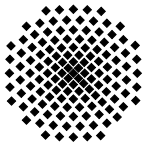


Universität Stuttgart
Fakultät 5

Prof. Dr.-Ing. U. G. Baitinger

Technische Informatik I

Diplomstudiengang Softwaretechnik
2. Semester

1 Der klassische Universalrechner

1.1 Das von Neumannsche Prinzip

1.2 Die Struktur des Universalrechners

Hauptspeicher, Befehlszähler,
Prozessor, Ein/Ausgabewerk

1.3 Digitale Prozessoren

Prozessorstrukturen
Befehlsformate
Ausführungsschritte

1.4 Digitale Steuerwerke

1.5 Digitale Rechenwerke

Digitale Funktionseinheiten
Wirkungsweise und Aufbau

1.6 Ein/Ausgabestrukturen

Kanal- und Buskonzept

2 Schaltungsebene

2.1 Transistoren im Digitalbetrieb

Bipolare Transistoren als Schalter
MOS-Transistoren als Schalter

2.2 MOS-Technologie

Photolithographie
NMOS-Technologie
CMOS-Technologie

2.3 Chip-Layout

Full-Custom IC Layout
Semi-Custom IC Layout

2.4 MOS-Schaltungstechnik

Statische MOS-Logik
Statische CMOS-Logik
Dynamische Domino-Logik

2.5 MOS-Speicherschaltungen

Speichermatrizen (RAM)
Statische MOS-Speicherzelle
Dynamische MOS-Speicherzelle

3 Logikebene

3.1 Die Darstellung von Schaltfunktionen

Indizierung, Funktionstabelle, KV-Diagramm
Strukturausdrücke, ausgewählte Schaltfunktionen

3.2 Kombinatorische Schaltungen („Schaltnetze“)

Darstellungsebenen und Entwurfsziele
Hauptsatz der Schaltalgebra
Minimierung nach Karnaugh-Veitch
Minimierung nach Quine-McCluskey
Lösung des Überdeckungsproblems

3.3 Sequentielle Schaltungen („Schaltwerke“)

Endliche diskrete Automaten
Elementare Schaltwerke
Entwurf von Schaltwerken
Kostenabschätzung

3.4 Programmierbare Digitalbausteine („PLD“)

Technologie und Schaltungstechnik
Logische Matrizen: ROM, PAL, PLA
Schaltwerksentwurf mit PLA

4 Register-Transfer-Ebene

4.1 Begriffsbestimmungen

4.2 Verbindungsstrukturen

Decoder, Multiplexer und Demultiplexer
Sammelleitung („Bus“)
Registerspeicher und Speichermatrizen

4.3 Verarbeitungseinheiten

Arithmetisch/Logische Einheit (ALU)

4.4 Mikroarchitektur-Synthese

Verhaltensbeschreibung
Scheduling und Allocation
Zuordnung und Baustein-Auswahl

4.5 Pipelining

Befehlspipeline und Befehlsausführung
Mehrstufige Pipeline

4.6 Entwurf eines Datenpfades

5 Algorithmische Ebene

5.1 Der Begriff des Algorithmus

5.2 Hardware-Beschreibungssprachen

HDL, VHDL, EDIF

5.3 Ein Entwurfsbeispiel

Verkehrsampelsteuerung

5.4 Der Entwurfsablauf

Deklaration der Systemtypen
Deklaration der Schnittstelle
Erstellen des Verhaltensmodells
Erstellen einer Testbeschreibung
Simulation des Verhaltensmodells

6 Architekturebene

6.1 Systemspezifikation

6.2 Die DLX-Architektur

Ein hypothetischer RISC-Prozessor

Die Lade/Speicher-Maschine DLX

Daten- und Befehlsformate

Ausgewählte Maschinenbefehle

Vollständige Befehlsliste

7 Hierarchie der Rechnersysteme

7.1 Die Entwicklung der Hardware

7.2 Der hierarchische Systementwurf

System- und Entwurfskomplexität
Zergliedernder Entwurfsstil („top-down design“)
Aufbauender Entwurfsstil („bottom-up design“)

7.3 Die Hierarchie der Entwurfsdaten

Rechnerarchitektur, Prozessor und Floorplan
Automat, Schaltwerk und Makrozellen
Zuordner, Schaltnetz und Zellen
Schaltglied, Schaltung und Polygone

7.4 Die Hierarchie der Entwurfsschritte

Spezifikation, logischer Entwurf
und physischer Entwurf

7.5 Der Entwurfsraum

Das „Y-Diagramm“ nach Gajski
Verhalten, Struktur und Geometrie

1 Der klassische Universalrechner

Prof. Dr.-Ing. U.G. Baitinger

1.1 Das von Neumannsche Prinzip

Arthur W. Burks, Hermann H. Goldstine, John von Neumann:
„Preliminary Discussion of the Logical Design of an Electronic Computing Instrument“,
Report to the U.S. Army Ordnance Department (1946).

Seit der Veröffentlichung dieses „vorläufigen“ Berichts durch John von Neumann umfaßt der klassische Universalrechner („General Purpose Computer“) die folgenden Funktionseinheiten:

- Der **Hauptspeicher** enthält sowohl Daten als auch Befehle, die einheitlich als „Informationen“ aufgefaßt werden. Der Speicherinhalt wird mit $bits \in \{0, 1\}$ dargestellt, d.h. zweiwertig („binär“). Der Hauptspeicher ist mit wahlfreiem Zugriff organisiert: Auf ein Datum oder einen Befehl kann mittels einer Speicheradresse direkt zugegriffen werden („Random Access Memory“, RAM).
- Das **Steuerwerk** nimmt Befehle aus dem Hauptspeicher entgegen, decodiert sie und erzeugt daraus eine geeignete zeitliche Sequenz von Steuersignalen für das Rechenwerk (falls erforderlich auch für andere Funktionseinheiten).
- Das **Rechenwerk** nimmt Daten aus dem Hauptspeicher entgegen, verknüpft sie aufgrund der genannten Steuersignale, d.h. nach Maßgabe der Befehle, und gibt das Ergebnis an den Hauptspeicher zurück.
- Das **Ein/Ausgabewerk** stellt die Kommunikation der oben genannten Funktionseinheiten mit den angeschlossenen Peripheriegeräten her (Tastaturen, Bildschirme, Disketten, Magnetplatten und dergl.).

Ferner hat der universell einsetzbare Digitalrechner nach John von Neumann et al. im wesentlichen folgende Merkmale:

- Nicht nur die Daten, sondern auch die **Befehle sind veränderbar** in einem Hauptspeicher vom Schreib/Lese-Typ abgespeichert.
- Die Speicheradresse des Befehls, der als nächster ausgeführt werden soll, wird entweder implizit durch einen **Befehlszähler** oder explizit durch einen Sprungbefehl erzeugt.
- Zwischen Befehlen und Daten wird zeitlich unterschieden, indem abwechselnd zwei unterschiedliche Phasen durchlaufen werden:
  - Während der **Befehlsholphase** wird der Hauptspeicher mit dem Inhalt des Befehlszählers oder mit einer expliziten Sprungadresse adressiert. Die aus dem Speicher gelesene Information wird an das Steuerwerk weitergegeben und als Befehl interpretiert.
  - Während der **Befehlsausführungsphase** wird der Hauptspeicher nacheinander mit zwei Operandenadressen adressiert, die im Befehl enthalten sind. Die aus dem Speicher gelesenen Informationen werden als Daten behandelt, in das Rechenwerk eingegeben und dort entsprechend dem Befehl verknüpft. Das Ergebnis wird in den Hauptspeicher geschrieben, meist an die Adresse des ersten der beiden Operanden.

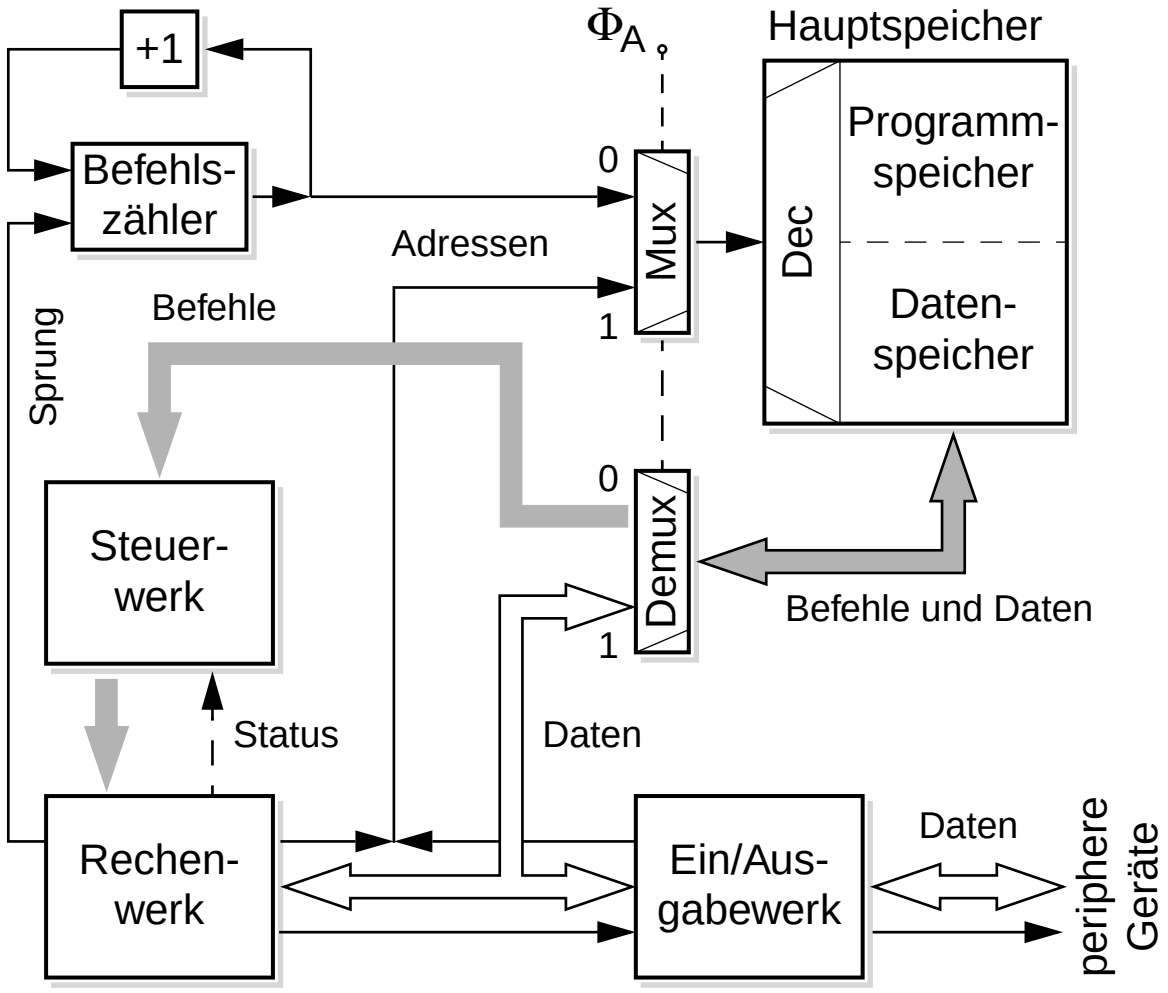


Bild 1.1: Logische Struktur eines Universalrechners

Nach dem von Neumannschen Prinzip wird zwischen Befehlen und Daten *zeitlich* unterschieden:

- Befehlsholphase: $\Phi_A = 0$
- Befehlsausführungsphase: $\Phi_A = 1$

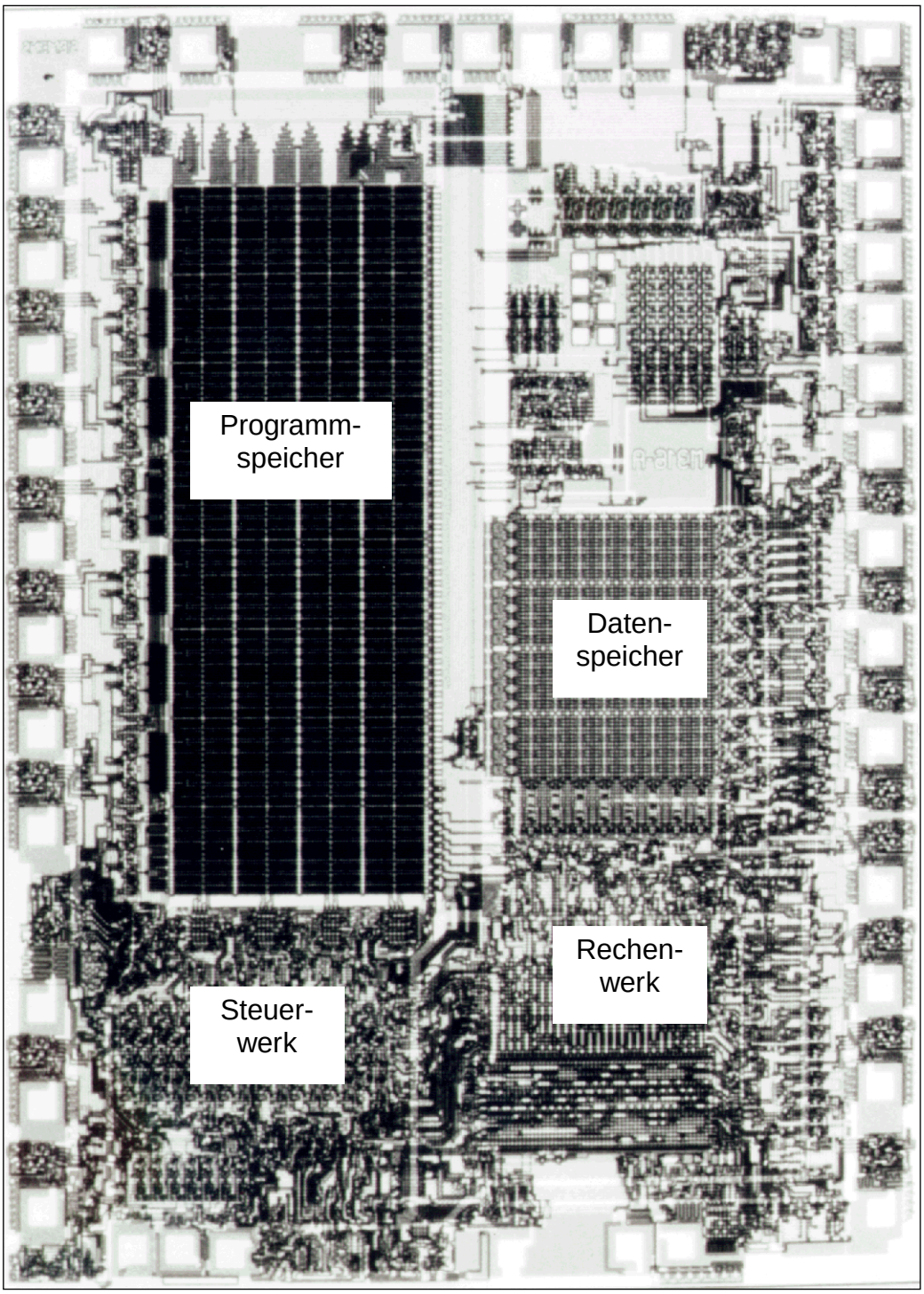


Bild 1.2: Physische Struktur eines Universalrechners.
Dieser Ein-Chip-Mikrocomputer befindet sich auf einem Siliziumchip von 25 mm^2 Fläche.

1.2 Die Struktur des Universalrechners

Ob Materialbearbeitung mit Maschinen, Energieumwandlung mit Generatoren und Motoren oder Informationsverarbeitung mit Computern - grundsätzlich gilt für jede Art von Verarbeitung: „Eingabe → Verarbeitung → Ausgabe“. So betrachtet ist der Computer ein Spezialfall der Verarbeitungstechnik, indem er eine der drei genannten, naturwissenschaftlich relevanten Kategorien verarbeitet.

Da ein komplizierter Verarbeitungsvorgang nicht in einem einzigen Schritt durchgeführt werden kann, teilt man ihn auf in eine geeignete Folge einfacherer Schritte, d.h. in einen Steuerungsablauf, der die zeitliche Abfolge der Verarbeitungsschritte steuert, und in die eigentliche Ausführung dieser Verarbeitungsschritte.

Beim Computer erfolgt die Eingabe von Informationen in der Regel über eine Tastatur, ihre Ausgabe am Bildschirm. Während der Verarbeitung kann ein Computer allerdings nicht ständig darauf warten, daß alle Informationen über die Tastatur eingegeben werden. Deshalb werden zunächst alle Daten und die damit durchzuführenden Verarbeitungsschritte, letztere in Form von Maschinenbefehlen, in einen Speicher geladen. Die gespeicherten Befehle dienen zur Steuerung, die gespeicherten Daten sind zur Ausführung bestimmt. Damit der Computer die Informationsverarbeitung selbständig durchführen kann, wird die Steuerung als digitaler Automat implementiert. Ein nach diesem Prinzip aufgebauter Computer ist als Universalrechner verwendbar („General Purpose Computer“).

Die Funktion des Universalrechners ergibt sich aus dem Zusammenspiel seiner funktionellen Einheiten. Das **Steuerwerk** holt aus dem **Programmspeicher** den als nächsten auszuführenden Befehl und teilt daraufhin dem **Rechenwerk** mit, an welcher Stelle des **Datenspeichers** die zugehörigen Daten zu finden und wie diese miteinander zu verknüpfen sind. Das Rechenwerk führt die Verknüpfung aus, legt das Ergebnis im Datenspeicher ab und meldet dem Steuerwerk den Vollzug. Daraufhin holt sich das Steuerwerk aus dem Programmspeicher den nächsten auszuführenden Befehl, und so fort. Man pflegt den Daten- und den Programmspeicher des Universalrechners als einheitlichen **Hauptspeicher** zusammenzufassen.

Da Steuerwerk und Rechenwerk eng gekoppelt sind, werden sie gemeinsam als **Prozessor** bezeichnet. Ein **Mikro-Prozessor** ist demnach ein in höchstintegrierter, mikroelektronischer Technologie („Very Large Scale Intergration“, VLSI) hergestellter, in seinen Abmessungen extrem kleiner Prozessor auf einem Siliziumchip von ca. 1,6 cm^2 Fläche (1998), aber mit der Leistungsfähigkeit eines Digitalrechners mittlerer Größenordnung.

Zusätzlich zu Bildschirm und Tastatur steht insbesondere zur Ein- und Ausgabe größerer Datenmengen eine Fülle externer Speichermedien zur Verfügung, wie Magnetbänder, -platten und -disketten, sowie Drucker aller Art. Den Ein-/Ausgabegeräten ist gemeinsam, daß sie als elektromechanische Geräte im Vergleich zum Prozessor und zum Hauptspeicher, die beide in Siliziumtechnologie realisiert sind, relativ viel Energie benötigen und relativ langsam sind. Man braucht daher zur Anpassung der Umgebung an den Rechnerkern, der wesentlich schneller arbeitet und einen weit niedrigeren Energiebedarf aufweist, ein **Ein-/Ausgabewerk**, das seinerseits wie ein weiterer Prozessor strukturiert ist.

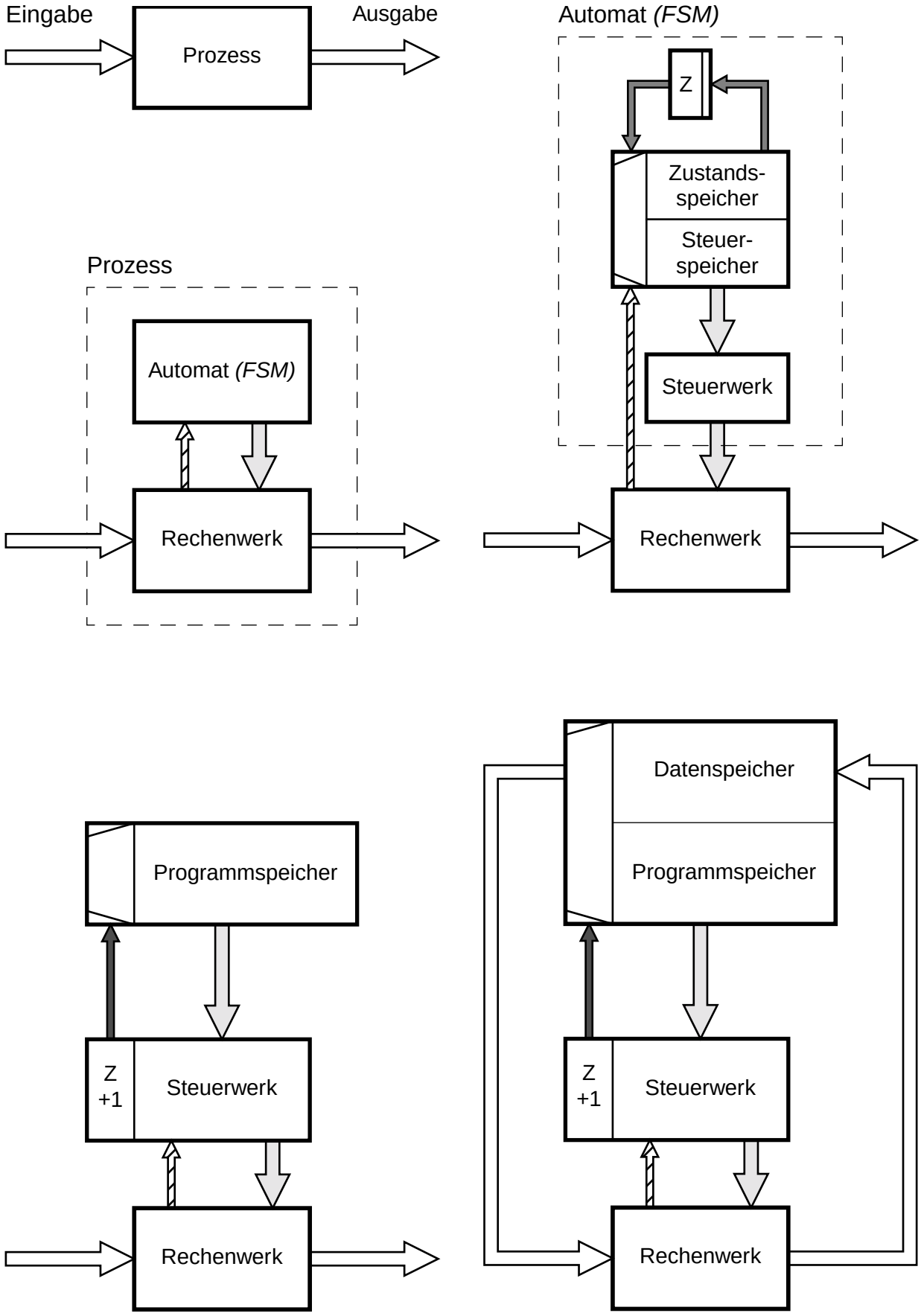


Bild 1.3: Vom Prozess zum Universalrechner

1.3 Digitale Prozessoren

1.3.1 Prozessorstrukturen

In der technischen Informationsverarbeitung verwendet man die funktionelle Begriffskette:

Daten-Eingabe → **Prozess** → Daten-Ausgabe

Der abstrakte Prozess muß zu seiner Ausführung auf einen konkreten **Prozessor** abgebildet werden. Tatsächlich ist es sogar so, daß der Prozess in eine **Programmstruktur** abgebildet wird, die binär codiert im Hauptspeicher des Universalrechners untergebracht ist. Die Begriffsklärungen sind jedoch noch nicht abgeschlossen, was im übrigen für eine junge Wissenschaft wie die Informatik charakteristisch ist:

- In der Steuerungstechnik faßt man den Hauptspeicher und das Steuerwerk „im engeren Sinn“ als **Steuerwerk** „im weiteren Sinn“ zusammen.
- In der Rechnertechnik ist es üblich, das Steuerwerk „im engeren Sinn“ und das Rechenwerk als **Prozessor** zusammenzufassen. Ferner faßt man Prozessor und Hauptspeicher als **Zentraleinheit** des informationsverarbeitenden Systems zusammen.

Prozessorstrukturen werden grundsätzlich aus zwei Anteilen gebildet:

- **Funktionseinheiten** („Knoten“)
  - zur Verarbeitung binär codierter Informationen durch ihre arithmetische oder logische Verknüpfung, z.B. im Rechenwerk („Arithmetic/Logic Unit“, ALU), und/oder
  - zur Speicherung, wie Eingangs- und Ausgangsregister, Puffer- und Registerspeicher, FIFO-Warteschlangen und LIFO-Stapelspeicher.
- **Strukturelemente** („Datenpfade“)
  - zur Kommunikation der Funktionseinheiten untereinander; ihre Topologie ist für die Prozessorstruktur bestimmend.
  - Als Strukturelemente werden entweder Punkt-zu-Punkt-Verbindungen verwendet, die individuell vermittelt und von der Informationsquelle zur -senke durchgeschaltet werden, oder
  - gemeinsam genutzte Sammelschienen, die in einer Konferenzschaltung *für alle* (lat. *omnibus*) zur Verfügung stehen („Bus-Verbindungen“).

Bei nur einem Bus muß dessen Benutzung den einzelnen Funktionseinheiten zeitlich exklusiv zugeteilt werden, bei mehreren Bussen sind zeitlich parallele Abläufe möglich, so daß Verarbeitungszeit eingespart werden kann. Ein elegantes Konzept zur Zeiteinsparung sieht vor, Informationen nacheinander auf einen Bus zu legen („Pipelining“); dazu sind zusätzliche Zwischenspeicher zur Informationstrennung notwendig.

Heute haben sich Bus-Verbindungen auch innerhalb der Prozessoren durchgesetzt, so daß sie nach ihrer internen **Bus-Struktur** klassifiziert werden können.

1.3.2 Befehlsformate

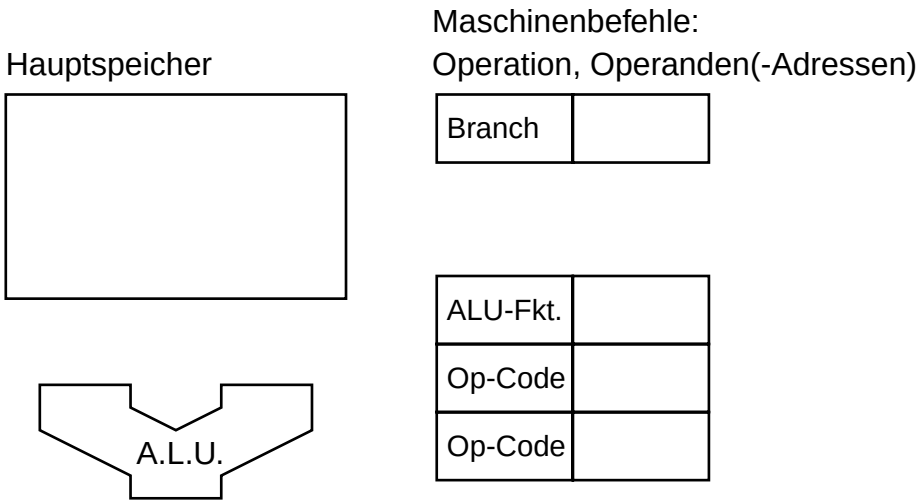


Bild 1.4: Abstrakter Prozessor

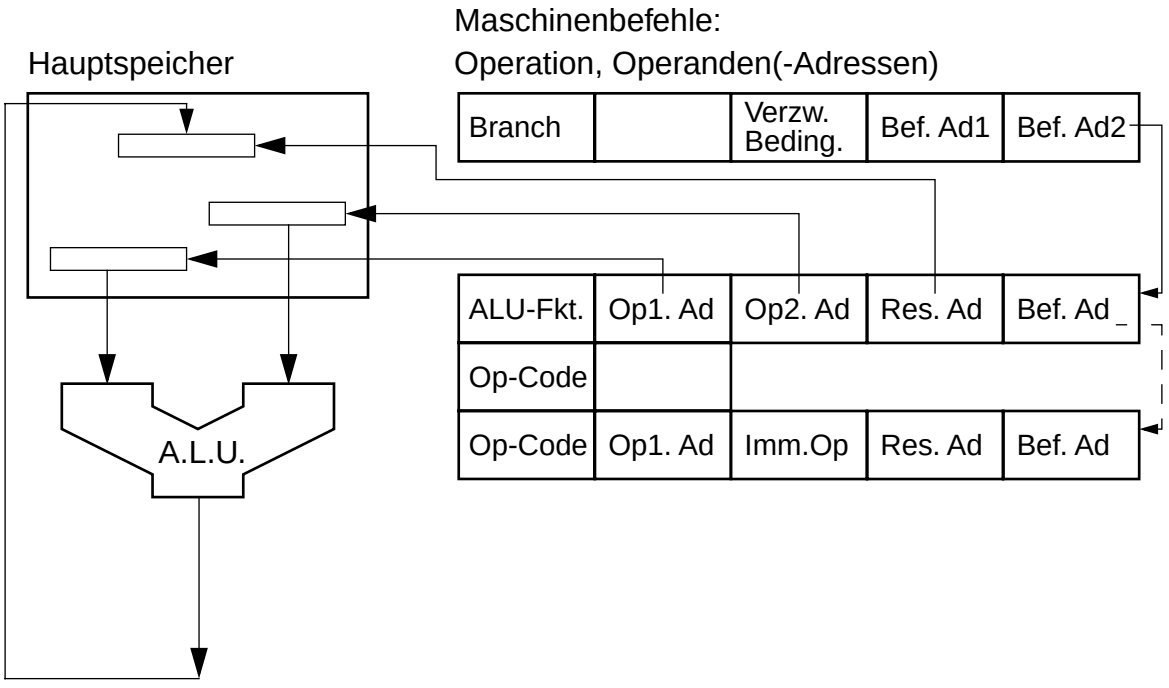


Bild 1.5: Vier-Adress-Befehlsformat

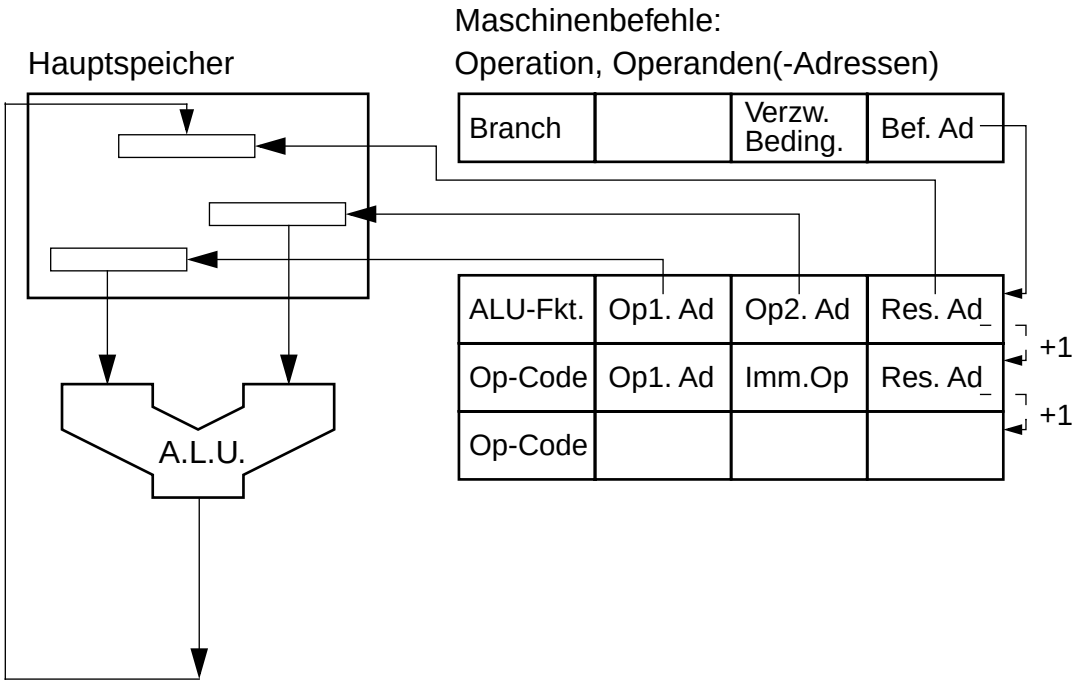


Bild 1.6: Drei-Adress-Befehlsformat

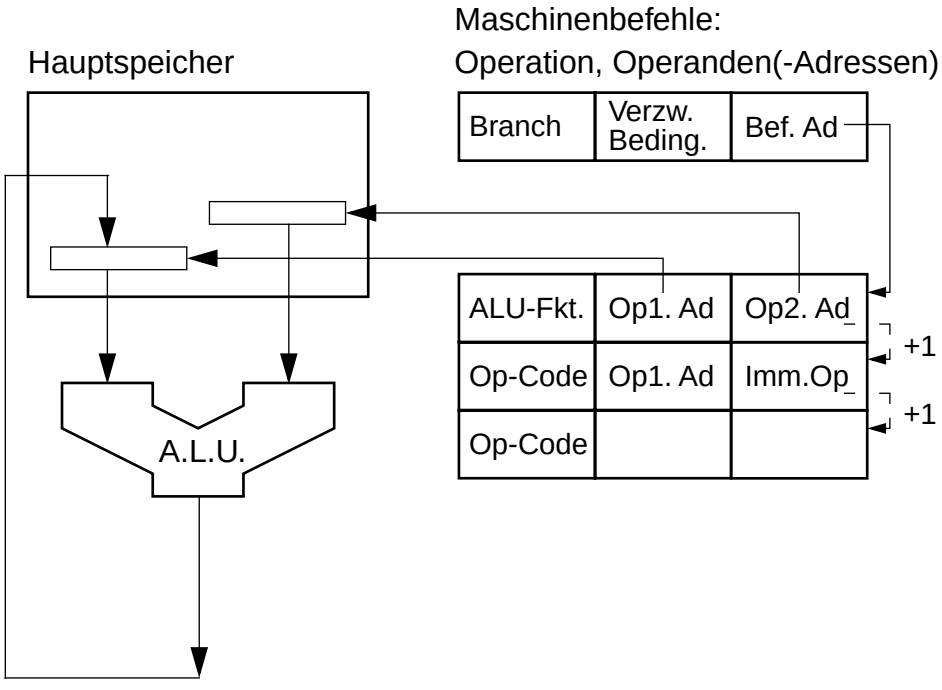


Bild 1.7: Zwei-Adress-Befehlsformat

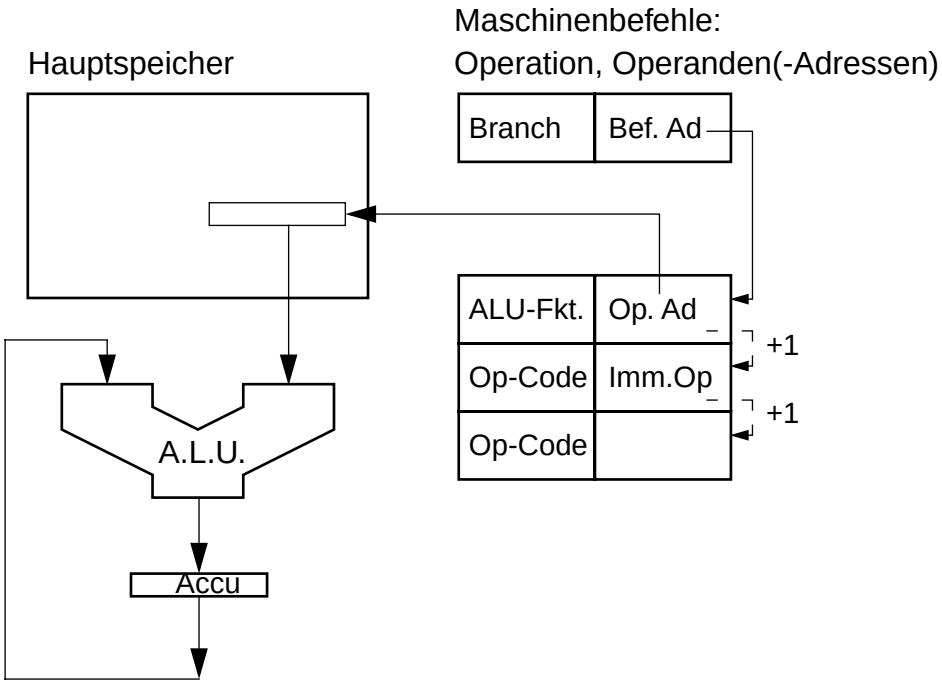


Bild 1.8: Ein-Adress-Befehlsformat

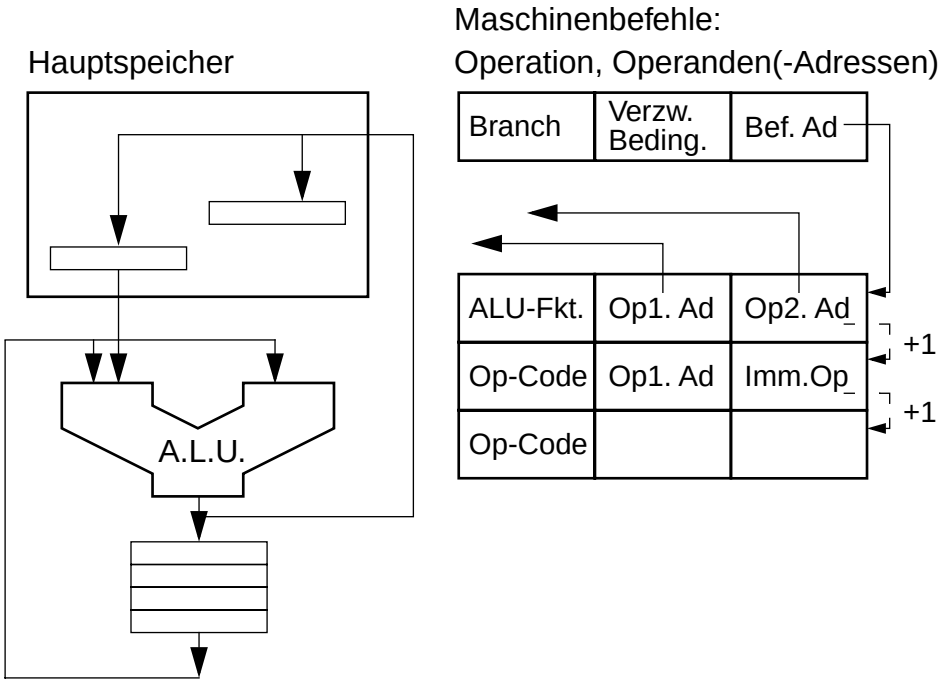


Bild 1.9: Zwei-Adress-Befehlsformat mit Registerspeicher

1.3.3 Befehlsausführungsschritte

Ein digitaler Universalrechner verfüge über

- Zwei-Adress-Befehle und einen Registerspeicher („Lade/Speicher-Architektur“).

Nach dem von Neumannschen Prinzip durchläuft der Rechner während der Programmausführung bei jedem Maschinenbefehl abwechselnd zwei unterschiedliche Phasen: die Befehlsholphase und die Befehlsausführungsphase. Damit kann er jeden Maschinenbefehl seines Befehlssatzes in folgenden Schritten abarbeiten:

1. Befehl holen *(Instruction Fetch)*, **IF**
   d.h. Hauptspeicher mit Befehlszählerinhalt adressieren
   und Befehlszähler erhöhen.

2. Befehl decodieren *(Instruction Decode)* **ID**
   und Quellregisterinhalte in A- bzw. B-Register holen.

3. Befehl ausführen *(Execute)* **EX**

   a) Lade-/Speicher-Befehl: effektive Hauptspeicheradresse berechnen
      und Datenregister laden;

   b) Verzweigungsbefehl: Zieladresse berechnen
      und Verzweigungsbedingung setzen;

   c) ALU-Befehl: Berechnung ausführen.

4. Speicherzugriff *(Memory Access)* **MEM**

   a) Lade-Befehl: Daten vom Hauptspeicher ins Datenregister laden;
      Speicher-Befehl: in umgekehrter Richtung speichern;

   b) Verzweigungsbefehl: Zieladresse in Befehlszähler laden,
      falls Verzweigungsbedingung erfüllt;

   c) ALU-Befehl: (nichts)

5. Rückschreiben *(Write Back)* **WB**

   a) Lade-Befehl: Datenregisterinhalt ins Zielregister laden;
      Speicher-Befehl: (nichts);

   b) Verzweigungsbefehl: (nichts);

   c) ALU-Befehl: Ergebnis ins Zielregister schreiben.

1.4 Digitale Steuerwerke

Ein digitales Steuerwerk kann als endlicher diskreter **Automat** betrachtet werden, wobei anzumerken ist, daß hier ein Automat ein mathematisches Abstraktum und keine technische Vorrichtung ist. Der Automat kann nur wohlunterschiedene („diskrete“) und abzählbare („digitale“) Zustände Z_i annehmen, deren Anzahl endlich ist („Finite State Machine“, FSM). Der Automatenzustand ist binär codiert und in einem Register gespeichert. Mit n Bits lassen sich 2^n verschiedene Automatenzustände codieren, z.B. $2^8 = 256$.

Die technische Implementierung eines Automaten wird **Schaltwerk** genannt. Es gibt unterschiedliche elektronische Technologien, um Schaltwerke zu realisieren:

- In „maßgeschneiderter Logik“ werden die Automatenfunktionen durch individuelle, meist zweistufige Schaltungen aus Und-Gattern gefolgt von Oder-Gattern realisiert.
- Bei regelmäßig strukturierten „Programmierbaren Logischen Anordnungen“ (PLA), die als elektronische Digitalbausteine erhältlich sind, sind die Und-Gatter in einer sog. Und-Matrix, die Oder-Gatter in einer Oder-Matrix zusammengefaßt.
- Bei Verwendung eines Festwertspeichers („Read Only Memory“, ROM) oder eines Schreib/Lese-Speichers („Random Access Memory“, RAM) entspricht der Decoder der Und-Matrix, die Speichermatrix der Oder-Matrix des PLA.

Vor allem bei der Alternative mit Speicherbausteinen wird die Verbindung zur Programmierung deutlich: Die ROM/RAM-Anordnung kann prinzipiell als der Hauptspeicher eines Digitalrechners betrachtet werden. Enthält die Speicheranordnung codierte Steuervektoren („Maschinenbefehle“), so müssen sie durch ein nachgeschaltetes Steuerwerk decodiert und sequenziert werden, um daraus eine geeignete Folge von Steuersignalen für das Rechenwerk zu erzeugen. Das Rechenwerk nimmt Daten entgegen, verknüpft sie nach Maßgabe des aktuellen Maschinenbefehls und gibt das Ergebnis aus. Da der Hauptspeicher ohnedies vorhanden ist, um das Programm zu speichern, wird er vergrößert und zusätzlich zur Datenspeicherung verwendet. Man beachte aber, daß die gespeicherten Daten nicht zur Struktur und Funktion des Automaten gehören.

Eine digitale Steuerungsfunktion kann durch einen „Automatengraphen“ beschrieben werden, der im Fall eines Prozessors eine charakteristische Topologie aufweist:

- die Decodierung des Operationscodes des aktuellen Maschinenbefehls während der Befehlsholphase wird durch eine mächtige Verzweigung dargestellt,
- die Befehlsausführungsphasen für die unterschiedlichen Befehlstypen durch weitgehend unverzweigte Ketten.

Wendet man das Prinzip des programmierten Automaten, der einen Befehlsspeicher enthält, rekursiv auch auf das Steuerwerk an, so daß dieses ebenfalls einem Automaten in Form einer Speicherstruktur entspricht, so bezeichnet man das darin enthaltene Steuerprogramm als **Mikroprogramm**, da es hierarchisch unterhalb der Befehlsebene, d..h. des **Hauptprogramms** liegt und diese interpretiert.

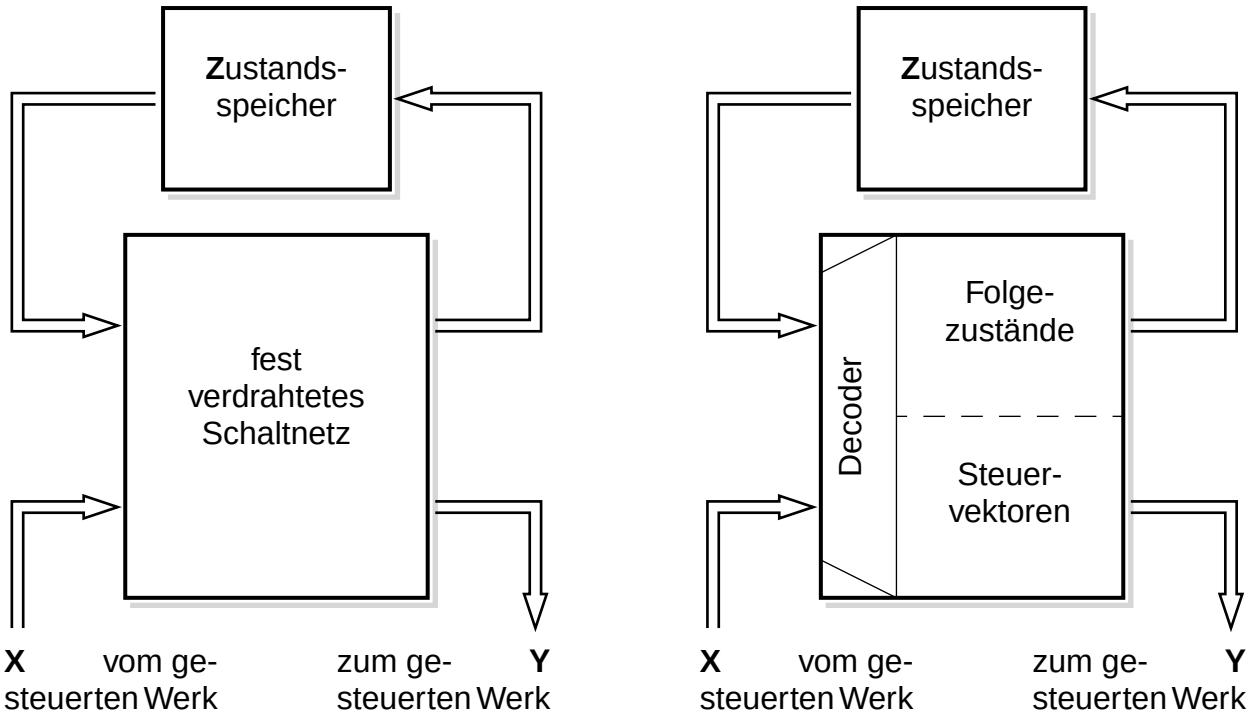


Bild 1.10: Automatenstrukturen allgemeiner Steuerwerke (schematisch)

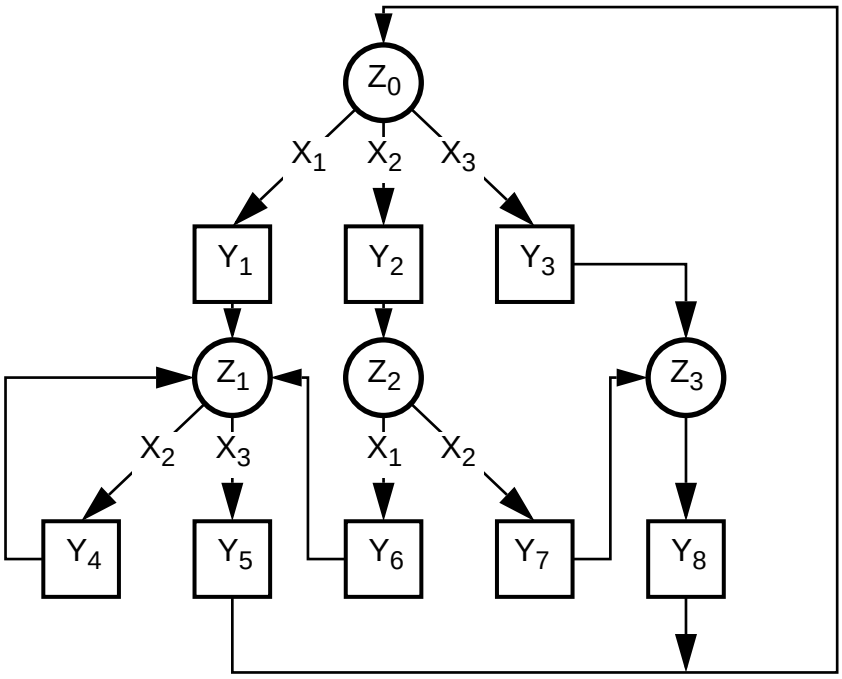


Bild 1.11: Automatengraph eines allgemeinen Steuerwerks (typisch vermascht)

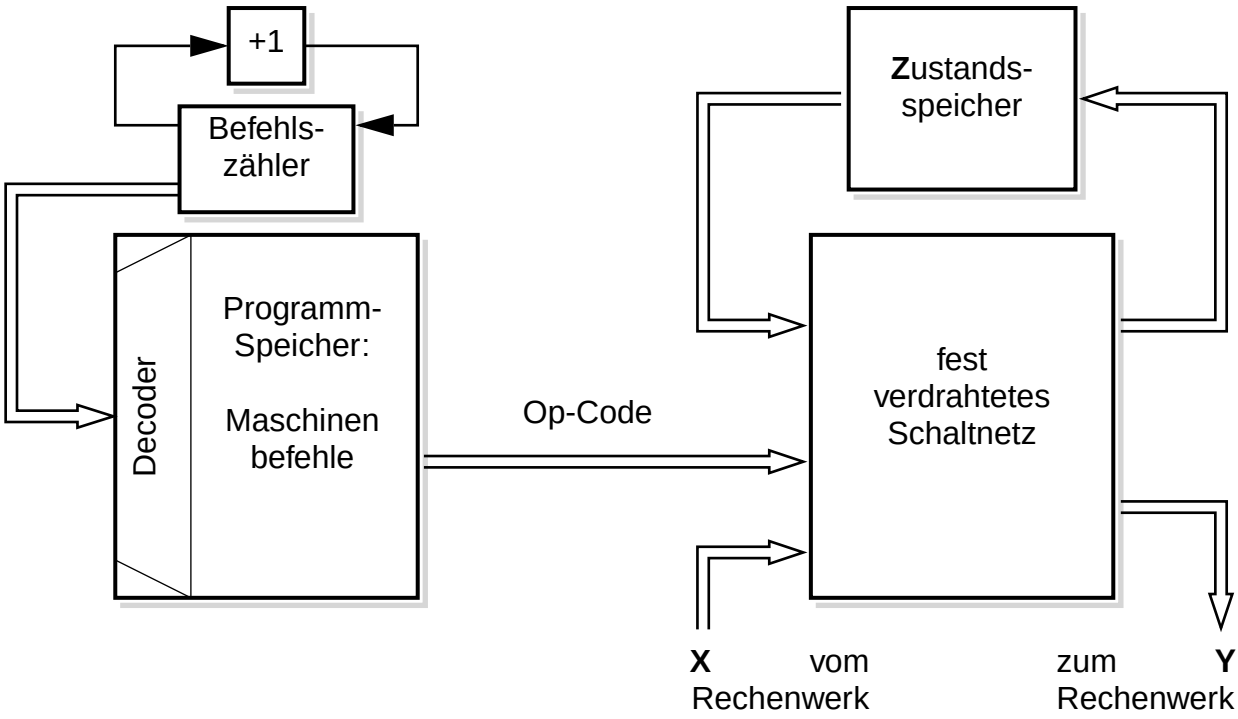


Bild 1.12: Programmspeicher und Prozessorsteuerwerk (schematisch, vereinfacht)

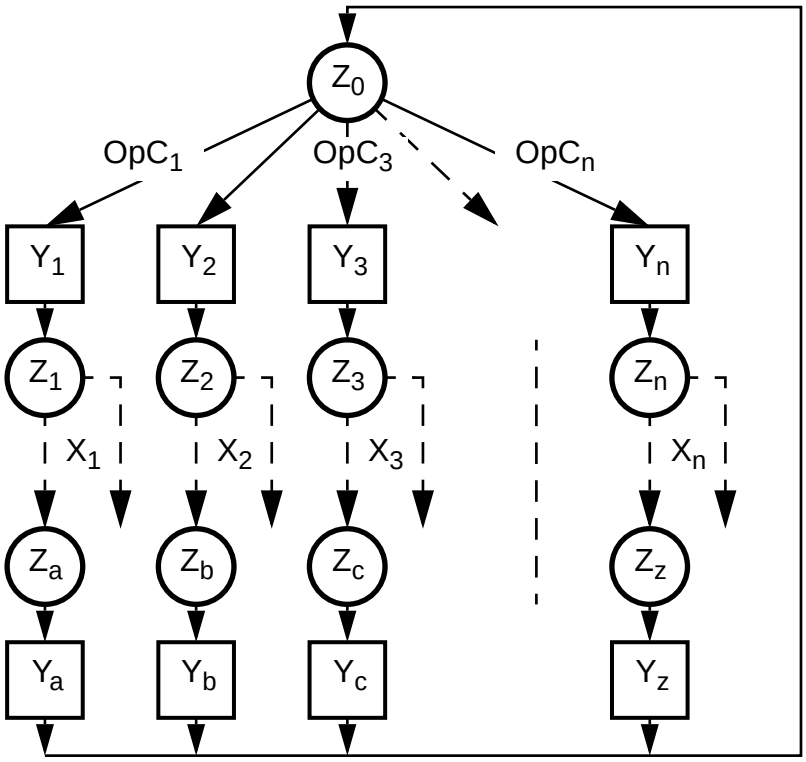


Bild 1.13: Automatengraph eines Prozessorsteuerwerks

1.5 Digitale Rechenwerke

1.5.1 Digitale Funktionseinheiten

Beim strukturellen Aufbau digitaler Rechenwerke unterscheidet man im wesentlichen drei Typen von Funktionseinheiten:

- **Datenquellen** und **Datensenken**, z.B. Speicherregister oder kleine, schnelle Pufferspeicher;
- **Datenpfade** als Transportmedien, z.B. interne Busse oder Punkt-zu-Punkt-Verbindungen, die je nach Bedarf durchgeschaltet oder blockiert werden;
- **Verknüpfungsschaltungen**, d.h. ein Vorrat an Verknüpfungsfunktionen:
  - arithmetische Operationen, wie die Grundrechnungsarten Addition + , Subtraktion - , Multiplikation * und Division / ;
  - logische Verknüpfungen, wie Konjunktion & („Und“), Disjunktion $\vee$ („Oder“) und Negation $\neg$ („Nicht“), aber auch komplexere, wie z.B. die Antivalenz $\oplus$ („exklusives Oder“);
  - einfache Transferoperationen ohne eine Veränderung der Daten;
  - Schiebeoperationen (n bit nach links: Multiplikation mit 2^n, n bit nach rechts: Division durch 2^n).

Die Funktionseinheiten eines Rechenwerks werden mit Entwurfsmethoden der digitalen Schaltungstechnik implementiert. Beliebig komplizierte Funktionen können nach den Regeln der Schaltalgebra in Schaltnetze mit elementaren logischen Gattern aufgelöst werden, die lokal wieder so zu größeren Funktionszellen zusammengefaßt werden, daß sie den Digitalbausteinen entsprechen, die die gewählte Technologie anbietet.

Struktur und Funktion digitaler Rechenwerke lassen sich im Prinzip auf *eine einzige* Schaltfunktion zurückführen. Es läßt sich nämlich mathematisch beweisen, daß sich beliebig komplizierte Verknüpfungsfunktionen in schaltalgebraische Ausdrücke umformen lassen, die nur einen Schaltfunktionstyp enthalten, z.B. ausschließlich „negierte Und“ = NAND ($\overline{\&}$) oder ausschließlich „negierte Oder“ = NOR ($\overline{\vee}$), was zur praktischen Konsequenz hat, daß nur ein Gattertyp notwendig ist, um beliebige Schaltnetze zu realisieren. Das soll mit zwei einfachen Beispielen erläutert werden.

1.5.2 Wirkungsweise digitaler Rechenwerke

Es gibt zwei völlig unterschiedliche Vorgehensweisen, um eine digitale Funktion auszuführen:

- Eine **Hardware-Lösung** besteht aus einem festverdrahteten Schaltnetz, das eine ausreichende Anzahl logischer Gatter enthält. Die Ausführung einer Funktion erfolgt zeitlich *parallel*.
- Eine **Software-Lösung** benötigt im Minimalfall ein einzelnes logisches Gatter, das aber mehrfach nacheinander durchlaufen werden muß, wozu zusätzliche Zwischenspeicher erforderlich sind. Die Ausführung einer Funktion erfolgt zeitlich *sequentiell*.

Beispiel 1.1: Hardware-Lösung

Nehmen wir als Beispiel die oben bereits erwähnte Schaltfunktion der Antivalenz ⊕. Sie kann nach den Regeln der Schaltalgebra in eine Disjunktion ∨ von Konjunktionen & aufgelöst werden:

$$y = (x_2 \oplus x_1) \tag{0.1}$$

$$y = (x_2 \,\&\, x_1) \vee (x_2 \,\&\, x_1) \tag{0.2}$$

Nach der de Morganschen Regel läßt sich dieser Ausdruck in einen anderen transformieren, der nur noch NAND-Verknüpfungen ($\overline{\&}$) enthält:

$$y = (\bar{x}_2 \,\overline{\&}\, x_1) \,\overline{\&}\, (x_2 \,\overline{\&}\, \bar{x}_1) \tag{0.3}$$

Man kann Gl.(1.3) isomorph in eine Schaltnetzstruktur mit NAND-Gattern umsetzen (Bild 1.12).

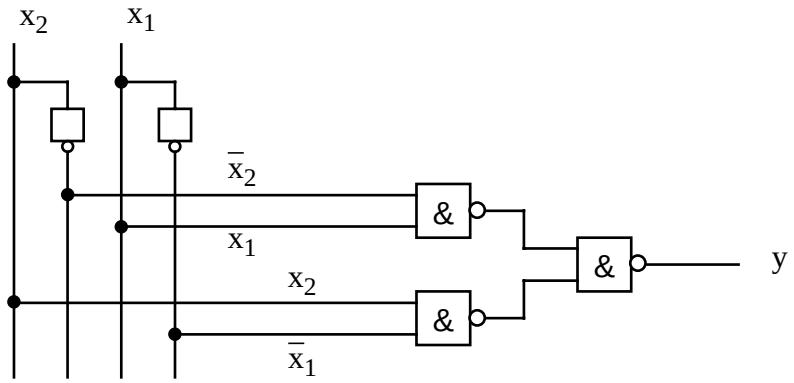


Bild 1.14: NAND-Schaltnetzstruktur der Antivalenz $(x_2 \oplus x_1)$

Beispiel 1.2: Software-Lösung

Für das gewählte Beispiel der Antivalenz ⊕ erfolgt die sequentielle Ausführung durch Mehrfachausnutzung eines NAND-Gatters mit Speicherung der Zwischenergebnisse, d.h. in diesem einfachen Beispiel wird die Antivalenz in fünf NAND-Schritten ($\overline{\&}$) ausgeführt:

1. Schritt: $y_1 = (x_2 \,\overline{\&}\, 1) = \bar{x}_2$ → Zwischenspeicherung y_1

2. Schritt: $y_2 = (x_1 \,\overline{\&}\, 1) = \bar{x}_1$ → Zwischenspeicherung y_2

3. Schritt: $y_3 = (y_1 \,\overline{\&}\, x_1)$ → Zwischenspeicherung y_3

4. Schritt: $y_4 = (x_2 \,\overline{\&}\, y_2)$ → Zwischenspeicherung y_4

5. Schritt: $y = (y_3 \,\overline{\&}\, y_4)$ → Ergebnisspeicherung y

Die sequentielle Ausführung bringt eine längere Verarbeitungszeit mit sich, während die parallele in einem Schritt erfolgt, d.h. **Hardware-Lösungen** sind starr, aber schneller und betriebssicherer, **Software-Lösungen** flexibel, aber langsamer und weniger reproduzierbar. („Mit einer reinen Software-Lösung kommt man in der Regel nicht durch den TÜV.“) Nach dem Stand der Technik kommen in der Informationstechnik beide Lösungsalternativen zum Einsatz.

Die sequentielle Abarbeitung zur Verringerung des Schaltungsaufwandes liegt der technischen Informationsverarbeitung historisch zugrunde: Bereits der erste Vorschlag durch John von Neumann (1946) beruht auf dieser Überlegung. Der Ersatz „schnellerer“ Hardware durch „langsamere“ Software lohnt sich, wenn erstere teuer und die Programmierung wenig aufwendig ist. Was jedoch die weitere Entwicklung informationsverarbeitender Systeme betrifft, beginnt sich das Verhältnis zur Zeit umzukehren: Der Preisverfall der Hardware durch die Fortschritte der Mikroelektronik und die Existenz umfangreicher und komplexer Software-Systeme, deren Pflege und Wartung aufwendig sind, machen es lohnend, anwendungsspezifische integrierte Schaltungen („Application Specific Integrated Circuits“, ASIC) einzusetzen, zu deren Entwurf sind leistungsfähige rechnergestützte Entwurfssysteme zur Verfügung stehen („Computer Aided Design“, CAD), so daß sich derzeit die Programmierung von Anwendungslösungen mit Mikroprozessoren verlagert zur Erstellung umfangreicher CAD-Systeme zum Entwurf von ASICs.

1.5.3 Aufbau digitaler Rechenwerke

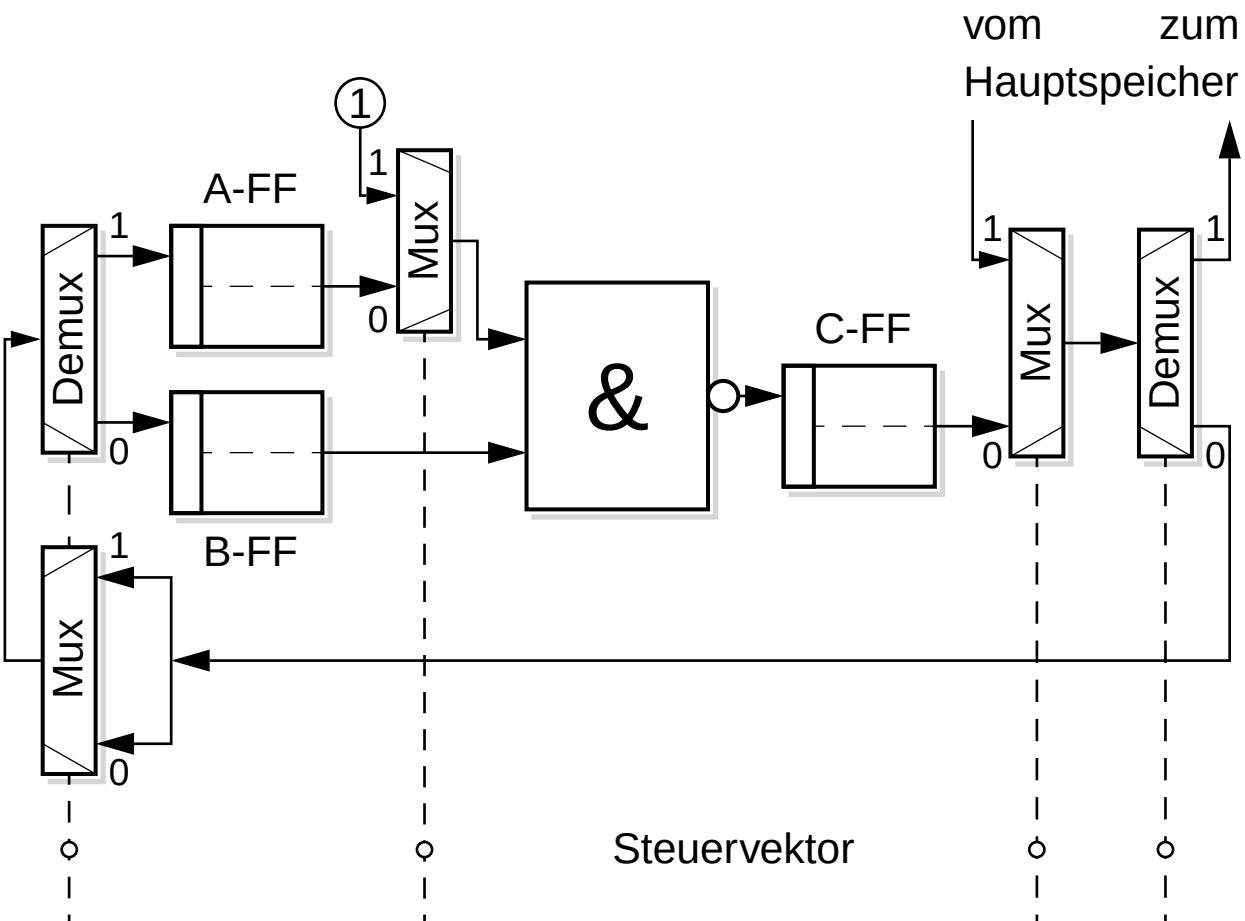
Bild 1.13 zeigt die Struktur eines elementaren Rechenwerks, das *2 bit* nach einer beliebigen Schaltfunktion zu einem Ergebnis von *1 bit* verknüpfen kann, das also auch die im Beispiel 1.2 vorgestellte Verarbeitung der Antivalenz $\oplus$ durch eine Software-Lösung leistet.

In den beiden Speicherzellen A-FF und B-FF („Flipflops“, die jeweils *1 bit* speichern können), die sich an den Eingängen des zentralen NAND-Gatters ($\overline{\&}$) befinden, werden nach Bedarf die beiden Operanden x_1 und x_2 sowie Zwischenergebnisse y_i gespeichert, das jeweilige Verknüpfungsergebnis in der Speicherzelle C-FF am Ausgang des NAND-Gatters. Beim Zuschalten der logischen Konstante 1 bewirkt das NAND-Gatter eine Negation der Eingangsvariable.

Die Datenpfade werden von Multiplexern (Mux) und Demultiplexern (Demux) durchgeschaltet. Die binären Ansteuervariablen für die (De-)Multiplexer werden vom Steuerwerk erzeugt. Man kann die Aneinanderreihung aller Ansteuervariablen als „Steuervektor“ für die Datenpfade bezeichnen. Synonym dazu sind die Begriffe **„Maschinenbefehl“** und **„Instruktion“**, die hier in binärer, uncodierter Form vorliegen.

Vereinfachend kann man die Mux/Demux-Paare durch **Busverbindungen** darstellen (Bild 1.14).

Erweitert man die nur *1 bit* führenden Datenpfade und Busverbindungen auf *n bit* Breite (n = 8 . . 16 . . 32) und ersetzt man die Einzelflipflops durch *n bit* breite **Register**, das NAND-Gatter durch eine **Arithmetisch/logische Einheit** („Arithmetic/Logic Unit“, ALU), die verschiedene arithmetische und/oder logische Operationen sowie ggf. auch Transfer- und Schiebeoperationen ausführen kann, so erhält man die allgemeine Struktur eines digitalen Rechenwerks nach Bild 1.15. Obiger Steuervektor ist bei *m* unterschiedlichen ALU-Funktionen durch Anfügen zusätzlicher *ld m* Steuerbits zu erweitern.



Steuervektor				
0	1	$HS(x_2) \rightarrow$ B-FF; $1\overline{\&}x_2 \rightarrow \overline{x}_2$	1	0
1	0	$\overline{x}_2 \rightarrow$ A-FF	0	0
0	0	$HS(x_1) \rightarrow$ B-FF; $\overline{x}_2\overline{\&}x_1 \rightarrow y_3$	1	0
-	-	$y_3 \rightarrow HS(y_3)$	0	1
0	1	$HS(x_1) \rightarrow$ B-FF; $1\overline{\&}x_1 \rightarrow \overline{x}_1$	1	0
1	0	$\overline{x}_1 \rightarrow$ A-FF	0	0
0	0	$HS(x_2) \rightarrow$ B-FF; $\overline{x}_1\overline{\&}x_2 \rightarrow y_4$	1	0
1	0	$y_4 \rightarrow$ A-FF	0	0
0	0	$HS(y_3) \rightarrow$ B-FF; $y_4\overline{\&}y_3 \rightarrow y$	1	0
-	-	$y \rightarrow HS(y)$	0	1

Bild 1.15: Darstellung eines elementaren Rechenwerks mit Multiplexer/Demultiplexer-Paaren. Steuervektoren zur Ausführung der Antivalenz.

