-Anweisungen)
- (rekursive) Funktionsaufrufe
- Unterprogrammaufrufe.

Diese zeitweilige Abweichung von der streng linearen Abfolge der Befehlsverarbeitung wird durch Sprungbefehle der Maschinensprache realisiert, die den Befehlszähler entsprechend inkrementieren/dekrementieren.

Die Menge der zu einem Programm gehörenden Daten ist nicht geordnet. Vielmehr treten die Adressen der Operanden von Befehlen in diesen selbst auf. Dies

sieht man am prinzipiellen Aufbau eines Maschinenbefehls (werttransformierende Instruktion):

$$\texttt{<val\_instr>::=<val\_rator>}\{\texttt{<rand>}\}^n\texttt{<dest>}$$

mit

<val_rator>	:	Operator der Maschinensprache
$\{$<rand>$\}^n$	:	Quellen von n Operanden
<dest>	:	Ziel des Resultatwertes

Dennoch macht es Sinn, aus programm-organisatorischen Gründen eine systematische Strukturierung des Speichers vorzunehmen (Laufzeitumgebung und Aktivierungsrekords).

4.1.2.2 Struktur und Arbeitsweise der Zentraleineit (CPU)

Die *Zentraleinheit* besteht aus

- dem *Prozessor*
- dem *Speicher*.

Der Prozessor besteht aus

- dem *Rechenwerk* (operational unit)
- dem *Steuerwerk* (control unit) .

Abbildung 4.3 zeigt den schematischen Aufbau einer Zentraleinheit.
Die Aufgaben der CPU sind:

(1) "klassische" Datenverarbeitung durch das Rechenwerk

(2) Befehlsinterpretation und Ablaufsteuerung durch das Steuerwerk

(3) Unterstützung des Prozeß-Scheduling auf der Ebene des Betriebssystems (OSE) durch das *Process-Controlblock-Register* (PCB) und die *Unterbrechungseinheit* (interupt unit IU).

Vereinfacht erfüllt der Prozessor die Aufgaben (1) und (2) nach folgendem Schema:

a) Fetch-Phase (Befehlszustand)

1. Ermitteln der Adresse des nächsten auszuführenden Befehls durch das Steuerwerk.
   Mitteilung dieser Adresse an den Speicher über den Adreßbus.
2. Auslesen des Befehls aus dem Speicher über den Datenbus.
   Übertragen des Befehls an das Steuerwerk.
   Decodieren des Befehls.
   Ausführen weiterer Operationen durch das Steuerwerk in Abhänigkeit vom Befehl.
   Umschalten des Steuerwerks auf den Datumszustand.

b) Execution-Phase (Datumszustand)

3. Berechnen der Adresse des Operanden.
   Mitteilen der Adresse an den Speicher.
   Bereitstellen des Operanden auf dem Datenbus.
4. Ausführen der Instruktion durch das Rechenwerk unter der Kontrolle des Steuerwerkes. Dabei steht der zweite Operand in einem Register bereit.
   Das Ergebnis wird an ein Register geliefert oder an eine Speicheradresse nach dem Schema:
   Berechnen der Speicheradresse für das Ergebnis und Mitteilung an den Speicher.
   Ablegen des Ergebnisses über den Datenbus im Speicher.
5. Das Steuerwerk schaltet auf den Befehlszustand (Sprung nach 1.).

(1) Das Rechenwerk

Hauptaufgabe: Verknüpfung von Operanden mit arithmetischen und logischen Operationen

Hauptkomponenten:

1. Arbeitsregister (R_0,. . .,R_n)
2. Arithmetisch-Logische Einheit (ALU)
3. Statusregister (SR)

1.) Arbeitregister: z.B. n=16

- besonders schnelle Speicher im Wortformat mit bitweisem/byteweisem Zugriff
- heute meist Allzweckregister zum Aufnehmen der Operanden und Zwischenspeicherung

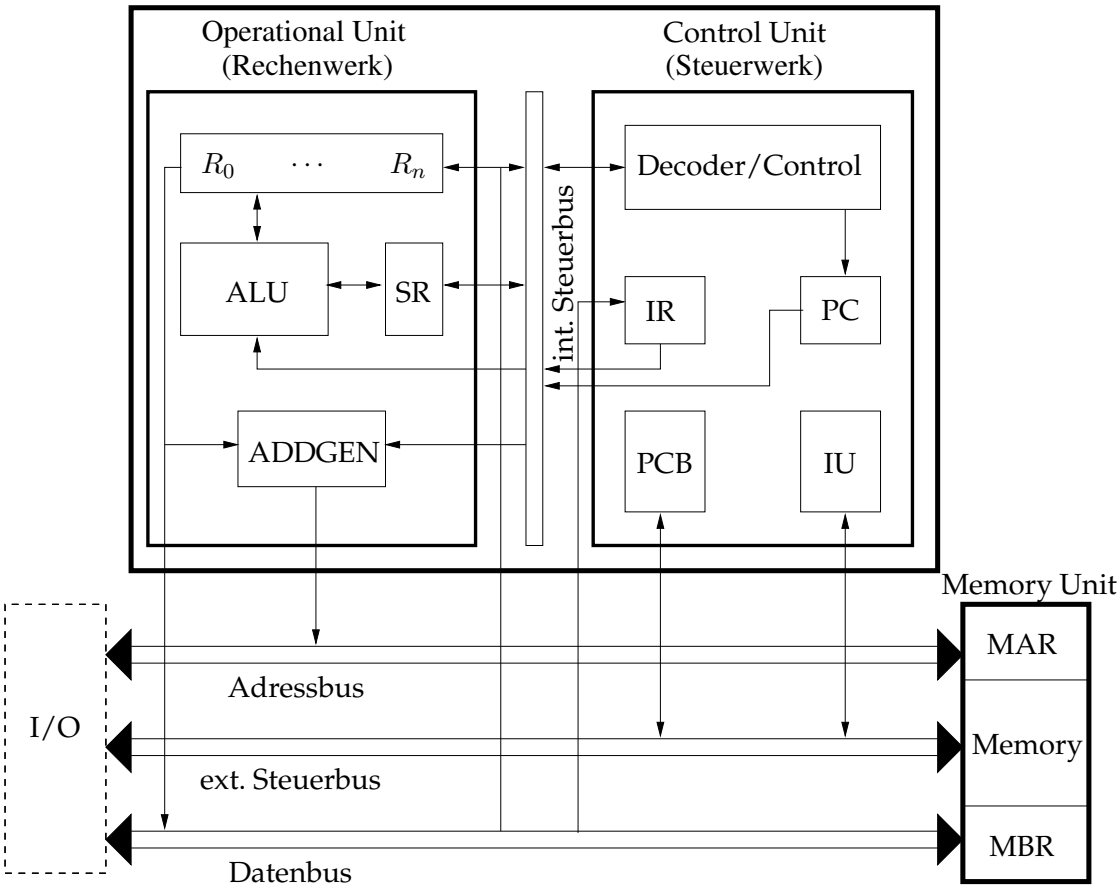


Abbildung 4.3: Aufbau einer CPU

- aber auch spezielle Aufgaben:
  Akkumulator (früher): ein Operand bzw. Ergebnis
  Adreßregister, Indexregister zur Adreßmodifikation des Operanden (des Ergebnisses). In unserem Modell erfolgt die Adreßberechnung durch die Adressierungseinheit ADDGEN.
  Stackregister TOS
  Instruktionsregister IR

2.) Verknüpfungslogik (ALU)

- führt Operationen über Operanden aus
- typische Operationen:
  - Transport-Operationen (Laden, Speichern / `LD, ST`)

- Boolesche-Operationen (UND, ODER / `AND, OR`)
- Schiebe-Operationen (Links-/Rechts-Schieben / `LSH, RSH`)
- Register-Manipulationen (`INC, DEC, KPL`)
- arithmetische Operationen (Addition, Subtraktion / `ADD, SUB`)
  (Multiplikation und Division werden unter Kontrolle des Steuerwerks durch Additionen und Schiebeoperationen realisiert)

3.) Statusregister (SR)
Das Statusregister verwaltet den tatsächlichen Status der ALU. Als Status wird der Zustand des Rechenwerkes nach der Ausführung von Operationen im Zweiphasenschema des Steuerwerkes bezeichnet.
Im Statusregister werden die Statusbedingungen nach der Durchführung arithmetischer Operationen festgehalten, dies sind z.B.

zero	– Resultat ergab Null
sign	– Resultat ergab negativen Wert
carry	– ein Übertrag war notwendig
overflow	– der Zahlenbereich wurde überschritten

Das Eintreten der Statusbedingung wird durch das Setzen der entsprechenden Bits im Statusregister auf Eins angezeigt.

4.) Adressierungseinheit (ADDGEN)
Hier erfolgt (soweit nicht bereits durch die ALU vorgenommen) die Berechnung von Adressen für Befehle und Operanden.
Nach der Decodierung des Modifikationsteils der Adresse im Befehl erfolgt die Adreßberechnung in Abhängigkeit der Adreßspezifikation (Registermode, Speichermode).

(2) Das Steuerwerk

Hauptaufgabe: Entschlüsselung von Befehlen und die Steuerung ihrer Ausführung
aber auch: Verkehr mit Speicher, (Peripherie,) interner Ablauf
Hauptkomponenten:

1. Decodier- und Steuerungseinheit
2. Instruktionszähler (PC)
3. Instruktionsregister (IR)
4. Process Controll Block Register (PCB)

5. Unterbrechungseinheit (IU)

Der Instruktions- oder Befehlszähler wird nach dem Lesen jedes Befehls entweder um eins erhöht (wenn der Befehl kein Sprungbefehl ist) oder auf die Sprungadresse gesetzt (die Rücksprungadresse wird u.U. gemerkt). Der Befehlszähler dient der Berechnung der Adresse des nächsten auszuführenden Befehls oder gibt diese direkt an.

Im Instruktionsregister wird der aus dem Speicher geholte Befehl gespeichert. Von dort wird er über den internen Steuerbus an die Decodier- und Steuerungseinheit weitergegeben.

Im Process Controll Block Register wird der Status der Programmausführung gehalten, z.B. Unterbrechbarkeit, Priorität, laufender Modus (z.B. Unterprogramm- oder Funktionsmodus).

Durch die Unterbrechungseinheit erfolgt die Steuerung der Programmunterbrechung durch Interrupts (z.B. durch Peripherie) und das Retten aktueller bzw. Laden neuer Registerinhalte. Weiterhin werden der Status und die Identität des unterbrechenden Peripheriegerätes ausgewertet.

Die Befehle der Fetch- und Execution-Phase entsprachen in den Prozessoren früherer Architekturkonzepte ca. 10 - 100 Operationen (*Mikrobefehle*) der Steuereinheit. Gegenwertige und künftige Architekturkonzepte kehren aber das Verhältnis der Instruktionen pro Maschinentakt um (siehe Abbildung 4.4). Dies wird nur durch prozessorinterne Parallelisierung erreicht, also Abweichung vom Konzept der von Neumann-Architektur.

Das Mißverhältnis zwischen dem semantisch bedeutsamen Anteil der Operationen und dem für Lade- und Transportaufgaben erforderlichen Aufwand heißt *von Neumann-Flaschenhals*.

Die Mikrobefehle werden entweder fest verdrahtet erzeugt oder einem Mikroprogrammspeicher entnommen.

(3) Der Speicher

Direkt adressierbarer Halbleiterspeicher

Hauptkomponenten:

1. Speicher-Adreßregister (MAR)
2. Speicher-Puffer-Register (MBR)
3. Bus-Leitungen
4. Arbeitspeicher (Memory)

1.) Speicher-Adreßregister (MAR):
Das MAR enthält die Adresse der Speicherzelle in die geschrieben/von der gelesen wird.

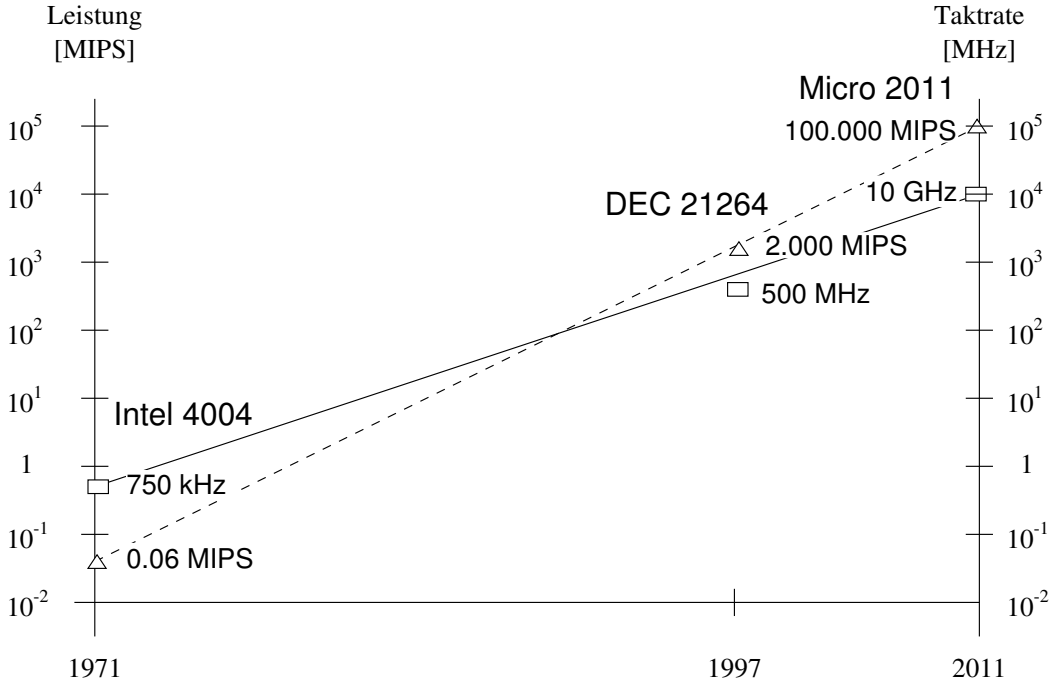


Abbildung 4.4: Rechenleistung und Taktrate von Mikroprozessoren

2.) Speicher-Pufferregister (MBR):
Das MBR nimmt Daten beim Lesen/Schreiben auf (bis Doppelwort Breite)

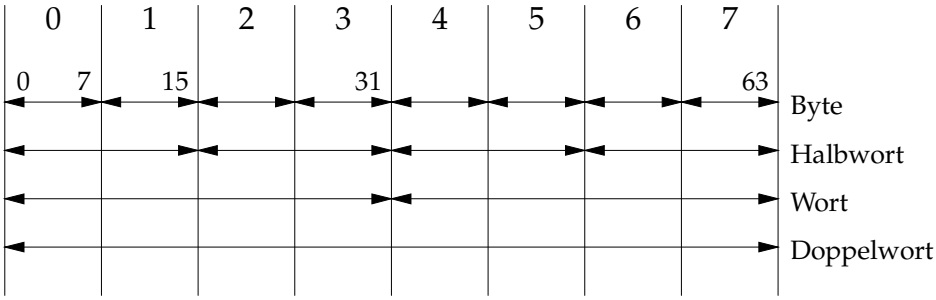
3.) Bus-Leitungen:
für Daten, Adressen, Status
Auf den Busleitungen erfolgen bitparallele Transporte, so umfaßt der Adreßbus z.B. k Leitungen.

4.) Arbeitsspeicher
Der Arbeitsspeicher ist wortweise organisiert, d.h. die Adressen haben Wortformat.
Der Adreßbereich ist durch das Intervall $[0 \ldots 2^k - 1]$ gegeben, z.B. k =16, 32 (z.B. SPARC) oder k =64 (z.B. ULTRASPARC). $k = 32$ entspricht einem maximalen Adreßbereich von 4 GByte.
In der Regel ist ein Rechner mit weniger Speicher bestückt, als der Adreßbereich zuläßt. So ist real nur der physikalische Adreßraum $[0 \ldots 2^n - 1]$ adressierbar, z.B. $n \in [24 \ldots 28]$: 16...256 MByte. Es ist heute keine Seltenheit mehr, über einen physikalischen Adreßbereich von 1 GByte zu verfügen.

Die Platzierung von Daten im Adreßraum erfolgt Byte-orientiert.

Beispiel: Platzierung von Daten im Adreßraum

32-Bit-Maschine:



Oft ist der logische Adreßraum (vom Programm benötigter Speicherbereich) größer als der physikalische Adreßraum. Abhilfe wird durch die *virtuelle Adressierung* geschaffen. Durch das Betriebssystem erfolgt die Speicherverwaltung (MMU), der Hintergrundspeicher (Plattenspeicher) wird seitenorientiert (z.B. Seitengröße 4 KByte) auf den Hauptspeicher abgebildet (s. Abb 4.5).
Außerdem erfolgt eine Unterteilung des logischen Adreßraums in Segmente, die

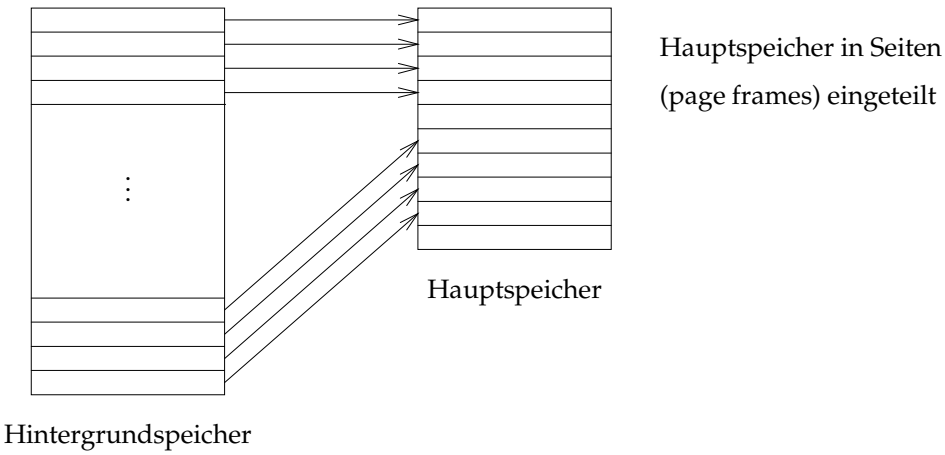


Abbildung 4.5: Abbildung des Hintergrundspeichers auf den Hauptspeicher

mehrere Seiten umfassen. Von einem Prozeß benötigte Segmente können seiten-

weise in den Hauptspeicher geladen oder in den Hintergrundspeicher ausgelagert werden.
Somit ergibt sich eine *virtuelle Adresse* aus dem Tripel
(Segment-Nummer, Seiten-Nummer, Seiten-Offset).

Bezüglich der Geschwindigkeit/des Speichervolumens ergibt sich die in Abbildung 4.6 dargestellte Speicherhierarchie.

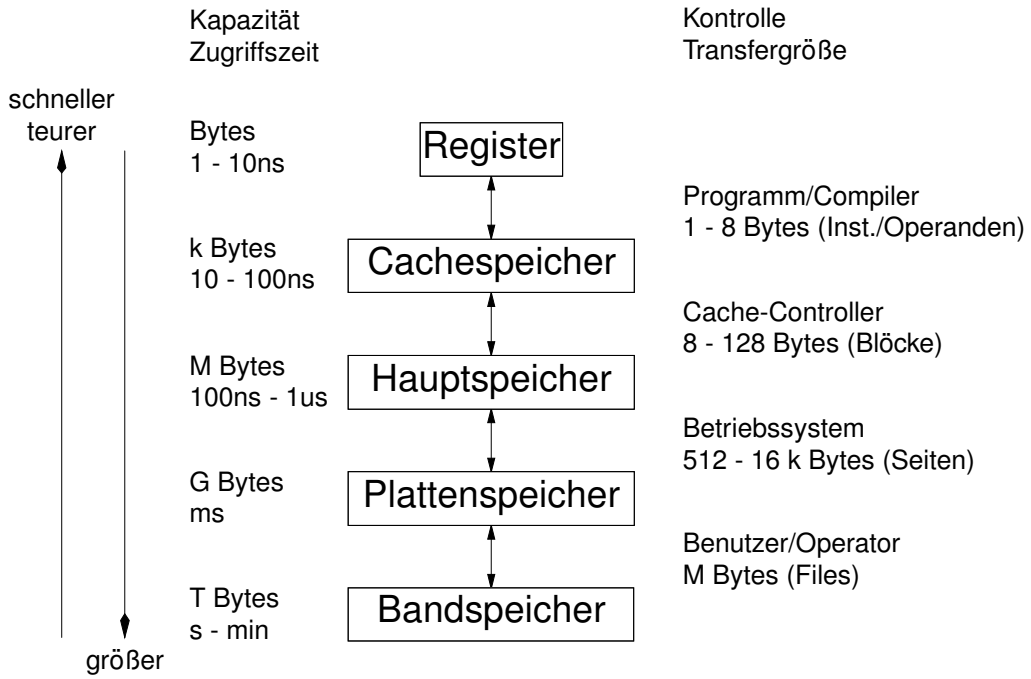


Abbildung 4.6: Hierarchie der Speicherkonzepte

Moderne Fertigungstechnologien von Prozessorchips beruhen auf homogenen Strukturentwürfen. Deshalb wird der größte Anteil von Transistoren auf einem Prozessorchip für die Realisierung von Speicher, d.h. Register und Cache eingesetzt. Der DEC-Prozessor 21264 z.B. verwendet von seinen 15.2 Mio Transistoren nur 6 Mio für die CPU-Logik.

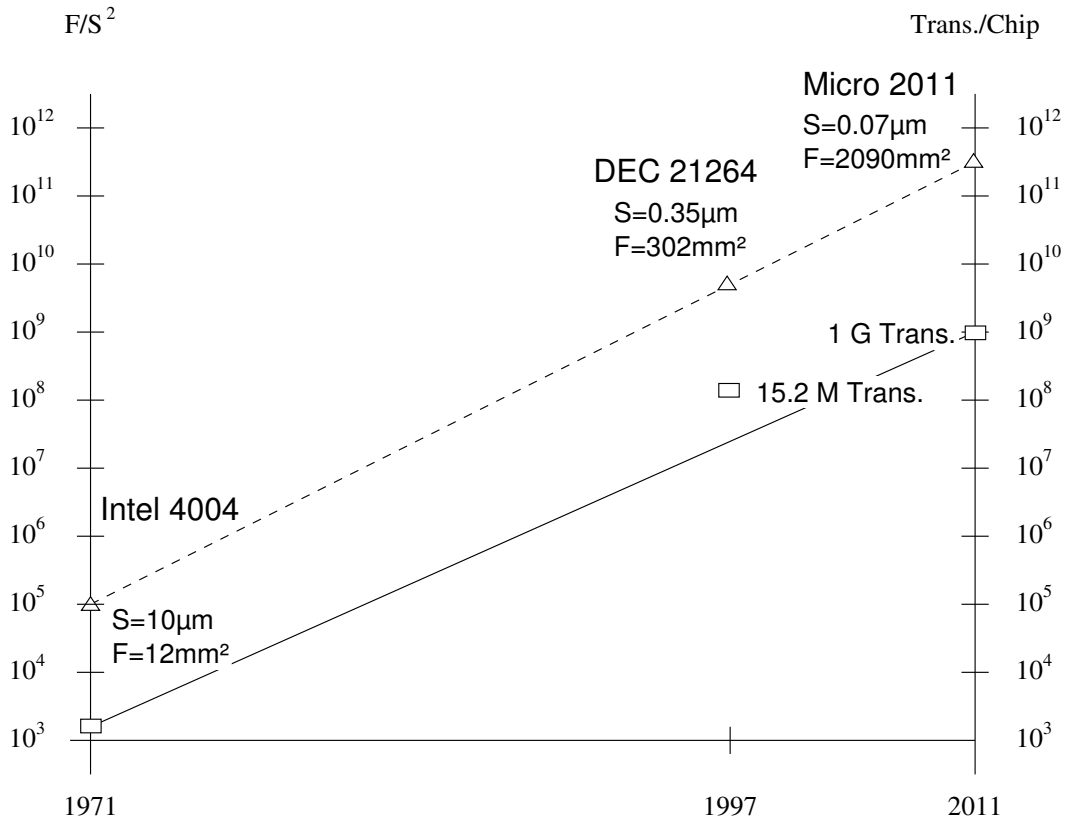
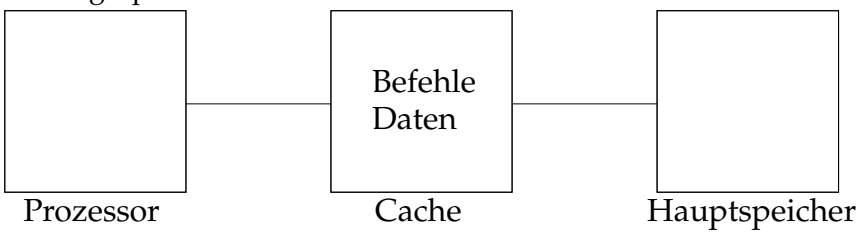


Abbildung 4.7: Integrationsdichte (Strukturgröße S, Chipfläche F)

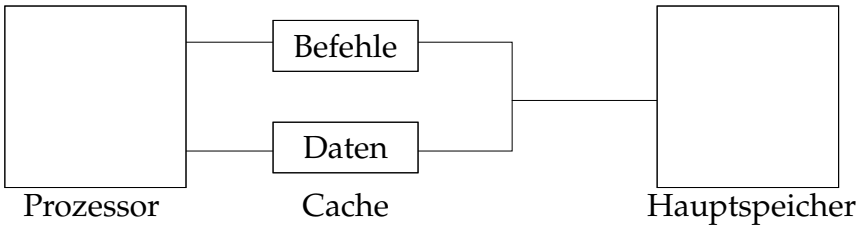
Architekturkonzepte von Cachespeichern:

1. *Princeton-Architektur*: Befehle und Daten werden in einem gemeinsamen Cache gespeichert.



   Vorteil: flexible Zuordnung Speicherplatz

2. *Harvard-Architektur*: Befehle und Daten werden in separaten Caches gespeichert.



Vorteil: gleichzeitiger Zugriff auf Befehle und Daten

Pseudocode des Zwei-Phasenkonzeptes einer von Neumann-Maschine

Annahmen: Ausführung einer triadischen (Drei-Adreß-) wertransformierenden Instruktion: 1. Operand in Register RI, 2. Operand im Speicher und Resultatspezifikation (Register RJ)

a) Befehlszustand des Steuerwerkes

1. ```
   ADDGEN:=PC;
   MAR:=ADDGEN; /* vom Adreßgenerator erzeugte Adresse */
   ```
2. ```
   MBR:= <M>; /* Inhalt der durch MAR angegebenen Speicherzelle
   */
   IR:=MBR;
   decodiere IR;
   if kein Sprungbefehl then
     PC:=PC+1; goto 3;
   else
     PC:=Sprungzieladresse; goto 1;
   endif
   ```

b) Datumszustand des Steuerwerkes

3. ```
   ADDGEN:=Adreßteil des Operanden; /* vom Decoder */
   MAR:=ADDGEN;
   MBR:=<M>;
   ```
4. ```
   RJ:=OP(MBR,RI);
   MAR:=ADDGEN; /* ADDGEN hält Zieladresse */
   MBR:=RJ;
   <M>:=MBR;
   ```
5. ```
   goto 1;
   ```
   ```

4.2 Organisation des Programm-Ablaufs

In diesem Abschnitt werden auf der Grundlage der Kenntnis von Grundprinzipien einer von Neumann-Architektur die Grundprinzipien der Abarbeitung einer in einer imperativen Programmiersprache codierten Problemlösung auf Maschinenebene behandelt. Hierzu ist zu klären,

- wie die Adressierung von Operanden erfolgt (Erinnerung: die Adressierung der Operatoren erfolgt durch den Befehlszähler),
- wie die Verwendung der Ressource Speicherplatz erfolgt,
- welche statischen (durch den Compiler angelegten) Organisationsstrukturen hierfür genutzt werden,
- welche dynamischen (während der Laufzeit der Programmabarbeitung angelegten) Organisationsstrukturen hierfür genutzt werden,
- welcher Befehlssatz auf Maschinenebene verfügbar ist und
- wie in Befehlen Operanden und Operationen codiert sind.

4.2.1 Die Berechnung von Ausdrücken

Zunächst soll skizziert werden, in welcher Weise eine in einer höheren Programmiersprache formulierte Problemlösung durch den Übersetzungsvorgang eines Compilers auf Maschinenebene angeboten wird.

Ein Prozessor ist als reaktives System aufzufassen:
Er verfügt über keinerlei Kontrollmechanismen um die semantische Adäquatheit einer Problemlösung sicherzustellen. Vielmehr wird “blind” jeder vom Programmzähler adressierte Befehl ausgeführt. Lediglich zur Steuerung dieser Befehlsausführung besitzt der Prozessor Mechanismen (z.B. Mikroprogramme).

Nur der Compiler kann die Semantik der Problemlösung in eine adäquate Verkettung von Maschinenbefehlen umsetzen, wobei diese Verkettung den Programmzähler steuert.

Prinzipiell hat ein Compiler die Aufgabe, komplexe Ausdrücke (Terme) und Anweisungen sowie Unterprogrammaufrufe in eine Folge von Maschinenbefehlen zu codieren, welche null-, ein- und zweistelligen elementaren Termen entsprechen.

Dabei muß die Vorrangregel "früher zu berechnen als" berücksichtigt werden. Wie aus Abschnitt 3.2.1 bekannt ist, stellt der Kantorovič-Baum den erzeugenden Graphen für diese partielle Ordnung dar.

Da der Kantorovič-Baum eine nicht-lineare Struktur ist, muß er topologisch äquivalent linearisiert werden.

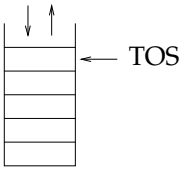
Compiler nutzen die Tatsache, daß die Abarbeitung eines als Kantorovič-Baum codierten Terms das Aufbrechen in eine adäquate Folge von elementaren Termen liefert. Außerdem entspricht dieses Aufbrechen einer Abarbeitung des Terms nach dem Kellerpinzip.

In diesem Abschnitt sollen verschiedene Möglichkeiten der Abbildungen eines Terms in eine Folge von elementaren Termen demonstriert werden. Dabei hat der Compiler unterschiedliche Ressourcen der Maschine zu berücksichtigen.

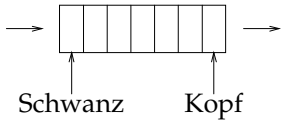
Hier wird also lediglich die Folge der elementaren Terme betrachtet, ohne ihre Realisierung als Maschinenbefehle zu berücksichtigen.

In Abschnitt 3.3.2 (Rekursion) wurde das Prinzip der Stack-/Stapel-/Kellerverarbeitung eingeführt. Ein Stack ist eine Datenstruktur mit den Operationen

- push: oben auflegen
- pop: von oben entfernen.



Ein Stack wird nach dem *LIFO-Prinzip* (Last-In, First-Out) organisiert.

> **Einschub:** Eine Datenstruktur von ähnlicher Bedeutung stellt eine *Warteschlange* (Queue) dar. Diese wird aber nach dem *FIFO-Prinzip* (First-In, First-Out) organisiert. Deshalb bietet sich ihre Verwendung bevorzugt zur Realisierung der OSE (Betriebssystem) für das Scheduling an.
>
> Die beiden Operationen zur Manipulation dieser Datenstruktur sind
>
> - Löschen (am Kopf)
> - Anfügen (am Schwanz).
>
> 
>

Zunächst soll eine Variante demonstriert werden, bei der vom Compiler die Berechnung eines arithmetischen Ausdruckes unter Verwendung zweier Stapel codiert wird:

linker Stapel - Operanden
rechter Stapel - Operatoren.

Dabei (s. Beispiel M1, Seite 219) wird der Term $t = a + b * c + d$ über die Berechnung von drei Zwischenergebnissen

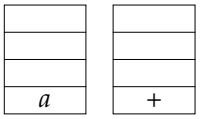
$z_1 := b * c$
$z_2 := a + z_1$
$z_3 := d + z_2$

ausgeführt. Auf Maschinenebene wäre also die Abarbeitung dieser Anweisungsfolge durch eine Codierung in einer Folge von Maschinenbefehlen zu realisieren. Implizit entspricht dies der Berechnung des Terms nach dem Kellerprinzip, ohne daß der Prozessor dies organisieren muß (wurde vom Compiler bei der Codierung realisiert).

Beispiel: M1: Ausdruck $t = a + b * c + d$

Der Pfeil über dem Eingabeterm gibt jeweils die Position des nächsten zu lesenden Zeichens an.

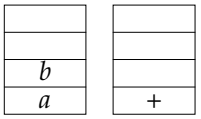
S1:



$a + \overset{\downarrow}{b} * c + d$

PUSH

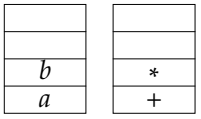
S2:



$a + b \overset{\downarrow}{*} c + d$

Priorität beachten
PUSH

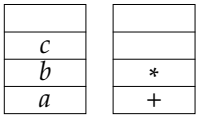
S3:



$a + b * \overset{\downarrow}{c} + d$

PUSH

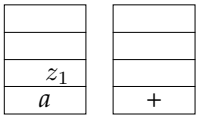
S4:



$a + b * c \overset{\downarrow}{+} d$

PUSH, Prior. beachten
POP
$z_1 := b * c$

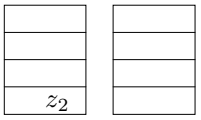
S5:



$a + b * c \overset{\downarrow}{+} d$
$z_1 := b * c$

PUSH, Prior. beachten
POP
$z_2 := a + z_1$

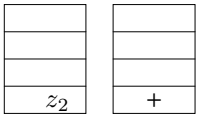
S6:



$a + b * c + d$
$z_2 := a + z_1$

PUSH

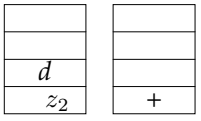
S7:



$a + b * c + \overset{\downarrow}{d}$

PUSH

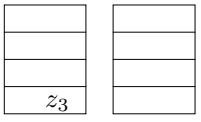
S8:



$a + b * c + d \overset{\downarrow}{}$

Einlesen des Terms beendet
POP
$z_3 := d + z_2$

S9:



$a + b * c + d \overset{\downarrow}{}$
$z_3 := d + z_2$

PUSH
Ausführung des Terms beendet

Wir halten fest:

Kellerprinzip

Von links nach rechts gelesen werden Operationen wenn nötig zurückgestellt und zurückgestellte Operationen werden ausgeführt, sobald möglich.

Aus dem Zwei-Phasenkonzept der von Neumann-Maschine folgt, daß auf Maschinenebene nur Operationen ausgeführt werden können, die höchstens **eine** Operation einer höheren Programmiersprache realisieren. Terme einer Programmiersprache werden also in eine Sequenz

nullstelliger fct $f : ()s_1$
einstelliger fct $f : (s_1)s_2$
oder
zweistelliger fct $f : (s_1, s_2)s_3$

elementarer Terme aufgebrochen.

Im folgenden Beispiel M2 soll die Kellerverarbeitung unter Verwendung lediglich eines Stacks in verschiedenen Varianten demonstriert werden. Der Compiler hat dazu die Ressourcen der Maschine zu berücksichtigen.

Zunächst soll eine Folge von Zwischenergebnissen (Bsp. M2/1) codiert werden, die den Prioritäten der Operationen folgt und äquivalent einem linearisierten Kantorovič-Baum ist.

Die sequentielle Abarbeitung eines Terms kann dadurch sichtbar gemacht werden, daß der Kantorovič-Baum topologieerhaltend in eine Zeile geschrieben wird.
Diese so dargestellte Sequenz spiegelt die *Postfix-Notation* des Terms wider.

Beispiel: M2

s. Seite 221
Für den Term

$$t = \frac{a * b + (a + d) * c}{a + b * b}$$

erhält man als klammerfreie Postfixnotation:

$$t{=}ab{*}ad{+}c{*}{+}abb{*}{+}/$$

Die klammerfreie Postfixnotation wird auch als *umgekehrte polnische Notation* bezeichnet.

Verfahren des Baumdurchlaufes mit Postfixordnung (rekursiv)

Ein Kantorovič-Baum heißt atomar, wenn er nur aus einem Operandenzeichen besteht. Dessen Postfixschreibweise ist das Operandenzeichen.

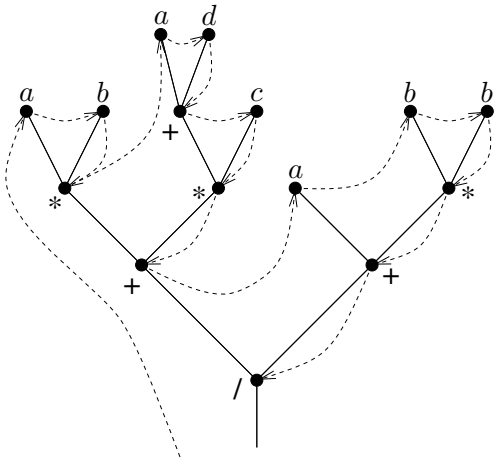
Die Postfixschreibweise eines nicht-atomaren Kantorovič-Baumes ergibt sich aus der Postfixschreibweise all seiner Teilbäume, konkateniert in der Reihenfolge von links nach rechts, gefolgt von dem Operationszeichen an der Wurzel des Baumes.

Im Beispiel M2 treten nur zweistellige Operatoren auf, der zugehörige Kantorovič-Baum ist ein dyadischer Baum. Also ergibt sich als Reihenfolge der Postfixordnung: linker Teilbaum – rechter Teilbaum – Wurzel.
Der linearisierte Kantorovič-Baum in Postfixordnung ist dem Kellerprinzip äquivalent.

Beispiel: M2

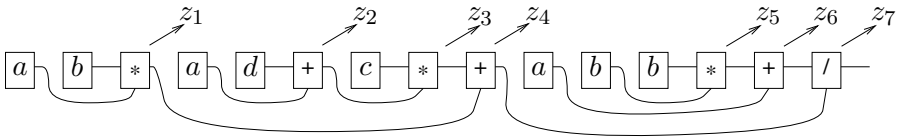
$$t = \frac{a * b + (a + d) * c}{a + b * b}$$

Kantorivič-Baum:



Die gestrichelten Pfeile veranschaulichen das rekursive Verfahren zur Linearisierung des Baumes.

Als linearisierten Baum erhält man:



Zunächst erkennt man aber, daß die Berechnung des Ausdruckes aus Bsp. M2 ähnlich wie in Bsp. M1 über eine Folge von Zwischenergebnissen erfolgt.
Diese Zwischenergebnisse werden jeweils nur einmal verwendet.

Beispiel: M2/1: Sequentialisierung der Zwischenergebnisse

$$
\begin{aligned}
z_1 &:= a * b; \\
z_2 &:= a + d; \\
z_3 &:= z_2 * c; \\
z_4 &:= z_1 + z_3; \\
z_5 &:= b * b; \\
z_6 &:= a + z_5; \\
z_7 &:= z_4 / z_6;
\end{aligned}
$$

Die benutzten Zwischenergebnisse sind stets die zuletzt berechneten und noch nicht benutzten. Sie folgen dem LIFO-Prinzip. Demzufolge bilden die Zwischenergebnisse einen Keller.
Folglich können Zwischenergebnis-Variable gespart werden, wenn *Stack-Variable* verwendet werden.
Dabei gilt für den Index i der Stack-Variablen $h[i]$ in Abhängigkeit davon, wieviele Stack-Variablen auf der rechten Seite der Zuweisung vorkommen:

$i := i + 1$, wenn keine Stack-Variable vorkommt
$i := i$, wenn eine Stack-Variable vorkommt
$i := i - 1$, wenn zwei Stack-Variable vorkommen.

Beispiel: M2/2: Verwendung von Kellervariablen (Zwischenergebnis-Keller)

	$i := 0$	
$h[i+1] := a * b;$	$i := i + 1;$	$h[1] := a * b;$
$h[i+1] := a + d;$	$i := i + 1;$	$h[2] := a + d;$
$h[i] := h[i] * c;$		$h[2] := h[2] * c;$
$h[i-1] := h[i-1] + h[i];$	$i := i - 1;$	$h[1] := h[1] + h[2];$
$h[i+1] := b * b;$	$i := i + 1;$	$h[2] := b * b;$
$h[i] := a + h[i];$		$h[2] := a + h[2];$
$h[i-1] := h[i-1]/h[i];$	$i := i - 1;$	$h[1] := h[1]/h[2];$

Am Ende steht das Ergebnis immer in der Variablen $h[1]$!
Es werden anstelle von 7 Zwischenergebnis-Variablen nur maximal zwei Stack-Variable benötigt.
Dies ist aber nur ein marginaler Fortschritt, weil diese Interpretation nichts an dem von der Maschine zu verarbeitenden Code ändert.
Vielmehr kommt der von Neumann-Flaschenhals voll zur Wirkung:
Bei jeder von der CPU auszuführenden Operation erfolgen Zugriffe auf den Hauptspeicher, um sowohl Befehle als auch Operanden auszulesen.
Dies ist typisch für eine *CISC-Architektur* (CISC=Complex Instruction Set Computer).

Charakterisierung einer CISC-Architektur:

- relativ kompakter Code mit komplexen Befehlen auf Maschinenebene
- relativ hoher Steueraufwand zur Realisierung der Befehle
- Ausführung eines Befehls (entsprechend Mikrocode-Firmware) erfordert viele Takte der CPU
- komplexe und vielseitige Adressierungsvarianten
- Befehle sehr unterschiedlicher Länge, deshalb auch unterschiedliche Taktzahlen für die Ausführung der Befehle (komplizierte Steuerung in CPU)

Im Gegensatz dazu sind heutige Rechner meist als *RISC-Architektur* realisiert (RISC=Reduced Instruction Set Computer).

Charakterisierung einer RISC-Architektur:

- im Vergleich zur CISC-Architektur längerer Code einfacherer Befehle auf Maschinenebene

  z.B. VAX-11/780 (CISC): 250 Befehle im Instruktionssatz
  R-3000 Prozessor (RISC): <100 Befehle
  (beide von DEC)

- kaum Steueraufwand zur Ausführung eines Befehls (fest verdrahtet)
- vorwiegend ein CPU-Takt zur Ausführung eines Befehls im Register-Mode
- einfache Adressierungsvarianten, da Befehle auf Cache-Speicher zugreifen (u.U. Daten-Cache und Programm-Cache). Cache-Speicher sind sehr schnelle/teure statische RAM-Speicher.

Die Befehle des R-3000 haben alle die gleiche Länge, es existieren nur drei Befehlsformate und eine Adressierungsart. Bei der VAX-11/780 existieren dahingegen 13 Adressierungsarten.

- Zur Kommunikation mit dem Arbeitsspeicher existieren nur spezielle Lade- und Speicherbefehle, die Befehle / Daten in den Cache laden.

- Die Kommunikation mit dem Cache erfolgt ebenfalls über Lade- und Speicherbefehle.

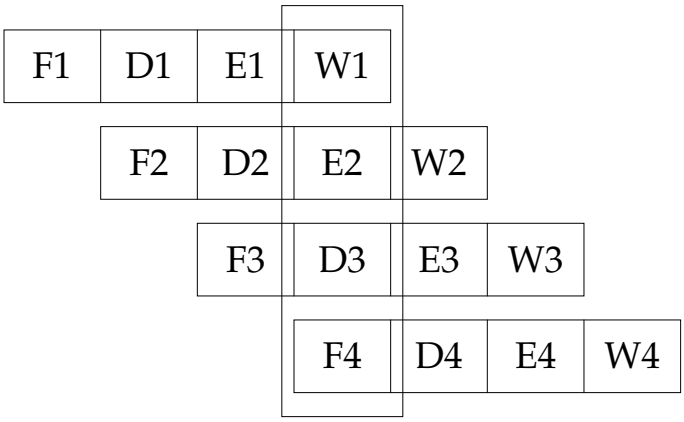
- Es existieren viele Register (z.B. 50).
  Die Operanden werden in Registern bereitgestellt, die Ergebnisse werden in Register abgelegt. Das heißt, Speichervariable (z.B. bezogen auf den Cache) kommen nur in elementaren Zuweisungen vor:

  $R_i := a$ Lesen aus dem Speicher
  $a := R_i$ Schreiben in den Speicher

- Wegen der guten Synchronisation der Ausführungszeit eines Befehls mit der Taktzeit der CPU können die Phasen

  - Fetch (Befehl holen)
  - Decode (Befehl decodieren)
  - Execute (Befehl ausführen)
  - Write (Ergebnis schreiben)

  aufeinander folgender Befehle parallelisiert ausgeführt werden. Dies heißt *Pipelining*. Eine vierstufige Pipeline hat folgende Gestalt:



Andere Partitionierungen der Befehlsausführung sind ebenfalls möglich.

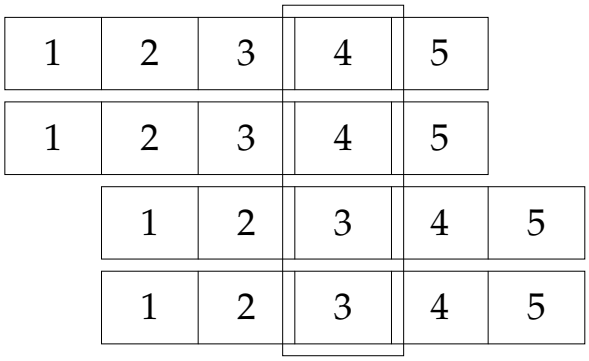
Obwohl die zur Bearbeitung eines Befehls erforderliche Zeit bei einer Pipelinestruktur nicht verkürzt wird (sie erfordert im Gegenteil einen geringen Mehraufwand), wird die Verarbeitungsrate nahezu um den Faktor der Quasi-Parallelität erhöht.

Im obigen Schema einer Pipeline ist angenommen, daß die Phasen der Befehlsausfürung gut mit der Taktfrequenz des Prozessors synchronisiert sind. Dann werden lediglich 1 CPI (cycles per instruction) zur Ausführung eines Befehls benötigt. Weitere Verbesserungen der Durchsatzrate erreichen moderne Architekturen von Prozessoren:

a) *Superskalare Pipeline*
Der Prozessor parallelisiert ein sequentiell geschriebenes Programm zur Laufzeit und kann pro Takt mehr als einen Befehl ausführen. Das erfordert, daß die Funktionseinheiten und Datenpfade des Prozessors entsprechend mehrfach vorhanden sind.

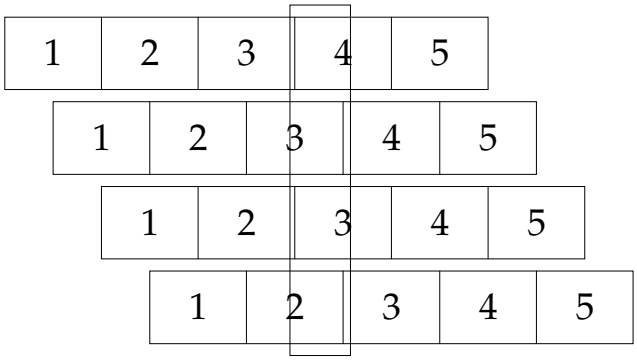
Bsp.: DEC-Alpha 21064 hat drei parallel arbeitende Funktionseinheiten, die von einem Decoder mit zwei Befehlen pro Takt versorgt werden (Superskalarität ist zwei).

Beispiel für 0.5 CPI:



b) *Superpipeline*
Eine Pipelinestufe wird weiter unterteilt mit dem Ziel, die Pipeline schneller zu takten und damit den Durchsatz zu erhöhen.

Beispiel für 0.5 CPI:



- Zur besseren Organisation der Pipelines existiert oft neben der CPU noch eine FPU (*Floating Point Unit*) mit eigenem Befehlssatz. Beispielsweise umfaßt der Befehlssatz der CPU der R-3000 (32-Bit-Maschine) 74 Befehle á 32 Bit und 32 allgemeine Register á 32 Bit. Die FPU der R-3000 verfügt über 17 Befehle á 32 Bit und 32 allgemeine Register á 64 Bit.

- Code-optimierende Compiler sind effizienter auf RISC-Architekturen. Sie reduzieren die Anzahl der Speicherzugriffe und die Möglichkeiten für Pipeline-Hazards.

Beispiel M2/3 (s. Seite 227) verdeutlicht das Prinzip der Codierung für eine RISC-Architektur im Speichermode (z.B. bzgl. Cache).

- *Registermode*: Quellen und Ziele von Operanden bzw. deren Adressen sind Register

- *Speichermode*: Speicheradressen werden mittels offsets (oder displacements) relativ zu in Registern bereitgestellten Basisadressen bestimmt

An diesem Beispiel ist es angebracht, die Verbindung zwischen der Stelligkeit einer Operation und der Adreßstruktur der entsprechenden Maschinenbefehle einzuführen.
Allgemein gilt für elementare Grundoperationen auf Maschinenebene:

a) 0-stellige Operation: erfordert Ein-Adreß-Befehl

$$h[i] := \text{“Wert”}$$

Ein-Adreß-Befehle beziehen sich auf Register oder Speicheradressen
z.B. Lade-Befehl (LD), Speicher-Befehl (ST)

b) 1-stellige Operation: erfordert Zwei-Adreß-Befehl

$$h[i] := \text{op}(h[j])$$

Angabe von Operanden- und Zieladresse
z.B. ABS

c) 2-stellige Operation: erfordert Drei-Adreß-Befehl

$$h[i] := \text{op}(h[j], h[k])$$

Angabe der Adressen von Operand1, Operand2, Zieladresse
z.B. MUL, DIV, ADD, SUB

Aber (siehe auch Abschnitt 4.3.1): Es müssen nicht alle Adressen unterschiedlich sein. Werden gleiche Adressen für einen Operanden und Ziel verwendet, sind folgende Begriffe gebräuchlich:

- monadischer Befehl: Zwei-Adreß-Befehl mit gleicher Operanden-/Ziel-Adresse
- dyadischer Befehl: Drei-Adreß-Befehl mit gleicher Adresse für einen Operanden und das Ziel
- triadischer Befehl: Drei-Adreß-Befehl mit expliziter Angabe aller drei Adressen

Beispiel: M2/3: RISC-Architektur als Kellermaschine (Speichermode)

	$i := 0$		
$h[i+1] := a;$	$i := i+1;$	$h[1] := a;$	`LD a`
$h[i+1] := b;$	$i := i+1;$	$h[2] := b;$	`LD b`
$h[i-1] := h[i-1] * h[i];$	$i := i-1;$	$h[1] := h[1] * h[2];$	`MUL`
$h[i+1] := a;$	$i := i+1;$	$h[2] := a;$	`LD a`
$h[i+1] := d;$	$i := i+1;$	$h[3] := d;$	`LD d`
$h[i-1] := h[i-1] + h[i];$	$i := i-1;$	$h[2] := h[2] + h[3];$	`ADD`
$h[i+1] := c;$	$i := i+1;$	$h[3] := c;$	`LD c`
$h[i-1] := h[i-1] * h[i];$	$i := i-1;$	$h[2] := h[2] * h[3];$	`MUL`
$h[i-1] := h[i-1] * h[i];$	$i := i-1;$	$h[1] := h[1] + h[2];$	`ADD`
$h[i+1] := a;$	$i := i+1;$	$h[2] := a;$	`LD a`
$h[i+1] := b;$	$i := i+1;$	$h[3] := b;$	`LD b`
$h[i+1] := b;$	$i := i+1;$	$h[4] := b;$	`LD b`
$h[i-1] := h[i-1] * h[i];$	$i := i-1;$	$h[3] := h[3] * h[4];$	`MUL`
$h[i-1] := h[i-1] + [i];$	$i := i-1;$	$h[2] := h[2] + h[3];$	`ADD`
$h[i-1] := h[i-1]/h[i];$	$i := i-1;$	$h[1] := h1[1]/h[2];$	`DIV`
			`ST m`

Im obigen Beispiel treten in der Präfixnotation des Mnemocodes dyadische Befehle auf (Ziel der Operation und Quelle eines Operanden sind gleich).

Eine Maschine, die sich auf das Prinzip der Kellerverarbeitung allein stützt, bezeichnet man als *Kellermaschine*. Reine Kellermaschinen findet man nur für spezielle Anwendungen realisiert.

Beispielsweise arbeiten manche Taschenrechner so. Aber auch der einst berühmte Transputer T800 von Inmos war eine Kellermaschine.

Kellermaschinen beziehen alle Operationen auf die oberste Kellervariable TOS (Top Of Stack), z.B.

PUSH - LOAD
POP - STORE

Arithmetische Operationen erfordern keine explizite Angabe der Operandenadressen, da sich einstellige Operationen stets auf TOS und zweistellige Operationen sich stets auf TOS und TOS-1 beziehen.

Deshalb ist der Mnemo-Code dieser Operationen (z.B. MUL, ADD) argumentfrei und Kellermaschinen werden auch als *Null-Adreß-Rechner* bezeichnet.

Beispiel M2/3 zeigt rechts den Mnemocode zur Berechnung des Terms bei der Interpretation als Kellermaschine.

Da RISC-Architekturen registerorientiert arbeiten, werden dort die Kellervariablen h[i] durch Register repräsentiert. Sie müssen aber adressiert werden, weil keine Kellerverarbeitung angenommen wurde.

Beispiel: M2/4: RISC-Architektur (Registermode)

$R1 := a;$	`LD R1,a`
$R2 := b;$	`LD R2,b`
$R1 := R1 * R2;$	`MUL R1,R2`
$R2 := a;$	`LD R2,a`
$R3 := d;$	`LD R3,d`
$R2 := R2 + R3;$	`ADD R2,R3`
$R3 := c$	`LD R3,c`
$R2 := R2 * R3;$	`MUL R2,R3`
$R1 := R1 + R2;$	`ADD R1,R2`
$R2 := a;$	`LD R2,a`
$R3 := b;$	`LD R3,b`
$R4 := b;$	`LD R4,b`
$R3 := R3 * R4;$	`MUL R3,R4`
$R2 := R2 + R3$	`ADD R2,R3`
$R1 := R1/R2;$	`DIV R1,R2`

In älteren Maschinen stand nur ein Register zur Aufnahme von Zwischenergebnissen zur Verfügung (der Akkumulator AC).

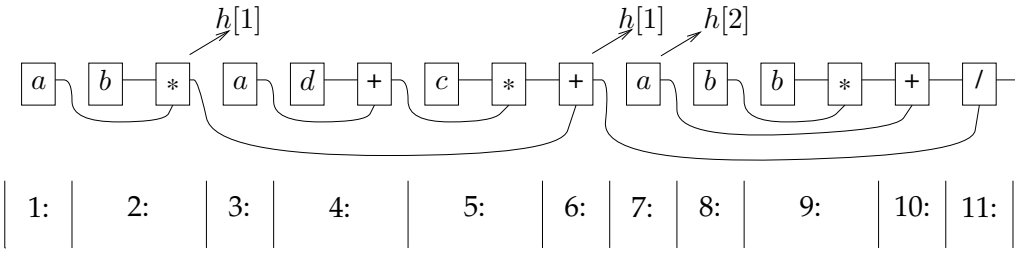
Dessen Adresse muß demzufolge nicht mehr explizit spezifiziert werden. Zwischenergebnisse und Speichervariable werden im Speichermode adressiert. Man bezeichnet Architekturen, die nach diesem Prinzip arbeiten als *Ein-Adreß-Rechner*.

Folgende Befehlsstrukturen werden verwendet:

- für zweistellige Operationen: $AC := AC\ \mathrm{op}(h[i])$
- für einstellige Operationen: $AC := \mathrm{op}(AC)$
- direkte Ladebefehle: $AC :=$ "Wert"
- adressierte Ladebefehle: $AC := h[i]$
- adressierte Speicherbefehle: $h[i] := AC$

Diese Ein-Adreßform läßt sich direkt aus dem linearisierten Kantorovič-Baum gewinnen:

Beispiel: M2/5: Ein-Adreß-Befehle mit Akkumulator



$1:$ $AC := a;$
$2:$ $AC := AC * b;$
$h[1] := AC;$
$3:$ $AC := a;$
$4:$ $AC := AC + d;$
$5:$ $AC := AC * c;$
$6:$ $AC := h[1] + AC;$
$h[1] := AC;$
$7:$ $AC := a;$
$h[2] := AC;$
$8:$ $AC := b;$
$9:$ $AC := AC * b;$
$10:$ $AC := h[2] + AC;$
$11:$ $AC := h[1]/AC;$

4.2.2 Speicherabbildung und Laufzeitumgebung

Normalerweise laufen zu jedem Zeitpunkt quasi gleichzeitig mehrere Prozesse, wenigstens das Betriebssystem und eine (graphische) Benutzeroberfläche.
Wird ein neuer Prozeß gestartet, so hat die Memory Managment Unit (MMU) der OSE dem Prozeß einen für seinen Code und zur Generierung dynamischer Speichervariabler ausreichenden Speicherplatz zuzuweisen (*Programmallokation*).
Dabei wird unterschieden zwischen

- statischer Programmallokation: erfolgt vor der Laufzeit des Prozesses,
- dynamischer Programmallokation: erfolgt während der Laufzeit des Prozesses.

Die OSE hat natürlich auch für die Freigabe von Speicherbereichen zu sorgen (*Delokation*). Dies kann dynamisch (während der Laufzeit eines Prozesses) oder nach dessen Beendigung (statisch) erfolgen.
Zu den Aufgaben der MMU gehört auch, Zerstückelungen des Adreßraumes durch Kompaktifizierung bzw. Speicherbereinigung (*garbage collection*) zu vermeiden (vgl. Abbildung 4.8). Dies erfordert, adreßabhängige Größen eines Prozesses an die korrigierte Speicherbelegung auszurichten (*Relokation*). Dies wird von einem Basisregister unterstützt, das die Basisadresse des gerade laufenden Prozesses hält. Das setzt voraus, daß die Adreßbezüge des Programms nicht absolut angegeben sind (Programm ist *verschieblich*).
Einem lauffähigen Programm (Annahme: nicht-triviales Programm rekursiver Struktur) wird vom Betriebssystem ein direkt adressierbarer Speicherbereich (*Region*) zugewiesen, dessen Größe sich nicht allein aus dem Umfang des Maschinencodes ergibt (vgl. Abbildung 4.9):

1. ein Adreßbereich fester Größe, in dem der Maschinencode des übersetzten Programms abgelegt wird;
2. ein Adreßbereich variabler Größe, der *Laufzeitstack*, in dem *Aktivierungsrekords* für Funktions-/Prozeduraufrufe auf- und abgebaut werden;
3. ein Adreßbereich variabler Größe, die Halde (*heap*), in dem zeigerverkettete Rekordstrukturen zur Laufzeit angelegt und wieder aufgelöst werden.

Der Speicherplatz eines Programmes bildet sich also

- aus einem statischen Anteil:

  Bereits der Compiler ordnet statischen Variablen ihren Speicherplatz entsprechend den deklarierten Typen zu.
  Hierzu gehören auch Funktionsaufrufe (entsprechend der Deklaration von Funktionen und Parametern).

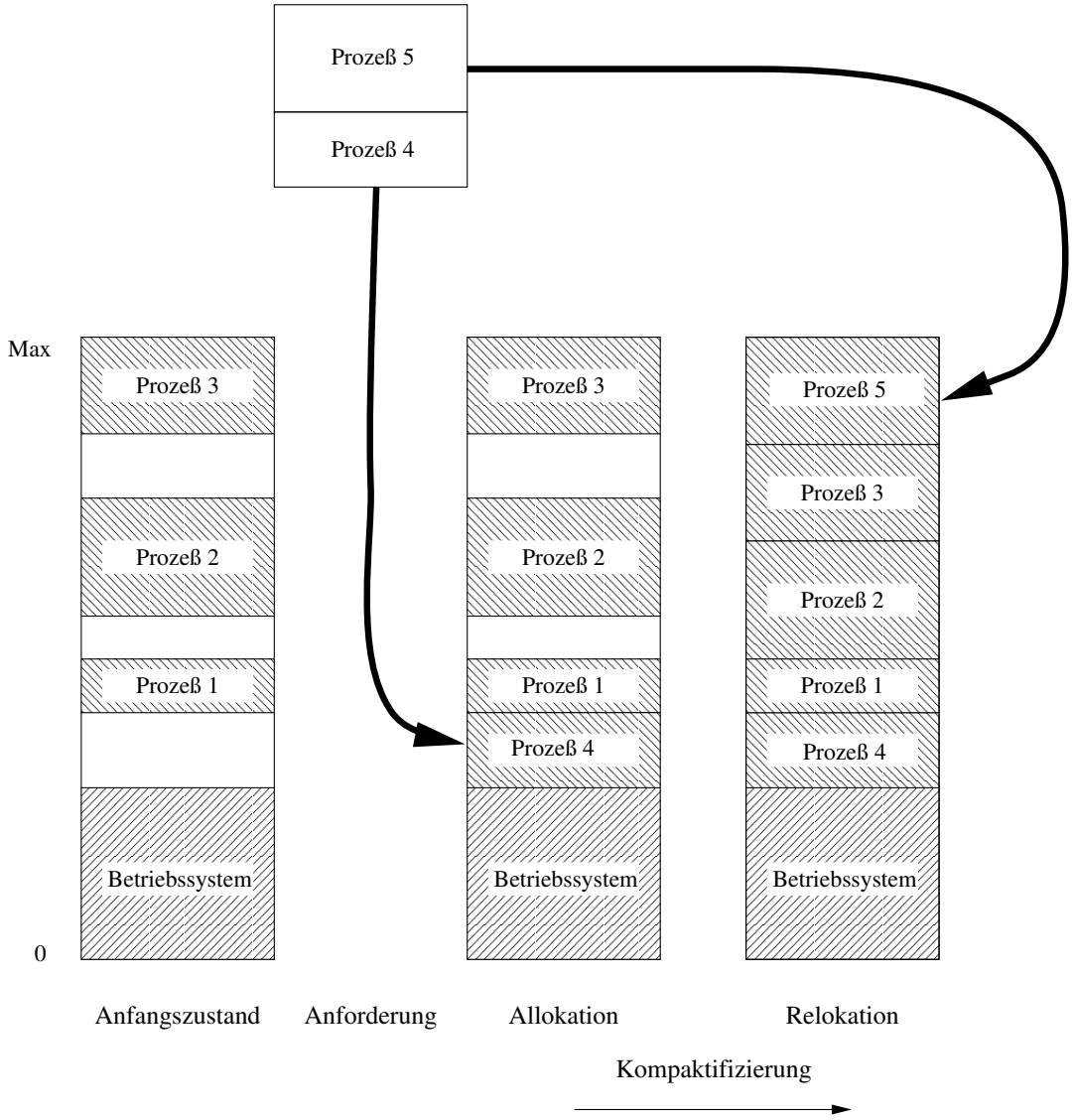


Abbildung 4.8: Aufgaben der MMU

- aus einem dynamischen Anteil: Laufzeitumgebung

  Die Speicherzuordnung dynamischer Variabler erfolgt erst zur Laufzeit der sie enthaltenden Programmstrukturen (umfaßt Heap und Laufzeitstack).

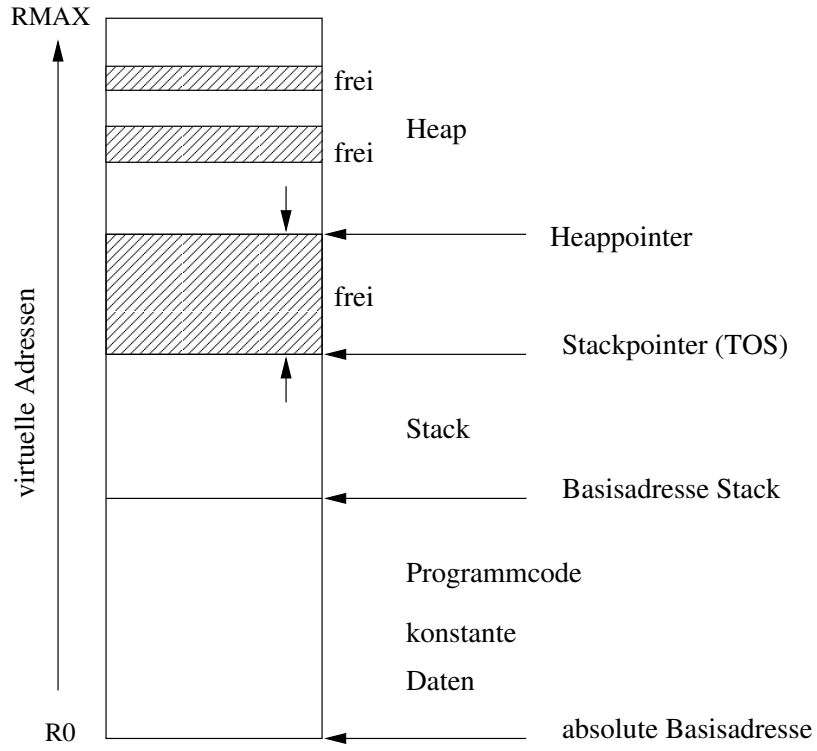


Abbildung 4.9: Region eines Programmes

4.2.2.1 Die Nutzung des Heap-Managers in "C"

In "C" sind externe Variable sogenannte statische Variable. Interne Variable, die nur innerhalb der zusammengesetzten Anweisungen zur Verfügung stehen, in denen sie deklariert werden, sind dynamische Variable.

Die dynamische Speicherverwaltung erfolgt automatisch beim Programmablauf unter der Regie der MMU der OSE.

In "C" hat aber der Programmierer die Möglichkeit, den Heap-Manager zu nutzen, um eine dynamische Speicherzuordnung zu organisieren. Hierzu stehen in der Datei `<stdlib.h>` mehrere Funktionen zur Verfügung.

Beispielanwendungen:

- Anlegen eines Feldes variabler Größe eines beliebigen Typs
- Konkatenieren zweier Strings variabler Größe

Funktionen in "C" zur Speicherverwaltung:

- `void *malloc(size_t Groesse)`
- `void free(void *Zeiger)`
- `void *realloc(void *Zeiger, size_t Groesse)`
- `void *calloc(size_t Anzahl, size_t Groesse)`

1. Funktion `malloc`
   - stellt einen Speicherbereich der Größe `Groesse` Bytes zur Verfügung (ohne spezielle Anfangsbelegung),
   - liefert als Funktionswert einen Zeiger auf den Anfang des Bereiches,
   - liefert einen Nullzeiger, wenn Speicherplatz nicht verfügbar,
   - `void *` repräsentiert einen nicht dereferenzierbaren Zeigertyp, der beliebigen Zeigervariablen zugewiesen werden kann,
   - Struktur erhält der Speicher erst durch die Angabe der verwendeten Zeigervariablen, indem `malloc` in einer anderen Funktion aufgerufen wird,
   - `size_t` ist vorzeichenlos und ganzzahlig, der Typ hängt vom Argument des Operators `sizeof()` der Headerdatei `<stddef.h>` ab.

Beispiel:

```
int dyn(int i)
{
  float *v;
  v=(float *) malloc(i*sizeof(float));
  ⋮
  free(v);
}
```

Die Funktion `dyn` stellt dynamisch i Speicherplätze vom Typ `float` bereit. Diese werden am Ende der Funktion wieder freigegeben.

---

2. Funktion `free`

   Da der Heap-Manager das Konzept "interne Variable" des C-Compilers nicht kennt, muß jede Funktion, die dynamischen Speicher zur internen Verwendung anfordert, diesen auch wieder freigeben. (Im obigen Beispiel existiert die interne Variable `v` zwar nur innerhalb der Funktion `dyn`, aber ohne Freigabe würde der reservierte Speicherbereich blockiert werden).

Das Argument der Funktion muß ein Zeiger sein, der vorher von `malloc` oder anderen Funktionen des Heap-Managers geliefert wurde.

3. Funktion `realloc`
   - stellt eine Kombination von `malloc` und `free` dar
   - wenn `Zeiger` = Nullzeiger, dann wirkt `realloc` wie `malloc`
   - wenn `Zeiger` $\neq$ Nullzeiger und `Groesse` = Null, dann wirkt `realloc` wie `free`
   - sonst Vergrößerung bzw. Verkleinerung des reservierten Speichers, je nachdem ob `Groesse` größer oder kleiner als die Größe des bisher reservierten Speichers ist.
4. Funktion `calloc`
   - stellt einen Zeiger auf den Anfang eines Feldes mit der Anzahl der Komponenten und jeweiliger Größe zur Verfügung
   - die Komponenten werden jeweils auf Null gesetzt.

4.2.2.2 Der Laufzeitstack

Beim Aufruf von Unterprogrammen oder Funktionen müssen zwei organisatorische Probleme gelöst werden:

1. Es muß eine Rücksprungadresse (Adresse des Wiedereintrittes in das aufzurufende Programm) zwischengespeichert werden.
2. Es müssen die Übergabe der Argumente (aktuelle Parameter oder Parameterwerte) und die Rückgabe des Ergebnisses organisiert werden.

Hierfür wird eine Stackverwaltung organisiert, die man *Laufzeitstack* der Programmabarbeitung nennt.

Im Falle geschachtelter Prozedurdefinitionen (wie bei Pascal oder ähnlichen Sprachen – in "C" ist dies nicht möglich) oder im Falle rekursiver Funktionsaufrufe (in "C" ist dies möglich) verbieten sich einfache register- oder speicherorientierte Methoden (vgl. Abbildung 4.10).

Mit der Deklaration einer Funktion wird dem Compiler mitgeteilt, wieviel Speicherplatz (Anzahl und Typ der Parameter) auf dem Stack reserviert werden muß. Außerdem muß die Rücksprungadresse dort gespeichert werden. Die Deklaration legt aber auch fest, von welchem Typ der Rückgabewert der Funktion ist. Danach richtet sich die Berechnung der Rückkehradresse als Inkrement des Befehlszählers zur Anfangsadresse der Funktion.