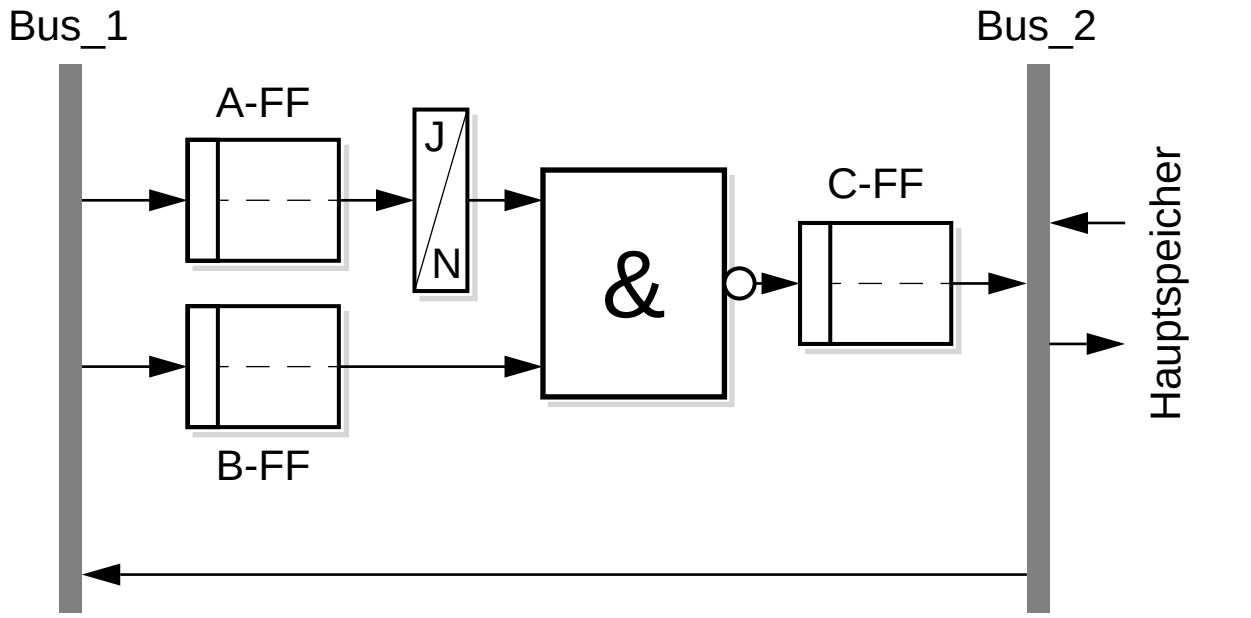


Bild 1.16: Vereinfachte Darstellung eines elementaren Rechenwerks mit Bussen

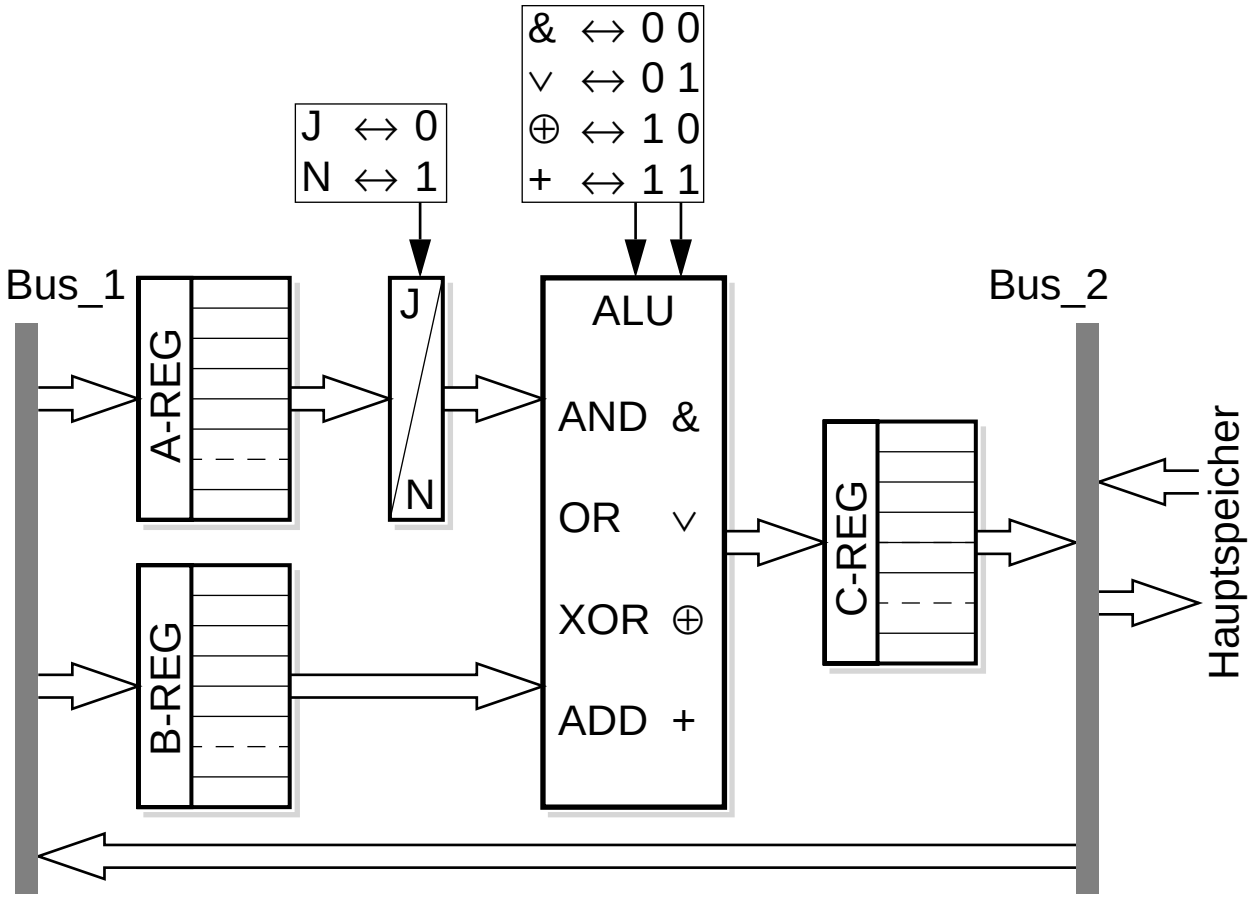


Bild 1.17: Typisches Rechenwerk „Arithmetic/Logic Unit“, ALU

1.6 Ein/Ausgabestrukturen

In einem informationsverarbeitenden System lassen sich grundsätzlich zwei technologisch unterschiedlich realisierte Bereiche unterscheiden:

- Die **Zentraleinheit** (Hauptspeicher, Steuerwerk und Rechenwerk) wird in der Technologie der *Mikroelektronik* realisiert, um möglichst hohe Verarbeitungsgeschwindigkeit, möglichst niedrigen Energiebedarf, möglichst hohe Zuverlässigkeit (da verschleißfrei) und möglichst niedrige Herstellungskosten (dank automatisierter Massenfertigung) zu erreichen.
- Die **Peripheriegeräte** (Tastaturen, Drucker, Band-, Platten- und Diskettenleser) müssen *elektromechanisch* realisiert werden und weisen daher, im Vergleich zur Zentraleinheit, eine niedrigere Arbeitsgeschwindigkeit, höheren Energiebedarf, geringere Zuverlässigkeit (durch mechanisch bewegte Teile) und höhere Herstellungskosten auf.

Zwischen beiden Technologien bedarf es einer Anpassung durch geeignete **Ein/Ausgabestrukturen**, die ebenfalls mit Mitteln der Informationsverarbeitung erfolgen kann. Dabei haben sich zwei unterschiedliche Konzepte herausgebildet:

- Das **Kanalkonzept**, bei dem der zentrale Prozessor alle Ein/Ausgabe-Operationen an einen speziellen Kanalprozessor delegiert, der über Steuereinheiten die Ein/Ausgabe-Geräte bedient, so daß topologisch eine Sternstruktur mit Hauptspeicher und dedizierten Prozessoren entsteht.
- Das **Buskonzept**, bei dem alle Partner, z.B. die Prozessor- und Speicherbausteine, in einer Art Konferenzschaltung ein gemeinsames Kommunikationsmedium benutzen, so daß sich topologisch eine Netzstruktur mit Prozessor- und Speicherbausteinen ergibt.

1.6.1 Das Kanalkonzept

Nach dem von Neumannschen Prinzip nimmt der zentrale Prozessor eine vom Befehlszähler bestimmte Sequenz von Maschinenbefehlen aus dem Programmbereich des Hauptspeichers entgegen:

- Entspricht der aktuelle Befehl einer arithmetischen oder logischen Verknüpfung, so wird er vom **Zentralprozessor** selbst ausgeführt.
- Handelt es sich dagegen um einen Ein/Ausgabe-Befehl, so wird seine Ausführung an den **Kanalprozessor** delegiert.

Solange der Kanalprozessor die relativ langsame Peripherie bedient, kann der Zentralprozessor weitere arithmetisch/logische Maschinenbefehle ausführen; beide Prozessoren arbeiten dann unabhängig voneinander. Der durch den Ein/Ausgabe-Befehl angestoßene Kanalprozessor ruft ein geeignetes **Kanalprogramm** auf, das sich ebenfalls im Hauptspeicher befindet. Dazu bringt der Ein/Ausgabe-Befehl die erforderlichen Informationen mit: Die Adresse des Ein/Ausgabe-Gerätes, die Adresse eines Datenbereichs im Hauptspeicher und die Art der Operation, d.h. ob von der Peripherie gelesen oder nach dorthin geschrieben werden soll. Da Ein/Ausgabe-Operationen mit elektromechanischen Geräten in Echtzeit ablaufen müssen, wird ein eigener Kanalbefehlssatz definiert, der vom Maschinenbefehlssatz des Zentralprozessors abweicht. Was den Entwicklungsaufwand betrifft, so entfallen nur etwa 15% auf die Mikroprogrammierung des Zentralprozessors; etwa 85% entfallen auf die Erstellung der Kanalprogramme.

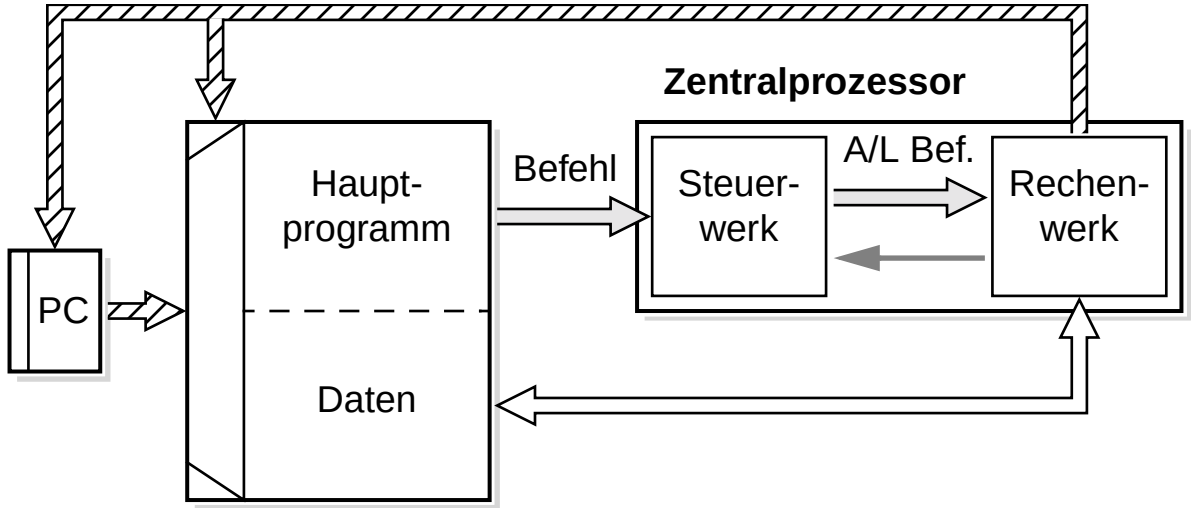


Bild 1.18: Universalrechner (PC: Program Counter, Befehlszähler)

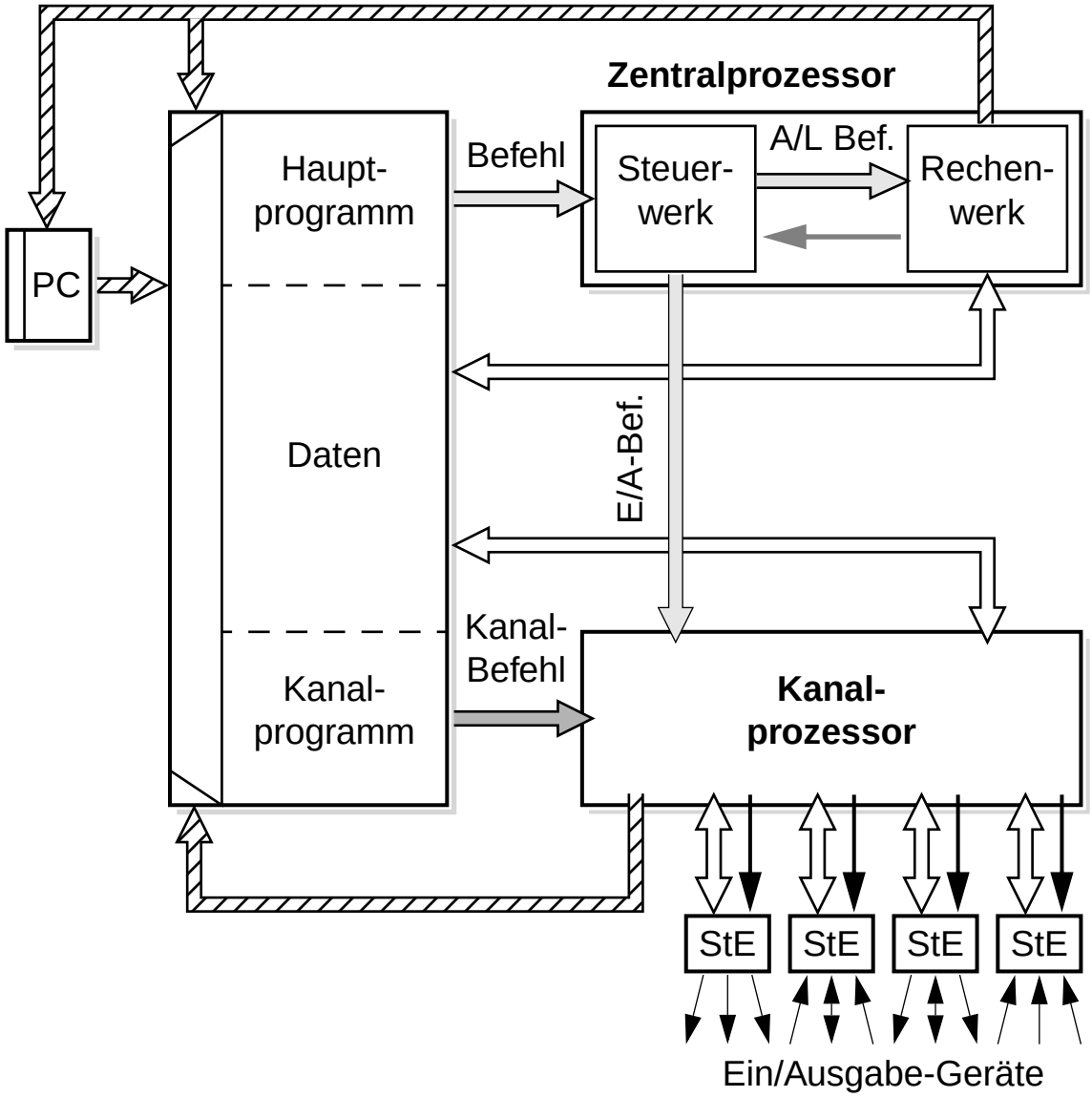


Bild 1.19: Das Kanalkonzept für Universalrechner

1.6.2 Das Buskonzept

Das Buskonzept wurde erstmals für die Ein/Ausgabe binär codierter Meßdaten vorgeschlagen und genormt („IEEE 488 Bus“). Der Bus als Medium zur digitalen Kommunikation kann als Sammelschiene aufgefaßt werden (lat. „omnibus“, d.h. für alle), die allen Teilnehmern in einer Konferenzschaltung gemeinsam zur Verfügung steht, aber abwechselnd und exklusiv genutzt wird. Um Kollisionen von auf den Bus gebrachten Signalen zu vermeiden, sind unterschiedliche Techniken entwickelt worden. Naheliegend ist eine selbsttätige Detektion von Kollisionen: Wie bei einer Diskussionsrunde spricht immer nur ein Teilnehmer; die andern hören zu, bis er aufhört, wobei man eine maximale Redezeit vereinbaren kann. Bei einer Gesprächspause beginnt ein Teilnehmer, der etwas zu sagen oder zu fragen hat, zu sprechen; beginnt zufällig gleichzeitig ein zweiter, so sollten beide verstummen und es nach einer Wartepause zufälliger Länge erneut versuchen. Dieses „Protokoll“ läßt sich auch für Bussysteme technisch nachbilden („Carrier Sense Multiple Access / Collision Detection“, CSMA/CD). Selbstverständlich ist es auch üblich, einen Diskussionsleiter, hier einen Buszuteiler einzusetzen.

Ein Bus umfaßt im wesentlichen drei Arten von Verbindungsleitungen:

- **Datenleitungen** zum Transport der Nutzinformation;
- **Steuerleitungen** zur Ansteuerung der angeschlossenen Peripheriegeräte;
- **Bus-Management-Leitungen** zum Betrieb des Bus selbst.

Die Steuerung eines **Ein/Ausgabe-Bus** und den Betrieb der angekoppelten Peripheriegeräte übernimmt ein (Mikro-)Prozessor. Bei den Peripheriegeräten unterscheidet man drei Betriebsarten:

- LISTEN: Geräte, die nur „hören“, z.B. Stromversorgungen;
- TALK: Geräte, die nur „sprechen“, z.B. Frequenzzähler;
- LISTEN/TALK: Geräte, die „hören und spechen“, z.B. einstellbare Meßgeräte.

Inzwischen hat sich das Buskonzept nicht nur für den Ein/Ausgabebereich bewährt, sondern auch für die interne Kommunikation innnerhalb der Zentraleinheit. Falls sie das Busprotokoll erfüllen, können an einen **Hauptspeicher-Bus** Speichereinheiten unterschiedlicher Technologien angeschlossen werden, wie z.B. Festwertspeicher („Read Only Memory“, ROM), die z.B. unveränderbare Rechnerprogramme enthalten, oder Schreib/Lesespeicher („Random Access Memory“, RAM) für die Zwischenspeicherung von Daten, wobei auch wieder ein oder mehrere (Mikro-)Prozessoren die Steuerung übernehmen. Für den Speicherbetrieb kommen noch Adressleitungen hinzu:

- n **Adressleitungen** zur Adressierung von 2^n Bytes in einem der Speicher
  (n = 16 . . 24 . . 32 bit; 2^n = 64 k . . 16 M . . 4 GByte);
- 2^m **Datenleitungen** zum Transport der Nutzinformation
  (2^m = 4 . . 8 . . 16 . . 32 bit; m = 2 . . 5);
- **Steuerleitungen** zum Betrieb der angeschlossenen Speicherbausteine;
- **Bus-Management-Leitungen** zum Betrieb des Speicherbus selbst.

Schließlich sind noch Bus-zu-Bus-Adapter zu erwähnen, die die Anpassung zwischen unterschiedlichen Busprotokollen vornehmen.

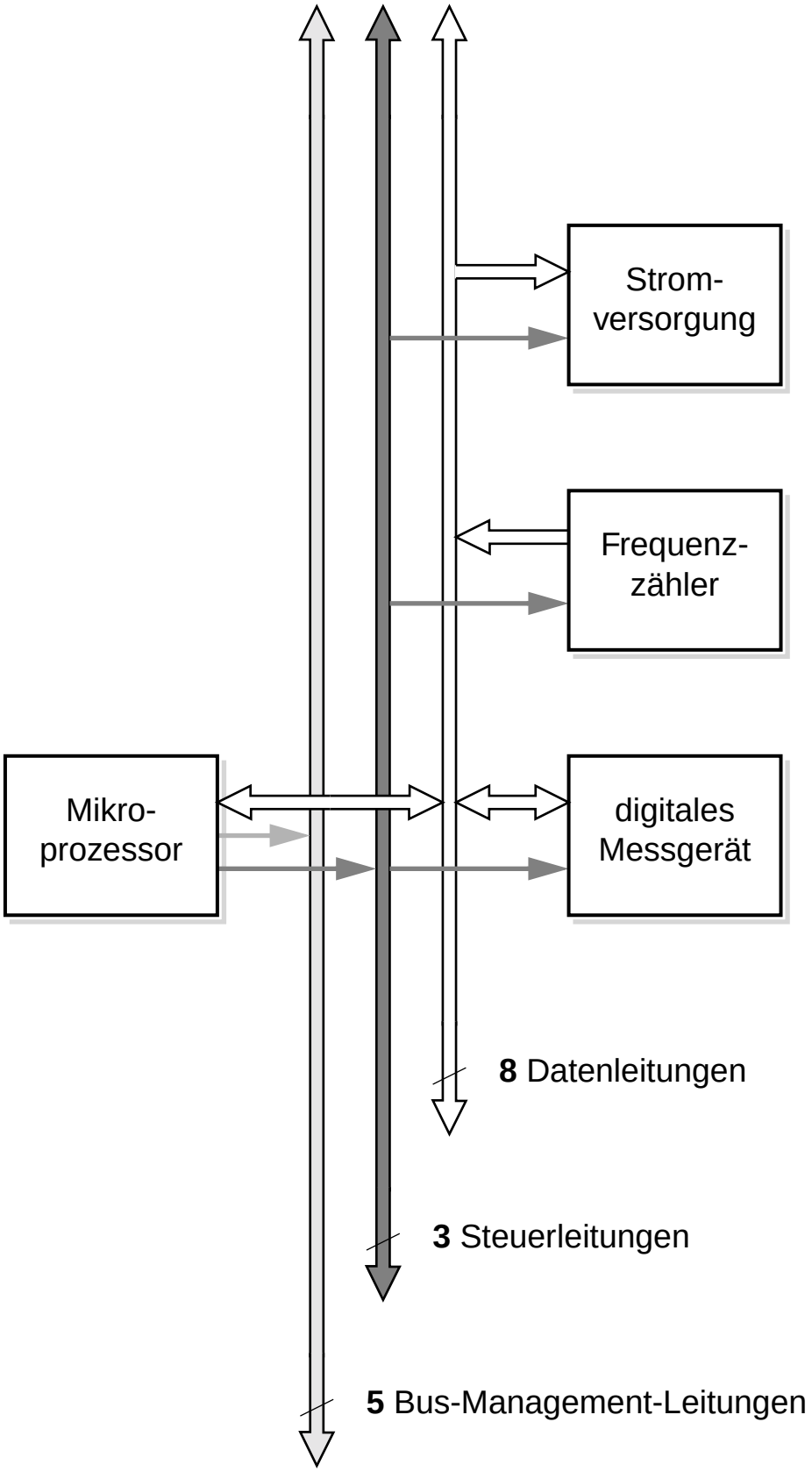


Bild 1.20: Ein/Ausgabe-Bus

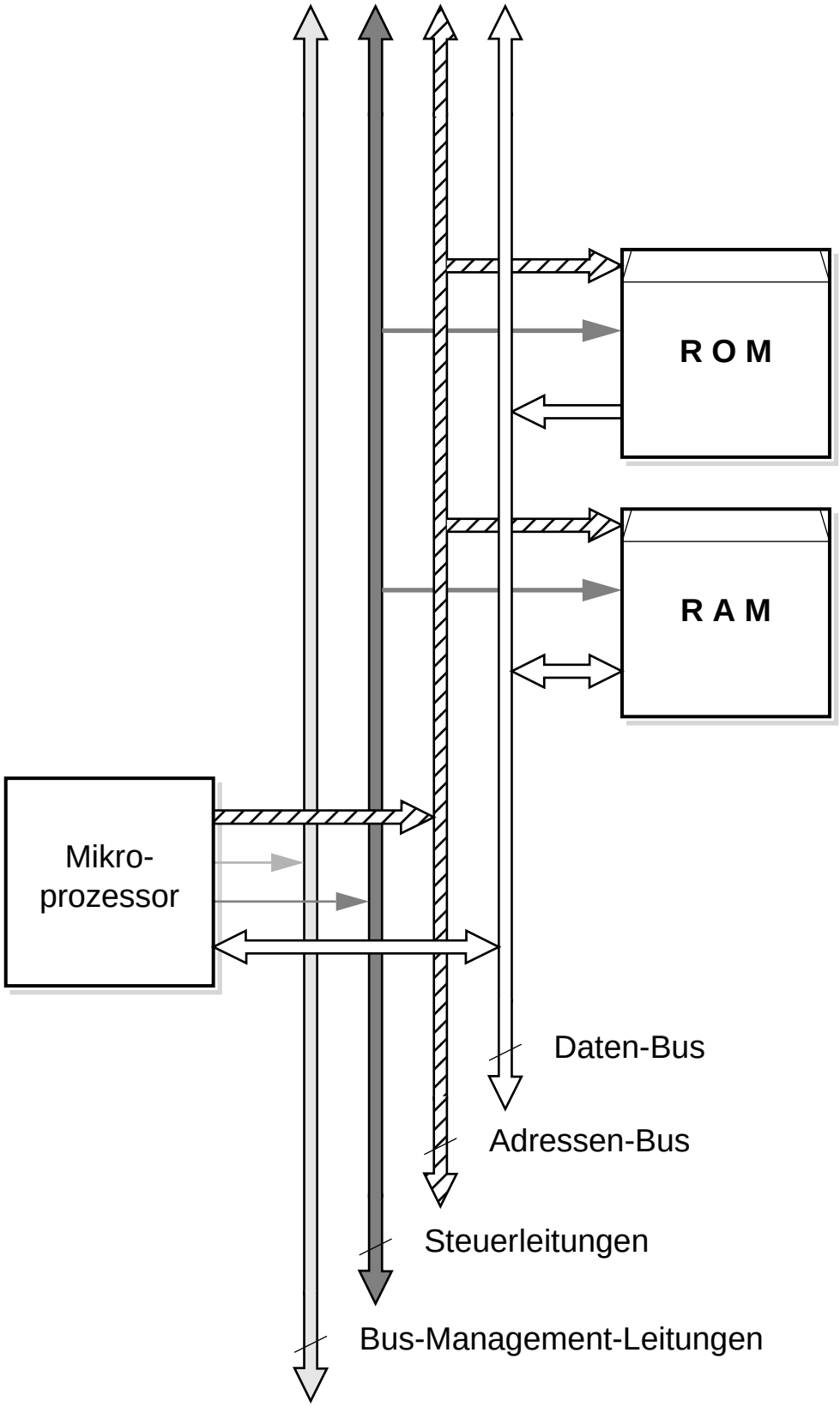
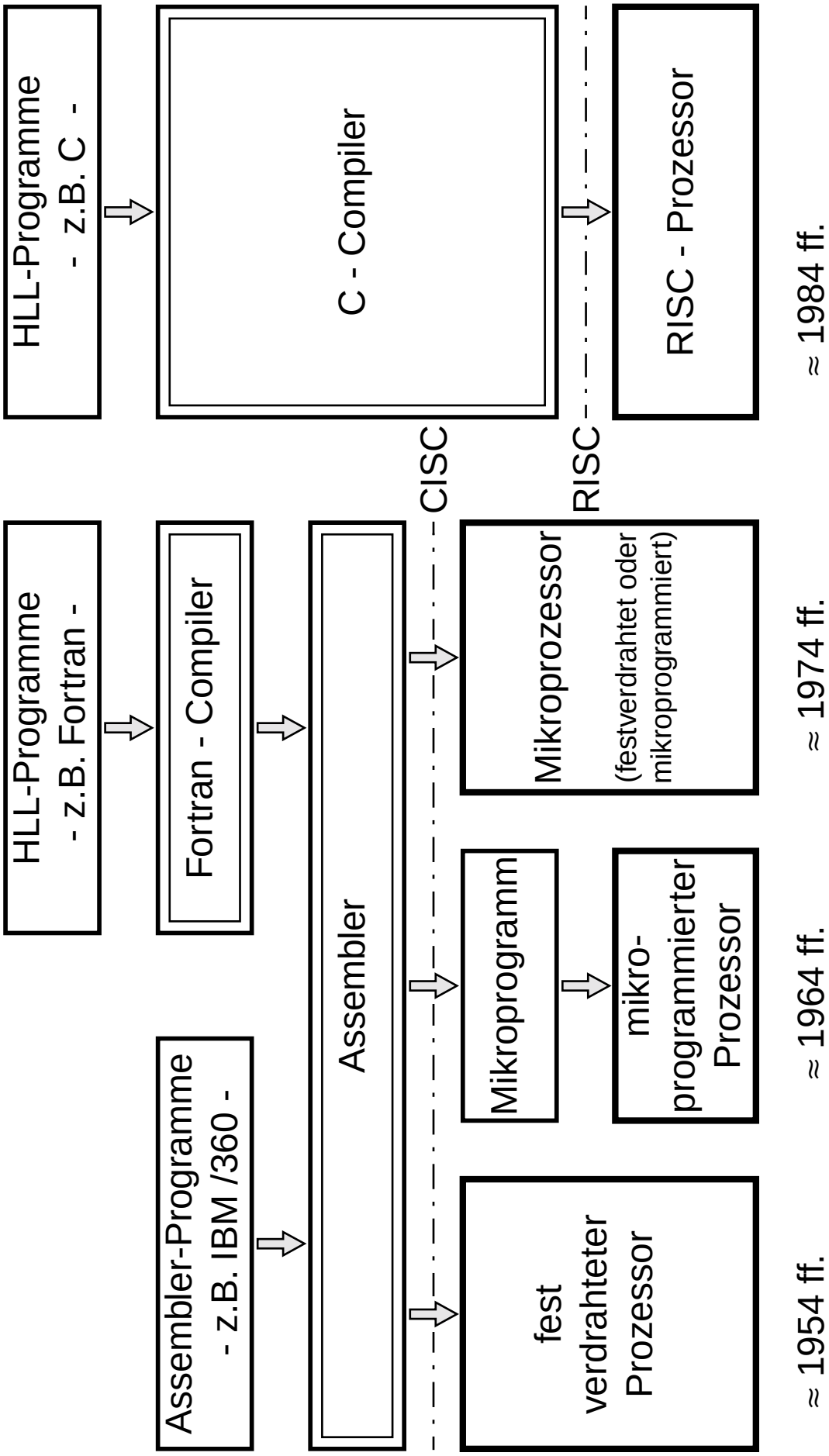


Bild 1.21: Hauptspeicher-Bus


HLL-Programme
- z.B. C -
C - Compiler
RISC - Prozessor
≈ 1984 ff.
CISC
RISC
HLL-Programme
- z.B. Fortran -
Fortran - Compiler
Assembler
Mikroprozessor
(festverdrahtet oder
mikroprogrammiert)
≈ 1974 ff.
Assembler-Programme
- z.B. IBM /360 -
Mikroprogramm
mikro-
programmierter
Prozessor
≈ 1964 ff.
fest
verdrahteter
Prozessor
≈ 1954 ff.

2 Schaltungsebene

Prof. Dr.-Ing. U.G. Baitinger

2.1 Transistoren im Digitalbetrieb

Die technische Informationsverarbeitung beruht auf der logischen Verknüpfung und Speicherung zweiwertig („binär“) codierter Informationen. Um sie mit elektrischen Signalen darzustellen, benötigt man zwei wohlunterschiedene („diskrete“) und somit abzählbare („digitale“) Signalpegel:

Lo ↔ niedriges Potential Hi ↔ hohes Potential

Den beiden Signalpegeln kann man binäre Variablenwerte {0,1} auf zweierlei Weise zuordnen:

positive Logik	negative Logik
Lo ↔ 0 ; Hi ↔ 1	Lo ↔ 1 ; Hi ↔ 0

2.1.1 Kontaktdarstellung von Transistoren

Bereits aus der Relaistechnik stammt die Unterscheidung von zwei Kontakttypen:

- Arbeitskontakt - im Ruhezustand geöffnet, im Arbeitszustand geschlossen (Bild 2.1 links)
- Ruhekontakt - im Ruhezustand geschlossen, im Arbeitszustand geöffnet (Bild 2.2 links)

Ein Schalter besitzt per definitionem genau zwei Stellungen:

- geöffnet = sperrend, hochohmig, niedriger Stromfluß, hoher Spannungsabfall
- geschlossen = leitend, niederohmig, hoher Stromfluß, niedriger Spannungsabfall

Man vergleiche dazu die Kennlinien der beiden Kontakttypen in Bild 2.1 und Bild 2.2 rechts.

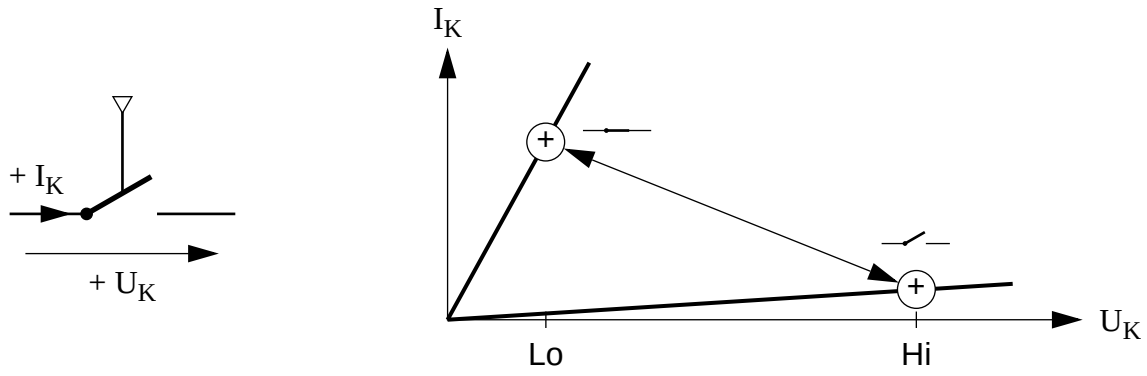


Bild 2.1: Arbeitskontakt mit Kennlinien

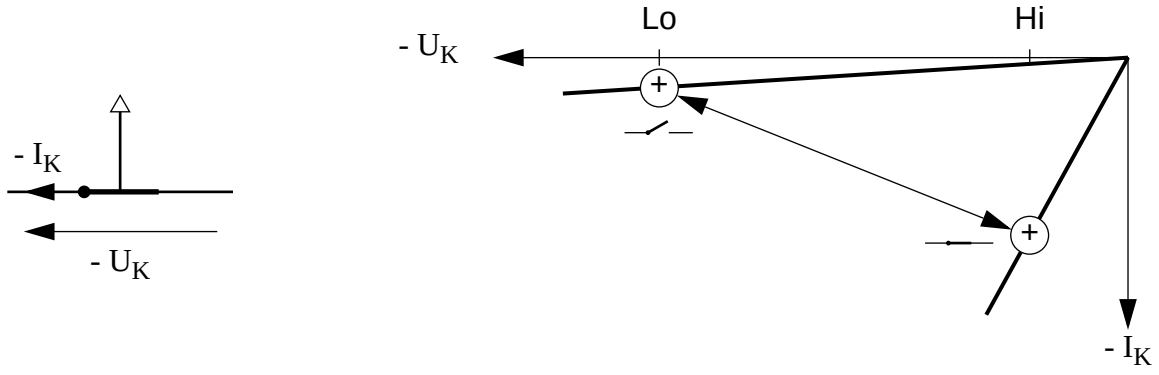


Bild 2.2: Ruhekontakt mit Kennlinien

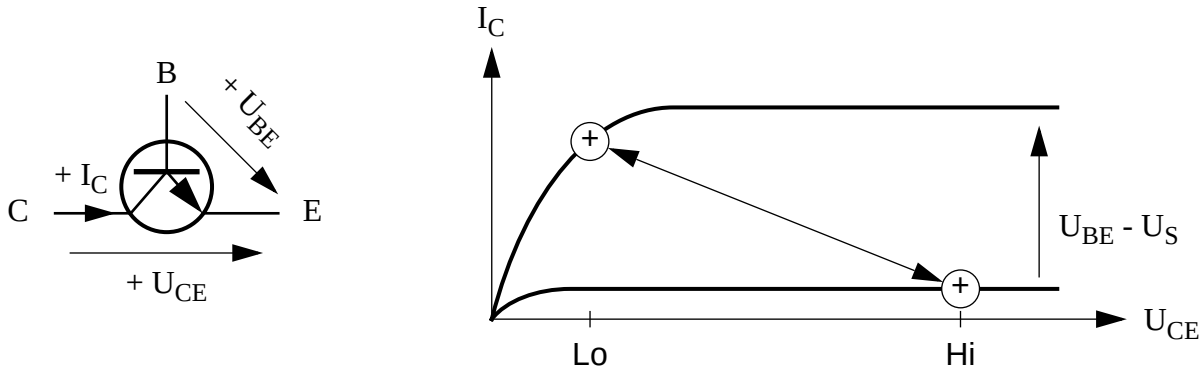


Bild 2.3: Bipolarer NPN-Transistor mit Ausgangskennlinien

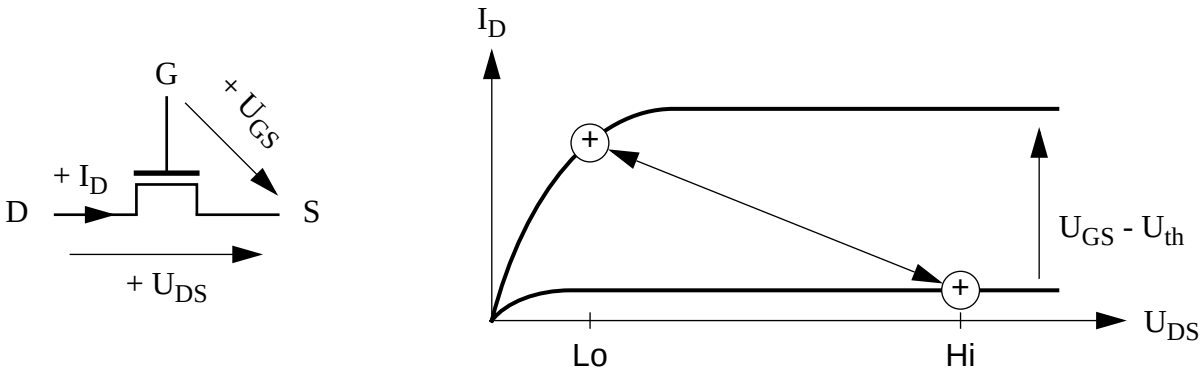


Bild 2.4: N-Kanal MOS-Transistor mit Ausgangskennlinien

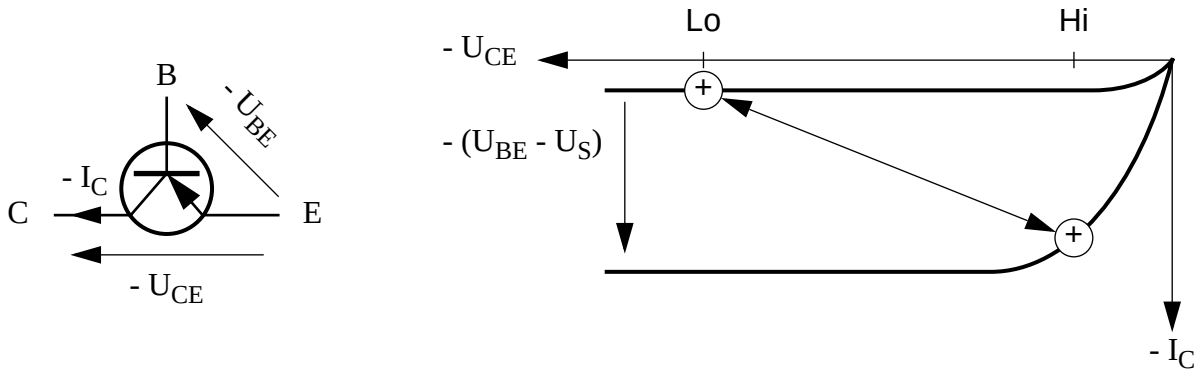


Bild 2.5: Bipolarer PNP-Transistor mit Ausgangskennlinien

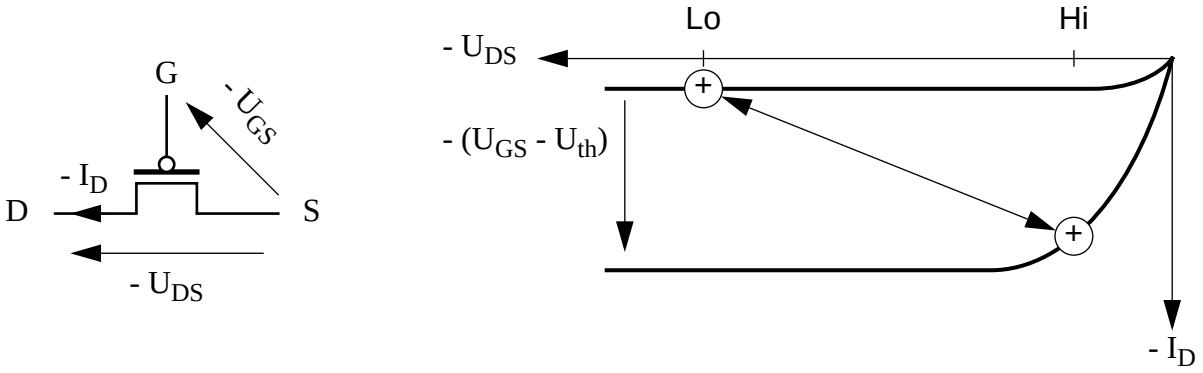


Bild 2.6: P-Kanal MOS-Transistor mit Ausgangskennlinien

Die Ausgangskennlinien sowohl von bipolaren als auch von MOS-Transistoren haben zwar keinen linearen Verlauf (und sie haben verschiedene physikalische Ursachen, obwohl sie sehr ähnlich verlaufen); man erhält jedoch ebenfalls zwei Betriebspunkte für den digitalen Betrieb, in Bild 2.3 bis Bild 2.6 mit (1) und (0) bezeichnet. Stark vereinfacht zeigen im Digitalbetrieb die Transistoren und die mechanistischen Relaiskontakte eine prinzipiell vergleichbare Arbeitsweise:

- Arbeitskontakt ↔ bipolarer Transistor vom NPN-Typ ↔ MOS-Transistor mit N-Kanal
- Ruhekontakt ↔ bipolarer Transistor vom PNP-Typ ↔ MOS-Transistor mit P-Kanal

2.1.2 Lastwiderstand und Lasttransistor

Neben den aktiven Bauelementen, den Schalttransistoren, benötigt man in Digitalschaltungen noch passive Bauelemente, um gegebenenfalls ein aktives Bauelement zu ersetzen, d.h. um einen Signalpegel zu erzeugen, der zwischen „Lo“ und „Hi“ liegt. In bipolarer Technologie lassen sich ohmsche Widerstände mit vertretbarem Aufwand herstellen (Bild 2.7); in MOS-Technologie dagegen muß als Kompromiß ein Transistor zum passiven Zweipol verdrahtet werden (Bild 2.8).

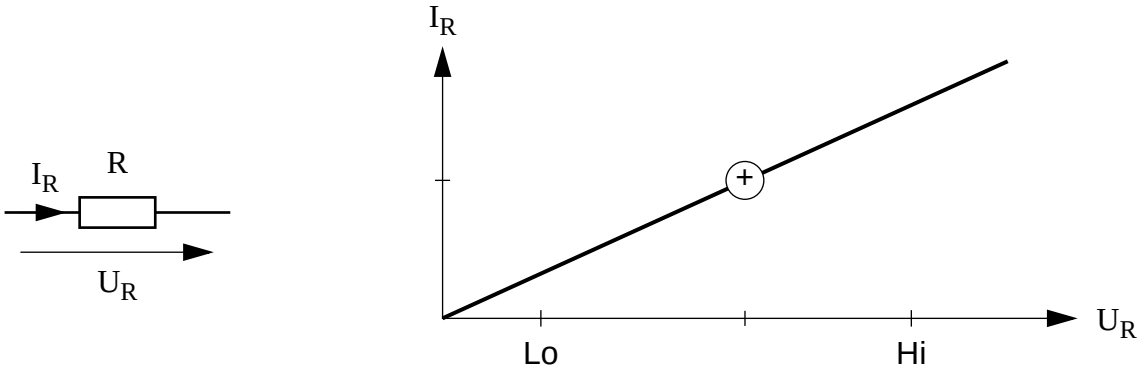


Bild 2.7: Ohmscher Lastwiderstand

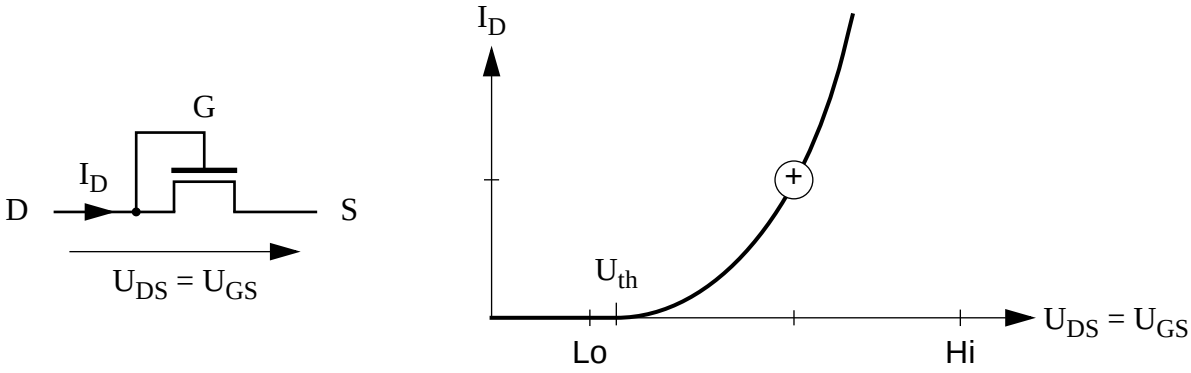


Bild 2.8: MOS-Lasttransistor

2.2 MOS-Technologie

2.2.1 Photolithographie und Dotierung

Der Leitungstyp eines Siliziumkristalls - ob er als Ladungsträger überwiegend Elektronen oder Defektelektronen enthält - wird durch Dotieren mit Donator- bzw. mit Akzeptoratomen beeinflußt. Dabei kann die in den Kristall einzubringende Störstellendichte außerordentlich gering sein. Hochreines Silizium enthält pro Kubikzentimeter $5 \cdot 10^{22}$ Siliziumatome und noch etwa $5 \cdot 10^{12}$ Boratome, die P-Leitung bewirken, so dass auf 10^{10} Kristallatome nur 1 Fremdatom kommt. Eine Dotierung muß diese Restverunreinigung um etwa eine Größenordnung übertreffen, um den Leitungstyp des Siliziummaterials eindeutig festzulegen. Im genannten Beispiel genügt dazu ein einziges Dotierungsatom auf eine Milliarde Siliziumatome, d.h. ein Dotierungsgrad von 10^{-9}. Die gleichmäßige Grunddotierung erfolgt in der Regel während des Einkristallziehens. Aus dem Einkristallstab von derzeit 12…16 cm Durchmesser werden dünne Scheiben („Wafer“) abgesägt und deren Oberfläche geschliffen, gereinigt und glatt poliert. Eine integrierte Schaltung hat auf einem Chip von wenigen Millimetern Kantenlänge Platz. Man kann daher auf einem Wafer einige Hundert integrierte Schaltkreise unterbringen.

Durch energetische Steuerung der Dotierungsvorgänge werden die Eindringtiefen der Dotierungsstoffe bestimmt und damit die *vertikalen* Abmessungen der entstehenden mikroelektronischen Bauelemente. Ihre *horizontale* Geometrie wird durch geeignete Maskierung der zu dotierenden Siliziumscheibe definiert. Hier zeigt sich der große Vorteil, den Silizium gegenüber anderen Halbleitermaterialien bietet: Seine Oberfläche kann mit Wasserdampf relativ einfach zu Quarzglas (SiO_2) oxidiert werden, das für die gebräuchlichen Dotierungsstoffe undurchlässig ist. Mittels photolithographischer Verfahren ätzt man dann Öffnungen in die Quarzglasschicht, so dass nur dort die Siliziumoberfläche ungeschützt zutage tritt; dadurch wird die Kristalloberfläche nur an bestimmten Stellen den Dotierungsstoffen ausgesetzt.

Das Bild zeigt die Verfahrensschritte. Die Oberfläche der Siliziumscheibe (Si) wird zunächst mit einer Quarzglasschicht (SiO_2) überzogen und dann mit Photolack beschichtet. Sodann wird eine Maske auf die Scheibe aufgelegt, die ein Muster aus geschwärzten und durchsichtigen Stellen enthält, entsprechend den herzustellenden integrierten Schaltungen. Die Maske, die auf die Siliziumscheibe aufgelegt wird („Wafer Mask“), besteht aus einer regelmäßigen Wiederholung von Einzelmasken („Chip Masks“), die die Anordnung der Strukturen innerhalb der einzelnen integrierten Schaltkreise definieren. Anschließend wird mit kurzwelligem Licht bestrahlt, wobei die belichteten Stellen des Photolacks polymerisieren. Nach Entfernen der Maske können die unbelichteten, nicht polymerisierten Stellen des Photolacks chemisch weggelöst werden. Dann taucht man die Scheibe in eine spezielle Säure, die die Quarzglasschicht (SiO_2) dort wegätzt, wo sie nicht durch Photolack geschützt ist. Mit einem anderen Lösungsmittel wird sodann der restliche Photolack entfernt. Damit wurde das Muster der Maske auf der Siliziumscheibe abgebildet. Während des anschließenden Dotierungsvorgangs können Fremdatome nur durch die Öffnungen in der Oxidschicht ins Innere des Siliziumkristalls eindringen. Die beim Ätzvorgang erzeugten Oxidöffnungen wachsen bei einer neuerlichen Oxidation wieder zu, um die entstehenden Bauelemente zu „versiegeln“.

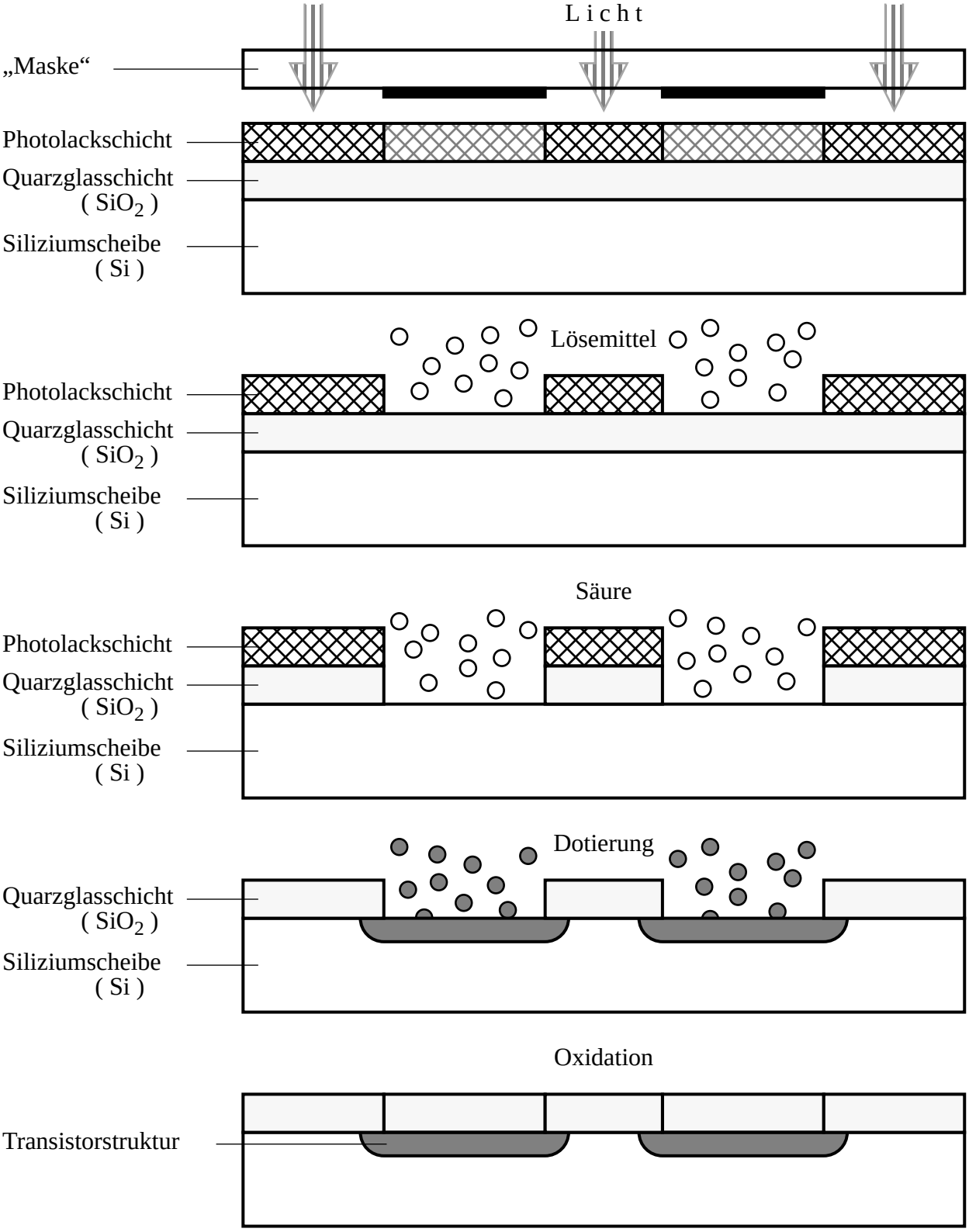


Bild 2.9: Fertigungsschritte zur photolithographischen Abbildung einer „Maske“

2.2.2 NMOS-Technologie

Das folgende Bild zeigt die wichtigsten Fertigungsschritte der Siliziumtechnologie zur Herstellung eines MOS-Transistors mit elektronenleitendem Kanal (NMOS). Dargestellt sind nur die Zwischenzustände nach der photolithographischen Abbildung der einzelnen „Masken“, hier insgesamt vier:

- Drain/Source-Gebiete dotieren
- Gate-Oxid aufwachsen
- Kontaktlöcher öffen
- Metallisierung aufbringen.

Vor jedem der gezeigten Zwischenzustände laufend photolithographische Fertigungsschritte, wie im vorherigen Bild am Beispiel der Dotierung dargestellt, sinngemäß ab.

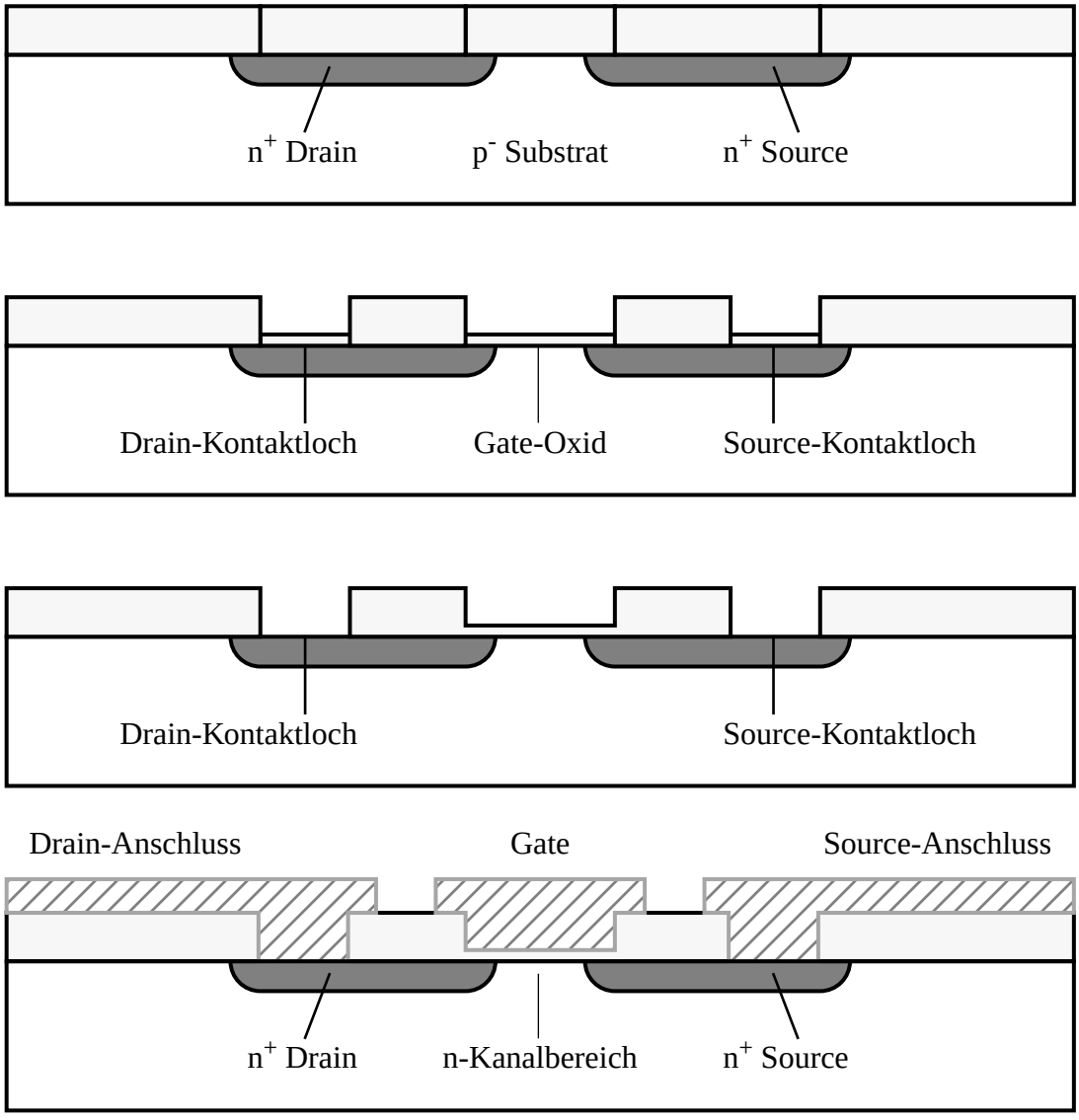


Bild 2.10: Wichtigste Fertigungsschritte eines MOS-Transistors (Querschnitt)

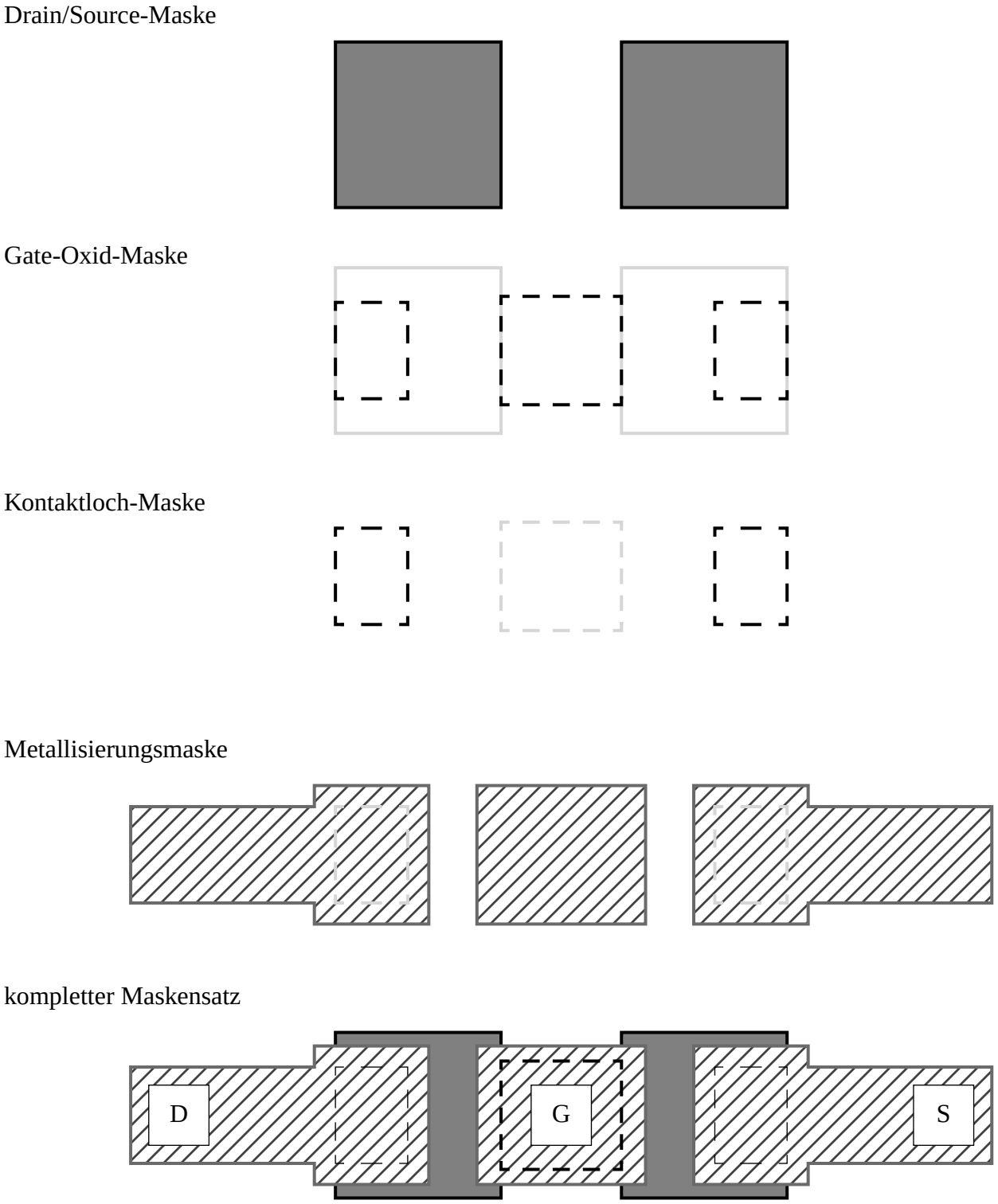


Bild 2.11: Fertigungsmasken eines MOS-Transistors (Draufsicht)

2.2.3 CMOS-Technologie

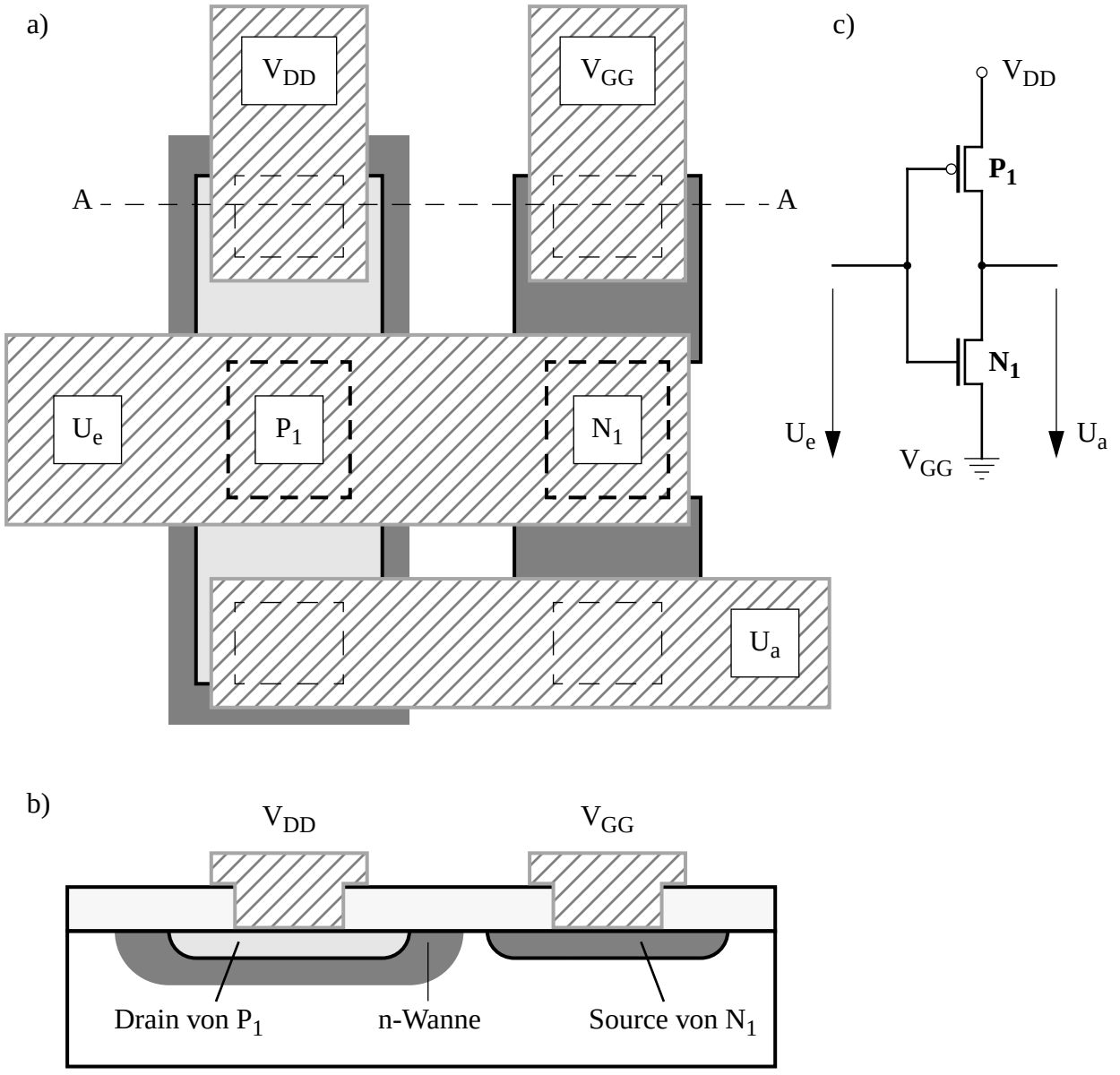


Bild 2.12: CMOS-Inverter; a) Layout / Draufsicht, b) Querschnitt A - A, c) Schaltbild

Das obige Bild zeigt unterschiedliche Ansichten einer einfachen *Inverterschaltung* in CMOS-Technologie, d.h. mit komplementären MOS-Transistoren. Man beachte, dass der NMOS-Transistor N_1 mit elektronenleitendem Kanal direkt in das p-leitende Silizium-Substrat eingebracht werden kann, während für den PMOS-Transistor P_1 mit defektelektronenleitendem Kanal zuvor eine n-leitende „Wanne“ erzeugt werden muss, in die er dann eingebracht wird.

Wie im nachfolgenden Abschnitt über „MOS-Schaltungstechnik“ im Einzelnen erläutert wird, stellt die Inverterschaltung die Grundlage für die komplexeren Verknüpfungs- und Speicherschaltungen in CMOS-Technologie dar.

2.3 Chip-Layout

2.3.1 Full-Custom IC Layout

Es handelt sich um maßgeschneiderte Schaltungen höchsten Integrationsgrades (10^4 ... < 10^6 Gatter pro Chip), die für eine bestimmte Anwendung speziell entwickelt werden, weil sie in großer Stückzahl benötigt werden („voll-kundenspezifische integrierte Schaltungen“).

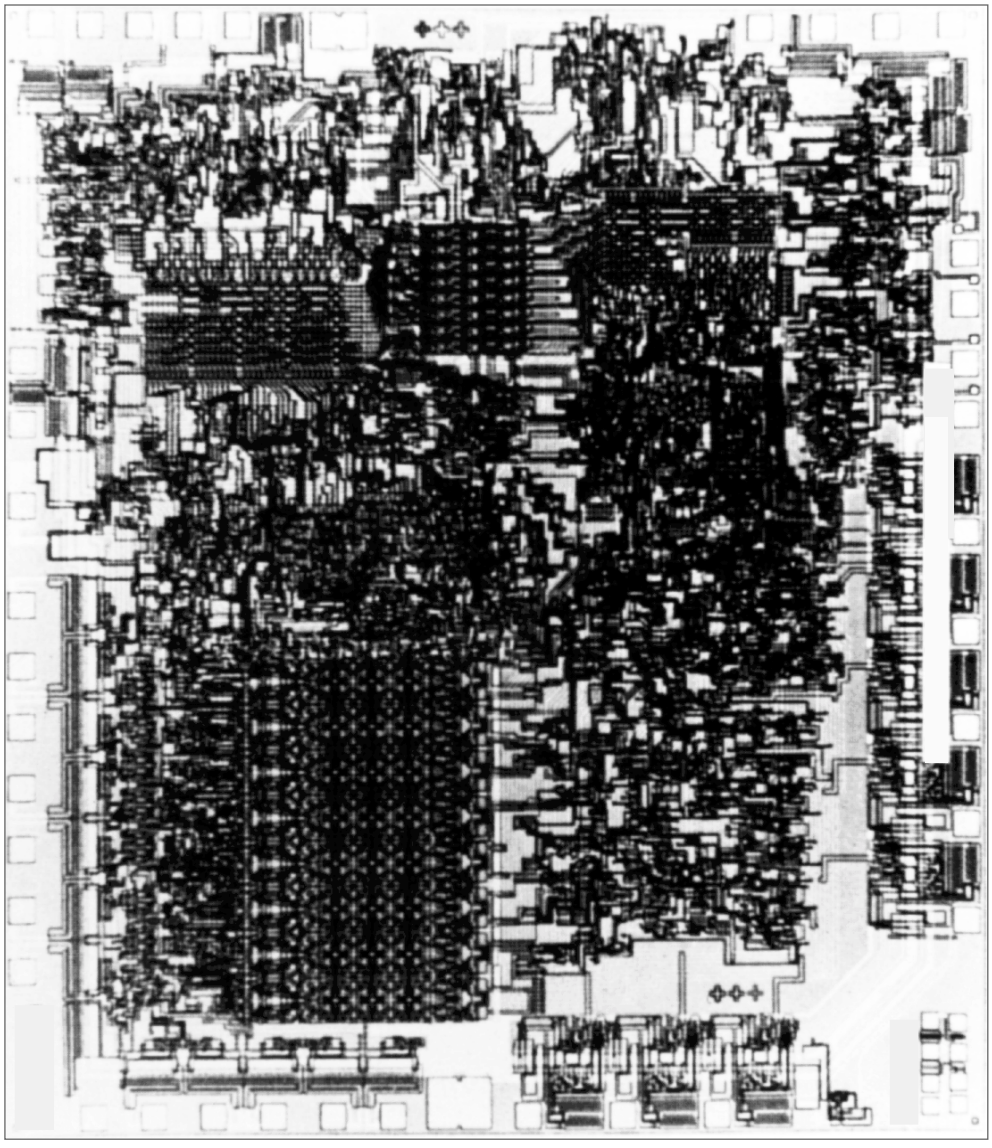


Bild 2.1: Layout einer voll-kundenspezifischen integrierten Schaltung

2.3.2 Semi-Custom IC Layout

„Gate-Arrays“ sind personalisierbare integrierte Schaltungen. Dank ihrer vordefinierten Grundstruktur können fast alle Fertigungsschritte (außer der Verdrahtung) vorab durchgeführt werden. Dem Anwender wird vom Technologieanbieter eine technologiespezifische *Zellbibliothek* bereitgestellt, die wiederverwendbare *Funktionszellen* enthält, die auf die *Basiszellen* des Gate-Arrays abzubilden sind. Nach Abschluß des Entwurfs sind nur noch die entsprechenden Verdrahtungsmuster zu fertigen („semi-kundenspezifische integrierte Schaltungen“).

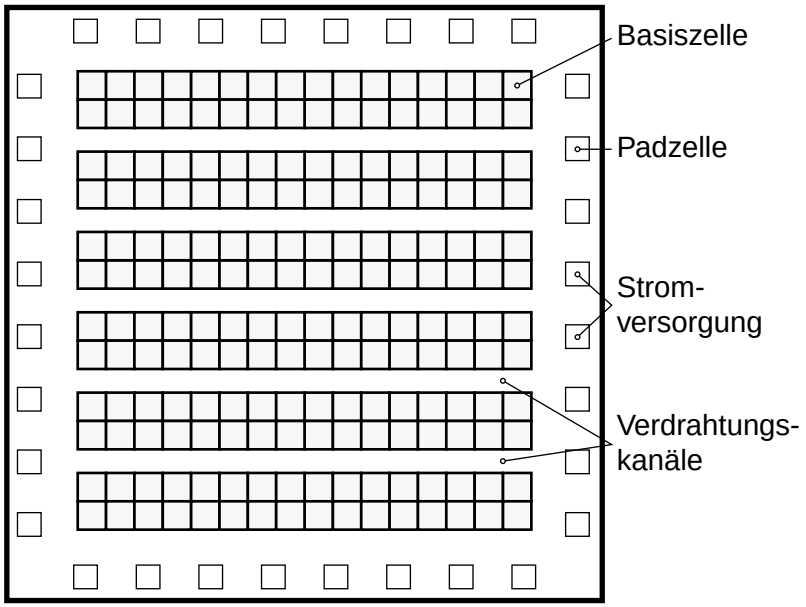


Bild 2.2: Grundstruktur („Master-Layout“) eines Gate-Arrays mit Säulenstruktur

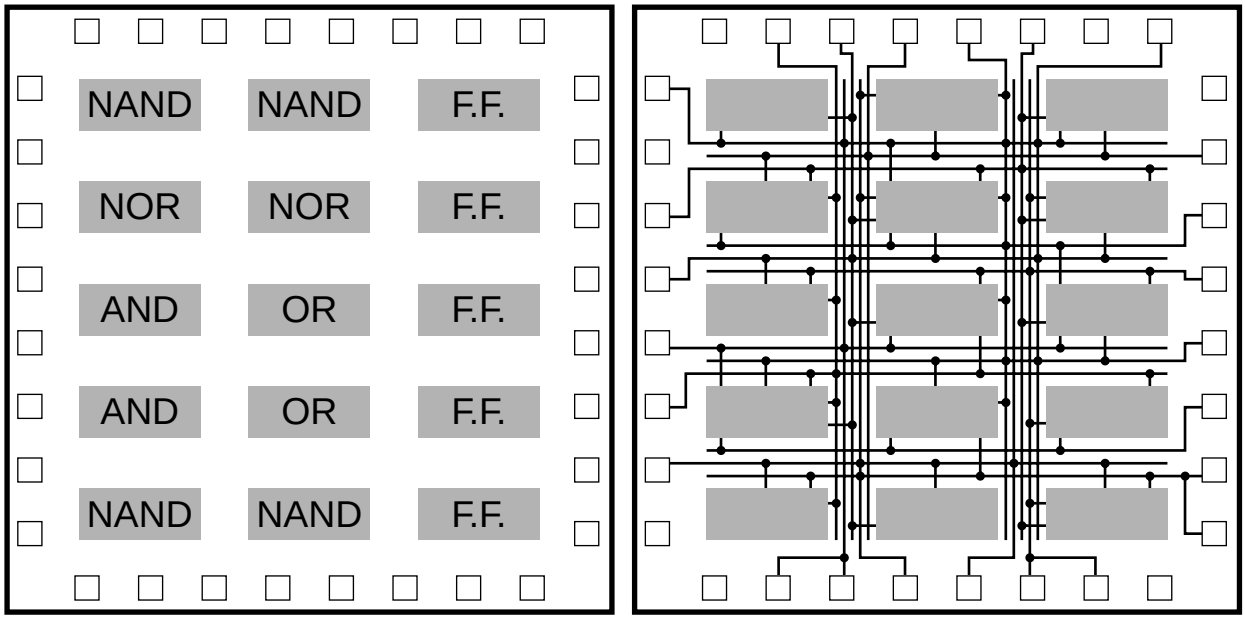


Bild 2.3: Gate-Array mit Matrixstruktur, zell-orientierter Entwurf
a) Platzierung („Intrazell-Verdrahtung“) b) Verdrahtung („Interzell-Verdrahtung“)

2.4 MOS-Schaltungstechnik

MOS steht für “Metal-Oxide-Silicon“, d.h. für Transistoren mit MOS-Schichtstruktur und ihre Verwendung in mikroelektronischen Schaltungen. Wir wollen uns hier auf den Anreicherungstyp mit isolierter Gate-Elektrode beschränken. Beim Schaltungsentwurf mit MOS-Transistoren sind drei charakteristische Eigenschaften besonders zu berücksichtigen:

- Für MOS-Schaltungen in integrierter Technologie ist typisch, daß nicht nur die aktiven Schalttransistoren sondern auch passive *„Arbeitswiderstände“* durch MOS-Transistoren realisiert werden müssen, da bei MOS-Fertigungsprozessen die Herstellung hochohmiger Widerstände problematisch ist.
- Ferner ist zu berücksichtigen, daß ein MOS-Transistor erst leitet, wenn seine Gate-Source-Spannung den Wert einer *Schwellspannung* U_{th} (“threshold voltage“) überschreitet. Somit ist der Einbau eines leitenden MOS-Transistors stets mit einer Pegelverschiebung um den Betrag dieser Schwellspannung verbunden, was bei Bedarf durch entsprechende Erhöhung der Betriebsspannung ausgeglichen werden muß.
- Schließlich sollte noch beachtet werden, daß jeder MOS-Transistor aufgrund seiner Bau- und Wirkungsweise mit einem inhärenten *MOS-Kondensator* von nicht zu vernachlässigender Kapazität behaftet ist, die bei Schaltvorgängen aufgeladen bzw. entladen werden muß.

Letztere Eigenart ist aber nicht nur störend. Vielmehr ist ein MOS-Transistor in der Lage, Ladung zu speichern und damit seinen augenblicklichen Betriebszustand kurzzeitig beizubehalten, auch wenn die ihn ansteuernden Schaltkreise bereits einen anderen Zustand angenommen haben. Dieser Effekt kann ausgenützt werden, um neben *statisch* betriebenen Logik- und Speicherschaltungen, die von Gleichspannungsquellen gespeist werden, auch *dynamische* zu entwickeln, die mit Taktgeneratoren betrieben werden. Dadurch werden Gleichströme vermieden, was zu extrem niedrigen Verlustleistungen führt. Nicht zuletzt deshalb ermöglicht die dynamische MOS-Schaltungstechnik den Aufbau höchstintegrierter Schaltungen (“Very Large Scale Integration“, VLSI) mit mehreren Millionen Transistoren pro Chip.

So beruhen statische MOS-Speicherzellen auf dem Prinzip des bistabilen Flipflops, die dynamischen auf der Ladungsspeicherung in Kondensatoren. Dynamische MOS-Speicher (“Dynamic Random Access Memories“, DRAM) haben nicht nur eine niedrigere Verlustleistung, sondern auch einen niedrigeren Flächenbedarf pro Speicherzelle, so daß sich größere Speicher auf einem Chip integrieren lassen. Statische MOS-Speicher (“Static Random Access Memories“, SRAM) dagegen haben eine niedrigere Zugriffszeit als die dynamischen, da sie keine Auffrischzyklen benötigen.

2.4.1 Statische MOS-Logik

Für den Digitalbetrieb von MOS-Transistoren vom N-Kanaltyp („NMOS-Transistoren“) gelten zwei wohlunterschiedene Spannungspegel mit der Schwellspannung U_{thN} des N-Kanals als Trennwert sowie einer oberen Grenze, die u.a. von der Betriebsspannung von $U_B > 0$, und einer unteren Grenze U_u, deren Wert wie nachfolgend gezeigt von der Dimensionierung der Schaltung abhängt:

$$U_u \le \textbf{Lo} < U_{thN} \qquad\qquad U_{thN} < \textbf{Hi} \le (U_B - U_{thN})$$

Es gelten folgende Zahlenwerte:

$U_B \approx + 2{,}5\ V$ $\qquad$ $U_{thN} \approx + 1{,}0\ V\ (\rightarrow + 0{,}5\ V)$

Die im folgenden Bild gezeigte Inverterschaltung ist eine elementare Verknüpfungsschaltung mit einer minimalen Anzahl von Bauelementen. Sie besteht aus einem Schalttransistor N_1 und einem zum Zweipol verdrahteten Lasttransistor N_L, da in dieser Technologie bekanntlich keine hochohmigen Widerstände hergestellt werden können. Ferner darf die Eingangskapazität C_a der gleichartigen MOS-Schaltkreise, die von der Inverterschaltung angesteuert werden, hier nicht vernachlässigt werden.

Legt man an die Eingangsklemme des NMOS-Inverters einen Potentialpegel „Lo“, der niedriger ist als der Wert der Schwellspannung, so sperrt der Schalttransistor N_1. Die Lastkapazität C_a wird dann über den leitenden Lasttransistor N_L auf den Pegel „Hi“ aufgeladen:

Lo: $U_e < U_{thN}$

Hi: $U_a = (U_B - U_{thN}) > U_{thN}$

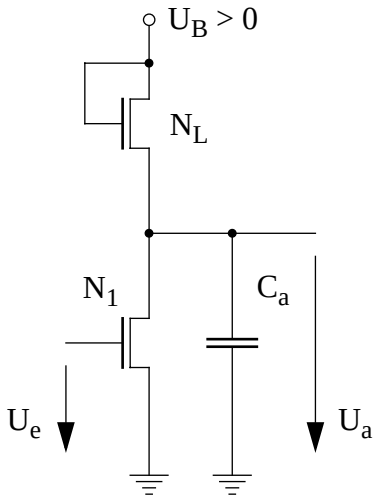


Bild 2.4: Inverterschaltung in statischer NMOS-Logik

Sobald die Ausgangsspannung auf den Pegel „Hi“ angestiegen ist, erreicht die Gate-Source-Spannung des bisher leitenden Lasttransistors N_L den Wert seiner Schwellspannung, d.h. er sperrt, so daß sein Sourcepotential U_a nicht weiter ansteigen kann. Der sich einstellende Ausgangspegel U_a kann direkt als Eingangspegel für gleichartige Schaltungsstufen dienen.

Nimmt im andern Fall der Eingangspegel den Wert „Hi“ an, liegt er also höher als die Schwellspannung, so leitet der Schalttransistor N_1 und entlädt die Lastkapazität C_a bis auf den Pegel „Lo“, der durch das Spannungsteilerverhältnis aus dem Schalttransistor N_1 und dem Lasttransistor N_L gegeben ist. Dabei muß N_1 niederohmig sein, d.h. sein Kanalbereich muß breit und kurz ausgelegt werden, N_L dagegen hochohmig, d.h. lang und schmal, da für den Pegel „Lo“ gelten muß:

Hi: $U_e = (U_B - U_{thN}) > U_{thN}$ $\qquad$ **Lo:** $U_a = U_{DS1} < U_{thN}$

Verwendet man MOS-Transistoren vom P-Kanaltyp („PMOS-Transistoren“), so gelten für deren Digitalbetrieb ebenfalls zwei wohlunterschiedene Spannungspegel, die aus physikalischen Gründen zu denen der NMOS-Logik komplementär sind: mit der Schwellspannung U_{thP} als Trennwert sowie einer unteren Grenze, die u.a. von der Betriebsspannung $U_B' < 0$ und einer oberen Grenze U_o, deren Wert wie im Fall der NMOS-Logik von der Dimensionierung der Schaltung abhängt:

$(U_B' - U_{thP}) \leq \mathbf{Lo} < U_{thP}$ $\qquad$ $U_{thP} < \mathbf{Hi} \leq U_o$

Es gelten folgende Zahlenwerte:

$U_B' \approx - 2{,}5\ V$ $\qquad$ $U_{thP} \approx - 1{,}0\ V\ (\rightarrow - 0{,}5\ V)$

Das folgende Bild zeigt eine elementare Inverterschaltung mit PMOS-Transistoren. Sie besteht aus einem Schalttransistor P_1 und einem zum Zweipol verdrahteten Lasttransistor P_L. Man erkennt die Symmetrie zu der im vorherigen Bild gezeigten Inverterschaltung mit NMOS-Transistoren. Da die NICHT-Funktion bekanntlich zu sich selbst komplementär ist, realisieren beide Inverterschaltungen dieselbe Schaltfunktion.

Legt man an die Eingangsklemme des PMOS-Inverters einen Potentialpegel „Hi“, der höher liegt als der (negative) Wert der Schwellspannung des P-Kanals, so sperrt der Schalttransistor P_1. Die Lastkapazität C_a wird dann über den leitenden Lasttransistor P_L auf den Pegel „Lo“ geladen:

Hi: $U_e > U_{thP}$

Lo: $U_a = (U_B' - U_{thP}) < U_{thP}$

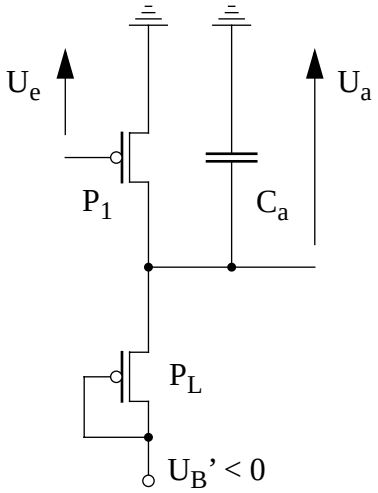


Bild 2.5: Inverterschaltung in statischer PMOS-Logik

Sobald die Ausgangsspannung auf den Pegel „Lo“ abgesunken ist, erreicht die Gate-Source-Spannung des bisher leitenden Lasttransistors P_L den Wert der Schwellspannung, d.h. er sperrt, so daß sein Sourcepotential U_a nicht weiter absinken kann. Der sich einstellende Ausgangspegel U_a kann direkt als Eingangspegel für gleichartige Schaltungsstufen dienen.

Nimmt im andern Fall der Eingangspegel den Wert „Lo“ an, liegt er also niedriger als der (negative) Wert der Schwellspannung, so leitet der Schalttransistor P_1 und entlädt (!) die Lastkapazität C_a bis auf den Pegel „Hi“, der durch das Spannungsteilerverhältnis aus dem Schalttransistor P_1 und dem Lasttransistor P_L gegeben ist. Dabei muß P_1 niederohmig sein, d.h. sein Kanalbereich muß breit und kurz ausgelegt werden, P_L dagegen hochohmig, d.h. lang und schmal, da für den Pegel „Hi“ gelten muß:

Lo: $U_e = (U_B' - U_{thP}) < U_{thP}$ **Hi:** $U_a = U_{DS1} > U_{thP}$

2.4.2 Statische CMOS-Logik

CMOS steht für “Complementary Metal-Oxide-Silicon“, d.h. für die Verwendung von MOS-Transistoren beider, zueinander *komplementärer* N- bzw. P-Kanaltypen in einer gemeinsamen Schaltung. Auch hierbei handelt es sich um eine statische Logik, obwohl keine Gleichströme fließen, da die zueinander komplementären MOS-Transistoren stets im Wechsel leiten bzw. sperren. Um den Preis eines komplizierteren Fertigungsprozesses kombiniert man NMOS- und PMOS-Transistoren und gewinnt dabei eine Schaltkreisfamilie mit extrem niedriger Verlustleistung.

Um auch kürzere Umschaltzeiten zu erreichen, kombiniert man die niederohmigen (!) Schalttransistoren des NMOS-Inverters von Bild 2.4 mit denen des PMOS-Inverters von Bild 2.5 und erhält so die in Bild 2.6 gezeigte elementare Inverterschaltung in CMOS-Technik.

Für den Digitalbetrieb von CMOS-Schaltungen gelten zwei wohlunterschiedene Spannungspegel mit den Schwellspannungen U_{thN} des N-Kanals bzw. U_{thP} des P-Kanals als Trennwerten sowie dem Bezugspotential („Erdpotential“) als unterer und der Betriebsspannung U_B als oberer Grenze, d.h. die Grenzwerte hängen im Gegensatz zur NMOS- oder PMOS-Logik nicht von der Dimensionierung der Schaltung ab:

$0 \le \mathbf{Lo} < [U_{thN} = (U_B + U_{thP})]$ $\qquad$ $[U_{thN} = (U_B + U_{thP})] < \mathbf{Hi} \le U_B$

Es sei an die Zahlenwerte erinnert:

$U_{thN} \approx + 1{,}0\ V$ $\qquad$ $U_{thP} \approx - 1{,}0\ V$

$U_B \approx + 2{,}5\ V$

Legt man an die Eingangsklemme des im nebenstehenden Bild gezeigten CMOS-Inverters einen Potentialpegel „Lo“, der niedriger liegt als der Wert der Schwellspannung $U_{thN} > 0$, so sperrt der Schalttransistor N_1. Gleichzeitig wird der Schalttransistor P_1 leitend, da sein Gatepotential dann niedriger liegt als der (negative) Wert der Schwellspannung $U_{thP} < 0$. Da er niederohmig ausgelegt ist und N_1 sperrt, lädt P_1 die Lastkapazität C_a relativ rasch und vollständig auf den Wert der Betriebsspannung U_B auf:

Lo: $U_e < [U_{thN} = (U_B + U_{thP})]$

Hi: $U_a = U_B$

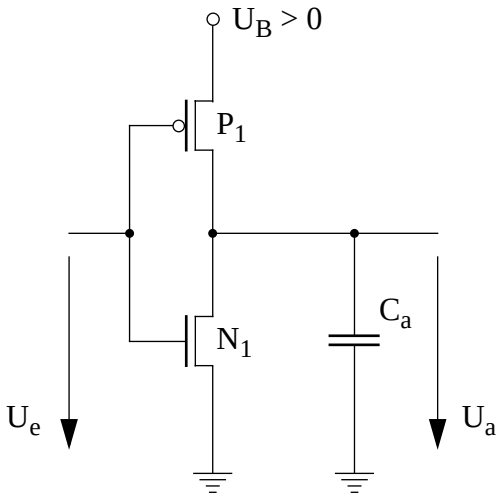


Bild 2.6: Inverterschaltung in statischer CMOS-Logik

Der sich einstellende Ausgangspegel „Hi“ kann direkt als Eingangspegel für gleichartige Schaltungsstufen dienen. - Nimmt im andern Fall der Eingangspegel den Wert „Hi“ an, liegt er also höher als die Schwellspannung des N-Kanals, so leitet der ebenfalls niederohmig ausgelegte Schalttransistor N_1 und entlädt die Lastkapazität C_a relativ rasch und vollständig („Lo“), da jetzt der Schalttransistor P_1 sperrt:

Hi: $U_e = U_B > [U_{thN} = (U_B + U_{thP})]$ $\qquad$ **Lo:** $U_a = 0$

Nach Bild 2.7 erhält man durch die Kombination der topologisch *komplementären* Reihen- bzw. Parallelschaltungen von technologisch *komplementären* CMOS-Schalttransistoren die NOR- und die NAND-Schaltung in statischer CMOS-Logik.

Die im folgenden Bild links gezeigte „NOR-Schaltung“ entspricht im einen Betriebszustand dem obigen CMOS-Inverter mit leitendem NMOS-Transistor (bei gleichzeitig sperrendem PMOS-Transistor), wenn das Potential von *mindestens einer* der Eingangsklemmen über die Schwellspannung U_{thN} angehoben wird. Dadurch wird die Lastkapazität C_a über mindestens einen der parallel ge-

schalteten, niederohmig ausgelegten NMOS-Transistoren relativ rasch und vollständig entladen, da mindestens einer der in Reihe geschalteten PMOS-Transistoren sperrt:

Hi: U_{e1} *oder* $U_{e2} > U_{thN}$ **Lo:** $U_{a1} = 0 < U_{thN}$ (0.1)

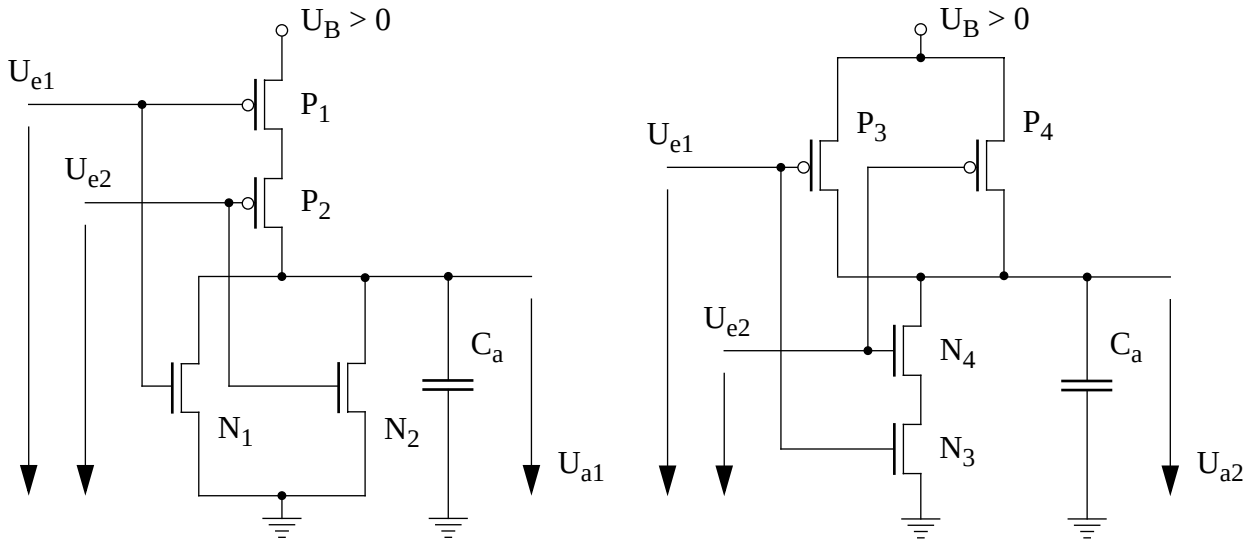


Bild 2.7: Realisierung von Schaltfunktionen in statischer CMOS-Logik
a) NOR-Schaltung (positive Logik) b) NAND-Schaltung (positive Logik)

Im anderen Betriebszustand entspricht die Schaltung dem Inverter mit sperrendem NMOS-Transistor (bei gleichzeitig leitendem PMOS-Transistor), wenn das Potential *aller* Eingangsklemmen unter die Schwellspannung U_{thN} abgesenkt wird. Dadurch wird die Reihenschaltung der niederohmig ausgelegten PMOS-Transistoren leitend und lädt die Lastkapazität C_a relativ rasch und vollständig auf den Wert der Betriebsspannung auf, da alle parallel geschalteten NMOS-Transistoren sperren:

Lo: U_{e1} *und* $U_{e2} < U_{thN}$ **Hi:** $U_{a1} = U_B > U_{thN}$ (0.2)

Somit realisiert die im obigen Bild links gezeigte CMOS-Schaltung in positiver Logik eine NOR-, in negativer eine NAND-Funktion.

Die im obigen Bild rechts gezeigte „NAND-Schaltung“ entspricht im einen Betriebszustand dem CMOS-Inverter nach Bild 2.6 mit leitendem NMOS-Transistor (bei gleichzeitig sperrendem PMOS-Transistor), wenn das Potential *aller* Eingangsklemmen über die Schwellspannung U_{thN} angehoben wird. Dadurch wird die Reihenschaltung der niederohmig ausgelegten NMOS-Transistoren leitend und entlädt die Lastkapazität C_a relativ rasch und vollständig, da alle parallel geschalteten PMOS-Transistoren sperren:

Hi: U_{e1} *und* $U_{e2} > U_{thN}$ **Lo:** $U_{a2} = 0 < U_{thN}$ (0.3)

Im anderen Betriebszustand entspricht die Schaltung dem Inverter mit sperrendem NMOS-Transistor (bei gleichzeitig leitendem PMOS-Transistor), wenn das Potential von *mindestens einer* der Eingangsklemmen unter die Schwellspannung U_{thN} abgesenkt wird. Dadurch wird die Lastkapazität C_a über mindestens einen der parallel geschalteten, niederohmig ausgelegten PMOS-Transi-

storen relativ rasch und vollständig auf den Wert der Betriebsspannung aufgeladen, da mindestens einer der in Reihe geschalteten NMOS-Transistoren sperrt:

Lo: U_{e1} *oder* $U_{e2} < U_{thN}$ **Hi:** $U_{a2} = U_B > U_{thN}$ (0.4)

Somit realisiert die im vorherigen Bild rechts gezeigte CMOS-Schaltung in positiver Logik eine NAND-, in negativer eine NOR-Funktion.

Fassen wir die Vorteile der CMOS-Schaltkreisfamilie zusammen:

- Niedrige Verlustleistung, da über die zueinander komplementären, im Gegentakt leitenden bzw. sperrenden MOS-Transistoren keine Gleichströme fließen. Beim Umschalten wird nur Wechselstromverlustleistung umgesetzt.
- Hohe Schaltgeschwindigkeit durch die Kombination ausschließlich niederohmiger Schalttransistoren, d.h. dank der Vermeidung hochohmiger Lastwiderstände.
- Hoher Störabstand durch Aufladen der zu treibenden Lastkapazität auf den vollen Wert der Betriebsspannung bzw. deren vollständiges Entladen dank der Vermeidung eines Spannungsteilers.

2.4.3 Dynamische Domino-Logik

Wegen der inhärenten Gatekapazitäten von MOS-Transistoren können die damit aufgebauten Schaltungen auch *dynamisch* betrieben werden. Sie entwickeln dann keine statische Verlustleistung, da im Betrieb keine Gleichströme fließen. Die hier vorgestellte Domino-Schaltungstechnik wird in CMOS-Technologie realisiert, d.h. sie verwendet MOS-Transistoren vom N-Kanal- und vom P-Kanaltyp. Deshalb können die Schaltungen mit einem einzigen Taktsignal betrieben werden.

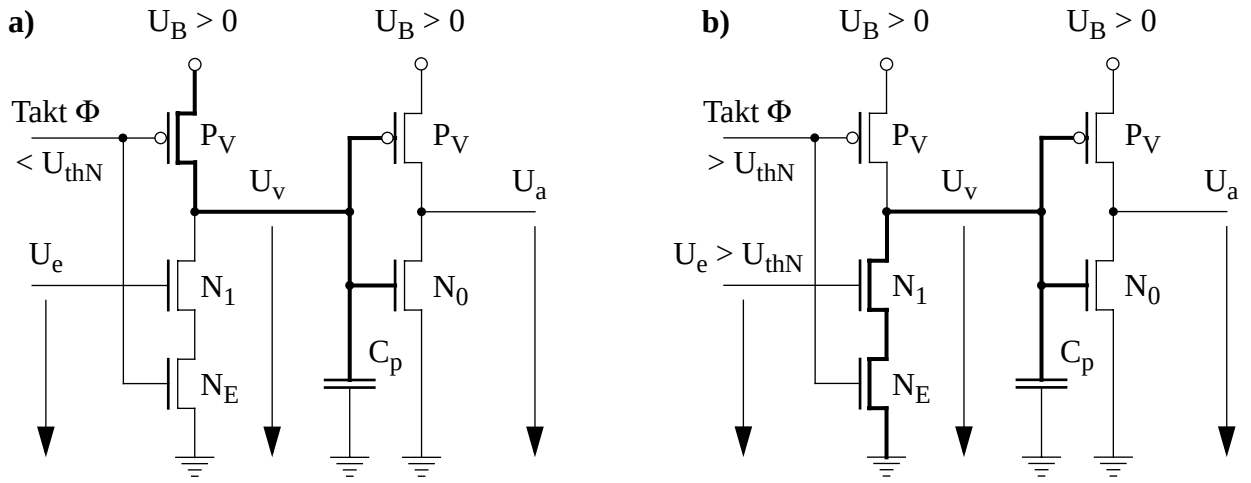


Bild 2.8: Grundschaltung in Domino-Logik.
a) unbedingte Vorladephase; b) bedingte Evaluierungsphase

Den Aufbau einer Grundschaltung in Domino-Logik zeigt das obige Bild. Ein PMOS-Transistor P_V und ein NMOS-Transistor N_E werden gemeinsam von einem Taktsignal Φ angesteuert; sie leiten/sperren jeweils wechselseitig, so daß kein Gleichstrom fließt. Zu diesem, im Gegentakt arbeitenden CMOS-Transistorpaar ist der von einem Eingangssignal U_e angesteuerte Schalttransistor N_1 in

Reihe geschaltet. Die Ausgangsstufe ist ein CMOS-Inverter, dessen inhärente Gatekapazität C_p für den dynamischen Betrieb der Schaltung verwendet wird. Dabei sind zwei Phasen zu unterscheiden:

- Während der ersten, der Vorladephase gilt:

  Lo: Taktpegel $\Phi < [U_{thN} = U_B + U_{thP}]$ (0.5)

  Der Vorladetransistor P_V leitet und lädt die Kapazität C_p, da der Evaluierungstransistor N_E sperrt, unabhängig vom Leitungszustand des Schalttransistors N_1 auf den Wert der Betriebsspannung $U_B \approx +2{,}5$ V auf (Bild 2.8a). Deshalb leitet der Transistor N_0 des Ausgangsinverters und senkt den Ausgangsspannungspegel („unbedingt“) auf Erdpotential ab:

  U_e beliebig **Lo:** $U_a = 0 < U_{thN}$ (0.6)

- Während der zweiten, der Evaluierungsphase gilt:

  Hi: Taktpegel $\Phi > [U_{thN} = U_B + U_{thP}]$ (0.7)

  Der Evaluierungstransistor N_E leitet und entlädt, falls abhängig vom Eingangssignal U_e der Schalttransistor N_1 ebenfalls leitet, die Kapazität C_p, da der Vorladetransistor P_V sperrt (Bild 2.8b). Dann leitet der Transistor P_0 des Ausgangsinverters und hebt den Ausgangsspannungspegel („bedingt“) auf den Wert der Betriebsspannung U_B an:

  Hi: $U_e = U_B > U_{thN}$ **Hi:** $U_a = U_B > U_{thN}$ (0.8)

  Andernfalls, wenn der Schalttransistor N_1 sperrt, kann der Evaluierungstransistor N_E die Kapazität C_p nicht entladen. Dann bleibt der Transistor N_0 des Ausgangsinverters leitend und hält den Ausgangsspannungspegel („bedingt“) auf Erdpotential:

  Lo: $U_e = 0 < U_{thN}$ **Lo:** $U_a = 0 < U_{thN}$ (0.9)

Man erkennt, daß die Grundschaltung in Domino-Logik die Schaltfunktion der „Identität“ realisiert. Sie ist, wie auch eine logische Verknüpfung mehrerer Eingangssignale, nur während der Evaluierungsphase gültig.

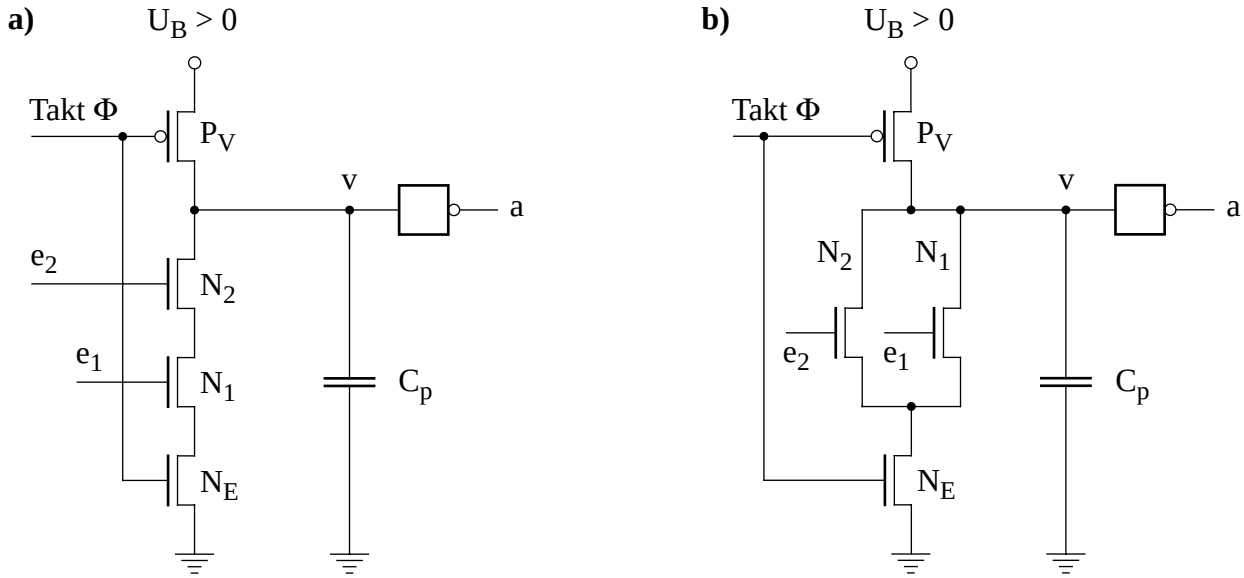


Bild 2.9: Domino-Logik in CMOS-Technologie.
a) UND-Schaltung (positive Logik); b) ODER-Schaltung (positive Logik).

Schaltet man nämlich zwischen das vom Taktsignal Φ angesteuerte CMOS-Transistorpaar eine Reihen- oder eine Parallelschaltung von NMOS-Schalttransistoren (oder bei komplizierteren Schaltfunktionen auch direkt ein komplexeres Netzwerk aus NMOS-Transistoren), so erhält man in positiver Logik eine UND-Schaltung (im obigen Bild links) bzw. eine ODER-Schaltung (im obigen Bild rechts). Aufgrund des Ausgangsinverters, der die für den dynamischen Betrieb erforderliche Kapazität beisteuert und außerdem eine Signalverstärkung liefert, sind die in Domino-Logik realisierten Schaltfunktionen grundsätzlich bejaht. Schaltet man mehrere Domino-Stufen in einer Kette hintereinander, so setzen sich die Verknüpfungen der Eingangssignale von Stufe zu Stufe fort (daher der Name dieser Schaltungstechnik).

2.5 MOS-Speicherschaltungen

2.5.1 Speichermatrizen (RAM)

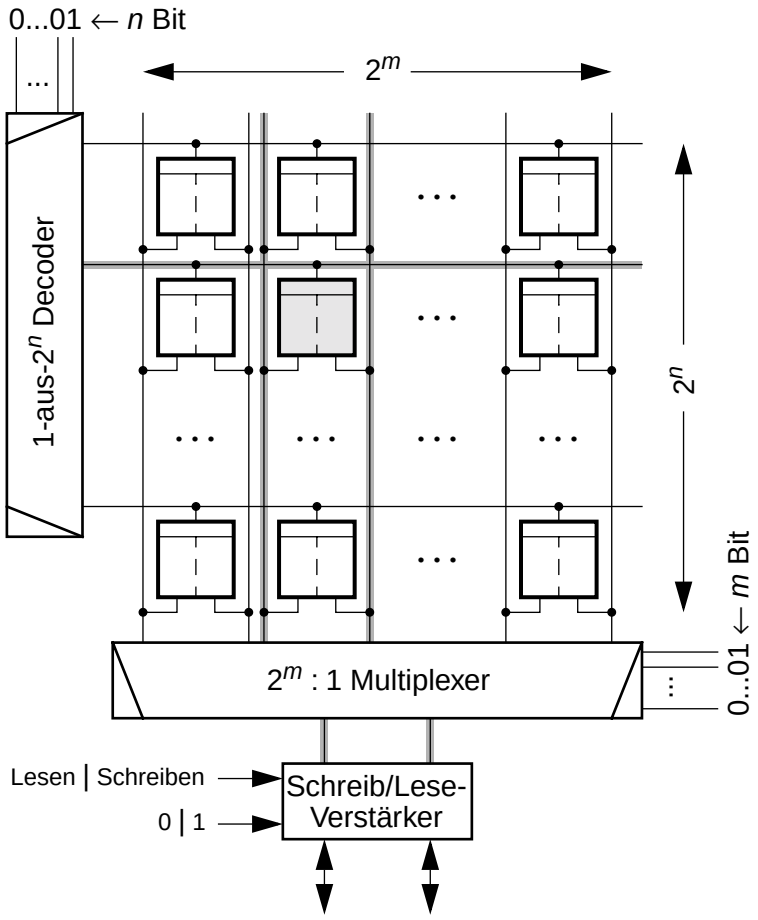


Bild 2.10: Blockschaltbild eines Schreib/Lese-Speichers mit wahlfreiem Zugriff (RAM)

Die Technologie der Mikroelektronik erlaubt es, Speicherzellen in extrem großer Anzahl auf einem gemeinsamen Silizium-Chip zu integrieren: Stand der Technik sind mehrere Millionen Bits pro Chip. Die Speicherzellen sind geometrisch in Zeilen und Spalten angeordnet. Eine Zeile von Speicherzellen kann z.B. ein Wort enthalten, dessen einzelne Bits sich in unterschiedlichen Spalten befinden. Man bezeichnet daher die Zeilen der Speichermatrix auch als „Wortrichtung", die Spalten als „Bitrichtung". Selektiert man gleichzeitig eine Zeile und eine Spalte der Speichermatrix, so erhält man den direkten Zugriff auf die Speicherzelle am Schnittpunkt. Dies wird als wahlfreier Zugriff bezeichnet („Random Access Memory", RAM).

Die Selektion einer Speicherzelle auf einem höchstintegrierten Speicherchip kann nicht direkt erfolgen. Eine Speichermatrix der Größe

2^n Zeilen $\cdot\ 2^m$ Spalten $= 2^{n+m}$ Beispiel: $2^{10+10} = 2^{20} = 1$ MegaBit,

die also etwa einer Million Speicherzellen enthält, würde sonst

2^n Zeilen $+ 2^m$ Spalten $= 2^n + 2^m$ $2^{10} + 2^{10} = 1.024 + 1.024 = \underline{2.048}$ (!)

äußere Anschlüsse benötigen, was technologisch nicht machbar ist. Codiert man jedoch die Zeilen- und Spaltenadressen („Wort- bzw. Bitadresse") im Dualcode, so gilt für die Anzahl der äußeren Anschlüsse:

$$ld\ 2^n \qquad + ld\ 2^m \qquad = n + m \qquad\qquad 10 + 10 = \underline{20}\ .$$

Zur Selektion genau einer Wort- bzw. einer Bitleitung müssen die dual codierten Wort- bzw. Bitadressen in einen „1-aus-*n*"-Code umgewandelt werden, bei dem definitionsgemäß immer nur ein Bit aktiv ist. Daher enthalten höchstintegrierte Speicherchips am Rande der eigentlichen Speichermatrix stets auch Decoder- und Multiplexerschaltungen, um die Anzahl der äußeren Anschlüsse des Chips in der geschilderten Weise niedrig zu halten, wie das vorhergehende Bild zeigt.

Das folgende Bild zeigt das Layout einer Speichermatrix mit wahlfreiem Zugriff (RAM).

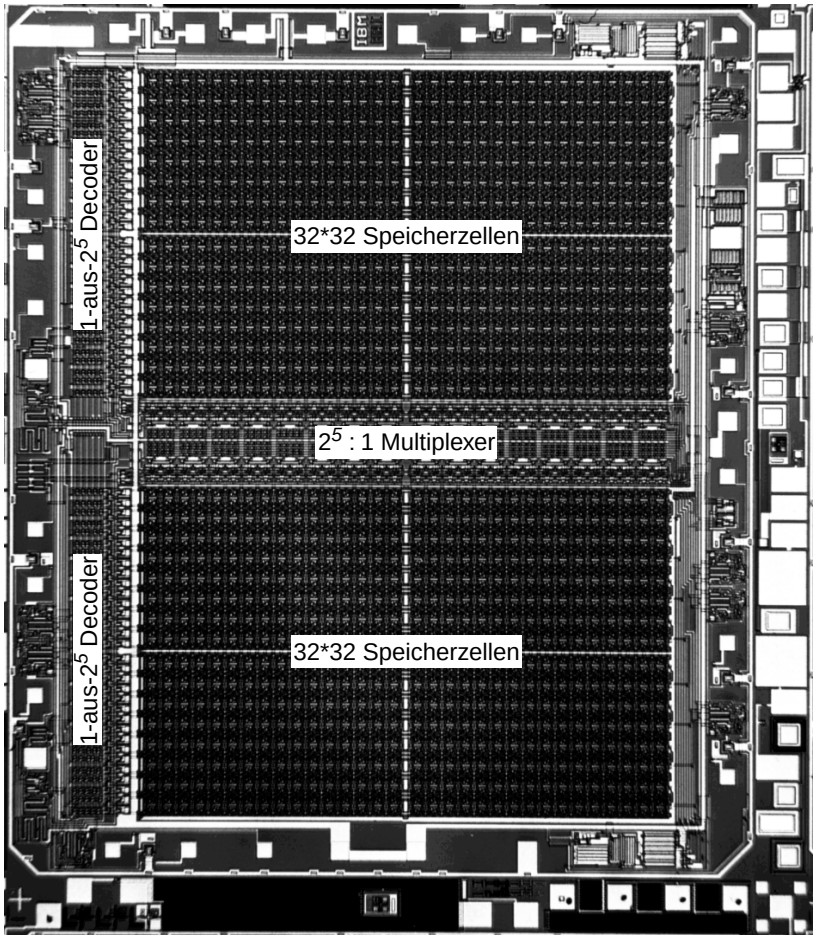


Bild 2.11: Chip-Layout eines Schreib/Lese-Speichers mit wahlfreiem Zugriff (32∗64 bit RAM)

2.5.2 Statische MOS-Speicherzelle

Zur Einführung in die Schaltungstechnik zeigt das folgende Bild die Implementierung eines ungetakteten Basis- und eines zustandsgetakteten Auffang-Flipflops mit logischen Gattern. Die Realisierung kann grundsätzlich in allen bekannten Schaltkreistechnologien erfolgen; hier wird die Realisierung in statischer NMOS-Logik gezeigt.

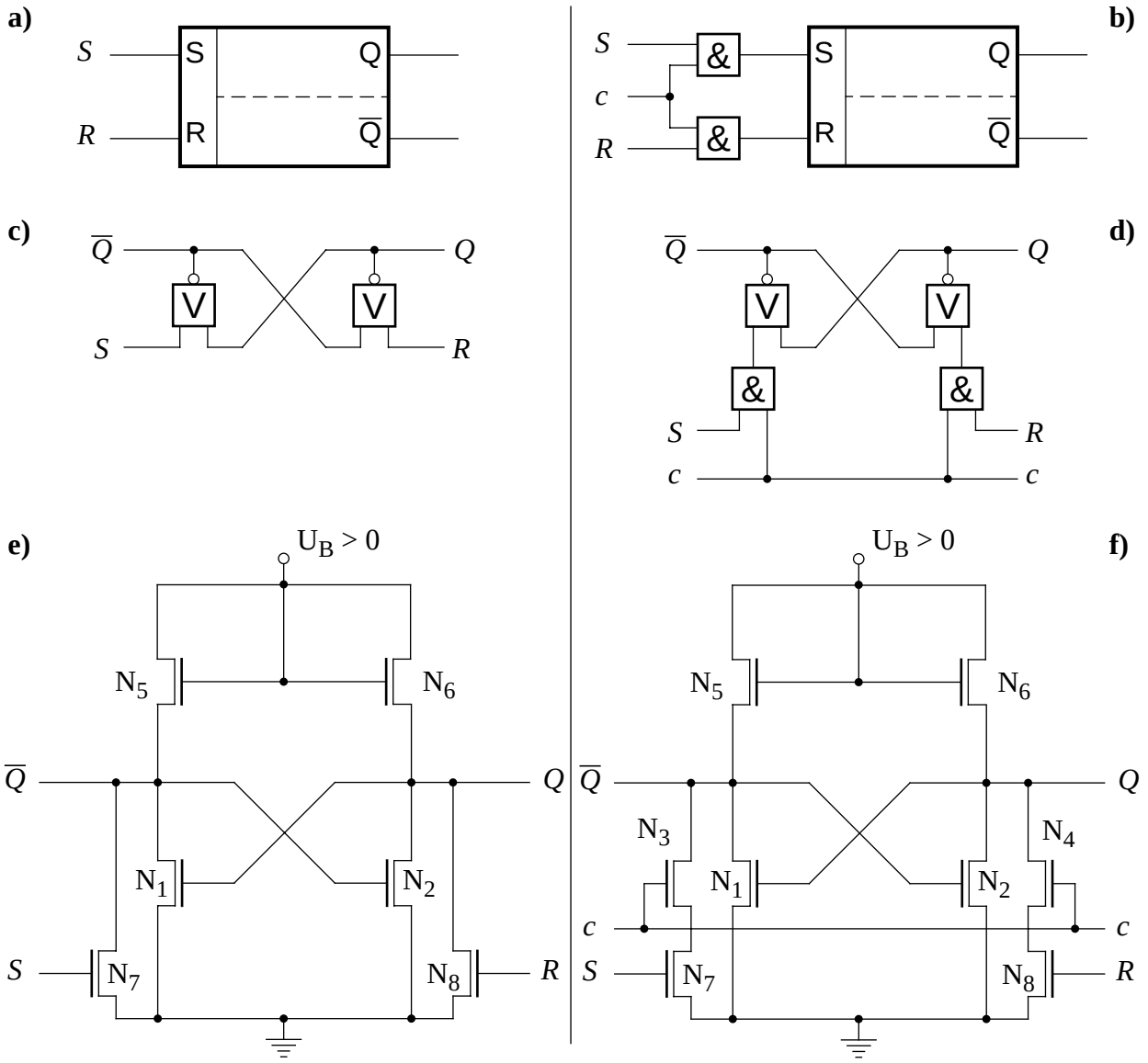


Bild 2.12: Basis-Flipflop (links) und Auffang-Flipflop (rechts). a) b) Schaltsymbole; c) d) Implementierungen mit logischen Gattern; e) f) Realisierungen mit NMOS-Transistoren.

Das folgende Bild zeigt die Schaltung einer bistabilen Speicherzelle, die aus sechs MOS-Transistoren besteht (“Six-Device Cell”). Es handelt sich um MOS-Transistoren mit N-Kanal vom Anreicherungstyp. Dieser Transistortyp wird nicht nur am meisten verwendet, sondern mit ihm läßt sich auch die Wirkungsweise der Speicherzelle anhand positiver Spannungswerte in positiver Logik erläutern, was das Verständnis erleichtert. Die Speicherzelle kann mit einer Betriebsspannung $U_B \approx + 2{,}5$ V betrieben werden.

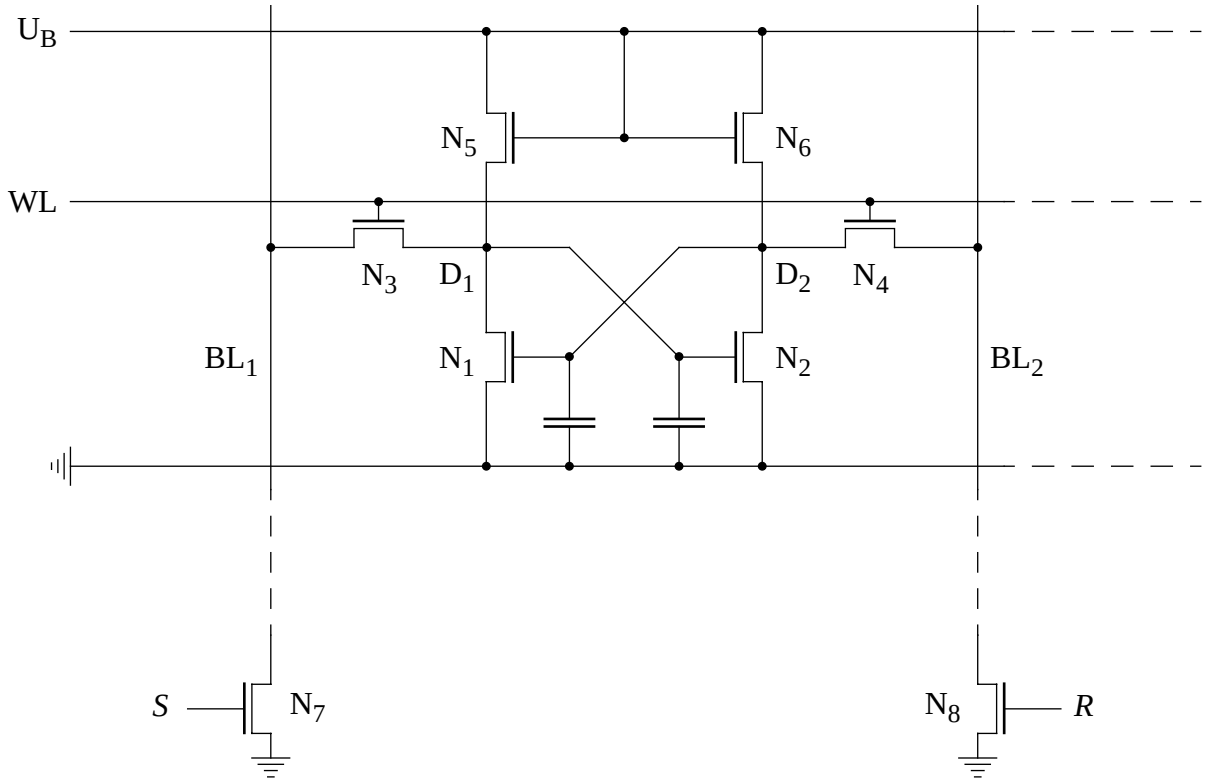


Bild 2.13: Schaltbild der statischen „Sechs-Transistor-Speicherzelle“ mit Lasttransistoren N_5, N_6. WL - Wortleitung; BL_1, BL_2 - Bitleitungspaar. - Literatur: J. S. Schmidt, “Integrated MOS Random-access Memory”, Solid-State Design, 21-25 (1965).

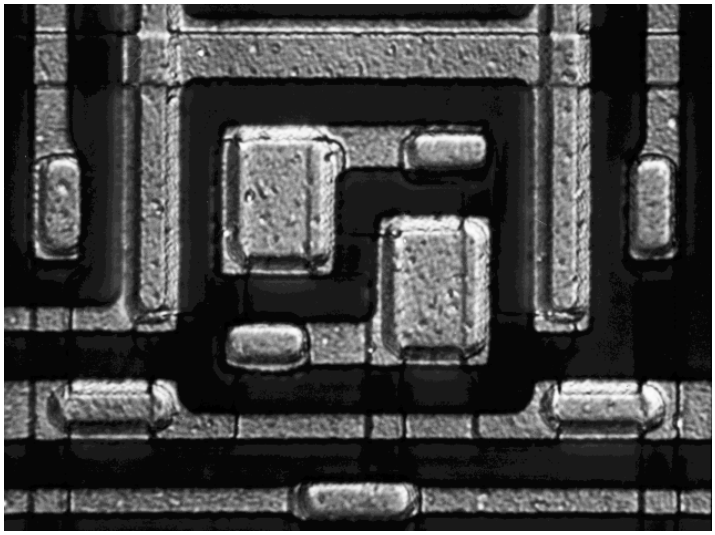


Bild 2.14: Layout der statischen „Sechs-Transistor-Speicherzelle“.
Literatur: R. Remshardt, U.G. Baitinger: „A High Performance Low Power 2048-bit Memory Chip in MOSFET Technology And Its Application”, IEEE J. Solid-State Circuits, Vol. SC-11, No.3 (June 1976), pp. 352-359

a) Betriebsart „Speichern“

Wenn die Speicherzelle unselektiert ist, liegt das Wortleitungspotential WL unterhalb der Schwellspannung; dann sind die Ein/Ausgangstransistoren N_3 und N_4 gesperrt:

Lo: $U_{WL} \approx 0 < U_{thN} \approx +1{,}0\ V$

Leitet Schalttransistor N_1, was dem gespeicherten Zustandsbit Q = **1** entsprechen möge, so liegen der Drainknoten D_1 und damit das Gate des Schalttransistors N_2 auf dem Potential „Lo“, d.h. unterhalb der Schwellspannung, da der Schalttransistor N_1 niederohmig gegenüber dem Lasttransistor N_5 ausgelegt ist. Dadurch wird Schalttransistor N_2 gesperrt. Das hat zur Folge, daß der Drainknoten D_2 über den Lasttransistor N_6 auf das Potential „Hi“ angehoben wird.

Lo: $U_{D1} < U_{thN2}$ **Hi:** $U_{D2} = (U_B - U_{thN6}) > U_{thN1}$

Dadurch bleibt Schalttransistor N_1 leitend, wie ursprünglich angenommen. - In gleicher Weise kann sich das komplementäre Zustandsbit Q = **0** stabil halten, da die Schaltung symmetrisch ist.

b) Betriebsart „Lesen“

Die Speicherzelle wird selektiert, indem das Potential der Wortleitung WL auf den Wert der Betriebsspannung U_B angehoben wird:

Hi: $U_{WL} = U_B > U_{thN}$

Dadurch werden die Ein/Ausgabetransistoren N_3 und N_4 leitend und verbinden die Zellknoten D_1 und D_2 mit den Bitleitungen BL_1 bzw. BL_2, die beide auf mindestens den Wert der Betriebsspannung aufgeladen sein mögen:

Hi: $U_{BL1} \geq U_B$ Hi: $U_{BL2} \geq U_B$

Leitet Schalttransistor N_1, während Schalttransistor N_2 sperrt, was dem gespeicherten Zustandsbit Q = **1** entsprechen möge, so fließt über den relativ hochohmigen Ein/Ausgabetransistor N_3 in Reihe mit dem niederohmigen Schalttransistor N_1 ein Strom, so daß am Zellknoten D_1 der Potentialpegel „Lo“ erhalten bleibt, während der gesättigt leitende Ein/Ausgabetransistor N_4 den Zellknoten D_2 auf dem Potentialpegel „Hi“ zumindest hält.

Lo: $U_{D1} < U_{thN2}$ **Hi:** $U_{D2} = (U_B - U_{thN4}) > U_{thN1}$

Wesentlich ist, daß das Gatepotential U_{D2} des als leitend angenommenen Schalttransistors N_1 sogar ansteigt, so daß sich über den Ein/Ausgabetransistor N_3 ein kräftiger Lesestrom ergibt, der aus Bitleitung BL_1 nachgeliefert wird. Er kann mit einem selektiv an das Bitleitungspaar angeschlossenen stromempfindlichen Differenzverstärker abgefühlt werden. - Ist in der Zelle das komplementäre Zustandsbit Q = **0** gespeichert, fließt der Lesestrom in symmetrischer Weise aus Bitleitung BL_2.

c) Betriebsart „Schreiben“

Soll in die Speicherzelle geschrieben, d.h. das Zustandsbit $Q \in \{0, 1\}$ geändert werden, wird sie wie beim Lesen durch Anheben des Potentials der Wortleitung WL selektiert, so daß die Ein/Ausgabetransistoren N_3 und N_4 leiten:

Hi: $U_{WL} = U_B > U_{thN}$

Soll der Schalttransistor N_2 leitend werden, was wie oben angenommen dem gespeicherten Zustandsbit Q = **0** entspricht, so muß die Bitleitung BL_2 auf Erdpotential, d.h. unter den Wert der Schwellspannung abgesenkt werden, während die Bitleitung BL_1 mindestens auf dem Wert der Betriebsspannung bleibt:

Hi: $U_{BL1} \geq U_B$ Lo: $U_{BL2} = 0 < U_{thN}$

Der gesättigt leitende Ein/Ausgabetransistor N_3 hebt das Potential des Zellknotens D_1 und damit das Gatepotential von Schalttransistor N_2 über den Wert der Schwellspannung an, so daß dieser leitet, während Schalttransistor N_1 gesperrt wird, da der Zellknoten D_2 über den linear leitenden Ein/Ausgabetransistor N_4 das niedrige Potential der angeschlossenen Bitleitung BL_2 annimmt :

Hi: $U_{D1} = (U_B - U_{thN3}) > U_{thN2}$ **Lo:** $U_{D2} = U_{BL2} = 0 < U_{thN1}$

Man erkennt, daß die *Geschwindigkeit*, mit der die statische Speicherzelle beim Schreiben ihren Zustand ändert, offenbar von der Aufladung der Gatekapazität des jeweils einzuschaltenden Schalttransistors über einen Ein/Ausgabetransistor abhängt. Das entspricht ganz der Wirkungsweise einer Inverterschaltung. Mit anderen Worten: Diese Speicherzelle wird von zwei kreuzgekoppelten Invertern gebildet, die aus Schalttransistor N_1 und Ein/Ausgabetransistor N_3 bzw. Schalttransistor N_2 und Ein/Ausgabetransistor N_4 bestehen. - Die Aufgabe der Lasttransistoren N_5 und N_6 besteht lediglich darin, während des Ruhezustands der Speicherzelle die Leckströme an den Drainknoten D_1 bzw. D_2 nachzuliefern (die dort zum Substrat abfließen), damit das Gatepotential des jeweils leitenden Schalttransistors nicht unter die Schwellspannung absinkt, so daß er leitend bleibt.

Die *Verlustleistung* dieser Speicherzelle wird hauptsächlich in dem Lasttransistor N_5 oder N_6 in Wärme umgesetzt, der sich in Reihe zum jeweils leitenden Schalttransistor befindet, da an ihm fast die gesamte Betriebsspannung ($U_B - U_{D1} \approx U_B$) abfällt (leider nutzlos, da er keinen Leckstrom nachzuliefern hat, weil er zur entladenen Gatekapazität führt). - Eine elegante Methode, dies zu vermeiden, besteht darin, die Lasttransistoren mit dem komplementären P-Kanaltyp auszuführen. Dadurch wird gewährleistet, daß Lasttransistor N_5 sperrt, wenn Schalttransistor N_1 leitet, so daß praktisch keine Gleichstromverlustleistung verbraucht wird, während andererseits Lasttransistor N_6 leitet, um den Leckstrom an der aufgeladenen Gatekapazität des leitenden Schalttransistors N_1 zu kompensieren. Man erkauft den Vorteil einer extrem niedrigen Verlustleistung der integrierten Speichermatrizen durch den komplizierteren CMOS-Herstellungsprozeß.

2.5.3 Dynamische MOS-Speicherzelle

Eine besonders sparsame und daher extrem kleine, dynamische Speicherzelle zeigt das nachfolgende Bild. Sie besteht aus einem Schalttransistor N_1 und einem Speicherkondensator C, der technologisch als MOS-Kapazität eines NMOS-Transistors ausgeführt wird (“One-Device Cell”).

Die Ein-Transistor-Speicherzelle wird selektiert durch Anheben des Potentials der Wortleitung WL über den Wert der Schwellspannung U_{thN}, so daß der Schalttransistor N_1 leitet und den Speicherkondensator C mit der Bitleitung BL verbindet. Diese nimmt das Potential des Speicherkondensators C an und kann es an den Eingang eines an die Bitleitung BL angeschlossenen spannungsempfindlichen Leseverstärkers weitergeben, dessen Eingangskapazität in der Regel leider relativ groß

ist im Vergleich zum Kapazitätswert des Speicherkondensators C. Es ist unbedingt erforderlich, eine gelesene Zelle unmittelbar danach aufzufrischen.

Der Vorteil der dynamischen Ein-Transistor-Speicherzelle liegt in ihrer minimalen Zellfläche; er muß jedoch durch entsprechend aufwendige Peripherieschaltkreise erkauft werden. Die niedrige Verlustleistung erlaubt die Integration extrem großer Speichermatrizen auf gegebener Chipfläche. Wegen des erforderlichen komplizierten Auffrischvorgangs ist die Zugriffszeit relativ hoch.

Man erkennt durch den Vergleich mit der statischen Sechs-Transistor-Speicherzelle nach Bild 2.17, daß die dynamische Ein-Transistor-Speicherzelle darin enthalten ist.

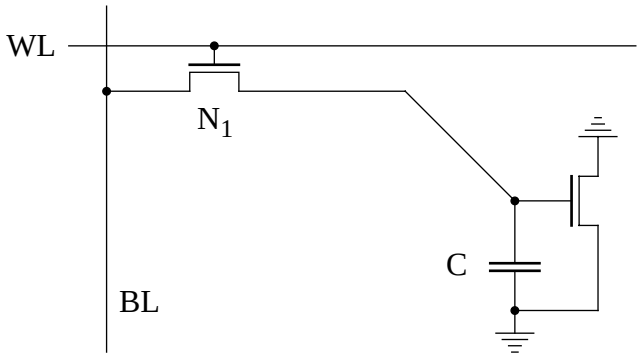


Bild 2.15: Schaltbild der dynamischen „Ein-Transistor-Speicherzelle“ mit Ladungsspeicher (C). WL - Wortleitung; BL - Bitleitung.
Literatur: L. Cohen et al., ”Single-transistor Cell Makes Room for More Memory on an MOS Chip”, Electronics, 69-75 (1971).

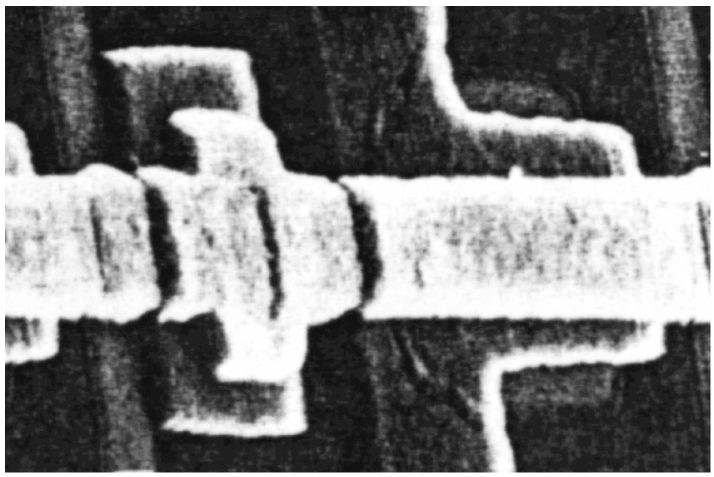


Bild 2.16: Layout der dynamischen „Ein-Transistor-Speicherzelle“.

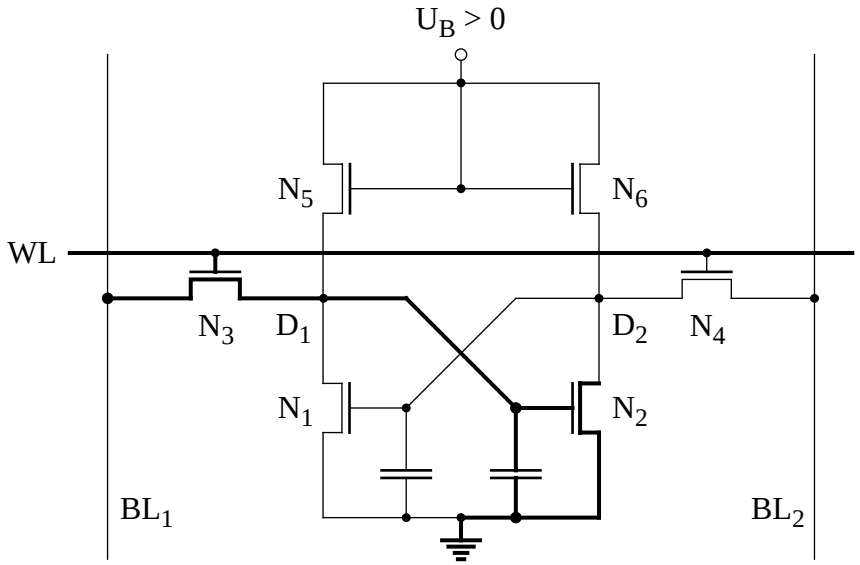


Bild 2.17: Statische „Sechs-Transistor-Speicherzelle“; hervorgehoben ist die darin enthaltene dynamische „Ein-Transistor-Speicherzelle“.

3 Logikebene

Prof. Dr.-Ing. U.G. Baitinger

3.1 Die Darstellung von Schaltfunktionen

Die Boolesche Algebra lehrt den Umgang mit diskreten („wohlunterschiedenen“), digitalen („abzählbaren“) Werten. Die Schaltalgebra ist ein binärer („zweiwertiger“) Sonderfall der Booleschen Algebra: Eine Variable *a* kann nur zwei verschiedene Werte $a \in \{0,1\}$ annehmen. *Schaltfunktionen* sind daher binäre Funktionen binärer Variabler; ihr Wertebereich ist dahingehend eingeschränkt. Da auch die Aussagenlogik für eine logische Aussage nur zwei Werte $A \in$ {falsch, wahr} zuläßt, ist auch sie ein binärer Sonderfall der Booleschen Algebra. Deshalb wird bei schaltalgebraischen Begriffen häufig das Adjektiv „logisch“ hinzugefügt. Folgende Begriffe sind deshalb äquivalent:

- Schaltfunktionen sind „logische Verknüpfungen“;
- Schaltglieder heißen auch „logische Gatter“;
- Funktionstabellen werden auch „Wahrheitstabellen“ genannt.

Schaltfunktionen können durch Listen, Tafeln, Tabellen oder Matrizen dargestellt werden; besonders anschaulich ist ihre Darstellung als Graph. Im Folgenden sollen die drei *elementaren* Schaltfunktionen eingeführt und durch ihre Relationsgraphen dargestellt werden.

3.1.1 Der Funktionsgraph

Für die Schaltfunktion der *Negation* $y = \bar{x}_0$ gilt, daß der Funktionswert $y = 1$ ist, wenn die Eingangsvariable $x_0 = 0$ ist; sonst gilt $y = 0$.

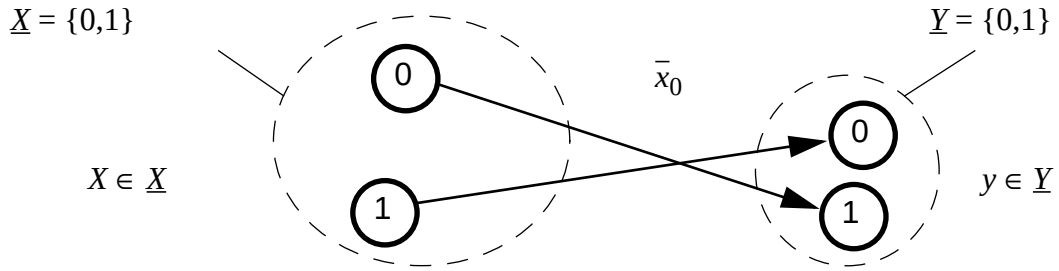


Bild 3.1: Relationsgraph der Negation $y = f(x_0) = \bar{x}_0$

Für die Schaltfunktion der *Konjunktion* $y = (x_1 \,\&\, x_0)$ gilt, daß der Funktionswert $y = 1$ ist, wenn die erste Eingangsvariable $x_1 = 1$ *und* die zweite Eingangsvariable $x_0 = 1$ ist; sonst gilt $y = 0$.

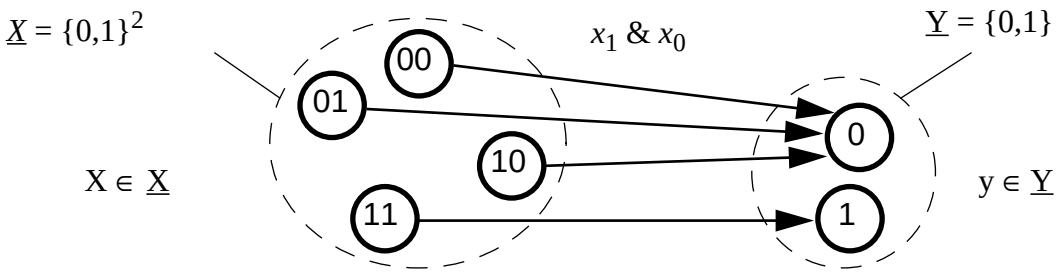


Bild 3.2: Relationsgraph der Konjunktion $y = (x_1 \,\&\, x_0)$

Für die Schaltfunktion der *Disjunktion* $y = x_1 \vee x_0$ gilt, daß der Funktionswert $y = 1$ ist, wenn die erste Eingangsvariable $x_1 = 1$ *oder* die zweite Eingangsvariable $x_0 = 1$ ist *oder* beide Eingangsvariablen $x_1 = x_0 = 1$ sind; sonst gilt $y = 0$.

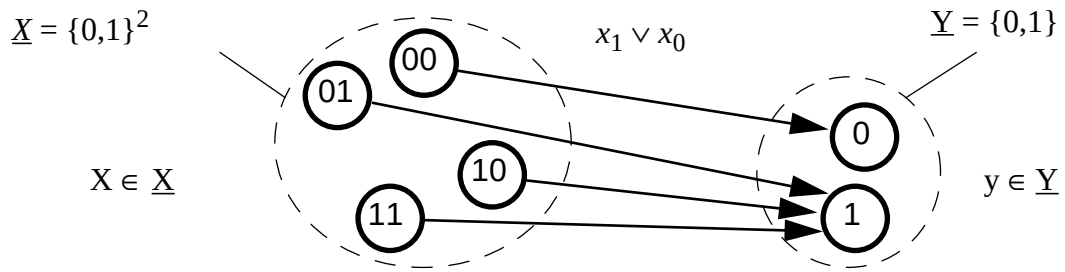


Bild 3.3: Relationsgraph der Disjunktion $y = (x_1 \vee x_0)$

Die Graphen lassen erkennen, daß Schaltfunktionen $y = f(X)$ eindeutige Abbildungen f: $\underline{X} \rightarrow \underline{Y}$ der Elemente $X \in \underline{X}$ einer Definitionsmenge $\underline{X}$ auf die Elemente $y \in \underline{Y}$ einer Zielmenge $\underline{Y}$ sind. Zur Unterstützung des Entwurfs digitaler Schaltungen auf der Logikebene wurde eine Fülle weiterer Darstellungsformen entwickelt, von denen nachfolgend die wichtigsten vorgestellt werden sollen.

3.1.2 Belegung und Indizierung

Man faßt die n Eingangsvariablen x_i einer Schaltfunktion f als Komponenten eines n-Tupels auf und bezeichnet diesen als „Eingangsbelegung" X der Schaltfunktion, wie das folgende Bild zeigt. Werden die Eingangsvariablen x_i auf spezielle Werte $x_{ij} \in \{0,1\}$ festgelegt, so erhält man eine spezielle Eingangsbelegung X_j. Um den Index j zu ermitteln, liest man die Werte der Komponenten x_{ij} wie die gewichteten Stellen einer Dualzahl (Grundzahl $R_B = 2$), die man dann zur kompakteren Darstellung in einen Index j in einem Zahlensystem mit einer höheren Grundzahl umwandelt, z.B. im Oktalsystem (Grundzahl $R_O = 2^3$) oder im Hexadezimalsystem (Grundzahl $R_X = 2^4$).

allgemeine Belegung:

$X \quad = (x_n, \ldots, x_i, \ldots, x_2, x_1);$

spezielle Werte:

$X_j \quad = (x_{nj}, \ldots, x_{ij}, \ldots, x_{2j}, x_{1j}); x_{ij} \in \{0,1\}$

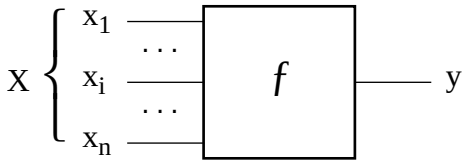


Bild 3.4: Eingangsbelegung X einer Schaltfunktion $y = f(X)$

Für die Menge $\underline{X}$ aller Belegungen und deren Mächtigkeit $|\underline{X}|$, d.h. die Anzahl aller Belegungen X_j gilt bei n Eingangsvariablen:

$\underline{X} \quad = \{ X_j \mid x_i \in \{0,1\}, i = 1 \ldots n \};$ $\qquad |\underline{X}| = 2^n$

Beispiel 3.1.1: Hextupel $X = (x_6, x_5, x_4, x_3, x_2, x_1)$

Es sei $n = 6$. Der obige Hextupel ist eine allgemeine Eingangsbelegung, gegeben sei die spezielle Eingangsbelegung $X_j = (0, 1, 1, 0, 1, 0)$. Gesucht ist der zugehörige Index j im Oktalsystem. Liest

man die spezielle Eingangsbelegung X_j als Dualzahl $(\,011\,010\,)_B$, so kann man diese in die Oktalzahl 32_O umwandeln, d.h. es gilt $j = 32$ und $X_j = X_{32}$. Aus der kompakten Darstellung $j = 32$ lassen sich die binären Werte der Eingangsvariablen x_{ij}, d.h. die Komponenten des 6-Tupels im gegebenen Fall leicht rekonstruieren: $x_6 = 0$, $x_5 = 1$, $x_4 = 1$, $x_3 = 0$, $x_2 = 1$, $x_1 = 0$.