Mit der Definition der Funktion wird auf dem Stack für alle lokalen Variablen der zu reservierende Speicherplatz festgelegt.
Ein *Aktivierungsrekord* (*activation record*) für Prozedur-/Funktionsaufrufe ist ein fest vorformatierter Rahmen (*frame*), in dem

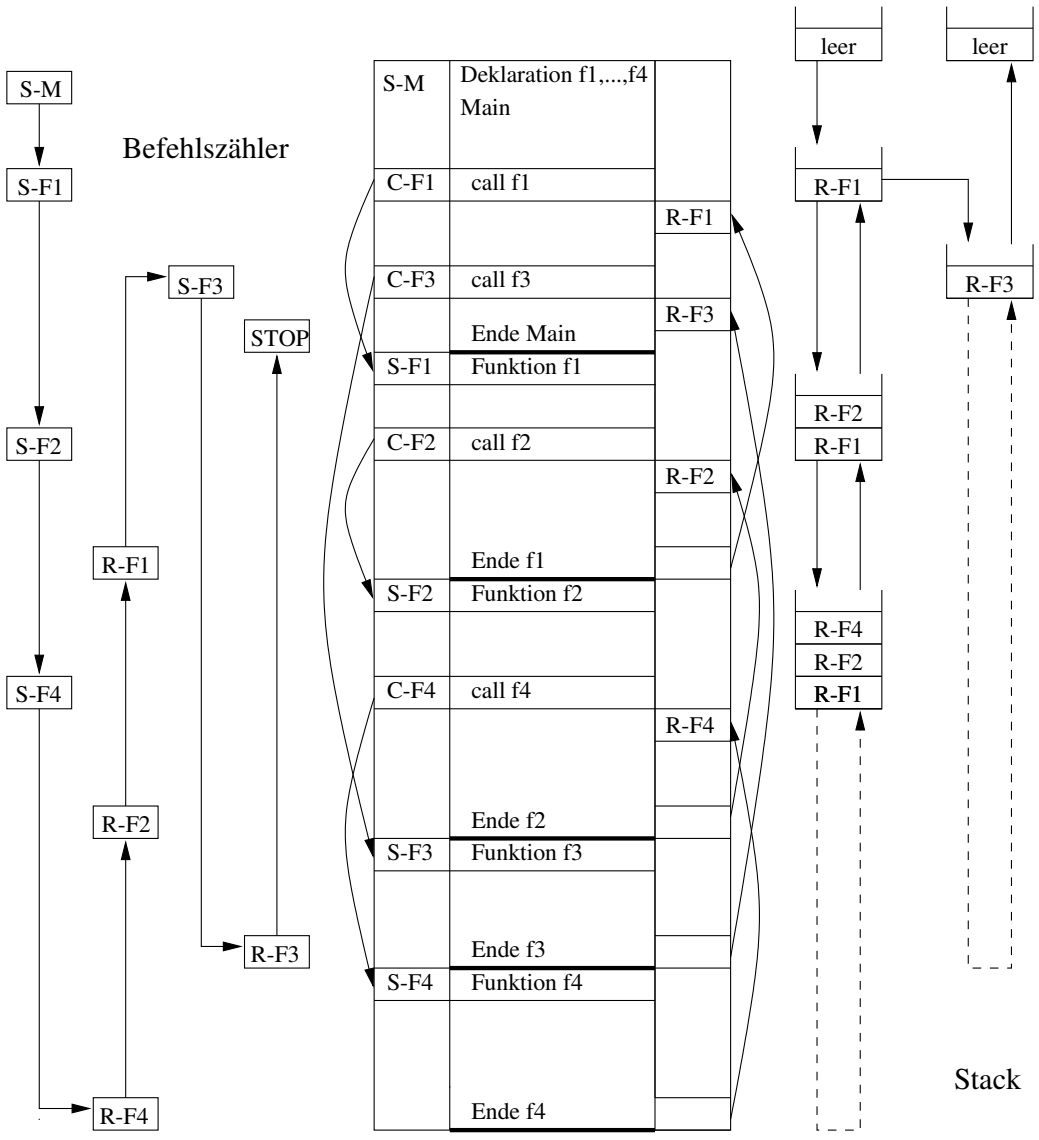


Abbildung 4.10: Verwaltung der Rücksprungadressen bei verschachtelten Funktionsaufrufen

- aktuelle Prozedur-/Funktionsparameter
- Instantiierungen lokaler Variabler
- gegebenenfalls vom Compiler erzeugte Hilfsvariable (*temporaries*)

organisiert werden.
Jeder Aktivierungsrekord besteht aus

- *Parameter-Übergabebereich* (*argument frame*)

  Aufnahme aktueller Werte-/Referenzparameter vom aufrufenden Programm/ Funktion/Prozedur

- *Kontrollblock* (*control block*)

  Verkettung der aufgerufenen Prozedur mit der aufrufenden Prozedur (z.B. bzgl. Rückkehradresse und Statusinformation)

- *Arbeitsbereich* (*workspace frame*)

  Organisation lokaler Variabler und Hilfsvariabler

Die aufrufende Funktion erzeugt das argument frame der aufgerufenen Prozedur sowie Teile des Kontrollblockes (z.B. Rückkehradresse).
Die aufgerufene Funktion baut den Rest des Kontrollblockes auf (Inhalte von Registern, die bei der Rückkehr wiederhergestellt werden müssen) und legt ihren Arbeitsbereich an.
Für den Abbau des Aktivierungsrekords sind die erzeugenden Funktionen verantwortlich.
In "C" werden Zeiger-Verkettungen dynamisch erzeugt (z.B. bei Rekursion) oder bei Funktionsaufrufen kopiert, wenn deren Verkettung bereits bei der Compilierung bekannt ist.
Abbildung 4.11 zeigt den Aufbau des Laufzeitstacks beim Aufruf einer Funktion g aus einer Funktion f heraus.
Die Verwaltung des Aktivierungsrekords einer aktiven Prozedur (eines aktiven Funktionsaufrufes) erfolgt durch einen Prozessor mittels spezieller Register (vgl. Abb.4.12):

TOS - top of stack: Zeiger auf die aktuelle Oberfläche des Laufzeitstacks
AFP - argument frame pointer: Zeiger auf die Basis des aktuellen Argumentframes
WFP - workspace frame pointer: Zeiger auf die Basis des aktuellen Arbeitsbereiches

Sie dienen der relativen Adressierung von Einträgen in den Aktivierungsrekords. Reine speicherorientierte Laufzeitstacks (memory stacks) sind typische Konstrukte für CISC-Architekturen: Es sind Hauptspeicherzugriffe mit der Konsequenz des von Neumann-Flaschenhalses erforderlich.

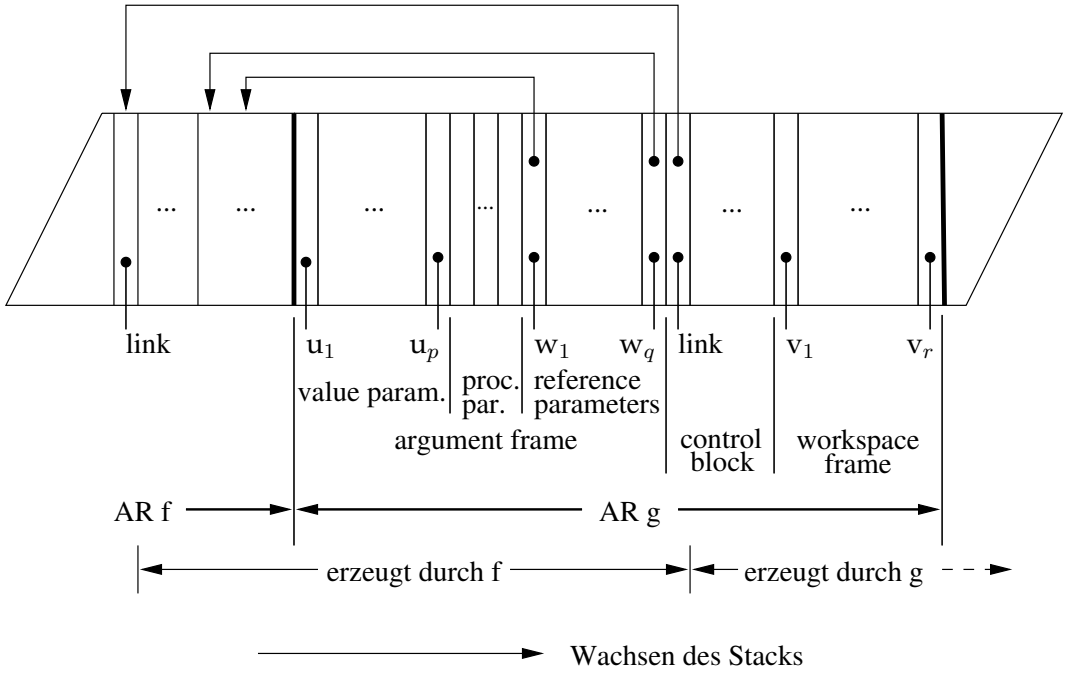


Abbildung 4.11: Aufruf der Funktion g durch Funktion f

Eine Alternative stellt die Parameterverwaltung für Unterprogramm-/Funktionsaufrufe in Registern (Registerspeicher) als *Registerstack* dar.

Die allgemeine Realisierung eines aus Registern aufzubauenden Caches bereitet technische Probleme. Im folgenden Abschnitt wird eine sehr effiziente, spezielle Lösung vorgestellt, die für SPARC-Prozessoren typisch ist.

4.2.2.3 Registerfenster

Typisches Kennzeichen von RISC-Architekturen ist die Beschränkung der Hauptspeicherzugriffe auf spezielle Lade- und Speicherbefehle. Alle anderen Befehle verwenden als Operanden nur Register. Auch die Aktivierungsrekords werden teilweise in Registern verwaltet. Insbesondere deshalb verfügen moderne RISC-Prozessoren zum Teil über mehrere Hundert Register.

Man unterscheidet heute verschiedene Klassen von RISC-Prozessoren, z.B.

1. SPARC (scalable processor architecture): SUN und viele andere

2. MIPS (Rx000, 80x86): AT&T, Intel

3. Power PC (performance optimization with enhanced RISC-performance computing): IBM, Apple

Zunehmend verschwindet dabei die einfache Unterscheidung von CISC- und RISC-Merkmalen.

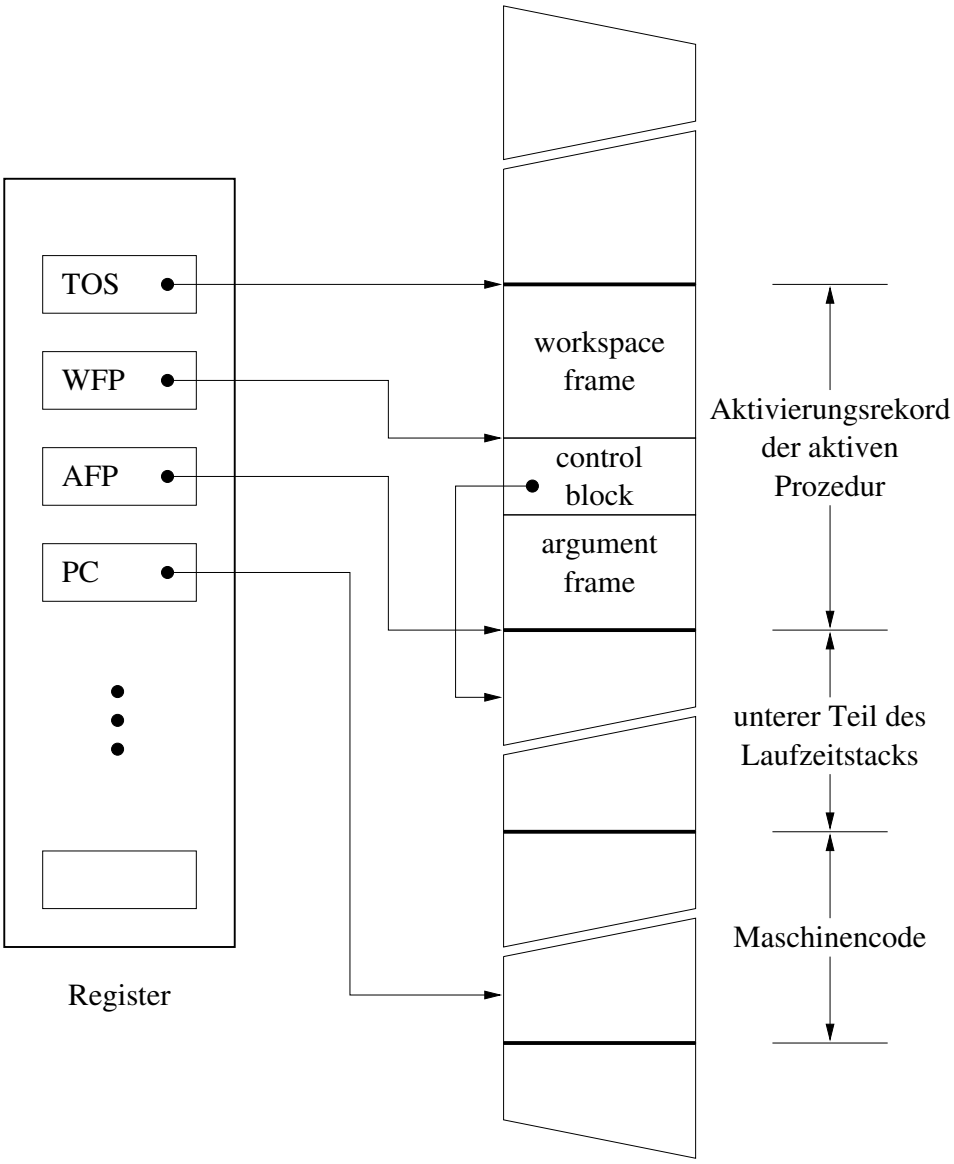


Abbildung 4.12: Registerverwaltung von Aktivierungsrekords

Der SPARC-Prozessor verwendet für eine Parameterübergabe ohne Speicherzugriffe sogenannte *Registerfenster*.
Ebenso wie softwareseitig die Programmiersprache "C" in engster Anlehnung an das Betriebssystem UNIX entworfen wurde und auch so genutzt wird, stellt der SPARC-Prozessor die hardwaremäßige Anpassung an UNIX dar.
In "C" kennt man drei Arten von Variablen:

- globale Variable: sind im gesamten Programm bekannt
- lokale Variable: sind nur innerhalb einer Funktion deklariert
- Eingabeparameter: sind als formale Parameter einer Funktion deklarierte Variable

Diese drei Arten von Variablen werden beim SPARC-Prozessor direkt von der Hardware unterstützt:

1. GLOBALS: globale Variable
2. LOCALS: lokale Variable
3. INS: Eingabe-Parameter der aufgerufenen Funktion
4. OUTS: Übergabe-Parameter der aufrufenden Funktion

Diese Variablen werden auf Register folgender Typen abgebildet:

- globale Integer-Register
- globale Gleitpunkt-Register
- gefensterte Integer-Register (LOCALS, INS, OUTS)

Ein *Registerfenster* besteht aus je einem Satz von Registern für INS, OUTS und LOCALS. Dabei überlappen sich die Register für INS und OUTS in der Weise, daß

> INS der aktuellen aufgerufenden Funktion $\equiv$ OUTS der aufrufenden Funktion

Bei Rückkehr vertauscht sich die Rolle der INS und OUTS.
Registerfenster sind zyklisch angeordnet. Da sowohl die Zahl der Register in einem Fenster als auch die Anzahl der Fenster begrenzt ist, müssen Registerfenster dennoch mit einem Speicherstack zur Unterprogrammverwaltung zusammenarbeiten.

Beispiel:

Der SPARC-4 hat maximal 32 Registerfenster mit jeweils 16 Registern:

8 lokale, $\underbrace{\text{8 INS, 8 OUTS}}_{\text{sind identisch}}$

8 globale (Integer-Register) sind nicht den Fenstern zugeordnet

Die überlappenden Registerfenster bewirken, daß bei Unterprogramm-Aufruf/Rückkehr die registerallozierten LOCALS und Referenz-Adressen nicht über den Laufzeitstack (memory stack) gerettet oder rekonstruiert werden müssen. Der Gewinn an verzichtbaren Lade-/Speicheroperationen im Maschinencode ist beachtlich:

Für große "C"-Programme bestehen bei CISC-Prozessoren ca. 50% des Codes aus Lade- und Speicheroperationen, bei RISC-Rechnern ohne Registerfenster sind es ca. 30%, bei SPARC-Prozessoren sind sie auf 20% reduziert. Abbildung 4.13 zeigt die Registerfenster eines SPARC-4-Prozessors. Dabei werden folgende Registernamen verwendet:

R_{24}...R_{31}	$\hat{=}$	i0 ...i7	INS
R_{16}...R_{23}	$\hat{=}$	l0 ...l7	LOCALS
R_{8} ...R_{15}	$\hat{=}$	o0...o7	OUTS
R_{0} ...R_{7}	$\hat{=}$	g0...g7	GLOBALS

Folgende Register sind besonders ausgezeichnet:

g0	Null-Wert, schreibgeschützt
sp $\hat{=}$ o6	Stackpointer (TOS)
fp $\hat{=}$ i6	Framepointer (AFP)
o7	Return-Adresse

Zur Identifizierung des jeweils aktuellen Fensters dienen der CWP (Current Window Pointer), der auf das aktive Fenster zeigt und die WIM (Window Invalid Mask) des PSR (Prozessor Status Register).

Durch den Compiler wird bei einem Funktionsaufruf (Compilation von "`call`") die SAVE-Instruktion verwendet. Diese bewirkt

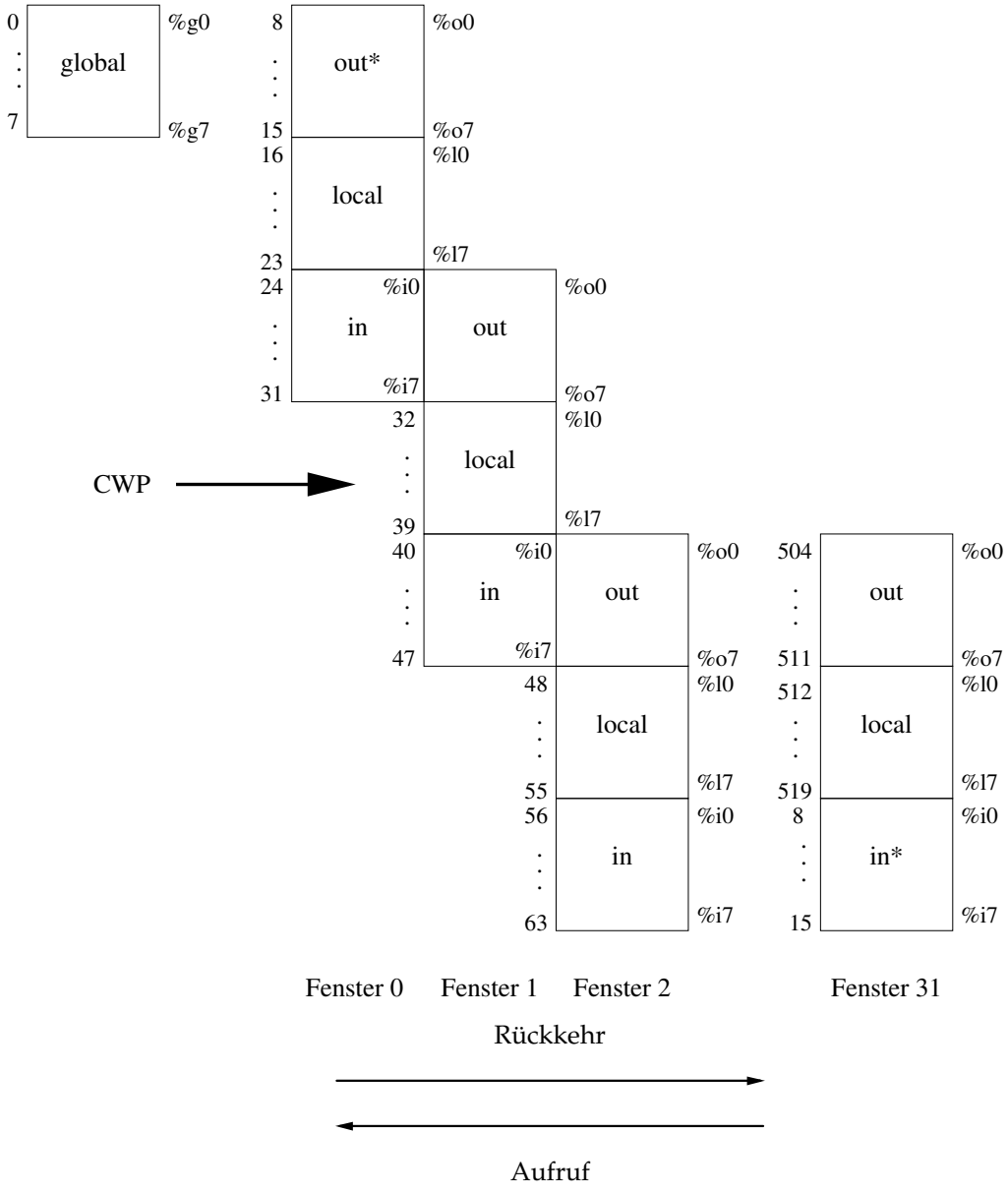


Abbildung 4.13: Registerfenster des SPARC-4

1. Dekrementierung des CWP mit der Wirkung

   - OUTS der aufrufenden Prozedur werden INS der aufgerufenen Prozedur (dies geschieht ohne Datentransport!)

- neue Sätze für OUTS und LOCALS werden angelegt

2. Inkrementierung des Stack-Pointers

Bei der Rückkehr aus einer Prozedur/Funktion werden die zurückzugebenden Werte in die INS des aktuellen Fensters geschrieben. Der Compiler fügt die Instruktion "`restore`" anstelle der Instruktion "`return`" ein. Diese bewirkt

1. Inkrementierung des CWP mit der Wirkung

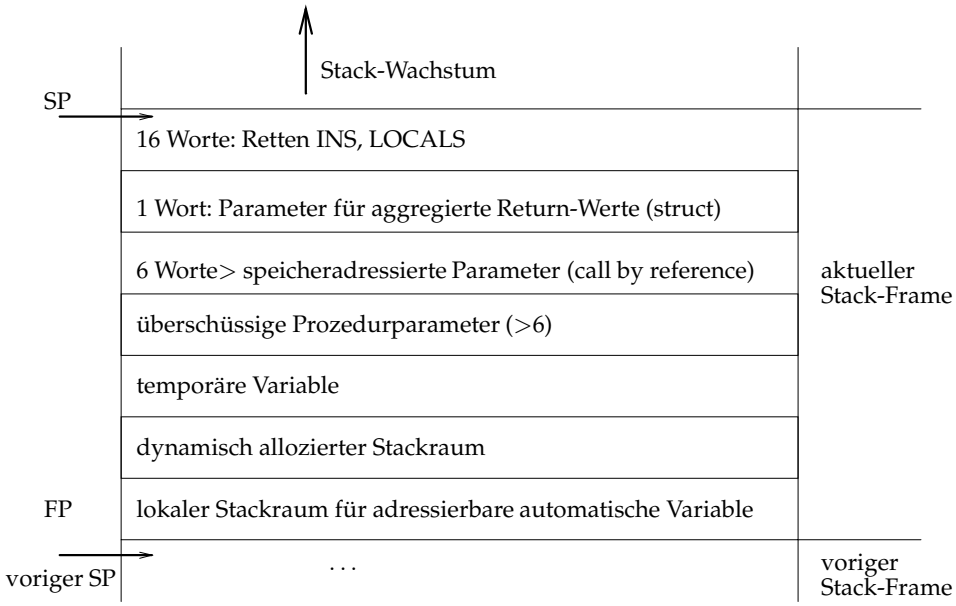
   - INS der aufgerufenen (nun abzuschließenden) Prozedur werden OUTS der aufrufenden Prozedur

2. Dekrementierung des Stack-Pointers

Die sechs variablen globalen Register werden für globale Variable/Pointer bei Unterprogrammaufrufen verwendet.
Der *Stack-Frame* ergänzt das Registerfenster. Er dient der Speicherung von

- Prozedurparametern (wenn mehr als 6)
- speicheradressierbaren Parametern (call by reference), da Register keine Speicheradressierung erlauben (bei SPARC)
- INS und LOCALS, die im Falle eines Registerfenster-Overflow gerettet werden
- adressierbaren lokalen Variablen (automatische Variable)
- vom Compiler erzeugten temporären Variablen
- geretteten Gleitpunkt-Registern.

Der Stack-Frame einer aktiven Prozedur/Funktion hat folgende Gestalt:


Stack-Wachstum
SP
16 Worte: Retten INS, LOCALS
1 Wort: Parameter für aggregierte Return-Werte (struct)
6 Worte> speicheradressierte Parameter (call by reference)
überschüssige Prozedurparameter (>6)
temporäre Variable
dynamisch allozierter Stackraum
FP
lokaler Stackraum für adressierbare automatische Variable
voriger SP
. . .
aktueller Stack-Frame
voriger Stack-Frame

4.3 Die Instruktionssatzarchitektur

Auf der Maschinenebene liegen sowohl die zu verarbeitenden Daten als auch die vom Prozessor auszuführenden Instruktionen/Befehle in binärer Form vor. Beide sind nur dadurch vom Prozessor voneinander zu unterscheiden, daß

- Zwänge genutzt werden können, die in der Folge der Abspeicherung des Programmcodes liegen,
- Befehle eine syntaktische Struktur haben, deren Decodierung und Befolgung zwangsläufig zur richtigen Interpretation des Codes führt.

Ein Prozessor ist also ein reaktives System. Er tut, was in den Befehlen codiert ist. Dies setzt voraus, daß Prozessorarchitektur und *Instruktionssatzarchitektur* als sich ergänzende Einheiten betrachtet werden.

Das bedeutet aber, daß jeder neu auf dem Markt erscheinende Prozessor seine eigene Maschinensprache erfordert. Dieser Umstand wird dadurch gemildert, daß sich mehr und mehr Prozessorfamilien herausbilden, die gewissen einheitlichen Gesichtspunkten folgen.

Die Programmierung auf Maschinenebene nutzt die Instruktionssatzarchitektur gewöhnlich als Sprachebene. In Ausnahmefällen kann diese aber auch selbst geändert werden, indem die Instruktionen auf der Ebene des Microcodes modifiziert werden.

Die Programmierung auf Maschinenebene ist schwierig, unbequem und fehleranfällig. Deshalb wird sie auf Spezialaufgaben (Prozeß-, Gerätesteuerung, ...) beschränkt sein.

Sie erfolgt nicht durch Komponieren von Binärzahlen, sondern in einem Mnemocode, der *Assemblersprache*, die eine 1:1 Relation zum binären Instruktionssatz besitzt.

4.3.1 Maschineninstruktionen

Maschineninstruktionen codieren

- die auszuführenden Operationen
- die zugeordneten Operanden.

Das Paradigma der imperativen Programmierung erfordert nicht nur die Spezifikation dessen, was zu berechnen ist, sondern auch, wie es zu berechnen ist.

Deshalb enthält der Instruktionssatz auch Möglichkeiten der Steuerung des Berechnungsablaufes:

- es existieren Steuer-Instruktionen

- es werden Operationen modifiziert, insbesondere bezüglich des Zugriffs auf die Operanden (Adreßmodifikationen)

Die Sprachelemente höherer Programmiersprachen werden auf Maschinenebene auf drei *Instruktionsklassen* abgebildet:

1. *wert-(zustands-)transformierende Instruktionen:*

   <val_instr> : <val_rator>{<rand>}^{n}<dest>

   mit

   <val_rator> : Operator der Instruktion
   <{<rand>}^{n}> : Quellen von n Operanden
   <dest> : Ziel des Resultatwertes

   Der Operator codiert den Typ der Operanden als einen der Maschinentypen:

boolean	: Byte-Format
integer	: Byte-, Halbwort-, Wort-Format
real	: Wort-, Doppelwort-Format
character	: Byte-Format
string	: Zeichensequenz bis zu 256 Byte

   Die Operatoren sind normalerweise monomorph getypt, d.h. die Operanden sind vom gleichen Typ.

   Zwischen folgenden *Adreß-Spezifikationen* wird unterschieden:

   Registermode Quellen und Ziel bzw. deren Adressen befinden sich in prozessoreigenen Registern

   Speichermode Quellen und Ziel bzw. deren Adressen befinden sich an Speicheradressen. Die Angabe dieser Adressen erfolgt meist relativ zu Basisadressen, die in Registern gehalten werden, d.h. zur Adressierung werden *offsets/displacements* angegeben.

2. *transportierende Instruktionen*:

   <trans_instr>:<trans_rator><source><dest>

   Sowohl Quelle als auch Ziel können Register oder Speicherzellen sein.

3. *steuernde Instruktionen*:

   <contr_inst> :<contr_rator>(<predicate>){<label>}^{n}

   mit

<contr_rator> : Kontrolloperator
<predicate> : optionales Tupel von Prädikaten, die vom Kontrolloperator ausgewertet werden
{<label>}^{n} : n alternative Sprungziele des Befehlszählers innerhalb des Programmcodes

Die steuernden Instruktionen dienen zur Realisierung bedingter/unbedingter Sprünge sowie für Unterprogramm-/Funktionsaufrufe und zur Rückkehr von diesen.

In Abhängigkeit von der Anzahl der Quellen/Ziele unterscheidet man

1. *monadische* Instruktionen: ein Operand bzw. Zieladresse

   Monadische Instruktionen beziehen sich zum Teil implizit auf Register (PC, Akkumulator, ...) oder Speicheradressen (z.B. Startadresse von Funktionen). Steuerinstruktionen sind monadisch. Die Rückkehr aus Unterprogrammen "ret" bzw. "return" erfolgt ohne Adreßspezifikation, weil sie sich implizit auf die im Laufzeitstack oder im Registerfenster gespeicherte Rücksprungadresse bezieht.

2. *dyadische* Instruktionen: zwei Operanden

   Die Adresse des zweiten Operanden ist auch die Zieladresse.

3. *triadische* Instruktionen: explizite Angabe zweier Quell- und einer Zieladresse

   Triadische Operationen werden bei RISC-Prozessoren gegenüber dyadischen Operationen bevorzugt.

Im folgenden sei zunächst eine hypothetische Maschinensprache für einen CISC-Prozessor angenommen. Hieran sollen einige typische Konzepte des Instruktionssatzes erläutert werden.
Die syntaktischen Regeln zur Konstruktion von Maschinenbefehlen sei in folgender Weise gegeben (Annahme: 32-Bit-Prozessor):

```
<instr>       : <monad_instr>|<dyad_instr>|<triad_instr>
<monad_instr> : <monad_op><addr>
<dyad_instr>  : <dyad_op><addr><addr>
<triad_instr> : <triad_op><addr><addr><addr>
<addr>        : <reg_mode><register>|
                <mem_mode><register><offset>{<length>}
<register>    : R0|...|R15|PC|TOS
<offset>      : <short_offset>|<long_offset>
<short_offset>: -2^7 ... 2^7 - 1     (n = 8 Bit)
<long_offset> : -2^23 ... 2^23 - 1   (n = 24 Bit)
<length>      : 0| ... |255          (in Byte)
```

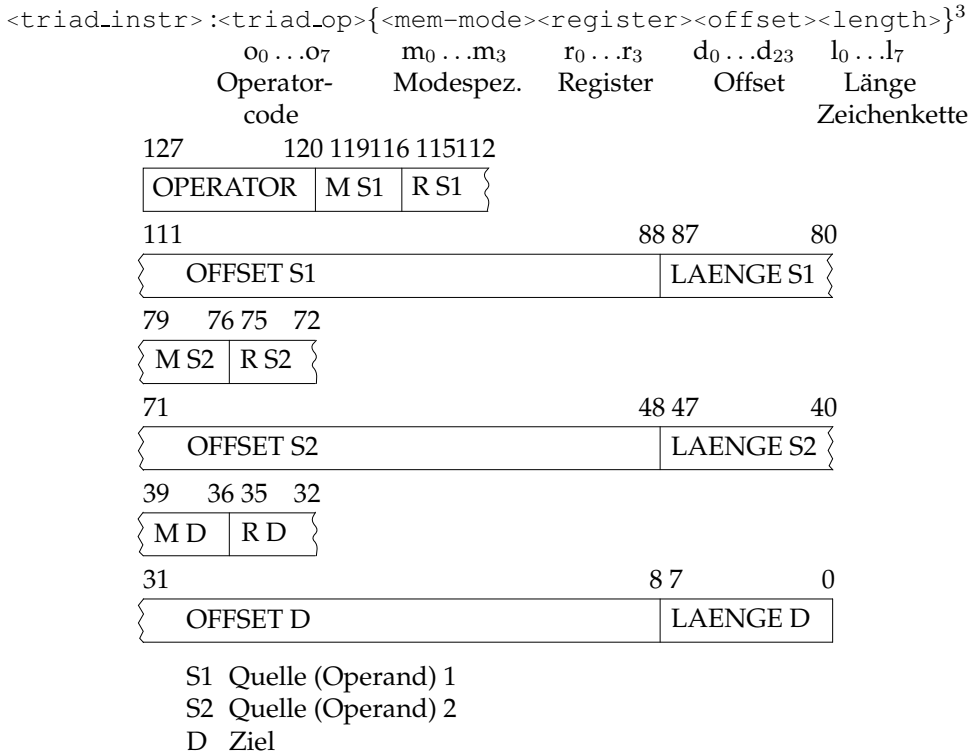
Die Offsets werden im Zweierkomplement angegeben.
Formate der Darstellung der Komponenten:

- Operationscode: 1 Byte
- <reg_mode><register>: 1 Byte
- <mem_mode><register>: 1 Byte
- <short_offset>: 1 Byte
- <long_offset>: 3 Byte
- <length>: 1 Byte

Hieraus ist eine Vielzahl unterschiedlicher Instruktionen konstruierbar, was typisch für eine CISC-Architektur ist (z.B. bei VAX-11/780 etwa 250 Instruktionen).

Beispiele:

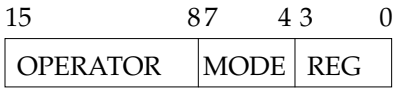
1. Die längste Instruktion ist eine triadische Operation mit long offset auf einer im Speicher abgelegten Zeichenkette: Länge = 16 Byte



S1 Quelle (Operand) 1
S2 Quelle (Operand) 2
D Ziel

2. Die kürzeste Instruktion mit expliziter Adreßangabe ist eine monadische Registermode-Instruktion: Länge = 2 Byte

<monad_instr>: <monad_op> <reg-mode> <register>
$o_0 \ldots o_7$ $m_0 \ldots m_3$ $r_0 \ldots r_3$
Operatorcode Modespezif. Registeradr.



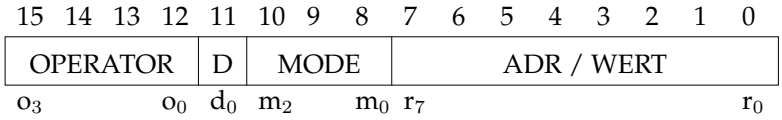
An diesen Beispielen wird deutlich:

- Die Instruktionen haben unterschiedliche Länge.
- Ein byteweiser Zugriff auf den Speicher ist erforderlich.
- Die Decodierung der Befehle benötigt extrem unterschiedliche Taktzeiten, d.h. es ist kein Pipelining möglich!
- Die Befehle zeichnen sich durch eine sehr reichhaltige Strukturierung aus, d.h. auch komplexe Operatoren und vielseitige Adreßmodifikationen sind möglich. Dies bringt evtl. einen kurzen Maschinencode mit sich, geht jedoch zu Lasten einer komplexen und damit zeitaufwendigeren Befehlsinterpretation.

Annahme: Einfaches (altes) Prozessorkonzept als 16-Bit-1-Adreß-Maschine

```
<monad_instr>:<monad_op><double> <mode><rand>
<rand>       :<mem_addr>|<const>
```

Formate der Darstellung der Komponenten:



<monad_op> : 4 Bit
<double> : 1 Bit
<mode> : 3 Bit
<rand> : 1 Byte (bzw. 3 Byte)

Tabelle 4.1 zeigt den Instruktionssatz dieser 1-Adreß-Maschine. Bei zweistelligen Operationen hält der Akkumulator einen der beiden Operanden, der andere Operand wird

OP-Code dual	OP-Code hex	Bedeutung der Operation	Mnemo-code
0000	0	Halt, Ende der Programmabarbeitung	STP
0001	1	Lade Operand in Akku	LAD
0010	2	Speichere Inhalt d. Akku in Speicher	STR
0011	3	Addiere Operand zum Akku	ADD
0100	4	Subtrahiere Operand vom Akku	SUB
0101	5	Multipliziere Operand mit Akku	MUL
0110	6	Dividiere Operand durch Akku	DIV
0111	7	Unbedingter Sprung	JMP
1000	8	Sprung, wenn Inhalt Akku $=$ Null	JEZ
1001	9	Sprung, wenn Inhalt Akku $\geq$ Null	JGZ
1010	A	Sprung, wenn Inhalt Akku $<$ Null	JLZ
1011	B	keine Operation (Reserve)	NOP
1100	C	Kellerbefehl (modif. Aufbau)	*
1101	D	Sprung in Unterprogramm	JSR
1110	E	Rücksprung von Unterprogramm	RET
1111	F	Indexbefehl (modif. MOD-Teil)	*

Tabelle 4.1: Instruktionssatz einer 16-Bit-1-Adreß-Maschine

durch seine Adresse oder seinen Wert angegeben. Das Ergebnis der Operation wird im Akkumulator abgelegt. Dies bedeutet, daß der Akkumulator stets durch Speicherzugriffe ge- bzw. entladen werden muß. Es wird außerdem ein Indexregister zugelassen.

1. *Datentransportbefehle*

   a) R[n] $\rightleftharpoons$ M[addr] (LAD,STR)
   R: hier nur Akku

   b) R[n] $\rightleftharpoons$ R[m] (Indexbefehle)
   R: hier nur Akku bzw. Indexregister

2. *Steuerbefehle*

   Steuerung von Abweichungen vom sequentiellen Programmablauf

   a) unbedingter Sprung (JMP)

   b) bedingter Sprung (JEZ, JGZ, JLZ)

   c) Unterprogrammsprünge (JSR, RET)

3. *arithmetische/logische Befehle*

a) arithmetische Operationen über Integer-Zahlen

hier z.B. `ADD, SUB, MUL, DIV`

b) arithm. Op. über Gleitpunktzahlen (hier nicht realisiert)

c) logische Operationen, bitweise Verknüpfungen der Operanden (hier nicht realisiert)

z.B. `AND, OR, XOR, NOT`

Beispiel:

Operand 1	Operat.	Operand 2	Erg.Datum	Bedeutung
11011100	AND	10001111	10001100	Konjunktion
11011100	OR	10001111	11011111	Disjunktion
11011100	XOR	10001111	01010011	exkl. Disj.
11011100	NOT		00100011	Negation

d) *Schiebebefehle*

Verschieben von Registerinhalten um k Bits nach links oder rechts

- *logischer Shift*: Nachziehen von Nullen
- *arithmetischer Shift*: Nachziehen von vorzeichengleichen Werten
- *zyklischer Shift*: Nachziehen der herausfallenden Stellen

Das Verschieben von Registerinhalten ist auch byteweise oder über Doppelworte möglich. Schiebebefehle finden z.B. zur Multiplikation bzw. Division mit 2^k durch Rechts-/Links-Shift Verwendung. Die Multiplikation kann durch Verschiebe- und Additionsoperationen realisiert werden.

Beispiel:

Operand	Arg.	Erg.Datum	Operat.	Bedeutung
11011110	2	01111000	SHLL	Shift links, logisch
11011110	2	01111000	SHLA	Shift links, arithmetisch
11011110	2	01111011	SHLZ	Shift links, zyklisch
11011110	2	00110111	SHRL	Shift rechts, logisch
11011110	2	11110111	SHRA	Shift rechts, arithmetisch
11011110	2	10110111	SHRZ	Shift rechts, zyklisch

4.3.2 Adressierungsmodi

Die unterschiedlichen Adressierungsmodi ergeben sich aus der Möglichkeit, Operanden oder deren Adressen durch Register- und/oder Speicherzugriff verfügbar zu ma-

chen. In CISC-Architekturen gibt es auch viele Mischtypen, die Register- und Speicherzugriffe im Befehl verbinden. Dabei sind die speziellen Adressierungsarten an typische Datenstrukturen angepaßt, z.B.

- basisrelative Verarbeitung von Listen, Strukturen, ...
- Kellerverarbeitung,

aber auch an die Tatsache, daß die absoluten Adressen compilierter Programme nicht bekannt sind (Notwendigkeit der *Relokatierbarkeit* bzw. *Verschieblichkeit*) und PC-relative Adressierungen vom Compiler generiert werden müssen.
Im folgenden werden unterschieden

- Immediate Adressierung
- Register-Adressierungsmodi: `reg, inc, dec`
- Speicher-Adressierungsmodi: `mem`
- Basis-Adressierung: `rel`
- Index-Adressierung: `idx`

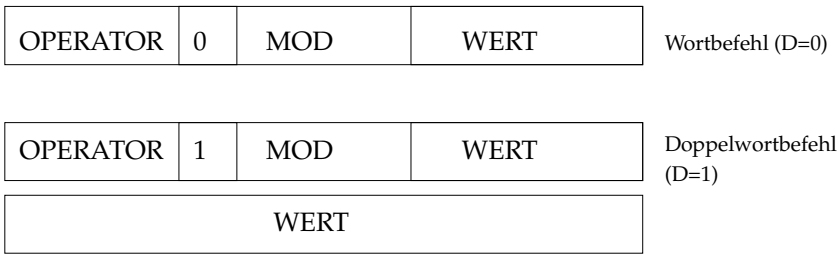
Es gelte folgende Notation zur formalen Definition der Adressierungsmodi:

`R[n]:`	: Inhalt von Register n
`[TOS]`	: Inhalt von TOS
`[PC]`	: Inhalt von PC
`[IR]`	: Inhalt von IR
`M[expr]`	: Speicherinhalt der durch `expr` spezifizierten Adresse
`rand`	: aus einem Register/einer Speicherzelle gelesenes bzw. in diese zu schreibendes Datum
`d`	: Offsets der Register-inkrement/-dekrement Adressierungsmodi als Teil der jeweiligen Operatoren (`d`=1: Byte, `d`=2: Halbwort, `d`=8: Doppelwort)
`D`	: Offset relativ zu einer Basisadresse (positiv oder negativ)
`ADR`	: Adresse im Adreßteil eines Befehls (auch Interpretation als Basisadresse)

4.3.2.1 Immediate Adressierung

Stellt der Operand eine Konstante dar, so bietet sich deren unmittelbare (*immediate*) Codierung in der Instruktion an.
Die Größe der Konstanten ist auf 1 Byte (Wortbefehle) oder 3 Byte (Doppelwortbefehl) begrenzt.

Beispiel:



4.3.2.2 Register-Adressierungsmodi

Register beinhalten die Operanden (`dir`) oder deren Adressen (`ind`). Nützlich ist die automatische Inkrementierung (`inc`) oder Dekrementierung (`dec`) der Speicheradresse, die sich als Registerinhalt darstellt. Diese Variante dient vor allem dem sequentiellen Zugriff auf Felder oder Stacks.

Beispiel: Stack-Adressierung relativ zu TOS

Indirekte Adressierung über Registeradressierung

`dec` realisiert `POP` (entnehmen)
`inc` realisiert `PUSH` (eintragen)

1. `reg_dir : rand ⇌ R[n]`
   Inhalt des Registers ist der Operand (Bitvektor, Zahl, Adresse)
2. `reg_ind : rand ⇌ M[R[n]]`
   Inhalt des Registers ist die Speicheradresse des Operanden
3. `inc_ind : R[n]:=R[n]+d`
   `rand ⇌ M[R[n]]`
   Wirkung wie 2., aber mit vorheriger Erhöhung der Speicheradresse
   (*Präinkrement-Adressierung*)
4. `dec_ind : rand ⇌ M[R[n]]`
   `R[n]:=R[n]-d`
   Wirkung wie 2., aber mit anschließender Reduzierung der Speicheradresse

(*Postdekrement-Adressierung*)

Es sind auch komplexere Adressierungen möglich, z.B.
`reg_ind_ind: rand ⇌ M[M[R[n]]]`
(in Verbindung mit `inc, dec` zur Adressierung von Listen, Feldern).

4.3.2.3 Speicher-Adressierungsmodi

Der Adreßteil des Befehls beinhaltet die Adresse des Operanden (`dir`) oder die Adresse der Adresse des Operanden (`ind`). Speicherindirekte Adressierung wird beispielsweise benutzt, um die Verschieblichkeit von Daten zu garantieren. D.h., deren Adresse wird erst zur Laufzeit des Programms bestimmbar.

Häufig kommt die speicherindirekte Adressierung mit Basis- oder Indexadressierung bzw. mit registerindirekter Adressierung vor (siehe oben).

5. `mem_dir : rand ⇌ M[ADR]`
   Der Inhalt des Adreßteils der Instruktion ist die Adresse eines Operanden. Der Operand wird *absolut* adressiert. Das ist meist nur in Doppelwortbefehlen sinnvoll. Die praktische Bedeutung der absoluten Adressierung ist gering.
6. `mem_ind : rand ⇌ M[M[ADR]]`
   Der Inhalt des Adreßteils der Instruktion ist eine Adresse, deren Speicherzelle die Adresse des Operanden enthält (*effektive Adresse*).
   Anwendungen ergeben sich z.B. beim Zugriff auf die Elemente eines Datenvektors (zusammen mit einer Inkrement-Adressierung), dessen Startadresse bekannt ist.

Für den effizienten Zugriff auf die Startadresse und auf die Komponenten von Datenstrukturen (Vektoren, Matrizen, Listen, Stacks) werden Speichermodus und Registermodus (evtl. mit automatischem Inkrement/Dekrement) verkoppelt. Diese Adressierungen heißen *Basis-* und *Indexadressierung*. In beiden Fällen wird eine Basisadresse mit einem Offset verbunden und damit eine bezüglich der Basis relative Adressierung erreicht.

4.3.2.4 Basisadressierung

Für die *basisrelative* Adressierung wird die in einem Register gehaltene Basisadresse mit dem in der Instruktion gehaltenen Offset direkt oder indirekt zur Ermittlung der effektiven Adresse verknüpft.

7. `rel_dir : rand ⇌ M[R[n]+D]`

Der um den Offset inkrementierte/dekrementierte Inhalt eines Registers (Basisregister) ist die Adresse des Operanden.

8. `rel_ind : rand ⇌ M[M[R[n]+D]]`
   Der um den Offset inkrementierte/dekrementierte Inhalt des Basisregisters ist die Adresse der Adresse des Operanden.

Bei der Kopplung von Basisadresse und speicherindirekter Adressierung erfolgt die Addition des Offsets mit der Basisadresse vor dem Auslesen der Adresse aus dem Speicher (*Präindizierung*).

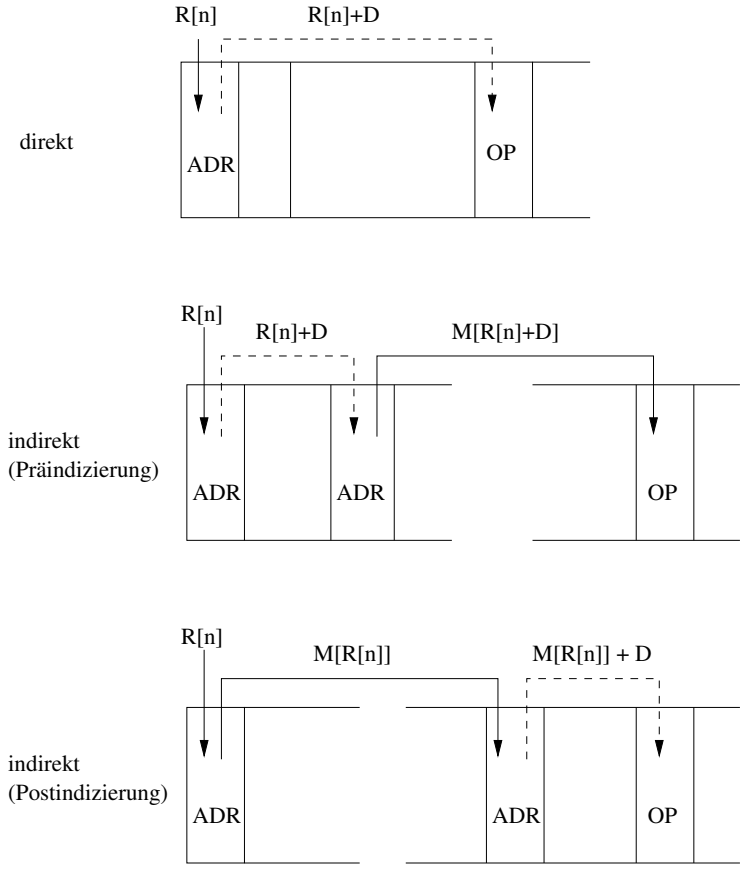


Abbildung 4.14: Arten der Basis-Adressierung

Beispiele:

zu 7. Basisadresse zeigt auf Anfang/Ende einer Tabelle

zu 8. Basisadresse zeigt auf eine Liste von Pointern

Vorstellbar ist auch

8.* `ind_rel : rand ⇌ M[M[R[n]]+D]`
Das Basisregister hält die Adresse der Adresse der Basis (indirekte Basisadresse), die um den Offset inkrementiert/dekrementiert die Adresse des Operanden ergibt (*Postindizierung*).
Beispielsweise könnte hier die Basisadresse als indirekter Verweis mittels Pointer auf eine Liste interpretiert werden.

Aus 8. und 8.* lassen sich mittels Pointern verkettete Listen adressieren. Die Version 8.* ist aber von größerer Bedeutung für die Indexadressierung.
Es werden feste Basisadressen zum direkten Zugriff auf Datenbereiche und variable Basisadressen zum relativen Zugriff auf Daten und Programmbereiche verwendet.
Eine besondere Rolle spielt die Verwendung des PC als Basisregister für

I. die Erzeugung verschieblicher Programme

II. die Definition PC-relativer Sprungmarken.

Dabei ist zu beachten, daß der PC immer auf den nächsten auszuführenden Befehl zeigt.

9. `rel_pc : rand ⇌ M[[PC]+D]`

zu I: Der Assembler übersetzt Hauptprogramme und Unterprogramme separat. Er übersetzt programminterne Adreßangaben relativ zur Startadresse des jeweiligen Programms.

Erst in Verbindung mit den Funktionen von Linker und Lader entstehen hieraus relokatierbare lauffähige Programme. Diese Funktionen werden später (Abschnitt 4.4) behandelt.

zu II: Der Offset für PC-relative Sprünge wird vom Assembler ermittelt. Dabei gilt, daß die Differenz zwischen dem Sprungziel und der Adresse der gerade zu assemblierenden Zeile (Stand des *Lokationszählers* LC) um Eins zu dekrementieren ist.

Für die Stackverwaltung kommt die TOS-relative Adressierung, u.U. verbunden mit automatischem Inkrement/Dekrement des Stackzählers (siehe 3. und 4.) zur Anwendung. Der Zugriff auf Stacks beschränkt sich nicht auf Kellerbefehle (`PUSH, POP`), z.B. bei der Modifikation von Laufzeitstacks.

4.3.2.5 Index-Adressierung

Eine basisrelative Adresse bildet man aus der Basisadresse und dem Offset (auch *Index* genannt). Basis-Adressierung und Index-Adressierung verhalten sich in folgender Weise dual zueinander:

Basis-Adressierung: Das spezifizierte Register hält die Basis. Der Index wird explizit im Befehl angegeben (neben der Registerbezeichnung).

Index-Adressierung: Das spezifizierte Register hält den Index (Offset). Als Basis dient die bisher modifizierte Adresse. Diese kann aus Speicher- oder Registeradressierung hervorgehen.

10. `idx_dir : rand ⇌ M[ADR+[IR]]`
    Der um den Inhalt des Indexregisters modifizierte Adreßteil der Instruktion ist die Adresse des Operanden.
11. `idx_ind : rand ⇌ M[M[ADR]+[IR]]`
    Die indirekt adressierte Basis wird um den Offset des Indexregisters modifiziert, um die Adresse des Operanden zu ergeben.
    Typisch für indirekt indizierte Adressierung ist, daß die Index-Modifizierung der Adresse nach dem Ermitteln der effektiven Basisadresse erfolgt (Postindizierung).

Damit bietet sich die Indexadressierung zur Adressierung von Feldelementen oder Tabelleneinträgen an, ohne daß die Basis mit der Basis des Feldes/der Tabelle übereinstimmen muß.
In Tabelle 4.2 sind noch einmal sämtliche Adressierungsmodi zusammengefaßt.

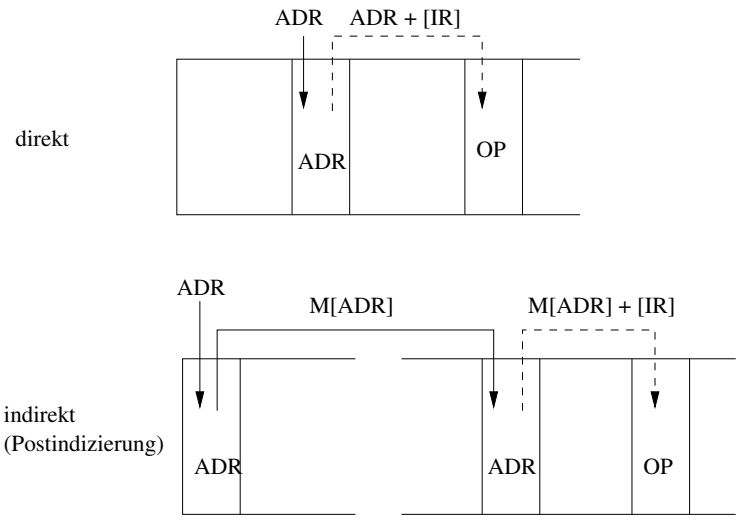


Abbildung 4.15: Arten der Index-Adressierung

1	reg_dir	`rand`	$\rightleftharpoons$	`R[n]`	`Rn`
2	reg_ind	`rand`	$\rightleftharpoons$	`M[R[n]]`	`(Rn)`
3	inc_ind	`rand`	$\rightleftharpoons$	`R[n]:=R[n]+d; M[R[n]]`	`+(Rn)`
4	dec_ind	`rand`	$\rightleftharpoons$	`M[R[n]]; R[n]:=R[n]-d`	`(Rn)-`
5	mem_dir	`rand`	$\rightleftharpoons$	`M[ADR]`	`@ADR`
6	mem_ind	`rand`	$\rightleftharpoons$	`M[M[ADR]]`	`(ADR)`
7	rel_dir	`rand`	$\rightleftharpoons$	`M[R[n]+D]`	`(Rn)D`
8	rel_ind	`rand`	$\rightleftharpoons$	`M[M[R[n]+D]]`	`((Rn)D)`
9	rel_pc	`rand`	$\rightleftharpoons$	`M[[PC]+D]`	`(PC)D`
10	idx_dir	`rand`	$\rightleftharpoons$	`M[ADR+[IR]]`	`@ADR(Rn)`
11	idx_ind	`rand`	$\rightleftharpoons$	`M[M[ADR]+[IR]]`	`(ADR)(Rn)`

Tabelle 4.2: Adressierungsmodi (Übersicht)

Vereinfachtes Adressierungsschema der 1-Adreß-Maschine

Außer den angegebenen 11 Adressierungsmodi haben CISC-Prozessoren weitere Möglichkeiten, Mischtypen zu bilden. Im Falle von RISC-Prozessoren wird wegen der reduzierten Bedeutung von Speicherzugriffen auch eine bedeutende Vereinfachung der Adreßmodifikation vorgenommen. Oft existieren nur eine oder zwei Adressierungsarten für Speicherzugriffe.

Für den eingeführten Modellprozessor existieren nur 3 Bits für Adreßmodifikationen, es gibt also 8 verschiedene Adressierungen (Tabelle 4.3). Darüber hinaus haben Kellerbefehle und Indexbefehle einen speziellen Aufbau. Die Maschine kennt als Register nur AC, PC, IR und TOS. Durch die oben besprochene duale Adressierung von Index (Offset) und Basis kann das Indexregister hier auch als Basisregister verwendet werden. Auf die Spezifizierung der Kellerbefehle wird hier verzichtet.

Die einfache Art, im Mnemocode für die Adressierungsart zwischen Basis- und Indexadressierung zu unterscheiden, funktioniert nur, weil hier lediglich ein Register zur Unterscheidung der Varianten nutzbar ist. Für den allgemeinen Fall sollen die Codierungen der Tabelle 4.3 gelten. Dabei gelte im Mnemocode die Reihenfolge Basis–Offset.

	Adreßmode	MOD	Mnemocode	Bemerkung
0	unmittelbar	000	`#WERT`	
1	mem_dir	001	`@ADR`	
2	mem_ind	010	`(ADR)`	
3	rel_dir	011	`BR.ADR`	Indexregister trägt Basisadresse
4	rel_ind	100	`(BR.ADR)`	
5	rel_pc	101	`PC.ADR`	
6	idx_dir	110	`IR.ADR`	Indexregister trägt Index
7	idx_ind	111	`(IR.ADR)`	

Tabelle 4.3: Adreßmodi der 1-Adreß-Maschine

Für die einfache Machine muß mit speziellen Befehlen dafür gesorgt werden, daß das Indexregister mit den geeigneten Daten versorgt wird. Hierfür sind die Möglichkeiten in der Tabelle 4.4 spezifiziert.

Op-Code	MOD	Mnemocode	Funktion
1111	001	`CLI`	Lade 0 in IR
1111	010	`LDI`	Lade [AC]→[IR]
1111	011	`LIA`	Lade [IR]→[AC]
1111	100	`INK #N`	Inkrementiere [IR]:=[IR]+N
1111	101	`DEK #N`	Dekrementiere [IR]:=[IR]-N
1111	110	`LDX @ADR`	Lade Inhalt Speicheradr. in IR (dir.)
1111	111	`LDX (ADR)`	Lade Inhalt Speicheradr. in IR (ind.)

Tabelle 4.4: Indexbefehle

Beispiel M2

In Beispiel M2 aus Abschnitt 4.2.1 haben wir gelernt, daß die Übersetzung eines Terms in eine Folge elementarer Terme je nach den Architekturprinzipien und Ressourcen des verfügbaren Prozessors zu sehr unterschiedlichen Befehlsfolgen führt. Zu den Ressourcen zählt auch der Adressierungsmodus. Hier soll die Variante M2/5 des Beispieles aufgegriffen werden, welche die Termberechnung auf einer 1-Adreß-Maschine darstellt. Deren Adreßmodi sind in Tabelle 4.3 dargestellt.

Es sei angenommen:

- Der Maschinencode beginne bei der hexadezimalen Adresse $10.
- Es gibt nur 1-Wort-Befehle.
- Die Termvariablen bilden einen Variablenvektor bei $60.
- Die Hilfsvariablen h[i] bilden einen Vektor bei $70. Es sei hier nicht ausgenutzt, daß die Hilfsvariablen als Stack angesprochen werden können.
- Die Basisadressen beider Variablenvektoren finden sich bei $50 bzw. $51. Diese Adressen können für speicherindirekte Zugriffe verwendet werden.

In den drei Varianten der Adressierungsmodi

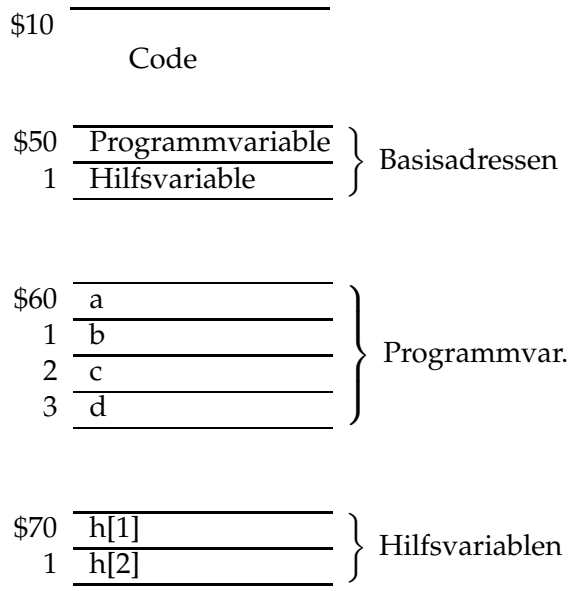
1. direkter Speichermodus
2. direkte Basisadressierung
3. indirekte Indexadressierung

wird die symbolische Befehlsfolge der 1-Adreß-Maschine auf jeweils eine nach dem verwendeten Modus spezifizierte Befehlsfolge abgebildet.

Beispiel: M2/6: Termberechnung mit 1-Adreß-Maschine

$$t = \frac{a * b + (a + d) * c}{a + b * b}$$

1. `AC:=a;`
2. `AC:=AC*b;`
3. `h[1]:=AC;`
4. `AC:=a;`
5. `AC:=AC+d;`
6. `AC:=AC*c;`
7. `AC:=h[1]+AC;`
8. `h[1]:=AC;`
9. `AC:=a;`
10. `h[2]:=AC;`
11. `AC:=b;`
12. `AC:=AC*b;`
13. `AC:=h[2]+AC;`
14. `AC:=h[1]/AC;`



Variante optimierter symbol. Code statt (9)-(14):

9*. `AC:=b;`

10*. `AC:=AC*b;`

11*. `AC:=AC+a;`

12*. `AC:=h[1]/AC;`

	1. Memory-Mode (direkt)	2. Basis-Adressierung	3. Index-Adresierung
1.	`LAD @$60`	`LDX @$50` `LAD BR.#0`	`CLI` `LAD (IR.$50)`
2.	`MUL @$61`	`MUL BR.#1`	`INK #1` `MUL (IR.$50)`
3.	`STR @$70`	`LDX @$51` `STR BR.#0`	`DEK #1` `STR (IR.$51)`
4.	`LAD @$60`	`LDX @$50` `LAD BR.#0`	`LAD (IR.$50)`
5.	`ADD @$63`	`ADD BR.#3`	`INK #3` `ADD (IR.$50)`
6.	`MUL @$62`	`MUL BR.#2`	`DEC #1` `MUL (IR.$50)`
7.	`ADD @$70`	`LDX @$51` `ADD BR.#0`	`DEC #2` `ADD (IR.$51)`
8.	`STR @$70`	`STR BR.#0`	`STR (IR.$51)`
9.	`LAD @$60`	`LDX @$50` `LAD BR.#0`	`LAD (IR.$50)`
10.	`STR @$71`	`LDX @$51` `STR BR.#1`	`INK #1` `STR (IR.$51)`
11.	`LAD @$61`	`LDX @$50` `LAD BR.#1`	`LAD (IR.$50)`
12.	`MUL @$61`	`MUL BR.#1`	`MUL (IR.$50)`
13.	`ADD @$71`	`LDX @$51` `ADD BR.#1`	`ADD (IR.$51)`

14.	`JEZ #1`
	`DIV @$70`
	
	`STP`
		`JEZ #1`
		`DIV BR.#0`
		
		`STP`
			`JEZ #2`
			`DEK #1`
			`DIV (IR.$51)`
			`STP`

Dabei wurde keine Codeoptimierung für den durch den linearisierten Kantorovič-Baum gelieferten Code berücksichtigt. Die Kommutativität der Summe im Nenner des Terms gestattet eigentlich, nur mit einer Stackvariablen auszukommen und zwei Elementarterme einzusparen. Der Vergleich der Befehlsfolgen für die drei Adressierungsmodi zeigt:

- die direkte Adressierung liefert den kürzesten Code,

- die Flexibilität von Basis- und Index-Adressierung kann bei der 1-Adreß-Maschine nicht auch zur Verkürzung des Codes genutzt werden,
- bei der Basisadressierung muß das Indexregister ständig umgeladen werden, die Verwendung von Stackbefehlen würde dies vermeiden,
- bei der Indexadressierung beziehen sich die Befehle zur Aktualisierung des Indexes auf die eine oder andere Basis. Das erschwert das Lesen des Codes.

Einen kompakten Assemblercode erhält man für die Termberechnung, wenn ein Drei-Adreß-Rechner mit 16 Registern vorausgesetzt wird. Die arithmetischen Operationen sollen register-orientiert formuliert sein. Der Binärcode des Maschinenprogramms ist länger als der für die 1-Adreß-Maschine, da register-orientierte 3-Adreßbefehle jeweils 96-Bit-Instruktionen darstellen:

OP-Code	: 8 Bits
Modus	: 4 Bits
Register-Source/Destination	: je 4 Bits
Offset-Source/Destination	: je 24 Bits.

Beispiel: M2/7: Termberechnung mittels Drei-Adreß-Maschine (Register-Modus, direkte Basisadressierung)

16 Register: R0 : Nullregister R[0]=0
R1 : Basisregister → Basisadressierung
R10 : Fehlerregister (arithmetischer. Fehler)

```
0.          MOVE  R0,R10            ; R[10] bedeutet OK
            MOVE  @VAR, R1          ; R[1] hält Basisadresse
1.          MUL   (R1)0, (R1)1, R2  ; R[2]=a*b
2.          ADD   (R1)0, (R1)3, R3  ; R[3]=a+d
3.          MUL   R3, (R1)2, R3     ; R[3]=(a+d)*c
4.          ADD   R2, R3, R2        ; R[2] ~ Zähler fertig
5.          MOVE  (R1)0, R3         ; R[3]=a
            MUL   (R1)1, (R1)1, R4  ; R[4]=b*b
6.          ADD   R3, R4, R3        ; R[3] ~ Nenner fertig
            JEZ   R3, #ERROR        ; Nenner auf Null testen
7.          DIV   R2, R3, R2        ; R[2] hält Ergebnis
            STP                     ; Haltepunkt für OK
    ERROR   MOVE  #ERFLG, R10       ; R[10]=$1000
            STP                     ; Haltepunkt für Fehler
    ERFLG   EQU   $1000             ; Fehlerkennung
    VAR     RES   W, 4              ; Variablenpuffer (4 Werte)
```

Der `MOVE`-Befehl ist ein allgemeiner Transport-Befehl zwischen Quelle und Ziel. Die `EQU`-Anweisung ist eine Zuordnungsanweisung, die keinen Maschinencode erzeugt. Die `RES`-Anweisung ist eine Datenanweisung zur Reservierung von Speicherplatz. Unter der Adresse `VAR` werden an diesen Stellen die Variablen `a`, `b`, `c`, `d` gespeichert.

4.4 Assemblerprogrammierung

Unter *Assemblerprogrammierung* soll eine mnemonisch codierte Problemformulierung verstanden werden, die eine (1:1)-Abbildung auf einen Maschinen-Code erlaubt.

Außer für echtzeitfähige oder gerätenahe Programmierung spielt die Assemblerprogrammierung für den Nutzer heute keine Rolle mehr. Assembler-Code dient aber durchaus als Zielsprache für Compiler (Übersetzer für höhere Programmiersprachen).

Folgende Aspekte werden behandelt:

- Prinzipien der Problemformulierung in Assemblersprache, insbesondere die Anwendung symbolischer Adressierung
- Funktionsweise des *Assemblers* (des Übersetzers in Maschinen-Code)
- Funktionsweise des *Binders* (die Verschmelzung von Programmsegmenten und die Berechnung externer Adreßbezüge)
- Funktionsweise des *Laders* (bezüglich der Generierung verschieblicher Adreßbezüge) (s. Abb. 4.16)

Damit aus dem Assembler-Code ein ausführbarer Maschinen-Code entsteht, werden als Zwischen-Codes

- der *Binder-Code* (Ausgabe des Assemblers, Eingabe des Binders) und
- der *Lader-Code* (Ausgabe des Binders, Eingabe des Laders)

erzeugt. Im Gegensatz zu den von einem Compiler erzeugten Zwischen-Codes (siehe

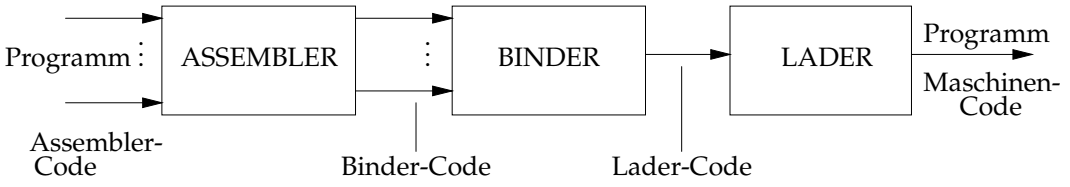


Abbildung 4.16: Umcodierung mittels Assembler, Binder und Lader

Abbildung 4.17) unterscheiden sich Binder-Code und Lader-Code vom ausführbaren Maschinen-Code nur bezüglich der Adreßspezifikationen.

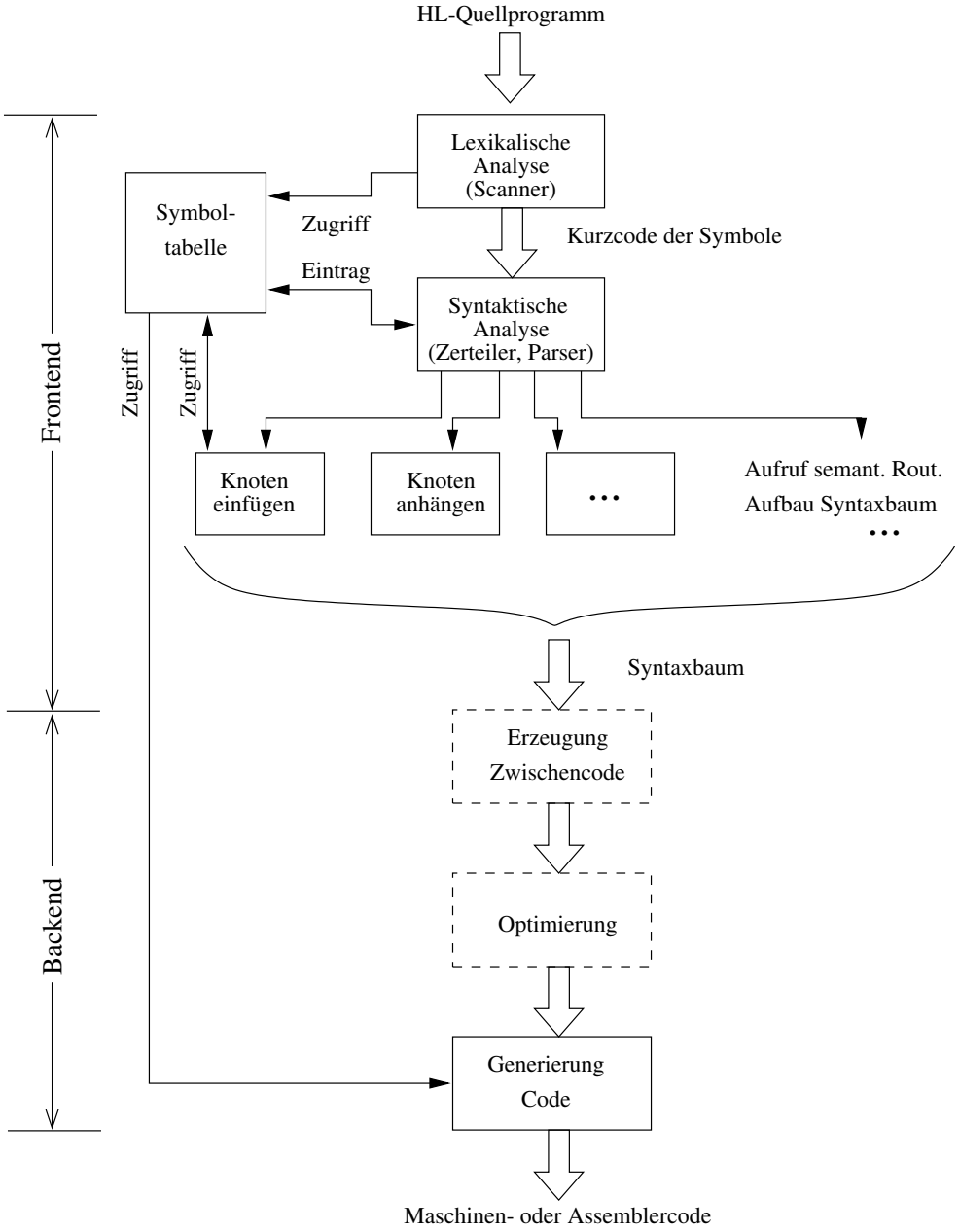


Abbildung 4.17: Ablauf der Compilation

Einschub:Ablauf der Compilation (zu Abbildung 4.17)

Zunächst soll der in Abbildung 4.17 skizzierte grundsätzliche Ablauf einer Compilation erläutert werden. Manche Verfahrensschritte findet man in vereinfachter Form auch in einem Assembler wieder.
Ein Compiler kann als Hintereinanderschaltung zweier Funktionskomplexe, des Frontend und des Backend, verstanden werden. Das Frontend dient der Analyse des von der verwendeten Programmiersprache abhängigen Quellprogramm-Codes. Das Backend erzeugt einen von der Zielmaschine (Prozessor) abhängigen Objektcode in Form von Maschinen- oder Assemblercode.
Diese Auftrennung erlaubt die Kombination verschiedener Frontends mit verschiedenen Backends, um Übersetzer für verschiedene Sprachen und Prozessoren aus einer Grundstruktur zu entwickeln.
Das Frontend führt eine lexikalische und syntaktische Analyse, sowie die Überprüfung des Einhaltens gewisser semantischer Regeln durch.