3.1.3 Die Funktionstabelle („Wahrheitstabelle“)

Schaltfunktionen haben eine endliche Definitionsmenge, d.h. die Menge $\underline{X}$ aller Belegungen X ist endlich: $|\,\underline{X}\,| = 2^n$. Sie können daher in Tabellenform dargestellt werden. Man schreibt alle 2^n Eingangsbelegungen X_j in aufsteigender Reihenfolge des Index j untereinander und schreibt den zugehörigen Funktionswert $y \in \{0,1\}$ jeweils daneben, so daß sich die „Funktionsspalte“ der betreffenden Schaltfunktion $y_i = f(X)$ ergibt. Bei n Eingangsvariablen hat die Wahrheitstabelle 2^n Zeilen und es sind 2^m mit $m = 2^n$ verschiedene Funktionsspalten möglich. Das folgende Bild zeigt die Struktur der Wahrheitstabelle für Schaltfunktionen mit $n = 1$ Eingangsvariablen: $X = (x_0)$. Sie hat $2^n = 2^1 = 2$ Zeilen und es sind $2^2 = 4$ unterschiedliche Funktionsspalten möglich, die wie folgt bezeichnet werden:

y_0	Konstante	0
y_1	Identität	x_0
y_2	*Negation*	$\overline{x_0}$
y_3	Konstante	1

j	x_0	y_0	y_1	y_2	y_3
0	0	0	0	1	1
1	1	0	1	0	1

Bild 3.5: Wahrheitstabelle aller Schaltfunktionen mit $n = 1$ Eingangsvariablen

Das nächste Bild zeigt die Struktur der Wahrheitstabelle für Schaltfunktionen mit $n = 2$ Eingangsvariablen: $X = (x_1, x_0)$. Sie hat $2^n = 2^2 = 4$ Zeilen und $2^4 = 16$ mögliche Funktionsspalten. Die meisten Funktionsspalten werden mit charakteristischen Namen bezeichnet. Man findet die booleschen Grundfunktionen der *Negation* , der *Konjunktion* und der *Disjunktion* , aber auch kompliziertere Schaltfunktionen, wie die Antivalenz („exklusives Oder“) und die negierten Schaltfunktionen NOR und NAND:

y_0	Konstante	0
y_1	*Konjunktion*	$x_1 \,\&\, x_0$
y_2	-	
y_3	bejahte Variable	x_1
y_4	-	
y_5	bejahte Variable	x_0
y_6	Antivalenz	$x_1 \oplus x_0$
y_7	*Disjunktion*	$x_1 \vee x_0$

y_F	Konstante	1
y_E	NAND-Funktion	$\overline{x_1 \,\&\, x_0}$
y_D	Implikation	$x_1 \rightarrow x_0$
y_C	negierte Variable	$\overline{x_1}$
y_B	Implikation	$x_1 \leftarrow x_0$
y_A	negierte Variable	$\overline{x_0}$
y_9	Äquivalenz	$x_1 \equiv x_0$
y_8	NOR-Funktion	$\overline{x_1 \vee x_0}$

j	x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_A	y_B	y_C	y_D	y_E	y_F
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
2	1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
3	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Bild 3.6: Wahrheitstabelle aller Schaltfunktionen mit *n* = 2 Eingangsvariablen

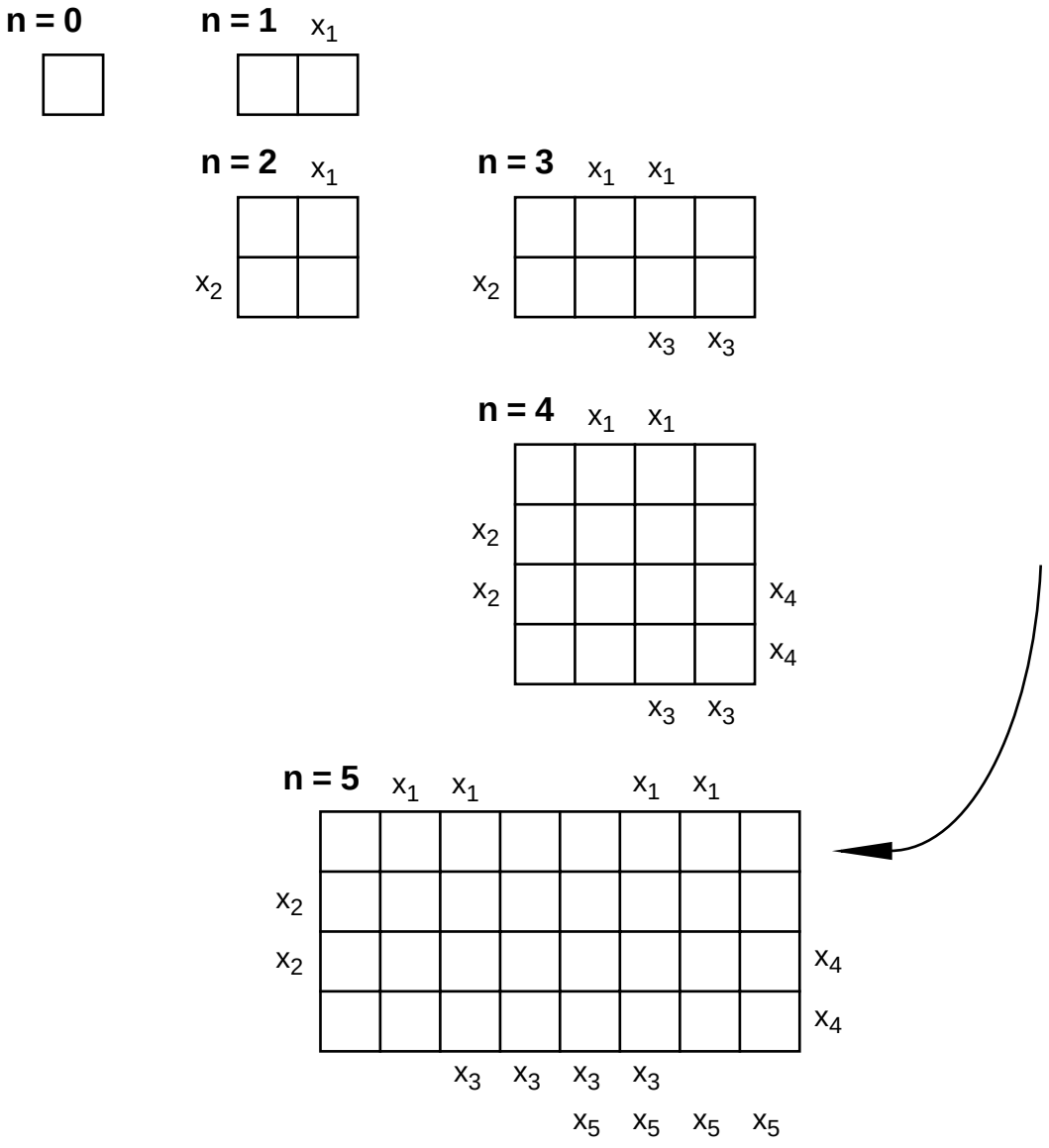


Bild 3.7: Erzeugung Karnaugh-Veitch-Diagramme („KV-Diagramme“)
für Schaltfunktionen mit *n* = 0 . . 5 Eingangsvariablen

3.1.4 Das Karnaugh-Veitch-Diagramm

Eine Funktionstabelle enthält zwar die vollständige Funktionsbeschreibung einer Schaltfunktion, doch läßt sie die Nachbarschaften zwischen zwei speziellen Eingangsbelegungen X_i und X_j, die die Grundlage zur Minimierung von Schaltfunktionen sind, nicht sofort erkennen. Dazu wurde das zweidimensionale *Karnaugh-Veitch-Diagramm* entwickelt („KV-Diagramm“). Das vorhergehende Bild zeigt, wie KV-Diagramme für eine schrittweise Zunahme der Anzahl $n \geq 0$ der Eingangsvariablen x_i durch Spiegelung an einer abwechselnd vertikalen bzw. horizontalen Achse konstruiert werden. Man stellt allerdings auch fest, daß sie für $n > 5$ Eingangsvariablen unhandlich groß und unübersichtlich werden.

Im KV-Diagramm

- werden die Zeilen und Spalten abwechselnd mit den bejahten Variablen des n-Tupels $X = (x_n , \ldots , x_i , \ldots , x_2 , x_1)$ bezeichnet (negierte Variablen gelten für die jeweils benachbarten, unbezeichneten Zeilen bzw. Spalten). Dadurch wird jeder speziellen Eingangsbelegung X_j ein *Feld* in der quadratischen oder halbquadratischen Anordnung von Feldern zugeordnet..
- In das Feld, das einer speziellen Eingangsbelegung X_j entspricht, trägt man den zugehörigen *Funktionswert* $y = f(X_j) \in \{0,1\}$ ein. (Die Felder können zusätzlich mit den Indizes j gekennzeichnet werden, obwohl die Lage eines Feldes durch die zugehörige Eingangsbelegung X_j eindeutig bestimmt ist.)

3.1.5 Der Strukturausdruck

Jede Schaltfunktion kann durch eine Gleichung der Form

$$y = [(a \otimes b) \otimes (c \otimes d)]$$

dargestellt werden. Die rechte Seite der Gleichung ist ein „Strukturausdruck” (auch „boolescher Ausdruck” genannt). Dabei stehen die Symbole $\otimes$ für beliebige, in der Regel unterschiedliche „boolesche” *Operatoren*, wie zum Beispiel:

- die elementaren Schaltfunktionen Konjunktion & und Disjunktion $\vee$
- oder die NOR-Funktion $\overline{\&}$ und die NAND-Funktion $\overline{\vee}$,
- aber auch für kompliziertere Schaltfunktionen, wie die Antivalenz $\oplus$ oder die Äquivalenz $\equiv$.

Die im Strukturausdruck enthaltenen Buchstaben sind *Operanden*, d.h. entweder binäre Variable oder, rekursiv, wiederum Strukturausdrücke der obigen Form. Zwei durch einen Operator verknüpfte Operanden bilden einen *Term*. Man erhält die Funktionsspalte der Wahrheitstabelle oder den Inhalt des KV-Diagramms für eine spezielle Schaltfunktion, indem man alle Eingangsbelegungen X_j nacheinander in den Strukturausdruck der Schaltfunktion einsetzt und die zugehörigen Funktionswerte $y = f(X_j) \in \{0,1\}$ ausrechnet.

Die Bezeichnung „Strukturausdruck” bedeutet, daß sie nach folgenden Regeln in eine jeweils gestaltgleiche („isomorphe”) Struktur mit logischen Gattern umgewandelt werden können:

- Jede *Klammer* entspricht einem logischen *Gatter*;
- jeder *Operator* entspricht der *Schaltfunktion* des betreffenden Gatters;
- jeder *Operand* entspricht einer *Eingangsklemme* des Gatters.

Beispiel 3.1.2:

Gegeben sei der Strukturausdruck $y = \{ [(\bar{x}_2) \,\&\, x_1] \vee [x_2 \,\&\, (\bar{x}_1)] \}$;
gesucht ist die isomorphe Schaltnetzstruktur. Das folgende Schaltbild zeigt das Ergebnis.

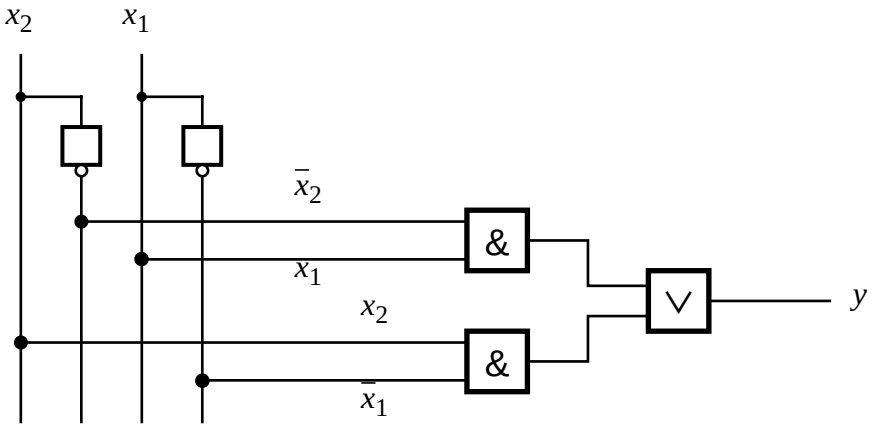


Bild 3.8: Zum gegebenen Strukturausdruck isomorphe Schaltnetzstruktur

Zur Schreibweise

Symbole & $\wedge$ •	$\leftrightarrow$	„logisches Produkt”	$\leftrightarrow$	Konjunktion
Symbole $\vee$ +	$\leftrightarrow$	„logische Summe”	$\leftrightarrow$	Disjunktion

Konjunktion (&) und Disjunktion ($\vee$) sind *gleichberechtigte* Operatoren, so daß Prioritäten explizit in Klammern gesetzt werden müssen:

$$y = (\bar{x}_2 \,\&\, x_1) \vee (x_2 \,\&\, \bar{x}_1)$$

Um die Isomorphie zwischen einem Strukturausdruck und der entsprechenden Schaltnetzstruktur deutlich zu zeigen, kann eine Schreibweise mit besonders aufwendiger Klammersetzung verwendet werden:

$$y = \{ [(\bar{x}_2) \,\&\, x_1] \vee [x_2 \,\&\, (\bar{x}_1)] \}$$

In der englischsprachigen Literatur findet man auch folgende Symbole für die betr. Operatoren:

$$y = (\neg x_2 \wedge x_1) \vee (x_2 \wedge \neg x_1) \qquad\qquad y = \neg x_2 \bullet x_1 \; + \; x_2 \bullet \neg x_1$$

Letztere erinnert an die Bezeichnung „logisches Produkt” für die Konjunktion (Symbole: &, $\wedge$, •) und „logische Summe” für die Disjunktion (Symbole: $\vee$, +). Außerdem wird mit dieser Schreibweise das in der Arithmetik gültige Prinzip: „Punktrechnung geht vor Strichrechnung” eingeführt, so daß die Konjunktion stillschweigend Priorität vor der Disjunktion erhält, was die Klammersetzung vereinfacht. Damit kommt man zu der im folgenden meist gebrauchten Kurzschreibweise:

$$\boxed{y = \bar{x}_2 \, x_1 \vee x_2 \, \bar{x}_1}$$

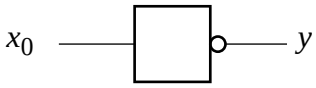
3.1.6 Ausgewählte Schaltfunktionen

Es läßt sich zeigen, daß alle logischen Verknüpfungen der Schaltalgebra ausschließlich mit den drei Schaltfunktionen Negation ($\neg$), Konjunktion (&) und Disjunktion ($\vee$) dargestellt werden können. Diese Operatoren bezeichnen *elementare Schaltfunktionen*; sie stellen ein *Basissystem* der Schaltalgebra dar. Sie sind im folgenden Bild zusammengestellt.

x_0	$y = \bar{x}_0$
0	1
1	0

	x_0
	1 | 0

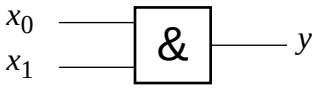
Negation



x_1	x_0	$y = (x_1 \,\&\, x_0)$
0	0	0
0	1	0
1	0	0
1	1	1

	x_0
	0 | 0
x_1	0 | 1

Konjunktion



x_1	x_0	$y = (x_1 \vee x_0)$
0	0	0
0	1	1
1	0	1
1	1	1

	x_0
	0 | 1
x_1	1 | 1

Disjunktion

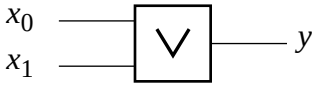


Bild 3.9: Funktionsbeschreibungen und Gattersymbole der drei elementaren Schaltfunktionen

Technisch wichtige Schaltfunktionen ergeben sich aus der Tatsache, daß ein elektronischer Verstärker die Schwingungsphase zwischen sinusförmigen Eingangs- und Ausgangssignalen um eine halbe Periode verschiebt, was im Impulsbetrieb bedeutet, daß aus einem hohen Eingangs- ein niedriger Ausgangspegel wird und umgekehrt. Mit andern Worten:

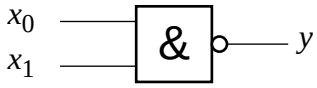
- Mit einer elektronisch realisierten logischen Negation ist eine Verstärkung des Ausgangssignals verbunden. Negierende Schaltfunktionen sind deshalb in größeren Netzwerken unverzichtbar.

Es läßt sich auch zeigen, daß alle logischen Verknüpfungen der Schaltalgebra entweder ausschließlich mit NAND-Gattern oder ausschließlich mit NOR-Gattern implementiert werden können: Die zugehörigen Operatoren ($\overline{\&}$ bzw. $\overline{\vee}$) stellen jeweils für sich allein ebenfalls *Basissysteme* der Schaltalgebra dar, was technisch bedeutet, daß im Prinzip ein einziger negierender und damit verstärkender Bausteintyp genügt, um auch komplexeste Schaltnetze damit aufzubauen. Sie sind im nächsten Bild zusammengestellt.

x_1	x_0	$y = (x_1 \overline{\&} x_0)$
0	0	1
0	1	1
1	0	1
1	1	0

x_1 \ x_0		
	1	1
1	1	0

NAND-Funktion



x_1	x_0	$y = (x_1 \overline{\vee} x_0)$
0	0	1
0	1	0
1	0	0
1	1	0

x_1 \ x_0		
	1	0
0	0	0

NOR-Funktion

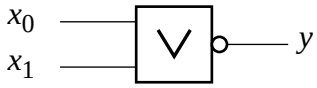


Bild 3.10: Funktionsbeschreibungen und Gattersymbole technisch wichtiger Schaltfunktionen

Im Hinblick auf technische Realisierungen, insbesondere durch elektronische Schaltkreise, ist deshalb folgende Regel zur Transformation bejahter in negierte Schaltfunktionen besonders wichtig:

$$\overline{(a \,\&\, b)} = \overline{a} \vee \overline{b} \qquad\qquad \overline{(a \vee b)} = \overline{a} \,\&\, \overline{b}$$

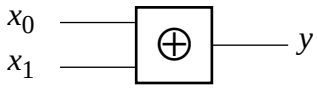
Bild 3.11: „DeMorgansche Regel“

Technisch sind nicht nur einfache NAND- und NOR-Gatter verfügbar, sondern auch solche, die *kompliziertere Schaltfunktionen* realisieren, so daß sie zur Vereinfachung einer komplexen Schaltnetzstruktur eingesetzt werden können, z.B. Antivalenz-, auch Exclusive-OR- oder kurz XOR-Gatter genannt, sowie Äquivalenz-Gatter, die im folgenden Bild zusammengestellt sind. Man erkennt übrigens aus der Wahrheitstabelle der Antivalenzfunktion, daß sie eine binäre Addition erzeugt, allerdings ohne Übertrag.

x_1	x_0	$y = (x_1 \oplus x_0)$
0	0	0
0	1	1
1	0	1
1	1	0

x_1 \ x_0		
	0	1
1	1	0

Antivalenz



x_1	x_0	$y = (x_1 \equiv x_0)$
0	0	1
0	1	0
1	0	0
1	1	1

x_1 \ x_0		
	1	0
0	0	1

Äquivalenz

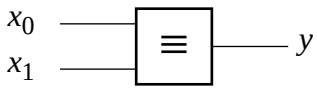


Bild 3.12: Funktionsbeschreibungen und Gattersymbole komplizierter Schaltfunktionen

3.2 Kombinatorische Schaltungen („Schaltnetze“)

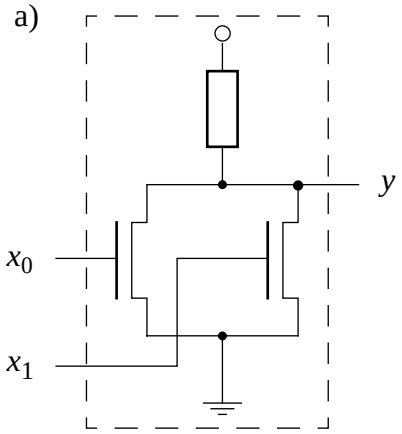
3.2.1 Darstellungsebenen und Entwurfsziele

Schaltnetze sind konkrete Implementierungen abstrakter Schaltfunktionen. Sie kombinieren die an ihrem Eingang anliegenden binären Größen zu einer binären Ausgangsgröße; man nennt sie deshalb auch *kombinatorische Schaltungen*. Schaltfunktionen können unterschiedlich dargestellt werden, z.B. durch Wahrheitstabellen, KV-Diagramme oder Strukturausdrücke. Schaltnetze werden, isomorph zum zugehörigen Strukturausdruck, mit Gattersymbolen dargestellt. Zur Terminologie:

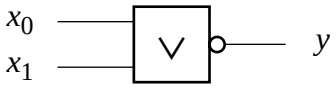
- *Implementierung*: Die Darstellung mit logischen Gattern („Logikebene”)
- *Realisierung*: Die Darstellung mit Transistoren und passiven Bauelementen („Schaltungsebene”)

Das nächste Bild zeigt eine Schaltung mit MOS-Transistoren, die in positiver Logik durch ein NOR-Gatter symbolisiert werden kann, dessen Schaltfunktion hier durch eine Wahrheitstabelle wiedergegeben wird. Das übernächste Bild zeigt eine Schaltnetzstruktur mit NOR-Gattern, die durch ein Symbol für eine allgemeine Schaltfunktion $f(X)$ abstrahiert werden kann, wobei die Funktion $f(X)$ z.B. wieder durch eine Wahrheitstabelle spezifiziert sein kann.

a)



b)

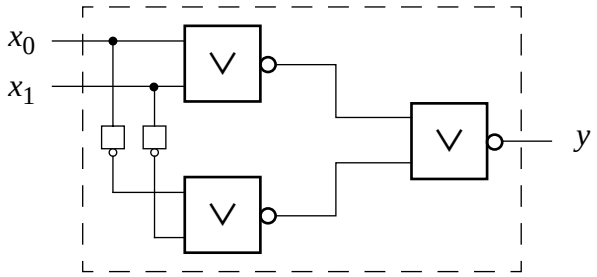


c)

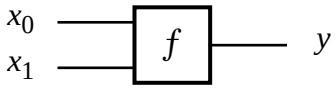
x_1	x_0	$y = \overline{x_1 \vee x_0}$
0	0	1
0	1	0
1	0	0
1	1	0

Bild 3.13: Realisierung eines logischen Gatters auf der Schaltungsebene
a) Schaltungsstruktur, b) NOR-Gattersymbol, c) NOR-Schaltfunktion

a)



b)



c)

x_1	x_0	$y = f(x_1, x_0)$
0	0	$y \in \{0,1\}$
0	1	$y \in \{0,1\}$
1	0	$y \in \{0,1\}$
1	1	$y \in \{0,1\}$

Bild 3.14: Implementierung eines Schaltnetzes auf der Logikebene
a) Schaltnetzstruktur, b) allg. Gattersymbol, c) allg. Schaltfunktion

Zur Senkung der Fertigungskosten ist man beim Schaltungsentwurf bestrebt, solche Schaltungen zu entwerfen, die den geforderten Zweck mit möglichst wenig technischem Aufwand erreichen. (Dabei läßt man die Entwurfskosten zunächst außer acht, obwohl sie für höchstintegrierte Schaltungen in hochkomplexen digitalen Systemen ein nicht unerheblicher Kostenfaktor sind). Die Fertigungstechnologien lassen sich grob in zwei Klassen einteilen.

- Die konventionelle Technologie der *teuren Gatter*: Werden digitale Schaltungen mit diskreten Bausteinen aufgebaut, so bestimmen diese die Größe, Geschwindigkeit und Zuverlässigkeit der entworfenen Systeme, während der Einfluß der Verdrahtung demgegenüber vernachlässigt werden kann.
- Die moderne Technologie der *teuren Verdrahtung*: Fläche, Geschwindigkeit und Zuverlässigkeit integrierter Chips werden vor allem durch die Verdrahtung bestimmt, während (z. B. bei MOS-Technologien) die Gatter unter der Verdrahtung liegen und ihre Schaltverzögerungen gegenüber den Laufzeiten auf den Leitungen vernachlässigt werden können.

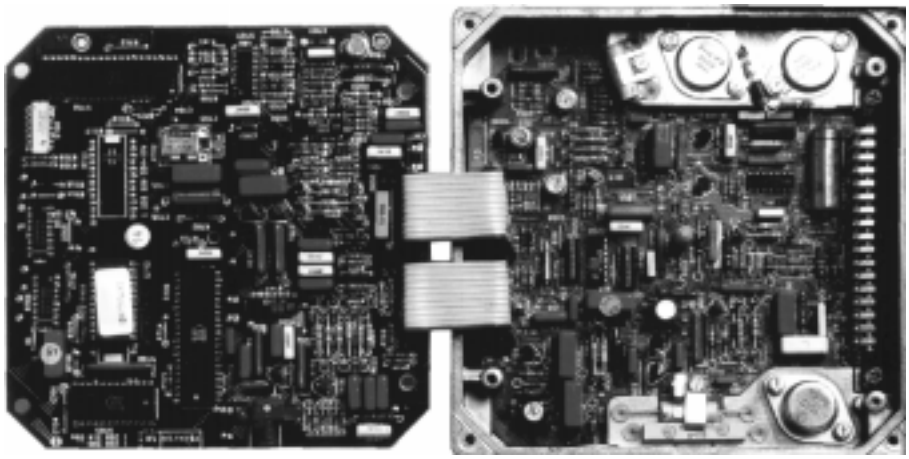


Bild 3.15: Die konventionelle Technologie der *teuren Gatter*

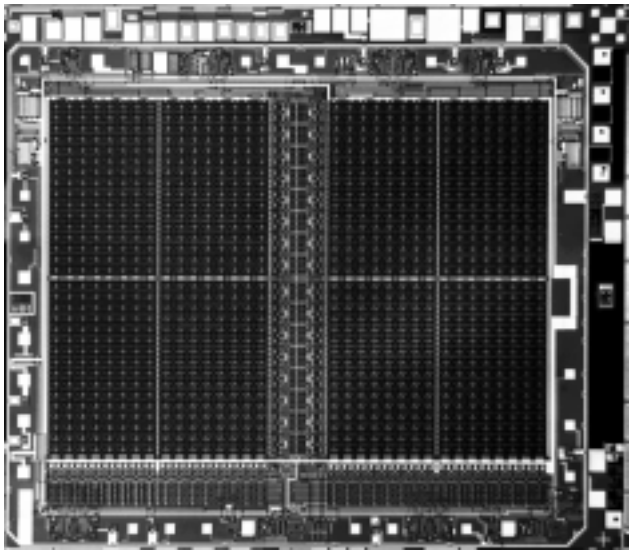


Bild 3.16: Die moderne Technologie der *teuren Verdrahtung*

Das bedeutet, daß das Entwurfsziel bei der konventionellen Technologie mit diskreten Bausteinen zunächst darin besteht, die Anzahl der Gatter und Eingangsklemmen zu minimieren (jede Eingangsgröße benötigt einen Transistor), während für die moderne Technologie der höchstintegrierten Schaltungen die Gesamtverdrahtung auf den Chips möglichst kurz ausgelegt werden muß. Doch bringt auch hier die Minimierung der Gatteranzahl in der Regel eine Verringerung des Verdrahtungsaufwandes mit sich. Deshalb sollen nachfolgend ausgewählte Minimierungsverfahren für logische Gatter vorgestellt werden, während Methoden der Verdrahtungsoptimierung, wie z.B. Floorplanning, nicht Gegenstand dieses Kapitels sind.

3.2.2 Der Hauptsatz der Schaltalgebra

Der Hauptsatz der Schaltalgebra besagt: „Jede Schaltfunktion $y = f(X)$ läßt sich durch zwei zueinander duale Strukturausdrücke darstellen.“ Zur Konstruktion der Strukturausdrücke benötigt man besondere Terme, die sogenannten Minterme *m* bzw. Maxterme *M*. Sie „adressieren“ die zu speziellen Eingangsbelegungen X_j gehörenden Funktionswerte $y_j \in \{0,1\}$, wie nachfolgend gezeigt werden wird.

- Ein *Minterm* $m_j(X)$ ist eine *Konjunktion* & aller *n* Komponenten eines *n*-Tupels, d.h. einer Eingangsbelegung *X*, die darin jeweils genau einmal - entweder bejaht oder verneint - vertreten sind. Die Konjunktion hat ihre *Einsstelle* bei $X = X_j$, d.h. $m_j(X_j) = 1$.
- Ein *Maxterm* $M_j(X)$ ist eine *Disjunktion* $\vee$ aller *n* Komponenten eines *n*-Tupels, d.h. einer Eingangsbelegung *X*, die darin jeweils genau einmal - entweder bejaht oder verneint - vertreten sind. Die Disjunktion hat ihre *Nullstelle* bei $X = X_j$, d.h. $M_j(X_j) = 0$.

Beispiel 3.5: $n = 2$; zu adressieren ist die spezielle Eingangsbelegung $Xj = (0, 1) = X_1$

Minterme *m*	Maxterme *M*
$m_0(X_1) = \bar{x}_2 \,\&\, \bar{x}_1 = 1 \,\&\, 0 = 0$	$M_0(X_1) = x_2 \vee x_1 = 0 \vee 1 = 1$
$m_1(X_1) = \bar{x}_2 \,\&\, x_1 = 1 \,\&\, 1 = \mathbf{1}$	$M_1(X_1) = x_2 \vee \bar{x}_1 = 0 \vee 0 = \mathbf{0}$
$m_2(X_1) = x_2 \,\&\, \bar{x}_1 = 0 \,\&\, 0 = 0$	$M_2(X_1) = \bar{x}_2 \vee x_1 = 1 \vee 1 = 1$
$m_3(X_1) = x_2 \,\&\, x_1 = 0 \,\&\, 1 = 0$	$M_3(X_1) = \bar{x}_2 \vee \bar{x}_1 = 1 \vee 0 = 1$

Die Konstruktion der nach dem Hauptsatz der Schaltalgebra existierenden beiden Strukturausdrükke erfolgt

- entweder durch Konjunktion aller Minterme m_j mit den jeweils zugehörigen Funktionswerten y_j und deren Verknüpfung durch eine Disjunktionskette (links)
- oder durch Disjunktion aller Maxterme M_j mit den jeweils zugehörigen Funktionswerten y_j und deren Verknüpfung durch eine Konjunktionskette (rechts).

Disjunktive Form	Konjunktive Form
DF $y = \bigvee(m_j \,\&\, y_j);\quad j = 0 \,..\, 2^n\text{-}1$	**KF** $y = \&(M_j \vee y_j);\quad j = 0 \,..\, 2^n\text{-}1$

Beispiel 3.2.4: $n = 2$; Liste der Antivalenz: $y_0 = 0 \oplus 0 = 0$; $y_1 = 0 \oplus 1 = 1$; $y_2 = 1 \oplus 0 = 1$; $y_3 = 1 \oplus 1 = 0$.

DF $y = \bigvee (m_j \,\&\, y_j)$; $j = 0 \,..\, 3$

$y = (m_0 \,\&\, y_0) \vee (m_1 \,\&\, y_1) \vee (m_2 \,\&\, y_2) \vee (m_3 \,\&\, y_3)$

$y = (m_0 \,\&\, 0) \vee (m_1 \,\&\, 1) \vee (m_2 \,\&\, 1) \vee (m_3 \,\&\, 0)$

$y = (0) \vee (m_1) \vee (m_2) \vee (0)$

$y = (m_1) \vee (m_2)$

$y = (\bar{x}_2 \,\&\, x_1) \vee (x_2 \,\&\, \bar{x}_1)$

KF $y = \&(M_j \vee y_j)$; $j = 0 \,..\, 3$

$y = (M_0 \vee y_0) \,\&\, (M_1 \vee y_1) \,\&\, (M_2 \vee y_2) \,\&\, (M_3 \vee y_3)$

$y = (M_0 \vee 0) \,\&\, (M_1 \vee 1) \,\&\, (M_2 \vee 1) \,\&\, (M_3 \vee 0)$

$y = (M_0) \,\&\, (1) \,\&\, (1) \,\&\, (M_3)$

$y = (M_0) \,\&\, (M_3)$

$y = (x_2 \vee x_1) \,\&\, (\bar{x}_2 \vee \bar{x}_1)$

Es genügt, bei der Konstruktion der Strukturausdrücke mit Mintermen nur die *Einsstellen* $y_j = 1$ der Schaltfunktion zu betrachten; entsprechend genügt bei Maxtermen die Betrachtung der *Nullstellen* $y_j = 0$. Man kommt so zu zwei einfacheren Formen des Hauptsatzes:

Disjunktive Normalform	Konjunktive Normalform
DNF $y = \bigvee m_j$; $j = 0 \,..\, 2^n\text{-}1$; $y_j = 1$	**KNF** $y = \& M_j$; $j = 0 \,..\, 2^n\text{-}1$; $y_j = 0$

Beispiel 3.2.5: $n = 2$; Liste der Antivalenz: $y_0 = 0 \oplus 0 = 0$; $y_1 = 0 \oplus 1 = 1$; $y_2 = 1 \oplus 0 = 1$; $y_3 = 1 \oplus 1 = 0$.

DNF $y = \bigvee m_j$; $j = 0 \,..\, 3$; $y_j = 1$

$y = (m_1) \vee (m_2)$

$y = (\bar{x}_2 \,\&\, x_1) \vee (x_2 \,\&\, \bar{x}_1)$

KNF $y = \& M_j$; $j = 0 \,..\, 3$; $y_j = 0$

$y = (M_0) \,\&\, (M_3)$

$y = (x_2 \vee x_1) \,\&\, (\bar{x}_2 \vee \bar{x}_1)$

Werden disjunktive bzw. konjunktive Normalformen (DNF bzw. KNF) als Strukturausdrücke betrachtet, so erkennt man deren Isomorphie zu zweistufigen Schaltnetzen mit UND-Gattern zur Implementierung der Konjunktionen (&) und ODER-Gattern für die Disjunktionen ($\vee$). Die folgenden beiden Bilder zeigen die Schaltnetzstrukturen, die zu der im obigen Beispiel ermittelten DNF der Antivalenz bzw. zu ihrer KNF isomorph ist.

Die Eigenschaften der Schaltalgebra sind in fünf Axiomen niedergelegt. Aus ihnen lassen sich alle Regeln und Sätze der Schaltalgebra herleiten. Im Hinblick auf technische Realisierungen, insbesondere durch elektronische Schaltkreise, ist die folgende Regel besonders wichtig:

DeMorgansche Regel

DNF $\overline{(\overline{a \vee b})} = \bar{a} \,\&\, \bar{b}$	KNF $\overline{(\overline{a \,\&\, b})} = \bar{a} \vee \bar{b}$

Liegt ein Strukturausdruck in DNF vor, so läßt er sich durch Anwendung der DeMorganschen Regel in eine einheitliche NAND-Form umwandeln, liegt er in KNF vor, in eine NOR-Form.

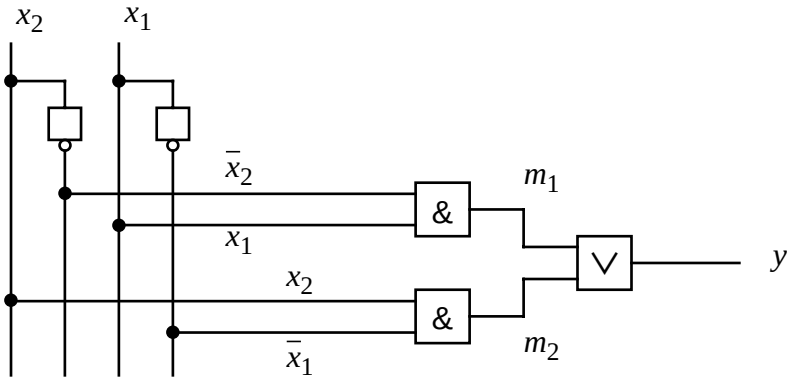


Bild 3.17: Schaltnetzstruktur der disjunktiven Normalform (DNF) der Antivalenz ($x_2 \oplus x_1$)

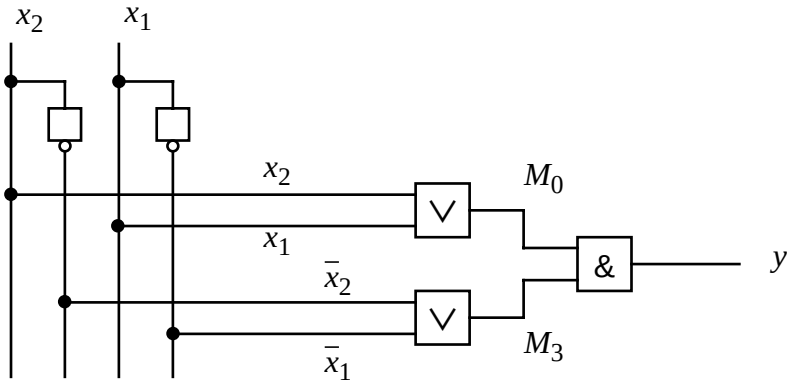


Bild 3.18: Schaltnetzstruktur der konjunktiven Normalform (KNF) der Antivalenz ($x_2 \oplus x_1$)

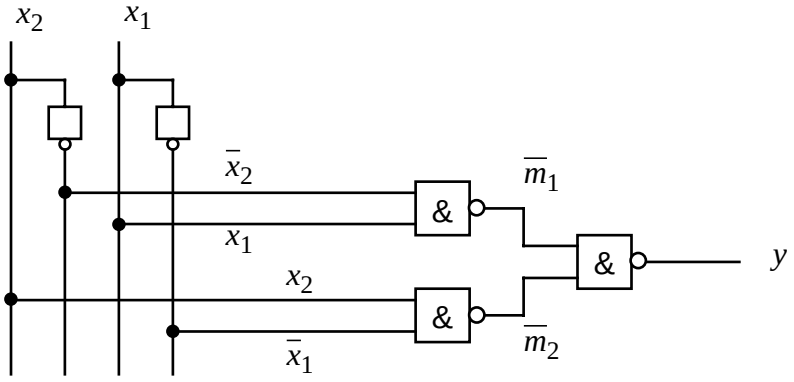


Bild 3.19: NAND-Schaltnetzstruktur aus der DNF der Antivalenz ($x_2 \oplus x_1$)

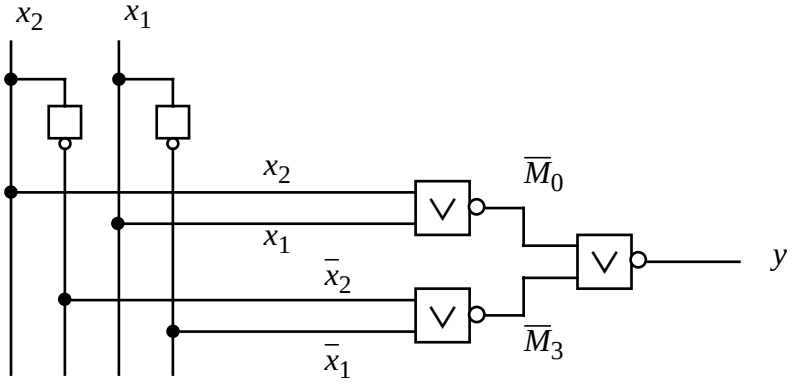


Bild 3.20: NOR-Schaltnetzstruktur aus der KNF der Antivalenz ($x_2 \oplus x_1$)

Beispiel 3.2.6: Antivalenz $y = (x_2 \oplus x_1)$; vgl. die Schaltnetzstrukturen in obigen beiden Bildern.

DNF $y = (\bar{x}_2 \,\&\, x_1) \vee (x_2 \,\&\, \bar{x}_1)$

$\bar{y} = \overline{(\bar{x}_2 \,\&\, x_1) \vee (x_2 \,\&\, \bar{x}_1)}$

$\bar{y} = \overline{(\bar{x}_2 \,\&\, x_1)} \,\&\, \overline{(x_2 \,\&\, \bar{x}_1)}$

in Kurzschreibweise:

$y = (\bar{x}_2 \,\overline{\&}\, x_1) \,\overline{\&}\, (x_2 \,\overline{\&}\, \bar{x}_1)$

KNF $y = (x_2 \vee x_1) \,\&\, (\bar{x}_2 \vee \bar{x}_1)$

$\bar{y} = \overline{(x_2 \vee x_1) \,\&\, (\bar{x}_2 \vee \bar{x}_1)}$

$\bar{y} = \overline{(x_2 \vee x_1)} \vee \overline{(\bar{x}_2 \vee \bar{x}_1)}$

in Kurzschreibweise:

$y = (x_2 \,\overline{\vee}\, x_1) \,\overline{\vee}\, (\bar{x}_2 \,\overline{\vee}\, \bar{x}_1)$

3.2.3 Grundlagen der Minimierung von Schaltnetzen

Es soll zunächst der Zusammenhang zwischen Termen und den Feldern im KV-Diagramm geklärt werden.

a) Mintermdarstellung

Gegeben sei ein n-Tupel X mit n Komponenten x_i , wobei $i = 1 \,..\, n$, für $n = 3$ mit der speziellen Belegung:

$X_j = (0, 1, 1) = X_3$

Der *Minterm* mit demselben Index $j = 3$ lautet:

$m_3(X) = \bar{x}_3 \,\&\, x_2 \,\&\, x_1$

Er ist so definiert, daß er seine Einsstelle bei X_3 hat:

$m_3(X_3) = \bar{0} \,\&\, 1 \,\&\, 1 = 1$

Deshalb bezeichnet man auch die Belegung X_j selbst als die Einsstelle E des Minterms m_j :

$E(m_j) = \{ X_j \mid j = 3 \}$

$E(m_3) = \{ X_3 \} = \{(0, 1, 1)\}$

Im nebenstehenden KV-Diagramm wird das markierte *Feld* von dieser Einsstelle, d.h. vom *Minterm* m_3 adressiert.

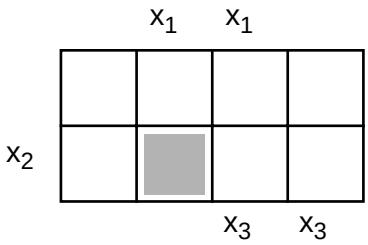


Bild 3.21: Feld im KV-Diagramm

Ein im Gegensatz zum Minterm willkürlich gewählter *Term* möge lauten:

$w(X) = \bar{x}_3 \,\&\, x_2$

Da hier der Wert der Komponente x_1 unerheblich ist, besitzt der Term w nicht nur eine einzelne Einsstelle E, sondern eine Einsstellenmenge $\underline{E}$:

$\underline{E}(w) = \{ X_j \mid j = 2 \,..\, 3 \}$

$= \{ X_2, X_3 \} = \{(0, 1, 0), (0, 1, 1)\}$

Man pflegt dann auch zu schreiben:

$\underline{E}(w) = \{ X_{2,3} \} = \{(0, 1, -)\}$

Im nebenstehenden KV-Diagramm wird der in markierte *Block* von dieser Einsstellenmenge, d.h. vom gewählten *Term* adressiert.

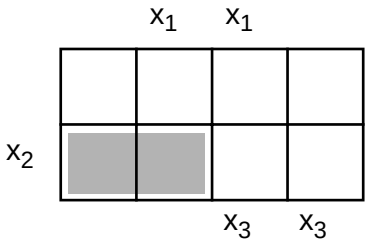


Bild 3.22: Block im KV-Diagramm

b) Maxtermdarstellung

Gegeben sei nach wie vor der 3-Tupel mit der speziellen Belegung:

$X_j = (0, 1, 1) = X_3$

Der *Maxterm* mit demselben Index $j = 3$ lautet:

$M_3(X) = x_3 \vee \bar{x}_2 \vee \bar{x}_1$

Er ist so definiert, daß er seine Nullstelle bei X_3 hat:

$M_3(X_3) = 0 \vee \bar{1} \vee \bar{1} = 0$

Deshalb bezeichnet man auch die Belegung X_j selbst als die Nullstelle N des Maxterms M_j :

$N(M_j) = \{ X_j \mid j = 3 \}$
$N(M_3) = \{ X_3 \} = \{(0, 1, 1)\}$

Im obigen KV-Diagramm wird das markierte *Feld* auch von dieser Nullstelle, d.h. vom *Maxterm* M_3 adressiert.

Ein im Gegensatz zum Maxterm willkürlich gewählter *Term* möge lauten:

$W(X) = x_3 \vee \bar{x}_2$

Da hier der Wert der Komponente x_1 unerheblich ist, besitzt der Term nicht nur eine einzelne Nullstelle N, sondern eine Nullstellenmenge $\underline{N}$:

$\underline{N}(W) = \{ X_j \mid j = 2 \,..\, 3 \}$
$= \{ X_2, X_3 \} = \{(0, 1, 0), (0, 1, 1)\} = \{(0, 1, -)\}$

Im obigen KV-Diagramm wird der markierte *Block* auch von dieser Nullstellenmenge, d.h. vom gewählten *Term* adressiert.

c) Freiheitsgrade

Ein *Block B* ist die Menge der n-Tupel X, die in den Werten bestimmter Komponenten x_i übereinstimmen, d.h. er ist eine Untermenge der Menge $\underline{X}$ aller Eingangsbelegungen X :

$B \subseteq \underline{X} = \{ 0, 1 \}^n$

Für *gebundene* Komponenten ist einer der beiden Werte $x_i \in \{ 0, 1 \}$ vorgeschrieben, dagegen muß jede *freie* Komponente mit beiden Werten belegt werden.

Beispiel 3.2.7:

$X = (x_5, x_4, x_3, x_2, x_1)$; gebunden seien x_4, x_2, x_1; frei seien x_5, x_3

Der Block ist dann wie folgt zu schreiben: $B = \{(-, 1, -, 0, 1)\}$

Es sei hier ausdrücklich auf die „Blockschreibweise“ hingewiesen:

- Die Notation mit einem *Strich* (-) bedeutet, daß für eine *Eingangsvariable* x_i beide Werte, d.h. $x_i = 0$ *und* $x_i = 1$ betrachtet werden müssen.

Man stellt fest:

- Ein Block mit r freien Komponenten besteht aus 2^r Belegungen.

Allerdings kann nicht jede Belegungsmenge ein Block von Belegungen sein. Wie in den folgenden KV-Diagrammen gezeigt, umfaßt ein Block stets eine 2-er Potenz symmetrisch angeordneter Felder. Zu jedem Block läßt sich genau eine Konjunktion w und genau eine Disjunktion W konstruieren.

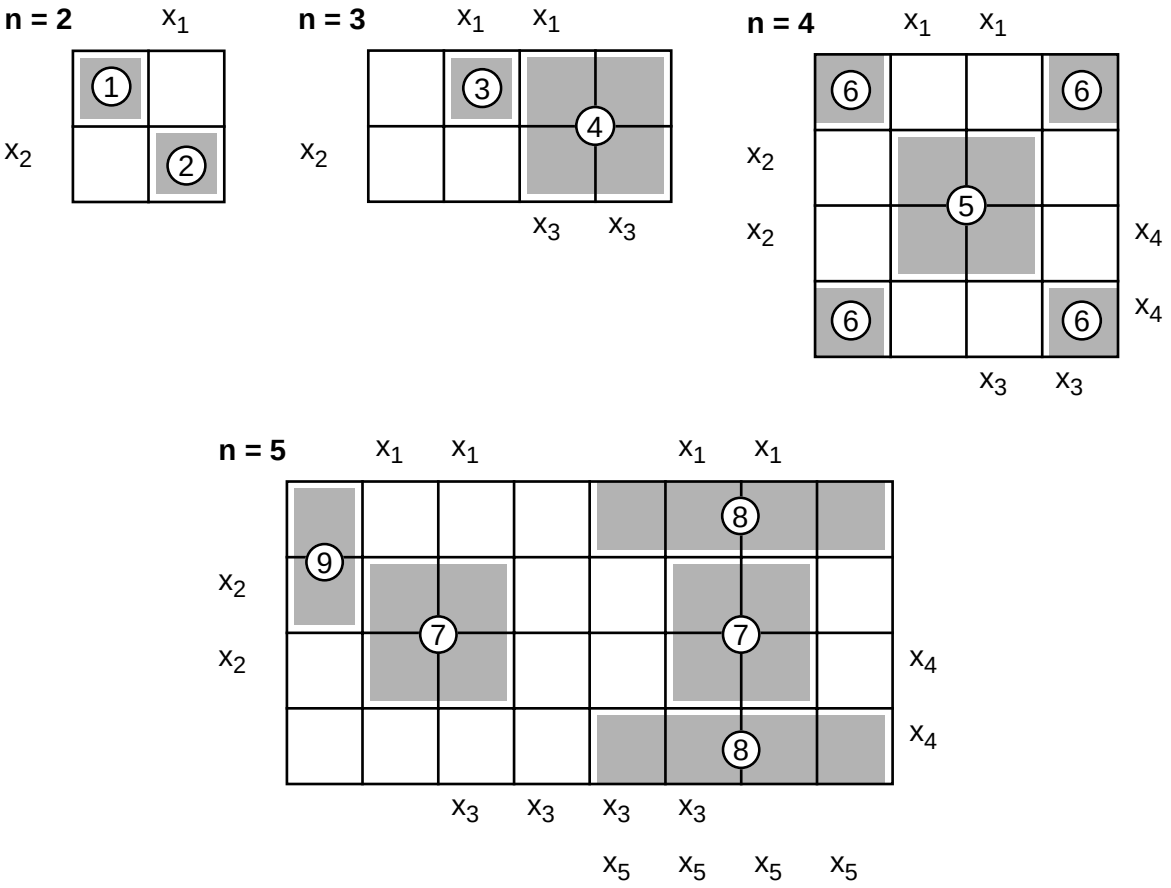


Bild 3.23: Ausgewählte Blöcke und Terme in KV-Diagrammen

(1) m_0; M_0

(2) m_3; M_3

(3) m_1; M_1

(4) $w = x_3$; $W = \bar{x}_3$

(5) $w = x_2 \,\&\, x_1$; $W = \bar{x}_2 \vee \bar{x}_1$

(6) $w = \bar{x}_2 \,\&\, \bar{x}_1$; $W = x_2 \vee x_1$

(7) $w = x_2 \,\&\, x_1$; $W = \bar{x}_2 \vee \bar{x}_1$

(8) $w = x_5 \,\&\, \bar{x}_2$; $W = \bar{x}_5 \vee x_2$

(9) $w = \bar{x}_5 \,\&\, \bar{x}_4 \,\&\, \bar{x}_3 \,\&\, \bar{x}_1$;
$W = x_5 \vee x_4 \vee x_3 \vee x_1$

Man vereinbart folgende Begriffe:

- Ein Block, der ausschließlich Einsstellen einer Schaltfunktion $y = f(X)$ umfaßt, heißt „Einsblock“; der zugehörige Term heißt „Implikant“.
- Ein Block, der ausschließlich Nullstellen einer Schaltfunktion $y = f(X)$ umfaßt, heißt „Nullblock“; der zugehörige Term heißt „Implikat“.

Beispiel 3.2.8:

Gegeben seien Eins- und Nullstellenmenge einer Schaltfunktion wie im nebenstehenden KV-Diagramm dargestellt.
Umrahmt sind zwei Blöcke:

- Ein Einsblock $B_E = \{(0, 1, 1, -)\}$;
  der zugehörige Implikant lautet $w_E = \bar{x}_4 \,\&\, x_3 \,\&\, x_2$
- Ein Nullblock $B_N = \{(0, 1, 0, -)\}$;
  das zugehörige Implikat lautet $W_N = x_4 \vee \bar{x}_3 \vee x_2$

	x_1	x_1		
	1	1	0	0
x_2	0	0	1	1
x_2	0	0	1	1
	0	0	0	0
			x_3	x_3

x_4 (rechts neben der dritten Zeile)
x_4 (rechts neben der vierten Zeile)

Bild 3.24: Null- und Einsblock

Je größer ein Block, desto kürzer der zugehörige Term.

- Die größtmöglichen Eins- bzw. Nullblöcke einer Schaltfunktion heißen „Primblöcke“; die zugehörigen Terme heißen „Primterme“.

Man ist nicht nur bestrebt, die *größtmöglichen* Blöcke zu bilden, um die *kürzestmöglichen* Terme zu erhalten, sondern auch die jeweils kleinere der Eins- bzw. Nullstellenmenge einer Schaltfunktion mit möglichst *wenigen* Primblöcken vollständig zu überdecken, da sie *wenigen* Primtermen im Strukturausdruck entsprechen, so daß der Implementierungsaufwand minimiert wird. Bei der Umsetzung des ermittelten Strukturausdrucks in ein isomorphes Schaltnetz entspricht jeder Term einem Gatter, jeder Operand im Term einer Eingangsklemme des Gatters, so daß kurze Terme kostengünstiger sind.

Der systematische Entwurf eines Schaltnetzes erfolgt grundsätzlich in drei Schritten:

- Ermittlung *aller* Primblöcke und damit aller Primterme der gegebenen Schaltfunktion;
- Ermittlung *aller* vollständigen irredundanten Überdeckungen der Schaltfunktion;
- Auswahl *einer* minimalen irredundanten Überdeckung.

3.2.4 Minimierung nach Karnaugh-Veitch

Die graphische Methode nach Karnaugh-Veitch eignet sich zum Handentwurf kleinerer Schaltnetze. Gegeben sei die Einsstellenmenge einer Schaltfunktion durch Einträge in das folgende KV-Diagramm. Mit Mintermen m_j ergibt sich die im Bild links gezeigte Überdeckung. Die Schaltfunktion sei vollständig definiert; daher erhält man ihre Nullstellenmenge durch Ergänzen der gegebenen Einsstellenmenge. Mit Maxtermen M_j ergibt sich deren im Bild rechts gezeigte Überdeckung.

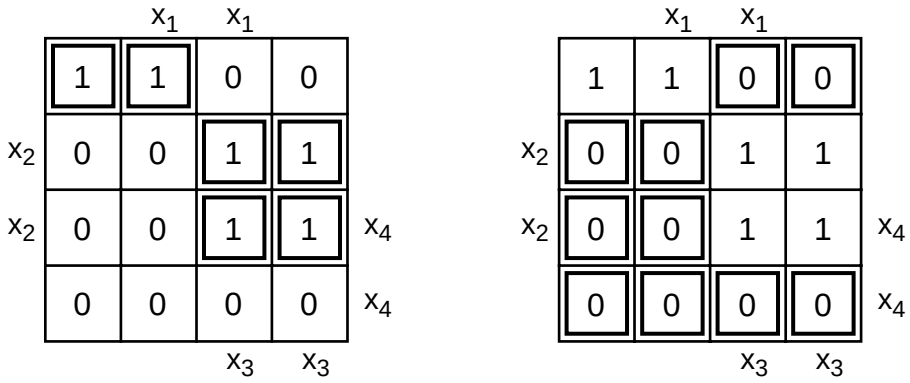


Bild 3.25: Vollständige Überdeckung der Eins- bzw. Nullstellenmenge einer Schaltfunktion; links: mit Mintermen, rechts: mit Maxtermen

Man könnte nun mit den Mintermen eine disjunktive Normalform (DNF) bilden mit sechs UND-Gattern zu deren Erzeugung (&) und einem ODER-Gatter zu deren Disjunktion ($\vee$), mit den Maxtermen eine konjunktive Normalform (KNF) mit sogar zehnODER-Gattern zu deren Erzeugung ($\vee$) und einem UND-Gatter zu deren Konjunktion (&). Diesen Aufwand gilt es zu minimieren.

a) Ermittlung aller Primblöcke und Primterme

Für die gegebene Schaltfunktion wurden in folgenden KV-Diagrammen alle Primblöcke gebildet:

Prim-Einsblöcke:

$B_E = \{(0, 0, 0, -), (-, 1, 1, -)\}$

Prim-Nullblöcke:

$B_N = \{(-, 1, 0, -), (-, 0, 1, -), (1, 0, -, -), (1, -, 0, -)\}$

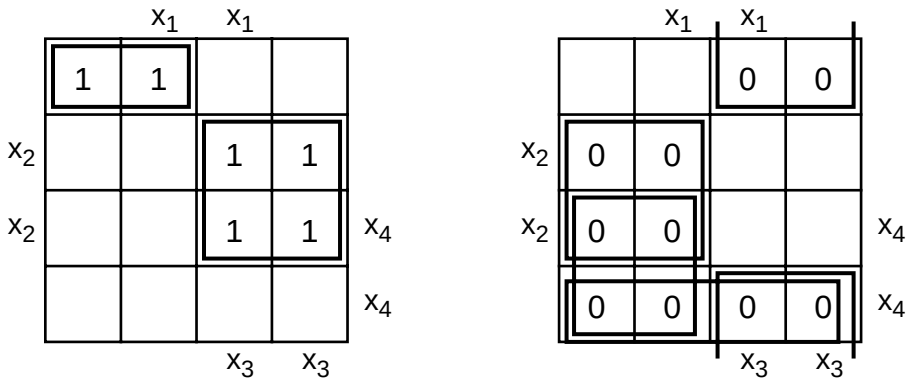


Bild 3.26: Vollständige Überdeckung der Eins- bzw. Nullstellenmenge einer Schaltfunktion; links: mit allen Prim-Einsblöcken, rechts: mit allen Prim-Nullblöcken.

Die Primeinsblöcke B_E bzw. die Primnullblöcke B_N entsprechen folgenden Primtermen:

Primimplikanten:

$p_A = \bar{x}_4\ \bar{x}_3\ \bar{x}_2$; $p_B = x_3\ x_2$

Primimplikate:

$P_A = \bar{x}_3 \vee x_2$; $P_B = x_3 \vee \bar{x}_2$;
$P_C = \bar{x}_4 \vee x_3$; $P_D = \bar{x}_4 \vee x_2$

b) Auswahl einer irredundanten Überdeckung

Es ist eine Auswahl aus den Primblöcken zu treffen, so daß sich mit möglichst wenigen Blöcken eine vollständige Überdeckung der Eins- bzw. Nullstellenmenge der Schaltfunktion ergibt („irredundante Überdeckung"). Im Fall der Einsstellen ist die Auswahl trivial, im Fall der Nullstellen gibt es zwei gleich aufwendige Möglichkeiten, wie das folgende Bild zeigt.

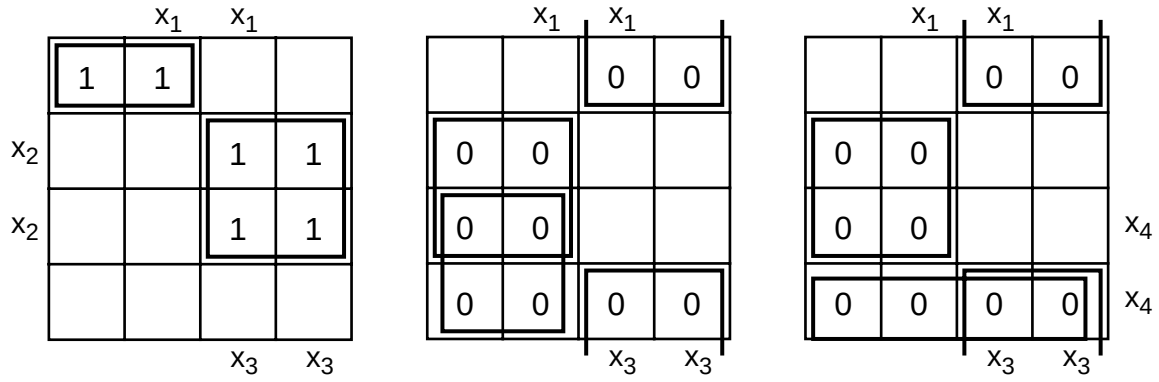


Bild 3.27: Irredundante Überdeckungen der Eins- bzw. Nullstellen der Schaltfunktion

Disjunktive Minimalform

DMF $y = p_A \vee p_B$
$= \bar{x}_4\ \bar{x}_3\ \bar{x}_2 \vee x_3\ x_2$

Konjunktive Minimalform

KMF $y = P_A\ P_B\ P_C$
$= (\bar{x}_3 \vee x_2)\ (x_3 \vee \bar{x}_2)\ (\bar{x}_4 \vee x_3);$
$y = P_A\ P_B\ P_D$
$= (\bar{x}_3 \vee x_2)\ (x_3 \vee \bar{x}_2)\ (\bar{x}_4 \vee x_2)$

Die Überdeckung der Einsstellen, d.h. die disjunktive Minimalform (DMF) führt bei diesem Beispiel zu minimalem Gatteraufwand. Die zur DMF isomorphe zweistufige Schaltung ist hier gezeigt.

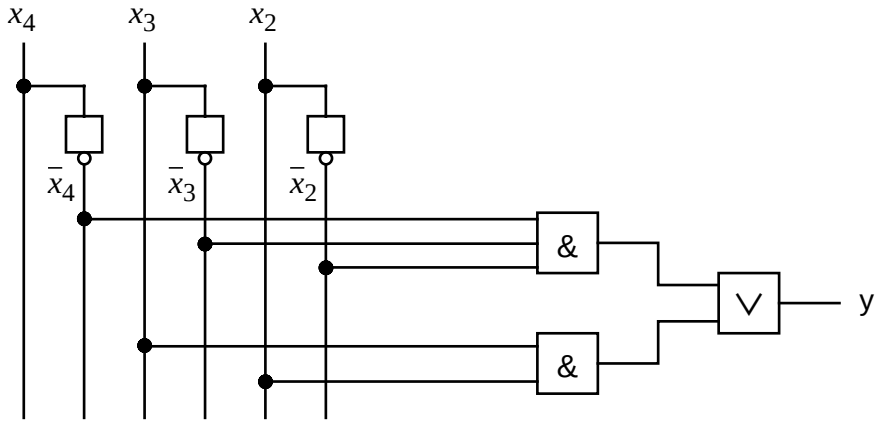


Bild 3.28: Minimierte Gatterschaltung in zweistufiger disjunktiver Minimalform (DMF)

3.2.5 Minimierung nach Quine-McCluskey

Die Methode nach Quine-McCluskey erlaubt die Ermittlung aller Primblöcke und damit aller Primterme einer Schaltfunktion. Es ist ein schaltalgebraisches Verfahren, das sich für den Handentwurf eignet, aber auch schon in rechnergestützten Verfahren der Logiksynthese implementiert wurde. Die Größe der synthetisierbaren Schaltungen wird jedoch durch den Speicherbedarf stark begrenzt,

so daß ein Syntheselauf unter Umständen vorzeitig abbricht, da der Tabellenbedarf dieser Methode zwischenzeitlich stark anwachsen kann, auch wenn er gegen Ende des Verfahrens wieder deutlich abnimmt.

Die Auswahl einer vollständigen minimalen irredundanten Überdeckung, z.B. aller Einsstellen der spezifizierten Schaltfunktion, aus den ermittelten Primblöcken ist nicht Bestandteil des Verfahrens nach Quine-McCluskey und muß daran anschließend durchgeführt werden. Im nächsten Abschnitt wird ein geeignetes Verfahren vorgestellt.

a) Verfahrensschritte

Gegeben sei die Einsstellenmenge einer Schaltfunktion, z.B. in disjunktiver Normalform (DNF). Alternativ kann auch ihre Nullstellenmenge in konjunktiver Normalform (KNF) dargestellt sein.

- Man schreibt die binären Eingangsbelegungen X_j, die den Einsstellen $y = 1$ der Schaltfunktion zugeordnet sind, *mit wachsendem Gewicht*, d.h. mit wachsender Anzahl von gebundenen Komponenten $x_i = 1$ in einer Liste untereinander.
- Belegungen, die sich in einer Komponente unterscheiden, d.h. in benachbarten Gewichtsklassen stehen, werden zu *Einsblöcken* zusammengefaßt und in eine neue Liste eingetragen.
- Einsblöcke, die die gleichen freien Komponenten haben und sich in nur einer gebundenen Komponente unterscheiden, werden zu *größeren Einsblöcken* zusammengefaßt und in eine neue Liste eingetragen.
- Blöcke, die zusammengefaßt werden konnten, werden abgehakt. Sie sind nicht prim, nehmen aber an weiteren Vergleichen teil.
- Blöcke, die nicht weiter zusammengefaßt werden konnten, sind die gesuchten *Primblöcke*.

Beispiel 3.2.9:

Gegeben sei ein 4-Tupel $X = (x_3, x_2, x_1, x_0)$ als Eingangsbelegung einer Schaltfunktion $y = f(X)$ sowie deren Einsstellenmenge $\underline{E}\,(w)$ durch die Menge der oktalen Indizes j derjenigen Eingangsbelegungen X_j, für die $y = f(X_j) = 1$ gilt: $\underline{E}\,(w) = \{\ 0, 1, 3, 4, 6, 10, 14, 17\}$

b) Ermittlung aller Primterme

In der folgenden Liste werden die gegebenen oktalen Indizes j in Dualzahlen umgewandelt, die nach Vereinbarung den binären 4-Tupeln, d.h. den Eingangsbelegungen X_j entsprechen:

j	0	1	3	4	6	10	14	17
Belegungen	0 000	0 001	0 011	0 100	0 110	1 000	1 100	1 111

Die binären Eingangsbelegungen werden mit wachsendem Gewicht in folgender Liste untereinander geschrieben. Man verfährt weiter nach den oben angegebenen Verfahrensschritten:

	Gewicht	*j*	Belegungen
	0	0	0 000
	1	1	0 001
		4	0 100
		10	1 000
	2	3	0 011
		6	0 110
		14	1 100
	3		
A	4	17	1 111

	Gewicht	*j*	Belegungen
B	0	0, 1	0 00-
		0, 4	0 -00
		0, 10	- 000
C	1	1, 3	0 0-1
D		4, 6	0 1-0
		4, 14	- 100
		10, 14	1 -00
	2		
	3		

	Gewicht	*j*	Belegungen
E	0	0, 4,	- -00
		10, 14	
	1		
	2		

	überdeckte Einsstellen	Primblöcke
A	17	1 111
B	0, 1	0 00-
C	1, 3	0 0-1
D	4, 6	0 1-0
E	0, 4, 10, 14	- -00

Den ermittelten Primblöcken A . . E entsprechen die folgenden Primimplikanten:

$$p_A = x_3\ x_2\ x_1\ x_0$$
$$p_B = \bar{x}_3 \bar{x}_2 \bar{x}_{\bar{1}}$$
$$p_C = \bar{x}_3 \bar{x}_2\ x_0$$
$$p_D = \bar{x}_3\ x_2\ \bar{x}_0$$
$$p_E = \bar{x}_1\ \bar{x}_0$$

3.2.6 Lösung des Überdeckungsproblems

Aus der Gesamtheit aller Primblöcke, die z.B. mit dem Verfahren nach Quine-McCluskey ermittelt worden sind, ist eine Auswahl zur vollständigen minimalen irredundanten Überdeckung aller Eins- oder aller Nullstellen der spezifizierten Schaltfunktion zu treffen. Dazu sind die Überdeckungstabelle und die Petrick-Funktion geeignet, wie hier an der Fortführung des obigen Beispiels gezeigt werden soll.

a) mit der Überdeckungstabelle

- Es wird eine Überdeckungstabelle aufgestellt, deren Spalten mit den Indizes *j* der gegebenen Einsstellen und deren Zeilen mit allen ermittelten Primblöcken (hier: A . . E) bezeichnet werden. Die *Markierungen* (×) geben an, welche Einsstellen von den Primblöcken jeweils überdeckt werden. Man kann sie z.B. anhand der Listen des Verfahrens nach Quine-McCluskey setzen.
- Man trifft eine *Auswahl* der Primblöcke, so daß alle Einsstellen der Schaltfunktion mindestens einmal überdeckt werden. Man arbeitet die Tabelle von rechts nach links ab: Blöcke, die als einzige eine Einsstelle überdecken, sind unverzichtbare „Kernblöcke“ (hier: A, E, D, C).
- Man bildet die *Disjunktion* der Primimplikanten, die den ausgewählten Primeinsblöcken entsprechen.
- Die so gewonnenene disjunktive Minimalform (DMF) der gegebenen Schaltfunktion kann in eine isomorphe zweistufige *Gatterstruktur* umgesetzt werden.

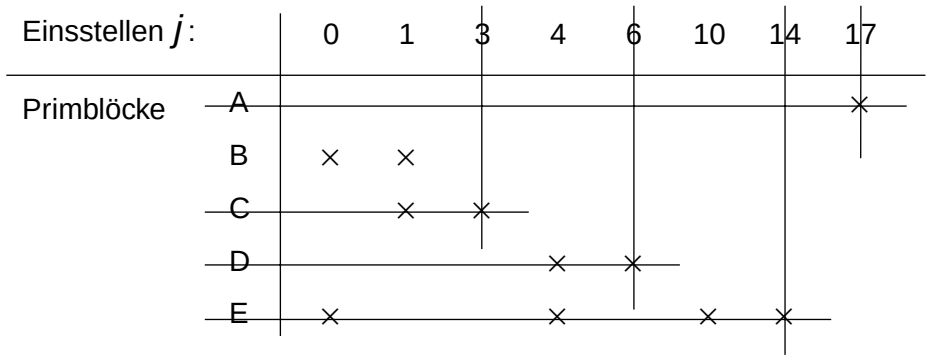


Bild 3.29: Überdeckungstabelle der gegebenen Schaltfunktion mit Auswahl der unverzichtbaren Kernblöcke. Überdeckt werden $j = 17$ von A, $j = 14$ und 10 von E, $j = 6$ von D, $j = 3$ von C. Redundant ist Block B: $j = 0$ wird von E, $j = 1$ von C überdeckt.

Den im obigen Bild ausgewählten Prim-Einsblöcken entsprechen die nachfolgenden Primimplikanten; ihre Disjunktion ergibt die disjunktive Minimalform (DMF) der gegebenen Schaltfunktion:

DMF $y = p_A \vee p_C \vee p_D \vee p_E$

$y = (x_3\, x_2\, x_1\, x_0) \vee (\bar{x}_3\, \bar{x}_2\, x_0) \vee (\bar{x}_3\, x_2\, \bar{x}_0) \vee (\bar{x}_1\, \bar{x}_0)$

b) mit der Petrick-Funktion

Die Ermittlung von Primtermen ist beim Entwurf von Schaltnetzen eine *universell* und auf verschiedenen Ebenen der Komplexität einsetzbare Methode:

- Aus den Primtermen einer Schaltfunktion werden deren Minimalformen gebildet.
- Die Primterme der Petrick-Funktion, der wiederum die Primterme einer Schaltfunktion zugrunde liegen, sind die irredundanten Überdeckungen dieser Schaltfunktion.

Zur Ermittlung der Primterme der Petrick-Funktion kann selbstverständlich jedes bekannte Verfahren dienen:

- die graphische Methode nach Karnaugh-Veitch mit KV-Diagrammen;
- das oben vorgestellte Verfahren nach Quine-McCluskey;
- eine schaltalgebraische Umformung der Petrick-Funktion.

Die Petrick-Funktion sagt für das obige Beispiel aus, daß sich die Einsstelle $j = 0$ mit den Blöcken B *oder* E überdecken läßt, die Einsstelle $j = 1$ mit den Blöcken B *oder* C usf. Von welchen Primblöcken die Einsstellen jeweils überdeckt werden, läßt sich aus den Listen des Verfahrens nach Quine-McCluskey entnehmen oder aus den Markierungen der Überdeckungstabelle. Für das obige Beispiel ergibt sich die Petrick-Funktion zu:

PF $= (B \vee E) \,\&\, (B \vee C) \,\&\, C \,\&\, (D \vee E) \,\&\, D \,\&\, E \,\&\, E \,\&\, A$

Schaltalgebraisch erhält man nach dem Absorptionsgesetz:

$$\begin{aligned} PF &= [(B \vee E) \,\&\, E \,\&\, E] \,\&\, [(B \vee C) \,\&\, C] \,\&\, [(D \vee E) \,\&\, D] \,\&\, A \\ &= [E] \,\&\, [C] \,\&\, [D] \,\&\, A \\ &= E \,\&\, C \,\&\, D \,\&\, A = A \,\&\, C \,\&\, D \,\&\, E \end{aligned}$$

Die Disjunktion der Primimplikanten, die den ausgewählten Primeinsblöcken entsprechen, ergibt wieder die disjunktive Minimalform (DMF) der gegebenen Schaltfunktion:

DMF $y = p_A \vee p_C \vee p_D \vee p_E$

$$y = (x_3\, x_2\, x_1\, x_0) \vee (\bar{x}_3\, \bar{x}_2\, x_0) \vee (\bar{x}_3\, x_2\, \bar{x}_0) \vee (\bar{x}_1\, \bar{x}_0)$$

Für dieses einfache Beispiel existiert nur ein Primterm der Petrickfunktion, d.h. nur eine irredundante Überdeckung. Aus der Überdeckungstabelle war dasselbe Ergebnis hervorgegangen: A, C, D und E sind Kernblöcke, Block B ist redundant. In einem komplizierteren Fall, aber mit ebenfalls fünf Primblöcken V, W, X, Y und Z könnte sich folgende Form der Petrick-Funktion ergeben:

PF $= (V \,\&\, X \,\&\, Z) \vee (V \,\&\, W \,\&\, Y \,\&\, Z)$

Diese wäre folgendermaßen zu interpretieren:

DMF $y = p_V \vee p_X \vee p_Z$ 1. Überdeckung

DMF $y = p_V \vee p_W \vee p_Y \vee p_Z$ 2. Überdeckung

Beides sind gültige irredundante Überdeckungen der Einsstellenmenge der Schaltfunktion mit den genannten Primblöcken bzw. den zugehörigen Primtermen. Die minimale Überdeckung ist die mit den wenigsten Primtermen. Hier benötigt man zur Implementierung drei UND-Gatter für die drei Primterme p_V, p_X und p_Z sowie ein ODER-Gatter mit drei Eingängen zur Bildung der Disjunktion.