Aufgaben der lexikalischen Analyse (im wesentlichen):

- Aufbau einer Symboltabelle aus dem Quellcode als Folge von Symbolen (Namen, Zahlen, Sonderzeichen, ...). Für jedes Symbol Erfassen des numerischen Symbolcodes, des Symbolwertes und der Symbolposition.
- Bedeutungslose Zeichen (Leerzeichen) und Kommentare werden überlesen.
- Prüfung auf lexikalische Fehler.

Unter lexikalischen Symbolen versteht man Zeichenfolgen des Quellprogramms, die aus syntaktischer Sicht nicht weiter zerlegt werden können (z.B. Namen oder Zahlen). Da sie von einfacher Struktur sind, können sie meist durch reguläre Grammatiken im Aufbau beschrieben und durch endliche Automaten erkannt werden.

Aufgaben der Syntaxanalyse:

- Prüfung der syntaktischen Korrektheit des Quellprogramms.
- Zerlegung der von der lexikalischen Analyse gelieferten Symbolfolgen in syntaktische Einheiten (z.B. Deklarationen, Definitionen, Anweisungen, Ausdrücke) und Aufbau eines Syntaxbaums (Kantorovič-Baum). Der Aufbau des Syntaxbaums erfolgt oft nur virtuell.

Die aus den Symbolen gebildeten Sätze haben gewissen Grammatiken unterworfene Strukturen. Die Identifikation dieser Sätze als Graphstrukturen erfolgt top-down

(mit Startsatzsymbol der Grammatik beginnend, bis der entstehende Satz dem Quellprogramm entpricht) oder bottom-up (Entwicklung des Syntaxbaums des Quellprogramms mit einem Satzsymbol an der Spitze des Baums aus Symbolfolgen und Teilbäumen). Die in Abbildung 4.17 angegebenen Knoten des Syntaxbaums beziehen sich auf Objekt-Knoten für jedem deklarierten Namen und Struktur-Knoten für jeden verwendeten Datentyp.

Aufgaben der Semantikanalyse:

- Prüfung des Einhaltens semantischer Regeln (Kontextbedingungen), also Beachtung der Gültigkeitsbereiche von deklarierten Namen.

- Aufbau von Datenstrukturen (Symbollisten, Zwischencode), die zur Codeerzeugung erforderlich sind.

Hierfür kommen attributierte Grammatiken zur Anwendung. Die Attribute beschreiben Eigenschaften von Symbolen des Syntaxbaums.
Auf die Schritte des Frontends wollen wir hier nicht näher eingehen.

Ein *Assembler* muß ähnlich wie ein Compiler eine lexikalische und eine syntaktische Analyse des Assembler-Codes durchführen. Anders als bei der Übersetzung höherer Programmiersprachen muß aber kein Syntaxbaum erzeugt werden. Dieser dient letztlich der Sequentialisierung des Maschinencodes. Assembler-Code ist bereits sequentialisiert.

Die Hauptaufgabe eines Assemblers besteht in der Transformation symbolischer Adreßangaben für Operanden in numerische Adressen eines *logischen Adreßraums*. Hierfür erfordert er zwei Pässe. Andernfalls wären die Einschränkungen in der Programmierung erheblich. Abbildung 4.18 zeigt die Struktur eines Assemblers. Der Assembler übersetzt jedes Segment (Modul) eines Programms getrennt von den anderen. Der logische Adreßraum bezieht sich also auf den Anfang eines Segments. Die hierbei auftretenden segmentübergreifenden Adreßbezüge werden vom *Binder* aufgelöst, indem er die Sequenz aller hintereinander verketteten Segmente bezüglich der Anfangsadresse des Hauptprogramms adressiert (s. Abb. 4.19). Er bildet die Operandenadresse in einen *virtuellen Adreßraum* ab. Dabei werden auch Adreßbezüge zu Systemroutinen hergestellt (z.B. zur Standardbibliothek von UNIX), ohne daß der Lader-Code diese Routinen enthalten muß (kann aber).

Der *Lader* bildet schließlich den virtuellen Adreßraum auf den *physischen realen Adreßraum* des Hauptspeichers ab. Der Lader ist als Teil des Betriebssystems zu sehen.

Es werden die Varianten

a) *dynamische Programmallokation* und

b) *statische Programmallokation*

unterschieden.

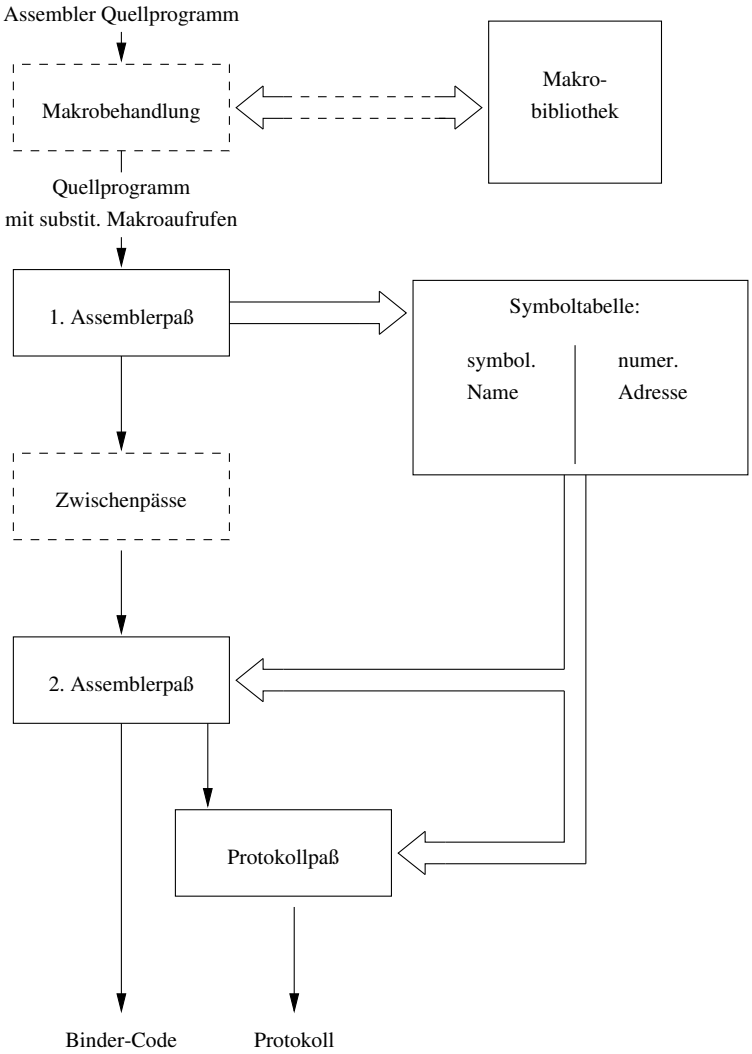


Abbildung 4.18: Struktur eines Assemblers

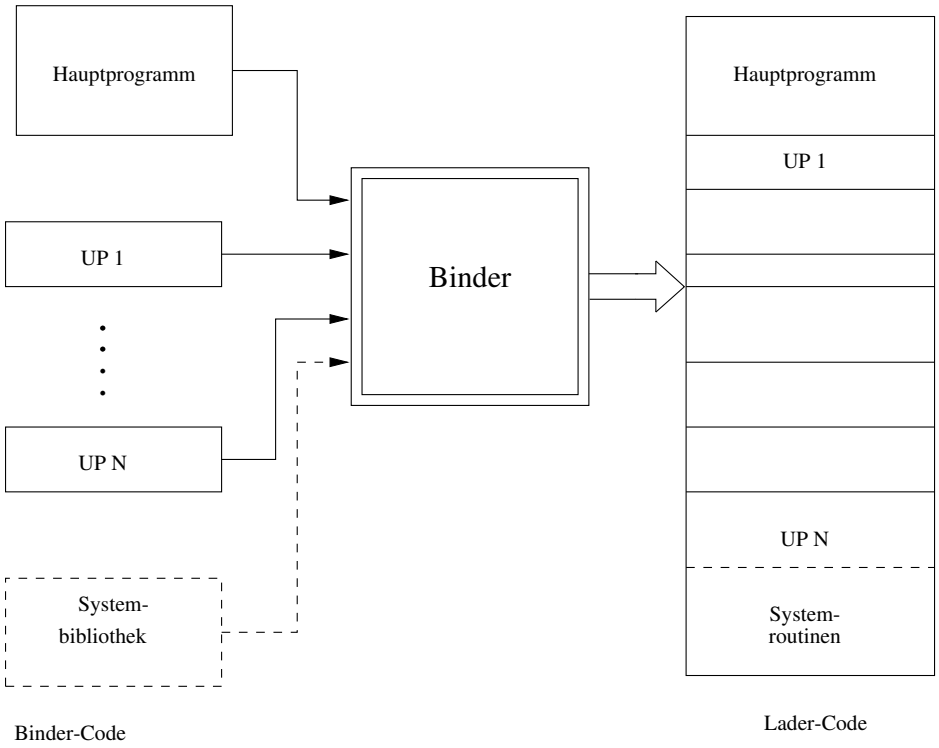


Abbildung 4.19: Binden eines Programms

Sie unterscheiden sich dadurch, daß die Auflösung der realen Adreßbezüge im voraus (statisch) oder zum Zeitpunkt des Aufrufes des Programmes (dynamisch) erfolgt. Der reale Adreßraum ist PC-relativ definiert. Die Ladeadresse des Programmes wird mit der Startadresse des Lader-Codes gleichgesetzt.

Insgesamt ergeben sich also folgende zu realisierende Adreßabbildungen:

1. Assembler:	symbolisch	$\longrightarrow$	logisch	(segmentrelativ)
2. Binder:	logisch	$\longrightarrow$	virtuell	(programmrelativ)
3. Lader:	virtuell	$\longrightarrow$	real	(PC-relativ)

4.4.1 Assemblersprache

Assemblerprogramme bestehen aus

- Maschinenbefehlen in symbolischer Darstellung von Operatoren und Operanden,
- Pseudobefehlen (Assemblerdirektiven), die keine ausführbaren Oprationen codieren,
- Kommentaren, die ebenfalls vom Assembler unterdrückt werden.

Maschinenbefehle sind zur Ausführungszeit/Laufzeit (run time) vom Prozessor ausführbare Operationen. Sie treten nach der Assemblierung bzw. nach dem Binden in binär codierter Form auf und stehen zur Ausführungszeit im Speicher.
Assemblerdirektiven sind Instruktionen für den Assembler, die während der Assemblierung ausgeführt werden. Sie erzeugen zwar keine Befehlswörter im Lader-Code und stehen auch nicht zur Ausführungszeit im Speicher zur Verfügung, können aber an den Lader mitzuteilende Informationen darstellen. Dies betrifft beispielsweise die Reservierung von Bereichen zur Zwischenspeicherung von Daten oder zum Hinterlegen von Konstanten.
Kommentare im Klartext dienen nur zur Dokumentation.
Die Assemblersprache wird

- durch die durch den Prozessor vorgegebenen Maschinenbefehle und
- durch die durch den Assembler vorgegebenen Datenbefehle und andere Assemblerdirektiven

sowie durch die Vorgaben zu ihrer Formulierung und Nutzung gebildet.
Die Syntax der Codierung von Maschinenbefehlen (Operator, Operanden, Ziel) haben wir im wesentlichen bereits kennengelernt. Im Gegensatz zur Maschinensprache werden Adreßbezüge oder Wertangaben aber symbolisch codiert. Zusätzlich zum obligatorischen (assemblerabhängigen) Code der Adressierungsmodi kann der Programmierer relativ freie Namen für Adressen und Daten wählen.

4.4.1.1 Struktur von Programmzeilen

Assemblerprogramme sind zeilenweise strukturiert. Jede Programmzeile enthält entweder einen Maschinenbefehl oder eine Assemblerdirektive. Jede Programmzeile setzt sich aus einer Folge von *Feldern* variabler Länge zusammen (vgl. Abb. 4.20). Diese Felder werden durch ein oder mehrere Leerzeichen getrennt. Die Zeile wird mit CR (Carriage Return - Wagenrücklauf) abgeschlossen.

Es gilt folgende Reihenfolge der Felder (von links nach rechts):

- Namensfeld (Label-Feld, Markenfeld)
- Operationsfeld
- Adreßfeld/Datenfeld
- Kommentarfeld.

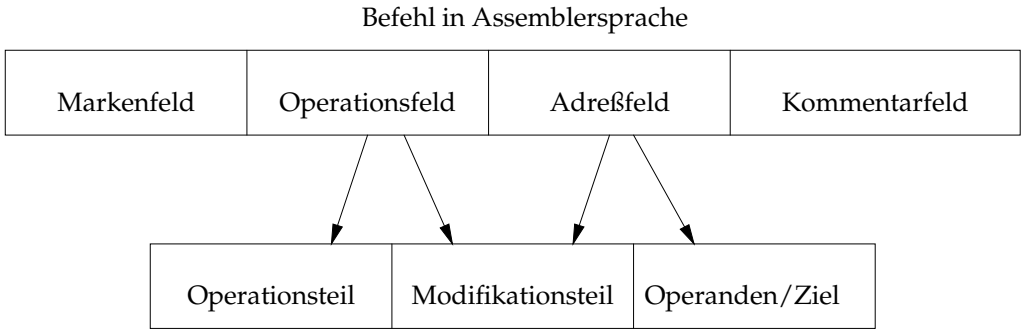


Abbildung 4.20: Felder einer Programmzeile

Beispiel:

Beispiel einer Maschineninstruktion:

```
ERROR  MOVE  @ERFLG,R10  ; R[10]=$1000
```

Beispiel einer Assemblerdirektive:

```
ERFLG  EQU  $1000
```

Anstelle des Adreßfeldes für Operanden steht hier ein Datenfeld.

Marken/Label können aus beliebigen Kombinationen von Buchstaben und Zahlen bestehen, solange sie nicht identisch sind mit dem im Operationsfeld möglichen Mnemocode oder mit einem ($*$) beginnen.

Kommentare, die eine ganze Zeile beanspruchen, sollen mit einem Doppelstern ($**$) beginnen. Ansonsten wird das Kommentarfeld mit einem Semikolon vom Adreßfeld abgetrennt.

Marken und Kommentare sind optional.

4.4.1.2 Assembler-Direktiven

Ein Assemblerprogramm setzt sich aus den Segmenten $S_1, \ldots, S_p$ zusammen. Jedes Segment wird separat assembliert. Alle Segmente werden vom Binder zu einem Lader-Code zusammengebunden.

a) `SEGMENT` und `END`

Die segmentrelative Adressierung beginnt mit der Direktive `SEGMENT` und endet mit der Direktive `END`. Zwischen `SEGMENT` und `END` befinden sich Bereiche mit Maschinenbefehlen sowie Bereiche zur Reservierung von Pufferbereichen und zur Speicherung von Konstanten (speicherrelevant, erzeugen jedoch keinen Maschinencode). Außerdem finden sich Zuordnungsanweisungen und Markendeklarationen, die beide weder speicher- noch coderelevant sind.

Beispiel:

			coderelevant	speicherrelevant
	SEGMENT		N	N
	PUBLIC	AD1, AD2	N	N
	EXTERN	EAD1, EAD2	N	N
TAST	EQU	$20	N	N
MON	EQU	$30	N	N
ALPHA	DAT	W, $1000B6F0	N	J
BETA	DAT	H,1,2,3,4	N	J
GAMMA	RES	W,4	N	J
START	LAD	@ALPHA	J	J
⋮	⋮	⋮	⋮	⋮
AD1	···	EAD1	J	J
AD2	···	EAD2	J	J
⋮	⋮	⋮	⋮	⋮
	END		N	N

Code- bzw. speicherrelevante Bereiche können an beliebiger Stelle eines Segmentes stehen. Der erste ausführbare Maschinenbefehl ist identisch mit dem ersten coderelevanten Bereich.

b) `PUBLIC` und `EXTERN`

Die `PUBLIC`-Direktive veranlaßt den Assembler, ein segmentspezifisches *public symbol dictionary* (*PSD*) zu erzeugen. Als `PUBLIC` werden die Marken des Seg-

mentes j deklariert, welche von anderen Segmenten $i \neq j$ referenziert werden können.

Umgekehrt werden als `EXTERN` die Marken in Segment j deklariert, die innerhalb j referenziert werden, aber in Segmenten $i \neq j$ definiert sind. Für diese legt der Assembler ein segmentspezifisches *external symbol dictionary* (*ESD*) an. Die Information über das Vorkommen von externen Marken muß der Assembler im Programmsegment j hinterlegen, damit durch den Binder die gültigen Adreßbezüge hergestellt werden können.

c) `EQU`

Die Assembleranweisung

```
S  EQU  A
```

ist eine Zuordnungsanweisung oder Wertzuordnung für das Symbol `S` durch den Wert des Ausdrucks `A`. Das durch die `EQU`-Anweisung definierte Symbol `S` kann als symbolische Adresse oder als absolute Konstante verwendet werden. Der Ausdruck `A` kann absolut oder relativ angegeben werden. Damit nimmt auch die symbolische Adresse einen absoluten oder relativen Wert an.

Die Angabe des Wertes von `A` erfolgt durch

- dezimale Zahl: `30`
- hexadezimale Zahl: `$1E`
- binäre Zahl: `%00011110`
- ASCII-Zeichen: `'0'`
- String: `'ABC'`
- Stand des PC: `*`

Diese Klassen von Wertangaben findet man auch in anderen Assemblerdirektiven oder als Konstanten von Maschinenbefehlen.

Beispiel:

Hier wird die `EQU`-Anweisung zur Definition von Konstanten bzw. Adressen verwendet.

```
BASIS  EQU  $1000    ; Basisadresse
INK    EQU  21       ; Wert einer Konstanten als Offset
NUM    EQU  'A'      ; Wert der Konstanten 'A'=$41
SHIFT  EQU  8        ; Wert einer Konstanten
M1     EQU  *        ; Einsprungadresse durch PC gegeben
       LDX  #BASIS   ; Lade Basisadr. nach Indexregister
```

```
        LAD  BR.#INK  ; Lade Akku mit Inhalt BASIS+INK
**                      M[BASIS+INK]≙$42
        SHLL #SHIFT   ; logische Verschiebung nach links
**                      um 8 Stellen
        ADD  #NUM     ; [AC]=$4241='BA'
        ⋮
        JEZ  #M1      ; Sprung nach M1, wenn [AC]=%0
        ⋮
```

Da die EQU-Anweisung nicht speicherrelevant ist, werden die Operanden unmittelbar adressiert.

---

d) DAT

Die Datenanweisung

```
[S]  DAT  A
```

generiert eine durch den Ausdruck A spezifizierte Konstante oder Liste von Konstanten. Die Markierung des durch A spezifizierten und mit konstantem Inhalt belegten Speicherplatzes kann durch die Angabe von S erfolgen. Der Datenbereich kann aber auch über eine Adreßrechnung angesprochen werden. Im Falle einer Liste von Komponenten gibt S die Adresse der ersten Konstanten in der Liste an.

Spezifizierung von A: <A>:<B|H|W|D>,<n_1>,...,<n_m> mit $n_i \in \mathbb{N}, \mathbb{Z}(, \mathbb{R})$

Spezifizierung von n_i: entsprechend den Formaten für A in der EQU-Anweisung

Im Unterschied zur Konstantendeklaration mittels der EQU-Anweisung verbraucht die DAT-Anweisung Speicherplatz. Die Inhalte sind damit auch änderbar.

Beispiele:

```
A   DAT   W,$B6F01248          ; 1 Wort
    DAT   H,$1000              ; 1 Halbwort
    DAT   H,32000              ; 1 Halbwort
B   DAT   B,1,2,3              ; 3 Bytes
C   DAT   B,'PAPA'             ; 4 Bytes
oder
C   DAT   W,$50415041          ; 1 Wort
oder
C   DAT   H,$5041,$5041        ; 2 Halbworte
```

```
DAT   B,%11,%1100,%11000000; 3 Bytes
⋮
LAD   @B                   ; [AC]=1
oder
LDX   #B                   ; [IR]=ADR B
LAD   BR.#0                ; [AC]=1
```

Die mit der Marke `C` symbolisierte `DAT`-Anweisung bzw. der mit der Marke `C` adressierte Datenbereich enthält den ASCII-String `'PAPA'`. Jedes ASCII-Zeichen benötigt 1 Byte. Also werden 4 Bytes mit dem entsprechenden Hexadezimal- bzw. Binärcode belegt.
Datenbereiche werden direkt, indirekt oder relativ adressiert.

---

e) `RES`

Die Datenanweisung

```
[S] RES A
```

reserviert einen durch den Ausdruck `A` spezifizierten unbelegten Datenbereich. Dieser Datenbereich ist speicherrelevant, seine Belegung nach dem Ladevorgang ist aber undefiniert. Erst die Programmausführung definiert die Inhalte. Bezüglich der Marke `S` gilt das gleiche wie bei der `DAT`-Anweisung.

Spezifizierung von `A`: `<A>:<B|H|W|D>,<n>` mit $n \in \mathbb{N}$,
n gibt die Anzahl der Pufferzellen vom Typ `B,H,W` oder `D` an (monomorph getypt).

Beispiel:

```
BLOCK   RES   D,10  ; 10 Doppelworte werden reserviert
        ⋮
        LAD   @BLOCK
```

4.4.1.3 Sprünge und Schleifen

Assemblersprachen enthalten gewöhnlich keine komfortablen Befehle zur Steuerung des Programmablaufes. Unbedingte und bedingte Sprünge sind jedoch obligatorisch. Bedingte Sprünge werden darauf zurückgeführt, daß ein Register auf einen Status hin überprüft wird.

Oft unterscheidet man zwischen

- *Branch-Instruktion*: PC-relative Angabe des Sprungzieles
- *Jump-Instruktion*: Übergabe einer Absolutadresse an den Programmzähler.

In unserem Mini-Modell-Code könnte dies durch

- Branch: rel–pc
- Jump: unmittelbar

erreicht werden. Allgemein gilt die Syntax

`BR_CC D` bzw. `JP_CC #ADR`

mit dem Bedingungs-Code

`<CC>: <N|NN|Z|NZ|...>.`

Beispiel:

```
START  LDX   #Daten       ; [IR]=ADR DATEN
       LAD   BR.#0        ; [AC]=1
       MUL   #2           ; [AC]=2
       SUB   BR.#1        ; [AC]=0
       STR   @FELD        ; [FELD]=0
       JEZ   PC.S1        ; bedinger Sprung nach S1
       JMP   #START       ; unbedingter Sprung nach START
S1     STP
DATEN  DAT   W,1,2,3
FELD   RES   W,2
```

Schleifen werden mittels bedingter Sprünge konstruiert. Selten bieten Assembler für Schleifen Makrokonstrukte an.

Die Feinheiten unterschiedlicher Schleifenkonstruktionen höherer Programmiersprachen werden auf Maschinenebene zu einer Frage des Geschmacks.

In "C" kennt man die `while`-, `do`- und `for`-Anweisungen zur Bildung von Schleifen:

```
{ while              }          entspricht   { if (Bedingung)     }
{ (Bedingung)        }                       {   do               }
{   Anweisung        }                       {     Anweisung      }
                                             {   while            }
                                             { (Bedingung)        }
```

```
do                          entspricht    Anweisung
  Anweisung                               while
while                                     (Bedingung)
(Bedingung)                                   Anweisung
```

```
for (Ausdr.1; Bedingung;    entspricht    Ausdr.1
Ausdr.2)                                  while
  Anweisung                               (Bedingung)
                                          {
                                            Anweisung;
                                            Ausdr.2;
                                          }
```

Man nennt die `while`- und `do`-Schleifen auch *Bedingungsschleifen* und die `for`-Schleifen *Zählschleifen*.

Beispiel: Berechnung der Fakultät

In "C" erreicht man durch explizite Schleifen die Berechnung der Fakultät durch folgende Passagen:

1. `while`-Anweisung:

```
int i,n,Fak;
...
i=Fak=1;
while (i<=n)
  Fak *=i++;    //Postinkrement, also: Fak=Fak*i; i=i+1;
```

2. `do`-Anweisung:

```
int i,n,Fak;
...
i=Fak=1;
do
  Fak *=i++;
while (i<=n);
```

3. `for`-Anweisung:

```
int i,n,Fak;
...
Fak=1;
for (i=1; i<=n;i++)
  Fak *=i;
```

Im Unterschied zur `while`-Anweisung wird bei der `do`-Anweisung die Anweisung ausgeführt bevor der Test der Bedingung erfolgt. In beiden Fällen wertet die `while`-Anweisung die Bedingung `(i+1)`$\leq$`n` aus (außer im 1. Durchlauf der `while`-Schleife).

In der Mini-Assemblersprache exisitiert nicht unmittelbar die Abfragemöglichkeit für `(i-n)`$\leq$`0`, um direkt die Ausführung der Anweisung anzuschließen. Deshalb wird eine etwas umständliche Folge von bedingten und unbedingten Sprüngen notwendig. Ansonsten werden die "C"-Anweisungen direkt umgesetzt.

1. while-Anweisung:

```
START  LAD   #INK    ; Initialisierung Laufvariable
       STR   @FAKV
       STR   @I
LOOP   SUB   @N
       JLZ   #GO     ; Prüfen der Abbruchbedingung
       JEZ   #GO
       JMP   #STOP
GO     ADD   @N
       MUL   @FAKV   ; Ausführen der Anweisung
       STR   @FAKV
       LAD   @I
       ADD   #INK    ; Postinkrement der Laufvariable
       STR   @I
       JMP   #LOOP
STOP   STP
I      RES   W,1
FAKV   RES   W,1
N      DAT   W,100
INK    EQU   1
       END
```

2. `do`-Anweisung:

```
START  LAD   #INK    ; Initialisierung Laufvariable
```

```
        STR   @FAKV   ; (*)
        STR   @I      ; (*)
LOOP    MUL   @FAKV   ; Ausführen der Anweisung
        STR   @FAKV
        LAD   @I
        ADD   #INK
        SUB   @N
        JLZ   #GO     ; Prüfen der Abbruchbedingung
        JEZ   #GO
        JMP   #STOP
GO      ADD   @N
        STR   I
        JMP   #LOOP
STOP    STP
I       RES   W,1     ; (**)
FAKV    RES   W,1     ; (**)
N       DAT   W,100
INK     EQU   1
        END
```

Der Befehl `(*)` könnte gespart werden, wenn `(**)` lauten würde:

```
        I     DAT     W,1
        FAKV  DAT     W,1
```

Die Umformung von `i` $\leq$ `n` in `(n-i)` $\geq$ `0` und die Nutzung der Anweisung `JGZ ADR` spart keinen Befehl gegenüber der obigen Variante der Abfrage:

```
        ⋮
        LAD   @I
        ADD   #INK
        SUB   @N
        JLZ   #GO
        JEZ   #GO
        JMP   #STOP
GO      ADD   @N
        ⋮
```

$\Longrightarrow$

```
        LAD   @N
        SUB   @I
        SUB   #INK
        JGZ   #GO
        JMP   #STOP
GO      ADD   @I
        ADD   #INK
```

4.4.2 Adreßabbildungen durch Assembler und Binder

Einen wesentlichen Anteil an der Konvertierung symbolischer in binäre Maschineninstruktionen hat die Konvertierung der Adreßbezüge.

Gegeben sei eine Zerlegung eines Assemblerprogramms in die Segmente $S_1, ..., S_m, m \geq 1$. Diese Segmente S_r werden einzeln assembliert, ihre symbolischen in logische Adreßbezüge konvertiert. Diese Abbildung in einen *segment-spezifischen Adreßraum*

$$\alpha_r^A : L_r \rightarrow [0, ..., n_r - 1]$$

ordnet jedem Label $l_{r_s} \in L_r$ aus der Menge der Label L_r des Segments r eine Adresse zu. Dabei ist n_r die in Bytes ausgedrückte Länge, welche zur Konvertierung aller Befehle oder Datenbereiche im Binär-Code erforderlich ist.

Mit der Analyse jeder Zeile eines Segmentes (Befehlsobjekt oder Datenobjekt) wird der segmentrelative *Adreßzähler* um einen Betrag inkrementiert, der zur Binär-Codierung des Objektes erforderlich ist. Damit zeigt der Adreßzähler aktuell stets auf die Startadresse des nächsten Objektes im Binär-Code.

Die Abbildung α_r^A führt zum Aufbau von *Symboltabellen*, die jeder symbolischen Adresse $l_{r_s} \in L_r$ den jeweiligen Zählerstand zuordnen. Dies erfolgt im *ersten Assemblerpaß* (s. Abb 4.21).

In einem *zweiten Assemblerpaß* werden die tatsächlichen Adreßmodifikationen (z.B. relativ zum PC oder relativ zum TOS) in den Maschinencode (Bindercode) eingesetzt (s. Abb. 4.22). Außerdem erfolgt hier die Bildung des Operationsfeldes des symbolischen Befehls.

Im ersten Assemblerpaß werden auch Adreßbezüge zwischen den Segmenten festgestellt. Ist S_r das aktuelle assemblierte Segment, treten folgende Fälle auf:

1. Ein Label $l_{r_s} \in L_r$ wird in S_r deklariert und referenziert (*lokales Label*).
2. Ein Label $l_{r_s} \in L_r$ wird in S_r deklariert aber nicht notwendigerweise referenziert (wenn in S_t referenziert, dann *public Label*; sonst überflüssig).
3. Ein Label $l_{t_s} \in L_t$ wird in S_r referenziert aber nicht deklariert (wenn in S_t deklariert, dann *external Label*; sonst Fehler).

Referenzierung der Adreßsymbole in S_r erfolgt durch Bezug auf $S_t \neq S_r$ wegen:

1. eine werttransformierende oder transportierende Instruktion in S_r nimmt Bezug auf ein Datenobjekt in $S_t \neq S_r$ oder
2. eine steuernde Instruktion in S_r nimmt Bezug auf ein Befehlsobjekt in $S_t \neq S_r$.

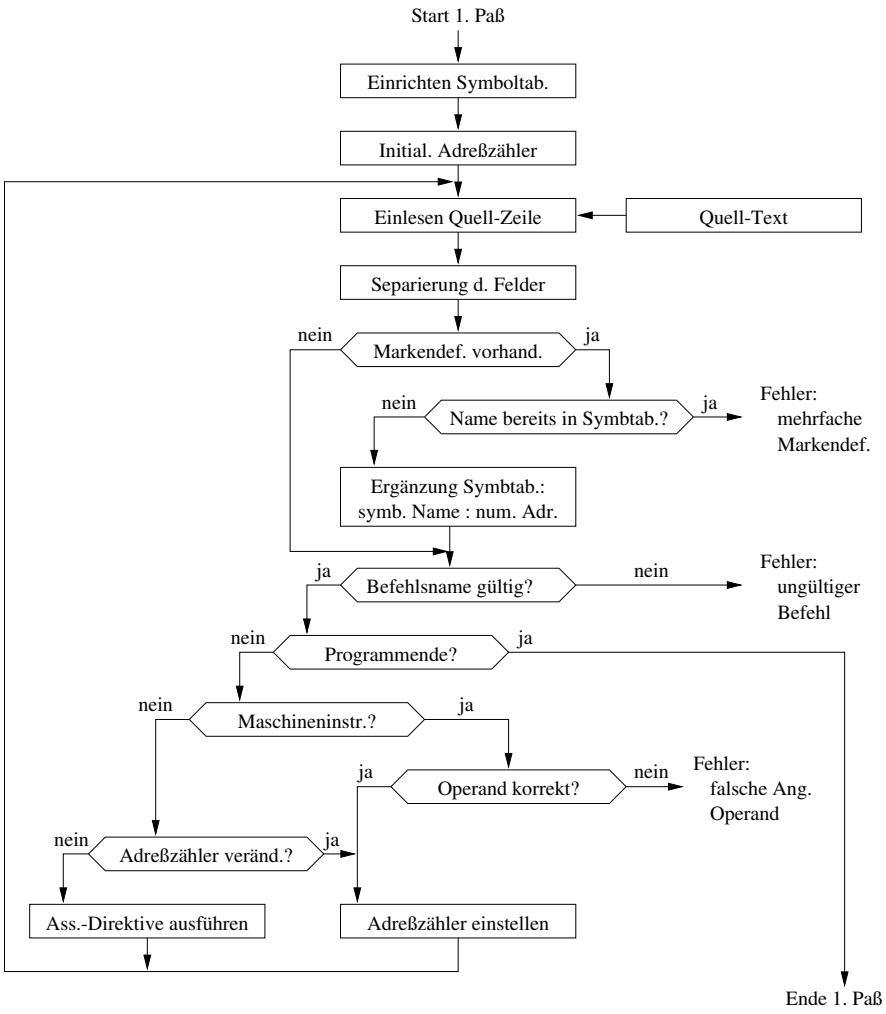


Abbildung 4.21: Struktur des 1. Assemblerpasses

Deshalb werden im ersten Assemblerpaß folgende Symboltabellen erstellt:

- LSD (local symbol dictionary): *segmentinterne Label*
- PSD (public symbol dictionary): *öffentliche Label*
- ESD (external symbol dictionary): *segmentexterne Label*

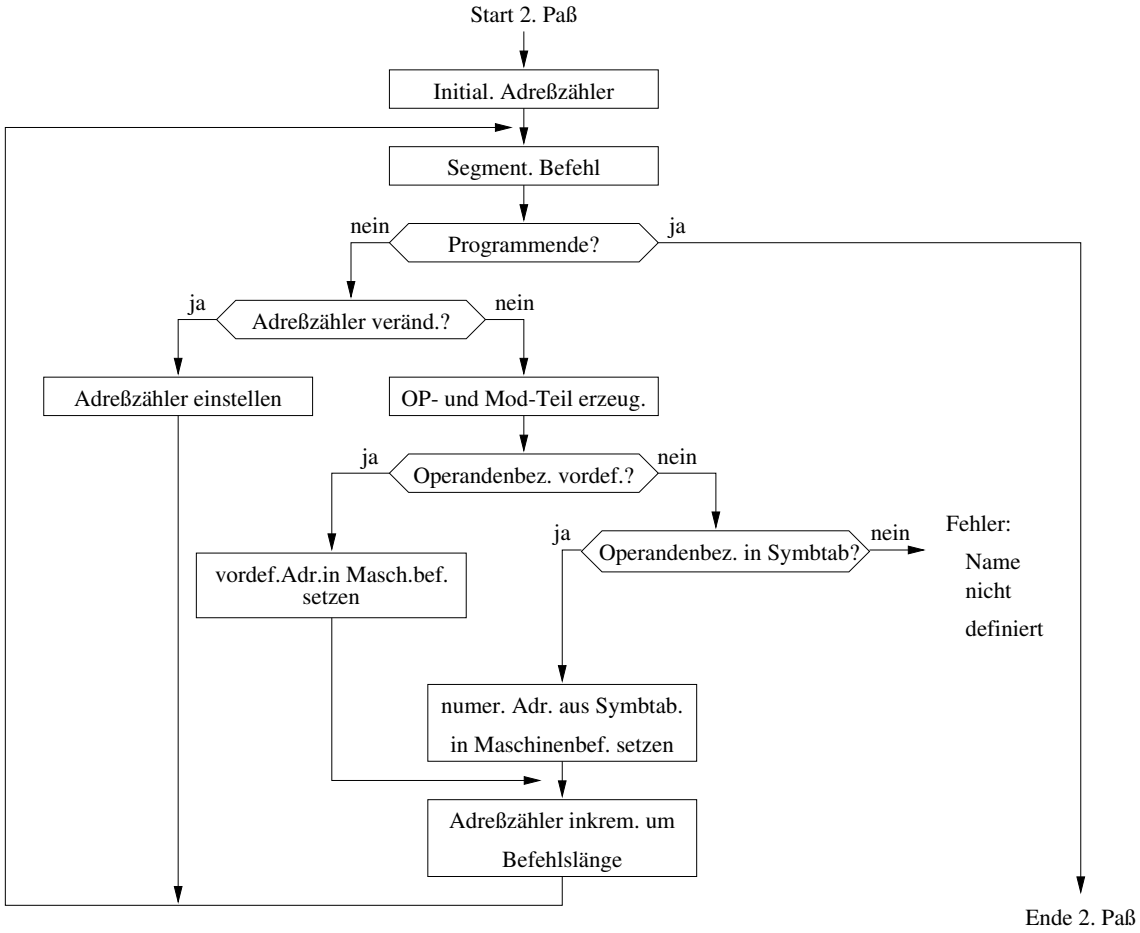


Abbildung 4.22: Struktur des 2. Assemblerpasses

Beispiel: Berechnung der Fakultät

```
        SEGMENT
        EXTERN  N,ERROR,MAX
        PUBLIC  FAKV,FAK
FAKV    DAT     D,1                                 8  [0...7]
I       DAT     W,1                                 4  [8...11]
INK     EQU     1
FAK     LAD     #INK    ; Initial. Laufvar.         4  [12...15]
LOOP    SUB     @N                                  4  [16...19]
```

```
        JLZ     #GO      ; Prüfen Abbruchbed.    4  [20...23]
        JEZ     #GO                              4  [24...27]
        JMP     #STOP                            4  [28...31]
GO      ADD     @N                               4  [32...35]
        MUL     @FAKV    ; Ausführen Anweis.     4  [36...39]
        SUB     @MAX                             4  [40...43]
        JGZ     #ERROR   ; Fehler: Überlauf      4  [44...47]
        ADD     @MAX                             4  [48...51]
        STR     @FAKV                            4  [52...55]
        LAD     @I                               4  [56...59]
        ADD     #INK     ; Inkrement Laufv.      4  [60...63]
        STR     @I                               4  [64...67]
        JMP     #LOOP                            4  [68...71]
STOP    RET                                      2  [72...73]
        END
                (*alignment*)                    2  [74...75]
                (*uninstalled N*)                4  [76...79]
                (*uninstalled ERROR*)            4  [80...83]
                (*uninstalled MAX*)              8  [84...91]
```

Symboltabellen:

LSD:	I	8	(4 Bytes)
	INK	12	(0 Bytes)
	LOOP	16	$(4 + 4 + 4 + 4 = 16$ Bytes)
	GO	32	$(10 \cdot 4 = 40$ Bytes)
	STOP	72	(2 Bytes)
PSD:	FAKV	0	(8 Bytes)
	FAK	12	(4 Bytes)
ESD:	N	76	(4 Bytes)
	ERROR	80	(4 Bytes)
	MAX	84	(8 Bytes)

Am Ende des zweiten Passes sind alle Offsets relativ zum Stand des PC berechnet. Dieser Binder-Code aller Segmente S_r wird durch den Binder so auf den virtuellen Adreßraum aller Segmente

$$V = \left[0, ..., \left(\left(\sum_{r=1}^{m} n_r\right) - 1\right)\right]$$

abgebildet, daß die Abbildung für das Segment S_r

$$\alpha_r^B : L_r \rightarrow V$$

zu der neuen *Basisadresse* des Segmentes S_r

$$a_r^B = \sum_{i=1}^{r-1} n_i$$

führt. Daraus ergibt sich für alle $l_{r_s} \in L_r$ die virtuelle Adresse

$$\alpha_{r_s}^B = \alpha_{r_s}^A + a_r^B.$$

Dabei muß die segmentübergreifende Referenzierbarkeit beachtet werden.
Sind E_r die externen Label in S_r und P_t seien die öffentlichen Label in S_t, dann ist S_r vollständig gebunden, wenn

$$(\forall e \in E_r)((\exists S_{t \neq r})(e \in P_t))$$

und für S_t eine Abbildung

$$\alpha_t^B : L_t \rightarrow V$$

vollständig spezifiziert ist.
Schließlich bildet der Lader den virtuellen Adreßraum V auf den realen Adreßraum des Arbeitsspeichers ab, so daß

$$\alpha_{r_s} = \alpha_{r_s}^B + a_{abs} \quad \forall l_{r_s} \in L_r, \forall r \in [1, ..., m]$$

mit der absoluten *Basisadresse*

$$a_{abs} \in \left[0, ..., 2^N - \left(\left(\sum_{r=1}^{m} n_r\right) - 1\right)\right].$$

5 Schaltfunktionen und Schaltnetze

In Kapitel 4 wurde die *top-down Sicht* auf die Architektur eines Rechners und die Abarbeitung von Programmen entwickelt. Dabei wurden wesentliche Elemente der *Zentraleinheit* eines Rechners (Rechenwerk, Steuerwerk, Speicher) bezüglich ihrer zustandsändernden und zustandsbeschreibenden Funktionalität entweder

- als Black Box oder
- als Einheiten zur Spezifizierung von Operatoren und Operanden im Rahmen der Maschinensprache

betrachtet.

In diesem und im nächsten Kapitel werden wir systematisch eine *bottom-up Sicht* auf die Funktion eines Rechners, insbesondere auf die Komponenten des Rechenwerkes und des Steuerwerkes entwickeln:

Kapitel 5: Es werden *Schaltnetze* (bzw. *kombinatorische Schaltwerke*) betrachtet, die eine Abbildung (*Schaltfunktion*)

$$F : X \to Y$$

realisieren. Dabei sind $X = \mathbb{B}^n,\ Y = \mathbb{B}^m,\ \mathbb{B} = \{0, \mathrm{L}\},\ n, m \in \mathbb{N}$, die Menge der Eingabesymbole bzw. die Menge der Ausgabesymbole.

Hieraus werden Schaltnetze entwickelt, für die derartige *gedächtnislose Abbildungen* ausreichend sind. Dies sind Codierer, Addierer, Multiplexer und Komparator.

Kapitel 6: Es werden sequentielle *Schaltwerke*eingeführt und *Automaten* als Maschinenmodelle zur Realisierung *gedächtnisbehafteter Abbildungen*der Art

$$G : S \times X \to Y$$
$$H : S \times X \to S$$

eingeführt. Dabei bezeichnet $S = \mathbb{B}^l,\ l \in \mathbb{N}$, die Menge interner Zustände eines Systems.

Während die Schaltfunktion F der kombinatorischen Schaltwerke Ausgaben $y \in Y$ erzeugt, die nur von den Eingaben $x \in X$ abhängen, ist die Ausgabe sequentieller Schaltwerke durch einen definierten Anfangszustand und eine Folge von Eingabesymbolen bestimmt. Ein typisches Beispiel sind Register.

Man kann sich Schaltwerke aus Schaltnetzen und Speichergliedern aufgebaut denken. Ihre Funktionalität hängt vom Zeitpunkt der Betrachtung ab. Demzufolge ist die Kopplung mit Taktsequenzen (Steuersignalen) von Interesse.

Das Kapitel wird einführend mit einem modernen Entwurfsschema für universell gestaltbare Schaltnetze (sogenannte Programmierbare Logischer Felder – *PLA*) und dem Prinzip rückgekoppelter Schaltungen zum Speichern von Zuständen (*Kippglieder*) bekannt machen.

Die Zielstellung dieser Kapitel besteht darin, Vorstellungen darüber zu vermitteln, wie die logischen Prinzipien zur Formulierung einer Problemlösung in einer (imperativen) Programmiersprache

- auf die Ressourcen des Rechners zurückgreifen (Rechenstrukturen, konkrete Gestalt des Prozessors)

und

- ihre Entsprechung in den arithmetisch-logischen Funktionsprinzipien der Rechnerbestandteile finden.

5.1 Schaltfunktionen und Schaltalgebra

Die Äquivalenz von Software-Strukturen und Hardware-Funktionalität ist eine algebraische Eigenschaft, die sich auf der Ebene binärer Repräsentationen zeigt. Sie läßt sich im Rahmen der *Booleschen Algebra* nachweisen und gestalten.

5.1.1 Boolesche Algebra

Der Begriff *Boolesche Algebra* geht auf den englischen Mathematiker *George Boole* (1815-1864) zurück. Ihre axiomatische Ausformulierung erfolgte allerdings erst 1904 durch *Huntington*. Erst in der ersten Hälfte dieses Jahrhunderts wurde die Boolesche Algebra als *Verband* mit speziellen Eigenschaften identifiziert.

Unter einem Verband versteht man eine algebraische Struktur, die aus Verallgemeinerungen von Beziehungen hervorgeht, die zwischen Teilmengen oder zwischen Teilstrukturen gewisser Strukturen wie Gruppen, Körpern u.s.w. auftreten.

Im Unterschied zu Abschnitt 3.2.2 werden wir hier die Boolesche Algebra über den Begriff Verband einführen.

Definition: (Verband)

Eine Menge V mit zwei zweistelligen Verknüpfungen

(Durchschnitt) $\bigcap : V \times V \to V$
(Vereinigung) $\bigcup : V \times V \to V$

heißt Verband $(V, \cap, \cup)$, wenn zwischen beliebigen Elementen $x, y, z, \cdots \in V$ folgende Eigenschaften festgestellt werden:

1. *Kommutativität*: $x \cap y = y \cap x$
   $x \cup y = y \cup x.$
2. *Assoziativität*: $(x \cap y) \cap z = x \cap (y \cap z)$
   $(x \cup y) \cup z = x \cup (y \cup z).$
3. *Absorption*: $x \cap (x \cup y) = x$
   $x \cup (x \cap y) = x.$

Wenn wir den beiden Operatoren die Bezeichnung Durchschnitt ($\cap$) bzw. Vereinigung ($\cup$) aus der Mengenalgebra geben, soll dies nicht einschränkend bedeuten, daß keine anderen Operationen hier stehen könnten. Wir werden aber bald feststellen, daß die Verwendung dieser Begriffe hier Sinn macht.

Beispiele: (Verband)

1. $\vee$: Menge der positiven ganzen Zahlen $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$
   $\cap$: größter gemeinsamer Teiler
   $\cup$: kleinstes gemeinsames Vielfaches
2. $\wedge$: Menge von aussagenlogischen Formeln
   $\cap$: UND
   $\cup$: ODER

Die Umkehrung der Absorption

$$x = x \cap (x \cup y)$$
$$x = x \cup (x \cap y)$$

heißt auch *Expansion*.

Aus diesen Eigenschaften lassen sich zwei wichtige Folgerungen ableiten:

1.	$x \cup x = x = x \cap x$	Idempotenzgesetz der Mengenalgebra
2.	$x \cap y = x \Leftrightarrow x \cup y = y$	Halbordnungsrelation

Beweis zu 1. (linke Seite):

$$\begin{array}{rcll} x \cup x & = & x \cup (x \cap (x \cup y)) & \text{Expansion} \\ & = & x \cup (x \cap z) & \text{Substitution} \\ & = & x & \text{Absorption} \end{array}$$

Beweis zu 2. (linke Seite):

$$\begin{array}{rcll} x \cap y = x & \Rightarrow & x \cup y = y & \\ (x \cap y) \cup y & = & x \cup y & \text{Termerweiterung} \\ y & = & x \cup y & \text{Absorption} \end{array}$$

(rechte Seite):

$$\begin{array}{rcll} x \cap y = x & \Leftarrow & x \cup y = y & \\ x \cap (x \cup y) & = & x \cap y & \text{Termerweiterung} \\ x & = & x \cap y & \text{Absorption} \end{array}$$

Eine wichtige weitere Eigenschaft eines Verbandes ist die Existenz einer *Halbordnungsrelation* auf V. Diese Eigenschaft haben nicht alle Verbände, aber die hier interessierenden.

Satz (Halbordnung):

Sei $(V, \cap, \cup)$ ein Verband und sei für alle $x, y \in V$ eine reflexive, transitive und antisymmetrische Relation $\subseteq$ definiert durch

$$x \subseteq y \Leftrightarrow x \cap y = x \Leftrightarrow x \cup y = y,$$

dann geht aus dem Verband eine Halbordnung $(V, \subseteq)$ hervor.

In der Mengenalgebra heißt die Relation $\subseteq$ Inklusion.

1. Reflexivität: $x \subseteq x$, klar wegen $x \cap x = x = x \cup x$

2. Antisymmetrie: $x \subseteq y,\ y \subseteq x \Rightarrow x = y$

$$\begin{aligned} x \subseteq y,\ y \subseteq x &\Leftrightarrow x \cap y = x,\ y \cap x = y \\ &\Rightarrow x = x \cap y = y \quad \text{Kommutativität} \\ x \subseteq y,\ y \subseteq x &\Leftrightarrow x \cup y = y,\ y \cup x = x \\ &\Rightarrow x = x \cup y = y \quad \text{Kommutativität} \end{aligned}$$

3. Transitivität: $x \subseteq y,\ y \subseteq z \Rightarrow x \subseteq z$

$$\begin{aligned} x \subseteq y,\ y \subseteq z &\Leftrightarrow x \cap y = x,\ y \cap z = y \\ &\Rightarrow x \cap z = x \cap y \cap z = x \cap y = x \\ &\Leftrightarrow x \subseteq z \end{aligned}$$

ebenso mit $x \cup y = y$

Aus dieser Halbordnungsrelation folgt die Existenz eines kleinsten und eines größten Elementes in V, falls V eine endliche Menge ist.

Definition: (kleinstes/größtes Element)

Interpretiert man die Relation $x \subseteq y$ als "y umfaßt x", so existiert in der Halbordnung $(V, \subseteq)$

a) ein *kleinstes Element* ("m"), das kein anderes Element aus V umfaßt und

b) ein *größtes Element* ("M"), das von keinem anderen Element aus V umfaßt wird.

Aus dem Halbordnungssatz folgen unmittelbar folgende Eigenschaften eines Verbandes:

a) $m \subseteq x \ \Leftrightarrow\ m \cap x = m \ \Leftrightarrow\ m \cup x = x$

b) $x \subseteq M \ \Leftrightarrow\ x \cup M = M \ \Leftrightarrow\ x \cap M = x.$

Damit ist das kleinste Element ebenfalls das *neutrale Element* der Vereinigung und das größte Element ist das neutrale Element des Durchschnittes.

Definition: (komplementärer Verband)

Eine Menge V, in der ein kleinstes und ein größtes Element existieren, bildet einen komplementären Verband $(V, \cap, \cup, \bar{})$, in dem die Operation der Komplementbildung für jedes $x \in V$ erklärt ist durch

$$M = x \cup \overline{x} \quad \text{und} \quad m = \overline{M}$$
$$\text{bzw.}$$
$$m = x \cap \overline{x} \quad \text{und} \quad M = \overline{m}.$$

In einem komplementären Verband werden das Element x und sein Komplement als Literal zusammengefaßt.

Definition: (Literal)

Eine Variable $\tilde{x} \in \{x, \overline{x}\},\ x, \overline{x} \in V$, die also wahlweise die Werte x oder $\overline{x}$ annehmen kann, heißt Literal.

In einem komplementären Verband gelten außerdem die Regeln:

1. $\overline{\overline{x}} = x$
2. $x = y \ \Leftrightarrow\ \overline{x} = \overline{y}$
3. $\overline{x \cup y} = \overline{x} \cap \overline{y}, \quad \overline{x \cap y} = \overline{x} \cup \overline{y}$ (de Morgan)
4. $x \subseteq y \ \Leftrightarrow\ \overline{x} \supseteq \overline{y}$
5. $x \subseteq y \ \Leftrightarrow\ x \cap \overline{y} = m \ \Leftrightarrow\ \overline{x} \cup y = M$

Beweis zu 4.: wegen des Satzes über die Halbordnungsrelation und de Morgan's Regeln gilt:

$x \subseteq y$	$\Leftrightarrow$	$x \cap y = x$	$\Leftrightarrow$	$x \cup y = y$	Def. Halbordnung
	$\Leftrightarrow$	$\overline{x \cap y} = \overline{x}$	$\Leftrightarrow$	$\overline{x \cup y} = \overline{y}$	Negation (2. Regel)
	$\Leftrightarrow$	$\overline{x} \cup \overline{y} = \overline{x}$	$\Leftrightarrow$	$\overline{x} \cap \overline{y} = \overline{y}$	de Morgan
	$\Leftrightarrow$	$\overline{x} \supseteq \overline{y}$			Def. Halbordnung

Aus den Eigenschaften eines komplementären Verbandes läßt sich die Existenz eines dualen komplementären Verbandes erklären.

Definition: (dualer komplementärer Verband)

Ein Verband $(V, \cup, \cap, \bar{}, M, m, \supseteq)$ geht als dualer komplementärer Verband aus einem Verband $(V, \cap, \cup, \bar{}, m, M, \subseteq)$ durch Vertauschen der zweistelligen Operationen, von größtem und kleinstem Element, der Halbordnung und durch Negation aller Literale hervor.

Die Bedeutung dieser Definition wird später im Zusammenhang mit dem Shannonschen Inversionssatz für Schaltfunktionen ersichtlich.

Definition: (distributiver Verband)

In einem distributiven Verband $(V, \cap, \cup)$ gelten die beiden Distributivgesetze:

$$x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$$

und

$$x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$$

für jedes $x, y, z \in V$.

Damit haben wir alle Voraussetzungen für die Definition einer Booleschen Algebra gelegt.

Definition: (Boolesche Algebra)

Ein komplementärer Verband $(V, \cap, \cup, \bar{})$ heißt Boolesche Algebra, falls $(V, \cap, \cup)$ ein distributiver Verband ist.

Diese Definition scheint auf den ersten Blick wenig mit der Algebra der Booleschen Variablen $\mathbb{B} = \{0, 1\}$ bzw. $\mathbb{B} = \{0, \mathrm{L}\}$ bzw. $\mathbb{B} = \{\mathrm{FALSE}, \mathrm{TRUE}\}$ zu tun zu haben. Denn über die Menge V wurden bisher noch keine Aussagen getroffen.

Definition: (Atom einer Booleschen Algebra)

Sei $(V, \cap, \cup, \bar{})$ eine Boolesche Algebra, dann heißt $a \in V$ Atom, falls $a \neq m$ und für alle $x \in V$ mit $x \subseteq a$ gilt $x = m$ oder $x = a$.

Die Definition der Atome bedeutet, daß diese bezüglich der Halbordnungsrelation direkt über dem kleinsten Element zu finden sind.

Definition: (atomare Boolesche Algebra)

Eine Boolesche Algebra heißt atomar, falls es zu jedem $x \in V$, $x \neq m$ ein Atom $a \in V$ gibt, so daß $a \subseteq x$ gilt.

Man kann zeigen, daß jede endliche Boolesche Algebra, die durch $|V| < \infty$ definiert ist, atomar ist. Schließlich gilt der

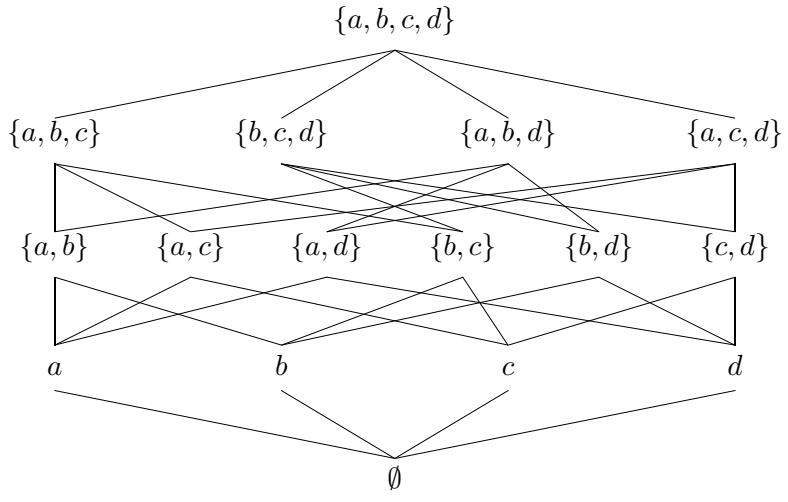
Hauptsatz für endliche Boolesche Algebren

Ist N die Anzahl der Atome einer endlichen Booleschen Algebra $(V, \cap, \cup, \bar{})$, so gilt $|V| = 2^N$.

Damit ist die *Mengenalgebra* über der Potenzmenge $\mathcal{P}(\Sigma), |\mathcal{P}(\Sigma)| = 2^{|\Sigma|}$ einer endlichen Menge $\Sigma, |\Sigma| = N$ eine endliche Boolesche Algebra. Diese Algebra hat N Atome. Dies sind die einelementigen Teilmengen von $P(\Sigma)$. Die zweistelligen Verknüpfungen $\cap$ und $\cup$ sind Durchschnitt und Vereinigung der Booleschen Algebra der Potenzmenge. Für kleinstes und größtes Element gilt $m = \emptyset$ und $M = \Sigma$.

Beispiel:

$\Sigma = \{a, b, c, d\}, N = 4, V = \mathcal{P}(\Sigma), |\mathcal{P}(\Sigma)| = 2^4 = 16$



Menge: 1

4

6

Atome: 4

leere Menge: 1

Die Atome von Σ sind a, b, c, d.

Alle endlichen Booleschen Algebren $(V, \cap, \cup, \bar{})$ sind isomorph zur Booleschen Algebra der Potenzmengen $(P(\Sigma), \cap, \cup, \bar{})$, wobei N die Anzahl der Atome ist. Irgend eine Potenzmenge ist abbildbar auf die Menge $\{1, 2, \ldots, 2^N\} \subset \mathbb{N}\backslash\{0\}$.

Definition: (Isomorphismus)

Eine eineindeutige Abbildung φ eines Verbandes V_1 auf einen Verband V_2 nennt man Isomorphismus, wenn für zwei beliebige Elemente $x, y \in V_1$ gilt:

1. $\varphi(x \cup y) = \varphi(x) \cup \varphi(y)$
2. $\varphi(x \cap y) = \varphi(x) \cap \varphi(y)$
3. $\varphi(\overline{x}) = \overline{\varphi(x)}$

Existiert eine solche Abbildung φ von einem Verband V_1 auf einen Verband V_2, so sind V_1 und V_2 isomorph zueinander.

Im Abschnitt 2.3.3 (Binärcodierung und Entscheidungsinformation) wurden die Atome einer Menge Σ und damit auch der Potenzmenge $\mathcal{P}(\Sigma)$ durch Binärworte $w \in \mathbb{B}^*$, $w =< w_1 \cdots w_n >$, $w_j \in \mathbb{B}$ codiert. Dies erfordert für alle Atome $w_i \in \Sigma$, $|\Sigma| = N$ einen mittleren Entscheidungsaufwand bzw. eine mittlere Länge der Binärworte

$$H = \sum_{i=1}^{N} p_i \operatorname{ld}\left(\frac{1}{p_i}\right) \quad [\text{bit}].$$

Im obigen Beispiel: $p_i = \frac{1}{4}$, also $H = \operatorname{ld} 4 = 2$ bit
a = <00>, b = <10>, c = <01>, d = <11>

Mit der Binärcodierung der Atome der Menge Σ haben wir eine isomorphe Abbildung der Booleschen Algebra $(2^N, \cap, \cup, \bar{}, \emptyset, \Sigma)$ auf die Boolesche Algebra $(\mathbb{B}, \wedge, \vee, \bar{}, 0, 1)$ realisiert.

Im Falle $V \equiv \mathbb{B} = \{0, 1\}$ existiert nur ein Atom, das mit dem größten Element des Verbandes identisch ist.

Damit ist gezeigt, daß alle der Potenzmengenalgebra zuzuordnenden Probleme durch Binärcodierung als Probleme der binären Booleschen Algebra behandelt werden können. Dies sind z.B.:

- Aussagenlogik: $\mathbb{B} = \{\text{FALSE}, \text{TRUE}\}$
  $\cup \Leftrightarrow \vee$ Disjunktion

	$\cap \Leftrightarrow \wedge$	Konjunktion
	$\dot{-} \Leftrightarrow \neg$	Negation
• Binärzahlenarithmetik:	$\mathbb{B} = \{0, 1\}$	
	$\cup \Leftrightarrow +$	Addition
	$\cap \Leftrightarrow *$	Multiplikation
	$\bar{} \Leftrightarrow \bar{}$	Negation
• Schaltfunktionen/Schaltalgebra:	$\mathbb{B} = \{0, \mathrm{L}\}$	
	$\cup \Leftrightarrow \vee$	Parallelschaltung
	$\cap \Leftrightarrow \wedge$	Reihenschaltung
	$\bar{} \Leftrightarrow \bar{}$	Negation

5.1.2 Schaltfunktionen und Schaltalgebra

Definition: (Schaltnetz nach DIN 44300/93)

Ein Schaltnetz ist ein Schaltwerk, dessen Wert am Ausgang zu irgendeinem Zeitpunkt nur vom Wert am Eingang zu diesem Zeitpunkt abhängt.

Ein Schaltnetz wird realisiert durch eine Schaltfunktion $F : \mathbb{B}^n \longrightarrow \mathbb{B}^m$,

$$Y = F(X)$$

mit $Y = (y_1, \cdots, y_m),\ X = (x_1, \cdots, x_n)$ und $x_i, y_i \in \mathbb{B} = \{0, \mathrm{L}\}$ als *Schaltvariable*. Wegen

$$\begin{aligned} y_1 &= f_1(x_1, \cdots, x_n) \\ y_2 &= f_2(x_1, \cdots, x_n) \\ \vdots &\quad \vdots \\ y_m &= f_m(x_1, \cdots, x_n) \end{aligned}$$

kann jede Schaltfunktion F durch ein System Boolescher Funktionen

$$f_i(x_1, \cdots, x_n),\ i = 1, \cdots, m,\ f_i : \mathbb{B}^n \to \mathbb{B}$$

realisiert werden (vgl. Abb. 5.1)

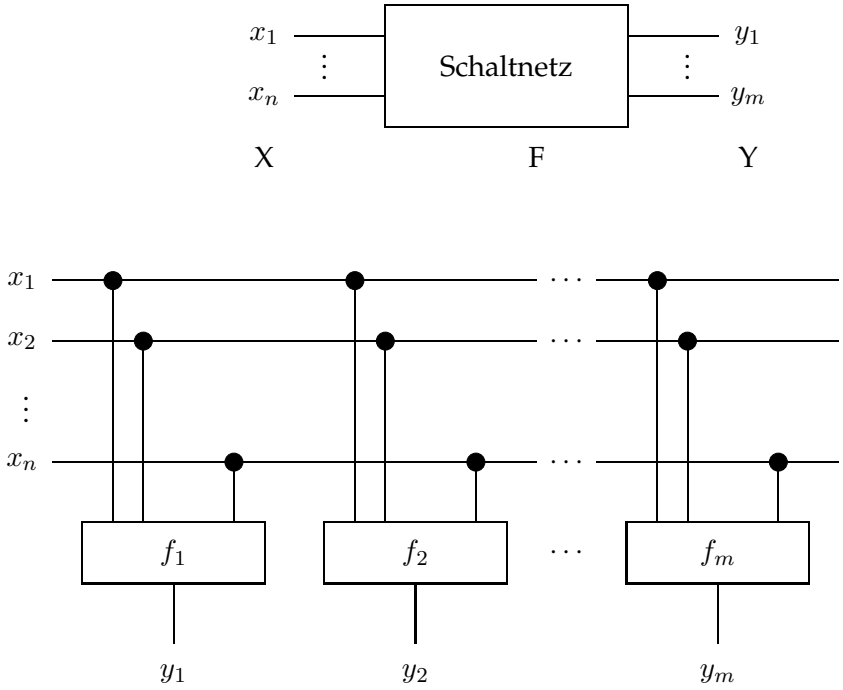


Abbildung 5.1: Realisierung eines Schaltnetzes als System Boolescher Funktionen

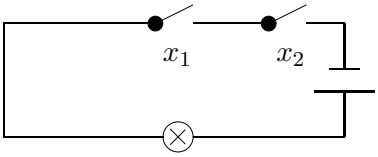
Es macht deshalb Sinn, die Betrachtungen zunächst auf Boolesche Funktionen zu beschränken und diese auch *Schaltfunktionen* zu nennen. Schaltfunktionen erfüllen die Gesetze der *Schaltalgebra* bezüglich der Verknüpfungen $\wedge, \vee, \bar{}$, wobei für $\mathbb{B} = \{0, \mathrm{L}\}$ gilt

$$\begin{aligned} \wedge &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \vee &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \bar{} &: \mathbb{B} \to \mathbb{B}. \end{aligned}$$

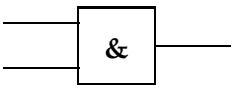
Schaltfunktionen lassen sich auf Parallel- und Reihenschaltung von Schaltern zurückführen, denn die elementare Operation "$\wedge$" entspricht der Reihenschaltung und die elementare Operation "$\vee$" entspricht der Parallelschaltung von Schaltern. In der Abbildung 5.2 sind für das UND-Glied und das ODER-Glied ein Ersatzschaltplan, die Wahrheitstafeln für die Funktionswerte aller Belegungen von $x_1, x_2 \in \mathbb{B}$ und die Schaltsymbole nach DIN 40 700 dargestellt.

Schalter sind selbst Nicht-Glieder, die den Schaltzustand zwischen "ein" und "aus" invertieren. Sie repräsentieren eine Boolesche Variable $x \in \mathbb{B}$.

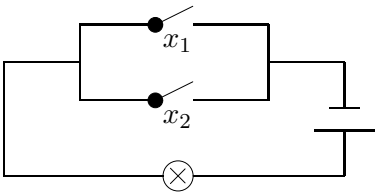
UND-Glied (AND): $\wedge$



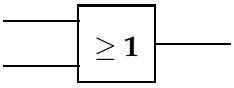
x_1 \ x_2	O	L
O	O	O
L	O	L



ODER-Glied (OR): $\vee$

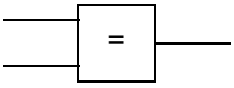


x_1 \ x_2	O	L
O	O	L
L	L	L



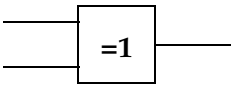
Äquivalenz: $\Leftrightarrow$

x_1 \ x_2	O	L
O	L	O
L	O	L



Antivalenz (XOR): $\not\Leftrightarrow$

x_1 \ x_2	O	L
O	O	L
L	L	O



NICHT-Glied (NOT): $\bar{}$

x	$\overline{x}$
O	L
L	O

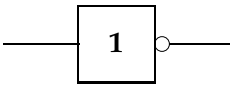


Abbildung 5.2: Wertetabellen und Schaltsymbole

idealer Schalter:
Der ideale Schalter entspricht der Booleschen Variablen x. Er schaltet leistungslos und zeitunabhängig (Abb. 5.3). Im Schaltplan werden die Ersatzwiderstände R (Arbeitswiderstand), R_I (Innenwiderstand) und R_s (Sperrwiderstand) verwendet.

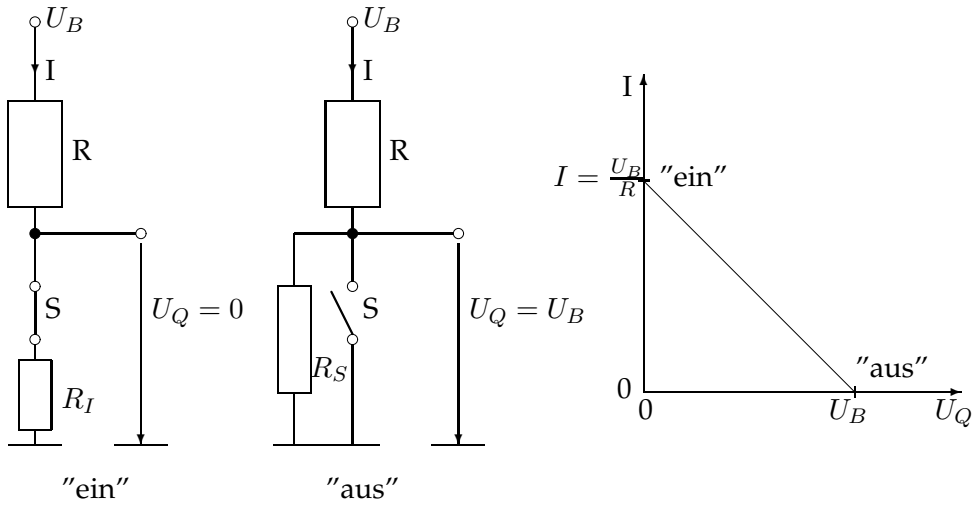


Abbildung 5.3: idealer Schalter

Schalterzustand "ein":	Innenwiderstand des Schalters $R_I = 0$, daraus folgt $I = \frac{U_B}{R},\ U_Q = 0$
Schalterzustand "aus":	Sperrwiderstand des Schalters $R_S = \infty$, daraus folgt $I = 0,\ U_Q = U_B$

Die vom Schalter aufgenommene Leistung $P = I \cdot U_Q$ ist immer Null, da entweder der Strom ("aus") oder die Spannung ("ein") Null ist.

realer Schalter:
Elektronische Schalter kommen als reale Schalter dem idealen Schalter am nächsten (Abb. 5.4).