3.2.7 Entwurfsbeispiel eines Schaltnetzes

Die Funktion eines binären Volladdierers sei durch seine Wahrheitstabelle unten links beschrieben. Er nimmt zwei binäre Variablen x_0 und x_1 sowie einen Übertrag x_2 aus einer vorherigen Additionsstufe entgegen und bildet daraus die binäre Summe *S* (”Sum”) sowie den neuen Übertrag *C* (”Carry”). Die Funktionsspalten für *S* und *C* werden in die zugehörigen KV-Diagramme übernommen.

j	x_2	x_1	x_0	S	C
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

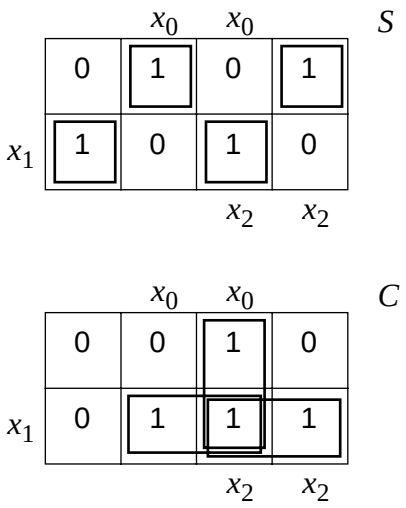


Bild 3.30: Funktionsbeschreibung eines Volladdierers (Wahrheitstabelle bzw. KV-Diagramme)

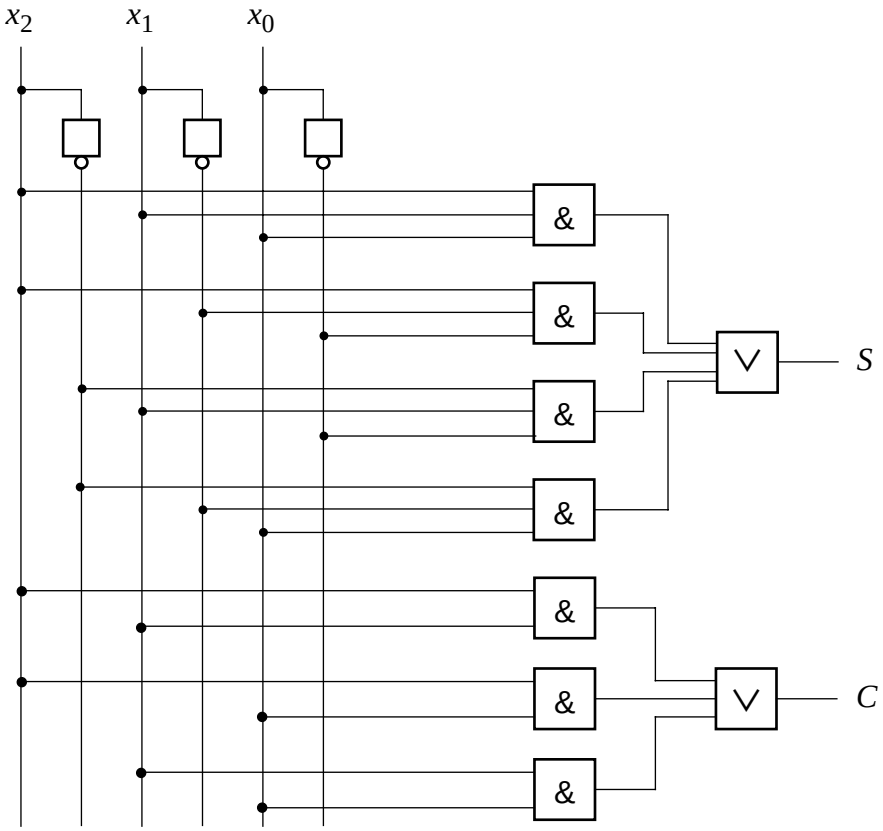


Bild 3.31: Schaltnetze zur Bildung der Summe *S* und des Übertrags *C*, isomorph zur DMF

Als disjunktive Minimalformen (DMF) der beiden Schaltfunktionen für die Summe *S* bzw. den Übertrag *C* erhält man durch Primblockbildung in den obigen beiden KV-Diagrammen:

$$S = (x_2\, x_1\, x_0) \vee (x_2\, \bar{x}_1\, \bar{x}_0) \vee (\bar{x}_2\, x_1\, \bar{x}_0) \vee (\bar{x}_2\, \bar{x}_1\, x_0) \qquad C = (x_2\, x_1) \vee (x_2\, x_0) \vee (x_1\, x_0)$$

Das dazu isomorphe Schaltnetz des Volladdierers zeigt das vorhergehende Schaltbild.

Man stellt bereits anhand obiger Strukturausdrücke fest, vor allem aber ist es aus dem Schaltbild zu erkennen, daß der Bedarf an logischen Gattern erheblich ist, vor allem deshalb, weil sich die Schaltfunktion für die Summe *S* nicht minimieren läßt. Auch eine Umformung nach der DeMorganschen Regel in ein ausschließlich aus NAND-Gattern bestehendes Schaltnetz würde keine Verringerung des Aufwandes bringen. Stehen jedoch nicht nur Grundgatter zur Verfügung, wie UND-, ODER-, NAND- bzw. NOR-Gatter, sondern komplexere Bausteine, wie Antivalenz- und Äquivalenz-Gatter sowie Multiplexer, so läßt sich der Bedarf an Bausteinen verringern und der Entwurfsablauf vereinfachen.

3.3 Sequentielle Schaltungen („Schaltwerke“)

3.3.1 Rückkopplung und Betriebsarten

Ergänzt man eine kombinatorische Schaltung („Schaltnetz”) durch eine Rückkopplung, so entsteht eine sequentielle Schaltung („Schaltwerk”), wie das folgende Bild zeigt. Schaltwerke sind konkrete Implementierungen abstrakter Automaten. Der Automatenbegriff, wie er in diesem Zusammenhang verstanden wird, soll im nächsten Abschnitt geklärt werden.

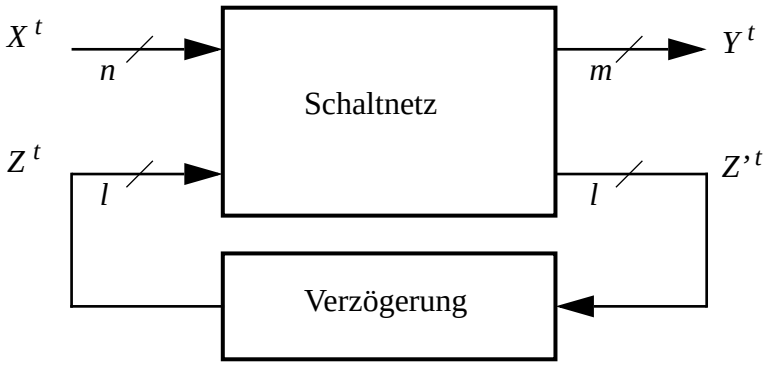


Bild 3.32: Allgemeine Betriebsart einer sequentiellen Schaltung („Schaltwerk”)

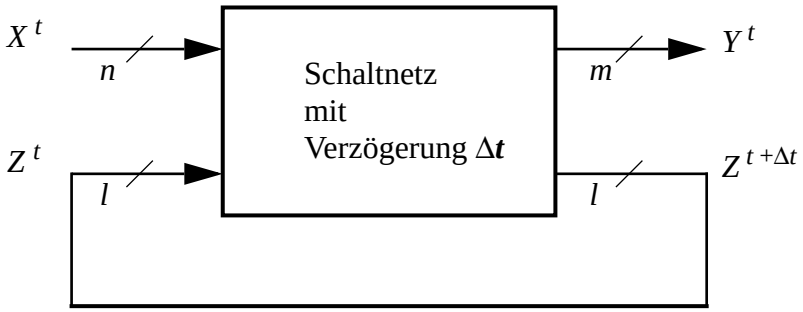


Bild 3.33: Ungetaktete („asynchrone”) Betriebsart eines Schaltwerks

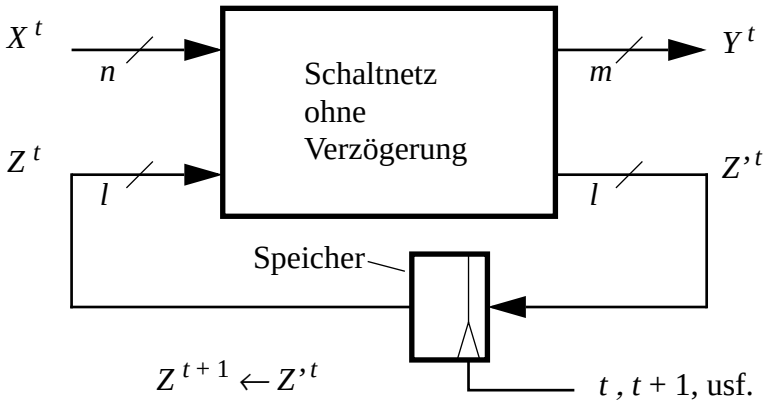


Bild 3.34: Getaktete („synchrone”) Betriebsart eines Schaltwerks

3.3.2 Ein/Ausgabe und Zustandsbegriff

Zunächst sollen die für Schaltwerke charakteristischen Begriffe geklärt werden, insbesondere der Zustandsbegriff, der sich aus der Rückkopplung von Ausgangsgrößen auf den Eingang ergibt. Wir wollen uns im Folgenden auf die getaktete („synchrone") Betriebsart konzentrieren.

- Eine Eingangsbelegung ist ein *n*-Tupel mit *n* binären Komponenten x_i :

  $X = (x_1, x_2, . . x_i . . , x_n);$ $\qquad x_i \in \{ 0, 1 \}; i = 1 . . n$

  Für die Eingangsbelegung gilt $X \in E$, wobei *E* das Eingabealphabet des Automaten ist. Eine bestimmte Eingangsbelegung wird mit X_j bezeichnet; dabei ist der oktale Index *j* gleich den als Dualzahl gelesenen binären Komponenten des *n*-Tupels, hier *n* = 3:

  $X_0 = (0, 0, 0)$ $\quad X_1 = (0, 0, 1)$ $\quad . . .$
  $. . .$ $\quad X_6 = (1, 1, 0)$ $\quad X_7 = (1, 1, 1)$

  Ein Belegungsblock *B* umfaßt 2^r Belegungen, wobei *r* die Anzahl der freien Komponenten x_i = - ist. Das Symbol (-) bezieht sich auf die Blockdarstellung der Eingangsbelegung; es entspricht beiden Werten 0 *und* 1:

  $B = (x_1, x_2, . . x_i . . , x_n);$ $\qquad x_i \in \{ 0, 1, - \}; i = 1 . . n$

- Eine Ausgangsbelegung ist ein *m*-Tupel mit *m* binären Komponenten y_j :

  $Y = (y_1, y_2, . . y_j . . , y_m);$ $\qquad y_j \in \{ 0, 1, * \}; j = 1 . . m$

  Für die Ausgangsbelegung gilt $Y \in A$, wobei *A* das Ausgabealphabet des Automaten ist. Das Symbol (∗) bezieht sich auf die Ausgangsbelegung; es gilt kein bestimmter Wert, d.h. er kann zu 0 *oder* 1 gesetzt werden.

- Ein Automatenzustand ist zunächst ein *symbolischer* Zustand:

  $Z \in S$

  wobei *S* die endliche Zustandsmenge des Automaten ist ("Finite State Machine", FSM). Der symbolische Zustand muß also kein binärer Tupel sein, sondern nur von allen anderen Zuständen desselben Automaten wohlunterschieden („diskret").

- Ein binär codierter Zustand ist ein *l*-Tupel mit *l* binären Komponenten q_k :

  $Q = (q_1, q_2, . . q_k . . , q_l);$ $\qquad q_k \in \{ 0, 1 \}; k = 1 . . l$ $\qquad | \underline{Q} | = 2^l \geq | \underline{S} |$

  Er implementiert einen symbolischen Zustand $Z \in S$. Die Zuordnung symbolischer Zustände zu binären *l*-Tupeln, d.h. die „Zustandscodierung" ist *frei* wählbar, wenn die Zustände des Automaten nach außen hin nicht in Erscheinung treten, wie es bei Automaten vom Moore- und vom Mealy-Typ der Fall ist. Eine bestimmter *codierter* Zustand wird mit Q_j bezeichnet; dabei ist der oktale Index *j* gleich den als Dualzahl gelesenen binären Komponenten des *l*-Tupels. Eine frei gewählte und damit gültige Zustandscodierung lautet zum Beispiel:

  $Z_0 = Q_2 = (1, 0)$ $\qquad Z_2 = Q_1 = (0, 1)$
  $Z_1 = Q_3 = (1, 1)$ $\qquad Z_3 = Q_0 = (0, 0)$

3.3.3 Endliche diskrete Automaten

Ein abstrakter Automat besteht aus einer Speicherfunktion, die den *Zustand* des Automaten enthält, sowie zwei Schaltfunktionen, die seine Folgezustände bzw. seine Ausgabe erzeugen. Letztere entsprechen seinem *Verhalten*. Seinen aktuellen Zustand - aus einer endlichen Anzahl wohlunterschiedener („diskreter") Zustände - hat er aufgrund eines Anfangszustandes und der gesamten Vorgeschichte aller bisher eingetroffenen Eingaben erreicht. Abstrakte Automaten erzeugen ihren Folgezustand mit einer

- Übergangsfunktion $Z^{t+1} = \delta(Z^t, X^t)$

Dabei ist Z^t der aktuelle Zustand zu einem diskreten Zeitpunkt t und Z^{t+1} der Folgezustand zum nächsten diskreten Zeitpunkt t+1, wobei die Taktperiode auf 1 normiert ist. Anhand der Ausgabefunktion λ lassen sich drei Automatentypen unterscheiden.

3.3.4 Medvedev-Automat

Beim einfachsten Automatentyp ist die Ausgabe Y gleich dem Automatenzustand Z; seine

- Ausgabefunktion $Y^t = Z^t$

ist trivial. Das folgende Bild zeigt links die *Struktur* des Medvedev-Automaten. Seine Ausgabefunktion λ ist eine Identität. Zur Beschreibung seines *Verhaltens* muß ein zeitlicher Ablauf in diskreten Schritten t, t+1, t+2 usf. dargestellt werden, wozu sich ein Graph bestens eignet:

- Man stellt die Zustände Z als Knoten dar und
- die Zustandsübergänge als gerichtete Kanten, die mit der Eingabe X gekennzeichnet werden.

Man erkennt am „Ablaufgraphen", rechts im Bild dargestellt, daß die Eingabe entscheidet (hier X_1 bzw. X_2), welcher Folgezustand (hier Z_1 oder Z_2) vom Zustandsknoten Z_0 aus erreicht wird, wie es die obige, für alle drei Automatentypen geltende Übergangsfunktion δ vorschreibt.

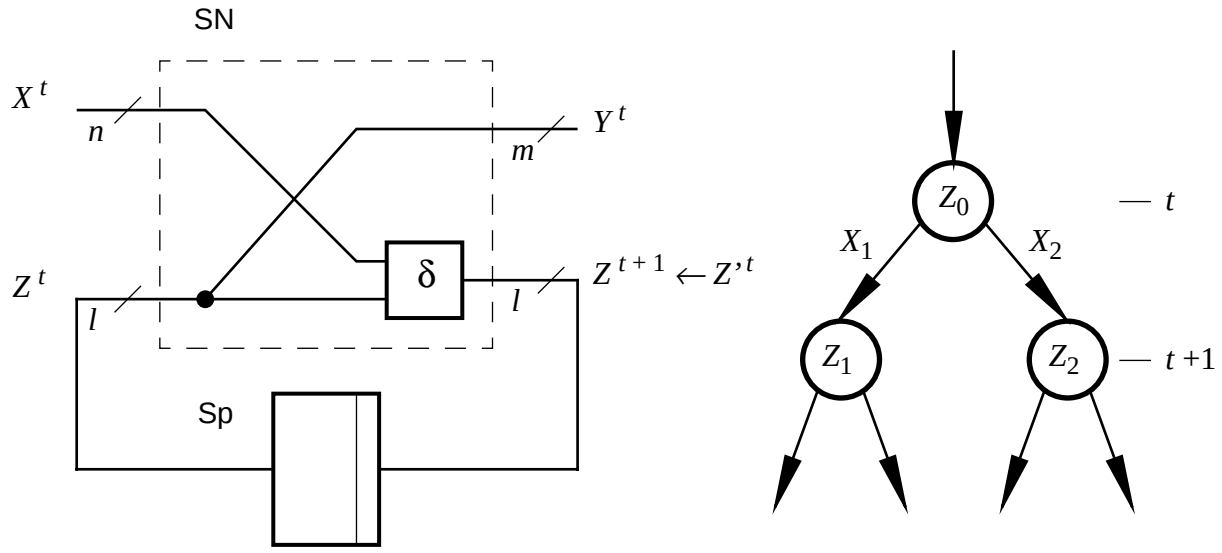


Bild 3.35: Medvedev-Automat; links: Struktur, SN - Schaltnetz, Sp - Speicherglied; rechts: Ablaufgraph. *X*: Eingabe, *Y*: Ausgabe, *Z*: Zustände, *t*: diskrete Zeit

3.3.5 Moore-Automat

Ist die Ausgabe *Y* eine Funktion des internen Automatenzustandes *Z*, so gilt die

- Ausgabefunktion $Y^t = \lambda(Z^t)$

Das folgende Bild zeigt links die Struktur des Moore-Automaten. Die Ausgabefunktion λ erzeugt die Ausgabe Y^t aufgrund des aktuellen Zustandes Z^t. Im Ablaufgraphen, rechts im Bild, wird das dadurch deutlich, daß man den Zustandsknoten *Z* die Ausgaben *Y* fest zuordnet. Die gerichteten Kanten des Graphen werden wieder mit den Eingaben *X* gekennzeichnet und entscheiden so über den Folgezustand.

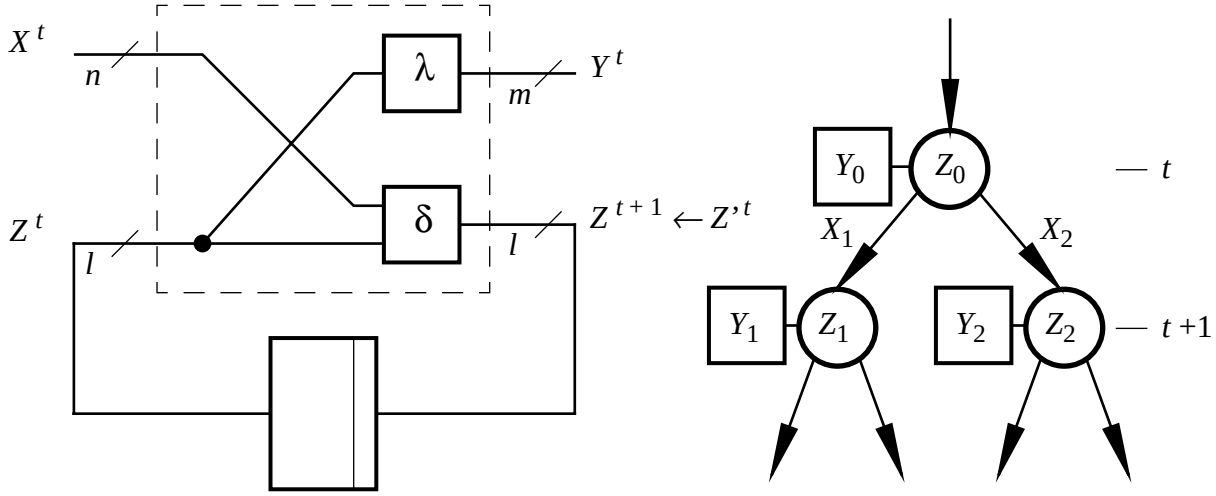


Bild 3.36: Moore-Automat; links: Struktur, rechts: Ablaufgraph

3.3.6 Mealy-Automat

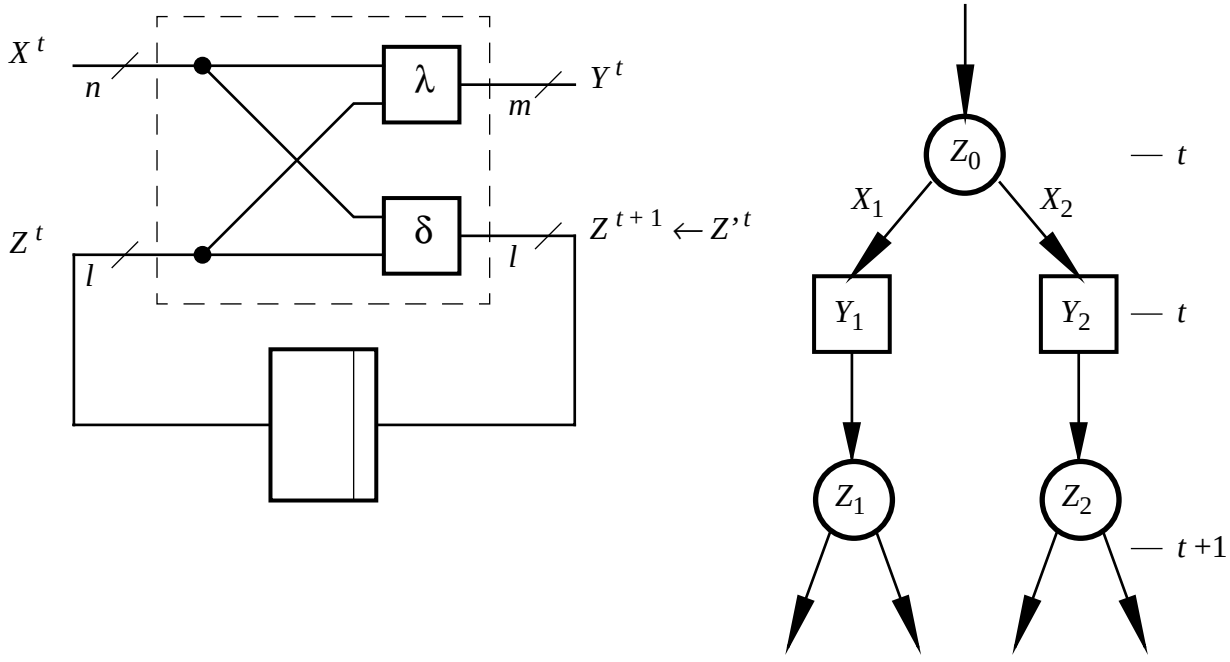


Bild 3.37: Mealy-Automat; links: Struktur, rechts: Ablaufgraph

Im kompliziertesten Fall ist die Ausgabe Y sowohl vom internen Automatenzustand Z, als auch von der Eingabe X abhängig; dann gilt die

- Ausgabefunktion $Y^t = \lambda(Z^t, X^t)$.

Das vorhergehende Bild zeigt links die Struktur des Mealy-Automaten. Die Ausgabefunktion λ erzeugt die Ausgabe Y^t sowohl aus dem aktuellen Zustand Z^t als auch anhand der Eingabe X^t. Im Ablaufgraphen im vorhergehenden Bild rechts wird das dadurch dargestellt, daß die Ausgaben Y in die Kanten eingefügt werden, da die Eingaben X, mit denen die Kanten gekennzeichnet sind, hier nicht nur - wie bei allen Automaten - über den Folgezustand, sondern auch über die Ausgabe entscheiden.

Zusammenfassend kann man sagen, daß der betreffende Automatentyp nicht nur an seinem Strukturbild, sondern auch an seinem Ablaufgraphen sofort zu erkennen ist.

3.3.7 Elementare Schaltwerke

Im Folgenden sollen einige einfache, aber häufig verwendete Schaltwerke vorgestellt werden, die beim Entwurf digitaler Schaltungen und Systeme unverzichtbar sind. Sie sind nur zweier Zustände fähig und werden deshalb lautmalerisch als „Flip-Flop" bezeichnet. Ihr aktueller Zustand muß von außen erkennbar sein, d.h. sie sind elementare Automaten vom Medvedev-Typ.

Bei den *asynchron* betriebenen Flipflops kommt die zeitliche Sequenz zwischen den Eingangs- und Ausgangsvariablen durch die Verzögerungszeiten Δt der internen Gatter zustande (linke Gleichung). Man kann die Betrachtungsweise vereinfachen, indem man letztere als verzögerungsfrei annimmt und nur zu regelmäßigen Zeitpunkten $t, t+1, t+2, \ldots$ neue Werte der Eingangsvariablen in das Flipflop übernimmt (rechte Gleichung):

$$Q \Rightarrow Q^t ; y \Rightarrow Q^{t+\Delta t} \qquad\qquad Q \Rightarrow Q^t ; y \Rightarrow Q^{t+1}$$

Für diesen *synchron* getakteten Betrieb unterscheidet man anhand des Takts c folgende Arten der Ansteuerung:

- Taktzustandsgesteuert: Der Wert 1 (oder der Wert 0) einer binären Taktvariablen $c \in \{0, 1\}$ aktiviert die Flipflop-Eingänge („Auffang-Flipflop").
- Taktflankengesteuert: Die ansteigende Vorderflanke (oder die abfallende Rückflanke) eines Taktsignals c aktiviert die Flipflop-Eingänge („dynamisches Flipflop").

Die Einführung eines neuen Schaltsymbols für Flipflops erscheint hier angebracht; denn die Angabe eines internen Gatterschaltbildes auf der Logikebene ist nicht immer möglich, eine Darstellung auf der Schaltkreisebene mit Transistoren hier zu detailliert.

a) SR-Flipflop

Wie beim asynchronen Beispiel erwähnt, hat ein SR-Flipflop drei Betriebsarten:

- Lesen/Speichern:
  Der aktuelle Wert der binären Zustandsvariablen wird beibehalten: $Q^t \rightarrow Q^{t+1}$
  Er soll als Ausgangsvariable erkennbar sein: $Q^t \rightarrow y^t$

- Setzen:
  Der Wert der Zustandsvariablen wird gesetzt: $1 \rightarrow Q^{t+1}$
- Rücksetzen:
  Der Wert der Zustandsvariablen wird rückgesetzt: $0 \rightarrow Q^{t+1}$

Diese Funktionsbeschreibung kann als Schaltfunktion in einer Wahrheitstabelle dargestellt werden, wobei sich die spezifizierten Betriebsarten an den vier Wertekombinationen (R, S) der Eingangsvariablen R und S orientieren, wie nachfolgend gezeigt. Aus der aktuellen Zustandsvariablen Q^t zum Zeitpunkt t ergibt sich nach einer normierten Verzögerung, d.h. zum Zeitpunkt t+1 die Folgezustandsvariable Q^{t+1}, die am Ausgang des Schaltwerks erscheint und rückgekoppelt wird. Die Eingangsbelegungen $X_j = (R, S, Q)$ sind hier teilweise in Blockschreibweise (-) zusammengefaßt. Das Bild zeigt auch das Schaltsymbol eines synchronen SR-Flipflop.

j	R^t	S^t	Q^t	Q^{t+1}	Betriebsart
0	0	0	0	0	Lesen/Speichern
1	0	0	1	1	Lesen/Speichern
2, 3	0	1	-	1	Setzen
4, 5	1	0	-	0	Rücksetzen
6, 7	1	1	0	*	(undefiniert)

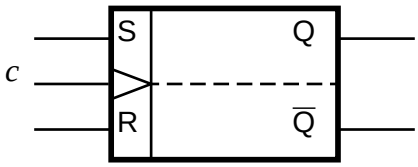


Bild 3.38: SR-Flipflop; Wahrheitstabelle und Schaltsymbol, vorderflankengetaktet

Die obige Wahrheitstabelle kann in eine „Ansteuertabelle" umgeformt werden, die sich an den vier möglichen Übergängen von Q^t nach Q^{t+1}orientiert, wie das folgende Bild zeigt. Für die Eingangsbelegungen $X_j = (R, S, Q)$ ergeben sich hier andere Blöcke als in der obigen Wahrheitstabelle, erkennbar am Symbol (-). Wie man der Ansteuertabelle entnehmen kann, verharrt das rückgekoppelte System für $R \vee S = 0$ in einem stabilen Zustand $Q^t = Q^{t+1}$. Da die Schaltung dann ihren Zustand beibehält, handelt es sich um eine *Speicherschaltung*. Man kann diesen Sachverhalt auch durch einen Ablaufgraphen veranschaulichen:

- Die beiden Zustände $Q \in \{0, 1\}$ werden durch zwei Knoten dargestellt,
- die Zustandsübergänge durch gerichtete Kanten, die mit den Eingangsvariablen R und S in Blockschreibweise (-) gekennzeichnet sind.

j	R^t	S^t	Q^t	Q^{t+1}
0, 4	-	0	0	0
2	0	1	0	1
5	1	0	1	0
1, 3	0	-	1	1

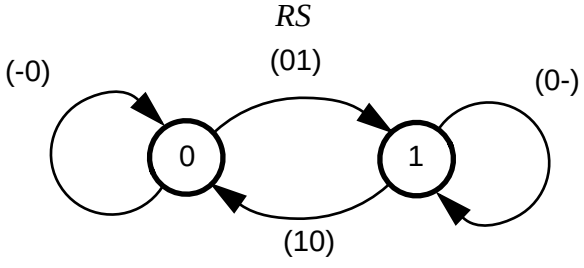


Bild 3.39: SR-Flipflop; Ansteuertabelle und Ablaufgraph

b) D-Flipflop

Ist man nur an der zeitlichen Verzögerung („*Delay*") einer Eingangsvariablen durch ein SR-Flipflop interessiert, d.h. sein aktueller Zustand Q^t ist unerheblich, so müssen die Eingangsvariablen R und S entsprechend der SR-Wahrheitstabelle im vorherigen Abschnitt *ungleich* sein. Mit einer neu benannten Eingangsvariablen D gilt dann:

$D = S = \overline{R}$ $\qquad\qquad$ $D^t \rightarrow Q^{t+1}$

Entsprechend geschaltet wirkt ein SR-Flipflop als sogenanntes *D-Flipflop*. Aus der SR-Wahrheitstabelle übernimmt man nur die Zeilen, für die die Eingangsvariablen R und S ungleich sind. Nach Umformen erhält man die Ansteuertabelle des D-Flipflop, die sich anschaulich als Ablaufgraph darstellen läßt.

j	D^t	Q^t	Q^{t+1}	Betriebsart
2, 3	1	-	1	Setzen
4, 5	0	-	0	Rücksetzen

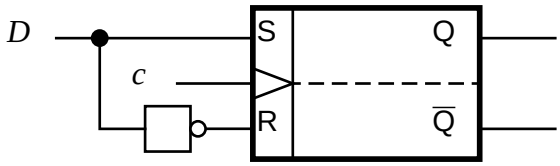


Bild 3.40: D-Flipflop; Wahrheitstabelle und Schaltung, vorderflankengetaktet

j	$\overline{R}^t = D^t = S^t$	Q^{t+1}
4, 5	0	0
2, 3	1	1

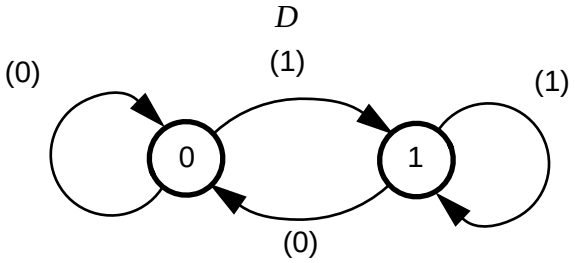


Bild 3.41: D-Flipflop; Ansteuertabelle und Ablaufgraph

c) JK-Flipflop

Wie beim asynchronen Beispiel erwähnt, hat ein JK-Flipflop eine vierte Betriebsart:

- Triggern:
  Der aktuelle Wert der binären Zustandsvariablen wird invertiert: $\overline{Q}^t \rightarrow Q^{t+1}$

j	K^t	J^t	Q^t	Q^{t+1}	Betriebsart
0	0	0	0	0	Lesen/Speichern
1	0	0	1	1	Lesen/Speichern
2, 3	0	1	-	1	Setzen
4, 5	1	0	-	0	Rücksetzen
6	1	1	0	1	Triggern
7	1	1	1	0	Triggern

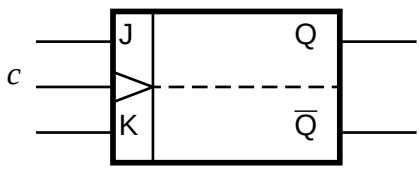


Bild 3.42: JK-Flipflop; Wahrheitstabelle und Schaltsymbol, vorderflankengetaktet

Die Erweiterung der Funktionsbeschreibung zeigt die obige Wahrheitstabelle; das Schaltsymbol eines synchronen JK-Flipflop ist ebenfalls angegeben. Das nächste Bild zeigt die Ansteuertabelle und den Ablaufgraphen des JK-Flipflops, wobei die Zustandsübergänge mit den Eingangsvariablen *K* und *J* in Blockschreibweise (-) gekennzeichnet sind.

j	K^t	J^t	Q^t	Q^{t+1}
0, 4	-	0	0	0
2, 6	-	1	0	1
5, 7	1	-	1	0
1, 3	0	-	1	1

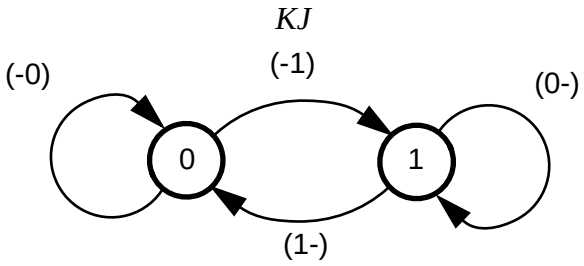


Bild 3.43: JK-Flipflop; Ansteuertabelle und Ablaufgraph

d) T-Flipflop

Ist man nur an der Invertierung („*Triggern*") des internen Zustands eines JK-Flipflop interessiert, d.h. daß sein Folgezustand die Negation des aktuellen Zustands sein soll, so müssen die Eingangsvariablen *K* und *J* entsprechend der JK-Wahrheitstabelle im vorherigen Abschnitt *gleich* sein. Mit einer neu benannten Eingangsvariablen *T* gilt dann:

$T = J = K \qquad \overline{Q}^t \rightarrow Q^{t+1}$

Entsprechend geschaltet wirkt ein JK-Flipflop als sogenanntes *T-Flipflop*. Aus der JK-Wahrheitstabelle übernimmt man nur die Zeilen, für die die Eingangsvariablen *K* und *J* gleich sind. Nach Umformen erhält man die Ansteuertabelle des T-Flipflop, die sich anschaulich als Ablaufgraph darstellen läßt.

j	T^t	Q^t	Q^{t+1}	Betriebsart
0	0	0	0	Lesen/Speichern
1	0	1	1	Lesen/Speichern
6	1	0	1	Triggern
7	1	1	0	Triggern

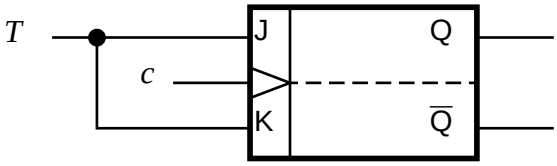


Bild 3.44: T-Flipflop; Wahrheitstabelle und Schaltung, vorderflankengetaktet

j	$K^t = T^t = J^t$	Q^{t+1}
0, 1	0	Q^t
6, 7	1	$\overline{Q}^t$

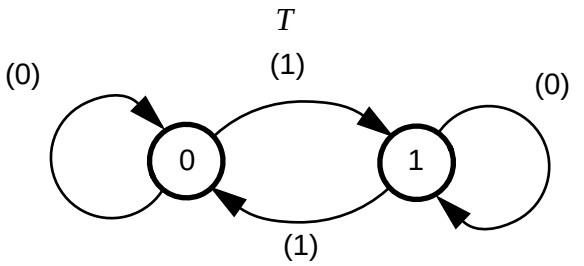


Bild 3.45: T-Flipflop; Ansteuertabelle und Ablaufgraph

3.3.8 Der Entwurf von Schaltwerken

Schaltwerke implementieren endliche diskrete Automaten (”Finite State Machines”, FSM). Angesichts ihrer Komplexität sollten sie in mehreren klar definierten Schritten, die aufeinander aufbauen, systematisch entworfen werden:

- Informale Spezifikation der Gesamtfunktion des Automaten in natürlicher Sprache.
- Umsetzung in eine formale Spezifikation, z.B. einen Ablaufgraphen oder eine Ablauftabelle.
- Codierung der diskreten Zustände des Automaten.
- Entwurf des Schaltnetzes zur Implementierung der Ausgabefunktion.
- Wahl der Speicherglieder im Rückkopplungspfad.
- Entwurf des Schaltnetzes zur Implementierung der Übergangsfunktion.
- Darstellung der Schaltwerksstruktur mit logischen Gattern und Speichergliedern.

Beispiel 3.3.1: Automat vom Moore-Typ

Z^t	X^t	Z^{t+1}	Y^t
	$x_1\, x_0$		$y_2\, y_1\, y_0$
0	0 0 0 1 1 -	1 0 2	1 0 *
1	0 0 1 0 - 1	1 3 2	* 0 1
2	0 0 1 0 - 1	0 3 0	0 * 1
3	- -	1	1 1 0

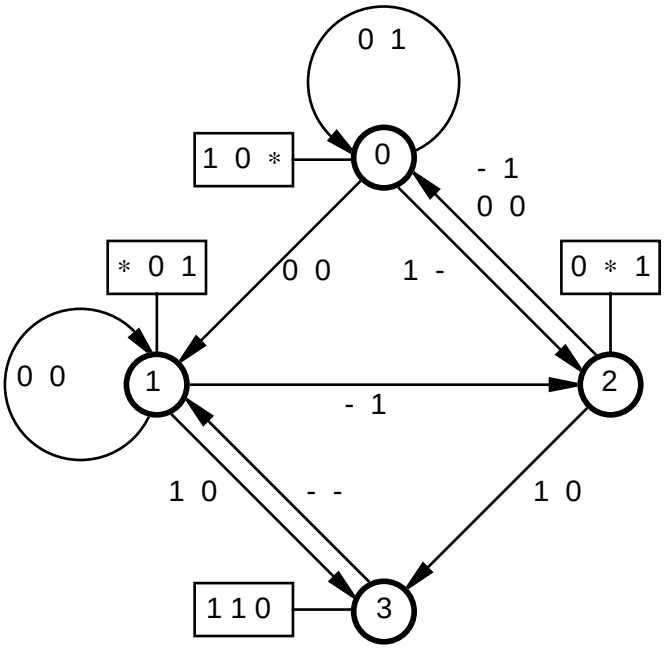


Bild 3.46: Spezifikation eines durch ein Schaltwerk zu implementierenden *Moore*-Automaten; links: Symbolische Ablauftabelle, rechts: Ablaufgraph.

a) Spezifikation

Soll ein digitales Schaltwerk implementiert werden, so ist zuerst die gewünschte Gesamtfunktion als verbale Aufgabenstellung in natürlicher Sprache zu beschreiben. Um einen systematischen Entwurf zu ermöglichen, muß diese „informale Spezifikation” durch die „formale Spezifikation” eines abstrakten Automaten geeigneten Typs erfaßt werden, z.B. durch dessen Ablauftabelle oder - anschaulicher - durch seinen Ablaufgraphen. Der zu implementierende Automat sei durch die

obige Ablauftabelle formal spezifiziert. Man erkennt sofort, daß es sich um einen Moore-Automaten handelt, da die Ausgabe Y^t nur vom aktuellen Zustand Z^t abhängt. Der Automat besitzt vier diskrete Zustände Z, die symbolisch mit den Ziffern 0 bis 3 durchnumeriert sind. Dieser symbolischen Ablauftabelle entspricht der Ablaufgraph im vorhergehenden Bild.

b) Zustandscodierung

Da ein Schaltwerk keine symbolischen Automatenzustände Z implementieren kann, sind diese in einem ersten Entwurfsschritt durch binäre Zustandsvariablen $Q = (q_n, \ldots, q_1, q_0)$ zu codieren. Da bei beiden üblicherweise zugrundegelegten Automaten vom Moore- bzw. Mealy-Typ der Automatenzustand von außen nicht erkennbar ist, ist man bei der Wahl der Codierung völlig frei, wobei auch die Zuordnung der einzelnen Codewörter zu den Symbolen frei gewählt werden kann. Nachfolgend einige gängige Beispiele:

- Relativ aufwendig ist der 1-aus-*n* Code, wobei *n* die Anzahl der zu codierenden Zustände ist.
- Kompakter ist ein Binärcode mit ld *n* Bits, um bis zu *n* Zustände zu codieren.

Um den Einfluß des gewählten Zustandscodes auf die Struktur des zu entwerfenden Schaltwerks und damit auf den Schaltungsaufwand an einem Beispiel aufzuzeigen, sollen zwei verschiedene Zustandscodierungen untersucht werden:

- Eine „naive" Zustandscodierung, bei der die zufällig gewählten Zustandssymbole 0 bis 3 als Dezimalzahlen aufgefaßt und in Dualzahlen umgewandelt werden, die als Codewörter dienen („Dualcode"), wie in der codierten Ablauftabelle im folgenden Bild links gezeigt.
- Eine Zustandscodierung, bei der sich beim Zählen jeweils nur 1 Bit ändert („Gray-Code"), wie in der codierten Ablauftabelle im folgenden Bild rechts gezeigt.

Z^t	Q^t	Y^t
	$q_1\ q_0$	$y_2\ y_1\ y_0$
0	0 0	1 0 *
1	0 1	* 0 1
2	1 0	0 * 1
3	1 1	1 1 0

Z^t	Q^t	Y^t
	$q_1\ q_0$	$y_2\ y_1\ y_0$
0	0 0	1 0 *
1	0 1	* 0 1
2	1 1	0 * 1
3	1 0	1 1 0

Bild 3.47: Codierte Ablauftabellen des spezifizierten Moore-Automaten;
Zustandscodierung: links im *Dualcode*, rechts im *Gray-Code*.

Die Ablauftabellen im vorhergehenden Bild werden später noch ergänzt. Für einen Moore-Automaten genügen sie zunächst, da bei diesem Typ die Ausgabe nur vom aktuellen Zustand abhängt: $Y^t = \lambda(Z^t)$.

c) Entwurf des Ausgabeschaltnetzes

Die Ausgangsvariablen $Y = (y_2, y_1, y_0)$ sollen durch eine Ausgabefunktion λ erzeugt werden, für die ein Schaltnetz zu entwerfen ist. Zur Minimierung werden die Werte der genannten Ausgangsvariablen aus den codierten Ablauftabellen des vorhergehenden Bildes entnommen und in KV-Diagramme eingetragen, wie es im folgenden Bild für den Dualcode, im übernächsten für den Gray-Code geschehen ist. Um die Unterschiede der beiden Zustandscodierungen und ihren Einfluß auf die Anordnung der Variablenwerte $(y_2, y_1, y_0) \in \{0, 1, *\}$ im KV-Diagramm zu verdeutlichen, wurde der betreffende Codierraum für die Zustände Z jeweils mit dargestellt.

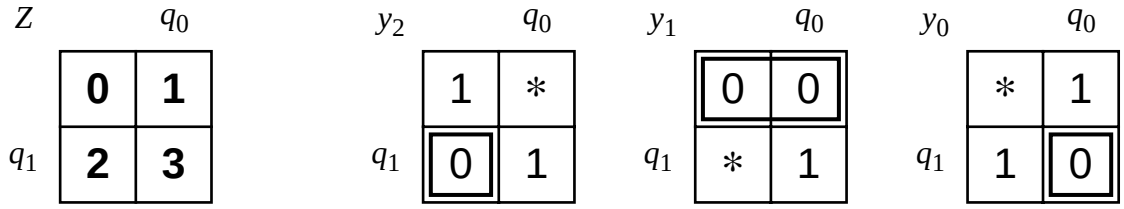


Bild 3.48: KV-Diagramme für die im *Dualcode* codierte Ablauftabelle nach Bild 3.16 links; Z: Codierraum; y_2, y_1, y_0: Ausgangsvariablen.

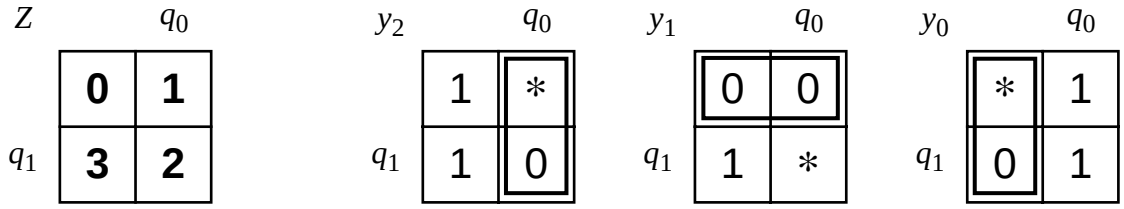


Bild 3.49: KV-Diagramme für die im *Gray-Code* codierte Ablauftabelle nach Bild 3.16 rechts; Z: Codierraum; y_2, y_1, y_0: Ausgangsvariablen.

Für dieses Beispiel bildet man in den KV-Diagrammen Überdeckungen der Nullstellen unter Einbeziehung von Freistellen (*), die zusätzliche Freiheitsgrade für den Entwurf bieten. Für den Dualcode findet man im obigen Bild folgende, teils triviale Primterme in konjunktiver Minimalform:

KMF $y_2 = \overline{q}_1 \vee q_0 \; ; \; y_1 = q_1 \; ; \; y_0 = \overline{q}_1 \vee \overline{q}_0$

Für den Gray-Code findet man im vorhergehenden Bild unter Einbeziehung von Freistellen (*) ausschließlich triviale, d.h. absolut aufwandsminimale Primterme in konjunktiver Minimalform:

KMF $y_2 = \overline{q}_0 \; ; \; y_1 = q_1 \; ; \; y_0 = q_0$

Die zur KMF jeweils isomorphen Schaltnetze werden im folgenden Bild einander gegenübergestellt.

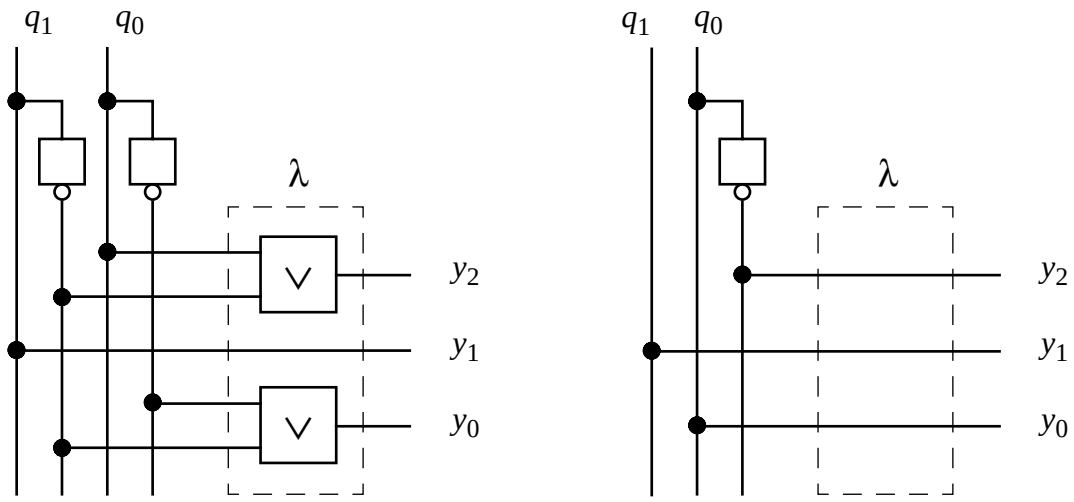


Bild 3.50: Implementierung der Ausgabefunktion $Y = (y_2, y_1, y_0) = \lambda(q_1, q_0)$;
Zustandscodierung: links im *Dualcode*, rechts im *Gray-Code*.

d) Wahl der Speicherglieder

Z^t	Q^t	X^t		Z^{t+1}	Q^{t+1}		Y^t
	$q_1\ q_0$	$x_1 x_0$	j		$D_1 D_0$	$T_1 T_0$	$y_2\, y_1 y_0$
0	0 0 0 0 0 0	0 0 0 1 1 -	00 01 02 03	1 0 2	0 1 0 0 1 1	0 1 0 0 1 1	1 0 *
1	0 1 0 1 0 1	0 0 1 0 - 1	04 06 05 07	1 3 2	0 1 1 0 1 1	0 0 1 1 1 0	* 0 1
2	1 1 1 1 1 1	0 0 1 0 - 1	14 16 15 17	0 3 0	0 0 1 0 0 0	1 1 0 1 1 1	0 * 1
3	1 0	- -	10 11 12 13	1	0 1	1 1	1 1 0

Bild 3.51: Codierte Ablauftabelle des hier spezifizierten Moore-Automaten;
Zustandscodierung im *Gray-Code*, Zustandsspeicherung in *T-Flipflops*.

Zur Speicherung der vier Automatenzustände des hier spezifizierten Moore-Automaten benötigt man bei beiden gewählten Codierungen mindestens zwei Flipflops. Als Speicherglieder stehen grundsätzlich alle vier bereits vorgestellten Flipfloptypen zur Verfügung:

- Setz-Rücksetz-Flipflops, Delay-Flipflops, JK-Flipflops oder Trigger-Flipflops.

Als Beispiel seien willkürlich *T-Flipflops* gewählt; ferner soll die Zustandscodierung im *Gray-Code* weiterverwendet werden. Das vorhergehende Bild zeigt die codierte Ablauftabelle für diese Zustandscodierung und den gewählten Flipflop-Typ. Zur Ermittlung der Ansteuervariablen T_1 bzw. T_0 sind in der codierten Ablauftabelle die Spalten für den aktuellen und den Folgezustand zeilenweise miteinander zu vergleichen: Jeder Zustandsübergang von Q^t nach Q^{t+1} ist bitweise, d.h. für gleiche Indizes zu prüfen. Falls sich das Zustandsbit q_i ändert, so gilt $T_i = 1$, sonst $T_i = 0$, wie der Ansteuertabelle und/oder dem Ablaufgraphen des T-Flipflops in Bild 3.14 zu entnehmen ist.

e) Entwurf des Übergangsschaltnetzes

Die Ansteuervariablen T_1 und T_0 der T-Flipflops sollen durch eine Übergangsfunktion δ erzeugt werden, für die ein Schaltnetz zu entwerfen ist. Die Funktionsbeschreibung der genannten Ansteuervariablen kann aus den entsprechenden Spalten der codierten Ablauftabelle entnommen und, wie nachfolgend gezeigt, zur Minimierung in KV-Diagrammen dargestellt werden. In den KV-Diagrammen bildet man die jeweils aufwandsminimalen Überdeckungen mit Primblöcken, hier der Nullstellen. Daraus erhält man die folgenden Primterme in konjunktiver Minimalform:

KMF $T_1 = (\bar{q}_1 \vee \bar{q}_0 \vee \bar{x}_1 \vee x_0) \,\&\, (q_1 \vee q_0 \vee x_1) \,\&\, (q_1 \vee x_1 \vee x_0)$

$T_0 = (q_1 \vee x_1 \vee \bar{x}_0) \,\&\, (q_1 \vee \bar{q}_0 \vee \bar{x}_0) \,\&\, (q_1 \vee \bar{q}_0 \vee x_1)$

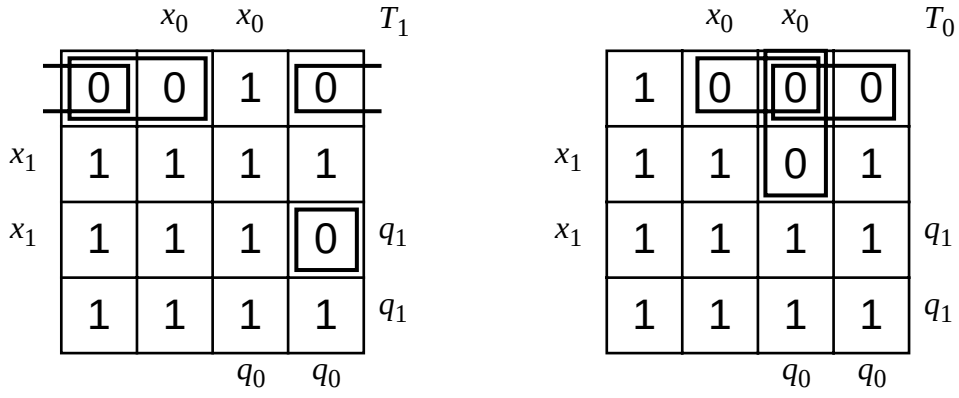


Bild 3.52: KV-Diagramme für die codierte Ablauftabelle nach Bild 3.20;
T_1, T_0: Ansteuervariablen der *T-Flipflops*.

f) Schaltwerksstruktur

Isomorph zu den ermittelten Strukturausdrücken (hier: KMF) ergeben sich für das Schaltwerk die im folgenden Bild dargestellten Gatterstrukturen des Ausgabeschaltnetzes λ und des Übergangsschaltnetzes δ, zusammen mit den (hier willkürlich) gewählten T-Flipflops.

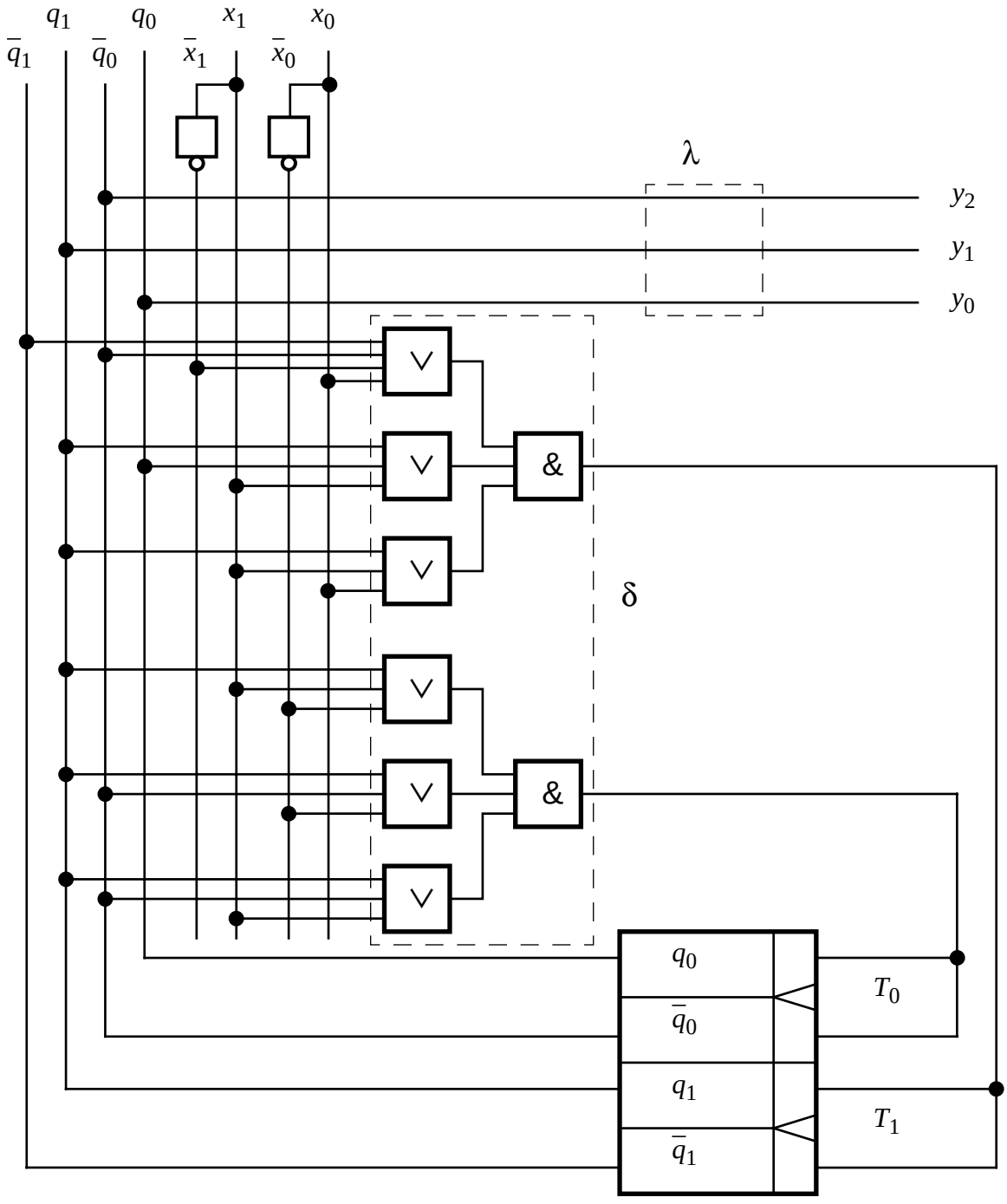


Bild 3.53: Schaltwerk zur Implementierung des hier spezifizierten *Moore*-Automaten; Zustandscodierung im *Gray-Code*, Zustandsspeicherung in *T-Flipflops*.

3.3.9 Kostenabschätzung

Zur Abschätzung des Implementierungsaufwandes mag eine vereinfachte Kostenbetrachtung genügen. Beispielsweise benötigt man in statischer Transistorschaltungstechnik für jede Eingangsvariable eines logischen Gatters einen Transistor und für jedes Gatter zusätzlich einen Arbeitswiderstand, wie das folgende Bild links zeigt. Nimmt man vereinfachend an, daß die Kosten für jedes Bauelement gleich seien, und normiert man auf diese Bauelementkosten, so erhält man für die

normierten Kosten der folgenden zweistufigen Schaltnetzstruktur (rechts), wobei im Trivialfall eines einzelnen Eingangsgatters (n=1) das Ausgangsgatter entfällt:

$$K(n>1) = 1 + 2n + (e_1 + e_2 + . . + e_n); \qquad K(n=1) = 1 + e_1$$

wobei n : Gatteranzahl in der Eingangsstufe, e_i : Anzahl der Primäreingänge.

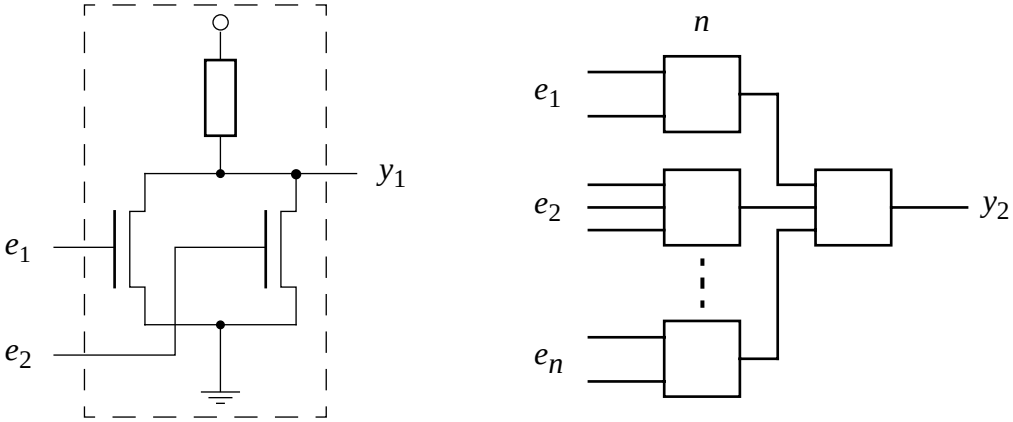


Bild 3.54: links: Realisierung eines logischen Gatters auf der Schaltungsebene;
rechts: Kenngrößen für Schaltnetzkosten; n: Gatteranzahl in der Eingangsstufe, e_i: Anzahl der Primäreingänge

Mit den oben ermittelten Strukturausdrücken in konjunktiver Minimalform (KMF) für die Ausgangsvariablen (y_2, y_1, y_0) sowie die Ansteuervariablen (T_1, T_0) der Flipflops erhält man die nachfolgenden normierten Schaltnetzkosten, wobei zur Ermittlung der Gesamtkosten des Schaltwerks die Kosten der zwei Flipflops zusätzlich zu berücksichtigen sind.

- Gray-Code, T-Flipflops:

$K(y_2) = 0$	$K(T_1) = 1 + 2 \times 3 + (4 + 3 + 3) = 17;$
$K(y_1) = 0$	$K(T_0) = 1 + 2 \times 3 + (3 + 3 + 3) = 16;$
$K(y_0) = 0$	$K(\delta) = K(T_1) + K(T_0) = 33;$
$K(\lambda) = K(y_2) + K(y_1) + K(y_0) = 0$	$K(SN) = K(\delta) + K(\lambda) = 33 + 0 =$ **33**

3.4 Programmierbare Digitalbausteine („PLD“)

3.4.1 Technologie und Schaltungstechnik

a) Technologische Randbedingungen

Die *Mikroelektronik* ermöglicht die Integration umfangreicher und komplexer elektronischer Schaltungen auf einem einzigen Halbleiterchip. Insbesondere die Siliziumtechnologie erlaubt es, dank der Eigenschaften des Siliziumdioxids, kleinste Strukturen im Mikrometerbereich herzustellen. Man ist bestrebt, ein mikroelektronisches System von gegebenem Umfang auf möglichst kleiner Chipfläche unterzubringen; denn je größer das Chip, desto größer die Gefahr, dass es einen Kristalldefekt enthält, und desto geringer ist die Ausbeute bei seiner Fertigung. Die Chipfläche wird vor allem durch die Verdrahtung auf dem Chip bestimmt, da die Gatter im wesentlichen unter der Verdrahtung im Kristallinnern liegen. Es gilt daher beim Chipentwurf, die Gesamtlänge der Verdrahtung zu minimieren, und zwar aus zwei Gründen: Nicht nur die Chipfläche, sondern auch die Laufzeiten auf dem Chip werden dadurch verringert.

Die Weiterentwicklung der Mikroelektronik wurde stets von der Technologie der Halbleiterspeicher angeführt, da Speicher *regelmäßig strukturiert* sind: Die Speicherzellen sind matrixförmig angeordnet, Speicherchips weisen ein regelmäßiges Verdrahtungsmuster auf. Die Dichte der Bauelemente ist auf einem Speicherchip in der Regel höher als auf einem Mikroprozessor, da dessen Strukturen komplexer und damit unregelmäßiger sind. Regelmäßige Strukturen sind übersichtlicher beim Entwurf, mit höherer Ausbeute zu fertigen und von längerer Lebensdauer im Betrieb. Man sucht daher die Fortschritte der Speichertechnologie auch für Prozessor- und Logikschaltungen zu nutzen, indem man sie z.B. matrixförmig und damit möglichst regelmäßig strukturiert.

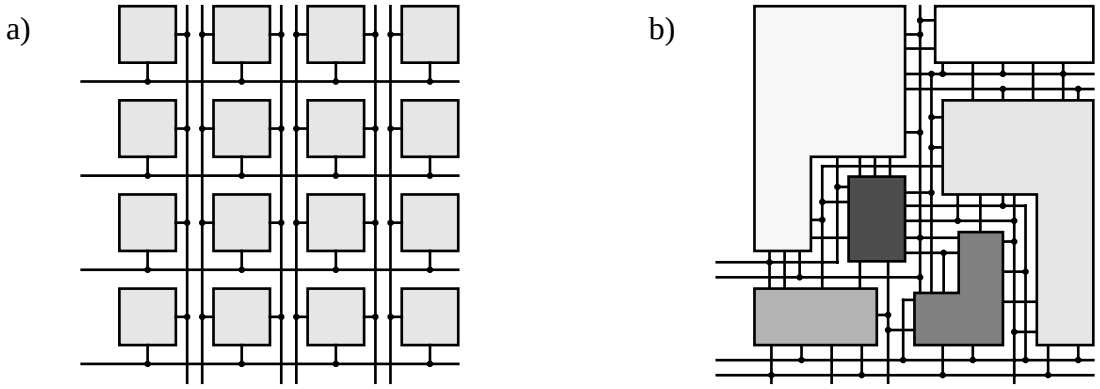


Bild 3.55: Grundstrukturen der Mikroelektronik; a) Speichermatrix; b) Mikroprozessor.

Einerseits setzt die Mikroelektronik hohe Investitionen an Gebäuden und Geräten voraus, so dass nur die automatisierte Herstellung großer Stückzahlen wirtschaftlich ist. Andererseits besteht ein großer Bedarf an anwendungsspezifischen Schaltungen, die unter Umständen nur in niedrigen Stückzahlen benötigt werden. Hier empfehlen sich *vordefinierte Strukturen*, die je nach Anwendung *personalisierbar* sind. Auch hier waren die Speicher die Vorreiter der Entwicklung: Alle Schaltfunktionen können durch Programmieren einer vorgefertigten Speichermatrix implementiert werden. Programmierbare Strukturen haben zudem den Vorteil der Flexibilität und Änderbarkeit, um Entwurfsfehler zu korrigieren und/oder neue Funktionen mit einzubeziehen.

b) Schaltkreistechnologien

Elektronische Schaltungen und Systeme können mit zwei prinzipiell unterschiedlichen Schaltkreistechnologien realisiert werden:

- Die bipolare Halbleitertechnik erlaubt die Herstellung von Transistoren mit NPN- oder PNP-Schichtstruktur sowie von PN-Dioden. Erstere sind aktive, letztere passive Bauelemente. Schaltungen mit bipolaren Bauelementen werden in der Regel statisch betrieben.
- Die unipolare Halbleitertechnik erlaubt die Herstellung von MOS-Transistoren mit N-Kanal oder P-Kanal. Beides sind aktive Bauelemente. Schaltungen mit MOS-Bauelementen können statisch oder, dank inhärenter MOS-Kapazitäten, auch dynamisch betrieben werden.

Aus der Fülle der Varianten, die bisher auf dem Gebiet der Schaltkreistechnologien entwickelt wurden, sollen hier nur zwei einfache Vertreter vorgestellt werden. Denn es geht hier nicht um schaltungstechnische Einzelheiten, sondern um strukturell-topologische Zusammenhänge zwischen Logik- und Schaltungsebene. Auch die schaltungstechnisch hochinteressante CMOS-Technologie - es sei nur an die dynamische Domino-Logik erinnert - ginge in diesem Zusammenhang über die Zielsetzung des vorliegenden Abschnitts hinaus.

Bei der Realisierung logischer Gatter mit elektronischen Bauelementen muß zuerst die Zuordnung der binären Variablenwerte (0, 1) zu zwei elektrischen Pegeln definiert werden, die sich durch einen entsprechenden Störabstand deutlich voneinander unterscheiden. Trifft man für die Eingangs- und die Ausgangsvariablen dieselbe Zuordnung, so gibt es zwei Möglichkeiten:

- Positive Logik: Der höhere Signalpegel wird dem Wert 1, der niedrigere dem Wert 0 zugeordnet.
- Negative Logik: Der höhere Signalpegel wird dem Wert 0, der niedrigere dem Wert 1 zugeordnet.

Die folgenden Ausführungen setzen die *positive* Logik stillschweigend voraus. Als elementare Schaltungsbeispiele zeigt Bild 10.2 a) ein UND-Gatter mit bipolaren Dioden, b) ein ODER-Gatter in derselben Schaltungstechnik, c) ein NOR-Gatter mit NMOS-Transistoren, dazu die entsprechenden logischen Gattersymbole. Die Schaltalgebra lehrt, insbesondere ihr Hauptsatz, daß sich alle Schaltfunktionen und damit beliebige Schaltnetze allein mit diesen Grundgattern implementieren lassen.

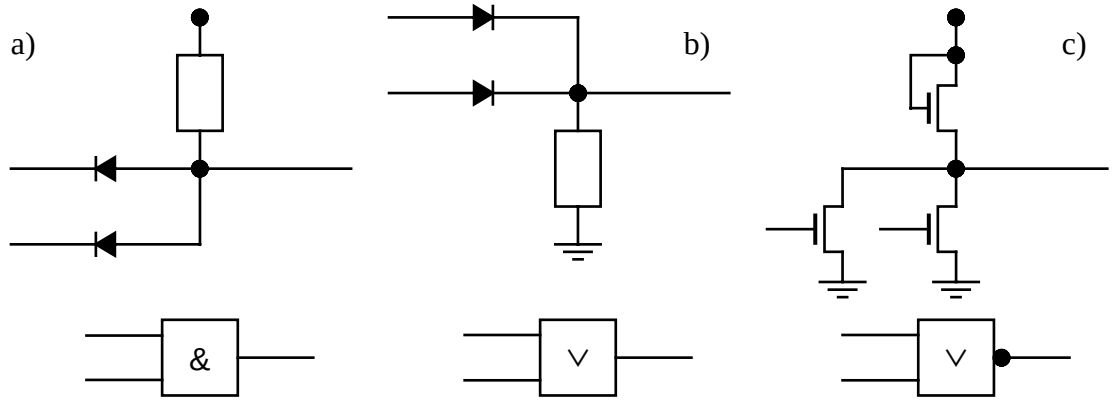


Bild 3.56: Schaltungstechnik; a) UND-Gatter mit Dioden, b) ODER-Gatter mit Dioden, c) NOR-Gatter mit NMOS-Transistoren.

Beispiel 3.12:

Gegeben sei folgender Strukturausdruck einer Schaltfunktion in disjunktiver Normalform:

DNF $y = (\bar{x}_2 \,\&\, x_1 \,\&\, \bar{x}_0) \vee (x_2 \,\&\, \bar{x}_1 \,\&\, x_0)$

Das isomorphe zweistufige Schaltnetz zeigt das folgende Bild. Es besteht aus zwei UND-Gattern mit je drei Eingängen zur Implementierung der beiden Konjunktionen (&) und einem ODER-Gatter mit zwei Eingängen zur Implementierung der Disjunktion ($\vee$).

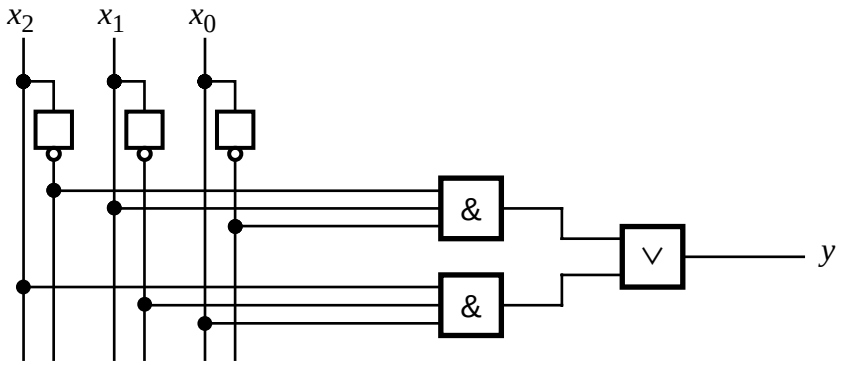


Bild 3.57: Zweistufiges Schaltnetz in disjunktiver Normalform (DNF)

Wendet man die DeMorgansche Regel auf die beiden Teilterme der obigen DNF an, so erhält man:

$$y = \overline{\overline{(\bar{x}_2 \,\&\, x_1 \,\&\, \bar{x}_0)}} \vee \overline{\overline{(x_2 \,\&\, \bar{x}_1 \,\&\, x_0)}} = \overline{(x_2 \vee \bar{x}_1 \vee x_0)} \vee \overline{(\bar{x}_2 \vee x_1 \vee \bar{x}_0)}$$

$$\bar{y} = (x_2 \,\overline{\vee}\, \bar{x}_1 \,\overline{\vee}\, x_0) \,\overline{\vee}\, (\bar{x}_2 \,\overline{\vee}\, x_1 \,\overline{\vee}\, \bar{x}_0)$$

Damit läßt sich der gegebene Strukturausdruck auch durch ein zweistufiges Schaltnetz implementieren, das ausschließlich aus NOR-Gattern ($\overline{\vee}$) besteht. Der verneinte Funktionswert $\bar{y}$ ist dabei durch einen einfachen Ausgangsinverter in den bejahten Funktionswert y umzuwandeln.

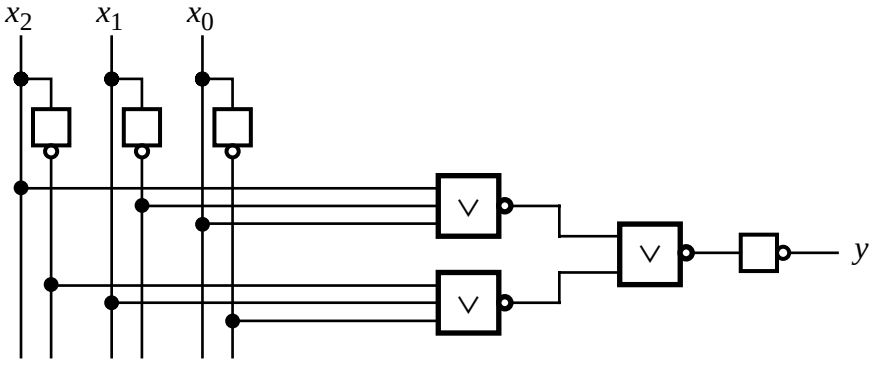


Bild 3.58: Zweistufiges NOR-Schaltnetz aus der DNF

c) Bipolare Schaltungstechnik

Die disjunktive Normalform (DNF) des obigen Beispiels (Bild 10.3) läßt sich in bipolarer Schaltungstechnik unter Verwendung der in Bild 10.2 a) und b) gezeigten Diodengatter direkt realisieren, wie Bild 10.5 zeigt. Die Schaltung besteht, soweit es die logischen Verknüpfungen betrifft, aus passiven Bauelementen, und sie wird statisch betrieben.