Angestrebt wird

"aus": R_S sehr groß
"ein": R_I sehr klein.

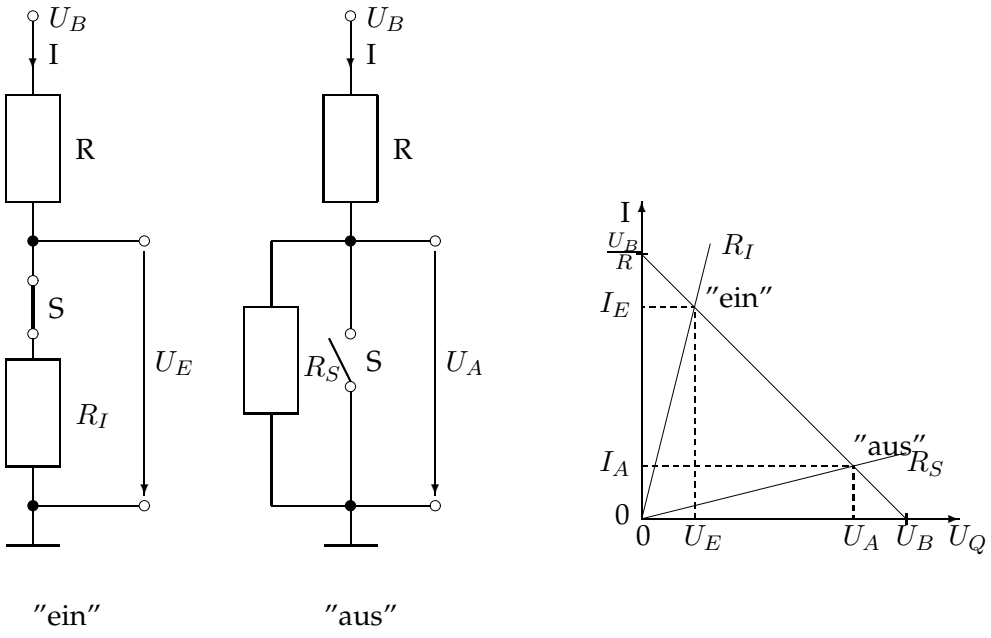


Abbildung 5.4: realer Schalter

Schalterzustand "ein": Die Widerstände R und R_I liegen in Reihe.
Im Arbeitspunkt ($R_I = R$) gilt
$I_E = \frac{U_B}{R+R_I}$, $U_E = \frac{U_B \cdot R_I}{R+R_I}$
Am Schalter fällt die Spannung U_E ab (nicht Null).

Schalterzustand "aus": Die Widerstände R und R_S liegen in Reihe.
Im Arbeitspunkt ($R_S = R$) gilt
$I_A = \frac{U_B}{R+R_S}$, $U_A = \frac{U_B \cdot R_S}{R+R_S}$
Trotz Schalterstellung "aus" fließt ein Strom I_A.

In beiden Zuständen nimmt der Schalter Leistung auf.

Transistor als Schalter:
In digitalen Schaltungen werden Nicht-Glieder durch Transistoren realisiert. Hierfür kommen unterschiedliche Bauarten zur Anwendung. In der Abbildung 5.5 wird ein Unipolartransistor vom Anreicherungstyp (Schwellspannung $U_T > 0$) erklärt. Es handelt sich um einem *MOS-FET* (Metal Oxide Semiconductor Field-Effect Transistor). Mit Hilfe der Gate-Source-Spannung U_{gs} wird die Drain-Source-Spannung U_{ds} gesteuert. In Abhängigkeit vom Verhältnis der Gate-Source Spannung zu einem Schwellwert $U_T > 0$

stellen sich ″ein″-Zustand und ″aus″-Zustand ein.

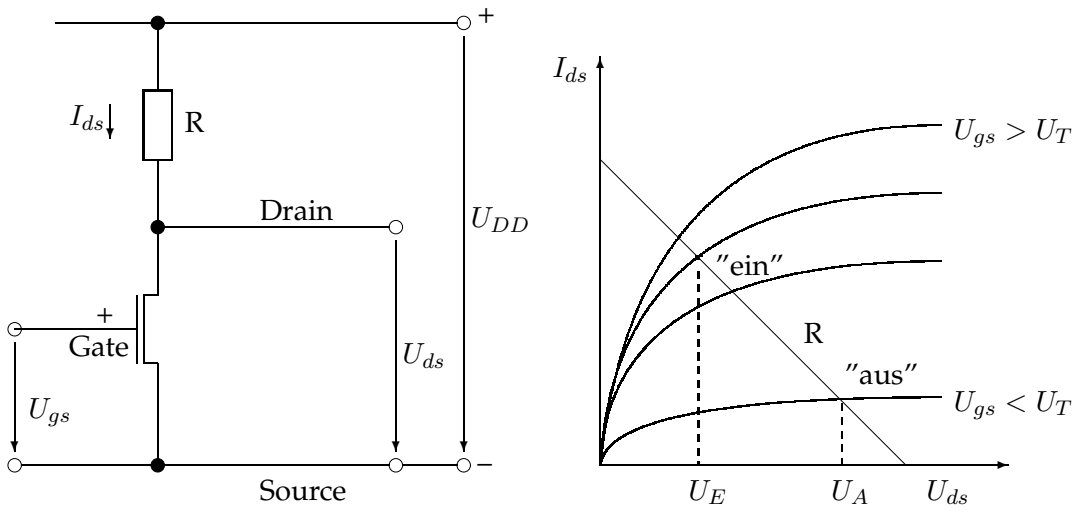


Abbildung 5.5: Transistor als Schalter

Schalterzustand ″ein″: Transistor leitend
$U_{gs} > U_T \Rightarrow (I_{ds} \gg 0) \Leftrightarrow (U_{ds} \approx 0)$
Schalterzustand ″aus″: Transistor gesperrt
$U_{gs} < U_T \Rightarrow (I_{ds} \approx 0) \Leftrightarrow (U_{ds} \gg U_T)$

Realisierung als Negations-Gatter:
Der Transistor arbeitet also als *NICHT-Glied* bzw. *Negations-Gatter*. In digitalen Schaltungen werden Glieder gewöhnlich als *Gatter* bezeichnet. Für die Realisierung der Negation werden allerdings zwei Transistoren in Reihe geschaltet (Abb. 5.6,5.7). Dabei macht man sich die Interpretation eines Transistors als spannungsgesteuerten Widerstand zu Nutze. Ein Transistor vom Anreicherungstyp realisiert den inneren und Sperrwiderstand (R_I, R_S) des Schalters, während der Arbeitswiderstand R durch einen Transistor vom Verarmungstyp ($U_T < 0$) realisiert wird. Für diesen gilt:

Schalterzustand ″ein″: $U_{gs} = U_T \Rightarrow I_{ds} \gg 0$
Schalterzustand ″aus″: $U_{gs} \ll U_T \Rightarrow (I_{ds} \approx 0) \Leftrightarrow (U_{ds} = U_{DD})$.

Allerdings wird hier die Schalterwirkung nicht genutzt.

Die negierende Schaltfunktion

U_{IN}	U_{OUT}
0	L
L	0

stellt also eine Idealisierung sowohl hinsichtlich der Pegel als auch des Zeitverhaltens dar.

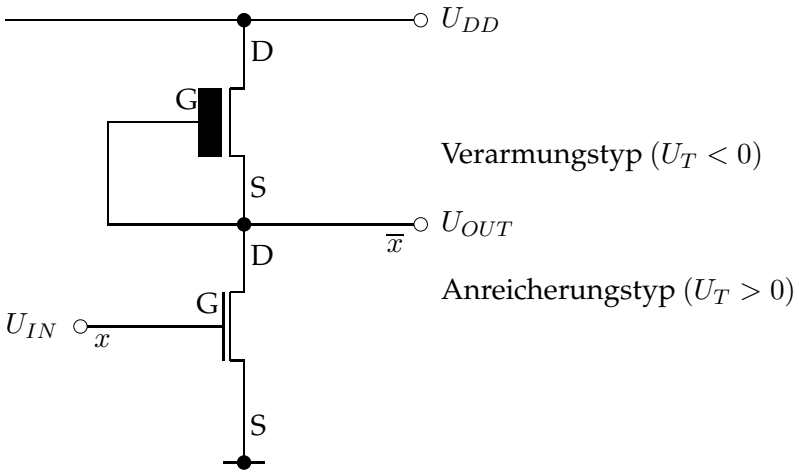


Abbildung 5.6: Realisierung der Negation

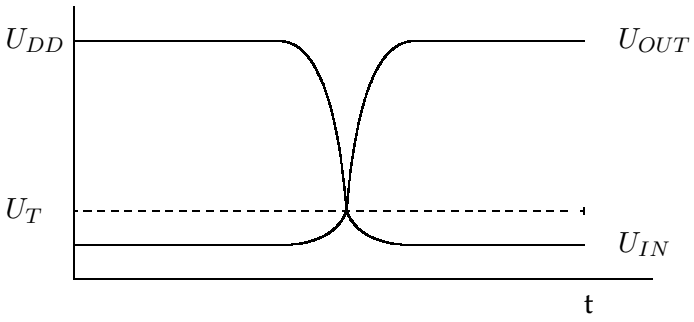


Abbildung 5.7: Spannungsverlauf des Negationsgatters (idealisiert)

Realisierung der Logik-Gatter:
Die Verknüpfungselemente $\wedge$ bzw. $\vee$ werden als AND- bzw. OR-Gatter durch Reihen- bzw. Parallelschaltung von Transistoren realisiert (Abb. 5.8,5.9).

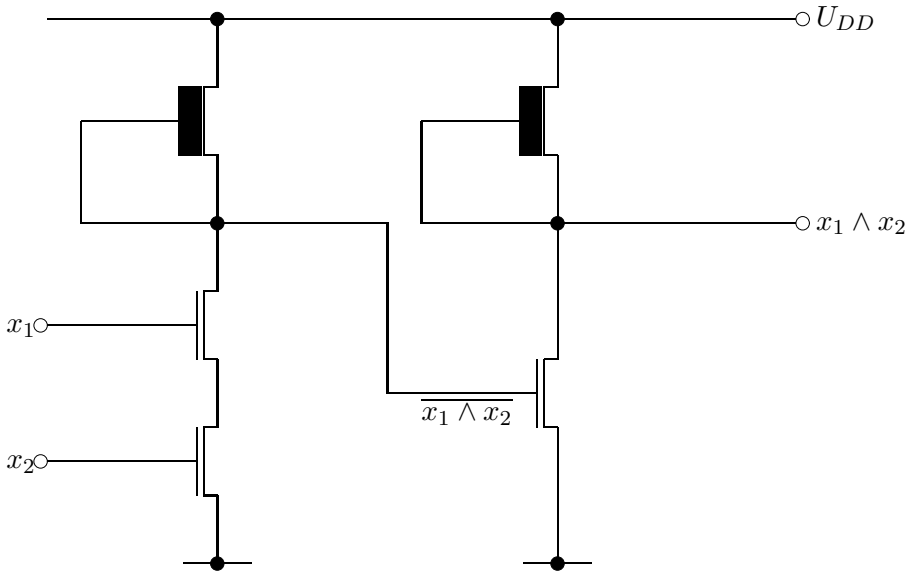


Abbildung 5.8: Realisierung des AND-Gatters

Schaltpläne:
Die Gatter für UND, ODER und NICHT bilden ein vollständiges Basissystem aller Schaltfunktionen. Mit Hilfe von Schaltplänen können die Verknüpfungsstrukturen von Schaltfunktionen dargestellt werden.

Definition: (Schaltplan)

Ein Schaltplan ist ein gerichteter Graph, der die Verknüpfungsoperatoren und die Klammerstruktur von Schaltfunktionen normgerecht nach DIN 40 700 unter Verwendung der Schaltsymbole darstellt.

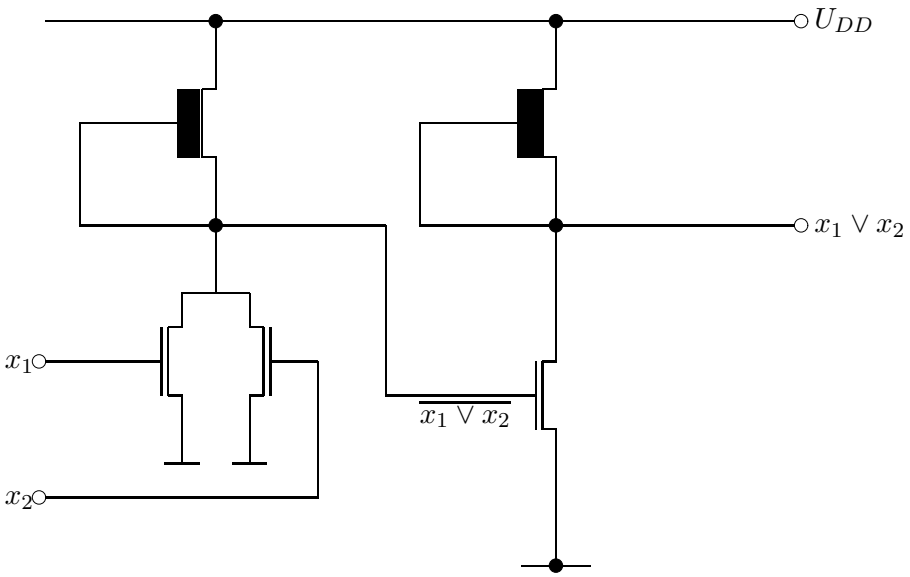
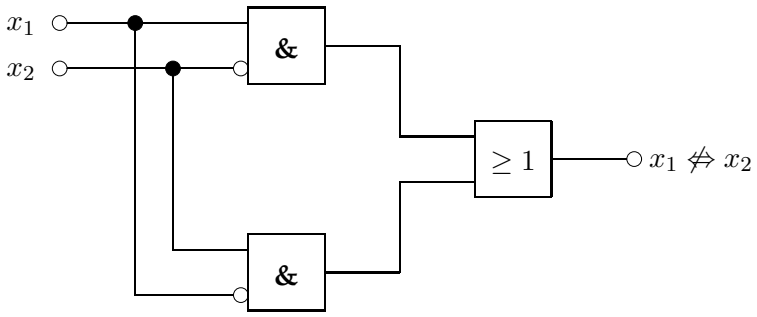


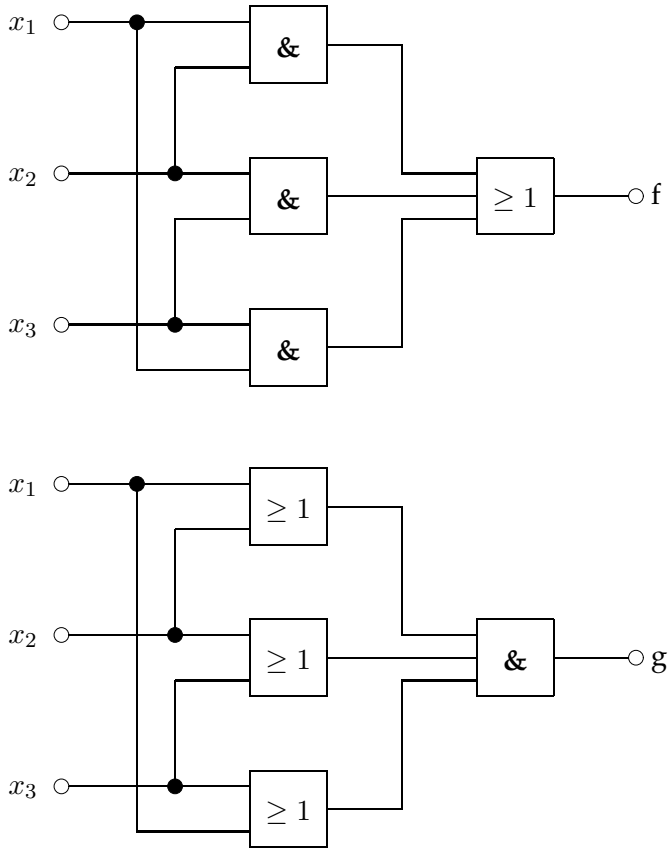
Abbildung 5.9: Realisierung des OR-Gatters

Beispiele:

1. Antivalenz $f(x_1, x_2) = (x_1 \not\Leftrightarrow x_2) = (x_1 \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2)$



2. 2-von-3-Mehrheitsfunktion $f(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$
   es gilt: f=g ! $g(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$



Monotonie einer Schaltfunktion:

Definition: (Monotonie einer Schaltfunktion)

Für zwei Eingangsvektoren $A = (a_1, \cdots, a_n)$ und $B = (b_1, \cdots, b_n)$ ist die Ordnung $A \leq B$ komponentenweise definiert durch $a_i \leq b_i, i = 1, \cdots, n$.

Eine Schaltfunktion f ist monoton, wenn gilt

$$(a_1, \cdots, a_n) \leq (b_1, \cdots, b_n) \;\Rightarrow\; f(a_1, \cdots, a_n) \leq f(b_1, \cdots, b_n)$$

für alle $A = (a_1, \cdots, a_n),\ B = (b_1, \cdots, b_n) \in \mathbb{B}^n$.

Aus der Ordnungsrelation eines Verbandes folgt, daß Schaltfunktionen solange monotone Funktionen sind, als nur die Operation AND bzw. OR zu ihrer Konstruktion verwendet werden. Dies gilt allgemein für n-stellige Funktionen.

Gilt für zwei Eingangsvektoren $A = (a_1, \cdots, a_n)$ und $B = (b_1, \cdots, b_n)$ die Ordnung $A \leq B$, so bleibt diese Ordnung durch die Abbildung f erhalten, wenn eine Darstellung von f nur mit $\wedge, \vee$ als Operationen möglich ist.

Dies ist eine unmittelbare Folge der Monotonie der Operationen AND und OR:

$$a_1 \leq b_1, a_2 \leq b_2 \;\Rightarrow\; a_1 \vee a_2 \;\leq\; b_1 \vee b_2\,,\; a_1 \wedge a_2 \;\leq\; b_1 \wedge b_2.$$

Wie leicht einzusehen ist (Umbenennung der Variablen), gilt dies auch für die Ordnungsrelation $\geq$.

Beweis: durch Belegung der Komponenten zweistelliger Eingangsvektoren

a_1	b_1	a_2	b_2	$a_1 \vee a_2$	$b_1 \vee b_2$	$a_1 \wedge a_2$	$b_1 \wedge b_2$
0	0	0	0	0	0	0	0
0	L	0	L	0	L	0	L
L	L	L	L	L	L	L	L

$a_1 \leq b_1 \quad a_2 \leq b_2 \quad a_1 \vee a_2 \;\leq\; b_1 \vee b_2 \quad a_1 \wedge a_2 \;\leq\; b_1 \wedge b_2$

Erst die Einführung des komplementären Verbandes mit der Eigenschaft

$$x \leq y \;\Leftrightarrow\; \overline{x} \geq \overline{y}$$

vermag diese Einschränkung der Gestaltung von Schaltfunktionen zu durchbrechen.

Eine n-stellige Boolesche Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}$ wird durch einen Vektor von 2^n konstanten Abbildungen $y_{(\alpha)}, 0 \leq \alpha \leq 2^n - 1, y_{(\alpha)} \in \mathbb{B}$, definiert:

$$g_f(y_{(0)}, \ldots, y_{(2^n-1)}).$$

Diese konstanten Abbildungen $y_{(\alpha)} = f(\beta_\alpha(x_n), \beta_\alpha(x_{n-1}), \ldots, \beta_\alpha(x_1))$ stehen für die Belegung $\beta_\alpha(x_i)$ der Booleschen Variablen $x_i, 1 \leq i \leq n$. Da eine Funktion mit n Argumenten, die jeweils zwei Werte annehmen können, insgesamt durch 2^n Argumentvarianten definiert wird, tragen 2^n konstante Abbildungen $y_{(\alpha)}$ zu ihrer Definition bei.
Der Index α ergibt sich durch die als Binärzahl $\alpha = (\beta(x_n) \ldots \beta(x_1))_2$ interpretierte Belegung $\beta(x_i)$ der Booleschen Variablen der konstanten Abbildung $y_{(\alpha)}$. So gilt für die Definition der ODER-Verknüpfung $f_{\text{ODER}}(x_1, x_2)$ die Wertetabelle

α	$\beta(x_2)$	$\beta(x_1)$	$y_{(\alpha)} = f(\beta_\alpha(x_2), \beta_\alpha(x_1))$
0	0	0	0
1	0	L	L
2	L	0	L
3	L	L	L

bzw. $g_{\text{ODER}}(0, \text{L}, \text{L}, \text{L}) = \text{L}$. Für jede andere Gestalt des Vektors der konstanten Abbildungen nimmt die Funktion g_{ODER} den Wert $y = 0$ an. Man bezeichnet die Funktion g_f,

$$g_f(y_{(0)}, \ldots, y_{(2^n-1)}) = \begin{cases} L & \text{, wenn für alle } \alpha : y_{(\alpha)} = f(\beta_\alpha(x_n), \ldots, \beta_\alpha(x_1)) \\ 0 & \text{sonst} \end{cases}$$

die *Indikatorfunktion* oder *charakteristische Funktion* der Booleschen Funktion f.
Hieraus folgt, daß es zu jeder Stelligkeit n, bzw. zu n Booleschen Variablen, genau 2^{2^n} n-stellige Boolesche Funktionen $f_\gamma(x_1, \ldots, x_n),\ 0 \leq \gamma \leq 2^{2^n} - 1$, gibt. Der Index γ der Booleschen Funktion f_γ ergibt sich durch die als Binärzahl $\gamma = (y_{(0)} \ldots y_{(2^n-1)})_2$ interpretierte Folge der konstanten Abbildung y_α, das heißt,

$$\gamma = \sum_{\alpha=0}^{2^n-1} y_{(\alpha)} 2^{2^n-1-\alpha}.$$

Also hat in diesem Schema die Funktion f_{ODER} den Index $\gamma = 7$, bzw. $f_{\text{ODER}} \equiv f_7$.
Der Wert einer Booleschen Funktion $f\gamma$,

$$y_\gamma = f_\gamma(x_1, \ldots, x_n)$$

ergibt sich also aus der Abbildung der Booleschen Variablen x_i auf die Ausgangsvariable $y_\gamma \in \mathbb{B}$ entsprechend den Regeln der Funktion in Abhängigkeit von der aktuellen Belegung $\beta(x_i)$ der Booleschen Variablen. Sind alle Belegungen konsistent, so daß $\forall\alpha : y_{(\alpha)} = f(\beta_\alpha(x_n), \ldots, \beta_\alpha(x_1))$, so folgt $g_f = \text{L}$, anderenfalls $g_f = 0$.

Unter den n-stelligen Booleschen Funktionen gibt es jeweils zwei spezielle, die konstant Null oder konstant Eins ergeben.

1. Nullfunktion: $y = f_0(x_1, \ldots, x_n) = 0,\ g_f(0_{(0)}, \ldots, 0_{(2^n-1)}) = \mathrm{L}$
2. Einsfunktion: $y = f_{2^{2^n}-1}(x_1, \ldots, x_n) = \mathrm{L},\ g_f(\mathrm{L}_{(0)}, \ldots, \mathrm{L}_{(2^n-1)}) = \mathrm{L}.$

Die ein- und zweistelligen Booleschen Funktionen sind wegen ihrer Bedeutung (insbesondere in der Logik) mit Namen belegt.

n=1: Funktion	Name	$\beta(x)$	0	L
		y_γ	$y_{(0)}$	$y_{(1)}$
$f_0(x) = 0$	Nullfunktion	y_0	0	0
$f_1(x) = x$	Identität	y_1	0	L
$f_2(x) = \overline{x}$	Negation	y_2	L	0
$f_3(x) = \mathrm{L}$	Einsfunktion	y_3	L	L

Die Nullfunktion realisiert unabhängig von der Eingabe $y_0 = 0$. Die Identität erzeugt eine Kopie der Eingabe, die Negation invertiert diese und die Einsfunktion ergibt den Wert $y_3 = \mathrm{L}$ unabhängig von der Eingabe.

n=2:

$\beta(x_2)$	0	0	L	L	$f(x_1, x_2)$	$f(x_1, x_2)$	
$\beta(x_1)$	0	L	0	L	Operationszeichen: $\wedge, \vee, \overline{\ }$	spez. Opz.	Name
y_γ	$y_{(0)}$	$y_{(1)}$	$y_{(2)}$	$y_{(3)}$			
f_0	0	0	0	0	$(x_1 \wedge \overline{x_1}) \wedge (x_2 \wedge \overline{x_2})$	0	Null
f_1	0	0	0	L	$x_1 \wedge x_2$	$x_1 \wedge x_2$	UND-Verknüpfung (AND)
f_2	0	0	L	0	$\overline{x_1} \wedge x_2$	$x_1 \not\Leftarrow x_2$	Inhibition
f_3	0	0	L	L	x_2	x_2	Transfer
f_4	0	L	0	0	$x_1 \wedge \overline{x_2}$	$x_1 \not\Rightarrow x_2$	Inhibition
f_5	0	L	0	L	x_1	x_1	Transfer
f_6	0	L	L	0	$(x_1 \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2)$	$x_1 \not\Leftrightarrow x_2$	XOR-Verkn. / Antivalenz
f_7	0	L	L	L	$x_1 \vee x_2$	$x_1 \vee x_2$	ODER-Verknüpfung (OR)
f_8	L	0	0	0	$\overline{x_1} \wedge \overline{x_2}$	$\overline{x_1 \vee x_2}$	NOR-Verknüpfung
f_9	L	0	0	L	$(x_1 \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$	$x_1 \Leftrightarrow x_2$	Äquivalenz
f_{10}	L	0	L	0	$\overline{x_1}$	$\overline{x_1}$	NOT-Verkn. / Negation
f_{11}	L	0	L	L	$\overline{x_1} \vee x_2$	$x_1 \Rightarrow x_2$	Implikation
f_{12}	L	L	0	0	$\overline{x_2}$	$\overline{x_2}$	NOT-Verkn. / Negation
f_{13}	L	L	0	L	$x_1 \vee \overline{x_2}$	$x_1 \Leftarrow x_2$	Implikation
f_{14}	L	L	L	0	$\overline{x_1} \vee \overline{x_2}$	$\overline{x_1 \wedge x_2}$	NAND-Verknüpfung
f_{15}	L	L	L	L	$(x_1 \vee \overline{x_1}) \vee (x_2 \vee \overline{x_2})$	L	Eins

Jede n-stellige Boolesche Funktion läßt sich in einem $(n+1)$-dimensionalen Einheits-Hyperwürfel darstellen. Dies läßt sich aber nur bis $n = 3$ visualisieren (s. Abb. 5.10 und 5.11). Jede der 2^{2^n} n-stelligen Booleschen Funktionen wird dabei durch 2^n Argumentvarianten repräsentiert. Diese repräsentieren den Binärcode des Index γ der Funktion f_γ.

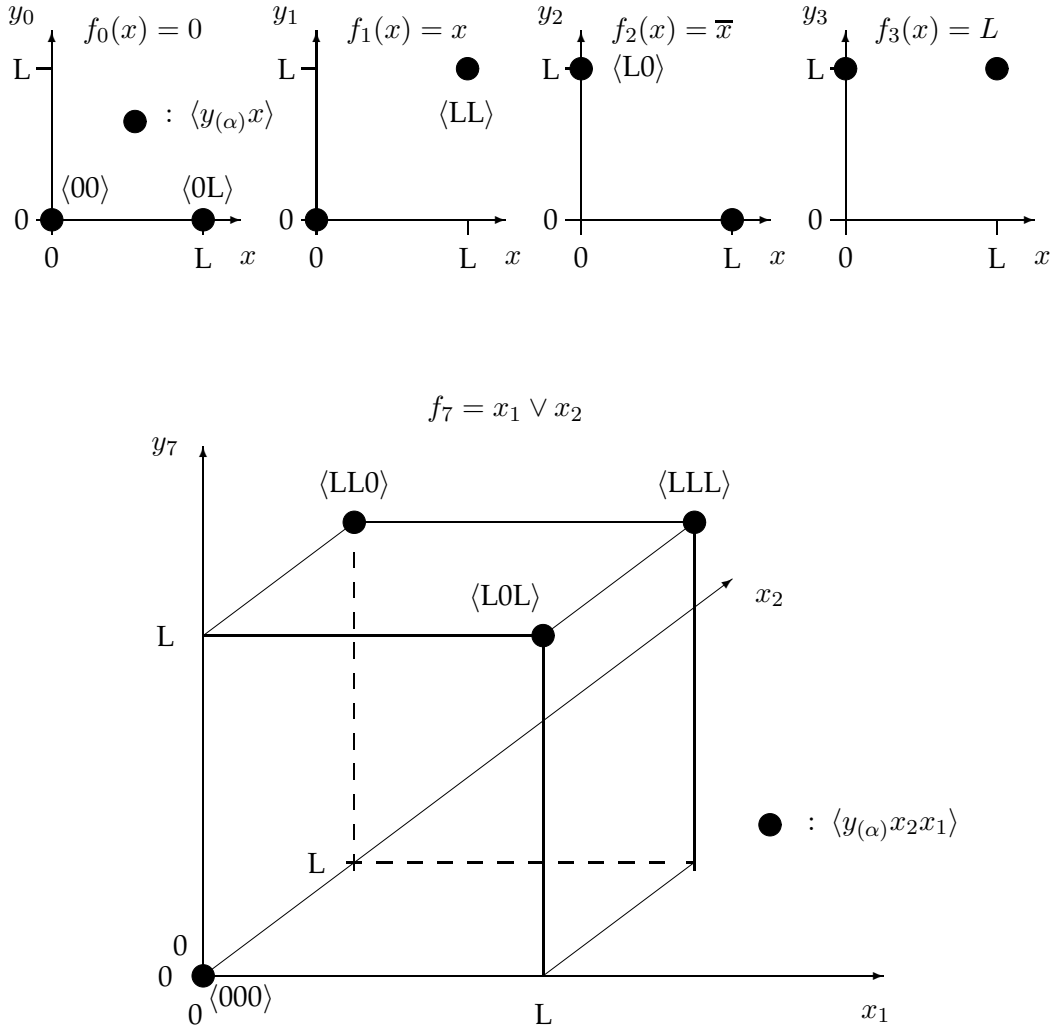


Abbildung 5.10: Visualisierung der Hyperwürfel (1)

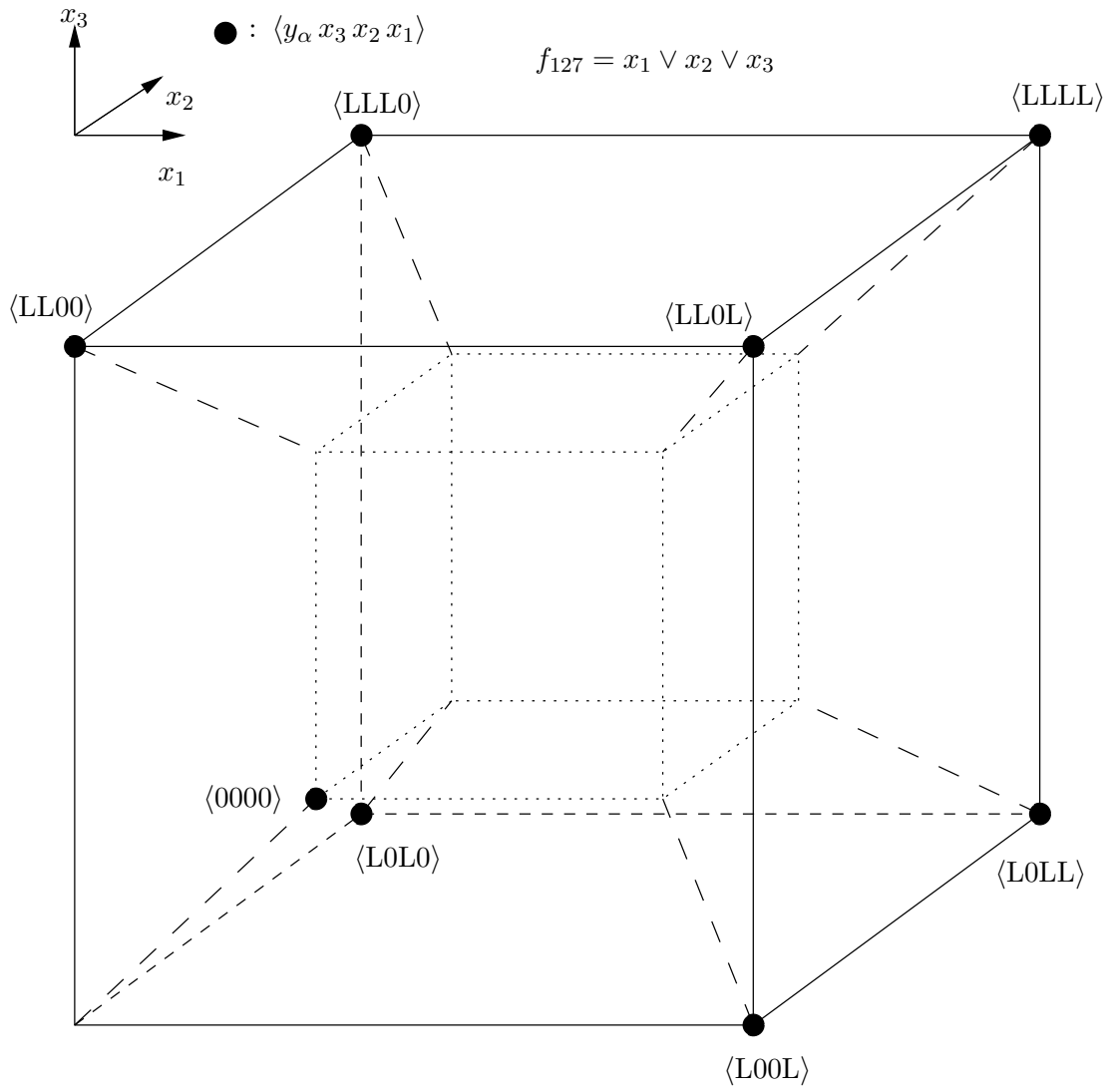


Abbildung 5.11: Visualisierung der Hyperwürfel (2)

Beispiele:

1. Die Funktion $f_\gamma = x_1$ op x_2 erhält den Index $\gamma = 7$. Damit repräsentiert f_7 die zweistellige ODER-Verknüpfung. Abbildung 5.10 zeigt die Darstellung von f_7 im Einheitswürfel.

α	$\beta(x_2)$	$\beta(x_1)$	$y_{(\alpha)}$	
0	0	0	0	$0 \cdot 2^3$
1	0	L	L	$1 \cdot 2^2$
2	L	0	L	$1 \cdot 2^1$
3	L	L	L	$1 \cdot 2^0$

$$\gamma = 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 7$$

2. Die Funktion $f_\gamma = x_1$ op x_2 op x_3 erhält den Index $\gamma = 127$. Damit repräsentiert f_{127} die dreistellige ODER-Verknüpfung.

α	$\beta(x_3)$	$\beta(x_2)$	$\beta(x_1)$	$y_{(\alpha)}$	
0	0	0	0	0	$0 \cdot 2^7$
1	0	0	L	L	$1 \cdot 2^6$
2	0	L	0	L	$1 \cdot 2^5$
3	0	L	L	L	$1 \cdot 2^4$
4	L	0	0	L	$1 \cdot 2^3$
5	L	0	L	L	$1 \cdot 2^2$
6	L	L	0	L	$1 \cdot 2^1$
7	L	L	L	L	$1 \cdot 2^0$

Im 4-dimensionalen Einheits-Hyperwürfel (s. Abb. 5.11) repräsentiert der äußere Kubus den Fall $y_{(\alpha)} = \mathrm{L}$ und der innere Kubus wird für $y_{(\alpha)} = 0$ besetzt. Hier steht der einzige Eintrag für die konstante Abbildung $0 = 0 \vee 0 \vee 0$.

5.1.3 Boolesche Terme

Ausdrücke der Art

$$t \equiv f(x_1, \cdots, x_n)$$

nennt man Boolesche Terme.
Nach Abschnitt 3.2.3 sind Boolesche Terme

1. die Konstanten 0 oder L (atomare Boolesche Terme)

2. die Eingangsvariablen x (atomare Boolesche Terme)
3. die aus den Eingangsvariablen x_i, $i = 1, \cdots, n$ durch Verknüpfungsoperationen "NOT", "AND" und "OR" gebildeten Ausdrücke t,
4. die aus t durch Negation gebildeten Ausdrücke $\overline{t}$,
5. die durch alle möglichen zweistelligen Operationen gebildeten Ausdrücke der Art t_1 op t_2, wenn t_1 und t_2 Boolesche Terme sind.

Jede Schaltfunktion läßt sich durch wenigstens einen Booleschen Term realisieren. Die Darstellung einer Schaltfunktion durch einen Booleschen Term ist allerdings nicht eindeutig (siehe Beispiel Schaltpläne der 2-von-3-Funktion, Seite 302). Die Definition Boolescher Terme verdeutlicht ein induktives Konstruktionsprinzip aus der Definition atomarer Boolescher Terme, das impliziert:

> Wenn die Gesetze der Booleschen Algebra für atomare Boolesche Terme a, b gelten, so gelten sie auch für die aus ihnen gebildeten Terme $t = a$ op b, für die Terme $t_3 = t_1$ op t_2, u.s.w.

Gesetze der Schaltalgebra

Für $x, y, z, \in \mathbb{B} = \{0, \mathrm{L}\}$ gilt

1.	*Idempotenz*	$x \wedge x = x$
		$x \vee x = x$
2.	*Kommutativität*	$x \wedge y = y \wedge x$
		$x \vee y = y \vee x$
3.	*Assoziativität*	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$
		$(x \vee y) \vee z = x \vee (y \vee z)$
4.	*Absorption*	$x \wedge (x \vee y) = x$
		$x \vee (x \wedge y) = x$
5.	*Distributivität*	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
		$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
6.	*neutrale Elemente*	$x \wedge \mathrm{L} = x$
		$x \vee 0 = x$
7.	*Komplement*	$x \wedge \overline{x} = 0 = \overline{\mathrm{L}}$
		$x \vee \overline{x} = \mathrm{L} = \overline{0}$
8.	*Involution*	$\overline{\overline{x}} = x$
9.	*De Morgan*	$\overline{x \wedge y} = \overline{x} \vee \overline{y}$
		$\overline{x \vee y} = \overline{x} \wedge \overline{y}$

Deshalb stehen in der Tabelle der Gesetze der Schaltalgebra die Variablen x, y und z sowohl für atomare Boolesche Terme als auch für Boolesche Terme selbst. Das impliziert, daß jedem Term durch $y = t,\ y \in \mathbb{B}$ ein Wahrheitswert bzw. Schaltzustand zugeordnet ist.
Die Umkehrung der Absorptionsregeln läßt sich auch als *Entwicklungsregeln* bezeichnen. Entwicklungsregeln sind oft nützlich, um mehr Freiheitsgrade bei der Umwandlung von Termen zu erhalten. Weitere nützliche Absorptionsregeln:

a) $(x \wedge y) \vee (x \wedge \overline{y}) = x$

b) $(x \vee y) \wedge (x \vee \overline{y}) = x$

Diese Regeln lassen sich allein aus den Gesetzen der Schaltalgebra beweisen. In Verallgemeinerung der de Morganschen Regeln gilt:

a) $\overline{\bigvee_{i=1}^{n} x_i} = \bigwedge_{i=0}^{n} \overline{x_i}$

b) $\overline{\bigwedge_{i=1}^{n} x_i} = \bigvee_{i=1}^{n} \overline{x_i}.$

Dabei steht $\bigvee_{i=1}^{n} x_i$ für $x_1 \vee x_2 \vee \cdots \vee x_n$.

Beweis. a) wird mittels vollständiger Induktion über n gezeigt.

$n = 1$: trivial

$n = 2$: entspricht de Morganschen Regeln, wird mittels Wahrheitstabelle gezeigt

x_1	x_2	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_1} \wedge \overline{x_2}$	$x_1 \vee x_2$	$\overline{x_1 \vee x_2}$
0	0	L	L	L	0	L
0	L	L	0	0	L	0
L	0	0	L	0	L	0
L	L	0	0	0	L	0

$n = k$: Induktionsannahme lautet "Gleichung gilt für alle n bis $n = k$".

$$t_k = \bigvee_{i=1}^{k} x_i \ \rightarrow\ t_{k+1} = t_k \vee x_{k+1}$$

Dann ist wegen de Morgan

$$\overline{t_{k+1}} = \overline{t_k} \wedge \overline{x_{k+1}}.$$

Nach Induktionsannahme ist

$$\overline{t_k} = \bigwedge_{i=1}^{k} \overline{x_i},$$

also ist

$$\overline{t_{k+1}} = \bigwedge_{i=1}^{k+1} \overline{x_i}.$$

□

Bei der Berechnung des Wertes Boolescher Terme und bei ihrer Umwandlung mit Hilfe der Gesetze der Schaltalgebra sind die Vorrangregeln der Schaltoperationen zu beachten. Diese lauten

"NOT" vor "AND" vor "OR" vor $\{\Rightarrow, \Leftrightarrow, \cdots\}$.

Vorrangregeln für Boolesche Operatoren

1. Der einstellige Operator $\neg$ bindet stärker als die zweistelligen Operatoren $(\wedge, \vee, \Rightarrow, \Leftrightarrow)$.
2. Der Operator $\wedge$ bindet stärker als $\vee$ ("Punktrechnung geht vor Strichrechnung"). Am schwächsten binden $\Rightarrow$ und $\Leftrightarrow$.
3. Ungeklammerte Boolesche Terme, die durch Operatoren gleicher Bindungsstärke erzeugt werden, werden von links nach rechts geklammert.

Bei Beachtung der starken Bindung von "AND" werden wir auf die explizite Angabe des Zeichens "$\wedge$" von jetzt an meist verzichten.

Beispiel:

$$\begin{aligned} a \vee (b \wedge c) &= a \vee b \wedge c &&= a \vee bc \\ (a \vee b) \wedge c &= (a \vee b)c &&\neq a \vee b \wedge c \end{aligned}$$

Die Darstellung von Schaltfunktionen durch Boolesche Terme ist nicht eindeutig.

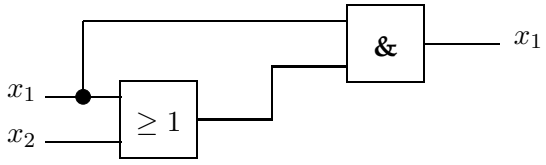
Definition: (Äquivalenz Boolescher Terme)

Zwei Boolesche Terme $t_1 \equiv f(x_1, \cdots, x_m)$ und $t_2 \equiv g(x_1, \cdots, x_n)$, $n \geq m$, sind äquivalent, d.h. $t_1 \Leftrightarrow t_2$, wenn für alle Belegungen der Eingangsvariablen der Booleschen Funktionen f und g die Terme gleiche Werte annehmen, also $y_1 = y_2$.

Beispiele:

a) Beweis des Absorptionsgesetzes: $t_1 = x_1(x_1 \vee x_2)$, $t_2 = x_1$

x_1	x_2	$x_1 \vee x_2$	y_1	y_2
0	0	0	0	0
0	L	L	0	0
L	0	L	L	L
L	L	L	L	L



b) $t_1 = x_1x_2 \vee \overline{x_1}$, $t_2 = \overline{x_1} \vee x_2$

x_1	x_2	$\overline{x_1}$	$x_1 \wedge x_2$	y_1	y_2
0	0	L	0	L	L
0	L	L	0	L	L
L	0	0	0	0	0
L	L	0	L	L	L

Der Term t_1 läßt sich auf $t_2 = f_{13}(x_1, x_2)$ reduzieren!
Unter Anwendung der algebraischen Gesetzmäßigkeiten folgt:

$$
\begin{array}{lcll}
x_1x_2 \vee \overline{x_1} & = & (x_1 \wedge x_2) \vee \overline{x_1} & \\
& = & \overline{x_1} \vee (x_1 \wedge x_2) & \text{Kommutativität} \\
& = & (\overline{x_1} \vee x_1) \wedge (\overline{x_1} \vee x_2) & \text{Distributivität} \\
& = & \mathrm{L} \wedge (\overline{x_1} \vee x_2) & \text{Komplement} \\
& = & \overline{x_1} \vee x_2 & \text{neutrales Element}
\end{array}
$$

Die Äquivalenz Boolescher Terme zeigt man durch

a) die Berechnung von Wahrheitstafeln (für $n \leq 3$).

b) algebraisch unter Anwendung der Gesetze der Schaltalgebra.

Beispiel:

Beweis der Gültigkeit von $(x_1 \vee x_2)(x_3 \vee x_4) = x_1x_3 \vee x_1x_4 \vee x_2x_3 \vee x_2x_4$

Setze $e = x_1 \vee x_2$

Hieraus folgt

$$\begin{aligned} e(x_3 \vee x_4) &= ex_3 \vee ex_4 \\ &= (x_1 \vee x_2)x_3 \vee (x_1 \vee x_2)x_4 \\ &= x_1x_3 \vee x_2x_3 \vee x_1x_4 \vee x_2x_4 \end{aligned}$$

Da zu jedem Verband ein dualer Verband zugeordnet werden kann, gilt der Shannonsche Inversionssatz als Verallgemeinerung der de Morganschen Regeln:

Shannonscher Inversionsatz

Jede nur mit Operatoren AND, OR und NOT gebildete Schaltfunktion kann dadurch negiert werden, daß die Operatoren für AND und OR miteinander vertauscht werden und jedes Literal negiert wird. Außerdem sind die Konstanten 0 und L zu vertauschen.

Beispiel:

Gegeben: $t_1 = x_1(x_2\overline{x}_3 \vee \overline{x}_2x_3)$ und $t_2 = \overline{x}_1 \vee (\overline{x}_2 \vee x_3)(x_2 \vee \overline{x}_3)$

Zeige, daß $t_2 = \overline{t_1}$

$$\begin{aligned} \overline{t_1} &= \overline{x_1(x_2\overline{x}_3 \vee \overline{x}_2x_3)} \\ &= \overline{x}_1 \vee \overline{(x_2\overline{x}_3 \vee \overline{x}_2x_3)} \\ &= \overline{x}_1 \vee \overline{(x_2\overline{x}_3)}\,\overline{(\overline{x}_2x_3)} \\ &= \overline{x}_1 \vee (\overline{x}_2 \vee \overline{\overline{x}_3})(\overline{\overline{x}_2} \vee \overline{x}_3) \\ &= \overline{x}_1 \vee (\overline{x}_2 \vee x_3)(x_2 \vee \overline{x}_3) \\ &= t_2 \end{aligned}$$

Es gilt auch $t_1 = x_1(x_2 \not\Leftrightarrow x_3)$ und $t_2 = \overline{x}_1 \vee (x_2 \Leftrightarrow x_3)$.

Zeige, daß $t_2 = \overline{t_1}$

$$\begin{aligned} \overline{t_1} &= \overline{x_1(x_2 \not\Leftrightarrow x_3)} \\ &= \overline{x}_1 \vee \overline{(x_2 \not\Leftrightarrow x_3)} \\ &= \overline{x}_1 \vee (x_2 \Leftrightarrow x_3) \\ &= t_2 \end{aligned}$$

Der Shannonsche Inversionssatz stellt eine Abkürzung der rekursiven Anwendung der de Morganschen Regeln zur Invertierung eines Termes bis zur Invertierung der atomaren Booleschen Terme dar.

Beispiel: S1/1

Beweis der Äquivalenz der Darstellung der 2-von-3-Mehrheitsfunktion $t = f(x_1, x_2, x_3)$ in den Varianten t_1 und t_2 mit

$$t_1 = x_1x_2 \vee x_1x_3 \vee x_2x_3 \text{ und } t_2 = x_1x_2 \not\Leftrightarrow x_1x_3 \not\Leftrightarrow x_2x_3$$

a) mittels Wahrheitstafel:

			A	B	C	D	y_1	E	y_2
x_1	x_2	x_3	x_1x_2	x_1x_3	x_2x_3	$A \vee B$	$D \vee C$	$A \not\Leftrightarrow B$	$E \not\Leftrightarrow C$
0	0	0	0	0	0	0	0	0	0
0	0	L	0	0	0	0	0	0	0
0	L	0	0	0	0	0	0	0	0
0	L	L	0	0	L	0	L	0	L
L	0	0	0	0	0	0	0	0	0
L	0	L	0	L	0	L	L	L	L
L	L	0	L	0	0	L	L	L	L
L	L	L	L	L	L	L	L	0	L

b) algebraisch:

Zurückführen von t_2 auf t_1:

$$\begin{aligned}
x_1x_2 \not\Leftrightarrow x_1x_3 &= x_1(x_2 \not\Leftrightarrow x_3) \\
&= x_1(x_2\overline{x}_3 \vee \overline{x}_2x_3) \\
(x_1x_2 \not\Leftrightarrow x_1x_3) \not\Leftrightarrow x_2x_3 &= x_1(x_2\overline{x}_3 \vee \overline{x}_2x_3)\overline{x_2x_3} \vee \overline{x_1(x_2\overline{x}_3 \vee \overline{x}_2x_3)}x_2x_3 \\
&= x_1(x_2\overline{x}_3 \vee \overline{x}_2x_3)(\overline{x}_2 \vee \overline{x}_3) \\
&\quad \vee (\overline{x}_1 \vee \underbrace{\overline{(x_2\overline{x}_3 \vee \overline{x}_2x_3)}}_{A})x_2x_3
\end{aligned}$$

Es gilt:

$$A = \overline{x_2\overline{x}_3} \wedge \overline{\overline{x}_2 x_3} = (\overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3)$$

Damit folgt:

$$\begin{aligned} t_2 &= B \vee C \\ \text{mit} & \\ B &= x_1(x_2\overline{x}_3 \vee \overline{x}_2 x_3)(\overline{x}_2 \vee \overline{x}_3) \\ C &= (\overline{x}_1 \vee (\overline{x}_2 \vee x_3)(x_2 \vee \overline{x}_3))x_2x_3. \end{aligned}$$

Damit erhält man:

$$\begin{aligned} B &= x_1x_2\overline{x}_2\overline{x}_3 \vee x_1\overline{x}_2\overline{x}_2x_3 \vee x_1x_2\overline{x}_3\overline{x}_3 \vee x_1\overline{x}_2x_3\overline{x}_3 \\ &= 0 \vee x_1\overline{x}_2x_3 \vee x_1x_2\overline{x}_3 \vee 0 \\ C &= \overline{x}_1x_2x_3 \vee \overline{x}_2x_2x_2x_3 \vee \overline{x}_2\overline{x}_3x_2x_3 \vee x_3x_2x_2x_3 \vee x_3\overline{x}_3x_2x_3 \\ &= \overline{x}_1x_2x_3 \vee 0 \vee 0 \vee x_2x_3 \vee 0 \end{aligned}$$

Also

$$\begin{aligned} t_2 &= x_1\overline{x}_2x_3 \vee x_1x_2\overline{x}_3 \vee \underbrace{\overline{x}_1x_2x_3 \vee x_2x_3}_{x_2x_3 \text{ wegen Absorption } A \vee AB = A} \\ &= x_1\overline{x}_2x_3 \vee x_1x_2\overline{x}_3 \vee x_2x_3 \\ &= (x_1\overline{x}_2x_3 \vee x_1x_2x_3) \vee (x_1x_2\overline{x}_3 \vee x_1x_2x_3) \vee x_2x_3 \\ &\quad \text{wegen 1. Expansion: } x_2x_3 = x_1x_2x_3 \vee x_2x_3 \\ &\quad \text{wegen 2. Idempotenz: } x_2x_3 = x_1x_2x_3 \vee x_1x_2x_3 \vee x_2x_3 \\ &= x_1x_2(\overline{x}_3 \vee x_3) \vee x_1x_3(\overline{x}_2 \vee x_2) \vee x_2x_3 \\ &= x_1x_2 \vee x_1x_3 \vee x_2x_3 \end{aligned}$$

□

Die Negation einzelner Variabler einer Schaltfunktion ist aus Gründen der technischen Realisierung oft erforderlich. Hierfür wird die Involution $\overline{\overline{t}} = t$ ausgenutzt.

Beispiel:

$$t = \overline{x_1 \vee x_2} \vee x_3 \;=\; \overline{x_1 \vee x_2} \vee \overline{\overline{x}_3} \;=\; \overline{(x_1 \vee x_2)\overline{x}_3}$$

Shannonscher Entwicklungssatz

Jede Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ kann für jede ihrer Variablen x_i durch

$$\begin{aligned} f(x_1, \cdots, x_i, \cdots, x_n) &= x_i f(x_1, \cdots, x_{i-1}, \mathrm{L}, x_{i+1}, \cdots, x_n) \\ &\vee \ \overline{x}_i f(x_1, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n) \end{aligned}$$

dargestellt werden.

Beweis. Für $x_i = \mathrm{L}$ verschwindet die zweite Hälfte der Disjunktion wegen $0 \wedge t = 0$ und wegen $L \wedge t = t$ und $0 \vee t = t$ sind linke und rechte Seite gleich. Für $x_i = 0$ verschwindet die erste Hälfte der Disjunktion. In beiden Fällen sind linke und rechte Seite der Gleichung identisch. □

Als Folge des Shannonschen Entwicklungsatzes gilt außerdem der spezielle Satz:

Ist eine Boolesche Funktion f darstellbar als $f = x_i \vee f_i$, wobei $f_i(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$ eine Boolesche Funktion von $n-1$ Variablen ist, so gilt

$$f = x_i \vee \overline{x}_i f_i.$$

Beweis. Anwendung des Entwicklungssatzes auf $f = x_i \vee f_i$:

$$f = x_i(\mathrm{L} \vee f_i) \vee \overline{x}_i f_i$$

Da $\mathrm{L} \vee f_i = \mathrm{L}$, folgt

$$f = x_i \vee \overline{x}_i f_i$$

□

Beispiel:

$$\begin{aligned} f &= x_1 \vee x_2 \vee x_3 \\ &= x_1 \vee \overline{x}_1(x_2 \vee x_3) && \text{1. Anwendung bzgl. } f_1 = x_2 \vee x_3 \\ &= x_1 \vee \overline{x}_1(x_2 \vee \overline{x}_2 x_3) && \text{2. Anwendung bzgl. } f_2 = x_3 \\ &= x_1 \vee \overline{x}_1 x_2 \vee \overline{x}_1 \overline{x}_2 x_3 \end{aligned}$$

Die rekursive Anwendung des Entwicklungssatzes führt schrittweise zur konjunktiven Ausklammerung aller Variablen. Hierauf werden wir im folgenden Abschnitt zurückkommen, wenn vollständige Operatorensysteme behandelt werden.

5.2 Darstellung, Synthese und Analyse von Schaltfunktionen

Die Darstellung Boolescher Funktionen von n Veränderlichen ist nicht eindeutig. Dies ist vom praktischen Standpunkt aus kein Nachteil. Vielmehr entsteht hieraus die Möglichkeit, optimale Realisierungen von Schaltfunktionen zu erhalten, die eine Anpassung an die verfügbaren Ressourcen erlauben.
Als Ressourcen-Kriterien kommen zum Beispiel in Frage:

- technologische Einfachheit der Gatterherstellung (begrenzt die Vielfalt logischer Grundfunktionen)
- technologische Einfachheit der Schaltnetzherstellung (kann ein Aufblähen der verwendeten Gatter (Redundanz) zugunsten struktureller Gleichförmigkeit bedeuten)
- Kosten- und/oder Platzersparnis/Energieersparnis (kann Schaltnetz mit minimaler Gatterzahl erfordern)

Oft muß eine Balance zwischen widersprüchlichen Entwurfszielen hergestellt werden.

5.2.1 Vollständige Verknüpfungsbasen

Hier sollen Beispiele für Mengen Boolescher Funktionen angegeben werden, die sämtliche Booleschen Funktionen von n Variablen darzustellen gestatten. Solche Operatormengen werden vollständige Verknüpfungsbasen genannt.

Definition: (vollständige Verknüpfungsbasis)

Eine Menge $\mathfrak{B} = \{f_1, \ldots, f_m\}$ von p-stelligen Booleschen Funktionen heißt funktional vollständig, wenn sich jede n-stellige Boolesche Funktion, $n \geq p$, allein durch Einsetzungen bzw. Kompositionen von Funktionen aus $\mathfrak{B}$ darstellen läßt.

Das Funktionensystem $\{\vee, \wedge, \bar{}\}$ ist funktional vollständig.

Beweis. Maximal n-malige Anwendung des Shannonschen Entwicklungssatzes.

Beispiel: S1/2

Darstellung der 2-von-3-Mehrheitsfunktion

$$f(x_1, x_2, x_3) = x_1x_2 \not\Leftrightarrow x_1x_3 \not\Leftrightarrow x_2x_3$$

im System $\{\vee, \wedge, \bar{}\}$.

Weg: Anwendung des Entwicklungssatzes (nicht auf $\vee, \wedge, \bar{}$ beschränkt).

x_1 ausklammern:

$$f = x_1(x_2 \not\Leftrightarrow x_3 \not\Leftrightarrow x_2x_3) \vee \overline{x}_1x_2x_3$$

Teilaufgabe: $x_2 \not\Leftrightarrow x_3 \not\Leftrightarrow x_2x_3 = x_2 \vee x_3$

x_2	x_3	$x_2 \not\Leftrightarrow x_3$	x_2x_3	$(x_2 \not\Leftrightarrow x_3) \not\Leftrightarrow x_2x_3$	$x_2 \vee x_3$
0	0	0	0	0	0
0	L	L	0	L	L
L	0	L	0	L	L
L	L	0	L	L	L

Hieraus folgt

$$f = x_1(x_2 \vee x_3) \vee \overline{x}_1x_2x_3.$$

Nach dem Entwicklungssatz gilt für die Ausklammerung von x_i

$$x_i \vee x_k = x_i(\mathrm{L} \vee x_k) \vee \overline{x}_ix_k = x_i \vee \overline{x}_ix_k.$$

Damit folgt:

$$f = x_1(x_2 \vee \overline{x}_2x_3) \vee \overline{x}_1x_2x_3 = x_1x_2 \vee x_1\overline{x}_2x_3 \vee \overline{x}_1x_2x_3.$$

Wegen Absorption gilt: $x_1x_2 = x_1x_2x_3 \vee x_1x_2$,
wegen Idempotenz gilt: $x_1x_2 = x_1x_2x_3 \vee x_1x_2x_3 \vee x_1x_2$.

Damit folgt:

$$\begin{aligned} f &= x_1x_2 \vee (x_1\overline{x}_2x_3 \vee x_1x_2x_3) \vee (\overline{x}_1x_2x_3 \vee x_1x_2x_3) \\ &= x_1x_2 \vee x_1x_3(\overline{x}_2 \vee x_2) \vee x_2x_3(\overline{x}_1 \vee x_1) \\ &= x_1x_2 \vee x_1x_3 \vee x_2x_3. \end{aligned}$$

Obwohl im obigen Beispiel die Negation nicht zur Darstellung benötigt wird, ist sie prinzipiell nicht verzichtbar. Aber einer der zweistelligen Operatoren ist verzichtbar.

Die Funktionensysteme $\{\vee, \bar{}\}$ und $\{\wedge, \bar{}\}$ sind funktional vollständig.

Beweis. Nach den de Morganschen Regeln und wegen der Involution gilt

a) $x \wedge y = \overline{\overline{x \wedge y}} = \overline{\overline{x} \vee \overline{y}}$

b) $x \vee y = \overline{\overline{x \vee y}} = \overline{\overline{x} \wedge \overline{y}}$

□

Das obige Beispiel legt nahe (*Ringsummenentwicklung*):

Das Funktionensystem $\{\not\Leftrightarrow, \wedge, \bar{}\}$ ist funktional vollständig.

Es existiert aber auch das komplementfreie System $\{\not\Leftrightarrow, \wedge, \mathrm{L}\}$. Dieses System ist äquivalent dem System $\{\not\Leftrightarrow, \wedge\}$, da $x \wedge \mathrm{L} = x$.

Beispiel:

$$\begin{aligned} f &= \overline{x}_1 x_2 x_3 \vee x_1 \overline{x}_2 x_3 \vee x_1 x_2 x_3 \\ &= \overline{x}_1 x_2 x_3 \not\Leftrightarrow x_1 \overline{x}_2 x_3 \not\Leftrightarrow x_1 x_2 x_3. \end{aligned}$$

Wegen $\overline{x} = x \not\Leftrightarrow \mathrm{L}$

L	x	$x \not\Leftrightarrow \mathrm{L}$	$\overline{x}$
L	0	L	L
L	L	0	0

folgt die Repräsentation im komplementfreien System $\{\not\Leftrightarrow, \wedge, \mathrm{L}\}$:

$$\begin{aligned} f &= (x_1 \not\Leftrightarrow \mathrm{L}) x_2 x_3 \not\Leftrightarrow x_1 (x_2 \not\Leftrightarrow \mathrm{L}) x_3 \not\Leftrightarrow x_1 x_2 x_3 \\ &= x_1 x_2 x_3 \not\Leftrightarrow x_2 x_3 \not\Leftrightarrow x_1 x_2 x_3 \not\Leftrightarrow x_1 x_3 \not\Leftrightarrow x_1 x_2 x_3. \end{aligned}$$

Wegen $x \not\Leftrightarrow x = 0$ und wegen $x \not\Leftrightarrow 0 = x$ folgt

$$f = x_1 x_3 \not\Leftrightarrow x_2 x_3 \not\Leftrightarrow x_1 x_2 x_3.$$

Das heißt, in der Ringsummenentwicklung können die negierten Variablen weggelassen werden.

Jede n-stellige Boolesche Funktion ist allein durch Terme der Funktionen "NAND": Δ bzw. $\overline{\wedge}$ oder "NOR": ∇ bzw. $\overline{\vee}$ darstellbar.

Beweis. Es wird gezeigt, daß die Funktionen AND, OR und NOT durch NAND bzw. NOR ausdrückbar sind.

a) NOR: ∇

1. $\begin{aligned} \overline{x} &= \overline{x} \wedge \overline{x} && \text{Idempotenz} \\ &= \overline{x \vee x} && \text{de Morgan} \\ &= x \nabla x && \text{Def. NOR} \end{aligned}$

2. $\begin{aligned} x \wedge y &= \overline{\overline{x \wedge y}} && \text{Involution} \\ &= \overline{\overline{x} \vee \overline{y}} && \text{de Morgan} \\ &= \overline{(x \nabla x) \vee (y \nabla y)} && (1) \\ &= (x \nabla x) \nabla (y \nabla y) && \text{Def. NOR} \end{aligned}$

3. $\begin{aligned} x \vee y &= \overline{\overline{x \vee y}} && \text{Involution} \\ &= \overline{\overline{x} \wedge \overline{y}} && \text{de Morgan} \\ &= (\overline{x} \wedge \overline{y}) \nabla (\overline{x} \wedge \overline{y}) && (1) \text{ für } z = \overline{x} \wedge \overline{y} \\ &= \overline{(x \vee y)} \nabla \overline{(x \vee y)} && \text{de Morgan} \\ &= (x \nabla y) \nabla (x \nabla y) && \text{Def. NOR} \end{aligned}$

b) NAND: Δ analog zu NOR.

□

Die Funktionen NAND und NOR sind die einzigen Verknüpfungsbasen mit dieser Eigenschaft. Dies soll in folgendem Satz formuliert werden:

> Ist f eine zweistellige Boolesche Funktion, die es gestattet, jede n-stellige Boolesche Funktion durch ihre Variablen und f zu beschreiben, so muß $f = \nabla$ oder $f = \Delta$ sein.

Beweis. Zu beweisen sind die Wahrheitstafeln

∇	0	L
0	L	0
L	0	0

Δ	0	L
0	L	L
L	L	0

für alle Belegungen der Variablen $x, y \in \mathbb{B}$ der Funktion $f(x, y)$.

a) $f(0,0) = \mathrm{L}$, da $f(0,0) = 0$ nicht die Darstellung der Negation impliziert (Forderung der Nichtmonotonie):

Wenn $f(0,0) = 0$, ist jeder Ausdruck, in dem nur 0 und f vorkommt, vom Wert 0. Dies steht aber wegen $\overline{0} = \mathrm{L}$ im Widerspruch zur Darstellbarkeit der Negation.

b) $f(\mathrm{L},\mathrm{L}) = 0$, da $f(\mathrm{L},\mathrm{L}) = \mathrm{L}$ ebenfalls wegen $\overline{\mathrm{L}} = 0$ nicht die Darstellung der Negation erlaubt.

c) $f(0,\mathrm{L}) = 0$ und $f(\mathrm{L},0) = \mathrm{L}$: Dies bedeutet $f(x,y) = \overline{y}$ bzw. $f(x,y) = x$ und steht im Widerspruch zur Voraussetzung, daß durch f jede Funktion darstellbar ist.

d) $f(0,\mathrm{L}) = L$ und $f(\mathrm{L},0) = 0$: Dies bedeutet $f(x,y) = y$ bzw. $f(x,y) = \overline{x}$ und führt zum gleichen Widerspruch wie c).

e) $f(0,\mathrm{L}) = 0$ und $f(\mathrm{L},0) = 0$: Dies entspricht dem Fall $f = \nabla$.

f) $f(0,\mathrm{L}) = \mathrm{L}$ und $f(\mathrm{L},0) = \mathrm{L}$: Dies entspricht dem Fall $f = \Delta$.

Damit sind nur die durch die Funktionen Δ bzw. ∇ realisierten Abbildungen geeignet, allein vollständige Verknüpfungsbasen $\{\Delta\}$ bzw. $\{\nabla\}$ für alle Booleschen Funktionen zu bilden. □

Es ist also möglich, jede Boolesche Funktion allein durch NAND-Gatter bzw. NOR-Gatter zu realisieren.

5.2.2 Karnaugh-Veitch-Diagramme

Im letzten Abschnitt wurden algebraische Darstellungsweisen für Boolesche Funktionen mit unterschiedlichen Verknüpfungsbasen behandelt. Wir kennen außerdem die Wahrheitstafel als tabellarische Darstellungsform einer Booleschen Funktion.

Das Karnaugh-Veitch-Diagramm oder KV-Diagramm ist eine graphische Darstellung von Wahrheitstafeln oder Booleschen Funktionen. Wir führen es an dieser Stelle ein, weil es sich insbesondere für die übersichtliche Darstellung konjunktiver und disjunktiver Terme (Abschnitt 5.2.3) und ihrer Vereinfachung (Abschnitt 5.3) auf graphischem Wege bewährt.

Definition: (KV-Diagramme)

Ein KV-Diagramm von $f : \mathbb{B}^n \to \mathbb{B}$ ist die graphische Darstellung der Wertetabelle von f, wobei jeder der 2^n konstanten Abbildungen $y_{(\alpha)} = f(\beta_\alpha(x_n), \ldots, \beta_\alpha(x_1)),\ y_{(\alpha)} \in \mathbb{B}$, für alle Belegungen β der Booleschen Variablen $x_i \in \mathbb{B},\ i = 1, \ldots, n$, ein Feld im KV-Diagramm entspricht.

Die Inhalte der Felder tragen die Werte $y_{(\alpha)} \in \mathbb{B}$. Die Felder werden numeriert mit der als Binärzahl $\alpha = (\beta_\alpha(x_n)\beta_\alpha(x_{n-1}) \ldots \beta_\alpha(x_1))_2$ interpretierten Belegung der Booleschen Variablen der konstanten Abbildung $y_{(\alpha)}$. Dabei erfolgt die Anordnung der Felder in der Weise, daß sich benachbarte Felder nur in der Belegung einer Variablen unterscheiden und daß die Ränder des KV-Diagramms als toroidal verbunden gedacht werden.

Konstruktion der KV-Diagramme für $n \leq 5$:

Die Konstruktion des Diagramms der Ordnung n entsteht durch Spiegelung des Diagramms der Ordnung $n - 1$. Die dabei entstehende Verdoppelung der Felder führt entweder zu einer quadratischen Anordnung oder beruht auf solcher.

Die jeweils neu hinzukommende Variable hat in den Feldern des alten Diagramms den Wert "0" und in den neuen Feldern den Wert "L". Mit geschweiften Klammern sind die Felder markiert, die zu $x_i = \mathrm{L}$ gehören. Abbildung 5.12 zeigt diese Konstuktion für $n \leq 4$.

Die KV-Diagramme entsprechen den planaren Entfaltungen der $(n+1)$-dimensionalen Hyper-Einheitswürfel der Booleschen Variablen. Deren Ecken-Koordinaten entsprechen den Binär-Nummern der Felder (s. Abb. 5.13). Zur Darstellung einer Booleschen Funktion in einem KV-Diagramm wird in den Feldern der Wert "L" eingetragen, wo die konstante Abbildung der Belegung der Variablen $y_{(\alpha)} = \mathrm{L}$ liefert. Dabei kann es zu Mehrfacheinträgen kommen.