Interessant ist, daß sich die Diodenschaltung topologisch so umgestalten läßt, daß zur Realisierung der logischen Verknüpfungen zwei Matrizen entstehen (Bild 10.6):

- für die Konjunktionen (&) eine UND-Matrix, d.h. die Zeilen bilden logische Produkte P_i,
- für die Disjunktion ($\vee$) eine ODER-Matrix, hier mit einer Spalte für den Funktionswert y.

Selbst dieses einfache Schaltbild zeigt, daß anwendungsspezifische Schaltfunktionen dadurch realisiert werden können, daß die Dioden in der UND-Matrix entsprechend der Spezifikation wahlweise an die bejahten oder die verneinten Eingangsvariablen x_i angeschlossen werden. Durch diese Maßnahme wird die vorstrukturierte Schaltung *personalisiert*.

d) MOS-Schaltungstechnik

Wird die disjunktive Normalform (DNF) desselben Beispiels wie oben gezeigt in eine zweistufige NOR-NOR-Form umgewandelt, so läßt sich die gegebene Schaltfunktion auch in MOS-Schaltungstechnik unter ausschließlicher Verwendung des in Bild 10.2 c) gezeigten NOR-Gatters realisieren. Die Schaltung in Bild 10.7 besteht aus aktiven, d.h. verstärkenden NMOS-Transistoren; sie wird hier statisch betrieben.

Auch die NMOS-Transistorschaltung läßt sich topologisch so umgestalten, daß zur Realisierung der logischen Verknüpfungen zwei Matrizen entstehen (Bild 10.8):

- für die Konjunktionen (&) eine UND-Matrix
- und für die Disjunktion ($\vee$) eine ODER-Matrix.

Die Namen der beiden Matrizen sind unabhängig davon, ob die logischen Verknüpfungen schaltungstechnisch durch eine echte UND-ODER-Schaltung oder, wie zumeist, durch eine zweistufige NOR-NOR-Schaltung realisiert werden.

Auch hier erfolgt die Personalisierung der Schaltung je nach anwendungsspezifischer Spezifikation dadurch, daß die Gatter der NMOS-Transistoren in der UND-Matrix wahlweise an die bejahten oder die verneinten Eingangsvariablen x_i angeschlossen werden.

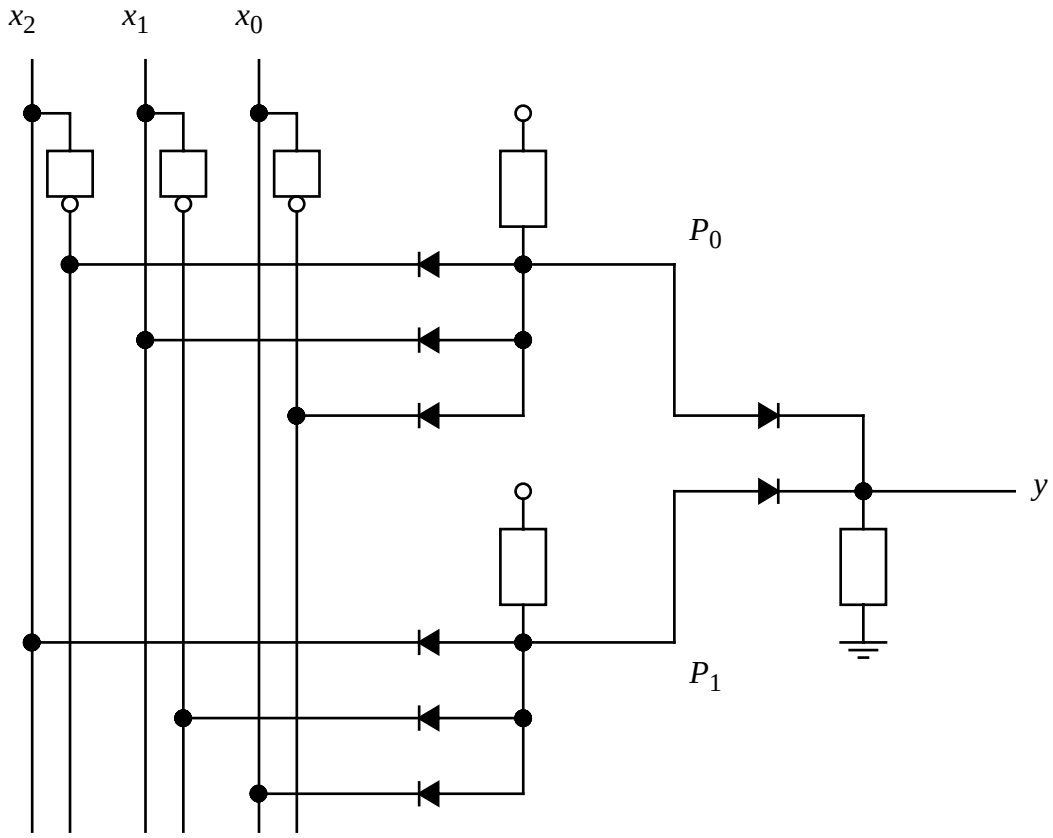


Bild 3.59: Realisierung des Schaltnetzes nach Bild 10.3 mit Diodengattern

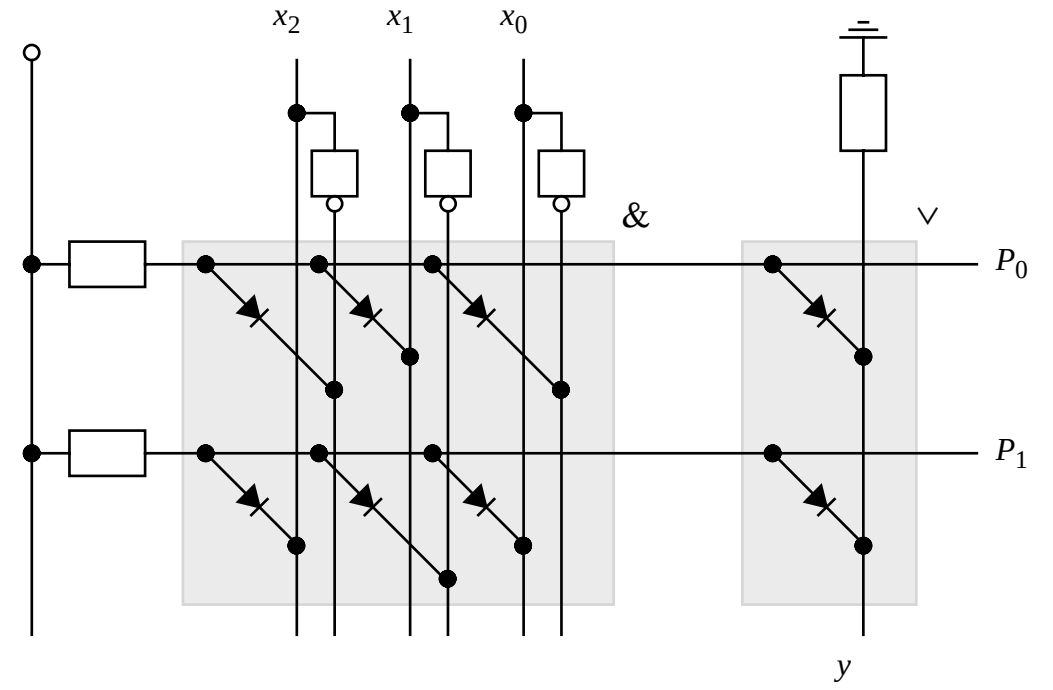


Bild 3.60: Topologische Umgestaltung der Diodenschaltung nach Bild 10.5
& = UND-Matrix, ∨ = ODER-Matrix.

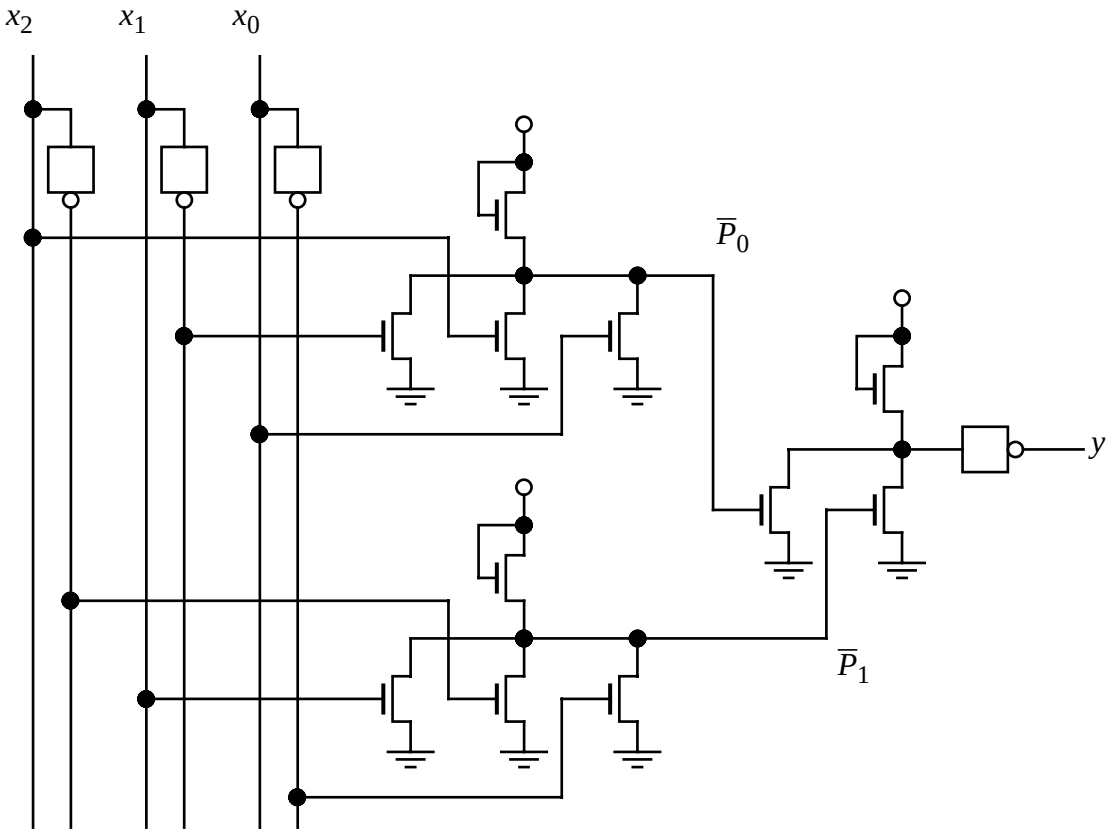


Bild 3.61: Realisierung des Schaltnetzes nach Bild 10.4 mit NMOS-Transistorgattern

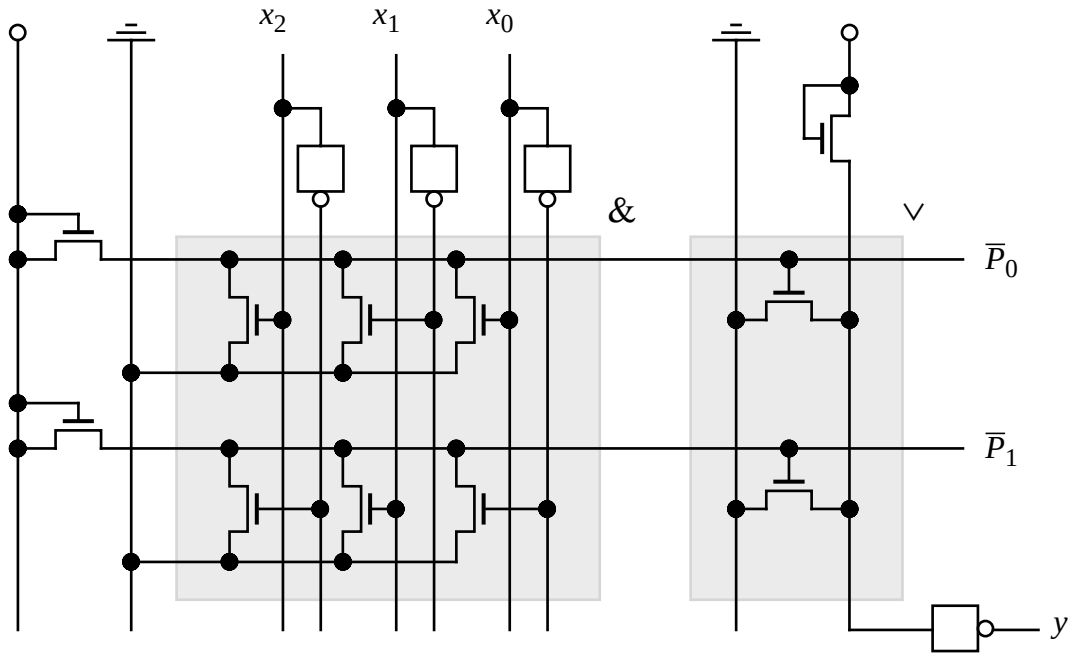


Bild 3.62: Topologische Umgestaltung der NMOS-Transistorschaltung nach Bild 10.7
& = UND-Matrix, ∨ = ODER-Matrix.

e) Allgemeines Strukturschema

Matrixförmige Logikschaltungen nutzen das Potential der Mikroelektronik bei Entwurf, Herstellung und Betrieb besser aus als unregelmäßige Strukturen. Da sie *programmierbar* sind, gehören sie zur Klasse der anwendungsspezifischen integrierten Schaltungen („Application Specific Integrated Circuits", ASIC). Indem man von schaltungstechnischen Einzelheiten absieht, können beide Schaltkreistechnologien, bipolar und MOS, durch das allgemeine Strukturschema im nachfolgenden Bild abstrahiert werden. Es besteht im wesentlichen aus einer UND-, gefolgt von einer ODER-Matrix, die beliebige logische Verknüpfungen in zweistufiger disjunktiver Form leisten.

Eine spezielle, anwendungsspezifische Schaltfunktion wird durch die *Personalisierung* der beiden Matrizen realisiert, wie es im Strukturschema durch die Anschlusspunkte angedeutet ist. Man kann die Personalisierung weiter abstrahieren und sie als *Programmierung* der Matrizen mit einem elementaren Binärcode betrachten, wie er nachfolgend angegeben wird. Man bezeichnet deshalb die in den folgenden Abschnitten zusammengestellten Varianten des allgemeinen Strukturschemas als Programmierbare Logische Bausteine („Programmable Logic Devices", PLD).

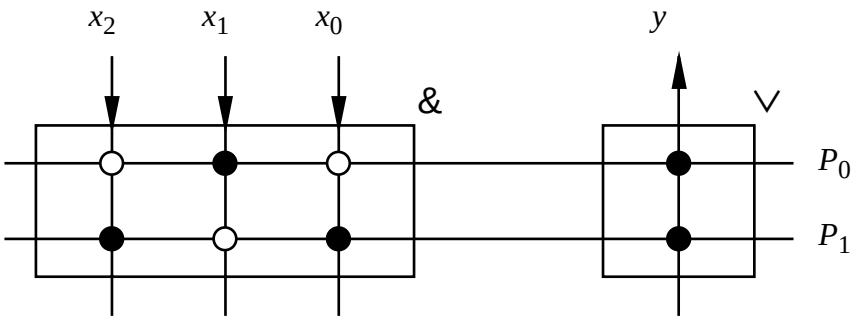


Bild 3.63: Allg. Strukturschema personalisierbarer Matrizen;
& = UND-Matrix; ∨ = ODER-Matrix.

- Die UND-Matrix erzeugt die Konjunktion („logisches Produkt") *Pj* der angeschlossenen Eingangsvariablen x_i. In dieser Matrix bedeutet:

		Binärcode
weißer Punkt:	die verneinte Eingangsvariable $\bar{x}_i$ ist angeschlossen	0
schwarzer Punkt:	die bejahte Eingangsvariable x_i ist angeschlossen	1
kein Punkt:	die Eingangsvariable x_i ist nicht angeschlossen	-

- Die ODER-Matrix erzeugt die Disjunktion („logische Summe") der angeschlossenen Konjunktionen *Pj*. In dieser Matrix bedeutet:

		Binärcode
schwarzer Punkt:	die Konjunktion *Pj* ist angeschlossen	1
kein Punkt:	die Konjunktion *Pj* ist nicht angeschlossen	0

Zur Realisierung von Bündelschaltfunktionen mit mehreren Funktionsspalten y_j kann die Anzahl der Spalten in der ODER-Matrix im obigen Bild entsprechend erhöht werden.

3.4.2 Festwertspeicher („ROM”)

Beispiel 3.4.4:

Ein binärer Volladdierer soll durch programmierbare Digitalbausteine, d.h. personalisierbare Schaltungen in Matrizenform realisiert werden. Er kann durch eine Wahrheitstabelle wie im nebenstehenden Bild spezifiziert werden. Dabei sind x_1 und x_0 die zusammen mit dem Übertrag c_0 aus der vorhergehenden Addition zu addierenden binären Variablen. Gebildet werden die Summe S und der neue Übertrag C_1.

j	c_0	x_1	x_0	S	C_1
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

Bild 3.64: Funktionsbeschreibung eines binären Volladdierers

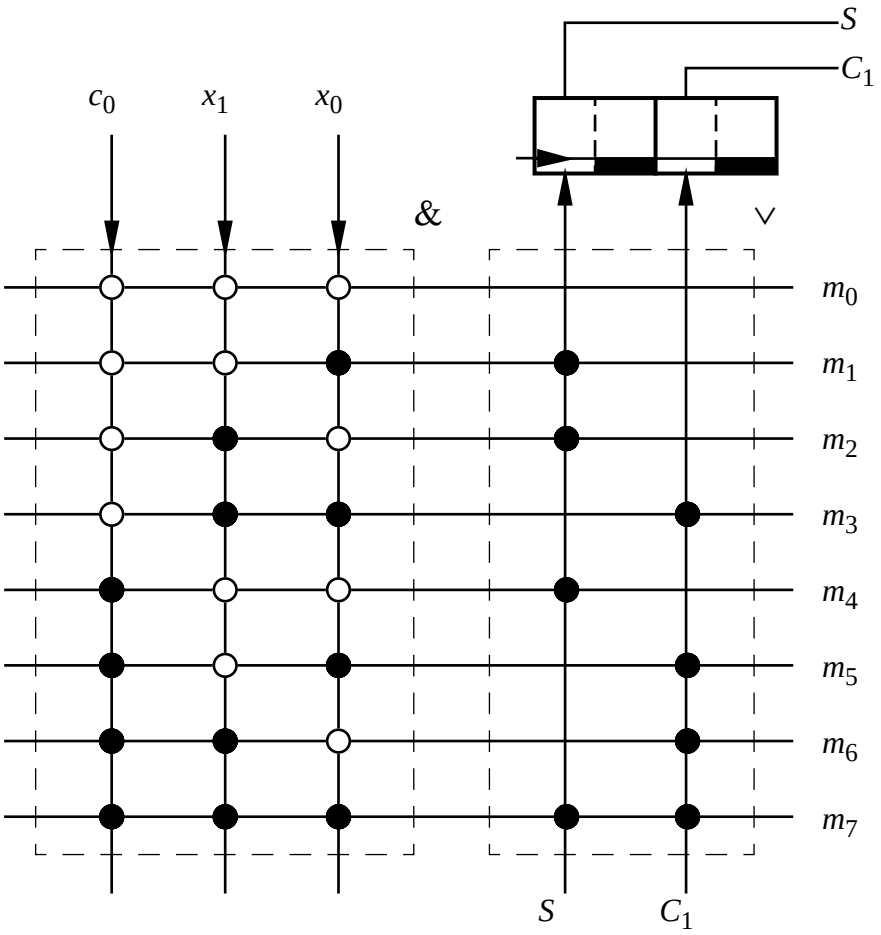


Bild 3.65: Festwertspeicher („Read Only Memory”, ROM);
& = vordefinierter 1-aus-8 Decoder, ∨ = programmierbare Speichermatrix.

Personalisiert man die UND-Matrix wie im vorhergehenden Bild für $n = 3$ Eingangsvariable schematisch gezeigt, so werden durch ihre 2^n Zeilen sämtliche Minterme m_j gebildet, wobei $j = 0 \ldots n-1$. Diese UND-Matrix entspricht dem 1-aus-*n* Decoder eines Speichers mit wahlfreiem Zugriff („Random Access Memory", RAM) mit Einebenenadressierung. Wie das vorhergehende Bild ebenfalls zeigt, wird die ODER-Matrix isomorph zu den beiden Funktionsspalten der Wahrheitstabelle für die Summe *S* und den neuen Übertrag C_1 personalisiert („programmiert"); sie ist die eigentliche Speichermatrix.

Der 1-aus-*n* Decoder wird vom Hersteller vordefiniert, die Speichermatrix anwendungsspezifisch programmiert. Wird der Inhalt der Speichermatrix vom Hersteller nach Anwenderangaben fest personalisiert, so spricht man von einem *Festwertspeicher* („Read Only Memory", ROM). Anmerkung: Die Bezeichnung RAM („Random Access Memory") beschränkt sich in der Praxis auf vom Anwender im laufenden Betrieb *schreibbare* Speicher mit wahlfreiem Zugriff, obwohl auch ein sog. ROM, wie man seinem Strukturschema entnehmen kann, dank des 1-aus-*n* Decoders den wahlfreien Zugriff auf den Inhalt der Speichermatrix erlaubt.

Man beachte ferner, dass der Festwertspeicher in diesem Zusammenhang als rein kombinatorisches Schaltnetz wirkt. Ein sequentielles Schaltwerk entsteht daraus erst dann, wenn die beiden im obigen Strukturschema des ROM angedeuteten Flipflops am Ausgang der Speichermatrix auf den Eingang des Decoders rückgekoppelt werden.

3.4.3 Programmierbare UND-Matrixlogik („PAL")

Nun soll die gegebene Bündelschaltfunktion des Volladdierers minimiert werden, wobei das Funktionsbündel hier aus der Summe S und dem negierten Übertrag $\overline{C}_1$ besteht. (Die Negation erfolgt im Vorgriff auf die im nächsten Abschnitt vorgestellte Realisierung und kann durch Verwendung des negierenden Ausgangs eines Ausgangsflipflops wieder aufgehoben werden.) Die folgenden KV-Diagramme zeigen, dass sich nur die Schaltfunktion für $\overline{c}_1$ minimieren lässt.

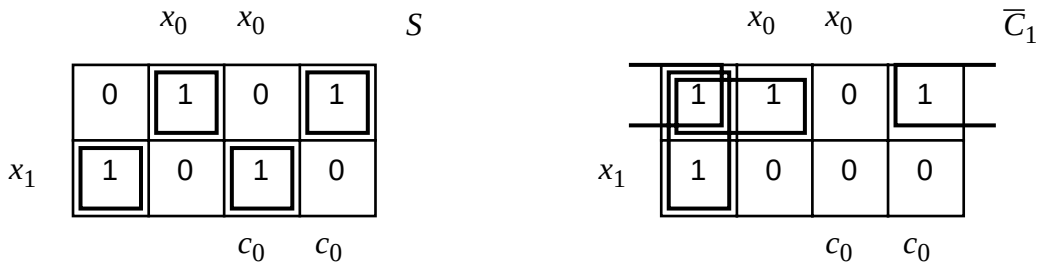


Bild 3.66: KV-Diagramme des Volladdierers mit Überdeckung durch Primblöcke

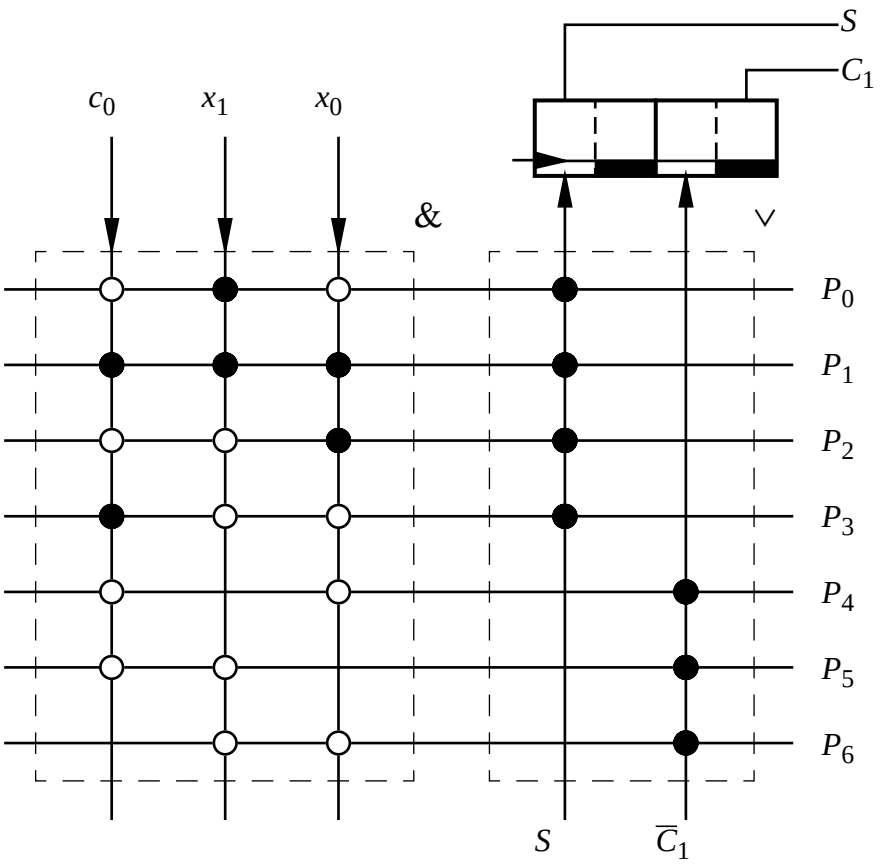


Bild 3.67: Programmierbare UND-Matrixlogik („Programmable AND-Array Logic", PAL); & = programmierbare UND-Matrix, ∨ = vordefinierte ODER-Matrix.

Man erhält durch Überdecken der Einsstellen mit Primblöcken in disjunktiver *Minimalform*:

DMF $S = (\bar{c}_0\, x_1\, \bar{x}_0) \vee (c_0\, x_1\, x_0) \vee (\bar{c}_0\, \bar{x}_1\, x_0) \vee (c_0\, \bar{x}_1\, \bar{x}_0) = P_0 \vee P_1 \vee P_2 \vee P_3$

$\overline{C}_1 = (\bar{c}_0\, \bar{x}_0) \vee (\bar{c}_0\, \bar{x}_1) \vee (\bar{x}_1\, \bar{x}_0) = P_4 \vee P_5 \vee P_6$

Die zur DMF isomorphe Personalisierung zeigt das nächste Bild. In der UND-Matrix des Bausteins werden die gewünschten Konjunktionen („Produktterme") P_j zeilenweise erzeugt. In einer durch den Hersteller des Bausteins vordefinierten ODER-Matrix ausreichender Größe werden sie dann disjunktiv verknüpft. Da nur die UND-Matrix anwendungsspezifisch personalisiert wird, während die ODER-Matrix vordefiniert ist, nennt man einen Baustein mit dieser Struktur *Programmierbare UND-Matrixlogik* („Programmable And-Array Logic", PAL).

Ein PAL benötigt weniger Zeilen als ein ROM, falls sich die zu realisierenden Schaltfunktionen minimieren lassen. Dies zeigt auch ein Vergleich des PAL-Strukturschemas mit dem des ROM, die für dieselbe Funktionsbeschreibung eines Volladdierers personalisiert wurden. Die Anzahl der Spalten dagegen ist durch die Anzahl der Eingangs- und Ausgangsvariablen, d.h. durch die Funktionsbeschreibung vorgegeben und deshalb in beiden Fällen gleich.

3.4.4 Programmierbare Logische Matrix („PLA”)

Die Anzahl der Spalten matrixförmiger Logikschaltungen ist, wie erwähnt, durch die Funktionsbeschreibung („Spezifikation”) vorgegeben. Das Entwurfsziel besteht daher in der Minimierung der Anzahl der Zeilen. Das Ergebnis konventioneller Minimierungsverfahren, z.B. mit KV-Diagrammen oder nach Quine-McCluskey, ist die Minimierung der Anzahl logischer Gatter. Sie werden, wie gezeigt, beim Entwurf von PALs vorteilhaft eingesetzt. Falls es nun gelingt, logische Produktterme P_j mehrfach auszunutzen, erhält man ein weiteres Minimierungspotential.

Man betrachte die folgenden KV-Diagramme: Überdeckt man die Einsstellen nicht notwendigerweise mit Primblöcken, sondern mit Blöcken, die in den KV-Diagrammen mehrerer Ausgangsvariabler vorkommen, hier S und $\overline{C}_1$, so kann man sie mehrfach ausnutzen, falls in der matrixförmig ausgelegten Schaltung nicht nur die UND-, sondern auch die ODER-Matrix personalisiert werden kann. Diese Überlegungen führen zur *Programmierbaren Logischen Matrix* („Programmable Logic Array”, PLA).

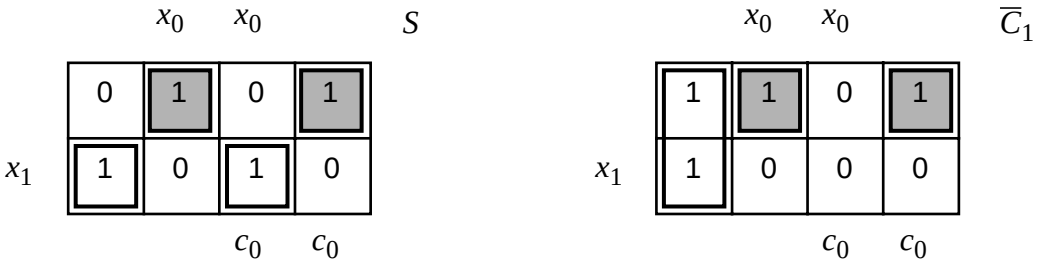


Bild 3.68: KV-Diagramme des Volladdierers mit Mehrfachausnutzung von Blöcken

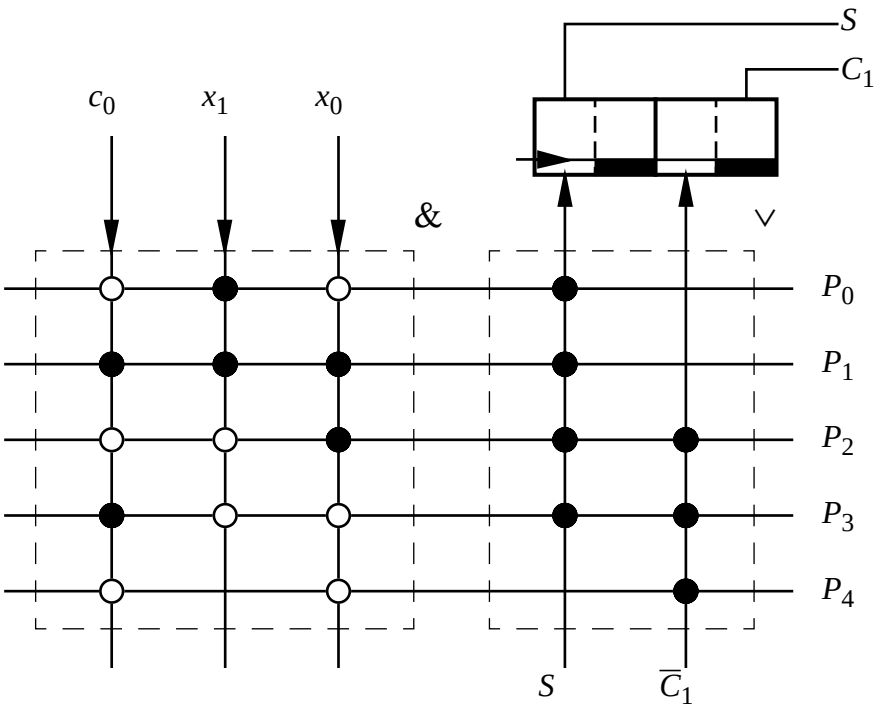


Bild 3.69: Programmierbare Logische Matrix („Programmable Logic Array”, PLA);
& = programmierbare UND-Matrix, ∨ = programmierbare ODER-Matrix.

Man erhält durch Überdecken der Einsstellen mit mehrfach ausgenutzten Blöcken in disjunktiver *Normalform*:

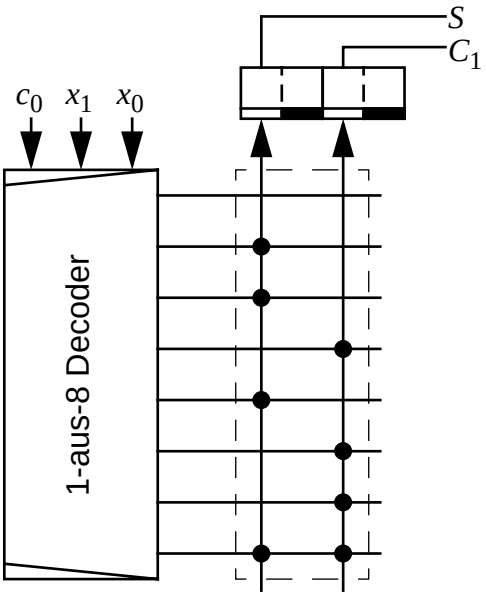
DNF $S = (\bar{c}_0\, x_1\, \bar{x}_0) \vee (c_0\, x_1\, x_0) \vee (\bar{c}_0\, \bar{x}_1\, x_0) \vee (c_0\, \bar{x}_1\, \bar{x}_0) = P_0 \vee P_1 \vee P_2 \vee P_3$

$\overline{C}_1 = (\bar{c}_0\, \bar{x}_1\, x_0) \vee (c_0\, \bar{x}_1\, \bar{x}_0) \vee (\bar{c}_0\, \bar{x}_0) = P_2 \vee P_3 \vee P_4$

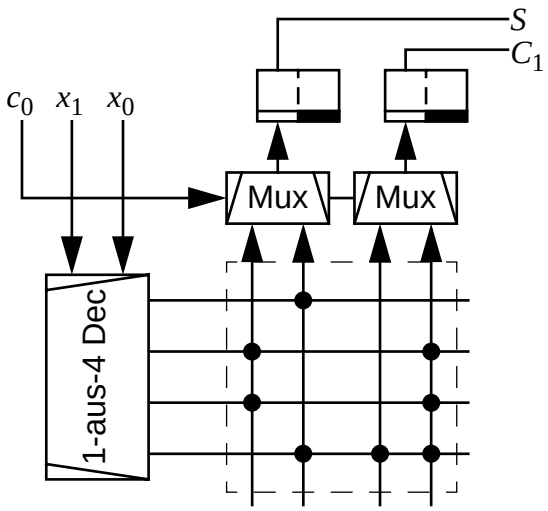
Das Beispiel des Volladdierers zeigt das vorhergehende Bild. Die UND-Matrix des PLA erzeugt zeilenweise die gewünschten Produktterme P_j, in der ODER-Matrix wählt man die jeweils benötigten Produktterme aus, d.h. beide Matrizen sind anwendungsspezifisch zu personalisieren. Im Vergleich zum PAL ergibt sich beim PLA eine Einsparung von zwei weiteren Produkttermen durch deren Mehrfachausnutzung.

3.4.5 Speicher mit wahlfreiem Zugriff („RAM")

a) Ein-Ebenen-Adressierung



b) Zwei-Ebenen-Adressierung



3.4.6 Schaltwerksentwurf mit PLA

Implementiert man mit einem PLA nicht nur (Bündel-)Schaltfunktionen, sondern endliche diskrete Automaten, so ergibt sich ein zusätzliches Minimierungspotential: Bei geeigneter binärer Codierung der Automatenzustände können sowohl das Überführungsschaltnetz δ, das die Folgezustände erzeugt, als auch das Ausgabeschaltnetz λ vereinfacht werden. Man findet in der Literatur eine Fülle heuristischer Algorithmen zur Zustandscodierung, die hier jedoch nicht Gegenstand der Betrachtung sind. Nachfolgend soll das Entwurfsbeispiel eines Schaltwerks aus dem vorhergehenden Kapitel speziell auf die PLA-Struktur abgestimmt werden.

a) Spezifikation

Das Schaltwerk wird im folgenden Bild durch seine symbolische Ablauftabelle und den zugehörigen Ablaufgraphen formal spezifiziert.

Z^t	X^t	Z^{t+1}	Y^t
	$x_1\, x_0$		$y_2\, y_1\, y_0$
00	0 0 0 1 1 -	01 00 11	1 0 *
01	0 0 1 0 - 1	01 10 11	* 0 1
11	0 0 1 0 - 1	00 10 00	0 * 1
10	- -	01	1 1 0

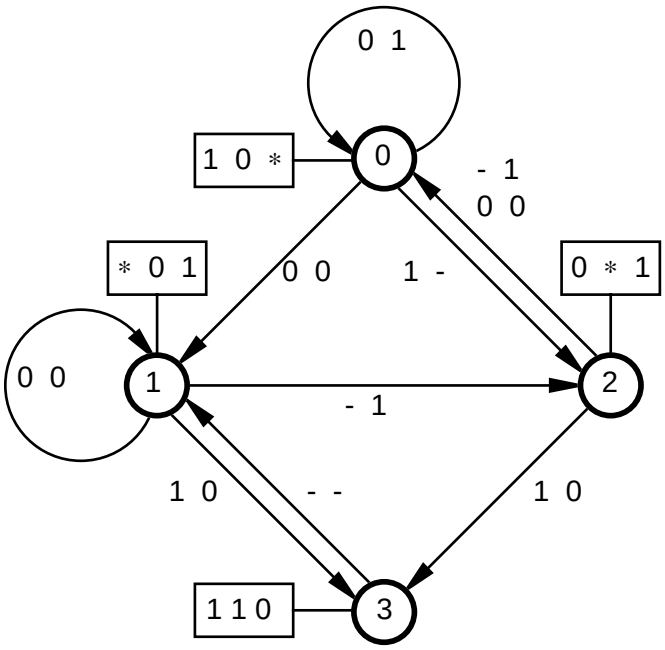


Bild 3.70: Spezifikation eines durch ein PLA zu realisierenden Moore-Automaten; links: Symbolische Ablauftabelle, rechts: Ablaufgraph.

b) Zustandscodierung

Willkürlich wurde der bekannte *Gray-Code* zur Zustandscodierung gewählt:

$$Z_0 = 0 \leftrightarrow (0, 0) \qquad Z_2 = 2 \leftrightarrow (1, 1)$$
$$Z_1 = 1 \leftrightarrow (0, 1) \qquad Z_3 = 3 \leftrightarrow (1, 0)$$

Die sich aus der gewählten Zustandscodierung ergebende codierte Ablauftabelle wurde bereits im Entwurfsbeispiel des vorhergehenden Kapitels wiedergegeben. Sie weist zehn Zeilen auf, die durch ebenso viele Zeilen („Produktleitungen") des PLA realisiert werden müssten. Dieser Aufwand soll minimiert werden.

c) Wahl der Speicherglieder

Zur Zustandsspeicherung seien nun willkürlich *D-Flipflops* gewählt. Für ihre Ansteuervariablen (D_1, D_0) wurde im Entwurfsbeispiel des vorhergehenden Kapitels eine Spalte angegeben; wegen der trivialen, nur verzögernden Funktion des D-Flipflops ist ihre Belegung mit der Codierung Q^{t+1} der Folgezustände identisch.

d) Minimierung des Bündelschaltnetzes

Zunächst soll die Minimierung konventionell durch das Bilden von Primblöcken erfolgen. Die Funktionsspalten für die Ansteuervariablen (D_1, D_0) der D-Flipflops werden aus der codierten Ablauftabelle des Entwurfsbeispiels im vorhergehenden Kapitel in die folgenden KV-Diagramme übernommen. Man findet bei der Minimierung durch Bilden von *Primblöcken*, daß sich die zur Minimierung der Zeilenanzahl von PLAs wichtige *Mehrfachverwendung* von Blöcken nur für zwei davon erreichen läßt (im folgenden Bild schraffiert hervorgehoben). Mit disjunktiven Minimalformen wären für die Ansteuervariablen (D_1, D_0) zunächst sechs Produktterme, d.h. Zeilen für das PLA erforderlich:

DMF $D_1 = (\bar{q}_1\, x_1) \vee \mathbf{(\bar{q}_1\, q_0\, x_0)} \vee (q_0\, x_1\, \bar{x}_0)$

$D_0 = (\bar{q}_1\, \bar{x}_1\, \bar{x}_0) \vee \mathbf{(\bar{q}_1\, q_0\, x_0)} \vee (\bar{q}_0\, x_1) \vee (q_1\, \bar{q}_0)$

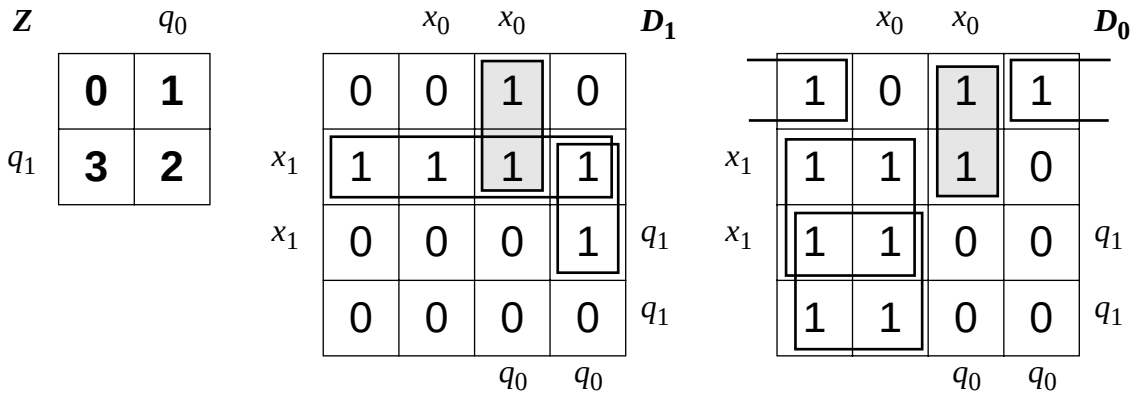


Bild 3.71: KV-Diagramme der Ansteuervariablen (D_1, D_0); Überdeckung mit Primblöcken; Zustandscodierung im Gray-Code, Zustandsspeicherung in D-Flipflops.

Die Funktionsspalten für die Ausgangsvariablen (y_2, y_1, y_0) des PLA-Schaltwerks werden ebenfalls aus der codierten Ablauftabelle des Entwurfsbeispiels im vorhergehenden Kapitel in die folgenden drei KV-Diagramme übernommen. Man findet bei der Minimierung durch Bilden von *Primblöcken* in den KV-Diagrammen der Ausgangsvariablen (y_2, y_1, y_0) in disjunktiver Minimalform zunächst drei Produktterme. Da keine Mehrfachverwendung der Primblöcke möglich ist, wären drei weitere und damit insgesamt neun Zeilen für das PLA erforderlich:

DMF $y_2 = \bar{q}_0$ $\qquad$ $y_1 = q_1$ $\qquad$ $y_0 = q_0$

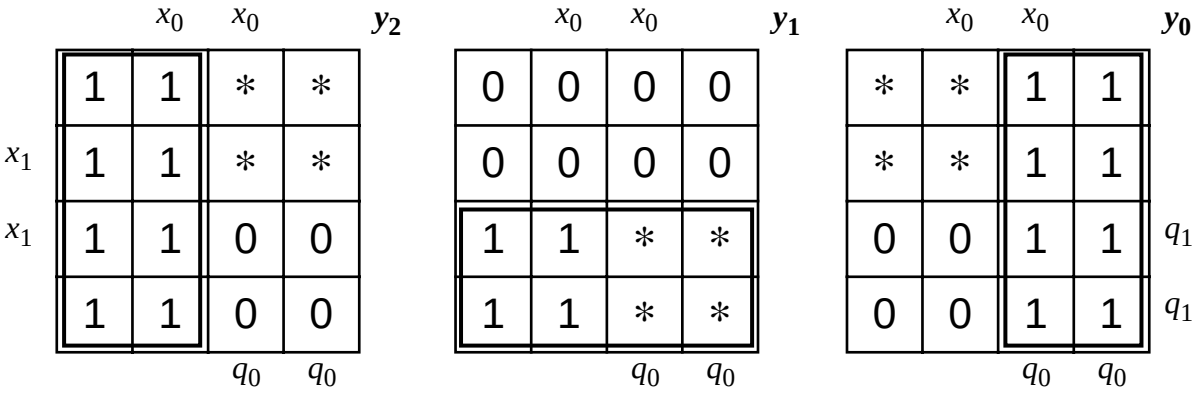


Bild 3.72: KV-Diagramme der Ausgangsvariablen (y_2, y_1, y_0); Überdeckung mit Primblöcken; Zustandscodierung im Gray-Code, Zustandsspeicherung in D-Flipflops.

Bei der Minimierung von PLAs kommt es auf die Minimierung der Zeilenanzahl an, die wie erläutert Konjunktionen („logische Produkte") implementieren. Dies wird, wie ebenfalls bereits erwähnt, durch *Mehrfachverwendung* von logischen Produkttermen erreicht. Zur Ermittlung der disjunktiven Formen der betreffenden Variablen müssen die Einsstellen in den KV-Diagrammen zwar vollständig mit Blöcken überdeckt werden, aber nicht notwendigerweise mit Primblöcken: Betrachtet man die obigen fünf KV-Diagramme gemeinsam und verkleinert man ggf. die überdeckenden Blöcke, ohne die vollständige Überdeckung der Einsstellen zu gefährden, so entdeckt man Möglichkeiten zur Mehrfachverwendung von Blöcken, wie es in den folgenden sechs KV-Diagrammen dargestellt ist. Wie der paarweise Vergleich dieser KV-Diagramme zeigt, benötigt man jetzt nur noch insgesamt sieben Produktterme, die durch entsprechende Zeilen im PLA implementiert werden. Ihre Anzahl konnte somit von ursprünglich zehn weiter verringert werden.

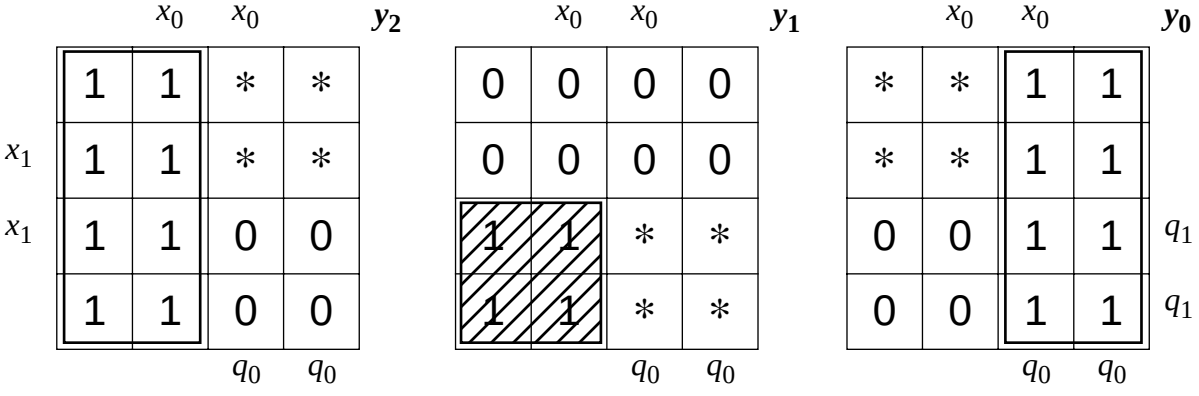


Bild 3.73: KV-Diagramme der Ausgangsvariablen; Überdeckung mit mehrfach verwendeten Blöcken; Zustandscodierung im Gray-Code, Zustandsspeicherung in D-Flipflops.

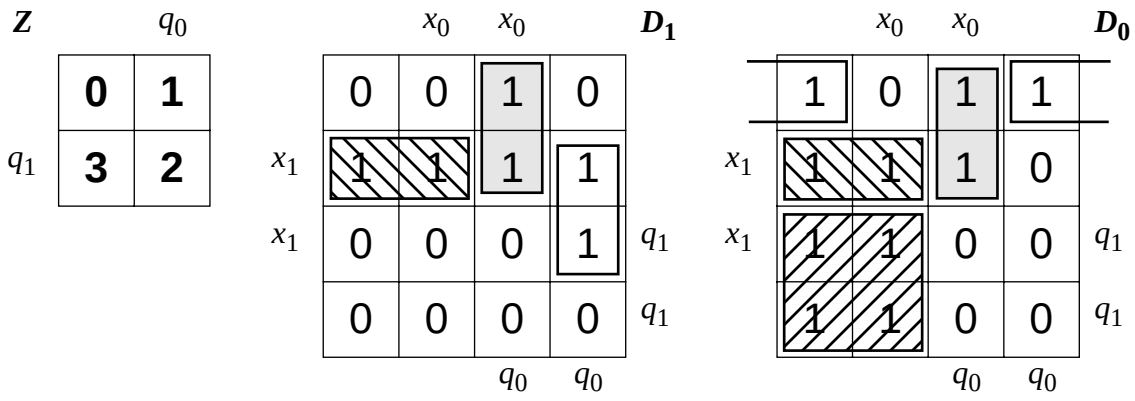


Bild 3.74: KV-Diagramme der Ansteuervariablen; Überdeckung mit mehrfach verwendeten Blökken; Zustandscodierung im Gray-Code, Zustandsspeicherung in D-Flipflops.

e) PLA-Schaltwerksstruktur

Zur Implementierung des Moore-Automaten, der im obigen Abschnitt 4.5.1 spezifiziert wurde, werden die oben ermittelten Konjunktionen (&), d.h. die logischen Produktterme P_i in die UND-Matrix, ihre disjunktive Verknüpfung ($\vee$) in die ODER-Matrix der PLA-Struktur abgebildet. Das Ergebnis des hier durchgeführten Entwurfsablaufs zeigt das folgende Gatterschaltbild.

Man erhält folgende Produktterme, d.h. Blöcke, die die Einsstellen vollständig überdecken:

$P_0 = \overline{q}_0$ $P_1 = q_1 \,\&\, \overline{q}_0$ $P_2 = q_0$

$P_3 = \overline{q}_1 \,\&\, \overline{q}_0 \,\&\, x_1$ $P_4 = \overline{q}_1 \,\&\, q_0 \,\&\, x_0$ $P_5 = q_0 \,\&\, x_1 \,\&\, \overline{x}_0$

$P_6 = \overline{q}_1 \,\&\, \overline{x}_1 \,\&\, \overline{x}_0$

DNF $y_2 = P_0$ $y_1 = \boldsymbol{P_1}$ $y_0 = P_2$

$D_1 = \boldsymbol{P_3} \vee \boldsymbol{P_4} \vee P_5$ $D_0 = \boldsymbol{P_1} \vee \boldsymbol{P_3} \vee \boldsymbol{P_4} \vee P_6$

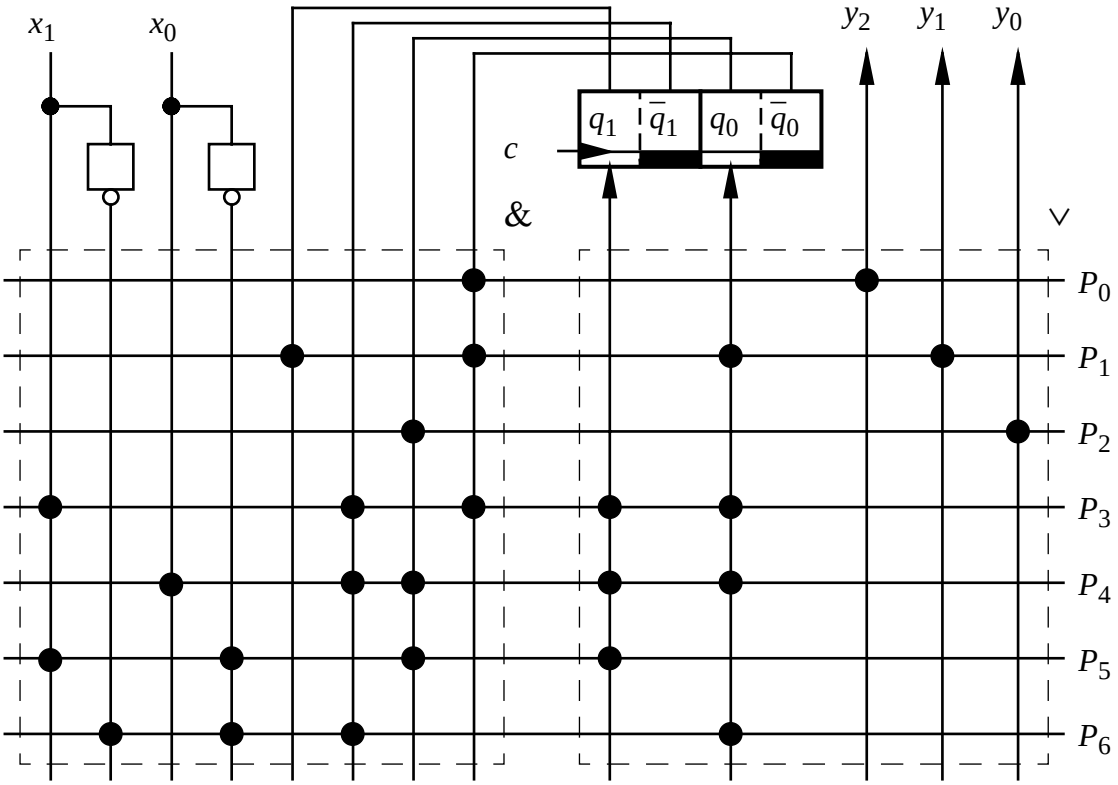


Bild 3.75: Realisierung des Moore-Automaten mit einem PLA;
& = personalisierte UND-Matrix, ∨ = personalisierte ODER-Matrix.

4 Register-Transfer-Ebene

Prof. Dr.-Ing. U.G. Baitinger

4.1 Begriffsbestimmungen

Die Registertransfer-Ebene abstrahiert von den Einzelheiten der konkreteren Logikebene und sie unterstützt die noch abstraktere Algorithmische Ebene, die das Verhalten der Systemeinheiten und ihr Zusammenwirken beschreibt. Sie verwendet als funktionelle Einheiten

- zur Informationsspeicherung: Register und Speichermatrizen („RAM“);
- zur Informationsverarbeitung: Digitale Rechenwerke („ALU“);
- zum Informationstransfer: Verbindungsleitungen („Busse“).

Die auf der Logikebene betrachteten Einheiten werden auf der RT-Ebene zu größeren Einheiten zusammengefasst:

- Binäre Variablen zu Vektoren, d.h. die Busbreiten sind in der Regel größer als 1 bit;
- Logische Gatter zu kombinatorischen Schaltungen, die Decoder, Multiplexer, Addierer, aber auch komplette Rechenwerke („ALU“) oder nur ein einzelnes logisches Gatter enthalten können;
- Flipflops zu Registern entsprechend den Bitbreiten der Vektoren und Busse; im einfachsten Fall kann ein Register auch nur aus einem einzelnen Flipflop bestehen.

Jeder *interaktiv,* d.h. mit einem Text- oder Graphik-Editor ausgeführte Entwurfsschritt, sei es in Hardware oder Software, muss durch Simulation auf Korrektheit überprüft werden („Validierung“), bevor der nächste Entwurfsschritt ausgeführt werden kann.

Automatisierte Entwurfsschritte, die mit einem rechnergestützten Synthesewerkzeug („CAD Tool“) ausgeführt werden, bedürfen dagegen grundsätzlich keiner Validierung, da sie in der Regel ein korrektes Entwurfsergebnis erzeugen („correctness by construction“), doch ist auch hier eine anschließende Simulation möglich und ratsam.

Beim rechnergestützten Entwurf mikroelektronischer Digitalsysteme können die zu validierenden Entwurfsdaten direkt als „Modellbeschreibung“ für den Simulator verwendet werden. Simulationen auf der RT-Ebene bilden folgende Vorgänge nach:

- den Zugriff auf Variablenwerte in Speicherplätzen,
- ihre Ausbreitung auf Busleitungen
- und ihre Verknüpfung in Rechenwerken, die auch als arithmetisch/logische Einheiten („Arithmetic/Logic Unit“, ALU) bezeichnet werden.

Im folgenden sollen für die RT-Ebene - nach einer Vorstellung der zu betrachtenden Funktionseinheiten - Methoden der strukturellen Synthese aus funktionalen Spezifikationen vorgestellt werden. Dabei geht es im wesentlichen um die zeitliche Ablaufplanung von Steuerungsfolgen, die Bereitstellung von Betriebsmitteln und die Zuweisung von Speicherplätzen.

4.2 Verbindungsstrukturen

4.2.1 Decoder *vs.* Demultiplexer

Decodierer (*decoder* - engl.) DIN 44 300

> „Ein Code-Umsetzer mit mehreren (n) Eingängen und (2^n) Ausgängen, bei dem für jede spezifische Kombination von Eingangssignalen immer nur je ein bestimmter Ausgang ein Signal abgibt."

Eine Funktionseinheit mit dem folgenden Gatterschaltbild wirkt, je nach Ansteuerung, entweder als *Decoder* oder als *Demultiplexer*. Sie kann daher wahlweise durch eines der beiden Schaltsymbole im übernächsten Bild abstrahiert werden.

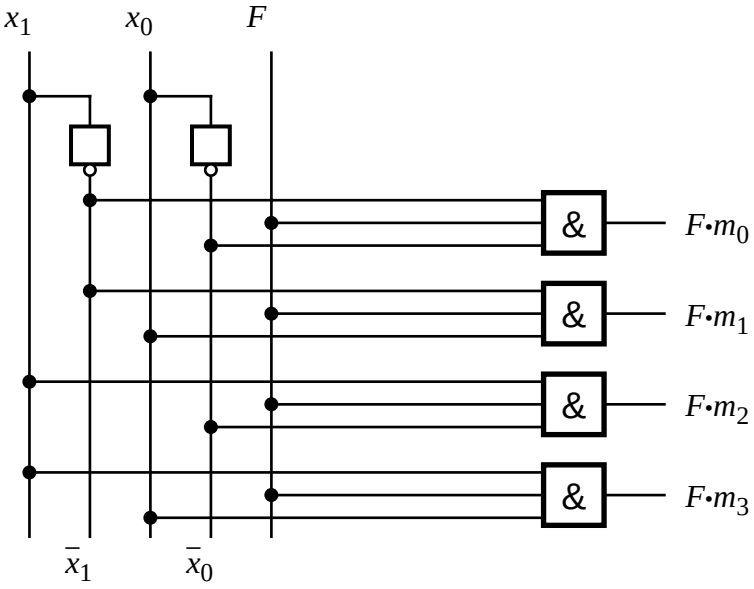


Bild 4.1: Gatterschaltbild eines Decoders bzw. Demultiplexers auf der Logik-Ebene

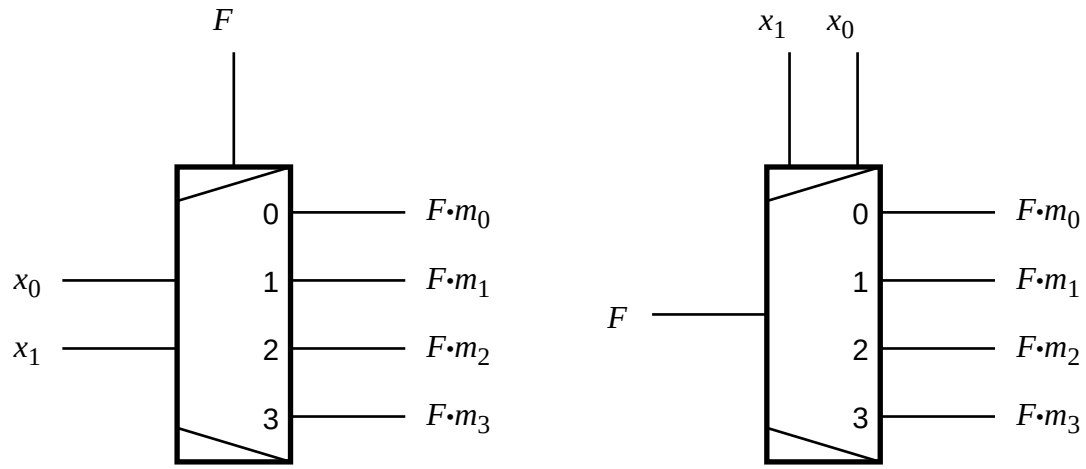


Bild 4.2: Schaltsymbole für Decoder und Demultiplexer auf der Registertransfer-Ebene

4.2.2 Multiplexer *vs.* Demultiplexer

Multiplexer (*multiplexer* - engl.) DIN 44 300

> „Eine Funktionseinheit, die Nachrichten von Nachrichtenkanälen *einer* Anzahl in Nachrichtenkanäle *anderer* Anzahl übergibt."

Eine Funktionseinheit mit dem folgenden Gatterschaltbild wirkt, je nach Ansteuerung, entweder als *Multiplexer* oder als *Demultiplexer.* Sie kann daher wahlweise durch eines der beiden Schaltsymbole im übernächsten Bild abstrahiert werden.

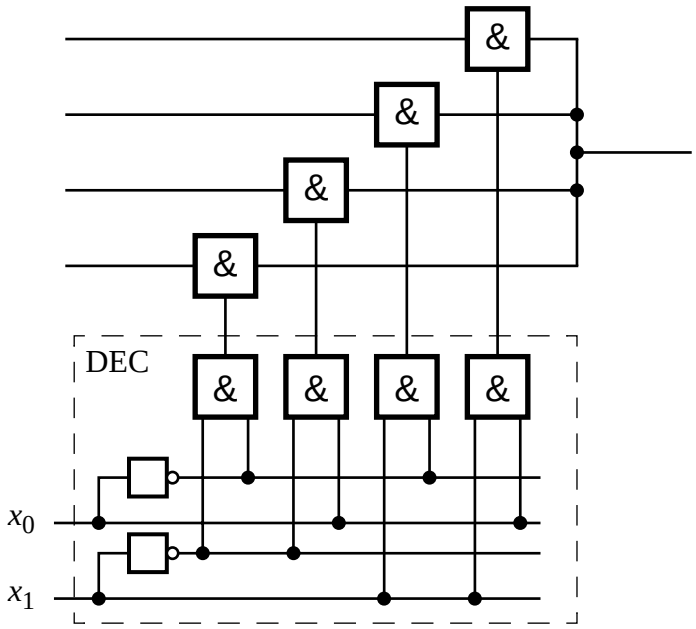


Bild 4.3: Gatterschaltbild eines Multiplexers bzw. Demultiplexers auf der Logik-Ebene

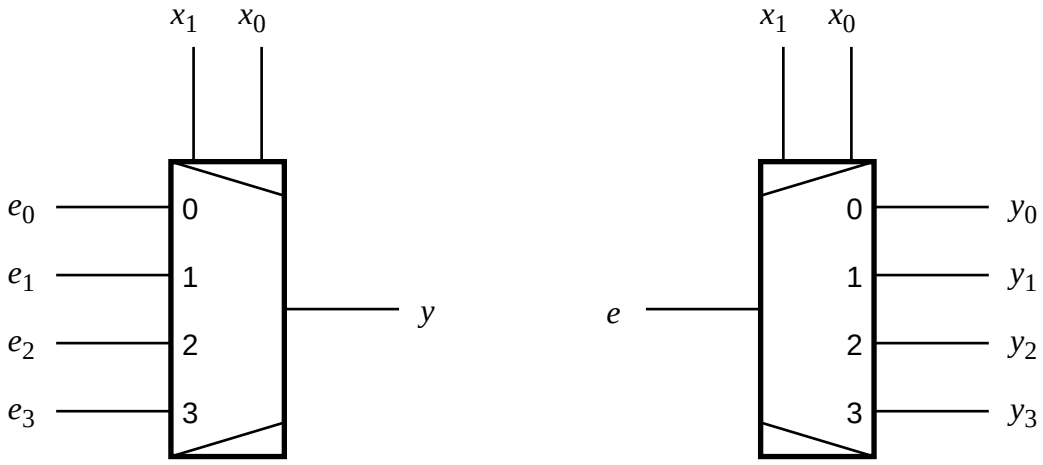


Bild 4.4: Schaltsymbole für Multiplexer und Demultiplexer auf der Registertransfer-Ebene

4.2.3 Sammelleitung („Bus“)

Die Bezeichnung „Bus” ist eine Abkürzung des Wortes *omnibus* (lat.), das *für alle* bedeutet. Ein Bus ist eine Sammelleitung, die *für alle* angeschlossenen Funktionseinheiten in einer Konferenzschaltung betrieben wird. Die „Breite“ *n* eines Bus, d.h. die Anzahl der übertragbaren Bits, bestimmt den maximal möglichen Datenfluss. Für jedes über den Bus zu transferierende Bit wird ein Multiplexer/Demultiplexer-Paar benötigt, wie das folgende Bild zeigt.

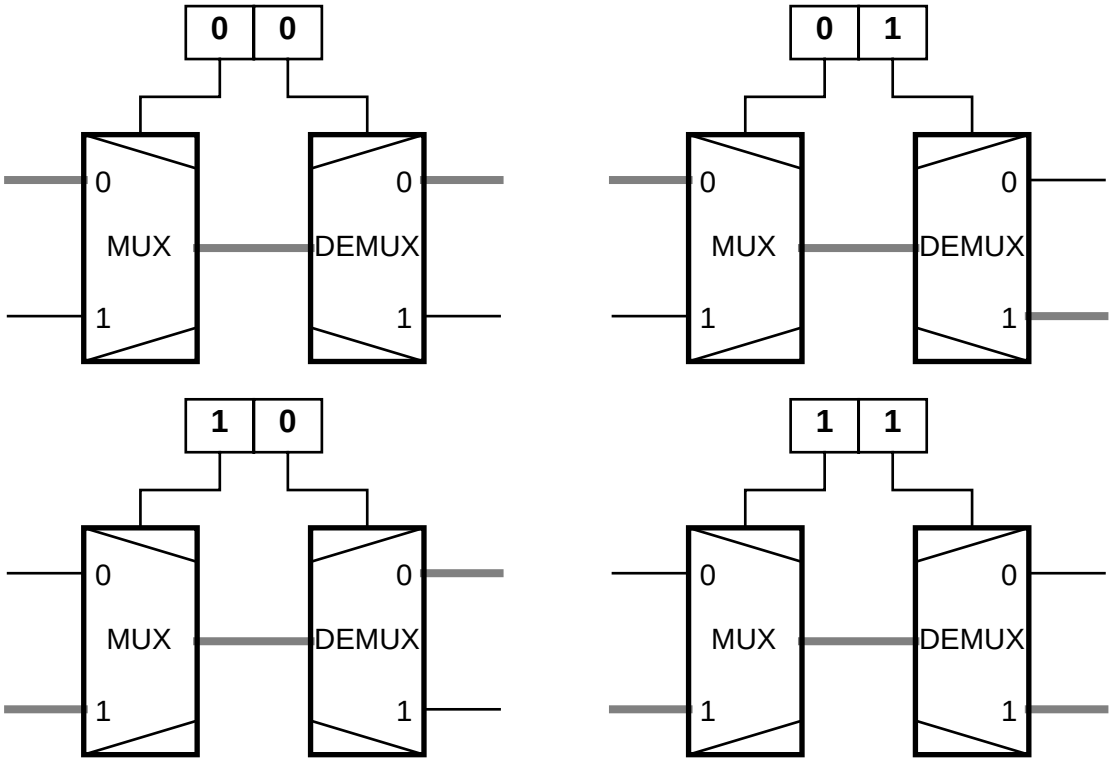
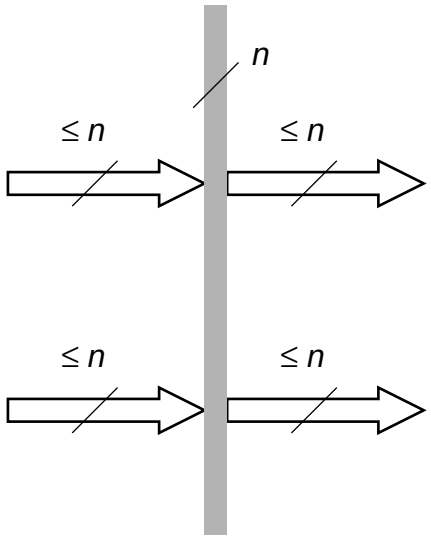


Bild 4.5: Die vier Ansteuerzustände des Mux/Demux-Paares einer 1 bit breiten Busleitung



n	Bus-Breite
$2^0 = 1$	1 bit
$2^3 = 8$	1 Byte
$2^4 = 16$	1 Halbwort
$2^5 = 32$	1 Vollwort

Bild 4.6: Vereinfachtes Schema eines *n* bit breiten Bus mit Zu- und Abgängen.
(Die Mux/Demux-Paare und ihre Ansteuerung werden nicht dargestellt.)

4.3 Speichereinheiten

Man kann die unterschiedlichen Speicheranordnungen nach der *Selektion*, d.h. der Art der Auswahl einer gespeicherten Einheit - wie Bit, Byte oder Wort - aus einer Menge gespeicherter Einheiten, nach ihrer funktionalen Organisation in drei Gruppen einteilen:

- Zellen sind punktförmige („0-dimensionale") Speicheranordnungen. Sie werden durch eine bistabile Speicherzelle realisiert und speichern 1 Bit. Für den Zugriff auf die Speicherzelle, um das Bit zu selektieren, benötigt man keine Adresse.
- Register sind linienförmige („1-dimensionale") Speicheranordnungen. Sie werden durch eine Reihe bistabiler Speicherzellen realisiert und speichern $b > 1$ Bits. Für den direkten Zugriff auf eine einzelne Speicherzelle, um ein Bit zu selektieren, benötigt man eine Adresse, die mit *ld b* Bits dual codiert wird.
- Speicher sind flächenförmige („2-dimensionale") Speicheranordnungen. Sie werden durch eine Matrix bistabiler Speicherzellen realisiert und speichern $w \cdot b >> 1$ Bits. Für den direkten Zugriff auf eine einzelne Speicherzelle, um ein Bit zu selektieren, benötigt man zwei Adressen, die mit *ld w* bzw. *ld b* Bits dual codiert werden.

4.3.1 Speicherzellen („Flipflops")

Zur Speicherung *binärer* Größen $Q \in \{0, 1\}$, die einen von zwei diskreten Werten annehmen können, benötigt man *bistabile* Speicherelemente, die zweier diskreter Zustände fähig sind („Flip-Flop"). Der aktuelle Zustand eines Flipflop, d.h. die zum Zeitpunkt t gespeicherte Zustandsvariable Q, wird abhängig von den Eingangsvariablen im nächsten Zeitpunkt $t + 1$ geändert oder beibehalten.

Man kann die Flipflops hinsichtlich ihrer Ansteuerung durch einen Takt c unterscheiden:

- Das Basis-Flipflop wird asynchron betrieben. Die Eingangsvariablen wirken unmittelbar auf den internen Zustand des Flipflop, eine Verzögerung zwischen Eingangs- und Ausgangsvariablen ergibt sich durch die internen Gatter („ungetaktetes Flipflop").
- Das Auffang-Flipflop wird synchron betrieben. Die Eingangsvariablen werden über Gatterschaltungen angelegt, die durch einen der beiden Werte einer Taktvariablen $c \in \{0, 1\}$ aktiviert werden („zustandsgetaktetes Flipflop").
- Dynamische Flipflops werden ebenfalls synchron betrieben. Sie verfügen über eine Eingangsschaltung, die entweder durch die Vorderflanke $c \rightarrow 1$ oder die Rückflanke $c \rightarrow 0$ eines Taksignals angesteuert wird („flankengetaktetes Flipflop").

Anschließend sind Schaltsymbole für Flipflops zusammengestellt, die die genannten Möglichkeiten zur Taktung berücksichtigen.

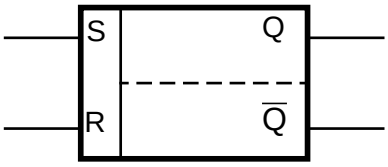


Bild 4.7: Basis-Flipflop, asynchron, ungetaktet

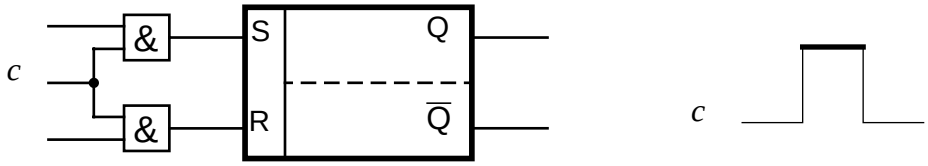


Bild 4.8: Auffang-Flipflop, synchron zustandsgetaktet

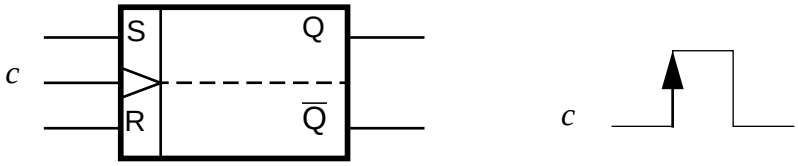


Bild 4.9: Dynamisches Flipflop, synchron vorderflankengetaktet

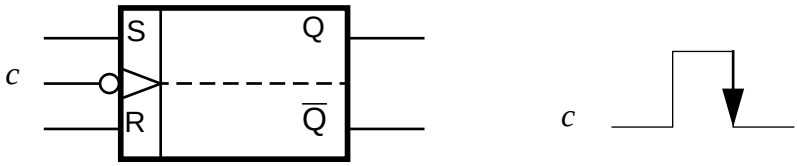


Bild 4.10: Dynamisches Flipflop, synchron rückflankengetaktet

Eine betriebssichere Anordnung zur Speicherung von 1 Bit ist das aus zwei synchronen Flipflops bestehende „Master-Slave-Flipflop", die mit den unterschiedlichen Flanken desselben Taktsignals angesteuert werden; dadurch ergibt sich eine Schiebemöglichkeit um „1/2 Bit".

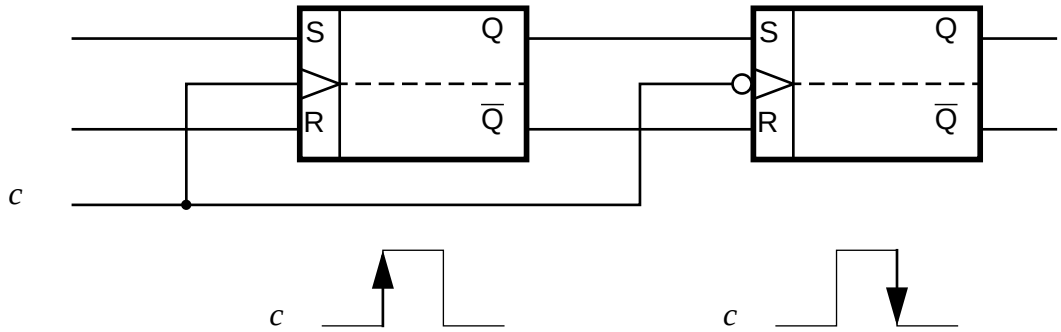


Bild 4.11: Master-Slave Flipflop

4.3.2 Registerspeicher

In umfangreicheren digitalen Schaltungen benötigt man zum schnellen Abspeichern und zur schnellen Rückgabe kleinerer Datenmengen einzelne Register, aber auch umfangreichere Registeransammlungen. Ein Register besteht aus je einem Speicherelement für jedes zu speichernde Bit. Man unterscheidet im wesentlichen:

- Schieberegister. In arithmetischen Operationen ist es des öfteren notwendig, eine gespeicherte Einheit, z.B. 1 Wort = 4 Bytes oder 1 Byte = 8 Bit, auf ein Taktsignal hin um eine bestimmte Anzahl Stellen nach links oder nach rechts zu verschieben.
- Zählerregister. In Digitalschaltungen müssen regelmäßig Vorgänge, Abläufe, Teiloperationen, Speicheradressen u. dergleichen ab- oder durchgezählt werden. Der Zähler ist deshalb ein unverzichtbares digitales Bauelement. Man erreicht die Zählfunktion durch zusätzliche Verknüpfungsschaltungen, die dafür sorgen, dass die Änderungshäufigkeit der Flipflopzustände im Register von Stufe zu Stufe halbiert wird.
- Speicherregister. Zur Synchronisation arithmetischer und/oder logischer Operationen ist es zweckmäßig, Zwischenergebnisse kurzzeitig abzuspeichern, bevor sie weiterverarbeitet werden.

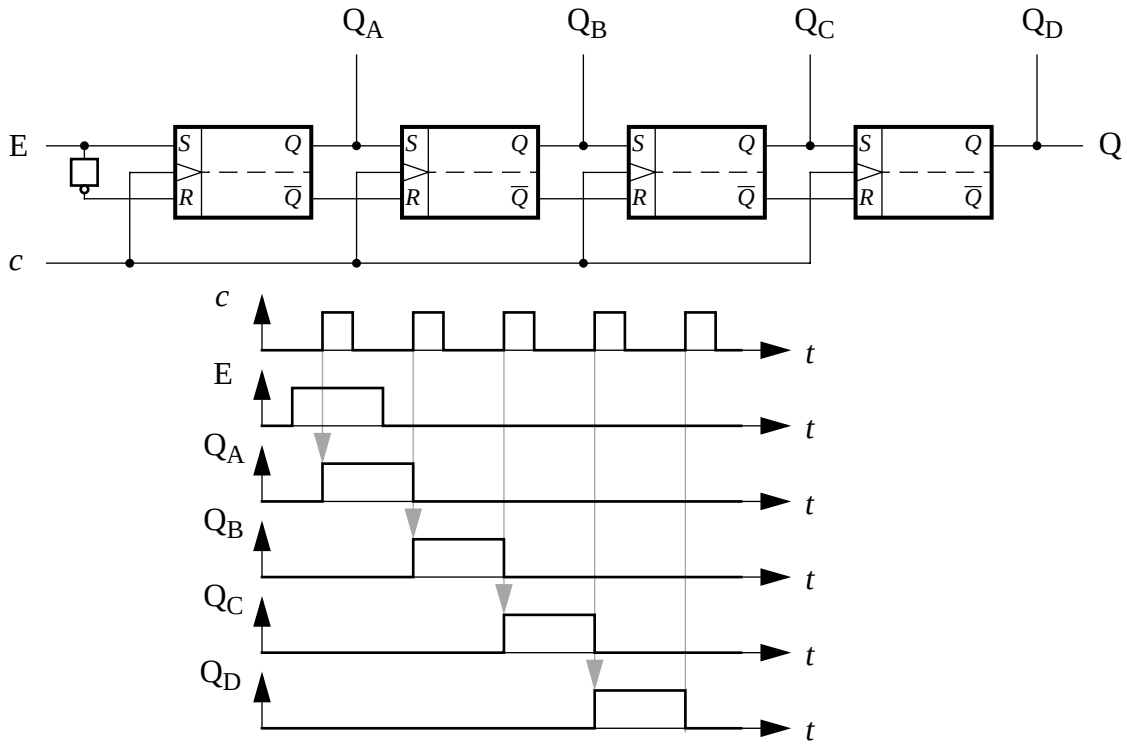


Bild 4.12: Schieberegister mit SR-Flipflops. *c* - Takteingang, vorderflankengetaktet

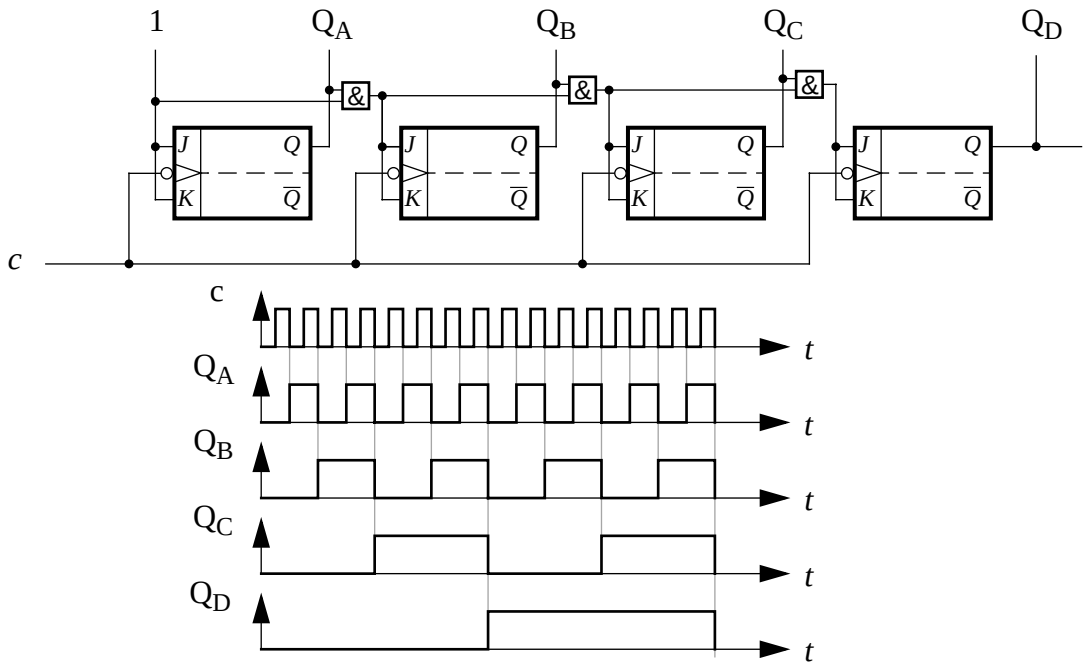


Bild 4.13: Zählerregister mit JK-Flipflops. *c* - Takteingang, rückflankengetaktet

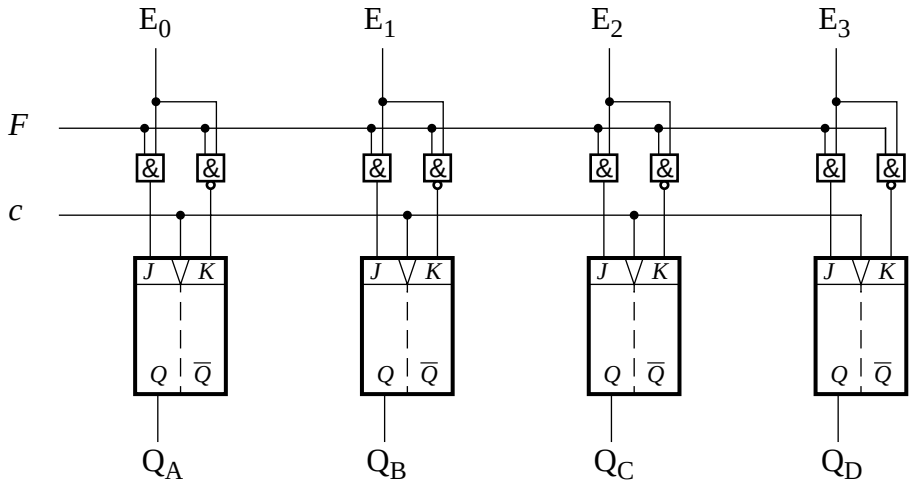


Bild 4.14: Speicherregister mit JK-Flipflops. *c* - Takteingang, vorderflankengetaktet; *F* - Freigabe

4.3.3 Speichermatrizen („RAM“)

Die Technologie der Mikroelektronik erlaubt es, Speicherzellen in extrem großer Anzahl auf einem gemeinsamen Silizium-Chip zu integrieren: Stand der Technik sind mehrere Millionen Bits pro Chip. Die Speicherzellen sind geometrisch in Zeilen und Spalten angeordnet. Eine Zeile von Speicherzellen kann z.B. ein Wort enthalten, dessen einzelne Bits sich in unterschiedlichen Spalten befinden. Man bezeichnet daher die Zeilen der Speichermatrix auch als „Wortrichtung”, die Spalten als „Bitrichtung”. Selektiert man gleichzeitig eine Zeile und eine Spalte der Speichermatrix, so erhält man den direkten Zugriff auf die Speicherzelle am Schnittpunkt. Dies wird als wahlfreier Zugriff bezeichnet („Random Access Memory”, RAM).

Die Selektion einer Speicherzelle auf einem höchstintegrierten Speicherchip kann nicht direkt erfolgen. Eine Speichermatrix der Größe

2^n Zeilen $\cdot\ 2^m$ Spalten $= 2^{n+m}$ Beispiel: $2^{10+10} = 2^{20} = 1$ MegaBit,

die also etwa einer Million Speicherzellen enthält, würde sonst

2^n Zeilen $+\ 2^m$ Spalten $= 2^n + 2^m$ $2^{10} + 2^{10} = 1.024 + 1.024 = \underline{2.048}$ (!)

äußere Anschlüsse benötigen, was technologisch nicht machbar ist. Codiert man jedoch die Zeilen- und Spaltenadressen („Wort- bzw. Bitadresse") im Dualcode, so gilt für die Anzahl der äußeren Anschlüsse:

$ld\ 2^n \quad + ld\ 2^m \quad = n + m$ $10 + 10 = \underline{20}$.

Zur Selektion genau einer Wort- bzw. einer Bitleitung müssen die dual codierten Wort- bzw. Bitadressen in einen „1-aus-*n*"-Code umgewandelt werden, bei dem definitionsgemäß immer nur ein Bit aktiv ist. Daher enthalten höchstintegrierte Speicherchips am Rande der eigentlichen Speichermatrix stets auch Decoderschaltungen, um die Anzahl der äußeren Anschlüsse des Chips in der geschilderten Weise niedrig zu halten, wie das nächste Bild zeigt.

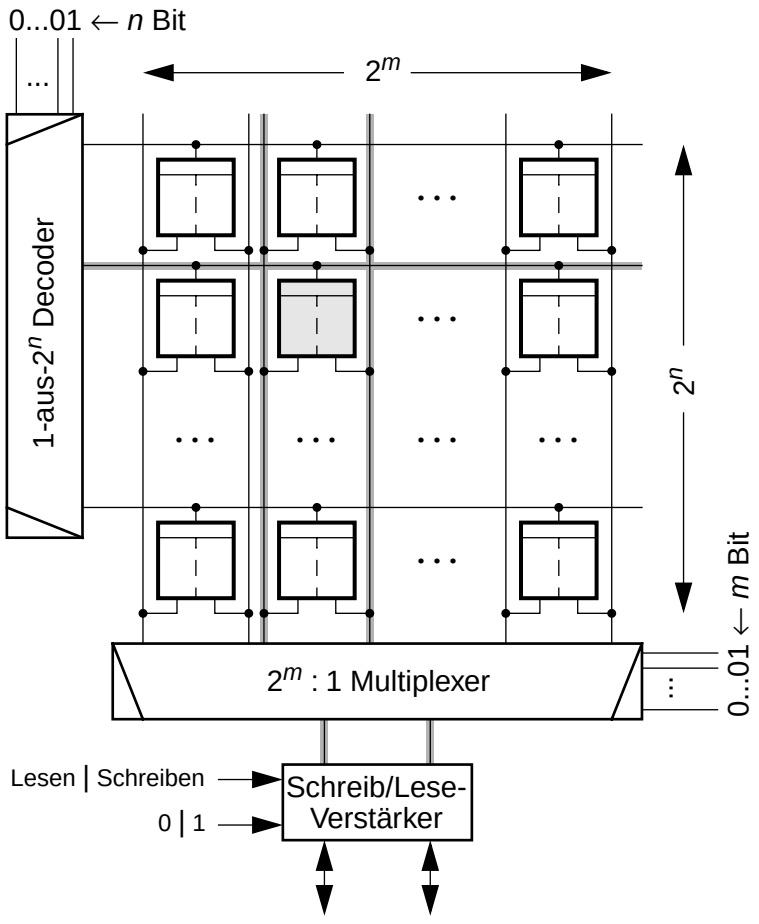


Bild 4.15: Organisation eines integrierten Speicherchips („Random Access Memory", RAM)

4.4 Verarbeitungseinheiten

4.4.1 Arithmetisch/Logische Einheit („ALU“)

Rechenwerke werden aufgrund ihrer Verknüpfungsfunktionen präziser als arithmetisch/logische Einheiten bezeichnet, engl. „Arithmetic/Logic Unit“, und daher im folgenden kurz „ALU“ genannt.

Vorgänge der technischen Informationsverarbeitung können durch informationsverarbeitende Systeme ausgeführt werden. Programmieren bedeutet so betrachtet die Abbildung von „Prozessen“ auf „Prozessoren“. Dazu muss man den strukturellen Aufbau eines konkreten Prozessors nicht in allen Einzelheiten kennen; es genügt, wenn man die funktionellen Zusammenhänge durch ein abstraktes Prozessormodell darstellt. Folgende Möglichkeiten können zur Modelldarstellung dienen:

- problem-orientierte Programmiersprachen
- maschinen-orientierte Programmiersprachen
- binäre Maschinenbefehle („Maschinen-Code“)

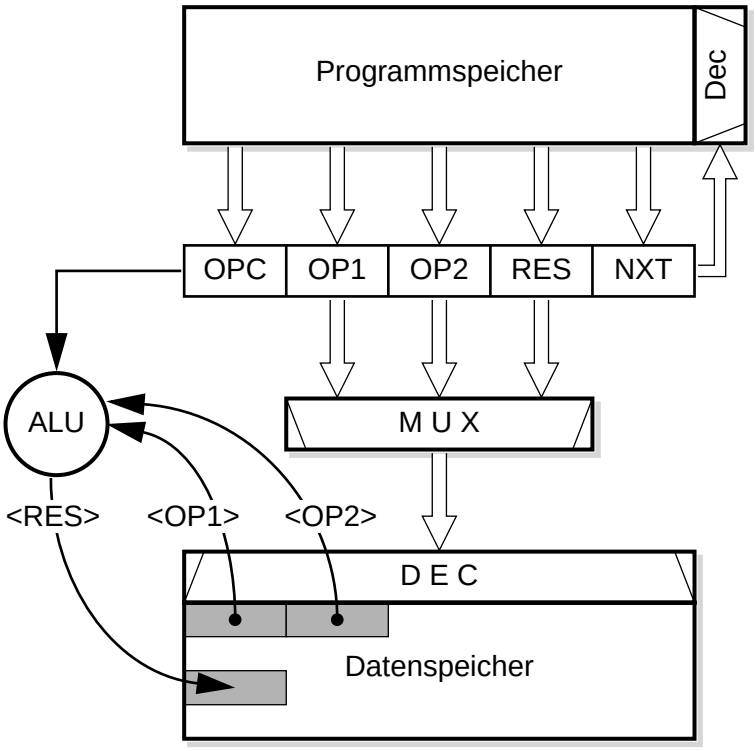


Bild 4.16: Strukturelle Darstellung eines Prozessormodells;
ALU - Arithmetic/Logic Unit (Rechenwerk); weitere Erläuterungen im folgenden Text.

Das obige Bild stellt ein *strukturelles* Prozessormodell dar. Aus dem Programmspeicher wird ein Maschinenbefehl ausgelesen:

- Das Feld OPC gibt die auszuführende arithmetische oder logische Operation in binär codierter Form an („Operationscode“).

- Die ebenfalls binär codierten Adressen OP1 und OP2 der beiden Operanden sowie die Adresse RES für das Resultat werden über einen Multiplexer nacheinander an den Decoder des Datenspeichers angelegt.
- Dadurch gelangen die aktuellen Werte <OP1> und <OP2> der Operanden zum Rechenwerk ALU, werden dort nach Maßgabe des Operationscodes OPC miteinander verknüpft und der Wert <RES> des Resultats wird an der Adresse RES im Datenspeicher abgespeichert.
- Das Adressfeld NXT bestimmt den als nächsten auszuführenden Befehl und wird deshalb an den Adressdecoder des Programmspeichers angelegt.

Es ist wichtig, zwischen dem aktuellen „Wert“ eines Operanden und seiner „Adresse“ im Speicher zu unterscheiden. Ein Operand wird durch seinen Namen eindeutig identifiziert und er hat einen ganz bestimmten Wert. Der Name des Operanden ist äquivalent zur Adresse einer im Hauptspeicher konkret vorhandenen Speicherstelle, sein Wert ist gleichbedeutend mit deren Inhalt:

Wert := < Name >; programmtechnische Betrachtung

Inhalt := < Adresse >; speichertechnische Betrachtung

Da der Name eines Operanden innerhalb eines Programms derselbe bleibt, sein Wert aber laufend geändert werden kann, spricht man auch von einer „Variablen“.

Die strukturell geprägte Darstellung lässt sich funktionell sehr viel kürzer und formaler fassen; insbesondere auf der Register-Transfer-Ebene genügt ein *funktionelles* Prozessormodell. Da jedem Maschinenbefehl eine Operation der ALU entspricht, lässt sich der Prozessor durch seinen Maschinenbefehlssatz vollständig beschreiben.

a) Vier-Adress-Befehlsformat

Wie im obigen Bild gezeigt enthält das aufwendigste Befehlsformat vier Speicheradressen:

OPC	OP1	OP2	RES	NXT

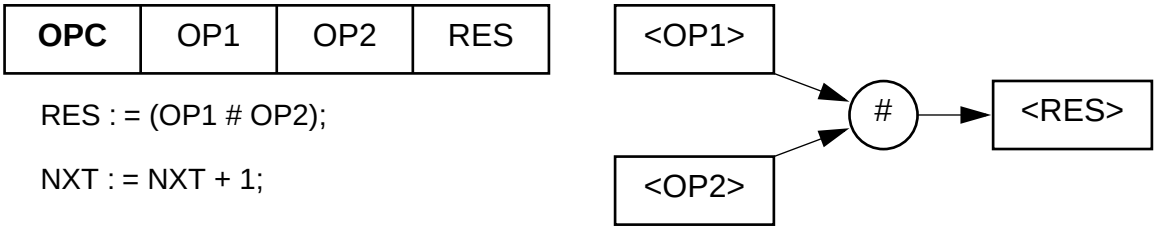
Derselbe Sachverhalt kann in einer „Programmiersprache“ wie folgt notiert werden:

RES := (OP1 # OP2);

Das Feld OPC ↔ # gibt eine beliebige arithmetisch/logische Operation an („Operationscode“).

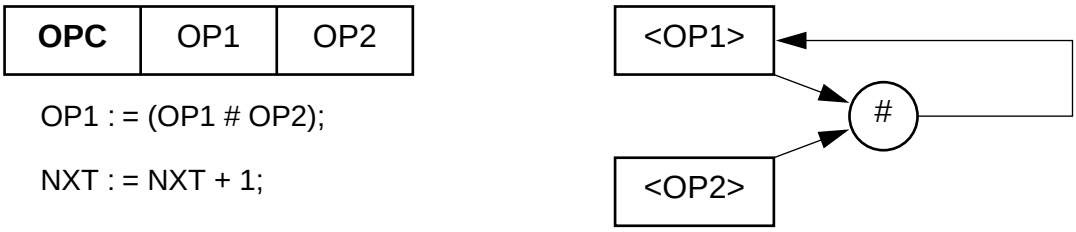
b) Drei-Adress-Befehlsformat

Da die Maschinenbefehle, aus denen ein Algorithmus besteht, aus Gründen der Übersichtlichkeit meist direkt nacheinander im Programmspeicher untergebracht werden, können die aufeinanderfolgenden Befehlsadressen nach dem „von Neumannsches Prinzip“ durch einen Befehlszähler erzeugt werden. (Soll die Sequenz unterbrochen werden, so ist ein Sprungbefehl einzufügen, der den Inhalt des Befehlszählers explizit setzt.) Das Feld NXT kann dann eingespart werden und man erhält ein Drei-Adress-Befehlsformat:



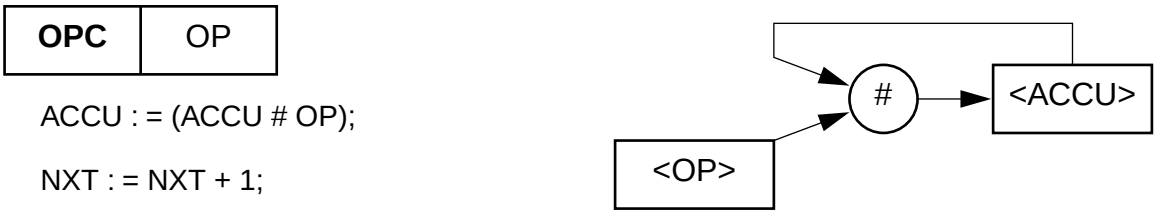
c) Zwei-Adress-Befehlsformat

Wird einer der Operanden nach seiner Verknüpfung nicht mehr benötigt, so kann man seine Speicherstelle für das Resultat verwenden. Meist wird der 1. Operand überschrieben, so dass man die Resultatadresse weglassen kann; man erhält das Zwei-Adress-Befehlsformat:



d) Ein-Adress-Befehlsformat

Schließlich ist es auch gebräuchlich, ein zusätzliches Register einzuführen, das das Resultat aufnimmt („Accumulator“); es dient gleichzeitig als 1. Operand für die nachfolgende Verknüpfung. Da dessen Adresse nicht angegeben zu werden braucht, da er sich eindeutig im Accumulator befindet, erhält man das Ein-Adress-Befehlsformat:



e) Zwei-Adress-Befehlsformat mit Registerspeicher

Stand der Technik ist, dass man nicht nur ein einzelnes Accumulatorregister einführt, sondern einen größeren Registerspeicher, was zur Folge hat, dass man eine zusätzliche Registeradresse benötigt. Man erhält wieder ein Zwei-Adress-Befehlsformat, das besonders häufig verwendet wird. Die beiden Operandenadressen adressieren wahlweise entweder den Datenbereich im Hauptspeicher oder einzelne Register im Registerspeicher.

4.5 Mikroarchitektur-Synthese

a) Verhaltensbeschreibung (“Specification”)

Synchrone Digitalsysteme werden mit einem einfachen Modell beschrieben, das „Steuerschritte“ als kleinste Zeiteinheit verwendet, die den „Zustandsübergängen“ eines Automaten mit einer endlichen Anzahl diskreter Zustände entsprechen. Die Verhaltensbeschreibung wird in eine äquivalente, unter Randbedingungen optimierte Verhaltensbeschreibung transformiert. Randbedingungen sind z.B. die Chip-Fläche, der Energieverbrauch und die Schaltgeschwindigkeit.

b) Ablaufplanung (“Scheduling”)

Die Ablaufplanung von Steuerungsfolgen(“Scheduling”) optimiert die Anzahl der benötigten Steuerschritte, d.h. die Anzahl der Zustände des steuernden Automaten, unter Berücksichtigung der verfügbaren Hardware und der Zykluszeit:

- Zuweisung von Operationen in der Verhaltensbeschreibung zu Steuerschritten
- Mehrfachausnutzung von Betriebsmitteln durch zeitlich gestaffelte Zuweisung

c) Bereitstellung (“Allocation”)

Die Bereitstellung von Betriebsmitteln (“Allocation of Resources“) optimiert den Aufwand der benötigten Hardware unter Berücksichtigung der gegebenen Zeiteinteilung:

- Verarbeitungseinheiten: Prozessoren, Addierer
- Speicherelemente: Register, Speicherplatz
- Verbindungen: Multiplexer, Busse, Demultiplexer

d) Zuweisung (“Assignment”)

Zuweisung von Funktionen zu Funktionseinheiten:

- Prozessorzuweisung zu Operationen
- Registerzuweisung zu Variablen
- Verbindungszuweisung zu Prozessor-Register-Paaren

e) Baustein-Auswahl (“Component Selection”)

zum Beispiel:

- in einem zeitkritischen Pfad: Carry-look-ahead Adder
- in zeitunkritischen Pfaden: Ripple-carry Adder

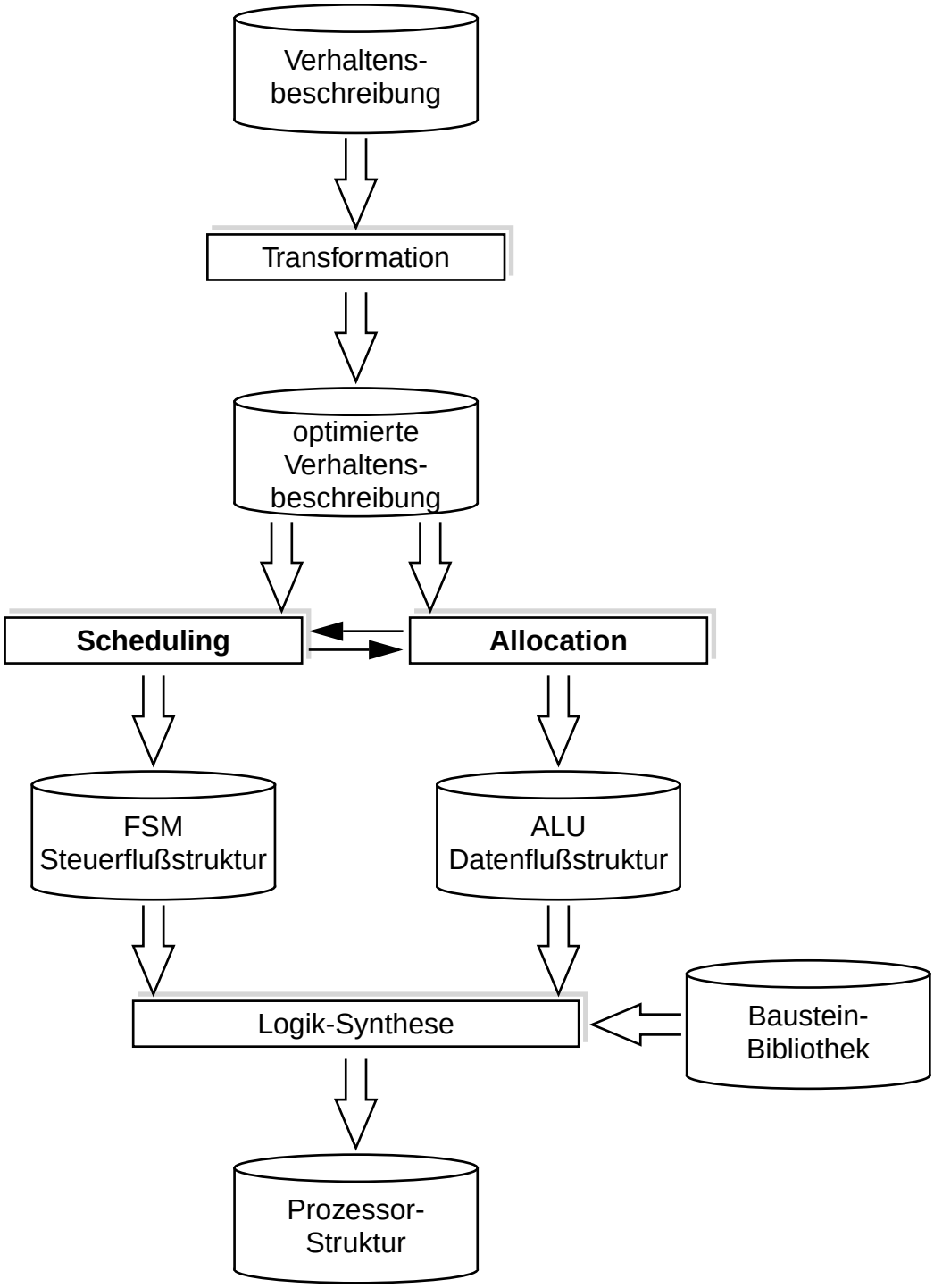


Bild 4.17: Ablauf der Synthese eines Digitalsystems aus seiner Verhaltensbeschreibung

4.6 Pipelining

4.6.1 Befehlspipeline

- Eine Kette von Verarbeitungsstufen, die wie ein Fließband arbeiten, nennt man eine *Pipeline*. Die auszuführenden Befehle werden entlang der Stufen verarbeitet.
- Zwischen den Pipelinestufen werden *Pufferspeicher* benötigt.
- *Pipelining* ist eine Implementierungsmethode, bei der mehrere Befehle überlappt abgearbeitet werden. Es nutzt die Parallelität zwischen den Befehlen in einem sequentiellen Befehlsstrom.
- RISC-Maschinen *(Reduced Instruction Set Computer)* benötigen zum Transport eines Befehls um einen Schritt in der Pipeline *einen* Taktzyklus; in jedem Zyklus wird ein neuer Befehl geholt.
- Pipelining reduziert die mittlere Ausführungszeit pro Befehl, d.h. es erhöht die pro Zeiteinheit beendete Anzahl der Befehle, den *Durchsatz*, aber nicht die Ausführungszeiten der einzelnen Befehle.
- Ziel des Pipeline-Entwurfs ist, eine Balance der Länge der Pipelinestufen zu erreichen. Dann gilt für die *mittlere Ausführungszeit pro Befehl* für eine mit Pipeline implementierte Maschine: „Befehlsausführungszeit der Maschine ohne Pipeline : Anzahl der Pipelinestufen“.

4.6.2 Befehlsausführung

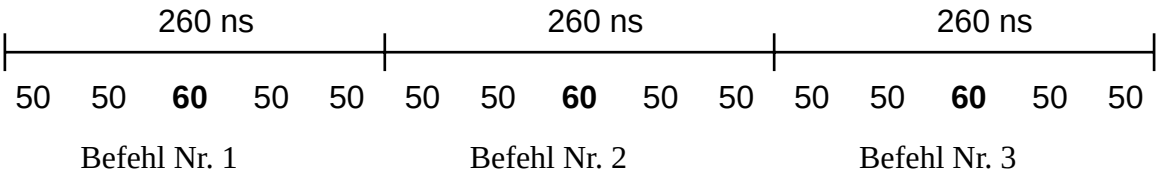
Ein digitaler Universalrechner verfüge über eine Lade/Speicher-Architektur und einen Registerspeicher; er bearbeite die Maschinenbefehle seines Befehlssatzes in folgenden Schritten:

1. Befehl holen *(Instruction Fetch)*, **IF**
   d.h. Hauptspeicher mit Befehlszählerinhalt adressieren und Befehlszähler erhöhen.
2. Befehl decodieren *(Instruction Decode)* **ID**
   und Quellregisterinhalte in A- bzw. B-Register holen.
3. Befehl ausführen *(Execute)* **EX**
   a) Lade-/Speicher-Befehl: effektive Hauptspeicheradresse berechnen und Datenregister laden;
   b) Verzweigungsbefehl: Zieladresse berechnen und Verzweigungsbedingung setzen;
   c) ALU-Befehl: Berechnung ausführen.
4. Speicherzugriff *(Memory Access)* **MEM**
   a) Lade-Befehl: Daten vom Hauptspeicher ins Datenregister laden; Speicher-Befehl: in umgekehrter Richtung speichern;
   b) Verzweigungsbefehl: Zieladresse in Befehlszähler laden, falls Verzweigungsbedingung erfüllt;
   c) ALU-Befehl: (nichts)
5. Rückschreiben *(Write Back)* **WB**
   a) Lade-Befehl: Datenregisterinhalt ins Zielregister laden; Speicher-Befehl: (nichts);
   b) Verzweigungsbefehl: (nichts);
   c) ALU-Befehl: Ergebnis ins Zielregister schreiben.

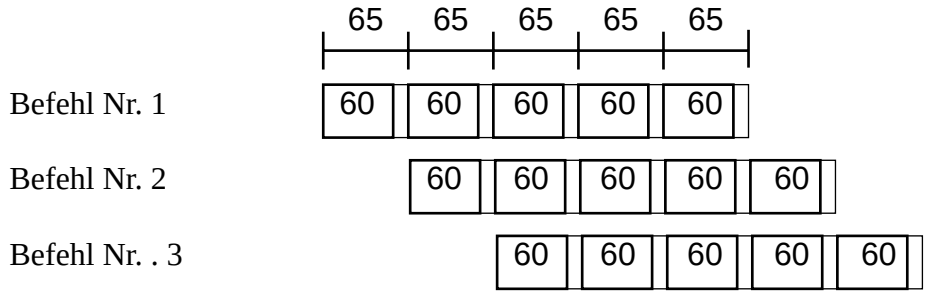
4.6.3 Mehrstufige Pipeline

Nr. des Befehls	Takt Nr. 1	2	3	4	5	6	7	8	9
Befehl *i*	IF	ID	**EX**	MEM	WB				
Befehl *i* + 1		IF	ID	**EX**	MEM	WB			
Befehl *i* + 2			IF	ID	**EX**	MEM	WB		
Befehl *i* + 3				IF	ID	**EX**	MEM	WB	
Befehl *i* + 4					IF	ID	**EX**	MEM	WB

sequentielle Ausführung ohne Pipeline



überlappte Ausführung mit Pipeline



4.7 Entwurf eines Datenpfades

a) Entwurfsziele

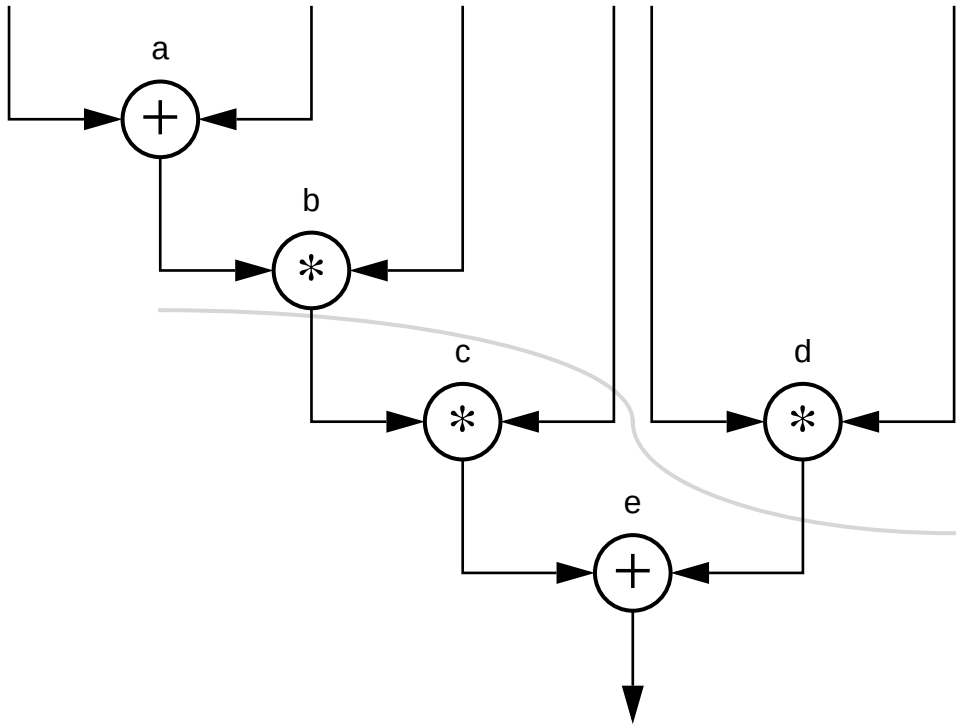
Erhöhung der Verarbeitungsleistung

- durch Pipelining

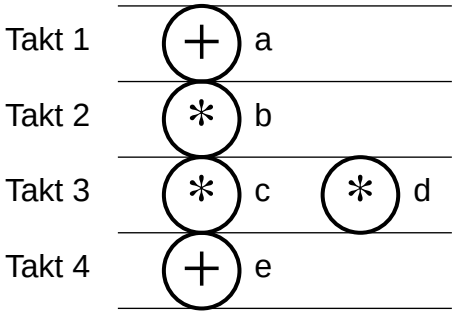
Senkung der Kosten

- durch Mehrfachausnutzung von Betriebsmitteln
- durch Baustein-Auswahl

b) Datenflussgraph (DFG)



c) Ablaufplanung unter zeitlichen Randbedingungen



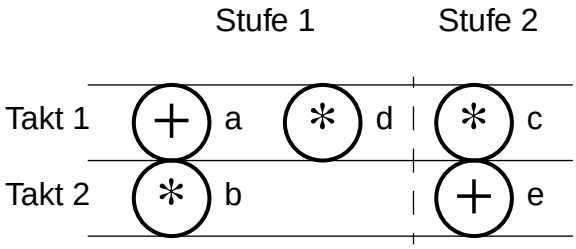


Bild 4.18: Ablaufplanung (“Scheduling”) Ablaufplanung (“Scheduling”) einer Pipeline

Funktion	Name	Gatter	Verzög.
*	Mpy1	100	50 ns
*	Mpy2	150	30 ns
*	Mpy4	200	20 ns
+	Add1	30	40 ns
+	Add2	50	20 ns
+	Add3	100	10 ns

Bild 4.19: Baustein-Bibliothek (“Component Library”)

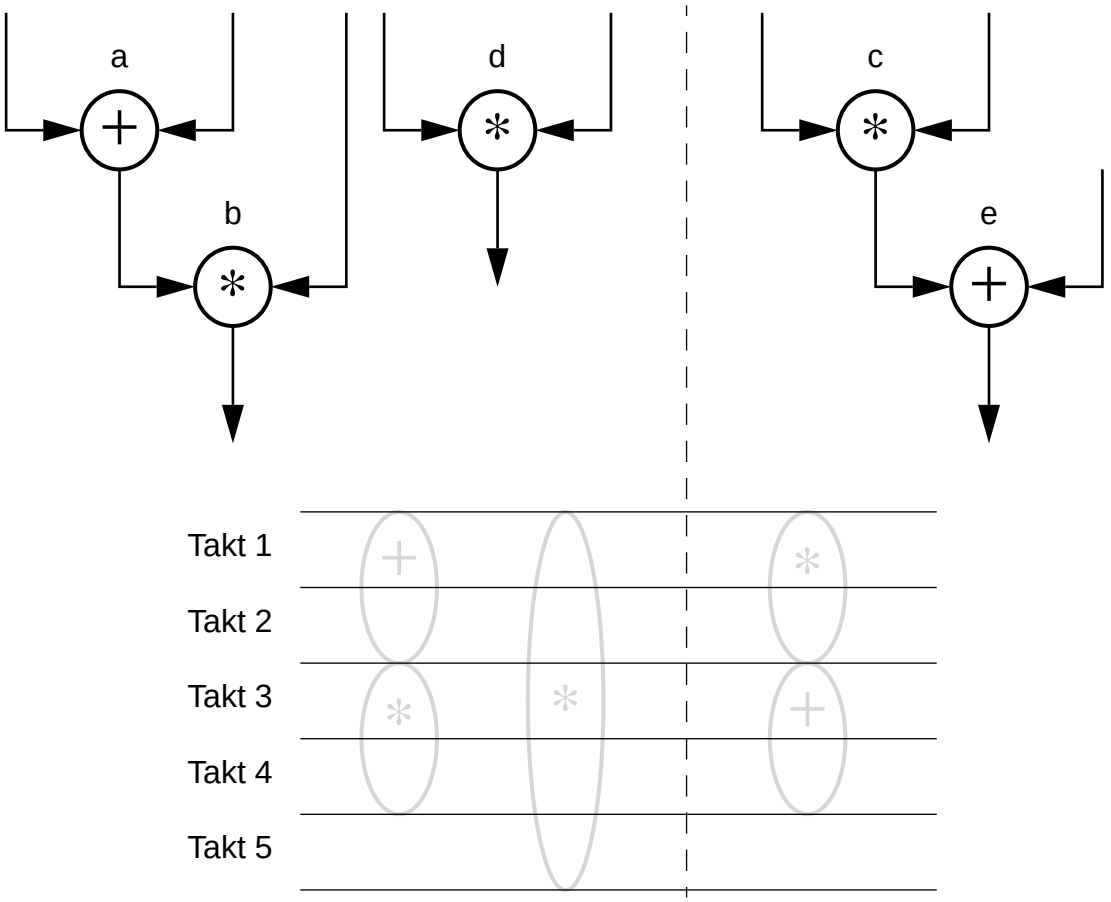
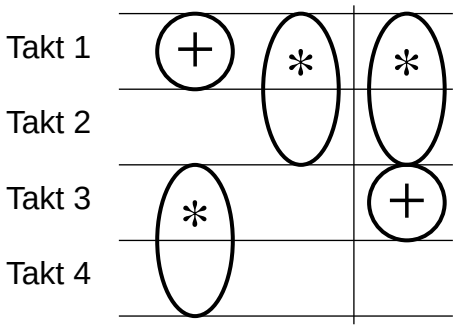


Bild 4.20: Ablaufplanung einer Pipeline mit Baustein-Auswahl aus obiger Bibliothek; zeitliche Randbedingung: maximal 5 Takte zu je 10 ns.

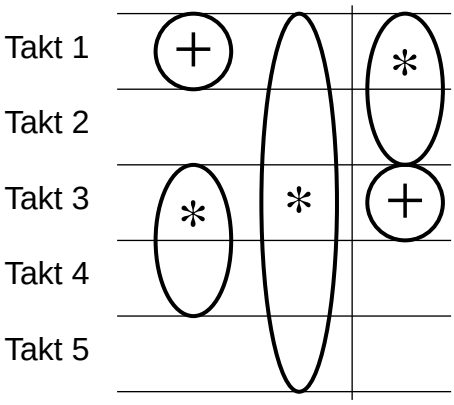
d) Ursprüngliche und endgültige Zuteilung



Addierer:

Multiplizierer:

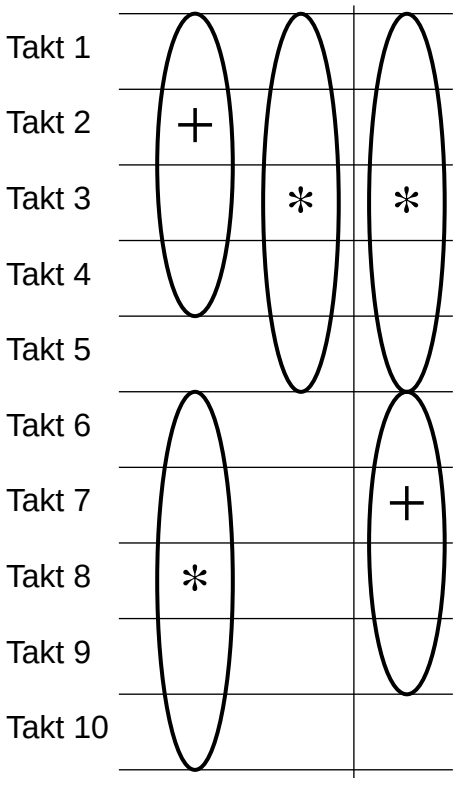
Kosten: 500 Gatter



Addierer:

Multiplizierer:

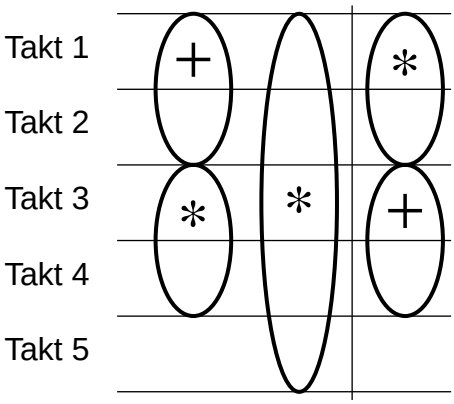
Kosten: 400 Gatter



Addierer:

Multiplizierer:

Kosten: 230 Gatter



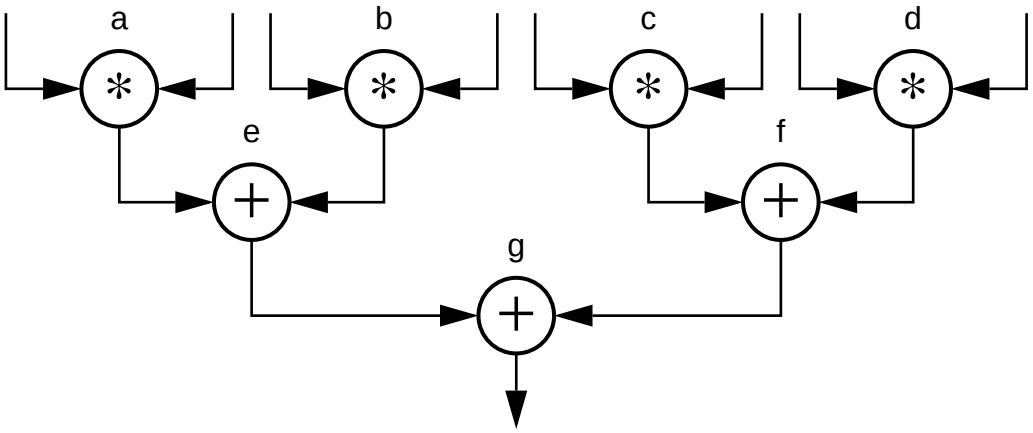
Addierer:

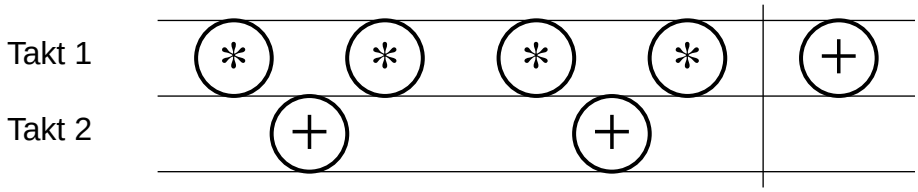
Multiplizierer:

Kosten: 350 Gatter

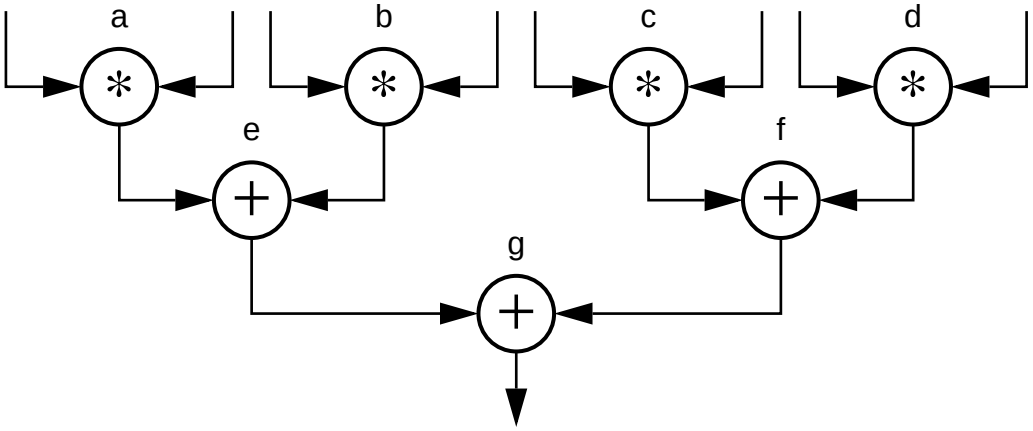
e) Modifikation der Pipeline

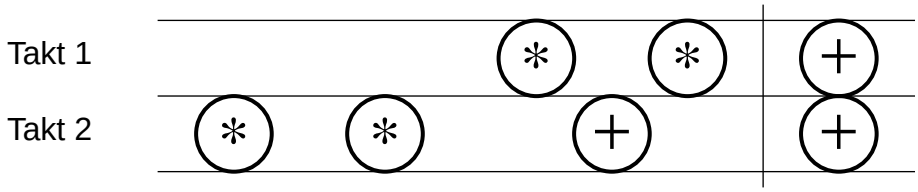
Pipeline 1





Pipeline 2





5 Algorithmische Ebene

Prof. Dr.-Ing. U.G. Baitinger

5.1. Der Begriff des Algorithmus

Ein „Algorithmus“ ist eine formale Vorschrift für einen Vorgang zur Verarbeitung von Informationen nach *Al-Khorezmi*. Ein „Algorithmus im engeren Sinn“ ist eine Rechenvorschrift, zum Beispiel die Verknüpfung von Zahlenwerten nach *Adam Riese*. Ein „Algorithmus im weiteren Sinn“ ist eine formale Vorschrift für einen Handlungsablauf, zum Beispiel ein Kochrezept nach *Art des Hauses*.

5.2. Hardware-Beschreibungssprachen

HDL: „**H**ardware **D**escription **L**anguages“ dienen zur formalen Beschreibung des *Verhaltens* digitaler Schaltungen, zunächst unabhängig von deren Struktur. Sie werden in frühen Phasen des zergliedernden Entwurfs verwendet. Anwendungsgebiete sind Simulation, Synthese und Verifikation.

VHDL: Die „**V**ery High-Speed Integrated Circuit **HDL**“ ist eine Hardware-Beschreibungssprache für höchstintegrierte Digitalsysteme hoher Geschwindigkeit mit ADA ähnlicher Syntax. VHDL unterstützt vor allem die Architektur-, die Algorithmische und die Registertransfer-Ebene.

EDIF: Das „**E**lectronic **D**esign **I**nterchange **F**ormat“ ist ein Datenaustauschformat mit LISP ähnlicher Syntax. EDIF unterstützt vor allem die Registertransfer-, die Logik- und die Schaltkreisebene.

5.3. Ein Entwurfsbeispiel

Die Verkehrsampeln an der Kreuzung zwischen einer Hauptstraße und einer Querstraße sind verkehrsabhängig zu steuern. Die Steuerung der Verkehrsampeln soll durch einen Automaten mit einer endlichen Anzahl diskreter Zustände erfolgen. Sensoren an beiden Einmündungen der Querstraße stellen die Anwesenheit von Fahrzeugen auf der Querstraße fest.

- Das Ampelpaar an der Hauptstraße soll nur dann auf ROT schalten, wenn auf der Querstraße ein Fahrzeug festgestellt wird.
- Das Ampelpaar für die Querstraße soll so lange GRÜN zeigen wie sich Fahrzeuge auf der Querstraße befinden, aber nicht länger als eine vorgegebene Zeitspanne.
- Wenn das Ampelpaar an der Hauptstraße wieder auf GRÜN schaltet, dann soll es mindestens während einer vorgegebenen Zeitspanne so bleiben.

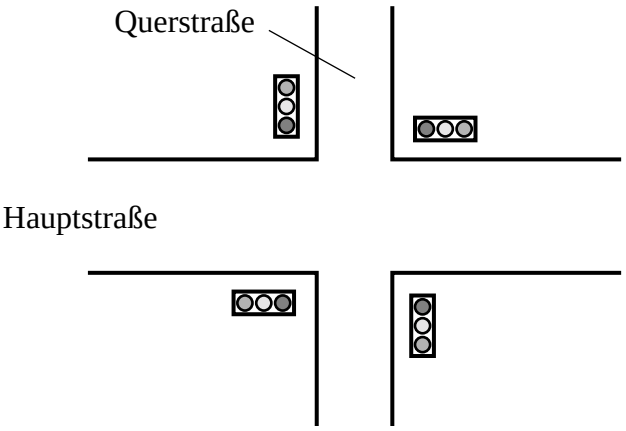


Bild 5.1: Verkehrsampeln an einer Straßenkreuzung

5.4. Der Entwurfsablauf

1. Deklaration der Datentypen:
   - diskrete Zustände des Systems festlegen;
   - Ausgabewerte festlegen, hier: Farben einer Verkehrsampel aufzählen.
2. Deklaration der Schnittstelle:
   - Eingabevariablen des Systems festlegen,
     hier: Sensoren stellen ein Fahrzeug auf der Querstraße fest;
   - Ausgabevariablen des Systems festlegen,
     hier: Ansteuerung der Ampelpaare an Hauptstraße und Querstraße.
3. Erstellung des Verhaltensmodells:
   - Ablaufgraph und Ablauftabelle des Systems;
   - Verhaltensbeschreibung des Systems in einer Hardware-Beschreibungssprache.
4. Festlegung eines Testverfahrens.
5. Simulation des Verhaltensmodells.

5.4.1 Deklaration der Datentypen

a) Verbale Spezifikation

Die Steuerung der Verkehrsampeln soll durch einen Automaten erfolgen, der eine endliche Anzahl diskreter Zustände besitzt, die das System einnehmen kann:

1. Im Ruhezustand zeigt das Ampelpaar an der Hauptstraße GRÜN, das für die Querstraße ROT.
2. Im nächsten Zustand zeigt das Ampelpaar an der Hauptstraße GELB, das für die Querstraße weiterhin ROT.
3. Im anschließenden Zustand zeigt das Ampelpaar an der Querstraße GRÜN, das für die Hauptstraße ROT.
4. Im letzten Zustand zeigt das Ampelpaar für die Querstraße GELB, das für die Hauptstraße weiterhin ROT.

Dann kehrt das System in den Ruhezustand zurück, d.h. die Systemzustände können als geschlossene Zählkette definiert werden.

b) Formale Spezifikation

Im ersten Abschnitt der formalen Spezifikation sind die Datentypen festzulegen, d.h. zu „deklarieren“. Hier sind es einfache Aufzählungen der Ampelfarben und Systemzustände.

```
-- Verkehrsampelsteuerung: Deklaration der Datenypen
package System_Typen is

 type Farbe   is (Gruen, Gelb, Rot, Unbekannt);

 type Zustand is (Hauptstr_Ampel_Gruen, Hauptstr_Ampel_Gelb,
                  Querstr_Ampel_Gruen,  Querstr_Ampel_Gelb);

end System_Typen;
```

5.4.2 Deklaration der Schnittstelle

a) Verbale Spezifikation

Dann ist die Schnittstelle des Systems zu seiner Umgebung zu definieren, d.h. die Eingabevariablen, auf die es reagieren, und die Ausgabevariablen, die es erzeugen soll. Das System verlässt seinen Ruhezustand, sobald die Sensoren ein Fahrzeug auf der Querstraße feststellen, d.h. es wird nur eine binäre Eingabevariable benötigt:

• `Wagen_auf_Querstr` ist entweder `wahr` oder `falsch`.

Das System steuert zwei Ampelpaare, d.h. es werden zwei Ausgabevariablen benötigt, deren aktuelle Werte den jeweils anzuzeigenden Ampelfarben entsprechen:

• `Hauptstr_Ampel` zur Ansteuerung des Ampelpaares an der Hauptstraße;

• `Querstr_Ampel` zur Ansteuerung des Ampelpaares für die Querstraße.

Ferner werden zwei Konstanten benötigt, die dem System vorgegeben werden:

• `Gn_Phase` als *minimale* Zeitspanne, während der das Ampelpaar an der Hauptstraße auf GRÜN bleiben muss, bzw. die *maximale* Zeitspanne, während der das Ampelpaar für die Querstraße auf GRÜN bleiben darf.

• `Gb_Phase` als Zeitspanne, während der ein Ampelpaar GELB zeigt.

b) Formale Spezifikation

```
--Verkehrsampelsteuerung: Deklaration der Ein/Ausgabeschnittstelle
use work.System_Typen.all;
entity Ampel_Steuerung is
 generic ( -- von aussen vorgegeben
  Gn_Phase : Time  -- Gruenphase;
  Gb_Phase : Time  -- Gelbphase);
 port ( -- Eingabe- und Ausgabevariablen
  Wagen_auf_Querstr in Boolean;
  Hauptstr_Ampel    out Farbe;
  Querstr_Ampel     out Farbe);
end Ampel_Steuerung;
```

5.4.3 Erstellung des Verhaltensmodells

Als nächstes ist das Verhalten des zu entwerfenden Systems zu beschreiben. Kann das System wie hier als endlicher, diskreter Automat modelliert werden, so umfaßt sein Verhalten im wesentlichen die systeminternen Vorgänge, die zu Zustandsübergängen führen, wobei nach jedem Zustandsübergang ein Zeitgeber für das Verweilen des Systems im neuen Zustand gestartet wird. Zur Modellierung des Systemverhaltens stehen unterschiedliche Darstellungsmittel zur Verfügung: der Ablaufgraph des Automaten und die dazu isomorphe Ablauftabelle, ferner die Hardware-Beschreibungssprachen, hier VHDL.

Beim vorliegenden Beispiel ist zu beachten, dass die Hauptstraße gegenüber der Querstraße bevorrechtigt ist, d.h. unter welchen Bedingungen das Ampelpaar an der Hauptstraße von GRÜN auf GELB umschaltet und wann das Ampelpaar für die Querstraße dies tun soll. Im ersten Fall müssen zwei Bedingungen erfüllt sein: auf der Querstraße muss ein Fahrzeug erscheinen und die Grünphase an der Hauptstraße muss bereits lange genug gedauert haben. Im zweiten Fall genügt eine Bedingung: entweder sind auf der Querstraße keine Fahrzeuge mehr vorhanden oder die Grünphase für die Querstraße hat bereits lange genug gedauert, obwohl sich noch Fahrzeuge auf ihr befinden.

a) Ablaufgraph

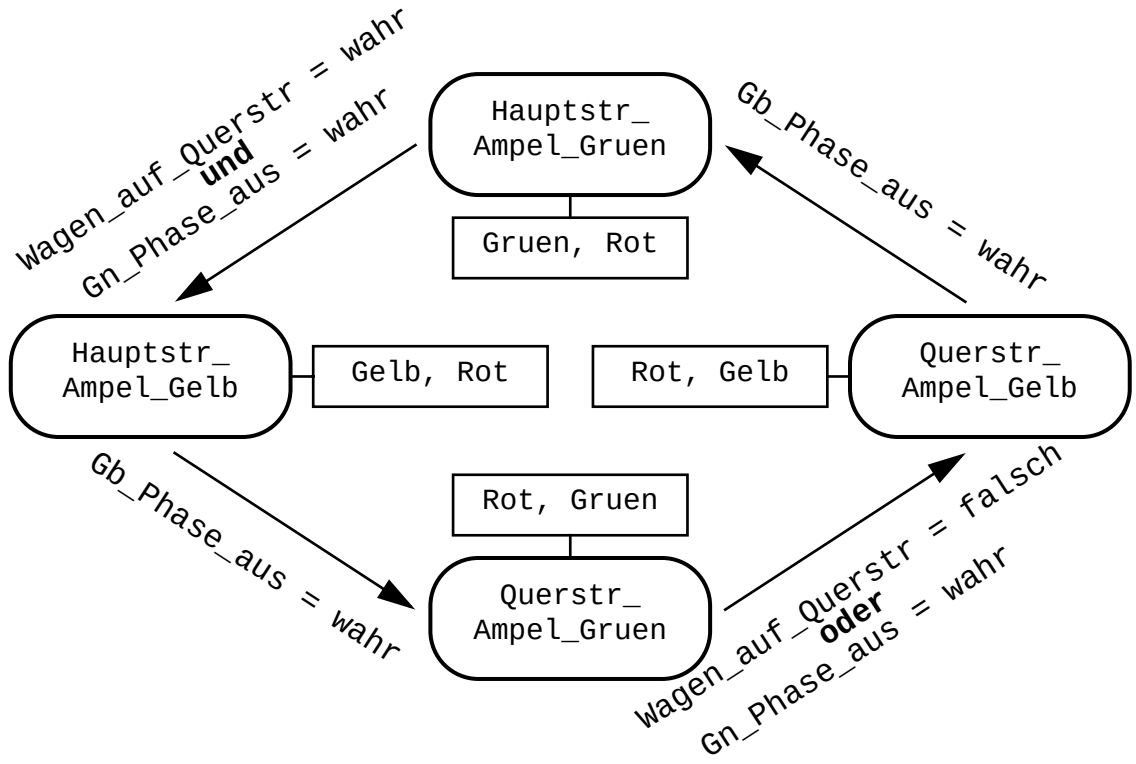


Bild 5.2: Ablaufgraph der Verkehrsampelsteuerung

b) Ablauftabelle

aktueller Zustand	Eingabe	Folge-zustand	Ausgabe Hauptstr_Ampel	Ausgabe Querstr_Ampel
Hauptstr_Ampel_Gruen	Wagen_auf_Querstr = wahr **und** Gn_Phase_aus = wahr	Hauptstr_Ampel_Gelb	Gruen	Rot
Hauptstr_Ampel_Gelb	Gb_Phase_aus = wahr	Querstr_Ampel_Gruen	Gelb	Rot
Querstr_Ampel_Gruen	Wagen_auf_Querstr = falsch **oder** Gn_Phase_aus = wahr	Querstr_Ampel_Gelb	Rot	Gruen
Querstr_Ampel_Gelb	Gb_Phase_aus = wahr	Hauptstr_Ampel_Gruen	Rot	Gelb

Bild 5.3: Ablauftabelle der Verkehrsampelsteuerung

c) Hardware-Beschreibungssprache VHDL

```
-- Verkehrsampelsteuerung: Verhaltensmodell
architecture Verhalten of Ampel_Steuerung is
 -- Initialisierung
 signal Ist_Zustand : Zustand := Hauptstr_Ampel_Gruen;
 signal Zeitgeber :       Boolean := falsch;
 signal Gn_Phase_aus :    Boolean := falsch;
 signal Gb_Phase_aus :    Boolean := falsch;
begin -- nebenlaeufige Prozesse
process begin              -- Automatenverhalten
 case Ist_Zustand is
  when Hauptstr_Ampel_Gruen => -- 1. Zustand
  if Wagen_auf_Querstr and Gn_Phase_aus
   then -- Zustandsuebergang
   Ist_Zustand <= Hauptstr_Ampel_Gelb;
   Zeitgeber <= not Zeitgeber;
  end if;
  when Hauptstr_Ampel_Gelb =>  -- 2. Zustand
  if Gb_Phase_aus
   then -- Zustandsuebergang
   Ist_Zustand <= Querstr_Ampel_Gruen;
   Zeitgeber <= not Zeitgeber;
  end if;
  when Querstr_Ampel_Gruen =>  -- 3. Zustand
  if not Wagen_auf_Querstr or Gn_Phase_aus
   then -- Zustandsuebergang
   Ist_Zustand <= Querstr_Ampel_Gelb;
   Zeitgeber <= not Zeitgeber;
  end if;
  when Querstr_Ampel_Gelb =>   -- 4. Zustand
  if Gb_Phase_aus
   then -- Zustandsuebergang
   Ist_Zustand <= Hauptstr_Ampel_Gruen;
   Zeitgeber <= not Zeitgeber;
  end if;
 end case;
 wait on Wagen_auf_Querstr, Gn_Phase_aus, Gb_Phase_aus;
end process;               -- Automatenverhalten
```

```
process begin                -- Zeitgeber
 Gn_Phase_aus <= falsch, wahr after Gn_Phase; -- Gruenphase
 Gb_Phase_aus <= falsch, wahr after Gb_Phase; -- Gelbphase
 wait on Zeitgeber;
end process;                 -- Zeitgeber

-- Zuordnung von Zustand und Ampelfarbe
-- fuer Hauptstr:
with Ist_Zustand select
 Hauptstr_Ampel <=
  Gruen when Hauptstr_Ampel_Gruen,
  Gelb  when Hauptstr_Ampel_Gelb,
  Rot   when Querstr_Ampel_Gruen or Querstr_Ampel_Gelb;

-- Zuordnung von Zustand und Ampelfarbe
-- fuer Querstr:
with Ist_Zustand select
 Querstr_Ampel <=
  Gruen when Querstr_Ampel_Gruen,
  Gelb  when Querstr_Ampel_Gelb,
  Rot   when Hauptstr_Ampel_Gruen or Hauptstr_Ampel_Gelb;

end Verhalten;
```

5.4.4 Festlegung eines Testverfahrens

Um das Verhalten eines manuell entworfenen Systemmodells zu validieren, wird ein Testverfahren benötigt, das zur Simulation des Modells verwendet wird. Das Testverfahren enthält „Stimuli“ zur Ansteuerung des zu testenden Systems durch eine ausgewählte Folge von Eingabewerten, ferner „Rezeptoren“ zur Aufnahme der Ein- und Ausgabewerte des Systems während der Simulation.

```
-- Verkehrsampelsteuerung: Deklaration des Testverfahrens
entity Tester is end Tester;
-- Modell des Testverfahrens
use work.System_Typen.all;
architecture Test_Verfahren of Tester is
 -- Eingabevariable fuer die Ampelsteuerung:
 signal Wagen_erkannt : Boolean := falsch;
 -- Ausgabe-Erfassung der Ampelsteuerung:
 signal Hauptstr : Farbe := Gruen;
 signal Querstr  : Farbe := Rot;
 -- Schablone fuer den Systemtest
 component Sockel
  generic ( -- vorgegebene Konstanten
   Gn_Phase : Time  -- Gruenphase;
   Gb_Phase : Time  -- Gelbphase);
  port ( -- Eingabe- und Ausgabevariablen
   Wagen_auf_Querstr in  Boolean;
   Hauptstr_Ampel    out Farbe;
   Querstr_Ampel     out Farbe);
 end component;
 -- Instanziierung des Systemtests
 Test_Steuerung : Sockel
  generic map (50 s, 20 s); -- Gruenphase, Gelbphase
  port map (Wagen_erkannt, Hauptstr, Querstr);
 -- Eingabe-Stimuli für die Simulation:
 Wagen_erkannt <= falsch,
  wahr after  10 s, falsch after  30 s,
  wahr after 100 s, falsch after 200 s;
end Test_Verfahren;
```

5.4.5 Simulation des Verhaltensmodells

Schließlich ist das Testverfahren zur Simulation des modellierten Systems zu konfigurieren. Die Konfiguration verbindet die Instanz mit dem Namen

• `Test_Steuerung` vom Typ `Sockel`

mit dem Verhaltensmodell, das simuliert werden soll:

• `Ampel_Steuerung(Verhalten)`.

```
-- Simulation der Verkehrsampelsteuerung
use work.all;
configuration Simulator of Tester is
 for Test_Verfahren
  for Test_Steuerung : Sockel use
   entity work.Ampel_Steuerung(Verhalten);
  end for;
 end for;
end Simulator;
```

Die folgende Tabelle zeigt die Simulationsergebnisse, die auch im weiteren Verlauf des Entwurfsprozesses bis zur Realisierung der Hardware als Referenz verwendet werden können: Falls nach einem der folgenden Entwurfsschritte, die die Systemstruktur verfeinern, die Ergebnisse einer detaillierteren Simulation denjenigen eines vorhergehenden Entwurfsschritts widersprechen, liegt ein Entwurfsfehler bei der Verfeinerung vor.

Zeit (s)	Wagen_erkannt	Hauptstr	Querstr
0	falsch	Gruen	Rot
10	wahr	"	"
30	falsch	"	"
100	wahr	Gelb	"
120	"	Rot	Gruen
170	"	"	Gelb
190	"	Gruen	Rot
200	falsch	"	"

Bild 5.4: Simulationsergebnisse der Verkehrsampelsteuerung

6 Architekturebene

Prof. Dr.-Ing. U.G. Baitinger

6.1 Systemspezifikation

Beim Entwurf digitaler Systeme der Mikroelektronik ist deren globale Funktionsbeschreibung in eine detaillierte Transistorstruktur abzubilden. Bei komplexen Systemen ist das nicht in einem einzigen Entwurfsschritt möglich; deshalb besteht der Entwurf aus einer Folge von Entwurfsschritten, die die Entwurfsdaten zunehmend verfeinern. Die umfassendste Spezifikation, nämlich die des Gesamtsystems, wird auf der Architekturebene erstellt, doch werden detailliertere Spezifikationen auch auf allen weiteren Entwurfsebenen benötigt: von der algorithmischen über die Registertransfer- und die Logik- bis zur Schaltungsebene.