KV-Diagramme können als spezielle Venn-Diagramme betrachtet werden (Boolescher Verband). Deshalb entsprechen die Booleschen Verknüpfungen bei L-Feldern den Mengenoperationen:

$$\cap \mathrel{\widehat{=}} \wedge, \quad \cup \mathrel{\widehat{=}} \vee, \quad {}^{-} \mathrel{\widehat{=}} {}^{-}.$$

Beispiel: S1/3 (Darstellung der 2-von-3-Mehrheitsfunktion)

$$f(x_1, x_2, x_3) = x_1x_2 \vee x_2x_3 \vee x_1x_3$$

Die Wertetafel wurde weiter vorn bereits berechnet:

x_4	x_3	x_2	x_1	α
0	0	0	0	0
0	0	0	L	1
0	0	L	0	2
0	0	L	L	3
0	L	0	0	4
0	L	0	L	5
0	L	L	0	6
0	L	L	L	7
L	0	0	0	8
L	0	0	L	9
L	0	L	0	10
L	0	L	L	11
L	L	0	0	12
L	L	0	L	13
L	L	L	0	14
L	L	L	L	15

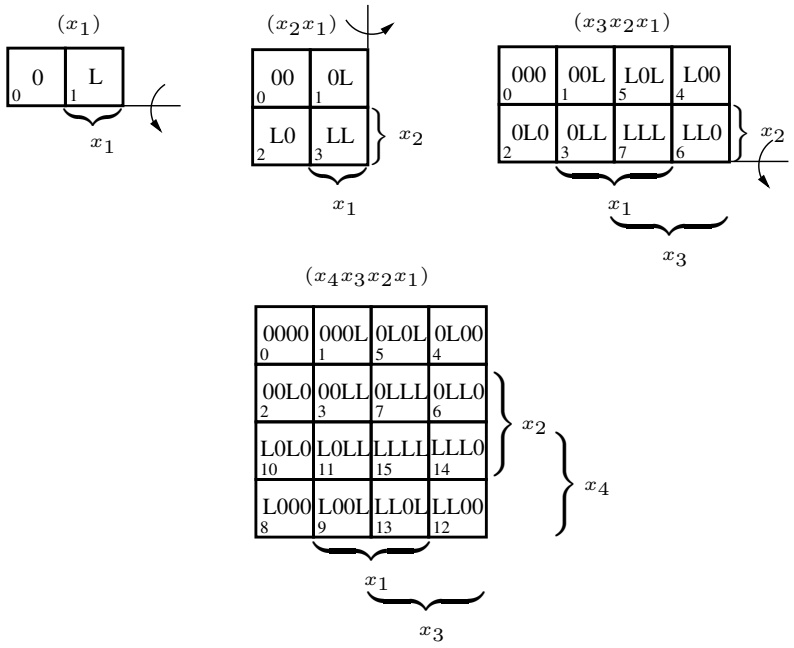
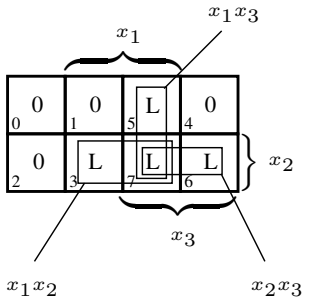


Abbildung 5.12: Konstruktion der KV-Diagramme

α	x_3	x_2	x_1	$y_{(\alpha)}$
0	0	0	0	0
1	0	0	L	0
2	0	L	0	0
3	0	L	L	L
4	L	0	0	0
5	L	0	L	L
6	L	L	0	L
7	L	L	L	L



Offensichtlich trägt die 2-von-3-Mehrheitsfunktion den Index $\gamma = 23$, kann also in der Folge mit $f_{23}(x_1, x_2, x_3)$ bezeichnet werden. Die Funktion f_{23} (x_1, x_2, x_3) liefert immer $y = \mathrm{L}$, wenn wenigstens einer der Konjunktionsterme $x_i x_j = \mathrm{L}$ liefert. Die Funktion ist deshalb als Vereinigungsmenge aller L-Felder repräsentiert.

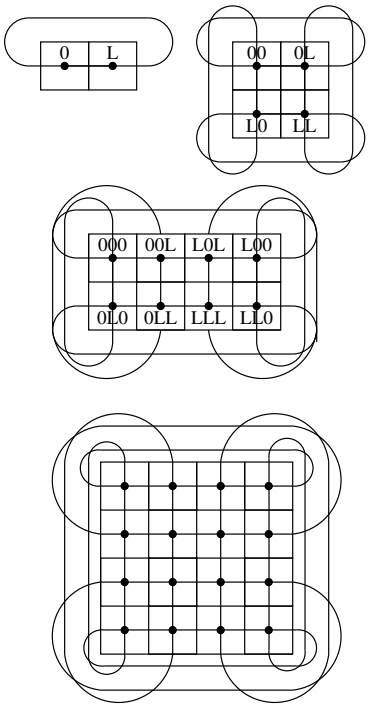
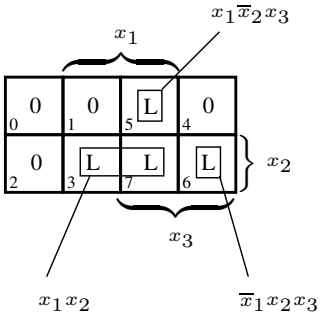


Abbildung 5.13: Torusstruktur der KV-Diagramme als planare Entfaltungen von Hyper-Einheitswürfeln

Dies legt nahe, daß es auch andere Darstellungen von $f_{23}(x_1, x_2, x_3)$ gibt, z.B.

$$f(x_1, x_2, x_3) = x_1x_2 \vee x_1\overline{x}_2x_3 \vee \overline{x}_1x_2x_3$$



5.2.3 Normalformen von Schaltfunktionen

Normalformen sind standardisierte Darstellungen von Schaltfunktionen, die gewisse Optimaleigenschaften für die Realisierung von Schaltplänen besitzen. Sie beruhen auf Konjunktionen bzw. Disjunktionen von Literalen.

5.2.3.1 Minterme

Ein *Konjunktionsterm*

$$t_j = \bigwedge_{i=1}^{m} \widetilde{x_i} \quad , \quad \widetilde{x_i} \in \{x_i, \overline{x}_i\}$$

der Länge $1 \leq m \leq n$ repräsentiert eine spezielle Boolesche Funktion. Er hat den Wert $y_j = \mathrm{L}$, wenn keines der Literale den Wert 0 hat, sonst ist $y_j = 0$.

Definition: (Minterm)

Seien x_i eine Boolesche Variable und

$$\widetilde{x_i} = x_i^{e_i} = \begin{cases} x_i & \text{für} \quad e_i = 1 \\ \overline{x}_i & \text{für} \quad e_i = 0 \end{cases}$$

ein Literal. Eine Boolesche Funktion $m_j : \mathbb{B}^n \to \mathbb{B}$,

$$m_j = \bigwedge_{i=1}^{n} x_i^{e_i}$$

heißt Minterm. Der Index j des Minterms berechnet sich aus der als Binärzahl interpretierten Folge der e_i nach

$$j = \sum_{i=1}^{n} e_i 2^{i-1}.$$

Wenn die e_i als Belegungen der Booleschen Variablen x_i interpretiert werden, entspricht offensichtlich der Index j des Minterms m_j dem Index α der konstanten Abbildung für diese Belegung. Dies folgt daraus, daß für die Indikatorfunktion eines Minterms m_j wegen der rein konjunktiven Verknüpfung gilt $g_{m_j}(y_{(0)}, \ldots, y_{(2^n-1)}) = g_{m_j}(y_{(j)})$.

Beispiel:

$$\begin{aligned} m_j &= x_1 x_2 \overline{x}_3 = \overline{x}_3 x_2 x_1 \\ j &= e_1 2^0 + e_2 2^1 + e_3 2^2 \\ &= 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 \\ &= 3 \end{aligned}$$

Für Minterme als n-stellige Boolesche Funktionen gilt die Beziehung

$$m_j \equiv f_\gamma(x_1, \ldots, x_n)$$

mit $\gamma = 2^{2^n - 1 - j}$ bzw. $j = 2^n - 1 - \text{ld}\ \gamma$. Für n Variable lassen sich maximal 2^n Minterme $m_0, \ldots, m_{2^n - 1}$ angeben.

Beispiel:

$n = 3: \quad f(x_1, x_2, x_3)$

j	m_j
0	$\overline{x}_3 \overline{x}_2 \overline{x}_1$
1	$\overline{x}_3 \overline{x}_2 x_1$
2	$\overline{x}_3 x_2 \overline{x}_1$
3	$\overline{x}_3 x_2 x_1$
4	$x_3 \overline{x}_2 \overline{x}_1$
5	$x_3 \overline{x}_2 x_1$
6	$x_3 x_2 \overline{x}_1$
7	$x_3 x_2 x_1$

Orthogonalität der Minterme:

Die Konjunktion zweier Minterme m_j und m_k ergibt

$$m_k m_j = \begin{cases} m_k & \text{für} \quad k = j \\ 0 & \text{für} \quad k \neq j \end{cases}$$

Beweis. a) $k = j$: folgt unmittelbar aus dem Idempotenzgesetz

b) $k \neq j$: zu wenigstens einem x_i in m_k kann ein $\overline{x_i}$ in m_j gefunden werden, so daß der Konjunktionsterm $t = m_k m_j$ wegen $x_i \overline{x}_i = 0$ den Wert 0 annimmt.

□

Minterme sind nicht nur spezielle n-stellige Boolesche Funktionen, vielmehr bilden sie eine (orthogonale) Basis aller n-stelligen Booleschen Funktionen. n-stellige Boolesche Funktionen und n-stellige Minterme stehen in einer Implikationsrelation zueinander.

Ein n-stelliger Term ist dann Minterm einer n-stelligen Booleschen Funktion, wenn er diese impliziert. Sein Index j heißt dann *einschlägiger Index*.

$$y_{(j)} = m_j(x_1, \ldots, x_n) = \mathrm{L} \;\Rightarrow\; y = f(x_1, \ldots, x_n) = \mathrm{L}.$$

Für diejenige Belegung $\beta(x_1), \ldots \beta(x_n) : (a_1, \ldots, a_n),\ a_i \in \mathbb{B}$, für die

$$y_{(j)} = m_j(a_1, \ldots, a_n) = \mathrm{L}$$

ist, ist auch

$$y = f(a_1, \ldots, a_n) = \mathrm{L}.$$

Minterme m_j einer n-stelligen Booleschen Funktion stehen also für die konstanten Abbildungen der Argumente dieser Funktion, die in konjunktiver Verknüpfung den Wert $y_{(j)} = \mathrm{L}$ annehmen. Die Indizes j und die Indizes α der konstanten Abbildungen $y_{(\alpha)} = \mathrm{L}$ einer Booleschen Funktion $f(x_1, \ldots, x_n)$ sind also identisch.

Beispiel: S1/4

2-von-3-Mehrheitsfunktion $\quad f_{23}(x_1, x_2, x_3) = x_1x_2 \vee x_2x_3 \vee x_1x_3$

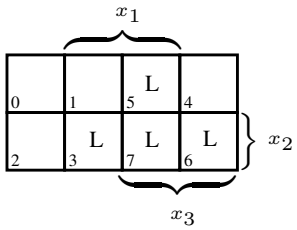
α	x_3	x_2	x_1	y	
0	0	0	0	0	
1	0	0	L	0	
2	0	L	0	0	
3	0	L	L	L	$\rightarrow m_3 = x_1x_2\overline{x}_3$
4	L	0	0	0	
5	L	0	L	L	$\rightarrow m_5 = x_1\overline{x}_2x_3$
6	L	L	0	L	$\rightarrow m_6 = \overline{x}_1x_2x_3$
7	L	L	L	L	$\rightarrow m_7 = x_1x_2x_3$

Damit kann jede n-stellige Boolesche Funktion als die disjunktive Verknüpfung derjenigen Minterme dargestellt werden, deren Wert L ist. Dies wird systematisch später behandelt.

Beispiel: S1/5

2-von-3-Mehrheitsfunktion

$$\begin{aligned} f_{23}(x_1, x_2, x_3) &= m_3 \vee m_5 \vee m_6 \vee m_7 \\ &= x_1 x_2 \overline{x}_3 \vee x_1 \overline{x}_2 x_3 \vee \overline{x}_1 x_2 x_3 \vee x_1 x_2 x_3 \end{aligned}$$



Die Minterme einer Booleschen Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}$ entsprechen umkehrbar eindeutig

- denjenigen Zeilen der Wahrheitstabellen, die den Funktionswert L haben,
- denjenigen Feldern der KV-Tafeln, die den Funktionswert L tragen.

Verschmelzung von Mintermen:

Zwei Minterme m_j und m_k, die sich nur in einem Literal $\widetilde{x_i}$ unterscheiden, lassen sich wegen der Absorptionsregel

$$m_j \vee m_k = t x_i \vee t \overline{x}_i = t(x_i \vee \overline{x}_i) = t$$

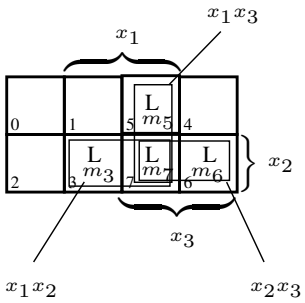
zu einem um ein Literal kürzeren Konjunktionsterm t verschmelzen.

Dabei sind auch Überlappungen der neuen Terme einer Booleschen Funktion zugelassen.

Beispiel: S1/6

2-von-3-Mehrheitsfunktion

$$\begin{aligned} f_{23}(x_1, x_2, x_3) &= m_3 \vee m_5 \vee m_6 \vee m_7 \\ &= x_1 x_2 \overline{x}_3 \vee x_1 \overline{x}_2 x_3 \vee \overline{x}_1 x_2 x_3 \vee x_1 x_2 x_3 \end{aligned}$$



$$\begin{aligned} m_3 \vee m_7 &= x_1 x_2 \overline{x}_3 \vee x_1 x_2 x_3 &= x_1 x_2 \\ m_6 \vee m_7 &= \overline{x}_1 x_2 x_3 \vee x_1 x_2 x_3 &= x_2 x_3 \\ m_5 \vee m_7 &= x_1 \overline{x}_2 x_3 \vee x_1 x_2 x_3 &= x_1 x_3 \end{aligned}$$

Damit folgt

$$f(x_1, x_2, x_3) = x_1 x_2 \vee x_2 x_3 \vee x_1 x_3$$

Man erkennt die Implikationsrelationen, z.B.

$$m_3 \Rightarrow x_1 x_2 \Rightarrow f_{23}.$$

Hier wird das Analogon zu Mengenoperationen deutlich. Es ist zu beobachten, daß kurze Terme relativ viele und lange Terme relative wenige Minterme "enthalten".

Expansion von Konjunktionstermen:

Aus Konjunktionstermen s lassen sich nach der Expansionsregel

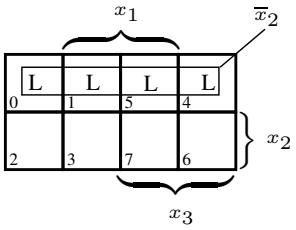
$$t = s(x_i \vee \overline{x}_i)$$

sukzessive längere Konjunktionsterme t und schließlich Minterme gewinnen.

Jedem Konjunktionsterm der Länge $m < n$ der Booleschen Funktion $f : \mathbb{B}^n \to \mathbb{B}$ entspricht im KV-Diagramm (bis $n = 4$) ein zyklisch geschlossenes Gebiet von 2^{n-m} mit dem Wert L markierten Feldern. Demzufolge kann jeder Term der Länge $m < n$ in $n - m$ Expansionsschritten in eine Disjunktion von 2^{n-m} Mintermen umgeformt werden.

Beispiel:

$t = f(x_1, x_2, x_3) = \overline{x}_2$



$$
\begin{aligned}
\overline{x}_2 &= \overline{x}_2(x_1 \vee \overline{x}_1) \quad \text{1. Entwicklung} \\
&= x_1\overline{x}_2 \vee \overline{x}_1\overline{x}_2 \\
&= x_1\overline{x}_2(x_3 \vee \overline{x}_3) \vee \overline{x}_1\overline{x}_2(x_3 \vee \overline{x}_3) \quad \text{2. Entwicklung} \\
&= x_1\overline{x}_2x_3 \vee x_1\overline{x}_2\overline{x}_3 \vee \overline{x}_1\overline{x}_2x_3 \vee \overline{x}_1\overline{x}_2\overline{x}_3 \\
&= m_5 \vee m_1 \vee m_4 \vee m_0
\end{aligned}
$$

5.2.3.2 Disjunktive Normalformen

Definition: (disjunktive Normalform (DNF))

Die disjunktive Normalform (DNF) einer Booleschen Funktion f ist eine Disjunktion von Konjunktionstermen:

$$
\begin{aligned}
f(x_1, \ldots, x_n) &= \bigvee_{j=1}^{p} t_j \qquad \text{mit} \\
t_j &= \bigwedge_{i=1}^{q_j} \widetilde{x}_{l_{ij}} \; ; \; \widetilde{x}_k \in \{x_k, \overline{x}_k\} \text{ und}
\end{aligned}
$$

$q_j, l_{ij} \in \{1, \ldots, n\}$.

Fast alle Booleschen Funktionen lassen mehrere Darstellungen als DNF zu.

Definition: (kanonische DNF (KDNF))

Eine DNF, welche nur Minterme enthält, heißt kanonische disjunktive Normalform (KDNF):

$$f(x_1, \ldots, x_n) = \bigvee_{j \in J} m_j,$$

wobei $J \subseteq \{0, 1, \ldots, 2^n - 1\}$ die Menge aller einschlägigen Indizes ist.

Darstellungssatz für Boolesche Funktionen:

Jede Boolesche Funktion hat genau eine KDNF.

Beweis. Eine Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ ist eindeutig durch die Argument-n-Tupel $(a_1, \ldots, a_n)_j$, $a_i \in \mathbb{B}$, $j \in J$ beschrieben. Für alle anderen n-Tupel $j \notin J$ nimmt sie den Wert 0 an.
Jedem n-Tupel $(a_1, \ldots, a_n)_j$, $j \in J$ entspricht eineindeutig ein Minterm m_j. Deshalb gibt es für jede Boolesche Funktion nur eine eindeutige Darstellung durch die Vereinigungsmenge ihrer Minterme. □

Die KDNF einer Booleschen Funktion läßt sich

- unmittelbar aus ihrer Wahrheitstabelle oder aus dem KV-Diagramm ablesen
- aus irgend einer DNF durch Expansion unter Beachtung
  - der Idempotenzregel bzw.
  - der Absorptionsregel

  gewinnen.

Beispiel: S1/7

2-von-3-Mehrheitsfunktion:

KDNF: $f(x_1, x_2, x_3) = m_3 \vee m_5 \vee m_6 \vee m_7$
mit $m_3 = x_1 x_2 \overline{x}_3$, $m_5 = x_1 \overline{x}_2 x_3$, $m_6 = \overline{x}_1 x_2 x_3$, $m_7 = x_1 x_2 x_3$

DNFa: $f(x_1, x_2, x_3) = x_1 x_2 \vee x_1 x_3 \vee x_2 x_3$

Die Expansion liefert

$$\begin{aligned} f &= x_1x_2(x_3 \vee \overline{x}_3) \vee x_1(x_2 \vee \overline{x}_2)x_3 \vee (x_1 \vee \overline{x}_1)x_2x_3 \\ &= x_1x_2x_3 \vee x_1x_2\overline{x}_3 \vee x_1x_2x_3 \vee x_1\overline{x}_2x_3 \vee x_1x_2x_3 \\ &\quad \vee\ \overline{x}_1x_2x_3 \quad \text{Idempotenz} \\ &= x_1x_2x_3 \vee x_1x_2\overline{x}_3 \vee x_1\overline{x}_2x_3 \vee \overline{x}_1x_2x_3 \end{aligned}$$

DNFb: $f(x_1, x_2, x_3) = x_1(x_2 \vee x_3 \vee \underline{x_2x_3}) \vee \overline{x}_1x_2x_3$
(DNFb erhält man aus DNFa durch Anwendung des Entwicklungssatzes)
Der Unterstrichene Term fällt wegen der Absorptionsregel weg:
$x_3 \vee x_2x_3 = x_3$
Entwicklung des Klammerinhalts nach x_2:
$f = x_1(x_2 \vee \overline{x}_2x_3) \vee \overline{x}_1x_2x_3 = x_1x_2 \vee x_1\overline{x}_2x_3 \vee \overline{x}_1x_2x_3$
Expansion des ersten Terms:
$f = x_1x_2x_3 \vee x_1x_2\overline{x}_3 \vee x_1\overline{x}_2x_3 \vee \overline{x}_1x_2x_3$

Aus dem Darstellungssatz für Boolesche Funktionen folgt für die Darstellung in KDNF der Satz

Darstellung als "Skalarprodukt"

Seien $f(x_1, \ldots, x_n)$ eine Boolesche Funktion und $y = (y_{(0)}, \ldots, y_{(2^n-1)})$, $y, y_{(i)} \in \mathbb{B}$, der Zeilenvektor der Werte ihrer 2^n konstanten Abbildungen für alle Belegungen ihrer Variablen $x_i \in \mathbb{B}$ sowie $m^T = (m_0, \ldots, m_{2^n-1})^T$ der Spaltenvektor aller möglicher 2^n n-stelliger Minterme, so hat f die Darstellung

$$f(x_1, \ldots, x_n) = \bigvee_{j=0}^{2^n-1} y_{(j)}m_j = y \wedge m^T.$$

Da sich Algebren ganz allgemein als Vektorräume darstellen lassen, existiert auch für die Boolesche Algebra wenigstens eine solche Darstellung. Minterme bilden ein "Basissystem" des Vektorraums n-stelliger Boolscher Funktionen in kanonischer disjunktiver Normalform (KDNF). Die Disjunktion ist die Additionsoperation in diesem Vektorraum und die Konjunktion ist die skalare Multiplikation der Basisvektoren (Minterme) mit den Werten der konstanten Abbildungen.
Dabei ist $y \wedge m^T$ das "Skalarprodukt", das hier durch Konjunktion der Vektoren erzeugt wird.

Beispiel:

$f(x_1, x_2) = x_1 \vee x_2$

j	x_2	x_1	m_j	$y_{(j)}$
0	0	0	$\overline{x}_2\overline{x}_1$	0
1	0	L	$\overline{x}_2 x_1$	L
2	L	0	$x_2\overline{x}_1$	L
3	L	L	$x_2 x_1$	L

$$f(x_1, x_2) = (0, \mathrm{L}, \mathrm{L}, \mathrm{L}) \wedge \begin{pmatrix} \overline{x}_2\overline{x}_1 \\ \overline{x}_2 x_1 \\ x_2\overline{x}_1 \\ x_2 x_1 \end{pmatrix}$$

Dies entspricht der Darstellung in KDNF:
$f(x_1, x_2) = \overline{x}_2 x_1 \vee x_2\overline{x}_1 \vee x_2 x_1$

Hieraus folgt der Satz:

Die Konjunktion und die Disjunktion zweier Boolescher Funktionen können mintermweise durchgeführt werden nach

$$f_1 \vee f_2 = \bigvee_{j=0}^{2^n-1} (y_{1,(j)} \vee y_{2,(j)}) m_j$$

$$f_1 \wedge f_2 = \bigvee_{j=0}^{2^n-1} (y_{1,(j)} \wedge y_{2,(j)}) m_j.$$

Letztendlich soll die Verbindung der KDNF zum Shannonschen Entwicklungssatz dargestellt werden.
Die Skalarprodukt-Darstellung einer Booleschen Funktion ist das Ergebnis der vollständigen konjunktiven Ausklammerung aller Variablen der Funktion mit Hilfe des Shannonschen Entwicklungssatzes nach dem Schema

$$\begin{aligned} f(x_1, \ldots, x_n) &= (x_1 \wedge f(\mathrm{L}, x_2, \ldots, x_n) \vee (\overline{x}_1 \wedge f(0, x_2, \ldots, x_n) \\ &= ((x_1 \wedge x_2 \wedge f(\mathrm{L}, \mathrm{L}, x_3, \ldots, x_n)) \vee \\ &\quad (x_1 \wedge \overline{x}_2 \wedge f(\mathrm{L}, 0, x_3, \ldots, x_n))) \vee \\ &\quad ((\overline{x}_1 \wedge x_2 \wedge f(0, \mathrm{L}, x_3, \ldots, x_n)) \vee \\ &\quad (\overline{x}_1 \wedge \overline{x}_2 \wedge f(0, 0, x_3, \ldots, x_n))) \\ &= \ldots, \end{aligned}$$

was schließlich liefert

$$f(x_1, \ldots, x_n) = \bigvee_{j=0}^{2^n-1} m_j f(a_1, \ldots, a_n) = \bigvee_{j=0}^{2^n-1} m_j y_{(j)}$$

mit den Belegungen $a_i \in \mathbb{B}$ der Booleschen Variablen x_i.

Die Mintermdarstellung einer Booleschen Funktion ist eine *disjunkte DNF* (*DDNF*). Das bedeutet, daß alle Konjunktionsterme disjunkt sind, also $t_i t_k = 0$ für $i \neq k$ liefern. Dies zeigt sich in der Orthogonalität der Minterme. Darüber hinaus liefert jeder Rekursionsschritt des Entwicklungssatzes für m-stellige Terme $t_i, m < n$, eine, wenn auch nicht kanonische, aber disjunkte disjunktive Normalform.

Beispiel:

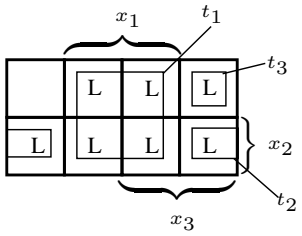
$f(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$

Die Anwendung des Entwicklungssatzes liefert

$$\begin{aligned} f = x_1 \vee \overline{x}_1(x_2 \vee \overline{x}_2 x_3) &= x_1 \vee \overline{x}_1 x_2 \vee \overline{x}_1 \overline{x}_2 x_3 \\ &= t_1 \vee t_2 \vee t_3. \end{aligned}$$

Wegen $x_i \overline{x}_i = 0$ gilt

$$t_1 t_2 = t_1 t_3 = t_2 t_3 = 0.$$



5.2.3.3 Maxterme und konjunktive Normalform

Ein *Disjunktionsterm*

$$t_j = \bigvee_{j=1}^{m} \widetilde{x}_i, \;\; \widetilde{x}_i \in \{x_i, \overline{x}_i\}$$

der Länge $1 \leq m \leq n$ repräsentiert eine spezielle Boolesche Funktion. Er hat den Wert $y_j = 0$, wenn alle Literale den Wert 0 haben, sonst gilt $y_j = \mathrm{L}$.

Definition: (Maxterm)

Seien x_i eine Boolesche Variable und

$$\widetilde{x}_i = x_i^{e_i} = \begin{cases} x_i & \text{für} \quad e_i = 1 \\ \overline{x}_i & \text{für} \quad e_i = 0 \end{cases}$$

ein Literal. Eine Boolesche Funktion $M_j : \mathbb{B}^n \to \mathbb{B}$,

$$M_j = \overline{m_j} = \bigvee_{i=1}^{n} x_i^{e_i}$$

heißt Maxterm. Der Index j eines Maxterms wird berechnet nach

$$j = 2^n - 1 - \sum_{i=1}^{n} e_i 2^{i-1} = \sum_{i=1}^{n} (1 - e_i) 2^{i-1}.$$

Ebenso gilt $m_j = \overline{M_j}$. Ein Maxterm nimmt für die Belegungen der Booleschen Variablen den Wert L an, für die ein Minterm den Wert 0 annimmt und umgekehrt. Der Index j eines Maxterms zeigt also gerade auf die konstante Abbildung eines Minterms, die den Wert Null liefert. Ein Maxterm steht also für $2^n - 1$ Minterme, deren konstante Abbildungen für diese den Wert Null liefern. Maxterme und Minterme gehen durch Negation und Anwendung der de Morganschen Regeln auseinander hervor.

Beispiel:

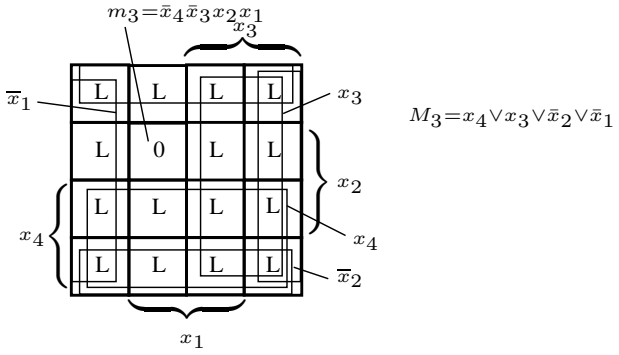
Für $f(x_1, x_2, x_3, x_4)$ gelte

$$m_3 \quad = \quad \overline{x}_4 \overline{x}_3 x_2 x_1 \quad \text{mit} \quad y_{(3)} = 0$$

Der zugehörige Maxterm ist M_3:

$$\begin{aligned} M_j &= \overline{m_3} = \overline{\overline{x}_4 \overline{x}_3 x_2 x_1} = \overline{\overline{x}_4 \overline{x}_3} \vee \overline{x_2 x_1} = x_4 \vee x_3 \vee \overline{x}_2 \vee \overline{x}_1 \\ j &= (1 - e_4) 2^3 + (1 - e_3) 2^2 + (1 - e_2) 2^1 + (1 - e_1) 2^0 \\ &= 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 3 \\ \Rightarrow M_3 &= \overline{m_3} \end{aligned}$$

KV-Darstellung von M_3:



Definition: (konjunktive Normalform (KNF))

Die konjunktive Normalform (KNF) einer Booleschen Funktion f ist eine Konjunktion von Disjunktionstermen:

$$f(x_1,\ldots,x_n) = \bigwedge_{j=1}^{p} t_j \text{ mit}$$

$$t_j = \bigvee_{i=1}^{q_j} \widetilde{x}_{l_{ij}} \; ; \; \widetilde{x}_k \in \{x_k, \overline{x}_k\} \text{ und}$$

$q_j, l_{ij} \in \{1,\ldots,n\}$.

Funktionen in KNF können in DNF überführt und aus Funktionen, die in DNF vorliegen kann ihre Darstellung in KNF gewonnen werden:

a) Überführung KNF $\rightarrow$ DNF: mittels Verallgemeinerung von

$$(A \vee B)(C \vee D) = AC \vee AD \vee BC \vee BD$$

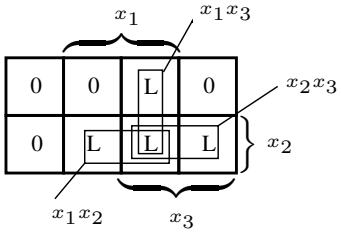
b) Überführung DNF $\rightarrow$ KNF: mittels Verallgemeinerung von

$$AB \vee CD = (A \vee C)(A \vee D)(B \vee C)(B \vee D)$$

oder: $f(x_1,\ldots,x_n) \rightarrow \overline{f}(x_1,\ldots,x_n)$ und Anwendung von de Morganschen Regeln auf $\overline{\overline{f}} = f$

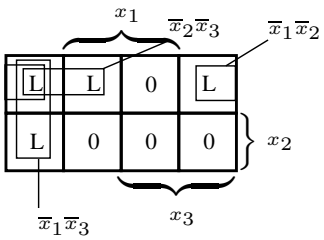
Beispiel: S1/8

2-von-3-Mehrheitsfunktion als DNF:



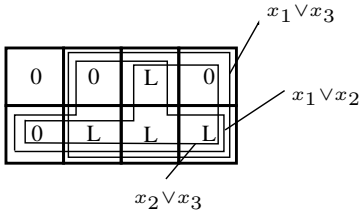
$$f(x_1, x_2, x_3) = x_1x_2 \vee x_1x_3 \vee x_2x_3$$

Die Negation liefert:



$$\overline{f}(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2 \vee \overline{x}_1\overline{x}_3 \vee \overline{x}_2\overline{x}_3$$

Erneute Negation liefert die KNF von f:



$$f = \overline{\overline{f}} = (x_1 \vee x_2)(x_1 \vee x_3)(x_2 \vee x_3)$$

Definition: (kanonische konjunktive Normalform (KKNF))

Eine KNF, die nur Maxterme enthält, heißt kanonische konjunktive Normalform (KKNF):

$$f(x_1, \ldots, x_n) = \bigwedge_{j \in \overline{J}} M_j,$$

wobei $J \subseteq \{0, 1, \ldots, 2^n - 1\}$ die Menge aller einschlägigen Indizes der Minterme $m_j = \overline{M}_j$ ist.

Maxterme sind "Basisfunktionen" der kanonischen konjunktiven Normalform (KKNF) n-stelliger Boolscher Funktionen.

Beispiel: S1/9

2-von-3-Mehrheitsfunktion

KDNF: $f = x_1x_2x_3 \vee \overline{x}_1x_2x_3 \vee x_1\overline{x}_2x_3 \vee x_1x_2\overline{x}_3$
$\overline{f} = \overline{x}_1\overline{x}_2\overline{x}_3 \vee x_1\overline{x}_2\overline{x}_3 \vee \overline{x}_1x_2\overline{x}_3 \vee \overline{x}_1\overline{x}_2x_3$
$\Rightarrow$ nach de Morgan folgt für $f = \overline{\overline{f}}$

KKNF: $f = \overline{\overline{f}} = (x_1 \vee x_2 \vee x_3)(\overline{x}_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x}_2 \vee x_3)(x_1 \vee x_2 \vee \overline{x}_3)$

Konjunktive Normalformen sind von weit geringerer Bedeutung als disjunktive Normalformen.

5.2.3.4 Primimplikanten

Während Minterme und Maxterme einer Menge von Variablen einer Booleschen Funktion nur von der Menge der Variablen abhängen und ihre Auswahl, aber nicht ihre Struktur von der Funktion abhängen, ist die Struktur von Primtermen oder Primimplikanten an die Funktion gebunden.

Jede Boolesche Funktion ist durch die Disjunktion von Primimplikanten darstellbar. Ein Primimplikant ist aber solcher nicht nur für *eine* Boolesche Funktion.

Die Implikation einer Booleschen Funktion f durch einen Minterm haben wir bereits kennengelernt.

Definition: (Implikation Boolescher Terme)

Seien t_1 und t_2 Boolesche Terme über der Variablenmenge $\{x_1, \ldots, x_n\}$ mit einer Darstellung im Basissystem $\{\wedge, \vee, \bar{}\}$. Dann impliziert der Term t_1 den Term t_2:

$$t_1 \Rightarrow t_2 \quad \Leftrightarrow \quad t_1 \wedge t_2 = t_1,$$

wenn gilt: für jede Belegung der Variablen aus $\{x_1, \ldots, x_n\}$ ist die Relation erfüllt

$$y_1 = t_1(x_1, \ldots, x_n) \equiv \mathrm{L} \quad \Rightarrow \quad y_2 = t_2(x_1, \ldots, x_n) \equiv \mathrm{L}.$$

Offensichtlich entspricht die Implikation der Ordnungsrelation "$\subseteq$" Boolescher Terme:

$$t_1 \subseteq t_2.$$

Die durch t_1 spezifizierte Funktion ist spezieller (stärker) als die durch t_2 spezifizierte oder umgekehrt, die durch t_2 repräsentierte Funktion ist allgemeiner (schwächer) als die durch t_1 repräsentierte.

Beispiel:

Gegeben seien eine 4-stellige Boolesche Funktion $f(x_1, x_2, x_3, x_4)$, sowie die Terme

$$\begin{aligned} t_1 &= m_{13} = x_1\overline{x}_2x_3x_4 \\ t_2 &= \overline{x}_2x_3x_4 \end{aligned}$$

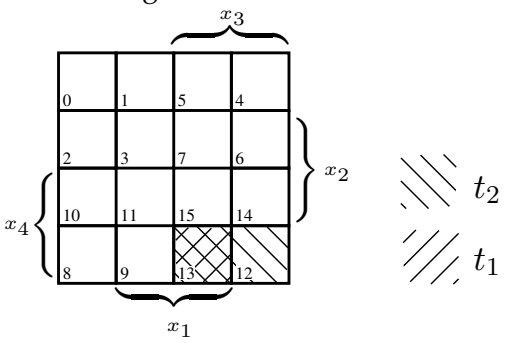
Es gilt: $t_1 = t_2 \wedge x_1$ oder $t_2 = (x_1 \vee \overline{x}_1)\overline{x}_2x_3x_4 = t_1 \vee \overline{x}_1\overline{x}_2x_3x_4 = t_1 \vee m_{12}$

$$\begin{aligned} y_1 = f(x_1, x_2, x_3, x_4) &= x_1\overline{x}_2x_3x_4 = \mathrm{L} \\ &\text{für} \quad x_1 = x_3 = x_4 = \mathrm{L},\ x_2 = 0 \\ y_2 = f(x_1, x_2, x_3, x_4) &= \overline{x}_2x_3x_4 = \mathrm{L} \\ &\text{für} \quad x_2 = 0,\ x_3 = x_4 = \mathrm{L} \end{aligned}$$

Also:

$$t_1 \wedge t_2 = t_1 \Leftrightarrow t_1 \Rightarrow t_2 \Leftrightarrow t_1 \subseteq t_2$$

Im KV-Diagramm wird diese Enthaltenseinsrelation unmittelbar erkannt.



Diese mengentheoretische Enthaltenseinsrelation Boolescher Terme kehrt sich allerdings bezüglich der oft benutzten Teiltermrelation zweier Konjunktionsterme um. Ein Konjunktionsterm s ist Teilterm eines Konjunktionstermes t, wenn t aus s durch Expansion mit $k \geq 0$ Literalen hervorgeht. Auch hier gilt bzgl. der repräsentierten Funktionen $t \subseteq s$, s umfaßt t. Gilt sogar $t \subset s$, so ist s echter Teilterm von t. Der Begriff Teilterm bezieht sich also auf die Menge der zur Repräsentation der Funktion eingesetzten Booleschen Variablen. Also ist im vorherigen Beispiel t_2 ein Teilterm von t_1.

Definition: (Implikant)

Ein Implikant I_f einer Booleschen Funktion $f(x_1, \ldots, x_n)$ ist ein Konjunktionsterm

$$I_f = \bigwedge_{i \in J \subseteq \{1,\ldots,n\}} \widetilde{x}_i,$$

für den gilt $I_f \Rightarrow f$, d.h.

$$y = I_f(x_1, \ldots, x_n) \equiv \mathrm{L} \;\Rightarrow\; y = f(x_1, \ldots, x_n) \equiv \mathrm{L}.$$

Nicht alle Teilterme von Implikanten sind wieder Implikanten der Funktion f. Hieraus folgt die Definition eines Primimplikanten.

Definition: (Primimplikant)

Ein Primimplikant $I_{\min}$ ist ein kürzester, nicht mehr um ein Literal reduzierbarer Implikant von f.

Mit anderen Worten:
Ist D_f eine DNF von $f(x_1, \ldots, x_n)$ und ist I_f ein Implikant dieser Funktion, dann ist I_f ein Primimplikant $I_{\min}$ von f, wenn

1. $I_f \Rightarrow D_f$
2. Ist $I_f \subset I'_f$, so gilt $I'_f \not\Rightarrow D_f$.

In der KV-Tafel entspricht einem Primimplikanten eindeutig ein größtes zu diesem Konjunktionsterm gehörendes (zyklisches) rechteckiges Feld, das mit der Funktion f verträglich ist, also keine Nullen enthält.

Beispiel: S2/1

Gegeben: $D_f = x_1\overline{x}_2 \vee x_1\overline{x}_4 \vee x_2\overline{x}_4 \vee \overline{x}_1x_3x_4$ ist eine DNF von $f(x_1, x_2, x_3, x_4)$
$= t_a \vee t_b \vee t_c \vee t_d$

Sind die Terme $t_a, \ldots, t_d$ Implikanten/Primimplikanten?

A) Test auf Implikanten

$D_f \wedge t_a \stackrel{!}{=} t_a$ als Voraussetzung dafür, daß $t_a = x_1\overline{x}_2$ Implikant ist.

$$\begin{aligned} D_f \wedge t_a &= x_1\overline{x}_2x_1\overline{x}_2 \vee x_1\overline{x}_4x_1\overline{x}_2 \vee x_2\overline{x}_4x_1\overline{x}_2 \vee \overline{x}_1x_3x_4x_1\overline{x}_2 \\ &= x_1\overline{x}_2 \vee x_1\overline{x}_2\overline{x}_4 \vee 0 \vee 0 \qquad \text{Absorption} \\ &= x_1\overline{x}_2(\mathrm{L} \vee \overline{x}_4) \;=\; x_1\overline{x}_2 \qquad \text{bzw. Shannon Entw.} \end{aligned}$$

Also ist t_a Implikant.
Für alle Terme $t_a, \ldots, t_d$ wird mittels Wahrheitstabelle gezeigt:

$$y(t_i) = \mathrm{L} \;\Rightarrow\; y(D_f) = \mathrm{L}$$

Die Implikation zeigt sich auch mittels der in Abschnitt 3.2.4 eingeführten Wertverlaufsinklusion: es darf zugelassen werden

$$y(t_i) = 0 \vee y(t_i) = \mathrm{L} \Rightarrow y(D_f) = \mathrm{L}$$

aber nie

$$y(t_i) = \mathrm{L} \Rightarrow y(D_f) = 0$$

α	x_4	x_3	x_2	x_1	t_a	t_b	t_c	t_d	D_f
0	0	0	0	0	0	0	0	0	0
1	0	0	0	L	L	L	0	0	L
2	0	0	L	0	0	0	L	0	L
3	0	0	L	L	0	L	L	0	L
4	0	L	0	0	0	0	0	0	0
5	0	L	0	L	L	L	0	0	L
6	0	L	L	0	0	0	L	0	L
7	0	L	L	L	0	L	L	0	L
8	L	0	0	0	0	0	0	0	0
9	L	0	0	L	L	0	0	0	L
10	L	0	L	0	0	0	0	0	0
11	L	0	L	L	0	0	0	0	0
12	L	L	0	0	0	0	0	L	L
13	L	L	0	L	L	0	0	0	L
14	L	L	L	0	0	0	0	L	L
15	L	L	L	L	0	0	0	0	0

$\rightarrow$ Alle Terme $t_a, \ldots, t_d$ sind Implikanten der Funktion.

B) Test auf Primimplikanten

- Schrittweises Abspalten von Literalen: $t_i = s_i(x_j \vee \overline{x_j})$
- Test, ob verbleibender Restterm s_i Implikant von D_f ist: existiert ein α, für das $s_i = \mathrm{L}$ und (als Widerspruch) $D_f = 0$ ist?

a) $t_a = x_1\overline{x}_2$

$s_a = x_1 \quad : \quad \alpha = 11, 15 \quad : \quad x_1 = \mathrm{L} \Rightarrow D_f = 0$
$s_a = \overline{x}_2 \quad : \quad \alpha = 0, 4, 8 \quad : \quad x_2 = 0 \Rightarrow D_f = 0$
$\longrightarrow t_a$ ist Primimplikant.

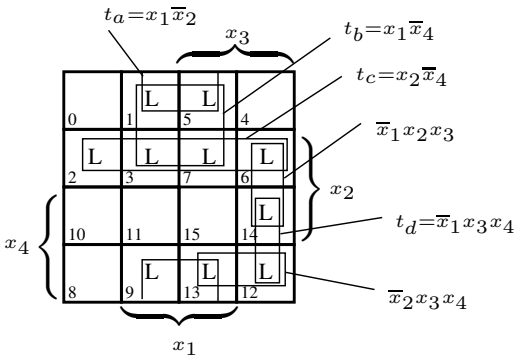
b) $t_b = x_1\overline{x}_4$
$s_b = x_1 \quad : \quad \alpha = 11, 15 \quad : \quad x_1 = \mathrm{L} \Rightarrow D_f = 0$
$s_b = \overline{x}_4 \quad : \quad \alpha = 0, 4 \quad : \quad x_4 = 0 \Rightarrow D_f = 0$
$\longrightarrow t_b$ ist Primimplikant.

c) $t_c = x_2\overline{x}_4$
$s_c = x_2 \quad : \quad \alpha = 10, 11, 15 \quad : \quad x_2 = \mathrm{L} \Rightarrow D_f = 0$
$s_c = \overline{x}_4 \quad : \quad \alpha = 0, 4 \quad : \quad x_4 = 0 \Rightarrow D_f = 0$
$\longrightarrow t_c$ ist Primimplikant.

d) $t_d = \overline{x}_1 x_3 x_4$
$s_d = \overline{x}_1 x_3 \quad : \quad \alpha = 4 \quad : \quad \overline{x}_1 x_3 = \mathrm{L} \Rightarrow D_f = 0$
$s_d = \overline{x}_1 x_4 \quad : \quad \alpha = 8, 10 \quad : \quad \overline{x}_1 x_4 = \mathrm{L} \Rightarrow D_f = 0$
$s_d = x_3 x_4 \quad : \quad \alpha = 15 \quad : \quad x_3 x_4 = \mathrm{L} \Rightarrow D_f = 0$
$\longrightarrow t_d$ ist Primimplikant.

D_f hat aber noch weitere Primimplikanten: $\overline{x}_1 x_2 x_3$, $\overline{x}_2 x_3 x_4$. Dies wird im KV-Diagramm sichtbar.

C) Primimplikanten im KV-Diagramm



Damit haben wir als Terme einer DNF kennengelernt:

- kürzeste Terme: Literale
- längste Terme: Minterme
- mittlere Terme: Primterme.

5.3 Minimierung von Schaltfunktionen

In diesem Abschnitt werden Minimalformen von Booleschen Funktionen aus beliebigen Darstellungen abgeleitet. Dabei wird häufig der Weg über eine DNF gegangen.

Definition: (Minimalform)

Sei eine Boolesche Funktion $f(x_1, \ldots, x_n)$ in Form der Disjunktion von Implikanten D_f gegeben. Dann ist eine Minimalform $D_{\min}$ gekennzeichnet durch eine minimale Anzahl von Implikanten mit

1. $D_{\min} \Leftrightarrow D_f$
2. Die Anzahl der Junktoren $\wedge, \vee$ in $D_{\min}$ ist minimal.

Mit der hier beschriebenen Minimierungsaufgabe ist keine reale Problemstellung verbunden, weil diese unbedingt die Einbeziehung von Kostenkriterien wie Gattertyp oder Randbedingungen wie zu erreichende Taktraten berücksichtigen müßten.

Außerdem sind bei VLSI-Schaltkreisen Entwurfsziele wie Regularität des Layouts oder Minimierung externer Anschlüsse oder der erforderlichen Leistungsaufnahme wesentlicher.

Der große Nutzen der Minimierung eines Schaltnetzes liegt im Austesten der Funktionalität. Ist n die Anzahl der Eingänge eines Schaltwerkes, verhält sich die Anzahl der Testmuster asymptotisch proportional $O(2^n)$!

5.3.1 Bestimmung aller Primimplikanten nach Quine-McCluskey

Die Suche nach einer DNF mit minimaler Anzahl von Konjunktionstermen und minimaler Anzahl von Literalen erfolgt in zwei Schritten:

1. Bestimmung aller Primimplikanten zu vorgegebener DNF D_f,
2. Konstruktion der minimalen DNF $D_{\min}$ aus diesen Primimplikanten.

Hintergrund für dieses zielgerichtete Vorgehen unter Nutzung von Primtermen ist der Satz:

Jede Minimalform $D_{\min}$ einer als DNF gegebenen Booleschen Funktion D_f ist die Disjunktion von Primimplikanten von D_f.

Beweis. Sei D_f eine DNF einer Funktion $f(x_1, \ldots, x_n)$. Dann ist $I_{\min}$ Primimplikant der Länge m von D_f, falls $I_{\min} \subset D_f$ und falls kein $I'_{\min}$ existiert mit $I'_{\min} \neq I_{\min}$ und $I'_{\min} \neq D_f$, so daß $I_{\min} \subset I'_{\min} \subset D_f$.
Angenommen, es gibt ein $D_{\min}$ mit $I_{\min} \subset D_{\min}$ und $I_{\min}$ sei nicht Primimplikant. Dann folgt hieraus:
Es gibt ein $I'_{\min}$ mit $I_{\min} \subset I'_{\min} \subset D_{\min}$ und $m' < m$ mit $m, m' \subset \{1, \ldots, n\}$, d.h. $I'_{\min}$ ist kürzer als $I_{\min}$ und $D_{\min}$ ist demnach nicht minimal. □

Aus dem im vorigen Abschnitt vorgestellten Beispiel zum versuchsweisen Testen auf Primimplikanten durch schrittweises Verkürzen um Literale ergibt sich folgender Algorithmus:

Algorithmus zum Primimplikanten-Test:

Sei I_f ein Implikant der Länge $m \leq n$ einer DNF D_f der Funktion $f(x_1, \ldots, x_n)$. Sei $\{i_1, \ldots, i_m\}$ die Menge der Indizes der in I_f vorkommenden Literale.

Der folgende Algorithmus liefert ein Prädikat $P(I_f)$ mit

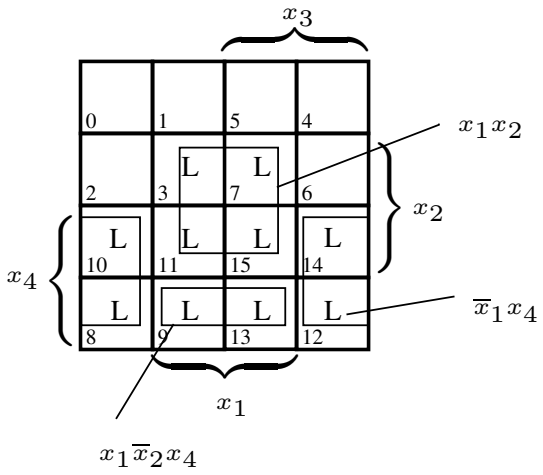
$$P(I_f) = \begin{cases} 0 & \text{falls} \quad I_f \text{ ist kein Primimplikant} \\ \mathrm{L} & \text{falls} \quad I_f \text{ ist Primimplikant} \end{cases}$$

1. Verkürze I_f um ein Literal $\widetilde{x}_k$ zu $I_f^{(k)}$, $k \in \{i_1, \ldots, i_m\}$.
   Wurde die Abspaltung aller Literale getestet, gehe zu (5).
2. Belege die restlichen Literale $\widetilde{x}_i$ von $I_f^{(k)}$ mit den Werten 0 (für $\overline{x}_i$) bzw. L (für x_i), so daß $y(I_f^{(k)}) = \mathrm{L}$.
3. Überprüfe die Implikation $I_f^{(k)} \Rightarrow f$ durch den Test $y(D_f) = \mathrm{L}$ für alle nicht nach (2) festgelegten Variablen.
4. Falls $y(D_f) = \mathrm{L}$, ist $I_f^{(k)}$ Implikant, also I_f kein Primimplikant: $P(I_f) = 0$. Gehe zu (6).
   Falls $y(D_f) = 0$, ist $I_f^{(k)}$ kein Implikant. Gehe zu (1) und fahre fort mit $j \neq k$.
5. I_f ist Primimplikant: $P(I_f) = \mathrm{L}$
6. Ende Test.

Beispiel: S3/1

$D_f = x_1x_2 \vee x_1\overline{x}_2x_4 \vee \overline{x}_1x_4$ sei eine DNF von $f(x_1, x_2, x_3, x_4)$

Entwickle aus D_f eine Darstellung $D_{\min} = \bigvee_{i=1}^{i_{\min}} I^i_{\min}$.



a) $I^a_f = x_1x_2$

$k = 2$: $I^{(2)}_f = x_1$, $y(I^{(2)}_f) = \mathrm{L}$, $D_f(\mathrm{L}, x_2, x_3, x_4) = x_2 \vee \overline{x}_2x_4$
wegen Entwicklungssatz $= x_2 \vee x_4$
$y(D_f) \neq \mathrm{L} \rightarrow I^{(2)}_f = x_1$ ist kein Implikant.

$k = 1$: $I^{(1)}_f = x_2$, $y(I^{(1)}_f) = \mathrm{L}$, $D_f(x_1, \mathrm{L}, x_3, x_4) = x_1 \vee \overline{x}_1x_4$
$= x_1 \vee x_4$
$y(D_f) \neq \mathrm{L} \rightarrow I^{(1)}_f = x_2$ ist kein Implikant.

$\longrightarrow I^a_f = x_1x_2$ ist Primimplikant.

b) $I^b_f = x_1\overline{x}_2x_4$

$k = 4$: $I^{(4)}_f = x_1\overline{x}_2$, $y(I^{(4)}_f) = \mathrm{L}$, $D_f(\mathrm{L}, 0, x_3, x_4) = x_4$
$y(D_f) \neq \mathrm{L} \rightarrow I^{(4)}_f = x_1\overline{x}_2$ ist kein Implikant.

$k = 2$: $I^{(2)}_f = x_1x_4$, $y(I^{(2)}_f) = \mathrm{L}$, $D_f(\mathrm{L}, x_2, x_3, \mathrm{L}) = x_2 \vee \overline{x}_2 = \mathrm{L}$

$$y(D_f) = \mathrm{L} \rightarrow I_f^{(2)} = x_1 x_4 \text{ ist Implikant!}$$

$\longrightarrow I_f^b = x_1 \overline{x}_2 x_4$ ist kein Primimplikant.

c) $I_f^c = \overline{x}_1 x_4$

$k = 4$: $I_f^{(4)} = \overline{x}_1, y(I_f^{(4)}) = \mathrm{L},\ D_f(0, x_2, x_3, x_4) = x_4$

$$y(D_f) \neq \mathrm{L} \rightarrow I_f^{(4)} = \overline{x}_1 \text{ ist kein Implikant.}$$

$k = 1$: $I_f^{(1)} = x_4, y(I_f^{(1)}) = \mathrm{L},\ D_f(x_1, x_2, x_3, \mathrm{L}) = x_1 x_2 \vee x_1 \overline{x}_2 \vee \overline{x}_1$

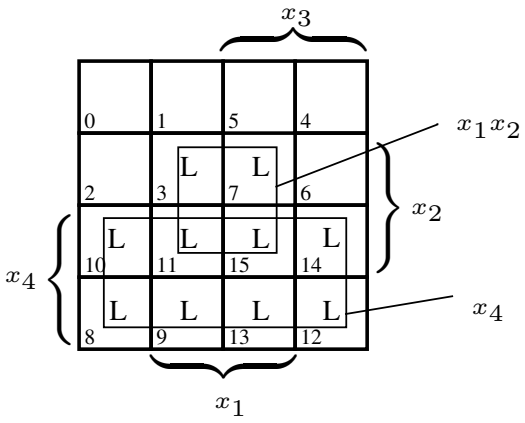
$$= x_1(x_2 \vee \overline{x}_2) \vee \overline{x}_1 = \mathrm{L}$$

$$y(D_f) = \mathrm{L} \rightarrow I_f^{(1)} = x_4 \text{ ist Implikant!}$$

$\longrightarrow I_f^c = \overline{x}_1 x_4$ ist kein Primimplikant.

Am KV-Diagramm stellt man fest, daß sich D_f aus den Primimplikanten $x_1 x_2$ und x_4 darstellen läßt:

$$D_{\min} = x_1 x_2 \vee x_4$$



Das iterierte Anwenden des Tests auf Primimplikanten liefert nicht zwangsläufig alle Primterme einer Funktion D_f, wenn D_f keine KDNF ist.
Liegt D_f als KDNF vor, kann das *Quine-McCluskey-Verfahren* angewendet werden.

a) **Idee von Quine**

1. Verschmelzung $m_i \vee m_j = m_{i,j}(x_k \vee \overline{x}_k) = m_{i,j}$, wenn sich m_i und m_j nur in einem Literal $\widetilde{x}_k$ unterscheiden.
   $m_{i,j}$ enthält noch $n - 1$ Literale.

2. versuchsweise Verschmelzung

$$m_{i,j} \vee m_{k,l} = m_{i,j,k,l}(x_m \vee \overline{x}_m) = m_{i,j,k,l}$$

usw.

3. Diejenigen Terme, die nicht mehr mit anderen gleicher Länge verschmolzen werden können, sind Primterme.

Dies ist eine Verallgemeinerung des Umgangs mit KV-Diagrammen auf n Variable.

b) **Weiterentwicklung von McCluskey**

1. Statt der Minterme werden die einschlägigen Argument-n-Tupel verwendet.
2. Terme gleicher Länge m werden spaltenweise angeordnet. Durch die Verschmelzung wegfallende Variable werden durch "–" gekennzeichnet.
3. Paarweise verschmelzbare Terme werden zusammenhängend notiert.

Als notwendige (aber nicht hinreichende) Bedingung für die Termverschmelzung gilt wegen

$$t_i x_k \vee t_i \overline{x}_k = t_i,$$

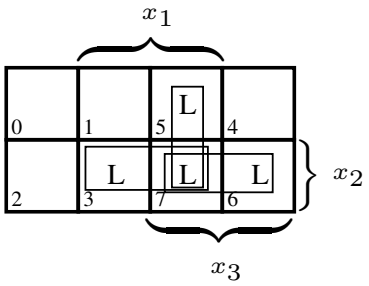
daß sich die Terme nur in einem Literal unterscheiden.

Beispiel: S1/10

Es wird gezeigt, daß die Darstellung der 2-von-3-Mehrheitsfunktion

$$f_{23}(x_1, x_2, x_3) = x_1 x_2 \vee x_1 x_3 \vee x_2 x_3$$

alle Primterme der Funktion explizit enthält.



Aus dem KV-Diagramm folgt die KDNF:

$$\begin{aligned} D_f &= m_3 \vee m_5 \vee m_6 \vee m_7 \\ &= x_1 x_2 \overline{x}_3 \vee x_1 \overline{x}_2 x_3 \vee \overline{x}_1 x_2 x_3 \vee x_1 x_2 x_3 \end{aligned}$$

	m_j				$m_{i,j}$				
j	x_3	x_2	x_1		i,j	x_3	x_2	x_1	
3	0	L	L		3,7	–	L	L	$\hat{=} x_1 x_2$
5	L	0	L		5,7	L	–	L	$\hat{=} x_1 x_3$
6	L	L	0		6,7	L	L	–	$\hat{=} x_2 x_3$
7	L	L	L						

Beispiel: S3/2

Aus dem KV-Diagramm folgt die KDNF

$$D_f^{\text{kan}} = m_3 \vee m_7 \vee m_8 \vee m_9 \vee m_{10} \vee m_{11} \vee m_{12} \vee m_{13} \vee m_{14} \vee m_{15}$$

Die Anwendung des Quine-McCluskey-Verfahrens (vgl. Abb 5.14) verläuft in folgenden Schritten:

- Ordnen der Minterme nach steigender Anzahl nicht negierter Variabler.
- Trennung der Mintermgruppen horizontal entsprechend der Anzahl nicht negierter Variabler.
- Vertikale Doppelstriche trennen die Verschmelzungsgrade der Minterme.
- Nach erfolgter Verschmelzung werden die Terme mit einem Häkchen markiert.
- Mehrfach auftretende Terme werden mit einem Kreuz markiert.
- Nicht abgehakt sind zum Schluß alle Primimplikanten.

Das Verfahren liefert folgende Primimplikanten:

$$\begin{aligned} x_1 x_2 &\hat{=} \text{– – L L} \\ x_4 &\hat{=} \text{L – – –} \end{aligned}$$

Damit erhält man: $D_{\min} = x_1 x_2 \vee x_4$.

m_i						$m_{i,j}$						$m_{i,j,k,l}$						$m_{i,j,k,\ldots}$				
i	x_4	x_3	x_2	x_1		i,j	x_4	x_3	x_2	x_1		i,j,k,l	x_4	x_3	x_2	x_1		$i,j,k,\ldots$	x_4	x_3	x_2	x_1
8	L	0	0	0	✓	8,9	L	0	0	–	✓	8,9,10,11	L	0	–	–	✓	8,9,10,11	L	–	–	–
3	0	0	L	L	✓	8,10	L	0	–	0	✓	8,9,12,13	L	–	0	–	✓	12,13,14,15				
9	L	0	0	L	✓	8,12	L	–	0	0	✓	8,10,9,11	L	0	–	–	×	8,9,12,13	L	–	–	–
10	L	0	L	0	✓	3,7	0	–	L	L	✓	8,10,12,14	L	–	–	0	✓	10,11,14,15				
12	L	L	0	0	✓	3,11	–	0	L	L	✓	8,12,9,13	L	–	0	–	×	8,10,12,14	L	–	–	–
7	0	L	L	L	✓	9,11	L	0	–	L	✓	8,12,10,14	L	–	–	0	×	9,11,13,15				
11	L	0	L	L	✓	9,13	L	–	0	L	✓	3,7,11,15	–	–	L	L						
13	L	L	0	L	✓	10,11	L	0	L	–	✓	3,11,7,15	–	–	L	L	×					
14	L	L	L	0	✓	10,14	L	–	L	0	✓	9,11,13,15	L	–	–	L	✓					
15	L	L	L	L	✓	12,13	L	L	0	–	✓	9,13,11,15	L	–	–	L	×					
						12,14	L	L	–	0	✓	10,11,14,15	L	–	L	–	✓					
						7,15	–	L	L	L	✓	10,14,11,15	L	–	L	–	×					
						11,15	L	–	L	L	✓	12,13,14,15	L	L	–	–	✓					
						13,15	L	L	-	L	✓	12,14,13,15	L	L	–	–	×					
						14,15	L	L	L	–	✓											

Abbildung 5.14: Beispiel S3/2 für das Quine-McCluskey-Verfahren

5.3.2 Minimierung mittels Primimplikantentabelle

Das Quine-McCluskey-Verfahren zur Ableitung der Primimplikanten einer Booleschen Funktion f setzt deren Darstellung in einer KDNF voraus. Demzufolge existieren in der Regel viele Primimplikanten für die Funktion f.

Die zu f gesuchte minimale DNF $D_{\min}$ muß aber nicht alle Primimplikanten von f enthalten. Ausdruck der Redundanz ist beispielsweise das Überlappen der Primimplikanten der gefundenen DNF im KV-Diagramm.

In diesem Abschnitt wird ein dreistufiges Verfahren der Suche nach $D_{\min}$ beschrieben, das die *Primimplikantentabelle* (*PIT*) nutzt. Hintergrund des Verfahrens ist die Klassifizierung der Primimplikanten einer Funktion in die Gruppen (bzgl. der Darstellung im KV-Diagramm):

a) *wesentliche* Primimplikanten: sie sind nicht verzichtbar, weil nur in ihnen ein bestimmter Minterm enthalten ist.

b) *überflüssige* Primimplikanten: sie sind verzichtbar, weil ihre Minterme bereits in anderen Primimplikanten enthalten sind.

c) *wahlweise obligatorische* Primimplikanten: wenigstens einer von ihnen ist erforderlich, wenn er auch partiell Überdeckungen mit wesentlichen Primimplikanten aufweist.

Bei diesen Interpretationen wurde die Betrachtung des KV-Diagramms zugrunde gelegt.

Das Quine-McCluskey-Verfahren liefert für Beispiel S3/1 in Abschnitt 5.3.1 (Seite 346) eine relativ eindeutige minimale DNF. Hingegen liefert die Primimplikanten-Entwicklung von Beispiel S2/1 (Abschnitt 5.2.3.4, Seite 341) keine Minimalform. Deshalb wird an Beispiel S2 und am weiteren Beispiel S4 die Problematik redundanter Primterme erläutert.

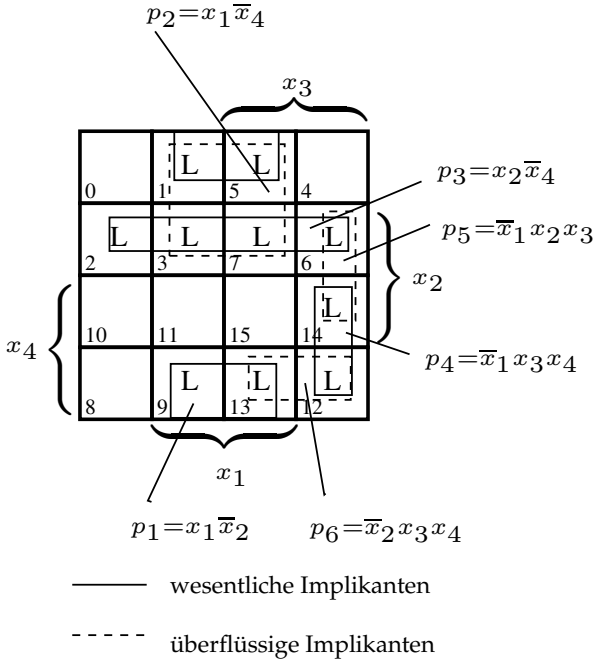
Beispiel: S2/2

$f(x_1, x_2, x_3, x_4)$ gegeben durch

$$D_f^{\text{kan}} = m_1 \vee m_2 \vee m_3 \vee m_5 \vee m_6 \vee m_7 \vee m_9 \vee m_{12} \vee m_{13} \vee m_{14}$$

m_i						$m_{i,j}$						$m_{i,j,k,l}$					
i	x_4	x_3	x_2	x_1		i,j	x_4	x_3	x_2	x_1		i,j,k,l	x_4	x_3	x_2	x_1	
1	0	0	0	L	✓	1,3	0	0	–	L	✓	1,3;5,7	0	–	–	L	
2	0	0	L	0	✓	1,5	0	–	0	L	✓	1,5;3,7	0	–	–	L	×
3	0	0	L	L	✓	1,9	–	0	0	L	✓	1,5;9,13	–	–	0	L	
5	0	L	0	L	✓	2,3	0	0	L	–	✓	1,9;5,13	–	–	0	L	×
6	0	L	L	0	✓	2,6	0	–	L	0	✓	2,3;6,7	0	–	L	–	
9	L	0	0	L	✓	3,7	0	–	L	L	✓	2,6;3,7	0	–	L	–	×
12	L	L	0	0	✓	5,7	0	L	–	L	✓						
7	0	L	L	L	✓	5,13	–	L	0	L	✓						
13	L	L	0	L	✓	6,7	0	L	L	–	✓						
14	L	L	L	0	✓	6,14	–	L	L	0							
						9,13	L	–	0	L	✓						
						12,13	L	L	0	–							
						12,14	L	L	–	0							

Hieraus folgt die Primimplikanten-DNF $D_f^{\text{prim}} = \bigvee_{i=1}^{6} p_i$ mit $p_1 = x_1\overline{x}_2$, $p_2 = x_1\overline{x}_4$, $p_3 = x_2\overline{x}_4$, $p_4 = \overline{x}_1x_3x_4$, $p_5 = \overline{x}_1x_2x_3$, $p_6 = \overline{x}_2x_3x_4$.



Offensichtlich ist $D_{\min} = p_1 \vee p_3 \vee p_4$ und $p_2,\ p_5,\ p_6$ sind überflüssig.

Algebraischer Nachweis, daß $p_2 = x_1\overline{x}_4$ ein überflüssiger Primterm ist:

Die DNF $D = x_1\overline{x}_2 \vee x_2\overline{x}_4 \vee x_1\overline{x}_4$ repräsentiert den relevanten Bereich des KV-Diagramms. Mit der Expansionsregel gilt

$$x_1\overline{x}_4 = x_1\overline{x}_4(x_2 \vee \overline{x}_2) = x_1x_2\overline{x}_4 \vee x_1\overline{x}_2\overline{x}_4.$$

Wegen $t \vee ts = t$ (Absorption) folgt

$$\begin{aligned} D &= x_2\overline{x}_4 \vee x_1x_2\overline{x}_4 \ \vee \ x_1\overline{x}_2 \vee x_1\overline{x}_2\overline{x}_4 \\ &= x_1\overline{x}_2 \vee x_2\overline{x}_4. \end{aligned}$$

Beispiel: S4

Gegeben: $D_f^{\text{kan}}(x_1, x_2, x_3) = m_1 \vee m_2 \vee m_5 \vee m_6 \vee m_7$

Hierfür ist im KV-Diagramm sofort zu erkennen:

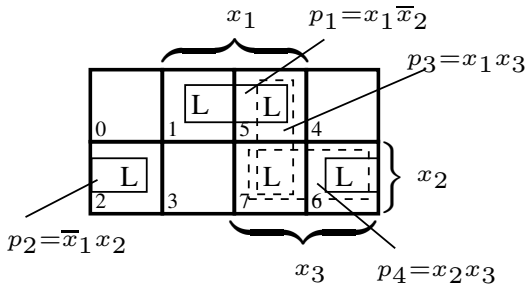
- wesentliche Primimplikanten: $p_1 = x_1\overline{x}_2,\ p_2 = \overline{x}_1 x_2$
- wahlw. obligatorische Primimplikanten: $p_3 = x_1 x_3,\ p_4 = x_2 x_4$

Also ist

$$D^1_{\min} = p_1 \vee p_2 \vee p_3$$

oder

$$D^2_{\min} = p_1 \vee p_2 \vee p_4.$$



——— wesentliche Implikanten

- - - - - wahlw. obligat. Impl.

wesentliche Primimplikanten

Seien $D^{\text{kan}}_f = \bigvee_{k=1}^{r} m_{j_k}$ eine KDNF einer Booleschen Funktion $f(x_1, \ldots, x_n)$ mit $r = |J|$, J Menge einschlägiger Indizes und $D_{\min} = \bigvee_{i=1}^{m} I^i_{\min}$ eine Minimalform von f. Dann sind alle wesentlichen Primimplikanten unter den $I^i_{\min}$ vertreten.

Ist $I^{i_0}_{\min}$ ein wesentlicher Primimplikant in $D_{\min}$, dann existiert in D^{kan}_f ein m_{j_0}, das für $I^{i_0}_{\min}$ einziger Implikant ist ($m_j \subseteq I^{i_0}_{\min}$).
Daraus folgt aber auch, daß $D_{\min}$ auch Primimplikanten enthalten kann, die mit anderen (wesentlichen) Primimplikanten gemeinsame Minterme besitzen. Dies macht die Primimplikantentabelle (PIT) deutlich.

Definition: (Primimplikantentabelle)

Die Primimplikantentabelle der KDNF einer Booleschen Funktion ist ein rechteckiges Schema mit m Zeilen für den Eintrag der Primimplikanten und n Spalten für den Eintrag der Minterme der Funktion. Für den Fall $m_{j_k} \subseteq I^i_{\min}$ erhält die Tabelle eine Markierung L an der Position (k, i), da $(m_{j_k} \subseteq I^i_{\min}) = \text{L}$ für (k, i).

Definition: (wesentlicher Primimplikant in PIT)

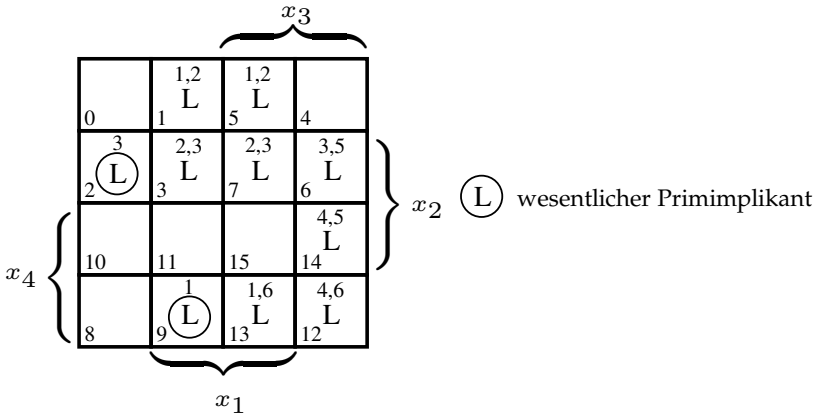
Als einziger durch m_{j_k} implizierter Primimplikant ist $I^i_{\min}$ ein wesentlicher Primimplikant, so daß es in der PIT in Spalte k nur einen Eintrag (bei Zeile i) gibt.

Beispiel: S2/3

Aus $D_f^{\text{kan}} = \bigvee_{k=1}^{10} m_{j_k}$ mit $m_{j_1} = m_1$, $m_{j_2} = m_2$, $m_{j_3} = m_3$, $m_{j_4} = m_5$, $m_{j_5} = m_6$, $m_{j_6} = m_7$, $m_{j_7} = m_9$, $m_{j_8} = m_{12}$, $m_{j_9} = m_{13}$, $m_{j_{10}} = m_{14}$ folgt

$$D_f^{\text{prim}} = \bigvee_{i=1}^{6} p_i$$

mit $p_1 = x_1\overline{x}_2$, $p_2 = x_1\overline{x}_4$, $p_3 = x_2\overline{x}_4$, $p_4 = \overline{x}_1x_3x_4$, $p_5 = \overline{x}_1x_2x_3$, $p_6 = \overline{x}_2x_3x_4$.
Im KV-Diagramm werden die Minterme durch Primterme belegt:



Hier sind p_1 und p_3 wesentliche Primimplikanten. Daraus folgt konstruktiv, daß p_4 obligatorischer Primimplikant sein muß und daß p_2, p_5 und p_6 überflüssig sind.

Die zugehörige PIT hat folgende Gestalt:

p_i \ m_{j_k}	1	2	3	5	6	7	9	12	13	14	wes. PI
1	L			L			[L]		L		×
2	L		L	L		L					
3		[L]	L		L	L					×
4								L		L	
5					L					L	
6								L	L		

Wesentliche Primimplikanten sind offensichtlich p_1 und p_3, da in den Spalten der Minterme 2 und 9 nur jeweils eine Markierung erfolgte. Man erkennt außerdem die Minterm-Überschneidungen überflüssiger mit wesentlichen Primimplikanten.

Minimierung Schritt 1:

1. Markiere alle wesentlichen Primimplikanten (die als einzige durch einen Minterm m_{j_k} impliziert werden).
2. Streiche in der PIT die Zeilen der wesentlichen Primimplikanten und alle Spalten, die durch sie markiert werden.

Daraus entsteht eine *verdichtete* PIT, die ein algebraisch schwächers, aber kleineres Ersatzproblem f' repräsentiert.
Begründung der Korrektheit: Offensichtlich gilt

$$p_i = \bigvee_{(k,i)=\mathrm{L}} m_{j_k}^{(i)}$$

(siehe auch KV-Diagramm).
Außerdem gilt, daß die Disjunktion aller Primterme der Disjunktion aller einschlägigen Minterme entspricht, also

$$D_f^{\mathrm{prim}} \Leftrightarrow D_f^{\mathrm{kan}}$$

oder

$$\bigvee_{i=1}^{m} p_i \Leftrightarrow \bigvee_{k=1}^{r} m_{j_k} \Leftrightarrow \bigvee_{i=1}^{m} \bigvee_{(k,i)=\mathrm{L}} m_{j_k}^{(i)}.$$

Die Streichung aller wesentlichen Primterme, einschließlich ihrer einschlägigen Minterme, ändert an dieser Äquivalenz (bezüglich eines f') nichts.

Ist die PIT leer, so ist eine Minimalform gefunden, die aus den wesentlichen Primimplikanten besteht.

Andernfalls ist in den Minimierungschritten 2 und 3 die Tabelle zu reduzieren.

Die verdichtete PIT repräsentiert ein kleineres, aber schwächer definiertes Ersatzproblem f', da durch das Streichen von Mintermen (wesentlicher Primimplikanten) auch die restlichen Primimplikanten transformiert werden. Es besteht die Implikation

$$p_i \Rightarrow p_i' = \bigvee_{\substack{(k,i)=\mathrm{L} \\ m_{j_k}^{(i)} \not\subseteq p_i^{\mathrm{wes.}}}} m_{j_k}^{(i)}$$

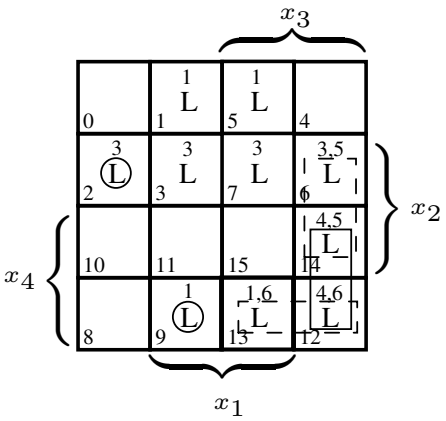
Beispiel: S2/4

Der erste Minimierungsschritt liefert folgende verdichtete PIT:

$p_i' \backslash m_{j_k}$	12	14
2		
4	L	L
5		L
6	L	

Es sind noch nicht alle obligatorischen Primimplikanten gefunden. Der Term p_2 ist überflüssig, da er nicht mehr in der Menge der einschlägigen Minterme repräsentiert ist.

Das KV-Diagramm nach Schritt 1 hat folgende Gestalt:



Die verdichtete PIT stellt das zu lösende Restproblem dar:

weitere wahlweise obligatorische Primimplikanten:

a) $p'_4 = m_{14} \vee m_{12} \;\rightarrow\; p_4 = \overline{x}_1 x_3 x_4$

oder

b) $(p'_5 = m_{14}) \vee (p'_6 = m_{12}) \;\rightarrow\; (p_5 = \overline{x}_1 x_2 x_3) \vee (p_6 = \overline{x}_2 x_3 x_4)$

Minimierung Schritt 2:

1. Streiche alle Spalten, deren Minterme in allen Primimplikanten vorkommen.
2. Streiche alle Zeilen mit identischen Markierungen bis auf jeweils eine (die zu Primimplikanten kürzester Länge gehört).

Begründung der Korrektheit:

zu 1: Der zugehörige Minterm trägt nicht zur Entscheidung bei.

Wegen $D_f^{\text{prim}} \Leftrightarrow D_f^{\text{kan}}$

und $\quad D_f^{\text{kan}} = \bigvee_{k=1}^{r-1} m_{j_k} \vee m_{j_r} = D_{f'}^{\text{kan}} \vee m_{j_r}$,

m_{j_r} sei in allen $I^i_{\min}(f)$ "enthalten": $m_{j_r} \subseteq I^i_{\min}(f),\ i = 1, \ldots, m'$

folgt $\quad D_{f'}^{\text{prim}} \Leftrightarrow D_{f'}^{\text{kan}}$.

f' steht für das "schwächere Ersatzproblem" mit $f \Rightarrow f'$, da $y(f) = \text{L} \Rightarrow y(f') = \text{L}$.

Ist also $D_{\min}(f')$ eine Minimierung für f, so ist auch es auch eine Minimierung für f'.

zu 2: Angenommen, p_{i_1} und p_{i_2} würden den gleichen Minterm m_{j_k} enthalten. Dann genügt in $D_{\min}$ die Repräsentation von m_{j_k} durch den kürzesten der beiden.

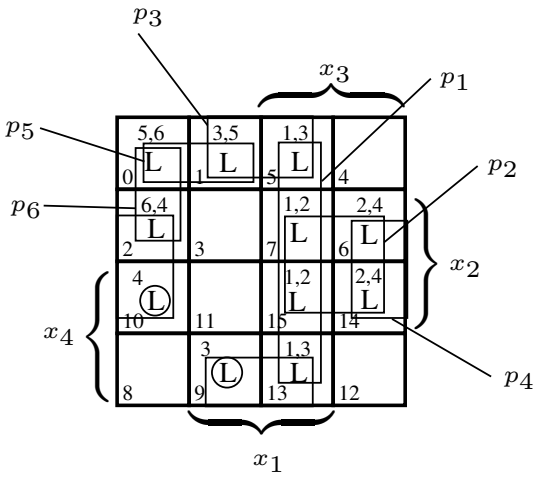
Beispiel S2 führt nicht auf diesen Schritt 2, sondern unmittelbar zu Schritt 3.

Beispiel: S5/1

$D_f^{\text{kan}} = m_0 \vee m_1 \vee m_2 \vee m_5 \vee m_6 \vee m_7 \vee m_9 \vee m_{10} \vee m_{13} \vee m_{14} \vee m_{15}$.

$p_1 = x_1 x_3$, $p_2 = x_2 x_3$, $p_3 = x_1 \overline{x}_2$, $p_4 = \overline{x}_1 x_2$, $p_5 = \overline{x}_2 \overline{x}_3 \overline{x}_4$, $p_6 = \overline{x}_1 \overline{x}_3 \overline{x}_4$

KV-Diagramm vor Schritt 1:

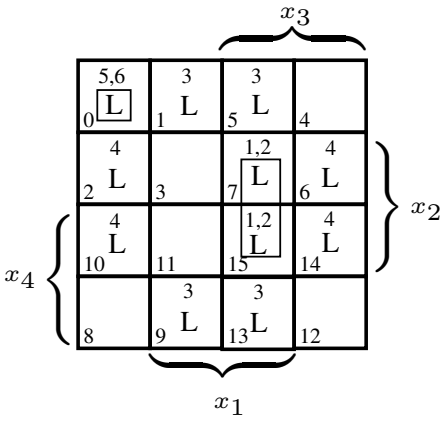


PIT vor Schritt 1:

p_i \ m_{j_k}	0	1	2	5	6	7	9	10	13	14	15
1				L		L			L		L
2					L	L				L	L
3		L		L			L		L		
4			L		L			L		L	
5	L	L									
6	L		L								

PIT und KV-Diagramm nach Schritt 1 / vor Schritt 2:

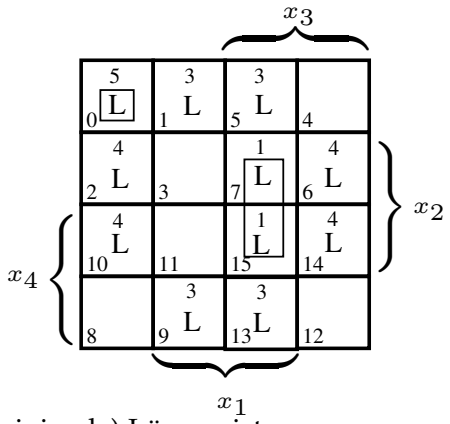
p'_i \ m_{j_k}	0	7	15
1		L	L
2		L	L
5	L		
6	L		



Die Primimplikanten p'_2 und p'_6 werden gestrichen.

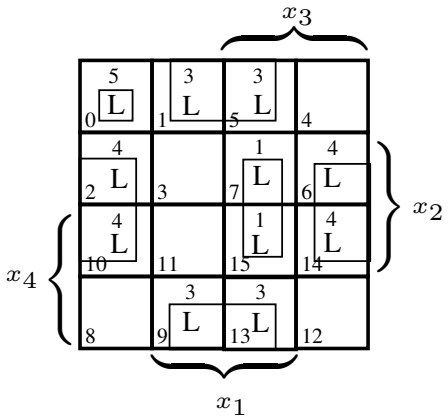
PIT und KV-Diagramm nach Schritt 2:

$p_i'' \backslash m_{j_k}$	0	7	15
1		L	L
5	L		



Diese disjunkte (aber nicht minimale) Lösung ist:

$$\begin{aligned} D^D(f) &= p_3 \vee p_4 \vee p_1'' \vee p_5'' \\ &= x_1\overline{x}_2 \vee \overline{x}_1 x_2 \vee x_1 x_2 x_3 \vee \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4 \end{aligned}$$



Minimierung Schritt 3:

Systematische Suche nach algebraischen Gesichtspunkten, die hier nicht näher spezifiziert werden soll.

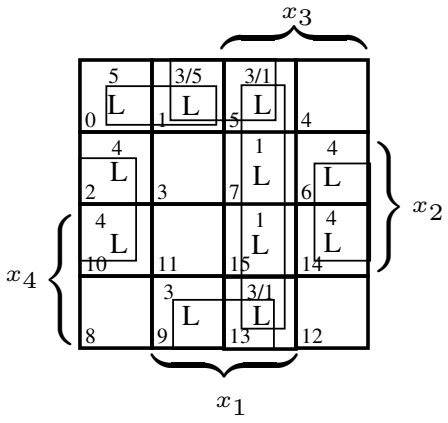
Beispiel: S5/2

Offensichtlich bestehen zwischen $p_1'' = x_1x_2x_3$ und $p_1 = x_1x_3$ bzw. zwischen $p_5'' = \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4$ und $p_5 = \overline{x}_2\overline{x}_3\overline{x}_4$ Implikationsrelationen

$$p_i'' \Rightarrow p_i.$$

Da die Lösung der Minimierungsaufgabe für f gesucht wird (und nicht für f''), wird diese Implikation für die Angabe der minimalen DNF von f genutzt, die nun aber nicht mehr disjunkt ist.

$$\begin{aligned} D_{\min}(f) &= p_3 \vee p_4 \vee p_1 \vee p_5 \\ &= x_1\overline{x}_2 \vee \overline{x}_1x_2 \vee x_1x_3 \vee \overline{x}_2\overline{x}_3\overline{x}_4 \end{aligned}$$



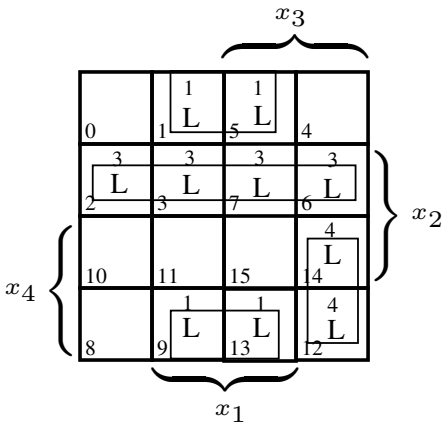
Beispiel: S2/5

Ausgehend von $p'_5 = m_{14} = \overline{x}_1 x_2 x_3 x_4$ und $p'_6 = m_{12} = \overline{x}_1 \overline{x}_2 x_3 x_4$ erhält man für

$$p'_5 \vee p'_6 = \overline{x}_1 x_3 x_4 (x_2 \vee \overline{x}_2) = \overline{x}_1 x_3 x_4 = p_4!$$

Also sind p_5 und p_6 überflüssige Primterme. Man erhält eine disjunkte minimale DNF

$$D^D_{\min}(f) = p_1 \vee p_3 \vee p_4$$



5.4 Spezielle Schaltnetze

5.4.1 Code-Wandlung und unvollständig definierte Schaltfunktionen

In Abschnitt 5.1.2 haben wir die Schaltfunktion $F : \mathbb{B}^n \to \mathbb{B}^m$ als System von m Booleschen Funktionen $f_i : \mathbb{B}^n \to \mathbb{B}$ eingeführt. Wir stellten fest, daß viele (aber nicht alle) Entwurfs- und Analyseaufgaben einer Schaltfunktion auf ihre Booleschen Funktionen reduziert werden können. Dies trifft auch auf die Minimierung zu.

Eine minimale Schaltfunktion setzt die Minimierung ihrer Booleschen Funktionen voraus, erfordert aber zusätzlich, möglichst viele Primterme p_j für möglichst viele Boolesche Funktionen f_i der Schaltfunktion F zu nutzen. Diese Optimierung gelingt am besten bei zweistufigen Schaltnetzen (Disjunktionen von Konjunktionen), also in Form von DNF.

Struktur eines optimalen zweistufigen Codewandlers:

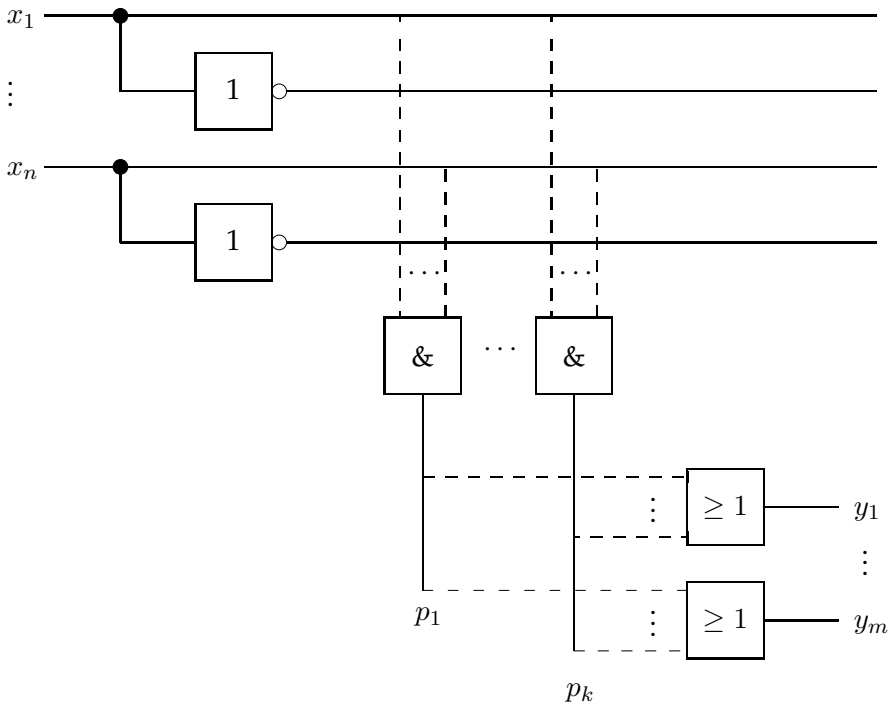


Abbildung 5.15: Zweistufiger Codewandler

Beispiel: $F : \mathbb{B}^2 \to \mathbb{B}^3$

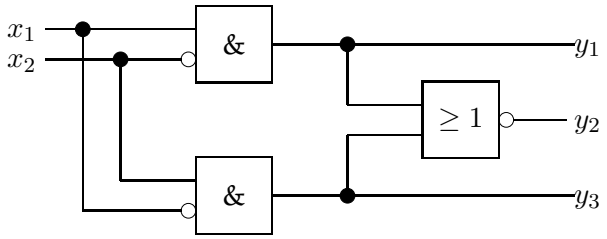
x_1	x_2	y_1	y_2	y_3
0	0	0	L	0
0	L	0	0	L
L	0	L	0	0
L	L	0	L	0

Die entsprechenden KDNFs sind:

$$\begin{aligned} y_1 &= x_1\overline{x}_2 \\ y_2 &= \overline{x}_1\overline{x}_2 \vee x_1x_2 \\ y_3 &= \overline{x}_1x_2 \end{aligned}$$

$$\begin{aligned} y_2 = x_1 \Leftrightarrow x_2 \quad &\longrightarrow \quad \overline{y}_2 = x_1 \not\Leftrightarrow x_2 = x_1\overline{x}_2 \vee \overline{x}_1x_2 = y_1 \vee y_3 \\ &\longrightarrow \quad y_2 = \overline{y_1 \vee y_3} \end{aligned}$$

Hieraus folgt das Schaltbild:



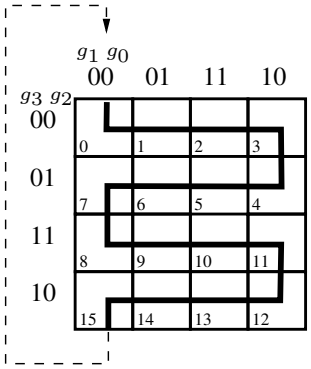
In Kapitel 2 haben wir die Codierung als Abbildung

$$C : R \to R'$$

zwischen den Repräsentationen R und R' kennengelernt. In Abschnitt 2.4 behandelten wir die in Computern üblichen Repräsentationen von Zahlen und Zeichen und die Konvertierung zwischen solchen Darstellungen.
Eine Schaltfunktion $F : \mathbb{B}^n \to \mathbb{B}^m$ kann als *Codewandler* (siehe Abb. 5.15) verstanden werden, der den Codevektor $X = (x_1, \cdots, x_n)$ in den Codevektor $Y = (y_1, \cdots, y_m)$ transferiert. Die Codevektoren können als Codewörter interpretiert werden.

Beispiel: Codewandlung Gray-Code ↔ Binär-Code

Ein *Gray-Code* zeichnet sich dadurch aus, daß der Code benachbarter Codewörter sich nur in einem Bit unterscheidet. Dies erreicht man z.B. für die Zahlen $0 \leq z < 16$ durch mäanderförmiges Abfahren der in einer $4 \times 4 -$ Matrix angeordneten 16 Zahlen (vgl. Abb. 5.16).



	Binär-Code				Gray-Code			
	d_3	d_2	d_1	d_0	g_3	g_2	g_1	g_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

Abbildung 5.16: Graycode.

Anwendung:
Analoge Längen-/Winkel-Codierung in Meßtechnik

$F : \mathbb{B}^n \to \mathbb{B}^m$ z.B. Binär-Code in Gray-Code

Erinnerung:
$z_b = \sum_{i=0}^{n_b-1} \alpha_i b^i$ ist die b-adische Zahlendarstellung im Stellenwertsystem.

Eine Dualzahl ist dann im Binärcode dargestellt, wenn sie dem Stellenwertsystem folgt. Daneben existieren also noch andere Codierungen als Dualzahl (z.B. Gray-Code), die nicht dem Stellenwertsystem entsprechen.

Einer Dezimalzahl $j = z_{10}$ entspricht eine Dualzahl z_2 im Binärcode. Seien $d^{(j)} = (d^{(j)}_{n-1} \cdots d^{(j)}_0)_2$ die Dualzahl im Binärcode einer Dezimalzahl $z^{(j)}_{10}$ und $g^{(j)} = (g^{(j)}_{n-1} \cdots g^{(j)}_0)_2$ der Gray-Code dieser Zahl ebenfalls als Dualzahl.
Mit den n Boolesche Funktionen y_i , $0 \leq i < n$, mit

$$y_i(d_{n-1}, \cdots, d_0) = \begin{cases} d_i \not\Leftrightarrow d_{i+1} & \text{für } 0 \leq i \leq n-2 \\ d_i & \text{für } i = n-1 \end{cases}$$

läßt sich die Funktionsvorschrift für den Gray-Code wie folgt darstellen:

Jeder Dezimalzahl $z^{(j)}_{10} = j_{10}$, $0 \leq j \leq 2^n - 1$, entspricht die Dualzahl $d^{(j)}$ im Binärcode als Vektor $d^{(j)} = (d^{(j)}_{n-1}, \cdots, d^{(j)}_1, d^{(j)}_0)$ der Belegung der Eingangsvariablen d_k , $0 \leq k \leq n-1$, bzw. die Dualzahl $g^{(j)}$ im Gray-Code als Vektor $g^{(j)} = (g^{(j)}_{n-1}, \cdots, g^{(j)}_1, g^{(j)}_0)$ mit

$$g^{(j)}_i = y^{(j)}_i = y_i(d^{(j)}_{n-1}, \cdots, d^{(j)}_1, d^{(j)}_0) = \begin{cases} d^{(j)}_i \not\Leftrightarrow d^{(j)}_{i+1} & \text{für } 0 \leq i \leq n-2 \\ d^{(j)}_i & \text{für } i = n-1 \end{cases}$$

z.B.: für $n = 4$

$$\begin{aligned} z^{(6)}_{10} &= 6_{10} \sim d^{(6)} = (0110)_2 \,,\, g^{(6)} = (0101)_2 \\ y^{(6)} &= (y^{(6)}_3, y^{(6)}_2, y^{(6)}_1, y^{(6)}_0) \text{ mit} \\ g^{(6)}_0 &= y^{(6)}_0(0,1,1,0) = 1 \\ g^{(6)}_1 &= y^{(6)}_1(0,1,1,0) = 0 \\ g^{(6)}_2 &= y^{(6)}_2(0,1,1,0) = 1 \\ g^{(6)}_3 &= y^{(6)}_3(0,1,1,0) = 0 \end{aligned}$$

bzw.

i	0	1	2	3
$d^{(6)}_i$	0	1	1	0
$d^{(6)}_{i+1}$	1	1	0	
$g^{(6)}_i$	1	0	1	0

Die Code-Wandlung erfolgt also für jede beliebige Dezimalzahl $0 \leq z_{10} \leq 2^n - 1$ stellenweise nach dem gleichen einfachen Prinzip der Bildung von XOR bzgl. benachbarter Bits im Binärcode, so daß ohne Minimierung ein einfacher Schaltplan folgt (siehe Abb. 5.17).

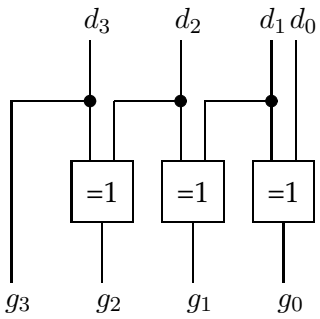


Abbildung 5.17: Codewandler Binär-Code → Gray-Code

Unvollständig definierte Funktionen enthalten konstante Abbildungen, deren Wert für die Funktion ohne Bedeutung ist. Deren Minterme heißen *"don't care-Terme"*. Enthält ein KV-Diagramm don't care-Terme, so lassen sich diese sehr gut bei der Minimierung verwenden.

Beispiel: BCD-Code (Binary Coded Decimal)

Im BCD-Code werden die einzelnen Dezimalziffern einer Zahl jeweils durch Halbbytes (4 Bits) codiert. Dabei sind zur Darstellung der Ziffern die Werte $10_{10}, \cdots, 15_{10}$ bzw. $(1010)_2, \cdots, (1111)_2$ nicht erforderlich.

Die Booleschen Funktionen $f_j(x_1, x_2, x_3, x_4)$ in der folgenden Tabelle repräsentieren die Dezimalziffern $z_{10}^{(j)}$.

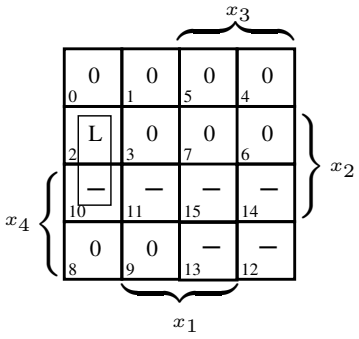
	Binär–Code				BCD–Code									
z_{10}	x_4	x_3	x_2	x_1	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
0	0	0	0	0	L	0	0	0	0	0	0	0	0	0
1	0	0	0	L	0	L	0	0	0	0	0	0	0	0
2	0	0	L	0	0	0	L	0	0	0	0	0	0	0
3	0	0	L	L	0	0	0	L	0	0	0	0	0	0
4	0	L	0	0	0	0	0	0	L	0	0	0	0	0
5	0	L	0	L	0	0	0	0	0	L	0	0	0	0
6	0	L	L	0	0	0	0	0	0	0	L	0	0	0
7	0	L	L	L	0	0	0	0	0	0	0	L	0	0
8	L	0	0	0	0	0	0	0	0	0	0	0	L	0
9	L	0	0	L	0	0	0	0	0	0	0	0	0	L
10	L	0	L	0										
⋮								don´t care						
15	L	L	L	L										

Im BCD–Code ist jede Ziffer nur durch einen Minterm repräsentiert. Die Minterme mit den Indizes $10, \ldots, 15$ sind don't care–Terme. Also gilt für die Dezimalziffer $z_{10}^{(j)}, 0 \leq j \leq 9$, die Darstellung durch die Boolesche Funktion

$$y_k = f_k(x_1, x_2, x_3, x_4) = m_k = \begin{cases} \text{L} & \text{für } k = j \\ 0 & \text{sonst} \end{cases}$$

Ein solcher Code heißt lokalisiert, im Unterschied zum Binär-Code der entsprechenden Dualzahl, der ein verteilter Code ist.

Indem die don't-care-Minterme beliebige Werte annehmen dürfen, gestattet das Quine-McCluskey-Verfahren die begrenzte Minimierung einiger Boolescher Funktionen (z.B. $j = 2, 3, 4, 5, 7$), während andere nicht minimiert werden können (z.B. $j = 0, 1$). (Achtung: Jede Ziffer j wird separat minimiert bzgl. des Mintermes m_j, da nur dieser den Wert L hat!). Abbildung 5.18 zeigt die Minimierung im KV-Diagramm für f_2.



$$\begin{aligned} f_2' &= m_2 \vee m_{10} \\ &= \overline{x}_1 x_2 \overline{x}_3 \overline{x}_4 \vee \overline{x}_1 x_2 \overline{x}_3 x_4 \\ &= \overline{x}_1 x_2 \overline{x}_3 \end{aligned}$$

Abbildung 5.18: Minimierung von f_2

Beispiel: 7-Segment-Code

Weit interessanter ist die Wandlung des Binär–Codes in den *7-Segment-Code*.

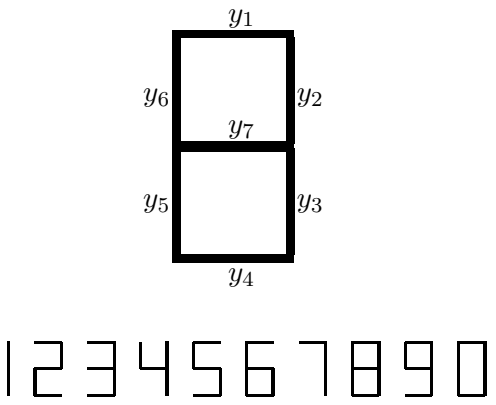


Abbildung 5.19: LCD-Anzeige

Anwendung: LCD-Anzeige der Ziffern $0 \leq z_{10} \leq 9$, z.B. auf Taschenrechnern.

Zur Darstellung einer Ziffer $z_{10}^{(j)}$ werden, wie Abbildung 5.19 zeigt, maximal 7 Liniensegmente $y_i^{(j)}$ verwendet. Die Darstellung einer Ziffer $z_{10}^{(j)}$ im 7–Segment–Code erfolgt durch die konjunktive Schaltfunktion

$$y^{(j)} = \bigwedge_{k=1}^{r_j} y_{i_k}^{(j)}$$

mit den Booleschen Funktionen

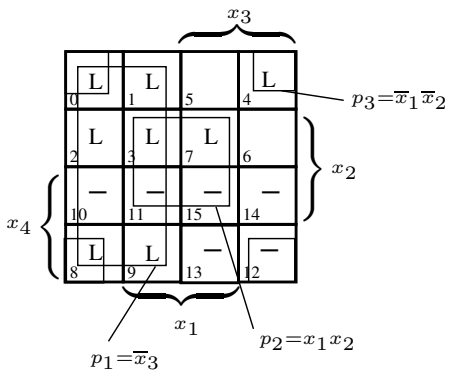
$$y_i^{(j)} = f(x_1, x_2, x_3, x_4) = \begin{cases} \text{L} & \text{wenn zur Darstellung erforderlich} \\ 0 & \text{sonst} \end{cases}$$

In horizontaler Richtung repräsentiert der Vektor $y^{(j)} = (y_1^{(j)}, \ldots, y_7^{(j)})$ die Schaltfunktion zur Darstellung der Ziffer $z_{10}^{(j)}$. Jedes $y_i^{(j)}$ stellt eine Boolesche Funktion dar, die als KDNF den Beitrag des Segments i zu allen Ziffern $z_{10}^{(j)}$, $0 \leq j \leq 9$, repräsentiert.

Da die Ziffern im Binär-Code repräsentiert werden, sind die Funktionen y_i für die Belegungen (L0L0) bis (LLLL) nicht eindeutig bestimmt. Dies kann bei der Minimierung nach Quine-McCluskey in Form von dont't care-Termen ausgenutzt werden. Dies wird am Beispiel y_2 explizit gezeigt (Abb. 5.20). Es gilt für diese Funktion $p_1 = \overline{x}_3$, $p_2 = x_1x_2$, $p_3 = \overline{x}_1\overline{x}_2$. Die don't–care–Minterme m_{13} und m_{14} sind unter den mit Sternen markierten don't–care–Mintermen nicht explizit aufgeführt, da sie nicht benötigt werden.

	Binär-Code				7-Segment-Code						
z_{10}	x_4	x_3	x_2	x_1	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	0	0	L	L	L	L	L	L	0
1	0	0	0	L	0	L	L	0	0	0	0
2	0	0	L	0	L	L	0	L	L	0	L
3	0	0	L	L	L	L	L	L	0	0	L
4	0	L	0	0	0	L	L	0	0	L	L
5	0	L	0	L	L	0	L	L	0	L	L
6	0	L	L	0	L	0	L	L	L	L	L
7	0	L	L	L	L	L	L	0	0	0	0
8	L	0	0	0	L	L	L	L	L	L	L
9	L	0	0	L	L	L	L	L	0	L	L
10	L	0	L	0							
⋮							don´t care				
15	L	L	L	L							

Das KV-Diagramm des Segments y_2 hat die Gestalt



Für die Funktionen y_i werden in Abb. 5.21 die Ergebnisse der Primimplikantenermittlung in KV-Diagrammen dargestellt. Wie leicht zu erkennen ist, sind für alle Funktionen die nach Quine-McCluskey gefundenen Primterme wesentlich, so daß sich eine Minimierung nach der PIT erübrigt.

m_i

i	x_4	x_3	x_2	x_1	
0	0	0	0	0	✓
1	0	0	0	L	✓
2	0	0	L	0	✓
4	0	L	0	0	✓
8	L	0	0	0	✓
3	0	0	L	L	✓
9	L	0	0	L	✓
*10	L	0	L	0	✓
*12	L	L	0	0	✓
7	0	L	L	L	✓
*11	L	0	L	L	✓
*15	L	L	L	L	✓

$m_{i,j}$

i,j	x_4	x_3	x_2	x_1	
0,1	0	0	0	–	✓
0,2	0	0	–	0	✓
0,4	0	–	0	0	✓
0,8	–	0	0	0	✓
1,3	0	0	–	L	✓
1,9	–	0	0	L	✓
2,3	0	0	L	–	✓
2,10	–	0	L	0	✓
4,12	–	L	0	0	✓
8,9	L	0	0	–	✓
8,10	L	0	–	0	✓
8,12	L	–	0	0	✓
3,7	0	–	L	L	✓
3,11	–	0	L	L	✓
9,11	L	0	–	L	✓
10,11	L	0	L	–	✓
7,15	–	L	L	L	✓
11,15	L	–	L	L	✓

$m_{i,j,k,l}$

i,j,k,l	x_4	x_3	x_2	x_1	
0,1;2,3	0	0	–	–	✓
0,2;1,3	0	0	–	–	×
0,4;8,12	–	–	0	0	
0,8;4,12	–	–	0	0	×
0,8;1,9	–	0	0	–	✓
0,8;2,10	–	0	–	0	✓
1,3;9,11	–	0	–	L	✓
1,9;3,11	–	0	–	L	×
2,3;10,11	–	0	L	–	✓
2,10;3,11	–	0	L	–	×
8,9;10,11	L	0	–	–	✓
8,10;9,11	L	0	–	–	×
3,7;11,15	–	–	L	L	
3,11;7,15	–	–	L	L	×

$p_2 = x_1x_2$

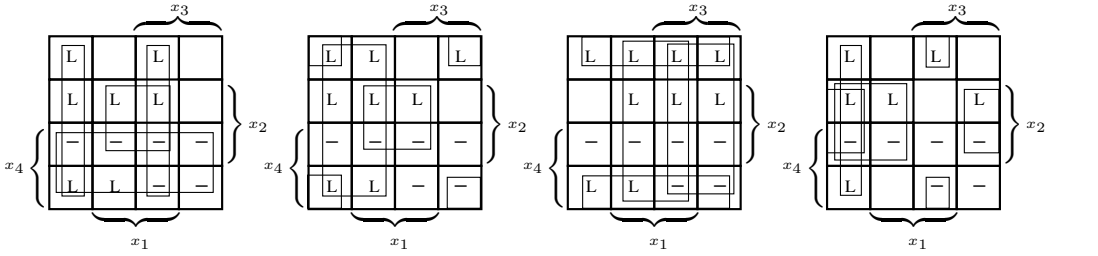
$p_3 = \overline{x}_1\overline{x}_2$

$m_{i,j,k,\ldots}$

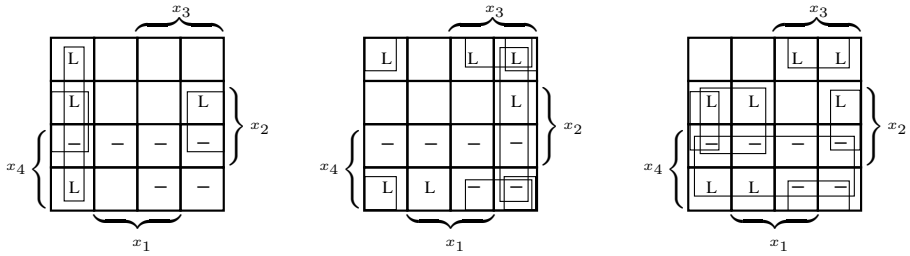
$i,j,k,\ldots$	x_4x_3	x_2	x_1	
0,1;2,3; 8,9;10,11	0	–	–	
0,8;1,9; 2,3;10,11	0	–	–	×
0,8;2,10; 1,3;9,11	0	–	–	×

$p_1 = \overline{x}_3$

Abbildung 5.20: Primimplikanten von y_2



$y_1 = x_4 \vee \overline{x}_1\overline{x}_3 \vee x_1x_3 \vee x_1x_2$ $y_2 = \overline{x}_3 \vee x_1x_2 \vee \overline{x}_1\overline{x}_2$ $y_3 = \overline{x}_3 \vee x_1x_2 \vee x_1\overline{x}_2$ $y_4 = \overline{x}_1x_2 \vee \overline{x}_1\overline{x}_3 \vee x_2\overline{x}_3 \vee x_1\overline{x}_2x_3$



$y_5 = \overline{x}_1x_2 \vee \overline{x}_1\overline{x}_3$ $y_6 = x_4 \vee \overline{x}_1\overline{x}_2 \vee \overline{x}_1x_3 \vee \overline{x}_2x_3$ $y_7 = \overline{x}_1x_2 \vee \overline{x}_2x_3 \vee x_2\overline{x}_3 \vee x_4$

Abbildung 5.21: Primimplikanten der Funktionen y_1 bis y_7

Zusammenfassend hier die minimalen DNF der Segmente-Funktionen:

$$
\begin{aligned}
y_1 &= x_4 \vee \overline{x}_1\overline{x}_3 \vee x_1x_3 \vee x_1x_2 \\
y_2 &= \overline{x}_3 \vee x_1x_2 \vee \overline{x}_1\overline{x}_2 \\
y_3 &= x_1 \vee \overline{x}_2 \vee x_3 \\
y_4 &= \overline{x}_1x_2 \vee \overline{x}_1\overline{x}_3 \vee x_2\overline{x}_3 \vee x_1\overline{x}_2x_3 \\
y_5 &= \overline{x}_1x_2 \vee \overline{x}_1\overline{x}_3 \\
y_6 &= x_4 \vee \overline{x}_1\overline{x}_2 \vee \overline{x}_1x_3 \vee \overline{x}_2x_3 \\
y_7 &= \overline{x}_1x_2 \vee \overline{x}_2x_3 \vee x_2\overline{x}_3 \vee x_4
\end{aligned}
$$

Abbildung 5.22 zeigt den Schaltplan für die 7-Segmentanzeige.

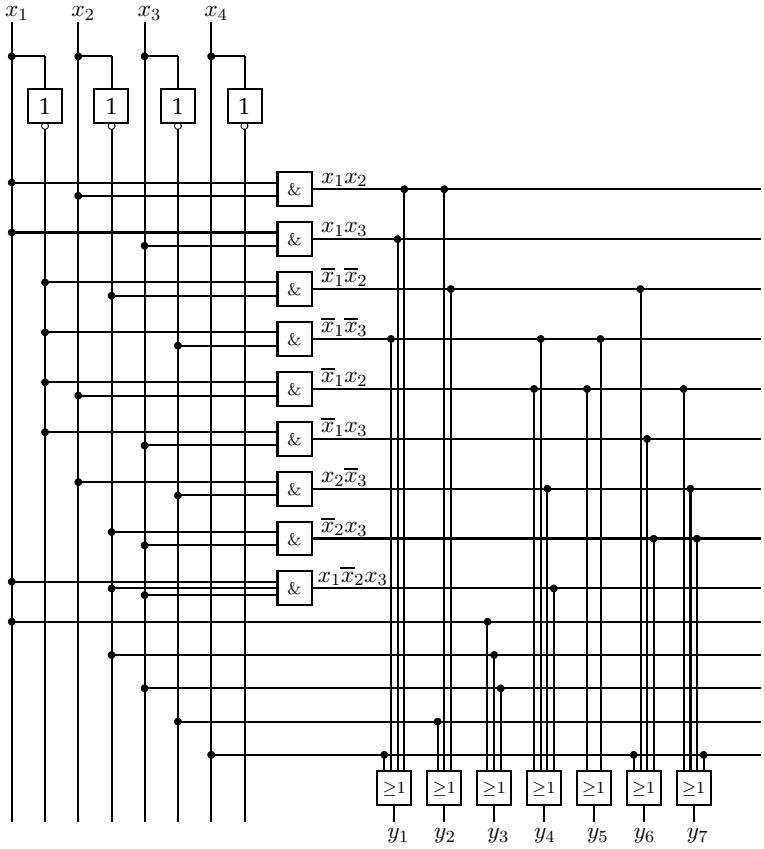


Abbildung 5.22: Schaltplan (7-Segment-Code)

Eine weitere Minimierung des Schaltfunktion $y^{(j)}$ wäre dadurch möglich, daß die Liniensegmente y_i zu Gruppen $y_{r\ldots s\ldots t} = y_r \cdots y_s \cdots y_t$ zusammengefaßt werden, daß also Wertcodierung zur Anwendung kommt. Z.B. gilt $y_{23} = y_2 y_3, y_{147} = y_1 y_4 y_7$. Also könnte beispielsweise folgende Bildungsregel der Schaltfunktion $y^{(j)}$ realisiert werden:

$$y^{(1)} = y_{23}, \qquad y^{(3)} = y_{23} y_{147}, \qquad y^{(9)} = y_{23} y_{147} y_6, \text{ usw.}$$

Während die hier demonstrierte separate Minimierung der sieben Booleschen Funktionen nur ein suboptimales Ergebnis darstellt, ist die globale Minimierung der Schaltfunktion zwar mit derartigen Heuristiken approximierbar, aber nur mit einigem algebraischem Aufwand auch erreichbar.

5.4.2 Schaltnetze für Auswahl, Verzweigung und Vergleich

Datenweichen heißen

a) für Eingangsdaten: Multiplexer oder *Auswahlschaltung*

b) für Ausgangsdaten: Demultiplexer oder *Verzweigungsschaltung*.

Schaltnetze für den Vergleich von Daten heißen *Komparatoren* .

5.4.2.1 Multiplexer

Definition: (Multiplexer)

Ein Multiplexer ist ein selektierendes Schaltnetz

$$y = F(x_0, \cdots, x_{n-1}, c_0, \cdots, c_{m-1}),$$

das aus n Eingangsvariablen x_i mittels $m = \mathrm{ld}\ n$ Steuervariablen c_j eine Ausgangsvariable $y = x_k$ mit $k = (c_{m-1} \cdots c_0)_2$ ermittelt.

Offensichtlich treten in einem derartigen Schaltnetz Eingangsvariable und Steuervariable als Boolesche Variable unterschiedlicher Priorität auf. Die Steuervariablen bestimmen die Logik des Multiplexers.

Beispiel: 2-MUX (4:1-Multiplexer)

Ein Multiplexer mit zwei Steuervariablen c_0, c_1 kann vier Eingangsvariable auf einen Ausgang schalten, denn $n = 2^m$.

i	c_1	c_0	y
0	0	0	x_0
1	0	L	x_1
2	L	0	x_2
3	L	L	x_3

Abbildung 5.23 zeigt das Schaltnetz für einen 2-MUX. Für die Ausgangsvariable y gilt

$$y = x_0\overline{c}_0\overline{c}_1 \vee x_1c_0\overline{c}_1 \vee x_2\overline{c}_0c_1 \vee x_3c_0c_1$$

Die Abbildung $y = f(c_0, c_1; x_0, x_1, x_2, x_3)$ stellt eine Boolesche Funktion für die zwei Steuervariablen dar. Die Eingabevariablen sind von nachgeordneter Priorität, d.h. ihre Belegung ist unerheblich für die Logik des Multiplexers.

Der Index i für die durchzustellende Eingangsvariable ergibt sich aus der als Binärzahl interpretierten Belegung der Steuervariablen. (Abb. 5.23)

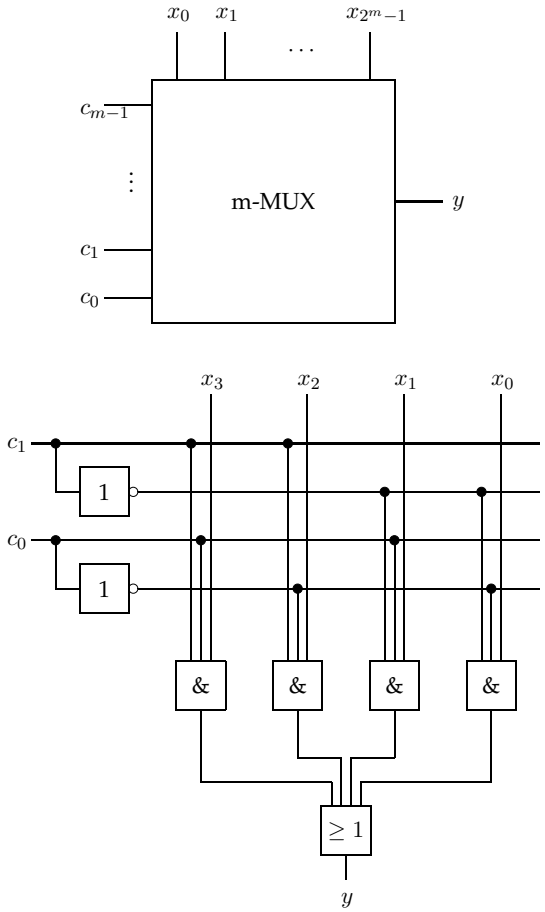


Abbildung 5.23: m-MUX-Gatter und 2-MUX Schaltnetz

Jeder Multiplexer läßt sich als 3-stufige Schaltung realisieren:

1. Stufe: Negation der Steuersignale

2. Stufe: Verbindung von Steuer- mit Eingangssignalen (z.B. UND-Gatter)

3. Stufe: Berechnung der Ausgangsvariablen (z.B. ODER-Gatter)

Auch andere Gatter als UND sind als Steuerelement in Stufe 2 verwendbar:

Seien x, y, c die Vektoren der Eingangsdaten, der Ausgangsdaten und der Vektor der Steuervariablen.

	Gatter-Typ	Wert der Steuervariablen c	Wirkung bzgl. y
1.	UND	0 bzw. L	0 bzw. x
2.	ODER	0 bzw. L	x bzw. L
3.	XOR	0 bzw. L	x bzw. $\overline{x}$

zu 1.: UND $\quad y = xc = \begin{cases} 0 & \text{für } c = 0 \\ x & \text{für } c = L \end{cases}$

zu 2.: ODER $\quad y = x \vee c = \begin{cases} x & \text{für } c = 0 \\ L & \text{für } c = L \end{cases}$

zu 3.: XOR $\quad y = x \not\Leftrightarrow c = \begin{cases} X & \text{für } c = 0 \\ \overline{x} & \text{für } c = L \end{cases}$

x	c	y
0	0	$0 = x$
0	L	$L = \overline{x}$
L	0	$L = x$
L	L	$0 = \overline{x}$

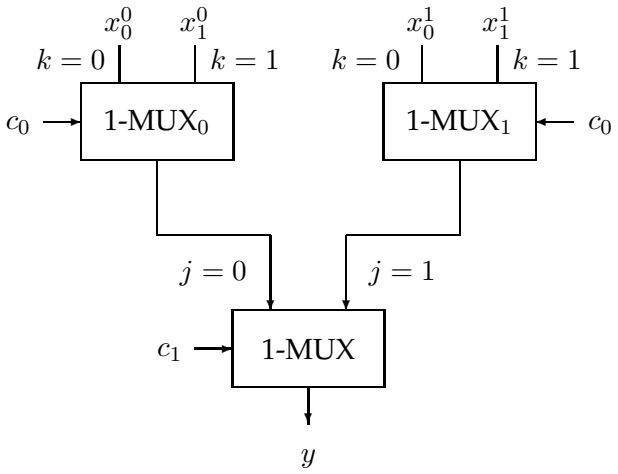
Als Nachteil der 3-stufigen MUX-Architektur ergibt sich die hohe Leitungsdichte (Fan-In):

2. Stufe: UND-Gatter $\sim m$
3. Stufe: ODER-Gatter $\sim 2^m = n$

Ausweg: Kaskade von m-MUX. Ein $2m$-MUX entspricht $2^m + 1$ Kopien von m-MUX.

Dieser Multiplex-Kaskadierung liegt die Idee zugrunde, daß der Index i des durchzuschaltenden Eingabedatums $x_i, i = (c_1 c_0)_2$, darstellbar ist durch $i = j \cdot 2^1 + k$. Dabei codiert $j = (c_1)_2$ zusammenhängende Teilmengen $x^j = (x_0^j, x_1^j)$ der Eingabedaten und $k = (c_0)_2$ stellt den Offset des Datums x_k^j im Segment x^j dar. Also decodiert die erste Schicht den Offset und die zweite Schicht die Basis (den Segmentindex). Dies hat den Vorteil, daß die Multiplexer beider Schichten gleich gestaltet sind.

Beispiel: 2-MUX, konstruiert aus drei 1-MUX



Beispiel:

Es sei zu realisieren $y = x_2$. Allgemein gilt für das Durchstellen von x_i: $i_{10} = (c_1 c_0)_2$. Also gilt für $i = 2 : 2_{10} = (\mathrm{L0})_2$. Wegen $i = j \cdot 2^1 + k$ folgt hieraus $j = 1$ und $k = 0$.

i	j/c_1	k/c_0	$i = j \cdot 2^1 + k$
0	0/0	0/0	0=0 · 2+0
1	0/0	1/L	1=0 · 2+1
2	1/L	0/0	2=1 · 2+0
3	1/L	1/L	3=1 · 2+1

Allgemeines Schema:

Ein $2m$-MUX selektiert die Eingangsvariable x_i mit $i = (c_{2m-1} \cdots c_0)_2$. Dieser $2m$-MUX wird realisiert als Parallelschaltung von 2^m m-MUX in der ersten Stufe und einem m-MUX in der zweiten Stufe. Dabei wird der Bereich der 2^{2m} Eingangsvariablen $x_0, \cdots, x_{2^{2m}-1}$ im vorraus in 2^m Segmente der Länge 2^m unterteilt.

Diese Segmente $x^j = (x_0^j, \ldots, x_{2^m-1}^j)$ enthalten die Eingabedaten $x_i \equiv x_k^j, i = j \cdot 2^m + k$, mit dem Segmentindex $j_{10} = (c_{2m-1} \ldots c_m)_2, 0 \leq j \leq 2^m - 1$, und dem Offset $k_{10} = (c_{m-1} \ldots c_0)_2, 0 \leq k \leq 2^m - 1$. Diesem Indizierungsschema des Eingabevektors $x = x^0 \circ x^1 \circ \ldots \circ x^{2^m-1}$ entspricht eine Zerlegung des Vektors der Steuervariablen $c = (c_0, \ldots, c_{2m-1}) = (c_0, \ldots, c_{m-1}) \circ (c_m, \ldots, c_{2m-1}) \equiv c^{(k)} \circ c^{(j)}$.

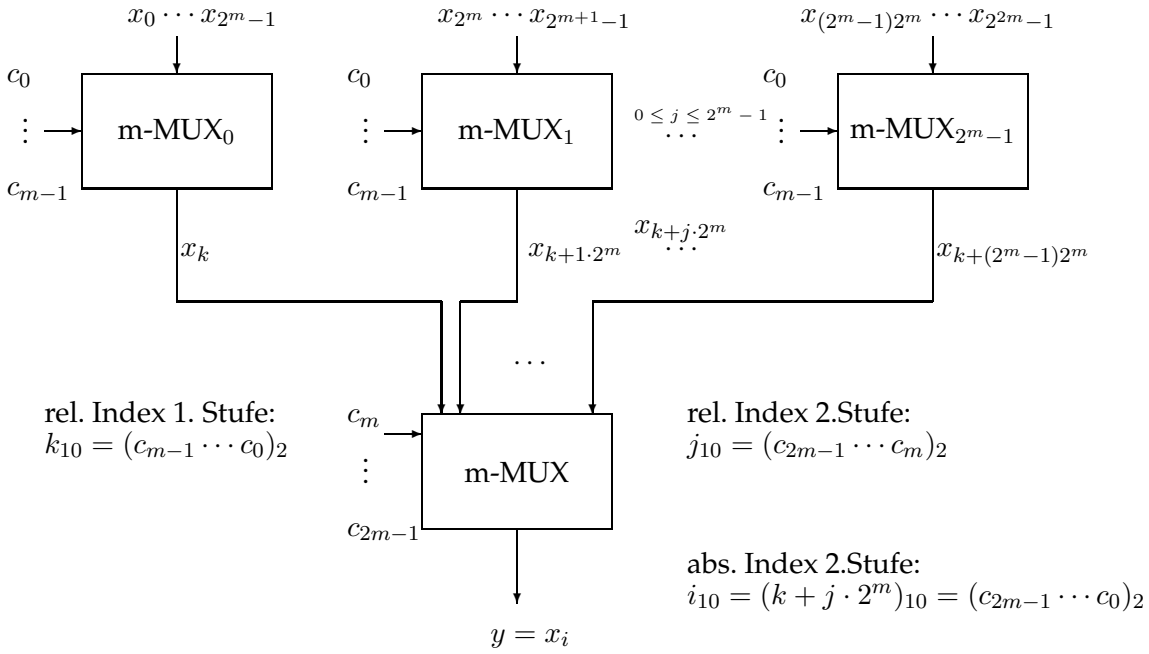


Abbildung 5.24: 2m-MUX als Kaskadierung von m-MUX

Der Steuervektor $c^{(k)}$ dient in der ersten Schicht der Decodierung des Offsets k. Damit wird, als Eingabe für das m-MUX der zweiten Schicht, aus jedem Segment x^j ein $x^j_k = x_{j2^m+k}$ ausgewählt.
Das m-MUX der zweiten Schicht erhält als Eingabe den Datenvektor $x_k, x_{k+2^m}, \cdots, x_{k+(2^m-1)2^m}$. Von diesem wird die Variable mit dem relativen Index $j = (c_{2m-1} \cdots c_m)_2$ ausgewählt. Also wird die Ausgabevariable repräsentiert durch $y = x_i$ mit $i = k + j \cdot 2^m$ (Abb. 5.24).
Im Falle der Realisierung des 2-MUX mittels Kaskadierung von 1-MUX (letztes Beispiel) gilt folglich ($m = 1$)

$$\begin{aligned} x &= x^0 \circ x^1 \text{ mit } x^j = (x^j_0, x^j_1) \\ c &= (c_0, c_1) = c^{(k)} \circ c^{(j)} \text{ mit } c^{(k)} = c_0,\ c^{(j)} = c_1, \text{ also} \\ k_{10} &= (c_0)_2,\ j_{10} = (c_1)_2 \end{aligned}$$

Für $i_{10} = (c_1 c_0)_2, 0 \le i \le 3$, folgt also

$$\begin{aligned} c &= (0,0):\ i = 0,\ j = 0,\ k = 0 \\ c &= (\mathrm{L},0):\ i = 1,\ j = 0,\ k = 1 \end{aligned}$$

$$
\begin{aligned}
c &= (0, \mathrm{L}) : \; i = 2, \; j = 1, \; k = 0 \\
c &= (\mathrm{L}, \mathrm{L}) : \; i = 3, \; j = 1, \; k = 1
\end{aligned}
$$

- Kaskadierte MUX ergeben gegenüber normalen MUX eine erhebliche Einsparung an Gattern:

normaler m-MUX:	$G(m) = 2^m(m+1) - 1$
	$m = 4 : 79, \; m = 8 : 2303$
kaskadiertes m-MUX:	$G(m) = 3(2^m - 1)$
	$m = 4 : 45, \; m = 8 : 765$

- Die Daten werden in der CPU von der Datenleitung auf das Rechenwerk (ALU) als Halbbyte oder als Byte multiplext. Dadurch wird der Aufwand reduziert.

- Das Multiplexen der Daten zwischen Registern und ALU erfolgt bitweise.

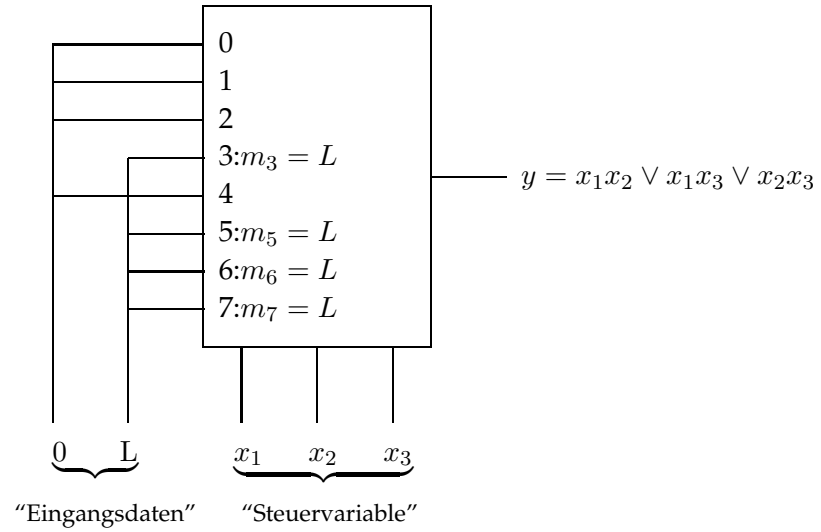
- Mittels Multiplexern lassen sich beliebige Boolesche Funktionen realisieren. Dabei werden die n Booleschen Variablen zu Steuervariablen des Multiplexers und die 2^n Eingänge der Wahrheitstabelle werden fest an 0 oder L angeschlossen.

Beispiel: 2-von-3-Mehrheitsfunktion

$f(x_1, x_2, x_3) : D_f^{kan}$

$D_f^{kan} = m_3 \vee m_5 \vee m_6 \vee m_7 \; = \; x_1 x_2 \overline{x}_3 \vee x_1 \overline{x}_2 x_3 \vee \overline{x}_1 x_2 x_3 \vee x_1 x_2 x_3$

Die Minterme werden auf L geschaltet.



5.4.2.2 Demultiplexer

Definition: (Demultiplexer)

Ein Demultiplexer ist ein verteilendes Schaltnetz

$$y_i = f_i(x, c_0, \cdots, c_{m-1}),$$

das eine Eingangsvariable x mittels m Steuervariablen c_j, $0 \leq j \leq m-1$, auf eine von $n = 2^m$ Ausgangsvariablen y_i, $0 \leq i \leq 2^m - 1$, entsprechend $i = (c_{m-1} \cdots c_0)_2$ schaltet. (Abb. 5.25)

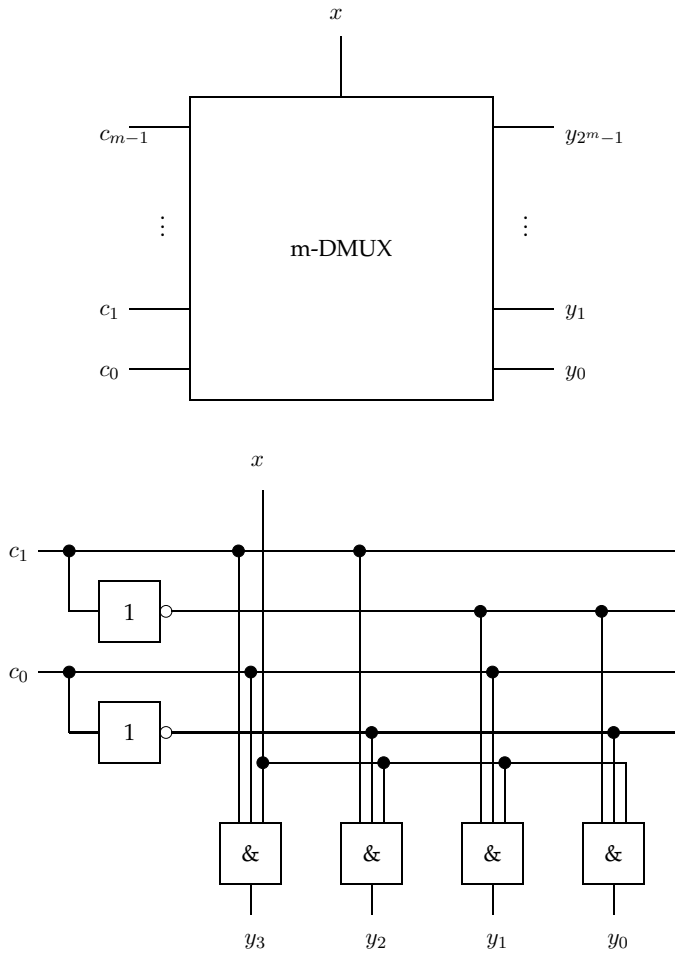


Abbildung 5.25: m-DMUX-Gatter und 2-DMUX Schaltnetz

Beispiel: 1:4-Demultiplexer (2-DMUX) (Abb. 5.25)

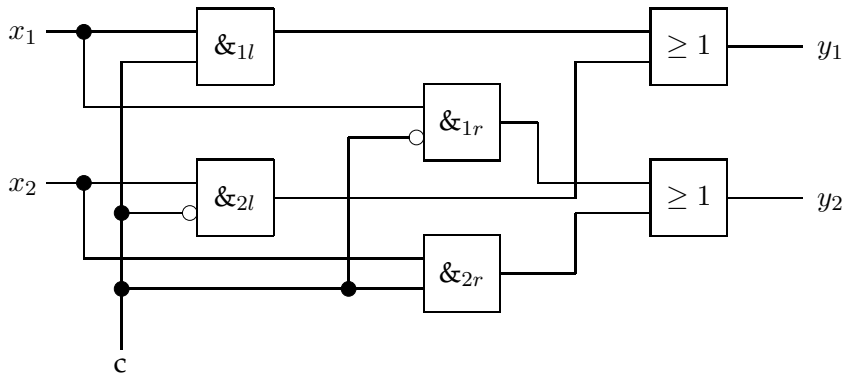
i	x	c_1	c_0	y_i
0	x	0	0	y_0
1	x	0	L	y_1
2	x	L	0	y_2
3	x	L	L	y_3

$$
\begin{aligned}
y_0 &= \overline{c}_0\overline{c}_1 x \\
y_1 &= c_0\overline{c}_1 x \\
y_2 &= \overline{c}_0 c_1 x \\
y_3 &= c_0 c_1 x
\end{aligned}
$$

Kombinationen von Multiplexern mit Demultiplexern eignen sich als Permutationsschaltnetz.

Beispiel: Weiche mit 2 Eingängen und 2 Ausgängen

y_1	y_2	c
x_1	x_2	L
x_2	x_1	0



Ausführliche Darstellung der Wertetabelle:

c	x_1	x_2	$\&_{1l}$	$\&_{2l}$	$\&_{1r}$	$\&_{2r}$	y_1	y_2
0	0	0	0	0	0	0	0	0
0	0	L	0	L	0	0	L	0
0	L	0	0	0	L	0	0	L
0	L	L	0	L	L	0	L	L
L	0	0	0	0	0	0	0	0
L	0	L	0	0	0	L	0	L
L	L	0	L	0	0	0	L	0
L	L	L	L	0	0	L	L	L

5.4.2.3 Komparatoren

Komparatoren sind Schaltnetze zum bitweisen Vergleich von Bitmustern (Boolesche Variable und Boolesche Vektoren) sowie von n-stelligen Binärzahlen (wertmäßiger Vergleich).

Anwendung: z.B. Auswertung von Bedingungen bei bedingten Sprüngen oder bei der Auswertung von Statusregistern (CPU).

a) Vergleich zweier n-stelliger Boolescher Vektoren

Geg.: $x^1 = (x_0^1, x_1^1, \cdots, x_{n-1}^1)$
$x^2 = (x_0^2, x_1^2, \cdots, x_{n-1}^2)$

Ges.: $y = F(x^1, x^2) = (y^1, y^2, y^3)$

mit

$$y^2 = \bigwedge_{i=0}^{n-1} (x_i^1 \Leftrightarrow x_i^2) = \mathrm{L} \quad \text{für } x^1 = x^2$$

$$y^1 = \bigwedge_{i=0}^{n-1} (\overline{x}_i^1 \, x_i^2) = \mathrm{L} \quad \text{für } x^1 < x^2$$

$$y^3 = \bigwedge_{i=0}^{n-1} (x_i^1 \, \overline{x}_i^2) = \mathrm{L} \quad \text{für } x^1 > x^2$$

Einer der drei Ausgänge y^j des Ergebnisvektors y wird den Wert $y^j = \mathrm{L}$ annehmen, wenn die Konjunktion über alle Stellen erfüllt ist. Von größerer praktischer Bedeutung sind aber Vergleiche über Untermengen der n Bits nach gleichem Schema. In beiden Fällen wird das Problem auf die Konjunktion von 1-Bit-Vergleichen zurückgeführt.

Annahme: 1-Bit-Vergleich $\sim y_i = F(x_i^1, x_i^2) = (y_i^1, y_i^2, y_i^3)$

a) $x_i^1 > x_i^2$: $y_i^3 = x_i^1 \overline{x}_i^2$ wegen $(y_i^3 = \mathrm{L}) \Leftrightarrow (x_i^1 = \mathrm{L}) \wedge (x_i^2 = 0)$

b) $x_i^1 < x_i^2 :\ y_i^1 = \overline{x}_i^1 x_i^2$ wegen $(y_i^1 = \mathrm{L}) \Leftrightarrow (x_i^1 = 0) \wedge (x_i^2 = \mathrm{L})$
c) $x_i^1 = x_i^2 :\ y_i^2 = x_i^1 \Leftrightarrow x_i^2$ wegen $(y_i^2 = \mathrm{L}) \Leftrightarrow (x_i^1 = x_i^2 = \mathrm{L}) \vee (x_i^1 = x_i^2 = 0)$
aber:
d) $(\overline{x_i^1 > x_i^2}) = (x_i^1 \leq x_i^2) \Leftrightarrow y_i^1 \vee y_i^2 = \overline{x_i^1 \overline{x}_i^2} = \overline{x}_i^1 \vee x_i^2$
wegen $((y_i^1 \vee y_i^2) = \mathrm{L}) \Leftrightarrow (y_i^3 = 0) \Leftrightarrow (\overline{y_i^3 = \mathrm{L}})$
oder

$$\begin{aligned} y_i^1 \vee y_i^2 &= \overline{x}_i^1 x_i^2 \vee (x_i^1 \Leftrightarrow x_i^2) \\ &= \overline{x}_i^1 x_i^2 \vee x_i^1 x_i^2 \vee \overline{x}_i^1 \overline{x}_i^2 \\ &= x_i^2 (x_i^1 \vee \overline{x}_i^1) \vee \overline{x}_i^1 \overline{x}_i^2 \\ &= x_i^2 \vee \overline{x}_i^2 \overline{x}_i^1 = \overline{x}_i^1 \vee x_i^2 \end{aligned}$$

Sei $y_i^4 = \overline{y_i^3} = y_i^1 \vee y_i^2$ eine weitere Boolesche Funktion.

x_i^2	x_i^1	$y_i^1 : x_i^1 < x_i^2$	$y_i^2 : x_i^1 = x_i^2$	$y_i^3 : x_i^1 > x_i^2$
0	0	0	L	0
0	L	0	0	L
L	0	L	0	0
L	L	0	L	0

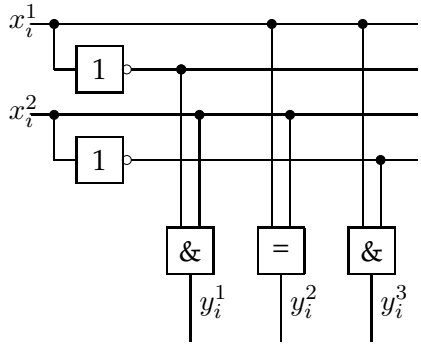


Abbildung 5.26: 1-Bit Vergleich

Der Vergleich zweier Bitvektoren erfolgt bitweise. So ist $y^3 = \mathrm{L}$ für $x^1 > x^2$ bzw. $(x^1 = \mathrm{L} = (\mathrm{L}_{(0)}, \cdots, \mathrm{L}_{(n-1)})) \wedge (x^2 = 0 = (0_{(0)}, \cdots, 0_{(n-1)}))$.
Schaltplan (für die Bedingung $x^1 > x^2$): entspricht der Konjunktion

$$y^3 = y_0^3 \wedge y_1^3 \wedge \ldots y_{n-1}^3$$

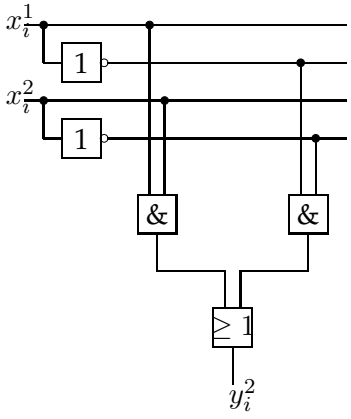
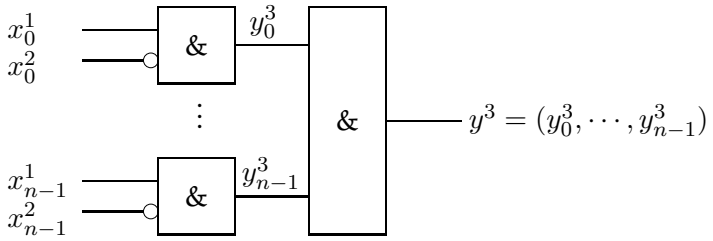
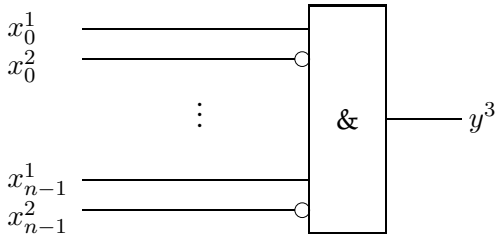


Abbildung 5.27: Äquivalenzfunktion y_i^2



bzw.



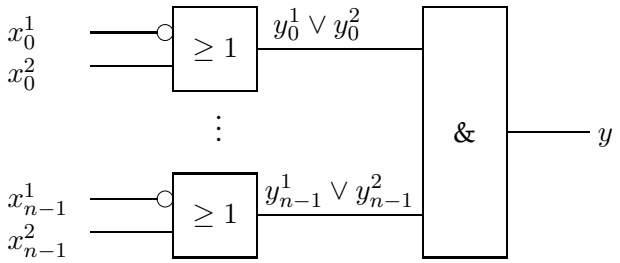
Die obere, kaskadierte Variante erfordert $n \cdot 2^2 + 2^n$ Minterme in der KDNF, die untere, solitäre Variante erfordert hingegen 2^{2n} Minterme. Die geeignete hardwaremäßige Strukturierung eines größeren Problems in kleinere ist also von Vorteil.