Ein System

- besteht aus einer Anzahl unterschiedlich komplizierter Funktionseinheiten, die in einer mehr oder weniger komplexen Kommunikationsstruktur zusammenarbeiten. In einem Graphen können erstere als „Knoten“, letztere als „Kanten“ dargestellt werden.

Das Systemverhalten

- ergibt sich aus der Gesamtheit aller Teilfunktionen in Verbindung mit deren wechselseitiger Kommunikation. („Das Ganze ist mehr als die Summe seiner Teile.“)

Die Systemspezifikation

- ist die verbale Beschreibung oder die formale Darstellung des Systemverhaltens. Sie ist Ausgangspunkt des Entwurfs und Vergleichsgrundlage für die Entwurfsergebnisse, die grundsätzlich nur im Rahmen der Spezifikation gültig sind. Eine Systemspezifikation umfaßt:
  - ein „Verhaltensmodell“ des zu entwerfenden Systems;
  - seine „Schnittstellendefinition“ zur Umgebung;
  - zusätzliche „Randbedingungen“ zur Lösungsfindung
    wie Größe, Schaltgeschwindigkeit und Energiebedarf.

Der Systementwurf

- ist die schrittweise Umwandlung der abstrakten Systemspezifikation in konkrete Strukturen, wobei die Strukturen immer komplexer, die Funktionseinheiten immer zahlreicher und immer weniger kompliziert werden. Der Entwurfsablauf leistet somit die Abbildung einer abstrakten Spezifikation in eine konkrete Technologie.

Ein Rechnersystem

- kann durch eine „Sprache“ (z.B. seinen Maschinenbefehlssatz) dargestellt werden, sowie einen „Interpretierer“, der diese Sprache versteht.

6.2 Die DLX-Architektur

(siehe folgende Seiten)

Die DLX-Architektur

Die DLX ist ein hypothetischer Universalrechner
mit Lade/Speicher-Architektur

Die DLX ist ein „Reduced Instruction Set Computer“:

- RISC-Architekturen werden einfachen, in Programmen statistisch am häufigsten genutzten Befehlen angepaßt;
- Kompliziertere Funktionen werden mit mehreren einfachen Befehlen simuliert, d.h. in Software nachgebildet.

Die DLX-Architektur berücksichtigt Messungen folgender Befehlssätze:

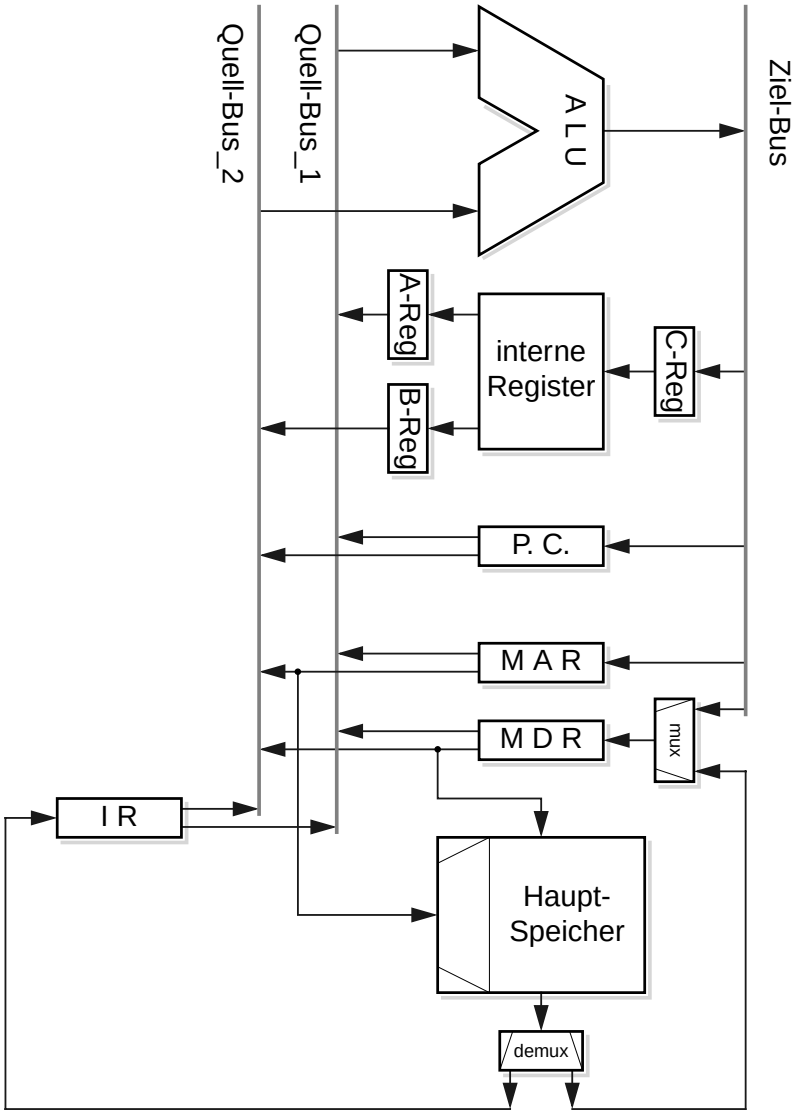
- DEC VAX-Architektur
- IBM /370-Architektur
- Intel 8086-Architektur

Kennzeichen der DLX-Maschine:

- einfacher Lade/Speicher-Befehlssatz
- Universalregistersatz
- Pipeline hoher Effizienz
- einfach zu decodierender Befehlssatz
- eine hoch effiziente Compiler-Technik

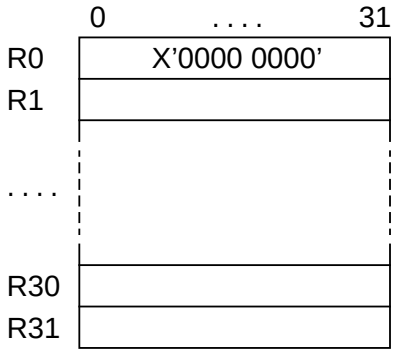
nach D.A.Patterson, J.L. Hennessy: “Computer Architecture, a Quantitative Approach”, Morgan Kaufmann Publ., Inc. (1996)

Rechenwerk der DLX

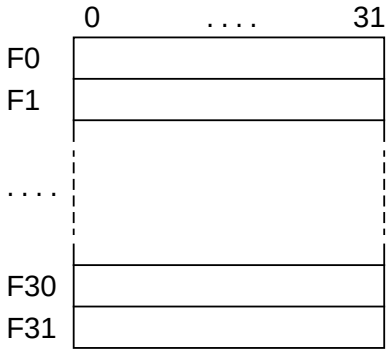


Die Lade/Speicher-Maschine DLX

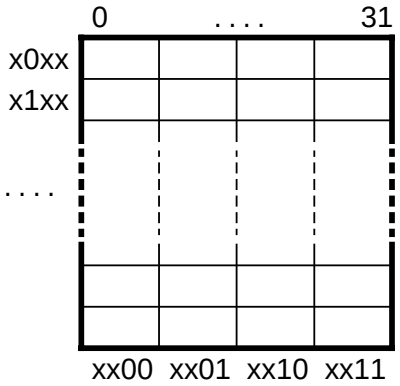
Universalregister
(General Purpose Register)



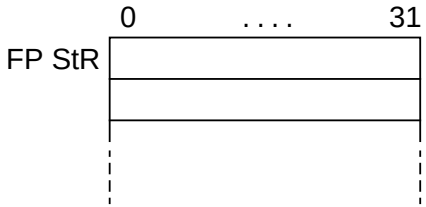
Gleitkommaregister
(Floating Point Register)



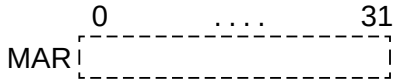
Hauptspeicher
(Main Memory)



spezielle Register
(Status Information)

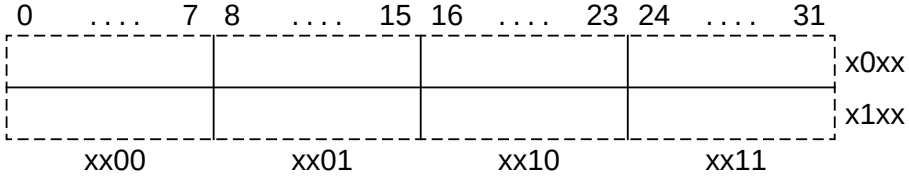


Speicher-Adressregister
(Memory Address Register)



Datenformate der DLX

Der Speicher ist byte-adressiert im „big endian“ Modus

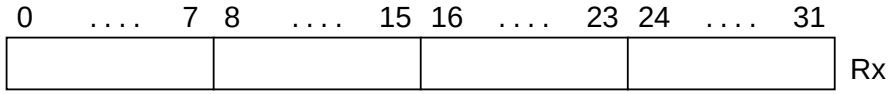


Datenformate, auf die jeweilige Typgrenze ausgerichtet:

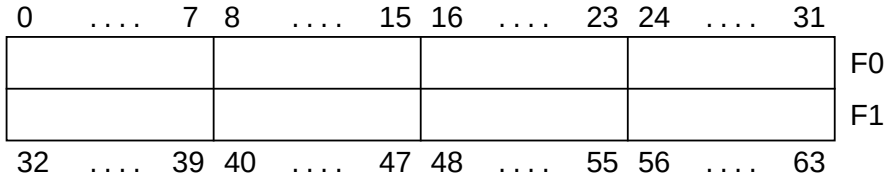
- 8 bit Byte
- 16 bit Halbwort
- 32 bit Wort
- 64 bit Doppelwort

Speicherzugriffe erfolgen mit Lade/Speicher-Befehlen

- zwischen Speicher und Universalregister
  byte-, halbwort- oder wortweise
  - beim Speichern auf Byte- bzw. Halbwortgrenze ausgerichtet
  - beim Laden in den niederwertigen Teil des Registers,
    höherwertiger Teil vorzeichenerweitert oder mit Null aufgefüllt

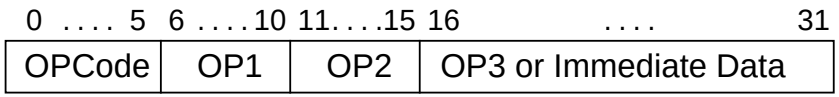


- zwischen Speicher und Gleitkommaregister
  mit einfacher oder doppelter Genauigkeit



Befehlsformate der DLX

Befehlsformat mit drei Operanden-Adressen



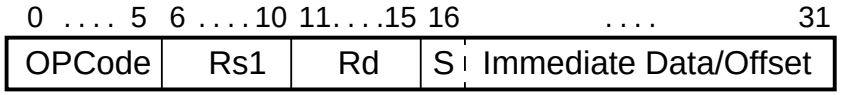
- alle Befehle sind 32 bit lang,
  im Speicher auf Wortgrenze ausgerichtet,
  Fortschaltung des Befehlszählers $PC \leftarrow PC + 4$
- 6 bit primärer Operationscode (OPCode)
  $\rightarrow 2^6 = 64$ verschiedene Befehle

Befehlsklassen

- Lade/Speicher-Operationen
- Arithmetisch/logische Operationen (ALU)
- Verzweigungen (bedingt)
- Sprünge (unbedingt)
- Gleitkomma-Operationen

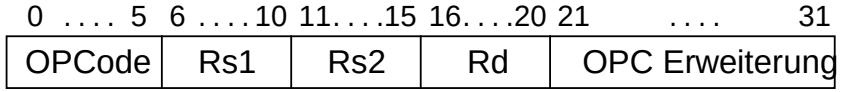
Befehlsformate der DLX

Immediate-Befehlsformat



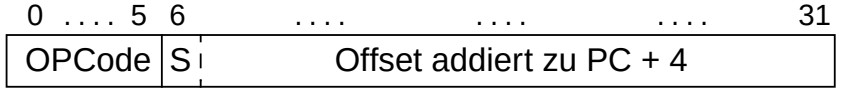
- Laden bzw. Speichern von Registerinhalten
- ALU-Operationen mit Immediate Data
- Verzweigungen (bedingt):
  Branch on Zero, Branch on not Zero
- Sprünge (unbedingt):
  Jump Register, Jump and Link Register

Register-Register-Befehlsformat



- ALU-Operationen mit Registerinhalten
- Transporte zwischen Spezial- und Universalregistern

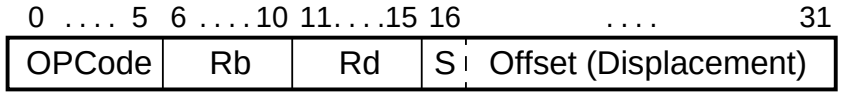
Jump-Befehlsformat



- Sprünge (unbedingt): Jump, Jump and Link
- Übergang zum Betriebssystem: Trap
- Rückkehr zum Anwenderprogramm: RFE

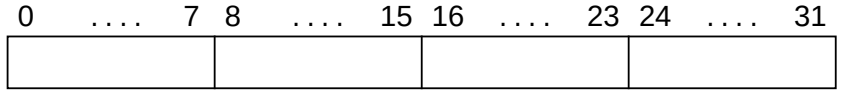
Lade/Speicher-Befehle 1

Immediate-Befehlsformat

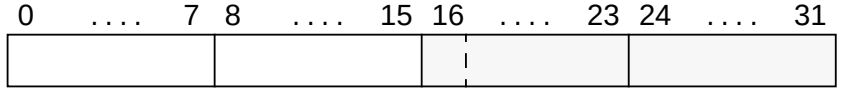


Eine einzige Adressierungsart:

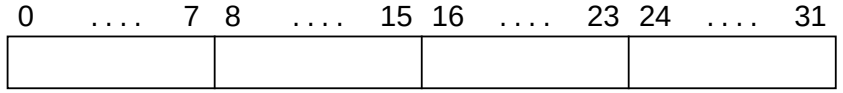
32 bit Basisregister (Rb)



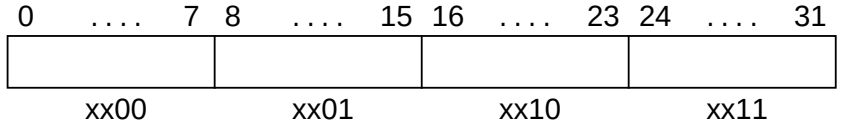
+ 16 bit *Offset (Displacement)* mit Vorzeichenerweiterung



= 32 bit Speicheradresse (MAR)

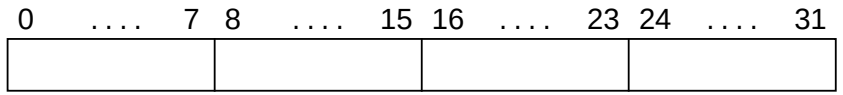


Byte-adressierter Speicher, 2^{32} = 4 GigaByte Adressraum



Laden bzw. Speichern eines Datenregisters (Rd)

- byte-, halbwort- oder wortweise



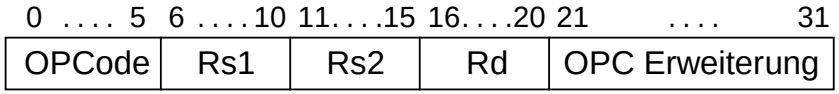
Lade/Speicher-Befehle 2

- alle nutzen eine einzige Adressierungsart
- für alle Datentypen verfügbar
- Werte im Speicher müssen ausgerichtet sein („aligned“)

Befehlsformate		Operationen
LW	R1,30(R2)	R1 ← $_{32}$ M[30+R2]
LW	R1,90(R0)	R1 ← $_{32}$ M[90+0]
LB	R1,40(R3)	R1 ← $_{32}$ $(M[40+R3]_0)^{24}$##M[40+R3]
LBU	R1,40(R3)	R1 ← $_{32}$ 0^{24}##M[40+R3]
LH	R1,40(R3)	R1 ← $_{32}$ $(M[40+R3]_0)^{16}$##M[40+R3]##M[41+R3]
LHU	R1,40(R3)	R1 ← $_{32}$ 0^{16}##M[40+R3]##M[41+R3]
LF	F0,50(R3)	F0 ← $_{32}$ M[50+R3]
LD	F0,50(R2)	F0##F1 ← $_{64}$ M[50+R2]
SW	50(R4),R3	M[50+R4] ← $_{32}$ R3
SF	40(R3),F0	M[40+R3] ← $_{32}$ F0
SD	40(R3),F0	M[40+R3] ← $_{32}$ F0; M[44+R3] ← $_{32}$ F1
SH	52(R2),R3	M[52+R2] ← $_{16}$ $R3_{16..31}$
SB	41(R3),R2	M[41+R3] ← $_{8}$ $R2_{24..31}$

ALU-Befehle 1

Register-Register-Befehlsformat

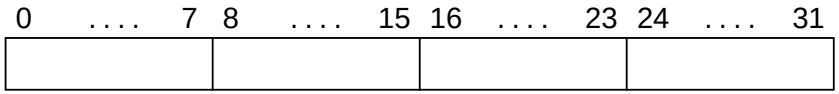


ALU-Operation durch OPCode und Erweiterung definiert:

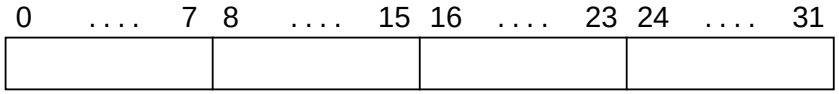
- `ADD, SUB`
- `AND, OR, XOR`
- `SLL, SRL, SRA` (schiebe links bzw. rechts)
- `Sxx` (vergl. & setze Register) `(LT,GT,LE,GE,EQ,NE)`

Rd ← Rs1 ALU Rs2

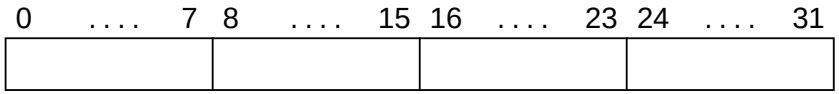
32 bit Quellregister (*source* Rs1)



32 bit Quellregister (*source* Rs2)

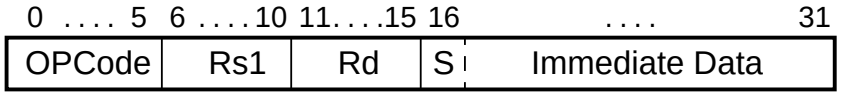


32 bit Zielregister (*destination* Rd)



ALU-Befehle 2

Immediate-Befehlsformat

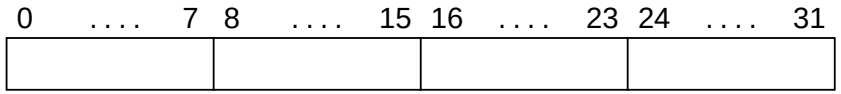


ALU-Operation durch OPCode definiert:

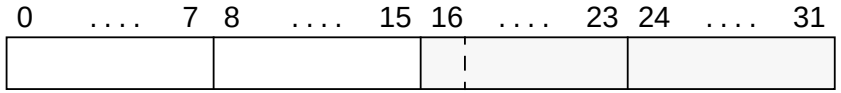
- `ADDI, SUBI`
- `ANDI, ORI, XORI`
- `SLLI, SRLI, SRAI` (schiebe links bzw. rechts)
- `SxxI` (vergl. & setze Register) `(LT,GT,LE,GE,EQ,NE)`

Rd ← Rs1 ALU Immediate

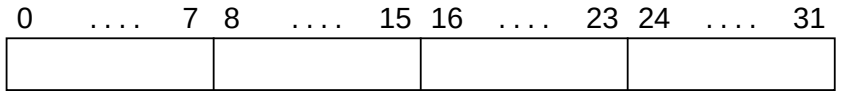
32 bit Quellregister (*source* Rs1)



16 bit *Immediate Data* mit Vorzeichenerweiterung



32 bit Zielregister (*destination* Rd)



ALU-Befehle 3

- mit Immediate Data (Befehlsbits 16 . . 31) oder
- nur mit Registerinhalten (RES^-, $OP1^-$, $OP2^-$Adresse)

Befehle	Befehlsformate	Operationen
Add	`ADD R1,R2,R3`	`R1 ← R2+R3`
Add immediate	`ADDI R1,R2,#3`	`R1 ← R2+3`
Load high immediate	`LHI R1,#42`	`R1 ← 42##0`^{16}
Shift left logical immediate	`SLLI R1,R2,#5`	`R1 ← R2<<5`
Set less than	`SLT R1,R2,R3`	`if (R2<R3) R1← 1 else R1← 0`

Steuerflussbefehle

Die DLX-Architektur enthält

- zwei Verzweigungsbefehle (bedingt): `BEQZ, BNEZ`
- Übergang zum Betriebssystem: `TRAP`
- Rückkehr zum Anwenderprogramm: `RFE`
- vier Sprungbefehle (unbedingt):

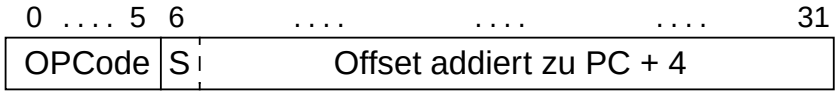
Zieladresse:	not Link	Link
PC-relativ	`J`	`JAL`
Registerinhalt	`JR`	`JALR`

Link: Speicherung des Inhalts des Befehlszählers (Rückkehradresse) nach R31 vor dem Sprung zum Unterprogramm (Prozeduraufruf)

Befehlsformate	Operationen
`J name`	`PC ← name;` $(PC+4)-2^{25} \leq$ `name <` $(PC+4)+2^{25}$
`JR R3`	`PC ← R3`
`JAL name`	`R31 ← PC+4; PC ← name;` $(PC+4)-2^{25} \leq$ `name <` $(PC+4)+2^{25}$
`JALR R2`	`R31 ← PC+4; PC ← R2`
`BEQZ R4,name`	`if (R4==0) PC ← name;` $(PC+4)-2^{15} \leq$ `name <` $(PC+4)+2^{15}$
`BNEZ R4,name`	`if (R4!=0) PC ← name;` $(PC+4)-2^{15} \leq$ `name <` $(PC+4)+2^{15}$

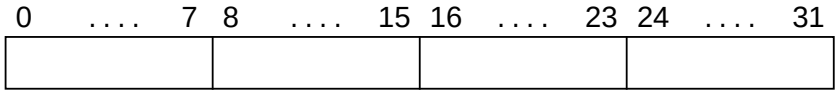
Sprungbefehle 1

Jump-Befehlsformat

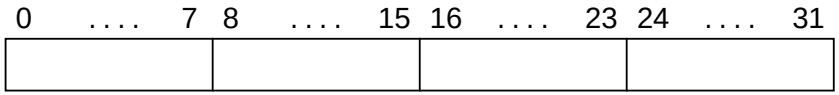


- Sprungbefehle (unbedingt):
  Jump (J), Jump and Link (JAL)
- Sprung erfolgt relativ zum Befehlszähler *(PC-relativ)*:

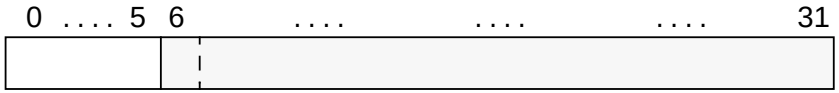
32 bit Befehlszähler *(Program Counter)*



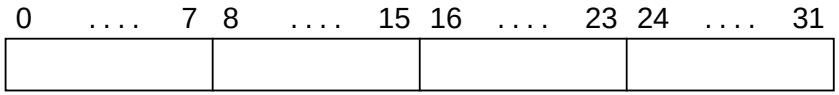
Fortschaltung des Befehlszählers $PC \leftarrow PC + 4$;
Link (nur bei JAL): $R31 \leftarrow PC$



+ 26 bit *Offset (Displacement)* mit Vorzeichenerweiterung

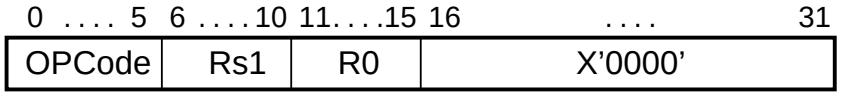


= Zieladresse im Befehlszähler $PC \leftarrow PC$ + Offset



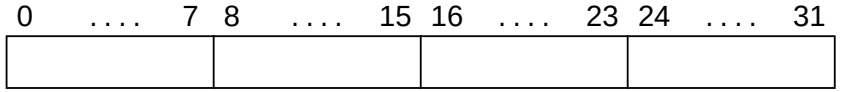
Sprungbefehle 2

Immediate-Befehlsformat

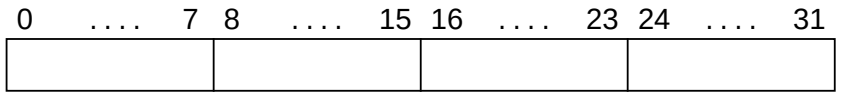


- Sprungbefehle (unbedingt):
  Jump Register (JR), Jump and Link Register (JALR)
- Inhalt von R0 = X'0000 0000'; Immediate = X'0000'
- Die Zieladresse befindet sich im Quellregister Rs1:

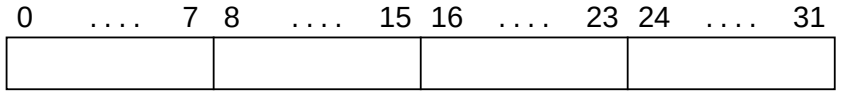
32 bit Befehlszähler *(Program Counter, PC)*



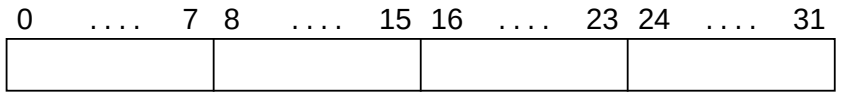
Fortschaltung des Befehlszählers $PC \leftarrow PC + 4$;
Link (nur bei JALR): $R31 \leftarrow PC$



32 bit Universalregister (Rs1)

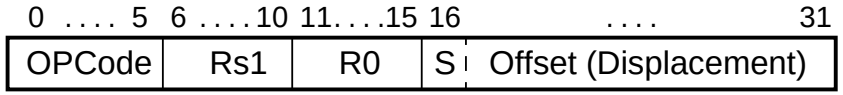


= Zieladresse im Befehlszähler $PC \leftarrow$ Rs1



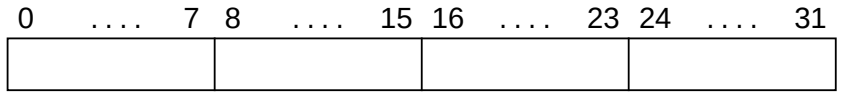
Verzweigungsbefehle

Immediate-Befehlsformat

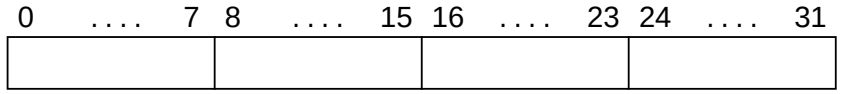


- Verzweigungsbefehle (bedingt):
  Branch on Zero (`BEQZ`); Branch on Not Zero (`BNEZ`)
- Das Quellregister (Rs1) wird getestet:
  bei `BEQZ` auf gleich Null; bei `BNEZ` auf ungleich Null
- Bedingung erfüllt: Zieladresse PC-relativ;
  Bedingung nicht erfüllt: Fortsetzung mit nächstem Befehl

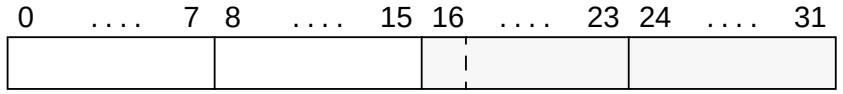
32 bit Befehlszähler *(Program Counter, PC)*



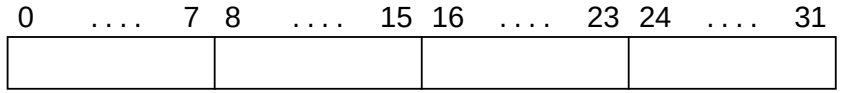
Fortschaltung des Befehlszählers $PC \leftarrow PC + 4$



+ 16 bit *Offset (Displacement)* mit Vorzeichenerweiterung



= Zieladresse im Befehlszähler $PC \leftarrow PC + \text{Offset}$

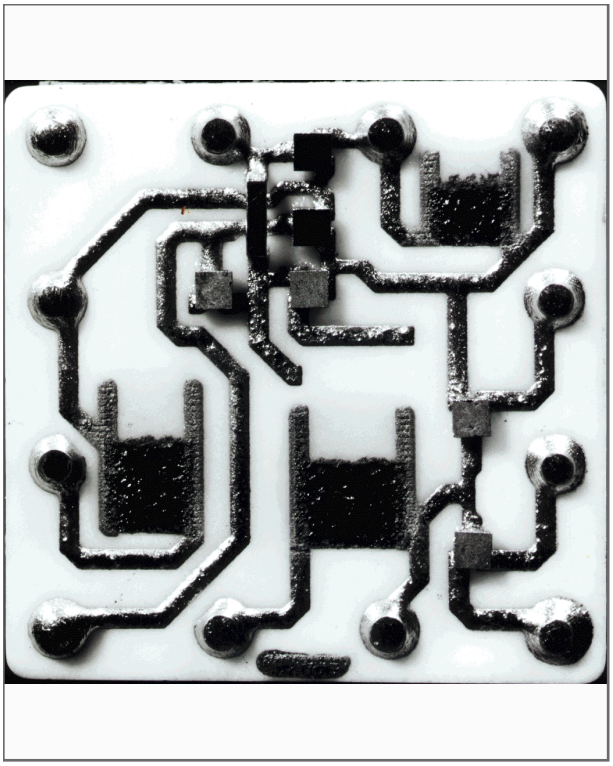


ti-I

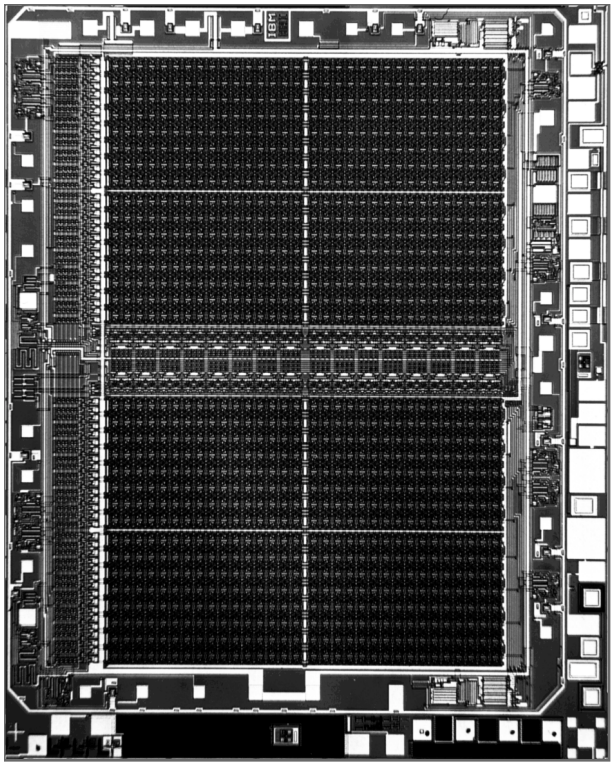
7 Hierarchie der Rechnersysteme

Prof. Dr.-Ing. U.G. Baitinger

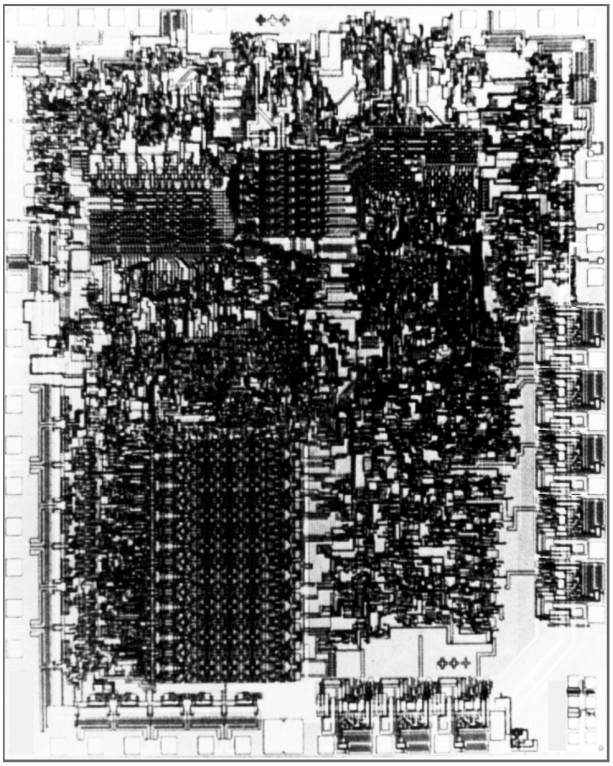
7.1 Die Entwicklung der Hardware



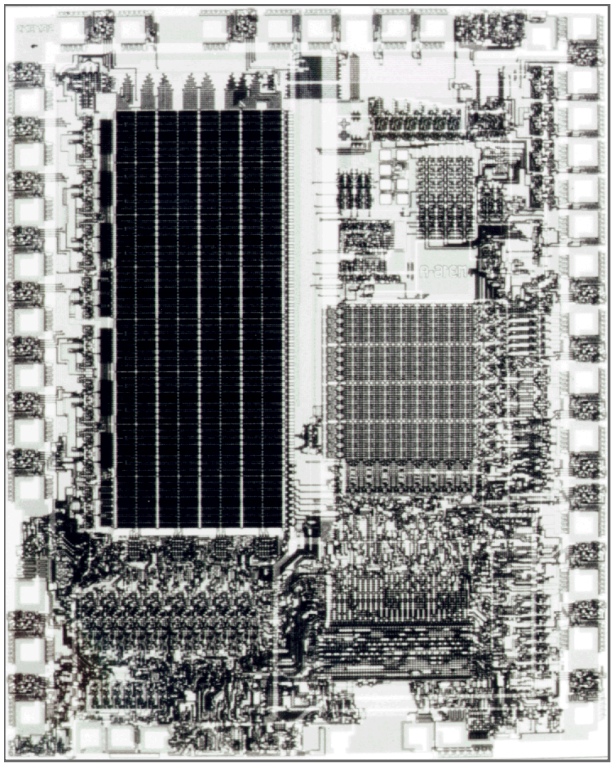
Logikmodul (1964)



Speicherchip (1972)



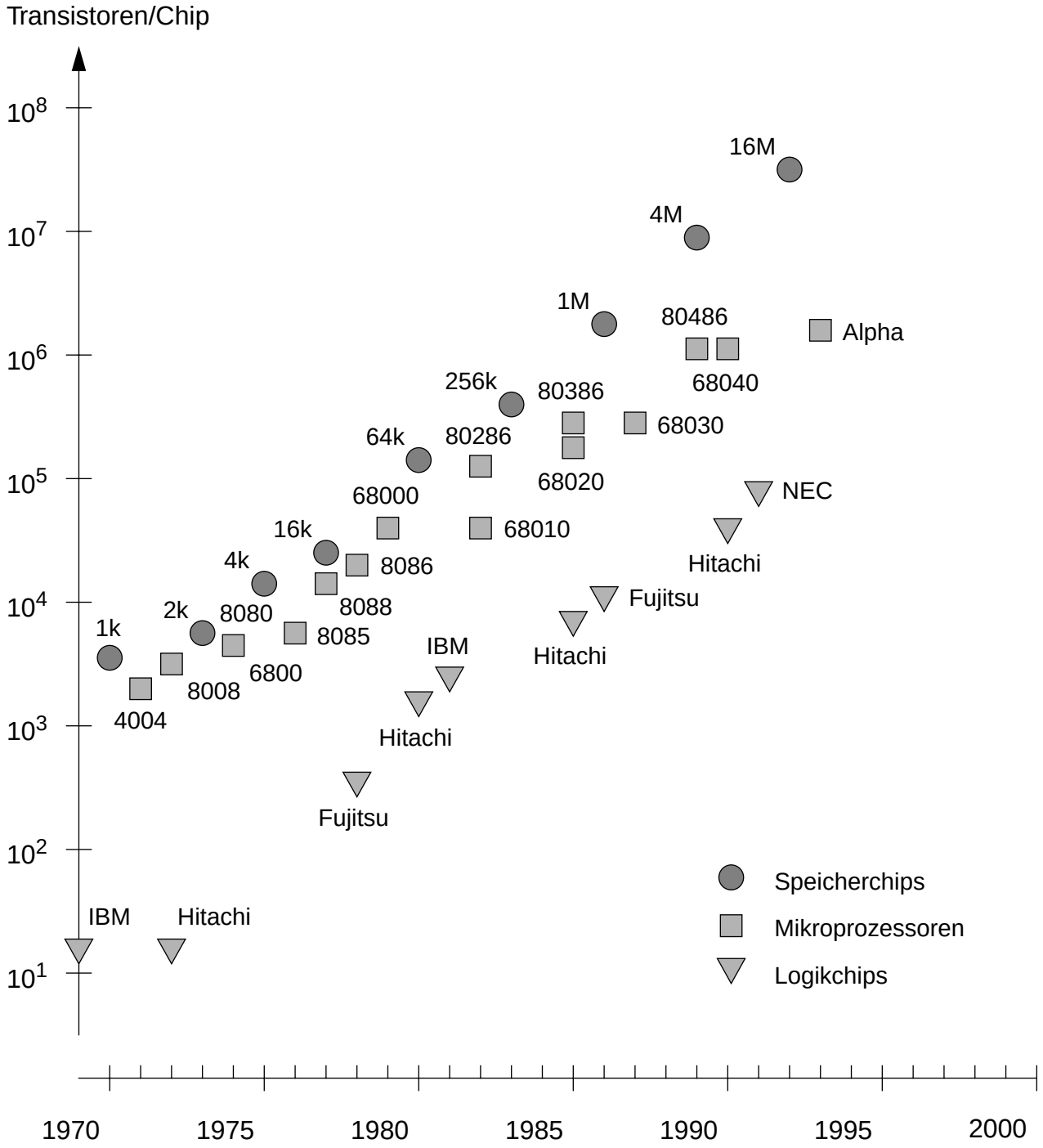
Mikroprozessor (1974)



Mikrocomputer (1984)

7.2 „Systemkomplexität“

Üblicherweise gilt die (relativ einfach zu ermittelnde) Anzahl der Transistoren pro Chip als die „Systemkomplexität“. Die zu jedem Zeitpunkt grundsätzlich höheren Werte für Speicherchips im Vergleich zu Mikroprozessoren und Logikchips zeigen aber, dass die regelmäßige Speicherstruktur höhere Transistordichten zulässt. Deshalb sollte zur Ermittlung einer „Systemkomplexität“ nicht nur die Anzahl der Transistoren auf einem Chip, sondern auch dessen Verdrahtungsstruktur mitberücksichtigt werden.



7.3 Hierarchischer Systementwurf

Systemkomplexität

Zur Beherrschung der Systemkomplexität ist das zu entwerfende Digitalsystem grundsätzlich aufteilbar:

- in ein **Rechenwerk**, das Datenpfade und einen Vorrat arithmetischer und logischer Funktionen bereitstellt, durch deren Kombination sich kompliziertere Funktionsabläufe ergeben,
- und ein digitales **Steuerwerk**, das als Automat mit einer endlichen Anzahl diskreter Zustände beschrieben werden kann, das solche Funktionsabläufe zeitlich steuert.

Entwurfskomplexität

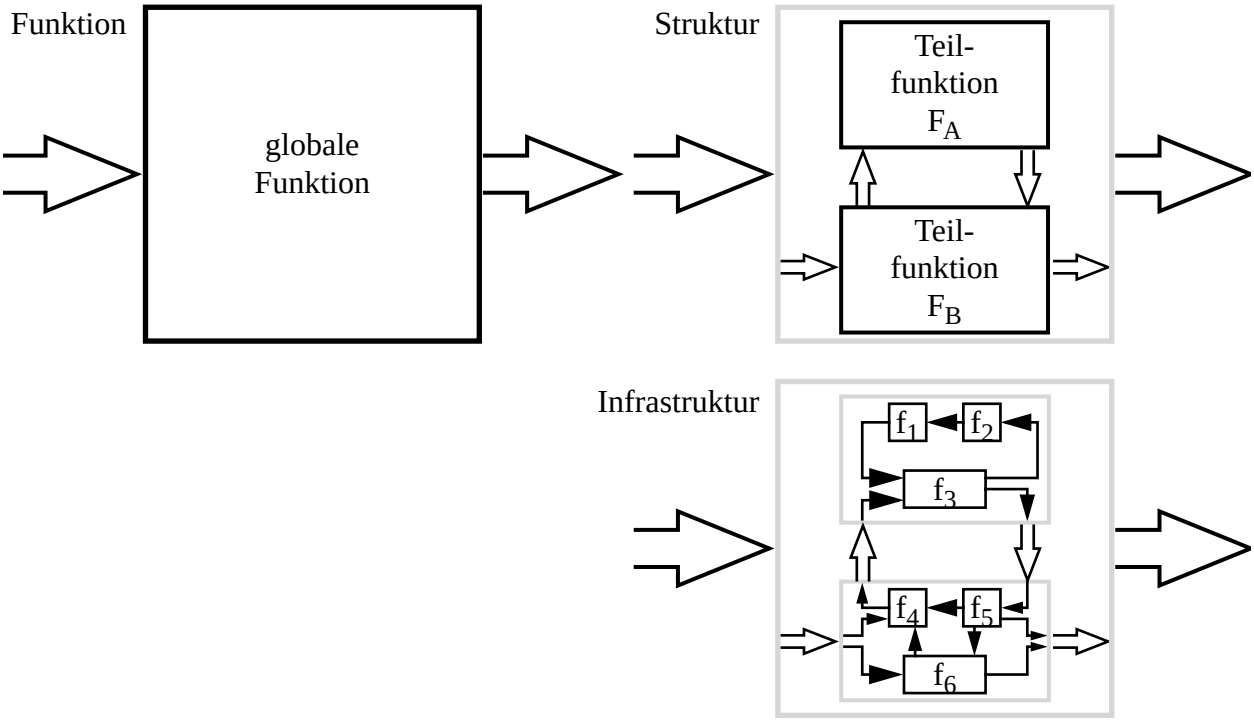
Zur Beherrschung der Entwurfskomplexität digitaler elektronischer Systeme unterscheidet man grob zwei Bereiche:

- den **logischen Entwurf**, der die Funktionen des Systems als schematische Schaltungsstruktur ohne Berücksichtigung geometrisch-physikalischer Randbedingungen implementiert,
- und den **physischen Entwurf**, der die logische Schaltungsstruktur in Form geometrischer Strukturen realisiert.

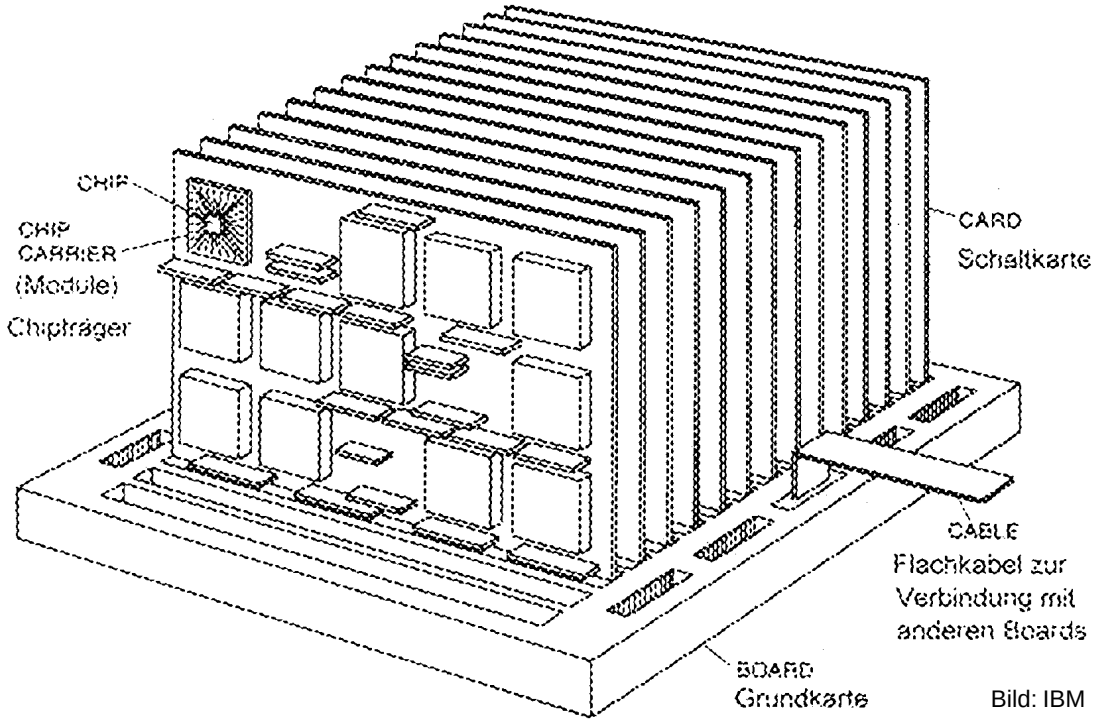
Zergliedernder Entwurfsstil ("top-down design"): Man beginnt auf einer Darstellungsebene hohen Abstraktionsgrades, d.h. mit einer globalen Funktionsbeschreibung des Gesamtsystems und verfeinert die Strukturen schrittweise bis auf die Ebene der Technologie.

Aufbauender Entwurfsstil ("bottom-up design"): Man beginnt auf einer Darstellungsebene niedrigen Abstraktionsgrades, d.h. mit detaillierten Strukturen, und faßt sie schrittweise zu größeren Funktionsblöcken zusammen, bis sich das Gesamtsystem ergibt.

7.4 Rekursive Verfeinerung

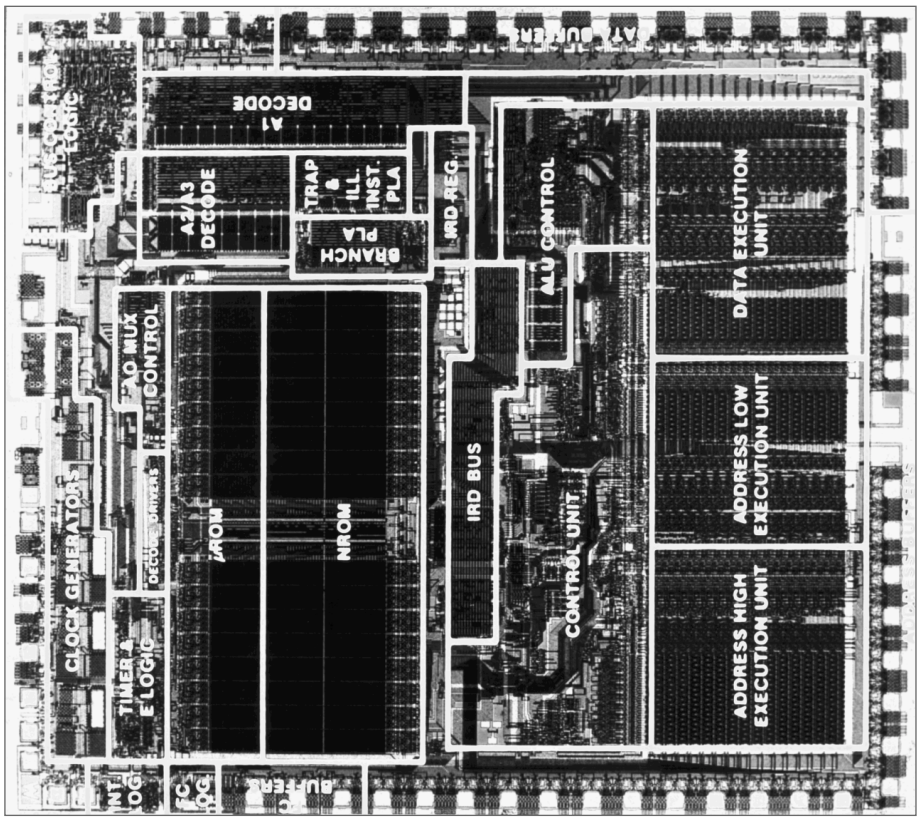


7.5 Hierarchie der Packungsebenen



Grundkarte mit aufgesteckten Schaltkarten

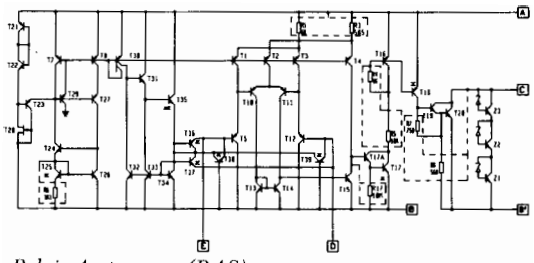
7.6 Hierarchischer Chip-Entwurf



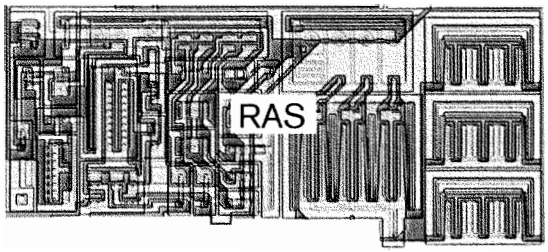
Hierarchisch gegliederter Mikroprozessor-Chip

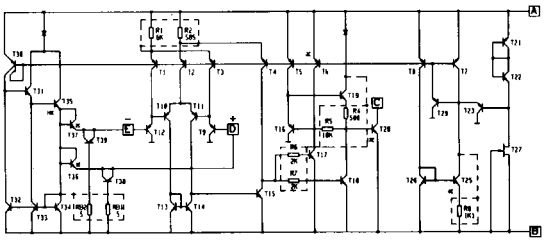
7.7 Zellbibliothek

Schaltkreis-Entwurf Layout-Entwurf

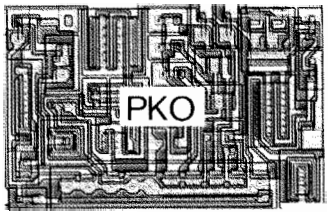


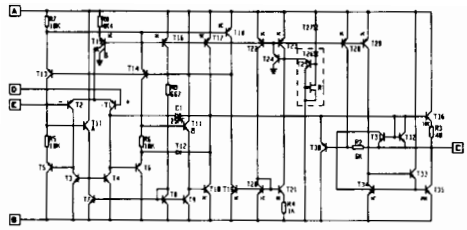
Relais-Ansteuerung (RAS)



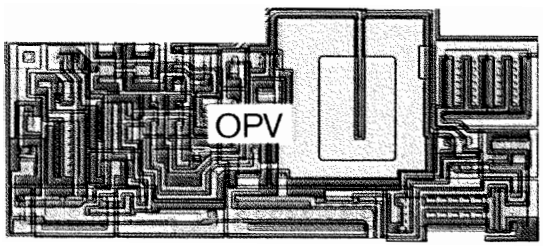


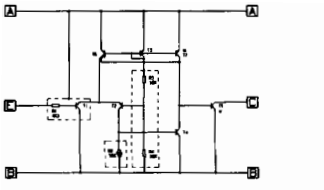
PNP-Komparator (PKO)





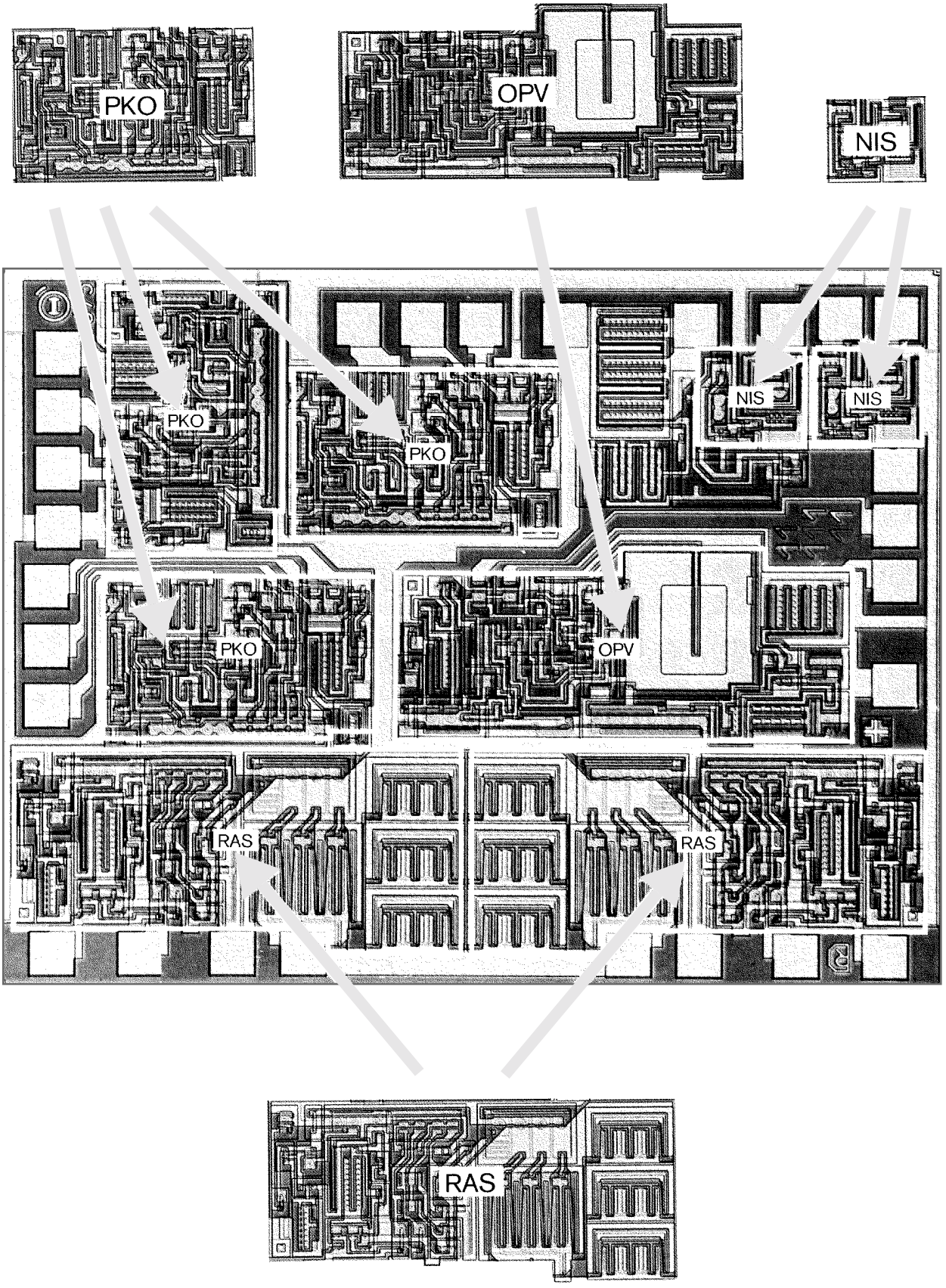
Operationsverstärker (OPV)





Nichtinvertierende Schaltstufe (NIS)

7.8 Platzierung & Verdrahtung



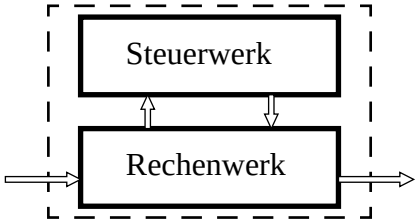
7.9 Hierarchie der Entwurfsdaten

Verhalten	logischer Entwurf →	Struktur	physischer Entwurf →	Geometrie

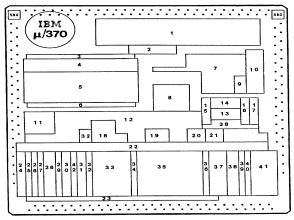
Rechner-Architektur

Beschreibung des Befehlssatzes und der Datenformate

Prozessor



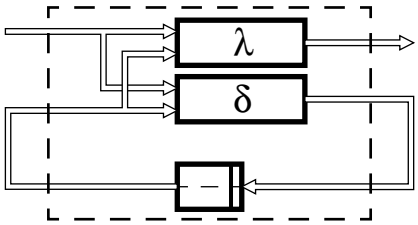
Floorplan



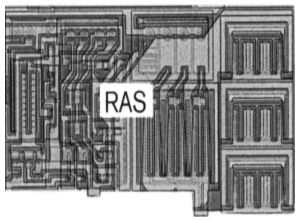
Automat

Beschreibung der Zustände und Zustandsübergänge
- durch Ablaufgraph
- Ablauftabelle
- (V)HDL-Text

Schaltwerk



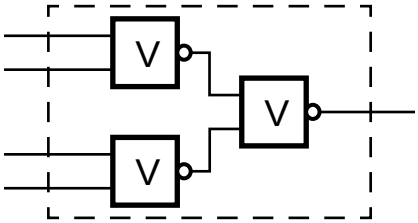
Makrozelle



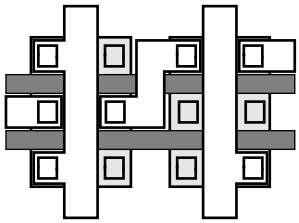
Zuordner

Beschreibung der Schaltfunktion
- durch KV-Diagramm
- boolesche Gleichung
- Wahrheitstabelle
- (V)HDL-Text

Schaltnetz



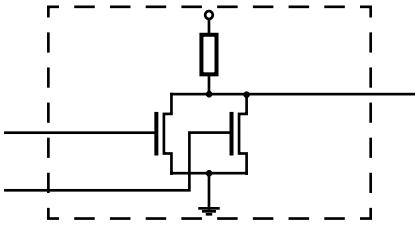
Zelle



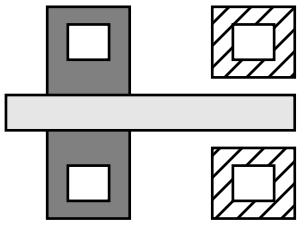
Schaltglied

Beschreibung der elementaren Schaltfunktion durch
- Wahrheitstabelle
- (V)HDL-Text

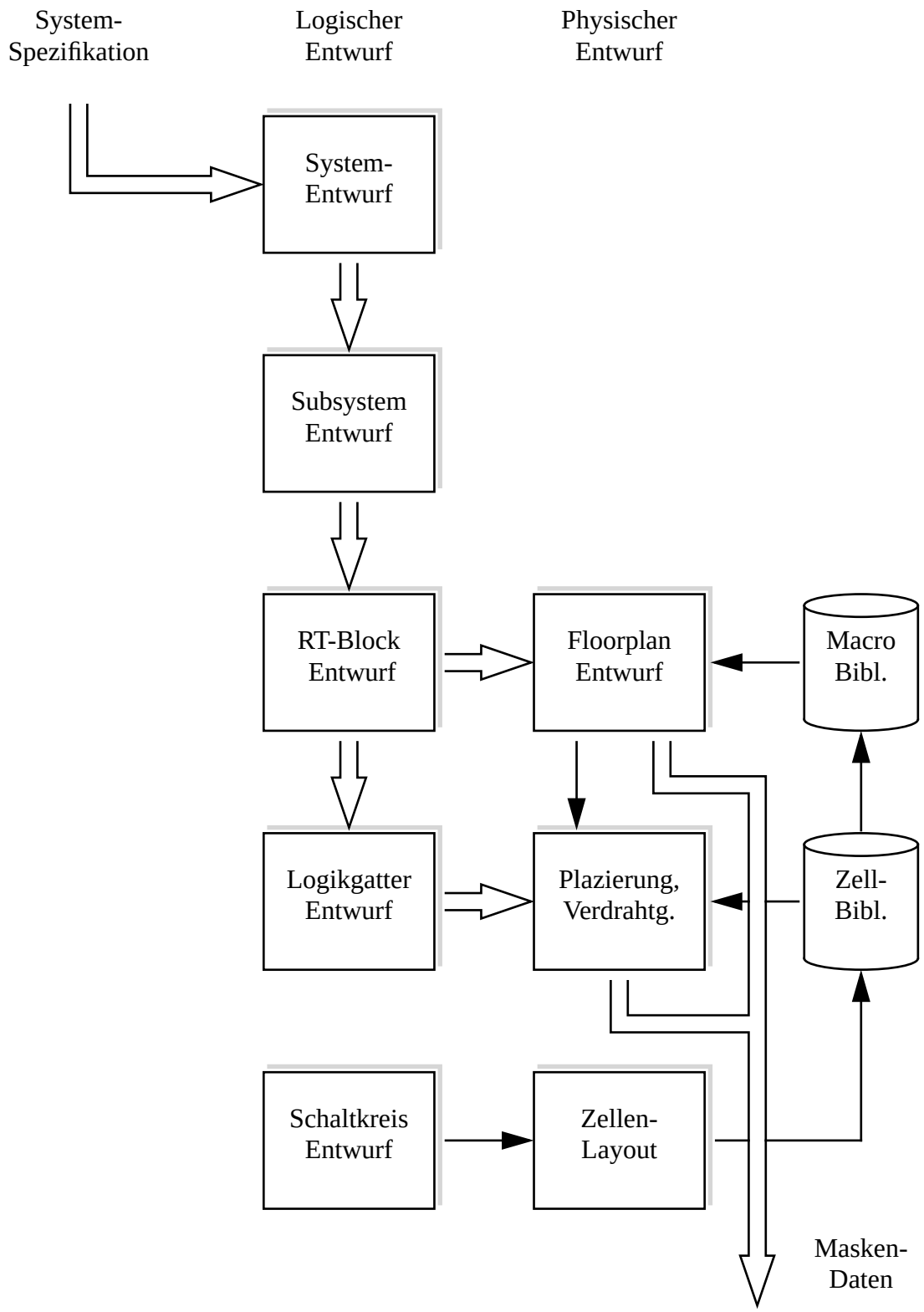
Schaltung



„Maske“

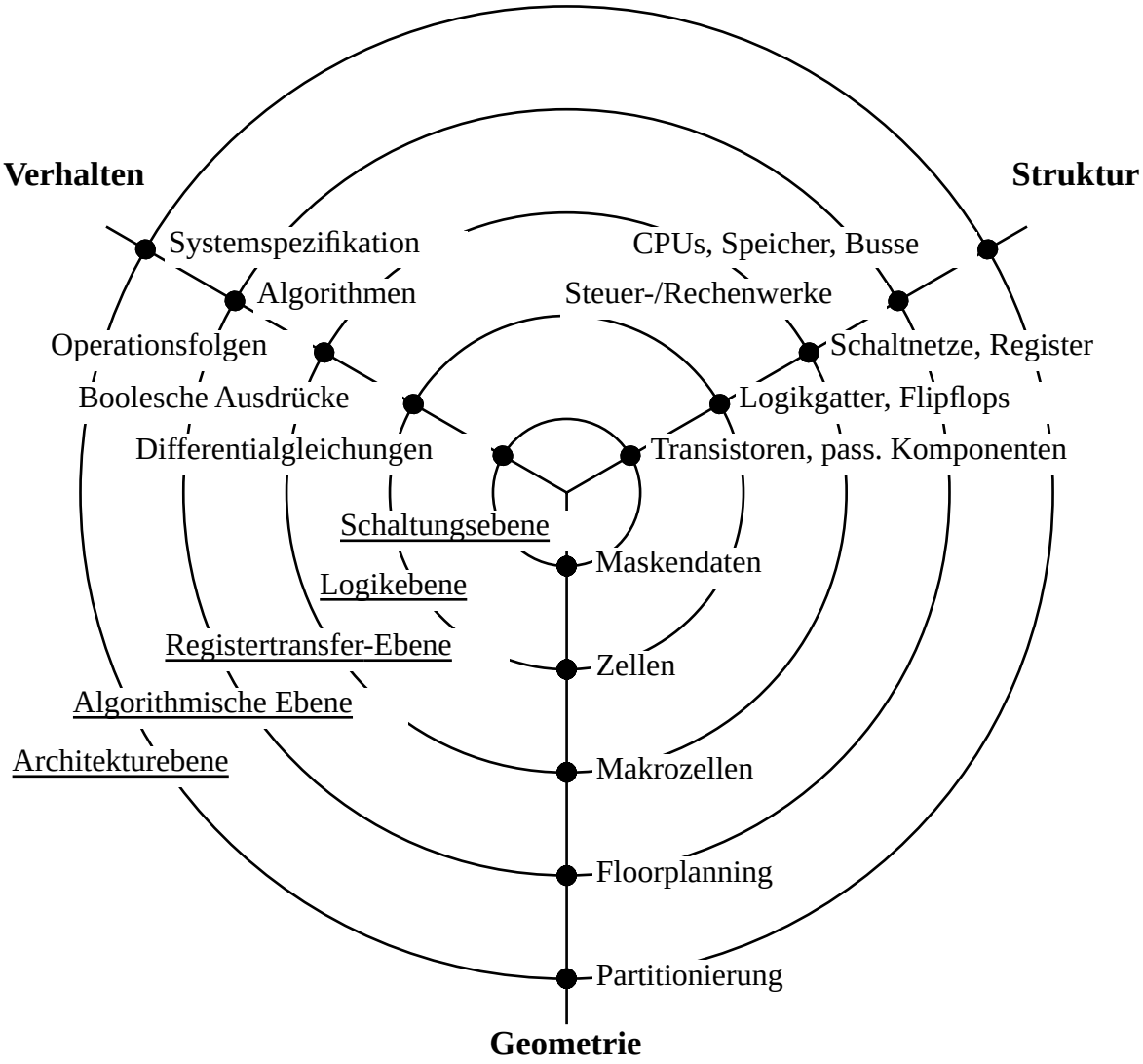


7.10 Hierarchie der Entwurfsschritte



7.11 Der Entwurfsraum

„Y-Diagramm“



7.12 Entwurfsmethodik

Der Entwurf schlägt die Brücke von einer abstrakten Spezifikation zum konkreten Produkt.

Entwurfsraum

- Die Koordinaten des „Y-Diagramms“ sind die drei unterschiedlichen *Sichten* eines zu entwerfenden elektronischen Systems.
- Die Kreise sind *Hierarchie-Ebenen*; nach außen nimmt die Abstraktion, nach innen die Detaillierung des Entwurfs zu.
- Die Schnittpunkte („Knoten“) symbolisieren *Entwurfsdaten* in unterschiedlichen Sichten und Abstraktionen.
- Die Verbindungslinien („Kanten“) symbolisieren *Entwurfsschritte*, d.h. Transformationen der Entwurfsdaten.

Entwurfsablauf

- Ein zergliedernder Entwurf beginnt mit der globalen *Systemspezifikation*, dem „Verhalten“ auf der abstraktesten Ebene. *Entwurfsziel* ist das Zentrum des Diagramms.
- Ein Entwurfsablauf ist ein *Pfad* durch den Entwurfsraum; denn die Komplexität des zu entwerfenden Systems ist nur in mehreren Entwurfsschritten zu bewältigen.

Entwurfsschritte

- Entwurfsschritte transformieren Entwurfsdaten: Sie ändern die Sicht auf das zu entwerfende System und/oder verfeinern seine Darstellung.

7.13 Entwurfswerkzeuge

Interaktive Graphik-Editoren

- Alle Entwurfsentscheidungen trifft der Entwerfer.
- Graphische Editoren für Schaltplaneingabe und Layout.
- Simulatoren und Entwurfsregelprüfer.
- Datenbank für die Entwurfsdaten.
- Die Ergebnisse aller Entwurfsschritte müssen durch Simulation validiert werden.

Synthese-Werkzeuge

- Grobe Granularität des Wissens: Das Entwurfswissen wird durch wenige komplexe Algorithmen erfaßt.
- Aus einer höheren Hardware-Beschreibungssprache wird automatisch eine spezielle Struktur synthetisiert.
- Algorithmen erzeugen regelmäßige Strukturen.
- Entwurfsstil gut angepaßt an PLA, PAL und Gate-Arrays.

Experten-Systeme

- Feine Granularität des Wissens: Die Entwurfserfahrung wird durch hunderte einfacher Entwurfsregeln erfaßt und in einer Wissensbasis gespeichert.
- Die Entwürfe sind unregelmäßig strukturiert, ähnlich zu Handentwürfen.
- Die Entwurfsregeln können an Änderungen der Technologie relativ einfach angepaßt werden.

7.14 Simulationsebenen

System-Simulation

- Elemente der Simulation sind größere Funktionseinheiten, wie Mikroprozessoren, Hauptspeicherbereiche und Ein/Ausgabesteuerungen, deren Zusammenwirken über Sammelleitungen („Bus“) *funktionell* modelliert wird.

Register-Transfer-Simulation

- Elemente der Simulation sind digitale Funktionseinheiten, d.h. Register und kombinatorische Blöcke, z.B. Addierer, deren Kommunikation über Busleitungen *funktionell* modelliert wird.
- Digitalsimulation mit binär codierten Worten („Bit Strings“), d.h. mit Vektoren aus binären Variablen und deren *diskreten* zeitlichen Veränderungen.

Logik-Simulation

- Elemente der Simulation sind logische Gatter und Flipflops; damit aufgebaute Netzwerke werden *strukturell* modelliert.
- Digitalsimulation mit zwei Signalpegeln bzw. binären Variablen („Bits“) mit den *diskreten* Werten 0, 1 und X = unbekannt.

Schaltkreis-Simulation

- Elemente der Simulation sind elektronische Komponenten (Transistoren, Dioden, Widerstände, Kondensatoren, Spulen); damit aufgebaute Netzwerke werden *strukturell* modelliert.
- Analogsimulation physikalischer Größen und Signale (Spannungen, Ströme, elektrische Leistung), die sich *kontinuierlich* veränderndern.

7.15 Simulation

Simulationsmodell

Informatorische Nachbildung des Verhaltens des entworfenen Systems auf einem Digitalrechner, um es mit den Entwurfsvorgaben („Spezifikation“) zu vergleichen.

- Digitale Elektronik: Das Simulationsmodell wird direkt aus dem Stromlaufplan bzw. der Netzliste abgeleitet.
- Mechanik, analoge Elektronik: Das Simulationsmodell wird als System physikalischer Gleichungen erstellt, die das Verhalten der entworfenen Vorrichtung beschreiben.

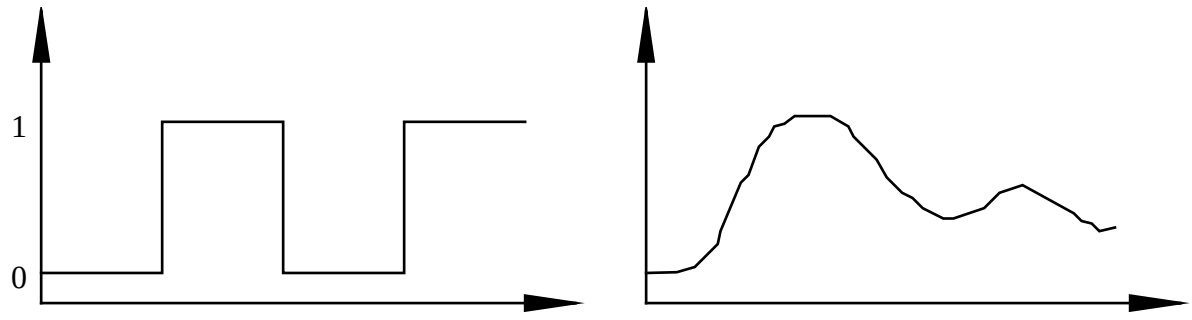
Stimuli

Ansteuerung des Simulationsmodells während des Simulationslaufs mit Eingabevariablen bzw. -signalen.

- Digitale Elektronik: Binäre Eingabe*variable* zur digitalen Simulation auf der Ebene logischer Gatter.
- Mechanik, analoge Elektronik: Zeitlich veränderliche Eingabe*signale* zur analogen Simulation, z.B. auf der Ebene elektronischer Schaltkreise.

Visualisierung

Graphische Darstellung der Ausgabevariablen bzw. -signale.



digitale Simulation analoge Simulation

7.16 Interaktiver Entwurfsschritt

mit Simulation