Beispiel:

Meldeschaltung für die Bedingung "x^2 hat mindestens an den Bits ein L, wo auch x^1 eines hat"

Bedingung: $x^2 \geq x^1 : x_i^2 = \mathrm{L} \rightarrow$ Wert von x_i^1 irrelevant
$x_i^2 = 0 \rightarrow x_i^1 = 0$ bzw. $\overline{x}_i^1 = \mathrm{L}$
für $i = 0, 1, \cdots, n-1$

Die Aufgabe entspricht offenbar der oben eingeführten Funktion $y^4 = \bar{y}^3$.
Ausgabefunktion: $y^4 = \bigwedge_{i=0}^{n-1} y_i^4$ mit $y_i^4 = y_i^1 \vee y_i^2 = x_i^2 \vee \overline{x}_i^1$ für $i = 1, \ldots, n-1$



b) Vergleich zweier n-stelliger Binärzahlen

Der Vergleich zweier Binärzahlen erfolgt wie der Vergleich zweier Bitvektoren mit der gleichen Länge, aber $\mathrm{L} \rightarrow 1$, $0 \rightarrow 0$ in den Bitvektoren.

Beispiel:

Gegeben: $z = (x_1 x_0)_2$, $z' = (x_1' x_0')_2$
Gesucht: $y^3 = z\overline{z}' \Leftrightarrow (z > z')$
Nur wenn $x_1 = x_1'$, muß $x_0 > x_0'$ getestet werden.

Es gilt offensichtlich $y^3 = y_1^3 \vee y_1^2 y_0^3$, d.h. $z > z'$, wenn $x_1 > x_1'$ oder $x_1 = x_1'$ und $x_0 > x_0'$.

Wertetafel:

	y_1^3	$y_1^2\, y_0^3$	y^3		
(0)	0	0	0	$x_1 \leq x_1'$	} weitere Analyse
(1)	0	L	L	$x_1 \leq x_1'$	} weitere Analyse
(2)	L	0	L	$x_1 > x_1'$	} $z > z'$
(3)	L	L	L	$x_1 \geq x_1'$	} $z > z'$

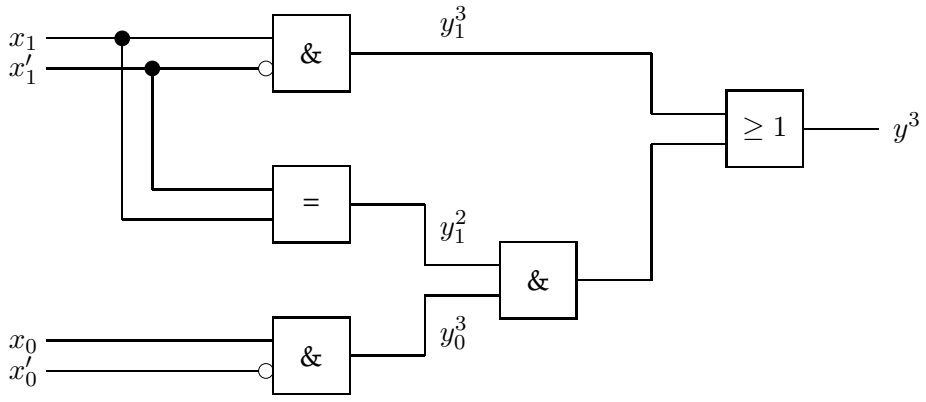
$z > z'$ gilt für
$(2):\ (10)_2 > (01)_2'\,,\ (10)_2 > (00)_2'$
$(2):\ (11)_2 > (01)_2'\,,\ (11)_2 > (00)_2'$
$(3):\ (01)_2 > (00)_2'\,,\ (11)_2 > (10)_2'$

Aufspaltung der "$\leq$"-Situationen (0), (1):

	y_0^3	y_1^2	$y_1^2\, y_0^3$	y^3			
(0)	0	0	0	0	$(x_1 < x_1') \wedge (x_0 \leq x_0')$	$\rightarrow$	$z < z'$
(0)	0	L	0	0	$(x_1 = x_1') \wedge (x_0 \leq x_0')$	$\rightarrow$	$z \leq z'$
(0)	L	0	0	0	$(x_1 < x_1') \wedge (x_0 > x_0')$	$\rightarrow$	$z < z'$
(1)	L	L	L	L	$(x_1 = x_1') \wedge (x_0 > x_0')$	$\rightarrow$	$z > z'$

$z > z'$ gilt für
$(1):\ (01)_2 > (00)_2'$
$(1):\ (11)_2 > (10)_2'$

Offensichtlich gilt die Äquivalenz der Fälle (1) und (3).

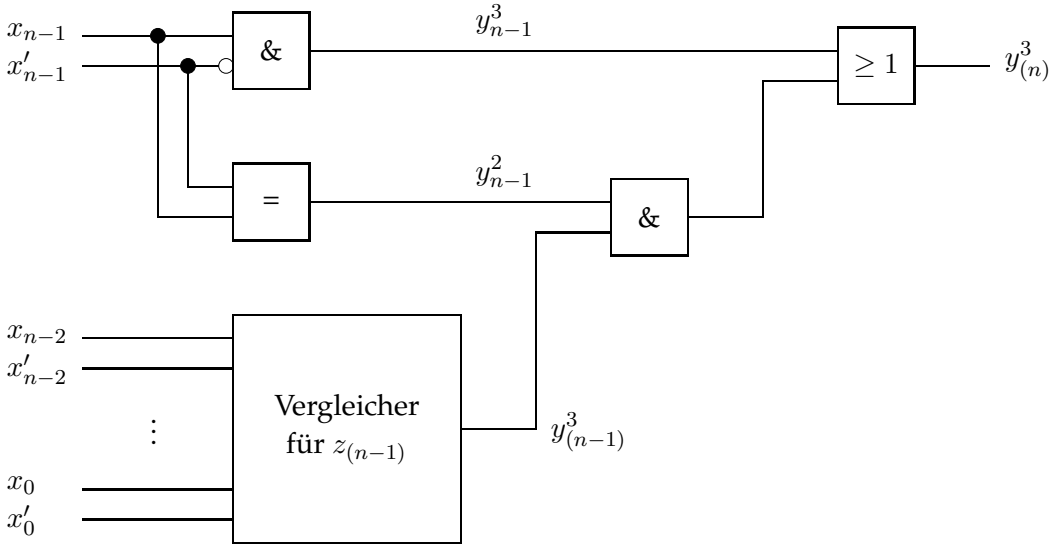


Verallgemeinerung: Vergleich n-stelliger Binärzahlen $(z > z')$.

Gegeben: $z_{(n)} = (x_{n-1}\, x_{n-2} \cdots x_1\, x_0)_2$
$z'_{(n)} = (x'_{n-1}\, x'_{n-2} \cdots x'_1\, x'_0)_2$
Gesucht: $y^3_{(n)} = (y^3_{n-1}, y^3_{n-2}, \cdots, y^3_1, y^3_0) \Leftrightarrow z\overline{z}' \Leftrightarrow (z > z')$
Bemerkung: Indizes in Klammern geben die Stelligkeit an.

Der Vergleich n-stelliger Binärzahlen erfolgt rekursiv durch den Vergleich (n-1)-stelliger Binärzahlen:

$$\begin{aligned} y^3_{(n)} &= y^3_{(n-1)}\, y^2_{n-1} \vee y^3_{n-1} \\ y^3_{(n)} &= y^3_{(n-1)}\, (x_{n-1} \Leftrightarrow x'_{n-1}) \vee x_{n-1}\overline{x}'_{n-1} \\ &= y^3_{(n-1)}\, (x_{n-1}x'_{n-1} \vee \overline{x}_{n-1}\overline{x}'_{n-1}) \vee x_{n-1}\overline{x}'_{n-1} \\ y^3_{(0)} &= 0 \end{aligned}$$



Beispiel:

Wie lautet die DNF von $y^3_{(3)} = f(x_0, x_1, x_2, x'_0, x'_1, x'_2)$ nach der rekursiven Formel?

Auflösung der Rekursion:

$$\begin{aligned} y^3_{(1)} &= x_0\overline{x}'_0 \\ y^3_{(2)} &= y^3_{(1)}\, (x_1x'_1 \vee \overline{x}_1\overline{x}'_1) \vee x_1\overline{x}'_1 \end{aligned}$$

$$
\begin{aligned}
& = x_0\overline{x}'_0x_1x'_1 \vee x_0\overline{x}'_0\overline{x}_1\overline{x}'_1 \vee x_1\overline{x}'_1 \\
y^3_{(3)} & = y^3_{(2)}\,(x_2x'_2 \vee \overline{x}_2\overline{x}'_2) \vee x_2\overline{x}'_2 \\
& = x_0\overline{x}'_0x_1x'_1x_2x'_2 \vee x_0\overline{x}'_0\overline{x}_1\overline{x}'_1x_2x'_2 \vee x_1\overline{x}'_1x_2x'_2 \vee \\
\vee & \quad x_0\overline{x}'_0x_1x'_1\overline{x}_2\overline{x}'_2 \vee x_0\overline{x}'_0\overline{x}_1\overline{x}'_1\overline{x}_2\overline{x}'_2 \vee x_1\overline{x}'_1\overline{x}_2\overline{x}'_2 \vee x_2\overline{x}'_2
\end{aligned}
$$

5.4.3 Addierwerke für Binärzahlen

Schaltungen zur Realisierung der Addition von Binärzahlen sind eigentlich Schaltwerke, weil sie taktgesteuert arbeiten. Dies hat ihre Ursache in der kalkülmäßigen Ausführung der Addition. Da die Ausführung des Additionskalküls aber den Entwurf paralleler und serieller Addierwerke (und Mischformen) zuläßt, sollen hier nur die Eigenschaften als Schaltnetze betrachtet werden.

5.4.3.1 Kalkülmäßige Addition von Binärzahlen

Prinzipiell erfolgt die Addition von Binärzahlen wie die von Dezimalzahlen (siehe Abschnitt 2.4.2.2). Dies hat seine Ursache in der Darstellung von Zahlen in einem Stellenwertsystem.
Gegeben seien zwei b-adische Zahlen $x = (\alpha_{n-1}\alpha_{n-2}\cdots\alpha_0)_b$ und $y = (\beta_{n-1}\beta_{n-2}\cdots\beta_0)_b$. Gesucht sei die b-adische Zahl $z = (\gamma_{n-1}\gamma_{n-2}\cdots\gamma_0)_b$ als Summe $z = x+y$ mit $\alpha_i, \beta_i, \gamma_i \in \sum_b$ für $i = 0, 1, \cdots, n-1$. Die Zahl z berechnet man durch sukzessives Addieren der Koeffizienten α_i und β_i, beginnend bei $i = 0$. Dabei ist an der Stelle i der ggf. auftretende Übertrag u_i von der Stelle $i-1$ zu berücksichtigen ($u_i \in \sum_b$):

$$
\begin{aligned}
\gamma_i &= (\alpha_i + \beta_i + u_i) \bmod b \\
u_i &= (\alpha_{i-1} + \beta_{i-1} + u_{i-1}) \operatorname{div} b \\
u_0 &= 0
\end{aligned}
$$

Beispiel: $7_{10} + 14_{10} = 21_{10}$

1	0	i
0	7	α_i
1	4	β_i
1	0	u_i
2	1	γ_i

$(0111)_2 + (1110)_2 = (10101)_2$

	3	2	1	0	i
	0	1	1	1	α_i
	1	1	1	0	β_i
	1	1	0	0	u_i
1	0	1	0	1	γ_i

Ein und dasselbe Problem stellt sich in verschiedenen Basissystemen unterschiedlich dar. Allgemein gilt, wenn b die verwendete Basis und n die Stelligkeit der Repräsentation sind, daß $N = b^n$ verschiedene Zahlen repräsentiert werden können.

Die Addition der Dezimalzahlen 7 und 14 ist für $n_{10} = 2$ ausführbar, weil $z_{10} = 21_{10} \leq 10^2 - 1 = 99$. Die Addition der entsprechenden Binärzahlen überschreitet die zur Darstellung der Summanden ausreichende Stelligkeit $n_2 = 4$, da $z_2 = (10101)_2 > 2^4 - 1 = 15$. Es tritt ein Überlauf auf, der für $n_2 = 5$ vermieden wird. Der Überlauf entspricht dem Übertrag des höchsten Bits.

Die kalkülmäßige Ausführung der Addition führt zu einer starken Reduzierung der zum Entwurf des Schaltnetzes erforderlichen Gatter. Das Problem ist vergleichbar mit der rekursiven Formulierung des Vergleiches von n-stelligen Binärzahlen (Abschnitt 5.4.2.3).

a) *Top-Down-Entwurf* eines 16-bit-Addierwerkes (monolithischer Baustein)
$A : \mathbb{B}^{16} \times \mathbb{B}^{16} \rightarrow \mathbb{B}^{17}$ (einschließlich Überlauf-Bit)
Jede der 17 Booleschen Funktionen a_i hat $2^{32} \sim 4 \cdot 10^9$ Argument-Tupel
Annahme: etwa 50% der Argument-Tupel führen zu $a_i = \mathrm{L}$

Daraus folgen etwa $2 \cdot 10^9 \cdot 17 = 3.4 \cdot 10^{10}$ Minterme zur Definition der 16-Bit-Addition.

Annahme: pro Minterm 15 UND-Gatter mit 2 Eingängen

$\rightarrow$ Die Addierschaltung erfordert ca. $5 \cdot 10^{11}$ UND-Gatter!

b) *Bottom-Up-Entwurf* eines 16-Bit-Addierwerkes (Partiallösungen)
Kalkülmäßige Addition mittels 15 Volladdierern (Bits 1 bis 15) und 1 Halbaddierer (Bit 0):

1 Volladdierer $\triangleq$ 2 Halbaddierer $\triangleq$ 7 UND-Gatter
1 Halbaddierer $\triangleq$ 3 UND-Gatter

Also werden 108 UND-Gatter für die kalkülmäßige Addition erforderlich sein.

Halbaddierer:

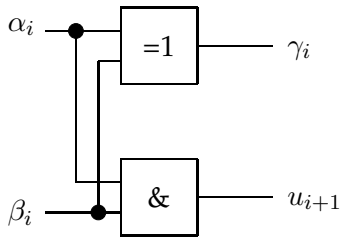
Definition: (Halbaddierer)

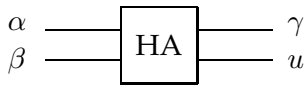
Ein Schaltnetz $F : \mathbb{B}^2 \to \mathbb{B}^2$, das zwei einstellige Binärzahlen addiert und Summe sowie Übertrag bildet, heißt Halbaddierer.

α_i	β_i	γ_i	u_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Aus der Wertetabelle ist erkennbar:

$$\begin{aligned} \gamma_i &= (\alpha_i \overline{\beta}_i) \vee (\overline{\alpha}_i \beta_i) = \alpha_i \not\Leftrightarrow \beta_i \\ u_{i+1} &= \alpha_i \wedge \beta_i \end{aligned}$$





Der Halbaddierer eignet sich zur Ausführung der Addition des Bits Null von n-stelligen Binärzahlen. Für höherwertige Bits i muß der Übertrag von Bit $i-1$ als dritter Summand berücksichtigt werden.

Definition: (Volladdierer)

Ein Schaltnetz $F : \mathbb{B}^3 \to \mathbb{B}^2$, das drei einstellige Binärzahlen addiert und daraus Summe und Übertrag bildet, heißt Volladdierer.

Sei $u_{i-1} = c_i$ der zu berücksichtigende Übertrag der vorherigen Summe, so daß $u_i = c_{i+1}$, $c_0 = 0$. Das Carry-Bit c_i ist dritter Summand für Bit i.

c_i	α_i	β_i	γ_i	u_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Aus der Wertetabelle ist erkennbar:

$$
\begin{array}{lcll}
\gamma_i & = & \alpha_i\overline{\beta}_i\overline{c}_i \vee \overline{\alpha}_i\beta_i\overline{c}_i \vee \overline{\alpha}_i\overline{\beta}_i c_i \vee \alpha_i\beta_i c_i & \text{KDNF} \\
 & = & \alpha_i \not\Leftrightarrow \beta_i \not\Leftrightarrow c_i & \text{D}_{\text{min}} \\
u_i & = & \alpha_i\beta_i\overline{c}_i \vee \alpha_i\overline{\beta}_i c_i \vee \overline{\alpha}_i\beta_i c_i \vee \alpha_i\beta_i c_i & \text{KDNF} \\
u_i & = & \alpha_i\beta_i \vee \alpha_i c_i \vee \beta_i c_i & \\
 & = & \alpha_i\beta_i \not\Leftrightarrow \alpha_i c_i \not\Leftrightarrow \beta_i c_i & \text{D}_{\text{min}}
\end{array}
$$

Abbildung 5.28 zeigt den Schaltplan und und das Schaltsymbol eines Volladdierers.

Die Funktion des Volladdierers läßt sich als Kaskadierung zweier Halbaddierer erklären. Der Übertrag im Volladdierer wird offensichtlich nach der in diesem Kapitel häufig als Beispiel verwendeten 2-von-3-Mehrheitsfunktion berechnet. Das Ergebnis der Addition wird nach einer Funktion berechnet, die man auch als 2-von3-*Minderheitsfunktion* bezeichnet. Für beide Funktionen, Übertrag und Ergebnis ist die Antivalenzoperation (bzw. XOR) die maßgebliche Operation. Zunächst erstaunt, daß $\gamma_i = 1$ für $\alpha_i = \beta_i = c_i = 1$. Führt man XOR für diese Argumente aber auf XOR für zwei Argumente zurück, wird das Ergebnis einsichtig.

Eine Normalform im vollständigen System $\{\not\Leftrightarrow, \wedge, \bar{}\}$ heißt Ringsummen-Normalform.

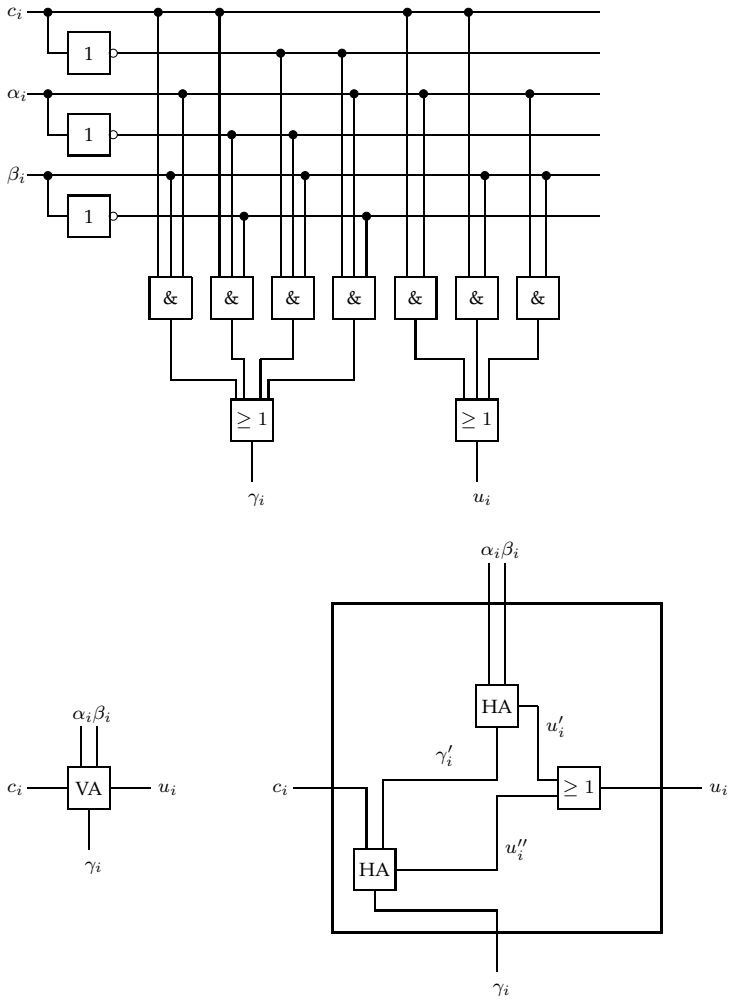


Abbildung 5.28: Volladdierer: Schaltnetz und Schaltsymbol

Definition: (Ringsummen-Normalform)

Eine durch die Antivalenz gebildete Normalform einer Booleschen Funktion f mit den Konjunktionstermen t_j

$$R_f = \mathop{\not\Leftrightarrow}_{j=1}^{p} t_j$$

heißt Ringsummen-Normalform (RNF). Sind die Terme t_j identisch mit den Mintermen $m_j, j \in J \subseteq \{0, \cdots, 2^n - 1\}$, so bildet

$$R_f^{kan} = \mathop{\not\Leftrightarrow}_{j \in J} m_j$$

die kanonische Ringsummen-Normalform (KRNF) der Funktion f.

Eigenschaften der Antivalenz (Ringsumme) im System $\{\not\Leftrightarrow, \wedge, \overline{\ }\}$:

Für alle $x, y, z \in \mathbb{B}$ gilt

1.) $x \not\Leftrightarrow \mathrm{L} = \overline{x},\ x \not\Leftrightarrow 0 = x$

2.) $x \not\Leftrightarrow x = 0,\ x \not\Leftrightarrow \overline{x} = \mathrm{L}$

3.) $x \not\Leftrightarrow y = y \not\Leftrightarrow x$ Kommutativität

4.) $x \not\Leftrightarrow (y \not\Leftrightarrow z) = (x \not\Leftrightarrow y) \not\Leftrightarrow z$ Assoziativität

5.) $x \wedge (y \not\Leftrightarrow z) = x \wedge y \not\Leftrightarrow x \wedge z$ Distributivität bzgl. $\wedge$

6.) $0 \not\Leftrightarrow 0 \not\Leftrightarrow \cdots \not\Leftrightarrow 0 = 0$

7.) $\underbrace{\mathrm{L} \not\Leftrightarrow \mathrm{L} \not\Leftrightarrow \cdots \not\Leftrightarrow \mathrm{L}}_{m - \text{mal}} = \begin{cases} \mathrm{L} & \text{falls } m \text{ ungerade} \\ 0 & \text{falls } m \text{ gerade} \end{cases}$

5.4.3.2 Serien- und Paralleladdierer

Das Addieren n-stelliger Binärzahlen kann bitseriell oder bitparallel erfolgen. Beide Varianten unterscheiden sich bezüglich

- Hardwareaufwand

- Addierzeit.

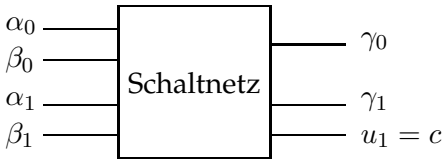
Serienaddierer:
Der Serienaddierer ist ein Schaltwerk, bestehend aus

- einem Schaltnetz
- Registern zum Aufnehmen von Summanden und Summe
- Speicherglied zum Zwischenspeichern des Übertrages.

In jedem Taktschritt der Dauer t wird eine Stelle addiert, so daß für die Addition n-stelliger Binärzahlen die Zeit $t_n = n \cdot t$ erforderlich ist. Der Serienaddierer hat keine praktische Bedeutung.
Der Serienaddierer ist ein Addierwerk mit minimalem Schaltungsaufwand aber maximaler Addierzeit!

Normalform-Paralleladdierer:
Die Addierzeit ist von der Stelligkeit der Summanden unabhängig. Alle Bits werden parallel in einem einzigen Schritt in einem dreistufigen Netzwerk mit den Ebenen NICHT, UND, ODER bearbeitet. Mit jeder zusätzlichen Stelle steigt der Hardwareaufwand sehr schnell an.

Beispiel: 2-Bit-Addition



c $\sim$ Überlauf-Bit (*Carry-Flag*)

Die Funktionsgleichungen der Ausgangsvariablen als DNF lauten:

$$\begin{aligned}
\gamma_0 &= \alpha_0\overline{\beta}_0 \vee \overline{\alpha}_0\beta_0 \\
&= \alpha_0 \not\Leftrightarrow \beta_0 \\
(u_0 &= \alpha_0\beta_0 = c_1) \\
\gamma_1 &= \alpha_0\beta_0\alpha_1\beta_1 \vee \alpha_0\beta_0\overline{\alpha}_1\overline{\beta}_1 \vee \overline{\alpha}_0\overline{\alpha}_1\beta_1 \vee \\
&\quad\ \overline{\alpha}_0\alpha_1\overline{\beta}_1 \vee \overline{\alpha}_0\overline{\alpha}_1\beta_1 \vee \overline{\beta}_0\alpha_1\overline{\beta}_1 \\
&= \alpha_1 \not\Leftrightarrow \beta_1 \not\Leftrightarrow c_1 \\
u_1 &= \alpha_1\beta_1 \vee \alpha_0\beta_0\alpha_1 \vee \alpha_0\beta_0\beta_1 \\
&= \alpha_1\beta_1 \vee \alpha_1 u_0 \vee \beta_1 u_0 \\
&= \alpha_1\beta_1 \not\Leftrightarrow \alpha_1 c_1 \not\Leftrightarrow \beta_1 c_1
\end{aligned}$$

Da $c_0 = 0$ und $x \not\Leftrightarrow 0 = x$, kann γ_0 ohne c_0 dargestellt werden.

Hieraus wird erkennbar, daß die Addition n-stelliger Summanden ein Schaltnetz mit $2n$ Eingängen und $n+1$ Ausgängen erfordert. Dabei sind 2^{2n-1} Minterme zu verknüpfen. Dies führt bei $n = 16$ bereits zu nicht mehr realistischen Gatterzahlen.

Der Normalform-Paralleladdierer ist ein Addiernetz mit minimaler Addierzeit und maximalem Schaltungsaufwand!

Ripple-Carry-Adder:

Der Ripple-Carry-Adder besteht aus einer Folge von n Volladdierern, die über das Carry-Bit jeder Stufe zu einer Kaskade verschaltet sind. Mit jeder weiteren Stelle wächst die Schaltungstiefe um zwei. Der Ripple-Carry Adder arbeitet nach dem *Pipeline-Prinzip*. Das heißt, daß nach der Berechnung der Stelle i die nächsten Operanden geladen werden können. Aber erst nach Durchlaufen der gesamten Kette liegt das komplette Ergebnis der Summenbildung vor. Die Addierzeit ist proportional der Stellenzahl der Summanden, ebenso der Schaltungsaufwand.

Der Ripple-Carry Adder zeichnet sich durch linear mit der Stellenzahl n wachsenden Addierzeit und Schaltungsaufwand aus!

In der Abbildung 5.29 ist oben ein reines 4-Bit R-C Addiernetz angegeben und unten eine Variante, die sich sowohl zur Addition als auch zur Subtraktion von Binärzahlen nach dem Zweierkomplement eignet.

Im Falle der Addition liegt an den Antivalenz-Gattern konstant 0 an, da $x \not\Leftrightarrow 0 = x$. Im Falle der Subtraktion liegt konstant 1 an, da $x \not\Leftrightarrow 1 = \bar{x}$. Außerdem wird dem 0-stelligen Volladdierer zusätzlich der Wert 1 aufaddiert. Damit wird auf x eine im Zweierkomplement dargestellte negative Zahl y addiert (siehe 2.4.2.2 - Darstellung ganzer Zahlen).

Die Subtraktion wird also durch eine Addition realisiert, wobei ein Summand im Zweierkomplement dargestellt wird.

Einschub: Darstellung und Subtraktion ganzer Zahlen

$$z = x - y = x + K_2(y) - 2^n$$

Zweierkomplement:	$K_2(y) = K_1(y) + 1$
	erhält man durch Addition einer Eins zum
	Einerkomplement $K_1(y)$
Einerkomplement:	$K_1(y) = (2^n - 1) - y$
	erhält man durch bitweises Invertieren aus y

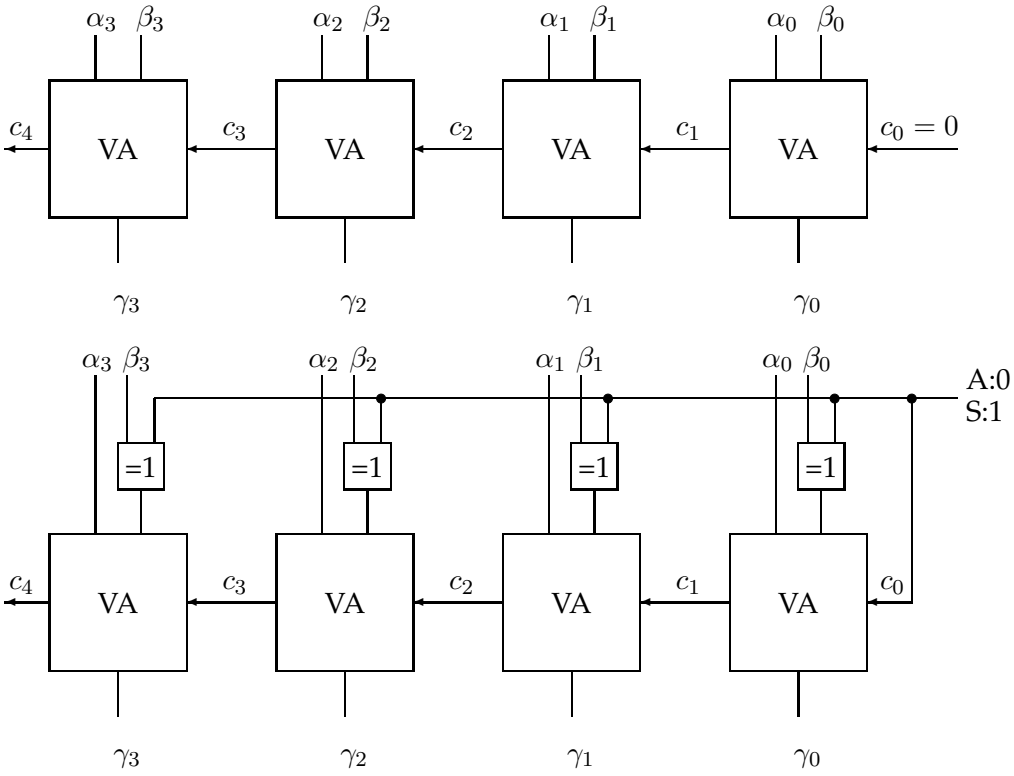


Abbildung 5.29: Ripple-Carry Adder. Oben: reiner Addierer, unten: wahlweise Addition oder Subtraktion

Für gegebene Stelligkeit n wird der darstellbare Bereich $N = 2^n$ für ganze Zahlen in folgender Weise aufgeteilt:

$$\begin{aligned} K_1(z) \quad &: \quad |z| \leq 2^{n-1} - 1 \text{ (symmetrisch)} \\ &\quad \text{mit } -0_{10} = \underbrace{11\cdots1}_{n}{}_2, +0_{10} = \underbrace{00\cdots0}_{n}{}_2 \\ K_2(z) \quad &: \quad -2^{n-1} \leq z \leq 2^{n-1} - 1 \text{ (asymmetrisch)} \\ &\quad \text{mit } 0_{10} = \underbrace{00\cdots0}_{n}{}_2 \end{aligned}$$

Beispiel: $n = 4, N = 2^4 = 16$

a) $K_1(z) : |z| \leq 2^3 - 1 = 7$
$z_{10} = 5 : z_2 = 0101_2$
$-5 = K_1(5) = (2^4 - 1) - 5 = -0_{10} - 5_{10} = 1111_2 - 0101_2 = 1010_2 = 10_{10}$

K	1	2	...	7	8	9	10	...	15	16
z_{10}	-7	-6	...	-1	-0	+0	1	...	6	7
z_2	1000	1001	...	1110	1111	0000	0001	...	0110	0111

b) $K_2(z) : -2^3 = -8 \leq z \leq 2^3 - 1 = 7$
$-5 = K_2(5) = K_1(5) + 1_{10} = 1010_2 + 0001_2 = 1011_2 = 11_{10}$

K	1	2	...	7	8	9	10	...	15	16
z_{10}	-8	-7	...	-2	-1	0	1	...	6	7
z_2	1000	1001	...	1110	1111	0000	0001	...	0110	0111

c) $z = x - y = x + (2^n - 1 - y) + 1 - 2^n = x + K_1(y) + 1 - 2^n = x + K_2(y) - 2^n$
$x_{10} = 3 : x_2 = 0011_2$, $y_{10} = 5 : y_2 = 0101_2$

$$\begin{array}{r|l} 3 & \\ -5 & \\ \hline -2 & \end{array}_{10} \qquad \begin{array}{r} 0011 \\ +\,1010 \quad K_1(5) \\ \hline 1101 \\ +\,0001 \\ \hline 1110 \quad K_2(5) \end{array}_{2} \qquad \begin{array}{r|l} 3 & \\ +2^4 - 1 - 5 = 10 & \\ \hline 13 & \\ +\,1 & \\ \hline 14 & \\ 14 - 2^4 = -2 & \end{array}_{10}$$

Carry-Look-Ahead Adder:
Der C-L-A Adder vereint die Parallelität des Normalform-Paralleladdierers mit dem linearen Anwachsen des Ripple-Carry Adders. Dies wird dadurch erreicht, daß

- der Übertrag jeder Stelle in einem zweistufigen Netzwerk aus den Eingangsdaten "vorausberechnet" (look ahead $\sim$ vorausschauen) wird und
- durch Rekursion an das Ergebnis der vorherigen Stelle angeknüpft wird.

Die Berechnung der Bitstellen-Summen erfolgt im Normalform-Paralleladdierer nach

$$\gamma_0 \quad = \quad \alpha_0 \not\Leftrightarrow \beta_0$$

$$\begin{aligned} \gamma_1 &= \alpha_1 \not\Leftrightarrow \beta_1 \not\Leftrightarrow c_1\,, \;\; c_1 = u_0 \\ \gamma_2 &= \alpha_2 \not\Leftrightarrow \beta_2 \not\Leftrightarrow c_2\,, \;\; c_2 = u_1 \end{aligned}$$

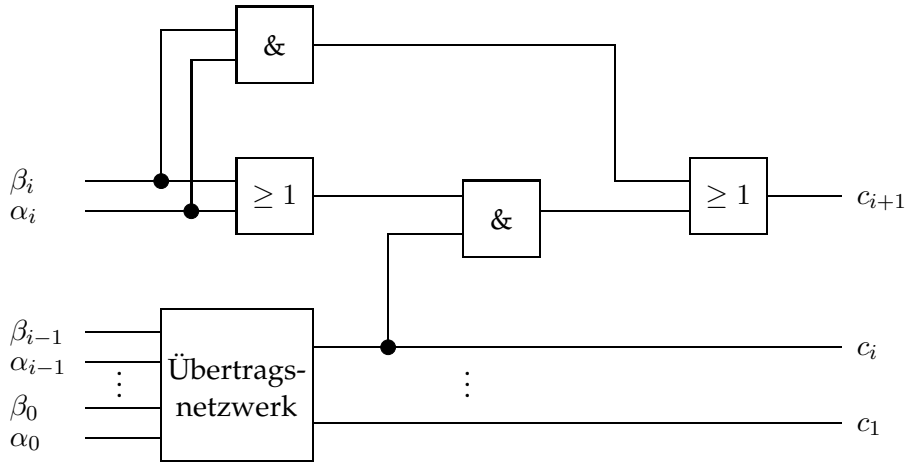
oder allgemein

$$\gamma_i = \alpha_i \not\Leftrightarrow \beta_i \not\Leftrightarrow c_i \text{ für } i = 0, \cdots, n-1 \text{ mit } c_0 = 0$$

Dabei entspricht das im Schritt i benötigte Carry-Bit c_i dem in Schritt $i-1$ berechneten Übertragsbit u_{i-1}. Also gilt die Rekursion

$$\begin{aligned} c_{i+1} &= \alpha_i\beta_i \vee \alpha_i c_i \vee \beta_i c_i \\ &= \alpha_i\beta_i \vee (\alpha_i \vee \beta_i)c_i \text{ für } i = 0, \cdots, n-1 \text{ mit } c_0 = 0 \end{aligned}$$

Der Übertrag einer Stelle wird also als Normalform aus den Eingangsvariablen dieser Stelle gebildet. Das entsprechende Schaltnetz (Abb. 5.30) ist zweistufig und von der Struktur her vergleichbar mit dem Schaltnetz des rekursiven Vergleichs von Binärzahlen.



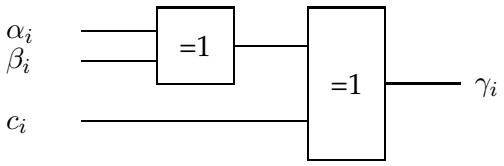


Abbildung 5.30: Rekursion beim Carry-Look-Ahead Adder

Das Carry-Bit c_n einer Addition zweier n-stelliger Summanden lautet nach Auflösung dieser Rekursion

$$c_n = \alpha_{n-1}\beta_{n-1} \vee \bigvee_{i=0}^{n-2} \left[\alpha_i\beta_i \bigwedge_{j=i+1}^{n-1} (\alpha_j \vee \beta_j) \right]$$

Zur Herleitung dieser Gleichung werden zwei Hilfsvariablen eingeführt.

$$\begin{aligned} g_i &= \alpha_i\beta_i \\ p_i &= \alpha_i \vee \beta_i \end{aligned}$$

Damit wird die Rekursion linearisiert

$$c_{i+1} = g_i \vee p_i c_i \; .$$

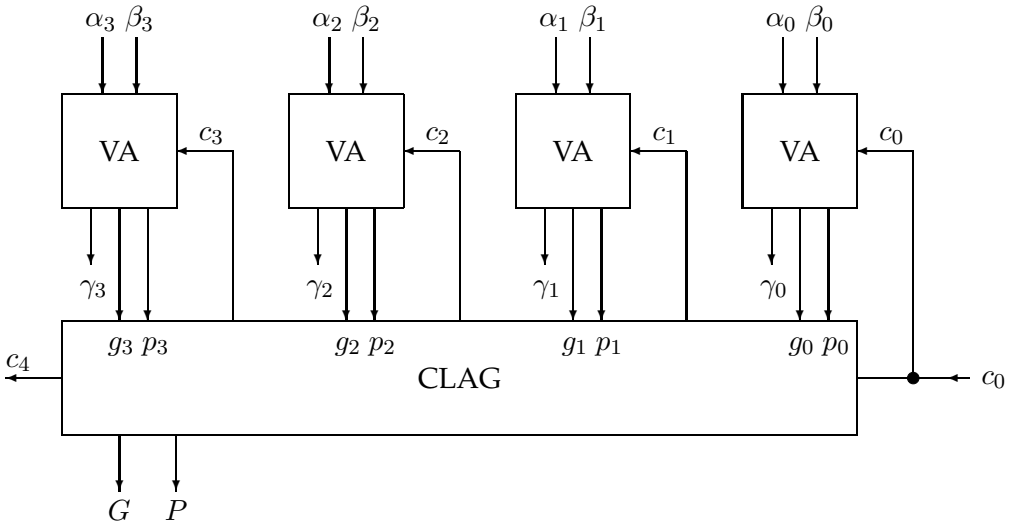
Bedeutung der Hilfsvariablen:

g_i: *Carry generate*: Übertrag c_{i+1} wird gebildet, wenn α_i und β_i gleich 1 sind
p_i: *Carry propagate*: Übertrag c_i wird weitergeleitet, wenn α_i oder β_i gleich 1 sind

Damit reduziert sich die Komplexität der Übertragsberechnung wesentlich. Obwohl die Schaltfunktionen von Stelle zu Stelle immer umfangreicher werden, sind lediglich zwei Takte bezüglich der Berechnung aus den Hilfsvariablen erforderlich. Der Grund hierfür liegt in der Normalform-Darstellung jeder Schaltfunktion.

Beispiel: 4-Bit C-L-A-Addierer mit Carry-Look-Ahead-Generator

$$\begin{aligned} c_1 &= g_0 \vee p_0 c_0 \\ c_2 &= g_1 \vee p_1 c_1 \\ &= g_1 \vee p_1 g_0 \vee p_1 p_0 c_0 \\ c_3 &= g_2 \vee p_2 c_2 \\ &= g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0 \\ c_4 &= g_3 \vee p_3 c_3 \\ &= \underbrace{g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0}_{G} \vee \underbrace{p_3 p_2 p_1 p_0}_{P} c_0 \\ c_4 &= G \vee P c_0 \end{aligned}$$



Der Übertrag c_i ist also eine DNF in $g_0, \cdots, g_{i-1}$ und $p_0, \cdots, p_{i-1}$ mit folgender Bildungsregel

$$c_i = g_{i-1} \vee \bigvee_{j=0}^{i-2} g_j \bigwedge_{k=j+1}^{i-1} p_k.$$

Auf den Beweis mittels vollständiger Induktion unter Anwendung der linearisierten Rekursion sei hier verzichtet. Die Hilfsvariablen g_i und p_i werden in den Volladdierern als Zwischenergebnisse gebildet und an den *Carry-Look-Ahead-Generator* (*CLAG*) zur Bildung der *Block-Generate* G und *Block-Propagate* P als Hilfsvariable für 4-Bit Blöcke weitergereicht. Der aktuelle CLAG übergibt G und P an den folgenden Block. Der IC SN 74181 stellt einen 4-Bit Addierer mit CLAG dar.

6 Schaltwerke

In diesem Kapitel werden sequentielle Schaltwerke und ihre abstrakten Beschreibungsmodelle als Automaten eingeführt. Schaltwerke zeichnen sich gegenüber Schaltnetzen dadurch aus, daß ihre Ausgaben $y_t \in Y$ zu einem bestimmten Zeittakt (oder Moment) sowohl von einer Folge von Eingabewerten $x_t \in X$ als auch von der Spezifik des Zustandes $s_t \in S$ abhängt. Also erzeugen Schaltwerke gedächtnisbehaftete Abbildungen der Art

$$G : S \times X \to Y$$

$$H : S \times X \to S$$

mit $X \in \mathbb{B}^n$, $Y \in \mathbb{B}^m$, $S \in \mathbb{B}^l$, $l, m, n \in \mathbb{N}$.

Ein Schaltwerk bildet also die Eingabemenge auf die Ausgabemenge ab, indem es die Menge S von Zuständen annehmen kann.

Schaltwerke sind wesentliche Funktionseinheiten eines von-Neumann-Rechners. Beispiele sind Rechen- und Steuerwerk der CPU und deren Komponenten. Die Zeit spielt für Schaltwerke eine große Rolle in der Weise, daß nur Aussagen zur Funktionalität eines Schaltwerkes bezüglich eines speziellen Augenblickes der Betrachtung abgeleitet werden können. Auch das Rechenwerk selbst benötigt eine verläßliche Zeitskala. Deshalb werden Taktsignale C neben den gewöhnlichen Eingabedaten X betrachtet. Sie synchronisieren die Zustandsfolgen des Schaltwerkes. Deshalb spricht man hier auch von synchronen Schaltwerken.

Definition: (synchrones Schaltwerk)

Bei einen synchronen Schaltwerk erfolgt der Übergang aus einem stabilen Anfangszustand in einen stabilen Folgezustand synchron mit dem Taktsignal. (siehe Abbildung 6.1)

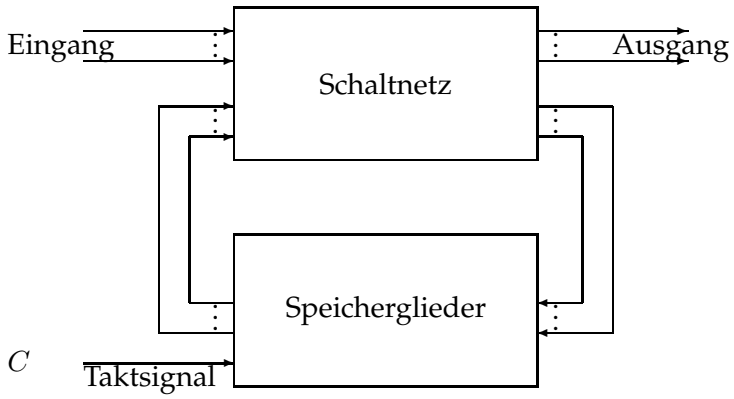


Abbildung 6.1: Prinzipieller Aufbau eines Synchron-Schaltwerkes

6.1 Programmierbare logische Felder (PLA)

Programmierbare logische Felder (PLA) stellen ein zu dem in Abschnitt 5.3 behandelten Minimierungsverfahren alternatives Entwurfsprinzip von Schaltnetzen dar. Die Behandlung gehört bezüglich der Systematik eigentlich in Kapitel 5, erfolgt aber hier, weil PLAs besonders für den hochintegrierten Entwurf von Schaltnetzen geeignet sind. Die Minimierung von Schaltnetzen mit dem Ziel einer minimalen DNF, gebildet aus Primtermen, hat den Nachteil, daß jedes Schaltnetz hochoptimiert für die zu realisierende Boolesche Funktion und sehr heterogen strukturiert ist (bzgl. Schaltgliedern und Zuleitungen). Dies stellt nur, wenn überhaupt, eine interessante Konzeption für MSI- oder LSI-Bausteine dar. Bei VLSI-Bausteinen überwiegt gegenüber dem Wunsch, Gatter einzusparen, der Wunsch, eine homogene und vielleicht flexibel verdrahtbare Grundkonfiguration zur Realisierung unterschiedlicher Schaltnetze bereitzustellen. Dies sind *PLAs*.

PLAs beruhen auf dem zweischichtigen Aufbau einer DNF:

1. Schicht:	Konjunktionen der Eingangsvariablen/-literale,
2. Schicht:	Disjunktionen der Konjunktionsterme.

Eventuell kann man noch eine Schicht Null unterscheiden, welche die Negationen der Variablen bereitstellt. Dies soll hier aber implizit mit Schicht 1 verbunden werden. Soll eine Schaltfunktion $F : \mathbb{B}^n \to \mathbb{B}^m$ durch Bildung von insgesamt p Konjunktionstermen realisiert werden nach dem Schema

$$F = (f_1, f_2, \ldots, f_m)$$

mit

$$f_i(x_1, \cdots, x_n) = \bigvee_{j=1}^{p_i} t_{k_{ij}} \qquad , p_i, k_{ij} \in \{1, \cdots, p\}$$

und

$$t_{k_{ij}}(x_1, \cdots, x_n) = \bigwedge_{l=1}^{q_{ij}} \tilde{x}_{k_{ijl}} \qquad , q_{ij}, k_{ijl} \in \{1, \cdots, n\}$$

bzw.

$$f_i(x_1, \cdots, x_n) = \bigvee_{j=1}^{p_i} \bigwedge_{l=1}^{q_{ij}} \tilde{x}_{k_{ijl}},$$

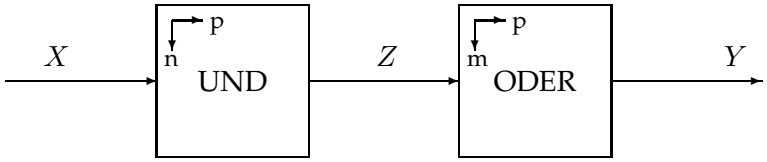
so führt die Verwendung von NAND-Gattern zu einem vereinfachten Entwurfsschema. NAND-Gatter bilden ein vollständiges Basissystem. Es gilt:

$$f_i = \overline{\overline{f}}_i = \overline{\bigwedge_{j=1}^{p_i} \overline{t_{k_{ij}}}},$$

so daß sowohl die Konjunktionsterme als auch die Booleschen Funktionen mit diesem Gattertyp erzeugt werden können (siehe Abb. 6.2).

Der regelmäßige Aufbau des Schaltnetzes läßt zwei Ebenen der Bildung von DNF erkennen, die sich als verkettete Matrizen interpretieren lassen:

1. Ebene : $(n \times p)$-Matrix bzw. UND-Matrix
2. Ebene : $(m \times p)$-Matrix bzw. ODER-Matrix



Hier repräsentiert Z das Zwischenergebnis in Gestalt der Folge von benötigten Konjunktionstermen $z_1, \ldots, z_p$. Im Falle $F = f$, d.h. $m = 1$, entartet die ODER-Matrix zu einem Vektor, der die DNF D_f darstellt.

Dies führt schließlich zur homogenen Struktur eines PLA. Dabei erweist es sich als vorteilhaft, die Knoten mit je zwei Ein- und Ausgängen auszustatten. Mit insgesamt 4 Knotentypen kann ein PLA jede Boolesche Funktion und jede Schaltfunktion realisieren (siehe Abb. 6.3).

Ein *PLA-Knoten* hat folgende logische Struktur: $k_i : \mathbb{B}^2 \to \mathbb{B}^2$

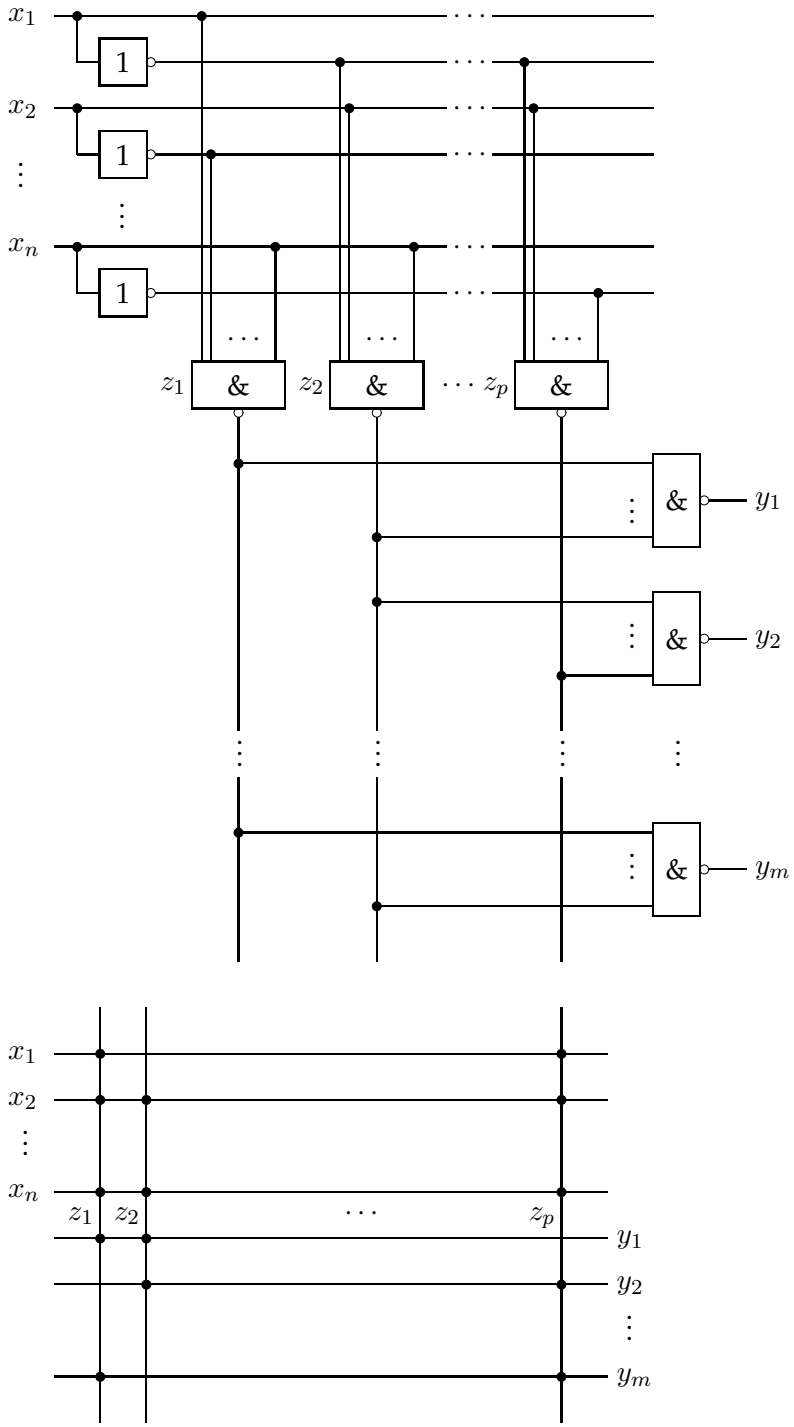
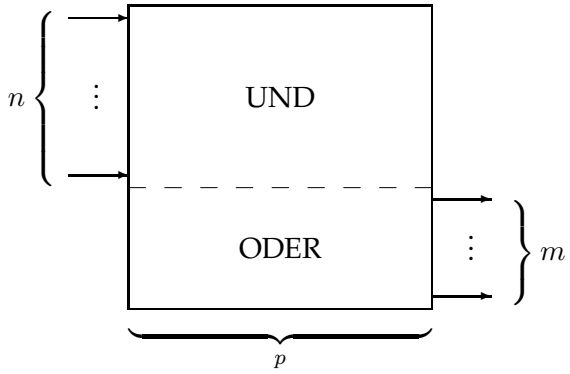


Abbildung 6.2: Oben: Schaltnetz aus NAND-Bausteinen zur Realisierung einer Schaltfunktion. Unten: Matrixförmige Struktur eines solchen Schaltnetzes



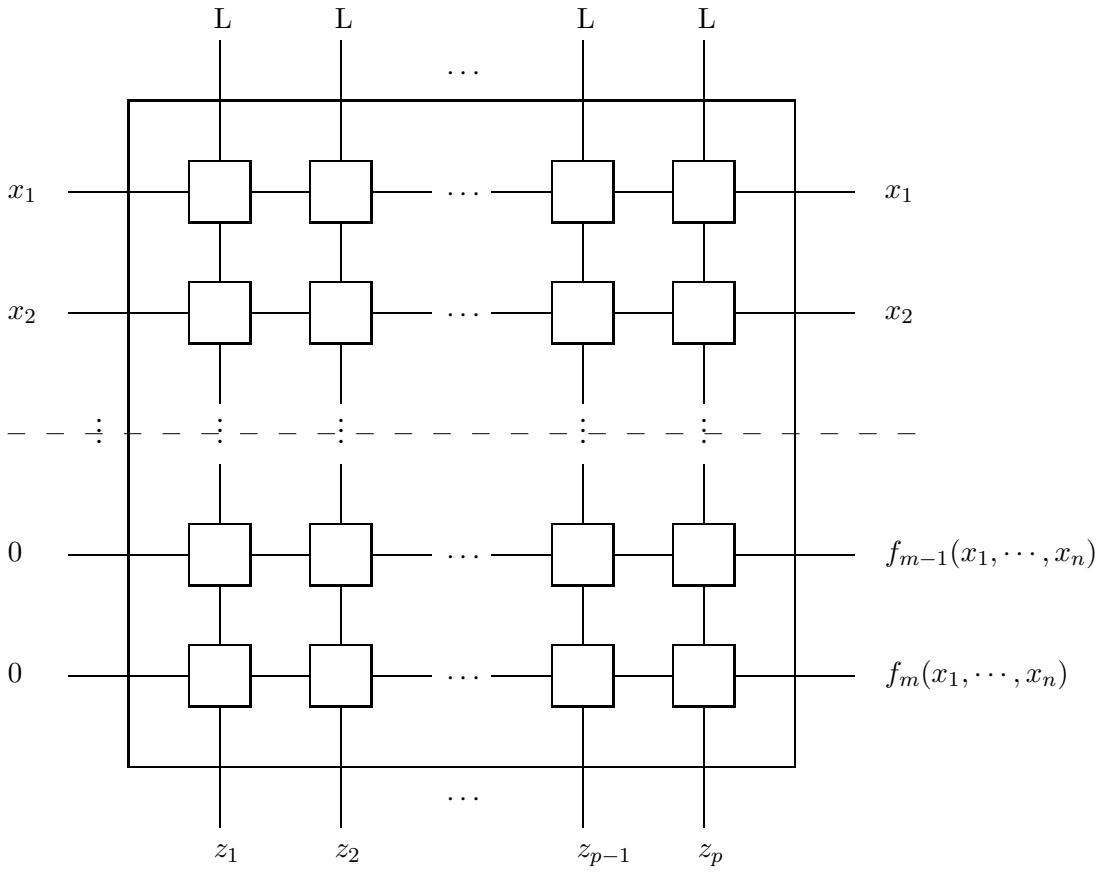
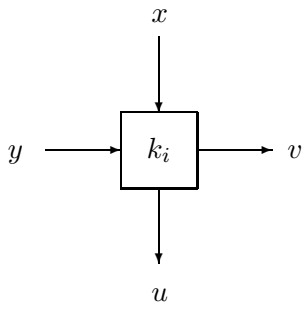
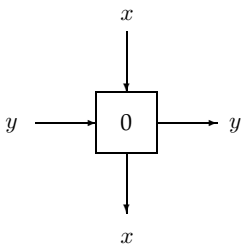


Abbildung 6.3: PLA-Matrix

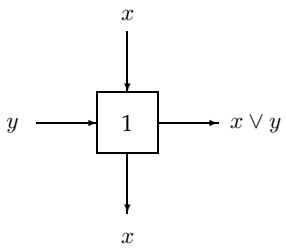


Eingänge: x, y Ausgänge: u, v

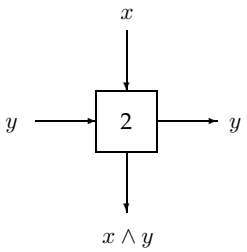
Die Bezeichnungen der Eingänge mit x und y ist nicht zu verwechseln mit der Verwendung dieser Symbole für Schaltfunktionen. Folgende 4 Grundtypen sind geeignet, beliebige Schaltfunktionen in einer matrixartigen Struktur zu realiseren:



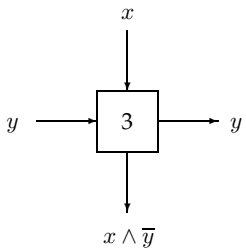
$k_0(x, y) = (x, y)$



$k_1(x, y) = (x, x \vee y)$



$k_2(x, y) = (x \wedge y, y)$



$k_3(x, y) = (x \wedge \overline{y}, y)$

Definition: (programmierbares logisches Feld (PLA))

Ein programmierbares logisches Feld besteht aus einer $\{(n+m) \times p\}$-Matrix von vier logischen Bausteintypen (Typ 0, $\cdots$, Typ 3), so daß in der oberen $\{n \times p\}$-Teilmatrix (UND-Matrix) nur die Typen 0, 2 oder 3 verwendet werden und in der unteren $\{m \times p\}$-Teilmatrix (ODER-Matrix) nur die Typen 0 oder 1 zur Anwendung kommen.

Konstruktion eines PLA mittels Grundbausteinen vom Typ $0, \cdots, 3$:

Annahme: $m = 1$, $F = f(x_1, \cdots, x_n) = \bigvee_{s=1}^{p} t_s$

Gesucht: PLA mit $n+1$ Zeilen und p Spalten mit den Knoten $K_{r,s}$, $1 \leq r \leq n+1$ und $1 \leq s \leq p$

a) UND-Ebene: $1 \leq s \leq p$, $1 \leq r \leq n$

In den Spalten der UND-Ebene werden die Konjunktionsterme t_s gebildet. Die Knotentypen werden folgendermaßen gewählt:

$$K_{r,s} = \begin{cases} k_2 & \text{falls} \quad t_s(\tilde{x_1}, \ldots, \tilde{x_n}) \wedge x_r = t_s \ (x_r \text{ kommt in } t_s \text{ vor}) \\ k_3 & \text{falls} \quad t_s(\tilde{x_1}, \ldots, \tilde{x_n}) \wedge \overline{x}_r = t_s \ (\overline{x}_r \text{ kommt in } t_s \text{ vor}) \\ k_0 & \text{sonst} \quad (\text{weder } x_r \text{ noch } \overline{x}_r \text{ kommen in } t_s \text{ vor}) \end{cases}$$

b) ODER-Ebene: $r = n+1$, $1 \leq s \leq p$
Die Funktion f wird in der ODER-Ebene durch Disjunktion der in der UND-Matrix gebildeten Konjunktionsterme gebildet. Dazu wird der Knotentyp 1 verwendet,

$$K_{n+1,s} = \begin{cases} k_1, \text{ falls } t_s \text{ zur Darstellung von } f \text{ erforderlich,} \\ k_0 \text{ sonst.} \end{cases}$$

Beispiel: Volladdierer $F_{VA} : \mathbb{B}^3 \to \mathbb{B}^2$, $F_{VA} = (\gamma_{VA}, u_{VA})$

Zunächst nur isolierte Betrachtung der Booleschen Funktionen für Summe γ_{VA} und Übertrag u_{VA}.

α	0	0	0	0	1	1	1	1
β	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
γ	0	1	1	0	1	0	0	1
u	0	0	0	1	0	1	1	1

$D_\gamma = \overline{\alpha}\overline{\beta}c \vee \overline{\alpha}\beta\overline{c} \vee \alpha\overline{\beta}\overline{c} \vee \alpha\beta c$ (Abb. 6.4)

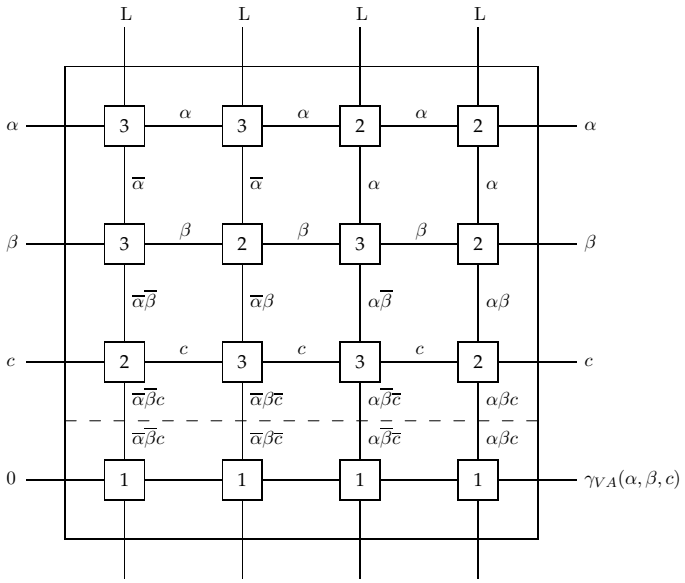


Abbildung 6.4: Volladdierer: PLA für D_γ

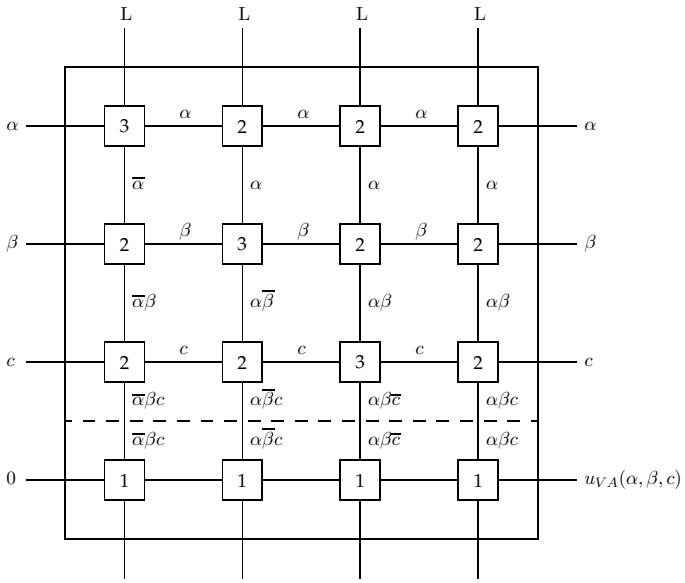


Abbildung 6.5: Volladdierer: PLA für D_u

$D_u = \overline{\alpha}\beta c \vee \alpha\overline{\beta}c \vee \alpha\beta\overline{c} \vee \alpha\beta c$ (Abb. 6.5)

Lediglich der Term $t_4 = \alpha\beta c$ kommt in beiden Funktionen vor.

Verkopplung von PLAs:

Annahme: $F : \mathbb{B}^n \to \mathbb{B}^m\ ,\ F = (f_1, \ldots, f_m)$

Gesucht: PLA mit $n + m$ Zeilen und p Spalten als Verkopplung der PLA$_i$, $1 \leq i \leq m$, der Booleschen Funktionen f_i. Dabei ist p gleich der Anzahl der verschiedenen Konjunktionsterme, die zur Bildung aller DNF D_{f_i} benötigt werden.

Die Verkopplung nicht in D_{f_i} benötigter Terme wird in der ODER-Ebene mittels Knoten vom Typ 0 realisiert (siehe Abb. 6.6).

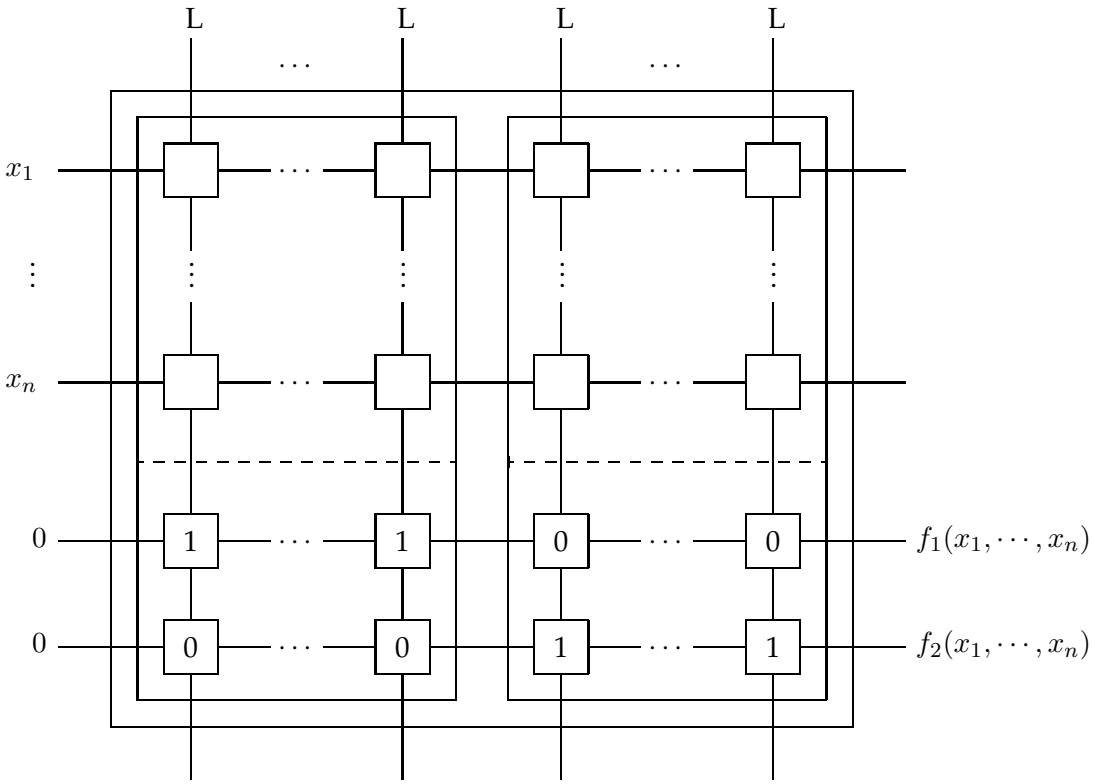


Abbildung 6.6: Verkoppelung von PLAs

Beispiel: Volladdierer

Der PLA des Volladdierers $\text{PLA}_{VA} = \text{PLA}_\gamma \cup \text{PLA}_u$ ergibt eine Matrix mit (3+2) Zeilen und 7 Spalten. (Abb. 6.7)

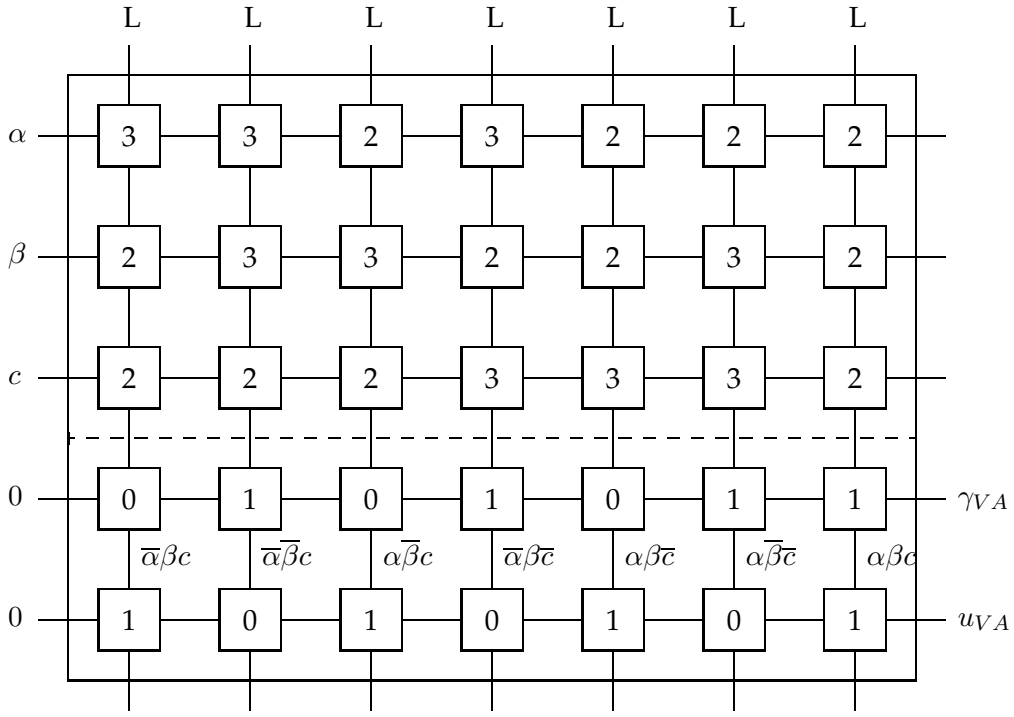


Abbildung 6.7: PLA für Volladdierer PLA_{VA}

Kapazität eines PLA:
Wieviele verschiedene Schaltfunktionen lassen sich mit einem PLA der Ausdehnung $(n+m) \times p$ realisieren? Diese Frage ist von Bedeutung, wenn PLAs als Chip gefertigt und programmierbar eingesetzt werden, oder wenn aus ihnen Speicher gefertigt werden. Eine Schaltfunktion $F : \mathbb{B}^n \to \mathbb{B}^m$ ist eine von $2^{m^{2^n}}$ möglichen Schaltfunktionen für derart spezifizierte Abbildungen. Eine n-stellige Eingabe erzeugt 2^n Minterme. Also wäre $p_{max} = 2^n$ eine obere Grenze für die Spaltenzahl einer PLA-Matrix und $(n+m)$ wäre die Anzahl der Zeilen. Ein PLA der Ausdehnung $(n+m) \times 2^n$ könnte alle $2^{m^{2^n}}$ Schaltfunktionen realisieren. In praktischen Fällen ist allerdings die Anzahl möglicher Konjunktionsterme stark zu beschränken.

Beispiel: PLA DM7575 (National Semiconductor, USA, 70er Jahre)

$n = 14, m = 8, p = 96$
UND-Ebene : 14·96 = 1344 Knoten
ODER-Ebene : 8·96 = 768 Knoten
} 2112 Knoten

$n = 14$ bedeutet $p_{max} = 2^{14} = 16384$ Minterme bzw. $2^{8^{2^{14}}} = 2^{131072}$ verschiedene Schaltfunktionen aber $p = 96$ bedeutet immer noch $2^{8^{96}} = 2^{768}$ verschiedene Schaltfunktionen und damit breite Anwendbarkeit.

Programmierung von PLAs:

Der Name Programmable Logic Array (PLA) weist darauf hin, daß die Belegung der Knoten mit Grundbausteinen der Typen 0 bis 3 programmierbar ist. Dies betrifft sowohl die UND-Matrix als auch die ODER-Matrix.

Die Programmierung kann hierbei auf zwei Arten erfolgen

a) hardwaremäßig: festes Einprägen (chemisch, photochemisch, ...) der Belegungsmatrix

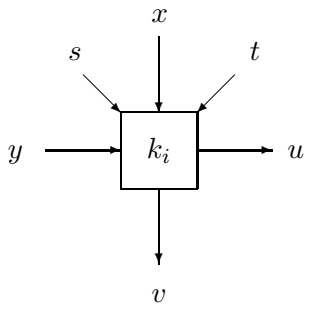
b) softwaremäßig: Mikroprogrammierung
Durch Auslesen von Festwertspeichern (ROM) werden Steuervariable zum Umschalten der Logik als zusätzliche Eingabedaten verwendet.

Aus den Kombinationen für festes/flexibles Programmieren der UND-/ODER-Matrix stellt sich heute folgendes Bild für den Einsatz der vorgestellten Prinzipien dar:

Logik-Baustein	UND-Matrix	ODER-Matrix	
ROM	fest	fest	Read Only Memory (Adreß-Dekodier- und Speichermatrix)
PROM, EPROM	fest	programmierbar	EPROM: Erasable Programmable ROM → mehrfach programmierbar; PROM: einmalig programmierbar
PAL	programmierbar	fest	Programmable Array Logic, $p \ll p_{max}$
PLA	programmierbar	programmierbar	wie auch bei PAL Rückkopplung Ausgang auf Eingang möglich

Der Begriff *Programmable Logic Device* (PLD) faßt alle diese Varianten zusammen.

Die Programmierung eines Knotens mit einer von 4 Grundfunktionen erfordert zwei programmierende Zuleitungen (s und t). Das heißt, das Knotenkonzept muß erweitert werden:



Baustein-Typ	s	t	v	u
k_0	0	0	x	y
k_1	0	L	x	$x \vee y$
k_2	L	0	$x \wedge y$	y
k_3	L	L	$x \wedge \overline{y}$	y

Die für $K = (n + m) \cdot p$ Knoten benötigten $2K$ binären Daten können in einem ROM gespeichert sein und von dort abgerufen werden.

Die Mikroprogrammierung eines hypothetischen Addierers eines von-Neumann-Rechners zeigt Abb. 6.8. Mit Hilfe der Steuersignale s und t wird das Rechenwerk zur Ausführung von Addition, Subtraktion, Multiplikation oder Division programmiert:

Signal	s	t	PLA-Aktion
0	0	0	Addieren
1	0	L	Subtrahieren
2	L	0	Multiplizieren
3	L	L	Dividieren

Dabei fällt auf, daß das PLA in ein Schaltwerk integriert ist. Als Speicherelemente dienen z.B. Register, die von der ODER-Matrix geladen werden und ihren Inhalt als Eingabe in die UND-Matrix einspeisen. Diese verzögerte und taktgesteuerte Rückkopplung der Ausgaben eines Schaltnetzes auf sich selbst ist ein charakteristisches Merkmal von Schaltwerken.

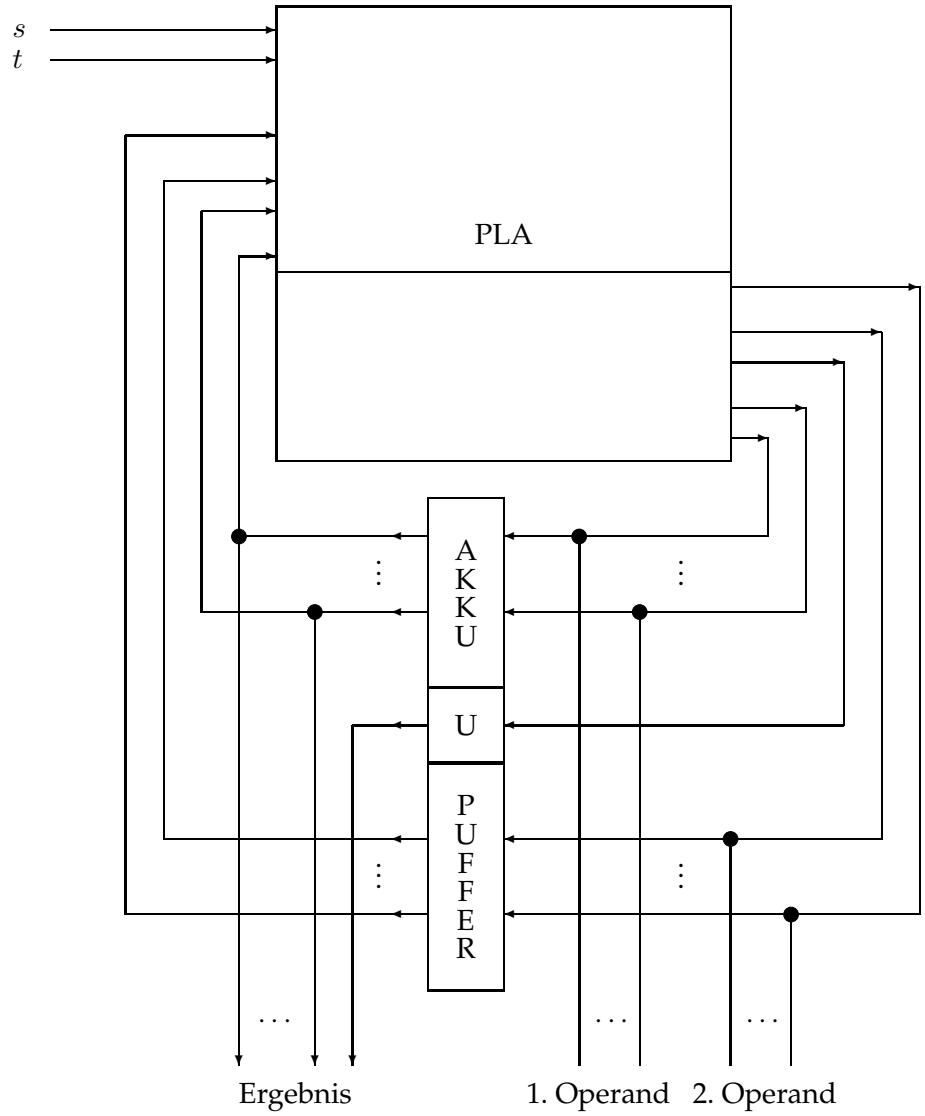


Abbildung 6.8: Hypothetische Mikroprogrammierung eines Addierers

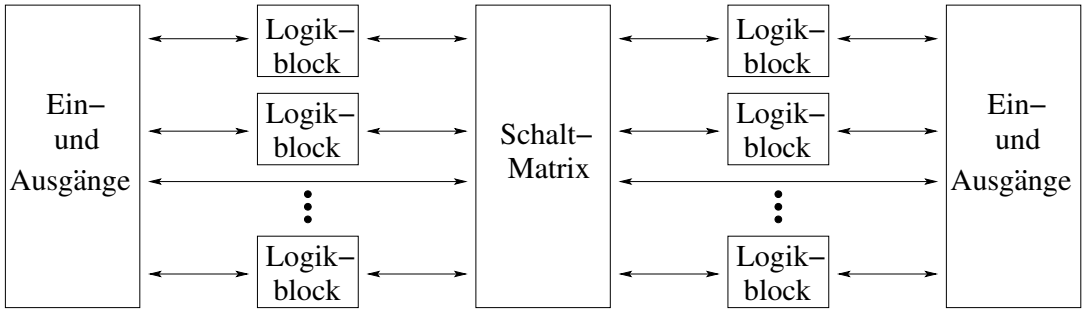


Abbildung 6.9: Struktur eines CPLDs

Komplexe Verschaltungen von PLDs in unterschiedlicher Form werden heute als Standard-Bausteine angeboten. Hierunter fallen

- komplexe PLDs (CPLDs)

- anwenderprogrammierbare Gate-Arrays (FPGAs).

CPLDs (siehe Abb. 6.9):

- PAL-ähnliche Teilstrukturen

- Schaltmatrix verbindet Logik-Blöcke und Ein-/Ausgänge

- jeder Logikblock realisiert komplexe Schaltfunktionen

- Programmierung mittels CMOS-Schalter

FPGAs – Field Programmable Gate Arrays (siehe Abb. 6.10)

- Zwischen den (einfachen) Logikblöcken befinden sich Verrahtungskanäle.

- Ein-/Ausgabe-Logik, Logikblöcke und Verdrahtung sind programmierbar.

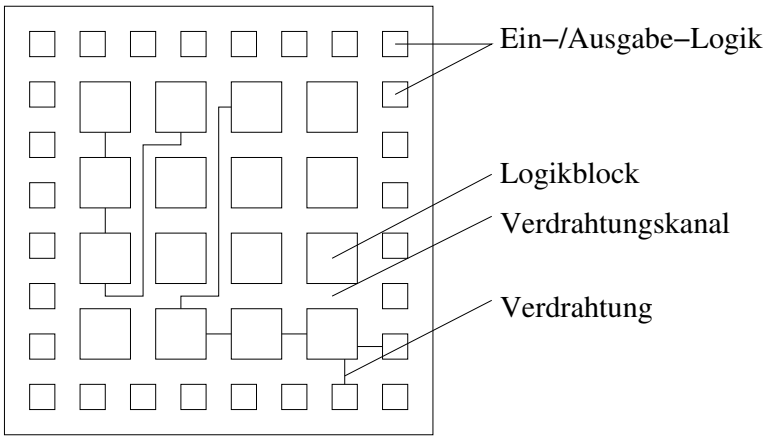


Abbildung 6.10: Struktur eines FPGAs

6.2 Speicherglieder

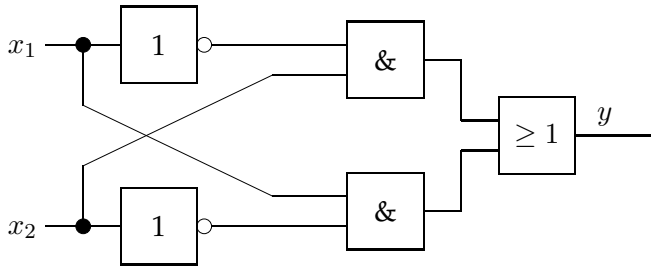
6.2.1 Schädliche und nützliche Zeiteffekte

Bisher wurde beim Entwurf oder bei der Analyse von Schaltfunktionen lediglich die aus der Verknüpfung von Schaltvariablen sich ergebende logische Funktionalität betrachtet. Dabei wurde vernachlässigt, daß Gatter physikalische Objekte sind,

- deren Zustände erst nach einer endlichen Dauer als stabil bezeichnet werden können und
- Zustandänderungen bei Veränderung der Belegung der Schaltvariablen auch nur in endlicher Zeit erfolgen.

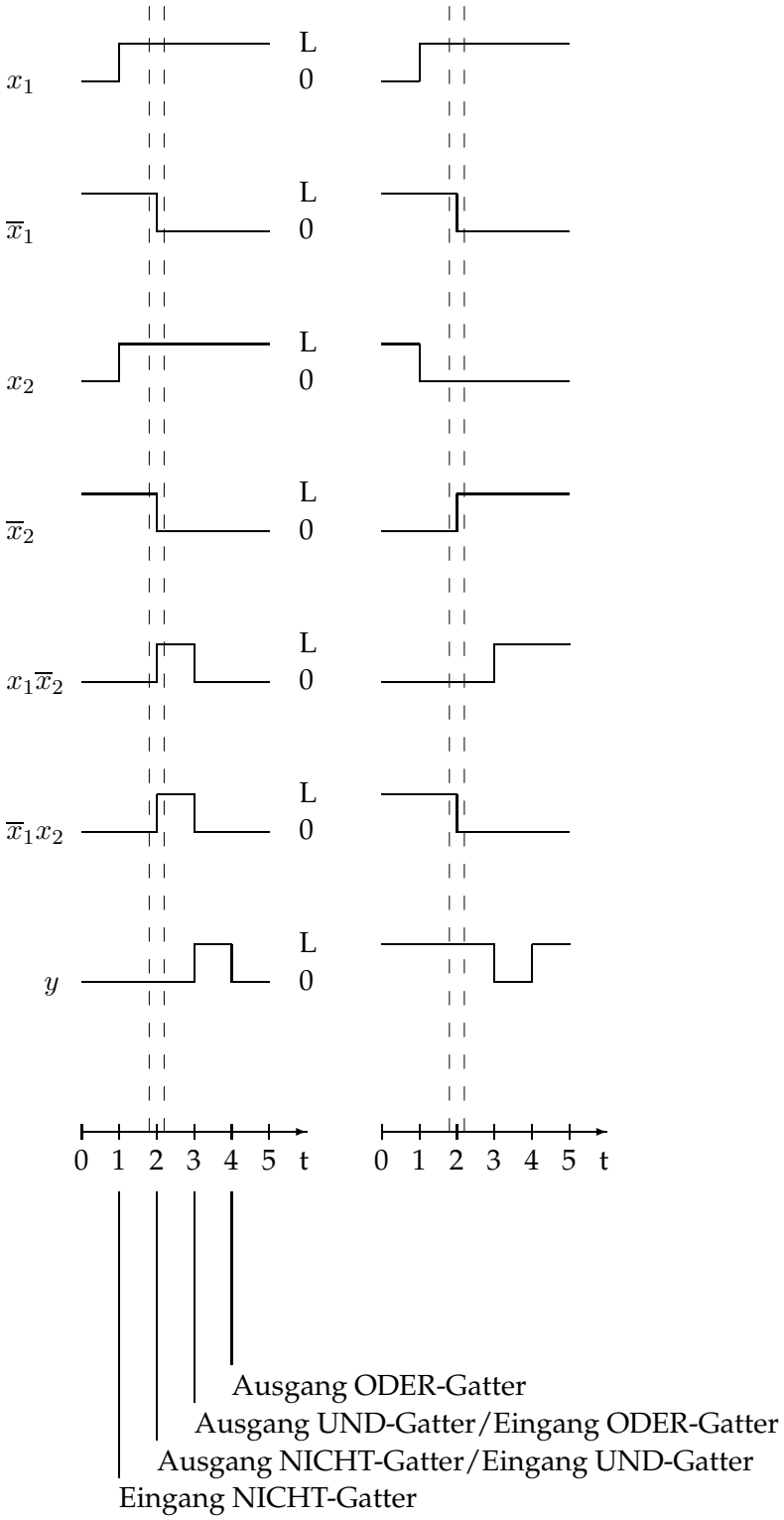
Die Vernachlässigung der Signallaufzeit und der Einschwingzeit bei realen Gattern kann zu Fehlern der realisierten Logik führen. Diese Synchronisationsfehler heißen *Hazards*.

Beispiel: Realisierung XOR (Antivalenz)



x_1	x_2	y	y_H
0	0	0	?
0	L	L	?
L	0	L	?
L	L	0	?

Die Negation der Variablen erfordert Zeit. Die Laufzeit der Signale wird vernachlässigt. Deshalb entsteht eine Desynchronisation an den UND-Gattern zwischen den Belegungen der Variablen.

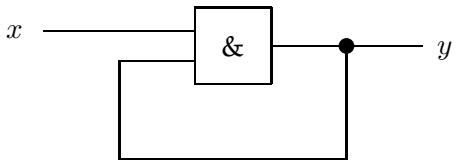

x_1
$\overline{x}_1$
x_2
$\overline{x}_2$
$x_1\overline{x}_2$
$\overline{x}_1x_2$
y
L
0
0 1 2 3 4 5 t
Ausgang ODER-Gatter
Ausgang UND-Gatter/Eingang ODER-Gatter
Ausgang NICHT-Gatter/Eingang UND-Gatter
Eingang NICHT-Gatter

links:
Übergang: $(x_1, x_2) = (0, 0) \rightarrow (x_1, x_2) = (\mathrm{L}, \mathrm{L})$
Soll: $y = 0 \rightarrow y = 0$
Ist: $y_H = \mathrm{L}$ für $(x_1, x_2) = (\mathrm{L}, \mathrm{L})$
da kurzzeitiges Anliegen von L für beide Eingänge der UND-Gatter. Dieses pflanzt sich fort mit Zeitverzögerung bis Ausgang ODER-Gatter

rechts:
Übergang: $(x_1, x_2) = (0, \mathrm{L}) \rightarrow (x_1, x_2) = (\mathrm{L}, 0)$
Soll: $y = \mathrm{L} \rightarrow y = \mathrm{L}$
Ist: $y_H = 0$ für $(x_1, x_2) = (\mathrm{L}, 0)$

Man nennt derartige Hazards statisch. Außerdem gibt es dynamische Hazards, die am Ausgang von Schaltnetzen entstehen können.
Laufzeiteffekte sind aber auch nützlich, weil sie rückgekoppelte Schaltnetze zu nutzen gestatten.
Annahme 1: rückgekoppeltes UND-Gatter

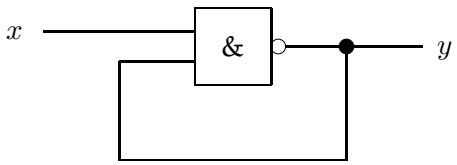
$$y = xy$$



x	y	y	
0	0	0	} $x = 0$: gilt immer $y = 0$
0	L	0	
L	0	0	} $x = \mathrm{L}$: zufällig eingestellter Wert für y bleibt erhalten
L	L	L	

Annahme 2: rückgekoppeltes NAND-Gatter (ideal)

$$y = \overline{xy} = \overline{x} \vee \overline{y}$$

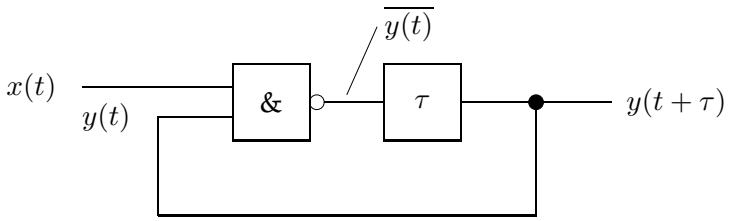


x	y	y	
0	0	L	} $\overline{x} = \mathrm{L}$: gilt immer $y = \mathrm{L}$
0	L	L	
L	0	L	} absurde Gleichung: $\overline{y} = y$!
L	L	0	

Annahme 3: rückgekoppeltes NAND-Gatter (real)
Berücksichtigung der Zeitverzögerung τ durch NAND-Gatter:

$$y(t+\tau) = \overline{x(t)} \vee \overline{y(t)}$$

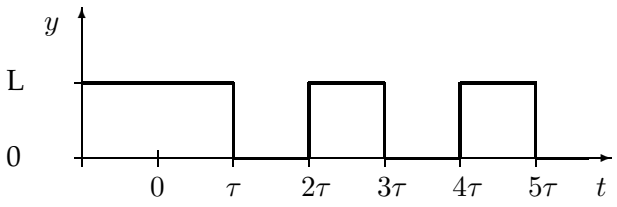
Je nach Schaltkreisfamilie liegt τ im Bereich 0.5 ... 5 Nanosekunden.



Das heißt, es wird ein ideales NAND-Gatter seriell gekoppelt mit einem Verzögerungsglied angenommen.
Daraus folgt für $x(t) = \mathrm{L}$: $y(t+\tau) = \overline{y(t)}$

t	0	τ	2τ	3τ	4τ	5τ
$x(t)$	L	L	L	L	L	L
$y(t)$	L	0	L	0	L	0
$y(t+\tau)$	0	L	0	L	0	L

Es entsteht eine Dauerschwingung! Dies ist ein instabiler Zustand, der für gleiche Eingaben zu unterschiedlichen Zeiten verschiedene Ausgaben erzeugt.



Rückgekoppelte Systeme können instabil sein, müssen es aber nicht. Insbesondere nutzt man in der Regelungstechnik die Rückkopplung der negativen Abweichung von Soll- und Istwert zur Stabilisierung von Prozessen.

6.2.2 Das RS-Flipflop

Ein Flipflop ist eine bistabile *Kippschaltung*, die sich dazu eignet, eine binäre Variable speichern zu können.

Definition: (Flipflop)

Ein Flipflop ist ein Speicherglied mit zwei stabilen Zuständen, das aus jedem der stabilen Zustände durch geeignete Ansteuerung in den anderen Zustand übergeht.

Die unterschiedliche Ansteuerung führt zu verschiedenen Flipflop-Typen.

Annahme 4: kreuzweise Rückkopplung von zwei NOR-Gattern

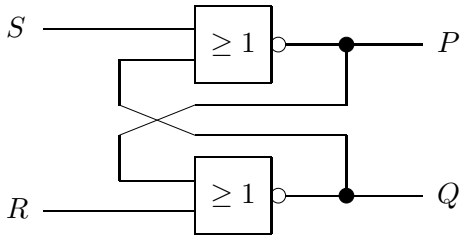
1 NOR-Gatter: $y(t+\tau) = \overline{x(t)}\ \overline{y(t)}$

$x(t) = 0 :\ y(t+\tau) = \overline{y(t)}$

2 NOR-Gatter:

a) $P(t+\tau) = \overline{S(t)}\ \overline{Q(t)}$

b) $Q(t+\tau) = \overline{R(t)}\ \overline{P(t)}$



Die Gleichungen der Ausgangsvariablen zeigen, daß sich beide gegenseitig bedingen. Sind beide Gatter mit identischem Zeitverhalten ausgestattet, liegt Symmetrie vor.

Aus der Definition eines NOR-Gatters folgt:

S	R	OR	NOR
0	0	0	L
0	L	L	0
L	0	L	0
L	L	L	0

Eine Eingabe mit $S = \mathrm{L}$ oder $R = \mathrm{L}$ erzeugt die Ausgabe Null.

Deshalb kann schrittweise auch die Abhängigkeit von P und Q von den Eingabedaten S und R abgeleitet werden.

1. $S(t) = \mathrm{L} \rightarrow \overline{S(t)} = 0 \;\Rightarrow\; P(t+\tau) = 0$ unabhängig von $Q(t)$
2. $R(t) = \mathrm{L} \rightarrow \overline{R(t)} = 0 \;\Rightarrow\; Q(t+\tau) = 0$ unabhängig von $P(t)$
3. $S(t) = 0\,,\; Q(t) = 0 \;\Rightarrow\; P(t+\tau) = \mathrm{L}$
4. $R(t) = 0\,,\; P(t) = 0 \;\Rightarrow\; Q(t+\tau) = \mathrm{L}$

Damit hat die Wertetabelle folgendes Aussehen:

Zustand	$S(t)$	$R(t)$	$P(t+\tau)$	$Q(t+\tau)$
0	0	0		
1	0	L	L(3)	0(2)
2	L	0	0(1)	L(4)
3	L	L	0(1)	0(2)

Die in Klammern gesetzten Zahlen beziehen sich auf obige Fallunterscheidungen.
Liegen S und R stationär an, so gilt auch die Tabelle für $t + n\tau$.
Im stationären Fall sind aber die Ausgänge für $S = R = 0$ unbestimmt:

5.a) $S(t) = 0\,,\; R(t) = 0\,,$ wenn $Q(t) = \mathrm{L} \;\Rightarrow\; P(t+\tau) = 0$

5.b) $S(t) = 0\,,\; R(t) = 0\,,$ wenn $P(t) = 0 \;\Rightarrow\; Q(t+\tau) = \mathrm{L}$

6.a) $S(t) = 0\,,\; R(t) = 0\,,$ wenn $Q(t) = 0 \;\Rightarrow\; P(t+\tau) = \mathrm{L}$

6.b) $S(t) = 0\,,\; R(t) = 0\,,$ wenn $P(t) = \mathrm{L} \;\Rightarrow\; Q(t+\tau) = 0$

Der Konflikt wird dadurch aufgelöst, daß der Übergang aus einem der beiden Zustände 1 oder 2 ($S = 0\,,\; R = \mathrm{L}$ oder $S = \mathrm{L}\,,\; R = 0$) in den Zustand 0 ($S = R = 0$) betrachtet wird:

7.) $2 \rightarrow 0$
$S(t) \stackrel{!}{=} 0\,,\; R(t) = 0\,,\; P(t) = 0\,,\; Q(t) = \mathrm{L}$

   a) liefert $P(t+\tau) = 0\,,\; Q(t+\tau) = \mathrm{L}$
   b) liefert $Q(t+\tau) = \mathrm{L}\,,\; P(t+\tau) = 0$

8.) $1 \rightarrow 0$
$S(t) = 0\,,\; R(t) \stackrel{!}{=} 0\,,\; P(t) = \mathrm{L}\,,\; Q(t) = 0$

   a) liefert $Q(t+\tau) = 0\,,\; P(t+\tau) = \mathrm{L}$
   b) liefert $P(t+\tau) = \mathrm{L}\,,\; Q(t+\tau) = 0$

Wir erkennen in den Fällen 7 und 8, daß beim Übergang zu $R = S = 0$ die Ausgangsvariablen des vorherigen Zustandes erhalten bleiben. Hierauf beruht die Verwendung des Flipflops als Speicherelement!

Die Belegungen $R = S = \mathrm{L}$ werden als unzulässig erklärt, weil sie dazu führen, daß $P = Q = 0$ wird. Das Fehlen einer komplementären Belegung dieser Variablen führt beim Übergang nach $R = S = 0$ zu nicht reproduzierbaren Zustandsübergängen.

Die Wertetabelle lautet nun:

Zustand	$S(t)$	$R(t)$	$P(t+\tau)$	$Q(t+\tau)$
0	0	0	(wie vorher: speichern)	
1	0	L	L	0
2	L	0	0	L
3	L	L	unzulässig	

Wir erkennen in der Tabelle, daß für die Zustände 1 und 2 gilt:

$$Q(t+\tau) = S(t)$$
$$P(t+\tau) = R(t).$$

Da sich offensichtlich P und Q zueinander komplementär verhalten, werden wir auf die Darstellung einer Ausgangsvariablen verzichten.

Im stationären Schaltungszustand lassen sich die Gleichungen a) $P(t+\tau) = \overline{S(t)}\ \overline{Q(t)}$ und b) $Q(t+\tau) = \overline{R(t)}\ \overline{P(t)}$ ineinander einsetzen, z.B.

$$Q = \overline{R}(S \vee Q) \qquad (*)$$

Daraus folgt:

a) für $Q = 0$: $0 = \overline{R}S \Leftrightarrow R = \mathrm{L}$ oder $S = 0$

b) für $Q = \mathrm{L}$: $\mathrm{L} = \overline{R} \Leftrightarrow R = 0$

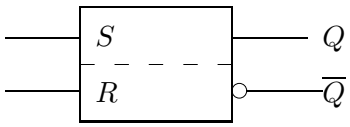
Aus der Gleichung $(*)$ folgt außerdem die Identität $Q = Q$ für $\overline{R} = \mathrm{L}$ (d.h. $R = 0$) und $S = 0$. Dies ist gerade die Speicherwirkung des Flipflop. Wird anstelle $Q(t+\tau)$ der Ausdruck Q^+ für den Folgezustand von Q verwendet, erhält man die Zustandfolge in der Form

Zustand	S	R	Q^+	$\overline{Q}^+$	
0	0	0	Q	$\overline{Q}$	Speicherzustand
1	0	L	0	L	Rücksetzzustand
2	L	0	L	0	Setzzustand
3	L	L	–	–	unzulässig

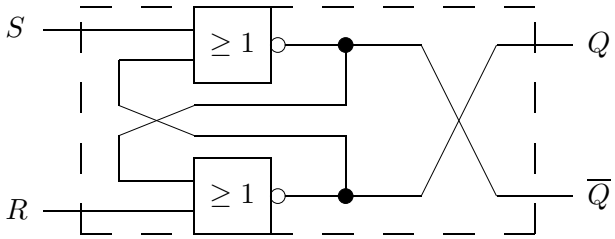
Die *Zustandsfolgetabelle* für Schaltwerke entspricht der Wertetabelle für Schaltnetze.

Das Verhalten des Flipflop wird also durch eine Ausgangsvariable hinreichend erfaßt. Die Eingangsvariable S heißt Setz-Variable, weil sie $Q = \mathrm{L}$ erzeugt. Die Variable R heißt Rücksetz-Variable, weil sie $Q = 0$ erzeugt.

Nach der Norm DIN 40700/93 wird aber noch die Variable $\overline{Q}$ eingeführt als Ausgangsvariable der Rücksetz-Eingangsvariablen. Der Name ist nun *RS-Flipflop*. Schaltzeichen des RS-Flipflops:



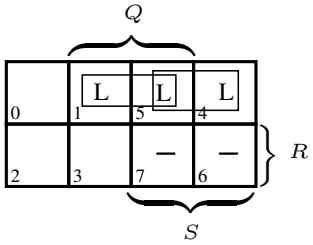
Schaltbild des RS-Flipflops:



Die *erweiterte Zustandsfolgetabelle* des RS-Flipflops mit 3 Variablen S, R, Q lautet:

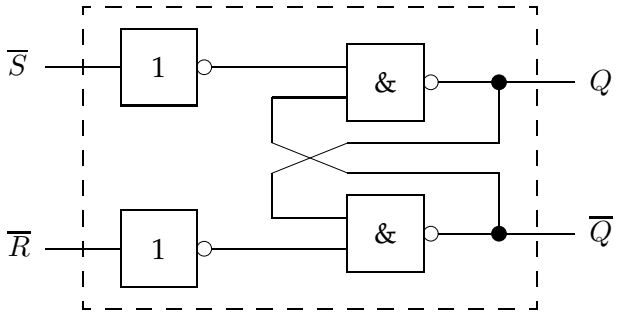
Zustand	S	R	Q	Q^+	Funktion (Zustand)
0	0	0	0	0	speichern (0)
1	0	0	L	L	speichern (0)
2	0	L	0	0	rücksetzen (1)
3	0	L	L	0	rücksetzen (1)
4	L	0	0	L	setzen (2)
5	L	0	L	L	setzen (2)
6	L	L	0	–	unzulässig
7	L	L	L	–	unzulässig

KV-Diagramm der Funktion $Q^+(S, R, Q)$:



DNF(a) $Q^+ = \overline{R}Q \vee \overline{R}S$; $SR = 0$
bzw.
DNF(b) $Q^+ = \overline{R}(S \vee Q)$; $SR = 0$

Das RS-Flipflop kann auch aus zwei NAND-Gattern gebildet werden. Seine Schaltfunktion bezeichnet man als *negative Logik*. Die NAND-Kippschaltung heißt auch *latch*.
Schaltbild:



6.2.3 Varianten des RS-Flipflop

Es gibt eine Palette von Varianten des RS-Flipflops, die vorwiegend technologisch begründet sind. Als Kriterien zur Unterscheidung der Varianten können betrachtet werden:

a) Spezifik der Wirkung der Eingangsvariablen auf den Ausgangszustand

b) Wirkungsweise des Taktsignals

Wenn im folgenden die Nutzung der Kippglieder in komplexeren Schaltwerken als elementare Speicherglieder diskutiert werden soll, muß Bezug genommen werden auf konkrete Realisierungen der Kippglieder.

Bezüglich der Wirkung der Eingangsvariablen auf die Ausgangsvariable unterscheidet man folgende Flipflops (FF):

- RS-FF
- D-FF
- T-FF
- JK-FF

Nach der Wirkung des Taktsignals unterscheidet man folgende Prinzipien:

- nicht taktgesteuert
- einfach taktgesteuert
  - taktzustandsgesteuert
  - taktflankengesteuert
- zweifach taktgesteuert (Master-Slave-Prinzip)
  - taktzustandsgesteuert
  - taktflankengesteuert

Wir werden als nützliches Beispiel das zweifach taktzustandsgesteuerte Verzögerungs-FF bzw. *D-MS-FF* betrachten.

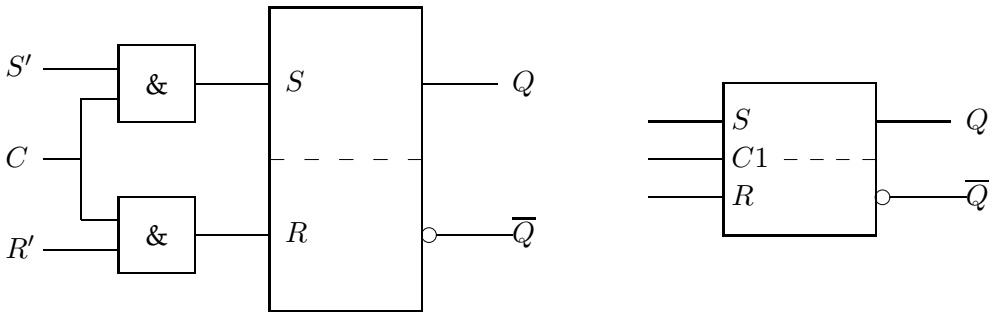
Beim RS-FF wird der anliegende Eingangszustand sofort wirksam. Das führt zu Problemen, wenn mehrere FF zusammengeschaltet werden. Aber auch die physikalische Individualität der eingesetzten Gatter führt wegen unterschiedlichen Zeitverhaltens nur in Ausnahmefällen zu deterministischem Kippverhalten.

Die Lösung des Problems liegt in der Taktsteuerung, also der Verwendung eines zusätzlichen Taktsignals C. Während die Eingangsvariablen R' und S' den Ausgangszustand des Flipflop bestimmen, bestimmt das Steuersignal C den Zeitpunkt des Wirksamwerdens der Eingangsvariablen.

aktiver Taktzustand: $(C = \mathrm{L}) \wedge (R', S')$

Setzen des Flipflops:	$S' = \mathrm{L}$
Rücksetzen des Flipflops:	$R' = \mathrm{L}$

passiver Taktzustand: $(C = 0) \wedge (R', S')$

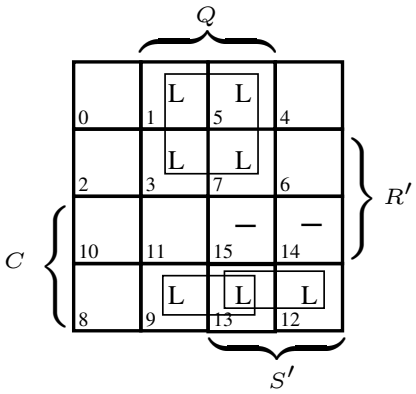


Damit erhält man folgende Zustandsfolgetabelle:

	C	S'	R'	Q^+	
passiver	0	0	0	Q	
Zustand	0	0	L	Q	keine Änderung des
	0	L	0	Q	Ausgangszustandes,
	0	L	L	Q	d.h. Speichern
aktiver	L	0	0	Q	Speichern
Zustand	L	0	L	0	Rücksetzen
	L	L	0	L	Setzen
	L	L	L	-	unzulässig

Die erweiterte Zustandsfolgetabelle und das KV-Diagramm haben folgende Gestalt:

	C	S'	R'	Q	Q^+
0	0	0	0	0	0
1	0	0	0	L	L
2	0	0	L	0	0
3	0	0	L	L	L
4	0	L	0	0	0
5	0	L	0	L	L
6	0	L	L	0	0
7	0	L	L	L	L
8	L	0	0	0	0
9	L	0	0	L	L
10	L	0	L	0	0
11	L	0	L	L	0
12	L	L	0	0	L
13	L	L	0	L	L
14	L	L	L	0	–
15	L	L	L	L	–



Für $C = \mathrm{L}$ muß das getaktete RS-Flipflop die gleiche Logik ergeben wie das nicht getaktete. Aus der erweiterten Zustandsfolgetabelle ergibt sich

$$Q^+ = QC\overline{R'} \vee CS'\overline{R'}$$

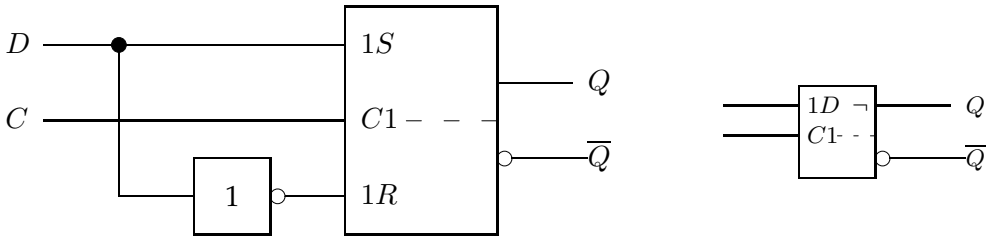
und wegen $CR' = R$ bzw. $CS' = S$

$$Q^+ = (S \vee Q)\overline{R}.$$

Den Nachteil des undefinierten Zustandes für $R' = S' = \mathrm{L}$ vermeidet das *D-Flipflop*. Dabei wird die Komplementarität von R' und S' im getakteten Zustand dadurch erzwungen, daß es nur ein Eingangssignal

$$D = S'$$

gibt.



Für $D = \mathrm{L}$ wird $S' = \mathrm{L}$ und $R' = 0$ und damit $Q^+ = \mathrm{L}$ (das Flipflop wird gesetzt). Für $D = 0$ wird $S' = 0$ und $R' = \mathrm{L}$ und damit $Q^+ = 0$ (das Flipflop wird zurückgesetzt). Beides gilt nur im aktiven Taktzustand $C = \mathrm{L}$.

Deswegen gilt die einfache Zustandsübertragungsfunktion

$$Q^+ = D.$$

Das D-FF mit Zustandssteuerung übernimmt um einen Takt verzögert den Wert der Eingangsvariablen D. Deshalb der Name D-FF von Delay.

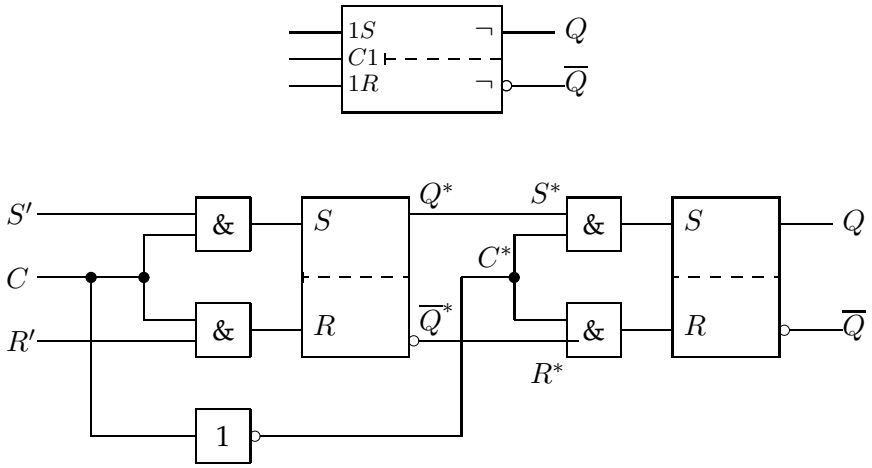
C	D	S	R	Q^+
0	0	0	0	Q
0	L	0	0	Q
L	0	0	L	0
L	L	L	0	L

Das D-FF mit Taktsteuerung funktioniert für $C = \mathrm{L}$ nur dann richtig, wenn während der gesamten aktiven Phase $D = const$ gilt. Schaltet man sequentiell mehrere solche D-FF hintereinander, kann dies nicht garantiert werden (z.B. bei Registern oder Zählern).

Bei solchen Anordnungen möchte man Eingänge und Ausgänge der einzelnen Flipflops gezielt separat takten.

Die Zwei-Zustandsteuerung (*Master-Slave-Prinzip*) bewirkt eine Art Zwischenspeicherung, so daß am Ausgang Daten ausgelesen werden können und unabhängig hiervon am Eingang ein neues Datum eingelesen werden kann.

Beim MS-FF werden zwei FF hintereinander geschaltet. Das hintere FF wird mit dem invertierten Takt des vorderen angesteuert. Das Schaltbild zeigt ein RS-MS-FF.



Es gilt

$$C = \mathrm{L} \quad : \quad \text{Master aktiv, Slave passiv}$$
$$C = 0 \quad : \quad \text{Master passiv, Slave aktiv.}$$

Bei Verwendung von D-FF als D-MS-FF kann nur der Master ein D-FF sein, während der Slave ein RS-FF ist.

6.3 Schaltwerke als Automaten

6.3.1 Mealy- und Moore-Automaten

Ein Automat ist eine Modellmaschine zur Beschreibung eines Systems. Ein Automat reagiert auf eine Eingabe mit einer Ausgabe, die von der Eingabe und vom momentanen Zustand des Systems abhängt. Zusätzlich zur Ausgabe eines Symbols kann der Automat eine Zustandsänderung vollführen.

Definition: (deterministischer endlicher Automat (DEA))

Ein 5-Tupel $A = (X, Y, S, G, H)$ ist die Darstellung eines deterministischen endlichen Automatens, falls

$$\begin{aligned} X &= \{x_0, x_1, \ldots, x_{N-1}\} \text{ eine Menge von Eingabesymbolen} \\ Y &= \{y_0, y_1, \ldots, x_{M-1}\} \text{ eine Menge von Ausgabesymbolen} \\ S &= \{s_0, s_1, \ldots, s_{L-1}\} \text{ eine Menge von Zuständen} \\ G &: X \times S \to Y,\ (x_i, s_j) \to y_k, \text{ die Ausgabefunktion} \\ H &: X \times S \to S,\ (x_i, s_j) \to s_k, \text{ die Zustandsübergangsfunktion} \end{aligned}$$

sind.

Endliche Automaten zeichnen sich durch einen Vorrat von endlich vielen verschiedenen Zuständen aus. Ist andernfalls die Menge möglicher Zustände nicht endlich, spricht man von *unendlichen Automaten* bzw. von einer *sequentiellen Maschine*.

Jeder endliche Automat läßt sich in ein Schaltwerk umsetzen, wenn

$$x_i \in \mathbb{B}^n,\ \ y_j \in \mathbb{B}^m,\ \ s_k \in \mathbb{B}^l\ \ \forall i, j, k, l, m, n \in \mathbb{N}.$$

Dann ist $N \leq 2^n,\ M \leq 2^m, L \leq 2^l$.

Im Falle $m = 1$ ist die Ausgabefunktion g.

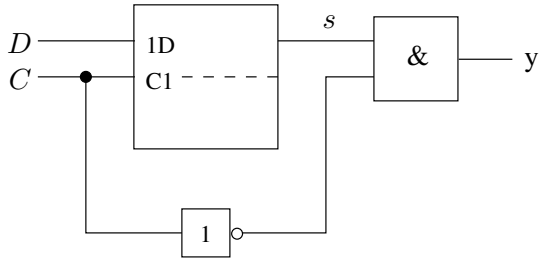
Beispiel: getaktetes D-Flipflop als deterministischer Automat

$$x = (C, D) \in \mathbb{B}^2 \quad , N = 2^2 = 4$$
$$y \in \mathbb{B} \quad , M = 2$$
$$s \in \mathbb{B} \quad , L = 2$$

C	D	S	R	s^+	y
0	0	0	0	s	s
0	L	0	0	s	s
L	0	0	L	0	0
L	L	L	0	L	0

$$h: \; s^+ = \begin{cases} D & \text{falls} \quad C = L \\ s & \text{falls} \quad C = 0 \end{cases}$$

$$g: \; y = s\overline{C}$$



$C =$ L: Speichern des Wertes von D
$C = 0$: Ausgeben des gespeicherten Wertes

Eine Verallgemeinerung der deterministischen endlichen Automaten stellen *nichtdeterministische endliche Automaten (NEA)* dar. Hierbei ist für einen Zustand s_i mittels eines Eingabesymboles x_j der Übergang in einen von mehreren Zuständen möglich. Die Zustandsübergänge werden statt durch eine Übertragungsfunktion durch eine *Übertragungsrelation*

$$H \subseteq S \times X \times S$$

gegeben. Dabei besagt ein Tripel $(s_i, x_j, s_k) \in H$, daß vom Zustand s_i durch Lesen des Symbols x_j ein Übergang in den Zustand s_k möglich ist. Der Nichtdeterminismus liegt darin begründet, daß der Automat bei Zustandsübergängen ggf. zwischen mehreren Zuständen "wählen" kann.

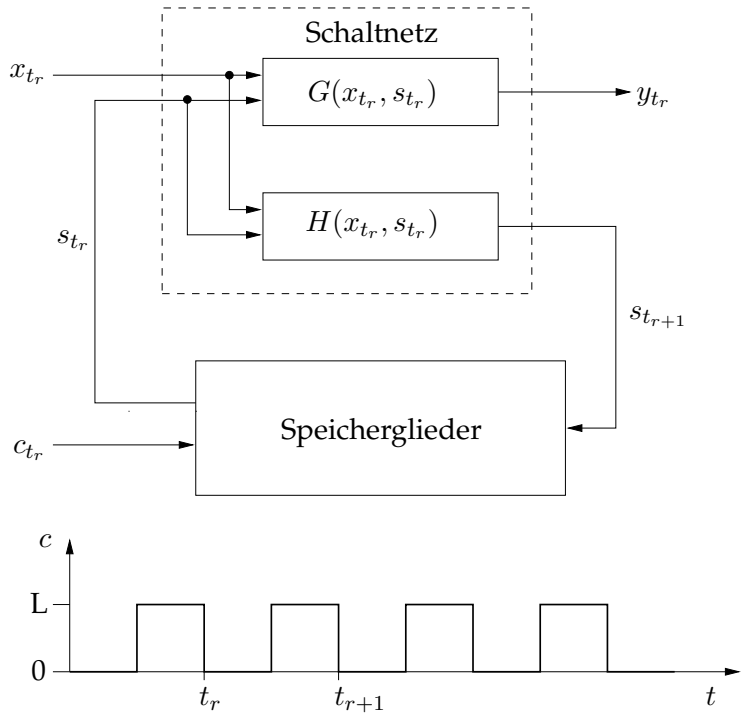


Abbildung 6.11: Mealy-Automat

Beispiel: getaktetes SR-Flipflop als nichtdeterministischer Automat

Als Beispiel für einen nichtdeterministischen Automaten sei ein getaktetes RS-Flipflop genannt. Für die Eingangskombination (C, S, R)=(L,L,L) besteht ein Konflikt bezüglich des Nachfolgezustands:

$$s^+ = \begin{cases} (S, R) & \text{falls} \quad (C = L) \wedge (S = \overline{R}) \\ (P, Q) & \text{falls} \quad ((C = 0) \wedge (P = \overline{Q})) \vee ((C = \mathrm{L}) \wedge (S = R = 0)) \\ s' \in \{(0, L), (L, 0)\} & \text{falls} \quad (C, S, R) = (L, L, L) \end{cases}$$

Ein deterministischer endlicher Automat gemäß obiger Definition ist ein *Mealy-Automat*.

Eine mögliche technische Realisierung (s. Abb 6.11) bildet die Zustandsübergangs- und Ausgabefunktion durch ein Schaltnetz ab, der Zustand des Systems wird in Speichergliedern festgehalten.

Während der zum Takt t_r gebildete Ausgabewert y_{t_r} zur Weiterverarbeitung zur Verfügung steht, wird der Folgezustand $s_{t_{r+1}}$ erst im Takt $r + 1$ im Schaltnetz als aktueller

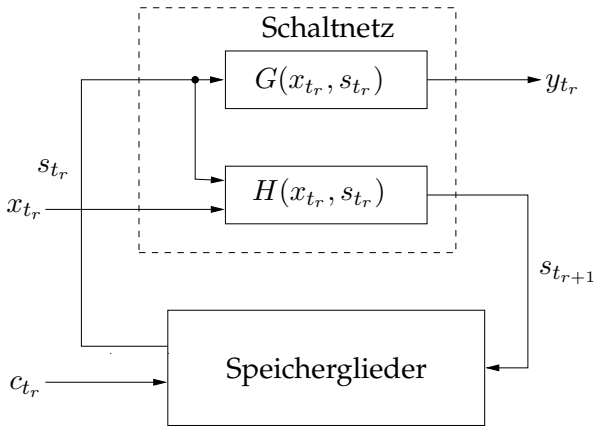


Abbildung 6.12: Moore-Automat

Zustand wirksam. Der Zustandsübergang vom Zustand s_{t_r} zum Zustand $s_{t_{r+1}}$ findet mit der fallenden Taktflanke statt (Abb. 6.11).
Der Mealy-Automat wird also vollständig durch die Schaltfunktionen

$$\begin{aligned} y_{t_r} &= G(x_{t_r}, s_{t_r}) \\ s_{t_{r+1}} &= H(x_{t_r}, s_{t_r}) \end{aligned}$$

beschrieben, wenn der Anfangszustand $s_{t_0} \in S$ wohl definiert ist.
Ein Sonderfall des Mealy-Automaten ist der *Moore-Automat* (s. Abb 6.12) für den Fall $G : S \to Y$. Die Ausgabefunktion ist also lediglich vom momentanen Zustand, nicht jedoch vom gelesenen Eingabesymbol abhängig, weshalb Moore-Automaten auch als zustandsorientierte Schaltwerke bezeichnet werden.
Zu jedem Mealy-Automaten läßt sich ein Moore-Automat und zu jedem Moore-Automaten läßt sich ein Mealy-Automat mit gleichem Ein-/Ausgabeverhalten konstruieren.
Für den Moore-Automaten gilt

$$\begin{aligned} y_{t_r} &= G(s_{t_r}) \\ s_{t_{r+1}} &= H(x_{t_r}, s_{t_r}). \end{aligned}$$

Da zum Zeitpunkt t_{r+1} der durch die Schaltfunktion $G(s_{t_{r+1}})$ gebildete Ausgabevektor $y_{t_{r+1}}$ gehört, folgt

$$y_{t_{r+1}} = G(H(x_{t_r}, s_{t_r})).$$

Schaltwerke, die außer dem Taktsignal kein Eingabesignal erfordern, heißen *autonome Automaten*.

Beispiele für Moore-Automaten sind synchrone Zähler und Register. Aber auch das D-FF ist ein Moore-Automat ($g : y = s\overline{C}$), da seine Ausgabe nicht von D abhängt.

Aus dem D-FF lassen sich *Schieberegister* für l Bit konstruieren, die selbst wieder einen Moore-Automaten bilden. Jede Komponente des Zustandsvektors $s = (s_0, \ldots, s_{l-1})$ wird durch ein D-FF verwaltet.

Die funktionelle Beschreibung von Automaten kann erfolgen durch

- Zustandsübergangstabellen
- Übergangsmatrizen
- KV-Diagramme
- Zustandsgraphen.

Eine Zustandsübergangs- oder Zustandsfolgetabelle hat folgenden Aufbau:

Eingangsvariable		Ausgangsvariable	
$s_{l-1} \cdots s_0$	$x_{n-1} \cdots x_0$	$s^+_{l-1} \ldots s^+_0$	$y_{m-1} \ldots y_0$

Wir verwendeten diese Tafel bereits als "erweiterte Wertetafel" bei der Behandlung der Flipflops.

Übergangsmatrizen haben folgenden Aufbau:

		Nachfolgezustand			
		s^+_0	s^+_1	$\cdots$	s^+_{l-1}
Zustand	s_0		$\vdots$		
	s_1	$\cdots$	(x/y)		
	$\vdots$				
	s_{l-1}				

Zustandsgraphen $\mathcal{G} = (K, E)$ bestehen aus einer Menge von Knoten K und Kanten E, wobei die Knoten den Zuständen S entsprechen und die gewichteten Kanten die durch die Eingaben $x \in X$ induzierten Zustandsübergänge anzeigen. Bei n-stelligen Eingaben $x_i \in X$ können demzufolge $N \leq 2^n$ Kanten von jedem Knoten ausgehen. Die Kanten werden beim Mealy-Automaten mit den die Übergänge erzeugenden Eingaben und den durch diese Übergänge erzeugten Ausgaben notiert. Beim Moore-Automaten sind die Zustandsübergänge eingabeunabhängig und alle von einem Knoten ausgehenden Kanten entsprechen den gleichen Ausgaben. Deshalb werden hier die Ausgaben direkt am Knoten markiert. Abbildung 6.13 zeigt den Zustandsgraphen eines Mealy-Automaten.

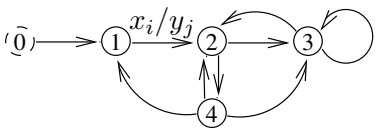


Abbildung 6.13: Zustandsgraph eines Mealy-Automaten

6.3.2 Register als Schaltwerke

Register sind selbst Schaltwerke und stellen durch ihre speichernde Eigenschaft auch komplexe Speicherglieder für komplexere Schaltwerke dar. In Rechenwerken dienen sie zur Zwischenspeicherung von

- Operanden
- Operatoren
- Adressen.

In der Speichereinheit dienen sie zur Kommunikation mit

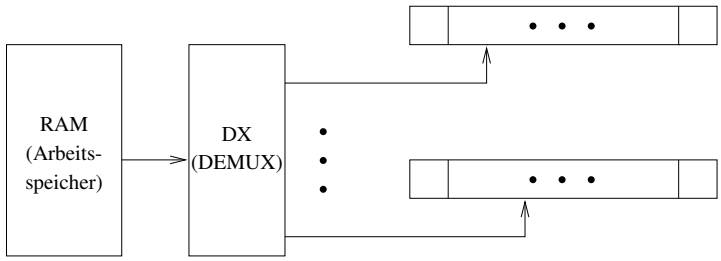
- dem Adreßbus (MAR)
- dem Datenbus (MDR)

Gewöhnlich wird die Kommunikation

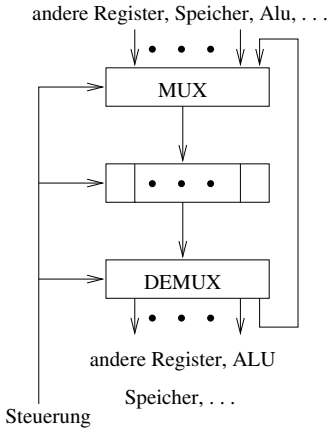
- zwischen den Registern
- zwischen Speichereinheit und Registern
- zwischen Registern und Recheneinheit

durch die Einbettung von Registern in ein komplexes Netz von Multiplexern (Zuführung der Daten von unterschiedlichen Quellen) bzw. Demultiplexern (Weitergabe der Daten an unterschieldliche Ziele) realisiert.

Beispiel: Laden von Registern mit Inhalten des Arbeitsspeichers



Beispiel: Einbettung von Registern in Netzwerk aus MX und DX



Das Prinzip des *Schieberegisters* besteht darin, daß Daten taktgesteuert durch eine Folge von Flipflop-Elementen in eine vorgegebene Richtung geschoben werden (s. Abb. 6.14). Ein Schieberegister ist ein *synchrones Schaltwerk*, das taktgesteuert eine Eingabe mehrstelliger Binärworte aufnehmen, speichern und weitergeben kann. Aufnahme und Weitergabe erfolgen seriell oder parallel. Dabei speichert jedes Speicherglied 1 Bit. Mit jedem Takt wird der Inhalt in ein benachbartes Speicherglied geschoben.

Schieberegister sind Moore-Automaten. Mit ihrer Hilfe können aber auch Mealy-Automaten (s. Abb. 6.15) konstruiert werden.

Sei ein Schieberegister für das Aufnehmen, das Speichern und Weitergeben von 4-Bit-Worten $x, y \in \mathbb{B}^4$ entworfen. Dann erfordert dies einen 4-Vektor $s = (s_0, s_1, s_2, s_3) \in \mathbb{B}^4$ von Zuständen, die durch 4 D-FF realisiert werden. Wir nehmen an, daß die x und y

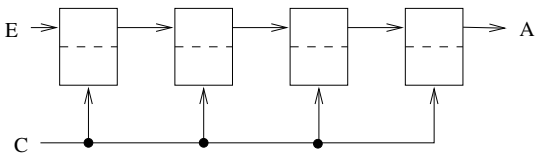


Abbildung 6.14: Schieberegister

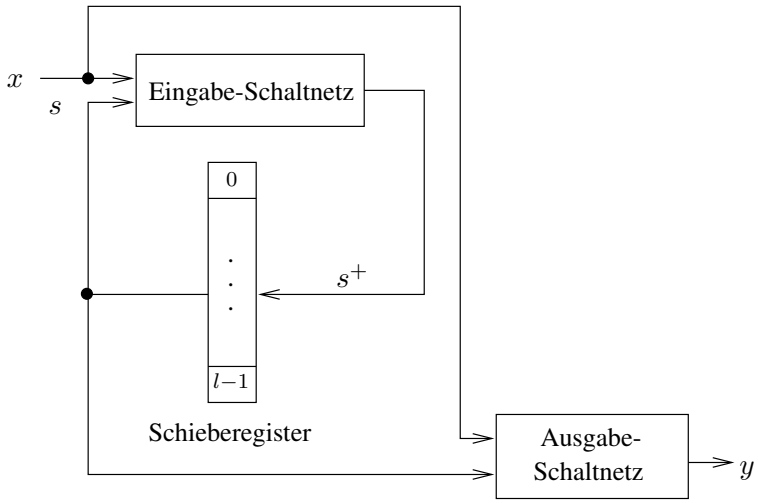


Abbildung 6.15: Konstruktion eines Mealy-Automaten mit Schieberegister

seriell als 1-dim. Daten angeboten bzw. weitergegeben werden. Das in Abb. 6.16 dargestellte Schaltwerk leistet diese Aufgabe. Am Beispiel des Datums $x = (\mathrm{LL0L})$, das in der Folge $x_{t_0} = \mathrm{L}$, $x_{t_1} = \mathrm{L}$, $x_{t_2} = 0$, $x_{t_3} = \mathrm{L}$ eingegeben wird, sei die Wertetabelle (s. Tab. 6.1) angegeben (Annahme: Rechtsschieberegister).

Der Schaltnetzentwurf beruht auf den allgemeinen Beziehungen, die durch die Übertragungsfunktionen

$$\begin{aligned} s_0^+ &= x \\ s_1^+ &= s_0 \\ s_2^+ &= s_1 \\ s_3^+ &= s_2 \end{aligned}$$

und durch die Ausgabefunktion

$$y = s_3$$

gegeben sind.

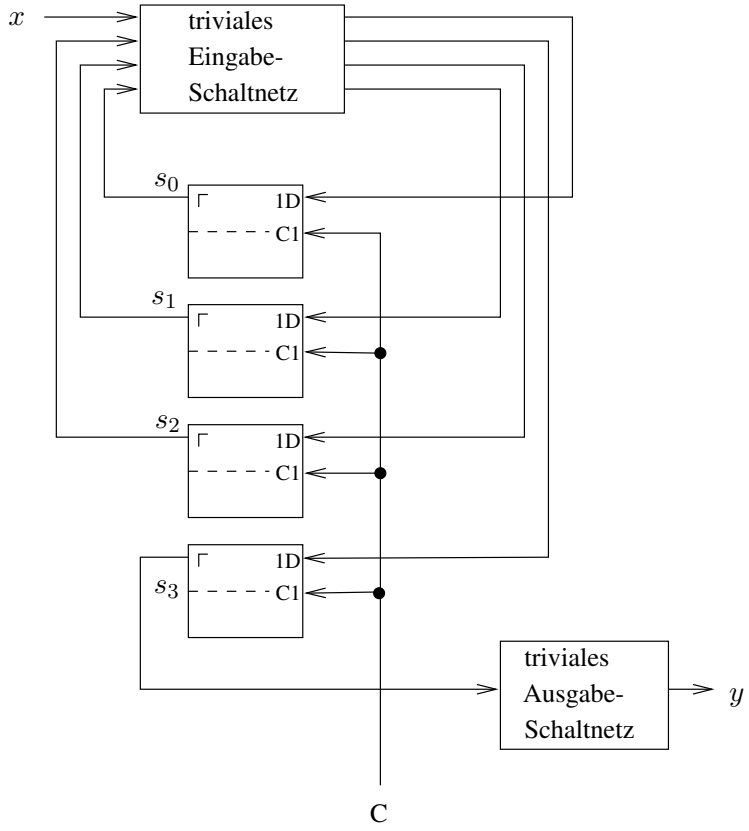


Abbildung 6.16: 4-Bit Schieberegister

Damit erhält man als Grundlage eines systematischen Schaltungentwurfes die Zustandsfolgetabelle (s. Tab. 6.2)

Schieberegister sind universelle Schaltwerke, die sich nicht nur zur taktgesteuerten Datenspeicherung/-weitergabe eignen, sondern auch als *Parallel-Seriell-Wandler* oder *Seriell-Parallel-Wandler* von Daten.

Setzt man Multiplexer als Datenwegschalter ein, dann sind in einer Schaltung alle folgenden Funktionen realisierbar:

a) seriell laden, seriell ausgeben

b) seriell laden, parallel ausgeben

c) parallel laden, seriell ausgeben

d) parallel laden, parallel ausgeben.

Takt n		Zustände nach Takt n				
	x	s_0	s_1	s_2	s_3	y
0	L	0	0	0	0	0
1	L	L	0	0	0	0
2	0	L	L	0	0	0
3	L	0	L	L	0	0
4	0	L	0	L	L	L
5	0	0	L	0	L	L
6	0	0	0	L	0	0
7	0	0	0	0	L	L

Tabelle 6.1: Wertetabelle 4-Bit Schieberegister

In der Abbildung 6.17 sind 4 D-FF mit 4 Multiplexern ergänzt. Jeder Multiplexer kann über zwei Steuersignale C_0, C_1 vier Dateneingänge auf einen Ausgang umschalten.

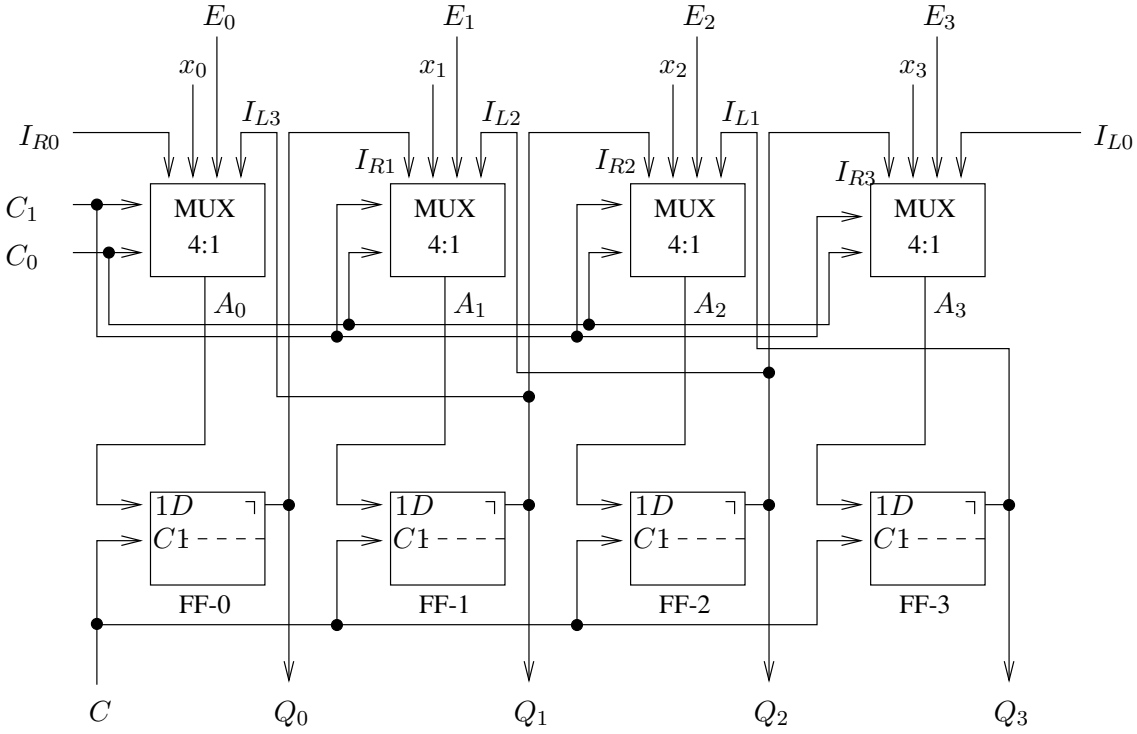


Abbildung 6.17: Rechts-/Linksschieberegister

x	s_3	s_2	s_1	s_0	s_3^+	s_2^+	s_1^+	s_0^+	y
0	0	0	0	0	0	0	0	0	0
0	0	0	0	L	0	0	L	0	0
0	0	0	L	0	0	L	0	0	0
0	0	0	L	L	0	L	L	0	0
0	0	L	0	0	L	0	0	0	0
0	0	L	0	L	L	0	L	0	0
0	0	L	L	0	L	L	0	0	0
0	0	L	L	L	L	L	L	0	0
0	L	0	0	0	0	0	0	0	L
0	L	0	0	L	0	0	L	0	L
0	L	0	L	0	0	L	0	0	L
0	L	0	L	L	0	L	L	0	L
0	L	L	0	0	L	0	0	0	L
0	L	L	0	L	L	0	L	0	L
0	L	L	L	0	L	L	0	0	L
0	L	L	L	L	L	L	L	0	L
L	0	0	0	0	0	0	0	L	0
L	0	0	0	L	0	0	L	L	0
L	0	0	L	0	0	L	0	L	0
L	0	0	L	L	0	L	L	L	0
L	0	L	0	0	L	0	0	L	0
L	0	L	0	L	L	0	L	L	0
L	0	L	L	0	L	L	0	L	0
L	0	L	L	L	L	L	L	L	0
L	L	0	0	0	0	0	0	L	L
L	L	0	0	L	0	0	L	L	L
L	L	0	L	0	0	L	0	L	L
L	L	0	L	L	0	L	L	L	L
L	L	L	0	0	L	0	0	L	L
L	L	L	0	L	L	0	L	L	L
L	L	L	L	0	L	L	0	L	L
L	L	L	L	L	L	L	L	L	L

Tabelle 6.2: Zustandsfolgetabelle

Dabei steuern die Kontrollsignale folgende Funktionen:

$(C_1, C_0) = (0, L)$:

Die Eingänge $(x_0, \ldots, x_3) = (0, \ldots, 0)$ werden zum Löschen des Registers durch-

gestellt. Mit dem folgenden Takt gilt $(Q_0, \ldots, Q_3) = (0, \ldots, 0)$.

$(C_1, C_0) = (0, 0)$:

Die Eingänge $I_{R0}, \ldots, I_{R3}$ werden auf die Ausgänge $A_0, \ldots, A_3$ durchgestellt. Damit arbeitet das Register als Rechtsschieberegister (FF-0 $\rightarrow$ FF-3).

$(C_1, C_0) = (L, L)$:

Die Eingänge $I_{L0}, \ldots, I_{L3}$ werden durchgestellt. Damit arbeitet das Register als Linksschieberegister (FF-3 $\rightarrow$ FF-0).

$(C_1, C_0) = (L, 0)$:

Die Eingänge $E_0, \ldots, E_3$ werden durchgestellt. Mit einem Takt werden diese Daten parallel von den D-FF an die Ausgänge $Q_0, \ldots, Q_3$ weitergegeben. Im nächsten Takt kann ein neues Datum $(E_0, \ldots, E_3)$ übernommen werden.

Literaturverzeichnis

[1] F.L. Bauer, G. Goos. *Informatik. Band1: Eine einführende Übersicht*. Springer, 1991.

[2] F.L. Bauer, G. Goos. *Informatik. Band2: Eine einführende Übersicht*. Springer, 1992.

[3] W. Bauer et al. *Studien- u. Forschungsführer Informatik*. Springer, 1989.

[4] R. Berghammer. *Informatik I und II. Vorlesungsskript 1995/96*. Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1996.

[5] M. Broy. *Informatik I: Problemnahe Programmierung*. Springer, 1992.

[6] M. Broy. *Informatik II: Rechnerstrukturen und maschinennahe Programmierung*. Springer, 1993.

[7] M. Broy. *Informatik III: Systemstrukturen und systemnahe Programmierung*. Springer, 1994.

[8] W. Coy. *Informatik-Spektrum 12*, 1989.

[9] W. Coy. *Aufbau und Arbeitsweise von Rechenanlagen*. Vieweg, 1992.

[10] P.J. Denning et al. *IEEE Computer Magazine*, Febr. 1989.

[11] G. Goos. *Vorlesungen über Informatik. Band I: Grundlagen und funktionales Programmieren*. Springer, 1995.

[12] G. Goos. *Vorlesungen über Informatik. Band II: Objektorientiertes Programmieren und Algorithmen*. Springer, 1996.

[13] B.W. Kernighan, D.M. Ritchie. *Programmieren in C*. C. Hanser, 1990.

[14] H. Klaeren. *Vom Problem zum Programm*. Teubner, Stuttgart, 1991.

[15] W. Kluge. *Informatik II. Vorlesungsskript 1987*. Inst. f. Informatik. u. Prakt. Math., CAU Kiel, 1987.

[16] W. Kluge, C. Aßmann. *Informatik für Ingenieure II. Vorlesungsskript 1996*. Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1996.

[17] B.O. Küppers. *Der Ursprung biologischer Information*. Piper, 1990.

[18] H. Liebig, T. Fink. *Rechnerorganisation*. Springer, 1993.

[19] H. Neumann, H.S. Stiehl. *Einführung in die Informatik für Mathematiker und Naturwissenschaftler. Vorlesungsskript 1992*. Fachbereich Informatik, Universität Hamburg, 1992.

[20] W. Oberschelp, G. Vossen. *Rechneraufbau und Rechnerstrukturen*. Oldenburg, 1994.

[21] R. P. Paul. *SPARC Architecture, Assembly Language Programming, & C*. Prentice Hall, 1994.

[22] P. Pepper. *Grundlagen der Informatik*. Oldenburg, 1995.

[23] *Spektrum der Wissenschaft*, Juli 1996.

[24] C.L. Tomdo, S.E. Gimpel. *Das C-Lösungsbuch*. C. Hanser, 1990. (zu Kernigham, Ritchie: Programmieren in C).

[25] C.F. v. Weizsäcker. *Die Einheit der Natur*. 1971.

[26] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, 1992.

Für die Vorlesung eignen sich besonders die Referenzen [1], [5], [6], [6], [7], [8], [13], [20],[22], [24].

Index

Unterstrichene Seitenzahlen verweisen auf die Definitionen der zugehörigen Begriffe.